Noodle Engine

V0.1.0

Dennis Wang

(dennis.wangkk@gmail.com)

索引

系统简介

Noodle 的设计目标是实现一个通用的分布式服务端框架，将网络接入、业务逻辑、数据存储等部分分离，提供一个基于消息传递的、异步的、可动态加载的、分布式的，可进行热部署的框架，最终为 7×24 小时不间断运行的系统提供框架级支持。

- **引擎核心**
  - 引擎
    - 从消息队列中取消息
    - 将消息按序分配给工作线程处理
  - 消息队列
    - 从消息源泉或其他组件产生的消息将在这里按到来的先后顺序排队
  - 线程池
    - 预先产生多个工作线程
  - 内存池
    - 从操作系统内存中预分配一部分内存
    - 提供给Noodle及组件使用
  - 组件管理器
    - 管理所有组件的生命周期
    - 管理所有组件实例的生命周期
  - 代理管理器
    - 管理所有实例的代理对象
  - 消息源管理器
    - 管理所有的消息源
    - 管理所有的消息源线程
  - 全局对象管理器
    - 管理所有在Noodle内的全局对象

- 日志
  - 提供单个Noodle实例的日志输出功能
- 配置管理器
  - 配置文件的加载
  - 配置文件的读取
- 提供给组件的运行环境调用接口
  - Noodle与组件互通信的接口
  - 组件调用Noodle系统功能的接口

- **终端**
  - 控制终端
    - 提供远程控制、查询单个 Noodle 实例的功能

- **组件**
  - 从功能上，组件可划分为核心组件和用户组件，核心组件与用户组件在实现上没有特殊的区别，核心组件是系统提供的一些具有通用功能的组件。
  - 用户的业务逻辑可在组件中实现，通过控制终端，任何组件都可在不停机的情况下被替换/更新/禁用/开启，替换/更新/禁用/开启的过程对最终用户是透明的。
  - 在系统的运行过程中，也可以动态加载/卸载新的组件，即，动态扩展/裁剪系统的功能。

- **分布式能力**
  - 互相通信的组件使用自定义消息体格式，通信的双方可以在同一台计算机上，或可互联的不同的计算机上。
  - 运行在不同计算机上的组件可以通过 Stub 组件和 Proxy 组件实现消息的互通，只需要通过配置文件，即可实现赋予无网络功能的组件网络消息传递的能力，传递的过程对组件是透明的，使单机系统平滑的过度为分布式系统。

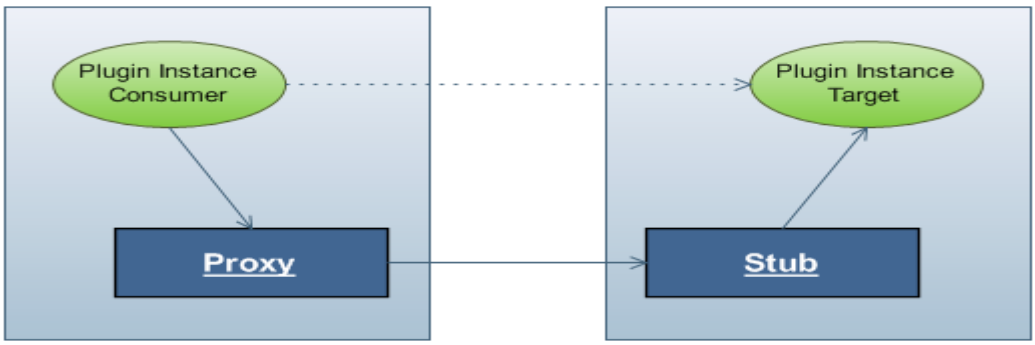

Illustration 1: Proxy & Stub

如上图所示，某个做为消费者组件实例，想发送一条消息给目标组件实例，由于目标组件实例与消费者组件不在同一物理机器上，通过系统配置，将一个 Proxy 组件置于消费者同一台物理计算机，将 Stub 组件置于目标同一台物理计算机。消费者的消息被 Proxy 路由到 Stub，再由 Stub 发送给目标组件，整个过程对于消费者和目标是完全透明的。理论上，可以通过配置形成所有组件（不论是否在同一台计算机上）的全连接网，即，任何组件都可以和任何其他组件用消息产生联系。对于小型的集群（几台到几十台），可以做到通过手工配置配置文件的方法来布置整个系统，但是对于大型的集群（几百到几千台），手工去配置每台机器是不现实的，需要配置文件自动生成及分发系统，这将在今后的版本中引入。

