

# ZNet 使用手册

文档版本:1.5

更新时间:2024-7-4

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

2023-10-11 追加性能工具箱资料

2023-10-12 追加 TCompute 概念+结构体概念,用概念说事因为代码无法讲完,代码可由大向小靠自己挖掘.

2023-10-12 追加编译类技术描述,具体代码大家自己去挖掘把

2023-10-17 追加 XNAT 小节

2023-11-21 追加 ZDB2 大数据和 C4 入网小节

2023-12-25 追加主循环中微泄漏分析方案小节

2024-1-8 追加 Z.FragmentBuffer 库的实现和原理小节

2024-1-26 追加内核库启动技术小节(许多原理和机制)

2024-7-4 追加非线性流程技术小节(简单介绍)

2024-8-8 追加 FS3 设计小节

2025-1-20 追加 ZNet 队列理论小结

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 .....	5
ZNet 的 6 种命令收发模型 .....	5
ZNet 队列理论 .....	6
ZNet 双通道模型 .....	7
ZNet 线程支持 .....	7
HPC Compute 线程分载方案 .....	7
万兆以太网支持 .....	8
使用 ZNet 必须知道主循环三件事 .....	9
第一件事:遍历并且处理每个 IO 的数据接收流程 .....	9
第二件事:遍历并且处理每个 IO 的数据发送流程 .....	9
第三件事:管理好每次遍历的 cpu 开销 .....	9
优化数据传输 .....	10
分析瓶颈 .....	10
当分析出性能瓶颈以后,接下来的工作就是解决瓶颈 .....	10
做服务器总是围绕硬件编程 .....	11
架桥 .....	12
ZNet 桥支持 .....	13
C4 启动脚本书写方式 .....	13
win shell 命令行方式:.....	13
linux shell 命令行方式 .....	13
代码方式 .....	13
C4 启动脚本速查 .....	14
函数:KeepAlive(连接 IP, 连接 Port, 注册客户端) .....	14
函数:Auto(连接 IP, 连接 Port, 注册客户端).....	14
函数: Client (连接 IP, 连接 Port, 注册客户端) .....	14
函数: Service (侦听 IP, 本机 IP, 侦听 Port, 注册服务器) .....	14
函数: Wait(延迟的毫秒) .....	15
函数: Quiet(Bool) .....	15
函数:SafeCheckTime(毫秒) .....	15
函数:PhysicsReconnectionDelayTime(浮点数,单位秒) .....	15
函数: UpdateServiceInfoDelayTime (单位毫秒) .....	15
函数: PhysicsServiceTimeout (单位毫秒) .....	15
函数: PhysicsTunnelTimeout (单位毫秒) .....	15
函数: KillIDCFaultTimeout (单位毫秒) .....	15
函数: Root (字符串) .....	16
函数: Password (字符串) .....	16
UI 函数: Title (字符串) .....	16
UI 函数: AppTitle (字符串) .....	16
UI 函数: DisableUI (字符串) .....	16
UI 函数: Timer (单位毫秒) .....	16
C4 Help 命令 .....	17
命令:help .....	17
命令:exit .....	17
命令:service(ip 地址, 端口) .....	17
命令:tunnel(ip 地址, 端口) .....	17
命令:reginfo() .....	17
命令:KillNet(ip 地址, 端口) .....	17

命令:Quiet(布尔) .....	17
命令:Save_All_C4Service_Config().....	18
命令: Save_All_C4Client_Config() .....	18
命令: HPC_Thread_Info()	18
命令:ZNet_Instance_Info()	18
命令: Service_CMD_Info()	18
命令: Client_CMD_Info()	18
命令: Service_Statistics_Info()	19
命令: Client_Statistics_Info().....	19
命令: ZDB2_Info()	19
命令: ZDB2_Flush()	19
ZNet 内核技术-锁复用 .....	20
ZNet 内核技术-Soft Synchronize .....	20
内核:Check_Soft_Thread_Synchronize .....	20
内核:Check_System_Thread_Synchronize .....	20
ZNet 内核技术-双主线程 .....	21
RTL 原主线程同步到次主线程 .....	21
次主线程同步到 RTL 原主线程 .....	21
双主线程开启以后的主循环 .....	21
ZNet 性能工具箱使用指南 .....	22
性能瓶颈分析 .....	22
排除 ZNet 重叠 Progress .....	23
敬畏服务器主循环 progress .....	23
ZNet 的内核技术:TCompute 线程模型简介绍 .....	24
ZNet 的内核技术:结构体系简单介绍 .....	25
ZNet 的内核技术:简单说下结构组合拳 .....	26
ZDB2 如何解决 Stream 写保护状态下的仿真读写 .....	27
回顾:设计泛结构 TBigList<>.....	28
回顾:设计脚本引擎 ZExpression .....	28
ZNet 的母体移植技术:Z.Parsing.....	29
OpCode 中的非线性流程支持技术: TOpCode_NonLinear.....	30
聊聊 TOpCode_NonLinear 的线程支持设计 .....	31
TOpCode_NonLinear 的衍生 .....	31
从交换机到拓扑简单说说 XNAT .....	32
性能分析:当 ZDB2 的数据规模达到 1 亿条 .....	33
再说 C4 入网机制 .....	34
当 C4 入网后需要干什么事情 .....	35
ZNet 内置 CPM 可以代替 FRP+Nginx.....	35
C4 主要用于大项目吗? .....	35
主循环内存微泄漏分析 .....	36
内核库在启动时都做了什么事情 .....	37
FS3 是如何被设计出来的 .....	41
如何推翻使用 ZNet 的项目 .....	42
如何使用 ZNet 开发 web 类项目 .....	42
ZNet 与 http 和 web.....	42
文本最后来一个极简 C4 的 CS demo.....	42



# 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

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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,万兆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 队列理论

队列机理叙述,当 ZNet 使用 SendStream,SendConsole 时,如果远程没有反馈,那么所有的队列都是等待状态.这种情况在 1 对 1 连接中,无伤大雅,大不了向应速度慢点,系统仍然能正常工作.

如果是群集服务器,1000 个前端发出请求,服务器 1 负责转发请求给服务器 2

如果服务器 1 使用 SendStream,SendConsole 机理发送,那么 1000 个前端会出现长时间等待,诸如一个请求 5 分钟才会响应.

ZNet 中的 SendCompleteBuffer\_NoWait\_Stream 是响应模型,但不会影响后续的队列,这里需要特别区分

- SendStream, SendConsole: 使用单 IO 工作, 无需双通道模型, 响应前后续队列一律等待中, 支持延迟响应
- SendCompleteBuffer\_NoWait\_Stream: 必须使用双通道模型, 不影响队列, 支持延迟响应

回到上面的问题,

如果是群集服务器,1000 个前端发出请求,服务器 1 负责转发请求给服务器 2

如果服务器 1 使用 SendCompleteBuffer\_NoWait\_Stream 机理发送,那么 1000 个前端会立即响应.

服务器 1 负责转发给服务器 2 使用桥技术

SendStream, SendConsole, SendCompleteBuffer\_NoWait\_Stream 都需要使用对应的桥类

```
. | { tool: client - bridge templet }
- | TOnResult_Bridge_Templet = class(TCore_Object_Intermediate) ...
| |
| { tool: client - bridge }
0 | TOnResult_Bridge = class(TOnResult_Bridge_Templet) ...
| |
| TOnResultBridge = TOnResult_Bridge;
| |
| { tool: client and service - free bridge }
- | TProgress_Bridge = class(TCore_Object_Intermediate) ...
| |
| { tool: state bridge }
- | TState_Param_Bridge = class(TCore_Object_Intermediate) ...
| |
| { tool: Service - free bridge }
- | TCustom_Event_Bridge = class(TCore_Object_Intermediate) ...
| |
| TCustomEventBridge = TCustom_Event_Bridge;
| |
| { tool: Service - stream bridge }
- | TStream_Event_Bridge = class(TCore_Object_Intermediate) ...
| |
| { tool: Service - console bridge }
- | TConsole_Event_Bridge = class(TCore_Object_Intermediate) ...
| |
| { tool: Service - free complete-buffer stream bridge }
- | TCustom_CompleteBuffer_Stream_Bridge = class(TCore_Object_Intermediate) ...
| |
| { tool: Service - complete-buffer stream bridge }
- | TCompleteBuffer_Stream_Event_Bridge = class(TCore_Object_Intermediate) ...
| |
| { tool: p2pVM bridge }
2 | TP2PVM_CloneConnectEventBridge = class(TCore_Object_Intermediate) ...
| |
```

# 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,甚至是并行程序,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

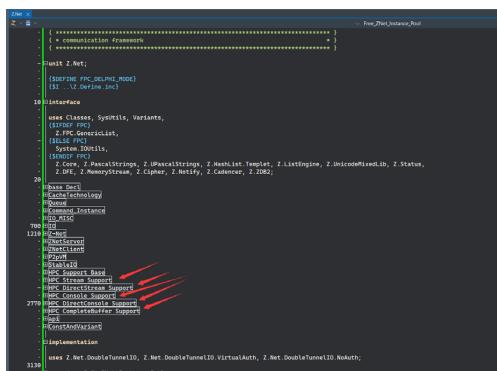
在 ZNet 中以 HPC 开头的折叠代码就是线程分载解决方案

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



```
znet.cs
[...]
{
    /* communication framework */
}
}

partial class ZNet
{
    [STAThread]
    static void Main(string[] args)
    {
        [...]
    }
}

interface
    [...]
    class Classes, SysUtil, Variants,
    {
        [...]
    }
    class SystemUtil,
    {
        [...]
    }
    class FileUtil,
    {
        [...]
    }
    class CacheTechnology
    {
        [...]
    }
    class Command_Instanced
    {
        [...]
    }
    class ID
    {
        [...]
    }
    class IObjectServer
    {
        [...]
    }
    class IObjectServer2
    {
        [...]
    }
    class ISocketIO
    {
        [...]
    }
    class IIPCStream
    {
        [...]
    }
    class IIPCStreamSupport
    {
        [...]
    }
    class IIPCConsoleSupport
    {
        [...]
    }
    class IIPCDirectControlSupport
    {
        [...]
    }
    class IIPCVarVariant
    {
        [...]
    }
    class Implementation
    {
        [...]
    }
}

uses Z_Net.DoubleTunnelIO, Z_Net.DoubleTunnelIO.VirtualAuth, Z_Net.DoubleTunnelIO.NoAuth;
```

The screenshot shows a code editor with the file 'znet.cs' open. Several red arrows are drawn from the text above to specific parts of the code, highlighting the 'IIPCStreamSupport', 'IIPCConsoleSupport', 'IIPCDirectControlSupport', and 'IIPCVarVariant' interfaces. These interfaces are part of the ZNet communication framework, which is described as being used for message passing between threads.

# 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 成功解决万兆以太问题

# 使用 ZNet 必须知道主循环三件事

## 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,一直守着服务器运行,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,哪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.全线程,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_Templat`: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 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 个非常重要的指标.

