10分钟入门响应式:Springboot整合kafka实现reactive

dafei1288 麒思妙想 7月25日

收录于话题

#scala 2 #星期五 30 #分布式系统 1

Springboot引入Reactor已经有一段时间了，笔者潜伏在各种技术群里暗中观察发现，好像scala圈子的同仁们，似乎对响应式更热衷一点。也许是因为他们对fp理解的更深吧，所以领悟起来障碍性更少一些的原因吧。尽管webflux对于数据库的支持，还不那么完善，也不妨我们试上一试。

首先请允许我引用全部的反应式宣言作为开篇，接下来会介绍webflux整合kafka做一个demo。

反应式宣言

在不同领域中深耕的组织都在不约而同地尝试发现相似的软件构建模式。希望这些系统会更健壮、更具回弹性 、更灵活，也能更好地满足现代化的需求。

近年来，应用程序的需求已经发生了戏剧性的更改，模式变化也随之而来。仅在几年前， 一个大型应用程序通常拥有数十台服务器、 秒级的响应时间、 数小时的维护时间以及GB级的数据。而今，应用程序被部署到了形态各异的载体上，从移动设备到运行着数以千计的多核心处理器的云端集群。用户期望着毫秒级的响应时间，以及服务100%正常运行（随时可用）。而数据则以PB计量。昨日的软件架构已经根本无法满足今天的需求。

我们相信大家需要一套贯通整个系统的架构设计方案， 而设计中必需要关注的各个角度也已被理清， 我们需要系统具备以下特质：即时响应性（Responsive）、回弹性（Resilient）、弹性（Elastic）以及消息驱动（Message Driven）。我们称这样的系统为反应式系统（Reactive System）。

反应式系统更加灵活、松耦合和 可伸缩。这使得它们的开发和调整更加容易。它们对系统的失败 也更加的包容， 而当失败确实发生时， 它们的应对方案会是得体处理而非混乱无序。反应式系统具有高度的即时响应性， 为用户提供了高效的互动反馈。

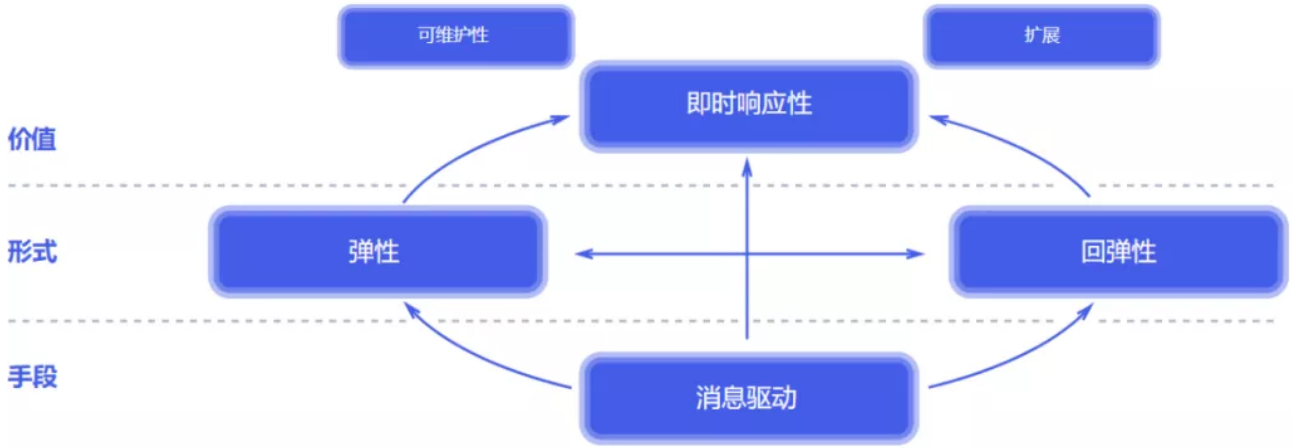
反应式系统的特质:

