

ZNet 使用手册

文档版本:1.1

更新时间:2023-10-11

更新日志:

2023-10-4 开启撰写

2023-10-6 完成 1.0 撰写

2023-10-9 追加双主线程和互斥锁技术资料

2023-10-9 追加性能工具箱资料

祝大家工作顺利,项目大发.

ZNet 相关网站

开源主站 <https://github.com/PassByYou888/ZNet>

开源备用站 <https://gitlab.com/passbyyou888/ZNet>

作者个人站 <https://zpascal.net>

ZNet 作者 qq600585

qq 群, 490269542(开源群),811381795(AI 群)

目录

| | |
|---|----|
| 名词和关键机制..... | 4 |
| ZNet 的 6 种命令收发模型..... | 4 |
| ZNet 双通道模型..... | 5 |
| ZNet 线程支持..... | 5 |
| HPC Compute 线程分载方案..... | 5 |
| 万兆以太网支持..... | 6 |
| 使用 ZNet 必须知道的主循环三件事..... | 7 |
| 第一件事:遍历并且处理每个 IO 的数据接收流程..... | 7 |
| 第二件事:遍历并且处理每个 IO 的数据发送流程..... | 7 |
| 第三件事:管理好每次遍历的 cpu 开销..... | 7 |
| 优化数据传输..... | 8 |
| 分析瓶颈..... | 8 |
| 当分析出性能瓶颈以后,接下来的工作就是解决瓶颈..... | 8 |
| 做服务器总是围绕硬件编程..... | 9 |
| 架桥..... | 10 |
| ZNet 桥支持..... | 11 |
| C4 启动脚本书写方式..... | 11 |
| win shell 命令行方式:..... | 11 |
| linux shell 命令行方式..... | 11 |
| 代码方式..... | 11 |
| C4 启动脚本速查..... | 12 |
| 函数:KeepAlive(连接 IP, 连接 Port, 注册客户端),..... | 12 |
| 函数:Auto(连接 IP, 连接 Port, 注册客户端)..... | 12 |
| 函数: Client (连接 IP, 连接 Port, 注册客户端)..... | 12 |
| 函数: Service (侦听 IP, 本机 IP, 侦听 Port, 注册服务器)..... | 12 |
| 函数: Wait(延迟的毫秒)..... | 13 |
| 函数:Quiet(Bool)..... | 13 |
| 函数:SafeCheckTime(毫秒)..... | 13 |
| 函数:PhysicsReconnectionDelayTime(浮点数,单位秒)..... | 13 |
| 函数: UpdateServiceInfoDelayTime (单位毫秒)..... | 13 |
| 函数: PhysicsServiceTimeout (单位毫秒)..... | 13 |
| 函数: PhysicsTunnelTimeout (单位毫秒)..... | 13 |
| 函数: KillIDCFaultTimeout (单位毫秒)..... | 13 |
| 函数: Root (字符串)..... | 14 |
| 函数: Password (字符串)..... | 14 |
| UI 函数: Title (字符串)..... | 14 |
| UI 函数: AppTitle (字符串)..... | 14 |
| UI 函数: DisableUI (字符串)..... | 14 |
| UI 函数: Timer (单位毫秒)..... | 14 |
| C4 Help 命令..... | 15 |
| 命令:help..... | 15 |
| 命令:exit..... | 15 |

| | |
|---|----|
| 命令:service(ip 地址, 端口)..... | 15 |
| 命令:tunnel(ip 地址, 端口)..... | 15 |
| 命令:reginfo()..... | 15 |
| 命令:KillNet(ip 地址, 端口)..... | 15 |
| 命令:Quiet(布尔)..... | 16 |
| 命令:Save_All_C4Service_Config()..... | 16 |
| 命令: Save_All_C4Client_Config()..... | 16 |
| 命令: HPC_Thread_Info()..... | 16 |
| 命令:ZNet_Instance_Info()..... | 16 |
| 命令: Service_CMD_Info()..... | 17 |
| 命令: Client_CMD_Info()..... | 17 |
| 命令: Service_Statistics_Info()..... | 17 |
| 命令: Client_Statistics_Info()..... | 17 |
| 命令: ZDB2_Info()..... | 17 |
| 命令: ZDB2_Flush()..... | 17 |
| ZNet 内核技术-锁复用..... | 18 |
| ZNet 内核技术-Soft Synchronize..... | 18 |
| 内核:Check_Soft_Thread_Synchronize..... | 18 |
| 内核:Check_System_Thread_Synchronize..... | 18 |
| ZNet 内核技术-双主线程..... | 19 |
| RTL 原主线程同步到次主线程..... | 19 |
| 次主线程同步到 RTL 原主线程..... | 19 |
| 双主线程开启以后的主循环..... | 19 |
| ZNet 性能工具箱使用指南..... | 20 |
| 性能瓶颈分析..... | 20 |
| 排除 ZNet 重叠 Progress..... | 21 |
| 敬畏服务器主循环 progress..... | 21 |
| 如何推翻使用 ZNet 的项目..... | 22 |
| 如何使用 ZNet 开发 web 类项目..... | 22 |
| ZNet 与 http 和 web..... | 22 |
| 文本最后来一个极简 C4 的 CS demo..... | 22 |

名词和关键机制

- **卡队列,卡服务器**:卡主循环,通讯不流畅,如果服务器带有 UI 系统,UI 也会表现出假死
- **阻塞队列,等待完成,等待队列**:等待是 ZNet 的特有机制,队列后面会等待前面完成,会严格按次序执行,等待会全部在本机等待,只有远程影响后,本机队列才会继续,不是把队列全部发送过去
- **序列化队列**:不会等待数据反馈,直接发数据,而数据的接收顺序是严格化的,按 1,2,3 序列发,那么接收也会是按 1,2,3 进行触发.例如使用序列化队列发送 100 条,再发送一条阻塞队列指令,待阻塞返回既表示 100 条序列已发送完成.同样,例如上传一个文件,耗时 1 小时,那么先发文件,再发条阻塞,待阻塞返回,既表示文件发送成功.
- **非序列化队列**:发送与接收均按严格序列机制处理,但是触发接收后 ZNet 会在某些子线程或协程中做解码这类程序处理,按 1,2,3 序列发送,接收数据以后会放到线程中处理,不会按 1,2,3 严格序列触发接收事件.非序列化常用于对数据前后无要求的通讯.

