# 实现UDP单播和广播

UDP 这样的无连接协议中，并没有持久化连接这样的概念，并且每个消息（一个UDP 数据报）都是一个单独的传输单元。此外，UDP 也没有TCP 的纠错机制。

通过类比，TCP 连接就像打电话，其中一系列的有序消息将会在两个方向上流动。相反，UDP 则类似于往邮箱中投入一叠明信片。你无法知道它们将以何种顺序到达它们的目的地，或者它们是否所有的都能够到达它们的目的地。

UDP的这些方面可能会让你感觉到严重的局限性，但是它们也解释了为何它会比TCP快那么多：所有的握手以及消息管理机制的开销都已经被消除了。显然，UDP很适合那些能够处理或者容忍消息丢失的应用程序，但可能不适合那些处理金融交易的应用程序。

本身作为无连接的不可靠的传输协议（适合频繁发送较小的数据包），他不会对数据包进行合并发送（也就没有Nagle算法之说了），他直接是一端发送什么数据，直接就发出去了，既然他不会对数据合并，每一个数据包都是完整的（数据+UDP头+IP头等等发一次数据封装一次）也就**没有粘包**一说了。

单播的传输模式，定义为发送消息给一个由唯一的地址所标识的单一的网络目的地。面向连接的协议和无连接协议都支持这种模式。

广播——传输到网络（或者子网）上的所有主机。

## Netty 的UDP 相关类

interface **AddressedEnvelope**<M, A extends SocketAddress>extends ReferenceCounted

定义一个消息，其包装了另一个消息并带有发送者和接收者地址。其中M 是消息类型；A 是

地址类型

class **DefaultAddressedEnvelope**<M, A extends SocketAddress>implements AddressedEnvelope<M,A>提供了interface AddressedEnvelope的默认实现

class **DatagramPacket** extends DefaultAddressedEnvelope<ByteBuf, InetSocketAddress> implements ByteBufHolder

扩展了DefaultAddressedEnvelope 以使用ByteBuf 作为消息数据容器。DatagramPacket是final类不能被继承，只能被使用。

**通过content()来获取消息内容**

**通过sender();来获取发送者的消息**

**通过recipient();来获取接收者的消息。**

interface **DatagramChannel** extends Channel

扩展了Netty 的Channel 抽象以支持UDP 的多播组管理

class **NioDatagramChannnel** extends AbstractNioMessageChannel implements DatagramChannel

定义了一个能够发送和接收Addressed-Envelope 消息的Channel 类型

Netty 的DatagramPacket 是一个简单的消息容器，DatagramChannel 实现用它来和远程节点通信。类似于在我们先前的类比中的明信片，它包含了接收者（和可选的发送者）的地址

以及消息的有效负载本身。

## UDP单播

看代码:udp工程下unicast包

## UDP广播

看代码:udp工程下broadcast包

# 服务器推送技术-短轮询和Comet

服务器推送技术干嘛用？就是让用户在使用网络应用的时候，不需要一遍又一遍的去手动刷新就可以及时获得更新的信息。大家平时在上各种视频网站时，对视频节目进行欢乐的吐槽和评论，会看到各种弹幕，当然，他们是用flash技术实现的，对于我们没有用flash的应用，一样可以实现弹幕。又比如在股票网站，往往可以看到，各种股票信息的实时刷新，上面的这些都是基于服务器推送技术。

## Ajax短轮询

就是用一个定时器不停的去网站上请求数据。

## Comet

“[服务器推](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8%E6%8E%A8)”是一种很早就存在的技术，以前在实现上主要是通过客户端的套接口，或是服务器端的远程调用。因为浏览器技术的发展比较缓慢，没有为“服务器推”的实现提供很好的支持，在纯浏览器的应用中很难有一个完善的方案去实现“服务器推”并用于商业程序。，因为 AJAX 技术的普及，gmail等等在实现中使用了这些新技术；同时“服务器推”在现实应用中确实存在很多需求。称这种基于 HTTP[长连接](https://baike.baidu.com/item/%E9%95%BF%E8%BF%9E%E6%8E%A5)、无须在浏览器端安装插件的“服务器推”技术为“Comet”。

