

架构师 ARCHITECT



本期专题：Netty之道

Netty系列之Netty高性能之道

Netty系列之Netty可靠性分析

推荐文章 | Article

几种线程池的实现算法分析

HBase高性能复杂条件查询引擎

特别专栏 | Column

腾讯云刘颖：块存储深度剖析

构建大型云计算平台分布式技术的实践



卷首语：我和 Netty 的故事

还在上学的时候，我特别好奇 QQ 是怎么实现的，为什么我一发消息我的好友马上就能收到且基本没有延迟，它的 原理是什么？大三的时候，我学习了 Java 语言，接触到了 Socket 编程，不知天高地厚的我当时居然想自己实现个简易 QQ，和几个朋友做了技术评估后觉得这个事好像不是那么难，紧接着我们就用 Swing 以及一些 Socket 编程知识完成了一个简易的局域网 QQ，虽然功能不是那么的稳定，但基本上还是可以实现私聊和群聊功能。这也许是我第一次接触网络通信，核心功能部分使用了同步 I/O 的 Socket 类库。

参加工作后，公司使用的 RPC 框架是 Hessian，Hessian 是一款基于 HTTP 协议的 RPC 框架，采用的是二进制 RPC 协议，但是在 Java 中，Hessian 的服务端需要使用 Tomcat 之类的容器，而它们的性能总是那么的不如人意。因为那会公司使用的 MVC 框架是 Play，所以我很早就接触到了 Netty，它的高性能、高可靠的特性早有耳闻。看了 Play 框架中 Netty 部分的代码以及 Netty 的官方案例后，我用 Netty 重新实现了 Hessian 的服务端，于是一款构建于 Netty 和 Hessian 基础上的高性能的 RPC 框架诞生了，我取名叫 Hetty（Hessian+Netty）。简单的[性能测试](#)之后，我发现 Hetty 的性能是之前的 4~5 倍，这次之后，我对 Netty 有了更深入的了解，使用 Netty 可以更简单的开发出高性能、可扩展、易读易维护的系统。

再到后来，我去了一家游戏公司，发现他们在服务器端大量使用了 Netty 框架，从来没有想到 Netty 在游戏行业已经得到了这么大范围的使用。不过再仔细想想，这一点都不惊奇，游戏服务器端，处了大量的业务逻辑外，其它部分其实都在玩 NIO，而 Netty 作为一款成熟的异步 NIO 通信框架，它的性能、扩展性、稳定性、使用难度都得到了业界的肯定。那 Netty 有哪些优点了？我认为有以下几点：

1. 文档齐全，社区活跃，API 简单，案例很多。
2. 支持多种协议，如 HTTP、FTP、SMTP。
3. 性能高，易扩展。
4. 周期性的版本迭代，成熟且稳定。

这么一看，工作生活中处处与 Netty 为伴，其实是 Netty 见证了我的成长，希望好的框架能让更多的人收益！

本期主编：郭蕾

目录

卷首语：我和 Netty 的故事

云计算时代的运维分工与理念变化：腾讯资深运维 Coati 的观点

作者 杨赛

嵌入式 OS 的现状、智能的物联网与未来的机器人

作者 罗未

虚拟研讨会：在低延迟环境中使用 Java

作者 Charles Humble，译者 夏雪

专题：Netty 之道

Netty 系列之 Netty 高性能之道

作者 李林锋

Netty 系列之 Netty 可靠性分析

作者 李林锋

使用 SQL Server 2014 内存数据库时需要注意的地方

作者 王枫

几种线程池的实现算法分析

作者 刘飞

HBase 高性能复杂条件查询引擎

作者 耿立超

腾讯云刘颖：块存储深度剖析

受访者 刘颖 作者 刘宇

构建大型云计算平台分布式技术的实践

作者 章文嵩

不得不的 S3 基础知识

作者 包研

JVM Bug：多个线程持有一把锁

作者 李嘉鹏

为什么 CDN 对移动客户端加速“没有”效果

作者 刘宇

封面植物

云计算时代的运维分工与理念变化：腾讯资深运维 Coati 的观点

作者 杨赛

最近几年随着云计算的兴起和 DevOps 理念的流行，软件工程师领域有关“运维也要会开发”、“运维要自动化”、甚至“运维工程师要失业”这样的话题开始被越来越多的提起并讨论。

今天 InfoQ 中文站邀请到的嘉宾是一位资深的运维工程师，他是从开发工程师转岗成运维的。运维工作的界限将产生怎样的变化？运维工程师未来的职业发展应该如何规划？运维工程师为了适应时代和技术的变化需要去学习什么？让我们听听他的观点。

嘉宾简介

赵建春（Coati），腾讯业务运维 T4 专家工程师，社交网络事业群运维总监，技术运营通道委员。04 年大学毕业后加入腾讯，先后参与过交友、音乐、贺卡、QQ 空间等业务的开发。06 年后和团队一起专注于技术运维，负责腾讯社交网络事业群社区类 WEB 业务的运维和建设工作至今。经历了业务规模从数十台设备到数万台设备的快速发展历程。过程中 Coati 在运维环境标准化，业务 Set 化，运维自动化及多地分布式部署等方面积累了丰富的实战经验。

[Coati](#) 是 [2014 年全球架构师峰会（ArchSummit）](#) 的联席主席之一。

InfoQ：Coati 你自己是开发转运维的背景，当时是什么因素导致你决定要做这个转变？

Coati：刚进入公司的时候，我主要做 QQ 交友、QQ 音乐等业务的开发工作。那时候的这些业务的规模还比较小，后来非常幸运的参与到一个新项目——QQ 空间的开发，负责日志和留言版 2 个模块。这个项目成为后来引领中国 WEB2.0 的标志性 SNS 产品，非常受欢迎。顺利完成我所负责的模块、QQ 空间发布之后，我又以项目经理的角色，和另外 4 名同事一起，共同完成并上线了当时为企业版 QQ（TMQQ）定制的简洁版空间 izone。

在当时，还没有太多可以让用户个性化展现自己的 SNS 类产品，QQ 空间这种满足 QQ 用户个性化展现自己的产品呈现出爆发式的增长。很快，服务的稳定性和速度出现很大挑战，故障不断。于是，团队决定暂停大功能版本的开发一段时间，把团队分成 2 部分，一部分同事负责性能优化，一部分负责运营版本开发。我被安排做为运营版本开发

的负责人，一方面负责日常运营版本的开发，同时为了保障产品质量，我们在做业务开发的同时还做了很多监控、容量管理、发布流程优化、编译自动化和灰度扩容迁移等工具和系统。

2006 年开始公司正好在全公司推广 D/O 分离，也在我所在的业务系统成立了专门的运营部，因为我们运营开发团队承担了几乎所有的运维类工作，所以运营部一直没有人员对口 QQ 空间业务。后来我和我的团队顺应公司的 D/O 分离大方向，调到了运营部，转型为业务运维，走到了技术领域一个更加细分的领域。

InfoQ：你如何对运维的工作进行划分？比如，哪些工作属于运维，哪些属于 DBA，哪些属于研发，哪些属于测试？在运维当中，哪些属于基础运维，哪些属于应用运维？在你负责的部门，是如何对运维工程师进行分工的？

Coati：我们有个两个理念，我也常给团队同事讲，叫做：

1. 减少运维对象
2. 专业分工

专业分工这点打一个比方，自打有人类以来就有了建筑行业。如果我们现在看建筑这个行业，1-2 个人也可以建造出一个结构简单的木屋或土屋。但如果要建造一个类似腾讯大厦的几十层的摩天大楼，必须靠一个分工细致的、在建筑业各个领域都很强的团队精密合作才可以。常常听到房地产相关的报道说如果房地产泡沫破裂，会影响上下游几十个行业，可见建筑领域的行业细分是多么的细致。

运维相比传承了几千年的房地产行业来说，发展还不到 10 年，也是互联网技术行业里分工最不明确的一个岗位，几乎什么都要懂，什么都要做，就好比让我们每个人都具备建造一座房子所有相关的知识和能力。但很明显，我们任何一个人都无法建造一座腾讯大厦，只能通过专业细分，做精一个领域，只有这样，我们才能成为某一领域的专家。

减少运维对象，实际上是专业分工的手段。我们把服务器类型、机房数量、QA 流程、容错架构、软件架构等都看成抽象的、需要运维去管理的“对象”，希望这些对象越少越好。因为对于运维来说，人员数量总是远少于开发的，对象越少，我们越是能够对这些对象进行更加深入和全面的掌握。而这种寻找、合并同类项的过程，其实也是专业细分的手段。

目前我们的团队是按这样的方式划分的：

- 接入层运维团队，负责所有从用户客户端发起-域名-lvs/tgw-web 服务器这个链条上的服务
- 数据层运维团队，负责后端从最底层的数据存储-cache-cache 前面的 access 层。
我们不叫 DBA，因为 DBA 的概念有些局限了

- 逻辑层运维团队：中间最复杂的各类架构的tcp/udp的socket服务器，运维成立了一个专门的团队，推广通用socket服务器，让开发只写这个服务器里面的业务逻辑部分，就像web服务器上的CGI一样。这个团队可以叫做逻辑层运维团队，他们的职责之一就是让开发个性化的socket server越少越好，最好没有
- 业务运维团队：3层分开维护后，需要有人或机制让3层很好的协调工作和运转。这个从人员方面讲就是业务运维团队。虽然我们希望没有开发个性化的socket server，但这毕竟是个美好的设想，实际环境中依然会有个性化，于是这个团队就负责这些个性化的socket server，同时协调3层运维团队来为业务整体提供服务，可以说是对口业务的运维线PM
- 基础运维（目前由逻辑层运维团队兼任）：从技术角度来看，3层的访问需要访问的串接，我们使用类似DNS的一个名字服务组件来串接3层的访问关系、一些3层都使用的公共运维和管理组件、以及和网络服务器接口的一些工作
- 网络和系统运维：由于公司有专门的网络和服务器团队支持各个BG，所以我们在网络和服务器部分的精力不用投入太多

总结来说，就是接入层运维、逻辑层运维、基础服务运维、数据层运维、业务运维和系统运维几大分类。我想这个分类，也会随着规模的变化不断细化。

对于几个小组，我们有对团队核心工作的明确定义，我在这里摘录其中一些条目让大家大概了解一下。从这些条目可以看出，分层的团队都有一条相同的能力要求——就是让自己所维护的对象变的一致和尽可能的少，从而提高效率。而每个团队也要有自己核心的建设方向，以便沉淀相关的能力。比如业务运维团队更多的是项目规划协调和业务架构优化分布能力。

接入运维：

1. 全面就近的业务接入覆盖及技术加速、节流
2. 用户接入问题的全方位诊断系统和方法
3. 组件的高度统一以及统一后的经验最大化利用，成本及质量最优、批量和一致的操作，提供必要的自助化能力

逻辑运维：

1. 标准组件推广，尽可能减少特殊组件
2. 三层共同需求的服务和组件的维护，提供透明化服务，承上启下
3. 组件的高度统一以及统一后的经验最大化利用，成本及质量最优、批量和一致的操作，提供必要的自助化能力

存储运维：

1. 硬盘和内存数据的快速迁移、分裂、组合能力
2. 清晰明确的仓库资源归属、成本核算等，使服务信息透明（数据集群化后的需要）

3. 组件的高度统一以及统一后的经验最大化利用，成本及质量最优、批量和一致的操作，提供必要的自助化能力
4. 保障数据 100% 安全

业务运维：

1. 协调运维团队资源，支持业务项目对运维团队的整体需求；对业务整体的质量、成本负责
2. 规划科学合理的业务分布、推进，实施业务的架构革新、改造
3. 规划建设以业务视图为视角的运维工具和监控平台

我们的运维和测试之间的分工比较明确，很少有交叉。由于我们有比较完善的发布和自动化测试系统，所以测试同事负责了 WEB 类版本从开发到测试，到预发布环境的所有工作，并且在版本测试通过后，由测试发布外网。而运维则负责全新业务搭建、扩容迁移、以及后台 SERVER 的更新（更新量小）。

和开发之间，界限就是现网由运维负责，但对于故障的响应和处理，我们一直有个传统就是开发和运维都要及时第一时间响应，以加快故障的修复效率，这个方面也非常感谢开发团队同事的长期支持。

InfoQ：为大规模系统做运维，应该是始于大型互联网公司的兴起，腾讯在大规模系统运维方面已经积累了很多年的经验。从您个人的经验，您感觉大规模系统运维的思路、理念在过去几年有什么变化？

Coati：确实，如前面所讲，在腾讯我感觉我们是从 06 年开始起步专门做业务运维的，到现在还不到 10 年时间。以我所在团队为背景说起过去几年的变化，我感觉有这几个大概的阶段：

1. 06 年前，业务规模普遍很小，当时所有开发同事自己维护自己负责的模块，甚至出问题后还要跑到机房去自己重启服务器，可以认为是运维的原始时代。
2. 06~08 年，各类国外 ITIL 管理理念引入的时代，突发事件、工作台、问题管理等流程系统。而运维也是不断的将自身的工作通过工具建设来提升，把终端操作转移到前端界面可视化操作。可以认为是工具和流程快速完善的几年。
3. 08~12 年，随着很多业务逐渐由百万级在线变成千万级甚至亿级同时在线，业务的 SET 化、全国分布、容错容灾、异地调度等架构优化成为除了工具效率改进工作外的一个重点。这个阶段可以说建立了海量运维架构体系的几年，工具建设和架构改进互相促进发展。
4. 12 年到现在，我想大家都在研究和努力怎么把业务云化，尝试做到不做干预的扩缩容变更吧。同时我们还在尝试，如何利用运维更全的信息、更多的数据的平台积累优势，让运维同事能够帮助到业务目标，促使业务成功。我们提出服务产品、服务研发、服务自己的口号，把产品放在第一位，自己放在最后一位，让大家以业务成功为导向，而不是一直关注自己的效率问题，只关注自动化运维，把

自己放在第一位。比如对资源消耗型业务分析 top 用户资源使用，让产品团队可以更好的设置商业化方案。

InfoQ：从前几年开始流行的 DevOps 理念提议将运维工作以结合脚本、工具的方式实现自动化。自动化运维其实是一整套体系，从开发环境、测试环境的搭建，到代码的集成、测试，到上线部署、回滚，以及一系列的监控、报警体系，都包含在内。你们在实现运维自动化的过程中都经历过哪些阶段？目前完成到了什么状态，下一步计划是什么？

Coati：我们的运维自动化是一个持续演进的过程，如果非要分几个阶段，我觉得是运维原始阶段的自动化脚本命令行执行->web 化界面形成独立的子系统->独立子系统整合成完整业务或组件 OMS->自动调度云化的一个过程。

目前的自动化运维以我 13 年 4 月在 QCon 北京分享的《[海量 SNS 社区网站高效运维探索](#)》中的管理方案为主，通过以业务和组件为维度的 OMS 管理系统做自动部署。同时我们这 2 年来一直在编织我们的长尾业务的“织云”项目，使用腾讯云的 LXC 的 container 管理平台进行云化。最新的进展是我们有 330 多个模块实现了无需人为干预的全自动化扩缩容。以 4 核为主的 Linux container 实例数 6300 多个，很好的解决了长尾业务的低负载以及变更少雷区多的问题。

下一步的设想是将实体机模块也逐渐接入这个系统，将方案统一。两者的主要区别在于设备资源的管理，container 可以随时创建和销毁，实体机则不行，受资源数量和类型限制更大。

InfoQ：你认为未来企业还需要雇佣维护单机服务器或者几台服务器的内网小集群的系统工程师吗？

Coati：我觉得还是会需要小集群的工程师。虽然未来一定是大公司越来越大、越巨无霸，对运维人员的需求会像我前面讲的会要求分工更加专业和深入，但同时互联网的疆界也会不断扩大，很多传统行业会逐渐拥抱互联网，也会不断有创业公司成长为中小企业，使用云平台的公司会越来越多。小企业不会请太多人，所以几个人的团队势必要求大家是全才，导致技术不会很深入；更多的小企业的工程师则可能是使用云平台等资源维护企业服务；而大企业则是需要分工更加精细的专业化运维团队。

查看原文：[云计算时代的运维分工与理念变化：腾讯资深运维 Coati 的观点](#)

相关内容

- [左耳朵耗子谈云计算：拼的就是运维](#)
- [云计算时代的运维与安全](#)
- [交付云计算的复杂性](#)



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

嵌入式 OS 的现状、智能的物联网与未来的机器人

作者 罗未

嵌入式开发是一个低调的领域。相比 Web 开发和企业级开发，嵌入式开发这一领域似乎很少在互联网上发出声音。随着智能设备的兴起，智能手环、手表、眼镜、灯泡等产品成为互联网企业的下一个目标，而物联网这一概念再次复苏，嵌入式开发开始引起很多互联网工程师的关注。

那么，现在的嵌入式开发是怎样的？相比十年前、二十年前有什么发展？“物联网”这一大概念下，应用开发者应从何切入？本次访谈，我们邀请到一位不那么低调的嵌入式开发者，来跟我们分享他对这些问题的看法。

嘉宾简介

罗未（Noel），豌豆机器小组（WRTnode machine team）发起人，致力于整合机械设计、嵌入式 Linux 开发、计算机视觉、机器学习方向，以开源的理念制造智能交互机器，希望为开源社区和大众市场带来各种伴随人类却又独立于人类的机器。个人出身于行业软件领域，3 年前转入硬件方向，经历过智能家居和路由器行业，现希望做一些让未来更近的事情。

[罗未](#)是 [2014 年全球架构师峰会（ArchSummit）](#)的联席主席之一。有关他的更多介绍可参考技术人攻略对他的访谈：[开放制造的机器之心](#)。

以下内容根据 InfoQ 中文站编辑跟罗未的沟通整理而成。

嵌入式操作系统现状

目前嵌入式设备主要分为两大类：MCU 设备和带 MMU 的 CPU 设备。

MCU（Micro Computing Unit），也就是我们常说的单片机，其特点是 Micro：主频大概在几十 MHz，内存在几 KB，Flash 非易失存储也是几十 KB，资源小，价格便宜。单片机这个领域从 80 年代、90 年代开始就一直有人玩，像是玩具、闹钟、计算器、电子表、工业控制等很多领域都有用到，应用广泛。单片机程序的特点是逻辑简单、实时性强没有等待，不像 Linux 那样会存在资源被其他程序占用的情况。

早期单片机程序一般都是裸写 C 代码的方式，用一个大循环把所有事情搞定，所有的底层功能——如资源分配、进程调度、DNS 查询、域名转换等，都要手写实现。前几年开始有一些基于 MCU 的操作系统，比如 [μC/OS](#)、[RT-Thread](#) 等，单片机有了操作系统就相当于资源分配、进程调度等工作不用手写了，可以交给系统去管理，程序员不用去管任务间协调的问题。这可以看作是第二代单片机开发环境。

近几年有一些新的单片机操作系统，比如 [Contiki](#)，这套系统的特点是把互联网特征作为基础的构建。这套系统很牛，用 10KB 以内的内核就提供了对 HTTP、TCP/IP 等协议的支持，让单片机上来就可以联网，让单片机开发者绕过了每次都要裸写这些基本功能的痛苦。

现在的单片机有些很神奇的应用，比如图像识别、语音识别，可以做到在视频上识别色块的程度。但是，单片机如果又要做图像识别又要上网，就会非常吃力，毕竟资源十分有限，需要有很高的开发能力把它们协调好，这种情况下就不能用操作系统了。

以上是单片机的情况。另外一种是更大一些的，就是自带 MMU（Memory Management Unit，内存管理单元）的设备。这种设备的主频一般在几百 MHz 以上，内存有几十 MB 以上，早些年的智能手机就差不多是这个配置，跟十几年前的 PC 机配置差不多，所以安装运行 Linux 系统是没有问题的。这类设备其实也做了十多年了，现在用的比较多的架构有两个：ARM 和 MIPS，都是商业的，现在新的硬件基本上都是这两种架构。

有很多发行版都专门为 ARM 做过安装包，比如流行的 Ubuntu 和 Debian。无论是 ARM 还是 MIPS，因为有了系统，开发起来要比在单片机上舒服多了，但也仍然有一个很麻烦的地方，那就是要做交叉编译。开发者一般都是在自己电脑上——大部分是 x86 架构——完成开发的，因此要用 x86 上的 ARM 编译器交叉编译出 ARM 的二进制文件，用 MIPS 编译器交叉编译出 MIPS 的二进制文件，才能在设备上运行，这为调试带来了不小的麻烦。为什么我们这个圈子门槛比较高，就是因为一般都是掌握了交叉编译的开发者才会进来玩。不过好在有一个叫做 [GDB \(GNU Debugger\)](#) 的工具可以做远程调试，减少一些麻烦。

物联网终端需要完成的工作

现在有一种 M2M（Machine to Machine）的思路，在终端用可以联网的单片机做最简单的事情，比如开关一个灯泡；终端直接跟家庭网络的网关（路由器）连接，或直接跟公网的云端连接，由云端做更复杂的计算和处理。

这种思路可以解决一部分问题，但是我觉得还不够。终端需要做更多的事情。

我认为终端需要是智能的，它们需要达到“机器人”的层面。现在我们说的机器人跟以前大家理解的那种人型机器人不同，现在所说的机器人是一种复杂控制系统，是软件，可以跑在各种各样不同姿态的设备上。机器人需要完成三项工作：

1. 感知：从传感器采集数据
2. 交互：网络传输（如 HTTP、TCP/IP）和物理控制
3. 智能：如图像识别、语音语义的理解、智能规划，需要抽象成智能的算法

现在的机器还处于太过依附于人类的状态，需要人告诉他要做什么。我觉得未来的机器应该自己知道要做什么事情。现在的人工智能、知识图谱的建立就是奔着这个方向去的，比如 Google 工程师训练机器，让机器在 Youtube 的视频里认识猫，这个涉及到一个很大的知识库和训练过程，需要云端的协助。但最终训练出来之后，其实猫的图像识别特征数据是很小的，可以放在终端的机器人里，他们自己就会认识猫了。这就好像婴儿的学习过程一样。

但是跟婴儿不同的是，机器天生是执行器。所以结合认知能力，让机器认识猫了之后，加上执行，是不是可以让机器自动的去抓猫或者逗猫玩？机器认识电梯之后，是不是能够自己去按电梯？机器认识无线充电站后，是不是能够自己跑到无线充电站上面蹲着充电？随着知识图谱的建模完善，事物和事物之间的联系能够被机器理解，机器人会变得越来越强大，越来越重要。

其实现在语音语义的知识图谱建设已经相对完善了，机器已经能够理解一些上下文之间的关系，比如你说到吃苹果，他就知道你说的是什么意思。我们现在在语音语义+网络这块直接使用了讯飞的服务，我们把工具链给他，他们帮我们生成了一个二进制包给我们，就很方便了。

技术上的挑战

上述这些工作当中，有些单片机可以完成的很好，有些不能。单片机可以采集一些简单的数据如位置、高度、重力加速度、四轴姿态、温度、湿度等，进来都是数字，只需要做 AD 转换。比较复杂的数据如声音、图像，单片机处理起来就比较困难，一般我们通过 Linux 的 USB 驱动来跑，需要 MMU 的芯片。但是单片机有一个特征是 Linux 无法满足的，就是实时性。很多物理控制对实时性的要求很高，比如四轴飞机的控制，严格要求 50Hz 的控制频率，即一秒进行 50 次计算来决定下一帧的动作，如果稍微有点资源抢占造成延迟，飞机就掉下来了。

为了同时达成实时计算+复杂性这两个目的，我们只好把两个芯片加在一起。但是两个芯片在一起，就成了一个分布式系统，有芯片级的通信问题，同时开发者还需要写两套代码，又要写单片机的交叉编译，又要做 Linux 开发，各种调试和测试的困难。Arduino 现在已经有一套挺完善的思路：首先它的传感器、控制器的库都很全，然后它做了一个 [ArduinoYUN](#) 的板子，就是一个 OpenWRT（一个超级精简的 Linux 发行版）+单片机的双芯片板子，然后它有一个万用固件——一个支持 [firmata 协议](#) 的库，算是一个翻译，只要符合这个协议就可以从 Linux 控制 Arduino，算是一种思路。但是我觉得这个思路有两个问题：第一，ArduinoYUN 的思路是以 MCU 开发为主，把 OpenWRT 当做单片机的透传模块，为单片机提供网络服务。放着强大的芯片在一边，用小小的单片机跑主

程序，感觉未免太浪费。第二，firmata 协议虽然简化了控制，但是又影响了实时性，在实时性要求较高的时候（比如四轴飞机），这种思路又无法满足需求了。

现在一些芯片公司已经开始意识到这个问题，开始考虑如何把两者封装成一个芯片，来满足实时性+复杂性的结合。我认为封装后应该要以 Linux 为主要的开发平台和软件运行平台，以 MCU 作为辅助以满足实时性需求。

所以，实时性+复杂性的结合是第一个挑战。第二个挑战是复杂运算的加速，比如 H.264/H.265 的视频压缩、图像识别的硬件加速，要不要放在机器人的芯片里？我觉得是需要的，但是不需要手机那么强的 GPU，有一个视频压缩的芯片放在里面就可以。终端如果能做视频压缩，多半也能做图像识别，那么终端机器人可以做的事情就更多。

第三个挑战是针对 Linux 内核本身的，就是在这种级别的计算平台上如何进行更合理的裁剪、做更合理的算法策略、执行策略。OpenWRT 的开发版现在我们做到 64MB 的运行时内存占用，而一般的路由器芯片都是 16MB、32MB。其实内存的空间占用倒不是大问题，因为现在内存很便宜，就算用到 128MB、256MB 也没什么，但是关键在于时间片的占用。所谓省资源其实就是两个意思：少占地儿+少占时间，这样才能低延迟。所以 Linux 内核如何解决这个问题，也是一个比较大的挑战。

这三个点可能是未来几年这个产业很多人的努力方向。

总结

相比十年前裸写 C 代码的场景，现在我们有图形化的界面，有 RESTful API，嵌入式开发的难度可以说已经大大降低了。虽然有上面提到的基础设施与开发工具的挑战，但我认为用不了几年时间也都能解决。网络连接现在已经基本不是问题，3G、4G、Wifi 已经足以支撑大部分智能设备的应用场景。

但是，仅有这些，到“智能的物联网”有很大的距离。机器需要学习更多、建立更多的知识图谱，才能变得更加强大。现在云端还没有太多现成可用的知识图谱，但我们仍然可以先从简单的事情做起，比如让机器人扫地，让机器人把空瓶子扔进垃圾桶，一点一点的改进它们。也希望有更多的开发者能够加入这一进程，让我们的世界变得更加完整。

感谢[杨赛](#)对本文的审校。

查看原文：[嵌入式 OS 的现状、智能的物联网与未来的机器人](#)

虚拟研讨会：在低延迟环境中使用 Java

作者 Charles Humble，译者 夏雪

以前，C 和 C++ 是低延迟环境事实上的选择，但现在 Java 使用的越来越多了。

InfoQ 有幸邀请到了这个领域的四位专家，跟他们一起讨论是什么推动了这一趋势，在这种情况下使用 Java 有哪些最佳实践。

与会者名单：

Peter Lawrey 是一位对低延迟和高吞吐量系统很感兴趣的 Java 顾问。他曾为多家对冲基金、交易公司和投资银行提供过服务。

Martin Thompson 是一位高性能和低延迟方面的专家，具有二十多年的大规模事务处理和大数据系统的工作经验，涉足过汽车、博彩、金融、移动和内容管理等领域。

Todd L. Montgomery 是 [Informatica Ultra Messaging](#) 的副总架构师，29West 低延迟消息传递产品的首席设计师和实现者。

Dr Andy Piper 最近从 Oracle 离职加入到了 [Push Technology](#)，任职首席技术官。

问题列表：

1. 我们如何理解低延迟？它和实时一样吗？它与高性能代码一般有怎样的联系？
2. 在其他情况下使用 Java 往往会提到以下优势：可以使用丰富的类库、框架和应用服务器等等，而且还有大量掌握它的程序员用户群。在编写低延迟代码时还有这些优势吗？如果没有，那与 C++ 相比 Java 有哪些优势？
3. JVM 是如何支持并发程序的？
4. 如果先不谈垃圾回收，那么 Java 还有哪些其他特有的技术有助于编写低延迟代码的技术（在 C++ 中不会用到的技术）？我能想到的有 JVM 预热，把所有类加载到 permgen 中以避免 IO，用于避免缓存未命中的 Java 专有技术等等。
5. 管理垃圾回收行为对大家用 Java 编写低延迟代码的方式有怎样的影响？
6. 在分析低延迟应用时，你有没有在性能的“峰值”和极端值背后发现任何常见原因或模式？
7. Java 7 已经开始支持基于 InfiniBand 设备的套接字直接协议（Sockets Direct Protocol, SDP）了。你是否已经看到有生产系统使用过它了？如果这项技术还没有被应用过，那么你看到有什么其他的解决方案吗？
8. 下面的问题并不局限于 Java，为什么我们需要尽量避免竞争？当我们无法避免竞争时如何能够更好地管理它？

9. 在过去的几年里，你们有没有为了用 Java 做低延迟开发而做出过改变？
10. Java 是否适用于其他对性能比较敏感的工作？你会把它用于高频交易（HFT）系统吗？能否举个例子？或者说 C++ 仍然是更好的选择？

问题 1.InfoQ：我们如何理解低延迟？它和实时一样吗？它与高性能代码一般有怎样的联系？

Lawrey：对延迟有严格要求的系统，延迟的时间甚至快到人们根本就看不到。这种延迟时间仅在 100 纳秒到 100 毫秒之间。

Montgomery：实时和低延迟完全不同。大多数人对于“实时”的观点是确定性，即严格控制（甚至界定）峰值的纯粹的速度。然而，通常“低延迟”意味着要最大限度地追求纯粹的速度，与此同时可以容忍个别轻微的偏差。当思考硬实时系统时，这一点是肯定的。低延迟的一个先决条件就是要始终对效率保持足够的敏锐。从系统的角度看，这种效率必须渗透到全部应用栈、操作系统和网络中。这意味着低延迟系统必须让所有其他组件都能达到机械和应的程度。另外，最近几年在低延迟系统中涌现了许多技术，它们源自于操作系统、编程语言、虚拟机、协议、其他系统开发领域乃至硬件设计的高性能技术。

Thompson：性能无非就是两点——吞吐率（比如单位/秒）和响应时间（有时也称为等待时间）。最重要的是给出量化的指标，而不能只是说它应该“很快”。实时有非常明确的定义，但却经常被误用。实时与具体系统相关，这些系统在不考虑系统负载的情况下对输入事件到其响应有实际的时间限制。在硬实时系统中如果不能满足这些限制，那么整个系统就会出现故障。大家可以想一下心脏起搏器或导弹控制系统，它们能帮我们更好地理解这一概念。

对于交易系统来说，实时系统更倾向于另外一种含义，那就是系统必须拥有高吞吐率并且尽快响应每个事件，这可以理解为“低延迟”。但如果有一次交易没有被及时处理并不代表整个系统发生了错误，所以严格意义上你不能把它称为实时。

好的交易系统要有高质量的执行，其中一个方面就是要有低延迟的响应，响应时间只能有很小的偏差。

Piper：简单来说延迟就是决策与行动之间的延时。在高性能计算环境下，低延迟通常意味着网络间的传输有很低的延迟时间或请求到响应间有很低的整体延迟时间。“低”的定义取决于具体环境，在互联网中低延迟可能是指在 200 毫秒以内，但是在交易应用中可能就是在 2 微秒之内了。从学术上来说低延迟与实时是有区别的，低延迟通常用百分比来度量，度量那些必须要掌握的异常值（未达到低延迟的情况）。而如果是实时的话，你就必须保证系统行为在最大延时之内响应，而不再是度量延迟百分比。你会发现实时系统很容易做成低延迟系统，但反过来就很难了。但是现在，人们渐渐地已经不再严格区分这些概念了，会混着用这些术语。

假设延迟是从请求到响应的整体延迟时间，那么很明显延迟会受到以下诸多方面的影响：CPU、网络、操作系统、应用甚至物理定律。

问题 2.InfoQ：在其他情况下使用 Java 往往会提到以下优势：可以使用丰富的类库、框架和应用服务器等等，而且还有大量掌握它的程序员用户群。在编写低延迟代码时还有这些优势吗？如果没有，那与 C++ 相比 Java 有哪些优势？

Lawrey：如果你的应用把 90% 的时间花在了 10% 的代码上，那么 Java 很难去优化那 10% 的代码，但却很容易编写和维护剩下的那 90% 的代码，尤其是当团队能力水平参差不齐的时候。

Montgomery：在资本市场（特别是算法交易）中有很多可以发挥作用的因素。更快的算法投入市场时往往拥有更多的优势。许多算法都有搁置期，尽快进入市场是充分发挥其优势的关键。与 C 或 C++ 截然不同，Java 有更多社区和可用的选择，这无疑是它强大的竞争优势。虽然有时候纯理论上的低延迟可以不考虑其他关注点。但我想目前 Java 和 C++ 的性能差异并不大，单纯从速度上讲并没有那么黑白分明。通过对垃圾回收技术、运行期编译执行技术优化与运行期的管理的改进，Java 在一向比较薄弱的性能方面增加了一些非常令人赞叹的优势，这些优势不会轻易被人们忽视的。

Thompson：我们用 Java 编写低延迟系统时很少使用第三方甚至标准类库，主要有以下两个原因：首先，许多类库在编写时并没有专门考虑过性能，所以通常达不到令人满意的吞吐率和响应时间。其次，它们通常会用锁来控制并发，这样就会产生大量的垃圾，当锁竞争和垃圾回收时响应时间就会有很大的变数。

Java 有几个最好的工具，这些工具可以支持任何语言，它们能非常有效地提升生产率。上市时间往往是构建交易系统时的一项关键要求，Java 通常在这时总能占得先机。

Piper：很多方面都反过来了，以前很难用 Java 写好的低延迟代码，因为 JVM 把开发人员与硬件隔离开。这是一个非常好的转变，不仅让 JVM 更快、更容易预测了，而且现在开发人员充分理解了 Java 的工作机制（特别是 Java 内存模型）以及它与底层硬件之间的映射方式（Java 可以称得上是第一个为程序员提供了全面的内存模型的流行语言了，C++ 也是在它之后才提供的），从而能够充分地发挥硬件的优势了。比如无锁、无等待技术都是比较好的例子，Martin Thompson 和我们公司（Push）一直在推进这些技术的应用，在我们自己的开发中已经非常成功地应用了这些技术。此外，由于这些技术越来越流行了，我们发现它们正被引入到标准库中（例如 [Disruptor](#)），所以开发人员在使用这些技术时已经无需再详细了解那些底层的行为了。

即使我们抛开这些技术不谈，Java 的安全优势（内存管理、线程管理等）往往也比 C++ 感觉上的性能优势更具价值，而且 JVM 供应商前段时间还声称主流的 JVM 通常比定制的 C++ 代码更快，因为它们可以进行跨应用的整体优化。

问题 3.InfoQ：JVM 是如何支持并发程序的？

Lawrey: Java 从一开始就内置了对多线程的支持，高并发支持标准已经有 10 年了。

Montgomery: JVM 是一个很好的并发程序平台。它的内存模型使开发人员可以在硬件抽象层上以统一的模式使用无锁技术，这个优点能让应用尽可能地发挥硬件的能力。无锁和无等待技术非常适合创建高效的数据结构，这正是开发社区中大家所急需的东西。此外，有一些用于并发的标准类库的结构非常易于使用，使应用可以更具弹性。不仅是 Java 使用了很多这样的结构，还有 C++11（如果抛开某些细节不谈的话）。C++11 的内存模型对于开发人员来说是一次非常巨大的进步。

Thompson: Java (1.5) 是第一个拥有详细定义的内存模型的重要语言。有了语言层的内存模型，就可以使开发人员在硬件抽象层上推理并发代码了。这一点至关重要，因为硬件和编译器将激进地重排我们的代码，这存在跨线程可见性问题。你可以使用 Java 编写很好的无锁算法，它能在低并可预估的延迟上实现非常惊人的吞吐量。Java 对锁也有很多的支持。然而，锁竞争时操作系统必须作为一个仲裁者介入，会消耗巨大的性能成本。有没有锁竞争会产生不同的延迟，通常有 3 个数量级的差异。

Piper: Java 从自身的 Java 语言规范开始就已经支持并发编程了——JLS 描述了许多支持并发的 Java 基本实体和结构。在基层是 `java.lang.Thread` 类，它用来创建和管理线程，关键字 `synchronized` 用来协调不同线程对共享资源的访问。除此之外，Java 还提供了完整的数据结构包 (`java.util.concurrent`)，从并发哈希锁到任务调度程序再到不同的锁类型，该包都已经针对并发编程进行了优化。Java 内存模型 (JMM) 是其中最大的一项支持，它是作为 JDK5 中的一部分并入到了 JLS 中。它确保了开发人员在处理多线程及其行为时可以有相应的预期。有了它之后开发人员就可以更容易地编写出高性能、线程安全的代码了。在开发 [Diffusion](#) 时，为了实现最佳的性能我们非常依赖于 Java 内存模型。

问题 4.InfoQ: 如果先不谈垃圾回收，那么 Java 还有哪些其他特有的技术有助于编写低延迟代码的技术（如果你用 C++ 就不会用到这些技术）？我能想到的有 JVM 预热，把所有类加载到 `permgen` 中以避免 IO，用于避免缓存未命中的 Java 专有技术等等。

Lawrey: Java 能让你编写、测试和剖析应用程序，使应用在有限的资源内更加有效。这使你能有更多的时间确保完成所有最重要的事。我见过很多 C 或 C++ 项目花了很多的时间去深度探讨底层，最终两个终端之间仍会很长时间的延迟。

