# 使用 Zipkin 和 Brave 实现分布式系统追踪（基础篇） - 推酷

http://www.tuicool.com/articles/f2qAZnZ

# 使用 Zipkin 和 Brave 实现分布式系统追踪（基础篇）

时间 2016-05-21 22:25:53  [显影之尘](http://www.tuicool.com/sites/nYrEnqN)

原文  [http://www.tangrui.net/implement-distributed-tracking-system-with-zipkin-and-brave/](http://www.tangrui.net/implement-distributed-tracking-system-with-zipkin-and-brave/?utm_source=tuicool&utm_medium=referral)

主题 [分布式系统](http://www.tuicool.com/topics/11000146)[MySQL](http://www.tuicool.com/topics/11030000)

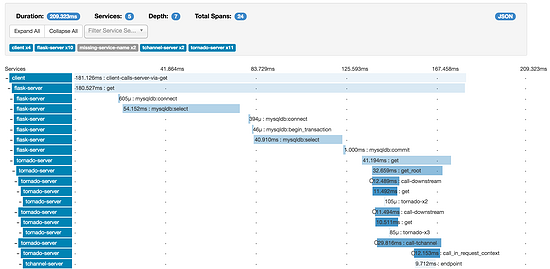


## 一、Zipkin

## 1.1、简介

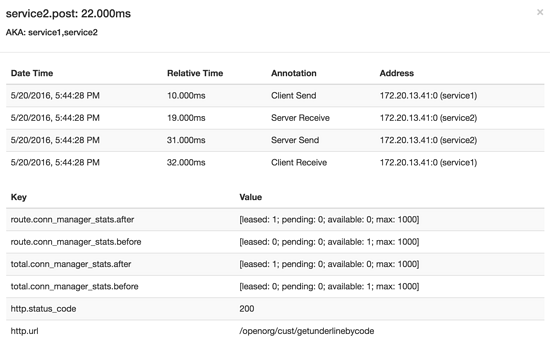
[***Zipkin***](http://zipkin.io/) 是一款开源的分布式实时数据追踪系统（Distributed Tracking System），基于 [***Google Dapper***](http://research.google.com/pubs/pub36356.html) 的论文设计而来，由 Twitter 公司开发贡献。其主要功能是聚集来自各个异构系统的实时监控数据，用来追踪微服务架构下的系统延时问题。

应用系统需要进行装备（instrument）以向 Zipkin 报告数据。Zipkin 的用户界面可以呈现一幅关联图表，以显示有多少被追踪的请求通过了每一层应用。



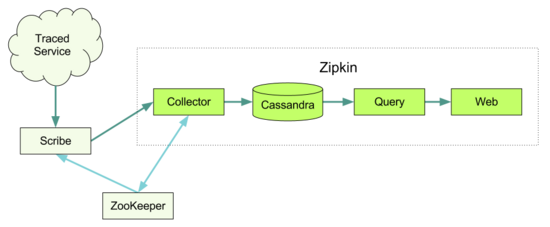
Zipkin 以 Trace 结构表示对一次请求的追踪，又把每个 Trace 拆分为若干个有依赖关系的 Span。在微服务架构中，一次用户请求可能会由后台若干个服务负责处理，那么每个处理请求的服务就可以理解为一个 Span（可以包括 API 服务，缓存服务，数据库服务以及报表服务等）。当然这个服务也可能继续请求其他的服务，因此 Span 是一个树形结构，以体现服务之间的调用关系。

Zipkin 的用户界面除了可以查看 Span 的依赖关系之外，还以瀑布图的形式显示了每个 Span 的耗时情况，可以一目了然的看到各个服务的性能状况。打开每个 Span，还有更详细的数据以键值对的形式呈现，而且这些数据可以在装备应用的时候自行添加。



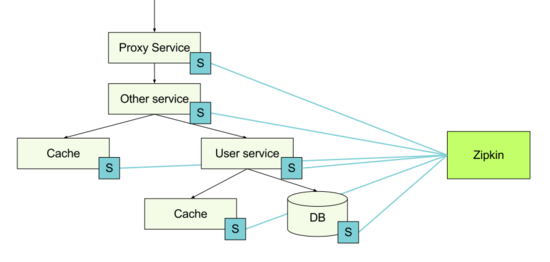
从图中可以看出如下的调用关系：整个调用链中有两个微服务 service1 和 service2，在 10ms（相对时间点）的时候，service1 作为客户端向 service2 发送了一个请求（Client Send），之后 service2 服务于 19ms 的时候收到请求（Server Receive），并用了 12ms 的时间来处理，并于 31ms 时刻将数据返回（Server Send），最后 service1 服务于 1ms 以后接收到此数据（Client Receive），因此整个过程共耗时 22ms。图中还给出了 service1 访问 service2 服务前后 Http Client 连接池的状态信息。

