上次已经简单的谈了一些MQTT协议的一些知识，今天就来就上次的知识具体的Java实现。

现在就来具体说说实现这一步吧。中间的时间也是有点久。

MQTT消息的发送和订阅都是依赖MQTT服务器的，没有MQTT服务器，你的客户端是无法订阅和发送消息的。所以在最开始的时候，可以选择性的在你的电脑上面安装一个MQTT服务器。MQTT服务器有很多，大家也可以在网上去找一些安装教程，这里因为和我要讲内容关系不大，所以不再累述。

MQTT协议中是没有发送者和接收者·的概念，所有的连接都是用户，所以一个MQTT连接既可以发送消息，也可以接收消息。就等于所有的连接都是客户端。下面我的客户端代码也是如此，因为公司这边接收的信息先是要进行认证，认证成功后再接收有用的信息。这时，客户端在根据设备的信息来控制网关上面的设备，达到远程控制设备的目的。因为要使用服务器来转发消息，所以对于服务器的测试也是比较重要的，但是我使用的是公司的服务器，所以这一块我的了解比较少。但是我这边有一些工具，谷歌浏览器的插件MQTTLens。可能会帮助你。（需要翻阅墙体）

MQTT使用的库也是有很多的，下面的网址也是列举了MQTT支持的库，有java的，也有c的。网址如下：<https://github.com/mqtt/mqtt.github.io/wiki/libraries>。因为最开始我的接触还是比较浅，使用的是：[Fusesource mqtt-client](https://github.com/fusesource/mqtt-client)。所以java的demo也是基于这个库的，但是后来和spring整合的时候发现有一些问题，因为spring支持的只有一个库，就是[Eclipse Paho Java](https://www.eclipse.org/paho/clients/java/)。但是原理都是一样的，大家可以自己去决定，我的简单的demo代码还是基于[Fusesource mqtt-client](https://github.com/fusesource/mqtt-client)。在下一篇Spring和MQTT整合中使用的是[Eclipse Paho Java](https://www.eclipse.org/paho/clients/java/)。

下面就说一说具体的思路，这边我的代码是基于公司的网关需求，所以先说一说公司网关的具体流程。首先，网关会一直发送身份验证消息，等待客户端认证，客户端认证通过后，会发送具体有用的信息。客户端这时在根据网关信息发送控制命令，到达控制的目的。在这个过程中，客户端有订阅和发送，所以一个客户端就练习了发送消息和订阅消息。这就是公司的具体操作流程。下面就说一说代码的流程。

运行时要使用jar包，也可使用maven，但是使用maven时要注意版本。

具体的jar包和maven依赖在网址：<https://gitee.com/iots/mqtt-client>

依赖为：

<dependency>

<groupId>org.fusesource.mqtt-client</groupId>

<artifactId>mqtt-client</artifactId>

<version>1.12</version>

</dependency>

下面开始编写demo

首先先要配置MQTT的一些配置，配置比较多，也很繁琐。

主要是配置主机号和端口号，根据自己的配置编写代码，在配置其他的一些细节配置，主要是和连接有关的。

代码如下：

// MQTT设置说明

// 设置主机号

mqtt.setHost("tcp://10.168.5.208:1883");

// 用于设置客户端会话的ID。在setCleanSession(false);被调用时，MQTT服务器利用该ID获得相应的会话。此ID应少于23个字符，默认根据本机地址、端口和时间自动生成

mqtt.setClientId("876543210");

// 若设为false，MQTT服务器将持久化客户端会话的主体订阅和ACK位置，默认为true

mqtt.setCleanSession(**false**);

// 定义客户端传来消息的最大时间间隔秒数，服务器可以据此判断与客户端的连接是否已经断开，从而避免TCP/IP超时的长时间等待

mqtt.setKeepAlive((**short**) 60);

// 服务器认证用户名

mqtt.setUserName("admin");

// 服务器认证密码

mqtt.setPassword("admin");

// 设置“遗嘱”消息的话题，若客户端与服务器之间的连接意外中断，服务器将发布客户端的“遗嘱”消息

mqtt.setWillTopic("willTopic");

