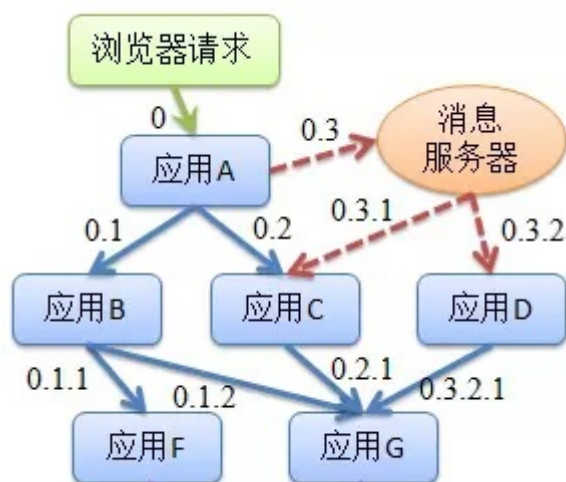
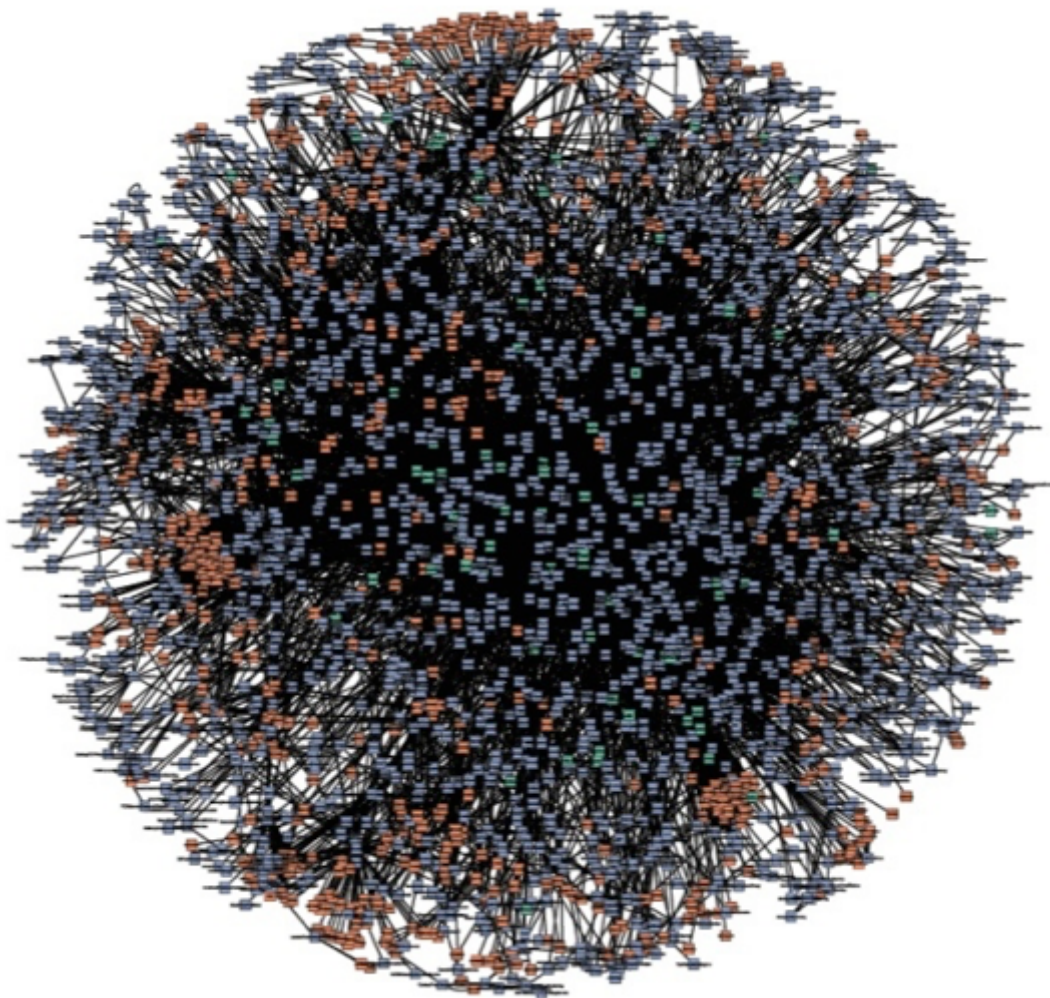


Spring Cloud Sleuth 链路追踪

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。

历史修订

本次修订日期: 2023-03-06

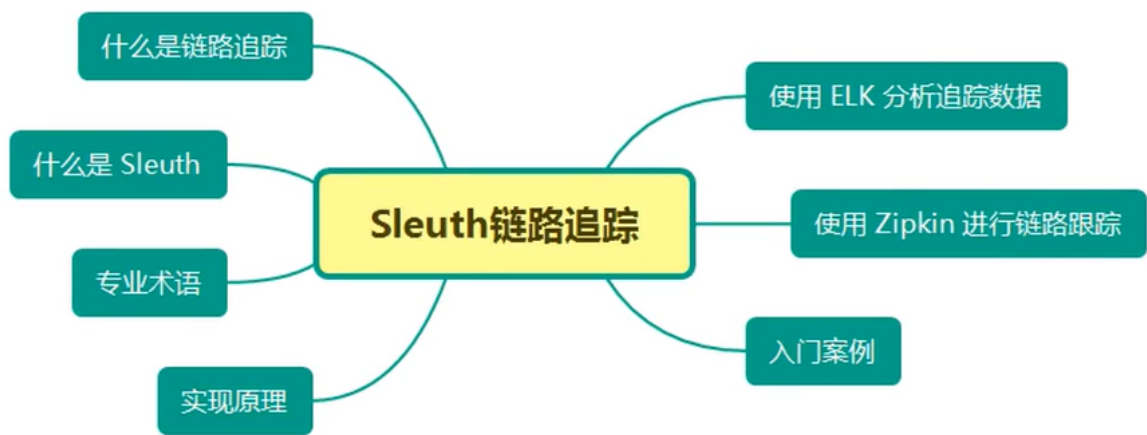
下次修订日期:

修订编号	修订日期	变更描述	说明
V0.1	2023-03-08	起草	李宗在
V0.2	2023-03-10	验证	李宗在
V0.3			

1. 技术介绍

- Spring Boot
- Spring Cloud
- Spring Cloud Gateway
- Feign
- Mybatis/Mybatis-Plus
- MySQL
- Docker
- Ubuntu
- Redis
- Postman
- Sentinel
- Nginx

