**Linkage**

* Purpose And Background

When establish a big Web Site, it's inevitable to use the distribution technology. After we divide the whole system into distributed part, we could call them as services each other. Each service could provide the specific function like Sending Message to the user when Login, Writting the record into the database.

But how to call a distributed service ? Is there a need that we establish a protocol for each service calling. Answer is definitely not, we need to design a unified middleware used between the service and the service caller.

Linkage is such a middleware which could bond all of the distributed services into one whole system. See the chart below, applications, services, db, cache are placed at distributed computers. Between each is linkage. We use linkage to deal with the communication with services and applications.

Application1

Application2

Application3

Service 1

Service 2

Service 3

DB

File System

Cache

* Overall Design

Chart below show the main structure of the linkage. Client side & Service side will communicate with each other by different type of io. NIO is now most widely used, so we use this type of io. But still keep the extension for other type of io. The wapper layer placed above the io layer. In this layer, we will wrapper/unwrapper message with the io protocol. The serialization/deserialization layer placed above the wrapper layer, it will use the serialization protocol. Above this layer is the service access/provider layer. In this layer, we could also define the calling and being called protocol.

Communication

Message Wrapper

Message Unwrapper

Communication

Serialization/Deserialization

ServiceAccess

Service Provider

Serialization/Deserialization

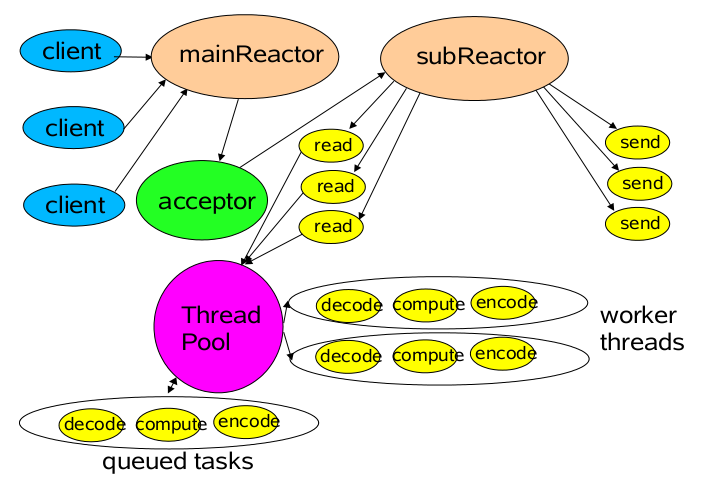
Client Side

Service Side

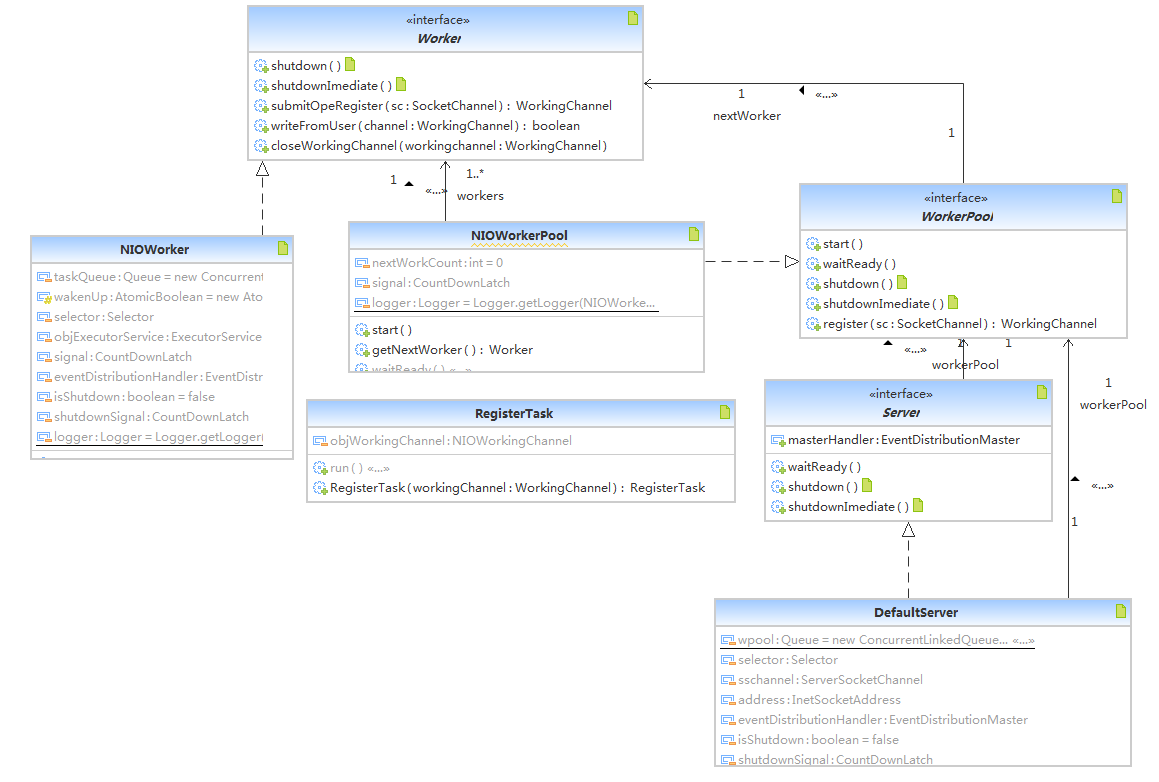
Linkage

* Communication Layer

Below chart shows the classic NIO model. There is one main reactor accept all connection then the acceptor will post the connect to the sub reactor. Sub reactor will deal with the read&write operation of the connection. Netty actually use this model for its nio case. We would use this model too in order to deal with more concurrent requrest.



Below chart is class diagram of nio part, the server act as the main reactor which will accept all of the connection request and register to the worker pool. The worker pool retains the worker and the worker is actually the sub reactor. The server will listen the port, once there is new connection coming. The server will register the channel to the worker through the worker pool.



* Wrap&Unwrap Layer

