

Spring Cloud Sleuth：分布式请求链路跟踪

摘要

Spring Cloud Sleuth 是分布式系统中跟踪服务间调用的工具，它可以直观地展示出一
次请求的调用过程，下面将对其用法进行详细介绍。

Spring Cloud Sleuth 简介

随着我们的系统越来越庞大，各个服务间的调用关系也变得越来越复杂。当客户端发起一个请求时，这个请求经过多个服务后，最终返回了结果，经过的每一个服务都有可能发生延迟或错误，从而导致请求失败。这时候我们就需要请求链路跟踪工具来帮助我们，理清请求调用的服务链路，解决问题。

给服务添加请求链路跟踪

我们将通过user-service和ribbon-service之间的服务调用来演示该功能，这里我们调用ribbon-service的接口时，ribbon-service会通过RestTemplate来调用user-service提供的接口。

- 首先给user-service和ribbon-service添加请求链路跟踪功能的支持；
- 在user-service和ribbon-service中添加相关依赖：

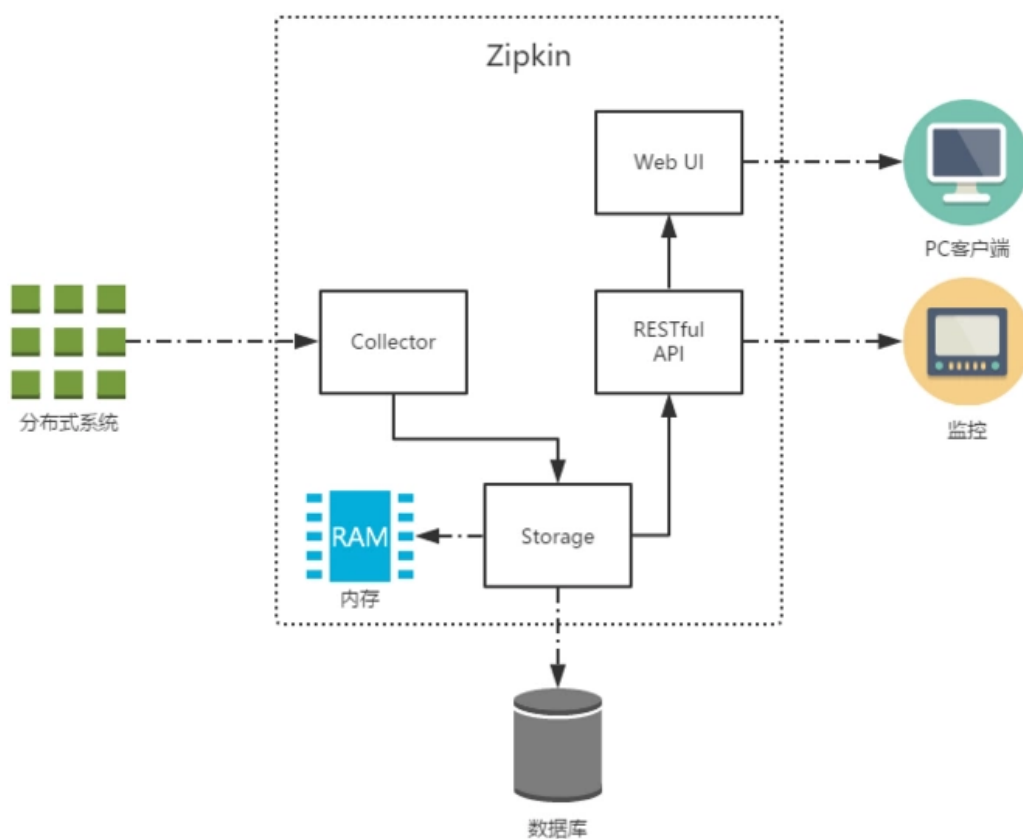
```
1 <!--使用分布式请求链路追踪时添加-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-zipkin</artifactId>
5 </dependency>
```

- 修改application.yml文件，配置收集日志的zipkin-server访问地址：

```
1 spring:
2   zipkin:
3     base-url: http://localhost:9411
4   sleuth:
5     sampler:
```

整合Zipkin获取及分析日志

Zipkin是Twitter的一个开源项目，它基于Google Dapper实现。我们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接口来辅助我们查询跟踪数据以实现分布式系统的监控程序，从而及时地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，它也提供了方便的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。



上图展示了Zipkin的基础架构，它主要有4个核心组件构成：

- Collector：收集器组件，它主要用于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，它主要对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。
- **RESTful API**: 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。
- **Web UI**: 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有直观地查询和分析跟踪信息。

安装使用

- SpringBoot 2.0以上版本已经不需要自行搭建zipkin-server，我们可以从该地址下载zipkin-server:

<https://repo1.maven.org/maven2/io/zipkin/java/zipkin-server/2.12.9/zipkin-server-2.12.9-exec.jar>

- 下载完成后使用以下命令运行zipkin-server:

```
1 java -jar zipkin-server-2.12.9-exec.jar
```

[illegible]

- Zipkin页面访问地址: <http://localhost:9411>

服务名: all Span名称: all 时间: 1 小时

根据Annotation查询: For example: http.path=/foo/bar/ and cluster=foo and cache: 持续时间 (μs) >= Ex: 100ms or 5s 数量: 10 排序: 耗时降序

查找 ?

Showing: 5 of 5 Services: all JSON

- 启动eureka-sever, ribbon-service, user-service:

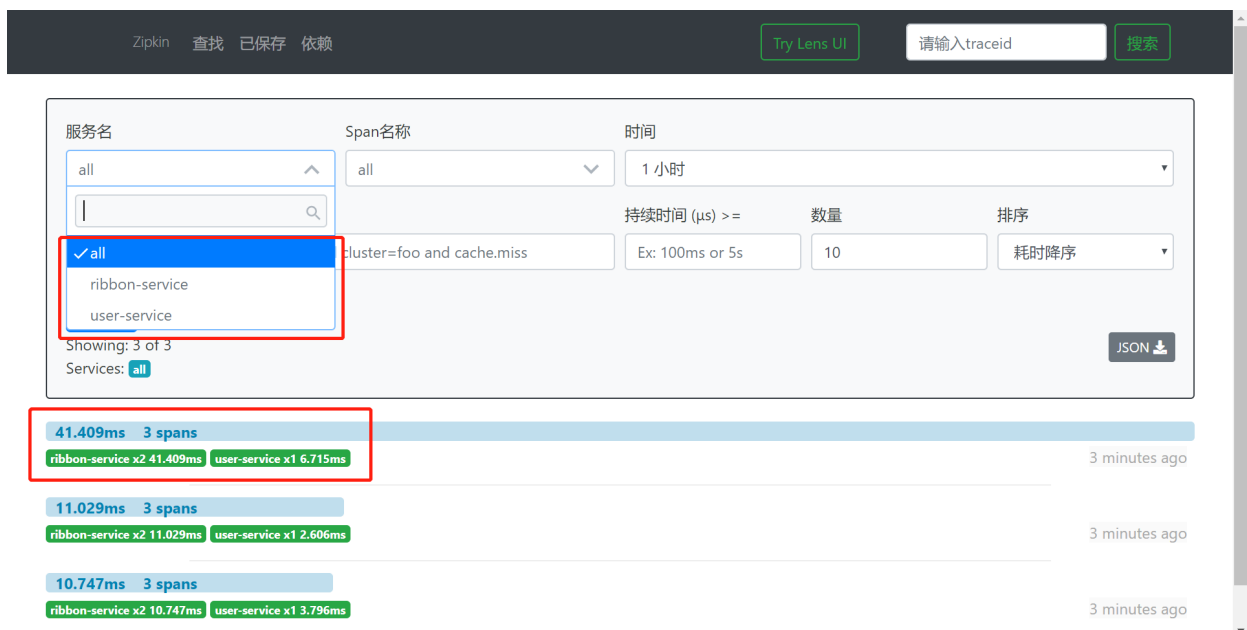
服务名: all Span名称: all 时间: 1 小时

根据Annotation查询: For example: http.path=/foo/bar/ and cluster=foo and cache: 持续时间 (μs) >= Ex: 100ms or 5s 数量: 10 排序: 耗时降序

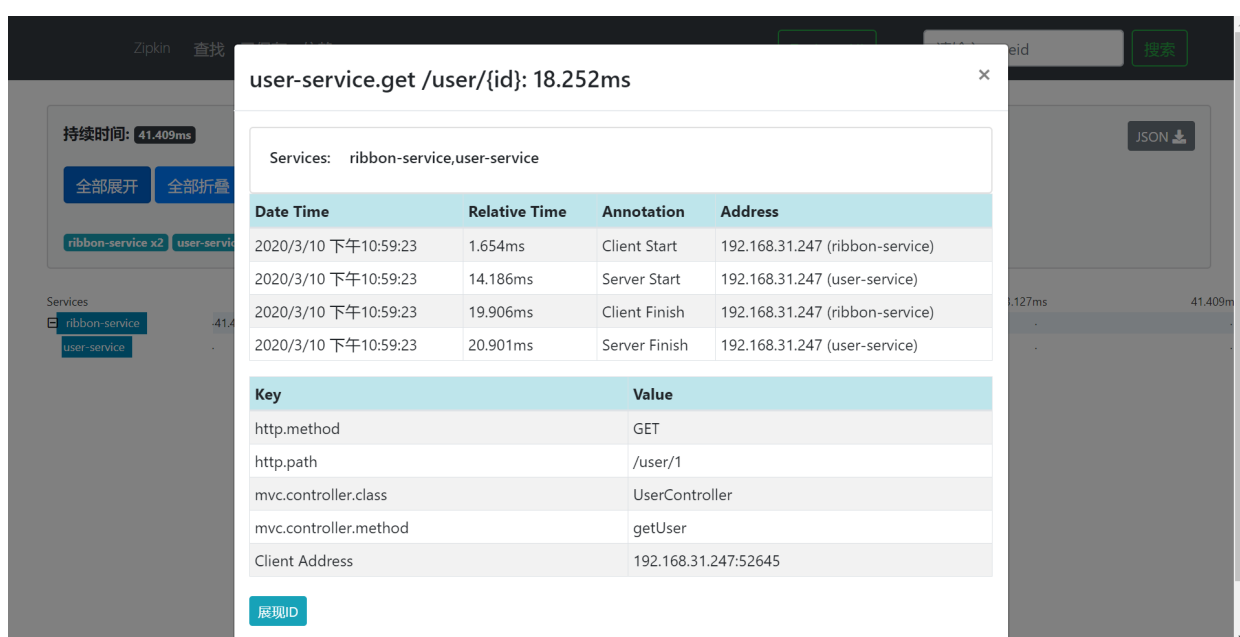
查找 ?

