28 Spring Cloud Stream 深入实践

更新时间：2019-07-26 09:34:26

> 机遇只偏爱那些有准备的头脑。
>
> ——巴斯德

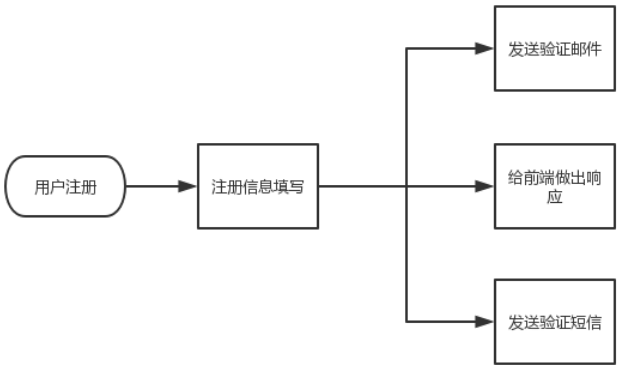
上篇文章和大家聊了 Spring Cloud Stream 的基本架构和基本用法，包括基本的消息收发、自定义消息通道、消息分组以及消息分区等。相信学习完之后，大家对于 Spring Cloud Stream 已经有了一个基本的认知。本文我想结合具体项目中的一些使用场景，再来带大家看看 Spring Cloud Stream 的用法。

异步处理

案例介绍

很多场景下，我们都是使用消息中间件而不是多线程来处理一些异步任务，这样可以更好地实现应用程序的解耦。一个常见的使用场景就是用户注册流程。一般来说，用户注册一个网站，可能都需要验证手机号码或者邮箱地址，如果不使用异步处理的话，我们的流程可能是这样的：

引入异步处理之后，我们就可以将验证信息的发送交给消息中间件去做，然后就可以快速给前端一个响应，优化后的流程像下面这样：

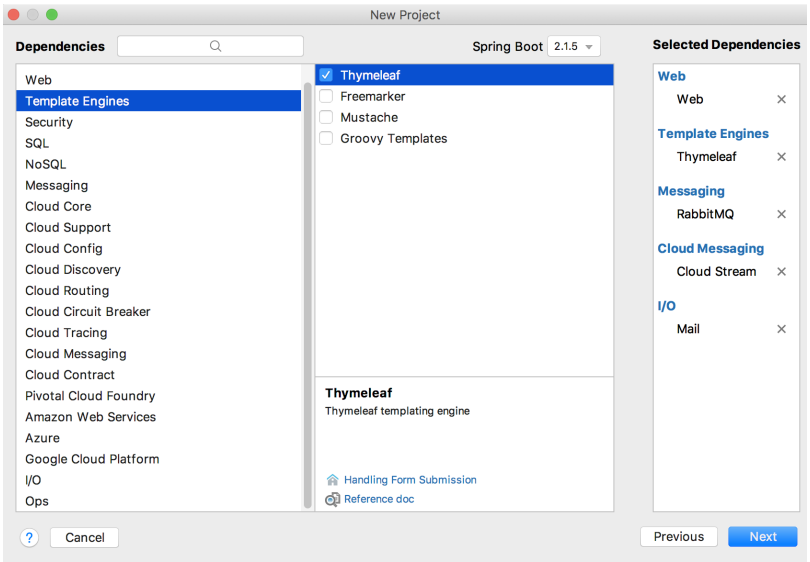


此时注册效率就会得到极大地提高。假设优化前每个流程需要 500ms，那么总共需要 2.5s，优化后则只需要 1.5s 就可以执行完步骤了。

接下来，我就通过一个简单的注册案例来向大家展示下 Spring Cloud Stream 在这个场景下的使用。

案例展示

首先我们来创建一个名为 streamdemo 的 Spring Boot 项目，创建时候添加四个依赖 Web、RabbitMQ、Cloud Stream、mail 以及 Thymeleaf，如下：



