26 Spring Cloud Bus 整合 RabbitMQ 与 Kafka

更新时间：2019-07-22 10:35:27

> 上天赋予的生命，就是要为人类的繁荣和平和幸福而奉献。
>
> ——松下幸之助

上篇文章和大家聊了 Docker 以及 RabbitMQ 和 Kafka 在 Docker 中的安装。软件装好之后，接下来我们就来看看 Spring Cloud Bus 给我们的微服务开发带来了哪些便利。

Spring Cloud Bus 简介

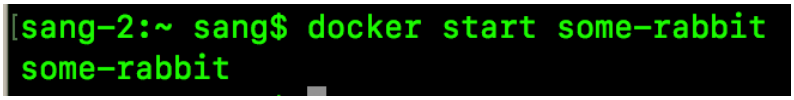
Spring Cloud Bus （消息总线）通过轻量级消息代理连接各个微服务，可以用来广播配置文件的更改或者服务监控的管理。在实际生产环境中，Spring Cloud Bus 主要是用来做微服务的监控或者微服务应用程序之间的通信，目前常见的实现方式是通过 AMQP 消息代理作为通道。

简单实践

首先我们先来启动 Docker 中安装的 RabbitMQ。如果你的 RabbitMQ 在上篇文章学习完之后，已经关闭了，那么本文不需要再运行 docker run 命令去启动 RabbitMQ 了，直接执行如下命令，启动已有的 docker 容器即可：

```
docker start some-rabbit
```

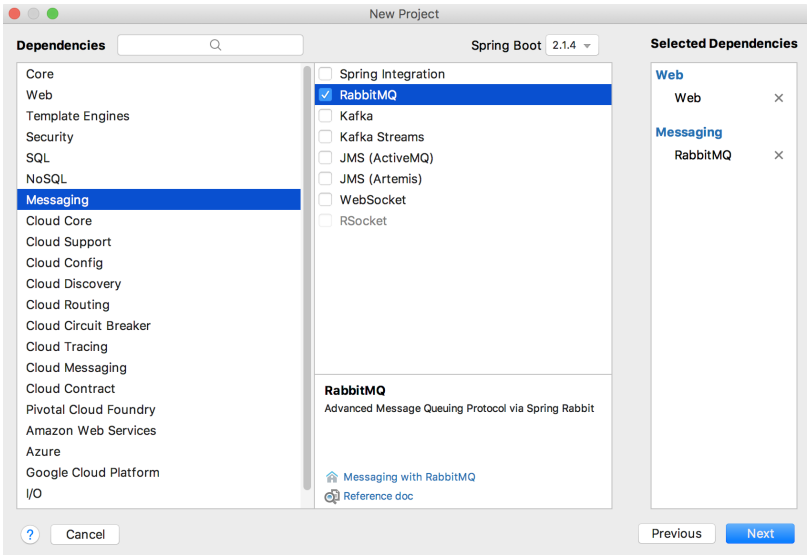
如下：



其中，some-rabbit 表示启动的容器名称，这样我们启动的是一个已有的 RabbitMQ 实例，相关参数和我们上文的都是一样的（如果需要再创建一个 RabbitMQ 容器，则可以继续执行上文的 docker run 命令，但是注意宿主机的端口、容器的名字不可以重复）。

容器启动成功之后，先通过一个简单的 Spring Boot 工程来和大家演示一下 RabbitMQ 消息的收发过程。

首先我们来创建一个名为 rabbitmq 的 Spring Boot 项目，创建时勾选两个依赖：Web 和 RabbitMQ ，如下：



工程创建完成后，我们首先在 application.properties 中配置一下 RabbitMQ 的基本信息，如下：

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

这个是 RabbitMQ 的基本连接信息，大家知道，这些信息将被注入到相应的 Bean 中，这里是注入到 RabbitProperties 对象中去，我们来看一点点这个对象的源码：

```
@ConfigurationProperties(prefix = "spring.rabbitmq")
public class RabbitProperties {

 /**
  * RabbitMQ host.
  */
 private String host = "localhost";

 /**
  * RabbitMQ port.
  */
 private int port = 5672;

 /**
  * Login user to authenticate to the broker.
  */
 private String username = "guest";

 /**
  * Login to authenticate against the broker.
  */
 private String password = "guest";

    //other
}
```

