liaokailin / **zipkin**

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data. https://github.com/liaokailin/zipkin

4 commits **1** branch **0** releases **1** contributor

Branch: **master** ▾ | **New pull request** | **Find file** | **Clone or download** ▾

liaokailin zipkin Latest commit 6a22d36 on 31 Jul 2016

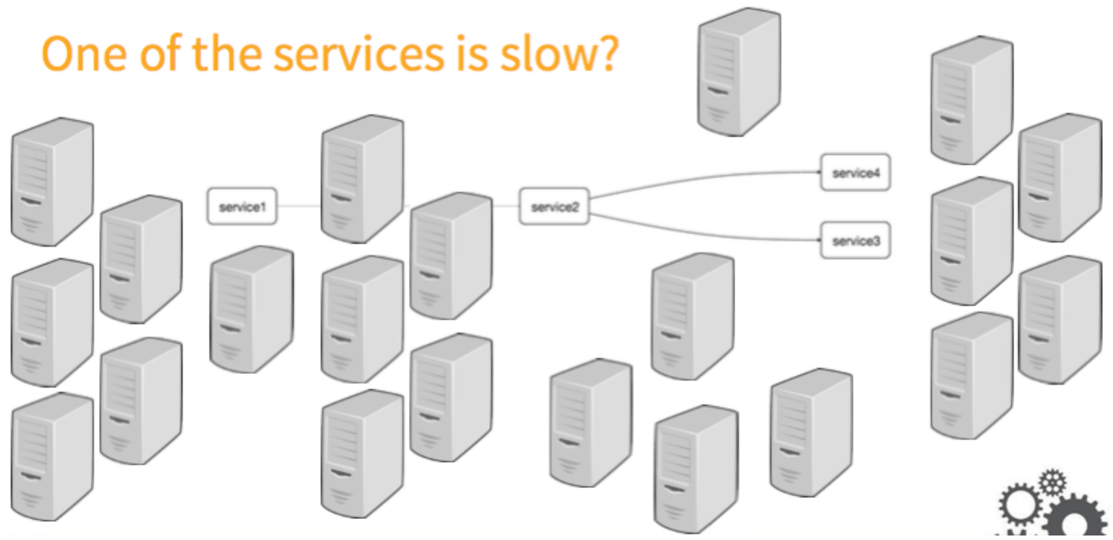
src/main	zipkin brave client	2 years ago
.gitignore	zipkin brave client	2 years ago
README.md	zipkin	2 years ago
pom.xml	zipkin brave client	2 years ago

README.md

zipkin

`zipkin` 为分布式链路调用监控系统，聚合各业务系统调用延迟数据，达到链路调用监控跟踪。

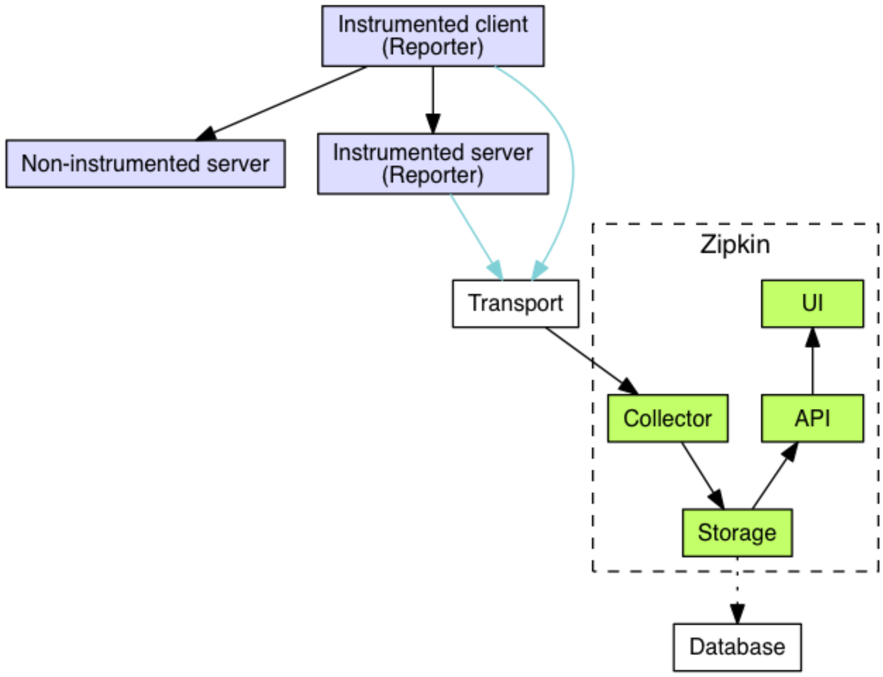
architecture



如图，在复杂的调用链路中假设存在一条调用链路响应缓慢，如何定位其中延迟高的服务呢？

- 日志： 通过分析调用链路上的每个服务日志得到结果

- zipkin：使用 `zipkin` 的 `web UI` 可以一眼看出延迟高的服务



如图所示，各业务系统在彼此调用时，将特定的跟踪消息传递至 `zipkin` ,zipkin在收集到跟踪信息后将其聚合处理、存储、展示等，用户可通过 `web UI` 方便 获得网络延迟、调用链路、系统依赖等等。

zipkin

`zipkin` 主要涉及四个组件 `collector` `storage` `search` `web UI`

- `Collector` 接收各service传输的数据
- `Cassandra` 作为 `Storage` 的一种，也可以是mysql等，默认存储在内存中，配置 `cassandra` 可以参考这里
- `Query` 负责查询 `Storage` 中存储的数据,提供简单的 `JSON API` 获取数据，主要提供给 `web UI` 使用
- `Web` 提供简单的web界面