In this layer, all the messages should be wrapped before sent to the client or server. In the other side the message will be unwrapped for using. For the tcp/ip protocol, one connection is like a river, the messages are the water. When sending two messages, the messages will arrived one by one or together or part of them. Therefore we cannot point where is the border of one message. We must use one mechanism to control it. Using protocol is one effective mean. When we receive some data, we could check if it is a complete message according to the protocol. If not we need to put the data to the buffer, then wait until the data in buffer contains one message or more. Then we extract the message from the buffer.

* **IO Procotol**

In this procotol, we define the packet. We usually call it message anyway. The packet would contain header&body. Below chart is one packet structure. From the body length, we could know the length of the body. Therefore, we know the border of the packet. The Header Start is the packet start, the max length body character is the packer end. Class IOProtocol Define this protocol, we could decrease the number of characters in the Header Start & Header End in order to improve the transmission performance. But it may cause some unexpected issue if the body contains the same characters. Anyway, special means could also be used here. Please note that the Body Length part is also fixed length which means if you have body length of 7 then the Body Length could be 0007. The premises is the fixed length of body length is 4. We will have fill the blanks with 0.

Example: $#####$0007\*#####\*MESSAGE

Header Start

Header End

Body Length

Body

* Serialization/Deserialization

Of course all of the request will be serialized before sent though the net. We need to define an unified Serialization&Deserialization protocol. For simple use, we will use xml to define the request xml and the response xml. This will occupy more bandwidth, however xml is an extendable format. We could change it later if the performance become a headache.

Below is the request & response xml format.

<request>

<requestid>10000</requestid>

<serviceName>&lt;testServiceNmae&gt;</serviceName>

<methodName>test23%^Method&amp;</methodName>

<version>@@!#$test.1.0</version>

<group>testGroup</group>

<list>

<arg>arg1</arg><arg>arg2&amp;\*^%</arg><arg>arg3</arg>

</list>

</request>

<response>

<requestid>100001212</requestid>

<result>sdsjdlfkj$@^!\*#!4457@$$</result>

</response>

* Service Access&Service Provider

In this layer, we provide the service access and service provider. We could use these two function to setup the service and access the service. Client would setup information of how to access the service and the server side should also setup the service. We need to conclude the calling and being called procotol.

