1. 简介 2

2. 依赖 2

3. 设计与实现 2

3.1. executor的设计与实现 2

3.2. runtime\_context的设计与实现 3

3.3. channel\_pipeline的设计与实现 5

3.4. data\_type的设计与实现 5

3.5. channel\_pipeline\_context的设计与实现 6

3.6. server\_bootstrap的设计与实现 7

3.7. client\_bootstrap的设计与实现 8

使用 8

# 简介

lockless是基于无锁化设计思想设计的socket通信框架。整个框架实现尽量避免使用锁以规避多锁对程序并发运行造成的影响。其中的Context-Handler的设计源于java的netty框架。Lockless对其使用asio c++实现，并加入了自己的一些想法和实现。

包括服务端部分和客户端部分。服务端部分以server\_bootstrap为核心，提供对指定端口的监听。客户端部分以client\_bootstrap为核心，提供对指定socket的连接。

# 依赖

boost：<http://www.boost.org>

asio：<http://think-async.com>

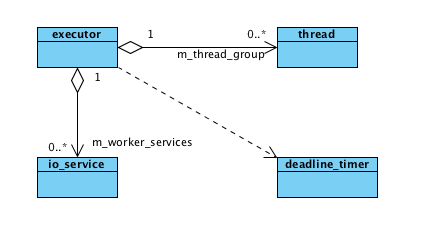
# 设计与实现

## executor的设计与实现

* + 1. 目标

实现运行asio::io\_service的守护线程池。

* + 1. 类设计



m\_thread\_group的size与m\_worker\_services的size是相等的。deadline\_timer用于保持io\_service一直运行。（保证run不退出）

## runtime\_context的设计与实现

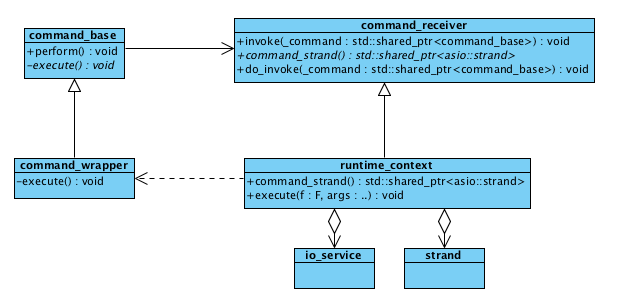
* + 1. 目标

串行化操作。既发生在该runtime\_context上的所有操作，都将在runtime\_context中m\_service所绑定的线程上顺序执行。这样做的目的是为了避免多线程环境中出现复杂的处理。

不同的runtime\_context，只要m\_service相同就会在相同的线程上串行化。

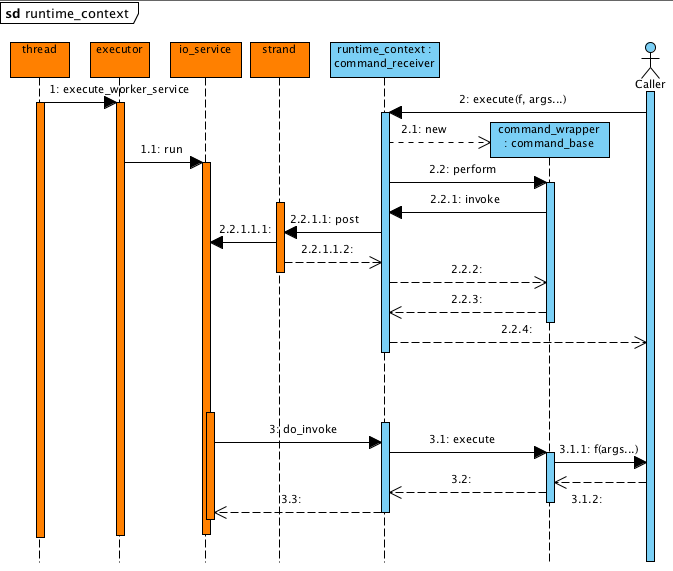
* + 1. 类设计

采取command设计模式。



runtime\_context充当了command模式中的receiver和invoker两个角色，command\_wrapper会根据f和args的不同构建不同的command。Invoker会调用receiver固定的函数do\_invoke，do\_invoke会执行command\_wrapper绑定的f(args…)。