#### 基于 AJAX 的长轮询

##### DeferredResult:

Spring mvc的控制层接收用户的请求之后，如果要采用异步处理，那么就要返回DeferedResult<>泛型对象。在调用完控制层之后，立即回返回DeferedResult对象，此时驱动控制层的容器主线程，可以处理更多的请求。

可以将DeferedResult对象作为真实响应数据的代理，而真实的数据是该对象的成员变量result，它可以是String类型，或者ModelAndView类型等。

业务处理完毕之后，要执行setResult方法，将真实的响应数据赋值到DeferedResult对象中。此时，异步线程会唤醒容器主线程。那么容器主线程会继续执行getResult方法，将真实数据响应到客户端。

#### SSE

严格地说，[HTTP 协议](http://www.ruanyifeng.com/blog/2016/08/http.html)无法做到服务器主动推送信息。但是，有一种变通方法，就是服务器向客户端声明，接下来要发送的是流信息（streaming）。

也就是说，发送的不是一次性的数据包，而是一个数据流，会连续不断地发送过来。这时，客户端不会关闭连接，会一直等着服务器发过来的新的数据流，视频播放就是这样的例子。本质上，这种通信就是以流信息的方式，完成一次用时很长的下载。

SSE 就是利用这种机制，使用流信息向浏览器推送信息。它基于 HTTP 协议，目前除了 IE/Edge，其他浏览器都支持。

SSE 与 WebSocket 作用相似，都是建立浏览器与服务器之间的通信渠道，然后服务器向浏览器推送信息。

总体来说，WebSocket 更强大和灵活。因为它是全双工通道，可以双向通信；SSE 是单向通道，只能服务器向浏览器发送，因为流信息本质上就是下载。如果浏览器向服务器发送信息，就变成了另一次 HTTP 请求。

SSE 也有自己的优点。

* SSE 使用 HTTP 协议，现有的服务器软件都支持。WebSocket 是一个独立协议。
* SSE 属于轻量级，使用简单；WebSocket 协议相对复杂。
* SSE 默认支持断线重连，WebSocket 需要自己实现。
* SSE 一般只用来传送文本，二进制数据需要编码后传送，WebSocket 默认支持传送二进制数据。
* SSE 支持自定义发送的消息类型。

##### HTTP 头信息

服务器向浏览器发送的 SSE 数据，必须是 UTF-8 编码的文本，具有如下的 **HTTP 头信息。**

Content-Type: text/event-stream

Cache-Control: no-cache

Connection: keep-alive

上面三行之中，第一行的Content-Type必须指定 MIME 类型为event-steam。

##### 信息格式

每一次发送的信息，由若干个message组成，每个message之间用\n\n分隔。每个message内部由若干行组成，每一行都是如下格式。

[field]: value\n

上面的field可以取四个值。

* data
* event
* id
* retry

此外，还可以有冒号开头的行，表示注释。通常，服务器每隔一段时间就会向浏览器发送一个注释，保持连接不中断。例子 : this is a test stream\n\n

###### data 字段

数据内容用data字段表示。

data: message\n\n

如果数据很长，可以分成多行，最后一行用\n\n结尾，前面行都用\n结尾。

data: begin message\n

data: continue message\n\n

下面是一个发送 JSON 数据的例子。

data: {\n

data: "foo": "bar",\n

data: "baz", 555\n

data: }\n\n

###### id 字段

数据标识符用id字段表示，相当于每一条数据的编号。

id: msg1\n

data: message\n\n

浏览器用lastEventId属性读取这个值。一旦连接断线，浏览器会发送一个 HTTP 头，里面包含一个特殊的Last-Event-ID头信息，将这个值发送回来，用来帮助服务器端重建连接。因此，这个头信息可以被视为一种同步机制。

###### event 字段

event字段表示自定义的事件类型，默认是message事件。浏览器可以用addEventListener()监听该事件。

event: foo\n

data: a foo event\n\n

data: an unnamed event\n\n

event: bar\n

data: a bar event\n\n

上面的代码创造了三条信息。第一条的名字是foo，触发浏览器的foo事件；第二条未取名，表示默认类型，触发浏览器的message事件；第三条是bar，触发浏览器的bar事件。