* Service Access

# Configure the client

client.service5.id=calculator

client.service5.name=calculator

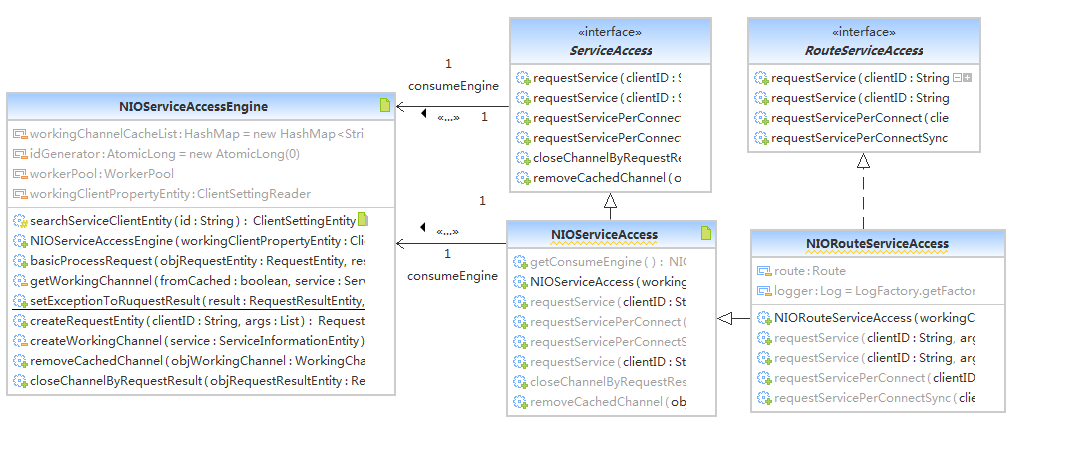
client.service5.method=add

client.service5.version=version

client.service5.group=test

Above is the client setting for the service. User could use the id to request the service. Below chart shows the class diagram, We define an engine for the nio access. In this engine we define the basic request function for the nio request. In this engine we would optionally establish the channel by caching or not. And close the channel after the request optionally.

We could also define other type of engine like bio, http as well. But the access class is required to extends the same interface(ServiceAccess, RouteServiceAccess). RouteServiceAccess is use for the Linkage center. We would talk it later on the Linkage Center paragraph.



* Service Provider

# Configure the service for the server

service.service2.name=calculator

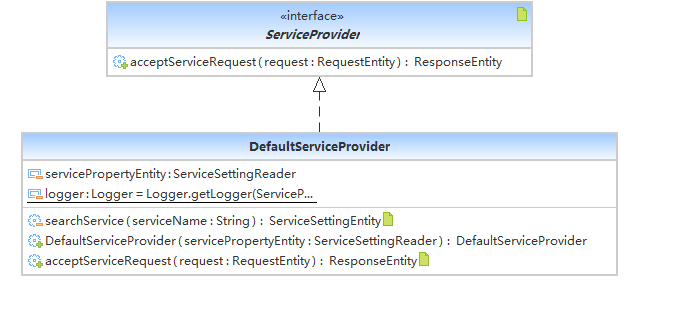
service.service2.interface=test.service.Calculator

service.service2.version=version

service.service2.group=test

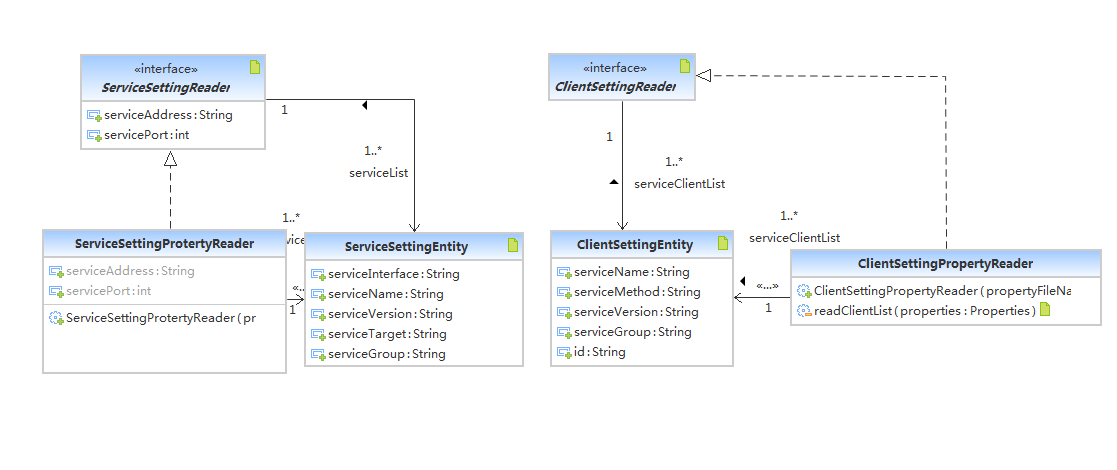
service.service2.target=test.service.CalculatorImpl

