分布式网游服务器集群

概要设计说明书

Version 1.9

baiy.cn

前言

有兄弟批评前言写的不好，那我就把谈情怀的部分删了，直接上干货吧...

本人在单服务器支撑千万量级 HTTP、TCP 并发的高效 IO 服务器构建，以及强一致、抗脑裂（Split Brain）的多活 IDC 分布式高可用（HAC）和高性能（HPC）计算集群等方面有多年积累。历经多家全球 500 强企业实际生产环境十几年考验。在分布式算法、UI 布局等方面拥有多个国家和国际发明专利。

我从小学习编程的原动力来自于姚仙的《仙一》（也想写一个那么棒的游戏……），一晃 20 多年，程序没少写，可游戏还是一点没做过~。在看到当今网游行业的服务器端单点性能普遍不佳，并且其整体架构仍处于“分区分服，相互分割”的落后模式时，我觉得结合本人多年来在此领域的积累，提供一套单点性能高，同时支持高性能（HPC）+高可用（HAC）的分布式网游服务器集群解决方案，对我来说可能是一个不错的切入点，于是便有了此文。

写作此文的目的有二：一是为各路同行抛砖引玉，拓展思路，同时我也真诚地希望听到来自游戏领域先辈们的反馈和建议；二是希望以此为引，召唤到有志于踏踏实实开发出好游戏的小伙伴们一起来做个好游戏 :-)

最后，我要感谢来自暴雪、完美、盛大、以及其它游戏开发团队的前辈们对本文的认真审评和宝贵建议。也由衷希望能够继续听到更多来自各路游戏行业前辈的声音。同时，更欢迎想要一起玩耍的前端同学联系我，我的 QQ、Email：asbai@msn.com，多谢 :-)

白杨，于 2015 年底

版本控制

版本号	修改时间	修改内容	修改人	审阅者
1.0	2015-10-27	创建	白杨	
1.1	2015-11-10	澄清一些细节	白杨	
1.2	2015-11-24	新增“前言”小节	白杨	
1.3	2015-12-21	更新一些数据	白杨	
1.4	2016-01-22	调整部分小节顺序	白杨	
1.5	2016-03-18	新增了“山寨防护”小节	白杨	
1.6	2016-09-05	调整“前言”小节	白杨	
1.7	2016-12-10	新增“3.2.3.3 基于 BYPSS 的高性能集群”小节	白杨	
1.8	2017-07-29	为“1.3 应用支撑平台”小节增加更多描述；个别内容增补	白杨	
1.9	2018-01-13	一些次要更新	白杨	

目录

1. 概述

如今的 MMOG 仍然处于分区分服的状态，更糟糕的是，由于其中大部分仍在使用低效的同步 IO 伺服模型，因此每台服务器能够支撑的最高并发数被限制在仅仅几千并发连接。有鉴于此，本技术方案将从单点性能到 HPC、HAC 集群两方面对此局面进行整体改进。

1.1 服务器单点性能

单点性能方面，使用汇编和异步 IO 进行了优化，通过 DMA+硬件中断实现内存 0 拷贝的高效网络伺服。性能、可靠性和可伸缩性都很强，可在 2011 年出厂的单台至强 5600 入门级 1U PC Server 上实现**上千万 TCP/HTTP 并发连接及实时组播支持**，详见：单点支撑千万量级并发的高效 IO 服务器组件。

单点并发 IO 能力的巨大提升使得服务器瓶颈从 IO 并发限制转移到了 CPU、内存等现代硬件资源较充沛和廉价的计算资源（游戏服务器中通常有大量角色逻辑和物理仿真计算）上。这大大提高了服务器的整体性能。作为实例，俄罗斯游戏开发商 Wargaming.net 采用了异步 IO 的网游“坦克世界”在 2011 年就实现了单服 25 万玩家同时在线的性能指标（http://games.qq.com/a/20111209/000503.htm）。

这就说明，在实施了恰当优化后，完全可以实现单服玩家同时在线达到十万至百万量级（取决于服务器配置、游戏逻辑复杂度以及脚本化程度等因素）的目标。比起目前业界普遍受限于“阻塞 IO+每进程/线程一连接”2000 至 5000 并发极限的低效模式，**可提供数千乃至数万倍的负载能力提升**。

1.2 服务器集群

对比精益求精、美轮美奂的客户端，服务器端架构一直是网游产品的传统短板：即使是强如《魔兽世界》（http://www.battlenet.com.cn/wow/zh/status）、《英雄联盟》（http://lol.qq.com/act/a20150326dqpd/）、《天涯明月刀》（http://wuxia.qq.com/server.shtml）这样国际和国内顶尖级的网游产品，也仍然沿用着分区分服、“单打独斗”的老旧架构。

相较于传统的分区分服模式，HAC+HPC 分布式集群对 MMOG 来说，有如下优势：

1. **统一的大世界**：所有服务器彼此连接成为一个大世界，玩家可以在大世界中自由探索、漫游和冒险，和所有其它玩家互动和交流。不再被分割在不同服务器孤岛小世界中（不同服务器间无法彼此互动），也不再需要为选择服务器而烦恼。

当然，即使在统一的大世界中，仍可对其中部分地图（或其中某些特殊实例）实行以等级或其它条件为标准的准入机制，或在某些地区实施“等级差异过大则禁止动武”之类的特殊限制，已达到新手保护、VIP 专享等目的。

2. **动态负载均衡（HPC）**：提供优秀的可伸缩性和横向扩展能力，不同服务器之间可动态地均衡负载。不会出现有些服务器负载很高，有些则处于轻载状态的尴尬情况，这有以下好处：

   a) **节省运营成本**：负载均衡化使得原先需要 200 台独立服务器节点才能支撑的业务如今只需要 100 台甚至更少节点组成的分布式集群即可支撑。

   b) **提升用户体验**：不再因为部分服务器负载过高导致玩家出现卡顿、掉线等不良游戏体验。

3. **多活 IDC 高可用（HAC）**：抗脑裂（Split Brain）的多活 IDC 架构确保即使集群中多个服务器节点同时宕机，甚至整座 IDC 由于市政施工、自然灾害、人为故障等原因下线，也可自动实时完成故障转移（Failover）和故障恢复（Failback）等动作，为玩家提供透明的高可用服务体验。

4. **支持非常复杂的世界观**，例如：每玩家、每工会一个专有 minecraft 小世界。Minecraft 类的沙箱环境允许玩家从原子级别从头定义属于自己的虚拟世界。将这样复杂和自由的世界观 MMOG 化需要只有通过分布式计算才能承担的大量计算资源。

   其它如宏大的虚拟游戏世界、1:1 克隆真实城市或地区等情形也是类似，总的来说：玩家自由度越大、仿真程度越高的世界，对运算和存储能力等资源的需求也就越多——但这样的世界往往更加精彩纷呈。而单点的性能总有天花板（在考虑高端硬件的性价比后就更是如此）。为此，从单点各自为政到分布式集群计算的转变将不可避免。

5. **山寨防护**：统一的大世界和复杂的世界观都需要依赖高性能的分布式服务器计算集群来实现，这在客观上大大增加了游戏被竞争对手模仿和抄袭的难度。这对山寨文化横行的国内游戏市场来说，显然尤为重要。

   此外，合理利用《白杨应用支撑平台》中的强加密通信组件和支持实时（on-the-fly）数据压缩和强加密的虚拟文件系统（VFS）等相关功能模组，还可大大增加网络协议的分析难度，同时加强对存储于客户端和服务器端磁盘数据的安全保护。这无疑可进一步提升敌手对产品进行分析、破解和抄袭的难度。

我们在分布式计算集群的架构和实现方面，拥有构建大规模分布式高性能（HPC）和高可用（HAC）计算集群的丰富经验——依托于我方成熟的 nano-SOA 大规模分布式架构，以及与之配合的，基于抗脑裂（Split Brain）多数派算法的，高可用，强一致分布式故障检测、服务发现、服务选举、分布式锁等分布式协调服务和消息分发与路由服务（BYPSS）等组件。我方可提供非常成熟可靠的 HAC+HPC 游戏服务器集群解决方案，详见：强一致多活 IDC 高可用（HAC）和高

性能（HPC）服务器集群。

1.3 应用支撑平台

《白杨应用支撑平台》作为我方的核心竞争力，使用汇编、C/C++ 构建，包含数百万行代码和上千个成熟的通用组件。上述单点高并发和多活 IDC 分布式集群等核心技术均由白杨应用支撑平台提供。多年来，基于支撑平台的各种产品已被广泛部署于包括兴业银行（China CIB）、中石油（CNPC）、华安保险（Sinosafe Insurance）、淘宝网（taobao.com）、易果网（yiguo.com）、烟台万华集团、法国兴业银行（SOCIETE GENERALE）、德尔福汽车（Delphi）、美联航（United Airlines）、GE（美国通用电气）、贝塔斯曼（Bertelsmann）、埃森哲（Accenture）、光大银行（CEB）、壹基金（One Foundation）、中国宋庆龄基金会、中国移动（China Mobile）、中国联通（China Unicom）等各大企业在内的不同生产环境中：

真实生产环境下的大范围部署不但为上层应用提供了可靠的、平台无关的底层环境，也进一步检验了支撑平台的可靠性、稳定性、可移植性、高效性等各方面指标。

支撑平台使用汇编、C/C++ 构建，支持 Windows、Linux、BSD、IBM AIX、HP-UX、Solaris、MAC OS X、vxWorks、QNX、DOS, WinCE (Windows Mobile), NanoGUI、eCos、RTEMS 以及 Android、iOS 等绝大多数主流操作系统。支持 x86/x64、ARM、IA64、MIPS、POWER、SPARC 等主流硬件平台。在提供了大量高品质可重用组件的同时，也确保了良好的可移植性。

上千成熟可靠的高品质功能组件可在性能、功能和稳定性等方面大幅提升软件产品的品质，并为产品的开发带来了难以想象的便利，例如：

单点支撑千万量级并发的高效 IO 服务器组件

支撑平台使用汇编和异步 IO 对网络服务组件进行了优化，通过 DMA + 硬件中断实现内存零拷贝的高效异步网络服务。性能、可靠性和可伸缩性都很强。可在 2011 年出厂的，当时售价不足 2 万元人民币的单台至强 5600 系入门级 1U PC Server 上支撑上千万 TCP/HTTP 并发连接。相对应地，一般由 Java / .NET 开发的服务器端，在相同配置的机器上，单点最高仅可支撑 3000 到 5000 并发，PHP 则更低（具体可参考《白杨应用支撑平台技术白皮书》：3.2.1、3.3.1 以及 3.3.2 小节）。

强一致多活 IDC 高可用（HAC）和高性能（HPC）服务器集群

强一致、抗脑裂（Split Brain）的多活 IDC 分布式高可用（HAC）和高性能（HPC）计算集群支持：独有的 nano-SOA 大规模分布式架构在保持高内聚、低耦合设计的前提下，将单点性能提升到了远超传统 SOA 架构的水平，同时简化了集群部署，提高了集群的可维护性。

白杨消息端口交换服务（BYPSS）：一种基于多数派算法的，强一致（抗脑裂）、高可用的分布式协调组件，可用于向集群提供服务发现、故障检测、服务选举、分布式锁等传统分布式协调服务，同时还支持消息分发与路由等消息中间件功能。由于通过专利算法消除了传统 Paxos/Raft 中的网络广播和磁盘 IO 等主要开销，再加上批量模式支持、并发散列表、高并发服务组件等大量其它优化，使得 BYPSS 可在延迟和吞吐均受限的跨 IDC 网络环境中支持 10 万节点、百亿端口量级的超大规模计算集群。

带强一致保证的多活 IDC 技术是现代高性能和高可用集群的关键技术，也是业界公认的主要难点。作为实例：2018 年 9 月 4 日微软美国中南区某数据中心空调故障导致 Office、Active Directory、Visual Studio 等服务下线近 10 小时不可用；2015 年 8 月 20 日 Google GCE 服务中断 12 小时并永久丢失部分数据；2015 年 5 月 27 日和 2016 年 7 月 22 日支付宝两次中断数小时；以及 2013 年 7 月 22 日微信服务中断数小时等重大事故均属于产品未能实现多活 IDC 架构，单个 IDC 故障导致服务全面下线的惨痛案例。

在上述方面，我方拥有超过十年的积累，掌握多项受国家和国际发明专利保护的分布式架构和算法。得益于这些领先的强一致、高可用、高性能分布式集群算法和架构，我们在蓝鲸、白豚、职业精等全线产品上，均实现了真正的多活 IDC 架构，为客户提供了无以伦比的数据可靠性和服务可用性保证。

分布式协调服务

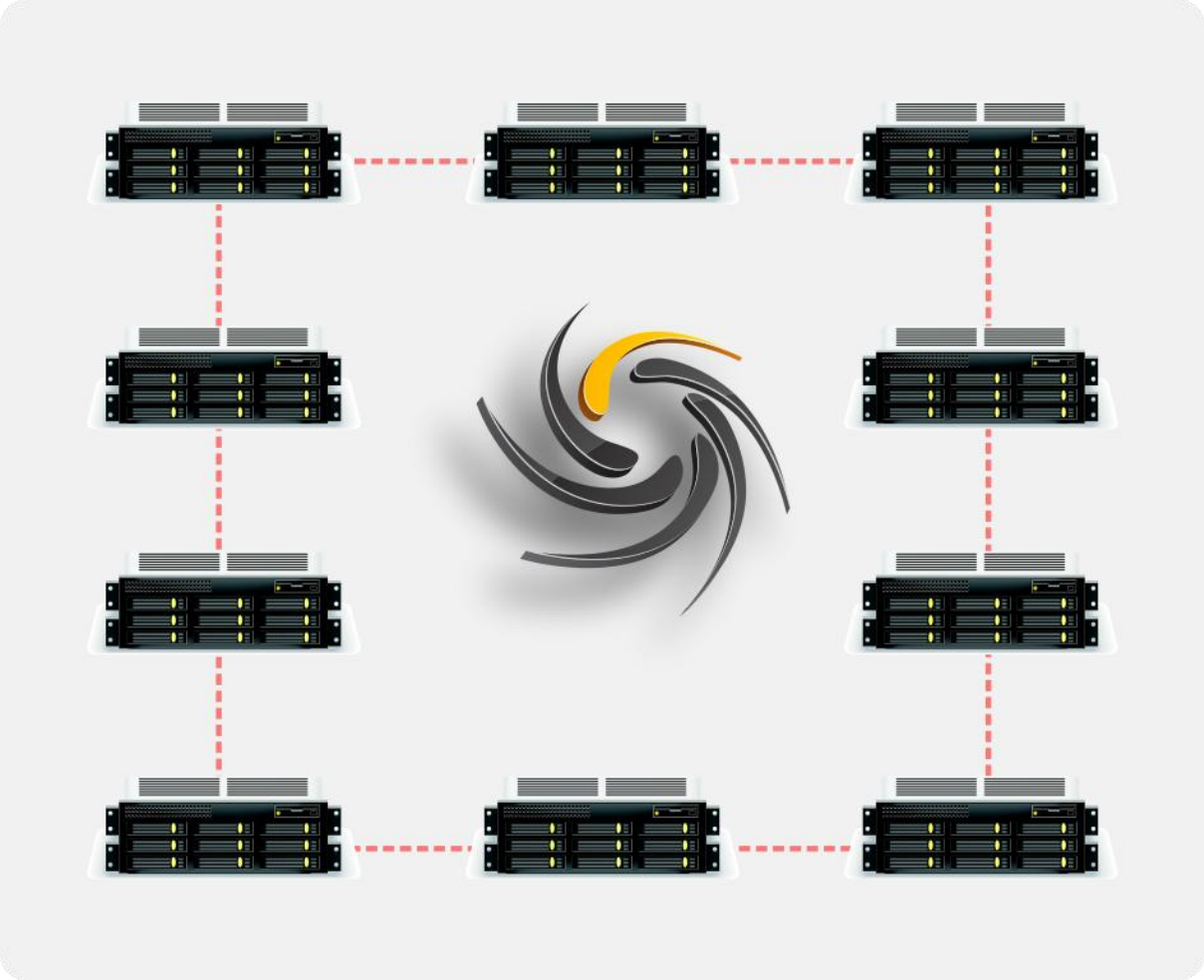

图 1

分布式协调服务为集群提供服务发现、服务选举、故障检测、故障转移、故障恢复、分布式锁、任务调度，以及消息路由和消息分发等功能。

分布式协调服务是分布式集群的大脑，负责指挥集群中的所有服务器节点协同工作。将分布式集群协调为一个有机整体，使其有效且一致地运转。实现可线性横向扩展的高性能（HPC）和高可用（HAC）分布式集群系统。

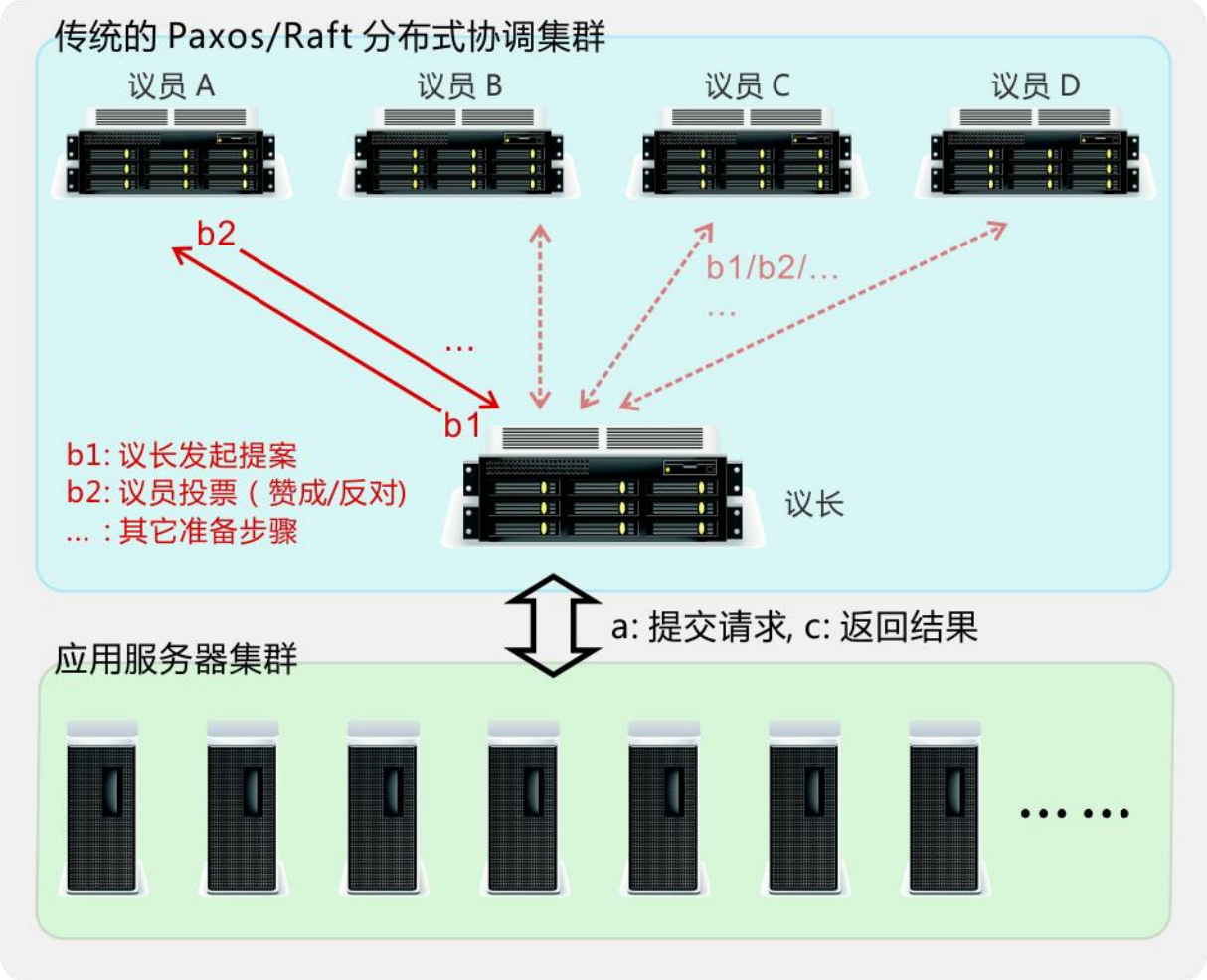

图 2

传统的 Paxos/Raft 分布式协调算法为每个请求发起投票，产生至少 2 到 4 次网络广播（b1、b2、…）和多次磁盘 IO。使其对网络吞吐和通信时延要求很高，无法部署在跨 IDC（城域网）环境。

