emetry-collector-contrib` 的地址。

> 其余的一些配置在后面会讲到。

```
curl http://127.0.0.1:9191/request\?name\=1232
```

然后我们触发一下 Java 客户端的入口，就可以在 JaegerUI 中查询到刚才的链路了。
`http://localhost:16686/search`

这样整个 `trace` 链路就串起来了。

Java 应用

下面来看看具体的应用代码里是如何编写的。

> Java 是基于 springboot 编写的，具体 springboot 的使用就不再赘述了。

因为我们应用是使用 gRPC 通信的，所以需要提供一个 `helloworld.proto` 的 pb 文件：

```
syntax = ”proto3”;

option go_package =
”google.golang.org/grpc/examples/helloworld/helloworld”;
option java_multiple_files = true;
option java_package = ”io.grpc.examples.helloworld”;
option java_outer_classname = ”HelloWorldProto”;

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user’s name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}

```

这个文件也没啥好说的，就定义了一个简单的 `SayHello` 接口。

```
<dependency>
  <groupId>net.devh</groupId>
  <artifactId>grpc-spring-boot-starter</artifactId>
  <version>3.1.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-stub</artifactId>
  <version>${grpc.version}</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-protobuf</artifactId>
```

```
  <version>${grpc.version}</version>
</dependency>

```

在 Java 中使用了 `grpc-spring-boot-starter` 这个库来处理 gRPC 的客户端和服务端请求。

```

```
grpc:
  server:
    port: 9192
  client:
    greeter:
      address: 'static://127.0.0.1:50051'
      enableKeepAlive: true
      keepAliveWithoutCalls: true
      negotiationType: plaintext

```

然后我们定义了一个接口用于接收请求触发 `gRPC` 的调用：

```

```
    @RequestMapping("/request")
    public String request(@RequestParam String name) {
        log.info("request: {}", request);
        HelloReply abc =
greeterStub.sayHello(io.grpc.examples.helloworld.HelloRequest.newBuild
er().setName(request.getName()).build());
        return abc.getMessage();
    }

```

Java 应用的实现非常简单，和我们日常日常开发没有任何区别；唯一的区别就是在启动时需要加入一个 `javaagent`以及一些启动参数。

```

```
java -javaagent:opentelemetry-javaagent-2.4.0-SNAPSHOT.jar \
-Dotel.traces.exporter=otlp \
-Dotel.metrics.exporter=otlp \
```
```

```
-Dotel.logs.exporter=none \
-Dotel.service.name=demo \
-Dotel.exporter.otlp.protocol=grpc \
-Dotel.propagators=tracecontext,baggage \
-Dotel.exporter.otlp.endpoint=http://127.0.0.1:5317 \
     -jar target/demo-0.0.1-SNAPSHOT.jar
```

下面来仔细看看这些参数

| 名称 | 作 race.SpanFromContext(ctx)
span.SetAttributes(attribute.String("request.name", in.Name))

```

我们使用 `span := trace.SpanFromContext(ctx)` 获取到当前的 span，然后调用 `SetAttributes` 就可以添加自定义的数据了。

> 对应的 Java 也有类似的函数。

除了新增 `attribute` 之外还可以新增 Event，Link 等数据，使用方式也是类似的。

```
// AddEvent adds an event with the provided name and options.
AddEvent(name string, options ...EventOption)

// AddLink adds a link.
// Adding links at span creation using WithLinks is preferred to calling AddLink
// later, for contexts that are available during span creation, because head
// sampling decisions can only consider information present during span creation.
AddLink(link Link)
```

自定义新增 span
```

==========

同理我们可能不局限于为某个 span 新增 attribute，也有可能想要新增一个新的 span 来记录关键的调用信息。

> 默认情况下只有 OpenTelemetry 实现过的组件的核心函数才会有 span，自己代码里的函数调用是不会创建span 的。

```
func (s *server) span(ctx context.Context) {
    ctx, span := tracer.Start(ctx, "hello-span")
    defer span.End()
    // do some work
    log.Printf("create span")
}
```