Showing: 5 of 5 Services: all JSON

- 多次调用 (Sleuth为抽样收集) ribbon-service的接口 <http://localhost:8301/user/1> , 调用完后查看Zipkin首页发现已经有请求链路跟踪信息了;



- 点击查看详情可以直观地看到请求调用链路和通过每个服务的耗时：

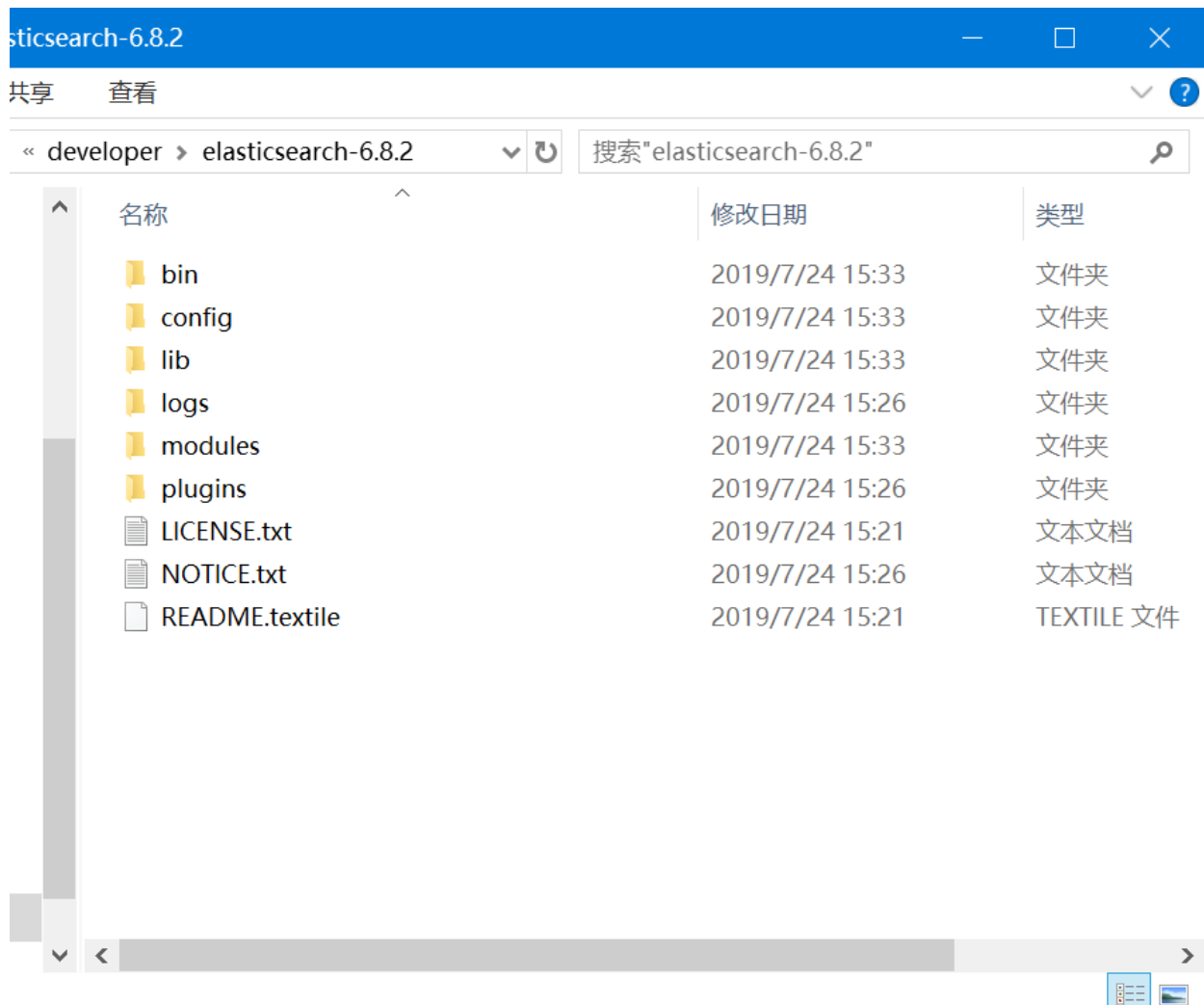


使用Elasticsearch存储跟踪信息

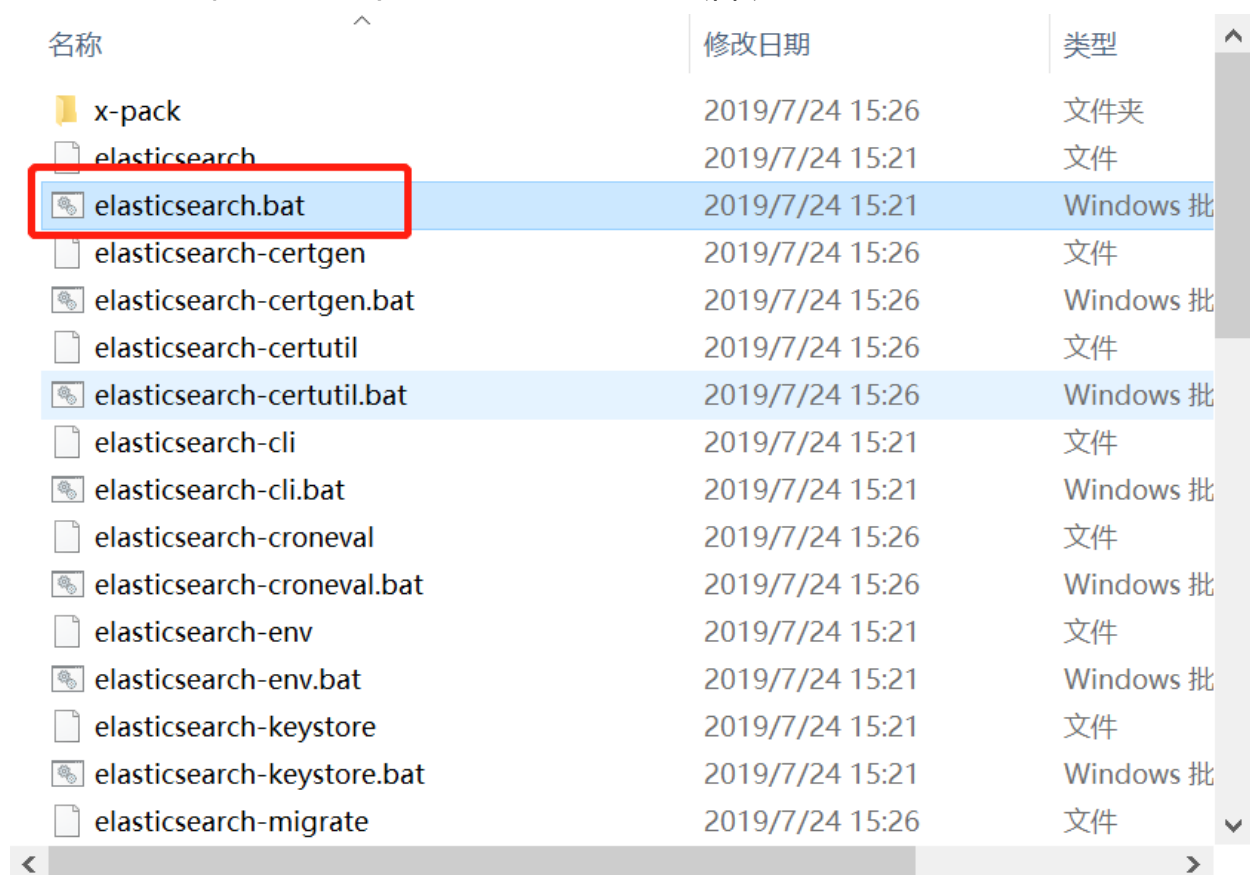
如果我们把zipkin-server重启一下就会发现刚刚的存储的跟踪信息全部丢失了，可见其是存储在内存中的，有时候我们需要将所有信息存储下来，这里以存储到Elasticsearch为例，来演示下该功能。

安装Elasticsearch

- 下载Elasticsearch6.8.2的zip包，并解压到指定目录，下载地址：
<https://www.elastic.co/cn/downloads/past-releases/elasticsearch-6-2-2>



- 运行bin目录下的elasticsearch.bat启动Elasticsearch



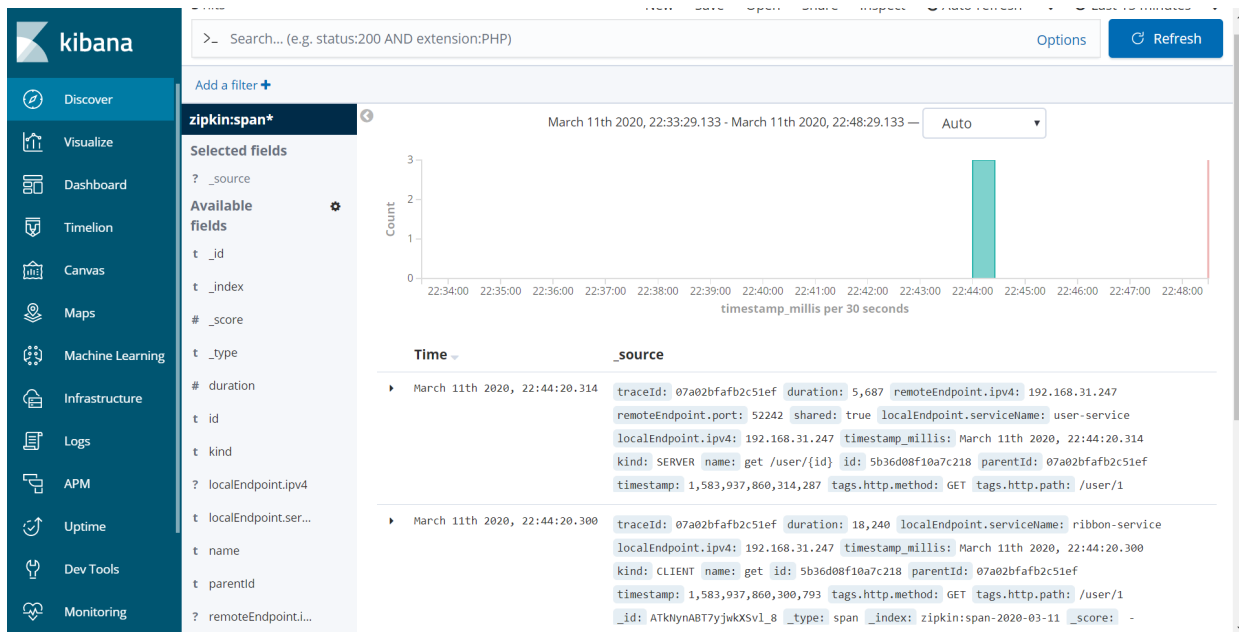
```
C:\WINDOWS\system32\cmd.exe
8480653312, xpack.installed=true, ml.max_open_jobs=20, ml.enabled=true}
[2020-03-10T23:13:03,585][INFO ][o.e.c.s.ClusterApplierService] [3tbWPa0] new_master {3tbWPa0} {3tbWPa0FQpCSUS7yiWqJVw} {7PNSbadrRASObC1bCt5uPw} {127.0.0.1} {127.0.0.1:9300} {ml.machine_memory=8480653312, xpack.installed=true, ml.max_open_jobs=20, ml.enabled=true}, reason: apply cluster state (from master [master {3tbWPa0} {3tbWPa0FQpCSUS7yiWqJVw} {7PNSbadrRASObC1bCt5uPw} {127.0.0.1} {127.0.0.1:9300} {ml.machine_memory=8480653312, xpack.installed=true, ml.max_open_jobs=20, ml.enabled=true} committed version [1] source [zen-disco-elected-as-master ([0] nodes joined)])
[2020-03-10T23:13:03,705][WARN ][o.e.x.s.a.s.m.NativeRoleMappingStore] [3tbWPa0] Failed to clear cache for realms [[]]
[2020-03-10T23:13:03,798][INFO ][o.e.g.GatewayService] [3tbWPa0] recovered [0] indices into cluster_state
[2020-03-10T23:13:04,023][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.triggered_watches] for index patterns [.triggered_watches*]
[2020-03-10T23:13:04,167][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.watches] for index patterns [.watches*]
[2020-03-10T23:13:04,256][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.watch-history-9] for index patterns [.watcher-history-9*]
[2020-03-10T23:13:04,310][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.monitoring-logstash] for index patterns [.monitoring-logstash-6-*]
[2020-03-10T23:13:04,389][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.monitoring-es] for index patterns [.monitoring-es-6-*]
[2020-03-10T23:13:04,468][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.monitoring-beats] for index patterns [.monitoring-beats-6-*]
[2020-03-10T23:13:04,547][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.monitoring-alerts] for index patterns [.monitoring-alerts-6-*]
[2020-03-10T23:13:04,610][INFO ][o.e.c.m.MetadataIndexTemplateService] [3tbWPa0] adding template [.monitoring-kibana] for index patterns [.monitoring-kibana-6-*]
[2020-03-10T23:13:04,799][INFO ][o.e.l.LicenseService] [3tbWPa0] license [41fe4784-fc86-46fe-a46f-f85840e09715] mode [basic] - valid
[2020-03-10T23:13:06,868][INFO ][o.e.h.n.Netty4HttpServerTransport] [3tbWPa0] publish_address {127.0.0.1:9200}, bound_addresses {127.0.0.1:9200}, {:::1}:9200
[2020-03-10T23:13:06,869][INFO ][o.e.n.Node] [3tbWPa0] started
```

修改启动参数将信息存储到Elasticsearch

- 使用以下命令运行，就可以把跟踪信息存储到Elasticsearch里面去了，重新启动也不会丢失；

```
1 # STORAGE_TYPE: 表示存储类型 ES_HOSTS: 表示ES的访问地址
2 java -jar zipkin-server-2.12.9-exec.jar --STORAGE_TYPE=elasticsearch --ES_HOSTS=localhost:9200
```

- 之后需要重新启动user-service和ribbon-service才能生效，重启后多次调用ribbon-service的接口<http://localhost:8301/user/1>;
- 如果安装了Elasticsearch的可视化工具Kibana的话，访问：<http://localhost:5601>，可以看到里面已经存储了跟踪信息：



更多启动参数参考

<https://github.com/openzipkin/zipkin/tree/master/zipkin-server#elasticsearch-storage>

Elasticsearch Storage

Zipkin's [Elasticsearch storage component](#) supports versions 5-7.x and applies when `STORAGE_TYPE` is set to `elasticsearch`.

