

ZNet 使用手冊

文檔版本:1.4

更新時間:2024-1-26

更新日誌:

2023-10-4 開啟撰寫

2023-10-6 完成 1.0 撰寫

2023-10-9 追加雙主執行緒和互斥鎖技術資料

2023-10-11 追加性能工具箱資料

2023-10-12 追加 TCompute 概念+結構體概念,用概念說事因為代碼無法講完,代碼可由大向小靠自己挖掘.

2023-10-12 追加編譯類技術描述,具體代碼大家自己去挖掘把

2023-10-17 追加 XNAT 小節

2023-11-21 追加 ZDB2 大數據和 C4 入網小節

2023-12-25 追加主迴圈中微洩漏分析方案小節

2024-1-8 追加 Z.FragmentBuffer 庫的實現和原理小節

2024-1-26 追加內核庫啟動技術小節(許多原理和機制)

祝大家工作順利,專案大發.

ZNet 相關網站

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

開源備用站 <https://gitlab.com/passbyyou888/ZNet>

作者個人站 <https://zpascal.net>

ZNet 作者 qq600585

qq 群, 490269542(開源群),811381795(AI 群)

目錄

名词和关键机制	错误!未定义书签。
ZNet 的 6 种命令收发模型	错误!未定义书签。
ZNet 双通道模型	错误!未定义书签。
ZNet 线程支持	错误!未定义书签。
HPC Compute 线程分载方案	错误!未定义书签。
万兆以太网支持	错误!未定义书签。
使用 ZNet 必须知道主循环三件事	错误!未定义书签。
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	错误!未定义书签。
分析瓶颈	错误!未定义书签。
当分析出性能瓶颈以后,接下来的工作就是解决瓶颈	错误!未定义书签。
做服务器总是围绕硬件编程	错误!未定义书签。
架桥	错误!未定义书签。
ZNet 桥支持	错误!未定义书签。
C4 启动脚本书写方式	错误!未定义书签。
win shell 命令行方式:	错误!未定义书签。
linux shell 命令行方式	错误!未定义书签。
代码方式	错误!未定义书签。
C4 启动脚本速查	错误!未定义书签。
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	错误!未定义书签。
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)	15

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()	16
命令: Client_CMD_Info()	16
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	错误!未定义书签。
ZNet 内核技术-Soft Synchronize	错误!未定义书签。
内核:Check_Soft_Thread_Synchronize	18
内核:Check_System_Thread_Synchronize	18
ZNet 内核技术-双主线程	错误!未定义书签。
RTL 原主线程同步到次主线程	错误!未定义书签。
次主线程同步到 RTL 原主线程	错误!未定义书签。
双主线程开启以后的主循环	错误!未定义书签。
ZNet 性能工具箱使用指南	错误!未定义书签。
性能瓶颈分析	错误!未定义书签。
排除 ZNet 重叠 Progress	错误!未定义书签。
敬畏服务器主循环 progress	错误!未定义书签。
ZNet 的内核技术:TCompute 线程模型简介绍	22
ZNet 的内核技术:结构体系简单介绍	23
ZNet 的内核技术:简单说下结构组合拳	24
ZDB2 如何解决 Stream 写保护状态下的仿真读写	错误!未定义书签。
回顾:设计泛结构 TBigList<>	26
回顾:设计脚本引擎 ZExpression	26
ZNet 的母体移植技术:Z.Parsing	27
从交换机到拓扑简单说说 XNAT	错误!未定义书签。
性能分析:当 ZDB2 的数据规模达到 1 亿条	29
再说 C4 入网机制	错误!未定义书签。
当 C4 入网后需要干什么事情	错误!未定义书签。
ZNet 内置 CPM 可以代替 FRP+Nginx	错误!未定义书签。
C4 主要用于大项目吗?	错误!未定义书签。
主循环内存微泄漏分析	错误!未定义书签。
内核库在启动时都做了什么事情	错误!未定义书签。
如何推翻使用 ZNet 的项目	错误!未定义书签。
如何使用 ZNet 开发 web 类项目	错误!未定义书签。
ZNet 与 http 和 web	错误!未定义书签。
文本最后来一个极简 C4 的 CS demo	错误!未定义书签。

名詞和關鍵機制

- **卡佇列,卡伺服器**:卡主迴圈,通訊不流暢,如果伺服器帶有 UI 系統,UI 也會表現出假死
- **阻塞佇列,等待完成,等待佇列**:等待是 ZNet 的特有機制,佇列後面會等待前面完成,會嚴格按次序執行,等待會全部在本機等待,只有遠端影響後,本機佇列才會繼續,不是把佇列全部發送過去
- **序列化佇列**:不會等待資料回饋,直接發資料,而資料的接收順序是嚴格化的,按 1,2,3 序列發,那麼接收也會是按 1,2,3 進行觸發.例如使用序列化佇列發送 100 條,再發送一條阻塞佇列指令,待阻塞返回既表示 100 條序列已發送完成.同樣,例如上傳一個檔,耗時 1 小時,那麼先發檔,再發條阻塞,待阻塞返回,既表示檔發送成功.
- **非序列化佇列**:發送與接收均按嚴格序列機制處理,但是觸發接收後 ZNet 會在某些子執行緒或協程中做解碼這類程式處理,按 1,2,3 序列發送,接收資料以後會放到執行緒中處理,不會按 1,2,3 嚴格序列觸發接收事件.非序列化常用於對資料前後無要求的通訊.

ZNet 的 6 種命令收發模型

1. **SendConsole**:支援加密,支援壓縮,阻塞佇列模型,每次發送後都會等回饋,例如,先發 100 條命令,最後發一條 SendConsole,回饋回來時也表示 100 條命令都已經發送成功.SendConsole 很輕量,適合收發低於 64K 的小文本.例如 json,xml,ini,yaml.
2. **SendStream**:支援加密,支援壓縮,阻塞佇列模型,每次發送後都會等回饋,所有收發資料都會以 TDFE 進行編碼解碼,由於 TDFE 具備資料容器能力,因此 SendStream 常被用於應用資料收發,SendStream 同樣具備阻塞佇列能力.在流量方面可以支援更大的資料.
3. **SendDirectConsole**:支援加密,支援壓縮,序列化佇列模型,不會等回饋,用於收發基於字串的序列化資料.
4. **SendDirectStream**:支援加密,支援壓縮,序列化佇列模型,不會等回饋,使用容器打包資料,可以收發更大的序列化資料.
5. **SendBigStream**:不支援加密和壓縮,序列化佇列模型,解決超大 Stream 收發,例如檔,超大區塊資料.SendBigStream 工作機制每次只發送一部分,一直等待信號出現才會繼續發送,不會擠爆 socket 緩衝區.物理網路的頻寬和延遲都會影響 SendBigStream 工作效率.
6. **SendCompleteBuffer**:不支援加密和壓縮,序列化佇列模型,不會等回饋,高速收發核心機制,萬兆乙太網支援的核心機制.CompleteBuffer 設計思路就是圍繞網路來複製 Buffer,高速收發是一個非常重要的機制,取名叫做 CompleteBuffer.

ZNet 雙通道模型

雙通道是設計層面的概念,表示接收和發送各是一個獨立通道連接,信號收發被區別設計,早期雙通道是創建兩個連接來工作.後來經歷了無數摸索和升級,現在的雙通道是建立在 p2pVM 基礎上,p2pVM 可以在單連接基礎上虛擬出無限多虛擬化連接,這些 p2pVM 連接在應用層都是雙通道.

試想一下,過去我們堆出多台伺服器,需要定義無數多的偵聽,連接,埠,現在用 **p2pVM** 來虛擬化一切連接.因為降低了後臺技術複雜性,這給後臺系統提供了堆大的可維護性和規範性空間.例如 **C4** 的所有服務全部走 **p2pVM** 雙通道.

ZNet 執行緒支持

ZNet 天生支援在執行緒中發送資料,甚至是並行程式,發送的資料將會是非序列化佇列。

ZNet 的資料接收環節總是位於 Progress 焦點中,也就是觸發 Progress 那個執行緒,在多數情況下,ZNet 都建議在主執行緒中執行 Progress 主迴圈,例如 C4 框架就是用主執行緒跑 Progress.

ZNet 可以支持在執行緒中跑 Progress 從而達到執行緒收發,但並不建議這樣幹,因為 ZNet 有更好執行緒分載方案.

HPC Compute 執行緒分載方案

在 ZNet 中以 HPC 開頭的折疊代碼就是執行緒分載解決方案

流程:當資料從主執行緒到達,觸發接收事件,啟動分載,這時資料會轉移,並且建立一個新執行緒來執行處理,這樣幹以後,伺服器可以運行於無卡頓狀態.因為保護了主執行緒邏輯,穩定性優於自己開 Critical 管理執行緒.

分載方案可以支援除 **BigStream** 之外的全部命令收發模型。

分載不可以重疊執行:命令->HPC->HPC->返回,只能:命令->HPC->返回

命令->HPC 被觸發時是走的資料從主執行緒轉移到執行緒,並沒有發生資料 copy

```

1 // =====
2 // =====
3 // =====
4 // =====
5 // =====
6 // =====
7 // =====
8 // =====
9 // =====
10 // =====
11 // =====
12 // =====
13 // =====
14 // =====
15 // =====
16 // =====
17 // =====
18 // =====
19 // =====
20 // =====
21 // =====
22 // =====
23 // =====
24 // =====
25 // =====
26 // =====
27 // =====
28 // =====
29 // =====
30 // =====
31 // =====
32 // =====
33 // =====
34 // =====
35 // =====
36 // =====
37 // =====
38 // =====
39 // =====
40 // =====
41 // =====
42 // =====
43 // =====
44 // =====
45 // =====
46 // =====
47 // =====
48 // =====
49 // =====
50 // =====
51 // =====
52 // =====
53 // =====
54 // =====
55 // =====
56 // =====
57 // =====
58 // =====
59 // =====
60 // =====
61 // =====
62 // =====
63 // =====
64 // =====
65 // =====
66 // =====
67 // =====
68 // =====
69 // =====
70 // =====
71 // =====
72 // =====
73 // =====
74 // =====
75 // =====
76 // =====
77 // =====
78 // =====
79 // =====
80 // =====
81 // =====
82 // =====
83 // =====
84 // =====
85 // =====
86 // =====
87 // =====
88 // =====
89 // =====
90 // =====
91 // =====
92 // =====
93 // =====
94 // =====
95 // =====
96 // =====
97 // =====
98 // =====
99 // =====
100 // =====

```

萬兆乙太網支援

ZNet 使用 CompleteBuffer 機制來支援萬兆乙太網

- 在 ZNet 內部 SendCompleteBuffer 可以工作與執行緒中:執行緒機制可以為大流量資料提供預處理這類先決處理條件,例如 10 條執行緒做資料生成,然後 SendCompleteBuffer
- SendCompleteBuffer 具備高流量緩衝能力,當成規模的調用 SendCompleteBuffer 將被緩衝到暫存檔案,待主迴圈觸發時成片的 CompleteBuffer 才會被發送出去:該機制對長佇列資料提供了緩存機制,但不可以暴力 Send,一般情況下,10 次 SendCompleteBuffer 可以配上一次 SendNull(阻塞佇列),這樣幹會讓整體網路的負載和吞吐保持良好狀態.
- SendCompleteBuffer 可以與 DirectStream 重疊
例如伺服器註冊的命令模型是 DirectStream,可以使用 SendCompleteBuffer 來發送
- SendCompleteBuffer 支援直接發送 TDFE 資料
- SendCompleteBuffer 可以與 Stream 阻塞模型重疊
如果伺服器註冊的命令模型是阻塞 Stream,用戶端使用 SendCompleteBuffer 發送時是非阻塞回應模式,既發送一個 buffer 資料,也會收到一個 buffer 資料,在這一過程中並不會出現阻塞等待.
- 萬兆乙太收發大資料如果不使用 CompleteBuffer,一個 100M 的資料可能會讓發送端先卡 5 秒並出現 UI 假死,發送出去以後,接收端又會卡頓 5 秒出現回應停頓,這是因為大量的資料複製,編碼,壓縮,解壓縮佔用了主執行緒的計算資源導致,這種機制只能處理體積非常小的資料.
- 在萬兆使用 CompleteBuffer 發送一個 100M 資料,從發送到接收,兩邊的處理延遲可以小於 10ms,這樣的伺服器模型永遠都是立即回應.
- 總結:執行緒+磁片緩存+SendNull 阻塞+DirectStream 重疊+Stream 重疊=用 CompleteBuffer 成功解決萬兆乙太問題

使用 ZNet 必須知道主迴圈三件事

第一件事:遍歷並且處理每個 IO 的資料接收流程

ZNet 的物理網路介面大都用獨立執行緒,這個執行緒是無卡的,當主執行緒被完全佔用,例如正在處理 100M 的壓縮+編碼任務,這時候接收執行緒內部仍然處在正常工作中.子執行緒接收的資料的並不會放在記憶體中:接收程式會根據資料體量來判斷是否用暫存檔案來暫存資料.

當主迴圈被觸發時,主迴圈會工作與對應的執行緒中,主迴圈永遠都是單執行緒模型,程式會從子執行緒中取出已經收到的資料,包括暫存檔案資料,然後進入網路資料粘包處理環節.

在資料粘包處理環節,ZNet 使用了大量記憶體投影技術來避免記憶體 copy,這是將記憶體位址映射成 TMemoryStream,在 ZNet 中記憶體投影使用 TMS64,TMem64 這類工具.

粘包處理系統會含有 cpu 時間消耗度,由於粘包系統裡面包含了序列包和 p2pVM 這類大型子系統,因此 ZNet 給出了 cpu 時間消耗度,該數值達到臨界點,粘包處理系統將會中斷粘包流程,並且在下次主迴圈才會繼續處理.例如發送了 1 萬條命令,粘包臨界點是 100ms,當達到 100ms 時粘包只處理了 2000 條命令,那麼剩下的 8000 條將會在下次粘包時進行處理.

