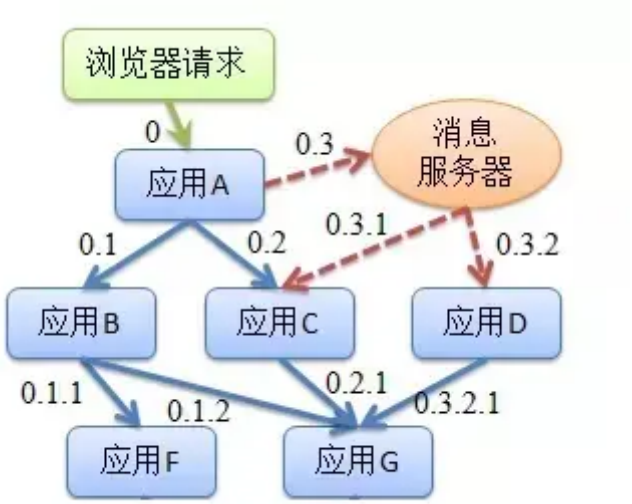
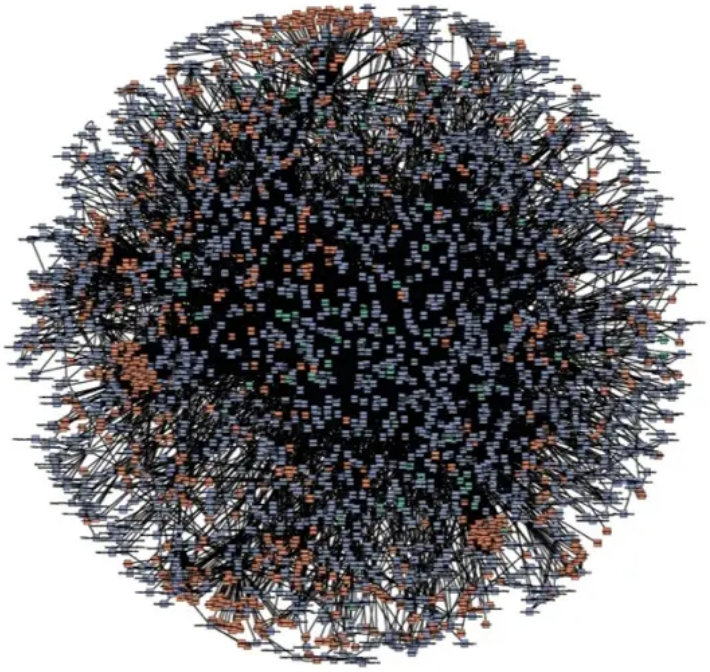
一、关于链路追踪

随着微服务架构的流行，服务按照不同的维度进行拆分，一次请求往往需要涉及到多个服务。互联网应用构建在不同的软件模块集上，这些软件模块，有可能是由不同的团队开发、可能使用不同的编程语言来实现、有可能布在了几千台服务器，横跨多个不同的数据中心。因此，就需要一些可以帮助理解系统行为、用于分析性能问题的工具，以便发生故障的时候，能够快速定位和解决问题。在复杂的微服务架构系统中，几乎每一个前端请求都会形成一个复杂的分布式服务调用链路。一个请求完整调用链可能如下图所示：



随着服务的越来越多，对调用链的分析会越来越复杂。它们之间的调用关系也许如下：



随着业务规模不断增大、服务不断增多以及频繁变更的情况下，面对复杂的调用链路就带来一系列问题：

- 如何快速发现问题？
- 如何判断故障影响范围？
- 如何梳理服务依赖以及依赖的合理性？
- 如何分析链路性能问题以及实时容量规划？

而链路追踪的出现正是为了解决这种问题，它可以在复杂的服务调用中定位问题，还可以在新人加入后台团队之后，让其清楚地知道自己所负责的服务在哪一环。

除此之外，如果某个接口突然耗时增加，也不必再逐个服务查询耗时情况，我们可以直观地分析出服务的性能瓶颈，方便在流量激增的情况下精准合理地扩容。

“链路追踪”一词是在 2010 年提出的，当时谷歌发布了一篇 Dapper 论文：Dapper，大规模分布式系统的跟踪系统，介绍了谷歌自研的分布式链路追踪的实现原理，还介绍了他们是怎么低成本实现对应用透明的。

