

# Apache SkyWalking in Action

YOUR GUIDE TO OBSERVABILITY AT SCALE



by Sheng Wu, Hong-tao Gao, Yi-xiong Cao,  
Yu-guang Zhao, & Can Li

Translated by Wing Wong



## **Apache SkyWalking in Action**

### **Your Guide to Observability at Scale**

Apache SkyWalking is a super-powerful observability platform and application performance monitor (APM) tool that was created to solve some of the thorniest problems faced by today's administrators: Identifying why and where a request is slow, distinguishing normal from deviant system performance, comparing apples-to-apples metrics across apps regardless of programming language, and attaining a complete and meaningful view of performance.

SkyWalking is one of the Apache Software Foundation's Top-Level Projects, and it provides a highly efficient, automated way to instrument microservices, cloud native, and container-based applications, with or without a service mesh. Its global community includes more than 200 code contributors. SkyWalking is in use at some of the world's largest companies, including Alibaba, Huawei, WeBank, youzan.com, and many more.

This book serves as a step-by-step guide to give readers a foundational understanding of distributed monitoring and tracing systems and a practical ability to navigate Apache SkyWalking. You'll learn about SkyWalking's core modules and the design concepts behind them so that you can get the most out of observing your system.

Use Apache SkyWalking for

- Monitoring the health of your services with distributed tracing collected with low payload;
- Observing your distributed system, on or off service mesh;
- Automating source code change and instrumentation: multiple language agents provided
- Advanced visualization: used in traces, metrics and topology map.

*"This book introduces the principles, applications and extensions of SkyWalking. Readers can get started immediately and apply SkyWalking in various use cases."*

**Lizan Zhou**

Founding Engineer, Tetrade.io,  
Core Maintainer of Envoy and Istio,  
Former Engineer, Google

"Apache SkyWalking is an open source platform for application performance monitoring and observability of distributed systems. It is widely used in the industry due to its sophisticated design and high efficiency. This book is first-authored by Sheng Wu, the founder of SkyWalking. It is informative, easy to use, and well written for learning and referencing."

**Jerry Tan Zhongyi**

Baidu Open Source Lead

"SkyWalking is an exemplary model of Apache in terms of both its product and community. It is a China-based project that has high international growth capabilities. It has been widely used in all walks of life and has become the fundamental support of many modern systems. In this book, readers will learn to use SkyWalking from scratch, master its core principles and design ideas, and benefit from experts' insights gleaned from years of experience in the community."

**Liang Zhang**

JD Digital Technology Center Architecture Expert,  
Vice President and Founder, Apache ShardingSphere

"Apache SkyWalking is Apache's top open source project. From data collection and analysis to visualization, it provides comprehensive solutions to address your needs. It is one of the first to provide support to the observability functions of open source projects such as Istio and MOSN. SkyWalking benefits from contributions from an active community and this book is the fruit of the community's experience. I highly recommend this book to those who want to learn about and research this subject."

**Jimmy Song**

Developer Advocate, Tetrade,  
Founder of ServiceMesher and the Cloud Native community

"Apache SkyWalking is an excellent open source APM solution that has become very popular with users. This book comprehensively introduces its usage and the design principles of various modules. It is highly recommended for microservice architects, developers, system operation and maintenance personnel, and readers interested in application performance management technology."

**Jiang Ning**

Huawei Open Source Competence Center Technical Expert, Apache Member

# Tetrate Service Bridge

You're holding the guide to observability at scale.  
Now, let Tetrate put you in control of your large, distributed system.

Whether you're connecting your microservices to monoliths for a fast and safe path to modernization or migrating your workloads to cloud, Tetrate Service Bridge gives you seamless and consistent observability, security, and management features across your application.

Tetrate Service Bridge (TSB) is an extensible, compute-agnostic, Zero Trust service mesh fabric built to fit your existing infrastructure.

Use the service mesh platform that combines the best of open source, including Istio, Envoy, and Apache SkyWalking: projects widely used and proven in production by the world's largest global companies.

Use Tetrate Service Bridge to observe, control, and secure all of your workloads, in any environment. Learn more at [www.tetrate.io](https://www.tetrate.io).

## Seamless.





# Apache SkyWalking in Action

Sheng Wu, Hong-tao Gao, Yi-xiong Cao,  
Yu-guang Zhao, and Can Li

Translated by  
Wing Wong



[www.tetrate.io](http://www.tetrate.io)

## **SkyWalking in Action: Your Guide to Observability at Scale**

By Sheng Wu, Hong-tao Gao, Yi-xiong Cao, Yu-guang Zhao, and Can Li

Translated by Wing Wong

Edited by Tevah Platt and Eileen AJ Connelly

Copyright © Sheng Wu, Hong-tao Gao, Yi-xiong Cao, Yu-guang Zhao, Can Li, Tetrade. All rights reserved.

Published by Tetrade

2020-10-26: E-book release

See <http://skywalking.apache.org/> and [www.tetrade.io](http://www.tetrade.io) for details and up to date information on Apache SkyWalking.

The Tetrade logo is a registered trademark of Tetrade.io

While the publisher and the authors have used good faith efforts to ensure the accuracy of the information contained within, the publisher and authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this information. Use of the information in this work is at your own risk. If any code samples or technical information in this work is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that you comply with appropriate licensing and rights.

## Overview

This book summarizes the use, project design, architectural modules and extensibility of Apache SkyWalking, and provides a comprehensive overview of Apache SkyWalking. There are 14 chapters in this book. The content of each chapter is outlined as follows:

<b>Preface</b>	7
<b>Chapter 1: Get to know Apache SkyWalking</b> This chapter outlines the use cases, background, and concepts of design in the SkyWalking project. Even though the development of open source projects is often unpredictable, understanding the core concepts is key to using SkyWalking successfully and to promoting consistency in the community in the long run.	10
<b>Chapter 2: SkyWalking installation and configuration</b> This chapter guides you from scratch to understand the installation and deployment process of the project, as well as the relevant configurations that are commonly used.	31
<b>Chapter 3: Live operations for Apache SkyWalking</b> In this chapter, the functions of SkyWalking are systematically presented through an illustration of a typical use case, so that readers can gain a comprehensive understanding of the system functions.	82
<b>Chapter 4: Lightweight-queue kernel</b> This chapter introduces the memory-based message queues in the SkyWalking Java agent, discussing its advantages, design goals, and implementation.	120
<b>Chapter 5: SkyWalking tracing model</b> This chapter introduces SkyWalking's proprietary tracing model. Although the project model was inspired by the landmark Google Dapper paper, its design for the distributed APM use case is remarkably different from other distributed tracing systems.	130
<b>Chapter 6: SkyWalking OAP Server modularized architecture</b> This chapter introduces the modularization of SkyWalking's Observability Analysis Platform (OAP). Modularization is a ubiquitous concept in the SkyWalking design. After reading this chapter, you will have a good understanding of the overall design concept of the OAP.	142
<b>Chapter 7: Observability Analysis Language system</b> This chapter introduces the Observability Analysis Language (OAL), a simple and compiled script language designed for SkyWalking, guides you through the SkyWalking stream processing model.	149

<b>Chapter 8: SkyWalking OAP Server cluster communication model</b>	
To help users plan network and deployment methods for a large-scale deployment, this chapter introduces the SkyWalking OAP cluster model, the data communication model and data flows in the cluster.	162
<b>Chapter 9: SkyWalking OAP Server storage models</b>	
SkyWalking is different from traditional applications due to its powerful expansion capabilities and flexibility. This chapter introduces you to the concepts of the defining methods, model fields and model extension methods of the data storage model with multiple examples.	168
<b>Chapter 10: Development of Java agent plug-ins</b>	
This chapter introduces the project structure, development methods, and development examples of SkyWalking's Java agent. Agent plug-in development is the most commonly used secondary development extension method.	180
<b>Chapter 11: Message communication mode between the agent and back-end</b>	
This chapter focuses on hands-on practice. The extension of agents and back-end communication modes is an important topic. The objectives for the project design will be elaborated and you will learn about how to implement code extension.	204
<b>Chapter 12: SkyWalking OAP Server monitoring and metrics</b>	
This chapter introduces you to an advanced feature: how to monitor the SkyWalking back-end. The monitoring system itself must also be monitored, which is a common feature of the large-scale deployment in SkyWalking.	236
<b>Chapter 13: The next-generation monitoring system -- SkyWalking observability for service mesh</b>	
Service mesh is still in the early stage of the development of technology stack, but SkyWalking is prepared to take the lead. This chapter guides you through the observability processes under the new architecture and technology.	243
<b>Chapter 14: Recent releases and the future of SkyWalking</b>	
SkyWalking 7 was released in March 2020 with many new features added. In this chapter, you will be introduced to method-level code performance analysis, a core feature of SkyWalking 7+.	250

# Preface

Written by Sheng Wu

## Why this book was written

Since 2010, distributed architecture has substantially transformed the entire IT ecosystem. The traditional SOA system as well as the more recent trend toward microservices, containerization, and Kubernetes, among others, have improved the fault tolerance and throughput capabilities of distributed systems. Nevertheless, the difficulty of monitoring increases with the extent of distribution. As a result, it is more and more difficult for the traditional monitoring system, which is based upon logs, indicators, and static deployment architecture, to keep up with the pace of system development.

From 2012 to 2015, I participated in China Unicom's first national centralized system and felt deeply troubled by the weak observability of the distributed system. It was at that moment I decided to start the SkyWalking project. As a community built upon a personal project, SkyWalking joined the Apache Incubator in late 2017. At the beginning of 2019, it quickly rose to become a top Apache project.

As a top Apache project, Apache SkyWalking has a strong developer community covering multiple countries around the world and is widely used by Fortune 500 companies. Currently, the industries using SkyWalking extend across a wide range of sectors, including internet services, ICT, banking, aviation, insurance, education, telecommunications, and electricity.

Until now, we haven't had a technical book available on the market that can systematically introduce Apache SkyWalking.

We hope this volume will help end users get going with Apache SkyWalking and to get the most out of its observability capabilities.



## Who this book is for

This book is suitable for all beginners, users, and secondary developers of Apache SkyWalking. It serves as a step-by-step guide, starting from the basics of the project, and moving deeper through design concepts and core modules. This book is your best choice if you want to systematically learn about Apache SkyWalking. If you would like to gain an understanding of modern distributed monitoring systems and distributed tracing, you will also find this book extremely helpful in terms of theoretical foundation and practical knowledge. If you want to build your own distributed monitoring system, Apache SkyWalking is a classic model and implementation worth studying.

## How to read this book

The chapters of this book are logically ordered in a way that gets you started, helps you apply the knowledge in practice, discusses the theory of design, and then goes into the core modules and advanced features. Readers will not be overwhelmed by too much theoretical content. At the same time, the in-depth discussion in the second half of the book steps up the basic use cases and explores possibilities for advanced implementation of the project. It even opens the door to participating in open source contributions.

## Errata and support

While we made every effort to ensure the accuracy of this book, the book was written under a tight time frame by a group of volunteers in the Apache SkyWalking community, and open source projects are diversified and rapidly changing. If you discover an error in the book or find that the content is no longer applicable to the new version available, we appreciate your feedback. Please submit your comments to [dev@skywalking.apache.org](mailto:dev@skywalking.apache.org) with [SkyWalking Book] as the subject line, or visit <https://lists.apache.org/list.html?dev@skywalking.apache.org> for any existing discussion threads of the relevant issues.

## Acknowledgements

First of all, I would like to thank the four co-authors of this book, Hong-tao Gao, Yi-xiong Cao, Yu-guang Zhao and Can Li for your continued contributions to the SkyWalking project and your great contributions to this book.

I am grateful for the unyielding support from my wife Ya-xin Liu and my family. I have invested a lot of time and energy in the past four years, from the establishment of SkyWalking to leading community development and participating in community contributions.

Thank you to hundreds of contributors, countless evangelists and bloggers in the Apache SkyWalking community, as well as producers of major conferences at home and abroad and major IT media for your

support in the project. You have helped spread the concept behind the project, attracted more newcomers, and made our community powerful.

Thank you to Fu-chuan Yang from Huazhang's China Machine Press for his assistance and support in the writing and publication of this book, allowing this book to be published with high quality.

Thank you to my colleagues at Tetrade for their support in the publication of this book. Thank you to Wing Wong for translating this book from Chinese to English. I gratefully acknowledge Tevah Platt and Eileen AJ Connelly for their assistance in the editing, and Aditi Khandewal for her coordination of the publication process. Their support allowed this book to be published successfully.

Sheng Wu  
October 2020

# Chapter 1:

## Get to know Apache SkyWalking

---

As of the publication of this book, SkyWalking is the first and only personal open source project in China that has developed into a top-level Apache project. As one of the industry's leading open source APM projects, Apache SkyWalking provides functional features that have only been offered by commercial APM or monitoring companies in the past. Having been applied in production and put to the test by a great number of enterprises, Apache SkyWalking is now widely used and backed up by a great many R&D as well as operation and maintenance teams.

This chapter provides an overview of the project itself, including its design goals and its development over time. Through this chapter, readers will gain a good understanding of the goals behind the SkyWalking project. Although this chapter does not discuss specific technical details or environment setup, it lays a foundation for learning and understanding the project one step at a time.

### 1.1 Introduction to SkyWalking

This section introduces the value, use cases and open source ecosystems of the SkyWalking project. This general description is designed to help you quickly understand the purposes behind these complex technologies.

#### 1.1.1 What is SkyWalking

SkyWalking is an application performance monitor (APM) and observability analysis platform designed for distributed systems. It offers multidimensional application performance analysis methods, including distributed topological maps, application performance metrics, traces, log correlation analysis and alarms.

The first thing to emphasize is that SkyWalking is aimed at microservices and distributed services, including services that are trending toward containerization. Such an environment entails a high level of complexity and variability in the dependencies between applications, making it impossible for design, development, operation and maintenance teams to understand the relationships between the systems and their operations in practice. The internal systems of major, large-sized enterprises often have scores of subsystems in which hundreds of services and thousands of instances would be running. Understanding the dependencies at this scale is the most important problem that SkyWalking is built to solve.

At the same time, with technological innovation and progress, the landscape of distributed frameworks is constantly changing. The use of multi-language RPC frameworks has become the general trend today, with Spring Cloud, gRPC, and Apache Dubbo as the most notable examples. Meanwhile, the direction of future development points toward service meshes, with Istio and Envoy as the key players. It is essential to set up a uniform monitoring platform to allow users to understand a highly complex distributed architecture.

Most importantly, SkyWalking guarantees availability under high-load conditions within production environments. Regular processing power at the level of tens of billions, lightweight process, pluggability, and easy customization are the primary reasons why SkyWalking has been so extensively used in just three years.

### 1.1.2 Developmental history of SkyWalking

The establishment and development of the SkyWalking project happened largely by chance in the early stage of development. If you look closely, you'll see huge differences in technology stack and design between the versions prior to the release of SkyWalking 3.2 and the updates following versions 5.x and 6.x. Here's why. When SkyWalking was established and became open sourced in 2015, it was a training project developed for distributed systems and designed to assist new employees of the company to learn about the complexity of distributed systems and how to build a monitoring system.

SkyWalking 3.2.x was the first milestone version. It introduced a design concept with a lightweight architecture as its key feature, and discontinued the use of big data storage technologies such as HBase. At that time, the SkyWalking multi-language agent protocol 1.0 was also introduced, and has since been supported by SkyWalking.

In December 2017, SkyWalking became the first personal project in China to enter into the Apache Incubator. This shows that the project community was ready to become part of the Apache Incubator and its prospect for the future was promising.

The project moved fast in 2018. In that year, the project team released SkyWalking 5, for which Huawei, Alibaba and other major manufacturers had shown their support, and this is when it started being widely used. By the end of 2018, the SkyWalking community welcomed the first sub-project of the ecosystem, the SkyWalking .NET Core agent. This marked the general acceptance of SkyWalking's tracing and header protocols into the community. On the basis of these protocols, construction of the architectural ecosystem began.

To facilitate the development of service mesh, which is seen as the next-generation distributed network architecture, SkyWalking 6 was released in 2019. Based on the experience, needs, and future plans in the development of the open source community over the past three years, and through a large number of top-level designs, SkyWalking 6 focuses on being protocol-oriented, lightweight, and modular, and provides a uniform solution for traditional probe monitoring and service mesh.

Many functional and design features of SkyWalking 6 carried over to 2020, as the community launched SkyWalking 7 and 8.

The development of community-based open source projects necessarily involves the participation of contributors and the growth of the project community. The number of code contributors to SkyWalking's main repository has jumped from 2 to more than 210 as of this writing. The number of GitHub stars in the project goes well over 10,000, making it the highest-ranked open source distributed tracing and APM project on GitHub.

### 1.1.3 Use cases for SkyWalking

SkyWalking is a component-based APM project designed for microservices, containerization and distributed systems. As early as in 2010, a time when service-oriented architecture (SOA) emerged, application system developers noticed that their debugging processes were increasingly complex. This made it very difficult to troubleshoot problems simply with the use of traditional logs in the event of the failure of web-based programs. Later, with the rise of microservices and service mesh, distributed architectures became widely adopted, such that there was a pressing need for enhanced program performance monitoring and troubleshooting.

This is where the SkyWalking project comes in. Inspired by Google's [Dapper paper](#) (2010) (the "Dapper paper") and assimilating the project's founding members' experience in APM solutions and Internet companies, SkyWalking introduces an application performance monitoring solution based on distributed tracing. At the same time, in view of the high traffic of business and characteristics of the R&D teams in China, SkyWalking first put forward the proposition of supporting 100% trace sampling in a high-traffic production environment. Currently, SkyWalking is the only APM system that supports this.



### ***A. SkyWalking is not just a tracing system***

As discussed in section 1.1.1, SkyWalking is first and foremost an application performance monitoring tool dedicated to application performance. Despite the fact that many regard SkyWalking, Zipkin, and Jaeger as competitors in the field of open source projects, the core members of these three communities in fact look at it differently.

In the context of language agents, SkyWalking has the functional feature of distributed tracing designed for application performance monitoring. It offers a high performance process and a solution for auto instrument agents; it also supports functions such as lightweight analysis of topological maps and application performance metrics. On the other hand, Zipkin and Jaeger are focused on the aspect of tracing in order to have as many tracing details as possible, while providing the recommendation of sampling during high level flows. With more experience in using both projects, the difference between the tracing structures of the two systems would become quite evident.

### ***B. SkyWalking is not an APM system based on big data***

The South Korean company Naver's APM open source project in 2012, Pinpoint, must not be left out in the discussion about APM. Pinpoint used to be the APM project with the most stars on GitHub before SkyWalking took its place in 2019. At first glance, Pinpoint and SkyWalking may feel functionally similar because both are part of the APM scene. But there are in fact essential differences between the two in terms of their use of the technology stack: while Pinpoint is based on HBase, SkyWalking uses a variety of storage methods, such as Elasticsearch, without supporting any big data technologies.

SkyWalking completely got rid of the big data stack since version 3. As one of the core systems of Ops, lightweightness and flexibility are on top of its list of concerns. Given that SkyWalking is designed to meet the basic requirement to monitor traffic at the level of hundreds of billions, it must make the deployment, operation and maintenance of the large-scale distributed systems easier, not harder. Were big data technologies adopted, it would only increase the difficulty of operation, maintenance and deployment.

At the same time, SkyWalking is designed to support the target system running in over 5,000 transactions per second (tps), which distinguishes it from Pinpoint. This is reflected in both the official testing and a great many performance comparisons on third-party blogs. At the initial stage of design for SkyWalking, a key concern of the team was to ensure that the agents would consistently provide 100% trace sampling and reasonable performance overhead (at less than 10%). Another notable specification is the ability for SkyWalking to fully control its own stack and computing model in order to match up to the APM computing requirements.

Apart from drawing comparisons to similar projects, there are major use cases that will help readers understand SkyWalking better.

### *C. SkyWalking is not a diagnostic system*

First of all, from a technical standpoint, the SkyWalking stack can trace the performance of every method. However, such use is not officially recommended, especially in a production environment. The reason is that tracing every method is the job of the performance diagnostic tool, not the APM system. The APM system requires long-term operation with low performance overhead under any given production environment. Since method monitoring expends a lot of memory and performance capacity, it is not suitable for high-traffic systems.

Of course, SkyWalking understands that usage scenarios vary between different teams, and hence provides method-level tracing in an optional plug-in for those using Spring framework hosting.

As an APM system, SkyWalking does not recommend method-level diagnosis with regularly added spans. Instead, it offers a more efficient and reasonable method of performance analysis, which is used for performance diagnostics in the production environment. To learn about this new feature of SkyWalking 7+, please refer to section 14.2.

### *D. SkyWalking can trace method parameters*

Parameter tracing is similar to method tracing. Even method parameter tracing for HTTP requests will put considerable pressure on application systems and APM. Although some plug-ins of SkyWalking (such as MySQL) allow users to manually enable parameter tracing, users are advised to continue keeping an eye on the performance overhead.

In addition, SkyWalking 7 introduced a performance diagnosis function through which method parameters are automatically captured.

## 1.1.4 The community and ecosystem of SkyWalking

The SkyWalking community is constantly growing from the Apache SkyWalking project and its subprojects. The relevant repositories are as follows.

Core code repository:

<https://github.com/apache/skywalking>

<https://github.com/apache/skywalking-rocketbot-ui>

<https://github.com/apache/skywalking-nginx-lua>

Protocol repository:

<https://github.com/apache/skywalking-data-collect-protocol>

<https://github.com/apache/skywalking-query-protocol>

Project official website source code hosting repository:

<https://github.com/apache/skywalking-website>

Container related repository:

<https://github.com/apache/skywalking-docker>

<https://github.com/apache/skywalking-kubernetes>

All codes and documentation of SkyWalking use the GitHub Tag to identify different versions. The documentation and codes of SkyWalking 6.0.0-GA are respectively as follows:

<https://github.com/apache/skywalking/tree/v6.0.0-GA>

<https://github.com/apache/skywalking/blob/v6.0.0-GA/docs/README.md>

The documentation and codes of SkyWalking 6.1 are respectively as follows:

<https://github.com/apache/skywalking/tree/v6.1.0>

<https://github.com/apache/skywalking/blob/v6.1.0/docs/README.md>

Users may download and use all published codes and corresponding documentation with simple steps.

In addition, the SkyWalking project also involves non-Apache organizations in the ecosystem, consisting of SkyAPM and SkyAPMTest.

- SkyAPM: The project address is <https://github.com/SkyAPM>; the following documents are all compatible with the SkyWalking ecosystem, which includes .NET, Node.js, PHP, GoLang, project documentation in Chinese and the Java non-Apache protocol-compatible plug-in.
- SkyAPMTest: The project address is <https://github.com/SkyAPMTest>; it includes the testing code or tools not yet available from Apache's official repository, as well as the sample codes used by the SkyWalking team in international conferences.

## 1.2 The architectural design of SkyWalking

As shown in Figure 1-1, the official architectural diagram provided by SkyWalking provides a self-explanatory overview of the overall architecture of SkyWalking. SkyWalking consists of four key parts below.

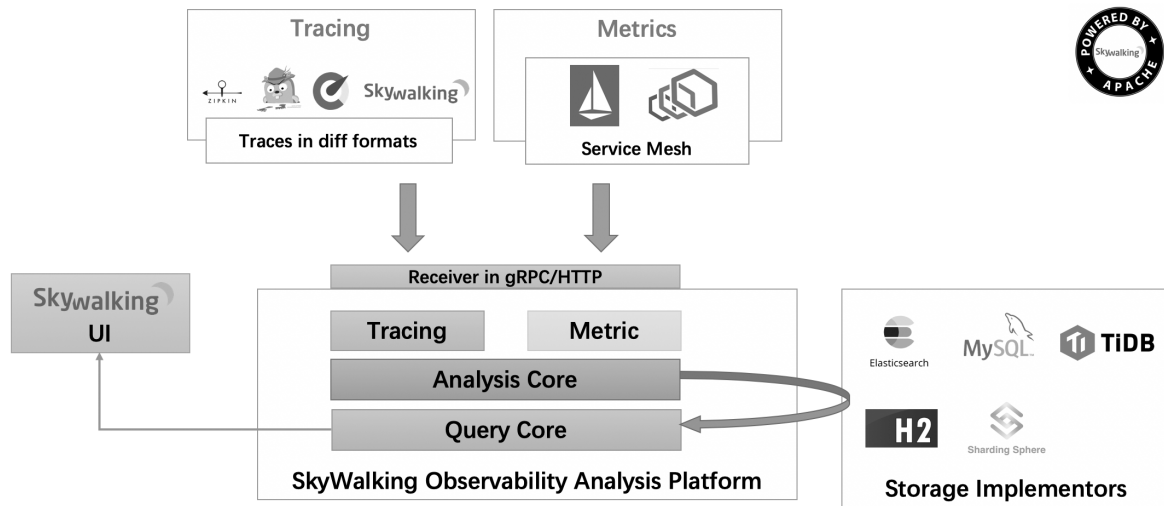


Figure 1-1 SkyWalking official architectural diagram

- **Agent:** The agent (corresponds to “Tracing” and “Metrics” in Figure 1-1) could either be a language agent or a protocol of other projects.
  - **Observability Analysis Platform (OAP), also known as OAP Server:** A highly modularized lightweight analysis program, which is composed of three parts, namely a receiver compatible with various agents, a stream-processing kernel, and a query kernel.
  - **Storage Implementors:** SkyWalking's OAP Server supports multiple ways of storage implementation, and provides a standard interface for implementation of other storage.
  - **UI module (SkyWalking):** Query and display statistical data through standard GraphQL protocol.
- SkyWalking generally follows the three key design principles below:

- Protocol-oriented design
- Modularized design
- Lightweight design

### 1.2.1 Protocol-oriented design

SkyWalking strictly adheres to a protocol-oriented design ever since version 5.x. SkyWalking has set out the following open protocols.

### A. Agent protocol

Agent protocols belong to four categories.

- Language agent report protocol: This protocol includes the registration of language agents, metrics data reporting, tracing data reporting, downstream commands, and the telemetry protocol used in service mesh. All language-based agents (such as Java, .NET, Node.js, PHP and GoLang) must adhere to this protocol definition. This protocol has become open in the form of gRPC services since the update of version 6.
- Language agent interaction protocol: Due to the nature of distributed tracing, agents must interact through service-to-service communication channels such as HTTP headers and MQ headers. This protocol defines the format of such interactions. All language-based agents (such as Java, .NET, Node.js, PHP and GoLang) must adhere to this protocol definition. Starting from v2, this protocol has adopted a different coding method from v1, with the addition of the mandatory requirement of Base64. The new v3 protocol is executed in SkyWalking 8, to allow for more simplified coding and new functionalities, such as the transparent transmission of business information and the ability to conduct agent interaction.
- Service mesh protocol: This protocol is a proprietary protocol abstracted by SkyWalking for service mesh. Any mesh-type services may directly upload telemetry data through this protocol for the purpose of computing service metrics and topological maps.
- Third party agreement: For large-scale third-party open source projects, especially key service mesh platforms Istio and Envoy, core protocol adaptation is provided. In particular, it supports seamless monitoring for the Istio and Envoy service mesh.

### B. Query protocol

The query protocol used is defined in the GraphQL format. Many users would be more familiar with query in the RESTful format. But taking into account better scalability and the flexibility of combined query, SkyWalking selected the open source language GraphQL, which was developed by Facebook in 2012. GraphQL has been extensively used in both open source and commercial projects. The pre-defined protocol format and the combined use of multiple queries translate into excellent integration capabilities between UI and third-party systems. In the version updates that followed SkyWalking 6.0.0-GA, the RocketBot UI has become the default UI. Such an update benefits from the flexibility of GraphQL, and during that process, the back-end protocol and implementation do not even have to be changed. Many companies in the community have designed the UI of cloud platform products based on this protocol.

There are six types of query protocol.



- Metadata query: query on registered services on SkyWalking, service instances, endpoints and other metadata information.
- Topological relationship query: query on the overall services or a single service, or the topological maps and dependency relationships of endpoints.
- Metrics index query: query on linear index.
- Aggregate index query: query on the mean value of a range and TopN, etc.
- Trace query: query on trace details.
- Alarm query.

In addition to the two major types of protocols, there are also intra-module protocols within some modules, such as exported data format protocols, alarm protocols, and dynamic configuration service protocols. These protocols are related to the default implementation of the execution modules and come under the scope of SkyWalking's default external integration protocol. With a modular design, these protocols may also be replaced, though for brevity the details are not discussed here.

### 1.2.2 Modular design

With open source projects as its central focus, the modular design aims to provide plug-in mechanisms and extension points for more customized services for companies, as well as secondary packaging and plug-in mechanisms for commercial products. It is very important for open source projects to carry out uniform upgrades. Modular design and clear interface boundaries make it easy to expand and keep up with the pace of open source version upgrades. This is an indispensable approach in the development of commercial packaging and open source products.

Large open source projects are all pushed forward by a great many businesses or for commercial purposes. It is impossible for individuals to complete them in their spare time. Although the SkyWalking project progressed out of personal projects in its first two years, it has had to adopt an open, inclusive, and scalable architecture to accommodate its fast-growing community and meet commercial demand. As a top-level Apache project, Apache SkyWalking has evolved from the business-friendly Apache 2.0 open source protocol. In the design, the possibility of customization and room for secondary development have also been thoroughly considered.

SkyWalking has made use of two different modularity mechanisms according to the different characteristics of the agent and the server.

At the Java agent's endpoint, Java adopts a more compact implementation, with the use of SPI to isolate the core services into a replacement state. Users may simply place the new service implementation in the plug-in folder to implement automatic replacement, very much similar to developing a plug-in.

The back-end (OAP Server) adopts the modularity as defined by YML. SkyWalking defines modularity as consisting of two parts, namely module and provider. A *module* is an abstract concept that provides definitions and declarations of open service methods. A *provider* needs to implement these service methods and declare other modules upon which this implementation depends. It is worth noting that a module itself does not declare dependencies. Dependencies are instead declared by the provider. This allows users to replace and add required modules, and then organize them.

### 1.2.3 Lightweight design

SkyWalking put forward a lightweight design concept at the initial stage of the design. Although APM is the core system for the operation and maintenance end, it comes second in line as a support system under the overall business structure and does not conduct the main business functions of the system. The vast majority of analysis systems require big data as their core technology, but there are problems with big data, such as high maintenance difficulties and high entry barriers. Against this background, the key concern for SkyWalking is to use the most lightweight model of Jar under the non-big data architecture to generate powerful analytical, expansion and modular capabilities.

## 1.3 Advantages of SkyWalking

The main advantage of SkyWalking is that it keeps up with current technological developmental trends and ensures that the same APM system is suitable for both traditional and cloud-native architectures. In addition, in terms of technical design concepts, SkyWalking provides greater compatibility and scalability, and is suitable for different user scenarios and customization requirements.

### 1.3.1 Uniformly supports traditional distributed systems framework and cloud-native infrastructure

In light of the development of services and microservices in the last decade, a framework that centers around the RPC and HTTP services as core communications technologies and a service registry dedicated to service registration and discovery has become the fully fledged “tradition” of microservices solutions. Some of the best known technologies are Spring Cloud and Apache Dubbo. When the project began to take shape in 2015, SkyWalking had already made such traditional distributed systems framework and auto-instrumentation agents its core features. SkyWalking could seamlessly support a developed distributed service framework, which allows users to easily shift to a traditional monitoring approach, without adding to the workload of the operation and maintenance teams.

In the meantime, since 2018, service mesh solutions formulated by Google, Lyft and the Cloud Native Computing Foundation (CNCF), and particularly the de facto standard open source service mesh projects Istio and Envoy, have grown in popularity. A service mesh provides an innovative way to secure, monitor, and manage microservices in a distributed environment, including workloads on Kubernetes. The SkyWalking team had been closely following this development and subsequently released SkyWalking 6 beta version when Istio 1.0.4 came on the market. Three months later, the stable update of SkyWalking 6 was released, providing core adaptive abilities catered specifically for the Istio and Envoy service mesh.

To put it another way, the service mesh has become part of a new language-neutral agent instrumentation for SkyWalking. With the use of SkyWalking's back-end OAP platform and UI, it can provide the same topological dependencies, service performance metrics and alarm settings for all services managed within the service mesh. More than 90% of the configuration is exactly the same as when agents of other languages (such as the Java agent) are used.

This is also the first ever solution that unifies agent languages and service mesh in open-source softwares. In enabling uniform monitoring capacity for the technology stack of different companies, these companies could better ensure consistency in their monitoring systems in future infrastructure upgrades. Such consistency concerns not only the operations of SkyWalking, but also the ecosystems built into SkyWalking by users themselves, such as alarm platforms, AIOps, baseline computing systems and elastic computing.

### 1.3.2 Easy to maintain

SkyWalking always adheres to the core principle of easy maintenance for all. It avoids bringing in excessive technology stack, which would otherwise make monitoring systems overly complex. The reason is that monitoring systems usually come second or even third in line in terms of system development priority. The best possible monitoring value should be squeezed out of the limited resources available in the environment. At the same time, requirements for operation and maintenance should be kept as low as possible. Under the cluster model of SkyWalking, many companies are collecting monitoring data and details at the level of tens of billions and beyond every day. In order to make it more user-friendly and easier to maintain, SkyWalking does not require the use of complex big data platforms.

At the same time, the SkyWalking cluster architecture is relatively simple. Users need only focus on their own data volume and prepare for a basic cluster operation and maintenance capability for different storage platforms (such as MySQL, TiDB and Elasticsearch). In this way, users could easily monitor traffic systems at the level of class of tens of billions.

### 1.3.3 High performance

SkyWalking will not reduce performance requirements because of the pursuit of simplicity and ease of maintenance. SkyWalking has a built-in scalable stream computing framework specifically designed for distributed monitoring (see Chapter 7). This computing framework has designed specific processes for monitoring data, and with the use of bytecode it is able to strike a good balance between scalability and performance.

In a typical case scenario of the Chinese chain supermarket operator Yonghui Superstores, 15 stations of OAP nodes and 20 stations of Elasticsearch nodes were able to support 250 types of services with up to 3TB monitoring data a day and more than ten billion of data traffic. Among the 6.x versions, the performance of SkyWalking has been significantly improved in each update from 6.0-GA to 6.4.

### 1.3.4 Favorable for secondary development and integration

There are two aspects to why SkyWalking is by nature favorable to secondary development and integration:

- Protocol-oriented and modular design. Being protocol-oriented ensures that it is easy for other agents to integrate as long as they learn the protocols. Modularization gives users great flexibility for customization. The switchable characteristic of module implementation allows users to personalize the distributed computing process, mode of cluster management and coordination, storage, alarm engine, and visualization sites.
- A large number of built-in SkyWalking implementations provide third-party network interfaces, HTTP or gRPC interfaces. Instead of modifying an existing program, users may use third-party programs for integration. This ensures the stability of the ecosystem during SkyWalking upgrades. The manner of network interface integration has also become more friendly with the growing significance of containerization.

Hundreds of public users of SkyWalking have used these extension methods in wide-ranging scenarios, customized elaborate internal systems, and ensured the stability and high versatility of the SkyWalking kernel.

## 1.4 Introducing essentials on the development of SkyWalking

- How does the SkyWalking Java agent's end achieve non-intrusive embedding?
- How to perform remote debugging during development or network troubleshooting?
- What is the latest service mesh supported by SkyWalking?
- With the above questions in mind, let's begin this section.

## 1.4.1 Introduction to JavaAgent

### A. Concept overview

The SkyWalking agent is a codeless automatic embedding solution based on Java's JavaAgent technique.

The JavaAgent loaded at startup (the "JavaAgent" hereafter refers to the JavaAgent loaded at startup) is a new feature introduced after JDK 1.5. This feature provides users with the ability to modify the bytecode after the JVM reads the bytecode file into the memory and before generating a Class object in the Java heap using the corresponding byte stream, and the JVM will also use the modified bytecode modified by users to create the Class object.

The SkyWalking agent relies on JavaAgent to intercept the corresponding bytecode data at specific points (certain methods of a certain class) and make AOP modifications. When the trace runs into a method that has been proxied by SkyWalking, SkyWalking will collect, transmit and report these key node information according to the proxy logic, through which the entire set of distributed traces may be restored.

### B. Create JavaAgent

The process of creating JavaAgent is as follows.

- 1) Write the `premain` startup program.

```
/**
 * @author caoyixiong
 */
public class SkyWalkingAgent {
    public static void premain(String args, Instrumentation instrumentation) {
        System.out.println("Hello, This is a SkyWalking Handbook JavaAgent demo");
    }
}
```

- 2) Write in `MANIFEST.MF`.

The `MANIFEST.MF` file is used to describe the information of the Jar files. In this file, we need to add the full path of the specified `premain` method.

Add the following code in `MANIFEST.MF`:

```
Manifest-Version: 1.0
Premain-Class: org.apache.skywalking.apm.agent.demo.SkyWalkingAgent
```

If the Apache Maven is used, the `premain` information must be declared in the corresponding POM files.

```
<build>
  <plugins>
```



```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.3.1</version>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
      </manifest>
      <manifestEntries>
        <Premain-Class>
          org.apache.skywalking.apm.agent.demo.SkyWalkingAgent
        </Premain-Class>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
</plugin>
</plugins>
</build>

```

After adding the relevant information, package it into a Jar file.

### 3) Write the test program.

```

package org.apache.skywalking.handbook.javaagent;
/**
 * @author caoyixiong
 */
public abstract class SkyWalkingTest {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("This is SkyWalkingTest main method");
    }
}

```

Add `-javaagent` to the JVM startup parameters of the test program: the absolute address of the Jar file associated with the `SkyWalkingAgent` referred to in Step 2.

### 4) Run the test program. The results are as follows.

```

Hello, This is a SkyWalking Handbook JavaAgent demo
This is SkyWalkingTest main method

Process finished with exit code 0

```

As you can see, there is no code modification or insertion to the test program, but the print result already contains the `JavaAgent` information.

The above example prints out a simple string. The sections below will demonstrate how to enhance the JavaAgent such that it could be used to measure method execution time.

## 5) Add ClassFileTransformer.

The main function of ClassFileTransformer is to modify the bytecode data loaded into the JVM through its transform method.

As an illustration, the tool used here will be the bytecode manipulation tool Javassist.

First, add the dependency of Javassist to the POM:

```
<dependency>
  <groupId>org.javassist</groupId>
  <artifactId>javassist</artifactId>
  <version>3.15.0-GA</version>
</dependency>
```

Then, create a new ClassFileTransformer class:

```
/**
 * @author caoyixiong
 */
public class SkyWalkingTransformer implements ClassFileTransformer {
    @Override
    public byte[] transform(ClassLoader loader,
        String className,
        Class<?> classBeingRedefined,
        ProtectionDomain
        protectionDomain, byte[]
        classfileBuffer) {
        // Only intercept the SkyWalkingTest test program
        if (!"org/apache/skywalking/apm/agent/demo/SkyWalkingTestt".
            equals(className)) {
            return null;
        }
        // Obtain Javassist ClassPool
        ClassPool cp =
            ClassPool.getDefault(); try {
            // Obtain SkyWalkingTest CtClass object in the ClassPool
            // (One-to-one relationship with the SkyWalkingTest Class object)
            CtClass ctClass = cp.getCtClass(className.replace("/", "."));
            // Find the corresponding main method
            CtMethod method = ctClass.getDeclaredMethod("main");
            // Add local variable - longType beginTime
            method.addLocalVariable("beginTime", CtClass.longType);
            // Insert 'Long beginTime = System.currentTimeMillis ();' before the main
            method
            method.insertBefore("long beginTime = System.currentTimeMillis();");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        // Print out the duration of execution time after
        the main method method.insertAfter (
        "System.out.print (\"duration : \");");
        method.insertAfter ( "System.out.println
        (System.currentTimeMillis()-beginTime);");
        // Return the modified bytecode data
        return ctClass.toBytecode();
    } catch (NotFoundException | CannotCompileException | IOException e)
    { e.printStackTrace();
    }
    // Return null, which means that the bytecode has not been modified
    return null;
}
}

```

6) Add the new **ClassFileTransformer** object to the **Instrumentation** instance of **JavaAgent**.

```

package org.apache.skywalking.handbook.javaagent;
import java.lang.instrument.Instrumentation;
/**
 * @author caoyixiong
 */
public class SkyWalkingAgent {
    public static void premain(String args, Instrumentation instrumentation) {
        System.out.println("Hello, This is a SkyWalking Handbook JavaAgent
        demo");
        instrumentation.addTransformer(new SkyWalkingTransformer());
    }
}

```

7) Repeat Step 2 to package the **JavaAgent** into a **Jar** file and load it into the **JVM** startup parameters of the test program, and then run it for the following results.

```

Hello, This is a SkyWalking Handbook JavaAgent demo
This is SkyWalkingTest main method
Total execution time: 0
Process finished with exit code 0

```

You can see the execution time of the main method.

Based on the characteristics of **JavaAgent**, the bytecode data may be modified during the bytecode load time to expand on the functional logic as desired. To clarify, the **SkyWalking** agent uses **Byte Buddy** as the bytecode manipulation tool for the agent rather than **Javassist**. **Javassist** is used here just to demonstrate the logic of manipulation of the bytecode more clearly.

### C. Processes and principles of JavaAgent

A JavaAgent that could measure execution time of the main method has been created in the preceding section. In this section, the processes and principles of the JavaAgent will be described. Figure 1-2 below shows an overview of the process of internal bytecode manipulation of a JavaAgent.

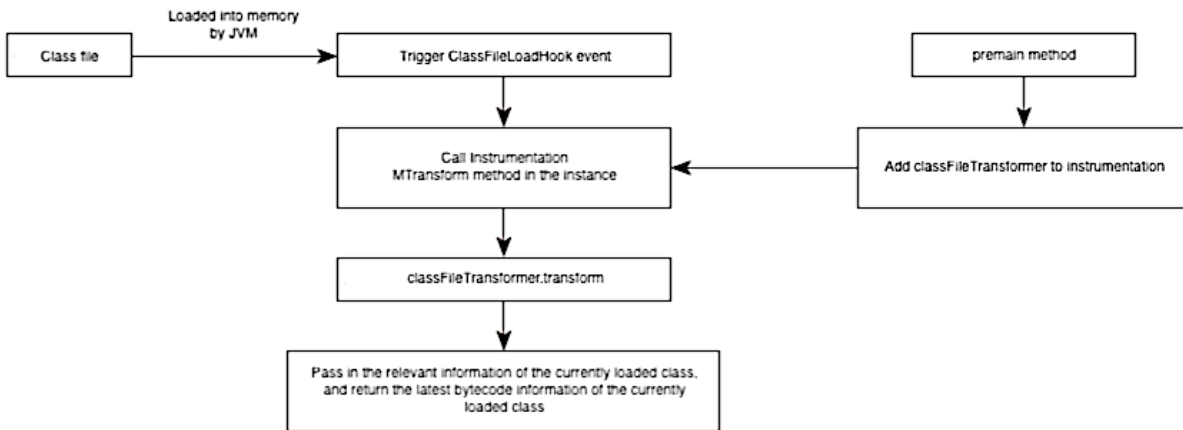


Figure 1-2 Process overview of JavaAgent internal bytecode manipulation

When the bytecode of a class is loaded into memory by the JVM, the JVM will trigger a `ClassFileLoadHook` event, and the JVM will traverse all the instrumentation instances in turn and execute all of the `transform` methods of the `ClassFileTransformer`.

A user-defined `ClassFileTransformer` instance has to be added onto the instrumentation instance in the `premain` method of JavaAgent.

The `premain` method and the `transform` method are two important methods in a JavaAgent.

The `premain` method is the entry point of JavaAgent. There are two ways to implement it. The codes are as follows.

```
public static void premain(String agentArgs, Instrumentation inst); //[1]
public static void premain(String agentArgs); //[2]
```

In the above codes, `agentArgs` is the `yyy` string passed along with `javaagent:xxx.jar=yyy`, and `Instrumentation` is the core class of assembly bytecode. If you wish to implement a JavaAgent that could modify the bytecode, you must apply the first method; if two methods are applied at the same time, only the first method will be executed.

The main function of the `transform` method is to transform bytecode data. The main input and output parameters are introduced below.

The main input parameters are as follows.

- `ClassLoader loader`: The `ClassLoader` of the currently loaded `Class`. If the `ClassLoader` is a bootstrap loader, it would be `null`.
- `String className`: The class name of the currently loaded `Class`. The name of the internal form of the fully qualified class and interface name defined in the Java virtual machine specification, such as `Java`, `util` or `List`.
- `byte[] classfileBuffer`: The bytecode data of the current class presented in a byte array. (Might be inconsistent with the data in class files, since the byte data here is the latest bytecode data of such type; in other words, the data might be the bytecode data after the original bytecode data is enhanced through other enhancement methods.)

The main output parameter is `byte[]`. If it is `null`, it means that the bytecode of the current class has not been modified; if it is not `null`, the JVM will use this bytecode data for subsequent processes, such as creating a `Class` object, initialization or proceed to the next transform method.

#### *D. Summary*

This section mainly introduces the technologies and principles related to `JavaAgent`, and guides the readers to write a simple `JavaAgent` program from scratch. The `JavaAgent` is a very important part of the `SkyWalking` operating mechanism. The non-intrusive nature of `JavaAgent` has set a strong foundation for the growth of `SkyWalking`.

### 1.4.2 Introduction to remote debugging

Remote debugging is used to debug application services deployed on a remote operating system. But first of all, there must be a local code that is synchronized with the remote application. It is very helpful to learn remote debugging, as it allows you to quickly understand live web applications in real time, which in turn facilitates problem solving within a short time. On top of that, remote debugging is regarded as the best way to debug agent code.

This section will explain the ways to enable remote debugging for the application service process, and how to locally debug the `SkyWalking Agent` code in a remote application.

#### *A. Enabling remote debugging in application service process*

It is very simple to enable remote debugging for application services. Just insert `-Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n` in the startup command.

The parameters are described as follows.

- `-XDebug`: Enable debugging.
- `-Xrunjdwp`: Load the JPDA reference implementation instance of JDWP.

- transport: Used for communication between the debugger and the process used by the remote application service.
- dt\_socket: Socket transmission.
- address=8000: The port number monitored by the remote debugging server.
- suspend=y/n: Suspend or start application services after the debug client establishes a connection.

Take the following startup command of the Spring Boot microservice as an example:

```
java -jar project.jar
```

The command to enable remote debugging is:

```
java -jar project.jar
Java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000, suspend=n -jar project.jar
```

This enables the remote debugging mode for the project application service.

#### a. Local debugging of remote SkyWalking Agent code

Open the code which is consistent with the SkyWalking Agent at the server end. Take the editor IDEA version 2019.2 as an example, open the Remote window by clicking Run → Edit Configurations → Remote, and you will see the interface shown in Figure 1-3.

The keywords are circled in the figure and described as follows.

1. Host, configure the remote server IP.
2. Port, the debugging port enabled by a remote application service, such as 8000.
3. Select the JDK version of the remote application service.
4. Select the local code module.

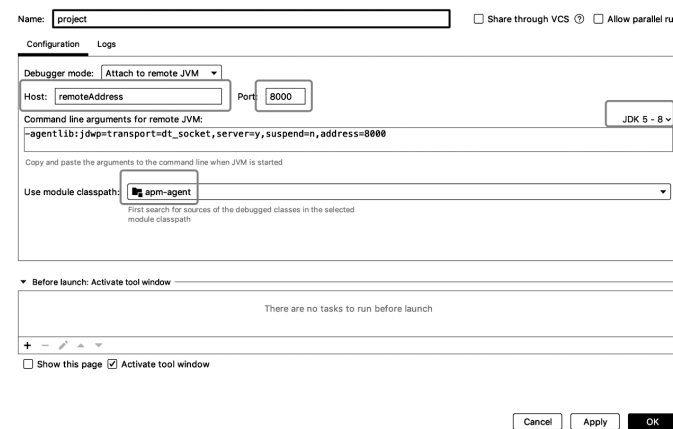


Figure 1-3 IntelliJ IDEA Remote window

Click “OK” to complete the configuration. Click “Debug” to begin remote debugging.

### 1.4.3 Introduction to service mesh

Service mesh is regarded as the infrastructure of the future generation of inter-service communications. A service mesh addresses observability, security, and resiliency issues that arise in a large distributed system by decoupling operations from development and controlling communications between services. For a basic introduction, see [The New Stack's "Primer: The Who, What and Why of Service Mesh"](#) and [Istio's documentation site](#).

Service mesh is an infrastructure dedicated to handling service-to-service communications. It reliably transmits requests through a complex service topology structure, and applications that deploy it generally have cloud-native features. In practice, a service mesh is usually implemented as a matrix of lightweight network proxies. These lightweight network proxies are deployed along with application code without the need to understand the implementation details of the application.

Service mesh has the following characteristics:

- The middle layer of communication between applications
- Lightweight network proxy
- No awareness of applications
- Various mechanisms for decoupling applications

At present, popular service mesh open source softwares, Istio and Linkerd, can both be directly integrated in Kubernetes or deployed separately in the VM.

SkyWalking is currently integrated with the service mesh scenario and has provided a comprehensive integration solution for Istio. A brief account on how Istio manages network traffic is given here. If you would like to learn more, please visit Istio's official website at <https://istio.io>.

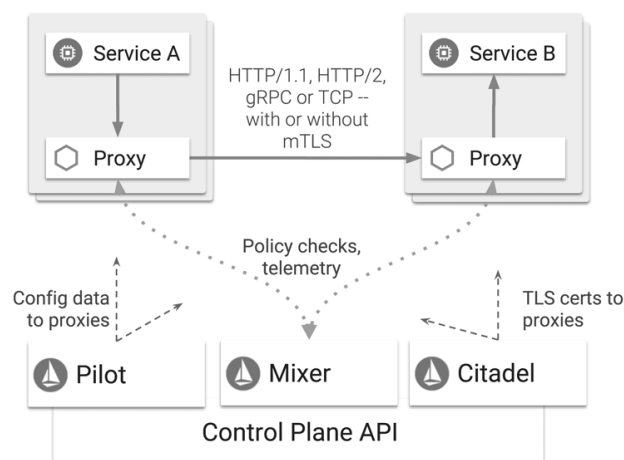


Figure 1-4 Istio architectural diagram

As shown in Figure 1-4, in order to direct traffic in the mesh, Istio needs to know the location of all endpoints and to which services they belong. In order to maintain its own service registry, Istio connects to a service discovery system. If, for example, Istio is installed on the Kubernetes cluster, it will automatically detect the services and endpoints in the cluster.

With this service registry, the Envoy proxy can redirect access requests to the relevant services. Most microservice-based applications have multiple instances to handle the service request, which requires the use of a load balancer. By default, the Envoy proxy uses a loop scheduling model to distribute traffic in each service and send requests to each member in turn.

In addition to basic service discovery and load balancing, Istio could also perform finer control over network traffic. For A/B testing scenarios, Istio could direct a specific proportion of traffic to a new version of the service, or apply different load balancing strategies to a specific subset of service instances based on the user's needs. For some scenarios not currently supported by Istio, the services may also be expanded through its extension points. As this part is not directly relevant to SkyWalking, the details are not discussed here.

## 1.5 Chapter summary

SkyWalking embodies both comprehensive features and cutting-edge technologies. This chapter provides an overview of the use cases, features and advantages of the SkyWalking project. It also allows readers to have an overall understanding of the project, and introduces readers to the several techniques most commonly used in the project development, which will facilitate further learning of the readers. The next chapter will be an introduction to the installation and use of the SkyWalking project to help you get started on SkyWalking.



## Chapter 2:

# SkyWalking installation and configuration

---

This chapter describes the compilation and project structure, as well as the deployment of the JavaAgent module, back-end module, UI module, and configuration information of the SkyWalking project. In this chapter, readers will learn how to build a simple SkyWalking environment and get a taste of the convenience and power of SkyWalking.

## 2.1 Project compilation and structure

This section introduces source code compilation and project structure to help readers better understand SkyWalking's source code structure.

### 2.1.1 Project compilation

#### *Build from GitHub*

The steps to build are as follows.

1. Prepare Git, JDK 8 and Maven 3.
2. Execute git clone in the terminal <https://github.com/apache/skywalking.git>.
3. Execute `cd skywalking/`.
4. Execute `git checkout [tagname]` to switch to the specific tag. (Optional, only if a specific version of the code to be compiled.)
5. Execute `git submodule init`.
6. Execute `git submodule update`.
7. Run `./mvnw clean package -DskipTests`.

All packaged und under the folder /dist (.tar.gz for Linux, .zip for Windows).

#### *A. Build from the source code released by Apache SkyWalking*

The steps to build are as follows.

- 1) Prepare Git, JDK 8 and Maven 3.
- 2) Download the source code of the relevant release version from the Apache SkyWalking official website.
- 3) Prepare JDK 8 and Maven 3.
- 4) Run `./mvnw clean package -DskipTests`.

All packaged files will be found under the folder /dist (.tar.gz for Linux, .zip for Windows).

#### *B. Advanced compilation*

SkyWalking is a complex Maven project that consists of many modules, some of which may take a long time to compile. If you would like to recompile a specific part of the project, there are different ways to achieve this.

- Compile agent package: `./mvnw package -Pagent,dist` or `make build.agent`.
- Compile backend package: `./mvnw package -Pbackend,dist` or `make build.backend`.
- Compile UI package: `./mvnw package -Pui,dist` or `make build.ui`.

#### *C. Build Docker image*

You can build Docker images for the backend service and UI with Makefile under the root folder (prior configuration of the Docker environment is required).

- Build all Docker images: `make docker.all`.
- Build Docker image for the backend service: `make docker.oap`.
- Build Docker image for UI: `make docker.ui`.

The HUB and TAG variables are used to provide the REPOSITORY and TAG of the Docker image. To get an OAP image named bar/oap:foo, run the following command:

```
HUB=bar TAG=foo make docker.oap
```

## 2.1.2 Project structure

SkyWalking is a Maven project made up of multiple sub-modules, each of which has an independent function. The following section will introduce you to the project structure of SkyWalking.

This section provides an analysis of the entire project by each module. Readers may wish to skim through this section first and proceed to the subsequent chapters, and then return later for a deeper understanding of the SkyWalking project structure.

After downloading the SkyWalking source code, you may view the outermost POM file of the SkyWalking project, which defines the top-level sub-modules of SkyWalking (which are further made up of the next level of sub-modules).

```
<modules>
  <module>apm-sniffer</module>
  <module>apm-application-toolkit</module>
  <module>oap-server</module>
  <module>apm-webapp</module>
  <module>apm-dist</module>
</modules>
```

### A. *apm-sniffer*

This module is part of SkyWalking's JavaAgent (to be introduced in Section 2.2). It consists of the following sub-modules:

```
<modules>
  <module>apm-agent</module>
  <module>apm-agent-core</module>
  <module>apm-sdk-plugin</module>
  <module>apm-toolkit-activation</module>
  <module>apm-test-tools</module>
  <module>bootstrap-plugins</module>
  <module>optional-plugins</module>
</modules>
```

The relevant functions of these sub-modules are introduced as follows:

- **apm-agent:** This module serves as the entry point for the JavaAgent.
- **apm-agent-core:** This module is the core processing logic of JavaAgent and has functions including automatic embedding and data collection.
- **apm-sdk-plugin:** This module contains stable third-party plug-ins that are supported by SkyWalking.
- **apm-toolkit-activation:** This module contains its own extension plug-ins that are supported by SkyWalking.

- `apm-test-tools`: This module is a testing tool plug-in.
- `bootstrap-plugins`: This module contains the JDK plug-ins that are supported by SkyWalking.
- `optional-plugins`: This module contains optional third-party plug-ins that are supported by SkyWalking.

### B. *apm-application-toolkit*

This is the extension pack module of SkyWalking (to be introduced in section 2.2.4). It consists of the following sub-modules:

```
<modules>
  <module>apm-toolkit-log4j-1.x</module>
  <module>apm-toolkit-log4j-2.x</module>
  <module>apm-toolkit-logback-1.x</module>
  <module>apm-toolkit-opentracing</module>
  <module>apm-toolkit-trace</module>
</modules>
```

The relevant functions of these sub-modules are introduced as follows:

- `apm-toolkit-log4j-1.x`: This module is an integration of SkyWalking and log4j 1.x.
- `apm-toolkit-log4j-2.x`: This module is an integration of SkyWalking and log4j 2.x.
- `apm-toolkit-logback-1.x`: This module is an integration of SkyWalking and logback 1.x.
- `apm-toolkit-opentracing`: This module is an extension pack for the opentracing protocol interface of SkyWalking.
- `apm-toolkit-trace`: This module is a custom-defined extension pack provided by SkyWalking.

### C. *oap-server*

This module is part of SkyWalking's backend (to be introduced in Section 2.3). It consists of the following sub-modules:

```
<modules>
  <module>server-core</module>
  <module>server-receiver-plugin</module>
  <module>server-cluster-plugin</module>
  <module>server-storage-plugin</module>
  <module>server-library</module>
  <module>server-starter</module>
  <module>server-query-plugin</module>
  <module>server-alarm-plugin</module>
  <module>server-testing</module>
  <module>oal-rt</module>
  <module>server-telemetry</module>
```

```
<module>oal-grammar</module>
<module>exporter</module>
<module>server-configuration</module>

</modules>
```

Due to the high scalability of SkyWalking, the modules are designed based on highly abstract interfaces. The manner in which each module completes specified functions is determined by the concrete implementation. For example, the main function of the module `server-storage-plugin` discussed above is backend storage; users can easily replace the concrete implementation on this layer by extending the SkyWalking interface (to be discussed in detail in Chapter 6).

The relevant functions of these sub-modules are introduced as follows:

- `server-core`: This module is the core processing logic of the Backend.
- `server-receiver-plugin`: This module contains the data collection method that is supported by the Backend, such as obtaining parameters from the JavaAgent's end, and obtaining metrics from Envoy.
- `server-cluster-plugin`: This module defines the cluster method for the Backend, e.g., through Consul and ZooKeeper.
- `server-storage-plugin`: This module defines the storage method for the Backend, e.g., through Elasticsearch and MySQL.
- `server-library`: This module preserves all information for dependency packages of the Backend.
- `server-starter`: This module is the entry point for the Backend.
- `server-query-plugin`: This module defines how the Backend queries data, e.g., through GraphQL.
- `server-alarm-plugin`: This module serves as the alarm for the Backend.
- `server-testing`: This is a Backend testing module.
- `server-telemetry`: This module is responsible for the network telemetry for the Backend, e.g., through Prometheus.
- `exporter`: This module is responsible for exposing the data interface of the Backend (to be introduced in section 2.3.11).
- `server-configuration`: This module is the Backend configuration center, e.g., Apollo and ZooKeeper.
- `oal-rt/oal-grammar`: This module is mainly responsible for the OAL system of the Backend (to be introduced in Chapter 7).

#### D. *apm-webapp*

This module serves as the UI of SkyWalking, which defines the port of the Web module and the interface address on Server's end.

### *E. apm-dist*

This is the packaged module of SkyWalking, which defines the packaging guidelines for the entire module.

## 2.2 Installation of JavaAgent

JavaAgent is the data transmitter in the SkyWalking system. Through simple deployment and configuration, users can obtain their businesses' trace data through a codeless, non-intrusive solution.

This section will provide guidance on how to install JavaAgent, and introduce its configuration parameters, supported plug-ins and certain advanced features. Through this section, readers will have a deeper understanding on how to use JavaAgent and apply advanced features for real-life business scenarios.

### 2.2.1 Installation method

#### *A. Download JavaAgent*

Download the latest release package from [the SkyWalking official website](#). Locate the agent directory. The internal structure of the agent is as follows:

```
+-- agent
  +-- activations
    apm-toolkit-log4j-1.x-activation.jar
    apm-toolkit-log4j-2.x-activation.jar
    apm-toolkit-logback-1.x-activation.jar
    ...
  +-- config
    agent.config
+-- optional-plugins
  apm-trace-ignore-plugin-6.2.0.jar
  ...
+-- plugins
  apm-dubbo-plugin.jar
  apm-feign-default-http-
  9.x.jar apm-httpClient-4.x-
  plugin.jar
  ...
  skywalking-agent.jar
```

#### *B. Introduction to the agent directory structure*

The agent directory structure is as follows.

- `skywalking-agent.jar`: This Jar package is the entry point of the SkyWalking JavaAgent as well as the core logic package.
- `plugins`: This folder stores the middlewares, frameworks, and repositories that are supported by SkyWalking.
- `optional-plugins`: This file stores optional middlewares, frameworks, and repositories that are supported by SkyWalking. There is a key distinction between plugin packages under the `optional-plugins` folder and under the `plugins` folder. While SkyWalking automatically loads plugin packages under the `plugins` folder, the desired plugin packages under the `optional-plugins` folder must be manually moved under the `plugins` folder.
- `config`: The `agent.config` stored in this folder is the default configuration of JavaAgent.
- `activations`: The plug-in packages used to activate the SkyWalking application toolkit are stored in this folder.

#### C. *Set the service name*

Modify `agent.service_name` in `config/agent.config`. This parameter is used to mark the name of the current service in SkyWalking.

#### D. *Configure the server address*

Configure `collector.backend_service` (backend address) in `config/agent.config`. Set `127.0.0.1:11800` as default address, such that it only communicates with the local backend.

#### E. *Increase startup parameters*

Add `-javaagent:${absolute_path}/skywalking-agent.jar` in the JVM startup parameters, and make sure that this parameter is in front of the `-jar` parameter. The `${absolute_path}` represents the absolute path of `skywalking-agent.jar` that is downloaded in Step 1.

#### F. *Start the application*

Start the `xxx.jar` application through Java `-javaagent:${absolute_path}/skywalking-agent.jar -jar xxx.jar`.

#### G. *Confirm that the agent has started successfully*

A log folder will appear under the `/agent` directory, in which you can determine whether the agent has started successfully. However, SkyWalking is not yet ready for use at this stage. Refer to Section 2.3 on how to launch the backend and UI to make SkyWalking available for use.