前三个依赖好理解，这和我们上篇文章所需要的依赖一样，后面两个则是用来发送邮件的，Mail 依赖用来添加邮件发送支持，Thymeleaf 则用来构建邮件发送模版。

Spring Boot 创建完成后，接下来我们还需要启动 Docker 容器中的 RabbitMQ 中间件，中间件启动成功之后，我们在 streamdemo 项目的 application.properties 中添加如下配置：

```
spring.rabbitmq.password=guest
spring.rabbitmq.username=guest
spring.rabbitmq.host=127.0.0.1
spring.rabbitmq.port=5672
```

这样首先确保我们的 Spring Boot 具有连接消息中间件的能力，然后我们来添加一个简单的注册接口：

```
@RestController
public class RegController {
    @Autowired
    RegService regService;
    @PostMapping("/doReg")
    public Map<String, Object> reg(String email, String phone, String password) {
        return regService.reg(email, phone, password);
    }
}
@Service
public class RegService {

    @Autowired
    RegChannel regChannel;

    public Map<String, Object> reg(String email, String phone, String password) {
        //数据写入数据库
        Map<String, Object> map = new HashMap<>();
        map.put("email", email);
        map.put("phone", phone);
        regChannel.output().send(MessageBuilder.withPayload(map).build());
        map.put("msg", "验证短信已经发送，请注意查收！");
        return map;
    }
}
```

这里为了简单处理，写入数据库的操作我就直接省略了。当服务端收到用户的注册信息时，先将信息保存到数据库中，然后向消息中间件发送消息。发送完成之后，剩下的验证消息发送就是其它服务模块的事情了，注册流程此时就可以直接返回了。

RegChannel 是一个自定义的消息通道，如下：

```
public interface RegChannel {
    String INPUT = "reg-input-channel";
    String OUTPUT = "reg-output-channel";

    @Output(OUTPUT)
    MessageChannel output();
    @Input(INPUT)
    SubscribableChannel input();
}
```

这里定义了两个消息通道，一个发送消息一个接收消息，应该不需要过多解释，和上篇文章基本一致。在实际生产环境中，根据项目的实际情况，我们可能会单独创建一个消息发送微服务，这里为了方便给大家演示，我将消息发送和接收放在同一个服务之中。当然，我们前文说过，这样定义之后，由于消息发送和接收不在同一个通道上，发送的消息是无法收到的，所以我们还需要在 application.properties 文件中继续添加如下配置：

```
spring.cloud.stream.bindings.reg-input-channel.destination=javaboy-topic
spring.cloud.stream.bindings.reg-output-channel.destination=javaboy-topic
```

然后我们再来定义一个消息消费者，用来读取消息中间件中的消息，如下：

```
@EnableBinding(RegChannel.class)
public class SendVerifyCodeService {
    @Autowired
    TemplateEngine templateEngine;

    @Autowired
    MailService mailService;

    @StreamListener(RegChannel.INPUT)
    public void sendVerifyCode(Map<String, Object> map) {
        //发送验证邮件和短信
        System.out.println("receive:" + map);
        Context ctx = new Context();
        String email = (String) map.get("email");
        ctx.setVariable("email", email);
        ctx.setVariable("code",(int)(Math.random()*10000));
        String mail = templateEngine.process("mailtemplate.html", ctx);
        mailService.sendHtmlMail("1510161612@qq.com",
                email,
                "欢迎注册XXX网站",
                mail);

    }

}
@Component
public class MailService {
    @Autowired
    JavaMailSender javaMailSender;

    public void sendHtmlMail(String from, String to,
                             String subject, String content){
        try {
            MimeMessage message = javaMailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message, true);
            helper.setTo(to);
            helper.setFrom(from);
            helper.setSubject(subject);
            helper.setText(content, true);
            javaMailSender.send(message);
        } catch (MessagingException e) {
            System.out.println("发送失败");
        }
    }
}
```

代码解释：

- 首先注入 TemplateEngine ，当我们在项目中引入 Thymeleaf 的依赖之后，就自动具备了这个 Bean 了，这个 Bean 一会儿用来将 Thymeleaf 模版渲染成 HTML 页面；
- 注入 MailService，这是一个我们封装好的邮件发送工具类；
- 监听邮件发送消息通道，在收到消息后，首先创建一个 Context 实例，这个实例中保存了我们即将渲染到 Thymeleaf 中的数据，然后向 Context 中保存两个变量，分别是 email 和生成的随机校验码 code ，这两个数据我们将在 Thymeleaf 模版中使用；
- 调用 TemplateEngine 中的 process 方法，将 Thymeleaf 模版渲染成 HTML 页面；
- 调用 MailService 中的 sendHtmlMail 方法，执行邮件发送工作。

