目录

[1. 高性能Server整体设计 2](#_Toc19481193)

[1.1 基本设计 2](#_Toc19481194)

[1.2 整体静态结构图 2](#_Toc19481195)

[1.3 整体动态流程图 2](#_Toc19481196)

[1.4 线程设计 3](#_Toc19481197)

[1.5 server端流程图 3](#_Toc19481198)

[1.6 支持协议 4](#_Toc19481199)

[1.7 超时处理 4](#_Toc19481200)

[1.8 Epoll管理 5](#_Toc19481201)

[2. 异步客户端设计 6](#_Toc19481202)

[2.1 结构设计 6](#_Toc19481203)

[2.3 概要设计 6](#_Toc19481204)

[2.5 异步调用根本 6](#_Toc19481205)

[3. 协程设计 7](#_Toc19481206)

[4. 二期问题 7](#_Toc19481207)

# 高性能Server整体设计

## 基本设计

由于服务器程序对性能要求非常高，单线程或者阻塞模式已经无法满足性能的要求，所以选择采用多线程，异步epoll，边缘触发的方式来实现。

不选择多进程的原因是因为广告投放系统通常有很多用户设置的数据，多经常需要每个进程都去加载数据，并且reload数据。内存开销是单进程的n被。

## 整体静态结构图

整体静态结构设计如下：

最底层是epoll thread，事件，Timer，有协议的解析和封装，Io service管理服务端的一些事件，由agent将来注册和操作事件，server connect管理服务端接收的连接，clientmgr中的client connect管理着作为客户端的每一个连接，epoll接收读写消息之后，转给connect对象去读写处理消息，消息解析处理之后，给到iohandle或者service handle。最终事件交由上层的handle interface去处理。

Handle Interfaces

service Handle thread

Io Handle thread

Client connect

Server connect

Io service

Timer

Epoll Threads(event)

Events Agent

Master and slave

Protocols

## 整体动态流程图

主线程启动socket服务之后，进行配置检查，一旦有配置修改，重新reload，并且主线程初始化master线程，slave线程。

Master线程负责接收客户端连接，并创建connection，并将新连接的fd读事件转到slave线程。

Slave线程负责处理读写事件，转到connection去处理，connection负责进行协议解析等等，协议解析之后扔给一个io handle/service handle的队列中，线程会循环消费消息，并转交给handle interface来处理事件。

ClientMgr负责管理客户端相关的连接对象，读写事件依然由Slave线程管理，处理流程和上面一致。

Handle中如果要加入超时机制，则设置超时时间，如果超时则调用注册对象的on\_timer()函数，ontimer由io handle thread触发。

Epoll线程中循环调用run\_timer,run\_event,进行超时处理和事件处理

Io handle线程和service handle线程也一样，调用runtimer/processtimer(必要时)以及processmessage。

Main thread

Init and reload config

Connection

Slave thread

io/handle thead

Master thread

Accept and init connection

Handle interface

Io/service Handle

## 线程设计

1：master线程：负责accept客户端

2：slave线程，负责监听服务的连接，和客户端的连接，有事件时，进行通知，回调

3：io handle线程：需要发送/写消息的时候，发送到io handle线程的队列中。在io handle调用连接对象进行协议封装之后，进行数据写入/发送。

3：service handle线程：在连接对象对协议进行解析成message之后，发送到service handle队列中，service handle循环发送给应用层处理

## server端流程图

http处理流程：



## 支持协议

目前server端支持两种基本协议：

1：标准http 协议（发送给多个dsp的http协议，第二期会支持）

2：自定义rapid协议（自定义的）

## 超时处理

因为server的设计是异步的，所以有可能在handle的时候便有超时的情况。

服务端：应用层自己控制，message中带有解析到消息的事件，如果有超时，则终止处理，并返回。

客户端：客户端的timer在io handle thread中，会不断的check是否有消息超时，如果有超时消息，则返回到之前发送的service handle去处理。

## Epoll管理

1：Epoll具备唤醒机制，用管道来实现。有新连接来需要唤醒。

2：master thread负责处理监听的fd事件，每个slave线程负责处理自己的epoll事件，master线程accept到的fd用来hash（或者编号，一个个取），落到哪个slave线程就用哪个线程来处理。客户端创建连接之后，也会将相应的读写事件注册到slave线程去监听。

# 异步客户端设计

异步客户端是基于server的基础上做开发的。

由于异步客户端本身的复杂性和多样性，以下设计为初步设计，后面可优化为协程处理。

## 结构设计

Slave thread event

Router 1

ClientMgr

Handle

interface

Io handle thread

Service handle thread

Client/connect

Router 2

Client/connect

Router n

shi

Slave thread event

## 概要设计

ClientMgr启动的时候将所有的客户端做一个connect连接。连接之后生成一个connect对象。并注册事件。如果没有连接成功的，会启动一个定时任务，1秒钟进行一次重连。同时也会启动一个定时任务，定期发送心跳包。

## 异步调用根本

需要在message中保存一个协议请求的request\_id，循环遍历timer的时候才知道是否已经超时，如果超时则删除对应的请求。（如rapid协议）

如果超时之后请求又回来了，因为之前保存的map里面发现没有了，则说明已经超时了，给个warning/info/trace。

对于普通的http请求，没有唯一id，无法进行异步调用，只能同步发送消息。

对于redies这种单线程server，如果以后实现了redis协议，则可以使用列表来保存，超时则删除列表的第一个元素。（或者超时就关闭连接）

# 协程设计

协程的本质就是开辟一段空间，在这段栈空间内，使你的程序能正常运行，需要保证你的变量不要发生栈溢出，切换出去之后保存栈空间，切回来之后恢复栈空间

当发生阻塞操作时，切出协程，切换到另外一个协程去运行，当阻塞的结果返回/或者超时的时候，切回协程处理。

第一期v1.0协程设计采用glibc原始的makecontext和swapcontext来实现协程。

并且最简单的实现sendreceive函数，先send，send之后进行receive，receive中设置好定时器等需要的参数，然后切换出协程，当有消息返回/或者定时器超时，切换回协程，继续处理，使业务端看起来完全时一个同步的接口，降低业务端编码和理解的难度。

# 二期问题

因为第一期支持协程的方式并不是特别通用的方式，比如和libco，spp，libgo等支持的方式都有一定差异，预计需要一个大的版本改版

1：去掉Service handle thread（到时第一期画的很多图就不完全正确了）

2：协程使用方式改进与协程该井

3：支持adx向多个dsp发送的方式

4：支持配置reload

5：利用协程实现部分阻塞函数支持hook