我们的专利算法则完全消除了此类开销。因此大大降低了网络负载，显著提升整体效率。并使得集群跨 IDC 部署（多活 IDC）变得简单可行。

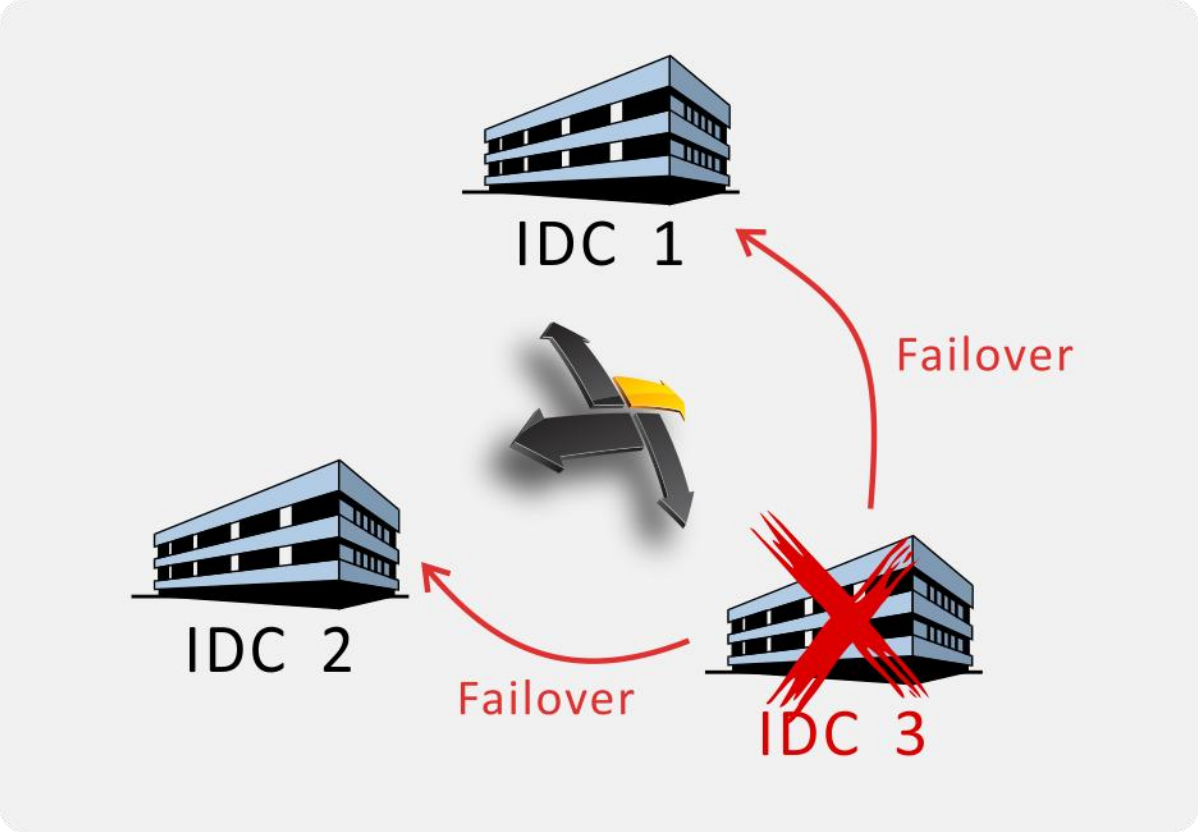

图 3

基于我方独有的分布式协调技术，可实现高性能、强一致的多活 IDC 机制。可在秒级完成故障检测和故障转移，即使整座 IDC 机房下线，也不会导致系统不可用。同时提供强一致性保证：即使发生了网络分区也不会出现脑裂（Split Brain）等数据不一致的情形。例如：

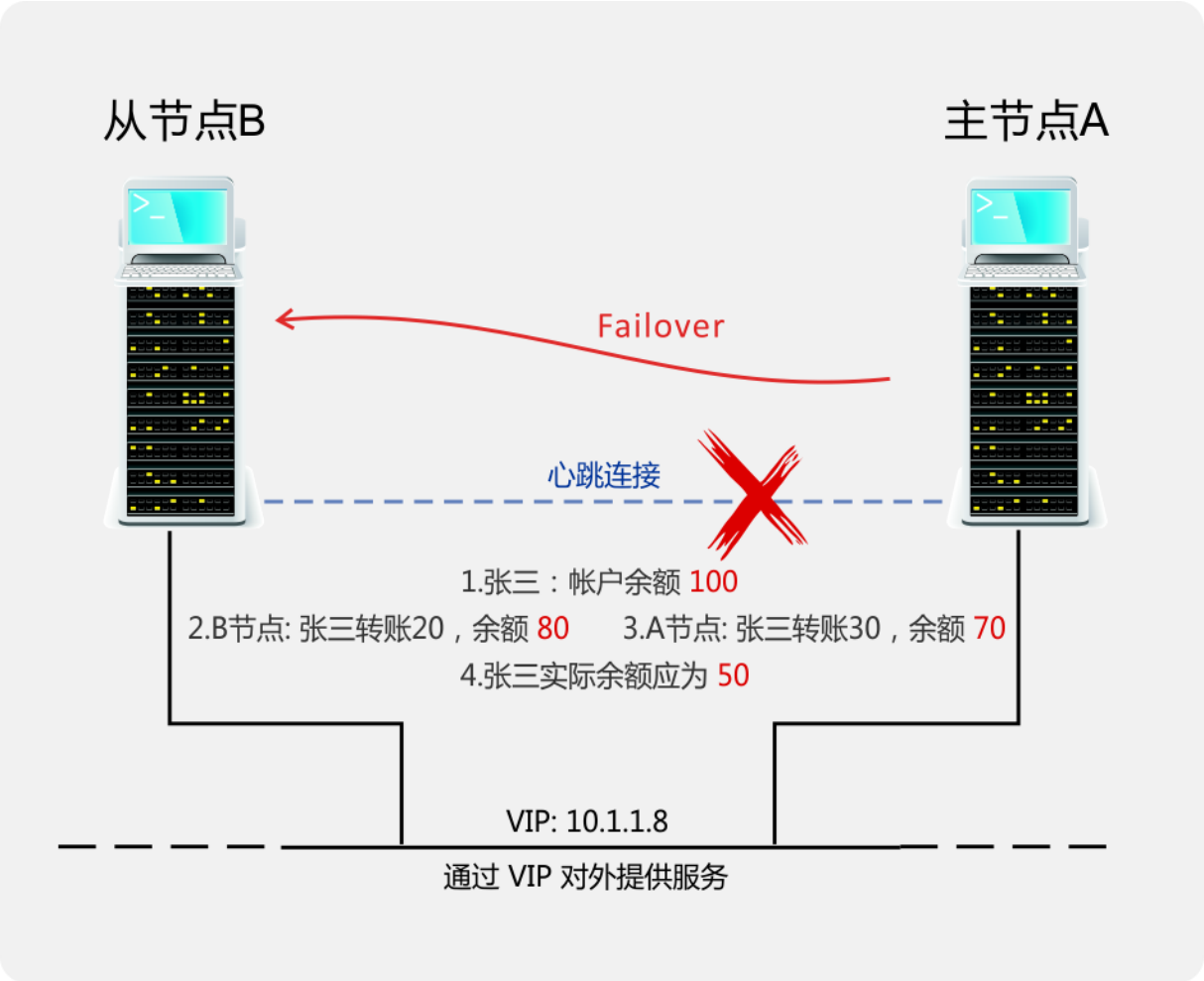

图 4

在传统的双机容错方案中，从节点在丢失主节点心跳信号后，会自动将自身提升为主节点，并继续对外提供服务，以实现高可用。在此种情形中，当主从节点均正常，但心跳连接意外断开时（网络分区），就会发生脑裂（Split Brain）问题，如图 4 所示：此时 A、B 均认为对方已下线，故将自己提升为主节点并分别对外提供服务，产生难以恢复的数据不一致。

我方 BYPSS 服务可提供与传统 Paxos/Raft 分布式算法相同水平的强一致性保证，从根本上杜绝脑裂问题的发生。

类似地：工行、支付宝等服务也有异地容灾方案（支付宝：杭州 → 深圳、工行：上海 → 北京）。但在其异地容灾方案中，两座 IDC 之间并无 Paxos 等分布式协调算法保护，因此无法实现强一致，也无法避免脑裂。

举例来说，一个在支付宝成功完成的转账交易，可能要数分钟甚至数小时后才会从杭州主 IDC 被异步地同步到深圳的灾备中心。杭州主 IDC 发生故障后，若切换到灾备中心，意味着这些未同步的交易全部丢失，并伴随大量的不一致。比如：商家明明收到支付宝已收款提示，并且在淘宝交易系统看到买家已付款，并因此发货。但由于灾备中心切换带来的支付宝交易记录丢失，导致在支付宝中丢失了相应的收入，但淘宝仍然提示买家已付款。因此，工行、支付宝等机构在主 IDC

发生重大事故时，宁可停止服务几个小时甚至更久，也不愿意将服务切换到灾备中心。只有在主 IDC 发生大火等毁灭级事故后，运营商才会考虑将业务切换到灾备中心（这也是灾备中心建立的意义所在）。

因此，异地容灾与我方的强一致、高可用、抗脑裂多活 IDC 方案具有本质区别。

由于消除了 Paxos/Raft 算法中的大量广播和分布式磁盘 IO 等高开销环节，配合支撑平台中的高并发网络服务器、以及并发散列表等组件。使得 BYPSS 分布式协调组件除了上述优势外，还提供了更多优秀特性：

批量操作：允许在每个网络包中，同时包含大量分布式协调请求。网络利用率极大提高，从之前的不足 5% 提升到超过 99%。类似于一趟高铁每次只运送一位乘客，与每班次均坐满乘客之间的区别。实际测试中，在单千兆网卡上，可实现 400 万次请求每秒的性能。在当前 IDC 主流的双口万兆网卡配置上，可实现 8000 万次请求每秒的吞吐。比起受到大量磁盘 IO 和网络广播限制，性能通常不到 200 次请求每秒的 Paxos/Raft 集群，有巨大提升。

超大容量：通常每 10GB 内存可支持至少 1 亿端口。在一台插满 64 根 DIMM 槽的 1U 尺寸入门级 PC Server 上（8TB），可同时支撑至少 800 亿对象的协调工作；在一台 32U 大型 PC Server 上（96TB），可同时支撑约 1 万亿对象的分布式协调工作。相对地，传统 Paxos/Raft 算法由于其各方面限制，通常只能有效管理和调度数十万对象。

综上，我方专利的分布式协调算法，在提供与传统 Paxos/Raft 算法相同等级的强一致性和高可用性保证之同时，极大地降低了系统对网络和磁盘 IO 的依赖，并显著提升了系统整体性能和容量。对于大规模、强一致分布式集群的可用性（HAC）和性能（HPC）等指标均有显著提升。

关于 BYPSS 服务的进一步描述，详见：3.2.3 白杨消息端口交换服务（BYPSS）。

高效、高强度的密码编码学组件

包含公钥算法、对称加密算法、数据编解码、散列和消息验证算法、数据压缩算法等基础功能组件。除此之外，支撑平台还提供了多个经过高度抽象、可开箱即用的高级密码编码学功能组件，例如：

支持实时压缩和强加密的虚拟文件系统（VFS），VFS 支持包括 AES（128/256）、TwoFish 等在内的数十种强加密算法，使用 AES-NI、SSE4 等汇编指令集优化，效率高。在蓝鲸、白豚、职业精等全线产品中，我们均使用该组件为产品的数据库和配置类数据提供整库级的实时压缩和强加密保护。另外还包括基于公钥体系架构（PKI）的强加密通信保护组件等。

近年来安全问题频发，亚马逊、沃尔玛、Yahoo、Linkedin（领英）、索尼、摩根大通、UPS、eBay、京东、支付宝、一号店、协程、12306、网易、CSDN、中国人寿、以及各大酒店集团（如家、汉庭、锦江、洲际、喜来登、万豪等）等国内外知名企业均频频报出大量用户信息泄露的严

重安全事件，安全保障已经刻不容缓。

我方所有数据库（整库）和本地配置数据均存放在我方自主研发的，支持实时（on-the-fly）数据压缩和强加密的虚拟文件系统（VFS）中进行全方位保护。支持数十种业界公认的强加密安全算法，即使系统管理员也无法窥视企业数据。

基于业界标准的强加密算法保证了即使未来出现了每秒钟可完成一千万亿次密钥破解尝试的超级计算机，也需要平均五千四百万亿年才能破解一个密钥。安全性得到了极大保证（具体可参考《白杨应用支撑平台技术白皮书》中的第 4 节）。

数据查询分析引擎

支撑平台中还包含了表达力优于 SQL 的查询分析引擎，自有查询引擎除了可以摆脱对特定 DBMS 的依赖，使我们的产品可以自由地在 MySQL、MS SQL Server、Oracle、DB2、SQLite 等 RDBMS 以及 MongoDB、Cassandra 等 NoSQL 数据库间灵活切换。更增加了基于 UNICODE 字符集的 ARE 高级正则查询、表中套表的关联查询、复杂虚拟字段等 SQL 没有的高级特性。

查询引擎通过汇编优化的 C/C++ 代码实现词法分析、语法分析、语义分析/中间代码生成、优化等步骤，并可在 2010 出厂的 Thinkpad W510（4 核 8 线程，主频 1.6G）笔记本上，仅用其中一核一线程即可达到每秒 1300 万次以上的表达式求值效率。

更多...

我们并不依靠“商业秘密”来保护核心竞争力。相反，我们使用更公开透明的商标、认证、版权、专利和公证保管等手段来保护自己的合法权益。因此，我们的技术细节均公开于相应的文档中，详情可见《白杨应用支撑平台技术白皮书》：

http://baiy.cn/doc/asp_whitepaper.pdf
或者：http://baiy.cn/doc/asp_whitepaper_en.pdf（英文版）

等文档，包括 Hacker News（全球最大的计算机科学新闻网站）、Google Blogger、CSDN 以及博客园在内的多家国内外媒体均转载或报道了这些论文。相较于“严守秘密”，我们相信公开透明下的大量同行审评，加上实际生产环境下的严酷考验，更有利于产品品质的提升。

2. 设计目标

系统在规划和设计阶段需要达到或满足如下要求：

* **用户数：**单点单库支持用户数不低于 2 千万，单点在线并发用户数支持不低于 50 万[(1)]。并可通过增加节点、升级到分布式 DBMS（分库分表）等方式来线性增加用户容量。

* **前台并发数：**前台伺服界面需要在主流 PC Server[(1)]上支持不少于 500 万 HTTP/TCP 并发连接。

* **后台并发数：**后台伺服界面需要在主流 PC Server[(1)]上支持不少于 500 HTTP 并发连接。

* **稳定性：**系统要能够长时间稳定连续地工作，期间无需人为干预。系统应支持以强一致、抗脑裂（Split Brain）的多活 IDC 高可用+高性能集群（HAC+HPC）方式来进行部署，以期在保证高性能的同时，为用户提供服务高可用性保障。

* **可靠性：**针对所有支持的任务，系统必须以可靠的方式完成。确保任务执行的原子性，如果在执行中发生不可避免的错误（如电源故障、磁盘损坏等），系统必须能够恢复到任务执行之前的状态。

  系统中的所有数据应以强一致多副本的形式被同时保存至至少 2 座数据中心内，数据中心之间应至少相隔十公里以上，以提高数据可靠性和服务可用性。

* **弹性：**随着网络规模不断增长，系统所承受的负载将会相应上升。不同的负载量和健壮性要求将直接导致不同的软／硬件平台选择和体系结构的变化（例如：从初期运行 Windows 系统的单台 PC 服务器+嵌入式数据库到后期运行 UNIX 系统的小型机集群+分布式数据库）。系统的设计必须保证足够的伸缩性，使其能够以较高的性价比同时适应不同规模的实际需求。

* **容量：**系统提供的横向扩展（HAC+HPC）能力应确保其可稳定支持单表万亿记录（行）、PB 级别的海量数据。

* **可扩展性：**系统必须能够很好地支持数据并行、任务并发以及流水线并行等各种并发模式，以结合可配置的内存及处理器管理策略来实现较好的纵向扩展能力。系统中的各主要部件应支持松耦合的分布式架构，以便于便捷地实现较高的横向扩展能力。

* **可移植性：**系统中的所有平台相关性均应被位于底层的应用支撑平台透明地封装。使得系统具备极高的可移植性特征，可以同时运行在不同的软硬件环境中。以期在简化演示、测试、调试等任务的同时，降低特定硬件平台对产品的粘度，为用户提供更丰富的平台选择。

* **安全性**：要求系统必须能对用户进行鉴权和访问控制——系统要有效地对用户的身份进行识别，管理和行为审查。数据通讯的可靠性、保密性和完整性必须以强加密的方式加以保证。数据在存储至 DBMS 时，也应进行强加密变换和保护。

* **易用性**：为用户提供简单易用、清晰明了、兼容各主流平台的图形化后台管理界面。并通过语言包机制支持多种语言文字的国际化界面。

* **易维护性**：要求在系统正常运行的过程中，不需要或仅需要很少人工干预和维护工作（fire and forget）。要做到故障发生时，在秒级自动检测并执行故障转移动作；故障修复后，自动感知并完成故障恢复操作。

* **灾难恢复与异地容错**：系统必须提供在线备份和恢复能力，使用户免于由自然灾害、硬件故障、系统崩溃等灾难事故带来的数据丢失。并为对可用性和可靠性要求较高的组织提供异地容灾和完好的多活 IDC HAC 支持。

注 1：以上单点性能应均可在当前主流入门级（2-3 万人民币/台）商用 PC Server 上实现。并可通过升级服务器配置（纵向扩展）的方式，来线性地提升系统单点性能和容量。

3. 关键设计考量

本节集中讨论分布式网游服务器集群系统设计中遇到的关键性设计抉择。

3.1 消息通信模式和性能调优

无论是基于 HTTP 还是裸 TCP，超高并发下的消息实时组播对服务器处理能力和底层网络 IO 性能均是一个重大挑战。这也是高并发网络游戏服务器设计中的一个关键问题。考虑数百万乃至数千万玩家同时在线时，每个玩家都需要连续接收处于其可视（可感知）范围内的其它玩家和 NPC 位置等状态变化的实时通知。这无疑需要超高并发和大规模实时组播技术的支持。

具体来说就是要求服务器集群在支持海量并发的同时，还要针对不同的地理位置（地图）局部分别进行连续、高密度的实时消息组播推送和其它复杂的，地图内甚至跨地图的消息交换服务（例如：聊天频道、交易信息发布等）。

为此，从架构、模式和算法上，我们开展了如下针对性调优设计：

* 单点性能调优：正如前文所述，依托于我方自主研发的多线程异步 IO+内存零拷贝高效网络服务组件。可在 2011 年出厂的单台入门级 1U PC Server 上提供上千万 TCP/HTTP 并发连接及实时组播支持。

  使用汇编优化的 C/C++ 静态代码来代替 Python、Lua、Java、C# 等脚本或动态语言来描述业务逻辑亦可在计算方面获得数十至数百倍的时空性能提升。同时为降低游戏编辑的使用门槛，在每块新业务逻辑开发初期仍可允许使用脚本语言进行描述，并于业务逻辑大体稳定且正式上线后，再将其（通过自动或手动的方式）转换为 C/C++ 代码（详见：3.3 游戏逻辑优化）。

  单点性能的全面优化可成千上万倍地提高游戏服务器端的负载密度，使得单个服务器节点可容纳更多玩家同时在线，并支持更为精致复杂的业务逻辑。

* 使用紧凑的 nano-SOA 架构（详见：3.2 nano-SOA 架构），将 SOA 中的绝大部分服务间消息通信约束在相同物理节点内部。这一方面极大地减少节点间通信的数量；另一方面也降低了可靠性、可用性、一致性等对消息中间件性能影响最大的几项指标的要求，进而大大提升了节点间消息通信的性能。

  通信量大大降低的同时，通信性能又显著提升——nano-SOA 从正反两方面彻底解决了传统 SOA 架构中，导致请求处理延迟高、性能差的消息通信瓶颈问题。而消息的处理延迟和吞吐能力等指标，对于对实时性要求非常高的网游来说，正是影响游戏体验的致命问题。

* 服务器节点/地图/玩家（用户）三级伺服架构：一个在线玩家在任意给定时刻只能位于（属于）一份地图实例；一份活动的地图实例在任意给定时刻只能属于一个服务器节点（属主节点）。这就进一步降低了服务器间的消息通信需求——位于同一地图实例内的所有玩家都归属于同一个属主节点，玩家的交互行为（无论是玩家与玩家之间还是玩家与 NPC 之间）又多发生在同一个地图实例内，跨地图实例的消息传递相对很少。

  此外，玩家可以在不同地图实例间自由往来，而地图实例的所有权也可在多个服务器节点间实时传递，这就实现了动态负载均衡（高性能）和自动故障转移（Failover，高可用）。

* 消息批量打包发送：玩家兴趣范围（AOI）通常是以玩家当前坐标为圆心，半径 N 米之内的一片区域（在遮挡地形等特殊环境下也可进行对应的优化）。MMOG 的一大挑战就是要分别向每个玩家实时地推送其 AOI 范围内，所有其它玩家和 NPC 的位置、血量、装备等状态变化。这也是 MMOG 集群相对于传统服务器集群架构的最大挑战。

  虽然上述几项措施既有效消除了绝大部分服务器节点间通信的必要，又显著提升了消息交换的性能。但服务器与客户端之间的大量实时消息推送仍然有进一步优化的余地：人类受自身生理限制，感知能力和神经反射速度也有其极限。人类的视听感知极限在毫秒级（通常几十毫秒），神经反射则更慢，需上百毫秒。

  因此实际上没有必要在每一物体每次发生状态变更时，都立刻向所有对其感兴趣的玩家一一推送通知。相反，可以将某段时间内的所有状态变更信息都缓存起来，并在该时间片结束时一次性推送给感兴趣的玩家。

  时间片的长度可由具体的游戏逻辑和场景而定，例如：在对实时性要求不高的普通地图上，时间片可以是 100 至 200 毫秒。一个玩家的 AOI 范围内可能包含数百甚至更多物体，对于玩家来说，这就从其 AOI 中每个物体每次发生状态变更都会产生一条推送消息，变成了最多每 100 或 200 毫秒收到一条推送消息（之中包含了这段时间内，其 AOI 范围内所有物体的状态变更通知）。这中批量打包技术无疑大大减少了消息推送的次数，同时也有效提高了网络利用率，提升了整体通信效率。

  消息打包时间片长度的实际值可随业务逻辑需要来动态调整，例如：对于实时性要求较高的战斗、竞赛等特殊场合，可以适当缩短实际参与竞技玩家的时间片长度，甚至临时关闭批量打包机制（将之间片长度置零），以优化实时性表现。与此同时，观众和裁判等玩家依然可维持较大时间片，以尽量优化网络 IO。

综上所述，本方案分别从单点性能、集群架构、业务模式和消息推送算法四个方面针对网游服务器端的架构、模式和算法进行了全面优化，为游戏行业提供了一整套高性能、高可用、高负载的服务器端整体解决方案。

