白杨应用支撑平台

技术白皮书

版本：2.45

http://baiy.cn

前言

白杨应用支撑平台从 2002 年发布第一版以来已有 16 年历史了。在这十几年里，我一共为支撑平台编写了超过一百万行汇编、C/C++代码。其中最基础也是最重要的 libutilitis 库几乎完全由我亲手实现。因为要实现 libutilitis 中包含的大量跨平台(*)功能封装，除了原本就比较了解的 Windows 平台，我也逐渐熟悉并喜爱上了 Linux、FreeBSD/NetBSD/OpenBSD、OpenSolaris 等各种优秀的操作系统。

对于支撑平台中的其它库，我的主要做法就是以 libutilitis 为基础，设计一套界面/框架。然后依赖没有版权问题而且品质稳定的第三方代码来实现具体功能。例如：libcrypto 中的密码编码学和数据压缩算法；libaudioio 中的音频编解码器；libmlgui 中的跨平台 UI 控件等等。在计入第三方开源代码后，应用支撑平台共包含超过五百万行代码。

多年来，基于支撑平台的各种产品已被广泛部署于包括：

- 国家电网（SGCC，全球前 2 强）
- 中石油（CNPC，全球前 5 强）
- GE（美国通用电气，全球前 5 强）
- 中国农业银行（ABC，全球 50 强）

- 兴业银行（CIB，全球 500 强）
- 光大银行（CEB）
- 华安保险（Sinosafe Insurance）
- 淘宝网（taobao.com，中国最大的电商平台）
- 法国兴业银行（SOCIETE GENERALE，欧洲第二大银行，全球 500 强）
- 德尔福汽车（Delphi，全球 500 强）
- 美国联合航空（United Airlines，全球 500 强）
- 中国联通（全球 500 强）
- 中国移动（全球 50 强）
- 贝塔斯曼（Bertelsmann，全球 500 强，全球最大的外包型呼叫中心）
- TPC 集团（法国电信企业，全球 30 万员工）
- 埃森哲（Accenture，全球 500 强）
- 艾德曼（Eldman，全球最大的公关公司）
- 中国宋庆龄基金会（由邓小平创立的国家基金）
- 宝马集团（BMW，全球 500 强）
- 陕汽集团（陕汽重卡）
- 壹基金（由李连杰、马云、马化腾、柳传志、牛根生等共同创建的慈善基金）
- 易果网（yiguo.com，中国领先的生鲜电商）
- 烟台万华集团

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。

注： 当前支持的操作系统主要包括：

- Win98/ME,WinNT4/2000/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, QNX, SMX, DOS, WinCE (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、RISC-V、IA64、MIPS、POWER、SPARC 等。

快速概览

正如前文所述，支撑平台使用汇编、C/C++ 构建，包含数百万行代码和上千个成熟的通用组件。支撑平台经过多家 500 强企业实际生产环境以及多个高负载电信、互联网和分布式计算环境十多年验证。支持各主流操作系统和硬件平台。

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（详见：3.2.1 高效 IO 框架、3.3.1 Web 扩展框架、以及 3.3.2 典型 Web 案例等小节）。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。

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（详见：5.4 nSOA 基础库－libapidbc、5.4.1 SOA vs. AIO、5.4.2 nSOA 架构等小节）。

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 网络环境中支持百万节点、万亿端口量级的超大规模计算集群（详见：5.4.3 消息端口交换服务小节）。

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、2016 年 7 月 22 及 2019 年 12 月 5 日支付宝多次中断数小时；以及 2013 年 7 月 22 日、2023 年 3 月 29 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

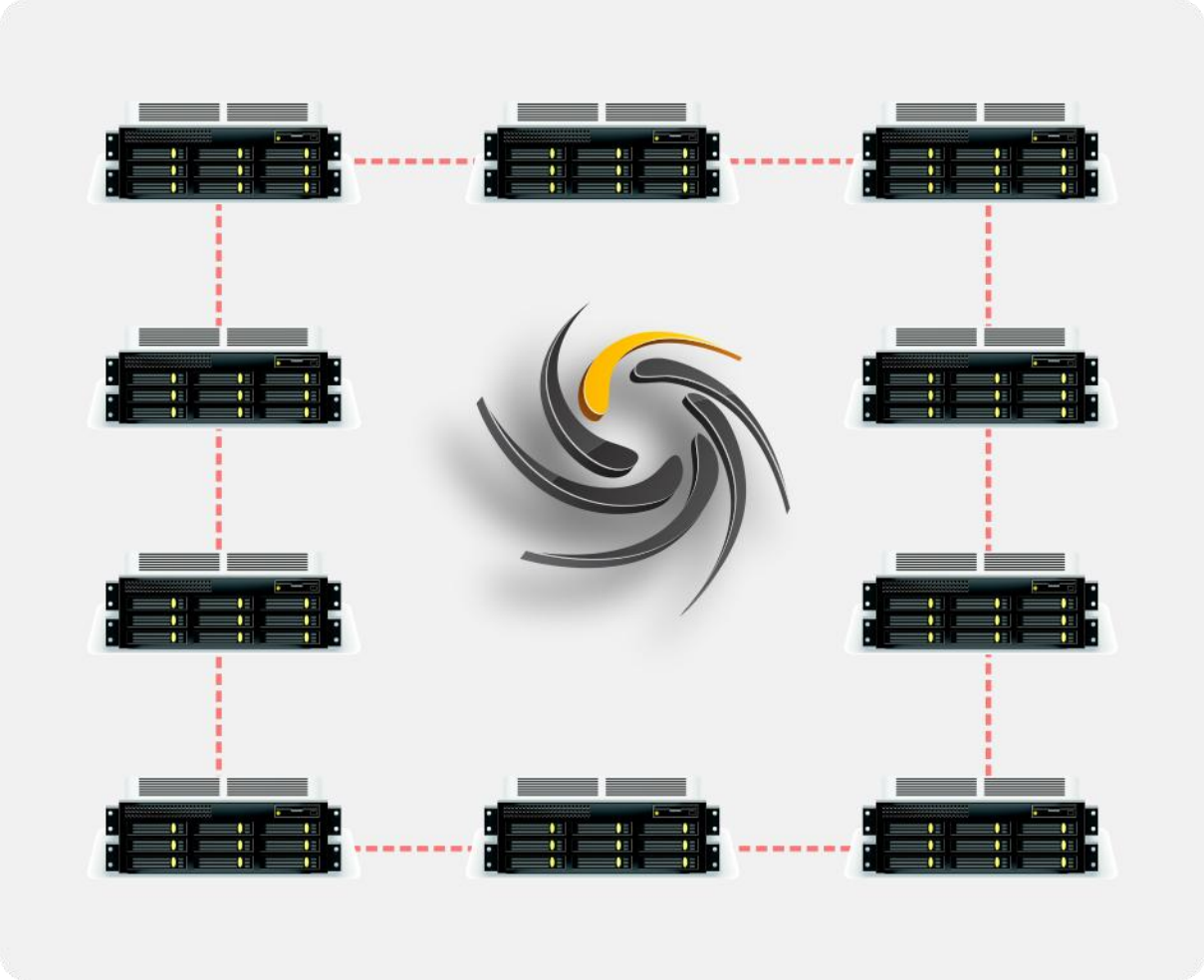


图 1

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。

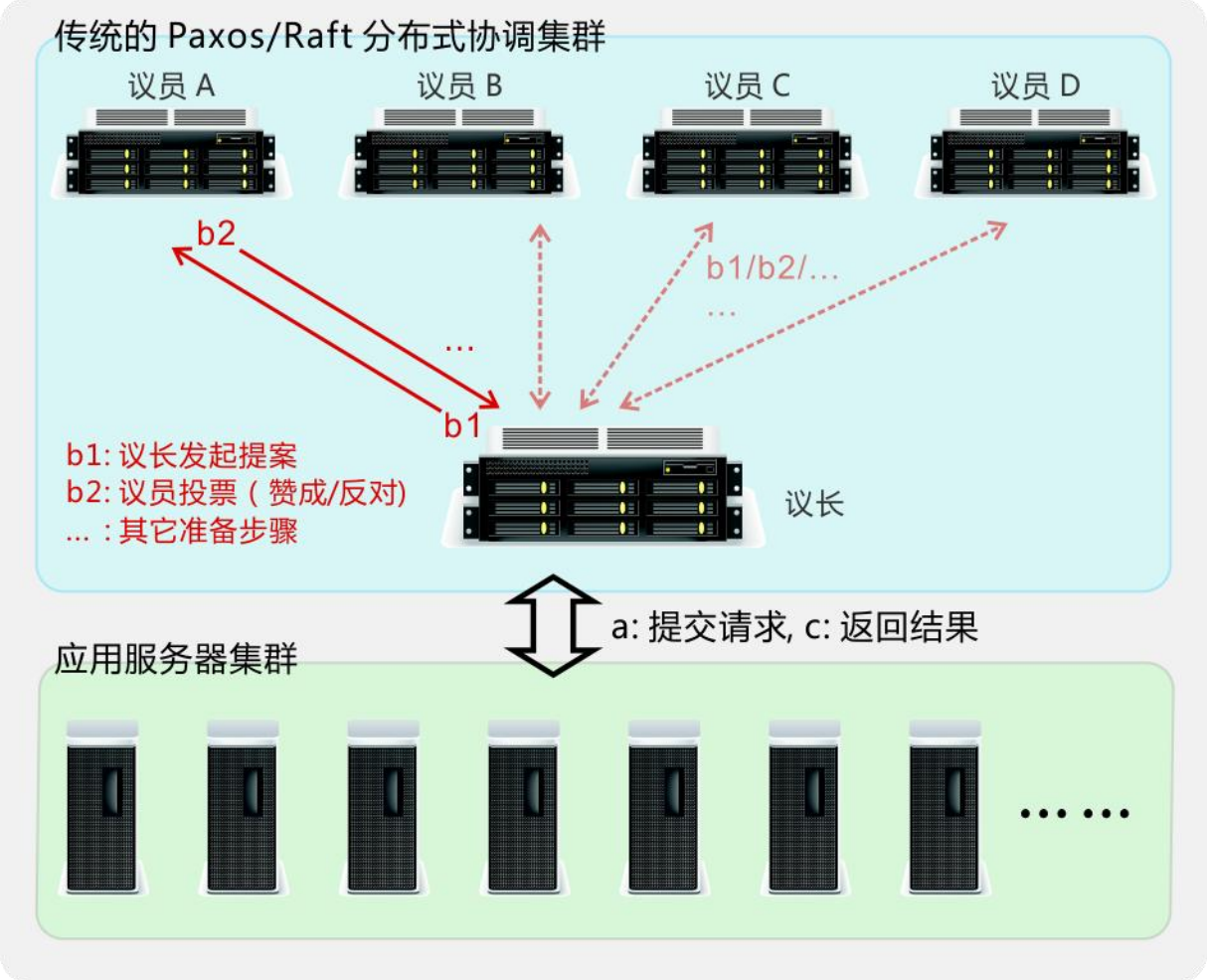


图 2

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（城域网或广域网）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。

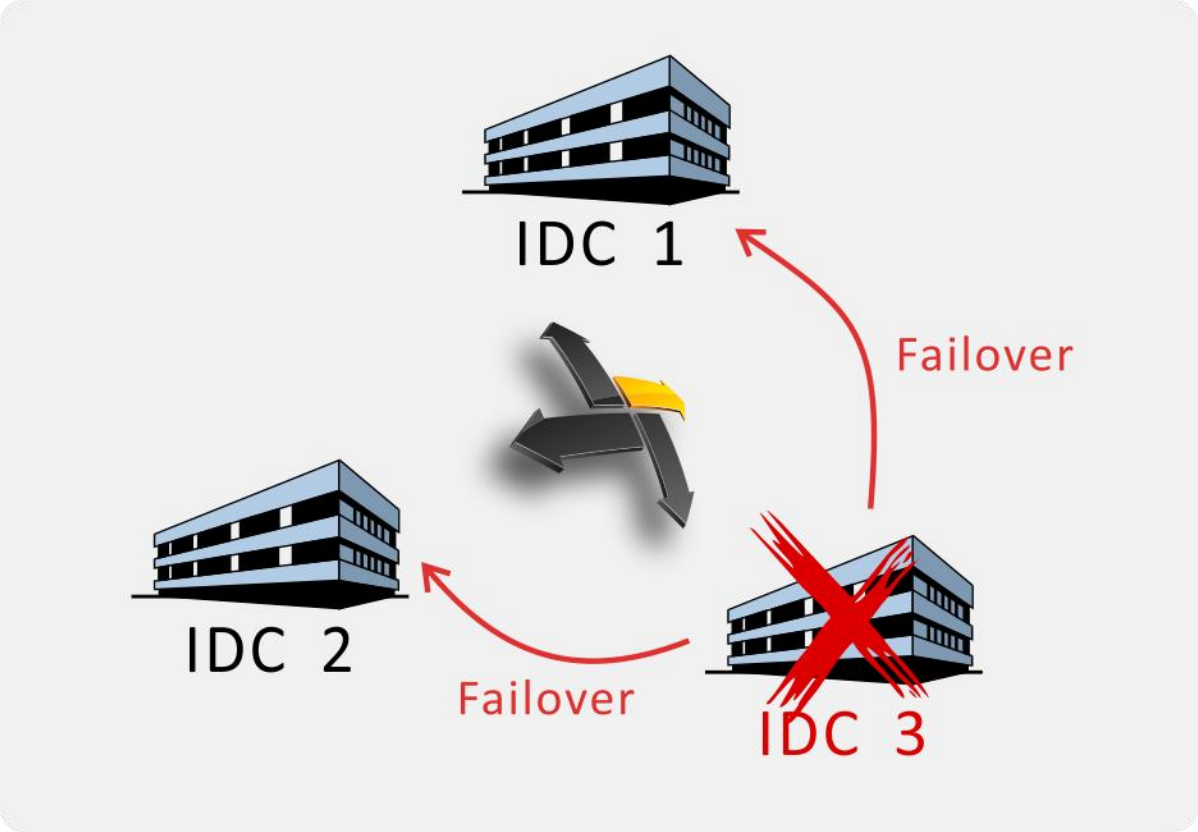


图 3

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。可在毫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：

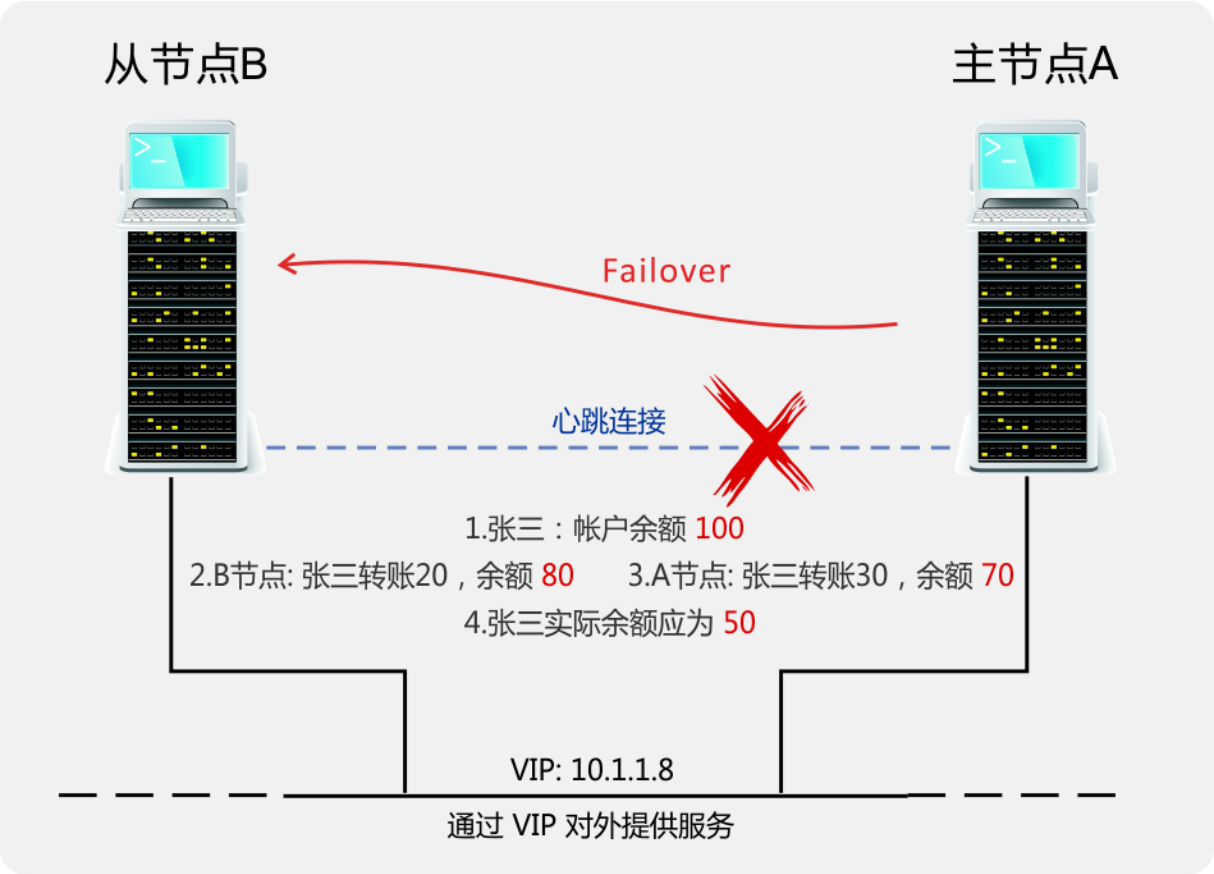


图 4

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，从根本上杜绝脑裂等不一致现象的发生。

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。

此外，Paxos / Raft 在经历过半节点同时故障下线并维修恢复的过程中，无法保证数据的强一致性，可能产生幻读等不一致问题（例如：在一个三节点集群中，节点 A 因为电力故障下线，一小时后节点 B 和 C 则因为磁盘故障下线。此时节点 A 恢复电力供应重新上线，紧接着管理员更换了节点 B 和 C 的磁盘并让它们分别恢复上线。此时整个集群 1 小时内的修改将全部丢失，集群退回到了 1 小时前 A 节点下线时的状态）。而 BYPSS 则从根本上避免了此类问题的发生，因此 BYPSS 拥有比 Paxos / Raft 更强的一致性保证。

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。

问题的本质在于 Paxos / Raft 等算法中，超过 99.99%的代价都消耗在了网络广播（投票）和落盘等行为上。而这些行为的目的就是要保证数据的可靠性（数据要同时存储在多数节点的持久化设备上）。而服务发现、服务选举、故障检测、故障转移、故障恢复、分布式锁、任务调度等分布式协调功能所涉及到的恰恰又都是没有长期保存价值的临时性数据。因此花费超过 99.99%的精力来持久化地保存它们的多个副本是毫无意义的——就算真的发生主节点下线等罕见灾难，我们也可以极高的效率，在瞬间就重新生成这些数据。

就好像张三买了一辆车，这辆车有个附加保险服务，其条款为：在张三万一发生了致命交通意外时，它能提供一种时光倒流机制，将其带回到意外发生之前的一瞬间来避免这场意外的发生。当然，这么牛的服务肯定也很贵，它大概需要预付张三家族在接下来的三生三世里能获得的所有财富。而且即使张三在驾驶这辆车的过程中，始终未发生过致命交通事故，那这些预先支付的服务费也是一分钱都不能减免的。这么昂贵的服务，且不说一般人一生中大概率都不会发生致命交通事故（更别提还要指定具体的某辆车）。即使真发生了，这个三代赤贫的代价也难说就值得吧？

而我们则为自己的汽车产品提供了另一种不同的附加服务：虽然没有时光倒流功能，但我们的服务可以在张三发生致命事故后，将所有受害方全体连车带人瞬间原地满血复活（是一根头发

丝都不会少、一块漆皮都不会掉那种满血）。最关键的是，该服务无需预先收取任何费用。张三只需要在每次这样的灾难发生以后，支付相当于其半个月的工资的再生技术服务费就可以了。

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：5.4.3 消息端口交换服务。

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（详见：4. 跨平台密码编码学算法库－libcrypto、4.1 密码编码学算法模块－algorithm 等小节）。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（详见：4.2 通用工具模块－facility）。

近年来安全问题频发，亚马逊、沃尔玛、Yahoo、Linkedin（领英）、OpenAI（ChatGPT）、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。

除此之外，我方独有的高性能网络安全隧道（BYST）组件可在保证通信安全的同时，为用户提供高性能、高吞吐和高网络利用率的 VPN 服务，在局域、城域以及广域网络中进一步帮助用户提升网络通信的性能和安全性（详见：5.6 安全隧道服务（BYST））。

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。

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（详见：3.3 通用工具模块－facility、5.4 nSOA 基础库－libapidbc 等小节）。

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，详见下文，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。

版本控制

版本号	修改时间	修改内容	修改人	审阅者
1.0	2007-07-21	创建，从老版概述文档迁移	白杨	
1.1	2007-08-09	修正文档组织结构（6.3.3 -> 6.4）；纠正个别措辞	白杨	
1.2	2008-01-04	新增 web 应用扩展库说明	白杨	
1.3	2008-03-19	加入 bz2 算法支持	白杨	
1.4	2009-12-02	更新 AIO 框架；新增 SCGI 支持	白杨	
1.5	2010-04-27	重构文档；新增 HTTP 支持	白杨	
1.6	2010-06-13	新增 Web 框架性能比较表	白杨	
1.7	2010-07-06	新增 HTTP Pipelining 描述	白杨	
1.8	2010-08-25	根据近期底层库的调整新增 LRU Cache 等组件	白杨	
1.9	2010-08-27	新增典型的 Web 应用节点工作模型说明	白杨	
1.10	2010-12-10	为 AIO 框架新增/dev/poll 和 pollset 支持	白杨	
1.11	2011-06-07	年度更新；新增 CConfig 第三方开发组件相关描述	白杨	
1.12	2012-03-15	年度更新；新增可变数据类型等组件相关描述	白杨	
1.13	2012-04-12	新增 LZ4 数据压缩算法	白杨	
2.0	2012-04-17	重构，将跨平台 GUI 框架和跨平台音频处理库转移到“6. 界面、媒体及其它工具”；新增“5. 数据处理工具”	白杨	
2.1	2012-04-19	新增 6.3 CConfig Language Binding 组件；新增 6.4 JavaScript 工具库－libbaiy 等小节	白杨	
2.2	2012-05-07	新增通用数据查询条件和查询语句解释器组件相关描述	白杨	
2.3	2012-05-19	新增查询辅助分词器相关描述	白杨	
2.4	2012-06-09	为 libbaiy 新增 JavaScript 版键树（Keyword Tree）容器相关描述	白杨	
2.5	2012-12-02	更新插画，修改部分版式	白杨	
2.6	2013-01-16	年度更新；修正一处引用失效错误	白杨	
2.7	2013-03-11	更新原子量和内存屏障相关描述	白杨	
2.8	2013-03-16	新增 SHA-3 散列算法支持	白杨	刘华松
2.9	2013-05-22	修正一处笔误	白杨	
2.10	2014-01-12	年度更新；新增跨平台的函数调用栈追踪机制	白杨	
2.11	2014-02-14	为 libbaiy.js 新增消息分发器等组件	白杨	
2.12	2014-06-07	为 libbaiy.js 新增任务队列组件	白杨	
2.13	2015-02-07	新增 libapidbc 功能库相关描述	白杨	
2.14	2015-03-23	为数据库和 memcached 服务补充关于分布式缓存及 NoSQL、NewSQL 的进一步论述	白杨	
2.15	2015-05-06	调整一些措辞；修正一些笔误；更新部分数据	白杨	
2.16	2015-05-30	一些次要更新	白杨	
2.17	2015-07-24	更新部分章节	白杨	

版本号	修改时间	修改内容	修改人	审阅者
2.18	2015-11-04	一些次要更新	白杨	
2.19	2015-12-21	更新部分数据	白杨	
2.20	2016-01-22	一些次要更新	白杨	
2.21	2016-04-13	更新部分章节	白杨	
2.22	2016-07-15	更新部分数据	白杨	
2.23	2016-10-04	新增 CRC32-C、ChaCha 以及 BLAKE2 等算法支持	白杨	
2.24	2016-12-06	新增“快速概览”小节	白杨	
2.25	2016-12-10	新增“基于 BYPSS 的高性能集群”小节	白杨	
2.26	2017-04-03	新增 CConfig 差量功能相关描述	白杨	
2.27	2017-07-07	新增“分布式协调服务”小节	白杨	
2.28	2017-09-06	新增“5.5 分布式全文搜索（FTS）服务”小节	白杨	
2.29	2018-01-13	一些次要更新	白杨	
2.30	2018-03-26	将“μ SOA”修改为“nano-SOA”，避免与“微服务”（micro-SOA）混淆	白杨	
2.31	2018-05-22	新增 ARIA、Kalyna、Simon、Speck、SM4、ThreeFish、SipHash、Poly1305、SM3 等算法支持	白杨	
2.32	2019-01-10	新增 BYDMQ 分布式消息队列组件相关说明	白杨	
2.33	2019-02-13	新增 CHAM、HIGHT、LEA、SIMECK、Rabbit 以及 HC-128 等算法支持	白杨	
2.34	2019-03-23	新增 SHAKE 算法支持	白杨	
2.35	2019-04-28	新增 BYST 安全隧道组件相关说明	白杨	
2.36	2019-07-13	一些次要更新	白杨	
2.37	2020-08-18	新增全球知产证书一瞥	白杨	
2.38	2021-03-03	新增一些 BYST 相关的补充说明	白杨	
2.39	2021-10-16	新增 LSH 散列算法和对应 HMAC 算法支持	白杨	
2.40	2021-10-26	个别次要更新	白杨	
2.41	2021-10-31	为 5.6 安全隧道服务（BYST）新增总结性描述	白杨	
2.42	2022-02-20	澄清个别 BYPSS/BYDMQ 相关细节；修正个别笔误	白杨	
2.43	2022-06-01	为 BYPSS 高性能集群新增与传统分布式缓存+DB 架构的对比总结	白杨	
2.44	2023-02-25	更新典型客户列表和专利列表	白杨	
2.45	2023-03-28	为 BYPSS 和 BYDMQ 新增毫秒级 failover 支持	白杨	

