Flux vs Mono ,Spring y la programación reactiva

27 junio, 2019 Por Cecilio Álvarez Caules — Deja un comentario

¿Flux vs Mono y la programación reactiva? . Todos estamos cada día más interesados en los conceptos que pertenecen a la programación Reactiva. Lamentablemente en muchas ocasiones es muy difícil entender cómo funciona este tipo de programación y como conceptos fundamentales de ella trabajan . Sobre todo si van apareciendo frameworks y más frameworks que generan capas de abstracción sobre nuestro código haciéndolo más difícil de comprender . Vamos a construir un ejemplo sencillo que nos ayude a entender cómo funcionan estas cosas. Para ello nos vamos a construir en un primer lugar un ejemplo básico de Spring y un servicio REST que nos permita identificar el problema ,vamos con ello. El primer paso es construir una aplicación clásica con Spring Boot para ello solicitaremos que el proyecto maven incluya lo siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.arquitecturajava</groupId>
  <artifactId>rest1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>rest1</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

```

```
33.
```



```
38.           <artifactId>spring-boot-maven-plugin</artifactId>
39.         </plugin>
40.       </plugins>
41.     </build>
42.
43.   </project>
```

Es momento de construir un sencillo servicio de Spring Framework el cual nos devuelva unas cadenas de texto elementales

```
1.  package com.arquitecturajava.rest1;
2.
3.  import org.springframework.stereotype.Service;
4.
5.  @Service
6.  public class HolaService {
7.
8.    public String hola() {
9.
10.     try {
11.       Thread.sleep(3000);
12.     } catch (InterruptedException e) {
13.       // TODO Auto-generated catch block
14.       e.printStackTrace();
15.     }
16.
17.     return "hola sincrono";
18.
19.   }
20.
21.   public String hola2() {
22.
23.     try {
24.       Thread.sleep(3000);
25.     } catch (InterruptedException e) {
26.       // TODO Auto-generated catch block
27.       e.printStackTrace();
28.     }
29.
30.
31.     return "hola sincrono 2";
32.
33.   }
34. }
```

Este código tiene de especial que cada vez que invocamos cada uno de los métodos del servicio hola y hola2 pondremos a dormir el Thread principal durante 3 segundos . Es momento de construir un servicio REST que invoque a estos métodos y nos permite combinarlos de alguna forma.

```
1.  package com.arquitecturajava.rest1;
2.
```

```
9.  public class ControladorHola {
```



```
14.    @RequestMapping("/hola")
15.    public String hola() {
16.
17.      return servicio.hola();
18.    }
19.
20.    @RequestMapping("/hola2")
21.    public String hola2() {
22.
23.      return servicio.hola2();
24.    }
25.
26.    @RequestMapping("/holas")
27.    public String holas() {
28.
29.      long t1= System.currentTimeMillis();
30.      String texto=servicio.hola()+ servicio.hola2();
31.      long t2= System.currentTimeMillis();
32.      System.out.println(t2-t1);
33.      return texto;
34.    }
35.  }
```

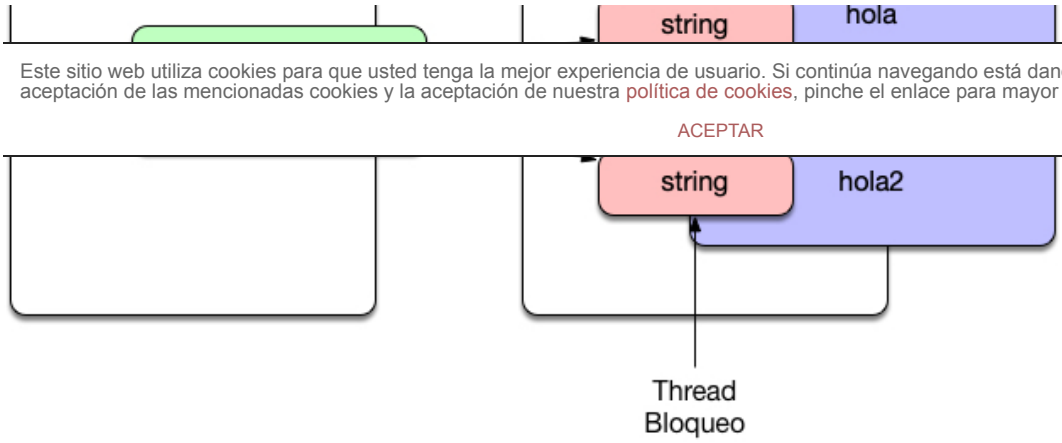
En este caso hemos inyectado con autowired el servicio y mapeado cada uno de sus métodos pero ademas hemos añadido un método extra que nos permite combinar ambos (el método holas) . Si ejecutamos esto vía Web nos daremos cuenta de que la ejecución tarda 6 segundos en mostrar el mensaje en el navegador.

hola sincronohola sincrono 2

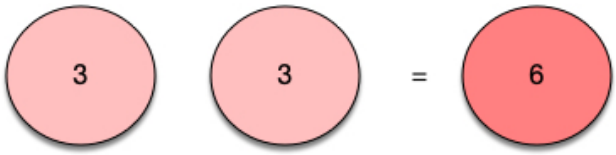
Si vemos los tiempos de ejecución por la consola con los logs que hemos realizado veremos que son 6 segundos:

```
2019-06-27 16:42:55.838  INFO 11487 --- [nio-8080-exec-1]
6007
```

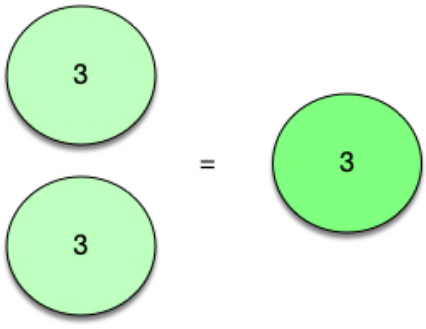
Es razonable ya que ambos métodos se ejecutan de forma sincrona , hay que esperar a que finalice el primero para ejecutar el segundo no tenemos ninguna otra opcion.



Si lo pensamos un poco más a detalle quizás cada método pudiera haberse ejecutado de forma asíncrona de tal forma que su ejecución no bloqueara o obligara a esperar para ejecutar el siguiente. Este es un tema interesante y es en el que Node.js basa sus principios . Es decir la ejecución del programa puede continuar mientras una petición asíncrona se resuelve de tal forma q por ejemplo si dos peticiones son asíncronas y cada una tarda 3 segundos se ejecuten ambas en paralelo .



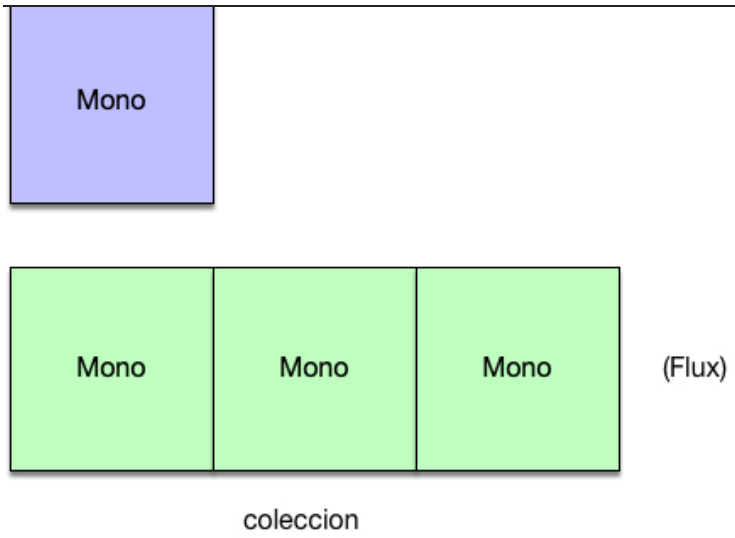
sincrono



asincrono

Esto evidentemente mejora los tiempos de respuesta ya que no bloqueamos la ejecución de nuestro código. El problema es que para conseguir este tipo de comportamiento necesitamos utilizar un nuevo framework a nivel de Spring y este framework es el que se denomina **Spring Reactor.**

sirve el tipo Flux? . Muy Sencillo Mono hace referencia a un elemento de programación



Si nos fijamos en el ejemplo anterior de programación síncrona tenemos una situación en la que cada método devuelve un String y otra que bueno aunque devolvemos un String podríamos decir que es un String compuesto de 2 cadenas cada una con su propio mensaje. Por lo tanto por hacer un simil nos encontramos ante una situación en la cual tenemos dos métodos que nos devuelven objetos Mono y un método que nos devolvería un Flux . Vamos a construir nuestro primer ejemplo con programación Reactiva. El primer paso es construirmos como siempre el proyecto con Spring Boot , en este caso no se trata de un proyecto normal y corriente sino que será reactivo.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
3.    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.    <modelVersion>4.0.0</modelVersion>
5.    <parent>
6.      <groupId>org.springframework.boot</groupId>
7.      <artifactId>spring-boot-starter-parent</artifactId>
8.      <version>2.1.6.RELEASE</version>
9.      <relativePath/> <!-- lookup parent from repository -->
10.   </parent>
11.   <groupId>com.arquitecturajava</groupId>
12.   <artifactId>rest2</artifactId>
13.   <version>0.0.1-SNAPSHOT</version>
14.   <name>rest2</name>
15.   <description>Demo project for Spring Boot</description>
16.
17.   <properties>
18.     <java.version>1.8</java.version>
19.   </properties>
20.
21.   <dependencies>
22.     <dependency>
```

```
29.         <artifactId>spring-boot-starter-test</artifactId>
```



```
34.         <artifactId>reactor-test</artifactId>
35.         <scope>test</scope>
36.       </dependency>
37.     </dependencies>
38.
39.     <build>
40.       <plugins>
41.         <plugin>
42.           <groupId>org.springframework.boot</groupId>
43.           <artifactId>spring-boot-maven-plugin</artifactId>
44.         </plugin>
45.       </plugins>
46.     </build>
47.
48.   </project>
```