即时响应性:： 只要有可能， 系统就会及时地做出响应。即时响应是可用性和实用性的基石， 而更加重要的是，即时响应意味着可以快速地检测到问题并且有效地对其进行处理。即时响应的系统专注于提供快速而一致的响应时间， 确立可靠的反馈上限， 以提供一致的服务质量。这种一致的行为转而将简化错误处理、 建立最终用户的信任并促使用户与系统作进一步的互动。

回弹性: 系统在出现失败时依然保持即时响应性。这不仅适用于高可用的、 任务关键型系统——任何不具备回弹性的系统都将会在发生失败之后丢失即时响应性。回弹性是通过复制、 遏制、 隔离以及委托来实现的。失败的扩散被遏制在了每个组件内部， 与其他组件相互隔离，从而确保系统某部分的失败不会危及整个系统，并能独立恢复。每个组件的恢复都被委托给了另一个（外部的）组件， 此外，在必要时可以通过复制来保证高可用性。（因此）组件的客户端不再承担组件失败的处理。

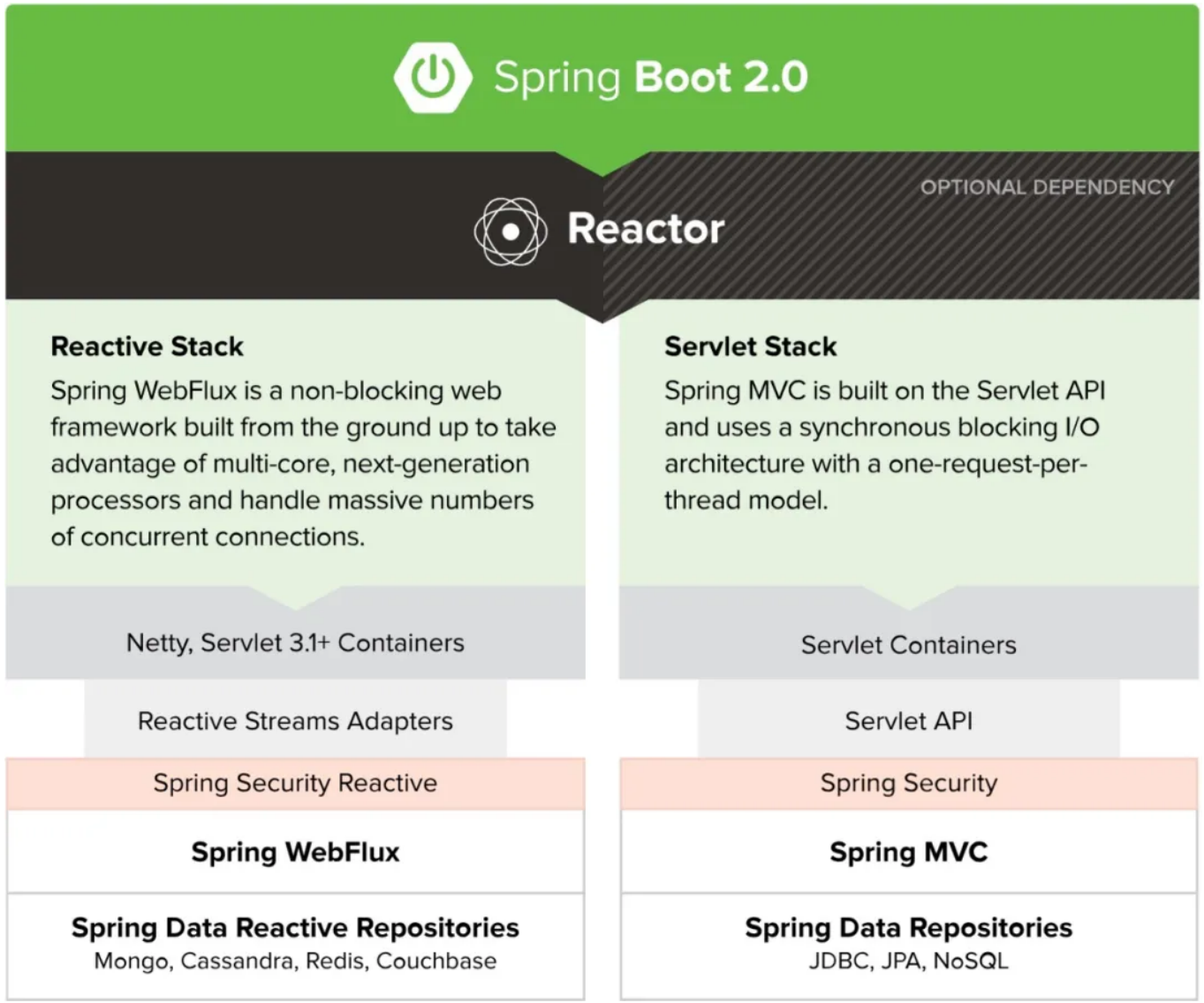
弹性： 系统在不断变化的工作负载之下依然保持即时响应性。反应式系统可以对输入（负载）的速率变化做出反应，比如通过增加或者减少被分配用于服务这些输入（负载）的资源。这意味着设计上并没有争用点和中央瓶颈， 得以进行组件的分片或者复制， 并在它们之间分布输入（负载）。通过提供相关的实时性能指标， 反应式系统能支持预测式以及反应式的伸缩算法。这些系统可以在常规的硬件以及软件平台上实现成本高效的弹性。