单纯的理解链路追踪，就是指一次任务的开始到结束，期间调用的所有系统及耗时（时间跨度）都可以完整记录下来。

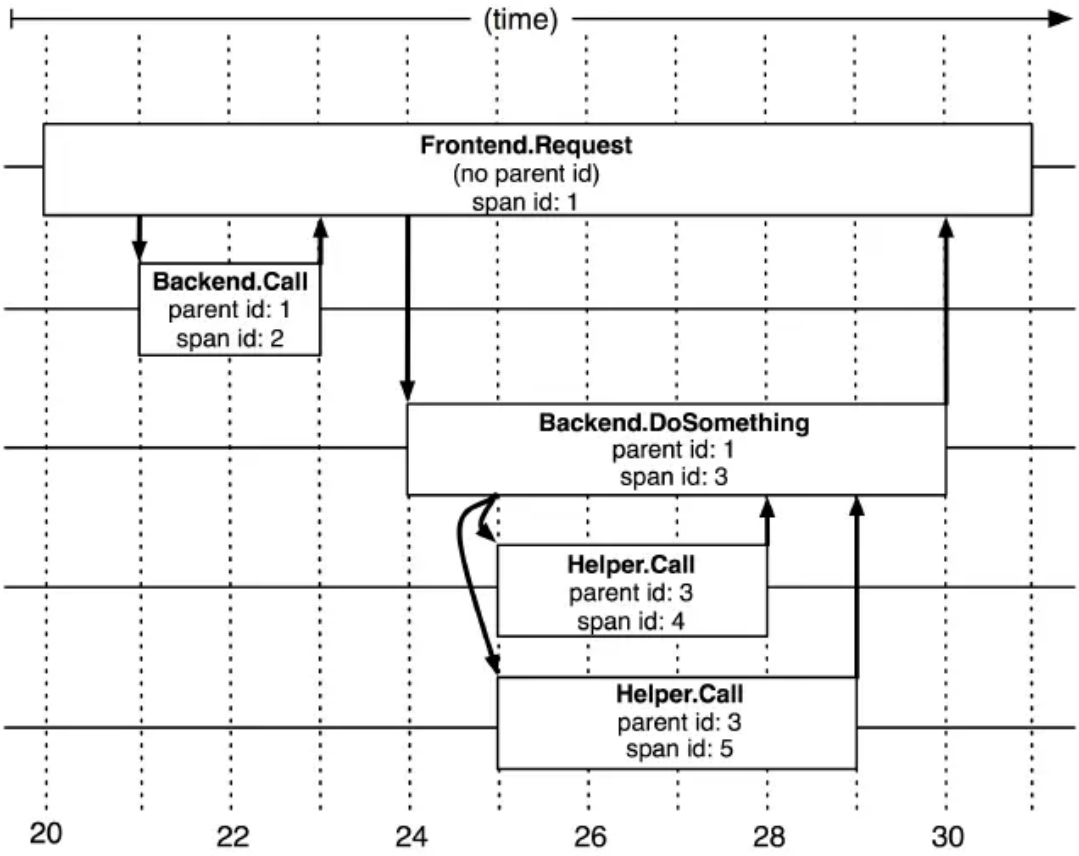
其实 Dapper 一开始只是一个独立的调用链路追踪系统，后来逐渐演化成了监控平台，并且基于监控平台孕育出了很多工具，比如实时预警、过载保护、指标数据查询等。

