链滴

rpush：多平台统一消息推送系统

作者：shuangmulin45

原文链接：https://ld246.com/article/1629807381870

来源网站：链滴

gitee代码传送

一个接口触达多平台（包括微信公众号、企业微信、钉钉、邮箱等任何想的到平台，都能一个接口一推送；极简的代码调用，极大减少业务方消息推送的代码量）；

同时提供基于netty和websocket的即时通讯实现，实现单聊、群聊等功能。开箱即用，采用SpringCloud微服务架构，扩展简单且没有单点问题。致力于包揽所有和消息推送有关的技术开发工作，节省开资源。

- 一个接口触达多平台，支持一个接口多平台同时发送
- 消息平台逻辑与业务逻辑的解耦，业务方不需要关心各个平台的对接实现，只需要关心：要用些平台发、要发给对应平台的哪些人、要发什么内容
- 极强的扩展性，要新增一个消息平台的支持，理论只需要新增几个类就能完成，且不需要写任前端代码即可获得该平台对应的ui交互（包括：配置交互、接收人维护、web手动消息发送交互等）。
- 当然支持web端手动发送消息
- 当然也支持定时任务
- 消息方案预设置
- 提供即时通讯实现，且支持服务器横向扩展
- 接收人导入
- 接收人按分组划分
- 消息日志
- ...

在线体验

http://159.75.121.163/

admin admin

目前支持的消息类型

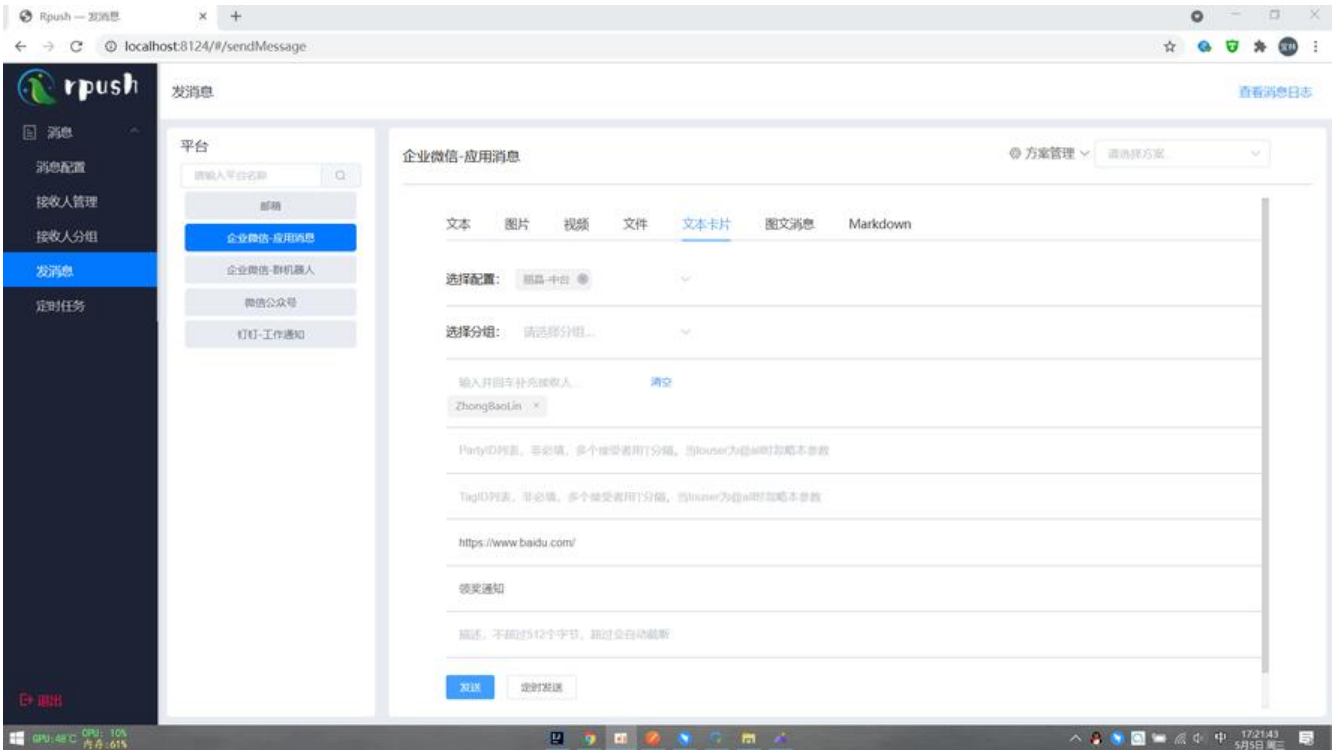
- 邮箱
- 企业微信-应用消息
  - 文本消息
  - 图片消息
  - 视频消息
  - 文本消息
  - 文本卡片消息
  - 图文消息
  - Markdown消息
- 企业微信-群机器人
  - 文本消息
  - 图片消息

- 图文消息
    - Markdown消息
  - 微信公众号
    - 文本消息
    - 图文消息
    - 模板消息
  - 钉钉-工作通知
    - 文本
    - Markdown
    - 链接消息
    - 卡片消息
    - OA消息
  - 钉钉-群机器人
    - 文本
    - Markdown
    - 链接消息
    - 卡片消息
    - FeedCard