2. 学习目标



3. 什么是链路追踪

“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。

4. 什么是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：

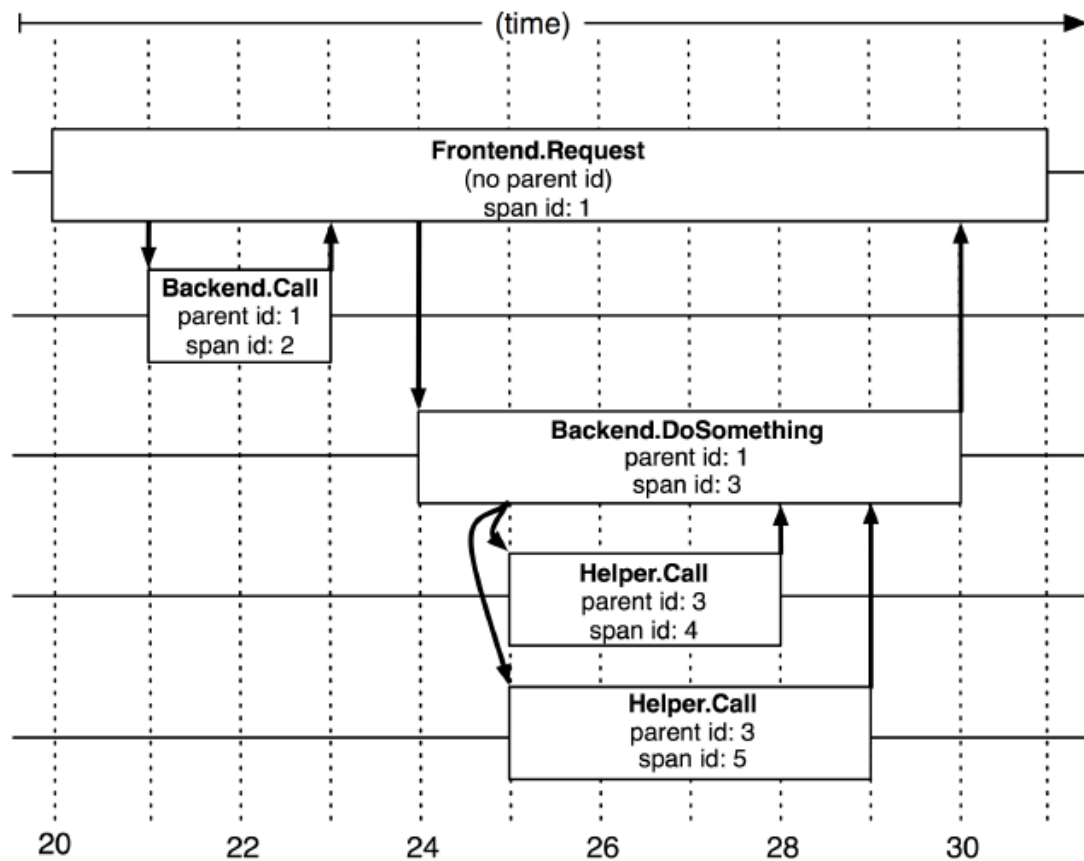
- **链路追踪**：通过 Sleuth 可以很清楚的看出一个请求都经过了那些服务，可以很方便的理清服务间的调用关系等。
- **性能分析**：通过 Sleuth 可以很方便的看出每个采样请求的耗时，分析哪些服务调用比较耗时，当服务调用的耗时随着请求量的增大而增大时，可以对服务的扩容提供一定的提醒。
- **数据分析，优化链路**：对于频繁调用一个服务，或并行调用等，可以针对业务做一些优化措施。
- **可视化错误**：对于程序未捕获的异常，可以配合 Zipkin 查看。

5. 专业术语

5.1 Span

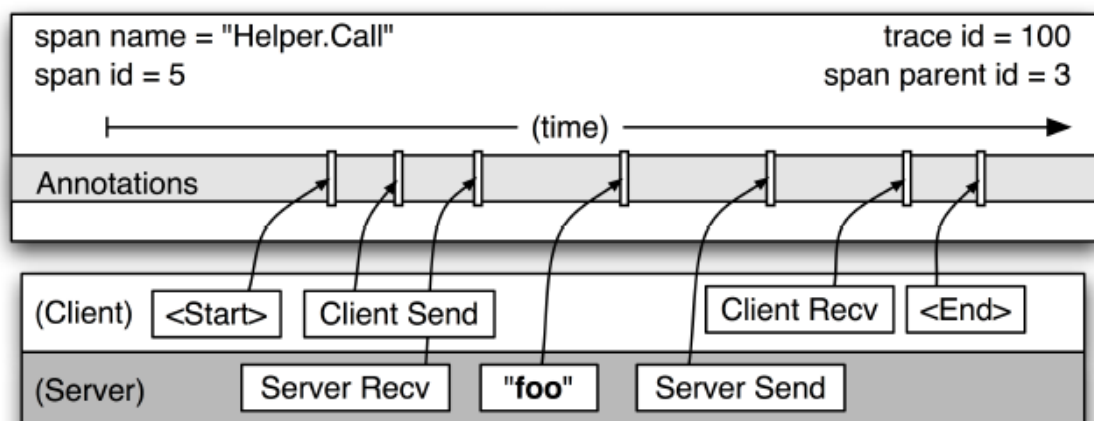
基本工作单位，一次单独的调用链可以称为一个 Span，Dapper 记录的是 Span 的名称，以及每个 Span 的 ID 和父 ID，以重建在一次追踪过程中不同 Span 之间的关系，图中一个矩形框就是一个 Span，前端从发出请求到收到回复就是一个 Span。

[



开始跟踪的初始跨度称为 **root span**。该跨度的 ID 的值等于跟踪 ID。

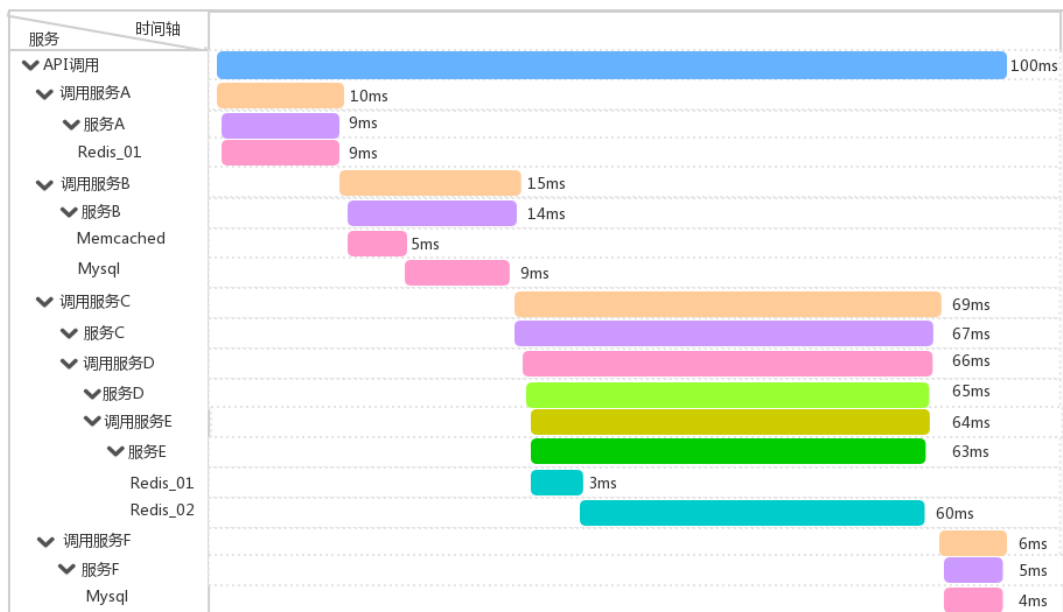
Dapper 记录了 span 名称，以及每个 span 的 ID 和父 span ID，以重建在一次追踪过程中不同 span 之间的关系。如果一个 span 没有父 ID 被称为 root span。所有 span 都挂在一个特定的 Trace 上，也共用一个 trace id。