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,其实这就是 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/diopc 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,fmx`,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/fmx,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

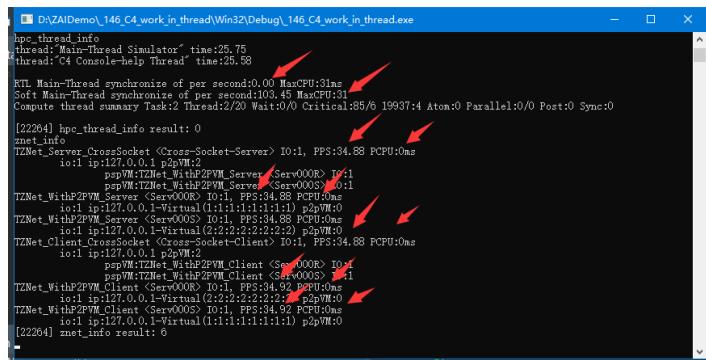
对应 **CPS\_Check\_System\_Thread**

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



```
D:\ZAI\Demo\146_C4_work_in_thread\Win32\Debug\146_C4_work_in_thread.exe
hpc_thread_info
Threads: "Main-Thread Simulator" time:25.75
Threads: "C4 Console-Help Thread" time:28.38
RTL Main-Thread synchronize of per second:0.00 MaxCPU:3ms
Soft Main-Thread synchronize of per second:103.49 MaxCPU:31
Compute thread summary Task:2 Thread:2/20 Wait:0/0 Critical:185/6 19937:4 Atom:0 Parallel:0/0 Post:0 Sync:0
[22244] hpc_thread_info result: 0
znet_info
TZNNet_Server_CrossSocket <Cross-Socket-Server> IO:1 PPS:34.88 PCPU:0ms
  ioc1 ip:127.0.0.1 p2pVM:2
    ppVM:TZNNet_WithP2PVM_Server <Serv000R> IO:1
    ppVM:TZNNet_WithP2PVM_Server <Serv000S> IO:1
TZNNet_WithP2PVM_Server <Serv000R> IO:1 PPS:34.88 PCPU:0ms
  ioc1 ip:127.0.0.1-Virtual(1:1:1:1:1:1) p2pVM:0
TZNNet_WithP2PVM_Server <Serv000S> IO:1 PPS:34.88 PCPU:0ms
  ioc1 ip:127.0.0.1-Virtual(2:2:2:2:2:2) p2pVM:0
TZNNet_Client_CrossSocket <Cross-Socket-Client> IO:1 PPS:34.88 PCPU:0ms
  ioc1 ip:127.0.0.1 p2pVM:2
    ppVM:TZNNet_WithP2PVM_Client <Serv000R> IO:1
    ppVM:TZNNet_WithP2PVM_Client <Serv000S> IO:1
TZNNet_WithP2PVM_Client <Serv000R> IO:1 PPS:34.92 PCPU:0ms
  ioc1 ip:127.0.0.1-Virtual(2:2:2:2:2:2) p2pVM:0
TZNNet_WithP2PVM_Client <Serv000S> IO:1 PPS:34.92 PCPU:0ms
  ioc1 ip:127.0.0.1-Virtual(1:1:1:1:1:1) p2pVM:0
[22244] znet_info result: 0
[22244]
```

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+主循环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
- **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!!

# ZNet 的内核技术:TCompute 线程模型简介绍

TCompute 并不是线程技术,而是一种使用线程编程的规范模型.

TCompute 有非常庞大的下游依赖体系,没有 TCompute,这些下游体系将会罢工

- Z-AI:AI 系统中的 GPU 驱动是线程绑定的,DNN-Thread 技术是在 GPU 和 CPU 各开一个线程,并且让其发生绑定关系,例如在识别时会传递一张图给 gpu,这是调用线程 api,再才是在线程中执行 `cuda copy`,最后 `caller gpu` 用 dnn 库做并行计算和取结果,整个 IO 过程都是线程化工作,而主线程走识别 IO 的流程大都属于 demo 演示 api 和机制时才会使用.正规 gpu 程序线程技术都是成规模来使用.
- AI 工具链:由于 AI 涉及大数据领域,所有的数据操作,不太可能是瞬间完成,因此都是开线程,其流程为,锁 UI,运行线程处理,UI 开锁.最典型的应用就是 `AI_Model_Builder`,经常会出现一个粒度级数据操作耗时好几分钟.
- ZDB1:这是一个古老的链条式数据库体系,在 2017 年,做过一次大规模升级:将 ZDB1 的查询流程封装成了 pipe,然后用主线程 `synchronize` 机制做查询调度.这里提示一下,在使用 `zdb1` 时,只要在主线程 `Check_Soft_Thread_Synchronize (100)`,查询速度就会提升至少一倍.次主线 `soft synchronize` 机制效率会优于 rti 库 `CheckSynchronize`.然而 ZDB1 的问题也是很明显的:2017 年对于线程的深度掌握不够,导致有形成体系的方案,只能算昙花一现,大都作为数据库伴侣,文件打包,安装程序等等小功能来使用.无法纵深!
- ZDB2:可以纵深技术体系,未来是立体形式,整个方案作为数据应用模型前后贯穿了 3 年在设计完善,光是筑基,就经历了 3 代应用体系,在目前被高级泛型结构+先进线程技术加持后,发展路线可基本明确:大数据地基支持技术体系,在 ZDB2 体系,一个数据库,可以由 10 张阵列盘,或则 10 个阵列系统共同负载运行,而驱动这些大阵列的所使用的技术方案就是线程,每一个数据库文件 api 都在一个独立线程中工作.ZDB2 在运行中,内部可以从数十到数百线程不等,即使在 10TB 规模的小数据库 IO 线程也可以吃掉>100GB 内存+>20 核的 cpu,大家不要用阵列系统的思维去看待 ZDB2,数据引擎系统和文件无法直接比较.
- 并行程序:并行程序是现代化流程的必须具备的技术方案,然而并行程序也有非常明显的问题,`fpc/d` 所使用的并行支持库并不理想:无法支持指定的相关性核心和超线程,甚至无法指定每次启动并行线程数,而最大的问题还是库的兼容和并行粒度模型,例如 `for` 并行粒度可以分块并行也可以折叠并行,这些地方都必须统一起来才能堆大,否则并行程序只能作为功能点来解决局部加速问题,不能作为现代化技术的组成部分,因为这无法堆大.内核并行技术在空间+时间复杂度+机理能优于 D 系+LCL 系.
- 第六代监控体系:第六代监控的 AI 服务器端,一台准 gpu 服务器+一个 AI 服务器应用程序,可以带 80 路 4k 视频,而每秒数据量计算公式为: $3840*2160*4*25*80=$ 大约每秒需要处理的数据量为 60G.这种变态的数据规模,需要会对拓扑+交换机+网络+线程+NUMA+CPU+GPU 整个架构有非常深入的把握和实践才能做出流程方案.并且这种流程无法做到一步到位,需要先从局部的单独环节,挨个模拟验证,挨个解决瓶颈和优化,然后才能组合流程.做成这件事的结果是算力成本直接下将 5-10 倍,这是从地狱到天堂.
- 全服务器体系:服务器主循环和线程是千丝万缕的关系,主循环一旦遇到带计算量的流程卡 30 秒会是家常便饭,人会一直守着服务器运行,观察状态,后面就是优化工作,可以这样来说,原生的 `TThread` 缺乏堆砌和调度机制,只有规范起来才可以满足线程间的互调互等,TCompute 是从计算机去机理挖掘出的线程模型,可以真正将线程系统堆大并把 `d/fpc` 规范统一起来.

# ZNet 的内核技术:结构体系简单介绍

这里的结构并不是计算机科学中的数据结构,而是算法应用的基础结构体系.

**算法应用是一种计算工程,这将需要统一化,标准化,可探索,有规律性,有社会性,结构体系是计算工程数据源头,是设计算法的前置设计.因为算法程序会需要人类付出时间代价,这些时间一旦达到某个度,算法方案也许就不太理想了.结构体系好比是一种把握 solve 的准备工作,没有这种准备,算法程序会变得举步维艰,任何想法的变现都需要时间,甚至很多时间.**

**以 ZDB2 为例,ZDB2 的核心工作只负责数据 IO,这些 IO 都是在非常复杂的调度下工作,这些工作的目的,就是给算法提供原始数据,算法再把数据转换成数据源,然后,才是计算流程.好比从 ZDB2 载入 10 亿条数据,用算法做给 10 亿某个 key 做一个加速,实现这件事情的流程是用泛结构,例如 TCritical\_Big\_Hash\_Pair\_Pool,因为 ZDB2 的 IO 全是线程化的,需要带锁的 hash 泛结构,这时候,在 IO 里面往 hash 结构堆数据指针就能完成一个最简单的 10 亿量的毫秒级查询了.待深入以后,就是在泛结构扩展,删除,修改,统计等等功能,而这一切,都是用流程来操作结构.当这件事被 solve,就完成专用数据引擎了,这和通用 DB 引擎不同,专用引擎是上天下地无所不能,只要愿意,可以拿 GPU 跑 sort+sum,而计算速度,性能,我想通用 DB 引擎只能望其项背.**

**以 ZNet 为例,在 ZNet 框架中,有大量的队列机制,例如 progress 里面遍历 TPeerIO 是先把所有的 IO 指针都 copy 到一个容器,然后再遍历容器,如果发现 cpu 时间消耗过大,就退出来,形成分片处理机制,待下次 progress 会继续处理这个 IO 容器,当容器全部处理完成再重新 copy 容器指针,并开启下一次的分片.另一方面,ZNet 的发送,接收,机制,也是用的容器,并不是往一个 stream 里面无脑 copy.当队列容器这种结构被大量使用以后,如果用 TList 机制处理队列就很废了,每次抽取首队列,都要经历一些 copy,相信 pas 圈很多人都埋怨过 TList,但 TList 连贯的 1D 数据空间,一旦发生队列抽取和删除,优化它就只能改指针,否则就是 copy,这种效率非常的蛋疼,最后,TList 的长度限制也是基本无解的.TList 的最大优点是使用简单,在 UI 这类小数据规模项目,没问题,放到服务器还是算了把..... ZNet 使用的数据容器都是链结构的,这种链结构是由小块内存通过 next 指针串联起来,非常适合队列抽取需求,在内核库多以 TBigList, TOrderStruct 来命名,试想一下,粘包流程,1 个包 1k,10M 数据就是 10000 个队列包,在高流量下,TList 计算能力和 TBigList 无法相比,这一差距,用测试程序跑出来会是 2000 倍.**

**以 Learn 统计学的方法为例,结构体系可以当成是 IO 语言,就是输入一种结构,输出又是一种结构.例如分类算法,这些算法原理基本上可以算初中生作业,先用某些计算方法,或则随机方法,生成一些 seed 形式的数据,然后再围绕 seed 做搜索和匹配,所有的分类算法,基本都是这种思路,变来变去,或许这会天马行空,但流程思路往往很简单!而数据接入和输出,这才是最麻烦的计算,在 600585 看来项目中的完整分类流程,前置处理 45%,计算 10%,后置处理 45%,总结一下:结构体系在统计流程中的占比会大于算法.大家平时看到调用一个 api,得到了 solve,流程直接封装,包几十个结构+类也很正常,这已经不是统计算法了,是解决方案,内部走的是 1,2,3,4,5 序列步骤来解决问题.人脸识别,推荐算法就是这种模型.**

**以线程为例,线程是个很大的体系,现代化程序无所不用线程,而线程和线程间的通讯,凭空写出来的结构是无法堆起来做项目的,线程互调互等,TTThreadPost,TSoft\_Synchronize\_Tool,这些泛结构在使用时基本都能 1-3 行解决互调互等,极简使用,要不然怎么能堆大?**

ZNet 体系中使用频率最高的大结构体: **TBigList<>, TBig\_Hash\_Pair\_Pool<>**

ZNet 体系常用小结构体: **TOrderStruct<>, TAtom<>, TPair<>**

ZNet 的泛结构常用 Hash 库 **Z.HashList.Ttemplate**

ZNet 经典链表库 **Z.ListEngine**

# ZNet 的内核技术:简单说下结构组合拳

以 **SVM** 和 **K-Cluster** 为例,SVM 可以算比较代表性的 nonlinear 算法,KC 则比较偏向 linear,这两者在计算流程上基本可以算一样:都是先输入,再做输入预处理,得到算子数据,再算出最终的分类结果.**SVM** 推导思路走维度切换,维度部分是整个 **SVM** 的核心思路,集中在遍历单维度数据方程上,通过遍历得到最优平面可切分的单维点,再把几个单维组合起来就是超平面切分点,然后在超平面从最大到最小按跨度切开,再用维度算聚类,思路流程上是走的编码解码路线,这一步可以有一堆优化措施,如果是不走优化的 svm 会是很简单的流程思路,而 svm 可以写出几千行,也可以几十行解决,因为有维度跨度,这些跨度可以充当一种记忆条件,也就是训练建模,使用模型这套流程,**svm** 也可以做的非常简单,**pas** 系也可用几十行来做出 **svm** 自动分类,前提是必须有结构体支持,以前我参考的经典做法主要来自 shogun(c++ 古典派 svm 的起源项目),大如感兴趣可以自己找来研究.**K-Mean** 推导则是随机数生成质心分类,临近的全部聚类,完成后再定义出新质心重新走流程,反复迭代几次就完成聚类了.在结构层面,只要定义出,输入输出,剩下来的基本可以直接交给开源的各种计算库去干.大家平时在网上查找资料,各种复杂公式这是一种思路上的表达语言,算法流程思路尤其主体部分都不太复杂.比较难理解的是非线领域,这一领域如果把自创的东西算进来,可以有上百种算法,而非线的核心思路上是给数据做解码编码预处理.