install

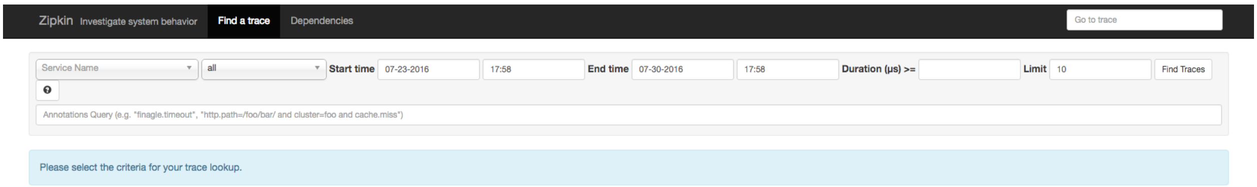
执行如下命令下载jar包

```
wget -O zipkin.jar 'https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exe
```

其为一个 `spring boot` 功能，直接运行jar

```
nohup java -jar zipkin.jar &
```

访问 http://ip:9411



Service Name
all
Start time 07-23-2016 17:58
End time 07-30-2016 17:58
Duration (μs) >=
Limit 10
Annotations Query (e.g. "finagle.timeout", "http.path=/foo/bar/ and cluster=foo and cache.miss")

Please select the criteria for your trace lookup.

terminology

使用 `zipkin` 涉及几个概念

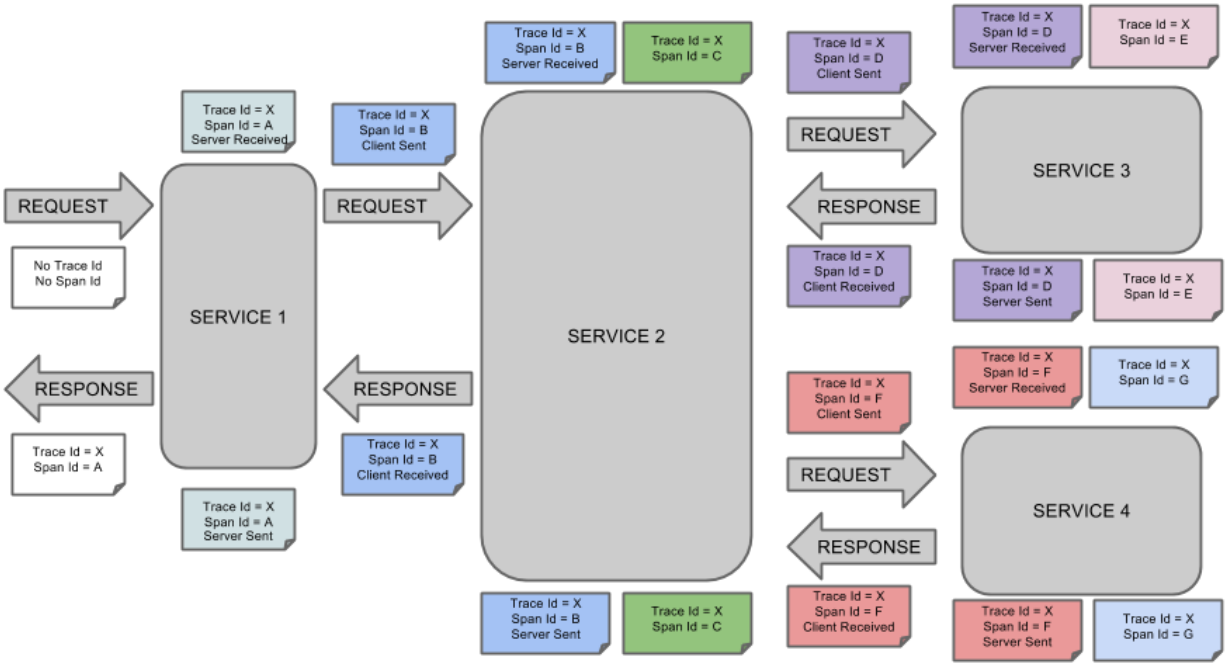
- `Span` :基本工作单元，一次链路调用(可以是RPC，DB等没有特定的限制)创建一个 `span` ，通过一个64位ID标识它， `span` 通过还有其他的数据，例如描述信息，时间戳，key-value对的(Annotation)tag信息， `parent-id` 等,其中 `parent-id` 可以表示 `span` 调用链路来源，通俗的理解 `span` 就是一次请求信息

- `Trace` :类似于树结构的 `Span` 集合，表示一条调用链路，存在唯一标识
- `Annotation` : 注解,用来记录请求特定事件相关信息(例如时间)，通常包含四个注解信息

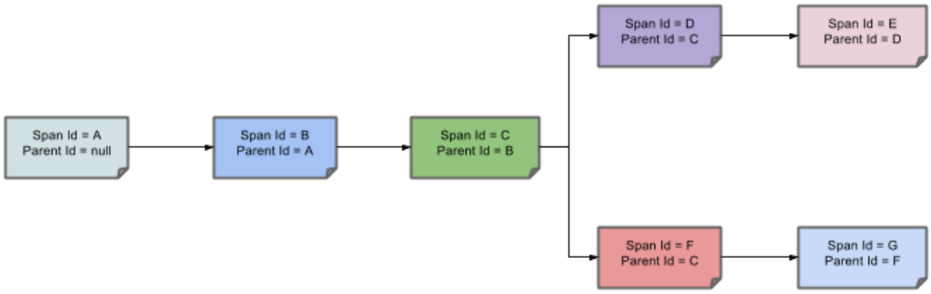
  > > cs - Client Start,表示客户端发起请求
  >
  > > sr - Server Receive,表示服务端收到请求
  >
  > > ss - Server Send,表示服务端完成处理，并将结果发送给客户端
  >
  > > cr - Client Received,表示客户端获取到服务端返回信息