* + 1. 时序设计

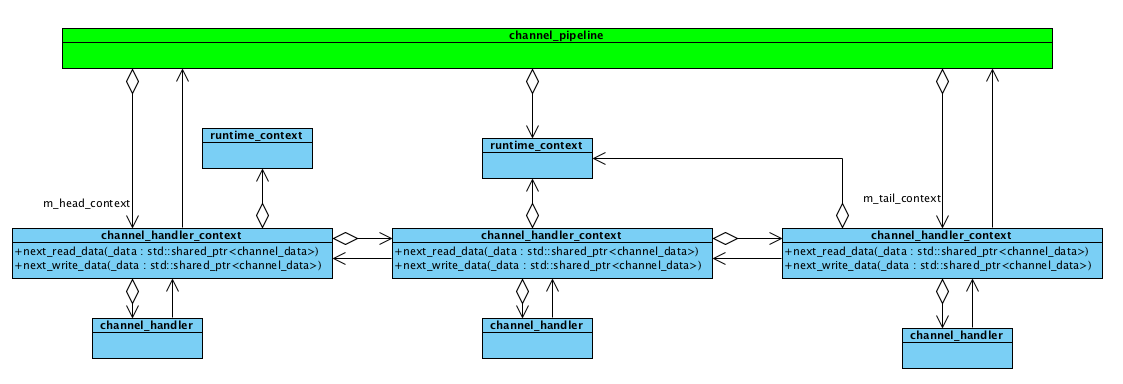


## channel\_pipeline的设计与实现

* + 1. 目标

分解读写数据处理的各个阶段，将不同职能的逻辑分散到不同的channel\_handler来处理。如： “加密/解密数据流”、“解析/包装协议”、“处理/生成协议”。

* + 1. 类设计



channel\_handler用来设定对读/写的某种类型（data\_type）的数据进行处理。如不能处理，则继续向下（next\_read\_data）/上（next\_write\_data）抛出该数据。

channel\_handler有standalone类型和普通类型之分，standalone的channel\_handler，具有独立的runtime\_context，在进行数据处理时，会放在下一个消息循环里处理。非standalone的channel\_handler会再处理数据时在一个for循环中，一起处理，他们共用runtime\_context，所以没必要放在不同的消息循环中处理。如果某一个channel\_handler里处理的内容是异步的，比较耗时，那么可以选择做成standalone模式，相当于可以选择把某些耗时的异步操作做成独立的channel\_handler，切不影响业务逻辑的顺序处理。

## data\_type的设计与实现

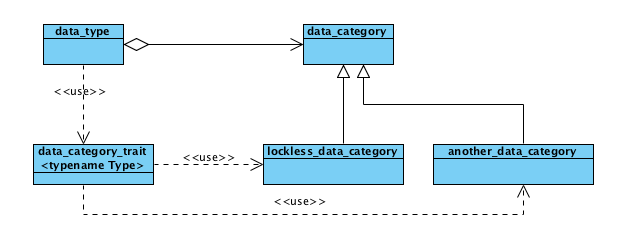
* + 1. 目标

为了实现数据类型的可扩展性要求。每个人在实现自己模块的时候，可以定义自己的category。

能够方便的将不同开发定义的data\_type转化成可读的message予以调试。

方便的使用enum class来对不同类型的数据予以区分，同时采用category的方式来区分不同域内的数据类型。说的简单点，就是不同category下，enum class转化成相同的int值仍然代表不同的数据类型。