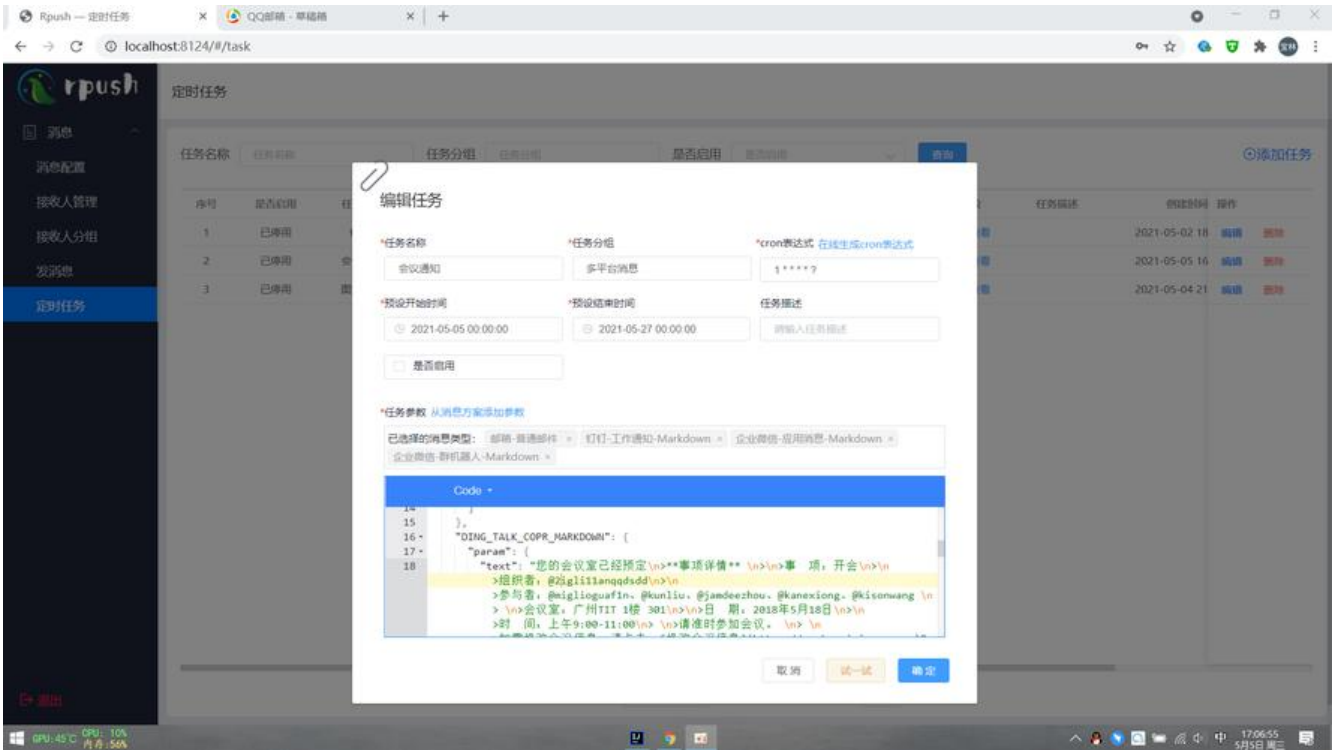
Rpush的架构决定了扩展一个消息平台的消息类型会非常简单，所以如果要扩展一个消息平台，大部时间都会花在查找该平台的对接文档上。后续会在工作之余加上其它的平台或消息类型。当然，欢迎与扩展（扩展一个消息平台的消息类型，只需要几个java类即可，不需要写任何前端代码，即可获得括ui交互内的所有功能）。