Montgomery: 这个问题可有些难度。有一点比较明显，JVM 的预热可以做适当的优化。然而，目前的 C++ 无法在运行期做类层次分析时优化一些类和方法的调用。在 C++ 中可以使用很多其他技术，或者在某些情况下并不需要这些技术。不论哪种语言的低延迟技术通常都有一些建议，告诉你最好不要去做哪些事，它们会带来很大的影响。在 Java 中需要避免的对低延迟应用有不良影响的做法不算很多。其中之一是不要使用特定的 API，比如 Reflection (反射) API。非常幸运的是，我们通常可以用更好的方案达成相同的结果。

Thompson: 答案就在你的问题里，:-)。从本质上说，Java 必须先预热才能使运行期达到稳定状态。一旦稳定下来 Java 就可以像本机语言一样快了，甚至在某些情况下会更快。Java 最大的弱点是缺少对内存层的控制。在主流处理器中一次缓存未命中就会丢失 500 条已经执行过的指令。为了避免缓存不能命中，我们需要控制内存层，以一种可预测的方式访问内存。为了达到这种程度的控制，并减轻垃圾回收的压力，我们通常要用 DirectByteBuffers 去创建数据结构，或者放弃堆而使用 Unsafe。这就可以做到精确的数据结构规划。如果 Java 引入对结构体数组的支持，就不必再这么做了。这并不需要切换语言，只是引入一些新的特性。

Piper: 这个问题似乎是一个伪命题。综合所有情况来看，编写低延迟程序还是编写其他特别关注性能的程序是非常类似的（无论 C++ 还是 Java），无非是让开发人员编写的代码在某些层进行间接处理后（比如，通过 JVM 或者通过 C++ 的类库、编译优化等）在硬件平台上运行，事实上这些细节的不同并不会造成多大的差异。优化本质上是一种演练，优化的规则一直都是像下面这样的描述：

1. 不要。
2. 也不要（只适用于专家）。

如果没达到你想要优化到的程度：

1. 看看你是否真的需要提速。
2. 剖析代码看看实际上把时间都花在了哪里。
3. 重点关注具有高回报值的区域，先不考虑其他部分。

当然现在你用工具做这些事的时候，Java 和 C++ 可能有不同的潜在热点，那只是因为它们本来就不同嘛。诚然，你需要比普通的 JAVA 程序员多了解一些细节（使用 C++ 也是如此）；相比而言使用 Java 时某些内容并不需要做过多深入地了解，因为在运行期已经对它们进行了充分地处理。你可能需要优化以下几个方面：可疑的代码路径、数据结构和锁。我们在 Diffusion 中采用了基准驱动法，我们不断地剖析我们的应用程序，寻找优化的可能性。

问题 5. InfoQ：管理垃圾回收行为对大家用 Java 编写低延迟代码的方式有怎样的影响？

Lawrey: 在不同的情况下有不同的解决方案。我首选的解决方案是把垃圾限制到最小，那么它就不会造成多大危害了。你可以把垃圾回收的次数降低到每天一次以内。

个人认为，这时候减少垃圾回收真正的理由是擦除尚未填满 CPU 缓存的垃圾。减少这些垃圾能为你提升 2 到 5 倍的代码性能。

Montgomery: 我发现大多数 Java 低延迟系统为了最小化甚至试图消除垃圾的产生已经做出了最大的努力。比方说，避免使用字符串都不算是什么稀罕事了。Informatica Ultra Messaging (UM)自己已经提供了特定的 Java 方法以迎合大量用户复用对象的需求，并避

免了一些使用模式。如果让我来猜的话，最常见的隐含式已经成为对象复用的流行用法。这一模式也受到了其他许多非低延迟类库的影响，比如 Hadoop。它目前已经成为社区内的一项常用技术，它为 API 或框架用户提供了选项和方法，使用户可以在低垃圾或者零垃圾的方式下使用它们。

除了代码实践方面的影响，运维也会对低延迟系统产生影响。正像我们说过的，许多系统将采取一些垃圾回收的创新。把垃圾回收的执行限制在每天特定的时间已经不再是一种罕见的方法。这意味着应用设计和运维需求是控制异常值和获得更多确定性的主要因素。

Thompson: 如果使用对象池（像前面的回答中所说的）就需要在 ByteBuffers 或空闲的堆中管理大部分数据结构，这使 Java 程序有了 C 一样的风格。如果我们拥有真正的并发垃圾回收器就可以避免这种结果。

Piper: `java.lang.String` 到底会有多大？不好意思，开个玩笑，实话实说，与其让每个程序员去修改他们的代码，还不如改进垃圾回收的行为。以 HotSpot 为例，从早期垃圾回收停顿以分钟计，到现在也走过了一段艰难漫长之路。许多改进都是由市场竞争驱动的，从延迟的角度来看，BEA JRockit 在过去的表现一向都比 HotSpot 要好很多，抖动要低得多。然而最近 Oracle 正在合并 JRockit 和 HotSpot 的代码库，原因恰恰是它们之间几乎没有多大差距了。在其他很流行的 JVM（比如 Azul 的 Zing）上也可以看到很多类似的改进，许多情况下开发人员试图“改进”垃圾回收的行为，但没有取得真正的效果，反而有的时候适得其反。

但是，这并不是说开发人员不能去管理垃圾回收，举例来说，合并和使用空闲的堆存储可以降低对象的分配，从而限制内存抖动。同时也应该想到，JVM 的开发人员也非常关心这些问题，所以你基本没必要自己去处理这些事，或者只需要购买一个商业的 JVM。最糟糕的是，你在这一方面优化应用的时候根本就不确定它是不是真的有问题，但从此以后由于这类技术绕过了 Java 垃圾回收那些非常实用的特性，从而增加了应用的复杂度，使之后的维护更加困难。

问题 6.InfoQ: 在分析低延迟应用时，你有没有在性能的“峰值”和极端值背后发现任何常见原因或模式？

Lawrey: IO 等待之类的。CPU 指令或数据缓存干扰。上下文切换。

Montgomery: 在 Java 里，大家开始更加充分地理解垃圾回收停顿了，我们很幸运能够用到更好的垃圾回收器，它更为实用。然而，系统影响对于所有语言都是共有的。在峰值背后的众多原因之中，躲藏着这么一个原因，那就是操作系统调度延迟。有时它是直接的延迟，而有时它是由于延迟引起的连锁反应，相比而言这更要命。在重负载之下某些操作系统的调度要比其他操作系统更好。出人意料的是，对于许多开发人员来说，糟糕的应用选择会经常使调度成为意外状况，而且往往难以充分调试。你需要注意来自于 I/O 的内在延迟和在某些系统上会发生的 I/O 竞争。一个好的假设是，任何 I/O 调用都

可能会在某些点和将在某些点上阻塞。往往关键是思考其内在的影响。请铭记，网络调用即 I/O。

还有许多网络方面的特定原因同样会造成糟糕的性能。我列举一下几条比较关键的原因。

- 网络通行要花时间。在广域网环境里，跨网间传播数据需要花费大量的时间。
- 以太网是非可靠的，它是提供可靠性之上的协议。
- 网内丢包引起延迟，这些延迟是由于中继和恢复以及类似于 TCP 队头阻塞的次级效应。
- 使用 UDP 时，由于资源匮乏造成在接收端发生各种方式的网内丢包。
- 由于交换机和路由器拥塞发生网内丢包。路由器和交换机是很自然的竞争点，它们之间产生竞争时丢包是权益之计。
- 可靠的网络介质（比如 InfiniBand）针对网络层的延迟在权衡之下会选择丢包。
 不过，丢包的最终结果同样还是会造成长期的延迟。

在很大程度上，大量使用网络的低延迟应用往往不得不考虑大量的延迟原因以及附加的网内抖动的来源。在很多低延迟应用中，抖动的最常见原因除了网络延迟外，有最大嫌疑的可能就是丢包了。

Thompson: 我清楚很多延迟峰值的原因。许多人都知道垃圾回收，除此之外我还清楚许多锁竞争、TCP 相关的问题以及许多 Linux 内核由于配置不当导致的相关问题。许多应用的算法设计得比较差，它们在突发情况下无法分摊那些开销很大的操作（比如 IO 和缓存未命中），从而形成排队效应。我发现算法设计常常是应用性能问题和延迟峰值的最主要的原因。

处理延迟峰值时，到达安全点的时间（TTS）是一个主要考虑因素。许多 JVM 操作需要通过使所有用户线程达到安全点才能中止这些线程。安全点检查通常是在方法返回上执行的。这需要安全点能够成为来自于撤销的倾向锁、某些 JNI 交互、未优化的代码直到许多垃圾回收阶段的任何事物。通常把所有线程发到安全点所花的时间比完成作业本身还要多得多。随后的工作是要花费巨大的成本去唤醒那些所有的线程让它们再次执行。让一个线程快速、可预见地到达安全点通常不是许多 JVM 考虑或优化的部分，比如对象克隆和数组复制。

Piper: 峰值最常见的原因是垃圾回收停顿，改善垃圾回收停顿最常用的方法是垃圾回收调优，这要优于实际去变更代码。例如，JDK6 和 JDK7 默认使用的是并行收集器（parallel collector），我们只是简单地把它换为并发标记清除收集器（concurrent mark sweep collector）就能使“全世界停止运行”的垃圾回收停顿产生巨大的变化，通常正是它导致的峰值。除此之外，你还要考虑到堆的大小。太大的堆会给垃圾回收带来更大的压力，会造成更长的停顿时间，一般情况下只需简单地消除内存泄漏和降低内存使用率就会使一个低延迟应用有截然不同的整体行为表现。

除了垃圾回收之外，延迟峰值的另一主要原因是锁竞争，但由于它通常难以确定，所以使它更加难以识别和处理。另外一定要牢记的是，任何时候应用都没有能力去处理它，它将产生延迟峰值。很多情况下都会产生锁竞争，有一些甚至在 JVM 控制之外，比如访问内核或操作系统资源。如果可以识别出这些限制，那么完全可以修改应用使它不使用这些资源，或者改变使用这些资源的时间。

问题 7.InfoQ: Java 7 已经开始支持基于 InfiniBand 设备的套接字直接协议（Sockets Direct Protocol, SDP）了。你是否已经看到过有生产系统使用过它了？如果这项技术还没有被应用过，那么你看到有什么其他的解决方案吗？

Lawrey: 因为它会产生相当多的垃圾，所以我没有把它用于以太网。在低延迟系统中，你希望把网络跳数降到最低，通常唯一不能移除的就是外部连接。这些通常都是以太网。

Montgomery: 我们见过的不多。在前面我提到过它，但我们还没有看到它被认真考虑过。Ultra Messaging 是用来在 SDP 和开发人员之间使用消息传递的接口。SDP 非常适合用于(R)DMA 访问模式，而不是基于推送的应用模式。虽然可以把 DMA 模式转变为推送模式，但很遗憾，这不适合用 SDP 来做。

Thompson: 我还没有见过它在实际环境中的应用。多数人使用类似于 OpenOnload 栈和那些来自于 Solarflare 或 Mellanox 之类的网络适配器。在极端情况下，我看到在 InfiniBand 上的 RDMA 使用预定义的锁无关算法直接从 Java 中访问共享的内存。

Piper: Oracle 的 Exalogic 和 Coherence 这两个产品已经用过 Java 和 SDP 有一段时间了，所以从这个意义上来说，我们已经见过这一特性在产品系统中应用过一段时间了。按照开发人员实际使用 Java SDP 代替某些第三方产品去支持目录的情况看，也没有太多，但是如果他能增加商业利益，那我们预计这一点会有所改变。我们自己已经使用了针对延迟优化过的硬件（例如 Solarflare10GbE 适配器），从核心驱动安装包中获得的好处要优于具体的 Java 调优。

问题 8.InfoQ: 下面的问题并不局限于 Java，为什么我们需要尽量避免竞争？当我们无法避免竞争时如何能够更好地管理它？

Lawrey: 对于追求极致的低延迟来说，这的确是个问题，但对于几微秒级的延迟这就不是个问题了。如果你无法避免这种情况，就要尽量把它影响降到最低吧。

Montgomery: 竞争总会发生的。对它的管理至关重要。处理竞争的最佳方法之一就是架构。“单一写原则”是一种有效的方法。其实，假设有一具单独的写入器，并围绕这一基本原则构建的话，那么就不会有竞争了。使单一的写的工作降到最低，你们会为完成的效果感到惊奇的。

异步行为是一个能够避免竞争的很好的方法。它总是围绕着这样的一个原则：“永远只做有用的工作”。

这通常也会变成单一写原则。我通常喜欢在竞争资源的单一写入器前放一个锁无关队列，用线程执行所有的写操作。线程什么都不做，它只是把写操作推入到队列中，然后这些写操作会在循环中执行。这么做非常有利于批处理。队列方面，一种无等待的方法在此有极大的帮助，从调用者的视角来看，就是在这里执行的异步行为。

Thompson: 一旦我们的算法中有了竞争，我们就会有一个基本成正比的瓶颈。竞争点形成队列，利特尔法则（[Little's Law](#)）开始起作用了。我们还可以使用阿姆达尔定律（[Amdahl's Law](#)）模拟竞争点的时序约束。大多数算法可以被重写以避免来自多线程或执行环境给定的并行加速（通常通过管道）的竞争。如果我们真的必须管理指定数据资源的竞争，那么处理器提供的原子指令往往是比锁更好的解决方案，因为它们是在永远不会涉及内核的用户空间运转的。新一代英特尔处理器（Haswell）扩展了这些指令，使硬件事务型内存支持数据原子性的少量更新。但很遗憾的是，Java很可能需要花上一段时间才能为程序员直接提供这种支持。

Piper: 对于低延迟应用来说，锁竞争可能是最大的一个性能障碍了。锁本身并没有多少性能开销，在无竞争情况下 Java 的同步锁也可以执行地非常好。然而，有竞争时锁的性能就会一落千丈了，不仅仅因为一个线程持有锁使其他线程拿不到同一个锁，还因为更多线程访问这个锁时会给 JVM 带来昂贵的锁管理成本，这个道理很容易理解。很明显，关键是要避免锁竞争，所以不要同步那些不需要同步的东西（比如，移除那些什么都不保护的锁、缩小锁的范围、降低锁持有的时间、不要混淆锁的职责等等）。另一种常用的技术是消除多线程访问，不要让多个线程去访问一个共享的数据结构；你可以把更新操作当成命令排成队列，这样就变成了单线程处理。这样就把锁竞争简单地归结成了添加在队列中的一个项目了，而它可以通过锁无关技术来实现自我的管理。

问题 9. InfoQ: 在过去的几年里，你们有没有为了用 Java 做低延迟开发而做出过改变？

Lawrey: 构建一个简单的系统，让它只做你想让它做的事情。尽可能地对它进行端到端的调优。优化和（或）重写那些你测量出瓶颈的地方。

Montgomery: 翻天覆地地变化。Ultra Messaging 创建于 2004 年。在那个时候，想把 Java 用于低延迟可不是一个很明智的选择。除了极少的几个人确实考虑过它。后来人越来越多。我想现在这种局面已经被彻底扭转了。Java 不仅是可行的，甚至可能成为低延迟系统的主要选择。Martin Thompson 和[Azul Systems'] Gil Tene 完成了这项伟大的工程，他们真正地推动了社区对此的态度转变。

Thompson: 最近几年的主要变化是持续完善锁无关和缓存友好算法。我喜欢经常参加一些与语言相关的论战，在这些论战中抛出观点来证明在性能方面算法比语言更加重要。不管是什语言，干净的代码（它展示了机器响应）更容易带来优异的性能。

Piper: Java 虚拟机和硬件正在不断地改进，低延迟开发永远都是一场军备竞赛，为的就是能够保持在目标架构的最佳位置。JVM 的 Java 内存模型和并行数据结构（这依赖于底层硬件的支持）的实现也更加强壮、可靠了，所以像那些锁无关、无等待技术也已经成为主流。硬件现在的发展方向也是在越来越多的执行内核的基础上追求越来越多的并发，所以那些充分发挥这些变化的优势技术，以及那些尽可能降低冲击（比如给避免锁竞争增加更多的权重）的技术正在成为开发环节的要点。

在 Diffusion，我们现在已经在标准版的 Intel 硬件上用标准版的 JVM 把延迟时间降到了 10 微秒之内。

问题 10. InfoQ: Java 是否适用于其他对性能比较敏感的工作？你会把它用于高频交易（HFT）系统吗？能否举个例子？或者说 C++仍然是更好的选择？

Lawrey: 以上市时间的角度来说（团队的可维护性和支持的综合能力），我认为 Java 是最好的。C 或 C++ 在你要使用 Java 和 FPGA（或 GPU）之间的空间一直在不断地缩小。

Montgomery: 对于大多数高性能工作来说，使用 Java 肯定都会是一个明智的选择。对于高频交易（HFT）来说，Java 几乎已经有了所需的一切。它有更加广阔的发挥空间，尽管最明显的是有了更多的内联函数。我认为，Java 在其他领域可以做得很好。就像低延迟那样，我认为它会让开发人员去乐意尝试也同样做到的。

Thompson: 如果有绝对充足的时间，我就能让 C（或 C++ 或 ASM）的程序性能比 Java 更好，但现在我没有那么长的时间。通常 Java 是一种非常快速的交付方式。如果 Java 有好的并发垃圾回收器、内存层的控制、无符号类型以及一些访问 SIMD 和并发基元的内联函数，那我可要变成一只非常快乐的兔子喽。

Piper: 我把 C++ 看作是一种最优化的选择。从上市时间、可靠性、高质量的角度来看，Java 是迄今为止首选的开发环境，所以我常把 Java 作为第一选择，除非确实有 Java 无法解决的瓶颈，否则不考虑换其他的语言，这已经成了我的口头禅。

座谈会小组成员简介

Peter Lawrey 是一名对低延迟和高吞吐率系统很感兴趣的 Java 咨询师。他曾为多家对冲基金、交易公司和投资银行提供过服务。Peter 在 StackOverflow 上的 Java 方面排在第 3 名，他的技术博客每月有 12 万的页面浏览数，他是 github 上 OpenHFT 项目最重要的开发人员。OpenHFT 项目包括有 Chronicle，它每秒最多支持 1 亿的持久化消息。Peter 每月会到性能 Java 用户组就不同的低延迟主题做两次免费的讲习会。

Todd L. Montgomery 是 29West 的 Messaging Business Unit（现在已经隶属于 Informatica）的副总架构师。Todd 作为 Informatica 的 Messaging Business Unit 的总架构

师负责 Ultra Messaging 产品系列的设计与实现，该产品系列已经有超过 170 多个产品在金融服务业得到了有效地应用。在过去，Todd 曾负责过 TIBCO 和 Talarian 的架构工作，还负责过 West Virginia University 的研究和演讲，曾为 IETF 做出过贡献，完成 NASA 在各个软件领域的研究工作。在消息平台、可靠的多路广播、网络安全、拥塞控制和软件质保等方面均有较深的资历，他以 20 年的实战开发经验为我们带来了一种独特的视角。

Martin Thompson 是一名高性能和低延迟专家，有着超过 20 年的大规模事务处理和大数据领域（包括自动化、博彩、金融、移动和内容管理）的从业经验。他认为机械和应（对硬件的理解，必将有助于软件的创造）是交付优雅的、高性能解决方案的基石。Martin 曾是 LMAX 的联合创始人和首席技术官，直至他离开去专门研究帮助他人使软件达到优越的性能。并发编程框架 Disruptor 是他创造的机械和应的其中一个实例。

Dr Andy Piper 近期加入到 Push Technology 的团队中出任首席技术官。Andy 之前是 Oracle Corporation 的技术总监，他在科技前沿有着超过 18 年的工作经验。在 Oracle 的时候，Andy 领导 Oracle Complex Event Processing (OCEP) 的开发，并推进国际化产品策略和创新。在 Oracle 之前，Andy 是 BEA Systems 的 WebLogic Server Core 的架构师，负责中间件基础架构技术。

查看英文原文：[Virtual Panel: Using Java in Low Latency Environments](#)

查看原文：[虚拟研讨会：在低延迟环境中使用 Java](#)

相关内容

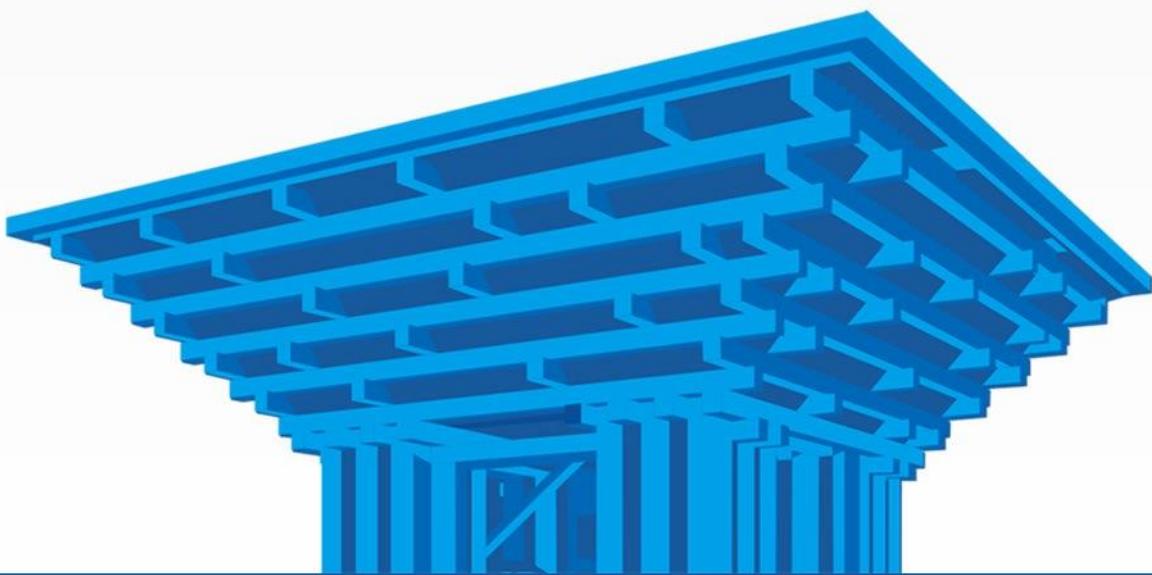
- [实现高性能 Java 解析器](#)
- [Java 里快如闪电的线程间通讯](#)
- [实现 Java 中的高性能解析器](#)
- [YourKit 发布了 Java Profiler 2013](#)
- [关于 Java 性能的 9 个谬论](#)

QCon 全球软件开发大会

2014年10月16-18日 上海光大会议中心国际大酒店
上海站 2014

这里都是干货！

- | | |
|-------------------------------|-----------------------|
| 01 知名移动应用案例分析 | 09 自动化运维 |
| 02 真实的云计算应用实践 | 10 移动开发中的痛点 |
| 03 大数据架构和行业应用 | 11 互联网思维对金融的挑战 |
| 04 扩展性、可用性、高性能 | 12 如何构建工程师文化团队 |
| 05 不仅仅是Java | 13 精益创业/创新 |
| 06 Hadoop, 超越MapReduce | 14 隐私和安全性 |
| 07 从技术到产品的不归路 | 15 智能硬件 |
| 08 没有后端 | |



专题：Netty 之道

Netty 是一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输的支持，作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果。

本专题是 InfoQ 中文站 Netty 系列文章的前两篇，分别是：

- Netty 高性能之道
- Netty 可靠性分析

作者介绍

李林锋，2007 年毕业于东北大学，2008 年进入华为公司从事高性能通信软件的设计和开发工作，有 6 年 NIO 设计和开发经验，精通 Netty、Mina 等 NIO 框架。Netty 中国社区创始人，《Netty 权威指南》作者。

联系方式：新浪微博 Nettying 微信：Nettying

专题 | Topic

Netty 系列之 Netty 高性能之道

作者 李林锋

1. 背景

1.1. 惊人的性能数据

最近一个圈内朋友通过私信告诉我，通过使用 Netty4 + Thrift 压缩二进制编解码技术，他们实现了 10W TPS（1K 的复杂 POJO 对象）的跨节点远程服务调用。相比于传统基于 Java 序列化+BIO（同步阻塞 IO）的通信框架，性能提升了 8 倍多。

事实上，我对这个数据并不感到惊讶，根据我 5 年多的 NIO 编程经验，通过选择合适的 NIO 框架，加上高性能的压缩二进制编解码技术，精心的设计 Reactor 线程模型，达到上述性能指标是完全有可能的。

下面我们就一起来看下 Netty 是如何支持 10W TPS 的跨节点远程服务调用的，在正式开始讲解之前，我们先简单介绍下 Netty。

1.2. Netty 基础入门

Netty 是一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输的支持，作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果。

作为当前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，一些业界著名的开源组件也基于 Netty 的 NIO 框架构建。

2. Netty 高性能之道

2.1. RPC 调用的性能模型分析

2.1.1. 传统 RPC 调用性能差的三宗罪

网络传输方式问题：传统的 RPC 框架或者基于 RMI 等方式的远程服务（过程）调用

采用了同步阻塞 IO，当客户端的并发压力或者网络时延增大之后，同步阻塞 IO 会由于频繁的 wait 导致 IO 线程经常性的阻塞，由于线程无法高效的工作，IO 处理能力自然下降。

下面，我们通过 BIO 通信模型图看下 BIO 通信的弊端：

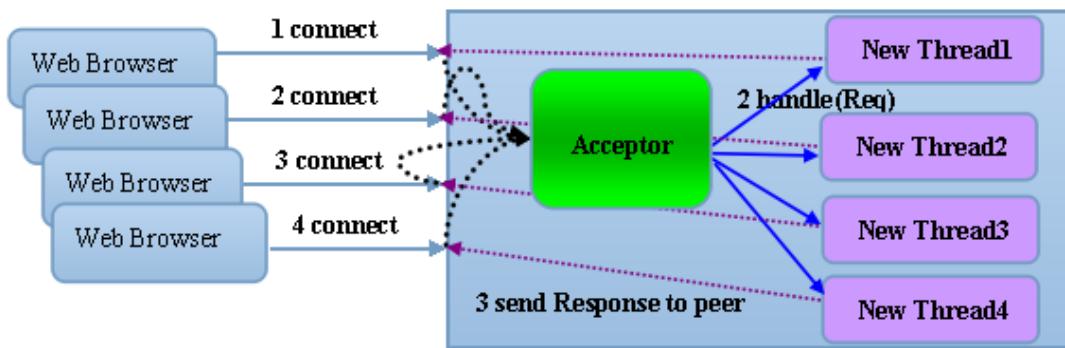


图 2-1 BIO 通信模型图

采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，接收到客户端连接之后为客户端连接创建一个新的线程处理请求消息，处理完成之后，返回应答消息给客户端，线程销毁，这就是典型的一请求一应答模型。该架构最大的问题就是不具备弹性伸缩能力，当并发访问量增加后，服务端的线程个数和并发访问数成线性正比，由于线程是 JAVA 虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能急剧下降，随着并发量的继续增加，可能会发生句柄溢出、线程堆栈溢出等问题，并导致服务器最终宕机。

序列化方式问题：Java 序列化存在如下几个典型问题：

- 1) Java 序列化机制是 Java 内部的一种对象编解码技术，无法跨语言使用；例如对于异构系统之间的对接，Java 序列化后的码流需要能够通过其它语言反序列化成原始对象（副本），目前很难支持；
- 2) 相比于其它开源的序列化框架，Java 序列化后的码流太大，无论是网络传输还是持久化到磁盘，都会导致额外的资源占用；
- 3) 序列化性能差（CPU 资源占用高）。

线程模型问题：由于采用同步阻塞 IO，这会导致每个 TCP 连接都占用 1 个线程，由

于线程资源是 JVM 虚拟机非常宝贵的资源，当 IO 读写阻塞导致线程无法及时释放时，会导致系统性能急剧下降，严重的甚至会导致虚拟机无法创建新的线程。

2.1.2. 高性能的三个主题

- 1) 传输：用什么样的通道将数据发送给对方，BIO、NIO 或者 AIO，IO 模型在很大程度上决定了框架的性能。
- 2) 协议：采用什么样的通信协议，HTTP 或者内部私有协议。协议的选择不同，性能模型也不同。相比于公有协议，内部私有协议的性能通常可以被设计的更优。
- 3) 线程：数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，Reactor 线程模型的不同，对性能的影响也非常大。

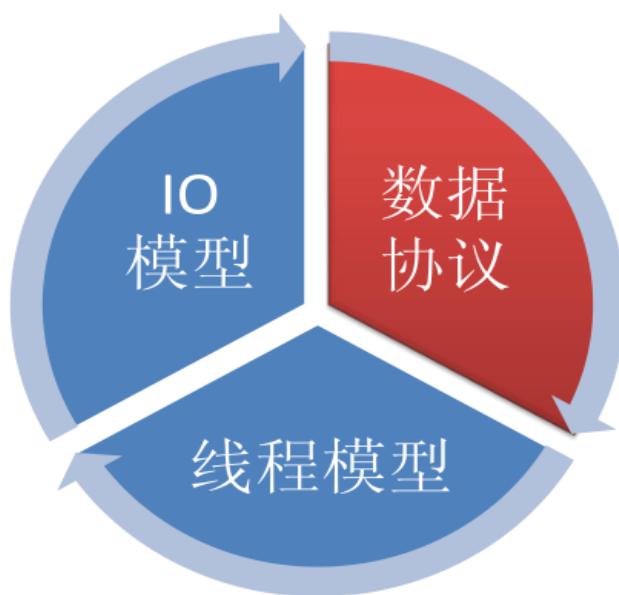


图 2-2 RPC 调用性能三要素

2.2. Netty 高性能之道

2.2.1. 异步非阻塞通信

在 IO 编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者 IO 多路复用技术进行处理。IO 多路复用技术通过把多个 IO 的阻塞复用到同一个 select 的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/

多进程模型比，I/O 多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降低了系统的维护工作量，节省了系统资源。

JDK1.4 提供了对非阻塞 IO（NIO）的支持，JDK1.5_update10 版本使用 epoll 替代了传统的 select/poll，极大的提升了 NIO 通信的性能。

JDK NIO 通信模型如下所示：

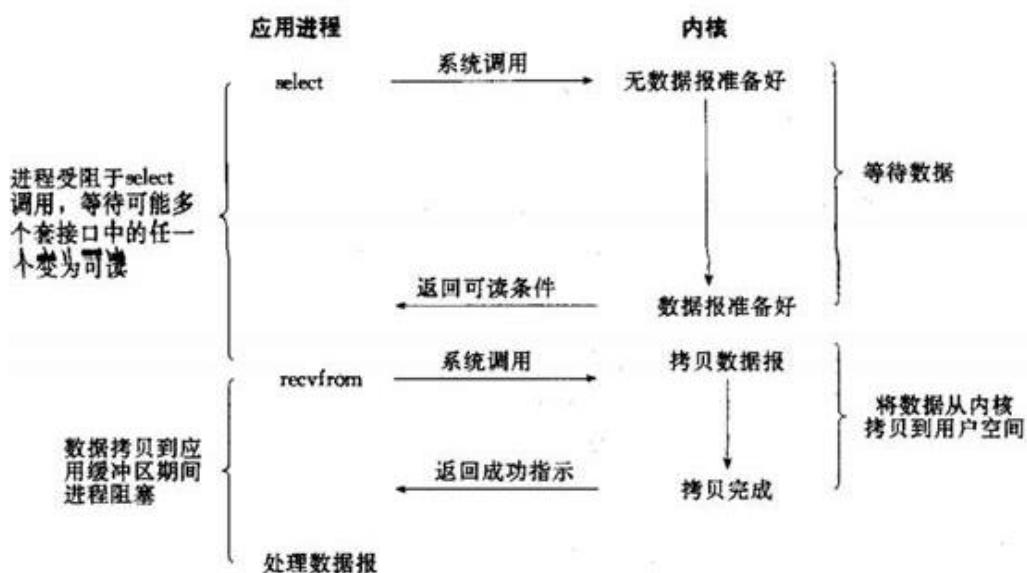


图 2-3 NIO 的多路复用模型图

与 Socket 类和 ServerSocket 类相对应，NIO 也提供了 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单，但是性能和可靠性都不好，非阻塞模式正好相反。开发人员一般可以根据自己的需要来选择合适的模式，一般来说，低负载、低并发的应用程序可以选择同步阻塞 IO 以降低编程复杂度。但是对于高负载、高并发的网络应用，需要使用 NIO 的非阻塞模式进行开发。

Netty 架构按照 Reactor 模式设计和实现，它的服务端通信序列图如下：

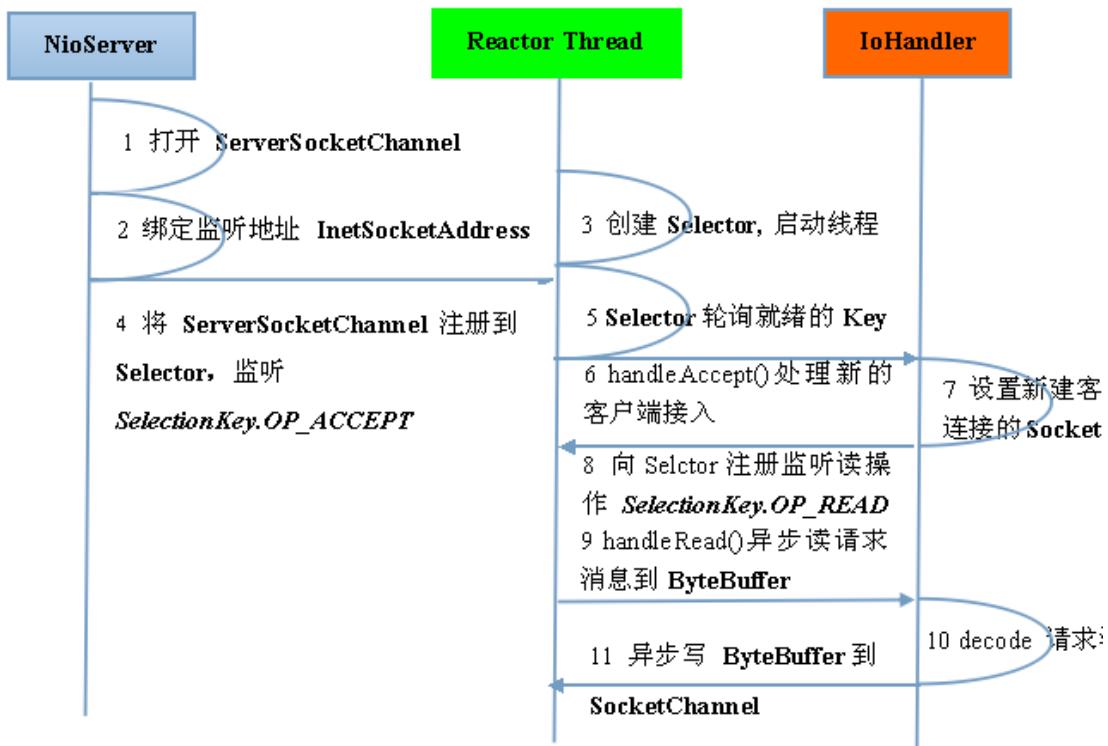


图 2-3 NIO 服务端通信序列图

客户端通信序列图如下：



图 2-4 NIO 客户端通信序列图

Netty 的 IO 线程 NioEventLoop 由于聚合了多路复用器 Selector，可以同时并发处理成百上千个客户端 Channel，由于读写操作都是非阻塞的，这就可以充分提升 IO 线程的运行效率，避免由于频繁 IO 阻塞导致的线程挂起。另外，由于 Netty 采用了异步通信模式，一个 IO 线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 IO 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

2.2.2. 零拷贝

很多用户都听说过 Netty 具有“零拷贝”功能，但是具体体现在哪里又说不清楚，本小节就详细对 Netty 的“零拷贝”功能进行讲解。

Netty 的“零拷贝”主要体现在如下三个方面：

- 1) Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存(HEAP BUFFERS)进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。
- 2) Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。
- 3) Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

下面，我们对上述三种“零拷贝”进行说明，先看 Netty 接收 Buffer 的创建：

```

@Override
public void read() {
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final int maxMessagesPerRead = config.getMaxMessagesPerRead();
    RecvByteBufAllocator.Handle allocHandle = this.allocHandle;
    if (allocHandle == null) {
        this.allocHandle = allocHandle = config.getRecvByteBufAllocator().newHandle();
    }
    if (!config.isAutoRead()) {
        removeReadOp();
    }

    ByteBuf byteBuf = null;
    int messages = 0;
    boolean close = false;
    try {
        int byteBufCapacity = allocHandle.guess();
        int totalReadAmount = 0;
        do {
            byteBuf = allocator.ioBuffer(byteBufCapacity);

```

图 2-5 异步消息读取“零拷贝”

每循环读取一次消息，就通过 ByteBufAllocator 的 ioBuffer 方法获取 ByteBuf 对象，下面继续看它的接口定义：

```

● ByteBuf io.netty.buffer.ByteBufAllocator.ioBuffer(int initialCapacity)
Allocate a ByteBuf, preferably a direct buffer which is suitable for I/O.

Parameters:
    initialCapacity

```

图 2-6 ByteBufAllocator 通过 ioBuffer 分配堆外内存

当进行 Socket IO 读写的时候，为了避免从堆内存拷贝一份副本到直接内存，Netty 的 ByteBuf 分配器直接创建非堆内存避免缓冲区的二次拷贝，通过“零拷贝”来提升读写性能。

下面我们继续看第二种“零拷贝”的实现 CompositeByteBuf，它对外将多个 ByteBuf 封装成一个 ByteBuf，对外提供统一封装后的 ByteBuf 接口，它的类定义如下：

```

Type hierarchy of 'io.netty.buffer.CompositeByteBuf' :

Object - java.lang
└─ ByteBuf - io.netty.buffer
   └─ AbstractByteBuf - io.netty.buffer
      └─ AbstractReferenceCountedByteBuf - io.netty.buffer
         └─ CompositeByteBuf - io.netty.buffer

```

图 2-7 CompositeByteBuf 类继承关系

通过继承关系我们可以看出 CompositeByteBuf 实际就是个 ByteBuf 的包装器，它将多个 ByteBuf 组合成一个集合，然后对外提供统一的 ByteBuf 接口，相关定义如下：

```
public class CompositeByteBuf extends AbstractReferenceCountedByteBuf {

    private final ResourceLeak leak;
    private final ByteBufAllocator alloc;
    private final boolean direct;
    private final List<Component> components = new ArrayList<Component>();
    private final int maxNumComponents;
    private static final ByteBuffer FULL_BYTEBUFFER = (ByteBuffer) ByteBuffer.allocate(1).position(1);

    private boolean freed;
```

图 2-8 CompositeByteBuf 类定义

添加 ByteBuf，不需要做内存拷贝，相关代码如下：

```
private int addComponent0(int cIndex, ByteBuf buffer) {
    checkComponentIndex(cIndex);

    if (buffer == null) {
        throw new NullPointerException("buffer");
    }

    int readableBytes = buffer.readableBytes();
    if (readableBytes == 0) {
        return cIndex;
    }

    // No need to consolidate - just add a component to the list.
    Component c = new Component(buffer.order(ByteOrder.BIG_ENDIAN).slice());
    if (cIndex == components.size()) {
        components.add(c);
        if (cIndex == 0) {
            c.endOffset = readableBytes;
        } else {
            Component prev = components.get(cIndex - 1);
            c.offset = prev.endOffset;
            c.endOffset = c.offset + readableBytes;
        }
    } else {
        components.add(cIndex, c);
        updateComponentOffsets(cIndex);
    }
    return cIndex;
```

图 2-9 新增 ByteBuf 的“零拷贝”

最后，我们看下文件传输的“零拷贝”：

```

@Override
public long transferTo(WritableByteChannel target, long position) throws IOException {
    long count = this.count - position;
    if (count < 0 || position < 0) {
        throw new IllegalArgumentException(
            "position out of range: " + position +
            " (expected: 0 - " + (this.count - 1) + ')');
    }
    if (count == 0) {
        return 0L;
    }

    long written = file.transferTo(this.position + position, count, target);
    if (written > 0) {
        transferred += written;
    }
    return written;
}

```

图 2-10 文件传输“零拷贝”

Netty 文件传输 DefaultFileRegion 通过 transferTo 方法将文件发送到目标 Channel 中，下面重点看 FileChannel 的 transferTo 方法，它的 API DOC 说明如下：

```

long java.nio.channels.FileChannel.transferTo(long position, long count,
                                              WritableByteChannel target) throws IOException

```

Transfers bytes from this channel's file to the given writable byte channel.