消息驱动：反应式系统依赖异步的消息传递，从而确保了松耦合、隔离、位置透明的组件之间有着明确边界。这一边界还提供了将失败作为消息委托出去的手段。使用显式的消息传递，可以通过在系统中塑造并监视消息流队列， 并在必要时应用回压， 从而实现负载管理、 弹性以及流量控制。使用位置透明的消息传递作为通信的手段， 使得跨集群或者在单个主机中使用相同的结构成分和语义来管理失败成为了可能。非阻塞的通信使得接收者可以只在活动时才消耗资源， 从而减少系统开销。



大型系统由多个较小型的系统所构成， 因此整体效用取决于它们的构成部分的反应式属性。 这意味着， 反应式系统应用着一些设计原则，使这些属性能在所有级别的规模上生效，而且可组合。世界上各类最大型的系统所依赖的架构都基于这些属性，而且每天都在服务于数十亿人的需求。现在，是时候在系统设计一开始就有意识地应用这些设计原则了， 而不是每次都去重新发现它们。

Springboot Webflux



引入springboot官网的一张图来解释Spring webflux和spring mvc的区别：

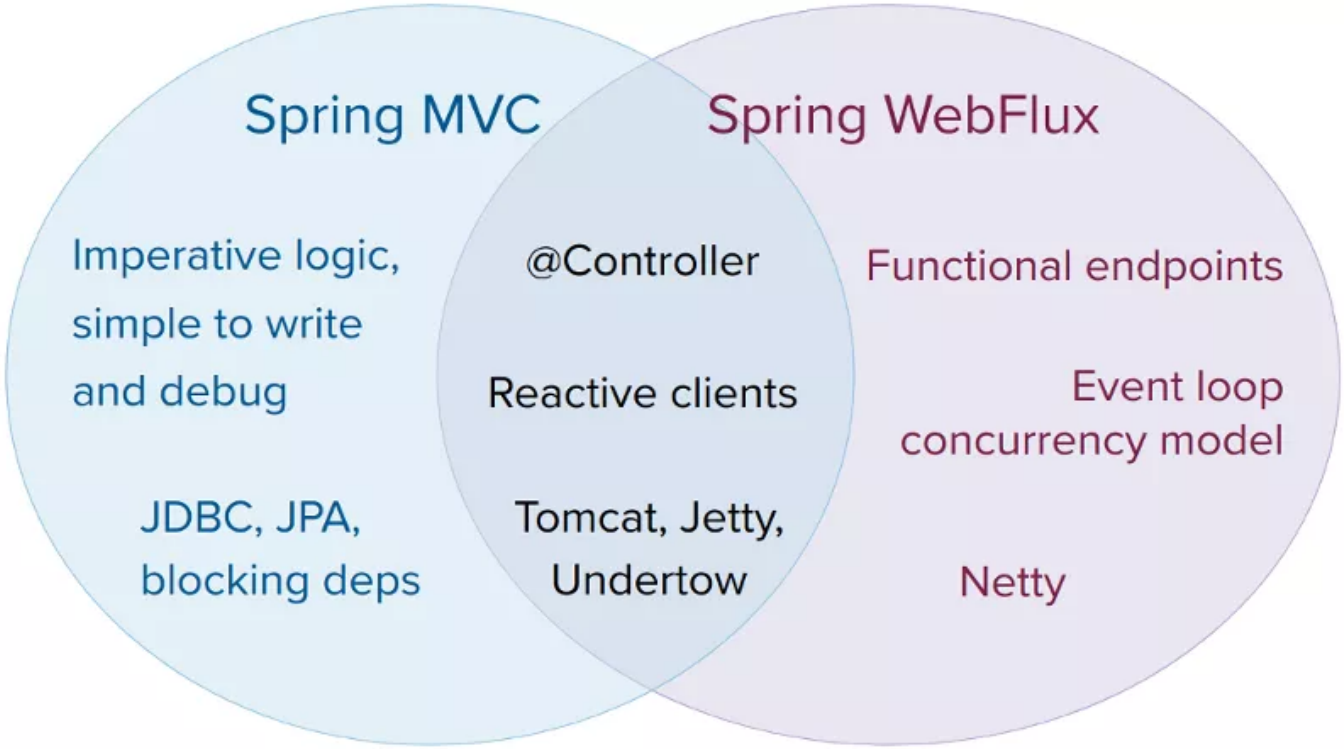
> Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture whth a one-request-per-thread model.

Spring MVC 构建在 Servlet API 之上，使用的是同步阻塞式 I/O 模型，什么是同步阻塞式 I/O 模型呢？就是说，每一个请求对应一个线程去处理。

> Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.

