分布式链路跟踪介绍

前几节课已经讲过几种监控微服务的方式，例如使用 spring Boot Actuator监控微服务实例，使用 Hystrix监控Hystrix Command等，本节课来讨论微服务“跟踪"

大家先看几个问题，对于一个大型的微服务架构系统，会有哪些**常见问题**?

如何串联调用链，快速定位问题

如何厘清微服务之间的依赖关系

如何进行各个服务接口的性能分折

如何跟踪业务流的处理顺序

Sleuth介绍及应用

spring Cloud Sleuth为 spring Cloud提供了分布式跟踪的解决方案，它大量借用了Google Dapper、 Twitter Zipkin和 Apache HTrace的设计

先来了解一下 Sleuth的术语， Sleuth借用了 Dapper的术语。

- span（跨度）：基本工作单元。 span用一个64位的id唯一标识。除ID外，span还包含其他数据，例如描述、时间戳事件、键值对的注解（标签）， spanID、span父 ID等。 span被启动和停止时，记录了时间信息。初始化 span被称为"rootspan"，该 span的 id和 trace的 ID相等。
- trace（跟踪）：一组共享"rootspan"的 span组成的树状结构称为 traceo trace也用一个64位的 ID唯一标识， trace中的所有 span都共享该 trace的 IDO
- annotation（标注）： annotation用来记录事件的存在，其中，核心annotation用来定义请求的开始和结束。

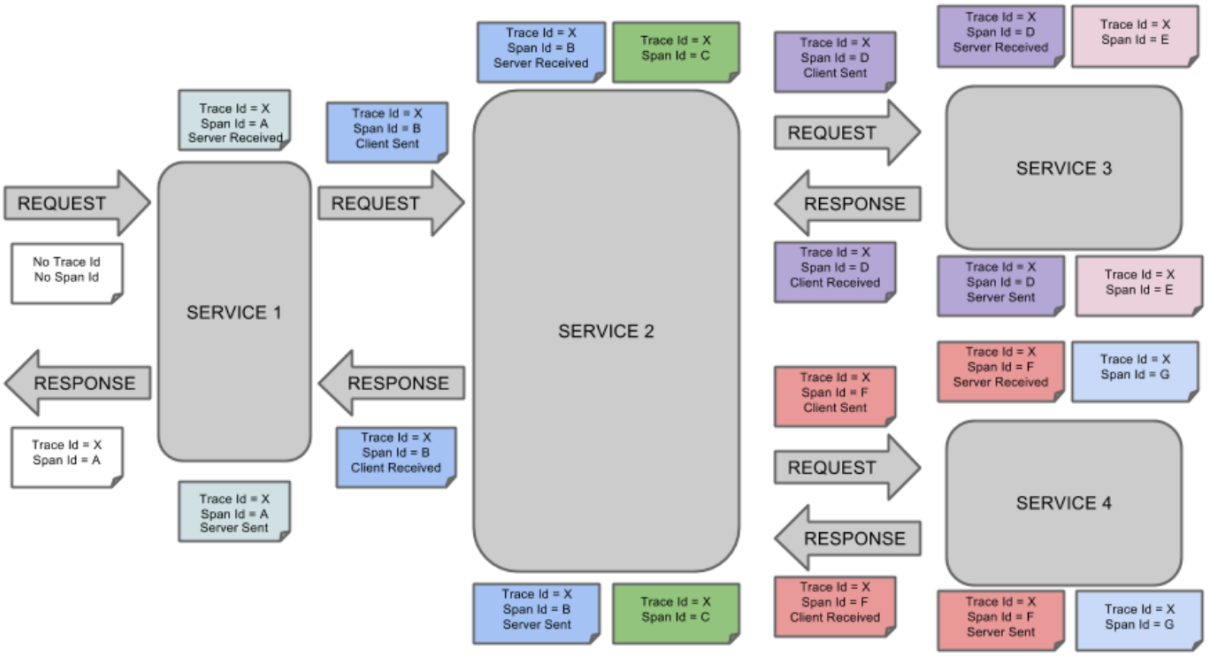
a) CS（ Client sent客户端发送）：客户端发起一个请求，该 annotation描述了span的开始。