## 2.2.2 Configuration parameters

Table 2-1 shows a partial list of the configurations supported in the Apache SkyWalking config/agent.config file. Refer to <https://github.com/apache/skywalking> for the latest configuration list.

Table 2-1 Partial list of configurations supported in Apache SkyWalking config/agent.config file

Attribute Name	Description	Default Value
agent.namespace	The namespace is used to isolate cross-process propagation headers; the header will become HeaderName:Namespace once configured	default-namespace
agent.service_name	Set a unique name for each service; multiple service instances of a service will have the same service name	Your_ApplicationName
agent.sample_n_per_3_secs	Represents the trace data collected every 3 seconds, negative value or zero means no sampling	-1
agent.span_limit_per_segment	The maximum number of spans in a single segment; spans going over this threshold will be discarded	300
agent.is_open_debugging_class	If true, SkyWalking will save all instrumented class files to the agent/debugging folder	true
agent.instance_uuid	SkyWalking treats the same instance uuid as a single instance; if empty, the SkyWalking Agent will generate a 32-bit uuid	Not set
collector.grpc_channel_check_interval	Check time interval (in seconds) of the channel status of gRPC	30
collector.app_and_service_register_check_interval	Check time interval (in seconds) of the registration status of applications and services	30
collector.backend_service	Backend service address of SkyWalking	127.0.0.1:11800
logging.level	Logging level for print process	DEBUG



logging.file_name	Log file name	skywalking-api.log
logging.dir	Log directory. An empty string means that the log will be printed under the agent/log folder	Empty string
logging.max_file_size	The maximum size of the log file. When the log file size exceeds this size, archive the current log file and write into a new file	300 × 1024 × 1024 ( 300MB )
jvm.buffer_size	The size of the buffer that collects JVM information	60 × 10
buffer.buffer_size	The size of the buffer as reported by trace data	60 × 10
buffer.channel_size	The size of the channel as reported by trace data	5
dictionary.service_code_buffer_size	Used to set the buffer size of service code	10 × 10 000
dictionary.endpoint_name_buffer_size	Used to set the buffer size of endpoint name	1000 × 10 000

### 2.2.3 Introduction

There are two types of plug-ins in SkyWalking's plug-in repository:

- **Stable plug-in:** These are plug-ins that are automatically loaded upon installation of SkyWalking; they are located under the agent/plugins folder.
- **Optional plug-in:** These plug-ins will not be loaded automatically; they are located under the agent/optional-plugins folder; If you would like to use optional plug-ins, you need to copy the Jar package of the relevant plug-in to the /agent/plugins folder.

#### A. *Stable plug-in*

The list below shows all stable plugins supported by SkyWalking as of version 6.2.0. The list will be updated as new versions of the project continue to be released. Please note that the following is for reference only and the official documentation shall prevail.

- **HTTP Server**
  - Tomcat 7
  - Tomcat 8

- Tomcat 9
- Spring Boot Web 4.x
- Spring MVC 3.x, 4.x 5.x with servlet 3.x
- Nutz Web Framework 1.x
- Struts2 MVC 2.3.x → 2.5.x
- Jetty Server 9
- Spring Webflux 5.x
- Undertow 2.0.0.Final → 2.0.13.Final
- RESTEasy 3.1.0.Final → 3.7.0.Final
- HTTP Client
  - Feign 9.x
  - Netflix Spring Cloud Feign 1.1.x, 1.2.x, 1.3.x
  - OkHttp 3.x
  - Apache httpcomponent HttpClient 4.2, 4.3
  - Spring RestTemplate 4.x
  - Jetty Client 9
  - Apache httpcomponent AsyncClient 4.x
- JDBC
  - Mysql Driver 5.x, 6.x, 8.x
  - H2 Driver 1.3.x → 1.4.x
  - Sharding-JDBC 1.5.x
  - ShardingSphere 3.0.0
  - ShardingSphere 3.0.0, 4.0.0-RC1
  - PostgreSQL Driver 8.x, 9.x, 42.x
- RPC framework
  - Dubbo 2.5.4 → 2.6.0
  - Dubbox 2.8.4
  - Apache Dubbo 2.7.0
  - Motan 0.2.x → 1.1.0
  - gRPC 1.x
  - Apache ServiceComb Java Chassis 0.1 → 0.5, 1.0.x
  - SOFARPC 5.4.0
- MQ
  - RocketMQ 4.x
  - Kafka 0.11.0.0 → 1.0
  - ActiveMQ 5.x
  - RabbitMQ 5.x
- NoSQL

- Redis
  - Jedis 2.x
  - Redisson Easy Java Redis client 3.5.2+
- MongoDB Java Driver 2.13-2.14,3.3+
- Memcached Client
  - Spymemcached 2.x
  - Xmemcached 2.x
- Elasticsearch
  - transport-client 5.2.x-5.6.x
  - SolrJ 7.0.0-7.7.1
- Service discovery
  - Netflix Eureka
- Spring ecosystem
  - Spring Core Async
  - SuccessCallback/FailureCallback/ListenableFutureCallback 4.x
- Hystrix: Latency and Fault Tolerance for Distributed Systems 1.4.20 → 1.5.12
- Scheduler
  - Elastic Job 2.x
- OpenTracing community support
- Canal: Alibaba's incremental subscription and consumer component based on MySQL binlog 1.0.25 → 1.1.2
- Vert.x ecosystem
  - Vert.x Eventbus 3.2+
  - Vert.x Web 3.x

### *B. Optional plug-ins*

Optional plug-ins are shown as follows.

- HTTP Server
  - Resin 3 (¹)
  - Resin 4 (¹)
- HTTP Gateway
  - Spring Cloud Gateway 2.1.x.RELEASE (²)
- JDBC
  - Oracle Driver (¹)
- NoSQL
  - Redis

- Lettuce 5.x <sup>(2)</sup>
- Distributed coordination
  - ZooKeeper 3.4.x <sup>(2)</sup> , except 3.4.4 )
- Spring ecosystem
  - Spring Bean annotations (@Bean, @Service, @Component, @Repository) 3.x and 4.x <sup>(2)</sup>
- JSON
  - GSON 2.8.x <sup>(2)</sup>
- SkyWalking optional plug-in
  - Trace-ignore-plugin. This plugin serves to exclude unwanted endpoints.
  - customize-enhance-plugin. This plug-in serves not to replace another plug-in. Rather, it is designed for user convenience. The behavior of this plug-in is very similar to @Trace toolkit, but it does not require modification of the code. It also has more functional powers, such as providing tag and log.

<sup>1</sup>Some plug-ins are released in third-party repositories due to license restrictions or incompatibility. These plug-ins can be obtained from the SkyAPM Java plug-in extension repository.

<sup>2</sup>These plug-ins may affect performance, or may only be used under certain conditions.

Three noteworthy optional plug-ins will be introduced below.

#### a) trace-ignore-plugin

##### *i) Introduction*

- The purpose of this plug-in is to exclude the unwanted endpoints.
- Multiple URL path patterns may be set, and any endpoint being matched will not be traced.
- Current matching rules follow the ant path matching pattern, such as /path/\*, /path/\*\* and /path/?.
- Copy apm-trace-ignore-plugin-x.jar to agent/plugins, restart the agent, and the plug-in will take effect.

##### *ii) How to configure*

The endpoint mode to be ignored may be configured in the following two ways. Configuration of the system environment variable is given higher priority.

- Configure by setting the system environment variable. Add `skywalking.trace.ignore_path` to the system environment variable. The value is the path to be ignored. Multiple paths are separated by the comma ",".
- Copy `/agent/optional-plugins/apm-trace-ignore-plugin/apm-trace-ignore-plugin.config` to the `/agent/config/` folder, then add the following exclusion rule:

- `trace.ignore_path=/your/path/1/**,/your/path/2/**`

## b) Spring Bean annotations

This plug-in can track all methods of beans which are annotated with `@Bean`, `@Service`, `@Component` and `@Repository`.

Why is this plug-in optional? The reason is that tracing beans creates a large number of spans whichever method is used, and this in turn uses up more CPU, memory, and network bandwidth. If you would like to trace as many methods as possible, make sure that the system workload has the capacity to provide such support.

## c) customize-enhance-plugin

### i) Introduction

SkyWalking provides a JavaAgent plug-in development guide to help developers build new plug-ins.

Rather than aiming to replace another plug-in, this plug-in is designed to improve convenience for users. This plug-in behaves similarly to `@Trace` toolkit, but no code modification is required. It also has additional functions such as providing tag and log.

### ii) How to configure

You may implement custom enhancement to the class through these 3 steps:

- 1) Activate the plug-in and move the plug-in from `optional-plugins/apm-customize-enhance-plugin.jar` to `plugin/apm-customize-enhance-plugin.jar`.
- 2) Configure `plugin.customize.enhance_file` in `agent.config` to specify the enhancement rule file, such as `/absolute/path/to/customize_enhance.xml`.
- 3) Configure the enhancement rule in `customize_enhance.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<enhanced>
  <class class_name="test.apache.skywalking.testcase.customize
    .service.TestService1">
    <method method="staticMethod()" operation_name="/is_static_method"
      static="true"/>
    <method method="staticMethod(java.lang.String,int.class,
      java.util.Map,java.util.List,[Ljava.lang.
        Object;)" operation_name="/is_static_method_args" static="true">
```

```

        <operation_name_suffix>arg[0]</operation_name_suffix>
        <operation_name_suffix>arg[1]</operation_name_suffix>
        <operation_name_suffix>arg[3].[0]</operation_name_suffix>
        <tag key="tag_1">arg[2].['k1']</tag>
        <tag key="tag_2">arg[4].[1]</tag>
        <log key="log_1">arg[4].[2]</log>
    </method>
    <method method="method()" static="false"/>
    <method method="method(java.lang.String,int.class)"
        operation_name="/method_2" static="false">
        <operation_name_suffix>arg[0]</operation_name_suffix>
        <tag key="tag_1">arg[0]</tag>
        <log key="log_1">arg[1]</log>
    </method>
    <method method="method(test.apache.skywalking.testcase.customize
        .model.Model0,java.lang.String,int.class)"
        operation_name="/method_3" static="false">
        <operation_name_suffix>arg[0].id</operation_name_suffix>
        <operation_name_suffix>arg[0].model1.name</operation_name_suffix>
        <operation_name_suffix>arg[0].model1.getId()</operation_name_
            suffix>
        <tag key="tag_os">arg[0].os.[1]</tag>
        <log key="log_map">arg[0].getM().['k1']</log>
    </method>
</class>
<class class_name="test.apache.skywalking.testcase.customize.service.TestService2">
    <method method="staticMethod(java.lang.String,int.class)" operation_
        name="/is_2_static_method" static="true">
        <tag key="tag_2_1">arg[0]</tag>
        <log key="log_1_1">arg[1]</log>
    </method>
    < method method="method([Ljava.lang.Object;)"
        operation_name="/method_4" static="false">
        <tag key="tag_4_1">arg[0].[0]</tag>
    </method>
    < method method="method(java.util.List,int.class)"
        operation_name="/method_5" static="false">
        <tag key="tag_5_1">arg[0].[0]</tag>
        <log key="log_5_1">arg[1]</log>
    </method>
</class>
<enhanced>

```

Table 2-2 shows the configuration descriptions of the above file.

Table 2-2 customize-enhance-plugin configuration

Configuration	Description
<code>class_name</code>	Class to be enhanced
<code>method</code>	Class interceptor method
<code>operation_name</code>	If configured, it will replace the default <code>operation_name</code> .
<code>operation_name_suffix</code>	Represents the dynamic data to be added after <code>operation_name</code>
<code>static</code>	Whether the method is static
<code>tag</code>	A tag will be added to the local span; the value of key to be indicated on the XML node
<code>log</code>	A log will be added to the local span; the value of key to be indicated on the XML node
<code>arg[x]</code>	Represents the parameter value input; for example, <code>args[0]</code> represents the first parameter
<code>. [x]</code>	When the object being parsed is an Array or List, this expression can be used to get the object on the relevant index.
<code>. ['key']</code>	When an object being parsed is Map, this expression can be used to get the map key.

## 2.2.4 Advanced Features

SkyWalking provides advanced features applicable to the real business environment. These features may be applied according to the user's actual needs.

### A. *Overriding configuration*

By default, SkyWalking provides the `agent.config` configuration file for the agent.

In a real business environment, there may be many instances of different services deployed on a real physical machine. The configurations of these instances will be fundamentally similar with just one or two exceptions. For example, if the instances differ only in `service_name`, this may be resolved by overriding

the configurations. Although the same agent.config file is loaded for all instances, the key information for each instance remains independent.

Overriding configuration means that users may override information in the configuration files in different ways. Currently, three configuration override methods are supported. As all three methods operate on equal terms, users may decide which method to use simply based on their personal preferences.

#### *i) System properties*

Adopt the configuration name in the skywalking+ configuration file as the configuration name of the system properties to override the value in the configuration file.

Example:

-Dskywalking.agent.service\_name=skywalking-demo means using skywalking-demo to override the agent.service\_name parameter in the configuration.

#### *ii) Agent parameters*

Add parameter configuration after the agent path of the JVM parameter:

```
-javaagent:/path/to/skywalking-agent.jar=[option1]=[value],[option2]=[value2]
```

Example: -javaagent:/path/to/skywalking-

agent.jar=agent.service\_name=skywalking-demo,logging.level=debug, which means that skywalking-demo is used to override the agent.service\_name parameter in the configuration, and debug is used to override the logging.level parameter in the configuration.

Note that if a separator (", " or "=") is used in the option or option value, single quotation marks must be added. See example below: -javaagent:/path/to/skywalking-agent.jar=agent.ignore\_suffix='.jpg,.jpeg'

#### *iii) System environment variables*

Override agent.application\_code and logging.level with the following configurations.

```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:Your_ApplicationName}

# Logging level
logging.level=${SW_LOGGING_LEVEL:INFO}
```

If the SW\_AGENT\_NAME environment variable already exists in your operating system and its value is skywalking-agent-demo, then the value of agent.service\_name will be overwritten as skywalking-agent-demo; otherwise, it will be set as Your\_ApplicationName.

Placeholder nesting is also supported. For example: \${SW\_AGENT\_NAME:\${ANOTHER\_AGENT\_NAME:Your\_ApplicationName}}. In this case, if the SW\_AGENT\_NAME environment variable



does not exist, but the `ANOTHER_AGENT_NAME` environment variable does exist and its value is `skywalking-agent-demo`, then the value of `agent.service_name` value will be overwritten as `skywalking-Agent-Demo`; otherwise, it will be set as `Your_ApplicationName`.

### *iii) Priority*

The priority in descending order is as follows:

Agent Parameters > System Properties > System Environment Variables > Configuration Files

## *B. Custom configuration files*

Overriding configuration is mainly applicable to the exceptional situation where the configurations of several instances are fundamentally the same and may be configured through overriding configurations. However, if the several instances have essentially different configurations and the configurations are being overridden, it will lead to an increase in the costs of operation and maintenance. In such a case, it is more sensible to use custom configuration files. With a custom configuration file, a different configuration file may be loaded into each instance such that all configuration files may be maintained uniformly.

If the user sets up the configuration file for the agent through this feature, the agent will load the configuration file that has been set up. This feature does not conflict with the configuration override feature introduced in Section 2.2.4.

### *i) How to use*

The content format of the specified configuration file must be the same as the default configuration file. Use `System.Properties(-D)` to set the specified configuration file path:

```
-Dskywalking_config=/path/to/agent.config
```

`/path/to/agent.config` is the absolute path of the specified configuration file.

### *ii) Priority*

Specified Agent Configuration File > Default Agent Configuration File

## *C. Client sampling*

Set the agent parameter `agent.sample_n_per_3_secs` for sampling on the client's end. The parameter indicates how much trace data is collected every 3 seconds. A negative number or zero means no

sampling is being conducted. As an illustration, `agent.sample_n_per_3_secs=400` means that only 400 trace data links are collected every 3 seconds.

#### D. TLS

Transport Layer Security (TLS) is a very common security solution for Internet data transmission. TLS is a protocol built on top of Transfer Control Protocol (TCP) in the transport layer. It encrypts the application-layer messages before transferring them to TCP for transmission to ensure security of the transmitted data.

For some SkyWalking use cases, the target application will be in one location (also known as VPC) while the SkyWalking backend will be in another location (VPC). There is a risk of interception of the communications between the two ends, leading to leakage of the transmitted content. Therefore, SkyWalking uses gRPC TLS to ensure communication security. It only supports non-mutual auth certificates.

Follow these steps to enable TLS.

- 1) Generate `ca.crt`, `server.crt` and `server.pem` through the SkyWalking script<sup>1</sup>.
- 2) Enable and configure TLS.
  - Agent's end: Move `ca.crt` to the `/ca` folder of agent package. As the `/ca` folder is not included in the distribution package, please create the folder yourself. Once the agent detects `/ca/ca.crt`, it will automatically enable TLS.
  - Server: Configure the TLS of `application.yml/core/default` as follows.

```
grpcSslEnabled: true grpcSslKeyPath: /path/to/server.pem
grpcSslCertChainPath: /path/to/server.crt grpcSslTrustedCAPath: /path/to/ca.crt
```

`path/to/` represents the absolute path of the file.

#### E. Namespace

##### *i) Background*

SkyWalking is a monitoring tool that collects metrics from distributed systems. In practice, large distributed systems contain hundreds of services and service instances. Therefore, it is likely that multiple teams and or even companies will be working together to maintain and monitor the

---

<sup>1</sup> To download the SkyWalking script, visit [https://github.com/apache/skywalking/blob/master/tools/TLS/tls\\_key\\_generate.sh](https://github.com/apache/skywalking/blob/master/tools/TLS/tls_key_generate.sh).

distributed system. Since each team or company is in charge of a different part, it is neither desirable or appropriate for them to share the metrics.

The namespace is used, in this context, to track the isolation status of the monitoring system.

#### *ii) How to set up the namespace*

Set up `agent.namespace` in the agent configuration file :

```
# The agent namespace
# agent.namespace=default-namespace
```

The key of SkyWalking's default header is SW6. After setting up `agent.namespace`, the header key will be changed to `namespace+ "-sw6"`. For example, if the namespace is `test`, the header key is `test-sw6`.

### *F. Application Toolkit API and OpenTracing API*

SkyWalking provides users with an API tool called the Application Toolkit API, which integrates logs and `TraceId`, customizes trace methods, modifies span information, and conducts cross-thread tracing. OpenTracing API is implemented by SkyWalking based on the OpenTracing standard. Users can use both of these APIs to construct the context of distributed tracing.

Here's how to use these two APIs.

#### *i) How to print trace context in log*

As of version 6.2.0, the log frameworks supported by SkyWalking are `log4j`, `log4j2`, and `logback`. Follow these steps to print the trace contexts.

##### *a) log4j*

- 1) Introduce toolkit dependency with Maven or Gradle.

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-log4j-1.x</artifactId>
  <version>{project.release.version}</version>
</dependency>
```

- 2) Configure the layout.

```
log4j.appender.CONSOLE.layout=TraceIdPatternLayout
```

- 3) Set up `%T` in `layout.ConversionPattern`.

```
log4j.appender.CONSOLE.layout.ConversionPattern=%d [%T] %-5p %c{1}:%L - %m%n
```

- 4) Activate SkyWalking tracer with `-javaagent`. Then, log4j will output traceId (if any). If the tracer is not activated, the output will be TID: N/A.

*b) log4j2*

- 1) Introduce toolkit dependency with Maven or Gradle.

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-log4j-2.x</artifactId>
  <version>{project.release.version}</version>
</dependency>
```

- 2) Configure `[traceid%]` in the `log4j2.xml` pattern.

```
<Appenders>
  <Console name="Console" target="SYSTEM_OUT">
    <PatternLayout pattern="%d [%traceId] %-5p %c{1}:%L - %m%n"/>
  </Console>
</Appenders>
```

- 3) Activate SkyWalking tracer with `-javaagent`. Then, log4j2 will output traceId (if any). If the tracer is not activated, the output will be TID: N/A.

*c) logback*

- 1) Introduce toolkit dependency with Maven or Gradle.

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-logback-1.x</artifactId>
  <version>{project.release.version}</version>
</dependency>
```

- 2) Configure `%tid` in the `logback.xml` Pattern.

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
    <layout class="org.apache.skywalking.apm.toolkit.log.logback.v1.x
      .TraceIdPatternLogbackLayout">
      <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%tid] [%thread] %-5level
        %logger{36} -%msg%n</Pattern>
    </layout>
  </encoder>
```

```
</appender>
```

- 3) Activate SkyWalking tracer with `-javaagent`. Then, logback will output `traceId` (if any). If the tracer is not activated, the output will be TID: N/A.

#### *ii) How to read / add Trace context through annotations or SkyWalking's native API*

- 1) Introduce toolkit dependency with Maven or Gradle.

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-trace</artifactId>
  <version>${skywalking.version}</version>
</dependency>
```

- 2) Obtain `traceId` through `TraceContext.traceId()` API.

```
import TraceContext;
...
modelAndView.addObject("traceId", TraceContext.traceId());
```

- 3) Customize Trace with the following API.

- Add `@Trace` annotation to the method to be traced. After adding, you can view the span information in the method-call stack.
- Add custom tags in the context lifecycle of the method being traced.
  - `ActiveSpan.error()`: Mark the current Span with error status.
  - `ActiveSpan.error(String errorMsg)`: Mark the current Span with error status and attach error message.
  - `ActiveSpan.error(Throwable throwable)`: Mark the current Span with error status and attach Throwable.
  - `ActiveSpan.debug(String debugMsg)`: Add a debug-level log message to the current Span.
  - `ActiveSpan.info(String infoMsg)`: Add an info-level log message to the current Span.

The sample codes are as follows:

```
ActiveSpan.tag("my_tag", "my_value");
ActiveSpan.error();
ActiveSpan.error("Test-Error-Reason");
ActiveSpan.error(new RuntimeException("Test-Error-Throwable"));
ActiveSpan.info("Test-Info-Msg");
ActiveSpan.debug("Test-debug-Msg");
```

#### *iii) How to manually implement trace cross-thread transfer*

1) Introduce toolkit dependency with Maven or Gradle.

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-trace</artifactId>
  <version>${skywalking.version}</version>
</dependency>
```

2) Complete cross-thread transfer with CallableWrapper or RunnableWrapper.

```
ExecutorService executorService = Executors.newFixedThreadPool(1);
    executorService.submit(CallableWrapper.of(new Callable<String>() {
        @Override public String call()
            throws Exception { return null;
        }
    }));
    ExecutorService executorService = Executors.newFixedThreadPool(1);
    executorService.execute(RunnableWrapper.of(new Runnable() {
        @Override public void run() {
            //your code
        }
    }));
```

3) Complete cross-thread transfer with @TraceCrossThread.

```
@TraceCrossThread
public static class MyCallable<String> implements Callable<String>
{
    @Override
    public String call() throws Exception {return null;
    }
}
...
ExecutorService executorService = Executors.newFixedThreadPool(1);
executorService.submit(new MyCallable());
```

*iv) How to use OpenTracing Java API*

1) Introduce toolkit dependency with Maven or Gradle.

```
<dependency>
  <groupId>org.apache.skywalking</groupId>
  <artifactId>apm-toolkit-opentracing</artifactId>
  <version>{project.release.version}</version>
</dependency>
```

2) Implement with OpenTracing tracer.

```
Tracer tracer = new SkywalkingTracer();
Tracer.SpanBuilder spanBuilder = tracer.buildSpan(
    "/yourApplication/yourService");
```

## 2.3 Back-end and UI deployment

This section will introduce you to the deployment method of the back-end collector (OAP Server) and front-end UI, and the main configurations in SkyWalking. You can deploy the back-end of SkyWalking in a physical machine, a virtual machine or a Kubernetes cluster. After reading this section, you will understand how the SkyWalking back-end collector and front-end UI operate. This will help you optimize your parameters based on actual business requirements.

### 2.3.1 Introduction to deployment in SkyWalking

You can download the latest version of the SkyWalking release package from the SkyWalking official website (<http://skywalking.apache.org/downloads/>). The release package contains the files needed to deploy the SkyWalking back-end and UI. After decompressing the package, you will see the following directories:

```
→ $ tree -d -L 1
.
├─ agent
├─ bin
├─ config
├─ licenses
├─ logs
├─ mesh-buffer
├─ oap-libs
├─ trace-buffer
└─ webapp
```

This is an overview of the directories:

- **agent:** The Jar package directory required by JavaAgent. It contains the Jar package and configuration required by the JavaAgent end. Refer to the previous sections for more information.
- **bin:** The startup script directory, which contains the scripts required to start the UI module and back-end services on Linux and Windows.
- **config:** The project configuration directory, which includes the main configuration file `application.yml` for back-end services, log configuration file `log4j2.xml`, alarm configuration file and datasource configuration file. You can modify these configurations based on your actual needs.
- **licenses:** The license file statement of the third-party package used by the project.
- **logs:** Log path. By default, the back-end module and UI module will output logs to this index after being started.
- **mesh-buffer:** The cache directory for receiving service-mesh data.
- **oap-libs:** This directory contains the Jar package required to operate the back-end module.
- **Trace-buffer:** The cache directory that receives the trace segment sent by the agent.

- webapp: The UI file deployment directory, which contains the UI deployment Jar package and the relevant configuration file `webapp.yml`.

Based on our understanding of the SkyWalking file structure, let's study the SkyWalking operating modules (see Figure 2-1). Generally speaking, there are four types of modules:

- Agent/Probe: Used to collect monitoring data from services, including information on call relationships, execution time, error reports and error stack. Service mesh agents belong to this type.
- Back-end services (Backend/OAP Server): Used to collect and process monitoring data reported by the agent, and to persist data to storage modules after data analysis and processing.
- Storage: Used to persist data processed by back-end services. SkyWalking provides various production-level storage implementation methods, such as Elasticsearch, MySQL and TiDB.
- UI module: Used for front-end query of various monitoring data and display to users.

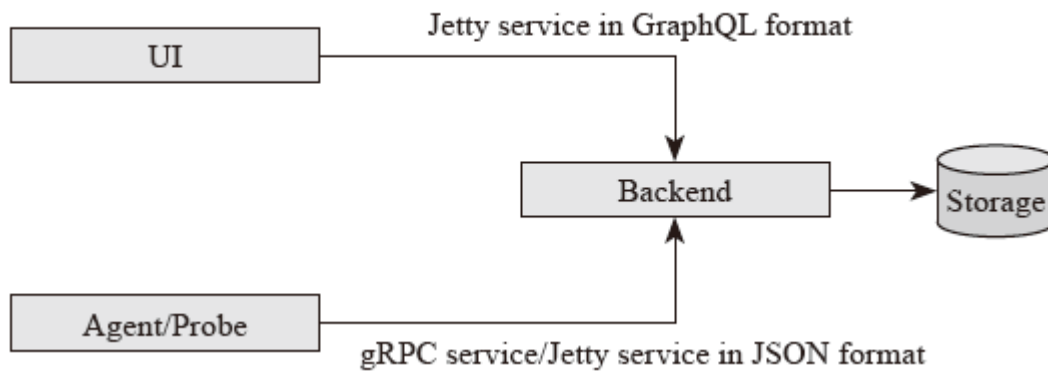


Figure 2-1 Operating modules

### 2.3.2 Quick start

Before deploying your own set of SkyWalking services, make sure the Java environment has been installed correctly. JDK 8 or above is required.

This section will provide you with a preliminary understanding of SkyWalking and a preview of its functions. This introduction is far from enough to explain the deployment techniques required in a long-term production environment. Refer to Section 3.3 for detailed guidance on the deployment process in a production environment.

The simplest way to start will be outlined here. First, enter the SkyWalking folder. Then, directly execute `bin/startup.sh` (for Linux or Mac users) or `bin/startup.bat` (for Windows users) on the command line.



The startup will start the back-end oapServer (that corresponds to startup script `oapService.sh`) and the UI module (that corresponds to startup script `webappService.sh`) at the same time. If the start is successful, a log that indicates a successful start will display in the console.

```
→ $ bin/startup.sh
SkyWalking OAP started successfully!
SkyWalking Web Application started successfully!
```

During quick start, the default configuration file will be read. The default oapServer configuration file is found in the `config` folder. The default UI module configuration file is found in the `webapp` folder. Under normal circumstances, after the console shows that the startup has been successful, the oapServer will become fully functional. The storage and service ports used in the quick start process are described below:

- Default storage of the back-end oapServer is the H2 database, so there is no need to deploy additional storage. You may refer to the relevant storage configurations in `config/application.yml` for more information.
- By default, the back-end oapServer uses the gRPC monitoring port 11800 and HTTP RESTful port 12800 to receive multiple languages (including Java, PHP, .Net Core, Node.js and Go) of agents and monitoring data from the data plane of service mesh.
- The default monitoring port is 8080 for the UI module, used to access the front page. The UI module proxies all back-end requests from the front-end (GraphQL) to port 12800 of the back-end oapServer.

### 2.3.3 Explaining application.yml

Operation of the SkyWalking back-end depends on the configuration of `application.yml`. Getting to know this configuration file is crucial to understanding what modules there are on the backend and how they run.

`application.yml` is configured in the YAML file format. The SkyWalking back-end is designed with a modular approach. Users can configure `application.yml` according to their actual needs and deploy a combination of plug-ins for each functional module. There are three aspects in the configuration of `application.yml`:

- Module name
- Module implementation (Provider)
- Specific configuration

For example, in the following configuration, `core` represents a module name, `default` represents a

default specific implementation of a core module (Provider), and `restHost`, `restPort`, etc. represent the specific configuration implemented by default. Similarly, this applies to all other modules.

```
core:
  default:
    restHost: 0.0.0.0
    restPort: 12800
    restContextPath:
    / gRPCHost:
    0.0.0.0
    gRPCPort: 11800
```

There are two types of back-end modules in SkyWalking: required modules and optional modules. The required modules constitute the basic conditions for back-end operations. The following is an overview of the main types of required modules:

- Core module: A basic infrastructure responsible for data analysis, and the scheduling and distribution of data streams.
- Cluster module: Manages multi-instance deployment and provides high-throughput data processing capabilities.
- Storage module: Responsible for persistent storage of Trace link data and metrics analysis data.
- Query module: Provides query interface for UI module.
  - In addition to the required modules, SkyWalking also provides optional modules with wide-ranging functions. Here are the main types of optional modules:
- Alarm module: Provides alarm capability for the analyzed data based on rules specified by the user.
- Receiver module: Receives monitoring data from multiple sources, including data from third parties.
- Telemetry module: Provides external monitoring systems with monitoring data from SkyWalking's back-end operations, and default implementation of the collection interface Prometheus and data collection process self-observability (so11y).
- Dynamic configuration module: Primarily based on `application.yml` and partly supports integration with dynamic configurations, such as supporting Apollo, ZooKeeper, Consul and etcd.
- Metric exporter module: Allows exporting of metrics data that are analyzed and processed by the SkyWalking back-end to other user-defined services, facilitating secondary development and display.

The specific functions and configurations of each required module are introduced below.

### A. Core module configuration

The core module is used to set up core configurations of the SkyWalking back-end, such as the gRPC port used for data collection, and data expiration time.

```
core:
  default:
    role: ${SW_CORE_ROLE:Mixed} # Mixed/Receiver/Aggregator restHost:
    ${SW_CORE_REST_HOST:0.0.0.0}
    restPort: ${SW_CORE_REST_PORT:12800} restContextPath:
    ${SW_CORE_REST_CONTEXT_PATH:/} gRPCHost:
    ${SW_CORE_GRPC_HOST:0.0.0.0}
    gRPCPort: ${SW_CORE_GRPC_PORT:11800}
    downsampling:
      - Hour
      - Day
      - Month
    enableDataKeeperExecutor: $ {SW_CORE_ENABLE_DATA_KEEPER_EXECUTOR: true}
    recordDataTTL: $ {SW_CORE_RECORD_DATA_TTL: 90} # Unit is minute minuteMetricsDataTTL:
    $ {SW_CORE_MINUTE_METRIC_DATA_TTL: 90} # Unit is minute hourMetricsDataTTL: $
    {SW_CORE_HOUR_METRIC_DATA_TTL: 36} # Unit is hour
    dayMetricsDataTTL: ${SW_CORE_DAY_METRIC_DATA_TTL:45} # Unit is day
    monthMetricsDataTTL: ${SW_CORE_MONTH_METRIC_DATA_TTL:18} # Unit is month
    enableDatabaseSession: ${SW_CORE_ENABLE_DATABASE_SESSION:true}
```

Configuration of the core module is relatively simple, and can be summarized into:

- The role of the current instance node
- Address and port configuration
- Metrics data downsampling
- Data validity period

Under the cluster deployment mode, communications occur between the back-end services. You can assign different roles to each node in the complex cluster deployment and network environment by specifying their roles. Currently, the following three roles are available:

- Mixed mode: The default role. In this mode, the back-end services have the following main functions:
  - Receiving data from Agent
  - Aggregation of L1 data
  - Communication between clusters (sending/receiving)
  - Aggregation of L2 data
  - Data persistent storage

- Alarming
- Receiver mode (data collection only): Receives data from the Agent's end, carries out basic processing of L1 data, and sends the data to other nodes for further processing. In this mode, the back-end services have the following main functions:
  - Receiving data from Agent
  - Aggregation of L1 data
  - Sending its data to other nodes in the cluster for processing
- Aggregator mode (data aggregation only): Accepts data processing requests from other nodes in the cluster. No data from the client is collected. In this mode, the back-end services have the following main functions:
  - Receiving data from other nodes in the cluster
  - Aggregation of L2 data
  - Data persistent storage
  - Alarming

### B. Cluster module configuration

First of all, decide whether to use the stand-alone deployment mode or the cluster deployment mode for the cluster module configuration in SkyWalking back-end services. In a production environment, the cluster deployment mode is usually recommended. If the cluster deployment mode is used, you will need to configure a global registry to enable change detection for cluster nodes. Currently, the services being supported are ZooKeeper, Consul, Nacos, and etcd. SkyWalking may also be deployed in the Kubernetes cluster and its OAP may be run through a cloud-native approach.

```
cluster:
  standalone:
#     Please check your ZooKeeper is 3.5+, However, it is also compatible with
#     ZooKeeper 3.4.x. Replace the ZooKeeper 3.5+ library the oap-libs folder with your
#     ZooKeeper 3.4.x library.
# zookeeper:
#     nameSpace: ${SW_NAMESPACE:""}
#     hostPort: ${SW_CLUSTER_ZK_HOST_PORT:localhost:2181}
#     #Retry Policy
#     baseSleepTimeMs: ${SW_CLUSTER_ZK_SLEEP_TIME:1000} # initial amount of
#     time to wait between retries
#     maxRetries: ${SW_CLUSTER_ZK_MAX_RETRIES:3} # max number of times to retry
#     # Enable ACL
#     enableACL: ${SW_ZK_ENABLE_ACL:false} # disable ACL in default
#     schema: ${SW_ZK_SCHEMA:digest} # only support digest schema
#     expression: ${SW_ZK_EXPRESSION:skywalking:skywalking}
# kubernetes:
```

```

#     watchTimeoutSeconds: ${SW_CLUSTER_K8S_WATCH_TIMEOUT:60}
#     namespace: ${SW_CLUSTER_K8S_NAMESPACE:default}
#     labelSelector: ${SW_CLUSTER_K8S_LABEL:app=collector,release=skywalking}
#     uidEnvName: ${SW_CLUSTER_K8S_UID:SKYWALKING_COLLECTOR_UID}
# consul:
#     serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
#     Consul cluster nodes, example: 10.0.0.1:8500,10.0.0.2:8500,10.0.0.3:8500
#     hostPort: ${SW_CLUSTER_CONSUL_HOST_PORT:localhost:8500}
# nacos:
#     serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
#     hostPort: ${SW_CLUSTER_NACOS_HOST_PORT:localhost:8848}
# etcd:
#     serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
#     etcd cluster nodes, example: 10.0.0.1:2379,10.0.0.2:2379,10.0.0.3:2379
#     hostPort: ${SW_CLUSTER_ETCD_HOST_PORT:localhost:2379}

```

By default, SkyWalking uses a stand-alone deployment mode, but this is not recommended in a production environment. You can choose any of the above registries for node registration to ensure high availability for the OAP services.

### C. Storage module configuration

SkyWalking provides a variety of storage solutions. Based on the characteristics of the technology stack and your familiarity with the solutions, you can choose to store the data in the Elasticsearch cluster, MySQL, or the TiDB cluster solution. Take Elasticsearch as an example:

```

storage:
  elasticsearch:
    nameSpace: ${SW_NAMESPACE:""}
    clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:localhost:9200}
    protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
    #trustStorePath: ${SW_SW_STORAGE_ES_SSL_JKS_PATH:"../es_keystore.jks"}
    #trustStorePass: ${SW_SW_STORAGE_ES_SSL_JKS_PASS:""}
    user: ${SW_ES_USER:""}
    password:${SW_ES_PASSWORD:""}
    indexShardsNumber: ${SW_STORAGE_ES_INDEX_SHARDS_NUMBER:2} indexReplicasNumber:
    ${SW_STORAGE_ES_INDEX_REPLICAS_NUMBER:0}
    # Those data TTL settings will override the same settings in core module.
    recordDataTTL: ${SW_STORAGE_ES_RECORD_DATA_TTL:7} # Unit is day
    otherMetricsDataTTL: ${SW_STORAGE_ES_OTHER_METRIC_DATA_TTL:45} # Unit is day
    monthMetricsDataTTL: ${SW_STORAGE_ESDATA_MONTH_TTL:18} # Unit is month
    bulkActions: ${SW_STORAGE_ES_BULK_ACTIONS:1000} # Execute the bulk every
    1000 requests
    flushInterval: ${SW_STORAGE_ES_FLUSH_INTERVAL:10} # flush the bulk every
    10 seconds whatever the number of requests
    concurrentRequests: ${SW_STORAGE_ES_CONCURRENT_REQUESTS:2} # the number of
    concurrent requests
    metadataQueryMaxSize: ${SW_STORAGE_ES_QUERY_MAX_SIZE:5000} segmentQueryMaxSize:
    ${SW_STORAGE_ES_QUERY_SEGMENT_SIZE:200}
  # h2:
  #     driver: ${SW_STORAGE_H2_DRIVER:org.h2.jdbcx.JdbcDataSource}
  #     url: ${SW_STORAGE_H2_URL:jdbc:h2:mem:skywalking-oap-db}

```

```
# user: ${SW_STORAGE_H2_USER:sa}
# metadataQueryMaxSize: ${SW_STORAGE_H2_QUERY_MAX_SIZE:5000} #
mysql:
# metadataQueryMaxSize: ${SW_STORAGE_H2_QUERY_MAX_SIZE:5000}
```

By default, SkyWalking uses the Elasticsearch cluster as its persistent storage database. If you choose to use MySQL or TiDB as the data storage tool, configure the database connection information in the `config/datasource-settings.properties` file and set the storage tool as MySQL.

#### D. Query module configuration

The query module processes API requests through the GraphQL framework (<https://graphql.org/>). Currently, configuration is required only for the `contextPath` of the API request. The HOST and port that provide the services are the `restHost` and `restPort` of the core module above.

```
query:
  graphql:
    path: ${SW_QUERY_GRAPHQL_PATH:/graphql}
```

### 2.3.4 Parameter overwriting

Although `application.yml` provides extensive and flexible configurations, users might still need to have different configurations for specific nodes in some cases. In this case, startup parameters or environment variables can be used for the purpose of overwriting. The descending order of priority is as follows:

Startup Parameter Specification > System Environment Variables > Specific Configuration Files in `application.yml`.

- 1) Configure parameters with startup parameters. The parameters configured with startup parameters enjoy the highest priority, and the configured key format is `ModuleName.ProviderName.SettingKey`. For example, to overwrite the `core.default.restHost` parameter to `127.0.0.1`, add the configuration `-Dcore.default.restHost=127.0.0.1` to the startup parameters.
- 2) Specify parameters with environment variables. From the previous section, we have seen that almost every item in `application.yml` can be configured with environment variables.

```
query:
  graphql:
    path: ${SW_QUERY_GRAPHQL_PATH:/graphql}
```

With the above code as an example, if there is `SW_QUERY_GRAPHQL_PATH` in the environment variable and the value is `/api`, the configuration value of `query.graphql.path` is `/api`; if this environment variable does not exist, then the value configured in the memory is `/graphql`. In addition, priority-setting for multiple environments is supported in environment variable

configuration. Using `{REST_HOST $: $ {ANOTHER_REST_HOST: 127.0.0.1}}` as an illustration, the system first determines if there is `REST_HOST` in the environment variable; if present, the value of `REST_HOST` is used; if not, the system then looks for the environment variable `ANOTHER_REST_HOST`. Accordingly, where the environment is not configured, the last value to be used is `127.0.0.1`.

### 2.3.5 IP and port settings

In the core module, you can configure the IP and port for back-end OAP service monitoring. Where the system has multiple network cards and IPs, the OAP service can be paired up with a certain IP. The configuration is as follows:

```
core:
  default:
    restHost: 0.0.0.0
    restPort: 12800
    restContextPath:
    / gRPCHost:
    0.0.0.0
    gRPCPort: 11800
```

These are the main pairs of IP and port being configured:

- `gRPCHost` and `gRPCPort`: A pair of IP and port that uses gRPC for transmission. Most agents will use the gRPC protocol to communicate with the back-end OAP, as gRPC has better transmission performance and stability.
- `restHost` and `restPort`: A pair of IP and port that uses the REST style HTTP interface to receive data. Suitable where a language used by the Agent does not support the gRPC protocol. The UI module also uses this port for data query through GraphQL.

Note that if you have specified an IP other than `0.0.0.0`— say, for example, the IP has been set to `172.9.13.28`— then the client will only be able to access the service through this IP. Even if the Agent is running on machine `172.9.13.28`, access through unpaired IPs, such as `localhost` and `127.0.0.1`, will not be available.

### 2.3.6 Cluster management configuration

When used in an actual production environment, in order to ensure the stability and robustness of the service, a cluster model is required in the deployment for OAP back-end. This section will discuss in detail how to perform cluster management on the OAP back-end, including using traditional cluster management tools to manage physical machine clusters and Kubernetes to manage cloud-native clusters.

### A. *Traditional cluster management tools*

OAP provides a variety of cluster management coordination tools for cluster management. Their functions include detecting back-end instances, and facilitating communication by acting as a medium for instances to obtain nodes. OAP currently supports four cluster management tools: ZooKeeper, Consul, Nacos, and etcd.

#### *i) ZooKeeper*

ZooKeeper is an open source distributed coordination service. Many distributed services use it as a coordination service. In OAP, set the `application.yml` cluster as ZooKeeper using the following configuration (ZooKeeper 3.4 or above is required):

```
cluster:
  zookeeper:
    nameSpace: ${SW_NAMESPACE:""}
    hostPort:
      ${SW_CLUSTER_ZK_HOST_PORT:localhost:2181}
    # Retry Policy
    baseSleepTimeMs: 1000 # initial amount of time to wait between
    retries
    maxRetries: 3 # max number of times to retry
    # Enable ACL
    enableACL: ${SW_ZK_ENABLE_ACL:false} # disable ACL in
    default
    schema: ${SW_ZK_SCHEMA:digest} # only support digest
    schema
    expression: ${SW_ZK_EXPRESSION:skywalking:skywalking}
```

The main configuration items in the above configuration are described as follows.

- `hostPort`: ZooKeeper service address in the format of `IP1:PORT1, IP2:PORT2, . . . , IPn:PORTn`.
- `enableACL`: Whether to open the ACL access control for ZooKeeper.
- `schema`: ACL schema of ZooKeeper.
- `expression`: ACL expression.
- `baseSleepTimeMs` and `maxRetries`: Used to set up parameters of the ZooKeeper curator client. The following points should be noted:
  - If ACL access control is enabled and the `/skywalking` node already exists in your cluster, then you need to ensure that the OAP has CREATE, READ, and WRITE privileges in relation to the node.
  - If there is no `/skywalking` node in ZooKeeper, the OAP service will automatically create the node and grant rights to designated users, and `znode` will also grant READ privilege.
  - If the schema is set as digest, the password in the expression must be configured in clear text.



In some cases, the gRPC IP and port mentioned above are not suitable for communication between instances in the cluster. The IP and port must be specified separately to allow such communication. In this case, you can add the following two configurations: `internalComHost` and `internalComPort`. As the names suggest, these two parameters are specifically used to configure the IPs and ports for instance registry and service discovery under ZooKeeper's coordination, so as to allow communication between instances in the cluster. The following is an example for reference:

```
zookeeper:
  namespace: ${SW_NAMESPACE:""}
  hostPort:
    ${SW_CLUSTER_ZK_HOST_PORT:localhost:2181}
  #Retry Policy
  baseSleepTimeMs: ${SW_CLUSTER_ZK_SLEEP_TIME:1000} # initial amount of
    time to wait between retries
  maxRetries: ${SW_CLUSTER_ZK_MAX_RETRIES:3} # max number of times to retry
  internalComHost: 172.10.4.10
  internalComPort: 11800 # Enable ACL
  enableACL: ${SW_ZK_ENABLE_ACL:false} # disable ACL in default
  schema: ${SW_ZK_SCHEMA:digest} # only support digest schema
  expression: ${SW_ZK_EXPRESSION:skywalking:skywalking}
```

## *ii) Consul*

Consul is a service mesh solution that provides a variety of functionalities, such as service discovery, configuration, and KV storage. Here, we use Consul's service registration and discovery to coordinate the OAP service cluster instance. By setting the cluster as consul, it will be able to support Consul configuration. The configuration is as follows:

```
cluster:
  consul:
    serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
    # Consul cluster nodes, example:
    10.0.0.1:8500,10.0.0.2:8500,10.0.0.3:8500
    hostPort: ${SW_CLUSTER_CONSUL_HOST_PORT:localhost:8500}
```

Similar to the ZooKeeper registration service, if the instance node cannot use the IP and port configured by the gRPC of the core module, it can also set up its own IP and port for Consul service registration and discovery. The parameter names would still be `internalComHost` and `internalComPort`.

## *iii) Nacos*

Nacos defines itself as a platform for dynamic service registration and discovery, configuration and service management when building cloud-native applications. We can also simply use its

dynamic service registration and discovery functions for OAP service discovery by setting the cluster as nacos.

```
cluster:
  nacos:
    serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
    # Nacos cluster nodes, example:
    10.0.0.1:8848,10.0.0.2:8848,10.0.0.3:8848
    hostPort: ${SW_CLUSTER_NACOS_HOST_PORT:localhost:8848}
    # Nacos Configuration namespace
    namespace: ${SW_CLUSTER_NACOS_NAMESPACE:"public"}
```

#### *iv) etcd*

etcd is a storage system for storing high-availability key-value pairs in a distributed system. Simply set the cluster as etcd.

```
cluster:
  etcd:
    serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
    #etcd cluster nodes, example: 10.0.0.1:2379,10.0.0.2:2379,10.0.0.3:2379
    hostPort: ${SW_CLUSTER_ETCD_HOST_PORT:localhost:2379}
```

### *B. Kubernetes cluster management*

Kubernetes is an open source, cloud-native container cluster management platform. It aims to improve simplicity and efficiency in the deployment of container-based applications. The SkyWalking back-end can be easily deployed in the management platform, and benefits from the efficient management of Kubernetes, which promises a high availability of OAP and UI components.

Regarding the deployment of the OAP cluster in Kubernetes, apart from using the service discovery services introduced earlier (such as ZooKeeper and Consul) for cluster management, the Kubernetes platform also has the ability to act as a service discovery component. It can obtain the endpoint location of individual OAP instances by accessing the API Server, and allow mutual access between different nodes in the cluster.

You can enable Kubernetes cluster management using the following configuration:

```
cluster:
  kubernetes:
    watchTimeoutSeconds: ${SW_CLUSTER_K8S_WATCH_TIMEOUT:60}
    namespace: ${SW_CLUSTER_K8S_NAMESPACE:default}
    labelSelector: ${SW_CLUSTER_K8S_LABEL:app=collector,release=skywalking} uidEnvName:
    ${SW_CLUSTER_K8S_UID:SKYWALKING_COLLECTOR_UID}
```

The variables in the above configuration are defined as follows.

- `watchTimeoutSeconds` represents the timeout period for the cluster manager to monitor the API Server. It is used in the Java SDK for Kubernetes. Generally, the cluster manager uses the SDK to build a persistent connection with the API Server. If there is no data interaction within the timeout period, the cluster manager will disconnect from the API Server and try again. The default value is 60 seconds.
- `namespace` is the namespace for OAP's Deployment.
- `labelSelector` is the label selector for the OAP Pod. With the `labelSelector`, the cluster manager may discover the locations of other instances in the cluster.
- `uidEnvName` injects the unique Pod ID for cluster management.

Refer to Section 2.3.7 for a full deployment example.

### *C. Agent and cluster communication*

Based on the previous IP and port settings, the OAP server mainly provides gRPC and HTTP RESTful interfaces for receiving data from client's end. As the SkyWalking project evolves, it is expanding to support many language agents. It currently supports multiple language agents, including Java, Go, Net Core, PHP, C#, NodeJS, Golang, and LUA. At the same time, it also supports major tracing platforms such as Zipkin and Jaeger, as well as telemetry data of the highly popular service mesh, Istio. The OAP client's end receives a variety of data types and have the following types of configurations in `application.yml`:

- `receiver-trace`: Receives trace data from the client's end in various languages via gRPC and HTTP RESTful.
- `receiver-register`: Receives metadata registration information of services, service instances and endpoints from the client's end via gRPC and HTTP RESTful.
- `receiver-sharing-server`: By default, trace data are received through the IP and ports of the core module. You can set up a separate port for receiving trace data under this item to isolate the trace data and cluster management data from each other on the interface level.
- `service-mesh`: Receives agent data from the service mesh via gRPC.
- `receiver-jvm`: Receives monitoring metrics data from JVM via gRPC.
- `istio-telemetry`: Receives data from Istio's Bypass Adaptor via gRPC.
- `envoy-metric`: Receives data from Envoy's `metrics_service` and Access Log Service (ALS).
- `receiver_zipkin`: Receives trace data from the Zipkin client through the HTTP interface.
- `receiver_jaeger`: Receives trace data from the Jaeger client via gRPC.

Note that `receiver_zipkin` and `receiver_jaeger` can currently only run under the Tracing mode. In

other words, they can only collect and display the trace data of their respective clients, and cannot analyze metrics data.

### 2.3.7 Kubernetes deployment

SkyWalking provides an official repository for deploying OAP and UI components to the Kubernetes platform.<sup>2</sup> In Section 2.3.6, we have seen how the cluster management component of OAP utilizes the API Server of Kubernetes to coordinate each node in the cluster. In this section, OAP deployment and the relevant details of UI component deployment will be discussed.

Figure 2-2 shows the folder structure of the skywalking-kubernetes repository. There have been many changes in the installation methods in the past. Refer to the relevant documentation in the repository for the latest installation details. Some points to note for the repository are as follows:

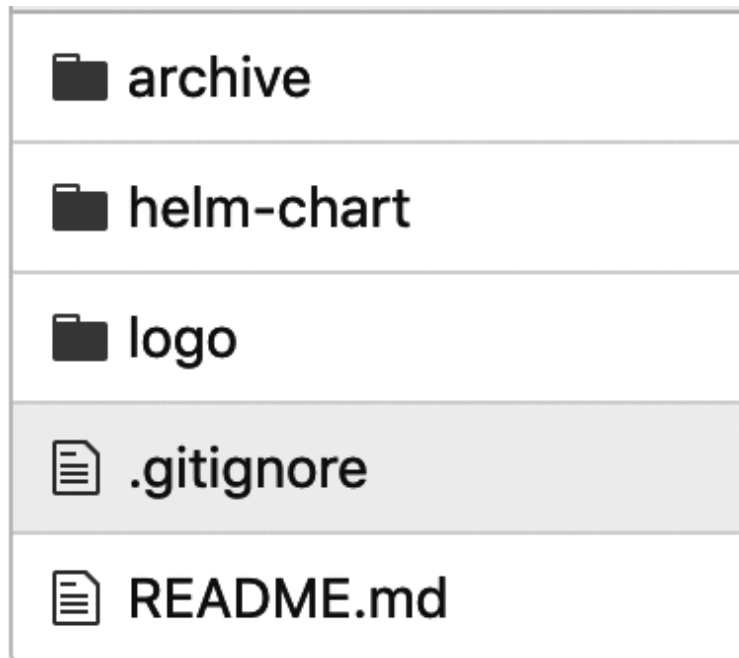


Figure 2-2 The folder structure of skywalking-kubernetes

#### A. *archive* folder

This folder contains original Kubernetes YML scripts in version 6.0. These are static scripts and cannot be applied to multiple installation environments. They are only for reference when users deploy to Kubernetes.

---

<sup>2</sup> <https://github.com/apache/skywalking-kubernetes>

## B. *helm-chart* folder

This is a Kubernetes YAML file that has been generated through Helm. Let's take version 6.4.0 as an example. Refer to Figure 2-3.

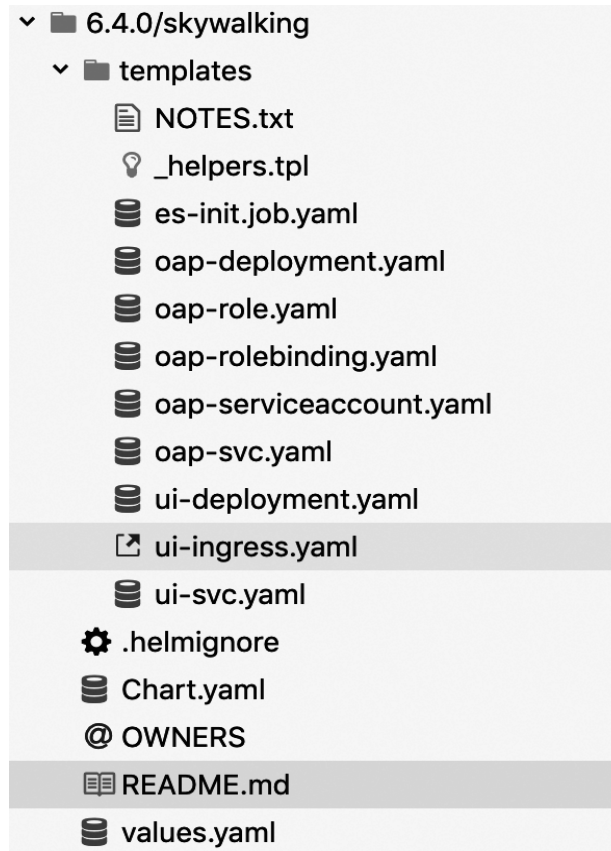


Figure 2-3 SkyWalking Helm Chart

We can see in the `Chart.yaml` file that the Chart is dependent on Elasticsearch (see Figure 2-4). Currently, the Chart only supports this type of storage module and the repository does not contain other types of storage modules. If users would like to use other types of storage modules, installation is required. To disable the Elasticsearch module, you can set `elasticsearch.enabled` as `false`.

The OAP mainly uses environment variables for configuration. If users want to overwrite multiple configuration files in a Docker container, they need to store the configuration files on the Kubernetes platform with ConfigMap, and then disable automatic generation of configuration files by the container through the environment variable `SW_LOAD_CONFIG_FILE_FROM_VOLUME=true`, and then mount these configuration files to the container. Currently, the Chart requires manual operation by the user.

The OAP role and rolebinding mainly serve to provide the OAP cluster manager with access to the API Server. Currently, the role provides three operation privileges for Pod, namely get, word, and list.

The OAP currently provides only internal access for the UI. If users would like to externally access the relevant OAP Endpoints, they have to add a new service or obtain access with port-forward.

```
apiVersion: v1
name: skywalking
home: https://skywalking.apache.org
version: 0.1.1
appVersion: 6.4.0
description: Apache SkyWalking APM System
icon: https://raw.githubusercontent.com/apache/skywalking-kubernetes/master/logo/s
sources:
- https://github.com/apache/skywalking-kubernetes
maintainers:
- name: hanahmily
  email: hanahmily@gmail.com
- name: innerpeacez
  email: innerpeace.zhai@gmail.com

dependencies:
- name: elasticsearch
  version: ~1.28.2
  repository: https://kubernetes-charts.storage.googleapis.com/
  condition: elasticsearch.enabled
```

Figure 2-4 SkyWalking's dependency on Elasticsearch

It should be emphasized that there is a `es-initjob` task in the deployment script, which is the initialization task of Elasticsearch. Its main function is to initialize the relevant indexes of Elasticsearch. A VM deployment starts from a node in the cluster, and then an index is created. For a Kubernetes cluster, if each Pod of Elasticsearch starts at the same time, it will lead to a concurrent index creation failure. Therefore, during the deployment of Kubernetes, the OAP startup does not create any indexes; rather, the task is responsible for creating all of the indexes. When OAP starts, you will notice in the back-end logs that it is waiting for index creation. Once the task is completed, the OAP instance can start normally.

The deployment of UI is very similar to the deployment of ordinary stateless services. Currently, Chart provides three external access modes for deployment: Ingress, NodePort, and LoadBalancer. You can use the parameters defined in the `values.yml` file to switch modes.

### 2.3.8 Back-end storage

SkyWalking's back-end storage implements a variety of storage forms that can be used in the production environment. You can choose which one to use according to the requirements of the technology stack and your proficiency in using them. If you are familiar with the architecture and code, you can also implement storage by yourself with the interface provided. Currently, Elasticsearch 6, MySQL, and TiDB are supported.

#### A. *Elasticsearch*

To choose Elasticsearch for storage, simply set the storage in application.yml as `elasticsearch`. At present, Elasticsearch is the most commonly used storage form. Elasticsearch clusters can be easily managed and horizontally expanded, and can adapt to data monitoring under wide-ranging conditions in various business scenarios.

By default, the OAP uses the `RestHighLevelClient` of the HTTP protocol to communicate with the server (i.e. the default Port 9200). See the following example of the configuration:

```
storage:
  elasticsearch:
    # nameSpace: ${SW_NAMESPACE:""}
    # user: ${SW_ES_USER:""} # User needs to be set when Http Basic
    # authentication is enabled
    # password: ${SW_ES_PASSWORD:""} # Password to be set when Http Basic
    # authentication is enabled
    # trustStorePath: ${SW_SW_STORAGE_ES_SSL_JKS_PATH:""}
    # trustStorePass: ${SW_SW_STORAGE_ES_SSL_JKS_PASS:""}
    clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:localhost:9200}
    protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
    indexShardsNumber: ${SW_STORAGE_ES_INDEX_SHARDS_NUMBER:2}
    indexReplicasNumber: ${SW_STORAGE_AS_INDEX_REPLICAS_NUMBER:0}
    # Those data TTL settings will override the same settings in core
    # module.
    recordDataTTL: ${SW_STORAGE_ES_RECORD_DATA_TTL:7} # Unit is day
    otherMetricsDataTTL: ${SW_STORAGE_ES_OTHER_METRIC_DATA_TTL:45} # Unit
    # is day
    monthMetricsDataTTL: ${SW_STORAGE_ES_MONTH_METRIC_DATA_TTL:18} # Unit is
    # month
    bulkActions: ${SW_STORAGE_ES_BULK_ACTIONS:2000} # Execute the bulk every
    # 2000 requests
    bulkSize: ${SW_STORAGE_ES_BULK_SIZE:20} # flush the bulk every 20mb
    flushInterval: ${SW_STORAGE_ES_FLUSH_INTERVAL:10} # flush the bulk
    # every 10 seconds whatever the number of requests
    concurrentRequests: ${SW_STORAGE_ES_CONCURRENT_REQUESTS:2} # the
    # number of concurrent requests
```

Users more familiar with Elasticsearch will know about these configurations, since these are the general configurations of Elasticsearch. The meaning of these configurations are briefly introduced below:

- `nameSpace`: if the namespace is configured, the index name generated by OAP in Elasticsearch will use the namespace as a prefix.
- `user/password`: If authentication is enabled in Elasticsearch, set the username and password required for access here.
- `trustStorePath/truststorePass`: The server can store the path and password for the trusted certificate after enabling HTTPS authentication.
- `clusterNodes`: The address for the Elasticsearch cluster. The multiple instances are separated by commas.
- `protocol`: The communication protocol. HTTP is used by default.
- `indexShardsNumber`: The number of shards for each index. For a private cluster, it can be set to the number of Elasticsearch instances to maximize performance.
- `indexReplicasNumber`: The number of replicas of the index shard. For SkyWalking, it can be set to 0, which means that no replicas are needed, in order to maximize performance.
- `bulkActions`, `bulkSize` and `flushInterval`: They respectively store the number of requests, request size, and flush interval time for Elasticsearch's batch processing. The OAP uses the BulkProcessor for batch writing. Once any of these parameters reaches the upper limit, batch writing to the Elasticsearch cluster will occur.
- `concurrentRequests`: The number of concurrent requests for writing to Elasticsearch.
- `recordDataTTL`, `otherMetricsDataTTL` and `monthMetricsDataTTL`: Configuration for data expiry time. This will be discussed in detail later.

We have seen that the monitoring data from SkyWalking can be stored in Elasticsearch. Besides, SkyWalking is also capable of receiving monitoring data from Zipkin and Jaeger and storing it in Elasticsearch. Simply set storage as `zipkin-elasticsearch` or `jaeger-elasticsearch`. The rest of the configurations are similar to the above descriptions.

### *B.MySQL/TiDB*

Similar to the configuration of Elasticsearch, if you want to use MySQL/TiDB as storage, simply set storage in `application.yml` as `mysql`, and configure database connection information in the `datasource-settings.properties` file. Since TiDB is fully compatible with MySQL, their configurations look the same. Here is an example of the configuration:



```
storage:
  mysql:
    metadataQueryMaxSize: ${SW_STORAGE_H2_QUERY_MAX_SIZE:5000}
```

The `datasource-settings.properties` are configured as follows. The connection pool is HikariCP. You can refer to its official documentation for detailed configuration.