###### retry 字段

服务器可以用retry字段，指定浏览器重新发起连接的时间间隔。

retry: 10000\n

两种情况会导致浏览器重新发起连接：一种是时间间隔到期，二是由于网络错误等原因，导致连接出错。

## 技术比较

**京东用的什么？Ajax短轮询，**这说明什么？这些技术并没有什么优劣之分，只有合不合适业务的问题。京东的痛点是什么？要用有限的资源来为千万级甚至上亿的用户提供服务，如果是用长连接，对于接入的服务器，比如说Nginx，是很大的压力，光是为用户维持这个长连接都需要成百上千的Nginx的服务器，这是很划不来的。因为对于京东这类购物网站来说，用户的浏览查询量是远远大于用户下单量的，京东需要注重的是服务更多的用户，而且相对于用户浏览页面的图片等等的流量而言，这点带宽浪费占比是很小的。所以我们看京东的付款后的实现，是用的短轮询机制，而且时长放大到了5秒。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **短轮询** | **Serlvet异步** | **SSE** | **WebSocket** |
| 浏览器支持度 | 最高 | 高 | 中 | 低 |
| 实时性 | 最低 | 较高 | 高 | 最高 |
| 代码实现复杂度 | 最容易 | 较容易 | 容易 | 最复杂 |

SSE和WebSocket相比的优势。最大的优势就是便利：不需要添加任何新组件，用任何你习惯的后端语言和框架就能继续使用。你不用为新建虚拟机、弄一个新的IP或新的端口号而劳神，就像在现有网站中新增一个页面那样简单。可以称为既存基础设施优势。

SSE的第二个优势是服务端的简洁。相对而言，WebSocket则很复杂，不借助辅助类库基本搞不定。WebSocket能做的，SSE也能做，反之亦然，但在完成某些任务方面，它们各有千秋。WebSocket是一种更为复杂的服务端实现技术，但它是真正的双向传输技术，既能从服务端向客户端推送数据，也能从客户端向服务端推送数据。

# WebSocket通信

**什么是WebSocket？**

WebSocket ——一种在2011 年被互联网工程任务组（IETF）标准化的协议。

WebSocket解决了一个长期存在的问题：既然底层的协议（HTTP）是一个请求/响应模式的交互序列，那么如何实时地发布信息呢？AJAX提供了一定程度上的改善，但是数据流仍然是由客户端所发送的请求驱动的。还有其他的一些或多或少的取巧方式(Comet)

WebSocket规范以及它的实现代表了对一种更加有效的解决方案的尝试。简单地说，WebSocket提供了“在一个单个的TCP连接上提供双向的通信……结合WebSocket API……它为网页和远程服务器之间的双向通信提供了一种替代HTTP轮询的方案。”

，但是最终它们仍然属于扩展性受限的变通之法。也就是说，WebSocket 在客户端和服务器之间提供了真正的双向数据交换。WebSocket 连接允许客户端和服务器之间进行全双工通信，以便任一方都可以通过建立的连接将数据推送到另一端。WebSocket 只需要建立一次连接，就可以一直保持连接状态。这相比于轮询方式的不停建立连接显然效率要大大提高。

Web浏览器和服务器都必须实现 WebSockets 协议来建立和维护连接。

**特点**

* HTML5中的协议，实现与客户端与服务器双向，基于消息的文本或二进制数据通信
* 适合于对数据的实时性要求比较强的场景，如通信、直播、共享桌面，特别适合于客户与服务频繁交互的情况下，如实时共享、多人协作等平台。
* 采用新的协议，后端需要单独实现
* 客户端并不是所有浏览器都支持

# WebSocket通信握手

**客户端的请求：**

Connection 必须设置 Upgrade，表示客户端希望连接升级。

Upgrade 字段必须设置 Websocket，表示希望升级到 Websocket 协议。

Sec-WebSocket-Key 是随机的字符串，服务器端会用这些数据来构造出一个 SHA-1 的信息摘要。把 “Sec-WebSocket-Key” 加上一个特殊字符串 “258EAFA5-E914-47DA-95CA-C5AB0DC85B11”，然后计算 SHA-1 摘要，之后进行 BASE-64 编码，将结果做为 “Sec-WebSocket-Accept” 头的值，返回给客户端。如此操作，可以尽量避免普通 HTTP 请求被误认为 Websocket 协议。

