NS4开发指南

1.框架介绍

NS4框架本质是两个应用加三套开发框架组合起来的分布式业务框架

1.1 使用范围

NS4框架主要适用于业务类型为消息流或者业务核心模型为流式业务的业务系统。它支持消息分发，传递，追踪，支持分步骤，分批次的消息处理，对于信息流，数据流等消息驱动型的业务尤为契合。

1.2 项目结构

NS4框架是一套MAVEN父子项目，由五个项目组成:

- NS_MQ

  负责和底层消息队列进行通信，提供了对消息队列进行操作的API

- NS_TRANSPORTER

  通过调用NS_MQ提供的API,对业务消息进行收取，处理，转发

- NS_CHAIN

  一个可选开发框架，负责对同一个jvm中的业务处理步骤进行链条式的整合，组成当前业务模块的业务处理流程。

- NS_CONTROLLER

  一个业务消息转发应用，负责将接收到的消息转给对应的业务模块进行处理，同时负责将业务模块根据整体业务进行关联。NS_CONTROLLER本质是一个独立的应用系统，构建于NS_TRANPORTOR和NS_CHAIN之上。

- NS_DISPATCHER

  NS4架构规定的消息入口，通过提供的HTTP服务接受业务系统边界外的http请求，并将请求转化成业务系统内部通信使用的消息协议格式。

2.基础入门

2.1 开发环境配置

开发语言:JAVA

JDK版本:JDK1.7

MAVEN版本:3.3以上

REDIS版本:3.0以上

以上是开发环境必备的组件和配置，其中java为开发语言，maven为项目编译打包部署必备， redis作为消息中间件使用。

2.2 运行

NS4运行至少需要启动三个jvm项目，才能完整运行。启动整个项目分为如下三步：

第一步：

> NS4消息入口是一个http接口，http服务是由NS_DISPATCHER项目提供的，所以我们第一件事情就是要把NS_DISPATCHER运行起来。要运行NS_DISPATCHER，请直接运行Bootstrap类的main方法即可。
>
> 默认的,NS_DISPATCHER会在本机(127.0.0.1)的8027端口上监听http请求,如果收到http请求，默认会将http请求转换成内部通信消息，并存储到本机(127.0.0.1)的redis中，默认访问的redis端口号是6379。

第二步：

> NS_CONTROLLER负责接收NS_DISPATCHER传入的消息，并根据配置进行消息分发。所以我们随后需要运行NS_CONTROLLER项目。(为了方便，以后叫NS_CONTROLLER项目为CONTROLLER)
>
> 在CONTROLLER项目中我们不能直接运行，需要配置一些东西。CONTROLLER要运行至少需要指定一个配置文件位置。这个配置文件需要通过java命令参数来指定。假设我现在指定java运行参数
>
> -Dconfigfile=nscontroller.xml
>
> 这个参数本质上是给CONTROLLER底层的NS_TRANSPORTER使用的，这个参数指明了NS_TRANSPORTER必须得配置文件位置，使得CONTROLLER能顺利利用NS_TRANSPORTER进行消息收发。关于NS_TRANSPORTER的问题，请参考这里。
>
> 默认情况下，CONTROLLER还会去classpath下去找关于NS_CHAIN需要的配置文件，默认路径是classpath下的nschainconfig目录，在这个目录下所有的xml文件会被认作是NS_CHAIN需要的配置文件集合。关于NS_CHAIN的问题，请参考这里。
>
> 当配置文件配置好后，可以通过调用com.creditease.ns.transporter.context.XmlAppTransporterContext的main方法来启动NS_CONTROLLER
>
> 关于如何配置NS_CONTROLLER请参考这里