An attempt is made to read up to `count` bytes starting at the given `position` in this channel's file and write them to the target channel. An invocation of this method may or may not transfer all of the requested bytes; whether or not it does so depends upon the natures and states of the channels. Fewer than the requested number of bytes are transferred if this channel's file contains fewer than `count` bytes starting at the given `position`, or if the target channel is non-blocking and it has fewer than `count` bytes free in its output buffer.

This method does not modify this channel's position. If the given position is greater than the file's current size then no bytes are transferred. If the target channel has a position, then bytes are written starting at that position and then the position is incremented by the number of bytes written.

This method is potentially much more efficient than a simple loop that reads from this channel and writes to the target channel. Many operating systems can transfer bytes directly from the filesystem cache to the target channel without actually copying them.

图 2-11 文件传输“零拷贝”

对于很多操作系统它直接将文件缓冲区的内容发送到目标 Channel 中，而不需要通过拷贝的方式，这是一种更加高效的传输方式，它实现了文件传输的“零拷贝”。

2.2.3. 内存池

随着 JVM 虚拟机和 JIT 即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓冲区 Buffer，情况却稍有不同，特别是对于堆外直接内存的分配和回

收，是一件耗时的操作。为了尽量重用缓冲区，Netty 提供了基于内存池的缓冲区重用机制。下面我们一起看下 Netty ByteBuf 的实现：

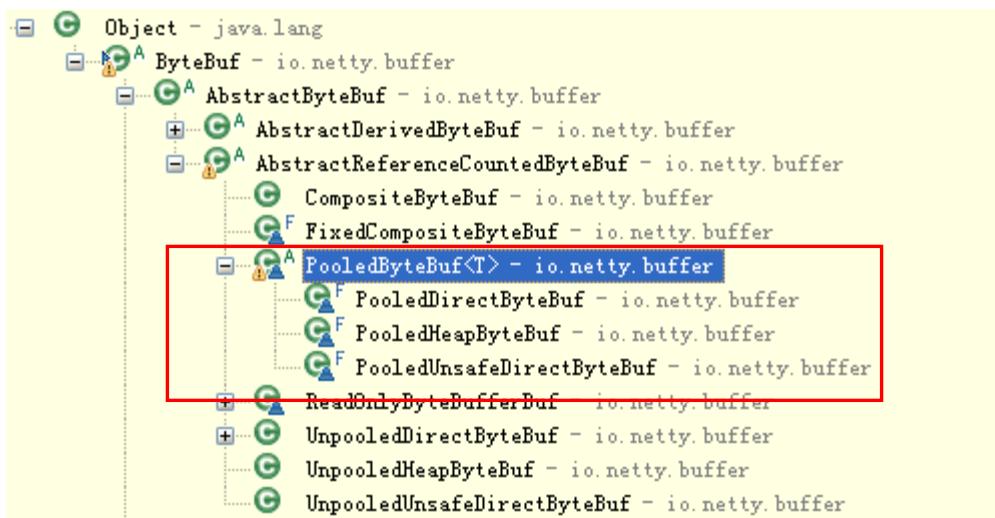


图 2-12 内存池 ByteBuf

Netty 提供了多种内存管理策略，通过在启动辅助类中配置相关参数，可以实现差异化的定制。

下面通过性能测试，我们看下基于内存池循环利用的 ByteBuf 和普通 ByteBuf 的性能差异。

用例一，使用内存池分配器创建直接内存缓冲区：

```

int loop = 3000000;
long startTime = System.currentTimeMillis();
ByteBuf poolBuffer = null;
for (int i = 0; i < loop; i++) {
    poolBuffer = PooledByteBufAllocator.DEFAULT.directBuffer(1024);
    poolBuffer.writeBytes(CONTENT);
    poolBuffer.release();
}
  
```

图 2-13 基于内存池的非堆内存缓冲区测试用例

用例二，使用非堆内存分配器创建的直接内存缓冲区：

```

long startTime2 = System.currentTimeMillis();
ByteBuf buffer = null;
for (int i = 0; i < loop; i++) {
    buffer = Unpooled.directBuffer(1024);
    buffer.writeBytes(CONTENT);
}

```

图 2-14 基于非内存池创建的非堆内存缓冲区测试用例

各执行 300 万次，性能对比结果如下所示：

```

The PooledByteBuf execute 300W times writing operation cost time is : 4125 ms
=====
The unPooledByteBuf execute 300W times writing operation cost time is : 95312 ms

```

图 2-15 内存池和非内存池缓冲区写入性能对比

性能测试表明，采用内存池的 ByteBuf 相比于朝生夕灭的 ByteBuf，性能高 23 倍左右（性能数据与使用场景强相关）。

下面我们一起简单分析下 Netty 内存池的内存分配：

```

@Override
public ByteBuf directBuffer(int initialCapacity, int maxCapacity) {
    if (initialCapacity == 0 && maxCapacity == 0) {
        return emptyBuf;
    }
    validate(initialCapacity, maxCapacity);
    return newDirectBuffer(initialCapacity, maxCapacity);
}

```

图 2-16 AbstractByteBufAllocator 的缓冲区分配

继续看 newDirectBuffer 方法，我们发现它是一个抽象方法，由 AbstractByteBufAllocator 的子类负责具体实现，代码如下：

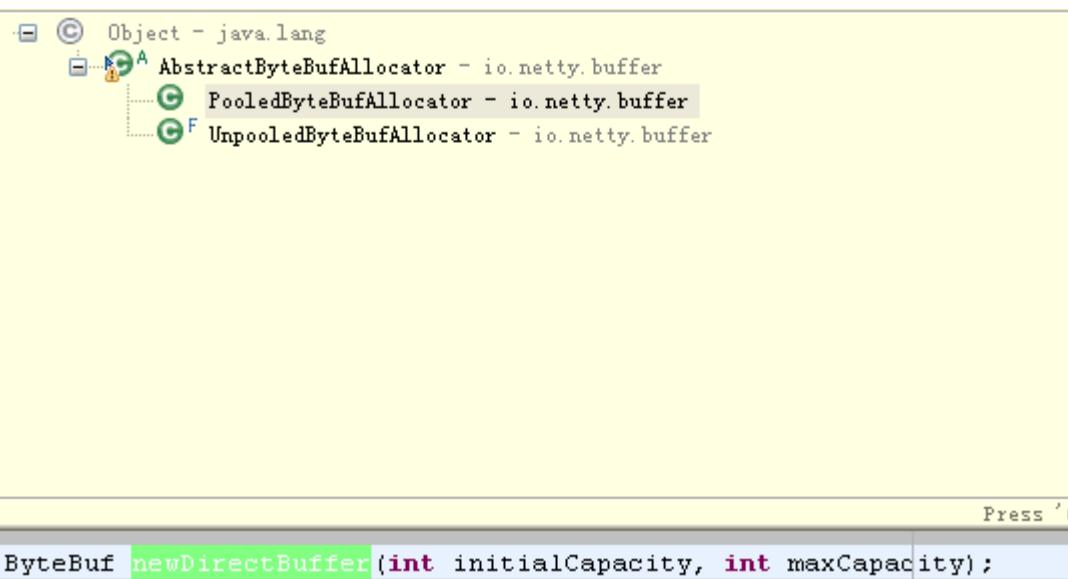


图 2-17 newDirectBuffer 的不同实现

代码跳转到 PooledByteBufAllocator 的 newDirectBuffer 方法，从 Cache 中获取内存区域 PoolArena，调用它的 allocate 方法进行内存分配：

```

@Override
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    PoolThreadCache cache = threadCache.get();
    PoolArena<ByteBuffer> directArena = cache.directArena;

    ByteBuf buf;
    if (directArena != null) {
        buf = directArena.allocate(cache, initialCapacity, maxCapacity);
    } else {
        if (PlatformDependent.hasUnsafe()) {
            buf = new UnpooledUnsafeDirectByteBuf(this, initialCapacity, maxCapacity);
        } else {
            buf = new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);
        }
    }

    return toLeakAwareBuffer(buf);
}
  
```

图 2-18 PooledByteBufAllocator 的内存分配

PoolArena 的 allocate 方法如下：

```

PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity,
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);
    allocate(cache, buf, reqCapacity);
    return buf;
)
  
```

图 2-18 PoolArena 的缓冲区分配

我们重点分析 newByteBuf 的实现，它同样是个抽象方法，由子类 DirectArena 和 HeapArena 来实现不同类型的缓冲区分配，由于测试用例使用的是堆外内存，

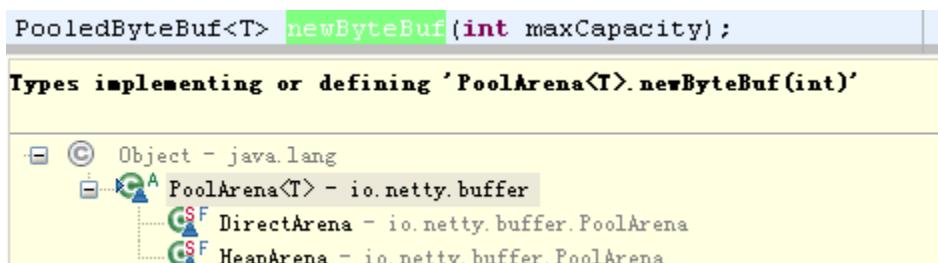


图 2-19 PoolArena 的 newByteBuf 抽象方法

因此重点分析 DirectArena 的实现：如果没有开启使用 sun 的 unsafe，则

```
@Override
protected PooledByteBuf<ByteBuffer> newByteBuf(int maxCapacity) {
    if (HAS_UNSAFE) {
        return PooledUnsafeDirectBuf.newInstance(maxCapacity);
    } else {
        return PooledDirectBuf.newInstance(maxCapacity);
    }
}
```

图 2-20 DirectArena 的 newByteBuf 方法实现

执行 PooledDirectByteBuf 的 newInstance 方法，代码如下：

```
static PooledDirectByteBuf newInstance(int maxCapacity) {
    PooledDirectByteBuf buf = RECYCLER.get();
    buf.setRefCnt(1);
    buf.maxCapacity(maxCapacity);
    return buf;
}
```

图 2-21 PooledDirectByteBuf 的 newInstance 方法实现

通过 RECYCLER 的 get 方法循环使用 ByteBuf 对象，如果是非内存池实现，则直接创建一个新的 ByteBuf 对象。从缓冲池中获取 ByteBuf 之后，调用 AbstractReferenceCountedByteBuf 的 setRefCnt 方法设置引用计数器，用于对象的引用计

数和内存回收（类似 JVM 垃圾回收机制）。

2.2.4. 高效的 Reactor 线程模型

常用的 Reactor 线程模型有三种，分别如下：

- 1) Reactor 单线程模型；
- 2) Reactor 多线程模型；
- 3) 主从 Reactor 多线程模型

Reactor 单线程模型，指的是所有的 IO 操作都在同一个 NIO 线程上面完成，NIO 线程的职责如下：

- 1) 作为 NIO 服务端，接收客户端的 TCP 连接；
- 2) 作为 NIO 客户端，向服务端发起 TCP 连接；
- 3) 读取通信对端的请求或者应答消息；
- 4) 向通信对端发送消息请求或者应答消息。

Reactor 单线程模型示意图如下所示：

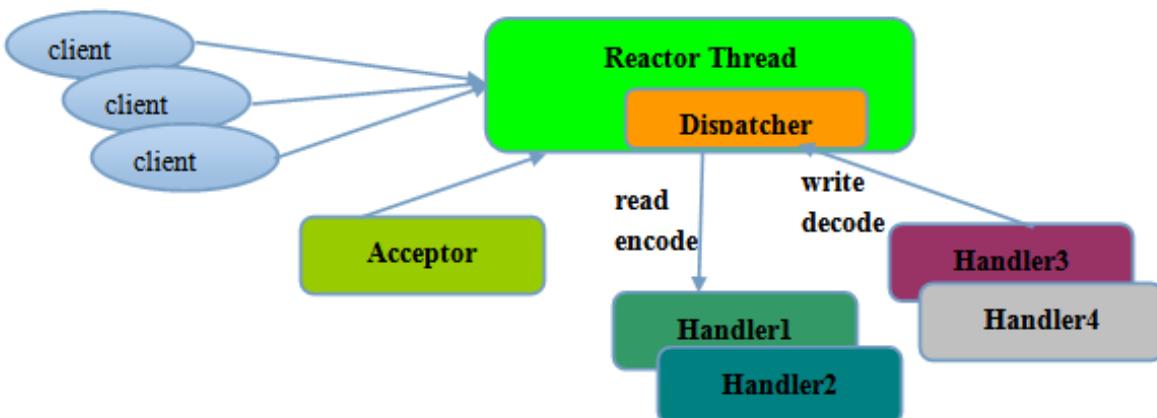


图 2-22 Reactor 单线程模型

由于 Reactor 模式使用的是异步非阻塞 IO，所有的 IO 操作都不会导致阻塞，理论上一个线程可以独立处理所有 IO 相关的操作。从架构层面看，一个 NIO 线程确实可以完成其承担的职责。例如，通过 Acceptor 接收客户端的 TCP 连接请求消息，链路建立成功之

后，通过 Dispatch 将对应的 ByteBuffer 派发到指定的 Handler 上进行消息解码。用户 Handler 可以通过 NIO 线程将消息发送给客户端。

对于一些小容量应用场景，可以使用单线程模型。但是对于高负载、大并发的应用却不合适，主要原因如下：

- 1) 一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的编码、解码、读取和发送；
- 2) 当 NIO 线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往进行重发，这更加重了 NIO 线程的负载，最终会导致大量消息积压和处理超时，NIO 线程会成为系统的性能瓶颈；
- 3) 可靠性问题：一旦 NIO 线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

为了解决这些问题，演进出了 Reactor 多线程模型，下面我们一起学习下 Reactor 多线程模型。

Reactor 多线程模型与单线程模型最大的区别就是有一组 NIO 线程处理 IO 操作，它的原理图如下：

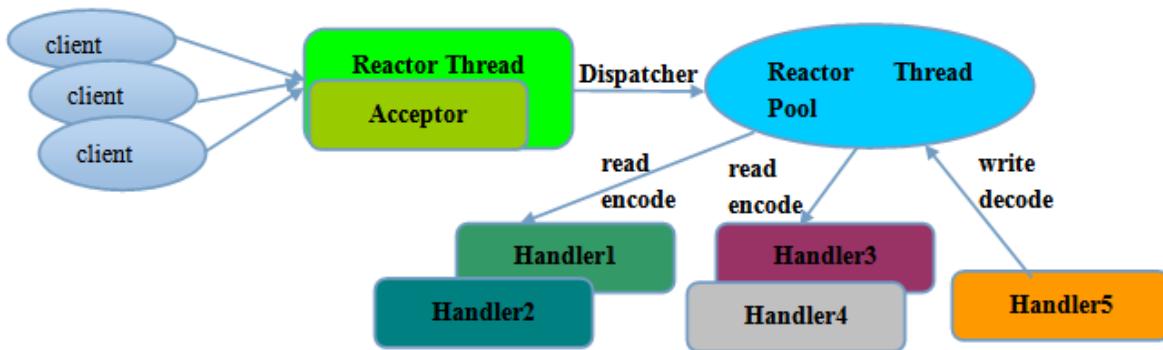


图 2-23 Reactor 多线程模型

Reactor 多线程模型的特点：

- 1) 有专门一个 NIO 线程-Acceptor 线程用于监听服务端，接收客户端的 TCP 连接请求；
- 2) 网络 IO 操作-读、写等由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线程池

实现，它包含一个任务队列和 N 个可用的线程，由这些 NIO 线程负责消息的读取、解码、编码和发送；

- 3) 1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，防止发生并发操作问题。

在绝大多数场景下，Reactor 多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个 NIO 线程负责监听和处理所有的客户端连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手消息进行安全认证，认证本身非常损耗性能。在这类场景下，单独一个 Acceptor 线程可能会存在性能不足问题，为了解决性能问题，产生了第三种 Reactor 线程模型-主从 Reactor 多线程模型。

主从 Reactor 线程模型的特点是：服务端用于接收客户端连接的不再是个 1 个单独的 NIO 线程，而是一个独立的 NIO 线程池。Acceptor 接收到客户端 TCP 连接请求处理完成后（可能包含接入认证等），将新创建的 SocketChannel 注册到 IO 线程池（sub reactor 线程池）的某个 IO 线程上，由它负责 SocketChannel 的读写和编解码工作。Acceptor 线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端 subReactor 线程池的 IO 线程上，由 IO 线程负责后续的 IO 操作。

它的线程模型如下图所示：

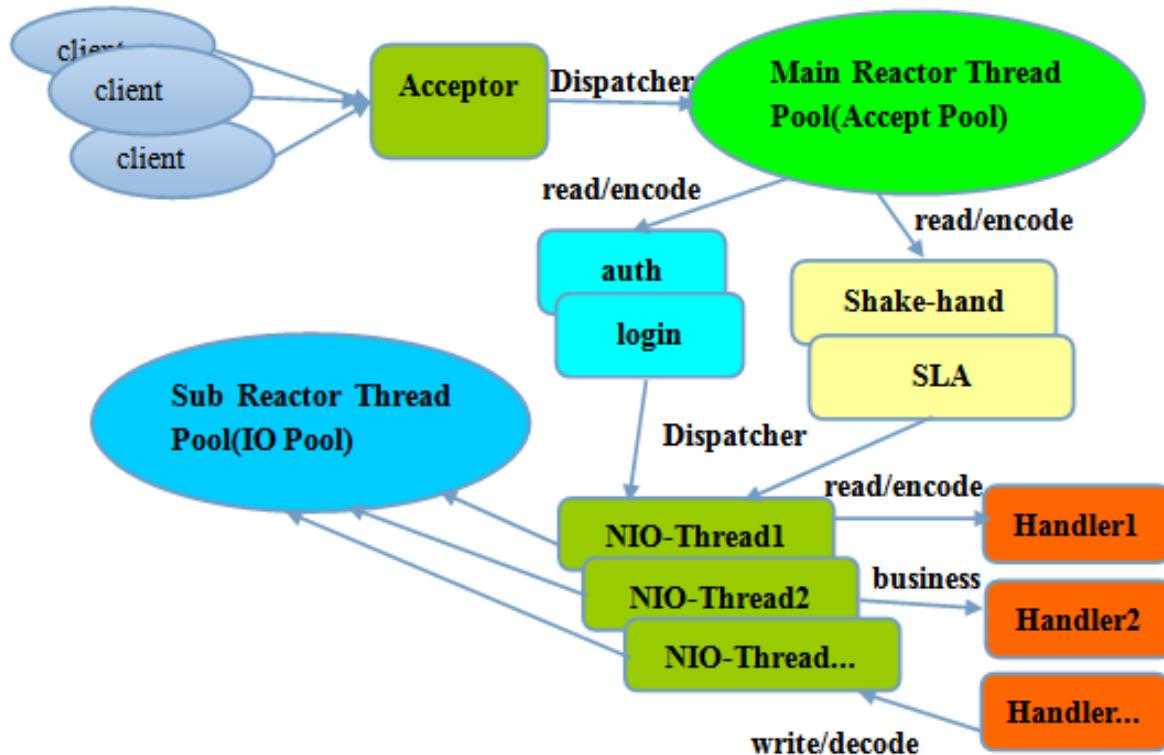


图 2-24 Reactor 主从多线程模型

利用主从 NIO 线程模型，可以解决 1 个服务端监听线程无法有效处理所有客户端连接的性能不足问题。因此，在 Netty 的官方 demo 中，推荐使用该线程模型。

事实上，Netty 的线程模型并非固定不变，通过在启动辅助类中创建不同的 EventLoopGroup 实例并通过适当的参数配置，就可以支持上述三种 Reactor 线程模型。正是因为 Netty 对 Reactor 线程模型的支持提供了灵活的定制能力，所以可以满足不同业务场景的性能诉求。

2.2.5. 无锁化的串行设计理念

在大多数场景下，并行多线程处理可以提升系统的并发性能。但是，如果对于共享资源的并发访问处理不当，会带来严重的锁竞争，这最终会导致性能的下降。为了尽可能的避免锁竞争带来的性能损耗，可以通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。

为了尽可能提升性能，Netty 采用了串行无锁化设计，在 IO 线程内部进行串行操作，

避免多线程竞争导致的性能下降。表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

Netty 的串行化设计工作原理图如下：

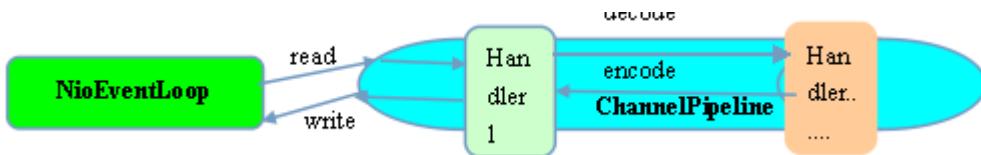


图 2-25 Netty 串行化工作原理图

Netty 的 NioEventLoop 读取到消息之后，直接调用 ChannelPipeline 的 fireChannelRead(Object msg)，只要用户不主动切换线程，一直会由 NioEventLoop 调用到用户的 Handler，期间不进行线程切换，这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

2.2.6. 高效的并发编程

Netty 的高效并发编程主要体现在如下几点：

- 1) volatile 的大量、正确使用；
- 2) CAS 和原子类的广泛使用；
- 3) 线程安全容器的使用；
- 4) 通过读写锁提升并发性能。

如果大家想了解 Netty 高效并发编程的细节，可以阅读之前我在微博分享的《多线程并发编程在 Netty 中的应用分析》，在这篇文章中对 Netty 的多线程技巧和应用进行了详细的介绍和分析。

2.2.7. 高性能的序列化框架

影响序列化性能的关键因素总结如下：

- 1) 序列化后的码流大小（网络带宽的占用）；
- 2) 序列化&反序列化的性能（CPU 资源占用）；
- 3) 是否支持跨语言（异构系统的对接和开发语言切换）。

Netty 默认提供了对 Google Protobuf 的支持，通过扩展 Netty 的编解码接口，用户可以实现其它的高性能序列化框架，例如 Thrift 的压缩二进制编解码框架。

下面我们一起看下不同序列化&反序列化框架序列化后的字节数组对比：

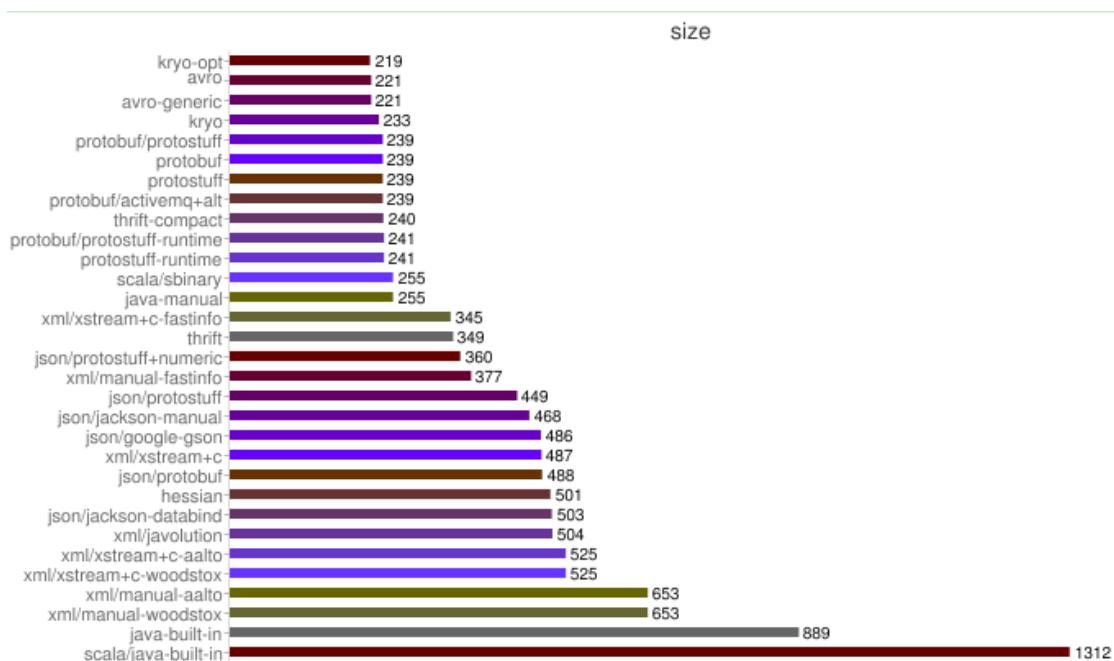


图 2-26 各序列化框架序列化码流大小对比

从上图可以看出，Protobuf 序列化后的码流只有 Java 序列化的 1/4 左右。正是由于 Java 原生序列化性能表现太差，才催生出了各种高性能的开源序列化技术和框架（性能差只是其中的一个原因，还有跨语言、IDL 定义等其它因素）。

2.2.8. 灵活的 TCP 参数配置能力

合理设置 TCP 参数在某些场景下对于性能的提升可以起到显著的效果，例如 SO_RCVBUF 和 SO_SNDBUF。如果设置不当，对性能的影响是非常大的。下面我们总结下对性能影响比较大的几个配置项：

- 1) SO_RCVBUF 和 SO_SNDBUF: 通常建议值为 128K 或者 256K;
- 2) SO_TCPNODELAY: Nagle 算法通过将缓冲区内的小封包自动相连, 组成较大的封包, 阻止大量小封包的发送阻塞网络, 从而提高网络应用效率。但是对于时延敏感的应用场景需要关闭该优化算法;
- 3) 软中断: 如果 Linux 内核版本支持 RPS (2.6.35 以上版本), 开启 RPS 后可以实现软中断, 提升网络吞吐量。RPS 根据数据包的源地址, 目的地址以及目的和源端口, 计算出一个 hash 值, 然后根据这个 hash 值来选择软中断运行的 cpu, 从上层来看, 也就是说将每个连接和 cpu 绑定, 并通过这个 hash 值, 来均衡软中断在多个 cpu 上, 提升网络并行处理性能。

Netty 在启动辅助类中可以灵活的配置 TCP 参数, 满足不同的用户场景。相关配置接口定义如下:

```

SF ALLOCATOR : ChannelOption<ByteBufAllocator>
SF RCVBUF_ALLOCATOR : ChannelOption<RecvByteBufAllocator>
SF MESSAGE_SIZE_ESTIMATOR : ChannelOption<MessageSizeEstimator>
SF CONNECT_TIMEOUT_MILLIS : ChannelOption<Integer>
SF MAX_MESSAGES_PER_READ : ChannelOption<Integer>
SF WRITE_SPIN_COUNT : ChannelOption<Integer>
SF WRITE_BUFFER_HIGH_WATER_MARK : ChannelOption<Integer>
SF WRITE_BUFFER_LOW_WATER_MARK : ChannelOption<Integer>
SF ALLOW_HALF_CLOSURE : ChannelOption<Boolean>
SF AUTO_READ : ChannelOption<Boolean>
SF SO_BROADCAST : ChannelOption<Boolean>
SF SO_KEEPALIVE : ChannelOption<Boolean>
SF SO_SNDBUF : ChannelOption<Integer>
SF SO_RCVBUF : ChannelOption<Integer>
SF SO_REUSEADDR : ChannelOption<Boolean>
SF SO_LINGER : ChannelOption<Integer>
SF SO_BACKLOG : ChannelOption<Integer>
SF SO_TIMEOUT : ChannelOption<Integer>
SF IP_TOS : ChannelOption<Integer>
SF IP_MULTICAST_ADDR : ChannelOption<InetAddress>
SF IP_MULTICAST_IF : ChannelOption<NetworkInterface>
SF IP_MULTICAST_TTL : ChannelOption<Integer>
SF IP_MULTICAST_LOOP_DISABLED : ChannelOption<Boolean>
SF TCP_NODELAY : ChannelOption<Boolean>

```

图 2-27 Netty 的 TCP 参数配置定义

2.3. 总结

通过对 Netty 的架构和性能模型进行分析，我们发现 Netty 架构的高性能是被精心设计和实现的，得益于高质量的架构和代码，Netty 支持 10W TPS 的跨节点服务调用并不是一件十分困难的事情。

感谢[张龙](#)对本文的审校，[郭蕾](#)对本文的策划。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至 editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)）或者腾讯微博（[@InfoQ](#)）关注我们，并与我们的编辑和其他读者朋友交流。

查看原文：[Netty 系列之 Netty 高性能之道](#)

Netty 系列之 Netty 可靠性分析

作者 李林锋

1. 背景

1.1. 宕机的代价

1.1.1. 电信行业

毕马威国际(KPMG International)在对 46 个国家的 74 家运营商进行调查后发现，全球通信行业每年的收益流失约为 400 亿美元，占总收入的 1%-3%。导致收益流失的因素有多种，主要原因就是计费 BUG。

1.1.2. 互联网行业

美国太平洋时间 8 月 16 日下午 3 点 50 分到 3 点 55 分（北京时间 8 月 17 日 6 点 50 分到 6 点 55 分），谷歌遭遇了宕机。根据事后统计，短短的 5 分钟，谷歌损失了 54.5 万美元。也就是服务每中断一分钟，损失就达 10.8 万美元。

2013 年，从美国东部时间 8 月 19 日下午 2 点 45 分开始，有用户率先发现了亚马逊网站出现宕机，大约在 20 多分钟后又恢复正常。此次宕机让亚马逊每分钟损失近 6.7 万美元，在宕机期间，消费者无法通过 Amazon.com、亚马逊移动端以及 Amazon.ca 等网站进行购物。

1.2. 软件可靠性

软件可靠性是指在给定时间内，特定环境下软件无错运行的概率。软件可靠性包含了以下三个要素：

- 1) 规定的时间：软件可靠性只是体现在其运行阶段，所以将运行时间作为规定的时间的度量。运行时间包括软件系统运行后工作与挂起(开启但空闲)的累计时间。由于软件运行的环境与程序路径选取的随机性，软件的失效为随机事件，所以运行时间属于随

机变量;

- 2) 规定的环境条件:环境条件指软件的运行环境。它涉及软件系统运行时所需的各种支持要素,如支持硬件、操作系统、其它支持软件、输入数据格式和范围以及操作规程等。不同的环境条件下软件的可靠性是不同的。具体地说,规定的环境条件主要是描述软件系统运行时计算机的配置情况以及对输入数据的要求,并假定其它一切因素都是理想的。有了明确规定了的环境条件,还可以有效判断软件失效的责任在用户方还是提供方;
- 3) 规定的功能:软件可靠性还与规定的任务和功能有关。由于要完成的任务不同,软件的运行剖面会有所区别,则调用的子模块就不同(即程序路径选择不同),其可靠性也就可能不同。所以要准确度量软件系统的可靠性必须首先明确它的任务和功能。

1.3. Netty 的可靠性

首先,我们要从 Netty 的主要用途来分析它的可靠性,Netty 目前的主流用法有三种:

- 1) 构建 RPC 调用的基础通信组件, 提供跨节点的远程服务调用能力;
- 2) NIO 通信框架, 用于跨节点的数据交换;
- 3) 其它应用协议栈的基础通信组件, 例如 HTTP 协议以及其他基于 Netty 开发的应用层协议栈。

以阿里的分布式服务框架 Dubbo 为例, Netty 是 Dubbo RPC 框架的核心。它的服务调用示例图如下:

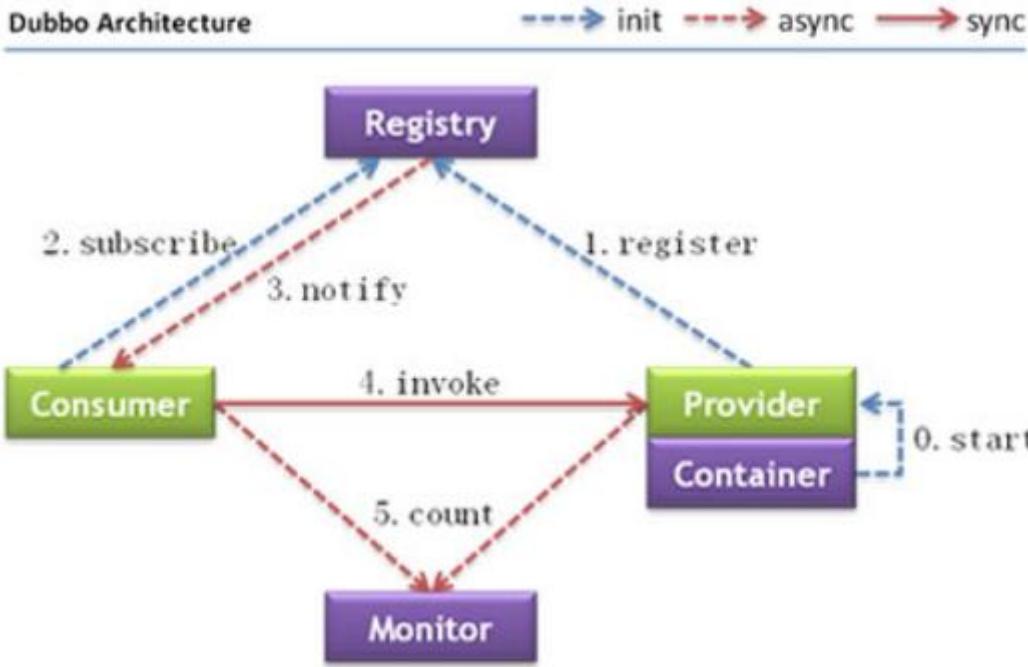


图 1-1 Dubbo 的节点角色说明图

其中，服务提供者和服务调用者之间可以通过 Dubbo 协议进行 RPC 调用，消息的收发默认通过 Netty 完成。

通过对 Netty 主流应用场景的分析，我们发现 Netty 面临的可靠性问题大致分为三类：

- 1) 传统的网络 I/O 故障，例如网络闪断、防火墙 Hang 住连接、网络超时等；
- 2) NIO 特有的故障，例如 NIO 类库特有的 BUG、读写半包处理异常、Reactor 线程跑飞等等；
- 3) 编解码相关的异常。

在大多数的业务应用场景中，一旦因为某些故障导致 Netty 不能正常工作，业务往往会陷入瘫痪。所以，从业务诉求来看，对 Netty 框架的可靠性要求是非常的高。作为当前业界最流行的一款 NIO 框架，Netty 在不同行业和领域都得到了广泛的应用，它的高可靠性已经得到了成百上千的生产系统检验。

Netty 是如何支持系统高可靠性的？下面，我们就从几个不同维度出发一探究竟。

2. Netty 高可靠性之道

2.1. 网络通信类故障

2.1.1. 客户端连接超时

在传统的同步阻塞编程模式下，客户端 Socket 发起网络连接，往往需要指定连接超时时间，这样做的目的主要有两个：

- 1) 在同步阻塞 I/O 模型中，连接操作是同步阻塞的，如果不设置超时时间，客户端 I/O 线程可能会被长时间阻塞，这会导致系统可用 I/O 线程数的减少；
- 2) 业务层需要：大多数系统都会对业务流程执行时间有限制，例如 WEB 交互类的响应时间要小于 3S。客户端设置连接超时时间是为了实现业务层的超时。

JDK 原生的 Socket 连接接口定义如下：

```
 /**
 * Connects this socket to the server with a specified timeout value.
 * A timeout of zero is interpreted as an infinite timeout. The connection
 * will then block until established or an error occurs.
 *
 * @param endpoint the <code>SocketAddress</code>
 * @param timeout the timeout value to be used in milliseconds.
 * @throws IOException if an error occurs during the connection
 * @throws SocketTimeoutException if timeout expires before connecting
 * @throws java.nio.channels.IllegalBlockingModeException
 *         if this socket has an associated channel,
 *         and the channel is in non-blocking mode
 * @throws IllegalArgumentException if endpoint is null or is a
 *         SocketAddress subclass not supported by this socket
 * @since 1.4
 * @spec JSR-51
 */
public void connect(SocketAddress endpoint, int timeout) throws IOException {
```