ZNet 的 6 种命令收发模型

- **SendConsole**:支持加密,支持压缩,阻塞队列模型,每次发送后都会等反馈,例如,先发 100 条命令,最后发一条 SendConsole,反馈回来时也表示 100 条命令都已经发送成功.SendConsole 很轻量,适合收发低于 64K 的小文本.例如 json,xml,ini,yaml.
- **SendStream**:支持加密,支持压缩,阻塞队列模型,每次发送后都会等反馈,所有收发数据都会以 TDFE 进行编码解码,由于 TDFE 具备数据容器能力,因此 SendStream 常被用于应用数据收发,SendStream 同样具备阻塞队列能力.在流量方面可以支持更大的数据.
- **SendDirectConsole**:支持加密,支持压缩,序列化队列模型,不会等反馈,用于收发基于字符串的序列化数据.
- **SendDirectStream**:支持加密,支持压缩,序列化队列模型,不会等反馈,使用容器打包数据,可以收发更大的序列化数据.
- **SendBigStream**:不支持加密和压缩,序列化队列模型,解决超大 Stream 收发,例如文件,超大内存块数据.SendBigStream 工作机制每次只发送一部分,一直等待信号出现才会继续发送,不会挤爆 socket 缓冲区.物理网络的带宽和延迟都会影响 SendBigStream 工作效率.
- **SendCompleteBuffer**:不支持加密和压缩,序列化队列模型,不会等反馈,高速收发核心机制,兆以太网支持的核心机制.CompleteBuffer 设计思路就是围绕网络来复制 Buffer,高速收发是一个非常重要的机制,取名叫做 CompleteBuffer.

ZNet 双通道模型

双通道是设计层面的概念,表示接收和发送各是一个独立通道连接,信号收发被区别设计,早期双通道是创建两个连接来工作.后来经历了无数摸索和升级,现在的双通道是建立在 p2pVM 基础上,p2pVM 可以在单连接基础上虚拟出无限多虚拟化连接,这些 p2pVM 连接在应用层都是双通道.

试想一下,过去我们堆出多台服务器,需要定义无数多的侦听,连接,端口,现在用 p2pVM 来虚拟化一切连接.因为降低了后台技术复杂性,这给后台系统提供了堆大的可维护性和规范性空间.例如 C4 的所有服务全部走 p2pVM 双通道.

ZNet 线程支持

ZNet 天生支持在线程中发送数据,甚至是并程序,发送的数据将会是非序列化队列.

ZNet 的数据接收环节总是位于 Progress 焦点中,也就是触发 Progress 那个线程,在多数情况下,ZNet 都建议在主线程中执行 Progress 主循环,例如 C4 框架就是用主线程跑 Progress.

ZNet 可以支持在线程中跑 Progress 从而达到线程收发,但并不建议这样干,因为 ZNet 有更好线程分载方案.

HPC Compute 线程分载方案

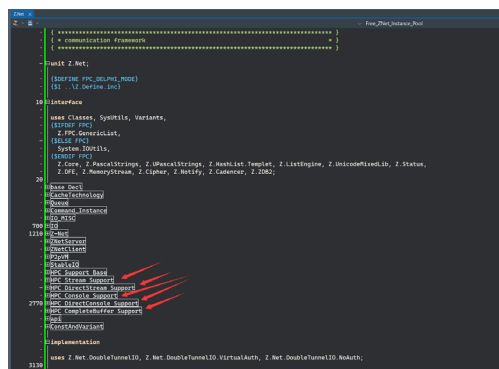
在 ZNet 中以 HPC 开头的折叠代码就是线程分载解决方案

流程:当数据从主线程到达,触发接收事件,启动分载,这时数据会转移,并且建立一个新线程来执行处理,这样干以后,服务器可以运行于无卡顿状态.因为保护了主线程逻辑,稳定性优于自己开 Critical 管理线程.

分载方案可以支持除 BigStream 之外的全部命令收发模型.

分载不可以重叠执行:命令->HPC->HPC->返回,只能:命令->HPC->返回

命令->HPC 被触发时是走的数据从主线程转移到线程,并没有发生数据 copy



万兆以太网支持

ZNet 使用 CompleteBuffer 机制来支持万兆以太网

- 在 ZNet 内部 SendCompleteBuffer 可以工作与线程中:线程机制可以为大流量数据提供预处理这类先决处理条件,例如 10 条线程做数据生成,然后 SendCompleteBuffer
- SendCompleteBuffer 具备高流量缓冲能力,当成规模的调用 SendCompleteBuffer 将被缓冲到临时文件,待主循环触发时成片的 CompleteBuffer 才会被发送出去:该机制对长队列数据提供了缓存机制,但不可以暴力 Send,一般情况下,10 次 SendCompleteBuffer 可以配上一次 SendNull(阻塞队列),这样干会让整体网络的负载和吞吐保持良好状态.
- SendCompleteBuffer 可以与 DirectStream 重叠
例如服务器注册的命令模型是 DirectStream,可以使用 SendCompleteBuffer 来发送
- SendCompleteBuffer 支持直接发送 TDFE 数据
- SendCompleteBuffer 可以与 Stream 阻塞模型重叠
如果服务器注册的命令模型是阻塞 Stream,客户端使用 SendCompleteBuffer 发送时是非阻塞响应模式,既发送一个 buffer 数据,也会收到一个 buffer 数据,在这一过程中并不会出现阻塞等待.
- 万兆以太收发大数据如果不使用 CompleteBuffer,一个 100M 的数据可能会让发送端先卡 5 秒并出现 UI 假死,发送出去以后,接收端又会卡顿 5 秒出现响应停顿,这是因为大量的数据复制,编码,压缩,解压缩占用了主线程的计算资源导致,这种机制只能处理体积非常小的数据.
- 在万兆使用 CompleteBuffer 发送一个 100M 数据,从发送到接收,两边的处理延迟可以小于 10ms,这样的服务器模型永远都是立即响应.
- 总结:线程+磁盘缓存+SendNull 阻塞+DirectStream 重叠+Stream 重叠=用 CompleteBuffer 成功解决万兆以太网问题