The following apply when `STORAGE_TYPE` is set to `elasticsearch`:

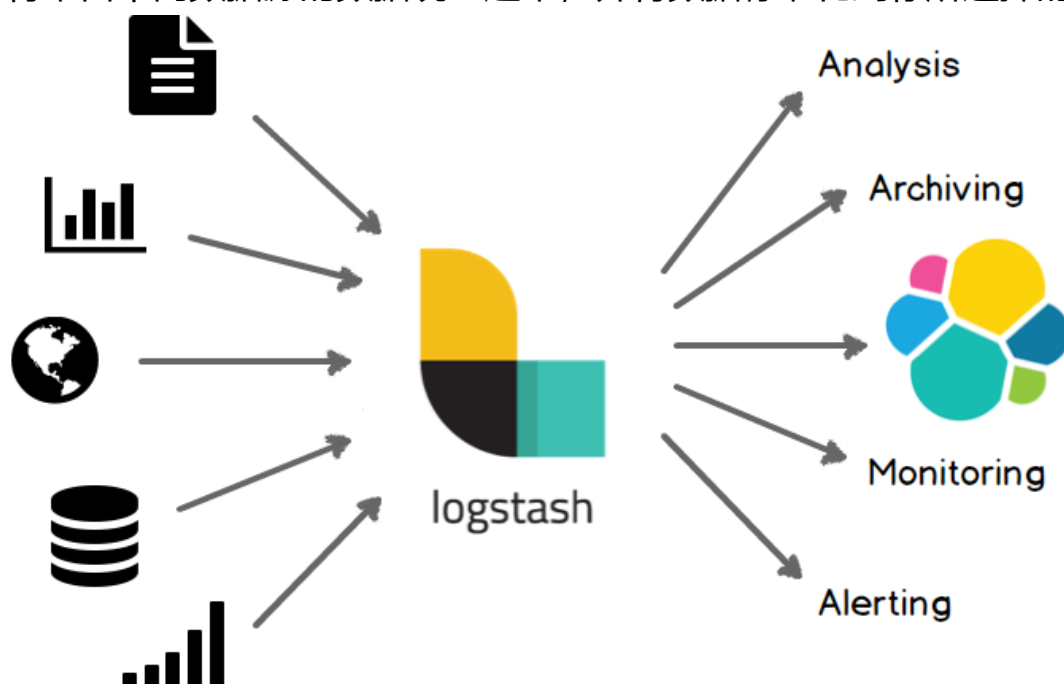
- * `ES_HOSTS`: A comma separated list of elasticsearch base urls to connect to ex. `http://host:9200`. Defaults to `"http://localhost:9200"`.
- * `ES_PIPELINE`: Indicates the ingest pipeline used before spans are indexed. No default.
- * `ES_TIMEOUT`: Controls the connect, read and write socket timeouts (in milliseconds) for Elasticsearch API. Defaults to 10000 (10 seconds)
- * `ES_INDEX`: The index prefix to use when generating daily index names. Defaults to `zipkin`.
- * `ES_DATE_SEPARATOR`: The date separator to use when generating daily index names. Defaults to `'-'`.
- * `ES_INDEX_SHARDS`: The number of shards to split the index into. Each shard and its replicas are assigned to a machine in the cluster. Increasing the number of shards and machines in the cluster will improve read and write performance. Number of shards cannot be changed for existing indices, but new daily indices will pick up changes to the setting. Defaults to 5.
- * `ES_INDEX_REPLICAS`: The number of replica copies of each shard in the index. Each shard and its replicas are assigned to a machine in the cluster. Increasing the number of replicas and machines in the cluster will improve read performance, but not write performance. Number of replicas can be changed for existing indices. Defaults to 1. It is highly discouraged to set this to 0 as it would mean a machine failure results in data loss.
- * `ES_USERNAME` and `ES_PASSWORD`: Elasticsearch basic authentication, which defaults to empty string. Use when X-Pack security (formerly Shield) is in place.
- * `ES_HTTP_LOGGING`: When set, controls the volume of HTTP logging of the Elasticsearch API. Options are BASIC, HEADERS, BODY

Example usage:

Logstash

Logstash介绍

Logstash是一个开源数据收集引擎，具有实时管道功能。Logstash可以动态地将来自不同数据源的数据统一起来，并将数据标准化到你选择的目的地。

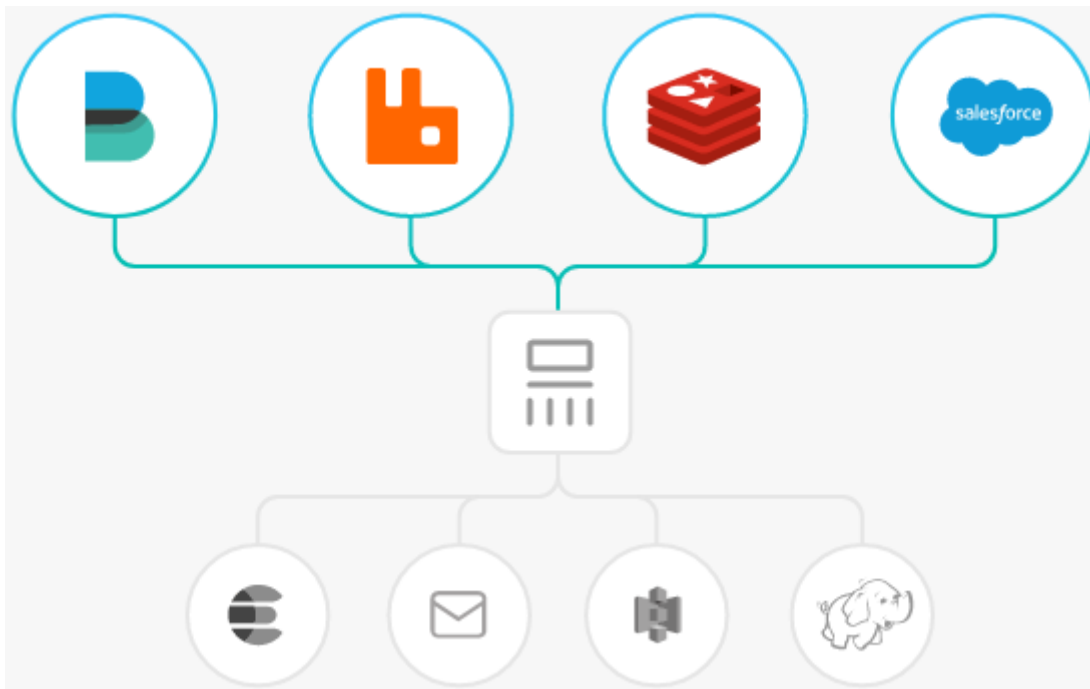


集中、转换和存储数据

Logstash是一个开源的服务器端数据处理管道，可以同时从多个数据源获取数据，并对其进行转换，然后将其发送到“存储”也就是Elasticsearch。

输入：采集各种样式、大小和来源的数据

数据往往以各种各样的形式，或分散或集中地存在于很多系统中。Logstash 支持各种输入选择，可以在同一时间从众多常用来源捕捉事件。能够以连续的流式传输方式，轻松地从您的日志、指标、Web 应用、数据存储以及各种 AWS 服务采集数据。

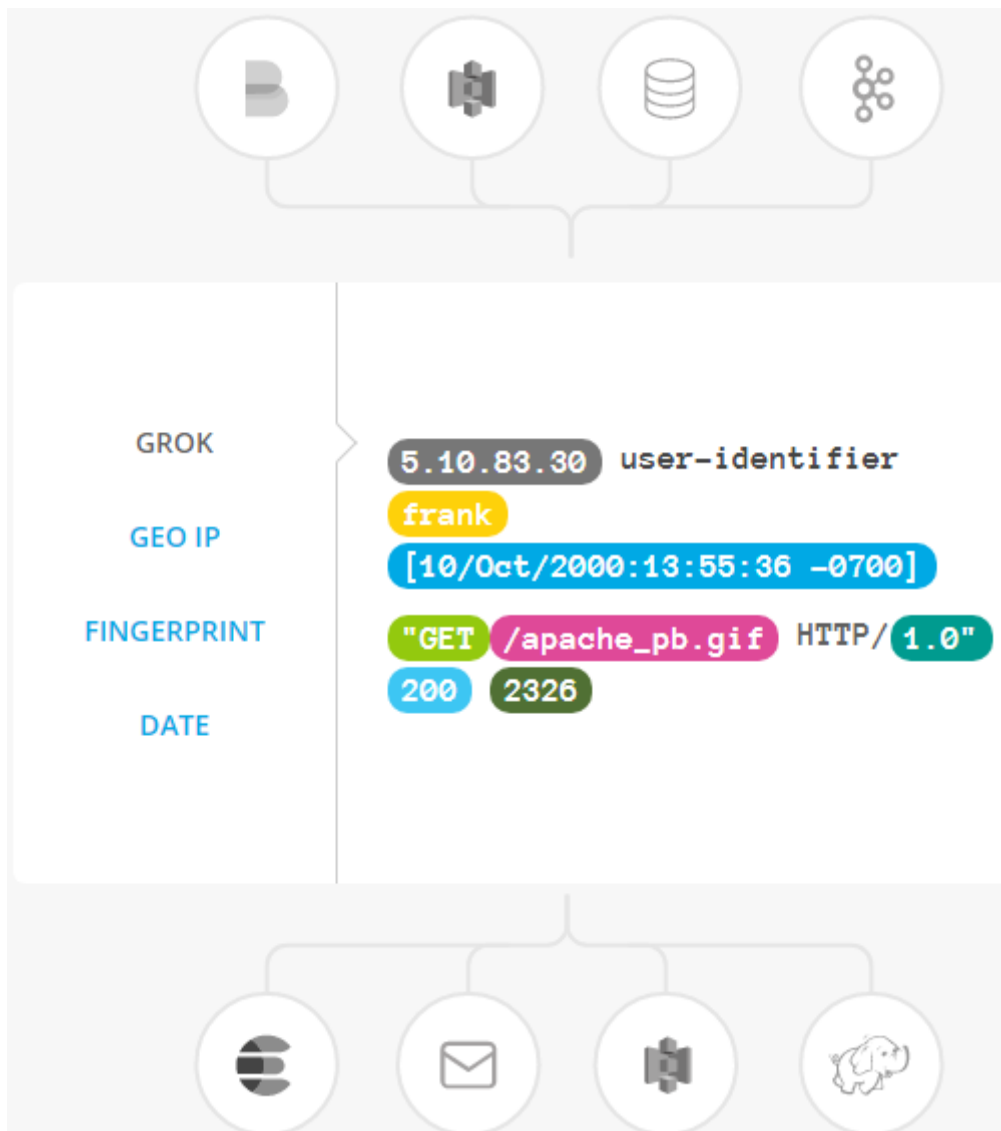


过滤器：实时解析和转换数据

数据从源传输到存储库的过程中，Logstash 过滤器能够解析各个事件，识别已命名的字段以构建结构，并将它们转换成通用格式，以便更轻松、更快速地分析和实现商业价值。

