MicroProfile によるマイクロサービスの分散トレース（Part1）：MicroProfile OpenTracing と Jaeger による可視化

2019 年 8 月 2 日 初版
2019 年 10 月 25 日 更新
数村 憲治

はじめに

マイクロサービスアーキテクチャー（MSA）は、たくさんのサービスが REST などによりコミュニケーションを行うアーキテクチャーです。このようなアーキテクチャーでは、従来のモノリシックシステムで使われていたメソッドトレースのような技術はそのままでは利用できず、分散トレースと呼ばれる技術が必要になります。分散トレース技術とは、1 つのリクエストが複数のサービスにまたがって処理をされる時に、どのサービスが呼び出されているか、どれぐらいの時間で処理されているか、などを調査できる技術になります。MSA では REST 呼出しを分散トレースする技術が必要になります。
「OpenTracing」は、分散トレース技術の一つで、現在デファクトスタンダードになりつつあります。MicoroProfile では、OpenTracing に対応した仕様、「MicroProfile OpenTracing」を提供しています。以下に、MicroProfile OpenTracing の使用方法を、可視化ツールの一つである「Jaeger」とともに紹介します。
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/201907/mptracing/（GitHub, Inc.）

REST サービスの作成

3 つのサービス、ServiceA、ServiceB、ServiceC を JAX-RS を使って作成し、それぞれ別のプロセスとして動作させます。ServiceA は ServiceB を呼び、ServiceB は ServiceC を呼ぶという関係（図 1）になります。

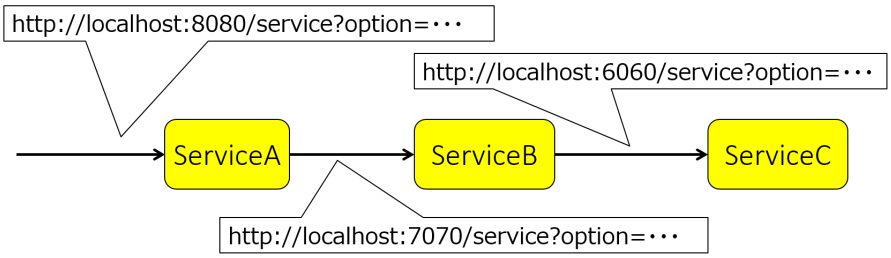

図 1

ServiceA のソースは以下のように、GET メソッドのエントリーポイントを持ち、JAX-RS クライアント API を使用し、ServiceB の呼び出しを行います。

```
@Path("service")
public class ServiceA {
  @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);
    return result;
  }
}
```


ServiceB のソースは以下のように、GET メソッドのエントリーポイントを持ち、JAX-RS クライアント API を使用し、ServiceC の呼び出しを行います。

```
@Path(”service”)
public class ServiceB {

  @Inject
  ServiceClient client;

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

@RequestScoped
class ServiceClient {
  public String call(String option) {
    String result = ClientBuilder
      .newClient()
      .target(”http://localhost:6060/service?option=” + option) // call ServiceC
      .request()
      .get(String.class);
    return result;
  }
}
```

また、ServiceC のソースでは、option パラメーターの値によって、意図的に 500ms スリープするコードを入れておきます。

```
public class ServiceC {
  @GET
  public String accept(@QueryParam(”option”) String option) {
    if (option.equals(”sleep”))
      try {
        Thread.sleep(500);   // intentionally delay response
      } catch (Exception e) {}
    return ”Service accepted : ” + option + ”¥n”;
  }
}
```

これらのソースは、JAX-RS の API しか使っておらず、OpenTracing 用のコードは一切含まれていません。すなわち、既存の JAX-RS のプログラムは、何も変更しなくても、トレースできるということを意味しています。

各サービスは、Web アプリケーションとして、それぞれ、「service-A.war」、「service-B.war」、「service-C.war」という war ファイルにパッケージしておきます。詳細については、pom.xml を参照してください。