图 2-1 JDK Socket 连接超时接口

对于 NIO 的 SocketChannel，在非阻塞模式下，它会直接返回连接结果，如果没有连接成功，也没有发生 IO 异常，则需要将 SocketChannel 注册到 Selector 上监听连接结果。所以，异步连接的超时无法在 API 层面直接设置，而是需要通过定时器来主动监测。

下面我们首先看下 JDK NIO 类库的 SocketChannel 连接接口定义：

```

* @throws AlreadyConnectedException
*         If this channel is already connected
*
* @throws ConnectionPendingException
*         If a non-blocking connection operation is already in progress
*         on this channel
*
* @throws ClosedChannelException
*         If this channel is closed
*
* @throws AsynchronousCloseException
*         If another thread closes this channel
*         while the connect operation is in progress
*
* @throws ClosedByInterruptException
*         If another thread interrupts the current thread
*         while the connect operation is in progress, thereby
*         closing the channel and setting the current thread's
*         interrupt status
*
* @throws UnresolvedAddressException
*         If the given remote address is not fully resolved
*
* @throws UnsupportedAddressTypeException
*         If the type of the given remote address is not supported
*
* @throws SecurityException
*         If a security manager has been installed
*         and it does not permit access to the given remote endpoint
*
* @throws IOException
*         If some other I/O error occurs
*/
public abstract boolean connect(SocketAddress remote) throws IOException;

```

图 2-2 JDK NIO 类库 SocketChannel 连接接口

从上面的接口定义可以看出，NIO 类库并没有现成的连接超时接口供用户直接使用，如果要在 NIO 编程中支持连接超时，往往需要 NIO 框架或者用户自己封装实现。

下面我们看下 Netty 是如何支持连接超时的，首先，在创建 NIO 客户端的时候，可以配置连接超时参数：

```

// Configure the client.
EventLoopGroup group = new NioEventLoopGroup();
try {
    Bootstrap b = new Bootstrap();
    b.group(group)
        .channel(NioSocketChannel.class)
        .option(ChannelOption.TCP_NODELAY, true)
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 3000)
        .handler(new ChannelInitializer<SocketChannel>() {

```

图 2-3 Netty 客户端创建支持设置连接超时参数

设置完连接超时之后，Netty 在发起连接的时候，会根据超时时间创建 ScheduledFuture 挂载在 Reactor 线程上，用于定时监测是否发生连接超时，相关代码如下：

```
// Schedule connect timeout.
int connectTimeoutMillis = config().getConnectTimeoutMillis();
if (connectTimeoutMillis > 0) {
    connectTimeoutFuture = eventLoop().schedule(new Runnable() {
        @Override
        public void run() {
            ChannelPromise connectPromise = AbstractNioChannel.this.connectPromise;
            ConnectTimeoutException cause =
                new ConnectTimeoutException("connection timed out: " + remoteAddress);
            if (connectPromise != null && connectPromise.tryFailure(cause)) {
                close(voidPromise());
            }
        }
    }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
}
```

图 2-4 根据连接超时创建超时监测定时任务

创建连接超时定时任务之后，会由 NioEventLoop 负责执行。如果已经连接超时，但是服务端仍然没有返回 TCP 握手应答，则关闭连接，代码如上图所示。

如果在超时期限内处理完成连接操作，则取消连接超时定时任务，相关代码如下：

```

@Override
public void finishConnect() {
    // Note this method is invoked by the event loop only if the connect
    // neither cancelled nor timed out.

    assert eventLoop().inEventLoop();
    assert connectPromise != null;

    try {
        boolean wasActive = isActive();
        doFinishConnect();
        fulfillConnectPromise(connectPromise, wasActive);
    } catch (Throwable t) {
        if (t instanceof ConnectException) {
            Throwable newT = new ConnectException(t.getMessage() + ": "
                newT.setStackTrace(t.getStackTrace());
            t = newT;
        }

        // Use tryFailure() instead of setFailure() to avoid the race against
        connectPromise.tryFailure(t);
        closeIfClosed();
    } finally {
        // Check for null as the connectTimeoutFuture is only created if
        // See https://github.com/netty/netty/issues/1770
        if (connectTimeoutFuture != null) {
            connectTimeoutFuture.cancel(false);
        }
    }
}

```

图 2-5 取消连接超时定时任务

Netty 的客户端连接超时参数与其它常用的 TCP 参数一起配置，使用起来非常方便，上层用户不用关心底层的超时实现机制。这既满足了用户的个性化需求，又实现了故障的分层隔离。

2.1.2. 通信对端强制关闭连接

在客户端和服务端正常通信过程中，如果发生网络闪断、对方进程突然宕机或者其它非正常关闭链路事件时，TCP 链路就会发生异常。由于 TCP 是全双工的，通信双方都需要关闭和释放 Socket 句柄才不会发生句柄的泄漏。

在实际的 NIO 编程过程中，我们经常會发现由于句柄没有被及时关闭导致的功能和可靠性问题。究其原因总结如下：

- 1) IO 的读写等操作并非仅仅集中在 Reactor 线程内部，用户上层的一些定制行为可能会导致 IO 操作的外逸，例如业务自定义心跳机制。这些定制行为加大了统一异常处

理的难度，IO 操作越发散，故障发生的概率就越大；

- 2) 一些异常分支没有考虑到，由于外部环境诱因导致程序进入这些分支，就会引起故障。

下面我们通过故障模拟，看 Netty 是如何处理对端链路强制关闭异常的。首先启动 Netty 服务端和客户端，TCP 链路建立成功之后，双方维持该链路，查看链路状态，结果如下：

```
C:\Documents and Settings\Administrator>netstat -ano!find "8080"
TCP    0.0.0.0:8080          0.0.0.0:0          LISTENING      5032
TCP    127.0.0.1:3410        127.0.0.1:8080      ESTABLISHED   3848
TCP    127.0.0.1:8080        127.0.0.1:3410      ESTABLISHED   5032
```

图 2-6 Netty 服务端和客户端 TCP 链路状态正常

强制关闭客户端，模拟客户端宕机，服务端控制台打印如下异常：

```
java.io.IOException: 远程主机强迫关闭了一个现有的连接。|
  at sun.nio.ch.SocketDispatcher.read0(Native Method)
  at sun.nio.ch.SocketDispatcher.read(SocketDispatcher.java:43)
  at sun.nio.ch.IOUtil.readIntoNativeBuf(IOUtil.java:223)
  at sun.nio.ch.IOUtil.read(IOUtil.java:192)
  at sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.java:379)
  at io.netty.buffer.UnpooledUnsafeDirectByteBuf.setBytes(UnpooledUnsafeDirectByteBuf.java:446)
  at io.netty.buffer.AbstractByteBuf.writeBytes(AbstractByteBuf.java:871)
  at io.netty.channel.socket.nio.NioSocketChannel.doReadBytes(NioSocketChannel.java:208)
  at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:119)
  at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:485)
  at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:452)
  at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:346)
  at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:794)
  at java.lang.Thread.run(Thread.java:744)
```

图 2-7 模拟 TCP 链路故障

从堆栈信息可以判断，服务端已经监控到客户端强制关闭了连接，下面我们看下服务器是否已经释放了连接句柄，再次执行 netstat 命令，执行结果如下：

```
C:\Documents and Settings\Administrator>netstat -ano!find "8080"
TCP    0.0.0.0:8080          0.0.0.0:0          LISTENING      4112
```

图 2-8 查看故障链路状态

从执行结果可以看出，服务端已经关闭了和客户端的 TCP 连接，句柄资源正常释放。由此可以得出结论，Netty 底层已经自动对该故障进行了处理。

下面我们一起看下 Netty 是如何感知到链路关闭异常并进行正确处理的，查看

AbstractByteBuf 的 writeBytes 方法, 它负责将指定 Channel 的缓冲区数据写入到 ByteBuf 中, 详细代码如下:

```
@Override
public int writeBytes(ScatteringByteChannel in, int length) throws IOException {
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length);
    if (writtenBytes > 0) {
        writerIndex += writtenBytes;
    }
    return writtenBytes;
}
```

图 2-9 AbstractByteBuf 的 writeBytes 方法

在调用 SocketChannel 的 read 方法时发生了 IOException, 代码如下:

```
@Override
public int setBytes(int index, ScatteringByteChannel in, int length) throws IOException {
    ensureAccessible();
    ByteBuffer tmpBuf = internalNioBuffer();
    tmpBuf.clear().position(index).limit(index + length);
    try {
        return in.read(tmpBuf);
    } catch (ClosedChannelException e) {
        return -1;
    }
}
```

图 2-10 读取缓冲区数据发生 IO 异常

为了保证 IO 异常被统一处理, 该异常向上抛, 由 AbstractNioByteChannel 进行统一异常处理, 代码如下:

```
private void closeOnRead(ChannelPipeline pipeline) {
    SelectionKey key = selectionKey();
    setInputShutdown();
    if (isOpen()) {
        if (Boolean.TRUE.equals(config().getOption(ChannelOption.ALLOW_HALF_CLOSURE))) {
            key.interestOps(key.interestOps() & ~readInterestOp);
            pipeline.fireUserEventTriggered(ChannelInputShutdownEvent.INSTANCE);
        } else {
            close(voidPromise());
        }
    }
}
```

图 2-11 链路异常退出异常处理

为了能够对异常策略进行统一, 也为了方便维护, 防止处理不当导致的句柄泄漏等问题, 句柄的关闭, 统一调用 AbstractChannel 的 close 方法, 代码如下:

```

@Override
public ChannelFuture close(ChannelPromise promise) {
    return pipeline.close(promise);
}

```

图 2-12 统一的 Socket 句柄关闭接口

2.1.3. 正常的连接关闭

对于短连接协议，例如 HTTP 协议，通信双方数据交互完成之后，通常按照双方的约定由服务端关闭连接，客户端获得 TCP 连接关闭请求之后，关闭自身的 Socket 连接，双方正式断开连接。

在实际的 NIO 编程过程中，经常存在一种误区：认为只要是对方关闭连接，就会发生 IO 异常，捕获 IO 异常之后再关闭连接即可。实际上，连接的合法关闭不会发生 IO 异常，它是一种正常场景，如果遗漏了该场景的判断和处理就会导致连接句柄泄漏。

下面我们一起模拟故障，看 Netty 是如何处理的。测试场景设计如下：改造下 Netty 客户端，双发链路建立成功之后，等待 120S，客户端正常关闭链路。看服务端是否能够感知并释放句柄资源。

首先启动 Netty 客户端和服务端，双方 TCP 链路连接正常：

```
C:\Documents and Settings\Administrator>netstat -ano |find "8080"
TCP      0.0.0.0:8080          0.0.0.0:0          LISTENING      5032
TCP      127.0.0.1:3410        127.0.0.1:8080      ESTABLISHED   3848
TCP      127.0.0.1:8080        127.0.0.1:3410      ESTABLISHED   5032
```

图 2-13 TCP 连接状态正常

120S 之后，客户端关闭连接，进程退出，为了能够看到整个处理过程，我们在服务端的 Reactor 线程处设置断点，先不做处理，此时链路状态如下：

```
C:\Documents and Settings\Administrator>netstat -ano |find "8080"
TCP      0.0.0.0:8080          0.0.0.0:0          LISTENING      3080
TCP      127.0.0.1:8080        127.0.0.1:3870      CLOSE_WAIT    3080
```

图 2-14 TCP 连接句柄等待释放

从上图可以看出，此时服务端并没有关闭 Socket 连接，链路处于 CLOSE_WAIT 状态，

放开代码让服务端执行完，结果如下：

```
C:\Documents and Settings\Administrator>netstat -ano | find "8080"
TCP      0.0.0.0:8080          0.0.0.0:0          LISTENING      3080
```

图 2-15 TCP 连接句柄正常释放

下面我们一起看下服务端是如何判断出客户端关闭连接的，当连接被对方合法关闭后，被关闭的 SocketChannel 会处于就绪状态，SocketChannel 的 read 操作返回值为-1，说明连接已经被关闭，代码如下：

```
byteBuf = allocator.ioBuffer(byteBufCapacity);
int writable = byteBuf.writableBytes();
int localReadAmount = doReadBytes(byteBuf);
if (localReadAmount <= 0) {
    // not was read release the buffer
    byteBuf.release();
    close = localReadAmount < 0;
    break;
}
```

图 2-16 需要对读取的字节数进行判断

如果 SocketChannel 被设置为非阻塞，则它的 read 操作可能返回三个值：

- 1) 大于 0，表示读取到了字节数；
- 2) 等于 0，没有读取到消息，可能 TCP 处于 Keep-Alive 状态，接收到的是 TCP 握手消息；
- 3) -1，连接已经被对方合法关闭。

通过调试，我们发现，NIO 类库的返回值确实为-1：

```

@Override
public int writeBytes(ScatteringByteChannel in, int length)
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length);
    if (writtenBytes > 0) {
        .... ① writtenBytes= -1
    }
    return
}
@Override

```

图 2-17 链路正常关闭，返回值为-1

得知连接关闭之后，Netty 将关闭操作位设置为 true，关闭句柄，代码如下：

```

if (close) {
    closeOnRead(pipeline);
    close = false;
}

```

图 2-18 连接正常关闭，释放资源

2.1.4. 故障定制

在大多数场景下，当底层网络发生故障的时候，应该由底层的 NIO 框架负责释放资源，处理异常等。上层的业务应用不需要关心底层的处理细节。但是，在一些特殊的场景下，用户可能需要感知这些异常，并针对这些异常进行定制处理，例如：

- 1) 客户端的断连重连机制；
- 2) 消息的缓存重发；
- 3) 接口日志中详细记录故障细节；
- 4) 运维相关功能，例如告警、触发邮件/短信等

Netty 的处理策略是发生 IO 异常，底层的资源由它负责释放，同时将异常堆栈信息以事件的形式通知给上层用户，由用户对异常进行定制。这种处理机制既保证了异常处理的安全性，也向上层提供了灵活的定制能力。

具体接口定义以及默认实现如下：

```

    /**
     * Calls {@link ChannelHandlerContext#fireExceptionCaught(Throwable)} to forward
     * to the next {@link ChannelHandler} in the {@link ChannelPipeline}.
     *
     * Sub-classes may override this method to change behavior.
     */
    @Skip
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
        ctx.fireExceptionCaught(cause);
    }
}

```

图 2-19 故障定制接口

用户可以覆盖该接口，进行个性化的异常定制。例如发起重连等。

2.2. 链路的有效性检测

当网络发生单通、连接被防火墙 Hang 住、长时间 GC 或者通信线程发生非预期异常时，会导致链路不可用且不易被及时发现。特别是异常发生在凌晨业务低谷期间，当早晨业务高峰期到来时，由于链路不可用会导致瞬间的大批量业务失败或者超时，这将对系统的可靠性产生重大的威胁。

从技术层面看，要解决链路的可靠性问题，必须周期性的对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。

心跳检测机制分为三个层面：

- 1) TCP 层面的心跳检测，即 TCP 的 Keep-Alive 机制，它的作用域是整个 TCP 协议栈；
- 2) 协议层的心跳检测，主要存在于长连接协议中。例如 SMPP 协议；
- 3) 应用层的心跳检测，它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。

心跳检测的目的就是确认当前链路可用，对方活着并且能够正常接收和发送消息。

做为高可靠的 NIO 框架，Netty 也提供了心跳检测机制，下面我们一起熟悉下心跳的检测原理。

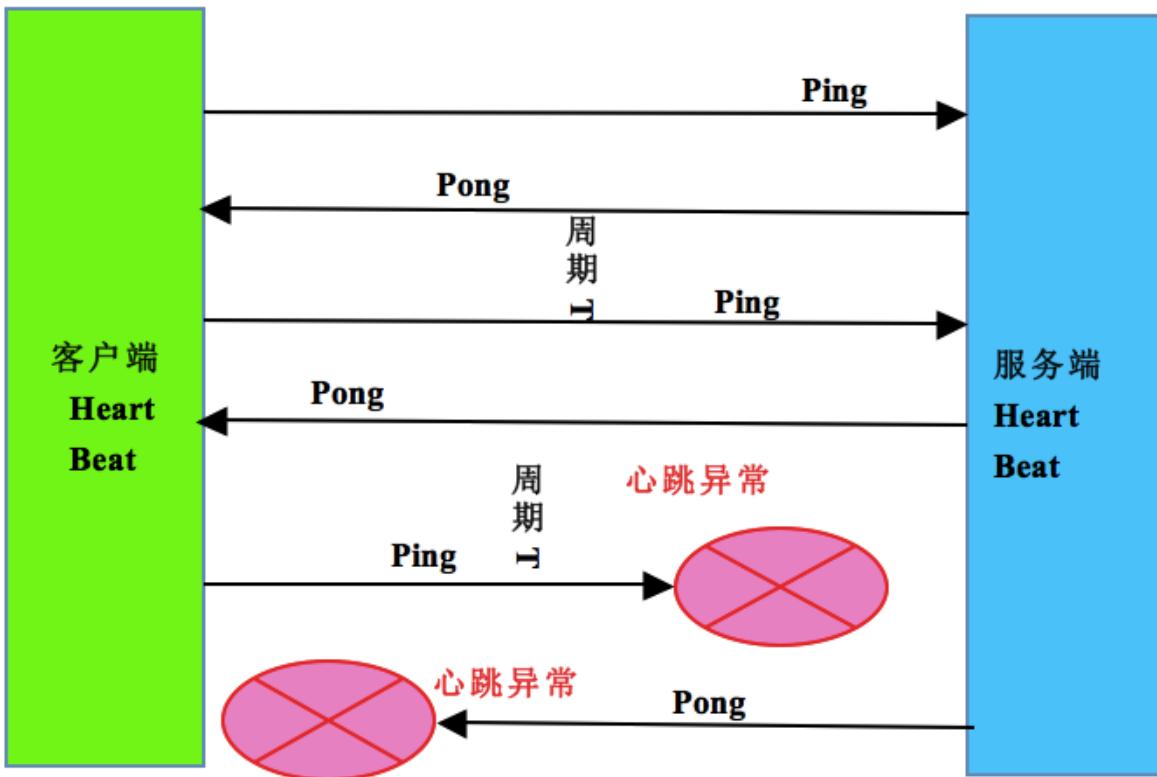


图 2-20 心跳检测机制

不同的协议，心跳检测机制也存在差异，归纳起来主要分为两类：

- 1) Ping-Pong 型心跳：由通信一方定时发送 Ping 消息，对方接收到 Ping 消息之后，立即返回 Pong 应答消息给对方，属于请求-响应型心跳；
- 2) Ping-Ping 型心跳：不区分心跳请求和应答，由通信双方按照约定定时向对方发送心跳 Ping 消息，它属于双向心跳。

心跳检测策略如下：

- 1) 连续 N 次心跳检测都没有收到对方的 Pong 应答消息或者 Ping 请求消息，则认为链路已经发生逻辑失效，这被称作心跳超时；
- 2) 读取和发送心跳消息的时候如果直接发生了 IO 异常，说明链路已经失效，这被称为心跳失败。

无论发生心跳超时还是心跳失败，都需要关闭链路，由客户端发起重连操作，保证链路能够恢复正常。

Netty 的心跳检测实际上是利用了链路空闲检测机制实现的，相关代码如下：

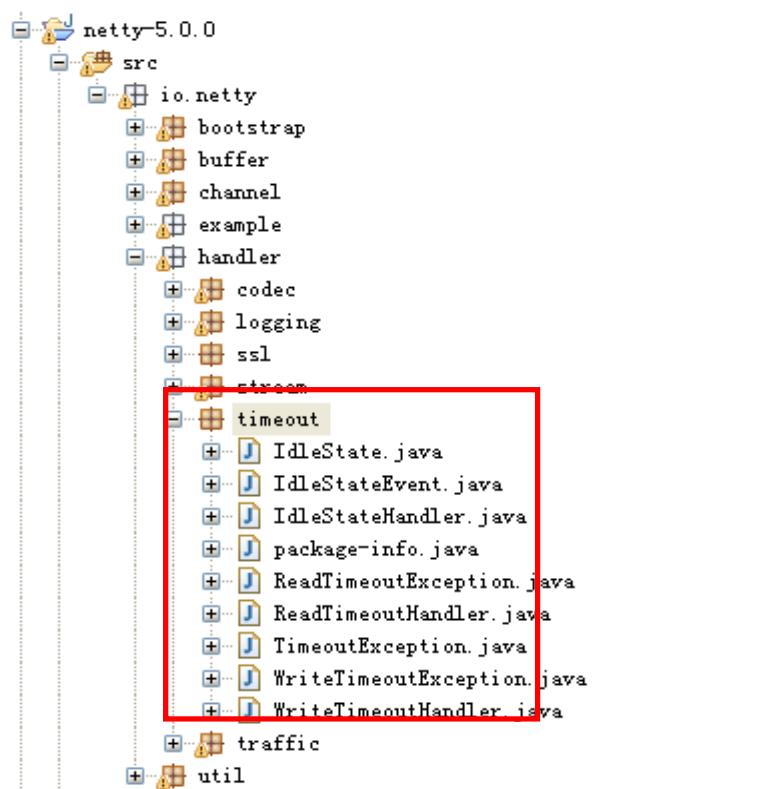


图 2-21 心跳检测的代码包路径

Netty 提供的空闲检测机制分为三种：

- 1) 读空闲，链路持续时间 t 没有读取到任何消息；
- 2) 写空闲，链路持续时间 t 没有发送任何消息；
- 3) 读写空闲，链路持续时间 t 没有接收或者发送任何消息。

Netty 的默认读写空闲机制是发生超时异常，关闭连接，但是，我们可以定制它的超时实现机制，以便支持不同的用户场景。

WriteTimeoutHandler 的超时接口如下：

```

    /**
     * Is called when a write timeout was detected
     */
protected void writeTimedOut(ChannelHandlerContext ctx) throws Exception {
    if (!closed) {
        ctx.fireExceptionCaught(WriteTimeoutException.INSTANCE);
        ctx.close();
        closed = true;
    }
}

```

图 2-22 写超时

ReadTimeoutHandler 的超时接口如下：

```

    /**
     * Is called when a read timeout was detected.
     */
protected void readTimedOut(ChannelHandlerContext ctx) throws Exception {
    if (!closed) {
        ctx.fireExceptionCaught(ReadTimeoutException.INSTANCE);
        ctx.close();
        closed = true;
    }
}

```

图 2-23 读超时

读写空闲的接口如下：

```

    /**
     * Is called when an (@link IdleStateEvent) should be fired. This implementation calls
     * (@link ChannelHandlerContext#fireUserEventTriggered(Object)).
     */
protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt) throws Exception {
    ctx.fireUserEventTriggered(evt);
}

```

图 2-24 读写空闲

利用 Netty 提供的链路空闲检测机制，可以非常灵活的实现协议层的心跳检测。在《Netty 权威指南》中的私有协议栈设计和开发章节，我利用 Netty 提供的自定义 Task 接口实现了另一种心跳检测机制，感兴趣的朋友可以参阅该书。

2.3. Reactor 线程的保护

Reactor 线程是 IO 操作的核心，NIO 框架的发动机，一旦出现故障，将会导致挂载在其上面的多路用复用器和多个链路无法正常工作。因此它的可靠性要求非常高。

笔者就曾经遇到过因为异常处理不当导致 Reactor 线程跑飞，大量业务请求处理失败的故障。下面我们一起看下 Netty 是如何有效提升 Reactor 线程的可靠性的。

2.3.1. 异常处理要当心

尽管 Reactor 线程主要处理 IO 操作，发生的异常通常是 IO 异常，但是，实际上在一些特殊场景下会发生非 IO 异常，如果仅仅捕获 IO 异常可能就会导致 Reactor 线程跑飞。为了防止发生这种意外，在循环体内一定要捕获 Throwable，而不是 IO 异常或者 Exception。

Netty 的相关代码如下：

```

    processSelectedKeysOptimized(selector.selectKeys(), true);
} else {
    processSelectedKeysPlain(selector.selectedKeys());
}
final long ioTime = System.nanoTime() - ioStartTime;

final int ioRatio = this.ioRatio;
runAllTasks(ioTime * (100 - ioRatio) / ioRatio);

if (isShuttingDown()) {
    closeAll();
    if (confirmShutdown()) {
        break;
    }
}
} catch (Throwable t) {
    logger.warn("Unexpected exception in the selector loop.", t);

    // Prevent possible consecutive immediate failures that lead to
    // excessive CPU consumption.
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // Ignore.
    }
}

```

图 2-25 Reactor 线程异常保护

捕获 Throwable 之后，即便发生了意外未知对异常，线程也不会跑飞，它休眠 1S，防止死循环导致的异常绕接，然后继续恢复执行。这样处理的核心理念就是：

- 1) 某个消息的异常不应该导致整条链路不可用；

- 2) 某条链路不可用不应该导致其它链路不可用;
- 3) 某个进程不可用不应该导致其它集群节点不可用。

2.3.2. 死循环保护

通常情况下，死循环是可检测、可预防但是无法完全避免的。Reactor 线程通常处理的都是 IO 相关的操作，因此我们重点关注 IO 层面的死循环。

JDK NIO 类库最著名的就是 epoll bug 了，它会导致 Selector 空轮询，IO 线程 CPU 100%，严重影响系统的安全性和可靠性。

SUN 在 JKD1.6 update18 版本声称解决了该 BUG，但是根据业界的测试和大家的反馈，直到 JDK1.7 的早期版本，该 BUG 依然存在，并没有完全被修复。发生该 BUG 的主机资源占用图如下：

Processes: 168 total, 5 running, 6 stuck, 157 sleeping, 1215 threads											
Load Avg: 6.79, 6.38, 6.13 CPU usage: 30.86% user, 13.37% sys, 55.76% idle SharedLibs: 16											
MemRegions: 92254 total, 4194M resident, 66M private, 761M shared. PhysMem: 1681M wired, 33											
VM: 421G vsiz, 1826M framework vsiz, 13779046(0) pageins, 5483724(0) pageouts. Networks:											
Disks: 10621319/2156 read, 10691231/3986 written.											
PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	#REGS	RPRVT	RSHRD	RSIZE	VPRVT
58552	java	182.2	02:28.72	17/1	1	77	213	11M	9528K	23M	107M
0	kernel_task	19.2	11:14:21	100/5	0	2	1181-	123M-	08	834M-	132M-
58569	top	17.3	00:11.65	1/1	0	34	32	2356K	216K	3104K	18M

图 2-26 epoll bug CPU 空轮询

SUN 在解决该 BUG 的问题上不给力，只能从 NIO 框架层面进行问题规避，下面我们看下 Netty 是如何解决该问题的。

Netty 的解决策略：

- 1) 根据该 BUG 的特征，首先侦测该 BUG 是否发生；
- 2) 将问题 Selector 上注册的 Channel 转移到新建的 Selector 上；
- 3) 老的问题 Selector 关闭，使用新建的 Selector 替换。

下面具体看下代码，首先检测是否发生了该 BUG：

```

try {
    int selectCnt = 0;
    long currentTimeNanos = System.nanoTime();
    long selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);
    for (;;) {
        long timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;
        if (timeoutMillis <= 0) {
            if (selectCnt == 0) {
                selector.selectNow();
                selectCnt = 1;
            }
            break;
        }

        int selectedKeys = selector.select(timeoutMillis);
        selectCnt++;
    }
}

```

图 2-27 epoll bug 检测

一旦检测发生该 BUG，则重建 Selector，代码如下：

```

try {
    for (SelectionKey key: oldSelector.keys()) {
        Object a = key.attachment();
        try {
            if (key.channel().keyFor(newSelector) != null) {
                continue;
            }

            int interestOps = key.interestOps();
            key.cancel();
            key.channel().register(newSelector, interestOps, a);
            nChannels++;
        } catch (Exception e) {
            logger.warn("Failed to re-register a Channel to the new Selector.", e);
            if (a instanceof AbstractNioChannel) {
                AbstractNioChannel ch = (AbstractNioChannel) a;
                ch.unsafe().close(ch.unsafe().voidPromise());
            } else {
                @SuppressWarnings("unchecked")
                NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
                invokeChannelUnregistered(task, key, e);
            }
        }
    }
}

```

图 2-28 重建 Selector

重建完成之后，替换老的 Selector，代码如下：

```

if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
    selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
    // The selector returned prematurely many times in a row.
    // Rebuild the selector to work around the problem.
    logger.warn(
        "Selector.select() returned prematurely () times in a row; rebuilding selector.",
        selectCnt);

    rebuildSelector();
    selector = this.selector;

    // Select again to populate selectedKeys.
    selector.selectNow();
    selectCnt = 1;
    break;
}

```

图 2-29 替换 Selector

大量生产系统的运行表明，Netty 的规避策略可以解决 epoll bug 导致的 IO 线程 CPU 死循环问题。

2.4. 优雅退出

Java 的优雅停机通常通过注册 JDK 的 ShutdownHook 来实现，当系统接收到退出指令后，首先标记系统处于退出状态，不再接收新的消息，然后将积压的消息处理完，最后调用资源回收接口将资源销毁，最后各线程退出执行。

通常优雅退出有个时间限制，例如 30S，如果到达执行时间仍然没有完成退出前的操作，则由监控脚本直接 kill -9 pid，强制退出。

Netty 的优雅退出功能随着版本的优化和演进也在不断的增强，下面我们一起看下 Netty5 的优雅退出。

首先看下 Reactor 线程和线程组，它们提供了优雅退出接口。EventExecutorGroup 的接口定义如下：

```

/**
 * Signals this executor that the caller wants the executor to be shut down.
 * (@link #isShuttingDown()) starts to return (@code true), and the executor
 * Unlike (@link #shutdown()), graceful shutdown ensures that no tasks are submitted
 * (usually a couple seconds) before it shuts itself down. If a task is submitted
 * it is guaranteed to be accepted and the quiet period will start over.
 *
 * @param quietPeriod the quiet period as described in the documentation
 * @param timeout      the maximum amount of time to wait until the executor is
 *                     shutdown regardless if a task was submitted during the quiet period
 * @param unit         the unit of (@code quietPeriod) and (@code timeout)
 *
 * @return the (@link #terminationFuture())
 */
Future<?> shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit);

```

图 2-30 EventExecutorGroup 优雅退出

NioEventLoop 的资源释放接口实现：

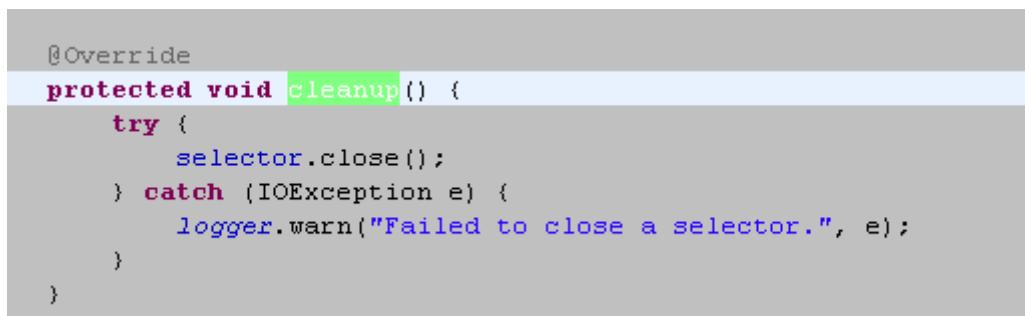


图 2-31 NioEventLoop 资源释放

ChannelPipeline 的关闭接口：

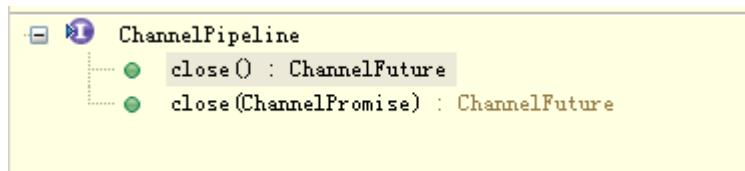


图 2-32 ChannelPipeline 关闭接口

目前 Netty 向用户提供的主要接口和类库都提供了资源销毁和优雅退出的接口，用户的自定义实现类可以继承这些接口，完成用户资源的释放和优雅退出。

2.5. 内存保护

2.5.1. 缓冲区的内存泄漏保护

为了提升内存的利用率，Netty 提供了内存池和对象池。但是，基于缓存池实现以后需要对内存的申请和释放进行严格的管理，否则很容易导致内存泄漏。

如果不采用内存池技术实现，每次对象都是以方法的局部变量形式被创建，使用完成之后，只要不再继续引用它，JVM 会自动释放。但是，一旦引入内存池机制，对象的生命周期将由内存池负责管理，这通常是个全局引用，如果不显式释放 JVM 是不会回收这部分内存的。

对于 Netty 的用户而言，使用者的技术水平差异很大，一些对 JVM 内存模型和内存

泄漏机制不了解的用户，可能只记得申请内存，忘记主动释放内存，特别是 JAVA 程序员。

为了防止因为用户遗漏导致内存泄漏，Netty 在 Pipe line 的尾 Handler 中自动对内存进行释放，相关代码如下：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    try {
        logger.debug(
            "Discarded inbound message () that reached at the tail of the pipeline. " +
            "Please check your pipeline configuration.", msg);
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

图 2-33 TailHandler 的内存回收操作

对于内存池，实际就是将缓冲区重新放到内存池中循环使用，代码如下：

```
@Override
protected final void deallocate() {
    if (handle >= 0) {
        final long handle = this.handle;
        this.handle = -1;
        memory = null;
        chunk.arena.free(chunk, handle);
        recycle();
    }
}
```

图 2-34 PooledByteBuf 的内存回收操作

2.5.2. 缓冲区内存溢出保护

做过协议栈的读者都知道，当我们对消息进行解码的时候，需要创建缓冲区。缓冲区的创建方式通常有两种：

- 1) 容量预分配，在实际读写过程中如果不够再扩展；
- 2) 根据协议消息长度创建缓冲区。

在实际的商用环境中，如果遇到畸形码流攻击、协议消息编码异常、消息丢包等问题时，可能会解析到一个超长的长度字段。笔者曾经遇到过类似问题，报文长度字段值竟然是 2G 多，由于代码的一个分支没有对长度上限做有效保护，结果导致内存溢出。系

统重启后几秒内再次内存溢出，幸好及时定位出问题根因，险些酿成严重的事故。

Netty 提供了编解码框架，因此对于解码缓冲区的上限保护就显得非常重要。下面，我们看下 Netty 是如何对缓冲区进行上限保护的：

首先，在内存分配的时候指定缓冲区长度上限：

```
/**  
 * Allocate a {@link ByteBuf} with the given initial capacity and the given  
 * maximal capacity. If it is a direct or heap buffer depends on the actual  
 * implementation.  
 */  
ByteBuf buffer(int initialCapacity, int maxCapacity);
```

图 2-35 缓冲区分配器可以指定缓冲区最大长度

其次，在对缓冲区进行写入操作的时候，如果缓冲区容量不足需要扩展，首先对最大容量进行判断，如果扩展后的容量超过上限，则拒绝扩展：

```
if (minWritableBytes > maxCapacity - writerIndex) {  
    throw new IndexOutOfBoundsException(String.format(  
        "writerIndex(%d) + minWritableBytes(%d) exceeds maxCapacity(%d): %s",  
        writerIndex, minWritableBytes, maxCapacity, this));  
}
```

图 2-35 缓冲区扩展上限保护

最后，在解码的时候，对消息长度进行判断，如果超过最大容量上限，则抛出解码异常，拒绝分配内存：

```
if (frameLength > maxFrameLength) {  
    long discard = frameLength - in.readableBytes();  
    tooLongFrameLength = frameLength;  
  
    if (discard < 0) {  
        // buffer contains more bytes than the frameLength so we can discard all now.  
        in.skipBytes((int) frameLength);  
    } else {  
        // Enter the discard mode and discard everything received so far.  
        discardingTooLongFrame = true;  
        bytesToDiscard = discard;  
        in.skipBytes(in.readableBytes());  
    }  
    failIfNecessary(true);  
    return null;  
}
```

图 2-36 超出容量上限的半包解码，失败

```
private void fail(long frameLength) {
    if (frameLength > 0) {
        throw new TooLongFrameException(
            "Adjusted frame length exceeds " + maxFrameLength +
            ": " + frameLength + " - discarded");
    } else {
        throw new TooLongFrameException(
            "Adjusted frame length exceeds " + maxFrameLength +
            " - discarding");
    }
}
```

图 2-37 抛出 TooLongFrameException 异常

2.6. 流量整形

大多数的商用系统都有多个网元或者部件组成，例如参与短信互动，会涉及到手机、基站、短信中心、短信网关、SP/CP 等网元。不同网元或者部件的处理性能不同。为了防止因为浪涌业务或者下游网元性能低导致下游网元被压垮，有时候需要系统提供流量整形功能。

下面我们一起看下流量整形(traffic shaping)的定义：流量整形（Traffic Shaping）是一种主动调整流量输出速率的措施。一个典型应用是基于下游网络结点的 TP 指标来控制本地流量的输出。流量整形与流量监管的主要区别在于，流量整形对流量监管中需要丢弃的报文进行缓存——通常是将它们放入缓冲区或队列内，也称流量整形（Traffic Shaping，简称 TS）。当令牌桶有足够的令牌时，再均匀的向外发送这些被缓存的报文。流量整形与流量监管的另一区别是，整形可能会增加延迟，而监管几乎不引入额外的延迟。