在 Go 中只需要手动 Start 一个 span 即可。

对应到 `Java` 稍微简单一些，只需要为函数添加一个注解即可。

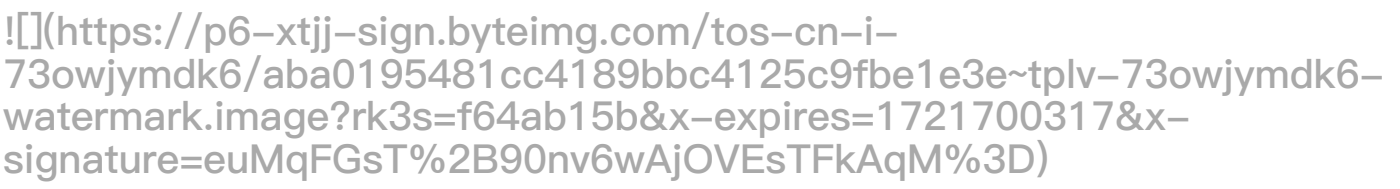
```
@WithSpan("span")
public void span(@SpanAttribute("request.name") String name) {
    TimeUnit.SECONDS.sleep(1);
    log.info("span:{}", name);
}
```

只不过得单独引入一个依赖：

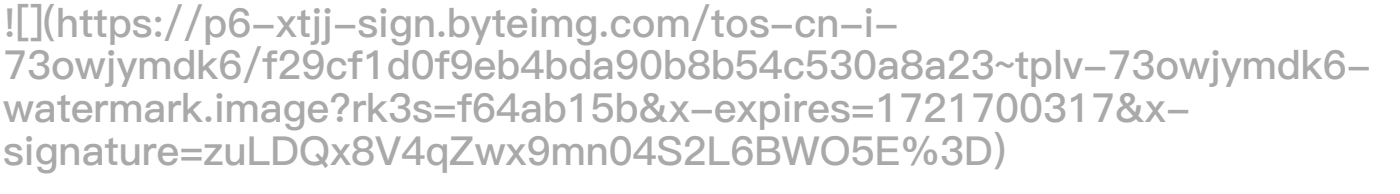
```
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-api</artifactId>
</dependency>
```

```
<dependency>
  <groupId>io.opentelemetry.instrumentation</groupId>
  <artifactId>opentelemetry-instrumentation-annotations</artifactId>
  <version>2.3.0</version>
</dependency>
```

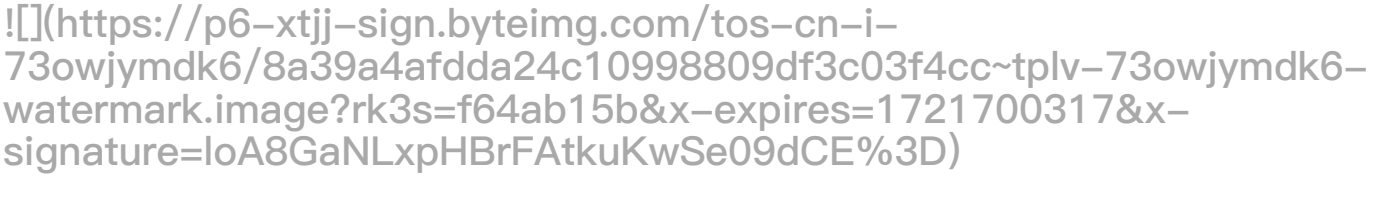
最终我们在 Jaeger UI 上看到的效果如下：



总结
==



最后总结一下，OpenTelemetry 支持许多流行的语言，主要分为两类：是否支持自动埋点。



> 这里 Go 也可以零代码埋点，是使用了 eBPF，本文暂不做介绍。

对于支持自动埋点的语言就很简单，只需要配置下 agent 即可；而原生的 Go 语言不支持自动埋点就得手动使用 OpenTelemetry 提供的 SDK 处理一些关键步骤；总体来说也不算复杂。

下一期会重点讲解如何使用 Metrics。

感兴趣的朋友可以在这里查看 Go 相关的源码：

* [github.com/crossoverJi…](http://cxyroad.com/”https://github.com/crossoverJie/k8s-combat”)

参考链接：

* [opentelemetry.io/docs/langua…](http://cxyroad.com/”https://opentelemetry.io/docs/languages/java/configuration/”)
* [github.com/open-teleme…](http://cxyroad.com/”https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/docs/supported-libraries.md”)
* [crossoverjie.top/2024/06/06/…](http://cxyroad.com/”https://crossoverjie.top/2024/06/06/ob/OpenTelemetry-trace-concept/”)

原文链接: https://juejin.cn/post/7391744486979076146