除了谷歌的 Dapper，还有一些其他比较有名的产品，比如阿里的鹰眼、大众点评的 CAT、Twitter 的 Zipkin、Naver（著名社交软件LINE的母公司）的 PinPoint 以及国产开源的 SkyWalking（已贡献给 Apache） 等。

二、Dapper

1、Span

基本工作单位，一次单独的调用链可以称为一个 Span，Dapper 记录的是 Span 的名称，以及每个 Span 的 ID 和父 ID，以重建在一次追踪过程中不同 Span 之间的关系，图中一个矩形框就是一个 Span，前端从发出请求到收到回复就是一个 Span。

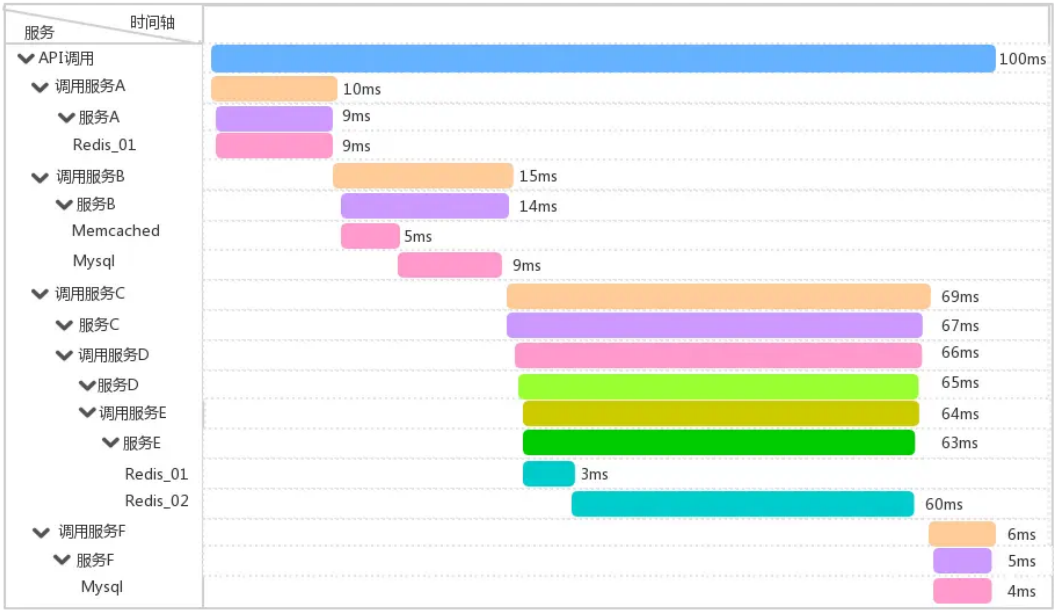


Dapper 记录了 span 名称，以及每个 span 的 ID 和父 span ID，以重建在一次追踪过程中不同 span 之间的关系。如果一个 span 没有父 ID 被称为 root span。所有 span 都挂在一个特定的 Trace 上，也共用一个 trace id。

2、Trace

一系列 Span 组成的树状结构，一个 Trace 认为是一次完整的链路，内部包含 n 多个 Span。Trace 和 Span 存在一对多的关系，Span 与 Span 之间存在父子关系。

举个例子：客户端调用服务 A 、服务 B 、服务 C 、服务 F，而每个服务例如 C 就是一个 Span，如果在服务 C 中另起线程调用了 D，那么 D 就是 C 的子 Span，如果在服务 D 中另起线程调用了 E，那么 E 就是 D 的子 Span，这个 C -> D -> E 的链路就是一条 Trace。如果链路追踪系统做好了，链路数据有了，借助前端解析和渲染工具，可以达到下图中的效果：