使用 ZNet 必须知道的主循环三件事

第一件事:遍历并且处理每个 IO 的数据接收流程

ZNet 的物理网络接口大都用独立线程,这个线程是无卡的,当主线程被完全占用,例如正在处理 100M 的压缩+编码任务,这时候接收线程内部仍然处在正常工作中.子线程接收的数据的并不会放在内存中:接收程序会根据数据体量来判断是否用临时文件来暂存数据.

当主循环被触发时,主循环会工作与对应的线程中,主循环永远都是单线程模型,程序会从子线程中取出已经收到的数据,包括临时文件数据,然后进入网络数据粘包处理环节.

在数据粘包处理环节,ZNet 使用了大量内存投影技术来避免内存 copy,这是将内存地址映射成 TMemoryStream,在 ZNet 中内存投影使用 TMS64,TMem64 这类工具.

粘包处理系统会含有 cpu 时间消耗度,由于粘包系统里面包含了序列包和 p2pVM 这类大型子系统,因此 ZNet 给出了 cpu 时间消耗度,该数值达到临界点,粘包处理系统将会中断粘包流程,并且在下次主循环才会继续处理.例如发送了 1 万条命令,粘包临界点是 100ms,当达到 100ms 时粘包只处理了 2000 条命令,那么剩下的 8000 条将会在下次粘包时进行处理.

在 ZNet 的实际运行中粘包流程几乎没有内存 copy,单线程里面的粘包处理能力每秒会达到数十万的处理水平.并不需要开辟线程或则协程在处理.做个线程加速这类想法,很不实际,这不仅会加大 ZNet 的内核复杂度,效率提升也会非常一般,只能是给 Newbie 解决了胡乱高速粘包,而正确的高速粘包,只需要使用 SendCompleteBuffer 发数据进来就行了.

第二件事:遍历并且处理每个 IO 的数据发送流程

ZNet 中的所有发送命令最终都会降落到具体每个 IO 里面

在这些 IO 中,会有个正在等待发送的命令数据队列,这些队列数据,有的会存在于内存,有的会存在于临时文件.

ZNet 会遍历并发送 IO 中的待发队列中的严格序列化数据,这些数据将会放到物理的 IO 待发缓冲区中,这一层的缓冲区就不是 ZNet 可以控制的了.

如果使用 p2pVM,StableIO 这类虚拟化通讯协议,在遍历发送时,严格序列化数据会直接被重新封装,然后再放到物理 IO 缓冲区.

发送的全部物理缓冲数据,是被拓扑网络的信号系统所控制,每个含有 tcp 标签的 ip 包,都需要有一个终端反馈信号(远程接收端,不是内网拓扑),这个包才能够被送到,这种反馈信号就是网络延迟.反馈在 ZNet 里面是发送速度的快慢.

第三件事:管理好每次遍历的 cpu 开销

ZNet 会记录每次训练遍历 IO 的时间开销,达到临界,遍历 IO 将会中止,下次主循环再继续遍历.这样干,当服务器达到一定负载以后,远程响应将会变慢,而服务器本身则是在单线程的主循环中走分片负载的技术路线.

优化数据传输

首先明确优化目标:避免客户端卡住 UI 以及避免服务器卡住主循环,这需要作为一个项目或产品整体对待.例如服务器卡主循环,那么响应速度将会出现延迟,前端发个请求过来,等上 5 秒才响应.

当明确目标后,首要工作是分析卡顿瓶颈,大多数 CS 或 web 型项目,可以直观通过客户端发送的命令来定位瓶颈,例如 GetDataList 命令,出现 5 秒卡顿,直接定位到服务器的 GetDataList 响应环节去就行了.但有时候,ZNet 命令吞吐量出现空前规模,例如数百台服务器之间,以及数千个 IOT 网络设备间的通讯,这些命令密密麻麻,这时就需要 ZNet 提供支持信息

分析瓶颈

如果未使用 C4 框架,需要自己把代码添加到服务器应用或则控制台去.

输出每条命令在服务器的 cpu 耗损

```
ZNet_Instance_Pool.Print_Service_CMD_Info;
```

输出服务器运行状态统计

```
ZNet_Instance_Pool.Print_Service_Statistics_Info;
```

如果使用了 C4 框架,可以直接在控制台输入 `Service_CMD_Info`

在输出命令中,会包含所有命令收发的 cpu 耗损,如果某些信息含有":HPC Thread"字样,表示这条命令使用了 HPC 线程分载,例如, GetDataList:HPC Thread": time 123078ms

表示 GetDataList 在 HPC 线程分载中最长的一次处理运行了 2 分钟

如果使用了 C4 框架,可以通过 `HPC_Thread_Info` 控制台命令,实时监控线程分载

线程分载会给每个线程赋予一条正在执行的命令信息,`HPC_Thread_Info` 会输出 C4 服务器的全部线程状态.然后,结合系统的任务和资源监视工具,一直守着务器运行,基本都能分析出性能瓶颈.

当分析出性能瓶颈以后,接下来的工作就是解决瓶颈

如果服务器走主线程路线,解决瓶颈只有两方面工作,首先是考虑线程分载用 HPC 函数开线程处理命令.其次是考虑在客户端使用 `SendCompleteBuffer` 来替代 `SendDirectStream`.