目录

1. 概述

应用支撑平台是一个产品的基石，也是产品与操作系统之间的通信接口，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。作为重要的通用组件，应用支撑平台在快速开发高质量跨平台应用中起着重要作用。

应用支撑平台为产品中的其它组件提供了众多通用功能，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、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、服务器框架等等。是系统实现跨平台、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、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、音频 IO、音频格式编解码、音频效果器、HTTP 协议和 Web 应用扩展等等。
* **跨平台数据处理功能**：包括支持 Excel 和 HTML 格式的跨平台报表生成库；支持 ODBC 和 ISO SQL/CLI 接口的数据库操作组件；以及 SQLite 嵌入式数据库引擎封装组件等。
* **分布式计算支持**：提出了 nano-SOA 架构及相关支持组件，包括：跨平台的插件接口，通用 API 注册和分发管理工具，以及支持强加密、数据分片和 CAS 风格乐观锁式更新算法的数据库连接器（DBC）接口，并实现了各类常用 DBC 插件。以及高可用、强一致、高性能的分布式协调和消息交换服务。
* **跨平台 GUI 设计框架**：封装各类窗体、控件、系统消息机制等系统相关功能，为 GUI 应用提供一个统一的，平台无关的设计框架。
* **平台无关的国际化支持**：为报表生成和 GUI 框架等各组件提供平台无关的多语言、国际化环境。

2. 总体构架

作为所有其它组件的基础，应用支撑平台在开发环境（硬件平台、编译环境和操作系统）与软件设计师之间提供一层平台无关的封装，为开发者提供一套可靠、高效、易用的跨平台通用功能组件和设计框架。

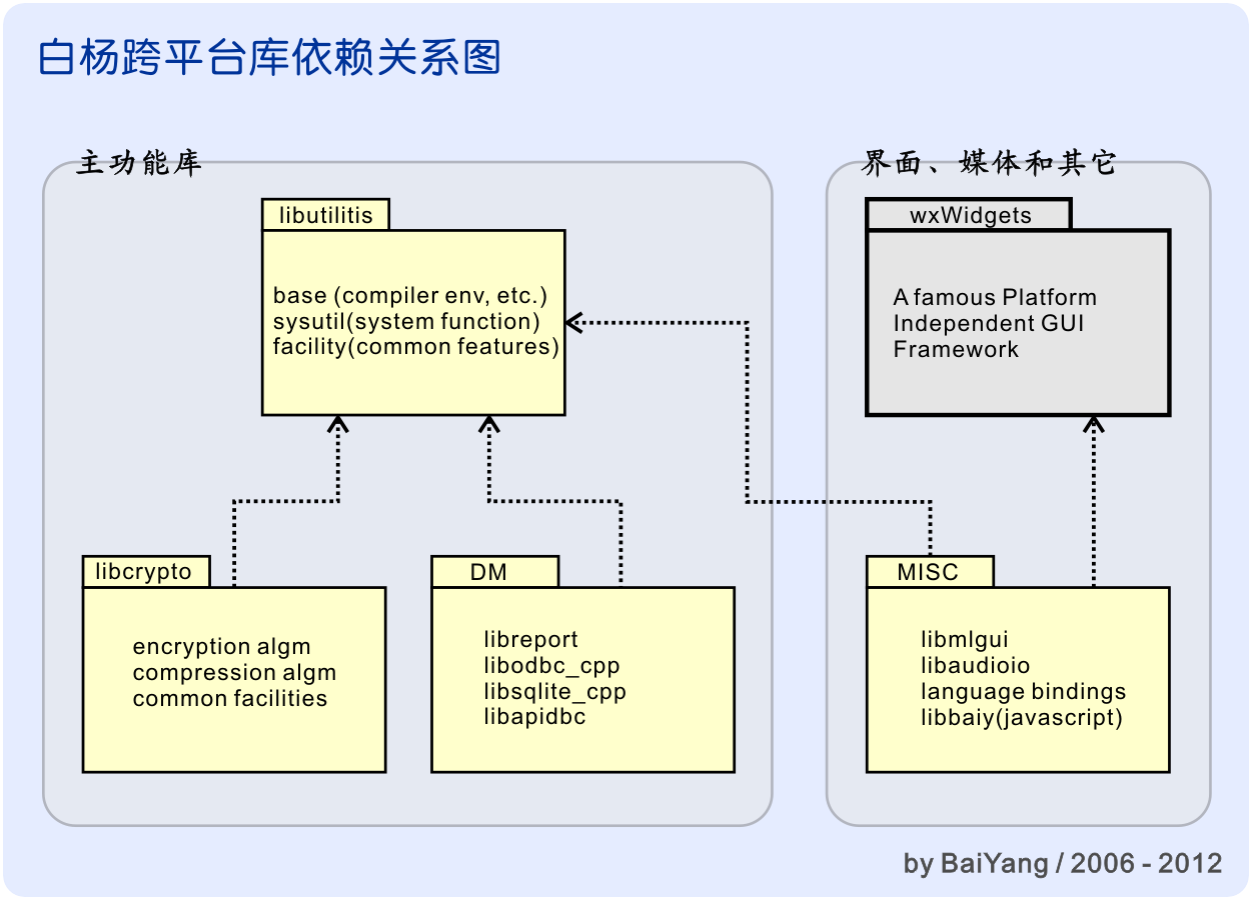


图 5

如图 5所示，应用支撑平台由以下几个相互关联的组件构成：

* **libutilitis**：封装所有与硬件平台、编译环境及操作系统相关的基础功能，并以此为基础，提供一般性的通用功能和设计框架。
* **libcrypto**：依赖 libutilitis 和第三方密码编码学算法库及压缩算法库实现。封装所有密码编码学和数据压缩、编码算法，并以此为基础实现一组与之相关的通用功能。
* 数据处理功能（DM），包括：
  - **libreport**：基于 libutilitis 实现，支持 Excel 2.0（BIFF）、Excel XP（ExcelML）、Excel 2007（xlsx）以及 HTML 等格式的跨平台报表生成库。支持自定义模板、自定义变

量、图表生成和国际化等功能。

- **libodbc_cpp**：基于 libutilitis 实现，作为 ODBC / ISO CLI 接口的 C++ 封装库，支持语句预编译、参数绑定、零拷贝结果集获取（结果集字段预绑定）等功能。
- **libsqlite_cpp**：基于 libutilitis、libcrypto 和 SQLite 引擎实现，是 SQLite 嵌入式数据库引擎的 C++ 封装库，支持语句预编译和参数绑定等功能。同时提供了基于 VFS 的整库强加密能力。
- **libapidbc**：基于 libutilitis、libcrypto、libodbc_cpp 和 libsqlite_cpp 实现，首先定义了一套跨平台的通用插件接口，并以此接口实现了一套常用数据库连接中间件。最后为各插件间的通信定义了一套完整的 API 注册、分发和请求排队工具。

* 界面和媒体库，包括：

- **libaudioio**：基于 libutilitis 实现，为用户提供一套平台无关的音频 IO 机制，同时提供各类音频格式的编解码器和一些通用效果滤镜，并在此基础上提供音频播放和录制等通用工具。
- **libmlgui**：基于 libutilitis 和 wxWidgets 框架实现。为用户提供一整套平台无关的国际化图形界面框架以及相关的配套通用功能。

在以下各章节中，我们将逐一介绍这些组件。

3. 跨平台基础功能库 – libutilitis

从体系架构上看，支撑平台位于整个产品的最底层，而 libutilitis 又是应用支撑平台的基础构造。libutilitis 的主要工作是封装所有与底层硬件平台、编译环境及操作系统相关的各种细节，向上呈现一套易用、一致、平台无关的开发接口。同时在此基础上提供一些常用工具和功能框架。

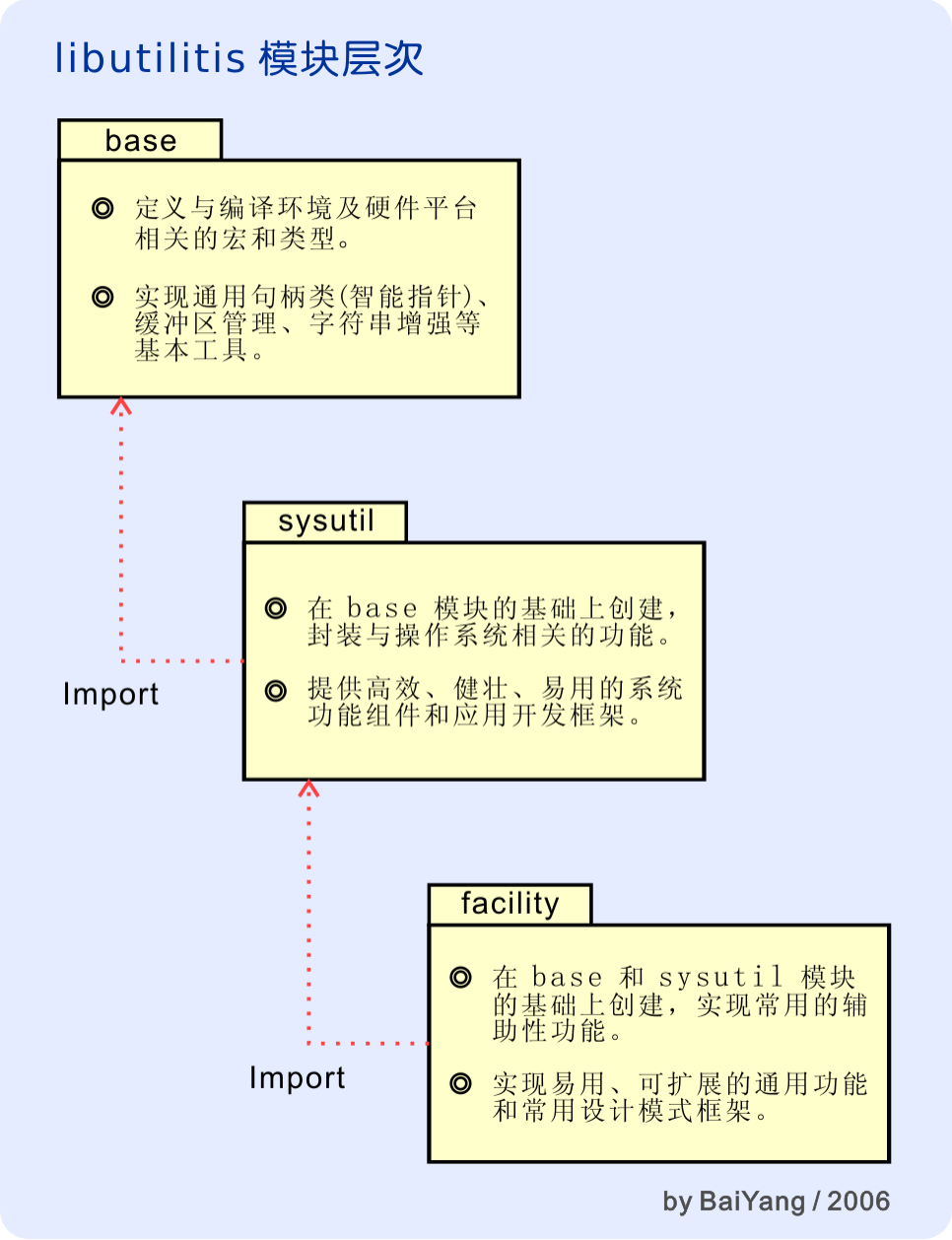


图 6

如图 6所示，libutilitis 本身又可分为如下 3 个相互依赖的模块：

3.1 基础模块 – base

基础模块封装硬件平台和编译环境相关的细节，为上层提供一致的基本工具。

在着手开发一个软件模块的时候，我们总是希望能为用户提供最强大的功能、最灵活易用的界面、所有细节都实现的健壮、敏捷、优雅而富于美感，又不失效率……遗憾的是，在大多数时候，能够同时满足这些条件的事物并非当前人类文明所能企及。很多时候，我们不得不在一些方面做出痛苦的妥协和折衷。

因此，在开始设计前，事先权衡各个要素的重要程度是十分有必要的。这使我们能够开发出行为一致，容易理解的界面。base 模块的设计权衡按优先顺序依次为：

1. **可靠性（健壮性），正确性**：要么执行任务，要么显式通知用户发生了错误。
2. **效率**：保证正确可靠的前提下，尽可能提高效率。
3. **易用性**：接口尽可能易于理解和使用，在可能产生非预期结果的位置给出醒目提示。
4. **可移植性**：尽可能降低软件在不同平台间移植的工作量。
5. **可维护性，可扩展性**：定义清晰的内部层次结构，在均衡以上各项目标后，尽可能保持易于扩展和维护的体系架构。

按照与实现细节的相关程度，基础模块还可以分为以下两个部分：

3.1.1 基础模块底层

基础模块的底层部分直接处理与硬件特性和编译环境相关的问题。为了保持最高的执行效率，这一层完全由复杂的宏魔术和成打的 typedef 构成。

底层作为整个库的基础存在，所有逻辑判断都依靠大量的宏魔术完成，在完全消除了运行时开销的同时，也使得它们难于使用和维护。libutilitis 的用户很少需要，也总是应当尽可能地避免直接使用它们（除了表意宏以外）。

同时，类似大多数可配置的库，可以通过在编译前指定或改变某些特定的开关宏来调整 libutilitis 中各种功能及行为方面的选项。

3.1.2 基础模块界面层

界面层封装底层实现细节，为用户提供更一致易用的接口。例如：

* 在不支持 64 位整型的编译环境中提供用户透明的 INT64 整型模拟。

* 为 32 位、64 位和指针类型提供 Acquire、Release 以及无屏蔽语义的原子量操作。原子量操作优先使用编译器 intrinsic/built in 方法和内联汇编实现。目前支持的平台包括：x86/x64、IA64、ARM、RISC-V、POWER、MIPS、SPARC 等。对于不支持硬件级原子量的平台，libutilitis 可通过操作系统 API 或使用基于互斥量散列集合优化的模拟实现等方式来提供原子量支持。此外，若目标平台为不提供线程支持的嵌入式环境，则所有原子量操作都将被降级为效率最高的无保护哑实现。

* 提供了针对读写、只读和只写的内存全屏障操作。与原子量支持类似，内存屏障操作优先使用编译器 intrinsic/built in 方法和内联汇编实现。目前支持硬件级内存屏障操作的平台与原子量相同。对于不支持硬件级内存屏障操作的平台，libutilitis 同样提供了使用互斥量模拟的实现。

  有关原子量和内存屏障的进一步讨论，请参考拙作《C++编码规范与指导》中的“原子操作和 volatile 关键字”一节。

* 提供了很多与平台/编译器相关的标识宏。例如：强制内联指令、DLL 符号表导出指令、内联汇编指令；以及当前硬件平台、编译器类型、编译器支持的特性（如：是否支持模板嵌套、是否支持散列容器）等等。

* 提供了一系列与底层平台相关的优化指令。如：分支预判优化、预取优化、与硬件平台类型相关的寄存器用法优化等等。

* 提供了全局对象初始化顺序保证机制。C++仅确保在同一编译单元内的全局对象按其定义顺序初始化。对于在不同编译单元内定义的全局对象的初始化顺序没有任何保证。在 GCC 等不支持自定义全局对象初始化顺序的编译器中，libutilitis 提供了一套编译器无关的全局对象初始化顺序保证机制。关于此话题的进一步讨论，详见拙作：《C++编码规范与指导》中的“全局对象初始化时的线程安全性和相互依赖性问题”一节。

* 实现了一套平台无关的调用栈跟踪工具，可获得当前上下文或指定上下文上的函数调用栈信息，包括模块名、源码文件名、行号、函数/方法名（支持 MSVC/GCC 名称粉碎）等信息。

在封装了底层实现细节同时，很多基础工具也在这里实现。例如：

* 通用句柄（带引用计数的智能指针）模板。通用句柄用于在绝大多数场合中替代传统 C 指针，其主要特性如下：

  - 自动管理：用户无需操心资源应当在何时，以及由谁来进行销毁。
  - 异常时安全：满足“资源申请即初始化（RAII）”语意，保证异常时不会产生内存泄

漏和程序错误。

- 高效：除创建、销毁和复制外，通用句柄的任何操作效率都与指针完全一致。即使在创建、销毁和复制时也仅增加了极高效的引用计数维护操作。
- 避免错误：有效防止内存泄漏和其它程序错误，大幅简化指针相关的程序设计。
- 自定义销毁策略和 NIL 值：虽然默认 NULL 为 NIL 值并使用异常安全的 delete 操作销毁对象。但程序员完全可以自定义销毁策略和 NIL 值。例如：对于操作文件句柄的模板，可以指定销毁策略为调用 close 函数，并指定 NIL 值为 INVALID_HANDLE。自定义的 NIL 值和销毁策略均作为模板参数传入并在编译时绑定，不会增加任何运行时的时间和空间开销。
- 支持静态句柄（无权绑定）。
- 支持 DontInit 方式的构造，可帮助创建高效的、多线程安全的局部静态对象。关于此话题的进一步讨论，详见拙作：《C++编码规范与指导》中的“局部静态对象初始化时的线程安全性问题”一节。
- 可以通过模板参数指定引用计数变量的类型。在使用默认的原子量类型来完成引用计数时，可保证句柄的多线程安全性。在不需要线程安全的场合，用户也可以自由选择性能更高的普通整型来实现引用计数机制。

* 临时句柄模板：临时句柄模板与前文介绍的通用句柄模板类似，也支持异常时安全的 RAII 语义、自定的销毁策略和自定义 NIL 值等特性。与通用句柄相比，唯一的区别就在于临时句柄不支持引用计数，用户需要通过显式地放弃所有权来进行指针传递。与常用来完成函数间及线程间对象传递的通用句柄不同，临时句柄通常仅在一个函数或代码块内保证 RAII 语义和异常时的安全性。由于无需引用计数，故其所有操作的时空效率都完全等同于标准指针。

* basic_buffer：与标准库 basic_string 模版兼容的高效缓冲区管理工具。basic_buffer 模板与 STL basic_string 完全兼容，但提供更高的时空效率以及更精细的存储管理机制。通过支持引用计数（reference counting）和写时拷贝（copy-on-write）、内存原地重分配、缓冲区预分配以及静态（无权）缓冲区等各项技术，basic_buffer 可以提供比 basic_string 高的多的时空效率。同时，basic_buffer 也针对 BLOB（basic_buffer<BYTE>）进行了特别优化。

* 字符串扩展工具：为 basic_buffer 或 basic_string 提供附加的扩展功能。例如：支持流式操作；支持类型转换；支持各类常见的字符串解析任务；支持各类反向操作；支持 UNICODE 字符集上的 BRE/ERE/ARE（TCL 8.2）正则表达式匹配；基于回调或符号表的换码操作等等。

* 高效链表节点模板：CListNode 模板封装了双向链表的节点相关操作。相对于 std::list 来说，CListNode 提供了 O(1) 复杂度且无需内存重分配的节点分离、交换、移动等更为底层也更为灵活的链表用法。在需要使用链表的场合，用户应当首先尝试使用 std::list 容器，仅在该容器无法满足需求时，才考虑使用 CListNode 来实现专用链表。

* LRU Cache 模板：基于最近最少使用（Least Recently Used，LRU）淘汰算法的缓存管理器，提供设置、删除、匹配、遍历和管理等完整的操作集合。可由用户选择使用散列（hash_map / unordered_map）、B 树（std::map）或其它 STL 兼容容器来进行 key-value 模式的索引和匹配。缓存管理器内部使用 CListNode 维护高效的 LRU 列表。

* 其它标准库扩展，如：使用静态缓冲区并兼容 std::vector 的 fixed_vector 模板；兼容 std::deque 的环形缓冲区容器；标准 C 库文件操作的封装类；以及通用的指针/下标访问到 iterator 封装和各类成员函数适配器等等。

* 异常处理封装：对 new 时异常、未捕获异常和 delete 时异常等情况提供满足 RAII 语义的异常处理封装。关于此话题的进一步讨论，详见拙作：《C++编码规范与指导》中的“异常”和“C++异常机制的实现方式和开销分析”等章节。

* 封底的错误处理机制：libutilitis 会捕获所有应用程序未处理的严重错误，并将他们输出到全局日志对象中。这些错误包括：C++运行时的错误，例如：未捕获的异常或者异常中的异常；以及操作系统报告的错误，例如：非法内存访问等。与此同时，出错上下文上的当前函数调用栈也将一同被输出到全局日志对象中。

综上所述，基础模块封装了所有与底层平台和编译环境相关的基本功能。libutilitis 以及应用支撑平台中的其它模块高度依赖这里定义的基本工具。

3.2 系统工具模块 – sysutil

系统工具模块基于基础模块构建，封装所有操作系统相关的功能，为用户提供一个平台无关、易用、可靠的系统功能交互界面。系统工具模块在各个设计要素间的权衡及优先顺序与基础模块一致。

系统工具模块的设计目标是封装绝大多数常用的操作系统级和硬件平台级服务。我们在经典操作系统教科书中能够找到的特征和功能，在这里几乎都有对应的功能接口。例如：

* 进程控制：实现了进程创建（包括以指定用户的身份创建子进程、输入输出重定向、创建隐形进程等）；终止进程；等待进程终止；优先级和调度算法设置；可抢占性设置；处理器粘滞性设置；内存、文件句柄等资源限制设置；CPU 时间、内存尺寸等资源用量查询；进程已加载的模块查询；系统已加载的进程查询；内存映射查询；根据给定地址反查所属模块（例如：根据一个函数指针反查提供该调用的 dll/so 模块）等等。

* 线程、TLS：支持创建、运行、挂起、继续、停止、杀死等状态控制；支持优先级、调度算法和可抢占性设置；支持处理器粘滞性和最优处理器指定设置；可获取线程当前运行状态和统计信息；可主动释放时间片或强制将当前时间片赋予其它线程。支持 TLS 存储区。

* 协程：协程（co-routine），又称为纤程（fiber）、co-process、用户线程（user thread）等，

是一种比线程更轻量级的并发机制。libutilitis 支持完整的协程操作，同时提供了一个基于线程池的协程运行环境和基本的 FIFO 调度算法。通过派生一个新的类型，用户也可根据自己的需求方便地定义不同的运行时环境和调度算法。

* 信号量、互斥量、事件（条件变量）、等同步机制。同时，对于支持硬件原子量的平台，libutilitis 还实现了快速互斥量（FUTEX）和快速信号量、自旋锁等高速同步机制。快速互斥量实现了支持递归调用的全用户态互斥量，由于不需要每次加锁和解锁的过程都进入内核模式，所以大大提高了其工作效率。快速互斥量支持比 Windows Critical Section 更丰富的功能（比如：超时等待），同时效率也要稍高于 Critical Section。快速信号量与此类似。libutilitis 中的快速互斥量和快速信号量均支持自旋锁操作，并且可以自动检测当前环境的在线处理器数量，在单处理器环境中自动 fallback 到标准上锁模式。在不提供硬件级原子量支持的平台，快速信号量等同于普通信号量，快速互斥量等同于普通互斥量。因此用户无需修改代码就能够以平台无关的方式使用对当前环境来说最高效的同步手段。
* 动态库（dll/so）加载工具：平台无关的动态库加载和 api 入口定位工具。
* 文件、网络、通信设备的同步及异步 IO 操作：libutilitis 封装了文件、网络（socket，支持 IPv4 和 IPv6）、串口、并口和管道等通信设备的 IO 操作。并提供了一套平台无关的异步 IO 框架（详见下文）。
* 文件映射和共享内存：支持读、写、执行和写时拷贝（COW）等权限设置。可以将文件映射或共享内存建立在指定的内存基址上。
* 目录管理：包含完整的磁盘卷和目录管理工具。支持目录、文件和子目录的遍历、复制、移动、删除、属性和权限设置等各种操作。可获得磁盘卷的拓扑结构和文件系统信息；可取得当前平台中所有已挂载（mount）卷设备的详细信息。
* 系统时钟、时区、DST 规则和时段操作：libutilitis 提供完整并且功能丰富的时间和历法相关操作。同时支持高精度性能计数器操作。
* 高精度时钟触发器：封装由操作系统提供的高精度周期性时钟触发回调机制。
* 系统日志：将日志消息记录到 syslogd（unix）或 System Event Service（Windows）中。
* 服务管理器：添加、删除和管理当前平台或指定计算机中的服务和驱动（Windows）。
* 服务（守护进程）框架：使用这个平台无关的框架开发 Windows Service 或 Unix Daemon。
* 字符集编码转换：支持 Windows API、POSIX libiconv、IBM libicu 和 ISO C locale API。根据当前平台的设置和字符集编码智能选择最优的编码转换器。
* 平台信息获取：与基础模块（base）中的那些预定义宏不同，libutilitis 提供了在运行时动态获取当前平台相关信息的工具。可获取的信息包括：操作系统类型、产品系列、版本号、Service Pack / Patch 号、本次开机时间、内存页尺寸、CPU 类型、CPU 宽度、CPU 字节序、处理器数量等等。
* 注册表访问、用户终端（字符界面）控制等其它各类常用功能。

* 内存有效性（读、写执行权限）检查、系统管理（注销、关机、重启）、环境变量扩展等杂项功能。

sysutil 模块同时也为不支持某些特定功能的平台提供对用户透明的虚拟层。例如：在不支持注册表操作的平台中提供功能完全兼容的虚拟注册表实现等等，此类功能在不同平台间进行编译时，能够自动切换到最适用于当前平台的实现。例如：在 Windows 中优先使用系统提供的注册表服务，在其它平台则使用 libutilitis 提供的虚拟注册表服务。

