Spring Cloud Sleuth：分布式服务跟踪

- 对于每个请求， 全链路调用的跟踪就变得越来越重要， 通过实现对请求调用的跟踪可以帮助我们快速发现错误根源以及监控分析每条请求链路上的性能瓶颈等

应用

- 服务注册中心: eureka-server
- 微服务应用: trace-1, 实现一 个 REST 接口 /trace-1, 调用该接口后将触发对trace-2 应用的调用
  - `web`、`eureka`、`ribbon`

```
@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class TraceApplication {
  private final Logger logger = Logger.getLogger(getClass());
  @Bean
  @LoadBalanced
  RestTemplate restTemplate() {
    return new RestTemplate();
  }
  @RequestMapping(value = "/trace-1", method = RequestMethod.GET)
  public String trace() {
    logger.info("===call trace-1===");
    return restTemplate().getForEn七ity("http://trace-2/trace-2",
String.class).getBody();
  }
  public static void main(String[] args) {
    SpringApplication.run(TraceApplication.class, args);
  }
```

```
# application.properties
spring.application.name=trace-1
server.port= 9101
eureka.client.serviceUrl.defaultZone=http://localhost:lll0/eureka/
```

- 微服务应用: trace-2, 实现一个 REST 接口/trace-2, 供 trace-1 调用。

```
@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class TraceApplication {
  private final Logger logger = Logger.getLogger(getClass());
```

```
  @RequestMapping(value = "/trace-2", method = RequestMethod.GET)
  public String trace() {
    logger.info("===<call trace-2>===");
    return "Trace";
  }
  public static void main(String[] args) {
    SpringApplication.run(TraceApplication.class, args);
  }
}
```

```
spring.application.name=trace-2
server.port=9102
eureka.client.serviceOrl.defaultZone= http://localhost:lll0/eureka/
```

- 实现跟踪
  - 依赖管理中增加spri ng-cloud-starter-sleuth 依赖即可
  - 形如[trace-1,f410ab57afd5c145, a9f2118fa2019684, false]的日志信息，实现分布式服务跟踪的重要组成部分
    - 第一个值: trace-1, 它记录了应用的名称，也就是 application.properes中spring.application.name参数配置的属性
    - 第二个值:f410ab57afd5c145，SpringCloudSleuth生成的一个ID,称为TraceID，它用来标识一条请求链路。 一条请求链路中包含一个TraceID, 多个SpanID
    - 第三个值: a9f2118fa2019684，Spring Cloud Sleuth生成的另外一 个 ID, 称为SpanID, 它表示一个基本的工作单元，比如发送一个HTTP请求
    - 第四个值: false, 表示是否要将该信息 输出到Zipkin等服务中来收集和展示
  - 在一次服务请求链路的调用过程中，会保待并传递同一个Trace ID，从而将整个分布于不同微服务进程中的请求跟踪信息串联起来。 以上面输出内容为例， trace-1 和trace-2同属于一个前端服务请求来源，所以它们的TraceID是相同的，处于同一 条请求链路中
- 跟踪原理
  - 分布式系统中的服务跟踪主要包括下面两个关键点：
    - 为了实现请求跟踪， 当请求发送到分布式系统的入口端点时， 只需要服务跟踪框架为该请求创建一个唯一的跟踪标识-TraceID， 同时在分布式系统内部流转的时候， 框架始终保待传递该唯一标识，直到返回给请求方为止。通过TraceID的记录，就能将所有请求过程的日志关联起来
    - 为了统计各处理单元的时间延迟， 当请求到达各个服务组件时， 或是处理逻辑到达 某个状态时，也通过一个唯一标识-SpanID来标记它的开始、 具体过程以及结束。对于每个Span来说，它必须有开始和结束两个节点，通 过记录开始 Span和结束Span的时间戳，就能统计出该Span的时间延迟，除了时间 戳记录之外，它还可以包含一些其他元数据，比如事件名称、 请求信息等
  - 在 Spring Boot 应用中，通过在工程中引入spring-cloud-starter-sleuth依赖之后，它会自动为当前应用构建起各通信通道的跟踪机制，比如：
    1. 通过诸如 RabbitMQ 、 Kafka C 或者其他任何 Spring Cloud Stream 绑定器实现的消息中间件) 传递的请求
    2. 通过 Zuul 代理传递的请求

