



Ant Technology Documentation

Distributed System Tracing

ProductVersion : 5.3.0
DocumentVersion : V20200810
Ant Technology

Copyright © 2020 Ant Technology. All Rights Reserved.

Without the prior written consent of Ant Technology, any organization, company or individual shall not extract, translate or reproduce the part or all of the contents in this document, and shall not spread or disseminate in any ways.

Trademark statement



And the other trademarks in related services are owned by Ant Technology. Other third-party registered trademarks mentioned in this document are owned by their owners.

Disclaimer

Due to product upgrades, adjustments, or other reasons, the contents of this document are subject to change. Ant Technology reserves the right to modify the contents of this document without any notice or prompt, and to post the updated user document from time to time in Ant Technology authorized channels. You should pay attention to the version changes of the document, and download or obtain the latest version of the document through the authorized channels. Ant Technology shall not be liable for any direct or indirect losses arising from improper use of the document.

Content

1 What is Distributed System Tracing	1
1.1 Overview	1
1.2 Architecture	1
1.3 Features	2
1.4 Scenarios	3
1.5 Limitations	4
1.6 Basic Terminologies	4
2 Quick start guide	5
3 Operation guide	6
3.1 Console overview	6
3.2 Install log collection client	7
3.3 Set application log association	9
3.4 View application topology	11
3.5 View application details	11
3.6 Search trace	13
3.7 Set custom tags	13
3.8 View trace details	15
4 SOFATracer	15
4.1 What is SOFATracer	15
4.2 TraceId and SpanId generation rules	16
4.3 Get started with SOFATracer	18
4.4 Spring MVC event tracking	19
4.5 HttpClient event tracking	22
4.6 DataSource event tracking	25
4.7 RestTemplate event tracking	29
4.8 OkHttp event tracking	32
4.9 Dubbo event tracking	34
4.10 Spring Cloud OpenFeign event tracking	38
4.11 SLF4J MDC integration	41
4.12 Asynchronous processing	43
4.13 Sampling mode	45
4.14 Data reporting to Zipkin	48
4.15 Tracer utility class	50
4.16 Tracer DRM switch	53
4.17 Tracer log configuration items	55
4.18 Log format	57

1 What is Distributed System Tracing

1.1 Overview

Distributed System Tracing (DST) is a financial solution that provides application observability for cloud native architectures. These architectures include distributed architectures and microservice architectures (such as Spring Cloud, SOFAShark, and Service Mesh).

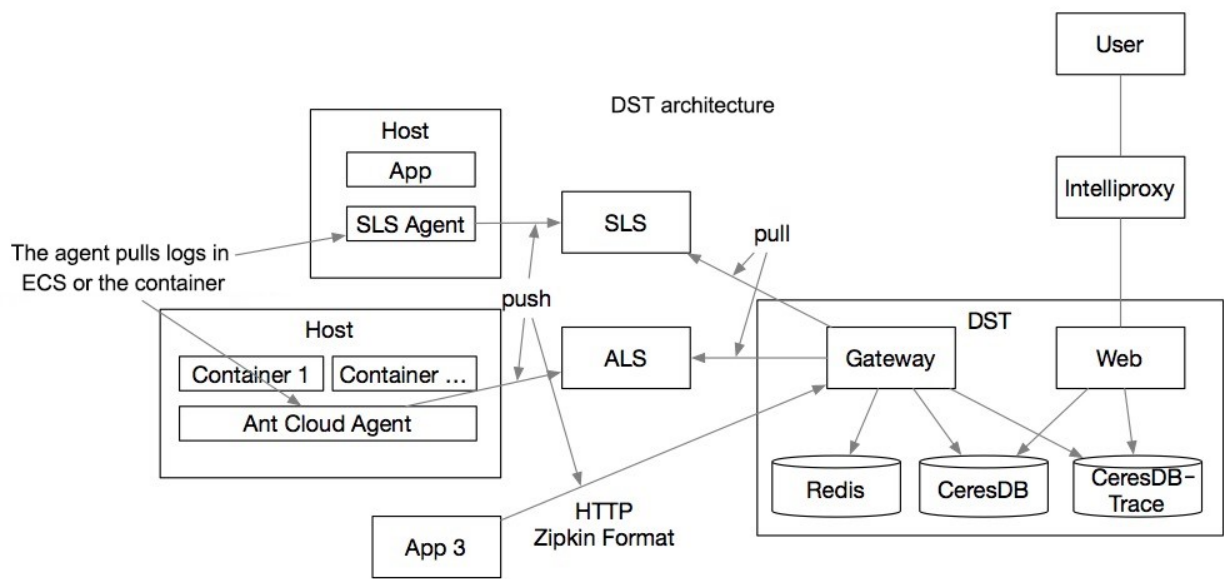
DST enables operators, developers, and architects to see the **complex calls**, **performance indicators**, **error messages**, and **association logs** between applications and services in large-scale microservice architectures. Then they can implement operations and development work such as root cause analysis, service governance, application development and debugging, performance management, performance tuning, architecture management, and responsibility determination for faults.

DST has the following advantages:

- **Full-link tracing:** DST can go deep into applications, services, databases, and messages to capture performance exceptions and identify failed components and services.
- **Easy to use:** DST can seamlessly integrate with applications on SOFAShark. The business systems of users can access DST without any modification to the business code for performance visualization and problem analysis.
- **Highly scalable:** DST conforms to the industrial OpenTracing standard and supports various mainstream programming frameworks and databases.

1.2 Architecture

The following figure shows the overall architecture of Distributed System Tracing (DST).



The overall DST architecture is divided into the following three modules.

Trace data collection channel

Currently, DST supports three data collection channels: the SLS channel of Apsara, the ALS channel of AntStack,

and the proactive report channel of apps. The first two channels collect data based on trace logs. The third channel directly reports trace data.

Trace data computation, analysis, and storage

This module mainly analyzes and computes the collected trace data and stores the results in the corresponding storage. Currently, DST uses the following storage resources:

- CeresDB: stores statistical metric data that is calculated based on trace data.
- CeresDB-Trace: stores detailed trace data.
- Redis: mainly stores some intermediate computation results that are generated during the computation process and some cache-related data.
- Database (MySQL or RDS): stores metadata related to product use and functions.

Trace-related analytical data query

This module is mainly responsible for query of front-end data. It displays the trace data and analytical computation results in the front end.

1.3 Features

Application topology discovery

Distributed System Tracing (DST) can constantly and automatically discover dependencies at the application layer. It displays in real time the application architecture in an end-to-end manner.

- **Full-link information display:** Displays applications and the response time, throughput, and state of their associated internal and external service systems. DST also displays the mutual impacts of services. If one service is disrupted, you can see immediately its impact on other services.
- **Back-end service performance management:** Monitors application performance on a fast and continuous basis, which allows you to get information of application problems first time.
- **Real-time running state:** Generates an application performance monitoring description by monitoring the golden signals (throughput, response time, and error rate), to enable you to quickly know the running state of every application.

Distributed cross-application tracing

DST tracks the complete process of every transaction, creates call time sequences of different service (application) interfaces, collects the performance data of each service on the trace, and implements tracing of transaction performance issues by service.

- **Multi-dimensional trace query:** Allows you to retrieve call traces based on the trace ID by different conditions (such as error and timeout), analyze the trace information, and query call trace collections in various scenarios.
- **Multi-perspective trace display:** Provides information such as trace diagrams, trace details,

sequence diagrams, and timelines, and displays system performance in an all-round and visual manner.

- **Business log association query:** Customizes the business logs of application systems and automatically associates with business errors and summaries of such errors to quickly locate the problems and trace the business information.

Root cause analysis

DST enables you to proactively detect performance issues, trace the slowest elements, view request parameters, analyze SQL statement execution, code errors, and exceptions, and analyze back-end bottlenecks.

- **Analysis of slow SQL:** Views SQL/NoSQL statements that are executed slowly. DST supports mainstream databases such as MySQL, OceanBase, RDS, Redis, MongoDB, and MemCached.
- **Analyzes backend errors and exceptions:** The Error Message feature detects online errors and exceptions in real time and allows you to view what the errors are about.

Deep application analysis

DST helps you conduct deep profile analysis of applications based on multiple dimensions. For example, you can perform basic performance analysis, middleware-layer analysis, exception analysis, and JVM analysis of applications. It helps you find connections between data from the bottom to top layers. Then you can thoroughly analyze the root causes of the application performance issues in distributed scenarios.

1.4 Scenarios

While resolving challenges faced by conventional single-architecture systems during rapid changes in business requirements, the distributed framework also brings about more complexity and management cost to development and operations. Applications and businesses restructured based on distributed frameworks usually face two major challenges:

- How do I quickly and accurately detect and locate problems when they occur to minimize the losses caused to my businesses?
- Can I optimize the performance of applications or downgrade the services? Where are the strong dependencies and the key paths? How do I make the budget? Can the systems analyze the nodes that first fluctuate, and accumulate test assets after massive online promotions or stress testing?

Scenario 1: Analyze and locate problems quickly

Service calls are intricate and complex in distributed scenarios, so it is difficult to analyze and locate problems. The DST system can quickly locate the problematic services and help you troubleshoot the nodes where the problems occur.

- **Complete application call topologies:** Automatically discovers the previous calls of a service

and its requests to all middleware, thereby drawing the complete call topology graph of the entire system.

- **Quick location of unhealthy applications:** Displays and identifies the unhealthy applications in the call topology graph, which allows you to quickly locate and analyze the problematic applications.
- **Service performance details:** Enables you to make in-depth and detailed performance analysis on each application in the call topology graph, from perspectives such as throughput, error rate, and response time.

Scenario 2: Application performance optimization

You can analyze the number of calls of each application in the call topology graph and their time cost, and identify the applications with a high or low load. This helps you use resources reasonably.

- **Call trace collection and summary:** Collects and summarizes all information about calls and analyzes the calls and responses of each application.
- **Critical path:** Quickly discovers the critical application path in the call topology graph of the entire system.
- **Optimization of unreasonable calls:** Immediately discovers and handles unreasonable calls, such as frequent database operations.

1.5 Limitations

Pay attention to the following limitations when you use Distributed System Tracing (DST).

Item	Range	Description
JDK language	JDK version 1.8	The JDK version must be 1.8.
SOFABoot version	3.2.1 or later	The requirement for SOFABoot 3.2.1 or later applies only when proactive reporting is enabled.

1.6 Basic Terminologies

Term	Description
Applicat ion	An application that functions as a part of a business system. An application can be a monolithic application or a microservice application based on a distributed framework.
Applicat ion topolog y	The visual presentation of calls and dependencies between applications.
SOFATra cer	SOFATracer is a component used to trace the call information in a distributed system. It logs various network calls in the call traces by using unified trace IDs to achieve perspective network calls. These logs can be used for quick failure detection and service governance.
TraceId	An ID that represents a unique request in the tracer. Generally, the system in a cluster that first processes the request generates the trace ID.
Error	The ratio of application errors based on the number of requests in a specified time period.

rate	
Throughput	The throughput trend of requests that an application processes in a specified time period.
RT distribution chart	The distribution chart of the request response time of an application in a specified time period. A green dot indicates request response success, and a red dot indicates request response failure.
Golden Signals	The four golden signals summarized by Google based on rich experience in distributed monitoring. The four golden signals help measure user experience, service interruption, business impact, and other issues at the service level, and mainly include the following four signals: throughput, response time, error rate, and saturation. The service level is usually measured by the first three signals.

