分布式链路追踪

分布式链路追踪（**Distributed Tracing**）其实是一个用来 **跟踪请求在分布式系统中流转路径** 的技术。

作用

你可以想象一下：
👉 在一个单体应用里，**请求只会从入口到数据库，路径很短，很容易知道哪里慢了。**
👉 但是在一个 **分布式系统**（例如微服务架构）里，一个用户请求可能会经过：

- **API 网关**
- **服务 A**（用户服务）
- **服务 B**（订单服务）
- **服务 C**（支付服务）
- **消息队列**
- **数据库**

链路复杂，出错或变慢时，很难知道是 **哪个环节** 导致的。

前端视角解释

为什么需要链路追踪?

在前端，**你发一个请求到后端，其实只是整个业务调用链的一部分。**

你点了一下 **下单按钮 → 前端发请求 → 后端服务 A → 调用服务 B → 调用服务 C → 再写数据库 → 返回结果。**

如果系统变得很大（有几十个甚至上百个微服务），那一个请求会“跳来跳去”。
这时候：

- **如果出错了，你很难知道到底是哪个环节挂了。**
- **如果慢了，你也很难知道到底是卡在数据库、还是某个服务。**

这就是为什么需要 **链路追踪** ——它帮你像**“导航仪”一样，把整个请求的调用路径都记录下来。**

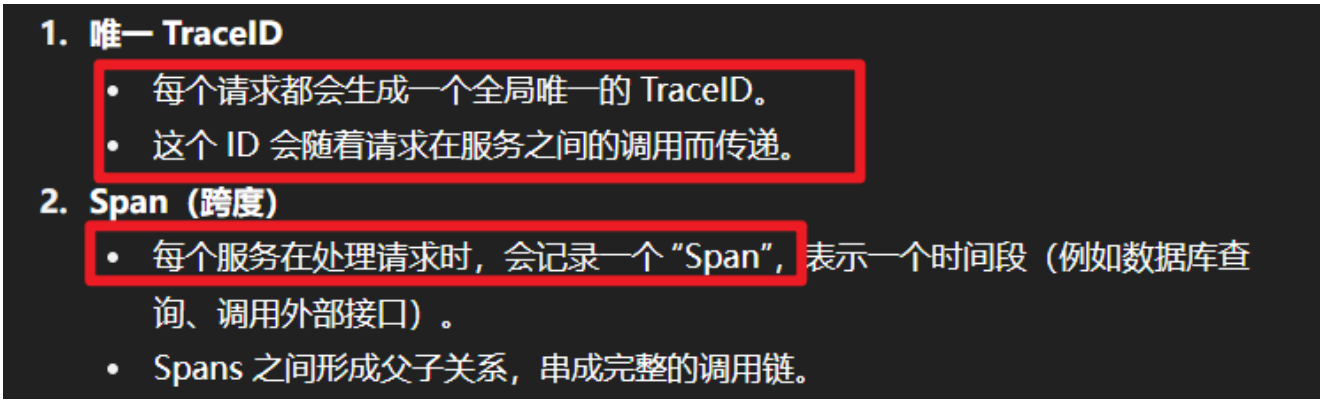
实现

分布式链路追踪是什么?

它是一套方法论 + 技术，用来 **跟踪一次请求在分布式系统中的完整调用链路。**

核心思想：

- 给**每一次请求一个 唯一 ID（TraceId）。**
- 请求**从前端到后端，再到下一个服务，都把这个 ID 传下去。**
- 同时在每个调用环节里，还会生成 **SpanId**（子调用 ID），**标记这个步骤属于整条链路的哪一部分。**
- 最终，整个调用链会被记录下来（通常存到日志系统或者链路追踪平台），方便你查询和分析。



每经过一个服务，就会生成一个spanId，代表一个请求经过一个服务

这些SpanID之间会形成父子关系，串成完整的调用链。

最终整个调用链路会被完整的记录下来

链路可视化

- 最终，整个调用链会被记录下来（例如保存在 Elasticsearch），
- 你可以在工具里（如 Jaeger、Zipkin、SkyWalking）直观地看到：

scss 复制代码

```
用户请求
  └── API 网关 (10ms)
        └── 用户服务 (50ms)
              ├── 调订单服务 (120ms)
              └── 查数据库 (30ms)
```

SpanID的传递机制

那这么多 spanId 怎么传递?