// 设置“遗嘱”消息的内容，默认是长度为零的消息

mqtt.setWillMessage("willMessage");

// 设置“遗嘱”消息的QoS，默认为QoS.ATMOSTONCE

mqtt.setWillQos(QoS.***AT\_LEAST\_ONCE***);

// 若想要在发布“遗嘱”消息时拥有retain选项，则为true

mqtt.setWillRetain(**true**);

// 设置版本

mqtt.setVersion("3.1.1");

// 失败重连接设置说明

// 客户端首次连接到服务器时，连接的最大重试次数，超出该次数客户端将返回错误。-1意为无重试上限，默认为-1

mqtt.setConnectAttemptsMax(10L);

// 客户端已经连接到服务器，但因某种原因连接断开时的最大重试次数，超出该次数客户端将返回错误。-1意为无重试上限，默认为-1

mqtt.setReconnectAttemptsMax(3L);

// 首次重连接间隔毫秒数，默认为10ms

mqtt.setReconnectDelay(10L);

// 重连接间隔毫秒数，默认为30000ms

mqtt.setReconnectDelayMax(30000L);

// 设置重连接指数回归。设置为1则停用指数回归，默认为2

mqtt.setReconnectBackOffMultiplier(2);

// Socket设置说明

// 设置socket接收缓冲区大小，默认为65536（64k）

mqtt.setReceiveBufferSize(65536);

// 设置socket发送缓冲区大小，默认为65536（64k）

mqtt.setSendBufferSize(65536);

// 设置发送数据包头的流量类型或服务类型字段，默认为8，意为吞吐量最大化传输

mqtt.setTrafficClass(8);

// 带宽限制设置说明

// 设置连接的最大接收速率，单位为bytes/s。默认为0，即无限制

mqtt.setMaxReadRate(0);

// 设置连接的最大发送速率，单位为bytes/s。默认为0，即无限制

mqtt.setMaxWriteRate(0);

// 选择消息分发队列

// 若没有调用方法setDispatchQueue，客户端将为连接新建一个队列。如果想实现多个连接使用公用的队列，显式地指定队列是一个非常方便的实现方法

mqtt.setDispatchQueue(Dispatch.*createQueue*("foo"));

上面都是一些配置的问题，具体情况自己决定配置。具体的配置也可以参考下面的网址，下面的网址也有详细的描述。

https://gitee.com/iots/mqtt-client

下面开始讲讲连接和订阅和发送主题

fusesource提供三种mqtt client api，分别为阻塞API，基于Futur的API和回调API。

其中**，**阻塞API是在MQTT.connectBlocking方法建立连接和提供阻断API的连接。

基于Futur的API则是：在MQTT.connectFuture方法建立连接，为您提供了一个与结合Futur的连接。所有操作的连接是无阻塞的，并且经由返回的结果。

回调API是最复杂的也是性能最好的，另外两种均是对回调API的封装。

因为回调API有些复杂，现在只是介绍回调API的封装。就是前两个，前两个的区别是第一个为阻塞的，第二个不是阻塞。下面开始代码演示。

**第一个**阻塞API。代码如下：

//接收消息

Message message = connection.receive();

//将收到的消息封装为消息类

ReceiveRealValiedateVO realValiedateVO= JSON.*parseObject*(message.getPayloadBuffer().toByteArray(), ReceiveRealValiedateVO.**class**);

System.***out***.println(realValiedateVO.getG\_id());

**if**(realValiedateVO.getS\_id().equals("1") && realValiedateVO.getS\_id().equals("1")) {

//新建json

JSONObject jsonParam = **new** JSONObject();

//封装json

jsonParam.put("s\_id",realValiedateVO.getS\_id());

jsonParam.put("g\_id",realValiedateVO.getG\_id());

jsonParam.put("seq",realValiedateVO.getSeq());

jsonParam.put("type",realValiedateVO.getType());

jsonParam.put("valid","true");

String s=jsonParam.toString();