- `BinaryAnnotation` :提供一些额外信息，一般已key-value对出现

概念说完，来看下完整的调用链路

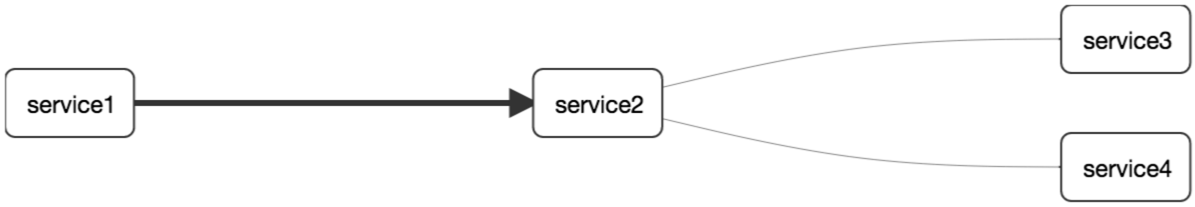


上图表示一请求链路，一条链路通过 `Trace Id` 唯一标识， `Span` 标识发起的请求信息，各 `span` 通过 `parent id` 关联起来，

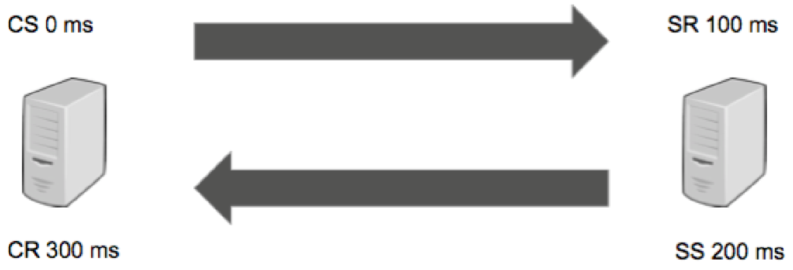


如图

整个链路的依赖关系如下:



完成链路调用的记录后，如何来计算调用的延迟呢，这就需要利用 `Annotation` 信息



> sr-cs 得到请求发出延迟

> ss-sr 得到服务端处理延迟

> cr-cs 得到真个链路完成延迟

brave

作为各调用链路，只需要负责将指定格式的数据发送给 `zipkin` 即可，利用brave可快捷完成操作。

首先导入jar包 `pom.xml`

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.6.RELEASE</version>
    </parent>



    <!-- https://mvnrepository.com/artifact/io.zipkin.brave/brave-core -->
    <dependencies>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>

        <dependency>
            <groupId>io.zipkin.brave</groupId>
            <artifactId>brave-core</artifactId>
            <version>3.9.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/io.zipkin.brave/brave-http -->
        <dependency>
            <groupId>io.zipkin.brave</groupId>
            <artifactId>brave-http</artifactId>
            <version>3.9.0</version>
        </dependency>
        <dependency>
            <groupId>io.zipkin.brave</groupId>
            <artifactId>brave-spancollector-http</artifactId>
            <version>3.9.0</version>
        </dependency>
        <dependency>
            <groupId>io.zipkin.brave</groupId>
            <artifactId>brave-web-servlet-filter</artifactId>
            <version>3.9.0</version>
        </dependency>

        <dependency>
            <groupId>io.zipkin.brave</groupId>
            <artifactId>brave-okhttp</artifactId>
            <version>3.9.0</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.13</version>
        </dependency>
        <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
            <version>4.5.1</version>
        </dependency>

    </dependencies>
```

利用 `spring boot` 创建工程

`Application.java`

```
package com.lkl.zipkin;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 *
 * Created by liaokailin on 16/7/27.
 */
@SpringBootApplication
public class Application {


    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(Application.class);
        app.run(args);


    }
}
```

建立 `controller` 对外提供服务

`HomeController.java`

```
RestController
@RequestMapping("/")
public class HomeController {

    @Autowired
    private OkHttpClient client;

    private  Random random = new Random();

    @RequestMapping("start")
    public String start() throws InterruptedException, IOException {
        int sleep= random.nextInt(100);
        TimeUnit.MILLISECONDS.sleep(sleep);
        Request request = new Request.Builder().url("http://localhost:9090/foo").get().build();
        Response response = client.newCall(request).execute();
        return " [service1 sleep " + sleep+" ms]" + response.body().toString();
    }

```

`HomeController` 中利用 `OkHttpClient` 调用发起http请求。在每次发起请求时则需要通过 `brave` 记录 `Span` 信息，并异步传递给 `zipkin` 作为被调用方(服务端)也同样需要完成以上操作.

`ZipkinConfig.java`

```
package com.lkl.zipkin.config;

import com.github.kristofa.brave.Brave;
import com.github.kristofa.brave.EmptySpanCollectorMetricsHandler;
import com.github.kristofa.brave.SpanCollector;
import com.github.kristofa.brave.http.DefaultSpanNameProvider;
import com.github.kristofa.brave.http.HttpSpanCollector;
import com.github.kristofa.brave.okhttp.BraveOkHttpRequestResponseInterceptor;
import com.github.kristofa.brave.servlet.BraveServletFilter;
import okhttp3.OkHttpClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Created by liaokailin on 16/7/27.
 */
@Configuration
public class ZipkinConfig {

