目录

[1. 高性能Server设计 2](#_Toc417057678)

[1.1 静态结构图 2](#_Toc417057679)

[1.2 整体动态流程图 2](#_Toc417057680)

[1.3 server端流程图 3](#_Toc417057681)

[1.4 支持协议 4](#_Toc417057682)

[1.5 超时处理 4](#_Toc417057683)

[1.6 Epoll管理 4](#_Toc417057684)

[2. 异步客户端设计 5](#_Toc417057685)

[2.1 结构设计 5](#_Toc417057686)

[2.2 支持协议 5](#_Toc417057687)

[2.3 概要设计 5](#_Toc417057688)

[2.4 Epoll管理 6](#_Toc417057689)

[2.5 异步调用根本 6](#_Toc417057690)

[2.6 超时处理 6](#_Toc417057691)

# 高性能Server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。

## 静态结构图

整体静态结构设计如下：

最底层是event，事件，还有协议的解析和封装，由agent将来注册和操作事件，并进行socket操作，conecting管理连接。最终事件交由上层的handle去处理。

Handle Interfaces

Connection

Socket

Events Agent

Master and slave

Epoll events

Protocols

Timer

## 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，协议解析之后最好是扔给一个handle去处理，这样就能有handle来处理事件。

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

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

Main thread

Init and reload config

Connection

Slave thread

Or work thead

Master thread

Accept and init connection

Handle

## 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：http 协议

2：custom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。

在slave线程处理请求的时候，检测是否有超时，如果有，则执行超时函数，并且invoke回来之后需要立即默认返回处理。

## 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。

# 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，以后可优化。

## 结构设计

Invoke thread

mgr

Select connecting

Invoke handle

Send/read

Encode/decode

Thread 1

Thread 2

Thread n

## 支持协议

1：需要支持adx到dsp这种特殊的http协议以及普通的http协议

2：支持自定义协议

## 概要设计

初始化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，再第一次invoke的时候继续连接。

将所有的connect放到一个invokemgr里面去管理，并指定connect的invokehandle（返回时用到），发送时：invoke时根据ip和端口去取连接。（如果接口有callback，则设置callback函数），如果取到了则，并且是有效的连接，则进行协议打包等，发送消息，

返回时，connect然后转到invokehandle去执行。如果有设置回调，则执行回调函数。

如果是adx向dsp发送的http协议，invoke回来之后，一个一个收集response，等到所有response都收到或者超时的时候，再去执行callback函数。（需要根据response中的request id来进行分组，所以这里要解一下pb，解析之后干脆把pb存这里，然后传到应用层，免得应用层再去解析）

这里考虑过invokemgr里面是否启动多线程，考虑到作为server的部分本来就是多线程的，也希望能更大的发挥异步客户端的效率，异步客户端选择了启用多线程。

所以这里设计和以前的框架有些不一样，（以前是invoke用biz线程，返回之后采用invoke线程接收，然后再转到biz）invoke出去的时候就去取了不同的线程。返回也是这些对应线程的handle去处理。知道处理结束都是invoke的线程来处理。

## Epoll管理

Epoll个数和线程个数一致，以及handle个数也一致，由invokemgr来管理。

## 异步调用根本

需要再invokehandle中保存一个http请求的request\_id（string类型,这个是对应adx的http invoke请求），如果是自定义协议，也需要保存协议的一个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custom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。

对于普通的http请求，没有唯一id，无法处理对应，除非invoke的服务端是单线程的。

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，超时则删除列表的最后一个元素。

## 超时处理

超时处理分两部分：

1：http server端的超时处理

由handle将超时设置到底层，然后由底层来触发。（一般只有invoke之后才会超时，是再invoke之后检测）

2：invoke超时，一般是invoke出去之后到receive回来的过程超时。会根据2.5提到的id来判断是否超时。

不管是否超时，都需要将消息扔到应用层处理，因为http server调用invoke的时候可能设置了回调函数。即使是invoke超时了也需要执行之前设置的回调。