效果展示

单个消息类型发送示例

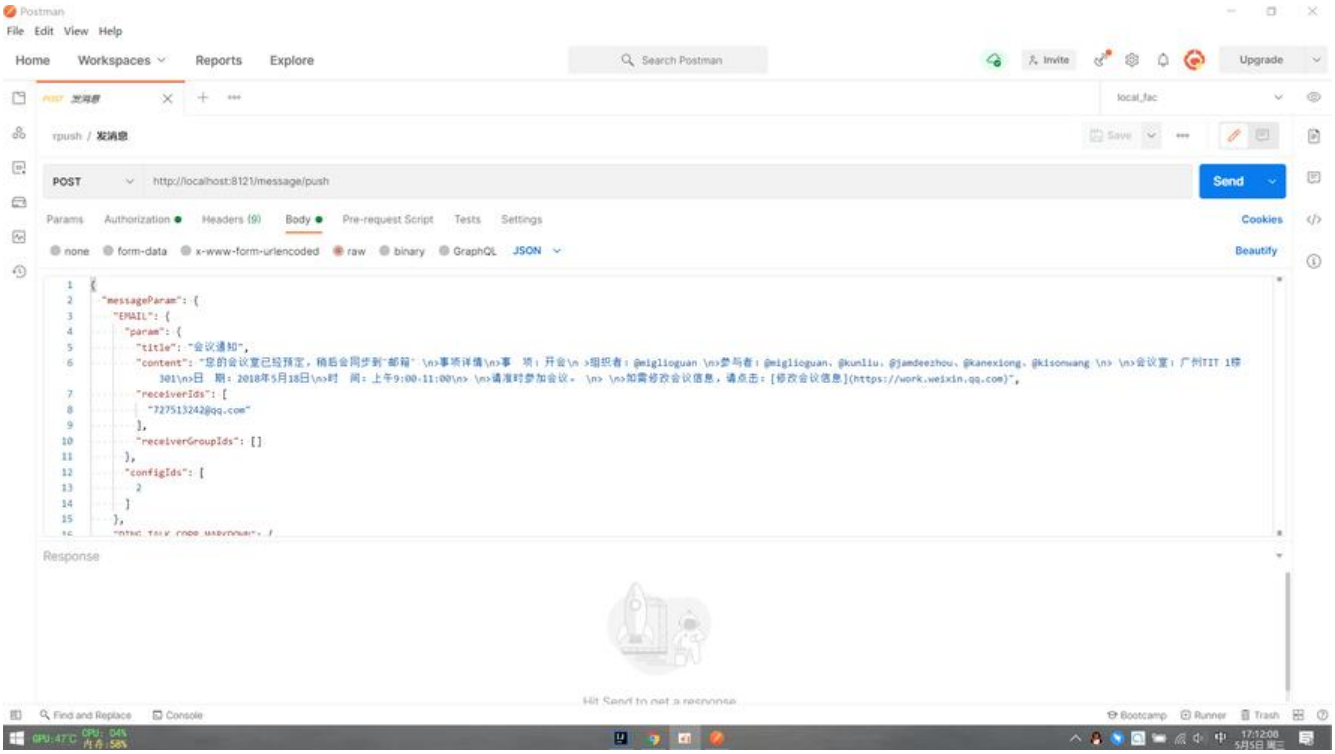


web端多平台发送示例



postman多平台发送示例

原文链接：rpush：多平台统一消息推送系统



用代码发消息

秉持”业务服务只负责发消息“的解耦原则，业务服务在需要发消息的时候，代码应该越简单越好。以，Rpush的发消息的sdk，一种消息只需要一行代码，有几种消息就有几行代码。比如这样：

```
/**
 * @author shuangmulin
 * @since 2021/6/8/008 11:37
 **/
public class RpushSenderTest {
    /**
     * 要发送的内容
     */
    public static final String content = "您的会议室已经预定 \n" +
            ">**事项详情** \n" +
            ">事　项：开会\n" +
            ">组织者：@miglioguan \n" +
            ">参与者：@miglioguan、@kunliu、@jamdeezhou、@kanexiong、@kisonwang \n" +
            "> \n" +
            ">会议室：广州TIT 1楼 301\n" +
            ">日　期：2021年5月18日\n" +
            ">时　间：上午9:00-11:00\n" +
            "> \n" +
            ">请准时参加会议。 \n" +
            "> \n" +
            ">如需修改会议信息，请点击：[修改会议信息](https://work.weixin.qq.com)";

    public static void main(String[] args) {
        // 企业微信-markdown消息
        MarkdownMessageDTO markdown = RpushMessage.WECHAT_WORK_AGENT_MARKDWN().content(content).receiverIds(Collections.singletonList("ZhongBaoLin")).build();
        // 企业微信-群机器人消息
```

原文链接：rpush：多平台统一消息推送系统

```
        TextMessageDTO text = RpushMessage.WECHAT_WORK_ROBOT_TEXT().content(content
.receiverIds(Collections.singletonList("ZhongBaoLin")).build();
        // 邮箱
        EmailMessageDTO email = RpushMessage.EMAIL().title("会议通知").content(content).buil
();
        RpushService.instance("baolin", "666666").sendMessage(markdown, text, email); // 填上
号密码，运行即可
    }
}
```

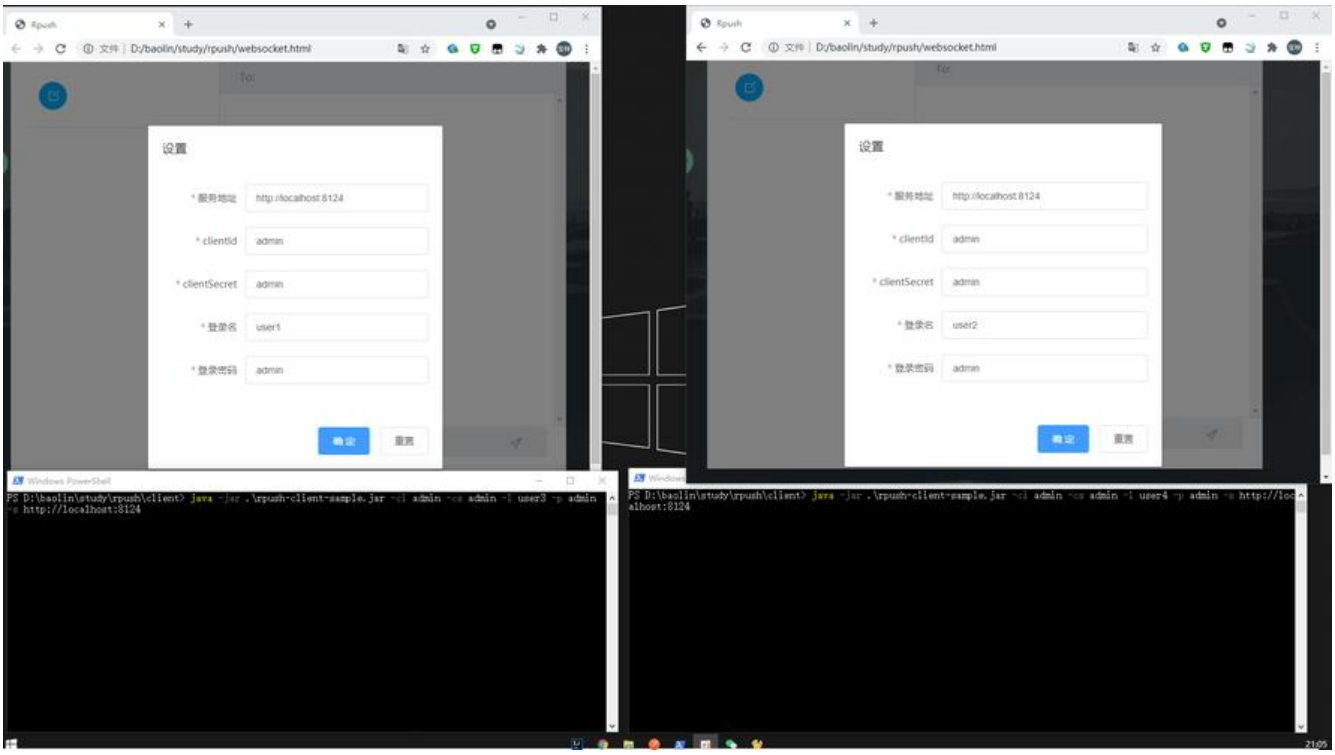
以上代码，一次发送了三种不同平台的不同消息类型，全部代码加起来也只需要四五行代码而已。要得以上效果，只需要maven引用rpush的sdk模块即可：

```
<project>
    <!-- 设置 jitpack.io 仓库 -->
    <repositories>
        <repository>
            <id>jitpack.io</id>
            <url>https://jitpack.io</url>
        </repository>
    </repositories>

    <dependencies>
        <!-- 添加rpush-sdk依赖 -->
        <dependency>
            <groupId>com.github.shuangmulin.rpush</groupId>
            <artifactId>rpush-sdk</artifactId>
            <version>v1.0.2</version>
        </dependency>
    </dependencies>
</project>
```

