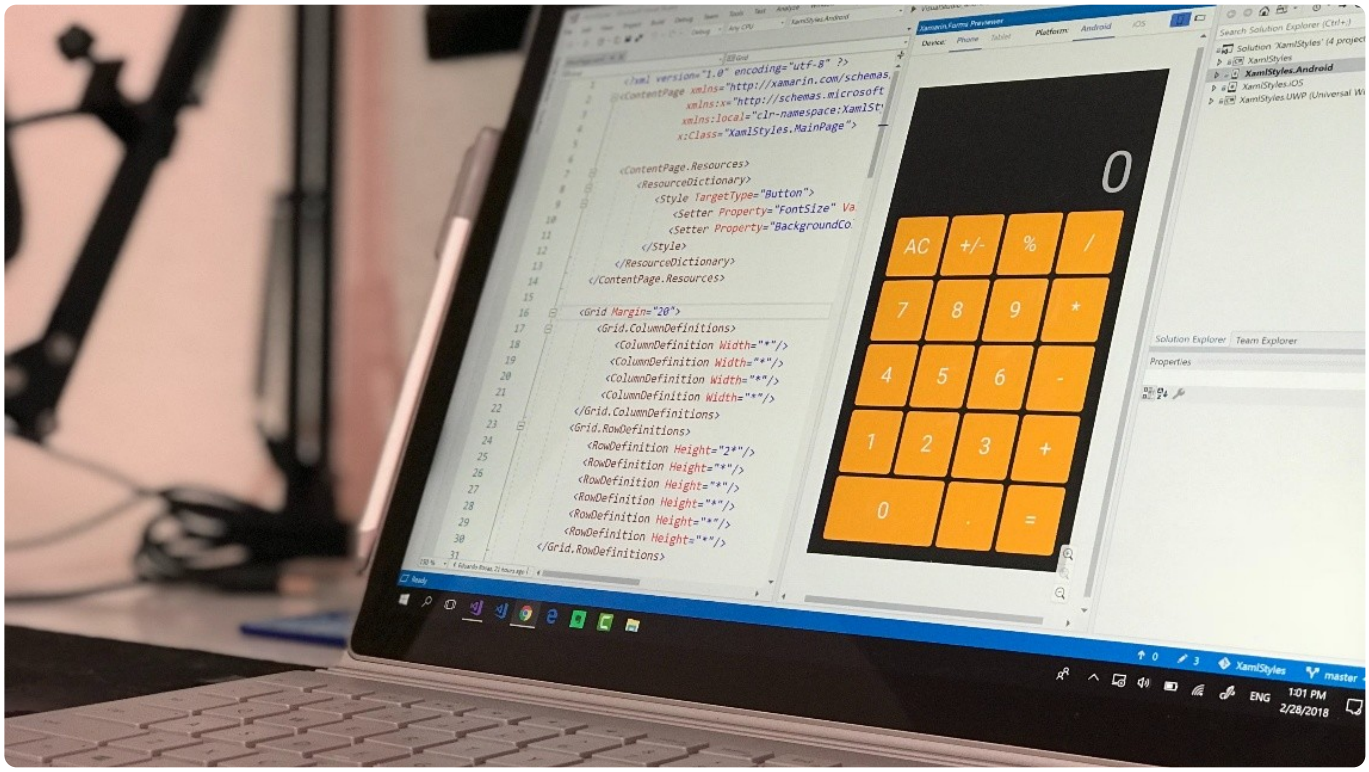
21 | 期末实战：为你的简约版IM系统，加上功能

2019-10-14 袁武林

即时消息技术剖析与实战

进入课程 >



讲述：袁武林

时长 12:02　大小 13.78M

你好，我是袁武林。

在期中实战中，我们一起尝试实现了一个简易版的聊天系统，并且为这个聊天系统增加了一些基本功能。比如，用户登录、简单的文本消息收发、消息存储设计、未读数提示、消息自动更新等。

但是期中实战的目的，主要是让你对 IM 系统的基本功能构成有一个直观的了解，所以在功能的实现层面上比较简单。比如针对消息的实时性，期中采用的是基于 HTTP 短轮询的方式来实现。

因此，在期末实战中，我们主要的工作就是针对期中实战里的消息收发来进行功能优化。

比如，我们会采用 WebSocket 的长连接，来替代之前的 HTTP 短轮询方式，并且会加上一些课程中有讲到的相对高级的功能，如应用层心跳、ACK 机制等。

希望通过期末整体技术实现上的升级，你能更深刻地体会到 IM 系统升级前后，对使用方和服务端压力的差异性。相应的示例代码我放在了GitHub里，你可以作为参考来学习和实现。