## 1.2、架构



如图所示，Zipkin 主要由四部分构成：收集器、数据存储、查询以及 Web 界面。Zipkin 的收集器负责将各系统报告过来的追踪数据进行接收；而数据存储默认使用 Cassandra，也可以替换为 MySQL；查询服务用来向其他服务提供数据查询的能力，而 Web 服务是官方默认提供的一个图形用户界面。

而各个异构的系统服务向 Zipkin 报告数据的架构如下图。



## 1.3、运行

使用 Docker 运行 Zipkin 最为简单，其过程如下：

gitclone https://github.**com**/openzipkin/docker-zipkin

**cd** docker-zipkin

docker-composeup

这样启动，默认会使用 Cassandra 数据库，如果想改用 MySQL，可以换做以下命令启动：

docker-compose -f docker-compose.yml -f docker-compose-mysql.yml up

启动成功以后，可以通过 http:// :8080 来访问。具体获取 IP 地址的方法请参阅 Docker 的相关文档。

## 二、Brave

## 2.1、简介

Brave 是用来装备 Java 程序的类库，提供了面向 Standard Servlet、Spring MVC、Http Client、JAX RS、Jersey、Resteasy 和 MySQL 等接口的装备能力，可以通过编写简单的配置和代码，让基于这些框架构建的应用可以向 Zipkin 报告数据。同时 Brave 也提供了非常简单且标准化的接口，在以上封装无法满足要求的时候可以方便扩展与定制。

## 2.2、初始化

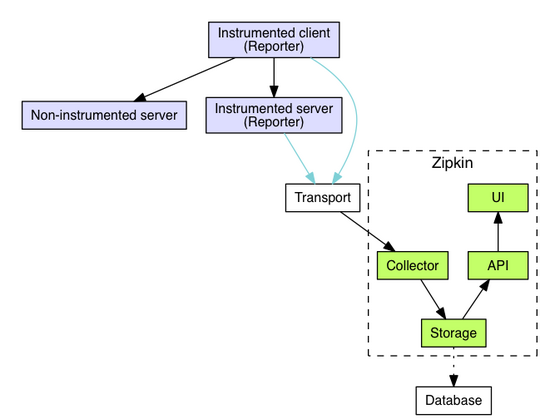
Brave 的初始化就是要构建 Brave 类的实例，该库提供了 Builder 类用来完成这件事情。

注：下文中约定，大写的 Brave 指该 Java 类库，而 Brave 类指 com.github.kristofa.brave.Brave 类型，而小写的 brave 指该类型的实例。

Brave.Builderbuilder = **new** Brave.Builder("serviceName");

Bravebrave = builder.build();

其中的 serviceName 是当前服务的名称，这个名称会出现在所有跟该服务有关的 Span 中。默认情况下，Brave 不会将收集到的监控数据发送给 Zipkin 服务器，而是会以日志的形式打印到控制台。如果需要将数据发送给服务器，就需要引入 HttpSpanCollector 类。当前版本（3.8.0）将这个类命名为 Collector，这个概念容易跟 Zipkin 自身的 Collector 相混淆，因此在 Issue [***#173***](https://github.com/openzipkin/brave/issues/173) 中官方建议将其更名为 Reporter，也就是说这个类是用来向 Zipkin 的 Collector 报告数据的。



使用 HttpSpanCollector 的方法如下：

Brave.Builderbuilder = **new** Brave.Builder("serviceName");

builder.spanCollector(HttpSpanCollector.create(

    "http://localhost:9411",

**new** EmptySpanCollectorMetricsHandler()));

Bravebrave = builder.build();

使用 HttpSpanCollector.create 方法可以创建该类的一个对象，第一个参数就是 Zipkin 服务的地址（默认部署时的端口为 9411）。

如果使用 Spring 的话，为了方便扩展，建议添加一个名为 ZipkinBraveFactoryBean 的类，其内容大致如下：

package net.tangrui.example.brave;