```
jdbcUrl=jdbc:mysql://localhost:3306/swtest
dataSource.user=root
dataSource.password=root@1234
dataSource.cachePrepStmts=true
dataSource.prepStmtCacheSize=250
dataSource.prepStmtCacheSqlLimit=2048
dataSource.useServerPrepStmts=true
dataSource.useLocalSessionState=true
dataSource.rewriteBatchedStatements=true
dataSource.cacheResultSetMetadata=true
dataSource.cacheServerConfiguration=true
dataSource.elideSetAutoCommits=true
dataSource.maintainTimeStats=false
```

Note that the MySQL driver package is not included in the project by default. If you would like to use MySQL or TiDB, manually download the relevant driver and place it in the `oap-libs` folder.

Most readers will be fairly familiar with MySQL. As for the emerging TiDB, it has opened a new world for distributed relational databases. Based on its own website description, TiDB is an open source distributed Hybrid Transactional and Analytical Processing (HTAP) database designed by the company PingCAP. It combines the benefits of the traditional RDBMS and NoSQL. TiDB is compatible with MySQL, supports unlimited horizontal scaling, and possesses a high level of consistency and availability. The goal of TiDB is to provide a one-stop solution for situations where Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP) are involved.

### *C. Data expiration settings*

In addition to the metadata of applications, the following two types of observable data are persisted in the storage system of SkyWalking:

- **Record:** Contains trace data and alarm record information. This type of data will only be recorded sequentially, and will not be aggregated or updated.
- **Metrics data:** For example, the response time data, heat map, success rate, counts per minute (CPM), error rate and other information of p99/p95/p90/p75/p50 (where p is the response time for the percentile request, e.g., p99 represents a user response time of 99%) in various dimensions (such as all, service, and endpoint). This kind of metrics data is usually calculated based on units such as minutes, hours, days, and months, and stored in different tables.

Generally as the amount of data increases, data will be deleted after a certain period in order to save storage space and improve resource utilization. In the core module of OAP, each storage implementation comes with a uniform configuration of automatic deletion capability and data retention time. The configurations and the relevant descriptions are as follows:

```
core:
  default:
    enableDataKeeperExecutor: ${SW_CORE_ENABLE_DATA_KEEPER_EXECUTOR:true} #
      Turn it off then automatically metrics data delete will be close.
    recordDataTTL: ${SW_CORE_RECORD_DATA_TTL:90} # Unit is minute
    minuteMetricsDataTTL: ${SW_CORE_MINUTE_METRIC_DATA_TTL:90} # Unit is
      minute
    hourMetricsDataTTL: ${SW_CORE_HOUR_METRIC_DATA_TTL:36} # Unit is hour
    dayMetricsDataTTL: ${SW_CORE_DAY_METRIC_DATA_TTL:45} # Unit is day
    monthMetricsDataTTL: ${SW_CORE_MONTH_METRIC_DATA_TTL:18} # Unit is month
```

- enableDataKeeperExecutor: Whether to enable automatic data deletion function.
- recordDataTTL: Validity time of Record data in minutes.
- minuteMetricsDataTTL: Validity period of Metrics data in minutes.
- hourMetricsDataTTL: Validity period of Metrics data in hours.
- dayMetricsDataTTL: Validity period of Metrics data in days.
- monthMetricsDataTTL: Validity period of Metrics data in months.

For the recommended storage Elasticsearch, you can also configure the private configuration for Elasticsearch storage, which will override the configuration of the core module.

```
# Those data TTL settings will override the same settings in core module.
recordDataTTL:. $ {SW_STORAGE_ES_RECORD_DATA_TTL:. 7} # Unit is Day
otherMetricsDataTTL: $ {SW_STORAGE_ES_OTHER_METRIC_DATA_TTL: 45} # Unit is Day
monthMetricsDataTTL: $ {SW_STORAGE_ES_MONTH_METRIC_DATA_TTL: 18 is} # the Unit is
month
```

Different from the core module, in addition to providing recordDataTTL, minuteMetricsDataTTL, hourMetricsDataTTL, dayMetricsDataTTL and monthMetricsDataTTL configurations, you can also configure otherMetricsDataTTL. otherMetricsDataTTL will affect the data validity period in the unit of minutes, hours, and days. If otherMetricsDataTTL is configured, it will be used first in the data validity period in minutes, hours, and days; otherwise, the previous configuration will be used.

### 2.3.9 Setting sampling rate of the server

It is possible to set the sampling rate for the agent of SkyWalking. By manipulating this setting, the ratio of trace data being reported can be controlled. However, if all agents are required to perform configuration modification and upgrade maintenance, it will become a very tedious process. An abrupt upgrade is bound to put a lot of pressure on the system storage. On the other hand, it is impossible to modify the agent configuration within a short time. In this case, you can set the sampling rate of the server. Setting the sampling rate on the server's end will only discard part of the trace data, and will not affect the accuracy of the metrics data statistics (such as that of service, service instance and endpoint). Therefore, it can be used to cope with the storage system pressure brought by high volumes of data reporting. Note that although setting sampling on the server's end will lead to the discarding of certain trace data, the OAP will still try to maintain the integrity of the traces.

Simply configure `sampleRate` in the `receiver-trace` module in `application.yml`, and set the precision to 1/10 000. If you don't want OAP to discard data, just set the `sampleRate` to 10 000.

```
receiver-trace:
  default:
    bufferPath: ../trace-buffer/ # Path to trace buffer files, suggest
      to use absolute path
    bufferOffsetMaxFileSize: 100 # Unit is MB
    bufferDataMaxFileSize: 500 # Unit is MB
    bufferFileCleanWhenRestart: false
    sampleRate: ${SW_TRACE_SAMPLE_RATE:1000} # The sample rate precision is
      1/10000. 10000 means 100% sample in default.
```

In the example, `bufferPath`, `bufferOffsetMaxFileSize`, `bufferDataMaxFileSize`, and `bufferFileCleanWhenRestart` are all cache configurations for receiving trace data files. They are used for temporary transition of data analysis and storage.

### 2.3.10 Alarm settings

In SkyWalking, you can flexibly set the alarms for various metrics. All alarm rules are configured in `config/alarm-settings.yml`. The configuration mainly consists of the following two parts:

- Alarm rule configuration: Configure the alarm rules for the conditions to be met by specified metrics.
- Webhooks: The external interface for alarm function when the alarm is triggered.

#### A. Alarm rule configuration

Let's first look at an example configuration. You can configure alarms for different metrics based on specified frequencies.

```

rules:
  # Rule unique name, must be ended with '_rule'.
  endpoint_percent_rule:
    # Metrics value need to be long, double or int
    metrics-name: endpoint_percent
    threshold:
      75 op: <
    # The length of time to evaluate the metrics
    period: 10
    # How many times after the metrics match the condition, will trigger alarm
    count: 3
    # How many times of checks, the alarm keeps silence after alarm triggered, default as same as
      period.
    silence-period: 10

  service_percent_rule:
    metrics-name: service_percent
    # [Optional] Default, match all services in this metrics
    include-names:
      - service_a
      - service_b
    threshold: 85
    op: <
    period: 10
    count: 4

```

- **rule name:** Name of the alarm rule. All names, such as `endpoint_percent_rule` and `service_percent_rule` in the example, must end with `_rule`, or else it will not be recognized.
- **metrics-name:** Metrics for the statistics. The actual value can only be type long, int or double. For example, the `endpoint_percent` and `service_percent` in the above example are metrics from the OAL (OAL to be introduced in Chapter 7). Different alarm metrics can be configured according to actual business needs.
- **threshold:** The threshold for triggering the alarm in the `metrics-name` configuration.
- **op:** Comparison operator. Can be set to greater than, less than or equal to.
- **period:** The time window for checking the alarm rules.
- **count:** During the time window, if the number of times of trigger reaches the configured number, an alarm will be generated.
- **silence-period:** The time interval between alarms to prevent multiple alarms within a short period of time.
- **include-names:** Specify the entity names for alarm. For service configuration, you can configure the name of the service; for endpoint, you can configure the name of the endpoint.

In order to facilitate easy deployment by users, SkyWalking's default release package already contains some commonly used alarm rules. The alarm rules can be viewed in `config/alarm-settings.yml`. You can also customize and modify these rules. The default rules are:

- In the last 3 minutes, the average response time of the application exceeds 1 second.
- In the last 2 minutes, the application's successful request rate is less than 80%.
- In the last 3 minutes, 90% of applications have a response time of more than 1 second.
- In the last 2 minutes, the average response time of the application instance dimension exceeds 1 second.
- In the last 2 minutes, the average response time of the endpoint dimension exceeds 1 second.

### B. Webhooks

When the alarm rule reaches the trigger condition, the alarm information needs to be delivered to the web service specified by the user through Webhooks. The format of the delivered alarm message is as follows, and the user needs to receive and parse the alarm message in this format.

- 1) Use HTTP to send and the method is POST.
- 2) Content-Type is application/json with the UTF-8 encoding.
- 3) The data structure is defined as `List<org.apache.skywalking.oap.server.core.alarm.AlarmMessage>`. The detailed structure is as follows:

```
public class AlarmMessage {  
    private int scopeId;  
    private String name;  
    private int id0;  
    private int id1;  
    private String ruleName;  
    private String alarmMessage;  
    private long startTime;  
}
```

The variables are explained as follows.

- `scopeId`: Type ID. You can check the different meanings of different `scopeIds` through `DefaultScopeDefine`.
- `name`: The specific name of the scope type, such as application name and endpoint name.
- `id0`: The entity ID relating to the specific scope, which corresponds to name in the database.
- `id1`: Currently not used.
- `ruleName`: The name of the rule defined in `alarm-settings.yml`.
- `alarmMessage`: Specific message for the alarm.

- `startTime`: Value of the millisecond when triggered by the alarm message.

### 2.3.11 Exporter settings

We have seen that SkyWalking offers important functions including collection and aggregation of application monitoring data, as well as data storage, data analysis, and alarm display. However, in certain use cases, secondary analysis and development of the data may be required. Then, the processed data will have to be forwarded again. The Exporter plays a key role in such a process.

The Exporter here refers to the `Metrics Exporter`. It is responsible for exporting the `Metrics` data after it has been analyzed by the OAP. If you need this function, you can enable the configuration in `application.yml`. The example is shown as follows. Currently, it supports exporting in the form of gRPC Exporter. This requires users to open a service port to receive metrics data sent by the OAP in accordance with the specified protobuf format (refer to `metric-exporter.proto` for the proto files).

```
exporter:
  grpc:
    targetHost: ${SW_EXPORTER_GRPC_HOST:127.0.0.1}
    targetPort: ${SW_EXPORTER_GRPC_PORT:9870}
```

The configuration is described as follows.

- `targetHost`: Target IP, which is the user-defined service address IP.
- `targetPort`: User-defined service port used to receive Metrics data sent by the OAP.

### 2.3.12 Detailed UI deployment

UI deployment is mainly composed of the deployment of a webapp proxy layer and front-end page. During compilation, `apm-webapp` automatically compiles the `skywalking-ui` front-end page module into the final output file. Therefore, we will only focus on how `apm-webapp` should be configured. The core configuration of `apm-webapp` is located in `webapp/webapp.yml`. The configuration file mainly consists of the following two parts.

- Monitoring port of the UI: Port for accessing the page through the browser.
- Port of the back-end OAP: No business logic underlies the webapp itself. All queries are forwarded to the REST service port of the OAP service through the Zuul proxy, and then pass through GraphQL. The default address is `127.0.0.1:12800`.

```

server:
    port: 8080

collector:
    path:
        /graphql
    ribbon:
        ReadTimeout: 10000
        # Point to all backend's restHost:restPort, split by ,
        listOfServers: 127.0.0.1:12800

```

After the configuration is complete, if there is no irregularity, start the UI module through `bin/webappService.sh(.bat)` to run normally.

## 2.4 Introduction to UI

SkyWalking's Dashboard is divided into three parts: top, middle, and bottom. The top includes the Feature Tab Selector Zone and the Reload Zone at top right. The middle part is the panel content. The bottom part is the Time Selector Zone. Functionalities vary for each version. Figure 2-5 is the official UI rendering of SkyWalking 7.0.

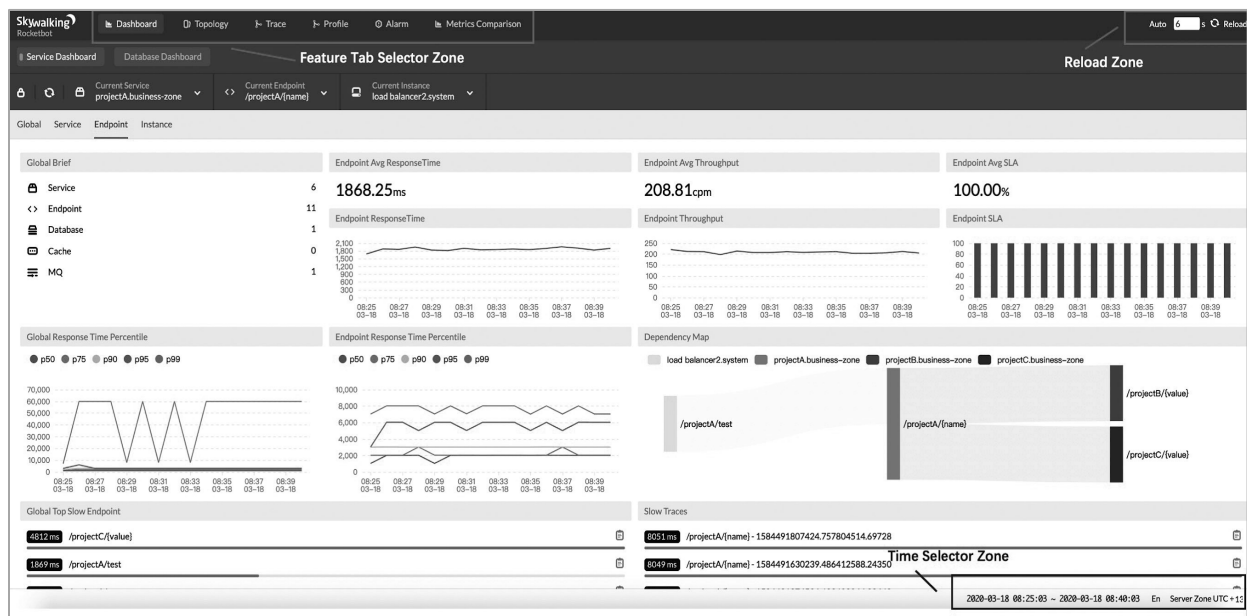


Figure 2-5 Official UI rendering of SkyWalking 7.0

### 2.4.1 Introduction to Dashboard

Dashboard provides service, service instance, Endpoint, database and other metrics-related information in the entity dimension, as shown in Figure 2-6.

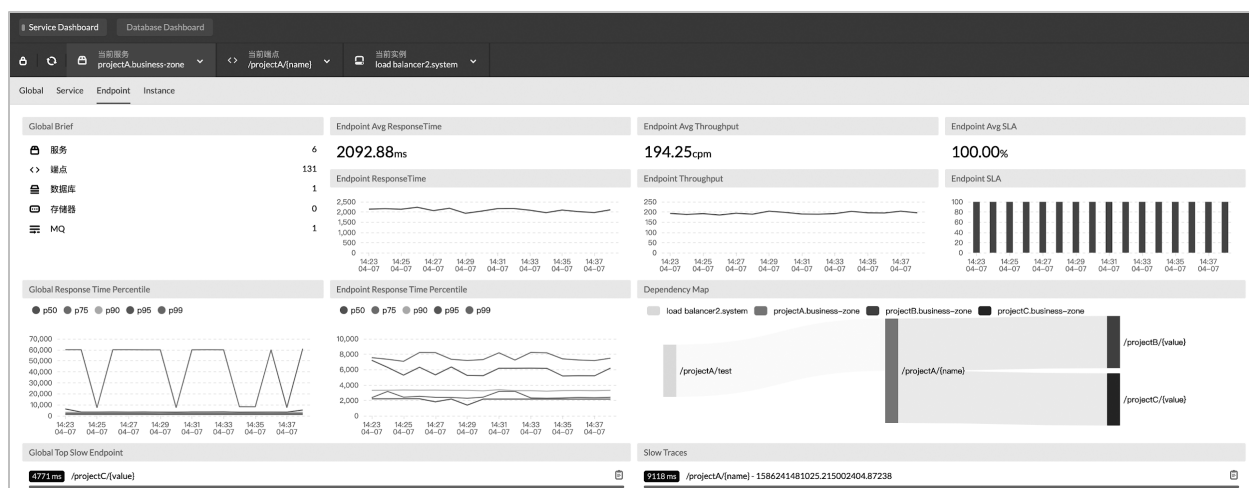


Figure 2-6 Metrics related information in the Dashboard entity dimension

The service instance contains several typical metrics, as shown in Figure 2-7.

- CPM: The number of calls per minute.
- Apdex: A standardized application performance index: Users can learn more about it [in this blog article](#).
- Average response time and response time percentile: The percentile reflects the long tail effect of response time. For example, p99=200ms means that 99% of the request response time can be less than or equal to 200ms.
- SLA: Indicates the success rate in SkyWalking.

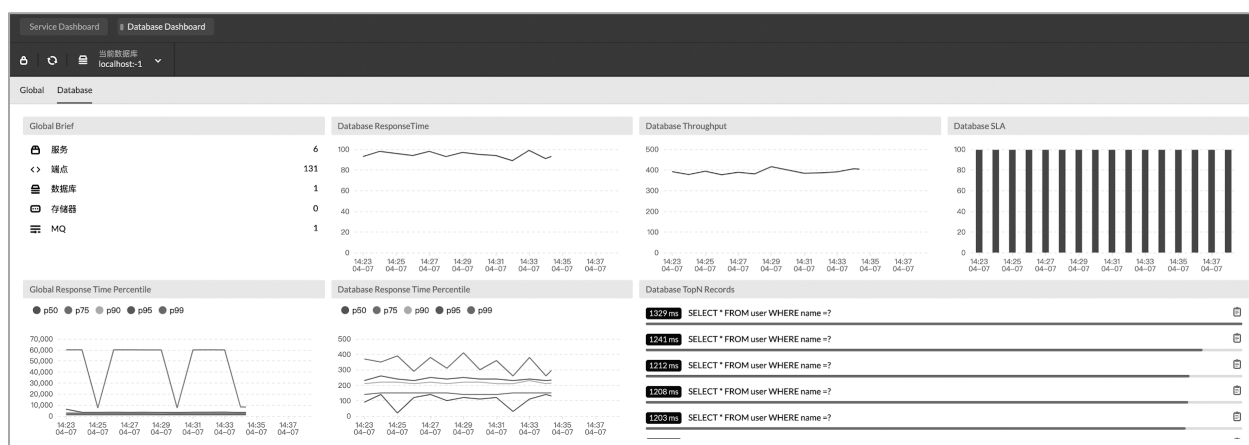


Figure 2-7 Service instance metrics

The database view shows the data performance index data calculated based on data collected by the application agent. The response time, success rate, and SQL slow queries in this calculation, including



performance problems caused by network issues and client program failures, are all collected by the client.

## 2.4.2 Introduction to Topology

The topology diagram as shown in Figure 2-8 represents the overall topology structure analyzed based on upstream data from the agent. The topology map supports clicking to display and drilling down the performance statistics, trace and alarm functions of a single service. You can also click on the dotted lines in the topology map to display the performance metrics between services and service instances, as shown in Figure 2-9.

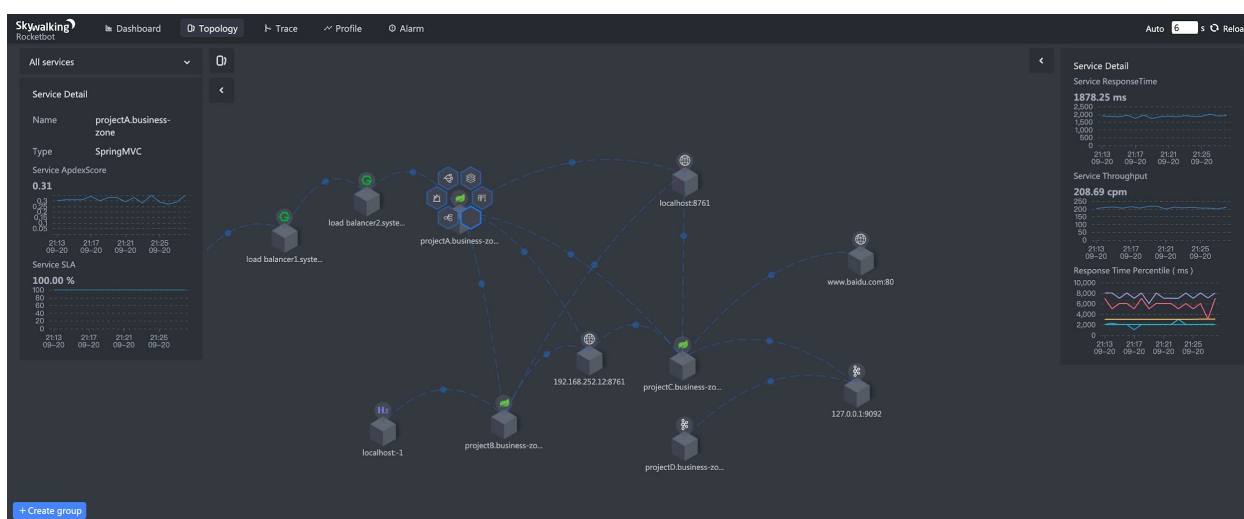


Figure 2-8 Topology diagram

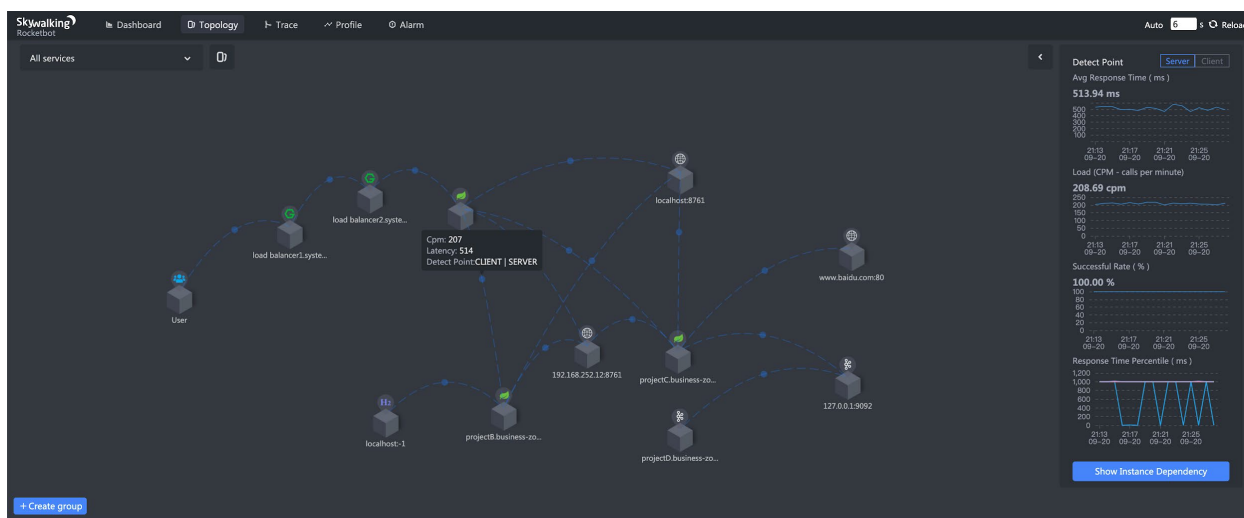


Figure 2-9 Expansion of the lines in the topology diagram

## 2.4.3 Trace View

Trace is a typical view of distributed tracing. SkyWalking provides three presentation formats, namely a list (see Figure 2-10), a tree diagram (see Figure 2-11) and a table (see Figure 2-12). Different diagrams allow users to view trace data from different perspectives, especially the elapsed time between Spans.



Figure 2-10 Trace view – List

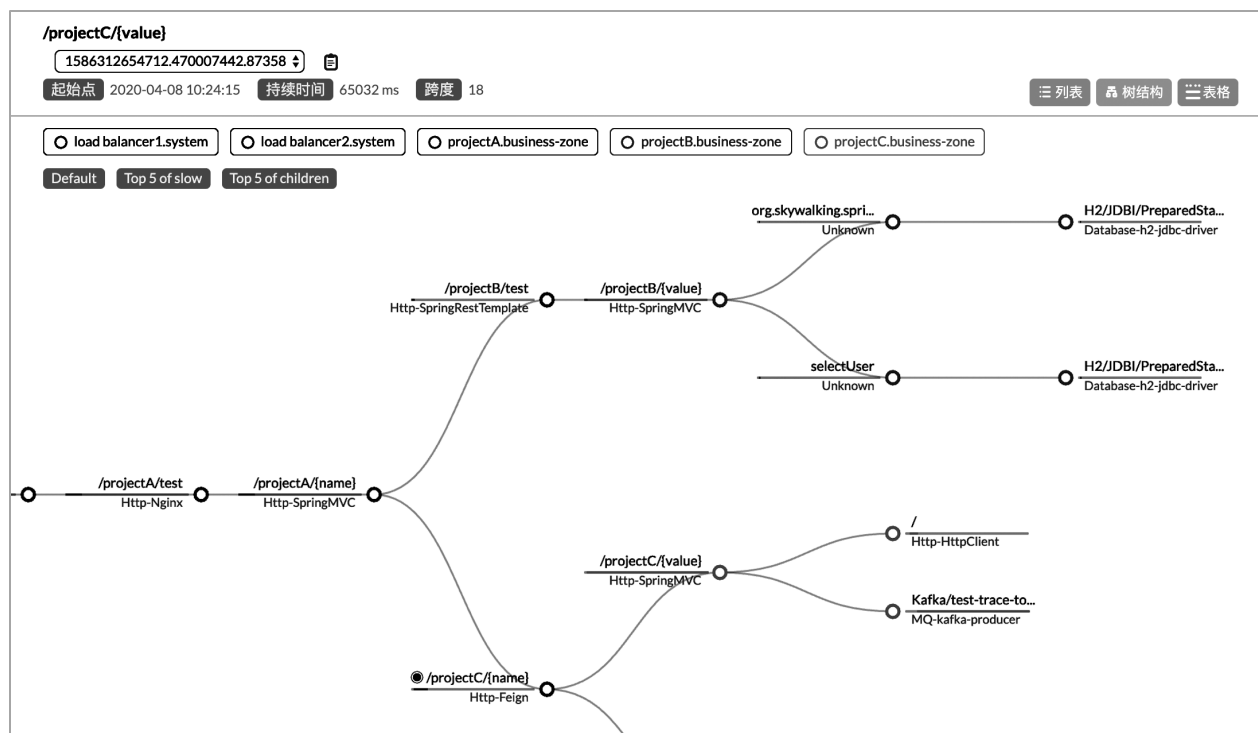


Figure 2-11 Trace view - Tree diagram

/projectC/{value}

1586312654712.470007442.87358

起始点

2020-04-08 10:24:15

持续时间

65032 ms

跨度

18

三列表

树结构

表格

Method	Start Time	Exec(ms)	Exec(%)	Self(ms)	API	Service
▼ /projectA/test	2020-04-08 10...	8309	<div></div>	0	Nginx	load balancer1.system
▼ /projectA/test	2020-04-08 10...	8309	<div></div>	1	Nginx	load balancer1.system
▼ /projectA/test	2020-04-08 10...	8308	<div></div>	0	Nginx	load balancer2.system
▼ /projectA/test	2020-04-08 10...	8308	<div></div>	0	Nginx	load balancer2.system
▼ /projectA/{name}	2020-04-08 10...	8308	<div></div>	0	SpringMVC	projectA.business-zone
▼ /projectB/test	2020-04-08 10...	1271	<div></div>	1	SpringRestTe...	projectA.business-zone
▼ /projectB/{value}	2020-04-08 10...	1270	<div></div>	1000	SpringMVC	projectB.business-zone
▼ org.skywalking.springcloud.test.projectb.dao.DatabaseOperateDa	2020-04-08 10...	0	<div></div>	0	-	projectB.business-zone
H2/JDBI/PreparedStatement/execute	2020-04-08 10...	0	<div></div>	0	h2-jdbc-driver	projectB.business-zone
▼ selectUser	2020-04-08 10...	270	<div></div>	0	-	projectB.business-zone
H2/JDBI/PreparedStatement/execute	2020-04-08 10...	270	<div></div>	270	h2-jdbc-driver	projectB.business-zone
▼ /projectC/{name}	2020-04-08 10...	7037	<div></div>	0	Feign	projectA.business-zone
▼ /projectC/{value}	2020-04-08 10...	65032	<div></div>	1004	SpringMVC	projectC.business-zone
/	2020-04-08 10...	4028	<div></div>	4028	HttpClient	projectC.business-zone
Kafka/test-trace-topic/Producer	2020-04-08 10...	60000	<div></div>	60000	kafka-produc...	projectC.business-zone

Figure 2-12 Trace view - Table

We have introduced the basics of UI. In Chapter 3, we will introduce the operation details and usage of the UI in more detail.

## 2.5 Chapter summary

In this chapter, we have gone through the deployment and configuration of SkyWalking's JavaAgent, back-end, and UI modules. We are confident that you have by now successfully deployed your own SkyWalking environment and experienced the convenience and power of SkyWalking. In the next chapter, we will enter the production environment and carry out live operations.

## Chapter 3:

# Live operations for Apache SkyWalking

---

After reading the first two chapters, you will already have had a preliminary understanding of SkyWalking. But without hands-on experience, it is impossible to fully appreciate the benefits of SkyWalking.

This chapter will introduce to you the two major architectural patterns and how SkyWalking approaches each. Then, a specific example will be used to show how to use SkyWalking for monitoring. Based on this example, we will introduce in detail the meaning of various monitoring metrics. You can compare different examples to gain a better understanding of these metrics. Finally, you will be guided on how to configure the monitoring module and set monitoring targets.

It is recommended that you operate the systems described here as you read, as this will promise better learning results.

## 3.1 SkyWalking and monolithic architecture

Monolithic application architecture is a widely used back-end system architectural model with a long history. In this section, you will learn about the evolution, advantages and disadvantages of this architecture, as well as receive guidance from SkyWalking on monitoring a monolithic architecture.

### 3.1.1 What is a monolithic architecture

Monolithic architecture is an architecture constructed by a single component, and is often an independent process separate from startup. “Monolith” literally means “a single piece of rock”. Single-

body applications have a single codebase and multiple modules. From the perspective of functionality, modules are generally classified into business modules and technology modules. Monolithic applications generally use a single build system to build the entire application and its dependent repositories, and it also has a single executable and deployable binary release package.

The software engineering community has long used this method to develop enterprise-level applications. Many companies have spent years building them. This architecture is sometimes also called a multi-layer system architecture, since the monolith is divided into three or more application layers, including the presentation layer, business logic layer, data storage layer, and application layer. Desktop browsers dominate today's market, so companies are interested in desktop or laptop devices with web browsers as the client, without the need to use an API to share data. The main reason is that browsers can only read HTML, CSS, and JavaScript. Although companies use Enterprise Data Bus (EDB), Electronic Data Interchange (EDI), and other data exchange formats for back-end component interaction, the use of a monolithic architecture is also able to meet their business needs.

A decade ago, most applications simply adopted the monolithic model. Word had it that the back-end of the Chinese e-commerce marketplace Taobao consisted of a single Java application. Taobao's senior architects subsequently confirmed this. This was a common phenomenon at the time. To further discuss monolithic architectures and their limitations, we will use a practical example for illustration.

Let's say there is an e-commerce application. The program is very simple. It pulls orders from the user, performs inventory processing, checks the available inventory, and sends the goods to the user. This application contains a variety of components: the front-end UI responsible for interacting with users, and back-end services such as checking inventory, maintaining bonus points, and sending customer orders.

A typical monolithic application architecture is shown in Figure 3-1. Apache or Nginx is used as the front-end load balancer. Java's Web application acts as the service layer. The data layer is a relational database. Let's illustrate using the example of MySQL.

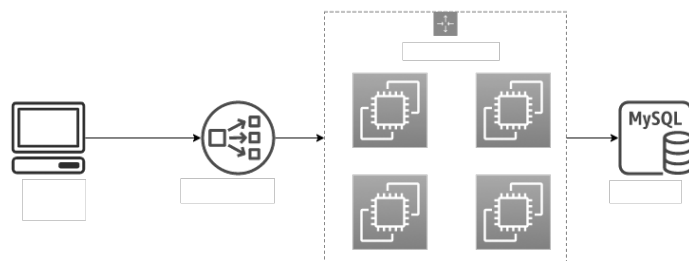


Figure 3-1 Monolithic architecture diagram

### 3.1.2 Advantages and disadvantages of monolithic application architecture

The advantages and disadvantages of this solution are outlined as follows. These are the major advantages of the monolithic architecture:

- First, it is easy to develop. The target tools and IDEs currently developed provide excellent support for this development model.
- Second, it is easy to deploy locally. You can easily use the War package to deploy it to a more suitable application, including Tomcat, Weblogin, and Websphere. The deployment process for these containers is very simple.
- Third, it is easy to scale. With the use of front-end load balancers such as Nginx and Apache, or with F5 for a higher performance, it is easy to evenly distribute traffic to the multiple instances being run.

But with the growth of the scale of softwares and the size of their development teams, the drawbacks of this architecture have become increasingly noticeable.

- First, due to the large size of the codebase of a monolithic application, it is very difficult for developers to modify the codes. At the same time, it has become extremely challenging for new team members to understand the entire code structure. This has significantly slowed down the overall development progress. Meanwhile, boundaries between modules are blurring. Therefore, it is rather difficult to build a uniform development repository to support all projects.
- Second, it adds to the burden of the local IDE. The large code structure has very high requirements for compilation and startup. If the developer's local development machine configuration is unable to meet the demands, there will be increased risk of failure to start.
- Third, it also adds to the burden of Web containers. Due to the large space required for the build packages (often exceeding 500MB), some open source Web containers are unable to load these applications. In this case, many would resort to commercial projects instead. However, the specifications required are often even higher, and generally there are licensing requirements. It follows that a lot more expenditure will be required for such a large-scale application deployment.
- Fourth, it is more difficult to conduct continuous construction. Since the codebase is relatively large, the high frequency for compilation and release entails a very time-consuming process. In the event that a certain part of the CI workflow fails, and that part is not introduced by the developer who operates the CI, the progress is bound to be severely slowed down.
- Fifth, the system scalability is dedicated only to the horizontal scalability of traffic. It is only the application traffic that can be scaled, not the data behind such traffic. There is a single MySQL node in Figure 3-1. All applications need to access this single data center. If the volume of data increases exponentially, a single data node is often unable to be scaled as required. Ultimately, this inability to scale extends to the entire application.

### 3.1.3 Applicability of SkyWalking to the monolithic architecture

After discussing the pros and cons of the monolithic architecture, the question to ask is whether it is possible to solve the issues with the monolithic architecture using the distributed tracing systems of SkyWalking. The traditional view is that tracing applications are not suitable for monolithic applications. Let's delve into this question. To clarify, tracing applications like SkyWalking can be applied to monolithic applications. SkyWalking provides a manual probing mode for LocalSpan on the agent's end that can show the call relations within the application from trace. Although this achieves the purpose of tracing the process, it is not a very cost-effective solution for the following reasons.

Since the deployments required for monolithic applications is relatively fewer, if a set of more complex and resource-consuming tracing applications are deployed at the same time, the overall cost performance would be rather low. In tracing, the 1:N resource evaluation model is used to describe the number of application instances required to meet the demand. In this ratio calculation, the overall cost performance for monolithic applications would be very low.

Second, the alternatives to a tracing system are far more powerful than what the tracing system is capable of. These include log systems, profile tools, remote debugging tools, and other troubleshooting devices designed for specific applications. They are all excellent methods for solving performance problems of monolithic applications, not to mention that they are much more efficient, reliable and cost-effective compared to a tracing system.

For these two reasons, SkyWalking's tracing system is not suitable for use on monolithic applications.

## 3.2 SkyWalking and the microservice architecture

The traditional data exchange format is not compatible with mobile applications, and the growing demand for mobile applications has made changes in the back-end architecture necessary. This is the main reason for the migration of the monolithic architecture to the microservice architecture.

As shown in Figure 3-2, the microservice architecture is an architectural method that uses multiple small units called services to build large enterprise-level application clusters. For example, an e-commerce microservice architecture cluster includes services such as product display, shopping cart, order, payment, and inventory.

These services can be deployed and tested separately. In this way, they can be handed over to different teams for independent maintenance. Uniform protocols, such as Dubbo, gRPC, and REST, are often used for service-to-service interactions.

The services can run on a single host or multiple hosts, and these services all run in their own independent processes. Each service can have its own database or storage layer. It is also possible to share a common data storage.

Microservices architecture doesn't only improve code and structure. Rather, it has essentially changed the overall back-end architecture in terms of the work process, and can bring about major cultural change for organizations that adopt them. Microservice is not the ultimate solution for every application, but it is indeed a possible solution for large enterprise applications.

As mentioned earlier, the traditional monolithic architecture might gradually lead to problems such as over-complicated codes, difficulty in maintenance, and low scalability. These problems can be solved with the use of a microservice architecture, which brings in the following benefits:

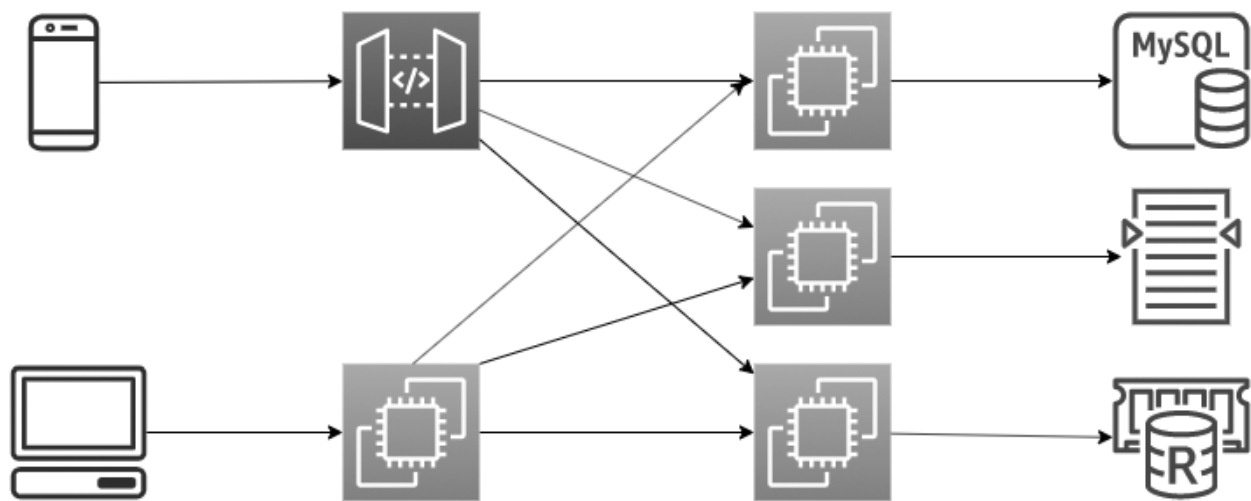


Figure 3-2 Microservice architecture diagram

- Each microservice is very small and focuses on specific functions or business requirements
- Microservices can be independently developed by a small development team (usually 2 to 5 developers)
- Microservices are loosely coupled, which means that each service is independent in its development and deployment
- Microservices can be developed in different programming languages, and allow the use of the latest technologies (frameworks, programming languages, programming practices, etc.)
- Microservices allow easy and flexible integration of automatic deployment using continuous integration tools
- Microservices are easy to scale according to needs



However, there are also shortcomings in microservices. For example, the substantial increase in operation overhead, requirement of DevOps skills, and complicated management and bug tracing due to the nature of distributed systems are all challenges. It is particularly important to ask questions on system observability. We pose the following questions:

- How many services does each call go through?
- When processing the request, what operations were conducted in each service?
- If the request slows down, where is the bottleneck?
- If the request fails, which service or part of the service is the reason for the problem?
- What is the difference between exception requests and normal requests?
- Why are some frequently called services not called in a certain request? Or why are some services called infrequently?
- What is the critical path of the call?

SkyWalking is the key to answering these questions. Now let us dive into some technical details of microservices and see how SkyWalking solves these problems.

### 3.2.1 Remote procedure call

Remote procedure call (RPC) is a type of data interaction mode between services. To put it simply, RPC abstracts network communication into remote procedure calls, making them as convenient as calling local subroutines. In this way, the level of complexity of communications is reduced. Developers no longer need to pay attention to the details of network programming. Instead, they can devote more time and energy into realizing the business logic, which in turn improves work efficiency. The core issue is accessibility. Let's consider the two main issues that it can address.

The first aspect is the network protocol. It consists of five layers, including the HTTP protocol, the gRPC protocol encapsulated on the HTTP/2 protocol, and the Dubbo protocol based on the TCP protocol.

The second aspect is service discovery, meaning how two services discover each other. The technologies involved are mainly DNS domain name resolution and the registry.

These two main functions constitute the core capabilities required to build a remote procedure call framework.

There are also some attractive additional features, including circuit breaking, mTLS security enhancement, traffic control, and non-functional requirements such as high availability, fault detection, and automatic transfer of bad requests. These enhancements and core components improve the convenience, reliability and performance of calls between services. At the same time, it makes the overall microservice environment more easily maintainable. It is crucial for a tracing framework like SkyWalking to have a good remote procedure call framework. A good remote procedure call framework must be able to carry some additional information, such that the tracing service can make use of this function to spread its own unique information across the trace and generate a data trace. The detailed process will be elaborated in Chapter 5. Here, we will briefly introduce the support for the HTTP protocol. SkyWalking uses HTTP header information and adds its own unique information header. This information header will be propagated throughout the trace. In this way, the service call process and request delays are marked on the trace.

SkyWalking provides the most extensive support for the remote procedure call framework. From the official plug-in repository, you can see that most are remote procedure call framework plug-ins. This shows that the remote procedure call framework is quickly flourishing. The leading frameworks in this field include Apache Dubbo, gRPC and Spring Cloud, and SkyWalking offers well developed solutions for them all.

### 3.2.2 External services

In addition to remote procedure calls, an important part of the microservice environment is its external services. External services usually include these types: middleware, data storage and API service.

#### *A. Middleware*

Middlewares are generally classified into the following two types.

The first type is the message queue. Queues such as Kafka and RocketMQ are commonly used for high-performance data processing. SkyWalking provides excellent support for this kind of message queue middleware, which can trace how data is written to the queue and how it is consumed. As data is processed asynchronously in the message queue, it leads to difficulty in locating the queue problem and the operation and maintenance of the message queue. For this issue, SkyWalking offers a good solution.

The second type is related to data storage, that is the database middleware. A common usage model is the data sharding middleware, such as Apache ShardingSphere. With SkyWalking, you can see how some database middleware splits a request into multiple database instances. You can also track how a database query is executed between data instances, which data instance responds more slowly, and which database instance execution has produced an error.

## *B. Database*

The database is a type of data storage service. SkyWalking supports major open source relational databases, including common H2, MySQL, and PG databases. For commercial databases like Oracle, SkyWalking supports them with the help of the community, and related codes and constructions are placed in the community repository. This is so mainly because components with commercial agreements are not allowed to be included in open source projects.

At the same time, SkyWalking also supports some new databases, such as the in-memory database Redis and the document-oriented database MongoDB.

At present, SkyWalking implants relevant plug-ins in the database driver to check the client's access to the database, and does not intrude upon the database server.

## *C. API service*

The last type of external service is the API service. Through SkyWalking, you can see the process of calling third-party APIs, and observe the quality of these services and their impact on the critical path of the system. On the plug-in level, SkyWalking supports a variety of API access. It supports the HTTP protocol and the gRPC protocol through auto-event tracking on the client's end. With these plug-ins, users can easily trace these external API services.

Most importantly, with SkyWalking's efficient and user-oriented customization capabilities, users can expand the existing open source plug-in repository according to their own system construction requirements, and include most of the key internal components of the enterprise within the tracing scope.

For example, most companies with good legacy IT systems in place will have some pre-existing components written in traditional languages such as C and Perl, or use proprietary transmission protocols. Users can implement SkyWalking's dissemination protocol by themselves to make sure these services are included in the monitoring system.

This is what makes SkyWalking powerful. Its openness, ease of use, and high customizability are key to the success of its community. At present, in addition to the core Java language agents in the community, SkyWalking supports multiple language agents such as .Net, PHP, Node.js, and GoLang. These agents are all incubated by the community and have wide user bases. Together, these language agents make up a huge sample repository and provide the participants with a stable and fast way to getting started.

You have now been introduced to the advantages of SkyWalking's approach to monitoring microservice architecture and its scope of application. In the next section, we will discuss a real-life use case to study how SkyWalking brings about the above effects.

### 3.3 Construction of the live environment

In this section, we will introduce the key functions of SkyWalking through the illustration of a real-life use case.

#### 3.3.1 SkyWalking background construction

We have introduced in Chapter 2 how to build SkyWalking's background services, namely the OAP and UI services. To make it simpler, we will use the docker-compose method to quickly start a set of SkyWalking back-end service stack in this use case.

First, access <https://github.com/apache/skywalking-docker> with git clone.

```
git clone https://github.com/apache/skywalking-docker
```

As an illustration, release version 6.6 is used here.

```
cd skywalking-docker/6/6.6/compose
```

The purpose of the default docker-compose service is to demonstrate the startup of the backend, so the parameters are set at a lower than required value. Therefore, it is necessary to adjust the compose files. Increase the ES heap and the length of the ready-made queue (default value of the ready-made queue is 200)

```
git diff:
environment:
  - discovery.type=single-node
  - bootstrap.memory_lock=true
-   - "ES_JAVA_OPTS =-Xms512m -Xmx512m"
+   - "ES_JAVA_OPTS=-Xms4096m -Xmx4096m"
+   - thread_pool.write.queue_size=1000
+   - thread_pool.index.queue_size=1000
ulimits:
  memlock:
    soft: -1
```

Now, you can start the background service.

```
docker-compose up -d
```

```
docker-compose logs -f | grep Start
```

When you have the log as shown in Figure 3-3, your startup has been successful.

```
2020-01-31 05:56:13,105 - org.apache.skywalking.oap.server.library.server.grpc.GRPCServer -69014 [main] INFO [] - Bind handler JVMMetricReportServiceHandler into gRPC server 0.0.0.0:11800
2020-01-31 05:56:13,497 - org.apache.skywalking.oap.server.library.server.jetty.JettyServer -69406 [main] INFO [] - start server, host: 0.0.0.0, port: 12800
2020-01-31 05:56:13,506 - org.eclipse.jetty.server.Server -69415 [main] INFO [] - jetty-9.4.2.v20170220
2020-01-31 05:56:13,584 - org.eclipse.jetty.server.handler.ContextHandler -69493 [main] INFO [] - Started o.e.j.s.ServletContextHandler@6bce4140(//,null,AVAILABLE)
2020-01-31 05:56:13,610 - org.eclipse.jetty.server.AbstractConnector -69519 [main] INFO [] - Started ServerConnector@20216016(HTTP/1.1,[http/1.1]){0.0.0.0:12800}
2020-01-31 05:56:13,611 - org.eclipse.jetty.server.Server -69520 [main] INFO [] - Started @69612ms
```

Figure 3-3 Startup log

### 3.3.2 Building the live cluster

We use <https://github.com/SkyAPMTest/skywalking-live-demo> to build the live environment. The steps are as follows:

#### 1) Download the agent release package

Go to the page at <http://skywalking.apache.org/downloads/> and select "Binary Distribution (Linux)" to download.

```
wget https://www-us.apache.org/dist/skywalking/6.6.0/apache-skywalking-apm-6.6.0.tar.gz
```

Unzip the compressed code package and obtain the path of the agent directory.

```
tar xvf apache-skywalking-apm-bin.tar.gz
cd apache-skywalking-apm-bin/agent
export AGENT_HOME='pwd'
```

#### 2) Build the live demo application

Clone the project address. Compile and package based on the documentation.

```
git clone https://github.com/SkywalkingTest/skywalking-live-demo.git
cd skywalking-live-demo
mvn clean package
```

Obtain the code package:

```
live-demo-assembly.tar.gz
```

After decompression, you will have:

```
bin eureka-service kafka_2.11-2.3.0 kafka_2.11-2.3.0.tgz logs projectA projectB projectC
projectD
```

In addition to the microservice project, you will find that the live demo application includes the middleware Kafka.

#### 3) Start the demo application cluster

Run the following script:

```
export AGENT_DIR=${AGENT_HOME}
cd ./live-demo/bin
./startup.sh
```

AGENT\_DIR is the agent directory in the release package, and COLLECTOR\_SERVER\_LIST is the gRPC access address of the OAP service.

Run JPS to observe whether all processes have started successfully:

```
$ jps
15056 projectD.jar
14689 Kafka
14690 eureka-service.jar
15020 projectB.jar
6780 Jps
15038 projectC.jar
14367 QuorumPeerMain
```

The bold part in the following OAP log further shows that the process instance has been successfully registered.

```
oap | 2020-01-31 06:00:44,247 - org.apache.skywalking.oap.server.receiver.
register.provider.handler.v6.grpc.RegisterServiceHandler -340156 [grpcServerPool-1-thread-4] INFO
[] - register service instance id=3 [UUID: ba6e40087c9941c7ad837c60c72057dd]
```

#### 4) Access the demo application cluster

Make a demo request to access projectA. Use curl to access or use the browser directly.

