Waverider设计杂谈

# 背景介绍

Waverider说的玄乎一点是一个***基于Master-Slave结构的分布式命令异步执行框架***，目的是支持多个Slave随时连接到Master机器，发送命令到Master端异步执行，Master端将命令执行结果再封装成命令发送到相应的Slave机器（或无须返回结果，视具体的命令而定），Slave也在接收到Master命令后异步执行然后视具体命令是否返回结果命令。如此往返来实现数据的交换。

实现中采取Slave主动，Master被动，Slave主动链接到Master，然后首先主动发送命令到Master。

其实Waverider最终的本质就是实现了Master和Slave之间的数据交换。本质是简单的，但是实现的过程中，还是有很多需要去考虑的。当然希望将Waverider实现的简单，简洁。

# 设计理念

Master和Slave之间通过相互发送命令(Command)实现交互，命令是一个抽象的概念，Command可以用来分发任务，也可以用来传输数据，或其他， 这完全由业务来决定怎么处理,，框架只定义了一个实际的命令：心跳检测命令(Heartbeat Command)。上层可以通过定义自己的命令，并提供相应的命令处理器，来实现任何形式的业务。框架提供的核心功能其实只有，底层的网络通信，Master/Slave关系的维系，命令的分发功能。

# 概要设计

****

**图3-1 系统概要图**

## 3.1 Command，执行命令

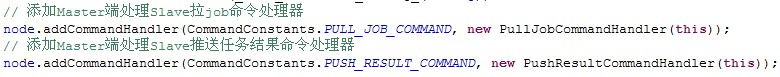
不管是Master还是Slave都有一个命令到命令消费接口的映射表，Master和Slave可以分别设定，上层系统可以定义自己一套命令然后提供命令相应的消费接口并注册到Master和Slave。系统会为Master和Slave默认注册一个心跳命令和心跳命令消费接口。

下面的示例是top-task-scheduler自定义的一套命令和消费接口：

**表3-1 命令到命令消费接口的映射表示例**

|  |  |  |
| --- | --- | --- |
| **命令** | **消费接口** | **描述** |
| **10** | class **PullJobCommandHandler** implements **CommandHandler** | **PULL\_JOB\_COMMAN**，Master要处理的Slave发出的拉取任务的命令 |
| **11** | class **PushResultCommandHandler** implements **CommandHandler** | **PUSH\_RESULT\_COMMAND**，Master要处理的Slave发出的推送任务结果的命令 |
| **12** | class **PushJobCommandHandler** implements **CommandHandler** | **PUSH\_JOB\_COMMAND**，Slave要处理的Master回送的推送任务的命令 |

在这个示例中，Master只需注册两个命令和命令的消费接口：



**图3-2 Master注册命令和命令消费接口示例代码**

Slave更简单只需注册一个命令和命令消费接口：



**图3-3 Slave注册命令和命令消费接口示例代码**

有了命令到命令消费接口的映射后，Master和Slave就知道，在接收到命令后，怎么去执行命令，其实只是简单的将命令分发给相应的处理器。

因为Slave是主动的，因此第一个命令（起始命令）是有Slave产生的，并发送到Master，这里就需要在Slave引入一个命令启动器，目前是支持在Slave中插入多个命令启动器的，目前Slave对命令产生器的支持并太合理，目前的做法是Slave会用一个线程定时依次调用每一个注册的命令产生器，如果命令产生器产生了命令，然后就将命令发送到Master，这里是有待改进的，目前还没想到到适合目前的设计的方法。

