

STOMP协议详解

<http://blog.csdn.net/chszs/article/details/46592777>

<http://nikipore.github.io/stompest/>

STOMP即Simple (or Streaming) Text Orientated Messaging Protocol，简单(流)文本定向消息协议，它提供了一个可互操作的连接格式，允许STOMP客户端与任意STOMP消息代理（Broker）进行交互。STOMP协议由于设计简单，易于开发客户端，因此在多种语言和多种平台上得到广泛地应用。

STOMP协议的前身是TTMP协议（一个简单的基于文本的协议），专为消息中间件设计。

STOMP是一个非常简单和容易实现的协议，其设计灵感源自于HTTP的简单性。尽管STOMP协议在服务器端的实现可能有一定的难度，但客户端的实现却很容易。例如，可以使用Telnet登录到任何的STOMP代理，并与STOMP代理进行交互。

STOMP协议与2012年10月22日发布了最新的STOMP 1.2规范。
要查看STOMP 1.2规范，见：<https://stomp.github.io/stomp-specification-1.2.html>

二、STOMP的实现

业界已经有很多优秀的STOMP的服务器/客户端的开源实现，下面就介绍一下这方面的情况。

1、STOMP服务器

项目名	兼容STOMP的版本	描述
Apache Apollo	1.0 1.1 1.2	ActiveMQ的继承者 http://activemq.apache.org/apollo
Apache		

ActiveMQ	1.0 1.1	流行的开源消息服务器 http://activemq.apache.org/
HornetQ	1.0	来自JBoss的消息中间件 http://www.jboss.org/hornetq
RabbitMQ	1.0 1.1 1.2	基于Erlang、支持多种协议的消息Broker，通过插件支持STOMP协议 http://www.rabbitmq.com/plugins.html#rabbitmq-stomp
Stampy	1.2	STOMP 1.2规范的一个Java实现 http://mrstampy.github.com/Stampy/
StompServer	1.0	一个轻量级的纯Ruby实现的STOMP服务器 http://stompserver.rubyforge.org/

这里只列了部分。

2、STOMP客户端库

项目名	兼容STOMP的版本	描述
activemessaging	1.0	Ruby客户端库 http://code.google.com/p/activemessaging/
onstomp	1.0 1.1	Ruby客户端库 https://rubygems.org/gems/onstomp
Apache CMS	1.0	C++客户端库 http://activemq.apache.org/cms/
Net::STOMP::Client	1.0 1.1 1.2	Perl客户端库 http://search.cpan.org/dist/Net-STOMP-Client/
Gozirra	1.0	Java客户端库 http://www.germane-software.com/software/Java/Gozirra/
libstomp	1.0	C客户端库，基于APR库 http://stomp.codehaus.org/C
Stampy	1.2	Java客户端库 http://mrstampy.github.com/Stampy/
stomp.js	1.0 1.1	JavaScript客户端库 http://jmesnil.net/stomp-websocket/doc/
stompest	1.0 1.1 1.2	Python客户端库，全功能实现，包括同步和异步 https://github.com/nikipore/stompest

StompKit	1.2	Objective-C客户端库，事件驱动 https://github.com/mobile-web-messaging/StompKit/
stompngo	1.0 1.1 1.2	Go客户端库 https://github.com/gmallard/stompngo
stomp.py	1.0 1.1 1.2	Python客户端库 https://github.com/jasonrbriggs/stomp.py
tStomp	1.1	TCL客户端库 https://github.com/siemens/tstomp