此外，系统工具模块也提供了一些与底层平台密切相关的应用框架。例如：系统服务框架封装了服务程序的标准流程和工作模式。使用这个框架构建的应用程序在 POSIX 环境下表现为一个 Daemon，而在 Windows 环境下则会以 Service 的身份与系统上的 Service Manager 协同工作。以及要在下一个小节中专门讨论的高效 IO 框架等等。系统级的框架和工具对很多关键应用的构建提供了极大帮助。它们在大大降低了跨平台移植成本的同时，通过高密度的重用提高了开发效率和代码品质。

虽然 libutilitis 应当做到尽可能地在各个平台上提供一致的功能，但显而易见地，仍然存在一些无法完全消除的差异。最有代表性的例子恐怕要属服务管理器了。WinNT 系列平台提供了服务管理器，统一控制和管理当前系统内的所有后台服务和驱动程序。类似的机制在绝大多数"POSIX 类"（un*x/linux 等）环境及 DOS 等环境中并不存在。很显然，因为涉及与系统中其它组件的广泛交互，类似功能难以在缺乏操作系统支持的前提下很好地仿真。

libutilitis 设计的一条重要准则是可靠、正确和完善。libutilitis 可以选择不包含某些功能，但是，一旦将某个功能供给用户，就必须保证该功能能够正确可靠地工作。对于某类特定功能来说，libutilitis 要么不予提供，要么必须给出完整明确的操作界面（例如：不会提供一个不支持文件和子目录遍历的目录访问类。这保证了用户不会因为 libutilitis 提供的组件缺少了某些基本操作而被迫绕开它，并且自行将相同的功能再重新实现一遍）。

基于以上设计准则，libutilitis 库中可能有极少量功能无法达到完全透明的跨平台能力。关于这方面详细信息，请参考 libutilitis 的配套用户手册。

基础模块与系统工具模块一起封装了绝大多数平台相关的服务，但是在一个真实项目中，还是有可能碰到需要直接访问操作系统相关特性和硬件资源的情况。例如：项目依赖某个第三方 COM 组件；代码的热点地区需要使用内联汇编进行优化等等。

实际上，C/C++ 语言最诱人的特点之一就是在保留了高级语言容易使用、功能丰富等优点的同时，提供了非常高的执行效率和直接访问底层硬件的能力。libutilitis 的设计目的从来就不是为用户在这些任务上设置屏障，相反地，libutilitis 致力于提供了一组专门的工具，帮助用户以更优雅、更便于移植的方式达到自己的设计目标。

libutilitis 旨在针对大多数通用功能，为用户提供一套完整、高效、可靠的平无关实现。我们深切地体会到，缺少这三个条件中的任意一个，都将促使用户绕开 libutilitis，转而自己实现某些对他（或对他的项目）而言十分关键的功能。libutilitis 在大量减少产品代码与底层平台直接交互的同时，也帮助用户以更结构化和更可控的方式完成那些必须直接与底层平台交互的任务。

例如：工具库中提供了用于判别当前编译环境和目标平台的宏，可以用于辨别编译器厂商、版本、编译器和标准库是否具备某方面能力；目标平台操作系统；目标平台 CPU 类型、位宽、字节序等细节信息。还包括了用来封装各编译器间不同内联汇编语法的宏，以及用于运行时动态获取平台类型和版本细节信息的工具类。

libutilitis 替用户完成大部分通用任务，并帮助用户以更方便、优雅和便于移植的方式实现那些必须的底层交互部分，最终产生代码更简洁、更健壮和更利于维护的产品。

3.2.1 高效 IO 框架

高效 IO 框架则封装了基于多线程的高并发、高负载 IO 伺服模型。总体来说，当前的 IO 伺服模型主要有以下几类：

* **模式 1**：同步、阻塞式 IO + 多线程/多进程，每连接一线程/进程的模型。这是最基本最容易实现的 IO 伺服模型，如著名的 apache web 服务器就使用这个模式工作。与此同时，这也是效率最低的伺服模式。主要问题体现在：为每个连接创建一个线程/进程开销很大；在高并发时服务器资源主要浪费在频繁的线程/进程创建和切换上；对于高并发+慢速连接 DDoS 攻击缺乏有效的防御力；对需要长连接的高并发应用支持不好（因为每个连接都要长期占用一个服务器线程或进程）等等。

* **模式 2**：同步、非阻塞 IO + 高效 poll（epoll/kqueue/event ports…） + 多线程，每就绪连接一线程的模型。此模式通过操作系统提供的高效轮询接口周期性地等待一个连接集合中的某些连接可用。然后对可用连接进行非阻塞的读写（即：从底层协议栈的接收缓冲区内读出数据或将数据拷贝到底层协议栈的发送缓冲区中），最后再次使用轮询接口进行等待。这种伺服模型的优点是可以使用很少的线程处理大量并发连接，可达到较高的时空效率。缺点是编程模型复杂，并且依赖于操作系统特定的 API。

* **模式 3**：异步 IO 模式 + 多线程，每活动连接一线程的伺服模型。在这一模型中，应用程序将需要的 IO 操作直接提交给操作系统，系统在该操作完成后通过回调机制通知应用程序。理论上，这是效率最高的 IO 伺服模式：因为在这一模型中，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。这里避免了模式 2 中的轮询等待操作和连接集合维护操作。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。此模式的主要缺点是编程模型复杂，并且其实际效率取决于底层操作系统的实现方式。

由此可见，理论上讲，模式 3 所使用的异步 IO 架构拥有最高的 IO 效率。但是实际情况却极大地取决于操作系统的实现方式。例如：Linux 和 Solaris 目前均不支持真正的，kernel 级的 socket AIO 操作，这些系统上的异步 IO 操作都是在用户模式下使用多线程+同步、阻塞式 IO（即：模式 1）模拟而成的。可以预见，使用这些系统提供的 AIO 服务只会带来严重的效率劣化。

而另一方面，由于 epoll、kqueue、port_get、/dev/poll 以及 pollset 等高效 poll 接口均已在相应的系统中实现了 O(1) 级的常量时间复杂度，再加上大部分现代操作系统均已通过针对内存页面的引用计数和写时拷贝机制完成了部分非阻塞 IO 的内存零拷贝实现。因此在实际环境中，具体何种 IO 模式能够为当前平台提供最高效率需要经由大量性能测试和内核源代码分析等工作才能最终确定。

libutilitis 实现的高效 IO 框架为用户提供了一套平台无关的 IO 机制，并且总是尝试使用对当前平台来说效率最高的 IO 伺服模式。具体说来：

* 在 WinNT 系列（NT/2k/xp/2k3/Vista/2k8/Win7....）平台上使用 Overlapped IO + IOCP。
* 在 WinCE 系列（WinCE / WinMobile）平台上使用 Overlapped IO + Event。
* 在 FreeBSD / Apple Mac OS X / HP-UX / IBM AIX 等支持内核级 AIO 的 posix 平台上使用 POSIX AIO + Realtime Signal。但由于各系统对高并发 IO 的支持方式不同，对于网络 AIO（Socket AIO）来说，有如下例外：
  - FreeBSD 中的 Socket AIO 使用 kqueue 来实现。
  - HP-UX v11 及以上版本的 Socket AIO 使用 /dev/poll 实现。
  - IBM AIX v6.1 及以上版本的 Socket AIO 使用 pollset 实现。
  - 其它平台上各类 AIO 均一致地使用 POSIX AIO + Realtime Signal 来实现。
* 在 Linux 上，使用 Nonblocking IO + epoll。
* 在 NetBSD / OpenBSD / DrangonFly 上使用 Nonblocking IO + kqueue。
* 在 (Open)Solaris 上使用 Nonblocking IO + Event Completion Framwork。
* 在 RTEMS / eCos / DOS 等不支持任何高效 IO 模型的环境下，使用线程池+阻塞 IO 模拟。

3.3 通用工具模块 – facility

通用工具模块基于以上两个模块构建，提供一些基础的常用算法、功能、设计模式和处理框架，主要用于简化项目实施和提高代码重用率。由于此模块基于 base 和 sysutil 实现，所以自然也实现了平台无关性。此模块中提供的工具包括：

* 各类常用的同步算法：提供了满足 RAII 准则的临界区、全同步、读者/写者模型、生产者/消费者模型等各类同步算法封装及相应的优化变种（如：快速信号量、快速互斥量优化、自旋锁优化等）。

* 带有时区和夏令时规则的时间、时段工具和历法工具及相应的时区、夏令时规则语句解释器。

* 支持复杂规则的命令行解析器。

* 多线程安全的消息队列：封装了用于线程间通信的高效消息队列机制，可以使用 std::deque、std::list、base 模块中定义的环形队列以及 std::priority_queue（优先级队列）等多种容器实现（作为模板参数传入）。队列使用生产者/消费者算法，并有多种针对不同用例调优的变体可供选择（作为模板参数传入，例如：使用快速互斥量和自旋锁的变体）。

  除了教科书式的经典消息队列外，libutilitis 还实现了一种允许覆盖写入的消息队列。这种消息队列允许生产者无限制的写入，但如果生产的速度超出了消费的速度而导致队列满，则新生产的元素将剔除队列中尚未被消费掉的最旧元素。这种允许溢出的消息队列主要在传递一些不需要可靠性保证，并且消息时效性很强的场合使用。

* 日志记录机制：libutilitis 为用户提供了一种分级记录日志信息的方式，用户可以为日志对象设置允许记录事件的最低级别（最低紧急程度）。每一个日志对象都可以同时绑定多个记录器（Logger）。Logge 代表一类用于存储日志信息的数据目标，例如：窗口、文件、系统日志服务等等。当用户向一个日志对象中写入日志时，它将被分发给所有绑定到这个对象上的 Logger。libutilitis 已实现了包括文件、终端窗口、标准输出设备、周期性文件、内存缓冲区、网络连接、系统日志记录服务（Windows Event Service / UNIX syslogd），以及 syslog 协议（RFC 3164）服务器在内的多种日志记录器。通过简单的派生，用户也可以非常方便地实现自己的日志记录器。

  日志对象还支持一种被称为过滤器的工具，日志过滤器为用户提供一种回调机制，监视和过滤当前对象的日志记录情况，并决定是否允许一条日志被记录。libutilitis 已为用户提供了基于通配符和正则表达式的日志消息过滤器。用户也可以非常方便地实现自己的过滤机制。

  为了提高并发性，并吸收日志浪涌时产生的延迟和性能下降，日志对象支持以非阻塞方式记录日志消息。在使用非阻塞模式时，应用程序将日志消息提交到一个消息队列，日志对象会在单独的工作线程内完成所有日志过滤和记录事务。用户无需等待日志记录被写入记录装置（如：磁盘、网络、屏幕等）即可返回继续工作。

* modem 控制功能（基于串口通信的 AT 指令控制）：支持完整的 AT 指令集、支持带超时的操作、支持拨号和背靠背连接，可用于低成本的远距离窄带传输。

* 由 Bell 实验室定义，广泛用于 Avaya 交换机及其它高可靠性领域中的 RSP（可靠会话协议，Reliable Session Protocol，基于 TCP 和串口通信类）协议。RSP 协议维护自己的收发窗口，并实现了超时重传、心跳检测，以及基于 RTT 实时自适应和消息窗口的流控算法。

* 基于消息的高效会话层协议：使用两种方式实现，AIO 版基于 libutilitis 高效 IO 框架实现，适合大型服务器等高并发、高负载场合。同步 IO 版简单易用，适合用于实现客户端和低负载服务器。

* HTTP 和 FTP 客户端：支持 FTP 被动模式、支持 HTTP Keep-Alive Connection、支持 SSL/TLS、支持 HTTP/FTP/SOCKS 代理。

* 基于高效 IO 框架和 HTTP/FastCGI/SCGI 协议的 Web 扩展框架（详见下文）。

* 键树和规则键树容器：键树是一种常用的容器，通常用于对某种键值信息的分层式前缀匹配，例如自动完成和电话区号匹配等等。规则键树的行为与普通键树相似，但是加入树中的 token 可以被指定的分隔符分成前后两部分。其中前半部分使用标准的键树命中；在前半部分命中之后，可以对匹配内容中的后续部分进行若干次可选的，用户指定的规则匹配（例如：可以追加正则表达式限定规则）。仅当规则匹配成功时，这个条目才被真正被认为是命中了的。

* 时钟触发器：与在 sysutil 模块内基于操作系统相关服务实现的高精度时钟触发器不同，这里实现的时钟触发器是基于 libutilitis 自己维护的计时线程而实现的。之所以要自行实现时钟触发器，是因为操作系统提供的高精度触发器通常有很多资源开销以及创建数量上的限制。例如：Windows 上的每个进程最多仅能同时创建 16 个高精度触发器。不但如此，创建高精度触发器还会修改整个系统的时钟中断工作频率，从而对系统性能造成整体负面影响。

  libutilitis 时钟触发器能够支持大量周期性触发的时钟任务，并且支持时钟分组模式。即：将所有时钟触发任务按照类型分为若干组，每组时钟任务都可以与其它组互不干扰地运行在专用的计时线程上。这样做有几个好处：首先，在不同线程中运行可以消除时钟间的相互干扰；其次，可以为不同的线程指定不同的时钟分辨率和优先级；再次，可以为需要较高精度（毫秒级）的时钟组打开系统时计校准功能。开启了系统时计校准功能后，该类型时钟触发器的触发间隔将根据计时线程的实际阻塞间隙和到期触发器执行消耗等多方面实际因素计算。

* 任务管理器（基于时钟触发器）：作业计划工具由两个部分组成：任务管理器和计划执行的任务。它们提供与时钟触发器非常相似的功能。实际上，任务管理器本身就是基于时钟触发器实现的。如果从头说起，每个应用程序中可以存在任意数量的计时线程（但是大部分应用程序只需要一个就足够了）；在每个计时线程中，又可以包含任意数量的时钟触发器；而作为一种特别的时钟触发器， 在每个任务管理器对象内又可以包含任意多个计划执行的任务。

  在 libutilitis 中提供以上三级定时触发体系决不是因为一时心血来潮，它们是从对实际应用的仔细观察中得来的。其中支持多个计时线程的原因已经在与时钟触发器相关的说明中详细地阐述了，下面来说说使用任务计划与直接使用时钟触发器的区别：

  - 触发器对象只能按照固定的时间间隔触发，但是可以为计划任务设置非常复杂的触发条件。

  - 多个相关触发器对象间的相互协作比较困难，但是计划任务可以按不同类别进行分组，

并方便地相互协作（通常每组任务分别放入不同的任务管理器中进行管理）。

- 计划任务支持按照不同的优先级顺序被执行。而触发器对象不支持。
- 计划任务通常在堆中创建，使用智能句柄维护，支持“fire and forget”语义。用户只需简单地创建任务，无需关心何时由谁销毁它。

* 消息处理框架：定义了支持职责链和命令模式的通用消息处理框架。并实现了消息预处理和消息分发机制。

* 原型工厂：定义了基于散列表或平衡树实现的高效原型工厂框架。

* 持久化框架：定义了对象持久化（序列化）框架，并且实现了两种对象集合序列化存储格式，其中一种可以支持随机访问，而另一种则主要针对长期归档优化。持久化的数据可以写入到任何数据目标，也可以从任意数据源读入。

* 虚拟注册表（CConfig）：提供了 Windows 注册表仿真服务。虚拟注册表的主要特性包括：基于 ISXF 格式实现，ISXF 是一种平台无关的二进制格式。这确保了虚拟注册表数据的跨平台能力和极高的读写性能；国际化，虚拟注册表中的字符串类型完全以 Unicode 字符集（UTF-8 格式）保存。辗转在各种不同语言环境的操作系统上也不会出现显示/保存乱码、数据不正确之类的问题；高效率，虚拟注册表被载入后，所有子键和值都存放在平衡树中。即使是在非常庞大的数据集中检索和访问也可以保持很高的效率；支持与 CSV、INI、JSON、XML 以及 Windows 系统注册表间的导入导出操作。

  CConfig 组件还提供了在任意两个对象之间计算差量的能力。差量（包括对值和子键的增改删等）数据使用紧凑高效的二进制格式存储。差量数据可以被应用于指定的 CConfig 对象上，以实现版本回退、多版本管理等版本控制功能。CConfig 组件还支持将二进制差量数据转换为人类可读的摘要信息，方便应用程序实现修订版审核以及变更审计等管理功能。

  由于 CConfig Schema 具备高效、跨平台、国际化、自解释、易扩展、使用灵活、以及支持 CSV、INI、JSON、XML 等常用数据交换格式等诸多优点，因此已被广泛用于不同（子）系统间的通信。为了尽可能降低合作伙伴及第三方开发商的使用成本，除了 CConfig 配置编辑器工具外，我们还分别提供了针对 JavaScript、C/C++、Java、.NET（C#、VB.NET、J#等）和 PHP 等各种常用语言的 CConfig 开发组件。

  关于 CConfig 的更多信息，请参考：6.2.4 通用图形控件；6.3 CConfig Language Binding 组件；以及 6.4 JavaScript 工具库－libbaiy。

* 字符串规则匹配：每一个字符串规则匹配表对象中，可以包含任意多个字符串匹配规则。用户可以在这些规则集合上对指定字符串进行匹配操作。libutilitis 目前支持的字符串匹配规则包括：范围规则、通配符规则、正则表达式规则和枚举规则等。

* 线程池：创建和管理可动态调整的线程池对象。

* 通用数据处理框架：libutilitis 定义了一套高效的，支持零内存拷贝的数据处理框架，并提供了丰富的数据源和数据目的，如：基于文件、网络、串口、对象队列、内存缓冲区的源和目的等。也实现了很多通用滤镜，如：容器滤镜、T 型滤镜等等。下文中将要介绍的 libaudioio 就是基于这套框架实现的。

* 虚拟文件系统（VFS）：VFS 是一个抽象的框架，任何一个可视为包含文件的目录结构都可以被封装为一个虚卷。按具体类别不同，VFS 又分为封装虚卷和基于文件的虚卷。

  libutilitis 实现的封装虚卷包括标准磁盘目录、基于 FTP 的 VFS 和基于 HTTP 的虚文件。与此同时，libutilitis 也定义了一个基本的，基于文件的虚卷：这个虚卷系统可以将包含任意多文件和子目录的文件夹打包成一个单一文件进行访问。并且支持为每个文件和目录追加任意复杂的元数据（通过为其追加一个虚拟注册表）。除此之外，libutilitis 中还有一个完全基于内存的虚卷实现。内存虚卷在创建临时数据和实现 memory cache 等场合十分有用。

  虚卷中可以包含虚卷，虚卷可以进行对性能无损的嵌套访问。此外，在 libcrypto 中还定义了一个支持实时压缩和强加密的文件型虚卷（详见下文）。

* 运行时环境管理器：运行时环境管理器提供以下通用的应用程序运行时功能：

  - 环境变量管理：维护一套应用程序自有的内部环境变量系统，并提供遍历和字符串扩展服务。变量管理器支持环境变量的递归解析以及引用操作系统环境变量的功能。
  - 文件预约删除服务：在创建一个临时文件之前，可事先进行预约删除。这样即使发生系统掉电等严重事故，也会保证最迟在下次启动此应用程序时删除这些文件。预约删除服务提供基于事务的完整性保证。
  - 临时文件生成服务：原子性地生成并打开一个临时文件。

* 基于 IP 地址和掩码的 IP 网络列表，支持 IPv4 和 IPv6。可执行的操作包括匹配、遍历、添加、删除、序列化等。可以用来实现网络黑名单或白名单。

* 系统间信息交换格式（Inter System eXchangeable Format）读写操作：系统间信息交换格式是一种平台无关、类型信息自描述的二进制数据编码格式。主要应用于数据序列化和网络信息交换等场合。ISXF 设计参考自 SUN XDR（NDR/RPC） 以及 ISO/IEC/ITU ASN.1 2002 DER/BER 二进制信息编码与交换方案，特别为 Intel 和较新的 ARM、RISC-V、MIPS 等主流处理器架构（LE 字节顺序）优化。ISXF 模板可以从任何数据源读入或向任何数据目标写入 ISXF 格式的信息。

* 可变数据类型：可变数据类型（CVarType）主要用于高度抽象的数据接口等需要使用动态类型的场合。类似于经典的 VARIANT 和_variant_t，可变数据类型实现了类似 JavaScript 等语言中的动态类型操作。与_variant_t 等实现不同的是，由于不需要考虑跨进程、跨语言传递等问题，因此 CVarType 可以通过引用计数和写时拷贝等技术实现高效的零拷贝传递，

从而大大提高时空效率。

* 实现平台无关的语言资源包，以及使用基于观察者（发布-订阅）模型的多语言控件框架。所有语言资源的匹配均为 O(1)算法，语言包中的每个条目均可包含任意条标题、信息提示、帮助信息和附加数据资源。语言包能够自动根据当前运行环境自动完成编码转换和字体匹配等工作。

* 高效 CSV、JSON 生成器和解析器，使用迭代式算法和手工优化的词法分析器，可高效低开销地实现大文件生成和解析。

* 基于文件和数据块的栈式操作。

* 周期性文件操作。提供了自动执行周期性（如：每日、每周、每月等等）新建的文件操作类，并可设置最多保留的文件数。这个功能常被用于服务器等需要长期运行产品中的日志记录等情形。

* 针对动态和静态内存缓冲区的文件式读写封装。

* 通用数据查询对象，定义了通用数据查询语句。Query 语句的格式为:“(限制条件) AND (查询条件) + 排序条件 + Limit/Offset 限制 + 高级选项”。其中“限制条件”和“查询条件”均可以由任意多个子表达式组成。每个子表达式中均可包含等于、不等、属于、大于、大于等于、小于、小于等于、通配符匹配、正则匹配等各类操作，数据查询对象将负责完成操作合法性检查。

  此外，多个子表达式之间可使用 AND、OR 等连词以及 NOT 等谓词进行串联。规定 AND 操作的优先级高于 OR，但支持括号表达式以便重新定义操作优先级。“排序条件”中可指定对任意多个字段进行顺序或倒序排序。

  单独设立“限制条件”是为了方便用户实现数据访问权限控制等功能。

* 数据查询引擎：查询引擎在数据查询对象的基础上完成语法解析、语义分析、中间代码生成和优化，并在最终生成树上执行相应查询。查询引擎使用“括号表达式 > AND > OR”的优先级顺序对表达式进行解析和求值，支持短路表达式（短路求值）。为了保证通用性和灵活性，子表达式的求值动作可由用户提供的访问者完成。

  除了使应用不依赖于特有数据库产品以外，查询引擎还提供了多种 SQL 语言不支持的高级查询特性，如：支持 UNICODE 大字符集的 ARE 高级正则表达式查询、支持表中套表的关联查询、业务数据和配置数据混合交叉查询、虚拟字段查询，以及其它应用自定义查询等。

  与支撑平台中的其它组件类似，查询引擎使用 C/C++实现，其热点代码在多种主流硬件平台上均使用汇编进行优化。其高效性和可靠性已在多家 500 强企业实际生产环境中被反复

验证。即使关闭了短路表达式等优化手段，在一台 2010 出厂的 Thinkpad W510（4 核 8 线程）笔记本上，仅用其中单核单线程（Intel Core i7 1.6GHz）也可对逻辑表达式（A and B or C and D）实现每秒超过 1300 万次求值的高性能。

* 查询辅助分词器：将指定的字符串转换为方便搜索的格式，包括去除所有标点符号、将字符转换为小写形式以及建立缩写等。例如：串"_Steven.Jobs_" 将被转换为：'steven jobs' 以及 'sj'。

  除此之外，查询辅助分词器还可以按照用户指定的语种将象形文字转换为对应的拉丁表达，并且可同时支持多语种。例如，可将指定汉字串按照要求分别转换为大陆汉语拼音、台湾通用拼音、日文罗马拼音和韩语罗马拼音等不同拉丁表达以方便搜索。如：使用大陆汉语拼音转换 "Calvin •赵" 将输出 'calvin zhao' 和 'cz'，使用台湾通用拼音则产生 'calvin jhao' 和 'cj'。类似地，转换 "13 叔" 的结果为 '13 shu' 和 '13s'。

  对于多音字，分词器将输出所有可能的排列组合。 例如：处理 "单田芳" 的结果为 'chan tian fang'、'dan tian fang'、'shan tian fang'，以及 'dtf'、'ctf'、和 'stf'。

通用工具模块的设计目的是通过加强组件级重用来帮助用户进一步简化完成常见的例行任务；减少代码错误；降低代码编写和维护难度，以及提高每行业务代码的平均表达能力。

此外，由于虚函数的每对象一指针成员和每调用一次基址偏移间接引用；RTTI 涉及的 type_info 静态链表遍历和比较；以及虚基类每对象一指针成员和基类数据成员间接寻址等额外开销，在 base 和 sysutil 模块中已刻意避免了除模版以外的其它 C++ 高级特性。但在 facility 模块中则可以适当地使用它们。关于上述 C++ 高级特性的具体分析，请参考拙作：《C++ 编码规范与使用指导》中的"RTTI、虚函数和虚基类的开销分析及使用指导"一节（http://baiy.cn）。

3.3.1 Web 扩展框架

libutilitis 支持包括 HTTP（RFC2616）/FastCGI（www.fastcgi.com）/SCGI（www.python.ca/scgi/）在内的多项 web 扩展协议。以下是他们的特性比较简表：