第三步：

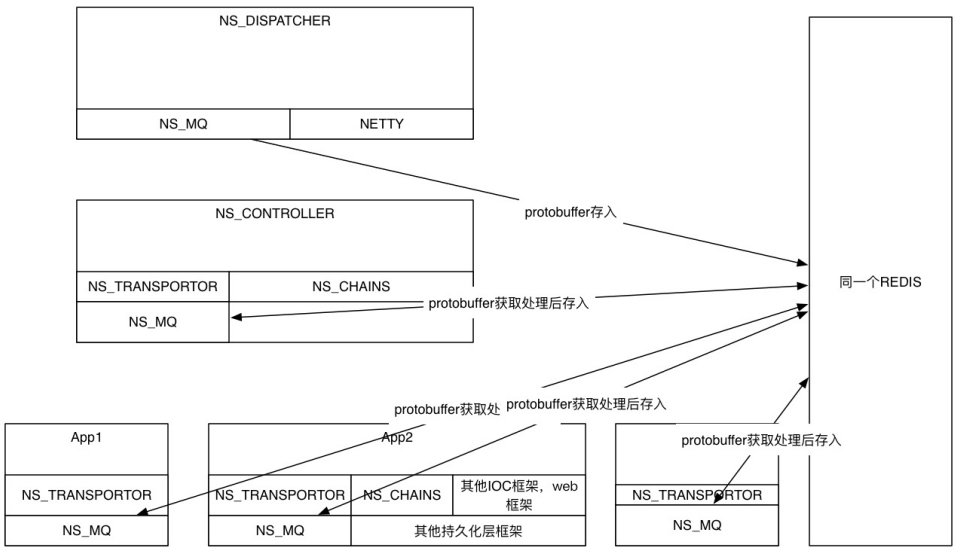
> 在这个步骤中我们需要启动自己的业务项目，在这个业务项目中，必须有以下三个前置条件：
>
> 1. 业务项目需要建立在NS_TRANSPORTER框架之上。
> 2. 业务项目的消息队列名称必须和CONTROLLER项目中配置的队列名一致。

> 3. 业务启动必须通过com.creditease.ns.transporter.context.XmlAppTransporterContext的main方法来启动

完成以上的三个步骤，一个基本的NS4系统就搭建好并运行起来了。

3. 项目架构

3.1 层次划分



以上的图揭示了NS4每个系统的层次结构

我们底层是以redis作为消息中间件

对消息中间件的操作被封装入了NS_MQ项目，它向上层提供了对消息队列的操作API接口。

在NS_MQ的上层是NS_TRANSPORTER,它本质是一套消息收发处理框架，它负责接收消息后反向回调业务代码，并将消息交给业务层处理。当业务层处理完毕后，它负责将处理后的消息返回给redis中。

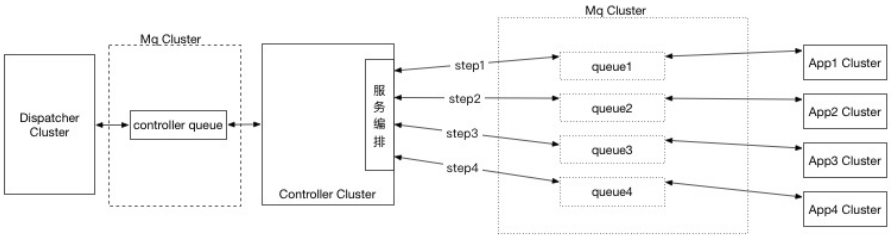
NS_CHAINS是一套开发辅助框架，它负责将一个模块的业务处理步骤解耦成一个个零散的任务，并可以随意以任何顺序做关联。

NS_CONTROLLER是一个项目，它本质上是一个独立的应用，它负责将整体业务分解成一个个节点，并通过配置将他们以一定的顺序关联起来，并通过消息机制，将这些节点结合起来形成一套业务系统。

NS_DISPATCHER也是一个项目，它是以NETTY框架作为基础，开发出的一个能提供基本的HTTP服务的独立应用。同时它也是业务系统和外部通讯的唯一边界。

3.2 运行流程

NS4整套系统本质上其实就是一套消息中间件服务加开发框架，整体的结构图如下：



以上的图显示了一个NS4整体分布式项目的运行流程，一个消息的运转流程按如下顺序：

1. NS_DISPATCHER收到http请求并将http请求转化为内部消息协议放入指定的消息队列中(根据配置文件)
2. NS_CONTORLLER从1步骤指定的队列接收到消息，并根据配置的服务编排开始按照顺序将消息发送到每个服务步骤对应的消息队列中。
3. 业务系统收到2步骤中NS_CONTROLLER指定的消息队列接收到消息，开始处理，处理完毕后，将结果返回。
4. NS_CONTROLLER收到业务系统的响应，开始根据配置好的服务，将返回的消息结果发送到下一个服务对应的消息队列中。