5.2 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：



5.3 Annotation

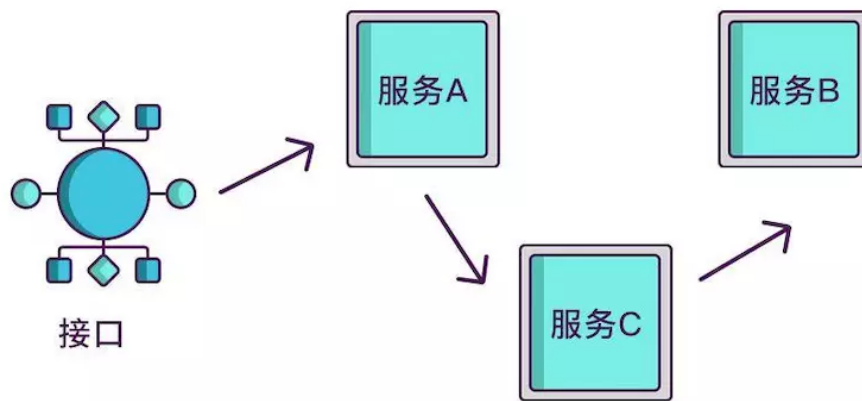
用来及时记录一个事件的存在，一些核心 annotations 用来定义一个请求的开始和结束。

- 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。

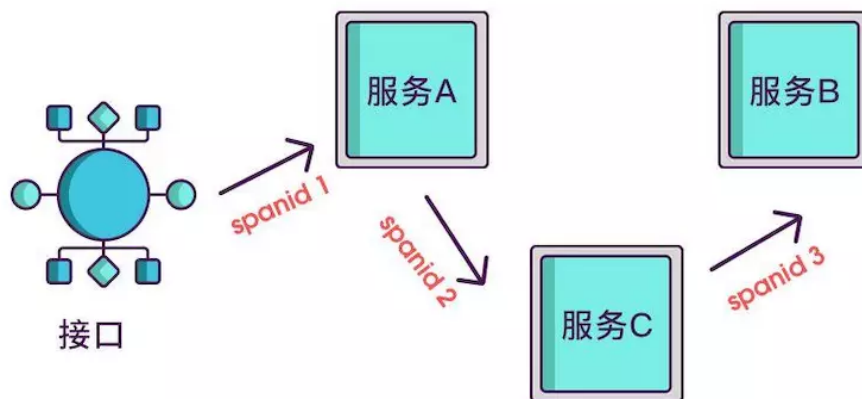
6. 实现原理

首先感谢张以诺制作的实现原理图。

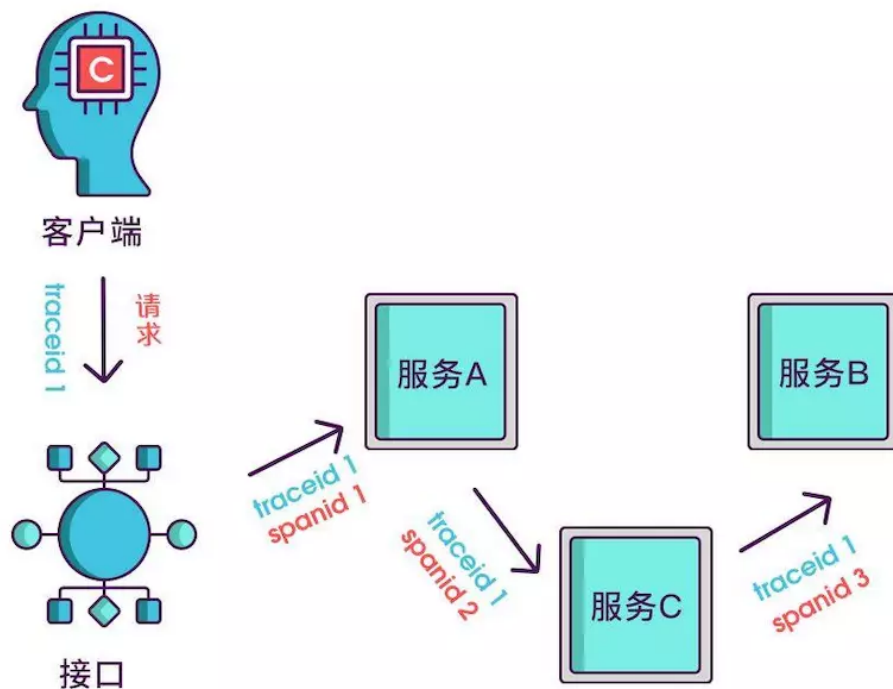
如果想知道一个接口在哪个环节出现了问题，就必须清楚该接口调用了哪些服务，以及调用的顺序，如果把这些服务串起来，看起来就像链条一样，我们称其为调用链。



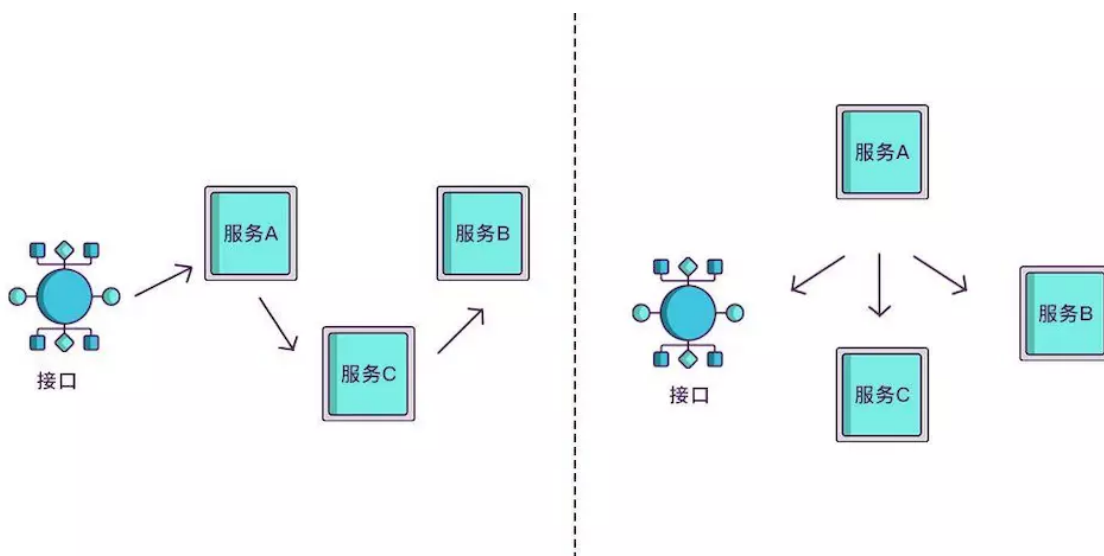
想要实现调用链，就要为每次调用做个标识，然后将服务按标识大小排列，可以更清晰地看出调用顺序，我们暂且将该标识命名为 spanid。



