



# 蚂蚁金服金融科技产品手册

## 分布式链路跟踪

产品版本：V5.3.0  
文档版本：V20200402  
蚂蚁金服金融科技文档

**蚂蚁金服金融科技版权所有 © 2020 , 并保留一切权利。**

未经蚂蚁金服金融科技事先书面许可，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。

## **商标声明**



及其他蚂蚁金服金融科技服务相关的商标均为蚂蚁金服金融科技所有。

本文档涉及的第三方的注册商标，依法由权利人所有。

## **免责声明**

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。蚂蚁金服金融科技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，并在蚂蚁金服金融科技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蚂蚁金服金融科技授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

# 目录

---

<b>1 什么是分布式链路跟踪.....</b>	<b>1</b>
1.1 概述.....	1
1.2 产品架构.....	1
1.3 功能特性.....	3
1.4 应用场景.....	5
1.5 使用限制.....	7
1.6 基础术语.....	7
<b>2 快速入门.....</b>	<b>8</b>
<b>3 管控指南.....</b>	<b>10</b>
3.1 控制台总览.....	10
3.2 安装日志采集客户端.....	12
3.3 设置应用日志关联.....	14
3.4 查看应用拓扑关系.....	16
3.5 查看应用详情.....	19
3.6 搜索调用链路.....	22
3.7 添加自定义业务搜索项.....	23
3.8 查看链路详情.....	25
<b>4 SOFATracer.....</b>	<b>26</b>
4.1 什么是 SOFATracer.....	26
4.2 TraceId 和 SpanId 生成规则.....	27
4.3 开始使用 SOFATracer.....	29
4.4 Spring MVC 埋点接入.....	30
4.5 HttpClient 埋点接入.....	32
4.6 DataSource 埋点接入.....	36
4.7 RestTemplate 埋点接入.....	40
4.8 OkHttp 埋点接入.....	42
4.9 Dubbo 埋点接入.....	45
4.10 Spring Cloud OpenFeign 埋点接入.....	48
4.11 集成 SLF4J MDC 功能.....	52
4.12 异步处理.....	54
4.13 采样模式.....	55
4.14 上传数据到 Zipkin.....	58
4.15 Tracer 工具类.....	60
4.16 Tracer DRM 开关.....	62
4.17 Tracer 日志配置项.....	64
4.18 日志格式.....	66
<b>5 常见问题.....</b>	<b>86</b>

# 1 什么是分布式链路跟踪

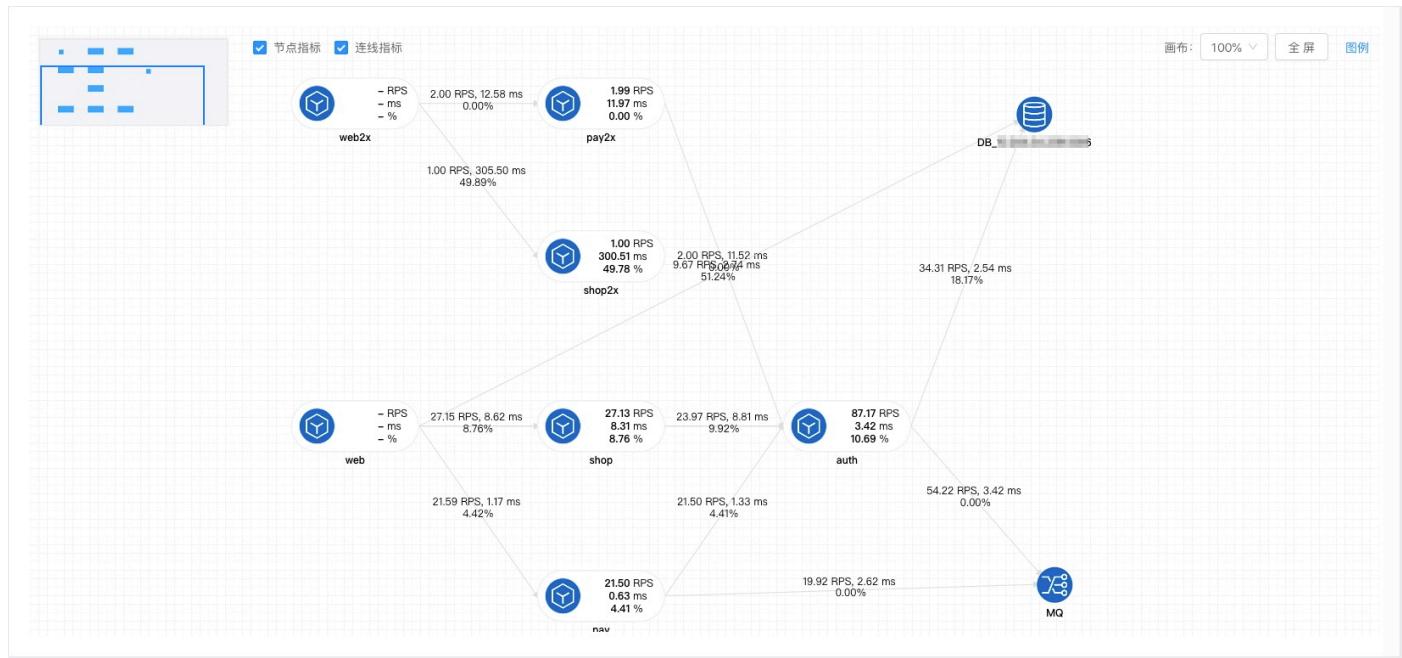
## 1.1 概述

分布式链路跟踪系统 ( Distributed System Tracing , 简称 DST ) 是面向分布式架构、微服务 ( Spring Cloud, SOFASStack、Service Mesh 等 ) 架构的云原生架构的应用可观察性 ( Observability ) 的金融级解决方案。

通过 DST , 运维人员、开发人员和架构师能看清楚复杂的大规模微服务架构下的应用及服务之间的**复杂调用关系、性能指标、出错信息与关联日志** , 从而实现故障根因分析、服务治理、应用开发调试、性能管理、性能调优、架构管控、故障定责等运维开发工作。

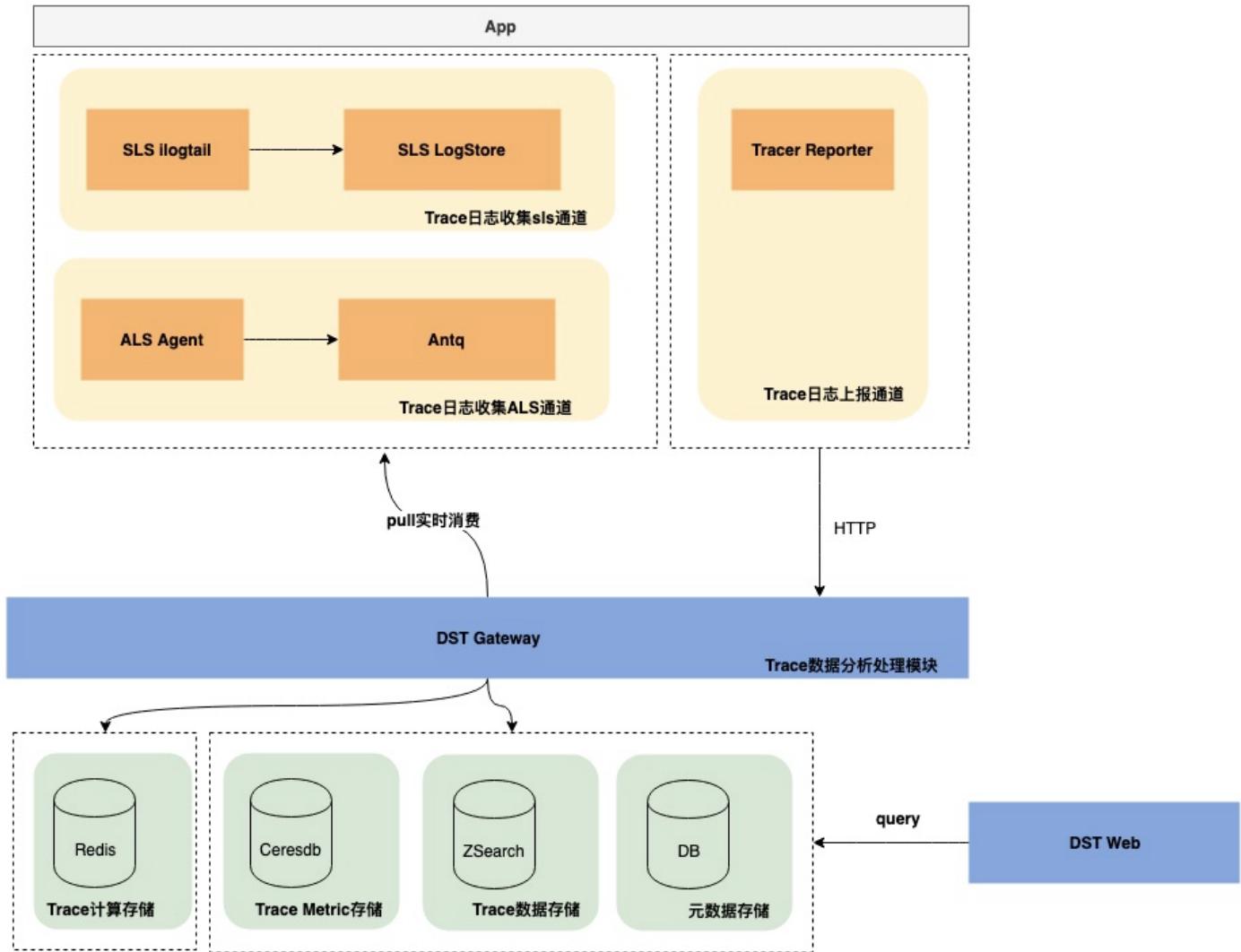
分布式链路跟踪具备以下特点 :

- **全链追踪** : 能够深入应用 , 服务 , 数据库 , 消息 , 捕获性能异常 , 识别出现故障的组件及服务。
- **易于使用** : 可以与金融科技平台上的应用进行无缝对接 , 用户的业务代码无需任何修改即可轻松接入 , 实现性能可视化与问题分析。
- **扩展性强** : 遵循业界 OpenTracing 标准 , 同时支持各类主流的编程框架与数据库等。



## 1.2 产品架构

分布式链路跟踪产品总体架构如下图所示 :



#### Trace 数据采集通道

目前支持 3 种数据的采集通道，分别是阿里云底座的 SLS 通道，Antstack 底座的 ALS 通道，以及应用主动上报的通道。前两种是基于 Trace 日志收集的方式，后一种是 Trace 数据直接上报。

#### Trace 数据计算分析存储

计算存储分析主要负责对收集到的 Trace 数据进行分析计算和将结果存储到对应的存储中。目前需要使用到的存储资源如下：

- ZSearch : ZSearch 主要负责存储 Trace 数据的明细数据。
- Ceresdb : 负责存储基于 Trace 数据计算出来的分析统计型的 Metric 数据。
- Redis : 主要负责计算过程中的一些中间计算结果以及一些 Cache 相关。
- DB ( MySQL 或 RDS ) : 负责存储产品使用和功能相关的元数据。

#### Trace 相关分析数据查询

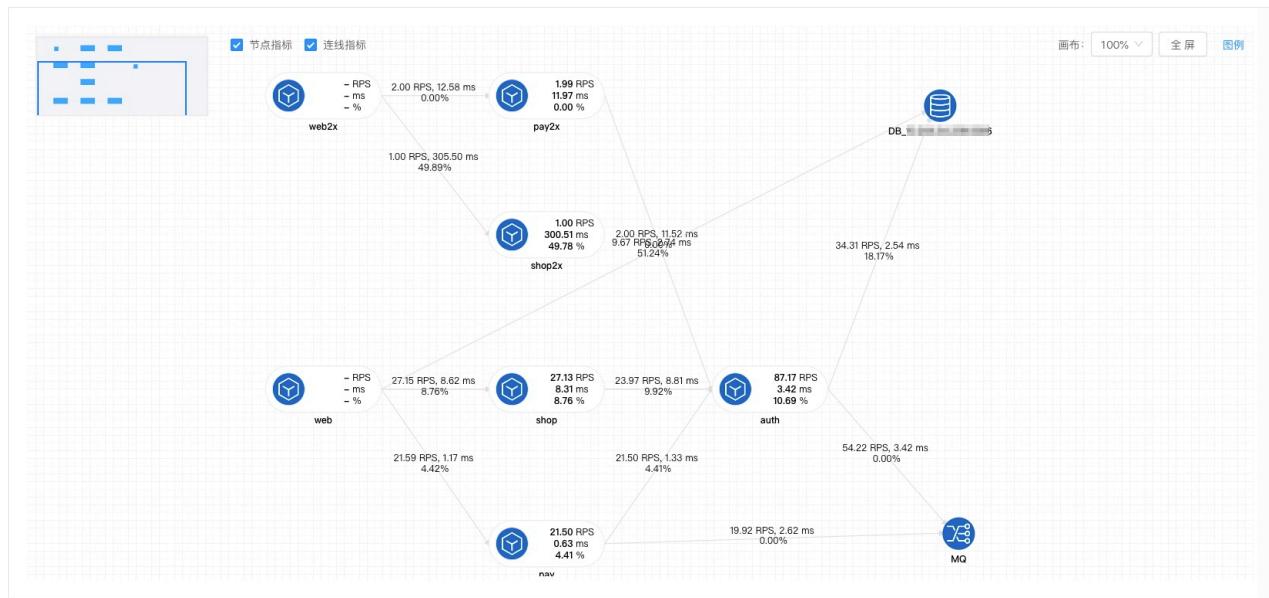
主要负责前端数据的查询，将 Trace 数据以及分析计算后的结果进行前端展示。

## 1.3 功能特性

### 应用拓扑发现

能持续地自动发现您整个应用层级中的依赖关系。实时以端到端的方式，展示应用架构

- 全链路信息展示**：展示应用程序及其关联内部、外部服务系统的响应时间、吞吐量和状态，同时显示了各个服务之间的相互影响。如果一项服务中断，您可以立即看到其他服务所受到的影响。
- 后端服务性能管理**：快速、持续地监控应用性能，让您在第一时间了解应用问题。
- 实时运行状态**：通过监控黄金指标（吞吐量，响应时间，错误率），可应用的性能监控描述，便于快速了解每个应用的运行状态。



### 分布式跨应用追踪

追踪每个交易的完整链路，按不同服务（应用）接口建立调用时间序列，收集链路上每个服务的性能数据，实现按服务追踪交易性能问题。

- 多维度链路查询**：根据 TraceId 按不同条件（错误、超时等）从多个维度检索调用链，分析链路信息，查询各场景下的调用链集合。
- 多视角链路展示**：提供链路图、链路详情、时序图、时间轴等，全方位可视化地展示系统性能。
- 业务日志关联查询**：自定义配置应用系统的业务日志，自动关联业务报错和摘要信息，快速定位问题和业务信息跟踪。

Trace Id:	请输入 traceId, 若无, 可使用条件查询	调用时间:	2019-12-09 01:47:50 ~ 2019-12-09 01:57:50
调用方式:	ALL	应用名:	结果: 失败 响应时长 (ms) :
中间件自定义搜索项: <a href="#">+添加</a>			
业务自定义搜索项: <a href="#">+添加</a>			
			<a href="#">查询</a> <a href="#">清空</a>

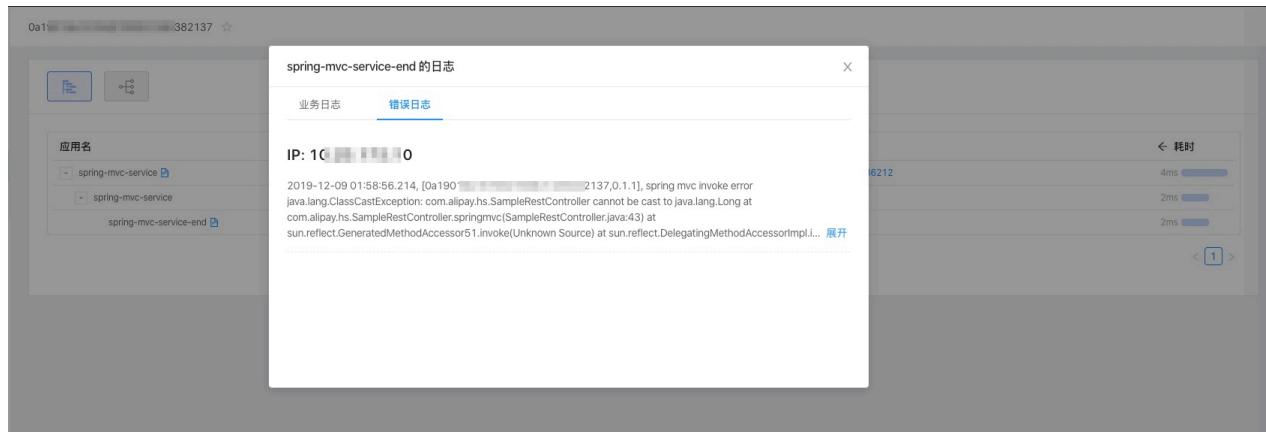
  

TraceId	调用时间	响应时长	调用结果	客户端	服务端	服务信息
00b...79668	2019-12-09 01:57	8 ms	● 失败	web(1... 84)	pay(1... 74)	PayServiceFacade#m1Pay
00b...79668	2019-12-09 01:57	21 ms	● 失败	web(1... 84)	auth(1... 13)	AuthFacade#m1CheckUserState
00l...79668	2019-12-09 01:57	1 ms	● 失败	web(1... 34)	pay(1... 4)	PayServiceFacade#m1Pay
00l...79668	2019-12-09 01:57	1 ms	● 失败	shop(1... 3)	auth(1... 5)	AuthFacade#m1CheckUserState
00ba...96668	2019-12-09 01:57	0 ms	● 失败	web(11... 4)	auth(1... 3)	AuthFacade#m1CheckUserState
00b...79668	2019-12-09 01:57	1 ms	● 失败	auth(1... 3)	DB(?)	UNKNOWN dtttest
00c...79668	2019-12-09 01:57	1 ms	● 失败	auth(1... 3)	DB(?)	UNKNOWN dtttest
00...179668	2019-12-09 01:57	6 ms	● 失败	web(11... 4)	auth(1... 8)	AuthFacade#m1CheckUserState
00b...79668	2019-12-09 01:57	8 ms	● 失败	web(1... 34)	auth(1... 3)	AuthFacade#m1CheckUserState

问题根源定位

主动发现性能问题，追踪最慢元素，查看请求参数，分析 SQL 语句执行、代码错误与异常，诊断后端瓶颈。

- **慢 SQL 分析**：查看执行缓慢的 SQL/NoSQL 语句。支持 MySQL、OceanBase、RDS、Redis、MongoDB、Memcached 等主流数据库。
  - **分析后端错误和异常**：错误信息功能可实时发现线上的错误异常，并查看错误的具体情况。



应用下钻

基于多种维度对应用进行深度剖面分析，如应用的基础性能分析、中间件层分析、异常问题分析、JVM分析等，帮助建立由底层到上层间的数据关联信息，从而深度分析分布式场景下的影响应用性能的问题根因。



## 1.4 应用场景

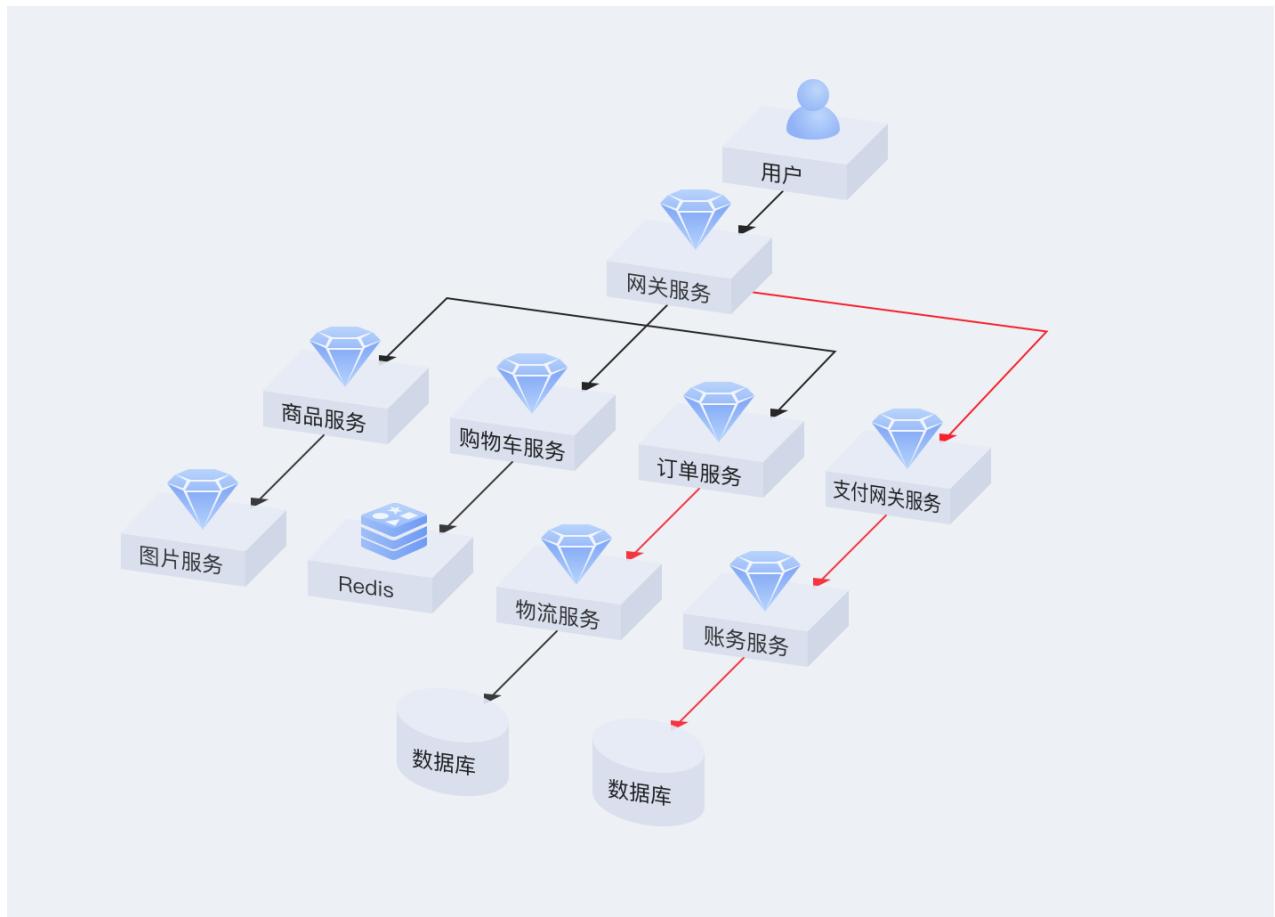
分布式框架在解决了传统的单块架构系统随着业务需求的快速变化而面临的挑战后，也为研发和运维增加了更大的复杂度和管理成本，基于分布式框架来进行架构改造的应用和业务通常会面临两大类挑战：

- 首先是在出现问题时，问题发现与定位如果能够快速精准，最大程度减少业务上带来的损失；
- 其次，应用的性能优化，服务能否降级，强依赖与关键路径在哪，如何做预算等在大促或者压测时能够分析链路中最早波动的点，沉淀压测资产等。