即时通讯

Rpush对即时通讯的实现方式比较包容，即对具体的连接实现做了解耦，不局限于某一种连接方式，可以netty，可以websocket，可以comt，当然也可以用原始的bio来做。这里展示websocke的网页端和netty实现的命令行客户端之间互相聊和群聊的效果（该示例的相关代码：客户端示例代码地址）：



一些比较核心的扩展点

1. 可自由扩展的消息平台和消息类型

在Rpush的设计里，消息被归类为“消息平台”和“消息类型”，分别对应如下两个枚举：

```
/**
 * 消息平台枚举
 **/
public enum MessagePlatformEnum {

    EMAIL(EmailConfig.class, "邮箱", "", "^[_a-z0-9-]+(\\.[_a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]+)
(\\.[a-z]{2,})$", true),
    WECHAT_WORK_AGENT(WechatWorkAgentConfig.class, "企业微信-应用消息", "", "", true),
    WECHAT_WORK_ROBOT(WechatWorkRobotConfig.class, "企业微信-群机器人", "", "", true),
    WECHAT_OFFICIAL_ACCOUNT(WechatOfficialAccountConfig.class, "微信公众号", "", "", true),
    DING_TALK_CORP(DingTalkCorpConfig.class, "钉钉-工作通知", "", "", true),
    RPUSH_SERVER(EmptyConfig.class, "rpush服务", "", "", true);
}

/**
 * 消息类型枚举
 **/
public enum MessageType {

    EMAIL("普通邮件 ", MessagePlatformEnum.EMAIL),
    RPUSH_SERVER("文本", MessagePlatformEnum.RPUSH_SERVER),

    // ==================================企业微信-应用==================
==================
    WECHAT_WORK_AGENT_TEXT("文本", MessagePlatformEnum.WECHAT_WORK_AGENT),
```

```
    WECHAT_WORK_AGENT_IMAGE("图片", MessagePlatformEnum.WECHAT_WORK_AGENT),
    WECHAT_WORK_AGENT_VIDEO("视频", MessagePlatformEnum.WECHAT_WORK_AGENT),
    WECHAT_WORK_AGENT_FILE("文件", MessagePlatformEnum.WECHAT_WORK_AGENT),
    WECHAT_WORK_AGENT_TEXTCARD("文本卡片", MessagePlatformEnum.WECHAT_WORK_AGENT),
    WECHAT_WORK_AGENT_NEWS("图文消息", MessagePlatformEnum.WECHAT_WORK_AGENT,
    WECHAT_WORK_AGENT_MARKDOWN("Markdown", MessagePlatformEnum.WECHAT_WORK_AGENT),

    // ==================================企业微信-群机器人==========================
    WECHAT_WORK_ROBOT_TEXT("文本", MessagePlatformEnum.WECHAT_WORK_ROBOT),
    WECHAT_WORK_ROBOT_IMAGE("图片", MessagePlatformEnum.WECHAT_WORK_ROBOT),
    WECHAT_WORK_ROBOT_NEWS("图文消息", MessagePlatformEnum.WECHAT_WORK_ROBOT),
    WECHAT_WORK_ROBOT_MARKDOWN("Markdown", MessagePlatformEnum.WECHAT_WORK_ROBOT),

    // ==================================微信公众号==================================
    WECHAT_OFFICIAL_ACCOUNT_TEXT("文本", MessagePlatformEnum.WECHAT_OFFICIAL_ACCOUNT),
    WECHAT_OFFICIAL_ACCOUNT_NEWS("图文消息", MessagePlatformEnum.WECHAT_OFFICIAL_ACCOUNT),
    WECHAT_OFFICIAL_ACCOUNT_TEMPLATE("模板消息", MessagePlatformEnum.WECHAT_OFFICIAL_ACCOUNT),

    // ==================================钉钉-工作通知==================================
    DING_TALK_COPR_TEXT("文本", MessagePlatformEnum.DING_TALK_CORP),
    DING_TALK_COPR_MARKDOWN("Markdown", MessagePlatformEnum.DING_TALK_CORP),
    DING_TALK_COPR_LINK("链接消息", MessagePlatformEnum.DING_TALK_CORP),
    DING_TALK_COPR_ACTION_CARD_SINGLE("卡片-单按钮", MessagePlatformEnum.DING_TAL_CORP),
    DING_TALK_COPR_ACTION_CARD_MULTI("卡片-多按钮", MessagePlatformEnum.DING_TALKCORP),
    DING_TALK_COPR_OA("OA消息", MessagePlatformEnum.DING_TALK_CORP),

    ;
}
```