如果服务器本身就是多线程设计路线,这会需要从锁,结构,流程这些地方下手来搞,如果从整体来优化多线程的服务器,事情很复杂:你会从问题的传导分析再到定位问题点,最后调整流程.最后大概率会使用一个算法来解决问题.例如 ZNet 作者的监控项目在搜索视频时总是等待很久,最终作者设计了一个按时间跨度存视频的加速算法.

服务器堆大以后,一个后台服务也许会到达 10 万行规模,很多优化工作,也会是 fixed bug 的工作.这些优化工作会区分,初期,中期,后期,越靠近初期,fixed bug 的频率越高,到后期,也许整个服务器后台都被推翻重构 1-2 次了,这是一个经验和设计上的问题.

做服务器总是围绕硬件编程

很多开源项目看似都很简单易懂,在真实的服务器项目中,规模也许会比开源项目大上 100 倍.从量变到质变,规模上升 100 倍,就不再是常规技术方案了.

很多人做服务器是调度数据库+通讯系统,如果是做 web,服务器还会包含设计 ui 系统.如果项目规模很小:通讯频率低+通讯数据量小,这种服务器怎么做都没有问题.

当服务器规模开始偏大:通讯频率高+通讯数据量大,将会面临:围绕硬件编程.

6 核 12 超线和 128 核 256 超线,这在硬件定位上是不一样的,它会影响线程和服务器的设计模型,6 核规格硬件几乎无法开出并行程序模型,那怕一个环节开出并行 for,也许整个服务器都会受牵连,6 核的只能开出常规线程对特别繁重的计算环节跑线程分载.当服务器运行于 128 核平台时,就没有 cpu 计算资源的问题了,而是合理安排计算资源,例如,4 线程的并行计算,有时候会比使用 40 个线程速度更快.最后是系统瓶颈,在不使用三方线程支持库情况下,win 系统的单进程最多只能同时使用 64 个超线,只有挂载了 TBB 这类库以后,才能同时使用 256 个超线.多线程,部分线程,单线程,这种服务器在设计之初定位就已经完全不一样.服务器程序的中心是围绕不同时代的硬件趋势,最大的道理硬件平台,框架设计环节是小道,只有明确了围绕硬件为中心,再来做服务器设计和编程,这才是正确的路线.

再比如 8 张 4T 全闪 sdd 配 192G 内存,以及 8 张 16T 的 hdd 配 1TB 内存,这种服务器在数据存储,缓存系统设计上也是不一样的,例如**数据规模到 30 亿条,空间占用到 30T,这种规模基本上 192G 内存会很吃紧**,但不是问题,仔细优化后也能跑,因为数据要**加速搜索或存储提速**永远都是用缓存,而当存储设备的硬件配置使用 hdd 并且容量达到 100T,内存 1TB,这种设计将比 192G 更加需要优化缓存,流量进来以后有可能光是写缓存就直接吃掉 0.99TB 内存,程序在设计之初就已经定位好了用 10G 来 hash 索引,开各种优化算法的结构,这 2 种不同硬件配置,在服务器的系统设计层面,是不一样的:192G 只需要考虑优化缓存规模,1TB 需要考虑优化缓存规模+防止崩溃,因为 hdd 写大数据遇到流量>阵列写入极限后非常容易崩溃,**大阵列的内存一旦用完,阵列的 IO 能力也许会下降到原有能力的 5%,与崩溃无差异,缓存控制(flush)将会是直接提上前台的核心机制之一**,这需要从整体上控制住大数据输入端,阵列写机制,硬件锁这些关键要素.

最后是 gpu,一旦服务器碰上 gpu,支持设备从 cpu 到存储几乎全都会是高配,这时候,服务器在程序设计环节将会彻底脱离古典的单线程方式,流程模型将会被流水线模型所取代,这些流水线会从一个作业系统到另一个作业系统流来流去.这时候 ZNet 程序会全体走 Buffer+线程的路线,并且这种模型只是把数据接进来,计算主体是一堆独立+巨大的计算支持系统.

架桥

架桥是一种通讯模式,有点偏设计模式,它能实实在在提升服务器群的编程效率.
架桥只能工作在带有反馈请求的命令模型中

- SendStream+SendConsole,带有队列阻塞机制的请求,会等响应
- SendCompleteBuffer_NoWait_Stream,不会阻塞队列的请求,不等响应

架桥的工作流程:

- A->发出请求->命令队列开始等待模型
- B->收到请求->请求进入延迟反馈模型->架桥 C
- C->收到请求->C 响应请求
- B->收到 C 响应->B 响应回 A
- A->收到 B 响应,跨服通讯流程完结

架桥是用 1-2 行代码解决繁琐的服务器群间数据传递流程,

```
procedure TDemo_Server.cmd_cb_bridge_stream(Sender: TCommandCompleteBuffer_NoWait_Bridge; InData, OutData: TDFE);  
var  
    bridge_: TCompleteBuffer_Stream_Event_Bridge;  
begin  
    // 架桥就是事件指向,剩下的让桥自动处理  
    bridge_ := TCompleteBuffer_Stream_Event_Bridge.Create(Sender);  
    deploy_bridge.DTNoAuth.SendTunnel.SendCompleteBuffer_NoWait_StreamM('cb_hello_world', InData, bridge_.DoStreamEvent);  
end;
```

在 ZNet-C4 框架中,服务器的种类数十种,它们,架桥技术是以高效方式来享受这些服务器资源.在 C4 框架中的全部服务器都支持架桥与被架桥.

编写 C4 服务器时只管按正常的通讯作业编程,直接考虑处理 C 端,无限堆砌.待完成后开个应用服务器,把所有的服务器以架桥方式全部调度起来使用即可.

ZNet 桥支持

ZNet 的桥支持就是事件原型,响应式通讯都会有反馈事件,让事件指向一个已有的自动程序流程,反馈事件触发时直接自动处理.