当然，要实现邮件发送工作，我们还需要在 application.properties 中配置一下连接邮件服务器的必备信息：

```
spring.mail.host=smtp.qq.com
spring.mail.port=465
spring.mail.username=1510161612@qq.com
spring.mail.password=igprkcldlddxiiae
spring.mail.default-encoding=UTF-8
spring.mail.properties.mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory
spring.mail.properties.mail.debug=true
```

这里的配置信息，我们在前面第 6 章中提到过，这里我就不再赘述了。唯一需要说的是，password 字段不是真正的 password ，是我们申请到的一个授权码，授权码的具体申请方式参考本文附录。最后我们再来看看放在 resources/templates 目录下的邮件模版：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>注册验证</title>
</head>
<body>
<div>注册验证</div>
<div>您的注册信息是：
    <table border="1">
        <tr>
            <td>邮箱地址</td>
            <td th:text="${email}"></td>
        </tr>
        <tr>
            <td>验证码</td>
            <td th:text="${code}"></td>
        </tr>
    </table>
</div>
<div>
    如果您未注册本站，请忽略本邮件。
</div>
</body>
</html>
```

在邮件模版中，我们将动态渲染邮箱地址和验证码两个变量。好了，做完这两个操作之后，接下来我们就可以启动我们的 Spring Boot 项目了。启动成功之后，通过 POSTMAN 发送一个注册请求，发送成功之后，我们就可以收到注册邮件了，这个比较容易，我就不展示了。

定时任务

定时任务各种各样，常见的定时任务比如日志备份，我们可能在每天凌晨 3 点去备份，这种固定时间的定时任务我们一般采用 cron 表达式就能轻松实现。还有一些比较特殊的定时任务，像大家看电影中的定时炸弹，3分钟后爆炸，这种定时任务就不太好用 cron 去描述，因为开始时间不确定，我们开发中有的时候也会遇到类似的需求，此时通过消息中间件就能够很方便地解决。

整体上来说，在 RabbitMQ 上实现定时任务有两种方式：

- 利用 RabbitMQ 自带的消息过期和私信队列机制，实现定时任务，这种方式较复杂；
- 使用 RabbitMQ 的 rabbitmq_delayed_message_exchange 插件来实现定时任务，这种方案较简单，使用较普遍。

这里主要向大家展示第二种用法。

实践案例

首先我们需要下载 rabbitmq_delayed_message_exchange 插件。

rabbitmq_delayed_message_exchange插件下载

下载完成后解压，然后在命令行执行如下命令，将下载文件拷贝到 Docker 容器中去：

```
cp /Users/sang/Downloads/rabbitmq_delayed_message_exchange-20171201-3.7.x.ez some-rabbit:/plugins
```

这里第一个参数是宿主机上的文件地址，第二个参数是拷贝到容器的位置。

接下来再执行如下命令进入到 RabbitMQ 容器中：

```
docker exec -it some-rabbit /bin/bash
```

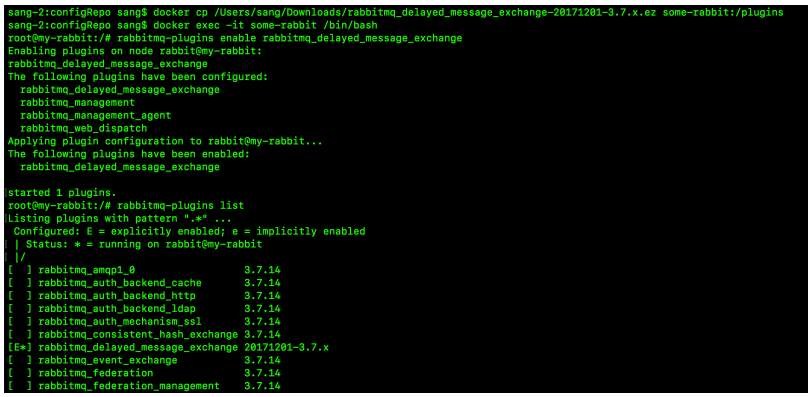
进入到容器之后，执行如下命令启用插件：