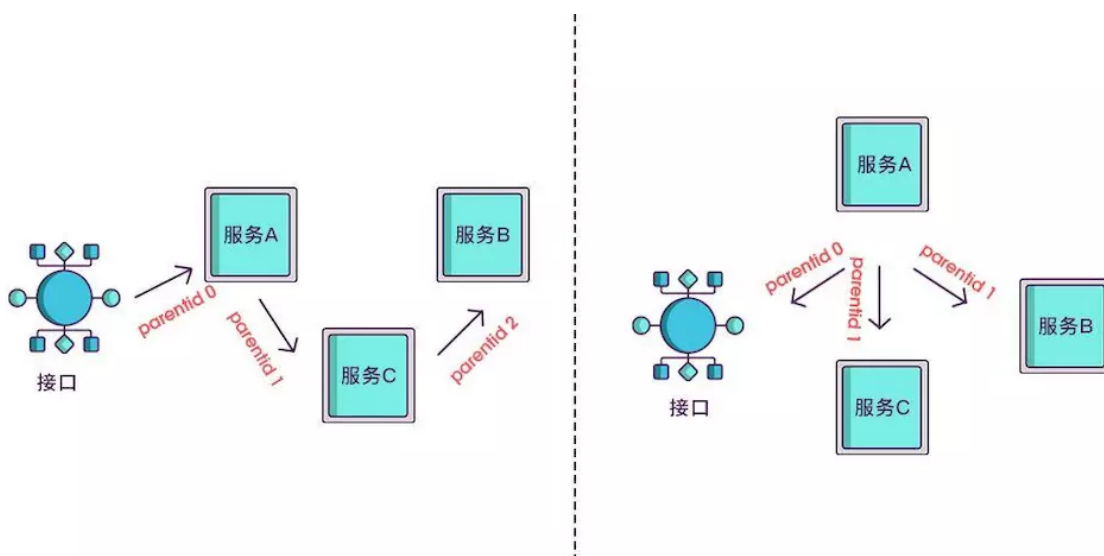
实际场景中，我们需要知道某次请求调用的情况，所以只有 spanid 还不够，得为每次请求做个唯一标识，这样才能根据标识查出本次请求调用的所有服务，而这个标识我们命名为 traceid。



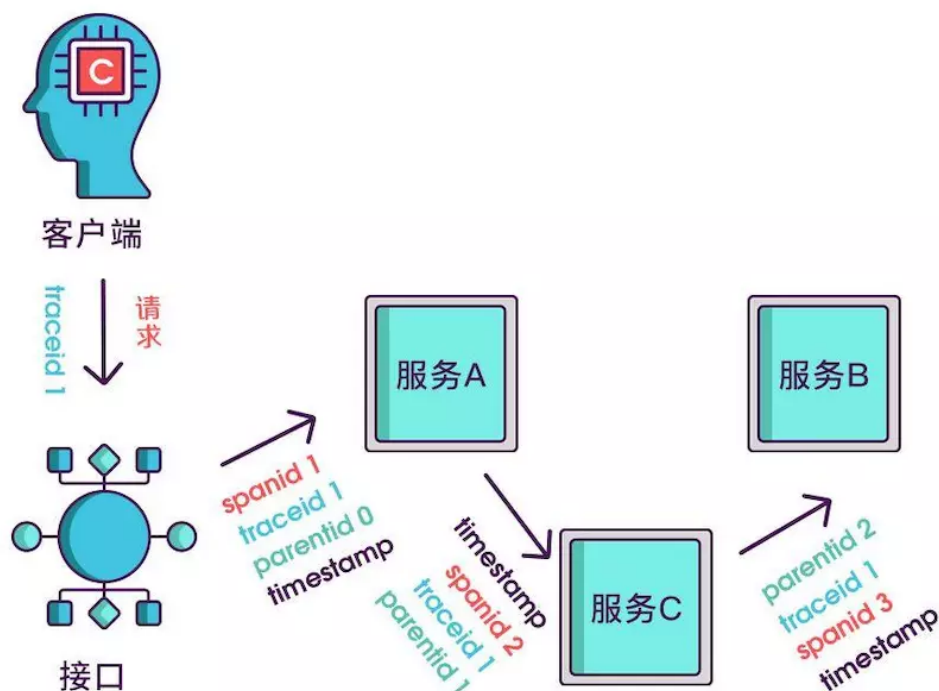
现在根据 spanid 可以轻易地知道被调用服务的先后顺序，但无法体现调用的层级关系，正如下图所示，多个服务可能是逐级调用的链条，也可能是同时被同一个服务调用。



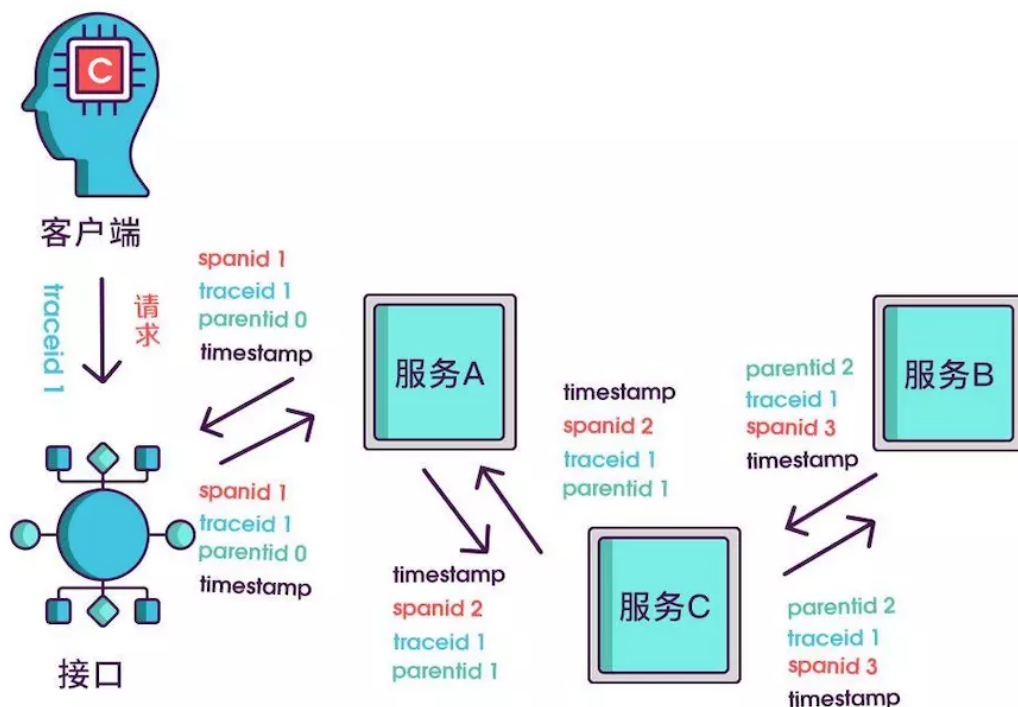
所以应该每次都记录下是谁调用的，我们用 parentid 作为这个标识的名字。



到现在，已经知道调用顺序和层级关系了，但是接口出现问题后，还是不能找到出问题的环节，如果某个服务有问题，那个被调用执行的服务一定耗时很长，要想计算出耗时，上述的三个标识还不够，还需要加上时间戳，时间戳可以更精细一点，精确到微秒级。