作成したサービスの起動

作成した 3 つのサービスを動かす前に、Jaeger と Launcher の準備をします。その後、Launcher を使ってサービスを起動します。

Jaeger の準備

可視化ツールの Jaeger を準備します。ここでは、以下のように、docker image を使います。

```
$ docker pull jaegertracing/all-in-one

$ docker run -d --name jaeger ¥
  -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 ¥
  -p 5775:5775/udp ¥
  -p 6831:6831/udp ¥
  -p 6832:6832/udp ¥
  -p 5778:5778 ¥
  -p 16686:16686 ¥
```



```
  -p 14268:14268 ¥
  -p 9411:9411 ¥
  jaegertracing/all-in-one
```

トレース対象のプログラムを動かす際には、以下の環境変数を設定しておきます。。

```
export JAEGER_SERVICE_NAME=DEMO201907
export JAEGER_AGENT_HOST=localhost
export JAEGER_AGENT_PORT=6831
```

また、Jaeger UI は、以下でアクセスできるようになります。

```
http://localhost:16686
```

Launcher の準備

MicroProfile OpenTracing を使用するために、MicroProfile の実装の一つである、「Launcher」を用意します。ここでは、以下より、2.0-alpha-1 版をダウンロードして利用します。

- https://github.com/fujitsu/launcher/releases/download/2.0-a01/launcher-2.0-a01.jar（GitHub, Inc.）

Launcher の使い方は、ダウンロードした「launcher-2.0-a01.jar」を任意の場所に置いて、java コマンドの-jar オプションに指定するだけです（インストール作業は不要です）。詳細な Launcher の使用方法は、ドキュメント「lncusg」を参照してください。

- https://github.com/fujitsu/launcher/blob/master/docs/Usage.adoc（GitHub, Inc.）

3 つのサービスの起動

作成した 3 つのサービス（war ファイル）を以下のように起動させますが、java コマンドの実行時には、前々節で紹介した以下の 3 つの環境変数を忘れずに設定してください。

```
export JAEGER_SERVICE_NAME=DEMO201907
export JAEGER_AGENT_HOST=localhost
export JAEGER_AGENT_PORT=6831
```

ServiceA

```
$ java -jar launcher-2.0-a01.jar --http-listener 8080 --https-listener 8081 --deploy service-A.war
```

ServiceB

```
$ java -jar launcher-2.0-a01.jar --http-listener 7070 --https-listener 7071 --deploy service-B.war
```

ServiceC

```
$ java -jar launcher-2.0-a01.jar --http-listener 6060 --https-listener 6061 --deploy service-C.war
```


Jaeger によるトレースの可視化

ServiceA に対して、以下のようなリクエストを何度か送信します。

```
$ curl -X GET "http://localhost:8080/service?option=run"
$ curl -X GET "http://localhost:8080/service?option=sleep"
```