Logstash 能够动态地转换和解析数据，不受格式或复杂度的影响：

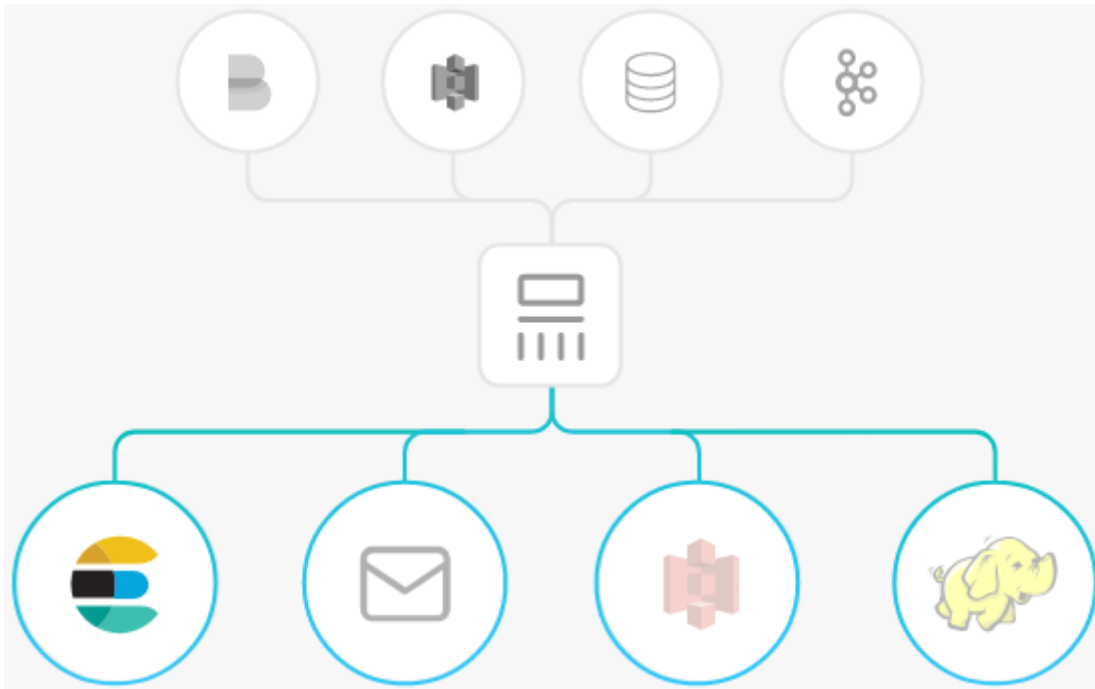
- 利用 Grok 从非结构化数据中派生出结构
- 从 IP 地址破译出地理坐标
- 将 PII 数据匿名化，完全排除敏感字段
- 整体处理不受数据源、格式或架构的影响



输出：选择存储，导出数据

尽管 Elasticsearch 是我们的首选输出方向，能够为我们的搜索和分析带来无限可能，但它并非唯一选择。

Logstash 提供众多输出选择，您可以将数据发送到您要指定的地方，并且能够灵活地解锁众多下游用例。



安装Logstash

官方<https://mirrors.huaweicloud.com/>访问缓慢，可以使用国内镜像下载华为云：<https://mirrors.huaweicloud.com/>

1、解压文件：

名称	修改日期	类型
bin	2020/3/13 0:42	文件夹
config	2020/3/13 0:42	文件夹
data	2020/2/29 1:56	文件夹
lib	2020/3/13 0:42	文件夹
logstash-core	2020/3/13 0:42	文件夹
logstash-core-plugin-api	2020/3/13 0:42	文件夹
modules	2020/3/13 0:42	文件夹
tools	2020/3/13 0:42	文件夹
vendor	2020/3/13 0:43	文件夹
x-pack	2020/3/13 0:43	文件夹
CONTRIBUTORS	2020/2/29 1:56	文件
Gemfile	2020/2/29 1:56	文件
Gemfile.lock	2020/2/29 1:56	LOCK 文件
LICENSE.txt	2020/2/29 1:56	文本文档
NOTICE.TXT	2020/2/29 1:56	文本文档

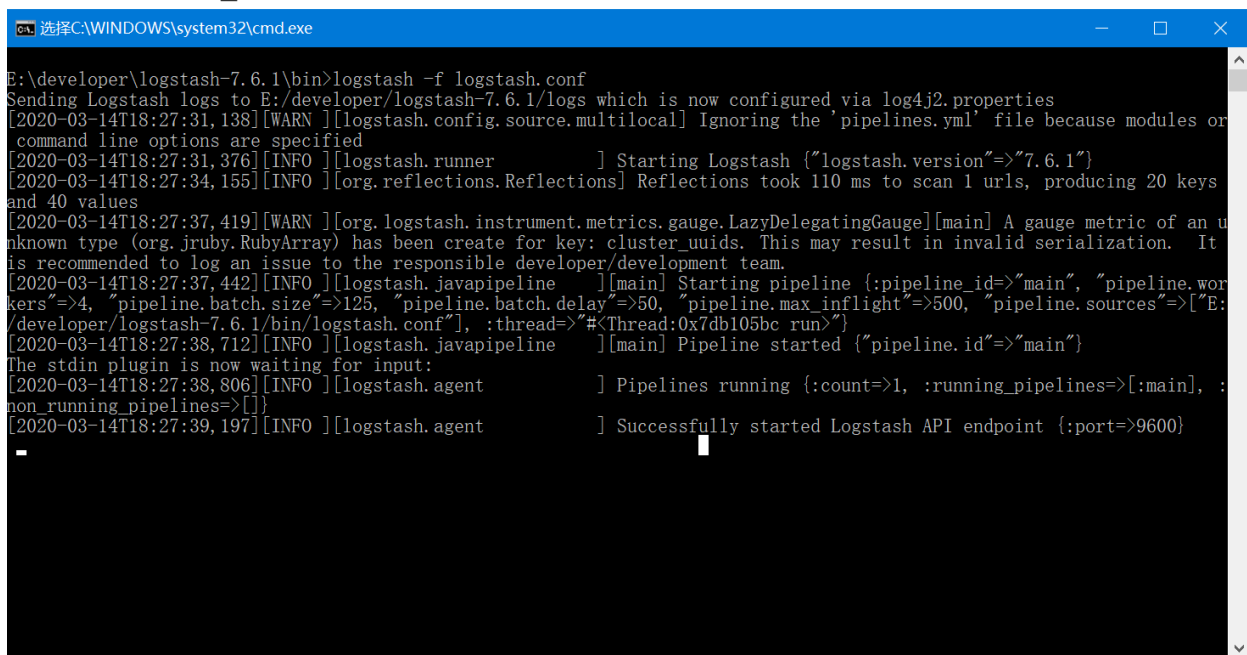
2、进入bin目录，新建文件 logstash.conf

```
1 input {
2   stdin{
3   }
4 }
5
6 output {
7   stdout{
8   }
9 }
```

3、在bin目录，新文件文件 run_default.bat

```
1 logstash -f logstash.conf
```

4、启动 run_default.bat

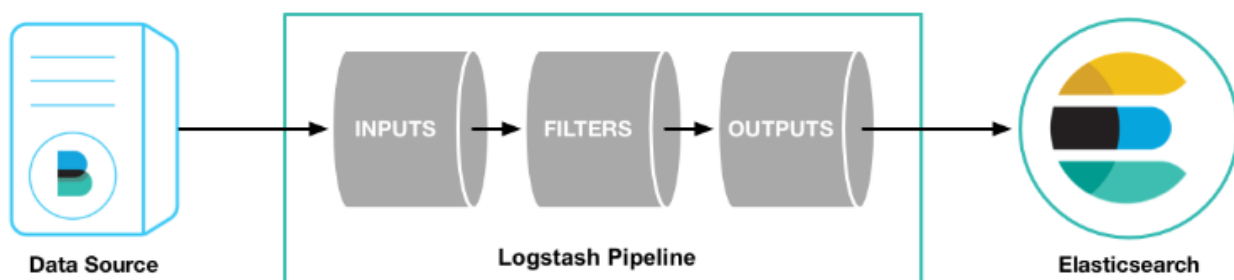


```
选择C:\WINDOWS\system32\cmd.exe
E:\developer\logstash-7.6.1\bin>logstash -f logstash.conf
Sending Logstash logs to E:/developer/logstash-7.6.1/logs which is now configured via log4j2.properties
[2020-03-14T18:27:31,138][WARN ][logstash.config.source.multilocal] Ignoring the 'pipelines.yml' file because modules or
command line options are specified
[2020-03-14T18:27:31,376][INFO ][logstash.runner] Starting Logstash {"logstash.version"=>"7.6.1"}
[2020-03-14T18:27:34,155][INFO ][org.reflections.Reflections] Reflections took 110 ms to scan 1 urls, producing 20 keys
and 40 values
[2020-03-14T18:27:37,419][WARN ][org.logstash.instrument.metrics.gauge.LazyDelegatingGauge][main] A gauge metric of an u
nknown type (org.jruby.RubyArray) has been create for key: cluster_uuids. This may result in invalid serialization. It
is recommended to log an issue to the responsible developer/development team.
[2020-03-14T18:27:37,442][INFO ][logstash.javapipeline][main] Starting pipeline {"pipeline_id"=>"main", "pipeline.wor
kers"=>4, "pipeline.batch.size"=>125, "pipeline.batch.delay"=>50, "pipeline.max_inflight"=>500, "pipeline.sources"=>["E:
/developer/logstash-7.6.1/bin/logstash.conf"], :thread=>#<Thread:0x7db105bc run>}
[2020-03-14T18:27:38,712][INFO ][logstash.javapipeline][main] Pipeline started {"pipeline.id"=>"main"}
The stdin plugin is now waiting for input:
[2020-03-14T18:27:38,806][INFO ][logstash.agent] Pipelines running {:count=>1, :running_pipelines=>[:main], :
non_running_pipelines=>[]}
[2020-03-14T18:27:39,197][INFO ][logstash.agent] Successfully started Logstash API endpoint {:port=>9600}
```

5、浏览器访问:<http://localhost:9600/>

```
localhost:9600
{"host":"LAPTOP-KUM04QTL","version":"7.6.1","http_address":"127.0.0.1:9600","id":"0da5da6d-c59f-4d2b-aecf-ida440de0c74","name":"LAPTOP-KUM04QTL","ephemeral_id":"483086aa-8del-4182-b5ff-2651b3d40a03","status":"green","snapshot":false,"pipeline":{"workers":4,"batch_size":125,"batch_delay":50},"build_date":"2020-02-29T01:53:17+00:00","build_sha":"0a73e90f49c005bffe8525d1b06728d17dbdf58","build_snapshot":false}
```

Logstash管道有两个必需的元素，输入和输出，以及一个可选元素过滤器。输入插件从数据源那里消费数据，过滤器插件根据你的期望修改数据，输出插件将数据写入目的地。



整合logstash

Spring Cloud Sleuth在与ELK平台整合使用时，实际上我们只要实现与负责日志收集的Logstash完成数据对接即可，所以我们需要为Logstash准备json格式的日志输出。由于Spring Boot应用默认使用了logback来记录日志，而Logstash自身也有对logback日志工具的支持工具，所以我们可以直接通过在logback的配置中增加对logstash的appender，就能非常方便的将日志转换成以json的格式存储和输出了。