Spring WebFlux 是一个异步非阻塞式的 Web 框架，它能够充分利用多核 CPU 的硬件资源去处理大量的并发请求。

我们不难看出，持久层上Webflux针对部分NoSQL有一定的支持，而对应传统的关系型数据库就不那么友善了，这也许就是目前大部分javaer还是做着Crud Boy吧，限制了他们响应式的梦想 ... ...



当然非阻塞IO并不是银弹，如果你想用它来提升应用的访问效率，那么还是放弃吧，引用下面一段话，作为回答

> Reactive and non-blocking generally do not make applications run faster.

WebFlux 并不能使接口的响应时间缩短，它仅仅能够提升吞吐量和伸缩性。

所以如果你的应用是 IO密集型，还是很建议你试一试的。好了国际惯例TICSMTC...

Talk Is Cheap, Show Me The Code

我们本次应用的流程大体如下：创建一个路由用于生产数据，写入kafka里，然后再由注册的kafka消费者，消费该数据

引入依赖

这次demo使用了gradle代替了maven

```
implementation group: 'io.projectreactor.kafka', name: 'reactor-kafka', version: '1.3.4'
implementation group: 'org.springframework.kafka', name: 'spring-kafka', version: '2.7.4'
implementation group: 'org.springframework.boot', name: 'spring-boot-starter-webflux', version: '2.5.2'
```