### 场景一：问题分析快速定位

在分布式场景下，服务调用错综复杂，问题分析与定位非常困难，分布式链路跟踪系统能迅速定位到有问题的服务，协助快速解决问题节点。

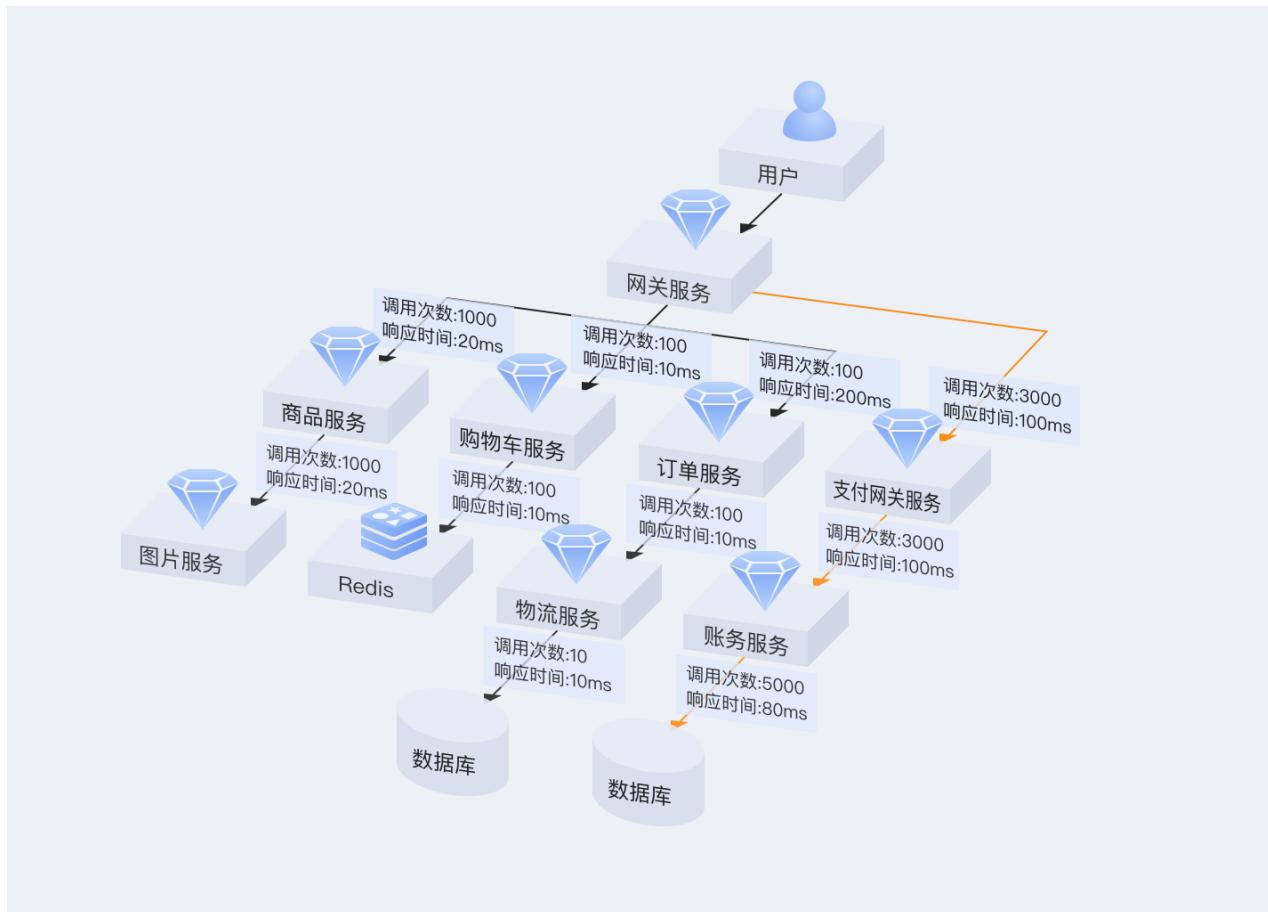
- 完整的应用调用拓扑关系**：自动发现该服务之前的调用以及对所有中间件的调用，绘制整个系统的完整调用拓扑关系。
- 快速定位不健康应用**：在调用关系拓扑中，对不健康应用进行显示标识，便于快速发现有问题应用并进行分析。
- 服务性能详情**：调用拓扑中的应用都可以单独进行下钻分析，可以从吞吐、错误率、响应时间等指标对应用性能进行详细分析。



## 场景二：应用性能优化

在调用关系拓扑中能对各个应用的调用次数以及耗时情况进行一个分析，找到负载较高以及负载较少的应用，对资源进行一个合理的利用。

- **调用链路聚合汇总**：所有的调用信息进行一个聚合汇总，对各个应用的调用情况以及响应情况进行分析。
- **关键路径**：快速发现整个系统调用拓扑中关键应用路径。
- **优化不合理调用**：及时发现某些不合理的调用并进行处理，如频繁进行数据库操作等。



## 1.5 使用限制

在使用分布式链路跟踪时，您需要注意以下几点使用限制。

限制项	限制范围	限制说明
JDK 语言	JDK 版本 1.8	JDK 版本 1.8
SOFABoot 版本	>= 3.2.1	如果使用主动上报功能的话需要 >=3.2.1，否则无此限制

## 1.6 基础术语

术语	说明
应用	泛指用于组成业务系统的应用，可以为单体应用也可以为基于分布式框架构成的微服务应用。
应用拓扑	拓扑是对应用间调用关系和依赖关系的可视化展示。
SOFATracer	SOFATracer 是一个用于分布式系统调用跟踪的组件，通过统一的 traceId 将调用链路中的各种网络调用情况以日志的方式记录下来，以达到透视化网络调用的目的。这些日志可用于故障的快速发现，服务治理等。
TraceID	TraceId 指的是 Tracer 中代表唯一一次请求的 ID，此 ID 一般由集群中第一个处理请求的系统产生。
错误率	应用在指定时间段内基于请求数的出错比例。

吞吐量	应用在指定时间段内处理请求的吞吐量走势。
RT分布图	应用在指定时间段内处理请求响应时间 ( Response Time ) 分布图。绿点代表请求响应成功；红点代表请求响应失败。
黄金指标	Four Golden Signals 是 Google 针对大量分布式监控的经验总结，4 个黄金指标可以在服务级别帮助衡量用户体验、服务中断、业务影响等层面的问题。主要关注以下四种类型的指标：吞吐量，响应时间，错误率以及饱和度。对于服务级别，通常使用前三个指标进行度量。

## 2 快速入门

本文介绍如何部署并使用分布式链路跟踪实时监控应用依赖、性能等信息并快速查询调用链路。

使用分布式链路跟踪前，您需要确保您的运行环境满足以下 前置条件，才可在产品控制台进行 应用分析，查看应用详情，链路搜索 以及 查看应用关联日志。

### 前置条件

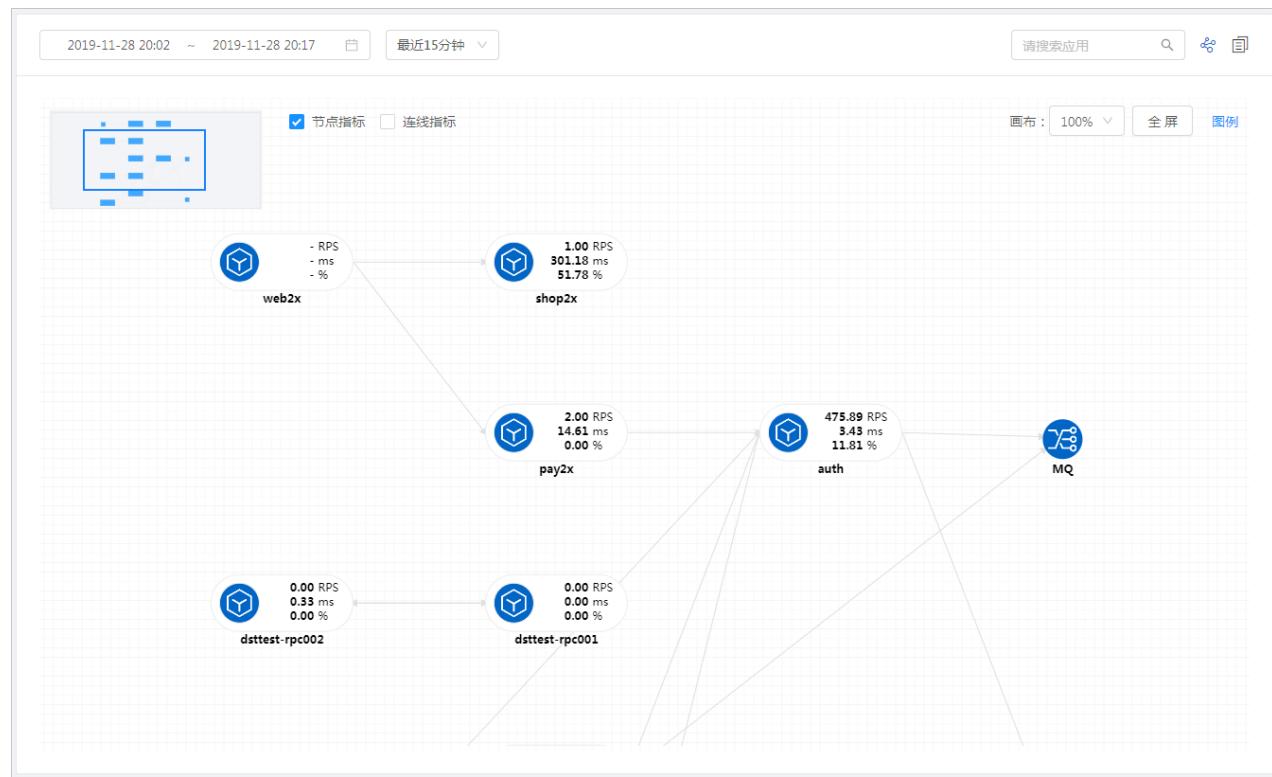
- 如果分布式链路跟踪服务使用 SOFABoot 技术栈：
  - 确保 SOFABoot 已升级至 3.2.1 或更高版本。
  - 确保 SOFABoot 工程的 pom.xml 文件引入了 Tracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

- 将应用发布在蚂蚁金服金融科技平台。发布流程，参见 [经典应用服务 > 应用部署](#)。
- 开启日志服务产品。分布式链路跟踪服务底层依赖于金融科技的日志服务产品收集应用日志信息，所以您必须在应用服务中开启日志服务产品。开通流程参见 [日志服务开通流程](#)。
- 安装日志采集客户端：可选择通过运维脚本批量安装或手动安装。
- 设置应用日志关联：为应用进行日志关联设置，以查看某一个链路环节所对应日志信息。

### 应用分析

在 [分布式链路跟踪控制台 > 应用分析](#) 页面，您可以查看到全局范围内运行的应用在指定时间段内的性能数据和拓扑结构。



### 查看应用详情

在应用拓扑图中，点击应用节点，出现 **查看应用详情** 按钮。点击该按钮，即可进入该应用的详情页，查看更多应用的监控数据，包括应用概览信息、服务调用信息、消息调用信息和数据库调用信息等。



### 链路搜索

在 **链路搜索** 页面，您可以通过多维自定义搜索，快速找到您想要查看的链路。关于自定义业务搜索，参见 [使用自定义业务搜索项](#)。

The screenshot shows a table with columns: 应用名 (Application Name), SpanId, IP, 调用类型 (Call Type), 状态 (Status), 服务信息 (Service Information), and 耗时 (Duration). The data includes:

应用名	SpanId	IP	调用类型	状态	服务信息	耗时
[-] web	0	127.0.0.1:34	DUMMY	成功	-	38ms
web	0.1	127.0.0.1:34	SOFARPC	成功	AuthFacade#m1CheckUserState	7ms
[-] auth	0.1	127.0.0.1:33	SOFARPC	成功	AuthFacade#m1CheckUserState	7ms
auth	0.1.1	127.0.0.1:33	MQ	成功	producer TP_DST_TEST1	5ms
MQ@TP_DST_TEST1	0.1.1	-	MQ	成功	producer TP_DST_TEST1	5ms

## 查看应用关联日志

在链路详情页，点击应用名旁边的日志按钮，可查看该应用关联的应用日志，如下图所示。

The screenshot shows a modal window titled "spring-mvc-service 的日志". It has tabs for "业务日志" (Business Log) and "错误日志" (Error Log), with "业务日志" selected. A dropdown menu shows "biz-log". The log content is as follows:

IP: 127.0.0.1:342

```

2019-11-29 18:34:31.348, [Oaa0...3962.0], indexSelf is running11.

2019-11-29 18:34:31.348, [Oaa0...3962.0], indexSelf is running10.

2019-11-29 18:34:31.348, [Oaa0...3962.0], indexSelf is invoked9.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is running8.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is running7.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is invoked6.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is running5.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is running4.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is invoked3.

2019-11-29 18:34:31.348, [Oaa02...3962.0], indexSelf is running2.

```

Pagination controls at the bottom: < 1 2 >

## 3 管控指南

### 3.1 控制台总览

您可以在分布式链路跟踪控制台 **总览** 页面查看当前环境下所有应用的监控统计信息。

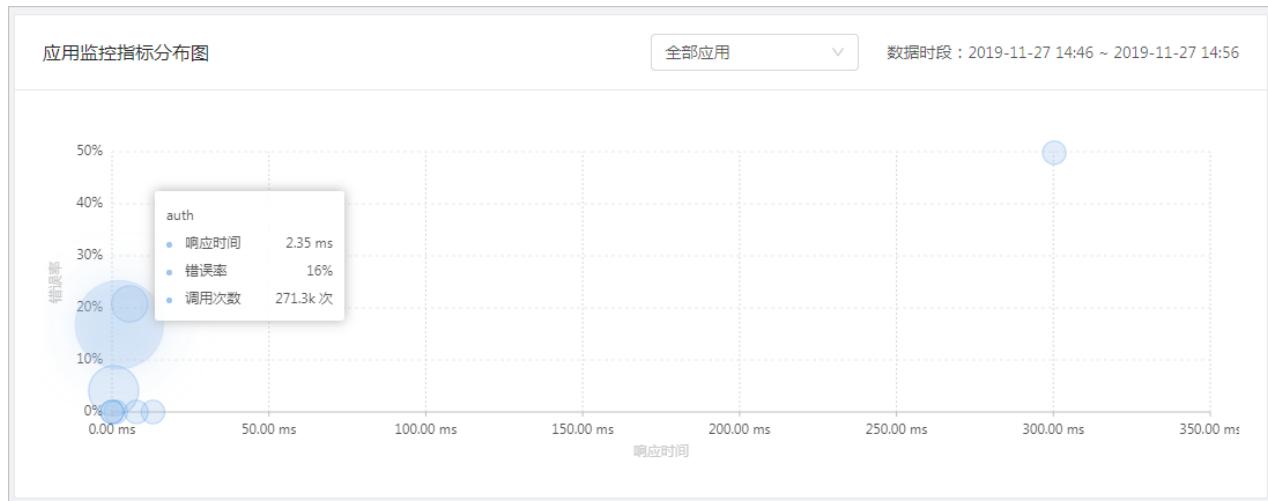
- 应用监控指标分布图
- 应用监控指标 TOP5 排行
- 收藏的应用与链路



## 应用监控指标分布图

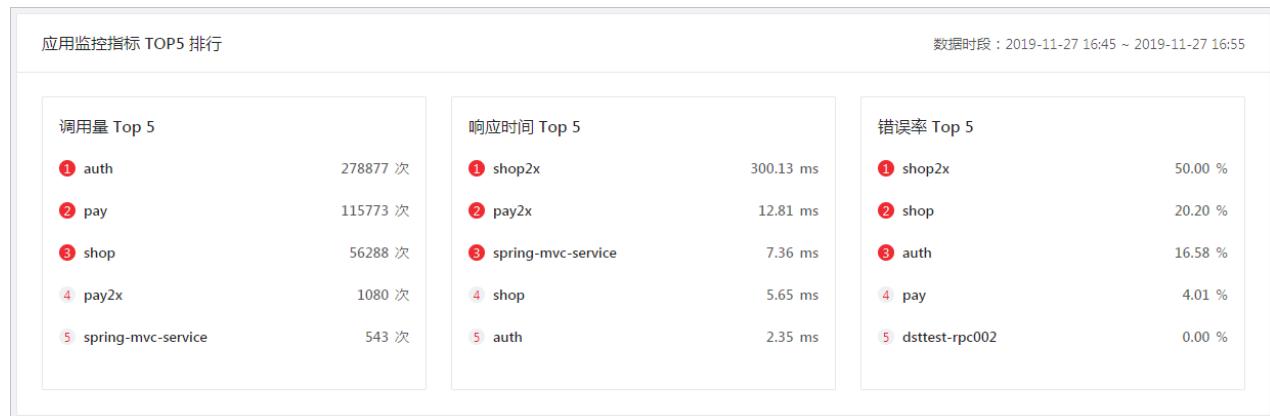
在 **应用监控指标分布图** 区域，您可以查看到在统计时间段内运行的所有应用的总体监控指标数据。如下图所示，一个圈代表一个应用，X 轴对应该应用的响应时间，Y 轴是该应用的错误率，而圈的大小则代表其请求量。鼠标悬浮于某一个应用，即可查看该应用的具体监控指标的数据。

点击 **全部应用** 下拉菜单，即可切换筛选查看 **请求量 Top10 应用**、**错误率 Top10 应用** 和 **响应时长 Top10 应用**。



## 应用监控指标 TOP5 排行

**应用监控指标 TOP5 排行** 模块实现了在默认统计时间段内全局范围内调用量、响应时间和错误率最高的分别 5 个应用及其具体的监控指标对应数据。点击应用名，即可跳转至该应用的 [应用详情](#) 页面。应用详细信息，可参见 [查看应用详情](#)。



## 收藏的应用与链路

此区域显示了收藏的所有应用和链路列表。

收藏的应用		收藏的链路		
应用名称	请求量 RPS	响应时间 ms	错误率 %	操作
dst-web	0.04	19.57	0.00	取消收藏
				< 1 >

### 收藏的应用列表

列出已收藏的应用，并显示相关性能信息，包括请求量、响应时间及错误率。点击应用名，会直接跳转至应用详情页面。详见 [查看应用详情](#)。

### 收藏的链路列表

列出已收藏的链路，并显示链路的相关信息，包括链路描述、操作人、收藏时间。点击链路名称，会直接跳转至该链路的 [链路详情](#) 页面。有关链路详情的信息，可参见 [查看链路详情](#)。

## 3.2 安装日志采集客户端

日志采集客户端 Logtail 可通过运维脚本批量安装或手动安装。

### 批量安装（推荐）

1. 创建指定模板。在 [金融分布式架构](#) 控制台页面，选择 [运维管理](#) > [经典应用服务](#) > [日常运维](#) > [指令模板](#)，在右侧窗口中点击 [创建模板](#)。在 [新建指令模板](#) 页面中，填入相应的环境信息。（如为 Linux 系统，脚本内容可以参考 [dst\\_sls\\_install.sh](#) 文件。）

下图是为 Linux 系统配置指令模板的示例：

描述: 上海金区 sls agent 安装

服务器账号: root

命令类型: 输入指令 [录入手本]

\* 指定文件路径: /usr/bin/customized\_cmd\_sls\_shfin.sh

\* 脚本内容:

```
#!/bin/sh

# instanceId should be changed in different workspace;
# region is sh/shfin/hz/hzfin/sgp;

instanceId="000001";
region="shfin";

# check param
if [ ! -n "$instanceId" ];then
    echo "date +%Y-%m-%d_%H:%M:%S" [DST] Invalid param instanceId.">>$dstErrorLog;
    exit 1;
elif [ ! -n "$region" -o ! ( $region = "sh" -o $region = "hz" -o $region = "hzfin" -o $region = "sgp" ) ];then
    echo "date +%Y-%m-%d_%H:%M:%S" [DST] Invalid param region, should be sh,shfin,hz,hzfin.">>$dstErrorLog;
    exit 1;
fi
```

**说明：**在不同 workspace 中执行脚本需要修改相应的 instanceId 和 region 参数：

- instanceId 为机器组标识，可以在 [分布式链路跟踪 > 设置](#) 页面查看。
- region 为域环境标志，如华东 2（上海），即 cn-shanghai，其值可参考 地域和可用区。

2. 执行运维脚本。在 [金融分布式架构](#) 控制台中选择 [经典应用服务 > 日常运维 > 服务器运维](#)，在右侧窗口中点击 **创建**。在 [创建服务器运维单](#) 页面中，选择刚才创建的模板指令和对应的机器列表，再勾选 **创建完成后自动执行**，点击 **创建**。

**说明：**建议先选择几台服务器验证环境是否正常，然后在整个环境中执行。

## 手动安装

1. 检查租户环境 ECS 的日志采集客户端（Logtail）是否已安装。

- 在 Linux 系统中，运行以下命令查看客户端状态：

```
sudo /etc/init.d/ilogtaild status
```

- 在 Windows 系统中，在 **服务** 中查看客户端运行状态：[控制面板 > 管理工具 > 服务](#)，在列表中查看 LogtailDaemon 和 LogtailWorker 两个 Windows 服务的运行状态。如未安装，可参阅文档：[如何在 Linux 系统或 Windows 系统中安装 Logtail](#)。

2. 添加机器组标识。

- 在 Linux 系统中，添加文件 /etc/ilogtail/user\_defined\_id，文件内容是实例标识（该值可在 [SOFA 应用中心](#) 首页查看），如：在 Linux 系统中，确保文件 /etc/ilogtail/user\_defined\_id 存在，并且确保文件内容需要有一行：机器组标识（可前往 [分布式链路跟踪 > 设置](#) 页面

获取该标识。 )。

日志配置

基本信息

机器组标识: NY...\_test

服务账号 AccessKey: ID / LTAIAF... GLNF Select / \*\*\*\*\* 验证 Access Key

应用列表

添加应用

> shop 日志采集: 2 创建人: dst

- 在 Windows 系统中，打开文件 user\_defined\_id（默认路径为 C:\LogtailData\user\_defined\_id），将其文件内容修改为 机器组标识。

### 3. 在 ECS 服务器上配置用户 ID 标识文件，授权分布式链路跟踪系统采集日志。

- 在 Linux 系统中，创建账号 ID 同名文件夹到目录 /etc/ilogtail/users 以配置用户标识，例如：
- 上海非金区 - 华东 2 ( 上海 ) :

```
sudo touch /etc/ilogtail/users/1665977623349188
```

- 在 Windows 系统中，创建账号 ID 同名文件夹到目录 C:\LogtailData\users 以配置用户标识，例如：

上海非金区 - 华东 2 ( 上海 ) :

```
C:\LogtailData\users\1665977623349188
```

## 3.3 设置应用日志关联

在分布式链路跟踪中，通过为应用进行日志关联设置，您可以在单链路显示时查看某一个链路环节所对应的应用日志信息。

### 日志关联原理

- 对应用日志输出的配置进行修改，更改日志输出 Pattern，在应用日志中，输出 TraceId 及 SpanId。
- 在云端应用中，设置需要采应用日志的应用名称，并配置日志名，及日志路径，进行日志采集。
- 在链路上点击应用节点，应用将会从日志中，根据应用名，通过 TraceId 及 SpanId 查找并显示对应的应用日志。

### 操作步骤

#### 配置本地应用日志打印

要实现链路与应用日志关联，本地应用日志打印时必须正确输出 TraceId 及 SpanId。分布式链路跟踪的 Tracer 集成了 SLF4J MDC 功能，所以您只需简单修改日志配置文件中的日志输出 Pattern，即可实现日志输出 Tracer 上下文 TraceId 以及 SpanId。

1. 确保 SOFABoot 应用引入了 Tracer 依赖。

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

2. 在 PatternLayout 中增加 %X{SOFA-TraceId} 和 %X{SOFA-SpanId} 配置。

```
<PatternLayout>
<pattern>%d %-5p %-32t [%X{SOFA-TraceId},%X{SOFA-SpanId}] - %m%n</pattern>
</PatternLayout>
```

完成以上配置后，实际输出的应用日志如下示例：

```
2019-12-10 23:59:50.214, [0a19018e157599359021261502137,0.1.1], spring mvc is invoked.
```

#### 云端添加应用日志

1. 进入分布式链路跟踪控制台页面，在左侧导航栏选择 **设置**。
2. 如您是第一次使用日志关联功能，您需要前往 RAM 控制台，创建并获取具有 AliyunLogFullAccess 权限策略的 AccessKey 及 AccessKeySecret。详细操作步骤，可参见 [创建 AccessKey](#)。
3. 获取 AK、SK 后，即可返回 [分布式链路跟踪 > 设置](#)，输入相应的 Access Key 与 Access Secret。