功能介绍

关于这次期末实战，希望你能够完成的功能主要包括以下几个部分：

1. 支持基于 WebSocket 的长连接。
2. 消息收发均通过长连接进行通信。
3. 支持消息推送的 ACK 机制和重推机制。
4. 支持客户端的心跳机制和双端的 idle 超时断连。
5. 支持客户端断线后的自动重连。

功能实现拆解

接下来，我们就针对以上这些需要升级的功能和新增的主要功能，来进行实现上的拆解。

WebSocket 长连接

首先，期末实战一个比较大的改变就是，将之前 HTTP 短轮询的实现，改造成真正的长连接。为了方便 Web 端的演示，这里我建议你可以使用 WebSocket 来实现。

对于 WebSocket，我们在客户端 JS（JavaScript）里主要是使用 HTML5 的原生 API 来实现，其核心的实现代码部分如下：

复制代码

```
if (window.WebSocket) {
    websocket = new WebSocket("ws://127.0.0.1:8080");
    websocket.onmessage = function (event) {
        onmsg(event);
    };

    // 连接建立后的事件监听
    websocket.onopen = function () {
        bind();
```

```
        heartBeat.start();
    }

    // 连接关闭后的事件监听
    websocket.onclose = function () {
        reconnect();
    };

    // 连接出现异常后的事件监听
    websocket.onerror = function () {
        reconnect();
    };

} else {
    alert(" 您的浏览器不支持 WebSocket 协议！")
```

页面打开时，JS 先通过服务端的 WebSocket 地址建立长连接。要注意这里服务端连接的地址是 ws:// 开头的，不是 http:// 的了；如果是使用加密的 WebSocket 协议，那么相应的地址应该是以 wss:// 开头的。

建立长连之后，要针对创建的 WebSocket 对象进行事件的监听，我们只需要在各种事件触发的时候，进行对应的逻辑处理就可以了。

比如，API 主要支持的几种事件有：长连接通道建立完成后，通过 onopen 事件来进行用户信息的上报绑定；通过 onmessage 事件，对接收到的所有该连接上的数据进行处理，这个也是我们最核心的消息推送的处理逻辑；另外，在长连接通道发生异常错误，或者连接被关闭时，可以分别通过 onerror 和 onclose 两个事件来进行监听处理。

除了通过事件监听，来对长连接的状态变化进行逻辑处理外，我们还可以通过这个 WebSocket 长连接，向服务器发送数据（消息）。这个功能在实现上也非常简单，你只需要调用 WebSocket 对象的 send 方法就 OK 了。

通过长连接发送消息的代码设计如下：

复制代码

```
var sendMsgJson = '{ "type": 3, "data": {"senderUid":' + sender_id + ',"recipientUid":'

websocket.send(sendMsgJson);
```

此外，针对 WebSocket 在服务端的实现，如果你是使用 JVM（Java Virtual Machine，Java 虚拟机）系列语言的话，我推荐你使用比较成熟的 Java NIO 框架 Netty 来做实现。

因为 Netty 本身对 WebSocket 的支持就很完善了，各种编解码器和 WebSocket 的处理器都有，这样我们在代码实现上就比较简单。

采用 Netty 实现 WebSocket Server 的核心代码，你可以参考下面的示例代码：

复制代码

```
EventLoopGroup bossGroup =
                    new EpollEventLoopGroup(serverConfig.bossThreads, new DefaultThread

EventLoopGroup workerGroup =
                    new EpollEventLoopGroup(serverConfig.workerThreads, new DefaultThrea

ServerBootstrap serverBootstrap = new ServerBootstrap().group(bossGroup, workerGroup).cl

ChannelInitializer<SocketChannel> initializer = new ChannelInitializer<SocketChannel>()
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        // 先添加 WebSocket 相关的编解码器和协议处理器
        pipeline.addLast(new HttpServerCodec());
        pipeline.addLast(new HttpObjectAggregator(65536));
        pipeline.addLast(new LoggingHandler(LogLevel.DEBUG));
        pipeline.addLast(new WebSocketServerProtocolHandler("/", null, true));
        // 再添加服务端业务消息的总处理器
        pipeline.addLast(websocketRouterHandler);
        // 服务端添加一个 idle 处理器，如果一段时间 Socket 中没有消息传输，服务端会强制断开
        pipeline.addLast(new IdleStateHandler(0, 0, serverConfig.getAllIdleSecond()));
        pipeline.addLast(closeIdleChannelHandler);
    }
}

serverBootstrap.childHandler(initializer);
serverBootstrap.bind(serverConfig.port).sync(
```