这里只列了部分。

三、STOMP协议分析

STOMP协议与HTTP协议很相似，它基于TCP协议，使用了以下命令：

CONNECT
SEND
SUBSCRIBE
UNSUBSCRIBE
BEGIN
COMMIT
ABORT
ACK
NACK
DISCONNECT

STOMP的客户端和服务端之间的通信是通过“帧”（Frame）实现的，每个帧由多“行”（Line）组成。

第一行包含了命令，然后紧跟键值对形式的Header内容。

第二行必须是空行。

第三行开始就是Body内容，末尾都以空字符结尾。

STOMP的客户端和服务端之间的通信是通过MESSAGE帧、RECEIPT帧或ERROR帧实现的，它们的格式相似。

=====

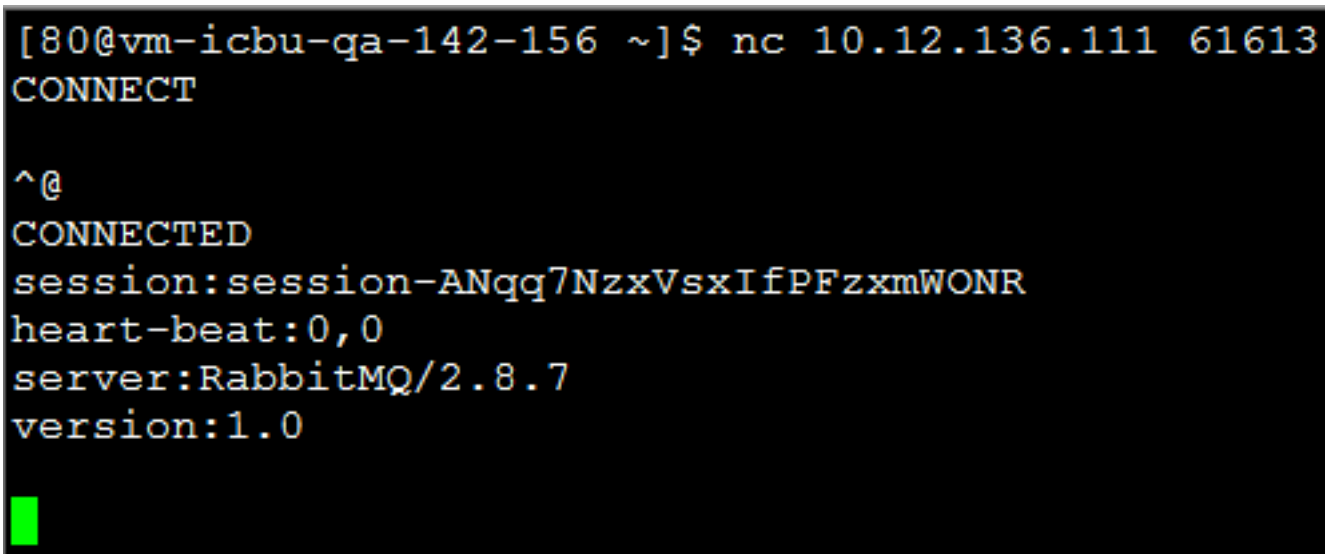
Stomp是一个简单的消息文本协议,它的设计核心理念就是简单与可用性, 官方文档: <http://stomp.github.com/stomp-specification-1.1.html>

现在我们就来实践一下Stomp协议, 你需要的是:

- 1.一个支持stomp消息协议的messaging server(譬如activemq, rabbitmq) ;
- 2.一个终端 (譬如linux shell);
- 3.一些基本命令与操作 (譬如nc, telnet)

1.建立连接

当我们 (Client端) 向服务器发送一个CONNECT Frame, 就向服务器发起了一个连接请求, 此时服务器端返回一个CONNECTED Frame表示建立连接成功, 其中头字段version表示采用的stomp协议版本 (这里默认是1.0)



```
[80@vm-icbu-qa-142-156 ~]$ nc 10.12.136.111 61613
CONNECT

^@
CONNECTED
session:session-ANqq7NzxVsxIfPFzxmWONR
heart-beat:0,0
server:RabbitMQ/2.8.7
version:1.0
```

ps: ^@符号 ctrl+@键(用来提交请求) , 删除之前输入的数据
ctrl+n+backspace键

当然Client端也可指定所支持的协议版本 (accept-version字段, 多个版本按递增顺序排列, 并用逗号分隔) ;

```
[80@vm-icbu-qa-142-156 ~]$ nc 10.12.136.111 61613
CONNECT
accept-version:1.0,1.1,2.0

^@
CONNECTED
session:session-QkGRI8Q4ermqE5vhW-6lU-
heart-beat:0,0
server:RabbitMQ/2.8.7
version:1.1
```