**请先验证 Access Key 信息**

请填入一个具有 AliyunLogFullAccess 权限策略的 Access Key/Secret

Access Key:	请输入Access Key
Access Secret:	请输入Access Secret

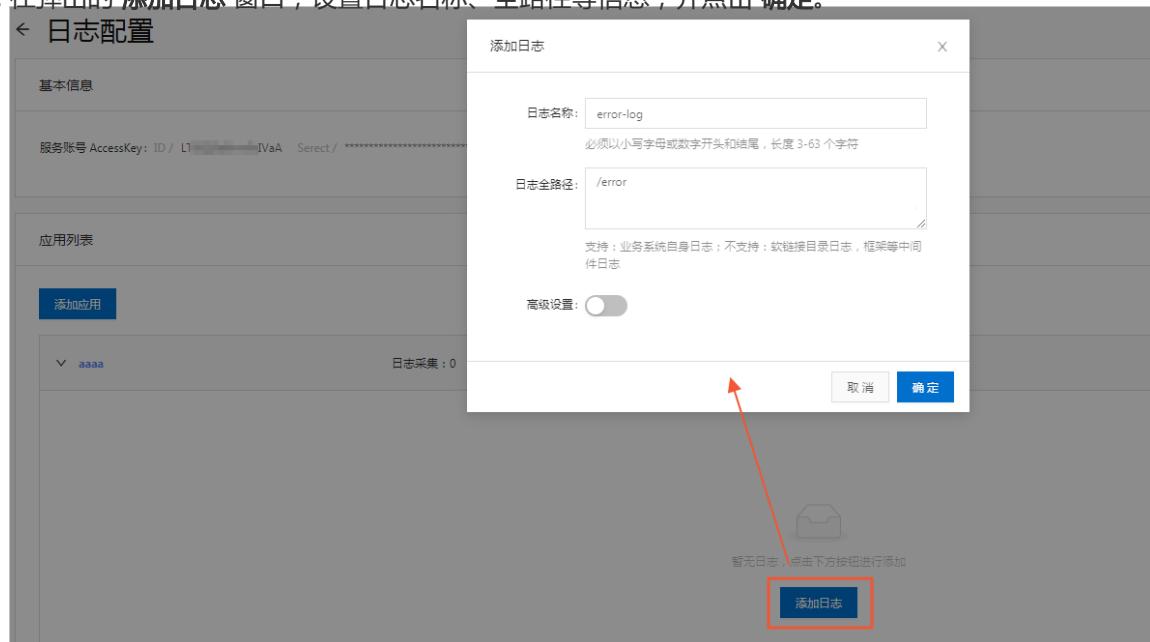
**验证**

4. 在 [日志配置](#) 页，点击 **添加应用**，在弹出的窗口中填入正确的应用名称后点击 **确定**。

**说明**：应用名必须与系统打印日志中的 appName 一致，否则无法获取应用数据。

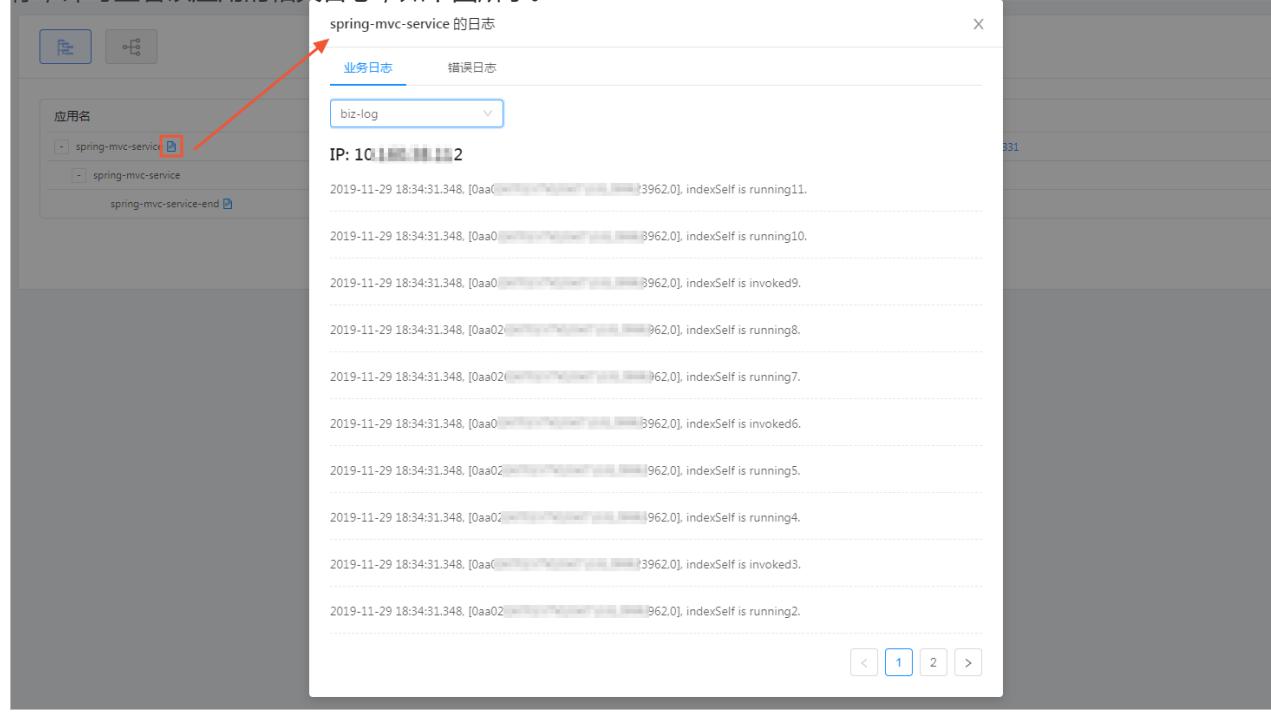
5. 应用添加成功后，点击 **添加日志**。

6. 在弹出的 **添加日志** 窗口，设置日志名称、全路径等信息，并点击 **确定**。



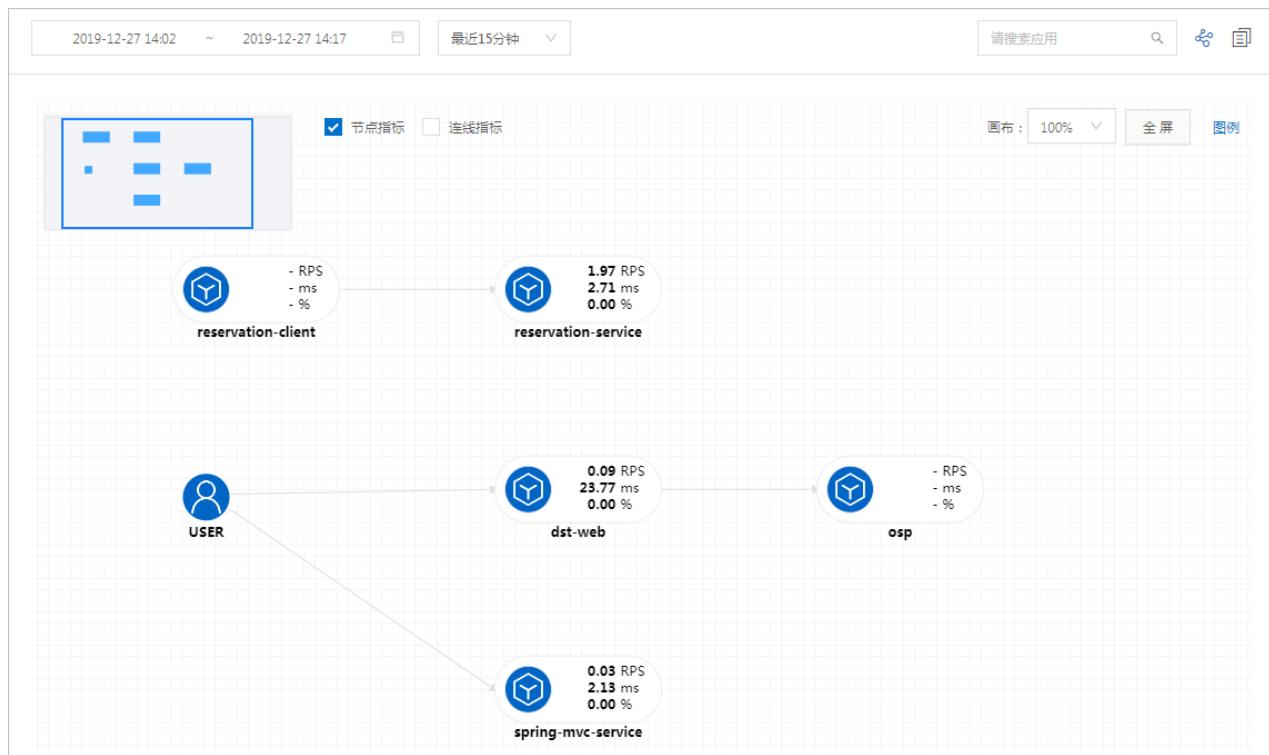
#### 查看应用关联日志

您可以前往链路搜索页面，找到该应用的相关链路，查看链路详情。在链路详情页，点击应用名右侧的日志图标，即可查看该应用的相关日志，如下图所示。



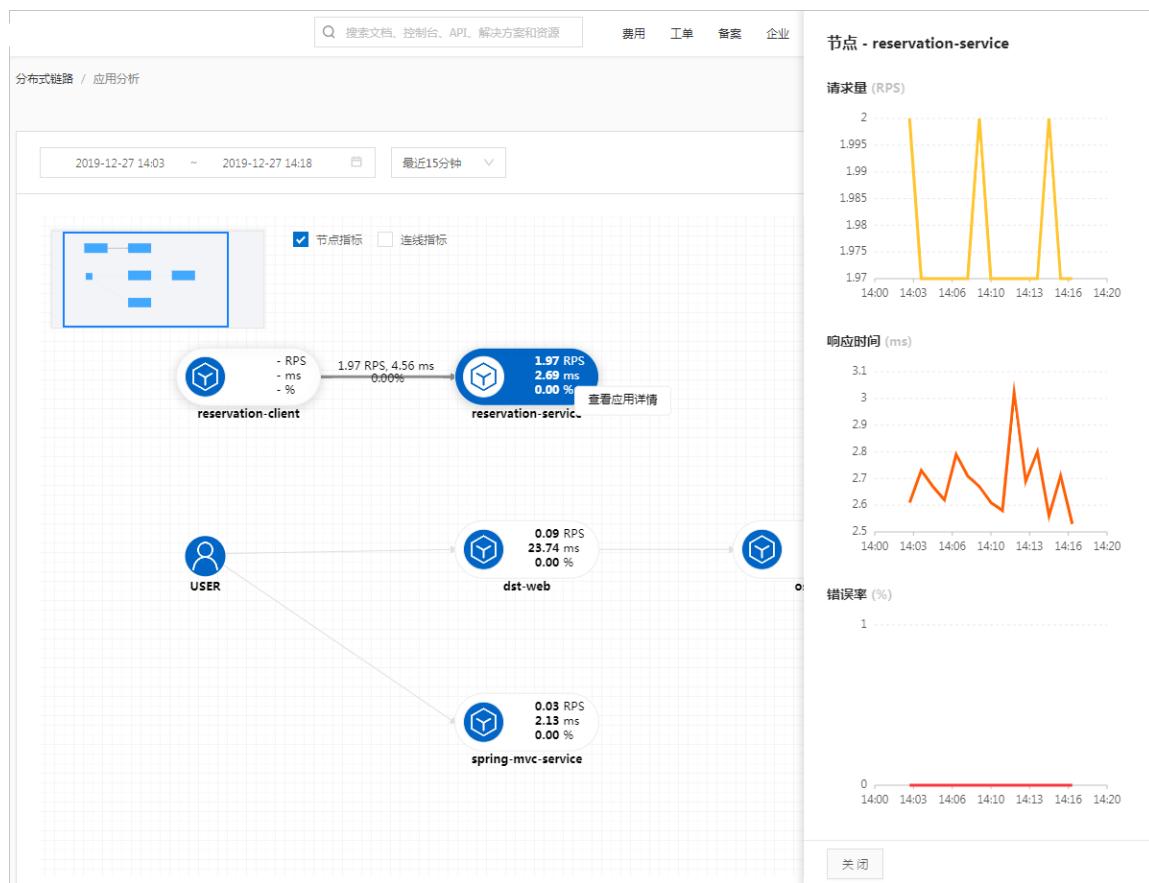
#### 3.4 查看应用拓扑关系

**应用分析** 页面展现了全局范围内运行的应用在指定时间段内的性能数据和拓扑结构。



### 查看应用拓扑图

1. 进入分布式链路跟踪控制台页面，左侧导航栏中选择 **应用分析**。
2. 在应用拓扑图的左上方，您可以选择时间范围，或者设置自定义的时间范围。默认时间范围为最近 15 分钟，最长时间间隔为 7 天。
3. 您可以勾选想要查看的指标数据，节点指标与连线指标：
  - 勾选 **节点指标** 后，即可在拓扑图中直接查看各个节点的性能指标数据。
  - 勾选 **连线指标** 后，拓扑图中将会显示各个节点之间调用链路的流量数据。
4. 在拓扑图中，点击任一应用节点或调用连线，可查看该应用或调用链路各性能指标的监控趋势图。



5. 点击一个应用节点后，该节点上方出现 **查看应用详情** 按钮。点击该按钮，即可前往该应用的详情页，详见 [查看应用详情](#)。

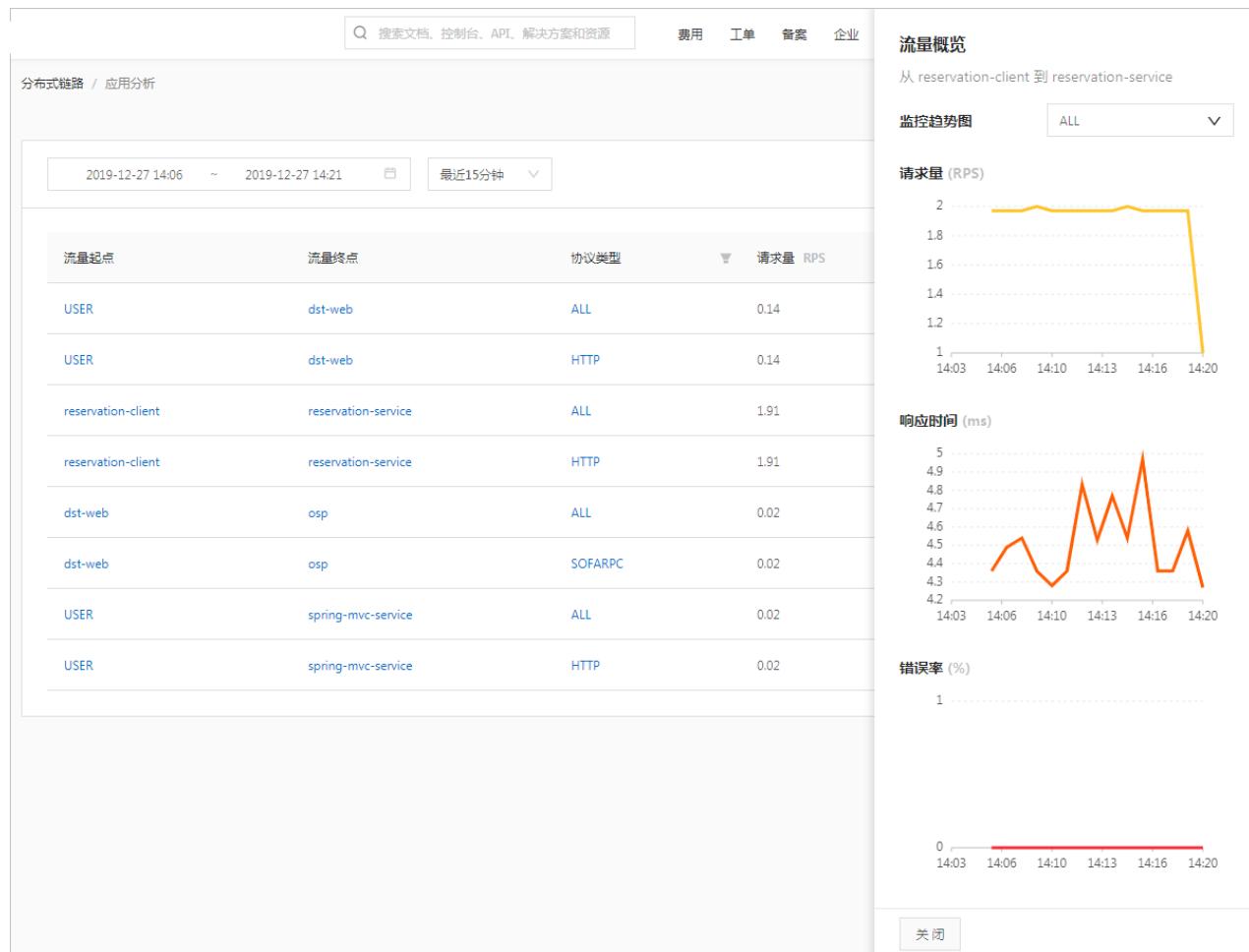
### 查看应用列表

在 **应用分析** 页面的右上角，点击 图标，可切换至应用列表视图，查看指定统计时段内的各个应用的概览信息，包括流量起点、流量终点、协议类型、请求量 (RPS)、响应时间 (ms) 以及错误率。

流量起点	流量终点	协议类型	请求量 RPS	响应时间 ms	错误率 %
USER	dst-web	ALL	0.11	22.15	0.00
USER	dst-web	HTTP	0.11	22.15	0.00
reservation-client	reservation-service	ALL	1.97	4.55	0.00
reservation-client	reservation-service	HTTP	1.97	4.55	0.00
dst-web	osp	ALL	0.01	13.92	0.00
dst-web	osp	SOFARPC	0.01	13.92	0.00
USER	spring-mvc-service	ALL	0.02	3.40	0.00
USER	spring-mvc-service	HTTP	0.02	3.40	0.00

您还可以对应用进行如下操作：

- 点击任意流量起点应用或终点应用名称，即可查看该节点详情，详见 [查看应用详情](#)。
- 点击协议类型，即可查看流量监控数据。



### 3.5 查看应用详情

[应用详情页](#) 展现了应用的详细监控数据，包括应用概览信息、服务调用信息、消息调用信息和数据库调用信息。

#### 操作步骤

- 进入分布式链路跟踪控制台页面，左侧导航栏中选择 [应用详情](#)。
- 在应用详情页的左上角，您可以在 [当前应用](#) 的下拉菜单中选择想要查看的目标应用。
- 在页面的右上方，您可以选择或自定义时间范围，默认范围是最近 15 分钟，支持的最长时间间隔为 7 天。
- 在当前页面，您可以切换页签，查看目标应用在指定时间范围内的各项应用数据：
  - 应用概览
  - 服务调用

- 消息
- 数据库

### 应用概览

**应用概览** 页签显示了当前应用的上下游拓扑及其应用指标趋势，包括应用的请求量、响应时间和错误率，如下图所示。



### 服务调用

**服务调用** 页签列出了当前应用发布和引用的所有服务及各个服务的概览信息，包括服务名、方法名、调用量、错误量和平均耗时。

应用概览 服务调用 消息 数据库

发布的服务 引用的服务

服务名 : com.████████████████.InstanceFacade:1.0 所属应用 : osp
方法名 调用量 次 平均耗时 ms 操作
getWorkspace 14.0 12.57 <a href="#">查看链路</a>
getInstanceId 1.0 21.00 <a href="#">查看链路</a>

服务名 : com.alipay.████████████████.ConfigFacade:1.0 所属应用 :
方法名 调用量 次 平均耗时 ms 操作
getOpenAPIConfig - <a href="#">查看链路</a>

共 2 条 < 1 >

此外，对于服务方法，您还可以进行以下操作，查看更多链路数据信息：

- 点击各项监控指标对应数据列的图表图标，即可查看该指标的趋势图。
- 点击 **查看链路** 按钮，可查看所有调用链路及其链路信息。点击 TraceID 会前往该链路的链路详情，详见 [查看链路详情](#)。

## 消息

**消息** 页签列出了当前应用发布和消费的所有信息及其每条消息的相关信息，包括消息主题 (topic)、消息事件码 (eventCode)、消息组 ID (groupId)、调用量、错误量和平均耗时。点击 **查看链路**，即可查看该消息所有调用链路的概览信息。

应用概览 服务调用 消息 数据库

我发布的消息 我消费的消息

Topic eventCode groupId 调用量 次 平均耗时 ms 操作
TP_DST_PAY_BY_CARD EC_PAY_BY_CARD S_dsttest_qx 8400 0.02 <a href="#">查看链路</a>
TP_DST_PAY_BY_BALANCE EC_PAY_BY_BALANCE S_dsttest_qx 8400 0.05 <a href="#">查看链路</a>

共 2 条 < 1 >

## 数据库

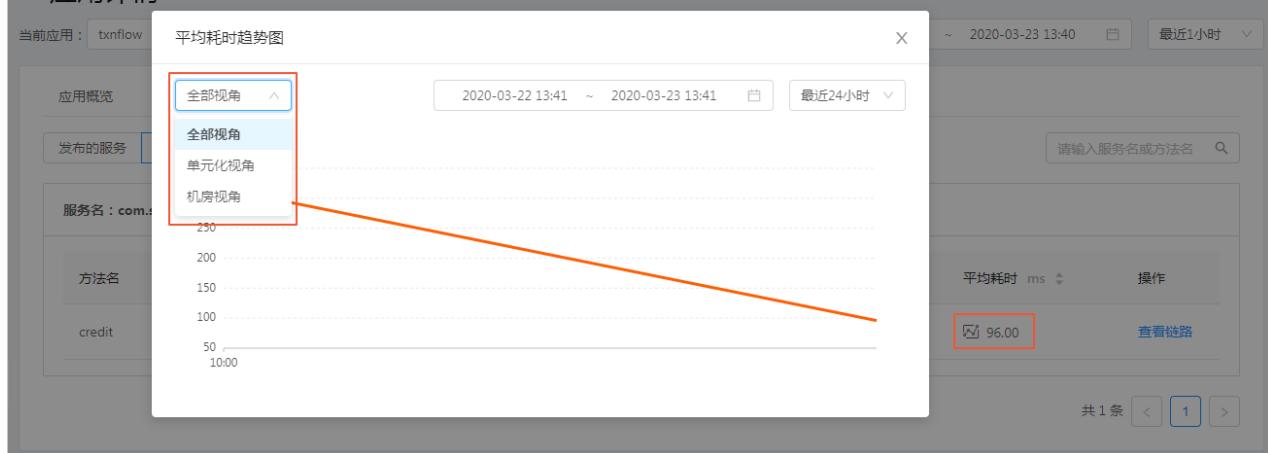
**数据库** 页签则列出了该应用调用的所有数据库及其相关信息，包括数据库地址、数据库名、调用量、错误量和平均耗时。点击 **查看链路**，即可查看数据库所有调用链路的概览信息。

应用概览	服务调用	消息	数据库		
数据库地址	数据库名	调用量 次	错误量 次	平均耗时 ms	操作
127.0.0.1:6350	dsttest	114221	31338	2.80	<a href="#">查看链路</a>
共 1 条 < 1 >					

### 单元化说明（仅专有云）

对于专有云开启了单元化功能的用户，在单元化部署环境下，点击趋势图图标查看相应监控指标的变化趋势时，您可以选择查看视角（单元化视角、机房视角），如下图所示。

[← 应用详情](#)



## 3.6 搜索调用链路

本文介绍了如何在 [链路搜索](#) 页面查询调用链路。

### 操作步骤

1. 进入分布式链路跟踪控制台，左侧导航栏中选择 [链路搜索](#)。
2. 在 [链路搜索](#) 页面上，您可以根据业务需求选择输入以下搜索项的值，并点击 [查询](#)。
  - 基础搜索项：
    - Trace Id：链路的唯一标识。
    - 调用时间：默认时间范围为最近 10 分钟。
    - 调用方式：可选项包括 ALL、RPC、HTTP、DB 与 MQ。
    - 应用名：应用的名称。
    - 结果：可选项包括全部、成功和失败。
    - 响应时长：可选择一个响应时间长度范围，单位 ms。
  - 自定义搜索项：
    - 中间件自定义搜索项：支持的搜索项包括 serviceName、methodName、statusCode、database、eventId、msgId、topic、eventCode。选择搜索项后，输入相应的 value 即可。
    - 业务自定义搜索项：根据您在 trace 链路中添加的自定义业务 tag（如身份证证、

手机号等），以 key=value 的格式输入对应的 tag 及其值。详见 [添加自定义业务搜索项](#)。

The screenshot shows a search interface for distributed tracing. At the top, there are fields for 'Trace Id' (请输入 traceld, 若无, 可使用条件查询), '调用时间' (调用时间: 2019-11-28 21:26:29 ~ 2019-11-28 21:36:29), '调用方式' (ALL), '应用名' (应用名: 全部), '响应时长 (ms)' (请输入 value, 回车确认), '中间件自定义搜索项' (请输入 serviceName=, 请输入 value, 回车确认), and '业务自定义搜索项' (请输入 key=value, 回车确认). Below these are 'Search' and 'Clear' buttons.