## 3.2 Slave结构和执行机制

既然Slave主动，那就先看看Slave的构造和Slave运行的机制



**图3-4 Slave内部构造**

Slave主要构造模块如下：

1. NetworkClient 网络通信客户端，负责和Master通信，并保持长链接。
2. CmdGenerator 命令产生器，负责周期的依次调用注册到Slave的命令启动器，产生命令，如果命令产生器产生了命令就通过NetworkClient发送到Master。
3. CmdDispatcher 命令分发执行器，负责不停的从NetworkClient接收命令，然后通过Slave的命令到命令消费接口的映射查找到消费接口并调用消费接口，如果命令执行有返回结果然后将命令执行的返回结果通过NetworkClient发送到Master。
4. HeartBeat 周期的通过NetworkClient发送心跳命令到Master，心跳命令中会承载一下Slave的系统信息。
5. MasterFailureMonitor 检测Master宕机的一个模块，他是依赖CmdDispatcher和HeartBeat模块的，目前没有实现，只是一个空架子，实现也是困难的，先搞个架子。

注意：

Slave发送心跳命令到Master后，Master是会返回一个包含整个集群的全局信息的心跳命令给Slave的，Slave这边对这个命令的处理是通过CmdDispatcher模块和其他命令一视同仁的处理的。只是Slave或默认注册这个心跳命令和消费接口。

Slave运行机制：

说白了Slave其实就只有几个在不同线程中运行的循环，有点类似事件循环。

**CmdGenerator命令产生器循环：**



**图3-5 Slave CmdGenerator命令产生器循环**

周期的依次调用注册到Slave的命令启动器，产生命令，如果命令产生器产生了命令就通过NetworkClient发送到Master。

CmdGenerator命令产生器循环采用单线程执行。

**CmdDispatcher循环：**

****

**图3-6 Slave CmdDispatcher循环**

CmdDispatcher利用但线程不停的尝试从NetworkClient取待执行的命令(如果NetworkClient没有可用的命令，CmdDispatcher会阻塞到直到有命令可用)，拿到命令后，利用Slave的命令与命令消费接口的映射查找到相应的命令消费接口，并调用该接口执行该命令，如果执行后返回结果(命令)，再通过NetworkClient将结果命令发送到Master，接着重复循环。

1. 取命令,从NetworkClient取待执行的命令。
2. 分发命令，利用Slave的命令与命令消费接口的映射查找到相应的命令消费接口
3. 命令消费接口执行命令
4. 如果命令执行有返回结果，调用NetworkClient发送结果到Master（异步）；
5. 回到第一步，继续循环。

**HeartBeat循环：**

HeartBeat循环比较简单，单线程定时的收集本Slave自身的信息，并构造一个Command通过NetworkClient发送到Master(异步)；

**MasterFailureMonitor：**

MasterFailureMonitor目前的想法是，在一定时间没收到Master的心跳后，就会采取相应的动作。

## 3.3 Master结构及运行机制

下面讲讲Master，Master要做的事情更多点, Master的构造及运行机制



**图3-7 Master内部构造**

Master主要构造模块如下：

1. NetworkServer网络通信服务端，监听在指定端口，接收Slave连接请求，负责与Slave通信。
2. SessionManager 管理Session的创建，回收，每当接受一个Slave连接请求，会为该Slave分配一个Session，这个Session会负责对应Slave整个生命周期的请求处理。
3. Session每一个存活的Slave，在Master端都会有一个对应的Session，这个Session会负责对应Slave整个生命周期的请求处理。

**NetworkServer 网络通信服务端**

NetworkServer 网络通信服务端，负责侦听网络，接受新Slave的连接，并将网络读写事件派发到相应Slave的Session的回调方法（onRead，onWrite，onException）。NetworkServer其实只有个核心的事件循环，不停的查询socket时间，并派发事件。



**图3-8 Master NetworkServer事件循环**

**SessionManager Session管理模块**

管理Session的创建，回收。

**Session**

每一个存活的Slave，在Master端都会有一个对应的Session，这个Session会负责对应Slave整个生命周期的请求处理。Session其实是一个Slave在Master端对应的处理线程，并且可以被回收重复利用。

一个Session其实就是一个事件处理循环，不停从自己的网络输入缓冲中解析Command，然后利用Master的命令与命令消费接口的映射查找到相应的命令消费接口并将执行结果写入到自己的网络输出缓冲中，并通知NetworkServer自己要写数据，如此重复。这样Session就一直在不停的处理对应Slave发来的Command。



**图3-9 Master Session事件处理循环**