4. NS_MQ框架介绍

4.1 核心类和接口

RedisMQTemplate类:封装了所有和消息队列的操作相关的API

MQConfig:存储了所有和底层消息中间件相关的配置

4.2 配置方案

默认的，在没有做任何配置的情况下，NS_MQ会自动访问本机(127.0.0.1)的6379端口的redis，如果没有，则会报异常。

通常，NS_MQ会去找classpath下一个名为ns_mq.properties的配置文件，这个配置文件中会存储着所有和底层消息中间件相关的属性。下面会列举一些关键的配置元素：

redis.type 1 代表redis单机 2 代表redis集群 默认为1

redis.single/cluster.host redis单机或者集群的主机地址(包含端口)

redis.single/cluster.maxTotal redis单机或者集群的最大连接数

redis.single/cluster.miniIdle redis单机或者集群的最小闲置连接数

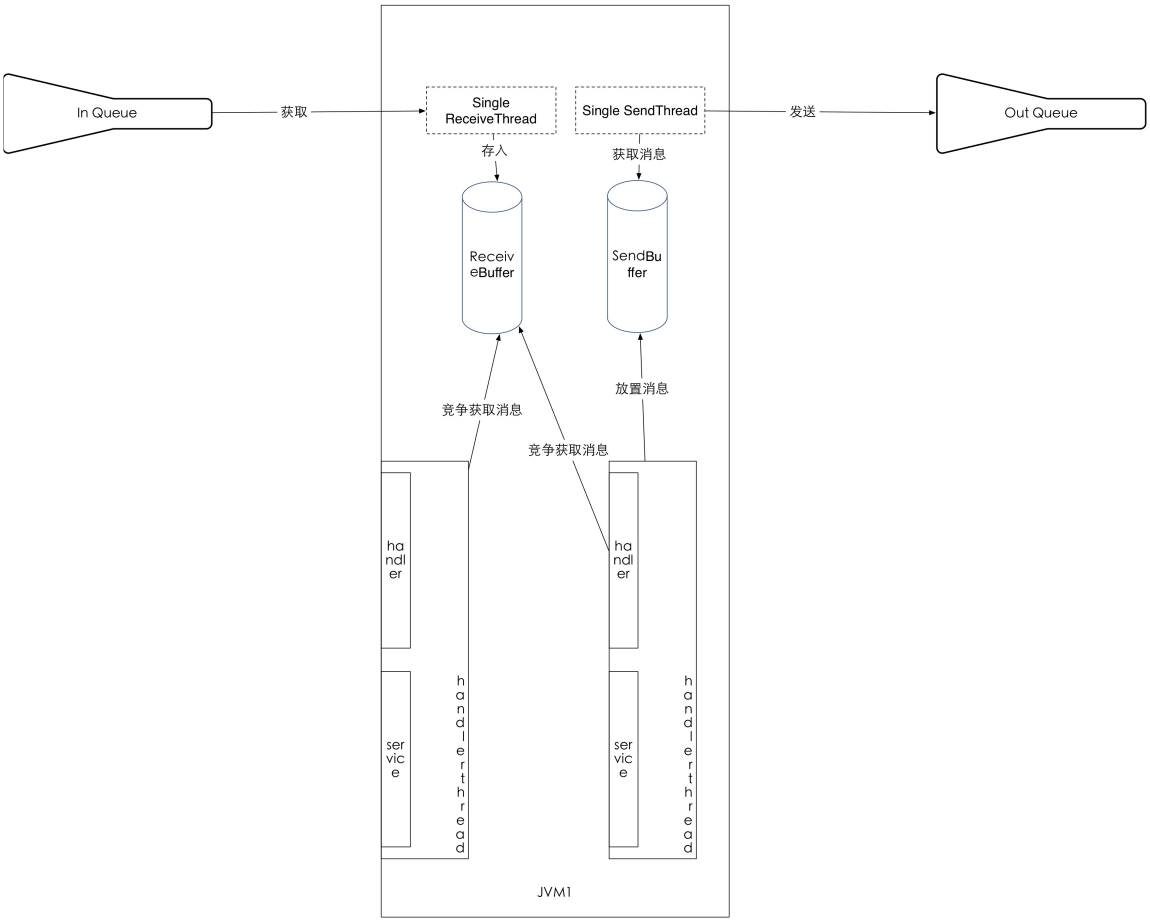
redis.single/cluster.maxIdle redis单机或者集群的最大闲置连接数