3、Annotation

用来及时记录一个事件的存在，一些核心 annotations 用来定义一个请求的开始和结束。

- cs - Client Sent：客户端发起一个请求，这个 annotation 描述了这个 span 的开始；
- sr - Server Received：服务端获得请求并准备开始处理它，如果 sr 减去 cs 时间戳便可得到网络延迟；
- ss - Server Sent：请求处理完成（当请求返回客户端），如果 ss 减去 sr 时间戳便可得到服务端处理请求需要的时间；
- cr - Client Received：表示 span 结束，客户端成功接收到服务端的回复，如果 cr 减去 cs 时间戳便可得到客户端从服务端获取回复的所有所需时间。

三、Spring Cloud Sleuth

Spring Cloud Sleuth 为 Spring Cloud 实现了分布式跟踪解决方案。兼容 Zipkin，HTrace 和其他基于日志的追踪系统，例如 ELK（Elasticsearch 、Logstash、 Kibana）。

Spring Cloud Sleuth 提供了以下功能：

- 链路追踪：通过 Sleuth 可以很清楚的看出一个请求都经过了那些服务，可以很方便的理清服务间的调用关系等。
- 性能分析：通过 Sleuth 可以很方便的看出每个采样请求的耗时，分析哪些服务调用比较耗时，当服务调用的耗时随着请求量的增大而增大时， 可以对服务的扩容提供一定的提醒。
- 数据分析，优化链路：对于频繁调用一个服务，或并行调用等，可以针对业务做一些优化措施。
- 可视化错误：对于程序未捕获的异常，可以配合 Zipkin 查看。

官方文档：https://docs.spring.io/spring-cloud-sleuth/docs/current/reference/html/index.html

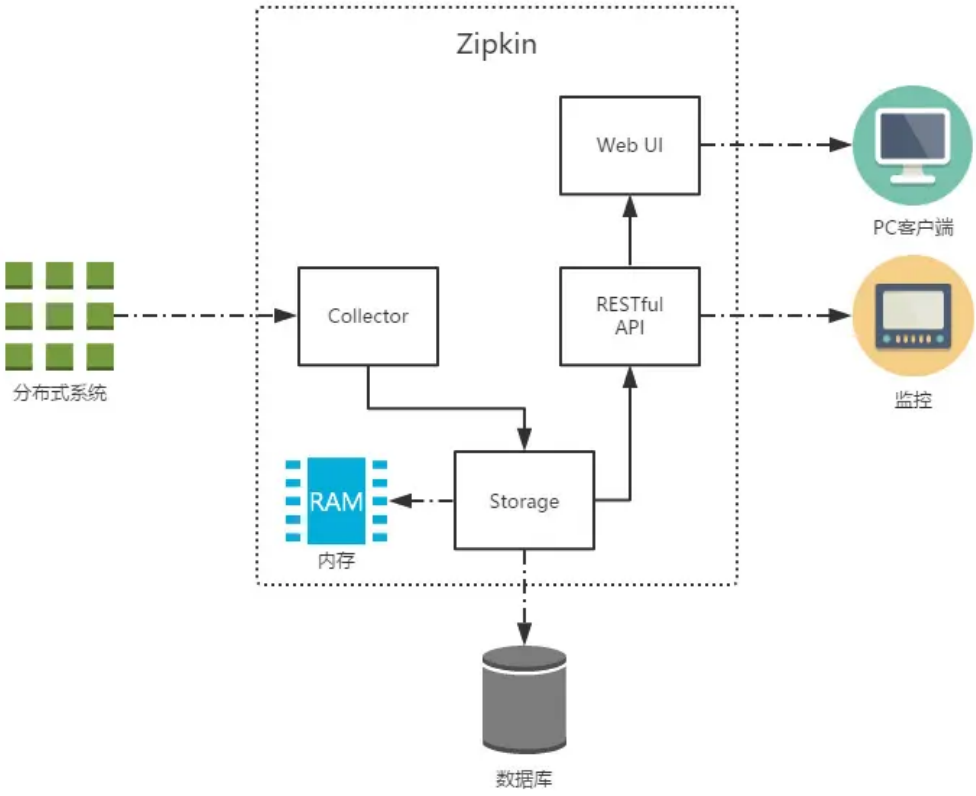
四、Zipkin

Zipkin 是 Twitter 公司开发贡献的一款开源的分布式实时数据追踪系统（Distributed Tracking System），基于 Google Dapper 的论文设计而来，其主要功能是聚集各个异构系统的实时监控数据。