2 Quick start guide

This document introduces how to deploy and use Distributed System Tracing (DST) to monitor application dependencies, performance, and other information in real time and to quickly search for traces.

Before using DST, ensure that your runtime environment meets the prerequisites so that you can perform the following operations in the product console: application analysis , viewing application details , trace search , and viewing application associated logs .

Prerequisites

1. If SOFABoot is used for DST:
 - Ensure that SOFABoot has been upgraded to version 3.2.1 or later.
 - Ensure that the Tracer dependency has been added to the pom.xml file of the SOFABoot project:
- ```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```
2. The application has been deployed in the Classic Application Service (CAS) console. For more information about the deployment process, see Classic Application Service > Application deployment .
  3. Log Service has been enabled. The underlying layer of DST collects application logs based on Ant Financial Technology’ s Log Service, so you must enable Log Service in the application services. For more information about the process, see Log Service activation process .
  4. Logtail has been installed manually or in batches with operations scripts.
  5. Application-log association has been configured for viewing the log information corresponding to a trace node.



## Application analysis

Log on to the DST console and click **Application analysis** to view the performance data and topology of global apps in the specified time span.

### View application details

Click an application node in the topology. Then click the **View application details** button that appears to access the application details page and view more monitoring data of the application, including the application overview, service calls, message calls, and database calls.

### Trace search

On the **Trace search** page, you can quickly find the trace you want to view through a multi-dimensional custom search. For more information about a business custom search, see Use business custom search items .

### View application associated logs

On the trace details page, you can click the log icon next to the application name to view the application associated logs.

## 3 Operation guide

### 3.1 Console overview

On the **Overview** page of the Distributed System Tracing (DST) console, you can view the monitoring statistics about all applications in the current environment.

- Distribution chart of application monitoring metrics
- Top 5 applications by monitoring metrics
- Favorite applications and traces

#### Distribution chart of application monitoring metrics

In the **Distribution of Application Monitoring Metrics** area, you can view the overall monitoring metrics about all the applications running in the statistics period. As shown in the following figure, a circle represents an application, the X-axis corresponds to the response time of the application, the Y-axis represents the error rate of the application, and the size of the circle indicates its number of requests. You can hover your mouse cursor over an application to view its detailed monitoring metrics.

You can click the **All applications** drop-down list to switch among the options of **10 apps with top requests**, **10 apps with top error rate**, and **10 apps with top response time**.

#### Top 5 applications by monitoring metrics

The **Top 5 applications by monitoring metrics** section displays the top 5 applications sorted respectively by global call amount, response time, and error rate in the default statistics period, as well as the detailed monitoring metrics about the applications. You can click an application name to access its **Application details** page. For more information, see [View application details](#) .

### Favorite applications and traces

This section displays the list of all favorite applications and traces.

#### List of favorite applications

All favorite applications are listed, and their related performance information is displayed, including the request amount, response time, and error rate. You can click an application name to directly access its details page. For more information, see [View application details](#) .

#### List of favorite traces

All favorite traces are listed, and related information is displayed, including the trace description, operator, and time being added to Favorites. You can click a trace name to directly access its **Trace details** page. For more information, see [View trace details](#) .

## 3.2 Install log collection client

Logtail, the log collection client, can be installed manually or in batches by using operations scripts.

#### Batch installation (recommended)

Create the specified template.

Log on to the **SOFASTack** console, choose **Operation management** > **Classic Application Service** > **Daily operation** > **Command template**, and click **Create template** on the right.

On the **Create command template** page, enter the corresponding environment information. For the sample script for a Linux system, see [dst\\_sls\\_install.sh](#). The following figure shows how to configure an instruction template on a Linux system.

新建指令模板

描述: 上海金区 sls agent 安装

服务器账号: root

指令类型:

\* 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 = "shfin" -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
```

**Note:** To run the script in different workspaces, modify the instanceId and region parameters.

- **instanceId** specifies the machine group ID. To obtain the value, choose **Distributed System Tracing > Settings**.
- **region** specifies the domain environment ID, for example, cn-shanghai , that is, China (Shanghai). For more information about the value, see Regions and zones.

Run the operations script.

Log on to the **SOFASTack** console, choose **Classic Application Service > Daily operation > ECS operation**, and click **Create** on the right.

On the **Create ECS operation ticket** page, select the created instruction template and corresponding machine list, select **Execute automatically after creation**, and click **Create**.

**Note:** It’ s recommended to select several servers to verify the environment and then run the script in the entire environment.

Manual installation

Check whether the log collection client (Logtail) has been installed for the ECS instance in the tenant environment.

- On the Linux system, run the following command to view the client status:

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

- On the Windows system, view the client status on the **Services** page by choosing **Control panel > Management Tools > Services**. View the status of the two Windows services in the list: LogtailDaemon and LogtailWorker. If Logtail has not been installed, see the documents about how to install Logtail on a Linux system or Windows system .

Add a machine group ID.

On the Linux system, add the `/etc/ilogtail/user_defined_id` file that contains `instanceId`. For example, on the Linux system, ensure that the `/etc/ilogtail/user_defined_id` file exists and contains a machine group ID. (To obtain the machine group ID, choose **Distributed System Tracing > Settings**).

On the Windows system, open the `user_defined_id` file (stored at `C:\LogtailData\user_defined_id` by default) and change the file content to the machine group ID.

3. Configure a user ID file on the ECS server to authorize DST to collect logs.

- On the Linux system, create a folder named after the account ID in the `/etc/ilogtail/users` directory to configure a user ID, for example:  
Non-financial region in Shanghai - East China (Shanghai):

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

- On the Windows system, create a folder named after the account ID in the `C:\LogtailData\users` directory to configure a user ID, for example:  
Non-financial region in Shanghai - East China (Shanghai):

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

### 3.3 Set application log association

Log association in Distributed System Tracing (DST) allows you to view the log information corresponding to a specific trace in single-trace display.

#### Principle

- The log output pattern in the application log output configuration file is modified to obtain the `TraceId` and `SpanId` in the application logs.
- The names of cloud applications are logged, and the log name and log path are set for collecting logs.
- When you click an application node on the trace, the application searches for and displays

the corresponding logs based on the application name, TraceId, and SpanId.

## Procedure

### Configure local application log printing

To associate a trace with application logs, the TraceId and SpanId must be obtained correctly during local application log printing. DST Tracer is integrated with the SLF4J MDC feature, allowing you to obtain TraceId and SpanId in the logs simply by modifying the log output pattern in the log configuration file.

1. Ensure that the Tracer dependency is introduced in the SOFABoot application.

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

2. Add the %X{SOFA-TraceId} and %X{SOFA-SpanId} settings to PatternLayout.

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

For example, after the previous settings are completed, the following application logs are actually generated:

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

### Add cloud application logs

1. Log on to the DST console and click **Settings** in the left-side navigation pane.
2. When you use the log association feature for the first time, log on to the RAM console, and create and obtain the AccessKey ID and AccessKey secret configured with the AliyunLogFullAccess permission policy. For more information, see [Create an AccessKey pair](#).
3. Choose **Distributed System Tracing** > **Settings** and enter the AccessKey ID and AccessKey secret obtained.
4. On the **Log configuration** page, click **Add application**. In the dialog box that appears, enter the correct application name and click **OK**.

**Note:** The application name must be the same as appName in the system logs. Otherwise, no application data can be obtained.

5. After the application is successfully added, click **Add log**.

6. In the **Add log** dialog box that appears, set the log name, log path, and other information, and click **OK**.

#### View application associated logs

You can go to the trace search page, find the traces related to the application, and view the trace details. On the trace details page, you can click the log icon on the right of the application name to view related logs.


### 3.4 View application topology

The **Application analysis** page displays the performance data and topology of global applications in the specified time span.

#### View the application topology

1. Log on to the DST console and click **Application analysis** in the left-side navigation pane.
2. In the upper-left corner of the application topology, select or customize a time span, which is the last 15 minutes by default and can be set to a maximum of 7 days.
3. Select the metrics data you want to view.
  - If you select **Node**, the topology displays the performance metrics about each node.
  - If you select **Connection line**, the topology displays the traffic on traces between different nodes.
4. In the topology, click any application node or trace to view the monitoring trend charts for the performance metrics of the application or trace.
5. Click an application node. Then click the **View application details** button that appears above the node. For more information, see View application details .

#### View the application list

In the upper-right corner of the **Application analysis** page, click the  icon to switch to the application list view and check the overview information about all the applications in the specified statistics period, including the traffic destination, protocol type, request amount (RPS), response time (ms), and error rate.

You can also perform the following operations on the applications:

- Click any application at the traffic origin or destination to view the application details. For more information, see View application details .
- Click the protocol type to view the traffic monitoring data.

### 3.5 View application details

The **Application details** page displays the detailed monitoring data of apps, including the app overview, service calls, message calls, and database calls.

### Procedure

1. Log on to the DST console and click **Application details** in the left-side navigation pane.
2. In the upper-left corner of the page, select the target application you want to view from the **Current application** drop-down list.
3. In the upper-right corner of the page, select or customize a time span, which is the last 15 minutes by default and can be set to a maximum of 7 days.
4. On the current page, switch between the tab pages to view the overall data of the target application in the specified time span:
  - Overview
  - Services
  - Messages
  - Databases

### Overview

The **Overview** tab displays the upstream/downstream topology of the current application and its metrics trends, including the request amount, response time, and error rate.

### Services

The **Services** tab displays the overview information about all the services published and referenced by the current application, including the service names, method names, calls, errors, and average time costs.

In addition, for the service method, you can perform the following operations to view more trace data:

- Click the chart icon in the data column for each monitoring metric to view its trend chart.
- Click **View trace** to view the information about all traces. Click a trace ID to view details about the trace. For more information, see [View trace details](#).

### Messages

The **Messages** tab displays all the messages published and consumed by the current application and the information related to each message, including the message topic (topic), message event code (eventCode), message group ID (groupId), calls, errors, and average time cost. Click **View trace** to view the overview information about all traces related to the message.

### Database

The **Databases** tab displays the information about all the databases called by the application, including the database address, database name, calls, errors, and average time cost. Click **View trace** to view the overview information about all traces related to the database.

### LDC (only for private cloud users)

Private cloud users who have enabled the LDC feature can choose a view (LDC or data center) after clicking the trend chart icon to check the change trends of corresponding monitoring metrics, as shown in the following figure.

## 3.6 Search trace

This topic describes how to search for call traces on the **Trace search** page.

### Procedure

1. Log on to the DST console and click **Trace search** in the left-side navigation pane.
2. On the **Trace search** page, set the following search items according to your business requirements and click **Search**.
  - Basic search items:
    - Trace Id: The unique ID of a trace.
    - Call time: The call time span, which is the last 10 minutes by default.
    - Call type: ALL, RPC, HTTP, DB, or MQ.
    - Application name: The name of an app.
    - Result: ALL, Succeeded, or Failed.
    - Response time: The response time span in ms.
  - Custom search items:
    - Custom middleware search items: serviceName, methodName, statusCode, database, eventId, msgId, topic, and eventCode. Select a search item and enter the corresponding value.
    - Custom business search items: Enter corresponding tags and their values in the format of key=value based on the custom business tags (such as ID card number and phone number) added to the traces. For more information, see Add business custom search items .
3. In the search result list, obtain the trace ID, call time, response time, result, client name and address, server name and address, and service information of the traces.
4. Click any trace ID to view trace details. For more information, see View trace details .