その後、Jeager のコンソールにアクセスすると、図 2 の画面が出ます。

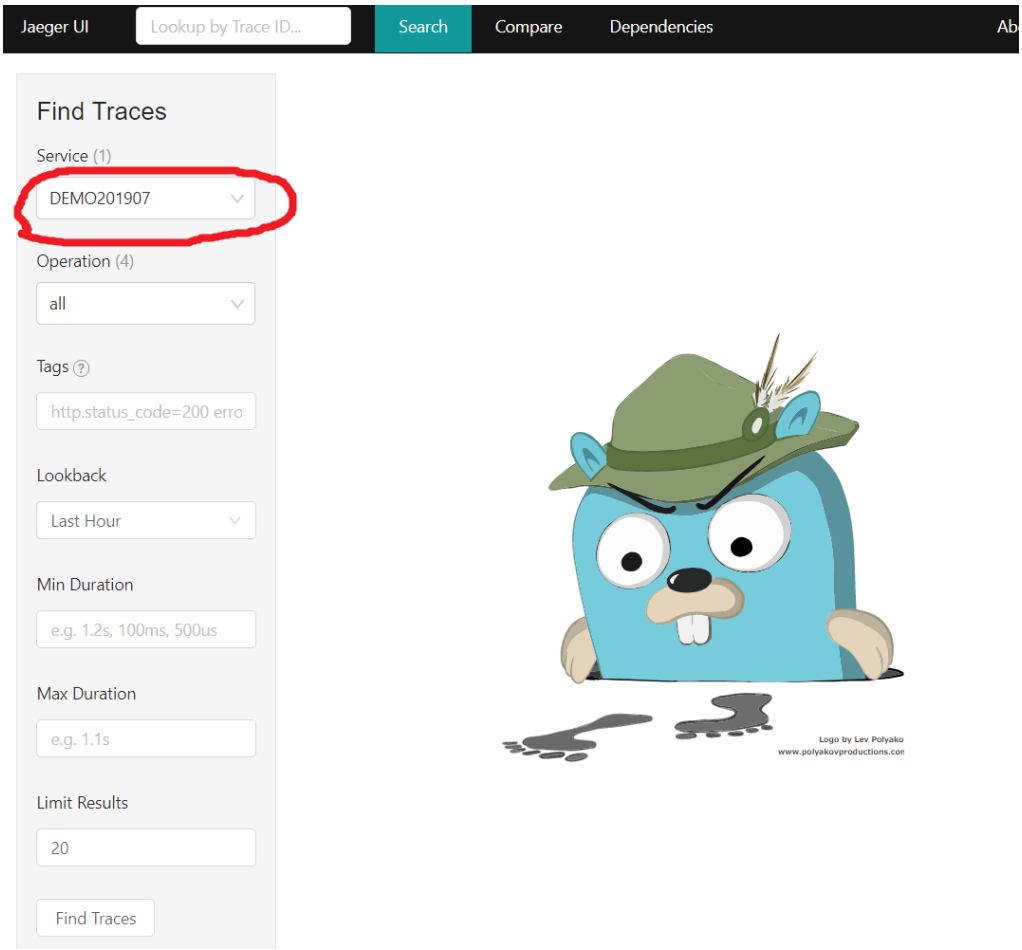

図 2

左上の Service フィールドに出ている「DEMO201907」というのは、環境変数「JAEGER_SERVICE_NAME」に設定した値が表示されています。