以高速范围搜索为例,这一领域还没有来得及编写 demo,它的目标是解决范围内的快速搜索,例如,数据量到 10 亿,并且数据随时在增删,要搜索时间范围和坐标范围,如果不给算法暴力遍历,也许一个流程会走好几分钟.比较有效做法是把时间和坐标范围按度量切开,例如磁盘阵列的文件坐标,可以按每 1M 切出一个区域用于 hash,处理范围时以 1M 作为一个小小跨度来记忆,时间切分同理,可以按分切,也可以按时切,切分以后只需要录入一次就可以精确定位了.ZNet 的做法是缓存指针,hash 跨度, 参照库为 Z.HashMinutes.Templat,实现和组合时主要使用 **TBig\_Hash\_Pair\_Pool<>+TBigList<>**.其中时间范围加速算法,主要用于搜索监控片段,基本上全都可以秒搜,坐标范围加速算法,主要解决仿真写缓存,这是模拟写入文件并且保证读写一致性的功能,需要在高速读写环境,例如写入 100 长度坐标在 1024 位置,这时候需要找到缓存 1024 位置一系列的 part buffer,这样才能完成文件读写数据一致性.

以双向配算法对为例: **bidirectional**,每完成一个配对,会遍历全部目标,如果不考虑优化,配对 1000 比 1000,计算量为千万,当每完成一次配对,剔除掉已配对的数据,计算量将会小很多,再配合线程,并行这些手段,双向配对可以做到非常快.而解决删除配对,用 **TList** 搞不定的,**TList** 它会重构 array buffer,非常耗 cpu,必须使用 **TBigList<>**.

以并行排序为例:我无法知道地球上最快的并行算法,我的方法是先分开存再排,例如 1-10, 11-20 各自分成独立块,然后再用并行排粒度块,最后排整块+构建输出.结构体就是使用 **TBigList<>**,只有 **TBigList** 才能支持 10 亿这类大数量,直接用内存指针会把排序搞非常复杂.

# ZDB2 如何解决 Stream 写保护状态下的仿真读写

用过 VMWare 的都接触过磁盘写保护:在写磁盘时 VMWare 会将数据写入到一个临时文件,而原始数据并不会更改,同时在读操作时写入数据会一致化(原始数据与最近写入数据相分离).底层仿真库 Z.FragmentBuffer.pas 从正面解决了上述问题,下面来详细说说仿真写入的解决方案.

首先不管数据规模多大,磁盘是单维空间,数据都按坐标放在里面,并且写入的数据永远都是一小段,在仿真写入算法中把这一小段数据定义成 Part.在 Stream 中的日常写入方法大都为,从 stream.pos=xx 到 stream.write(xx)的循环行为.stream.write 的长度等同于 Part 的长度.当读取时,会先读入原始数据,再用 part 里面的数据做覆盖.这样就实现了仿真读写的一致化.我在解决这一流程中专门开辟了一个 Test case 保证一致化处理流程是正确的.这类 Test Case 工作还包括了对同一 Part 的反复 stream.write(我将它定义成 Part update 机制,例如 Stream 从 100 位置有时写入 1k,有时候又写入 100k,Part Update 机制完全随机不确定),以及多个 Part 可能发生合并情况的处理机制,处理这些繁琐的底层机制需要做到一步到位,不留尾巴.

当解决一致化以后接下来是优化计算,因为当 Part 数据达到一定规模,例如 100 万个,读写时会从 Part 数据里面找出对应的数据做读写,如何快速找到 Part 是一个非常重要功能.我考虑过使用 B 树方法,但 B 只对静态数据友好,后来,我分析了一下寻找 Part 可能运行的代码量(大致的 cpu 计算开销),得到了一个相对比较准确的结果:当 Part 数据达到 100 万个,并且精确定位到目标的平均时间开销大约 10-50ms,现在,已经可以定位到那个寻找 Part 的循环程序瓶颈.

这时候,使用跨度 hash 表来减少寻找 Part 的计算开销,跨度 hash 大致思路是按 1024Kb 做为一个跨度,例如当 stream.pos=1536kb 并且写入 1024kb,那么就是位于跨度 1024kb-2048kb 区间,这时候循环瓶颈就从 100 万次减少了 2-10 次左右,优化提升大约在 10 万倍,cpu 开销也从 50ms 下降到了 0ms.现在仿真写入已经具备了实用性.

接下来就是把 Stream 的仿真 IO 接入到 ZDB2,因为 ZDB2 一旦接入就可以做到数据安全了,因为每次写入都会在内存中仿真,并不会直接写磁盘 IO,这可以不用担心断电数据丢失损坏.工作机制为,替代原 IO 的全部读写,当 flush 时,先建立一个临时文件,把 part 先写入临时文件,然后 api 用 FlushFileBuffers 确保物理写入,然后再写入 ZDB2,当成功以后,删除临时文件,如果写入过程中断电,ZDB2 会从临时文件恢复.

在实际应用过程中,我发现 hdd 走阵列路线非常慢,数据量一旦>10GB 到 flush 会耗时 3-5 分钟,而 ZDB2 单库的日常空间使用经常也会从 500G-2TB,这里又是另一层程序主结构上的优化了,未来建议使用仿真写入的硬件设备最好走全闪的配置路线.虽然全闪空间不大,这在存储效率上会大幅提升,剩下很多很多优化工作.

另一方面,Z.FragmentBuffer.pas 库的命名与它的功能定位是匹配的,本身它并不会傻瓜化的接管 Stream,而是作为单维 Buffer 的处理计算,所以使用 Fragment+buffer 来命名.

By.qq600585

# 回顾:设计泛结构 **TBigList<>**

TBigList 并不是一蹴而就设计到位.这要回顾一下历史内容

1. 2015 年 TBigList 的最初前身是位于 Z.ListEngine 库的 THashList,这是作者编写的第一个 Hash 库,内部大量使用 TList 做转换保存功能使用
2. 2016-2017 左右,THashList 出现了序列还原需求,已无法追忆还原什么序列,总之,就是 add 123 以后,就应该可以直接从 THashList 还原 123 顺序,这种需求,导致了 THashList 内部结构从单一化到链条化的转变.
3. 2020 年 TOrderStruct 出现了,当时决定将全系代码从古典结构逐步移植成泛结构.
4. 2020 年 TBigList 被编写出来,TBigList 纲出炉时,由于 test case 写的很到位,顺带也修复了 THashList 顽固百年漏洞,这是非常大重大的修复.之后, TPair<>, TBig\_Hash\_Pair\_Pool<>, 被相续编写出来
5. 2021 年 ZNet 全部代码一律删除 TList,使用 TBigList<>泛结构替代.
6. 2021 年某天无聊,编写了一个 BigList pk TList 的小 demo,TList 全部使用最优删除,追加,修改的方法与 BigList 进行了一次性能 pk,结果是,BigList 处理能力几乎为 TList 的 2000 倍

TBigList 的设计思路:首先要解决高速队列+Int64 级数据量+可以在大规模循环代码种的堆砌编程:必须解决 for 这种循环需求.其中,解决 for 循环需求甚至高于对性能的要求,简单来说就是 for 必须是一种很简单流程模型,不可以用匿名函数.

最终,TBigList 才被设计成了今天的通用泛结构,在此基础上,后面,ZDB2,各种新算法,新结构体系,应运而生,这些结构和算法太多太多了,Z-AI 体系直接都不用提及了.

ZDB2 被编写出来,其实就已宣告自主技术进入大数据时代了.后面是用时间来迭代.

# 回顾:设计脚本引擎 **ZExpression**

很早很早以前,编译器一直是作者心里遗憾,当初的想法:理论懂一堆,如果不能动手写一次,理论就会是空谈,这件事必须有切身体会才行,吃喝玩乐过日子毫无意义(今天的感受是没钱的日子才是毫无意义).

- 字符串解析是面对的第一个问题,字符串解析程序非常复杂,最开始是做分子+分词函数,然后,开始尝试做符号解析,逆波兰,加减乘除,把各种字符转换成原型结构.最终,解决了词法转换.
- 第二步,开始尝试解决词法语义结构,诸如,词法合法性,这里的词法结构是树型的.因为  $1+(1-2)$  的词法结构等同于  $1+a$ ,而  $a$  结构是 $(1-2)$ .
- 第三步,开始尝试把词法结构翻译成可执行程序的 opcode 码,这一机制,用的是仿真 opcode 码来实现,机制上与 x86 译码工作方式是相同的,差异是对码表不同.
- 第四步,开始大幅度加强 Parsing 支持环节,无限逼近 bison+yacc,技术细节这里省略一下,放在下一节来说.
- 第五步,开始应用 ZExpression 体系:C4 启动,ZAI 脚本,6 代脚本,pascal 代码重写模型,这些都包含在 ZExpression 体系.

# ZNet 的母体移植技术:Z.Parsing

ZNet 是作为母体+载体的巨型代码项目,这些代码其实都是用机器编译技术解析+重构生成而来,大家平时使用 prp 移植 ZS,或则 prp 升级 ZNet,里面是一个数据模型+解析重构技术在工作.

编译器中的解析+重构技术,它的思路借鉴了 bison+flex/lex+yacc 体系.

简单举个粒子

```
if(1+1=2) kill;
```

```
if 1+1 = 2 do kill
```

上面是两种完全不同的词法体,在 bison/yacc 体系是用代号表达式,来描述这些词法体,也就是脚本语言,然后再用编码流程让他们形成统一化能表达流程的结构数据.

在 Z.Parsing 体系中,有一种机制可以无招胜有招,探头技术

探头技术是建立在词性的前置工作上,Parsing 体系会先将,数字,符号,浮点,字符串,备注,ASCII,等等,他们将会被词性划分,形成词性链结构.

例如 if(1+1=2) kill,它的词性链为:ascii,symbol,num,symbol,num,symbol....

探头技术,是对词性链做近似判断,然后进入分支流程.

探头技术可以区分,不同的词性结构组合+词法体,它会形成条件范式,这些条件范式,正是接监的 bison/yacc 的设计思路.

当词法程序具备了条件检测以后,就可以用蚂蚁一边爬一边用探头条件范式开分支程序处理随机性极大的手写词法体了.

也正是有了强大的探头机制,pascal 重写模型技术才能正确且完整的解析 pas 代码并且重构的自己想要的目标代码了.

在另一个方向,国际化的备注和字符串机器翻译技术,也是使用的 Z.Parsing 体系,而字符串翻译非常简单暴力,就是找词性为字符串和备注的数据结构翻译,然后重构成不同的语言.国际化项目大家有空可以去看

看 <https://github.com/PassByYou888/zTranslate>

Z.Parsing 的探头是灵活自由的,指哪就能打哪,在蚂蚁程序中可以做出 bison/yacc 无法实现的东西.这时  
候,Z.Parsing 是统治现金流的技术体系:只要使用 Z.Parsing 那个人能正确编写出自己的语言,并且他能让集体一致性的  
使用他的预言,那么他对软件,游戏,各种程序,将成为教父一般的人物,这将会让他拥有高于任何投资人,总经理,董事  
长的绝对技术话语权.如果公司估值 1000 万,使用 Z.Parsing 做出公司生产系统那个人,他身价占据一半我也一点不会  
惊讶.因为公司依赖产品,产品依赖集体的生产,生产核心技术依赖 Z.Parsing 用户.

在另一方面,Z.Parsing 体系,无论 HTML,JS,Pas,C++,XML,都能搞定.

# OpCode 中的非线性流程支持技术: TOpCode\_NonLinear

解释一下什么是非线性流程:以线性流程 `a(b(c()))` 为例,调用次序为 `c() -> b(n) -> a(n)`,当 `c` 函数不再需要立即返回同时堆栈机制不复存在,并且程序仍然以 `c() -> b(n) -> a(n)` 调用次序来执行,这种就是非线性流程.

从另外一个角度来解释非线性流程:在安卓系统中,非线性流程通常以 `lambda` 函数这类机制通过异步事件方式来走流程,从而完成 `c() -> b(n) -> a(n)`.