3.2 nano-SOA 架构

3.2.1 SOA vs. AIO

长久以来，服务器端的高层架构大体被区分为对立的两类：SOA（Service-oriented architecture）以及 AIO（All in one）。SOA 将一个完整的应用分割为相互独立的服务，每个服务提供一个单一标准功能（如：会话管理、交易评价、用户积分等等）。服务间通过 RPC、WebAPI 等 IPC 机制暴露功能接口，并以此相互通信，最终组合成一个完整的应用。

而 AIO 则相反，它将一个应用规约在一个独立的整体中，SOA 中的不同服务在 AIO 架构下呈现为不同的功能组件和模块。AIO 应用的所有组件通常都运行在一个地址空间（同一进程）内，所有组件的代码也常常放在同一个产品项目中一起维护。

AIO 的优势是部署简单，不需要分别部署多个服务，并为每个服务实现一套高可用集群。与此同时，由于可避免网络传输、内存拷贝等 IPC 通信所带来的大量开销，因此 AIO 架构的单点效率通常远高于 SOA。

另一方面，由于 AIO 架构中组件依赖性强，组件间经常知晓并相互依赖对方的实现细节，因此组件的可重用性及可替换性差，维护和扩展也较困难。特别是对于刚加入团队的新人来说，面对包含了大量互相深度耦合之组件和模块的“巨型项目”，常常需要花费大量努力、经历很多挫折并且犯很多错误才能真正接手。而即使对于老手来说，由于模块间各自对对方实现细节错综复杂的依赖关系，也容易发生在修改了一个模块的功能后，莫名奇妙地影响到其它看起来毫不相干功能的情况。

与此相反，SOA 模型部署和配置复杂——现实中，一个大型应用常常被拆分为数百个相互独立的服务，《程序员》期刊中的一份公开发表的论文显示，某个国内“彻底拥抱”SOA 的著名（中国排名前 3）电商网站将他们的 Web 应用拆分成了一千多个服务。可以想象，在多活数据中心的高可用环境内部署成百上千个服务器集群，并且配置他们彼此间的协作关系是多大的工作量。最近的协程(ctrip.com)网络瘫痪事件也是因为上千个服务组成的庞大 SOA 架构导致故障恢复缓慢。

除了部署复杂以外，SOA 的另一个主要缺点就是低效——从逻辑流的角度看，几乎每次来自客户端的完整请求都需要依次流经多个服务后，才能产生最终结果并返回用户端。而请求（通过消息中间件）每“流经”一个服务都需要伴随多次网络 IO 和磁盘访问，多个请求可累计产生较高的网络时延，使用户请求的响应时间变得不可确定，用户体验变差，并额外消耗大量资源。

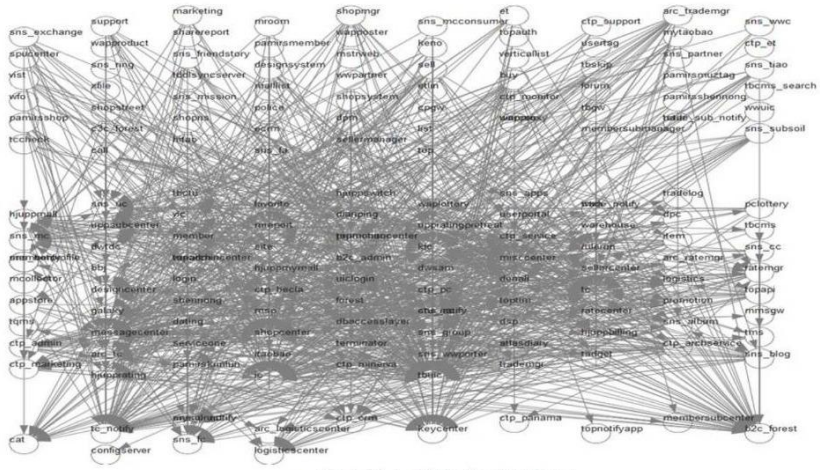

图 5 混乱的 SOA 依赖关系（图片来自互联网）

此外，无论是每个 Service 各自连接不同的 DBMS 还是它们分别接入同一个后端分布式 DBMS 系统，实现跨服务的分布式事务支持工作都要落到应用层开发者手中。而分布式事务（XA）本身的实现复杂度恐怕就已超过大部分普通应用了，更何况还需要为分布式事务加上高可靠和高可用保证——需要在单个数据切片上使用 Paxos/Raft 或主从+Arbiter 之类的高可用、强一直性算法，同时在涉及多个数据切片的事务上使用 2PC/3PC 等算法来保证事务的原子性。因此 SOA 应用中的跨 Service 事务基本都只能退而求其次，做到最终一致性保证，即便如此，也需要增加大量的额外工作——在稍微复杂点的系统里，高可用，并能在指定时间内可靠收敛的最终一致性算法实现起来也不是那么容易。

与此同时，大部分 SOA 系统还经常需要使用消息中间件来实现消息分发服务。如果对消息中间件的可用性（部分节点故障不会影响正常使用）、可靠性（即使在部分节点故障时，也确保消息不丢失、不重复、并严格有序）、功能性（如：发布/订阅模型、基于轮转的任务分发等）等方面有所要求的话，那么消息中间件本身也容易成为系统的瓶颈。

SOA 架构的优点在于其高内聚、低耦合的天然特性。仅通过事先约定的 IPC 接口对外提供服务，再配合服务间隔离（通常是在独立节点中）运行的特质，SOA 架构划分出了清晰的接口和功能边界，因此可以被非常容易地重用和替换（任何实现了兼容 IPC 接口的新服务都可替换已有的老服务）。

从软件工程和项目管理的视角来看，由于每个服务本身通常有足够高的内聚性，并且单个服务实现的功能也较独立，因此相对于 AIO 意大利面式的，相互交织的结构来说，SOA 的服务非常便于维护——负责某一服务的开发人员只需要看好自己这一亩三分地即可，只要保持服务对外提供的 API 没有发生不兼容的变化，就不需要担心修改代码、替换组件等工作会影响到其它“消费者”。

同时，由多个独立服务所组成的应用也更容易通过加入新服务和重新组合现有服务来进行功能变更和扩展。

3.2.2 nano-SOA 架构

在经历了大量实际项目中的权衡、思索和实践后，我逐步定义、实现和完善了能够兼两者之长的“nano-SOA”架构。并将功能插件（IPlugin）、数据库连接器（DBC）API Nexus 等关键性的基础功能组件封装在了《白杨应用支撑平台》中，以提供便于重用的高品质基础架构。

在 nano-SOA 架构中，独立运行的服务被替换成了支持动态插拔的跨平台功能插件（IPlugin）；而插件则通过（并仅可通过）API Nexus 来动态地暴露（注册）和隐藏（注销）自身所提供的功能接口，同时也使用 API Nexus 来消费其它插件提供服务。

nano-SOA 完全继承了 SOA 架构高内聚、低耦合的优点，每个插件如独立的服务一样，有清晰的接口和边界，可容易地被替换和重用。在可维护性上，nano-SOA 也与 SOA 完全一致，每个插件都可以被单独地开发和维护，开发人员只需要管好自己维护的功能插件即可。通过加入新插件以及对现有功能插件的重新组合，甚至可比 SOA 模式更容易地对现有功能进行变更和扩展。

而在性能方面，由于所有功能插件都运行在同一个进程内，因此通过 API Nexus 的相互调用不需要任何网络 IO、磁盘访问和内存拷贝，也没有任何形式的其它 IPC 开销，因此其性能和效率均可与 AIO 架构保持在相同量级。

与此同时，nano-SOA 的部署与 AIO 同样简单——部署在单个节点即可使用，只需部署一个集群即可实现高可用和横向扩展。在配置方面也远比 SOA 简单，仅需要比 AIO 应用多配置一个待加载模块列表而已，并且这些配置也可通过各种配置管理产品来实现批量维护。简单的部署和配置过程不但简化了运营和维护工作，也大大方便了开发和测试环境的构建。

此外，nano-SOA 也在极大程度上避免了对消息中间件的依赖，取而代之的是通过 API Nexus 的直接 API 调用；或是在需要削峰填谷的场合中，使用由内存零拷贝和无锁算法高度优化的线程间消息队列。这一方面大大增加了吞吐，避免了延迟，另一方面也避免了部署和维护一个高可用的消息分发服务集群所带来的巨大工作量——nano-SOA 集群内的节点间协作和协调通信需求已被将至最低，对消息分发的可靠性、可用性和功能性都没有太高要求。在多数情况下，使用 Gossip Protocol 等去中心化的 P2P 协议即足以满足需要，有时甚至可以完全避免这种集群内的节点间通信。

从 nano-SOA 的角度看，也可以将 DBC 视作一种几乎所有服务器端应用都需要使用的基础功能插件，由于其常用性，因此他们被事先实现并加进了 libapidbc 中。由此，通过提供 IPlugin、API Nexus 以及 DBC 等几个关键组件，libapidbc 为 nano-SOA 架构奠定了良好的基础设施。

当然，nano-SOA 与 SOA 和 AIO 三者间并不是互斥的选择。在实际应用场景中，可以通过三者间的有机组合来达成最合理的设计。例如：对于视频转码等非常耗时并且不需要同步等待其完成并返回结果的异步操作来说，由于其绝大部分开销都耗费在了视频编解码计算上，因此将其作为插件加入其它 App Server 就完全没有必要，将它作为独立的服务，部署在配置了专用加速硬件的服务器集群上应该是更好的选择。

3.2.3 白杨消息端口交换服务（BYPSS）

白杨消息端口交换服务（BYPSS，读作“bypass”）设计用于单点支撑百亿量级端口、十万量级节点规模，每秒处理百万至千万量级消息的高可用、强一致、高性能分布式协调和消息交换服务。其中关键概念包括：

* 连接（Connection）：每个客户端（应用集群中的服务器）节点至少与端口交换服务保持一个 TCP 长连接。

* 端口（Port）：每个连接上可以注册任意多个消息端口，消息端口由一个 UTF-8 字符串描述，必须在全局范围内唯一，若其它客户端节点已注册了相同的消息端口，则端口注册失败。

端口交换服务对外提供的 API 原语包括：

* 等待消息（WaitMsg）：客户端集群中的每个节点均应保持一个到端口交换服务的 TCP 长连接，并调用此方法等待消息推送。此方法将当前客户端连接由消息发送连接升级为消息接收连接。

  每个节点号只能对应一个消息接收连接，若一个节点尝试同时发起两个消息接收连接，则较早的那个接收连接将被关闭，并且绑定到当前节点上的所有端口都将被注销。

* 续租（Relet）：若端口交换服务在指定的间隔内未收到来自某个消息接收连接的续租请求，则判定该节点已经下线，并释放所有属于该节点的端口。续租操作用来周期性地向端口交换服务提供心跳信号。

* 注册端口（RegPort）：连接成功建立后，客户端应向端口交换服务注册所有属于当前节点的消息端口。可以在一个端口注册请求中包含任意多个待注册端口，端口交换服务会返回所有注册失败（已被占用）的端口列表。调用者可以选择是否需要为注册失败的端口订阅端口注销通知。

  需要注意的是，每次调用 WaitMsg 重建消息接收连接后，都需要重新注册当前节点上的所有端口。

* 注销端口（UnRegPort）：注销数据当前节点的端口，可一次提交多个端口执行批量注销。

* 消息发送（SendMsg）：向指定的端口发送消息（BLOB），消息格式对交换服务透明。若指定的端口为空串，则向端口交换服务上的所有节点广播此消息；亦可同时指定多个接收端口，实现消息组播。若指定的端口不存在，则安静地丢弃该消息。客户端可在一次请求中包含多个消息发送命令，主动执行批量发送，服务器端也会将发往同一节点的消息自动打包，实现消息批量推送。

* 端口查询（QueryPort）：查询当前占用着指定端口的节点号，及其 IP 地址。此操作主要用于实现带故障检测的服务发现，消息投递时已自动执行了相应操作，故无需使用此方法。可在同一请求中包含多个端口查询命令，执行批量查询。

* 节点查询（QueryNode）：查询指定节点的 IP 地址等信息。此操作主要用于实现带故障检测的节点解析。可以一次提交多个节点，实现批量查询。

端口交换服务的客户端连接分为以下两类：

* **消息接收连接（1:1）**：接收连接使用 WaitMsg 方法完成节点注册并等待消息推送，同时通过 Relet 接口保持属于该节点的所有端口被持续占用。客户端集群中的每个节点应当并且仅能够保持一个消息接收连接。此连接为长连接，由于连接中断重连后需要重新进行服务选举（注册端口），因此应尽可能一直保持该连接有效并及时完成续租。

* **消息发送连接（1:N）**：所有未使用 WaitMsg API 升级的客户端连接均被视为发送连接，发送连接无需通过 Relet 保持心跳，仅使用 RegPort、UnRegPort、SendMsg 以及 QueryPort 等原语完成非推送类的客户端请求。客户端集群中的每个节点通常都会维护一个消息发送连接池，以方便各工作线程高效地与端口转发服务保持通信。

与传统的分布式协调服务以及消息中间件产品相比，端口转发服务主要有以下特点：

* 功能性：端口转发服务将标准的消息路由功能集成到了服务选举（注册端口）、服务发现（发送消息和查询端口信息）、故障检测（续租超时）以及分布式锁（端口注册和注销通知）等分布式协调服务中。是带有分布式协调能力的高性能消息转发服务。通过 QueryPort 等接口，也可以将其单纯地当作带故障检测的服务选举和发现服务来使用。

* 高并发、高性能：由 C/C++/汇编实现；为每个连接维护一个消息缓冲队列，将所有端口定义及待转发消息均保存在内存中（Full in-memory）；主从节点间无任何数据复制和状态同步开销；信息的发送和接收均使用纯异步 IO 实现，因而可提供高并发和高吞吐的消息转发性能。

* 可伸缩性：在单点性能遭遇瓶颈后，可通过级联上级端口交换服务来进行扩展（类似 IDC 接入、汇聚、核心等多层交换体系）。

* 可用性：两秒内完成故障检测和主备切换的高可用保证，基于多数派的选举算法，避免由网络分区引起的脑裂问题。

* 一致性：保证任意给定时间内，最多只有一个客户端节点可持有某一特定端口。不可能出现多个客户端节点同时成功注册和持有相同端口的情况。

* 可靠性：所有发往未注册（不存在、已注销或已过期）端口的消息都将被安静地丢弃。系统保证所有发往已注册端口消息有序且不重复，但在极端情况下，可能发生消息丢失：

- 端口交换服务宕机引起主从切换：此时所有在消息队列中排队的待转发消息均会丢失；所有已注册的客户端节点均需要重新注册；所有已注册的端口（服务和锁）均需要重新进行选举/获取（注册）。

- 客户端节点接收连接断开重连：消息接收连接断开或重连后，所有该客户端节点之前注册的端口均会失效并需重新注册。在接收连接断开到重连的时间窗口内，所有发往之前与该客户端节点绑定的，且尚未被其它节点重新注册的端口之消息均被丢弃。

可见，白杨消息端口转发服务本身是一个集成了故障检测、服务选举、服务发现和分布式锁等分布式协调功能的消息路由服务。它通过牺牲极端条件下的可靠性，在保证了强一致、高可用、可伸缩（横向扩展）的前提下，实现了极高的性能和并发能力。

可以认为消息端口交换服务就是为 μSOA 架构量身定做的集群协调和消息分发服务。μSOA 的主要改进即：将在 SOA 中，每个用户请求均需要牵扯网络中的多个服务节点参与处理的模型改进为大部分用户请求仅需要同一个进程空间内的不同 BMOD 参与处理。

这样的改进除了便于部署和维护，以及大大降低请求处理延迟外，还有两个主要的优点：

* 将 SOA 中，需要多个服务节点参与的分布式事务或分布式最终一致性问题简化成为了本地 ACID Transaction 问题（从应用视角来看是如此，对于分布式 DBS 来说，以 DB 视角看来，事务仍然可以是分布式的），这不仅极大地简化了分布式应用的复杂度，增强了分布式应用的一致性，也大大减少了节点间通信（由服务间的 IPC 通信变成了进程内的指针传递），提高了分布式应用的整体效率。

* 全对等节点不仅便于部署和维护，还大大简化了分布式协作算法。同时由于对一致性要求较高的任务都已在同一个进程空间内完成，因此节点间通信不但大大减少，而且对消息中间件的可靠性也不再有过高的要求（通常消息丢失引起的不一致可简单地通过缓存超时或手动刷新来解决，可确保可靠收敛的最终一致性）。

在此前提下，消息端口交换服务以允许在极端情况下丢失少量未来得及转发的消息为代价，来避免磁盘写入、主从复制等低效模式，以提供极高效率。这对 nano-SOA 来说是一种非常合理的选择。

3.2.3.1 极端条件下的可靠性

传统的分布式协调服务通常使用 Paxos 或 Raft 之类基于多数派的强一致分布式算法实现，主要负责为应用提供一个高可用、强一致的分布式元数据 KV 访问服务。并以此为基础，提供分布式锁、消息分发、配置共享、角色选举、服务发现、故障检测等分布式协调服务。常见的分布式协调服务实现包括 Google Chubby（Paxos）、Apache ZooKeeper（Fast Paxos）、etcd（Raft）、Consul（Raft+Gossip）等。

Paxos、Raft 等分布式一致性算法的最大问题在于其极低的访问性能和极高的网络开销：对这些服务的每次访问，无论读写，都会产生至少 2 到 4 网络广播——以投票的方式确定本次访问经过多数派确认（读也需要如此，因为主节点需要确认本次操作发生时，自己仍拥有多数票支持，仍是集群的合法主节点）。

在实践中，虽可通过降低系统整体一致性或加入租期机制来优化读操作的效率，但其总体性能仍十分低下，并且对网络 IO 有很高的冲击：Google、Facebook、Twitter 等公司的历次重大事故中，很多都是由于发生网络分区或人为配置错误导致 Paxos、Raft 等算法疯狂广播消息，致使整个网络陷入广播风暴而瘫痪。

此外，由于 Paxos、Raft 等分布式一致性算法对网络 IO 的吞吐和延迟等方面均有较高要求，而连接多座数据中心机房（IDC）的互联网络通常又很难满足这些要求，因此导致依赖分布式协调算法的强一致（抗脑裂）多活 IDC 高可用集群架构难以以合理成本实现。作为实例：2015 年 8 月 20 日 Google GCE 服务中断 12 小时并永久丢失部分数据；2015 年 5 月 27 日和 2016 年 7 月 22 日支付宝两次中断数小时；2013 年 7 月 22 日微信服务中断数小时；以及 2017 年 5 月英国航空瘫痪数日等重大事故均是由于单个 IDC 因市政施工（挖断光纤）等原因下线，同时未能成功构建多活 IDC 架构，因此造成 IDC 单点依赖所导致的。

前文也已提到过：由于大部分采用 SOA 架构的产品需要依赖消息中间件来确保系统的最终一致性。因此对其可用性（部分节点故障不会影响正常使用）、可靠性（即使在部分节点故障时，也确保消息不丢失、不重复、并严格有序）、功能性（如：发布/订阅模型、基于轮转的任务分发等）等方面均有较严格的要求。这就必然要用到高可用集群、节点间同步复制、数据持久化等低效率、高维护成本的技术手段。因此消息分发服务也常常成为分布式系统中的一大主要瓶颈。

与 Paxos、Raft 等算法相比，BYPSS 同样提供了故障检测、服务选举、服务发现和分布式锁等分布式协调功能，以及相同等级的强一致性、高可用性和抗脑裂（Split Brain）能力。在消除了几乎全部网络广播和磁盘 IO 等高开销操作的同时，提供了数千、甚至上万倍于前者的访问性能和并发处理能力。可在对网络吞吐和延迟等方面无附加要求的前提下，构建跨多个 IDC 的大规模分布式集群系统。

与各个常见的消息中间件相比，BYPSS 提供了一骑绝尘的单点百万至千万条消息每秒的吞吐和路由能力——同样达到千百倍的性能提升，同时保证消息不重复和严格有序。

然而天下没有免费的午餐，特别是在分布式算法已经非常成熟的今天。在性能上拥有绝对优势的同时，BYPSS 必然也有其妥协及取舍——BYPSS 选择放弃极端（平均每年 2 次，并且大多由维护引起，控制在低谷时段，基于实际生产环境多年统计数据）情形下的可靠性，对分布式系统的具体影响包括以下两方面：

* 对于分布式协调服务来说，这意味着每次发生 BYPSS 主节点故障掉线后，所有的已注册端口都会被强制失效，所有活动的端口都需要重新注册。

  例如：若分布式 Web 服务器集群以用户为最小调度单位，为每位已登陆用户注册一个消

息端口。则当 BYPSS 主节点因故障掉线后，每个服务器节点都会得知自己持有的所有端口均已失效，并需要重新注册当前自己持有的所有活动（在线）用户。

幸运的是，该操作可以被批量化地完成——通过批量端口注册接口，可在一次请求中同时提交多达数百万端口的注册和注销操作，从而大大提升了请求处理效率和网络利用率：在 2013 年出厂的至强处理器上（Haswell 2.0GHz），BYPSS 服务可实现每核（每线程）100 万端口/秒的处理速度。同时，得益于我方自主实现的并发散列表（每个 arena 都拥有专属的汇编优化用户态读者/写者高速锁），因此可通过简单地增加处理器核数来实现处理能力的线性扩展。

具体来说，在 4 核处理器+千兆网卡环境下，BYPSS 可达成约每秒 400 万端口注册的处理能力；而在 48 核处理器+万兆网卡环境下，则可实现约每秒 4000 万端口注册的处理能力（测试时每个端口名称的长度均为 16 字节），网卡吞吐量和载荷比都接近饱和。再加上其发生概率极低，并且恢复时只需要随着对象的加载来逐步完成重新注册，因此对系统整体性能几乎不会产生什么波动。