- `TOnResult_Bridge_Templet`:桥反馈事件原型模板
- `TProgress_Bridge`:主循环桥,挂接到 ZNet 的主循环后,每次 `progress` 都会触发事件,这种模型在早期 ZNet 还没有解决 `hpc` 分载线程做数据搜索大流程时,主循环桥常被用于分片计算,例如 1000 万的数据搜索在主循环干就是每次 `progress`,搜索 10 万条,以此保证服务器不卡.
- `TState_Param_Bridge`:以布尔状态反馈的桥
- `TCustom_Event_Bridge`:半自动化响应式模型桥,需要编程的桥,例如访问 10 台服务器,待全部访问完,再一次性响应给请求端.C4 大量使用.
- `TStream_Event_Bridge`:`SendStream` 的自动化响应桥
- `TConsole_Event_Bridge`:`SendConsole` 的自动化响应桥
- `TCustom_CompleteBuffer_Stream_Bridge`:半自动化的 `CompleteBuffer` 响应事件桥
- `TCompleteBuffer_Stream_Event_Bridge`: `CompleteBuffer` 的自动化响应桥

C4 启动脚本书写方式

win shell 命令行方式:

```
C4.exe "server('0.0.0.0','127.0.0.1',8008,'DP')" "KeepAlive('127.0.0.1',8008,'DP')"
```

linux shell 命令行方式

```
./C4 \  
"server('0.0.0.0','127.0.0.1',8008,'DP')" \  
"KeepAlive('127.0.0.1',8008,'DP')" \  

```

代码方式

```
C40AppParsingTextStyle := TTextStyle.tsC; //为了方便书写脚本,使用 C 风格文本表达式  
C40_Extract_CmdLine([  
  'Service("0.0.0.0", "127.0.0.1", 8008, "DP")',  
  'Client("127.0.0.1", 8008, "DP")'  
]);
```

C4 启动脚本速查

函数:KeepAlive(连接 IP, 连接 Port, 注册客户端),

参数重载:KeepAlive(连接 IP, 连接 Port, 注册客户端, 过滤负载)

别名,支持参数重载:KeepAliveClient, KeepAliveCli, KeepAliveTunnel, KeepAliveConnect, KeepAliveConnection, KeepAliveNet, KeepAliveBuild

说明:客户端连接服务器,部署型入网连接,如果连接目标不成功会一直尝试,连接成功后会自动启动断线重连模式.在部署服务器群时,主要使用 KeepAlive 方式入网,无论 C4 的构建参数怎么变化,KeepAlive 会总是反复尝试不成功的连接,KeepAlive 方式解决了部署服务器的启动顺序问题,只要在脚本中使用 KeepAlive 入网可以无视服务器部署顺序问题.KeepAlive 不会搜索整个通讯服务器栈,需要在 C4 网络中部署 DP 服务,KeepAlive 这样才能跨服入网.简单解释:使用 KeepAlive 入网 C4,需要挂载一个 DP 服务.

函数:Auto(连接 IP, 连接 Port, 注册客户端)

参数重载:Auto(连接 IP, 连接 Port, 注册客户端, 过滤负载)

别名,支持参数重载:AutoClient, AutoCli, AutoTunnel, AutoConnect, AutoConnection, AutoNet, AutoBuild

说明:客户端连接服务器,非部署型的入网机制,入网失败后无法自动化反复入网,Auto 是自动型入网连接,可以工作于没有 DP 服务的 C4 网络,适用于在有人操作启动的服务器使用,入网一旦成功就会进入断线重连模式.

函数: Client (连接 IP, 连接 Port, 注册客户端)

不支持参数重载

别名,支持参数重载:Cli, Tunnel, Connect, Connection, Net, Build

说明:客户端连接服务器,非部署型的入网机制,入网失败后无法自动化反复入网,Client 函数需要 C4 目标 IP 的网络有 DP 服务才能入网.

函数: Service (侦听 IP, 本机 IP, 侦听 Port, 注册服务器)

参数重载: Service (本机 IP, 侦听 Port, 注册服务器)

别名,支持参数重载:Server, Serv, Listen, Listening

说明:创建并启动 C4 服务器

函数: Wait(延迟的毫秒)

别名,支持参数重载:Sleep

说明:启动延迟,因为 win32 命令行如果不使用 powershell 脚本,处理延迟执行比较麻烦

函数:Quiet(Bool)

说明:安静模式,默认值 False

函数:SafeCheckTime(毫秒)

说明:长周期检查时间,默认值 45*1000

函数:PhysicsReconnectionDelayTime(浮点数,单位秒)

说明:C4 入网以后,如果物理连接断线,重试连接的时间间隔,默认值:5.0

函数: UpdateServiceInfoDelayTime (单位毫秒)

说明:DP 调度服务器的更新频率,默认值 1000

函数: PhysicsServiceTimeout (单位毫秒)

说明:物理服务器的连接超时,默认值 15*60*1000=15 分钟

函数: PhysicsTunnelTimeout (单位毫秒)

说明:物理客户端的连接超时,默认值 15*60*1000=15 分钟

函数: KillIDCFaultTimeout (单位毫秒)

说明:IDC 故障判定,断线时长判定,达到该值触发 IDC 故障,断线的客户端会被彻底清理掉
默认值 h24*7=7 天

函数: **Root** (字符串)

说明:设置 C4 工作根目录,默认值为.exe 文件目录,或则 linux execute prop 文件名录.

函数: **Password** (字符串)

说明:设置 C4 的入网密码,默认值为 DTC40@ZSERVER

UI 函数: **Title** (字符串)

说明:只能工作与 C4 的标注 UI 模板,设置 UI 窗口标题

UI 函数: **AppTitle** (字符串)

说明:只能工作与 C4 的标注 UI 模板,设置 APP 标题

UI 函数: **DisableUI** (字符串)

说明:只能工作与 C4 的标注 UI 模板,屏蔽 UI 操作

UI 函数: **Timer** (单位毫秒)

说明:只能工作与 C4 的标注 UI 模板,设置 UI 环境下的主循环毫秒周期

C4 Help 命令

Help 命令是 C4 内置的服务器维护+开发调试命令.无论是 console 还是 ui,都内置了 help 命令,这些命令是通用的.