以录入系统举例:它的业务流程为,打开录入界面->录入->提交录入结果给服务器->等服务器反馈->完成录入,这一套流程在实现时就是非线性流程,而在 `TOpCode_NonLinear` 支持下,该业务流程会以如下方式书写

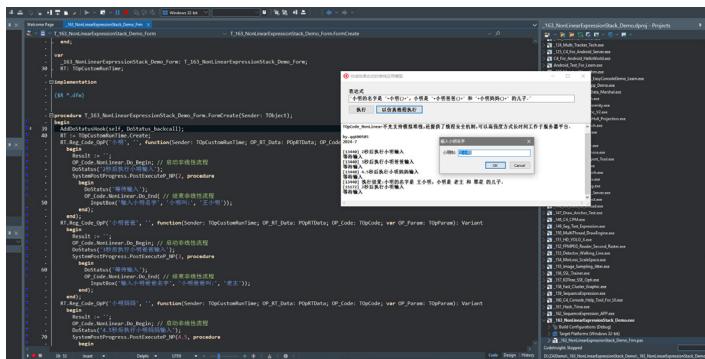
`"关闭录入窗口(提交到服务器("http://127.0.0.1/"),打开录入窗口并等待完成录入())"`

上面这段是代码,并不是业务语言,它可以被非线性流程直接翻译并且执行.我们需要编写 3 个函数

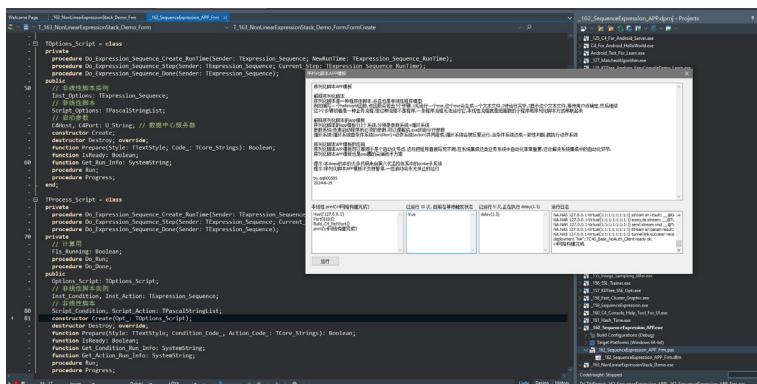
- 打开录入窗口并等待完成录入():(返回 json):这里 `show form` 出来输入,按下确定按钮,给输入数据做个编码,例如编码成 `Json`,然后 `End_Result(my json)`,给非线性流程一个结束信号
- 提交到服务器(url,json)(返回 bool):把 `my json` 发给 `http://127.0.0.1`,等待 `http` 响应,然后给非线性流程一个结束信号
- 关闭录入窗口():关闭录入窗口

当编写完成这三个函数后,一条表达式可以直接走完录入业务流程.

解释一下非线性流程的核心技术:这里先说明一个堆栈的技术问题,非线性流程是以软件仿真物理堆栈,但这种工作简单之极,毫无困难.核心技术是对线程的控制,以及非线性函数接口方式和使用非线性技术,这有许多设计上的考虑.核心细节可以参考编号为 163 的 demo



简单说一下非线性技术体系的衍生,`TOpCode_NonLinear` 是非线性技术在 `zExpression` 体系下的原子级支持,编码为 162 的 demo 则是基于 `TOpCode_NonLinear` 给出了非线性的序列化程序块的应用支持(`TExpression_Sequence`),该应用模型比较复杂,这是从 6 代监控的 `subscribe` 服务器模块剥离而出的 app 框架



一句化总结非线性流程:堆栈机制属于计算机科学领域,非线性是用表达式从正面解决了庞大流程调度问题,这是以超越堆栈视线的方式来工作,在未来非线性流程会应用于 z 系的各种项目中,因为 z 系总是遇到一个问题就去设计一个脚本语言,而非线性流程技术给出大规模应用的地基.

# 聊聊 TOpCode\_NonLinear 的线程支持设计

在设计层面,上面的红框是线程 1,负责主循环并且运行 OpCode,下面的红框则是线程 2,负责更新线程 1 的状态机,因为这些状态机都使用原子变量,这是线程安全的.

```
// TOpCode_NonLinear calling mechanism: no longer call in the stack mode,
// the new mechanism is to pave the way for the calling sequence structure from deep to shallow, and then execute the sequence once again
// The TOpCode_NonLinear call mechanism can support non-linear processes and accurately identify locations such as exceptions and trackers within the stack
348 TOpCode_NonLinear = class(TCore_Object_Intermediate)
private
  FAuto_Free_OpCode: Boolean;
  FRoot_OpCode: TOpCode;
  FOpCode_RunTime: TOpCustomRunTime;
  FOwner_Pool_Ptr: TOpCode_NonLinear_Pool_.PQueueStruct;
  FStack : TOpCode_NonLinear_Stack;
  FFirst_Execute_Done: Boolean;
  FIs_Running: Boolean;
  FIs_Wait_End: Boolean;
  FEnd_Result: Variant;
  FOn_Done_C: TOn_OpCode_NonLinear_Done_C;
  FOn_Done_M: TOn_OpCode_NonLinear_Done_M;
  FOn_Done_P: TOn_OpCode_NonLinear_Done_P;
  FOn_Step_C: TOn_OpCode_NonLinear_Step_C;
  FOn_Step_M: TOn_OpCode_NonLinear_Step_M;
  FOn_Step_P: TOn_OpCode_NonLinear_Step_P;
protected
  procedure Build_Stack();
  procedure Reset_OpCode_Non_linear();
  procedure Do_Init(); virtual;
public
  constructor Create_From_OpCode(Auto_Free_OpCode: Boolean; Root_OpCode : TOpCode; opRT : TOpCustomRunTime);
  constructor Create_From_Expression(TS_: TTextStyle; Expression_: SystemString; opRT : TOpCustomRunTime);
  destructor Destroy; override;
  procedure Reinit(); virtual;
  procedure Execute(); virtual;
  procedure Process(); virtual; // main-loop
  function Wait_End(): Variant;
360
361
370
  // The begin/end is a nonlinear process controller in OpCode event
  procedure Do_Begin();
  procedure Do_End(); overload;
  property End_Result: Variant read FEnd_Result write FEnd_Result;
  property Result : Variant read FEnd_Result write FEnd_Result;
  procedure Do_End(Result___: Variant); overload;
  procedure Do_Error();
  property First_Execute_Done: Boolean read FFirst_Execute_Done;
  property Is_Running: Boolean read FIs_Running;
  property Is_Wait_End: Boolean read FIs_Wait_End;
  property FAuto_Free_OpCode: Boolean read FAuto_Free_OpCode write FAuto_Free_OpCode;
  property FOpCode: TOpCode read FRoot_OpCode;
  property FOpRunTime: TOpCustomRunTime read FOpCode.RunTime;
  property FStack___: TOpCode_NonLinear_Stack read FStack__;
  property FOn_Done_C: TOn_OpCode_NonLinear_Done_M read FOn_Done_M write FOn_Done_C;
  property FOn_Done_M: TOn_OpCode_NonLinear_Done_M read FOn_Done_M write FOn_Done_M;
  property FOn_Done_P: TOn_OpCode_NonLinear_Done_P read FOn_Done_P write FOn_Done_P;
  property FOn_Step_M: TOn_OpCode_NonLinear_Step_M read FOn_Step_M write FOn_Step_M;
  property FOn_Step_C: TOn_OpCode_NonLinear_Step_C read FOn_Step_C write FOn_Step_C;
  property FOn_Step_P: TOn_OpCode_NonLinear_Step_P read FOn_Step_P write FOn_Step_P;
  class procedure Test();
end;
```

TOpCode\_NonLinear 设在线程时刻意回避了使用 TThreadPost 这类机制(位于 Core 库),这会增加框架上的设计复杂性,并且也会导致出现难以发现的 bug,例如当使用 do\_end,不小心 do\_end 两次,这会导致出现难以预料的结果.

TOpCode\_NonLinear 如果需要工作于主线程,有两条可选择路线

- 使用 System\_NonLinear\_Pool.Post\_Execute 方法,直接把执行代码扔给主线程即可,不需要自己管理主循环
- 在 Form 开个 timer 或则在主循环流程,以循环调用 OpCode\_NonLinear.progress();

2024-7-7 补充信息:TOpCode\_NonLinear 具有自动化的 OpCode 缓存机制,进去的代码会在编译后缓存,不会每次构建都跑去编译一次代码.TOpCode\_NonLinear 可以高效率在主线程工作,单线程运行能力大约在每秒(10+W/s).

## TOpCode\_NonLinear 的衍生

TOpCode\_NonLinear 需要特别说明一下,它是单函数性质的,函数可以写成 a(b(c())),但不可以写成 c(),b(),a().

- **TOpCode\_NonLinear\_Pool:**这是并发机制的 TOpCode\_NonLinear 容器,同时 TOpCode\_NonLinear\_Pool 也是一种主循环框架,在大多数情况下可以开一个线程,那个线程负责运行 TOpCode\_NonLinear\_Pool,然后在外面把需要执行的 Code 扔进去即可,好处就是不需要费脑自己在线程中实现 TOpCode\_NonLinear+主循环.
- **TExpression\_Sequence:**这是一个序列化流程支持库,它会将 TOpCode\_NonLinear 由上到下,再由左到右依次执行,等同于一个没有 if,for,变量机制的程序流程块.TExpression\_Sequence 在 TOpCode\_NonLinear 基础上构建而出,天生支持非线性流程.TExpression\_Sequence 是线程安全的,并且支持多线程并发.

# 从交换机到拓扑简单说说 XNAT

先挖最底层的 IP 包,IP 包构建在驱动程序的以外数据帧基础上(Ethernet),这一层的概念是建立在电频协调,协商速度,电缆规格这些硬件基础上,很多时候这很笼统,有人喜欢说以太层,也有人喜欢说物理链路(软件链路又是什么呢?),总之,不被概念绑架:IP 包是构建在物理通讯层上面的协议结构,而协议结构又分了很多很多种类.

IP 上面,会有 `Ipv4-tcp,Ipv6-tcp,Ipv4-UDP,Ipv6-UDP,icmp,arp`,组合起来跑拓扑.例如网卡通电,入网,这是个 `icmp+arp`,又例如 `tcp` 发一个包,送达出去,可能不会发生什么事情,但如果中间有 `wifi` 或则目的地是遥远的某个节点,可能会收到一个 `icmp`,这表示 `tcp` 报文不可送达,通常来说,我们在网卡驱动里面看到很多很多应用协议,大都走的就是 `icmp`.

理解 ip 层,只需要从 p2p 这种点对点通讯入手就可以了,ip 是在点对点基础上,扩展而出的拓扑网络协议,ip 数据永远都在网络里面转来转去,各种协议和硬件,路由,交换机,NAT,防火墙,就是控制这些 ip 包的流转.

每一个 TCP 包,正常情况下,只要未到达第一个路由节点,就是网关节点,那么就会有个 ICMP 发回来,告诉你这个 TCP 包不可到达,你需要重发.UDP 包则没有这种机制.我经常从国内向国外服务器上传大数据,单工 TCP 的速度,基本会达到限速级,这是并不是因为线路很快,而是第一个路由网关节点在主要消化我发送的 TCP.

回到 XNAT,ZNet 自带的 XNAT 底层走的是 `CompleteBuffer`,这是一种程序的模型,它是整个 Znet 最接近 TCP 原生数据包的协议,主要用于快速发送数据.至于速度会有多块,这取决于链路和网关,而链路和网关是决定了 TCP 的关键,Znet 走的是异步通讯模型,在异步模型所有的数据都是队列化处理,卡队列这种情况,包在 IOCP/EPOLLO 程序里面很难感觉+发现,但对于链路和网关,只要收到了 `icmp` 报文:这个 `tcp` 不可到达,那么 IOCP 的 `send` 队列流程机制就会暂停,这在并发程序模型里面是看不出来的.

`CompleteBuffer` 会出现万兆对传千兆,或则对传百兆:Znet 非常重视缓存管理,万兆传千兆并没有自动化的处理机制,而是默认全部让数据排队,堆在内存里面让硬件慢慢消化.在通讯程序层面,Znet 是要求:如果连续发送 `completebuffer`,当达到某个度,就发一条 `Send_Null` 命令,这是因为 Znet 在底层对大量 `completebuffer` 的数据做了自动化暂存功能(用硬盘保存),但如果不行 `send_null`,那么 `completebuffer` 是无法暂存的,而会全部一并扔给通讯接口,例如 `crosssocket,diocp,ics,synapse`,这些接口是把数据无脑堆内存里面,当服务器万兆传 100G 给千兆会按时间流逝发生崩溃.这也是很多人使用 XNAT 传 FTP 大文件,内存爆炸的原因.但是在通讯程序里面,加了 `send_null`,那么队列中的所有 `completebuffer` 就会暂存.因为 `send_null` 会让队列等待一个反馈,然后才会继续后面的队列,在这过程中,`completebuffer` 的数据保存在硬盘中待发.如果硬盘够大,万兆传千兆,100G 是可以被成功消化掉的.