大家看到，这里每一项都有一个默认值，而且默认值我们写的也是一致的，所以，如果你的 RabbitMQ 的访问地址是本机地址，并且端口、用户名、密码都是默认的话，那么这里其实也可以不用配置。

在 RabbitMQ 中，所有的消息生产者提交的消息都会交由 Exchange 进行再分配，Exchange 会根据不同的策略将消息分发到不同的 Queue 中。 RabbitMQ 中一共提供了四种不同的 Exchange 策略，分别是 Direct 、 Fanout 、 Topic 以及 Header ，这四种不同的策略，前三种使用频率较高，第四种使用频率较低，下面分别对这四种不同的 Exchange Type 进行简单介绍。

Direct

DirectExchange 的路由策略是将消息队列绑定到一个 DirectExchange 上，当一条消息到达 DirectExchange 时会被转发到与该条消息 routing key 相同的 Queue 上，例如消息队列名为 “hello-queue” ，则 routing key 为 “hello-queue” 的消息会被该消息队列接收。DirectExchange 的配置如下：

```
@Configuration
public class RabbitDirectConfig {
    public final static String DIRECTNAME = "sang-direct";
    @Bean
    Queue queue() {
        return new Queue("hello-queue");
    }
    @Bean
    DirectExchange directExchange() {
        return new DirectExchange(DIRECTNAME, true, false);
    }
    @Bean
    Binding binding() {
        return BindingBuilder.bind(queue())
                .to(directExchange()).with("direct");
    }
}
```

代码解释：

1. 首先提供一个消息队列Queue，然后创建一个DirectExchange对象，三个参数分别是名字，重启后是否依然有效以及长期未用时是否删除；
2. 创建一个Binding对象将Exchange和Queue绑定在一起；
3. DirectExchange和Binding两个Bean的配置可以省略掉，即如果使用DirectExchange，可以只配置一个Queue的实例即可。

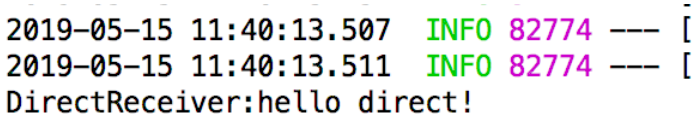
接下来配置一个消费者，如下：

```
@Component
public class DirectReceiver {
    @RabbitListener(queues = "hello-queue")
    public void handler1(String msg) {
        System.out.println("DirectReceiver:" + msg);
    }
}
```

通过 @RabbitListener 注解指定一个方法是一个消息消费方法，方法参数就是所接收到的消息。然后在单元测试类中注入一个 RabbitTemplate 对象来进行消息发送，如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitmqApplicationTests {
    @Autowired
    RabbitTemplate rabbitTemplate;
    @Test
    public void directTest() {
        rabbitTemplate.convertAndSend("hello-queue", "hello direct!");
    }
}
```

确认RabbitMQ已经启动，然后启动 Spring Boot 项目，启动成功后，运行该单元测试方法，在 Spring Boot 控制台打印日志如下图：



Fanout

FanoutExchange 的数据交换策略是把所有到达 FanoutExchange 的消息转发给所有与它绑定的 Queue 上，在这种策略中，routing key 将不起任何作用，FanoutExchange 配置方式如下：