命令:help

说明:显示可用命令列表

命令:exit

别名:close

说明:关闭服务器

命令:service(ip 地址, 端口)

重载参数: service(ip 地址)

重载参数: service()

别名:server,serv

说明:服务器内部信息报告,包括物理服务器信息,p2pVM 服务器,连接数量,流量,服务器内置启动参数.如果空参数会简易报告.

命令:tunnel(ip 地址, 端口)

重载参数: tunnel(ip 地址)

重载参数: tunnel()

别名:client,cli

说明:客户端内部信息报告.如果空参数会简易报告.

命令:reginfo()

说明:输出已经注册的 c4 服务,c4 的每个服务都会有对应的 CS 模块,例如 DP 会有 dp 服务器+dp 客户端.

命令:KillNet(ip 地址, 端口)

重载参数: KillNet (ip 地址)

说明:直接以 IDC 故障方式杀掉对应的 c4 网络服务

命令:Quiet(布尔)

重载参数: SetQuiet(布尔)

说明:切换安静模式,在安静模式下,服务器不会输出日常命令执行状态,但出错有提示,例如命令模型执行异常

命令:Save_All_C4Service_Config()

说明:

立即保存当前服务器参数,这是一个服务器扩展参数,当 c4 堆大以后服务器参数太多太多,shell 命令行最长限制是 8192,在正常情况下,根本无法在 shell 命令写太多启动参数,因此 c4 提供了文件形式的参数载入方式,在默认情况下,并没有参数文件

通过 Save_All_C4Service_Config() 可以生成后缀为.conf 的服务器参数文件,.conf 文件存放在当前服务器目录对应的 depnd 子目录中,.conf 是个 ini 格式的配置文件.

如果使用.conf 作为服务器参数来启动 c4,命令行的参数将会被覆盖.

Save_All_C4Service_Config() 多用于首次运行服务器时部署启动参数使用,主要是作用是减少命令行的输入规模.真实系统集成中,命令行达到一个 200,300 字符,这是非常不易于阅读修改的.因此.conf 启动参数是部署 c4 的重要环节.

命令: Save_All_C4Client_Config()

说明:立即保存当前构建完成的所有 C4 客户端参数,作用与 Save_All_C4Service_Config() 基本一致.在系统集成工作中,客户端参数都很少,这是可以直接写进 shell 命令行的,但如果要美化命令行,使其易于阅读,那就用文件参数把.

命令: HPC_Thread_Info()

说明:立即输出当前进程中的全部 TCompute 线程实例,Z 系线程一律使用 TCompute 创建与执行,并且每个线程都会有个 thread_info 的字符串标识符,用于识别这条线程的作用.在 ZNet 中线程会非常繁多,有 HPC 分载线程,CompleteBuffer 后台解码编码线程,ZDB2 线程.如果直接使用 RT 库自带的 TThread,那么 HPC_Thread_Info() 是不会输出该线程状态的.

该命令多用于服务器调试,分析性能瓶颈,找 bug 时使用

命令:ZNet_Instance_Info()

别名: ZNet_Info()

说明:立即输出 ZNet 的全部 IO 实例,包括物理连接,p2pVM 连接.多用于诊断连接状态,分析 C4 入网时遇到的问题.

命令: **Service_CMD_Info()**

别名: **Server_CMD_Info()**

说明:立即输出服务器中全部命令模型的 cpu 消耗度统计状态,这些命令会非常多,数百个.多用于在分析性能瓶颈定位用. **Service_CMD_Info()**也会包含发送命令的次数统计,但不包含发送命令的 cpu 消耗度.

命令: **Client_CMD_Info()**

别名: **Cli_CMD_Info()**

说明:立即输出全部客户端命令模型的 cpu 消耗度统计,与 **Service_CMD_Info()**格式几乎相同,因为 C4 是个交互网络,客户端统计会折射出服务器的延迟.

命令: **Service_Statistics_Info()**

别名: **Server_Statistics_Info()**

说明:立即输出全部服务器的内部统计信息,包括 IO 的触发频率,加密计算频率,主循环频率,收发的数据量等等关键信息,服务器会包括物理服务器+p2pVM 服务器

命令: **Client_Statistics_Info()**

别名: **Cli_Statistics_Info()**

说明:立即输出全部客户端的内部统计信息, 输出格式与 **Service_Statistics_Info()**几乎相同

命令: **ZDB2_Info()**

说明:立即输出 ZDB2 的数据库状态,ZDB2 是一套分层次架构的数据库系统,目前 ZDB2 已经进步到第三代体系,这里的 **ZDB2_Info()**也是输出第三代 ZDB2 体系,在 C4 框架集成的 FS2,FS,这类服务,凡是 2021 年做出的 C4 服务,都是第二代 ZDB2 体系,无法被 **ZDB2_Info()**统一化的输出状态. **ZDB2_Info()**输出三代体系的信息量和设计非常庞大,这里只能一笔带过:在第六代监控的数据库和后台用 **ZDB2_Info()**看状态会是一个好办法.

命令: **ZDB2_Flush()**

说明:将 ZDB2 写缓存立即刷入物理设备, 使用信息与 **ZDB2_Info()**都有非常庞大的信息量,这里只能一笔带过:在第六代监控的数据后台调试阵列系统硬件时用的命令,需要结合磁盘缓存监控,内存监控,物理 IO 监控一起来使用.作用是分析出阵列系统的 IO 瓶颈.

ZNet 内核技术-锁复用

Critical-Section 是操作系统基于硬件的线程锁技术

进程中所包含的线程越多,Critical-Section 就会对应越多,在系统监视器,都可以看到线程数量+进程的句柄数量,这两者数量多了以后,整个系统也许会不太稳定,至少在分析进程或则系统崩溃时,目标进程的线程+句柄是 2 个非常重要指标.

通常来说,每个线程会对应至少 1 个以上的 Critical-Section 句柄,看具体流程编写.

