1. 通用通讯领域的建设

原生的NF所使用的通讯模块是基于libevent来完成的封装，而JAVA没有libevent版本的基础库，但java拥有一个比较强大的netty，本篇内容我将以一个简单的通讯领域的建设作为例子，全面的介绍一个插件和相应模块的建设。其中出现的领域划分知识请期待后期分享的内容

1. 公共接口建设
   1. NFINet

接口定义了网络协议的基本操作内容，如消息头的解析，报文的编码和解码等操作

* 1. NFINetModule

接口定义了系统初始、使用网络模块的基本操作内容，这个是场景划分的上下文定义。

* 1. NFIProtoco

接口定义了网络通讯中报文的协议内容，本例子协议内容如下所示，大家可以根据NF C++版本自行改动：



* 1. NFISocketModule

链接的模块接口是继承于网络接口NFINetModule，主要目的是对不同的链接提供不同的操作方法扩展。

1. 插件的配置项
   1. 插件的描述

插件的建设需要在pom内定义插件的基本描述：

*<!-- 定义插件的属性 -->*<**properties**>  
 <**plugin.id**>nf\_net\_plugin</**plugin.id**>//系统中唯一识别插件的模块  
 <**plugin.class**>com.noahframe.plugins.net.NFNettyPlugin</**plugin.class**> //插件的初始入口类  
 <**plugin.version**>1.0.0</**plugin.version**>//插件版本。目前只支持3位版本描述  
 <**plugin.provider**>zoocee</**plugin.provider**>//作者  
 <**plugin.dependencies** />//详细说明  
</**properties**>

* 1. 插件的依赖

插件环境可以动态的加载插件内的jar包到运行环境，这样的操作和配置可以减少系统运行环境对jar依赖的复杂度，若有多个插件加载同一个jar包（不同版本的情况），可以在运行环境进行配置，然后在插件以provided的方式进行加载。以netty库为例如下进行配置：

<**dependencies**>  
 <**dependency**>  
 <**groupId**>io.netty</**groupId**>  
 <**artifactId**>netty-transport</**artifactId**>  
 <**version**>4.1.16.Final</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>io.netty</**groupId**>  
 <**artifactId**>netty</**artifactId**>  
 <**version**>3.10.5.Final</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>com.noahframe</**groupId**>  
 <**artifactId**>NFMessageDefine</**artifactId**>  
 <**version**>1.0.0</**version**>  
 </**dependency**>  
</**dependencies**>

* 1. 插件的编译环境设置

插件的编译环境是固定的，具体相应的内容可以查找maven的资料学习其中内容：

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>2.5.1</**version**>  
 <**configuration**>  
 <**annotationProcessors**>  
 <**annotationProcessor**>com.noahframe.nfcore.api.plugin.processor.ExtensionAnnotationProcessor</**annotationProcessor**>  
 </**annotationProcessors**>  
 </**configuration**>  
 </**plugin**>  
  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-antrun-plugin</**artifactId**>  
 <**version**>1.6</**version**>  
 <**executions**>  
 <**execution**>  
 <**id**>unzip jar file</**id**>  
 <**phase**>package</**phase**>  
 <**configuration**>  
 <**target**>  
 <**unzip src="target/${project.artifactId}-${project.version}.${project.packaging}"  
 dest="target/plugin-classes"** />  
 </**target**>  
 </**configuration**>  
 <**goals**>  
 <**goal**>run</**goal**>  
 </**goals**>  
 </**execution**>  
 </**executions**>  
 </**plugin**>  
  