首先**创建服务器的 ServerBootstrap 对象**。Netty 作为服务端，从 ServerBootstrap 启动，ServerBootstrap 对象主要用于在服务端的某一个端口进行监听，并接受客户端的连接。

接着，**通过 ChannelInitializer 对象，初始化连接管道中用于处理数据的各种编解码器和业务逻辑处理器。**比如这里，我们就需要添加为了处理 WebSocket 协议相关的编解码器，还要添加服务端接收到客户端发送的消息的业务逻辑处理器，并且还加上了用于通道 idle 超时管理的处理器。

最后，**把这个管道处理器链挂到 ServerBootstrap，再通过 bind 和 sync 方法，启动 ServerBootstrap 的端口进行监听**就可以了。

核心消息收发逻辑处理

建立好 WebSocket 长连接后，我们再来看一下最核心的消息收发是怎么处理的。

刚才讲到，客户端发送消息的功能，在实现上其实比较简单。我们只需要通过 WebSocket 对象的 send 方法，就可以把消息通过长连接发送到服务端。

那么，下面我们就来看一下服务端接收到消息后的逻辑处理。

核心的代码逻辑在 WebSocketRouterHandler 这个处理器中，消息接收处理的相关代码如下：

复制代码

```
 @Override
protected void channelRead0(ChannelHandlerContext ctx, WebSocketFrame frame) throws Exce
    // 如果是文本类型的 WebSocket 数据
    if (frame instanceof TextWebSocketFrame) {
        // 先解析出具体的文本数据内容
        String msg = ((TextWebSocketFrame) frame).text();
        // 再用 JSON 来对这些数据内容进行解析
        JSONObject msgJson = JSONObject.parseObject(msg);
        int type = msgJson.getIntValue("type");
        JSONObject data = msgJson.getJSONObject("data");

        long senderUid = data.getLong("senderUid");
        long recipientUid = data.getLong("recipientUid");
        String content = data.getString("content");
        int msgType = data.getIntValue("msgType");
        // 调用业务层的 Service 来进行真正的发消息逻辑处理
        MessageVO messageContent = messageService.sendNewMsg(senderUid, recipientUid, c

        if (messageContent != null) {
            JSONObject jsonObject = new JSONObject();
            jsonObject.put("type", 3);
```

```
            jsonObject.put("data", JSONObject.toJSON(messageContent));
                        ctx.writeAndFlush(new TextWebSocketFrame(JSONObject.toJSONStrin
        }
    }
}
```

这里的 WebSocketRouterHandler，我们也是采用事件监听机制来实现。由于这里需要处理“接收到”的消息，所以我们只需要实现 channelRead0 方法就可以。

在前面的管道处理器链中，因为添加了 WebSocket 相关的编解码器，所以这里的 WebSocketRouterHandler 接收到的都是 WebSocketFrame 格式的数据。

接下来，我们从 WebSocketFrame 格式的数据中，解析出文本类型的收发双方 UID 和发送内容，就可以调用后端业务模块的发消息功能，来进行最终的发消息逻辑处理了。

最后，把需要返回给消息发送方的客户端的信息，再通过 writeAndFlush 方法写回去，就完成消息的发送。

不过，以上的代码只是处理消息的发送，那么针对消息下推的逻辑处理又是如何实现的呢？

刚刚讲到，客户端发送的消息，会通过后端业务模块来进行最终的发消息逻辑处理，这个处理过程也包括消息的推送触发。

因此，我们可以在 messageService.sendNewMsg 方法中，等待消息存储、未读变更都完成后，再处理待推送给接收方的消息。

你可以参考下面的核心代码：

复制代码

```
private static final ConcurrentHashMap<Long, Channel> userChannel = new ConcurrentHashM

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, WebSocketFrame frame) throws
        // 处理上线请求
        long loginUid = data.getLong("uid");
        userChannel.put(loginUid, ctx.channel());
    }
public void pushMsg(long recipientUid, JSONObject message) {
```

```
    Channel channel = userChannel.get(recipientUid);
    if (channel != null && channel.isActive() && channel.isWritable()) {
        channel.writeAndFlush(new TextWebSocketFrame(message.toJSONString()));
    }
}
```