在 ZNet 的實際運行中粘包流程幾乎沒有記憶體 copy,單執行緒裡面的粘包處理能力每秒會達到數十萬的處理水準.並不需要開闢執行緒或則協程在處理.做個執行緒加速這類想法,很不實際,這不僅會加大 ZNet 的內核複雜度,效率提升也會非常一般,只能是給 Newbie 解決了胡亂高速粘包,而正確的高速粘包,只需要使用 SendCompleteBuffer 發資料進來就行了.

第二件事:遍歷並且處理每個 IO 的資料發送流程

ZNet 中的所有發送命令最終都會降落到具體每個 IO 裡面

在這些 IO 中,會有個正在等待發送的命令資料佇列,這些佇列資料,有的會存在於記憶體,有的會存在於暫存檔案.

ZNet 會遍歷並發送 IO 中的待發佇列中的嚴格序列化資料,這些資料將會放到物理的 IO 待發緩衝區中,這一層的緩衝區就不是 ZNet 可以控制的了.

如果使用 p2pVM,StableIO 這類虛擬化通訊協定,在遍歷發送時,嚴格序列化資料會直接被重新封裝,然後再放到物理 IO 緩衝區.

發送的全部物理緩衝資料,是被拓撲網路的信號系統所控制,每個含有 tcp 標籤的 ip 包,都需要有一個終端回饋信號(遠端接收端,不是內網拓撲),這個包才能夠被送到,這種回饋信號就是網路延遲.回饋在 ZNet 裡面是發送速度的快慢.

第三件事:管理好每次遍歷的 cpu 開銷

ZNet 會記錄每次訓練遍歷 IO 的時間開銷,達到臨界,遍歷 IO 將會中止,下次主迴圈再繼續遍歷.這樣幹,當伺服器達到一定負載以後,遠端回應將會變慢,而伺服器本身則是在單執行緒的主迴圈中走分片負載的技術路線.

優化資料傳輸

首先明確優化目標:避免用戶端卡住 UI 以及避免伺服器卡住主迴圈,這需要作為一個專案或產品整體對待.例如伺服器卡主迴圈,那麼回應速度將會出現延遲,前端發個請求過來,等上 5 秒才回應.

當明確目標後,首要工作是分析卡頓瓶頸,大多數 CS 或 web 型專案,可以直觀通過用戶端發送的命令來定位瓶頸,例如 GetDataList 命令,出現 5 秒卡頓,直接定位到伺服器的 GetDataList 回應環節去就行了.但有時候,ZNet 命令輸送量出現空前規模,例如數百台伺服器之間,以及數千個 IOT 網路設備間的通訊,這些命令密密麻麻,這時就需要 ZNet 提供支援資訊

分析瓶頸

如果未使用 C4 框架,需要自己把代碼添加到伺服器應用或則控制台去.

輸出每條命令在伺服器的 cpu 耗損

```
ZNet_Instance_Pool.Print_Service_CMD_Info;
```

輸出伺服器運行狀態統計

```
ZNet_Instance_Pool.Print_Service_Statistics_Info;
```

如果使用了 C4 框架,可以直接在控制台輸入 Service_CMD_Info

在輸出命令中,會包含所有命令收發的 cpu 耗損,如果某些資訊含有":HPC Thread"字樣,表示這條命令使用了 HPC 執行緒分載,例如, GetDataList:HPC Thread": time 123078ms

表示 GetDataList 在 HPC 執行緒分載中最長的一次處理運行了 2 分鐘

如果使用了 C4 框架,可以通過 HPC_Thread_Info 控制台命令,即時監控執行緒分載

執行緒分載會給每個執行緒賦予一條正在執行的命令資訊,HPC_Thread_Info 會輸出 C4 伺服器的全部執行緒狀態.然後,結合系統的任務和資源監視工具,一直守著務器運行,基本都能分析出性能瓶頸.

當分析出性能瓶頸以後,接下來的工作就是解決瓶頸

如果伺服器走主執行緒路線,解決瓶頸只有兩方面工作,首先是考慮執行緒分載用 HPC 函數開執行緒命令.其次是考慮在用戶端使用 SendCompleteBuffer 來替代 SendDirectStream.

如果伺服器本身就是多執行緒設計路線,這會需要從鎖,結構,流程這些地方下手來搞,如果從整體來優化多執行緒的伺服器,事情很複雜:你會從問題的傳導分析再到定位問題點,最後調整流程.最後大概率會使用一個演算法來解決問題.例如 ZNet 作者的監控專案在搜索視頻時總是等待很久,最終作者設計了一個按時間跨度存視頻的加速演算法.

伺服器堆大以後,一個後臺服務也許會到達 10 萬行規模,很多優化工作,也會是 fixed bug 的工作.這些優化工作會區分,初期,中期,後期,越靠近初期,fixed bug 的頻率越高,到後期,也許整個伺服器後臺都被推翻重構 1-2 次了,這是一個經驗和設計上的問題.

做伺服器總是圍繞硬體程式設計

很多開源項目看似都很簡單易懂,在真實的伺服器專案中,規模也許會比開源項目大上 100 倍.從量變到質變,規模上升 100 倍,就不再是常規技術方案了.

很多人做伺服器是調度資料庫+通訊系統,如果是做 web,伺服器還會包含設計 ui 系統.如果專案規模很小:通訊頻率低+通訊資料量小,這種伺服器怎麼做都沒有問題.

當伺服器規模開始偏大:通訊頻率高+通訊資料量大,將會面臨:**圍繞硬體程式設計**.

6 核 12 超線和 128 核 256 超線,這在硬體定位上是不一樣的,它會影響執行緒和伺服器的設計模型,6 核規格硬體幾乎無法開出並行程式模型,那怕一個環節開出並行 for,也許整個伺服器都會受牽連,6 核的只能開出常規執行緒對特別繁重的計算環節跑執行緒分載.當伺服器運行於 128 核平臺時,就沒有 cpu 計算資源的問題了,而是合理安排計算資源,例如,4 執行緒的平行計算,有時候會比使用 40 個執行緒速度更快.最後是系統瓶頸,在不使用三方執行緒支援函式庫情況下,**win 系統的單進程最多只能同時使用 64 個超線**,只有掛載了 TBB 這類庫以後,才能同時使用 256 個超線.全執行緒,部分執行緒,單執行緒,這種伺服器在設計之初定位就已經完全不一樣.伺服器程式的中心是圍繞不同時代的硬體趨勢,最大的道理硬體平臺,框架設計環節是小道,只有明確了圍繞硬體為中心,再來做伺服器設計和程式設計,這才是正確的路線.

再比如 8 張 4T 全閃 sdd 配 192G 記憶體,以及 8 張 16T 的 hdd 配 1TB 記憶體,這種伺服器在資料存儲,緩存系統設計上也是不一樣的,例如**資料規模到 30 億條,空間佔用到 30T,這種規模基本上 192G 記憶體會很吃緊**,但不是問題,仔細優化後也能跑,因為資料要**加速搜索或存儲提速**永遠都是用緩存,而當存放裝置的硬體設定**使用 hdd 並且容量達到 100T,記憶體 1TB,這種設計將比 192G 更加需要優化緩存**,流量進來以後有可能光是寫緩存就直接吃掉 0.99TB 記憶體,程式在設計之初就已經定位好了用 10G 來 hash 索引,開各種優化演算法的結構,這 2 種不同硬體設定,在伺服器的系統設計層面,是不一樣的:192G 只需要考慮優化緩存規模,1TB 需要考慮優化緩存規模+防止崩潰,因為 hdd 寫大資料遇到流量>陣列寫入極限後非常容易崩潰,**大陣列的記憶體一旦用完,陣列的 IO 能力也許會下降到原有能力的 5%,與崩潰無差異,緩存控制(flush)將會是直接提上上臺的核心機制之一**,這需要從整體上控制住大資料登錄端,陣列寫機制,硬體鎖這些關鍵要素.

最後是 gpu,一旦伺服器碰上 gpu,支援設備從 cpu 到存儲幾乎全都會是高配,這時候,**伺服器在程式設計環節將會徹底脫離古典的單執行緒方式,流程模型將會被流水線模型所取代**,這些流水線會從一個作業系統到另一個作業系統流來流去.這時候 ZNet 程式會全體走 Buffer+執行緒的路線,並且這種模型只是把資料接進來,計算主體是一堆獨立+巨大的計算支援系統.

架橋

架橋是一種通訊模式,有點偏設計模式,它能實實在在提升伺服器群的程式設計效率.

架橋只能工作在帶有回饋請求的命令模型中

- SendStream+SendConsole,帶有佇列阻塞機制的請求,會等回應
- SendCompleteBuffer_NoWait_Stream,不會阻塞佇列的請求,不等回應

架橋的工作流程:

- A->發出請求->命令佇列開始等待模型
- B->收到請求->請求進入延遲回饋模型->架橋 C
- C->收到請求->C 回應請求
- B->收到 C 響應->B 響應回 A
- A->收到 B 響應,跨服通訊流程完結

架橋是用 1-2 行代碼解決繁瑣的伺服器群間資料傳遞流程,

```
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,伺服器的種類數十種,它們,架橋技術是以高效方式來享受這些伺服器資源.在 C4 框架中的全部伺服器都支援架橋與被架橋.

編寫 C4 伺服器時只管按正常的通訊作業程式設計,直接考慮處理 C 端,無限堆砌.待完成後開個應用伺服器,把所有的伺服器以架橋方式全部調度起來使用即可.

ZNet 橋支持

ZNet 的橋支援就是事件原型,回應式通訊都會有回饋事件,讓事件指向一個已有的自動程式流程,回饋事件觸發時直接自動處理.

- `TOnResult_Bridge_Templet`:橋回饋事件原型範本
- `TProgress_Bridge`:主迴圈橋,掛接到 ZNet 的主迴圈後,每次 `progress` 都會觸發事件,這種模型在早期 ZNet 還沒有解決 hpc 分載執行緒做資料搜索大流程時,主迴圈橋常被用於分片計算,例如 1000 萬的資料搜索在主迴圈幹就是每次 `progress`,搜索 10 萬條,以此保證伺服器不卡.
- `TState_Param_Bridge`:以布林狀態回饋的橋
- `TCustom_Event_Bridge`:半自動化回應式模型橋,需要程式設計的橋,例如訪問 10 台伺服器,待全部訪問完,再一次性回應給請求端.C4 大量使用.
- `TStream_Event_Bridge`:`SendStream` 的自動化回應橋
- `TConsole_Event_Bridge`:`SendConsole` 的自動化回應橋
- `TCustom_CompleteBuffer_Stream_Bridge`:半自動化的 `CompleteBuffer` 回應事件橋
- `TCompleteBuffer_Stream_Event_Bridge`: `CompleteBuffer` 的自動化回應橋

C4 啟動腳本書寫方式

win shell 命令列方式:

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

linux shell 命令列方式

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

```

代碼方式

```
C40AppParsingTextStyle := TTextStyle.tsC; //為了方便書寫腳本,使用 C 風格文本運算式  
C40_Extract_CmdLine([  
  'Service("0.0.0.0", "127.0.0.1", 8008, "DP")',  
  'Client("127.0.0.1", 8008, "DP")'])];
```

C4 啟動腳本速查

函數:KeepAlive(連接 IP, 連接 Port, 註冊用戶端),

參數重載:KeepAlive(連接 IP, 連接 Port, 註冊用戶端, 過濾負載)

別名,支持參數重載:KeepAliveClient, KeepAliveCli, KeepAliveTunnel, KeepAliveConnect, KeepAliveConnection, KeepAliveNet, KeepAliveBuild

說明:用戶端連接伺服器,部署型入網連接,如果連接目標不成功會一直嘗試,連接成功後會自動啟動斷線重連模式.在部署伺服器群時,主要使用 KeepAlive 方式入網,無論 C4 的構建參數怎麼變化,KeepAlive 會總是反復嘗試不成功的連接,KeepAlive 方式解決了部署伺服器的啟動順序問題,只要在腳本中使用 KeepAlive 入網可以無視伺服器部署順序問題.KeepAlive 不會搜索整個通訊伺服器棧,需要在 C4 網路中部署 DP 服務,KeepAlive 這樣才能跨服入網.簡單解釋:使用 KeepAlive 入網 C4,需要掛載一個 DP 服務.

函數:Auto(連接 IP, 連接 Port, 註冊用戶端)

參數重載:Auto(連接 IP, 連接 Port, 註冊用戶端, 過濾負載)

別名,支持參數重載:AutoClient, AutoCli, AutoTunnel, AutoConnect, AutoConnection, AutoNet, AutoBuild

說明:用戶端連接伺服器,非部署型的入網機制,入網失敗後無法自動化反復入網,Auto 是自動型入網連接,可以工作於沒有 DP 服務的 C4 網路,適用於在有人操作啟動的伺服器使用,入網一旦成功就會進入斷線重連模式.

函數: Client (連接 IP, 連接 Port, 註冊用戶端)

不支持參數重載

別名,支持參數重載:Cli, Tunnel, Connect, Connection, Net, Build

說明:用戶端連接伺服器,非部署型的入網機制,入網失敗後無法自動化反復入網,Client 函數需要 C4 目標 IP 的網路有 DP 服務才能入網.

函數: Service (偵聽 IP, 本機 IP, 偵聽 Port, 註冊伺服器)

參數重載: Service (本機 IP, 偵聽 Port, 註冊伺服器)

別名,支持參數重載:Server, Serv, Listen, Listening

說明:創建並啟動 C4 伺服器

函數: Wait(延遲的毫秒)

別名,支持參數重載:Sleep

說明:啟動延遲,因為 win32 命令列如果不使用 powershell 腳本,處理延遲執行比較麻煩

函數: Quiet(Bool)

說明:安靜模式,預設值 False

函數: SafeCheckTime(毫秒)

說明:長週期檢查時間,預設值 45*1000

函數: PhysicsReconnectionDelayTime(浮點數,單位秒)

說明:C4 入網以後,如果物理連接斷線,重試連接的時間間隔,預設值:5.0

函數: UpdateServiceInfoDelayTime (單位毫秒)

說明:DP 調度伺服器的更新頻率,預設值 1000

函數: PhysicsServiceTimeout (單位毫秒)

說明:物理伺服器的連接逾時,預設值 15*60*1000=15 分鐘

函數: PhysicsTunnelTimeout (單位毫秒)

說明:物理用戶端的連接逾時,預設值 15*60*1000=15 分鐘

函數: KillIDCFaultTimeout (單位毫秒)

說明:IDC 故障判定,斷線時長判定,達到該值觸發 IDC 故障,斷線的用戶端會被徹底清理掉
預設值 h24*7=7 天

函數: Root (字串)

說明:設置 C4 工作根目錄,預設值為.exe 檔目錄,或則 linux execute prop 檔案名錄.