下面介绍一下，如何实现面向Logstash的日志输出配置：

- 在pom.xml依赖中引入logstash-logback-encoder依赖，具体如下：

```
1 <dependency>
```

```
2 <groupId>net.logstash.logback</groupId>
3 <artifactId>logstash-logback-encoder</artifactId>
4 <version>4.6</version>
5 </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。由于logback-spring.xml的加载在application.properties之前，所以之前的配置logback-spring.xml无法获取到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，具体内容如下：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3   <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
4
5   <springProperty scope="context" name="springAppName" source="spring.application.name"/>
6   <!-- 日志在工程中的输出位置 -->
7   <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>
8   <!-- 控制台的日志输出样式 -->
9   <property name="CONSOLE_LOG_PATTERN"
10     value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]){yellow} %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>
11
12   <!-- 控制台Appender -->
13   <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
14     <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
15       <level>INFO</level>
16     </filter>
17     <encoder>
18       <pattern>${CONSOLE_LOG_PATTERN}</pattern>
19       <charset>utf8</charset>
20     </encoder>
```

```
21 </appender>
22
23 <!-- 为logstash输出的json格式的Appender -->
24 <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
25 <file>${LOG_FILE}.json</file>
26 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
27 <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
28 <maxHistory>7</maxHistory>
29 </rollingPolicy>
30 <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
31 <providers>
32 <timestamp>
33 <timeZone>UTC</timeZone>
34 </timestamp>
35 <pattern>
36 <pattern>
37 {
38 "severity": "%level",
39 "service": "${springAppName:-}",
40 "trace": "%X{X-B3-TraceId:-}",
41 "span": "%X{X-B3-SpanId:-}",
42 "exportable": "%X{X-Span-Export:-}",
43 "pid": "${PID:-}",
44 "thread": "%thread",
45 "class": "%logger{40}",
46 "rest": "%message"
47 }
48 </pattern>
49 </pattern>
50 </providers>
51 </encoder>
52 </appender>
53
54 <root level="INFO">
55 <appender-ref ref="console"/>
56 <appender-ref ref="logstash"/>
57 </root>
58 </configuration>
```


对logstash支持主要通过名为logstash的appender实现，内容并不复杂，主要是对日志信息的格式化处理，上面为了方便调试查看我们先将json日志输出到文件中。

完成上面的改造之后，我们再将快速入门的示例运行起来，并发起对trace-1的接口访问。此时我们可以在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logstash的appender输出的日志文件，其中记录了类似下面格式的json日志：

```
1 {"@timestamp":"2020-03-14T09:57:58.970+00:00","severity":"INFO","service":"trace-1","trace":"589ee5f7b860132f","span":"a9e891273affb7fc","exportable":"false","pid":"19756","ad":"http-nio-9101-exec-1","class":"c.d.TraceApplication$$EnhancerBySpringCGLIB$$a9604da6","rest":"===<call trace-1>==="}
2 {"@timestamp":"2020-03-14T09:57:59.061+00:00","severity":"INFO","service":"trace-1","trace":"589ee5f7b860132f","span":"2df8511ddf3d79a2","exportable":"false","pid":"19756","ad":"http-nio-9101-exec-1","class":"o.s.c.a.AnnotationConfigApplicationContext","rest":"Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@64951f38: startup date [Sun Dec 04 14:57:59 CST 2016]; parent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4b8c8f15"}
```

我们除了可以通过上面的方式生成json文件之外，也可以使用LogstashTcpSocketAppender将日志内容直接通过Tcp Socket输出到logstash服务端，比如：

```
1 <appender name="logstash" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
2   <destination>127.0.0.1:9600</destination>
3   ...
4 </appender>
```

分布式服务跟踪（跟踪原理）

分布式系统中的服务跟踪在理论上并不复杂，它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，同时在分布式系统内部流转的时候，框架始终保持传递该唯一标识，直到返回给请求方为止，这个唯一标识就是前文中提到的Trace ID。通过Trace ID的记录，我们就能将所有请求过程日志关联起来。
- 为了统计各处理单元的时间延迟，当请求达到各个服务组件时，或是处理逻辑到达某个状态时，也通过一个唯一标识来标记它的开始、具体过程以及结束，该标识就是我们前文中提到的Span ID，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。

在快速入门示例中，我们轻松实现了日志级别的跟踪信息接入，这完全归功于spring-cloud-starter-sleuth组件的实现。在Spring Boot应用中，通过在工程中引入spring-cloud-starter-sleuth依赖之后，它会自动的为当前应用构建起各通信通道的跟踪机制，比如：

- 通过诸如RabbitMQ、Kafka（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
- 通过Zuul代理传递的请求
- 通过RestTemplate发起的请求

在快速入门示例中，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会为在该请求的Header中增加实现跟踪需要的重要信息，主要有下面这几个（更多关于头信息的定义我们可以通过查看org.springframework.cloud.sleuth.Span的源码获取）：

- X-B3-TraceId：一条请求链路（Trace）的唯一标识，必须值
- X-B3-SpanId：一个工作单元（Span）的唯一标识，必须值

- X-B3-ParentSpanId: 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

我们可以通过对`trace-2`的实现做一些修改来输出这些头部信息, 具体如下:

```
1 @RequestMapping(value = "/trace-2", method = RequestMethod.GET)
2 public String trace(HttpServletRequest request) {
3     logger.info("===<call trace-2, TraceId={}, SpanId={}>===",
4         request.getHeader("X-B3-TraceId"), request.getHeader("X-B3-SpanId"));
5     return "Trace";
6 }
```

通过上面的改造, 我们再运行快速入门的示例内容, 并发起对`trace-1`的接口访问, 我们可以得到如下输出内容。其中在`trace-2`的控制台中, 输出了当前正在处理的`TraceId`和`SpanId`信息。

```
1 -- trace-1
2 INFO [trace-1,a6e9175ffd5d2c88,8524f519b8a9e399,true] 10532 --- [nio-9101-exec-2] icationEnhancerBySpringCGLIB27aa9624 : ===<call trace-1>===
3
4 -- trace-2
5 INFO [trace-2,a6e9175ffd5d2c88,ce60dcf1e2ed918f,true] 1208 --- [nio-9102-exec-3] icationEnhancerBySpringCGLIBa7d84797 : ===<call trace-2, TraceId=a6e9175ffd5d2c88, SpanId=be4949ec115e554e>===
```

为了更直观的观察跟踪信息, 我们还可以在`application.properties`中增加下面的配置:

```
1 logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG
```

通过将Spring MVC的请求分发日志级别调整为`DEBUG`级别, 我们可以看到更多跟踪信息:

```
1 -- trace-1
2 2020-03-14 09:26:52.663 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true] 10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet : DispatcherServlet with name 'dispatcherServlet' processing GET request for [/trace-1]
3 2020-03-14 09:26:52.666 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true] 10532 --- [nio-9101-exec-2] o.s.w
```

```

eb.servlet.DispatcherServlet : Last-Modified value for [/trace-1] is: -1
4 2020-03-14 09:26:52.685 DEBUG [trace-1,a6e9175ffd5d2c88,8524f519b8a9e399,true] 10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet : Null ModelAndView returned to DispatcherServlet with name 'dispatcherServlet': assuming HandlerAdapter completed request handling
5 2020-03-14 09:26:52.685 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true] 10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet : Successfully completed request
6
7 -- trace-2
8 2020-03-14 09:26:52.673 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true] 1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet : DispatcherServlet with name 'dispatcherServlet' processing GET request for [/trace-2]
9 2020-03-14 09:26:52.679 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true] 1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet : Last-Modified value for [/trace-2] is: -1
10 2020-03-14 09:26:52.682 DEBUG [trace-2,a6e9175ffd5d2c88,ce60dcf1e2ed918f,true] 1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet : Null ModelAndView returned to DispatcherServlet with name 'dispatcherServlet': assuming HandlerAdapter completed request handling
11 2020-03-14 09:26:52.683 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true] 1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet : Successfully completed request

```

分布式服务跟踪（收集原理）

下面介绍一下关于Zipkin收集跟踪信息的过程细节，以帮助我们更好地理解Sleuth生产跟踪信息以及输出跟踪信息的整体过程和工作原理。

数据模型

我们先来看看Zipkin中关于跟踪信息的一些基础概念。由于Zipkin的实现借鉴了Google的Dapper，所以它们有着类似的核心术语，主要有下面几个内容：

- **Span**：它代表了一个基础的工作单元。我们以HTTP请求为例，一次完整的请求过程在客户端和服务端都会产生多个不同的事件状态（比如下面所说的四个核心Annotation所标识的不同阶段），对于同一个请求来说，它们属于一个工作单元，所以同一HTTP请求过程中的四个Annotation同属于一个Span。每一个不同的工作单元都通过一个64位的ID来唯一标识，

称为Span ID。另外，在工作单元中还存储了一个用来串联其他工作单元的ID，它也通过一个64位的ID来唯一标识，称为Trace ID。在同一条请求链路中的不同工作单元都会有不同的Span ID，但是它们的Trace ID是相同的，所以通过Trace ID可以将一次请求中依赖的所有依赖请求串联起来形成请求链路。除了这两个核心的ID之外，Span中还存储了一些其他信息，比如：描述信息、事件时间戳、Annotation的键值对属性、上一级工作单元的Span ID等。

- **Trace**：它是由一系列具有相同Trace ID的Span串联形成的一个树状结构。在复杂的分布式系统中，每一个外部请求通常都会产生一个复杂的树状结构的Trace。
- **Annotation**：它用来及时地记录一个事件的存在。我们可以把Annotation理解为一个包含有时间戳的事件标签，对于一个HTTP请求来说，在Sleuth中定义了下面四个核心Annotation来标识一个请求的开始和结束：
 - **cs** (Client Send)：该Annotation用来记录客户端发起了一个请求，同时它也标识了这个HTTP请求的开始。
 - **sr** (Server Received)：该Annotation用来记录服务端接收到了请求，并准备开始处理它。通过计算**sr**与**cs**两个Annotation的时间戳之差，我们可以得到当前HTTP请求的网络延迟。
 - **ss** (Server Send)：该Annotation用来记录服务端处理完请求后准备发送请求响应信息。通过计算**ss**与**sr**两个Annotation的时间戳之差，我们可以得到当前服务端处理请求的时间消耗。
 - **cr** (Client Received)：该Annotation用来记录客户端接收到服务端的回复，同时它也标识了这个HTTP请求的结束。通过计算**cr**与**cs**两个Annotation的时间戳之差，我

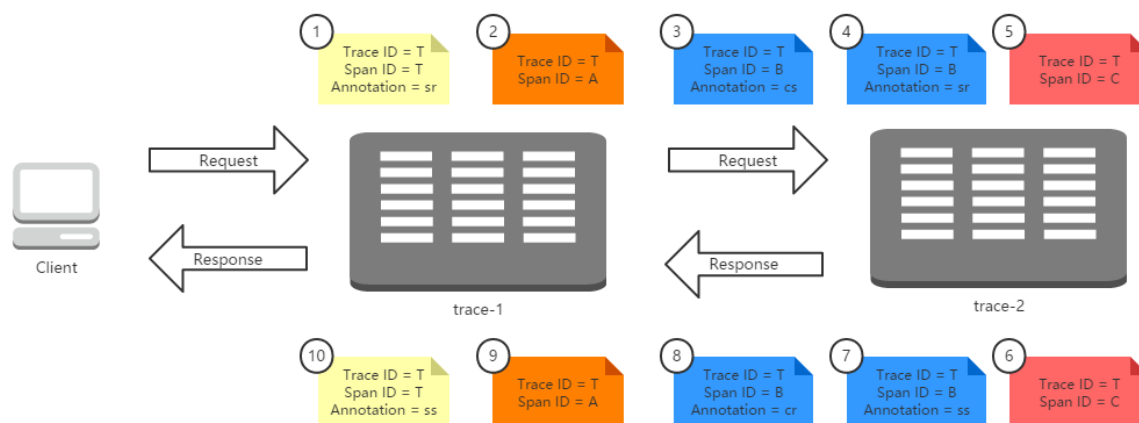
们可以得到该HTTP请求从客户端发起开始到接收服务端响应的总时间消耗。

- **BinaryAnnotation**：它用来对跟踪信息添加一些额外的补充说明，一般以键值对方式出现。比如：在记录HTTP请求接收后执行具体业务逻辑时，此时并没有默认的**Annotation**来标识该事件状态，但是有**BinaryAnnotation**信息对其进行补充。