协议	高效 IO 框架	同步 IO+线程池	Keep-Alive 支持
FastCGI	是	是	是
SCGI	是	是	否
HTTP	是	是	是

如上表所示，针对这三个协议，libutilitis 均提供了高效 IO 框架和同步 IO+线程池两种实现模式。

同步 IO+线程池架构

同步 IO+线程池的伺服模式主要用于较容易地实现低负载 web 应用。其工作模型如下图所示：

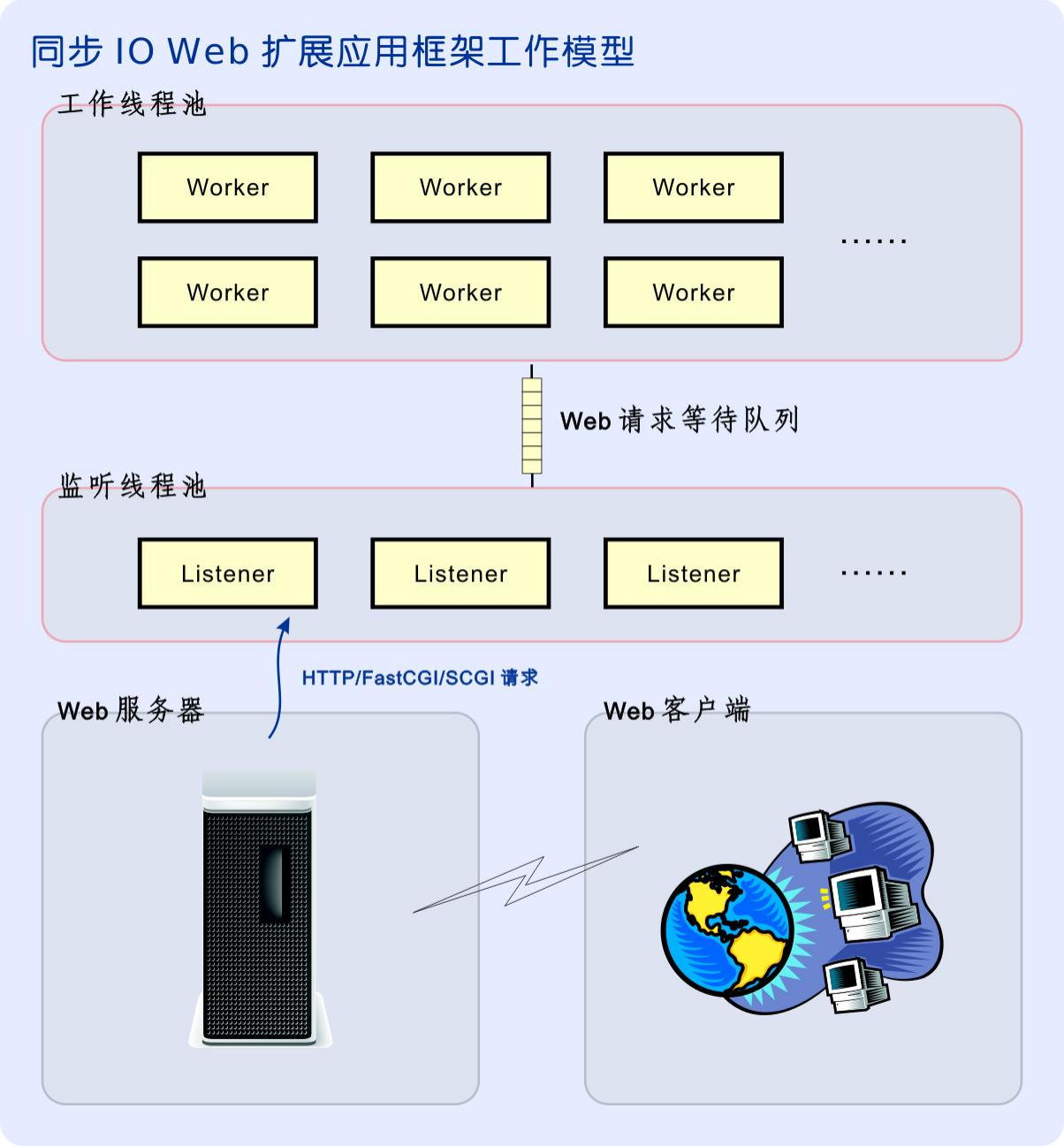


图 7

如图 7 所示，基于同步 IO 和线程池的 Web 扩展应用框架由三个主要部分组成，依次为：监听线程池、Web 请求等待队列以及工作线程池。

监听线程池等待由 Web 由反向代理或浏览器发来的 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。为了增强高负载场景下的并发性，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。

Web 请求等待队列负责将所有待处理的请求进行排队，在下文将要讨论的分段处理模型中，这个队列也负责排队尚未完全处理的请求中间状态。Web 请求等待队列的最大尺寸可由用户配置。

工作线程池不断从等待队列的首部取出待处理的请求，读取、分析客户端发来的消息内容，据此生成响应消息，并返回处理结果。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。

根据面向的应用不同，基于同步 IO 和线程池的 Web 扩展应用框架可以支持两种不同的伺服模式：

* **阻塞 IO 模式**：在阻塞 IO 模式中，工作线程一旦获得一个 Web 请求，就对其进行完整的读取、分析、计算处理、结果生成和消息返回等操作流程。等该请求处理完成后再继续下一个请求。阻塞 IO 模式是最简单、最容易实现的伺服模式。但是在高负载的复杂应用中存在并发响应能力较低的缺点。

* **分段处理模式**：在分段处理模式中，每个到达的请求都会被分成若干小节加以处理。每处理完一个小节，工作线程就将这个请求和与其相关的工作状态压入等待队列的尾端重新排队。然后从队列首部取出下一个请求继续处理。这种处理模式在高负载环境下提供比阻塞 IO 模式更均衡的服务器资源和网络带宽分配。但是需要保存用户的中间状态。

高效异步 Web 扩展框架

与同步模式不同，libutilitis 使用 sysutil 模块提供的高效 IO 框架实现了一套基于异步 IO+回调通知的**高效异步 Web 扩展框架**。即使在一台 2002 年出厂的老旧 AMD AthlonXP 2600+（单核/单线程@1.8GHz）设备上，此框架也可以轻易支撑数万并发连接。在一台 2011 出厂的双路 Intel 至强 56xx 入门级 1U PC Server 上，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 C/C++的 Web 应用框架具备几何级数的性能优势。

更有甚者，即使抛掉 .NET / Java / PHP / RoR / Python 等相对低效的应用逻辑部分，而与 Nginx / Lighttpd / Charokee 以及 IIS 等一线 Web 服务器做最直接的性能比较，Web 扩展框架的表现亦不遑多让，Apache 之类每连接一线程的低效架构自然更是被全面超越。性能评估的最佳方式莫过于进行实际测试。作为参考，以下是 Web 扩展框架和 IIS、Nginx 在不同平台上的每秒请求数（RPS）

比较：

Web 服务	平台	每秒请求数（RPS）
IIS	Windows 2k3 / TinkPad T61 Core 2 Duo Mobile @2.0GHz	33500
libutilitis	Windows 2k3 / TinkPad T61 Core 2 Duo Mobile @2.0GHz	33200
Apache	Windows 2k3 / TinkPad T61 Core 2 Duo Mobile @2.0GHz	5650
IIS/ASP.NET	Windows 2k3 / TinkPad T61 Core 2 Duo Mobile @2.0GHz	5270
Nginx	Ubuntu 8.04LTS / VMWare Single Core Guest @ Tinkpad T61	17000
libutilitis	Ubuntu 8.04LTS / VMWare Single Core Guest @ Tinkpad T61	18900
Nginx/PHP	Ubuntu 8.04LTS / VMWare Single Core Guest @ Tinkpad T61	160

表格 1: Web 扩展框架 RPS 性能比较

每秒请求数测试主要专注于检测底层应用框架在每个请求上花费的开销。为此，我们在测试时仅返回一个简单的“Hello World”页面以便将内容生成和传输方面的干扰降到最低。所有 Windows 平台上的测试都是在一台 2007 年出厂的 IBM ThinkPad T61 笔记本上完成的（配备 2.0GHz 酷睿 2 双核移动处理器和 4GB DDR2 800 双通道内存），操作系统为 Windows 2003 SP2。所有 Linux 平台上的测试均是在上述环境下的 VMWare 虚拟机中完成的（Guest 环境为单 CPU/768MB 内存），操作系统为 Ubuntu 8.04LTS。其中 Nginx 的配置已经过尽可能的优化，否则在禁用了 gzip 压缩以后的默认配置下，其测试结果仅为 12400 RPS。

上表中的所有成绩均是在连续进行 10 次测试后取平均值的结果。从测试结果中可看出，libutilitis Web 扩展框架的性能已远超 ASP.NET / PHP 等真正的竞争对手，而与 IIS / Nginx 等高性能 Web Server 不相上下。造成这一现象的直接原因就是 libutilitis 与这些产品一样，都使用了底层操作系统和硬件所支持的最高效的 IO 架构。例如：在 Windows 平台上，我们与 IIS 一样使用异步 IO+IOCP 的架构；在 Linux 平台上，我们与 Nginx 使用一样的非阻塞 IO+epoll 的架构等等。

此外，以上所有测试均是在 100 并发连接/10 万次连续请求的压力下完成的。100 并发连接既不会暴露 Apache / PHP / ASP.NET 等架构随着并发数的增高效率急剧下滑的缺陷，又能够将测试平台上的 CPU、网卡和内存等可用资源利用率发挥到最高。

当然，以上测试仅反应了 Web 应用的一个刻面。对于一个给定的，运行于指定平台上的 Web 应用框架，其性能通常可以通过以下三个刻面来分别进行观察：

* **最大并发数**：在给定的测试平台上，Web 框架能够支持的最大并发 HTTP 连接数。
* **最大每秒请求数**：在给定的测试平台上，Web 框架能够支持的最大每秒请求数。
* **动态内容生成效率**：在给定的测试平台上，实时生成图片、报表、页面等指定内容时的算法效率。

应当指出的是：在准备开始一项测试之前，应尽可能地单纯化测试环境以排除其它刻面对测试结果造成的影响。例如：前文中讨论的每秒请求数测试就是在小尺寸的“Hello World”页面和

适中的并发数量下进行的，确保在最大程度上排除了其它刻面对测试造成的干扰。

尽可能保持测量环境的单纯非常重要，这是所有现代科学的基础。例如：物理学、化学等学科中常见的“零阻力”、“绝对水平”、“1 标准大气压”、“零摄氏度”等预设环境均是如此，测试一款轿车 0-100km 加速性能时，隐含的风速（静稳）、地面情况（平整，无雨雪）、坡度（水平）等先决条件也是这样。单纯的测量环境利于发现物质的规律和特质，也便于对不同产品间的性能差异进行分项比较。

最大并发数对于 Web 框架来说绝对是一项硬性指标。它与 Web 应用的可用性、健壮性、以及在高负载/DDoS 攻击/慢速连接攻击环境下的生存力等方面有非常密切的联系。在前文所述的 T61 测试平台上，libutilitis Web 扩展框架可支撑超过 20 万并发连接的高压情形，这是 PHP/Java/ASP.NET 等其它 Web 框架绝对无法达到的。而关于每秒请求数的话题，前文已有较详细的讨论和比较。

至于动态内容生成效率，其实就是编程语言和数据库之间的效率比拼。C/C++相对于 PHP / Java / C# / Ruby / Perl / Python 等各类常用 Web 应用开发语言的性能优势已无需多言，Internet 上有很多权威的 benchmark comparison 可供参考。而各类数据库 / memory cache 产品间的性能比较由于与 Web 框架的选择相关性不大，故已超出本文讨论范畴。

我们深知，与性能相关的话题永远是充满争议并且难以单靠纯理论做出正确选择的。实践是检验真理的唯一标准，在 performance measurement 方面尤其如此。因此，我们欢迎任何 A/B 测试或性能评估的邀请。

不久之前，随着硬件性能的不断提高，程序的效率似乎已成为了只有操作系统、数据库和大型应用等少数软件才需要关心的问题。今天，日益恶化的环境问题、逐渐上升的能源成本以及渐渐成为主流的云计算和虚拟化模式（将一台物理服务器分割为多台 VPS 使用）使应用程序的效率问题又重新受到重视。我们希望能够帮助客户持续提升产品质量、提高应用效率、提升每瓦性能因子，以达到降低能源需求、减少碳排放、以及更好地适应云计算和虚拟化环境的目标。

长连接（Keep-Alive）和流水线（HTTP Pipelining）模式

对于高并发应用来说，长连接（Keep-Alive）模式在省略了多次请求之间不断重复的 TCP 建链（三次握手）、拆链（四次挥手）和流控初始化等操作的同时，也大大节约系统端口资源（TIME-WAIT 池），因此对高性能网络应用至关重要。FastCGI 通过 Begin Request 消息中的 FCGI_KEEP_CONN 标志位开启 Keep-Alive 模式，HTTP 则通过标准的 Connection: 消息头开启或关闭 Keep-Alive 模式。libutilitis 为 HTTP（1.1 或 1.0）以及 FastCGI 提供了良好且符合规范的 Keep-Alive 支持。

在启用了长连接的环境中，libutilitis 还提供了满足 HTTP1.1 规范的流水线（HTTP Pipelining）支持。在流水线模式中，客户端可以连续地发送多个请求而不必等待服务器返回响应：

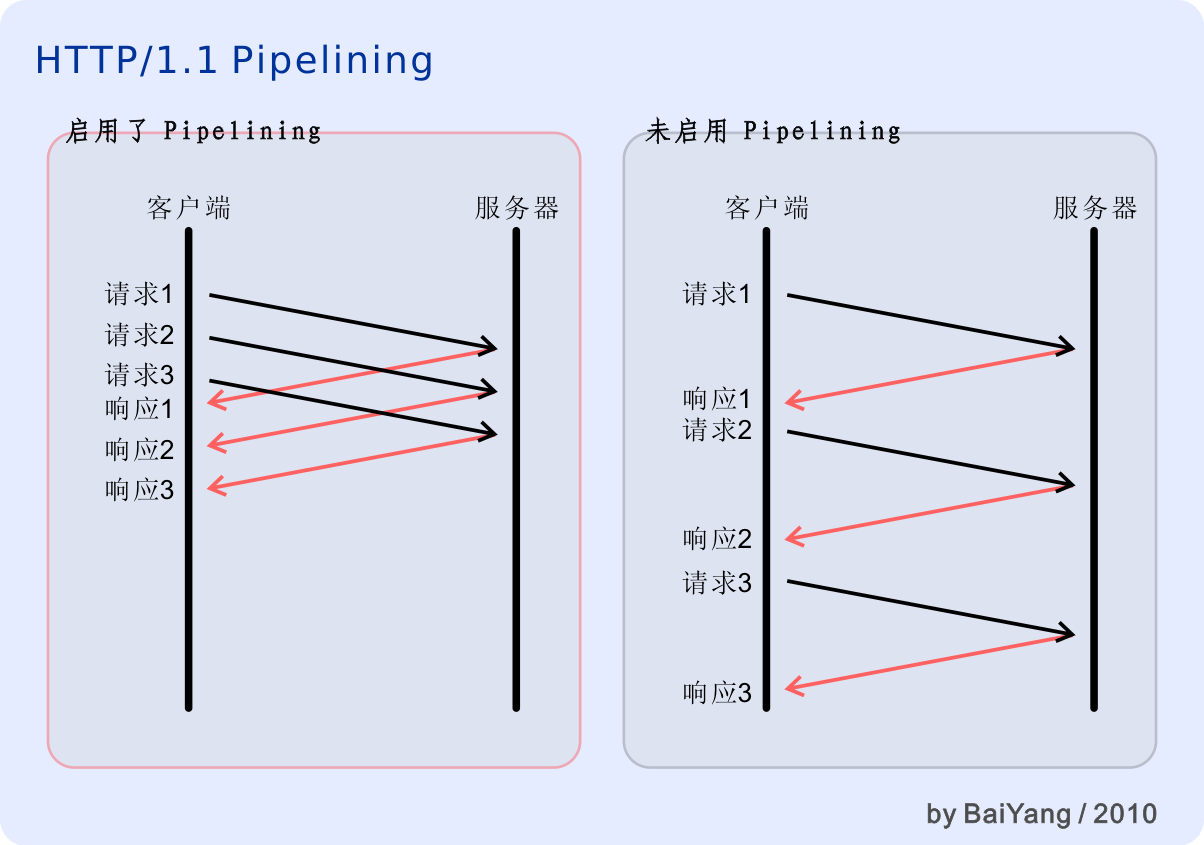


图 8

如图 8所示，通过连续不断地发送多个请求，HTTP Pipelining 技术避免了原有的“停-等”协议。因此在很大程度上降低了通信和处理延迟；提高了网络利用率和吞吐量，从而在整体上大幅改善了用户体验。

3.3.2 典型 Web 案例

以下是一个典型的高负载 web 应用示例：

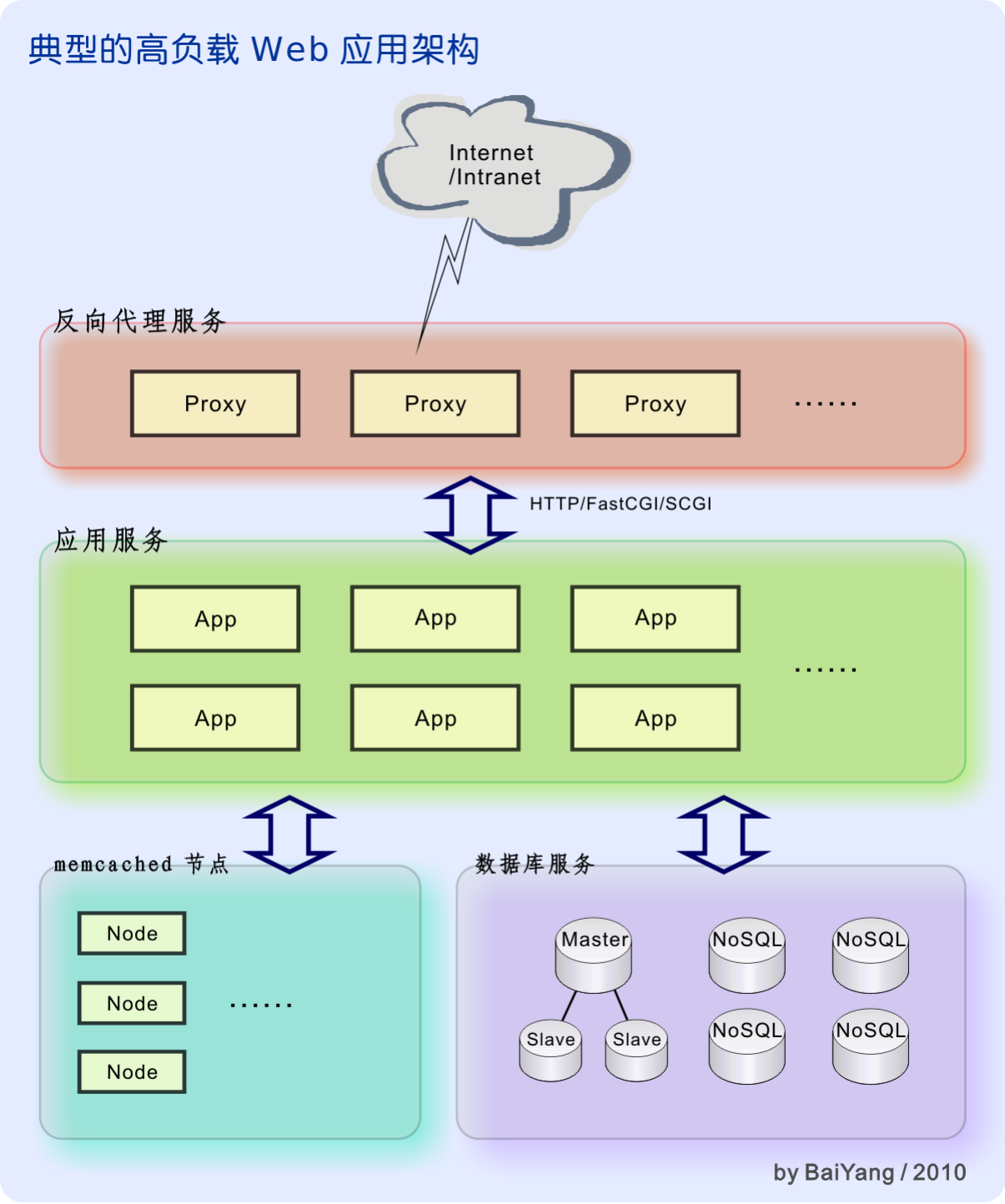


图 9

图 9 展示了一个典型的，三层架构的高性能 Web 应用。这种成熟的架构多年以来已被广泛部署于包括 Google、Yahoo、Facebook、Twitter、Wikipedia 在内的诸多大型 Web 应用中。

反向代理服务

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，代理服务器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内容生成任务，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服务漏洞所影响。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：通常使用轮转（Round Robin）或最少连接数优先等策略完成基于客户请求的负载均衡；也可以使用 SSI 等技术将一个客户请求拆分成若干并行计算部分分别提交到多个应用服务器。

分布式的 cache 加速：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。这实际上就构成了 CDN 网络。

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，动态生成的响应结果进行缓存。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（SSL Offloading）：为与客户端的通信启用 SSL/TLS 加密保护。

故障检测和容错：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。

URL 别名：对外建立统一的 url 别名信息，屏蔽真实位置。

应用混搭：通过 SSI 和 URL 映射技术将不同的 web 应用混搭在一起。

协议转换：为使用 SCGI 和 FastCGI 等协议的后端应用提供协议转换服务。

目前比较有名的反向代理服务包括：Apache httpd+mod_proxy、IIS+ARR、Squid、Apache Traffic Server、Nginx、Cherokee、Lighttpd、HAProxy 以及 Varnish 等等。

应用服务

应用服务层位于数据库等后端通用服务层与反向代理层之间，向上接收由反向代理服务转发而来的客户端访问请求，向下访问由数据库层提供的结构化存储与数据查询服务。

应用层实现了 Web 应用的所有业务逻辑，通常要完成大量的计算和数据动态生成任务。应用层内的各个节点不一定是完全对等的，还可能以 SOA、nano-SOA 等架构拆分为不同服务集群。结合 libutilitis 提供的高效异步 Web 扩展框架，使用 C/C++可以实现功能性和有效性均远超其它竞争对手的 Web 应用。

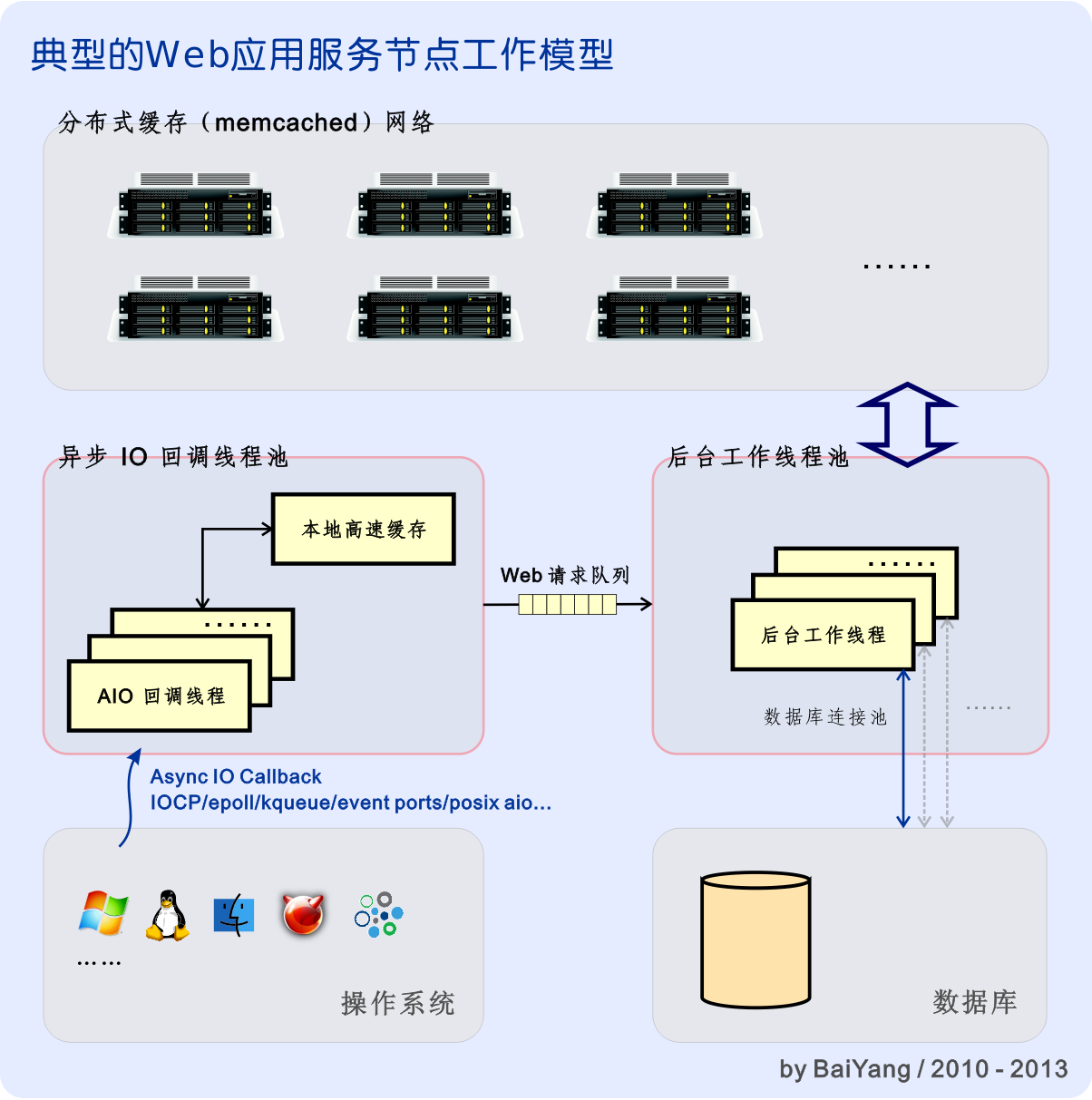


图 10

图 10 给出了一个典型的高并发、高性能应用层节点工作模型。每个 Web 应用节点（在图 9 中由标有“App”字样的方框表示）通常都会工作在自己的服务器（物理服务器或 VPS）之上，多个应用节点可以有效地并行工作，以方便地实现横向扩展。

在上图所示的例子中，Web 应用节点由 IO 回调线程池、Web 请求队列以及后台工作线程池等三个重要部分组成，其伺服流程如下：

1. 当一个 Web 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 AIO 回调线程，对这个已到达的 Web 请求进行处理。