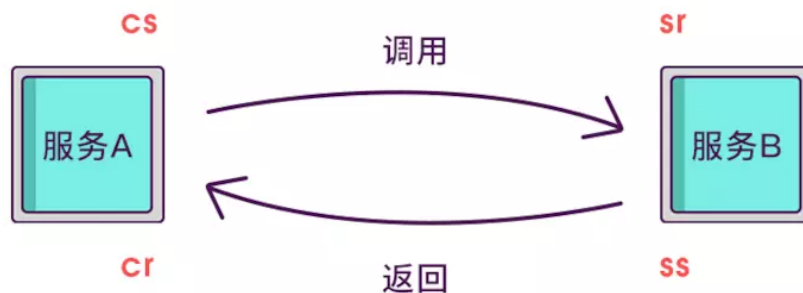
只记录发起调用时的时间戳还算不出耗时，要记录下服务返回时的时间戳，有始有终才能算出时间差，既然返回的也记了，就把上述的三个标识都记一下吧，不然区分不出是谁的时间戳。



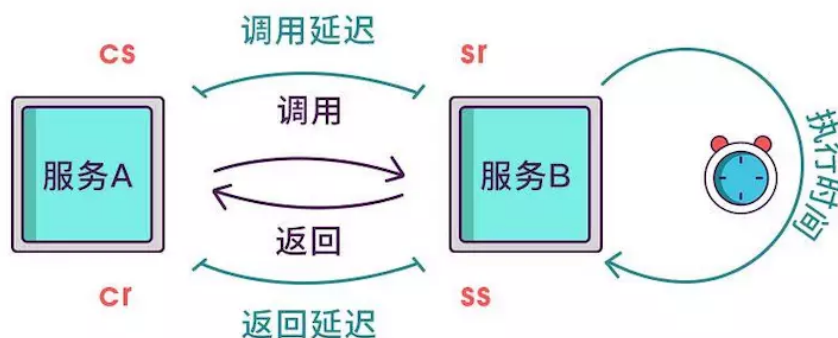
虽然能计算出从服务调用到服务返回的总耗时，但是这个时间包含了服务的执行时间和网络延迟，有时候我们需要区分出这两类时间以方便做针对性优化。那如何计算网络延迟呢？我们可以把调用和返回的过程分为以下四个事件。

- Client Sent 简称 cs，客户端发起调用请求到服务端。
- Server Received 简称 sr，指服务端接收到了客户端的调用请求。

- Server Sent 简称 ss，指服务端完成了处理，准备将信息返给客户端。
- Client Received 简称 cr，指客户端接收到了服务端的返回信息。



假如在这四个事件发生时记录下时间戳，就可以轻松计算出耗时，比如 $sr - cs$ 就是调用时的网络延迟， $ss - sr$ 就是服务执行时间， $cr - ss$ 就是服务响应的延迟， $cr - cs$ 就是整个服务调用执行的时间。



其实 span 内除了记录这几个参数之外，还可以记录一些其他信息，比如发起调用服务名称、被调服务名称、返回结果、IP、调用服务的名称等，最后，我们再把相同 parentid 的 span 信息合成一个大的 span 块，就完成了整个完整的调用链。

7. 环境准备

sleuth-demo 聚合工程。 SpringBoot 2.2.4.RELEASE 、 Spring Cloud Hoxton.SR1 。

- eureka-server01 : 注册中心
- eureka-server02 : 注册中心
- gateway-server : Spring Cloud Gateway 服务网关
- product-service : 商品服务，提供了根据主键查询商品接口
<http://localhost:7070/product/{id}> 根据多个主键查询商品接口
<http://localhost:7070/product/listByIds>

- `order-service`：订单服务，提供了根据主键查询订单接口 `http://localhost:9090/order/{id}` 且订单服务调用商品服务。

DS Replicas			
localhost			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
EUREKA-SERVER	n/a (2)	(2)	UP (2) - 192.168.31.103:8761, 192.168.31.103:8762
GATEWAY-SERVER	n/a (1)	(1)	UP (1) - 192.168.31.103:9000
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.31.103:9090
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.31.103:7070

8. 入门案例

8.1 添加依赖

在需要进行链路追踪的项目中（服务网关、商品服务、订单服务）添加 `spring-cloud-starter-sleuth` 依赖。

```
<!-- spring cloud sleuth 依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

8.2 记录日志

在需要链路追踪的项目中添加 `logback.xml` 日志文件，内容如下（logback 日志的输出级别需要是 DEBUG 级别）：

注意修改 `<property name="log.path" value="${catalina.base}/gateway-server/logs"/>` 中项目名称。

日志核心配置：`%d{yyyy-MM-dd HH:mm:ss.SSS} [%${applicationName},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-}] [%thread] %-5level %logger{50} - %msg%n`

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 日志级别从低到高分TRACE < DEBUG < INFO < WARN < ERROR < FATAL，如果设置为WARN，则低于WARN的信息都不会输出 -->
<!-- scan：当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true -->
<!-- scanPeriod：设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒。当scan为true时，此属性生效。默认的时间间隔为1分钟。 -->
```

```

<!-- debug：当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。 -->
<configuration scan="true" scanPeriod="10 seconds">
    <!-- 日志上下文名称 -->
    <contextName>my_logback</contextName>
    <!-- name的值是变量的名称，value的值是变量定义的值。通过定义的值会被插入到logger上下文中。定义变量后，可以使"${}"来使用变量。 -->
    <property name="log.path" value="${catalina.base}/gateway-server/logs" />
    <!-- 加载 Spring 配置文件信息 -->
    <springProperty scope="context" name="applicationName"
source="spring.application.name" defaultValue="localhost" />
    <!-- 日志输出格式 -->
    <property name="LOG_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS}
[${applicationName}],%X{X-B3-TraceId:-},%X{X-B3-SpanId:-}] [%thread] %-5level
%logger{50} - %msg%n" />

    <!--输出到控制台-->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <!--此日志appender是为开发使用，只配置最底级别，控制台输出的日志级别是大于或等于此级别的日志信息-->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
            <!-- 设置字符集 -->
            <charset>UTF-8</charset>
        </encoder>
    </appender>

    <!-- 输出到文件 -->
    <!-- 时间滚动输出 level为 DEBUG 日志 -->
    <appender name="DEBUG_FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 正在记录的日志文件的路径及文件名 -->
        <file>${log.path}/log_debug.log</file>
        <!--日志文件输出格式-->
        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
            <charset>UTF-8</charset> <!-- 设置字符集 -->
        </encoder>
        <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 日志归档 -->
            <fileNamePattern>${log.path}/debug/log-debug-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!--日志文件保留天数-->
            <maxHistory>15</maxHistory>
        </rollingPolicy>
        <!-- 此日志文件只记录debug级别的 -->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>DEBUG</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>

```

```

        </filter>
    </appender>

    <!-- 时间滚动输出 level为 INFO 日志 -->
    <appender name="INFO_FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 正在记录的日志文件的路径及文件名 -->
        <file>${log.path}/log_info.log</file>
        <!--日志文件输出格式-->
        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
            <charset>UTF-8</charset>
        </encoder>
        <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 每天日志归档路径以及格式 -->
            <fileNamePattern>${log.path}/info/log-info-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!--日志文件保留天数-->
            <maxHistory>15</maxHistory>
        </rollingPolicy>
        <!-- 此日志文件只记录info级别的 -->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>INFO</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>

    <!-- 时间滚动输出 level为 WARN 日志 -->
    <appender name="WARN_FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 正在记录的日志文件的路径及文件名 -->
        <file>${log.path}/log_warn.log</file>
        <!--日志文件输出格式-->
        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
            <charset>UTF-8</charset> <!-- 此处设置字符集 -->
        </encoder>
        <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${log.path}/warn/log-warn-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <!-- 每个日志文件最大100MB -->
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!--日志文件保留天数-->
            <maxHistory>15</maxHistory>
        </rollingPolicy>
        <!-- 此日志文件只记录warn级别的 -->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>WARN</level>