服务器此时返回的CONNECTED Frame中会列出它所支持的协议版本号中最高的那个（如上图的version: 1.1）

如果服务器端不支持客户端所列举的协议版本（比如这里的2.1），那么服务器会返回一个ERROR Frame并且列举出服务器自己所支持的协议版本（如下图的version:1.0,1.1）

```
[80@vm-icbu-qa-142-156 ~]$ nc 10.12.136.111 61613
CONNECT
accept-version:2.1

^@
ERROR
message:Version mismatch
content-type:text/plain
version:1.0,1.1
content-length:31

Supported versions are 1.0,1.1
```

2.消息传递

客户端一旦与服务器端建立连接，那么就可发送下列Frame进行消息传递

SEND
SUBSCRIBE
UNSUBSCRIBE
ACK

NACK
BEGIN
COMMIT
ABORT
DISCONNECT

SEND Frame 用来将客户端消息发送到目的地（destination）,因此它必须指定一个destination头字段，另外在所有头字段之后，新起一个空行，之后就是需要发送的消息（譬如这里的 hello stomp!)

```
SEND
destination:/queue/my_queue

hello stomp!
@
```

此时"hello stomp!" 这条消息就被发送到了队列 my_queue中去

现在再起一个客户端clinet_a，来接受这个队列（myqueue）中的消息

```
SUBSCRIBE
id:client_a
destination:/queue/my_queue

^@
MESSAGE
subscription:client_a
destination:/queue/my_queue
message-id:T_client_a@@session-w6FM_nV4NLsEzOyCvh9iFx@@1
content-length:14

hello stomp!
```

SUBSCRIBE Frame 表示客户端希望订阅某一个目的地（destination）的消息（这里是/queue/my_queue），其中头字段id表示在一个会话连接里，唯一标示一个订阅者（subscription），头字段destination标示该订阅者

(subscription) 需要订阅的目的地（这里是一个队列， /queue/my_queue)

紧接着，我们就接受到服务器端发送来的消息（MESSAGE Frame），其中 message_id:唯一标示了这条消息（后面我们会使用这个消息id进行ack,uack操作），content-length:标示消息体的长度，在所有这些头后面，新起一个空行就是消息内容（如这里的hello stomp!)

如此时我们希望订阅另一个destination，该如何办呢？，是不是再发送一个SUBSCRIBE Frame就好了？

```
SUBSCRIBE
id:client_a
destination:/queue/other_queue

^@
ERROR
message:Processing error
content-type:text/plain
version:1.0,1.1
content-length:17

Processing error
ERROR
message:not_allowed
content-type:text/plain
version:1.0,1.1
content-length:43

attempt to reuse consumer tag 'T_client_a'
```

结果发现服务器端返回了ERROR Frame 告诉我们SUBSCRIBE失败，原来同一个subscription id只能订阅一个destination，要想订阅另一个destination，必须先发送UNSUBSCRIBE Frame,然后再SUBSCRIBE 到新的目的地

```
SUBSCRIBE
id:client_A
destination:/queue/my_queue

^@
UNSUBSCRIBE
id:client_A

^@
SUBSCRIBE
id:client_A
destination:/queue/other_queue

^@
```

UNSCUBSCRIBE Frame中的id标示需要取消订阅的subscription，然后我们在订阅到新的destination（这里是/queue/other_queue),可以发现订阅成功，服务器没有再发送ERROR Frame

至此，我们就模拟出了一个PTP（point-to-point）消息模型，下面我们也模拟下另外一个pub/sub消息模型：

准备工作，新起两个连接订阅到/topic/my_topic上，如下图：

clinet_a

```
[80@vm-icbu-qa-142-157 ~]$ nc 10.12.136.111 61613
CONNECT

^@
CONNECTED
session:session-QU9EIfAQkkAkEPsAl5OO3Q
heart-beat:0,0
server:RabbitMQ/2.8.7
version:1.0

SUBSCRIBE
id:client_a
destination:/topic/my_topic

^@
```

client b


```
[80@vm-ae-qa-142-94 ~]$ nc 10.12.136.111 61613
CONNECT

^@
CONNECTED
session:session-QrF5Q1yjKXd8DRfvie3BPV
heart-beat:0,0
server:RabbitMQ/2.8.7
version:1.0

SUBSCRIBE
id:client_b
destination:/topic/my_topic

^@
```