    @Autowired
    private ZipkinProperties properties;


    @Bean
    public SpanCollector spanCollector() {
        HttpSpanCollector.Config config = HttpSpanCollector.Config.builder().connectTimeout(properties.getC
                .compressionEnabled(properties.isCompressionEnabled()).flushInterval(properties.getFlushInt
        return HttpSpanCollector.create(properties.getUrl(), config, new EmptySpanCollectorMetricsHandler()
    }


    @Bean
    public Brave brave(SpanCollector spanCollector){
        Brave.Builder builder = new Brave.Builder(properties.getServiceName());  //指定state
        builder.spanCollector(spanCollector);
        builder.traceSampler(Sampler.ALWAYS_SAMPLE);
        Brave brave = builder.build();
        return brave;
    }

    @Bean
    public BraveServletFilter braveServletFilter(Brave brave){
        BraveServletFilter filter = new BraveServletFilter(brave.serverRequestInterceptor(),brave.serverRes
        return filter;
    }

    @Bean
    public OkHttpClient okHttpClient(Brave brave){
        OkHttpClient client = new OkHttpClient.Builder()
                .addInterceptor(new BraveOkHttpRequestResponseInterceptor(brave.clientRequestInterceptor(),
                .build();
        return client;
    }
}
```

- `SpanCollector` 配置收集器
- `Brave` 各工具类的封装,其中 `builder.traceSampler(Sampler.ALWAYS_SAMPLE)` 设置采样比率，0-1之间的百分比
- `BraveServletFilter` 作为拦截器，需要 `serverRequestInterceptor` , `serverResponseInterceptor` 分别完成 `sr` 和 `ss` 操作
- `OkHttpClient` 添加拦截器，需要 `clientRequestInterceptor` , `clientResponseInterceptor` 分别完成 `cs` 和 `cr` 操作,该功能由 brave中的 `brave-okhttp` 模块提供，同样的道理如果需要记录数据库的延迟只要在数据库操作前后完成 `cs` 和 `cr` 即可，当然brave提供其封装。

以上还缺少一个配置信息 `ZipkinProperties.java`

```
package com.lkl.zipkin.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

/**
 * Created by liaokailin on 16/7/28.
 */
```

```
@Configuration
@ConfigurationProperties(prefix = "com.zipkin")
public class ZipkinProperties {

    private String serviceName;

    private String url;

    private int connectTimeout;

    private int readTimeout;

    private int flushInterval;

    private boolean compressionEnabled;

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public int getConnectTimeout() {
        return connectTimeout;
    }

    public void setConnectTimeout(int connectTimeout) {
        this.connectTimeout = connectTimeout;
    }

    public int getReadTimeout() {
        return readTimeout;
    }

    public void setReadTimeout(int readTimeout) {
        this.readTimeout = readTimeout;
    }

    public int getFlushInterval() {
        return flushInterval;
    }

    public void setFlushInterval(int flushInterval) {
        this.flushInterval = flushInterval;
    }

    public boolean isCompressionEnabled() {
        return compressionEnabled;
    }

    public void setCompressionEnabled(boolean compressionEnabled) {
        this.compressionEnabled = compressionEnabled;
    }

    public String getServiceName() {
        return serviceName;
    }

    public void setServiceName(String serviceName) {
        this.serviceName = serviceName;
    }
}
```

则可以在配置文件 `application.properties` 中配置相关信息

```
com.zipkin.serviceName=service1
com.zipkin.url=http://110.173.14.57:9411
com.zipkin.connectTimeout=6000
com.zipkin.readTimeout=6000
com.zipkin.flushInterval=1
com.zipkin.compressionEnabled=true
server.port=8080
```

那么其中的 `service1` 即完成，同样的道理，修改配置文件(调整 `com.zipkin.serviceName` ,以及 `server.port`)以及 `controller` 对应的方法构造若干服务

`service1` 中访问 `http://localhost:8080/start` 需要访问 `http://localhost:9090/foo` ,则构造 `server2` 提供该方法

`server2` 配置

```
com.zipkin.serviceName=service2
com.zipkin.url=http://110.173.14.57:9411
com.zipkin.connectTimeout=6000
com.zipkin.readTimeout=6000
com.zipkin.flushInterval=1
com.zipkin.compressionEnabled=true


server.port=9090
```

`controller` 方法

```
    @RequestMapping("foo")
    public String foo() throws InterruptedException, IOException {
        Random random = new Random();
        int sleep= random.nextInt(100);
        TimeUnit.MILLISECONDS.sleep(sleep);
        Request request = new Request.Builder().url("http://localhost:9091/bar").get().build();  //service3
        Response response = client.newCall(request).execute();
        String result = response.body().string();
        request = new Request.Builder().url("http://localhost:9092/tar").get().build();  //service4
        response = client.newCall(request).execute();
       result += response.body().string();
        return " [service2 sleep " + sleep+" ms]" + result;
    }
```

在 `server2` 中调用 `server3` 和 `server4` 中的方法

方法分别为

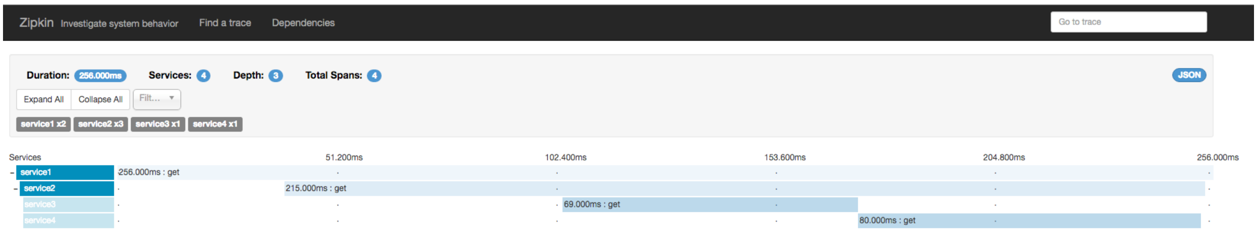
```
 @RequestMapping("bar")
    public String bar() throws InterruptedException, IOException {  //service3 method
        Random random = new Random();
        int sleep= random.nextInt(100);
        TimeUnit.MILLISECONDS.sleep(sleep);
        return " [service3 sleep " + sleep+" ms]";
    }