- 在搜索结果列表中，您可以获取链路的 TraceId、调用时间、响应时长、调用结果、客户端名称及地址、服务端名称及地址和服务信息。

The screenshot shows a table of search results with columns: TraceId, 调用时间 (Call Time), 响应时长 (Response Time), 调用结果 (Call Result), 客户端 (Client), 服务端 (Service), and 服务信息 (Service Info). The table lists 10 entries, each with a timestamp from 2019-11-28 21:21:50 to 21:30:0 ms, and various service names like auth, web, pay, and MQ. The last two rows show session-related services: SessionService#logout and SessionService#login. Navigation buttons at the bottom right include <, 1, 2, 3, 4, 5, ..., 1000, >.

- 点击任一链路的 TraceId，即可进入调用链路的详情页，查看链路详情。详见 [查看调用链路详情](#)。

[链路搜索](#) 页面底部还列出了您的链路搜索历史和已收藏的链路。

- 在搜索历史区域，点击 [清空历史](#) 即可清楚搜索记录。
- 您还可以点击链路右侧的收藏按钮，收藏链路，方便以后快速查看。

## 3.7 添加自定义业务搜索项

在遇到必须使用手机号或流水号来查询链路信息等个性化场景中，可以将这些信息设置为自定义 tag，添加到 span 中。

### 前置条件

- 应用必须基于 SOFABOOT 技术栈
- Tracer 必须满足如下条件：
  - 版本  $\geq 3.0.5$
  - 必须使用主动上报模式
  - 业务代码必须显式依赖 Tracer

### 说明：

- 主动上报模式：当 Tracer 版本  $\geq 3.0.5$  时，会默认使用该模式，低于该版本时，请先升级版本。
- 显式依赖 Tracer 指业务类代码中要调用 Tracer 类，如 SofaTraceContext 等。该操作会带来一定侵入性，请做好封装。

## 添加方法

您可以通过在业务类中调用 Tracer 类的方式，来设置自定义 tag 及对应的值，示例代码如下：

```
public void handlerXxx(Map<String, String> request) {  
    SofaTraceContext ctx = SofaTraceContextHolder.getSofaTraceContext();  
    SofaTracerSpan span = ctx.getCurrentSpan();  
    //显式依赖 Tracer，会带来一定侵入性，请做好封装。  
    if (span != null) {  
        //将参数添加到自定义tag : xxx_phone 中  
        span.setTag("xxx_phone", request.get("phone"));  
    }  
  
    // 处理业务逻辑  
    // ...  
}
```

## 结果验证

您可通过 **链路搜索** 功能中的 **业务自定义搜索项**，输入自定义 tag 及对应的参数，来验证自定义 tag 的添加结果。

假设自定义 tag 为 **xxx\_phone**，对应值为 **123**，如下所示：



The screenshot shows the 'Business Customized Search Item' input field with the value 'xxx\_phone=123' highlighted by a red box. Below the search bar is a table with columns: TraceId, 调用时间 (Call Time), 响应时长 (Response Time), 调用结果 (Call Result), 客户端 (Client), 服务端 (Service), and 服务信息 (Service Information). There are four rows of data, each corresponding to a trace ID starting with 'Oba...' and ending with '12857'. The '服务信息' column for the first row shows 'web(1 84)'.

TraceId	调用时间	响应时长	调用结果	客户端	服务端	服务信息
Oba...12857	2019-12-12 18:16	198 ms	● 失败	USER(?)	web(1 84)	-
Oba...2857	2019-12-12 18:16	50 ms	● 失败	USER(?)	web(11 84)	-
Oba...857	2019-12-12 18:16	4 ms	● 失败	USER(?)	web(11 84)	-
Oba...2857	2019-12-12 18:16	24 ms	● 失败	USER(?)	web(1 84)	-

然后，在目标链路的详情页，鼠标悬浮于 **服务信息** 列上，会弹出 **服务信息详情**，其中就包含了自定义 tag 及对应参数值，如下所示：

应用名	SpanId	IP	调用类型	状态	服务信息
- web	0	10.10.10.84	DUMMY	● 失败	
web	0.1	10.10.10.84	SOFARPC	● 成功	AuthFacade#m1CheckUserState
- auth	0.1	10.10.10.105	SOFARPC	● 成功	AuthFacade#m1CheckUserState
auth	0.1.1	11.11.11.105	MQ	● 成功	producer TP_DST_TEST1
MQ@TP_DST_TEST1	0.1.1	-	MQ	● 成功	producer TP_DST_TEST1
auth	0.1.2	11.11.11.105	DB	● 成功	SELECT dsttest
DB@dsttest	0.1.2	10.10.10.39	DB	● 成功	SELECT dsttest
web	0.2	11.11.11.34	SOFARPC	● 失败	ShopFacade#m1GetRecommendItems
- shop	0.2	10.10.10.51	SOFARPC	● 失败	ShopFacade#m1GetRecommendItems
shop	0.2.1	11.11.11.51	SOFARPC	● 失败	AuthFacade#m1CheckUserState
auth	0.2.1	10.10.10.105	SOFARPC	● 失败	AuthFacade#m1CheckUserState

### 3.8 查看链路详情

链路详情页提供两种图表模式，展示了该链路的详细调用信息。您不仅可以查看到调用链路的所有细节，例如服务端信息、Trace 日志，也能通过上下游的拓扑图，理解该链路中的调用关系。

#### 单链路时序图展示

链路详情页默认显示该链路的时序图，展示了全量最详细的调用信息，用于深入了解所有调用链路细节。

- 您可以直观地获取到此调用链路关联应用的应用名、SpanId、IP 地址、调用类型、调用状态（成功 /失败）、服务信息概览和耗时。

应用名	SpanId	IP	调用类型	状态	服务信息
- web	0	10.10.10.34	DUMMY	● 成功	38ms
web	0.1	10.10.10.34	SOFARPC	● 成功	AuthFacade#m1CheckUserState
- auth	0.1	10.10.10.33	SOFARPC	● 成功	AuthFacade#m1CheckUserState
auth	0.1.1	11.11.11.33	MQ	● 成功	producer TP_DST_TEST1
MQ@TP_DST_TEST1	0.1.1	-	MQ	● 成功	producer TP_DST_TEST1

- 将鼠标悬浮至服务信息一栏，即可查看到更多详情，例如客户端信息、服务端信息、服务名、方法名、耗时、结果等。

应用名	SpanId	IP	调用类型	状态	服务信息
- web	0	10.10.10.34	DUMMY	● 成功	
web	0.1	10.10.10.34	SOFARPC	● 成功	AuthFacade#m1CheckUserState
- auth	0.1	10.10.10.33	SOFARPC	● 成功	AuthFacade#m1CheckUserState
auth	0.1.1	11.11.11.33	MQ	● 成功	producer TP_DST_TEST1
MQ@TP_DST_TEST1	0.1.1	-	MQ	● 成功	producer TP_DST_TEST1
auth	0.1.2	10.10.10.33	DB	● 成功	
DB@dsttest	0.1.2	10.10.10.33	DB	● 成功	
web	0.2	10.10.10.34	SOFARPC	● 成功	
- shop	0.2	10.10.10.73	SOFARPC	● 成功	ShopFacade#m1GetRecommendItems

- 点击应用名旁边的日志图标，就可查看该应用的日志信息，包括业务日志和错误日志。

**说明：**要使用日志功能，您必须提前进行应用日志配置和关联，详见 [设置应用日志关联](#)。

## 查看链路单元化信息（仅专有云）

**说明**：该功能仅适用于专有云开启了单元化功能的用户。

单元化部署环境下，在链路详情页单链路时序图中，您还可以查看到该链路的单元化信息，包括服务端和客户端分别所在的机房与单元、该调用链路的来源 Zone 与去向 Zone，如下图所示。

链路详情						
应用名	SpanId	IP	调用类型	状态	机房	单元化
txnflow	0	172.17.0.72	MVC	● 成功	-	-
txnflow	0.1	172.17.0.72	DB	● 成功	nmgpoc01	-
DB@poc_tf	0.1	172.17.0.39	DB	● 成功	nmgpoc01	-
txnflow	0.2	172.17.0.72	UNKNOWN	● 成功	nmgpoc01	单元内
txnflow	0.2.1	172.17.0.72	UNKNOWN	● 成功	nmgpoc01	-
txnflow	0.2.2	172.17.0.72	UNKNOWN	● 成功	nmgpoc01	-
txnflow	0.2.3	172.17.0.72	SOFARPC	● 成功	nmgpoc01	单元内

## 单链路拓扑图展示

点击拓扑图图标，即可切换至单链路拓扑图展示页，查看该链路的上下游调用拓扑关系以及各节点或调用关系间的性能数据（请求量、响应时间和错误率），如下图所示。

0aa026701575023671331299923962 ⭐

The screenshot shows a network monitoring interface with a grid background. At the top left are two icons: a document with a gear and a document with three nodes. To the right are filter and search icons. The top right corner has buttons for '画布' (Canvas) at 100%, '全屏' (Full Screen), and '图例' (Legend). Below these are two checked checkboxes: '节点指标' (Node Metrics) and '连线指标' (Link Metrics). On the left, there is a large rectangular box with a blue border containing three small blue squares. In the center, there is a sequence of three nodes connected by arrows. The first node is a user icon labeled 'USER'. The second node is a hexagon icon labeled 'spring-mvc-service' with metrics: 1次 17.00 ms 0.00%. The third node is another hexagon icon labeled 'spring-mvc-service-end' with metrics: 1.00 次 13.00 ms 0.00 %. Arrows indicate the flow from the user to the service and then to the end point.

## 4 SOFATracer

## 4.1 什么是 SOFATracer

SOFATracer 是蚂蚁金服基于 [OpenTracing 规范](#) 开发的分布式链路跟踪系统，其核心理念就是通过一个全局的 TraceId 将分布在各个服务节点上的同一次请求串联起来。通过统一的 TraceId 将调用链路中的各种网络调用情况以日志的方式记录下来同时也提供远程汇报到 [Zipkin](#) 进行展示的能力，以此达到透视化网络调用的目的。

## 功能描述

**基于 OpenTracing 规范提供分布式链路跟踪解决方案**

基于 [OpenTracing 规范](#) 并扩展其能力提供链路跟踪的解决方案。各个框架或者组件可以基于此实现，通过在各个组件中埋点的方式来提供链路跟踪的能力。

**提供异步落地磁盘的日志打印能力**

基于 [Disruptor](#) 高性能无锁循环队列，提供异步打印日志到本地磁盘的能力。框架或者组件能够在接入时，在异步日志打印的前提下可以自定义日志文件的输出格式。SOFATracer 提供两种类似的日志打印类型即摘要日志和统计日志，摘要日志：每一次调用均会落地磁盘的日志；统计日志：每隔一定时间间隔进行统计输出的日志。

**支持日志自清除和滚动能力**

异步落地磁盘的 SOFATracer 日志支持自清除和滚动能力，支持按照按照天清除和按照小时或者天滚动的能力

**基于 SLF4J MDC 的扩展能力**

SLF4J 提供了 MDC ( Mapped Diagnostic Contexts ) 功能，可以支持用户定义和修改日志的输出格式以及内容。SOFATracer 集成了 SLF4J MDC 功能，方便用户在只简单修改日志配置文件即可输出当前 Tracer 上下文的 TraceId 和 SpanId。

**界面展示能力**

SOFATracer 可以将链路跟踪数据远程上报到开源产品 [Zipkin](#) 做分布式链路跟踪的展示。

**统一配置能力**

配置文件中提供丰富的配置能力以定制化应用的个性需求。

## 应用场景

解决在实施大规模微服务架构时的链路跟踪问题，达到透视化网络调用的目的，并可用于故障的快速发现，服务治理等。

**组件埋点**

目前 SOFATracer 支持 Spring MVC、标准 JDBC 接口实现的数据库连接池 ( DBCP、Druid、c3p0、tomcat、HikariCP、BoneCP )、HttpClient、Dubbo、Spring Cloud OpenFeign 等开源组件，其他开源组件 ( 如 MQ、Redis ) 埋点支持在开发中。

## 4.2 TraceId 和 SpanId 生成规则

## TraceId 生成规则

SOFATracer 通过 TraceId 来将一个请求在各个服务器上的调用日志串联起来，TraceId 一般由接收请求经过的第一个服务器产生。

产生规则是：服务器 IP + 产生 ID 时候的时间 + 自增序列 + 当前进程号，比如：

```
0ad1348f1403169275002100356696
```

前 8 位 0ad1348f 即产生 TraceId 的机器的 IP，这是一个十六进制的数字，每两位代表 IP 中的一段，我们把这个数字，按每两位转成 10 进制即可得到常见的 IP 地址表示方式 10.209.52.143，大家也可以根据这个规律来查找到请求经过的第一个服务器。

后面的 13 位 1403169275002 是产生 TraceId 的时间。之后的 4 位 1003 是一个自增的序列，从 1000 涨到 9000，到达 9000 后回到 1000 再开始往上涨。最后的 5 位 56696 是当前的进程 ID，为了防止单机多进程出现 TraceId 冲突的情况，所以在 TraceId 末尾添加了当前的进程 ID。

说明：TraceId 目前的生成的规则参考了阿里的鹰眼组件。

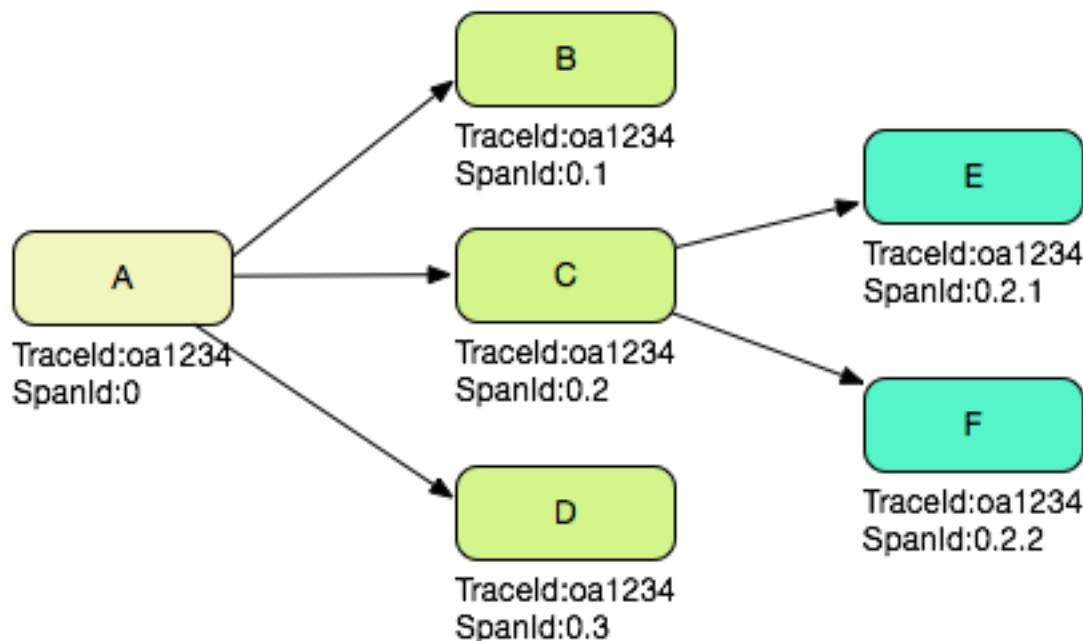
## SpanId 生成规则

SOFATracer 中的 SpanId 代表本次调用在整个调用链路树中的位置。

假设一个 Web 系统 A 接收了一次用户请求，那么在这个系统的 SOFATracer MVC 日志中，记录下的 SpanId 是 0，代表是整个调用的根节点，如果 A 系统处理这次请求，需要通过 RPC 依次调用 B，C，D 三个系统，那么在 A 系统的 SOFATracer RPC 客户端日志中，SpanId 分别是 0.1，0.2 和 0.3，在 B，C，D 三个系统的 SOFATracer RPC 服务端日志中，SpanId 也分别是 0.1，0.2 和 0.3；如果 C 系统在处理请求的时候又调用了 E，F 两个系统，那么 C 系统中对应的 SOFATracer RPC 客户端日志是 0.2.1 和 0.2.2，E，F 两个系统对应的 SOFATracer RPC 服务端日志也是 0.2.1 和 0.2.2。

根据上面的描述可以知道，如果把一次调用中所有的 SpanId 收集起来，可以组成一棵完整的链路树。

假设一次分布式调用中产生的 TraceId 是 0a1234（实际不会这么短），那么根据上文 SpanId 的产生过程，如下图所示：



说明 : SpanId 目前的生成的规则参考了阿里的鹰眼组件。

### 4.3 开始使用 SOFATracer

在使用 SOFATracer 时，您需要注意不同组件对应的 SOFATracer 版本和 JDK 版本。

#### 环境准备

要使用 SOFABoot，需要先准备好基础环境，SOFABoot 依赖以下环境：

- JDK 7 或 JDK 8
- 需要采用 Apache Maven 3.2.5 或者以上的版本来编译

#### 示例列表

下面所有示例工程均为 SOFABoot 工程（同时支持 SpringBoot 工程中使用），关于如何创建 SOFABoot 工程请参考 SOFABoot 快速入门。

- 组件接入
  - Spring MVC 埋点接入
  - HttpClient 埋点接入
  - DataSource 埋点接入
  - RestTemplate 埋点接入
  - OkHttp 埋点接入
  - SOFARPC 埋点接入
  - Dubbo 埋点接入
  - Spring Cloud OpenFeign 埋点接入

- 采样模式
- 上报数据到 Zipkin

## 4.4 Spring MVC 埋点接入

本文档将介绍如何使用 SOFATracer 对 SpringMVC 进行埋点。假设您已经基于 SOFABoot 构建了一个简单的 Spring Web 工程，那么可以通过以下步骤进行操作：

1. 引入 Tracer 依赖
2. 添加 Controller
3. 运行工程
4. 查看日志

### 引入 Tracer 依赖

在 SOFABoot 的 Web 项目中引入如下 Tracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

添加 Tracer starter 依赖后，可在 SOFABoot 的全局配置文件中添加配置项目以定制 Tracer 的行为。详情见 Tracer 配置项说明。

### 添加 Controller

如果您的 Web 工程中没有基于 Spring MVC 框架构建的 Controller，那么可以按照如下方式添加一个 Controller；如果有 Controller，那么可直接访问相应的服务。

```
@RestController
public class SampleRestController {

    private static final String template = "Hello, %s!";

    private final AtomicLong counter = new AtomicLong();

    @RequestMapping("/greeting")
    public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {
        Greeting greeting = new Greeting();
        greeting.setSuccess(true);
        greeting.setId(counter.incrementAndGet());
        greeting.setContent(String.format(template, name));
        return greeting;
    }

    public static class Greeting {
```

```
private boolean success = false;
private long id;
private String content;

public boolean isSuccess() {
    return success;
}

public void setSuccess(boolean success) {
    this.success = success;
}

public long getId() {
    return id;
}

public void setId(long id) {
    this.id = id;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}
```

## 运行工程

可以将 SOFABoot 工程导入到 IDE 中，工程编译正确后，运行工程里面中的 main 方法启动应用。以上面添加的 Controller 为例，可以通过在浏览器中输入 <http://localhost:8080/greeting> 来访问 REST 服务，结果类似如下：

```
{
  success: true,
  id: 1,
  content:"Hello, World!"
}
```

## 查看日志

在 SOFABoot 的配置文件 application.properties 中可定义日志打印目录。假设配置的日志打印目录是 ./logs，即当前应用的根目录，应用名设置为 spring.application.name=mvc-client，那么在当前工程的根目录下可以看到类似如下结构的日志文件：

```
-- tracelog
|-- spring-mvc-digest.log
|-- spring-mvc-stat.log
```

打开 `spring-mvc-digest.log` 可看到具体的输出内容。下面是一条日志记录的例子：

```
2018-07-17 20:01:34.719,mvc-client,0a0fe91a1531828894436100149692,0,http://localhost:8080/greeting,GET,200,-1B,49B,281ms,http-nio-8080-exec-1,
```

各输出字段的具体含义，详见 [日志格式 > Spring MVC 日志](#)。

#### Tracer 配置项说明

SOFATracer 配置项	说明	默认值
<code>logging.path</code>	日志输出目录	SOFATracer 会优先输出到 <code>logging.path</code> 目录下；如果没有配置日志输出目录，那默认输出到 <code>\$(user.home}</code>
<code>com.alipay.sofa.tracer.disableDigestLog</code>	是否关闭所有集成 SOFATracer 组件摘要日志打印	false
<code>com.alipay.sofa.tracer.disableConfiguration[\$logType]</code>	关闭指定 <code>\$(logType)</code> 的 SOFATracer 组件摘要日志打印。 <code>\$(logType)</code> 是指具体的日志类型，如： <code>spring-mvc-digest.log</code>	false
<code>com.alipay.sofa.tracer.tracerGlobalRollingPolicy</code>	SOFATracer 日志的滚动策略	<code>yyyy-MM-dd</code> ：按照天滚动； <code>yyyy-MM-dd.HH</code> ：按照小时滚动。 默认不配置按照天滚动。
<code>com.alipay.sofa.tracer.tracerGlobalLogReserveDay</code>	SOFATracer 日志的保留天数	默认保留 7 天
<code>com.alipay.sofa.tracer.statLogInterval</code>	统计日志的时间间隔，单位：秒	默认 60 秒统计日志输出一次
<code>com.alipay.sofa.tracer.baggageMaxLength</code>	透传数据能够允许存放的最大长度	默认值 1024
<code>com.alipay.sofa.tracer.springmvc.filterOrder</code>	SOFATracer 集成在 Spring MVC 的 Filter 生效的 Order	- 2147483647 ( <code>org.springframework.core.Ordered#HIGHEST_PRECEDENCE + 1</code> )
<code>com.alipay.sofa.tracer.springmvc.urlPatterns</code>	SOFATracer 集成在 SpringMVC 的 Filter 生效的 URL Pattern 路径	<code>/* 全部生效</code>

## 4.5 HttpClient 埋点接入

说明：此功能适用于 HttpClient 4.3.x - 4.5.x 版本。

[HttpClient](#) 是 Apache 项目的开源组件，用来提供高效的、最新的、功能丰富的支持 HTTP 协议的客户端编程工具包。

您可以在 SOFABoot 工程中引入并使用 HttpClient 打印 Tracer 日志。

本文将演示如何使用 SOFATracer 对 HttpClient 进行埋点。假设您已经基于 SOFABoot 构建了一个简单的 Spring Web 工程，那么可以通过如下步骤进行操作：

1. 引入 Maven 依赖
2. 添加提供 RESTful 服务的 Controller
3. 构造并发起对 HttpClient 的调用
4. 运行工程
5. 查看日志