靠的是 **请求头里的链路追踪信息。**

遵循 W3C Trace Context标准

```
traceparent: 00-<traceId>-<spanId>-01
```

`<traceId>`：**唯一的链路 ID（从前端传过来，或者后端生成）。**

`<spanId>`：**当前服务的 spanId。**

每次服务接到请求时：

1. **读出 traceId（保持不变）。**
2. **生成新的 spanId。**
3. **把 traceId + 新 spanId 继续往下传。**

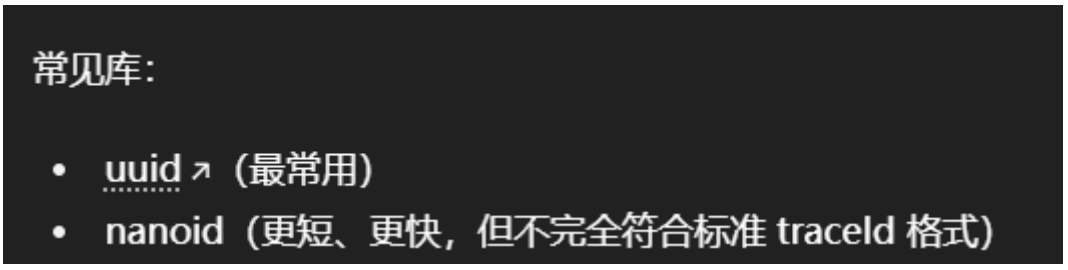
举例

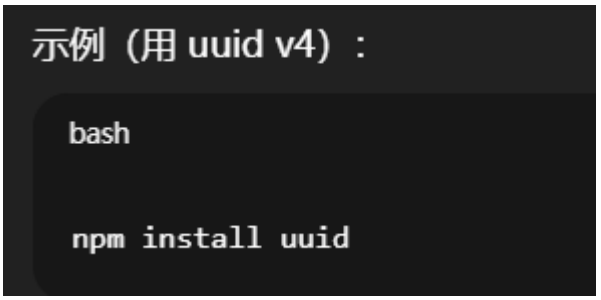
如下：

- 一个请求 **只有一个 traceId。**
- 会有**多个 spanId（每个服务 / 操作生成一个）。**
- 前端 **不用管 spanId**，只负责生成/传递 traceId。
- spanId 的生成和维护交给后端的追踪系统。

前端生成TraceID

在前端，最常见的做法是用 **UUID 库** 来生成。






```bash
npm install uuid
```

```ts
import { v4 as uuidv4 } from 'uuid';

// 生成一个 UUID
const traceId = uuidv4().replace(/-/g, '');
console.log(traceId); // 比如：550e8400e29b41d4a716446655440000
```

👉 注意：标准 **TraceId 要 16 字节（128bit），通常用 32 位十六进制字符串表示。**
所以我们一般会把 uuid 里的 - 去掉，得到一个 32 位的 hex 字符串。

TraceId 和 SpanId 的长度和格式

按照 **W3C Trace Context 规范**

TraceId

- 长度：16 字节 = 128 bit
- 格式：32 个十六进制字符（小写）
- 例子：`4bf92f3577b34da6a3ce929d0e0e4736`

SpanId

- 长度：8 字节 = 64 bit
- 格式：16 个十六进制字符（小写）
- 例子：`00f067aa0ba902b7`

版本号

- 2 个十六进制字符，当前一般是 `00`。

trace-flags

- 2 个十六进制字符，表示采样信息，`01` 表示开启采样，`00` 表示不采样。

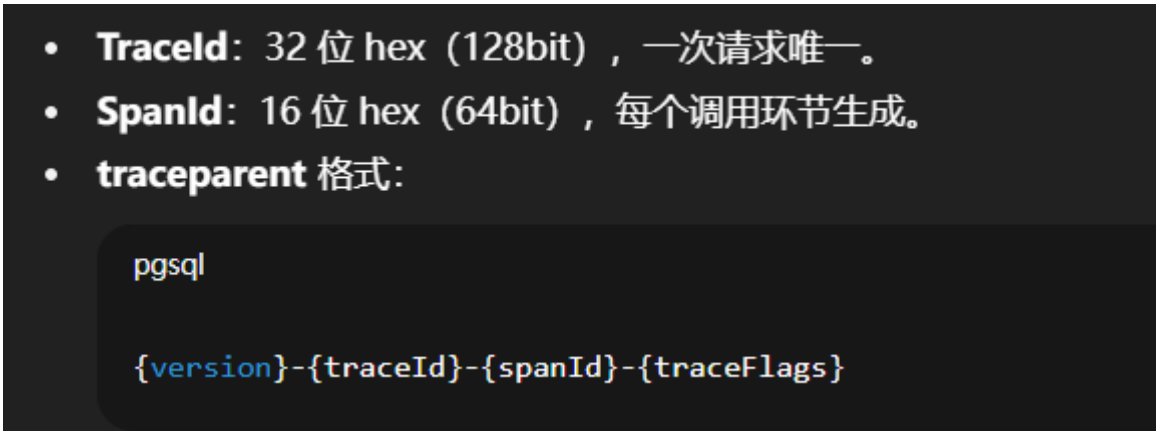
traceparent 的格式

W3C 定义了 `traceparent` 请求头，格式如下：

```
traceparent: {version}-{trace-id}-{span-id}-{trace-flags}
```

- `version`：`00`
- `trace-id`：32 个十六进制字符（TraceId）
- `span-id`：16 个十六进制字符（SpanId）
- `trace-flags`：2 个十六进制字符（一般 `01` 表示采样，`00` 表示不采样）

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
```



前端把traceid放入请求

前端职责：生成 TraceId，放到 header（`X-Trace-Id` 或 `traceparent`），传给后端。

SpanId： 一般由后端生成，前端不需要管。

方式 1：自定义 header（最常见，简单）

```
import axios from "axios";
import { v4 as uuidv4 } from "uuid";

const traceId = uuidv4().replace(/-/g, ''); // 32位hex

axios.get("/api/order", {
  headers: {
    "X-Trace-Id": traceId
  }
});
```

方式 2：用标准的 traceparent（更规范）

```
import axios from "axios";
import { v4 as uuidv4 } from "uuid";

// 生成 TraceId
const traceId = uuidv4().replace(/-/g, ''); // 32 位
// 前端不用生成 spanId，先给一个假的 spanId，全靠后端更新
const spanId = "0000000000000000";

const traceparent = `00-${traceId}-${spanId}-01`;

axios.get("/api/order", {
  headers: {
    "traceparent": traceparent
  }
});
```

👉 这样完全符合 W3C 规范，后端的 tracing 系统（Jaeger、SkyWalking、Zipkin 等）能直接解析。