```
@Configuration
public class RabbitFanoutConfig {
    public final static String FANOUTNAME = "sang-fanout";
    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange(FANOUTNAME, true, false);
    }
    @Bean
    Queue queueOne() {
        return new Queue("queue-one");
    }
    @Bean
    Queue queueTwo() {
        return new Queue("queue-two");
    }
    @Bean
    Binding bindingOne() {
        return BindingBuilder.bind(queueOne()).to(fanoutExchange());
    }
    @Bean
    Binding bindingTwo() {
        return BindingBuilder.bind(queueTwo()).to(fanoutExchange());
    }
}
```

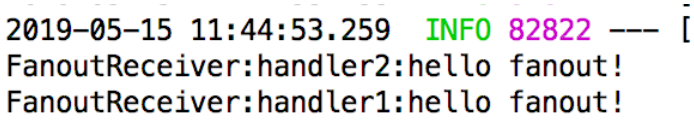
在这里首先创建 FanoutExchange ，参数含义与创建 DirectExchange 参数含义一致，然后创建两个 Queue ，再将这两个 Queue 都绑定到 FanoutExchange 上。接下来创建两个消费者，如下：

```
@Component
public class FanoutReceiver {
    @RabbitListener(queues = "queue-one")
    public void handler1(String message) {
        System.out.println("FanoutReceiver:handler1:" + message);
    }
    @RabbitListener(queues = "queue-two")
    public void handler2(String message) {
        System.out.println("FanoutReceiver:handler2:" + message);
    }
}
```

两个消费者分别消费两个消息队列中的消息，然后在单元测试中发送消息，如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitmqApplicationTests {
    @Autowired
    RabbitTemplate rabbitTemplate;
    @Test
    public void fanoutTest() {
        rabbitTemplate
        .convertAndSend(RabbitFanoutConfig.FANOUTNAME,
                null, "hello fanout!");
    }
}
```

注意这里发送消息时不需要 routing key ，指定 exchange 即可，routing key 可以直接传一个 null。
确认RabbitMQ已经启动，然后启动Spring Boot项目，启动成功后，执行单元测试方法，控制台打印日志如下图：



可以看到，一条消息发送出去之后，所有和该 FanoutExchange 绑定的 Queue 都收到了消息。

Topic

TopicExchange 是比较复杂但也是比较灵活的一种路由策略，在 TopicExchange 中，Queue 通过 routing key 绑定到 TopicExchange 上，当消息到达 TopicExchange 后，TopicExchange 根据消息的 routing key 将消息路由到一个或者多个 Queue上。TopicExchange 配置如下：

```
@Configuration
public class RabbitTopicConfig {
    public final static String TOPICNAME = "sang-topic";
    @Bean
    TopicExchange topicExchange() {
        return new TopicExchange(TOPICNAME, true, false);
    }
    @Bean
    Queue xiaomi() {
        return new Queue("xiaomi");
    }
    @Bean
    Queue huawei() {
        return new Queue("huawei");
    }
    @Bean
    Queue phone() {
        return new Queue("phone");
    }
    @Bean
    Binding xiaomiBinding() {
        return BindingBuilder.bind(xiaomi()).to(topicExchange())
                .with("xiaomi.#");
    }
    @Bean
    Binding huaweiBinding() {
        return BindingBuilder.bind(huawei()).to(topicExchange())
                .with("huawei.#");
    }
    @Bean
    Binding phoneBinding() {
        return BindingBuilder.bind(phone()).to(topicExchange())
                .with("#.phone.#");
    }
}
```

代码解释：

1. 首先创建 TopicExchange，参数和前面的一致。然后创建三个 Queue，第一个 Queue 用来存储和 “xiaomi” 有关的消息，第二个 Queue 用来存储和 “huawei” 有关的消息，第三个 Queue 用来存储和 “phone” 有关的消息；
2. 将三个 Queue 分别绑定到 TopicExchange 上，第一个 Binding 中的 “xiaomi.#” 表示消息的 routing key 凡是以 “xiaomi” 开头的，都将被路由到名称为 “xiaomi” 的 Queue 上；第二个 Binding 中的 “huawei.#” 表示消息的 routing key 凡是以 “huawei” 开头的，都将被路由到名称为 “huawei” 的 Queue 上；第三个 Binding 中的 “#.phone.#” 则表示消息的 routing key 中凡是包含 “phone” 的，都将被路由到名称为 “phone” 的 Queue 上。