Your search history and favorite traces are displayed at the bottom of the **Trace search** page.

- In the search history section, you can click **Clear** to delete the search history.
- You can click **Favorites** on the right of a trace to add it to your favorites, for a quick search subsequently.

## 3.7 Set custom tags



In scenarios where a phone number or serial number is required for trace information query, you can configure the information as custom tags and add them to the span.

### Prerequisites

- The application is based on SOFABoot.
- Tracer meets the following conditions:
  - The version is 3.0.5 or later.
  - Proactive reporting is enabled.
  - The business code explicitly depends on Tracer.

### Note:

- When Tracer 3.0.5 or a later version is used, proactive reporting is enabled by default. If the version of Tracer is earlier than 3.0.5, upgrade it first.
- Explicit dependency on Tracer means that the business code needs to call Tracer classes, for example, SofaTraceContext. This operation is intrusive. Therefore, proper encapsulation is required.

### Add tags

You can call Tracer classes in the business code to set custom tags and corresponding values. The sample code is as follows:

```
public void handlerXxx(Map<String, String> request) {
 SofaTraceContext ctx = SofaTraceContextHolder.getSofaTraceContext();
 SofaTracerSpan span = ctx.getCurrentSpan();
 // Explicit dependency on Tracer is intrusive. Therefore, proper encapsulation is required.
 if (span != null) {
 // Add the parameter to the custom tag: xxx_phone.
 span.setTag("xxx_phone", request.get("phone"));
 }

 // Process the service logic.
 // ...
}
```

### Verification

To verify the added custom tags, click **Trace search** and enter the custom tags and corresponding values in the **Custom business search items** text box.

Assume that the custom tag is **xxx\_phone** and the corresponding value is **123**.

On the details page of the target trace, hover the cursor over the **Service information** column. **Service information details** appears, displaying the custom tag and corresponding value.

## 3.8 View trace details

The trace details page displays two types of charts, showing the detailed call information about the trace. You cannot only view all the details about the trace, such as server information and trace logs, but also understand the call relationship in the trace based on the upstream/downstream topology.

### Single-trace sequence diagram

The trace details page displays the sequence diagram of a trace by default, showing the most detailed call information for your in-depth understanding of the trace.

- You can directly obtain the following information about the trace-associated apps: app name, SpanId, IP address, call type, call status (successful/failed), service information overview, and time cost.
- You can hover your cursor over the service information column to view more details, such as the client information, server information, service name, method name, time cost, and results.
- You can click the log icon next to the app name to view the logs of the app, including business logs and error logs.

**Note:** Before using the log feature, configure app-log association. For more information, see [Configure application log association](#).

### View LDC information (only for private cloud users)

**Note:** This feature is only applicable to private cloud users who have enabled the LDC feature.

When the LDC feature is enabled, you can also view the LDC information in the single-trace sequence diagram on the trace details page, which covers the data center and LDC where the server and client are located, as well as the source zone and destination zone of the trace.

### Single-trace topology

You can click the topology icon to switch to the single-trace topology page and view the upstream/downstream call topology of the trace and the performance data (request amount, response time, and error rate) between nodes or call relationships.

## 4 SOFATracer

### 4.1 What is SOFATracer

SOFATracer is a Distributed System Tracing (DST) solution developed by Ant Financial based on the [OpenTracing specifications](#). Its core idea is to associate a request on different servers with a global TraceId. Various network calls in the trace are logged based on the unified TraceId and remotely reported to [Zipkin](#) for display, to achieve perspective network calls.

## Features

### DST solution based on the OpenTracing specifications

SOFATracer provides a DST solution by extending the [OpenTracing specifications](#). Various frameworks or components can be implemented on this basis, enabling tracing through event tracking in each component.

### Asynchronous logs generated to disks

SOFATracer asynchronously generates logs to local disks based on [Disruptor](#), a high-performance lock-free circular queue. When a framework or component is being connected, you can customize the output format of the log file in asynchronous log output mode. SOFATracer generates two similar types of logs: digest logs and statistical logs. The digest logs are generated to disks upon each call. The statistical logs are generated at regular intervals.

### Log auto-clearance and rolling

SOFATracer logs asynchronously generated to disks can be automatically cleared by day and rolled by hour or day.

### Extension based on SLF4J MDC

SLF4J provides the Mapped Diagnostic Contexts (MDC) feature, allowing you to customize and modify the log output format and content. SOFATracer is integrated with the SLF4J MDC feature so that you can simply modify the log configuration file to provide the TraceId and SpanId of the current SOFATracer context.

### UI display

SOFATracer can remotely report tracing data to the open-source [Zipkin](#) for display.

### Unified configuration

The configuration file provides extensive configuration options to help you customize the app as needed.

## Scenarios

SOFATracer implements tracing in a large-scale microservice architecture, enables perspective network calls, and facilitates quick failure detection and service governance.

### Event tracking for components

At present, SOFATracer is applicable to open-source components such as Spring MVC, database connection pools (DBCP, Druid, C3P0, Tomcat, HikariCP, and BoneCP) implemented based on standard JDBC interfaces, HttpClient, Dubbo, and Spring Cloud OpenFeign. Event tracking for other open-source components such as MQ and Redis are under development.

## 4.2 TraceId and SpanId generation rules

### TraceId generation rule

SOFATracer associates the call logs of a request on multiple servers based on TraceId, which is usually generated by the first server that the request passes.

TraceId consists of the following information: server IP address + ID generation time + auto-increment sequence + current process ID. The following is an example:

```
0ad1348f1403169275002100356696
```

The first eight characters 0ad1348f indicate a hexadecimal number representing the IP address of machine generating the TraceId. With each two digits indicating a section of the IP address, the first eight characters can be converted into a common IP address in the decimal format, such as 10.209.52.143. In this way, you can find the first server that the request passes.

The next 13 digits 1403169275002 specify the TraceId generation time. The last four digits 1003 indicate an auto-increment sequence, which increases from 1000 to 9000, returns to 1000 after reaching 9000, and increases from 1000 again. The last five digits 56696 indicate the current process ID, which is added to the end of the TraceId to avoid conflicts when there are multiple processes running on one instance.

**Note:** The current TraceId generation rule is determined with reference to Alibaba EagleEye.

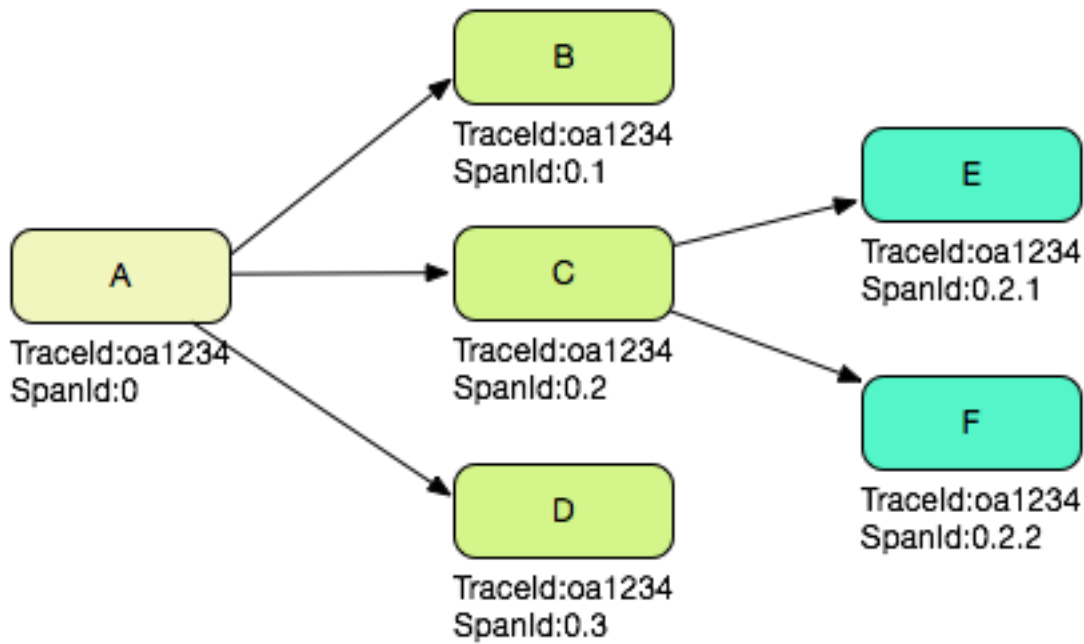
### SpanId generation rule

SpanId in SOFATracer indicates the location of the current call in the entire call tree.

Assuming that Web system A receives a user request, the SpanId recorded in the SOFATracer MVC logs of the system is 0, which indicates the root node of the entire call. If system A needs to call systems B, C, and D through RPCs in turn for request processing, the SpanIds recorded in the SOFATracer RPC client logs of system A are 0.1, 0.2, and 0.3, and those recorded in the SOFATracer RPC server logs of systems B, C, and D are also 0.1, 0.2, and 0.3. If system C further calls systems E and F, the SpanIds recorded in the SOFATracer RPC client logs of system C are 0.2.1 and 0.2.2, and those recorded in the SOFATracer RPC server logs of systems E and F are also 0.2.1 and 0.2.2.

In this way, all the SpanIds in a call can be collected to form a complete call tree.

Assume that the TraceId generated in a distributed call is 0a1234, which is longer in practices. The following figure shows the SpanId generating process.



**Note:** The current SpanId generation rule is determined with reference to Alibaba EagleEye.

### 4.3 Get started with SOFATracer

When you use SOFATracer, pay attention to the version mapping among SOFATracer, JDK, and different components.

#### Prepare the environment

To use SOFABoot, prepare the basic environment first. SOFABoot depends on the following environment:

- JDK 7 or JDK 8
- Apache Maven 3.2.5 or later versions for compilation

#### Examples

All the following sample projects are SOFABoot projects. SpringBoot projects are also supported. For more information about how to create a SOFABoot project, see SOFABoot quick start guide.

- Component integration
  - Spring MVC event tracking
  - HttpClient event tracking
  - DataSource event tracking
  - RestTemplate event tracking
  - OkHttp event tracking
  - SOFARPC event tracking

- Dubbo event tracking
- Spring Cloud OpenFeign event tracking
- Sampling mode
- Data reporting to Zipkin

## 4.4 Spring MVC event tracking

This document describes how to enable event tracking for Spring MVC by using SOFATracer. If you have built a simple Spring Web project based on SOFABoot, you can perform the following operations:

1. Introduce SOFATracer dependency
2. Add a controller
3. Run a project
4. View logs

### Introduce SOFATracer dependency

Introduce the following SOFATracer dependency in the Web project of SOFABoot:

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

After adding the SOFATracer starter dependency, add configuration items to the global configuration file of SOFABoot to customize the SOFATracer behavior. For more information, see SOFATracer configuration items .