```

```

        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
</appender>

<!-- 时间滚动输出 level为 ERROR 日志 -->
<appender name="ERROR_FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
    <!-- 正在记录的日志文件的路径及文件名 -->
    <file>${log.path}/log_error.log</file>
    <!--日志文件输出格式-->
    <encoder>
        <pattern>${LOG_PATTERN}</pattern>
        <charset>UTF-8</charset> <!-- 此处设置字符集 -->
    </encoder>
    <!-- 日志记录器的滚动策略，按日期，按大小记录 -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${log.path}/error/log-error-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
        <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
            <maxFileSize>100MB</maxFileSize>
        </timeBasedFileNamingAndTriggeringPolicy>
        <!--日志文件保留天数-->
        <maxHistory>15</maxHistory>
        <!-- 日志量最大 10 GB -->
        <totalSizeCap>10GB</totalSizeCap>
    </rollingPolicy>
    <!-- 此日志文件只记录ERROR级别的 -->
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <level>ERROR</level>
        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
</appender>

<!-- 对于类路径以 com.example.logback 开头的Logger,输出级别设置为warn,并且只输出到控制台 -->
<!-- 这个logger没有指定appender,它会继承root节点中定义的那些appender -->
<!-- <logger name="com.example.logback" level="warn"/> -->

<!--通过 LoggerFactory.getLogger("myLog") 可以获取到这个logger-->
<!--由于这个logger自动继承了root的appender, root中已经有stdout的appender了,自己这边又引入了stdout的appender-->
<!--如果没有设置 additivity="false",就会导致一条日志在控制台输出两次的情况-->
<!--additivity表示要不要使用rootLogger配置的appender进行输出-->
<logger name="myLog" level="INFO" additivity="false">
    <appender-ref ref="CONSOLE"/>
</logger>

<!-- 日志输出级别及方式 -->
<root level="DEBUG">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="DEBUG_FILE"/>
    <appender-ref ref="INFO_FILE"/>
    <appender-ref ref="WARN_FILE"/>
    <appender-ref ref="ERROR_FILE"/>
</root>

```



```
</configuration>
```

8.3 访问接口

访问：<http://localhost:9000/order-service/order/1>，结果如下：

服务网关打印信息：

```
[gateway-server, 95aa725089b757f8, 95aa725089b757f8]
```

商品服务打印信息

```
[product-service, 95aa725089b757f8, e494e064842ce4e8]
```

订单服务打印信息

```
[order-service, 95aa725089b757f8, f4ee41a6dcf08717]
```

通过打印信息可以得知，整个链路的 `traceId` 为：95aa725089b757f8，`spanId` 为：e494e064842ce4e8 和 f4ee41a6dcf08717。

查看日志文件并不是一个很好的方法，当微服务越来越多日志文件也会越来越多，查询工作会变得越来越麻烦，Spring 官方推荐使用 Zipkin 进行链路跟踪。Zipkin 可以将日志聚合，并进行可视化展示和全文检索。