接下来针对三个 Queue 创建三个消费者，如下：

```
@Component
public class TopicReceiver {
    @RabbitListener(queues = "phone")
    public void handler1(String message) {
        System.out.println("PhoneReceiver:" + message);
    }
    @RabbitListener(queues = "xiaomi")
    public void handler2(String message) {
        System.out.println("XiaoMiReceiver:"+message);
    }
    @RabbitListener(queues = "huawei")
    public void handler3(String message) {
        System.out.println("HuaWeiReceiver:"+message);
    }
}
```

然后在单元测试中进行消息的发送，如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitmqApplicationTests {
    @Autowired
    RabbitTemplate rabbitTemplate;
    @Test
    public void topicTest() {
        rabbitTemplate.convertAndSend(RabbitTopicConfig.TOPICNAME,
                "xiaomi.news","小米新闻..");
        rabbitTemplate.convertAndSend(RabbitTopicConfig.TOPICNAME,
                "huawei.news","华为新闻..");
        rabbitTemplate.convertAndSend(RabbitTopicConfig.TOPICNAME,
                "xiaomi.phone","小米手机..");
        rabbitTemplate.convertAndSend(RabbitTopicConfig.TOPICNAME,
                "huawei.phone","华为手机..");
        rabbitTemplate.convertAndSend(RabbitTopicConfig.TOPICNAME,
                "phone.news","手机新闻..");
    }
}
```

根据 RabbitTopicConfig 中的配置，第一条消息将被路由到名称为 “xiaomi” 的 Queue 上，第二条消息将被路由到名为 “huawei” 的 Queue 上，第三条消息将被路由到名为 “xiaomi” 以及名为 “phone” 的 Queue 上，第四条消息将被路由到名为 “huawei” 以及名为 “phone” 的 Queue 上，最后一条消息则将被路由到名为 “phone” 的 Queue 上。

确认 RabbitMQ 已经启动，然后启动 Spring Boot 项目，启动成功后，运行单元测试方法，控制台打印日志如下图：

```
2019-05-15 12:05:02.650  INFO 82947 --- [
XiaoMiReceiver:小米手机..
PhoneReceiver:华为手机..
HuaWeiReceiver:华为手机..
```

```
2019-05-15 12:04:52.800  INFO
PhoneReceiver:小米手机..
XiaoMiReceiver:小米新闻..
HuaWeiReceiver:华为新闻..
PhoneReceiver:手机新闻..
```

Header

HeadersExchange 是一种使用较少的路由策略，HeadersExchange 会根据消息的 Header 将消息路由到不同的 Queue 上，这种策略也和 routing key 无关，配置如下：

```
@Configuration
public class RabbitHeaderConfig {
    public final static String HEADERNAME = "sang-header";
    @Bean
    HeadersExchange headersExchange() {
        return new HeadersExchange(HEADERNAME, true, false);
    }
    @Bean
    Queue queueName() {
        return new Queue("name-queue");
    }
    @Bean
    Queue queueAge() {
        return new Queue("age-queue");
    }
    @Bean
    Binding bindingName() {
        Map<String, Object> map = new HashMap<>();
        map.put("name", "sang");
        return BindingBuilder.bind(queueName())
                .to(headersExchange()).whereAny(map).match();
    }
    @Bean
    Binding bindingAge() {
        return BindingBuilder.bind(queueAge())
                .to(headersExchange()).where("age").exists();
    }
}
```

这里的配置大部分和前面介绍的一样，差别主要体现的 Binding 的配置上。第一个 bindingName 方法中，whereAny 表示消息的 Header 中只要有一个 Header 匹配上 map 中的 key/value ，就把该消息路由到名为 “name-queue” 的 Queue 上。这里也可以使用 whereAll 方法，表示消息的所有 Header 都要匹配。whereAny 和 whereAll 实际上对应了一个名为 x-match 的属性。bindingAge 中的配置则表示只要消息的 Header 中包含 age ，不管 age 的值是多少，都将消息路由到名为 “age-queue” 的 Queue 上。

接下来创建两个消息消费者：