收集机制

在理解了Zipkin的各个基本概念之后，下面我们结合前面例子来详细介绍和理解Spring Cloud Sleuth是如何对请求调用链路完成跟踪信息的生产、输出和后续处理的。

首先，我们来看看Sleuth在请求调用时是怎么样来记录和生成跟踪信息的。下图展示了我们在本章节中实现示例的运行全过程：客户端发送了一个HTTP请求到trace-1，trace-1依赖于trace-2的服务，所以trace-1再发送一个HTTP请求到trace-2，待trace-2返回响应之后，trace-1再组织响应结果返回给客户端。

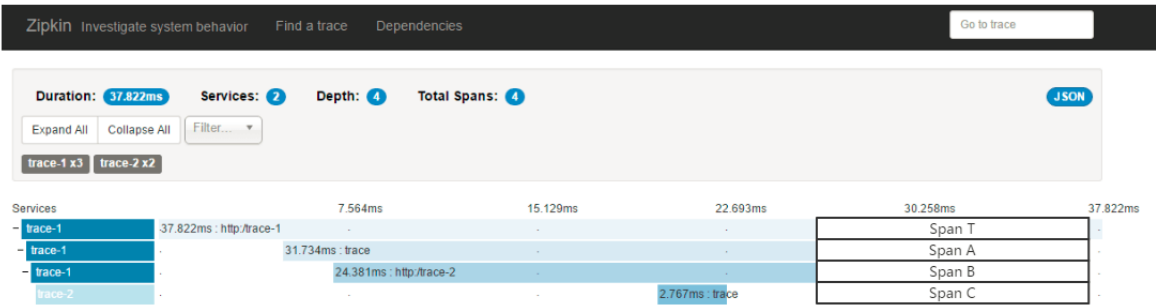


在上图请求过程中，我们为整个调用过程标记了10个标签，它们分别代表了该请求链路运行过程中记录的几个重要事件状态，我们根据事件发生的时间顺序我们为这些标签做了从小到大的编号，1代表请求的开始、10代表请求的结束。每个标签中记录了一些我们上面提到过的核心元素：Trace ID、Span ID以及Annotation。由于这些标签都源自一个请求，所以他们的Trace ID相同，而标签1和标签10是起始和结束节点，它们的Trace ID与Span ID是相同的。

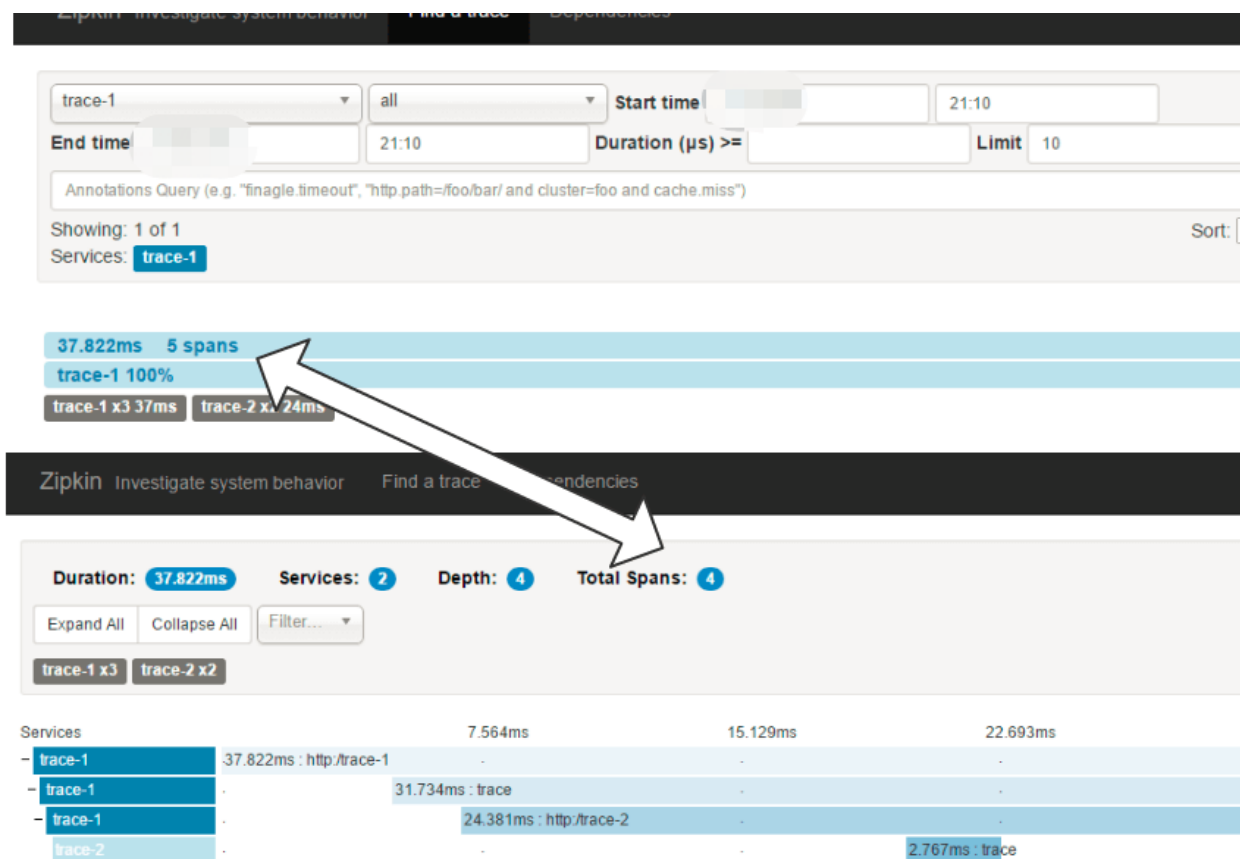
根据Span ID，我们可以发现在这些标签中一共产生了4个不同ID的Span，这4个Span分别代表了这样4个工作单元：

- Span T：记录了客户端请求到达trace-1和trace-1发送请求响应的两个事件，它可以计算出客户端请求响应过程的总延迟时间。
- Span A：记录了trace-1应用在接收到客户端请求之后调用处理方法的开始和结束两个事件，它可以计算出trace-1应用用于处理客户端请求时，内部逻辑花费的时间延迟。
- Span B：记录了trace-1应用发送请求给trace-2应用、trace-2应用接收请求，trace-2应用发送响应、trace-1应用接收响应四个事件，它可以计算出trace-1调用trace-2的总体依赖时间（cr - cs），也可以计算出trace-1到trace-2的网络延迟（sr - cs），也可以计算出trace-2应用用于处理客户端请求的内部逻辑花费的时间延迟（ss - sr）。
- Span C：记录了trace-2应用在接收到trace-1的请求之后调用处理方法的开始和结束两个事件，它可以计算出trace-2应用用于处理来自trace-1的请求时，内部逻辑花费的时间延迟。

在图中展现的这个4个Span正好对应了Zipkin查看跟踪详细页面中的显示内容，它们的对应关系如下图所示：



我们在Zipkin服务端查询跟踪信息时（如下图所示），在查询结果页面中显示的spans是5，而点击进入跟踪明细页面时，显示的Total Spans又是4，为什么会出现span数量不一致的情况呢？



实际上这两边的span数量内容有不同的含义，在查询结果页面中的5 spans代表了总共接收的Span数量，而在详细页面中的Total Span则是对接收Span进行合并后的结果，也就是前文中我们介绍的4个不同ID的Span内容。下面我们来详细研究一下Zipkin服务端收集客户端跟踪信息的过程，看看它到底收到了哪些具体的Span内容，从而来理解Zipkin中收集到的Span总数量。

为了更直观的观察Zipkin服务端的收集过程，我们可以通过实现消息中间件方式收集跟踪信息的程序进行调试。通过在Zipkin服务端的消息通道监听程序中增加断点，我们就能清楚的知道客户端都发送了一些什么信息到Zipkin的服务端。在spring-cloud-sleuth-zipkin-stream依赖包中的代码并不多，我们很容易的就能找到定义消息通道监听的实现类：

org.springframework.cloud.sleuth.zipkin.stream.ZipkinMessageListener。它的具体实现如下所示，其中SleuthSink.INPUT定义了监听的输入通道，默认会使用名为sleuth的主题，我们也可以通过Spring Cloud Stream的配置对其进行修改。

```
1 @MessageEndpoint
2 @Conditional(NotSleuthStreamClient.class)
3 public class ZipkinMessageListener {
```



```

4
5  final Collector collector;
6
7  @ServiceActivator(inputChannel = SleuthSink.INPUT)
8  public void sink(Spans input) {
9      List<zipkin.Span> converted = ConvertToZipkinSpanList.convert(input);
10     this.collector.accept(converted, Callback.NOOP);
11 }
12
13 ...
14
15 }

```

从通道监听方法的定义中我们可以看到Sleuth与Zipkin在整合的时候是有两个不同的Span定义的，一个是消息通道的输入对象

`org.springframework.cloud.sleuth.stream.Spans`，它是sleuth中定义的用于消息通道传输的Span对象，每个消息中包含的Span信息定义在

`org.springframework.cloud.sleuth.Span`对象中，但是真正在zipkin服务端使用的并非这个Span对象，而是zipkin自己的`zipkin.Span`对象。所以，在消息通道监听处理方法中，对sleuth的Span做了处理，每次接收到sleuth的Span之后就将其转换成Zipkin的Span。

下面我们可以尝试在`sink(Spans input)`方法实现的第一行代码中加入断点，并向`trace-1`发送一个请求，触发跟踪信息发送到RabbitMQ上。此时我们通过DEBUG模式可以发现消息通道中都接收到了两次输入，一次来自`trace-1`，一次来自`trace-2`。下面两张图分别展示了来自`trace-1`和`trace-2`输出的跟踪消息，其中`trace-1`的跟踪消息包含了3条span信息，`trace-2`的跟踪消息包含了2条span信息，所以在这个请求调用链上，一共发送了5个span信息，也就是我们在Zipkin搜索结果页面中看到的spans数量信息。

```

▼ P input = {Spans@8609}
  ▼ I host = {Host@8610}
    ► I serviceName = "trace-1"
    ► I address = "192.168.20.65"
    ► I port = {Integer@8621} "9101"
  ▼ I spans = {ArrayList@8611} size = 3
    ► 0 = {Span@8616} "[Trace: e9a933ec50d180d6, Span: 1ae2e9a317faa422, Parent: c48122fa096bffe8, exportable:true]"
    ► 1 = {Span@8617} "[Trace: e9a933ec50d180d6, Span: c48122fa096bffe8, Parent: e9a933ec50d180d6, exportable:true]"
    ► 2 = {Span@8618} "[Trace: e9a933ec50d180d6, Span: e9a933ec50d180d6, Parent: null, exportable:true]"

```

```

▼ P input = {Spans@8587}
  ▼ f host = {Host@8589}
    ▶ f serviceName = "trace-2"
    ▶ f address = "192.168.20.65"
    ▶ f port = {Integer@8593} "9102"
  ▼ f spans = {ArrayList@8590} size = 2
    ▶ 0 = {Span@8595} "[Trace: e9a933ec50d180d6, Span: 36194e4182985c4e, Parent: 1ae2e9a317faa422, exportable:true]"
    ▶ 1 = {Span@8596} "[Trace: e9a933ec50d180d6, Span: 1ae2e9a317faa422, Parent: c48122fa096bffe8, exportable:true]"