b) SR（ server Received服务器端接收）：服务器端获得请求并准备处理它。如果用 SR减去 CS时间戳，就能得到网络延迟。

c) SS（ server sent服务器端发送）：该 annotation表明完成请求处理（当响应发回客户端时）。如果用 SS减去 SR时间戳，就能得到服务器端处理请求所需的时间。

d) CR（ Client Received客户端接收）： span结束的标识。客户端成功接收到服务器端的响应。如果 CR减去 CS时间戳，就能得到从客户端发送请求到服务器响应的所需的时间。

下图详细描述了请求依次经过Service1--Service2--Service3--Service4时，span、trace、annotation的变化



Sleuth整合Zipkin实现分布式链路跟踪

整合sleuth

见示例：11-ms-simple-provider-user-trace

添加依赖

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

启动项目，访问地址：http://localhost:8000/1，查看后台日志，发现有trace和span的日志打印

```
rameworkServlet 'dispatcherServlet'
patcherServlet': initialization started
patcherServlet': initialization completed in 41 ms
 uri [/1] that should not be sampled [false]
.TraceFilter  : No parent span present - creating a new span
nterceptor    : Handling span [Trace: 8be14bc165469e11, Span: 8be14bc165469e11, Parent: null, exportable:false]
nterceptor    : Adding a method tag with value [findById] to a span [Trace: 8be14bc165469e11, Span: 8be14bc165469e11, Parent: null, exportable:false]
nterceptor    : Adding a class tag with value [UserController] to a span [Trace: 8be14bc165469e11, Span: 8be14bc165469e11, Parent: null, exportable:false]
5_0_0_ from user user0_ where user0_.id=?
.TraceFilter  : Closing the span [Trace: 8be14bc165469e11, Span: 8be14bc165469e11, Parent: null, exportable:false] since the response was successful
```

Zipkin简介

Zipkin是 Twitter开源的分布式跟踪系统，基于 Dapper的论文设计而来。它的主要功能是收集系统的时序数据，从而追踪微服务架构的系统延时等问题。 Zipkin还提供了一个非常友好的界面，来帮助分析追踪数据。官网地址：http://zipkin.io

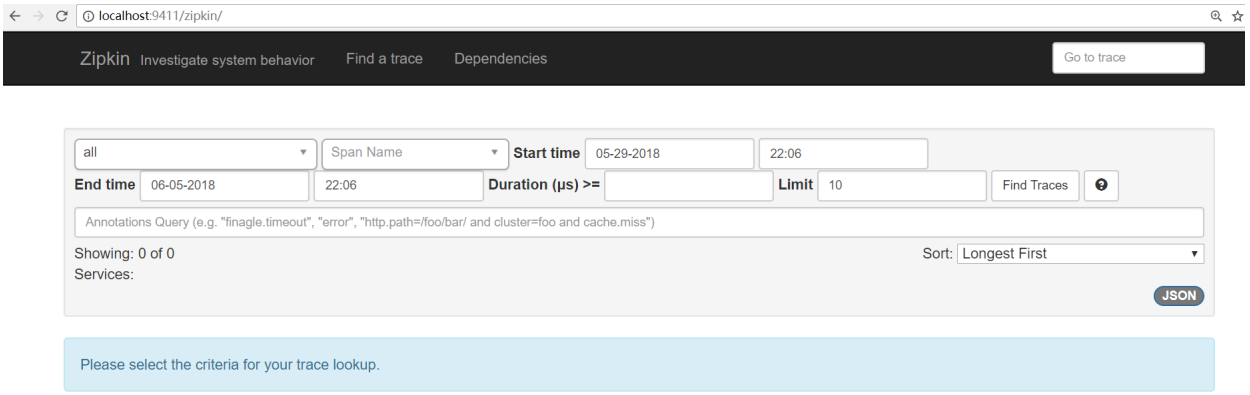
编写Zipkin Server

见示例：11-ms-trace-zipkin-server

添加依赖，并在启动类上添加注解@EnableZipkinServer，声明一个Zipkin Server

```
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-server</artifactId>
</dependency>
```