### Add a controller

If your Web project contains no Spring MVC-based controller, you can add one in the following way. If a controller already exists, you can directly access the corresponding service.

```
@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.setMsg(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;
 }
}
}
```

### Run a project

Introduce the SOFABoot project in IDE, correctly compile the project, and run the main method in the project to start the application. Using the preceding controller as an example, you can enter <http://localhost:8080/greeting> in the browser to access the RESTful service. The result is similar to the following:

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

### View logs

You can define the log output directory in the `application.properties` file of SOFABoot. Assume that the log output directory is `./logs`, which is the root directory of the current application, and that the

application name is `spring.application.name=mvc-client`. In this case, you can find log files in a structure similar to the following in the root directory of the current project:

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

`spring-mvc-digest.log` contains the detailed output content. The following shows a sample log record:

```
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,
```

For more information about the output fields, see [Log formats > Spring MVC logs](#) .

SOFATracer configuration items

SOFATracer configuration item	Description	Default value
logging.path	The log output directory.	SOFATracer preferentially generates logs to the logging.path directory. If no log output directory is configured, SOFATracer generates logs to the \${user.home} directory by default.
com.alipay.sofa.tracer.disableDigestLog	Specifies whether to disable digest log output for all SOFATracer components.	false
com.alipay.sofa.tracer.disableConfiguration[\${logType}]	Specifies whether to disable digest log output for the SOFATracer component with the specified \${logType}. \${logType} specifies the log type, for example, spring-mvc-digest.log.	false
com.alipay.sofa.tracer.tracerGlobalRollingPolicy	The SOFATracer log rolling policy.	.yyyy-MM-dd specifies rolling by day. .yyyy-MM-dd_HH specifies rolling by hour. Rolling by day is not configured by default.
com.alipay.sofa.tracer.tracerGlobalLogReserveDay	The SOFATracer log retention period in days.	SOFATracer logs are retained for 7 days by default.
com.alipay.sofa.tracer.statLogInterval	The statistical log output interval in seconds.	Statistical logs are generated every 60s by default.
com.alipay.sofa.tracer.baggageMaxLength	The maximum length of penetration data that can be stored.	The default value is 1024.
com.alipay.sofa.tracer.springmvc.filterOrder	The order for the filters integrated by SOFATracer in Spring MVC to take effect.	-2147483647 (org.springframework.core.Ordered#HIGHEST_PRECEDENCE + 1)
com.alipay.sofa.tracer.springmvc.urlPatterns	The URL patterns for the filters integrated by SOFATracer in Spring MVC to take	/* All take effect.



	effect.	
--	---------	--

## 4.5 HttpClient event tracking

**Note:** This feature is applicable to HttpClient 4.3.x to 4.5.x.

[HttpClient](#) is an open-source Apache component that provides an efficient and up-to-date HTTP-based client programming toolkit with diverse features.

You can introduce HttpClient in the SOFABoot project for printing SOFATracer logs.

This document introduces how to enable event tracking for HttpClient by using SOFATracer. Assuming that you have built a simple Spring Web project based on SOFABoot, you can perform the following operations:

1. Import the Maven dependency
2. Add a controller providing RESTful services
3. Build HttpClient and call the RESTful service
4. Run a project
5. View logs

### Import the Maven dependency

To use HttpClient for printing SOFATracer logs, introduce the following Maven dependency in the pom.xml file of the project:

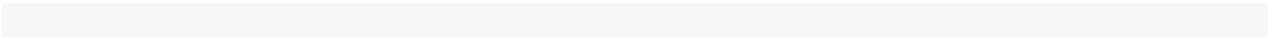
- SOFATracer dependency

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

- SOFATracer-based HttpClient plug-in  
To use HttpClient, a third-party open-source component, for trace calls, SOFATracer provides an extension plug-in, that is, tracer-enterprise-httpclient-plugin:

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

- HttpClient dependency



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

### Add a controller providing RESTful services

Sample code:

```
@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;
 }
}
```

### Build HttpClient and call the RESTful service

To ensure correct event tracking and log printing for the SOFATracer-based HttpClient in the project, use the `com.alipay.sofa.tracer.enterprise.plugins.SofaTracerEnterpriseHttpClientBuilder` class to build an HttpClient instance and explicitly call the `clientBuilder` method.

`SofaTracerEnterpriseHttpClientBuilder` provides the `clientBuilder` (synchronous) and `asyncClientBuilder` (asynchronous) methods to build an `org.apache.http.impl.client.HttpClientBuilder` instance for which event tracking is implemented by SOFATracer. The method signature is as follows:

```
// The synchronous call builder
public static HttpClientBuilder clientBuilder(HttpClientBuilder clientBuilder);

// The synchronous call builder, with the fields of the current and target apps explicitly displayed:
public static HttpClientBuilder clientBuilder(HttpClientBuilder clientBuilder,
```

```
String currentApp, String targetApp);

// The asynchronous call builder
public static HttpAsyncClientBuilder asyncClientBuilder(HttpAsyncClientBuilder httpAsyncClientBuilder);

// The asynchronous call builder, with the fields of the current and target apps explicitly displayed:
public static HttpAsyncClientBuilder asyncClientBuilder(HttpAsyncClientBuilder httpAsyncClientBuilder,
String currentApp, String targetApp) ;
```

#### Sample code

- Build a synchronous call of HttpClient.

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

- Build an asynchronous call of HttpClient.

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

When the HttpClient instance built by SofaTracerEnterpriseHttpClientBuilder calls the RESTful service, trace data is tracked for SOFATracer.

#### Run a project

1. Introduce the SOFABoot project in IDE.
2. Compile and run the main method in the project to start the app.
3. Initiate the `http://localhost:8080/httpclient` RESTful service provided by the controller by using the HttpClient instance built by `SofaTracerEnterpriseHttpClientBuilder`.

For more information, see SOFABoot quick start guide.

#### View logs

The SOFATracer log directory is defined in the `application.properties` file of SOFABoot.

The following example assumes that the log output directory is `./logs`, which is the root directory of the current app, and that the current and target app names input by `SofaTracerEnterpriseHttpClientBuilder` are `testSyncClient` and `testSyncServer`, respectively. In this case, you can find a log file with a structure similar to the following in the root directory of the current project:

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

Using HttpClient synchronous calls as an example, httpclient-digest.log (the digest log file) is as follows:

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

Using HttpClient synchronous calls as an example, httpclient-stat.log (the statistical log file) is as follows:

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

For more information about the HttpClient log fields, see [Log formats > HttpClient logs](#) .

## 4.6 DataSource event tracking

This document describes how to implement event tracking for DataSource by using SOFATracer.

SOFATracer 2.2.0 is implemented based on standard JDBC interfaces and is applicable to event tracking for standard database connection pools, for example, DBCP, Druid, C3P0, Tomcat, HikariCP, and BoneCP. The following shows how to enable the event tracking feature of SOFATracer.

If you have built a simple Spring Web project based on SOFABoot, you can perform the following operations:

1. Introduce the Maven dependency
2. Configure a data source
3. Configure a local application
4. Create a RESTful service
5. Run a project
6. View logs

### Introduce the Maven dependency

#### Introduce the SOFATracer dependency

Introduce the following SOFATracer dependency in the Web project of SOFABoot:

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

Then add configuration items to the global configuration file of SOFABoot to customize SOFATracer

behavior.

#### Introduce the H2Database dependency

For convenience, H2Database is used for testing. Introduce the following dependency:

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

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

#### Introduce the connection pool dependency

Introduce dependency packages of required connection pools, for example, Druid, C3P0, Tomcat, DBCP, and HikariCP.

```
<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>
```

#### Configure a data source

For example, if the connection pool is HikariCP, create a Spring configuration file named datasource.xml and define the following content:

```
<?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>
```

### Configure a local application

#### Required settings:

You must set the application name when introducing SOFATracer; otherwise, the application fails to start. This attribute is consistent with the requirement of SOFABoot. A configuration example is as follows:

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

#### Optional settings:

To properly run the sample project, you need to set the H2Database attributes. In addition, to conveniently view the logs, you need to set the log path. A configuration example is as follows:

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

The path to the H2 Web console.
spring.h2.console.path=/h2-console
Enable the H2 Web console, with a default value of false.
spring.h2.console.enabled=true
Allow remote access to the H2 Web console.
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
```

### Create a RESTful service

Create a RESTful service and trigger execution of SQL statements to conveniently view the SOFATracer logs for SQL execution. Table creation is triggered in the following RESTful service creation process.

```
@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;
 }

}
```

### Run a project

Import the SOFABoot project into the IDE, correctly compile the project, and run the main method in the project to start the application. Then access [localhost:8080/create](http://localhost:8080/create) in the browser to run the RESTful service.

### View logs

View the SOFATracer logs for SQL execution in `./logs/datasource-client-digest.log` and `./logs/datasource-client-stat.log`.

- Sample `datasource-client-digest.log`:

```
{
 "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": ""
}
```

- Sample `datasource-client-stat.log`:

```
{
 "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"}
```

## Others

After you introduce the SOFATracer dependency in the SOFABoot project, DataSource event tracking is automatically enabled. To disable DataSource event tracking, set the following switch:

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

## Considerations

- You must set the application name when introducing SOFATracer; otherwise, the application fails to start. The attribute name is `spring.application.name`.

SOFATracer 2.2.0 is implemented based on standard JDBC interfaces and is theoretically applicable to event tracking for all standard database connection pools, for example, DBCP and BoneCP. In the Spring Boot environment, automatic event tracking is available for the DBCP, Druid, C3P0, Tomcat, and HikariCP connection pools, and therefore you only need to introduce the SOFATracer dependency. In a non-Spring Boot environment or for other types of connection pools such as BoneCP, manual configuration is required, for example:

```
<bean id="smartDataSource" class="com.alipay.sofa.tracer.plugins.datasource.SmartDataSource" init-
method="init">
<property name="delegate" ref="simpleDataSource"/>
<!-- Application name -->
<property name="appName" value="yourAppName"/>
<!-- Database name -->
<property name="database" value="yourDatabase"/>
<!-- Database type: MySQL or 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="abcdefgh"/>
...
</bean>
```

As shown above, you need to additionally configure

`com.alipay.sofa.tracer.plugins.datasource.SmartDataSource` provided by SOFATracer.

## 4.7 RestTemplate event tracking

This document describes how to enable event tracking for RestTemplate by using SOFATracer based on the [sample project](#).

Assuming that you have built a simple Spring Web project based on SOFABoot, you can perform the following operations:

1. Introduce dependencies



2. Configure a project
3. Add a controller providing RESTful services
4. Obtain RestTemplate
5. Start an app
6. View logs

### Introduce dependencies

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

### Configure a project

Add the parameters required by SOFATracer to the application.properties file of the project, including spring.application.name (the name of the current app) and logging.path (the log output directory).

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

### Add a controller providing RESTful services

Add a simple controller to the project code, for example:

```
@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;
 }
}
```

### Build RestTemplate

Build RestTemplate by using an API to call the RESTful service.

Build a synchronous call instance of RestTemplate.

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

Build an asynchronous call instance of RestTemplate.

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

### Obtain RestTemplate

Obtain RestTemplate through automatic injection.