Send端发送广播消息：

```
SEND
destination:/topic/my_topic

hello everybody!
^@
```

我们去检查下两个消息接收客户端，果然发现收到了这条广播消息^_^
client a

```
MESSAGE
subscription:client_a
destination:/topic/my_topic
message-id:T_client_a@@session-QU9EIfAQkkAkEPsAl5OO3Q@@1
content-length:17

hello everybody!
```

client b

```
MESSAGE
subscription:client_b
destination:/topic/my_topic
message-id:T_client_b@@session-QrF5Q1yjKXd8DRfvie3BPV@@1
content-length:17

hello everybody!
```

默认情况下，只要服务器端发送消息，就认为客户端接收成功（即ack模式为auto），若我们需要更严格的消息保证，则必须采用client模式，即由客户端确认消息的接受

服务器发送了一个消息到/queue/ackqueue

```
SEND
destination:/queue/ackqueue

hello word!
^@
```

此时我们的客户端client_a确实接受到了该消息

```
SUBSCRIBE
id:client_a
destination:/queue/ackqueue
ack:client

^@
MESSAGE
subscription:client_a
destination:/queue/ackqueue
message-id:T_client_a@@session-gAqD5YzCoMlvUdaLoy_7HG@@2
content-length:12

hello word!
```

虽然该消息已经发送到客户端，但是由于该消息没有确认（ack），则该消息还保存在队列/queue/ackqueue中（直到客户端确认才删除），我们可以通过rabbitmq提供的命令来查看（sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged）：

```
Listing queues ...
ackqueue          0          1
```

若客户端在确认消息前与客户端断开连接，那么服务器可能(根据不同server的设计而不同)会选择将该消息发送给另一个subscription（这里是client_b）

```
SUBSCRIBE
id:client_b
destination:/queue/ackqueue

^@
MESSAGE
subscription:client_b
destination:/queue/ackqueue
message-id:T_client_b@@session-Aai2u8-ExidC4JeNAPhOJb@@1
content-length:12

hello word!
```

可以看出rabbit是选择将它发送给另一个subscription， 我们在使用命令查ack_queue队列中是否还有未确认的消息

```
Listing queues ...
ackqueue          0          0
```

结果发现没有了 ^_^

若是客户端希望确认一个消息，该如何做呢？，只要发送一个ACK Frame即可!

在ACK Frame中，subscription标示是谁确认消息， message-id标示是确认哪条消息（即MESSAGE Frame中携带的message-id）

```
SUBSCRIBE
id:client_a
destination:/queue/ackqueue
ack:client

^@
MESSAGE
subscription:client_a
destination:/queue/ackqueue
message-id:T_client_a@@session-g7zb2CegDprHHcpecy1Ohe@@1
content-length:13

HOW ARE YOU!
ACK
subscription:client_a
message-id:T_client_a@@session-g7zb2CegDprHHcpecy1Ohe@@1

^@
```

另外我在rabbitmq的测试结果发现，ack具有累积效应(因为这里SUBSCRIBE帧的ack头被设置成了‘client’)，譬如接收了10条消息，如果你ack了第8条消息，那么1-7条消息都会被ack，只有9-10两条消息还保持未ack状态

除了有ACK Frame，还有NACK Frame，它表示客户端未成功接收某条消息，这时候服务器可以选择重发或者丢弃（对于rabbitmq，我的测试结果是选择重发）

3.断开连接

说完了连接的建立，消息的发送与接收，现在我们来看看客户端如何与服务端断开连接的，更重要的是如何安全的断开连接

```
[80@vm-icbu-qa-142-156 ~]$ nc 10.12.136.111 61613
CONNECT

^@
CONNECTED
session:session-QAKvouSWXhnakBEG3J7Rf4
heart-beat:0,0
server:RabbitMQ/2.8.7
version:1.0

DISCONNECT
receipt:101

^@
RECEIPT
receipt-id:101
```