为了说明这个问题，考虑 10 亿用户同时在线的极端情形，即使应用程序为每个用户分别注册一个专用端口（例如：用来确定用户属主、完成消息分发等），那么在故障恢复后的第一秒内，也不可能出现“全球 10 亿用户心有灵犀地同时按下刷新按钮”的情况。相反，基于 Web 等网络应用的固有特性，这些在线用户通常要经过几分钟、几小时甚至更久才会逐步返回服务器（同时在线用户数=每秒并发请求数 x 用户平均思考时间）。即使按照比较严苛的“1 分钟内全部返回”（平均思考时间 1 分钟）来计算，BYPSS 服务每秒也仅需处理约 1600 万条端口注册请求。也就是说，一台配备了 16 核至强处理器和万兆网卡的入门级 1U PC Server 即可满足上述需求。

作为对比实例：官方数据显示，淘宝网 2015 年双十一当天的日活用户数（DAU）为 1.8 亿，同时在线用户数峰值为 4500 万。由此可见，目前超大型站点瞬时并发用户数的最高峰值仍远低于前文描述的极端情况。即使再提高数十倍，BYPSS 也足可轻松支持。

* 另一方面，对于消息路由和分发服务来说，这意味着每次发生 BYPSS 主节点故障掉线后，所有暂存在 BYPSS 服务器消息队列中，未及发出的待转发消息都将永久丢失。可喜的是，μSOA 架构不需要依赖消息中间件来实现跨服务的事务一致性。因此对消息投递的可靠性并无严格要求。

  意即：μSOA 架构中的消息丢失最多导致对应的用户请求完全失败，此时仍可保证数据的全局强一致性，绝不会出现“成功一半”之类的不一致问题。在绝大多数应用场景中，这样的保证已经足够——即使支付宝和四大行的网银应用也会偶尔发生操作失败的问题，这时只要资金等帐户数据未出现错误，那么稍候重试即可。

  此外，BYPSS 服务也通过高度优化的异步 IO，以及消息批量打包等技术有效降低了消息在服务器队列中的等待时间。具体来说，这种消息批量打包机制由消息推送和消息发送机制两方面组成：

BYPSS 提供了消息批量发送接口，可在一次请求中同时提交数以百万计的消息发送操作，从而大大提升了消息处理效率和网络利用率。另一方面，BYPSS 服务器也实现了消息批量打包推送机制：若某节点发生消息浪涌，针对该节点的消息大量到达并堆积在服务器端消息队列中。则 BYPSS 服务器会自动开启批量消息推送模式——将大量消息打包成一次网络请求，批量推送至目的节点。

通过上述的批量处理机制，BYPSS 服务可大大提升消息处理和网络利用效率，确保在大部分情况下，其服务器端消息队列基本为空，因此就进一步降低了其主服务器节点掉线时，发生消息丢失的概率。

然而，虽然消息丢失的概率极低，并且 nano-SOA 架构先天就不怎么需要依赖消息中间件提供的可靠性保证。但仍然可能存在极少数对消息传递要求很高的情况。对于此类情况，可选择使用下列解决方案：

- 自行实现回执和超时重传机制：消息发送方对指定端口发送消息，并等待接收该消息处理回执。若在指定时段内未收到回执，则重新发送请求。

- 直接向消息端口的属主节点发起 RPC 请求：消息发送方通过端口查询命令获取该端口属主节点的 IP 地址等信息，并直接与该属主节点建立连接、提交请求并等待其返回处理结果。BYPSS 在此过程中仅担当服务选举和发现的角色，并不直接路由消息。对于视频推流和转码、深度学习等有大量数据流交换的节点间通信，也建议使用此方式，以免 BYPSS 成为 IO 瓶颈。

- 使用第三方的可靠消息中间件产品：若需要保证可靠性的消息投递请求较多，规则也较复杂，也可单独搭建第三方的可靠消息分发集群来处理这部分请求。

综上所述，可以认为 BYPSS 服务就是为 nano-SOA 架构量身定做的集群协调和消息分发服务。BYPSS 和 nano-SOA 架构之间形成了扬长避短的互补关系：BYPSS 以极端条件下系统整体性能的轻微波动为代价，极大提升了系统的总体性能表现。适合用来实现高效率、高可用、高可靠、强一致的 nano-SOA 架构分布式系统。

3.2.3.2 BYPSS 特性总结

BYPSS 和基于 Paxos、Raft 等传统分布式一致性算法的分布式协调产品特性对比如下：

项目	BYPSS	ZooKeeper、Consul、etcd…
可用性	高可用，支持多活 IDC	高可用，支持多活 IDC
一致性	强一致，主节点通过多数派选举	强一致，多副本复制
并发性	千万量级并发连接，可支持数十万并发节点	不超过 5000 节点
容量	每 10GB 内存可支持约 1 亿消息端口；每 1TB 内存可支持约 100 亿消息端口；两级并发散列	通常最高支持数万 KV 对。开启了变更通知时则更少。

项目	BYPSS	ZooKeeper、Consul、etcd…
	表结构确保容量可线性扩展至 PB 级。	
延迟	相同 IDC 内每次请求延迟在亚毫秒级（阿里云中实测为 0.5ms）；相同区域内的不同 IDC 间每次请求延迟在毫秒级（阿里云环境实测 2ms）。	由于每次请求需要至少 2 到 4 次网络广播和多次磁盘 IO，因此相同 IDC 中的每操作延迟在十几毫秒左右；不同 IDC 间的延迟则更长（详见下文）。
性能	每 1Gbps 网络带宽可支持约 400 万次/秒的端口注册和注销操作。在 2013 年出厂的入门级至强处理器上，每核心可支持约 100 万次/秒的上述端口操作。性能可通过增加带宽和处理器核心数量线性扩展。	算法本身的特性决定了无法支持批量操作，不到 100 次每秒的请求性能（由于每个原子操作都需要至少 2 到 4 次网络广播和多次磁盘 IO，因此支持批量操作毫无意义，详见下文）。
网络利用率	高：服务器端和客户端均具备端口注册、端口注销、消息发送、端口查询、节点查询等原语的批量打包能力，网络载荷比可接近 100%。	低：每请求一个独立包（TCP Segment、IP Packet、Network Frame），网络载荷比通常低于 5%。
可伸缩性	有：可通过级联的方式进行横向扩展。	无：集群中的节点越多（因为广播和磁盘 IO 的范围更大）性能反而越差。
分区容忍	无多数派分区时系统下线，但不会产生广播风暴。	无多数派分区时系统下线，有可能产生广播风暴引发进一步网络故障。
消息分发	有，高性能，客户端和服务器均包含了消息的批量自动打包支持。	无。
配置管理	无，BYPSS 认为配置类数据应交由 Redis、MySQL、MongoDB 等专门的产品来维护和管理。当然，这些 CMDB 的主从选举等分布式协调工作仍可由 BYPSS 来完成。	有，可当作简单的 CMDB 来使用，这种功能和职责上的混淆不清进一步劣化了产品的容量和性能。
故障恢复	需要重新生成状态机，但可以数千万至数亿端口/秒的性能完成。实际使用中几无波动。	不需要重新生成状态机。

上述比较中，延迟和性能两项主要针对写操作。这是因为在常见的分布式协调任务中，几乎全部有意义的操作都是写操作。例如：

操作	对服务协调来说	对分布式锁来说
端口注册	成功：服务选举成功，成为该服务的属主。 失败：成功查询到该服务的当前属主。	成功：上锁成功。 失败：上锁失败，同时返回锁的当前属主。
端口注销	放弃服务所有权。	释放锁。
注销通知	服务已下线，可更新本地查询缓存或参与服务竞选。	锁已释放，可重新开始尝试上锁。

上表中，BYPSS 的端口注册对应 ZooKeeper 等传统分布式产品中的“写/创建 KV 对”；端口注销对应“删除 KV 对”；注销通知则对应“变更通知”服务。

由此可见，为了发挥最高效率，在生产环境中通常不会使用单纯的查询等只读操作。而是将查询操作隐含在端口注册等写请求中，请求成功则当前节点自身成为属主；注册失败自然会返回

请求服务的当前属主，因此变相完成了属主查询（服务发现/名称解析）等读操作。

需要注意的是，就算是端口注册等写操作失败，其实还是会伴随一个成功的写操作。因为仍然要将发起请求的当前节点加入到指定条目的变更通知列表中，以便在端口注销等变更事件发生时，向各个感兴趣的节点推送通知消息。因此写操作的性能差异极大地影响了现实产品的实际表现。

注：以上所述 nano-SOA 架构和 BYPSS 分布式协调算法均受到多项国家和国际发明专利保护。

3.2.3.3 基于 BYPSS 的高性能集群

从高性能集群（HPC）的视角来看，BYPSS 与前文所述的传统分布式协调产品之间，最大的区别主要体现在以下两个方面：

1. 高性能：BYPSS 通过消除网络广播、磁盘 IO 等开销，以及增加批处理支持等多种优化手段使分布式协调服务的整体性能提升了上万倍。

2. 大容量：约每 10GB 内存 1 亿个消息端口的容量密度，由于合理使用了并发散列表等数据结构，使得容量和处理性能可随内存容量、处理器核心数量以及网卡速率等硬件升级而线性扩展。

由于传统分布式协调服务的性能和容量等限制，在经典的分布式集群中，多以服务或节点作为单位来进行分布式协调和调度，同时尽量要求集群中的节点工作在无状态模式。服务节点无状态的设计虽然对分布式协调服务的要求较低，但同时也带来了集群整体性能低下等问题。

与此相反，BYPSS 可轻松实现每秒千万次请求的处理性能和百亿至千亿量级的消息端口容量。这就给分布式集群的精细化协作构建了良好的基础。与传统的无状态集群相比，基于 BYPSS 的精细化协作集群能够带来巨大的整体性能提升。

我们首先以最常见的用户和会话管理功能来说明：在无状态的集群中，在线用户并无自己的属主服务器，用户的每次请求均被反向代理服务随机地路由至集群中的任意节点。虽然 LVS、Nginx、HAProxy、TS 等主流反向代理服务器均支持基于 Cookie 或 IP 等机制的节点粘滞选项，但由于集群中的节点都是无状态的，因此该机制仅仅是增加了相同客户端请求会被路由到某个确定后台服务器节点的概率而已，仍无法提供所有权保证，也就无法实现进一步的相关优化措施。

而得益于 BYPSS 突出的性能和容量保证，基于 BYPSS 的集群可以用户为单位来进行协调和调度（即：为每个**活动用户**注册一个端口），以提供更优的整体性能。具体的实现方式为：

1. 与传统模式一样，在用户请求到达反向代理服务时，由反向代理通过 HTTP Cookie、IP 地址或自定义协议中的相关字段等方式来判定当前请求应该被转发至哪一台后端服务器节点。若请求中尚无粘滞标记，则选择当前负载最轻的一个后端节点来处理。

2. 服务器节点在收到用户请求后，在本地内存表中检查该用户的属主是否为当前节点。

   a) 若当前节点已是该用户属主，则由此节点继续处理用户请求。

   b) 若当前节点不是该用户的属主，则向 BYPSS 发起 RegPort 请求，尝试成为该用户的属主。此请求应使用批量方式发起，以进一步提高网络利用率和处理效率。

      i. 若 RegPort 请求成功，说明当前节点已成功获取该用户的所有权，此时可将用户信息由后端数据库加载到当前节点的本地缓存中（应使用批量加载优化），并继续处理此用户相关请求。

      ii. 若 RegPort 请求失败，说明指定用户正归于另一个节点管辖，此时应重新设置反向代理能够识别的 Cookie 等粘滞字段，将其指向正确的属主节点。并要求反向代理服务或客户端重试请求。

与传统架构相比，考虑到无状态服务也需要通过 MySQL、Memcached 或 Redis 等技术来实现专门的用户和会话管理机制，因此以上实现并未增加多少复杂度，但是其带来的性能提升却非常巨大，对比如下：

项目	BYPSS HPC 集群	传统无状态集群
1 运维	省去用户和会话管理集群的部署和维护成本。	需要单独实施和维护用户管理集群，并为用户和会话管理服务提供专门的高可用保障，增加故障点、增加系统整体复杂性、增加运维成本。
2 网络	几乎所有客户请求的用户匹配和会话验证工作都得以在其属主节点的内存中直接完成。内存访问为纳秒级操作，对比毫秒级的网络查询延迟，性能提升十万倍以上。同时有效降低了服务器集群的内部网络负载。	每次需要验证用户身份和会话有效性时，均需要通过网络发送查询请求到用户和会话管理服务，并等待其返回结果，网络负载高、延迟大。 由于在一个典型的网络应用中，大部分用户请求都需要在完成用户识别和会话验证后才能继续处理，因此这对整体性能的影响很大。
3 缓存	因为拥有了稳定的属主服务器，而用户在某个时间段内总是倾向于重复访问相同或相似的数据（如自身属性，自己刚刚发布或查看的商品信息等）。因此服务器本地缓存的数据局部性强、命中率高。 相较于分布式缓存而言，本地缓存的优势非常明显： 1. 省去了查询请求所需的网络延迟，降低了网络负载（详见本表“项目 2”中的描述）。	无专属服务器，用户请求随机到达集群中的任意服务器节点；本地缓存命中率低；各节点重复缓存的内容多；需要以更高的成本为代价依赖分布式缓存。 后端数据库服务器的读压力高，要对其进行分库分表、读写分离等额外优化。

项目	BYPSS HPC 集群	传统无状态集群
	2. 直接从内存中读取已展开的数据结构，省去了大量的数据序列化和反序列化工作。 与此同时，如能尽量按照某些规律来分配用户属主，还可进一步地提升服务器本地缓存的命中率。例如： a) 按租户（公司、部门、站点）来分组用户； b) 按区域（地理位置、游戏中的地图区域）来分组用户； c) 按兴趣特征（游戏战队、商品偏好）来分组用户。 等等，然后尽量将属于相同分组的用户优先分配给同一个（或同一组）服务器节点。显而易见，选择合适的用户分组策略可极大提升服务器节点的本地缓存命中率。 这使得绝大部分与用户或人群相关的数据均可在本地缓存命中，不但提升了集群整体性能，还消除了集群对分布式缓存的依赖，同时大大降低了后端数据库的读负载。	
4 更新	由于所有权确定，能在集群全局确保任意用户在给定时间段内，均由特定的属主节点来提供服务。再加上现代服务器突发宕机故障的概率也较低。 因此可以将用户属性中频繁变化但重要性或时效性较低的部分缓存在内存中，待积累一段时间后再批量更新至数据库。这可大大降低后端数据库服务器的写压力。 例如：商城系统可能随着用户的浏览（比如每次查看商品）进程，随时收集并记录用户的偏好信息。若每次用户查看了新商品后，都需要即时更新数据库，则负载较高。再考虑到因为服务器偶发硬件故障导致丢失最后数小时商品浏览偏好数据完全可以接受，因此可由属主节点将这些数据临时保存在本地缓存中，每积累数小时再批量更新一次数据库。 再比如：MMORPG 游戏中，用户的当前位置、状态、	由于用户的每次请求都可能被转发到不同服务器节点来处理，因此无法实现累积写入优化和批量写入优化。后端数据库的写负担非常重。 存在多个节点争抢更新同一条记录的问题，进一步加重了数据库负担。 为此要对后端数据库进行额外的分库分表等优化，还会引发“需要由应用层来自行处理分布式事务”等副作用。

项目	BYPSS HPC 集群	传统无状态集群
	经验值等数据随时都在变化。属主服务器同样可以将这些数据变化积累在本地缓存中，并以适当的间隔（比如：每 5 分钟一次）批量更新到数据库中。 这不但极大地降低了后端数据库要执行的请求数量，而且将多个用户的数据在一个批量事务中打包更新也大大减少数据库操作时的磁盘刷新动作，进一步提升了效率。 此外，由专门的属主节点发起对用户属性的更新也避免了无状态集群中多个节点同时请求更新同一对象时的争抢问题，进一步提高了数据库性能。	
5 推送	由于同一用户发起的所有会话均被集中在同一个属主节点内统一管理，因此可非常方便地向用户推送即时通知消息（Comet）。 若发送消息的对象与消息接消息的收用户处于相同节点，则可直接将该消息推送给收件人麾下的所有活动会话。 否则只需将消息定向投递到收件人的属主节点即可。消息投递可使用 BYPSS 实现（直接向收件人对应端口发消息，应启用消息批量发送机制来优化），亦可通过专用的消息中间件（如：Kafka、RocketMQ、RabbitMQ、ZeroMQ 等）来完成。 若按照本表“项目 3”中描述的方法，优先将关联更紧密的用户分配到相同属主节点的话，则可大大提升消息推送在相同节点内完成的概率，此举可显著降低服务器间通信的压力。 因此我们鼓励针对业务的实际情况来妥善定制用户分组策略，合理的分组策略可实现让绝大部分消息都在当前服务器节点内本地推送的理想效果。 例如：对游戏类应用，可按地图对象分组，将处于相同地图副本内的玩家交由同一属主节点进行管理——传统 MMORPG 中的绝大部分消息推送都发生在同一地图副本内的玩家之间（AOI 范围）。 再比如：对于 CRM、HCM、ERP 等 SaaS 应用来说，	由于同一用户的不同会话被随机分配到不同节点处理，因此需要开发、部署和维护专门的消息推送集群，同时专门确保该集群的高性能和高可用性。 这不但增加了开发和运维成本，而且由于需要将每条消息先投递到消息推送服务后，再由该服务转发给客户端，因此也加重了服务器集群的内部网络负载，同时也加大了用户请求的处理延迟。

项目	BYPSS HPC 集群	传统无状态集群
	可按照公司来分组，将隶属于相同企业的用户集中到同一属主节点上——很显然，此类企业应用中，近 100%的通信都来自于企业内部成员之间。 这样即可实现近乎 100%的本地消息推送率，达到几乎消除了服务器间消息投递的效果，极大地降低了服务器集群的内部网络负载。	
6 平衡	集群可使用主被动负载平衡相结合的手段进行调度。 被动平衡：集群中的每个节点均会定期将其麾下不再活跃的用户和会话卸载掉，同时批量通知 BYPSS 服务释放这些用户所对应的端口。此算法实现了宏观上的负载平衡（以较长的时间周期来说，集群是平衡的）。 主动平衡：集群会通过 BYPSS 服务推选出负载平衡协调节点，该节点连续监视集群中各个节点的负载情况，并主动发出指令进行负载调度（如：要求 A 节点将其麾下 5000 位用户的所有权转移给 B 节点）。不同于宏观层面的被动平衡，主动平衡机制可以在更短的时间片内，以迅捷的反应速度来达成集群的快速配平。 主动平衡通常在集群中的部分节点刚刚从故障中恢复（因此处于空载状态）时效果明显，它比被动平衡反应更加迅速。如：在一个多活 IDC 集群中，某个 IDC 的光缆故障刚刚被修复而恢复上线时。	若启用了反向代理中的节点粘滞选项，则其负载平衡性与 BYPSS 集群的被动平衡算法相当。 若未启用反向代理中的节点粘滞选项，则在从故障中恢复时，其平衡性低于 BYPSS 主动平衡集群。与此同时，为了保证本地缓存命中率等其它性能指标不被过分劣化，管理员通常不会禁用节点粘滞功能。 另外，SOA 架构的多个服务间，容易产生负载不平衡，出现一些服务超载，另一些轻载的情况，nano-SOA 集群则无此弊端。

值得一提的是，这样的精准协作算法并不会造成集群在可用性方面的任何损失。考虑集群中的某个节点因故障下线的情况：此时 BYPSS 服务会检测到节点已下线，并自动释放属于该节点的所有用户。待其用户向集群发起新请求时，该请求会被路由到当前集群中，负载最轻的节点。这个新节点将代替已下线的故障节点，成为此用户的属主，继续为该用户提供服务（见前文中的步骤 2-b-i）。此过程对用户透明，不需要在客户端中加入额外的处理逻辑。

上述讨论以几乎所有网络应用中都会涉及的用户和会话管理功能为例，为大家展示了 BYPSS HPC 集群精细协调能力的优势。但在多数真实应用中，并不只有用户管理功能。除此之外，应用中通常还会包含可供其用户操作的其它对象。例如在优酷、土豆、youtube 等视频网站中，除了用户以外，至少还有“可供播放的视频”这种对象。

下面我们就以“视频对象”为例，探讨如何使用 BYPSS 的精细化调度能力来大幅提升集群

性能。

在这个假想的视频点播类应用中，与前文描述的用户管理功能类似，我们首先通过 BYPSS 服务为每个**活动的视频对象**选取一个属主节点。其次，我们将视频对象的属性分为以下两大类：

1. 普通属性：包含了那些较少更新，并且尺寸较小的属性。如：视频封面和视频流数据在 S3 / OSS 等对象存储服务中的 ID、视频标题、视频简介、视频标签、视频作者 UID、视频发布时间等等。这些属性均符合读多写少的规律，其中大部分字段甚至在视频正式发布后就无法再做修改。

   对于这类尺寸小、变化少的字段，可以将其分布在当前集群中，各个服务器节点的本地缓存内。本地缓存有高性能、低延迟、无需序列化等优点，加上缓存对象较小的尺寸，再配合用户分组等进一步提升缓存局部性的策略，可以合理的内存开销，有效地提升应用整体性能（详见下文）。

2. 动态属性：包含了所有需要频繁变更，或尺寸较大的属性。如：视频的播放次数、点赞次数、差评次数、平均得分、收藏数、引用次数，以及视频讨论区内容等。

   我们规定这类尺寸较大（讨论区内容）或者变化较快（播放次数等）的字段只能由该视频对象的属主节点来访问。其它非属主节点如需访问这些动态属性，则需要将相应请求提交给对应的属主节点来进行处理。

   意即：通过 BYPSS 的所有权选举机制，我们将那些需要频繁变更（更新数据库和执行缓存失效），以及那些占用内存较多（重复缓存代价高）的属性都交给对应的属主节点来管理和维护。这就形成了一套高效的分布式计算和分布式缓存机制，大大提升了应用整体性能（详见下文）。

此外，我们还规定对视频对象的任何写操作（不管是普通属性还是动态属性）均必须交由其属主来完成，非属主节点只能读取和缓存视频对象的普通属性，不能读取动态属性，也不能执行任何更新操作。

