# MicroProfile によるマイクロサービスの分散トレース（Part2）：MicroProfile OpenTracing と Jaeger による可視化



2019 年 10 月 25 日
数村 憲治

## はじめに

MicroProfile OpenTracing と可視化ツール Jaeger を使って分散トレースをする方法について、Part1 に引き続き紹介します。Part1 では、JAX-RS のアプリケーションに対して、OpenTracing の API を何も使わずにトレースする方法を紹介しました。Part2 では、OpenTracing API を使い、より細かなトレース情報を出力する方法を紹介します。

Part2 でも Part1 同様に MicroProfile の実装の一つである「Launcher」および Jaeger を使用し説明します。Launcher の準備や起動方法などは、Part1 の記事を参照してください。なお、Part2 で例として使用するプログラムは、前回使用した 3 つの ServiceA、ServiceB、ServiceC をベースに修正していきます。
なお、ここで紹介するプログラムの完全なソースコードは、以下で参照できます。

- https://github.com/fujitsu/app_blog/tree/master/201010/mptracing2/（GitHub, Inc.）

## メソッド呼出しのトレース

JAX-RS を使ったリモート呼出しは、アプリケーションにトレース用の特別コードを記述しなくてもトレースすることができましたが、逆に、プロセス内でのメソッド呼出しは、デフォルトではトレースできません。
以下は、Part1 で使用した ServiceB.java の一部ですが、accept()メソッドから呼び出している ServiceClient.call()はトレースされず、accept()メソッドから直接 HTTP でリモート呼出しをしているように見えました。

```
  @GET
  public String accept(@QueryParam("option") String option) {
    return client.call(option);
  }

class ServiceClient {
  public String call(String option) {
    String result = ClientBuilder
      .newClient()
      .target("http://localhost:6060/service?option=" + option) // call ServiceC
```

このような JAX-RS 以外のメソッド呼出しをトレースするには、トレースしたいメソッドに、@Traced を付加します。

```
@RequestScoped
class ServiceClient {
  @Traced
  public String call(String option) {
    String result = ClientBuilder
      .newClient()
      .target("http://localhost:6060/service?option=" + option) // call ServiceC
```

 

このように修正したプログラムを実行すると、以下のようなトレースが得られ、ServiceB.accept()から ServiceClient.call()が呼ばれていることが分かります。

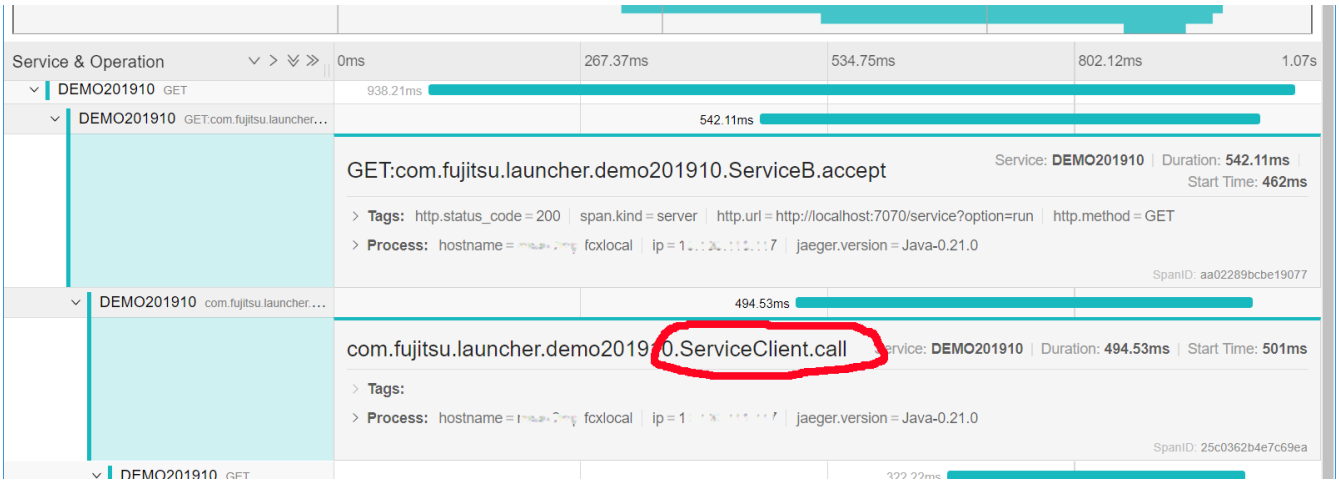


図 1

## メソッド内の一部をトレースする

トレースをする際に、メソッド全体ではなく一部のコードに費やす時間を測定したいとか、if の分岐の状況を確認したい、ということがあります。そういう時には、OpenTracing の API を使用することで実現することができます。
Part1 で使用した、ServiceC クラスの accept メソッドの if 内だけトレースをしてみます。

```
public class ServiceC {
  @Inject
  Tracer tracer;

  @GET
  public String accept(@QueryParam("option") String option) {
    if (option.equals("sleep")) {
      Span span = tracer.buildSpan("PORTION").start();
      try {
          Thread.sleep(500);   // intentionally delay response
      } catch (Exception e) {}
      span.log("only portion");
      span.finish();
    }
    return "Service accepted : " + option + "¥n";
  }
```

io.opentracing.Tracer クラスと io.opentracing.Span クラスを使います。Tracer.buildSpan()で、トレース情報を作成し、start()によりトレースの開始を指示、そして、Span.finish()により、トレースの終了を指示しています。また、Span.log()により、トレースに付加情報をつけることができます。
このように修正したプログラムを実行すると、以下のようなトレースが得られます。