* + 1. 类设计



该设计源于对std::error\_code的分析。

data\_category\_trait用来根据不同的enum class选择不同的data\_category。判断data\_type是否相等需要依据两个值，一个是enum class转化的int值，另外一个是通过enum class确定的category是否相同。

## channel\_pipeline\_context的设计与实现

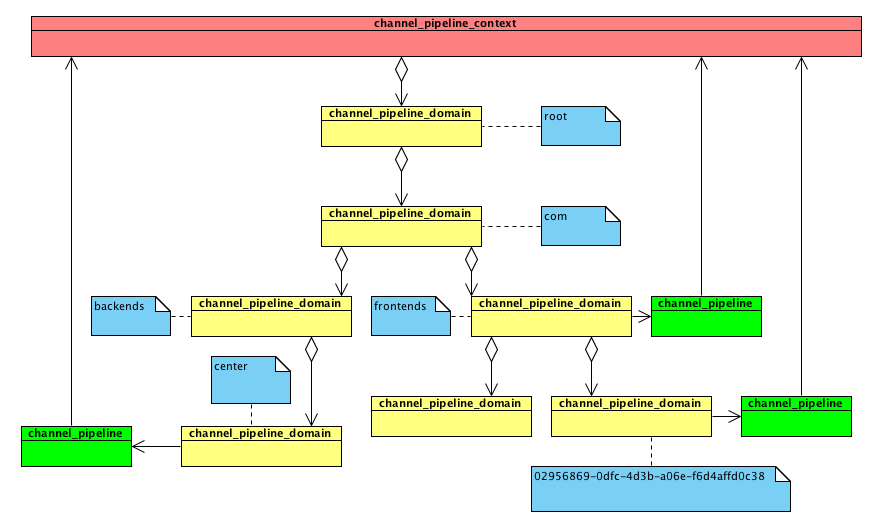
* + 1. 目标

用于处理多个channel\_pipeline共存，通讯的问题。Channel\_pipeline的组织采用了类似域名的组织结构。如root.com.backends.center。pipeline在加入到某一个channel\_pipeline\_context之后，会成为其中的一个节点。

需要实现向确定的某一个pipeline读写数据，某域内的某一个pipeline读写数据（区域内的某一个），向一批叶子节点读写数据（区域内广播）

对channel\_pipeline\_context的操作由单独的executor实现，避免出现多线程读写的问题。

* + 1. 类设计



root这个节点会一直存在

并不是所有的节点一定会有pipeline

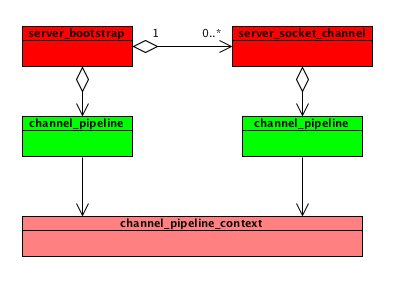
com.frontends. 02956869-0dfc-4d3b-a06e-f6d4affd0c38是其中一个pipeline。注意，com这个节点并不存在pipeline。

## server\_bootstrap的设计与实现

* + 1. 目标

开启一个服务器，监听在某一个端口上，当有客户端连接时，新建一个pipeline对其进行业务处理。

* + 1. 类设计



server\_bootstrap本身的channel\_pipeline用来处理server\_bootstrap本身的逻辑。

server\_socket\_channel的pipeline用来处理与客户端连接有关的逻辑。

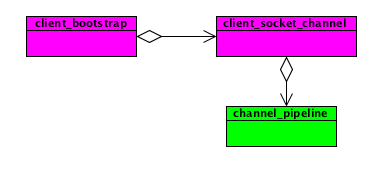
多个pipeline注册在不同的pipeline\_context上可以实现他们之间的通信。

## client\_bootstrap的设计与实现

* + 1. 目标

开启一个客户端，连接到指定的服务器socket，连接时建立一个pipeline进行业务处理。

* + 1. 类设计



# 使用