这里拿“企业微信-应用的文本类型”的消息举例。假设现在要在Rpush实现这个类型的消息，步骤下：

1. 定义企业微信的配置类，如下：

```
/**
 * 企业微信配置
 **/
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
@AllArgsConstructor
```

```
@Builder
public class WechatWorkAgentConfig extends Config {
    private static final long serialVersionUID = -9206902816158196669L;

    @ConfigValue(value = "企业ID", description = "在此页面查看：https://work.weixin.qq.com/ework_admin/frame#profile")
    private String corpId;
    @ConfigValue(value = "应用Secret")
    private String secret;
    @ConfigValue(value = "应用agentId")
    private Integer agentId;

}
```

里面的字段就按对应平台需要的字段去定义就行，比如这里的企业微信就只有三个字段需要配置。而个字段上的@ConfigValue

注解，是用来自动生成页面的，也就是说，只需要打上这个注解，就可以自动在页面上生成对应的增改查的界面和交互（无需写一行前端代码）。

2. 在 MessagePlatformEnum和MessageType

里加上对应的枚举，即WECHAT_WORK_AGENT(WechatWorkAgentConfig.class, "企业微信-应用息", "", "", true)

和WECHAT_WORK_AGENT_TEXT("文本", MessagePlatformEnum.WECHAT_WORK_AGENT),。里要注意下平台枚举的第一个参数就是第一步定义的配置类的Class。

3. 定义企业微信-应用-文本消息的参数，如下：

```
/**
 * 企业微信消息发送DTO
 **/
@EqualsAndHashCode(callSuper = true)
@Data
@SuperBuilder
@NoArgsConstructor
@AllArgsConstructor
public class TextMessageDTO extends BaseMessage {
    private static final long serialVersionUID = -3289428483627765265L;

    /**
     * 接收人分组列表
     */
    @SchemeValue(type = SchemeValueType.RECEIVER_GROUP)
    private List<Long> receiverGroupIds;

    /**
     * 接收人列表
     */
    @SchemeValue(type = SchemeValueType.RECEIVER)
    private List<String> receiverIds;

    @SchemeValue(description = "PartyID列表，非必填，多个接受者用‘|’分隔。当touser为@al时忽略本参数")
    private String toParty;
```

```
    @SchemeValue(description = "TagID列表，非必填，多个接受者用‘|’分隔。当touser为@all忽略本参数")
    private String toTag;

    @SchemeValue(type = SchemeValueType.TEXTAREA, description = "请输入内容...")
    private String content;

}
```

同样的，里面的字段根据该消息类型需要的字段去定义就行。比如企业微信-应用-文本消息就只需要个content内容字段以及接收人相关的字段。这里涉及到的@SchemeValue

注解，同样也是用来自动生成页面交互的，即只需要打上这个注解，就能自动在发消息页面生成对应ui和交互。同时可以使用com.regent.rpush.route.utils.sdk.SdkGenerator类自动生成sdk代码。

4. 实现 com.regent.rpush.route.handler.MessageHandler接口，正式写发消息的代码。

```
/**
 * 企业微信文本消息handler
 **/
@Component
public class AgentTextMessageHandler extends MessageHandler<TextMessageDTO> {

    @Override
    public MessageType messageType() {
        return MessageType.WECHAT_WORK_AGENT_TEXT;
    }

    @Override
    public void handle(TextMessageDTO param) {
        // 具体的发消息代码
    }
}
```

这里有以下需要关心的点：

- 接口上的泛型填第3步定义的类
- 实现 messageType方法，返回当前类要处理的消息类型
- 实现 handle方法，写发消息的代码，里面的参数是自动解析到这个方法的，直接使用即可