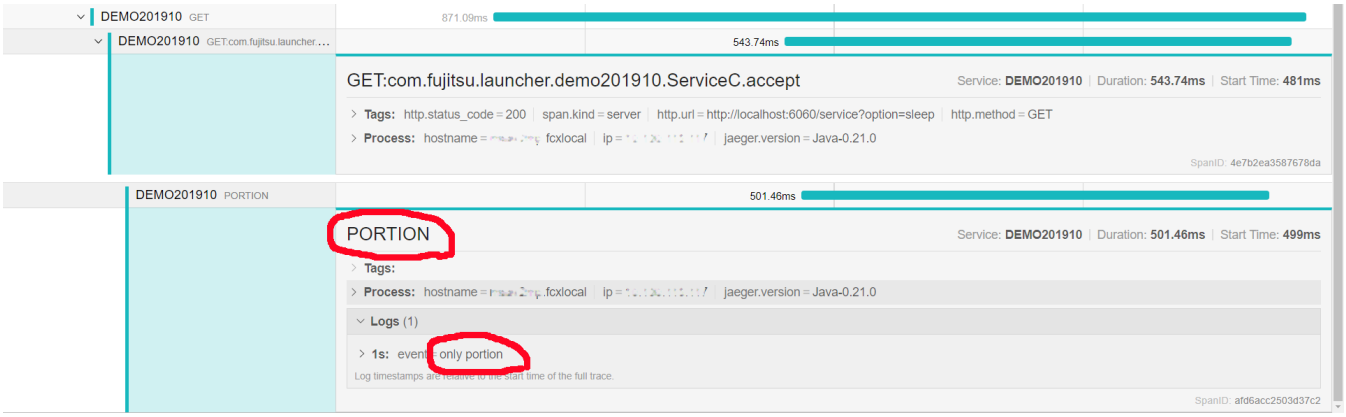


図 2

図中の PORTION というのは、Tracer.buildSpan()に指定した文字列で、トレースに名前を付けることができます。Span.log()に指定した文字列は、Jaeger では、event として表示されています。

## JAX-RS の呼出しでログを追加する

前項では、新たに作ったトレースの中でログを入れていましたが、デフォルトの JAX-RS の呼出しトレースの中にログを入れることもできます。
先ほど同様、Tracer クラスを使用します。Tracer.activeSpan()で、現在の Span、すなわち、デフォルトの JAX-RS 呼出しの Span を取り出すことができます。Part1 で使用した、ServiceA クラスの accept メソッド内に、ログを入れてみます。

```
public class ServiceA {
  @Inject
  Tracer tracer;

  @GET
  public String accept(@QueryParam("option") String option) {
    String result = ClientBuilder
      .newClient()
      .target("http://localhost:7070/service?option=" + option) // call ServiceB
      .request()
      .get(String.class);
    tracer.activeSpan().log("serviceA returns " + result);
    return result;
  }
```

このように修正したプログラムを実行すると、以下のようなトレースが得られます。

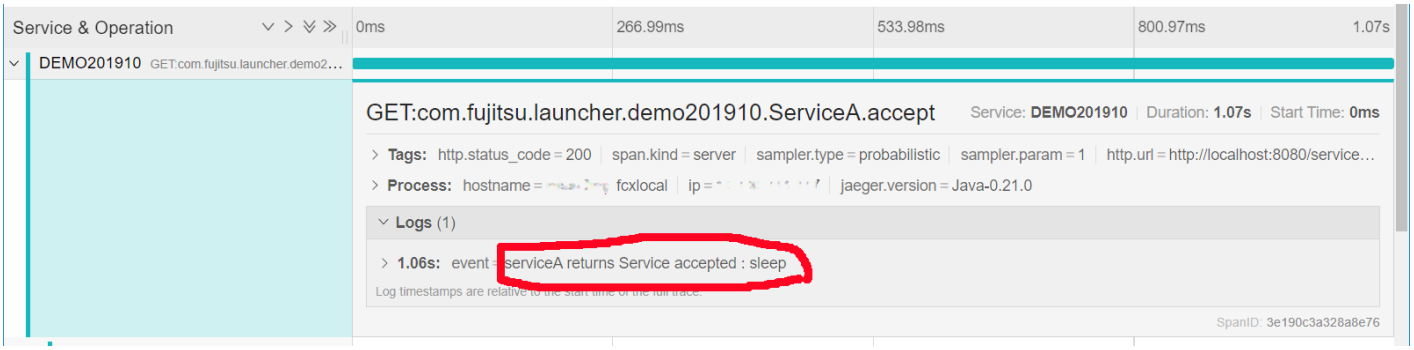


図 3

ServiceA.accept()のトレース中の event に挿入したログが表示されていることが分かります。

 

## まとめ

モノリシックアプリケーションでのトレースは、プロセス内のメソッドトレースが主でしたが、マイクロサービスでのトレースは分散トレースが重要になり、従来のツールや技術とは違うものを使用しなければなりません。Part1、Part2 の二回にわたり、MicroProfile OpenTracing と Jaeger を使った手法を紹介しましたが、マイクロサービスでの分散トレースの参考になれば幸いです。