// 省略所有的 import

public **class** **ZipkinBraveFactoryBean** **implements** **FactoryBean**<**Brave**> {

  private final String serviceName;

  private final String zipkinHost;

  private Braveinstance;

  public void setServiceName(final String serviceName) {

**this**.serviceName = serviceName;

  }

  public void setZipkinHost(final String zipkinHost) {

**this**.zipkinHost = zipkinHost;

  }

  private void createInstance() {

**if** (**this**.serviceName == null) {

**throw** **new** BeanInitializationException("Property serviceName

must be set.");

    }

    Brave.Builderbuilder = **new** Brave.Builder(**this**.serviceName);

**if** (**this**.zipkinHost != null && !"".equals(**this**.zipkinHost)) {

      builder.spanCollector(HttpSpanCollector.create(

**this**.zipkinHost, **new** EmptySpanCollectorMetricsHandler()));

    }

**this**.instance = builder.build();

  }

  @Override

  public BravegetObject() throws Exception {

**if** (**this**.instance == null) {

**this**.createInstance();

    }

**return** **this**.instance;

  }

  @Override

  public Class<?> getObjectType() {

**return** Brave.class;

  }

  @Override

  public boolean isSingleton() {

**return** true;

  }

}

然后只需要在 application-context.xml 配置文件中使用该 FactoryBean 就可以了：

<beanid="brave"

**class**="net.tangrui.example.brave.ZipkinBraveFactoryBean"

  p:serviceName="serviceName"

  p:zipkinHost="http://localhost:9411"/>

## 2.3、装备标准的 Servlet 应用

Brave 提供了 brave-web-servlet-filter 模块，可以为标准的 Servlet 应用添加向 Zipkin 服务器报告数据的能力，需要做的就是在 web.xml 文件增加一个 BraveServletFilter。

不过这个 Filter 在初始化的时候需要传入几个参数，这些参数可以通过 brave 对象的对应方法获得，但是注入这些构造参数，最简单的办法还是使用 Spring 提供的 DelegatingFilterProxy。

在 web.xml 中添加如下内容（最好配置为第一个 Filter，以便从请求最开始就记录数据）：

<**filter**>

  <**filter-name**>braveFilter</**filter-name**>

  <**filter-class**>

    org.springframework.web.filter.DelegatingFilterProxy

  </**filter-class**>

  <**init-param**>

    <**param-name**>targetFilterLifecycle</**param-name**>

    <**param-value**>true</**param-value**>

  </**init-param**>

</**filter**>

<**filter-mapping**>

  <**filter-name**>braveFilter</**filter-name**>

  <**url-pattern**>/\*</**url-pattern**>

  <**dispatcher**>REQUEST</**dispatcher**>

  <**dispatcher**>FORWARD</**dispatcher**>

  <**dispatcher**>INCLUDE</**dispatcher**>

  <**dispatcher**>ERROR</**dispatcher**>

</**filter-mapping**>

然后在配置文件中添加以下内容（创建 brave Bean 的有关代码请参考上文）：

<!-- 注意：这里的 id 要使用和 web.xml 中的 filter-name 同样的值 -->

<**beanid="braveFilter"**

  class="com.github.kristofa.brave.servlet.BraveServletFilter">

  <**constructor-arg**

    value="#{brave.serverRequestInterceptor()}"/>

  <**constructor-arg**

    value="#{brave.serverResponseInterceptor()}"/>

  <**constructor-arg**>

    <**bean**

      class="com.github.kristofa.brave.http.DefaultSpanNameProvider"/>

  </**constructor-arg**>

</**bean**>

最后一个类 com.github.kristofa.brave.http.DefaultSpanNameProvider 存在于 brave-http 模块中。当使用 Maven 或 Gradle 来管理项目的话，brave-http 会随着 brave-web-servlet-filter 的引入被自动关联进来。

一切无误的话就可以启动服务。如果给定了 zipkinHost 参数，数据就会被发送到指定的 Zipkin 服务器上，然后可以在其 Web 界面上看到相关内容；否则会有类似如下的信息打印到系统控制台（做了格式美化）：

{

  "traceId": "27bf14862307cd99",

  "name": "post",

  "id": "d79a683e2900c293",

  "parentId": "27bf14862307cd99",

  "timestamp": 1.463737111294e+15,

  "duration": 772000,

  "annotations": [

    {

      "endpoint": {

        "serviceName": "service1",

        "ipv4": "172.20.13.41"

