# **ZeroMQ简介**

mq是消息队列（message queue）的简称，目前有多种消息队列可用，包括RabbitMQ、Kafka等，它们各有特色，可以结合具体的项目需求使用。

ZeroMQ简称Zmq，或者0mq，核心引擎由c++编写，是基于消息队列的多线程网络库库，在对传统的标准socket接口扩展的基础上形成的特色消息通信中间件，其对套接字类型、连接处理、帧、甚至路由的底层细节进行抽象，提供跨越多种传输协议的套接字。

Zmq提供了异步消息队列的抽象，具有多种消息通信模式，能够实现消息过滤，能够无缝对接多种传输协议。

简言之，使用socket时，需要显式地建立连接、销毁连接、选择协议（TCP/UDP）和处理错误等，而ZMQ屏蔽了这些细节，让网络编程更简单。

ZeroMQ是网络通信中新的一层，介于应用层和传输层之间（按照TCP/IP划分），是一个可伸缩层，可并行运行，分散在分布式系统间。

ZMQ不是单独的服务，ZMQ看起来像是一套嵌入式的网络链接库，但工作起来更像是一个并发式的框架。它封装了网络通信、消息队列、线程调度等功能，向上层提供简洁的API，应用程序通过加载库文件，调用API函数来实现高性能网络通信。

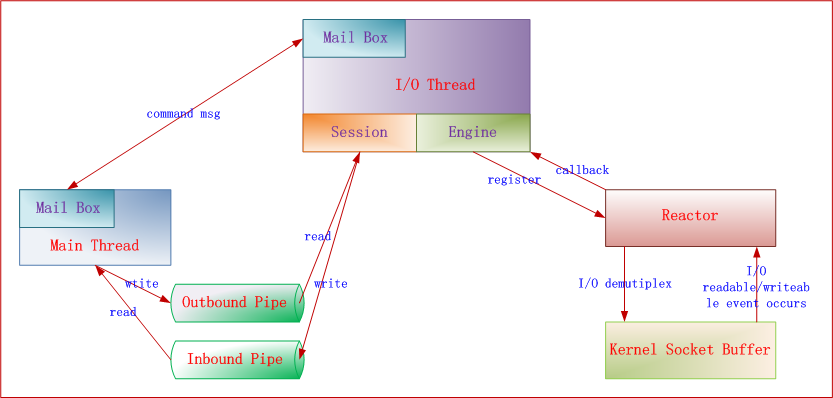
它提供的套接字可以在多种协议中传输消息，如线程间、进程间、TCP、广播等。通信协议配置简单，用类似于URL形式的字符串指定即可，格式分别为inproc://、ipc://、tcp://、pgm://。ZeroMQ会自动根据指定的字符串解析出协议、地址、端口号等信息。你可以使用套接字构建多对多的连接模式，如扇出、发布-订阅、任务分发、请求-应答等。ZMQ的快速足以胜任集群应用产品。它的异步I/O机制让你能够构建多核应用程序，完成异步消息处理任务。ZMQ有着多语言支持，并能在几乎所有的操作系统上运行。

### 主线程与I/O线程：

I/O线程，ZMQ根据用户调用zmq\_init函数时传入的参数，创建对应数量的I/O线程。每个I/O线程都有与之绑定的Poller，Poller采用经典的Reactor模式实现。

Poller根据不同操作系统平台使用不同的网络I/O模型（select、poll、epoll、devpoll、kequeue等），所有的I/O操作都是异步的，线程不会被阻塞。。

主线程与I/O线程通过Mail Box传递消息来进行通信。



Server，在主线程创建zmq\_listener，通过Mail Box发消息的形式将其绑定到I/O线程，I/O线程把zmq\_listener添加到Poller中用以侦听读事件。

Client，在主线程中创建zmq\_connecter，通过Mail Box发消息的形式将其绑定到I/O线程，I/O线程把zmq\_connecter添加到Poller中用以侦听写事件。

Client与Server第一次通信时，会创建zmq\_init来发送identity，用以进行认证。认证结束后，双方会为此次连接创建Session，以后双方就通过Session进行通信。

每个Session都会关联到相应的读/写管道， 主线程收发消息只是分别从管道中读/写数据。Session并不实际跟kernel交换I/O数据，而是通过plugin到Session中的Engine来与kernel交换I/O数据。