函數: Password (字串)

說明:設置 C4 的人網密碼,預設值為 DTC40@ZSERVER

UI 函數: Title (字串)

說明:只能工作與 C4 的標注 UI 範本,設置 UI 視窗標題

UI 函數: AppTitle (字串)

說明:只能工作與 C4 的標注 UI 範本,設置 APP 標題

UI 函數: DisableUI (字串)

說明:只能工作與 C4 的標注 UI 範本,遮罩 UI 操作

UI 函數: Timer (單位毫秒)

說明:只能工作與 C4 的標注 UI 範本,設置 UI 環境下的主迴圈毫秒週期

C4 Help 命令

Help 命令是 C4 內置的伺服器維護+開發調試命令.無論是 console 還是 ui,都內置了 help 命令,這些命令是通用的.

命令:help

說明:顯示可用命令清單

命令:exit

別名:close

說明:關閉伺服器

命令:service(ip 位址, 埠)

重載參數: service(ip 地址)

重載參數: service()

別名:server,serv

說明:伺服器內部資訊報告,包括物理伺服器資訊,p2pVM 伺服器,連接數量,流量,伺服器內置啟動參數.如果空參數會簡易報告.

命令:tunnel(ip 位址, 埠)

重載參數: tunnel(ip 地址)

重載參數: tunnel()

別名:client,cli

說明:用戶端內部資訊報告.如果空參數會簡易報告.

命令:reginfo()

說明:輸出已經註冊的 c4 服務,c4 的每個服務都會有對應的 CS 模組,例如 DP 會有 dp 伺服器+dp 用戶端.

命令:KillNet(ip 位址, 埠)

重載參數: KillNet (ip 地址)

說明:直接以 IDC 故障方式殺掉對應的 c4 網路服務

命令:Quiet(布林)

重載參數: SetQuiet(布林)

說明:切換安靜模式,在安靜模式下,伺服器不會輸出日常命令執行狀態,但出錯有提示,例如命令模型執行異常

命令: **Save_All_C4Service_Config()**

說明:

立即保存當前伺服器參數,這是一個伺服器擴展參數,當 c4 堆大以後伺服器參數太多太多,shell 命令列最長限制是 8192,在正常情況下,根本無法在 shell 命令寫太多啟動參數,因此 c4 提供了檔形式的參數載入方式,在預設情況下,並沒有參數檔

通過 **Save_All_C4Service_Config()**可以生成尾碼為.conf 的伺服器參數檔,.conf 檔存放在當前伺服器目錄對應的 depnd 子目錄中,.conf 是個 ini 格式的設定檔.

如果使用.conf 作為伺服器參數來啟動 c4,命令列的參數將會被覆蓋.

Save_All_C4Service_Config()多用於首次運行伺服器時部署啟動參數使用,主要是作用是減少命令列的輸入規模.真實系統集成中,命令列達到一個 200,300 字元,這是非常不易於閱讀修改的.因此.conf 啟動參數是部署 c4 的重要環節.

命令: **Save_All_C4Client_Config()**

說明:立即保存當前構建完成的所有 C4 客端參數,作用與 **Save_All_C4Service_Config()**基本一致.在系統集成工作中,用戶端參數都很少,這是可以直接寫進 shell 命令列的,但如果要美化命令列,使其易於閱讀,那就用文件參數把.

命令: **HPC_Thread_Info()**

說明:立即輸出當前進程中的全部 **TCompute** 執行緒實例,Z 系執行緒一律使用 **TCompute** 創建與執行,並且每個執行緒都會有個 **thread_info** 的字串識別字,用於識別這條執行緒的作用.在 ZNet 中執行緒會非常繁多,有 **HPC 分載執行緒**,**CompleteBuffer 後臺解碼編碼執行緒**,**ZDB2 執行緒**.如果直接使 RT 庫自帶的 **TThread**,那麼 **HPC_Thread_Info()**是不會輸出該執行緒狀態的.

該命令多用於伺服器調試,分析性能瓶頸,找 bug 時使用

命令: **ZNet_Instance_Info()**

別名: **ZNet_Info()**

說明:立即輸出 ZNet 的全部 IO 實例,包括物理連接,p2pVM 連接.多用於診斷連接狀態,分析 C4 入網時遇到的問題.

命令: **Service_CMD_Info()**

別名: **Server_CMD_Info()**

說明:立即輸出伺服器中全部命令模型的 cpu 消耗度統計狀態,這些命令會非常多,數百個.多用於在分析性能瓶頸定位.用 **Service_CMD_Info()**也會包含發送命令的次數統計,但不包含發送命令的 cpu 消耗度.

命令: **Client_CMD_Info()**

別名: **Cli_CMD_Info()**

說明:立即輸出全部用戶端命令模型的 cpu 消耗度統計,與 **Service_CMD_Info()**格式幾乎相同,因為 C4 是個交互網路,用戶端統計會折射出伺服器的延遲.

命令: **Service_Statistics_Info()**

別名: **Server_Statistics_Info()**

說明:立即輸出全部伺服器的內部統計資訊,包括 IO 的觸發頻率,加密計算頻率,主迴圈頻率,收發的資料量等等關鍵資訊,伺服器會包括物理伺服器+p2pVM 伺服器

命令: **Client_Statistics_Info()**

別名: **Cli_Statistics_Info()**

說明:立即輸出全部用戶端的內部統計資訊,輸出格式與 **Service_Statistics_Info()**幾乎相同

命令: **ZDB2_Info()**

說明:立即輸出 ZDB2 的資料庫狀態,ZDB2 是一套分層次架構的資料庫系統,目前 ZDB2 已經進步到第三代體系,這裡的 **ZDB2_Info()**也是輸出第三代 ZDB2 體系,在 C4 框架組成的 FS2,FS,這類服務,凡是 2021 年做出的 C4 服務,都是第二代 ZDB2 體系,無法被 **ZDB2_Info()**統一化的輸出狀態. **ZDB2_Info()**輸出三代體系的信息量和設計非常龐大,這裡只能一筆帶過:在第六代監控的資料庫和後臺用 **ZDB2_Info()**看狀態會是一個好辦法.

命令: **ZDB2_Flush()**

說明:將 ZDB2 寫緩存立即刷入物理設備,使用資訊與 **ZDB2_Info()**都有非常龐大的信息量,這裡只能一筆帶過:在第六代監控的資料後臺調試陣列系統硬體時用的命令,需要結合磁片緩存監控,記憶體監控,物理 IO 監控一起來使用.作用是分析出陣列系統的 IO 瓶頸.

ZNet 內核技術-鎖複用

Critical-Section 是作業系統基於硬體的執行緒鎖技術

進程中所包含的執行緒越多,Critical-Section 就會對應越多,在系統監視器,都可以看到執行緒數量+進程的控制碼數量,這兩者數量多了以後,整個系統也許會不太穩定,至少在分析進程或則系統崩潰時,目標進程的執行緒+控制碼是 2 個非常重要指標.

通常來說,每個執行緒會對應至少 1 個以上的 Critical-Section 控制碼,看具體流程編寫.

Z 系內核對 Critical-Section 是走的複用路線,ZNet 伺服器運行起來會在監視看到控制碼峰值,但是這不是真實 Critical-Section,需要通過命令 c4 控制台輸入 hpc_thread_info 才能看到真實的 Critical-Section 和執行緒狀態.

ZNet 內核技術-Soft Synchronize

Soft Synchronize 技術是模擬 rtl 的主執行緒 Synchronize.

ZNet 的設計機制大量依賴主執行緒,因此大量使用 Thread Synchronize 體系,在 ZNet 的非同步通訊庫中 DIOCP/CrossSocket 執行緒間調用也使用了 Thread Synchronize 機制,控制執行緒啟停等操作.其中用的比較多的還是 WaitFor 執行緒間的互斥等待,假如佇列沒有處理完成,給執行緒發 exit 命令容易卡在裡面,這時候問題往往由外面程式沒有正確清空執行緒間執行調用,清理執行緒間執行程式這種操作,其實就是 CheckSynchronize,這是一個主執行緒專用的同步佇列執行調用.凡是執行緒中出現了 Synchronize 操作,只能通過 CheckSynchronize 回應,在 vcl form 體系中這是有 application 自動調用的,如果繞開 application 這需要掌握主迴圈技術.

例如給出了 Thread.OnTerminate 中,如果外面不給 CheckSynchronize,會一直不觸發事件.

Soft Synchronize 解決了執行緒間的事件傳遞機制,同時替代 CheckSynchronize.

內核:Check_Soft_Thread_Synchronize

執行模擬主執行緒 Synchronize 代碼,不會執行 RTL 系統 Synchronize 代碼.Z 系一律使用該方式處理執行 Synchronize 代碼,包括 DIOCP/Cross/ICS8/ICS9/Indy/Synapse.例如當非同步庫 cross/diocp 使用 waitfor 操作,這會讓 wait 過程中,執行模擬主執行緒的 Synchronize 一直工作.

當編譯開關 Core_Thread_Soft_Synchronize 被關閉時,將使用 RTL 系統 Synchronize 機制.

該 API 主要支援程式需要在雙主執行緒環境下運行.

內核:Check_System_Thread_Synchronize

執行 RTL 系統 Synchronize 代碼,同時也會執行模擬主執行緒 Synchronize 代碼自動處理狀態,無視編譯開關 Core_Thread_Soft_Synchronize 打開或關閉.

該 API 主要支援程式在主執行緒環境下運行.

ZNet 內核技術-雙主執行緒

ZNet 可以在單進程同時開兩個主執行緒,當雙主線模型啟動以後,會發生如下事情:

- ZNet 全系和 ZNet 包含的各種庫,一律工作於次主執行緒.
- RTL 全系,包括原生 lcl,vcl,fmx,一律工作于原主執行緒.
- 次主線和原主線會各自維持自己的主迴圈,主迴圈技術這裡省略
- 次主線和原主線互相訪問資料需要使用 Synchronize 技術
- 雙主執行緒技術可以支援 win/android/ios 以及 fpc 所構建的 Linux 程式
- 帶 UI 的程式,跑伺服器不會再有卡頓感
- 把 ZNet 放在 1 個 dll/ocx 運行,等同於開了 2 個 exe,其中 exe 與 dll 各走一條主執行緒
- 次主執行緒完全可以跑 http,c4,znet

RTL 原主執行緒同步到次主執行緒

在雙主執行緒模式以後,從 RTL 主執行緒訪問次主執行緒的資料

```
TCompute.Sync(procedure  
begin  
    // 這裡訪問 ZNet 裡面的資料,包括處理 c4,cross,diocp,ics 這些通訊資料  
end);
```

次主執行緒同步到 RTL 原主執行緒

在雙主執行緒模式以後,就是從 ZNet 訪問 RTL 主執行緒的資料,就是從 ZNet 訪問 VCL/FMX

```
TThread.Synchronize(TThread.CurrentThread, procedure  
begin  
    // 這裡訪問和修改 vcl/fmx,UI 在這些資料  
end);
```

雙主執行緒開啟以後的主迴圈

ZNet 次主執行緒 API,Check_Soft_Thread_Synchronize,位於 Z.Core.pas 庫

RTL 原主執行緒 API,CheckSynchronize,位於 vcl-System.Classes.pas/lcl-classes.pas 庫

ZNet 性能工具箱使用指南

CPS 工具箱=Caller Per second tool.所有 cps 計數週期為 1 秒.位於 Z.Core.pas 庫.

- **CPS_Check_Soft_Thread**:次主迴圈效能計數器.
- **CPS_Check_System_Thread**:RTL 主迴圈效能計數器.

存取方法, CPS_Check_Soft_Thread.CPS,該值為每秒調用次數

ZNet 的所有實例都內置了 CPS 效能計數器,用於計算伺服器主迴圈每秒調用頻率以及 cpu 佔用..