### 引入 Maven 依赖

为了使用 HttpClient 打印 Tracer 日志，您必须在项目的 pom.xml 文件中引入以下 Maven 依赖：

- SOFATracer 依赖

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

### 基于 SOFATracer 的 HttpClient 插件

为了使得 HttpClient 这个第三方开源组件能够支持 SOFATracer 的链路调用，SOFATracer 提供了 HttpClient 的插件扩展，即 tracer-enterprise-httpclient-plugin：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-httpclient-plugin</artifactId>
</dependency>
```

- HttpClient 依赖

```
<dependency>
<groupId>org.apache.httpcomponents</groupId>
<artifactId>httpclient</artifactId>
<!-- 版本 4.3.x - 4.5.x -->
<version>4.5.3</version>
</dependency>
<dependency>
<groupId>org.apache.httpcomponents</groupId>
<artifactId>httpasyncclient</artifactId>
<!-- 版本 4.1.x -->
<version>4.1.3</version>
</dependency>
```

### 添加一个提供 RESTful 服务的 Controller

示例代码：

```
@RestController
public class SampleRestController {

    private final AtomicLong counter = new AtomicLong(0);

    /**
     * Request http://localhost:8080/httpclient?name=
     * @param name name
     * @return Map of Result
     */
    @RequestMapping("/httpclient")
    public Map<String, Object> greeting(@RequestParam(value = "name", defaultValue = "httpclient") String name) {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("count", counter.incrementAndGet());
        map.put("name", name);
        return map;
    }
}
```

### 构造 HttpClient 并发起对 RESTful 服务的调用

为了使得工程中使用 SOFATracer 的 HttpClient 能够正确埋点和打印日志，您必须使用 com.alipay.sofa.tracer.enterprise.plugins.SofaTracerEnterpriseHttpClientBuilder 类去构造 HttpClient 的实例，并显式调用 clientBuilder 方法。

SofaTracerEnterpriseHttpClientBuilder 类提供了 clientBuilder ( 同步 ) 和 asyncClientBuilder ( 异步 ) 方法，以构造出一个经过 SOFATracer 埋点的org.apache.http.impl.client.HttpClientBuilder 实例。方法签名如下：

```
// 同步调用构造方法
public static HttpClientBuilder clientBuilder(HttpClientBuilder clientBuilder) ;

// 同步调用构造方法，并在日志中显示当前应用和目标应用的字段
public static HttpClientBuilder clientBuilder(HttpClientBuilder clientBuilder,
String currentApp, String targetApp);

// 异步调用构造方法
public static HttpAsyncClientBuilder asyncClientBuilder(HttpAsyncClientBuilder httpAsyncClientBuilder);

// 异步调用构造方法，并在日志中显示当前应用和目标应用的字段
public static HttpAsyncClientBuilder asyncClientBuilder(HttpAsyncClientBuilder httpAsyncClientBuilder,
String currentApp, String targetApp) ;
```

### 代码示例

- 构造 HttpClient 同步调用示例：

```
HttpClientBuilder httpClientBuilder = HttpClientBuilder.create();
//SOFATracer
SofaTracerEnterpriseHttpClientBuilder.clientBuilder(httpClientBuilder,"testSyncClient","testSyncServer");
CloseableHttpClient httpClient = httpClientBuilder.setConnectionManager(connManager).disableAutomaticRetries()
.build();
```

- 构造 HttpClient 异步调用示例：

```
RequestConfig requestConfig =  
RequestConfig.custom().setSocketTimeout(6000).setConnectTimeout(6000).setConnectionRequestTimeout(6000).bu  
ild();  
HttpAsyncClientBuilder httpAsyncClientBuilder = HttpAsyncClientBuilder.create();  
//tracer  
SofaTracerEnterpriseHttpClientBuilder.asyncClientBuilder(httpAsyncClientBuilder, "testAsyncClient", "testAsyncServer  
");  
CloseableHttpAsyncClient asyncHttpclient = httpAsyncClientBuilder.setDefaultRequestConfig(requestConfig).build();
```

通过 SofaTracerEnterpriseHttpClientBuilder 构造的 HttpClient 实例在发起对上文的 RESTful 服务调用的时候，就会埋点 SOFATracer 的链路数据。

## 运行工程

1. 将 SOFABoot 工程导入到 IDE 中。
2. 编译并运行工程里面中的 main 方法启动应用。
3. 通过 SofaTracerEnterpriseHttpClientBuilder 构造的 HttpClient 发起对 Controller 提供的 <http://localhost:8080/httpclient> RESTful 服务。

具体操作方法请参考 SOFABoot 快速入门。

## 查看日志

Tracer 日志的目录路径在 SOFABoot 的配置文件 application.properties 中定义。

在以下示例中，假设配置的日志打印目录是 ./logs，即当前应用的根目录，同时假设通过 SofaTracerEnterpriseHttpClientBuilder 传入的当前应用名称为 testSyncClient 和目标应用名称 testSyncServer，那么在当前工程的根目录下可以看到类似如下结构的日志文件：

```
-- tracelog  
|-- httpclient-digest.log  
|-- httpclient-stat.log
```

以 HttpClient 同步调用为例，摘要日志 httpclient-digest.log 如下：

```
2018-10-09  
20:57:29.923,testSyncClient,0a0fe9271539089849647100179895,0,http://localhost:49685/httpclient,GET,200,0B,-  
1B,275ms,main,testSyncServer,,
```

以 HttpClient 同步调用为例，统计日志 httpclient-stat.log 如下：

```
2018-10-09 20:57:30.596,testSyncClient,http://localhost:49685/httpclient,GET,1,275,Y,F
```

有关 HttpClient 日志字段的具体含义，参见 日志格式 > HttpClient 日志。

## 4.6 DataSource 埋点接入

本文档将介绍如何使用 SOFATracer 对 DataSource 进行埋点。

SOFATracer 2.2.0 基于标准的 JDBC 接口实现，支持对标准的数据库连接池（如 DBCP、Druid、c3p0、tomcat、HikariCP、BoneCP）埋点。下面演示如何接入 SOFATracer 埋点能力。

假设您已经基于 SOFABoot 构建了一个简单的 Spring Web 工程，那么可以通过如下步骤进行操作：

1. 引入 Maven 依赖
2. 配置数据源
3. 本地应用配置
4. 新建 REST 服务
5. 运行工程
6. 查看日志

### 引入 Maven 依赖

#### 引入 Tracer 依赖

在 SOFABoot 的 Web 项目中引入如下 Tracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

添加 Tracer 依赖后，可在 SOFABoot 的全局配置文件中添加配置项目以定制 Tracer 的行为。

#### 引入 H2Database 依赖

为了方便，此处使用 H2Database 内存数据库测试，引入如下依赖：

```
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
</dependency>

<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
```

#### 引入连接池依赖

引入所需的连接池依赖包，如 druid, c3p0, tomcat, dbcp, Hikari 等。

```
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>druid</artifactId>
<version>1.0.12</version>
</dependency>
<dependency>
<groupId>c3p0</groupId>
<artifactId>c3p0</artifactId>
<version>0.9.1.1</version>
</dependency>
<dependency>
<groupId>org.apache.tomcat</groupId>
<artifactId>tomcat-jdbc</artifactId>
<version>8.5.31</version>
</dependency>
<dependency>
<groupId>commons-dbcp</groupId>
<artifactId>commons-dbcp</artifactId>
<version>1.4</version>
</dependency>
<dependency>
<groupId>com.zaxxer</groupId>
<artifactId>HikariCP-java6</artifactId>
<version>2.3.8</version>
</dependency>
```

## 配置数据源

此处以 HikariCP 为例，新建一个名为 datasource.xml 的 Spring 配置文件，并定义如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- dataSource pool -->
    <bean id="simpleDataSource" class="com.zaxxer.hikari.HikariDataSource" destroy-method="close" primary="true">
        <property name="driverClassName" value="org.h2.Driver"/>
        <property name="jdbcUrl" value="jdbc:h2:~/test"/>
        <property name="username" value="sofa"/>
        <property name="password" value="123456"/>
    </bean>
</beans>
```

## 本地应用配置

- 必要配置：

引入 Tracer 需要强制配置应用名，否则应用启动失败。这一属性和 SOFABoot 框架要求一致。配置示例如下：

```
spring.application.name=SOFATracerDataSource
```

**非必要配置：**

为了此示例工程正常运行，需要配置 H2Database 属性；另为了方便查看日志，需要配置日志路径。配置示例如下：

```
# logging path
logging.path=./logs

# h2 web consloe 路径
spring.h2.console.path=/h2-console
# 开启 h2 web consloe， 默认为 false
spring.h2.console.enabled=true
# 允许远程访问 h2 web consloe
spring.h2.console.settings.web-allow-others=true

spring.datasource.username=sofa
spring.datasource.password=123456
spring.datasource.url=jdbc:h2:~/test
spring.datasource.driver-class-name=org.h2.Driver
```

**新建 REST 服务**

新建一个 REST 服务，触发 SQL 语句执行，便于查看 SQL 的 Tracer 记录。在以下 REST 服务创建中，触发了一个建表操作。

```
@RestController
public class SimpleRestController {

    @Autowired
    private DataSource simpleDataSource;

    @RequestMapping("/create")
    public Map<String, Object> create() {
        Map<String, Object> resultMap = new HashMap<String, Object>();
        try {
            Connection cn = simpleDataSource.getConnection();
            Statement st = cn.createStatement();
            st.execute("DROP TABLE IF EXISTS TEST;");
            + "CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));";
            resultMap.put("success", true);
            resultMap.put("result", "CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));");
        } catch (Throwable throwable) {
            resultMap.put("success", false);
            resultMap.put("error", throwable.getMessage());
        }
        return resultMap;
    }
}
```

**运行工程**

可以将 SOFABoot 工程导入到 IDE 中，工程编译正确后，运行工程里面中的 main 方法启动应用。启动后，通过在浏览器中访问 [localhost:8080/create](http://localhost:8080/create) 来执行上述 REST 服务。

## 查看日志

可以在 ./logs/datasource-client-digest.log 和 ./logs/datasource-client-stat.log 看到 SQL 执行的 Tracer 日志。

- datasource-client-digest.log 示例：

```
{"time":"2019-09-02  
21:31:31.566","local.app":"SOFATracerDataSource","traceId":"0a0fe91d156743109138810017302","spanId":  
"0.1","span.kind":"client","result.code":"00","current.thread.name":"http-nio-8080-exec-  
1","time.cost.milliseconds":"15ms","database.name":"test","sql":"DROP TABLE IF EXISTS TEST;  
CREATE TABLE TEST(ID INT PRIMARY KEY%2C NAME  
VARCHAR(255));","connection.establish.span":"128ms","db.execute.cost":"15ms","database.type":"h2","data  
base.endpoint":"jdbc:h2:~/test:-1","sys.baggage":"","biz.baggage":""}
```

- datasource-client-stat.log 示例：

```
{"time":"2019-09-02  
21:31:50.435","stat.key":{"local.app":"SOFATracerDataSource","database.name":"test","sql":"DROP TABLE IF EXISTS  
TEST;  
CREATE TABLE TEST(ID INT PRIMARY KEY%2C NAME  
VARCHAR(255))"},"count":1,"total.cost.milliseconds":15,"success":true,"load.test":"F"}
```

## 其他

在 SOFABoot 工程引入 Tracer 依赖后，会自动默认开启 DataSource 的埋点。如需禁止启动 DataSource 埋点，可以通过如下开关配置。

```
com.alipay.sofa.tracer.datasource.enable=false
```

## 注意事项

- 引入 SOFATracer 需要强制配置应用名，否则应用启动失败。属性名称为：spring.application.name

SOFATracer 2.2.0 基于标准的 JDBC 接口实现，理论上支持对所有标准的数据库连接池（如 DBCP，BoneCP 等）埋点。在 Spring Boot 环境，对于 DBCP、Druid、c3p0、tomcat、HikariCP 五种连接池支持自动埋点，即用户只需要引入 SOFATracer 依赖即可。在非 Spring Boot 环境或者对其他连接池（如 BoneCP）还需要增加手动配置，比如：

```
<bean id="smartDataSource" class="com.alipay.sofa.tracer.plugins.datasource.SmartDataSource" init-  
method="init">  
    <property name="delegate" ref="simpleDataSource"/>  
    <!--应用名称 -->  
    <property name="appName" value="yourAppName"/>  
    <!--数据库名称 -->  
    <property name="database" value="yourDatabase"/>  
    <!--数据库类型，支持MYSQL, ORACLE-->  
    <property name="dbType" value="MYSQL"/>  
</bean>
```

```
<bean id="simpleDataSource" class="com.jolbox.bonecp.BoneCPDataSource" destroy-method="close">
<property name="driverClass" value="com.mysql.jdbc.Driver"/>
<property name="jdbcUrl" value="jdbc:mysql://127.0.0.1/yourdb"/>
<property name="username" value="root"/>
<property name="password" value="abcdefg"/>
...
</bean>
```

如上所示，您需要额外配置 SOFATracer 提供的  
com.alipay.sofa.tracer.plugins.datasource.SmartDataSource。

## 4.7 RestTemplate 埋点接入

本文将参考 [示例工程](#) 演示如何使用 SOFATracer 对 RestTemplate 进行埋点。

假设您已经基于 SOFABoot 构建了一个简单的 Spring Web 工程，那么可以通过如下步骤进行操作：

1. 引入依赖
2. 工程配置
3. 添加 Controller
4. 获取 RestTemplate
5. 启动应用
6. 查看日志

### 引入依赖

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

### 工程配置

在工程的 application.properties 文件下添加 SOFATracer 要使用的参数，包括 spring.application.name 用于标示当前应用的名称；logging.path 用于指定日志的输出目录。

```
# Application Name
spring.application.name=TestTemplateDemo
# logging path
logging.path=./logs
```

### 添加一个提供 RESTful 服务的 Controller

在工程代码中，添加一个简单的 Controller，例如：

```
@RestController
public class SampleController {
    private final AtomicLong counter = new AtomicLong(0);
    @RequestMapping("/rest")
    public Map<String, Object> rest() {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("count", counter.incrementAndGet());
        return map;
    }

    @RequestMapping("/asyncrest")
    public Map<String, Object> asyncrest() throws InterruptedException {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("count", counter.incrementAndGet());
        Thread.sleep(5000);
        return map;
    }
}
```

## 构造 RestTemplate

以 API 方式构造 RestTemplate 发起一次对上文的 RESTful 服务的调用：

### 构造 RestTemplate 同步调用实例

```
RestTemplate restTemplate = SofaTracerRestTemplateBuilder.buildRestTemplate();
 ResponseEntity<String> responseEntity = restTemplate.getForEntity(
 "http://sac.alipay.net:8080/rest", String.class);
```

### 构造 RestTemplate 异步调用实例

```
AsyncRestTemplate asyncRestTemplate = SofaTracerRestTemplateBuilder
.buildAsyncRestTemplate();
 ListenableFuture<ResponseEntity<String>> forEntity = asyncRestTemplate.getForEntity(
 "http://sac.alipay.net:8080/asyncrest", String.class);
```

## 获取 RestTemplate

以自动注入的方式获取 RestTemplate：

```
@Autowired
RestTemplate restTemplate;
```

## 启动应用

启动 SOFABoot 应用，将会在控制台中看到启动打印的日志：

```
2018-10-24 10:45:28.683 INFO 5081 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX
exposure on startup
```

```
2018-10-24 10:45:28.733 INFO 5081 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2018-10-24 10:45:28.736 INFO 5081 --- [ main] c.a.s.t.e.r.RestTemplateDemoApplication : Started RestTemplateDemoApplication in 2.163 seconds (JVM running for 3.603)
```

调用成功的日志如下：

```
2018-10-24 10:45:28.989 INFO 5081 --- [ main] c.a.s.t.e.r.RestTemplateDemoApplication : Response is {"count":1}
2018-10-24 10:45:34.014 INFO 5081 --- [ main] c.a.s.t.e.r.RestTemplateDemoApplication : Async Response is {"count":2}
2018-10-24 10:45:34.014 INFO 5081 --- [ main] c.a.s.t.e.r.RestTemplateDemoApplication : test finish .....
```

### 查看日志

在上面的 application.properties 里面，配置的日志打印目录是 ./logs，即当前应用的根目录（您可以根据自己的实践需要进行配置），在当前工程的根目录下可以看到类似如下结构的日志文件：

```
./logs
├── spring.log
└── tracelog
├── resttemplate-digest.log
├── resttemplate-stat.log
├── spring-mvc-digest.log
├── spring-mvc-stat.log
├── static-info.log
└── tracer-self.log
```

示例中通过构造两个 RestTemplate（一个同步一个异步）发起对同一个 RESTful 服务的调用，调用完成后可以在 restTemplate-digest.log 中看到类似如下的日志：

### 摘要日志

```
{"time":"2019-09-03
10:33:10.336","local.app":"RestTemplateDemo","traceId":"0a0fe9271567477985327100211176","spanId":"0",
"span.kind":"client","result.code":"200","current.thread.name":"SimpleAsyncTaskExecutor-1",
"time.cost.milliseconds":"5009ms","request.url":"http://localhost:8801/asyncrest","method":"GET","req.size.bytes":0,"resp.size.bytes":0,"sys.baggage":"","biz.baggage":""}
```

### 统计日志

```
{"time":"2019-09-03
10:34:04.130","stat.key":{"method":"GET","local.app":"RestTemplateDemo","request.url":"http://localhost:8801/asyncrest"},"count":1,"total.cost.milliseconds":5009,"success":"true","load.test":"F"}
```

## 4.8 OkHttp 埋点接入

本文将基于 [示例工程](#) 介绍如何使用 SOFATracer 对 OkHttp 进行埋点。

假设您已经基于 SOFABoot 构建了一个简单的 Spring Web 工程，那么可以通过如下步骤进行操作：

1. 引入依赖
2. 工程配置
3. 添加 Controller
4. 构造 OkHttp Client 调用实例
5. 启动应用
6. 查看日志

## 引入依赖

```
<!-- SOFATracer 依赖 -->
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
<!-- okhttp 依赖 -->
<dependency>
<groupId>com.squareup.okhttp3</groupId>
<artifactId>okhttp</artifactId>
<version>3.12.1</version>
</dependency>
```

## 工程配置

在工程的 application.properties 文件下添加 SOFATracer 要使用的参数，包括spring.application.name 用于标示当前应用的名称；logging.path 用于指定日志的输出目录。

```
# Application Name
spring.application.name=OkHttpClientDemo
# logging path
logging.path=./logs
# port
server.port=8081
```

## 添加一个提供 RESTful 服务的 Controller

在工程代码中，添加一个简单的 Controller，例如：

```
@RestController
public class SampleRestController {

    private final AtomicLong counter = new AtomicLong(0);

    /**
     * Request http://localhost:8081/okhttp?name=sofa
     * @param name name
     * @return Map of Result
}
```

```
/*
@RequestMapping("/okhttp")
public Map<String, Object> greeting(@RequestParam(value = "name", defaultValue = "okhttp") String name) {
    Map<String, Object> map = new HashMap<>();
    map.put("count", counter.incrementAndGet());
    map.put("name", name);
    return map;
}
}
```

### 构造 OkHttp 发起一次对上文的 RESTful 服务的调用

构造 OkHttp Client 调用实例的代码示例如下：

```
OkHttpClientInstance httpClient = new OkHttpClientInstance();
String httpGetUrl = "http://localhost:8081/okhttp?name=sofa";
String responseStr = httpClient.executeGet(httpGetUrl);
```

### 启动应用

启动 SOFABoot 应用，在控制台中看到启动打印的日志如下：

```
2019-04-12 13:38:09.896 INFO 51193 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2019-04-12 13:38:09.947 INFO 51193 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8081 (http)
2019-04-12 13:38:09.952 INFO 51193 --- [ main] c.a.s.t.e.okhttp.OkHttpDemoApplication : Started OkHttpDemoApplication in 3.314 seconds (JVM running for 4.157)
```

当有类似如下的日志时，说明 OkHttp 的调用成功：

```
2019-04-12 13:38:10.205 INFO 51193 --- [ main] c.a.s.t.e.okhttp.OkHttpDemoApplication : Response is {"count":1,"name":"sofa"}
```

### 查看日志

在上面的 application.properties 里面，配置的日志打印目录是 ./logs，即当前应用的根目录（您可以根据自己的实践需要进行配置），在当前工程的根目录下可以看到类似如下结构的日志文件：

```
./logs
├── spring.log
└── tracelog
├── okhttp-digest.log
├── okhttp-stat.log
├── spring-mvc-digest.log
├── spring-mvc-stat.log
├── static-info.log
└── tracer-self.log
```

示例中通过构造 OkHttp 对象发起 RESTful 服务的调用，调用完成后可以在 okhttp-digest.log 中看到类似如

下的日志：

```
{"time": "2019-09-03 11:35:28.429", "local.app": "OkHttpDemo", "traceId": "0a0fe9271567481728265100112783", "spanId": "0", "span.kind": "client", "result.code": "200", "current.thread.name": "main", "time.cost.milliseconds": "164ms", "request.url": "http://localhost:8081/okhttp?name=sofa", "method": "GET", "result.code": "200", "req.size.bytes": 0, "resp.size.bytes": 0, "remote.app": "", "sys.baggage": "", "biz.baggage": ""}
```

## 4.9 Dubbo 埋点接入

在本文档将基于 [工程地址](#) 演示如何使用 SOFATracer 对 Dubbo 进行埋点。

### 基础环境

本案例使用的各框架组件的版本如下：

- SOFABoot 3.1.1/SpringBoot 2.1.0.RELEASE
- SOFATracer 2.4.0/3.0.4
- JDK 8

本案例包括三个子模块：

- tracer-sample-with-dubbo-consumer：服务调用方
- tracer-sample-with-dubbo-provider：服务提供方
- tracer-sample-with-dubbo-facade：接口

### 原理

SOFATracer 对象 Dubbo 的埋点实现依赖于 Dubbo 的 SPI 机制来实现，Tracer 中基于 [调用拦截扩展](#) 自定义了 DubboSofaTracerFilter 用于实现对 Dubbo 的调用埋点。由于 DubboSofaTracerFilter 并没有成为 Dubbo 的官方扩展，因此在使用 SOFATracer 时需要安装 [调用拦截扩展](#) 中所提供的方式进行引用，即：

```
<!-- 消费方调用过程拦截 -->
<dubbo:reference filter="dubboSofaTracerFilter"/>
<!-- 消费方调用过程缺省拦截器，将拦截所有reference -->
<dubbo:consumer filter="dubboSofaTracerFilter"/>

<!-- 提供方调用过程拦截 -->
<dubbo:service filter="dubboSofaTracerFilter"/>
<!-- 提供方调用过程缺省拦截器，将拦截所有service -->
<dubbo:provider filter="dubboSofaTracerFilter"/>
```

### 新建 SOFABoot 工程作为父工程

在创建好一个 Spring Boot 的工程之后，接下来就需要引入 SOFABoot 的依赖，首先，需要将上文中生成的 Spring Boot 工程的 zip 包解压后，修改 Maven 项目的配置文件 pom.xml，将

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>${spring.boot.version}</version>
<relativePath/>
</parent>
```

替换为：

```
<parent>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofaboot-enterprise-dependencies</artifactId>
<version>${sofa.boot.version}</version>
</parent>
```

这里的 \${sofa.boot.version} 指定具体的 SOFABoot 版本，参考 SOFABoot 版本说明。

### 新建 tracer-sample-with-dubbo-facade

提供一个接口：

```
public interface HelloService {
    String SayHello(String name);
}
```

### 新建 tracer-sample-with-dubbo-provider

在工程模块的 pom 文件中添加 SOFATracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

SOFATracer 版本受 SOFABoot 版本管控，如果使用的 SOFABoot 版本不匹配，则需要手动指定 tracer 版本，且版本需高于 2.4.0.

在工程的 application.properties 文件下添加相关参数：