2. 在 AIO 回调池中的工作线程接收到一个已到达的 Web 请求后，首先尝试对该请求进行预处理。在预处理过程中，将会使用位于本地的高速缓存来避免成本较高的数据库查询。如果本地缓存命中，则直接将缓存中的结果（仍然以异步 IO 的方式）返回客户端，并结束本次请求。

3. 如果指定的 Web 请求要求查询的数据无法被本地缓存命中，或者这个 Web 请求需要数据库写入操作，则该请求将被 AIO 回调线程追加到指定的队列中，等待后台工作线程池中的某个空闲线程对其进行进一步处理。

4. 后台工作线程池中的每个线程都分别维护着两条长连接：一条与底层到数据库服务相连，另一条则连接到分布式缓存（memcached）网络。通过让每个工作线程维护属于自己的长连接，后台工作线程池实现了数据库和分布式缓存连接池机制。长连接（Keep-Alive）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处理效率和网络利用率。

5. 后台工作线程在 Web 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，后台工作线程首先尝试使用分布式缓存服务命中该请求中的查询操作，如果网络缓存未命中或该请求需要数据库写入等进一步处理，则直接通过数据库操作来完成这个 Web 请求。

6. 当一个 Web 请求被处理完成后，后台工作线程会将处理结果作为 Web 响应以异步 IO 的方式返回到指定客户端。

上述步骤粗略描述了一个典型 Web 应用节点的工作方式。值得注意的是，由于设计思想和具体功能的差异，不同的 Web 应用间，无论在工作模式或架构上都可能存在很大的差异。

需要说明的是，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、内存泄露以及内存耗尽等问题，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。

作为典型案例：包括搜索引擎、Gmail 邮件服务在内的大部分 Google Web 应用均是使用 C/C++ 实现的。得益于 C/C++语言的高效和强大，Google 在为全球 Internet 用户提供最佳 Web 应用体验的同时，也实现了在其遍及全球的上百万台分布式服务器上完成一次 Web 搜索，总能耗仅需 0.0003 kW·h 的优异表现。关于 Google Web 应用架构以及硬件规模等进一步讨论，请参考：http://en.wikipedia.org/wiki/Google 以及 http://en.wikipedia.org/wiki/Google_search。

数据库和 memcached 服务

数据库服务为上层 Web 应用提供关系式或结构化的数据存储与查询支持。取决于具体用例，Web 应用可以使用数据库连接器之类的插件机制来提供对不同数据库服务的访问支持。在这种架构下，用户可以灵活地选择或变更最适合企业现阶段情况的不同数据库产品。例如：用户可以在 POC 阶段使用 SQLite 之类的嵌入式引擎完成快速部署和功能验证；而在应用的初期阶段切换到廉价的 MySQL 数据库解决方案；等到业务需求不断上升，数据库负载不断加重时再向 Clustrix、MongoDB、Cassandra、MySQL Cluster、ORACLE 等更昂贵和复杂的解决方案进行迁移。

Memcached 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 Web 应用来说，Memcached 常被用作一种重要的数据库访问加速服务，因此它不是一个必选组件。用户完全可以等到现实环境下的数据库服务出现了性能瓶颈时在部署它。值得强调的是，虽然 memcached 并不是一个必选组件，但通过其在 YouTube、Wikipedia、Amazon.com、SourceForge、Facebook、Twitter 等大型 Web 应用上的多年部署可以证明：memcached 不但能够在高负载环境下长期稳定地工作，而且可以戏剧性地提升数据查询的整体效率。有关 memcached 的进一步讨论，请参考：http://en.wikipedia.org/wiki/Memcached。

当然，我们也应该注意到：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、查询结果集等）缓存机制。既然分布式缓存有如此多的缺陷，而数据库产品又自带了优秀的缓存机制，它为何又能够成为现代高负载 Web App 中的重要基石呢？

其根本原因在于：对于十年前的技术环境来说，当时十分缺乏横向扩展能力的 RDBMS（SQL）系统已成为了严重制约 Web App 等网络应用扩大规模的瓶颈。为此，以 Google BigTable、Facebook Cassandra、MongoDB、SequoiaDB 为代表的 NoSQL 数据库产品，以及以 memcached、redis 为代

表的分布式缓存产品纷纷粉墨登场，并各自扮演了重要作用。

与 MySQL、ORACLE、DB2、MS SQL Server、PostgreSQL 等当时的“传统”SQL 数据库产品相比，无论 NoSQL 数据库还是分布式缓存产品，其本质上都是以牺牲前者的强一致性为代价，来换取更优的横向扩展能力。

应当看到，这种取舍是在当时技术条件下做出的无奈、痛苦的抉择，系统因此而变得复杂——在需要事务和强一致性保障，并且数据量较少的地方，使用无缓存层的传统 RDBMS；在一致性方面有一定妥协余地，并且读多写少的地方尽量使用分布式缓存来加速；在对一致性要求更低的大数据上使用 NoSQL；如果数据量较大，同时对一致性要求也较高，就只能尝试通过对 RDMBS 分库分表等方法来尽量解决，为此还要开发各种中间件来实现数据访问的请求分发和结果集合并等复杂操作……各种情形不一而足，而它们的相互组合和交织则再次加剧了复杂性。

回顾起来，这是一个旧秩序被打破，新秩序又尚未建立起来的混乱时代——老旧 RDBMS 缺乏横向扩展能力，无法满足新时代的大数据处理需求，又没有一种能够替代老系统地位，可同时满足大部分用户需求的普适级结构化数据管理方案。

这是一个青黄不接的时代，而 BigTable、Cassandra、memcached 等产品则分别是 Google、Facebook 以及 LiveJournal 等厂商在那个时代进行“自救”的结果。这样以：“花费最小代价，满足自身业务需求即可”为目标的产物自然不太容易具备很好的普适性。

然而今天（2015），我们终于就快要走出这个窘境。随着 Google F1、MySQL Cluster（NDB）、Clustrix、VoltDB、MemSQL、NuoDB、MyCat 等众多 NewSQL 解决方案的逐步成熟以及技术的不断进步，横向扩展能力逐渐不再成为 RDBMS 的瓶颈。今天的架构设计师完全可以在确保系统拥有足够横向扩展能力的同时，实现分布式的事务级（XA）强一致性保证：

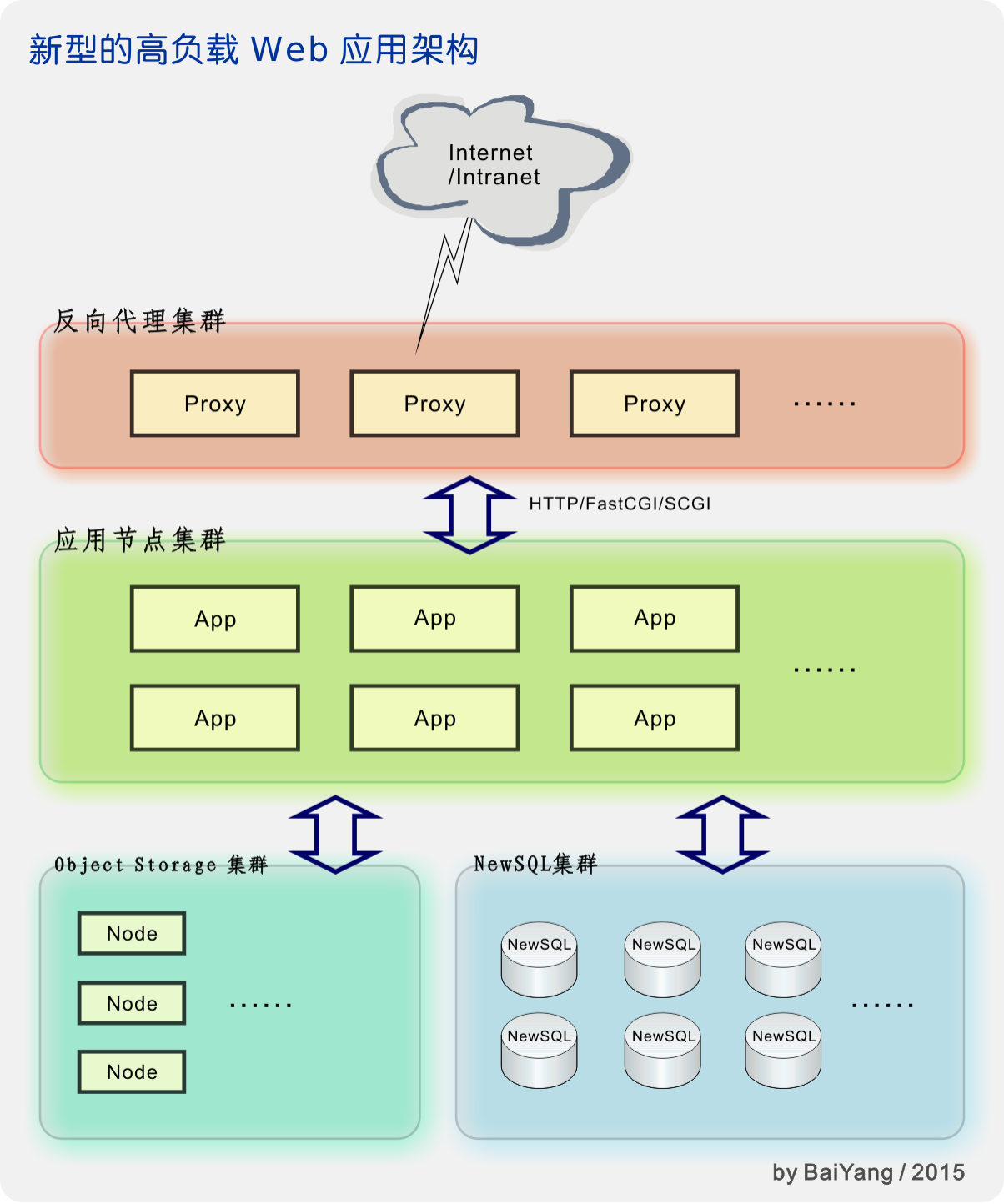


图 11

如上图所示，在 NewSQL 具备了良好的横向扩展能力后，架构中不再迫切需要分布式缓存和 NoSQL 产品来弥补这方面的短板，这使得设计和开发工作再次回归到了最初的简洁和清晰。而对象存储（Object Storage）服务则提供了对音频、视频、图片、文件包等海量非结构化 BLOB 数据的存储和访问支持。

这样简洁、清晰、朴素的架构使一切看起来仿佛回归到了多年以前，对象存储服务就像 FAT、

NTFS、Ext3 等磁盘文件系统，NewSQL 服务则好像当年 MySQL、SQL Server 等“单机版”数据库。但一切却又已不同，业务逻辑、数据库和文件存储均已演进成为支持横向扩展的高性能、高可用集群，在性能、容量、可用性、可靠性、可伸缩性等方面有了巨大的飞跃：人类总是以螺旋上升的方式不断进步——在每一次看似回归的变迁中，实包含了本质的升华。

随着 GlusterFS、Ceph、Lustre 等可 mount 且支持 Native File API 的分布式文件系统越来越成熟和完善，也有望于大部分场合下逐渐替换现有的对象存储服务。至此 Web App 架构的演进才能算是完成了一次重生——这还算不上是涅槃，当我们能够在真正意义上实现出高效、高可用的通用多虚一（Single System Image）系统时，涅槃才真正降临。那时的我们编写分布式应用与如今编写一个单机版的多线程应用将不会有任何区别——进程天然就是分布式、高可用的！

三层架构的可伸缩性

小到集中部署于单台物理服务器或 VPS 内，大到 Google 遍及全球的上百万台物理服务器所组成的分布式应用。前文描述的三层 Web 应用架构体现出了难以置信的可伸缩性。

具体来说，在项目验证、应用部署和服务运营的初期阶段，可以将以上三层服务组件集中部署于同一台物理服务器或 VPS 内。与此同时，可通过取消 memcached 服务，以及使用资源开销小并且易于部署的嵌入式数据库产品来进一步降低部署难度和系统整体资源开销。

随着项目运营的扩大和负载的持续加重，当单服务器方案和简单的纵向扩展已无法满足项目运营负荷时，用户即可通过将各组件分布式地运行在多台服务器内来达到横向扩展的目的。例如：反向代理可通过 DNS CNAME 记录轮转或 3/4 层转发（LVS、HAProxy 等）的方式实现分布式负载均衡。应用服务则可由反向代理使用基于轮转或最小负载优先等策略来实现分布式和负载均衡。此外，使用基于共享 IP 的服务器集群方案也能够实现负载均衡和容错机制。

与此类似，memcached 和数据库产品也都有自己的分布式运算、负载均衡以及容错方案。此外，数据库访问性能瓶颈可通过切换至 NoSQL/NewSQL 数据库产品，或使用主-从数据库加复制等方式来提升。而传统 SQL 数据库查询性能则可通过部署 memcached 或类似服务来改善。

3.3.3 FastCGI？SCGI？HTTP！

虽然 libutilitis 同时支持了三种协议，但是在没有其它理由的前提下，我们推荐使用 HTTP 作为构建 Web 应用的首选协议。理由很简单：首先，由于无需协议转换，并且能够很好地支持 Keep-Alive 长连接，因此 HTTP 能够提供最高的效率。其次，作为当今使用最广泛的协议，HTTP 拥有最高的产品支持度，各家产品对 HTTP 的实现通常也是最稳定的。

如果由于某些原因无法使用 HTTP 的话（例如：需要部署基于 IIS 6.0 或更早版本的反向代理服务），那么通常的第二选项应该是 SCGI，相比 FastCGI 来说，SCGI 协议最大的优势就是简单。

越简单的协议实现起来越容易，当然也就越不容易产生缺陷。作为一种复杂的，基于消息的，支持多路复用和长连接的网络协议，FastCGI 在很多 Web 服务器上没有被正确地实现。例如：截止至当前版本，IIS（版本 7.5）和 Apache（版本 2.2）上的远程 FastCGI 插件都存在不同程度的缺陷。

同时，由于避免了 FastCGI 繁琐的消息分类打包机制，SCGI 也更容易被高效地实现。FastCGI 比起 SCGI 来说，唯一的优势只有支持长连接这个特性了。遗憾的是，在 IIS、Apache、Nginx、Lighttpd、Zeus 以及 Cherokee 等支持 FastCGI 的主流 Web 服务器中，只有 Apache 准备在即将推出的版本 2.3 中的 mod_proxy_fcgi 模块内支持 FastCGI 长连接特性（当然，Nginx、Lighttpd、Zeus 和 Cherokee 都分别提供了高效且正确，但不支持长连接的 FastCGI 扩展，但它们与 SCGI 相比没有任何优势）。因此，由于 FastCGI 协议复杂，各家的支持良莠不齐，所以在没有其它方面考量的前提下，SCGI 应作为开发 Web 应用的第二选择。

4. 跨平台密码编码学算法库 – libcrypto

密码编码学算法库封装所有由应用支撑平台提供的，与密码编码学相关的算法和工具。由于是基于 libutilitis 实现，libcrypto 可以免除几乎所有算法优化部分以外的平台相关操作。

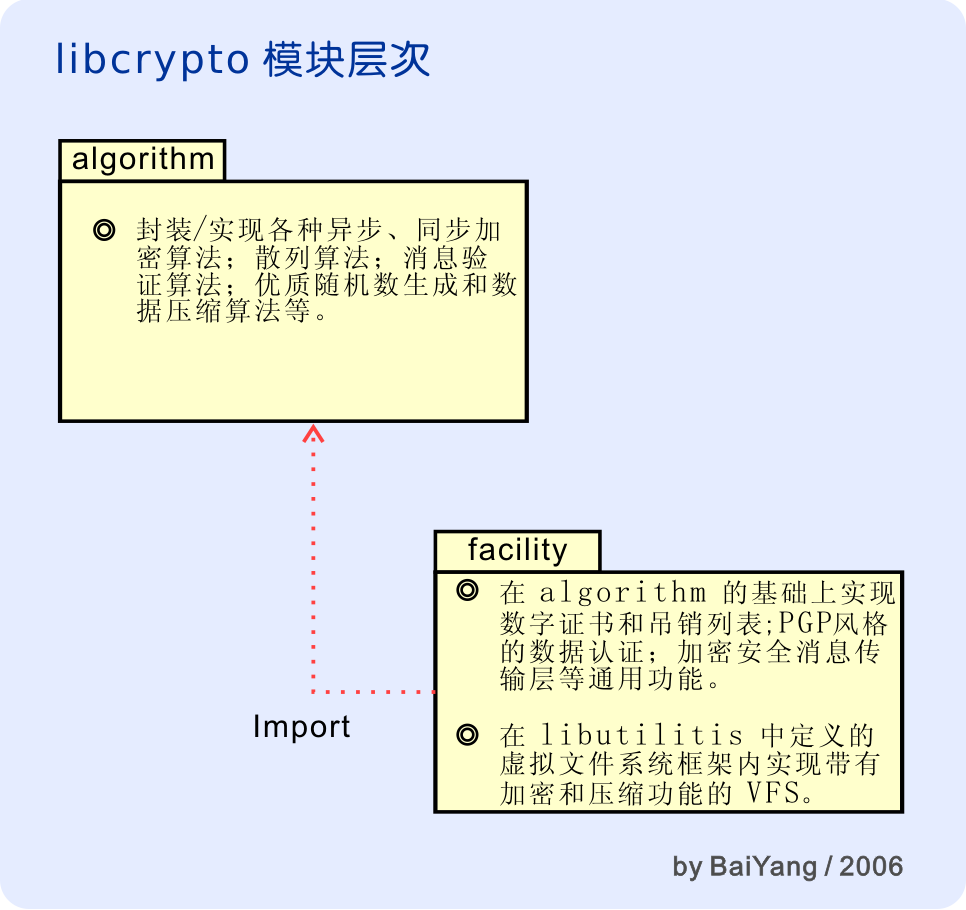


图 12

如图 12 所示，libcrypto 库使用两层结构实现。以下分别说明：

4.1 密码编码学算法模块 – algorithm

密码编码学算法模块封装所有基础算法。由于当今流行的几乎所有相关算法均可以通过 Internet 等渠道免费获得，这就极大的简化了我们的实现和调试工作。我们只需将它们归类并统一进行封装即可。

algorithm 模块目前支持以下算法：

4.1.1 支持的块加密算法

算法	支持的密钥长度
AES	128bit、192bit、256bit
BlowFish	32bit、64bit、96bit、128bit、192bit、256bit、384bit、448bit
IDEA	128bit
MARS	128bit、192bit、256bit、384bit、448bit
DES	56bit
DES-EDE2	128bit
DES-EDE3	192bit
CAST5	40bit、64bit、96bit、128bit
CAST6	128bit、160bit、192bit、224bit、256bit
SAFER-K	64bit、128bit
SAFER-SK	64bit、128bit（SAFER-K 的加强版，修正了其密钥计划攻击弱点）
TwoFish	32bit、64bit、96bit、128bit、192bit、256bit
Serpent	32bit、64bit、96bit、128bit、192bit、256bit
ARIA	128bit、192bit、256bit
Kalyna	128bit、256bit、512bit
Simon	96bit、128bit、192bit、256bit
Speck	96bit、128bit、192bit、256bit
SM4	128bit
ThreeFish	256bit、512bit、1024bit
CHAM	128bit、256bit
HIGHT	128bit
LEA	128bit、192bit、256bit
SIMECK	64bit、128bit

对于以上所有算法，均支持如下加密模式：

```
缩写说明：
  IN    — 输入向量
  OUT   — 输出向量（未用于和明文加密前）
  ENC   — 加密算法
  K     — 加密密钥
  P     — 明文
  C     — 密文
  XOR   — 异或
  <<    — 左移
  BSIZE — 算法的加密块尺寸
  COUNT — 计数器
```

* **计数器（CTR）模式：**IN(N) = ENC(K, COUNT++), C(N) = IN(N) XOR P(N)； CTR 模式

被广泛用于 ATM 网络安全和 IPSec 应用中，相对于其它模式而言，CRT 模式具有如下特点：

- 硬件效率：允许同时处理多块明文 / 密文。
- 软件效率：允许并行计算，可以很好地利用 CPU 流水等并行技术。
- 预处理：算法和加密盒的输出不依靠明文和密文的输入，因此如果有足够的保证安全的存储器，加密算法将仅仅是一系列异或运算，这将极大地提高吞吐量。
- 随机访问：第 i 块密文的解密不依赖于第 i-1 块密文，提供很高的随机访问能力。在用来加密需要随机访问的大块数据（如：虚拟文件系统）时，随机访问能力也有效增加了加密强度，这使得我们在加密每一个数据块时不用让算法恢复初始状态重新开始一轮加密。而不支持随机访问的模式，由于在加密每一个数据块前都要进行重置，在这种情况下只能工作在类似于电码本（ECB）的模式。
- 可证明的安全性：能够证明 CTR 至少和其他模式一样安全（CBC, CFB, OFB, ...）
- 简单性：与其它模式不同，CTR 模式仅要求实现加密算法，但不要求实现解密算法。对于 AES 等加/解密本质上不同的算法来说，这种简化是巨大的。
- 无填充，可以高效地替代流式加密使用。

* **密文块链接（CBC）模式：**IN(N) = P(N) XOR C(N-1), C(N) = ENC(K, IN(N))；在 CTR 出现前广泛使用的块加密模式，用于安全的分组（迭代式）加密和认证。

* **密文反馈（CFB）模式：**IN(N) = C(N-1) << (BSIZE-j), C(N) = ENC(K, IN(N)) <<(BSIZE-j) XOR P(N)；其中 j 为每次加密的位数。CFB 模式与 CBC 模式原理相似，但一次仅处理 j 位数据，其余 BLOCKSIZE - j 位丢弃。由于以上性质，CFB 模式可以在不损失安全性的前提下，将块加密变为流式加密。但是该模式也是比较浪费的，因为在每轮加解密中都丢弃了大部分结果（j 通常为一字节——8 位，而块加密算法中每块的尺寸通常为 64、128 或 256 位不等）。

* **输出反馈（OFB）模式：**IN(N) = OUT(N-1) << (BSIZE-j), C(N) = ENC(K, IN(N)) <<(BSIZE-j) XOR P(N), OUT(N) = ENC(K, IN(N)) << (BSIZE-j)；该模式与 CFB 模式基本相同，只不过本次输入是上次迭代中尚未与明文异或时的输出。与 CFB 模式一样，OFB 模式也可以作为流加密模式使用，除此之外，由于每次迭代的输入不是上次迭代的密文，从而保证了较强的容错能力，即：对一块（一字节）密文的传输错误不会影响到后继密文。但是，由于输入没有经过密文叠加使得其抗篡改攻击的能力较差，通常需要与消息验证算法或数字签名算法配套使用。OFB 通常用于噪音水平较高的通信连接以及一般的流式应用中。

* **电码本（ECB）模式：**IN(N) = P(N), C(N) = ENC(K, IN(N))；最简单但最不安全的加密方式。每次迭代的输入都使用相同密钥进行无变换的直接加密。对于同样的明文片断，总会产生相同的，与之对应的密文段。抗重复统计和结构化分析的能力较差。一次性加密的最坏情况（即：每次输入的明文都小于等于 BSIZE 时）就是电码本模式。仅在一次一密，或交换极少数据时考虑使用 ECB 模式。

4.1.2 支持的流式加密算法

算法	支持的密钥长度
SEAL	160bit（目前最快的对称加密算法，速度 8 倍于 AES128，但虽然有免费的实现代码，但在美国地区 IBM 公司对其享有专利限制）
MARC4	32bit、64bit、96bit、128bit、192bit、256bit、384bit、448bit、512bit、768bit、1024bit、1536bit、2048bit（RSA RC4 的改良版本，去掉了 ARC4 中不太安全的前 256 字节头部）
Panama	256bit
Salsa20	128bit、256bit
XSalsa20	128bit、256bit（增强了 Salsa20 的不可预测性）
Sosemanuk	128bit、192bit、256bit
ChaCha8	128bit、256bit
ChaCha12	128bit、256bit
ChaCha20	128bit、256bit
Rabbit	128bit
HC-128	128bit、256bit

以上所有块加密和流式加密算法均由 libcrypto 中的同步加密算法对象封装实现。

4.1.3 支持的公钥算法

libcrypto 目前支持公钥宽度分别为 512、768、1024、2048、4096、8192、16384 位的 RSA 算法。并完整支持基于公钥的加密和签名操作。

4.1.4 支持的散列算法

算法	散列长度
CRC32	32bit 4Bytes
CRC32-C	32bit 4Bytes
ADLER32	32bit 4Bytes
MD5	128bit 16Bytes
MD2	128bit 16Bytes
SHA1	160bit 20Bytes
SHA224	224bit 28Bytes
SHA256	256bit 32Bytes
SHA384	384bit 48Bytes
SHA512	512bit 64Bytes
Panama	256bit 32Bytes

算法	散列长度
Whirlpool	512bit 64Bytes
TIGER	192bit 24Bytes
RIPEMD128	128bit 16Bytes
RIPEMD256	256bit 32Bytes
RIPEMD160	160bit 20Bytes
RIPEMD320	320bit 40Bytes
SHA3-224	224bit 28Bytes
SHA3-256	256bit 32Bytes
SHA3-384	384bit 48Bytes
SHA3-512	512bit 64Bytes
BLAKE2-256	256bit 32Bytes
BLAKE2-512	512bit 64Bytes
SM3	256bit 32Bytes
SHAKE128 (256)	256bit 32Bytes
SHAKE256 (512)	512bit 64Bytes
LSH-224	224bit 28Bytes
LSH-256	256bit 32Bytes
LSH-384	384bit 48Bytes
LSH-512	512bit 64Bytes

4.1.5 支持的消息验证算法

支持以上算法列表中，除 CRC32、ADLER32 和 SHA-3 以外所有散列算法所对应的 HMAC 消息验证算法。

此外，还支持 SipHash64、SipHash128 以及 Poly1305<AES> 等非 HMAC 家族的消息验证算法。

需要注意的是，SHA-3、Blake2、SHAKE 等新式散列算法不受 length-extension 攻击影响，因此无需使用复杂的 HMAC 变换，只需要将共享密钥简单拼接到消息首部即可计算出安全的消息验证码。