Sec-WebSocket-Version 表示支持的 Websocket 版本。RFC6455 要求使用的版本是 13，之前草案的版本均应当弃用。

**服务器端：**

Upgrade: websocket

Connection: Upgrade

依然是固定的，告诉客户端即将升级的是 Websocket 协议，而不是mozillasocket，lurnarsocket或者shitsocket。

然后， Sec-WebSocket-Accept 这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key 。

后面的， Sec-WebSocket-Protocol 则是表示最终使用的协议。

至此，HTTP已经完成它所有工作了，接下来就是完全按照Websocket协议进行

# WebSocket通信-STOMP

WebSocket是个规范，在实际的实现中有HTML5规范中的WebSocket API、WebSocket的子协议STOMP。

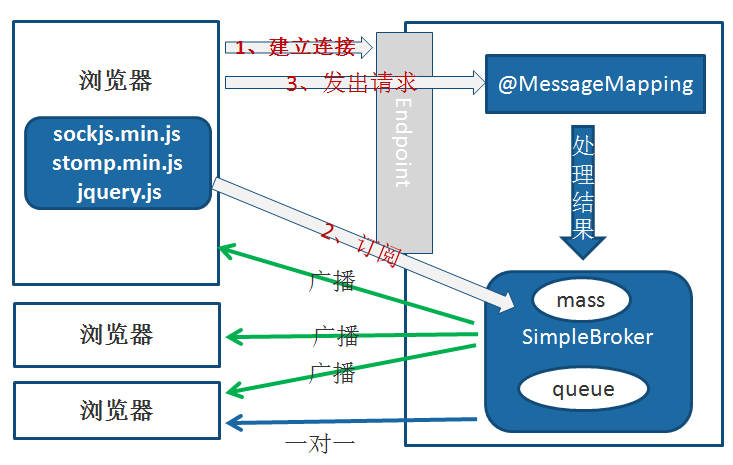
STOMP(Simple Text Oriented Messaging Protocol)

* 简单(流)文本定向消息协议
* 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。是属于消息队列的一种协议, 和AMQP, JMS平级. 它的简单性恰巧可以用于定义websocket的消息体格式. STOMP协议很多MQ都已支持, 比如RabbitMq, ActiveMq。
* 生产者（发送消息）、消息代理、消费者（订阅然后收到消息）
* STOMP是基于帧的协议

# WebSocket通信实现

## SpringBoot

#### 基于Stomp的聊天室/IM的实现



具体实现：参考 stomp模块下的代码

和WebSocket的集成

#### 和WebSocket的集成

具体实现：参考 websocket模块下的代码

## Netty

由IETF 发布的WebSocket RFC，定义了6 种帧，Netty 为它们每种都提供了一个POJO 实现。



看代码

# 高级通信服务实现-设计自己的协议栈

## 定义

通信协议从广义上区分，可以分为公有协议和私有协议。由于私有协议的灵活性，它往往会在某个公司或者组织内部使用，按需定制，也因为如此，升级起来会非常方便，灵活性好。绝大多数的私有协议传输层都基于TCP/IP，所以利用Netty的NIO TCP协议栈可以非常方便地进行私有协议的定制和开发。

私有协议本质上是厂商内部发展和采用的标准，除非授权，其他厂商一般无权使用该协议。私有协议也称非标准协议，就是未经国际或国家标准化组织采纳或批准，由某个企业自己制订，协议实现细节不愿公开，只在企业自己生产的设备之间使用的协议。私有协议具有封闭性、垄断性、排他性等特点。

## 跨节点通信

在传统的Java应用中，通常使用以下4种方式进行跨节点通信。

（1）通过RMI进行远程服务调用；

（2）通过Java的Socket+Java序列化的方式进行跨节点调用；

（3）利用一些开源的RPC框架进行远程服务调用，例如Facebook的Thrift，Apache的Avro等；

（4）利用标准的公有协议进行跨节点服务调用，例如HTTP+XML、RESTful+JSON或者WebService。

跨节点的远程服务调用，除了链路层的物理连接外，还需要对请求和响应消息进行编解码。在请求和应答消息本身以外，也需要携带一些其他控制和管理类指令，例如链路建立的握手请求和响应消息、链路检测的心跳消息等。当这些功能组合到一起之后，就会形成私有协议。