```
@Autowired
RestTemplate restTemplate;
```

### Start an app

Start the SOFABoot app and view the following start logs in the console:

```
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)
```

The call success logs are as follows:

```
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
```

### View logs

The log output directory in the application.properties file is ./logs, which is the root directory of the current app and can be modified as required. You can find a log file in a structure similar to the following in the root directory of the current project:

```
./logs
├── spring.log
└── tracelog
```

```

├─ resttemplate-digest.log
├─ resttemplate-stat.log
├─ spring-mvc-digest.log
├─ spring-mvc-stat.log
├─ static-info.log
└─ tracer-self.log

```

In the example, two RestTemplate instances (synchronous and asynchronous) are built to call one RESTful service. After the call is completed, you can find logs similar to the following in **restTemplate-digest.log**:

#### Digest logs

```

{"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.siz
e.bytes":0,"resp.size.bytes":0,"sys.baggage":"","biz.baggage":""}

```

#### Statistical logs

```

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

```

## 4.8 OkHttp event tracking

This topic describes how to enable event tracking for OkHttp by using SOFATracer based on the [sample project](#).

Assuming that you have built a simple Spring Web project based on SOFABoot, you can perform the following operations:

1. Introduce dependencies
2. Configure a project
3. Add a controller providing RESTful services
4. Build OkHttp and call the RESTful service
5. Start application
6. View logs

#### Introduce dependencies

```

<!-- SOFATracer dependency -->
<dependency>
<groupId>com.alipay.sofa</groupId>

```

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

### Configure a project

Add the parameters required by SOFATracer to the application.properties file of the project, including spring.application.name (the name of the current application) and logging.path (the log output directory).

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

### Add a controller providing RESTful services

Add a simple controller to the project code, for example:

```
@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;
 }
}
```

### Build OkHttp and call the RESTful service

Sample code for building a call instance of the OkHttp client:

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

### Start application

Start the SOFABoot application and view the following start logs in the console:

```
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)
```

When logs similar to the following are generated, the service call of OkHttp succeeds:

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

### View logs

The log output directory in the application.properties file is `./logs`, which is the root directory of the current application and can be modified as required. You can find a log file in a structure similar to the following in the root directory of the current project:

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

In the example, an OkHttp object is built to call the RESTful service. After the call is completed, you can find the following logs in `okhttp-digest.log`:

```
{"time":"2019-09-03
11:35:28.429","local.app":"OkHttpDemo","traceId":"0a0fe9271567481728265100112783","spanId":"0","span.kind":"cl
ient","result.code":"200","current.thread.name":"main","time.cost.milliseconds":"164ms","request.url":"http://localho
st: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 event tracking

This document shows how to enable event tracking for Dubbo by using SOFATracer based on the [project address](#).

### Basic environment

The versions of components used in this case are as follows:

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

This example involves three submodules:

- tracer-sample-with-dubbo-consumer: The service caller.
- tracer-sample-with-dubbo-provider: The service provider.
- tracer-sample-with-dubbo-facade: The API.

## Principle

SOFATracer enables event tracking for Dubbo based on Dubbo's SPI mechanism. SOFATracer customizes DubboSofaTracerFilter based on the [call interception extension](#) for event tracking for Dubbo calls. DubboSofaTracerFilter is not an official extension of Dubbo. Therefore, when you use SOFATracer, install the method provided in the [call interception extension](#) to reference the filter:

```
<!-- Intercept the consumer's call procedure. -->
<dubbo:reference filter="dubboSofaTracerFilter"/>
<!-- The default interceptor for the consumer's call procedure, which intercepts all references. -->
<dubbo:consumer filter="dubboSofaTracerFilter"/>

<!-- Intercept the provider's call procedure. -->
<dubbo:service filter="dubboSofaTracerFilter"/>
<!-- The default interceptor for the provider's call procedure, which intercepts all services. -->
<dubbo:provider filter="dubboSofaTracerFilter"/>
```

## Create a SOFABoot project as a parent project

After you create a Spring Boot project, introduce the SOFABoot dependency. First, decompress the zip package of the Spring Boot project generated previously and modify the pom.xml file of the Maven project.

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

Replace the above with the following.

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

`${sofa.boot.version}` specifies a SOFABoot version. For more information, see SOFABoot release notes.

### Create tracer-sample-with-dubbo-facade

Provide an API:

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

### Create tracer-sample-with-dubbo-provider

Add the SOFATracer dependency to the pom file of the project:

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

The SOFATracer version is controlled by the SOFABoot version. If the versions do not match, manually specify a SOFATracer version later than 2.4.0.

Add related parameters to the application.properties file of the project:

```
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
The filter must be configured.
dubbo.provider.filter=dubboSofaTracerFilter
Dubbo Protocol
dubbo.protocol.name=dubbo
Dubbo Registry
dubbo.registry.address=zookeeper://localhost:2181
logging.path=./logs
```

Publish Dubbo services by using annotations:

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

### Create tracer-sample-with-dubbo-consumer

Add the SOFATracer dependency to the pom file of the project:

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

Add related parameters to the application.properties file of the project:

```
spring.application.name=dubbo-consumer
dubbo.registry.address=zookeeper://localhost:2181
The filter must be configured.
dubbo.consumer.filter=dubboSofaTracerFilter
logging.path=./logs
```

Reference the services.

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

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

### Test the custom buildpack

Start the tracer-sample-with-dubbo-provider and tracer-sample-with-dubbo-consumer projects in sequence and view the following logs:

- 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":"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-server-digest.log

```
{"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","me
```



```
thod":"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,"succe
ss":"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"}
```

### Compatibility with Dubbo 2.6.x

SOFATracer 2.4.1 and 3.0.6 are compatible with Dubbo 2.6.x. For more information, see [tracer-dubbo-2.6.x](#).

#### Note:

- SOFATracer logs are stored in your root directory by default.
- The SOFATracer log directory depends on log-sofa-boot-starter and is determined by logging.path configured in application.properties. Therefore, for a Spring Boot project, you need to introduce log-sofa-boot-starter.

## 4.10 Spring Cloud OpenFeign event tracking

This document describes how to enable event tracking for Spring Cloud OpenFeign by using SOFATracer.

### Basic environment

The versions of components used in this case are as follows:

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

This example involves two sub-projects:

- tracer-sample-with-openfeign-provider: the service provider

- tracer-sample-with-openfeign-consumer: the service caller

### Create a SOFABoot project as a parent project

After you create a Spring Boot project, introduce the SOFABoot dependency. First, decompress the zip package of the Spring Boot project generated previously and modify the pom.xml file of the Maven project.

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

Replace the above with the following.

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

`${sofa.boot.version}` specifies a specific open-source SOFABoot version. For more information, see SOFABoot release notes.

### Create tracer-sample-with-openfeign-provider

Add the SOFATracer dependency to the pom.xml file of the project.

```
<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>
```

The SOFATracer version is controlled by the SOFABoot version. If the versions do not match, manually specify a SOFATracer version later than 3.0.4.

Add related parameters to the application.properties file of the project:

```
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
```

Simple resource class

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

### Create tracer-sample-with-openfeign-consumer

Add the SOFATracer dependency to the pom.xml file of the project.

```
<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>
```

Add related parameters to the application.properties file of the project.

```
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
```

Define feign resources.

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

Enable service discovery and feign annotations.

```
@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();
 }
}
```

### Test the custom buildpack

Start the tracer-sample-with-openfeign-provider and tracer-sample-with-openfeign-consumer projects in sequence and visit <http://localhost:8082/> in the browser. Then view the logs:

The log output directory in the application.properties file is `./logs`, which is the root directory of the current application and can be modified as required. You can find a log file in a structure similar to the following in the root directory of the current project:

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

In this example, the controller provided by Spring MVC acts as a request entrance, and the OpenFeign client initiates an access call to the downstream resource. The logs are basically as follows:

```
{
 "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 SLF4J MDC integration

SLF4J provides the Mapped Diagnostic Contexts (MDC) feature, allowing you to customize and modify the log output format and content. This document describes the SLF4J MDC feature integrated in SOFATracer, which enables you to obtain TraceId and SpanId of the current SOFATracer context simply by modifying the log configuration file.

## Usage

Ensure that the following SOFATracer dependencies have been introduced in the project:

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

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

Modify the PatternLayout attribute in the existing log configuration file:

- logback example

```
<!-- This is the Appender of the SOFATracer MDC feature and can be deleted in practices. -->
<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}/archtype-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}/archtype-test-client/mdc-example.log.%d{yyyy-MM-dd}</FileNamePattern>
<!-- log preserve 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: log level 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 example

```
<?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. Add the %X{SOFA-TraceId} and %X{SOFA-SpanId} settings to PatternLayout.

- %X{SOFA-TraceId}: corresponding to TraceId. It is replaced with TraceId of the current SOFATracer context in actual execution. If no SOFATracer context is available, it is replaced with a null string.
- %X{SOFA-SpanId}: corresponding to SpanId. It is replaced with SpanId of the current SOFATracer context in actual execution. If no SOFATracer context is available, it is replaced with a null string.

You can run the following command in the business code to print logs, without additional modifications:

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

The actual output logs are as follows:

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

## 4.12 Asynchronous processing

### Use java.lang.Runnable in threads

If you enable a new thread in the code by running `java.lang.Runnable` or asynchronously handle some businesses by using a thread pool, you must transfer the SOFATracer log context from the parent thread to the child thread. `com.alipay.common.tracer.core.async.SofaTracerRunnable` provided by SOFATracer

executes this operation by default. You can use it in the following way:

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

### Use `java.util.concurrent.Callable` in threads

If you enable a new thread in the code by running `java.util.concurrent.Callable` or asynchronously handle some businesses by using a thread pool, you must transfer the SOFATracer log context from the parent thread to the child thread. `com.alipay.common.tracer.core.async.SofaTracerCallable` provided by SOFATracer executes this operation by default. You can use it in the following way:

```
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();
```

This example assumes that the object type of the return result for `java.util.concurrent.Callable` is `java.lang.Object`, but the type can be changed as required in practices.

### SOFATracer for thread pools and asynchronous scenarios

#### Asynchronous scenario

For asynchronous calls, such as RPCs, an RPC request is initiated before a result is returned for the previous RPC request. There is a time gap between the return of the callback of the previous RPC request and the initiation of a new PRC request. In that period, `TracerContext` in the current thread has not been cleared, and therefore `spanId` automatically increases while `traceId` remains the same.

In this scenario, for asynchronous calls, SOFATracer clears `tracerContext` in the current thread in advance to ensure the correctness of the trace, instead of clearing it at the CR stage upon the return of the callback.

#### Thread pool

At present, event tracking for SOFARPC and Dubbo is implemented in the same way when a single thread or a thread pool is used.

- Synchronous calls: A thread in the thread pool is assigned for handling an RPC request, and the thread is occupied until the request ends. This prevents the next RPC request from mistakenly using the tracerContext data of the request.
- Asynchronous calls: The context is cleared in advance, but not in the callback, thereby avoiding data confusion.
- Callback-based asynchronous calls: They are essentially asynchronous calls. Therefore, the processing is the same as that for asynchronous calls.

#### Related topic

- [Sample project](#)

## 4.13 Sampling mode

At present, SOFATracer provides two sampling modes: a fixed-ratio sampling mode implemented based on BitSet and a user-defined sampling mode.

This document describes how to use the sampling modes with examples. The examples are based on the tracer-sample-with-springmvc project and are basically the same except for the **application.properties** file.

#### Fixed ratio sampling

Add sampling configuration items to **application.properties**

```
Sampling ratio: 0 to 100
com.alipay.sofa.tracer.samplerPercentage=100
Type and name of the sampling mode
com.alipay.sofa.tracer.samplerName=PercentageBasedSampler
```

#### Verification method

- When the sampling ratio is 100, digest logs are generated each time.
- When the sampling ratio is 0, no logs are generated.
- When the sampling ratio is between 0 and 100, logs are generated at a specific probability.

Verify the results based on 10 requests.

1. When the sampling ratio is 100, digest logs are generated each time.

Start the project, enter <http://localhost:8080/springmvc> in the browser, refresh the URL 10 times, and verify the following logs:

```
{"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
":"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. When the sampling ratio is 0, no logs are generated.