由此，我们可以简单地推断出视频对象访问的大体业务逻辑如下：

1. 在普通属性的读取类用户请求到达服务器节点时，检查本地缓存，若命中则直接返回结果，否则从后端数据库读取视频对象的普通属性部分并将其加入到当前节点的本地缓存中。

2. 在更新类请求或动态属性读取类请求到达服务器节点时，通过本地内存表检查当前节点是否为对应视频对象的属主。

   a) 若当前节点已是该视频的属主，则由当前节点继续处理用户请求：读操作直接从当前节点的本地缓存中返回结果；写操作视情形累积在本地缓存中，或直接提交给后

端数据库并更新本地缓存。

b) 若当前节点不是该视频的属主，但在当前节点的名称解析缓存表中找到了与该视频匹配的条目，则将当前请求转发给对应的属主节点。

c) 若当前节点不是该视频的属主，同时并未在当前节点的名称解析缓存表中查找到对应的条目，则向 BYPSS 发起 RegPort 请求，尝试成为该视频的属主。此请求应使用批量方式发起，以进一步提高网络利用率和处理效率。

i. 若 RegPort 请求成功，说明当前节点已成功获取该视频的所有权，此时可将视频信息由后端数据库加载到当前节点的本地缓存中（应使用批量加载优化），并继续处理此视频相关请求。

ii. 若 RegPort 请求失败，说明指定视频对象正归于另一个节点管辖，此时可将该视频及其对应的属主节点 ID 加入到本地名称解析缓存表中，并将请求转发给对应的属主节点来处理。

注意：由于 BYPSS 能够在端口注销时（无论是由于属主节点主动放弃所有权，还是该节点故障宕机），向所有对此事件感兴趣的节点推送通知。因此名称解析缓存表不需要类似 DNS 缓存的 TTL 超时淘汰机制，仅需在收到端口注销通知或 LRU 缓存满时删除对应条目即可。这不但能够大大增强查询表中条目的时效性和准确性，同时也有效地减少了 RegPort 请求的发送次数，提高了应用的整体性能。

与经典的无状态 SOA 集群相比，上述设计带来的好处如下：

项目	BYPSS HPC 集群	传统无状态集群
1 运维	基于所有权的分布式缓存架构，省去 Memcached、Redis 等分布式缓存集群的部署和维护成本。	需要单独实施和维护分布式缓存集群，增加系统整体复杂性。
2 缓存	普通属性的读操作在本地缓存命中，若使用“优先以用户偏好特征来分组”的用户属主节点分配策略，则可极大增强缓存局部性，增加本地缓存命中率，降低本地缓存在集群中各个节点上的重复率。 正如前文所述，相对于分布式缓存而言，本地缓存有消除网络延迟、降低网络负载、避免数据结构频繁序列化和反序列化等优点。 此外，动态属性使用基于所有权的分布式缓存来实现，避免了传统分布式缓存的频繁失效和数据不一致等问题。同时由于动态属性仅被缓存在属主节点上，因此也显著提升了系统整体的内存利用率。	无专属服务器，用户请求随机到达集群中的任意服务器节点；本地缓存命中率低；各节点重复缓存的内容多；需要以更高的成本为代价依赖额外的分布式缓存服务。 后端数据库服务器的读压力高，要对其实施分库分表、读写分离等额外优化。 此外，即使为 Memcached、Redis 等产品加入了基于 CAS 原子操作的 Revision 字段等改进，这些独立的分布式缓存集群仍无法提供数据强一致

项目	BYPSS HPC 集群	传统无状态集群
		保证（意即：缓存中的数据与后端数据库里的记录无法避免地可能发生不一致）。
3 更新	由于所有权确定，能在集群全局确保任意视频对象在给定时间段内，均由特定的属主节点来提供写操作和动态属性的读操作等相关服务，再加上现代服务器突发宕机故障的概率也较低。 因此可以将动态属性中频繁变化但重要性或时效性较低的部分缓存在内存中，待积累一段时间后再批量更新至数据库。这可大大降低后端数据库服务器的写压力。 例如：视频的播放次数、点赞次数、差评次数、平均得分、收藏数、引用次数等属性都会随着用户点击等操作密集地变化。若每次发生相关的用户点击事件后，都需要即时更新数据库，则负载较高。而在发生“属主节点由于硬件故障宕机”等极端情况时，丢失几分钟的上述统计数据完全可以接受。因此，我们可以将这些字段的变更积累在属主节点的缓存中，每隔数分钟再将其统一地批量写回后端数据库。 这不但极大地降低了后端数据库收到的请求数量，而且将多个视频的数据在一个批量事务中打包更新，也大大减少数据库操作时的磁盘刷新动作，进一步提升了效率。 此外，由专门的属主节点单独发起对视频记录的更新也避免了无状态集群中多个节点同时请求更新同一对象时的争抢问题，进一步提高了数据库性能。	由于每次请求都可能被路由到不同服务器节点来处理，因此无法实现累积写入优化和批量写入优化。后端数据库服务器的写负担非常重。存在多个节点争抢更新同一条记录的问题，这进一步加重了数据库负担。 为此要对后端数据库进行额外的分库分表等优化，还会引发“需要由应用层来自行处理分布式事务”等副作用。
4 平衡	集群可使用主被动负载平衡相结合的手段进行调度。 被动平衡：集群中的每个节点均会定期将其麾下不再活跃的视频对象卸载掉，同时批量通知 BYPSS 服务释放这些视频对应的端口。此算法实现了宏观上的负载平衡（以较长的时间周期来说，集群是平衡的）。	在从故障中恢复时，其平衡性低于 BYPSS 主动平衡集群。正常情况下则相差不大。 另外，SOA 架构的多个服务间，容易产生负载不平衡，出现一些服务超载，另一些轻载的情况，nano-SOA 集群则无此弊端。

项目	BYPSS HPC 集群	传统无状态集群
	主动平衡：集群会通过 BYPSS 服务推选出负载平衡协调节点，该节点连续监视集群中各个节点的负载情况，并主动发出指令进行负载调度（如：要求 A 节点将其麾下 10000 个视频对象的所有权转移给 B 节点）。不同于宏观层面的被动平衡，主动平衡机制可以在更短的时间片内，以迅捷的反应速度来达成集群的快速配平。 主动平衡通常在集群中的部分节点刚刚从故障中恢复（因此处于空载状态）时效果明显，它比被动平衡反应更加迅速。如：在一个多活 IDC 集群中，某个 IDC 的光缆故障刚刚被修复而恢复上线时。	

与前文提及的用户管理案例类似，上述精准协作算法不会为集群的服务可用性方面带来任何损失。考虑集群中的某个节点因故障下线的情况：此时 BYPSS 服务会检测到节点已下线，并自动释放属于该节点的所有视频对象。待用户下次访问这些视频对象时，收到该请求的服务器节点会从 BYPSS 获得此视频对象的所有权并完成对该请求的处理。至此，这个新节点将代替已下线的故障节点成为此视频对象的属主（见前文中的步骤 2-c-i）。此过程对用户透明，不需要在客户端中加入额外的处理逻辑。

以上对“用户管理”和“视频服务”案例的剖析均属抛砖引玉。在实际应用中，BYPSS 通过其高性能、大容量等特征提供的资源精细化协调能力可适用于包括互联网游戏、电信、物联网、大数据批处理、大数据流式计算等广泛领域。

3.2.4 白杨分布式消息队列（BYDMQ）

白杨分布式消息队列服务（BYDMQ，读作“by dark”）是一种强一致、高可用、高性能、高吞度、低延迟、可线性横向扩展的分布式消息队列服务。可支持单点千万量级的并发连接以及单点每秒千万量级的消息转发性能，并支持集群的线性横向扩展。

BYDMQ 自身亦依赖 BYPSS 来完成其服务选举、服务发现、故障检测、分布式锁、消息分发等分布式协调工作。BYPSS 虽然也包含了高性能的消息路由和分发功能，但其主要设计目还是为了传递任务调度等分布式协调相关的控制类信令。而 BYDMQ 则专注于高吞吐、低延迟的大量业务类消息投递等工作。将业务类消息转移到 BYDMQ 后，可使 BYPSS 的工作压力显著降低。

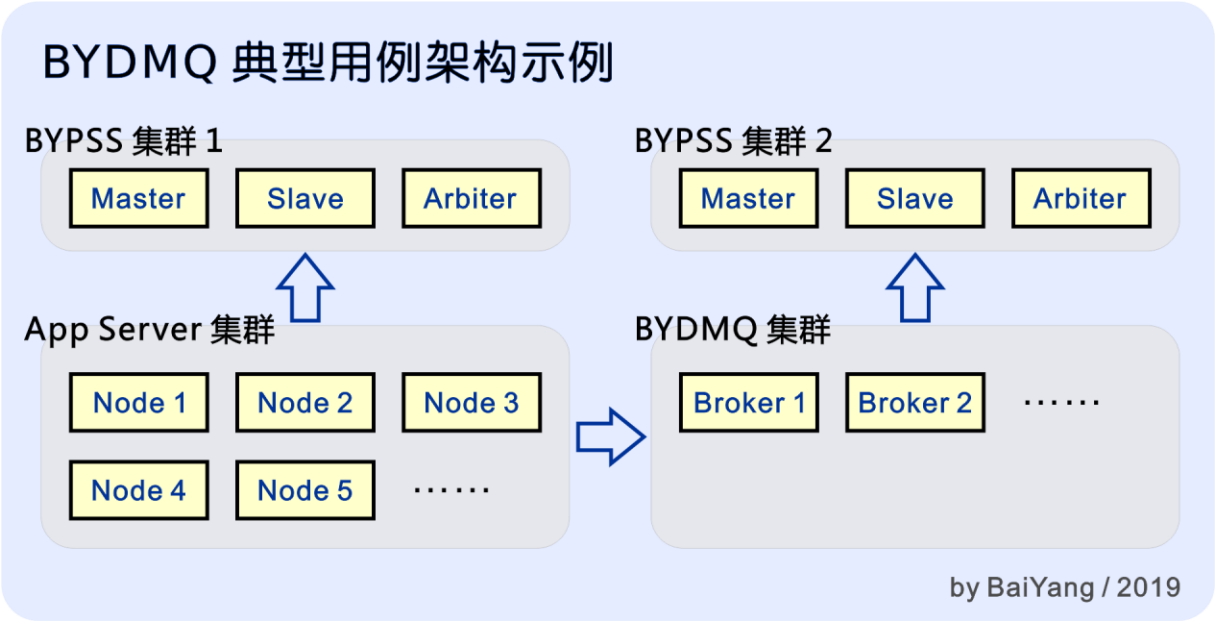

图 6

如图 6所示，在典型用例中，BYDMQ 与 App Server 集群各自拥有一套独立的 BYPSS 集群，它们分别负责各自的分布式协调任务，App 集群依赖 BYPSS1 完成分布式协调，而其消息通信则依赖 BYDMQ 集群来完成。

不过在研发、测试环境，或业务量不大的生产环境中，也可以让 AppServer 和 BYDMQ 集群共享同一套 BYPSS 服务。另外需要指出的是，此处描述的“独立集群”仅是指逻辑上的独立。而从物理上说，即使是两个逻辑上独立的 BYPSS 集群，也可以共享物理资源。例如：一个 Arbiter 节点完全可以被多个 BYPSS 集群所共享；甚至两个 BYPSS 集群中的 Master、Slave 节点还可以互为主备，这样即简化了运维管理负担，又能够有效节约服务器硬件和能源消耗等资源。

在继续介绍 BYDMQ 的主要特性前，我们首先要澄清一个概念，即：消息队列（消息中间件，MQ）的可靠性问题。众所周知，“可靠的消息传递”包含三个要素——消息在投递过程中，要能够做到不丢失、不乱序和不重复才能称之为可靠。令人遗憾的是，这世间目前并不真正存在同时满足以上三个条件的消息队列产品。或者换句话说，我们目前尚无法在可接受的成本范围内实现同时满足以上三要素的消息队列产品。

要说明这个问题，请考虑以下案例：

图 7

如上图所示，在这一案例中，消息生产者由节点 A、B、C 组成，而消息消费者包含了节点 X、Y、Z，生产者与消费者之间通过一个消息队列连接。现在消息生产者已经生产了 5 条消息，并将它们依序成功提交到了消息队列。

在这样的情形下，我们来逐一讨论消息投递的可靠性问题：

* **消息不丢失**：这点是三要素里最容易保证的。可拆分为两步来讨论：

  - 存储可靠性：可以通过将每条消息同步复制到消息队列服务中的其它节点（Broker）并确保落盘来保证；同时需要使用 Paxos、Raft 等分布式共识协议来确保多个副本间的一致性。但是显而易见，与无副本的纯内存方案相比，由于增加了磁盘 IO、网络复制以及共识投票等步骤，此方案会极大（数千甚至上万倍）地降低消息队列服务的性能。

  - ACK 机制：在生产者向消息队列提交消息，以及消息队列向消费者投递消息时，均增加 ACK 机制来确认消息投递成功。发送方在指定时间内未收到 ACK 机制则重发（重新投递）消息。

    以上两步措施虽可在很大程度保证消息不丢失（至少一次送达），但是可以看到其开销十分巨大，对性能的劣化非常显著。

    与此同时，也应该注意到多副本间的故障检测和主从切换、以及消息收发时的超时重传等技术手段均会引入各自的延迟。而且其中每个步骤中引入的延迟通常都超过数秒钟。

    在真正的用户场景中，这些额外的延迟使得“消息不丢失”的保证除了平白增加了巨大开销以外，大多没有任何实际用处：如今的用户在发起一个请求（如：打开一个链接、提交一个表单等）后，很少有耐心等待许久——若等待数秒后仍然没有响应，他们早就关闭页面离开或 F5 重新刷新了。

此时无论用户是关闭了页面还是重新发起了请求，已经延迟了几秒（甚至更久）才到达的那条消息（老请求）都已经没有了价值。不但如此，处理这些请求更是在白白消耗网络、运算和存储资源而已——因为其处理结果已经无人问津。

* **消息不重复**：考虑前文提到的 ACK 机制：消费者在处理完一条消息后，需要向消息队列服务回复一条对应的 ACK 信令来确认该消息已被消费。例如：假设上图中的 1 号消息是一个转账请求，消息队列将该消息投递给节点 X 后，节点 X 必须在处理完该转账请求后，向消息队列服务发生一条形如“ACK：Msg=1”的信令来告知消息队列服务，该消息已被处理。而 MQ 无法确保消息不重复的矛盾即在于此：

  仍然按照上例中的假设，MQ 将消息 1 投递到了节点 X，但在规定的时间内却并未收到来自节点 X 的确认（ACK）信令，此时有很多种可能，例如：

  - 消息 1 未被处理：由于网络故障，节点 X 并未收到该消息。
  - 消息 1 未被处理：节点 X 收到了消息，但由于故障掉电而未能及时保存和处理该消息。
  - 消息 1 已被处理：节点 X 收到了并处理了该消息，但由于故障掉电而并能及时向 MQ 服务返回对应的 ACK 信令。
  - 消息 1 已被处理：节点 X 收到了并处理了该消息，但由于网络故障，对应的 ACK 信令未能成功返回至 MQ 服务。

    等等。由此可见，在消息投递超时后，MQ 服务是无法得知该消息是否已被消费的。雪上加霜的是，由于前文所述的原因（不能让用户等太久），这个超时通常还要设置的尽量短，这就让 MQ 服务正确感知实际情况变得更加不可能。

    此时为了保证消息不丢失，MQ 通常会假设消息未被处理，而重新分发该消息（例如在超时后，将消息 1 重新分发给节点 Y）。而这就势必无法再保证消息不重复了，反之亦然。

* **消息不乱序**：从上例中可以看出，所谓“消息不乱序”是指 MQ 中的消息要按照先来后到，以“1、2、3、4、5”的顺序逐一被消费。要保证严格的不乱序，就要求 MQ 必须等待一条消息处理结束（收到 ACK）后，才能继续分发队列中的下一条消息，这至少带来了以下问题：

  - 首先，MQ 中的多条消息无法被并行地消费。例如：MQ 无法将消息 1、2、3 同时分别派发给节点 X、Y、Z，这使得大量消费者节点长期处于饥饿（空闲）状态，甚至于即使在正在执行消息处理的节点上（比如节点 X）也会有大量处理器核心、SMT 单元等计算资源被浪费。

- 其次，在处理一条消息的过程中，所有其后续消息均只能处于等待状态。若一条消息投递失败（超时），则在其“超时-重新投递”期间内，则会长时间阻塞其所有后续消息，使得它们无法被及时处理。

  由此可见，保证消息严格有序会极大地影响系统整体的消息处理性能、增加硬件采购和运维成本，同时也会显著破坏用户体验。

由以上论述可知，现阶段尚无在合理成本下提供消息可靠传递的 MQ 产品问世。在此前提下，目前的解决方案主要是依赖 App Server 自身的业务逻辑（如：等幂操作、持久化状态机等）算法来克服这些问题。

反过来说：无论使用号称多“可靠”的 MQ 产品，现在的 App 业务逻辑中也均需要处理和克服上述种种消息投递不可靠的情形。既然 MQ 本质上做不到消息可靠，同时 App 也已经克服了这些不可靠性，那又何必再花费性能被劣化几千、甚至几万倍的代价来在 MQ 层实现支持“分布式存储 +ACK 机制”的方案呢？

基于上述思想，BYDMQ 并不像 RabbitMQ、RocketMQ 等产品那样，提供所谓（实际无法达到）的“可靠性”保证。相反，BYDMQ 采用“尽力送达”的模式，仅在确保不损失性能的前提下，尽可能地保证消息被可靠送达。

正如前文所述，由于 App 已经克服了消息传递过程中偶尔出现的不可靠。因此这样的设计抉择在极大提升了系统性能之余，并未实际增加业务逻辑的开发工作量。

基于上述设计理念的 BYDMQ 包含了以下特性：

* 与 BYPSS 一样基于白杨应用支撑平台中的各优质跨平台组件实现，如：单点支持千万量级并发的网络服务器组件；支持多核线性扩展的并发散列表容器等等。这些优质高性能组件使得 BYDMQ 在可移植性、可扩展性、高容量、高并发处理能力等方便均拥有非常好的表现。

* 与 BYPSS 一样，在客户端和服务器端均拥有成熟的消息批量打包机制。支持连续消息的自动批量打包，大大提高网络利用率和消息吞吐量。

* 与 BYPSS 一样支持 pipelining 机制：使得客户端无需等待一条指令的响应结果即可连续发送下一条指令，显著降低了命令处理延迟、提高了网络吞吐、有效增加了网络利用率。

* 每个客户端（App Server）可注册一个专属 MQ，并通过长连接 + 心跳的方式保活，对应的属主 Broker（BYDMQ 节点）也通过此长连接向客户端实时推送到达的消息（带有批量打包机制）。

* 客户端通过一致性散列算法来推测一个 MQ 的属主（Broker），Broker 在首次收到针对指定 MQ 的请求（如：注册、发消息等）时，将通过 BYPSS 服务来竞选成为该 MQ 的属

主。若竞选成功则继续处理，竞选失败则引导客户端重新连接到正确的属主节点。

与此同时，BYDMQ 会通过 BYPSS 服务实时感知集群变化（如：现有 Broker 下线、新的 Broker 上线等），并将这些变化实时推送到每个客户端节点。这就保证了除非 BYDMQ 集群正在发生剧烈变化（大量 Broker 节点上线或下线等），否则通过一致性散列算法推定属主的准确性是非常高的，从而基本无需再次重定向请求。

此外，即使一致性散列算法推定错误，该 MQ 的实际属主也会被客户端节点自动记忆到本地快查表中，确保下次向这个 MQ 发送消息时能够直接投递到正确的 Broker。

这种由客户端直接向对应 MQ 之属主（Broker）投递消息的方法避免了服务器集群中的复杂路由和消息的多次中转，将消息投递的网络路由降至最简，极大地提升了消息投递的效率，有效降低了网络负载。

通过 BYPSS 来为 MQ 选举属主节点的做法则为集群提供了“每个 MQ 在全局范围内唯一”的一致性保证。与此同时，BYPSS 服务也负责在各个 Broker 节点之间分发一些控制类信令（如：节点上下线通知等等），使 BYDMQ 集群可以被更好地统一协调和调度。

* 所有待分发的消息仅存储在对应 MQ 属主（Broker）节点的内存中，避免了写盘、复制、共识投票等大量无用开销。

* 发送方可以为每条消息分别设置其生命期（TTL）和发送失败时的最大重试次数。可根据消息的类型和价值精确控制其消耗的资源。对时效性短或不重要的请求，可及时使其失效，避免在各个环节继续空耗资源，反之亦然。

* 支持分散投递：当指定 MQ 所属客户端节点未上线，或其连接断开超过指定的时长后，此消息队列中的所有待投递消息将被随机发送到任意一个仍可正常工作的 MQ 中。

  分散投递方案预期客户端（App Server）也是一个基于 BYPSS 的 nano-SOA 架构集群。此时若一个 App Server 节点因为运维、硬件故障或网络分区等原因下线，则该节点下辖的所有对象都会被管理该集群的 BYPSS 释放。

  此时系统可随机将发往此节点的请求散布至其它仍正常工作的节点，可让这些目标节点通过 RegPort 重新获取该请求相关对象的所有权，并接替其已经下线的原属主节点继续完成处理。这就大大降低了在一个 App Server 节点异常下线的瞬间，与之相关请求的失败率，优化了客户体验。同时随机散布也使得下线节点麾下的对象被集群中仍然工作的其它节点均分，保证了集群的负载平衡。

综上，BYDMQ 通过在一定程度上牺牲了本就无法真正保证的消息可靠性，再配合消息打包、pipelining、属主直接投递等方式，极大地提升了消息队列服务的单点性能。同时得益于 BYPSS 为其引入的强一致、高可用、高性能的分布式集群计算能力，使其拥有了优异的线性横向扩展能力。加之其对每条消息的灵活控制、以及分散投递等特性，最终为用户提供了一款超高性能的高