通过发送DISCONNECT Frame表示向服务器发送一个断开连接请求，其中receipt表示服务器收到请求后请告知客户端，并返回一个相同的receipt-id

至此关于Stomp绝大部分概念，我们已经实践完毕，如需更详细的还请翻阅官方文档 ^_^

=====

<http://www.cnblogs.com/davidwang456/p/4449428.html>

[http://nikipore.github.io/stompest/sync.html?](http://nikipore.github.io/stompest/sync.html?highlight=stompest%20sync#module-stompest.sync.client)

[highlight=stompest%20sync#module-stompest.sync.client](http://nikipore.github.io/stompest/sync.html?highlight=stompest%20sync#module-stompest.sync.client)

STOMP Frames

STOMP是基于帧的协议，它假定底层为一个2-way的可靠流的网络协议（如TCP）。客户端和服务端通信使用STOMP帧流通讯。帧的结构看起来像：

```
COMMAND
header1:value1
header2:value2
```

```
Body^@
```

帧以command字符串开始,以EOL结束,其中包括可选回车符 (13字节),紧接着是换行符 (10字节)。command下面是0个或多个<key>:<value>格式的header条目,每个条目由EOL结束。一个空白行 (即额外EOL) 表示header结束和body开始。body连接着NULL字节。本文档中的例子将使用^@,在ASCII中用control-@表示,代表NULL字节。NULL字节可以选择跟多个EOLs。欲了解更多关于STOMP帧的详细信息,请参阅[Augmented BNF](#)节本文件。

本文档中引用的所有command 和header 名字都是大小写敏感的。

只有**SEND**, **MESSAGE**, 和**ERROR**帧有body。所有其他的帧不能有body。

- 标准header

content-length

content-type

receipt

任何客户端帧(除了CONNECT帧)都可以为receipt header指定任何值。这会让服务端应答带有RECEIPT的客户端帧的处理过程。

如果client或者server收到重复的header条目,只有第一个会被用作header条目的值。其他的值仅仅用来维持状态改变,或者被丢弃。

例如,如果client收到:

```
MESSAGE
foo:World
foo:Hello
```

^@

foo header的值为world.

大小限制

为了客户端滥用服务端的内存分配,服务端可以设置可分配的内存大小:

- 单个帧允许帧头的个数
- header中每一行的最大长度
- 帧体的大小

如果超出了这些限制，server应该向client发送一个error frame,然后关闭连接.

连接延迟

STOMP servers必须支持client快速地连接server和断开连接。这意味着server在连接重置前只允许被关闭的连接短时间地延迟.

结果就是，在socket重置前client可能不会收到server发来的最后一个frame(比如ERROR或者RECEIPTframe去应答DISCONNECTframe)

• Client Frames

client可以发送下列列表以外的frame，但是STOMP1.2 server会响应ERROR frame,然后关闭连接。

- [SEND](#)
- [SUBSCRIBE](#)
- [UNSUBSCRIBE](#)
- [BEGIN](#)
- [COMMIT](#)
- [ABORT](#)
- [ACK](#) //ACK消息
- [NACK](#)
- [DISCONNECT](#)

SUBSCRIBE 消息

SUBSCRIBE frame用于注册给定的目的地. 和SENDframe一样，SUBSCRIBEframe需要包含destination header表明client想要订阅目的地。被订阅的目的地收到的任何消息将通过MESSAGE frame发送给client。 **ack** header控制着确认模式。

例子:

```
SUBSCRIBE
id:0
destination:/queue/foo
ack:client           //ACK头

^@
```

如果server不能成功创建此次订阅，那么server将返回ERROR frame然后关闭连接。

STOMP服务器可能支持额外的服务器特定的头文件，来自定义有关订阅传递语义。

SUBSCRIBE消息的 id 头

一个单连接可以对应多个开放的servers订阅,所以必须包含id header去唯一标示这个订阅. 这个id frame可以把此次订阅与接下来的MESSAGE frame和UNSUBSCRIBE frame联系起来。

在相同的连接中，不同的订阅必须拥有不同订阅id。

SUBSCRIBE消息的 ack 头

ack header可用的值有auto, client, client-individual, 默认为auto.