说说 FRP,这是阻塞协议,阻塞做对穿,对等到达,万兆传千兆时,万兆会降成千兆来传,同时也不消耗内存和磁盘缓存空间.但异步不同,异步是全堆队列里面处理!!

在拓扑网络中,往往一个应用出口就是一台机器,但这台机器连接到网络数百台服务器,当 app 访问服务器时,出口往往只是流量代理,真正工作的却是网内的数百服务器.这个就是服务器群集.在现代化基建中,群集要发挥作用,NAT,正反代理,端口映射等等,都是经常会用到的技术.而近期的 Znet 新版本针对群集问题,给出了 XNAT 的大规模代理方案:100 台服务器没关系,用 100 个端口映射.

这一小节,简单说了一下我在 IDC 的应用思路,如有不周,大家请多多包涵.

# 性能分析:当 ZDB2 的数据规模达到 1 亿条

ZDB2 具备(Rolling)滚动存储能力,当数据规模达到临界点将会删头加尾,而删头加尾可以外部程序调度也可以内部自定义调度.

当数据规模达到 1 亿条,磁盘占用空间大约 100G-20T.这时候 rolling 有两种工作方式

1,在 TZDB2\_Th\_Engine\_Marshal 使用 Remove\_First\_Data\_For\_All\_Th\_Engine,这种方法主要提供外部调度程序删头(移除最早的数据,一般会在线程中调用它),这会有调用延迟,这些延迟 90%都来自于内存释放(destroy),而从磁盘移除数据几乎没有延迟(1:1000).达到 1 亿条后,每次删头规模,多则千万,少则也是上百万,这些延迟也许高达 5 分钟,在 5 分钟内,所有的数据被 busy 机制卡住暂存于内存,而在流程环节,ZDB2 具有防卡机制,append 数据会立即返回,只有当流量大到 5 分钟的数据量,系统内存装不下才会崩溃.

2,内置方式删头,内置删头是在每次追加数据 append 实时进行的,这种方式并不是流式删头(不等 destroy),而是开了一个子线程让它在侦测是否达到删头条件来删头.这样干的目的是减少 append 延迟,因为主线程的延迟会传导给别的程序模块,例如服务器既包含 ZDB2 体系也包含了 web 服务,端口代理服务,高流量延迟会导致某些服务严重受阻,好比端口代理正在实时直播,延迟将会导致直播卡顿.

当数据量达到 1 亿条的 ZDB2 崩溃模型

ZDB2 走的读写流水线,所有的读写数据永远都在排队处理, Remove\_First\_Data\_For\_All\_Th\_Engine,会在队列插入上百万条删除请求,当系统无法实时消化,那么写数据队列消耗的内存将会无限增加,直至系统崩溃.

当 hdd 阵列忙,读写数据将会出现延迟,队列无法被消化时内存暴增,直至系统崩溃.这种操作多出现于省钱买小内存,hdd 阵列规模过小,以及,在数据库工作中,阵列出现三方读写操作,例如大规模 copy 文件.比较适合 ZDB2 体系的阵列是至少 6-10 盘的 hdd,并且内存>预计空间的 2.5%,例如计划存储 10TB 数据,内存不低于 250G.另外,sdd 和集成化的 m2,nvme 对内存的需求要宽裕很多,内存只需要达到 1%,计划存储 10TB 数据,内存达到 100G 就可以正常工作.

当数据量达到 1 亿条,zdb2-flush 如何工作

Flush 操作是将内存中的暂存数据直接写入磁盘,ZDB2 使用分块算法写盘,当数据规模 1 亿条,单库文件的分块写盘规模将会高达 500M-2000M 不等, TZDB2\_Th\_Engine\_Marshal 是多库集成化的体系,往往一个 flush,会引发 100 个单库的写盘操作, 单次 flush 的 IO 写入规模甚至会达到 200G.这时候,hdd 阵列会达到瓶颈并且进入高延迟模式,而高延迟将会引发数据暂存模型无限增加内存.

ZDB2 在 idc 经历了数次宕机后,flush 算法被修改成了差分化分块写盘模型,差分写盘会先对数据块按坐标序列排序(sort position),如果发现分块过大(高于 100M),将会拆分成 100 个 1M 的小块,这时候,数据对比方法,检查数据是否修改,然后,再向 IO 进行写盘操作.最终的 flush 消耗从单次 200G 下降到 20G,这可以在半分钟内完成一次 flush 操作.

本节中的 Flush 包含了 IO 防止断电丢数据的读写分离机制.

当数据量达到 1 亿条所有的数据库模型都将变得不太实用,大部分系统方案会选择开辟分布式,用拆分数据库这些手段来解决大数据问题,ZDB2 也是走的拆分数据方法,只是 ZDB2 使用 TZDB2\_Th\_Engine\_Marshal 直接用集成化方法将子数据自动化管理起来了.1 亿条规模下的真实项目,不光是阵列,还会涉及到拆分网络服务,例如把存储服务和应用服务器分离开来.而作者是集成路线:web,zdb2,rtsp,hls,monitor 全部集成在一个服务里面来搞,思路则是开分载线程.

By.qq600585

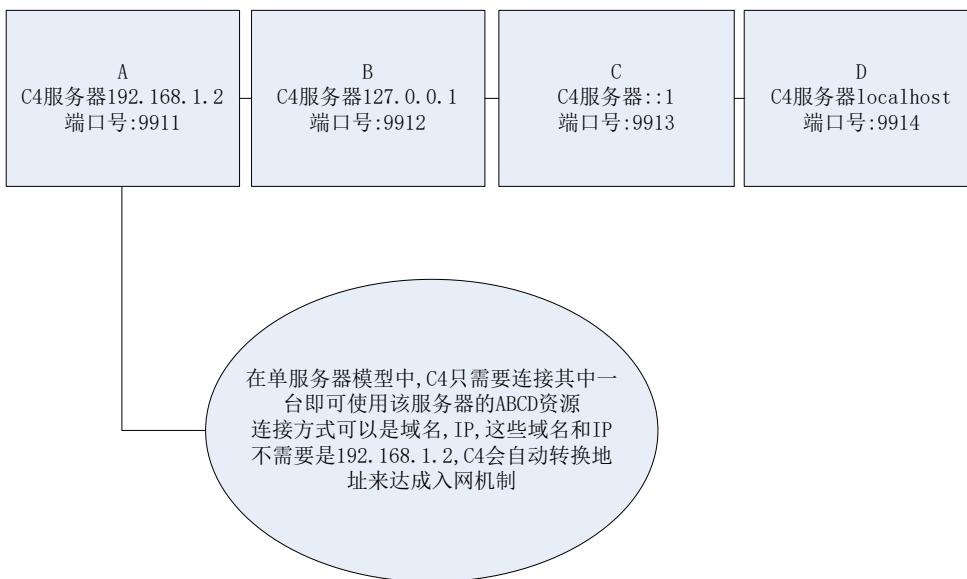
# 再说 C4 入网机制

C4 将服务器统一化,一个完整项目可以有若干机柜(每个机柜 5-8 台),将这些服务器协调统一就是 C4 的工作.

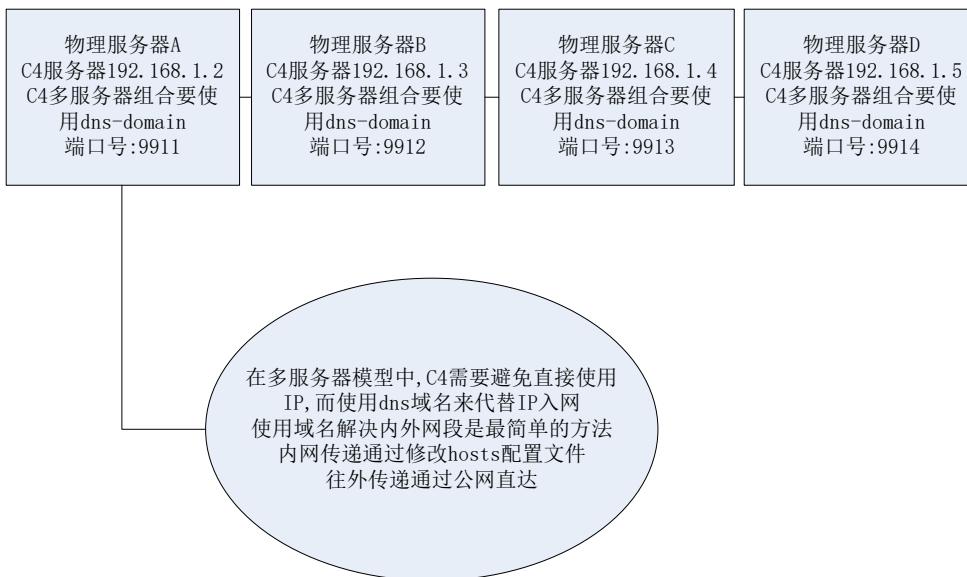
另一方面,最常用的服务器模型是一台高性能服务器,在单台服务器模型开一堆服务进程,这些服务可以是文件服务,数据服务,登录服务,内网穿透服务,在真实的项目中,这类服务会开很多,这是集成式的.导致服务器群会有很多很多 IP 地址,域名,端口.而使用 IP,DNS 就是一种 C4 的入网机制:C4 入网的本质是建立连接.

C4 地址转换机制:当 C4 连接目标网络,例如 123.net,而目标网络的上报地址 192.168.1.123,这时候,192.168.1.123 地址将被转换成 123.net,该方式可以满足常规单台 CS 组网模型:堆服务端并编写前端接口,然后,借助 C4 使用海量的服务器计算资源.

在单服务器中的 C4 入网机制:在单台服务器地址会自动转换,当使用 127.0.0.1+::1+192.168.1.2 这类不同的地址来表达入口时 C4 都可以自动转换出来并且入网.注意:单服务器如果插 10 张网卡+10ip 这时候要用 127.0.0.1 环回地址来代替物理地址 192.168.1.x->127.0.0.1 这样来干 C4 才能自动完成地址转换.



服务器群 C4 入网机制:使用域名来代替 IP,这样来解决内外网段各走自己的线路!这是最简单 C4 并网.



# 当 C4 入网后需要干什么事情

简单来说:C4 入网后在当前进程中就会多出许多可以直接使用的服务器资源.

C4 规则:任何 C4 服务器都是 Service+Client 的结合体,例如在 C4 中编写一个 MyService,那么就必须编写所对应的 **服务+客户端**,这是 C4 规格,而当部署完 C4 后,MyService 就是一项计算资源.

当系统集成时 C4 的成功入网代表整个服务器群进入工作状态:在系统集成中,各项服务器资源会非常多,例如 2 个机柜,每个机会 5 台服务器,每台服务器开一大堆服务,例如 web,文件,hyperv,kvm,视频,图片等等,这些都是把后台做大的一种堆砌模式(无限堆砌服务器),C4 对于堆大是一种自动化的模型,因为靠人工堆砌服务器群这是不现实的.因为系统集成是一项大工程,我们可以设顶它的复杂度为 n5,而人类的极限梳理能力为 n3,当解决方案的适应复杂度无法与真实 solve 复杂度相匹配时,会出现难以修改,维护,升级,更新的情况,当 C4 入网后,系统集成的 n5 难度会下降到 n3 以下,也许我这样说很多人不理解,当物理服务器>5 台的项目,C4 规则会收获先苦后甜的结果.

## ZNet 内置 CPM 可以代替 FRP+Nginx

很多 CS 项目使用 FRP+Nginx 来统一化穿透多台服务器,在 ZNet 中只需要在 C4 引进 CPM 就可以解决穿透问题.

**CPM = Cluster Port Mapping**,群集端口映射技术

CPM 在 C4 体系可以 10 行内解决穿透问题,并且这是可以编程的大规模穿透机制,例如多机房部署,多服务器部署,穿透往往是 ip+端口数据配置形式,但在 CPM 系统,穿透可以是编程形式,简单来说,可编程形式的穿透可以具有自动化穿网能力,而且 CPM-imp 代码非常小,很容易修改定义.

举个粒子,项目含有,vnc,remote desktop,sql,my server,vpn,smb 很多很多协议,并且这些协议和服务分散在群集中的各个服务器,这时候古典的做法是配置一台主服务器,再配置群集各个子服务器.对 CPM 来说整个群集只需要部署 my server,不需要管配置表,通常 CPM 可以支持到上千个端口.如果需要,甚至可以把终端直接 CPM 到公网.另一方面,CPM 也是极简化的可编程映射,只要能 ZNet 的环境都可以 CPM.