      },

      "timestamp": 1.463737111294e+15,

      "value": "cs"

    },

    {

      "endpoint": {

        "serviceName": "service1",

        "ipv4": "172.20.13.41"

      },

      "timestamp": 1.463737112066e+15,

      "value": "cr"

    }

  ],

  "binaryAnnotations": [

    {

      "key": "route.conn\_manager\_stats.after",

      "value": "[leased: 1; pending: 0; available: 0; max: 1000]",

      "endpoint": {

        "serviceName": "service1",

        "ipv4": "172.20.13.41"

      }

    },

    {

      "key": "route.conn\_manager\_stats.before",

      "value": "[leased: 0; pending: 0; available: 0; max: 1000]",

      "endpoint": {

        "serviceName": "service1",

        "ipv4": "172.20.13.41"

      }

    },

    {

      "key": "total.conn\_manager\_stats.after",

      "value": "[leased: 1; pending: 0; available: 0; max: 1000]",

      "endpoint": {

        "serviceName": "service1",

        "ipv4": "172.20.13.41"

      }

    },

    {

      "key": "total.conn\_manager\_stats.before",

      "value": "[leased: 0; pending: 0; available: 0; max: 1000]",

      "endpoint": {

        "serviceName": "service1",

        "ipv4": "172.20.13.41"

      }

    }

  ]

}

## 2.3、装备 Spring MVC 应用

Brave 自带了 brave-spring-web-servlet-interceptor 模块，因此装备 Spring MVC 项目变得非常容易，只需要在配置文件中添加一些 interceptor 就好了：

<**mvc:interceptors**>

  <**bean**

    class="com.github.kristofa.brave.spring.ServletHandlerInterceptor">

    <**constructor-argvalue="#**{brave.serverRequestInterceptor()}"/>

    <**constructor-argvalue="#**{brave.serverResponseInterceptor()}"/>

    <**constructor-arg**>

      <**bean**

        class="com.github.kristofa.brave.http.DefaultSpanNameProvider"/>

    </**constructor-arg**>

    <**constructor-argvalue="#**{brave.serverSpanThreadBinder()}"/>

</**bean**>

</**mvc:interceptors**>

## 2.4、装备 MySQL 服务

brave-mysql 模块在 JDBC 驱动层面添加了一些拦截器，可以对 MySQL 的查询进行监控。在使用之前也需要通过 Spring 进行一下配置。

<bean

**class**="com.github.kristofa.brave.mysql.MySQLStatementInterceptorManagementBean" destroy-method="close">

  <**constructor-argvalue="#{brave.clientTracer()}"**/>

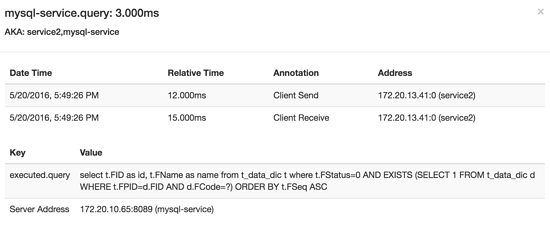
</**bean**>

该配置的目的是要给 MySQLStatementInterceptorManagementBean 类注入一个 ClientTracer 实例，这个实例会在后来的 MySQL JDBC 驱动的拦截器中被使用。初始化完成以后只需要在连接字符串中添加如下参数就可以了：

？statementInterceptors=**com**.github.kristofa.brave.mysql.MySQLStatementInterceptor&zipkinServiceName=myDatabaseService

其中的 zipkinServiceName 用来指定该 MySQL 服务的名称，如果省略的话，会默认以 mysql-${databaseName} 的形式来呈现。

这里需要特别说明一点，因为 MySQL 服务是跟 Java 服务分离的，因此上文初始化 brave 对象时提供的服务名称，并不适用于 MySQL 服务，所以才需要在这里另外指定。



可以看出，添加了 statement interceptor 之后，可以看到 service2 请求 MySQL 查询的起止时间，以及执行的 SQL 语句等信息。

## 三、总结

本文主要介绍了 Zipkin 服务和其 Java 库 Brave 的一些基本概念及原理，并且针对 Brave 开箱提供的一些装备组件进行了详细的使用说明。在后面进阶篇的文章中，会对如何扩展 Brave 以实现自定义监控信息的内容进行介绍，敬请期待