Start the project, enter <http://localhost:8080/springmvc> in the browser, refresh the URL 10 times, and go to the `./logs/tracerlog/` directory. Check that no `spring-mvc-digest.log` file is generated.

3. When the sampling ratio is between 0 and 100, logs are generated at a specific probability.

Assume that the sampling ratio is 20.

- Refresh the request 10 times.

```
{
 "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": "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": "0a0fe8ec154173692177410319846",
 "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": ""
}
```

- Refresh the request 20 times.

```
{
 "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": "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": "0a0fe8ec154173692177410319846",
 "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": "0a0fe8ec154173692243810359846",
 "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": "0a0fe8ec154173692281510379846",
 "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": ""
}
```

**Note:** Sampling is performed by 20%, and the test results are only for your reference.

## User-defined sampling mode

SOFATracer provides the Sampler API for calculating the sampling ratio, allowing you to customize your sampling policy. The following describes how to use the user-defined sampling mode with a simple example.

Customize a sampling rule class

```
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
 }
}
```

In the sample method, you can determine whether to generate logs based on the current value of **sofaTracerSpan**. In the example, whether sampling is required depends on the value of **isServer**. **isServer=true** indicates that no sampling is required. Otherwise, sampling is required.

Add sampling configuration items to **application.properties**

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

You can verify the related test result.

## 4.14 Data reporting to Zipkin

This document describes how to integrate Zipkin in SOFATracer for data reporting. If you have built a simple Spring Web project based on SOFABoot, you can perform the following operations:

1. Introduce dependencies
2. Configure the file
3. Start the Zipkin server

4. Start application
5. View the display on the Zipkin server

## Introduce dependencies

### Introduce SOFATracer dependency

Introduce the SOFATracer dependency to the project.

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

### Configure Zipkin dependency

Zipkin's data reporting feature is not enabled in SOFATracer by default. Therefore, to use SOFATracer for data reporting, add the following Zipkin dependency:

```
<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>
```

## Configure the file

Add the parameters required by SOFATracer to the **application.properties** file of the project, including **spring.application.name** (the name of the current application) and **logging.path** (the log output directory).

```
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
```

## Start the Zipkin server

Start the Zipkin server for receiving and displaying the trace data reported by SOFATracer. For more information about how to configure and set up the Zipkin server, see this topic.

## Start application

You can introduce the project in IDE and run the main method in the generated project to start the application. You can also directly run `mvn spring-boot:run` in the root directory of the project and view the start logs in the console.

```
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.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)
```

You can enter <http://localhost:8080/helloZipkin> in the browser to access the RESTful service. The result is similar to the following:

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

#### View the display on the Zipkin server

Go to the Zipkin server UI. If the Zipkin server URL is <http://localhost:9411>, access the URL and search for helloZipkin (because the local access address is localhost:8080/helloZipkin). The trace diagram is displayed.

#### Run a Spring project

For a general Spring project, you can generally use Tomcat or Jetty as a servlet container to start an application. For more information, see [Use SOFATracer in a Spring project](#).

## 4.15 Tracer utility class

### Dummy log context utility class

SOFATracer provides the `DummyContextUtil` class for performing operations on dummy log contexts.

**Note:** When you create a dummy log context with `DummyContextUtil`, call the corresponding destroy method for timely destruction. We recommend that you write the call to the destroy method into the finally method.

#### Create a dummy log context

Definition: `public static void createDummyLogContext()` throws Exception

Description: Create a dummy log context and incorporate it in the thread context. If the current thread already contains a log context, an exception is generated.

### Clear a dummy log context

Definition: public static void clearDummyLogContext() throws Exception

Description: Clear the current dummy log context. If the log context in the current thread context is not DummyLogContext, an exception is generated.

Set the full-process stress testing flag and call it with the clearDummyLogContext class. The code is as follows:

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

### SOFATracer context utility class

This class mainly provides a unified method for business systems to obtain TraceId, and also provides an API for obtaining and configuring penetration data. It is namedTracerContextUtil and used as follows:

#### Obtain TraceId in the current SOFATracer context

Sample code:

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

Note:

- A null string is returned if the current SOFATracer context is null.
- A null string is returned if the current SOFATracer context is not null but TraceId is null.
- TraceId is returned if neither the current SOFATracer context nor TraceId is null.

**Important:** The prerequisite for the return of TraceId is that TracerContext exists. There are two scenarios of creating TracerContext:

- In the SOFA MVC system, TracerContext is created as an entry for each access. If you want to activate SOFATracer for MVC requests, configure mvc\_toolbox\_plugin=ACTIVE in the sofa-config file to activate the MVC plug-in.
- In the SOFA CORE system, TracerContext is created when middleware (such as RPC) is called for the first time.

#### Configure penetration data

Sample code:

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

Note: TracerException is generated in the following three cases:

- The current input key or value is null.
- The current SOFATracer context is null.
- The size of penetration data exceeds the maximum limit (1024 characters at present).

### Description about the key:

The key of the API is shared globally and therefore must be unique. Two keys are reserved in SOFATracer: uid and mark.

#### Obtain penetration data in SOFATracer

Sample code:

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

Note:

- The null string is returned if the current SOFATracer context is not null but the penetration data obtained from the SOFATracer context based on the key is null.
- The penetration data is returned if neither the current SOFATracer context nor the penetration data obtained from the SOFATracer context based on the key is null.
- An exception is generated if the key is null.
- An exception is generated if the current SOFATracer context is null.

#### Clone the SOFATracer log context of the current thread

If the business system has its own thread pool for transaction processing, developers may need to obtain the SOFATracer log context of the current thread and incorporate it in the child thread. In versions earlier than SOFATracer 1.0.15, the SOFATracer context of the parent thread is obtained through an API and then incorporated in the child thread. In this way, the parent thread and child thread share one SOFATracer log context. In this case, modification of one thread may affect the other thread.

Therefore, SOFATracer 1.0.15 provides an API for cloning the SOFATracer log context of the current thread. If the SOFATracer log context needs to be incorporated in the child thread, the context can be copied from the parent thread through the API to avoid the mutual interference between them. The sample code is as follows:

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

Note:

- The null string is returned if the SOFATracer log context of the current thread is null.
- A clone of the SOFATracer log context of the current thread is returned if the context is not

null.

Full-process utility class

The LoadTestUtil class provides a static method for identifying full-process stress testing.

Definition: public static boolean isLoadTestMode()

Note: **This class determines whether the current test is full-process stress testing.** The value **false** is returned if no log context can be obtained from the current thread context. The value **true** is returned if the log context is obtained and the stress testing flag is T. Otherwise, **false** is returned.

When full-process stress testing is enabled, the log file is generated in the `${logging.path}/logs/tracelog/shadow/` directory, and the penetration data fields in the log file contain `mark=T`.

4.16 Tracer DRM switch

Dynamic configuration items

SOFATracer generates the detailed logs and statistical logs for various middleware calls by default. Despite this, SOFATracer has minor influence on the performance because the logs are generated asynchronously. However, SOFATracer provides a dynamic configuration switch to allow you to disable the middleware digest logs as needed.

SOFATracer provides dynamic configuration switches in two dimensions:

- Global switch
- Application-level switch

Global switch

With the global switch, the microservice can be globally pushed to all systems using SOFATracer. The global switch is enabled in the dynamic configuration background. The ID of the dynamically configured resource is:

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

The following resource attribute is contained:

```
disableMiddlewareDigestLog
```

The push values for the attribute are described as follows:

- **true**: Disable the digest log feature of all applications.
- **false** or **null string**: Enable the digest log feature of all applications.
- **log1=true&log2=true**: Disable the digest log feature of a specific application. log1 and log2



are the keys of the logs. You can list multiple types of logs to be disabled by using an ampersand (&) between each. Currently, the following keys are available:

- rpc-client: the digest log feature of RPCClient
- rpc-server: the digest log feature of RPCServer
- msg-publisher: the digest log feature of MSG Publisher
- msg-subscriber: the digest log feature of MSG Subscriber
- http-client: the digest log feature of HTTPClient
- mvc: the digest log feature of MVC
- zdal-db: the digest log feature of ZDALDB
- zdal-tair: the digest log feature of ZDALTAIR

### Push example:

```
// Disable the digest log feature of all applications.
true

// Enable the digest log feature of all applications.
false

// Disable the digest log feature of ZDALDB.
zdal-db=true

// Disable the digest log feature of ZDALDB and RPCClient.
zdal-db=true&rpc-client=true
```

To enable the digest log feature that has been disabled for an application, remove the corresponding push value, for example:

```
// The digest log feature of ZDALDB and RPCClient has been disabled.
zdal-db=true&rpc-client=true

// To enable the digest log feature of RPCClient, set the push value as follows:
zdal-db=true
```

### Application-level switch

The attribute of the application-level switch is similar to that of the global switch, except for the dynamic configuration ID.