```
http://<projectA-ip>:8764/projectA/test
```

Continue to access the address with scripts or load testing tools. Since the OAP uses the asynchronous registration mode at the time of service registration, you can obtain tracing data only after multiple access.

### 3.4 Live operations

This section will explain in detail the meaning of various monitoring metrics generated by SkyWalking to help you better understand the operating status and health of the monitoring system. It will also draw connections on the dependencies between these metrics to help you better understand the internal operating mechanisms of SkyWalking.

This section will first introduce the various dimensions of monitoring metrics. These dimensions are the built-in concepts of SkyWalking. Understanding them is the basis for mastering the monitoring metrics. Then, the core monitoring functions will be introduced. This includes general viewing of monitoring metrics and using topology diagrams to observe the system architecture. Finally, you will be guided on

how to resolve frequently encountered problems in the production system, including extracting critical paths, troubleshooting failed services or requests, and troubleshooting slow services or requests.

### 3.4.1 Observing the various dimensions of microservices

SkyWalking provides observability capabilities for services, service instances, and endpoints. There is a hierarchical relationship between these three dimensions. Both service instances and endpoints must come under a single service, but there is no necessary connection between service instances and endpoints. Here are the definitions:

- **Service:** Represents a series or set of workloads that provide the same behavior to requests. When using service agents or SDK, you can define the name of the service. Where it is undefined, SkyWalking will use the name defined by the user on the platform, such as Istio.
- **Service instance:** Each workload in the above set of workloads is called an instance. Just like Pod in Kubernetes, a service instance is not necessarily a process on the operating system. But when a service agent is used, a service instance is essentially a real process on the operating system.
- **Endpoint:** A request path received for a particular service, such as the URI path of HTTP and the class service name and method signature for the gRPC class.

Let's look at a specific example from the UI. As shown in Figure 3-4, projectA is the service, /projectA/{name} is the endpoint, and {name} indicates that the endpoint is a URL with path parameters. projectA-pid:23536@remote-worker-hk is the service instance, where remote-worker-hk is the hostname of the machine, and 23536 is the process number of the instance.

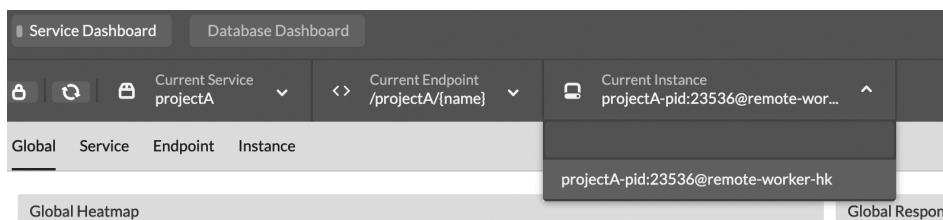


Figure 3-4 Monitoring dimensions

All the metadata in this live environment is as shown in Figure 3-1.

Table 3-1 Dimensional metadata

Service	Service instance	End point
projectA	projectA-pid:23536@remote-worker-hk	/projectA/{name}

projectB	projectB-pid:23002@remote-worker-hk	/projectB/{value}
projectC	projectC-pid:23191@remote-worker-hk	/projectC/{value}
projectD	projectD-pid:23370@remote-worker-hk	Kafka/test-trace-topic/Consumer/test

### 3.4.2 Observability metrics

The core function of the monitoring system is to observe the monitoring metrics. The UI of SkyWalking provides an overview function for various metrics. This function can display multiple monitoring metrics in a single dimension. This interface can be used not only as a query function, but also as a Network Operations Center (NOC) Dashboard of the monitoring system. Turn on the automatic refresh function in the upper right corner of Figure 3-5, and the metrics will be updated according to the refresh frequency.

Figure 3-5 shows the dashboard for service monitoring metrics, such as the number of services, average response time, throughput, service levels, response time percentile, service dependency map, and TopN records. The UI also provides a Database Dashboard for displaying database-related metrics, as shown in Figure 3-6. In addition to the same metrics as the Service Dashboard, slow SQL queries is a very useful feature as it can help users quickly troubleshoot database access problems. In the future, more middleware metrics (including queues and caches) will be displayed.

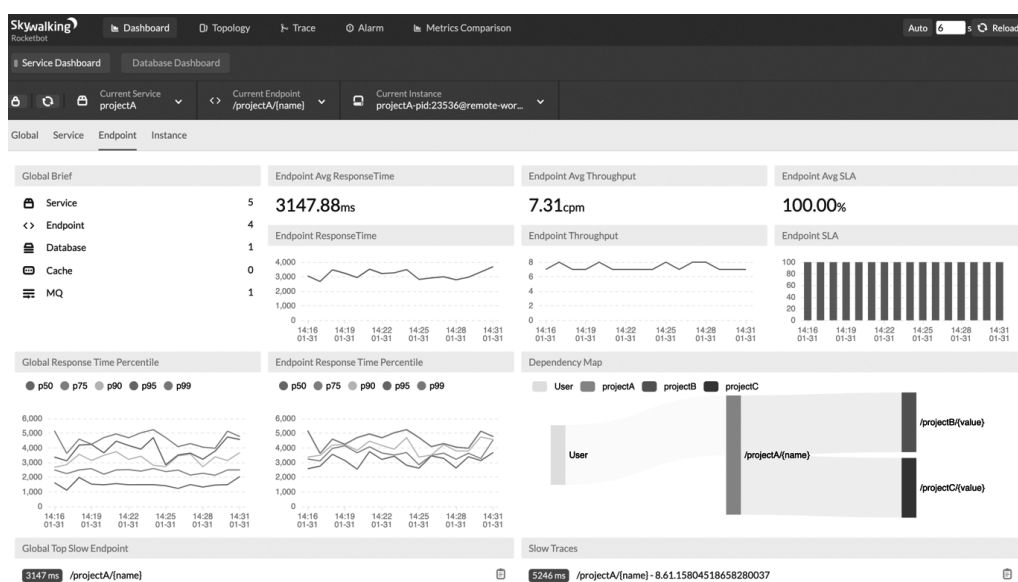


Figure 3-5 Service Dashboard





Figure 3-6 Database Dashboard

It is possible to customize the metrics dashboard, as shown in Figure 3-7. Users can customize information such as the types of metrics, their positions on the page, and their width and height, so as to create a screen in line with their needs, or to customize multiple sets of screens for different purposes.

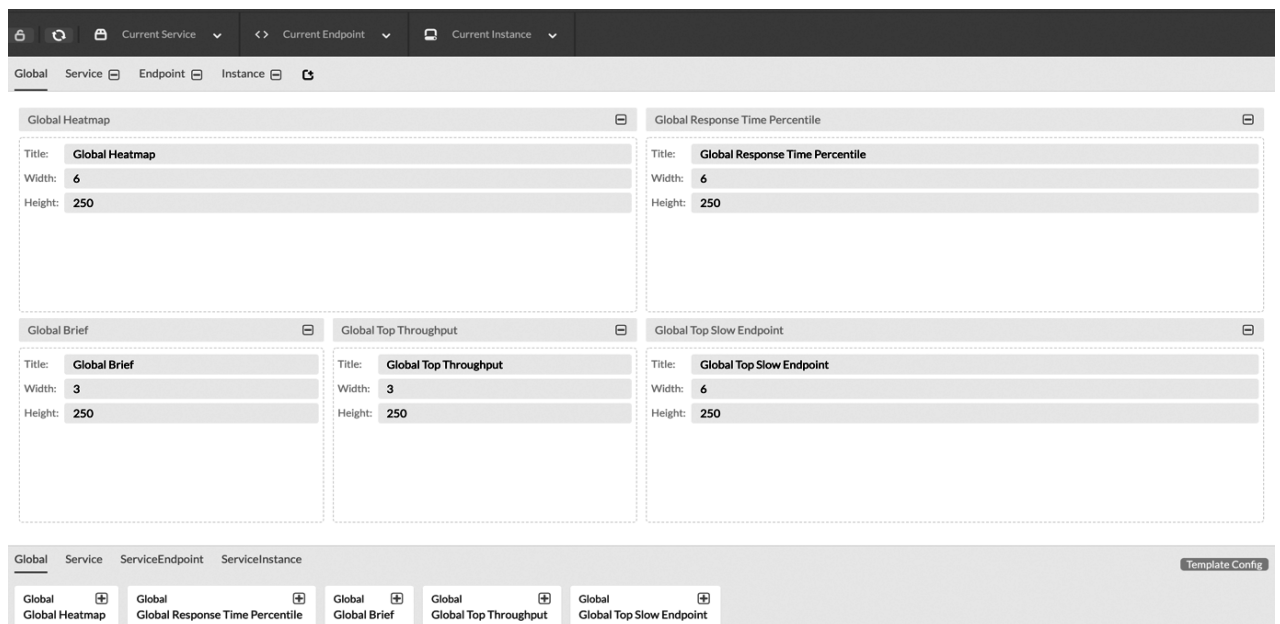


Figure 3-7 Customizing the dashboard

### 3.4.3 Observability system architecture

Having built the live environment successfully, you can get to know the system architecture of the environment through the UI topology diagrams. Learning about the architecture of the topology diagram will help you better understand the live operations to be introduced in the subsequent sections. Note that topology diagrams are limited to topology of services, and do not include service instances and endpoints.

Figure 3-8 shows the topological relationship in the live environment. The user accesses the cluster through the projectA service, which is a Spring MVC service. projectA accesses two Spring MVC services, namely projectB and projectC. projectB accesses the local H2 database. projectC visits [www.baidu.com](http://www.baidu.com) and writes data to a Kafka message queue at the same time. projectD consumes data from this Kafka. projectA, projectB, and projectC access the 8761 service at the same time. As seen from the installation process, this is the Eureka service.

This environment is a set of standard microservice architecture, which also contains various external services. The user can click through to details in the UI: e.g., you could click on projectA, as shown in Figure 3-8, for more information. As shown in Figure 3-9, you can quickly connect to other monitoring metrics pages related to the services through the pop-up hexagonal menu. At the same time, the services directly associated with projectA are highlighted to make it easier for users to see the dependencies between the services.

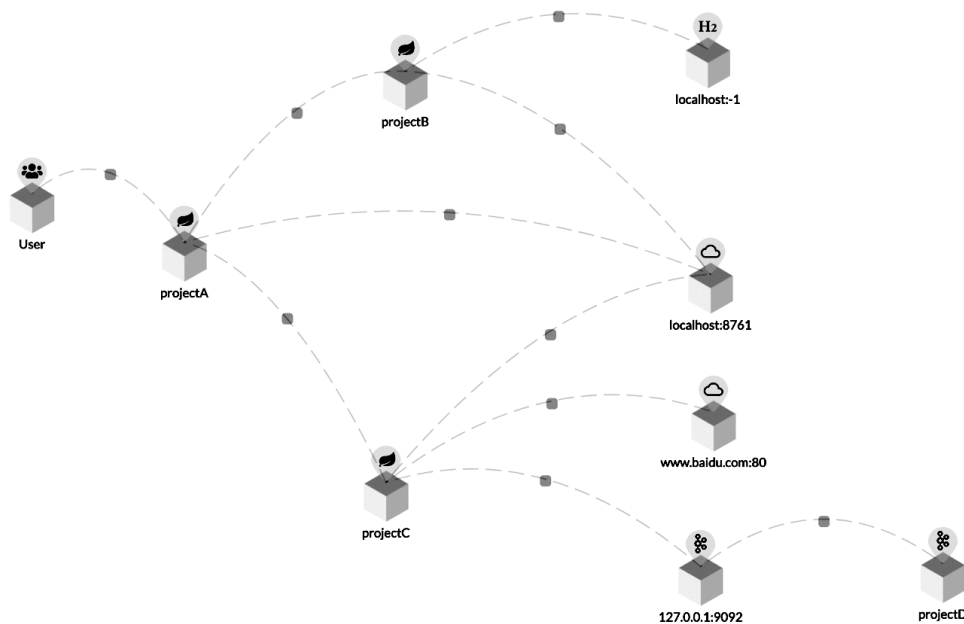


Figure 3-8 Microservice topology diagram

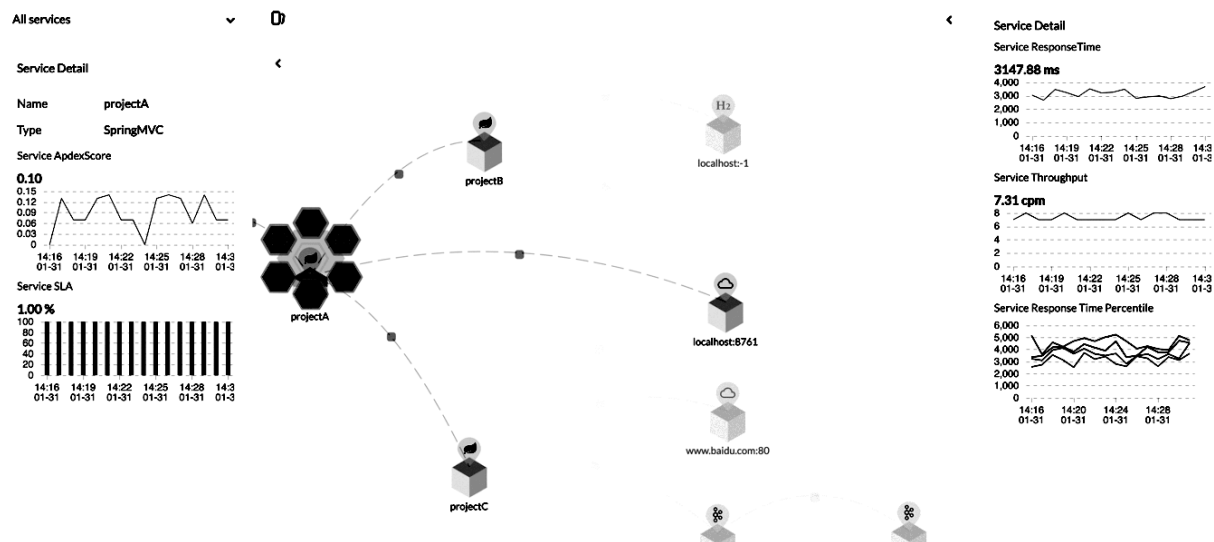


Figure 3-9 Service details

### A. Generation of User Points

You may already have noticed that a User Point is not an actual entity. It has not been configured during the installation process, so how did it come about? To learn more within the interface, you can click the midpoint of connection between User and projectA, as shown in Figure 3-10.

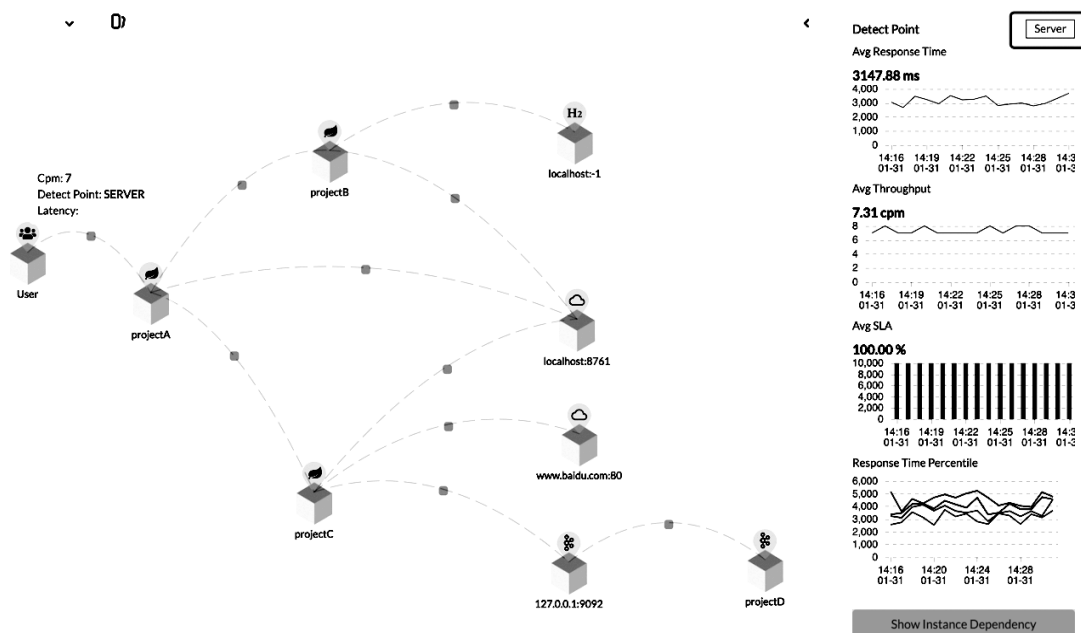


Figure 3-10 Service call details

As you can see, the right side of Figure 3-10 shows the relevant metrics for the calling of projectA by User. The "Detect Point" in the upper right corner is set to be Server, which means that this call relationship is generated by the Server side. In other words, it is generated by projectA. Therefore, all monitoring metrics as seen on the right are sent by projectA to the OAP service. So how did User come about?

Without doubt, it is generated by the OAP. The rule for the generation is as follows: if the call data only comes from the upstream service and not the downstream service, then a point is generated on the topology map to indicate that there are users accessing the upstream service.

Although this map represents a specific manual operation, it is likely generated by program calls or other services that have not been detected by SkyWalking. Therefore, it is more accurate to understand the User as the entry point to the monitoring target system. The live environment as seen here uses a script to continuously generate traffic without being accessed by actual users.

### B. Access to external services

In addition to the Server, there is also a Client in the Detect Point. Figure 3-11 shows a typical form of Client. It means that all monitoring data comes from projectB. This access method is mostly used for access to external services, including data repositories, API services, and middlewares.

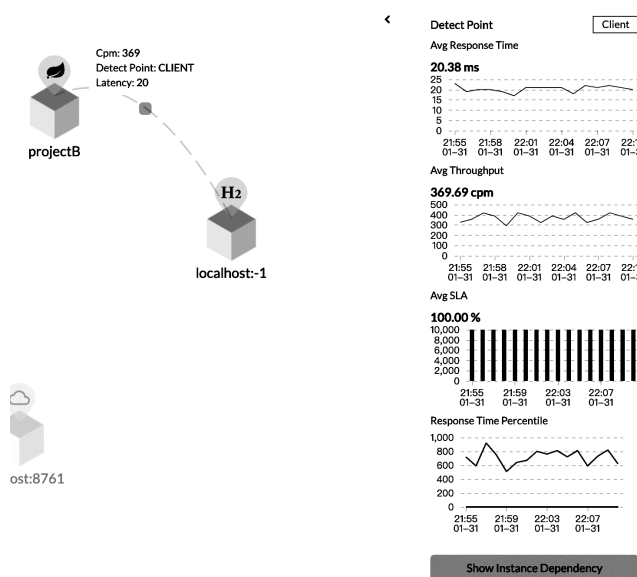


Figure 3-11 Client call details

Similar to the User, the points of these external services are also generated by the OAP for the same reason: there is no way to install SkyWalking monitoring on these external services. But unlike the User, these external services can be identified by accessing the Client driver of the service. As shown in Figure 3-11, this is an H2 database.

SkyWalking's Agent has a large number of plug-ins for identifying these external services. Major middlewares and databases are currently being covered.

### C. Direct access to monitored services

What if the two services monitored by SkyWalking are being called directly? The answer is that the Detect Point will generate two sets of server and client data, as shown in Figure 3-12 and Figure 3-13.

You can see that most of the metrics here are similar, with the exception of the response time. For response time, the latency rate on the client end is higher. The reason there is a higher value on the client side than on the server end is that the response time for the client end also includes the network delay.

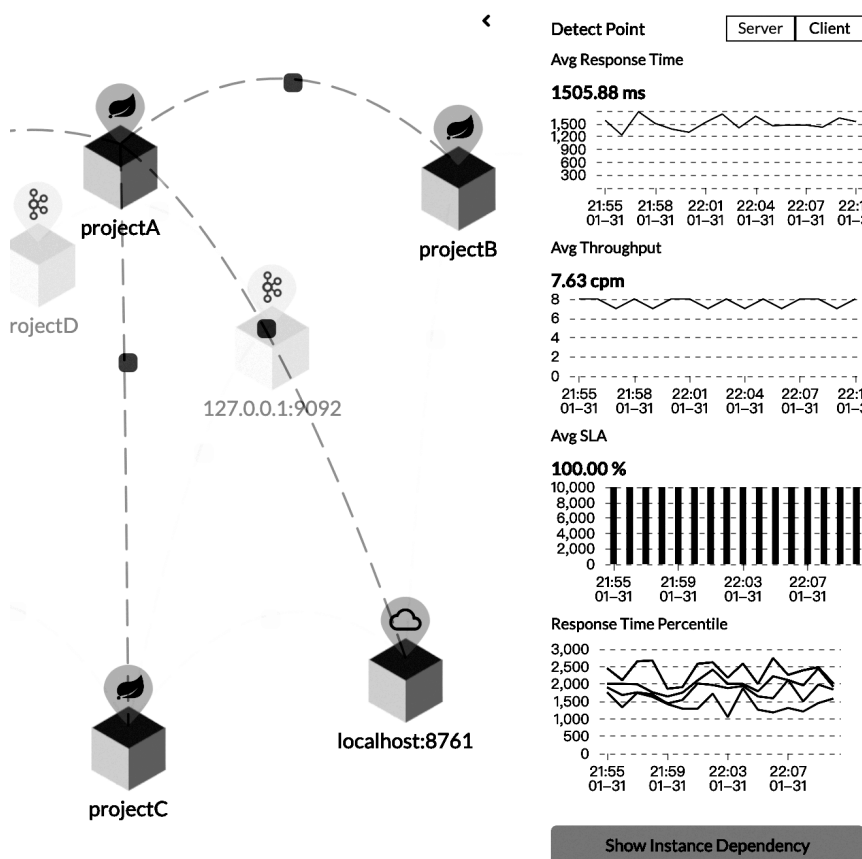


Figure 3-12 Client call details

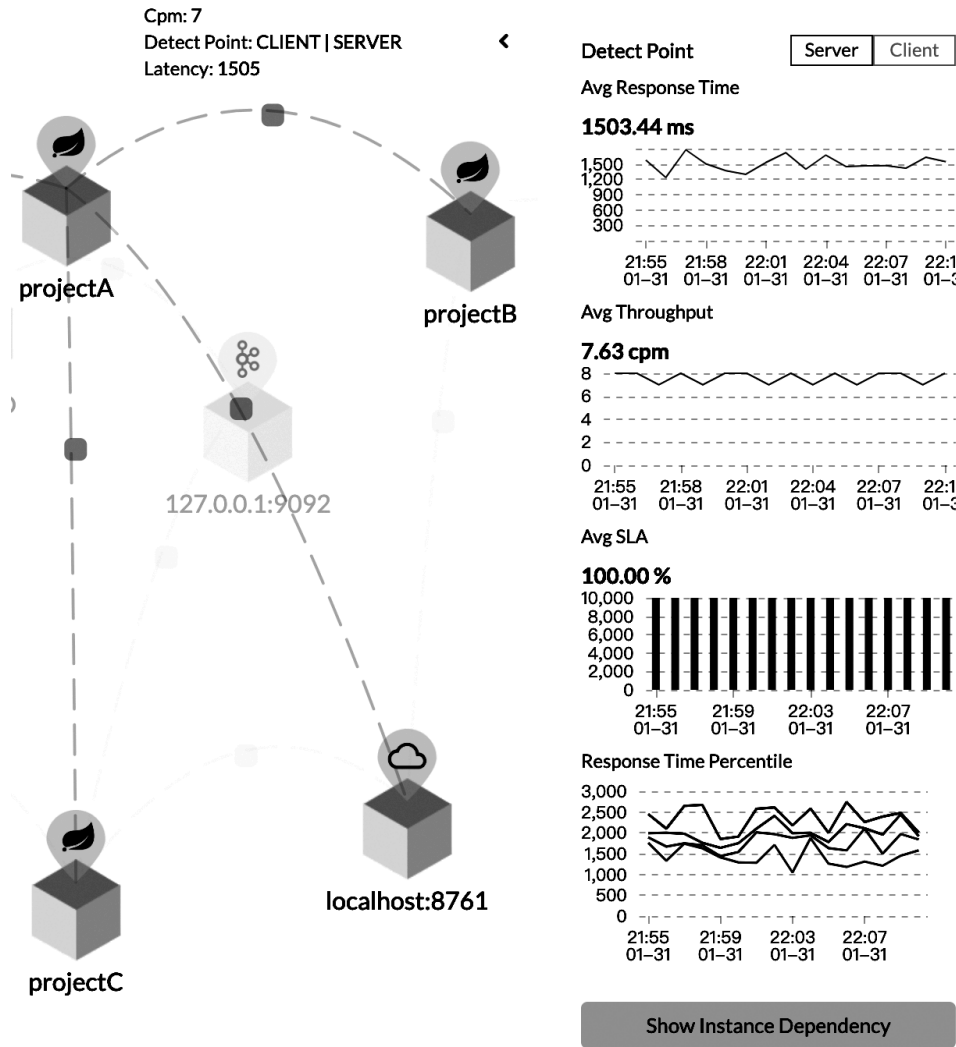


Figure 3-13 Server call details

Assuming that some users don't mind the delay, they may ask whether it is possible to have only one segment of data, in order to reduce the amount of data being written. In fact, the difference will be larger than being imagined. Say we place a hypothetical proxy between projectA and projectB. Assuming this proxy will filter out 10% of the traffic (returning it to projectA 403), and projectA has sent 100 requests, these will be the results:

- The range of average response time on the client end will be reduced, since 10% of bad requests will be faster than normal requests;
- The throughput of the client end will be more than that of the server end, since the server end only receives 90% of the requests;
- The SLA on the client end is only 90%, while that on the server end is 100% (90% of the traffic being received was successful).

This virtual proxy can be a real load balancer, a sidecar in a service mesh, or even a packet loss issue in your vulnerable network environment, so this is a scenario with a wide range of applications.

Based on the above principles, it is not recommended to retain data on only one end, as this often causes difficulty in troubleshooting. Where there is only a single end of data, there will be many limitations in place, making it impossible to effectively collect monitoring data to a sufficient level.

### 3.4.4 Extracting a critical path

A critical path is a graph formed by related service traces that directly affect the performance of the system. The critical path is likely not just a line, and could come in the form of a graph.

Since the system is dynamically changing, SkyWalking's critical path analysis is all related to time. Such analysis could be divided into two types: path analysis for a single request and path analysis for a period of time. They use the following tools respectively:

#### *A. Query with tracing*

Tracing, or distributed tracing, is an important means for debugging and monitoring application performance, especially for applications under the microservice architecture. The benefits of distributed tracing have already been discussed in Section 3.2. In short, tracing data helps identify defects and errors in the system.

Let's take a look at the mostly commonly used trace data in SkyWalking. Figure 3-14 shows the list of trace data. In this demonstration, the segment in the trace is taken as the unit. In the middle of the figure is the entire trace, which includes the first trace segment.

The trace includes five trace segments, namely loadbalancer1, loadbalancer2, projectA, projectB, and projectC. These five segments come from these five services, as represented by different colors.

Each segment comprises multiple dots, and each dot represents a span, which is the smallest unit of a trace. A segment is at a higher level than a span.

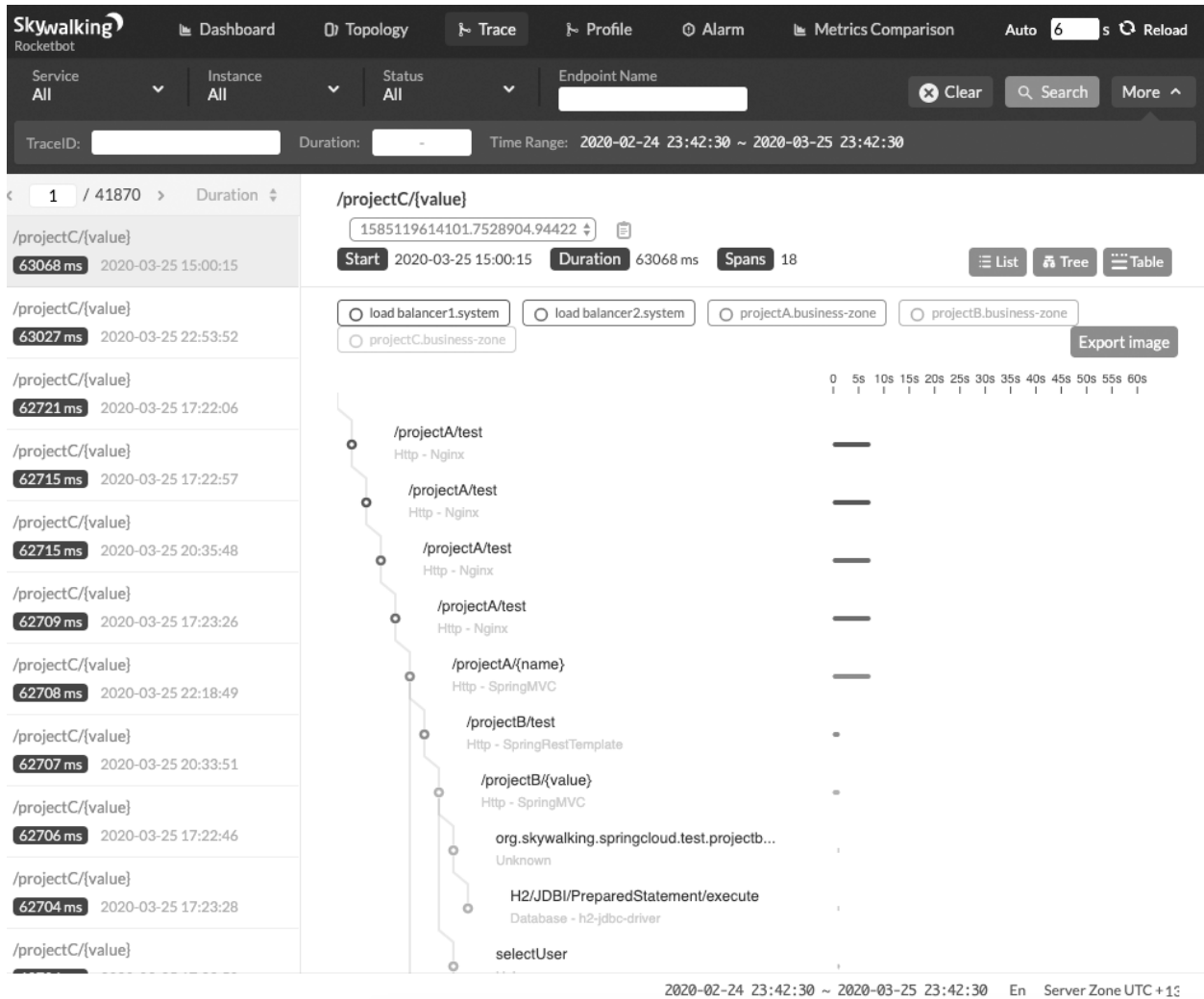


Figure 3-14 Trace data list

Segments are rare in general tracing platforms, but they are key to the high performance of SkyWalking. Generally, all spans generated in a service instance will be merged into one segment, and then sent out in batches. This not only ensures a higher transmission performance, but also reduces the pressure on calculation and storage, thereby achieving the goal of high throughput.

The rightmost side of Figure 3-14 is the latency bar graph. In this bar graph, you can easily identify the part with high latency, and at the same time gain an overall impression of the latency of each span in the trace. Therefore, the list view is very helpful for observing the latency.

Based on these fundamentals of tracing, let's use the table view as shown in Figure 3-15 to locate the critical path. First, we determine the critical path based on the latency.



We can see that there are two entry points to the request: projectA and Kafka's consumer. As Kafka consumption will not affect user experience, let's focus on the entry request for projectA.

/projectA/{name}

8.61.15804795714751079

Start 2020-01-31 22:06:11 Duration 5248 ms Spans 13

List Tree Table

Method	Start Time	Gap(...)	Exec(ms)	Exec(%)	Self(m...)	API	Service
▼ /projectA/{name}	2020-01-31 22...	0	5248	<div></div>	2	SpringMVC	projectA
▼ /projectB/test	2020-01-31 22...	0	2746	<div></div>	3	SpringRestTem...	projectA
▼ /projectB/{value}	2020-01-31 22...	0	2743	<div></div>	1001	SpringMVC	projectB
> org.skywalking.springcloud.test.projectb.dao.DatabaseOperateDao.saveUse	2020-01-31 22...	0	968	<div></div>	0	-	projectB
> selectUser	2020-01-31 22...	0	774	<div></div>	0	-	projectB
▼ /projectC/{name}	2020-01-31 22...	0	2500	<div></div>	1	Feign	projectA
▼ /projectC/{value}	2020-01-31 22...	0	2499	<div></div>	2012	SpringMVC	projectC
/	2020-01-31 22...	0	484	<div></div>	484	HttpClient	projectC
Kafka/test-trace-topic/Producer	2020-01-31 22...	0	3	<div></div>	3	kafka-producer	projectC
▼ org.skywalking.springcloud.test.projectd.MessageConsumer.consumer()	2020-01-31 22...	0	2	<div></div>	1	-	projectD
Kafka/test-trace-topic/Consumer/test	2020-01-31 22...	0	1	<div></div>	1	kafka-consumer	projectD

Figure 3-15 Table view

projectA has accessed projectB and projectC respectively. From the Exec(%) column, we can see that these two requests accounted for about half of the total latency of projectA, so both projectB and projectC are points on the critical path.

Based on the same method, let's go on and analyze projectB. It is found that the two database calls account for about 60% of the delay, so they are also on the critical path.

The write latency of Kafka in projectC is very low and can be excluded from the critical path. Access to baidu.com has to be included in the critical path.

So far, we have only analyzed from the standpoint of latency. If we also consider the concept of error rate, the Kafka write requires further analysis. If the exception generated by the write is directly passed to the outermost layer, then the write is a critical operation; otherwise, it can be completely excluded from the critical path.

### B. Using topology diagram

If you would like to observe the critical path over a period of time, the topology map is an excellent tool. Figures 3-16 and 3-17 show the call details of projectA → projectB and projectA → projectC, respectively. We can use  $\text{Avg Latency} = \text{Avg Response Time} \times \text{Avg Throughput}$  to calculate the

traffic distribution of the request. From the data in the figures, we can see that projectB and projectC equally account for the response time of projectA.

Similar to a single analysis, after analyzing some non-critical paths, you can artificially inject some errors to reduce their SLA, so as to observe its response to the entrance SLA. When the SLA is less than 100%, the icon will change color, and the extent of the impact can be observed through the view shown in Figure 3-18.

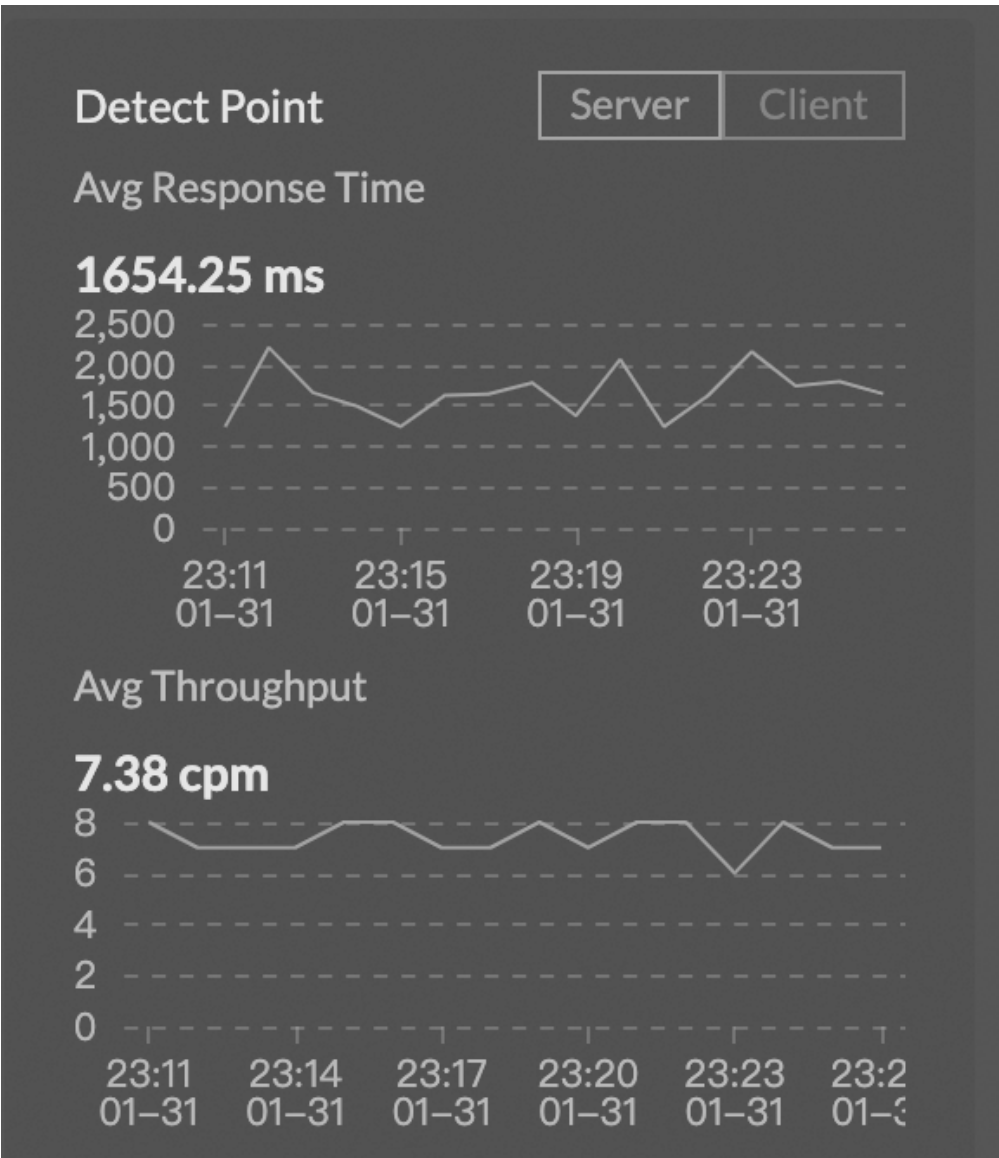


Figure 3-16 projectA calls projectB

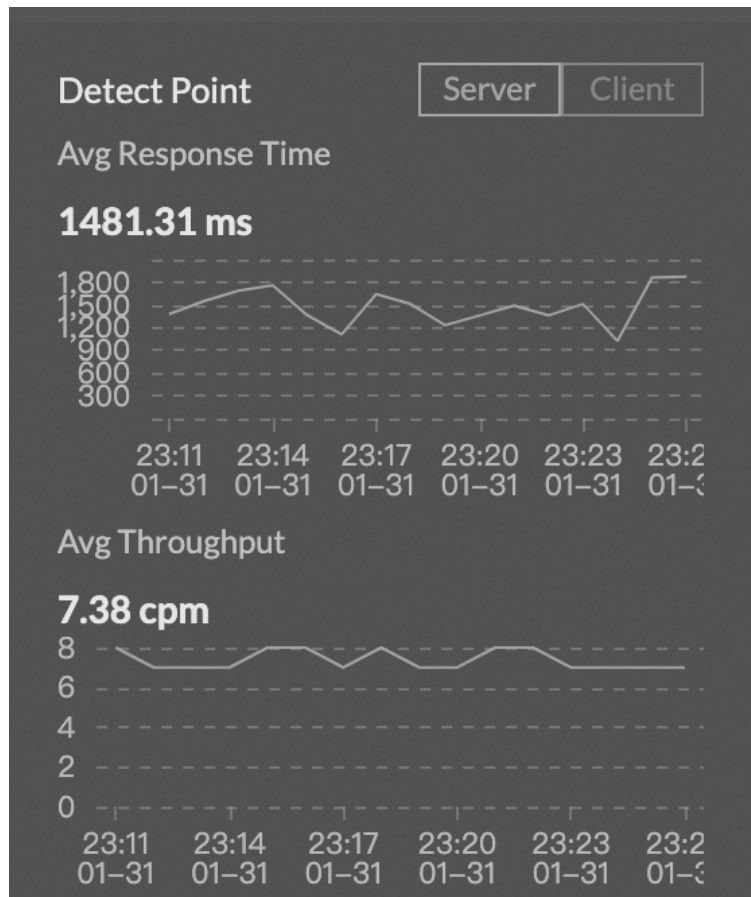


Figure 3-17 projectA calls projectC

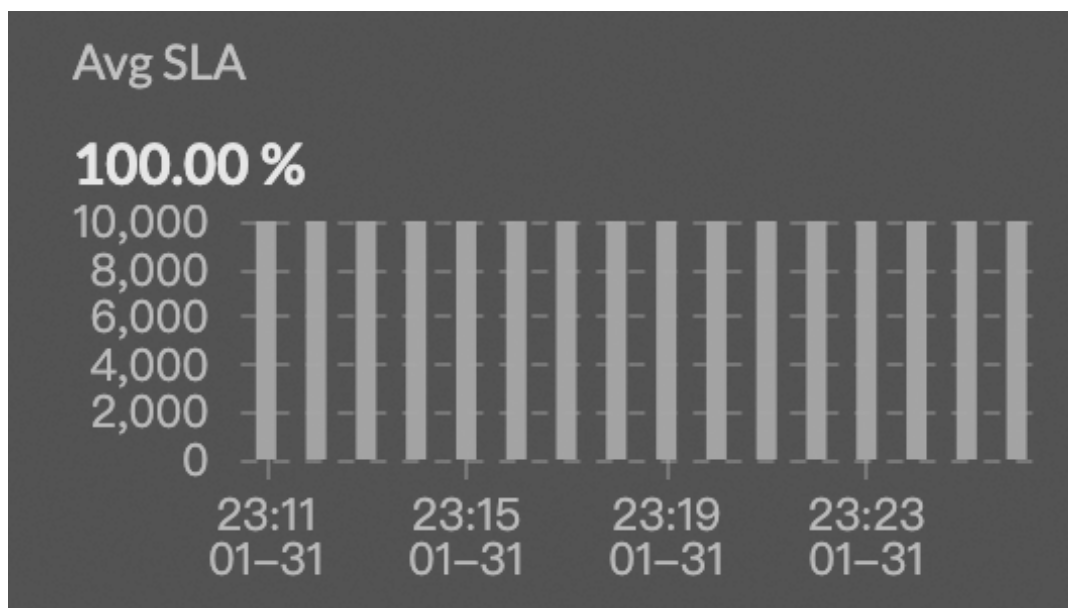


Figure 3-18 Service level bar chart

### 3.4.5 Looking up failed services or requests

By now, you should have gained a general understanding of the live environment and the critical path. In this section, we will use SkyWalking to look up failed services or requests.

From our observation, there are currently no failed nodes in this cluster, so let's manually introduce some errors into the system. Here, we will be using projectB to perform the relevant operations.

```
--- a/projectB/src/main/java/org/skywalking/springcloud/test/projectb/service/ ServiceController.java
+++ b/projectB/src/main/java/org/skywalking/springcloud/test/projectb/service/ ServiceController.java
@@ -18,6 +18,9 @@ public class ServiceController {
    public String home(@PathVariable("value") String value) throws InterruptedException {
        Thread.sleep(new Random().nextInt(2) * 1000);
        operateDao.saveUser(value);
+   if (new Random().nextInt(3) == 0) {
+       throw new RuntimeException("Select user error");
+   }
    operateDao.selectUser(value);
    return value + "-" + UUID.randomUUID().toString();
    }
```

Now that we have introduced an error, the most obvious change is as shown in Figure 3-19.

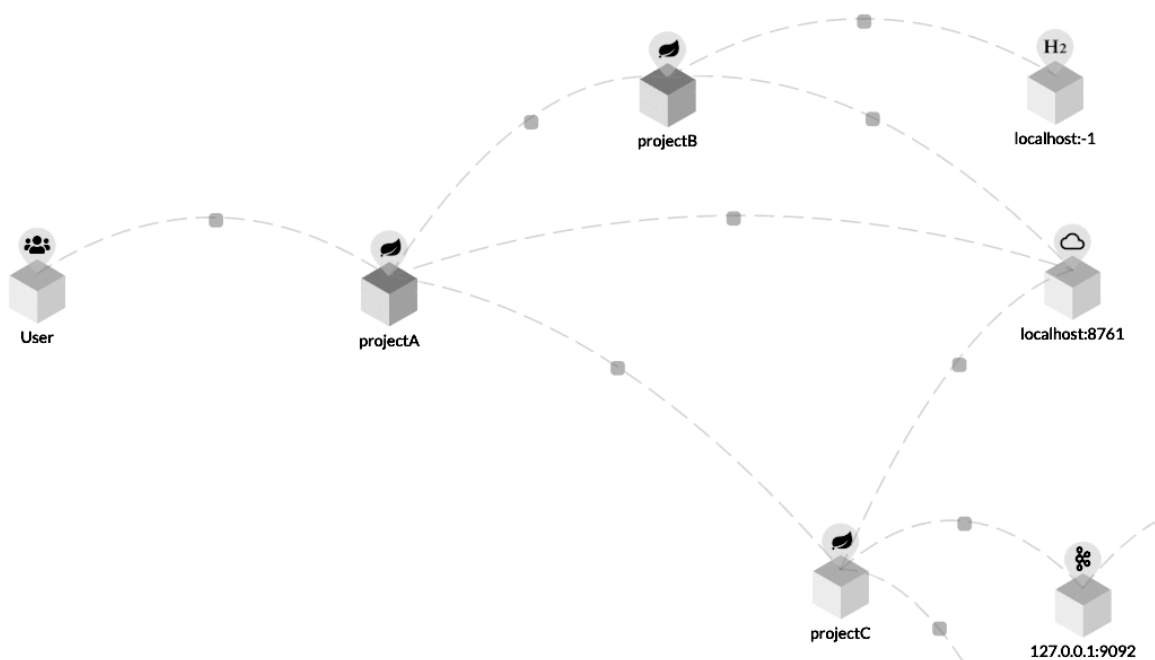


Figure 3-19 Service topology diagram

It can be seen from the figure that not only projectB has changed color, but projectA has also. Now, let's take a look at their SLA values.

From Figures 3-20 and 3-21, we can see that their SLAs are basically the same. It can be further confirmed that the errors of projectA itself result from projectB.

Not only can the topology map and monitoring metrics be checked for errors, we can also obtain error notifications using the alarm function as shown in Figure 3-22.

If you have configured the Webhook for alarm, you will receive the relevant alarm notification when an error occurs.

Then, we go on to ask what exactly has caused the error. The trace query function comes into play.

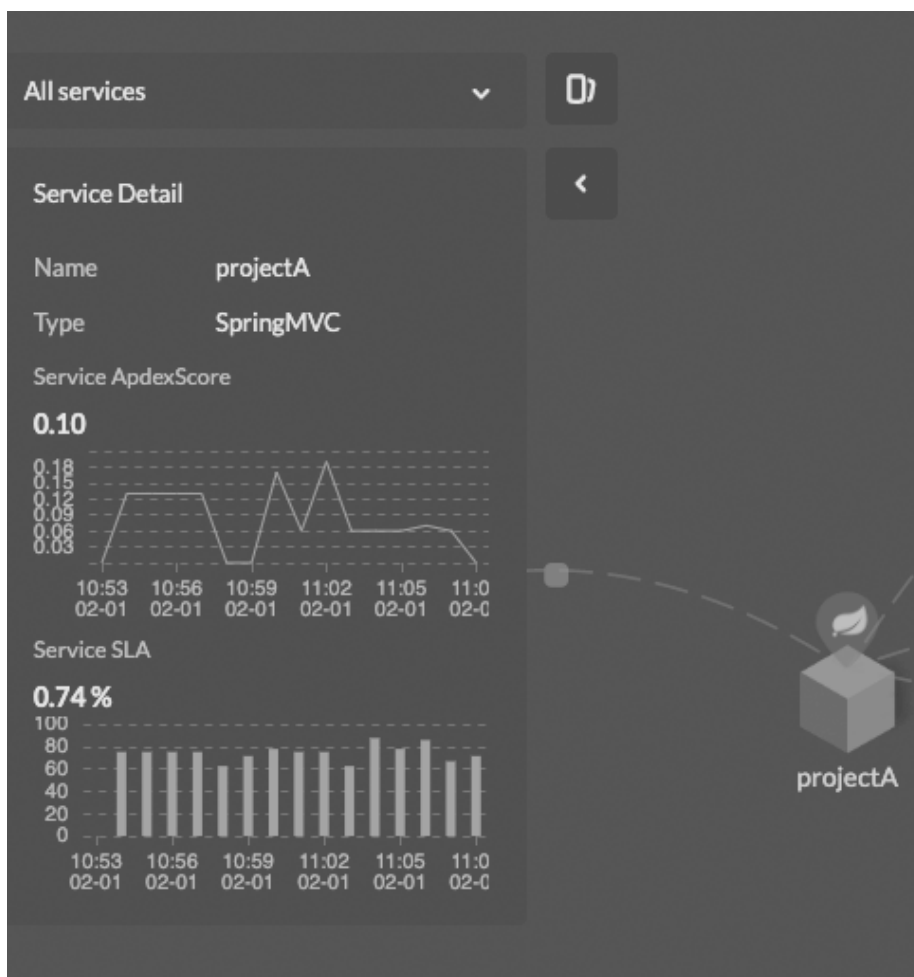


Figure 3-20 projectA service details

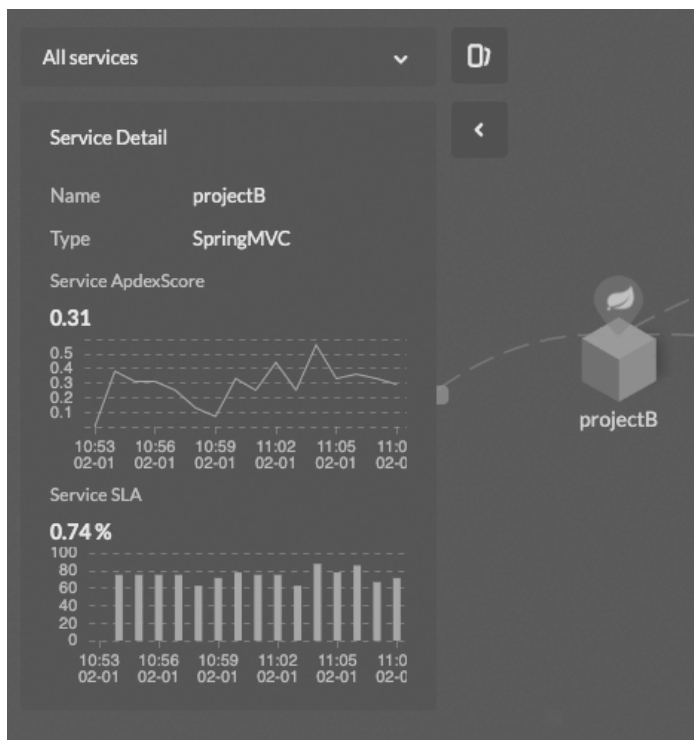


Figure 3-21 projectB service details

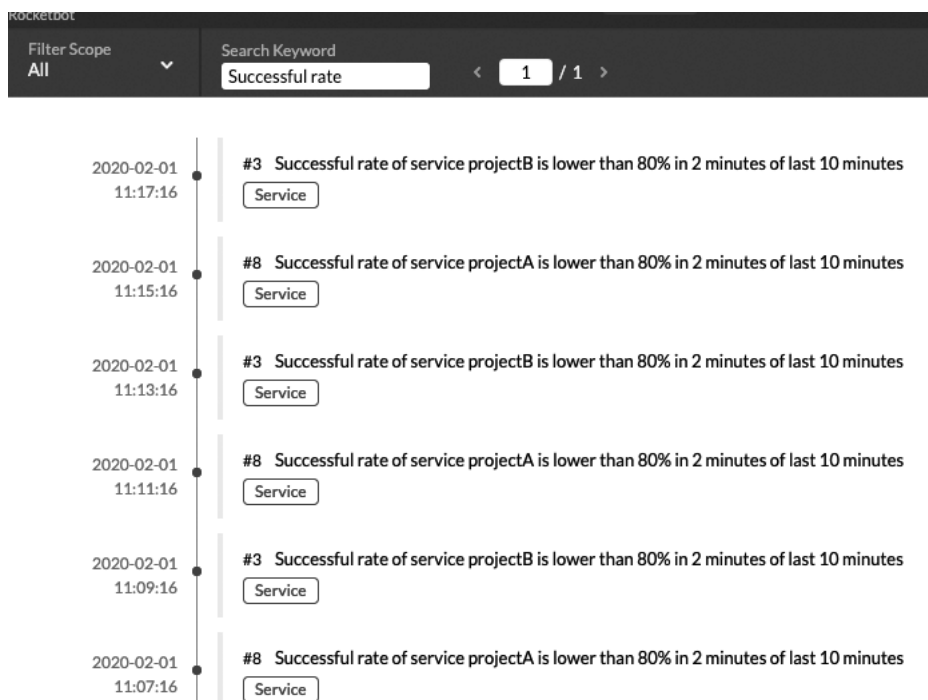


Figure 3-22 Service alarm report

Select the trace data whose status is Error, as shown in Figure 3-23. We can see from the trace list that the main or root cause for the request failure is an error reported by projectB. We can then click projectB to view the specific reason for the error, as shown in Figure 3-24.

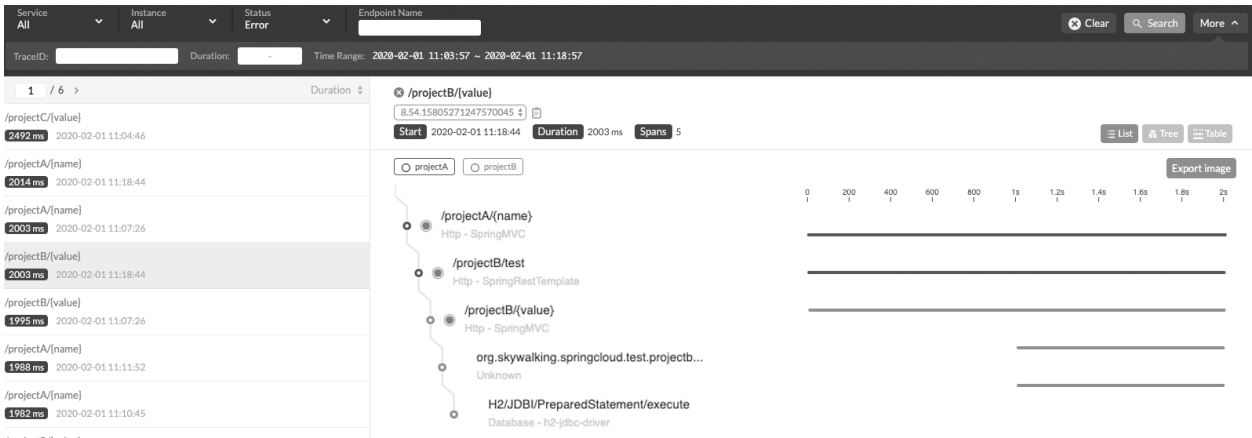


Figure 3-23 Trace data for Error status query



Figure 3-24 Error details

You will find the error being introduced from the stack information and error message. So far, we have used a variety of tools to analyze the reasons for the failure of the request and service.

### 3.4.6 Looking up a slow service or request

Having resolved the error problem, let's focus on the problem caused by latency. The main reason for system latency is that there are slow services or occasional slow requests within the microservice cluster.

Before looking up the slow services, we need to understand that "slow" is a relative concept. It can be explained in the following aspects:

#### A. Dimensions

The definition of "slow" varies in terms of different dimensions. A service may contain multiple endpoints, and each endpoint has different response requirements, resulting in very different overall response requirements for the entire service.

#### B. Time

Different services may have different response requirements in a day, a week, or a month, which is often determined by business needs.

Another specific issue relating to time is the time frame. Some services emphasize better overall throughput, so it is sufficient to have a fast average response time within a broader range (for example, within six hours). In contrast, some services require immediate response capabilities of the system, so it is crucial for the system to offer a perfect response performance in each small time slice.

#### C. Algorithm

There are many ways to calculate latency, including using the instantaneous value, average value, contour value, line graph, and heat map. Each algorithm has different meanings, and has its pros and cons. SkyWalking recommends using the response time percentage graph shown in Figure 3-25 to observe latency. This graph uses 50%, 75%, 90%, 95%, and 99% request response lines (on the left side of the figure, the lines are arranged from bottom to top). This is an excellent way to measure the overall impact of latency and to give users a fuller grasp of the current system latency.

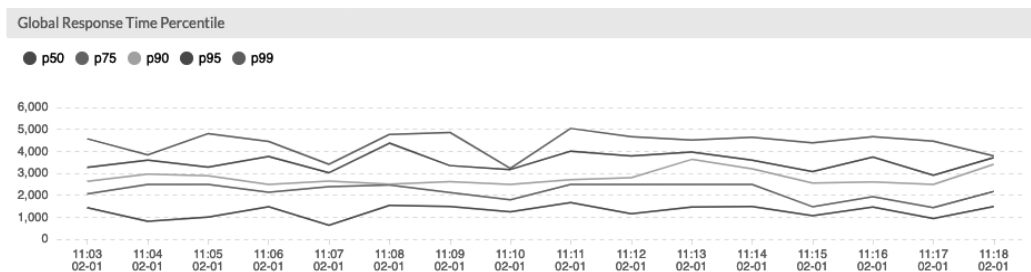


Figure 3-25 Response time percentage chart



#### D. Experience

Finally, the perception of latency varies for different individuals, so it is difficult to quantify the degree of "slowness" by using traditional methods. Currently, SkyWalking only supports the calculation of Apdex scores. As for the level of satisfaction represented by the scores, users may refer to the standards in the link, or set their own scoring criteria as appropriate.

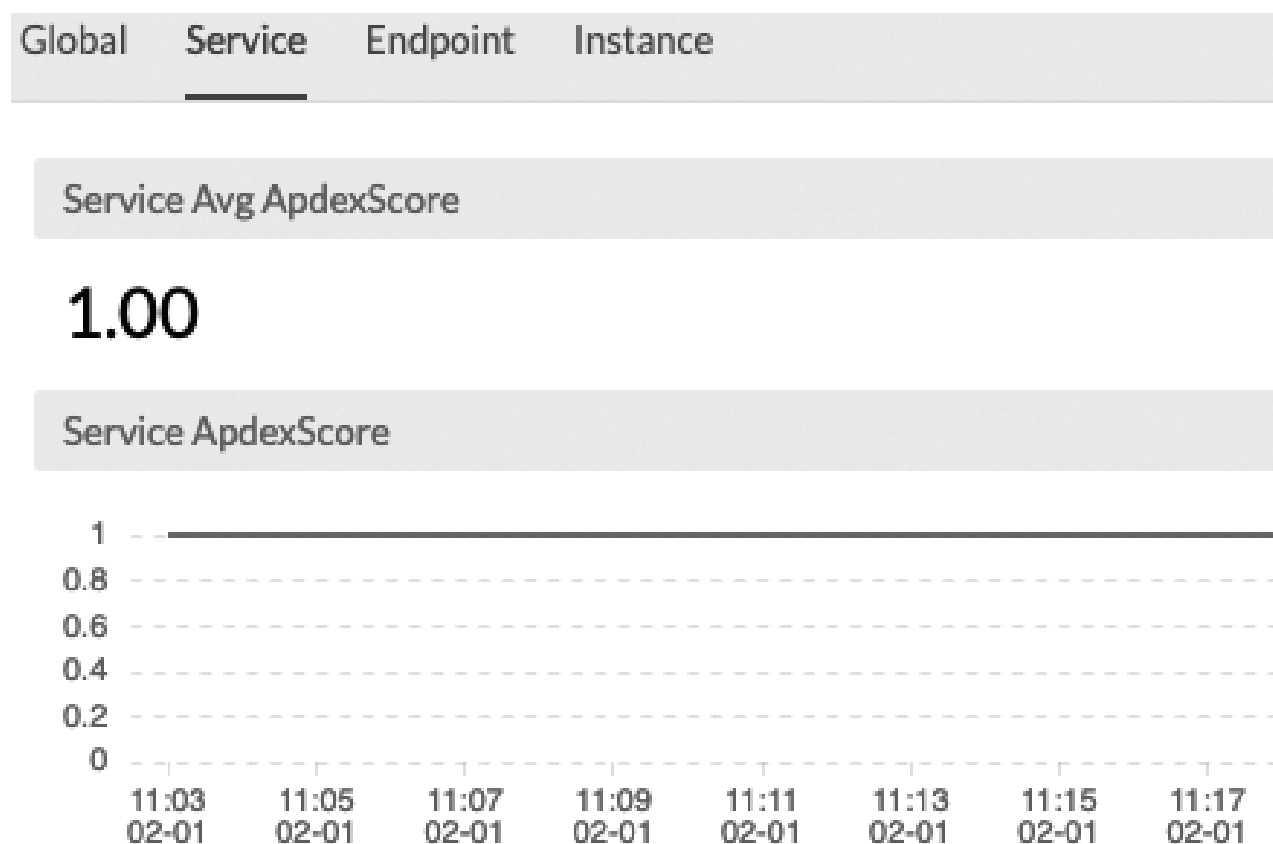


Figure 3-26 Apdex metrics

After determining the standard on latency, we can find out the root cause of the latency through the topology diagram. Here, let's take projectC in Figure 3-27 as an example.

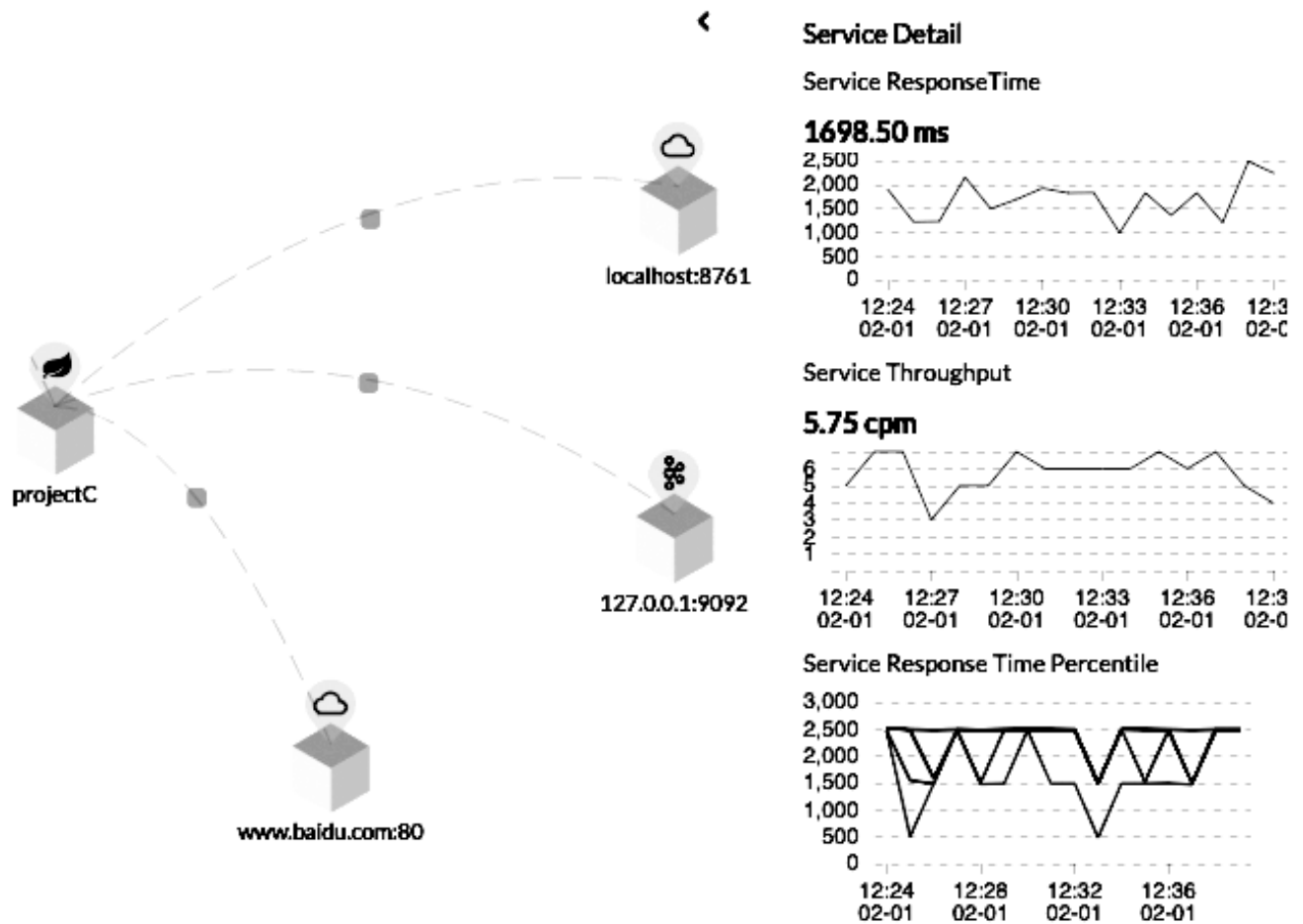


Figure 3-27 projectC service details

The average latency is around 1.7s. There are two external services dependent on it. Let's look at the latency of each service, as shown in Figure 3-28 and Figure 3-29 respectively.

These two external services took around 0.5s in total, and projectC itself spent around 1.2s so the main latency arose from the internal logic of projectC.

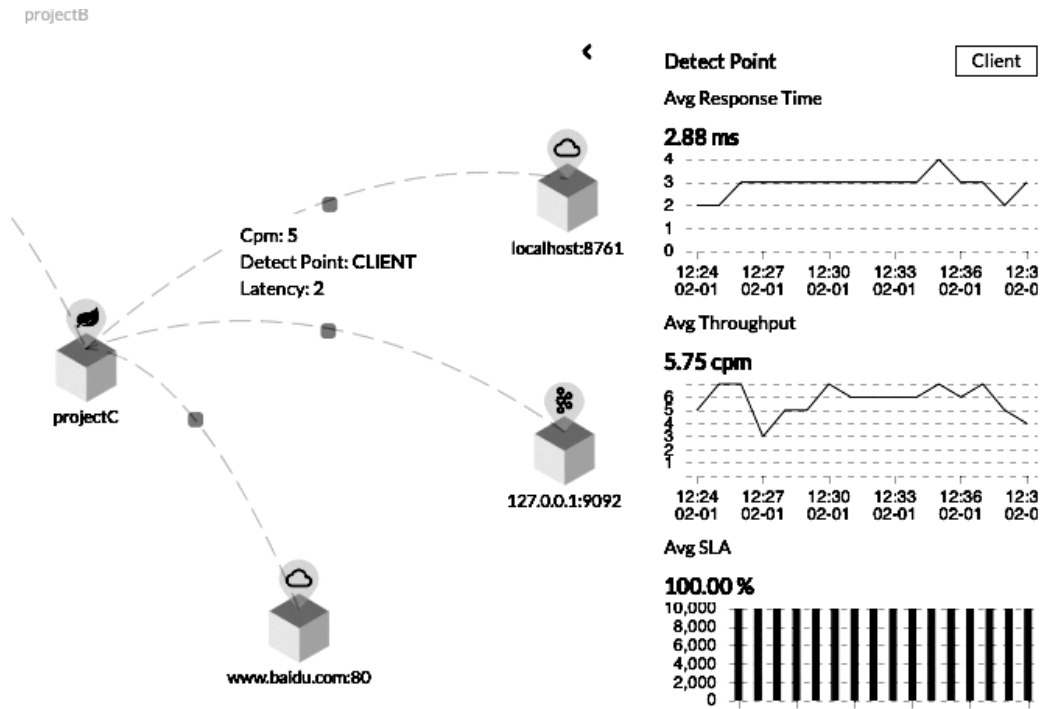


Figure 3-28 projectC calls the external service 127.0.0.1:9092

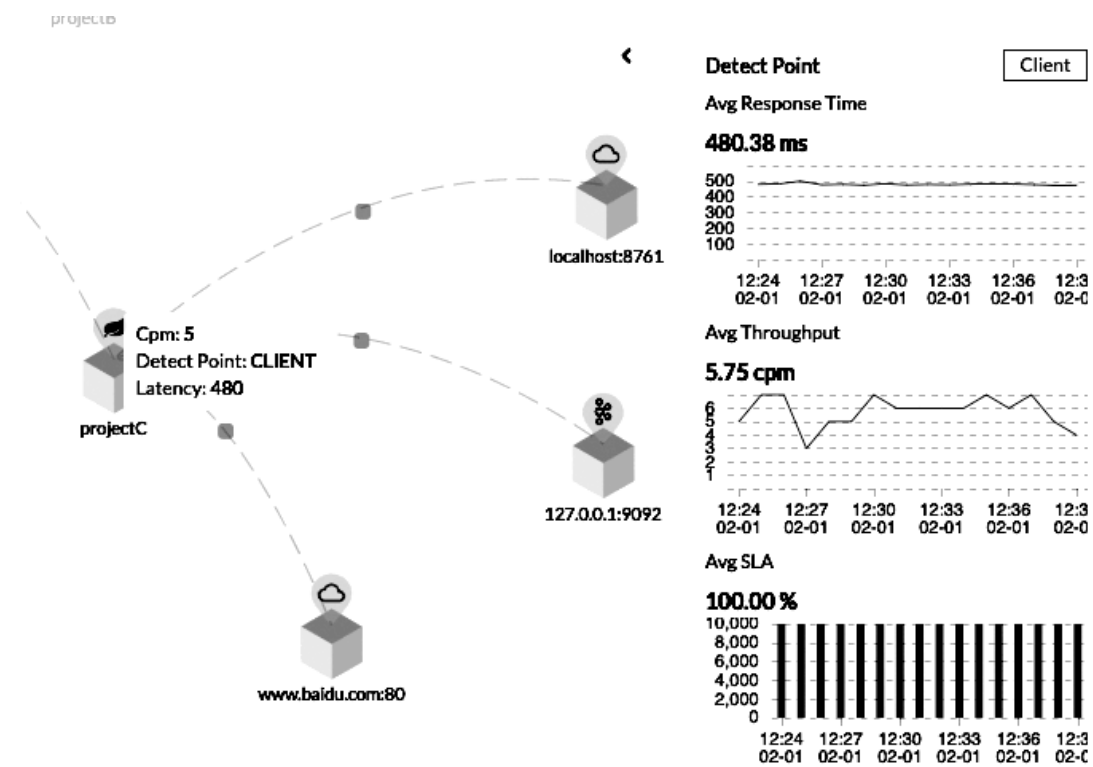


Figure 3-29 projectC calls the external service www.baidu.com

We can confirm the above reasoning based on the trace data. As shown in Figure 3-30, the latency of external services is relatively small.

Let's take a look at the code of projectC to further locate the problem.

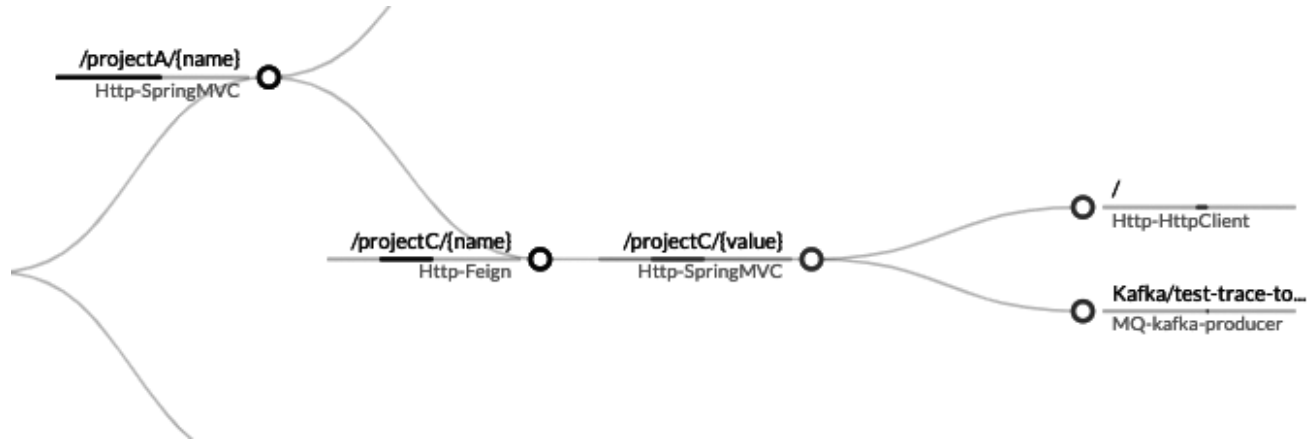


Figure 3-30 Trace data from projectC's call on external services

```
@RequestMapping("/projectC/{value}")
public String home(@PathVariable("value") String value) throws
    InterruptedException, IOException {
    Thread.sleep(new Random().nextInt(3) * 1000);
    httpClientCaller.call("http://www.baidu.com");

    Producer<String, String> producer = new KafkaProducer<>(
        producerProperties);
    ProducerRecord<String, String> record = new ProducerRecord<String, String>(topicName,
        Integer.toString(1), Integer.toString(1));
    record.headers().add("TEST", "TEST".getBytes());
    producer.send(record);
    producer.close();

    return value + "-" + UUID.randomUUID().toString();
}
```

On the third line, there is a function to randomly increase the latency. We can see this is where the slow service is generated.

Another commonly found slow request comes from slow SQL. As shown in Figure 3-31, the metrics page of the Database lists the major slow SQL queries, allowing users to quickly solve them.






Database TopN Records		
996 ms	INSERT INTO user(name) VALUES(?)	
992 ms	SELECT * FROM user WHERE name =?	
987 ms	INSERT INTO user(name) VALUES(?)	
986 ms	INSERT INTO user(name) VALUES(?)	
983 ms	INSERT INTO user(name) VALUES(?)	

Figure 3-31 Slow SQL list

### 3.4.7 Alarm settings

Finally, let's discuss alarms in the live environment. We must first understand how these alarm messages are generated. Let's start with the introduction of the service-level objective (SLO).

SLO is developed from Google's SRE system. Before going into the details of SLO, the average user should already have had a basic understanding of the service-level agreement (SLA). Due to different use cases, SLA can be used to describe many different things. For clarity, Google broke them down into different concepts.

First, it can describe a service-level indicator (SLI), which is a quantitative measure that precisely defines the quality of the service provided. Most services consider request latency (how long it takes to return a response to the request) as a key SLI. Other common SLIs include error rate and system throughput (often measured in requests per second). Usually the measurement results are summarized, i.e., raw data are collected in the measurement window, and then converted to a ratio, average or percentile.

Then comes the SLO, which is the target value or the range of service levels measured by SLI. Therefore, the structure of SLO is:  $SLI \leq \text{target value}$ , or  $\text{lower limit} \leq SLI \leq \text{upper limit}$ . For example, we may decide to adopt such an SLO: the average query request latency time should be less than 100ms.

With the SLO settings, we can set the SLA and use SkyWalking to monitor them and configure alarm thresholds.

#### A. SLO settings

Selecting the appropriate SLO is not easy.

First, you may not know what value to choose. For online HTTP requests, the query per second (QPS) metric is basically determined by the user's needs. Therefore, the SLO should not be set based on this metric.

Second, if you would like the average latency of each request to be less than 200ms, such a goal may in turn encourage developers to achieve lower latency levels or to invest in more equipment. Although this seems to be an advantage, the purpose of SLO will be defeated over time.

The third point is more nuanced. These two SLIs (QPS and latency) may have a certain correlation. Higher QPS usually leads to higher latency, and the performance of the service will fluctuate widely after reaching certain thresholds.

For the above reasons, it is necessary to actively communicate with users and set reasonable goals when setting up the SLO. Users often have over-ideal requirements for performance. If there is no process for setting goals, it is very easy to get an unrealistic performance goal, and even the performance goal will run counter to the system design and maintenance goals.

### *B. Setting alarm threshold*

With SLO, you can set alarm rules based on this information. The following is an example of setting the alarm threshold:

```
rules:
  # Rule unique name, must be ended with '_rule'.
  endpoint_percent_rule:
    # Metrics value need to be long, double or int
    metrics-name: endpoint_percent
    threshold:
      75 op: <
    # The length of time to evaluate the
    metrics
    period: 10
    # How many times after the metrics match the condition, will
    trigger alarm
    count: 3
    # How many times of checks, the alarm keeps silence after alarm triggered,
    default as same as period.
    silence-period: 10

  service_percent_rule:
    metrics-name: service_percent
    # [Optional] Default, match all services in this metrics
    include-names:
      - service_a
      - service_b
    threshold: 85
    op: <
    period: 10
    count: 4
```

Two alarm rules are shown here.

SLI corresponds to metrics-name and include-names. If the latter does not exist, all services of this monitoring metric will be configured with this alarm rule. The alarm threshold only supports numerical values. Period and count have to be used in combination. For example, "period=10, count=3" means that the alarm is triggered when the alarm condition is met 3 times every 10 minutes. silence-period means that if an alarm persists after being triggered, the alarm will go off only after this time period. It is consistent with period by default. This parameter is designed to prevent frequent alarms from affecting the judgment of the maintenance team.

### *C. Alarm message notifications*

SkyWalking does not have a plug-in alarm notification mechanism, but uses Webhook to push the alarm information to an HTTP service, and then the HTTP service decides how to process the information. The HTTP service may elect to send it to various notification channels, such as email, WeChat, and mobile text messages. It may also trigger various automatic repair mechanisms. These strategies are completely up to the user's control. The alarm message format is as follows:

```
[
  {
    "scopeId":1,
    "scope":"SERVICE",
    ",
    "name":"serviceA",
    "id0":12,
    "id1":0,
    "ruleName":"service_resp_time_rule",
    "alarmMessage":"alarmMessage xxxx",
    "startTime":1560524171000
  },
  {
    "scopeId":1,
    "scope":"SERVICE",
    ",
    "name":"serviceB",
    "id0":23,
    "id1":0,
    "ruleName":"service_resp_time_rule",
    "alarmMessage":"alarmMessage yyy",
    "startTime":1560524171000
  }
]
```

Starting from version 6.5.0 of SkyWalking, it is possible to configure alarm rules and alarm report notifications through the dynamic configuration module. Since it takes effect dynamically, the rules will take effect only after a time window has elapsed.

#### D. Removing alarm report

After getting notified or looking up the alarm message, you can search/browse the alarm page, as shown in Figure 3-32.

First of all, there are different dimensions to alarm messages. Therefore, it is possible that alarm messages from different dimensions in fact represent the same problem. For example, the service alarm message from projectA and the alarm message from the process instance most likely represent the same piece of information.

Second, it can be observed that the SLA alarm of projectB is derived based on whether the value is lower than 80% for more than 2 minutes within every 10 minutes. The interval between the occurrence of each alarm is about 3 minutes, which is consistent with the default configuration of the following system.

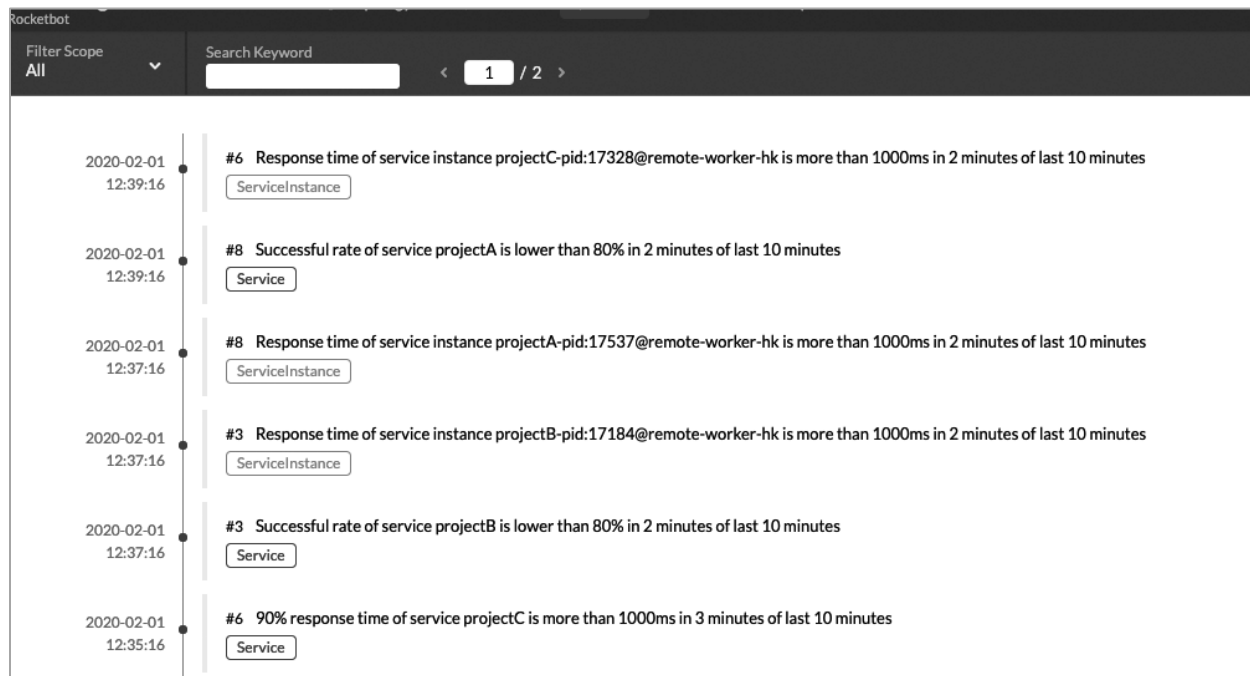


Figure 3-32 Alarm query



```

service_sla_rule:
  # Metrics value need to be long, double or int
  metrics-name: service_sla
  op: "<"
  threshold:
    8000
  # The length of time to evaluate the metrics
  period: 10
  # How many times after the metrics match the condition, will trigger alarm
  count: 2
  # How many times of checks, the alarm keeps silence after alarm triggered,
    default as same as period.
  silence-period: 3
  message: Successful rate of service {name} is lower than 80% in 2 minutes of
    last 10 minutes

```

Now, let's remove the error code introduced by projectB.

