MERT ÇALIŞKAN

Eclipse MicroProfile：軽量なエンタープライズJavaのオプション

マイクロサービスと分散アプリケーションのために設計されたJava EEアーキテクチャに迫る

Eclipse MicroProfileは、Java EEの力を使うマイクロサービスを開発するための小型プラットフォームを定義するものです。MicroProfile 1.0が最初にリリースされたのは、JavaOne 2016でのことでした。そこでは、IBM、Payara、Tomitribe、Red Hatの各ベンダーから4種類のランタイムが発表されました。それぞれのランタイムの名前は、以下のとおりです。

- IBM WebSphere Liberty
- Payara MicroProfile
- TomEE（Tomitribe製）
- WildFly Swarm（Red Hat製）

すべてのランタイムには、JAX-RS 2.0、Contexts and Dependency Injection（CDI）1.2、JSON-P 1.0という3つのコアJava EE仕様が含まれています。MicroProfileが推進されてきた要因は、マイクロサービスをベースとするアーキテクチャの構築基盤を提供する点にあります。そのため、MicroProfileのランタイムは、Java EE仕様の非常に小さなサブセットとなるように定義されています。本記事は、Java EE仕様になじみがあり、マイクロサービスをベースとするアーキテクチャに関する基本的な知識を持ち合わせている方がご覧になることを前提としています。

2017年8月にリリースされたMicroProfile 1.1には、MicroProfile-Config 1.0の最終版が含まれています。MicroProfile-Configの詳細は、次のセクションで説明します。

MicroProfile 1.2は、本号が発刊される直前にリリースされ、2018年にはバージョン2.0がリリースされる計画となっています。MicroProfile 1.2はJava EE 7の仕様に対応しており、バージョン2.0はJava EE 8の仕様に対応する予定です。

現在のEclipse MicroProfileでは、構成、ヘルス・チェック、フォルト・トレランス、メトリック、セキュリティという5つの基本的概念が重視されています。こういった機能を合わせれば、構成可能でレジリエンスを備えるとともに、

堅牢、セキュアといった特徴を兼ね備えたマイクロサービスを実装できるはずです。本記事では、MicroProfileの仕様に対応するAPIの詳しい使用方法を紹介します。MicroProfileの仕様に基づくサービスは、マイクロサービスの問題にうまく対処でき、しかもAPIは単純です。この点はきっと気に入っていただけるのではないかと思います。

MicroProfile-Config

MicroProfile-Config仕様では、実装と構成を分離する機能が提供されます。この分離により、1つの生成物を複数の場所にデプロイする際に、それぞれの動作を変えることができます。また、マイクロサービスの構成変更も実現できます。たとえば、生成物の再パッケージ化や再デプロイを行わずに、動的な方法で構成を変更できます。

MicroProfile-Config機能では、さまざまなソース（システム・プロパティ、環境変数、META-INF/microprofile-config.propertiesファイルや、カスタムのConfigSource実装など）から構成を集約し、1つの統合された一覧としてユーザーに提示することによって、前述の仕組みを実現しています。

現在利用できるAPIのバージョンは、1.1です。生成物は、Listing 1に示す依存性定義を使ってMavenから取得できます。この生成物は、すでにMavenのセントラル・リポジトリで公開されています。なお、このAPIを実行するためには、Java 8以降が必要です。MavenベースのプロジェクトにMicroProfile-Configを追加する方法を次に示します。

リスト1：

```
<dependency>
  <groupId>org.eclipse.microprofile.config</groupId>
  <artifactId>microprofile-config-api</artifactId>
  <version>1.1</version>
</dependency>
<repositories>
```

これはAPIを参照しているだけであるため、ベンダーが配布しているMicroProfileランタイムの実装が必要になる点に注意してください。それでは、APIの使用方法を示すコード・サンプルを見てみます。

後ほど方法を紹介しますが、構成はアノテーションを用いて簡単に注入できます。構成はプログラムから取得することも可能です。Listing 2に例を示します。

■ **リスト2：**

```
public class ConfigUsageExample {

  public void useTheConfig() {
    Config config = ConfigProvider.getConfig();

    String baseUrl =
      config.getValue("server.base.url", String.class);

    invokeEndpoint(baseUrl);
  }
}
```