品质分布式消息队列产品。

3.3 游戏逻辑优化

为保证公正性和一致性，包括 NPC 行为、剧情触发、边缘检测、攻击判定等大量涉及到游戏中物体间的互动逻辑都需要交由服务器执行。正如前文所述，在完成了异步 IO 和零内存拷贝等高并发方面的优化后，服务器的瓶颈将会从 IO 转移到上述运算（处理器和内存开销）资源上。

在具体实践中，如边缘检测等物理引擎类逻辑大多已由汇编、C/C++ 等静态语言高效实现。但行为和剧情类逻辑由于经常是交由策划、文案等非专业开发人员来编制，为降低使用门槛，通常就会由 Python、Lua 等脚本语言来描述。

Python、Lua、JavaScript、Java、C#、PHP 等动态语言一方面放宽了编辑人员的使用门槛，另一方面却极大地降低了计算机的运算效率，这主要体现在以下几方面：

* 动态类型、动态代码生成、反射（Reflection）、GC、JIT/字节码解析，以及引用计数、访问检查、异常保护、虚拟机、沙箱等额外运行时检查和保护机制均会大大影响程序计算性能，同时大大提升程序内存开销。

  因此，如论在时间意义还是空间意义上，与经过汇编热点优化的 C/C++ 静态代码相比，这些动态程序都有相当大的额外开销。与充分优化的汇编/C/C++ 实现相比，其时空性能劣化可达几十，甚至数百倍。

* 每次与底层原生代码、系统和硬件交互时，都需要执行内存分配、内存拷贝、编码转换、上下文切换等操作，比起汇编/C/C++ 代码中直接函数调用来说，性能差异达到几万至几十万倍。即使 Lua、FakeScript 等针对此问题专门进行了优化的嵌入式脚本引擎，其调用开销也比原生代码内的直接函数调用要高成千上万倍以上。

  不幸的是，包括游戏逻辑在内的大部分嵌入式脚本都需要以很高的频率与原生代码大量交互（被原生代码回调或调用原生 API）。实际上，多数嵌入式脚本在实际运行中的最大开销就是自此处产生。

* 一台服务器上要同时支撑数以万计的玩家、NPC 和道具等有状态物体，每个物体都必须拥有自己的状态机和处理逻辑。所有这些物体均需要依照当前状态、自身逻辑、玩家输入等因素实时改变状态并触发对应行为，因此需要大量计算。

由此可见，在真实生产环境中依赖脚本语言来发布业务逻辑，会极大地影响系统的负载能力。但另一方面，业务人员通常又不具备直接使用汇编、C/C++ 编写高度优化代码的专业能力。

为此，我们拟采用三阶段的方式来平衡业务逻辑编写难度及其时空效率：

1. 在新地图、副本、剧情等业务模块开发初期，仍然使用脚本语言对其进行描述。以降低编辑和策划人员使用门槛，提升其工作效率。此时可基于 AST 或 Bytecode VM 解释执行，以方便修改和调试。

2. 在新业务模块公测以及正式运营的初期阶段，使用 JIT、AOT 等技术将上述脚本编译为 Native Code 执行。由于可通过工具自动完成，因此就可以很低的成本在较大程度上提升脚本的时间效率。

   除此之外，还可通过一些手段来提升脚本与原生代码之间的交互性能（如：LuaJIT 的 FFI library 等）。但这些优化需要调整脚本代码，并要求专业开发人员事先为编辑人员封装一系列 API 调用工具代码（大多是一些接口声明和代理类定义脚本）。

   但应当看到，这样的优化措施对空间效率基本不会产生任何提升——相反地，JIT 等技术还会在很大程度上增加其内存开销，从而使空间效率进一步下滑。此外，上述优化手段对时间效率的提升也有其极限，其最理想的结果可比解释执行快数倍到数十倍，但仍比由汇编、C/C++ 手工优化的代码性能相差几倍至几十倍不等。

3. 成熟的游戏逻辑通常已较为稳定，它们的变化通常都可被规约在配置（而非代码）层面。对于这类已经成熟稳定的业务逻辑，可使用 C/C++ 代码重新描述，同时根据实际需要对其中的热点部分使用汇编优化。

   无论在时间效率上还是空间效率上，这都彻底保证了执行性能的最优化。代价则是增加了额外的开发工作。但对于一个活跃的网络游戏来说，这点代价还是值得的。更何况增加新的地图、副本和剧情主要更新的部分也大多在配置层面，代码层面的游戏逻辑在很大程度上可沿用已有部分。增加一些分支剧情或副本常常只需对逻辑代码做少量更新，甚至仅需要修改配置。这就使得其工作量常常没有想象中那样巨大。

对一个活跃度较高的、已经进入了稳定运营阶段的网游来说，其 90% 以上的业务逻辑代码应以上述第三种形态存在。正在开发和调试的第二种形态则应约束在 10% 以内。至于第一种形态则仅应出现在开发和内测环境中。

基于上述考量，AngleScript 是目前为止最适合完成此类任务的脚本语言，原因是：

1. AngleScript 同时支持 Bytecode VM、JIT、AOT 等执行优化技术。

2. AngleScript 引擎中包含了特别设计的，能够与 Native Code 紧密结合的接口机制，配合其精心设计的 GC 模型，能够使对象同时被脚本环境和宿主环境共享，这就免除绝大部分内存拷贝和编码转换等上下文切换操作，因此极大第提升了效率。

3. 可通过关闭个别仅用于极端条件的保护机制，来进一步提升其脚本执行性能。由于服务器端脚本本身即需要由可信任的人员来编写，因此这不会产生什么问题——对于他们来说类似的保护功能并无意义，编写一个简单的死循环也足以拖垮整台服务器。

4. AngleScript 虽与其它脚本语言同样易于使用（通过沙箱环境屏蔽了底层硬件和操作系统细节；通过 GC 自动管理资源等等），但其语法兼容 C++。因此将其移植为 C++ Native Code 十分简单。

当然，在游戏活跃度尚未到达一定规模前，出于运维综合成本和当时的人力安排（开发组可能还在忙于完善游戏引擎之类更基础的部分）等方面的考量，也不必过于急迫地执行第三步优化。可以在人气（负载）逐渐提升，并且开发团队其它方面工作也逐渐完成（能够腾出人手）后，再从最热点的部分开始，逐步实施上述优化。

3.4 数据访问优化

数据访问的性能优化主要可以从缓存和存储两个层面来分析：

3.4.1 数据存储优化

数据存储层面的优化主要通过横向扩展的形式来解决。对于结构化的数据存储（DBMS）来说，与 MySQL、ORACLE、DB2、MS SQL Server、PostgreSQL 等"传统" RMDBS（SQL）数据库产品相比，当今无论是以 Cassandra、Scylla、MongoDB、SequoiaDB、ArangoDB 为代表的 NoSQL 数据库产品，还是 MySQL Cluster（NDB）、TiDB、Clustrix、VoltDB、PostgreSQL-XL、NuoDB、CockroachDB 等众多 NewSQL 产品都提供了成熟、可靠且有效的数据库横向扩展（Scaling out）解决方案。数据库层面的分布式和横向扩展早已不再成为问题。

非结构化存储方面更是百花齐放，从 FastDFS、TFS、GridFS、HDFS 等对象存储；AWS S3、阿里云 OSS、七牛云、又拍云等对象存储服务；到 GlusterFS、Ceph、Lustre 等可 mount 且支持 Native File API 的分布式文件系统。均能够在保持近乎线性大规模横向扩展能力的同时，提供非常高的可用性、可靠性和性能保证。

3.4.2 缓存访问优化

以 Memcached 为代表的分布式缓存作为一个完全基于内存和<Key, Value>对的分布式数据对象缓冲服务，拥有令人难以置信的查询效率以及一个优雅的，无需服务器间通信的大型分布式架构。对于高负载网络应用来说，Memcached 常被用作一种重要的数据库访问加速服务。

但从另一方面看，以 memcached 为代表的分布式缓存系统，其本质上是一种以牺牲一致性为代价来提升平均访问效率的妥协方案——缓存服务为数据库中的部分记录增加了分布式副本。对于同一数据的多个分布式副本来说，除非使用 Paxos、Raft 等一致性算法，不然无法实现强一致性保证。

矛盾的是，memory cache 本身就是用来提升效率的，这使得为了它使用上述开销高昂的分布式强一致性算法变得非常不切实际：目前的分布式强一致性算法均要求每次访问请求（无论读写）都需要同时访问包括后台数据库主从节点在内的多数派副本——显然，这还不如干脆不使用缓存来的有效率。

另外，即使是 Paxos、Raft 之类的分布式一致性算法也只能在单个记录的级别上保证强一致。意即：即使应用了此类算法，也无法凭此提供事务级的强一致性保证。除此之外，分布式缓存也增加了程序设计的复杂度（需要在访问数据库的同时尝试命中或更新缓存），并且还增加了较差情形下的访问延迟（如：未命中时的 RTT 等待延迟，以及节点下线、网络通信故障时的延迟等）。

与此同时，可以看到：从二十年前开始，各主流数据库产品其实均早已实现了成熟、高命中率的多层（磁盘块、数据页、结果集等）缓存机制。分布式缓存的出现，有其不可避免的时代特性——主要是在 RMDBS 系统尚无法有效完成横向扩展时，用于访问加速及负载均衡的一种补充服务。换句话说，分布式缓存系统的出现，主要是为了弥补单机数据库系统内存资源和网络 IO 能力均十分有限的瓶颈。

在数据库产品已可较好支持大规模分布式部署的今天，一个额外的分布式缓存层已经没有太大意义——具备横向扩展能力的数据库产品已不存在上述瓶颈。这时，额外的缓存层不但降低了系统的整体一致性，也引入了更多的网络 IO 和数据序列化、反序列化等操作。

相反，每个服务器节点的本地缓存则没有上述问题：本地缓存既不需要对数据进行序列化和反序列化，也不会增加网络 IO 和访问延迟——直接在本机内存命中即可。因此，配合底层分布式数据库内置分布式缓存机制的服务器节点本地缓存才是现代网络服务器架构中的最优选择。

此外，从服务器集群的角度来看，各个服务器节点中的高速本地缓存最终仍然组成了一个分布式缓存环境。比起 memchached 等外部缓存集群来说，它不但能提供真正的强一致保证，并且效率更高。关于此话题的详细讨论，请参考：3.2.3.3 基于 BYPSS 的高性能集群。

以上理论具体到网游服务器来说则更是如此：前文所述的“服务器节点/地图/玩家（用户）”三级伺服架构保证了任一在线用户在任意给定时间只属于一个地图实例，而任一地图实例在任何给定时间上仅属于一个确定的属主节点。

这样的逐级从属关系确保了每个玩家和他/她所感兴趣的绝大部分对象都粘滞在同一个属主服务器节点内，这种天然的缓存分布式算法确保了集群中每个节点本地缓存的利用率和命中率均十分突出。换句话说，这就确保了极高的内存利用率（每个服务器节点内的缓存数据基本都会被用到，没有浪费内存来缓存不会被使用的对象）和 IO 性能（每次 IO 访问都有很高的概率命中服务器节点内的本地缓存），同时也可为必要的数据字段提供强一致性保证。

此外，对于血量等临时状态，可执行全缓存策略（无数据库写入）；而对于经验等对可靠性要求不高（在发生异常宕机时，可容忍丢失数秒或数分钟内的更新），且更新较为频繁的字段，可使用 Buffer 机制临时缓冲在服务器内存，然后周期性地（如：每分钟一次）批量更新到后端数据库中，以提高写入效率。

4. 系统总体架构

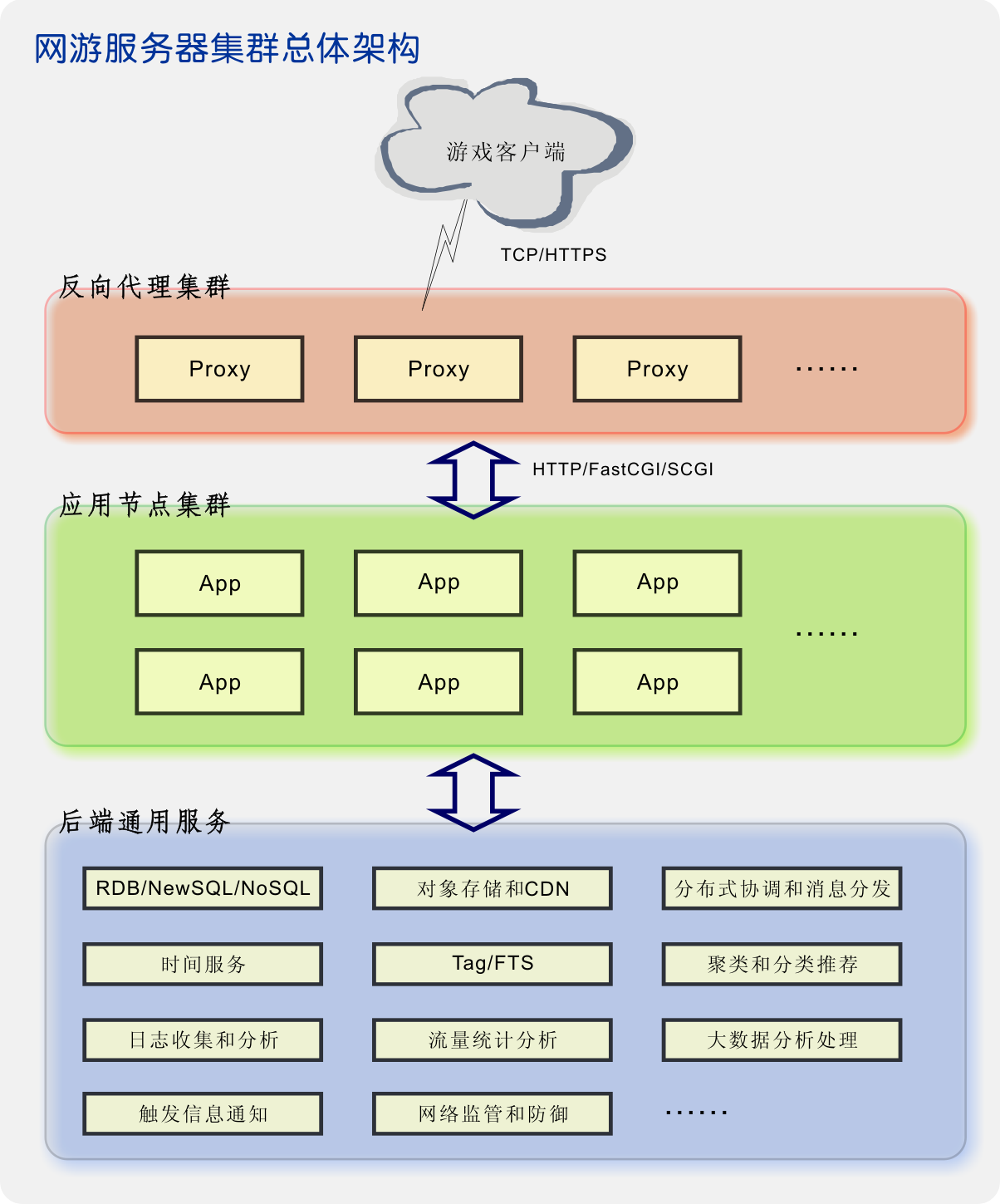

图 8

如图 8所示，分布式网游服务器集群系统主要分为后端服务、App 集群以及游戏客户端等部分。以下依循自底向上的顺序依次进行讨论。

4.1 后端通用服务

为应用提供支持的后端服务通用性较强，也多存在成熟的现有产品。因此这部分主要基于现有产品和服务来构建。

4.1.1 结构化存储（RDB、NewSQL、NoSQL）服务

数据库作为结构化数据的存储中心，存储所有与业务相关的数据，为所有数据访问提供 ACID 保证，并提供事务（Transaction）、索引和查询优化和等高级特性。系统通过《白杨应用支撑平台》中提供的 DBC、查询对象以及查询分析引擎等组件实现了跨数据模型（SQL / NoSQL / NewSQL）透明支持不同种类底层 DBMS 的目标，允许服务随着业务的发展从单点、主备到分布式集群逐步完成横向扩展的需求。

分布式网游服务器集群系统初期可在开发和生产环境均使用零部署、零维护的嵌入式引擎来实现快速、廉价的实施和部署，最大程度地降低实施和运维成本。随着业务的逐步发展，生产环境可逐渐迁移到 MySQL 主从集群（RDS 服务）、分布式 NewSQL 集群或分布式 NoSQL 集群等可用性更高，横向扩展能力更强的 DBMS 平台上。

此外，还可基于由《白杨应用支撑平台》提供的，支持实时（on-the-fly）数据压缩和强加密的虚拟文件系统（VFS）来为数据提供整库强加密保护。

4.1.2 对象存储和 CDN 服务

严格来说，对象存储和 CDN 是两个相互独立的服务。但由于这两个服务在业务上通常存在较紧密的集成，因此将他们放在一起讨论。

对象存储（Object Storage）提供非结构化大数据对象的 KV 访问服务。底层通常使用分布式文件系统来实现，出于对访问性能上的考量，通常需要对小文件进行特别优化（比如：将所有小于 64KB 的文件统一存放在一个内置的 DBMS 内）。对象存储通常使用集群，基于 EC 码或多副本机制来提供服务高可用和数据高可靠保证。

CDN 作为一种就近内容访问加速服务，原理非常简单，实现也不复杂，只要解决好回源和同步问题即可。但部署一套高效可靠的 CDN 网络需要投入的硬件建设成本非常高（个运营商的 BGP 线路、机房等）。

由于第三方可提供的对象存储和 CDN 服务目前已非常成熟和廉价，因此分布式网游服务器集群系统在初期和中期将直接通过租用七牛、又拍、腾讯、阿里云以及 AWS 等供应商提供的相关服务来实现该功能。

4.1.3 分布式协调和消息分发服务

分布式协调服务负责实现服务发现、服务选举、故障检测以及分布式锁等职能。消息分发服务则实现了消息路由和分发算法。这些服务对于任何分布式系统而言，都属于核心基础组件，算法相对复杂，实现难度高。

分布式网游服务器集群系统可使用《白杨应用支撑平台》中提供的现有 BYPSS、BYDMQ 等服务组件来实现上述功能，详见：3.2.3 白杨消息端口交换服务（BYPSS）以及 3.2.4 白杨分布式消息队列（BYDMQ）等小节。

4.1.4 时间服务

时间服务是经常被低估的重要网络服务，时间校准不仅是为了精确的计时和日志审核等业务目的，也是很多分布式算法能够正确运行的基础条件。小到分布式 ID 生成算法，大到分布式服务器的 ACID Transaction 一致性保证都需要集群中的各个服务器节点间能够实现精确的时钟同步。

NTP 协议在局域网环境中可实现亚毫秒（几十到几百微秒）级的时钟同步。配合 GPS 等权威时间源即可保证服务器集群中所有节点时间的精准和高度同步。IDC 运营商和 IaaS 云计算供应商大多提供 NTP 服务。若有必要，也可以极低的成本自行搭建 NTP 服务器集群。

4.1.5 标签（TAG）和全文搜索（FTS）服务

灵活的标签系统需要使用一种基于集合的包含关系高效匹配索引（MVA、MVF 内存索引），常用的 DBMS 通常不提供这类索引的支持。相反，此功能多由使用反向索引提供全文搜索服务的各产品来实现。加之标签功能与全文搜索服务在使用场景上通常也有较紧密的联系，因此我们将这两项功能合并在一起讨论。

现代全文搜索服务的构建大体上可拆分为分词（Tokenization）、构建全量倒排索引（Full Inverted Index）以及相关性评分等几个部分。对于拉丁语系来说，分词阶段的主要工作是去词根和匹配近义词等。对于汉藏语系（中文）等象形文字来说，问题要复杂的多——单词之间缺乏明显的边界，也不能简单地使用词典来进行划分，需要联系语义和上下文进行理解。例如：“妈妈亲了我叔叔也亲了我”、“北京大学生前来应聘”、“梁启超生前住在这里”、“我亲妹妹被打了”等等。

此外，还要解决好中英文混排、繁简混排、近义词匹配、词性分析等问题。目前比较好的中文分词算法包括 mm segment、mp segment 等。比较常用的分词工具包含 friso、结巴分词以及 scws 等。

在倒排索引和相关性评分部分，已有 Sphinx、Xapian、Apache Lucene 以及 Lucy 等较成熟的开源解决方案。当然，在使用这些解决方案的同时，还要考虑好如何将其搭建为高可用和高可靠集群的问题。

分布式网游服务器集群系统初期直接使用由阿里云、AWS 等第三方供应商提供的开放搜索服务。后期可根据实际情况选择使用 Sphinx + friso 等方案自建 TAG 和 FTS 平台。

4.1.6 聚类和分类推荐服务

聚类和分类算法是现代广告和推荐服务的基础，也是在线大数据实时处理领域中的重要分支。包括 Google、百度在内的广告业巨头均依赖这些算法生存和盈利。

聚类和分类推荐算法在游戏中尚未有大规模应用的场合，其有限的推荐类需求在初期完全可以历史喜好（Tags）匹配+相关性+广告权重的方式，由标签和全文搜索服务来覆盖。在实际运营一段时间后，若真的凸显了其对高精度、大规模推荐服务的切实需求，则可考虑构建专用的高性能计算集群来完成相应工作。

4.1.7 日志收集和分析服务

日志收集和分析服务以分布式、高可用的方式为系统中各节点产生的日志消息提供集中存储、分拣和分析服务。日志服务虽不是系统中的必须组件，但有助于提升系统运维水平，降低运维人员的工作负担。

由于是可选服务，因此在系统上线初期并不急于选型和绑定。随着业务发展和集群规模扩大，可考虑使用阿里云 SLS 等现有产品和服务。

另一方面，应注意到日志服务功能上其实是电信计费系统去掉了批价（计费引擎）、复杂事件处理（CEP）、预付费支持、中继路由支持、鉴权和授权支持等功能模块后的一个子集（电信计费服务本身可以看做是对电信交换机产出日志的处理系统）。其对可靠性的要求也远低于电信计费系统。