Construyendo un ejemplo

Podemos observar como hemos añadido el starter de WebFlux que es el que necesitamos en Spring para poder usar los conceptos fundamentales de Reactor. El siguiente paso es volver a construir nuestras clases de Spring pero usando en este caso programación asíncrona con los conceptos de Mono y Flux

```
1.  package com.arquitecturajava.rest2;
2.
3.  import java.time.Duration;
4.
5.  import org.springframework.stereotype.Service;
6.
7.  import reactor.core.publisher.Mono;
8.
9.  @Service
10. public class HolaService {
11.
12.   public Mono<String> hola() {
13.
14.     return Mono.just("hola asincrono").delayElement(Duration.ofSeconds(3));
15.
16.   }
17.
18.   public Mono<String> hola2() {
19.
20.     return Mono.just("hola asincrono").delayElement(Duration.ofSeconds(3));
21.
22.
23.   }
24. }
```

El código se parece muchisimo al anterior solo que tiene la parte especial de que tratamos con el concepto de Mono en este caso . Cada vez que invoquemos estos métodos tardarán en devolvernos el valor 3 segundos cada uno de ellos . Eso sí no son métodos que se bloqueen uno

```
2.
7.  import reactor.core.publisher.Flux;
8.  import reactor.core.publisher.Mono;
9.
10. @RestController
11. @RequestMapping
12. public class ControladorHola {
13.
14.   @Autowired
15.   HolaService servicio;
16.
17.   @RequestMapping("/hola")
18.   public Mono<String> hola() {
19.
20.     return servicio.hola();
21.   }
22.
23.   @RequestMapping("/hola2")
24.   public Mono<String> hola2() {
25.
26.     return servicio.hola2();
27.   }
28.
29.   @RequestMapping("/holas")
30.   public Flux<String> holas() {
31.
32.     Mono<String> mono1= servicio.hola();
33.     Mono<String> mono2=servicio.hola2();
34.
35.     Flux<String> flujo=Flux.concat(mono1,mono2);
36.
37.     return flujo;
38.   }
39. }
```

Cómo podemos ver el método de /holas se encarga de combinar ambos tipos Monos , recordemos que cada uno de estos tipos es único y representa un elemento. Por lo tanto la combinación de ambos será un tipo Flux que es lo que devolvemos . Si ejecutamos nuestro código con Spring Boot y arrancamos el proyecto nos podremos dar cuenta que hay muchas cosas que cambian. La primera de ellas es que este código no se ejecuta sobre el paraguas de Tomcat lo cual es lo más habitual dentro del mundo Java sino que lo hace dentro de Netty un framework de programación asíncrona puro.

```
[           main] c.a.rest2.Rest2Application
[           main] c.a.rest2.Rest2Application
[           main] o.s.b.web.embedded.netty.NettyWebServer
[           main] c.a.rest2.Rest2Application
```

2. Hot vs Cold Observable con Rx.js
3. Introducción a RxJava y sus observables

Descargar PDF

Archivado en: Spring 5

Deja un comentario

Tu dirección de correo electrónico no será publicada.

Comentario

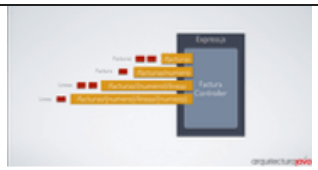
Nombre

Correo electrónico

Web

Este sitio usa Akismet para reducir el spam. Aprende cómo se procesan los datos de tus comentarios.

Cursos Gratuitos



Introduccion JPA

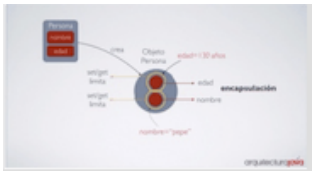
Java Herencia

Java JDBC

Servlets

Mis Cursos de Java

Programación Orientada a Objeto en Java



Java APIS Core

Java Web

Pack Java Core

CONTACTO

contacto@arquitecturajava.com