4.1.6 支持的数据压缩算法

目前支持 GZIP (RFC 1952)、ZLIB (RFC 1950)、BZ2、LZO 以及 LZ4（用于实时数据压缩）等算法。对以上所列算法均支持迭代式和一过式压缩。

其中 ZIP 和 BZIP 算法均成名已久，本文不再赘述。下面简单介绍一下 LZO 和 LZ4 算法。LZO 算法是著名的高效无损压缩算法，由于该算法稳定、出色的表现，被广泛用于各个领域，包括：NASA 的"机遇"号和"勇气"号火星探测器；著名的 UPX 可执行文件压缩工具等等。LZO 算法的工作效率十分惊人，其压缩时仅需 64KB 工作空间，解压缩时则完全不需要任何额外工作空间。在一台内存带宽仅（memcpy()）为 60MB/S 的 Intel Pentium 133 上，LZO 的解压和压缩速度可分别达到 20MB/S 和 5MB/S。在压缩率相差不大的前提下，比 ZIP、RAR 等其它著名压缩算法效率高出很多倍。

LZ4 是一款新兴的实时数据压缩算法。与老牌的 LZO 算法相比，在压缩率基本相当的前提下，LZ4 提供了更优的执行效率：其压缩和解压速度是 LZO 的 1 到 4 倍。

4.1.7 支持的数据编解码算法

libcrypto 目前支持最为常用的 HEX、BASE64 和 BASE64-URL 等数据编解码方案。

4.1.8 优质随机数生成算法

libcrypto 能够根据当前运行环境，通过 CryptoAPI、/dev/random 等接口获取优质随机数种子，并结合使用安全散列算法和同步加密算法产生指定长度的优质随机数序列。

4.2 通用工具模块 – facility

通用工具模块在密码编码学算法模块的基础上构建。作为包含于应用支撑平台中的标准组件，为软件设计师提供密码编码学相关的通用灵巧工具集。在通用工具模块中，包含了以下内容：

* **数字证书**：使用公钥签名和散列算法，实现满足 PKI 应用需求的数字证书和证书吊销列表（CRL）。
* **PGP 风格的加密算法**：使用公钥加密和签名算法以及同步加密算法实现类 PGP 的数据加密和签名机制。
* **安全消息传输层**：使用公钥算法、同步加密算法、消息验证算法和数据压缩算法实现面向消息的透明安全传输层。
* **支持压缩和加密选项的 VFS**：使用数据散列算法、数据压缩算法和同步加密算法，实现兼容 libutilitis 中虚拟文件系统框架的，基于文件并支持实时访问的 VFS 工具。支持迭代式密码安全变换。如果用户启用了加密选项，对文件目录列表信息和每个文件、目录的元数据也将给与同等强度的加密保护。

* **许可协议**：结合使用压缩、数字签名和内容混淆算法为用户提供一种通用的版权许可协议验证和保护工具。

5. 数据处理工具

包含报表生成、嵌入式数据库引擎以及数据库访问接口等数据处理相关组件。

5.1 报表生成库 – libreport

报表生成库基于 libutilitis 实现，可通过自定义模板为指定的数据集生成特定格式的报表。报表生成库支持以下功能：

* 可生成 Excel 2.0（BIFF）、Excel XP（ExcelML）、Excel 2007（xlsx）以及 HTML 等格式的报表。
* 不依赖 Microsoft Excel 等任何第三方组件，杜绝任何 License 问题和第三方组件依赖性问题。
* 支持导出包括折线图、柱状图、饼图、以及甘特图在内的各种图表（除 Excel 2.0 和 Excel XP 以外）。
* 可自由自定义变量和常量，可以套用公式；支持包括日期、时间、数字、文本等各种字段类型。
* 可自定义字体、图文、配色等主题风格；支持国际化。
* 高性能、低开销：支持迭代式（逐条）数据加载和生成，数据仅需一次扫描。

libreport 针对常用格式提供了功能丰富的国际化跨平台报表生成工具。杜绝对任何第三方组件的依赖不但避免了任何可能的 License 问题，而且还很好的保持了组件部署和使用上的简便性、高效性以及跨平台能力。此外，通过各大企业生产环境中的多年实践检验，其功能、性能和稳定性均十分出色。

5.2 ODBC 封装库 – libodbc_cpp

ODBC 封装库基于 libutilitis 实现，ODBC 通用数据库接口作为 ISO 标准，受到了各大平台和几乎所有数据库产品的广泛支持。此外，包括 MS SQL Server、DB2、MySQL、Firebird、PostgreSQL、MySQL Cluster、Clutrix、OceanBase、InfiniDB、MemSQL、Greenplum 以及 Teradata 在内的大部分 SQL/NewSQL 数据库产品都已将其作为 Native Client API 来实现（请注意 psqlODBC 与 libpq 之间没有依赖关系，都是 PostgreSQL 的 Native Client Library。虽然在默认编译选项下 psqlODBC 会使用 libpq 完成一些对性能要求不高的公共操作）。

ODBC 封装库实现了以下功能：

* 以 Connection、Statement 和 Result set 的概念对 ODBC 接口进行了分组和封装。
* 支持语句预编译和动态参数绑定。对所有类型的参数都支持零拷贝绑定。
* 可对查询结果集中的字段进行零拷贝预绑定或后绑定。
* 由于使用了 libutilitis 中，带引用计数和写时拷贝的字符串以及 BLOB 类型，因此所有的零拷贝绑定操作都对用户透明，用户无需为此付出任何额外的努力。
* 支持 ODBC 连接池。
* 可以方便地完成连接和语句级别的各项常用参数设置。包括超时、最大结果集尺寸、最大字段尺寸等各种通用属性。也支持直接向 DM 和 Driver 发送和查询各种自定义或高级选项。
* 支持表格/索引存在性检查、表格/数据库信息获取、中止当前操作等各类管理性操作。
* 兼容微软 32 位和 64 位 ODBC 库、unixODBC、iODBC、以及 DB2 CLI 等常用 ODBC 或 SQL/CLI 实现。

ODBC 不但受到各大平台和产品的广泛支持，也是目前工作效率最高的通用数据库接口。ODBC 的零拷贝设计避免了 OLEDB、ADO、JDBC、ADO.NET 等接口的带来的大量内存拷贝和格式转换开销。上述各数据库产品均使用 ODBC 接口实现其 Native Client Library 的事实本身就说明其接口效率十分值得称道。

除此之外，ODBC 也十分适合在不损失效率的前提下，用来规避 MySQL 等数据库产品 Client Labrary 的 CopyLeft 许可协议限制问题。

5.3 SQLite 封装库 – libsqlite_cpp

SQLite（http://sqlite.org）是一款非常著名的嵌入式数据库引擎，该产品从公开发布至今已有超过 10 年历史。SQLite 的稳定性和功能性均毋庸置疑，包括 Microsoft、Google、GE（General Electric）、Apple、Oracle（Sun）、Nokia（Symbian）、Mozilla（Firefox）、Adobe、Toshiba、McAfee 等多家知名企业在内的用户都已将 SQLite 广泛应用于他们的关键产品中。SQLite 的关键特性包括：

* 在非常宽松的协议下免费开源。
* 支持很多种操作系统和硬件平台。
* 能够可靠地支持数 T 字节、数亿条记录的大型数据库。
* 提供 ACID 保证，支持包括主键、外键、复合索引、事务、视图、触发器等在内的绝大部分 SQL92 标准功能；支持规范化的 SQL 查询语言。

* 单一数据库文件，跨平台格式。
* 在线备份支持，不间断服务的情况下进行在线备份。
* 小巧（仅 300KB），数据库引擎完全嵌入应用程序内部，无需单独安装任何数据库服务（Server less）。无需任何配置和管理性干预（Zero-configuration）。
* 效率高，内嵌式的集成引擎省去了数据传输和编解码的开销，同时支持自动查询评估和优化。

libsqlite_cpp 基于 libutilitis 和 libcrypto 等库实现，是 SQLite 的 C++ 封装，包含以下功能：

* 以 DB（Connection）、Statement 和 Result set 的概念对 SQLite 接口进行了分组和封装。
* 支持语句预编译和动态参数绑定，支持零拷贝参数绑定。
* 可对查询结果集中的字段进行零拷贝绑定。
* 可以方便地对操作超时、WAL 模式、Shared Cache 模式等各种属性进行设置。
* 支持表格/索引存在性检查、表格/数据库信息获取、中止当前操作等各类管理性操作。
* 提供支持实时强加密功能的 SQLite VFS Driver（EncVFS），可对包括主数据库、临时数据库、Attach 数据库及各类日志文件在内的所有数据进行实时强加密，确保不会发生信息泄露。EncVFS 可使用 libcrypto 中支持的所有块加密以及流式加密算法（请参考：4.1.1 支持的块加密算法、4.1.2 支持的流式加密算法）。启用 EncVFS 功能将会为 libsqlite_cpp 引入对 libcrypto 的依赖。

与此同时，libsqlite_cpp 在不影响效率和功能的前提下，尽可能地提供与 libodbc_cpp 保持一致的界面和语义，方便用户在两者间进行迁移。

5.4 nSOA 基础库 – libapidbc

libapidbc 可以被拆分为相互关联的三个部分：首先，它定义了一套跨平台的通用功能插件接口（IPlugin），带有如下特征：

* 插件通常以动态链接库（dll/so）的形式提供，仅需要对外暴露一个形如“extern "C" void* CreateInstance(void);”的接口。也可以静态库或源码的形式直接嵌入到项目中，此时无需暴露任何外部接口。
* 插件可携带和接受任意复杂的 CConfig 配置信息，配置信息被区隔为常规配置、高级配置和内部配置等多个部分，每个部分的内容可分别由插件类别或具体实现自行定义。
* 每个插件均可携带两个包含任意资源的虚拟文件系统（VFS）——用于对外服务的资源虚卷（通常包含页面、图片、语言包等资源），以及对内自用的私有虚卷（例如：包含报表格式模板、数据字段映射表等内部资源）。

* 插件可以根据处理器、操作系统、发行版（MBCS / UNICODE）等当前环境因素自动执行最优匹配。

IPlugin 定义了一套完整、自描述、灵活、可管理的接口规范。以此为基础，libapidbc 定义了数据库连接器（DBC）插件类型。DBC 提供了如下功能：

* 作为数据中间件，DBC 比 libsqlite_cpp、libodbc_cpp 等更易用，无需编写任何 SQL 或 NoSQL 语句，可直接使用与数据库产品无关的 CConfig 配置项来定义表、索引和数据分片规则等。
* 支持基于 Revision 字段的 CAS（Compare and Swap）式原子更新操作。此算法可以较好地解决多个节点争抢更新同一记录的问题。
* 提供对用户透明的数据加密服务，为底层数据库添加数据传输和存储时的可靠强加密保护。在底层产品或服务不支持存储及传输强加密的场合，将由中间件来透明地完成相应的强加密保护。
* 提供对用户透明的数据压缩服务，为底层数据库添加数据传输和存储时的实时数据压缩支持。用户可以分别为数据库中的每张表或每个集合单独配置压缩选项。透明压缩服务可与数据加密等其它服务同时启用。
* 使用 libutilitis 中的通用数据查询对象来描述复杂查询，用户无需编写任何查询语句，也不用考虑底层数据库产品兼容性。通过图形化的界面或使用数据查询对象接口即可构造数据库产品无关的复杂查询表达式。
* 智能地使用 libutilitis 中的数据查询引擎来提供诸如：支持 UNICODE 和 ARE 规则的正则匹配、支持表中套表的关联查询等各类高级查询功能，并支持虚拟字段和用户自定义查询操作（比如：用户可以对部门和地理位置定义“属于”查询操作等等）。与此同时仍尽可能地使用底层数据库产品提供的索引等高效查询能力来提高效率。
* 高可移植性，libapidbc 提供常用数据产品的 DBC 插件，用户也可容易地自行扩展符合 DBC 接口规范的新插件。由于 DBC 对外暴露的配置、查询、更新、插入、以及事务等接口均与具体产品无关，因此简单地更换 DBC 插件即可方便地在不同数据库产品间进行切换。这极大地降低了应用对某个具体数据库产品的依赖性。
* 可以为一个 DBC 实例同时配置任意多个数据库服务器地址，DBC 插件可以此自动完成故障转移（高可用）、负载平衡等功能。

libapidbc 提供的另一个功能组件是 API Nexus。通过在 API Nexus 上注册 API Dispatcher，功能模块可以动态地将自己当前能够提供的服务，以 API 的形式暴露给其它模块。动态的 API 注册和注销机制匹配了插件的动态插拔能力。

为了解决插件间的初始化顺序依赖问题，API Nexus 还特别提供了异步（消息队列）语义的 API 调用能力——在发起一次异步调用时，若接收调用的模块尚未加载，则本次调用将被 API Nexus 临时存放在一个请求队列中，待该模块加载后，再按照调用顺序逐一处理这些请求。

除此之外，libapidbc 还提供了 ETL 映射辅助工具、自定义高级查询功能辅助工具、API 请求 locale 自动适配组件等多种辅助性功能组件。同时还包含一套集成了服务选举、服务发现、故障检测和分布式锁等分布式协调功能的消息路由服务（详见：5.4.3 消息端口交换服务）。

5.4.1 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 5）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。

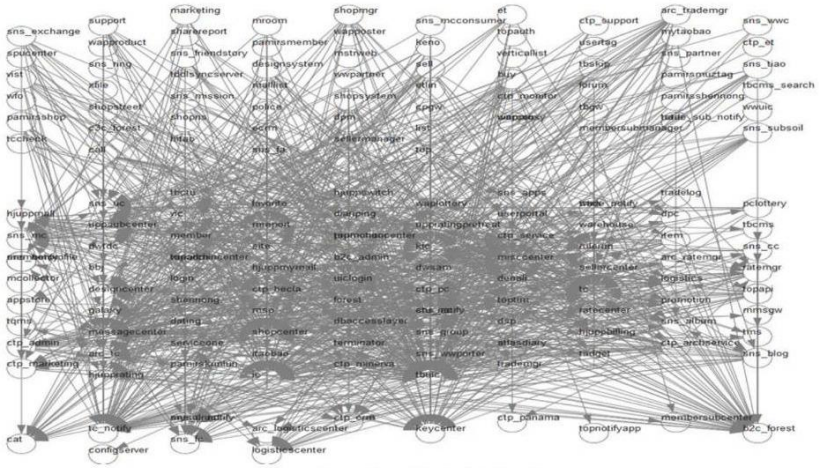


图 13 混乱的 SOA 依赖关系（图片来自互联网）

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、强一致性算法，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。

5.4.2 nSOA 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。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。

5.4.3 消息端口交换服务（BYPSS）

白杨消息端口交换服务（BYPSS，读作“bypass”）设计用于单点支撑万亿量级端口、百万量级节点规模，每秒处理千万至十亿量级消息的高可用、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：最低 5 毫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 nano-SOA 架构量身定做的集群协调和消息分发服务。nano-SOA 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。

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 次网络广播——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：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、2016 年 7 月 22 及 2019 年 12 月 5 日支付宝多次中断数小时；2013 年 7 月 22 日、2023 年 3 月 29 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（挖断光缆）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，nano-SOA 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：nano-SOA 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。

  当然，实际上不存在完全可靠的消息队列服务（能保证消息不丢失、不重复、不乱序）。因此在确实需要实现跨应用服务器节点的分布式事务时，建议通过 BYPSS 以及 BYDMQ 配合 SAGA 等模式来实现，详见：5.4.4 分布式消息队列服务（BYDMQ）。

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。

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，但难以支持多活 IDC。
一致性	强一致，主节点通过多数派选举。 读写操作均提供强一致保证。	写入强一致，多副本复制；大部分实现为提升性能，读取时牺牲了一致性（仅 Consul 支持配置成强一致读取模式）。
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 内存可支持至少 1 亿消息端口；每 1TB 内存可支持至少 100 亿消息端口；两级并发散列表结构确保容量可线性扩展至 PB 级。	通常最高支持数十万 KV 对。开启了变更通知时则更少。
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。在现代处理器和双口 4 万兆网卡上可达 3 亿次操作/秒的处理能力。	算法本身的特性决定了无法支持批量操作，相同测试条件下不到 200 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。

操作	对服务协调来说	对分布式锁来说
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。

HAC Manager 工具

BYPSS 中还包括了一个名为 HAC Manager 的配套工具，此工具利用 BYPSS 来完成服务选举和故障检测等工作。对于竞选相同服务的多个 HAC Manager 实例来说，竞选成功的节点可以执行用户配置的指令来启动该节点上的服务（Service/Daemon），相应地，在失去所有权时则停止该服务。

配合 DRBD、HAST、DataKeeper、DFS、Ceph、GlusterFS、Lustre 等分布式存储技术或 SAN 等共享型存储方案，HAC Manager 可以方便地将一个单机服务（如：传统 SQL 数据库、全文搜索引擎、报表生成服务等）转化为抗脑裂（Split Brain）的高可用集群（HAC）。

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。

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：每 10GB 内存至少 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 可轻松实现每秒数千万次请求的处理性能和万亿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 等有状态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“项目 2”中的描述）。 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。 3. 仅由属主节点缓存对应数据也避免了分布式缓存与 DB 之间的不一致问题，提供了强一致保证。 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	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，要对其实施分库分表、读写分离等额外优化。 分布式缓存与 DB 之间存在无法避免的数据不一致问题（除非在分布式缓存与 DB 间使用 Paxos 等协议来保证一致性，但随之而来的高昂性能损失也将使分布式缓存失去意义——这比不用分布式缓存还慢）。

项目	BYPSS HPC 集群	传统无状态集群
	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、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。	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。
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）。	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

项目	BYPSS HPC 集群	传统无状态集群
	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 BYDMQ 等专用的高性能消息中间件来完成。 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，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。	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。
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）。	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

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 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。	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	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。

项目	BYPSS HPC 集群	传统无状态集群
	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。	
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）。 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。	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。

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。

通过上述两个案例可以看出，相对于传统的 Redis + MySQL 等现有分布式缓存 + DB 的集群架构，BYPSS 集群还存在如下额外优势：

1. BYPSS 集群**保证强一致性**：正如前文所述，BYPSS 提供强一致、多活 IDC 高可用的高可用和高性能集群计算（HAC+HPC）能力。而 Redi 等分布式缓存，即使加入了 Revision 字段并利用其实现原子（CAS）操作，仍然无法保证其与 DB 之间的强一致性，只能通过加入失效超时（TTL）等机制来缓解数据不一致带来的危害（但此举同时也引入了缓存击穿、缓存雪崩等其它问题）。

2. BYPSS 集群可提供**数万倍量级的网络和 CPU 性能提升**：BYPSS 集群以纳秒级的内存直接访问替代了毫秒级的 Redis 网络查询，性能提升几个数量级。同时避免了每次缓存访问时频繁的数据序列化和反序列化过程，显著降低了 CPU 和网络开销。

3. BYPSS 集群可**完全避免缓存击穿**（失效）问题：由于 BYPSS 集群中的每个对象均仅缓存在其属主节点上，仅该对象的唯一属主节点有权向 DB 发送访问请求，因此不可能发生缓存击穿（不会产生针对某个热点对象的并发 DB 查询）。

4. BYPSS 集群可**轻易避免缓存穿透**（无效）问题：由于 BYPSS 集群中的每个对象均有其唯一属主节点，因此很容易在属主节点上对无效（不存在等）的对象 ID 进行标记和过滤（例如：高性能 ID 并发散列集合、布隆过滤器等）。因此可以轻易避免缓存穿透问题。

5. BYPSS 集群可**完全避免缓存雪崩**（大面积失效）问题：首先，由于 BYPSS 集群中缓存与 DB 数据间是强一致的，因此无需设置一个缓存过期时间来缓和缓存数据不一致的危害。因此缓存雪崩的一大前提：大量缓存同时到期就不复存在了。其次，由于消除了专门的缓存集群，所有缓存数据均内嵌在 App 服务器内，自然也就不存在“缓存节点宕机”这种问题了。与此同时，BYPSS 提供的强一致多活 IDC 高可用集群能力可保证 App Server 集群不会出现大规模宕机事故（不可抗力除外，真发生不可抗事件导致集群整体下线的话，那跟缓存雪崩已不是一个级别的问题了）。

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、电信、物联网、大数据批处理、大数据流式计算等广泛领域。

我们以后还会陆续增加更多实用案例，以供大家参考。

5.4.4 分布式消息队列服务（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。

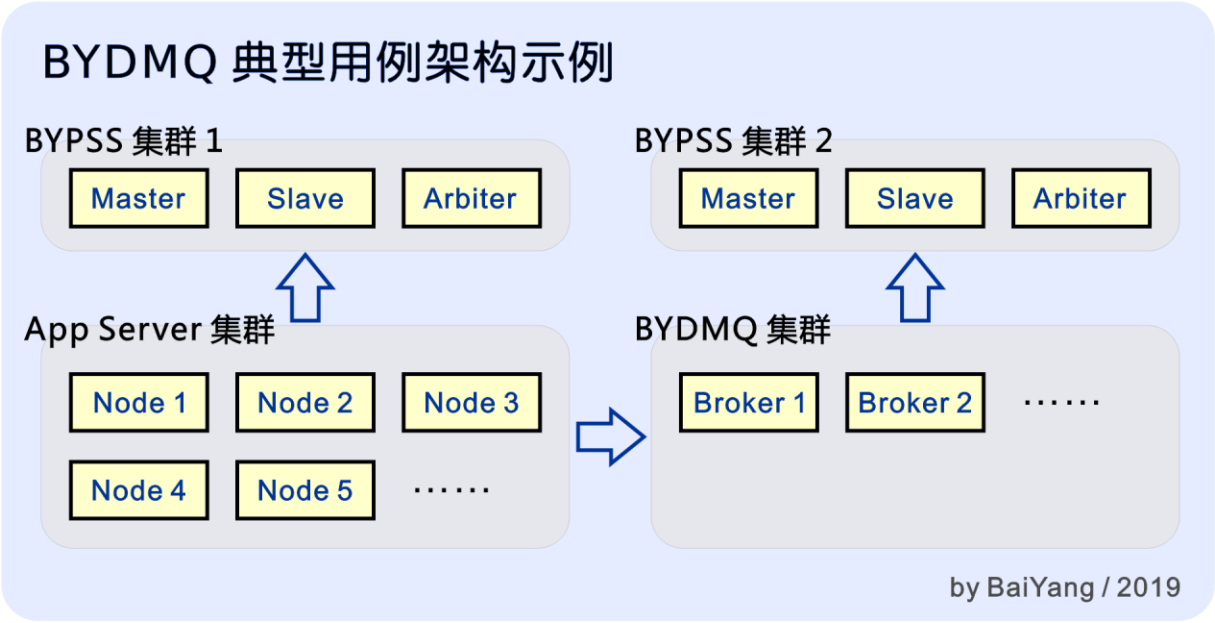


图 14

如图 14 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 15

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。

5.5 分布式全文搜索（FTS）服务

分布式全文搜索服务（BYDFTS）基于著名的 Sphinx 全文搜索引擎以及 BYPSS 分布式协调服务而实现。为全文搜索（FTS）和标签（Tag）匹配服务提供了一套完整的强一致、高可用（HAC）、高性能（HPC）的分布式集群解决方案。其主要实现方式可参考 5.4.3 消息端口交换服务相关说明，此处不再赘述。

除此之外，BYDFTS 还以 UDF 插件等形式实现了以下常用功能扩展：

* 分布式的跨表（跨索引）关联（JOIN）查询：为 Sphinx 添加了高效的分布式跨索引 JOIN 支持，弥补了目前市面上包括 Solr / Elasticsearch（Lucene）、Xapian、Sphinx 在内的所有主流全文搜索引擎均无法支持分布式跨表 JOIN 查询的缺陷。

  本扩展支持高效 LRU 本地缓存，并可通过标准的 DBC 插件（详见：5.4 nSOA 基础库－libapidbc）接入后端数据存储（如：SQLite、MySQL、MongoDB、Cassandra 等）。

  本扩展支持以下三种分布式数据处理方案：

  - 共享实例：所有分布式 BYDFTS 节点均共享相同的后端存储服务（如：MySQL、MS SQL Server、PostgreSQL、MongoDB、Cassandra 等）。此方案架构简单明了，同时后端存储服务亦可通过 NewSQL、NoSQL 集群的方式实现高可用以及横向扩展。缺点是需要额外部署后端存储服务集群，增加了实施和运维的负担。

  - 镜像复制：所有分布式 BYPDFTS 节点互相镜像相同的数据，通常与 SQLite 等本地后端存储搭配使用。依照“大表分片，小表复制”的原则，此方案适合用于关联查询的辅表尺寸较小时。

  - 数据分片：将数据按照一致性散列算法进行分片，数据保存和读取等操作均依此散列值被映射到与之对应的属主节点上完成。此方案支持复制组，可将每个数据分片同时复制到一组节点当中，以提供数据高可靠和服务高可用保障。

* 对 Tag 集合（MVA 属性）的 Favor Rank 算法：可以对两组 Tags 进行比对，并按照指定规则返回其匹配度。例如：可将结果集中每个文档的标签数据与当前用户喜好进行复杂的加权匹配，并将其匹配度计入相关性算法。

* 对时间日期字段的 Timeness Rank 算法：可按照指定规则为时间戳等字段打分，返回其时效性系数。时效系数同样可作为相关性算法中的一个因素，影响搜索结果的排序规则。