Above is the setting for service in the server. Below is the class diagram. It's easy to understand.



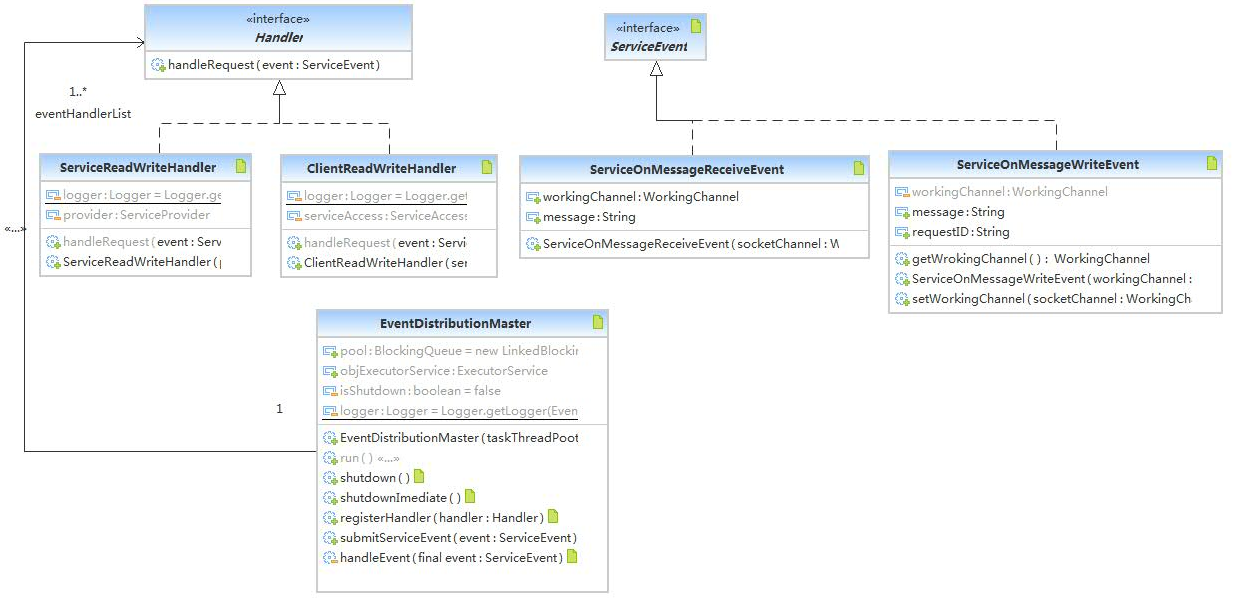
* Service Setting Reader

Client and Service Setting Reader will be responsible for reading the client and service setting. The setting could be read from different ways. We now define a property read which could read from the property file. We could also read from xml or text file.



* Event Driven Model

In the IO layer, when the message received, the IO layer will post the event to the master. The handlers will be registered to the master. There are threads running background which will deal with the event and submit the job into thread pool. IO layer won't care the event handling. It only put the event into the event pool in the master. Threads running background will fetch the event from the event pool then submit a job(event being handling by the handlers) into the thread pool. By using this model, the concurrence will be improved a lot. The message received/sent and how to deal with the message are separated.



* Linkage Center

Sometimes, there are services cluster. But we will pick up one service from the service list. When the service is available, the service will be regitered to the Linkage Center. When the service is off, the service will be unregistered. The client will get the service list from the Linkage Center.

Linkage Center

Service

Service

Register/Unregister

Client

Fetch Service List

Service

* [Performance](javascript:void(0);) [index](javascript:void(0);)

Performance is