构建实体

该实体，用于在kafka中传输

```
package wang.datahub.dto;

public class Warehouse {
  private Long id;
  private String name;
  private String label;
  private String lon;
  private String lat;

  public Long getId() {
     return id;
  }

  public void setId(Long id) {
     this.id = id;
  }

  public String getName() {
     return name;
  }

  public void setName(String name) {
     this.name = name;
  }

  public String getLabel() {
     return label;
  }
```

```
    public void setLabel(String label) {
        this.label = label;
    }

    public String getLon() {
        return lon;
    }

    public void setLon(String lon) {
        this.lon = lon;
    }

    public String getLat() {
        return lat;
    }

    public void setLat(String lat) {
        this.lat = lat;
    }

    @Override
    public String toString() {
        return "Warehouse{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", label='" + label + '\'' +
                ", lon='" + lon + '\'' +
                ", lat='" + lat + '\'' +
                '}';
```

```
  }
}
```

构建service

用于mock数据，并将对象发送至kafka

```
package wang.datahub.service;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import reactor.kafka.sender.SenderResult;
import wang.datahub.dto.Warehouse;

import java.util.Random;

@Service
public class WarehouseService {

  public static final String[] WAREHOUSE_NAME = new String[]{"天津仓库","北京仓库","上海仓库","广州仓库","深圳仓库"};
  public static final String[] WAREHOUSE_LEVEL = new String[]{"A级","B级","C级","D级","E级"};


  public Warehouse mock(long id) {
     Random random = new Random();
     try {
        Thread.sleep(random.nextInt(2000));
     } catch (InterruptedException e) {
     }

     Warehouse warehouse = new Warehouse();
     warehouse.setId(id);
     warehouse.setName(WAREHOUSE_NAME[random.nextInt(WAREHOUSE_NAME.length)]);
```

```
        warehouse.setLabel(WAREHOUSE_LEVEL[random.nextInt(WAREHOUSE_LEVEL.length)]);
        warehouse.setLon(random.nextDouble()+"");
        warehouse.setLat(random.nextDouble()+"");
        return warehouse;
    }


    @Autowired
    private ReactiveKafkaProducerTemplate template;

    public static final String WAREHOUSE_TOPIC = "warehouse";

    public Mono<Boolean> add(Warehouse warehouse) {
        Mono<SenderResult<Void>> resultMono = template.send(WAREHOUSE_TOPIC, warehouse.getId(), warehouse);
        return resultMono.flatMap(rs -> {
            if(rs.exception() != null) {
                System.out.println("send kafka error" + rs.exception());
                return Mono.just(false);
            }
            return Mono.just(true);
        });
    }


}
```

构建handler 并 注册 route

构建handler

```
package wang.datahub.handler;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import wang.datahub.dto.User;
import wang.datahub.dto.Warehouse;
import wang.datahub.service.WarehouseService;

@Component
public class WarehouseHandler {
  @Autowired
  WarehouseService warehouseService;
  private long i = 1;
  public Mono<ServerResponse> addWarehouse(ServerRequest request) {
     //mock 数据
     Warehouse warehouse = warehouseService.mock(i++);
     Mono<Boolean> tag = warehouseService.add(warehouse);
     return ServerResponse.ok().body(tag,Boolean.class);
  }
}
```

注册route，用于访问

```
package wang.datahub.route;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RequestPredicates;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerResponse;
import wang.datahub.handler.UserHandler;
import wang.datahub.handler.WarehouseHandler;

@Configuration
public class Routes {

  @Autowired
  UserHandler userHandler;

  @Autowired
  WarehouseHandler warehouseHandler;

  @Bean
  public RouterFunction<ServerResponse> routersFunction(){
     return RouterFunctions
          .route(RequestPredicates.GET("/api/warehouse"),warehouseHandler::addWarehouse);
  }
}
```

构建kafka消费者

```
package wang.datahub.kafka.consumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.kafka.KafkaProperties;
import org.springframework.kafka.core.reactive.ReactiveKafkaConsumerTemplate;
import org.springframework.stereotype.Service;
import reactor.kafka.receiver.ReceiverOptions;
import wang.datahub.dto.Warehouse;
import wang.datahub.service.WarehouseService;

import javax.annotation.PostConstruct;
import java.util.Collections;

@Service
public class WarehouseConsumer {
  @Autowired
  private KafkaProperties properties;

  @PostConstruct
  public void consumer() {
     ReceiverOptions<Long, Warehouse> options = ReceiverOptions.create(properties.getConsumer().buildProperties());
     options = options.subscription(Collections.singleton(WarehouseService.WAREHOUSE_TOPIC));
     new ReactiveKafkaConsumerTemplate(options)
          .receiveAutoAck()
          .subscribe(record -> {
              System.out.println("Warehouse Record:" + record);
          });
  }
}
```