启动项目，访问地址：http://localhost:9411/zipkin/，效果如下图



简单讲解下图中各个查询条件的含义：

第一列表示**Service Name**，也就是各个微服务spring.application.name的值。第二列表示**Span**的名称，all表示所有。**Start time**和**End time**，分别用于指定起始时间和截止时间。**Duration**表示持续时间，即**Span**从创建到关闭所经历的时间。**Limit**表示查询几条数据。类似于 MySQL数据库中的 limit关键词。**Annotations Query**，用于自定义查询条件。

微服务整合Zipkin

用户微服务整合Zipkin

见示例：11-ms-simple-provider-user-trace-zipkin

添加主要依赖

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

在配置文件中新增如下内容

```yaml
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
```

spring.zipkin.base-url：指定Zipkin的地址。

spring.sleuth.sampler.percentage：指定需采样的请求的百分比，默认值是0.1，即10%。这是因为在分布式系统中，数据量可能会非常大，因此采样非常重要。但是我们示例数据少最好配置为1全采样

订单微服务整合Zipkin类似，见示例：11-ms-simple-consumer-order-trace-zipkin

启动这两个项目，再启动Zipkin服务，访问订单微服务：http://localhost:8010/user/1，然后再次查看Zipkin服务：http://localhost:9411/zipkin/，能查询到微服务调用的跟踪日志

用Elasticsearch存储Zipkin的数据

见示例：11-ms-trace-zipkin-server-elasticsearch

添加依赖

```xml
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-storage-elasticsearch-http</artifactId>
  <version>2.3.1</version>
</dependency>
```

配置文件application.yml里增加elasticsearch连接配置

```yaml
server:
  port: 9411
zipkin:
  storage:
    type: elasticsearch
    elasticsearch:
      cluster: elasticsearch
      hosts: http://localhost:9200
      index: zipkin
      index-shards: 5
      index-replicas: 1
```