性能瓶頸分析

啟動任意 C4 程式,命令列敲 **hpc_thread_info**,得到如下回饋

```
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 主迴圈每秒調用次數

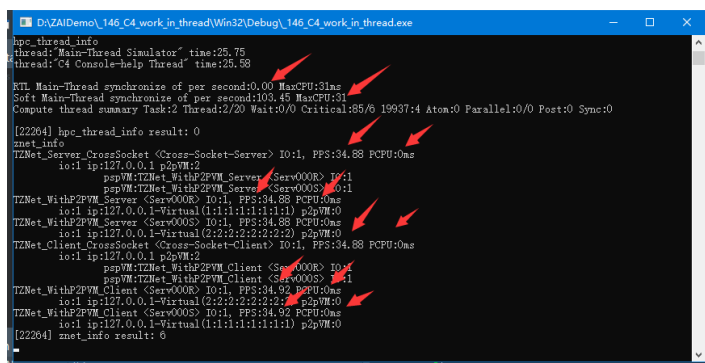
對應 **CPS_Check_System_Thread**

Soft Main-Thread synchronize of per second:167.32,次主迴圈每秒調用次數

對應 **CPS_Check_Soft_Thread**

結果:這個 C4 程式,使用的是次主迴圈技術,主迴圈每秒發生 167 次調用,調用頻率越高說明流暢度越好,如果程式發生卡頓,就需要檢查卡頓的點:也許某個函數佔用了傳導給了主迴圈,導致 CPS 過低,尤其開了 timer 這類事件的程式.

命令列敲 **znet_info**,得到如下回饋