## C4 主要用于大项目吗?

使用 C4 主要走迭代形式路线,例如 C4 早期的服务器都是 ZDB2 的第一代体系,随着 AI 项目的推进,第一代体系会慢慢淡出,并且开始逐步使用 ZDB2 的第三代体系来重新复现,用户登录系统,文件系统,Key-Value 数据库系统.随着三代的推进,第一代体系并不会被 replace,而是保留在 C4 作为老项目支持.

服务器型的项目在 C4 中是走的迭代路线,每次有迭代就开个新服务,因为 C4 是个体系,往往一个服务会依赖一大堆小服务,这些大小服务往往版本都不会统一,例如 RandSeed 服务,这是一个用于在全 C4 网络生成唯一 ID 数字的服务,它可以被第一代 C4+ZDB2 体系使用,也可以被第三代使用.

**C4 并不是设计给大项目使用,而是走更符合服务器体系迭代的路线,服务器项目不会开发出来就一直按兵不动,而一直会保持升级迭代,很多时候 CS 形式通讯,一个迭代,也许整个 CS 都需要一起升级,这时候直接开个新的 C4 服务器吧,运营中换一下端口,拿一个月来过渡新老版本兼容.**

项目永远都在迭代,因为服务器不会一步到位,即使开发到位,在运营过程,也会有各种维护升级.

# 主循环内存微泄漏分析

微泄漏极难分析,例如 10 万行大流程如果发生大内存泄漏,可以根据运行状态来分析,而微泄漏往往出在手误,这时分析微泄漏,FastMM 这类技术并不好用,这是利用 `debug+tracer+report` 形式来进行分析,需要把大流程独立剥离出来慢慢分析,动则一周,多则数月.

微泄漏的典型例子是主循环程序,这种程序设计就是让服务器永不关机,在主循环中出现泄漏,主体内存占用 100G,微泄漏问题大约每 3 天增加 1-3G 左右,这时传统做法如上所述,流程规模一旦大了只能单独剥离出来以仿真方式运行找泄漏点.试想一下,载入 100G 的数据源,再去分析 1-3G 的微泄漏,干这件事情会非常反人类.

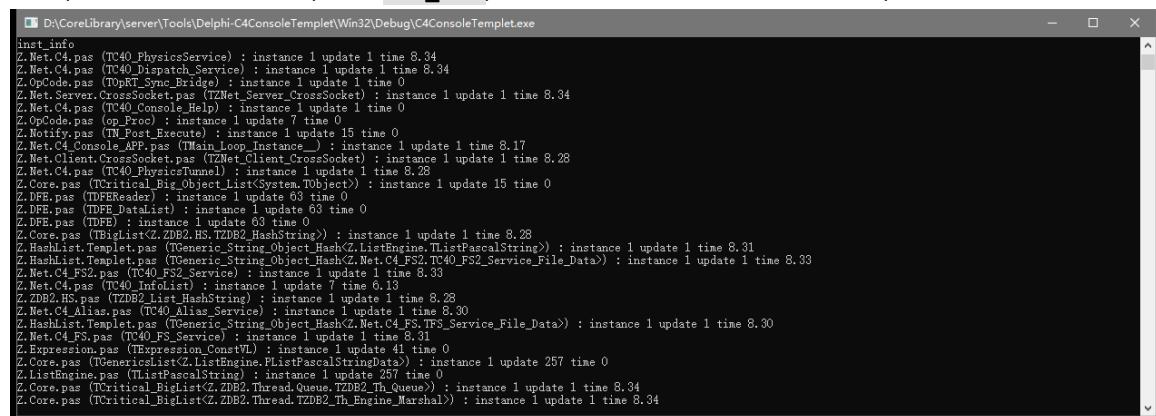
Z 系内核在服务器微泄漏给出了解决办法,首先打开编译定义“`Intermediate_Instance_Tool`”

```
// Intermediate_Instance_Tool is used to trace activity instances, which can affect performance after startup.  
// Intermediate_Instance_Tool help debug and analyze the state of large programs  
{$DEFINE Intermediate_Instance_Tool}
```

当 `Intermediate_Instance_Tool` 被打开以后,所有基于 `TCore_Object_Intermediate` 的实例都会进入计数模式,包括泛型结构,每次构建实例,计数器会+1,destroy 则-1.

使用时只需要将 `TMyObj=class(tobject)` 替换成 `TMyObj=class(TCore_Object_Intermediate)` 就行了,一般来说直接使用替换功能就行,Z 系内核提供了 `TObject,TinterfaceObject,TPersistent`,三种基类.

然后,开 C4 框架命令程序,输入 `Inst_Info`,得到当前服务器的全部实例计数,这些数据具备分析微泄漏条件.

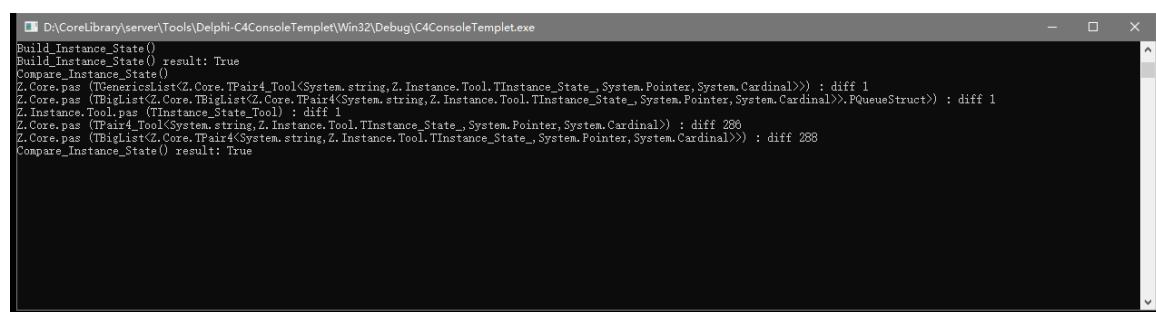


```
D:\CoreLibrary\server\Tools\Delphi-C4ConsoleTemplet\Win32\Debug\C4ConsoleTemplet.exe  
inst_info  
Z.Core.pas (TC40_Physics) : instance 1 update 1 time 8.34  
Z.Net.C4.pas (TC40_Dispatch_Service) : instance 1 update 1 time 8.34  
Z.OpCodes.pas (TopRT_Sync_Bridge) : instance 1 update 1 time 0  
Z.Net.Server.CrossSocket.pas (TNet_Server_CrossSocket) : instance 1 update 1 time 8.34  
Z.Net.C4.pas (TC40_Console_Help) : instance 1 update 1 time 0  
Z.OpCodes.pas (op_Proc) : instance 1 update 7 time 0  
Z.Notify.pas (TN_Post_Execute) : instance 1 update 15 time 0  
Z.Net.C4_Console_APP.pas (TMain_Loop_Instance_) : instance 1 update 1 time 8.28  
Z.Net.Client.CrossSocket.pas (TNet_Client_CrossSocket) : instance 1 update 1 time 8.28  
Z.Net.C4.pas (TC40_PhysicsTunnel) : instance 1 update 1 time 8.28  
Z.Core.pas (TC40_Big_object_List<System TObject>) : instance 1 update 15 time 0  
Z.DBE.pas (TDFEReader) : instance 1 update 63 time 0  
Z.DBE.pas (TDFE_Datalist) : instance 1 update 63 time 0  
Z.DBE.pas (TDFE) : instance 1 update 63 time 0  
Z.Core.pas (TBigList<Z.ZDB2_HS,TZDB2_HashString>) : instance 1 update 1 time 8.28  
Z.HashList.Templet.pas (TGeneric_String_Object_Hash<Z.ListEngine.TListPascalString>) : instance 1 update 1 time 8.31  
Z.HashList.Templet.pas (TGeneric_String_Object_Hash<Z.Net.C4.FS2.TC40_FS2_Service_File_Data>) : instance 1 update 1 time 8.33  
Z.Net.C4_FS2.pas (TC40_FS2_Service) : instance 1 update 1 time 8.33  
Z.Net.C4.pas (TC40_Infolist) : instance 1 update 7 time 0.13  
Z.ZDB2_HS.pas (TZDB2_List_HashString) : instance 1 update 1 time 8.28  
Z.Net.C4_Alias.pas (TC40_Alias_Service) : instance 1 update 1 time 8.30  
Z.HashList.Templet.pas (TGeneric_String_Object_Hash<Z.Net.C4.FS2.PCFSS_Service_File_Data>) : instance 1 update 1 time 8.30  
Z.Net.C4_FS2.pas (TC40_FS2_Service) : instance 1 update 63 time 8.31  
Z.Expression.pas (TExpression_ConstW4) : instance 1 update 41 time 0  
Z.Core.pas (TGenericsList<Z.ListEngine.TListPascalStringData>) : instance 1 update 257 time 0  
Z.ListEngine.pas (TListPascalString) : instance 1 update 257 time 0  
Z.Core.pas (TCritical_BigList<Z.ZDB2_Thread.Queue,TZDB2_Th_Queue>) : instance 1 update 1 time 8.34  
Z.Core.pas (TCritical_BigList<Z.ZDB2_Thread.TZDB2_Th_Engine_Marshal>) : instance 1 update 1 time 8.34
```

更直接的一种做法是,先做一个当前实例状态副本,让服务器运行一段时间,例如 2 小时,然后做一次比对,得到 2 小时内的新增实例状态,到这一步,几乎可以定位出问题出的具体位置.

创建实例状态副本:`Build_Instance_State`

以当前实例与副本做比对:`Compare_Instance_State`



```
D:\CoreLibrary\server\Tools\Delphi-C4ConsoleTemplet\Win32\Debug\C4ConsoleTemplet.exe  
Build_Instance_State()  
Build_Instance_State() result: True  
Compare_Instance_State()  
Z.Core.pas (TGenericsList<Z.Core.TPair4<System.String,Z.Instance.Tool.TInstance_State_,System.Pointer,System.Cardinal>>) : diff 1  
Z.Core.pas (TBigList<Z.Core.TBigList<Z.Core.TPair4<System.String,Z.Instance.Tool.TInstance_State_,System.Pointer,System.Cardinal>>,PQueueStruct>) : diff 1  
Z.Instance_Tool.pas (TInstance_State_Tool) : diff 1  
Z.Core.pas (TPair4<System.String,Z.Instance.Tool.TInstance_State_,System.Pointer,System.Cardinal>) : diff 286  
Z.Core.pas (TBigList<Z.Core.TPair4<System.String,Z.Instance.Tool.TInstance_State_,System.Pointer,System.Cardinal>>) : diff 286  
Compare_Instance_State() result: True
```

Diff 值为差值,286 表示在 2 小时中,某个泛型的增加了 286 个实例.

到这一步,我们已经可以定位出微泄漏的问题所在点了,剩下来就是检查和该泛型相关的流程进行修复

分析微泄漏不需要重启服务器,也不需要动不动开 100GB 的数据仿真,直接在服务器运行中就能做到,对于运营项目这是极方便的维护方法.