启动项目，访问一次订单服务：http://localhost:8010/user/1

查看elasticsearch后端数据是否存储成功：http://localhost:9200/_search

补充

1、spring boot admin监控邮件发送

见示例08-ms-spring-boot-admin和08-ms-provider-user

在08-ms-spring-boot-admin的依赖里加入

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
    <version>1.5.9.RELEASE</version>
</dependency>
```

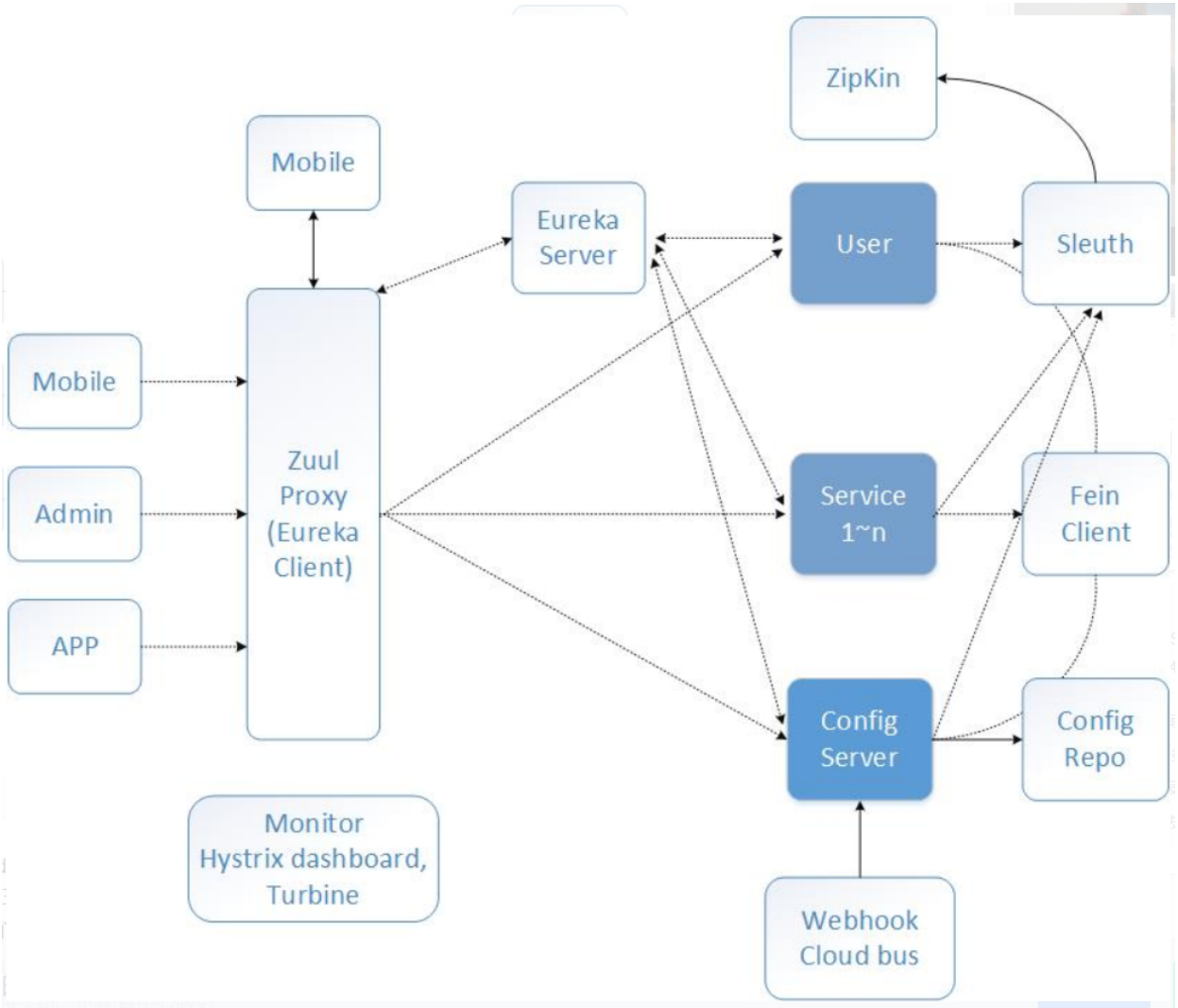
配置文件application.yml内容如下

```
server:
  port: 9999
spring:
  application:
    name: service-admin
  mail:
    host: smtp.163.com
    username: aaronrao2018@163.com
    password: 2018shijiebei
    properties:
      mail.debug: false
      mail.smtp.auth: true
  boot:
    admin:
      notify:
        mail:
          to: 3376224996@qq.com
          from: aaronrao2018@163.com
#         ignore-changes: UNKNOWN:UP
      routes:
        endpoints: env,metrics,dump,jolokia,info,configprops,trace,logfile,refresh,flyway,liquibase,heapdump,loggers,auditevents,hystrix.stream
```

2、hystrix的历史数据监控方案

- 修改源码类HystrixSampleSseServlet.java将历史数据存入mq异步分析处理
- 利用一些第三方工具访问hystrix.stream接口拿到实时数据分析处理

3、spring cloud 整体架构图



从上图可以看出Spring Cloud各个组件相互配合，合作支持了一套完整的微服务架构。

- 其中Eureka负责服务的注册与发现，很好将各服务连接起来
- Hystrix 负责监控服务之间的调用情况，连续多次失败进行熔断保护。
- Hystrix dashboard,Turbine 负责监控 Hystrix的熔断情况，并给予图形化的展示
- Spring Cloud Config 提供了统一的配置中心服务
- 当配置文件发生变化的时候，Spring Cloud Bus 负责通知各服务去获取最新的配置信息
- 所有对外的请求和服务，我们都通过Zuul来进行转发，起到API网关的作用
- 监控我们使用Sleuth+Zipkin+springAdmin将所有的请求数据记录下来，方便我们进行后续分析

Spring Cloud从设计之初就考虑了绝大多数互联网公司架构演化所需的功能，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等。这些功能都是以插拔的形式提供出来，方便我们系统架构演进的过程中，可以合理的选择需要的组件进行集成，从而在架构演进的过程中会更加平滑、顺利。