redis.single/cluster.connectionTimeout redis单机或者集群的尝试连接的超时时间(尚未连接到服务需要等待的时间)

redis.single/cluster.socketTimeout redis单机或者集群连接后socket闲置的超时时间

5. NS_TRANSPORTER框架介绍

5.1 框架架构



以上的图展示了整个NS_TRANSPORTER的整体架构，整套框架收发处理消息分为如下三个步骤：

1. 首先由接收消息的线程(Fetcher线程)通过NS_MQ从底层消息中间件获取消息并放入到本地消息缓存
2. 消息处理线程(Handler线程)从本地消息缓存中取出消息，并调用业务层的方法实现对消息进行处理，处理完毕后，将处理后的消息放到本地发送缓存。
3. 发送线程(Sender线程)从本地发送缓存取出消息后，将消息通过NS_MQ将消息放入底层消息中间件。

5.2 核心类和接口

ServiceMessage: 对各个模块之间传递的消息的java封装，包含了模块间通信需要知道的任何信息

Worker:业务层需要实现此接口的doWork方法,实现此接口的对象会被NS_TRANSPORTER的Handler线程回调用来对ServiceMessage中的信息进行处理。

ActionWorker:已经部分封装好的抽象类，实现了Worker接口，业务层可以直接继承这个抽象类，简化开发。

5.3 配置方案

默认的，NS_TRANSPORTER会去找名为configfile的系统变量，这个系统变量的值就是NS_TRANSPOTER需要的配置文件所在的路径，NS_TRANSPORTER会找到这个xml配置文件，并解析相关的配置后启动。

NS_TRANSPORTER相关的配置文件模板如下:

```
<queues>
     <prefix></prefix>
     <launchers>
          <launcher>
              <class name="类的全名" method="method方法名" property="" />
              <class name="类的全名" static-method="method方法名" />
          </launcher>
     </launchers>
       <inqueues>
          <queue>
              <name></name>
              <fetchernum></fetchernum>
              <buffersize></buffersize>
              <handlersize></handlersize>
              <serviceClass></serviceClass>
              <sendernum></sendernum>
          </queue>
     </inqueues>
     </queues>
```

以上xml模板中有如下几个关键元素需要注意:

launcher:用来定义在整个框架完全运行之前需要执行的方法。

queue:在这个元素下 name元素表示需要监听的底层消息中间件的队列名。fetchernum表示监听消息

队列并获取消息的线程数，默认是1。 buffersize表示本地接收/发送消息队列的大小默认是100。handlersize表示处理消息的线程数，默认是10。serviceclass表示具体的处理消息的业务类，这个类必须实现了Worker接口。sendernum表示从本地发送消息队列中获取消息后发送到底层消息中间件的线程数

6. NS_CHAIN框架介绍

6.1 框架架构

由于NS_CHAIN本质是一个纯开发框架，故暂时忽略此框架的框架架构。

6.2 核心类和接口

暂略

6.3 配置方案

这里详细说一下NS_CHAINS的配置。

NS_CHAINS启动时会去找系统变量chainconfig，这个变量的值就是NS_CHAINS配置文件所在的路径。NS_CHAINS支持配置目录(目录下的所有xml格式文件都被视作NS_CHAINS框架的配置文件)和配置文件。

对于NS_CHAINS的配置格式我们大致列举出关键要素如下:

catalog:这个相当于一个完整的服务或者一个命名空间，是NS_CHAINS对外服务的基本单位，NS_CHAINS外部系统只能看到catalog。

command:这是NS_CHAINS任务执行的最小单位，所有执行任务都可以以command的形式被调用执行。

chain:这是一个command的容器，可以将多个command的任务组合成一个执行链路。

group:这个一个command的组合，它可以将多个command组合成一个整体，并按照配置顺序执行。

同时NS_CHAINS的配置具有完整的逻辑语法，支持if条件判断，while循环结构和顺序结构。

7. NS_DISPATCHER应用介绍

7.1 框架架构

NS_DISPATCHER本质是一个独立的建立在Netty框架上的一个能提供http服务的独立应用，所以框架结构在这里从略。

7.2 核心类和接口

NS_DISPATCHER是以NETTY框架为基础的，所以其核心类就是如下的几个协议处理器:

HttpDispatcherServerHandler:主要负责解析传入的http请求，并封装成对应的java对象交给HttpRPCHandler做进一步处理。

HttpRPCHandler:主要接收上一步封装好的java对象，并取出对应的请求参数，请求内容等，封装成系统内部传输用的协议对象，并可以以同步请求响应模式/异步发送模式将协议对象放入底层消息中间件。

7.3 配置方案

NS_DISPATCHER启动会去找ns_dispatcher.properties文件，下面介绍配置的关键元素:

http.port:指定了http服务的监听端口

dispatcher.pool.num:指定了dispatcher的并发线程数，dispatcher的性能和这个参数有非常大的关系

dispatcher.queuename:封装好的内部协议消息要放入的队列的名字

NS_DISPATCHER也支持https，所以，如果在ns_dispatcher.properties文件中有如下几个选项，那么NS_DISPATCHER也会启动对应的https服务

ca.path:指明了可信任证书的路径

key.path:指明了公钥的路径

https.port:指明了https服务监听的端口

8. NS_CONTROLLER应用介绍

8.1 框架架构

NS_CONTROLLER本质是建立在NS_TRANSPORTER和NS_CHAINS上的独立应用，核心就是NS_TRANSPORTER的架构加NS_CHAINS的辅助，故不再重复列举其架构。

8.2 核心类和接口

DefaultPublishCommand:这是NS_CONTROLLER对于NS_CHAINS的一个扩展，它支持同步发送消息，并等待消息的响应，并可以设置等待响应的超时时间。同时，还支持异步发送消息，不需要等待消息的响应。

8.3 配置方案

遵循NS_TRANSPORTER和NS_CHAINS的配置规则，所以不再赘述。

注意:在NS_CONTROLLER中对于NS_CHAINS的配置做了一些功能扩展，主要是添加了publish的配置元素，这个随后可以提供配置模板。

9. 项目部署

9.1 部署方案

如果要部署整个NS4项目，请按照以下步骤进行：

- 部署NS_DISPATCHER项目：

  NS_DISPATCHER项目是一个Maven项目，首先需要 通过mvn:package deploy将整个项目打成一个zip包，并上传到服务器，然后解压成一个目录。在这个目录中，有如下几个子目录:bin,config,lib,logs。其中，bin目录中包含了DISPATCHER的启动脚本。config目录存放了NS_DISPATCHER必须的配置文件。lib目录存放了NS_DISPATCHER所需要的所有jar包。logs目录存放了所有NS_DISPATCHER打印的日志。

- 部署NS_CONTROLLER项目：

  NS_CONTROLLER项目也是一个Maven项目，需要 通过mvn:package deploy将整个项目打成一个zip包。目录结构同NS_DISPATCHER项目，此处不再赘述。

- 部署业务代码：

  业务代码请自行按照各个团队的规则部署。

10. 运行日志

10.1 日志分类

NS4项目将日志大致分成了四类：

*fram.log:系统日志，属于整个NS4底层系统内部的日志，包括系统的启动，线程的启动关闭等信息。

*biz.log:业务日志，所有业务相关的日志统统会被导向到这里。

*flow.log:消息流日志，这里记录了系统所有消息的流转信息。

*mq.log:这里记录所有对底层消息中间件进行操作的信息。

10.2 如何查看日志

业务报错：

如果业务报错，基本所有的报错信息都会在*biz.log中查到。

消息流转：

如果是消息发送响应的问题，基本上在*flow.log中可以查到或者推断出相关的信息。

底层消息中间件交互：

如果消息流转无法推断出问题，或者无法查到对应的消息，就需要转到*mq.log中进行查询。

11. 其他

11.1 常见问题

NS4系统本质是一个以消息为通信机制的分布式系统，经常出现的问题分成以下两部分：

1. 业务异常

   由于业务本身是由底层NS_TRANSPORTER回调来执行的，当业务出现异常的时候，很可能由于没有合适的被catch到，从而被底层的NS_TRANSPOTER框架捕获。

   对于没有在*biz.log和stdoout.log中查找到的问题，可以去查看下*flow.log的日志，看是否出现了异常被底层NS_TRANSPOTER捕获了。

2. 底层异常

   有些情况，业务本身并没有出现问题，但是由于消息通信出现了问题，会导致业务没有执行，对于这种情况我们需要首先从消息入口处即NS_DISPATCHER的*flow.log中查找到对应的messageId，然后根据消息流转路径，一步步去对应的部署机器上查询。