    @RequestMapping("tar")
    public String tar() throws InterruptedException, IOException { //service4 method
        Random random = new Random();
        int sleep= random.nextInt(1000);
        TimeUnit.MILLISECONDS.sleep(sleep);
        return " [service4 sleep " + sleep+" ms]";
    }
```

将工程修改后编译成 `jar` 形式

执行

```
nohup java -jar server4.jar &
nohup java -jar server3.jar &
nohup java -jar server2.jar &
nohup java -jar server1.jar &
```

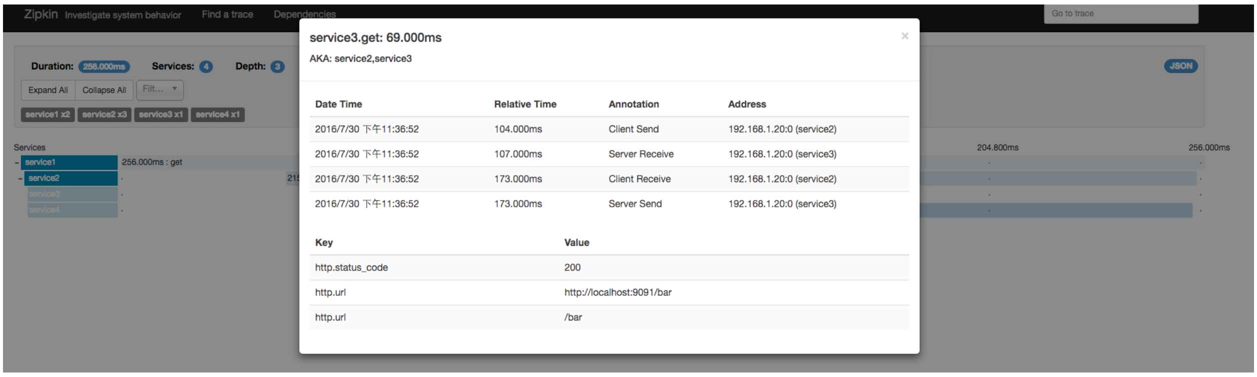
访问 `http://localhost:8080/start` 后查看 `zipkin` 的 `web UI`



Duration: 256.000ms Services: 4 Depth: 3 Total Spans: 4

service1 x2 service2 x3 service3 x1 service4 x1

点击条目可以查看具体的延迟信息



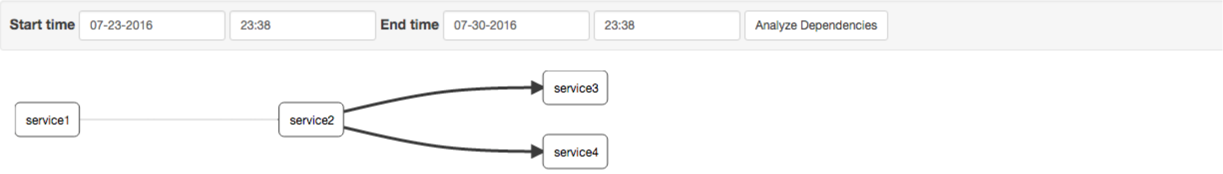
service3.get: 69.000ms

AKA: service2,service3

Date Time	Relative Time	Annotation	Address
2016/7/30 下午11:36:52	104.000ms	Client Send	192.168.1.20:0 (service2)
2016/7/30 下午11:36:52	107.000ms	Server Receive	192.168.1.20:0 (service3)
2016/7/30 下午11:36:52	173.000ms	Client Receive	192.168.1.20:0 (service2)
2016/7/30 下午11:36:52	173.000ms	Server Send	192.168.1.20:0 (service3)

Key	Value
http.status_code	200
http.url	http://localhost:9091/bar
http.url	/bar

服务之间的依赖为



Start time 07-23-2016 23:38 End time 07-30-2016 23:38

brave 源码

以上完成了基本的操作，下面将从源码角度来看下 `brave` 的实现

首先从 `SpanCollector` 来入手

```
@Bean
    public SpanCollector spanCollector() {
        HttpSpanCollector.Config config = HttpSpanCollector.Config.builder().connectTimeout(properties.getC
                .compressionEnabled(properties.isCompressionEnabled()).flushInterval(properties.getFlushInt
        return HttpSpanCollector.create(properties.getUrl(), config, new EmptySpanCollectorMetricsHandler()
    }
```

从名称上看 `HttpSpanCollector` 是基于 `http` 的 `span` 收集器,因此超时配置是必须的，默认给出的超时时间较长， `flushInterval` 表示 `span` 的传递 间隔，实际为定时任务执行的间隔时间.在 `HttpSpanCollector` 中覆写了父类方法 `sendSpans`