因此我们也正在利用业余时间，从蓝鲸计费系统中裁剪出一个分布式、高性能、高可用的日志收集和分析服务。若必要，届时亦可将分布式网游服务器集群的日志收集和分析功能迁移到该产品上。

4.1.8 流量统计分析服务

网站流量统计分析服务用于统计 PV、UV，客户浏览器/移动平台分布，客户来源、页面跳出

率、客户地理位置分布、客户屏幕尺寸统计等信息。对于非页游类网游来说，这方面需求相对较简单。免费开放的 Google Analytics、百度统计等服务足以满足分布式网游服务器集群系统初/中期运维需求。因此短期内无需付费购买或自行实现。

4.1.9 大数据分析和处理服务

随着分布式网游服务器集群业务的持续运营，业务数据不断积累。当业务数据到达一定量级后，即可启用分布式计算的模式定期或者实时地进行数据分析，以更好地辅助商业决策，提高付费率和玩家满意度，实现商务智能（BI）。

由于缺乏业务数据的积累，因此分布式网游服务器集群系统在初中期暂时无需大数据分析和处理服务。在数据积累量达到预期值后，应根据当时情况，优先考虑使用后端 NewSQL/NoSQL DBMS 服务所提供的 MapReduce 等分布式大数据在线分析能力。待其处理能力或功能上无法满足需求后，再考虑使用 Storm、Hadoop、HPCC、Gearman、Mesos 等方案来搭建专门的大数据处理平台，或租用由阿里云、AWS 等第三方供应商提供的大数据处理服务。若有必要，也可基于 BYPSS 服务自行构建分布式任务调度系统。

4.1.10 触发信息通知服务

触发类信息通知主要包括触发邮件和触发短信。触发类邮件服务的主要难点在于，想要达成较高的邮件投递成功率，首先需以会员身份加入国内和国际上，各 ESP 组织的白名单。其次要实现 SPF、DKIM、DMARC 等技术规范，并加入相应的管理组织。同时要随时获取来自上述个组织和 ISP 的反馈信息，并将信誉值始终维持在一个较高的水平。另外技术上，也需要在快速投递，出错重传等部分实现一些优化措施。

触发短信投递在传统上本是一个简单的服务，本来无非是选择一个投递成功率和价格均令人满意的服务供应商即可。但今年以来，随着我国在相关方面监管力度的大大加强，很多此类服务的供应商均已销声匿迹。中国电信（天翼开放平台）、中国移动等 ISP 也均已停用了对外发送任意触发短信的相关接口。目前，各供应商只提供基于模板的触发短信发送接口，并需要先完成企业身份验证和模板前置审批的程序。至于群发短信接口，更是已在国内被彻底禁用。

目前国内可较好解决上述问题，保持较高送达率、成功率和实时性的触发类信息投递服务供应商较少，主要有 SendCloud、Submail 等。由于自行搭建此类服务需要投入的关系政策、硬件资源等非研发类成本较高（此类服务的研发成本反而较低），因此分布式网游服务器集群系统在初中期只能选择依赖第三方供应商提供的相关服务，并接受相关限制（如：每天发送条数、信息模板预审核等）。待业务规模上升到一定量级，也可择机选择自建平台。

4.1.11 网络监管和防御服务

排除一些附加增值服务外，网络监管、入侵检测和防御几乎等于阿里云、AWS 等 IaaS 云服务供应商的全部核心功能了。更底层的系统（Linux）以及虚拟化方案（Xen、KVM、LXC 等）由于已有成熟的开源解决方案，因此反而不需要云服务供应商来自行研发。

也就是说，构建好了这些服务，也就基本上搭建好了一个云计算平台，可见其规模之庞大，工作量之繁重。所幸的是，阿里云、AWS 等云计算供应商以较完善地构建好了相关平台，并以基本免费的服务形式开放给用户使用。同时，他们亦均开放了各自的平台的二次开发接口（WebAPI），以方便用户采用可编程的方式自行扩展其功能，实现更高的自动化运维水平。

在监管方面，分布式网游服务器集群系统将依赖云计算供应商提供的网络监管平台，初期直接使用供应商提供的 B/S 架构 UI 界面来进行管理，其后可根据业务发展需要，通过调用相应的 API 自行开发自动化程度更高的监控和管理工具。用来补充通过 UI 界面手工操作的不足，提升运维工作的自动化和智能化水平。

在攻击检测和防御方面，主要依靠底层《白杨应用支撑平台》中提供的相关功能组件来提高自身的健壮性，做好恶意访问过滤、应用层和连接层高并发洪水攻击防御、慢速连接攻击检测和防御、SQL 注入避免等防御工作。与此同时，也应充分利用供应商提供的 DDoS 防御、安全体检、端口扫描等相关免费服务，以及我方自行部署的反向代理等服务来进行辅助的安全加固。此外，在遭遇大规模 DDoS 攻击时，也可专门购买供应商提供的 DDoS 高防服务。

4.1.12 CMDB 服务

配置信息分发服务主要用于自动化分发和更新配置信息，以及自动化地更新应用程序组件等用途。分布式网游服务器集群系统可使用《白杨应用支撑平台》中提供的现有 CMDB 服务来实现上述功能。

4.2 App 服务器集群

App 服务节点是系统的核心部分，依托底层提供各通用服务，实现了分布式网游服务器集群系统中的所有业务逻辑。由于使用了先进的 nano-SOA 架构，在同时保留了 SOA 架构高内聚、低耦合优点的前提下，App 集群中的所有节点得以以完全对等的形式来实现。

得益于 nano-SOA 架构，我们：

* 只要维护一套 App 节点集群即可实现负载均衡和高可用，不必为应用中的每个服务部署和维护单独的高可用集群。运维工作量大大减轻，服务器、存储等资源的购买或租用成

本也大大降低。

* 避免了大量节点间通信，时间和空间效率均大大提升。可以使用更少的硬件资源和运维投入来服务更多用户。并实现更短的加载时间，更快速的即时请求响应，以提供更优质的客户体验。

* 正如前文所述，完全对等的集群模式不仅消除了大部分节点间通信的需求，而且避免了对高可用、高可靠的消息中间件之依赖。这从根本上解决了传统 SOA 模式中，由消息中间件引入的各种可靠性、可用性和性能瓶颈。

基于 nano-SOA 的设计思想，我们将一个 App 节点划分为基础平台和可插拔业务模块两大部分。其中基础平台提供底层功能封装、网络伺服框架、数据库连接、存储管理、服务管理、配置管理、插件管理、角色管理、用户管理、会话管理等各种基本服务。而玩家工会、玩家大厅、交易管理、物理引擎、游戏逻辑等具体业务逻辑则由不同的业务模块分别实现。

有关 App 服务节点设计的详细信息，请参考：5. 系统总体设计。

4.3 反向代理集群

位于三层构架中最外层的反向代理服务器负责接受用户的接入请求，在实际应用中，反向代理服务通常至少还要完成以下列表中的一部分任务：

连接管理：分别维护客户端和应用服务器的连接池，管理并关闭已超时的长连接。

攻击检测和安全隔离：由于反向代理服务无需完成任何动态页面生成任务，所有与业务逻辑相关的请求都转发至后端应用服务器处理。因此反向代理服务几乎不会被应用程序设计或后端数据漏洞所影响。反向代理的安全性和可靠性通常仅取决于产品本身。在应用服务的前端部署反向代理服务器可以有效地在后端应用和远程用户间建立起一套可靠的安全隔离和攻击检测机制。

如果需要的话，还可以通过在外网、反向代理、后端应用和数据库等边界位置添加额外的硬件防火墙等网络隔离设备来实现更高的安全性保证。

负载均衡和请求路由：将玩家的请求路由到服务器集群中的正确属主节点进行处理。对于不具备节点粘滞性的请求，使用最少连接数或 Round Robin 等算法进行负载均衡调度。

分布式的 cache 加速（CDN）：可以将反向代理分组部署在距离热点地区地理位置较近的网络边界上。通过在位于客户较近的位置提供缓冲服务来加速网络应用。

静态文件伺服：当收到静态文件请求时，直接返回该文件而无需将该请求提交至后端应用服务器。

动态响应缓存：对一段时间内不会发生改变的动态生成响应进行缓存，避免后端应用服务器频繁执行重复查询和计算。

数据压缩传输：为返回的数据启用 GZIP/ZLIB 压缩算法以节约带宽。

数据加密保护（Encryption Offloading）：为与客户端的通信启用基于 TLS 或私有协议的加密保护。

容错：跟踪后端应用服务器的健康状况，避免将请求调度到发生故障的服务器。

用户鉴权：完成用户登陆和会话建立等工作。

地址别名：对外建立统一的地址别名信息，遮掩资源的真实位置。

对页游类游戏，可使用 HAProxy 等业界成熟的现有产品作为反向代理服务。对端游和手游，则应自行实现更为高效智能的反向代理服务，以提供更好的灵活性和更多功能支持。

4.4 客户端

网游客户端可以为端游、手游或页游，其具体设计和实现不在本方案讨论范围内。

5. 系统总体设计

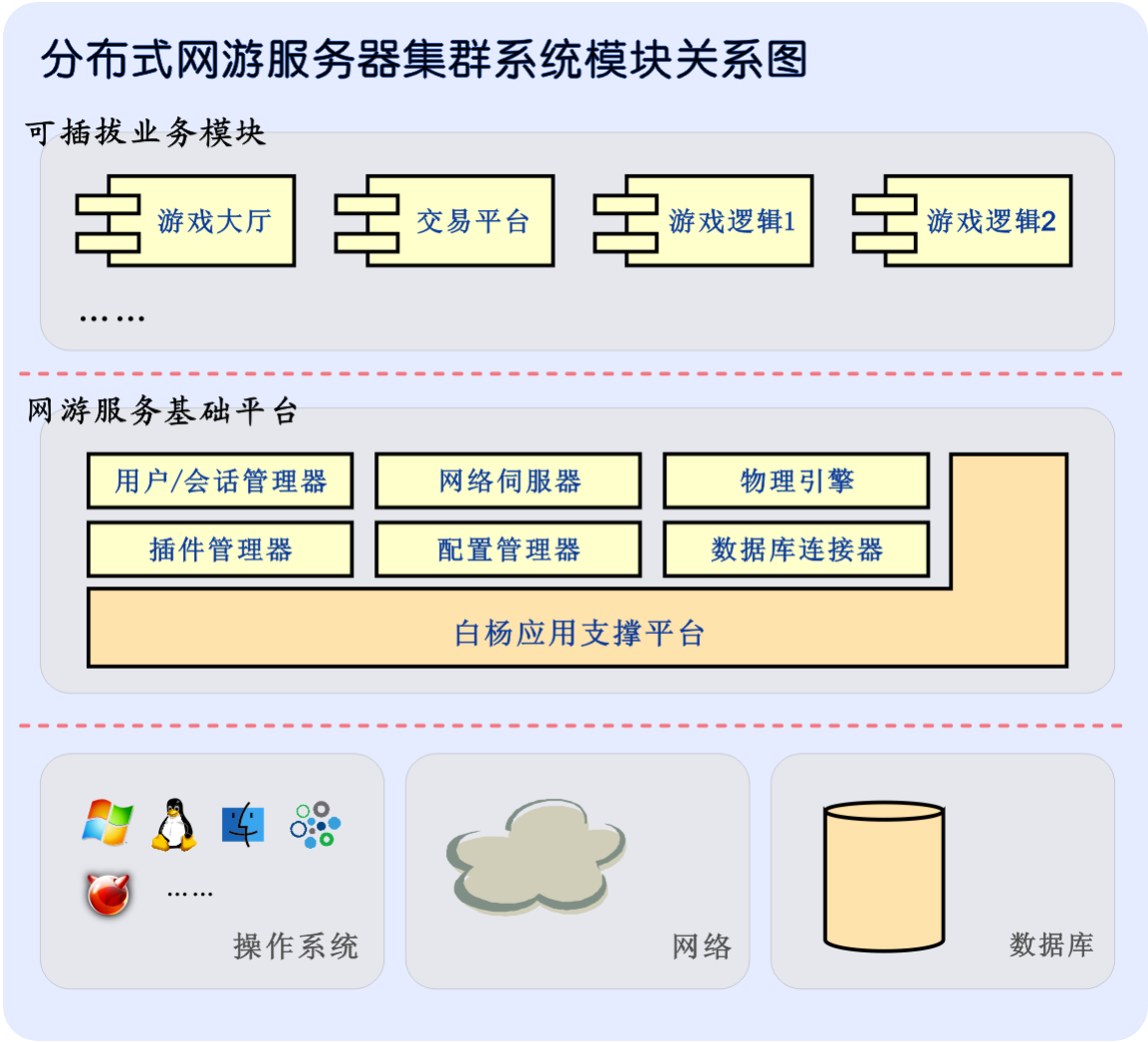

图 9

如图 9所示，分布式网游服务器集群系统主要可分为网游服务基础平台和可插拔业务模块两大层面，以下逐一讨论：

5.1 网游服务基础平台

基础平台向下封装所有与操作系统、网络、数据库访问，以及其它与后端通用服务相关的底层操作；向上则作为各个业务模块的运行支撑环境，并为这些业务模块提供用户管理、配置管理、Web 伺服等基础服务。

在架构层次上，分布式网游服务器集群基础平台又可细分为《白杨应用支撑平台》和分布式网游服务器集群基础平台两部分：

5.1.1 白杨应用支撑平台

应用支撑平台是整个分布式网游服务器集群系统的基石，是分布式网游服务器集群与底层操作系统及硬件平台间的重要接口，应用支撑平台在完整封装操作系统功能的同时，还提供各种通用工具和框架。应用支撑平台是高质量、低成本快速开发及支持跨平台应用的关键组件。

5.1.1.1 应用支撑平台功能描述

应用支撑平台为分布式网游服务器集群系统提供了众多基础功能，包括：

* 跨平台底层支持：封装所有与操作系统相关的操作，如：信号量、原子操作、共享内存／文件映射、进程、线程、网络操作（Socket）、文件管理、服务控制、注册表访问、进程间通讯、服务器框架、异步 IO 框架等等。是分布式网游服务器集群系统实现跨平台、多平台的关键组件。

* 通用功能：包括用户权限管理、基于 PKI 体系结构的强加密算法、常用网络协议、二进制和字符集编码转换、自动化脚本引擎、表单处理、数据压缩、任务管理、日志记录等等。

* 平台无关的高效 Web 扩展框架。

* 平台无关的复杂数据结构表示以及支持实时压缩和强加密的虚拟文件系统。

* nano-SOA 架构基础设置，包括：平台无关的功能插件框架；灵活，支持实时（on-the-fly）数据压缩和强加密保护的数据库连接器（DBC）插件；通用 API 注册和分发机制（API Nexus）；ETL 辅助工具；基于平台查询分析引擎的自定义高级查询辅助工具；以及 BYPSS 分布式协作与消息分发服务等等。

* 平台无关的国际化支持：为用户提供一个平台无关的多语言、国际化工作环境。

等等，共上千功能组件。应用支撑平台完整封装了几乎所有与底层系统相关的功能，以及很多常用的算法、框架和功能组件。

5.1.1.2 应用支撑平台设计框架

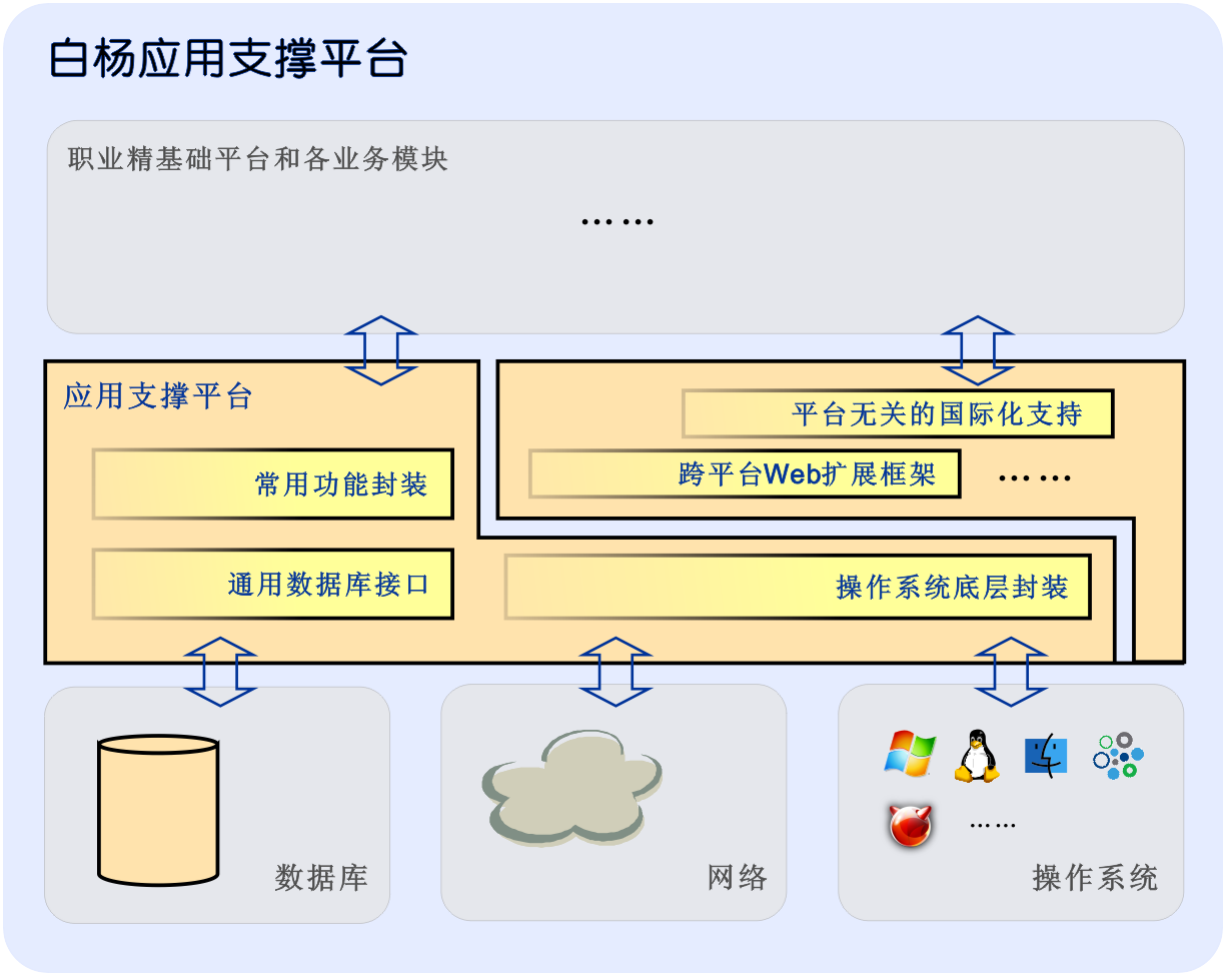

图 10

如图 10 所示，白杨应用支撑平台位于操作系统、网络、数据库系统等底层环境及其它分布式网游服务器集群基础服务和业务模块等高层应用之间，是分布式网游服务器集群的构造基础。向下封装各类底层的、操作系统和硬件平台相关的通用功能，向上为分布式网游服务器集群系统提供稳定强大、平台无关的通用服务和常用组件。

跨平台框架统一封装了进程；线程；线程本地存储；信号量；共享内存；命名管道；文件映射；原子操作；消息机制；网络访问；文件系统管理；系统时钟；任务计划；服务控制；动态库装载等各类操作系统功能。

同时提供了基于 PKI 体系架构的高强度数据加密；基于数字证书的 License 生成和验证；数据压缩算法；通用句柄操作；数据编码转换；字体访问；字符集转换；数据库访问；报表处理；图像生成；用户鉴权和访问控制机制；音视频输入输出及编解码；嵌入式自动化脚本引擎；nano-SOA 框架等各类通用功能。

支撑平台从一开始就是专为对负载和可靠性要求很高的企业及互联网行业而设计的。其中的

每个组件都会在保证可靠性的前提下，使用对当前底层平台来说，最高效的方式予以实现。

例如：支撑平台中的网络 IO 和 Web 扩展框架总是会尽可能地利用底层系统所支持的最优模式来完成网络访问：在 Windows 上使用 IOCP；在 Linux 上使用 epoll；在 FreeBSD/NetBSD/OpenBSD 等 BSD 环境中使用 kqueue；在 Solaris 上使用 event ports；在 HP-UX 上使用 /dev/poll、在 IBM AIX 上使用 pollset、而在其它 POSIX 环境下使用 POSIX AIO + Realtime Signal 等等。

再比如：根据实际情况直接使用内联汇编或编译器 Intrinsics 对原子量操作、内存屏障以及字节序交换等热点功能进行优化。并直接使用这些已被极度优化的组件构建快速互斥量、自旋锁、以及快速信号量等组件。

白杨应用支撑平台完全使用高效强大的 C++ 语言和少量汇编代码进行开发，由一组跨平台的功能组件构成，全面支持当今流行的各种软、硬件环境[(1)]。目前已被广泛应用于多个大型项目[(2)]，其高效性、可靠性和成熟度已被广泛验证。

使用统一的应用支撑平台和用户界面组件构建软件将为开发者带来很多显而易见的好处，例如：

* 更短的开发周期和更低的开发成本：通过重用已良好封装的组件，有效缩短开发周期，减少项目整体费用，显著提高开发速度。

* 更高的产品质量和更低的项目风险：大量重用经过岁月积累和时间考验的组件对于提高代码整体质量来说，显然具有很强的正面效果。普遍使用成熟稳定的组件也将有助于降低项目失败或延期的风险。

* 跨平台能力和国际化支持：统一应用支撑平台和统一用户界面组件库封装了几乎所有系统相关的操作，并提供了很强的跨平台支持能力，以它们为基础开发的应用因此具有平台无关特性。典型情况下，跨平台移植的全部工作仅仅是对应用的简单重编译。同时，国际化作为统一用户界面组件的一个重要附加功能，为应用提供了平台无关的国际化支持能力。