```
projectB/src/main/java/org/skywalking/springcloud/test/projectb/service/ServiceController.java
```

Package it again and deploy the projectB service. You will find that the relevant alarm message has been removed.

### 3.5 Chapter summary

In this chapter, we have introduced two architectural models and SkyWalking's support for these two models. We have explained why they are mainly applicable to microservice scenarios. Then, using the illustration of a use case in the live environment, we have introduced how to troubleshoot and solve various issues. From this example, you will have learnt about the key functions of SkyWalking.

In the next chapter, we will study the implementation, model, and design of SkyWalking, and learn about how SkyWalking implements the various features introduced in this chapter.

## Chapter 4:

# Lightweight-queue kernel

---

The asynchronous decoupling between data collection and data reporting in SkyWalking is done through a kernel with a lightweight queue. In this chapter, you will be introduced to the design and principle of the lightweight-queue kernel. After reading this chapter, you will have a basic understanding of this type of kernel, and learn how to develop customized queues according to actual business requirements.

### 4.1 What is a lightweight-queue kernel?

The lightweight-queue kernel is a producer-consumer in-memory message queue built on the basis of a lockless ring buffer. Its main function is to create a buffered asynchronous memory queue between the producer and the consumer to prevent data backlog and blockage resulting from a much higher rate of data production than the rate of sending data to the back-end by SkyWalking's data collection end.

The components of the lightweight-queue kernel are Buffer, Channel and DataCarrier, which will be introduced below.

#### 4.1.1 Buffer

Buffer is the data carrier in the kernel of the SkyWalking queue, and the data in the queue is stored in the buffer.

Buffer has the following properties:

- `Object[] buffer`: Queue for data storage.
- `AtomicRangeInteger index`: An atomic loop index, which implements a ring queue together with Buffer.

- `BufferStrategy`: Queue strategy.
- `List<QueueBlockingCallback<T>> callbacks`: Callback function when data is blocked.

Let's study these four properties in detail and learn about their relationships.

`Object[] buffer` is an `Object` array with a fixed length (determined by the `buffer.buffer_size` configuration introduced in Chapter 2). `AtomicRangeInteger index` is an atomic loop index based on the atomic class `AtomicIntegerArray`. The maximum value of this index is the length of `Object[] buffer`. `Object[] buffer` provides a medium for data storage, and the `AtomicRangeInteger index` offers atomicity and cyclicity. These two parts form SkyWalking's lightweight-queue kernel in the form of a lockless ring.

When the Producer produces the data, by referring to the `AtomicRangeInteger index`, it determines which index of the `Object[] buffer` is to be used to store the current data.

`BufferStrategy` is a queue strategy. It is a solution for the problem that arises when the Producer stores data in a Buffer index, and the relevant index and the old data are not consumed.

`List<QueueBlockingCallback<T>> callbacks` is a callback function whose production data of Producer is loop-blocked under the blocking strategy `BufferStrategy.BLOCKING`.

#### 4.1.2 Channel

`Channel` is the carrier that manages `Buffer`. It has the following properties:

- `Buffer<T>[] bufferChannels`: Buffer array collection, which is the data carrier in the queue kernel. Figure 4-1 shows the structural relationship between `Channel` and `Buffer`.
- `IDataPartitioner<T> dataPartitioner`: When data is written into `Channel`, the `dataPartitioner` will determine which `Buffer` the data should be stored in and the number of retries to be taken in the event of data storage failure.
- `BufferStrategy strategy`: `BufferStrategy` in `Channel` is kept consistent with the content of `Buffer`.
- `Long size`: The size that `Channel` is able to accommodate. The value is the quantity of `Buffer` in `Channel` multiplied by the length of the buffer array within each `Buffer`.

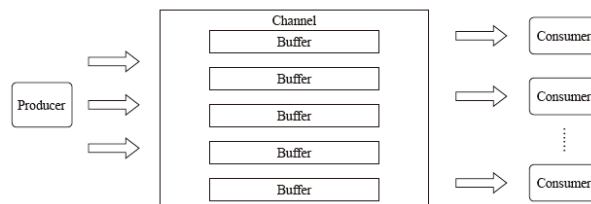


Figure 4-1 Structural relationship between `Channel` and `Buffer`

### 4.1.3 DataCarrier

DataCarrier is a lightweight-queue kernel gateway. The queue kernel interacts with other SkyWalking modules through DataCarrier.

DataCarrier needs the parameters CHANNEL\_SIZE and BUFFER\_SIZE during initialization. The former is used to determine how many buffer queues there are in the current Channel, and the latter is used to determine the size of each buffer queue.

## 4.2 Producer-consumer coordination

In the previous section, we have seen that lightweight queues are based on producer-consumer in-memory message queues. This section mainly introduces how producers of the lightweight-queue kernel produce data in the queue and how consumers consume data in the queue correctly.

### 4.2.1 Producing messages

#### *A. How to produce messages*

The production of messages to the queue must be done through DataCarrier. Let's take the example of the queue in the trace data reporting module of SkyWalking to introduce how producers produce messages.

First, initialize DataCarrier. The code is as follows:

```
@DefaultImplementor
public class TraceSegmentServiceClient implements ... {
    ...
    private volatile DataCarrier<TraceSegment> carrier;
    ...
    @Override
    public void boot() throws Throwable {
        ...
        carrier = new DataCarrier<TraceSegment>(CHANNEL_SIZE, BUFFER_SIZE);
        carrier.setBufferStrategy(BufferStrategy.IF_POSSIBLE);
        carrier.consume(this, 1);
    }
    ...
}
```

There are three parts in the initialization process for the DataCarrier:

- 1) Set the number of Buffer queues in the DataCarrier Channel and the length of each Buffer queue;
- 2) Set BufferStrategy as IF\_POSSIBLE (as introduced earlier in Section 4.1);
- 3) Set the current DataCarrier consumer (to be introduced in Section 4.2.2).

Upon initialization of DataCarrier, you can use its API to produce messages to the queue.

```
@DefaultImplementor
public class TraceSegmentServiceClient implements {
    ...
    @Override
    public void afterFinished(TraceSegment
        traceSegment) { if
        (traceSegment.isIgnore()) {
            return;
        }
        if (!carrier.produce(traceSegment)) {
            ...
        }
    }
    ...
}
```

The production and use of the queue is relatively simple. You can produce data in the queue through `DataCarrier.produce(Data)`.

## *B. Message production principles*

### *i) Data distribution*

In the previous section, we have introduced the use of the SkyWalking lightweight-queue kernel producer. In this section, we will introduce the principles underlying message production by producers.

As introduced in Section 4.1, a lightweight queue is mainly composed of multiple Buffers. Since Buffer is the final data carrier, the producer has to do the following two things when producing data:

- a) Determine which Buffer the current data should be stored in;
- b) Determine where the data is to be stored in the Buffer.

SkyWalking uses the partition method of the `IDataPartitioner<T>` `dataPartitioner` interface to determine which Buffer the data is to be stored in. This interface has the following definition:

```
public interface IDataPartitioner<T> {  
    int partition(int total, T data);  
    int maxRetryCount();  
}
```

The meaning of the partition method `int partition(int total, T data)` is as follows: There are two input parameters. The total is the number of Buffer queues in the Channel, and data is the production data. The return value is the index value of the specific Buffer.

The default implementation of `IDataPartitioner` in Channel is an infinite loop between the first Buffer and the last Buffer. The specific implementation is as follows:

```
public class SimpleRollingPartitioner<T> implements IDataPartitioner<T> {  
    private volatile int i = 0;  
    @Override  
    public int partition(int total, T data) {  
        return Math.abs(i++ % total);  
    }  
}
```

Once it has been determined which Buffer the data is to be stored in, you can start storing data in the Buffer queue.

## *ii) Data storage*

After confirming the specific Buffer to be used, you should determine where the data should be stored in the Buffer. There is an index in Buffer that serves to determine the location of storage for the data in the Buffer (see Figure 4-2).

The index continues to grow from 0 up to the maximum length of the Buffer queue. Then, the cycle begins again from 0.

After selecting the specific location for the data in the Buffer, if the current location is empty, it will be stored directly and the whole process will be complete. Whereas, if there is unconsumed data in the location being chosen, SkyWalking offers three kinds of Buffer strategies to deal with this situation:



Figure 4-2 Buffer index

- **BLOCKING:** During loop blocking, the producer waits for the index space of the current Buffer queue to become empty (which is the default strategy), and the Buffer callback method will be called. The user may perceive whether the data has been blocked by setting the callback method.
- **OVERWRITE:** Override old data with new data.
- **IF\_POSSIBLE:** Find the nth unit in the current index. If there is space, save it; if not, discard it. N is determined by `maxRetryCount` in `IDataPartitioner<T>` `dataPartitioner` interface. Its default value is 3.

## 4.2.2 Message consumption

In the previous section, we have introduced how the producer of the queue kernel produces messages into the queue. This section explains how the consumer end of the lightweight queue consumes data.

### A. How to consume messages

Here is an example of the queue in the trace data reporting module of SkyWalking. Consumers of the queue need to implement the `org.apache.skywalking.apm.commons.datacarrier.consumer.IConsumer` interface. The code is as follows:

```
public interface IConsumer<T> { void init();
    void consume(List<T> data);
    void onError(List<T> data, Throwable t);
    void onExit();
}
```

The functions declared by each method in the `IConsumer` interface are as follows:

- **void init:** A function triggered when the implementation class of `IConsumer` is instantiated.
- **void consume(List<T> data):** The callback entry of consumption data. The data in the current consumer pull queue is the input parameter.
- **void onError(List<T> data, Throwable t):** Callback function when consumption fails.
- **void onExit():** This function is called when `DataCarrier.shutdownConsumers()` is explicitly called.

Once the consumer object is implemented, the next step is to let the consumer pull data from the queue. The lightweight queue completes consumption of queue data for the consumer using the listener mode. In other words, if consumers want to consume the data in the queue, they need to register themselves on the DataCarrier. In Section 4.2.1, we have introduced the initialization process for DataCarrier. In this process, `carrier.consume(this, 1)` represents the act of registration.

```
public DataCarrier consume(IConsumer<T> consumer, int num) {  
    return this.consume(consumer, num, 20);  
}
```

The two input parameters for this method respectively represent the consumer object and the number of threads. This method will eventually create a number of consumer threads with `IConsumer<T> consumer` as the consumer to consume queue data. This method also internally calls an overloaded function, but there is an additional input parameter. This parameter, represented by `20`, is `consumeCycle`, which is used to determine the interval time of the consumption cycle (to be introduced in the next section).

Binding the two sections above, the queue for the user terminal for the consumer, the consumer only needs to create an object and the corresponding consume (`List <T> data`) to achieve their consumption data logic process, and then register the object in DataCarrier. The consumption logic of the entire queue is now complete.

### *B. Principles for message consumption*

In the previous sub-section, we have introduced the use of SkyWalking's lightweight-queue kernel consumers. In this section, you will learn about the internal principles of message consumption by consumers. Let's start with Buffer queue allocation and the consumption logic.

#### *i) Buffer queue allocation*

As mentioned in Section 4.2.1, there are multiple Buffer queues in a Channel, and each Buffer queue is written into the data. So, how do consumers consume data on multiple Buffer queues?

Consumer objects need to input the number of consumer threads when registering with DataCarrier, and each consumer thread undergoes Buffer queue allocation before running. Buffer queue allocation refers to the correspondence between Buffer and consumer threads. For example, if there are five Buffer queues and five consumer threads, in order to prevent the



duplication of messages, the SkyWalking lightweight-queue kernel requires that the Buffer interval consumed by each consumer thread must not overlap (this is also the reason why this lockless queue can ensure thread safety). So, in this case, each consumer thread corresponds to only one Buffer queue.

But this example is only one of the many possible situations. The specific situation of Buffer queue allocation is determined by the number of Buffer queues and consumer threads. In general, there are two kinds of situations regarding the number of Buffer queues and consumer threads:

1) *The number of Buffer queues is greater than or equal to the number of consumer threads*

In this case, each consumer thread links up with one or more Buffer queues in order. If there are five Buffer queues and three consumer threads, consumer thread 1 links up with Buffer1 and Buffer4; consumer thread 2 links up with Buffer2 and Buffer5; and consumer thread 3 links up with Buffer3. If the number of Buffer queues is equivalent to the number of consumer threads, then each consumer thread links up with a single Buffer queue.

2) *The number of Buffer queues is less than the number of consumer threads*

In this case, each Buffer queue may be consumed by multiple consumer threads at the same time, but the Buffer interval for the consumption will not overlap. The specific distribution logic is as follows:

- a) Each consumer thread links up with Buffer queue association. Figure 4-3 illustrates the example of five consumer threads and three Buffers.

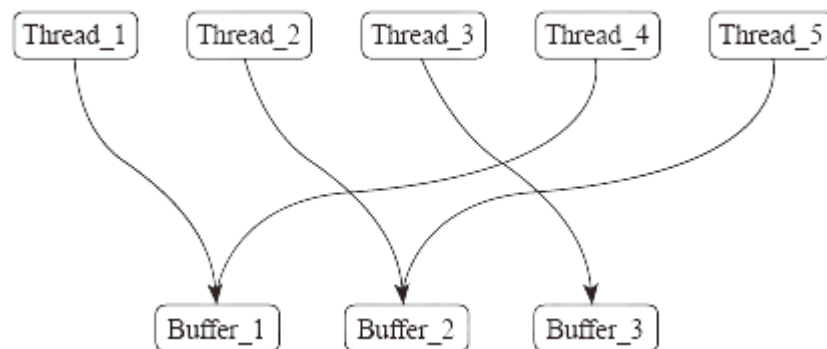


Figure 4-3 Relational illustration when the number of consumption threads is more than the number of Buffers

As shown in Figure 4-3, each consumer thread corresponds to a Buffer queue. Some Buffer queues correspond to two consumer threads (Buffer\_1 and Buffer\_2), while some only correspond to a single consumer thread (Buffer\_3).

- b) The consumer threads traverse the Buffer queues in order. Divide Buffer length by the number of consumer threads for this Buffer, and divide the Buffer queue index interval equally between the consumer threads for this Buffer. For Buffer\_1 in Figure 4-3, if the length of Buffer\_1 is 500, Thread\_1 will link up with Buffer\_1 queue index in the interval between 0 - 249; Thread\_4 will link up with Buffer\_1 queue index in the interval between 250 - 499. The specific distribution is shown in Figure 4-4:

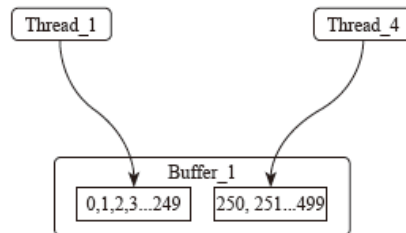


Figure 4-4 Synchronous consumption of a single buffer by multiple threads

## ii) Consumption logic

After the buffer queue interval corresponding to each consumption thread has been allocated, the next step is to execute the consumption logic. The code for implementation is as follows:

```
@Override
public void run() {
    running = true;
    final List<T> consumeList = new ArrayList<T>(1500);
    while (running) {
        if (!consume(consumeList)) {
            try {
                Thread.sleep(consumeCycle);
            } catch (InterruptedException e) {
            }
        }
    }
    // consumer thread is going to stop
    // consume the last time
    consume(consumeList);
}
```

```

        consumer.onExit();
    }

```

Each consumer thread initializes a `consumeList` with an initial capacity of 1500, and then starts the consumption cycle. The consumption logic traverses the Buffer queue interval corresponding to the consumer thread from beginning to end, stores it in the `consumeList` when it encounters a non-null value, and sets the index under the Buffer queue as null.

The interval of each cycle is the millisecond of the `consumeCycle`. This parameter is input when the consumer registers with the `DataCarrier`.

After obtaining all the consumable data, the consumer's void `consume(List<T> data)` will be called to consume data:

```

private boolean consume(List<T> consumeList) {
    ...
    if (!consumeList.isEmpty()) { try {
        consumer.consume(consumeList);
    } catch (Throwable t) {
        consumer.onError(consumeList, t);
    } finally {
        consumeList.clear();
    }
    return true;
}
return false;
}

```

The structure of this method is relatively simple, and it can clearly demonstrate the role of the declared method in the `IConsumer` interface.

## 4.3 Chapter summary

In this chapter, you have been introduced to the design concept and details of SkyWalking's lightweight-queue kernel. Here, SkyWalking's trace data reporting module has been used for the analysis, but it is not the only place where lightweight queues are used. Lightweight queues have an important role and are extensively used in SkyWalking. The API design for the lightweight-queue kernel is clear and simple. You can modify the queue parameters and further customize the queue according to your actual business requirements.

## Chapter 5:

# SkyWalking trace model

---

The trace model is the foundation of a distributed tracing system. Nowadays, distributed tracing technologies are widely used in production-level systems. Were there no innovative changes in the trace model, such wide usage wouldn't have been possible. On the basis of classic theories, SkyWalking's trace model pursues actual production results.

In this chapter, you will be introduced to the classic trace model, the different technical routes available, and the pros and cons of these different routes. Then, you will learn about the characteristics of the SkyWalking trace model and its advantages compared to the classic model.

## 5.1 Getting started with the trace model

Modern Internet services are usually implemented in the form of complex large-scale distributed systems. These applications may consist of a collection of software modules developed by different teams. They may be in different programming languages, and may span multiple physical devices or even thousands of computers. In such an environment, it is essential to have monitoring and diagnostic tools to assist users in understanding system behavior and related performance issues.

### 5.1.1 Dapper and trace model

Dapper is Google's production distributed system tracing infrastructure. It is designed to meet the tracing and monitoring needs of ultra-large-scale systems. It offers features such as low overhead, application-level transparency, and multi-environment deployment. Having been used within Google for more than 10 years, Dapper's success can be attributed to its sampling and strict restrictions on external

components. Of course, the most remarkable results came about by virtue of the efforts of the developers and the operations team who used Dapper to accomplish their work.

At the very beginning, Dapper was an independent tracing tool. It later developed into a monitoring platform. At the same time, many different tools derived from it and some were not even foreseen by its designers. Google described some analytic tools built with [the Dapper paper](#) (2010), shared statistics and information on exceptional conditions regarding internal usage, and provided an informative summary of its experience. Google built Dapper to offer developers a solution to analyze the behavior of complex distributed systems, and proved that this system is very helpful to companies of this magnitude.

After briefly introducing the background of Dapper, let's take a look at the main problems targeted by Dapper. Figure 5-1 shows a typical distributed system.

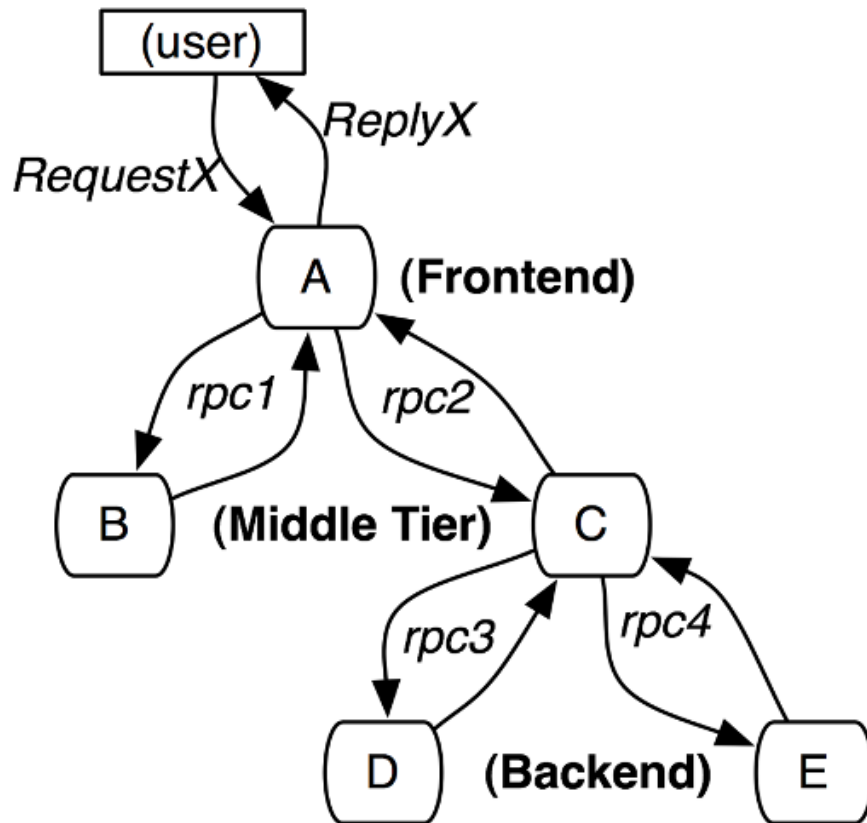


Figure 5-1 Distributed system topology diagram

At the time of publication of the Dapper paper (2010), there were the following two methods for troubleshooting in a complex distributed system.

### A. The “black box” method

The “black box” method is based on the principle that the target system should be regarded as a “black box”, and the focus should be placed on messages between systems. In Figure 5-2, the thin line represents the data actually obtained, and the thick line represents the internal call of the service node, which involves data inferred by the system.

When data collection for the thin line is completed, statistical algorithms such as regression analysis are used to recombine these fragments into a complete trace.

### B. The “tagging” method

The “tagging” method requires tagging the messages. As shown in Figure 5-3, this method uses a global trace ID combined with some other tags, such as parent ID and child ID. This allows the message to form relationships at the data level, thus stringing together an entire trace.

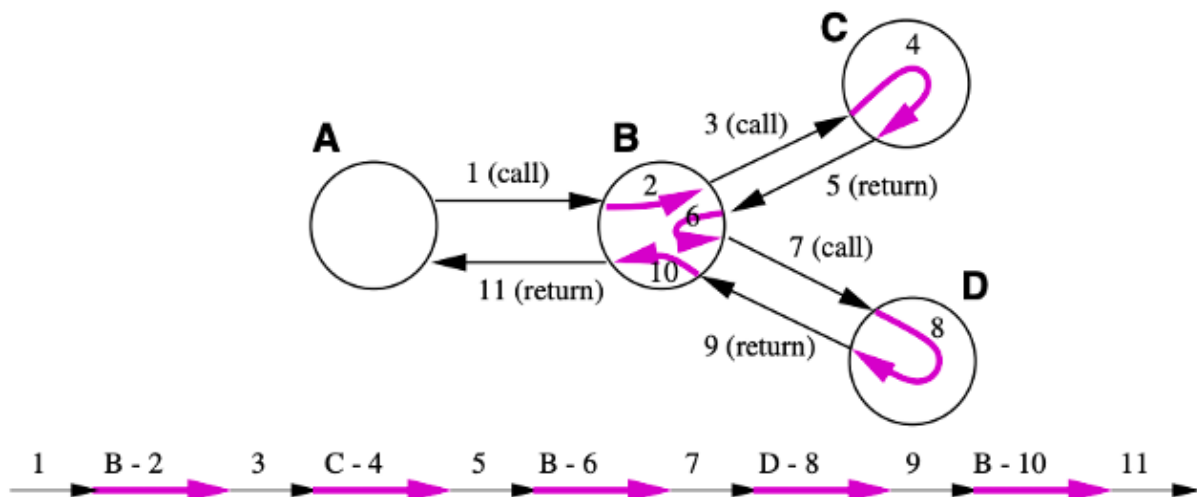


Figure 5-2 The “black box” method

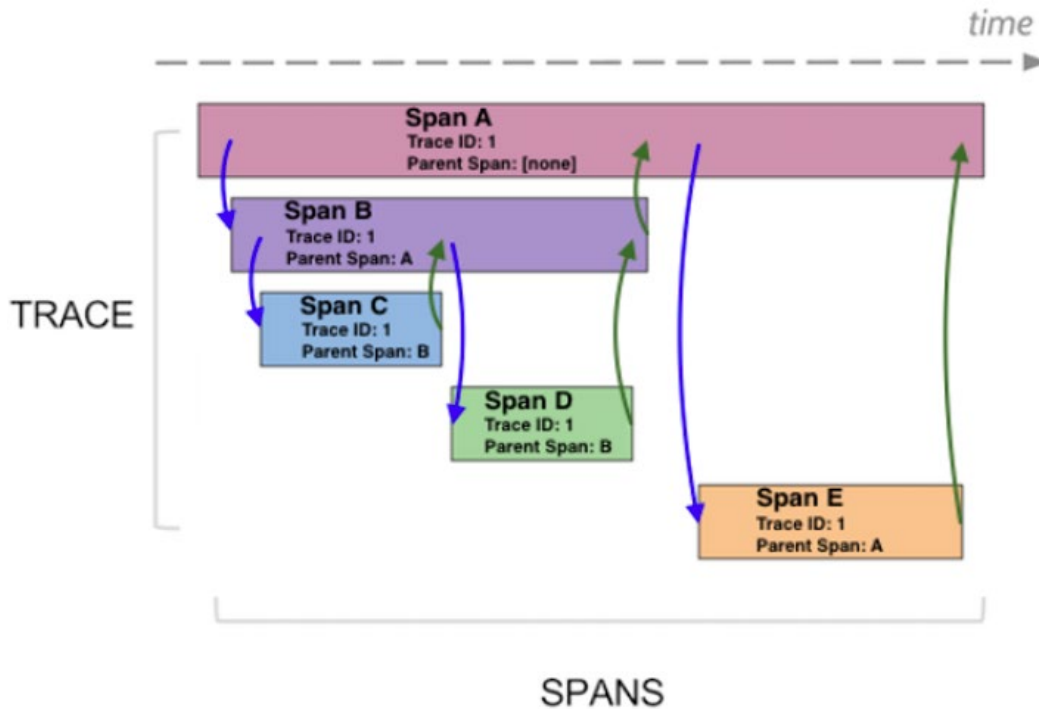


Figure 5-3 The “tagging” method

Compared with the tagging method, the black box method has the advantage of being non-intrusive to the body of the message body and easier to deploy. However, it requires more data for analysis in order to obtain more accurate results. An obvious disadvantage of the tagging method is that it needs to enter into the target system to add additional labels.

Dapper uses the latter method of tagging. Since various RPC calls inside Google share a common repository, additional tags can be added when only part of the code is changed.

Usually, the tracing target of Dapper is an RPC nested tree. However, in practice, it is also often used in non-RPC scenarios, such as SMTP mail sending, external incoming HTTP requests, and SQL access to databases. Therefore, Dapper is concerned with call trees, Spans and message tags, and does not impose the limitation that the data must be coming from RPC.

### 5.1.2 Typical trace model

Now let's learn about a typical trace model, which comes from Dapper and is widely used by systems like Zipkin and SkyWalking. Without doubt, SkyWalking has made some improvements to this model. But as the typical model is easier to understand, this will be our starting point to understand SkyWalking's own model.

Figure 5-4 shows a typical trace tree model, which is composed of a group of interrelated nodes generally known as Spans. The connection between the two ends of Span indicates the relationship between it and its parent Span. A Span usually contains a timestamp, the start and end time of Span, the traceId for the entire trace tree, current Span ID, parent Span ID, as well as some other application-related information used to store the current Span.

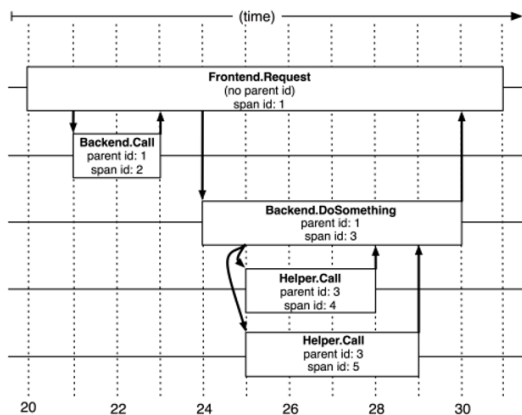


Figure 5-4 Classic trace tree model

The SpanId and ParentId in the Span are essential for linking up the entire trace tree. The Span without the ParentId is known as the Root Span. All Spans under the same trace tree Span share an identical traceID. All IDs should be unique. Each Span represents a call, and each additional layer of service calls will result in an increase in the level of the trace tree.

Figure 5-5 shows a typical RPC Span. The start and end times of the Span are obtained by the trace agent from the target RPC repository. "foo" is a user-defined tag, which is stored in the back-end together with other Span tags.

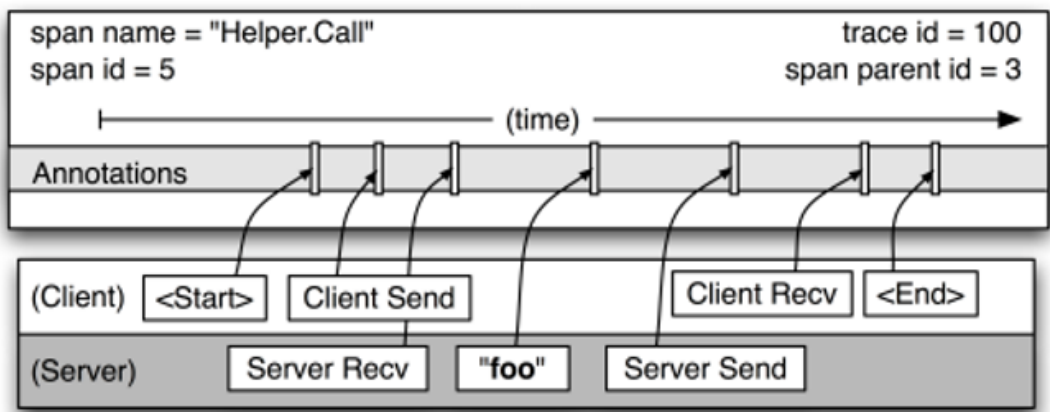


Figure 5-5 RPC Span



Note that a Span can contain the data of multiple nodes. In this example, Span contains data from both the client and server. Since the data generally come from two hosts, it is particularly important to ensure the consistency of the time between multiple hosts.

## 5.2 SkyWalking trace model and protocol

In this section, you will be introduced to the trace model of SkyWalking, with a focus on the differences between SkyWalking's model and the classic model. You will also learn about the trace protocol. This will be especially helpful for those who wish to develop language agents.

### 5.2.1 SkyWalking trace model

Figure 5-6 shows a set of commonly found trace data in SkyWalking, where each dot represents a Span. The difference between SkyWalking's model and the classic trace model is highlighted as follows:

- There are Segments. SkyWalking innovatively came up with the concept of Segment. A Segment is the collection of all Spans of the trace in a process. If multiple threads work together to produce a trace, they will only create a single Segment, not multiple. Different colors are used to identify different segments in the UI. In Figure 5-6 in, projectA, projectB and projectC represent three different Segments respectively.
  - There are various reasons for introducing the concept of Segment. First, it is based on performance considerations. By combining all Spans in the same instance and sending them in batches, efficiency will be greatly improved, especially when there is a higher number of Spans. Second, the use of Segments allows users to observe the call relationship between services, and enriches the visual content.
- There is no Root Span. SkyWalking's first Span without a parent Span is not called Root Span. Since the model supports multiple entries, there is no unique root node. As a result, SkyWalking has removed the concept of Root Span.
- One call is saved in two Spans. As shown in Figure 5-6, RPC calls are stored in two Spans. The advantage of this design is that back-end processing will be relatively simple. As there is no need to perform the obstructive act of data integration, the throughput of data processing will be improved. At the same time, it prevents the analysis back-end from hanging when receiving Server data.

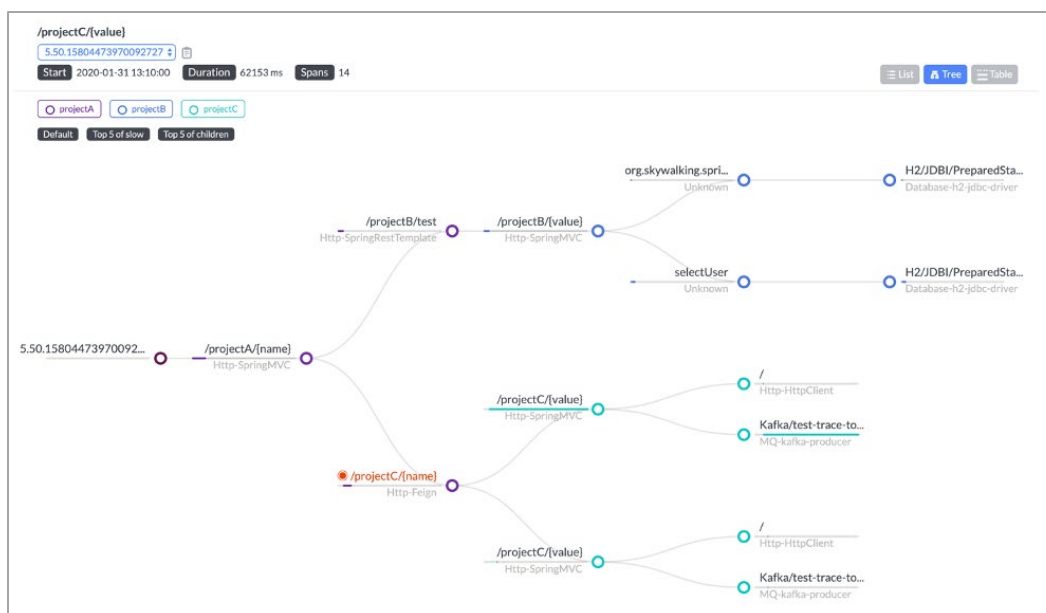


Figure 5-6 Trace data of SkyWalking

Let's look at another type of SkyWalking trace data, as shown in Figure 5-7.

Kafka/test-trace-topic/Consumer/test							
14.45.15804479863243797							
Start 2020-01-31 13:19:55 Duration 4 ms Spans 13							
Method	Start Time	Gap(ms)	Exec(ms)	Exec(%)	Self(ms)	API	Service
✓ /projectA/{name}	2020-01-31 13:...	0	2587		1	SpringMVC	projectA
✓ /projectB/test	2020-01-31 13:...	0	1507		98	SpringRestTemplate	projectA
✓ /projectB/{value}	2020-01-31 13:...	0	1409		1001	SpringMVC	projectB
org.skywalking.springcloud.test.projectb.dao.DatabaseOperateDao.saveUser(java.lang.String)	2020-01-31 13:...	0	0		0	-	projectB
H2/JDBI/PreparedStatement/execute	2020-01-31 13:...	0	0		0	h2-jdbc-driver	projectB
selectUser	2020-01-31 13:...	0	408		0	-	projectB
H2/JDBI/PreparedStatement/execute	2020-01-31 13:...	0	408		408	h2-jdbc-driver	projectB
✓ /projectC/{name}	2020-01-31 13:...	0	1079		1	Feign	projectA
✓ /projectC/{value}	2020-01-31 13:...	0	1078		1039	SpringMVC	projectC
/	2020-01-31 13:...	0	38		38	HttpClient	projectC
Kafka/test-trace-topic/Producer	2020-01-31 13:...	0	1		1	kafka-producer	projectC
✓ org.skywalking.springcloud.test.projectd.MessageConsumer.consumer()	2020-01-31 13:...	0	4		1	-	projectD
Kafka/test-trace-topic/Consumer/test	2020-01-31 13:...	0	3		3	kafka-consumer	projectD

Figure 5-7 Multi-Entry Span trace data

This data reflects SkyWalking's ability to support multiple Entry Spans at the same time. We can see that Kafka's message production and consumption can be stored in the same trace.

## 5.2.2 SkyWalking data transfer protocol

In this section, you will be introduced to the relevant data transfer protocols used by SkyWalking. The specific content of each protocol will not be explained in detail here, since the protocols will change as time passes. The focus will be on the logic behind these protocols so as to help you understand what they mean.

### *A. Registration and heartbeat protocols*

The registration protocols are stored in <https://github.com/apache/skywalking-data-collect-protocol/blob/v6.6.0/register/Register.proto>. It includes protocols on service registration, service instance registration, and registration of other entities. The first two protocols are the most important ones.

In the registration process, first you need to register the service, and then use the service ID returned by the server to register the protocol. Two main points to note here:

- The registration process is performed asynchronously, so the registration result is likely to return as NULL. When using this service, you should put it in a loop until you return to the normal result, before you finally send out the data. After each cycle, you will need to wait for a while before continuing the operation.
- The operation can be executed in batches, so carefully check whether the returned result corresponds to the client's requirements.

The rest of the protocol in the registration are optional components that improve the efficiency of data transmission.

After the registration is completed, the service instance needs to send a heartbeat to the back-end. The protocol is <https://github.com/apache/skywalking-data-collect-protocol/blob/v6.6.0/register/InstancePing.proto>. This protocol is used to help the back office determine if the process is alive.

Note that there is no register in v3.

### *B. Data collection protocol*

The data collection protocol can be found at <https://github.com/apache/skywalking-data-collect-protocol/blob/v6.6.0/language-agent-v2/trace.proto>. Note the following points:

i) The Segment object stores all Spans during a single access within a process.

ii) Spans can be classified as entry, local, and exit. They are described as follows:

- Entry indicates a request to enter the segment. It is usually an HTTP or RPC service, such as SpringMVC and httpServer.
- Local indicates an internal request within Segment. It is usually a function call or cross-thread call.
- Exit indicates a call out of Segment. It is usually an httpClient, where the database driver performs operations such as accessing the database.

iii) SegmentRef is used to link up different Spans. It contains relevant fields as well as redundant fields to speed up analysis and processing in the back-end. Ref is designed as an array to support multiple Entries.

- In most cases, the service call involves only one Entry. In this case, there is only one value in Ref.
- In some cases, there are two Entries. Typically, if the MQ consumer consumes in batches and these messages come from different producers, there will be two or more Entry Spans.

iv) Some fields in the protocol contain both ID and name. Change name to ID in the relevant parts of the registration protocol. Then, you can use ID in place of name for data transmission. In this way, the amount of data being sent can be reduced, improving transmission efficiency.

### *C. Protocols for other areas*

This is a set of protocols, including JVM metrics protocol, CLR metrics protocol, and general service mesh protocols. These protocols are designed for specific scenarios, so they can be adopted according to users' actual needs.

SkyWalking will add more field protocols in the future to support more use cases.

## 5.3 Context propagation protocol for the SkyWalking agent

SkyWalking's propagation protocol is a key component that links Segment and Span together.

### 5.3.1 Propagation model

SkyWalking's propagation protocol is much more complex than traditional distributed tracing protocols. In terms of its specifications, it shares similarities with the protocols of commercial APM systems. The following is an example:

```
1-TRACEID-SEGMENTID-3-5-2-IPPORT
```

The fields in the protocol are connected by "-", and all string types are usually calculated using Base64 encode. Since there is only one set of data, storage can be done with just a single field.

After the string is formed, it can be placed into the RPC protocol to propagate across the entire set of microservices, as shown in Figure 5-8.

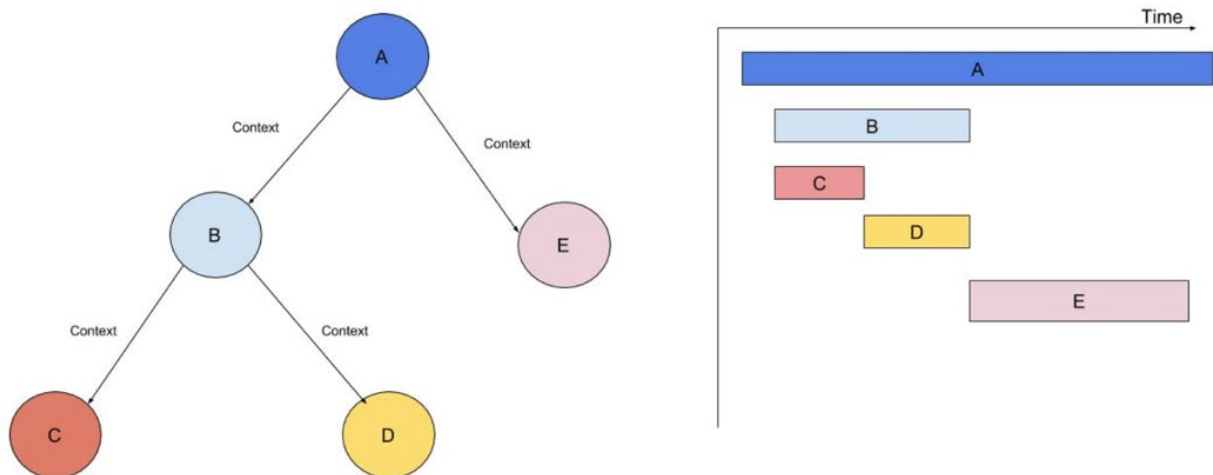


Figure 5-8 Context propagation across services

### 5.3.2 Propagation context

The propagation context is a key-value pair, where the key is sw6, and the value consists of multiple fields, including required and optional fields.

These are the required fields:

- Sampling mark: 0 or 1; 0 means no sampling, 1 means sampling is currently required.
- TraceId: Global trace ID made up of three long type numbers.

- Parent SegmentId: ID of the Segment one level above the current Segment. It is also made up of three long type numbers.
- Parent SpanId: The ID of the Span one level above. It should be found in the Segment one level above.
- Parent service instance ID: Registration ID of the process one level above.
- Entry service instance ID: Registration ID of the entry service instance.
- Destination address for the request: A network address used by the client to access the server. It is not necessarily IP and PORT. Some other possibilities include a character string, or a network address ID registered with a registration protocol.

In addition to the required fields, there are also some optional fields for users. These are two optional fields:

- Entry endpoint: Name of the endpoint for the entry service. It can be a character string, or the ID after registration.
- Parent service endpoint: Endpoint for the entry of the upper-level service. It can be a character string or the ID after registration.

Figure 5-9 shows how to nest the propagation context in the HTTP protocol. Place the propagation context in the Header in the HTTP protocol. Other RPC protocols generally provide user-defined fields, in which the propagation context protocol can be written, and then propagated.

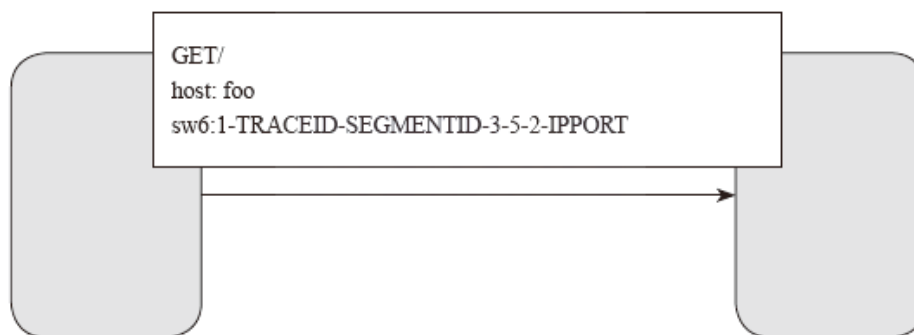


Figure 5-9 HTTP protocol propagation context

## 5.4 SkyWalking protocol v3

SkyWalking has fully adopted protocol v3 in version 8.0. This upgrade has addressed the issue of incompatibility of the trace protocol and header protocol with the existing version. After a comprehensive evaluation by the community, a decision was made to cancel the exchange registration between data ID and name, and only the string type name was kept in all protocols. Therefore, the protocols as described

above remain fundamentally the same in terms of content and logic. Only the registration mentioned in Section 5.2.2 above has been removed, and other protocols no longer contain the ID fields.

## 5.5 Chapter summary

In this chapter, we have introduced SkyWalking's trace model. First, the classic trace model is introduced using the Dapper model. Then, the SkyWalking model is compared against the classic model to observe their differences. Lastly, the propagation protocol is introduced.

After reading this chapter, you will have had a basic understanding of the details and logic of the SkyWalking trace model. This is particularly beneficial to those who will use and maintain SkyWalking. It will be of tremendous help if users decide to develop their own business and language probes.

In the next chapter, you will learn about the architecture and design of SkyWalking's back-end.

## Chapter 6:

# SkyWalking OAP Server modularized architecture

---

The modularized architecture is the nucleus of the SkyWalking back-end OAP Server architecture. Learning about SkyWalking's modularized design will help you better understand the startup sequence, configuration files, source code, possible extension points and extension methods of back-end services. Everyone, including beginners, should give this chapter a careful read. Beginners will find Sections 6.1 and 6.2 particularly helpful, as they explain the startup errors output by the OAP Server after incorrectly modifying the configuration files.

## 6.1 Modularized framework

A modularized framework is a program that does not use hard-coded coupling for program linking, but instead organizes itself into multiple modules with no coupling relationships between the modules, and defines only the open service interfaces for the modules (Java API). Each module may have multiple providers (ModuleProvider), but only one provider of each module can be activated during each startup.

### 6.1.1 Module and module providers

The above-mentioned modules and module providers shall be explained further. The module definition (ModuleDefine) can be freely created, and each module definition needs to contain the following key information.

- **Module name:** Any character string name. It must be unique in the system and must not overlap with other loaded modules.



- A list of open API services: Each interface must implement `org.apache.skywalking.oap.server.library.module.Service`. There are no methods in this interface. It only requires explicit declaration of the program to make clear that this interface will be used across modules.

Take the cluster coordinator module as an example. It is used to ensure the normal operation of the OAP module in the cluster mode. It contains the following definitions:

```
public class ClusterModule extends ModuleDefine { public static final String NAME = "cluster";
    public ClusterModule() {
        super(NAME);
    }
    @Override public Class[] services() {
        return new Class[] {ClusterRegister.class, ClusterNodesQuery.class};
    }
}
```

Cluster is the name of this module. ClusterRegister and ClusterNodesQuery are the open service interfaces of this module. All module providers must provide the actual providers for these two interfaces.

The module provider (ModuleProvider) is attached to the module definition, and each module provider can only be defined for one module. Module providers are required to register the provider classes of all open service interfaces to ensure the completion of all functionalities of this module. The following logic must be implemented:

- Names of the ModuleProvider and character string must be unique in all providers of the same module.
- The module to which it belongs, which points to the module definition above.
- Related configurations: Different providers of the same module provide very different configurations due to the variations in technical details. Therefore, the configuration information is defined at the provider level, instead of the module level. For example, the two providers of the memory module, MySQL and Elasticsearch, are distinct in terms of access characteristics and parameters. Here you can use any JavaBean as the configuration class. The modularized framework will initialize the configuration class through dependency injection.
- List of module dependencies: Dependencies and the related configurations are similar. As different providers have very different technical details, their module dependencies widely vary.
- Registration of provider class of open service interface list: The module provider requires that all method provider classes are registered in the module provider initialization phase (prepare), and the service is guaranteed to be available after the initialization phase. Therefore, in the service initialization phase, most modules will use a blocking method to complete the initial loading of

the response. Note that at this stage, the module provider cannot call the services of any other modules. Otherwise, an exception of "Still in preparing stage" will be asserted.

- The module provider will also provide `notifyAfterCompleted` notifications for the phases of start and global start. This is related to the module startup sequence and dependencies, which will be introduced in detail in Section 6.2.

For the Cluster module in the module definition, you can refer to `ClusterModuleZookeeperProvider` to view the definition method and logic of the module providers.

## 6.1.2 Module management configuration file

The modularized kernel of the OAP Server is driven by `application.yml` in the `Config` folder. This file manages and drives which modules are to be started and which providers of each module are to be used. `OAPServerBootstrap` is the main entry point of the entire OAP Server. It uses the modularized kernel `ModuleManager` to load the configuration file.

The `application.yml` file is in the standard YAML format. From a logical standpoint, it can be classified into three layers, namely: module definition, module provider definition, and module provider parameter definition. The following shows an excerpt of a typical module configuration file:

```
core:
  default:
    # Mixed: Receive agent data, Level 1 aggregate, Level 2
    aggregate # Receiver: Receive agent data, Level 1
    aggregate
    # Aggregator: Level 2 aggregate
    role: ${SW_CORE_ROLE:Mixed} # Mixed/Receiver/Aggregator
    restHost: ${SW_CORE_REST_HOST:0.0.0.0}
    restPort: ${SW_CORE_REST_PORT:12800}
    restContextPath: ${SW_CORE_REST_CONTEXT_PATH:/}
    gRPCHost: ${SW_CORE_GRPC_HOST:0.0.0.0}
    gRPCPort: ${SW_CORE_GRPC_PORT:11800}
    downsampling:
      - Hour
      - Day
      - Month
```

This configuration module can be interpreted as: OAP needs to start the core module at this time, and the module provider is set as default.<sup>3</sup> The third level configuration, such as `role`, `restHost`, and `restPort`, are all configuration parameters for the default provider.

In addition to the hierarchical structure of module configuration, note also the characteristics related to configuration parameters:

- Configuration parameters may not be fully included in the `application.yml` file officially released by Apache. The file only provides most of the optional configurations. A better way to obtain all configurations is to quickly read the source code. For example, the configuration parameter class for the default provider of the core module `CoreModuleProvider` is defined and executed as `CoreModuleConfig`, so all the properties of this class are parameters that can be used in `application.yml`.
- Parameter value setting method: All configuration properties can be written as `${variable name: default value}`. For example, `port` can be written as `${SW_CORE_REST_PORT:12800}`. This variable name refers to the ability to override this parameter value at startup through system environment variables. This is very important in the container environment, or when switching between testing, quasi-production, and production environments. It ensures that all environments are able to use the same OAP Server binary package or image, as long as the environment variables correspond to the specified environment.

SkyWalking provides more than 10 default modules and multiple providers. In the next section, you will learn about dependency and startup for these modules.

## 6.2 Module startup and module dependency

In the previous section, we introduced modules, module providers, and module management configuration files. It has been mentioned that these module providers have a dependency relationship with other modules. In this section, we will introduce this dependency definition model in detail.

First, module dependency is the dependency of a module provider on other modules. If there are modules A, B, C, and D, there are no dependencies among these four modules. Rather, provider a1 of module A depends on modules C and D. Since provider a1 of module A requires the use of the open APIs of modules C and D, a1 provider has to declare such a dependency. The dependency relationship between module instances is transitive. If provider b1 for module B is dependent on module A, logically modules C and D

---

<sup>3</sup> If SkyWalking offers only a single type of provider internally, or recommends adoption of this provider, default will be set as the module name.

must be started before module A, and module B should be started last. In essence, module dependency aims to solve the problem of module startup sequence.

More importantly, modules are not allowed to use APIs to call each other directly. In the dependency definition for Maven, it is not necessary for provider b1 of module B to directly depend on provider a1 of module A; rather, dependency is on module definition A, meaning that the service API of module A is used directly, and not the provider. Therefore, when you read the code, you will find a call pattern similar to this:

```
moduleManager.find(CoreModule.NAME).provider().getService(SourceReceiver.class);
```

This shows that it is ready to initiate a cross-module service API call, and it is looking for providers through the modularized kernel. To be clear, SkyWalking's modularized kernel does not impose restrictions on this kind of lookup API. Therefore, if the module provider does not define module dependencies in the service provider but uses this module at runtime, the service may start normally, whereas the actual operation may show irregularities. From the standpoint of design, SkyWalking does not perform module dependency tests in the running state, in order to reduce unnecessary performance consumption.

## 6.3 Module replaceability

In view of the concept of the module provider, it is obvious that modules are replaceable. Even within SkyWalking's official providers, there are already multiple module providers, such as cluster coordinator module, storage module, and configuration center module. For example, the cluster coordinator module includes the providers of almost all popular service discovery mechanisms, such as the default stand-alone mode, ZooKeeper, etcd, Consul, and Kubernetes. These providers can configure different provider names and configuration parameters through application.yml to activate different service discovery mechanisms.

For example, the configuration for the ZooKeeper cluster coordinator provider is as follows:

```
cluster:
  zookeeper:
    namespace: ${SW_NAMESPACE:""}
    hostPort:
      ${SW_CLUSTER_ZK_HOST_PORT:localhost:2181}
    #Retry Policy
    baseSleepTimeMs: ${SW_CLUSTER_ZK_SLEEP_TIME:1000} # initial
      amount of time to wait between retries
    maxRetries: ${SW_CLUSTER_ZK_MAX_RETRIES:3} # max number of times
      to retry
    # Enable ACL
```

```

enableACL: ${SW_ZK_ENABLE_ACL:false} # disable ACL in
default
schema: ${SW_ZK_SCHEMA:digest} # only support digest
schema
expression: ${SW_ZK_EXPRESSION:skywalking:skywalking}

```

The configuration for the Kubernetes cluster coordinator is as follows:

```

cluster:
  kubernetes:
    watchTimeoutSeconds: ${SW_CLUSTER_K8S_WATCH_TIMEOUT:60}

    namespace: ${SW_CLUSTER_K8S_NAMESPACE:default}
    labelSelector:
      ${SW_CLUSTER_K8S_LABEL:app=collector,release=skywalking}
    uidEnvName: ${SW_CLUSTER_K8S_UID:SKYWALKING_COLLECTOR_UID}

```

Regarding module replaceability, note that SkyWalking's modularized platform does not allow two modules to be used at the same time, otherwise an exception message of "xxx is defined as 2nd provider" will appear at startup. If the user would like to integrate two modules and form a principal-subordinate relationship or a mirror image relationship between them, the user would need to set up a new module provider for integration through hard-coding. Although this method seems to be strict, it ensures that the semantics of the module provider are clear, and prevents the abuse of modularization.

## 6.4 Module provider selector

Starting from version 7.0.0, SkyWalking provides a new way to activate the module provider. The following is an excerpt of centralized management in the module management configuration file of version 7.0.0:

```

cluster:
  selector: ${SW_CLUSTER:standalone}
  standalone:
    zookeeper:
      nameSpace: ${SW_NAMESPACE:""}
      hostPort: ${SW_CLUSTER_ZK_HOST_PORT:localhost:2181}
    kubernetes:
      watchTimeoutSeconds: ${SW_CLUSTER_K8S_WATCH_TIMEOUT:60}
      namespace: ${SW_CLUSTER_K8S_NAMESPACE:default}
      labelSelector: ${SW_CLUSTER_K8S_LABEL:app=collector,release=skywalking}

```

The selector is a newly added configuration item that can execute the required module providers through name matching. This mode is more user-friendly than the method of modifying `application.yml` in version 6.x, and can effectively avoid the irregularity message of "xxx is defined as 2nd provider" as earlier mentioned in Section 6.3. At the same time, it ensures that the full text of the module management file is always included in the mirror image, and switching by means of annotations is no longer required.

This configuration item is optional. It ensures that the old module management file can still run normally and is forward compatible.

## 6.5 New modules

In Section 6.1, we introduced five important components of module definition:

- Module definition
- Open API service list
- Module provider
- Module provider related configuration
- Service provider module dependency definition

In addition, after completing the above definition, it is necessary to activate the module and module provider through the SPI definition file. The following two files must be added and used in the `resources/META-INF/services` folder of the specified module and module provider code:

- `org.apache.skywalking.oap.server.library.module.ModuleDefine`
- `org.apache.skywalking.oap.server.library.module.ModuleProvider`

Finally, declare this module in the `application.yml` file and specify the provider for the module. In this way, when the OAP Server starts, this module will be loaded and started in the order specified in the service dependency declaration.

## 6.6 Chapter summary

This chapter is relatively short and technically difficult, but it has provided the prerequisite knowledge for understanding the OAP Server design and helps improve code literacy.

# Chapter 7:

## Observability Analysis Language system

---

Observability Analysis Language (OAL) is a proprietary scripting language designed by SkyWalking and used since version 6.0. It describes the operation mode of SkyWalking's back-end analysis platform OAP, and the types and parameters of analysis metrics. At the same time, the metrics data generated by OAL is the data source for subsequent data export and alarms. It is necessary to first understand OAL and its design principles in order to appreciate the core of the SkyWalking back-end analysis platform.

### 7.1 What is OAL

OAL is a scalable and lightweight compiled language that can be customized by users to describe the analysis process. OAL uses SkyWalking's built-in compiler to compile Java class files in runtime (starting from SkyWalking version 6.3), and uses SkyWalking's stream computing engine to load and run.

SkyWalking's stream computing engine supports the following two stream computing definition modes:

- Defined by hard-coding: Mainly for non-index type of stream computing data, such as metadata, data relationships, and other detailed data.
- Defined by OAL: Mainly used for metrics data. Statistical data aggregations for specific services, service instances or their dependencies.

Based on the two stream computing definition modes above, OAL can be classified into the following two categories:

- Disabling specific stream computing command: Used to close the stream computing process defined by hard-coding through OAL.

- Metrics computing definition command: Used to define the computing process of the metrics.

These two modes define the scalability and restriction mechanisms of stream computing both positively and negatively. This allows the expansion of computing power through code and OAL, and also offers the ability to subsequently disable the default computing included in the kernel.

## 7.2 OAL implementation principle

This section describes the implementation principle of OAL. Let's start this off with the introduction of SkyWalking's stream computing engine. SkyWalking has built in an ultra lightweight stream computing engine.

The stream computing engine accepts data sources of type `Source` (`org.apache.skywalking.oap.server.core.source.Source`). Through the specified type of `Dispatcher` (`org.apache.skywalking.oap.server.core.analysis.SourceDispatcher`), each `Source` is converted to raw data types that can be processed by the stream computing engine. These are the four raw data types currently available in SkyWalking's stream computing:

- Inventory data: Metadata, such as service name definition and Endpoint definition.
- Record data: Detailed data, such as trace and access log.
- Metrics data: Data for the metrics. Most OAL metrics generate this type of data.
- TopN data: Periodic sampling data, such as periodic collection of slow SQL.

These four types of data are respectively processed by five stream computing models:

- Registration model (`InventoryStreamProcessor`): Used for Inventory data. The registration model does not produce statistical data aggregation. However, due to the need to register or update for the first time, the merging process is necessary to ensure that the update frequency is not too high. Moreover, data of the same service may be received at a different node in the back-end cluster, which is a distributed merging process. Entities with the same ID will be merged into the same cluster instance, and the data will be updated after completing the merge with storage.
- Record model (`RecordStreamProcessor`): Used for Record data. The characteristics of detailed data is the large amount of data without the need to aggregate, such that the processing workflows are completed within each respective OAP instance. The detailed data is saved in storage using the techniques of caching, asynchronous batch processing, and stream writing.



- **Metrics model (MetricsStreamProcessor):** Used for Metrics data. This is the most typical distributed statistical process for SkyWalking metrics statistics. The two-stage (L1 and L2) aggregation in SkyWalking is used here. The L1 aggregation is used for entities with the same ID. Aggregate computing (L1) is performed in the current OAP node, and then hash routing is performed through the ID. In the distributed environment, there is a secondary collection of discrete data (L2). After undergoing aggregate computing with the data in the storage, the storage process is complete.
- **Sampling model (TopNStreamProcessor):** Used for TopN data. Sampling data is the only type of data that is not fully saved. The sampling data in the current OAP node is selected according to the sorting relationship (a user-defined sorting algorithm), where the TopN sets of data that meet the conditions are filtered out and persisted to storage every ten minutes. This computing model does not perform distributed aggregation. Instead, it reduces data samples as much as possible, and confirms the final TopN value only in the query stage. Theoretically speaking, the selected TopN data of the query should not be greater than the number of TopN samples per node to ensure the accuracy of the data.
- **Non-stream interaction model (NoneStreamingProcessor):** This is a new interaction model added to SkyWalking version 7.0.0. It mainly supports the page interactions and directly reads and writes to the database in a manner similar to the traditional CRUD operations.

The OAL focuses on the automated generation of metrics model analysis and adopts runtime compilation. The compilation process is completed during the startup state, and the Java bytecode is generated and handed over to the JVM to run. Therefore, there is no difference between the OAL in the runtime state and the hard-code defined stream computing, and the operation efficiency would not be affected.

There are three stages in the work processes of OAL:

- 1) **Syntax and lexical analysis:** This stage is completed by the user-defined Antlr parser. If you are interested in Antlr or parsing, you can find the files of `OALLexer.g4` and `OALParser.g4` at `oap-server/oap-grammar/src/main/antlr4/org/apache/skywalking/oal/rt/grammar`. These two files are descriptions for the lexical tree and grammatical tree, respectively. It is advisable to read these files in combination with the OAL syntax in Section 7.3.
- 2) **Dynamic code generation:** Javassist is used to assist in generating runtime code. Before the release of SkyWalking 6.3, the code was generated in the compiled state and packaged into a Jar package to run. However, this approach was not a container-friendly method, since the mirror image had to be repackaged every time a change is made. Starting from SkyWalking 6.3, the dynamic code technology allows the generated code to be directly introduced to the JVM and run. In the `oap-server/oal-rt/src/main/resources/code-templates` folder, you can find the generation logic for the code blocks. Note that some source codes have been optimized by the

compiler, and so they may be different from the compilation method directly supported by Javassist.

- 3) Stream computing registration: Using the internal stream computing initialization API, notify the stream computing engine of the stream computing process defined by the dynamic code.

## 7.3 OAL syntax

OAL is mainly composed of two syntaxes: metrics computing definition syntax and disable syntax.

### 7.3.1 Syntax of metrics computing definition

The grammatical format of the metrics computing definition is as follows.