它可以收集各个服务器上请求链路的跟踪数据，并通过 Rest API 接口来辅助我们查询跟踪数据，实现对分布式系统的实时监控，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，每个服务向 Zipkin 报告计时数据，Zipkin 会根据调用关系生成依赖关系图，帮助我们直观的搜索跟踪信息和分析请求链路明细。Zipkin 提供了可插拔数据存储方式：In-Memory、MySql、Cassandra 以及 Elasticsearch。

分布式跟踪系统还有其他比较成熟的实现，例如：Naver 的 PinPoint、Apache 的 HTrace、阿里的鹰眼 Tracing、京东的 Hydra、新浪的 Watchman，美团点评的 CAT，Apache 的 SkyWalking 等。

共有四个组件构成了 Zipkin：



- Collector：收集器组件，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，处理收集器接收到的跟踪信息，默认将信息存储在内存中，可以修改存储策略使用其他存储组件，支持 MySQL，Elasticsearch 等。
- Web UI：UI 组件，基于 API 组件实现的上层应用，提供 Web 页面，用来展示 Zipkin 中的调用链和系统依赖关系等。
- RESTful API：API 组件，为 Web 界面提供查询存储中数据的接口。

官网地址：https://zipkin.io/

四、使用

1、添加依赖

```
...
```

```
<!--zipkin链路追踪-->
<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>
<!--mysql监控依赖-->
<dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-instrumentation-mysql8</artifactId>
    <version>5.10.1</version>
</dependency>
...
```

2、配置

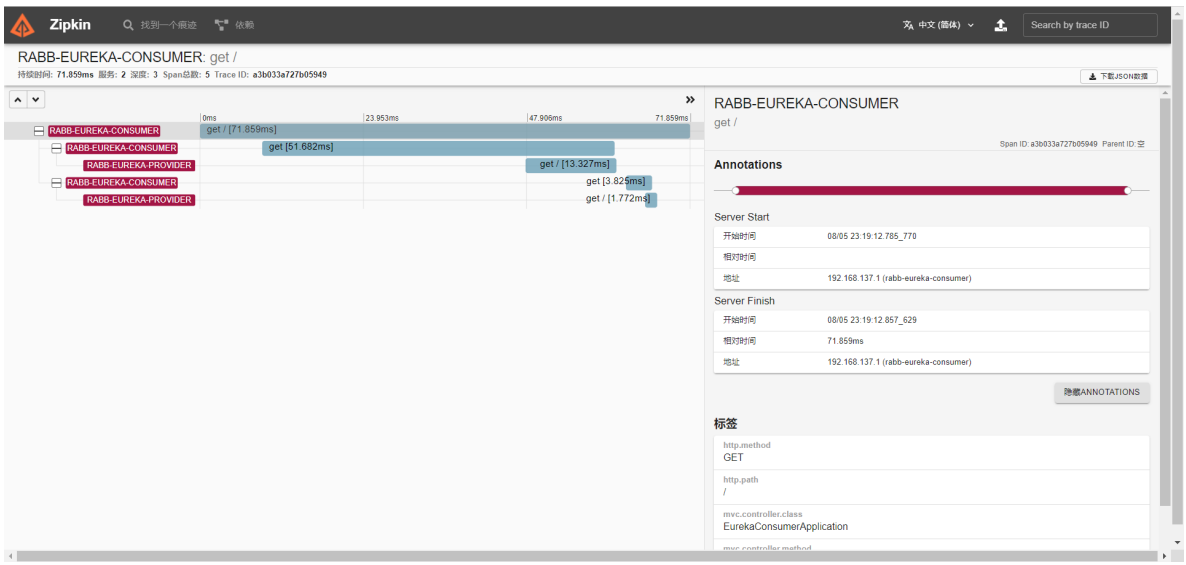
```
...
spring:
  zipkin:
    base-url: http://localhost:9411/ # 服务端地址
    sender:
      type: web                      # 数据传输方式，web 表示以 HTTP 报文的形式向服务端发送数据
  sleuth:
    sampler:
      probability: 1.0               # 收集数据百分比，默认 0.1（10%）
...
```