* 极高的运行时效率：100％ 高效 C++/汇编编码，无论是时间效率还是空间效率上都没有任何妥协。

白杨应用支撑平台包含繁多的功能和复杂的设计。其复杂度甚至还要远超分布式网游服务器集群系统本身。作为一个完善、独立的产品，应用支撑平台的具体细节已经超出了本文的讨论范围，关于和应用支撑平台相关的进一步信息，请参考：《应用支撑平台技术白皮书》（http://baiy.cn/doc/asp_whitepaper.pdf）。

注 1：我们当前支持的操作系统主要包括：

- Win98/ME, NT4/2000/XP/2k3/Vista/2k8/Win7/2k8r2/Win8/8.1/2012/2012r2 等全系列 Windows 产品。
- Linux、FreeBSD、NetBSD、IBM AIX、HP-UX、Solaris、MAC OS X 系统及众多 Un*x/POSIX 系统。

- vxWorks, Nuclear, QNX, SMX, DOS, Windows CE (Windows Mobile), NanoGUI、eCos、RTEMS、Android、iOS 等嵌入式系统。

当前支持的主要硬件平台包括：x86/x64、ARM、IA64、MIPS、POWER、SPARC 等。

注 2：应用支撑平台的典型客户中，包括了兴业银行（China CIB）、中石油（CNPC）、华安保险（Sinosafe Insurance）、淘宝网（taobao.com）、烟台万华集团、法国兴业银行（SOCIETE GENERALE）、德尔福汽车（Delphi）、美联航（United Airlines）、GE（美国通用电气）、贝塔斯曼（Bertelsmann）、埃森哲（Accenture）、中国光大银行（CEB）等各大企业。

5.1.2 基础服务群

分布式网游服务器集群基础平台中的服务群构建于《白杨应用支撑平台》之上。以应用支撑平台为依托，实现了诸如：用户管理、配置管理、插件管理、数据库连接管理以及前台/后台 Web 伺服器等各类基础服务。此外，基础服务群在与应用支撑平台一起为各个业务模块提供运行环境的同时，也负责对正在运行的各业务模块进行管理。

5.1.2.1 基础服务群功能描述

基础服务群实现以下功能：

* 鉴于配置数据与业务数据相分离的设计思想，需要专门提供可靠的配置信息存储服务。该服务通常由带有自动备份和自动版本回退机制、使用强加密和实时数据压缩的本地 VFS 来实现。但在需要实现横向扩展时，也可以基于远程分布式存储或独立的 DBMS 来实现。

* 为业务数据实现后端 DBMS 的访问和连接管理机制。并为其提供对用户透明的实时数据压缩和强加密保护、基于 Revision 的分布式乐观锁、高级自定义查询等功能。

* 与第三方服务或子键全文搜索服务对接，为业务数据实现基于倒排索引的高效 TAG 和 FTS 搜索支持。

* 与第三方对象存储和 CDN 服务对接，为系统提供附件上传，以及访问授权和鉴权等功能。

* 基于 BYPSS 服务实现节点间消息分发和集群协作功能。

* 与第三方平台对接，实现触发消息通知功能。

* 实现可插拔功能插件管理器。

* 在插件机制的基础上，实现业务模块管理器。完成对各业务模块的启动、停止、加载、卸载、启用、禁用等各种调度和管理工作。与此同时，由于不同业务模块间可能存在依赖关系，因此业务模块管理器内，还需要建立一套可靠的模块间通信机制。

* 实现分布式、可扩展的用户管理和会话管理机制。

* 实现服务器端物理引擎，提供边缘检测、伤害判定等高度优化的通用功能。

* 实现分布式的即时消息分发和推送机制。

* 实现待办事务伺服器，为用户提供待办事务计划和提醒。

* 实现计划任务伺服器，提供任务预约执行功能。

* 实现自动更新伺服器，依赖后端 CMDB 服务实现配置和应用程序自动更新功能。

* 对接后端日志收集和分析服务。

* 对接后端流量统计分析服务。

* 与后端大数据分析和处理服务对接，实现基于周期性和实时大数据在线分析的 BI 逻辑。

* 分别实现用于前台访问和后台管理界面的网络伺服器框架。网络通信协议根据具体需要使用 HTTP 或 TCP，框架应满足设计要求中定义的各项指标，支持高效的消息推送机制，并允许各业务模块灵活地进行功能扩展。

* 依照实际需要实现支持@功能的讨论区、地理位置管理器、时区管理器和时段管理器等通用组件。

5.1.2.2 基础服务群设计框架

考虑到设计要求中的高并发、低延迟、大规模组播和消息推送长连接等各项指标，前台网络伺服器框架需使用高效的异步 IO 架构来实现，其工作模型如下图所示：

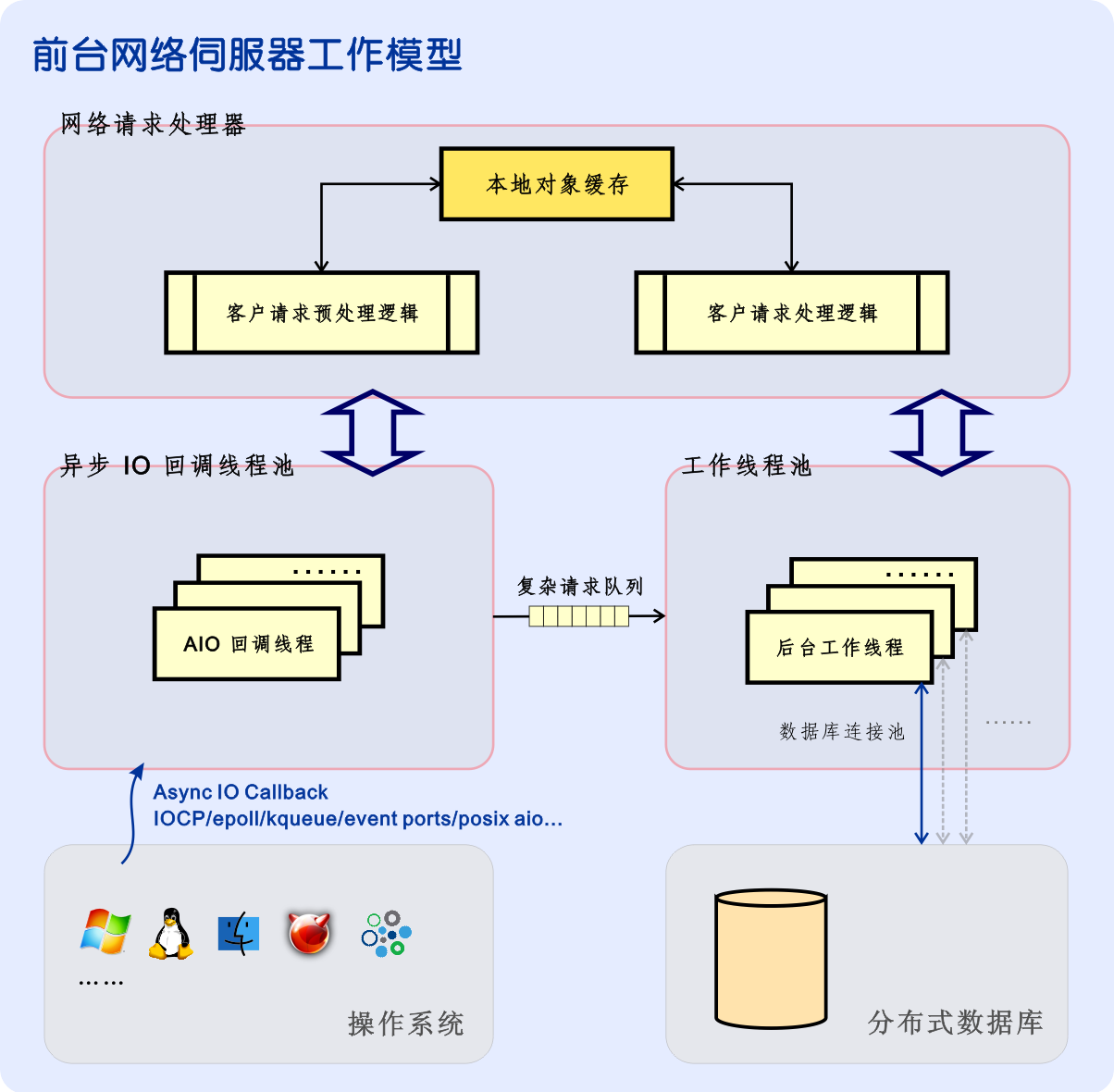

图 11

如图 11 所示，前台网络伺服器工作流程如下：

* 当一个客户端请求到达后，底层操作系统通过 IOCP、epoll、kqueue、event ports、real time signal (posix aio)、/dev/poll、pollset 等各类与具体平台紧密相关的 IO 完成（或 IO 就绪）回调机制通知异步框架的 AIO 回调线程，对这个已到达的请求进行处理。

* 在 AIO 回调池中的工作线程接收到一个已到达的请求后，首先尝试对该请求进行预处理。大部分针对临时状态等瞬时数据、以及配置类数据的访问请求均可在预处理过程中完成。多数能够命中本地缓存（cache）的业务类数据查询、以及对一致性要求不高，允许延迟写入（buffer）的业务类数据更新操作也可在此阶段处理。如果满足上述条件，预处理请求成功，则直接将结果（仍然以异步 IO 的方式）返回客户端，并结束本次请求。

* 如果指定的请求要求查询的数据无法被本地缓存命中；或者这个请求需要通过式地写入数据库；亦或此请求需要以同步语义访问 BYPSS 服务（通常为端口注册，绝大部分消息发送和端口注销操作都可以异步方式在预处理阶段完成）。则该请求将被 AIO 回调线程追加到特定的复杂请求队列中，等待工作线程池中的某个空闲线程对其进行进一步处理。

* 工作线程池中的每个线程都分别维护着一条与底层到数据库服务相连的长连接。另外，工作线程也可随时通过 BYPSS 连接池重用已有的 BYPSS 客户端连接。通过让每个工作线程维护属于自己的数据库长连接，以及共享 BYPSS 连接池等机制，后台工作线程池以最恰当的方式分别实现了数据库和 BYPSS 服务的长连接池机制。长连接（Keep-Alive）机制通过为不同的请求重复使用同一条网络连接大大提高了应用程序和网络 IO 的处理效率。

* 工作线程在复杂请求队列上等待新的请求到达。在从队列中取出一个新的请求后，工作线程将该请求与可用的数据库连接等资源一同提交给基础平台或指定业务模块的请求处理器进行处理。

* 请求处理器首先使用工作线程指定的数据库长连接等资源处理指定的客户端请求（如果存在匹配的 API 规则，则将其送交相应的业务模块进行处理）。然后将处理结果提交给底层网络通信框架，将结果通过异步 IO 的方式返回给客户端。

应注意到：前台网络伺服器中的客户请求处理器及客户请求预处理均允许业务模块进行注册。所有已正确注册的业务模块都可以通过 API 匹配等方式参与请求的处理和预处理机制。这就使得业务模块可以对客户端界面和 API 接口进行扩展，以此向用户暴露自己的外部接口。

需要进一步说明的是，与 epoll/kqueue/event ports 等相位触发的通知机制不同，对于 Windows IOCP 和 POSIX AIO Realtime Signal 这类边缘触发的 AIO 完成事件通知机制，为了避免操作系统底层 IO 完成队列（或实时信号队列）过长或溢出导致的内存缓冲区被长时间锁定在非分页内存池，在上述系统内的 AIO 回调方式实际上是由两个独立的线程池和一个 AIO 完成事件队列组成的：一个线程池专门负责不间断地等待系统 AIO 完成队列中到达的事件，并将其提交到一个内部的 AIO 完成队列中（该队列工作在用户模式，具有用户可控的弹性尺寸，并且不会锁定内存）；与此同时另一个线程池等待在这个内部 AIO 完成队列上，并且处理不断到达该队列的 AIO 完成事件。这样的设计降低了操作系统的工作负担，避免了在极端情况下可能出现的消息丢失和内存泄露，同时也可以帮助操作系统更好地使用和管理非分页内存池。

在这一模型中，应用程序可将需要传输的内存地址直接提交给底层硬件，硬件通过 DMA 直接在这个内存位置完成 IO 操作，这就实现了内存零拷贝。在完成 IO 操作后，硬件通过触发中断通知操作系统，并由操作系统回调应用程序。不但如此，由于可以向底层驱动并发地提交多个 IO 请求，这就使操作系统和底层硬件有机会实现操作合并（例如：将多个消息合并到一个网络帧或磁盘 IO 请求中完成读写）以及最优化请求顺序（例如：调度磁头从最近的磁道开始读写请求）等等。

得益于上述高效异步 IO 模式以及使用汇编代码的精细手工优化，《白杨应用支撑平台》中的高并发 Web 伺服组件可提供极高的 IO 吞吐和并发能力。即使在一台 2011 出厂的双路 Intel 至强 5600 入门级 1U PC Server 上（当时售价 RMB 2 万以内），亦可达到单点支持千万量级超高并发的水平。与 IIS+asp.net / Apache+php / Nginx+php 以及相应的 Java / Python / RoR 等方案相比，基于汇编/C/C++的 Web 应用框架具备几何级数的性能优势。

相反，由于不需要考虑高并发和高负载等使用场景，后台管理 Web 伺服器使用较简单的多线程，每连接一线程的阻塞 IO 模式：

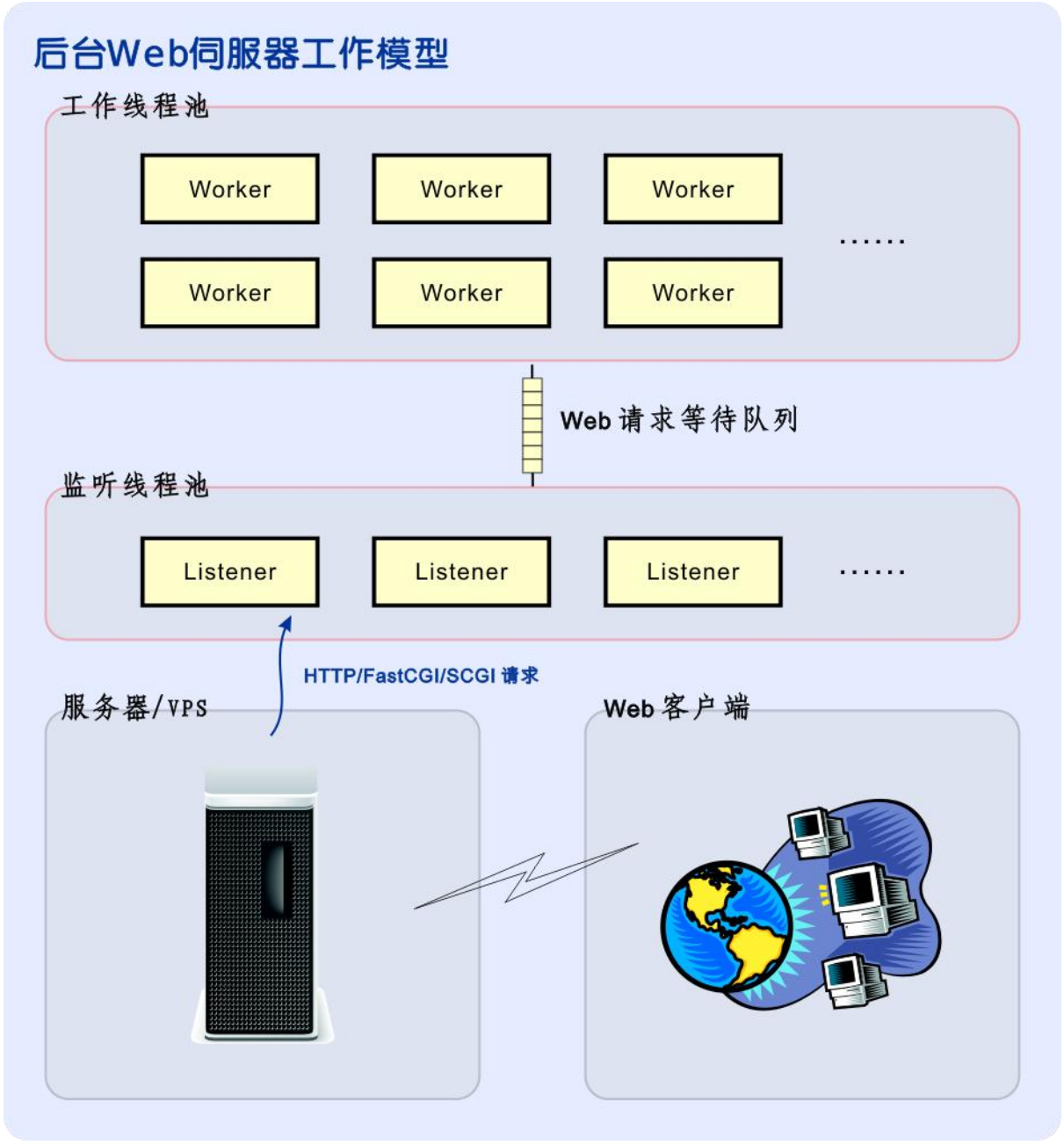

图 12

如图 12 所示，后台 Web 伺服器工作流程如下：

* 监听线程池等待到达用户和会话管理服务器的 Web 请求，并完成与该请求相关的初始化工作。然后将其压入 Web 请求队列的尾端，并继续监听下一个请求。为了增强高负载模式下的并发性，这些工作由单独的线程池来完成。监听线程池中的线程数量可以由用户配置，也可以根据负载情况动态调整。

* 工作线程池不断从等待队列的首部取出待处理的请求，将其提交给 Web 请求处理器。Web 请求处理器可视情况将其转交给相应的业务模块，最后返回处理结果。工作线程池可以根据当前负载状况和用户配置的参数进行动态调整。在处于轻载或空闲时，工作线程的数量保持在用户指定的最小值。当负载持续加重时，为提供额外的并发处理能力，线程数量会逐渐增加，直到达到用户指定的最大值为止。当业务高峰结束，负载量持续下降时，应用框架会逐渐回收已空闲的工作线程，直到达到用户指定的最小值为止。工作线程的回收策略可以由用户指定。

与 IO 密集型的网络伺服器不同，物理引擎是典型的计算密集型组件。简单说来，物理引擎就是以物理规律对游戏世界中各种物体间的相互影响进行仿真的组件。除简单的刚体以外，现代物理引擎还可对粒子（大规模物体）、流体和软体等物理对象进行复杂的运动、旋转、碰撞反应（如：变向、破碎、弹跳、形变等等）等物理学运算。除此之外，优秀的物理引擎还可对包含齿轮、皮带、铰链、关节等复杂机械装置的复合物体进行仿真。

毫无疑问，即使仅针对单个玩家，上述仿真能力无疑也需要巨大的运算量和专门的协处理（如：GPU）加速才能实现。从当前网游发展阶段来看服务器端由于要同时伺服大量玩家，其物理引擎部分必须经过高度优化和精简——仅包含涉及公平性和一致性的部分，如：针对简单刚体的边缘检测、碰撞判定、伤害判定等。而将主要用来增强游戏效果的复杂运算交由每个玩家各自客户端上的 GPU 等加速设备来分别完成。

不过仍应看到，随着玩家对更自由的沙箱仿真世界越来越向往，计算能力不断增强，计算成本不断下降，对于服务器端实现完整物理引擎的呼声一定会不断提高。因此，服务器端物理引擎的品质和表现很可能会成为未来网游产品的核心竞争力。

除上述组件以外，基础服务群中的其它组件均容易理解和实现，本文不再赘述。

5.2 可拔插业务模块

正如前文所述，分布式网游服务器集群系统遵循 nano-SOA 架构，将系统划分成了基础平台和可插拔业务模块两个部分。

从平台的角度讲，每个可插拔业务模块都是一个运行在网游服务基础平台上的插件。业务模块依托于《白杨应用支撑平台》和网游服务基础平台提供的运行时环境，消费基础平台提供的各项服务，并服从基础平台的管理和调度命令。在必要时，多个业务模块间也可通过应用支撑平台

提供的调用分发（API Nexus）机制来实现相互通信。

从用户的角度讲，诸如游戏大厅、交易平台、主线剧情、游戏副本等所有实际业务功能其实都是由各个业务模块实现的。一个完整的分布式网游服务器集群 App 节点就是由网游服务基础平台加上各个业务模块组合而成的。

从这个角度看，基础平台好似一套空房间，而业务模块则是各种可以摆放在房间中的家居物件。房间提供了水电煤、电信等基础设施，与其中部署的家具和设备一起为入驻人员提供针对特定用途优化的运行时支持环境。从这个角度出发，也可将《白杨应用支撑平台》看做构造房间和家具的材料，如：混凝土、预制板、管道、连接件、板材等。实际上，在这个类比中，支撑平台的品质足可比拟一款在多种极端环境下久经考验的，钛合金与特种陶瓷构成的复合材料。支撑平台为构建优质应用提供了非常坚实的基础。

5.2.1 可拔插业务模块功能描述

nano-SOA 架构的特性和优势在本文 3.2 nano-SOA 架构等小节中，已被充分论述。分布式网游服务器集群系统需要实现的业务模块群根据具体游戏产品不同而有所变化，以下列举几类常见的业务模块：

* **游戏大厅**：提供点卡兑换、客服支持和地图选择等功能入口。
* **交易平台**：实现玩家与玩家之间，以及运营商与玩家之间的道具等游戏资源交易功能。
* **游戏逻辑**：可由任意多个 BMOD 组成，分别描述主线剧情、支线剧情、副本剧情、以及 NPC 行为策略等游戏逻辑。由于涉及玩家、NPC 等大量对象的状态实时跟踪和计算，因此游戏逻辑很容易成为服务器端计算性能的瓶颈，有关其优化策略的说明，可参考：3.3 游戏逻辑优化。

5.2.2 可拔插业务模块设计框架

可拔插业务模块与数据库连接器插件、统计分析插件一样，遵循标准的插件接口，并通过专用的管理器进行调度和管理。从外部观察，业务模块只要遵循《白杨应用支撑平台》中的定义，实现相应的插件接口即可。从内部实现来看，每个业务模块都各不相同，本文不再一一赘述。