配置springboot的kafka信息

```
spring:
kafka:
  producer:
    bootstrap-servers: 172.18.70.184:9092
    key-serializer: org.apache.kafka.common.serialization.LongSerializer
    value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
  consumer:
    bootstrap-servers: 172.18.70.184:9092
    key-serializer: org.apache.kafka.common.serialization.LongSerializer
    value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
    group-id: warehouse-consumers
```

构建完整应用

加载kafka配置

```
package wang.datahub;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.kafka.KafkaProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import reactor.kafka.sender.SenderOptions;

@Configuration
public class KafkaConfig {
  @Autowired
  private KafkaProperties properties;

  @Bean
  public ReactiveKafkaProducerTemplate reactiveKafkaProducerTemplate() {
     SenderOptions options = SenderOptions.create(properties.getProducer().buildProperties());
     ReactiveKafkaProducerTemplate template = new ReactiveKafkaProducerTemplate(options);
     return template;
  }
}
```

启动应用

```
package wang.datahub;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.reactive.config.EnableWebFlux;

@SpringBootApplication
@EnableWebFlux
public class WebfluxApplication {

  public static void main(String[] args) {
     SpringApplication.run(WebfluxApplication.class, args);
 }
}
```

kafka no zookeeper

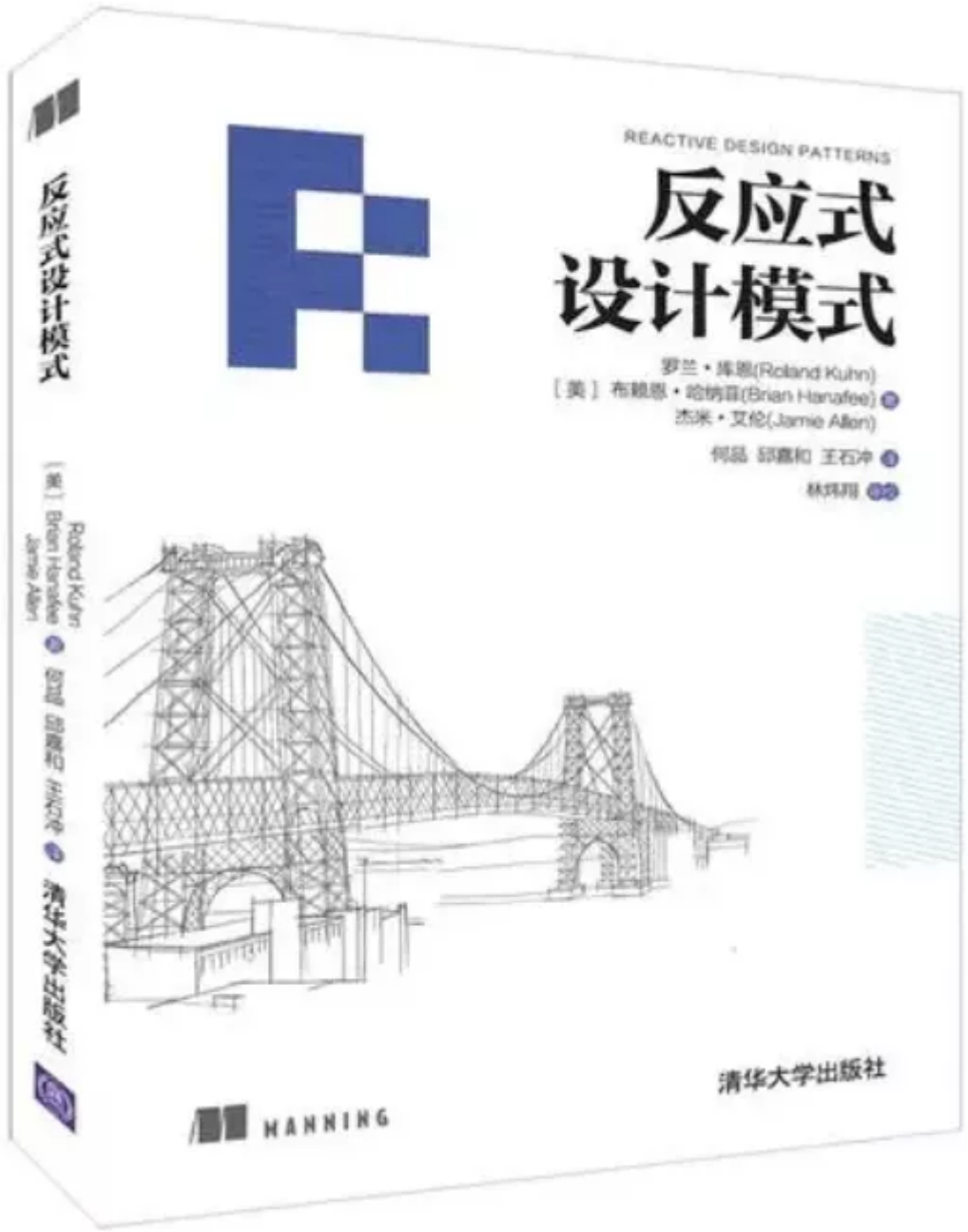
启动kafka还需要额外的zk配置，是不是让你很不爽呢？2.8开始，kafka已经开始准备着手去掉外部zk了，尽管现在还不推荐上生产环境，至少是一个好的开始，下面我们简单的看下，如何抛弃zk

```
⚡ root@DESKTOP-2J030JA ▒ /mnt/e/devlop/envs/kafka_2.13-2.8.0 ▒ bin/kafka-storage.sh random-uuid
dKcraOLdTH6PCYuGizZ1nw
⚡ root@DESKTOP-2J030JA ▒ /mnt/e/devlop/envs/kafka_2.13-2.8.0 ▒ bin/kafka-storage.sh format -t dKcraOLdTH6PCYuGizZ1nw -c
config/kraft/server.properties
Formatting /tmp/kraft-combined-logs
⚡ root@DESKTOP-2J030JA ▒ /mnt/e/devlop/envs/kafka_2.13-2.8.0 ▒ bin/kafka-server-start.sh config/kraft/server.properties
```

完整代码库地址：https://github.com/dafei1288/reactor_kafka_springboot_demo

写在最后

之所以写了这篇文章，也是因为最近在读的这本书，刚读了2章，就被其吸引了，对反应式也开始有了兴趣，尽管这本书里面的案例都是scala和akka的，但是还是挺推荐读一读的



同时感谢群里的几位大佬的催更以及指点，排名不分先后

@superleo @简单 @箱子 @龙叁 @Jacob_Bishop @tofu @星辰 @sfq

感谢你能看到这里，如果本文对您有一点点帮助，希望您与我交流，也期待您的转发和关注支持。

参考链接

https://www.reactivemanifesto.org/zh-CN

https://www.cnblogs.com/quanxiaoha/p/10713782.html

https://blog.csdn.net/qq_28423433/article/details/81221933?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-6.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-6.control

https://www.cnblogs.com/niechen/p/9303451.htmlhttps://www.jianshu.com/p/15d0a2bed6da

收录于话题 #星期五·30个

上一篇
手把手构建基于 GBase8s 的 Flink connector

下一篇
Spring to ZIO 101

喜欢此内容的人还喜欢

浅谈开源之道

麒思妙想

90% 中国人都缺乏的维生素，一张图教你吃回来。

混知

亳州境内这两条国道将改扩建→

药都时空