```
D:\ZANDemo\146_C4_work_in_thread\Win32\Debug\146_C4_work_in_thread.exe  
hpc_thread_info  
thread:"Main-Thread Simulator" time:25.75  
thread:"C4 Console-help Thread" time:25.58  
RTL Main-Thread synchronize of per second:0.00 MaxCPU:31ms  
Soft Main-Thread synchronize of per second:103.45 MaxCPU:31  
Compute thread summary Task:2 Thread:2/20 Wait:0/0 Critical:85/6 19937:4 Atom:0 Parallel:0/0 Post:0 Sync:0  
[22264] hpc_thread_info result: 0  
znet_info  
TZNet_Server CrossSocket <Cross-Socket-Server> IO:1, PPS:34.88 PCPU:0ms  
io:1 ip:127.0.0.1 p2pVM:0  
pspVM:TZNet_WithP2PVM_Server <Serv000R> IO:1  
pspVM:TZNet_WithP2PVM_Server <Serv000S> IO:1  
TZNet_WithP2PVM_Server <Serv000R> IO:1, PPS:34.88 PCPU:0ms  
io:1 ip:127.0.0.1-Virtual(1:1:1:1:1:1:1) p2pVM:0  
TZNet_WithP2PVM_Server <Serv000S> IO:1, PPS:34.88 PCPU:0ms  
io:1 ip:127.0.0.1-Virtual(2:2:2:2:2:2:2) p2pVM:0  
TZNet_Client CrossSocket <Cross-Socket-Client> IO:1, PPS:34.88 PCPU:0ms  
io:1 ip:127.0.0.1 p2pVM:2  
pspVM:TZNet_WithP2PVM_Client <Serv000R> IO:1  
pspVM:TZNet_WithP2PVM_Client <Serv000S> IO:1  
TZNet_WithP2PVM_Client <Serv000R> IO:1, PPS:34.92 PCPU:0ms  
io:1 ip:127.0.0.1-Virtual(2:2:2:2:2:2:2) p2pVM:0  
TZNet_WithP2PVM_Client <Serv000S> IO:1, PPS:34.92 PCPU:0ms  
io:1 ip:127.0.0.1-Virtual(1:1:1:1:1:1:1) p2pVM:0  
[22264] znet_info result: 0
```

PPS:ZNet 中的 progress 每秒調用頻率

PCPU: ZNet 中的 progress 最大 cpu 消耗

結果:定位出主迴圈的長流程到底卡在哪個實例中,這些實例可以是伺服器,也可以是用戶端.定位完成後,通過 **Service_CMD_Info** 和 **Client_CMD_Info** 找執行命令的卡頓瓶頸.

如果程式未使用 C4,可以使用 API, **ZNet_Instance_Pool.Print_Status**,直接輸出狀態

排除 ZNet 重疊 Progress

首先 progress 具有自動防閉環機制,progress 包 progress 不會閉環.

C4 已經優化過 progress 重疊問題,每次調用 C4Progress 可以確保每個 ZNet 實例只會觸發一次 Progress

在非 C4 框架中 Progress 可以被 p2pVM,DoubleTunnel 自動觸發,一個主執行緒迴圈可能會引發 2-5 倍的 progress,這是無意義的消耗,如果伺服器負載多了,反復重疊的 progress 會讓分片負載無法準確估算.當需要精確優化這是必須解決的問題.

使用如下程式範式

- Server.Progress;
- Server.Disable_Progress; 這裡遮罩調以後,後面不會觸發 server.progress
- other.progress;
- Server.Enabled_Progress;

p2pVM 的物理隧道實例每次 progress 會自動遍歷裡面的全部 p2pVM 虛擬連接.

當幹完這些事以後,使用 ZNet_Instance_Pool.Print_Status 查看 cps 變化,如果 ZNet 實例的 cps 值與主迴圈 cps 相近,那 progress 基本沒問題.接下來測試一下連接,處理命令,待全部通過,那麼解決重疊 progress 問題宣告完結.

敬畏伺服器主迴圈 progress

幾乎所有的伺服器優化工作都會面臨主迴圈問題,這裡涉及了非常多的解決辦法

- **執行緒**:如果主迴圈的代碼支持執行緒安全,那麼用執行緒會很不錯,執行緒安全不是鎖住就安全,而是執行緒+主迴圈不會因為主迴圈開在執行緒中而發生卡鎖.
- **分片**:分片技術是把計算量切割出來,每次主迴圈只運行一部分
- **狀態機**:狀態機是讓主迴圈在某種環境下,直接省略不處理某些代碼.尤其事涉及到 for,while 這類流程
- **結構和演算法優化**:在主迴圈中會處理大量的結構,對結構的優化,例如 hash,biglist,可以有效提升主迴圈效率.
- **CPU 指令級別的優化可以無視**:例如 sse,avx,這種優化難搞不說,很難數倍提升,無法和演算法級優化相提並論,演算法優化普遍起步就是 10 倍提升.

主迴圈是主執行緒的命脈,90%伺服器的調度程式都使用主執行緒來完成,子執行緒和協程大多用於分擔某些特殊任務.例如在 pas 圈很多伺服器喜歡上一個 ui,隨時看到服務求的運行狀態,這種 vcl/fmx/lcl 路線的 ui 都是跑在主執行緒下,這會被主迴圈深深影響.

而全執行緒化的伺服器,例如 erlang,go,會有一個執行緒調度問題,這裡會使用許多複雜的程式機制來控制執行緒間的協作問題,並且全執行緒化計算的伺服器並不會讓項目變得十分流暢,更不會天下無敵,因為伺服器底層被硬體極限影響.

主迴圈+子執行緒,未來仍然會是伺服器的主要模型.主迴圈沒問題可以等同於已經解決好了 50%的伺服器性能瓶頸!!

ZNet 的內核技術:TCompute 執行緒模型簡介紹

TCompute 並不是執行緒技術,而是一種使用執行緒程式設計的規範模型.

TCompute 有非常龐大的下游依賴體系,沒有 TCompute,這些下游體系將會罷工

- Z-AI:AI 系統中的 GPU 驅動是執行緒綁定的,DNN-Thread 技術是在 GPU 和 CPU 各開一個執行緒,並且讓其發生綁定關係,例如在識別時會傳遞一張圖給 gpu,這是調用執行緒 api,再才是在執行緒中執行 cuda copy,最後 caller gpu 用 dnn 庫做平行計算和取結果,整個 IO 過程都是執行緒化工作,而主執行緒走識別 IO 的流程大都屬於 demo 演示 api 和機制時才會使用.正規 gpu 程式執行緒技術都是成規模來使用.
- AI 工具鏈:由於 AI 涉及大資料領域,所有的資料操作,不太可能是瞬間完成,因此都是開執行緒,其流程為,鎖 UI,運行執行緒,UI 開鎖.最典型的應用就是 AI_Model_Builder,經常會出現一個細微性級數據操作耗時好幾分鐘.
- ZDB1:這是一個古老的鏈條式資料庫體系,在 2017 年,做過一次大規模升級:將 ZDB1 的查詢流程封裝成了 pipe,然後用主執行緒 synchronize 機制做查詢調度.這裡提示一下,在使用 zdb1 時,只要在主執行緒 Check_Soft_Thread_Synchronize (100),查詢速度就會提升至少一倍.次主線 soft synchronize 機制效率會優於 rtl 庫 CheckSynchronize.然而 ZDB1 的問題也是很明顯的:2017 年對於執行緒的深度度掌握不夠,導致有形成體系的方案,只能算曇花一現,大都作為資料庫伴侶,檔打包,安裝程式等等小功能來使用.無法縱深!
- ZDB2:可以縱深技術體系,未來是立體形式,整個方案作為資料應用模型前後貫穿了 3 年在設計完善,光是築基,就經歷了 3 代應用體系,在目前被高級泛型結構+先進執行緒技術加持後,發展路線可基本明確:大資料地基支援技術體系,在 ZDB2 體系,一個資料庫,可以由 10 張陣列盤,或則 10 個陣列系統共同負載運行,而驅動這些大陣列的所使用的技術方案就是執行緒,每一個資料庫檔 api 都在一個獨立執行緒中工作.ZDB2 在運行中,內部可以從數十到數百執行緒不等,即使在 10TB 規模的小資料庫 IO 執行緒也可以吃掉>100GB 記憶體+>20 核的 cpu,大家不要用陣列系統的思維去看待 ZDB2,資料引擎系統和檔無法直接比較.
- 並行程式:並行程式是現代化流程的必須具備的技術方案,然而並行程式也有非常明顯的問題,fpc/d 所使用的並行支援函式庫並不理想:無法支持指定的相關性核心和超執行緒,甚至無法指定每次啟動平行線程數,而最大的問題還是庫的相容和並行細微性模型,例如 for 並行細微性可以分塊並行也可以折疊並行,這些地方都必須統一起來才能堆大,否則並行程式只能作為功能點來解決局部加速問題,不能作為現代化技術的組成部分,因為這無法堆大.內核並行技術在空間+時間複雜度+機理能優於 D 系+LCL 系.
- 第六代監控體系:第六代監控的 AI 伺服器端,一台准 gpu 伺服器+一個 AI 伺服器應用程式,可以帶 80 路 4k 視頻,而每秒資料量計算公式為: $3840 \times 2160 \times 4 \times 25 \times 80$ =大約每秒需要處理的資料量為 60G.這種變態的資料規模,需要會對拓撲+交換機+網路+執行緒+NUMA+CPU+GPU 整個架構有非常深入的把握和實踐才能做出流程方案.並且這種流程無法做到一步到位,需要先从局部的單獨環節,挨個類比驗證,挨個解決瓶頸和優化,然後才能組合流程.做成這件事的結果是算力成本直接下將 5-10 倍,這是從地獄到天堂.
- 全伺服器體系:伺服器主迴圈和執行緒是千絲萬縷的關係,主迴圈一旦遇到帶計算量的流程卡 30 秒會是家常便飯,人會一直守著伺服器運行,觀察狀態,後面就是優化工作,可以這樣來說,原生的 TThread 缺乏堆砌和調度機制,只有規範起來才可以滿足執行緒間的互調互等,TCompute 是從電腦去機理挖掘出的執行緒模型,可以真正將執行緒系統堆大並把 d/fpc 規範統一起來.

ZNet 的內核技術:結構體系簡單介紹

這裡的結構並不是電腦科學中的資料結構,而是演算法應用的基礎結構體系。

演算法應用是一種計算工程,這將需要統一化,標準化,可探索,有規律性,有社會性,結構體系是計算工程資料來源頭,是設計演算法的前置設計.因為演算法程式會需要人類付出時間代價,這些時間一旦達到某個度,演算法方案也許就不太理想了.結構體系好比是一種把握 solve 的準備工作,沒有這種準備,演算法程式會變得舉步維艱,任何想法的變現都需要時間,甚至很多時間.

以 ZDB2 為例,ZDB2 的核心工作只負責資料 IO,這些 IO 都是在非常複雜的調度下工作,這些工作的目的,就是給演算法提供原始資料,演算法再把資料轉換成資料來源,然後,才是計算流程.好比從 ZDB2 載入 10 億條資料,用演算法做給 10 億某個 key 做一個加速,實現這件事情的流程是用泛結構,例如 TCritical_Big_Hash_Pair_Pool,因為 ZDB2 的 IO 全是執行緒化的,需要帶鎖的 hash 泛結構,這時候,在 IO 裡面往 hash 結構堆資料指標就能完成一個最簡單的 10 億量的毫秒級查詢了.待深入以後,就是在泛結構擴展,刪除,修改,統計等等功能,而這一切,都是用流程來操作結構.當這件事被 solve,就完成專用資料引擎了,這和通用 DB 引擎不同,專用引擎是上天下地無所不能,只要願意,可以拿 GPU 跑 sort+sum,而計算速度,性能,我想通用 DB 引擎只能望其項背.

以 ZNet 為例,在 ZNet 框架中,有大量的佇列機制,例如 progress 裡面遍歷 TPeerIO 是先把所有的 IO 指標都 copy 到一個容器,然後再遍歷容器,如果發現 cpu 時間消耗過大,就退出來,形成分片處理機制,待下次 progress 會繼續處理這個 IO 容器,當容器全部處理完成再重新 copy 容器指標,並開啟下一次的分片.另一方面,ZNet 的發送,接收,機制,也是用的容器,並不是往一個 stream 裡面無腦 copy.當佇列容器這種結構被大量使用以後,如果用 TList 機制處理佇列就很廢了,每次抽取首佇列,都要經歷一些 copy,相信 pas 圈很多人都埋怨過 TList,但 TList 連貫的 1D 資料空間,一旦發生佇列抽取和刪除,優化它就只能改指標,否則就是 copy,這種效率非常的蛋疼,最後,TList 的長度限制也是基本無解的.TList 的最大優點是使用簡單,在 UI 這類小資料規模專案,沒問題,放到伺服器還是算了把..... ZNet 使用的資料容器都是鏈結構的,這種鏈結構是由小塊記憶體通過 next 指標串聯起來,非常適合佇列抽取需求,在內核庫多以 TBigList, TOrderStruct 來命名,試想一下,粘包流程,1 個包 1k,10M 資料就是 10000 個佇列包,在高流量下,TList 計算能力和 TBigList 無法相比,這一差距,用測試程式跑出來會是 2000 倍.

以 Learn 統計學的方法為例,結構體系可以當成是 IO 語言,就是輸入一種結構,輸出又是一種結構.例如分類演算法,這些演算法原理基本上可以算初中生作業,先用某些計算方法,或則隨機方法,生成一些 seed 形式的資料,然後再圍繞 seed 做搜索和匹配,所有的分類演算法,基本都是這種思路,變來變去,或許這會天馬行空,但流程思路往往很簡單!而資料接入和輸出,這才是最麻煩的計算,在 600585 看來專案中的完整分類流程,前置處理 45%,計算 10%,後置處理 45%,總結一下:結構體系在統計流程中的占比會大於演算法.大家平時看到調用一個 api,得到了 solve,流程直接封裝,包幾十個結構+類也很正常,這已經不是統計演算法了,是解決方案,內部走的是 1,2,3,4,5 序列步驟來解決問題.人臉識別,推薦演算法就是這種模型.

以執行緒為例,執行緒是個很大的體系,現代化程式無所不用執行緒,而執行緒和執行緒間的通訊,憑空寫出來的結構是無法堆起來做項目的,執行緒互調互等,TThreadPost,TSoft_Synchronize_Tool,這些泛結構在使用時基本都能 1-3 行解決互調互等,極簡使用,要不然怎麼能堆大?

ZNet 體系中使用頻率最高的大結構體: TBigList<>, TBig_Hash_Pair_Pool<>

ZNet 體系常用小結構體: TOrderStruct<>, TAtom<>, TPair<>

ZNet 的泛結構常用 Hash 庫 Z.HashList.Templat

ZNet 經典鏈表庫 Z.ListEngine

ZNet 的內核技術:簡單說下結構組合拳

以 SVM 和 K-Cluster 為例,SVM 可以算比較代表性的 nonlinear 演算法,KC 則比較偏向 linear,這兩者在計算流程上基本可以算一樣:都是先輸入,再做輸入預處理,得到運算元資料,再算出最終的分類結果.SVM 推導思路走維度切換,維度部分是整個 SVM 的核心思路,集中在遍歷單維度數據方程上,通過遍歷得到最優平面可切分的單維點,再把幾個單維組合起來就是超平面切分點,然後在超平面從最大到最小按跨度切開,再用維度算聚類,思路流程上是走的編碼解碼路線,這一步可以有一堆優化措施,如果是不走優化的 svm 會是很簡單的流程思路,而 svm 可以寫出幾千行,也可以幾十行解決,因為有維度跨度,這些跨度可以充當一種記憶條件,也就是訓練建模,使用模型這套流程,svm 也可以做的非常簡單,pas 系也可用幾十行來做出 svm 自動分類,前提是必須有結構體支援,以前我參考的經典做法主要來自 shogun(c++古典派 svm 的起源項目),大如感興趣可以自己找來研究.K-Mean 推導則是亂數產生質心分類,臨近的全部聚類,完成後再定義出新質心重新走流程,反復反覆運算幾次就完成聚類了.在結構層面,只要定義出,輸入輸出,剩下來的基本可以直接交給開源的各種計算庫去幹.大家平時在網上查找資料,各種複雜公式這是一種思路上的表達語言,演算法流程思路尤其主體部分都不太複雜.比較難理解的是非線領域,這一領域如果把自創的東西算進來,可以有上百種演算法,而非線的核心思路上是給資料做解碼編碼預處理.

以高速範圍搜索為例,這一領域還沒有來得及編寫 demo,它的目標是解決範圍內的快速搜索,例如,資料量到 10 億,並且資料隨時在增刪,要搜索時間範圍和座標範圍,如果不給演算法暴力遍歷,也許一個流程會走好幾分鐘.比較有效的做法是把時間和座標範圍按度量切開,例如磁碟陣列的檔座標,可以按每 1M 切出一個區域用於 hash,處理範圍時以 1M 作為一個小小跨度來記憶,時間切分同理,可以按分切,也可以按時切,切分以後只需要錄入一次就可以精確定位了.ZNet 的做法是緩存指針,hash 跨度, 參照庫為 Z.HashMinutes.Templet,實現和組合時主要使用 TBig_Hash_Pair_Pool<>+TBigList<>.其中時間範圍加速演算法,主要用於搜索監控片段,基本上全都可以秒搜,座標範圍加速演算法,主要解決模擬寫緩存,這是模擬寫入檔並且保證讀寫一致性的功能,需要在高速讀寫環境,例如寫入 100 長度座標在 1024 位置,這時候需要找到緩存 1024 位置一系列的 part buffer,這樣才能完成檔讀寫資料一致性.

以雙向配演算法對為例: bidirectional,每完成一個配對,會遍歷全部目標,如果不考慮優化,配對 1000 比 1000,計算量為千萬,當每完成一次配對,剔除掉已配對的資料,計算量將會小很多,再配合執行緒,並行這些手段,雙向配對可以做到非常快.而解決刪除配對,用 TList 搞不定的,TList 它會重構 array buffer,非常耗 cpu,必須使用 TBigList<>.

以並行排序為例:我無法知道地球上最快的平行算法,我的方法是先分開存再排,例如 1-10, 11-20 各自分成獨立塊,然後再用並行排細微性塊,最後排整塊+構建輸出.結構體就是使用 TBigList<>,只有 TBigList 才能支援 10 億這類大數量,直接用記憶體指標會把排序搞非常複雜.

ZDB2 如何解決 Stream 防寫狀態下的模擬讀寫

用過 VMWare 的都接觸過磁片防寫:在寫磁片時 VMWare 會將資料寫入到一個暫存檔案,而原始資料並不會更改,同時在讀操作時寫入資料會一致化(原始資料與就近寫入資料相分離).底層模擬庫 Z.FragmentBuffer.pas 從正面解決了上述問題,下面來詳細說說模擬寫入的解決方案.

首先不管資料規模多大,磁片是單維空間,資料都按座標放在裡面,並且寫入的資料永遠都是一小段,在模擬寫入演算法中把這一小段資料定義成 Part.在 Stream 中的日常寫入方法大都為,從 stream.pos=xx 到 stream.write(xx)的迴圈行為.stream.write 的長度等同於 Part 的長度.當讀取時,會先讀入原始資料,再用 part 裡面的資料做覆蓋.這樣就實現了模擬讀寫的一致化.我在解決這一流程中專門開闢了一個 Test case 保證一致化處理流程是正確的.這類 Test Case 工作還包括了對同一 Part 的反復 stream.write(我將它定義成 Part update 機制,例如 Stream 從 100 位置有時寫入 1k,有時候又寫入 100k,Part Update 機制完全隨機不確定),以及多個 Part 可能發生合併情況的處理機制,處理這些繁瑣的底層機制需要做到一步到位,不留尾巴.

當解決一致化以後接下來是優化計算,因為當 Part 資料達到一定規模,例如 100 萬個,讀寫時會從 Part 資料裡面找出對應的資料做讀寫,如何快速找到 Part 是一個非常重要功能.我考慮過使用 B 樹方法,但 B 只對靜態資料友好,後來,我分析了一下尋找 Part 可能運行的代碼量(大致的 cpu 計算開銷),得到了一個相對比較準確的結果:當 Part 資料達到 100 萬個,並且精確定位到目標的平均時間開銷大約 10-50ms,現在,已經可以定位到那個尋找 Part 的迴圈程式瓶頸.

這時候,使用跨度 hash 表來減少尋找 Part 的計算開銷,跨度 hash 大致思路是按 1024Kb 做為一個跨度,例如當 stream.pos=1536kb 並且寫入 1024kb,那麼就是位於跨度 1024kb-2048kb 區間,這時候迴圈瓶頸就從 100 萬次減少了 2-10 次左右,優化提升大約在 10 萬倍,cpu 開銷也從 50ms 下降到了 0ms.現在模擬寫入已經具備了實用性.

接下來就是把 Stream 的模擬 IO 接入到 ZDB2,因為 ZDB2 一旦接入就可以做到資料安全了,因為每次寫入都會在記憶體中模擬,並不會直接寫磁片 IO,這可以不用擔心斷電資料丟失損壞.工作機制為,替代原 IO 的全部讀寫, 當 flush 時,先建立一個暫存檔案,把 part 先寫入暫存檔案,然後 api 用 FlushFileBuffers 確保物理寫入,然後再寫入 ZDB2,當成功以後,刪除暫存檔案,如果寫入過程中斷電,ZDB2 會從暫存檔案恢復.