9. Zipkin链路追踪

9.1 什么是Zipkin

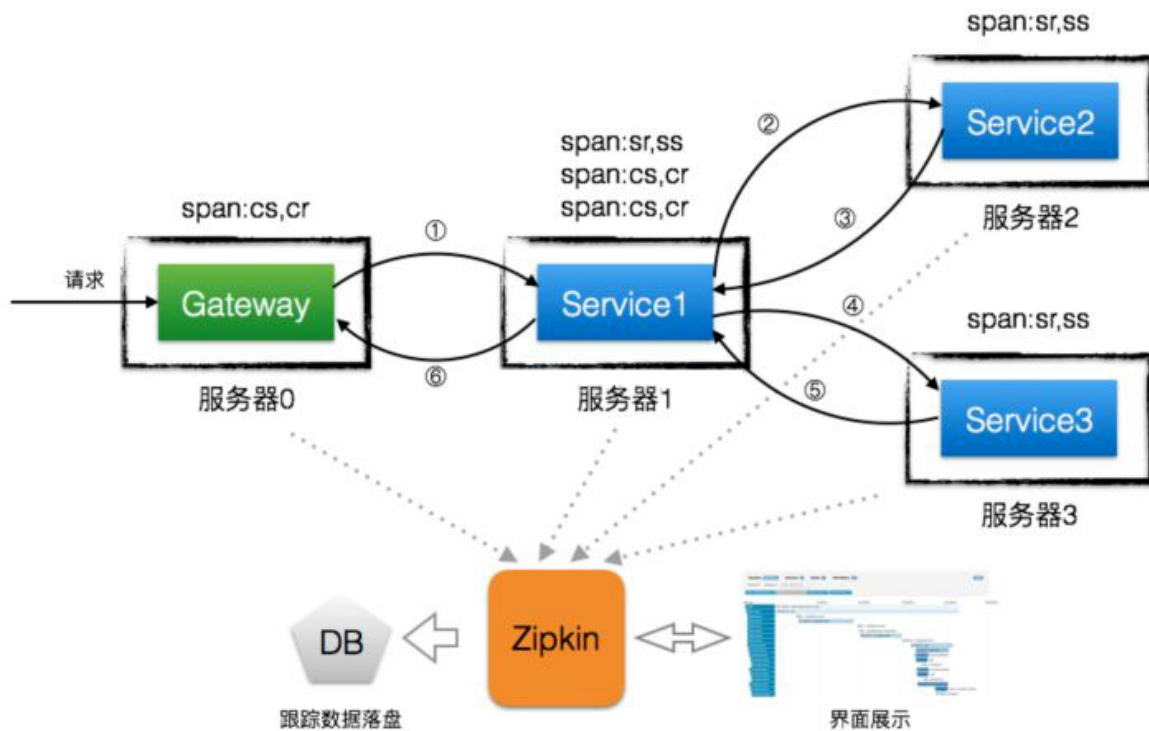


ZIPKIN

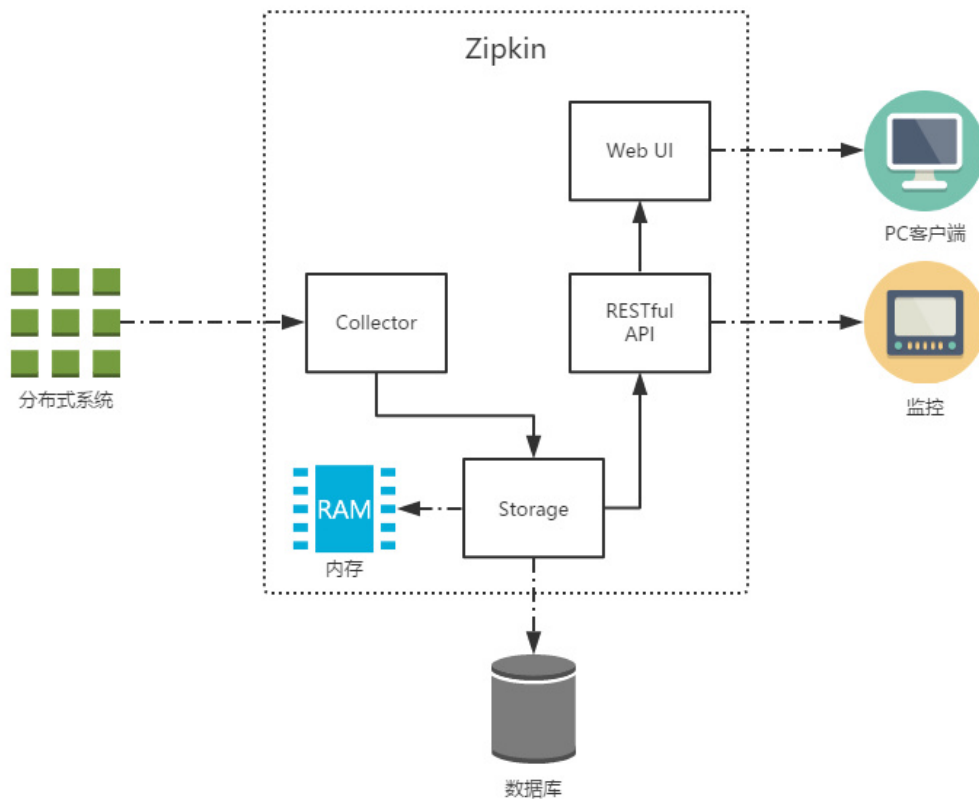
Zipkin 是 Twitter 公司开发贡献的一款开源的分布式实时数据追踪系统（Distributed Tracking System），基于 Google Dapper 的论文设计而来，其主要功能是聚集各个异构系统的实时监控数据。

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。



9.2 工作原理



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。

Zipkin 分为两端，一个是 Zipkin 服务端，一个是 Zipkin 客户端，客户端也就是微服务的应用，客户端会配置服务端的 URL 地址，一旦发生服务间的调用的时候，会被配置在微服务里面的 Sleuth 的监听器监听，并生成相应的 Trace 和 Span 信息发送给服务端。发送的方式有两种，一种是消息总线的方式如 RabbitMQ 发送，还有一种是 HTTP 报文的方式发送。

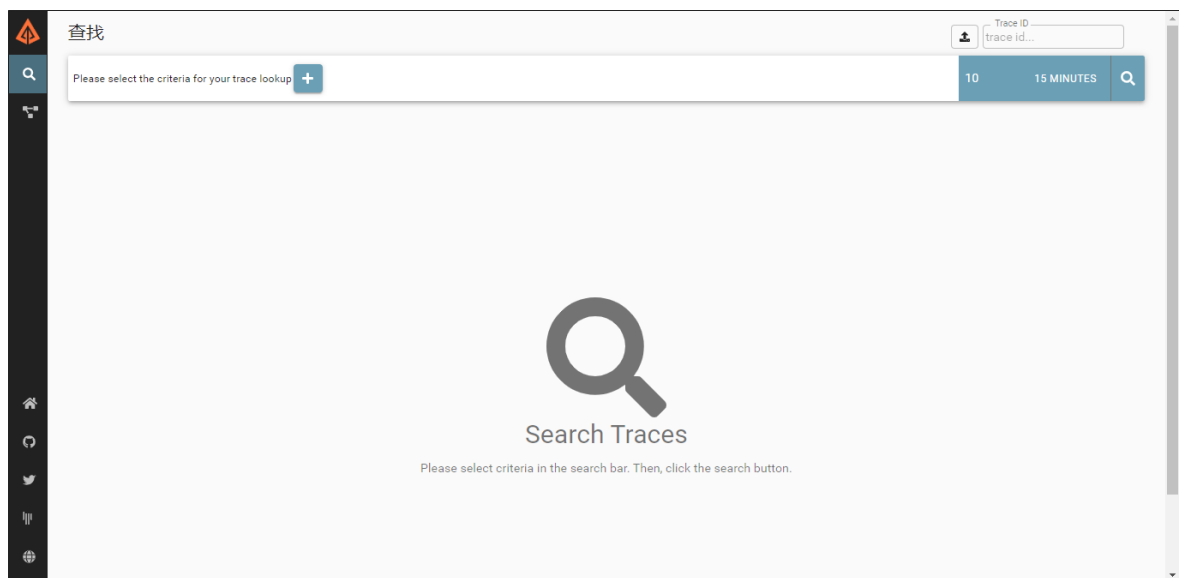
9.3 服务端部署

服务端是一个独立的可执行的 jar 包，官方下载地址：https://search.maven.org/remote_content?g=io.zipkin&a=zipkin-server&v=LATEST&c=exec，使用 `java -jar zipkin.jar` 命令启动，端口默认为 9411。我们下载的 jar 包为：zipkin-server-2.20.1-exec.jar，启动命令如下：


```
java -jar zipkin-server-2.20.1-exec.jar
```

访问：<http://localhost:9411/> 结果如下：

目前最新版界面。



之前旧版本界面。

 Zipkin 查找 已保存 依赖

[Try Lens UI](#)

[搜索](#)

服务名

Span名称

远程服务名

时间

all

Span Name

Remote Service Name

15 分钟

根据Annotation查询

持续时间 (µs) >=

数量

排序

For example: http.path=/foo/bar/ and cluster=foo and cache.miss

Ex: 100ms or 5s

10

耗时降序

[查找](#) [?](#)

请选择查找的条件。

9.4 客户端部署

点击链接观看：[Zipkin 客户端部署视频](#)（获取更多请关注公众号「哈喽沃德先生」）

9.4.1 添加依赖

在需要进行链路追踪的项目中（服务网关、商品服务、订单服务）添加 `spring-cloud-starter-zipkin` 依赖。

```
<!-- spring cloud zipkin 依赖 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

9.4.2 配置文件

在需要进行链路追踪的项目中（服务网关、商品服务、订单服务）配置 Zipkin 服务端地址及数据传输方式。默认即如下配置。