```

点开一个具体的Span内容，我们可以看到如下所示的结构，它记录了Sleuth中定义的Span详细信息，包括该Span的开始时间、结束时间、Span的名称、Trace ID、Span ID、Tags（对应Zipkin中的BinaryAnnotation）、Logs（对应Zipkin中的Annotation）等我们之前提到过的核心跟踪信息。

```

▼ f spans = {ArrayList@8611} size = 3
  ▼ 0 = {Span@8616} "[Trace: e9a933ec50d180d6, Span: 1ae2e9a317faa422, Parent: c48122fa096bffe8, exportable:true]"
    f begin = 1480467275817
    f end = 1480467277375
    ▶ f name = "http/trace-2"
    f traceId = -1609698301747035946
    ▼ f parents = {ArrayList@8627} size = 1
      ▶ 0 = {Long@8670} "-4287106912984760344"
    f spanId = 1937367676413977634
    f remote = false
    f exportable = true
    ▼ f tags = {LinkedHashMap@8628} size = 4
      ▶ 0 = {LinkedHashMap$Entry@8636} "http.path" -> "/trace-2"
      ▶ 1 = {LinkedHashMap$Entry@8637} "http.url" -> "http://trace-2/trace-2"
      ▶ 2 = {LinkedHashMap$Entry@8638} "http.method" -> "GET"
      ▶ 3 = {LinkedHashMap$Entry@8639} "http.host" -> "trace-2"
    f processId = null
    ▼ f logs = {ArrayList@8629} size = 2
      ▶ 0 = {Log@8649} "Log{timestamp=1480467275818, event='cs'}"
      ▶ 1 = {Log@8650} "Log{timestamp=1480467277375, event='cr'}"
    f savedSpan = null
    ▶ f startNanos = {Long@8630} "46627595152818"
    ▶ f durationMicros = {Long@8631} "1558124"

```

介绍到这里仔细同学可能会有一个这样的疑惑，在明细信息中展示的Trace ID和Span ID的值为什么与列表展示的概要信息中的Trace ID和Span ID的值不一样呢？实际上，Trace ID和Span ID都是使用long类型存储的，在DEBUG模式下查看其明细时自然是long类型，也就是它的原始值，但是在查看Span对象的时候，我们看到的是通过toString()函数处理过的值，从sleuth的Span源码中我们可以看到如下定义，在输出Trace ID和Span ID时都调用了idToHex函数将long类型的值转换成了16进制的字符串值，所以在DEBUG时我们会看到两个不一样的值。

```

1 public String toString() {
2     return "[Trace: " + idToHex(this.traceId) + ", Span: " + idToHex(this.spanId)

```

```

3 + ", Parent: " + getParentIdIfPresent() + ", exportable:" + this.exportable + "]];
4 }
5
6 public static String idToHex(long id) {
7     return Long.toHexString(id);
8 }

```

在接收到Sleuth之后我们将程序继续执行下去，可以看到经过转换后的Zipkin的Span内容，它们保存在一个名为converted的列表中，具体内容如下所示：

```

▼ converted = (ArrayList@8672) size = 3
  ▼ 0 = (Span@8687) {"traceld":"e9a933ec50d180d6","name":"http/trace-2","id":"1ae2e9a317faa422","parentId":"c48122fa096bffe8","timestamp":1480467275817000}
    ▶ traceld = -1609698301747035946
    ▶ name = "http/trace-2"
    ▶ id = 1937367676413977634
    ▶ parentId = (Long@8670) "-4287106912984760344"
    ▶ timestamp = (Long@8693) "1480467275817000"
    ▶ duration = (Long@8694) "1557000"
    ▶ annotations = (Collections$UnmodifiableRandomAccessList@8695) size = 2
      ▶ 0 = (Annotation@8712) {"endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101},"timestamp":1480467275818000,"value":"cs"}
      ▶ 1 = (Annotation@8713) {"endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101},"timestamp":1480467277375000,"value":"cr"}
    ▶ binaryAnnotations = (Collections$UnmodifiableRandomAccessList@8696) size = 5
      ▶ 0 = (BinaryAnnotation@8701) {"key":"http.host","value":"trace-2","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}
      ▶ 1 = (BinaryAnnotation@8702) {"key":"http.method","value":"GET","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}
      ▶ 2 = (BinaryAnnotation@8703) {"key":"http.path","value":"/trace-2","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}
      ▶ 3 = (BinaryAnnotation@8704) {"key":"http.url","value":"http://trace-2/trace-2","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}
      ▶ 4 = (BinaryAnnotation@8705) {"key":"sa","value":true,"endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}
    ▶ debug = null

```

上图展示了转换后的Zipkin Span对象的详细内容，我们可以看到很多熟悉的名称，也就是之前我们介绍的关于zipkin中的各个基本概念，而这些基本概念的值我们也都可以在之前sleuth的原始span中找到，其中annotations和binaryAnnotations有一些特殊，在sleuth定义的span中没有使用相同的名称，而是使用了logs和tags来命名。从这里的详细信息中，我们可以直观的看到annotations和binaryAnnotations的作用，其中annotations中存储了当前Span包含的各种事件状态以及对应事件状态的时间戳，而binaryAnnotations则存储了对事件的补充信息，比如上图中存储的就是该HTTP请求的细节描述信息，除此之外，它也可以存储对调用函数的详细描述（如下图所示）。

```

▶ binaryAnnotations = (Collections$UnmodifiableRandomAccessList@8239) size = 3
  ▶ 0 = (BinaryAnnotation@8272) {"key":"lc","value":"unknown","endpoint":{"serviceName":"trace-2","ipv4":"192.168.31.189","port":9102}}
  ▶ 1 = (BinaryAnnotation@8273) {"key":"mvc.controller.class","value":"TraceApplication","endpoint":{"serviceName":"trace-2","ipv4":"192.168.31.189","port":9102}}
  ▶ 2 = (BinaryAnnotation@8274) {"key":"mvc.controller.method","value":"trace","endpoint":{"serviceName":"trace-2","ipv4":"192.168.31.189","port":9102}}

```

下面我们再来详细看看通过调试消息监听程序接收到的这5个Span内容。首先，我们可以发现每个Span中都包含有3个ID信息，其中除了标识Span自身的ID以及用来标识整条链路的traceld之外，还有一个之前没有提过的parentId，它是用来标识各Span父子关系的ID（它的值来自与上一步执行单元Span的ID），

通过parentId的定义我们可以为每个Span找到它的前置节点，从而定位每个Span在请求调用链中的确切位置。在每条调用链路中都有一个特殊的Span，它的parentId为null，这类Span我们称它为Root Span，也就是这条请求调用链的根节点。为了弄清楚这些Span之间的关系，我们可以从Root Span开始来整理出整条链路的Span内容。下表展示了我们从Root Span开始，根据各个Span的父子关系最后整理出的结果：

Host	Span ID	Parent Span ID	Annotation
trace-1	e9a933ec50d180d6	null	[sr, ss]
trace-1	c48122fa096bffe8	e9a933ec50d180d6	
trace-1	1ae2e9a317faa422	c48122fa096bffe8	[cs, cr]
trace-2	1ae2e9a317faa422	c48122fa096bffe8	[sr, ss]
trace-2	36194e4182985c4e	1ae2e9a317faa422	

上表中的Host代表了该Span是从哪个应用发送过来的；Span ID是当前Span的唯一标识；Parent Span ID代表了上一执行单元的Span ID；Annotation代表了该Span中记录的事件（这里主要用来记录HTTP请求的四个阶段，表中内容作了省略处理，只记录了Annotation名称（sr代表服务端接收请求，ss代表服务端发送请求，cs代表客户端发送请求，cr代表客户端接收请求），省略了一些其他细节信息，比如服务名、时间戳、IP地址、端口号等信息）；Binary Annotation代表了事件的补充信息（Sleuth的原始Span记录更为详细，Zipkin的Span处理后会去掉一些内容，对于有Annotation标识的信息，不再使用Binary Annotation补充，在上表中我们只记录了服务名、类名、方法名，同样省略了一些其他信息，比如：时间戳、IP地址、端口号等信息）。

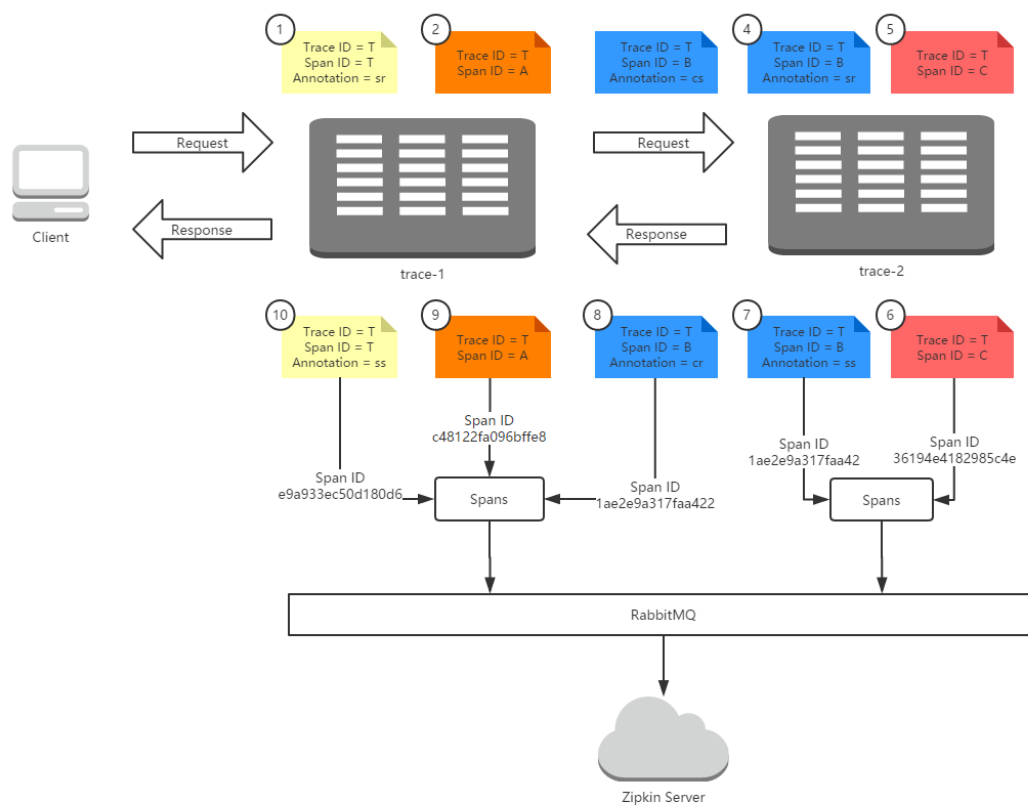
通过收集到的Zipkin Span详细信息，我们很容易的可以将它们与开始时介绍的一次调用链路中的10个标签内容联系起来：

- `Span ID = T`的标签有2个，分别是序号1和10，它们分别表示这次请求的开始和结束。它们对应了上表中ID为e9a933ec50d180d6的Span，该

Span的内容在标签10执行结束后，由`trace-1`将标签1和10合并成一个Span发送给Zipkin Server。

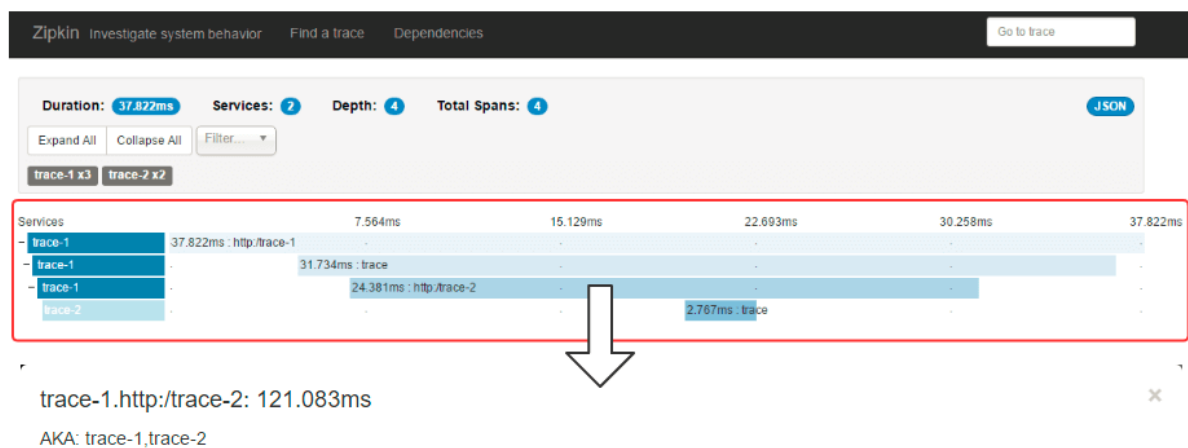
- `Span ID = A`的标签有2个，分别是序号2和9，它们分别表示了`trace-1`请求接收后，具体处理方法调用的开始和结束。该Span的内容在标签9执行结束后，由`trace-1`将标签2和9合并成一个Span发送给Zipkin Server。
- `Span ID = B`的标签有4个，分别是序号3、4、7、8，该Span比较特殊，它的产生跨越了两个实例，其中标签3和8是由`trace-1`生成的，而标签4和7则是由`trace-2`生成的，所以该标签会拆分成两个Span内容发送给Zipkin Server。`trace-1`会在标签8结束的时候将标签3和8合并成一个Span发送给Zipkin Server，而`trace-2`会在标签7结束的时候将标签4和7合并成一个Span发送给Zipkin Server。
- `Span ID = C`的标签有2个，分别是序号5和6，它们分别表示了`trace-2`请求接收后，具体处理方法调用的开始和结束。该Span的内容在标签6执行结束后，由`trace-2`将标签5和6合并成一个Span发送给Zipkin Server。