在實際應用過程中,我發現 hdd 走陣列路線非常慢,資料量一旦>10GB 到 flush 會耗時 3-5 分鐘,而 ZDB2 單庫的日常空間使用經常也會從 500G-2TB,這裡又是另一層程式主結構上的優化了,未來建議使用模擬寫入的硬體設備最好走全閃的配置路線.雖然全閃空間不大,這在存儲效率上會大幅提升,剩下很多很多優化工作.

另一方面,Z.FragmentBuffer.pas 庫的命名與它的功能定位是匹配的,本身它並不會傻瓜化的接管 Stream,而是作為單維 Buffer 的處理計算,所以使用 Fragment+buffer 來命名.

By.qq600585

回顧:設計泛結構 TBigList<>

TBigList 並不是一蹴而就設計到位.這要回顧一下歷史內容

1. 2015 年 TBigList 的最初前身是位於 Z.ListEngine 庫的 THashList,這是作者編寫的第一個 Hash 庫,內部大量使用 TList 做轉換保存功能使用
2. 2016-2017 左右,THashList 出現了序列還原需求,已無法追憶還原什麼序列,總之,就是 add 123 以後,就應該可以直接從 THashList 還原 123 順序,這種需求,導致了 THashList 內部結構從單一化到鏈條化的轉變.
3. 2020 年 TOrderStruct 出現了,當時決定將全系代碼從古典結構逐步移植成泛結構.
4. 2020 年 TBigList 被編寫出來,TBigList 綱出爐時,由於 test case 寫的很到位,順帶也修復了 THashList 頑固百年漏洞,這是非常大重大的修復.之後,TPair<>, TBig_Hash_Pair_Pool<>, 被相續編寫出來
5. 2021 年 ZNet 全部代碼一律刪除 TList,使用 TBigList<>泛結構替代.
6. 2021 年某天無聊,編寫了一個 BigList pk TList 的小 demo,TList 全部使用最優刪除,追加,修改的方法與 BigList 進行了一次性能 pk,結果是,BigList 處理能力幾乎為 TList 的 2000 倍

TBigList 的設計思路:首先要解決高速佇列+Int64 級數據量+可以在大規模迴圈代碼種的堆砌程式設計:必須解決 for 這種迴圈需求.其中,解決 for 迴圈需求甚至高於對性能的要求,簡單來說就是 for 必須是一種很簡單流程模型,不可以用匿名函數.

最終,TBigList 才被設計成了今天的通用泛結構,在此基礎上,後面,ZDB2,各種新演算法,新結構體系,應運而生,這些結構和演算法太多太多了,Z-AI 體系直接都不用提及了.

ZDB2 被編寫出來,其實就已宣告自主技術進入大資料時代了.後面是用時間來反覆運算.

回顧:設計腳本引擎 ZExpression

很早很早以前,編譯器一直是作者心裡遺憾,當初的想法:理論懂一堆,如果不能動手寫一次,理論就會是空談,這件事必須有切身體會才行,吃喝玩樂過日子毫無意義(今天的感受是沒錢的日子才是毫無意義).

- 字串解析是面對的第一個問題,字串解析程式非常複雜,最開始是做分字+分詞函數,然後,開始嘗試做符號解析,逆波蘭,加減乘除,把各種字元轉換成原型結構.最終,解決了詞法轉換.
- 第二步,開始嘗試解決詞法語義結構,諸如,詞法合法性,這裡的詞法結構是樹型的.因為 $1+(1-2)$ 的詞法結構等同於 $1+a$,而 a 結構是 $(1-2)$.
- 第三步,開始嘗試把詞法結構翻譯成可執行程式的 opcode 碼,這一機制,用的是模擬 opcode 碼來實現,機制上與 x86 解碼工作方式是相同的,差異是對碼表不同.
- 第四步,開始大幅度加強 Parsing 支持環節,無限逼近 bison+yacc,技術細節這裡省略一下,放在下一節來說.
- 第五步,開始應用 ZExpression 體系:C4 啟動,ZAI 腳本,6 代腳本,pascal 代碼重寫模型,這些都包含在 ZExpression 體系.

ZNet 的母體移植技術:Z.Parsing

ZNet 是作為母體+載體的巨型代碼項目,這些代碼其實都是用機器編譯技術解析+重構生成而來,大家平時使用 prp 移植 ZS,或則 prp 升級 ZNet,裡面是一個資料模型+解析重構技術在工作.

編譯器中的解析+重構技術,它的思路借鑒了 bison+flex/lex+yacc 體系.

簡單舉個例子

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

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

上面是兩種完全不同的詞法體,在 bison/yacc 體系是用代號運算式,來描述這些詞法體,也就是指令碼語言,然後再用編碼流程讓他們形成統一化能表達流程的結構資料.

在 Z.Parsing 體系中,有一種機制可以無招勝有招,探頭技術

探頭技術是建立在詞性的前置工作上,Parsing 體系會先將,數位,符號,浮點,字串,備註,ASCII,等等,他們將會被詞性劃分,形成詞性鏈結構.

例如 if(1+1=2) kill,它的詞性鏈為:ascii,symbol,num,symbol,num,symbol....

探頭技術,是對詞性鏈做近似判斷,然後進入分支流程.

探頭技術可以區分,不同的詞性結構組合+詞法體,它會形成條件範式,這些條件範式,正是接監的 bison/yacc 的設計思路.

當詞法程式具備了條件檢測以後,就可以用螞蟻一邊爬一邊用探頭條件範式開分支程式處理隨機性極大的手寫詞法體了.

也正是有了強大的探頭機制,pascal 重寫模型技術才能正確且完整的解析 pas 代碼並且重構的自己想要的目標代碼了.

在另一個方向,國際化的備註和字串機器翻譯技術,也是使用的 Z.Parsing 體系,而字串翻譯非常簡單暴力,就是找詞性為字串和備註的資料結構翻譯,然後重構成不同的語言.國際化項目大家有空可以去看看

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

Z.Parsing 的探頭是靈活自由的,指哪就能打哪,在螞蟻程式中可以做出 bison/yacc 無法實現的東西.這時候,Z.Parsing 是統治現金流的技術體系:只要使用 Z.Parsing 那個人能正確編寫出自己的語言,並且他能讓集體一致性的使用他的預言,那麼他對軟體,遊戲,各種程式,將成為教父一般的人物,這將會讓他擁有高於任何投資人,總經理,董事長的絕對技術話語權.如果公司估值 1000 萬,使用 Z.Parsing 做出公司生產系統那個人,他身價佔據一半我也一點不會驚訝.因為公司依賴產品,產品依賴集體的生產,生產核心技術依賴 Z.Parsing 用戶.

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

從交換機到拓撲簡單說說 XNAT

先挖最底層的 IP 包,IP 包構建在驅動程式的以外資料幀基礎上(Ethernet),這一層的概念是建立在電頻協調,協商速度,電纜規格這些硬體基礎上,很多時候這很籠統,有人喜歡說乙太層,也有人喜歡說物理鏈路(軟體鏈路又是什麼呢?),總之,不被概念綁架:IP 包是構建在物理通訊層上面的協定結構,而協定結構又分了很多很多種類.

IP 上面,會有 Ipv4-tcp,Ipv6-tcp,Ipv4-UDP,Ipv6-udp,icmp,arp,組合起來跑拓撲.例如網卡通電,入網,這是個 icmp+arp,又例如 tcp 發一個包,送達出去,可能不會發生什麼事情,但如果中間有 wifi 或則目的地是遙遠的某個節點,可能會收到一個 icmp,這表示 tcp 報文不可送達,通常來說,我們在網卡驅動裡面看到很多很多應用協議,大都走的就是 icmp.

理解 ip 層,只需要從 p2p 這種點對點通訊入手就可以了,ip 是在點對點基礎上,擴展而出的拓撲網路通訊協定,ip 資料永遠都在網路裡面轉來轉去,各種協定和硬體,路由,交換機,NAT,防火牆,就是控制這些 ip 包的流轉.

每一個 TCP 包,正常情況下,只要未到達第一個路由節點,就是閘道節點,那麼就會有個 ICMP 發回來,告訴你這個 TCP 包不可到達,你需要重發.UDP 包則沒有這種機制.我經常從國內向國外伺服器上傳大資料,單工 TCP 的速度,基本會達到限速級,這是並不是因為線路很快,而是第一個路由閘道節點在主要消化我發送的 TCP.

回到 XNAT,ZNet 自帶的 XNAT 底層走的是 CompleteBuffer,這是一種程式的模型,它是整個 Znet 最接近 TCP 原生資料包的協定,主要用於快速發送資料.至於速度會有多塊,這取決於鏈路和閘道,而鏈路和閘道是決定了 TCP 的關鍵,Znet 走的是非同步通訊模型,在非同步模型所有的資料都是佇列化處理,卡佇列這種情況,包在 IOCP/EPOLL 程式裡面很難感覺+發現,但對於鏈路和閘道,只要收到了 icmp 報文:這個 tcp 不可到達,那麼 IOCP 的 send 佇列流程機制就會暫停,這在併發程式模型裡面是看不出來的.

CompleteBuffer 會出現萬兆對傳千兆,或則對傳百兆:Znet 非常重視緩存管理,萬兆傳千兆並沒有自動化的處理機制,而是預設全部讓資料排隊,堆在記憶體裡面讓硬體慢慢消化.在通訊程式層面,Znet 是要求:如果連續發送 completebuffer,當達到某個度,就發一條 Send_Null 命令,這是因為 Znet 在底層對大量 completebuffer 的資料做了自動化暫存功能(用硬碟保存),但如果不加 send_null,那麼 completebuffer 是無法暫存的,而會全部一併扔給通訊介面,例如 crosssocket,diocp,ics,synapse,這些介面是把資料無腦堆記憶體裡面,當伺服器萬兆傳 100G 給千兆會按時間流逝發生崩潰.這也是很多人使用 XNAT 傳 FTP 大檔,記憶體爆炸的原因.但是,在通訊程式裡面,加了 send_null,那麼佇列中的所有 completebuffer 就會暫存.因為 send_null 會讓佇列等待一個回饋,然後才會繼續後面的佇列,在這過程中,completebuffer 的資料保存在硬碟中待發.如果硬碟夠大,萬兆傳千兆,100G 是可以被成功消化掉的.

說說 FRP,這是阻塞協議,阻塞做對穿,對等到達,萬兆傳千兆時,萬兆會降成千兆來傳,同時也不消耗記憶體和磁片緩存空間.但非同步不同,非同步是全堆佇列裡面處理!!

在拓撲網路中,往往一個應用出口就是一台機器,但這台機器連接到網路數百台伺服器,當 app 訪問伺服器時,出口往往只是流量代理,真正工作的卻是網內的數百台伺服器.這個就是伺服器叢集.在現代化基建中,群集要發揮作用,NAT,正反代理,埠映射等等,都是經常會用到的技術.而近期的 Znet 新版本針對群集問題,給出了 XNAT 的大規模代理方案:100 台伺服器沒關係,用 100 個埠映射.

這一小節,簡單說了一下我在 IDC 的應用思路,如有不周,大家請多多包涵.

性能分析:當 ZDB2 的資料規模達到 1 億條

ZDB2 具備(Rolling)滾動存儲能力,當資料規模達到臨界點將會刪頭加尾,而刪頭加尾可以外部程式調度也可以內部自訂調度.

當資料規模達到 1 億條,磁片佔用空間大約 100G-20T.這時候 rolling 有兩種工作方式

1,在 TZDB2_Th_Engine_Marshal 使用 Remove_First_Data_For_All_Th_Engine,這種方法主要提供外部調度程式刪頭(移除最早的資料,一般會在執行緒中調用它),這會有調用延遲,這些延遲 90%都來自於記憶體釋放(destroy),而從磁片移除數據幾乎沒有延遲(1:1000).達到 1 億條後,每次刪頭規模,多則千萬,少則也是上百萬,這些延遲也許高達 5 分鐘,在 5 分鐘內,所有的資料被 busy 機制卡住暫存於記憶體,而在流程環節,ZDB2 具有防卡機制,append 資料會立即返回,只有當流量大到 5 分鐘的資料量,系統記憶體裝不下才會崩潰.

2,內置方式刪頭,內置刪頭是在每次追加資料(append)即時進行的,這種方式並不是流程式刪頭(不等 destroy),而是開了一個子執行緒讓它在偵測是否達到刪頭條件來刪頭.這樣幹的目是減少 append 延遲,因為主執行緒的延遲會傳導給別的程式模組,例如伺服器既包含 ZDB2 體系也包含了 web 服務,埠代理服務,高流量延遲會導致某些服務嚴重受阻,好比埠代理正在即時直播,延遲將會導致直播卡頓.

當資料量達到 1 億條的 ZDB2 崩潰模型

ZDB2 走的讀寫流水線,所有的讀寫資料永遠都在排隊處理, Remove_First_Data_For_All_Th_Engine,會在佇列插入上百萬條刪除請求,當系統無法即時消化,那麼寫資料佇列消耗的記憶體將會無限增加,直至系統崩潰.

當 hdd 陣列忙,讀寫資料將會出現延遲,佇列無法被消化時記憶體暴增,直至系統崩潰.這種操作多出現於省錢買小記憶體,hdd 陣列規模過小,以及在資料庫工作中,陣列出現三方讀寫操作,例如大規模 copy 檔.比較適合 ZDB2 體系的陣列是至少 6-10 盤的 hdd,並且記憶體>預計空間的 2.5%,例如計畫存儲 10TB 資料,記憶體不低於 250G.另外,sdd 和集成化的 m2,nvme 對記憶體的需求要寬裕很多,記憶體只需要達到 1%,計畫存儲 10TB 資料,記憶體達到 100G 就可以正常工作.

當資料量達到 1 億條,zdb2-flush 如何工作

Flush 操作是將記憶體中的暫存資料直接寫入磁片,ZDB2 使用分塊演算法寫盤,當資料規模 1 億條,單庫檔的分塊寫盤規模將會高達 500M-2000M 不等, TZDB2_Th_Engine_Marshal 是多庫集成化的體系,往往一個 flush,會引發 100 個單庫的寫盤操作,單次 flush 的 IO 寫入規模甚至會達到 200G.這時候,hdd 陣列會達到瓶頸並且進入高延遲模式,而高延遲將會引發資料暫存模型無限增加記憶體.

ZDB2 在 idc 經歷了數次宕機後,flush 演算法被修改成了差分化分塊寫盤模型,差分寫盤會先對資料塊按座標序列排序(sort position),如果發現分塊過大(高於 100M),將會拆分成 100 個 1M 的小塊,這時候,資料對比方法,檢查資料是否修改,然後,再向 IO 進行寫盤操作.最終的 flush 消耗從單次 200G 下降到 20G,這可以在半分鐘內完成一次 flush 操作.

本節中的 Flush 包含了 IO 防止斷電丟資料的讀寫分離機制.

當資料量達到 1 億條所有的資料庫模型都將變得不大實用,大部分系統方案會選擇開關分散式,用拆分資料庫這些手段來解決大資料問題,ZDB2 也是走的拆分數據方法,只是 ZDB2 使用 TZDB2_Th_Engine_Marshal 直接用集成化方法將子資料自動化管理起來了.1 億條規模下的真實專案,不光是陣列,還會涉及到拆分網路服務,例如把存儲服務和應用伺服器分離開來.而作者是集成路線:web,zdb2,rtsp,hls,monitor 全部集成在一個服務裡面來搞,思路則是開分載執行緒.

By.qq600585

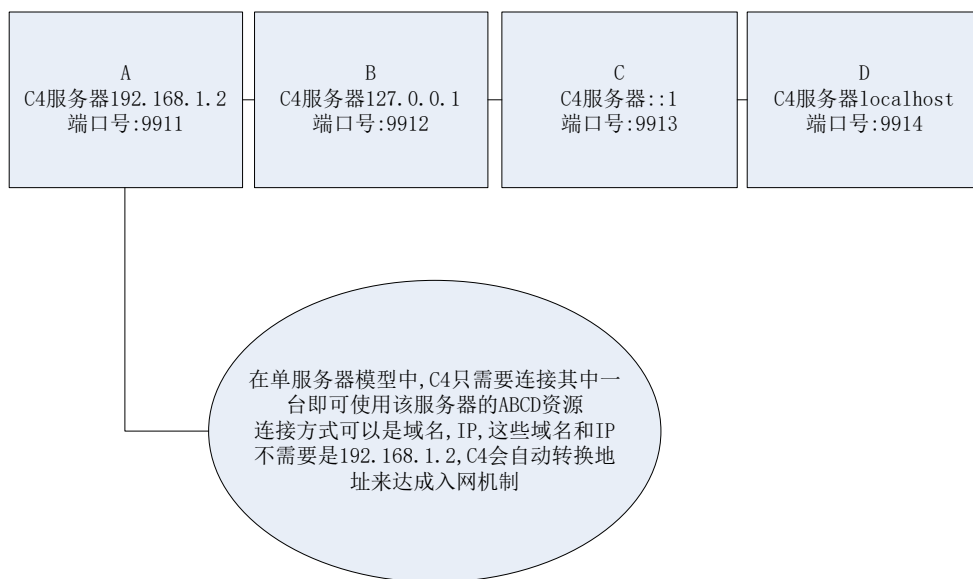
再說 C4 入網機制

C4 將伺服器統一化,一個完整項目可以有若干機櫃(每個機櫃 5-8 台),將這些伺服器協調統一就是 C4 的工作.

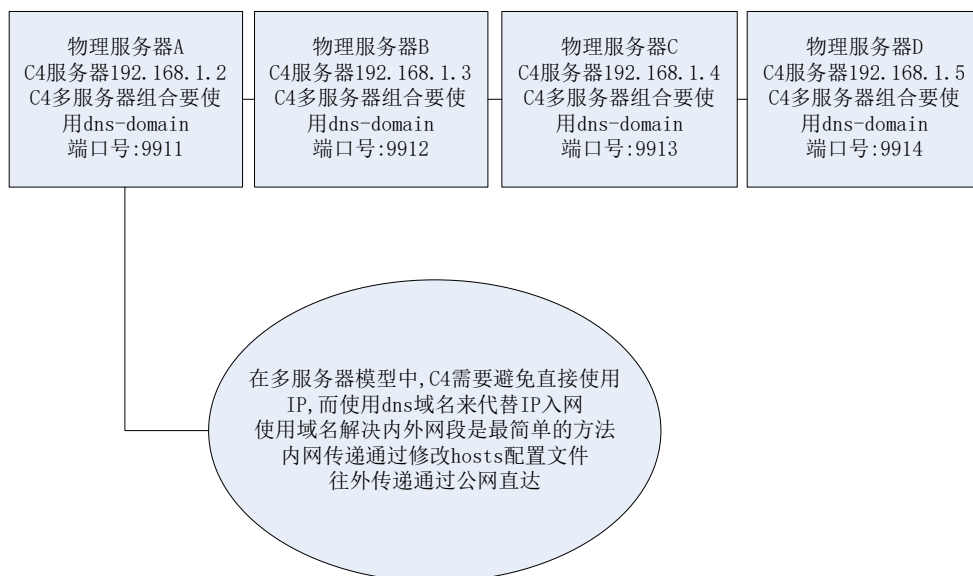
另一方面,最常用的伺服器模型是一台高性能伺服器,在單台伺服器模型開一堆服務進程,這些服務可以是檔服務,資料服務,登錄服務,內網穿透服務,在真實的專案中,這類服務會開很多,這是集成式的.導致伺服器群會有很多很多 IP 位址,功能變數名稱,埠.而使用 IP,DNS 就是一種 C4 的入網機制:C4 入網的本質是建立連接.

C4 位址轉換機制:當 C4 連接目標網路,例如 123.net,而目標網路的上報位址 192.168.1.123,這時候,192.168.1.123 位址將被轉換成 123.net,該方式可以滿足常規單台 CS 組網模型:堆服務端並編寫前端介面,然後,借助 C4 使用海量的伺服器計算資源.

在單伺服器中的 C4 入網機制:在單台伺服器位址會自動轉換,當使用 127.0.0.1+::1+192.168.1.2 這類不同的位址來表達入口時 C4 都可以自動轉換出來並且入網.注意:單伺服器如果插 10 張網卡+10ip 這時候要用 127.0.0.1 環回位址來代替物理位址 192.168.1.x->127.0.0.1 這樣來幹 C4 才能自動完成位址轉換.



伺服器群 C4 入網機制:使用功能變數名稱來代替 IP,這樣來解決內外網段各走自己的線路!這是最簡單 C4 並網.



當 C4 入網後需要幹什麼事情

簡單來說:C4 入網後在當前進程中就會多出許多可以直接使用的伺服器資源。

C4 規則:任何 C4 伺服器都是 Service+Client 的結合體,例如在 C4 中編寫一個 MyService,那麼就必須編寫所對應的服務+用戶端,這是 C4 規格,而當部署完 C4 後,MyService 就是一項計算資源。

當系統集成時 C4 的成功入網代表整個伺服器群進入工作狀態:在系統集成中,各項伺服器資源會非常多,例如 2 個機櫃,每個機櫃 5 台伺服器,每台伺服器開一大堆服務,例如 web,檔,hyperv,kvm,視頻,圖片等等,這些都是把後臺做大的一種堆砌模式(無限堆砌伺服器),C4 對於堆大是一種自動化的模型,因為靠人工堆砌伺服器群這是不現實的.因為系統集成是一項大工程,我們可以設頂它的複雜度為 n5,而人類的極限梳理能力為 n3,當解決方案的適應複雜度無法與真實 solve 複雜度相匹配時,會出現難以修改,維護,升級,更新的情況,當 C4 入網後,系統集成的 n5 難度會下降到 n3 以下,也許我這樣說很多人不理解,當物理伺服器>5 台的專案,C4 規則會收穫先苦後甜的結果。

ZNet 內置 CPM 可以代替 FRP+Nginx

很多 CS 專案使用 FRP+Nginx 來統一化穿透多台伺服器,在 ZNet 中只需要在 C4 引進 CPM 就可以解決穿透問題。

CPM = Cluster Port Mapping,群集埠映射技術

CPM 在 C4 體系可以 10 行內解決穿透問題,並且這是可以程式設計的大規模穿透機制,例如多機房部署,多伺服器部署,穿透往往是 ip+埠資料配置形式,但在 CPM 系體,穿透可以是程式設計形式,簡單來說,可程式設計形式的穿透可以具有自動化穿網能力,而且 CPM-imp 代碼非常小,很容易修改定義。

舉個例子,項目含有,vnc,remote desktop,sql,my server,vpn,smb 很多很多協定,並且這些協定和服務分散在群集中的各個伺服器,這時候古典的做法是配置一台主要伺服器,再配置群集各個子伺服器.對 CPM 來說整個群集只需要部署 my server,不需要管配置表,通常 CPM 可以支援到上千個埠.如果需要,甚至可以把終端直接 CPM 到公網.另一方面,CPM 也是極簡化的可程式設計映射,只要能 ZNet 的環境都可以 CPM。

C4 主要用於大項目嗎?

使用 C4 主要走反覆運算形式路線,例如 C4 早期的伺服器都是 ZDB2 的第一代體系,隨著 AI 項目的推進,第一代體系會慢慢淡出,並且開始逐步使用 ZDB2 的第三代體系來重新複現,使用者登錄系統,檔案系統,Key-Value 資料庫系統.隨著三代的推進,第一代體系並不會被 replace,而是保留在 C4 作為老專案支持。

伺服器型的項目在 C4 中是走的反覆運算路線,每次有反覆運算就開個新服務,因為 C4 是個體系,往往一個服務會依賴大堆小服務,這些大小服務往往版本都不會統一,例如 RandSeed 服務,這是一個用於在全 C4 網路生成唯一 ID 數位的服務,它可以被第一代 C4+ ZDB2 體系使用,也可以被第三代使用。

C4 並不是設計給大專案使用,而是走更符合伺服器體系反覆運算的路線,伺服器專案不會開發出來就一直兵不動,而一直會保持升級反覆運算,很多時候 CS 形式通訊,一個反覆運算,也許整個 CS 都需要一起升級,這時候直接開個新的 C4 伺服器吧,運營中換一下埠,拿一個月來過度新老版本相容。

專案永遠都在反覆運算,因為伺服器不會一步到位,即使開發到位,在運營過程,也會有各種維護升級。

主迴圈記憶體微洩漏分析

微洩漏極難分析,例如 10 萬行大流程如果發生大記憶體洩漏,可以根據運行狀態來分析,而微洩漏往往出在手誤,這時分析微洩漏,FastMM 這類技術並不好用,這是利用 debug+tracer+report 形式來進行分析,需要把大流程獨立剝離出來慢慢分析,動則一周,多則數月。

微洩漏的典型例子是主迴圈程式,這種程式設計就是讓伺服器永不關機,在主迴圈中出現洩漏,主體記憶體佔用 100G,微洩漏問題大約每 3 天增加 1-3G 左右,這時傳統做法如上所述,流程規模一旦大了只能單獨剝離出來以模擬方式運行找洩漏點。試想一下,載入 100G 的資料來源,再去分析 1-3G 的微洩漏,幹這件事情會非常反人類。

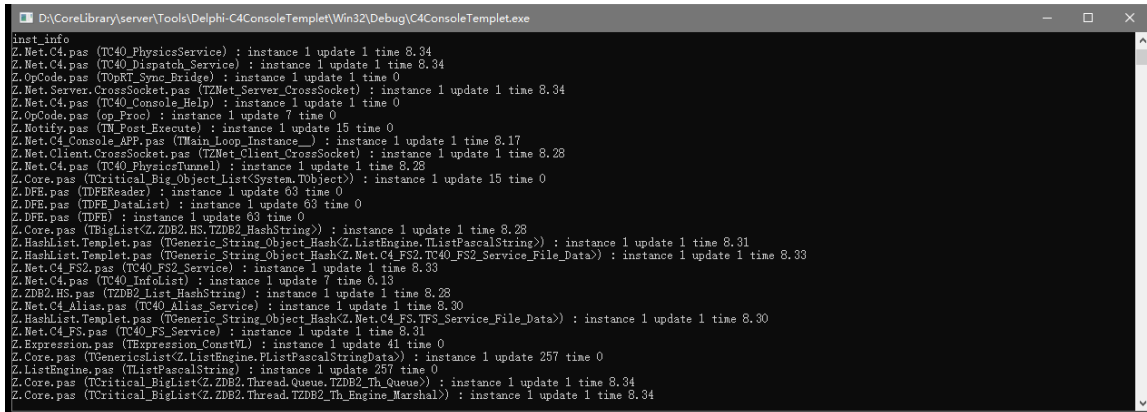
Z 系內核在伺服器微洩漏給出了解決辦法,首先打開編譯定義“Intermediate_Instance_Tool”