# 内核库在启动时都做了什么事情

Z.Core 库是整个 Z 系的中央内核,它和 fpc/delphi 无关,下图这些变量大多是某些小体系的全局变量+类的初始化

```
. initialization
. // float exception
| SetExceptionMask([exInvalidOp, exDenormalized, exZeroDivide, exOverflow, exUnderflow, exPrecision]);
500 // raise event
| On_Raise_Info := nil;
// instance trace
| Inc_Instance_Num := __Inc_Instance_Num__;
| Dec_Instance_Num := __Dec_Instance_Num__;
// critical recycle pool
| Init_System_Critical_Recycle_Pool();
// thread Technology
| Core_Main_Thread := TCore_Thread.CurrentThread;
Core_Main_Thread_ID := MainThreadID;
Used Soft_Synchronize := {$IFDEF Core_Thread_Soft_Synchronize}True{$ELSE Core_Thread_Soft_Synchronize}False{$ENDIF Core_Thread_Soft_Synchronize};
MainThread_Sync_Tool := TSoft_Synchronize_Tool.Create(Core_Main_Thread);
On_Check_Soft_Thread_Synchronize := Do_Check_Soft_Thread_Synchronize;
On_Check_System_Thread_Synchronize := Do_Check_System_Thread_Synchronize;
OnCheckThreadSynchronize := nil;
// cps
CPS_Check_Soft_Thread.Reset;
CPS_Check_System_Thread.Reset;
// parallel
519 WorkInParallelCore := TAtomBool.Create(True);
520 ParallelCore := WorkInParallelCore;
// MM hook
| GlobalMemoryHook := TAtomBool.Create(True);
// timetick
| Core_Runtime_Tick := C_Tick_Day * 3;
Core_Step_Tick := TCore_Thread.GetTickCount();
// global critical
| Init_Critical_System();
// random
| InitMT19937Rand();
CoreInitiatedTimeTick := GetTimeTick();
// thread pool
| InitCoreThreadPool(
|   if_(IsDebuging, 2, CpuCount * 2),
|   if_(IsDebuging, 2, {$IFDEF LimitMaxParallelThread}8{$ELSE LimitMaxParallelThread}CpuCount * 2{$ENDIF LimitMaxParallelThread}));
// thread progress
| MainThreadProgress := TThreadPost.Create(Core_Main_Thread_ID);
MainThreadProgress.OnStep := False;
| MainThreadPost := MainThreadProgress;
// thread Synchronize state
| Enabled_Check_Thread_Synchronize_System := True;
| Main_Thread_Synchronize_Running := False;
finalization
```

**SetExceptionMask([exInvalidOp, exDenormalized, exZeroDivide, exOverflow, exUnderflow, exPrecision]);**

设置浮点异常过滤器,异常是浮点计算程序的一种标准模型,例如某些 **cpu** 流片的浮点指令错误,浮点除 0,空浮点,浮点过大,精度丢失,在 x64 架构编译器会默认使用 64 位整数来模拟浮点,这时许多异常是来自 x64 模拟信号,而许多 x86 架构编译出来的浮点代码会使用 **fpu,mmx,sse** 这类独立浮点芯片(**fpu** 通常都集成在 **cpu** 内部),这时候浮点异常规则是当 **fpu** 计算失误,返回 **Nan** 这类无效状态时触发的,这些计算都会报出异常,并不是 **fpu** 过程异常,这里注意区分,**fpu** 永远是计算完成后,数值不对才会出现浮点异常,Z 系会把这些异常全部屏蔽.当 Z 系屏蔽后,如果发生浮点计算错误,例如 0 除和除 0,这时候一律返回 **Nan** 值,这代表一个无效的计算结果.如果浮点程序写得很大,例如高斯计算,差分计算,金子塔计算,在这类大计算流程中 **Nan** 会让人找不到问题所在,这时候,可以将过滤器直接屏蔽掉,或则人工打开.

**On\_Raise\_Info := nil;**

Z 系内核有个 **RaiseInfo** 的 api,作用是简化 **raise Exception.Create()** 的书写规则,**RaiseInfo** 会给堆栈程序抛出一个异常,而在抛出异常之前,会调用一次 **On\_Raise\_Info** 事件.**所有 Z 系异常都会通过 RaiseInfo 来抛出**.因此截获该事件就等同于截获了 Z 系的全部异常事件.在另一方面,**DisposeObject** 主要用于代替 **Obj.Free** 方法,当出现异常时,Z 系也会调用一次 **On\_Raise\_Info**,然后才会给堆栈抛异常.

**Inc\_Instance\_Num := \_\_Inc\_Instance\_Num\_\_;**

**Dec\_Instance\_Num := \_\_Dec\_Instance\_Num\_\_;**

实例跟踪技术的回调事件,这里的指向是个空函数.这是一个鸡和蛋的矛盾先生问题解决办法:到底是先有鸡还是现有单,因为 Z 系全部实例都可以产生被跟踪的数据,那么跟踪程序本身以及跟踪程序所依赖的更加底层的实例怎么办?难道比实例跟踪技术更加底层的实例都需要被跟踪吗,在设计跟踪技术前,确实是考虑到全体实例一律启动跟踪技术,在实际解决中,跟踪程序套跟踪程序,这是一个矛盾的功能,而解决办法就是交给上帝来处理(不确定场景就用事件挂接技术):内核在启动时,跟踪程序是空调用,当依赖于内核跟踪程序被启动以后,回调事件才被赋值,这时候跟踪程序才会发生效果,跟踪程序内部是一套复杂的数据系统,它会记忆全部实例的创建和销毁,主要解决大流程内存泄露问题.

### Init\_System\_Critical\_Recycle\_Pool();

这是线程临界区的回收池,临界区在 Z 系内核会区分软临界和硬临界,软临界使用无限循环等原子信号,硬临界在等原子信号中会触发 cpu 时间周期,硬临界在等信号中,从任务管理器看起来就是 `cpu=0`. 临界回收池是一种对临界系统句柄再利用,Z 系的所有软+硬临界句柄都是回收模型,当临界区被释放时句柄并不会消失,同时临界区句柄如果是 `Acquire` 状态那么会被 `Release` 后再回收,简单来说,Z 系服务器如果大量使用线程临界,那么在系统监视器里面几乎句柄开销会恒定,而任务管理器中会有一个 `cpu` 内核时间,这种内核时间是表示调用操作系统内核的延迟时间,内核延迟越大等同于操作系统对服务器的稳定性影响越大,Z 系服务器和大规模并行算法的内核延迟很小这都取决于大量应用回收池这类机制,参看 [锁复用](#).

```
// thread Technology
Core_Main_Thread := TCore_Thread.CurrentThread;
Core_Main_Thread_ID := MainThreadID;
Used_Soft_Synchronize := {$IFDEF Core_Thread_Soft_Synchronize}True{$ELSE Core_Thread_Soft_Synchronize}False{$ENDIF Core_Thread_Soft_Synchronize};
MainThread_Sync_Tool := TSoft_Synchronize_Tool.Create(Core_Main_Thread);
On_Check_Soft_Thread_Synchronize := Do_Check_Soft_Thread_Synchronize;
On_Check_System_Thread_Synchronize := Do_Check_System_Thread_Synchronize;
OnCheckThreadSynchronize := nil;
```

主次线程分离技术,可参考 [双主线程](#) 和 [同步技术](#) 章节,上述的代码都是初始化双主线程的全局变量. 双主线程就是 UI 一个主线程体系, 服务器走次主线程体系, 程序体系总是走的规范+规则, UI 的主线程体系就是走的 `vcl, lcl, fmx` 的主线路线, 而服务器的次主线程体系则是使用 Z 系主线程体系规则.

双主线最奇妙的地方: 当未开启双主线模型前, 两种主线模型可以互相兼容, 因为这时候主线程只有一个, 而当双主线模型被开启后就需要区分主线和次主线了, 具体细节我也编写了许多章节文档和 demo, 大家可以自己研究.

因为 Z 系自己实现了 `Synchronize` 机制, 其中有许多非常晦涩的硬件机制, 文档篇幅和人性化的阅读习惯不支持描述这些东西, 靠自己研究把.

```
// cps
CPS_Check_Soft_Thread.Reset;
CPS_Check_System_Thread.Reset;
```

CPS=caller of per second, 由于每个线程都会有自己的主循环, 例如 `vcl, lcl, fmx` 的主循环就是循环处理消息, 而次线程的主循环是一个事件调用接口, 在该接口中需要自己来编写主循环, 通常来说, 例如 `C4, ZNet, DrawEngine` 这类框架都会有自己的 `Process`, 这就是主循环. 这里的 CPS 是从主次线程检查它线程同步中的性能计数器, 例如有 10 个子线程, 他们在工作中会偶尔 `Synchronize` 到主线程执行一下, 但是每次 `Synchronize` 到主次线程时, 并不能马上执行而是放到一个队列, 直到主次线程 `CheckSynchronize` 才会被执行, 通常来说, 如果 application 处于 `run` 状态, 它会在活动消息循环中反复使用 `CheckSynchronize`, 另一方面 `CheckSynchronize` 不能记录每次同步的时间消耗, 同时也无法按秒单位的时间统计出调用频率. 对于前端, 例如手机, `win32`, 只要用起来不觉得卡顿那就是没问题, 但在服务器端, 会有性能泄露, 例如从数十种线程分支找到每次调用传递延迟达到 2 秒那个线程会需要数据, 而 CPS 就是记录同步执行延迟+同步调用频率的数据工具.

```
// parallel
WorkInParallelCore := TAtomBool.Create(True);
ParallelCore := WorkInParallelCore;
```

并行体系的全局动态开关, 这是一个 `Bool` 原子变量, 如果为 `True`, 并行程序会在多线程模型下运行, 否则就在当前线程下运行, 当 `False` 时 `ParallelFor()` 这类 api 会直接工作与当前线程, 并不会开新线程, `WorkInParallelCore` 变量可以在程序运行中, 或则启动时来动态的开关: 如果以前的 `ParallelFor()` 还在运行中, 这时候改变 `WorkInParallelCore` 变量不会影响以前的并行程序, 只会影响以后的并行机制. 提示: 并行开关不光可以全局变量, 也可以局部变量, 例如并行 A 用 4 个线程, 并行 B 只用 1 个线程, 并行 C 则直接工作于主线程, 另一方面通过修改编译器预定义(`Z.Define.inc`)也可以做到控制并行开关.

```
// MM hook
GlobalMemoryHook := TAtomBool.Create(True); |
```

内存 MM 库的 Hook 状态开关: 普通程序可以忽略. 在 ZDB1 设计中, 数据是 `free` 形式, 自己设计数据实例, 当用户自己的实例缓存达到某个限度, 然后, 启动缓存管理流程, 以此来实现高速数据遍历. 因为用户自己设计数据实例是一个非常随机的内存开销, 因此直接在底层 MM 环节做了一个定向线程的 hook 接口, 例如在 A 线程创建执行数据实例创建和读取 `TmyInst.create, load...`, 这时候, ZDB1 会记录 A 线程创建实例使用的内存开销, 这将为后面的缓存管理提供运行数据层面上依据. `GlobalMemoryHook` 是控制全局 MM 的流程开关, Z 系 MM-Hook 库总共有 4 层嵌套, ZDB1 是其中一层, 其它 3 层是在用户层程序使用, 只有在它打开时, MM 才会做定向线程记录. 一旦关闭所有 MM-Hook 都将失效, 同时 `alloc, realloc, free` 这三个 MM 操作将得到提速.

```
// timetick  
Core_RunTime_Tick := C_Tick_Day * 3;  
Core_Step_Tick := TCore_Thread.GetTickCount();
```

由于早期设备和系统的常规时间刻度都是 32 位整型,1 秒=1000ms,因此 32 位整型最多也只能记录 30-60 天的时间刻度.

Z 系所使用的时间刻度如果通用于各种硬件和系统,那么就不能直接使用 GetTickCount 这类 api,而 QueryPerformanceCounter 这类高精度刻度 api 是有平台和硬件相关性的,也是不能直接使用的,移植时会因为刻度单位,误差性,带来很多小问题!

后来,几经权衡,开了一个全局变量用于绕行 60 天的时间刻度限制,具体做法就是在启动时使用 GetTickCount 记录当前刻度,在获取刻度时通过符号计算达到绕行目的.主要是为 Z 系的 GetTimeTick 提供安全刻度,也许 GetTimeTick 精度误差会在 16ms 内,这对前后台已足够.另一方面 GetTickCount 取出的刻度是 32 位整型,Z 系的 GetTimeTick 是 64 位整型.

```
// global critical  
Init_Critical_System();
```

Init\_Critical\_System();是初始全程序的互斥区,此处有 3 个重要程序互斥区

1. GetTimeTick 使用的互斥区,每次调用 GetTimeTick 会锁一下,然后,更新全局变量,GetTimeTick 在线程中也是安全的,另一方面 GetTimeTick 流程大约 6 行整数型计算代码,翻译成机器码大约 15-20 行汇编,GetTimeTick 的锁是极快的,0.001% 的锁延迟几乎不存在多线程性能耗损.
2. AtomInc/AtomDec 原子数操作,在 delphi 平台原子操作 api 使用 AtomicIncrement, AtomicIncrement 在 x64/x86/ARM 这些平台是一种汇编命令并不是系统级别 api,而 fpc 平台是不提供原子 api 的操作,在 fpc 平台 Z 系就是用互斥区锁一下,然后再做 Atom 的模拟操作.而 fpc 平台的互斥区锁就是在 Init\_Critical\_System() 做初始化.如果使用 delphi 可以忽视这一环节,在 fpc 中使用 AtomInc 其性能是不如 Delphi 的.
3. 在 Delphi 平台中 TObject 对象内置了互斥区变量,使用方法为, TMonitor.Enter(Obj), 而 FPC 平台中 LCL-TObject 并不提供互斥锁变量,因此 Z 系提供一种 TObject 于互斥锁的配对结构,这种结构是基于 TBigList 构建的实例反查算法,作用是查找与 TObject 配对的互斥锁变量,然后模拟出 TMonitor.Enter(Obj), 当反查程序开始工作时需要锁一下,这种互斥锁就是在 Init\_Critical\_System() 初始化.模拟 TMonitor.Enter(Obj) 这种做法在 2017 年 Z 系早期的程序中会比较常见,而现在几乎已经放弃 TMonitor.Enter(Obj) 的做法,因为这种做法就是用一行代码简单的实现互斥锁,线程的程序并不在意多写两行,自己开个互斥锁变量来控制多线程机制,这样不光是程序更容易维护,同时也解决了 fpc 的兼容机制问题.

```
// random  
InitMT19937Rand();  
CoreInitiatedTimeTick := GetTimeTick();
```

CoreInitiatedTimeTick := GetTimeTick(), 这一行是内核的启动时间,给外部程序提供参照数据用的,这一行没有含义.