## 消息模型

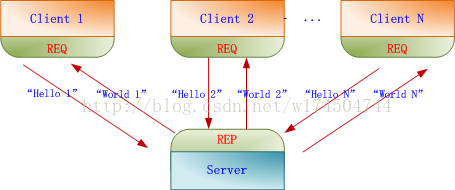
ZeroMQ将消息通信分成4种模型，分别是一对一结对模型（Exclusive-Pair）、请求回应模型（Request-Reply）、发布订阅模型（Publish-Subscribe）、推拉模型（Push-Pull）。这4种模型总结出了通用的网络通信模型，在实际中可以根据应用需要，组合其中的2种或多种模型来形成自己的解决方案。

#### 1.一对一结对模型 PAIR

最简单的1:1消息通信模型，可以认为是一个TCP Connection，但是TCP Server只能接受一个连接。数据可以双向流动，这点不同于后面的请求回应模型。

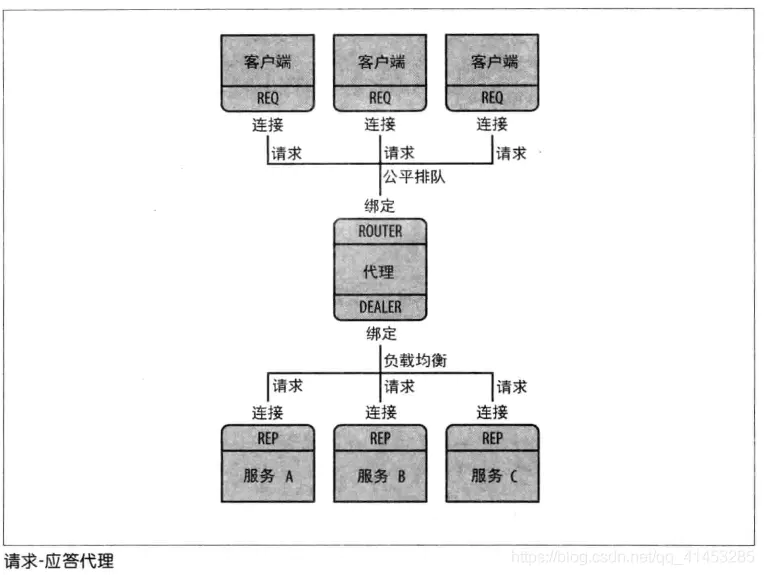
#### 请求回应模型 REQ/REP Router/Dealer

由请求端发起请求，然后等待回应端应答。一个请求必须对应一个回应，从请求端的角度来看是发-收配对，从回应端的角度是收-发对。跟一对一结对模型的区别在于请求端可以是1~N个。该模型主要用于远程调用及任务分配等。Echo服务就是这种经典模型的应用。



Router/Dealer

用于扩展REQ/REP套接字的高级模式。



多对多的网络中，Router/Dealer 模式很有用。假设我们有 N 个 Reply server，M 个 Request client，若要保证高可用性，正常而言，双方需要一个 M x N 的 full mesh 的网络才能保证任何一个 client 能够和任何一个 server 建立连接。通过在中间加一层 Router/Dealer，M x N 的连接被简化成 M + N。网络的复杂度大大降低。

**注意事项**：

使用REQ-REP套接字发送和接受消息是需要遵循一定规律的。客户端首先使用zmq\_send()发送消息，再用zmq\_recv()接收，如此循环。如果打乱了这个顺序（如连续发送两次）则会报错。类似地，服务端必须先进行接收，后进行发送。

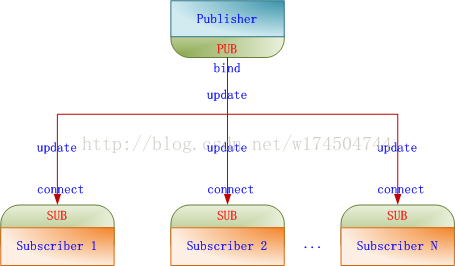
理论上你可以连接千万个客户端到这个服务端上，同时连接都没问题，程序仍会运作得很好。你可以尝试一下先打开客户端，再打开服务端，可以看到程序仍然会正常工作。

对于字符串，需要注意的一点是，ZMQ不会关心发送消息的内容，只要知道它所包含的字节数。这意味着，ZMQ的字符串是有长度的，且传送时不加结束符。当使用c语言接收时，应注意申请比长度多一个字节的存储空间，并置位结束符’/0’，否则在打印字符串时可能得到奇怪的结果。

#### 发布订阅模型

Pub/Sub 是消息传输非常常见也是非常有用的一种模式，将数据的发布者和订阅者解耦 —— 发布者者只管产生数据，而不必关心谁是订阅者，有多少订阅者，发布端单向分发数据，且不关心是否把全部信息发送给订阅端。