3、启动zipking

```
java -jar zipkin-server-2.23.2-exec.jar
```

4、启动应用

启动应用后调用接口，访问http://localhost:9411/



五、源码

1、自动配置类

spring-cloud-sleuth-autoconfigure-3.0.1.jar

spring.factories

```
# Auto Configuration
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

...

org.springframework.cloud.sleuth.autoconfig.instrument.web.TraceWebAutoConfigura
tion,\
org.springframework.cloud.sleuth.autoconfig.instrument.web.client.TraceWebClient
AutoConfiguration,\
org.springframework.cloud.sleuth.autoconfig.instrument.web.client.feign.TraceFei
gnClientAutoConfiguration,\
org.springframework.cloud.sleuth.autoconfig.instrument.web.client.TraceWebAsyncC
lientAutoConfiguration,\

...

# Environment Post Processor
org.springframework.boot.env.EnvironmentPostProcessor=\
org.springframework.cloud.sleuth.autoconfig.TraceEnvironmentPostProcessor,\

...
```

2、日志输出链路信息

首先看下`TraceEnvironmentPostProcessor`类，可以看到在这个类中覆盖了`logging.pattern.level`配置，因此在集成了sleuth依赖后日志输出会存在应用名称以及traceId和spanId

```
class TraceEnvironmentPostProcessor implements EnvironmentPostProcessor {
    private static final String PROPERTY_SOURCE_NAME = "defaultProperties";

    TraceEnvironmentPostProcessor() {
    }

    public void postProcessEnvironment(ConfigurableEnvironment environment,
SpringApplication application) {
        Map<String, Object> map = new HashMap();
        if
(Boolean.parseBoolean(environment.getProperty("spring.sleuth.enabled", "true")))
{
            map.put("logging.pattern.level", "%5p
[${spring.zipkin.service.name:${spring.application.name:}},%X{traceId:-},%X{span
Id:-}]");
        }

        this.addOrReplace(environment.getPropertySources(), map);
    }

    ...
}
```

3、记录trace以及span信息

在`TraceWebServletConfiguration`类中中注册了一个过滤器`TracingFilter`

```
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnSleuthWeb
@ConditionalOnWebApplication(
    type = Type.SERVLET
)
@ConditionalOnClass({HandlerInterceptorAdapter.class})
@Import({SpanCustomizingAsyncHandlerInterceptor.class})
class TraceWebServletConfiguration {
    TraceWebServletConfiguration() {
    }

    @Configuration(
        proxyBeanMethods = false
    )
    @ConditionalOnProperty(
        value = {"spring.sleuth.web.servlet.enabled"},
        matchIfMissing = true
    )
    static class ServletConfiguration {
        ServletConfiguration() {
        }

        @Bean
        TraceWebAspect traceWebAspect(Tracer tracer, CurrentTraceContext
currentTraceContext, SpanNamer spanNamer) {
            return new TraceWebAspect(tracer, currentTraceContext, spanNamer);
        }

        @Bean
        FilterRegistrationBean traceWebFilter(BeanFactory beanFactory,
SleuthWebProperties webProperties) {
            FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean(new LazyTracingFilter(beanFactory), new
ServletRegistrationBean[0]);
            filterRegistrationBean.setDispatcherTypes(DispatcherType.ASYNC, new
DispatcherType[]{DispatcherType.ERROR, DispatcherType.FORWARD,
DispatcherType.INCLUDE, DispatcherType.REQUEST});
            filterRegistrationBean.setOrder(webProperties.getFilterOrder());
            return filterRegistrationBean;
        }

        @Bean
        @ConditionalOnMissingBean
        TracingFilter tracingFilter(CurrentTraceContext currentTraceContext,
HttpServerHandler httpServerHandler) {
            return TracingFilter.create(currentTraceContext, httpServerHandler);
        }

        @Configuration(
            proxyBeanMethods = false
        )
```

```
        @ConditionalOnClass({WebMvcConfigurer.class})
        @Import({TraceWebMvcConfigurer.class})
        protected static class TraceWebMvcAutoConfiguration {
            protected TraceWebMvcAutoConfiguration() {
            }
        }
    }
}
```