システム・プロパティを使用して、実行時にこの構成を簡単にオーバーライドすることができます。Listing 3のスニペットに例を示します。ここでは、server.base.urlをオーバーライドしています。

■ **リスト3：**

```
java -jar myservice.jar \
   -Dserver.base.url=http://another.server/endpoint
```

Listing 4に示すように、@Injectアノテーションおよび@ConfigPropertyアノテーションを使用してBeanに注入することで、構成にアクセスすることもできます。

■ **リスト4：**

```
@ApplicationScoped
public class InjectedConfigUsageExample {

  @Inject
  @ConfigProperty(name="server.base.url")
  private String baseUrl;
```

```
    @Inject
    @ConfigProperty(name="server.port")
    private Optional<Integer> port;
}
```

指定されているプロパティserver.base.urlは、提供される構成ソースのいずれかで定義されている必要があります。定義が存在しない場合は、DeploymentExceptionがスローされます。一方、server.portプロパティはOptionalとして定義されているため、構成内にこのプロパティが存在しなくても、DeploymentExceptionは発生しません。

MicroProfile-FaultTolerance

MicroProfile-FaultTolerance仕様では、レジリエンスがあるアプリケーションの実装戦略が提供されます。この仕様の狙いは、TimeOut、RetryPolicy、Fallback、CircuitBreaker、Bulkheadなどの機能に着目し、実行ロジックと実行自体を分離することにあります。以下に、それぞれの考え方をコード・サンプルとともに示します。

Timeout：この機能では、マイクロサービスの実行が無応答状態になることを防ぎます。マイクロサービスにタイムアウト値を設定しておくことは重要です。この設定により、代替の実行メカニズム（Fallback、RetryPolicy、Circuit Breakerなど）を適用できるようになります。

Listing 5に示すような、@Timeoutアノテーションが付加されたメソッドには、タイムアウト・ポリシーが適用されます。そして、実行時間が、指定されたタイムアウト値を超えた時点でTimeoutExceptionがスローされます。アノテーションはクラス・レベルでも使用できます。その場合、タイムアウト・ポリシーはクラスで定義されているすべてのメソッドに適用されます。

リスト5：

```
@Timeout(500) // 500ミリ秒のタイムアウト
public List<Recommendation> fetchRecommendations() {
    // リコメンデーションの集計
}
```

@Timeoutは、@Asynchronousとともに使用することが推奨されています。この併用により、別のスレッドでのマイクロサービス実行が可能になるため、いつでも終了できるようになります。

RetryPolicy：この機能では、接続性の問題を克服するためのリトライ・ポリシーが提供されます。Listing 6に示す@Retryアノテーションは、メソッド・レベルまたはクラス・レベルで適用できます。両方が存在する場合、メソッド・レベルの定義は、クラス・レベルの定義よりも優先されます。ここに示す例では、最大リトライ回数は3回、最大リトライ時間は10,000ミリ秒（10秒）になっています。指定した時間を超えた場合、リトライ回数にかかわらず、リトライは実行されなくなります。

リスト6：

```
@Retry(maxRetries = 3, maxDuration= 10000)
public void invokeService() {
   // 外部サービスの起動
}
```

Listing 7に示すように、RetryPolicyは例外に対して設定することもできます。これは、I/O処理など、外部リソースを扱う際の例外ハンドリングに便利です。

リスト7：

```
@Retry(retryOn = {IOException.class})
public void readFile() {
   // ...
}
```

Fallback：この機能では、実行が失敗した場合の代替ハンドリング方法が提供され、Timeout、RetryPolicy、CircuitBreakerがそれぞれの取り決めに基づいて失敗した場合に呼び出されます。タイムアウト値によって制限が設けられたメソッドに対するFallbackの定義と、そのハンドラの実装をListing 8に示します。この例では、service.retrieveAmount()メソッドの実行時間が3秒を超えた場合、ハンドラが実行されて値42が返ります。