首先，我们在处理用户建连上线的请求时，会先在网关机内存记录一个“当前连接用户和对应的连接”的映射。

当系统有消息需要推送时，我们通过查询这个映射关系，就能找到对应的连接，然后就可以通过这个连接，将消息下推下去。

复制代码

```
public class NewMessageListener implements MessageListener {
    @Override
    public void onMessage(Message message, byte[] pattern) {
        String topic = stringRedisSerializer.deserialize(message.getChannel());
        // 从订阅到的 Redis 的消息里解析出真正需要的业务数据
        String jsonMsg = valueSerializer.deserialize(message.getBody());
        logger.info("Message Received --> pattern: {}，topic:{}，message: {}", new Strir
        JSONObject msgJson = JSONObject.parseObject(jsonMsg);
        // 解析出消息接收人的 UID
        long otherUid = msgJson.getLong("otherUid");
        JSONObject pushJson = new JSONObject();
        pushJson.put("type", 4);
        pushJson.put("data", msgJson);

        // 最终调用网关层处理器将消息真正下推下去
        websocketRouterHandler.pushMsg(otherUid, pushJson);

    }
}

@Override
public MessageVO sendNewMsg(long senderUid, long recipientUid, String content, int msgTy

    // 先对发送消息进行存储、加未读等操作
    //...
    // 然后将待推送消息发布到 Redis
    redisTemplate.convertAndSend(Constants.WEBSOCKET_MSG_TOPIC, JSONObject.toJSONString
```

然后，我们可以基于 Redis 的发布 / 订阅，实现一个消息推送的发布订阅器。

在业务层进行发送消息逻辑处理的最后，会将这条消息发布到 Redis 的一个 Topic 中，这个 Topic 被 NewMessageListener 一直监听着，如果有消息发布，那么监听器会马上感知到，然后再将消息提交给 WebSocketRouterHandler，来进行最终消息的下推。

消息推送的 ACK

我在“04 | ACK 机制：如何保证消息的可靠投递？”中有讲到，当系统有消息下推后，我们会依赖客户端响应的 ACK 包，来保证消息推送的可靠性。如果消息下推后一段时间，服务端没有收到客户端的 ACK 包，那么服务端会认为这条消息没有正常投递下去，就会触发重新下推。

关于 ACK 机制相应的服务端代码，你可以参考下面的示例：

复制代码

```
public void pushMsg(long recipientUid, JSONObject message) {
    channel.writeAndFlush(new TextWebSocketFrame(message.toJSONString()));
    // 消息推送下去后，将这条消息加入到待 ACK 列表中
    addMsgToAckBuffer(channel, message);
}
public void addMsgToAckBuffer(Channel channel, JSONObject msgJson) {
    nonAcked.put(msgJson.getLong("tid"), msgJson);
    // 定时器针对下推的这条消息在 5s 后进行 " 是否 ACK" 的检查
    executorService.schedule(() -> {
        if (channel.isActive()) {
            // 检查是否被 ACK，如果没有收到 ACK 回包，会触发重推
            checkAndResend(channel, msgJson);
        }
    }, 5000, TimeUnit.MILLISECONDS);
}
long tid = data.getLong("tid");
nonAcked.remove(tid);
private void checkAndResend(Channel channel, JSONObject msgJson) {
    long tid = msgJson.getLong("tid");
    // 重推 2 次
    int tryTimes = 2;
    while (tryTimes > 0) {
        if (nonAcked.containsKey(tid) && tryTimes > 0) {
            channel.writeAndFlush(new TextWebSocketFrame(msgJson.toJSONString()));
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        tryTimes--;
    }
```

```
33 }
```

用户在上线完成后，服务端会在这个连接维度的存储里，初始化一个起始值为 0 的序号（tid），每当有消息推送给客户端时，服务端会针对这个序号进行加 1 操作，下推消息时就会携带这个序号连同消息一起推下去。

消息推送后，服务端会将当前消息加入到一个“待 ACK Buffer”中，这个 ACK Buffer 的实现，我们可以简单地用一个 ConcurrentHashMap 来实现，Key 就是这条消息携带的序号，Value 是消息本身。

当消息加入到这个“待 ACK Buffer”时，服务端会同时创建一个定时器，在一定的时间后，会触发“检查当前消息是否被 ACK”的逻辑；如果客户端有回 ACK，那么服务端就会从这个“待 ACK Buffer”中移除这条消息，否则如果这条消息没有被 ACK，那么就会触发消息的重新下推。