次に、左下の「Find Traces」ボタンを押すと、図 3 の画面が出ます。

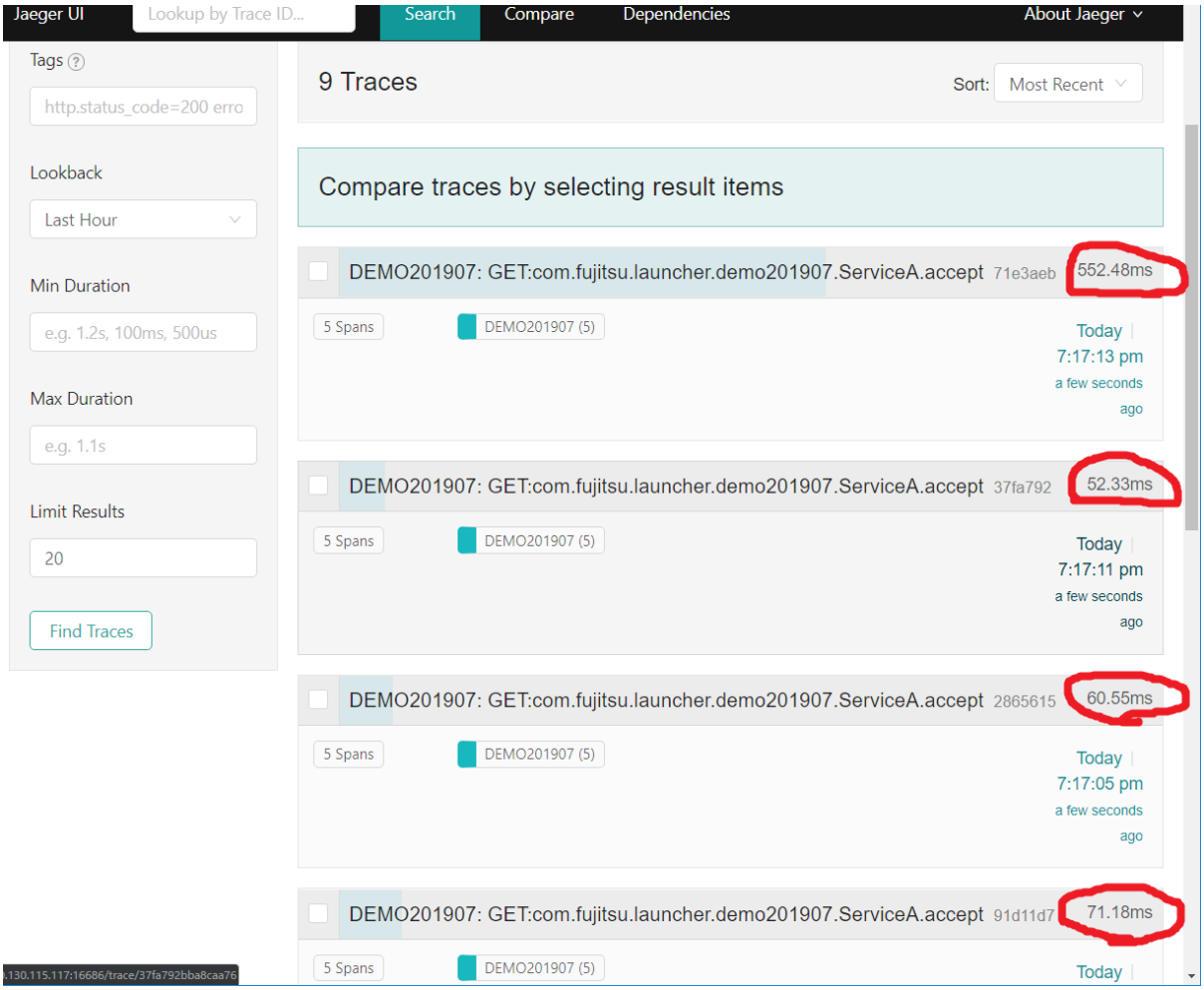

図 3

図 3 では、ServiceA.accept へのリクエストが画面上 4 つ表示されています。それぞれのトレース情報の右上に、リクエストに要した時間が表示されています。よく見ると、一番上のトレースが 552.48ms で、残りの 3 つのトレース時間に比べて、10 倍近くかかっていることが分かります。