```
// Declare Metrics
METRICS_NAME = from(SCOPE.(* | [FIELD][,FIELD ...]))
    [.filter(FIELD OP [INT | STRING])]
    .FUNCTION([PARAM][, PARAM ...])
```

The above sentence consists of the following parts:

- Metrics variable definition: This is the name of the logic used in subsequent queries, computations, alarms, storage, etc. This name must be unique in OAL. It is recommended to use lower case and use the short underscore "\_" as a separator.
- Keyword "from": Used to define the beginning of the sentence.
- Scope and its properties: Used to identify the computing subject. The SkyWalking OAL has a large number of built-in Scopes used to describe the remote sensing data entities being received. Each entity possesses multiple properties, which will be introduced later. All subsequent computations will be the statistical aggregate results based on this entity object.
- filter (optional): The filter determines which data is to be included into the final statistics based on the properties in Scope. At the entry point of stream analysis, 100% of the data is output into the computing stream, and each metric determines whether the data is to be filtered or fully taken into account in the computation. There may be multiple filter functions and the filtering process may undergo multiple stages, such as `service_2xx = from(Service.*).filter(responseCode >= 200).filter(responseCode < 400).cpm()`.
- Computable function "FUNCTION": Used to perform function computing for the filtered data, such as percentage, maximum, minimum and heat map calculations. Different functions have different parameters, which are to be introduced at the end of this section.

The main project of SkyWalking already contains the default OAL script, which can be found in the resource folder under the oap-server/server-starter module, such as [https://github.com/apache/skywalking/blob/v7.0.0/oap-server/server-bootstrap/src/main/resources/official\\_analysis.oal](https://github.com/apache/skywalking/blob/v7.0.0/oap-server/server-bootstrap/src/main/resources/official_analysis.oal). You can also find the file under the same name in the config folder of the binary distribution package.

### A. Definition of Scope

There are many types of Scopes, which support scalability. Here, the most important Scopes will be explained in detail. This context will help you better understand other types of Scopes as well.

Scope is directly related to the registration model. There are four basic concepts in SkyWalking: All, Service, Service Instance, and Endpoint. They are introduced as follows:

- All represents global access.
- Service is a set of programs with the same logical abstraction and generally refers to a cluster formed by instances having the same business functions.
- Service Instance is the smallest unit of service. Multiple service instances constitute a service. Under a JavaAgent use case, a service instance is a JVM process. In service mesh, a service instance is generally a Pod.
- Endpoint is a logical concept that represents the logical entity providing independent functions in a service. It is usually found in the forms of service URI, and service class name + method name + parameter type. It generally exists on all entities of the service and does not depend on any instances. Endpoints do not have unique mapping. Rather, the SkyWalking plug-in and server logic determines what kind of data can be used as an endpoint.

Based on these concepts, SkyWalking has built seven types of core Scopes. Scope is constructed according to the requirements of different entities during computing and aggregation of each Scope. Therefore, each Scope has its own independent aggregation key (except for All). In the aggregation computing and storage, the aggregation key and the timestamp project constitute the unique primary key. The timestamp is determined by the dimension of the statistical time. For example, if the service performs statistical computing for its metrics at 18:40 on January 28, 2017, then the minute-level timestamp is 201701281840, and the hour-level timestamp is 2017012818.

Tables 7-1 to 7-7 below describe the properties of each Scope.

Table 7-1 All properties

Name	Description	Aggregation key	Field Type
name	Service name of the request		String
serviceInstanceName	Service instance name of the request		String

endpoint	Endpoint name of the local request (URI, service method, etc.)		String
latency	Request time spent		int (milliseconds)
status	Request status, success / failure		bool (true indicates success)
responseCode	Return code of the HTTP request, such as 200, 302, and 40.		int
type	Local request type, such as Database, HTTP, RPC, and gRPC.		int

Table 7-2 Service properties

Name	Description	Aggregation key	Field Type
id	Service ID of the request	Yes	int
name	Service name of the request		String
serviceName	Service instance name of the request		String
endpointName	Endpoint name of the local request (URI, service method, etc.)		String
latency	Request time spent		int
status	Return code of the HTTP request, such as 200, 302, and 404.		bool (true indicates success)
type	Local request type, such as Database, HTTP, RPC, and gRPC.		enum
responseCode	Return code of the HTTP request, such as 200, 302, and 404.		int

Table 7-3 ServiceInstance properties

Name	Description	Aggregation key	Field Type
id	Service instance ID of the request	Yes	int
name	Service instance name of the request		String
serviceName	Service name of the request		String
endpointName	Endpoint name of the local request (URI, service method, etc.)		String
latency	Request time spent		int
status	Return code of the HTTP request, such as 200, 302, and 404.		bool (true indicates success)
type	Local request type, such as Database, HTTP, RPC, and gRPC.		enum
responseCode	Return code of the HTTP request, such as 200, 302, and 404.		int

Table 7-4 Endpoint properties

Name	Description	Aggregation key	Field Type
id	Endpoint ID of the request	Yes	int
name	Endpoint name of the request		String
serviceName	Service name of the request		String
serviceInstanceName	Service instance name requested locally		String
latency	Request time spent		int

status	Return code of the HTTP request, such as 200, 302, and 404.		bool (true indicates success)
type	Local request type, such as Database, HTTP, RPC, and gRPC.		enum
responseCode	Return code of the HTTP request, such as 200, 302, and 404.		int

Table 7-5 ServiceRelation properties

Name	Description	Aggregation key	Field Type
sourceServiceId	Service ID of the initiator of this request	Yes	int
sourceServiceName	Service name of the initiator of the request		String
sourceServiceInstanceName	Service instance name of the initiator of the request		String
destServiceId	Service ID of the provider of the request	Yes	int
destServiceName	Service name of the provider of the request		String
destServiceInstanceName	Service instance name of the provider of the request		String
endpoint	Endpoint name of the request		String
componentId	ID of the technical component used in the request	Yes	int
latency	Request time spent		int
status	Return code of the HTTP request, such as 200, 302, and 404.		bool (true indicates success)
type	Local request type, such as Database, HTTP, RPC, and gRPC.		enum
responseCode	Return code of the HTTP request, such as 200, 302, and 404.		int

detectPoint	Location of the detection point of local request, such as the client, server, or proxy		enum
-------------	--	--	------

Table 7-6 ServiceInstanceRelation properties

Name	Description	Aggregation key	Field Type
sourceServiceInstanceId	Service instance ID of the initiator of the request	Yes	int
sourceServiceName	Service name of the initiator of the request		String
sourceServiceInstanceName	Service instance name of the initiator of the request		String
destServiceInstanceId	Service instance ID of the provider of the request	Yes	int
destServiceName	Service name of the provider of the request		String
destServiceInstanceName	Service instance name of the provider of the request		String
endpoint	Endpoint name of the request		String
componentId	ID of the technical component used in the request	Yes	int
latency	Request time spent		int
status	Return code of the HTTP request, such as 200, 302, and 404.		bool (true indicates success)
type	Local request type, such as Database, HTTP, RPC, and gRPC.		enum
responseCode	Return code of the HTTP request, such as 200, 302, and 404.		int
detectPoint	Location of the detection point of local request, such as the client, server, or proxy		enum

Table 7-7 EndpointRelation properties

Name	Description	Aggregation key	Field Type
endpointId	Parent endpoint ID	Yes	int
endpoint	Parent endpoint name		String
childEndpointId	Child endpoint ID	Yes	int
childEndpoint	Child endpoint name		String
endpoint	Endpoint name of the request		String
componentId	ID of the technical component used in the request	Yes	int
rpcLatency	Request time spent		int
status	Return code of the HTTP request, such as 200, 302, and 404.		bool (true indicates success)
type	Local request type, such as Database, HTTP, RPC, and gRPC.		enum
responseCode	Return code of the HTTP request, such as 200, 302, and 404.		int
detectPoint	Location of the detection point of local request, such as the client, server, or proxy		enum

### B. Definition of filter

The filter function performs conditional filtering on all Scope entry data, and all filtered data is subject to the final statistical computing process.

The filter function supports the operators `==`, `>`, `>=`, `<`, and `<=`. Among them, `==` supports numbers, enumerations and strings, and other operators target numbers.



The filter function can perform multi-level operations. The relationship between multiple levels is "And". As of the publication of this book, the "or" operator is not yet supported. Since the general logic of monitoring index computing is not that complicated, complex expressions are usually not required. At the same time, users can choose to scale up the operators in order to create more complex calculation logic. For example, to calculate the number of requests per minute with an HTTP return value of 4xx, filter can be written as

```
service_instance_4xx = from(ServiceInstance.*).filter(responseCode >= 400)
    .filter(responseCode <500).cpm();
```

### C. Definition of FUNCTION

FUNCTION is used for the final computing. The following computable functions are currently included in the code:

- i) count: Counting function. Indicates the number of occurrences per unit time.
- ii) cpm: The number of calls per minute. This calculation changes with the statistical time range. If time is precise to the minute, this number indicates the average number of support in this minute. If time is precise to the day, this number indicates the number of executions on the relevant day, divided by 1,440 minutes ( $24 \times 60$ ).
- iii) doubleAvg: Calculate the average value of the double type.
- iv) longAvg: Calculate the average value of the long type.
- v) maxDouble: Calculate the maximum value in the current time period for the double type.
- vi) maxLong: Calculate the maximum value in the current time period for the long type.
- vii) p50, p75, p90, p95, and p99: The current time period, calculated by percentile (see Figure 7-1).

If a set of data is sorted from small to large and the corresponding cumulative percentage is calculated, the value of the data corresponding to a certain percentage is called the percentile. The percentile can be expressed as a set of  $n$  observed values arranged in numerical order, and the value corresponding to  $p\%$  is called the  $p$ th percentile. For example, if 99 values or 99 points are used to divide observed values arranged in order of size into 100 equal parts, the 99th value or point are called percentiles.

The five values of p50, p75, p90, p95, and p99 are displayed on the same metrics graph in SkyWalking by default, which is generally called a five-line response time graph (on the left side of the graph, the values increase from bottom to top). It is used to determine the stability of response time and the long tail effect of the response.

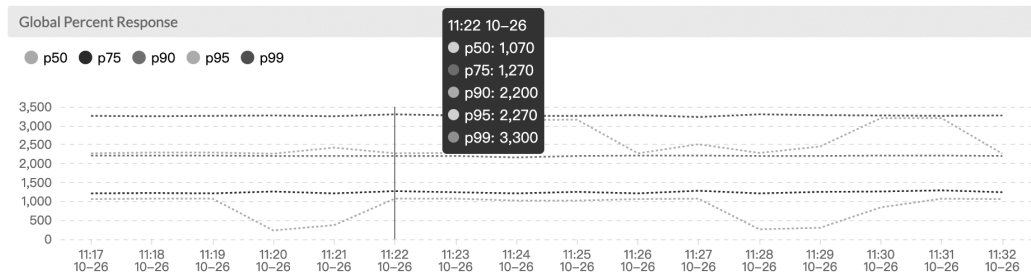


Figure 7-1 Five-line response time percentile graph

This function contains a parameter that represents the grouping of time precise to the millisecond. For example, `service_p99 = from(Service.latency).p99(10)` indicates a statistical accuracy of 10 milliseconds for the percentile. A response time of 10 to 20 milliseconds is regarded as a response time of 10 to 20 groups.

viii) `thermodynamic: heatmap` function (see Figure 7-2).

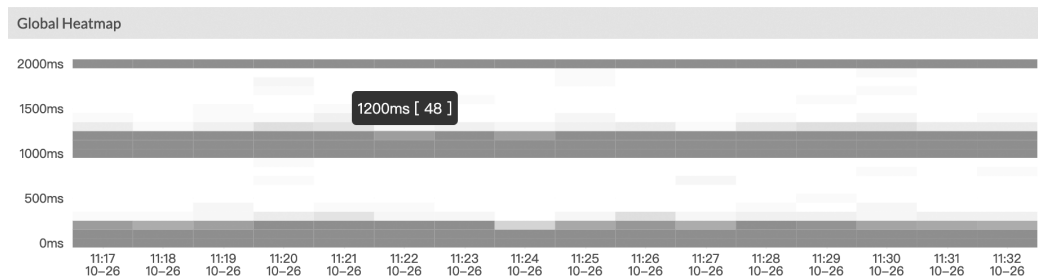


Figure 7-2 SkyWalking heat map

The heat map is generally used to display a numerical matrix. The darker the color of the node, the more requests that fall within this range of access time. Heat maps are generally used to show the distribution of requests in each time window for the corresponding response time interval. Compared to the five-line percentile graph, the heat map can more clearly show the distribution of requests.

This function contains two input parameters. The first one is the same as the percentile and represents the input parameter. The second one is the total number of groups. SkyWalking recommends setting precision to 100ms and grouping value to 20, meaning that data can be counted within groups, precise to 100ms within 0 to 2 seconds of access time.

### 7.3.2 disable syntax

The syntax of `disable (stream name)` is relatively simple. It is mainly used to disable the processing stream defined by hard-coding.

For example, ZipkinSpanRecord defines a processing stream named `zipkin_span` by hard-coding. Then, when zipkin-related processing is not required, you can use `disable(zipkin_span)` in OAL to disable the related processing stream.

What is the significance of disabling? In terms of function, not disabling the processing stream will not cause any issues with functionality. And in terms of performance, if an unnecessary processing stream is enabled, a small amount of the memory and CPU time will be occupied. This will have a limited impact on performance.

At the end of the SkyWalking OAL, we have listed all processing streams started using hard-coded methods that may need to be disabled (see below). You can choose to disable them according to your needs.

```
// Disable unnecessary hard core stream, targeting @Stream#name
//////////
// disable(segment);
// disable(endpoint_relation_server_side);
// disable(top_n_database_statement);
// disable(zipkin_span);
// disable(jaeger_span);
// disable(profile_task);
// disable(profile_task_log);
// disable(profile_task_segment_snapshot);
```

## 7.4 Chapter summary

As the core of back-end OAP metrics analysis, OAL has an important role in secondary development and customization of back-end analysis. In this chapter, the composition and usage of OAL have been explained in detail. Users who need to develop and scale up the OAL engine may improve their code literacy based on the knowledge acquired in this chapter.

## Chapter 8:

# SkyWalking OAP Server cluster communication model

---

The SkyWalking back-end OAP cluster provides several stream computing models tailored to the requirements of the aggregation of monitoring data. The functional requirements of these communication models may be better understood based on the fundamentals of the OAL system in Chapter 7. Mastering the cluster communication protocol and communication model is the theoretical basis for conducting large-scale cluster deployment, as well as cluster model customization and expansion.

Note that the OAP cluster communication model specifically refers to the distributed computing model between back-end OAP cluster nodes. SkyWalking's official agents or the agents implemented according to the SkyWalking protocol are directly connected to the back-end via HTTP or gRPC. There are two options for load balancing: to achieve a lightweight load on the client side, meaning to configure multiple addresses and choose one at random; or to adopt developed proxy solutions, such as Envoy and Nginx.

SkyWalking does not implement agent services for OAP clusters. This design is deliberate and there are technical reasons behind the decision. First, the target services being monitored are often deployed in multiple VPC subnets, while OAP clusters and business system clusters are usually deployed in an independent VPC cluster (the OAP cluster belongs to the VPC of the operations and maintenance team). Therefore, a simple mechanism of service discovery would not work well enough. Second, the OAP cluster supports switching between multiple cluster management components. The implementation on

the agent end will cause the agent package to become significantly larger, making it more difficult to update and use.

## 8.1 Stream computing

SkyWalking OAP is designed to monitor data in a distributed computing system. In the process of stream computing, there is no guarantee of data consistency, nor does the concept of transactions exist. Therefore, in extreme cases, the loss of monitoring data may occur, even though the back-end OAP cluster would be made available to the greatest possible extent.

Metrics computing is regarded as a kind of distributed aggregate computing. The current default OAP cluster stream computing is responsible for the following two tasks:

- Task 1: Data reception and analysis, and data aggregation in the current OAP node, using OAL or other aggregation modes.
- Task 2: Distributed aggregation. The data aggregated in Task 1 is routed to a specific node according to predefined routing rules for secondary collection and subsequent storage. This explains why service discovery is required between OAP nodes.

In these two tasks, there are two roles for OAP nodes: Receiver (Task 1) and Aggregator (Task 2). By default, in order to reduce the difficulty of deployment, all nodes adopt a Mixed role where Tasks 1 and 2 are performed together (see Figure 8-1). In large-scale deployment, you can choose to separate the roles according to network traffic needs, and carry out a two-level deployment mode, as shown in Figure 8-2.

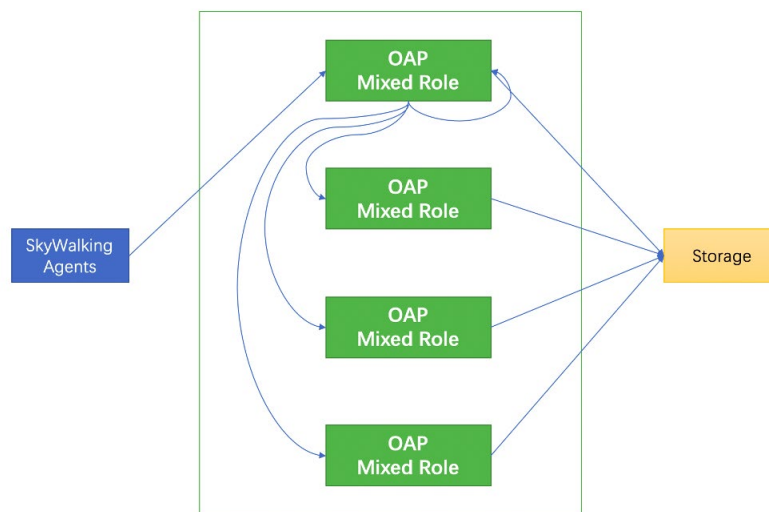


Figure 8-1 Hybrid deployment mode

In Chapter 7, four computing models in relation to OAL were mentioned: registration model, record model, metrics model, and sampling model. Among them, the record model and the sampling model are directly processed in the node for the Receiver role and then stored, while the registration model and the metrics model use distributed stream computing, meaning that the node for the Aggregator role is used.

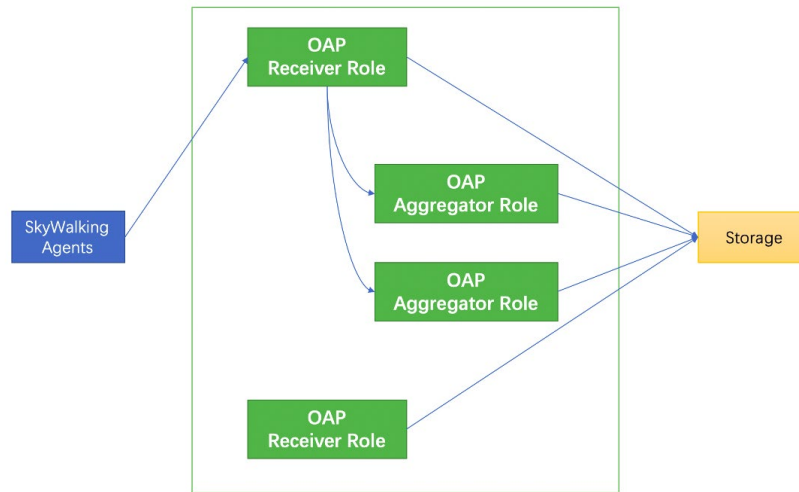


Figure 8-2 Separate deployment mode

- In order to ensure the serialization of registration and update, the registration model adopts the "select first" routing strategy. In other words, the first node is selected from the sorted list of nodes recorded in the cluster coordinator (in the order of registration addresses). Therefore, in the entire set of distributed clusters, there is only one node truly responsible for registration. Nevertheless, the registration data and updated data will be reduplicated and consolidated in the two-stage registration process, so as to minimize network traffic.
- The metrics model is a distributed computing model that consumes the most computing resources. It is the core computing model supported by the entire stream computing system. In the computing process, the OAP Server is selected with the "hash select" routing strategy and corresponding to the relevant computed entity, such as the hash value of Service ID, Endpoint ID, etc.

## 8.2 Communication protocol

The communication protocol in OAP adopts the gRPC stream mode. The protocol is defined as follows:

```

syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.apache.skywalking.oap.server.core.remote.grpc.proto";

```

```

service RemoteService {
    rpc call (stream RemoteMessage) returns (Empty) {
    }

}

message RemoteMessage {
    string nextWorkerName =
    1; RemoteData remoteData
    = 3;
}

message RemoteData {
    repeated string
    dataStrings = 1; repeated
    int64 dataLongs = 2;
    repeated double
    dataDoubles = 3; repeated
    int32 dataIntegers = 4;
    repeated IntKeyLongValuePair dataIntLongPairList = 5;
}

message
    IntKeyLongValuePair
    {
        int32 key = 1;
        int64 value = 2;
    }

message Empty {
}

```

This protocol does not contain business field names. Serialization is performed according to the data type and field definition order to reduce the transmission of non-data fields. In each function of OAL, some fields are marked with `@Column`. These fields serialize the field values into the transmission protocol according to the field type and the order of declaration. Then, they are passed to the designated OAP server according to the routing protocol for deserialization, following the same rules.

You can find the `serialize.ftl` and `deserialize.ftl` files under the `oal-rt` module. This is a script of the OAL dynamic code generator, which describes the generation process of serialization and deserialization. The serialization process is as follows:

```

public    org.apache.skywalking.oap.server.core.remote.grpc.proto.RemoteData.
          Builder serialize() {
org. apache.skywalking.oap.server.core.remote.grpc.proto.RemoteData.Builder
    remoteBuilder = org.apache.skywalking.oap.server.core.remote.grpc.
    proto.RemoteData.newBuilder();
<#list serializeFields.stringFields as field>
    remoteBuilder.addDataStrings(${field.getter}());

```

```

</#list>

<#list serializeFields.longFields as field>
    remoteBuilder.addDataLongs(${field.getter}());
</#list>

<#list serializeFields.doubleFields as field>
    remoteBuilder.addDataDoubles(${field.getter}());
</#list>

<#list serializeFields.intFields as field>
    remoteBuilder.addDataIntegers(${field.getter}());
</#list>
<#list serializeFields.intKeyLongValueHashMapFields as field>
    java.util.Iterator iterator = super.getDetailGroup()
        .values().iterator();
    while (iterator.hasNext()) {
        remoteBuilder.addDataIntLongPairList(((org.apache.skywalking
            .oap.server.core.analysis.metrics
            .IntKeyLongValue)(iterator.next()).serialize());
    }
</#list>

    return remoteBuilder;
}

```

## 8.3 Cluster coordinator

Based on the above description of the cluster mode, you will see that the functions for which the SkyWalking OAP cluster coordinator is responsible are relatively simple. It is mainly responsible for the following:

- Register the OAP Server with Aggregator role;
- On the OAP Server with Receiver role, read the list of OAP Server with Aggregator role.

To conduct the above functions, you can use existing cluster coordinators such as ZooKeeper, etcd, Kubernetes, API Server, Consul, and Nacos. The code is set out as follows. You simply have to comment out the default standalone provider, activate (uncomment, or use the selector in version 7.0.0) other cluster managers, and configure the corresponding parameters to implement the relevant cluster manager.

```

cluster:
  standalone
  :
  # Please check your ZooKeeper is 3.5+, However, it is also compatible
  # with ZooKeeper 3.4.x. Replace the ZooKeeper 3.5+
  # library the oap-libs folder with your ZooKeeper 3.4.x library.

```



```

# zookeeper:
#   namespace: ${SW_NAMESPACE:""}
#   hostPort: ${SW_CLUSTER_ZK_HOST_PORT:localhost:2181} # #Retry Policy
#   baseSleepTimeMs: ${SW_CLUSTER_ZK_SLEEP_TIME:1000} # initial amount of
#   time to wait between retries
#   maxRetries: ${SW_CLUSTER_ZK_MAX_RETRIES:3} # max number of times to
#   retry # # Enable ACL
#   enableACL: ${SW_ZK_ENABLE_ACL:false} # disable ACL in default
#   schema: ${SW_ZK_SCHEMA:digest} # only support digest schema
#   expression: ${SW_ZK_EXPRESSION:skywalking:skywalking}
#   kubernetes:
#     watchTimeoutSeconds: ${SW_CLUSTER_K8S_WATCH_TIMEOUT:60}
#     namespace: ${SW_CLUSTER_K8S_NAMESPACE:default}
#     labelSelector: ${SW_CLUSTER_K8S_LABEL:app=collector,release=skywalking}
#     uidEnvName: ${SW_CLUSTER_K8S_UID:SKYWALKING_COLLECTOR_UID}
#   consul:
#     serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
#     Consul cluster nodes, example:
#       10.0.0.1:8500,10.0.0.2:8500,10.0.0.3:8500
#     hostPort: ${SW_CLUSTER_CONSUL_HOST_PORT:localhost:8500}
#   nacos:
#     serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
#     # Nacos Configuration namespace
#     namespace: ${SW_CLUSTER_NACOS_NAMESPACE:"public"}
#     hostPort: ${SW_CLUSTER_NACOS_HOST_PORT:localhost:8848}
#   etcd:
#     serviceName: ${SW_SERVICE_NAME:"SkyWalking_OAP_Cluster"}
#     etcd cluster nodes, example:
#       10.0.0.1:2379,10.0.0.2:2379,10.0.0.3:2379
#     hostPort: ${SW_CLUSTER_ETCD_HOST_PORT:localhost:2379}

```

If necessary, you can develop more cluster managers. You simply need to implement two interfaces: `Cluster Register` and `ClusterNodesQuery` for provider registration and list query. In most cases, the default cluster coordinator is sufficient. Customized cluster coordinators are commonly used in private registries or other container management platforms (which are non-Kubernetes).

## 8.4 Chapter summary

This chapter systematically introduces the operational model of the OAP cluster, as well as the roles and communication protocols of the cluster nodes. By now, you should have had a good understanding of the operation of the OAP cluster. In the next chapter, you will be introduced to the storage model, which is the final, persistent form of data storage and update under the same mode of stream computing. The knowledge acquired in Chapter 7 provides an excellent foundation for understanding the storage model in the next chapter.

## Chapter 9:

# SkyWalking OAP Server storage models

---

This chapter provides a detailed explanation of the SkyWalking OAP storage models. First, the four types of back-end storage model structures will be introduced to provide you with an overview of various OAP storage models, which will be very helpful for secondary development and UI function customization. Then, the relationships among the four models will be introduced using the example of the receiver plugin of the OAP back-end trace. Finally, on the storage process of the metrics model, you will be introduced to the process of aggregation and analysis using the OAL script by Source to generate metric data.

## 9.1 Introduction to the model structure

In Chapter 7, we have seen that there are four storage models for the OAP back-end. The structures of these storage models will be introduced below.

### 9.1.1 Registration model

The registration model (Inventory) has to inherit the `RegisterSource` class. There are four properties for the base class `RegisterSource`:

- `sequence`: Using an auto-increment sequence ensures that the sequence of the registration model is incrementally continuous and is globally unique in the same registration model.
- `registerTime`: Used to record the time of the first record in the registration model.
- `heartbeatTime` and `lastUpdateTime`: As their names suggest, they respectively mark the statuses of the registered data as “alive.”

Currently, these are the registration models of OAP:

- **ServiceInventory:** A service name registration model that records the definition of all service names, including the `service_name` configured by the Agent, peer resources, and the registration model analysis of the call address of the RPC framework.
- **ServiceInstanceInventory:** A service instance registration model that regards each service process under the service name as the dimension.
- **NetworkAddressInventory:** A network address registration model with IP+ port as the dimension.
- **EndpointInventory:** An endpoint registration model with endpoint name as the dimension.

SkyWalking optimizes network transmission efficiency and storage computing power by designing and using registration models of various dimensions.

From the standpoint of the agent, when the agent starts, it will register and communicate with the OAP according to the process information of this instance and the service name in the agent configuration file, and obtain the service name and registration sequence information of the service instance. When the process information is reported, the registration information can be used to optimize network transmission efficiency.

From the standpoint of reporting, when the agent reports Trace Segment information, the endpoint name will be mapped to the endpoint name sequence through the endpoint registration model (EndpointInventory), and the peer resource will use the network address Registration model (NetworkAddressInventory) to convert the address of the peer resource into the network address registration model sequence value. This would greatly reduce the size of the data package when reporting the Trace Segment.

From the standpoint of streaming computing, the index property defined by the OAL script stores the sequence property in the registration model to achieve storage optimization. At the same time, information from the registration model will be cached in the form of JVM to allow fast transformation of sequence data into full registration information in the OAP.

In Table 9-1, ServiceInventory (service name registration model) is taken as an example to introduce the common properties in each registration model structure, as well as the types and descriptions of the property fields unique to the relevant model structure.

Table 9-1 Types and descriptions of service name property fields

Field Name	Field Type	Parent property	Field description
sequence	int	Yes	A serial number that is continuously incremented; an unsigned integer
registerTime	long	Yes	First registration time
heartbeatTime	String	Yes	Heartbeat time, indicates that the service is still alive
lastUpdateTime	int	Yes	Last update time
name	String		Service name Format 1: IP:port+ semicolon separator, which is the analyzed node Format 2: skywalking.agent.serviceName, which is the Agent node
isAddress	int		Where an IP address 0: service instance 1: IP address
addressId	int		Network address ID; if isAddress==1, the value of addressId is the ID mapping for NetworkAddressInventory
nodeType	int		Node type 0: Unknown node 1: Database node 2: RPCFramework 3: Http node 4: Message Queue (MQ) node 5: Cache node -1: UNRECOGNIZED node
mappingServiceId	int		Service name ID mapping

### 9.1.2 Record model

The record model (Record) requires an integration of the Record class. The base class Record has the timeBucket property, which is responsible for recording the time window where the current record is located. At present, the OAP has the following record models:

- **SegmentRecord:** The Trace Segment record model. The trace data sent over by the SkyWalking Agent is received and parsed by the skywalking-trace-receiver-plugin to obtain the Trace Segment record.
- **AlarmRecord:** The alarm record data model. Upon definition of the alarm rules in the OAP, an alarm record data model will be generated when the metrics trigger the alarm rules. For the defined rules of alarm metrics and Webhook configurations, see <https://github.com/apache/skywalking/blob/master/docs/en/setup/backend/backend-alarm.md>.
- **JaegerSpanRecord and ZipkinSpanRecord:** Implementing the plug-ins of other distributed data receiving protocols to complete the Span records of other distributed traces. You can learn about the relevant plug-in receiving process under the folder of SkyWalking's receiver plug-in at <https://github.com/apache/skywalking/tree/master/oap-server/server-receiver-plugin>.

Let's take the example of receiving and parsing the SegmentRecord sent by the SkyWalking Agent (i.e. the Trace Segment record model data) to introduce the common properties found in each record model structure, as well as the field types and descriptions of the properties unique to the relevant model structure, as shown in Table 9-2.

Table 9-2 Types and descriptions of the property fields of the Trace Segment record model

Field Name	Field Type	Parent property	Field description
timeBucket	long	Yes	Time window in seconds; the format is yyyyMMddHHmm
segmentId	String		Unique identifier for Segment
traceId	String		Distributed ID based on snowflake algorithm
serviceId	int		Service ID
serviceInstanceId	int		Service instance ID
endpointName	String		Endpoint name obtained through association of endpointId with EndpointInventor
endpointId	int		Endpoint ID

startTime	long		Trace start time
endTime	long		Trace end time
latency	int		Latency in milliseconds
isError	int		Where there is an irregularity
dataBinary	byte[]		Binary data encoded by Base64
version	int		Segment version with a compatible design

### 9.1.3 Metrics model

The metrics model (Metrics) needs to inherit the Metrics class. The metrics data is a storage model generated by the OAP through OAL script aggregation and analysis of Source data. The following is the OAL script statement:

```
service_instance_sla = from(ServiceInstance.*).percent(status == true);
```

The metrics model calculation `service_instance_sla` (service instance stability) is obtained by analyzing the percentage of the data whose stability status is successful (`status == true`) in the `ServiceInstance` Source data. In this sentence, the Metrics Name variable `service_instance_sla` defines the index name, and the Scope `ServiceInstance` used after the keyword “from” defines the logical properties `entity_id` and `service_id`. Using the operation rule of percent increases the corresponding percentage detailed properties of the `service_instance_sla` metrics model. Therefore, the metrics storage model has its index name defined by Metrics Name, and the primary key property of logic for the index structure is defined by the Source. The OAL operation rule defines the property of the detailed analysis results. (In Section 9.3, we will take this OAL script as an example to describe the relationship between the metrics storage model and OAL.)

### 9.1.4 Sampling model

The sampling model (TopN) has to inherit the TopN class. The base class TopN has these four important properties:

- **statement:** Used to describe the key information of the sampled data, such as to record the SQL description statements and Redis operation commands.
- **latency:** Used to record the latency of the sampling data. The sorting algorithm can be implemented according to latency properties to obtain sampling data with a relatively high latency in a specified period of time.
- **traceId:** Used to describe the associated distributed trace ID of the sampling data.
- **serviceId:** Used to record the service ID.

In Table 9-3 below, based on the sampling records sorted by latency TopNDatabaseStatement (currently there is only TopNDatabaseStatement in the sampling model by default), the example of DB sampling will be used to introduce the types and descriptions of common property fields of the sampling model.

Table 9-3 Types and descriptions of common property fields of the DB sampling model

Field Name	Field Type	Parent property	Field description
timeBucket	long	Yes	Time window for the purpose of query
statement	String	Yes	statement description, which can be used to store SQL statements
latency	long	Yes	Latency in milliseconds
traceId	String	Yes	Distributed trace ID
serviceId	int	Yes	Service ID

## 9.2 Connection between the storage models

To understand the relationship between models, it is necessary to first understand the overall execution process of the OAP. In terms of design, the OAP receiver end is developed for plug-ins, and the parent folder of all plug-ins can be found at `oap-server/server-receiver-plugin`. After the receiver plug-ins receive the data as specified, the OAP will use it to create a `SourceBuilder` to define `Source`. Each `Source` will construct four storage models through the specified `Dispatcher`. The relationship between these four models is as follows:

- a) In order to optimize the transmission efficiency and storage capacity of SkyWalking, other models will connect with the registration model through the ID of the registration model, and obtain detailed data of the corresponding registration model through the ID.
- b) A distinctive feature of the record model is its large data volume. In storage optimization, the ID of the registration model is used to connect with the registration model to improve storage capacity. In relation to UI item query and display, considering that there may be joint query and issues causing page data display problems, the relevant properties will be added as necessary to the record model in the process of its construction.
- c) Most OALs generate metrics models. Scope in the OAL defines Source, which defines the logical primary key of metrics details and the logical rules of operation. The data structure of the metrics model is associated with the registration model through the logical primary key, and the property value calculated by the metrics rule is stored by implementing FUNCTION in the OAL.
- d) The sampling model stores sampling data within a specified period of time based on custom sorting rules. For example, the latency storage model for latency sampling executed by DB is a storage approach that sorts Trace Segment records by the execution time of DB and selects TopN Trace Segment records within a specified time. Connection is made possible by linking up the trace IDs of the records.

Let's take the trace data receiver `skywalking-trace-receiver-plugin` at the core of OAP as an example. This example will show you the flow of data on the receiver end, from the process of receiving data, building `SourceBuilder`, defining `Source` and `Dispatcher` data, to entering the storage queue. This will help you understand the conversion of Trace Segment data into each model and the relationship between the models. Figure 9-1 shows the flow chart for the collection and storage of trace data (`skywalking-trace-receiver-plugin`).

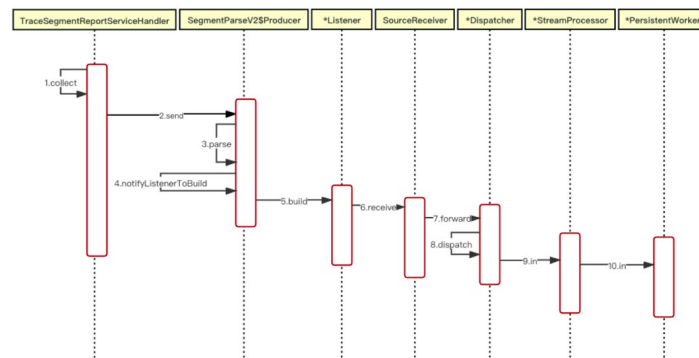


Figure 9-1 Flow chart for the collection and storage of trace data



### 1) Stage 1

The `TraceSegmentReportServiceHandler` class is responsible for collecting Trace Segment data. The `collect` method is the entry method that receives Trace Segment stream data and consumes it by calling the `send` method of the `SegmentParseV2.Producer` class.

### 2) Stages 2 to 4

Inside the `send` method of the `SegmentParseV2.Producer` class, it parses the Trace Segment stream data and uses the registered model ID in the stream data to associate this stream data with the registered model. For example, the properties related to the service or network resource definition are connected with the service name registration model and the network address registration model, and the endpoint properties are connected with the Endpoint registration model. Then, all listeners are notified to complete the creation of `Source Builder`. The DB execution latency sampling storage model implements a custom sorting algorithm through the latency property in Trace Segment, completes the sampling of the DB latency data within a period of time, connects it to the trace detail data through `TraceId`, and connects it with the registration model through `ServiceId` of the DB instance.

### 3) Stages 5 to 8

\*After the `Listener` class (all listeners that implement `SpanListener`) receives the Trace Segment data from upstream, the `Source` object being created will be called using the `dispatcher` method of the specified `Dispatcher` class to complete `Source` conversion to the storage model

The metrics model performs stream computing on the specified `Source` data through OAL and obtains the analysis results of the logical primary key of the `Source` and the stream data. The metrics model is associated with `Source` through the logical primary key defined in `Source`, such as `OAL: service_instance_sla = from(ServiceInstance.*). percent(status == true)`. The metrics model `service_instance_sla` uses the logical primary key of `Source ServiceInstance` to connect with `ServiceInstance`. The logical primary key of the `ServiceInstance` is the Sequence of the service instance registration model, which then implements the logical connection between the metrics model and the registration model.

### 4) Stages 9 to 10

In the `in` method of the `*StreamProcessor` class (the stream computing executor of four storage models), the input parameter storage model calls the specified `*PersistentWorker` to insert the storage model into the storage processor. For example, a record model `SegmentRecord` calls

RecordPersistentWorker to insert SegmentRecord into the storage processor to store the Trace Segment record data.

## 9.3 Relationship between storage model and OAL

The storage model can quickly build a metrics model through the OAL. After each plug-in on the OAP receiver end receives a message, it will build a large number of Sources through hard-coding based on the received data. Each Source will be converted to a storage model through SourceDispatcher, where the Source defined in the OAL Scope property will be calculated through FUNCTION defined by the OAL, and the metrics storage model will be converted. The properties in the storage model are the logical primary keys of Source and the FUNCTION class of properties. Here, we will illustrate the relationship between the OAL and the storage model with the service instance stability computing in the default OAL script.

The default OAL script is found in the `official_analysis.oal` file in the resource folder of the `oap-server/server-starter` project. The OAL script for the storage model computing of service instance stability is:

```
service_instance_sla = from(ServiceInstance.*).percent(status == true);
```

The metrics model `service_instance_sla` is obtained from the Source data of the computed subject `ServiceInstance`. `ServiceInstance` inherits the Source class, which describes the service instance data in the received remote sensing data entity that can be used for stream computing. The logical primary key in the service instance data would then be connected with the `ServiceInstanceInventory` (service instance registration model). The package path of `ServiceInstance` is located at `org.apache.skywalking.oap.server.core.source.ServiceInstance`, and the field types and descriptions of its important properties are shown on Table 9-4.

Table 9-4 Types and descriptions of service instance entity fields

Field name	Field type	Field description
scope	int	The unique identifier of <code>ServiceInstanceInventory</code> , which must not be repeated in all application service lists
timeBucket	long	Time window in seconds; the format is <code>yyyyMMddHHmm</code>

entityId	String	Logical primary key, i.e. ID
id	int	Service instance ID
serviceId	int	Service ID
name	String	Service instance ID name property connected with ServiceInstanceInventory Format: skywalking.agent.serviceName-pid: process ID @ machine name
serviceName	String	Service ID is the name property connected with ServiceInventory Format: service_name property in Agent configuration file
endpointName	String	Endpoint name: Endpoint name ID in the name property connected with EndpointInventory
latency	int	Latency
status	boolean	Status (true means success)
responseCode	int	Response status code: Used in combination with type
type	RequestType	Protocol type: database, HTTP, RPC, gRPC

The FUNCTION function of the OAL script is percent, and the input parameter of the function is status == true. The classpath for implementing the function percent is org.apache.skywalking.oap.server.core.analysis.metrics.PercentMetrics. For the types and descriptions of important property fields, see Table 9-5.

Table 9-5 Types and descriptions of property fields of PercentMetrics class

Field name	Field type	Field description
total	int	Total amount; used to calculate all data that has been streamed
match	long	Matched amount; stream computed data matched to the OAL script

percentage	String	Percentage; when the computable function is called, the calculation of percentage within the unit time window will be completed
timeBucket	int	Time window

In the PercentMetrics class, use the @Entrance annotation to complete the data aggregation of the percent function in the OAL script.

```
@Entrance
public final void combine(@Expression boolean isMatch) {
    if (isMatch) {
        match++;
    }
    total++;
}
```

The input parameter isMatch is status == true, which indicates whether the status property of ServiceInstance is successful. If successful, the number of matches will be increased, and the statistics of all streamed data are increased as well. On the other hand, if status is unsuccessful, the number of matches will not be increased. The computable function calculate is used to calculate the stability of the service instance. The source code of calculate is as follows:

```
@Override public void calculate() {
    percentage = (int)(match * 10000 / total);
}
```

By comparing match and total, the ratio of data whose status is successful in a time window is calculated, and the stability of each service instance is calculated. The UI display of the service instance stability metrics model is shown in Figure 9-2.

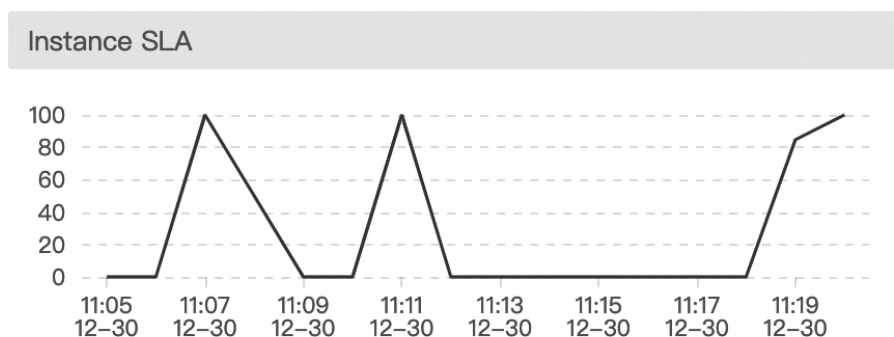


Figure 9-2 UI display of service instance stability

## 9.4 Chapter summary

The SkyWalking OAP classifies storage models into four types, namely registration model, record model, metrics model and sampling model. Taking the connection of the four storage models in the Trace Segment receiver plug-in described in this chapter and the service instance stability metrics model implemented by the OAL as the starting point, the OAL can be used for a large number of OAP Source for secondary development by users.

In Chapters 8 and 9, we've looked at the cluster communication and server storage models for the SkyWalking OAP, and you should have had a better understanding of the back-end. In the next chapter, we will begin to discuss secondary development.

## Chapter 10:

# Development of Java agent plug-ins

---

This chapter mainly introduces the development of Java agent plug-ins for the SkyWalking project, including the relevant prerequisite knowledge and development practices. In this chapter, you will get to know the developmental process of SkyWalking's Java agent plug-ins, as well as learn how to independently develop agent plug-ins.

## 10.1 Basic concepts

This section mainly introduces the basic concepts of core objects involved in the developmental process of SkyWalking's Java agent plug-ins, including Span, Trace Segment, ContextCarrier, and ContextSnapshot. After mastering them, you will find it much easier to understand the APIs related to these core objects to be introduced in Section 10.2.

### 10.1.1 Span

Span is a very important concept in a distributed tracing system. It can be alternatively understood as a method call, a program block call, an RPC call, or database access. If you would like to learn more about the concept of Span, you may refer to [the Dapper paper](#) (2010) and OpenTracing.

SkyWalking's Span concept is similar to the Dapper paper and OpenTracing, but it has been extended. Span can be broadly classified into two categories, LocalSpan and RemoteSpan, depending on whether it is a cross-thread or cross-process trace, respectively.

LocalSpan represents an ordinary Java method call and is irrelevant to cross-process. It is primarily used to record key logic code blocks in the current process, or to record trace information of asynchronous thread execution after cross-threading.

RemoteSpan can be further classified into the categories of EntrySpan and ExitSpan:

- EntrySpan represents an application service provider or server entry endpoint, such as the server entry of the Web container, the consumer of the RPC server, and the consumer of the message queue;
- ExitSpan (referred to as LeafSpan in the early versions of SkyWalking) represents an application service client or message queue producer, such as a Redis call of the Redis client, a database query from the MySQL client, a request from an RPC component, and a message produced by a message queue producer.

### 10.1.2 Trace Segment

Trace Segment is a concept unique to SkyWalking. It usually refers to the aggregation of all Spans belonging to the same operation of a thread in a language that supports multi-threading. These Spans have the same unique identifier of SegmentID. The entity class corresponding to Trace Segment is located at `org.apache.skywalking.apm.agent.core.context.trace.TraceSegment`. The important properties are as follows:

- `TraceSegmentId`: The unique identifier of this Trace Segment operation. It is generated using the snowflake algorithm to ensure global uniqueness.
- `Refs`: The upstream references of this Trace Segment. For most upstream RPC calls, `Refs` has only one element. However, in the case of a message queue or batch processing framework, the upstream may consist of multiple application services, so there will be multiple elements.
- `Spans`: For storage and belongs to the set of Spans of this Trace Segment.
- `RelatedGlobalTraces`: The Trace ID of this Trace Segment. In most cases, it only contains one element. In the case of a message queue or batch processing framework, the upstream will consist of multiple application services, thus resulting in multiple elements.
- `Ignore`: Whether to ignore. If `Ignore` is true, then this Trace Segment will not be uploaded to the back-end of SkyWalking.
- `IsSizeLimited`: The limitation on the number of Spans belonging to this Trace Segment. The initial size can be configured through the `config.agent.span_limit_per_segment` parameter. The default length is 300. If it exceeds the configured value, it will become NoopSpan when

creating a new Span. NoopSpan represents a Span implementation without any actual operations and is used to keep the expenses on memory and GC as low as possible.

- `CreateTime`: The creation time of this Trace Segment.

### 10.1.3 ContextCarrier

An important problem to be solved by distributed tracing is the connection of cross-process call traces. ContextCarrier is designed to solve this problem. Say there are two application services: client A and server B. When A calls B once, the steps of cross-process propagation are as follows:

- 1) Client A creates an empty ContextCarrier .
- 2) Create an ExitSpan through the ContextManager#createExitSpan method, or use ContextManager#inject to send in the ContextCarrier and initialize it in the process.
- 3) Use ContextCarrier.items() to place all the elements of ContextCarrier in the request information during the call, such as HTTP HEAD, Dubbo RPC framework attachments, and message header of the message queue Kafka.
- 4) ContextCarrier is transmitted to the server along with the request.
- 5) Server B receives the request with ContextCarrier and extracts all information related to ContextCarrier.
- 6) Create EntrySpan through the ContextManager#createEntrySpan method, or use ContextManager#extract to establish a distributed call connection, that is, to pair up server B and client A.

### 10.1.4 ContextSnapshot

In addition to cross-process, support is also required for cross-threading. For example, asynchronous threads (message queues in memory) are commonly found in Java. Cross-thread and cross-process are similar in that both require the propagation of context. Their only difference is that cross-thread does not require serialization. The following are the steps of cross-thread propagation:

- 1) Use the ContextManager#capture method to obtain the ContextSnapshot object.
- 2) Let the child thread access ContextSnapshot through method parameters or being carried by existing parameters.
- 3) Use ContextManager#continued in the child thread.



## 10.2 Use of core object-related API

In this section, you will be introduced to the use of the key APIs involved in the development of SkyWalking's Java agent plug-ins. This will help you master the usage of each API and learn how to use the correct APIs to complete the development of the Java agent plug-ins.

### A. *ContextCarrier#items*

In the case of cross-process tracing, *ContextCarrier#items* is used to complete the trace data management of two processes. Taking the HTTP request as an example, the following two scenarios are dealt with:

#### *Scenario*

1.

Pair up the trace information of the sending process with header and send it out through the client. The relevant code is as follows:

```
CarrierItem next = contextCarrier.items();
while (next.hasNext()) {
    next = next.next();
    httpRequest.setHeader(next.getHeadKey(),
        next.getHeadValue());
}
```

#### *Scenario 2:*

The receiving server pairs up the trace information with this receiving process by parsing the header. The relevant code is as follows:

```
CarrierItem next = contextCarrier.items();
while (next.hasNext()) {
    next = next.next();
    next.setHeadValue(request.getHeader(next.getHeadKey()));
}
```

### B. *ContextManager#createEntrySpan*

The provider of an application service or the receiverpoint of the server, such as the server entry of the Web container, the RPC server or the consumer of the message queue, must create an *EntrySpan* when called. The *ContextManager#createEntrySpan* has to be used to complete such a process. The relevant code is as follows:

```
ContextManager.createEntrySpan(operationName, contextCarrier);
```

*ContextManager#createEntrySpan* API has the following two key input parameters:

- **operationName:** Define the operation method name of this EntrySpan, such as the request path of the HTTP interface. Note that the operationName must be exhaustive. For example, when the RESTful interface matches `/path/{id}`, the true value of `id` must not be recorded. This is because when SkyWalking reports data, operationName will be mapped in the local dictionary table and use the mapping value of operationName for transmission, both in order to reduce the length of operationName and for trace message transmission performance considerations. Therefore, it is necessary to ensure that the operationName is exhaustive, otherwise the dictionary table may become too large.
- **contextCarrier:** In order to allow cross-process tracing, the upstream trace information must be paired up with this trace through `ContextCarrier #items`. For more information on the use of the relevant API, refer to the usage of `ContextCarrier#items`.

### C. *ContextManager#extract*

In a message queue or batch processing framework, because the upstream consists of multiple application services, there are multiple elements. In this case, `ContextManager#extract` is used to pair up the trace information of multiple upstream applications with the current trace. The following code shows a situation where the message queue Kafka framework bundles up multiple upstream application traces together in batch processing:

```
for (ConsumerRecord<?, ?> record : consumerRecords) {
    ContextCarrier contextCarrier = new ContextCarrier();
    CarrierItem next = contextCarrier.items();
    while (next.hasNext()) {
        next = next.next();
        Iterator<Header> iterator = record.headers().headers
            (next.getHeadKey()).iterator();
        if (iterator.hasNext()) {
            next.setHeadValue(new String(iterator.next().value()));
        }
    }
    ContextManager.extract(contextCarrier);
}
```

### D. *ContextManager#createExitSpan*

On the sending endpoint of an application service client or message queue producer, such as an one-time memory access of a Redis client, one-time database query of a MySQL client, or one-time request of an

RPC component, when a request occurs, the client process must use `ContextManager#createExitSpan` to create `ExitSpan`. The relevant code is as follows:

```
ContextManager.createExitSpan(operationName, contextCarrier, peer);
```

The `ContextManager#createExitSpan` API has the following three key input parameters:

- `operationName`: Define the operation method name of this `ExitSpan`. Note that the `operationName` must be exhaustive, and the details are consistent with the input parameter `operationName` of `ContextManager#createEntrySpan`.
- `contextCarrier`: For the purpose of cross-process tracing, the trace information of this thread is paired up with the header. For more information on the use of the relevant API, refer to the usage of `ContextCarrier#items`.
- `peer`: Downstream address in `ip:port` format. If the downstream system cannot download the agent, such as Redis, MySQL, and other resources, all downstream addresses must be written into the `peer` parameter in the `ip:port;ip:port` format.

#### *E. ContextManager#inject*

`ContextManager#inject` is a less commonly used API. You can use this API to control the call timing for `ContextManager#inject` in the `ExitSpan` method.

#### *F. ContextManager#createLocalSpan*

For tracing of key local logic code blocks in the process, or to enable tracing of asynchronous thread execution, use `ContextManager#createLocalSpan` to create `LocalSpan`. The relevant code is as follows:

```
ContextManager.createLocalSpan(operationName)
```

`operationName` is a key input parameter for `ContextManager#createLocalSpan` API. It defines the name of the operation method of this `LocalSpan`. Note that the `operationName` must be exhaustive, and the details are consistent with the input parameter `operationName` of `ContextManager#createEntrySpan`.

#### *G. ContextManager#capture*

When performing tracing across processes, we need to pass along the trace snapshot of the parent thread. `ContextManager#capture` is used to obtain the snapshot. Snapshots are usually passed to child threads by parameter-modifying data or methods. The code for obtaining a snapshot of the current thread is as follows:

```
ContextSnapshot snapshot = ContextManager.capture()
```

#### *H. ContextManager#continued*

`ContextManager#continued` is used to connect the parent process snapshot with the child thread. The relevant code is as follows:

```
ContextManager.continued(contextSnapshot);
```

#### *I. ContextManager#stopSpan*

Whichever type of Span is used, the stop method must be called to end the tracing of the Span. The relevant code is as follows:

```
ContextManager.stopSpan(span);
```

#### *J. ContextManager#isActive*

If you are not sure whether there is an unfinished Span in the current thread, calling the stop method abruptly may cause an agent irregularity. This API can be used to determine whether there is an active Span in the current thread. The relevant code is as follows:

```
ContextManager.isActive();
```

#### *K. ContextManager#activeSpan*

This API is used to obtain the active Span in the current process. The relevant code is as follows:

```
AbstractSpan activeSpan = ContextManager.activeSpan();
```

#### *L. AbstractSpan#setComponent*

Component is an important property in Span. The agent components supported by SkyWalking are defined in the files `org.apache.skywalking.apm.network.trace.component.ComponentsDefine` and `/oap-server/server-core/src/test/resources/component-libraries.yml`. Assigning Component to Span requires the use of the

`AbstractSpan#setComponent` API. For example, the code for assigning Tomcat component to Span is as follows:

```
span.setComponent(ComponentsDefine.TOMCAT);
```

#### M. `AbstractSpan#setLayer`

Layer can be understood as a simple display of Span, which is of great significance for page display. Currently there are five types of Layer in SkyWalking, namely:

- `SpanLayer.Database`, which represents the database;
- `SpanLayer.RPCFramework`, which represents the RPC framework;
- `SpanLayer.Http`, which represents the HTTP request received by the Web server;
- `SpanLayer.MQ`, which represents the message queue; and
- `SpanLayer.Cache`, which represents the cache database.

For example, the current Span component is a database type component. We must use the following API to mark this Span:

```
span.setLayer(SpanLayer.Database);  
SpanLayer.asDB(span); // recommend
```

#### N. `AbstractSpan#tag`

In some cases, the corresponding tag has to be added in the current Span. `AbstractSpan#tag` can be used to record Span. The meaning of the keys of the following tags are explained:

- `Tags.URL`: The key is `url`, which records the URL of the incoming request.
- `Tags.STATUS_CODE`: The key is `status_code`, which records the HTTP status code of the response. It is usually used to record a status code greater than or equal to 400.
- `Tags.DB_TYPE`: The key is `db.type`, which records the database type, such as SQL, Redis, and Cassandra.
- `Tags.DB_INSTANCE`: The key is `db.instance`, which records the name of the database instance.
- `Tags.DB_STATEMENT`: The key is `db.statement`, which records SQL statements for database access.
- `Tags.DB_BIND_VARIABLES`: The key is `db.bind_vars`, which records the bind variables of the SQL statement.
- `Tags.MQ_QUEUE`: The key is `mq.queue`, which records the queue name of the message middleware.
- `Tags.MQ_BROKER`: The key is `mq.broker`, which records the broker address of the message middleware.

- `Tags.MQ_TOPIC`: The key is `mq.topic`, which records the topic name of the message middleware.
- `Tags.HTTP.METHOD`: The key is `http.method`, which records the HTTP request method.

The following are two ways to record tags:

```
Tags.URL.set(span, request.getURI().toString()); // recommend
span.tag("key", "value");
```

#### *O. AbstractSpan#log*

*log is a common log, usually with multiple fields at a certain point in time. It typically records the stack information in the Span when an error occurs in the current method, or records a common log with multiple fields at a point in time. API usage is as follows:*

```
span.log(System.currentTimeMillis(), even)
ContextManager.activeSpan().errorOccurred().log(t);
```

#### *P. AbstractSpan#errorOccurred*

log usually records the stack information in the Span when an error occurs in the current method, or a common log with multiple fields at a certain point in time. The usage of API is as follows:

```
ContextManager.activeSpan().errorOccurred();
```

#### *Q. AbstractSpan#setPeer*

peer represents peer resource in the `ip:port` format. If the downstream system cannot download the agent, such as Redis, MySQL and other resources, all downstream addresses must be written into the peer parameter in the `ip:port;ip:port` format. The relevant API is as follows:

```
span.setPeer("ip port");
```

#### *R. AsyncSpan*

In some cases, this API must be used in order to set Span properties such as tag, log, or end time in another thread. The steps are as follows:

- 1) Call `AsyncSpan#prepareForAsync` in the original context.
- 2) Propagate the Span to the other threads and complete the records of the corresponding properties.
- 3) When all operations are ready, you can call `#asyncFinish` in any thread to end the call.

- 4) When the AsyncSpan#prepareForAsync of all Spans is complete, the tracing context will be finished and sent back to the back-end service (judging by the number of API executions).

## 10.3 Project structure of agent plug-in

This section mainly introduces the project structure of the SkyWalking agent plug-in. You will get to know the role of each package in the agent plug-in provider for the source code of SkyWalking agent plug-in, and how each class under the package is designed. This section is the foundation for understanding the live development of agent plug-in in the next section.

### 10.3.1 Introduction to the project structure

The agent plug-in structure mainly consists of two parts: the definition of the interception form and an interceptor implementing the interception form. As an illustration, the project structure of the Apache Dubbo plug-in is shown in Figure 10-1.

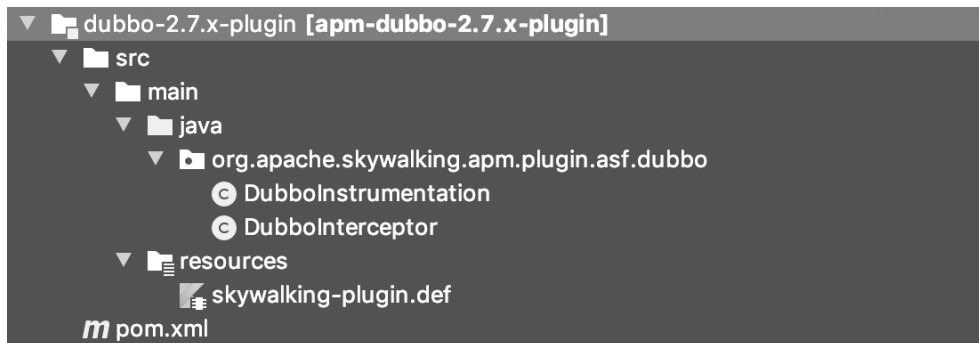


Figure 10-1 Apache Dubbo plug-in project structure

The DubboInstrumentation class defines the interception form of the Dubbo agent plug-in. In other agent plug-ins, using the suffix \*Instrumentation to represent the interception form of an agent plug-in class name is recommended. The structure of the agent plug-in interception method is mainly composed of two parts: matcher definition and interceptor implementation. When the method of the plug-in is successfully matched by the matcher, the interceptor of the agent plug-in will be executed.

The DubboInterceptor class is an interceptor that implements Dubbo. In the DubboInstrumentation class, it is implemented by the interceptor used to define the interception point. Similarly, using the suffix \*Interceptor to represent the interceptor class name that implements the interception form is recommended. In the interceptor, users can fulfill their requirements before, during and after the interception point is executed according to their needs. In order for SkyWalking Agent to discover and

load the agent plug-in, it is necessary to create the `skywalking-plugin.def` file in the resource folder to define the correspondence between the agent and the interception form, in the form of the data structure of a dictionary table. The key value is the plug-in name, which must be globally unique. Conventionally speaking, the recommended name is target component + version number, and the value is the intercepted class path + class name.

The source code of Apache Dubbo `skywalking-plugin.def` is:

```
dubbo=org.apache.skywalking.apm.plugin.asf.dubbo.DubboInstrumentation
```

### 10.3.2 Defining the interception form

Any form of interception is possible for the SkyWalking Agent, since it uses bytecode tools for implementation. For brevity, we will only introduce the definitions of the enhanced class interception and the two most commonly used interception methods.

#### *A. Definition of enhanced class interception*

To define enhanced class interception, you need to inherit `AbstractClassEnhancePluginDefine` and override the method. The code is as follows:

```
protected abstract ClassMatch enhanceClass();
```

Return the object `ClassMatch`. There are many matching methods for interception classes. Users can also develop custom matching methods according to their needs. The following describes the implementation of three officially implemented matching methods.

- `byName`: Implement matching of interception classes through class path + class name. Avoid using `*.class`. `getName()` to obtain class path + class name, since there will be potential `ClassLoader` problems.
- `byClassAnnotationMatch`: Implement interception class matching through class annotations. Note that annotations inherited from the parent class are not supported.
- `byHierarchyMatch`: Implement matching of interception classes through the parent class or interface. Note that if there is a multi-layer inheritance relationship between interfaces, abstract classes, and classes, multiple interceptions may occur. Therefore, it is not recommended to use `byHierarchyMatch` unless necessary. Otherwise, unexpected situations may occur.

Sample source code for enhanced class interception definition:

```
@Override
```



```
protected ClassMatch enhanceClassName() {
    return byName("org.apache.dubbo.monitor.support.MonitorFilter");
}
```

### *B. Interception form of the constructor method*

To define the interception of the constructor method, you need to inherit the base class `AbstractClassEnhancePluginDefine` and override the method. The code is as follows:

```
public abstract ConstructorInterceptPoint[] getConstructorsInterceptPoints();
```

For each `ConstructorInterceptPoint` element of the return value `ConstructorInterceptPoint[]`, two methods need to be defined: (a) `getConstructorMatcher`, which is the matcher of the constructor method; and (b) `getConstructorInterceptor`, which is the agent plug-in interceptor of the constructor method. By defining these two methods, the objects of the `ConstructorInterceptPoint` element can be created. When the constructor method of the enhanced class is called and the constructor method is completely matched by the matcher, the agent plug-in interceptor of this constructor method is executed.

### *C. The interception form of the instance method*

To define the interception of the instance method, you need to inherit `ClassInstanceMethodsEnhancePluginDefine` and rewrite the method. The code is as follows:

```
public abstract InstanceMethodsInterceptPoint[] getInstanceMethodsInterceptPoints();
```

For each `InstanceMethodsInterceptPoint` element of the return value `InstanceMethodsInterceptPoint[]`, three methods need to be defined: (a) `getMethodsMatcher`, which is the matcher of instance method; (b) the `getMethodsInterceptor`, which is the interceptor of the instance method agent plug-in; and (c) `isOverrideArgs`, that is, whether to override the parameters. If you want to rewrite the objects of the parameters, you need to change the return value of the `isOverrideArgs` method to true. The interception form of static method is basically the same as the interception form of instance method. Simply inherit `ClassStaticMethodsEnhancePluginDefine` and rewrite corresponding to the abstract method. Here is the sample source code of the interception form of the instance method:

```
@Override
public InstanceMethodsInterceptPoint[]
    getInstanceMethodsInterceptPoints() {return new
    InstanceMethodsInterceptPoint[] {
        new InstanceMethodsInterceptPoint() {
```

```

        //The instance method intercepts the matcher
        ElementMatcher<MethodDescription> getMethodsMatcher();
        // Corresponding interception class
        String getMethodsInterceptor();
        // Change reference parameters in the interceptor
        boolean isOverrideArgs();

    }

};

}

```

### 10.3.3 Interceptors implementing the interceptor form

The interceptor class of the agent plug-in allows developers to perform non-intrusive interception of methods before, during and after the execution is irregular, and complete the trace development by calling the SkyWalking Agent core API. The source code of the instance method interceptor base class is as follows:

```

public interface InstanceMethodsAroundInterceptor {

    // Before method is executed
    void beforeMethod(EnhancedInstance objInst, Method method,
        Object[] allArguments, Class<?>[] argumentsTypes,
        MethodInterceptorResult result) throws Throwable;

    //After the method is executed
    Object afterMethod(EnhancedInstance objInst, Method method,
        Object[] allArguments, Class<?>[] argumentsTypes, Object
        ret) throws Throwable;

    // Exception for method execution
    void handleMethodException(EnhancedInstance objInst, Method method, Object[]
        allArguments,
        Class<?>[] argumentsTypes,
        Throwable t);
}

```

Note that for the `beforeMethod` method, if you want to modify the input parameters, you must change the return value of the `isOverrideArgs` method to `true` when defining the interception form, otherwise the modified parameters will not take effect.

## 10.4 Live development of agent plug-in

In this section, we will describe the implementation process of the SkyWalking plug-in. More specifically, we will introduce the development of the cross-process plug-in Dubbo and the cross-thread plug-in Spring @Async. Before developing a plug-in, you must first understand the call relationship in the project.

Since the SkyWalking Agent focuses on the topology of nodes and the connection of traces, note the following points: call traces, cross-thread connection points, information aggregation of peer cluster resources. After this section, you will learn how to find the best starting point for event tracing, and master the developmental process of cross-process and cross-thread plug-ins.

The design of each plug-in will be introduced in three parts: introduction to the framework, definition of interception, and implementation of interceptors.

### 10.4.1 Design of agent plug-in

After you have mastered the framework of the enhanced plug-in and the use of the API provided by the agent, you may begin designing the plug-in. As a starting point, you can begin with the following three aspects and design the plug-in from the standpoint of tracing.

#### *A. Framework interceptor (filter)*

Frameworks that implement protocol-oriented cross-process calls generally extend the implementation of interceptors or filters to users. The interceptor design of the framework interceptor (filter) is usually used to scale up the execution process of the business logic without destroying such logic. Such a design allows the implementation of tracing in a non-intrusive manner. Through the enhancement of the interceptor (filter), it implements tracing and monitoring when the framework receives or sends traffic. Using the Apache Dubbo RPC framework as an example, the interceptor of Apache Dubbo intercepts the remote method execution of RPC when the consumer and provider send the call process. After the interception, the call information then enters into the code block implemented by the interceptor in the form of parameters. Therefore, plug-in developers can further design the plug-in by starting with the interceptor design and source code exposed by the framework.

#### *B. Core execution method*

For any framework, there is always one or more core methods that could handle all execution processes. It is recommended that you find a suitable core execution method from the stack frame analysis of thread execution or the source code analysis of the web search framework to be used as the interception method in the design agent. Using the Apache Kafka message queue framework as an example, on the producer client end, by looking at the stack frame executed by the Debug thread, we can conclude that the Kafka producer sends messages through the method of `org.apache.kafka.clients.producer.KafkaProducer#doSend(ProducerRecord<K, V> record, Callback callback)`. By intercepting and enhancing this method, you can obtain each message being sent. The binding of trace data may be completed by extending the transmission-oriented properties of the ContextCarrier object to these

messages. The execution status of each message sent by the producer of the Apache Kafka message queue may be obtained, so as to record properties such as the start and end time of the `doSend` method, and whether there are irregularities.

### *C. Trace data binding*

Binding is required for trace data occurring across threads and processes. Otherwise, the traces may be broken.

For cross-thread cases, SkyWalking Agent offers the alternative methods of cross-thread trace binding with the API of `ContextSnapshot` or `Async Span`. If you can find the exact location of asynchronous execution threads, the `ContextSnapshot` API is recommended. If there is asynchronous threading in a responsive framework, the `Async Span` API is recommended.

For cross-process cases, you may first examine the transmission protocol used by the framework to see whether there are properties that can be used for trace data binding, such as transmission-oriented message header, HTTP header and attachment of the Apache Dubbo transmission object. Even if the transmission structure of the framework does not have these properties, you can rewrite the transmission structure to bind the trace data to the transmission structure. Pay special attention to the compatibility of the consumer when it is not connected with the agent.

Based on the discussion above on the implementation of tracing, we have analyzed and learned about the framework. By now, you should already know which forms of interception and trace data binding are used to design the plug-in. In the next two subsections, we will get to know how cross-process and cross-thread plug-ins are designed by going through the Apache Dubbo RPC framework and Spring `@Async` framework.

## 10.4.2 Apache Dubbo agent plug-in

### *A. Introduction to the framework*

Apache Dubbo is the most popular RPC framework in China. This section mainly introduces the call trace of Apache Dubbo from the standpoint of tracing. Based on the characteristics of the call trace, we can design interception points, interceptor and trace data binding, and implement the plug-in. The Apache Dubbo call trace is shown in Figure 10-2.

The lower part is the consumer call trace, and the upper part is the provider call trace. Both the call traces of the provider and consumer of Dubbo service pass through the `Filter` layer. This is also the interceptor

(filter) mentioned in the previous section for users to implement non-intrusive business extensions. In the Dubbo framework, MonitorFilter is used by default to collect trace data and to send it to the monitoring center. Therefore, Dubbo's agent plug-in enhances the MonitorFilter class, intercepts the core method invoke in the MonitorFilter class, implements binding and transmission of trace information and traffic in the interceptor class of plug-in as well as the concatenation of provider and consumer traces.

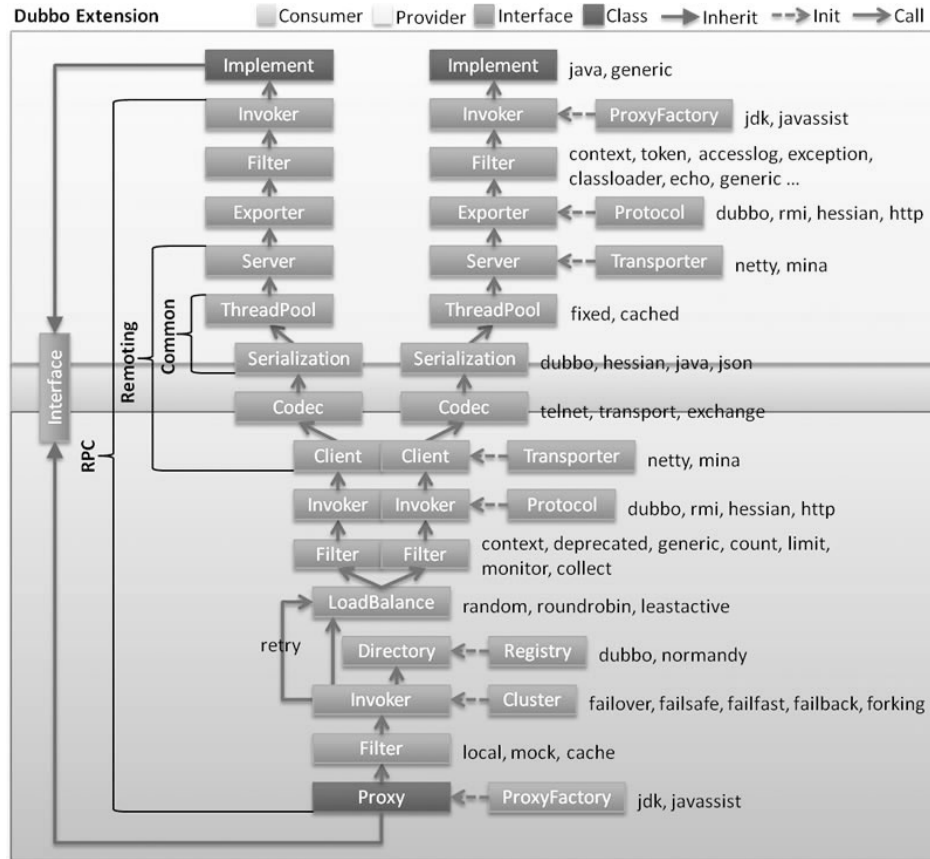


Figure 10-2 Apache Dubbo call trace

### B. Definition of interception

Based on the introduction of the Dubbo call trace of the RPC framework from the standpoint of tracing in the previous section, the agent plug-in has to enhance the `org.apache.dubbo.monitor.support.MonitorFilter#invoke` method and add an interceptor for the core method invoke. The definition of interception depends on whether the method is a static method or an instance method. In binding trace information and traffic, it should first be determined whether, in addition to the business request, there is other information being passed in the traffic. From the `MonitorFilter#invoke` source code, we can see that the `invoke` method is an instance method. The Dubbo traffic request has the

attachments property for users to add additional information to the traffic, so the trace information and traffic can be bound by such property for the purpose of spreading trace information across processes. The source code of the interception definition for Apache Dubbo is as follows:

```
public class DubboInstrumentation extends ClassInstanceMethodsEnhancePluginDefinition {
    @Override
    protected ClassMatch enhanceClass() {
        // Enhanced instance class
        return NameMatch.byName("org.apache.dubbo.monitor
            .support.MonitorFilter");
    }

    @Override
    public ConstructorInterceptPoint[] getConstructorsInterceptPoints() {
        // Unenhanced constructor method
        return null;
    }

    @Override
    public InstanceMethodsInterceptPoint[] getInstanceMethodsInterceptPoints()
    {
        return new InstanceMethodsInterceptPoint[] {
            new InstanceMethodsInterceptPoint() {
                // Enhanced instance class method matcher
                @Override
                public ElementMatcher<MethodDescription> getMethodsMatcher() {
                    return named("invoke");
                }

                // Method interception class
                @Override
                public String getMethodsInterceptor() {
                    return "org.apache.skywalking.apm.plugin.asf
                        .dubbo.DubboInterceptor";
                }

                // Unchanged reference parameter
                @Override
                public boolean isOverrideArgs() {
                    return false;
                }
            }
        };
    }
}
```

### C. Definition of interceptor

When the Dubbo service call trace passes through the `MonitorFilter#invoke` method, it will enter the `DubboInterceptor` of the interceptor class of agent plug-in. For Dubbo service consumers, it is necessary to define `ExitSpan`. Specifically, the operation involves calling the API of the `createExitSpan` and `stopSpan` methods, binding the trace information to the `Attachment` property of the Dubbo RPC Context object across methods, and passing such information to the Dubbo service provider. The `EntrySpan` has to be defined for Dubbo service providers. The operation requires calling the API of the `createEntrySpan` and `stopSpan` methods. Then, when the `createEntrySpan` method is called, the trace information sent by the consumer is parsed and bound to the trace information of the current thread, such that the consumer and provider traces are concatenated. The source code of the `MonitorFilter#invoke` instance method is as follows:

```
public class DubboInterceptor implements InstanceMethodsAroundInterceptor {

    @Override
    public void beforeMethod(EnhancedInstance objInst, Method method,
        Object[] allArguments, Class<?>[] argumentsTypes,
        MethodInterceptResult result) throws Throwable {
        Invoker invoker = (Invoker)allArguments[0];
        Invocation invocation =
            (Invocation)allArguments[1]; RpcContext rpcContext
            = RpcContext.getContext(); boolean isConsumer =
            rpcContext.isConsumerSide(); URL requestURL =
            invoker.getUrl();

        AbstractSpan span;
        final String host = requestURL.getHost();
        final int port = requestURL.getPort();
        // Determine whether it is an upstream consumer
        if (isConsumer) {
            final ContextCarrier contextCarrier = new ContextCarrier();
            // Create ExitSpan
            span = ContextManager.createExitSpan(operationName,
                contextCarrier, host + ":" + port);
            CarrierItem next = contextCarrier.items();
            // Place the trace information of the upstream consumer in the attachment and pass it
            // to the downstream producer
            while (next.hasNext())
                { next =
                  next.next();
                  rpcContext.getAttachments().put(next.getHeadKey(),
                      next.getHeadValue());
                }
        } else {
```

```

        // Place the trace information in the attachment into this trace
        ContextCarrier contextCarrier = new ContextCarrier();
        CarrierItem next = contextCarrier.items();
        while (next.hasNext())
        {
            next = next.next();
            next.setHeadValue(rpcContext.getAttachment(next.getHeadKey()));
        }
        // Create EntrySpan
        span = ContextManager.createEntrySpan(operationName, contextCarrier);
    }
    // Save the Span tag into the url
    Tags.URL.set (span, url);
    // Span is defined as component of the Dubbo class
    span.setComponent(ComponentsDefine.DUBBO);
    // Set Layer of Span as RPC framework
    SpanLayer.asRPCFramework(span);
}

@Override
public Object afterMethod(EnhancedInstance objInst,
    Method method, Object[] allArguments, Class<?>[] argumentsTypes,
    Object ret) throws Throwable {
    Result result = (Result)ret;
    if (result != null && result.getException() != null) {
        ContextManager.activeSpan().errorOccurred().log(result.getException());
    }
    // Stop Span
    ContextManager.stopSpan
    (); return ret;
}
}

```

To load the agent plug-in, the Dubbo interception has to be defined in the `skywalking-plugin.def` definition file of the SkyWalking Agent plug-in. The source code is as follows:

```
dubbo=org.apache.skywalking.apm.plugin.asf.dubbo.DubboInstrumentation
```

The development of the Dubbo agent plug-in is now complete.

### 10.4.3 Spring @Async agent plug-in

#### A. Introduction to the framework

Spring is the industry standard for domestic Java back-end service development. The asynchronous thread component `@Async` mainly implements the asynchronous decoupling of the main business process



and the thread of additional business logic. For example, after the generation of order, an SMS is asynchronously sent with Spring @Async. In projects using the Spring framework, the annotation of Spring @Async on a method indicates that it should run on a separate thread. The return type of this method will become CompletableFuture<T>, which is a requirement for asynchronous implementation. Figure 10-3 shows the Spring @Async asynchronous component class.

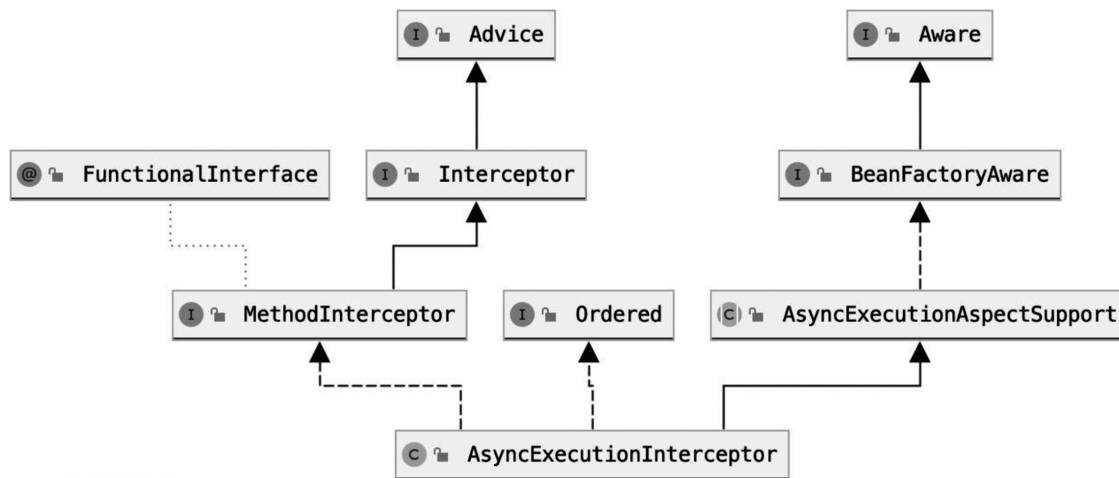


Figure 10-3 Spring @Async asynchronous component class

From this figure, we can see that the AsyncExecutionInterceptor implements the MethodInterceptor. The method being called is intercepted by using the entry point in the AOP, and the implementation method is executed by the asynchronous thread. The source code of the asynchronous thread execution method is as follows:

```

@Nullable
protected Object doSubmit(Callable<Object> task, AsyncTaskExecutor executor,
    Class<?> returnType) {
    if (CompletableFuture.class.isAssignableFrom(returnType)) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                return task.call();
            } catch (Throwable var2) {
                throw new CompletionException(var2);
            }
        }, executor);
    } else if (ListenableFuture.class.isAssignableFrom(returnType)) {
        return ((AsyncListenableTaskExecutor)executor).submitListenable(task);
    } else if (Future.class.isAssignableFrom(returnType)) {
        return executor.submit(task);
    } else {

```

```

        executor.submit(task);
        return null;
    }
}

```

The invoke method of the AsyncExecutionInterceptor constructs the method annotated with @Async as a task, which is then handed over to the thread pool for execution through the doSubmit method. This is a crucial step in cross-threading. From the perspective of tracing, when calling the doSubmit method, the ContextSnapshot of the parent thread (trace snapshot) must be bound to the task parameter, and the execution method of the task class must be called again. Then, when the child thread executes the method call of the task parameter, the LocalSpan is created, and the trace information of the parent and child threads is bound together.

### *B. Definition of interception*

Based on the Spring @Async component of the asynchronous framework from the perspective of tracing introduced in the previous section, the agent plug-in must enhance the org.springframework.aop.interceptor.AsyncExecutionAspectSupport#doSubmit method and add an interceptor for the core method doSubmit. The definition of interception depends on whether the method is a static method or an instance method, and the binding of trace information and thread requires the extension of the corresponding task object. From the source code of AsyncExecutionAspectSupport#doSubmit, we can see that the doSubmit method is an instance method. The first parameter of the method is Callable<Object> task, which is an asynchronous task object that must be extended. Therefore, we need to rewrite the Callable<Object> task in order to deliver the ContextSnapshot (trace snapshot) in asynchronous threads. The source code of the interception definition of the Spring @Async component is as follows:

```

public class AsyncExecutionInterceptorInstrumentation extends ClassInstanceMethodsEnhancePluginDefine
{
    @Override
    public ConstructorInterceptPoint[] getConstructorsInterceptPoints() {
        // Unenhanced constructor method
        return new ConstructorInterceptPoint[0];
    }

    @Override
    public InstanceMethodsInterceptPoint[] getInstanceMethodsInterceptPoints() {
        return new InstanceMethodsInterceptPoint[]{
            new InstanceMethodsInterceptPoint() {
                @Override
                public ElementMatcher<MethodDescription> getMethodsMatcher() {
                    // Enhanced instance class method matcher, and the first parameter is Callable

```

```

        return named("doSubmit").and(takesArgumentWithType(0,
            "java.util.concurrent.Callable"));
    }

    @Override

    public String getMethodsInterceptor() {
        // Method interception class
        return "org.apache.skywalking.apm.plugin.spring.async.DoSubmitMethodInterceptor";
    }

    @Override
    // Enable parameter modification
    public boolean isOverrideArgs() {
        return true;
    }
}

};

}

@Override
public ClassMatch enhanceClass() {
    // Enhanced instance class
    return byName("org.springframework.aop.interceptor
        .AsyncExecution AspectSupport");
}
}

```

### C. Definition of the interceptor

When the main thread calls a method marked by `@Async`, the method will be encapsulated into a task object, and the task object will be executed by the asynchronous thread pool in the `AsyncExecutionAspectSupport#doSubmit` method. When the task object passes through the `doSubmit` method, it will enter the interceptor `DoSubmitMethodInterceptor` of the agent plug-in interception. At this point, you need to rewrite the task object in the interceptor `DoSubmitMethodInterceptor`. First, add the `ContextSnapshot` (trace snapshot) property to the task object and rewrite the execution method of the task object in the child thread. The execution method is used to complete the creation of `LocalSpan` and to bind the traces with the parent thread through the `ContextSnapshot` (trace snapshot) property.

Rewrite the source code of the `SWCallable` object of `Callable<Object>` task as follows:

```

public class SWCallable<V> implements Callable<V> {

    private static final String OPERATION_NAME = "SpringAsync";

    private Callable<V> callable;

```

```

// Trace snapshot property of the parent thread ContextSnapshot (Trace snapshot)
private ContextSnapshot snapshot;

SWCallable(Callable<V> callable, ContextSnapshot snapshot) {
    this.callable = callable;
    this.snapshot = snapshot;
}

@Override
public V call() throws Exception {
    // Rewrite the execution method of the child thread and complete the creation of LocalSpan
    AbstractSpan span = ContextManager.createLocalSpan(SWCallable
        .OPERATION_NAME);
    span.setComponent(ComponentsDefine.SPRING_ASYNC); try {
        // Trace binding with parent thread through ContextSnapshot (Trace snapshot) property
        ContextManager.continued(snapshot);
        return callable.call();
    } catch (Exception e) {
        ContextManager.activeSpan().errorOccurred().log(e);
        throw e;
    } finally {
        ContextManager.stopSpan();
    }
}
}

```

Before executing `AsyncExecutionAspectSupport#doSubmit`, it is necessary to obtain the main thread of `ContextSnapshot (Trace snapshot)` and rewrite the task object. The source code of the Spring `@Async` component trace snapshot delivery is as follows:

```

public class DoSubmitMethodInterceptor implements InstanceMethodsAroundInterceptor {

    @Override
    public void beforeMethod(EnhancedInstance objInst, Method method, Object[]
        allArguments, Class<?>[] argumentsTypes, MethodInterceptorResult result)
        throws Throwable {
        // If the current trace exists, save the snapshot in the first parameter object
        if (ContextManager.isActive()) {
            // Obtain the trace snapshot of the main thread ContextManager.capture(),
            and rewrite the task object as SWCallable
            allArguments[0] = new SWCallable((Callable) allArguments[0],
                ContextManager.capture());
        }
    }
}

```

To load the agent plug-in, you need to define the interception of Spring @Async in the skywalking-plugin.def definition file of the SkyWalking Agent plug-in. The source code is as follows:

```
spring-async-annotation=org.apache.skywalking.apm.plugin.spring.async.define.  
AsyncExecutionInterceptorInstrumentation
```

The development of the Spring @Async agent plugin is now complete.

## 10.5 Chapter summary

There are two main steps in the development of the SkyWalking agent plug-in: the definition of the interception form and the interceptor implementing the interceptor form. The form of interception depends on the trace process of the framework. You can learn how to design the interception form through the cross-process RPC framework Apache Dubbo and the cross-thread framework Spring @Async. Then, in order to implement tracing, develop the plug-in with the API of an appropriate trace object based on the trace characteristics of the framework.

## Chapter 11:

# Message communication mode between the agent and back-end

---

In this chapter, you will be introduced to the principles of the message communication mode between SkyWalking's agent and back-end, and how it is expanded. After reading this chapter, you will have a better understanding of the communication mode between the agent and the back-end, and you will be able to customize the expansion using SkyWalking 's expansion method.

## 11.1 Why the official default does not provide multiple methods

The agent and back-end message communication modes mainly belong to two categories: registration communication and data reporting communication. For these two types of communication modes, SkyWalking officially supports communication via gRPC and HTTP/1.1. The message queue is not used by default. This is because the deployment, performance consumption and storage of the message queue are not economical from the perspective of resource utilization. From the standpoint of monitoring, message queues are not recommended either. The biggest difference between APM and log collection is that APM needs to quickly analyze trace data and generate metrics data, topological data, and alarms. The message queue is used as a secondary cache, which significantly increases the latency of the analysis.

At the same time, for SkyWalking, it is important to ensure the stability of the core communication mode and provide an open expansion method to allow a large community organization to expand the specific communication mode. This can greatly enhance the community activity of Apache SkyWalking and allow more people to be involved in the development of the project, which in turn promotes the open source

model. Therefore, although this is not recommended, we do not reject it either. In this chapter, we will introduce how to expand this communication mode through the official interface.

## 11.2 Analyzing the communication mechanism

This section will describe the principles of registration communication and data reporting communication. You will get to know the data communication process and the source code structure between the agent and the back-end.

Before introducing registration communication between the agent and the backend, let's first learn about these basic properties and concepts of SkyWalking.

- **Service:** Represents a service system, which is defined by `Config.Agent.service_name` in the agent configuration, and **Service** with the same name indicates that it is the same **Service**.
- **ServiceInstance:** A specific process instance in **Service**.
- **NetworkAddress:** Network IP address, such as the address for calling MySQL.
- **EndpointName:** Port name, such as the URL of HTTP.
- **SERVICE\_ID:** The unique ID of the **Service** dimension assigned by the back-end maintained by Agent in local memory.
- **SERVICE\_INSTANCE\_ID:** The unique ID of the **ServiceInstance** dimension assigned by the back-end maintained by Agent in local memory.
- **NetworkAddressDictionary:** It maintains the key-value pairs of the network addresses encountered by the agent and the unique IDs distributed by the back-end to these network addresses. When reporting duplicate network addresses, it uses a unique ID to replace the real network address for uploading, reducing the size of network data packages and improving performance.
- **EndpointNameDictionary:** It maintains the key-value pairs of the port names encountered by the agent and the unique IDs distributed by the back-end to these port names. It has the same function as the property above.
- **Commands:** Command messages returned to the Agent by the Backend, and Commands executed by Agent.

### 11.2.1 Registration communication between the agent and the back-end

Registration communication between the agent and the back-end in SkyWalking is mainly classified into the following five types:

- **ServiceRegister:** Requests the back-end to obtain ServiceId according to Config.Agent.service\_name.
- **ServiceInstanceRegister:** Requests the back-end to obtain ServiceInstanceId according to ServiceId+UUID.
- **ServiceInstancePing:** Regular heartbeat communication. The Backend will return Commands.
- **NetworkAddressRegister:** Regularly sends the network address unassigned with a unique ID to the backend to obtain the ID.
- **EndPointNameRegister:** Regularly sends the port name unassigned with a unique ID to the back-end to obtain the ID.

### *Agent registration*

The main purpose for the Agent initiating registration communication is to register a large amount of duplicate information (such as application name, IP address, and port name) to the back-end through a one-time network communication. The back-end assigns a unique ID that subsequent Agents may use for communication with the back-end. In this way, the pressure of data communication in the network can be reduced, and the back-end will also use this ID for subsequent stream computing.

Figure 11-1 shows the flowchart for registration communication in the Agent registration end.

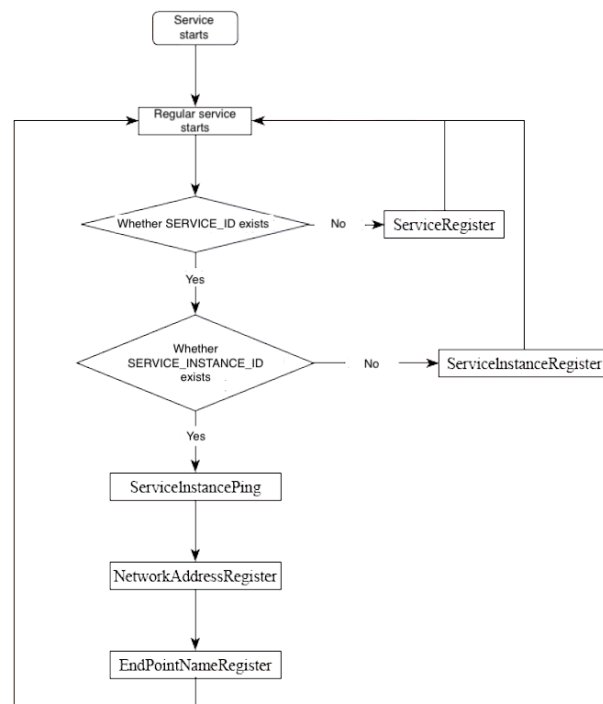




Figure 11-1 The agent registration communication process

For example, when a service with `Config.Agent.service_name` as `demo_app` starts, the following process will occur:

- 1) The service will initiate a `ServiceRegister` request with a parameter of `demo_app` to obtain the unique `ServiceId` of the service whose `ServiceName` is `demo_app` in the back-end.
- 2) The service initiates a `ServiceInstanceRegister` request with the parameters `ServiceId` + `UUID` + current timestamp + machine OS information to obtain the unique `ServiceInstanceId` of the current service instance in the back-end.
- 3) After the first two steps are successfully completed, the `ServiceId` and `ServiceInstanceId` will always be stored in the service memory, and Steps 4 to 6 will be executed synchronously.
- 4) The service will initiate a `serviceInstancePing` request with `ServiceInstanceId` + `UUID` + current timestamp as the parameters. The back-end will continuously update the latest heartbeat time and return `Commands` to the Agent.
- 5) The service will save all the `NetworkAddresses` it encounters in the `unRegisterServices` parameter in the `NetworkAddressDictionary` during the collection of trace data, and will send these parameters to the Backend to obtain the `NetworkAddressId` of these `NetworkAddresses` in the back-end.
- 6) This step is similar to Step 5. The service will save all the port names it encounters in the `unRegisterEndpoints` in the `EndpointNameDictionary` during the collection of trace, and will send these parameters to the Backend in batches in order to obtain the `NetworkAddressId` of these `NetworkAddresses` in the back-end.

The relevant registration code details are as follows:

```
@DefaultImplementor
public class ServiceAndEndpointRegisterClient implements BootService, Runnable,
    GRPCChannellListener {
    ...
    @Override
    public void run() {
        ...
        if (RemoteDownstreamConfig.Agent.SERVICE_ID ==
            DictionaryUtil.nullValue() ) { if (registerStub !=
            null) { // Initiate ServiceRegister request
                ServiceRegisterMapping serviceRM= registerStub
                    .withDeadlineAfter(10, TimeUnit.SECONDS)
```

```

        .doServiceRegister(Services.newBuilder()
        .addServices(Service.newBuilder()
        .setServiceName(Config.Agent.SERVICE_NAME)).build());

if (serviceRegisterMapping != null) {
    List<KeyIntValuePair> registereds =
    serviceRM.getServicesList();
    for (KeyIntValuePair registered : registereds) {

        if (Config.Agent.SERVICE_NAME.equals(registered.getKey())) {
            Agent.SERVICE_ID = registered.getValue();
            // Save the ServiceId returned by Backend
            shouldTry = true;
        }
    }
}
} else {
    if (registerStub!= null) {
        if (Agent.SERVICE_INSTANCE_ID == DictionaryUtil.nullValue())
            {ServiceInstanceRegisterMapping iM= registerStub
            .withDeadlineAfter(10, TimeUnit.SECONDS)
            .doServiceInstanceRegister(ServiceInstances.newBuilder()
            .addInstances(ServiceInstance.newBuilder()
            .setServiceId(RemoteDownstreamConfig.Agent.SERVICE_ID)
            .setInstanceUUID(INSTANCE_UUID)
            .setTime(System.currentTimeMillis())
            .addAllProperties(OSUtil.buildOSInfo()))
            .build()); // Initiate ServiceInstanceRegister request

        for (KeyIntValuePair serviceInstance :
        iM.getServiceInstancesList()) { if
        (INSTANCE_UUID.equals(serviceInstance.getKey()))
        {
            int serviceInstanceId =
            serviceInstance.getValue();
            if (serviceInstanceId !=
            DictionaryUtil.nullValue()) {

                // Save the ServiceInstanceId returned by Backend
                Agent.SERVICE_INSTANCE_ID = serviceInstanceId;

                // Save the timestamp of the successful registration
                Long now = nowSystem.currentTimeMillis();
                Agent.INSTANCE_REGISTERED_TIME = now;
            }
        }
    }
} else {
    final Commands commands = serviceInstancePingStub

```

```

.withDeadlineAfter(10, TimeUnit.SECONDS)
.doPing(ServiceInstancePingPkg.newBuilder()
.setServiceInstanceId(Agent.SERVICE_INSTANCE_ID)
.setTime(System.currentTimeMillis())
.setServiceInstanceUUID(INSTANCE_UUID)
.build()); // Initiate heartbeat communication, Backend returns
Commands information

```

```

NetworkAddressDictionary.INSTANCE
.syncRemoteDictionary(
registerBlockingStub.withDeadlineAfter(10,
TimeUnit.SECONDS)); // Initiate NetworkAddress
registration

```

```

EndpointNameDictionary.INSTANCE
.syncRemoteDictionary(registerBlockingStub
.withDeadlineAfter(
10,
TimeUnit.SECONDS))
; // Initiate Endpoint registration

```

```

ServiceManager.INSTANCE
.findService(CommandService.class)
.receiveCommand(commands); // Execute Commands returned by
Backend

```

```

    }
  }
}
...
}

```

### A. *Back-end receiver*

In the previous section, we analyzed the internal process of the Agent registration end in the registration communication mechanism. The receiving interface of the Backend also corresponds to the registration interface of Agent, and it is mainly classified into the following two categories:

- `org.apache.skywalking.oap.server.receiver.register.provider.handler.v6.grpc.RegisterServiceHandler`:

It completes the functions of `doServiceRegister`, `doServiceInstanceRegister`, `doEndpointRegister` and, `doNetworkAddressRegister`.

- `org.apache.skywalking.oap.server.receiver.register.provider.handler.v6.grpc.ServiceInstancePingServiceHandler`:

It mainly completes the `doServiceInstancePing` function.

- In the following, we will look into the source code of the Backend receiver:
- i) `doServiceRegister`: Returns its unique Service NameId according to the ServiceName sent by the Agent.

```
@Override public void doServiceRegister(Services request,
    StreamObserver<ServiceRegisterMapping> responseObserver) {
    ServiceRegisterMapping.Builder builder = ServiceRegisterMapping.newBuilder();
    request.getServicesList().forEach(service -> {
        String serviceName =
            service.getServiceName(); if
            (logger.isDebugEnabled()) {
                logger.debug("Register service, service code: {}", serviceName);
            }
        // Create ServiceId or obtain the ServiceId corresponding to the
        // existing ServiceName
        int serviceId = serviceInventoryRegister
            .getOrCreate(serviceName, null);

        if (serviceId != Const.NONE) {
            KeyIntValuePair value = KeyIntValuePair.newBuilder()
                .setKey(serviceName)
                .setValue(serviceId).build();
            builder.addServices(value);
        }
    });

    responseObserver.onNext(builder.build());
    responseObserver.onCompleted();
}
```

There are multiple ServiceNames in the sender's Request (the Agent sender will only send one). The server traverses all ServiceNames and calls `serviceInventoryRegister.getOrCreate()` to obtain or create a new ServiceId (the implementation of this method will not be explained in this chapter, so you may read the relevant source code on your own), and to combine this ServiceId and its ServiceName as a Pair and return it to the sender.

- ii) doServiceInstanceRegister: Obtains the unique ID of this instance according to the ServiceId + UUID + current timestamp + Machine OS sent by the Agent.

```
@Override public void doServiceInstanceRegister(ServiceInstances request,
        StreamObserver<ServiceInstanceRegisterMapping> responseObserver) {

    ServiceInstanceRegisterMapping.Builder
        builder =
            ServiceInstanceRegisterMapping
                .newBuilder() ;

    request.getInstancesList().forEach(instance -> {
        ServiceInventory serviceInventory =
            serviceInventoryCache
                .get(instance.getServiceId());

        JsonObject instanceProperties = new
            JsonObject();
        List<String> ipv4s = new ArrayList<>();
        // The OS information of the instance sent by the agent is sorted according to
        the following categories

        for (KeyStringValuePair property :
            instance.getPropertiesList()) { String key =
                property.getKey();
                switch (key) {
                    case HOST_NAME:
                        instanceProperties.addProperty(HOST_NAME,
                            property.getValue()); break;
                    case OS_NAME:
                        instanceProperties.addProperty(OS_NAME,
                            property.getValue()); break;
                    case LANGUAGE:
                        instanceProperties.addProperty(LANGUAGE,
                            property.getValue()); break;
                    case "ipv4":
                        ipv4s.add(property.getV
                            alue()); break;
                    case PROCESS_NO:
                        instanceProperties.addProperty(PROCESS_NO,
                            property.getValue()); break;
                }
            }
        instanceProperties.addProperty(IPV4S, ServiceInstanceInventory
            .PropertyUtil .ipv4sSerialize(ipv4s));
        // Construct the name of the current instance based on different information
        String instanceName =
            serviceInventory.getName(); if
```

```

(instanceProperties.has(PROCESS_NO))
{
    instanceName += "-pid:" + instanceProperties.get(PROCESS_NO)
        .getAsString();
}
if (instanceProperties.has(HOST_NAME)) {
    instanceName += "@" + instanceProperties.get(HOST_NAME).getAsString();
}
// Create ServiceInstanceId or obtain the ServiceInstanceId
corresponding to the existing ServiceInstance
int serviceInstanceId = serviceInstanceInventoryRegister
    .getOrCreate
        (instance.getServiceId(),instanc
            eName,
            instance.getInstanceUUID(),
            instance.getTime(),
            instanceProperties);

if (serviceInstanceId != Const.NONE) {
    builder.addServiceInstances(KeyIntValuePair.n
        ewBuilder()
            .setKey(instance.getInstanceUUID())
            .setValue(serviceInstanceId));
}
});
responseObserver.onNext(builder.build());
responseObserver.onCompleted();
}

```

In `ServiceInstanceRegister`, the receiver obtains the name of the current service according to the `ServiceId` from the data of `Request`, then assembles the information according to the system OS information, and finally obtains or creates a new `ServiceInstanceId` primarily through `serviceInstance-InventoryRegister.getOrCreate()`.

- iii) `doEndpointRegister`: Return its unique ID according to the port name sent by the Agent.

```

@Override public void doEndpointRegister(Endpoints request, StreamObserver
    <EndpointMapping> responseObserver) {
    EndpointMapping.Builder builder =
        EndpointMapping.newBuilder();

    request.getEndpointsList().forEach(e
        ndpoint -> { int serviceId =
            endpoint.getServiceId();
            String endpointName = endpoint.getEndpointName();
            // Create the Id of the Endpoint corresponding to the serviceId, or return it
            if it is already created
            int endpointId = inventoryService.getOrCreate(

```

```

        serviceId, endpointName, DetectPoint
        .fromNetworkProtocolDetectPoint(endpoint.getFrom()));

    if (endpointId != Const.NONE) {
        builder.addElements(EndpointMappingElement.newBuilder()
            .setServiceId(serviceId)
            .setEndpointName(endpointName)
            .setEndpointId(endpointId)
            .setFrom(endpoint.getFrom()));
    }
});

responseObserver.onNext(builder.build());
responseObserver.onCompleted();
}

```

The receiver obtains the ServiceId and EndpointName from the sender's Request, then obtains the corresponding unique ID through `inventoryService.getOrCreate()`, and returns it to the sender.

- iv) `doNetworkAddressRegister`: Return its unique ID according to the network address sent by the Agent.

```

@Override
public void doNetworkAddressRegister(NetAddresses request,
    StreamObserver<NetAddressMapping> responseObserver) {
    NetAddressMapping.Builder builder = NetAddressMapping.newBuilder();
    request.getAddressesList().forEach(networkAddress -> {
        // Create the AddressId corresponding to networkAddress , or return it if
        // it is already created
        int addressId = networkAddressInventoryRegister
            .getOrCreate (networkAddress, null);

        if (addressId != Const.NONE) {
            builder.addAddressIds(KeyIntValuePair.newBuilder()
                .setKey(networkAddress)
                .setValue(addressId));
        }
    });

    responseObserver.onNext(builder.build());
    responseObserver.onCompleted();
}

```

The receiver obtains the `NetworkAddress` from the sender's `Request`, and obtains its corresponding unique ID through `networkAddress InventoryRegister.getOrCreate()`, and then returns it to the sender. For network address registration communication, unlike `doServiceInstanceRegister` and `doEndpointRegister`, the sender does not need to send `ServiceId`. The reason is that `NetworkAddress` is global, while the names of service instances and service ports are all service-level.

- v) `ServiceInstancePing`: Regular heartbeat communication. The Backend will return `Commands`.

```
@Override public void doPing(ServiceInstancePingPkg Request, StreamObserver
    <Commands> responseObserver) {

    int serviceInstanceId =

    request.getServiceInstanceId();

    long heartBeatTime = request.getTime();

    // Update ServiceInstance heartbeat time
    serviceInstanceInventoryRegister.heartbeat(serviceInstanceId,
        heartBeatTime);

    // Query the corresponding instance information according to serviceInstanceId
    ServiceInstanceInventory serviceInstanceInventory =
        serviceInstanceInventoryCache.get(
            serviceInstanceId);
    if
        (Objects.nonNull(serviceInstanceInven
            tory)) {
        serviceInventoryRegister.heartbeat(
            serviceInstanceInventory.getServiceId(),
            heartBeatTime);
        responseObserver.onNext(Commands.getDefaultInstance());
    } else {
        // If empty, it means that the ServiceId and ServiceInstanceId held by the
        // current agent are both illegal data.
        // Need to reset the agent end and let the agent end re-register to obtain the
        // legal ServiceId and ServiceInstanceId

        // Create reset command
        final ServiceResetCommand resetCommand =
            commandService.newResetCommand(
                request.getServiceInstanceId(
                ), request.getTime(),
```



```

        request.getServiceInstanceIdUI
        D());
        final Command command = resetCommand.serialize().build();
        final Commands nextCommands = Commands.newBuilder()
            .addCommands (command).build();
        responseObserver.onNext(nextCommands);
    }

    responseObserver.onCompleted();
}

```

Compared with other registered communication interfaces, the functions of the doPing interface may be more diverse. In the doPing method, the receiver takes out the corresponding ServiceInstanceId and heartBeatTime from the Request for data update, and then updates the update time of the ServiceInstance and the update time of the Service in turn. If the information corresponding to the ServiceInstance does not exist, meaning that the ServiceInstanceId currently sent is an illegal ID (for example, the persistent data of Backend is deleted when the Agent is running normally), the receiver will encapsulate a ServiceResetCommand (service reset command) to be returned to the Agent sender, and the Agent sender will perform the corresponding operation after receiving the instruction sent by the server.

## 11.2.2 Data reporting communication between the agent and back-end

Data reporting communication means that after the agent collects trace data, it sends the data to the back-end for subsequent data analysis. The analysis is conducted by the Agent sender and the back-end receiver.

### A. Agent sender analysis

SkyWalking's default data reporting communication method is gRPC, and its entry class is `org.apache.skywalking.apm.agent.core.remote.TraceSegmentServiceClient`.

The relevant data code is as follows:

```

@DefaultImplementor
public class TraceSegmentServiceClient implements BootService, IConsumer
    <TraceSegment>, TracingContextListener, ... {
    ...
    public void boot() throws Throwable {
        ...
    }
}

```

```

        // Initialize the memory queue
        carrier = new DataCarrier<TraceSegment>(CHANNEL_SIZE, BUFFER_SIZE);
        carrier.setBufferStrategy (BufferStrategy.IF_POSSIBLE);
        carrier.consume(this, 1);
    }

    ...

    @Override
    public void consume(List<TraceSegment> data) {
        // Initialize gRPC sending client
        StreamObserver<UpstreamSegment> Observer = serviceStub
            .withDeadlineAfter(10, TimeUnit.SECONDS)
            .collect(. );
        try {
            for (TraceSegment segment : data) {
                UpstreamSegment upstreamSegment = segment.transform();
                // Send data
                upstreamSegmentStreamObserver.onNext(upstreamSegment);
            }
        } catch (Throwable t) {
            logger.error(t, "Transform and send UpstreamSegment to
                collector fail.");
        }
        upstreamSegmentStreamObserver.onCompleted();
        ...
    }

    ...

    @Override
    // The SkyWalking agent will call this method after collecting the trace data to pass
    // in the trace data
    public void afterFinished(TraceSegment
        traceSegment) { if
        (traceSegment.isIgnore()) {
            return;
        }
        if (!carrier.produce(traceSegment)) { if
            (logger.isDebugEnabled()) {

                logger.debug("One trace segment has been
                    abandoned,
                    cause by buffer is full.");
            }
        }
    }
    ...
}

```

We will use the following class as an entry point to analyze the entire process of the SkyWalking data reporting communication between the agent and back-end.

The `TraceSegmentServiceClient` class mainly inherits the following interfaces:

- `org.apache.skywalking.apm.agent.core.boot.BootService`
- `org.apache.skywalking.apm.commons.datacarrier.consumer.IConsumer`
- `org.apache.skywalking.apm.agent.core.context.TracingContextListener`

The above three interfaces complete the creation of `TraceSegmentServiceClient`, collect trace data, and send all processes of trace data. Let's analyze the specific functions of these interfaces.

i) `org.apache.skywalking.apm.agent.core.boot.BootService`

```
public interface BootService {  
    void prepare() throws Throwable;  
    void boot() throws Throwable;  
    void onComplete() throws Throwable;  
    void shutdown() throws Throwable;  
}
```

The SkyWalking agent manages this object and its life cycle through `org.apache.skywalking.apm.agent.core.boot.ServiceManager`. It has three interfaces: `prepare()`, `boot()` and `onComplete()`, which correspond to the preparation, start, and completion phases of the current object. It is mainly responsible for initialization in these phases. `shutdown()` corresponds to the moment of destruction of the current object. It is mainly responsible for destruction of the object in this phase.

The `BootService` interface is the core interface of the SkyWalking agent. You may read the relevant source code for more information.

ii) `org.apache.skywalking.apm.commons.datacarrier.consumer.IConsumer`

```
public interface IConsumer<T> {  
    void init();  
    void consume(List<T> data);  
    void onError(List<T> data, Throwable t);  
    void onExit();  
}
```

This is the consumer interface of the SkyWalking lightweight-queue kernel. This has been described in detail in Chapter 4.

iii) `org.apache.skywalking.apm.agent.core.context.TracingContextListener`

```
public interface TracingContextListener {  
    void afterFinished(TraceSegment traceSegment);  
}
```

This is the core component for obtaining trace data. After the SkyWalking agent collects trace data, the data will be passed in by calling the `void afterFinished(TraceSegment traceSegment);` method.

With our knowledge of the above interfaces and their functions, let's take a look at how these interfaces coordinate with one another to report trace data to the back-end. The relevant process is summarized as follows:

- 1) The instantiation of the `TraceSegmentServiceClient` object begins.
- 2) `TraceSegmentServiceClient.boot()`: Initialize the lightweight-queue `DataCarrier` inside the agent.
- 3) When the agent `TracingContext` collects trace data, it will call back the `TracingContextListener.afterFinished (TraceSegment traceSegment)` interface (which is registered after the `boot()` initialization). This interface internally writes trace data into `DataCarrier`.
- 4) When there is data in the `DataCarrier` queue, `IConsumer.consumer(List data)` will be called. `TraceSegmentServiceClient` is responsible for sending trace data to the back-end through gRPC.

## B. Back-end receiver analysis

The entrance of the back-end receiver is at `org.apache.skywalking.oap.server.receiver.trace.provider.handler.v6.grpc.TraceSegmentReportServiceHandler`. With SkyWalking's excellent encapsulation, the calculation and processing of trace data are all encapsulated in `Collect(StreamObserver<Commands>)` in the `send` method of `SegmentParseV2.Producer segmentProducer`. The relevant code is as follows:

```

public class TraceSegmentReportServiceHandler extends ... {
    private final SegmentParseV2.Producer segmentProducer;
    public TraceSegmentReportServiceHandler(SegmentParseV2.Producer
        segmentProducer, ModuleManager moduleManager) {
        this.segmentProducer = segmentProducer;
        ...
    }

    @Override

    public StreamObserver<UpstreamSegment> collect(StreamObserver<Commands>
        observer) {
        return new StreamObserver<UpstreamSegment>() {
            @Override public void onNext(UpstreamSegment segment) {
                segmentProducer.send(segment, SegmentSource.Agent);
            }
            ...
        };
    }
}

```

## 11.3 Extending communication modes

Officially, SkyWalking only provides the gRPC communication method. However, the official gRPC may not be the best method for some users (one possible reason being that the company has its own infrastructure that is more stable and robust). In this section, we will show you how to extend new communication modes, such as using other communication modes to extend registration communication and data reporting communication.

Since extension in SkyWalking is conducted through SPI, the communication classes of the Agent can be found in the `org.apache.skywalking.apm.agent.core.boot.BootService` file in the `src/main/resources` folder under the `apmagent-core` Module.

```

org.apache.skywalking.apm.agent.core.remote.TraceSegmentServiceClient
org.apache.skywalking.apm.agent.core.context.ContextManager
org.apache.skywalking.apm.agent.core.sampling.SamplingService
org.apache.skywalking.apm.agent.core.remote.GRPCChannelManager
org.apache.skywalking.apm.agent.core.jvm.JVMService
org.apache.skywalking.apm.agent.core.remote.ServiceAndEndpointRegisterClient
org.apache.skywalking.apm.agent.core.context.ContextManagerExtendService
org.apache.skywalking.apm.agent.core.commands.CommandService
org.apache.skywalking.apm.agent.core.commands.CommandExecutorService

```

`org.apache.skywalking.apm.agent.core.remote.TraceSegmentServiceClient` is the entry point for the data reporting communication between the agent and the back-end.

`org.apache.skywalking.apm.agent.core.remote.ServiceAndEndpointRegisterClient` is the entry point for the registration communication between the agent and the back-end.

If you have read the source code, you may have noticed that there is a `@DefaultImplementor` annotation on the `TraceSegmentServiceClient` and `ServiceAndEndpointRegisterClient` classes. This annotation is the default implementation of some core function classes defined by the SkyWalking agent end. Any core function class that needs to extend the SkyWalking agent end has to add the `@OverrideImplementor` annotation, and the value in the `@OverrideImplementor` annotation must be the default implementation class and inherited default implementation class of the core function class being extended. In other words, if you want to expand the registration communication and data reporting classes, you must indicate `@OverrideImplementor(ServiceAndEndpointRegisterClient.class)` and `@OverrideImplementor(TraceSegmentServiceClient.class)` on the newly implemented classes, and inherit `ServiceAndEndpointRegisterClient` and `TraceSegmentServiceClient` respectively.

This extension is independently extended in the form of an independent module through SkyWalking's SPI extension method. In the end, you only need to place the extended Jar package in a specified folder of SkyWalking to conduct the extension.

### 11.3.1 Extending registration communication with HTTP

In Section 11.2, we introduced registration communication between the agent and the back-end, which involve five types of registration communication. In this section, we will introduce how to extend registration communication. The extension of the communication method is not directly related to the internal logic of each registration communication, so the extension method of each registration communication is the same. You will be introduced to service registration, service instance registration, and heartbeat communication, that is, the implementation of the back-end and agents of `ServiceRegister`, `ServiceInstanceRegister`, and `ServiceInstancePing` respectively (the codes provided include the implementation of all registration methods).

#### A. Agent end

Follow these two steps to extend the Agent end.

- 1) Add a new `HttpClient` class to encapsulate the internal details of HTTP communication, and its back-end registration address is inserted through the Java environment variables.