```
@Override
  protected void sendSpans(byte[] json) throws IOException {
    // intentionally not closing the connection, so as to use keep-alives
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    connection.setConnectTimeout(config.connectTimeout());
    connection.setReadTimeout(config.readTimeout());
    connection.setRequestMethod("POST");
    connection.addRequestProperty("Content-Type", "application/json");
    if (config.compressionEnabled()) {
      connection.addRequestProperty("Content-Encoding", "gzip");
      ByteArrayOutputStream gzipped = new ByteArrayOutputStream();
      try (GZIPOutputStream compressor = new GZIPOutputStream(gzipped)) {
        compressor.write(json);
      }
      json = gzipped.toByteArray();
    }
    connection.setDoOutput(true);
    connection.setFixedLengthStreamingMode(json.length);
    connection.getOutputStream().write(json);

    try (InputStream in = connection.getInputStream()) {
      while (in.read() != -1) ; // skip
    } catch (IOException e) {
      try (InputStream err = connection.getErrorStream()) {
        if (err != null) { // possible, if the connection was dropped
          while (err.read() != -1) ; // skip
        }
      }
      throw e;
    }
  }
}
```

可以看出最终 `span` 信息是通过 `HttpURLConnection` 实现的，同样道理就可以推理 `brave` 对 `brave-spring-resttemplate-interceptors` 模块的实现， 只是换了一种 `http` 封装。

`Brave`

```
 @Bean
    public Brave brave(SpanCollector spanCollector){
        Brave.Builder builder = new Brave.Builder(properties.getServiceName());  //指定state
        builder.spanCollector(spanCollector);
        builder.traceSampler(Sampler.ALWAYS_SAMPLE);
        Brave brave = builder.build();
        return brave;
    }
```

`Brave` 类包装了各种工具类

```
public Brave build() {
            return new Brave(this);
        }
```

创建一个 `Brave`

```
private Brave(Builder builder) {
        serverTracer = ServerTracer.builder()
                .randomGenerator(builder.random)
                .spanCollector(builder.spanCollector)
                .state(builder.state)
                .traceSampler(builder.sampler).build();

        clientTracer = ClientTracer.builder()
                .randomGenerator(builder.random)
                .spanCollector(builder.spanCollector)
                .state(builder.state)
                .traceSampler(builder.sampler).build();

        localTracer = LocalTracer.builder()
                .randomGenerator(builder.random)
                .spanCollector(builder.spanCollector)
                .spanAndEndpoint(SpanAndEndpoint.LocalSpanAndEndpoint.create(builder.state))
                .traceSampler(builder.sampler).build();

        serverRequestInterceptor = new ServerRequestInterceptor(serverTracer);
        serverResponseInterceptor = new ServerResponseInterceptor(serverTracer);
        clientRequestInterceptor = new ClientRequestInterceptor(clientTracer);
        clientResponseInterceptor = new ClientResponseInterceptor(clientTracer);
        serverSpanAnnotationSubmitter = AnnotationSubmitter.create(SpanAndEndpoint.ServerSpanAndEndpoint.cr
        serverSpanThreadBinder = new ServerSpanThreadBinder(builder.state);
        clientSpanThreadBinder = new ClientSpanThreadBinder(builder.state);
    }
```

封装了 `*Tracer` , `*Interceptor` , `*Binder` 等

其中 `serverTracer` 当服务作为 `服务端` 时处理 `span` 信息， `clientTracer` 当服务作为 `客户端` 时处理 `span` 信息

`Filter`

`BraveServletFilter` 是 `http` 模块提供的拦截器功能，传递 `serverRequestInterceptor` , `serverResponseInterceptor` , `spanNameProvider` 等参数 其中 `spanNameProvider` 表示如何处理 `span` 的名称，默认使用 `method` 名称, `spring boot` 中申明的 `filter bean` 默认拦截所有请求

```
@Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) throws

        String alreadyFilteredAttributeName = getAlreadyFilteredAttributeName();
        boolean hasAlreadyFilteredAttribute = request.getAttribute(alreadyFilteredAttributeName) != null;

        if (hasAlreadyFilteredAttribute) {
            // Proceed without invoking this filter...
            filterChain.doFilter(request, response);
        } else {

            final StatusExposingServletResponse statusExposingServletResponse = new StatusExposingServletRe
            requestInterceptor.handle(new HttpServerRequestAdapter(new ServletHttpServerRequest((HttpServle

            try {
                filterChain.doFilter(request, statusExposingServletResponse);
            } finally {
                responseInterceptor.handle(new HttpServerResponseAdapter(new HttpResponse() {
                    @Override
                    public int getHttpStatusCode() {
                        return statusExposingServletResponse.getStatus();
                    }
                }));
            }
        }
    }
```

首先来看 `requestInterceptor.handle` 方法,

```
 public void handle(ServerRequestAdapter adapter) {
        serverTracer.clearCurrentSpan();
        final TraceData traceData = adapter.getTraceData();

        Boolean sample = traceData.getSample();
        if (sample != null && Boolean.FALSE.equals(sample)) {
            serverTracer.setStateNoTracing();
            LOGGER.fine("Received indication that we should NOT trace.");
        } else {
            if (traceData.getSpanId() != null) {
                LOGGER.fine("Received span information as part of request.");
                SpanId spanId = traceData.getSpanId();
                serverTracer.setStateCurrentTrace(spanId.traceId, spanId.spanId,
                        spanId.nullableParentId(), adapter.getSpanName());
            } else {
                LOGGER.fine("Received no span state.");
                serverTracer.setStateUnknown(adapter.getSpanName());
            }
            serverTracer.setServerReceived();
            for(KeyValueAnnotation annotation : adapter.requestAnnotations())
            {
                serverTracer.submitBinaryAnnotation(annotation.getKey(), annotation.getValue());
            }
        }
    }
```

其中 `serverTracer.clearCurrentSpan()` 清除当前线程上的 `span` 信息，调用 `ThreadLocalServerClientAndLocalSpanState` 中的