流量整形的原理示意图如下：

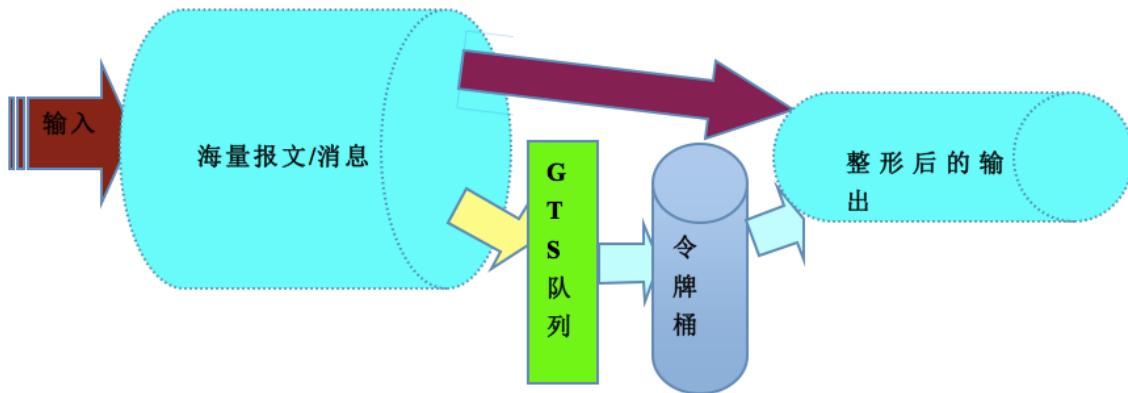


图 2-38 流量整形原理图

作为高性能的 NIO 框架，Netty 的流量整形有两个作用：

- 1) 防止由于上下游网元性能不均衡导致下游网元被压垮，业务流程中断；
- 2) 防止由于通信模块接收消息过快，后端业务线程处理不及时导致的“撑死”问题。

下面我们就具体学习下 Netty 的流量整形功能。

2.6.1. 全局流量整形

全局流量整形的作用范围是进程级的，无论你创建了多少个 Channel，它的作用域针对所有的 Channel。

用户可以通过参数设置：报文的接收速率、报文的发送速率、整形周期。相关的接口如下所示：

```

/**
 * Create a new instance
 *
 * @param executor
 *         the (@link ScheduledExecutorService) to use for the (@link TrafficCounter)
 * @param writeLimit
 *         0 or a limit in bytes/s
 * @param readLimit
 *         0 or a limit in bytes/s
 * @param checkInterval
 *         The delay between two computations of performances for
 *         channels or 0 if no stats are to be computed
 */
public GlobalTrafficShapingHandler(ScheduledExecutorService executor, long writeLimit,
                                    long readLimit, long checkInterval) {
    super(writeLimit, readLimit, checkInterval);
    createGlobalTrafficCounter(executor);
}

```

图 2-39 全局流量整形参数设置

Netty 流量整形的原理是：对每次读取到的 ByteBuf 可写字节数进行计算，获取当前的报文流量，然后与流量整形阈值对比。如果已经达到或者超过了阈值，则计算等待时间 delay，将当前的 ByteBuf 放到定时任务 Task 中缓存，由定时任务线程池在延迟 delay 之后继续处理该 ByteBuf。相关代码如下：

```
@Override  
public void channelRead(final ChannelHandlerContext ctx, final Object msg) throws Exception {  
    long size = calculateSize(msg);  
    long curtime = System.currentTimeMillis();  
  
    if (trafficCounter != null) {  
        trafficCounter.bytesRecvFlowControl(size);  
        if (readLimit == 0) {  
            // no action  
            ctx.fireChannelRead(msg);  
  
            return;  
        }  
    }  
}
```

图 2-40 动态计算当前流量

如果达到整形阈值，则对新接收的 ByteBuf 进行缓存，放入线程池的消息队列中，稍后处理，代码如下：

```
TimeUnit.MILLISECONDS);  
} else {  
    // Create a Runnable to update the next handler in the chain. If one was  
    // just be reused to limit object creation  
    Runnable bufferUpdateTask = new Runnable() {  
        @Override  
        public void run() {  
            ctx.fireChannelRead(msg);  
        }  
    };  
    ctx.executor().schedule(bufferUpdateTask, wait, TimeUnit.MILLISECONDS);  
    return;  
}
```

图 2-41 缓存当前的 ByteBuf

定时任务的延时时间根据检测周期 T 和流量整形阈值计算得来，代码如下：

```

/*
 * private static long getTimeToWait(long limit, long bytes, long lastTime, long curtime) {
 *     long interval = curtime - lastTime;
 *     if (interval <= 0) {
 *         // Time is too short, so just lets continue
 *         return 0;
 *     }
 *     return (bytes * 1000 / limit - interval) / 10 * 10;
 * }
 */

```

图 2-42 计算缓存等待周期

需要指出的是，流量整形的阈值 limit 越大，流量整形的精度越高，流量整形功能是可靠性的一种保障，它无法做到 100% 的精确。这个跟后端的编解码以及缓冲区的处理策略相关，此处不再赘述。感兴趣的朋友可以思考下，Netty 为什么做不到 100% 的精确。

流量整形与流控的最大区别在于流控会拒绝消息，流量整形不拒绝和丢弃消息，无论接收量多大，它总能以近似恒定的速度下发消息，跟变压器的原理和功能类似。

2.6.2. 单条链路流量整形

除了全局流量整形，Netty 也支持单链路的流量整形，相关的接口定义如下：

```

/*
 * public ChannelTrafficShapingHandler(long writeLimit,
 *                                     long readLimit, long checkInterval) {
 *     super(writeLimit, readLimit, checkInterval);
 * }
 */

```

图 2-43 单链路流量整形

单链路流量整形与全局流量整形的最大区别就是它以单个链路为作用域，可以对不同的链路设置不同的整形策略。

它的实现原理与全局流量整形类似，我们不再赘述。值得说明的是，Netty 支持用户自定义流量整形策略，通过继承 AbstractTrafficShapingHandler 的 doAccounting 方法可以定制整形策略。相关接口定义如下：

```
 /**
 * Called each time the accounting is computed from the Traffic
 * This method could be used for instance to implement almost r
 *
 * @param counter
 *         the TrafficCounter that computes its performance
 */
@SuppressWarnings("unused")
protected void doAccounting(TrafficCounter counter) {
    // NOOP by default
}
```

图 2-44 定制流量整形策略

3. 总结

尽管 Netty 在架构可靠性上面已经做了很多精细化的设计，以及基于防御式编程对系统进行了大量可靠性保护。但是，系统的可靠性是个持续投入和改进的过程，不可能在一个版本中一蹴而就，可靠性工作任重而道远。

从业务的角度看，不同的行业、应用场景对可靠性的要求也是不同的，例如电信行业的可靠性要求是 5 个 9，对于铁路等特殊行业，可靠性要求更高，达到 6 个 9。对于企业的一些边缘 IT 系统，可靠性要求会低些。

可靠性是一种投资，对于企业而言，追求极端可靠性对研发成本是个沉重的包袱，但是相反，如果不重视系统的可靠性，一旦不幸遭遇网上事故，损失往往也是惊人的。对于架构师和设计师，如何权衡架构的可靠性和其它特性之间的关系，是一个很大的挑战。通过研究和学习 Netty 的可靠性设计，也许能够给大家带来一些启示。

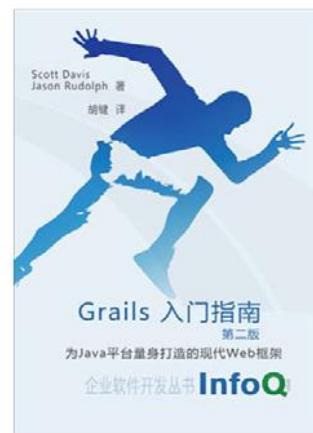
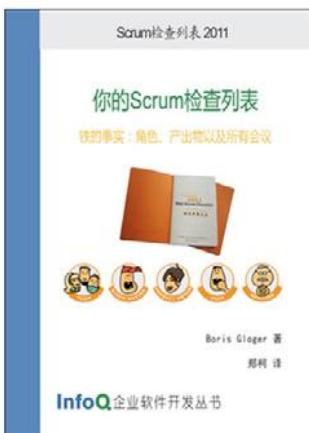
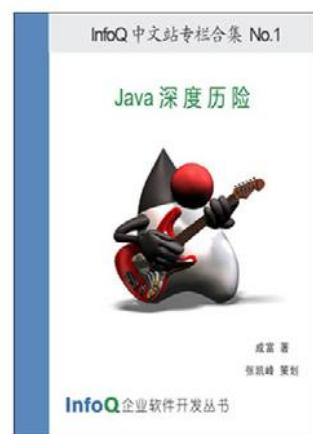
感谢郭蕾对本文的审校和策划。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至 editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)）或者腾讯微博（[@InfoQ](#)）关注我们，并与我们的编辑和其他读者朋友交流。

查看原文：[Netty 系列之 Netty 可靠性分析](#)

InfoQ 软件开发丛书

欢迎免费下载



商务合作: sales@cn.infoq.com

读者反馈/内容提供: editors@cn.infoq.com

使用 SQL Server 2014 内存数据库时需要注意的地方

作者 王枫

本文从产品设计和架构角度分享了 Microsoft 内存数据库方面的使用经验，希望你在阅读本文之后能够了解这些新的对象、概念，从而更好地设计你的架构。

内存数据库，指的是将数据库的数据放在内存中直接操作。相对于存放在磁盘上，内存的数据读写速度要高出很多，故可以提高应用的性能。微软的 SQL Server 2014 已于 2014 年 4 月 1 日正式发布，SQL 2014 一个主要的功能即为内存数据库。

下面，我将着重介绍使用 SQL Server 2014 内存数据库时需要注意的地方。

关于内存数据库

SQL Server 2014 内存数据库针对传统的表和存储过程引入了新的结构： memory optimized table（内存优化表）和 native stored procedure（本地编译存储过程）。

默认情况下 Memory optimized table 是完全持久的（即为 durable memory optimized table），如传统的基于磁盘的表上的事务一样，并且完全持久的事务也是支持原子、一致、隔离和持久（ACID）的。所不同的是内存优化表的整个表的主存储是在内存中，即为从内存读取表中的行，和更新这些行数据到内存中。并非像是传统基于磁盘的表按照数据库页面装载数据库。内存优化表的数据同时还在磁盘上维护着另一个副本，但仅用于持续性目的。在数据库恢复期间，内存优化的表中的数据再次从磁盘装载。创建持久的内存优化表方法如下：

```
CREATE TABLE DurableTbl
  (AccountNo      INT           NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH
    (BUCKET_COUNT = 28713)
  ,CustName       VARCHAR(20)   NOT NULL
  ,Gender         CHAR          NOT NULL /* M or F */
  ,CustGroup      VARCHAR(4)    NOT NULL /* which customer group he/she belongs
    to */
  ,Addr           VARCHAR(50)   NULL      /* No address supplied is acceptable */
  ,Phone          VARCHAR(10)   NULL      /* Phone number */
)
```

```
WITH (MEMORY_OPTIMIZED=ON, DURABILITY=SCHEMA_AND_DATA)
```

除了默认持久的内存优化表之外，还支持 non-durable memory optimized table（非持久化内存优化表），不记录这些表的日志且不在磁盘上保存它们的数据。这意味着这些表上的事务不需要任何磁盘 IO，但如果服务器崩溃或进行故障转移，则无法恢复数据。创建非持久化内存优化表方法如下：

```
CREATE TABLE NonDurableTbl
(AccountNo      INT           NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH
(BUCKET_COUNT = 28713)
,CustName       VARCHAR(20)    NOT NULL
,Gender         CHAR          NOT NULL /* M or F */
,CustGroup      VARCHAR(4)     NOT NULL /* which customer group he/she belongs
to */
,Addr           VARCHAR(50)    NULL      /* No address supplied is acceptable */
,Phone          VARCHAR(10)    NULL      /* Phone number */
)
WITH (MEMORY_OPTIMIZED=ON, DURABILITY=SCHEMA_ONLY)
```

Native compiled stored procedure（本地编译存储过程）是针对传统的存储过程而言的，是本机编译存储过程后生成 DLL，由于本机编译是指将编程构造转换为本机代码的过程，这些代码由处理器指令组成，无需进一步编译或解释。与传统 TSQL 相比，本机编译可提高访问数据的速度和执行查询的效率。故通过本机编译的存储过程，可在存储过程中提高查询和业务逻辑处理的效率。创建方法本地编译存储过程方法如下：

```
CREATE PROCEDURE dbo.usp_InsertNonDurableTbl
@AccountNo int,
@CustName nvarchar(20),
@Gender char(1),
@CustGroup varchar(4),
@Addr varchar(50),
@Phone varchar(10)
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'English')
BEGIN
    INSERT INTO [dbo].[DurableTbl]
    ([AccountNo]
    ,[CustName]
```

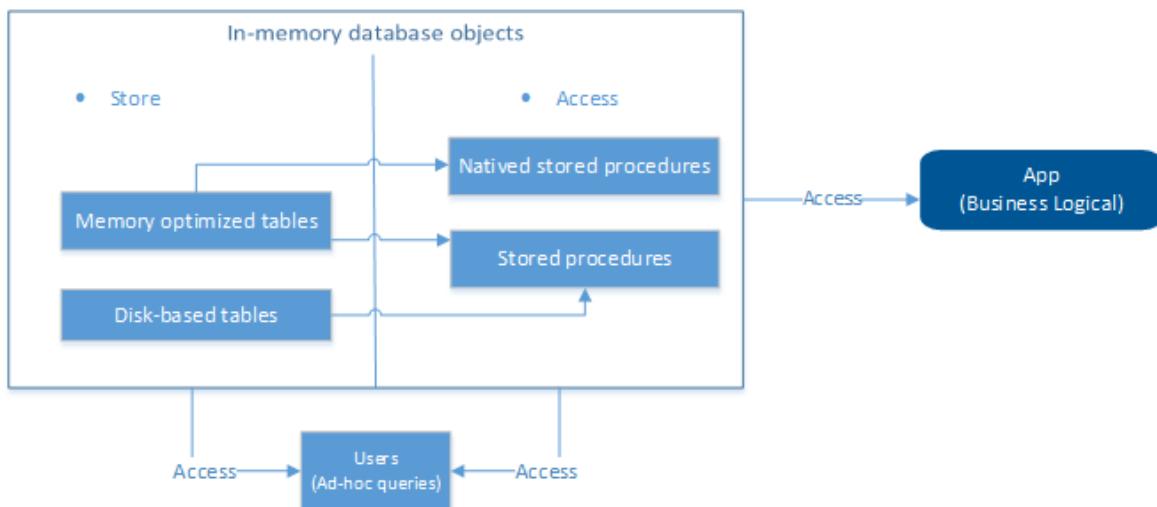
```

        , [Gender]
        , [CustGroup]
        , [Addr]
        , [Phone])
VALUES  (@AccountNo
        , @CustName
        , @Gender
        , @CustGroup
        , @Addr
        , @Phone)

END
END
GO

```

内存数据库既可以包含内存优化表和本地编译存储过程，又可以包含基于磁盘的表和传统存储过程，各个对象之间数据存储、和访问的架构如下所示：



使用场景

传统基于磁盘的表，通常会遇到内存页面置换、死锁、造成了吞吐量有限、事务延迟较长等问题，内存数据库的内存优化表由于常驻内存，适用于低延迟、高并发、快速数据传输和装载等场景。各场景的使用、机制具体如下：

低延迟：由于内存优化表和本地编译存储过程直接生成 DLL，本机编译可提高访问数据的速度和执行查询的效率响应速度快，作为参与处理业务逻辑的存储过程而言，大大降低了存储过程作为中间层执行和访问的效率。提高了应用的访问效率，降低了延迟性。

内存优化表的创建和装载过程如下：

CREATE TABLE DDL

Code generation and compilation

Table DLL produced

Table DLL loaded

本地编译存储过程的创建和装载过程如下：

CREATE PROC DDL

Query optimization

Code generation and compilation

Procedure DLL produced

Procedure DLL loaded

对于基于磁盘的表和内存优化表，我们可以在以下示例中对比内存优化表：创建两个同样结构的表，一个为基于磁盘的表包含 1700 万条记录，当使用常规存储过程查询一条记录，查询时间为 67ms；

```
SQL Server Execution Times:
  CPU time = 0 ms,  elapsed time = 67 ms.
SQL Server parse and compile times:
  CPU time = 0 ms,  elapsed time = 0 ms.

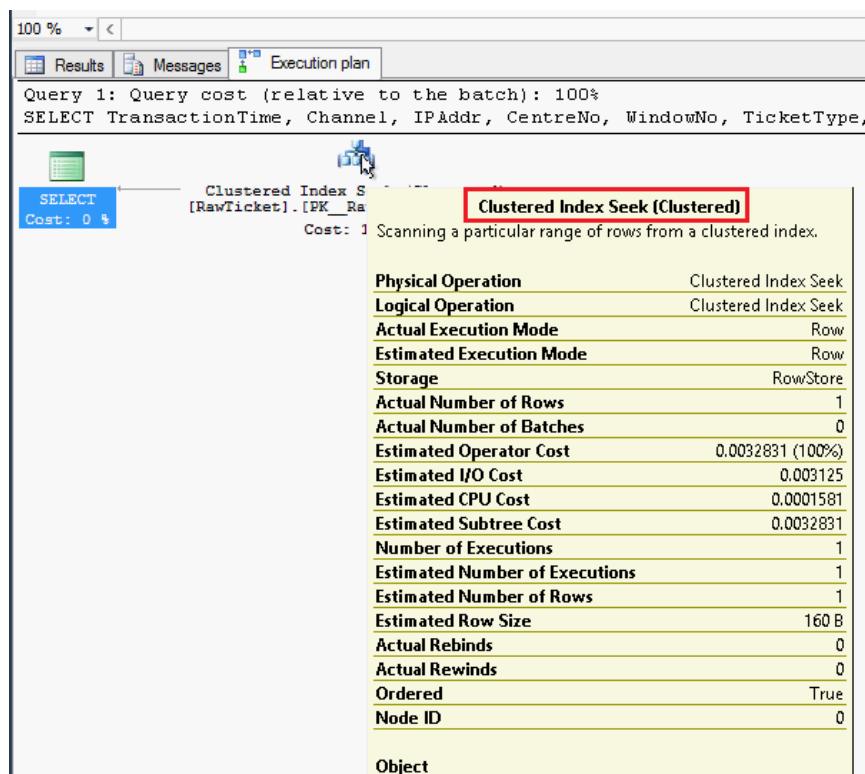
SQL Server Execution Times:
  CPU time = 0 ms,  elapsed time = 0 ms.
```

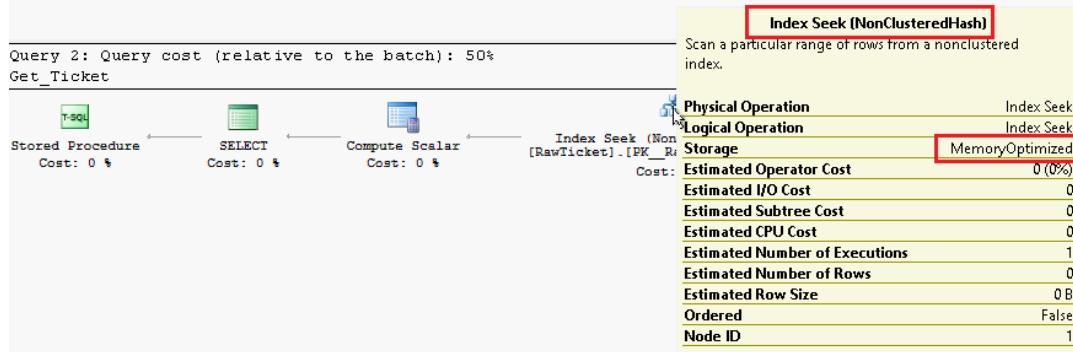
另一个为内存优化表包含 1 亿条记录。当使用本地编译存储过程查询内存优化表，所需的执行时间不到 1 毫秒。

```
SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.

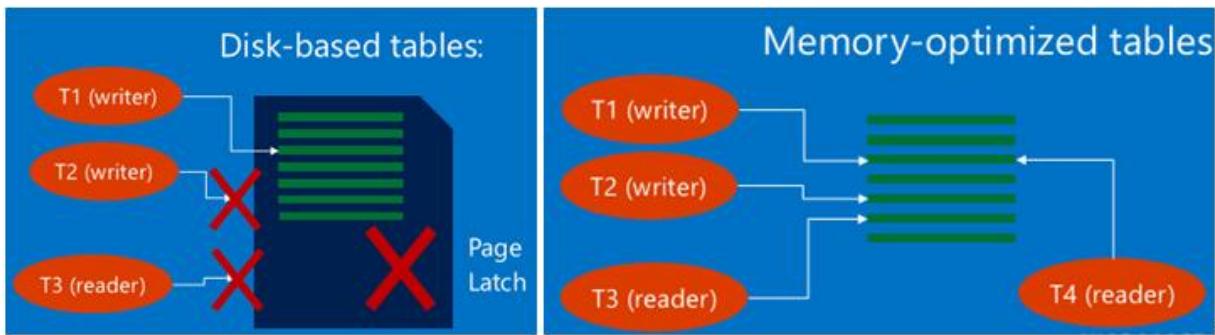
SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
```

当我们进一步查看两个存储过程的执行计划，发现第一个已经使用聚集索引检索，第二个本地编译存储过程如所预期的，是基于内存优化表的索引检索。

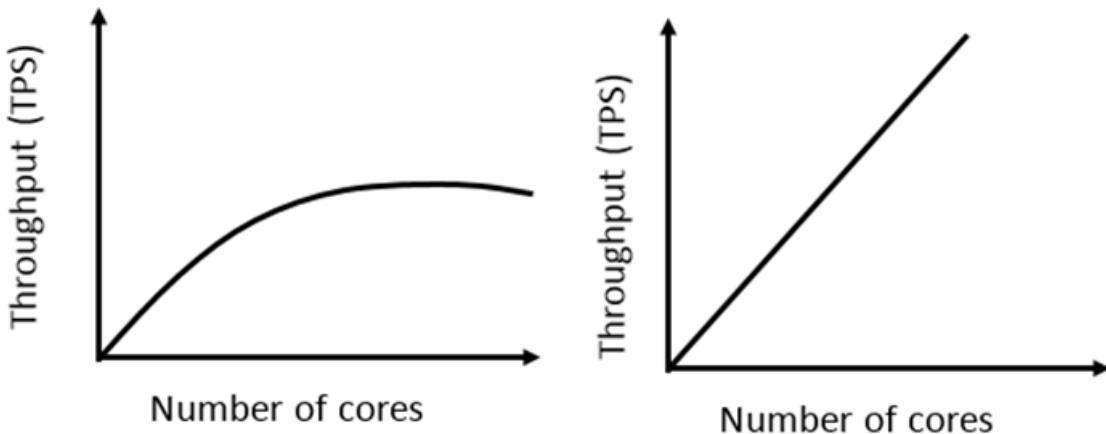




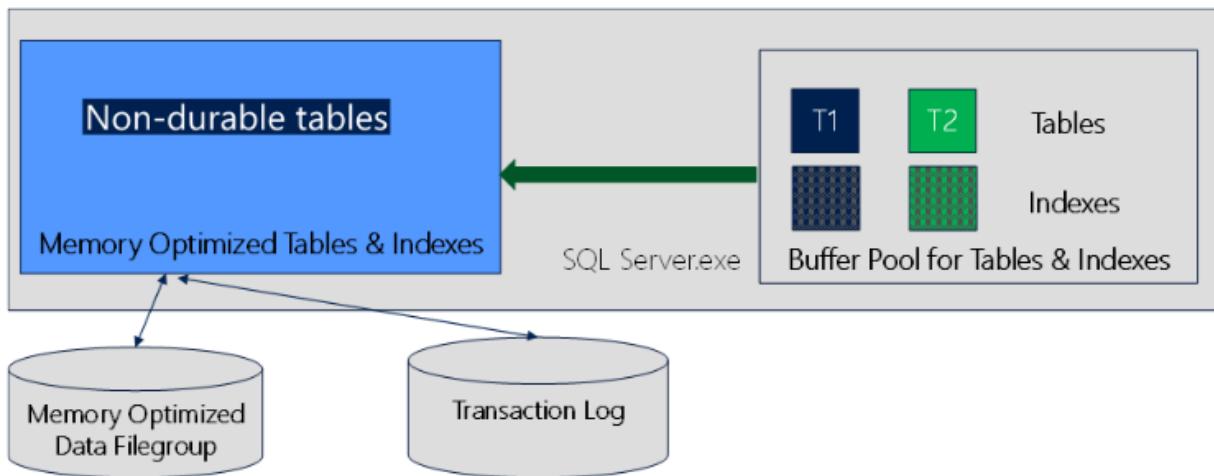
高吞吐量: 由于内存优化表直接从内存中读取、写入数据，当访问数据时，不再使用 latch，故不同于基于磁盘的表，对于 insert/update/delete 的操作，latch 争用、以及死锁问题随即消失。



与此同时，可大大提高了应用的吞吐量。 随着配置的增加，其性能呈直线上升。



快速数据传输、装载: 由于非持久化内存优化表仅常驻内存，并无基于磁盘的副本。当需要将一些外部数据通过 ETL 装载到内存数据库，可以使用无任何 IO 和 logging 的非持久化内存优化表作为过渡表，可有效的加快装载数据库的速度。

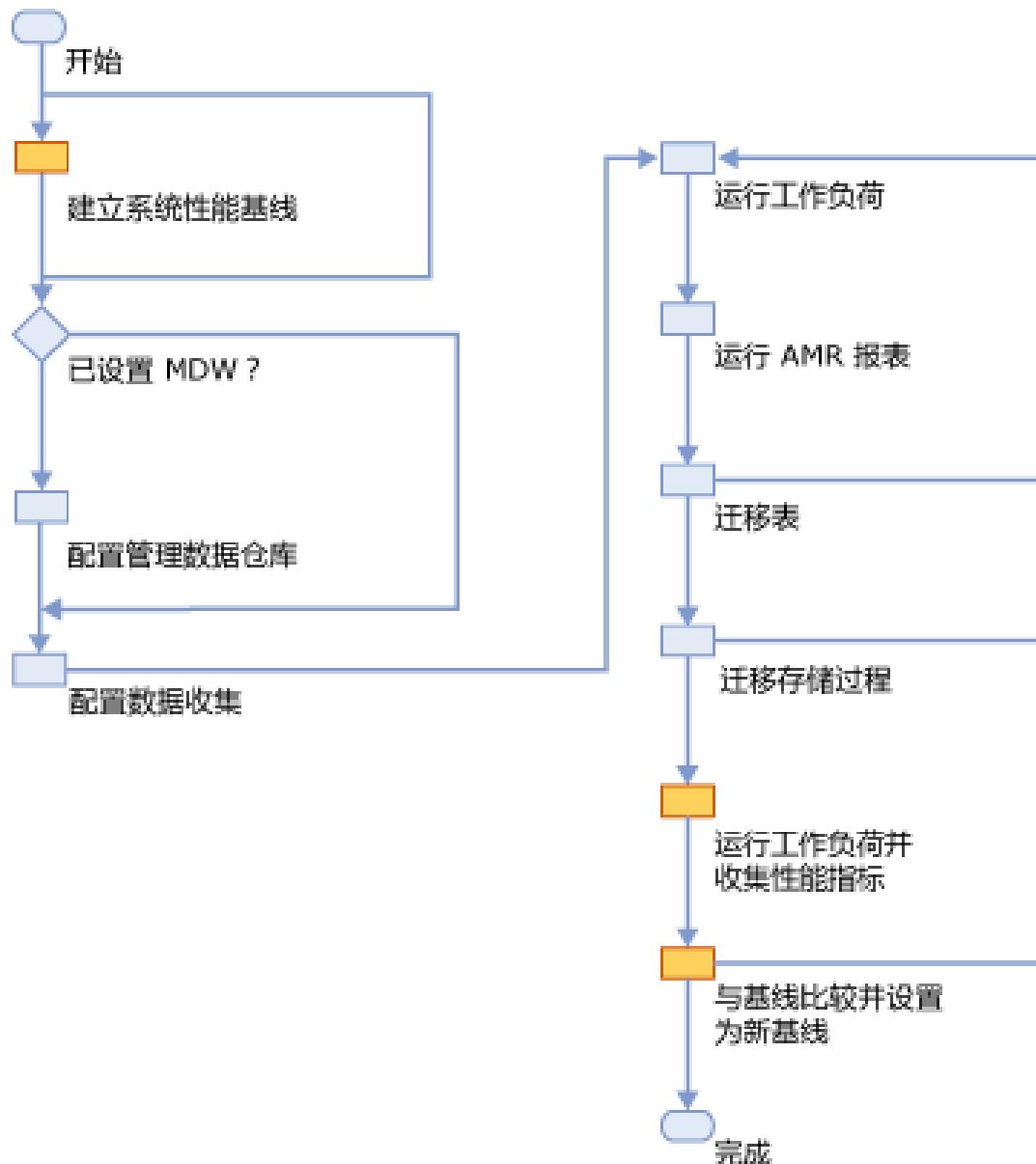


内存数据库设计与性能

并非所有的场景都可以利用到 OLTP 的内存数据库的优势，针对符合内存数据库[使用场景](#)的需求，需确定哪些对象适合转化为内存优化表和本地编辑存储过程，对于已经存在的系统的表对象，如何迁移这些对象。

选择合适的内存优化表

SQL Server 2014 提供了 AMR 即为 Analysis, Migration and Reporting，此工具可来检测哪些基于磁盘的表和存储过程适合迁移到内存数据库中。下面的流程图给出了建议的工作流程：



经常用于做为核心基线的一些指标如下：

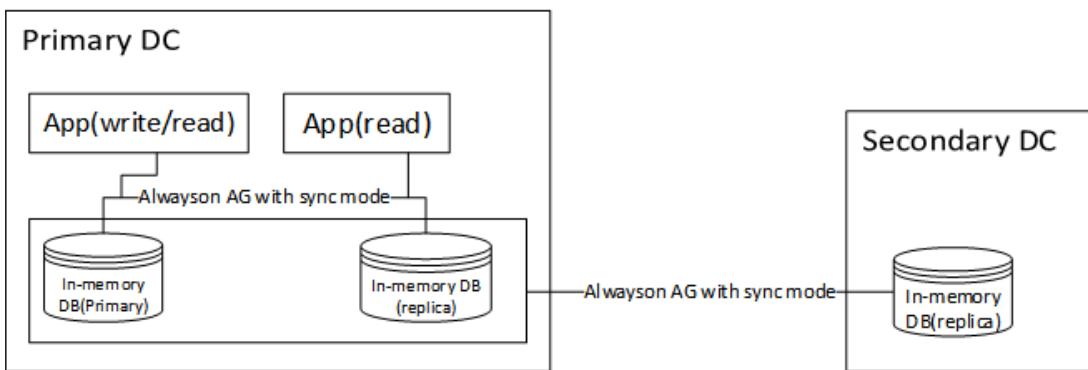
- SQL Server 的 CPU 占用率。
- SQL Server 的内存占用率。
- SQL Server 的 I/O 活动。
- 处理事务时，实例的事务吞吐量。

当已经确定哪些表需要调整为内存优化表，可针对内存优化顾问的“表内存优化顾问”所列出来的清单一一调整，且评估每个表对内存的使用量。

通常在实际生产环境中，为了保证服务的高可用性和数据的完整性、安全性，几乎很少有数据库为单实例结构，紧接着面临的问题是，如何实现内存数据库的高可用性。

内存数据库的高可用性

SQL 2014 的内存数据库与现在有诸如群集、Alwayson、replication 等高可用技术完全集成，故基于内存数据库的基础上，搭建 SQL Server Alwayson Availability Group，考虑到同一数据中心带宽和网络延迟优于跨数据中心，可在同一数据中心采用同步模式作为高可用，不同数据库中心采用异步模式作为灾备。架构如下：



由于内存数据库本身常驻内存，在设计架构时需要注意不同高可用的局限性：

群集：考虑到数据库服务的高可用性，传统基于磁盘的数据库经常采用数据库群集保证应用服务的不间断性。同样内存数据库适用于数据库群集，故 Active/Passive、Active/Active、以至于 M/N（多个活动节点/多个被动节点）模式的群集均可考虑内存数据库，所需注意的是：

- 在故障切换时，由于内存优化表需要将所有数据装载到内存中，切换时间比基于磁盘的表时间略长。
- 非持久性内存优化表由于磁盘并未存放数据副本，在故障切换时，数据内容会被清空。

Alwayson：在 SQL 2012 中出现的新功能 Alwayson availability group 可为数据库提供多个同步或者异步的数据库副本，在 SQL 14 中内存数据库与 Alwayson availability group 可完全集成。依赖于 Alwayson 的部署向导，内存数据库可像传统数据库一样，快速加入 Alwayson availability group 中，所需注意的是：

- 在切换主从数据库时，切换时间较快，由于依赖于 alwayson 的事务日志记录的 redo 进程，无需从磁盘重新装载数据库到内存中。
- 若内存数据库中包含非持久性内存优化表，由于无法依赖于事务日志，非持久性内存优化表的数据仅存在于 primary 节点。

通常 Alwayson 也被使用于本地数据库的高可用性，和异地数据库的灾备场景，与内存优化表的结合在性能上，对于主从节点之间网络延迟、传递的事务的大小、以及内存数据库所在的磁盘是否较快，均可影响其性能。

Replication: 复制是将数据和数据库对象从一个数据库复制和分发到另一个数据库，然后在数据库之间进行同步以保持一致性的一种技术。内存数据库中的内存优化表可作为单向事务性复制的订阅方，所需注意的是：

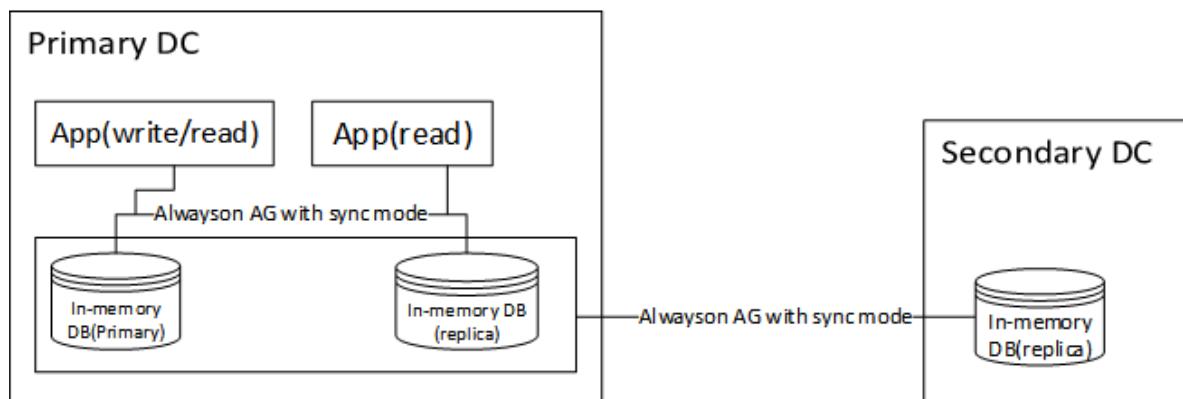
- 内存优化表的行数据限制在 8060 bytes 一下。
- 复制订阅方的数据类型要遵循内存优化表的限制。

数据库架构设计

由于持久性内存优化表需要在服务启动时，将数据装载到内存中，这涉及对现有 RTO 有一定量的影响。在设计内存数据库文件组的架构时，需注意完全持久的内存优化表的大小、以及装载数据的速度。

在由架构和业务数据量确定内存优化表的大小的前提下，可通过多个 Container 提升内存数据库的数据装载的速度。

由于每个 Container 包含着检查点文件对（Checkpoint File Pairs 即为 CFPs），CFP 由数据文件和差异文件构成，内存优化表中的数据存储在 CFP 中。为提高数据库服务启动时 RTO，在为内存优化数据库创建多个 container 时，可并行处理不同 Container 内的检查点文件对，即为提高装载数据到内存数据库的速度。



例如创建 Container 可在创建数据库时创建，或者一个或多个 container 添加到 MEMORY_OPTIMIZED_DATA 文件组，脚本如下所示：

```

CREATE DATABASE InMemory_DBTest ON
    PRIMARY (NAME = [InMemory_DB_hk_fs_data], FILENAME =
'D:\InMemory_DBTest\InMemory_DB_data.mdf'),
    FILEGROUP [InMemory_DB_fs_fg] CONTAINS MEMORY_OPTIMIZED_DATA