```
@Component
public class HeaderReceiver {
    @RabbitListener(queues = "name-queue")
    public void handler1(byte[] msg) {
        System.out.println("HeaderReceiver:name:"
                + new String(msg, 0, msg.length));
    }
    @RabbitListener(queues = "age-queue")
    public void handler2(byte[] msg) {
        System.out.println("HeaderReceiver:age:"
                + new String(msg, 0, msg.length));
    }
}
```

注意这里的参数用 byte 数组接收。然后在单元测试中创建消息的发送方法，这里消息的发送也和 routing key 无关，如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitmqApplicationTests {
    @Autowired
    RabbitTemplate rabbitTemplate;
    @Test
    public void headerTest() {
        Message nameMsg = MessageBuilder
                .withBody("hello header! name-queue".getBytes())
                .setHeader("name", "sang").build();
        Message ageMsg = MessageBuilder
                .withBody("hello header! age-queue".getBytes())
                .setHeader("age", "99").build();
        rabbitTemplate.send(RabbitHeaderConfig.HEADERNAME, null, ageMsg);
        rabbitTemplate.send(RabbitHeaderConfig.HEADERNAME, null, nameMsg);
    }
}
```

这里创建两条消息，两条消息具有不同的 header ，不同 header 的消息将被发到不同的 Queue 中去。

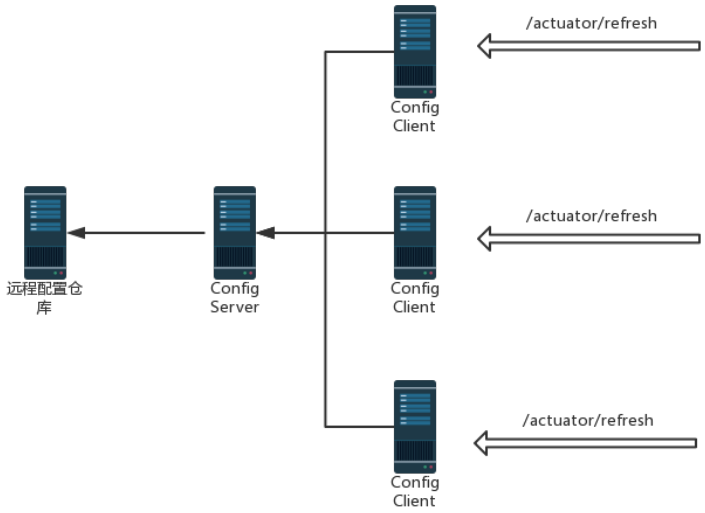
确认 RabbitMQ 已经启动，然后启动 Spring Boot 项目，启动成功后，执行单元测试方法，结果如下图：

```
2019-05-15 12:13:31.056  INFO 83021 --- [
HeaderReceiver:name:hello header! name-queue
HeaderReceiver:age:hello header! age-queue
```