3. 通过 RestTemplate 发起的请求

- 由于 trace-1对 trace-2发起的请求是通过 RestTemplate实现的， 所以 spring-cloud-starter-sleuth 组件会对该请求进行处理 。 在发送到 trace-2 之前， Sleuth会在该请求的Header中增加实现跟踪需要的重要信息，主要有下面这几个：
  - X-B3-TraceId: 一 条请求链路 (Trace) 的唯 一 标识， 必需的值。
  - X-B3-Spanld: 一个工作单元 (Span) 的唯一 标识， 必需的值。
  - X-B3-ParentSpanld: 标识当前工作单元所属的上一 个工作单元 ， Root Span（请求链路的第一个工作单元) 的该值为空。
  - X-B3-Sampled: 是否被抽样输出的标志， 1 表示需要被输出 ， 0 表示不需要被输出 。
  - X-Span-Name: 工作单元的名称。
  - 可以通过对trace-2 的实现做一些修改来输出这些头部信息

```
@RequestMapping(value = "/trace-2", method =
RequestMethod.GET)
public Stringtrace(HttpServletRequest request) {
  logger.info("===<call trace-2, Traceid={}, Spanid={}>===",
  request.getHeader("X-B3-Traceid"), request.getHeader("X-B3-
Spanid"));
  return "Trace";
}
```

- 为了更直观地观察跟踪信息，还可以在 application.properties 中增加下面的配置：`logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG`，通过将SpringMVC的请求分发日志级别调整为DEBUG级别，可以看到更多跟踪信息

- 抽样收集
  - 通过Trace ID和Span ID已经实现了对分布式系统中的请求跟踪， 而记录的跟踪信息最终会被分析系统收集起来， 并用来实现对分布式系统的监控和分析功能， 比如， 预警延迟过长的请求链路、 查询请求链路的调用明细等
  - 分析系统在收集跟踪信息的时候， 需要收集多少跟踪信息才合适呢?
  - 在 Sleuth 中采用了抽象收集的方式来为跟踪信息打上收集标记， 在日志信息中看到的第4个布尔类型 的值， 它代表了该信息是否要被后续的跟踪信息收集器获取和存储
  - Sleuth 中的抽样收集策略是通过 Sampler 接口实现的
    - 通过实现 isSarnpled 方法，Spring Cloud Sleuth 会在产生跟踪信息的时候调用它来为跟踪信息生成是否要被收集的标志。 需要注意的是， 即使 isSampled 返回了 false, 它仅代表该跟踪信息不被输出到后续对接的远程分析系统(比如 Zipkin), 对于请求的跟踪活动依然会进行， 所以我们在日志中还是能看到收集标识为 false 的记录
    - 默认情况下，Sleuth 会使用 PercentageBasedSarnpler 实现的抽样策略，以请求百分比的方式配置和收集跟踪信息。 可以通过在 application.properties 中配置参数对其百分比值进行设置， 它的默认值为0 .1, 代表收集10%的请求跟踪信息`spring.sleuth.sampler.percentage= 0.1`
    - 也可以通过创建 AlwaysSarnpler 的 Bean (它实现的 isSarnpled 方法始终返回true) 来

覆盖默认的PercentageBasedSarnpler策略
    - 由于跟踪日志信息数据的价值往往仅在最近的一段时间内非常有用， 比如一周。 那么我们在设计抽样策略时， 主要考虑在不对系统造成明显性能影响的情况下， 以在日志保留时间窗内充分利用存储空间的原则来实现抽样策略

与Logstash整合

- 引入基于日志的分析系统帮助集中收集、 存储和搜索这些跟踪信息
    - 比如ELK平台， 它可以轻松地帮助我们收集和存储这些跟踪日志， 同时在需要的时候我们也可以根据Trace ID来轻松地搜索出对应请求链路 相关的明细日志
        - ElasticSearch 是一 个开源分布式搜索引擎， 它的特点有: 分布式， 零配置， 自动发现， 索引自动分片， 索引副本机制，RESTful风格接口， 多数据源， 自动搜索负载等。
        - Logstash是一个完全开源的工具， 它可以对日志进行收集、 过滤， 并将其存储供以后使用
        - Kibana 也是一 个开源和免费的工具， 它可以为Logstash 和ElasticSearch提供日志分析友好的Web界面， 可以帮助汇总、 分析和搜索重要数据日志
    - Spring Cloud Sleuth在与ELK平台整合使用时， 实际上只要实现与负责日志收集的 Logstash 完成数据对接即可， 所以需要为Logstash准备JSON格式的日志输出。
    - 由于 Spring Boot 应用默认使用logback来记录日志，而Logstash自身也有对logback日志工具的支持工具， 所以可以直接通过在logback的配置中增加对Logstash的Appender, 就能 非常方便地将日志转换成以JSON的格式存储和输出了
- 引入依赖

```
<dependency>
  <groupid>net.logstash.logback</groupid>
  <artifactid>logstash-logback-encoder</artifactid>
  <version>4.6</version>
</dependency>
```