```

```

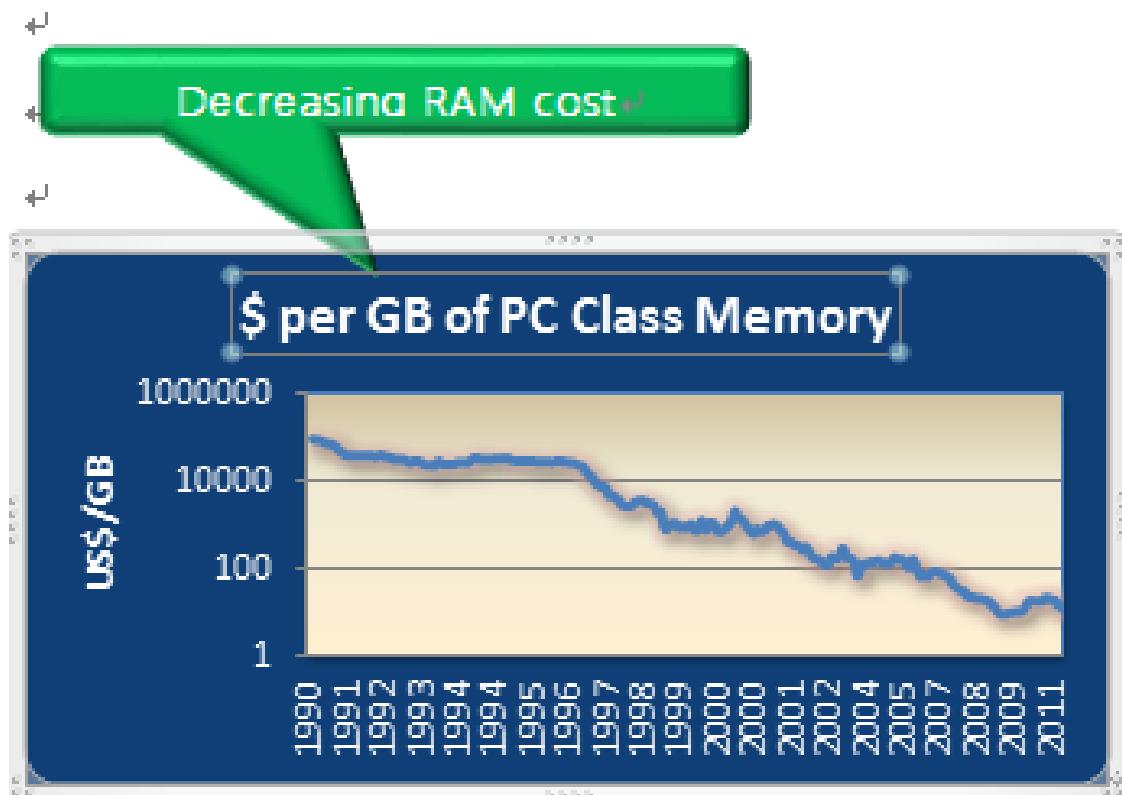
(NAME = [InMemory_DB_fs_dir], FILENAME = 'D:\ InMemory_DBTest\
InMemory_DB_hk_fs_dir'),
(NAME = [InMemory_DB_fs_dir2], FILENAME = 'D:\ InMemory_DBTest\
InMemory_DB_hk_fs_dir2'),
(NAME = [InMemory_DB_fs_dir3], FILENAME = 'D:\ InMemory_DBTest\
InMemory_DB_hk_fs_dir3')
LOG ON (name = [test_log], Filename='D:\ InMemory_DBTest\
InMemory_DB.ldf', size=100MB)
COLLATE Welsh_100_BIN
Go

```

此外，并在不同的驱动器上分配这些 Container，以实现更多带宽来将数据传输到内存中。由于内存数据库引擎会根据轮询法跨 Container 分发数据文件和差异文件，为提高 Container 对磁盘的带宽的性能，应在每个磁盘均衡数据文件和差异文件。

对于设计内存优化表时，需要考虑 bucket 的数量，一般来讲建议 bucket 的数量为预估表记录的 1-2 倍。

相对于磁盘，内存的数据读写速度要高出几个数量级，将数据保存在内存中相比从磁盘上访问能够极大地提高应用的性能。由于内存数据库是以牺牲内存资源为代价换取数据处理实时性的，以下图表显示了近些年计算机硬件（内存）飞速发展，为内存数据库的使用带来了可能性。



内存数据库在使用硬件资源与传统表有着一定的特殊性，为了提高内存数据库性能，对存储内存数据库的各方面的资源有着比传统数据库更高的要求。可参考如下具体需求：

内存：所有内存优化表是常驻内存的，因此需足够的物理内存来存储内存优化表。但这并不意味着需要将整个数据库放入内存中，而是仅将频繁访问的热数据常驻内存优化表中。且最高可以支持到 256GB 的数据量。

可使用如下脚本查看内存优化表的内存使用量：

```
select object_name(object_id), * from sys.dm_db_xtp_table_memory_stats
```

磁盘：同样存在 log 和 data 两类文件。Log 文件依然记录事务信息。针对于持久性的内存优化表，为了降低 log IO 的竞争、保证低延迟，一般建议至少 SSD。

CPU：可根据 OLTP 环境的负载考虑 CPU 的配置，如两个 CPU socket 支撑一个中等级别的服务器。

Network：针对于单机的内存数据库，由于数据存储于数据库服务器的内存中，对于数据交互仍然为应用层到数据层的访问，如以往数据交互，对于网络并未有较高的依赖性。对于内存数据库应用于数据库高可用和异地灾备的情况下（如同步/异步模式的 Always-on），同一数据中心的网络延迟，以及不同数据中的网络延迟对于使用与高可用性和灾备的内存数据库的事务有一定量的影响。

维护管理内存数据库

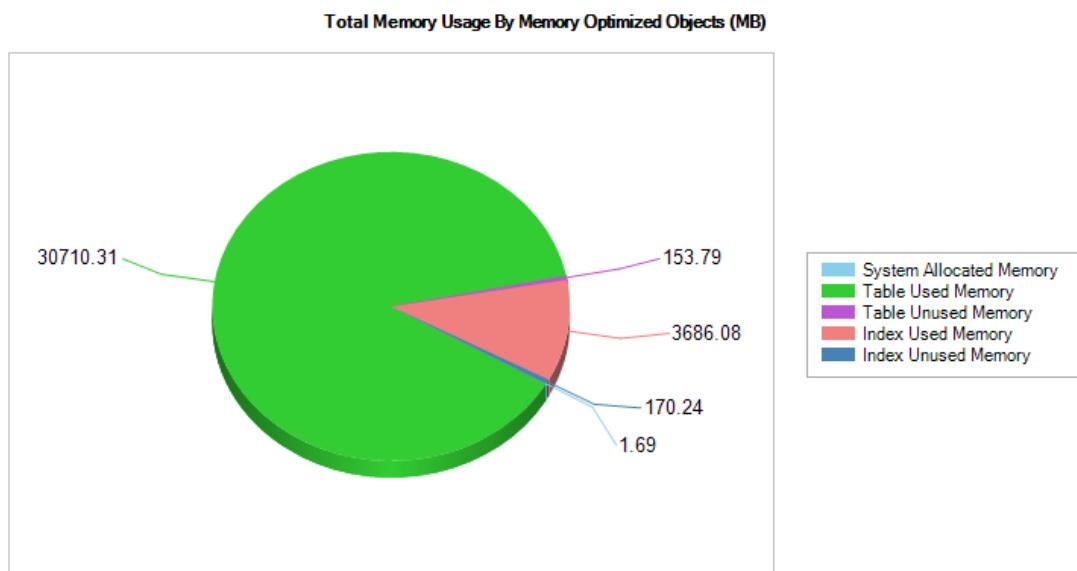
由于内存数据库对内存有着较大的依赖，在管理内存方面，可以考虑使用 Resource governor 来管理内存数据库。需注意如下：

- 通过指定 Resource governor 的 hard limit (如 80%) 来确保其它内部资源和非内存优化表的内存使用量。
- 每个 resource pool 可以包含多个内存数据库，但是一个内存数据库在同一时刻只关联一个 resource pool。

Memory Usage Report 是 SSMS 自带的监控内存使用量的报表，可以快速的查看现有缓存的内存优化对象的使用情况：

This report provides detailed data on the utilization of memory space by memory optimized objects within the Database.

Total Memory Usage By Memory Optimized Objects:	34,722.11 MB
---	--------------



备份在日常维护管理数据库中也极为重要，对持久性内存优化表，内存优化表作为数据库对象中的一部分，被包含在常规数据库备份策略中，故传统的全备、差异备份、日志备份策略无需更改，即可实现对内存优化表的备份。

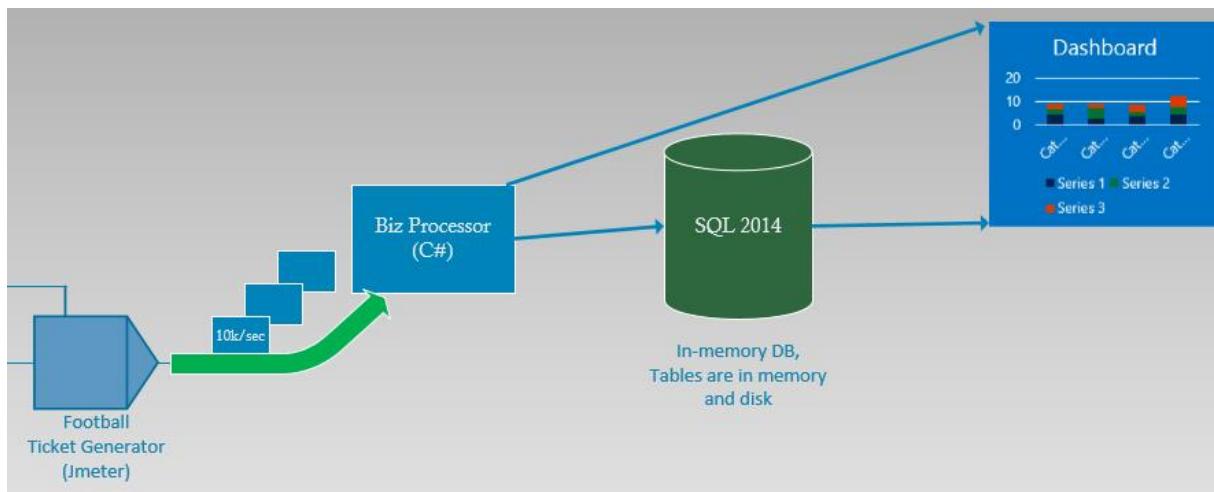
香港赛马会案例参考

有关香港赛马会对于 SQL Server 2014 的采购时，内存数据库的技术验证中的应用场景和性能测试指标，读者可参考下面的文章。

<http://cw.com.hk/news/hk-jockey-club-and-centraline-tap-new-release-sql-server-2014?page=0,0>

技术架构

在技术验证的性能测试中，香港赛马会以每秒处理 10000 的票据，且同时处理生成的 1.5 到 2 倍的赌注，端到端的处理时间在 1 秒以下，内存数据库端的执行时间在毫秒级别。概念验证架构设计如下图所示：



概念验证中主要分为四大模块：

- 票据生成器以每秒 10000 的速度不断的向业务逻辑层发送票据；
- 业务逻辑层通过调用本地存储过程和生成缓存的方式将原始数据转换为票据和投注；
- SQL Server 2014 的内存数据库通过本地编译存储过程向内存优化表插入和查询数据；
- WCF 的 dashboard 通过本地编译存储过程和直接读取逻辑层的缓存数据库将数据以热图、气泡图、线图以及图表的形式展示最新的投注、最高的投资、以及不同渠道的投注情况等等。

参考文档

内存优化表

[http://msdn.microsoft.com/zh-cn/library/dn511014\(v=sql.120\).aspx](http://msdn.microsoft.com/zh-cn/library/dn511014(v=sql.120).aspx)

The Memory Optimized Filegroup

<http://msdn.microsoft.com/en-US/us-en/library/dn639109.aspx>

<http://blogs.technet.com/b/dataplatforminsider/archive/2013/08/01/hardware-considerations-for-in-memory-oltp-in-sql-server-2014.aspx>

High Availability Support for In-Memory OLTP databases

<http://msdn.microsoft.com/en-us/library/dn635118.aspx>

<http://blogs.technet.com/b/dataplatforminsider/archive/2013/11/05/in-memory-oltp-high-availability-for-databases-with-memory-optimized-tables.aspx>

Replication to Memory-Optimized Table Subscribers

<http://msdn.microsoft.com/zh-cn/library/dn600379.aspx>

确定表或存储过程是否应移植到内存中 OLTP

<http://msdn.microsoft.com/zh-cn/library/dn205133.aspx>

内存优化顾问

<http://msdn.microsoft.com/zh-cn/library/dn284308.aspx>

本机编译顾问

<http://msdn.microsoft.com/zh-cn/library/dn358355.aspx>

感谢[马国耀](#)对本文的审校，[刘大玮](#)对本文的策划。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至 editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)）或者腾讯微博（[@InfoQ](#)）关注我们，并与我们的编辑和其他读者朋友交流。

查看原文：[使用 SQL Server 2014 内存数据库时需要注意的地方](#)

相关内容

- [开源 SQL in Hadoop 解决方案：我们处于什么位置？](#)
- [Phoenix：在 Apache HBase 上执行 SQL 查询](#)
- [JetBrains 0xDBE: DBA 和 SQL 开发人员的专属工具](#)
- [8 个值得关注的 SQL-on-Hadoop 框架](#)
- [不要就这么放弃了 SQL](#)

几种线程池的实现算法分析

作者 刘飞

1. 前言

在阅读研究线程池的源码之前，一直感觉线程池是一个框架中最高深的技术。研究后才发现，线程池的实现是如此精巧。本文从技术角度分析了线程池的本质原理和组成，同时分析了 JDK、Jetty6、Jetty8、Tomcat 的源码实现，对于想了解线程池本质、更好的使用线程池或者定制实现自己的线程池的业务场景具有一定指导意义。

2. 使用线程池的意义

- 复用：类似 WEB 服务器等系统，长期来看内部需要使用大量的线程处理请求，而单次请求响应时间通常比较短，此时 Java 基于操作系统的本地调用方式大量的创建和销毁线程本身会成为系统的一个性能瓶颈和资源浪费。若使用线程池技术可以实现工作线程的复用，即一个工作线程创建和销毁的生命周期期间内可以执行处理多个任务，从而总体上降低线程创建和销毁的频率和时间，提升了系统性能。
- 流控：服务器资源有限，超过服务器性能的过高并发设置反而成为系统的负担，造成 CPU 大量耗费于上下文切换、内存溢出等后果。通过线程池技术可以控制系统最大并发数和最大处理任务量，从而很好的实现流控，保证系统不至于崩溃。
- 功能：JDK 的线程池实现的非常灵活，并提供了很多功能，一些场景基于功能的角度会选择使用线程池。

3. 线程池技术要点：

从内部实现上看，线程池技术可主要划分为如下 6 个要点实现：



图 1 线程池技术要点

- **工作者线程 worker:** 即线程池中可以重复利用起来执行任务的线程，一个 worker 的生命周期内会不停的处理多个业务 job。线程池“复用”的本质就是复用一个 worker 去处理多个 job，“流控”的本质就是通过对 worker 数量的控制实现并发数的控制。通过设置不同的参数来控制 worker 的数量可以实现线程池的容量伸缩从而实现复杂的业务需求
- **待处理工作 job 的存储队列:** 工作者线程 workers 的数量是有限的，同一时间最多只能处理最多 workers 数量个 job。对于来不及处理的 job 需要保存到等待队列里，空闲的工作者 work 会不停的读取空闲队列里的 job 进行处理。基于不同的队列实现，可以扩展出多种功能的线程池，如定制队列出队顺序实现带处理优先级的线程池、定制队列为阻塞有界队列实现可阻塞能力的线程池等。流控一方面通过控制 worker 数控制并发数和处理能力，一方面可基于队列控制线程池处理能力的上限。
- **线程池初始化:** 即线程池参数的设定和多个工作者 workers 的初始化。通常有一开始就初始化指定数量的 workers 或者有请求时逐步初始化工作者两种方式。前者线程池启动初期响应会比较快但造成了空载时的少量性能浪费，后者是基于请求量灵活扩容但牺牲了线程池启动初期性能达不到最优。
- **处理业务 job 算法:** 业务给线程池添加任务 job 时线程池的处理算法。有的线程池基于算法识别直接处理 job 还是增加工作者数处理 job 或者放入待处理队列，也有的线程池会直接将 job 放入待处理队列，等待工作者 worker 去取出执行。
- **workers 的增减算法:** 业务线程数不是持久不变的，有高峰期。线程池要有自己的算法根据业务请求频率高低调节自身工作者 workers 的数量来调节线程池大小，从而实现业务高峰期增加工作者数量提高响应速度，而业务低峰期减少工作者数来节

省服务器资源。增加算法通常基于几个维度进行：待处理工作 job 数、线程池定义的最大最小工作者数、工作者闲置时间。

线程池终止逻辑：应用停止时线程池要有自身的停止逻辑，保证所有 job 都得到执行或者抛弃。

4. 几种线程池的实现细节

结合上面的技术点，列举几种线程池实现方式。

- 工作者 workers 与待处理工作队列实现方式举例：

实现	工作者 workers 结构与并发保护	待处理工作队列结构
JDK	使用了 HashSet 来存储工作者 workers，通过可重入锁 ReentrantLock 对其进行并发保护。每个 worker 都是一个 Runnable 接口。	使用了实现接口 BlockingQueue 的阻塞队列来存储待处理工作 job，并把队列作为构造函数参数，从而实现业务可以灵活的扩展定制线程池的队列。业务也可使用 JDK 自身的同步阻塞队列 SynchronousQueue、有界队列 ArrayBlockingQueue、无界队列 LinkedBlockingQueue、优先级队列 PriorityBlockingQueue。
Jetty6	同样使用了 HashSet 存储工作者 workers，通过 synchronized 一个对象进行 HashSet 的并发保护。每个工作者实际上是一个 Thread 的扩展。	使用了数组存储待处理的 job 对象 Runnable。数组初始化容量为 _maxThreads 个，使用变量 _queued 计算保存当前内部待处理 job 的个数即数组 length。超过数组最大值时，扩大 _maxThreads 个容量，因此数组永远够用够大，容量无界。同样是用 synchronized 一个对象的方式实现同步。
Jetty8	使用了 ConcurrentLinkedQueue 存储工作者 workers，利用 JDK 基于 CAS 算法的实现提高了并发效率，同时也降低了线程池并发保护的复杂程度。针对队列 ConcurrentLinkedQueue 无法保证	与 JDK 相同实现，使用了基于接口 BlockingQueue 的阻塞队列来存储待处理工作 job，也支持在线程池构造函数的参数中传入队列类型。同时，Jetty8 内部默认未设置队列类型场景可自动设置使用 2 种队列：有界无法扩容的 ArrayBlockingQueue 及 Jetty

	size()实时性问题引入原子变量 AtomicInteger 统计工作者数量。	自身定制扩展实现的可扩容队列 BlockingArrayQueue。
Tomcat	基于 JDK 的 ThreadPoolExecutors 实现，复用 JDK 业务	复用 JDK 业务

- 线程池初始化与处理业务 job 算法举例：

实现	线程池构造与工作者初始化	处理业务 job 的算法
JDK	<p>1. 基于多个构造参数实现灵活初始化，几个核心参数如下：</p> <p>corePoolSize: 核心工作者数</p> <p>maximumPoolSize: 最大工作者数</p> <p>keepAliveTime: 超过核心工作者数时闲置工作者的存活时间。</p> <p>workQueue: 待处理 job 队列，即前面提到的 BlockingQueue 接口。</p> <p>2. 默认初始化后不启动工作者，等待有请求时才启动。可以通过调用线程池接口提前启动核心工作数个工作者线程，也可以启动业务期望的多个工作者线程。</p>	<p>1. 工作者 workers 数量低于核心工作者数 corePoolSize 时会优先创建一个工作者 worker 处理 job，处理成功则返回。</p> <p>2. 工作者 workers 数量高于核心工作者数时会优先把 job 放入到待处理队列，放入队列成功时处理结束。</p> <p>3. 步骤 2 中入队失败会识别工作者数是否还小于最大工作者数 maximumPoolsize，小于的话也会新创建一个工作者 worker 处理 job。</p> <p>4. 拒绝处理</p>
Jetty6	<p>1. 同样支持设置多个参数：</p> <p>_spawnOrShrinkAt: 扩容/缩容阀值</p> <p>_minThreads: 最小工作者数</p> <p>_maxThreads: 最大工作者数</p> <p>_maxIdleTimeMs: 闲置工作者最大闲置超时时间</p>	<p>1. 查找闲置的工作者 worker，找到则派发 job。</p> <p>2. 没有闲置的工作者，将 job 存入待处理数组。</p> <p>3. 当识别到数组中待处理 job 超过扩容阀值参数时，扩容增加工作者处理 job</p>

	2. 初始化后直接启动_minThreads个工作者线程	4. 否则不处理
Jetty8	1. 配置参数类似 Jetty6，去除了_spawnOrShrinkAt 阈值参数。 2. 初始化后直接启动_minThreads个工作者线程	非常简单，直接将待处理 job 入队。
Tomcat	1. 基于 JDK 线程池的构造方法 2. 来请求时启动工作者	处理方法复用 JDK 的，但是在开始提交前扩展了 JDK 的功能，实现了可以统计提交数 submittedCount 的能力

- 线程池工作者 worker 的增减机制举例：

实现	工作者增加算法	工作者减少算法
JDK	1. 待处理 job 来时，工作者 workers 数量低于核心工作者数 corePoolSize 时。 2. 待处理 job 来时，workers 数超过核心数小于最大工作者数且入待处理队列失败场景。 3. 业务调用线程池的更新核心工作者数接口时，若发现扩容，会增加工作者数。	1. 待处理任务队列里没有 job 并且工作者 workers 数量超过了核心工作者数 corePoolSize。 2. 待处理任务队列里没有 job 并且允许工作者数量小于核心工作者参数为 true，此场景会至少保留一个工作者线程。
Jetty6	1. 启动线程池时会启动_minThreads 个工作者线程 2. 待处理的 job 数量高于了阈值参数且工作者数没有达到最大值时会增加工作者。	如下三个条件同时满足时会减少工作者： 1. 待处理任务数组中没有待处理 job 2. 工作者 workers 数量超过了最小工作者数_minThreads 3. 闲置工作者线程数高于了阀值参数

	3. 调用线程池接口 <code>setMinThreads</code> 更新最小工作者数时会根据需要增加工作者。	
Jetty8	1. 启动线程池时启动最小工作者参数个工作者线程 2. 已经没有闲置工作者或者闲置工作者的数量已经小于待处理的 job 的总数 3. 调用线程池接口 <code>setMinThreads</code> 更新最小工作者数时	如下三个条件同时满足时会减少工作者： 1. 待处理任务队列里没有待处理的 job 2. 工作者 workers 总数超过了最小工作者参数配置 <code>_minThreads</code> 3. 工作者线程的闲置时间超时
Tomcat	同 JDK 增加工作者算法	复用 JDK 减少算法，同时定制扩展延迟参数，超过参数时，直接抛出异常到外面来终止线程池工作者。

5. 小结

对比几种线程池实现，JDK 的实现是最为灵活、功能最强且扩展性最好的，Tomcat 即基于 JDK 线程池功能扩展实现，复用原有业务的同时扩充了自己的业务。Jetty6 是完全自己定制的线程池业务，耦合线程池众多复杂的业务逻辑到线程池类里面，逻辑相对最为复杂，扩展性也非常差。Jetty8 相对 Jetty6 的实现简化了很多，其中利用了 JDK 中的同步容器和原子变量，同时实现方式也越来越接近 JDK。

6. 参考源码

- JDK 源码类： `java.util.concurrent.ThreadPoolExecutor`
- Jetty6 源码类： `org.mortbay.thread.QueuedThreadPool`
- Jetty8 源码类： `org.eclipse.jetty.util.thread.QueuedThreadPool`
- Tomcat 源码类： `org.apache.tomcat.util.threads.ThreadPoolExecutor`

感谢郭蕾对本文的审校。

查看原文：[几种线程池的实现算法分析](#)

HBase 高性能复杂条件查询引擎

作者 耿立超

写在前面

在这次的审稿过程中有幸得到了 Ted Yu 和梁堰波先生的反馈，大家就一些感兴趣的内容进行了讨论。该方案由一个智能交通解决方案演变而来，设计之初仅寄希望于通过二级索引提升查询性能，由于在前期架构时充分考虑了通用性以及对复杂条件的支持，在后来的演变中逐渐被剥离出来形成了一个通用的查询引擎。Ted Yu 对“查询决策器”表示了关心，他指出类似的组件同时也是 Phoenix, Impala 用于支持 SQL 查询的核心组件，但是这类组件很难引入到 HBase 中，因为 HBase 专注于 byte[] 的操作。对此，方案在设计时避开了“SQL 解析”和“在各种数据类型与 byte[] 之间进行转化”的棘手问题，而是使用了一组可以描述查询的 Query API，这与 Hibernate 中提供 Criteria 接口的做法非常相似，在 Hibernate 中既支持 HQL 语句的查询又支持使用 Criteria 接口以编程方式描述的查询，对于我们来说选择类似后者的做法实现起来要快速和容易的多，而查询条件中的值在构造之初就以 byte[] 的形式传递，避免了决策器解析时的类型判定和转化问题。

题记

——索引的实质是另一种编排形式的数据冗余，高效的检索源自于面向查询特别设计的编排形式，如果再辅以分布式的计算框架，就可以支撑起高性能的大数据查询。

正文

Apache HBase™是一个分布式、可伸缩的 NoSQL 数据库，它构建在 Hadoop 基础设施之上，依托于 Hadoop 的迅猛发展，HBase 在大数据领域的应用越来越广泛，成为目前 NoSQL 数据库中表现最耀眼，呼声最高的产品之一。像其他 NoSQL 数据库一样，HBase 也有其适用范围，就应对复杂条件的查询来说，一般认为它并不是非常适合，熟悉 HBase 的开发人员对此应该有一定的体会，但是基于普遍的需求，开发者们希望 HBase 在保持高性能优势的同时能对复杂条件的查询给予一定的支持，而本文将要介绍的正是一种在 HBase 现行机制下以非侵入式实现的基于二级多列索引的高性能复杂条件查询引擎。

问题

目前 HBase 主要应用在结构化和半结构化的大数据存储上，其在插入和读取上都具有极高的性能表现，这与它的数据组织方式有着密切的关系，在逻辑上，HBase 的表数据按 RowKey 进行字典排序，RowKey 实际上是数据表的一级索引（Primary Index），由于 HBase 本身没有二级索引（Secondary Index）机制，基于索引检索数据只能单纯地依靠 RowKey，为了能支持多条件查询，开发者需要将所有可能作为查询条件的字段一一拼接到 RowKey 中，这是 HBase 开发中极为常见的做法，但是无论怎样设计，单一 RowKey 固有的局限性决定了它不可能有效地支持多条件查询。通常来说，RowKey 只能针对条件中含有其首字段的查询给予令人满意的性能支持，在查询其他字段时，表现就差强人意了，在极端情况下某些字段的查询性能可能会退化为全表扫描的水平，这是因为字段在 RowKey 中的地位是不等价的，它们在 RowKey 中的排位决定了它们被检索时的性能表现，排序越靠前的字段在查询中越具有优势，特别是首位字段具有特别的先发优势，如果查询中包含首位字段，检索时就可以通过首位字段的值确定 RowKey 的前缀部分，从而大幅度地收窄检索区间，如果不包含则只能在全体数据的 RowKey 上逐一查找，由此可以想见两者在性能上的差距。

受限于单一 RowKey 在复杂查询上的局限性，基于二级索引（Secondary Index）的解决方案成为最受关注的研究方向，并且开源社区已经在这方面已经取得了一定的成果，像 ITHBase、IHBase 以及华为的 hindex 项目，这些产品和框架都按照自己的方式实现了二级索引，各自具有不同的优势，同时也都有一定局限性，本文阐述的方案借鉴了它们的一些优点，在确保非侵入的前提下，以高性能为首要目标，通过建立二级多列索引实现了对复杂条件查询的支持，同时通过提供通用的查询 API，以及完全基于配置的索引结构，完全封装了索引的创建和使用细节，使之成为一种通用的查询引擎。

原理

“二级多列索引”是针对目标记录的某个或某些列建立的“键-值”数据，以列的值为键，以记录的 RowKey 为值，当以这些列为条件进行查询时，引擎可以通过检索相应的“键-值”数据快速找到目标记录。由于 HBase 本身并没有索引机制，为了确保非侵入性，引擎将索引视为普通数据存放在数据表中，所以，如何解决索引与主数据的划分存储是引擎第一个需要处理的问题，为了能获得最佳的性能表现，我们并没有将主数据和索引分表储存，而是将它们存放在了同一张表里，通过给索引和主数据的 RowKey 添加特别设计的 Hash 前缀，实现了在 Region 切分时，索引能够跟随其主数据划归到同一 Region 上，即任意 Region 上的主数据其索引也必定驻留在同一 Region 上，这样我们就能够把从索引抓取目标主数据的性能损失降低到最小。与此同时，特别设计的 Hash 前缀还在逻辑上把索引与主数据进行了自动的分离，当全体数据按 RowKey 排序时，排在前面的都是索引，我们称之为索引区，排在后面的均为主数据，我们称之为数据区。最后，通过给索引和主数据分配不同的 Column Family，又在物理存储上把它们隔离了起来。逻辑和物理上的双重隔离避免了将两类数据存放在同一张表里带来的副作用，防止了它们之间的相互干扰，降低了数据维护的复杂性，可以说这是在性能和可维护性上达到的最佳平衡。

Sample表 -> Region 1 (RowKey:0000-0099)

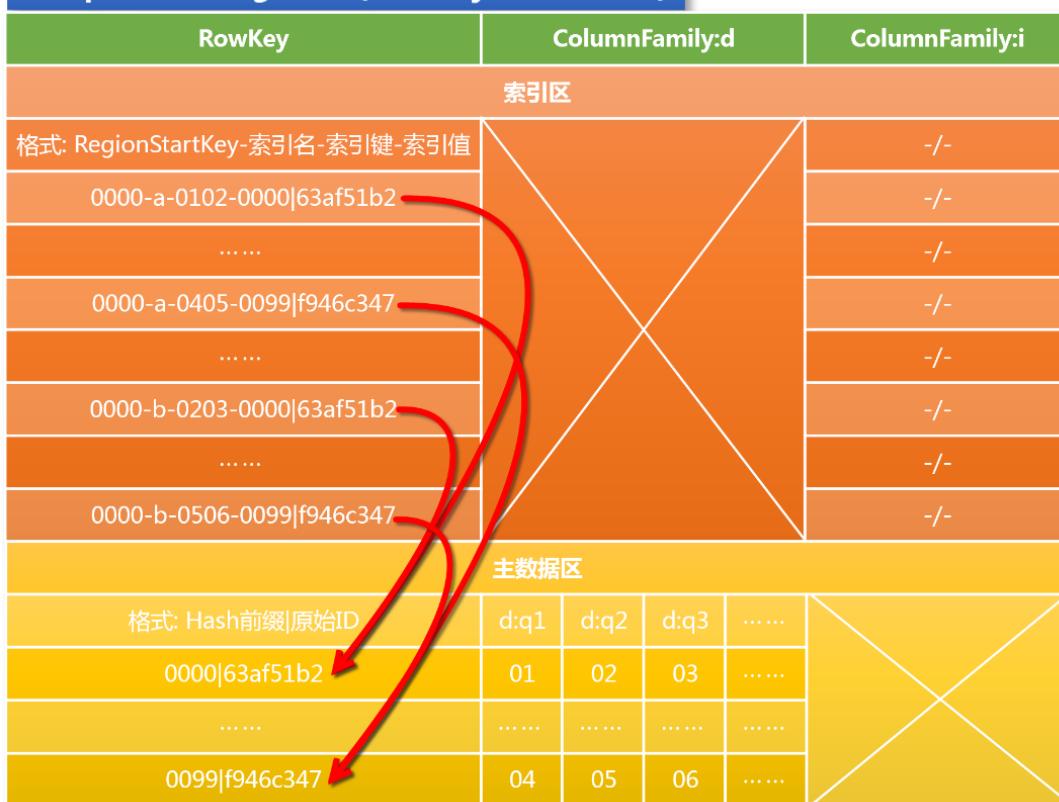


图 1: Sample 表 Region 1 的数据逻辑视图

让我们通过一个示例来详细了解一下二级多列索引表的结构，假定有一张 Sample 表，使用四位数字构成 Hash 前缀[iii]，范围从 0000 到 9999，规划切分 100 个 Region，则 100 个 Region 的 RowKey 区间分别为[0000,0099], [0100,0199], ……, [9900,9999]，以第一个 Region 为例，请看图 1，所有数据按 RowKey 进行字典排序，自动分成了索引区和主数据区两段，主数据区的 Column Family 是 d，下辖 q1,q2,q3 等 Qualifier，为了简单起见，我们假定 q1,q2,q3 的值都是由两位数字组成的字符串，索引区的 Column Family 是 i，它不含任何 Qualifier，这是一个典型的“Dummy Column Family”，作为区别于 d 的另一个 Column Family，它的作用就是让索引独立于主数据单独存储。接下来是最重要的部分，即索引和主数据的 RowKey，我们先看主数据的 RowKey，它由四位 Hash 前缀和原始 ID 两部分组成，其中 Hash 前缀是由引擎分配的一个范围在 0000 到 9999 之间的随机值，通过这个随机的 Hash 前缀可以让主数据均匀地散列到所有的 Region 上，我们看图 1，因为 Region 1 的 RowKey 区间是[0000,0099]，所以没有任何例外，凡是且必须是前缀从 0000 到 0099 的主数据都被分配到了 Region 1 上。接下来看索引的 RowKey，它的结构要相对复杂一些，格式为：RegionStartKey-索引名-索引键-索引值，与主数据不同，索引 RowKey 的前缀部分虽然也是由四位数字组成，但却不是随机分配的，而是固定为当前 Region 的 StartKey，这是非常重要而巧妙的设计，一方面，这个值处在 Region 的 RowKey 区间之内，它确保了索引必定跟随其主数据被划分到同一个 Region 里；另一方面，这个值是 RowKey 区间内的最小值，这保证了在同一

Region 里所有索引会集中排在主数据之前。接下来的部分是“索引名”，这是引擎给每类索引添加的一个标识，用于区分不同类型的索引，图 1 中展示了两种索引：a 和 b，索引 a 是为字段 q1 和 q2 设计的两列联合索引，索引 b 是为字段 q2 和 q3 设计的两列联合索引，依次类推，我们可以根据需要设计任意多列的联合索引。再接下来就是索引的键和值了，索引键是由目标记录各对应字段的值组成，而索引值就是这条记录的 RowKey。

现在，假定需要查询满足条件 $q1=01$ and $q2=02$ 的 Sample 记录，分析查询字段和索引匹配情况可知应使用索引 a，也就是说我们首先确定了索引名，于是在 Region 1 上进行 scan 的区间将从主数据全集收窄至 $[0000-a, 0000-b)$ ，接着拼接查询字段的值，我们得到了索引键：0102，scan 区间又进一步收窄为 $[0000-a-0102, 0000-a-0103)$ ，于是我们可以很快地找到 0000-a-0102-0000|63af51b2 这条索引，进而得到了索引值，也就是目标数据的 RowKey：0000|63af51b2，通过在 Region 内执行 Get 操作，最终得到了目标数据。需要特别说明的是这个 Get 操作是在本 Region 上执行的，这和通过 HTable 发出的 Get 有很大的不同，它专门用于获取 Region 的本地数据，其执行效率是非常高的，这也是为什么我们一定要将索引和它的主数据放在同一张表的同一个 Region 上的原因。

架构

在了解了引擎的工作原理之后来我们来看一下它的整体架构：

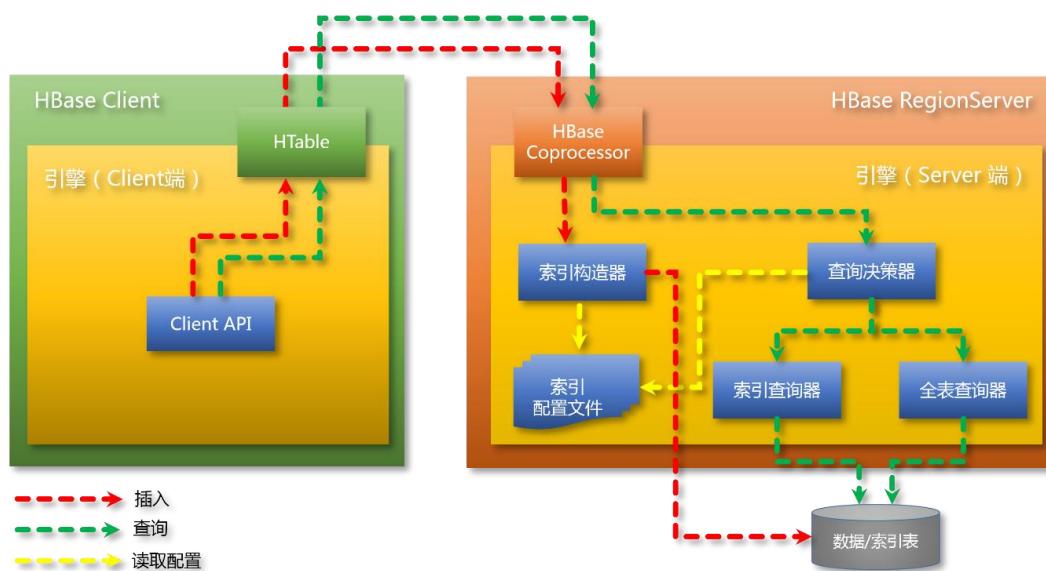


图 2：引擎的整体架构

引擎构建在 HBase 的 Coprocessor 机制之上，由 Client 端和 Server 端两部分构成，对于查询而言，查询请求从 Client 端经由 HTable 的 coprocessorExec 方法推送到所有的 RegionServer 上，RegionServer 接收到查询请求后使用“查询决策器”分析查询条件，

比对索引元数据，在找到适合该查询的最优索引后，解析索引区间，然后委托“索引查询器”基于给定的最优索引和解析区间进行数据检索，如果没有找到合适的索引则委托“全表查询器”进行全表扫描。当各 RegionServer 的局部查询结果返回之后，引擎的 Client 端还负责对它们并进行合并汇总和排序，从而得到最终的结果集。对于插入而言，当主数据试图写入时会被 Coprocessor 拦截，委托“索引构造器”根据“索引配置文件”创建指向当前主数据的所有索引，然后一同插入到数据表中。

让我们来深入了解一下引擎的几个核心组件。对于引擎的客户端来讲，最重要的组件是一套用于表达复杂查询请求的 Query API，在这套 API 的设计上我们借鉴了 IHBase 的一些做法，通过对查询条件（Condition）进行抽象和建模，得到一套典型的基于“复合模式”（Composite Pattern）的 Class Hierarchy，使之能够优雅地表达基于 AND 和 OR 的多重复合条件。以图 1 所示的 Sample 表为例，使用 Query API 构造一个查询条件为“ $(q1=01 \text{ and } q2 < 02) \text{ or } (q1=03 \text{ and } q2 > 04)$ ”的 Java 代码如下：

```

private static Query createQuery(){
    Query query = new Query();
    query.setTable("Sample");
    Condition condition = Condition.or(
        Condition.and(
            Condition.condition("d", "q1", Operator.EQ, "01"),
            Condition.condition("d", "q2", Operator.LT, "02")
        ),
        Condition.and(
            Condition.condition("d", "q1", Operator.EQ, "03"),
            Condition.condition("d", "q2", Operator.GT, "04")
        )
    );
    query.setCondition(condition);
    return query;
}