リスト8：

```
@Timeout(3000)
@Fallback(MyFallback.class)
Long getAmount() {
```

```
    return service.retrieveAmount();
}

public class MyFallback implements FallbackHandler<Long> {
  Long handle(ExecutionContext context) {
    return 42;
  }
}
```

CircuitBreaker：この機能は、失敗が繰り返される状況を防ぎ、マイクロサービスの起動をできるだけ早く失敗させるものです。Listing 9に、メソッド呼出しに対するCircuitBreakerの定義例を示します。CircuitBreakerは、Timeout、Fallback、Asynchronous、Bulkhead、RetryPolicyとともに使用できます。

リスト9：

```
@CircuitBreaker(successThreshold = 10,
                requestVolumeThreshold = 4,
                failureRatio=0.75,
                delay = 1000)
public void invokeService() {
   // 外部サービスの起動
}
```

このCircuitBreakerでは、直近4回の連続したinvokeService()メソッド呼出しのうち、3回が失敗（requestVolumeThreshold × failureRatio）した場合、回路を開放する（すなわち、メソッド呼出しが失敗する）ことを指定しています。すると、この回路は1秒間（delay）開放されたままとなり、その後、半開の状態に遷移します。半開の状態では、サービスの呼出しが少なくとも1回失敗した場合、回路は開放されたままになります。回路がクローズ状態に遷移するためには、10回連続で呼出しが成功（successThreshold）する必要があります。クローズはデフォルトの状態で、メソッド呼出しが失敗せずに期待どおりの動作をするという意味です。

Bulkhead：この機能では、システム全体の障害を防ぐために、マイクロサービスへの同時リクエスト数を制限します。これは、実行が失敗するコードと、システム内で機能しているその他の部分との間に設けた防護壁と考える

ことができます。

ここで提供されているのは、スレッドプールの分離またはセマフォの分離です。@Asynchronousアノテーションを使用して、スレッドプールを分離することができます。Listing 10の例では、同時リクエスト数を3、待機キューのサイズを5に制限しています。制限値に達した場合、スレッドは待機キューに格納されます。待機キューも制限値に達した場合、BulkheadExceptionがスローされます。

リスト10：

```
@Asynchronous
@Bulkhead(value = 3, waitingThreadQueue = 5)
public Future<Connection> acquireConnection() {
  Connection conn = createConnection();
  return CompletableFuture.completedFuture(conn);
}
```

セマフォ形式の防護壁の定義例をListing 11に示します。

リスト11：

```
@Bulkhead(value = 3)
public Future<Connection> acquireConnection() {
  Connection conn = createConnection();
  return CompletableFuture.completedFuture(conn);
}
```

MicroProfile-HealthCheck

MicroProfile-HealthCheck仕様では、プロデューサを使ってマイクロサービスの健全性を診断するAPIや、状態の計測を行うAPIが提供されます。サービスのヘルス・チェックを行うプロシージャは、MicroProfileのランタイムが提供してくれるものと思います。そこで、プロシージャの実装方法と、レスポンスとして返される実行結果について見てみます。

プロシージャは、HealthCheckインタフェースを実装する必要があり、Responseのインスタンスを結果として返します。この実装の詳細をListing 12に示します。

■ **リスト12：**

```
@FunctionalInterface
public interface HealthCheck {
   Response call();
}

public abstract class Response {
   public enum State { UP, DOWN }
   public abstract String getName();
   public abstract State getState();
   public abstract
     Optional<Map<String, Object>> getAttributes();
   ...
}
```

HealthCheckサービスの実装例をListing 13に示します。

■ **リスト13：**

```
public class SuccessfulCheck implements HealthCheck {
   @Override
   public Response call() {
      return Response.named("successful-check").up();
   }
}
```

プロデューサはHTTPエンドポイントを提供する必要がありますが、TCPやJava Management Extensions（JMX）などのプロトコルをサポートすることもできます。プロデューサに送信されるリクエストは、プロトコル固有のもので構いませんが、レスポンスは適切なHTTPステータス・コードがマッピングされたJSON形式である必要があります。ステータス・コードとその詳しい説明をTable 1に示します。