```
  @Override
    public void setCurrentServerSpan(final ServerSpan span) {
        if (span == null) {
            currentServerSpan.remove();
        } else {
            currentServerSpan.set(span);
        }
    }
```

`currentServerSpan` 为 `ThreadLocal` 对象

```
private final static ThreadLocal<ServerSpan> currentServerSpan = new ThreadLocal<ServerSpan>() {
```

回到 `ServerRequestInterceptor#handle()` 方法中 `final TraceData traceData = adapter.getTraceData()`

```
 @Override
    public TraceData getTraceData() {
        final String sampled = serverRequest.getHttpHeaderValue(BraveHttpHeaders.Sampled.getName());
        if (sampled != null) {
            if (sampled.equals("0") || sampled.toLowerCase().equals("false")) {
                return TraceData.builder().sample(false).build();
            } else {
                final String parentSpanId = serverRequest.getHttpHeaderValue(BraveHttpHeaders.ParentSpanId.
                final String traceId = serverRequest.getHttpHeaderValue(BraveHttpHeaders.TraceId.getName())
                final String spanId = serverRequest.getHttpHeaderValue(BraveHttpHeaders.SpanId.getName());

                if (traceId != null && spanId != null) {
                    SpanId span = getSpanId(traceId, spanId, parentSpanId);
                    return TraceData.builder().sample(true).spanId(span).build();
                }
            }
        }
        return TraceData.builder().build();
    }
```

其中 `SpanId span = getSpanId(traceId, spanId, parentSpanId)` 将构造一个 `SpanId` 对象

```
 private SpanId getSpanId(String traceId, String spanId, String parentSpanId) {
        return SpanId.builder()
            .traceId(convertToLong(traceId))
            .spanId(convertToLong(spanId))
            .parentId(parentSpanId == null ? null : convertToLong(parentSpanId)).build();
   }
```

将 `traceId` , `spanId` , `parentId` 关联起来，其中设置 `parentId` 方法为

```
public Builder parentId(@Nullable Long parentId) {
      if (parentId == null) {
        this.flags |= FLAG_IS_ROOT;
      } else {
        this.flags &= ~FLAG_IS_ROOT;
      }
      this.parentId = parentId;
      return this;
    }
```

如果 `parentId` 为空为根节点，则执行 `this.flags |= FLAG_IS_ROOT` ,因此后续在判断节点是否为根节点时，只需要执行 `(flags & FLAG_IS_ROOT) == FLAG_IS_ROOT` 即可.

构造完 `SpanId` 后看

```
    serverTracer.setStateCurrentTrace(spanId.traceId, spanId.spanId,
                        spanId.nullableParentId(), adapter.getSpanName());
```

设置当前 `Span`

```
 public void setStateCurrentTrace(long traceId, long spanId, @Nullable Long parentSpanId, @Nullable String
        checkNotBlank(name, "Null or blank span name");
        spanAndEndpoint().state().setCurrentServerSpan(
            ServerSpan.create(traceId, spanId, parentSpanId, name));
    }
```

`ServerSpan.create` 创建 `Span` 信息

```
 static ServerSpan create(long traceId, long spanId, @Nullable Long parentSpanId, String name) {
        Span span = new Span();
        span.setTrace_id(traceId);
        span.setId(spanId);
        if (parentSpanId != null) {
            span.setParent_id(parentSpanId);
        }
        span.setName(name);
        return create(span, true);
    }
```

构造了一个包含`Span`信息的`AutoValue_ServerSpan`对象

通过`setCurrentServerSpan`设置到当前线程上

继续看`serverTracer.setServerReceived()`方法

```
public void setServerReceived() {
        submitStartAnnotation(zipkinCoreConstants.SERVER_RECV);
    }
```

为当前请求设置了`server received event`

```
void submitStartAnnotation(String annotationName) {
        Span span = spanAndEndpoint().span();
        if (span != null) {
            Annotation annotation = Annotation.create(
                currentTimeMicroseconds(),
                annotationName,
                spanAndEndpoint().endpoint()
            );
            synchronized (span) {
                span.setTimestamp(annotation.timestamp);
                span.addToAnnotations(annotation);
            }
        }
    }
```

在这里为`Span`信息设置了`Annotation`信息,后续的

```
 for(KeyValueAnnotation annotation : adapter.requestAnnotations())
            {
                serverTracer.submitBinaryAnnotation(annotation.getKey(), annotation.getValue());
            }
```

设置了`BinaryAnnotation`信息，`adapter.requestAnnotations()`在构造`HttpServerRequestAdapter`时已完成

```
 @Override
    public Collection<KeyValueAnnotation> requestAnnotations() {
        KeyValueAnnotation uriAnnotation = KeyValueAnnotation.create(
                TraceKeys.HTTP_URL, serverRequest.getUri().toString());
        return Collections.singleton(uriAnnotation);
    }
```

以上将`Span`信息(包括sr)存储在当前线程中，接下来继续看`BraveServletFilter#doFilter`方法的`finally`部分

```
 responseInterceptor.handle(new HttpServerResponseAdapter(new HttpResponse() {
                   @Override  //获取http状态码
                   public int getHttpStatusCode() {
                       return statusExposingServletResponse.getStatus();
                   }
               }));
```

`handle`方法