次に、上から 2 つ目のトレースをクリックすると、図 4 の画面になります。

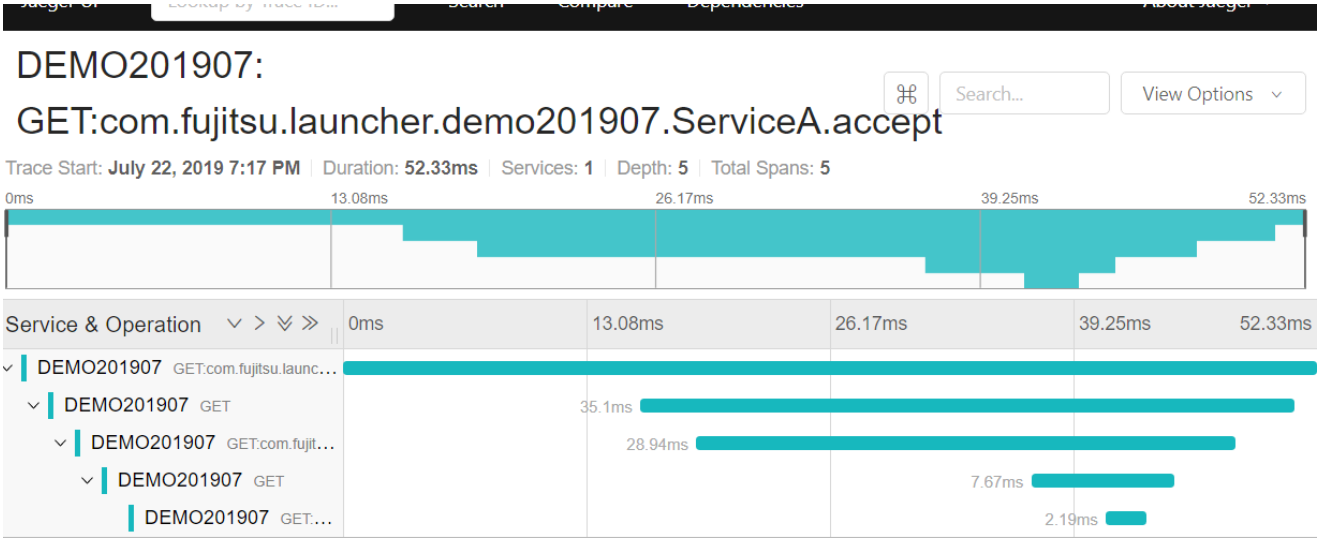

図 4

各サービス間の呼出し経路と、経過時間が表示されています。サービスのところをクリックすると、図 5 のように、HTTP のステータスコード、URL などの各サービスの詳細が出ます。

