

## 3、项目搭建

---

### 3.1 项目概述

---

搭建一套微服务环境，实现多点日志采集，用于web请求的访问链路跟踪，包含以下跟踪点：

- 请求的前台页面
- 请求到达nginx的转发记录
- 请求的后台方法
- 请求的业务输出标记
- 远程的方法调用（如有涉及）

### 3.2 设计目标

---

#### 3.2.1 检索维度

可以按常用维度做到快速检索：

- 某次请求
- 某个用户
- 某个终端

#### 3.2.2 分布式环境

- 机器以集群形式部署多台
- 微服务存在docker等容器化部署场景
- 不同机器的同一服务日志要做到集中展示

#### 3.2.3 低入侵

在保障日志内容详尽的前提下，尽量做到业务代码无感知

#### 3.2.4 低耦合

日志服务不要影响主业务的进行，日志down机不能阻断业务执行

#### 3.2.5 时效性

日志在分布式环境中无法做到完全实时，但要尽量避免过长的时间差

#### 3.2.6 缓冲与防丢失

在日志服务不可用时，对应用的日志产出要做到暂存，缓冲，防止丢失

## 3.3 软件设计

---

### 3.3.1 基本概念

本项目中，与请求链路相关的概念约定如下：

- rid (requestId)：一次请求的唯一标示，生成后一直传递到调用结束
- sid (sessionId)：用户会话相关，涉及登陆时存在，不登陆的操作为空
- tid (terminalId)：同一个终端的请求标示，可以理解为同一个设备。可能对应多个用户的多次请求

### 3.3.2 前台页面

- 页面首次加载时生成tid，写入当前浏览器的cookie，后续请求都会携带
- 页面请求时，生成唯一性的rid作为一次请求的发起
- 如果用户session作为sid，用于识别同一用户的行为

### 3.3.3 nginx

- 方案一：通过accesslog可以打印，filebeat采集本地文件方式可以送入kafka通道
- 方案二：lua脚本可以直接送入kafka
- logstash获取kafka日志数据，进入es

### 3.3.4 java服务端

- gateway作为统一网关，处理三个维度变量的生成与下发
- 各个微服务集成kafka，将自己需要的日志送入kafka

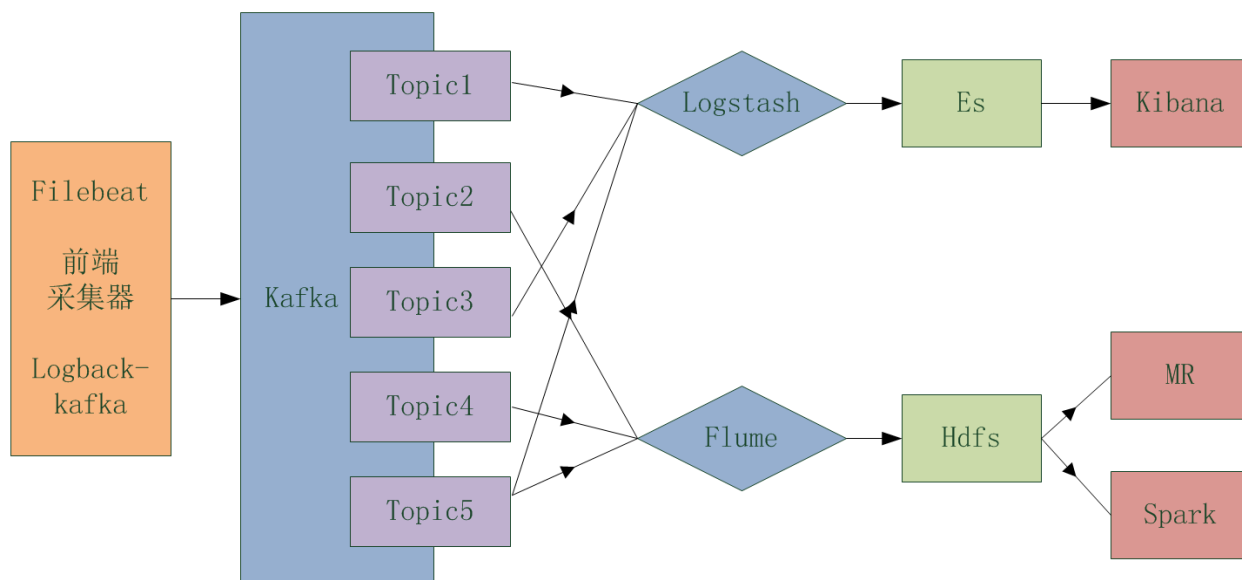
### 3.3.5 方法调用

- Aop切面，拦截器等系统组件可以以插件形式介入日志采集，缺点是无法灵活做到个性化定制
- 注解操作灵活，可以自由定制输出内容，缺点是对代码存在轻微入侵

### 3.3.6 远程调用

- 远程方法之间的调用，三个id通过显式参数形式进行传递，由当前服务到下一个服务

### 3.3.7 系统拓扑



## 3.4 框架搭建

搭建一个微服务项目，集成日志平台的对接。

### 3.4.1 模块划分

- 父pom：maven项目的父pom，用于整个项目的依赖继承，版本传递和约束
- nacos：springcloud的注册中心和配置中心
- gateway：微服务网关，请求链路进入后台的第一道关口
- web：springboot集成web组件，用于模拟我们的上层web项目
- user：基于springcloud微服务模块，用于模拟实际项目中的一个用户微服务
- utils：存放日常使用的一些工具类