```
// 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}
```

當 Intermediate_Instance_Tool 被打開以後,所有基於 TCore_Object_Intermediate 的實例都會進入計數模式,包括泛型結構,每次構建實例,計數器會+1,destroy 則-1。

使用時只需要將 TMyObj=class(tobject)替代成 TMyObj=class(TCore_Object_Intermediate)就行了,一般來說直接使用替換功能就行,Z 系內核提供了 TObject,TinterfacedObject,TPersistent,三種基類。

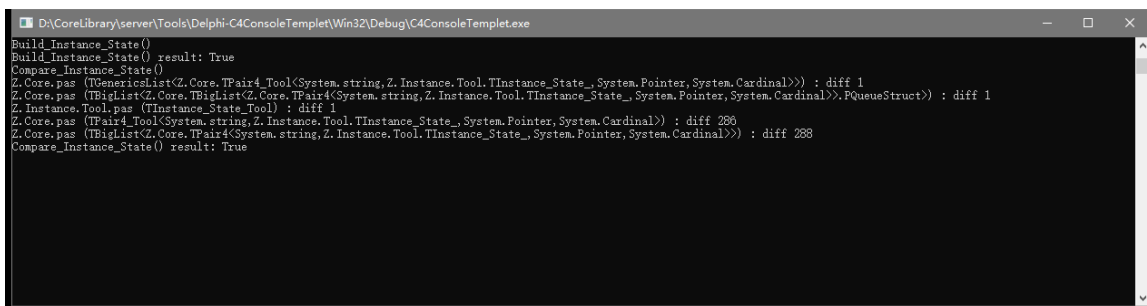
然後,開 C4 框架命令程式,輸入 Inst_Info,得到當前伺服器的全部實例計數,這些資料具備分析微洩漏條件。



更直接的一種做法是,先做一個當前實例狀態副本,讓伺服器運行一段時間,例如 2 小時,然後做一次比對,得到 2 小時間的新增實例狀態,到這一步,幾乎可以定位出問題出的具體位置。

創建實例狀態副本:Build_Instance_State

以當前實例與副本做比對:Compare_Instance_State



Diff 值為差值,286 表示在 2 小時中,某個泛型的增加了 286 個實例。

到這一步,我們已經可以定位出微洩漏的問題所在點了,剩下來就是檢查和該泛型相關的流程進行修復

分析微洩漏不需要重啟伺服器,也不需要動不動開 100GB 的資料模擬,直接在伺服器運行中就能做到,對於運營項目這是極方便的維護方法。

內核庫在啟動時都做了什麼事情

Z.Core 庫是整個 Z 系的中央內核,它和 fpc/delphi 無關,下圖這些變數大多是某些小體系的全域變數+類的初始化

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

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

設置浮點異常篩檢程式,異常是浮點計算程式的一種標準模型,例如某些 cpu 流片的浮點指令錯誤,浮點除 0,空浮點,浮點過大,精度丟失,在 x64 架構編譯器會預設使用 64 位元整數來類比浮點,這時許多異常是來自 x64 類比信號,而許多 x86 架構編譯出來的浮點代碼會使用 fpu,mmx,sse 這類獨立浮點晶片(fpu 通常都集成在 cpu 內部),這時候浮點異常規則則是當 fpu 計算失誤,返回 NaN 這類無效狀態時觸發的,這些計算都會報出異常,並不是 fpu 過程異常,這裡注意區分,fpu 永遠是計算完成後,數值不對才會出現浮點異常,Z 系會把這些異常全部遮罩.當 Z 系遮罩後,如果發生浮點計算錯誤,例如 0 除和除 0,這時候一律返回 NaN 值,這代表一個無效的計算結果.如果浮點程式寫的很大,例如高斯計算,差分計算,金子塔計算,在這類大計算流程出 NaN 會讓人找不到問題所在,這時候,可以將篩檢程式直接遮罩掉,或則人工打開.

On_Raise_Info := nil;

Z 系內核有個 RaiseInfo 的 api,作用是簡化 raise Exception.Create()的書寫規則,RaiseInfo 會給堆疊程式拋出一個異常,而在拋出異常之前,會調用一次 On_Raise_Info 事件.所有 Z 系異常都會通過 RaiseInfo 來拋出.因此截獲該事件就等同於截獲了 Z 系的全部異常事件.在另一方面,DisposeObject 主要用於代替 Obj.Free 方法,當出現異常時,Z 系也會調用一次 On_Raise_Info,然後才會給堆疊拋異常.

Inc_Instance_Num := __Inc_Instance_Num__;

Dec_Instance_Num := __Dec_Instance_Num__;

實例跟蹤技術的回檔事件,這裡的指向是個空函數.這是一個雞和蛋的矛盾先生問題解決辦法:到底是先有雞還是現有單,因為 Z 系全部實例都可以產生被跟蹤的資料,那麼跟蹤程式本身以及跟蹤程式所依賴的更加底層的實例怎麼辦?難道比實例跟蹤技術更加底層的實例都需要被跟蹤嗎,在設計跟蹤技術前,確實是考慮到全體實例一律啟動跟蹤技術,在實際解決中,跟蹤程式套跟蹤程式,這是一個矛盾的功能,而解決辦法就是交給上帝來處理(不確定場景就用事件掛接技術):內核在啟動時,跟蹤程式是空調用,當依賴於內核跟蹤程式被啟動以後,回檔事件才被賦值,這時候跟蹤程式才會發生效果,跟蹤程式內部是一套複雜的資料系統,它會記憶全部實例的創建和銷毀,主要解決大流程記憶體洩露問題.