```
# Spring boot application
spring.application.name=dubbo-provider
# Base packages to scan Dubbo Component: @org.apache.dubbo.config.annotation.Service
dubbo.scan.base-packages=com.alipay.sofa.tracer.examples.dubbo.impl
## Filter 必须配置
dubbo.provider.filter=dubboSofaTracerFilter
# Dubbo Protocol
```

```
dubbo.protocol.name=dubbo
## Dubbo Registry
dubbo.registry.address=zookeeper://localhost:2181
logging.path=./logs
```

使用注解方式发布 Dubbo 服务：

```
@Service
public class HelloServiceImpl implements HelloService {
@Override
public String SayHello(String name) {
return "Hello , "+name;
}
}
```

#### 新建 tracer-sample-with-dubbo-consumer

在工程模块的 pom 文件中添加 SOFATracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

在工程的 application.properties 文件下添加相关参数：

```
spring.application.name=dubbo-consumer
dubbo.registry.address=zookeeper://localhost:2181
# Filter 必须配置
dubbo.consumer.filter=dubboSofaTracerFilter
logging.path=./logs
```

服务引用：

```
@Reference(async = false)
public HelloService helloService;

@Bean
public ApplicationRunner runner() {
return args -> {
logger.info(helloService.SayHello("sofa"));
};
}
```

#### 测试

先后启动 tracer-sample-with-dubbo-provider 和 tracer-sample-with-dubbo-consumer 两个工程，然后查看以下日志：

### dubbo-client-digest.log

```
{"time":"2019-09-02 23:36:08.250","local.app":"dubbo-consumer","traceId":"1e27a79c156743856804410019644","spanId":"0","span.kind":"client","result.code":"000","current.thread.name":"http-nio-8080-exec-2","time.cost.milliseconds":"205ms","protocol":"dubbo","service":"com.glmapper.bridge.boot.service.HelloService","method":"SayHello","invoke.type":"sync","remote.host":"192.168.2.103","remote.port":"20880","local.host":"192.168.2.103","client.serialize.time":35,"client.deserialize.time":5,"req.size.bytes":336,"resp.size.bytes":48,"error":"","sys.baggage":"","biz.baggage":""}
```

### dubbo-server-digest.log

```
{"time":"2019-09-02 23:36:08.219","local.app":"dubbo-provider","traceId":"1e27a79c156743856804410019644","spanId":"0","span.kind":"server","result.code":"000","current.thread.name":"DubboServerHandler-192.168.2.103:20880-thread-2","time.cost.milliseconds":"9ms","protocol":"dubbo","service":"com.glmapper.bridge.boot.service.HelloService","method":"SayHello","local.host":"192.168.2.103","local.port":"62443","server.serialize.time":0,"server.deserialize.time":27,"req.size.bytes":336,"resp.size.bytes":0,"error":"","sys.baggage":"","biz.baggage":""}
```

### dubbo-client-stat.log

```
{"time":"2019-09-02 23:36:13.040","stat.key":{"method":"SayHello","local.app":"dubbo-consumer","service":"com.glmapper.bridge.boot.service.HelloService"},"count":1,"total.cost.milliseconds":205,"success":true,"load.test":"F"}
```

### dubbo-server-stat.log

```
{"time":"2019-09-02 23:36:13.208","stat.key":{"method":"SayHello","local.app":"dubbo-provider","service":"com.glmapper.bridge.boot.service.HelloService"},"count":1,"total.cost.milliseconds":9,"success":true,"load.test":"F"}
```

## 对 Dubbo 2.6.x 版本的兼容

SOFATracer 中 2.4.1/3.0.6 版本开始也提供了对于 Dubbo 2.6.x 版本系列的支持，详见 [tracer-dubbo-2.6.x](#)。

### 注意：

- 默认情况下，tracerlog 日志将会打在用户根目录下。
- SOFATracer 日志目录依赖 log-sofa-boot-starter 来根据 application.properties 配置的 logging.path 来决定，因此如果是 springboot 工程，请引入 log-sofa-boot-starter。

## 4.10 Spring Cloud OpenFeign 埋点接入

本文将演示如何使用 SOFATracer 对 Spring Cloud OpenFeign 进行埋点。

## 基础环境

本案例使用的各框架组件的版本如下：

- Spring Cloud Greenwich.RELEASE
- SOFABoot 3.1.1/SpringBoot 2.1.0.RELEASE
- SOFATracer 3.0.4
- JDK 8

本案例包括两个子工程：

- tracer-sample-with-openfeign-provider 服务提供方
- tracer-sample-with-openfeign-consumer 服务调用方

## 新建 SOFABoot 工程作为父工程

在创建好一个 Spring Boot 的工程之后，接下来就需要引入 SOFABoot 的依赖，首先，需要将上文中生成的 Spring Boot 工程的 zip 包解压后，修改 Maven 项目的配置文件 pom.xml，将

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>${spring.boot.version}</version>
<relativePath/>
</parent>
```

替换为：

```
<parent>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofaboot-enterprise-dependencies</artifactId>
<version>${sofa.boot.version}</version>
</parent>
```

这里的 \${sofa.boot.version} 指定具体的 SOFABoot 开源版版本，参考 SOFABoot 版本说明。

## 新建 tracer-sample-with-openfeign-provider

在工程模块的 pom.xml 文件中添加 SOFATracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
</dependency>
```

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

SOFATracer 版本受 SOFABoot 版本管控，如果使用的 SOFABoot 版本不匹配，则需要手动指定 tracer 版本，且版本需高于 3.0.4。

在工程的 application.properties 文件下添加相关参数

```
spring.application.name=tracer-provider
server.port=8800
spring.cloud.zookeeper.connect-string=localhost:2181
spring.cloud.zookeeper.discovery.enabled=true
spring.cloud.zookeeper.discovery.instance-id=tracer-provider
```

简单的资源类

```
@RestController
public class UserController {
    @RequestMapping("/feign")
    public String testFeign(HttpServletRequest request) {
        return "hello tracer feign";
    }
}
```

**新建 tracer-sample-with-openfeign-consumer**

在工程模块的 pom 文件中添加 SOFATracer 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

在工程的 application.properties 文件下添加相关参数：

```
spring.application.name=tracer-consumer
server.port=8082
```

```
spring.cloud.zookeeper.connect-string=localhost:2181
spring.cloud.zookeeper.discovery.enabled=true
spring.cloud.zookeeper.discovery.instance-id=tracer-consumer
```

定义 feign 资源：

```
@FeignClient(value = "tracer-provider", fallback = FeignServiceFallbackFactory.class)
public interface FeignService {
    @RequestMapping(value = "/feign", method = RequestMethod.GET)
    String testFeign();
}
```

开启服务发现和feign注解：

```
@SpringBootApplication
@RestController
@EnableDiscoveryClient
@EnableFeignClients
public class FeignClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(FeignClientApplication.class, args);
    }

    @Autowired
    private FeignService feignService;

    @RequestMapping
    public String test(){
        return feignService.testFeign();
    }
}
```

## 测试

先后启动 tracer-sample-with-openfeign-provider 和 tracer-sample-with-openfeign-consumer 两个工程；然后浏览器访问：<http://localhost:8082/>。然后查看日志：

在上面的 application.properties 里面，配置的日志打印目录是 ./logs 即当前应用的根目录（您可以根据自己的实践需要进行配置），在当前工程的根目录下可以看到类似如下结构的日志文件：

```
./logs
├── spring.log
└── tracelog
    ├── feign-digest.log
    ├── feign-stat.log
    ├── spring-mvc-digest.log
    ├── spring-mvc-stat.log
    ├── static-info.log
    └── tracer-self.log
```

示例中通过 SpringMvc 提供的 Controller 作为请求入口，然后使用 openfeign client 发起向下游资源的访问

调用，日志大致如下：

```
{"time":"2019-09-03 10:28:52.363","local.app":"tracer-consumer","traceId":"0a0fe9271567477731347100110969","spanId":"0.1","span.kind":"client","result.code":"200","current.thread.name":"http-nio-8082-exec-1","time.cost.milliseconds":"219ms","request.url":"http://10.15.233.39:8800/feign","method":"GET","error":"","req.size.bytes":0,"resp.size.bytes":18,"remote.host":"10.15.233.39","remote.port":"8800","sys.baggage":"","biz.baggage":""}
```

## 4.11 集成 SLF4J MDC 功能

SLF4J 提供了 MDC ( Mapped Diagnostic Contexts ) 功能，可以支持用户定义和修改日志的输出格式以及内容。本文将介绍 Tracer 集成的 SLF4J MDC功能，方便用户在只简单修改日志配置文件的前提下输出当前 Tracer 上下文 TraceId 以及 SpanId 。

### 使用方式

保证项目中已经引入如下的 Tracer 相关依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-core</artifactId>
</dependency>

<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-extensions</artifactId>
</dependency>
```

修改已有的日志配置文件中的 PatternLayout 属性：

- logback 示例

```
<!-- 此为 Tracer MDC 功能展示范例 appender , 实际使用时可删除 -->
<appender name="MDC-EXAMPLE-APPENDER" class="ch.qos.logback.core.rolling.RollingFileAppender">
<append>true</append>
<filter class="ch.qos.logback.classic.filter.LevelFilter">
<level>${logging.level}</level>
<onMatch>ACCEPT</onMatch>
<onMismatch>DENY</onMismatch>
</filter>
<file>${logging.path}/archetype-test-client/mdc-example.log</file>
<!-- to generate a log file everyday with a longest lasting of 30 days -->
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
<!-- logfile name with daily rolling-->
<fileNamePattern>${logging.path}/archetype-test-client/mdc-example.log.%d{yyyy-MM-dd}</fileNamePattern>
<!-- log perserve days-->
```

```

<MaxHistory>30</MaxHistory>
</rollingPolicy>
<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
<!--output format : %d is for date , %thread is for thread name , %-5level : loglevel with 5
character %msg : log message , %n line breaker-->
<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %X{SOFA-TraceId} %X{SOFA-
SpanId} %logger{50} - %msg%n</pattern>
<!-- encoding -->
<charset>UTF-8</charset>
</encoder>
</appender>

```

### log4j2 示例

```

<?xml version="1.0" encoding="UTF-8"?> <configuration>
<Appenders>
<Console name="CONSOLE" target="SYSTEM_OUT">
<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} %5p [%X{SOFA-TraceId},%X{SOFA-SpanId}] ----
%m%n" />
</Console>

<File name="File" fileName=".//logs/test.log">
<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} %5p [%X{SOFA-TraceId},%X{SOFA-SpanId}] ----
%m%n" />
</File>
</Appenders>
<Loggers>
<root level="info">
<AppenderRef ref="CONSOLE"/>
<AppenderRef ref="File"/>
</root>
</Loggers>
</configuration>

```

### 3. 在 PatternLayout 中增加 %X{SOFA-TraceId} 和 %X{SOFA-SpanId} 配置：

- %X{SOFA-TraceId}：对应 TraceId，实际运行时将会被替换为当前 Tracer 上下文的 TraceId，如果当前不存在 Tracer 上下文，则会被替换为空字符串。
- %X{SOFA-SpanId}：对应 SpanId，实际运行时将会被替换为当前 Tracer 上下文的 SpanId，如果当前不存在 Tracer 上下文，则会被替换为空字符串。

那么在业务代码中，可通过如下命令正常打印日志，不需要额外修改：

```
logger.info("rest mdc example");
```

实际输出的日志如下：

```
2018-02-05 15:52:45.037 [SOFA-REST-WORK-3-1] INFO 0a0fe86f151781716496210012303 0 MDC-EXAMPLE - rest
mdc example
```

## 4.12 异步处理

## 线程中使用 java.lang.Runnable

如果用户在代码中通过 `java.lang.Runnable` 新启动了线程或者采用了线程池去异步地处理一些业务，那么需要将 SOFATracer 日志上下文从父线程传递到子线程中去，SOFATracer 提供的 `com.alipay.common.tracer.core.async.SofaTracerRunnable` 默认完成了此操作，您可以按照以下方式使用：

```
Thread thread = new Thread(new SofaTracerRunnable(new Runnable() {
    @Override
    public void run() {
        //do something your business code
    }
})));
thread.start();
```

## 线程中使用 java.util.concurrent.Callable

如果用户在代码中通过 `java.util.concurrent.Callable` 新启动线程或者采用了线程池去异步地处理一些业务，那么需要将 SOFATracer 日志上下文从父线程传递到子线程中去，SOFATracer 提供的 `com.alipay.common.tracer.core.async.SofaTracerCallable` 默认完成了此操作，您可以按照以下方式使用：

```
ExecutorService executor = Executors.newCachedThreadPool();
SofaTracerCallable<Object> sofaTracerSpanSofaTracerCallable = new SofaTracerCallable<Object>(new
Callable<Object>() {
    @Override
    public Object call() throws Exception {
        return new Object();
    }
});
Future<Object> futureResult = executor.submit(sofaTracerSpanSofaTracerCallable);
//do something in current thread
Thread.sleep(1000);
//another thread execute success and get result
Object objectReturn = futureResult.get();
```

这个示例中，假设 `java.util.concurrent.Callable` 返回结果的对象类型是 `java.lang.Object`，实际使用时可以根据情况替换为期望的类型。

## SOFATracer 对线程池、异步调用场景下的支持

### 异步场景

异步调用，以 RPC 调用为例，每次 RPC 调用请求出去之后不会等待到结果返回之后才去发起下一次处理，这里有个时间差，在前一个 RPC 调用的 callback 回来之前，又一个新的 RPC 请求发起，此时当前线程中的 TracerContext 没有被清理，则 spanId 会自增，tracerId 相同。

对于上面这种情况，SOFATracer 在对于异步情况处理时，不会等到 callback 回来之后，调用 cr 阶段才会清理，而是提前就会清理当前线程的 tracerContext 上下文，从而保证链路的正确性。

### 线程池

目前来说，不管是 SOFARPC 还是 Dubbo 的埋点实现，在使用单线程或者线程池时，情况是一样的：

- 同步调用，线程池中分配一个线程用于处理 RPC 请求，在请求结束之前会一直占用线程；此种情况下不会造成下一个 RPC 请求错拿上一个请求的 tracerContext 数据问题
- 异步调用，由于异步回调并非是在 callback 中来清理上下文，而是提前清理的，所以也不会存在数据串用问题。
- callback 异步回调，这个本质上就是异步调用，所以处理情况和异步调用相同。

#### 相关链接

- [示例工程](#)

### 4.13 采样模式

目前 SOFATracer 提供了两种采样模式，一种是基于 BitSet 实现的基于固定采样率的采样模式，另外一种是提供给用户自定义实现采样的采样模式。

本文将通过示例来演示如何使用。本示例基于 tracer-sample-with-springmvc 工程，除 application.properties 之外，其他均相同。

#### 基于固定采样率的采样模式

在 application.properties 中增加采样相关配置项

```
#采样率 0~100
com.alipay.sofa.tracer.samplerPercentage=100
#采样模式类型名称
com.alipay.sofa.tracer.samplerName=PercentageBasedSampler
```

#### 验证方式

- 当采样率设置为100时，每次都会打印摘要日志。
- 当采样率设置为0时，不打印
- 当采样率设置为0~100之间时，按概率打印

以请求 10 次来验证下结果。

- 当采样率设置为 100 时，每次都会打印摘要日志。

启动工程，浏览器中输入：<http://localhost:8080/springmvc>，并且刷新地址10次，查看日志如下

：

```
{"time":"2018-11-09
11:54:47.643","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173568757510019269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":"
1,"resp.size.bytes":0,"time.cost.milliseconds":68,"current.thread.name":"http-nio-8080-exec-
1","baggage":""}
{"time":"2018-11-09
11:54:50.980","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569097710029269","spanId":"
1,"baggage":""}
```

```

0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":3,"current.thread.name":"http-nio-8080-exec-2","baggage":""}
{"time":"2018-11-09
11:54:51.542","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569153910049269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":3,"current.thread.name":"http-nio-8080-exec-4","baggage":""}
 {"time":"2018-11-09
11:54:52.061","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569205910069269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-6","baggage":""}
 {"time":"2018-11-09
11:54:52.560","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569255810089269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-8","baggage":""}
 {"time":"2018-11-09
11:54:52.977","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569297610109269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":1,"current.thread.name":"http-nio-8080-exec-10","baggage":""}
 {"time":"2018-11-09
11:54:53.389","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569338710129269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-2","baggage":""}
 {"time":"2018-11-09
11:54:53.742","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569374110149269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":1,"current.thread.name":"http-nio-8080-exec-4","baggage":""}
 {"time":"2018-11-09
11:54:54.142","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569414010169269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-6","baggage":""}
 {"time":"2018-11-09
11:54:54.565","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173569456310189269","spanId":"
0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-8","baggage":""}
 
```

## 2. 当采样率设置为 0 时，不打印日志。

启动工程，浏览器中输入：<http://localhost:8080/springmvc>；并且刷新地址 10 次，查看  
./logs/tracerlog/ 目录，没有 spring-mvc-digest.log 日志文件

## 3. 当采样率设置为 0~100 之间时，按概率打印日志。

此处设置采样率为 20。

- 刷新 10 次请求

```

{"time":"2018-11-09
12:14:29.466","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173686946410159846
","spanId":"0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"2
00","req.size.bytes":1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-
nio-8080-exec-5","baggage":""}
 {"time":"2018-11-09
12:15:21.776","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec154173692177410319846
","spanId":"0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code":"2
00","req.size.bytes":1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-

```

```
nio-8080-exec-2","baggage":""}
```

- 刷新 20 次请求

```
{"time":"2018-11-09
12:14:29.466","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec1541736869464101598
46","spanId":"0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code
":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-
5","baggage":""}
{"time":"2018-11-09
12:15:21.776","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec1541736921774103198
46","spanId":"0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code
":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-
2","baggage":""}
{"time":"2018-11-09
12:15:22.439","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec1541736922438103598
46","spanId":"0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code
":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":1,"current.thread.name":"http-nio-8080-exec-
6","baggage":""}
{"time":"2018-11-09
12:15:22.817","local.app":"SOFATracerSpringMVC","traceId":"0a0fe8ec1541736922815103798
46","spanId":"0.1","request.url":"http://localhost:8080/springmvc","method":"GET","result.code
":"200","req.size.bytes":-
1,"resp.size.bytes":0,"time.cost.milliseconds":2,"current.thread.name":"http-nio-8080-exec-
8","baggage":""}
```

说明：按 20% 进行采样，测试结果仅供参考。

## 自定义采样模式

SOFATracer 中提供了一个采样率计算的接口 Sampler，如果你需要定制化自己的采样策略，可通过实现此接口来进行扩展。下面通过一个简单示例来演示如何使用自定义采样模式。

### 自定义采样规则类

```
public class CustomOpenRulesSamplerRuler implements Sampler {

    private static final String TYPE = "CustomOpenRulesSamplerRuler";

    @Override
    public SamplingStatus sample(SofaTracerSpan sofaTracerSpan) {
        SamplingStatus samplingStatus = new SamplingStatus();
        Map<String, Object> tags = new HashMap<String, Object>();
        tags.put(SofaTracerConstant.SAMPLER_TYPE_TAG_KEY, TYPE);
        tags = Collections.unmodifiableMap(tags);
        samplingStatus.setTags(tags);
    }
}
```

```
if (sofaTracerSpan.isServer()) {  
    samplingStatus.setSampled(false);  
} else {  
    samplingStatus.setSampled(true);  
}  
return samplingStatus;  
  
@Override  
public String getType() {  
    return TYPE;  
}  
  
@Override  
public void close() {  
    // do nothing  
}
```

在 sample 方法中，用户可以根据当前 sofaTracerSpan 提供的信息来决定是否进行打印。这个案例是通过判断 isServer 来决定是否采样，isServer=true 即不采样，否则采样。

在 application.properties 中增加采样相关配置项

```
com.alipay.sofa.tracer.samplerName.samplerCustomRuleClassName=com.alipay.sofa.tracer.examples.springmvc.sa  
mpler.CustomOpenRulesSamplerRuler
```

相关试验结果，您可以自行验证。

## 4.14 上传数据到 Zipkin

本文将演示如何使用 SOFATracer 集成 Zipkin 进行数据上报展示。假设您已经基于 SOFABoot 构建了一个简单的 Spring Web 工程，那么可以通过如下步骤进行操作：

1. 引入依赖
2. 配置文件
3. 启动 Zipkin 服务端
4. 启动应用
5. 查看 Zipkin 服务端展示

引入依赖

添加 SOFATracer 依赖

工程中添加 SOFATracer 依赖：

```
<dependency>  
<groupId>com.alipay.sofa</groupId>
```

```
<artifactId>tracer-sofa-boot-starter</artifactId>
</dependency>
```

### 配置 Zipkin 依赖

考虑到 Zipkin 的数据上报能力不是 SOFATracer 默认开启的能力，所以期望使用 SOFATracer 做数据上报时，需要添加如下的 Zipkin 数据汇报的依赖：

```
<dependency>
<groupId>io.zipkin.zipkin2</groupId>
<artifactId>zipkin</artifactId>
<version>2.11.12</version>
</dependency>
<dependency>
<groupId>io.zipkin.reporter2</groupId>
<artifactId>zipkin-reporter</artifactId>
<version>2.7.13</version>
</dependency>
```

### 配置文件

在工程的 application.properties 文件下添加一个 SOFATracer 要使用的参数，包括spring.application.name 用于标示当前应用的名称；logging.path 用于指定日志的输出目录。

```
# Application Name
spring.application.name=SOFATracerReportZipkin
# logging path
logging.path=../logs
com.alipay.sofa.tracer.zipkin.enabled=true
com.alipay.sofa.tracer.zipkin.baseUrl=http://localhost:9411
```

### 启动 Zipkin 服务端

启动 Zipkin 服务端用于接收 SOFATracer 汇报的链路数据，并做展示。Zipkin Server 的搭建可以参考此文档进行配置和服务端的搭建。

### 启动应用

可以将工程导入到 IDE 中运行生成的工程里面中的 main 方法启动应用，也可以直接在该工程的根目录下运行 mvn spring-boot:run，将会在控制台中看到启动日志：

```
2018-05-12 13:12:05.868 INFO 76572 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter:
'SpringMvcSofaTracerFilter' to urls: [/]
2018-05-12 13:12:06.543 INFO 76572 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped"{{/helloZipkin}}"onto public java.util.Map<java.lang.String, java.lang.Object>
com.alipay.sofa.tracer.examples.zipkin.controller.SampleRestController.helloZipkin(java.lang.String)
2018-05-12 13:12:07.164 INFO 76572 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on
port(s): 8080 (http)
```

可以通过在浏览器中输入 <http://localhost:8080/helloZipkin> 来访问 REST 服务，结果类似如下：

```
{  
content:"Hello, SOFATracer Zipkin Remote Report!",  
id: 1,  
success: true  
}
```

### 查看 Zipkin 服务端展示

打开 Zipkin 服务端界面，假设部署的 Zipkin 服务端的地址是 <http://localhost:9411>，打开 URL 并搜索 helloZipkin（由于本地访问的地址是 localhost:8080/helloZipkin），可以看到展示的链路图。

### Spring 工程运行

对于一般的 Spring 工程，通常使用 tomcat/jetty 作为 servlet 容器来启动应用。具体工程，可参考 [在 Spring 工程中使用 SOFATracer](#)。

## 4.15 Tracer 工具类

### 虚拟日志上下文工具类

Tracer 提供 DummyContextUtil 类，对虚拟日志（dummy log）上下文进行操作。

**说明：**使用 DummyContextUtil 创建虚拟日志上下文时，必须及时调用对应的销毁方法进行销毁。建议将对销毁方法的调用写入 finally 方法中。

#### 创建虚拟日志上下文

定义：public static void createDummyLogContext() throws Exception

说明：创建一个虚拟日志上下文，并且将其设置到线程上下文中。如果当前线程上下文中已经有了一个日志上下文，则抛出异常。

#### 清理虚拟日志上下文

定义：public static void clearDummyLogContext() throws Exception

说明：清除当前的虚拟的日志上下文。如果当前线程上下文中的日志上下文不是 DummyLogContext，则抛出异常。

设置全链路压测标记，通过 clearDummyLogContext 类进行调用，如下代码所示：

```
DummyContextUtil.createDummyLogContext();  
DummyContextUtil.addPenetrateAttribute(PenAttrKeyEnum.LOAD_TEST_MARK,"T");  
DummyContextUtil.clearDummyLogContext();
```