### 3.4.2 nacos

Nacos 支持基于 DNS 和基于 RPC 的服务发现（可以作为springcloud的注册中心）、动态配置服务（可以做配置中心）、动态 DNS 服务。

- 1) 快速启动参考：<https://nacos.io/zh-cn/docs/quick-start.html>
- 2) 下载解压，可以修改conf/application.properties，配置端口信息
- 3) 启动：sh /opt/app/nacos/bin/startup.sh -m standalone


### 3.4.7 父pom


- 1) 访问 <http://start.spring.io>
- 2) 选择版本，并填写相关坐标


Project	<b>Maven Project</b>	Gradle Project				
Language	<b>Java</b>	Kotlin Groovy				
Spring Boot	2.3.0 M2	2.3.0 (SNAPSHOT)	2.2.5 (SNAPSHOT)	<b>2.2.4</b>	2.1.13 (SNAPSHOT)	2.1.12
Project Metadata	Group com.itheima.logdemo					
	Artifact parent					


### 3) 选择相关依赖组件


#### Selected dependencies

**Nacos Service Discovery**  
 Service discovery with Alibaba Nacos.
 

**Nacos Configuration**  
 Support for externalized configuration in a distributed system, auto refresh when configuration changes.
 

**Spring Web**  
 Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 

**Thymeleaf**  
 A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
 

**Ribbon [Maintenance]**  
 Client-side load-balancing with Spring Cloud Netflix and Ribbon.
 

### 4) 生成方式选explorer，因为我们只需要pom作为父坐标，其他文件不需要

 pom.xml | [Download](#) | [Copy!](#)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.2.4.RELEASE</version>
9         <relativePath/> <!-- Lookup parent from repository -->
10    </parent>
11    <groupId>com.itheima.logdemo</groupId>
12    <artifactId>parent</artifactId>
13    <version>0.0.1-SNAPSHOT</version>
14    <name>parent</name>
15    <description>Demo project for Spring Boot</description>
16
17    <properties>
18        <java.version>1.8</java.version>
19        <spring-cloud-alibaba.version>2.2.0.RELEASE</spring-cloud-alibaba.version>
20        <spring-cloud.version>Hoxton.SR1</spring-cloud.version>
21    </properties>
```

5) 拷贝文件到idea，并修改类型为pom

```
<packaging>pom</packaging>
```

### 3.4.3 web子模块

1) 使用maven命令行工具，创建maven项目

#操作演示：打开dos或者idea的terminal，或者gitbash等工具，进入交互模式，逐步创建项目

```
mvn archetype:generate
```

#也可以一步到位直接指定参数

```
mvn archetype:generate -DgroupId=com.itheima.logdemo -DartifactId=web -
-DpackageName=com.itheima.logdemo.web -DarchetypeArtifactId=maven-archetype-
quickstart -DinteractiveMode=false
```

2) 添加cloud代码：

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class App {
    public static void main(String[] args) {
        new SpringApplicationBuilder(App.class).run(args);
    }
}
```

3) 增加配置文件：bootstrap.properties

```
#服务名
spring.application.name=web
#端口号
server.port=8002
#配置中心url
spring.cloud.nacos.config.server-addr=39.98.133.153:9105
#注册中心
spring.cloud.nacos.discovery.server-addr=39.98.133.153:9105
```

### 3.4.4 user子模块

- 1) mvn步骤同上
- 2) 增加配置文件: bootstrap.properties

```
#服务名
spring.application.name=user
#端口号
server.port=8003
#配置中心url
spring.cloud.nacos.config.server-addr=39.98.133.153:9105
#注册中心
spring.cloud.nacos.discovery.server-addr=39.98.133.153:9105
```

### 3.4.5 工具包utils

- 1) mvn步骤同上
- 2) 引入常用坐标

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.62</version>
</dependency>
```

- 3) 定义LogBean, 记录日志信息

```
package com.itheima.logdemo.utils;

import com.alibaba.fastjson.JSON;

public class LogBean {
    private String rid,sid,tid,from,message;
```

```

    public LogBean(String rid, String sid, String tid, String from, String
message) {
        this.rid = rid;
        this.sid = sid;
        this.tid = tid;
        this.from = from;
        this.message = message;
    }

    //getter and setter...