```
rabbitmq-plugins enable rabbitmq_delayed_message_exchange
```

启用成功之后，还可以通过如下命令查看所有安装的插件，看看是否有我们刚刚安装过的插件，如下：

```
rabbitmq-plugins list
```

命令的完整执行过程如下图：



OK，配置完成之后，接下来我们执行 exit 命令退出 RabbitMQ 容器，然后开始编码，接下来的案例我们直接在前文的基础上进行，我就不再另外单独搭建工程了。

首先我们来自定义一个消息通道，如下：

```
public interface DelayMsgChannel {
    String INPUT = "delay_msg_input";
    String OUTPUT = "delay_msg_output";

    @Input(INPUT)
    SubscribableChannel input();

    @Output(OUTPUT)
    MessageChannel output();
}
```

这里无需多做解释，接下来我们再来定义一个消息消费者：

```
@EnableBinding(DelayMsgChannel.class)
public class DelayMessageRecevier {
    @StreamListener(DelayMsgChannel.INPUT)
    public void recevier(String msg) {
        System.out.println("receive:" + msg + ">>>" + new Date());
    }

}
```

大家看到，这里的所有定义，都是和前面的案例一样，好像并没有体现出消息延迟相关的配置。

接下来，我们再在 application.properties 中添加如下配置：

```
spring.cloud.stream.bindings.delay_msg_input.destination=delay_msg
spring.cloud.stream.bindings.delay_msg_output.destination=delay_msg
spring.cloud.stream.rabbit.bindings.delay_msg_output.producer.delayed-exchange=true
spring.cloud.stream.rabbit.bindings.delay_msg_input.consumer.delayed-exchange=true
```

- 前两行配置表示配置消息队列；
- 后面两行表示分别在消息消费者和生产者中启用消息延迟功能。

配置完成之后，在添加一个消息生产者，如下：

```
@RestController
public class DelayMsgController {
    @Autowired
    DelayMsgChannel delayMsgChannel;
    @GetMapping("/delay")
    public void hello() {
        System.out.println("message send：" +new Date());
        delayMsgChannel.output().send(MessageBuilder.withPayload("delay message!").setHeader("x-delay", 3000).build());
    }
}
```

注意，和前文不一样的地方是，这里的消息生产者多了一个延迟的头字段。另外，我们在消息发送时还打印出时间日志，这样方便判断消息是否延迟。

做完这些事情之后，我们就可以启动项目了，启动成功之后，访问 `/delay` 接口，消息消费者就可以收到消息了。对比消息发送和接收时间，就可以发现消息延迟了三秒之后才收到，如下图：