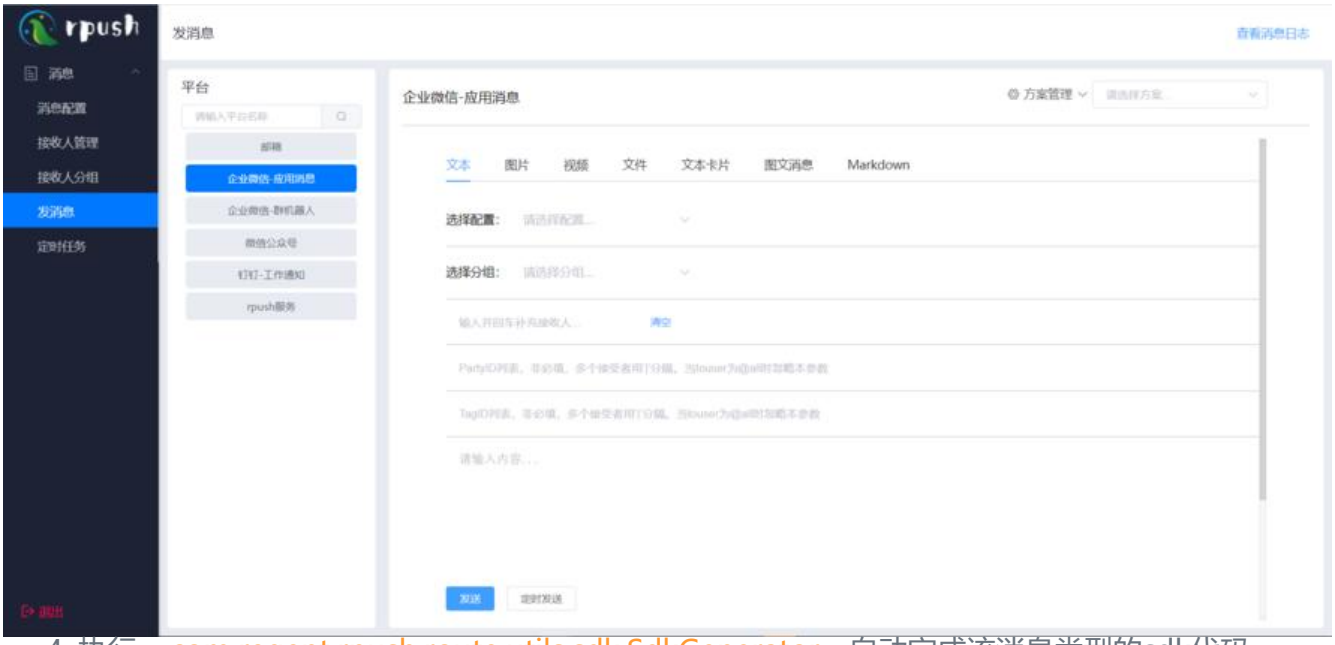
到这里，就不需要多做任何其它的事了。也就是说，做完以上四个步骤，就已经完成了一个消息类型扩展。事做的少，获得的功能并不少：

1. 自动获得对应平台配置的增删改交互和ui

原文链接：rpush：多平台统一消息推送系统

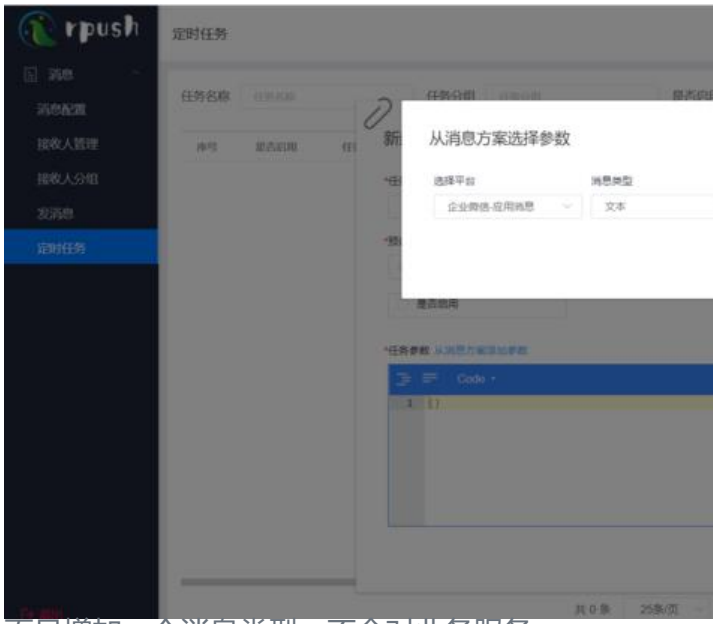
2. 自动获得对应平台接收人和接收人分组的增删改交互和ui（包括导入功能）

3. 自动获得该消息类型手动发送消息的交互和ui



4. 执行 com.regent.rpush.route.utils.sdk.SdkGenerator，自动完成该消息类型的sdk代码

5. 自动获得该消息类型的定时任务的增删改交互和ui



而且增加一个消息类型，不会对业务服务前正在使用的消息类型有任何影响，是纯粹的叠加”能力“。

2. 可自由扩展的即时通讯实现

在rpush的架构里，投递一个消息的流程大致可以概括为：调用统一的接口向路由服务投递消息-》路服务查出消息目标所在的服务器地址-》路由服务向对应的服务器传递消息-》对应的服务找到对应的话发送消息。

这里的扩展点在最后一步，即用户和服务器的会话维护。要实现服务端向客户端推送消息，会有比较的解决方案，比如用netty起一个nio服务器，客户端去连netty服务器或者服务端用socketio提供websocket实现，客户端按websocket的方式连服务器或者用comet实现长连接让客户端连等等。

这里做成扩展点的一个比较重要的考虑点就是要实现不同端之间的消息通信，比如上面例子里的命令和网页之间的聊天，或者实现移动端和网页之间的聊天。

RpushClient

不管是什么技术实现的服务端推送，都会有一个“客户端”性质的类，比如netty会有Channel，socktio提供的websocket会有SocketIOClient