5.6 安全隧道服务（BYST）

白杨安全隧道服务（BYST，读作“best”）为用户提供了一种端到端的安全隧道服务，主要支持以下特性：

1. 支持 PSK 和 PKI 鉴权：可使用预分配密钥和公钥体系架构进行鉴权。

2. 支持数十种强加密算法：支持数十种块加密和流式加密算法，具体列表可参考 4.1 密码编码学算法模块－algorithm。

3. 可与我方自主研发的 EAL5+ 级别智能卡硬件无缝结合，提供极高安全等级的防护能力。

4. 支持实时数据压缩：基于 lz4 算法的高性能实时数据压缩，可通过配置选项开启。

5. 支持消息完整性校验：使用高性能散列算法计算校验和，以确保消息的可靠性和完整性（校验和与压缩后的数据均会被加密后再传输）。此功能可通过配置选项开启。

6. 支持防回放攻击：可通过配置项启用并设置防回放攻击的时间窗口范围。

7. 数据混淆：不同于 OpenVPN、SSL、SSH、IPSec 等现有方案，本隧道协议无任何可被观测的特征。不知道密钥的第三方拦截者只能观察到随机二进制字节流，难以通过任何有效手段侦测出通信双方正在使用本隧道协议。

   不但自己无特征，BYST 还可以扮演 http 等合法白名单协议（BYST over http），即便是在某些仅允许白名单协议的极端环境下，也可正常工作。

   与此同时，BYST 还支持 HTTP chunked transfer encoding，可以做到每个 package（几十 KB~几 MB）仅增加 3.5 个 Bytes 左右的开销，数据膨胀率低于 0.01%，避免了笨重的 HTTP 包头导致数据膨胀的问题。远远低于 SSL/TLS、SSH、h2、WebSocket 等其它白名单协议所带来的额外通信开销，在实现白名单通信的同时仍维持了极高的网络利用率。

8. 高效率：得益于基于汇编优化、零内存拷贝以及 DMA＋ 硬件中断的纯异步 IO 通信组件（详见：3.2.1 高效 IO 框架），即使在受限的硬件平台上，BYST 也可以满足高性能和高并发业务的严苛需求。

   此外，得益于我方成熟的 IO 批量打包、专利的分布式 N:M 动态连接池加速，以及高性能数据实时压缩等算法，BYST 显著提升了网络利用率（高载荷比）和网络吞吐量。

与此同时，相比于各类现有方案，已将握手（协商）、挥手、确认等环节进行了精心化简的 BYST 隧道协议亦明显具备更低的通信延迟。

9. 高性能和高可用集群：支持基于 BYPSS（详见：5.4.3 消息端口交换服务（BYPSS））的高性能和高可用多活 IDC 集群部署。

   此外还支持多路径自动验活和路由自动切换等功能。可通过增加备用链路来显著提升服务整体可用性，同时也支持多个链路之间的自动流量聚合及负载平衡等功能。

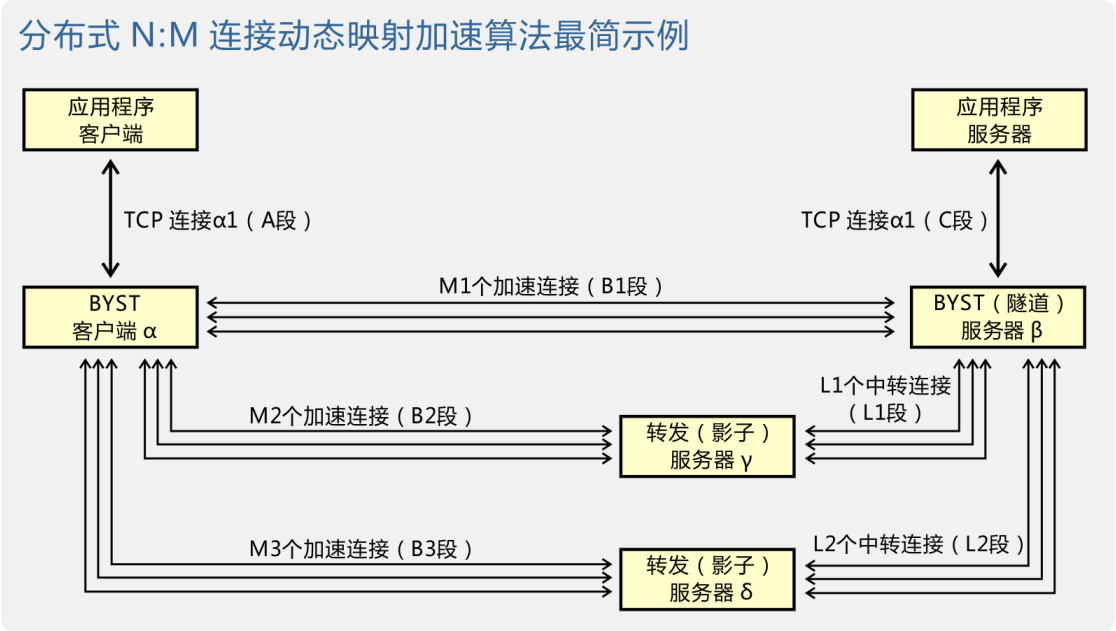
10. 线路级实时补偿：独有的线路级实时跟踪和预测智能算法，对各线路的丢包和抖动分别进行实时跟踪和补偿。

BYST 服务主要用于在 Internet、卫星、微波、SDH（MSTP）、以及跨区域光纤专线等广域或城域网环境中建立安全可靠的数据传输通道，通过强加密和强校验算法来保证数据的安全性、可靠性和完整性，并通过数据压缩降低带宽成本。同时还可通过难以被分析的混淆算法来保护其隧道通信不会被识别、拦截和屏蔽。

综上所述，BYST 主要带来如下技术优势：

1. 保障通信安全：提供带有强加密、强校验以及防回放攻击机制的安全通信隧道。

2. 打满带宽：得益于独有的 IO 自动批量打包、实时数据压缩，以及**我方专利的分布式 N:M:N 动态连接映射加速算法**，BYST 可做到打满用户带宽上限，显著提升 Site-to-Site Tunnel 通信性能。大量用户实测表明，在同城通信（MAN）环境中，BYST 仅启用单点 N:M 加速即可提升超过 6 倍的带宽吞吐；而在异地通信（WAN）环境中，BYST 的单点 N:M 加速则可实现高达 70 倍的吞吐性能提升。而分布式 N:M 加速则可在此基础上随着分布式加速节点的增加而线性提升聚合吞吐率。



3. 防止误杀：正如前文所述，为提高通信性能以及降低握手延迟，BYST 本身被设计为完全无特征的通信协议。此外，通过数据强加密、IO 自动批量打包、实时数据压缩、以及我方专利的分布式 N:M:N 动态连接映射加速算法等数据混淆机制，BYST 所承载的所有上层协议也将失去其可识别的特征。而 BYST over HTTP 的白名单协议通信支持则进一步保证了其优秀的防火墙通过性。

4. 代理鉴权：BYST 可通过 PSK、PKI 以及 EAL5+ 等级的安全智能卡硬件等技术提供安全可靠的双端鉴权访问，从而免除了上层应用间相互对接时还需要自行实现 CHAP、IKE、LDAP 等复杂算法来完成鉴权的麻烦。

6. 界面、媒体及其它工具

包含跨平台音频 IO 框架（libaudioio），跨平台的国际化 GUI 组件框架（libmlgui）等等。

6.1 跨平台音频 IO 库 – libaudioio

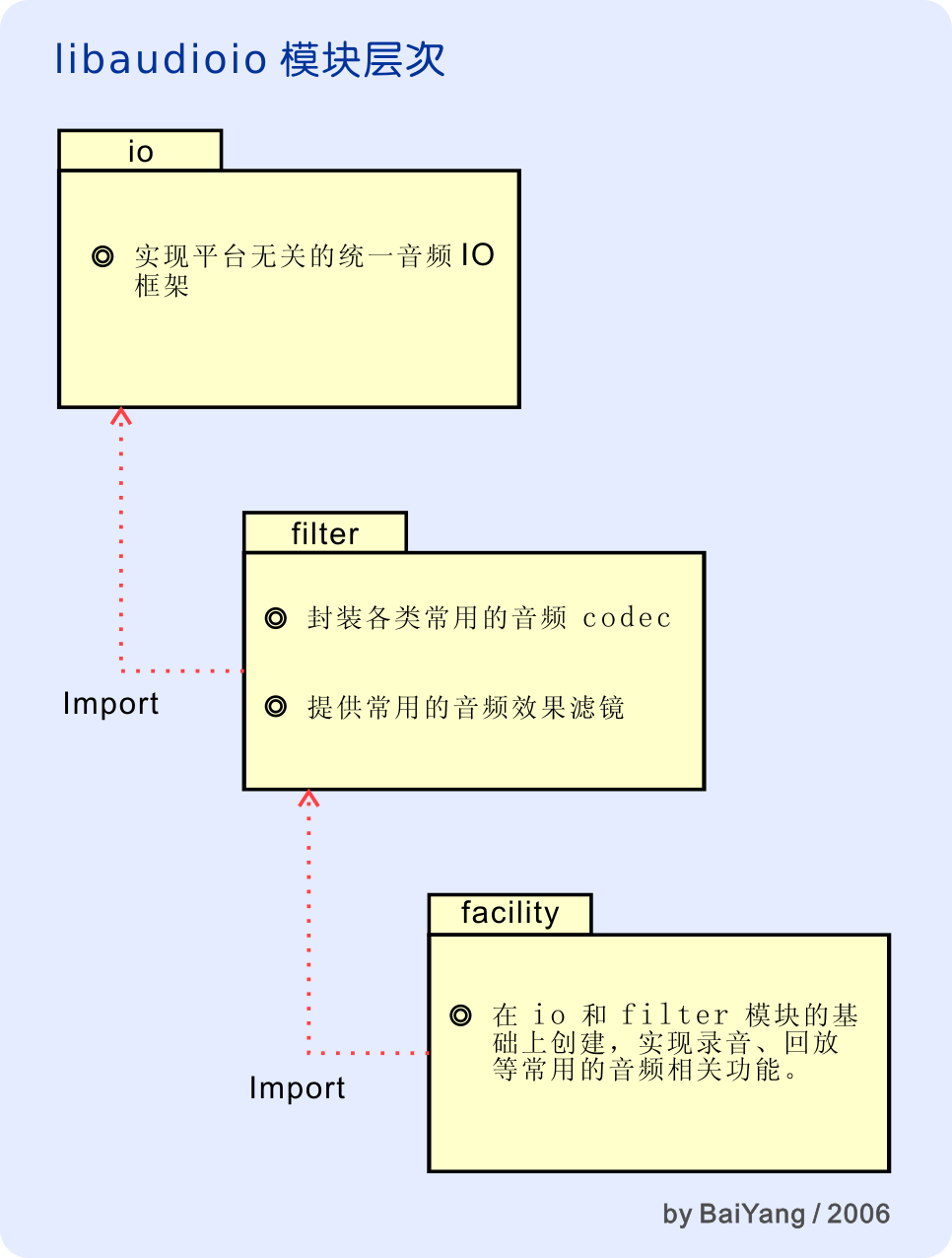


图 16

如图 16 所示，libaudioio 库采用与 libutilitis 以及 libcrypto 相似的模块层次结构实现。位于最

底层的 IO 模块负责封装所有与具体操作系统相关的音频 IO 接口操作，向上提供一个统一的音频 IO 操作界面。

IO 模块目前支持以下音频接口：

- Audio Session API (Win3264)
- WDM Kernel Stream (Win32)
- DirectSound (Win3264)
- Multi Media Extension (Win3264)
- Core Audio (Macintosh)
- Sound Manager (传统 Macintosh)
- Advanced Linux Sound Architecture (Linux)
- Audio Science HPI (Linux)
- Audio Library (un*x)
- Open Sound System (un*x)
- ASIO (平台无关)
- JACK (平台无关)

向上提供一致的，基于 Command 模式的异步通信界面。

filter 模块建立在 IO 模块之上，遵循 libutilitis 中定义的通用数据滤镜框架，支持在数据源（Data Source）和数据目的（Data Sink）之间使用各种数据滤镜自由地组成任何滤镜链。filter 模块又可分为两个层次：codec 子层和效果器子层。

在 Codec 子层定义的所有对象中，仅包含数据源（解码）和目的（编码）。涵盖对 wav（包括 x-law 及 g.7xx、gsm 等子格式）、caf、au、snd、voc、mpx（mp1、mp2 和 mp3）、mpc（Musepack）、flac、ogg vorbis 等多种常用格式的编解码支持。其中，考虑到版权问题，对 mpx 和 mpc 仅提供解码支持。

这里要特别强调 au 格式的重要性。作为传统 unix 音频格式，au 不但拥有广泛的用户群，更是唯一一个支持将采样长度信息标记在文件尾端（只要文件头中的相应字段被标记为-1）的非压缩音频记录格式。这个特性在生成临时文件和流式 IO 等场合中非常重要。

在 filter 模块的效果器子层内，我们定义了各种音频效果滤镜。由于目前大多数产品对音频处理的要求不高，这里仅实现了衰减器（音量控制）和采样转换器等基本滤镜。

与 libutilitis 及 libcrypto 一样，处在 libaudioio 库最高层的模块也名为 facility。其中封装音频处理相关通用高层功能。鉴于目前大多数软件产品对音频处理的要求都不高，这里仅定义了一套

通用音频录制和音频播放工具。

6.2 跨平台国际化 GUI 组件框架 – libmlgui

与经典操作系统功能和各类算法不同，GUI 在各个平台上的实现从根本（机制和模式）上就是千差万别的，其具体细节十分复杂。对于一般的软件开发组织来说，自己从头开始构建一套完整、健壮的跨平台 GUI 框架工作量十分巨大。

libmlgui 库借助迄今已有超过 24 年历史的著名开源 GUI 框架：wxWidgets 为基础。结合 libutilitis 等组件，实现了一个带有国际化支持和各种附加功能的通用 GUI 框架。

libmlgui 分以下四个层次结构实现：

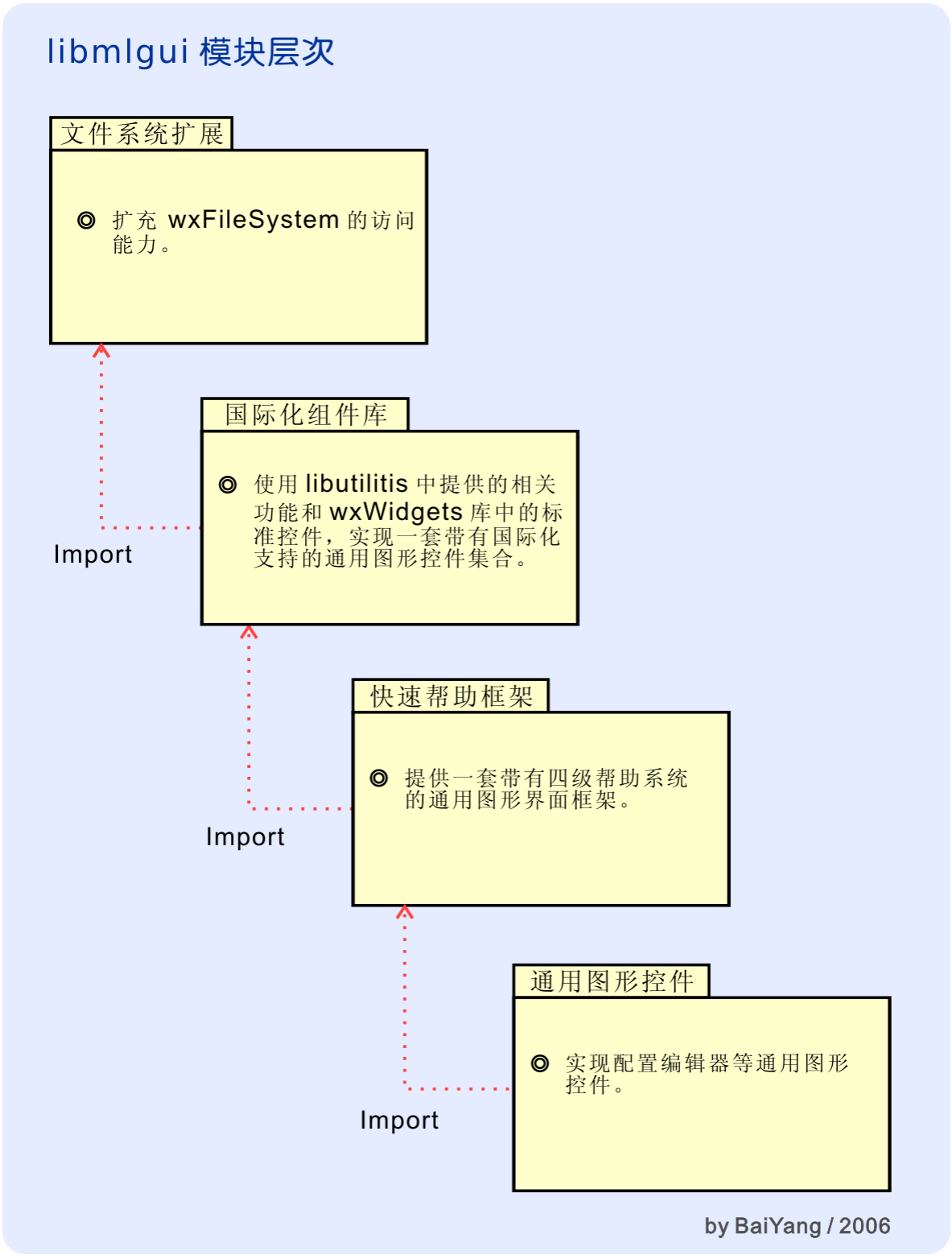

libmlgui 模块层次
文件系统扩展
◎ 扩充 wxFileSystem 的访问能力。
国际化组件库
◎ 使用 libutilitis 中提供的相关功能和 wxWidgets 库中的标准控件，实现一套带有国际化支持的通用图形控件集合。
Import
快速帮助框架
◎ 提供一套带有四级帮助系统的通用图形界面框架。
Import
通用图形控件
◎ 实现配置编辑器等通用图形控件。
Import
by BaiYang / 2006

图 17

6.2.1 文件系统扩展

在此块中，我们将 libutilitis 内定义的虚拟文件系统等机制扩展为 wxWidgets wxFileSystem 框架内的相应驱动层。具体包括：

- 实现内存文件系统：使用 libutilitis 中的虚拟注册表（CConfig）组件，实现支持目录层次结构和零内存拷贝访问，同时满足 wxFileSystem 界面的虚拟内存文件系统。
- 实现基于 VFS 的文件系统：使用 libutilitis 中定义的 VFS 框架，使 wxFileSystem 支持将某一个或几个文件虚拟为一个只读文件系统的功能。同时，由于 libcrypto 中支持加密和压缩功能的 VFS 也基于 libutilitis 中的 VFS 框架实现，故这一扩展等于同时为 wxFileSystem 加入了支持此类 VFS 的功能。

  值得指出的是：由于 VFS 系统分为“文件型（如前文所述，可以将某种文件虚拟为一个文件系统的类型）”和“封装型（封装 FTP 等真实存在的文件目录系统为满足 VFS 操作界面的对象）”两种。故这一扩展也同时为 wxFileSystem 接口加入了基于 FTP、HTTP 等封装式 VFS 的访问能力。

6.2.2 国际化组件库

结合在 libutilitis 中定义的快速语言包（使用哈希索引表）和多语言对象类，以及 wxWidgets 中的按钮、菜单、窗体等各种 GUI 控件，可以方便地实现一个带多语言支持的图形控件库。

下面我们以最常用的 GUI 控件：按钮类为例，看一看前文所述的多语言 GUI 控件到底如何实现：

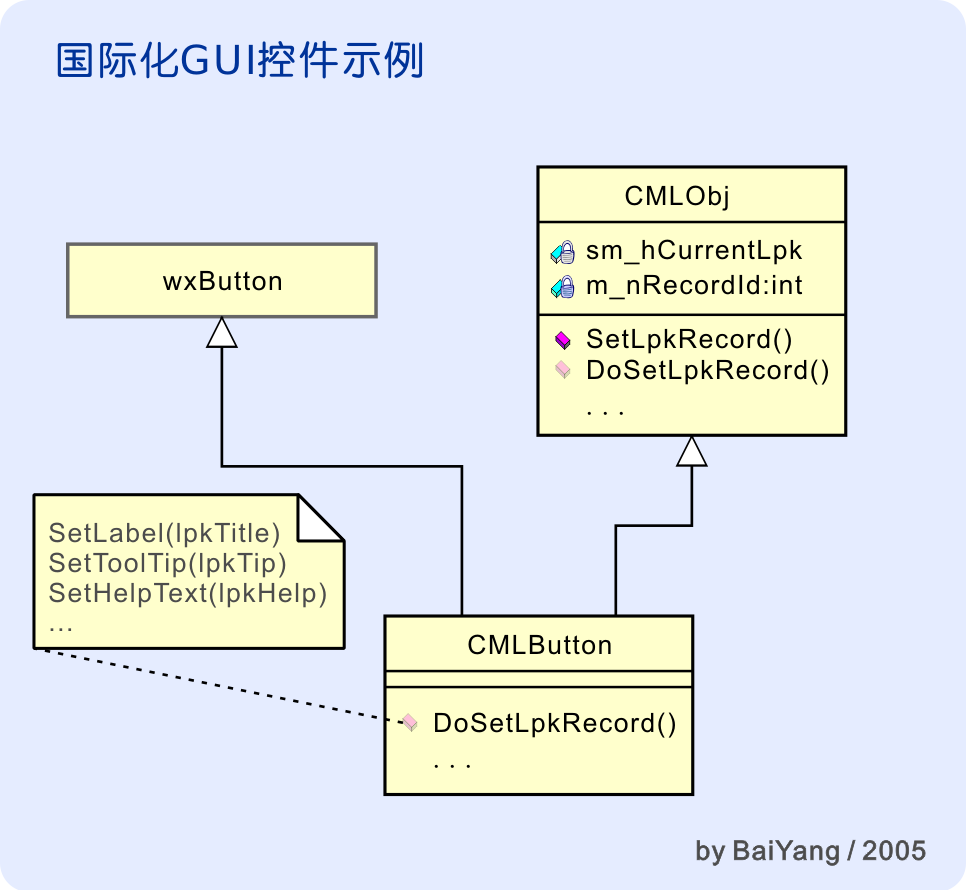


图 18

图 18 展示了 liblmgui 中的多语言控件看起来大概是什么样子。在 libutilitis 中，随同语言包一起定义了所有多语言 UI 控件的公共基类：CMLObj。这个类以静态成员的方式保留了当前语言包的一个句柄（这里的“句柄”是一种支持引用计数和自定义销毁策略的智能指针），另一个非静态整形成员则保存着当前对象使用的语言资源 ID。

一个支持多语言的图形按钮控件同时从 wxWidgets 库的标准按钮类（wxButton）和 CMLObj 类派生而来。通过正确地重载其中的 DoSetLpkRecord 虚方法，我们就可以在每个图形对象初始化时设置好恰当的语言、字体等信息。

此外，这里还结合使用了 libutilitis 提供的字符集编码转换功能以及 wxWidgets 提供的字体映射机制尽最大可能向用户正确地呈现这些语言资源。

需要指出的是：为了便于描述，在这里我们故意省略了一些必要功能。例如：支持语言包实时更换所必须的 Refresh 机制等。此外，对于列表、组合框等比较复杂的控件，语言包也会通过相应（如：允许一条记录同时对应多个值）的机制予以解决。

同时，此模块也支持从前文所述的各种 wxFileSystem 中加载语言包。

6.2.3 快速帮助框架

软件设计师们发现，用户对于应用程序的要求似乎总是没有止境，在提供丰富功能的同时，用户还希望提高软件的易用性。为了回应这一挑战，除了努力设计出高度一致、符合逻辑，且易于理解的用户界面以外，有必要实现一套通用的快速帮助应用框架。

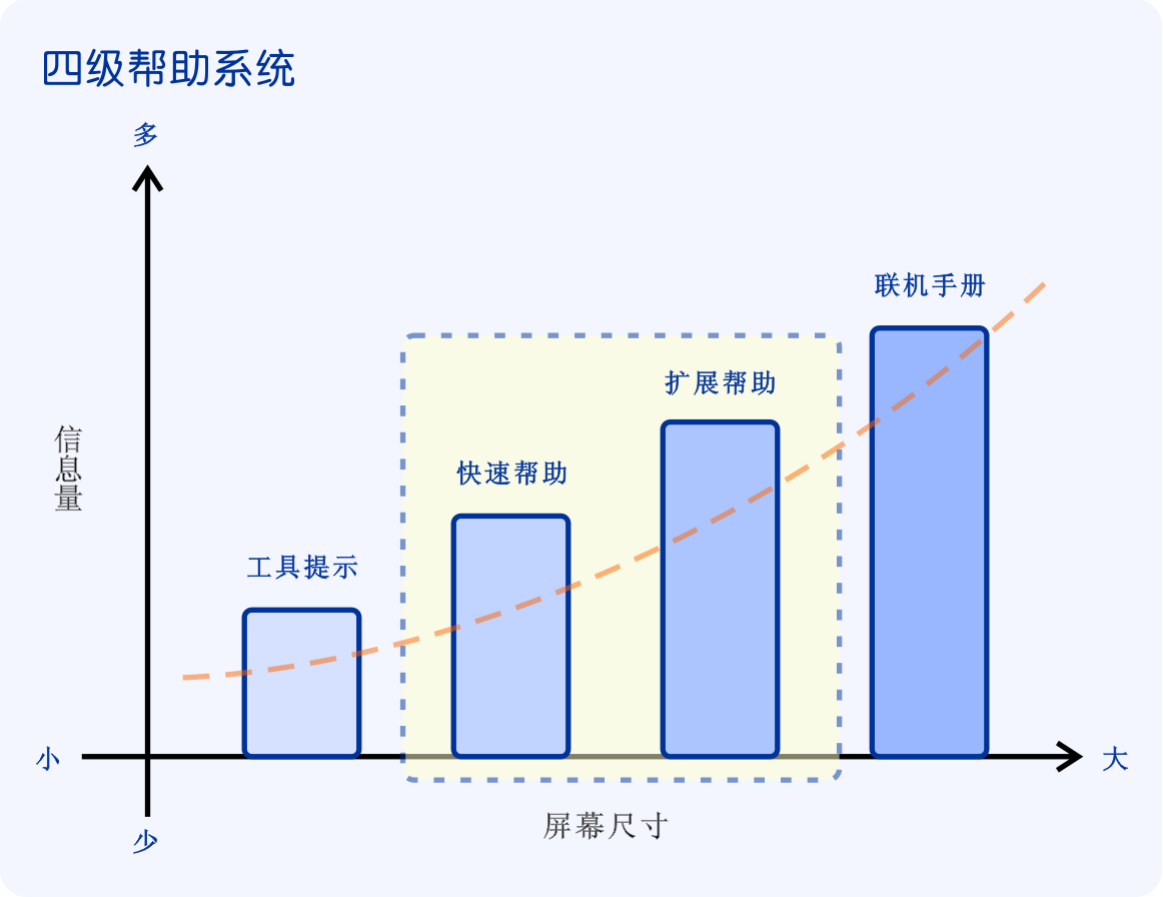


图 19

如图 19 所示，在一个启用了快速帮助系统的应用中，帮助信息由传统的两类扩展为四类。在这种四级帮助系统中，随着级别的升高，帮助内容会变得越来越全面和详尽。但与此同时，随着级别的每一次提升，帮助系统所占用的屏幕空间和牵扯的用户注意力也会随之增加。