```
2019-05-22 16:56:07.505  WARN 96718 --- [nio-8080-exec-1] .
message send: Wed May 22 16:56:11 CST 2019
2019-05-22 16:56:11.886  INFO 96718 --- [nio-8080-exec-2] c
2019-05-22 16:56:11.893  INFO 96718 --- [nio-8080-exec-2] c
2019-05-22 16:56:11.897  INFO 96718 --- [nio-8080-exec-2] c
2019-05-22 16:56:11.898  INFO 96718 --- [nio-8080-exec-2] c
receive:delay message!>>>Wed May 22 16:56:14 CST 2019
```

限流削峰

消息中间件另外一个广泛使用的场景就是限流削峰，大家知道解决高并发问题是一揽子方案，而不是靠某一种策略就能解决高并发问题的，那么限流削峰就是这一揽子方案中的一个。

以商品秒杀为例，请求如果直接进入到业务层，由于业务层处理比较复杂，例如库存检查、库存冻结、余额检查、余额冻结、订单生成、余额扣减、库存扣减、生成流水、余额解冻以及库存解冻等，这一套流程下来，耗时还是比较长的，在高并发环境下可能会把业务层搞瘫痪。

此时我们可以加入一个消息队列实现限流削峰，即所有的请求都先进入到消息队列中，业务模块再去消息队列中读取消息、挨个处理，整个过程还可以进行流量控制，这样就可以有效降低业务模块的压力。同时，在秒杀过程中，那些进入消息队列较晚的消息，肯定是秒杀不到商品的，这时这个请求就可以直接处理，可以直接给用户返回秒杀失败或者商品已售空。

小结

本文通过一个简单的例子向大家展示了 Spring Cloud Stream 在项目中的使用。实际上，Spring Cloud Stream 使用场景还是非常多的，例如 A 服务调用 B 服务，如果不需要及时知道 B 服务的执行结果，此时就可以引入消息中间件，如果 A 需要当时就知道 B 的执行结果，那么此时引入消息中间件就不合理了。把握住这一点，就能在项目中合理使用消息中间件和 Spring Cloud Stream 了。通过本文的介绍，相信大家已经发现，无论是哪一种场景，如果单纯从技术角度来说，用法基本上都是一样的，所以我们这里就给大家举两个典型例子就可以了。

附录

邮件协议

我们经常会听到各种各样的邮件协议，比如 SMTP、POP3、IMAP，那么这些协议有什么作用、有什么区别？我们先来讨论一下这个问题。

SMTP 是一个基于 TCP/IP 的应用层协议，江湖地位有点类似于 HTTP，SMTP 服务器默认监听的端口号为 25 。看到这里，小伙伴们可能会想到，既然 SMTP 协议是基于 TCP/IP 的应用层协议，那么我是不是也可以通过 Socket 发送一封邮件呢？回答是肯定的。

生活中我们投递一封邮件要经过如下几个步骤：

1.深圳的小王先将邮件投递到深圳的邮局；
2.深圳的邮局将邮件运送到上海的邮局 ；
3.上海的小张来邮局取邮件 。

这是一个缩减版的生活中邮件发送过程。这三个步骤可以分别对应我们的邮件发送过程，假设从 aaa@qq.com 发送邮件到 111@163.com：

1. aaa@qq.com 先将邮件投递到腾讯的邮件服务器；
2. 腾讯的邮件服务器将我们的邮件投递到网易的邮件服务器；
3. 111@163.com登录网易的邮件服务器查看邮件。

邮件投递大致就是这个过程，这个过程就涉及到了多个协议，我们来分别看一下。

SMTP 协议全称为 Simple Mail Transfer Protocol ，译作简单邮件传输协议。它定义了邮件客户端软件与 SMTP 服务器之间，以及 SMTP 服务器与 SMTP 服务器之间的通信规则。也就是说 aaa@qq.com 用户先将邮件投递到腾讯的 SMTP 服务器这个过程就使用了 SMTP 协议，然后腾讯的 SMTP 服务器将邮件投递到网易的 SMTP 服务器这个过程也依然使用了 SMTP 协议， SMTP 服务器就是用来收邮件。而 POP3 协议全称为 Post Office Protocol，译作邮局协议，它定义了邮件客户端与 POP3 服务器之间的通信规则。那么该协议在什么场景下会用到呢？当邮件到达网易的 SMTP 服务器之后，111@163.com 用户需要登录服务器查看邮件，这个时候该协议就用上了：邮件服务商都会为每一个用户提供专门的邮件存储空间，SMTP服务器收到邮件之后，就将邮件保存到相应用户的邮件存储空间中，如果用户要读取邮件，就需要通过邮件服务商的 POP3 邮件服务器来完成。最后，可能也有小伙伴们听说过 IMAP 协议，这个协议是对 POP3 协议的扩展，功能更强，作用类似，这里不再赘述。

发送QQ邮件准备工作

安全起见，QQ 邮箱在使用 Java 代码发送邮件时，无法直接使用密码，而是需要通过授权码认证，授权码获取需要首先登录QQ邮箱网页版，点击上方的设置按钮：

然后点击账户选项卡：

在账户选项卡中找到开启POP3/SMTP选项，如下：

点击开启，开启相关功能，开启过程需要手机号码验证，按照步骤操作即可，不赘述。开启成功之后，即可获取一个授权码，将该号码保存好，在使用 Java 代码登录时，这个授权码就是密码。

27 构建消息驱动的微服务

29 分布式链路跟踪和 Spring Cloud Sleuth

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论