系统架构

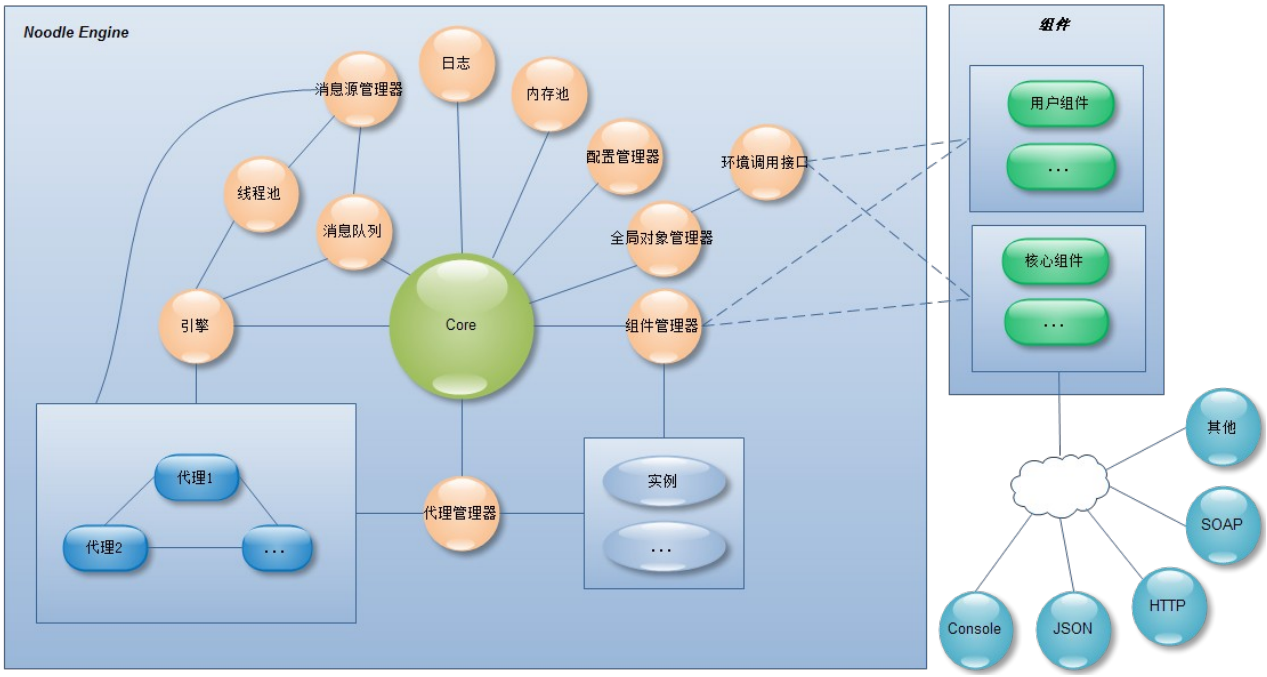

插图 2：系统架构

Core 作为整个框架的起始点，在启动时，Core 会启动所有的全局对象，并会将这些全局对象注册到全局对象管理器，全局对象管理器在系统中有唯一的实例，任何全局对象都可以通过全局对象管理器访问其他全局对象。框架通过环境调用接口为组件提供与框架交互的能力，比如：内存池内存分配、消息传递、配置文件读取等功能，每个组件在加载后会产生一到多个实例，每个实例会对每个对它的引用产生一到多个代理，代理是访问组件实例的唯一方法，也是系统热部署功能的基础。组件必须实现一个预定义的接口，这个接口会在组件的不同生命周期被框架调用，比如：组件加载、卸载等。组件从功能上分为两类：消息源组件和用户逻辑组件。消息源组件是外部消息接入的唯一方法，负责将外部消息规格化和格式转换。用户逻辑组件负责处理由消息源产生的消息。为方便开发人员使用，框架预定义提供了一些核心组件，这些组件也可以被开发人员所开发的同样功能组件所替换。有两个特殊功能的组件（Proxy、Stub），为实现框架的分布式功能提供了功能性的支撑。

编译与安装

Noodle 依赖于 boost 库，请先下载并安装 boost1.38（包含）以上版本。按以下顺序执行命令：

编译

```
$ tar zxf noodle-engine-0.1.0.tar.gz
```

$ cd noodle-engine-0.1.0

$ autoreconf

$ ./configure --prefix=YourInstallDIR

$ make

安装

$ make install

目录结构

Illustration 3: 安装后的目录结构

如图所示 bin 目录存放可执行程序，conf 目录存放配置文件，logs 存放 noodle 实例的日志文件，pid 目录存放 noodle 实例的 PID 文件，plugins 目录存放组件，lib 目录存放由自动化编译系统产生的动态库和静态库文件。