### Tracer 上下文工具类

此工具类的主要作用是为业务方提供获取 traceId 的统一方法，另外还提供了获取和设置穿透数据的 API，类

名为 TracerContextUtil，使用方法如下：

获取当前的 Tracer 上下文中 traceId

代码示例：

```
String traceId = TracerContextUtil.getTraceId();
```

说明：

- 如果当前的 Tracer 上下文为空，此方法返回一个空字符串。
- 如果当前的 Tracer 上下文不为空，但是 traceId 为空，则返回一个空字符串。
- 如果当前的 Tracer 上下文不为空，同时 traceId 不为空，则返回 traceId 本身。

**重要**：获取 traceId，前提是存在 TracerContext，而 TracerContext 的创建分为两种情形：

- SOFA MVC 系统：每一次访问时作为入口创建一个 TracerContext，如果想激活 MVC 请求的 Tracer 功能，还需要在 sofa-config 配置文件里配置上 mvc\_toolbox\_plugin=ACTIVE 来激活 MVC 的插件。
- SOFA CORE 系统：作为 RPC 等中间件首次发生调用的时候创建。

**设置穿透数据**

代码示例：

```
TracerContextUtil.putPenetrateAttribute("Hello", "World");
```

说明：

- 如果当前传入的 key 或者 value 为 null，则抛出 TracerException。
- 如果当前的 Tracer 的上下文为空，则抛出 TracerException。
- 当加入该穿透数据，如果整体穿透数据量超过了最大的限制（目前最大的限制为 1024 个字符），则抛出 TracerException。

**关于 key 的说明：**

由于此接口的 key 是全站共享的，所以使用这个 key 的时候，必须保证不和其他 key 冲突，Tracer 中保留关键 key 有 uid 和 mark 两个字段。

**获取 Tracer 中的穿透数据**

代码示例：

```
TracerContextUtil.getPenetrateAttribute("Hello");
```

说明：

- 如果当前的 Tracer 的上下文不为空，并且 Tracer 上下文中根据 key 取到的穿透数据为 null，则返回一个空字符串。
- 如果当前的 Tracer 的上下文不为空，并且 Tracer 上下文中根据 key 取到的穿透数据不为 null，则返回穿透数据。
- 如果 key 为 null，则抛出该异常。
- 如果当前的 Tracer 的上下文为空，则抛出该异常。

#### 克隆当前线程的 Tracer 日志上下文

如果业务系统有自己的线程池来处理一些事情，那么业务开发人员就可能需要获取当前线程的 Tracer 日志上下文，来设置到子线程里面去。在 Tracer 1.0.15 之前的版本，都是直接通过 API 来获取父线程的 Tracer 上下文，然后设置到子线程里面的。这样，父子线程就共享一份 Tracer 日志上下文，有可能出现在一个线程里修改，却影响了另一个线程的情况。

所以在 Tracer 1.0.15 中，提供一个克隆当前线程的 Tracer 日志上下文的 API。如果有需要往子线程中设置的话，可以采用此 API 来从父线程中复制一份，防止出现相互影响的状况，代码示例如下：

```
AbstractLogContext abstractLogContext = TracerContextUtil.cloneLogContext();
```

说明：

- 如果当前线程的 Tracer 日志上下文为空，则返回 null。
- 如果当前线程的 Tracer 日志上下文不为空，则返回此上下文的一份克隆。

#### 全链路工具类

提供静态方法对全链路压测进行判断，类名为：LoadTestUtil。

定义：public static boolean isLoadTestMode()

说明：判断当前是否为全链路压测。如果当前线程上下文中取不到日志上下文，则返回 false。如果能够取到并且压测标记为 T，则返回 true；否则，返回 false。

当全链路压测模式开启时，日志文件会打印在 \${logging.path}/logs/tracelog/shadow/ 目录下，同时在日志文件的透传数据的字段会有mark=T 标识。

## 4.16 Tracer DRM 开关

### 动态配置项

默认情况下，Tracer 会生成各种网络中间件调用的详细日志和统计日志。由于 Tracer 的日志生成是异步的，所以 Tracer 对性能的影响很小。但是，对于特殊需要，Tracer 提供了动态配置开关用于关闭中间件的 digest 日志。

Tracer 的动态配置设置了两个维度的开关：

- 全局开关

- 应用维度开关

## 全局开关

所谓全局开关，就是推送这个微服务，会推送到所有使用了 Tracer 的系统，是全站级别的推送。动态配置的后台已经配置了全局的动态配置。这个动态配置的资源的 ID 是：

```
Alipay.middlewareTracer:name=com.alipay.common.tracer.manage.TracerDrm
```

里面有一个资源属性：

```
disableMiddlewareDigestLog
```

关于这个字段的具体推送值说明如下：

- 推送值为 **true**：关闭所有的 digest 日志。
- 推送值为 **false** 或 空字符串：打开所有的 digest 日志。
- 推送值为 **log1=true&log2=true**：关闭特定的 digest 日志，其中 log1 和 log2 是日志的 key，关闭多个日志可以用 & 符号连接。目前可用的 key 如下：
  - rpc-client : RPCCClient digest 日志
  - rpc-server : RPCServer digest 日志
  - msg-publisher : MSG Publisher digest 日志
  - msg-subscriber : MSG Subscriber digest 日志
  - http-client : HTTPClient digest 日志
  - mvc : MVC digest 日志
  - zdal-db : ZDALDB digest 日志
  - zdal-tair : ZDALTAIR digest 日志

推送示例：

```
// 关闭所有的 digest 日志
true

// 打开所有的 digest 日志
false

// 关闭 zdal db 的 digest 日志
zdal-db=true

// 关闭 zdal db 和 rpc-client 的 digest 日志
zdal-db=true&rpc-client=true
```

若您已经关闭了多个日志，此时想打开某个日志，需要从原来的推送值中把对应的日志去掉，例如：

```
// 已经关闭 zdal db 和 rpc-client 的 digest 日志  
zdal-db=true&rpc-client=true  
  
// 这个时候要打开 rpc-client 日志，应该这样推送  
zdal-db=true
```

## 应用维度开关

应用维度开关的属性和全局开关一样，两者区别在于动态配置的 ID 不同：

```
Alipay.${appname}:name=com.alipay.common.tracer.manage.TracerDrm
```

默认情况下，这个动态配置是没有在动态配置后台配置的。如果需要使用，应用需要自行加上这个配置。操作步骤如下：

在本地 SOFABoot 工程中，引入以下依赖：

```
<dependency>  
<groupId>com.alipay.sofa</groupId>  
<artifactId>ddcs-enterprise-sofa-boot-starter</artifactId>  
<exclusions>  
<exclusion>  
<artifactId>tracer</artifactId>  
<groupId>com.alipay.common</groupId>  
</exclusion>  
</exclusions>  
</dependency>  
  
<dependency>  
<groupId>com.alipay.common</groupId>  
<artifactId>tracer-manage</artifactId>  
</dependency>
```

2. 配置动态配置资源 Bean，示例如下：

```
<bean id="tracerDrm" class="com.alipay.common.tracer.manage.TracerDrm" init-method="init">  
<constructor-arg name="appName" value="tracerDrmBoot"/>  
</bean>
```

3. 前往 **微服务平台 > 微服务 > 动态配置** 控制台页面，添加应用元数据，新增并推送相应的动态配置。详细步骤，可参见 [开始使用动态配置 > 云端管控动态配置类](#)。

## 4.17 Tracer 日志配置项

应用在引入 SOFATracer 后，可以在 Spring Boot 的配置文件 application.properties 中添加相关配置项来定制 SOFATracer 的相关行为。

SOFATracer 的日志输出目录，可以在 application.properties 中配置 logging.path 的路径，那么其日志输出路径

为 \${logging.path}/tracelog ; 如果没有配置 logging.path , 那么 SOFATracer 的默认输出路径为 \${user.home}/logs/tracelog。

#### SpringBoot 工程配置

SOFATracer 配置项	说明	默认值
logging.path	日志输出目录	SOFATracer 会优先输出到 logging.path 目录下 ; 如果没有配置日志输出目录 , 那默认输出到 \${user.home}
com.alipay.sofa.tracer.disableDigestLog	是否关闭所有集成 SOFATracer 组件摘要日志打印	false
com.alipay.sofa.tracer.disableConfiguration[\$logType]	关闭指定 \${logType} 的 SOFATracer 组件摘要日志打印。 \${logType} 是指具体的日志类型 , 如 : spring-mvc-digest.log	false
com.alipay.sofa.tracer.tracerGlobalRollingPolicy	SOFATracer 日志的滚动策略	.yyyy-MM-dd : 按照天滚动 ; .yyyy-MM-dd_HH : 按照小时滚动。 默认不配置按照天滚动
com.alipay.sofa.tracer.tracerGlobalLogReserveDay	SOFATracer 日志的保留天数	默认保留 7 天
com.alipay.sofa.tracer.statLogInterval	统计日志的时间间隔 , 单位 : 秒	默认 60 秒统计日志输出一次
com.alipay.sofa.tracer.baggageMaxLength	透传数据能够允许存放的最大长度	默认值 1024
com.alipay.sofa.tracer.zipkin.enabled	是否开启 SOFATracer 远程上报数据到 Zipkin	true : 开启上报 ; false : 关闭上报。 默认不上报
com.alipay.sofa.tracer.zipkin.baseUrl	SOFATracer 远程上报数据到 Zipkin 的地址 , com.alipay.sofa.tracer.zipkin.enabled=true 时配置此地址才有意义	格式 : http://\${host}:\${port}
com.alipay.sofa.tracer.springmvc.filterOrder	SOFATracer 集成在 SpringMVC 的 Filter 生效的 Order	-2147483647 ( org.springframework.core.Ordered#HIGHEST_PRECEDENCE + 1 )
com.alipay.sofa.tracer.springmvc.urlPatterns	SOFATracer 集成在 SpringMVC 的 Filter 生效的 URL Pattern 路径	/* 全部生效
com.alipay.sofa.tracer.jsonOutput	是否以 json 格式输出日志	true , 如果期望较少日志空间占用 , 可以使用非 json 格式输出 ( 日志顺序与 JSON 格式顺序一致 )

#### 非 SpringBoot 工程配置

在非 SpringBoot 工程中 , 可以通过在 classpath 下新建一个 sofa.tracer.properties 配置文件 , 配置项如下 :

SOFATracer 配置项	说明	默认值
disable_middleware_digest_log	是否关闭中间件组件摘要日志打印	false
disable_digest_log	关闭摘要日志打印。	false
tracer_global_rolling_policy	SOFATracer 日志的滚动策略	.yyyy-MM-dd : 按照天滚动 ; .yyyy-MM-dd_HH : 按照小时滚动。 默认不配置按照天滚动

tracer_global_log_reserve_day	SOFATracer 日志的保留天数	默认保留 7 天
stat_log_interval	统计日志的时间间隔，单位：秒	默认 60 秒统计日志输出一次
tracer_penetrate_attribute_max_length	透传数据能够允许存放的最大长度	默认值 1024
tracer_async_appender_allow_discard	是否允许丢失日志	false
tracer_async_appender_is_out_discard_number	丢失日志数	0
spring.application.name	应用名	-
tracer_sampler_strategy_name_key	采样策略名	-
tracer_sampler_strategy_custom_rule_class_name	采样规则 spi 实现的类的全限定名	-
tracer_sampler_strategy_percentage_key	采样比率	-
com.alipay.sofa.tracer.jsonOutput	是否以json格式输出日志	true，如果期望较少日志空间占用，可以使用非 json 格式输出（日志顺序与 JSON 格式顺序一致）

## 4.18 日志格式

SOFATracer 支持 Spring MVC、标准 JDBC 接口实现的数据库连接池（DBCP、Druid、c3p0、tomcat、HikariCP、BoneCP）、HttpClient、Dubbo、Spring Cloud OpenFeign 等开源组件。组件埋点接入后，即可查看相关 Tracer 日志。本文将介绍如下几种日志及其日志格式。

- Spring MVC 日志
- HttpClient 日志
- DataSource 日志
- SOFARPC 日志
- OkHttp 日志
- RestTemplate 日志
- Dubbo 日志
- Spring Cloud OpenFeign 日志
- RPC 转 JVM 日志
- SOFARest 日志
- SOFA MVC 日志
- 异常日志
- 静态信息日志

### Spring MVC 日志

SOFATracer 集成 SpringMVC 后输出 MVC 请求的链路数据格式，默认为 JSON 数据格式。

**Spring MVC 摘要日志 ( spring-mvc-digest.log )**

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
request.url	请求地址
method	http method
req.size.bytes	请求大小
resp.size.bytes	响应大小
sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time":"2019-09-03
10:33:10.336","local.app":"RestTemplateDemo","traceId":"0a0fe9271567477985327100211176","spanId":"0.1","span.kind":"server","result.code":"200","current.thread.name":"http-nio-8801-exec-2","time.cost.milliseconds":"5006ms","request.url":"http://localhost:8801/asyncrest","method":"GET","req.size.bytes": -1,"resp.size.bytes":0,"sys.baggage":"","biz.baggage":""}
```

**Spring MVC 统计日志 ( spring-mvc-stat.log )**

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含 local.app、request.url、和 method 字段。

key		表达含义
time		日志打印时间
stat.key	local.ap p	当前应用名
	request. url	请求 URL
	method	请求 HTTP 方法
count		本段时间内请求次数
total.cost.milliseconds		本段时间内的请求总耗时 ( ms )

success	请求结果：Y 表示成功(1 开头和 2 开头的结果码算是成功的，302 表示的重定向算成功，其他算是失败的)；N 表示失败
load.test	压测标记：T 是压测；F 不是压测

样例：

```
{"time": "2019-09-03
10:34:04.129", "stat.key": {"method": "GET", "local.app": "RestTemplateDemo", "request.url": "http://localhost:8801/async
rest"}, "count": 1, "total.cost.milliseconds": 5006, "success": "true", "load.test": "F"}
```

## HttpClient 日志

SOFATracer 集成 sofa-tracer-httpclient-plugin 插件后输出 HttpClient 请求的链路数据，默认为 JSON 数据格式。

### HttpClient 摘要日志 ( httpclient-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
request.url	请求地址
method	http method
req.size.bytes	请求大小
resp.size.bytes	响应大小
sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time": "2019-09-02
23:43:13.191", "local.app": "HttpClientDemo", "traceId": "1e27a79c1567438993170100210107", "spanId": "0", "span.kind": "client", "result.code": "200", "current.thread.name": "I/O dispatcher
1", "time.cost.milliseconds": "21ms", "request.url": "http://localhost:8080/httpclient", "method": "GET", "req.size.bytes": 0, "resp.size.bytes": -1, "remote.app": "", "sys.baggage": "", "biz.baggage": ""}
```

备注：应用名称可以通过 SofaTracerHttpClientBuilder 构造 HttpClient 实例时以入参的形式传入。

### HttpClient 统计日志 ( httpclient-stat.log )

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含local.app、request.url、和 method 字段。

key		表达含义
time		日志打印时间
stat.key	local.app	当前应用名
	request.url	请求 URL
	method	请求 HTTP 方法
count		本段时间内请求次数
total.cost.milliseconds		本段时间内的请求总耗时 ( ms )
success		请求结果：Y 表示成功(1 开头和 2 开头的结果码算是成功的，302 表示的重定向算成功，其他算是失败的)；N 表示失败
load.test		压测标记：T 是压测；F 不是压测

样例：

```
{"time":"2019-09-02 23:44:11.785","stat.key":{"method":"GET","local.app":"HttpClientDemo","request.url":"http://localhost:8080/httpclient"}, "count":2,"total.cost.milliseconds":229,"success":"true","load.test":"F"}
```

## DataSource 日志

SOFATracer 对标准的 JDBC 数据源进行埋点，输出 SQL 语句执行链路数据，默认日志输出为 JSON 数据格式。

### DataSource 摘要日志 ( datasource-client-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
database.name	数据库名称
sql	sql 执行语句
connection.establish.span	sql 执行建连时间

db.execute.cost	sql执行时间
database.type	数据库类型
database.endpoint	数据库url
sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time":"2019-09-02
21:31:31.566","local.app":"SOFATracerDataSource","traceId":"0a0fe91d156743109138810017302","spanId":"0.1","span.kind":"client","result.code":"00","current.thread.name":"http-nio-8080-exec-1","time.cost.milliseconds":"15ms","database.name":"test","sql":"DROP TABLE IF EXISTS TEST;
CREATE TABLE TEST(ID INT PRIMARY KEY%2C NAME
VARCHAR(255));","connection.establish.span":"128ms","db.execute.cost":"15ms","database.type":"h2","database.endpoint":"jdbc:h2:~/test:-1","sys.baggage":"","biz.baggage":""}
```

#### DataSource 统计日志 ( datasource-client-stat.log )

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含 local.app、database.name、和 sql 字段。

key	表达含义	
time	日志打印时间	
stat.key	local.app	当前应用名
	database.name	数据库名称
	sql	sql执行语句
count	本段时间内sql执行次数	
total.cost.milliseconds	本段时间内sql执行总耗时 ( ms )	
success	请求结果：Y 表示成功；N 表示失败	
load.test	压测标记：T 是压测；F 不是压测	

样例：

```
{"time":"2019-09-02
21:31:50.435","stat.key":{"local.app":"SOFATracerDataSource","database.name":"test","sql":"DROP TABLE IF EXISTS TEST;
CREATE TABLE TEST(ID INT PRIMARY KEY%2C NAME
VARCHAR(255))"},"count":1,"total.cost.milliseconds":15,"success":"true","load.test":"F"}
```

#### SOFARPC 日志

SOFATracer 集成在 SOFARPC ( 5.4.0 及之后的版本 ) 后输出链路数据的格式，默认为 JSON 数据格式，具体的字段含义解释如下：

#### RPC 客户端摘要日志 ( rpc-client-digest.log )

key	表达含义
timestamp	日志打印时间
tracerId	TraceId
spanId	SpanId
span.kind	Span 类型
local.app	当前 appName
protocol	协议 ( bolt/rest )
service	服务接口信息
method	方法名
current.thread.name	当前线程名
invoke.type	调用类型 ( sync/callback/oneway/future )
router.record	路由记录 ( DIRECT/REGISTRY )
remote.ip	目标 ip
remote.app	目标 appName
local.client.ip	本机 ip
result.code	返回码 ( 00=成功/01=业务异常/02=RPC逻辑错误/03=超时失败/04=路由失败 )
req.serialize.time	请求序列化时间 ( 单位 ms )
resp.deserialize.time	响应反序列化时间 ( 单位 ms )
resp.size	响应大小 ( 单位 Byte )
req.size	请求大小 ( 单位 Byte )
client.conn.time	客户端连接耗时 ( 单位 ms )
client.elapse.time	调用总耗时 ( 单位 ms )
local.client.port	本地客户端端口
baggage	透传的 baggage 数据 ( kv 格式 )

样例：

```
{"timestamp": "2018-05-20
17:03:20.708", "tracerId": "1e27326d1526807000498100185597", "spanId": "0", "span.kind": "client", "local.app": "SOFATracerRPC", "protocol": "bolt", "service": "com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0", "method": "sayDirect", "current.thread.name": "main", "invoke.type": "sync", "router.record": "DIRECT", "remote.app": "samples", "remote.ip": "127.0.0.1:12200", "local.client.ip": "127.0.0.1", "result.code": "00", "req.serialize.time": "33", "resp.deserialize.time": "39", "resp.size": "170", "req.size": "582", "client.conn.time": "0", "client.elapse.time": "155", "local.client.port": "59774", "baggage": ""}
```

RPC 服务端摘要日志 ( rpc-server-digest.log )

key	表达含义
timestamp	日志打印时间
tracerId	TraceId

spanId	SpanId
span.kind	Span 类型
service	服务接口信息
method	方法名
remote.ip	来源 ip
remote.app	来源 appName
protocol	协议 ( bolt/rest )
local.app	当前 appName
current.thread.name	当前线程名
result.code	返回码 ( 00=成功/01=业务异常/02=RPC逻辑错误 )
server.pool.wait.time	服务端线程池等待时间 ( 单位 ms )
biz.impl.time	业务处理耗时 ( 单位 ms )
resp.serialize.time	响应序列化时间 ( 单位 ms )
req.deserialize.time	请求反序列化时间 ( 单位 ms )
resp.size	响应大小 ( 单位 Byte )
req.size	请求大小 ( 单位 Byte )
baggage	透传的 baggage 数据 ( kv 格式 )

样例：

```
{"timestamp": "2018-05-20
17:00:53.312", "tracerId": "1e27326d1526806853032100185011", "spanId": "0", "span.kind": "server", "service": "com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0", "method": "sayDirect", "remote.ip": "127.0.0.1", "remote.app": "SOFATracerRPC", "protocol": "bolt", "local.app": "SOFATracerRPC", "current.thread.name": "SOFA-BOLT-BIZ-12200-5-T1", "result.code": "00", "server.pool.wait.time": "3", "biz.impl.time": "0", "resp.serialize.time": "4", "req.deserialize.time": "38", "resp.size": "170", "req.size": "582", "baggage": ""}
```

RPC 客户端统计日志 ( rpc-client-stat.log )

key	表达含义
time	日志打印时间
stat.key	日志关键 key
method	方法信息
local.app	客户端 appName
service	服务接口信息
count	调用次数
total.cost.milliseconds	总耗时 ( 单位 ms )
success	调用结果 ( Y/N )

样例：

```
{"time": "2018-05-18
07:02:19.717", "stat.key": {"method": "method", "local.app": "client", "service": "app.service:1.0"}, "count": 10, "total.cost.milliseconds": 17, "success": "Y"}
```

#### RPC 服务端 统计日志 ( rpc-server-stat.log )

key	表达含义
time	日志打印时间
stat.key	日志关键 key
method	方法信息
local.app	客户端 appName
service	服务接口信息
count	调用次数
total.cost.milliseconds	总耗时 ( 单位 ms )
success	调用结果 ( Y/N )

样例：

```
{"time": "2018-05-18
07:02:19.717", "stat.key": {"method": "method", "local.app": "client", "service": "app.service:1.0"}, "count": 10, "total.cost.milliseconds": 17, "success": "Y"}
```

#### OkHttp 日志

SOFATracer 集成 OkHttp 后输出请求的链路数据格式，默认为 JSON 数据格式。

#### OkHttp 摘要日志 ( okhttp-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
request.url	请求 URL
method	请求 HTTP 方法
result.code	HTTP 返回状态码
req.size.bytes	Request Body 大小
resp.size.bytes	Response Body 大小
time.cost.milliseconds	请求耗时 ( ms )
current.thread.name	当前线程名

remote.app	目标应用
baggage	透传的 baggage 数据

样例：

```
{"time": "2019-09-03
11:35:28.429", "local.app": "OkHttpDemo", "traceId": "0a0fe9271567481728265100112783", "spanId": "0", "span.kind": "client", "result.code": "200", "current.thread.name": "main", "time.cost.milliseconds": "164ms", "request.url": "http://localhost:8081/okhttp?name=sofa", "method": "GET", "result.code": "200", "req.size.bytes": 0, "resp.size.bytes": 0, "remote.app": "", "sys.baggage": "", "biz.baggage": ""}
```

#### OkHttp 统计日志 ( okhttp-stat.log )

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含 local.app、request.url、和 method 字段。

key	表达含义		
time	日志打印时间		
stat.key	local.app	当前应用名	
	request.url	请求 URL	
	method	请求 HTTP 方法	
count	本段时间内请求次数		
total.cost.milliseconds	本段时间内的请求总耗时 ( ms )		
success	请求结果：Y 表示成功；N 表示失败		
load.test	压测标记：T 是压测；F 不是压测		

样例：