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

The app-level switch is not configured in the dynamic configuration background by default. However, you can enable dynamic configuration for an application as required. The procedure is as follows:

Introduce the following dependency in the local SOFABoot project:

```
<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. Configure the dynamic configuration resource bean, for example:

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

1. Log on to the console, choose **Microservice platform > Microservices > Dynamic configuration**, add application metadata, and then add and push corresponding dynamic configurations. For more information, see [Start to use dynamic configuration > Manage and control the configuration class on the cloud](#).

4.17 Tracer log configuration items

After introducing SOFATracer in an application, you can add related configuration items to the application.properties file of Spring Boot to customize the SOFATracer behavior.

You can configure logging.path in the application.properties file as the log output directory of SOFATracer, which is \${logging.path}/tracelog. If logging.path is not configured, the default log output directory of SOFATracer is \${user.home}/logs/tracelog.

Configure Spring Boot project

SOFATracer configuration item	Description	Default value
logging.path	The log output directory.	SOFATracer preferentially generates logs to the logging.path directory. If no log output directory is configured, SOFATracer generates logs to the \${user.home} directory by default.
com.alipay.sofa.tracer.disableDigestLog	Specifies whether to disable digest log output for all SOFATracer components.	false
com.alipay.sofa.tracer.disableConfiguration[\${logType}]	Specifies whether to disable digest log	false

	output for the SOFATracer component with the specified <code>{logType}</code> . <code>{logType}</code> specifies the log type, for example, <code>spring-mvc-digest.log</code> .	
<code>com.alipay.sofa.tracer.tracerGlobalRollingPolicy</code>	The SOFATracer log rolling policy.	<code>.yyyy-MM-dd</code> specifies rolling by day. <code>.yyyy-MM-dd_HH</code> specifies rolling by hour. Rolling by day is not configured by default.
<code>com.alipay.sofa.tracer.tracerGlobalLogRetentionDay</code>	The SOFATracer log retention period in days.	SOFATracer logs are retained for 7 days by default.
<code>com.alipay.sofa.tracer.statLogInterval</code>	The statistical log output interval in seconds.	Statistical logs are generated every 60s by default.
<code>com.alipay.sofa.tracer.baggageMaxLength</code>	The maximum length of penetration data that can be stored.	The default value is 1024.
<code>com.alipay.sofa.tracer.zipkin.enabled</code>	Specifies whether to enable SOFATracer to remotely report data to Zipkin.	true: Enable data reporting. false: Disable data reporting. Data reporting is disabled by default.
<code>com.alipay.sofa.tracer.zipkin.baseUrl</code>	The address of Zipkin to which SOFATracer remotely reports data. This configuration item is valid only when <code>com.alipay.sofa.tracer.zipkin.enabled</code> is set to true.	The format is <code>http://{host}:{port}</code> .
<code>com.alipay.sofa.tracer.springmvc.filterOrder</code>	The order for the filters integrated by SOFATracer in Spring MVC to take effect.	-2147483647 ( <code>org.springframework.core.Ordered#HIGHEST_PRECEDENCE + 1</code> )
<code>com.alipay.sofa.tracer.springmvc.urlPatterns</code>	The URL patterns for the filters integrated by SOFATracer in Spring MVC to take effect.	<code>/*</code> All take effect.
<code>com.alipay.sofa.tracer.jsonOutput</code>	Specifies whether to generate logs in the JSON format.	The default value is <b>true</b> . To reduce the space occupied by logs, you can use a non-JSON format for log output. The log output sequence is the same as that of logs in the JSON format.

Configure a non-Spring Boot project

In a non-Spring Boot project, you can create a `sofa.tracer.properties` configuration file in the classpath directory. The configuration items are as follows.

SOFATracer configuration item	Description	Default value
<code>disable_middleware_digest_log</code>	Specifies whether to disable middleware digest logs.	false
<code>disable_digest_log</code>	Disables digest logs.	false
<code>tracer_global_rolling_policy</code>	The SOFATracer log rolling policy.	<code>.yyyy-MM-dd</code> specifies rolling by day. <code>.yyyy-MM-dd_HH</code> specifies rolling by hour. Rolling by day is not configured by default.

tracer_global_log_reserve_day	The SOFATracer log retention period in days.	SOFATracer logs are retained for 7 days by default.
stat_log_interval	The statistical log output interval in seconds.	Statistical logs are generated every 60s by default.
tracer_penetrate_attribute_max_length	The maximum length of penetration data that can be stored.	The default value is 1024.
tracer_async_appender_allow_discard	Specifies whether to allow loss of logs.	false
tracer_async_appender_is_out_discard_number	The number of lost logs.	0
spring.application.name	Application name.	-
tracer_sampler_strategy_name_key	The sampling policy name.	-
tracer_sampler_strategy_custom_rule_class_name	The fully qualified name of the class implemented by the sampling rule SPI.	-
tracer_sampler_strategy_percentage_key	The sampling ratio.	-
com.alipay.sofa.tracer.jsonOutput	Specifies whether to generate logs in the JSON format.	The default value is <b>true</b> . To reduce the space occupied by logs, you can use a non-JSON format for log output. The log output sequence is the same as that of logs in the JSON format.

4.18 Log format

SOFATracer is applicable to open-source components such as Spring MVC, database connection pools (DBCP, Druid, C3P0, Tomcat, HikariCP, and BoneCP) implemented based on standard JDBC interfaces, HttpClient, Dubbo, and Spring Cloud OpenFeign. You can view related SOFATracer logs after a component is connected with event tracking enabled. This document describes several types of logs and their formats.

- Spring MVC logs
- HttpClient logs
- DataSource logs
- SOFARPC logs
- OkHttp logs
- RestTemplate logs
- Dubbo logs
- Spring Cloud OpenFeign logs
- RPC-2-JVM log
- SOFARest logs
- SOFA MVC logs
- Error logs
- Static information logs

Spring MVC logs

After integrating Spring MVC, SOFATracer generates trace data for MVC requests in the JSON format by default.

Spring MVC digest logs (spring-mvc-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
request.url	The request URL.
method	The HTTP method.
req.size.bytes	The request size.
resp.size.bytes	The response size.
sys.baggage	The baggage data transparently transmitted on the system side.
biz.baggage	The baggage data transparently transmitted on the business side.

Example:

```
{
 "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 statistical logs (spring-mvc-stat.log)

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the **local.app**, **request.url**, and **method** fields.

Key		Description
time		The log print time.
stat. key	local.a pp	The name of the current app.
	reques t.url	The request URL.

	method	The request HTTP method.
count		The number of requests in a period.
total.cost.milliseconds		The total time cost of requests in a period, in ms.
success		The request result. Y indicates success. A result code starting with 1 or 2 indicates success, 302 indicates redirection success, and any other value indicates a failure. N indicates a failure.
load.test		The stress testing flag. T indicates stress testing and F indicates otherwise.

Example:

```
{ "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 logs

After integrating sofa-tracer-httpclient-plugin, SOFATracer generates trace data for HttpClient requests in the JSON format by default.

HttpClient digest logs (httpclient-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
request.url	The request URL.
method	The HTTP method.
req.size.bytes	The request size.
resp.size.bytes	The response size.
sys.baggage	The baggage data transparently transmitted on the system side.
biz.baggage	The baggage data transparently transmitted on the business side.

Example:

```
{
 "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": ""
}
```

Note: You can enter the app name as an input parameter when constructing the HttpClient instance by using SofaTracerHttpClientBuilder.

HttpClient statistical logs (httpclient-stat.log)

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the local.app, request.url, and method fields.

key		Description
time		The log print time.
stat.key	local.app	The name of the current app.
	request.url	The request URL.
	method	The request HTTP method.
count		The number of requests in a period.
total.cost.milliseconds		The total time cost of requests in a period, in ms.
success		The request result. Y indicates success. A result code starting with 1 or 2 indicates success, 302 indicates redirection success, and any other value indicates a failure. N indicates a failure.
load.test		The stress testing flag. T indicates stress testing and F indicates otherwise.

Example:

```
{
 "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 logs

SOFATracer provides event tracking for standard JDBC data sources and generates the trace data for SQL statement execution in the JSON format by default.

DataSource digest logs (datasource-client-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.

spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
database.name	The database name.
sql	The SQL statement.
connection.establish.span	The time when a connection is established for SQL execution.
db.execute.cost	The SQL execution time.
database.type	The database type.
database.endpoint	The database URL.
sys.baggage	The baggage data transparently transmitted on the system side.
biz.baggage	The baggage data transparently transmitted on the business side.

Example:

```
{
 "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 statistical logs (datasource-client-stat.log)**

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the local.app, database.name, and sql fields.

Key		Description
time		The log print time.
stat.key	local.app	The name of the current app.
	database.name	The database name.
	sql	The SQL statement.
count		The number of SQL executions in a period.
total.cost.milliseconds		The total time cost of SQL execution in a period, in ms.
success		The request result. Y indicates success, and N indicates a failure.
load.test		The stress testing flag. T indicates stress testing and F indicates otherwise.

Example:



```
{
 "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 logs

After being integrated in SOFARPC 5.4.0 or a later version, SOFATracer generates trace data in the JSON format by default. The fields are described as follows.

RPC client digest logs (rpc-client-digest.log)

Key	Description
timestamp	The log print time.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
local.app	The name of the current app.
protocol	The protocol (Bolt/REST).
service	The service API.
method	The method name.
current.thread.name	The name of the current thread.
invoke.type	The call type (sync/callback/oneway/future).
router.record	The router records (DIRECT/REGISTRY).
remote.ip	The destination IP address.
remote.app	The name of the target app.
local.client.ip	The local IP address.
result.code	The result code. 00 indicates success, 01 indicates a business exception, 02 indicates an RPC logic error, 03 indicates a timeout failure, and 04 indicates a routing failure.
req.serialize.time	The request serialization time, in ms.
resp.deserializ.e.time	The response deserialization time, in ms.
resp.size	The response size, in bytes.
req.size	The request size, in bytes.
client.conn.time	The client connection time cost, in ms.
client.elapse.time	The total time cost of calls (ms).
local.client.port	The local client port.
baggage	The transparently transmitted baggage data, in KV format.

Example:

```
{
 "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 server digest log (rpc-server-digest.log)

Key	Description
timestamp	The log print time.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
service	The service API.
method	The method name.
remote.ip	The source IP address.
remote.app	The name of the source app.
protocol	The protocol (Bolt/REST).
local.app	The name of the current app.
current.thread.name	The name of the current thread.
result.code	The result code. 00 indicates success, 01 indicates a business exception, and 02 indicates an RPC logic error.
server.pool.wait.time	The server thread pool waiting time, in ms.
biz.impl.time	The business processing time, in ms.
resp.serialize.time	The response serialization time, in ms.
req.deserialize.time	The request deserialization time, in ms.
resp.size	The response size, in bytes.
req.size	The request size, in bytes.
baggage	The transparently transmitted baggage data, in KV format.

Example:

```
{
 "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 client statistical logs (rpc-client-stat.log)

Key	Description
time	The log print time.
stat.key	The key of the log.
method	The method.
local.app	The name of the client app.
service	The service API.
count	The number of calls.
total.cost.milliseconds	The total time cost, in ms.
success	The call result. Valid values are Y and N.

Example:

```
{"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 server statistical log (rpc-server-stat.log)

Key	Description
time	The log print time.
stat.key	The key of the log.
method	The method.
local.app	The name of the client app.
service	The service API.
count	The number of calls.
total.cost.milliseconds	The total time cost, in ms.
success	The call result. Valid values are Y and N.

Example:

```
{"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 logs

After integrating OkHttp, SOFATracer generates the request trace data in the JSON format by default.

OkHttp digest logs (okhttp-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
request.url	The request URL.
method	The request HTTP method.
result.code	The HTTP status code.
req.size.bytes	The size of the request body.
resp.size.bytes	The size of the response body.
time.cost.milliseconds	The request processing duration, in ms.
current.thread.name	The name of the current thread.
remote.app	The target app.
baggage	The transparently transmitted baggage data.

Example:

```
{
 "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 statistical logs (okhttp-stat.log)

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the **local.app**, **request.url**, and **method** fields.

Key		Description
time		The log print time.
stat.key	local.app	The name of the current app.
	request.url	The request URL.
	method	The request HTTP method.
count		The number of requests in a period.
total.cost.milliseconds		The total time cost of requests in a period, in ms.
success		The request result. Y indicates success, and N indicates a failure.
load.test		The stress testing flag. T indicates stress testing and F indicates otherwise.

Example:

```
{
 "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 logs

After integrating RestTemplate, SOFATracer generates the request trace data in the JSON format by default.