所以，根据上面的分析，Zipkin总共会收到5个Span：一个Span T，一个Span A，两个Span B，一个Span C。结合之前请求链路的标签图和这里的Span记录，我们可以总结出如下图所示的Span收集过程，读者可以参照此图来理解Span收集过程的处理逻辑以及各个Span之间的关系。



虽然，Zipkin服务端接收到了5个Span，但就如前文中分析的那样，其中有两个Span ID=B的标签，由于它们来自于同一个HTTP请求（trace-1对trace-2的服务调用），概念上它们属于同一个工作单元，因此Zipkin服务端在前端展现分析详情时会把这两个Span合并了来显示，而合并后的Span数量就是在请求链路详情页面中Total Spans的数量。

下图是示例的一个请求链路详情页面，在页面中显示了各个Span的延迟统计，其中第三条Span信息就是trace-1对trace-2的HTTP请求调用，通过点击它可以查看该Span的详细信息，点击后会以模态框的方式弹出Span详细信息（如图下半部分），在弹出框中详细展示了Span的Annotation和BinaryAnnotation信息，在Annotation区域我们可以看到它同时包含了trace-1和trace-2发送的Span信息，而在BinaryAnnotation区域则展示了该HTTP请求的详细信息。

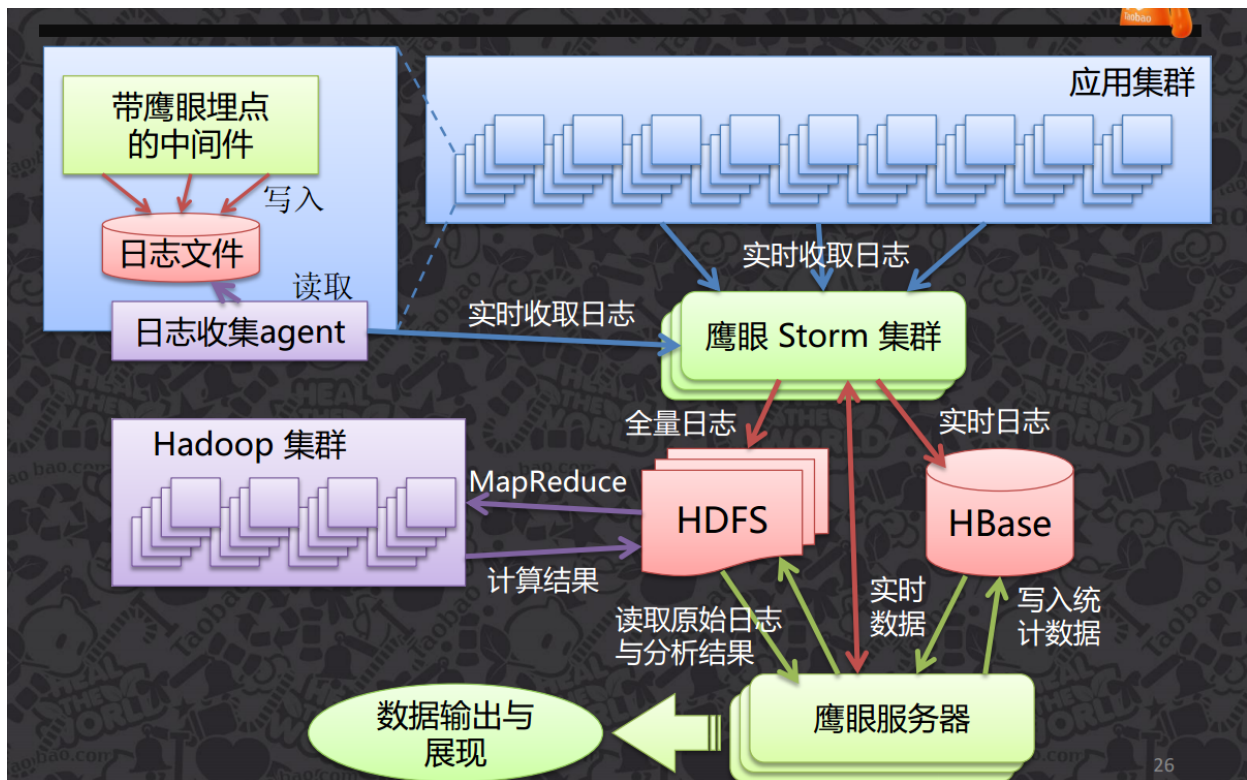


Date Time	Relative Time	Annotation	Address	Annotation信息
	38.000ms	Client Send	192.168.31.189:9101 (trace-1)	
	88.000ms	Server Receive	192.168.31.189:9102 (trace-2)	
	143.000ms	Client Receive	192.168.31.189:9101 (trace-1)	
	159.000ms	Server Send	192.168.31.189:9102 (trace-2)	

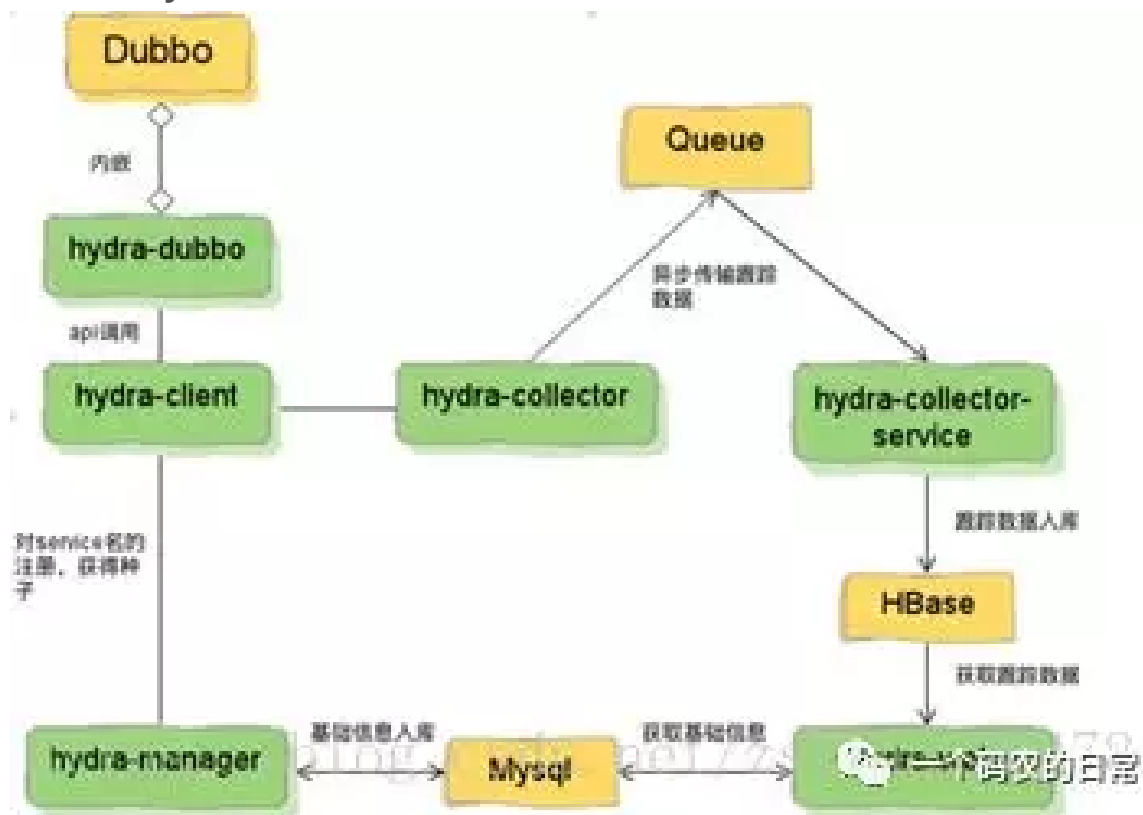
Key	Value	BinaryAnnotation信息
http.host	trace-2	
http.method	GET	
http.path	/trace-2	
http.url	http://trace-2/trace-2	
Server Address	192.168.31.189:9101 (trace-1)	

拓展视野

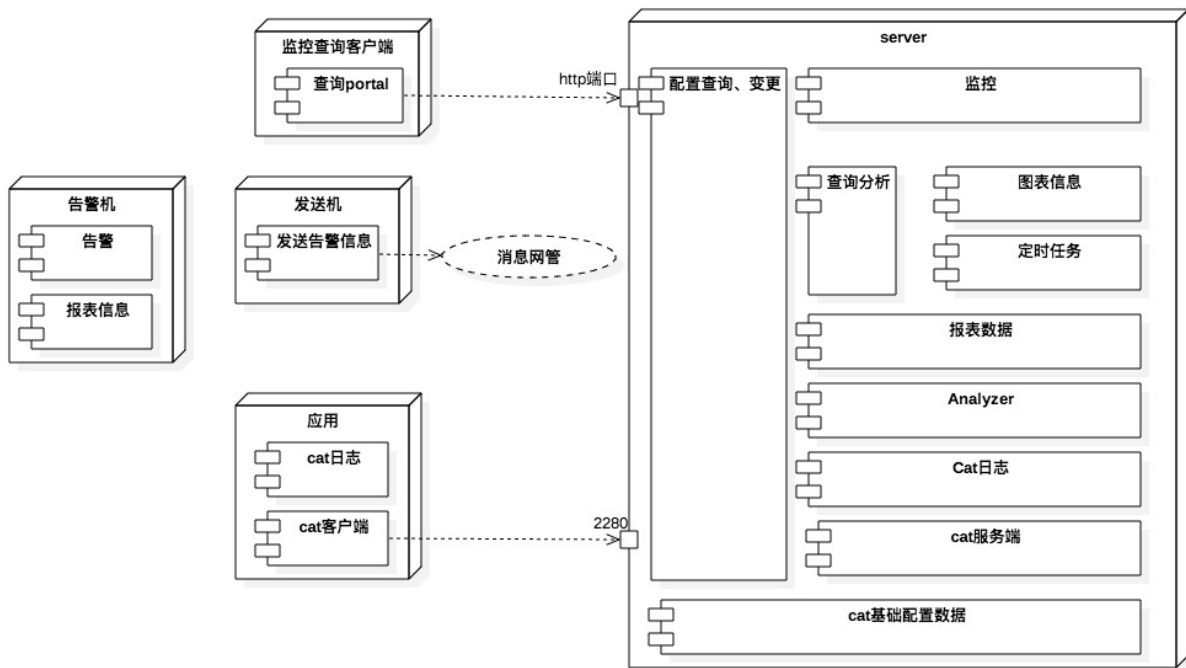
淘宝“鹰眼”



京东的 "Hydra"



大众点评的 "CAT"



微博Watchman