- 在工程/resource 目录下创建 bootstrap.properties 配置文件，将 spring.application.name=trace-1 配置移动到该文件中去
    - 由于 logback-spring.xml 的加载在 application.properties 之前， 所以之前的配置 logback-spring.xml 无法获取 spring.application.name 属性， 因此这里将该属性移动到最先加载的 bootstrap.properties 配置文件中
- 在工程/resource 目录下创建 logback 配置文件 logback-spring.xml

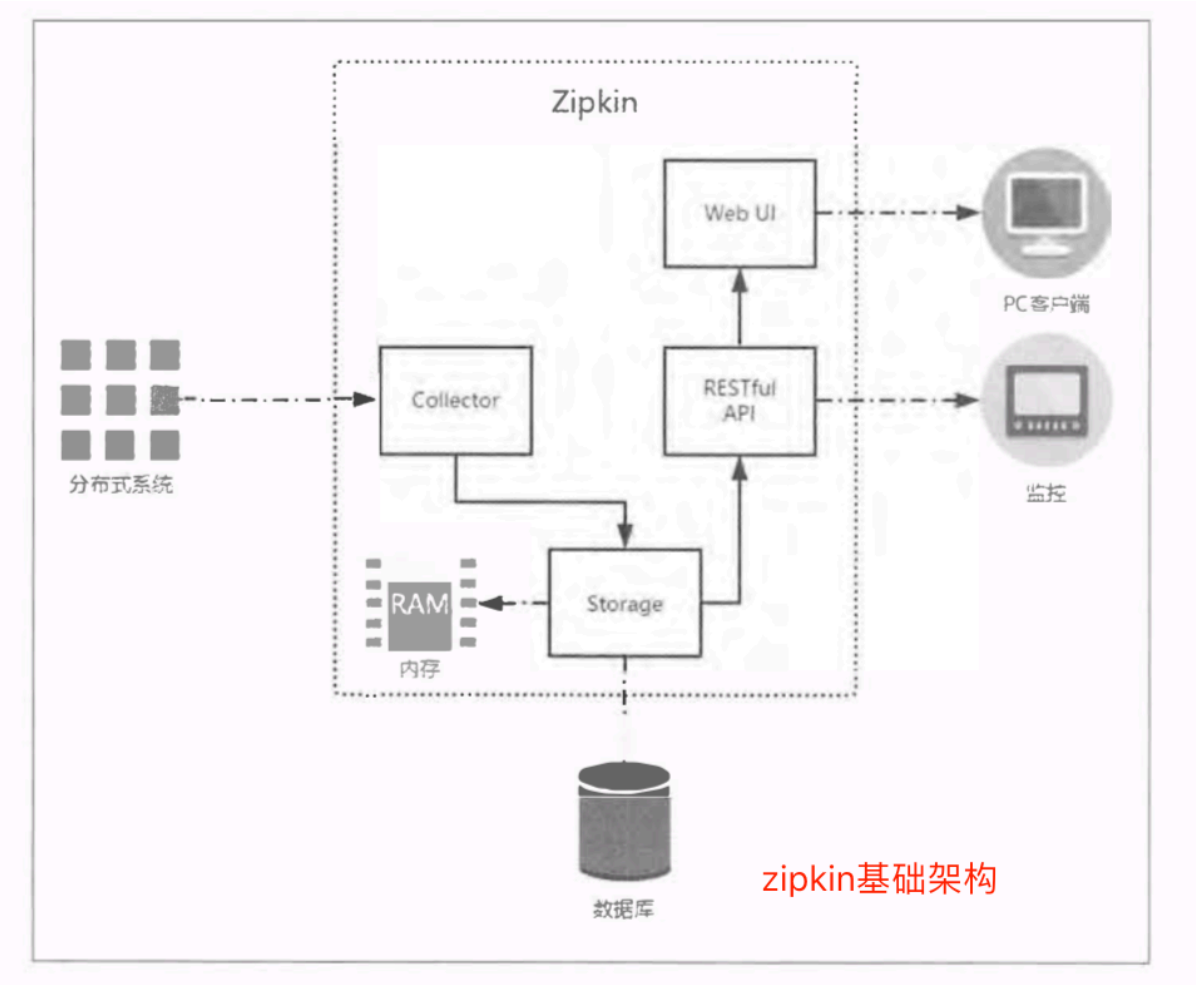
```

```

- 在 trace-1 和 trace-2 的工程目录下发现有一个 build 目录， 下面分别创建了以各自应用名称命名的 JSON 文件， 该文件就是在 logback-spring.xml中配置的名为 logs七ash 的 Appender 输出的日志文件，其中记录了类似下面格式的 JSON日志
- 除了可以通过上面的方式生成 JSON 文件外， 还可以使用 LogstashTcpSocketAppender 将日志内容直接通过 Tep Socket 输出到 Logstash 服务端