如果发布端开始发布信息时，订阅端尚未连接上来，则这些信息会被直接丢弃。订阅端未连接导致信息丢失的问题，可以通过与请求回应模型组合来解决。订阅端只负责接收，而不能反馈，且在订阅端消费速度慢于发布端的情况下，会在订阅端堆积数据。该模型主要用于数据分发。天气预报、微博明星粉丝可以应用这种经典模型。



**注意事项：**

在使用SUB套接字时，必须使用setsockopt()方法来设置订阅的内容。如果你不设置订阅内容，那将什么消息都收不到。订阅信息可以是任何字符串，可以设置多次。只要消息满足其中一条订阅信息，SUB套接字就会收到。订阅者可以选择不接收某类消息，也是通过zmq\_setsockopt()方法实现的。

PUB-SUB套接字组合是异步的。客户端在一个循环体中使用zmq\_recv()接收消息，如果向SUB套接字发送消息则会报错；类似地，服务端可以不断地使用zmq\_send()发送消息，但不能在PUB套接字上使用zmq\_recv()。

关于PUB-SUB套接字，还有一点需要注意：你无法得知SUB是何时开始接收消息的。就算你先打开了SUB套接字，后打开PUB发送消息，这时SUB还是会丢失一些消息的，因为建立连接是需要一些时间的。很少，但并不是零。

关于如何使发布者和订阅者同步，只有当订阅者准备好时发布者才会开始发送消息的方法可以深入研究，有一种简单的方法来同步PUB和SUB，就是让PUB延迟一段时间再发送消息。现实编程中不建议使用这种方式，因为它太脆弱了，而且不好控制。

另一种同步的方式则是认为发布者的消息流是无穷无尽的，因此丢失了前面一部分信息也没有关系。

几点说明：

订阅者可以连接多个发布者，轮流接收消息；

如果发布者没有订阅者与之相连，那它发送的消息将直接被丢弃；

如果你使用TCP协议，那当订阅者处理速度过慢时，消息会在发布者处堆积。可以使用阈值（HWM）来保护发布者。

从ZeroMQ v3.x开始，当使用(tcp:// or ipc://)连接协议时，消息的过滤在发布端，使用epgm://时，在订阅端过滤。在ZeroMQ v2.x中，所有消息的过滤是在订阅者处进行的。也就是说，发布者会向订阅者发送所有的消息，订阅者会将未订阅的消息丢弃。

#### 推拉模型

使用扇入或扇出的形式组装多个节点，可以产生多个步骤或循环，用于构建并行处理架构。也称作管道模式、流水线模式。该模型主要用于多任务并行，提高任务处理效率。

Server端作为Push端，而Client端作为Pull端，如果有多个Client端同时连接到Server端，则Server端会在内部做一个负载均衡，采用平均分配的算法，将所有消息均衡发布到Client端上。与发布订阅模型相比，推拉模型在没有消费者的情况下，发布的消息不会被消耗掉；在消费者能力不够的情况下，能够提供多消费者并行消费解决方案。

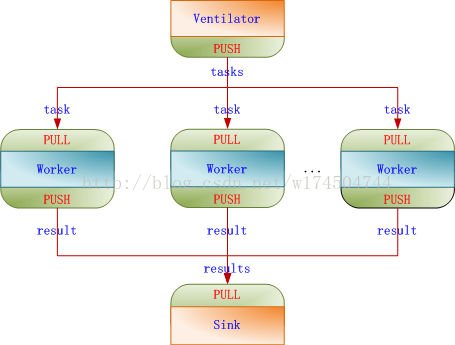
例如：

zmq的并行处理模型可以进行超级计算：

任务分发器会生成大量可以并行计算的任务

有一组worker会处理这些任务，现实中，worker可能散落在不同的计算机中，利用GPU（图像处理单元）进行复杂计算

结果收集器会在末端接收所有worker的处理结果，进行汇总



**注意事项：**

worker上游和任务分发器相连，下游和结果收集器相连，这就意味着你可以开启任意多个worker。但若worker是绑定至端点的，而非连接至端点，那我们就需要准备更多的端点，并配置任务分发器和结果收集器。所以说，任务分发器和结果收集器是这个网络结构中较为稳定的部分，因此应该由它们绑定至端点，而非worker，因为它们较为动态。