。而对于rpush来说，只关心它们的一个共有的能力：消息投递。即RpushClient接口：

```
/**
 * 客户端
 **/
public interface RpushClient {

    /**
     * 推送消息
     */
    void pushMessage(NormalMessageDTO message);

    void close();
}
```

只要实现了这个接口的类，不管是什么技术的实现，都被认为是rpush的客户端。也就是说，netty也，websocket也好，只要提供给rpush这个接口的能力即可，从而达到解耦具体实现的目的。

目前rpush已经做了netty和socketio两个实现，分别对应com.regent.rpush.server.socket.nio.NioSocketChannelClient

和com.regent.rpush.server.socket.websocket.WebSocketClient两个类。

netty客户端sdk

rpush提供了netty对应的客户端的sdk，项目依赖rpush-client即可，使用也非常简单，只需要几行码即可。

```
public class Main {
    public static void main(String[] args) {
        RpushClient rpushClient = new RpushClient(servicePath, registrationId); // 填上rpush服地址和id
        rpushClient.addMsgProcessor(new PingIgnoreMsgProcessor()); // 忽略心跳消息
        rpushClient.start(); // 向服务端发起连接
        rpushClient.addMsgProcessor(msg -> {
            // 处理接收到的消息
            return false;
```

```
        });
    }
}
```

关于架构

rpush目前主要提供两大功能，一个是消息分发，另一个是即时通讯功能。消息分发由路由服务rpush-oute提供，即时通讯的长连接维护由socket服务rpush-server服务提供。

1. 可自由集群的路由服务

为了保证消息投递的统一性以及解耦消息分发和即时通讯之间的关系，路由服务只做一件事，即负责消息分发到各个平台，也就是说rpush提供的即时通讯功能，对路由服务来说和其他第三方平台没什区别，都被视为一个平台。

所以在架构层面，如果只需要用到消息分发的功能，就不需要部署rpush-server

服务，只需要部署eureka、zuul和路由服务即可。zuul作为系统对外的入口，隔离掉了路由服务器和户端，同时路由服务又是无状态的，这样就使得路由服务可以根据实际业务情况自由集群，即想加一路由服务就加，想减一台就减。

2. 可自由集群的socket服务

rpush-server作为socket服务，主要功能就是维护客户端的长连接。这个服务的承载能力直接决定了时通讯功能一次可以在线多少用户，所以这个服务毫无疑问必须要是可集群部署的。

假设部署5台socket服务，都正常配置eureka为注册中心。 为了实现socket服务的集群，一个客户端接rpush服务的流程为：

1. 客户端问路由服务要一个可用的socket服务器ip和端口
2. 路由服务通过合适的负载均衡算法得到一个可用的socket服务器ip和端口并返回给客户端
3. 客户端向拿到的socket服务发起长连接
4. 连接成功后，对应的socket服务器维护服务级别的session信息，然后向路由服务汇报该客户端路由服务保存该客户端和socket服务的对应关系
5. 客户端与对应的socket服务保持一定频率的心跳，并在心跳失败判定连接断开后重新发起以上程，直到再次连接成功

客户端基于以上步骤上线之后，其他客户端向该客户端投递消息的流程为：

1. 客户端请求路由服务提供的消息投递接口（这个就是前面说到的消息投递接口，因为socket服维护的长连接对路由服务来说和其他第三方平台没什么区别，所以消息投递的方式也是一样的）
2. 路由服务实现的socket服务消息处理器（ com.regent.rpush.route.handler.RpushMessageandler），根据消息上的目标客户端id找到对应的socket服务，并向该服务投递消息
3. socket服务从自己维护的session里找到目标客户端，最终完成消息投递

实现以上流程之后，socket服务就可以做到自由集群了。

上面说的流程偏理论化，有几个技术实现点这里做一下详细说明：

1. 路由服务如何通过合适的负载均衡算法得到一个可用的socket服务器ip和端口？

实现的手段其实非常的简单暴力。首先由socket服务提供一个查询本机ip和端口的接口，路由服务直通过ribbon去请求这个接口，然后自定义一个负载均衡规则类，来实现socket服务的选择：

```
/**
 * 路由->Socket服务端请求的实例选择
 */
public class ServerBalancer extends ZoneAvoidanceRule {

    @Override
    public Server choose(Object o) {
        // ...
        // 用默认的负载均衡算法选出一个可用的socket服务（这里的算法可以根据实际业务更改）
        return super.choose(o);
        // ...
    }

}
```

在配置文件里配置这个“规则类”：

```
rpush-server:
  ribbon:
    NFLoadBalancerRuleClassName: com.regent.rpush.route.loadbalancer.ServerBalancer
```

路由服务在向socket服务请求的时候会”经过“这个”规则类“，然后由这个“规则类”来选出一个用的socket服务。最终socket服务的端口和ip信息，也是由选中的socket服务通过这次请求返回给路服务的。