```
public enum HttpClient {INSTANCE;
```

```

private CloseableHttpClient closeableHttpClient;
private Gson gson;
private String backendRegisterAddress;

HttpClient() {
    closeableHttpClient = HttpClients.createDefault();
    gson = new Gson();
    // Obtain back-end interface address
    backendRegisterAddress = System.getProperties()
        .getProperty("backendRegisterAddress");
    if (StringUtil.isEmpty(backendRegisterAddress)) {
        throw new RuntimeException("load http register plugin,
            but Address is null");
    }
}

// Perform Http call
public String execute(String path, Object data) throws IOException {
    HttpPost httpPost = new HttpPost("http://" + getIpPort() + path);
    httpPost.setEntity(new StringEntity(gson.toJson(data)));
    CloseableHttpResponse response = closeableHttpClient.execute(httpPost);
    HttpEntity httpEntity = response.getEntity();
    return EntityUtils.toString(httpEntity);
}

// Perform address load balancing
private String getIpPort() {
    if (!StringUtil.isEmpty(backendRegisterAddress)) {
        String[] ipPorts = backendRegisterAddress.split(",");
        if (ipPorts.length == 0) {
            return null;
        }
        ThreadLocalRandom random = ThreadLocalRandom.current();
        return ipPorts[random.nextInt(0, ipPorts.length)];
    }
    return null;
}
}

```

- 2) The default gRPC registration communication class implemented by SkyWalking is `ServiceAndEndpointRegisterClient`. We need to create a new class `ServiceAndEndpoint-HttpRegisterClient` inherited from `ServiceAndEndpointRegisterClient`, annotate with `@OverrideImplementor(ServiceAndEndpointRegisterClient.class)`, and add the `org.apache.skywalking.apm.agent.core.boot.BootService` file under the `resources/META-INF/services` folder, add the declaration of this class in `BootService`, and implement the following methods.

- `run()`: The general structure is the same as `ServiceAndEndpointRegisterClient`. The difference is that `ServiceAndEndpointRegisterClient` mainly uses gRPC for interface communication, while in `Service-AndEndpointHttpRegisterClient`, HTTP is used for interface communication.
- Other methods may be kept consistent with the `ServiceAndEndpointRegisterClient` class.

The internal details of the `run()` method are as follows:

```
if (RemoteDownstreamConfig.Agent.SERVICE_ID == DictionaryUtil.nullValue()) {
    // Means that the registration is successful; if the registration is successful, this
    // ServiceId is not a nullValue
    JSONArray jsonElements = gson.fromJson(
        HttpClient.INSTANCE.execute(
            SERVICE_REGISTER_PATH,
            Lists.newArrayList(Config.Agent.SERVICE_NAME)),
        JSONArray.class); // Request the back-end registration interface, and deserialize the
        // return parameters

    if (jsonElements != null && jsonElements.size() > 0) {
        for (JsonElement jsonElement : jsonElements) {
            JsonObject jsonObject = jsonElement.getAsJsonObject();
            String serviceName = jsonObject.get(SERVICE_NAME).getString();
            int serviceId = jsonObject.get(SERVICE_ID).getAsInt();

            // Filter out the ServiceId with the current ServiceName
            if (Config.Agent.SERVICE_NAME.equals(serviceName)) {
                // Assignment of value, and registration logic will no longer be applied
                // subsequently
                RemoteDownstreamConfig.Agent.SERVICE_ID = serviceId;
                shouldTry = true;
            }
        }
    }
} else {
    if (RemoteDownstreamConfig.Agent.SERVICE_INSTANCE_ID == DictionaryUtil.
        nullValue()) {
        // Registration of Service instance corresponding to Service registration
        JSONArray jsonArray = new JSONArray();
        JsonObject mapping = new JsonObject();
        jsonArray.add(mapping);
        // Create the request parameters for Service instance registration
        mapping.addProperty(SERVICE_ID, RemoteDownstreamConfig.Agent.SERVICE_
            ID); // ServiceId
        mapping.addProperty(INSTANCE_UUID, AGENT_INSTANCE_UUID); // Randomly generated UUID
        mapping.addProperty(REGISTER_TIME, System.currentTimeMillis());
        // Current time of registration
    }
}
```



```

// Some OS parameters of the instance
mapping.addProperty(INSTANCE_PROPERTIES, gson.toJson(
    OSUtil.buildOSInfo()));

JSONArray response = gson.fromJson(

    HttpClient.INSTANCE.execute(
        SERVICE_INSTANCE_REGISTER_PATH,
        jsonArray),
        JSONArray.class);
for (JsonElement serviceInstance : response) {
    String agentInstanceUUID = serviceInstance
        .getAsJsonObject()
        .get(INSTANCE_UUID)
        .getAsString();

    if (AGENT_INSTANCE_UUID.equals(agentInstanceUUID)) {
        // Filter out the ServiceId of the current instance and assign value to it
        int serviceInstanceId = serviceInstance
            .getAsJsonObject()
            .get(INSTANCE_ID)
            .getAsInt();
        if (serviceInstanceId != DictionaryUtil.nullValue()) {
            Agent.SERVICE_INSTANCE_ID = serviceInstanceId;
            Agent.INSTANCE_REGISTERED_TIME = System.currentTimeMillis();
        }
    }
}
} else { // Create registration parameters for heartbeat communication
    JsonObject jsonObject = new JsonObject();
    jsonObject.addProperty(INSTANCE_ID, Agent.SERVICE_INSTANCE_ID);
    jsonObject.addProperty(HEARTBEAT_TIME, System.currentTimeMillis());
    jsonObject.addProperty(INSTANCE_UUID, AGENT_INSTANCE_UUID);

    // Conduct heartbeat communication
    JsonObject response = gson.fromJson(
        HttpClient.INSTANCE.execute(
            SERVICE_INSTANCE_PING_PATH,
            jsonObject),
            JsonObject.class);

    // Obtain the command information issued by the server
    final Commands commands = gson.fromJson(
        response.get(INSTANCE_COMMAND).getAsString(),
        Commands.class);
    ServiceManager.INSTANCE.findService(CommandService.class).
        receiveCommand (commands);

    NetworkAddressHttpDictionary.INSTANCE.syncRemoteDictionary();
}

```

```

        EndpointNameHttpDictionary.INSTANCE.syncRemoteDictionary();
    }
}

```

### *B. Backend receiver*

The Backend receiver in SkyWalking version 5 supports the HTTP mode. However, in version 6, only the gRPC registration mode was retained and the HTTP registration mode was removed. Therefore, you can refer to the HTTP registration mode of SkyWalking version 5 to develop the HTTP registration communication mode for the new version of SkyWalking.

The Backend receiver of the registration communication is located in the `skywalking-register-receiver-plugin` module. The `org.apache.skywalking.oap.server.receiver.register.provider.handler.v5.rest` path is the receiver of the HTTP registration mode in SkyWalking version 5.

Before starting to write the interface, let's first learn how to use the SkyWalking encapsulation API to create an HTTP interface for registration communication (this method exposes the interface with Host and Port in SkyWalkingCore). SkyWalking encapsulates `HttpServlet` through `org.apache.skywalking.oap.server.library.server.jetty.JettyJsonHandler`. Simply inherit `JettyJsonHandler` and implement three abstract methods:

```

public abstract String pathSpec(); // Define the Path of the current HTTP interface
protected abstract JsonElement doGet(HttpServletRequest req) // For defining the Get method
protected abstract JsonElement doPost(HttpServletRequest req) // For defining the Post method

```

Then, add the current `JettyJsonHandler` to `JettyHandlerRegister` in the start method of `org.apache.skywalking.oap.server.receiver.register.provider.RegisterModule-Provider` to expose the HTTP interface.

The HTTP registration interface code for SkyWalking 5 is as follows:

```

public class RegisterModuleProvider extends ModuleProvider

{
    @Override public void start() {

        //v1
        JettyHandlerRegister jettyHandlerRegister = getManager()
            .find(SharingServerModule.NAME)
            .provider()
            .getService(JettyHandlerRegister.class);
        jettyHandlerRegister.addHandler(new ApplicationRegisterServletHandler(

```

```

        getManager()));
jettyHandlerRegister.addHandler(new InstanceDiscoveryServletHandler(getManager(
)));
jettyHandlerRegister.addHandler(new
InstanceHeartBeatServletHandler(getManager()));
jettyHandlerRegister.addHandler(
new NetworkAddressRegisterServletHandler(getManager()));
jettyHandlerRegister.addHandler(new
ServiceNameDiscoveryServiceHandler(getManager()));
    }
}

```

Now that we've seen how to develop the HTTP interface in the existing SkyWalking mode, let's begin to implement the Backend receiver of the HTTP registration communication.

The relevant steps are as follows:

- 1) Add `HttpRegisterModule` and set `ModuleName` to `http-receiver-register`.

```

public class HttpRegisterModule extends ModuleDefine {
    public HttpRegisterModule() {
        super("http-receiver-register");
    }
    @Override
    public Class[] services() {
        return new Class[0];
    }
}

```

- 2) Add `HttpRegisterModuleProvider` and inherit `RegisterModuleProvider`, set the `ModuleDefine` of `Provider` to `HttpRegisterModule`, and add `ModuleDefine` and `ModuleProvider` declarations in the `/resources/META-INF/services` folder. The declarations are the qualified class names of `HttpRegisterModule` and `HttpRegisterModule Provider` respectively.

```

public class HttpRegisterModuleProvider extends RegisterModuleProvider {
    @Override
    public Class<? extends ModuleDefine> module()
        { return HttpRegisterModule.class;
    }
    @Override
    public void start() {
        super.start();
        // Obtain instance of JettyHandlerRegister
    }
}

```

```

        JettyHandlerRegister jettyHandlerRegister = getManager()
            .find(SharingServerModule.NAME)
            .provider()
            .getService(JettyHandlerRegister.class);
        // TODO to add Handler
    }
}

```

3) Add `ServiceRegisterServletHandler` class and implement `JettyJsonHandler`. Implement the following method:

- Pass `ModuleManager moduleManager` into its constructor function. This `moduleManager` is the entry point for obtaining `IServiceInventoryRegister`, and `SkyWalking` also obtains the unique ID of the corresponding Service through `IServiceInventoryRegister`.
- The `pathSpec()` method is implemented as `"/service/register"`.
- Implement the `doPost()` method, and complete the conversion from `ServiceName` to ID in this method (to learn more about the conversion process, you can refer to the default implementation of `SkyWalking`).
- Register this Handler to `JettyHandlerRegister` in the start method of `HttpRegisterModuleProvider`.

The relevant code is as follows:

```

public class ServiceRegisterServletHandler extends JettyJsonHandler {

    private final IServiceInventoryRegister
    serviceInventoryRegister; private Gson gson = new Gson();
    private static final String SERVICE_NAME =
    "sn"; private static final String SERVICE_ID =
    "si";

    public ServiceRegisterServletHandler(ModuleManager moduleManager) {
        // Obtain the execution entry instance of ServiceRegister
        serviceInventoryRegister = moduleManager
            .find(CoreModule.NAME)
            .provider()
            .getService(IServiceInventoryRegister.class);
    }

    @Override public String pathSpec() {
        return "/ V6 /-Service / Register"; // Declare Path of Http interface
    }

    @Override protected JsonElement doGet(HttpServletRequest req) throws
    ArgumentsParseException {
        throw new UnsupportedOperationException(); // Get method is not supported
    }
}

```

```

@Override protected JsonElement doPost(HttpServletRequest req)
    throws ArgumentsParseException {
    JSONArray responseArray = new
    JSONArray(); try {
        JSONArray serviceNames = gson.fromJson(req.getReader(),
        JSONArray.class);
        for (int i = 0; i < serviceNames.size(); i++) {
            String serviceName = serviceNames.get(i).getAsString();
            // Add or obtain ServiceId
            int serviceId = serviceInventoryRegister
                .getOrCreate (serviceName, null);
            JsonObject mapping = new JsonObject();
            mapping.addProperty(SERVICE_NAME,
            serviceName); mapping.addProperty(SERVICE_ID,
            serviceId); responseArray.add(mapping);
        }
    } catch (IOException e) {
        logger.error(e.getMessage(), e);
    }
    return responseArray; // Return
}
}

```

- 4) After completing the steps above, simply add the following code in the back-end application.yml configuration:

```

http-receiver-register:
  default:

```

The extension is now enabled.

### 11.3.2 Extending data reporting communication with Kafka

In this section, we will guide you on how to extend data report communication with the commonly used message queue Kafka.

#### A. Agent sender

There are two steps:

- 1) Add a new Kafka sending Client class. This class serves to complete the initialization of the Kafka Producer and the sending of messages. Both Brokers and Topic of Kafka use Java environment variables to perform the injection. The relevant code is as follows:

```

/**
 * @author caoyixiong
 */

```

```

public class KafkaClient {
    private static final ILog logger = LogManager.getLogger(KafkaClient.class);
    private Gson gson = new Gson();
    private Producer<String, byte[]> producer;
    private String brokers;
    private String topic;

    public KafkaClient() {
        // Get the properties of brokers and topic through environment variables
        brokers = System.getProperties().getProperty("skyWalkingKafkaBrokers");
        topic = System.getProperties().getProperty("skyWalkingKafkaTopic");
        if (StringUtil.isEmpty(brokers)) {
            throw new RuntimeException("load kafka upload trace plugin, but kafka brokers is null");
        }
        if (StringUtil.isEmpty(topic)) {
            throw new RuntimeException("load kafka upload trace plugin, but kafka topic is null");
        }
        logger.info("skyWalkingKafkaBrokers is " + brokers);
        logger.info("skyWalkingKafkaTopic is " + topic);
        // Initialize Kafka Producer instance
        Properties properties = new Properties();
        properties.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG, brokers);
        properties.put(ProducerConfig.RETRIES_CONFIG, 3);
        properties.put(ProducerConfig.BATCH_SIZE_CONFIG, 16 * 1024);
        properties.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        properties.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 32 * 1024 * 1024);
        properties.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, 10 * 1024 * 1024);
        properties.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            ByteArraySerializer.class.getName());
        Thread.currentThread().setContextClassLoader(null);
        producer = new KafkaProducer<String, byte[]>(properties);
    }

    // Send Kafka message
    public void send(UpstreamSegment upstreamSegment) {producer.send(new
        ProducerRecord<String, byte[]>(
            this.topic,
            upstreamSegment.toByteArray(),
            new KafkaCallBack(upstreamSegment));
    }

    public void close()
    {producer.close();
    }
    // Callback function sent by Kafka

```

```

class KafkaCallBack implements Callback {
    private final UpstreamSegment upstreamSegment;

    public KafkaCallBack(UpstreamSegment upstreamSegment)
    {this.upstreamSegment = upstreamSegment;
    }

    @Override
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        if (exception == null) {
            // send success
            logger.error("trace segment send success");
        } else {
            logger.error("trace segment send failure");
        }
    }
}

```

- 2) Add the Kafka data reporting class in SkyWalking. This class inherits from `TraceSegmentServiceClient`, and is annotated by `@OverrideImplementor` (`TraceSegmentServiceClient.class`). Then, add the SPI implementation declaration under the `resources/META-INF/services` folder.

According to the analysis in Section 11.2.2, the following steps must be completed:

- 1) Complete the instantiation of `KafkaClient` in the previous step.
- 2) Complete the instantiation of `DataCarrier`.
- 3) Add the current object to `TracingContext.ListenerManager`, so that after the SkyWalking agent has collected trace data, it can pass the data to it.
- 4) After the trace data has been passed in, it is then transferred to the memory queue `DataCarrier`.
- 5) Send trace data to the back-end via Kafka in `org.apache.skywalking.apm.commons.datacarrier.consumer.IConsumer.consume(List<T>)`.
- 6) When the object is destroyed, move the current object out of `TracingContext.ListenerManager` and close `DataCarrier` and `KafkaClient`.

The relevant code is as follows:

```

/**
 * @author caoyixiong
 * @Date: 2019/9/22

```

```

*/
@OverrideImplementor(TraceSegmentServiceClient.class)
public class TraceSegmentKafkaServiceClient extends TraceSegmentServiceClient {
    private static final ILog logger = LogManager.getLogger(TraceSegmentServiceClient.class);

    private volatile DataCarrier<TraceSegment> carrier;
    private KafkaClient kafkaClient;

    @Override
    public void prepare() throws Throwable {
    }

    @Override
    public void boot() throws Throwable {
        kafkaClient = new KafkaClient(); // Initialize KafkaClient
        carrier = new DataCarrier<TraceSegment>(CHANNEL_SIZE, BUFFER_SIZE); // Initialize DataCarrier
        carrier.setBufferStrategy(BufferStrategy.IF_POSSIBLE);
        carrier.consume(this, 1);
    }

    @Override
    public void onComplete() throws Throwable {
        // Add this object to the callback of TracingContext, and when trace data is collected,
        // Call back the afterFinished method below
        TracingContext.ListenerManager.add(this);
    }

    @Override
    public void shutdown() throws Throwable {
        TracingContext.ListenerManager.remove(this);
        carrier.shutdownConsumers();
        kafkaClient.close();
    }

    @Override
    public void init() {
    }

    @Override
    public void consume(List<TraceSegment> data) { // Consume trace data
        try {
            for (TraceSegment segment : data) {
                UpstreamSegment upstreamSegment = segment.transform();
                kafkaClient.send(upstreamSegment); // Send data to Kafka
            }
        } catch (Throwable t) {
            logger.error(t, "Transform and send UpstreamSegment to collector fail.");
        }
    }

    @Override

```



```

        public void onError(List<TraceSegment> data, Throwable t) {

        }

        @Override
        public void onExit() {
        }

        @Override
        public void afterFinished(TraceSegment traceSegment) {
            if (traceSegment.isIgnore()) {
                return;
            }
            if (!carrier.produce(traceSegment)) {
                if (logger.isDebugEnabled()) {
                    logger.debug("One trace segment has been abandoned, cause
                        by buffer is full.");
                }
            }
        }
    }
}

```

### B. Backend receiver

The receiver of the newly extended Kafka is very similar to the receiver of SkyWalking's gRPC. Follow these seven steps to complete the implementation:

- 1) Implement the Consumer end of Kafka, pull messages from the KafkaServer regularly, and consume them with the corresponding Kafka Handler.

```

/**
 * @author caoyixiong
 */
public class KafkaServer {
    private static final Logger logger =
        LoggerFactory.getLogger(KafkaServer.class);
    private final String brokers;
    private final String topic;
    private CopyOnWriteArrayList<KafkaHandler> kafkaHandlers = new CopyOnWrite
        ArrayList<>();
    private Consumer<String, byte[]> consumer;

    public KafkaServer(String brokers, String topic) {
        this.brokers = brokers;
        this.topic = topic;
        initialize();
    }

    // Add Handler for consumption message
    public void addHandler(KafkaHandler kafkaHandler)
        {kafkaHandlers.add(kafkaHandler);
}

```

```

    }
    // Initialize Kafka Consumer
    private void the initialize () {
        Properties props = new Properties();
        props.put("bootstrap.servers", brokers);
        props.put("group.id", "sw_group");
        props.put("enable.auto.commit", "true ");
        props.put("auto.commit.interval.ms", 1000);
        props.put("session.timeout.ms", 120000);
        props.put("max.poll.interval.ms", 600000);
        props.put("max.poll.records", 100);
        props.put("key.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
            "org.apache.kafka.common.serialization.ByteArrayDeserializer");
        consumer = new KafkaConsumer<String, byte[]>(props);
        consumer.subscribe(Collections.singletonList(topic));
    }
    // Start consuming messages
    public void start() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    ConsumerRecords<String, byte[]> records =
                        consumer.poll(1000);
                    logger.info ( "kafka message obtained:" + records.count ());
                    for (KafkaHandler kafkaHandler : kafkaHandlers)
                    {
                        kafkaHandler.doConsumer(records);
                    }
                }
            }
        }).start();
    }
}

```

## 2) Add KafkaUploadTraceServiceModuleConfig to maintain Kafka's Brokers and Topic parameters.

```

/**
 * @author caoyixiong
 */
class KafkaUploadTraceServiceModuleConfig extends TraceServiceModuleConfig {
    private String kafkaBrokers;
    private String topic;
    ...
}

```

### 3) Add KafkaUploadTraceModule and set ModuleName to kafka-upload-trace.

```
/**
 * @author caoyixiong
 */
public class KafkaUploadTraceModule extends ModuleDefine {
    public static final String NAME = "kafka-upload-trace";
    public KafkaUploadTraceModule() {
        super(NAME);
    }
    @Override
    public Class[] services() {
        return new Class[]{ISegmentParserService.class};
    }
}
```

### 4) Add TraceSegmentReportKafkaServiceHandler to consume trace data.

```
/**
 * @author caoyixiong
 */
public class TraceSegmentReportKafkaServiceHandler implements KafkaHandler {
    // As introduced in section 11.2.2, this is the entry point of SkyWalking trace data
    private final SegmentParseV2.Producer segmentProducer;
    public TraceSegmentReportKafkaServiceHandler (
        SegmentParseV2.Producer segmentProducer, ModuleManager
        moduleManager) {
        this.segmentProducer = segmentProducer;
    }

    @Override
    public void doConsumer(ConsumerRecords<String, byte[]> records) {
        // Conduct circular consumption of messages
        for (ConsumerRecord<String, byte[]> record : records)
            {HistogramMetrics.Timer timer =
            histogram.createTimer(); try {
                segmentProducer.send(UpstreamSegment.parseFrom(
                    record.value()),
                    SegmentSource.Agent);
            } catch (InvalidProtocolBufferException e) {
                logger.error(e.getMessage(), e);
            } finally {
                timer.finish();
            }
        }
    }
}
```

```
}
```

- 5) Add `KafkaUploadTraceModuleProvider` inherited from `TraceModuleProvider`, and set its `Module-Config` to `KafkaUploadTraceServiceModuleConfig` and `ModuleDefine` to `KafkaUploadTraceModule`. The implementation method is basically the same as that of `TraceModuleProvider`. The difference is that in the `start()` method, you need to start the `KafkaConsumer Server` mentioned in Step 1 and the `TraceSegmentReportKafka ServiceHandler` that consumes messages.

```
/**
 * @author caoyixiong
 */
public class KafkaUploadTraceModuleProvider extends TraceModuleProvider

    {private final KafkaUploadTraceServiceModuleConfig moduleConfig

        ...

        public KafkaUploadTraceModuleProvider() {
            this.moduleConfig = new KafkaUploadTraceServiceModuleConfig();
        }

        @Override public Class<? extends ModuleDefine>
            module() {return KafkaUploadTraceModule.class;
        }

        @Override public ModuleConfig createConfigBeanIfAbsent()
            {return moduleConfig;
        }

        ...

        @Override public void start() throws ModuleStartException {
            // Initialize Kafka Service
            KafkaServer kafkaServer = new KafkaServer(
                moduleConfig.getKafkaBrokers(),
                moduleConfig.getTopic());
            // Initialize the Handler that consumes Kafka messages
            TraceSegmentReportKafkaServiceHandler handler =
            new TraceSegmentReportKafkaServiceHandler(segmentProducerV2,
                getManager());
            kafkaServer.addHandler(handler);
            // Start consuming Kafka messages
            kafkaServer.start();
        }

        ...
    }
```

6) Add the SPI declarations of ModuleDefine and ModuleProvider under the resources/META-INF/services folder.

7) Add the following code to application.yml configuration in the back-end to enable the extension.

```
kafka-upload-trace:
  default:
    kafkaBrokers: 127.0.0.1:9092 // Kafka Brokers address
    topic: test // The corresponding Topic
    // The remaining parameters are the same as the parameter configuration in SkyWalking's
    default receiver-trace
    bufferPath: ${SW_RECEIVER_BUFFER_PATH:../trace-buffer/} # Path to trace
    buffer files, suggest to use absolute path
    bufferOffsetMaxFileSize: ${SW_RECEIVER_BUFFER_OFFSET_MAX_FILE_SIZE:100} #
    Unit is MB
    bufferDataMaxFileSize: ${SW_RECEIVER_BUFFER_DATA_MAX_FILE_SIZE:500} # Unit is
    MB
    bufferFileCleanWhenRestart: ${SW_RECEIVER_BUFFER_FILE_CLEAN_WHEN_
    RESTART:false}
    sampleRate: ${SW_TRACE_SAMPLE_RATE:10000} # The sample rate precision is
    1/10000. 10000 means 100% sample in default.
    slowDBAccessThreshold: ${SW_SLOW_DB_THRESHOLD:default:200,mongodb:100} # The
    slow database access thresholds. Unit ms.
```

## 11.4 Chapter summary

In this chapter, we have introduced the basic principles of registration communication between the agent and the back-end, data reporting communication between the agent and the backend in SkyWalking, and how to extend other communication methods based on the default methods in SkyWalking. You may also extend other communication methods based on what you have learned in this chapter.

## Chapter 12:

# SkyWalking OAP Server monitoring and metrics

---

As an application performance observability tool, SkyWalking has strong self-operation and self-maintenance capabilities. In order to maintain the stability of unattended operation over time, the following features are implemented in the design:

- The storage module has a lifecycle setting that periodically cleans up expired data, removing the need for regular cleaning by operation and maintenance personnel.
- There is a built-in high-availability cluster that effectively copes with the unavailability of the cluster.
- Each main component has a file cache function. When the cluster is temporarily unavailable, data can be cached and processed again after the cluster is restored.

Despite the above-mentioned features, once SkyWalking is deployed in a production environment, it still faces these challenges:

- Agent data transmission delays;
- Data write latency;
- Limited resources at the node where the OAP is located, causing a process to freeze;
- Back-end storage related issues.

At the same time, enterprises generally need to provide alarm capabilities for production systems. Although its SLO may be lower than the first-class production system, it is still necessary to set corresponding alarm metrics for SkyWalking at the production level.

In response to the scenario described above, the Telemetry module of the SkyWalking back-end analysis service OAP is used to manage the exposure of monitoring metrics:

```
telemetry:  
  none:
```

The preferred way to export metrics is to use Prometheus. Use the following configuration to enable OAP's Prometheus metrics export endpoint:

```
telemetry:  
  prometheus:
```

This endpoint is exposed on `http://0.0.0.0:1234/` or `http://0.0.0.0:1234/metric` by default. Use the above endpoint to configure the crawling rules of the Prometheus service.

If you would like to change the listening address and port, for instance to change the listening address to `127.0.0.1:1534`, the configuration is as follows:

```
telemetry:  
  prometheus:  
    host: 127.0.0.1  
    port: 1534
```

After Prometheus has obtained the data, you can view the metrics in Grafana. SkyWalking provides two Grafana templates: the trace template and the service mesh template. Users may choose either or a combination of both for monitoring based on their own needs. They may also use these templates as the foundation for their custom development.

## 12.1 Monitoring metrics for trace mode

The trace template may be found at

<https://github.com/apache/skywalking/blob/v6.6.0/docs/en/setup/backend/telemetry/trace-mode-grafana.json>.

Figure 12-1 below shows the result after installing the trace template.

The monitoring metrics of the trace mode are explained as follows.

### A. *Process performance metrics*

As shown in Figure 12-2, these metrics include the number of processes (instances number), CPU, Java virtual machine garbage collection time (GC Time), and virtual machine memory (Memory).

The latter three are relevant to the case of a single process. When SkyWalking processes a large amount of trace data, it has to consume a lot of resources. At the same time, any resource restriction in place would adversely affect other metrics. For example, if the memory is too small, the GC will increase, which would eventually lead to excessive consumption of CPU resources. Users or operation and maintenance personnel should monitor these metrics and allocate adequate resources for SkyWalking.

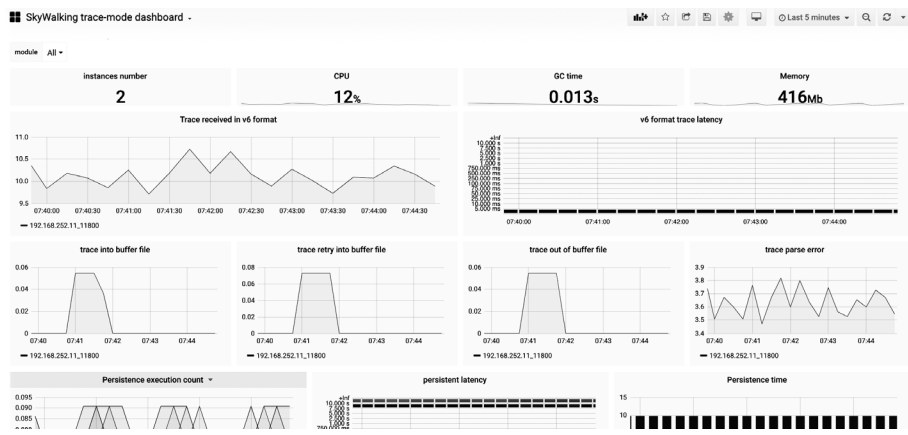


Figure 12-1 Trace mode monitoring metrics

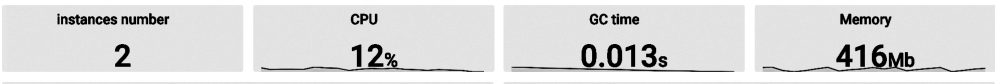


Figure 12-2 Process performance metrics

The number of processes corresponds to clusters. If the value is less than the minimum number of clusters, an alarm should be issued, because a cluster that is too small cannot cope with high volumes of traffic and may cause an avalanche effect on the entire system.

### B. Agent registration metrics

The key metrics at the time of agent registration are shown in Figure 12-3, and the module name follows after the "register worker latency-" prefix. The modules include Service, Network, Service\_Instance, and Endpoint. If you discover that a service cannot be queried within the expected time interval, you may examine these metrics.



### C. Data receiving metrics

As shown in Figure 12-4, data receiving metrics include write buffer delay, rewrite buffer ratio, the number of outBuffers, and the number of trace data processing errors. When the data is received, it will be written into the file cache as required based on the back-end processing conditions, so as to reduce the load on memory. However, any error in the file system may prevent receipt of the data. In response to this risk, there are many receiving metrics for file writing and reading that help users locate the relevant issues.

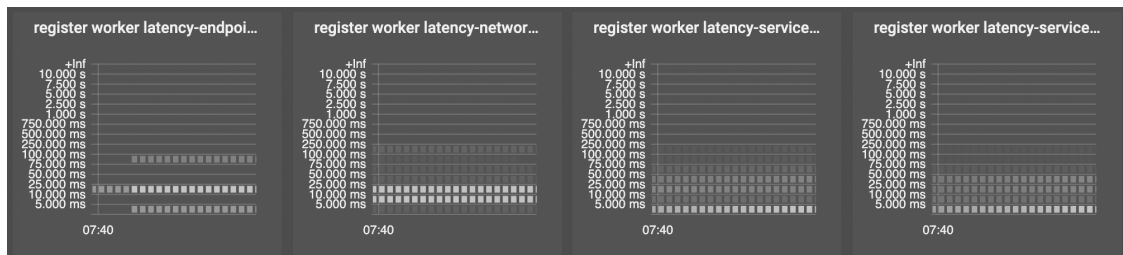


Figure 12-3 Agent registration metrics

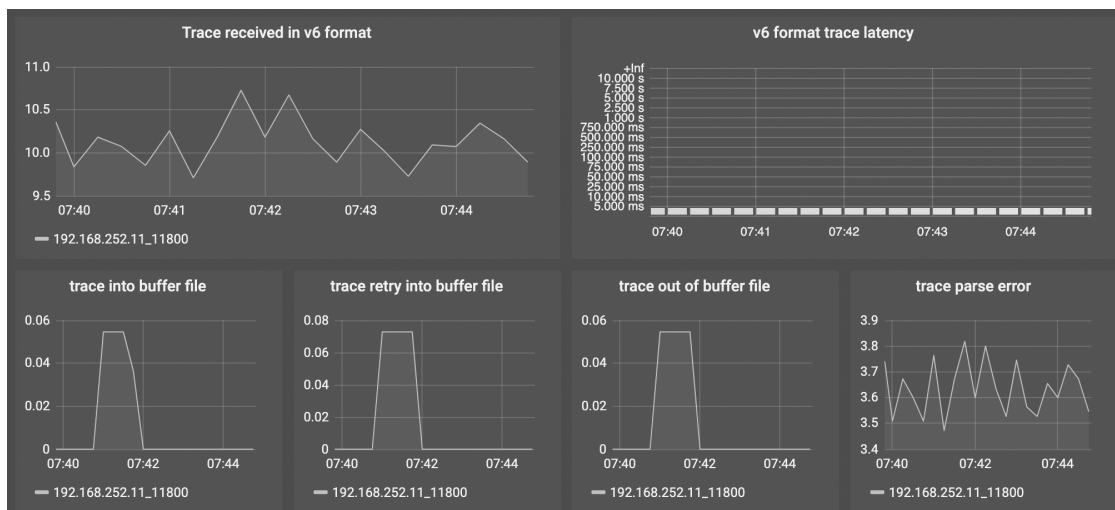


Figure 12-4 Data receiving metrics

### D. Analyzing aggregate metrics

Aggregate computing is a process of computing dispersed metrics in aggregate, accompanied by a downsampling process, meaning to aggregate the default minute-based metrics into hour-based, day-based and month-based metrics.

There is only one metric in the aggregation process, as shown in Figure 12-5, but its label generates many independent monitoring metrics based on the dimensions mentioned above. Users should first deal with key performance issues, or process them according to the configured monitoring metrics.

### E. Storage metrics

SkyWalking adopts batch storage and performs a flush operation every few seconds, as shown in Figure 12-6. Therefore, the monitoring metrics consolidate the display of the execution quantity and execution delay of the flush timer. If the flush efficiency is relatively low, the process performance metrics should be used to determine whether the problem comes from the OAP or the back-end storage.

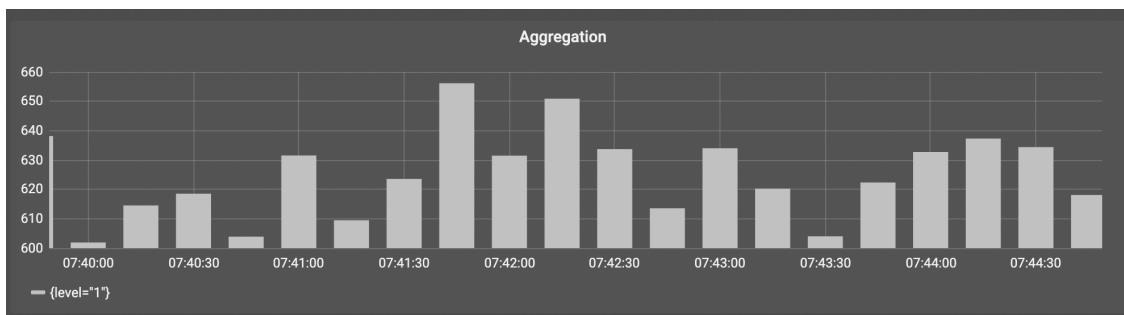


Figure 12-5 Aggregation analysis metrics

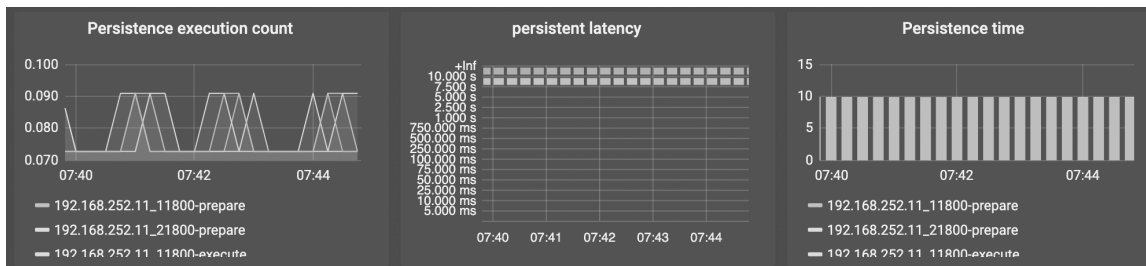


Figure 12-6 Storage metrics

## 12.2 Monitoring metrics for service mesh use cases

The service mesh template address is as follows:

<https://github.com/apache/skywalking/blob/v6.6.0/docs/en/setup/backend/telemetry/mesh-mode-grafana.json>.

The result after the installation is shown in Figure 12-7.

The service mesh metrics and trace metrics share the same classification, except that their names are different, which is reflected in the data receiving metrics. The metrics on the trace page indicate the

status of receiving trace data. Accordingly, the metrics on the service mesh page indicate the status of receiving service mesh data. Refer to Section 12.1 for the detailed definitions.

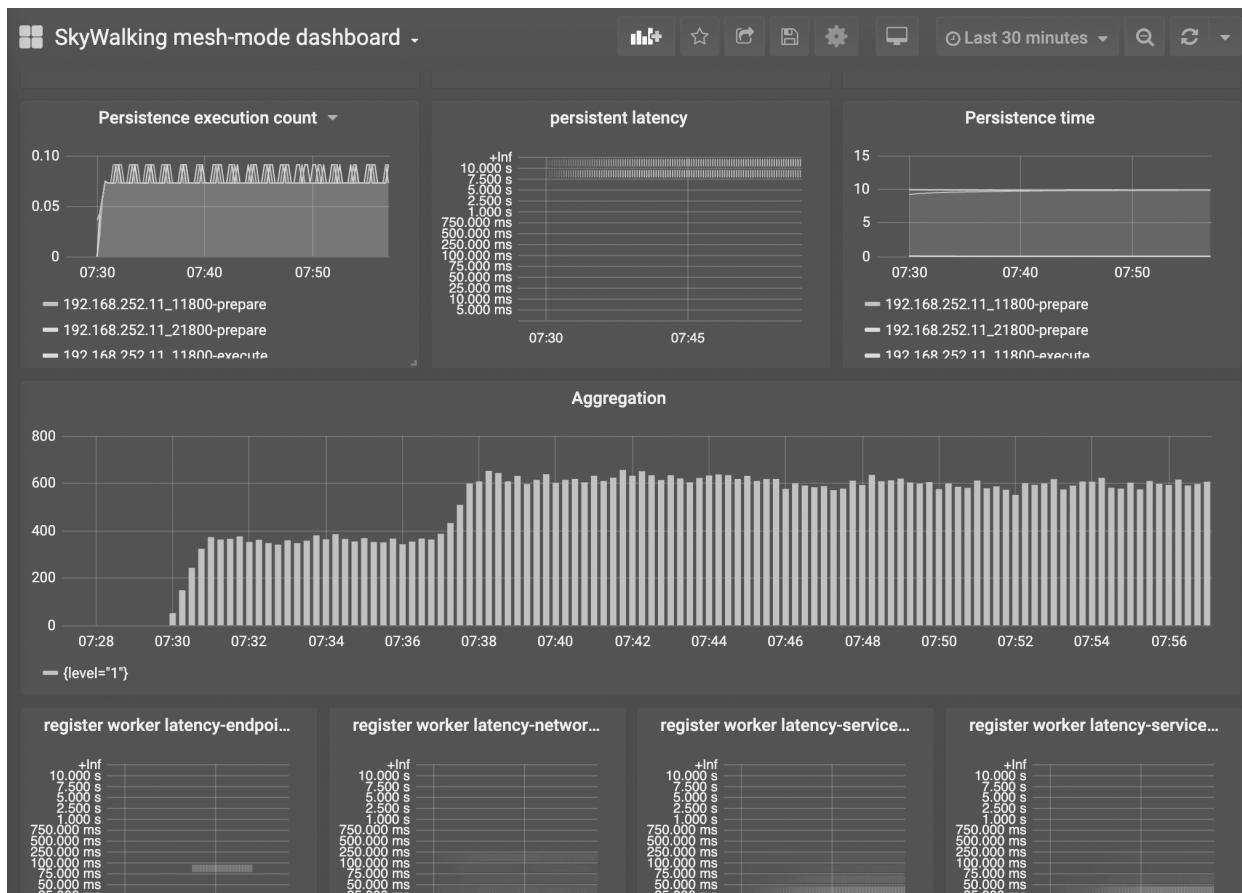


Figure 12-7 Monitoring metrics for the service mesh use case

## 12.3 Self-monitoring

As a monitoring system, can SkyWalking monitor its own operation status? The answer is yes.

In fact, SkyWalking offers a kind of self-monitoring capability:

```
receiver-solliy:
default:
telemetry:
solliy:
```

It has one more receiver than Prometheus to collect the monitoring metrics of the Prometheus port and convert them to SkyWalking metrics. Therefore, the self-monitoring mechanism of SkyWalking is dependent on the Prometheus module.

We can see that SkyWalking's monitoring metrics for its own processes are very different from those used to monitor microservice applications. While microservice applications focus on the quality of service calls, as a data processing platform, SkyWalking contains many metrics related to data processing and storage in addition to calls, which makes it difficult for SkyWalking's own model to adapt to the requirements of monitoring. For that reason, the default implementation mode is based upon Prometheus.

In the production system, users not only monitor the performance of the OAP, but also monitor the back-end storage. Apart from the fact that a series of data generated by SkyWalking will be finally written into this storage, a large number of query operations will also be performed on the storage during its operation. Therefore, users should set up an appropriate monitoring system for availability management according to storage characteristics.

## 12.4 Chapter summary

In this chapter, we have introduced the self-monitoring mechanism provided by the SkyWalking OAP Server to handle trace and service mesh use cases. We have also used the combination of Prometheus and Grafana to implement the collection and visual display of monitoring metrics. Constantly adjusting the parameters and improving the design of the monitoring system is a key aspect to be considered. After this chapter, you will not only have learned how to monitor the operating status of the OAP Server, but also understood the key components that affect its operation and the interaction between them, laying a solid foundation for the better operation and maintenance of the OAP Server.

## Chapter 13:

# The next-generation monitoring system— SkyWalking observability for service mesh

---

In Section 1.4.3, we introduced the basic concept of service mesh. In this chapter, you will learn how SkyWalking observes a service mesh.

Service mesh monitoring is often known as observability, and it is way more powerful than the traditional monitoring system. It generally contains features that include monitoring, alarms, visualization, distributed tracing, and log analysis. Observability is a superset of monitoring. Monitoring considers the target system to be a "black box" that displays the system status by tracking its key metrics and reports irregular conditions. On top of that, an observability platform gives you the additional feature of "problem finding," providing users with the ability to interactively locate problems through visualization, distributed tracing and log analysis.

In a typical scenario for an SRE, a failed target application is located through the monitoring system, and the product engineer then locates the specific problem at the code level. To maintain service mesh-based microservice clusters, the SRE requires the comprehensive capabilities offered by observability to locate more specific problems. This process is similar to debugging operations in microservice clusters.

Observability is the core issue to be natively dealt with by the service mesh. As "next-generation" infrastructure, service mesh can be used as a foundation for building observable components upon it. This is a much easier way than building observability components into the individual services. At the same time, with the implementation of infrastructure and as standards are being gradually set down, the

observable components will undergo a stable evolution, rather than being overthrown by the changes in application technology stack. For these reasons, the observability features offered by service mesh promises to see vibrant growth and great commercial potential.

In this chapter, we will first introduce SkyWalking's observability model, and then introduce SkyWalking's observation methods and future technology development trends using the examples of Istio and Envoy.

## 13.1 SkyWalking observability model

### 13.1.1 Monitoring metrics

SkyWalking mainly uses the "black box" tracing model to generate service mesh monitoring metrics. Unlike the classic "black box" algorithm, SkyWalking does not use a regression model to generate a single piece of trace data. Instead, it directly uses an analytical engine to create monitoring metrics and topology maps.

As shown in Figure 13-1, SkyWalking obtains the requested data (1, 3, 5, 7, 9, 11) marked as odd in the figure from the service mesh data plane. The traditional "black box" algorithm will try to restore the traces marked as even numbers to form a complete call trace. SkyWalking will directly perform summary statistical calculations, work out the monitoring metrics between the two nodes, and then use these paired data to construct a topology map over any given period of time.

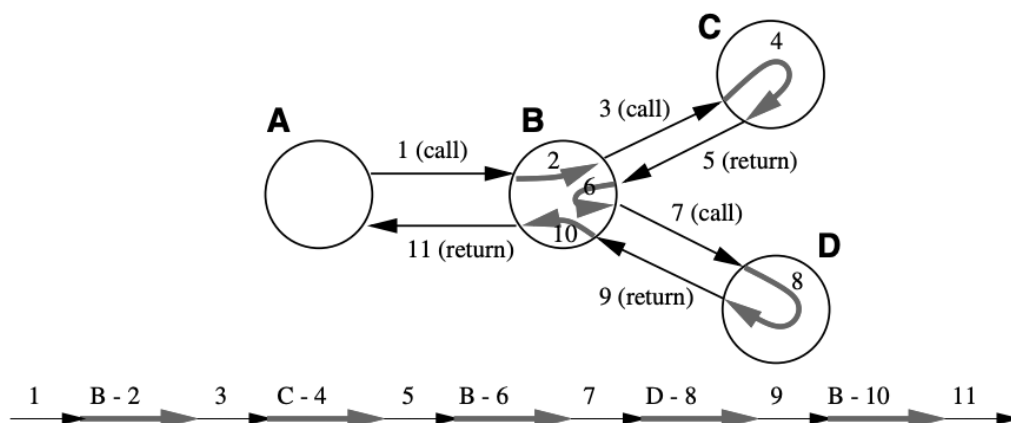


Figure 13-1 Service mesh traffic diagram

Therefore, under SkyWalking's service mesh mode, the trace function is missing, while all other functionalities are in place. This is a compromise between efficiency and functional integrity. Of course,

if you would like to use the trace function, you can adopt another set of SkyWalking cluster implementation.

The universal service mesh protocol can be found at <https://github.com/apache/skywalking-data-collect-protocol/blob/v6.6.0/service-mesh-probe/service-mesh.proto>. Currently, SkyWalking only supports Istio. If you would like to have the support of other service mesh platforms, you can use this protocol to write monitoring data into SkyWalking.

Let's look at the core content of the protocol:

```
message ServiceMeshMetric {
    int64 startTime = 1;
    int64 endTime = 2;
    string sourceServiceName = 3;
    int32 sourceServiceId = 4;
    string sourceServiceInstance = 5;
    int32 sourceServiceInstanceId = 6;
    string destServiceName = 7;
    int32 destServiceId = 8;
    string destServiceInstance = 9;
    int32 destServiceInstanceId = 10;
    string endpoint = 11;
    int32 latency = 12;
    int32 responseCode = 13;
    bool status = 14;
    Protocol protocol = 15;
    DetectPoint detectPoint = 16;
}
```

As shown in the protocol, the content is mainly about writing double-ended information for a one-time call. Note that in order to obtain the correct topology map, the service ID must be consistent. If you need to generate a topological diagram of  $A \rightarrow B \rightarrow C$ , you need to generate the following two pieces of data:

```
sourceServiceId = A
...
destServiceId = B
sourceServiceId =
B
...
destServiceId = C
```

### 13.1.2 Alarms and visualization

Service mesh's monitoring metrics and distributed tracing metrics are computed in aggregate with a uniform engine such that its alarm system can be reused. One thing to note here is the mapping of dimensions.

Take the Kubernetes environment as an example. Given that it has rich built-in resources, the question is what resources are used to map to SkyWalking's Service? There is a wide range of possibilities: Deployment, Service, Statefulset or even some types of Custom Resource. This requires users to carry out the relevant designs and map specific targets according to their own system conditions. The current official approach is to use Statefulset to map to Service, as it can point to a variety of secondary resources, and has excellent monitoring capabilities as well. If you have any customized requirements, you can also add it by yourself.

Visualization is similar to alarms. As long as the dimensions are properly defined, the monitoring metrics and topology diagrams will be shown according to the relevant dimensions.

### 13.1.3 Distributed tracing and logging

By now, you should already have understood the basic principles of distributed tracing. Theoretically, service mesh cannot bring about any changes to tracing. Since service mesh only controls the ingress and egress of traffic, simply injecting more tracing context to the proxy and sidecar would not lead to the spreading of the entire context within the cluster. Therefore, the service itself has to be injected into the tracing context.

Some suggest that not adding the propagation module to the service mesh would in effect avoid excessive consumption without affecting tracing. However, note that the more markers are traced, the better the system status can be understood, which in turn helps locate problems.

Here is an example to illustrate the effect of adding tracing capabilities to service mesh components. If a service response times out, traditionally, we would not be able distinguish between a network problem and a service problem. But with the `inbound` agent of the service mesh, we can determine what the problem is by looking at whether the agent contains any data. If the `inbound` contains data, it means that the problem lies in the target service, whereas if there is no data in the `inbound`, it is likely a network problem.

SkyWalking does not cover the collection and storage of *logs* from the standpoint of system design, but some users will use `LocalSpan` to write business logs into them in practice. At the same time, since SkyWalking will introduce business expansion fields after 7.0.0, it is foreseeable that more users will use SkyWalking as a system for receiving and analyzing logs in the future. The combination of logs, distributed tracing and monitoring metrics is also the development goal of SkyWalking's back-end analysis.



## 13.2 Observing Istio's monitoring metrics

SkyWalking conducts aggregate analysis primarily through Istio's monitoring metrics. Since Istio does not support SkyWalking's tracing context propagation feature, this will not be covered in our discussion. Let's discuss the two integration modes of SkyWalking and Istio.

### 13.2.1 ALS mode integration

SkyWalking can also perform related system integration with Envoy's ALS (Access Log Service) (see Figure 13-2). The advantage of integrating with Envoy is that it can send access logs to SkyWalking receivers very efficiently with minimal delay. But the disadvantage is that the current ALS sends a lot of data, which will potentially affect the processing performance and network bandwidth of SkyWalking; at the same time, all analysis modules rely on lower-level access logs, and some Istio-related features cannot be identified. For example, in this mode, only Envoy metadata can be recognized, and Istio virtual services cannot be effectively recognized. Note that in the topology diagram shown in 13-2, we do not find the `istio-policy` component. This is because the communication between this component and the sidecar is not forwarded through Envoy, so this cannot be obtained from ALS information.

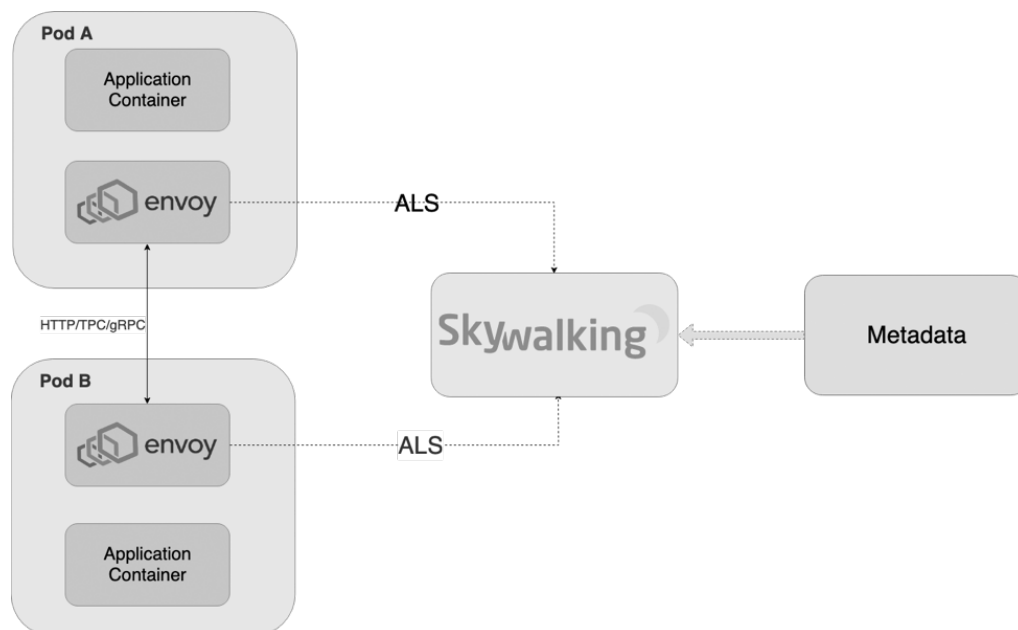


Figure 13-2 SkyWalking and ALS

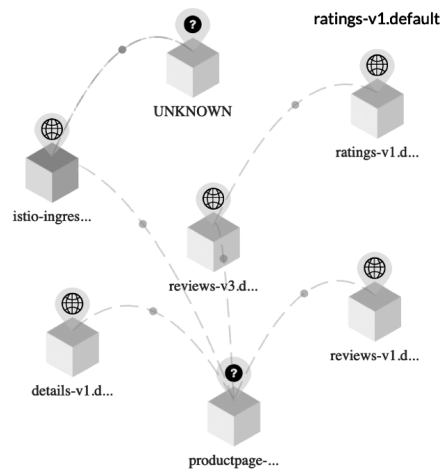


Figure 13-6 Service topology map generated with ALS

## 13.3 Observing the technological development of Istio

Both Istio and SkyWalking are currently undergoing rapid development. The following aspects of observability in Istio are in development:

- Mixer is removed. Due to the performance issues of Mixer, it will be removed. The first integration model introduced in Section 13.2 will soon be part of history.
- Envoy Wasm will replace Mixer as the main observability tool. Istio implements a WebAssembly (WASM) runtime for Envoy. With WebAssembly plug-ins for Envoy, developers can write their own code, compile it to WebAssembly plug-ins, and configure Envoy to execute it. Service-level metrics of Envoy in Telemetry V2 are supported by two plug-ins, namely metadata-exchange and stats.

In the future, SkyWalking will be thoroughly integrated with Envoy Wasm technology, which will bring the following benefits:

- Flexible development: Wasm technology is similar to Nginx's LuaJIT. Great flexibility is provided by relying on C++ and Rust languages.
- Excellent performance: Since the Wasm code will be compiled into Envoy, its performance is well guaranteed.
- Wide-ranging functionalities: Different plug-in combinations are provided according to different use cases for the application of wide-ranging functionalities.

The implications of these developments will likely impact SkyWalking, Envoy, and Istio in the following ways:

- Envoy and Istio may offer support for the tracing propagation protocol used by SkyWalking.
- Finer control of the granularity of data sent by Envoy to the OAP: The current data passed in the ALS mode is too complicated and cannot be reduced in size. With the Wasm plug-in, it is hoped that a better control may be exercised on it.
- Support for other control planes: Wasm extensions will make it easier to use the Envoy data plane with other service mesh control planes.

## 13.4 Chapter summary

In this chapter, we have introduced SkyWalking's monitoring model for service mesh, and focused on its integration with Istio. You should now have an in-depth understanding of the SkyWalking observability service mesh use case, and be able to anticipate the direction of its future development.

## Chapter 14:

# Recent releases and the future of SkyWalking

---

SkyWalking 7 and 8 have been released since the original publication of this guide came out. These recent versions maintain a high degree of compatibility with previous releases, so you don't have to worry about major changes. In this chapter, we walk through new features and the future of SkyWalking.

## 14.1 Notable features of SkyWalking 7+

The following highlights some new features that have a profound impact on users.

### 14.1.1 Java agent no longer supports JDK 1.6 and 1.7

Since most commercial and open source JDKs and Java libraries no longer support JDK versions before 1.8, the SkyWalking community decided to give up the support after completing a questionnaire survey at the end of 2019 and gave up support to ensure the reliability and stability of SkyWalking's dependency repository.

For a very small number of users of JDK 1.6 and 1.7, SkyWalking 6.x agents can be used, and SkyWalking 7 still supports their reported data.

### 14.1.2 Support for new production-level storage implementations

In SkyWalking v6, we have always supported H2, MySQL, TiDB, and Elasticsearch as storage implementations. Among them, H2 is mainly used for demonstrations and experiments. MySQL is suitable for small-scale use cases. TiDB is relatively new from the standpoint of technology, so the scope of use and use case feedback are still very limited. Elasticsearch 6.x and 7.x are well-deserved first choices for

hyper-scale storage deployments. SkyWalking's large-sized users use Elasticsearch as storage, which collects more than tens of billions of monitoring information every day.

At the same time, the SkyWalking community is constantly looking for other possibilities. In v7, we have added two new options:

- **ShardingSphere:** This is also used in the Apache community. It is a supplement to MySQL's horizontal scalability and can well meet the usage requirements of medium-sized users. At the same time, because ShardingSphere Proxy is transparent to applications, no modification of the SkyWalking code is necessary, and the integration can be completed through native routing rules.
- **InfluxDB + MySQL hybrid storage solution:** InfluxDB is a widely used time series database. More than 90% of the monitoring data has high timing characteristics. At the same time, we still use MySQL/H2 for metadata storage, which cannot be stored in the time series database. For medium-sized users and large-sized users who have purchased the enterprise version of InfluxDB, this will cost less than Elasticsearch. At the same time, this implementation also gives users who like OpenTSDB storage the opportunity to switch to a new implementation.

### 14.1.3 HTTP request parameter collection

Parameter collection is the most frequently asked and most controversial feature. The SkyWalking team understands the importance of parameter values for performance diagnosis, but at the same time, the huge performance consumption caused by the participating parameters may have a devastating blow to the system. In v7, we provide two parameters requiring manual operation by the user:

- `plugin.tomcat.collect_http_params`
- `plugin.springmvc.collect_http_params`

These two parameters are combined with `plugin.http.http_params_length_threshold` to control the length of the parameter value, and the requested parameters can be collected. Needless to say, both the monitored system and SkyWalking need to take on a lot of burden. In Section 14.2, we will introduce a more cost-effective diagnostic mode of performance profiling.

### 14.1.4 HTTP collection protocol and Nginx monitoring

Java, PHP, GoLang, .NET Core, and Node.js have always been the main focus of support for the SkyWalking agent. As of 7.0.0, we have reinstated the HTTP/1.1 network protocol that is no longer

available in the entire v6. The Nginx + Lua agent (<https://github.com/apache/skywalking-nginx-lua>) has also successfully launched with the help of the Apache APISIX project. As the most commonly used load balancing and gateway middleware in China, Nginx's monitoring support makes up for a previous defect in functionality, and makes the call tracing and topology more comprehensive and precise.

#### 14.1.5 Further optimization of Elasticsearch storage

As the most widely used storage implementation at present, Elasticsearch has undergone many refactorings and optimizations during 6.0 to 6.3, and it currently works well in a great number of production environments. In v7 and v8, we have added more monitoring metrics, including topology details, and even other monitoring endpoints. With this update, the number of Elasticsearch indexes has become a new challenge. Starting from 7.0.0, we have conducted further integration of the indexes. The metrical indexes precise to the minute, hour, and day have been merged into a single index, and the overall number of indexes has dropped by half. At the same time, since the hour-based and day-based indexes are precise only to 1/60 and 1/1440 of the minute-based index respectively, the performance of the index after the index merge is almost the same as that of the original minute-based index.

With this feature, we can confidently develop more monitoring metrics. At the same time, SkyWalking has been exploring the possibility of browser monitoring. Therefore, the latest upgrade also prepares for the functionality of browser monitoring.

## 14.2 Code performance analysis

The core features of SkyWalking 7's new functions can be analyzed as attributes. It truly evaluates and diagnoses the execution performance of a single method in a production environment. Moreover, it combines SkyWalking APM's Metrics and distributed tracing capabilities to safely perform performance analysis in a high-stress and high-sensitivity production environment.

### 14.2.1 Basic Principles of Performance Analysis

The fact that most program operation models are created based on the general concept of threads makes performance analysis possible. Most of the business logic runs in a single thread.

Code-level performance analysis uses method stack snapshots to analyze and summarize method execution, and to estimate the speed of code execution.

Once performance profiling is activated, it will periodically take snapshots of the specified thread, as well as summarize and analyze all the snapshots. If two consecutive snapshots contain the same method stack, it means that the method in this stack is likely to be at its execution state within this time interval. Thus, the cumulative period of the relevant time interval during which consecutive snapshots are taken is the estimated method execution time. The time estimation method is shown in Figure 14-1.

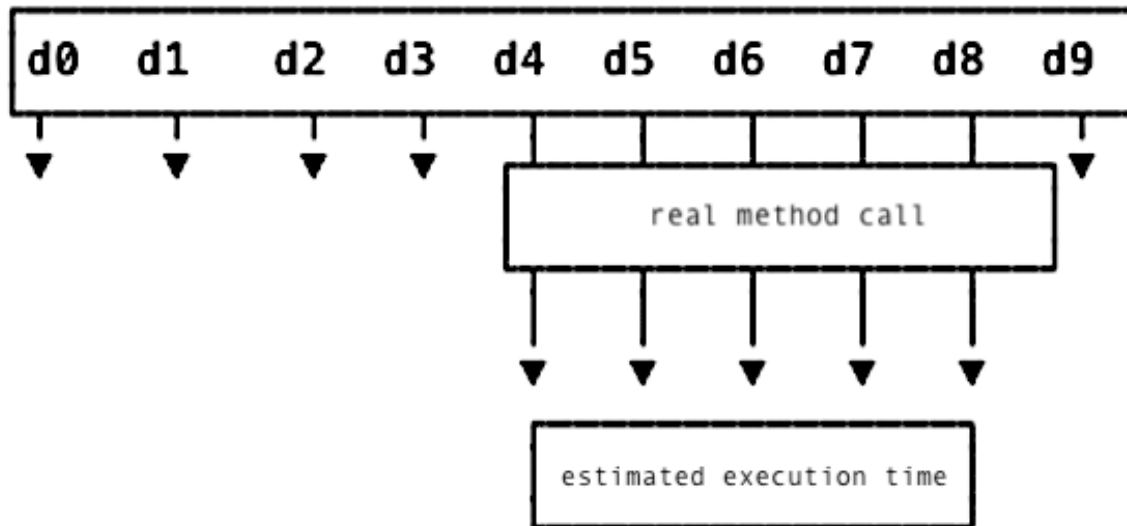


Figure 14-1 Time estimation method

In Figure 14-1, d0 to d9 represent 10 consecutive memory stack snapshots. The actual method execution time is between d3 and d4, and the end time is between d8 and d9. Performance profiling does not indicate the exact execution time of the method, but it can estimate the cumulative period of the time intervals during the method execution time of d4 to d8 when the four snapshots are taken. This is regarded as a very accurate estimate of time.

### 14.2.2 Functional characteristics of performance analysis

Performance profiling can monitor the stack information of threads well, with the following advantages:

- Accurate problem finding by going directly to the code method and lines of code.
- No need for routine addition and deletion of points for event tracing, which greatly reduces the labor costs.
- Reduced pressure and performance risks from excessive event tracing on the target system and monitoring system.

- Usage on demand with no consumption of the system at rest times and stable consumption during use.

These are powerful advantages compared to traditional monitoring. We have also mentioned these features [in the article titled "Apache SkyWalking: Use Profiling to Fix the Blind Spot of Distributed Tracing"](#).

### 14.2.3 Use cases

With the explanation set out above, the principles of performance analysis should now be easily understood. The most frequently asked question by users is the aspect of its performance consumption. Many believe that thread snapshots are highly performance-consuming. In fact, it depends on the size of the scope of the user's thread snapshots.

As a powerful APM system, SkyWalking is fully prepared for high-quality monitoring in the production environment, both in terms of function and performance. In versions prior to 7.0.0, through topology maps, metrics, and distributed tracing, performance issues can be well delimited. This delimitation is demonstrated in that the slow request service, service instance, endpoint, and specific code range can be probed. As an interactive function, performance analysis is initiated for a specific endpoint of a characteristic service after the pre-delimited operation is completed. In the process of analysis, the degree of parallelism is strictly limited (the default maximum will not exceed 10), and the interval between thread snapshots cannot be less than 10ms. For specific functions of slow methods, a minimum agent accuracy of 10ms is sufficient. Moreover, performance analysis does not allow multiple analysis instructions in the same time range, which also ensures that the pressure on a single service is manageable.

In short, a lot of restrictions are already in place, and it is recommended that users set the parameters before use to ensure an accurate and effective performance analysis.

In addition, starting from 7.1.0, performance analysis will automatically activate the parameter collection function introduced in Section 14.1.3, and more advanced features may be activated in the future to help users determine the bugs in coding by combining Trace and analytical results.

## 14.3 SkyWalking 8

SkyWalking 8 maintains the logical consistency between the agent and the back-end protocol. Two years after the release of SkyWalking 3.2, the support for the old version of the protocol has ceased. Version 8.0 sets a new standard of the protocol for the first time. At the same time, metadata information such as



registration and ID exchange has been completely removed, streamlining the processes of system operation and maintenance as well as upgrades.

Another key feature of 8.0 is the introduction of MeterSystem, which allows the traditional tracing mode to run in conjunction with service mesh mode and allows users to collect custom metrics. Tracing and service mesh are focused on topology and service traffic metrics, but the MeterSystem can report any business metric that's of interest to the user, such as database access performance, for example, or Christmas order rates, or the percentage of users who register or make purchases. Data for that metric would then be represented visually on the SkyWalking UI dashboard. The indicator panel data and topology map can be derived from Envoy's metrics, and the tracing analysis can support Istio's telemetry. The dashboard can import/export through a JSON format and the customizable metrics on the dashboard can be configured with options such as metric name, entity type (service, instance, endpoint or all), label values, etc. The logic and prototype configuration of UI to describe the dashboard, tab, and component are described in the UI template.

## 14.4 Chapter summary

In this chapter, we have outlined some key new features of SkyWalking. The project and the community are constantly evolving. You can visit our GitHub repository at <https://github.com/apache/skywalking>, or subscribe to our official developer mailing list at [dev@skywalking.apache.org](mailto:dev@skywalking.apache.org) to know about the latest developments in SkyWalking.

To subscribe to the mailing list, drop us an email at [devsubscribe@skywalking.apache.org](mailto:devsubscribe@skywalking.apache.org) and follow the instructions in the reply.