1. 当ack为auto时，client收到server发来的消息后不需要回复ACK frame. server假定消息发出去后client就已经收到。这种确认方式可以减少消息传输的次数.
2. 当ack为client时, client必须发送ACK frame给servers, 让它处理消息. 如果在client发送ACK frame之前连接断开了，那么server将假设消息没有被处理，可能会再次发送消息给另外的客户端。

client发送的ACK frame被当作是积累的确认。这就意味这种确认方式会去操作ACK frame指定的消息和订阅的所有消息.

譬如接收了10条消息，如果你ack了第8条消息，那么1-7条消息都会

被ack，只有9-10两条消息还保持未ack状态。

由于client不能处理某些消息，所以client应该发送NACK frame去告诉server它不能消费这些消息。

1. 当ack模式是client-individual，确认工作就像上面的'client'确认模式(除了由客户端发送的ACK或NACK帧)不会被累计。这意味着，后续ACK, NACK消息帧，也不能影响前面的消息的确认。

ACK 消息

用来确认订阅的消息被消费了.

```
ACK
id:12345
transaction:tx1
```

^@

• Server Frames

server偶尔也会发送frame给客户端(除了连接最初的CONNECTED frame).

这些frames为: * [MESSAGE](#) * [RECEIPT](#) * [ERROR](#)

RECEIPT帧

server成功处理带有receipt头的client frame后， 将发送RECEIPT frame到client. RECEIPT frame必须包含receipt-id header,它的值为client frame中receipt header的值。

```
RECEIPT
receipt-id:message-12345
```

^@

RECEIPT frame是作为server处理的client frame后的应答. 既然STOMP是基于流的，那么receipt也是对server已经收到所有的frames的累积确认。但是，

以前的frames可能并没有被完全处理。如果clients断开连接，以前接收到的frames应该继续被server处理。

=====

Examples

If you use ActiveMQ to run these examples, make sure you enable the STOMP connector, (see [here](#) for details):

```
<!-- add this to the config file "activemq.xml" -->
<transportConnector name="stomp" uri="stomp://0.0.0.0:61613"/>
```

For debugging purposes, it is highly recommended to turn on the logger on level DEBUG:

```
import logging
logging.basicConfig()
logging.getLogger().setLevel(logging.DEBUG)
```

Producer

```
from stompest.config import StompConfig
from stompest.sync import Stomp

CONFIG = StompConfig('tcp://localhost:61613')
QUEUE = '/queue/test'

if __name__ == '__main__':
    client = Stomp(CONFIG)
    client.connect()
    client.send(QUEUE, 'test message 1'.encode())
    client.send(QUEUE, 'test message 2'.encode())
    client.disconnect()
```

Consumer

```
from stompest.config import StompConfig
```

```

from stompest.protocol import StompSpec
from stompest.sync import Stomp
from stompest.error import StompConnectionError, StompProtocolError
import traceback

CONFIG = StompConfig('tcp://localhost:61613')
QUEUE = '/queue/test'

if __name__ == '__main__':
    client = Stomp(CONFIG)
    client.connect()
    client.subscribe(QUEUE, {StompSpec.ACK_HEADER:
StompSpec.ACK_CLIENT_INDIVIDUAL})    //客户端的ack模式为client_individual
    while True:
        try:
            if self.client.canRead(timeout=5):
                frame = client.receiveFrame()    //If we are not connected,
this method will raise a StompConnectionError. Keep in mind that this
method will block forever if there are no frames incoming on the wire. Be
sure to use peek with self.canRead(timeout) before!
                print('Got %s' % frame.info())
                client.ack(frame)
            except (StompConnectionError, StompProtocolError) as e:
                self.client.connect()
            except Exception as e:
                self.logger.debug(traceback.format_exc())
        client.disconnect()

```