与Zipkin整合

- 在 ELK 平台中的数据分析维度缺少对请求链路中各阶段时间延迟的关注
- Zipkin是Twitter的一个开源项目， 它基于 Google Dapper 实现。可以使用它来收集各个服务器上请求链路的跟踪数据， 并通过它提供的 REST API 接口来辅助查询跟踪数据以实现对分布式系统的监控程序， 从而及时发现系统中出现的延迟升高问题并找出系统性能瓶颈的根源
- 除了面向开发的 API 接口之外， 它还提供了方便的 UI 组件来帮助我们直观地搜索跟踪信息和分析请求链路明细， 比如可以查询某段时间内各用户请求的处理时间等



  - Collector: 收集器组件， 它主要处理从外部系统发送过来的跟踪信息， 将这些信息转换为Zipkin内部处理的Span格式，以支持后续的存储、分析、展示等功能
  - Storage: 存储组件，它主要处理收集器接收到的跟踪信息， 默认会将这些信息存储在内存中。 我们也可以修改此存储策略， 通过使用其他存储组件将跟踪信息存储到数据库中
  - RESTfulAPI: API组件， 它主要用来提供外部访问接口。 比如给客户端展示跟踪信息， 或是外接系统访问以实现监控等
  - WebUI: UI组件， 基于API组件实现的上层应用。 通过UI组件， 用户可以方便而又直观地查询和分析跟踪信息
- HTTP收集
  - 在Spring CloudSleuth中对Zipkin的整合进行了自动化配置的封装， 所以我们可以很轻松地引入和使用它
  - Sleuth与Zipkin的基础整合过程
    - 第一步，搭建Zipkin Server

        - `zipkin-server`工程，主类 @EnableZipkinServer注解来启动ZipkinServer

```
<dependency>
  <groupid>io.zipkin.java</groupid>
  <artifactid>zipkin-server</ar七ifactid>
</dependency>
<dependency>
  <groupid>io.zipkin.java</groupid>
  <artifactid>zipkin-autoconfigure-ui</artifactid>
</dependency>
```

```
# application.properties
# 设置服务端口号为9411(客户端整合时，自动化配置会连接9411端口，所以在服务端设置了端口为9411的话，客户端可以省去这个配置
spring.application.name=zipkin-server
server.port=9411
# 访问
http://localhost:9441/
```

    - 第二步，为应用引入和配置Zipkin服务
        - 对应用做一些配置，以实现将跟踪信息输出到Zipkin Server。
            - `spring.zipkin.base-url=http://localhost:9411`

```
<dependency>
  <groupid>org.springframework.cloud</groupid>
  <artifactid>spring-cloud-sleuth-zipkin</artifactid>
</dependency>
```

    - 测试与分析
        - 向trace-1的接口发送几个请求http://localhost: 9101/trace-1
        - 当在日志中出现 跟踪信息的最后一个值为true的时候， 说明该跟踪信息会输出给ZipkinServer, 所以此时可以在ZipkinServer的管理页面中选择合适的查询条件，单击FindTraces按钮， 就可以查询出刚才在日志中出现的跟踪信息了(也可以根据日志中的TraceID, 在页面右上角的输入框中来搜索)
        - 单击下方trace-1端点的跟踪信息， 还可以得到Sleuth 跟踪到的详细信息， 其中包括我们关注的请求时间消耗等
        - 单击导航栏中的Dependencies菜单， 还可以查看ZipkinServer根据跟踪信息分析生成的系统请求链路依赖关系图，

消息中间件收集

- Spring Cloud Sleuth 在整合 Zipkin 时，不仅实现了以 HTTP 的方式收集跟踪信息，还实现了通过消息中间件来对跟踪信息进行异步收集的封装。 通过结合 Spring Cloud Stream，我们可以非常轻松地让应用客户端将跟踪信息输出到消息中间件上， 同时 Zipkin 服务端从消息中间件上异步地消费这些跟踪信息

- 第一步，客户端配置
  - `spring-cloud-starter-sleuth`、`spring-cloud-sleuth-stream`、`spring-cloud-starter-stream-rabbit`
  - 在 application.properties 配置中去掉 HTTP 方式实现时使用的 spring.zipkin.base-url 参数， 并根据实际部署情况，增加消息中间件的相关配置，比如， RabbitMQ 的配置信息
- 第二步，zipkin-server 服务端
  - `spring-cloud-sleuth-zipkin-stream`(实现从消息中间件收集跟踪信息的核心封装， 其中包含了用千整合消息中间件的核心依赖、Zipkin服务端的核心依赖，以及一些其他通常会被使用的依赖)、`spring-cloud-starter-stream-rabbit`、`zipkin-autoconfigure-ui`
- 测试，通过向trace-1的接口发送几个请求`http://localhost:9101/trace-1`