Z 系内核对 Critical-Section 是走的复用路线,ZNet 服务器运行起来会在监视看到句柄峰值,但是这不是真实 Critical-Section,需要通过命令 c4 控制台输入 hpc_thread_info 才能看到真实的 Critical-Section 和线程状态.

ZNet 内核技术-Soft Synchronize

Soft Synchronize 技术是仿真 rtl 的主线程 Synchronize.

ZNet 的设计机制大量依赖主线程,因此大量使用 Thread Synchronize 体系,在 ZNet 的异步通讯库中 DIOCP/CrossSocket 线程间调用也使用了 Thread Synchronize 机制,控制线程启停等操作.其中用的比较多的还是 WaitFor 线程间的互斥等待,假如队列没有处理完成,给线程发 exit 命令容易卡在里面,这时候问题往往由外面程序没有正确清空线程间执行调用,清理线程间执行程序这种操作,其实就是 CheckSynchronize,这是一个主线程专用的同步队列执行调用.凡是线程中出现了 Synchronize 操作,只能通过 CheckSynchronize 响应,在 vcl form 体系中这是有 application 自动调用的,如果绕开 application 这需要掌握主循环技术.

例如给出了 Thread.OnTerminate 中,如果外面不给 CheckSynchronize,会一直不触发事件.

Soft Synchronize 解决了线程间的事件传递机制,同时替代 CheckSynchronize.

内核:Check_Soft_Thread_Synchronize

执行仿真主线程 Synchronize 代码,不会执行 RTL 系统 Synchronize 代码.Z 系一律使用该方式处理执行 Synchronize 代码,包括 DIOCP/Cross/ICS8/ICS9/Indy/Synapse.例如当异步库 cross/diocp 使用 waitfor 操作,这会让 wait 过程中,执行仿真主线程的 Synchronize 一直工作.

当编译开关 Core_Thread_Soft_Synchronize 被关闭时,将使用 RTL 系统 Synchronize 机制.

该 API 主要支持程序需要在双主线程环境下运行.

内核:Check_System_Thread_Synchronize

执行 RTL 系统 Synchronize 代码,同时也会执行仿真主线程 Synchronize 代码

自动处理状态,无视编译开关 Core_Thread_Soft_Synchronize 打开或关闭.

该 API 主要支持程序在主线程环境下运行.

ZNet 内核技术-双主线程

ZNet 可以在单进程同时开两个主线程,当双主线模型启动以后,会发生如下事情:

- ZNet 全系和 ZNet 包含的各种库,一律工作于次主线程.
- RTL 全系,包括原生 lcl,vcl,fmf,一律工作于原主线程.
- 次主线和原主线会各自维持自己的主循环,主循环技术这里省略
- 次主线和原主线互相访问数据需要使用 Synchronize 技术
- 双主线程技术可以支持 win/android/ios 以及 fpc 所构建的 Linux 程序
- 带 UI 的程序,跑服务器不会再有卡顿感
- 把 ZNet 放在 1 个 dll/ocx 运行,等同于开了 2 个 exe,其中 exe 与 dll 各走一条主线程
- 次主线程完全可以跑 http,c4,znet

RTL 原主线程同步到次主线程

在双主线程模式以后,从 RTL 主线程访问次主线程的数据

```
TCompute.Sync(procedure
```

```
begin
```

```
    // 这里访问 ZNet 里面的数据,包括处理 c4,cross,diocp,ics 这些通讯数据
```

```
end);
```

次主线程同步到 RTL 原主线程

在双主线程模式以后,就是从 ZNet 访问 RTL 主线程的数据,就是从 ZNet 访问 VCL/FMX

```
TThread.Synchronize(TThread.CurrentThread, procedure
```

```
begin
```

```
    // 这里访问和修改 vcl/fmf,UI 在这些数据
```

```
end);
```

双主线程开启以后的主循环

ZNet 次主线程 API, `Check_Soft_Thread_Synchronize`, 位于 Z.Core.pas 库

RTL 原主线程 API, `CheckSynchronize`, 位于 vcl-System.Classes.pas/lcl-classes.pas 库

ZNet 性能工具箱使用指南

CPS 工具箱=Caller Per second tool.所有 cps 计数周期为 1 秒.位于 Z.Core.pas 库.

- **CPS_Check_Soft_Thread**:次主循环性能计数器.
- **CPS_Check_System_Thread**:RTL 主循环性能计数器.

访问方法, CPS_Check_Soft_Thread.CPS,该值为每秒调用次数

ZNet 的所有实例都内置了 CPS 性能计数器,用于计算服务器主循环每秒调用频率以及 cpu 占用..

性能瓶颈分析

启动任意 C4 程序,命令行敲 **hpc_thread_info**,得到如下反馈

```
RTL Main-Thread synchronize of per second:0.00
Soft Main-Thread synchronize of per second:167.32
Compute thread summary Task:16 Thread:16/80 Wait:0/0 Critical:336/46989 19937:16 Atom:0 Parallel:0/0 Post:0 Sync:0
```

RTL Main-Thread synchronize of per second:0.00,RTL 主循环每秒调用次数

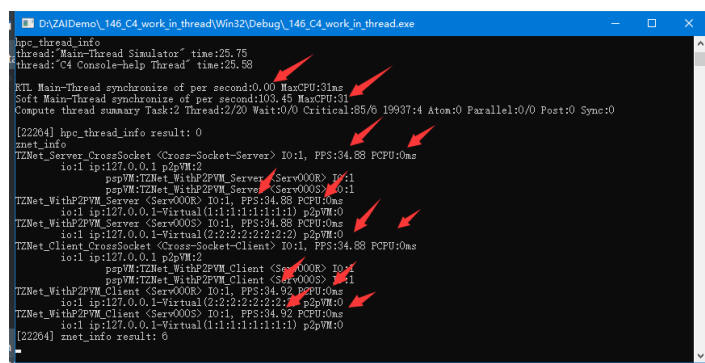
对应 **CPS_Check_System_Thread**

Soft Main-Thread synchronize of per second:167.32,次主循环每秒调用次数

对应 **CPS_Check_Soft_Thread**

结果:这个 C4 程序,使用的是次主循环技术,主循环每秒发生 167 次调用,调用频率越高说明流畅度越好,如果程序发生卡顿,就需要检查卡顿的点:也许某个函数占用了传导给了主循环,导致 CPS 过低,尤其开了 timer 这类事件的程序.