```
{"time": "2019-09-03
11:43:06.975", "stat.key": {"method": "GET", "local.app": "OkHttpDemo", "request.url": "http://localhost:8081/okhttp?name=sofa"}, "count": 1, "total.cost.milliseconds": 174, "success": "true", "load.test": "F"}
```

#### RestTemplate 日志

SOFATracer 集成 RestTemplate 后输出请求的链路数据格式，默认为 JSON 数据格式。

#### RestTemplate 摘要日志 ( resttemplate-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId

span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
request.url	请求地址
method	http method
req.size.bytes	请求大小
resp.size.bytes	响应大小
sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time":"2019-09-03
10:33:10.336","local.app":"RestTemplateDemo","traceId":"0a0fe9271567477985327100211176","spanId":"0","span.kind":"client","result.code":"200","current.thread.name":"SimpleAsyncTaskExecutor-1","time.cost.milliseconds":"5009ms","request.url":"http://localhost:8801/asyncrest","method":"GET","req.size.bytes":0,"resp.size.bytes":0,"sys.baggage":"","biz.baggage":""}
```

#### RestTemplate 统计日志 ( resttemplate-stat.log )

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含 local.app、request.url、和 method 字段。

key	表达含义	
time	日志打印时间	
stat.key	local.app	当前应用名
	request.url	请求 URL
	method	请求 HTTP 方法
count	本段时间内请求次数	
total.cost.milliseconds	本段时间内的请求总耗时 ( ms )	
success	请求结果：Y 表示成功；N 表示失败	
load.test	压测标记：T 是压测；F 不是压测	

样例：

```
{"time":"2019-09-03
10:34:04.130","stat.key":{"method":"GET","local.app":"RestTemplateDemo","request.url":"http://localhost:8801/asyncrest"},"count":1,"total.cost.milliseconds":5009,"success":true,"load.test":F}
```

#### Dubbo 日志

SOFATracer 集成 Dubbo 后输出请求的链路数据格式，默认为 JSON 数据格式。

## Dubbo 服务消费方摘要日志 ( dubbo-client-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
protocol	协议
service	服务接口
method	调用方法
invoke.type	调用类型
remote.host	目标主机
remote.port	目标端口
local.host	本地主机
client.serialize.time	请求序列化时间
client.deserialize.time	响应反序列化时间
req.size.bytes	Request Body 大小
resp.size.bytes	Response Body 大小
error	错误信息
sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time": "2019-09-02 23:36:08.250", "local.app": "dubbo-consumer", "traceId": "1e27a79c156743856804410019644", "spanId": "0", "span.kind": "client", "result.code": "00", "current.thread.name": "http-nio-8080-exec-2", "time.cost.milliseconds": "205ms", "protocol": "dubbo", "service": "com.glmapper.bridge.boot.service.HelloService", "method": "SayHello", "invoke.type": "sync", "remote.host": "192.168.2.103", "remote.port": "20880", "local.host": "192.168.2.103", "client.serialize.time": 35, "client.deserialize.time": 5, "req.size.bytes": 336, "resp.size.bytes": 48, "error": "", "sys.baggage": "", "biz.baggage": ""}
```

## Dubbo 服务提供方摘要日志 ( dubbo-server-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
protocol	协议
service	服务接口
method	调用方法
invoke.type	调用类型
local.host	本地主机
local.port	本地端口
server.serialize.time	响应序列化时间
server.deserialize.time	请求反序列化时间
req.size.bytes	Request Body 大小
resp.size.bytes	Response Body 大小
error	错误信息
sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time": "2019-09-02 23:36:08.219", "local.app": "dubbo-provider", "traceId": "1e27a79c156743856804410019644", "spanId": "0", "span.kind": "server", "result.code": "00", "current.thread.name": "DubboServerHandler-192.168.2.103:20880-thread-2", "time.cost.milliseconds": "9ms", "protocol": "dubbo", "service": "com.glmapper.bridge.boot.service.HelloService", "method": "SayHello", "local.host": "192.168.2.103", "local.port": "62443", "server.serialize.time": 0, "server.deserialize.time": 27, "req.size.bytes": 336, "resp.size.bytes": 0, "error": "", "sys.baggage": "", "biz.baggage": ""}
```

#### Dubbo 统计日志

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含 local.app、service、和 method 字段。

key	表达含义	
time	日志打印时间	
stat.key	local.app	当前应用名
	method	调用方法

	service	服务名
count		本段时间内请求次数
total.cost.milliseconds		本段时间内的请求总耗时 ( ms )
success		请求结果 : Y 表示成功 ; N 表示失败
load.test		压测标记 : T 是压测 ; F 不是压测

样例 :

- dubbo-client-stat.log

```
{"time":"2019-09-02 23:36:13.040","stat.key":{"method":"SayHello","local.app":"dubbo-consumer","service":"com.glmapper.bridge.boot.service.HelloService"},"count":1,"total.cost.milliseconds":205,"success":true,"load.test":F}
```

- dubbo-server-stat.log

```
{"time":"2019-09-02 23:36:13.208","stat.key":{"method":"SayHello","local.app":"dubbo-provider","service":"com.glmapper.bridge.boot.service.HelloService"},"count":1,"total.cost.milliseconds":9,"success":true,"load.test":F}
```

## Spring Cloud OpenFeign 日志

SOFATracer 集成 Spring Cloud OpenFeign 后输出请求的链路数据格式，默认为 JSON 数据格式。

### Spring Cloud OpenFeign 摘要日志 ( feign-digest.log )

以 JSON 格式输出的数据，相应 key 的含义解释如下：

key	表达含义
time	日志打印时间
local.app	当前应用名
traceId	TraceId
spanId	SpanId
span.kind	Span 类型
result.code	状态码
current.thread.name	当前线程名
time.cost.milliseconds	span 耗时
request.url	请求地址
method	http method
error	错误信息
req.size.bytes	请求大小
resp.size.bytes	响应大小

sys.baggage	系统透传的 baggage 数据
biz.baggage	业务透传的 baggage 数据

样例：

```
{"time":"2019-09-03 10:28:52.363","local.app":"tracer-consumer","traceId":"0a0fe9271567477731347100110969","spanId":"0.1","span.kind":"client","result.code":"200","current.thread.name":"http-nio-8082-exec-1","time.cost.milliseconds":219,"request.url":"http://10.15.233.39:8800/feign","method":"GET","error":"","req.size.bytes":0,"resp.size.bytes":18,"remote.host":"10.15.233.39","remote.port":8800,"sys.baggage":{},"biz.baggage":{}}
```

#### Spring Cloud OpenFeign 统计日志 ( feign-stat.log )

stat.key 即本段时间内的统计关键字集合，统一关键字集合唯一确定一组统计数据，包含 local.app、request.url、和 method 字段。

key	表达含义		
time	日志打印时间		
stat.key	local.app	当前应用名	
	request.url	请求 URL	
	method	请求 HTTP 方法	
count	本段时间内请求次数		
total.cost.milliseconds	本段时间内的请求总耗时 ( ms )		
success	请求结果：Y 表示成功；N 表示失败		
load.test	压测标记：T 是压测；F 不是压测		

样例：

```
{"time":"2019-09-03 10:29:34.528","stat.key":{"method":"GET","local.app":"tracer-consumer","request.url":"http://10.15.233.39:8800/feign"},"count":2,"total.cost.milliseconds":378,"success":true,"load.test":F}
```

#### RPC 转 JVM 日志

##### RPC 转 JVM 详细日志 ( rpc-2-jvm-digest.log )

在 1.0.16 的 Tracer 版本中，Tracer 增加了合并部署的 RPC 转 JVM 的详细日志，日志格式如下：

- 日志打印时间
- 当前应用名
- TraceId
- RpcId
- 服务名
- 方法名
- 目标系统名

- 调用耗时
- 当前线程名
- 系统穿透数据 ( kv 格式 , 用于传送系统灾备信息等 )
- 穿透数据 ( kv 格式 )

#### 样例

```
2015-04-27 17:51:47.711,test,0a0f61eb14301283076901001,0,com.alipay.SampleService,hello,testTarget,21ms,main,
```

注意 : 默认的情况下 , RPC 转 JVM 的详细日志是关闭的 , 需要通过 DRM 推送来打开 , 详见 Tracer 的 DRM 说明。

#### RPC 转JVM调用统计日志 ( 每一分钟打印一次 ) ( rpc-2-jvm-stat.log )

- 日志打印时间
- fromApp ( 即 currentApp )
- toApp
- 服务名
- 方法名
- 本段时间内调用次数
- 本段时间内的调用总耗时
- 结果 ( Y/N , rpc-2-jvm 没有统计结果 , 所有的都是 Y )
- 全链路压测标志 ( T/F )

#### 样例 :

```
2014-05-21 19:18:52.484 from,to,DummyService,dummyMethod,596,60041,Y,T
```

#### SOFARest 日志

#### SOFAREST 日志

每一次 SOFAREST 调用会生成相应的客户端和服务端执行日志 , 日志格式如下。

#### SOFAREST 客户端日志

SOFAREST 客户端日志名为 rest-client-digest.log , 包含以下信息 :

字段	描述
时间戳	日志打印时间
应用名	当前应用名称
TraceId	25 位的 TraceId

RpcId	调用链路的 RpcId
URL	请求 URL
方法名	方法名称
目标应用	目标应用名称
目标地址	目标机器 IP 地址
结果码	返回结果码
请求大小	请求发送的数据大小，以 byte 为单位
响应大小	响应请求数据大小，以 byte 为单位
线程名	当前线程名
系统穿透数据	以 key-value 格式记录，用于传送系统灾备信息等
穿透数据	以 key-value 格式记录

**样例：**

```
2015-11-04
11:00:26.556,test,0a0fe9cf14466060265051002,0,http://localhost,insert,anotherApp,10.20.30.40,100,100B,100B,49ms
,main,test=test&
```

#### SOFAREST 服务端日志

SOFAREST 服务端日志名为 rest-server-digest.log，包含以下信息：

字段	描述
时间戳	日志打印时间
应用名	当前应用名称
TraceId	25 位的 TraceId
RpcId	调用链路的 RpcId
URL	请求 URL
方法名	方法名称
来源应用	来源应用名称
来源地址	来源应用 IP 地址
结果码	返回结果码
请求大小	请求发送的数据大小，以 byte 为单位
响应大小	响应请求数据大小，以 byte 为单位
线程名	当前线程名
系统穿透数据	以 key-value 格式记录，用于传送系统灾备信息等
穿透数据	以 key-value 格式记录

**样例：**

2015-11-04  
11:00:24.475,test,0a0fe9cf14466060243451001,0,http://localhost,insert,fromApp,fromAddress,400,100B,100B,128ms,  
main,test=test&

**SOFAREST 客户端统计日志**

SOFAREST 客户端统计日志名为 rest-client-stat.log , 每分钟打印一次 , 该日志包含以下信息 :

字段	描述
时间戳	日志打印时间
来源应用名	fromApp , 即当前应用的名称
目标应用名	toApp , 即目标应用的名称
URL	请求 URL
方法	方法名称
被调用次数	本段时间内的被调用次数
请求处理总耗时	本段时间内的请求处理总耗时
结果	<ul style="list-style-type: none"> <li>• Y : 成功</li> <li>• N : 失败</li> </ul>
全链路压测标志	<ul style="list-style-type: none"> <li>• T : True , 表示当前线程中能获取到日志上下文。</li> <li>• F : False , 表示当前线程中获取不到日志上下文。</li> </ul>

**样例 :**

2015-11-04 11:00:28.445,test,anotherApp,http://localhost,insert,1,49,Y,F

**SOFAREST 服务端统计日志**

SOFAREST 服务端统计日志名为 rest-server-stat.log , 每分钟打印一次 , 该日志包含以下信息 :

字段	描述
时间戳	日志打印时间
来源应用名	fromApp , 即当前应用的名称
目标应用名	toApp , 即目标应用的名称
URL	请求 URL
方法	方法名称
被调用次数	本段时间内的被调用次数
请求处理总耗时	本段时间内的请求处理总耗时
结果	<ul style="list-style-type: none"> <li>• Y : 成功</li> <li>• N : 失败</li> </ul>

全链路压测标志	<ul style="list-style-type: none"><li>T : True , 表示当前线程中能获取到日志上下文。</li><li>F : False , 表示当前线程中获取不到日志上下文。</li></ul>
---------	--

样例：

```
2015-11-04 11:00:26.445,fromApp,test,http://localhost,insert,1,128,N,F
```

## SOFA MVC 日志

SOFA MVC 摘要日志 ( sofa-mvc-digest.log )

- 日志打印时间
- 当前应用名
- TraceId
- RpcId
- 请求 URL
- 请求方法
- Http 状态码
- Request Body 大小
- Response Body 大小
- 请求耗时 ( MS )
- 当前线程名
- 穿透数据 (kv格式)

样例：

```
2014-09-01  
00:00:01.631,tbapi,0ad643e114095008015728852,0,http://tbapi.alipay.com/gateway.do,POST,200,1468B,2161B,59m  
s,catalina-exec-71,uid=13&mark=F&
```

SOFA MVC 统计日志 ( sofa-mvc-stat.log )

- 日志打印时间
- 当前应用名
- 请求 URL
- 请求方法
- 本段时间内请求次数
- 本段时间内的请求总耗时
- 请求结果(1 开头和 2 开头的结果码算是成功的，302表示的重定向算成功，其他算是失败的)

- 压测标记

样例：

```
2014-09-01 00:03:22.559,tbapi,http://tbapi.alipay.com/trade/batch_payment.htm,GET,2,11,Y,F
```

## 异常日志

默认情况下，异常日志都记录在 middleware\_error.log 文件中。

### 通用异常日志基本格式

以下为异常日志记录的常见信息：

字段	描述
时间戳	日志生成时间 ( UTC+08:00 )
TraceId	25 位的 TraceId。前 8 位表示服务器 IP，而后 13 位为生成 TraceId 的时间，后 4 位为自增序列。
RpcId	调用链路的 RpcId
故障类型	根据实际情况，显示相应故障类型
故障源	以数据形式表示故障源： 故障源1 故障源2 故障源3 ..... 依次类推
故障上下文	以映射形式表示故障上下文： fromUser=khotyn&toUser=nytohk&
异常 Stack	具体异常类型

样例：

```
2015-02-10
22:47:20.499,trade,q241234,0.1,timeout_error,trade|RPC,&protocol=&targetApp=&paramTypes=int|String&invokeType=&methodName=refund&targetUrl=&serviceName=RefundFacade&.,java.lang.Throwable
at com.alipay.common.tracer.CommonTracerTest.testMiddleware(CommonTracerTest.java:43)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)
at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)
at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
```

### 中间件异常日志

中间件产品的异常日志格式与通用异常日志格式一致，唯一的区别是故障源的第一项为故障发生的集群，如下所示：

故障发生的集群 | 其他故障源1 | 其他故障源2

#### RPC 客户端异常日志

RPC 客户端的异常日志直接在中间件异常日志 ( middleware\_error.log ) 中生成，其基本格式和中间件异常日志的格式一致，主要有以下区别：

- 故障源的第二项为 RPC : 故障发生的集群|RPC|其他故障源1|其他故障源2
- 故障类型为以下几种：
  - biz\_error : 业务异常
  - address\_route\_error : 地址路由异常
  - timeout\_error : 超时异常
  - unknown\_error : 未知异常
- 故障上下文中有下列这些信息：
  - serviveName : 服务名
  - methodName : 方法名
  - protocol : 协议
  - invokeType : 调用类型
  - targetUrl : 目标 URL
  - paramTypes : 请求参数类型 ( value 的格式为 param1|param2 )

样例：

```
2015-02-10
22:47:20.499,trade,q241234,0.1,timeout_error,trade|RPC,&protocol=&targetApp=&paramTypes=int|String&invokeType=&methodName=refund&targetUrl=&serviceName=RefundFacade&,,java.lang.Throwable
at com.alipay.common.tracer.CommonTracerTest.testMiddleware(CommonTracerTest.java:43)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)
at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)
at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
```

#### RPC 服务端异常日志

RPC 服务端的异常日志直接在中间件异常日志 ( middleware\_error.log ) 中生成，其基本格式和中间件异常日志的格式一致，主要有以下区别：

- 故障源的第二项为 RPC : 故障发生的集群|RPC|其他故障源1|其他故障源2
- 故障类型为以下几种：
  - biz\_error : 业务异常
  - unknown\_error : 未知异常
- 故障上下文中有下列这些信息：
  - serviveName : 服务名
  - methodName : 方法名
  - protocol : 协议
  - invokeType : 调用类型
  - callerUrl : 调用方 URL
  - callerApp : 调用方应用名
  - paramTypes : 请求参数类型 ( value 的格式为 param1|param2 )

样例：

```
2015-02-10
22:47:20.505,trade,q241234,0.1,unknown_error,trade|RPC,protocol=&callerUrl=&paramTypes=int|String&callerApp=&invokeType=&methodName=refund&serviceName=RefundFacade&.,java.lang.Throwable
at com.alipay.common.tracer.CommonTracerTest.testMiddleware(CommonTracerTest.java:45)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
```

#### 静态信息日志

Tracer 静态信息日志 ( static-info.log ) 会在启动的时候将当期进程的一些静态信息打印出来，目前日志格式如下：

- 进程 ID
- 当前 IP
- 当前 Zone

样例：

```
84919,10.15.233.110,GZ001
```

## 5 常见问题

本文汇总了分布式链路跟踪服务（DST）在使用过程中的一些常见问题及对应的解决方案。

- 应用正常运行却无法在控制台被展示
  - 多维查询没有结果，或者搜索链路为空
  - 如何配置 tracer，以便按照小时分割 trace 日志？
  - RPC Tracer 打印不出 rpc-client-digest.log
- 

**应用正常运行却无法在控制台被展示**

**现象**

**如题**

**原因**

分布式链路跟踪服务底层依赖于蚂蚁金服金融科技的日志服务进行应用日志信息的收集，金融科技日志服务未开通或未按照日志采集客户端均可导致跟踪信息无法显示。

**解决方法**

确认您已经完成以下操作步骤：

1. 在应用服务中开启日志服务产品，参见 [日志服务开通流程](#)。
  2. 安装日志采集客户端 Logtail 进行日志采集，安装流程参见 [快速开始](#)。
- 

**多维查询没有结果，或者搜索链路为空**

**现象**

**如题**

**原因**

要查询的链路调用时间超过有效时间（缓存 7 天）或日志采集未正确配置。

**解决方案**

进行以下检查：

- 链路调用时间是否在有效时间内。分布式链路跟踪系统会对日志作 7 天缓存，系统无法查询到超出缓存时间的链路数据。
  - 日志服务是否开启及日志采集客户端是否正常安装，参见问题 [应用正常运行却无法在控制台被展示](#)
- 

**如何配置 tracer，以便按照小时分割 trace 日志？**

**需求背景**

- 在默认设置中，中间件的 tracelog 是按天来分割的。每 24 小时生成一个新的日志。
- 客户想要在 SOFABoot 商业版中对 tracelog 每小时分割一次。

## 解决方案

在 pom.xml 中添加 tracer 商业版的 jar 包。

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

在 application.properties 中添加配置：com.alipay.sofa.tracerGlobalRollingPolicy=yyyy-MM-dd\_HH  
。

## RPC Tracer 打印不出 rpc-client-digest.log

### 现象

RPC Tracer 打印不出 rpc-client-digest.log，但是能打印 rpc-client-stat.log，而且，实际上已经调用了 RPC 服务。

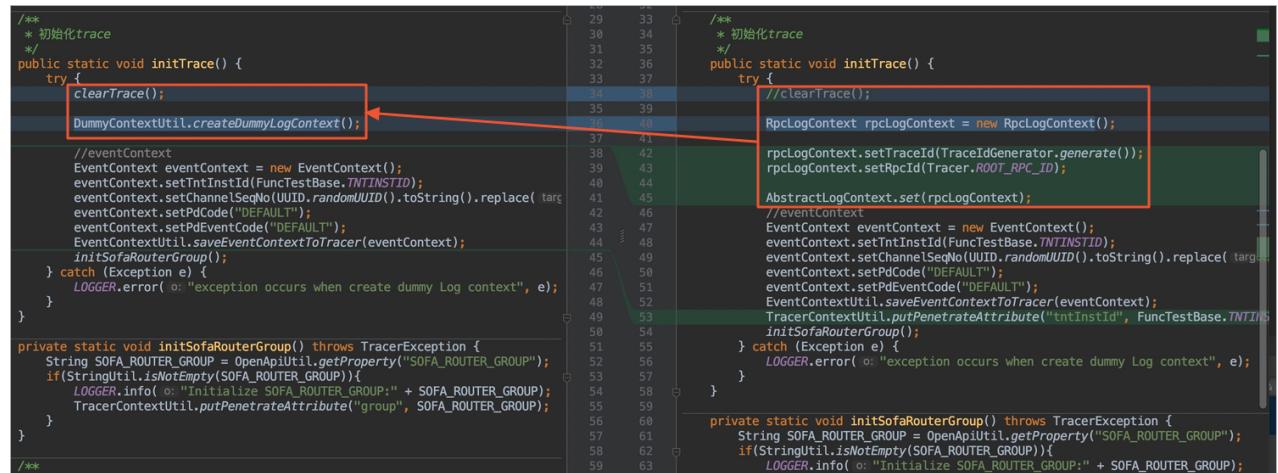
### 原因

客户的代码中手动进行了日志初始化设置，其中 isSampled = false。

## 解决方案

SOFABoot 框架会自动打印日志，不需要手动创建，因此需要修改代码。

示例如下：



```
/** * 初始化trace */
public static void initTrace() {
    try {
        clearTrace();
        DummyContextUtil.createDummyLogContext(); // This line is highlighted with a red box and an arrow pointing to its equivalent in the right version.
        //eventContext
        EventContext eventContext = new EventContext();
        eventContext.setTntInstId(FuncTestBase.TNTINSTID);
        eventContext.setChannelSeqNo(UUID.randomUUID().toString().replace( targ...
        eventContext.setPdCode("DEFAULT");
        eventContext.setPdEventCode("DEFAULT");
        EventContextUtil.saveEventContextToTracer(eventContext);
        initSofaRouterGroup();
    } catch (Exception e) {
        LOGGER.error(o: "exception occurs when create dummy Log context", e);
    }
}

private static void initSofaRouterGroup() throws TracerException {
    String SOFA_ROUTER_GROUP = OpenApiUtil.getProperty("SOFA_ROUTER_GROUP");
    if(StringUtil.isNotEmpty(SOFA_ROUTER_GROUP)){
        LOGGER.info(o: "Initialize SOFA_ROUTER_GROUP:" + SOFA_ROUTER_GROUP);
        TracerContextUtil.putPenetrateAttribute("group", SOFA_ROUTER_GROUP);
    }
}
/**
```

```
/** * 初始化trace */
public static void initTrace() {
    try {
        clearTrace();
        RpcLogContext rpcLogContext = new RpcLogContext();
        rpcLogContext.setTraceId(TraceIdGenerator.generate());
        rpcLogContext.setRpcId(Tracer.ROOT_RPC_ID);
        AbstractLogContext.set(rpcLogContext);
        //eventContext
        EventContext eventContext = new EventContext();
        eventContext.setTntInstId(FuncTestBase.TNTINSTID);
        eventContext.setChannelSeqNo(UUID.randomUUID().toString().replace( targ...
        eventContext.setPdCode("DEFAULT");
        eventContext.setPdEventCode("DEFAULT");
        EventContextUtil.saveEventContextToTracer(eventContext);
        TracerContextUtil.putPenetrateAttribute("tntInstId", FuncTestBase.TNTINSTID);
        initSofaRouterGroup();
    } catch (Exception e) {
        LOGGER.error(o: "exception occurs when create dummy Log context", e);
    }
}

private static void initSofaRouterGroup() throws TracerException {
    String SOFA_ROUTER_GROUP = OpenApiUtil.getProperty("SOFA_ROUTER_GROUP");
    if(StringUtil.isNotEmpty(SOFA_ROUTER_GROUP)){
        LOGGER.info(o: "Initialize SOFA_ROUTER_GROUP:" + SOFA_ROUTER_GROUP);
    }
}
```