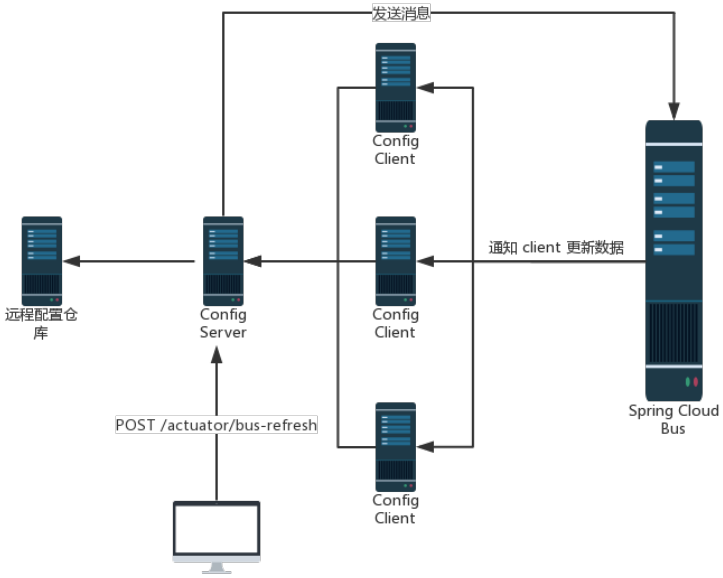
好了，上面这是和大家分享一下 RabbitMQ 的基本用法 。

动态刷新配置

使用 Spring Cloud Bus 我们可以轻松实现配置文件的动态刷新，在使用 Spring Cloud Bus 之前，我们动态刷新配置文件大致的架构图如下：



可以看到，当配置文件发生变化时，我们需要挨个向 Config Client 发送 `/actuator/refresh` 请求，才能实现 Config Client 上配置文件的动态刷新，这种操作显然很麻烦很费事，结合 Spring Cloud Bus ，我们可以对这个图做进一步的优化，如下：



可以看到，当引入 Spring Cloud Bus 之后，当我们配置文件发生变化时，我们可以指向 Config Server 发送一条更新请求，再由 Config Server 给 Spring Cloud Bus 发送消息；Spring Cloud Bus 收到消息之后，再去自动通知 Config Client 去完成数据更新。在整个过程中，开发者只需要向 Config Server 发送一条消息即可，很明显，这种方式的效率比我们之前动态刷新配置的效率要高很多，接下来我们就来看下这个东西要怎么实现。

这种更新方式实际上分为两种策略，一种是 Spring Cloud Bus 通知所有的 Config Client 更新配置文件，另外一种则是 Spring Cloud Bus 通知部分 Config Client 更新配置文件（由于配置仓库中保存了很多 Config Client 的配置数据，有的时候配置文件发生变化，只是某一个 Config Client 的配置发生变化，这种情况下就没有必要通知所有的 Config Client去更新数据），两种更新方式也有略微的差别，下面我来分别介绍。

批量刷新

首先我们需要搭建 Spring Cloud Config 环境，这里简单起见，我就不重复搭建了，直接在 8-3 小节的基础上来完成。

首先我们需要在 config_server 和 config_client 两个模块上分别添加 Spring Cloud Bus 相关的依赖，如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

添加完成后，再分别给 config_client 和 config_server 模块配置 RabbitMQ，配置信息如下：

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

由于我们的 config_server 一会儿将提供 /actuator/bus-refresh 接口，因此我们需要配置让这个端口暴露出来，如下：

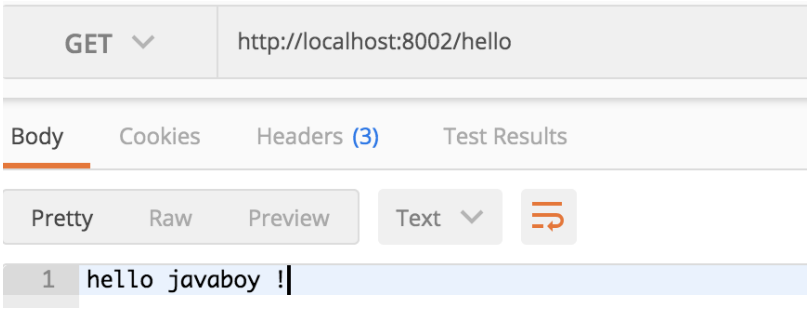
```
management.endpoints.web.exposure.include=bus-refresh
```

同时，由于我们给 config_server 中的所有接口添加了保护，因此 /actuator/bus-refresh 是无法直接访问的，我再添加一个 Spring Security 的配置类，在配置类中对权限再做一些配置，如下：