我们需要做一些同步的工作，等待worker全部启动之后再分发任务。这点在ZMQ中很重要，且不易解决。连接套接字的动作会耗费一定的时间，因此当第一个worker连接成功时，它会一下收到很多任务。所以说，如果我们不进行同步，那这些任务根本就不会被并行地执行。你可以自己试验一下。

任务分发器使用PUSH套接字向worker均匀地分发任务（假设所有的worker都已经连接上了），这种机制称为负载均衡

结果收集器的PULL套接字会均匀地从worker处收集消息，这种机制称为公平队列

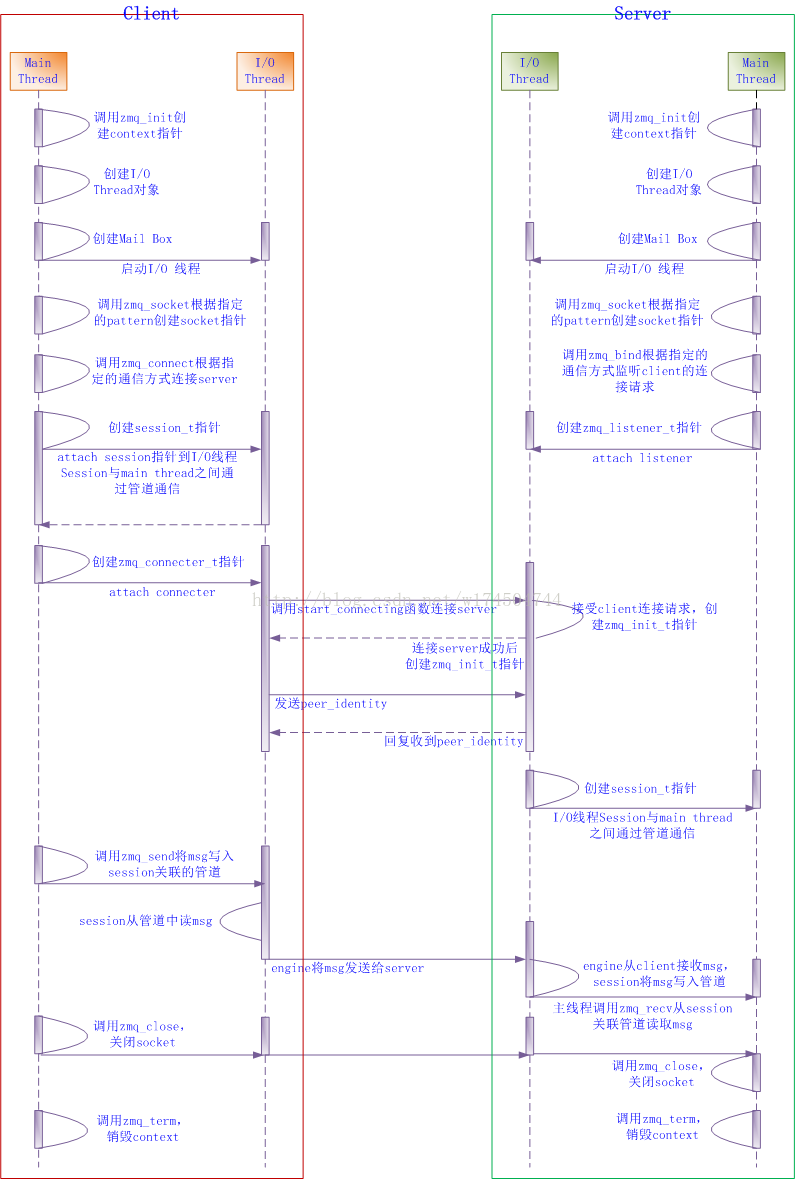
## 通信协议

提供进程内、进程间、机器间、广播等四种通信协议。通信协议配置简单，用类似于URL形式的字符串指定即可，格式分别为inproc://、ipc://、tcp://、pgm://。ZeroMQ会自动根据指定的字符串解析出协议、地址、端口号等信息。如：