应用层心跳

在了解了如何通过 WebSocket 长连接，来完成最核心的消息收发功能之后，我们再来看下，针对这个长连接，我们如何实现新增加的应用层心跳功能。

应用层心跳的作用，我在第 8 课“智能心跳机制：解决网络的不确定性”中也有讲到过，主要是为了解决由于网络的不确定性，而导致的连接不可用的问题。

客户端发送心跳包的主要代码设计如下，不过我这里的示例代码只是一个简单的实现，你可以自行参考，然后自己去尝试动手实现：

复制代码

```
// 每 2 分钟发送一次心跳包，接收到消息或者服务端的响应又会重置来重新计时。
var heartBeat = {
    timeout: 120000,
    timeoutObj: null,
    serverTimeoutObj: null,
    reset: function () {
        clearTimeout(this.timeoutObj);
        clearTimeout(this.serverTimeoutObj);
        this.start();
    },
    start: function () {
```

```
        var self = this;
        this.timeoutObj = setTimeout(function () {
            var sender_id = $("#sender_id").val();
            var sendMsgJson = '{ "type": 0, "data": {"uid":' + sender_id + ',"timeout":
            websocket.send(sendMsgJson);
            self.serverTimeoutObj = setTimeout(function () {
                websocket.close();
                $("#ws_status").text(" 失去连接！");
            }, self.timeout)
        }, this.timeout)
    },
}
```

客户端通过一个定时器，每 2 分钟通过长连接给服务端发送一次心跳包，如果在 2 分钟内接收到服务端的消息或者响应，那么客户端的下次 2 分钟定时器的计时，会进行清零重置，重新计算；如果发送的心跳包在 2 分钟后没有收到服务端的响应，客户端会断开当前连接，然后尝试重连。

我在下面的代码示例中，提供的“服务端接收到心跳包的处理逻辑”的实现过程，其实非常简单，只是封装了一个普通回包消息进行响应，代码设计如下：

复制代码

```
@Override
protected void channelRead0(ChannelHandlerContext ctx, WebSocketFrame frame) throws Exce
    long uid = data.getLong("uid");
    long timeout = data.getLong("timeout");
    logger.info("[heartbeat]: uid = {} , current timeout is {} ms, channel = {}", uid, t
    ctx.writeAndFlush(new TextWebSocketFrame("{\"type\":0,\"timeout\":" + timeout + "}"
}
```

我们实际在线上实现的时候，可以采用前面介绍的“智能心跳”机制，通过服务端对心跳包的响应，来计算新的心跳间隔，然后返回给客户端来进行调整。

好，到这里，期末实战的主要核心功能基本上也讲解得差不多了，细节方面你可以再翻一翻我在GitHub上提供的示例代码。

对于即时消息场景的代码实现来说，如果要真正达到线上使用的程度，相应的代码量是非常庞大的；而且对于同一个功能的实现，根据不同的使用场景和业务特征，很多业务在设计上

也会有较大的差异性。

所以，实战课程的设计和示例代码只能做到挂一漏万，我尽量通过最简化的代码，来让你真正了解某一个功能在实现上最核心的思想。并且，通过期中和期末两个阶段的功能升级与差异对比，使你能感受到这些差异对于使用方体验和服务端压力的改善，从而可以更深刻地理解和掌握前面课程中相应的理论点。

小结

今天的期末实战，我们主要是针对期中实战中 IM 系统设计的功能，来进行优化改造。

比如，**使用基于 WebSocket 的长连接**，代替基于 HTTP 的短轮询，来提升消息的实时性，并增加了**应用层心跳、ACK 机制**等新功能。

通过这次核心代码的讲解，是想让你能理论结合实际地去理解前面课程讲到的，IM 系统设计中最重要的部分功能，也希望你能自己尝试去动手写一写。当然，你也可以基于已有代码，去增加一些之前课程中有讲到，但是示例代码中没有实现的功能，比如离线消息、群聊等。

最后再给你留一个思考题：**ACK 机制的实现中，如果尝试多次下推之后仍然没有成功，服务端后续应该进行哪些处理呢？**

以上就是今天课程的内容，欢迎你给我留言，我们可以在留言区一起讨论，感谢你的收听，我们下期再见。

上一篇　20 | 存储和并发：万人群聊系统设计中的几个难点

精选留言 (2)

写留言

yangzi
2019-10-14

回答问题：如果多次下推仍然没有成功，关闭客户端连接，将未回执的消息存储为离线消息。

墙角儿的花
2019-10-14

回答问题，关闭清除客户端连接和待ACK列表

展开