## 协议栈功能设计

#### 协议栈功能描述

Netty协议栈承载了业务内部各模块之间的消息交互和服务调用，它的主要功能如下。

（1）基于Netty的NIO通信框架，提供高性能的异步通信能力；

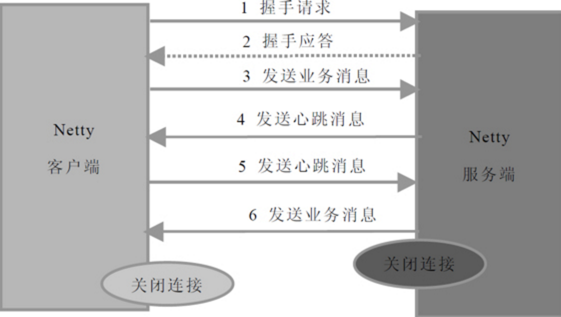
（2）提供消息的编解码框架，可以实现POJO的序列化和反序列化；

（3）提供基于IP地址的白名单接入认证机制；

（4）链路的有效性校验机制；

（5）链路的断连重连机制。

#### 通信模型



（1）Netty协议栈客户端发送握手请求消息，携带节点ID等有效身份认证信息；

（2）Netty协议栈服务端对握手请求消息进行合法性校验，包括节点ID有效性校验、节点重复登录校验和IP地址合法性校验，校验通过后，返回登录成功的握手应答消息；

（3）链路建立成功之后，客户端发送业务消息；

（4）链路成功之后，服务端发送心跳消息；

（5）链路建立成功之后，客户端发送心跳消息；

（6）链路建立成功之后，服务端发送业务消息；

（7）服务端退出时，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接。

备注：需要指出的是，Netty协议通信双方链路建立成功之后，双方可以进行全双工通信，无论客户端还是服务端，都可以主动发送请求消息给对方，通信方式可以是TWO WAY或者ONE WAY。双方之间的心跳采用Ping-Pong机制，当链路处于空闲状态时，客户端主动发送Ping消息给服务端，服务端接收到Ping消息后发送应答消息Pong给客户端，如果客户端连续发送N条Ping消息都没有接收到服务端返回的Pong消息，说明链路已经挂死或者对方处于异常状态，客户端主动关闭连接，间隔周期T后发起重连操作，直到重连成功。

#### 消息定义

Netty协议栈消息定义包含两部分：

消息头；消息体。

Netty消息定义表

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 长度 | 描述 |
| Header | Header | 变长 | 消息头定义 |
| Body | Object | 变长 | 对于请求消息，它只是方法的参数，对于响应消息，它是返回值 |

Netty协议消息头定义（Header）

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 长度 | 描述 |
| crcCode | Int | 32 | Netty消息校验码 |
| Length | Int | 32 | 整个消息长度 |
| sessionID | Long | 64 | 会话ID |
| Type | Byte | 8 | 0:业务请求消息  1：业务响应消息  2：业务one way消息  3握手请求消息  4握手应答消息  5：心跳请求消息  6：心跳应答消息 |
| Priority | Byte | 8 | 消息优先级：0~255 |
| Attachment | Map<String,Object> | 变长 | 可选字段，由于推展消息头 |

#### 链路的建立

Netty协议栈支持服务端和客服端，对于使用Netty协议栈的应用程序而言，不需要刻意区分到底是客户端还是服务器端，在分布式组网环境中，一个节点可能既是客户端也是服务器端，这个依据具体的用户场景而定。

Netty协议栈对客户端的说明如下：如果A节点需要调用B节点的服务，但是A和B之间还没有建立物理链路，则有调用方主动发起连接，此时，调用方为客户端，被调用方为服务端。

考虑到安全，链路建立需要通过基于Ip地址或者号段的黑白名单安全认证机制，作为样例，本协议使用基于IP地址的安全认知，如果有多个Ip，通过逗号进行分割。在实际的商用项目中，安全认证机制会更加严格，例如通过密钥对用户名和密码进行安全认证。