在传统的两级帮助系统中，"工具提示"有占用屏幕空间小、牵扯用户注意力少等优点，但是它提供的信息量通常也很少，并且无法稳定地停留在屏幕中供用户仔细阅读。而"联机帮助"则恰恰相反：其内容详尽、组织结构清晰，但占用屏幕空间大、牵扯用户精力多，这一缺点直接造成大部分用户根本没有兴趣阅读软件产品的联机帮助。

四级帮助系统通过增加"快速帮助"及"扩展帮助"机制，为用户在以上两方面提供了良好的平衡点。

快速帮助机制

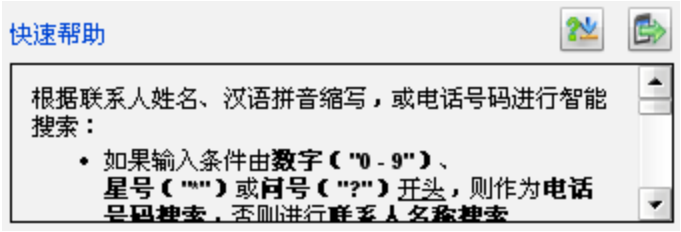


图 20

用户只需轻点（或使用键盘上的 TAB 键将焦点切换至）感兴趣的位置，与之相关的帮助信息就会实时显示在当前程序中，某个与图 20 相似的快速帮助区域内。快速帮助区通常位于窗体的下端或两侧。在用户已能进行熟练操作或需要更多屏幕空间时，也可以使用上例中的“  ”按钮将其隐藏。

为了加强表达能力，快速帮助区能够支持标准 HTML 格式文本，并可显示 ico、bmp、png、gif、jpg、pcx、xpm 等多种常用格式的位图。此外，还具备允许软件设计师自定义点击行为的超链接功能（例如：点击联接后打开特定的联机帮助页，或加载某个应用程序等等）。

为了保持可移植性以及节约 CPU、内存等用户资源的使用，快速帮助中的 HTML 窗体仅使用 wxWidgets 的相关控件和 libutilitis 中提供的功能实现，不使用 Internet Explorer 等第三方 WebView 组件。

扩展的快速帮助机制

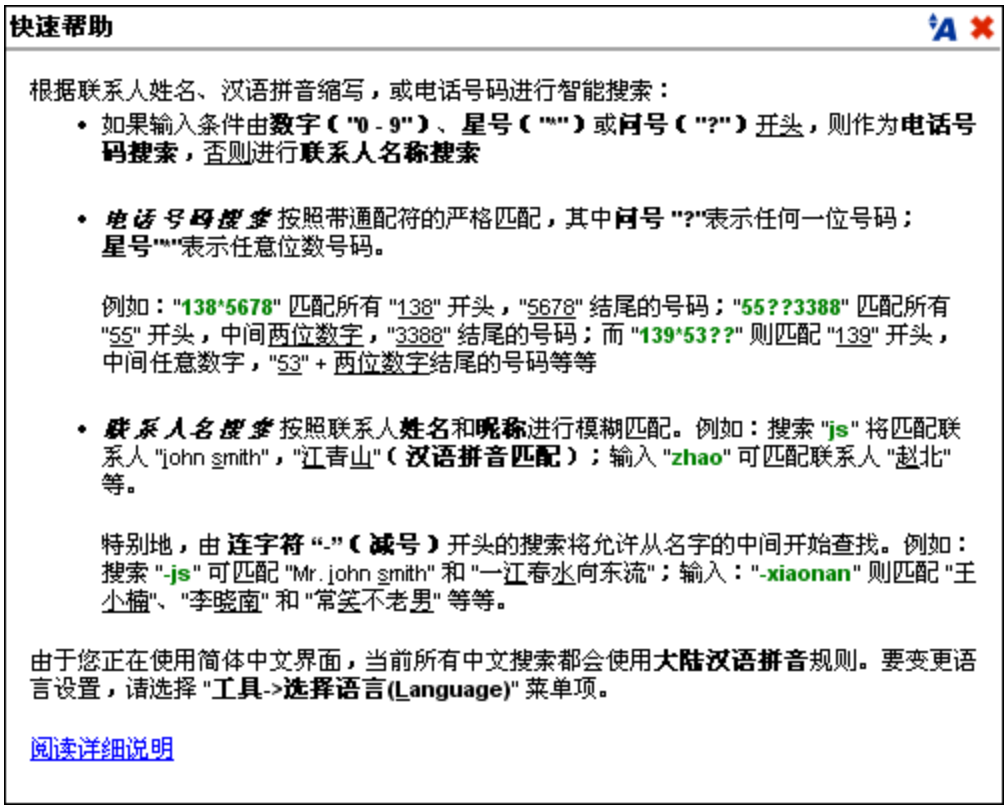


图 21

大量实践证明，快速帮助机制可以在大部分情况下满足用户需求，但有时用户可能需要更加舒适的阅读环境。例如：更大的阅读区域，可调整字体的大小。经过扩展的快速帮助正是为解决这样的用例而设计。在上例中，用户只需点击"  "按钮，即可在一个更为舒适的弹出式窗口中继续他的阅读（如图 21）。

快速帮助框架能够自动从控件及语言包中获取并显示正确的帮助信息，同时也支持从 wxFileSystem 中直接加载指定的 HTML 页面和位图。

6.2.4 通用图形控件

这里定义了若干常用的国际化图形功能控件，包括配置编辑工具和遇忙等待对话框等等。这里重点介绍一下其中的 CConfig 通用配置编辑工具：

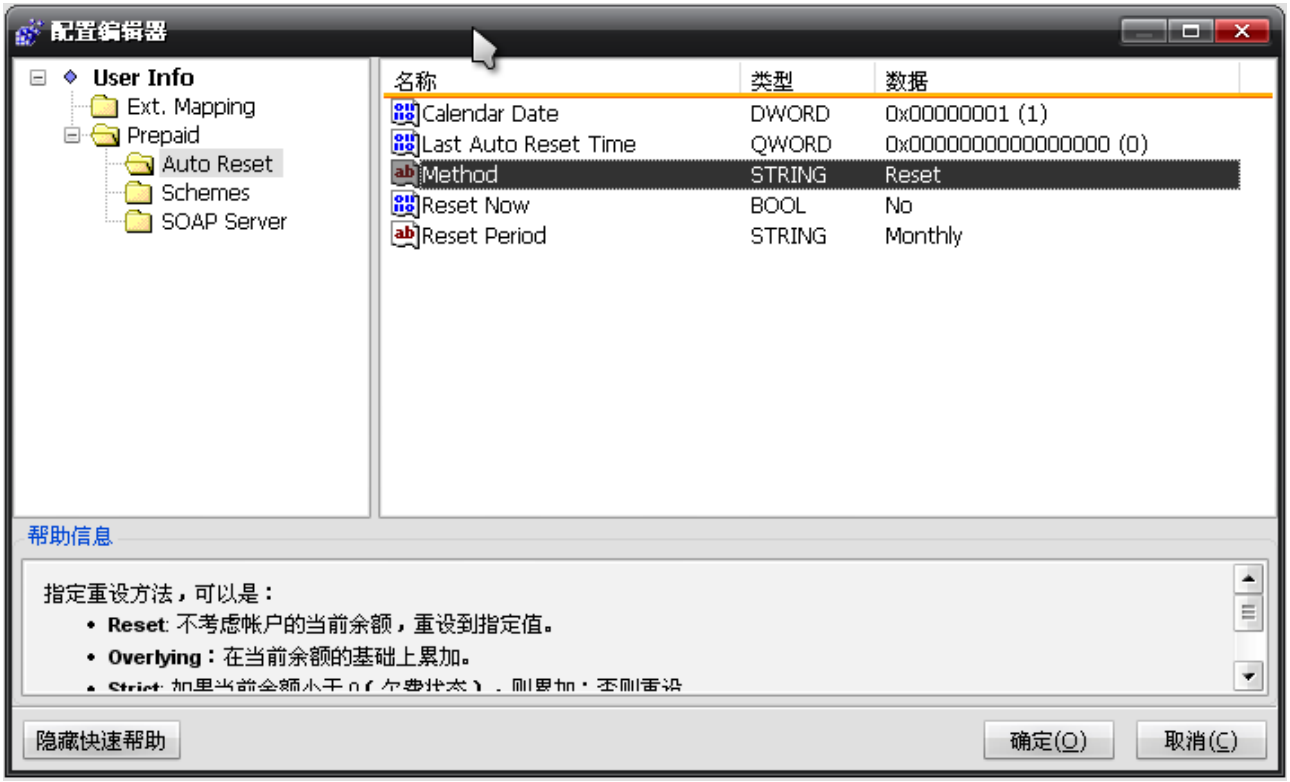


图 22

从图 22 中可以看出，该控件带有自己的快速帮助栏，这就可以方便地为用户提供当前选项或子键的用法说明等实时帮助信息。


编辑值
编辑
名称:
RF Rule
类型:
STRING_SET
内容(V):
[5-8]\d{6,7}
^\s+.*C
^\d*798
帮助信息
浏览和编辑选定值的内容
对于 DWORD 和 QWORD 类型：1-9 开头为十进制；x, 0x 开头为十六进 制；0开头后跟 0-7 为八进制；b, 0b开头为二进制
对于 BOOL 类型：以 T(t) 或 Y(y) 开头的内容表示'真'，其余表示 '假'
隐藏快速帮助
确定(O)
取消(C)

图 23

图 24

图 23 和图 24 分别展示了配置编辑控件中提供的值编辑和搜索功能。

至于配置编辑控件中具体会出现什么样的子键目录结构、这些子键中具体包含了哪些值，以及与他们对应的快速帮助信息是什么。这些问题均取决于软件设计人员或用户将何种虚拟注册表传递给编辑器。

在描述 libutilitis 库时，我们曾经提到了其中有一种在非 Windows 平台模拟注册表服务的组件。实际上，由于这个工具的跨平台特性，以及它很容易就可以做到比 Windows 注册表更优越的访问性能，而且具备可以保存在几乎任何地方（比如：VFS、内存、网络介质，甚至直接嵌入到可执行模块中等等）的特性。所以从本质上讲，这里的“配置编辑控件”其实就是一个虚拟注册表（CConfig）编辑器。它可以用来浏览和编辑指定虚拟注册表对象中的任意树形目录结构和配置项，并实时显示与当前配置项对应的快速帮助信息。这种灵活的设计使得高级配置编辑控件能够较好地适应各种复杂应用。

当然，除了"高级配置编辑对话框"之外，通用控件模块中还应包含诸如：遇忙等待对话框、语言选择对话框等其它通用控件，这些控件的含义都相当简单直白，这里不再赘述。

6.3 CConfig Language Binding 组件

正如前文所述，CConfig 是 libutilitis 提供的一种用于存储复杂信息的树形数据结构，支持丰富的数据类型。具有跨平台、高效、国际化和交互格式灵活多样等特性。

由于 CConfig 已被广泛用于各产品的配置管理、API、WebAPI 和模块接口间产生传递等场合，因此为了方便用户和第三方进行二次开发，我们为 CConfig 编写了针对 C/C++、Java、C#、F#、VB.NET、PHP、以及 JavaScript 等各类主流语言和开发环境的 Language Binding。

CConfig Language Binding 组件以面向对象的形式提供了以下功能：

* 打开和保存：支持 JSON、CSV、XML、INI、以及二进制（ISXF）格式 CConfig 数据的打开和保存（JavaScript 版仅支持 CSV、INI 和 JSON 格式）。
* 值访问：支持对值的读、写、替换、新建和删除操作；支持类型识别和存在性检查。
* 子键访问：支持对子键的新建、删除、剥离、存在性检查操作；支持对子键的导入、导出、替换、插入等操作。
* 搜索和遍历：支持对子键和值的遍历操作；支持对子键和值进行通配符或正则过滤。
* 状态信息：当前对象是否为空；当前对象是否已被改变；可获取子键、值数量；相等性比较等。
* 多线程安全：CConfig 组件上的所有操作都是线程安全的，同时也支持显式的、带超时的独占访问锁（Lock/TryLock/Unlock）操作（JavaScript 版除外，因为不需要类似操作）。此外，JavaScript 以外的 CConfig 组件还使用了带次数限制的自旋锁机制来进一步提高访问效率。

关于 CConfig Language Binding 组件的详细信息，请参考文档《CConfig 组件参考手册》。

6.4 JavaScript 工具库 – libbaiy

实现 libbaiy 的初衷是为了方便 B/S 架构的应用中，浏览器端代码与基于应用支撑平台的服务器端代码间的通信和交互。libbaiy 分为功能库和界面库两部分。

6.4.1 功能库

libbaiy 功能库实现计算和数据编解码等功能性操作，包括：

* 密码编码学相关算法，如：SHA1 散列算法；HMAC 消息验证码算法；HEX 与 BLOB 间

的编解码算法等。

* 完全满足 RFC4180 规范的 CSV 格式数据解析和生成工具。支持自定义分隔符、换行符和引用符；支持引用字段内的引用符换码。
* CConfig 组件，用于方便、高效地访问 CConfig Schema，详见：6.3 CConfig Language Binding 组件。CConfig 组件对子键和值的访问效率均为 O(1) 或 O(log(N))，具体取决于底层 JavaScript 引擎如何实现关联容器。
* 语言包访问组件，可加载在 libutilitis 中定义并实现的语言资源包内容。除了可访问所有语言包记录外，还支持包括语言名、兼容的代码页和 ISO 规范名、字符集和字符集编码、原始编码等元信息的访问。语言包组件对以上所有字段的访问效率均为 O(1) 或 O(log(N))，具体取决于底层 JavaScript 引擎如何实现关联容器。
* 支持自定义排序规则（如：中文->汉语拼音；中文->台湾通用拼音；日文->罗马拼音等）的国际化字符串比较和排序算法；支持多列复合升降序排序的表格（二维数组）排序算法；以及支持严格匹配、通配符匹配和正则匹配的字符串规则编译器等国际化字符串处理工具。
* 键树容器组件，键树（Keyword Tree）作为一种常见的 key -> value 数据结构，主要用于实现高效前缀匹配。在 libbaiy 中实现的 JavaScript 版接口与 libutilitis 中的 C++ 版兼容，不过其功能仅为 C++ 版的一个最小子集。
* 消息分发组件，消息分发器（Message Dispatcher / API Nexus）可根据不同类别，将消息分发到对应的处理器中。若指定类别的处理器尚未被注册，则由分发器负责将消息暂存于一个临时消息队列中，并在对应的消息处理器注册后，按照 FIFO 顺序完成分发（用于解耦模块间依赖）。
* 任务队列组件，用于管理和顺序执行一系列既定任务。任务步骤可事先定义，也可在运行时灵活调整。在所有任务执行完毕后，或发生未捕获异常时，可触发 Finally 回调。任务队列组件主要用于解决 JavaScript 环境中，大量异步模式引发的回调依赖管理（回调地狱）等问题。
* 移动平台工具，为 iPad、iPhone、Android 等移动设备提供的辅助工具，包括移动平台运行时判定、将 Touch 事件模拟为鼠标事件的拖放操作适配器等。
* 其它杂类功能，如：浏览器无关的 XHR（XmlHttpRequest）对象工厂；JavaScript 动态加载（同步或异步）和跨站加载；CSS 样式表动态加载和卸载；以拉伸的方式为指定的 DOM 对象设置和变更背景图片；测试当前浏览器平台；测试当前浏览器 LongPolling 兼容性；跨浏览器鼠标事件 Hook；跨 iframe 消息传递；16 位和 32 位字节序交换；http header 解析工具等等。

6.4.2 UI 界面库

提供基于 ExtJS 的通用 UI 控件及相关框架。所有控件均支持国际化和主题功能。

界面库目前主要实现了以下功能：

* 快速帮助扩展框架，实现了浏览器端简化版的 6.2.3 快速帮助框架。
* 浏览器端简化版的通用配置编辑器，详见下文。
* 其它杂类功能，如：为 ExtJS 表格添加本地化排序能力的补丁；带有羽化阴影效果，支持延迟弹出和延迟隐藏功能的 load mask 控件等。

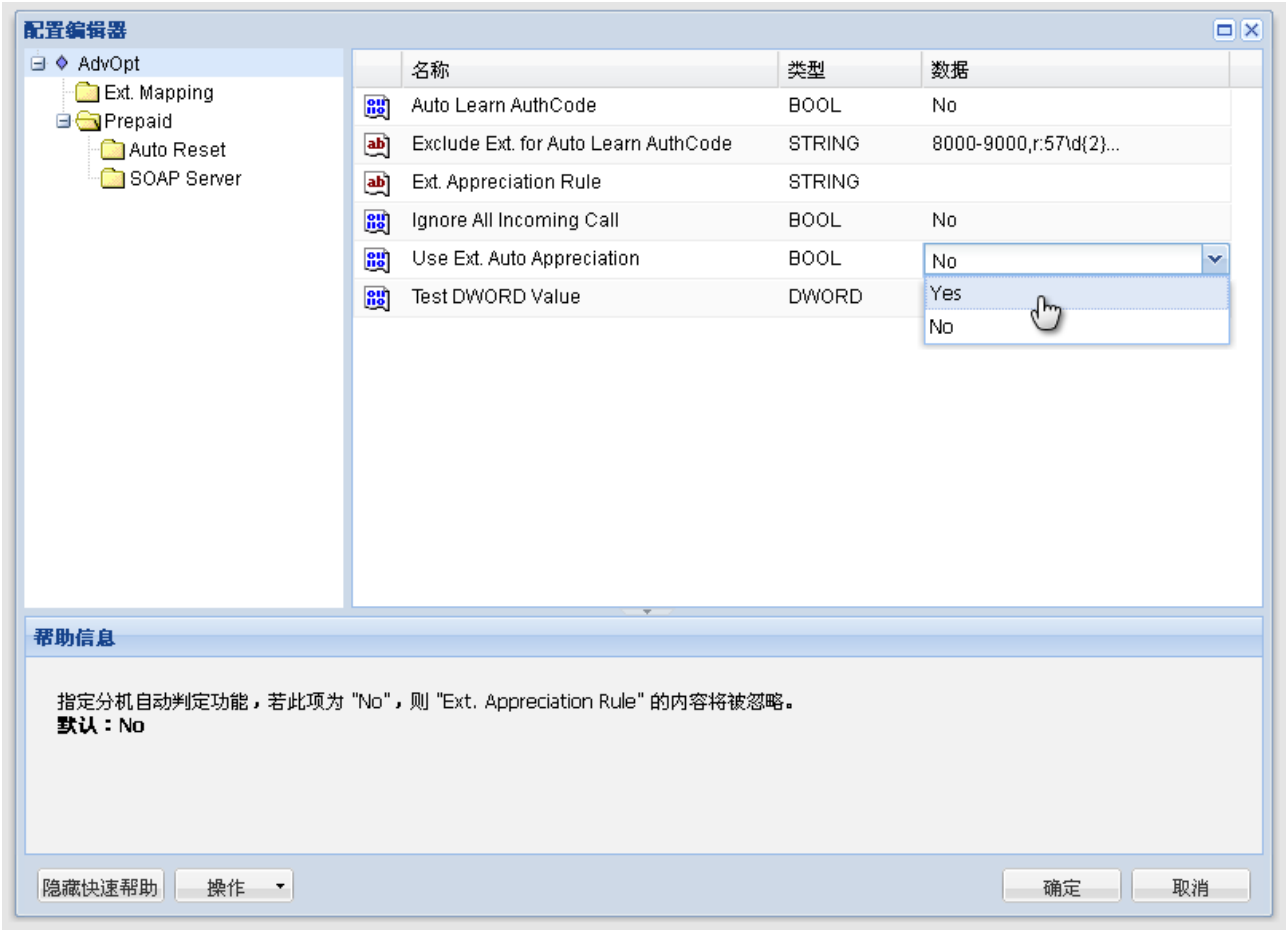


图 25

图 25 展示了 JavaScript 版的 CConfig 通用配置编辑器控件，其界面布局和功能均与 6.2.4 通用图形控件中的通用配置编辑器控件类似（两者甚至使用相同的语言包），这里不再赘述。

7. 应用支撑平台的错误处理机制

应用支撑平台工作在整个应用程序的最底层，这通常也是最容易产生错误的位置。另一方面，针对某个具体错误的处理却常常只能由位于高层的业务逻辑来决定。

C++ 中的异常机制非常适合处理这样的情况。首先：异常的使用免除了通过返回值和公共变量的方式逐条判断每个操作是否成功的麻烦，以及忘记此类判断所产生的痛苦和错误；也避免了经常要从错误发生点逐级返回到能够处理该错误的位置之尴尬局面。此外，异常机制也利于实现更有层次，也更为结构化的错误处理方式。

从效率方面考虑，C++ 的异常机制无疑也是非常适合用于错误处理的，其理由如下：

1. 异常机制仅在错误发生后启用。在正常情况下，异常机制几乎不影响程序执行的效率。此外，由于免除了为逐条操作判断返回值和/或检查公共错误变量，启用异常机制可能反而会增加程序的运行效率。

2. 即使在发生异常时，也仅有异常捕获和栈回退操作涉及与当前函数调用栈深度相关的 O(N) 算法。但是也要看到，传统的逐级返回至错误处理地点的机制，其算法复杂度也与之相当。

基于以上理由，我们为应用支撑平台定义了一套异常类层次结构：

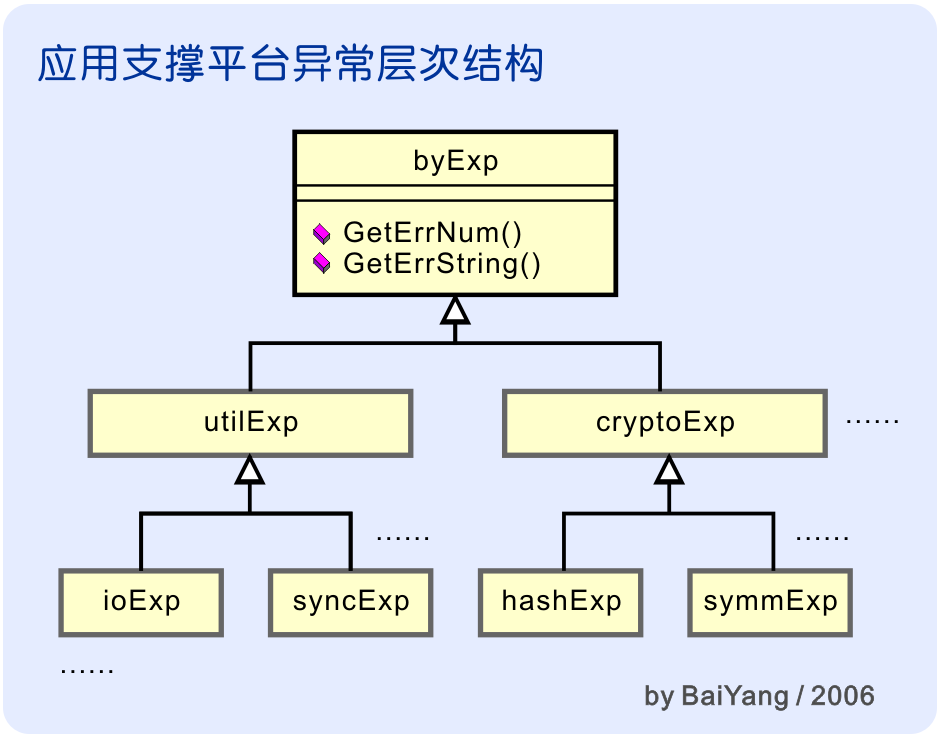


图 26

图 26 向我们展示了应用支撑平台异常类层次结构的一个局部特写。它虽不完整，但已足够用来说明应用支撑平台异常处理机制的设计思路。这是一个简单的四级结构——虽然图中只画出了其中三级，但可以想象，在"ioExp"下还会派生出"fileExp"、"socketExp"等类型。在 syncExp 下则有"mutexExp"和"semExp"等等。

在这里，每一个被抛出的异常对象都带有三种信息：首先是它在整个异常类层次结构中的位置（即：它的类型信息）；其次是一个错误码；然后是针对该错误的具体描述。实际上，在一些比较复杂的异常对象里，还会加上一个原因码（Reason Code）。这样，位于高层的异常处理器就可以在不同的粒度（例如：是处理所有密码编码学相关错误还是仅处理散列校验错误）上实现相应的错误处理策略。

关于异常使用方式的进一步讨论，请参考本人拙作：《C++ 编码规范与使用指导》中的“异常”一节，关于异常的编译器实现细节和性能分析，请参考其中“C++异常机制的实现方式和开销分析”一节。

至此，我们已对白杨应用支撑平台完成了一次快速浏览。有关进一步详细信息，请参考其中各组件的用户手册和开发文档。