启动与终止

- 将当前目录转移至安装目录下的 bin 目录，用下面命令启动系统

  $ ./ne -c ../conf/noodle.conf

  此时，系统默认在安装目录下的 pid 目录下，会产生主进程的 PID 文件。

- 关闭系统，假定PID文件名为noodle.pid，此文件可在配置文件中配置

  $ ./ne -c ./conf/noodle.conf -p ../pid/noodle.pid -s stop

- 重启系统

  $ ./ne -c ./conf/noodle.conf -p ../pid/noodle.pid -s restart

- 重新加载配置文件

  $ ./ne -c ./conf/noodle.conf -p ../pid/noodle.pid -s reload

- 以后台进程方式启动

  $ ./ne -c ../conf/noodle.conf -d

- 指定日志文件路径和日志级别

  $ ./ne -c ../conf/noodle.conf -l ../logs/noodle.log -e 5

  日志文件路径和级别也可在配置文件中配置。

- 启动多个系统实例

  为每个实例提供不同的配置文件和日志文件，如果用同一个配置文件启动多个实例，多个console组件会监听同一个服务端口，导致启动失败。

  $ ./ne -c ../conf/noodle0.conf -l ../logs/noodle0.log -p ../pid/noodle0.pid

  $ ./ne -c ../conf/noodle1.conf -l ../logs/noodle1.log -p ../pid/noodle1.pid

  以上两行命令，启动了两个不同的系统实例。

- 关闭所有系统实例

  $ killall -15 ne

- 启动Console

  在安装目录的bin目录下。

  $ ./console 127.0.0.1 12345

  连接成功后会看到欢迎信息，输入help可看到详细命令行语法。

接口

运行环境接口

运行环境接口是组件与 Noodle 交互的纽带，运行环境接口提供的方法分类如下（详见附录）：

- 消息管理
  - 消息的产生
  - 消息的销毁
  - 设置和获取消息属性
  - 随机地获取某个组件的实例ID
  - 向其他组件发送消息
- 内存池的内存分配与释放

    - 从内存池分配指定大小内存
    - 释放内存到内存池
- 配置文件
    - 读取整数
    - 读取双精度浮点数
    - 读取字符串
    - 读取布尔值
    - 读取数组
    - 读取字典
- 日志
    - 写日志

组件接口

组件接口为用户编写 Noodle 组件时所实现的接口，接口方法按功能分为如下类别（详见附录）：

- 组件的加载/卸载

  用户通过接口方法的参数传递，得到 Noodle 环境接口指针。
- 组件实例的建立/销毁
    - 每建立一个实例，用户实现的建立实例的方法会被调用一次。
    - 每销毁一个实例，用户实现的销毁实例方法会被调用一次。
- 消息的产生/处理
    - 对于消息源组件，通过用户实现的产生消息方法返回消息对象指针，消息对象利用 Noodle 环境接口的方法产生并销毁。
    - 每条需要组件处理的消息，会调用用户实现的消息处理方法处理。

常见问题

待补充。

附录

配置文件示例

```
ne {
   system {
      pid = '../pid/noodle.pid'   # PID(Process Identifier)文件路径
      daemon = true               # 是否以守护进程方式启动
      pool {
         max = 1024000000  # 内存池所能占用操作系统内存的最大值
         begin = 4         # 内存池所能分配内存大小的最小值
         end = 1024000     # 内存池所能分配内存大小的最大值
         gap = 8           # 内存池内，相临内存块大小之差
         factor = 0.2      # 内存池内，内存块预分配系数
         singleChunk = true   # 是否禁用预分配
      }
      engine {
         worker = 2   # 引擎工作线程数量，一般与 CPU 核心数量相同
      }
      log {
         name = '../logs/noodle.log'  # 日志文件路径
         /*
         ERG    = 1,   ERGENT
         ALERT = 2,   ALERT
         ERR    = 3,   ERROR
         WARN = 4,   WARNING
         NOTI = 5,      NOTICE
         INFO = 6,      INFORMATION
         DEBUG = 7,   DEBUG
         */
         level = 5      # 日志级别
         flush = true  # 是否为每条写入刷新输出缓冲区
```

```
        }
    }
    plugin {
        path = "../plugins"  # 组件目录
        plugins = (          # 组件配置
            #<组件名称> => [ <组件 library 名称>, <是否是消息源>, <所生成的实例> ]
            "plugin3" => [ "libfcgitest.so", false, [ 3, 10, 11, 12, 13 ] ],
            "console" => [ "libpluginconsole.so", true, [ 50 ] ],
            "fastcgi" => [ "libfastcgi.so", true, [ 100 ] ]
        )
    }
}
console {
    ip = "127.0.0.1"  # 所监听的 IP 地址
    port = 12345      # 端口号
}
fastcgi {
    ip = "127.0.0.1"              # 所监听的 IP 地址，  Web Server 的配置中与这个 IP 相同
    #domain = "whatever";         # 保留
    port = 30000                  # 端口号
    concurrent = 4                # 并发线程数
    backLog = 1000                # 等待队列长度
    receiver = "plugin3"          # 接收 FastCGI 属性的组件名称
    maxBufferLength = 10240 # 每个线程的接收/发送缓冲区大小
}
```

分布式系统配置示例

待补充。

运行环境接口

见源代码 noodle-engine-0.1.0/include/ne.hpp

组件接口

见源代码 noodle-engine-0.1.0/include/plugin.hpp