    @Override
    public String toString() {
        return JSON.toJSONString(this);
    }
}

```

4) web和user引入ustils的坐标

### 3.4.6 logback-kafka

1) 以web项目为例，springboot默认使用logback做日志，添加kafka依赖

<https://github.com/danielwegener/logback-kafka-appender>

```

<dependency>
    <groupId>com.github.danielwegener</groupId>
    <artifactId>logback-kafka-appender</artifactId>
    <version>0.2.0-RC2</version>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.2.2</version>
</dependency>

```

2) logback配置文件，配置前需要先从km创建demo topic，并配置logstash配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="30 seconds" debug="false">
    <!-- LOGGER PATTERN 根据个人喜好选择匹配 -->
    <property name="logPattern"
        value="logback:[ %-5level] [%date{HH:mm:ss.SSS}] %logger{96}
[%line] [%thread]- %msg%n"></property>

    <!-- 控制台的标准输出 -->

```

```

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <charset>UTF-8</charset>
    <pattern>${logPattern}</pattern>
  </encoder>
</appender>

<!-- This example configuration is probably most unreliable under failure
conditions but wont block your application at all -->
<appender name="kafka"
class="com.github.danielwegener.logback.kafka.KafkaAppender">
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <pattern>%msg%n</pattern>
  </encoder>
  <topic>demo</topic>
  <!-- we don't care how the log messages will be partitioned -->
  <keyingStrategy
class="com.github.danielwegener.logback.kafka.keying.NoKeyKeyingStrategy"/>
  <!-- use async delivery. the application threads are not blocked by
logging -->
  <deliveryStrategy
class="com.github.danielwegener.logback.kafka.delivery.AsynchronousDeliveryStra
tegy"/>
  <!-- each <producerConfig> translates to regular kafka-client config
(format: key=value) -->
  <!-- producer configs are documented here:
https://kafka.apache.org/documentation.html#newproducerconfigs -->
  <!-- bootstrap.servers is the only mandatory producerConfig -->
  <producerConfig>bootstrap.servers=39.98.133.153:9103</producerConfig>
  <!-- don't wait for a broker to ack the reception of a batch. -->
  <producerConfig>acks=0</producerConfig>
  <!-- wait up to 1000ms and collect log messages before sending them as
a batch -->
  <producerConfig>linger.ms=1000</producerConfig>
  <!-- even if the producer buffer runs full, do not block the
application but start to drop messages -->
  <producerConfig>max.block.ms=0</producerConfig>
  <!-- define a client-id that you use to identify yourself against the
kafka broker -->
  <producerConfig>client.id=${HOSTNAME}-${CONTEXT_NAME}-logback-
relaxed</producerConfig>
  <!-- there is no fallback <appender-ref>. If this appender cannot
deliver, it will drop its messages. -->
</appender>

<logger name="kafka">
  <appender-ref ref="kafka"/>
</logger>

```



```

    <root level="INFO">
        <appender-ref ref="STDOUT"/>
    </root>

</configuration>

```

3) 启动web，测试log日志通道，进kibana查看日志输出情况

```

private final Logger logger = LoggerFactory.getLogger("kafka");

@GetMapping("/test")
public Object test(){
    logger.info("this is a test");
    return "this is a test";
}

```

## 3.4.7 手写KafkaAppender

1) utils项目的pom中引入spring-kafka

```

<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.2.2</version>
</dependency>

```

2) utils中定义kafka appender

```

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.core.AppenderBase;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;

import java.util.HashMap;
import java.util.Map;

public class KafkaAppender extends AppenderBase<ILoggingEvent> {

```

```

//定义属性, 可以从logback.xml配置文件中获取
private String topic,brokers;

private KafkaTemplate kafkaTemplate;

@Override
public void start() {
    Map<String, Object> props = new HashMap<>();
    //连接地址
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers);
    //重试, 0为不启用重试机制
    props.put(ProducerConfig.RETRIES_CONFIG, 1);
    //控制批处理大小, 单位为字节
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
    //批量发送, 延迟为1毫秒, 启用该功能能有效减少生产者发送消息次数, 从而提高并发量
    props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
    //生产者可以使用的最大内存字节来缓冲等待发送到服务器的记录
    props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 1024000);
    //键的序列化方式
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
    //值的序列化方式
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    this.kafkaTemplate = new KafkaTemplate<Integer, String>(new
DefaultKafkaProducerFactory<>(props));
    super.start();
}

@Override
public void stop() {
    super.stop();
}

@Override
protected void append(ILoggingEvent iLoggingEvent) {
    kafkaTemplate.send(topic,iLoggingEvent.getMessage());
}

public String getTopic() {
    return topic;
}

public void setTopic(String topic) {
    this.topic = topic;
}

public String getBrokers() {

```

```
        return brokers;
    }

    public void setBrokers(String brokers) {
        this.brokers = brokers;
    }
}
```

3) web的logback中配置appender

```
<appender name="kafka" class="com.itheima.logdemo.utils.KafkaAppender">
    <topic>demo</topic>
    <brokers>39.98.133.153:9103</brokers>
</appender>
```

## 3.5总结

---

- 1) 日志收集环境的一些设计目标
- 2) 前期准备工作，搭建基础微服务框架
- 3) 需要注意，springcloud默认使用logback做日志
- 4) 两种方式展示如何在cloud环境中集成kafka和中间件平台
- 5) 测试日志的完整通道