zmq\_bind(responder, "tcp://\*:5555"); // server

zmq\_connect(requester, "tcp://localhost:5555"); // clinet

## 工作流程

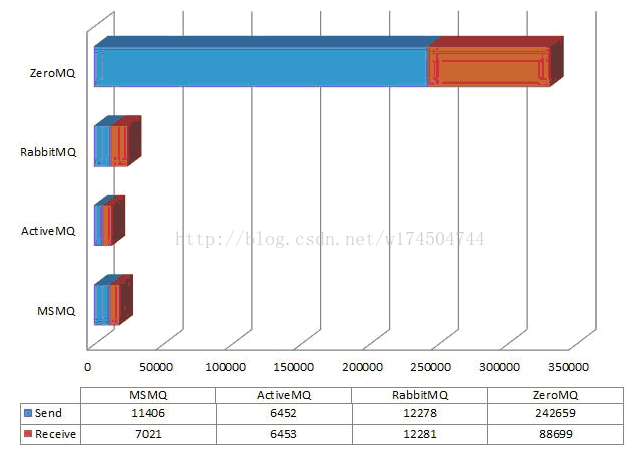


## 性能分析

目前，市面上类似的产品不少，主要有4种：MSMQ（微软产品）、ActiveMQ（Java）、RabbitMQ(Erlang)、ZeroMQ（C++）。除ZeroMQ外，其它3款产品都是一个单独服务或者进程，需要单独安装和运行，且对环境有一定依赖。其中，MSMQ在非Windows平台下安装非常复杂，ActiveMQ需要目标机器上已经安装了Java，RabbitMQ需要Erlang环境。而ZeroMQ是以库的形式存在，由应用程序加载、运行即可。但是ZeroMQ仅提供非持久性的消息队列。

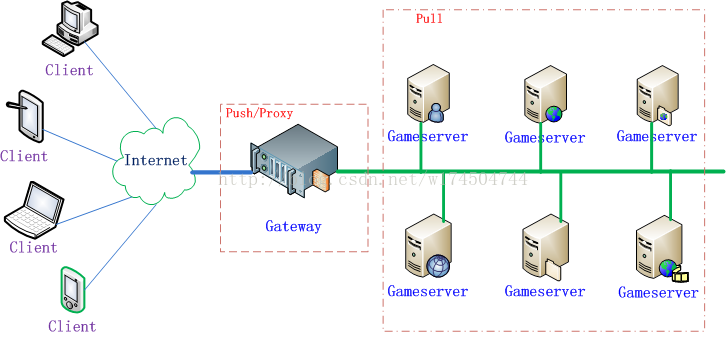
下图是来自于Internet的性能测试数据。显示的是每秒钟发送和接受的消息数。整个过程共产生1百万条1K的消息，测试环境为Windows Vista。从测试数据可以看出，ZeroMQ的性能远远高于其它3个MQ。

但是测试数据仅供参考，因为缺少必须的环境参数和性能指标，比如：CPU参数、内存参数、消息模型、通信协议、极限时消耗CPU百分比、极限时消耗内存百分比等。



## 应用场景

应用ZeroMQ的Push-Pull模型实现联众游戏服务器的“热插拔”、负载均衡和消息派发。按照如图8部署服务器，Push端充当Gateway，作为一组游戏服务器集群最上层的一个Proxy，起负载均衡的作用，所有Gameserver作为Pull端。当一个请求到达Push端（Gateway）时，Push端根据一定的分配策略将任务派发到Pull端（Gameserver）。以联众某款游戏A为例，游戏A刚上线时，预计最大同时在线人数是10W，单台Gameserver并发处理能力为1W，需要10台Gameserver，由于游戏A可玩性非常好，半个月后最大同时在线人数暴增到50W，那么不需要在某天的凌晨将Gateway和Gameserver停机，只需要随时在机房新添加40台Gameserver，启动并连接到Gateway即可。

ZeroMQ中对Client和Server的启动顺序没有要求，Gameserver之间如果需要通信的话，Gameserver的应用层不需要管理这些细节，ZeroMQ已经做了重连处理。

## Zmq特点

**简单**

仅仅提供24个API接口，风格类似于BSD Socket。

处理了网络异常，包括连接异常中断、重连等。

改变TCP基于字节流收发数据的方式，处理了粘包、半包等问题，以msg为单位收发数据，结合Protocol Buffers，可以对应用层彻底屏蔽网络通信层。

对大数据通过SENDMORE/RECVMORE提供分包收发机制。

通过线程间数据流动来保证同一时刻任何数据都只会被一个线程持有，以此实现多线程的“去锁化”。

通过高水位HWM来控制流量，用交换SWAP来转储内存数据，弥补HWM丢失数据的缺陷。

服务器端和客户端的启动没有先后顺序。

**灵活**

支持多种通信协议，可以灵活地适应多种通信环境，包括进程内、进程间、机器间、广播。

支持多种消息模型，消息模型之间可以相互组合，形成特定的解决方案。

**跨平台**

支持Linux、Windows、OS X等。

**多语言**

可以绑定C、C++、Java、.NET、Python等30多种开发语言。

**高性能**

相对同类产品，性能卓越。