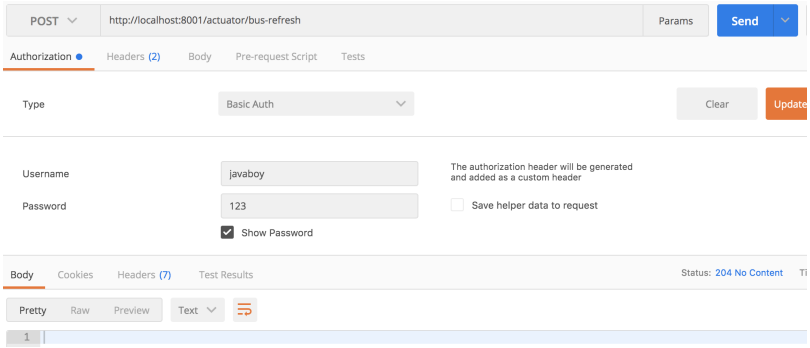
```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .anyRequest().authenticated()
                .and()
                .httpBasic()
                .and()
                .csrf().disable();
    }
}
```

注意，这里的配置首先是配置所有的请求都必须登录后才能访问，然后配置允许 **HttpBasic** 登录，这样我们在发起 **/actuator/bus-refresh** 请求时，就可以直接通过 **HttpBaisc** 来配置认证信息了。

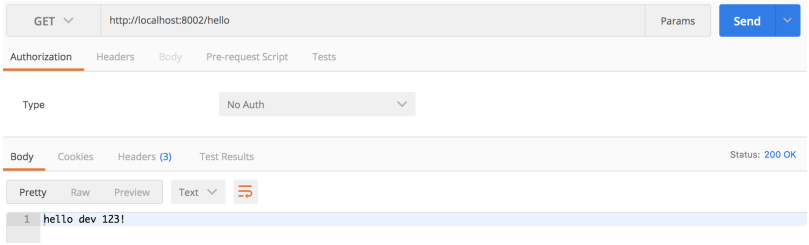
配置完成后，分别启动 eureka、config_server 以及 config_client，访问 config_client 的 /hello 接口，结果如下：



此时，我们修改配置文件，提交到远程仓库，然后向 config_server 发送一个 POST 请求，如下：



注意这个请求，我们设置了 Authorization 的方式为 Basic Auth ，然后填入我们的用户名密码信息，再发送 POST 请求，否则请求响应码为 401 。请求成功之后，我们再次访问 config_client 的 /hello 接口，发现数据已经发生变化了。



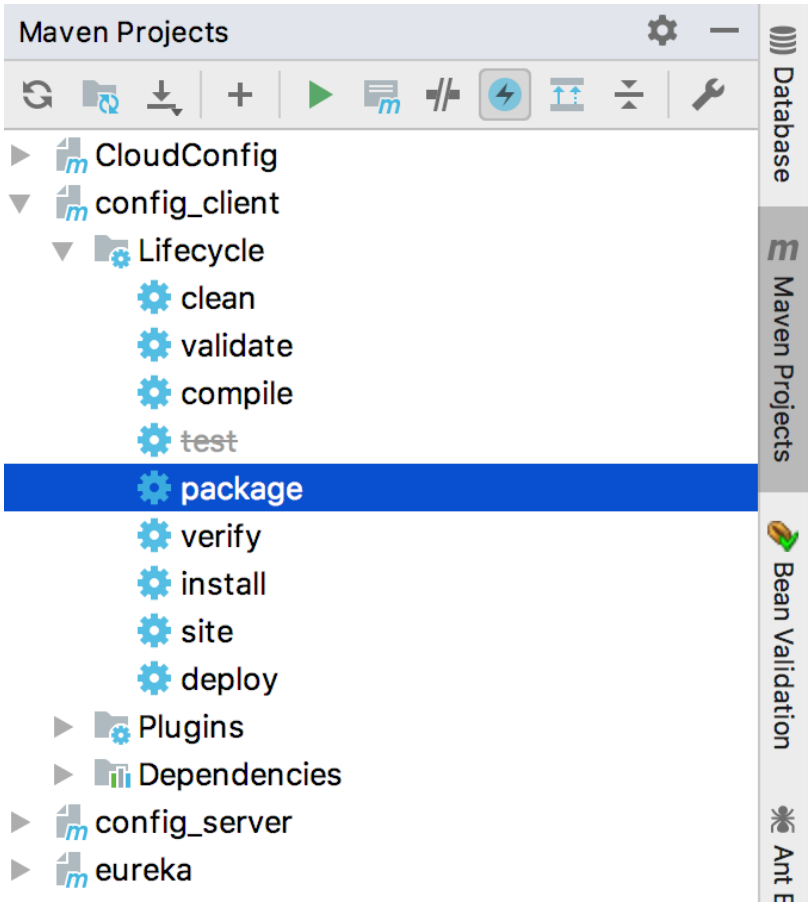
这种方式，所有的 config_client 都会收到 Spring Cloud Bus 的消息，然后去更新自身的数据，但有的时候我们可能只需要某一部分 config_client 更新数据，其它的不更新数据，那么这种需求该如何处理呢？