サンプルのJSONレスポンスをリスト14に示します。できるだけ詳しい情報を提供するために、データの内部にキーと値のペアを定義できるようになっています。

リスト14：

```
{
 "outcome":"UP",
 "checks": [
  {
   "id": "diskCheck",
   "result":"UP",
   "data": {
    "disk.free.space":"20gb"
   }
  }
 ]
}
```

MicroProfile-HealthCheck仕様では、使われているヒープの量、過剰なガベージ・コレクション、ディスク領域の枯渇、デッドロックが発生したスレッドなどを確認できます。

MicroProfile-Security

多くの場合、マイクロサービスに基づくアーキテクチャで使われるセキュリティは、RESTfulサービスのセキュリティ手法と似たものになっています。通常、RESTfulサービスはステートレスです。そのため、セキュリティ・コンテキストはクライアントから提供されるトークンを使ってリクエストごとに再作成されます。MicroProfile-

ステータス・コード	説明
200	チェックを実行した結果、肯定的な結果が返された
204	ランタイムにヘルス・チェックがインストールされていない
503	チェックを実行した結果、否定的な結果が返された
500	問題が発生したため、チェックの実行を完了できなかった

表1：ステータス・コード

Security仕様は、トークンに基づくこのアプローチに従っており、認証や、ロールに基づく認可の処理にJSON Web Token（JWT）を使用しています。JWTトークン形式の例をListing 15に示します。

リスト15：

```
{
 "iss":"https://server.example.com",
 "sub":"24400320",
 "preferred_username":"jdoe",
 "aud":"s6BhdRkqt3",
 "nonce":"n-0S6_WzA2Mj",
 "exp":1311281970,
 "iat":1311280970,
 "auth_time":1311280969,
 "realm_access":{
  "roles":[
   "role-in-realm",
   "user",
   "manager"
  ]
 },
 "resource_access":{
  "my-service":{
   "roles":[
    "role-in-my-service"
   ]
  }
 }
}
```

業界は、モノリシック・アーキテクチャから、よりマイクロサービスに基づいたモデルへと方向を変えつつあります。そして、Java EEエコシステムの新しい仕様には、そういったニーズを満たす兆しが見え始めています。

リスト15のイタリック体の行は、JWTの標準機能とは別にこの仕様で定義されているアドオンに当たる部分です。ユーザー名はpreferred_usernameクレーム内にあり、レルムで付与されているロールはrealm_accessクレーム内にあります。resource_accessクレームは、サービスに固有な一連のロールを定義しています。

まとめ

業界は、モノリシック・アーキテクチャから、よりマイクロサービスに基づいたモデルへと方向を変えつつあります。そして、Java EEエコシステムの新しい仕様には、そういったニーズを満たす兆しが見え始めています。Java EEは、誕生してからの20年間でさまざまなリリースを経て長い道のりを歩んできました。そのため、Java EEは、マイクロサービスや現在登場しつつある新しい仕様の土台として十分に成熟しています。Eclipse MicroProfileは、CDIやJAX-RSなどのコアJava EE仕様を、新たなソリューションを構築するための基盤として使用しています。そういったソリューションは、マイクロサービス・エコシステムを形成する要求に応えるものです。本記事で説明した4つの仕様は、その始まりにすぎません。近い将来、ワーキング・グループが策定を行っているその他の仕様の詳細も明らかになるでしょう。</article>

[編集注：（最新情報）本号の発刊直前に、オラクルがMicroProfileイニシアチブに参加したことを発表しました。]

Mert Çalis¸kan（@mertcal）：Java Champion。『PrimeFaces Cookbook』（Packt Publishing、2013年）および『Beginning Spring』（Wiley Publications、2015年）の共著者で、Java Magazineでも多くの記事を執筆。トルコ最大のJavaユーザー・グループであるAnkara JUGに積極的に貢献しつつ、Payara FoundationでPayara Serverの開発に従事している。