```
 public void handle(ServerResponseAdapter adapter) {
        // We can submit this in any case. When server state is not set or
        // we should not trace this request nothing will happen.
        LOGGER.fine("Sending server send.");
        try {
            for(KeyValueAnnotation annotation : adapter.responseAnnotations())
            {
                serverTracer.submitBinaryAnnotation(annotation.getKey(), annotation.getValue());
            }
            serverTracer.setServerSend();
        } finally {
            serverTracer.clearCurrentSpan();
        }
    }
```

首先配置 `BinaryAnnotation` 信息，然后执行 `serverTracer.setServerSend` ,在 `finally` 中清除当前线程中的 `Span` 信息(不管前面是否清楚成功,最终都将执行该不走)， `ThreadLocal` 中的数据要做到有始有终

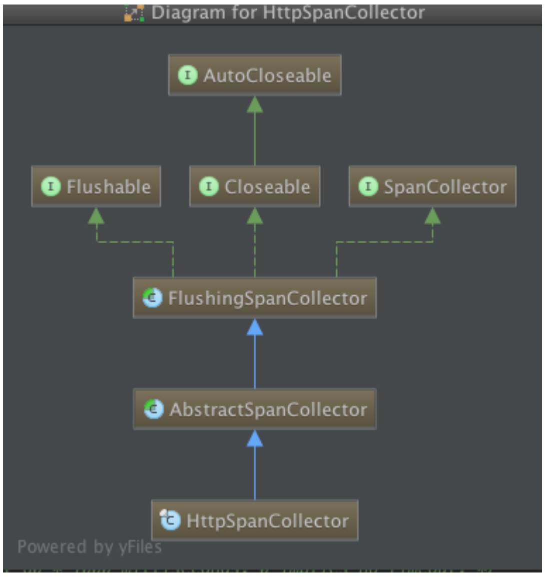
看 `serverTracer.setServerSend()`

```
public void setServerSend() {
        if (submitEndAnnotation(zipkinCoreConstants.SERVER_SEND, spanCollector())) {
            spanAndEndpoint().state().setCurrentServerSpan(null);
        }
    }
```

终于看到 `spanCollector` 收集器了，说明下面将看是收集 `Span` 信息,这里为 `ss` 注解

```
boolean submitEndAnnotation(String annotationName, SpanCollector spanCollector) {
        Span span = spanAndEndpoint().span();
        if (span == null) {
          return false;
        }
        Annotation annotation = Annotation.create(
            currentTimeMicroseconds(),
            annotationName,
            spanAndEndpoint().endpoint()
        );
        span.addToAnnotations(annotation);
        if (span.getTimestamp() != null) {
            span.setDuration(annotation.timestamp - span.getTimestamp());
        }
        spanCollector.collect(span);
        return true;
    }
```

首先获取当前线程中的 `Span` 信息，然后处理注解信息，通过 `annotation.timestamp - span.getTimestamp()` 计算延迟, 调用 `spanCollector.collect(span)` 进行收集 `Span` 信息，那么 `Span` 信息是同步收集的吗？肯定不是的，接着看



调用 `spanCollector.collect(span)` 则执行 `FlushingSpanCollector` 中的 `collect` 方法

```
@Override
  public void collect(Span span) {
    metrics.incrementAcceptedSpans(1);
    if (!pending.offer(span)) {
      metrics.incrementDroppedSpans(1);
    }
  }
```

首先进行的是 `metrics` 统计信息，可以自定义该 `SpanCollectorMetricsHandler` 信息收集各指标信息,利用如 `grafana` 等展示信息

`pending.offer(span)` 将 `span` 信息存储在 `BlockingQueue` 中，然后通过定时任务去取出阻塞队列中的值，偷偷摸摸的上传 `span` 信息

定时任务利用了 `Flusher` 类来执行，在构造 `FlushingSpanCollector` 时构造了 `Flusher` 类

```
 static final class Flusher implements Runnable {
    final Flushable flushable;
    final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

    Flusher(Flushable flushable, int flushInterval) {
      this.flushable = flushable;
      this.scheduler.scheduleWithFixedDelay(this, 0, flushInterval, SECONDS);
    }

    @Override
    public void run() {
      try {
        flushable.flush();
      } catch (IOException ignored) {
      }
    }
  }
```

创建了一个核心线程数为1的线程池，每间隔 `flushInterval` 秒执行一次 `Span` 信息上传，执行 `flush` 方法

```
@Override
  public void flush() {
    if (pending.isEmpty()) return;
    List<Span> drained = new ArrayList<Span>(pending.size());
    pending.drainTo(drained);
    if (drained.isEmpty()) return;

    int spanCount = drained.size();
    try {
      reportSpans(drained);
    } catch (IOException e) {
      metrics.incrementDroppedSpans(spanCount);
    } catch (RuntimeException e) {
      metrics.incrementDroppedSpans(spanCount);
    }
  }
```

首先将阻塞队列中的值全部取出存如集合中，最后调用 `reportSpans(List<Span> drained)` 抽象方法，该方法在 `AbstractSpanCollector` 得到覆写

```
@Override
  protected void reportSpans(List<Span> drained) throws IOException {
    byte[] encoded = codec.writeSpans(drained);
    sendSpans(encoded);
  }
```

转换成字节流后调用 `sendSpans` 抽象方法发送 `Span` 信息，此时就回到一开始说的 `HttpSpanCollector` 通过 `HttpURLConnection` 实现的 `sendSpans` 方法。

more about is here