客户端与服务端链路建立成功之后，由客户端发送握手请求消息，握手请求消息的定义如下

（1） 消息头的type字段值为3；

（2） 可选附件数为0；

（3） 消息头为空

（4） 握手消息的长度为22个字节

服务端接收到客户端的握手请求消息之后，如果IP校验通过，返回握手成功应答消息给客户端，应用层链路建立成功。握手应答消息定义如下：

（1）消息头的type字段值为4

（2）可选附件个数为0；

（3）消息体为byte类型的结果，0：认证成功；-1认证失败；

链路建立成功之后，客户端和服务端就可以互相发送业务消息了。

#### 链路的关闭

由于采用长连接通信，在正常的业务运行期间，双方通过心跳和业务消息维持链路，任何一方都不需要主动关闭连接。

但是，在以下情况下，客户端和服务端需要关闭连接：

（1）当对方宕机或者重启时，会主动关闭链路，另一方读取到操作系统的通知信号得知对方REST链路，需要关闭连接，释放自身的句柄等资源。由于采用TCP全双工通信，通信双方都需要关闭连接，释放资源；

（2）消息读写过程中，发生了I/O异常，需要主动关闭连接；

（3）心跳消息读写过程发生了I/O异常，需要主动关闭连接；

（4）心跳超时，需要主动关闭连接；

（5）发生编码异常等不可恢复错误时，需要主动关闭连接。

#### 可靠性设计

Netty协议栈可能会运行在非常恶劣的网络环境中，网络超时、闪断、对方进程僵死或者处理缓慢等情况都有可能发生。为了保证在这些极端异常场景下Netty协议栈仍能够正常工作或者自动恢复，需要对他的可靠性进行统一规划和设计。

##### 心跳机制

在凌晨等业务低谷时段，如果发生网络闪断、连接被Hang住等问题时，由于没有业务消息，应用程序很难发现。到了白天业务高峰期时，会发生大量的网络通信失败，严重的会导致一段时间进程内无法处理业务消息。为了解决这个问题，在网络空闲时采用心跳机制来检测链路的互通性，一旦发现网络故障，立即关闭链路，主动重连。

当读或者写心跳消息发生I/O异常的时候，说明已经中断，此时需要立即关闭连接，如果是客户端，需要重新发起连接。如果是服务端，需要清空缓存的半包信息，等到客户端重连。

##### 重连机制

如果链路中断，等到INTEVAL时间后，由客户端发起重连操作，如果重连失败，间隔周期INTERVAL后再次发起重连，直到重连成功。

为了保持服务端能够有充足的时间释放句柄资源，在首次断连时客户端需要等待INTERVAL时间之后再发起重连，而不是失败后立即重连。

为了保证句柄资源能够及时释放，无论什么场景下重连失败，客户端必须保证自身的资源被及时释放，包括但不现居SocketChannel、Socket等。

重连失败后，需要打印异常堆栈信息，方便后续的问题定位。

##### 重复登录保护

当客户端握手成功之后，在链路处于正常状态下，不允许客户端重复登录，以防止客户端在异常状态下反复重连导致句柄资源被耗尽。

服务端接收到客户端的握手请求消息之后，首先对IP地址进行合法性校验，如果校验成功，在缓存的地址表中查看客户端是否已经登录，如果登录，则拒绝重复登录，返回错误码-1，同时关闭TCP链路，并在服务端的日志中打印握手失败的原因。

客户端接收到握手失败的应答消息之后，关闭客户端的TCP连接，等待INTERVAL时间之后，再次发起TCP连接，知道认证成功。

为了防止由服务端和客户端对链路状态理解不一致导致的客户端无法握手成功问题，当服务端连续N次心跳超时之后需要主动关闭链路，清空改客户端的地址缓存信息，以保证后续改客户端可以重连成功，防止被重复登录保护机制拒绝掉。

##### 测试

1. 正常情况
2. 客户端宕机，服务器应能清除客户端的缓存信息，允许客户端重新登录
3. 服务器宕机，客户端应能发起重连

4、在LoginAuthRespHandler中进行注释，可以模拟当服务器不处理客户端的请求时，客户端在超时后重新进行登录。