```
Init_System_Critical_Recycle_Pool());
```

這是執行緒臨界區的回收池,臨界區在 Z 系內核會區分軟臨界和硬臨界,軟臨界使用無限迴圈等原子信號,硬臨界在等原子信號中會觸發 cpu 時間週期,硬臨界在等信號中,從工作管理員看起來就是 cpu=0.臨界回收池是一種對臨界系統控制碼再利用,Z 系的所有軟+硬臨界控制碼都是回收模型,當臨界區被釋放時控制碼並不會消失,同時臨界區控制碼如果是 Acquire 狀態那麼會被 Release 後再回收,簡單來說,Z 系伺服器如果大量使用執行緒臨界,那麼在系統監視器裡面幾乎控制碼開銷會恒定,而工作管理員中會有一個 cpu 核心程式的時間,這種核心程式的時間是表示調用作業系統內核的延遲時間,內核延遲越大等同於作業系統對伺服器的穩定性影響越大,Z 系伺服器和大規模平行算法的內核延遲很小這都取決於大量應用回收池這類機制,參看[鎖複用](#).

```
// 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;
```

主次執行緒分離技術,可參考[雙主執行緒](#)和[同步技術](#)章節,上述的代碼都是初始化雙主執行緒的全域變數.雙主執行緒就是 UI 一個主執行緒體系,伺服器走次主執行緒體系,程式體系總是走的規範+規則,UI 的主執行緒體系就是走的 vcl,lcl,frm 的主線路綫,而伺服器的次主執行緒體系則是使用 Z 系主執行緒體系規則.

雙主線最奇妙的地方:當未開啟雙主線模型前,兩種主線模型可以互相相容,因為這時候主執行緒只有一個,而當雙主線模型被開啟後就需要區分主線和次主線了,具體細節我也編寫了許多章節文檔和 demo,大家可以自己研究.

因為 Z 系自己實現了 Synchronize 機制,其中有許多非常晦澀的硬體機制,文檔篇幅和人性化的閱讀習慣不支持描述這些東西,靠自己研究把.

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

CPS=caller of per second,由於每個執行緒都會有自己的主迴圈,例如 vcl,lcl,frm 的主迴圈就是迴圈處理消息,而次執行緒的主迴圈是一個事件調用介面,在該介面中需要自己來編寫主迴圈,通常來說,例如 C4,ZNet,DrawEngine 這類框架都會有自己的 Process,這就是主迴圈.這裡的 CPS 是從主次執行緒檢查它執行緒同步中的效能計數器,例如有 10 個子執行緒,他們在工作中會偶爾 Synchronize 到主執行緒執行一下,但是每次 Synchronize 到主次執行緒時,並不能馬上執行而是放到一個佇列,直到主次執行緒 CheckSynchronize 才會被執行,通常來說,如果 application 處於 run 狀態,它會在活動消息迴圈中反復使用 CheckSynchronize,另一方面 CheckSynchronize 不能記錄每次同步的時間消耗,同時也無法按秒單位的時間統計出調用頻率.對於前端,例如手機,win32,只要用起來不覺得卡頓那就是沒問題,但在伺服器端,會有性能洩露,例如從數十種執行緒分支找到每次調用傳遞延遲達到 2 秒那個執行緒會需要資料,而 CPS 就是記錄同步執行延遲+同步調用頻率的資料工具.

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

並行體系的全域動態開關,這是一個 Bool 原子變數,如果為 True,並行程式會在多執行緒模型下運行,否則就在當前執行緒下運行,當 False 時 ParallelFor() 這類 api 會直接工作與當前執行緒,並不會開新執行緒,WorkInParallelCore 變數可以在程式運行中,或則啟動時來動態的開關:如果以前的 ParallelFor() 還在運行中,這時候改變 WorkInParallelCore 變數不會影響以前的並行程式,只會影響以後的並行機制.提示:並行開關不光可以全域變數,也可以區域變數,例如並行 A 用 4 個執行緒,並行 B 只用 1 個執行緒,並行 C 則直接工作於主執行緒,另一方面通過修改編譯器預定義(Z.Define.inc)也可以做到控制並行開關.

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

記憶體 MM 庫的 Hook 狀態開關:普通程式可以忽略.在 ZDB1 設計中,資料是 free 形式,自己設計資料實例,當使用者自己的實例緩存達到某個限度,然後,啟動緩存管理流程,以此來實現高速資料遍歷.因為使用者自己設計資料實例是一個非常隨機的記憶體開銷,因此直接在底層 MM 環節做了一個定向執行緒的 hook 介面,例如在 A 執行緒創建執行資料實例創建和讀取 TmyInst.create,load...,這時候,ZDB1 會記錄 A 執行緒創建實例使用的記憶體開銷,這將為後面的緩存管理提供運行資料層面上依據.GlobalMemoryHook 是控制全域 MM 的流程開關,Z 系 MM-Hook 庫總共有 4 層嵌套,ZDB1 是其中一層,其它 3 層是在使用者層程式使用,只有在它打開時,MM 才會做定向執行緒記錄.一旦關閉所有 MM-Hook 都將失效,同時 alloc,realloc,free 這三個 MM 操作將得到提速.

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

由於早期設備和系統的常規時間刻度都是 32 位整型,1 秒=1000ms,因此 32 位整型最多也只能記錄 30-60 天的時間刻度。

Z 系所使用的時間刻度如果通用於各種硬體和系統,那麼就不能直接使用 GetTickCount 這類 api,而 QueryPerformanceCounter 這類高精度刻度 api 是有平臺和硬體相關性的,也是不能直接使用的,移植時會因為刻度單位,誤差性,帶來很多小問題!

後來,幾經權衡,開了一個全域變數用於繞行 60 天的時間刻度限制,具體做法就是在啟動時使用 GetTickCount 記錄當前刻度,在獲取刻度時通過符號計算達到繞行目的.主要是為 Z 系的 GetTimeTick 提供安全刻度,也許 GetTimeTick 精度誤差會在 16ms 內,這對前後臺已足夠。

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();是初始全程式的互斥區,此處有 3 個重要程式互斥區

1. GetTimeTick 使用的互斥區,每次調用 GetTimeTick 會鎖一下,然後,更新全域變數,GetTimeTick 在執行緒中也是安全的,另一方面 GetTimeTick 流程大約 6 行整數型計算代碼,翻譯成機器碼大約 15-20 行彙編,GetTimeTick 的鎖是極快的,0.001%的鎖延遲幾乎不存在多執行緒性能耗損。
2. AtomInc/AtomDec 原子數操作,在 delphi 平臺原子操作 api 使用 AtomicIncrement, AtomicIncrement 在 x64/x86/ARM 這些平臺是一種彙編命令並不是系統級別 api,而 fpc 平臺是不提供原子 api 的操作,在 fpc 平臺 Z 系就是用互斥區鎖一下,然後再做 Atom 的模擬操作.而 fpc 平臺的互斥區鎖就是在 Init_Critical_System()做初始化.如果使用 delphi 可以忽視這一環節,在 fpc 中使用 AtomInc 其性能是不如 Delphi 的。
3. 在 Delphi 平臺中 TObject 物件內置了互斥區變數,使用方法為 TMonitor.Enter(Obj),而 FPC 平臺中 LCL-TObject 並不提供互斥鎖變數,因此 Z 系提供一種 TObject 於互斥鎖的配對結構,這種結構是基於 TBigList 構建的實例反查演算法,作用是查找與 TObject 配對的互斥鎖變數,然後類比出 TMonitor.Enter(Obj),當反查程式開始工作時需要鎖一下,這種互斥鎖就是在 Init_Critical_System()初始化.模擬 TMonitor.Enter(Obj)這種做法在 2017 年 Z 系早期的程式中會比較常見,而現在幾乎已經放棄 TMonitor.Enter(Obj)的做法,因為這種做法就是用一行代碼簡單的實現互斥鎖,執行緒的程式並不在意多寫兩行,自己開個互斥鎖變數來控制多執行緒機制,這樣不光是程式更容易維護,同時也解決了 fpc 的兼容機制問題。

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

CoreInitTimeTick := GetTimeTick(),這一行是內核的啟動時間,給外部程式提供參照資料用的,這一行沒有含義。

InitMT19937Rand(),這一行是初始化 MT19937 亂數的資料結構,MT19937 是一種隨機跨度非常均勻的亂數產生演算法,同時 MT19937 也覆蓋了 pas,c,c++,go,java,c#眾多開發語言,但 Z 系 MT19937 並不是一個簡單的隨機庫,Z 系 MT19937 是一種體系,Z 系解決多執行緒隨機性問題,Z 系得 MT19937 在每個執行緒中都會有自己的獨立 Seed 運算元,可以做到在大規模在執行緒和並行程式引入統計學演算法,因為統計方法大量使用亂數,雖然某一些統計學演算法庫會給出針對亂數實例的功能,這種需要構建一個隨機演算法實例,然後再把實例傳遞給統計演算法,這種做法遠遠不如執行緒內置獨立亂數實例,例如,我們要移植一個 K-Mean-Cluster,幾乎不需要改代碼,直接 copy 過來,然後把系統內置的 Random 函數替換成 MT19937Rand32 就可以了。

在另一方面,Z 系的 MT19937 是配對的,每次調用亂數時,Z 系都會檢測執行緒配對,這種配對機制與上一節的 TMonitor.Enter(Obj)非常類似,但是 MT19937 只會配對執行緒和 MT19937 實例.除此之外,Z 系提供了 MT19937 的優化執行緒配對,如果執行緒模型走的是 TCompute,那麼配對時候會繞過實例搜索,TCompute 內置了 MT19937 直接在現成使用即可,在 TCompute 執行緒模型下 MT19937 的生成效率是極限的。

Z 系的 MT19937 從底層機理角度解決統計學方法跑執行緒化的大難題.同時 MT19937 有許多變種形式的使用方法,例如自訂實例,自訂種子等等,具體細節 Demo 和文檔會比較少,更多的需要大家自己去研究內核實現。


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

這是初始化 Z 系執行緒池,原型為:InitCoreThreadPool(最大執行緒數限制,最大平行線程細微性)

這一行的白話文解答為:如果調試模式,最大 2 執行緒

如果是 Release 模型:執行緒池最大執行緒=CpuCount*2,並行程式細微性由外部定義決定要麼 8 個要麼 CpuCount*2

InitCoreThreadPool 函數被調用時,會初始化一大堆變數和互斥鎖,然後,它會創建一個 Dispatch 這類調度性質的獨立執行緒,其作用是調度執行緒的回收,創建,啟動.調度機制所使用的技術就是狀態機和結構.

當使用 TCompute.Run 時,TCompute 會把參數以資料 Copy 形式,發送給 Dispatch 調度執行緒,調度執行緒這時會從執行緒池的資料結構去尋找空置執行緒,如果找到了,就把 TCompute 的參數發過去,然後啟動函數,如果沒有找到,這時候會檢查和等待由 InitCoreThreadPool 定義的最大執行緒限制,如果被限制了最大執行緒,它會一直等待,如果不被限制,那就創建一個新執行緒,然後再把參數發過去執行.

值得說明的地方: LimitMaxComputeThread,這是一個預編譯開關,它必須打開以後,調度執行緒才會真正做到對最大執行緒數的限制,如果是關閉狀態,調度執行緒沒有找到回收執行緒時就會創建一個新執行緒. LimitMaxComputeThread 預設是關閉狀態,配置於 Z.Define.inc 預編譯中.

TCompute 的每一個執行緒在執行結束時都會有一個存活週期,只有在存活週期中,TCompute 執行緒才可以被調度執行緒重複使用,默認存活週期時間為 1000ms,由 InitCoreThreadPool 函數內部賦予.說明:存活週期會影響 app 的關閉速度,如果是高頻率+高速調用的 shell 程式,存貨週期可以調整成 100ms,因為 app 關閉時會等全部執行緒結束,把存活週期調小即可視線高頻率 shell 模型.

並行細微性與最大執行緒數的關係:並行模型一律依賴於 TCompute,當並行細微性小於最大執行緒限制時平行線程會暢通無阻執行.而當最大執行緒限制為 20,當前已經已有 18 個運行中執行緒,這時候並行細微性為 20,那麼真正執行並行程式的執行緒只會有 2 個執行緒,它會反復執行,模型為:必須啟動完 20 個執行緒,TCompute 會反復先執行 2 個平行線程,待這兩個平行線程執行完,再執行 2 個,一直跑滿 20 個執行緒.通常來說,最大執行緒數量都是不限制的,當預編譯開關 LimitMaxComputeThread 是關閉時,並行程式只要指定 20,那麼 20 個執行緒都會瞬間進入工作狀態,並不會等待.

Z 系並行細微性為 8 的含義:並行程式的計算目的是加速 for 的處理能力,通常來說,並行提速達到 8 倍會是一個極限,更多的會是 2-4 倍左右的提速,具體要看並行程式的計算類型,例如浮點計算,浮點需要區分小浮點流程和大浮點流程,在小浮點流程中直接 for 會比並行更快,因為並行會有一個啟動時間,在大浮點流程中,浮點計算會以上百行起步,這種才適合使用平行計算.在 x64 架構中,浮點計算通常是用整數模型,走 sse 的臨時寄存器做符號計算,sse 通常內置於每一個 cpu 核裡面,並不是全核共用 sse,sse 是指令集的一部分,而模擬化浮點 sse 的輸出和輸入指標都是目標記憶體位址,這時候浮點傳遞會有一個記憶體 copy 時間,這是在不同設備的資料傳遞,這種 copy 時間會限制並行的性能.一般情況下,浮點的 copy 很小,高頻率記憶體完全可以勝任把浮點 copy 跑滿,提速幾十倍,但這樣一干,cpu 就沒有空間可以運行別的性能需求程式.在除卻浮點計算之外,在多路 cpu 架構的伺服器中,當 cpu 跑滿,就連 pci 的傳遞管線都會被影響,反應出來會是往 gpu 裡面傳遞資料出現明顯延遲甚至卡頓,因為多路 copy 需要依賴於 cpu 去做資料的 copy 計算.而當並行程式遇上字串,資料結構這類處理,copy 機制會大量使用,這時候即使滿核運行,也會被記憶體頻寬所限制.總結一下 8 的含義,給硬體留一點計算資源,並行數量並不是越多越快,並行程式只要能提速 4 倍以後就算 ok,最後是 8 的中文拼寫代表發,發財.在以前,許多做大資料的人,給我說,排序,字串轉換,他們都用並行開滿來跑,其實把,圍繞字串這種計算領域,開滿計算耗時會遠大於 8 執行緒,被記憶體+北橋限制,也許應該讓他們瞭解一下 Z 的平行計算思路.

另外一點:在 TCompute 執行緒中,TCompute.Run 總是開執行緒執行,使用上並沒有多餘的概念和思路,內部如何工作無需關心細節,因為本小節是說內核啟動,對於 TCompute 就不做深入說明了.

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

這是一個主執行緒 Post 框架的全域初始化,執行緒 Post 從字面來理解就是往目標執行緒裡面提交一個可執行調用,主執行緒 Post 框架就是往主執行緒提交可執行調用,因為執行緒都會有 synchronization 方法,但 synchronization 會等待調用退出,Post 則是發送後立即返回.

TThreadPost 實例內部有非常多的屬性和模式,OneStep 是每一個 Progress 是否只處理一次執行,例如佇列有 10 個可執行提交,而 OneStep 為 True,那麼就需要 10 次 Progress,當 OneStep 為 False,那麼每次 Progress 都會執行全佇列.另外, TThreadPost 還具備 MT19937 隨機化能力,每一次執行時都會重置一下 MT19937 的亂數種子.

如何推翻使用 ZNet 的專案

如果之前使用 ZS 走物理雙通道的專案,推翻直接換 C4.

如果之前使用非 C4 雙通道,推翻直接換 C4.

如果之前使用 ics,indy,win socket 的非 web 類專案,直接推翻換 C4,全面碾壓

如果專案已經是 C4,可以換個註冊名,RegisterC40('MY_Serv', TMY_Serv, TMY_Cli),然後基於 C4 再重新開一個項目 RegisterC40('MY_Serv2.0', TMY_Serv2, TMY_Cli2).簡單來說就是大改走增量,單元名加個版本號,Reg 新服務就行了.

如何使用 ZNet 開發 web 類專案

資料通訊層直接 ZNet 跑,UI 層用 webapi 訪問 ZNet 的專案即可.例如 Post+Get 基本能覆蓋 90% 的 webapi 需求,ZNet 自帶一個 webapi 的 demo 項目.

ZNet 與 http 和 web

ZNet=大型 CS 伺服器

http=通訊協定

web=全球廣域網路,互聯網

web 包含了 ZNet,在 web 環境下用 apache,nginx 橋通訊模組的大型網站比比皆是,或許讀者有空可以試試用二級功能變數名稱或則域伺服器來做橋接模組和分流,內部如果涉及到大資料或則複雜協定,直接用 ZNet 做個通訊層包 api 給 web 用.

文本最後來一個極簡 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.tstC, [
    '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