InitMT19937Rand(), 这一行是初始化 MT19937 随机数的数据结构,MT19937 是一种随机跨度非常均匀的随机数生成算法,同时 MT19937 也覆盖了 pas,c,c++,go,java,c#众多开发语言,但 Z 系 MT19937 并不是一个简单的随机库,Z 系 MT19937 是一种体系,Z 系解决多线程随机性问题,Z 系得 MT19937 在每个线程中都会有自己的独立 Seed 算子,可以做到在大规模在线程和并行程序引入统计学算法,因为统计方法大量使用随机数,虽然某一些统计学算法库会给出针对随机数实例的功能,这种需要构建一个随机算法实例,然后再把实例传递给统计学算法,这种做法远远不如线程内置独立随机数实例,例如,我们要移植一个 K-Mean-Cluster,几乎不需要改代码,直接 copy 过来,然后把系统内置的 Random 函数替换成 MT19937Rand32 就可以了.

在另一方面,Z 系的 MT19937 是配对的,每次调用随机数时,Z 系都会检测线程配对,这种配对机制与上一节的 TMonitor.Enter(Obj) 非常类似,但是 MT19937 只会配对线程和 MT19937 实例.除此之外,Z 系提供了 MT19937 的优化线程配对,如果线程模型走的是 TCompute,那么配对时候会绕过实例搜索,TCompute 内置了 MT19937 直接在现成使用即可,在 TCompute 线程模型下 MT19937 的生成效率是极限的.

Z 系的 MT19937 从底层机理角度解决统计学方法跑线程化的大难题.同时 MT19937 有许多变种形式的使用方法,例如自定义实例,自定义种子等等,具体细节 Demo 和文档会比较少,更多的需要大家自己去研究内核实现.

```
// thread pool
| InitCoreThreadPool(
|     if_(IsDebuging, 2, CpuCount * 2),
|     if_(IsDebuging, 2, {$IFDEF LimitMaxParallelThread}8{$ELSE LimitMaxParallelThread}CpuCount * 2{$ENDIF LimitMaxParallelThread}});
```

这是初始化 Z 系线程池,原型为:InitCoreThreadPool(最大线程数限制,最大并行线程粒度)

这一行的白话文解答为:如果调试模式,最大 2 线程

如果是 Release 模型:线程池最大线程=CpuCount\*2,并行程序粒度由外部定义决定要么 8 个要么 CpuCount\*2

InitCoreThreadPool 函数被调用时,会初始化一大堆变量和互斥锁,然后,它会创建一个 Dispatch 这类调度性质的独立线程,其作用是调度线程的回收,创建,启动.调度机制所使用的技术就是状态机和结构.

当使用 TCompute.Run 时,TCompute 会把参数以数据 Copy 形式,发送给 Dispatch 调度线程,调度线程这时会从线程池的数据结构去寻找空置线程,如果找到了,就把 TCompute 的参数发过去,然后启动函数,如果没有找到,这时候会检查和等待由 InitCoreThreadPool 定义的最大线程限制,如果被限制了最大线程,它会一直等待,如果不被限制,那就创建一个新线程,然后再把参数发过去执行.

值得说明的地方: LimitMaxComputeThread,这是一个预编译开关,它必须打开以后,调度线程才会真正做到对最大线程数的限制,如果是关闭状态,调度线程没有找到回收线程时就会创建一个新线程. LimitMaxComputeThread 默认是关闭状态,配置于 Z.Define.inc 预编译中.

TCompute 的每一个线程在执行结束时都会有一个存活周期,只有在存活周期中,TCompute 线程才可以被调度线程重复使用,默认存活周期时间为 1000ms,由 InitCoreThreadPool 函数内部赋予.说明:存活周期会影响 app 的关闭速度,如果是高频率+高速调用的 shell 程序,存活周期可以调整成 100ms,因为 app 关闭时会等全部线程结束,把存活周期调小即可视线高频率 shell 模型.

并行粒度与最大线程数的关系:并行模型一律依赖于 TCompute,当并行粒度小于最大线程限制时并行线程会畅通无阻执行.而当最大线程限制为 20,当前已经已有 18 个运行中线程,这时候并行粒度为 20,那么真正执行并行程序的线程只会有 2 个线程,它会反复执行,模型为:必须启动完 20 个线程,TCompute 会反复先执行 2 个并行线程,待这两个并行线程执行完,再执行 2 个,一直跑满 20 个线程.通常来说,最大线程数量都是不限制的,当预编译开关 LimitMaxComputeThread 是关闭时,并行程序只要指定 20,那么 20 个线程都会瞬间进入工作状态,并不会等待.

Z 系并行粒度为 8 的含义:并行程序的计算目的是加速 for 的处理能力,通常来说,并行提速达到 8 倍会是一个极限,更多的会是 2-4 倍左右的提速,具体要看并行程序的计算类型,例如浮点计算,浮点需要区分小浮点流程和大浮点流程,在小浮点流程中直接 for 会比并行更快,因为并行会有一个启动时间,在大浮点流程中,浮点计算会以上百行起步,这种才适合使用并行计算.在 x64 架构中,浮点计算通常是用整数模型,走 sse 的临时寄存器做符号计算,sse 通常内置于每一个 cpu 核里面,并不是全核共享 sse,sse 是指令集的一部分,而模拟化浮点 sse 的输出和输入指针都是目标内存地址,这时候浮点传递会有一个内存 copy 时间,这是在不同设备的数据传递,这种 copy 时间会限制并行的性能.一般情况下,浮点的 copy 很小,高频率内存完全可以胜任把浮点 copy 跑满,提速几十倍,但这样一干,cpu 就没有空间可以运行别的性能需求程序.在除却浮点计算之外,在多路 cpu 架构的服务器中,当 cpu 跑满,就连 pci 的传递管线都会被影响,反应出来会是往 gpu 里面传递数据出现明显延迟甚至卡顿,因为多路 copy 需要依赖于 cpu 去做数据的 copy 计算.而当并行程序遇上字符串,数据结构这类处理,copy 机制会大量使用,这时候即使满核运行,也会被内存带宽所限制.总结一下 8 的含义,给硬件留一点计算资源,并行数量并不是越多越快,并行程序只要能提速 4 倍以后就算 ok,最后是 8 的中文拼写代表发,发财.在以前,许多做大数据的人,给我说,排序,字符串转换,他们都用并行开来跑,其实把,围绕字符串这种计算领域,开满计算耗时会远大于 8 线程,被内存+北桥限制,也许应该让他们了解一下 Z 的并行计算思路.

另外一点:在 TCompute 线程中,TCompute.Run 总是开线程执行,使用上并没有多余的概念和思路,内部如何工作无需关心细节,因为本节是说内核启动,对于 TCompute 就不做深入说明了.

```
// thread progress
| MainThreadProgress := TThreadPost.Create(Core_Main_Thread_ID);
| MainThreadProgress.OneStep := False;
| MainThreadPost := MainThreadProgress;
```

这是一个主线程 Post 框架的全局初始化,线程 Post 从字面来理解就是往目标线程里面提交一个可执行调用,主线程 Post 框架就是往主线程提交可执行调用,因为线程都会有 synchronization 方法,但 synchronization 会等待调用退出,Post 则是发送后立即返回.

TThreadPost 实例内部有非常多的属性和模式,OneStep 是每一个 Progress 是否只处理一次执行,例如队列有 10 个可执行提交,而 OneStep 为 True,那么就需要 10 次 Progress,当 OneStep 为 False,那么每次 Progress 都会执行全队列.另外, TThreadPost 还具备 MT19937 随机化能力,每一次执行时都会重置一下 MT19937 的随机数种子.

# FS3 是如何被设计出来的

## FS3=File System 3.0

在设计层面 FS3 是个极大跨度,并且彻底解决了大文件数据库管理难题.

在早期基于 C4 框架设计的网盘使用 FS2.0 模型,这套模型是单线程数据库(ZDB2.0 的初级形态,单库最大容量 130TB,不支持阵列),其次是在上传和管理大文件时需要自己编程解决大文件切分和拼装(上传大文件以 1,2,3,4,5 这类碎片进行分块传输,管理需要存储分块索引,最后下载也需要以碎片进行拼接,最关键的地方是使用大文件需要自己编程).FS2.0 给出这样的使用规则是因为 FS2.0 的内部机制本身很复杂.

FS2/3 的身份定位是服务器之间的文件支持模型,例如 C 端往 VM 服务器发送一个文件,这时候服务器会通过 FS 系统来传递该文件,也就是服务器和服务器之间的文件支持协议,它是并行的.当 C 端与服务器建立一个文件传输,这时候 VM 服务器与 FS 服务器会开辟一条虚拟隧道(p2pVM),对于 C 端来说这与普通 CS 无差异,但服务器会是云负载模型,FS 是文件支持的负载终端,VM 服务器则只负责转发任务.

在另一方面 FS2 有缓存管理机制,在一个时段内同一个文件只会触发一次传输协议,反复上传下载不会触发物理传输.  
**综合上述,FS2.0 的整体机制偏复杂,因此 FS3.0 需要从新设计.**

FS3.0 在存储系统使用了自定义数据引擎从底层解决了阵列难题,这一跨度非常大,直接从碎片结构的云服务器过度到数据中心,如果可以,它能让几万台阵列服务器一起来工作,轻松装下全世界的文件.

解决自定义数据引擎这一步非常困难,我首次设计的数据引擎使用线程池来调度任务,后来,直接砍掉这种设计,数据引擎内部不会自动管理线程池,而是放在外面自己开线程去增删查改.因为多线程需要统一化设计,ZDB2.0 的阵列系统本身就是线程模型,在自定义数据引擎再来一个全线程模型只会让引擎更加复杂,这会让未来维护升级举步维艰,或则是难以复制,例如通过小幅修改 FS3.0 更换成图片和视频的存储引擎.

FS3.0 在解决数据引擎环节,顺手给出了一个轻量级的 ZDB2.0 阵列支持库:Lite 大数据库,给出 Lite 库的作用是简化设计数据引擎复杂度.

FS3.0 从根本上解决了大数据难题,例如在 FS3.0 的 C 端,可以直接 post 和 get 大文件,相比 2.0 这不再需要自己编程去解决分块问题,直接解放了有文件数据接口功能的项目.(FS3.0 在大文件支持环节内部也是碎片工作,但给了优化算法,做到了在几乎 0 延迟的条件下,get 到大文件 body 任意位置的数据,而这些大文件可以有上亿个).

意义非凡:FS3.0 是一种对程序模型摸索的成果,后面将会是可预见的用户信息数据库,各种大数据库被一一实现.最近的一个项目,或则就是六代监控的起飞把.

# 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,推翻直接换 C4.

如果之前使用 ics,indy,win socket 的非 web 类项目,直接推翻换 C4,全面碾压

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 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

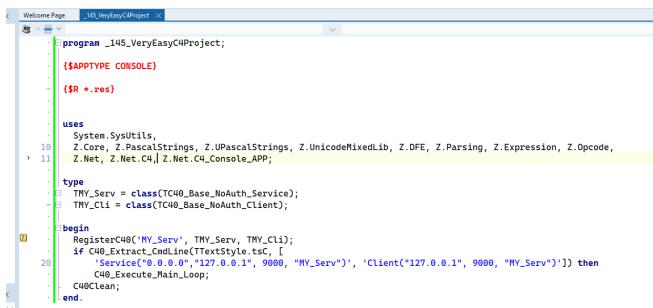
ZNet=大型 CS 服务器

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 _1u5_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(TC4_Base_NoAuth_Service);
  TMY_CLI = class(TC4_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;
  C40Clean;
end.
```

全文完.

2023-12-25

by.qq600585