```

图 3：引擎客户端的 Query API 示意代码

查询请求到达 Server 端以后，由 Coprocessor 委派查询决策器进行分析以确定使用何种查询策略应对，这是查询处理流程上的一个关键结点。查询决策器需要分析查询请求的各项细节，包括条件字段、排序字段和排序，然后和索引的元数据进行比对找出性能最优的索引，有时候对于一个查询请求可能会有多个适用索引，但是查询性能却有高下之分，因此需要对每一个候选索引进行性能评估，找出最优者，性能评估的方法是看哪个索引能最大限度地收窄检索区间。索引的元数据来自于索引配置文件，图 4 展示了一份简单的索引配置，配置中描述的正是图 1 中使用的索引 a 和 b 的元数据，索引元数据主要是由索引名和一组 field 组成，field 描述的是索引针对的目标列

（ColumnFamily:Qualifier）。实际的索引配置通常比我们看到的这份要复杂，因为在生成索引时有很多细节需要通过索引配置给出指引，比如如何处理不定长字段，目标列使用正序还是倒序（例如时间数据在 HBase 中经常需要按补值进行倒序处理），是否需要使用自定义格式化器对目标列的值进行格式化等等，完全配置化的索引元数据使创建和维护索引的成本大大降低，为上层应用根据实际需求灵活设计索引提供了保障。

```

<?xml version="1.0" encoding="UTF-8"?>
<indexes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="index-conf.xsd">
    <index>
        <name>a</name>
        <table>Sample</table>
        <fields>
            <field columnFamily="d" qualifier="q1" length="2"/>
            <field columnFamily="d" qualifier="q2" length="2"/>
        </fields>
    </index>
    <index>
        <name>b</name>
        <table>Sample</table>
        <fields>
            <field columnFamily="d" qualifier="q2" length="2"/>
            <field columnFamily="d" qualifier="q3" length="2"/>
        </fields>
    </index>
</indexes>

```

图 4：一份简单的索引配置文件

在确定最优索引之后，查询决策器开始基于最优索引对查询条件进行解析，解析的结果是一组索引区间，区间内的数据未必都满足查询条件，但却是通过计算所能得到的最小区间，索引查询器就在这些区间上进行检索，通过配备的专用 Filter 对区间内的每一条数据进行最后的匹配判断。图 5 展示了一个条件为 $q1=01 \text{ and } 01 \leq q2 \leq 03$ 的查询请求在 Sample 表 Region 1 上的解析和执行过程。

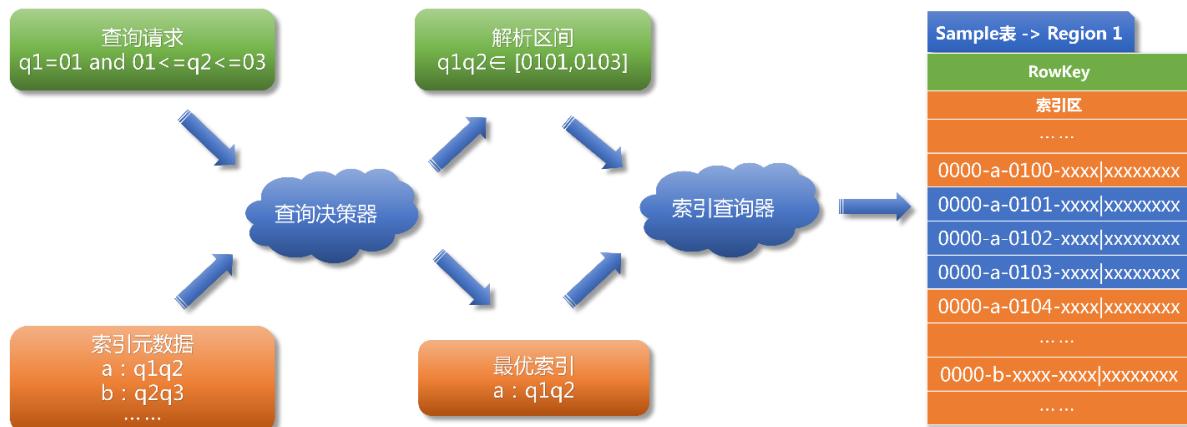


图 5：查询请求 $q1=01 \text{ and } 01 \leq q2 \leq 03$ 在 Sample 表 Region 1 上的解析和执行过程示意

对于那些找不到索引的查询请求来说，查询决策器将委派全表查询器处理，全表查询器将跳过索引区，从主数据区开始通过配备的专用 Filter 进行全表扫描。显然，相对于索引查询，全表扫描的执行效率是很低的，它的存在是为了在所有索引都不适用的情况下起“托底”作用，以此保证任意复杂条件的查询都能得到处理，所以这里引出一个非常重要的问题，就是在索引查询和全表扫描之间的选择与权衡问题。通常人们总是希望所有的查询都越快越好，虽然从理论上讲建立覆盖任意条件查询的索引是可能的，但这是不现实的，因为创建索引是有代价的，除了占用大量的存储空间之外还会影响到数据插入的性能，所以不能无节制地创建索引，理性的做法是分析并筛选出最为常用的查询，针对这些查询建立相应的索引，优化查询性能，而对于那些较为“生僻”的查询则使用全表扫描的方式进行处理，以此在存储成本、插入性能和查询性能之间找到一种理想的平衡。最后要补充说明的是，不管是使用索引查询还是进行全表扫描，这些动作都是通过 Coprocessor 机制分发到所有 Region 上去并发执行的，即使是全表扫描其性能也将远超过 HBase 原生的 Scan 操作！

应用

由于引擎设计之初就以非侵入性为前提，所以引擎的部署与集成就与引入第三方类库无异，唯一需要上层应用提供的是面向数据表的索引配置文件。设计索引主要以业务需求为导向，先分析并梳理出常用的查询用例，然后针对查询用例所涉及的字段和排序要求按相似性进行分组，尽可能让单个索引同时支持多种相近的查询，减少索引的种类和数量，提升索引复用率。在这方面如下设计原则可供参考（注：以下原则均以“不考虑排序”为前提）：

- N 个字段组合的查询只需要建立一个包含该 N 个字段的索引，建立按这个 N 字段其他顺序排列的索引是没有意义的。因此，以 N 个字段组合为条件的查询只需要 $C(n, n)=1$ 个索引。
- 一个包含 N 个字段的索引同时是以从第 1 到第 N-1 个字段为条件的查询索引，以及从第 1 到第 N-2 个字段为条件的查询索引，依此类推，也是仅以第 1 个字段为条件的查询索引。因此，包含 N 个字段的索引总计可以支持 $C(n, 1)=n$ 种查询组合。
- 基于上述两点，任意一个索引的字段组合不应该是另一个索引字段组合的前缀部分，这样设计的索引才会有较高的复用率。

假如某表有 A、B、C、D 四个字段，在不考虑排序的前提下，如果要用索引支持以任意字段或字段组合为条件的查询，则索引的设计方法如下：四字段索引只需要一个，假定取 ABCD（它将同时支持 ABCD、ABC、AB 和 A 四种查询）。三字段索引分别以 A、B、C、D 开头向后循环取足三个字段，得到：ABC、BCD（它将同时支持 BCD、BC 和 B 三种查询）、CDA（它将同时支持 CDA、CD 和 C 三种查询）和 DAB（它将同时支持 DAB、DA 和 D 三种查询），其中 ABC 是 ABCD 的前缀，故舍弃。按照同样的方法，两字段索引要分别从保留下来的三个三字段索引中依次以每一个字段开头取足两个字段，然后去除重复和前缀重叠的索引，最终得到 DB（它将同时支持 DB 和 D 两种查询）和 AC（它将同时支持 AC 和 A 两种查询），总计是 6 个索引，最后可以根据实际需求剪裁掉不需要的索引。

在上述原则的表述中特别注明了“不考虑排序”这个前提，对于索引来说，“排序”是一个很“敏感”的要求，索引本身只有一种排序（即按索引首字段进行的字典排序），如果查询请求的排序与索引排序不同，则索引直接出局，即使它们的字段完全匹配，也就是说排序会极大地消弱索引的复用度，对于我们的引擎来说，排序字段应该受到严格的控制。实际上，很多大数据系统都需要对排序进行限制，比如淘宝上的商品检索，可供排序的字段只有人气、销量、信用和价格，因为排序需要针对数据全集进行计算，如果不是针对有限的排序字段建立索引或是离线计算并缓存结果，按任意字段排序的查询是很难在线返回的。

小结

综合前文所述，方案主要有如下几个显著的优势：

1. 高性能：引擎的高性能源自两方面，一是二级多列索引，二是基于 Coprocessor 的并行计算
2. 非侵入性：引擎构建在 HBase 之上，既没有对 HBase 进行任何改动，也不需要上层应用做任何妥协
3. 高度可配置：索引元数据是完全基于配置的，可以轻便灵活地创建和维护索引
4. 通用性：引擎的前端查询接口和后端索引处理都是基于通用目标设计的，不依赖于任何具体表

限于 HBase 自身的特点，方案本身也有一定的局限性，一是它不能随意地支持任意的条件查询，这一点前文已经给出了分析和建议，二是在插入主数据时需要伴随插入多份索引从而对写入性能产生了一定的影响，如何控制写入和查询的竞争关系需要根据系统的读写比进行权衡，对于数据写入实时性要求不高或者数据是离线导入的系统来说，可以考虑使用批量导入工具，特别是以直接生成 HFile 的方式导入的话可以在很大程度上消除引入索引后的写入压力。

[1] 理论上基于 HBase 的 Filter 机制可以实现任意复杂条件的查询，但是那样做就彻底放弃了 RowKey 作为索引的利用价值，大多数查询的性能都将变得非常差。

[2] Hash 前缀的长度和 Region 数量有着密切的关系，由于索引和主数据的分配高度依赖 RowKey 前缀和 Region 的 RowKey 区间，引擎严禁 Region 进行自动切分，开发人员需要在前期对 Region 数量和前缀长度进行规划，本例中取四位前缀意味着最多可以支持 10000 个 Region。

关于作者

耿立超，架构师，CSDN 博客专家，博客 <http://blog.csdn.net/bluishglc> 目前正从事大数据领域的研发工作，对企业级应用架构、SaaS、分布式存储和领域驱动设计有丰富的实践经验，喜欢摄影和旅行。

感谢 [Ted Yu](#) 对本文的审校，[包研](#)对本文的策划。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至 editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)）或者腾讯微博（[@InfoQ](#)）关注我们，并与我们的编辑和其他读者朋友交流。

查看原文：[HBase 高性能复杂条件查询引擎](#)

相关内容

- [Phoenix：在 Apache HBase 上执行 SQL 查询](#)
- [HBase ORM SimpleHBase 设计](#)
- [HBase 0.98 引入了基于单元格的安全](#)
- [HBase 优化案例分析：Facebook Messages 系统问题与解决方案](#)
[Facebook 的 HBase 解决方案](#)



营销能力



广告平台

分发能力



开放平台



分发渠道



公众账号

运营能力



腾讯云分析



信鸽推送



关键因子



大数据

基础能力



云服务器



云数据库



对象存储



弹性web引擎



负载均衡



云存储



云硬盘



云安全



云监测



移动加速

腾讯云简介

腾讯云致力于打造最高质量、最佳生态的公有云服务平台。基于QQ、微信、QQ空间、腾讯游戏等海量业务的技术架构和精细化互联网运营经验，腾讯云为广大企业和开发者提供云计算、云数据、云运营等一体化云端服务能力，助力企业建立灵活高效的IT架构，轻松连接未来。腾讯云提供的产品安全可靠、稳定易用，包括云服务器、云存储、云数据库和弹性web引擎等基础云计算服务以及腾讯云分析（MTA）、腾讯云推送（信鸽）等大数据运营服务。针对不同领域独特需求，腾讯云还推出一系列行业解决方案，例如微信解决方案、游戏解决方案、移动应用解决方案、视频解决方案等等。云端生态，价值共享，了解更多腾讯云信息，请见：<http://www.qcloud.com/>



O2O解决方案

聚合最丰富微信服务商，打造品牌
微信公众号，助力企业微信服务



游戏解决方案

依托腾讯雄厚技术基础，打造游戏
一站式解决方案，为游戏成功助力



移动解决方案

汇聚移动专属能力，助力移动开
发者轻松构建和运行应用程序



云端生态·价值共享
www.qcloud.com



腾讯云刘颖：块存储深度剖析

受访者 刘颖 作者 刘宇

个人简介 刘颖，腾讯云总架构师，先后从事过外网统一接入网关，流量实时监控和在线回放系统，CDN，高性能防火墙，虚拟化调度管理平台，云应用引擎(CEE)，分布式网络块设备的设计和研发，以及Linux TCP/IP 协议栈和内存管理优化。致力于建设高性能、高可用、安全的云平台基础设施。

QCon 是由 InfoQ 主办的全球顶级技术盛会，每年在伦敦、北京、东京、纽约、圣保罗、杭州、旧金山召开。自 2007 年 3 月份首次举办以来，已经有包括传统制造、金融、电信、互联网、航空航天等领域的近万名架构师、项目经理、团队领导者和高级开发人员参加过 QCon 大会。

InfoQ：大家好，我是 InfoQ 的社区编辑刘宇，也是 Puppet 实战作者。今天非常荣幸邀请到腾讯云的总架构师刘颖来做本次专访。首先请刘颖介绍一下腾讯云目前的一些情况以及未来的发展方向。

刘颖：腾讯云是 2013 年 9 月 9 号正式对外开放的，即对所有的开发者进行服务。其实在 9 月 9 号之前，它一直服务于腾讯的开放平台，在腾讯的开放平台上，也成长出了很多优秀的开发商，包括游戏、工具、电商等各种应用。经过开放平台几年的磨炼，也达到了比较高的可用性，也积累了比较丰富的运营大型云平台的经验，因此今年 9 月 9 号，我们决定全面开放。

我们云的特点大概是这样的，主要是 IaaS，也是一个 PASS 平台，它提供云主机，SQL、NoSQL、分布式块存储；也提供腾讯一些特有的服务，比如移动加速，提高移动网络的可用性和可靠性以及延时，腾讯云推动，帮助海量消息妙计触达终端。除了各种基础的服务，也针对了游戏、微信等行业提供了一个比较完整的解决方案，让开发者更关注于应用逻辑，其它的事情云平台都帮它搞定。

腾讯云未来除了要进一步的提高服务的可用性，还将更深入的去理解一些垂直行业，做出更符合用户需求的云服务。

InfoQ：关于腾讯的块存储方面，能否从它的特点和功能方面再展开谈一下？

刘颖：块存储的表现形式是磁盘，分布式的存储在网络上的各个存储节点，所以叫网络块存储。用户可以在快存储上面，搭建任何存储的类型。比如 Mysql，Memcached，或者 MongoDB，非常灵活，就像本地磁盘一样。

另外，它的特点主要有两个：第一，它的性能，特别是随机的性能是比较好的，如果是

本地磁盘的话，它的随机性能取决于块磁盘的极限的性能，如果在网络上，可以把磁盘打散存放，一个磁盘可以分布在若干个存储节点上面，这样可以达到并发的读和写的能力，所以说对随机类型的读写，它的性能优势是比较高的。第二，存储可以通过几个副本的方式，或者是一些冗余编码的方式来达到一个比较高安全性。另外，建了一个专门的存储网络，针对块存储这种大快网络报文的访问形式，延时更低，吞吐量更高。

InfoQ：在随机读写方面是如何去保证，有哪些技术手段？如何对比数据的？

刘颖：网络上延时上的开销，这是网络存储里面，必须要面对的一个问题。首先我们可以针对存储的特点去把这个网络变得更高效。磁盘读和写的平均块是比较大的，我们可以把这个网络的 MTU 做得更大，也就是说减少报文来回的次数，通过这样一种方式，降低网络的开销。其次，我们还可以在本地磁盘通过 Cache 的方式，让一些读写不经过网络。第三，可以针对后端的存储做分级。我们可以做一层 SSD，或者内存的 Cache，用户的写，直接就写在 Cache 上面，针对一些具体的 IO 的类型，比如说如果是一个顺序的读，我们可以采用预读的方式，顺序的写，可以通过适当的 I/O 合并来达到一个更好的吞吐量。

InfoQ：你们现在做块的存储，整个性能的差异和传统做法的区别是什么？

刘颖：其实传统的做法可能更多是一些专用的硬件，他们通过一些光纤的方式来做，成本比较高，而且它的可扩展性也不是很好。而我们是基于通用服务器，这种低成本高性能的方式，这对比传统方式而言，优势更加明显。

InfoQ：如果后端做这种分布式存储的情况下，在同一个机房之内，这些服务器你们不是采用的光纤吗？

刘颖：我们在机房之间会有一个专门的光纤，至少是以 10G 为单位，跟通常是几十 G 或者是上百 G 高速的一个网络，用于机房之间同步数据的多个副本，这样能保证整个数据更高的安全性和可用性。但是在机房内部，可能更多就是不会用专门的光纤去搭这种存储的网络，因为这个成本会很高。

InfoQ：分布式的块存储和我们所了解的淘宝的 TFS 是类似的这种架构吗？

刘颖：可能有不同的呈现形式，有的分布式文件系统，可能对外呈现的是一种文件的表示方式；还有的是 REST 的表现方式；还有一些是专用的，自己特定的一些接口，可能对外呈现的形式不一样，当然路径也会不一样。如果对外呈现的接口不一样，它可能中间会经过一些自己的路径。而块存储直接给用户呈现的是一个块，输入和 TFS 是不一样的，最终可以以文件的方式去落地，大家的选择不一样，落地方式就会不一样。

InfoQ：在块存储的安全方面是如何做好保障的？

刘颖：其实我们是完全打散的，后端有一个大的存储池。用户的块可能存在于任何一个地方，而不是说固定的把后端某几台机器的某几个块划给他，而是随机的，用户来了一个块，是根据当时后端存储每一个 load 节点负载的情况来进行选择。不会说事先就会放好一个一一的对应关系，不是一个静态的，是动态的。中间可能会根据整个集群的负载甚至做一些搬迁来保证集群的负载尽量是均衡的。对于安全性而言，首先监控要做好，你需要知道每一块盘它的 IO 是什么样的，是不是符合预期的，是过大还是什么样。如果是过大了，我们要采用一些留空的方式，本质上就是一个 I/O 的 QoS，如果是发现这个盘的 I/O 过大，它会做一些限流的方式，把这个 I/O 让它慢下来。当整个集群还不是太忙的只有，它可以用得多一点，当集群变得比较忙的时候，就会对其中一些用户，特别是使用量很大的用户，会让他的 I/O 慢下来，来保证整个集群的安全。

InfoQ：除了 I/O 之外，还有其他地方需要考虑的吗？

刘颖：当然也要考虑到网络的情况，因为所有的 I/O 都是通过网络的方式到达后端的。所以说我们也要去做对用户的 I/O 带宽，也就是网络做很好的监控，从用户发起的地方源头，到最后端整个路径上每一个链条上都要做监控，让它没有瓶颈。监控是一个前期，如果说发现了，通过一些在线的扩容的方式，或者是限流的方式，让整个平台的安全性得到保障。

InfoQ：腾讯在监控方面运用了哪些技术？数据采集的力度如何？

刘颖：监控方面基本上我们统计是实时的，就是说用户的每一个请求都会记录我们的统计里面。数据是一个实时的统计，并非一个采样的统计。每一个请求，都会经过采集和统计的一个模块。它会有两个维度，第一个是针对每一个盘的维度，另外是针对整个集群里面的维度，不同的维度，都会有不同的统计。每一个维度，超过限制的时候，它都会报警出来。里面有节点故障了，通过主动探测和 watch 的方式，及时的屏蔽，集群的容量或者整体性能不够，动态添加存储节点。

InfoQ：最后，关于块存储方面的内容，你还有其他需要补充分享的吗？

刘颖：块存储在云里面是一个非常基础的服务，有了网络块存储以后，结合 SDN 的特性，虚拟机的自动迁移和快速的故障恢复才有可能性，会极大的提升云主机的可用性和弹性能力。另外块存储也是 I/O 里面最重要的一个环节，所以它的性能，我们也在一直不断的去提升，去满足各种场景下的性能。另外还有一些对延时非常敏感的一些类型，我们也在通过各种的方式，或者是不同的产品形态去满足用户的诉求，这是我们接下来需要持续努力的地方。

查看原文：[腾讯云刘颖：块存储深度剖析](#)

特别专栏 | Column

构建大型云计算平台分布式技术的实践

作者 章文嵩

本文基于章文嵩博士在 2014 年 7 月 18 日的全球架构师峰会 ArchSummit 上的主题演讲《构建大型云计算平台分布式技术的实践》整理而成。演讲 slides 可[从 ArchSummit 官网下载](#)。

演讲者简介

章文嵩博士是阿里集团的高级研究员与副总裁，主要负责基础核心软件研发和云计算产品研发、推进网络软硬件方面的性能优化、搭建下一代高可扩展低碳低成本电子商务基础设施。他也是开放源码及 Linux 内核的开发者，著名的 Linux 集群项目 LVS（Linux Virtual Server）的创始人和主要开发人员。LVS 集群代码已在 Linux 2.4 和 2.6 的官方内核中，保守估计全世界有几万套 LVS 集群系统在运行着，创造了近十亿美金的价值。加入阿里前，他是 TelTel 的首席科学家与联合创始人，曾为国防科技大学计算机学院副教授。他在设计和架构大型系统、Linux 操作系统、系统软件开发、系统安全和软件开发管理上有着丰富的经验。章文嵩博士在 2009 年加入阿里之后，先后负责淘宝的核心系统研发与阿里巴巴集团的基础研发，2013 年 10 月开始同时负责阿里云的系统研发与阿里巴巴集团的基础研发工作。

本演讲主要分为五个部分：

1. 云计算的挑战与需求
2. ECS 的分布式存储设计
3. SLB、RDS 与 OCS 的设计
4. 全链路监控与分析系统
5. 未来工作展望

云计算的挑战与需求

云计算跟淘宝在业务特点上有较大的不同，其中最大的不同就在于：淘宝、天猫是由四千多个小应用去支持的，都是分布式设计，很多情况下即使一两个应用宕机了，也不影响整体的服务，可以按部就班的修复。对于淘宝而言，只有交易量下降了 10% 以上的情况会算做是 P1 故障，开始计算全站不可用的时间。

而对于云计算的场景而言，一个云主机宕机了，对这个客户来说就是 100% 的不可用，而这可能是这个客户的全部“身家性命”。所以，云计算平台对可靠性、稳定性的需求

是非常高的。以前我们可能网络遇到问题，但是上层应用设计得好，就把这个问题隐蔽掉了；而对于云平台，要求是更高的可靠性，而且数据不能丢，系统稳定，性能还要好——目前尽量跟用户自己买物理机的性能差不多，另外要能够快速定位问题，最好在用户发现问题之前就先把问题解决了，让用户感知不到。还有就是成本要低，比用户自己买服务器便宜是底线。

ECS 的分布式存储设计

ECS 是阿里云的云服务器产品线，也是我们销量最大的产品。其背后是分布式文件存储，支持快照制作、快照回滚、自定义镜像、故障迁移、网络组隔离、防攻击、动态升级等功能。ECS 的管理基于一个庞大的控制系统，目前一个控制系统可以控制 3600 台物理机的规模，未来计划要做到 5000 台到两万台。

这其中，数据可靠性是极为关键的。阿里云以前的做法是数据写入的时候同步写三份到分布式存储上的 chunk server 上之后才算成功，这种实现的开销大，延时长，造成当时阿里云的用户抱怨性能不好。后来，我们做了 2-3 异步，即同步写 2 份确保成功，异步写第三份，IO 性能上得到一定的改善。我们现在对这个过程再做优化：读写性能优化的关键在于返回成功的时间，因为吞吐率是时间的倒数，延时缩短性能就会提升。缩短延时的思路之一就是将原本过长的路程截断以进行缩短，同时保证数据的可靠性。其具体思路为：

- SSD+SATA 的混合存储方案，在 chunk server 上做二级存储。这个方案目前在 vm 上做到的 randwrite-4K-128 可达 5500 IOPS 左右
- cache 机制
- 以多线程事件驱动架构重构 TDC 和 Chunk Server 的实现，做到一个 IO 请求在物理机上只用一个线程完成所有工作，避免锁和上下文切换

下面详细介绍一下这几个机制的设计。

IO 路径上的各层 cache 与写 IO 的几种模式探索

从应用发出请求到数据写入磁盘的路径上有三层 cache，依次是应用程序的 user cache（如 MySQL buffer pool）、操作系统的缓存（如 Linux page cache）、以及存储硬件的 cache（如磁盘的缓存）。

由此可以引申出如下几种写 IO 的模式：

- **buffer write**，写入目标是 guest OS 的 page cache，通过 writeback 刷到硬盘的缓存，然后再通过自动刷或者 sync 命令触发的方式刷到持久化存储介质上。这种写方案的速度很快，缺点是数据完整性无法得到严密保证（取决于回写的策略），而且回写有可能引起阻塞而影响服务质量

- direct write, 从应用直接写到硬件上的缓存, 绕过操作系统的 page cache。比如 MySQL 引擎自己有缓存机制, 就可以使用 direct write 写到硬盘缓存然后再通过 sync 命令刷到下面的存储介质。绕过 page cache 的好处是避开了回写的影响, 但数据仍然不是绝对可靠, sync 完毕之前数据仍然是不安全的
 - write+sync, 写入 page cache 的同时即调用 sync/fsync 直接写到存储介质, sync 返回算成功。此方式的好处是数据足够安全, 缺点是慢, 具体等待时间随着操作系统内存使用情况的不同而不同
 - O_SYNC, 加了此标签的写入操作会在数据写入硬盘缓存时同步刷到碟片上
- 以上就是系统提供的几种机制。以本地 SAS 盘作为参考, 在虚拟机中以 4k 的块大小做 dd 的写入速度, buffer write 平均在 212MB/s, direct write 平均在 68MB/s, 而 direct+sync 则平均在 257kB/s。实际应用中可以根据不同情况、不同应用选择不同的方式, 一般来说 buffer write 和 direct write 是主流, 两者加起来占据了 97% 的写操作。

云计算环境中的 IO

以上分析的是本地的情况, 写入的目标是本地的硬盘缓存与存储介质。那么在云计算环境中, 我们不仅可以选择本地, 还可以有分布式存储。分布式存储相当于本地的存储介质, 我们目前的思路是在其上加一层分布式缓存系统作为本地硬盘缓存的替代。相当于整个写 IO 路径在云计算环境中变成了:

VM SYNC->PV 前端 FLUSH->后端->host->cache 系统->分布式存储系统

为了确保数据完整性, 我们的语义全部符合 POSIX, 将语义由以上路径从 VM 透传 IO 全链路。

cache 系统的效果

我们用以下指令对 ECS 的写性能进行测试:

```
./fio -direct=1 -iodepth=1 -rw=randwrite -ioengine=libaio -bs=16k -numjobs=2 -runtime=30 -  
group_reporting -size=30G -name=/mnt/test30G
```

在 iodepth=1 的状态, 纯 SATA 分布式存储只有 200 左右的 iops, 平均延时在 8ms, 抖动幅度(标准方差)达到 7ms。

加入 SSD cache 系统之后, iops 提升到 600 左右, 平均延时降低到 3ms, 抖动幅度降低至 2ms 左右。

```
./fio -direct=1 -iodepth=8 -rw=randwrite -ioengine=libaio -bs=16k -numjobs=2 -runtime=30 -  
group_reporting -size=30G -name=/mnt/test30G
```

增加 iodepth 到 8 的状态，纯 SATA 分布式存储的 iops 提升至 2100 左右，平均延时在 7ms，抖动幅度依然是 7ms 左右。

加入 SSD cache 之后，iops 提升到 2900 左右，平均延时在 5ms 左右，抖动幅度约为 1ms。

以上是 cache 方案的两点好处：

1. 加速写请求。未来我们也会加入对读请求的加速
2. 降低分布式存储系统的抖动对上层应用的影响。这种抖动在高并发的情况下对延时的影响相当大，Google 的 Jeff Dean 于 2013 年 2 月发表于 CACM 上的 The Tail at Scale 一文详细描述了这个影响：“如果有 1% 的概率请求延迟超过 1S，并发 100 个请求，然后等待所有请求返回，延时超过 1S 的概率为 63%”

ECS 不同的存储选择

目前在 ECS 上可以有几种实例选择：背后是纯 SATA 存储集群的实例，适合大部分应用；对于 IO 性能要求更高的应用可以选择混合存储集群；我们未来还会推出性能更高的纯 SSD 集群，预计将在 11 月/12 月推出，目前的测试数据是物理机 chunk server 可以做到最高 18 万的 iops，虚机上可以把万兆跑满，iops 在 9 万左右，目前的问题就是跑满的状态需要消耗 6 颗 HT CPU，这一部分还有待优化。

另外，对于 Hadoop、HBase、MongoDB 这样本身已经考虑了 3 副本的系统，阿里云还提供了 SATA 本地磁盘和 SSD 本地磁盘的 ECS，减少不必要的冗余以降低成本。

以上就是我们对云服务器产品 ECS 的一些优化工作。云服务器理论上可以用来跑任何东西，但是通用的方案不适合做所有的事情。因此，阿里云同时提供了一些细分产品，在特定应用场景下将取舍做到极致——

SLB、RDS 与 OCS

SLB 是阿里云的负载均衡产品，提供了 4 层的（基于 LVS）和 7 层的（基于 Tengine），支持等价路由和 Anycast 跨机房容灾，同时具备防攻击的特性。一台 12 物理核机器的 SLB 的正常转发性能在 1200 万左右的 pps，心跳可以做几千台；而同等配置的 ECS（千兆网络）的转发性能只有 70 万左右的 pps，心跳也只能做两台。

RDS 是阿里云的数据库服务，跑在物理机上（而非虚拟机）。RDS 数据通道采用标准的三层架构，每层都做到机房和部件冗余，无状态设计；中间层提供了安全防护、流量调度和桥接的功能，管理通道以元数据库（MySQL）为中心，消息驱动，各组件异步通信，无状态支持热升级，一个控制系统下可以管理数万个 MySQL 实例。RDS 依赖于很多其他团队开发的组件，包括用 SLB 做负载均衡，接 ODPS 做过滤分析，SLS 做日志收集，OSS 做备份，OAS 做冷数据的备份，用精卫做分表，以及全链路的控制系统和组件监控。同等配置下，RDS 的 tps 要比 ECS 高两、三倍。

OCS 是阿里云的缓存服务，基于 Tair 搭建，前面的 Proxy 负责了安全访问控制、QoS、流控的工作。OCS 目前是一个集群都在一个机房，可随时扩容，对用户提供了全面的监控数据和图形展示。性能方面，OCS 上目前 99% 的请求都做到了 2ms 以内响应，去年双十一，整个 OCS 集群的能力做到了一秒内可处理一亿个请求。同等配置下，OCS 的成本要比 ECS 上自建 Memcached 便宜一半。

全链路监控与分析系统

监控分析系统目前在 RDS 上用的比较重。坦白讲去年 RDS 遇到很多问题，很大一部分问题就是闪断：背后的机器故障时，MySQL 实例会迁移，这时候如果客户端的应用做得好，应用会自动发起重连的请求，保持原先的连接，但很多应用做的时候并没有考虑这个问题。那时候很多游戏厂商遇到这个问题，让他们改程序也很困难，不可能一个一个帮助他们优化，所以就需要后端帮他们的实例做保持连接和重连的工作。

所以我们建立起全链路的监控，收集所有的 SQL 日志、网络行为和用户行为，注入到一个 Kafka 集群，然后用 JStorm 和 Spark 做实时分析，ODPS 做离线分析。目前每天的 SQL 日志语句的量级在几十个 T，可以在秒级发现问题，比如发现请求慢了，则会给用户提醒是否没有建索引，而网络异常、连接中断的情况则会及时报警。

目前这套系统先用在 RDS 上，未来各个云产品需要将自己的异常分析都抽象出来注入到这个系统当中，完成全产品线的全链路监控。

未来工作展望

首先，ECS 上全路径 IO 还需要持续优化，力求在全国、全球做到最好的性能。这涉及到 Cache 策略的优化，带 SSD 的读写缓存，存储与计算分离，万兆纯 SSD 集群，动态热点迁移技术，GPU 支持，LXC/cgroups 支持等。比如纯 SSD 的集群，iops 已经挖掘的很高的情况，如何降低 CPU 消耗？Cache 现在为了快速，往下刷的频率是比较高的，这方面的策略能否优化，做批量刷？以前部署的 SATA 集群，是否都加上 SSD 缓存？如果本地缓存的命中率在 90% 以上，是否可以做计算节点和存储节点分离，这样可以让计算和存储按自己的需求发展。未来实现动态的热点迁移，可以在云计算上要实现更高的超配，当一台物理机发生比较忙的情况下，系统能自动将一些实例迁移到比较闲的机器

上。目前淘宝的聚石塔、阿里小贷都已经在阿里云，未来会将淘宝无缝迁移到云平台上并降低成本，这些都是 ECS 上未来需要做的工作。

RDS 方面，目前支持 MySQL 和 SQL Server，计划加入 PostgreSQL 以方便 Oracle 用户往这边迁移。容灾方面，目前是双机房容灾，成本还比较高，是否可以通过非常高速的非易失性网络存储来存储 redo log，容量不需要大，数据存储在分布式文件系统，做一个低成本的 RDS 方案，只是用户需要容忍几十秒的 MySQL 实例宕机重启的时间？这需要架构师做取舍，看我们要放弃些什么以得到一些东西。

另外，全链路的监控与分析系统，我们还需要进一步应用到全线云产品之上。未来还会推出更多的云产品，包括无线网络加速、AliBench 服务质量监测（目前在内部使用）、OCR 识别服务、深度学习的 CNN/DNN 计算服务等。

感谢杨赛对本文的整理。

查看原文：[构建大型云计算平台分布式技术的实践](#)

相关内容

- [左耳朵耗子谈云计算：拼的就是运维](#)
- [阿里云计算资深总监唐洪谈飞天现状以及 5K 项目发展](#)
- [云计算与大片：基于阿里云的渲染农场](#)
- [云计算就是服务：阿里云总裁王坚博士访谈录](#)
- [阿里云开发者大会：企业如何看待云计算](#)

特别专栏 | Column

不得不知的 S3 基础知识

作者 包研

2006 年，亚马逊 AWS 推出了第一个对外的云服务 S3，一种面向互联网的存储，通过 API 就可以控制存储对象，相对于传统的磁盘和数据库，S3 使用更简便且无需维护。截至到 2013 年 Q2，S3 上存储了 2 万亿个数据对象。在 7 月 29 日进行的 InfoQ 在线课堂《您必须了解的 S3 基础知识》上，亚马逊 AWS 资深技术讲师张波、解决方案架构师张荣典回答了网友的提问。现将 Q/A 实录整理如下：

问：如果用 S3 Hosting forum 行吗？

答：S3 比较适合静态的数据，如果你的应用里面用到动态数据，比如说类似数据库这样的数据，推荐拿 EC2 配合 S3 来使用，来 Hosting 你的论坛的。

问：用 S3 作静态网站，还需要单独的 EC2 主机做 Web server 吗？

答：静态网站 S3 就可以服务。不需要 EC2 主机。

问：S3 是否支持对象分块查重更新功能？还是需要应用来实现这样的功能？

答：这个问题可能有两个方面了，第一个是否支持对象分块查看，其实 S3 里头有一个功能，叫 range based download，所以你在 http 请求的时候可以指定对该对象访问的 range。那我理解可能你主要是想问这个。

问：S3 会对内容扫描杀毒吗？

答：S3 是个简单的存储服务，它不会对用户的数据做任何扫描，或者读取操作。正好和大家分享一下 S3 的设计理念，S3 就是一个存储的平台，它做的是原子化的 API，做得非常的健壮，采用分布式因此非常稳定和可靠。像杀毒这些增值功能，我们会留给合作伙伴来支持，比如说今天听课的朋友想在 S3 上做一些附加的一些 feature，都可以去做，我们把这个广阔的空间就留给大家。

问：请问 S3 价格怎么计算？

答：S3 的价格是比较经济的，它收费主要分两部分，一部分是每月每 GB 的容量来计费。另外一个是按照 S3 上数据对象的下载的数据量来进行收费的，但把数据上传到 S3 上是完全免费的，我们有很多客户的他们的路联网应用充分利用了这个免费的功能，比如说 Dropbox、Dropcam 这类客户很好的享受了这种好处。

问：S3 在中国有数据中心吗？

答：大家注意到，我们在去年发布了一个中国的 region。目前 AWS 中国这个 region——bjs 是在一个有限公开预览的阶段。可以确定的告诉大家，S3 在中国是有的。

问：删除后马上读，会读到旧的数据。最终一致性多久（延迟）能保证？

答：读数据的时延，取决于当时的并发吞吐量和所读取的数据对象的大小。通常情况下，最终一致性的时延可能是在几百毫秒到几秒。

问：有关 AWS 的安全方面有讲座么？我觉得这反面应该讲讲。

答：安全确实是个很重要的话题，覆盖的范围比较广，比较适合在一个单独的话题里讨论。单就 S3 来说，客户可以通过服务器端加密的方式加强安全。同时，也可以在上传数据之前加密，数据传输过程中可以通过 SSL 的方式进行加密。

问：Region 和 Available Zone (AZ) 有啥区别？来自中国的请求，是如何被发送到新加坡 Region，而不是发送到美西 Region？

答：Region 是一个相对比较大的地理区域，是 AWS 提供服务的可用区（Available Zone）的集合。目前 AWS 有 10 个 Region，美国 4 个，亚洲有新加坡、悉尼、东京和北京。可用区是在一个 Region 里面，提供高可用的数据中心的集合。可用区之间会有足够的距离，来实现故障隔离。同时提供高速链路互联，数据可以同步的方式在可用区间复制。