//发送特定主题的消息

connection.publish("datasources/1/1/req\_ack", s.getBytes(), QoS.***AT\_LEAST\_ONCE***, **false**);

}

//打印主题

System.***out***.println(message.getTopic());

//byte[] payload = message.getPayload();

System.***out***.println(String.*valueOf*(message.getPayloadBuffer()));

// process the message then:

message.ack();

//连接断开

//connection.disconnect();

具体的解释都在代码里了，剩下就没有什么了。

要注意的点就是连接中断的处理，和对于服务器的处理。

第二种就是使用future连接,代码如下：

//使用future连接

FutureConnection connection = mqtt.futureConnection();

Future<Void> f1 = connection.connect();

f1.await();

//订阅消息

Future<**byte**[]> f2 = connection.subscribe(**new** Topic[]{**new** Topic("datasources/1/1", QoS.***AT\_LEAST\_ONCE***)});

//

**byte**[] qoses = f2.await();

// 发送身份验证消息.

//Future<Void> f3 = connection.publish("foo", "Hello".getBytes(), QoS.AT\_LEAST\_ONCE, false);

// 接收订阅消息..

Future<Message> receive = connection.receive();

// 打印消息.

Message message = receive.await();

System.***out***.println(String.*valueOf*(message.getPayloadBuffer()));

//回应

message.ack();

//

Future<Void> f4 = connection.disconnect();

f4.await();

第三个是最难的，我这边的代码也是有点乱，直接上代码吧。

// 监听

connection.listener(**new** Listener() {

@Override

**public** **void** onPublish(UTF8Buffer topicmsg, Buffer msg, Runnable ack) {

// utf-8 is used for dealing with the garbled

String topic = topicmsg.utf8().toString();

String payload = msg.utf8().toString();

System.***out***.println(topic + " " + payload);

String Amsg = AuthenticationSendDemo.*Authentication*(topic, payload);

**if** (topic.equals("datasources/req")) {

// 重起一个阻塞线程

connection.getDispatchQueue().execute(**new** Runnable() {

**public** **void** run() {

connection.publish("datasources/17/01/req\_ack", Amsg.getBytes(), QoS.***AT\_LEAST\_ONCE***, **false**,

**new** Callback<Void>() {

@Override

**public** **void** onSuccess(Void args) {

// 表示发布主题成功

System.***out***.println("发布成功！");

System.***out***.println("发布的消息" + Amsg);

}

@Override

**public** **void** onFailure(Throwable throwable) {

// 表示发布主题失败

System.***out***.println("发布失败！");

}

});

}

});

}

// 表示监听成功

ack.run();

}

@Override

**public** **void** onFailure(Throwable value) {

// 表示监听失败

}

// execute only once when connection is ended

@Override

**public** **void** onDisconnected() {

// 表示监听到断开连接

System.***out***.println("断开连接！！");

}

// execute only once when connecting started

@Override

**public** **void** onConnected() {

// 表示监听到连接成功

System.***out***.println("haha");

System.***out***.println();

}

});

因为代码中使用到了线程和回调，我对于这两个掌握的也不是很好，也不再这里乱扯，有大佬知道比较好的写法最好指点一下。在这里感谢。

三种写法都写完了，下面谈一谈感想和中间遇到的问题。

以为看具体的文档实在太多了，现在公司还在忙着赶项目，我这边时间也不是很多，代码的整理以后有时间在说。我感觉最重要的还是对于协议的一些掌握和体会，这些要比上面的代码重要的多，因为你最终的代码还是要和项目整合的，和Spring整合的时候你会发现这些都是框架提供好了，你需要做的就是填参数，但是整合中遇到的问题的解决办法都是你从写上面的代码中得到的。

因为刚开始写代码，所以代码中的注释也是非常多的，这里也不再累述。写上面的代码的时候遇到了很多的问题，解决的网站都在我第一篇MQTT博客中，比如MQTT的官网，网上的文章都是抄的，要不就是一知半解（我也是）。最终还是看自己的深入体会。

就这样吧，结束。