RestTemplate digest logs (resttemplate-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
request.url	The request URL.
method	The HTTP method.
req.size.bytes	The request size.
resp.size.bytes	The response size.
sys.baggage	The baggage data transparently transmitted on the system side.
biz.baggage	The baggage data transparently transmitted on the business side.

Example:

```
{
 "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 statistical logs (resttemplate-stat.log)

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the **local.app**, **request.url**, and **method** fields.

Key	Description
-----	-------------

time		The log print time.
stat.key	local.app	The name of the current app.
	request.url	The request URL.
	method	The request HTTP method.
count		The number of requests in a period.
total.cost.milliseconds		The total time cost of requests in a period, in ms.
success		The request result. Y indicates success, and N indicates a failure.
load.test		The stress testing flag. T indicates stress testing and F indicates otherwise.

Example:

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

Dubbo logs

After integrating Dubbo, SOFATracer generates the request trace data in the JSON format by default.

Dubbo client digest logs (dubbo-client-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
protocol	The protocol.
service	The service API.
method	The call method.
invoke.type	The call type.
remote.host	The target host.
remote.port	The target port.
local.host	The local host.
client.serialize.time	The request serialization time.
client.deserialize.time	The response deserialization time.

req.size.bytes	The size of the request body.
resp.size.bytes	The size of the response body.
Error	The error message.
sys.baggage	The baggage data transparently transmitted on the system side.
biz.baggage	The baggage data transparently transmitted on the business side.

Example:

```
{
 "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 server digest logs (dubbo-server-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
protocol	The protocol.
service	The service API.
method	The call method.
invoke.type	The call type.
local.host	The local host.
local.port	The local port.
server.serialize.time	The response serialization time.
server.deserialize.time	The request deserialization time.
req.size.bytes	The size of the request body.
resp.size.bytes	The size of the response body.
Error	The error message.
sys.baggage	The baggage data transparently transmitted on the system side.

biz.baggage	The baggage data transparently transmitted on the business side.
-------------	------------------------------------------------------------------

Example:

```
{
 "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 statistical logs

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the **local.app**, **service**, and **method** fields.

Key		Description
time		The log print time.
stat.key	local.app	The name of the current app.
	method	The call method.
	service	The service name.
count		The number of requests in a period.
total.cost.milliseconds		The total time cost of requests in a period, in ms.
success		The request result. Y indicates success, and N indicates a failure.
load.test		The stress testing flag. T indicates stress testing and F indicates otherwise.

Example:

- 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 logs

After integrating Spring Cloud OpenFeign, SOFATracer generates the request trace data in the JSON format by default.



Spring Cloud OpenFeign digest logs (feign-digest.log)

The following table describes the keys in data generated in the JSON format.

Key	Description
time	The log print time.
local.app	The name of the current app.
traceId	The trace ID.
spanId	The span ID.
span.kind	The span type.
result.code	The status code.
current.thread.name	The name of the current thread.
time.cost.milliseconds	The span size.
request.url	The request URL.
method	The HTTP method.
Error	The error message.
req.size.bytes	The request size.
resp.size.bytes	The response size.
sys.baggage	The baggage data transparently transmitted on the system side.
biz.baggage	The baggage data transparently transmitted on the business side.

Example:

```
{
 "time": "2019-09-03 10:28:52.363",
 "local.app": "tracer-consumer",
 "traceId": "0a0fe927156747731347100110969",
 "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": ""
}
```

Spring Cloud OpenFeign statistical logs (feign-stat.log)

stat.key is the set of statistical keywords in a period, which uniquely determines a set of statistical data, including the **local.app**, **request.url**, and **method** fields.

Key		Description
time		The log print time.
stat.key	local.app	The name of the current app.
	request.url	The request URL.
	method	The request HTTP method.
count		The number of requests in a period.
total.cost.milliseconds		The total time cost of requests in a period, in ms.
success		The request result. Y indicates success, and N indicates a failure.

load.test	The stress testing flag. T indicates stress testing and F indicates otherwise.
-----------	--------------------------------------------------------------------------------

Example:

```
{ "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-2-JVM logs

RPC-2-JVM detailed logs (rpc-2-jvm-digest.log)

SOFATracer 1.0.16 provides the combined RPC-2-JVM detailed logs that contain the following information:

- The log print time.
- The name of the current app.
- The trace ID.
- The RPC ID.
- The service name.
- The method name.
- The target system name.
- The call duration.
- The name of the current thread.
- The system penetration data, in KV format. The data is used to transfer system disaster recovery information or other information.
- The penetration data, in KV format.

Example:

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

Note: The RPC-2-JVM detailed logs are disabled by default and can be enabled through DRM pushing. For more information, see SOFATracer DRM .

RPC-2-JVM call statistical logs (output every minute) (rpc-2-jvm-stat.log)

- The log print time.
- The source app, namely, the current app.
- The destination app.
- The service name.

- The method name.
- The number of calls in a period.
- The total time cost of calls in a period.
- The result. Valid values are Y and N. No statistical result is available for RPC-2-JVM, and the result is always Y.
- The full-process stress testing flag. Valid values are T and F.

Example:

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

SOFAREST logs

SOFAREST logs

The corresponding client and server execution logs are generated for each SOFAREST call in the following formats:

SOFAREST client logs

The SOFAREST client log file is named rest-client-digest.log, containing the following information.

Field	Description
time	The log print time.
local.app	The name of the current app.
traceId	The 25-character trace ID.
RpcId	The RPC ID of the trace.
request.url	The request URL.
method	The method name.
toApp	The name of the target app.
remote.ip	The IP address of the target machine.
result.code	The result code.
req.size	The data size of a request, in bytes.
resp.size	The data size of a response, bytes.
current.thread.name	The name of the current thread.
sys.baggage	The data recorded in the key-value format for transferring system disaster recovery information.
biz.baggage	The data recorded in the key-value format.

Example:

```
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 server logs

The SOFAREST server log file is named rest-server-digest.log, containing the following information:

Field	Description
time	The log print time.
local.app	The name of the current app.
traceId	The 25-character trace ID.
RpcId	The RPC ID of the trace.
request.url	The request URL.
method	The method name.
fromApp	The name of the source app.
Source URL	The IP address of the source app.
result.code	The return result code.
req.size	The data size of a request, in bytes.
resp.size	The data size of a response, in bytes.
current.thread.name	The name of the current thread.
sys.baggage	The data recorded in KV format for transferring system disaster recovery information.
biz.baggage	The data recorded in KV format.

Example:

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

SOFAREST client statistical logs

The SOFAREST client statistical log file is named rest-client-stat.log, which is generated every minute and contains the following information.

Field	Description
time	The log print time.
local.app	The name of the current app.
toApp	The name of the target app.
request.url	The request URL.
method	The method name.
count	The number of calls in a period.

total.cost.milliseconds	The total request processing duration in a period.
success	<ul style="list-style-type: none"><li>• Y: success</li><li>• N: failure</li></ul>
full-process stress testing	<ul style="list-style-type: none"><li>• T: The log context can be obtained from the current thread.</li><li>• F: No log context can be obtained from the current thread.</li></ul>

Example:

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

SOFAREST server statistical logs

The SOFAREST server statistical log file is named rest-server-stat.log, which is generated every minute and contains the following information.

Field	Description
time	The log print time.
local.app	The name of the current app.
toApp	The name of the target app.
request.url	The request URL.
method	The method name.
count	The number of calls in a period.
total.cost.milliseconds	The total request processing duration in a period.
success	<ul style="list-style-type: none"><li>• Y: success</li><li>• N: failure</li></ul>
full-process stress testing	<ul style="list-style-type: none"><li>• T: The log context can be obtained from the current thread.</li><li>• F: No log context can be obtained from the current thread.</li></ul>

Example:

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

SOFA MVC logs

SOFA MVC digest logs (sofa-mvc-digest.log)

- The log print time
- The name of the current app

- The trace ID
- The RPC ID
- The request URL
- The request method
- The HTTP status code
- The size of the request body
- The size of the response body
- The request processing duration, in ms
- The name of the current thread
- The penetration data, in KV format

Example:

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

**SOFA MVC statistical logs (sofa-mvc-stat.log)**

- The log print time.
- The name of the current app.
- The request URL.
- The request method.
- The number of requests in a period.
- The total request processing duration in a period.
- The request result. A result code starting with 1 or 2 indicates success, 302 indicates redirection success, and any other value indicates a failure.
- The stress testing flag.

Example:

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

**Error logs**

All error logs are recorded in the middleware\_error.log file by default.

**Basic format of general error logs**

Error logs usually contain the following information:

Field	Description
-------	-------------

time	The log generation time (UTC+8).
traceId	The 25-character TraceId. The first eight digits indicate the server IP address, the next 13 digits indicate the TraceId generation time, and the last four digits indicate an auto-increment sequence.
RpcId	The RPC ID of the trace.
failure type	The failure type is displayed according to the actual situation.
failure source	The failure source is displayed in the following data form: Failure source 1 Failure source 2 Failure source 3 ...
failure context	The failure context is displayed in the following mapping form: fromUser=khotyn&toUser=nytohk&
error stack	The specific error type.

Example:

```
2015-02-10
22:47:20.499,trade,q241234,0.1,timeout_error,trade|RPC,&protocol=&targetApp=¶mTypes=int|String&invokeT
ype=&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)
```

Middleware error logs

The middleware error logs are in the same format as that of general error logs except that the first item of the failure source field is the failed cluster. The following is an example:

```
Failed cluster | Other failure source 1 | Other failure source 2
```

RPC client error logs

The RPC client error logs are directly generated in the middleware\_error.log file basically in the same format as that of the middleware error logs, except for the following differences:

- The second item of the failure source field is RPC: Failed cluster|RPC|Other failure source 1|Other failure source 2
- The failure types are as follows:

- biz\_error: the business error
- address\_route\_error: the address routing error
- timeout\_error: the timeout error
- unknown\_error: the unknown error
- The failure context contains the following information:
  - serviceName: the service name
  - methodName: the method name
  - protocol: the protocol
  - invokeType: the call type
  - targetUrl: the target URL
  - paramTypes: the types of request parameters, in the format of param1|param2

Example:

```
2015-02-10
22:47:20.499,trade,q241234,0.1,timeout_error,trade|RPC,&protocol=&targetApp=¶mTypes=int|String&invokeT
ype=&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 server error logs

The RPC server error logs are directly generated in the middleware\_error.log file basically in the same format as that of the middleware error logs, except for the following differences:

- The second item of the failure source field is RPC: Failed cluster|RPC|Other failure source 1|Other failure source 2
- The failure types are as follows:
  - biz\_error: the business error
  - unknown\_error: the unknown error
- The failure context contains the following information:
  - serviceName: the service name



- `methodName`: the method name
- `protocol`: the protocol
- `invokeType`: the call type
- `callerUrl`: the caller's URL
- `callerApp`: the name of the caller's app
- `paramTypes`: the types of request parameters, in the format of `param1|param2`

Example:

```
2015-02-10
22:47:20.505,trade,q241234,0.1,unknown_error,trade|RPC,protocol=&callerUrl=¶mTypes=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)
```

### Static information logs

When a process starts, **static-info.log** of SOFATracer records static information about the process, including:

- Process ID
- Current IP address
- Current zone

Example:

```
84919,10.15.233.110,GZ001
```