```
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%)
```

9.4.3 访问接口

访问：<http://localhost:9000/order-service/order/1> 结果如下：

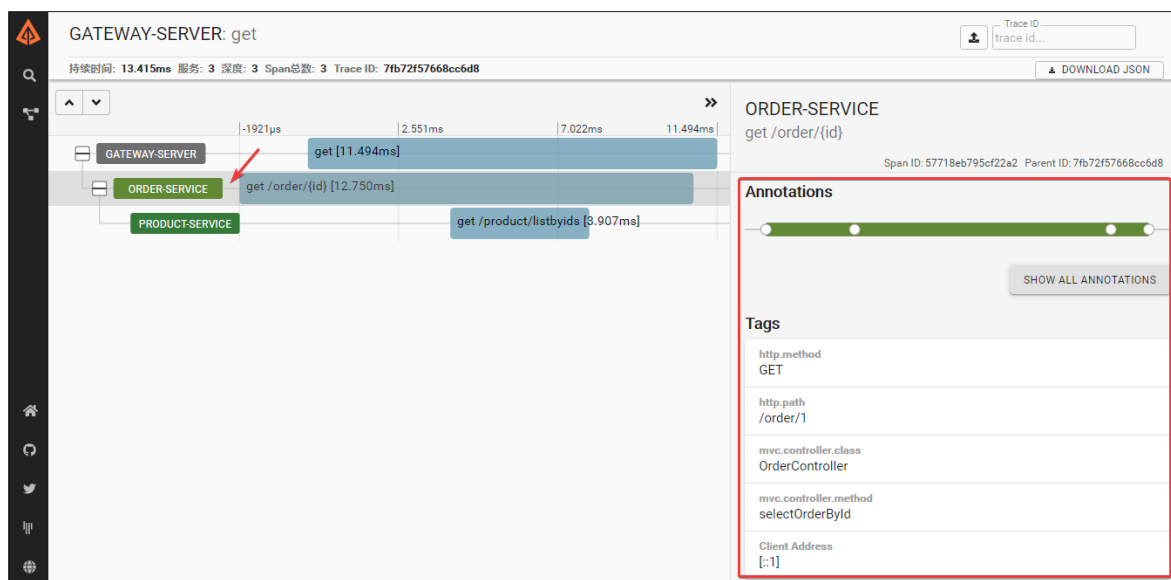


新版操作如下：

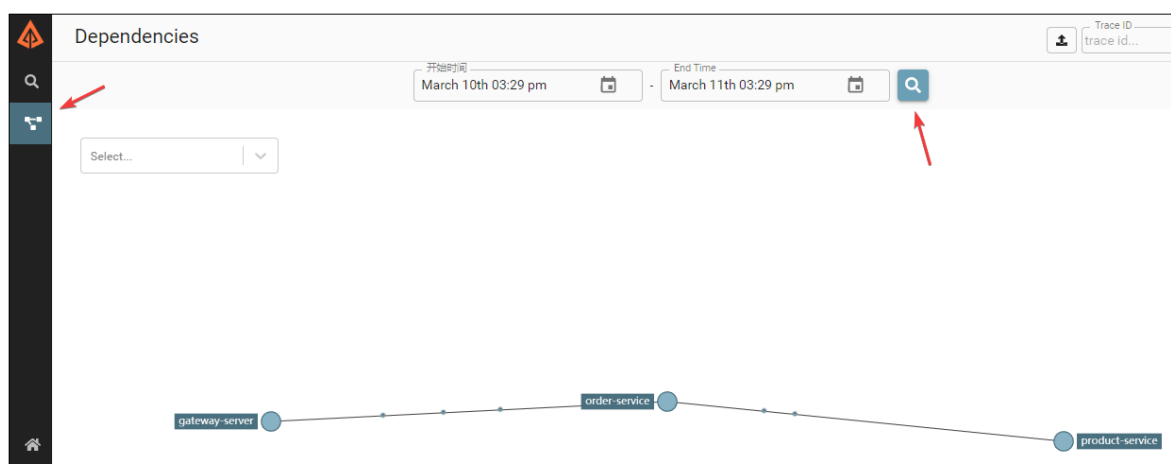
访问：<http://localhost:9411/> 根据时间过滤点击 搜索 结果如下：



点击对应的追踪信息可查看请求链路详细。

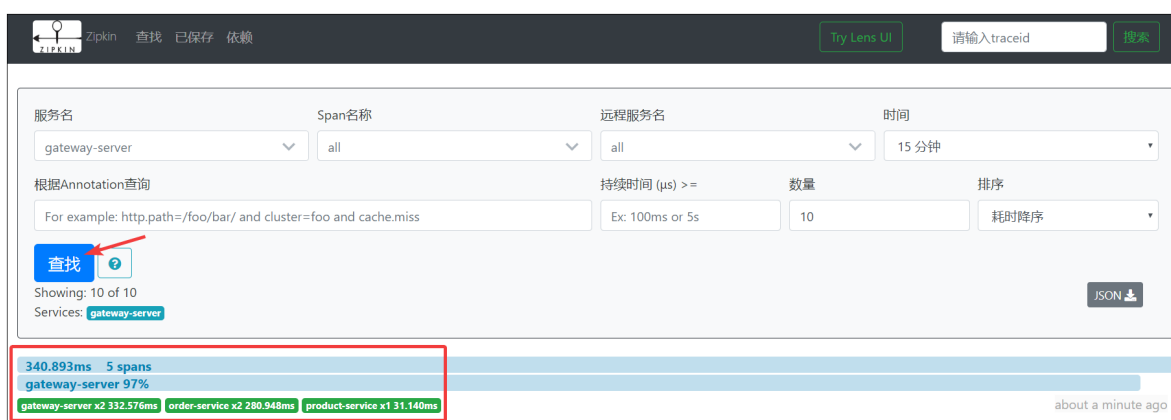


通过依赖可以查看链路中服务的依赖关系。

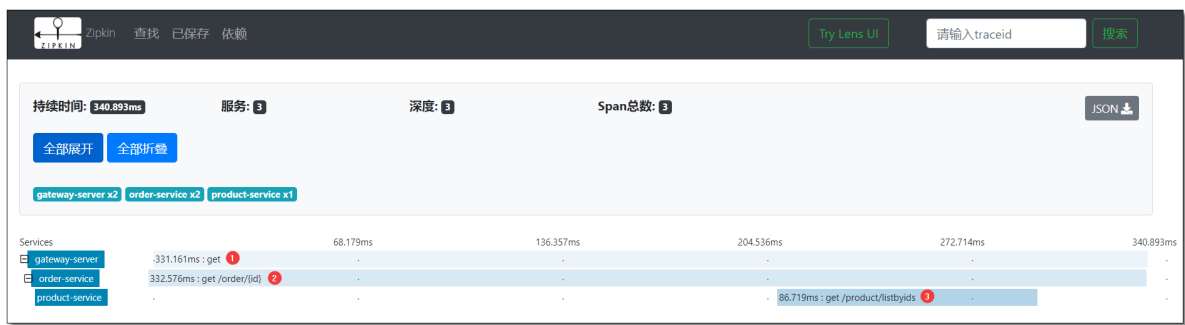


旧版操作如下：

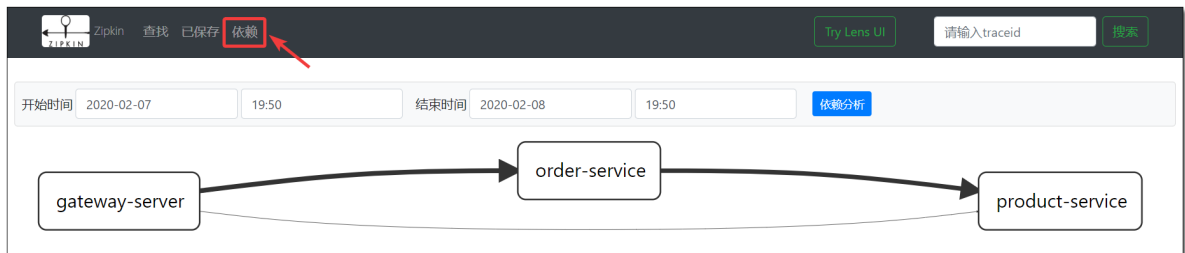
访问：<http://localhost:9411/> 点击 查找 结果如下：



点击对应的追踪信息可查看请求链路详细。



通过依赖可以查看链路中服务的依赖关系。



Zipkin Server 默认存储追踪数据至内存中，这种方式并不适合生产环境，一旦 Server 关闭重启或者服务崩溃，就会导致历史数据消失。Zipkin 支持修改存储策略使用其他存储组件，支持 MySQL，Elastic search 等。