当然这个规则类不是只做这一件事，还有一个问题也需要这个类来完成。

2. 消息投递的时候，路由服务如何根据消息上的目标客户端id找到对应的socket服务？

首先，在客户端与某一个socket服务连接成之后，客户端与socket服务之间的关系需要保存起来（mysql或redis）。然后新增一个feign的请求拦截器（com.regent.rpush.route.loadbalancer.MessageRequestInterceptor
）：

```
@Component
public class MessageRequestInterceptor implements RequestInterceptor {

    /**
     * 存放本次消息投递的目标socket服务id
     */
    static final ThreadLocal<String> SERVER_ID = new ThreadLocal<>();

    @Autowired
    private IRpushServerOnlineService rpushServerOnlineService;

    @SuppressWarnings("MismatchedQueryAndUpdateOfCollection")
    @Override
    public void apply(RequestTemplate requestTemplate) {
        String url = requestTemplate.url();
        String method = requestTemplate.method();
        if (!"/push".equals(url) || !"POST".equals(method)) {
```

```
            // 只处理消息投递接口
            return;
        }

        // 如果是消息推送，需要给接收端连接的服务端投放消息，在服务端集群的情况下，要找到对
的服务端
        String body = new String(requestTemplate.body());
        JSONObject jsonObject = new JSONObject(body);
        String sendTo = jsonObject.getStr("sendTo"); // 拿到目标客户端的id
        String serverId = ""; // 从redis或mysql查到该客户端对应的socket服务id
        SERVER_ID.set(serverId); // 添加到当前线程里
    }

}
```

这个“拦截类”配合上面的”规则类“，就能在路由服务向socket服务传递消息时准确的找到对应的s
cket服务。 完整的”规则类“：

```
/**
 * 路由->Socket服务端请求的实例选择
 */
public class ServerBalancer extends ZoneAvoidanceRule {

    @Override
    public Server choose(Object o) {
        try {
            // 从拦截类里看有没有指定服务端实例
            String serverId = MessageRequestInterceptor.SERVER_ID.get();
            if (StringUtils.isEmpty(serverId)) {
                // 如果没有指定服务端实例，用默认的负载均衡算法
                return super.choose(o);
            }
            // 如果指定了服务端实例，说明是消息传递，用指定好的实例向socket服务发请求
            List<Server> servers = getLoadBalancer().getAllServers();
            for (Server server : servers) {
                if (StringUtils.equals(server.getId(), serverId)) {
                    return server;
                }
            }
            throw new IllegalArgumentException("没有可用的RPUSH_SERVER实例");
        } finally {
            MessageRequestInterceptor.SERVER_ID.remove();
        }
    }
}
```

而且有了这两个类，路由服务向socket服务传递消息的代码也会非常的”干净“：

```
@Component
public class RpushMessageHandler extends MessageHandler<RpushMessageDTO> {

    // ...
    @Override
```

```
    public void handle(RpushMessageDTO param) {
        List<String> sendTos = param.getReceiverIds();

        for (String sendTo : sendTos) {
            // ...
            messagePushService.push(build); // 路由服务直接调用接口请求即可，” 规则类“和”拦
类“屏蔽掉了其它逻辑，所以这里不需要关心会不会发给错误socket服务
        }
    }
}
```

3. 其它

1. 队列。路由服务内部用 Disruptor环形队列做了异步处理，尽可能地让消息推送接口更快地返。如果是并发量较高的情况，可以加入kafka，路由服务直接监听kafka的消息，以此来提升服务整体能。

2. 缓存。客户端的上线信息可根据情况做多级缓存。即路由服务内部缓存+redis缓存，当然加的存越多，缓存一致性的问题就越复杂，需要考虑的情况也会更多。redis也是需要根据实际情况来决定否要集群部署。

3. 监控。可使用Spring Boot Admin做服务状态监控。

用docker-compose快速部署一个Rpush服务

```
version: '2'
services:
  nginx:
    image: nginx
    container_name: nginx
    ports:
      - 80:80
    volumes:
      - /data/nginx/conf/nginx.conf:/etc/nginx/nginx.conf
      - /data/nginx/log:/var/log/nginx
      - /data/nginx/html:/usr/share/nginx/html
  rpush-eureka:
    image: shuangmulin/rpush-eureka
    container_name: rpush-eureka
    ports:
      - 8761:8761
  rpush-zuul:
    image: shuangmulin/rpush-zuul
    environment:
      - eureka-service-ip=172.16.0.11
      - eureka-service-port=8761
    container_name: rpush-zuul
    ports:
      - 8124:8124
  rpush-route:
    image: shuangmulin/rpush-route
    environment:
      - eureka-service-ip=localhost
      - eureka-service-port=8761
```

```
      - jdbc.url=jdbc:mysql://localhost:3306/rpush?useUnicode=true&characterEncoding=utf-
&useSSL=false&serverTimezone=GMT%2B8
      - jdbc.username=root
      - jdbc.password=123456
      - super-admin.username=superadmin
      - super-admin.password=superadmin
      - jwtSigningKey=fjksadjfklds
    container_name: rpush-route
    ports:
      - 8121:8121
  rpush-server:
    image: shuangmulin/rpush-server
    environment:
      - eureka-service-ip=localhost
      - eureka-service-port=8761
    container_name: rpush-server
    ports:
      - 8122:8122
  rpush-scheduler:
    image: shuangmulin/rpush-scheduler
    environment:
      - eureka-service-ip=localhost
      - eureka-service-port=8761
      - jdbc.url=jdbc:mysql://localhost:3306/rpush?useUnicode=true&characterEncoding=utf-
&useSSL=false&serverTimezone=GMT%2B8
      - jdbc.username=root
      - jdbc.password=123456
      - super-admin.username=superadmin
      - super-admin.password=superadmin
      - jwtSigningKey=fasdferear
    container_name: rpush-scheduler
    ports:
      - 8123:8123
```

运行 docker-compose up -d之后，直接访问8124端口即可