逐个刷新

首先我们来对 config_client 做一点点改造，给每一个 config_client 实例取一个 instance-id ，添加如下配置即可：

```
eureka.instance.instance-id=${spring.application.name}:${server.port}
```

这行配置表示 config_client 的实例 id 是由 服务名:端口 组成，配置完成后，点击 IDEA 右边的 Maven Project 对 config_client 进行打包，如下：



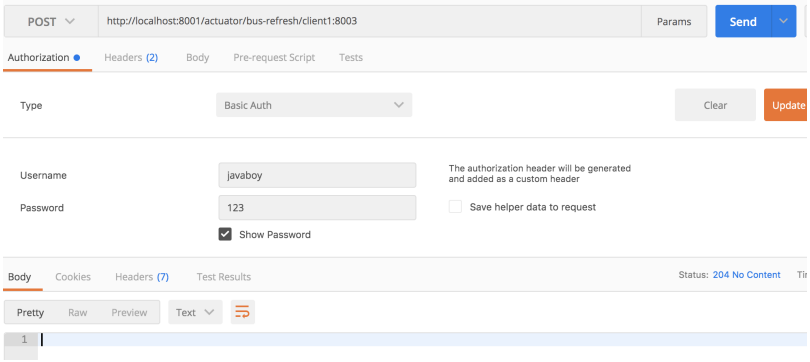
打包完成后，进入到 target 目录下，执行如下命令先启动一个 config_client 实例：

```
java -jar config_client-0.0.1-SNAPSHOT.jar --server.port=8002
```

然后换个端口再启动一个 config_client 实例：

```
java -jar config_client-0.0.1-SNAPSHOT.jar --server.port=8003
```

两个实例都启动之后，它们的 instance-id 是不一样的，一个是 client1-8002 ，另外一个是 client1-8003 ，接下来我们再次更新配置文件并且上传到远程仓库，然后给 config_server 发送请求时，像下面这样去发送：



现在的请求地址变为了 `http://localhost:8001/actuator/bus-refresh/client1:8003` ，最后面的地址就是指 config_client 的 id，这个表示只发送更新通知给 instance-id 为 client1:8003 的 config_client，其它的 config_client 将不会收到配置文件更新通知。

当这个 POST 请求发送成功之后，我们刷新端口为 8002 的 config_client 发现没有什么变化，再去刷新端口为 8003 的 config_client，发现数据已经更新了。

小结

本文主要和大家聊了聊 RabbitMQ 的基本用法以及利用 Spring Cloud Bus 实现配置文件的动态刷新，相比前面第 8 章学到的配置文件动态刷新方式，这种动态刷新方式效率更高。那么这就是最好的方案吗？其实不见得，后面我们还会向大家介绍 Spring Cloud Alibaba 中的相关组件，可以让大家感受到更加丝滑的配置文件刷新。

25 Docker 简介与消息中间件安装

27 构建消息驱动的微服务

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