命令行敲 **znet_info**,得到如下反馈



PPS:ZNet 中的 progress 每秒调用频率

PCPU: ZNet 中的 progress 最大 cpu 消耗

结果:定位出主循环的长流程到底卡在哪个实例中,这些实例可以是服务器,也可以是客户端.定位完成后,通过 **Service_CMD_Info** 和 **Client_CMD_Info** 找执行命令的卡顿瓶颈.

如果程序未使用 C4,可以使用 API, **ZNet_Instance_Pool.Print_Status**,直接输出状态

排除 ZNet 重叠 Progress

首先 progress 具有自动防死循环机制,progress 包 progress 不会死循环.

C4 已经优化过 progress 重叠问题,每次调用 C40Progress 可以确保每个 ZNet 实例只会触发一次 Progress

在非 C4 框架中 Progress 可以被 p2pVM,DoubleTunnel 自动触发,一个主线程循环可能会引发 2-5 倍的 progress,这是无意义的消耗,如果服务器负载多了,反复重叠的 progress 会让分片负载无法准确估算.当需要精确优化这是必须解决的问题.

使用如下程序范式

- Server.Progress;
- Server.Disable_Progress; 这里屏蔽调以后,后面不会触发 server.progress
- other.progress;
- Server.Enabled_Progress;

p2pVM 的物理隧道实例每次 progress 会自动遍历里面的全部 p2pVM 虚拟连接.

当干完这些事以后,使用 ZNet_Instance_Pool.Print_Status 查看 cps 变化,如果 ZNet 实例的 cps 值与主循环 cps 相近,那 progress 基本没问题.接下来测试一下连接,处理命令,待全部通过,那么解决重叠 progress 问题宣告完结.

敬畏服务器主循环 progress

几乎所有的服务器优化工作都会面临主循环问题,这里涉及了非常多的解决办法

- **线程**:如果主循环的代码支持线程安全,那么用线程会很不错,线程安全不是锁住就安全,而是线程+进程都不会因为主循环开在线程中而发生卡锁.
- **分片**:分片技术是把计算量切割出来,每次主循环只运行一部分
- **开状态机**:状态机是让主循环在某种环境下,直接省略不处理某些代码.尤其事涉及到 for,while 这类流程
- **结构和算法优化**:在主循环中会处理大量的结构,对结构的优化,例如 hash,biglist,可以有效提升主循环效率.
- **CPU 指令级别的优化可以无视**:例如 sse,avx,这种优化难搞不说,很难数倍提升,无法和算法级优化相提并论,算法优化普遍起步就是 10 倍提升.

主循环是主线程的命脉,90%服务器的调度程序都使用主线程来完成,子线程和协程大多用于分担某些特殊任务.例如在 pas 圈很多服务器喜欢上一个 ui,随时看到服务求的运行状态,这种 vcl/fmx/lcl 路线的 ui 都是跑在主线程下,这会被主循环深深影响.

而全线程化的服务器,例如 erlang,go,会有一个线程调度问题,这里会使用许多复杂的程序机制来控制线程间的协作问题,并且全线程化计算的服务器并不会让项目变得十分流畅,更不会天下无敌,因为服务器底层被硬件极限影响.

主循环+子线程,未来仍然会是服务器的主要模型.主循环没问题可以等同于已经解决好了 50%的服务器性能瓶颈!!

如何推翻使用 ZNet 的项目

如果之前使用 ZS 走物理双通道的项目,推翻直接换 C4.

如果之前使用非 C4 双通道,推荐直接换 C4.

如果项目已经是 C4,可以换个注册名,RegisterC40('MY_Serv', TMY_Serv, TMY_Cli),然后基于 C4 再重新开一个项目 RegisterC40('MY_Serv2.0', TMY_Serv2, TMY_Cli2).简单来说就是大改走增量,单元名加个版本号,Reg 一下新服务就行了.

如何使用 ZNet 开发 web 类项目

数据通讯层直接 ZNet 跑,UI 层用 webapi 访问 ZNet 的项目即可.例如 Post+Get 基本能覆盖 90%的 webapi 需求,ZNet 自带一个 webapi 的 demo 项目.

ZNet 与 http 和 web

ZNet=大型 CS 服务器

http=通讯协议

web=全球广域网,互联网

web 包含了 ZNet,在 web 环境下用 apache,nginx 桥通讯模块的大型网站比比皆是,或许读者有空可以试试用二级域名或则域服务器来做桥接模块和分流,内部如果涉及到大数据或则复杂协议,直接用 ZNet 做个通讯层包 api 给 web 用.

文本最后来一个极简 C4 的 CS demo

```
program _145_VeryEasyC4Project;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  Z.Core, Z.PascalStrings, Z.UPascalStrings, Z.UnicodeMixedLib, Z.DFE, Z.Parsing, Z.Expression, Z.Opcode,
  Z.Net, Z.Net.C4, Z.Net.C4_Console_APP;

type
  TMY_Serv = class(TC40_Base_NoAuth_Service);
  TMY_Cli = class(TC40_Base_NoAuth_Client);

begin
  RegisterC40('MY_Serv', TMY_Serv, TMY_Cli);
  if C40.Extract_CmdLine(TTextStyle.tsC, [
    'Service("0.0.0.0","127.0.0.1", 9000, "MY_Serv")', 'Client("127.0.0.1", 9000, "MY_Serv")']) then
    C40.Execute_Main_Loop;
  C40.Clean;
end.
```

全文完.

2023-10-6

by.qq600585