 <**plugin**>  
 <**artifactId**>maven-assembly-plugin</**artifactId**>  
 <**version**>2.3</**version**>  
 <**configuration**>  
 <**descriptors**>  
 <**descriptor**>  
 ../assembly/assembly.xml  
 </**descriptor**>  
 </**descriptors**>  
 <**appendAssemblyId**>false</**appendAssemblyId**>  
 </**configuration**>  
 <**executions**>  
 <**execution**>  
 <**id**>make-assembly</**id**>  
 <**phase**>package</**phase**>  
 <**goals**>  
 <**goal**>attached</**goal**>  
 </**goals**>  
 </**execution**>  
 </**executions**>  
 </**plugin**>  
  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-jar-plugin</**artifactId**>  
 <**version**>2.4</**version**>  
 <**configuration**>  
 <**archive**>  
 <**manifestEntries**>  
 <**Plugin-Id**>${plugin.id}</**Plugin-Id**>  
 <**Plugin-Class**>${plugin.class}</**Plugin-Class**>  
 <**Plugin-Version**>${plugin.version}</**Plugin-Version**>  
 <**Plugin-Provider**>${plugin.provider}</**Plugin-Provider**>  
 <**Plugin-Dependencies**>${plugin.dependencies}</**Plugin-Dependencies**>  
 </**manifestEntries**>  
 </**archive**>  
 </**configuration**>  
 </**plugin**>  
  
 <**plugin**>  
 <**artifactId**>maven-deploy-plugin</**artifactId**>  
 <**configuration**>  
 <**skip**>true</**skip**>  
 </**configuration**>  
 </**plugin**>  
  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-surefire-plugin</**artifactId**>  
 <**configuration**>  
 <**skip**>true</**skip**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

1. 创建插件

插件的创建很简单主要是继承了NFIPlugin，可以复制以下内容更改相应的类名进行创建

**public class** NFNettyPlugin **extends** NFIPlugin {  
 **public** NFNettyPlugin(PluginWrapper wrapper) {  
 **super**(wrapper);  
 }  
 @Override  
 **public void** Install() {

//因为在一定场景下模块的加载需要模块进行一定顺序的加载，在初始的时候如果在插件内需要定义加载模块的顺序可以在此进行初始化。详细内容可以参考NFConfigPlugin插件内Install的写法  
 System.***out***.println(**"NFNettyPlugin.Install()"**);  
 }  
 @Override  
 **public void** Uninstall() {  
 System.***out***.println(**"NFNettyPlugin.Uninstall()"**);  
 }  
}

1. 创建模块

模块的基础类是NFIModule，所有插件内模块必须都是NFIModule的子类，这样在编译的时候才能识别到这个插件，还需要注意的是，每个模块在系统内都是单例存在，然后增加@Extension注解，然后对模块引用的接口进行实现就可以了，如下所示：

@Extension  
**public class** NFNetModule **extends** NFISocketModule {

**private static** NFNetModule *SingletonPtr*=**null**;  
  
**public static** NFNetModule GetSingletonPtr()  
{  
 **if** (**null**==*SingletonPtr*) {  
 *SingletonPtr*=**new** NFNetModule();  
 **return** *SingletonPtr*;  
 }  
 **else** {  
 **return** *SingletonPtr*;  
 }  
}  
  
  
**public** NFNetModule()  
{  
 **pPluginManager** = NFPluginManager.*GetSingletonPtr*();  
  
 **mnBufferSize** = 0;  
 **nLastTime** = GetPluginManager().GetNowTime();  
 **m\_pNet** = **null**;  
 **mxReceiveCallBack**=**new** HashMap<Integer, NET\_RECEIVE\_FUNCTOR>();  
 **mxEventCallBackList**=**new** ArrayList<NET\_EVENT\_FUNCTOR>();  
 **mxCallBackList**=**new** ArrayList<NET\_RECEIVE\_FUNCTOR>();  
}

**……**

}

建设完模块后就可以将模块的ID加入到运行环境项目中的Plugin.xml内，让系统启动的时候可以加载：

<**XML**>  
 <**NFServer**>  
 <**Plugin Name="nf\_log\_plugin"** />  
 <**Plugin Name="nf\_kernel\_plugin"** />  
 <**Plugin Name="nf\_config\_plugin"** />  
 <**Plugin Name="nf\_net\_plugin"** />  
 <**ConfigPath Name="../"** />  
 </**NFServer**>  
</**XML**>

加载好了之后就可以在其他模块中使用Net模块的功能了

NFISocketModule m\_SocketModule= **pPluginManager**.FindModule(NFISocketModule.**class**);