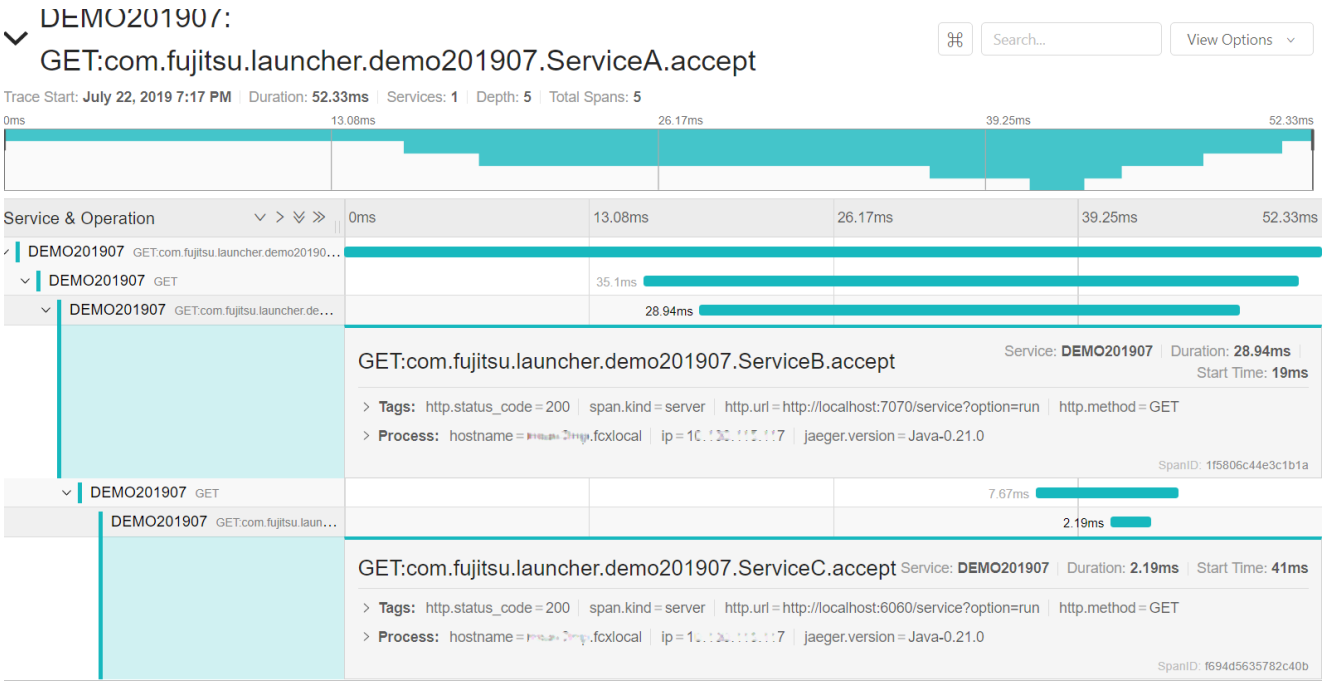

図 5

次に、トレース一覧の画面 (図 3) にもどって、一番上のトレースをクリックすると、図 6 の画面になります。同様に、ServiceC.accept の詳細情報が確認できます。URL を見ると、「option=sleep」が指定されていることが分かり、意図的に入れた「Thread.sleep(500)」のルートに入り、このメソッドで 502.16ms 費やすことになったことが分かります。

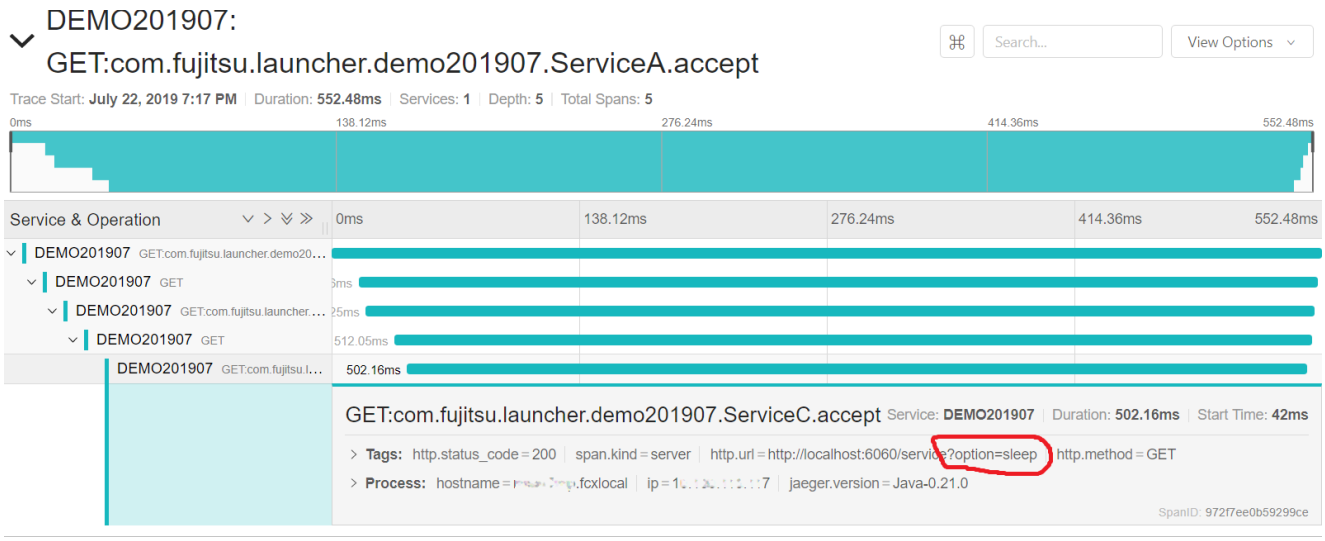

図 6

まとめと予告

今回 Part1 では、OpenTracing API は一切使わず、既存の JAX-RS のみのプログラムでもサービスの呼出し経路や、サービスの詳細情報を見ることができました。MSA ではサービス間の呼出し関係が複雑になりがちですが、MicroProfile OpenTracing を使うことで、呼出し関係やボトルネックの可視化ができるようになります。

次回 Part2 では、OpenTracing API を使って、より詳細なトレース情報の取り方を見ていきます。