过滤器中会添加trace以及span信息，并将这些信息保存到上下文中

```
public final class TracingFilter implements Filter {
    final ServletRuntime servlet = ServletRuntime.get();
    final CurrentTraceContext currentTraceContext;
    final HttpServerHandler handler;

    public static TracingFilter create(CurrentTraceContext currentTraceContext,
HttpServerHandler httpServerHandler) {
        return new TracingFilter(currentTraceContext, httpServerHandler);
    }

    TracingFilter(CurrentTraceContext currentTraceContext, HttpServerHandler
httpServerHandler) {
        this.currentTraceContext = currentTraceContext;
        this.handler = httpServerHandler;
    }

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest)request;
        HttpServletResponse res = this.servlet.httpServletResponse(response);
        TraceContext context =
(TraceContext)request.getAttribute(TraceContext.class.getName());
        if (context != null) {
            Scope scope = this.currentTraceContext.maybeScope(context);

            try {
                chain.doFilter(request, response);
            } finally {
                scope.close();
            }

        } else {
            // 从Http Request的Header里获取Span数据，如果Header中存在X-B3-TraceId,X-
B3-SpanId,X-B3-ParentSpanId属性，
            // 就说明调用链前一个节点已经生成Span，并传递下来，这时可以直接使用这些Span数
据。否则，创建一个新的Span。
            Span span = this.handler.handleReceive(new
HttpServletRequestWrapper(req));
            // 记录一些Span的属性
            request.setAttribute(SpanCustomizer.class.getName(), span);
            request.setAttribute(TraceContext.class.getName(), span.context());
            TracingFilter.SendHandled sendHandled = new
TracingFilter.SendHandled();
            request.setAttribute(TracingFilter.SendHandled.class.getName(),
sendHandled);
            Throwable error = null;
```

```
            // 保存当前的Span信息到上下文中
            Scope scope = this.currentTraceContext.newScope(span.context());
            boolean var17 = false;

            try {
                var17 = true;
                chain.doFilter(req, res);
                var17 = false;
            } catch (Throwable var22) {
                error = var22;
                throw var22;
            } finally {
                if (var17) {
                    if (this.servlet.isAsync(req)) {
                        this.servlet.handleAsync(this.handler, req, res, span);
                    } else if (sendHandled.compareAndSet(false, true)) {
                        HttpServerResponse responseWrapper =
HttpServletResponseWrapper.create(req, res, error);
                        this.handler.handleSend(responseWrapper, span);
                    }

                    scope.close();
                }
            }

            if (this.servlet.isAsync(req)) {
                this.servlet.handleAsync(this.handler, req, res, span);
            } else if (sendHandled.compareAndSet(false, true)) {
                HttpServerResponse responseWrapper =
HttpServletResponseWrapper.create(req, res, error);
                this.handler.handleSend(responseWrapper, span);
            }

            scope.close();
        }
    }

    public void destroy() {
    }

    public void init(FilterConfig filterConfig) {
    }

    static final class SendHandled extends AtomicBoolean {
        SendHandled() {
        }
    }
}
```