问：关于 AWS 架构规划设计方面的问题，需要我们自己做吗？还是 AWS 根据公司的实际情况来设计？

答：客户可以联系 AWS 的销售和 SA（系统架构师）一起进行系统的规划设计。

问：在数据一致性方面，在更新操作的时候为什么没保证强一致性么？是基于什么方面的考虑？

答：分布式系统中都会遇到著名的 CAP 问题，C 是一致性，A 是服务的可用性，P 是分区容忍性。三者之中只能保证满足两个。最终一致性保证了分区容忍性和服务的可用性，比较适合对一致性要求没有那么严格的应用。很多互联网应用，尤其是在访问静态类型的数据的时候，这样的最终一致性足够满足应用的需要。

问：ELB 是在 http 请求时进行负载均衡。非 http 服务器，比如 RTMP 服务或者 Socket 服务器，前端如何进行负载均衡？

答：ELB 可以支持 4 层和 7 层。你的需求可以配置 4 层的 listener -tcp。另外，如果做流媒体，可以使用 S3 host 内容，配合 CloudFront 做分发。

问：如果跨 Region 访问 S3 效率如何，比如前端应用在 ap-southeast，S3 在 Us-West？

答：不推荐把 S3 和前端应用跨 Region 部署。S3 的存储桶可以通过 copy 或 sync 这两个命令在 Region 之间同步。

问：对象键名与分布式的性能是什么关系？键名随机性越大分布的区域越多吗？

答：S3 使用存储桶名和键名的前缀进行分区。在 aws.amazon.com/s3 的网站上，S3 的开发文档中有专门一章介绍如何通过键名的随机分布来提升性能。

问：针对目前数据泄露等信息安全方面的考虑，AWS 会有什么措施保证不重现“斯诺登”事件？

答：从 S3 来说，可以通过 client 端加密对象，或者 server 端加密对象。

问：对客户网络有什么样的要求？

答：主要看你自己应用的客户和客户端在那里，数据在 S3 的哪个 Region。

问：S3 有没有免费的试用空间？比如分配给开发者 4MB 左右的试用空间，我们可以自己动手进行测试一下？

答：注册 AWS Free Tier 可以享受 12 个月 5GB 的免费空间。

问：S3 是否有详细的访问日志和分析工具？

答：有 access log 功能，可以详细记录对 S3 存储桶的访问记录。

问：应用开发与 S3 接口示例在哪里下载？

答：<http://aws.amazon.com/cn/s3/developer-resources/>

查看原文：[InfoQ 在线问答：亚马逊 AWS S3 的热点问题](#)

AWS 解决方案&白皮书下载

白皮书推荐

《向AWS迁移或新建应用的最佳实践》，本文将重点强调创建新的云应用程序或将现有的应用程序移植到云端的概念、原则和最佳实践。您将了解云计算带来的一些业务与技术优势，以及目前已可利用的AWS服务。

《在AWS上保证应用安全的最佳实践》，在一个多租户环境中，云架构师常常就安全问题表示担忧。安全性问题应在云应用程序架构的每一层都能落实。物理安全问题通常是由您的服务提供商处理，这是使用云的一个额外优点。网络和应用程序层级的安全是您的责任，您应该执行最佳实践，以便使之尽可能地适用于您的业务。在本文中，您将了解一些特定的工具、功能和如何确保您的云计算应用程序在AWS环境中安全的指导方针。建议充分利用这些工具和功能，以便实现基本安全，然后再酌情使用标准的或合适的方法来执行附加的安全最佳实践。

更多白皮书

《我们如何应对风险？了解AWS安全概述》

《AWS与云计算：传统IT面临机遇和挑战》

《AWS的价格魅力：按需付费，无需签署长期合同》

《在AWS上保证应用安全的最佳实践》

《AWS云与自有IT基础设施的经济性比较》



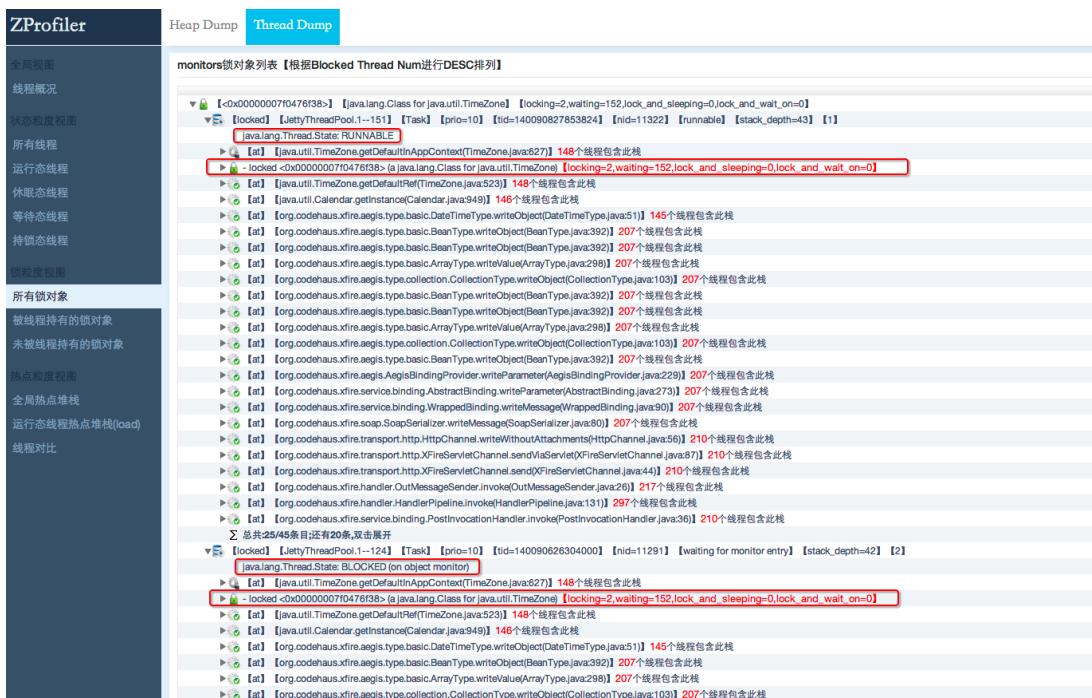
避开那些坑 | Void

JVM Bug：多个线程持有一把锁

作者 李嘉鹏

JVM 线程 dump Bug 描述

在 JAVA 语言中，当同步块（Synchronized）被多个线程并发访问时，JVM 中会采用基于互斥实现的重量级锁。JVM 最多只允许一个线程持有这把锁，如果其它线程想要获得这把锁就必须处于等待状态，也就是说在同步块被并发访问时，最多只会有一个处于 RUNNABLE 状态的线程持有某把锁，而另外的线程因为竞争不到这把锁而都处于 BLOCKED 状态。然而有些时候我们会发现处于 BLOCKED 状态的线程，它的最上面那一帧在打印其正在等待的锁对象时，居然也会出现-locked 的信息，这个信息和持有该锁的线程打印出来的结果是一样的(请看下图)，但是对比其他 BLOCKED 态的线程却并没有都出现这种情况。当我们再次 dump 线程时又可能出现不一样的结果。测试表明这可能是一个偶发的情况，本文就是针对这种情况对 JVM 内部的实现做了一个研究以寻找其根源。



jstack 命令的整个过程

上面提到了线程 dump，那么就不得不提执行线程 dump 的工具---jstack，这个工具是 Java 自带的工具，和 Java 处于同一个目录下，主要是用来 dump 线程的，或许大家也有

使用 kill -3 的命令来 dump 线程，但这两者最明显的一个区别是，前者的 dump 内容是由 jstack 这个进程来输出的，目标 JVM 进程将 dump 内容发给 jstack 进程(注意这是没有加-m 参数的场景，指定-m 参数就有点不一样了，它使用的是 serviceability agent 的 api 来实现的，底层通过 ptrace 的方式来获取目标进程的内容，执行过程可能会比正常模式更长点)，这意味着可以做文件重定向，将线程 dump 内容输出到指定文件里；而后者是由目标进程输出的，只会产生在目标进程的标准输出文件里，如果正巧标准输出里本身就有内容的话，看起来会比较乱，比如想通过一些分析工具去分析的话，要是该工具没有做过滤操作，很可能无法分析。因此一般情况我们尽量使用 jstack，另外 jstack 还有很多实用的参数，比如 jstack pid >thread_dump.log，该命令会将指定 pid 的进程的线程 dump 到当前目录的 thread_dump.log 文件里。

jstack 是使用 Java 实现的，它通过给目标 JVM 进程发送一个 threaddump 的命令，目标 JVM 的监听线程 (attachListener) 会实时监听传过来的命令(其实 attachListener 线程并不是一启动就创建的，它是 lazy 创建启动的)，当 attachListener 收到 threaddump 命令时会调用 thread_dump 的方法来处理 dump 操作(方法在 attachListener.cpp 里)。

```
static jint thread_dump(AttachOperation* op, OutputStream* out) {
    bool print_concurrent_locks = false;
    if (op->arg(0) != NULL && strcmp(op->arg(0), "-l") == 0) {
        print_concurrent_locks = true;
    }

    // thread stacks
    VM_PrintThreads op1(out, print_concurrent_locks);
    VMThread::execute(&op1);

    // JNI global handles
    VM_PrintJNI op2(out);
    VMThread::execute(&op2);

    // Deadlock detection
    VM_FindDeadlocks op3(out);
    VMThread::execute(&op3);

    return JNI_OK;
}
```

从上面的方法可以看到，jstack 命令执行了三个操作：

- VM_PrintThreads：打印线程栈
- VM_PrintJNI：打印 JNI

- **VM_FindDeadlocks:** 打印死锁

三个操作都是交给 VMThread 线程去执行的，VMThread 线程在整个 JAVA 进程有且只会有一个。可以想象一下 VMThread 线程的简单执行过程：不断地轮询某个任务列表并在有任务时依次执行任务。任务执行时，它会根据具体的任务决定是否会暂停整个应用，也就是 stop the world，这是不是让我们联想到了我们熟悉的 GC 过程？是的，我们的 ygc 以及 cmsgc 的两个暂停应用的阶段(init_mark 和 remark)都是由这个线程来执行的，并且都要求暂停整个应用。其实上面的三个操作都是要求暂停整个应用的，也就是说 jstack 触发的线程 dump 过程也是会暂停应用的，只是这个过程一般很快就结束，不会有明显的感觉。另外内存 dump 的 jmap 命令，也是会暂停整个应用的，如果使用了-F 的参数，其底层也是使用 serviceability agent 的 api 来 dump 的，但是 dump 内存的速度会明显慢很多。

VMThread 执行任务的过程

VMThread 执行的任务称为 vm_oeration，在 JVM 中存在两种 vm_oeration，一种是需要在安全点内执行的(所谓安全点，就是系统处于一个安全的状态，除了 VMThread 这个线程可以正常运行之外，其他的线程都必须暂停执行，在这种情况下就可以放心执行当前的一系列 vm_oeration 了)，另外一种是不需要在安全点内执行的。而这次我们讨论的线程 dump 是需要在安全点内执行的。

以下是 VMThread 轮询的逻辑：

```
void VMThread::loop() {
    assert(_cur_vm_operation == NULL, "no current one should be executing");

    while(true) {
        ...
        //已经获取了一个 vm_operation
        if (_cur_vm_operation->evaluate_at_safepoint()) {
            //如果该 vm_operation 需要在安全点内执行
            _vm_queue->set_drain_list(safepoint_ops);
            SafepointSynchronize::begin(); //进入安全点
            evaluate_operation(_cur_vm_operation);
            do {
                _cur_vm_operation = safepoint_ops;
                if (_cur_vm_operation != NULL) {
                    do {
                        VM_Operation* next = _cur_vm_operation->next();
                        _vm_queue->set_drain_list(next);
                        evaluate_operation(_cur_vm_operation);
                    }
                }
            }
        }
    }
}
```

```

        _cur_vm_operation = next;
        if (PrintSafepointStatistics) {
            SafepointSynchronize::inc_vmop_coalesced_count();
        }
    } while (_cur_vm_operation != NULL);
}

if (_vm_queue->peek_at_safepoint_priority()) {
    MutexLockerEx mu_queue(VMOperationQueue_lock,
                           Mutex::_no_safepoint_check_flag);
    safepoint_ops = _vm_queue->drain_at_safepoint_priority();
} else {
    safepoint_ops = NULL;
}
} while(safepoint_ops != NULL);
_vml_queue->set_drain_list(NULL);
SafepointSynchronize::end(); //退出安全点
} else { // not a safepoint operation
    if (TraceLongCompiles) {
        elapsedTimer t;
        t.start();
        evaluate_operation(_cur_vm_operation);
        t.stop();
        double secs = t.seconds();
        if (secs * 1e3 > LongCompileThreshold) {
            tty->print_cr("vm %s: %3.7f secs]", _cur_vm_operation->name(),
                           secs);
        }
    } else {
        evaluate_operation(_cur_vm_operation);
    }
    _cur_vm_operation = NULL;
}
}
...
}

```

在这里重点解释下在安全点内执行的 vm_opration 的过程， VMThread 通过不断循环从 _vm_queue 中获取一个或者几个需要在安全点内执行的 vm_opertion，然后在准备执行这些 vm_opration 之前先通过调用 SafepointSynchronize::begin()进入到安全点状态，在执行完这些 vm_opration 之后，调用 SafepointSynchronize::end()，退出安全点模式，恢复之前暂停的所有线程让他们继续运行。对于安全点这块的逻辑挺复杂的，仅仅需要记住在进入安全点模式的时候会持有 Threads_lock 这把线程互斥锁，对线程的操作都需要获

取到这把锁才能继续执行，并且还会设置安全点的状态，如果正在进入安全点过程中设置_state 为_synchronizing，当所有线程都完全进入了安全点之后设置_state 为_synchronized 状态，退出的时候设置为_not_synchronized 状态。

```
void SafepointSynchronize::begin() {
    ...
    Threads_lock->lock();
    ...
    _state = _synchronizing;
    ...
    _state = _synchronized;
    ...
}

void SafepointSynchronize::end() {
    assert(Threads_lock->owned_by_self(), "must hold Threads_lock");
    ...
    _state = _not_synchronized;
    ...
    Threads_lock->unlock();
}
```

线程 dump 中的 VM_PrintThreads 过程

回到开头提到的 JVM 线程 Dump 时的 Bug，从我们打印的结果来看也基本猜到了这个过程：遍历每个 Java 线程，然后再遍历每一帧，打印该帧的一些信息(包括类，方法名，行数等)，在打印完每一帧之后然后打印这帧已经关联了的锁信息，下面代码就是打印每个线程的过程：

```
void JavaThread::print_stack_on(outputStream* st) {
    if (!has_last_Java_frame()) return;
    ResourceMark rm;
    HandleMark hm;

    RegisterMap reg_map(this);
    vframe* start_vf = last_java_vframe(@_map);
    int count = 0;
    for (vframe* f = start_vf; f; f = f->sender() ) {
        if (f->is_java_frame()) {
```

```

javaVFrame* jvf = javaVFrame::cast(f);
java_lang_Throwable::print_stack_element(st, jvf->method(),
jvf->bci());
if (JavaMonitorsInStackTrace) {
    jvf->print_lock_info_on(st, count);
}
} else {
    // Ignore non-Java frames
}
count++;
if (MaxJavaStackTraceDepth == count) return;
}
}

```

和我们这次问题相关的逻辑，也就是打印"-locked"的信息是正好是在
jvf->print_lock_info_on(st, count)这行里面，请看具体实现：

```

void javaVFrame::print_lock_info_on(outputStream* st, int frame_count) {
    ResourceMark rm;
    if (frame_count == 0) {
        if (method()->name() == vmSymbols::wait_name() &&
            instanceKlass::cast(method()->method_holder())->name() ==
            vmSymbols::java_lang_Object()) {
            StackValueCollection* locs = locals();
            if (!locs->is_empty()) {
                StackValue* sv = locs->at(0);
                if (sv->type() == T_OBJECT) {
                    Handle o = locs->at(0)->get_obj();
                    print_locked_object_class_name(st, o, "waiting on");
                }
            }
        } else if (thread()->current_park_blocker() != NULL) {
            oop obj = thread()->current_park_blocker();
            Klass* k = Klass::cast(obj->klass());
            st->print_cr("\t- %s <" INTPTR_FORMAT "> (a %s)", "parking to wait for",
", (address)obj, k->external_name());
        }
    }
}

GrowableArray<MonitorInfo*>* mons = monitors();
if (!mons->is_empty()) {

```

```
bool found_first_monitor = false;
for (int index = (mons->length()-1); index >= 0; index--) {
    MonitorInfo* monitor = mons->at(index);
    if (monitor->eliminated() && is_compiled_frame()) {
        if (monitor->owner_is_scalar_replaced()) {
            Klass* k = Klass::cast(monitor->owner_klass());
            st->print("\t- eliminated <owner is scalar replaced> (a %s)",
k->external_name());
        } else {
            oop obj = monitor->owner();
            if (obj != NULL) {
                print_locked_object_class_name(st, obj, "eliminated");
            }
        }
        continue;
    }
    if (monitor->owner() != NULL) {
        const char *lock_state = "locked";
        if (!found_first_monitor && frame_count == 0) {
            markOop mark = monitor->owner()->mark();
            if (mark->has_monitor() &&
                mark->monitor() == thread()->current_pending_monitor()) {
                lock_state = "waiting to lock";
            }
        }
        found_first_monitor = true;
        print_locked_object_class_name(st, monitor->owner(), lock_state);
    }
}
}
```

看到上面的方法，再对比线程 dump 的结果，我们会发现很多熟悉的东西，比如 waiting on, parking to wait for, locked, waiting to lock，而且也清楚了它们分别是在什么情况下会打印的。

那为什么我们的例子中 BLOCKED 状态的线程本应该打印 waiting to lock,但是为什么却打印了 locked 呢，那说明 if (mark->has_monitor() && mark->monitor() == thread()->current_pending_monitor()) 这个条件肯定不成立，那这个在什么情况下不成立呢？在验证此问题前，有必要先了解下 markOop 是什么东西，它是用来干什么的？

markOop 是什么

markOop 描述了一个对象(也包括了 Class)的状态信息，Java 语法层面的每个对象或者 Class 在 JVM 的结构表示中都会包含一个 markOop 作为 Header，当然还有一些其他的 JVM 数据结构也用它做 Header。markOop 由 32 位或者 64 位构成，具体位数根据运行环境而定。

下面的结构图包含 markOop 每一位所代表的含义，markOop 的值根据所描述的对象的类型(比如是锁对象还是正常的对象)以及作用的不同而不同。就算在同一个对象里，它的值也是可能会不断变化的，比如锁对象，在一开始创建的时候其实并不知道是锁对象，会当成一个正常对象来创建(在对象的类型并没有设置偏向锁的情况下，其 markOop 值可能是 0x1)，但是随着我们执行到 synchronized 的代码逻辑时，就知道其实它是一个锁对象了，它的值就不再是 0x1 了，而是一个新的值，该值是对应栈帧结构里的监控对象列表里的某一个内存地址。

```

// 32 bits:
// -----
//      hash:25 ----->| age:4    biased_lock:1 lock:2 (normal
object)
//      JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased
object)
//      size:32 ----->| (CMS free
block)
//      PromotedObject*:29 ----->| promo_bits:3 ----->| (CMS
promoted object)
//
// 64 bits:
// -----
//  unused:25 hash:31 -->| unused:1    age:4    biased_lock:1 lock:2 (normal
object)
//  JavaThread*:54 epoch:2 unused:1    age:4    biased_lock:1 lock:2 (biased
object)
//  PromotedObject*:61 ----->| promo_bits:3 ----->| (CMS
promoted object)
//  size:64 ----->| (CMS
free block)
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs
&& normal object)
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs
&& biased object)
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs
&& CMS promoted object)

```

```
// unused:21 size:35 -->| cms_free:1 unused:7 ----->| (COOPs
&& CMS free block)
```

就最后的 3 位而言，其不同的值代表不同的含义：

```
enum { locked_value          = 0, //00
       unlocked_value        = 1, //01
       monitor_value         = 2, //10
       marked_value          = 3, //11
       biased_lock_pattern  = 5 //101
};
```

上面的判断条件“mark->has_monitor()”其实就是判断最后的 2 位是不是 10，如果是，则说明这个对象是一个监控对象，可以通过 mark->monitor()方法获取到对应的结构体：

```
bool has_monitor() const {
    return ((value() & monitor_value) != 0);
}

ObjectMonitor* monitor() const {
    assert(has_monitor(), "check");
    // Use xor instead of &~ to provide one extra tag-bit check.
    return (ObjectMonitor*) (value() ^ monitor_value);
}
```

将一个普通对象转换为一个 monitor 对象的过程(就是替换 markOop 的值)请参考为 ObjectSynchronizer::inflate 方法，能进入到该方法说明该锁为重量级锁，也就是说这把锁其实是被多个线程竞争的。

了解了 markOop 之后，还要了解下上面那个条件里的 thread()->current_pending_monitor()，也就是这个值是什么时候设置进去的呢？

线程设置等待的监控对象的时机

设置的逻辑在 ObjectMonitor::enter 里，关键代码如下：

```
...
{
    JavaThreadBlockedOnMonitorEnterState jtbes(jt, this);
```

```

DTRACE_MONITOR_PROBE(contended_enter, this, object(), jt);
if (JvmtiExport::should_post_monitor_contended_enter()) {
    JvmtiExport::post_monitor_contended_enter(jt, this);
}
OSThreadContendState osts(Self->osthread());
ThreadBlockInVM tbivm(jt);
Self->set_current_pending_monitor(this); //设置当前 monitor 对象为当前线程等待的 monitor 对象
for (;;) {
    jt->set_suspend_equivalent();
    EnterI (THREAD) ;
    if (!ExitSuspendEquivalent(jt)) break ;
    _recursions = 0 ;
    _succ = NULL ;
    exit (false, Self) ;

    jt->java_suspend_self();
}
Self->set_current_pending_monitor(NULL);
}
...

```

设置当前线程等待的 monitorObject 是在有中文注释的那一行设置的，那么出现 Bug 的原因是不是正好在设置之前进行了线程 dump 呢？

水落石出

在 JVM 中只会有一个处于 RUNNABLE 状态的线程，也就是说另外一个打印"-locked"信息的线程是处于 BLOCKED 状态的。上面的第一行代码：

```
JavaThreadBlockedOnMonitorEnterState jtbmes(jt, this);
```

找到其实现位置：

```

JavaThreadBlockedOnMonitorEnterState(JavaThread *java_thread,
ObjectMonitor *obj_m) :
JavaThreadStatusChanger(java_thread) {
assert((java_thread != NULL), "Java thread should not be null here");
_active = false;
}

```

```

    if (is_alive() && ServiceUtil::visible_oop((oop) obj_m->object()) &&
obj_m->contentions() > 0) {
        _stat = java_thread->get_thread_stat();
        _active = contended_enter_begin(java_thread); //关键处
    }
}

static bool contended_enter_begin(JavaThread *java_thread) {
    set_thread_status(java_thread,
java_lang_Thread::BLOCKED_ON_MONITOR_ENTER); //关键处
    ThreadStatistics* stat = java_thread->get_thread_stat();
    stat->contended_enter();
    bool active = ThreadService::is_thread_monitoring_contention();
    if (active) {
        stat->contended_enter_begin();
    }
    return active;
}

```

上面的 contended_enter_begin 方法会设置 java 线程的状态为
`java_lang_Thread::BLOCKED_ON_MONITOR_ENTER`, 而线程 dump 时根据这个状态打
印的结果如下:

```

const char* java_lang_Thread::thread_status_name(oop java_thread) {
    assert(JDK_Version::is_gte_jdk15x_version() && _thread_status_offset != 0, "Must have thread status");
    ThreadStatus status =
(java_lang_Thread::ThreadStatus)java_thread->int_field(_thread_status_offset);
    switch (status) {
        case NEW : return "NEW";
        case RUNNABLE : return "RUNNABLE";
        case SLEEPING : return "TIMED_WAITING (sleeping)";
        case IN_OBJECT_WAIT : return "WAITING (on object monitor)";
        case IN_OBJECT_WAIT_TIMED : return "TIMED_WAITING (on object
monitor)";
        case PARKED : return "WAITING (parking)";
        case PARKED_TIMED : return "TIMED_WAITING (parking)";
        case BLOCKED_ON_MONITOR_ENTER : return "BLOCKED (on object monitor)";
        case TERMINATED : return "TERMINATED";
        default : return "UNKNOWN";
    }
}

```

```
};  
}
```

正好对应我们 dump 日志中的信息"BLOCKED (on object monitor)" 也就是说这行代码被正常执行了，那问题就可能出在 JavaThreadBlockedOnMonitorEnterState jtbmes(jt, this) 和 Self->set_current_pending_monitor(this) 这两行代码之间的逻辑里了：

```
JavaThreadBlockedOnMonitorEnterState jtbmes(jt, this);  
DTRACE_MONITOR_PROBE(contended_enter, this, object(), jt);  
if (JvmtiExport::should_post_monitor_contended_enter()) {  
    JvmtiExport::post_monitor_contended_enter(jt, this);  
}  
OSThreadContendState osts(Self->osthread());  
ThreadBlockInVM tbivm(jt);  
Self->set_current_pending_monitor(this); // 设置当前 monitor 对象为当前线程等待的 monitor 对象
```

于是检查每一行的实现，前面几行都基本可以排除了，因为它们都是很简单的操作，下面来分析下 ThreadBlockInVM tbivm(jt) 这一行的实现：

```
ThreadBlockInVM(JavaThread *thread)  
: ThreadStateTransition(thread) {  
    thread->frame_anchor()->make_walkable(thread);  
    trans_and_fence(_thread_in_vm, _thread_blocked);  
}  
  
void trans_and_fence(JavaThreadState from, JavaThreadState to) {  
    transition_and_fence(_thread, from, to);  
}  
  
static inline void transition_and_fence(JavaThread *thread,  
JavaThreadState from, JavaThreadState to) {  
    assert(thread->thread_state() == from, "coming from wrong thread  
state");  
    assert((from & 1) == 0 && (to & 1) == 0, "odd numbers are transitions  
states");  
    thread->set_thread_state((JavaThreadState)(from + 1));  
    if (os::is_MP()) {  
        if (UseMembar) {  
            OrderAccess::fence();  
        }  
    }  
}
```

```

    } else {
        InterfaceSupport::serialize_memory(thread);
    }
}

if (SafePointSynchronize::do_call_back()) {
    SafePointSynchronize::block(thread);
}
thread->set_thread_state(to);
CHECK_UNHANDLED_OOPS_ONLY(thread->clearUnhandledOops());
}
...
}

```

也许我们看到可能造成问题的代码了：

```

if (SafePointSynchronize::do_call_back()) {
    SafePointSynchronize::block(thread);
}

```

想象一下，当这个线程正好执行到这个条件判断，然后进去了，从方法名上来说是不是意味着这个线程会 block 住，并且不往后走了呢？这样一来设置当前线程的 pending_monitor 对象的操作就不会被执行了，从而在打印这个线程栈的时候就会打印"-locked"信息了，那么纠结是否正如我们想的那样呢？

首先来看条件 SafePointSynchronize::do_call_back() 是否一定会成立：

```

inline static bool do_call_back() {
    return (_state != _not_synchronized);
}

```

上面的 VMThread 执行任务的过程中说到了这个状态，当 vmThread 执行完了 SafePointSynchronize::begin() 之后，这个状态是设置为 _synchronized 的。如果正在执行，那么状态是 _synchronizing，因此，当我们触发了 jvm 的线程 dump 之后，VMThread 执行该操作，而且还在执行线程 dump 过程前，但是还只是 _synchronizing 的状态，那么 do_call_back() 将会返回 true，那么将执行接下来的 SafePointSynchronize::block(thread) 方法：

```

void SafePointSynchronize::block(JavaThread *thread) {

```

```

assert(thread != NULL, "thread must be set");
assert(thread->is_Java_thread(), "not a Java thread");

ttyLocker::break_tty_lock_for_safepoint(os::current_thread_id());

if (thread->is_terminated()) {
    thread->block_if_vm_exited();
    return;
}

JavaThreadState state = thread->thread_state();
thread->frame_anchor()->make_walkable(thread);

switch(state) {
    case _thread_in_vm_trans:
    case _thread_in_Java:           // From compiled code
        thread->set_thread_state(_thread_in_vm);

        if (is_synchronizing()) {
            Atomic::inc (&TryingToBlock) ;
        }

        Safepoint_lock->lock_without_safepoint_check();
        if (is_synchronizing()) {
            assert(_waiting_to_block > 0, "sanity check");
            _waiting_to_block--;
            thread->safepoint_state()->set_has_called_back(true);

            DEBUG_ONLY(thread->set_visited_for_critical_count(true));
            if (thread->in_critical()) {
                increment_jni_active_count();
            }

            if (_waiting_to_block == 0) {
                Safepoint_lock->notify_all();
            }
        }

        thread->set_thread_state(_thread_blocked);
        Safepoint_lock->unlock();
        Threads_lock->lock_without_safepoint_check(); //关键代码
        thread->set_thread_state(state);
        Threads_lock->unlock();
        break;
    ...
}

```

```
if (state != _thread_blocked_trans &&
    state != _thread_in_vm_trans &&
    thread->has_special_runtime_exit_condition()) {
    thread->handle_special_runtime_exit_condition(
        !thread->is_at_poll_safepoint() && (state !=
_thread_in_native_trans));
}

void Monitor::lock_without_safepoint_check (Thread * Self) {
    assert (_owner != Self, "invariant") ;
    ILock (Self) ;
    assert (_owner == NULL, "invariant");
    set_owner (Self);
}

void Monitor::lock_without_safepoint_check () {
    lock_without_safepoint_check (Thread::current());
}
```

看到上面的实现可以确定，Java 线程执行时会调用 `Threads_lock->lock_without_safepoint_check()`，而 `Threads_lock` 因为被 `VMThread` 持有，将一直卡死在 `ILock (Self)` 这个逻辑里，从而没有设置 `current_monitor` 属性，由此验证了我们的想法。

Bug 修复

在了解了原因之后，我们可以简单的修复这个 Bug。将下面两行代码调换下位置即可：

```
ThreadBlockInVM tbivm(jt);
Self->set_current_pending_monitor(this); // 设置当前 monitor 对象为当前线程等待的
monitor 对象
```

该 Bug 不会对生产环境产生影响，本文主要是和大家分享分析问题的过程，希望大家碰到疑惑都能有一查到底的劲儿，带着问题，不断提出自己的猜想，然后不断验证自己的猜想，最终解决问题。

感谢[郭董](#)对本文的审校和策划。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至 editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)）或者腾讯微博（[@InfoQ](#)）关注我们，并与我们的编辑和其他读者朋友交流。

查看原文：[JVM Bug:多个线程持有一把锁](#)

为什么 CDN 对移动客户端加速“没有”效果

作者 刘宇

Google web 性能优化工程师和开发大使、《High-Performance Browser Networking》作者 Ilya Grigorik 近日发布了一篇名为《为什么 CDN 对移动客户端加速“没有”效果》的博客，描述了移动（无线）网络的特殊性，以及如何建设一个适用于移动 CDN 的构想。

Ilya 首先吐槽了目前的 CDN 在移动客户端加速方面的不给力。从他们的移动客户端性能监控数据来看，传统 CDN 的优化效果非常不明显，所以他希望有一个对移动网络支持更好的、特殊的移动 CDN 网络。

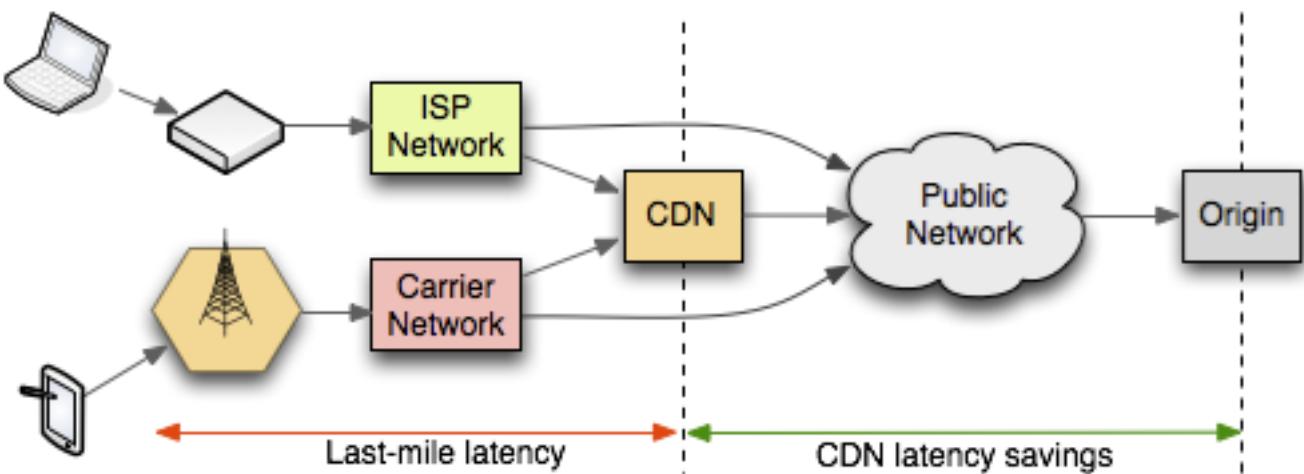
对于传统 CDN 在无线网络上的效果，Ilya 认为人们普遍有两种误解：1、传统 CDN 对移动客户端和对宽带网络的绝对优化效果差不多；2、这不是要不要“无线 CDN”的问题，而是运营商网络的问题。

Ilya 首先提供了一个参考数据，用于分析无线网络延迟的主要组成部分：

- 客户端位于西海岸；服务端位于东海岸。
- 美国东西海岸之间的网络延迟是 50ms。
- 服务端的响应延迟是 50ms。
- 共享客户端 Last-mile 延迟为：光纤约 18ms，电缆约 26ms，DSL 约 44ms。
- 无线客户端 Last-mile 延迟为：4G 约 50ms，3G 约 200ms。

注：Last-mile 最后一公里，通信行业经常使用“最后一公里”来指代从通信服务提供商的机房交换机到用户计算机等终端设备之间的连接。

下图显示使用 CDN 时用户访问流程和延迟信息



使用一个 CDN 做内容分发加速

CDN 加速需要在世界各地[对等点](#)的各种数据中心部署 CDN 高速缓存服务器，并尽可能的将数据部署在离用户最近的地方。换句话说，在最理想的情况下，CDN 服务器会立即定位客户端所在的 ISP/运营商网络，客户端发起请求，所引发的 last-mile 延迟时长为：客户端断开 ISP/运营商网络和命中时 CDN 服务器立即返回的响应时间。因此：

1. CDN 减少了 propagation latency；
2. 在缓存了静态资源的情况下，CDN 还减少了 server response time；

继续前面的例子，假设 CDN 服务器进行了网络优化配置（东海岸到西海岸的延迟时间不是 50ms 而是 5ms）和我们请求 CDN 未命中源站的情况下客户端到 CDN 节点的延迟是 5ms。对于采用光纤的客户端，新的总时间为 last-mile 往返加 CDN 响应时间的总和：18+5+5+5+18，即 51ms。因此，增加 CDN 的好处就是将我们总的请求时间由 186ms 降低到了 51ms：在总延迟上有 365% 的改善。

我们可以来看下不使用 CDN 和使用 CDN 加速时相关的性能数据，如下表所示：

	Last-mile	Coast-to-Coast (low)	Server Response	Total (ms)	Improvement
Fiber	18	50	50	186	
Cable	26	50	50	202	
DSL	44	50	50	238	
4G	50	50	50	250	
3G	200	50	50	550	
CDN + Fiber	18	5	5	51	-135 ms (365%)

CDN + Cable	26	5	5	67	-135 ms (301%)
CDN + DSL	44	5	5	103	-135 ms (231%)
CDN + 4G	50	5	5	115	-135 ms (217%)
CDN + 3G	200	5	5	415	-135 ms (133%)

采用同样的方法重复计算每个连接的基本信息，就可以得到一个不幸的趋势：

1. last-mile 的延迟最高，CDN 的相对有效性越差
2. 考虑到 CDN 服务器一般都放置在 ISP 网络之外，这就意味着节点的选择非常有意义
3. CDN 对于改善 last-mile 的延迟还是有一定效果的

CDN 帮助减少数据传播和服务端响应延迟时间。如果你衡量优化前后的对比，就会发现 CDN 几乎没有做移动客户端的优化：例如，3G 用户普遍获得 33% 的优化效果。

在边缘节点上的运营和业务维护成本

一个很明显的策略是：移动缓存服务器到更靠近客户的位置以提高终端到终端的延迟，而不是将节点部署在运营商网络之外。那么，我们是否可以将节点部署在运营商内部？原则上是可以的，现在许多运营商已经部署了自己的缓存服务器。然而在现实中，存在如下问题：

1. 对等点的数量相对比较少，CDN 只能部署在世界各地众所周知的几十个位置。然而，移动服务器到运营商网络内部需要与每个运营商单独结算，所以，通常情况下，服务器部署在共享数据中心（对等点）。
2. 我们假设 CDN 已经和某个 ISP 达成某种协议，理想情况下尽可能将服务器部署靠近他们的客户（在无线电天线塔和其它信号聚合点）的位置。这样做将需要大量硬件设备，这将导致维护和升级成为运维的恶梦，并打开了一个安全问题。例如，你将要部署一个第三方的 TLS 终端节点来操作网络，解决你不能直接访问网络的问题。总之，这是一个成本、安全和物流的恶梦。
3. 许多互联网运营商长期以来一直在尝试提升“档次”并提供 CDN 服务。然而，运营商也存在不同的问题：很难签订客户，因为大多数网站对于和每个运营商单独签署协议丝毫不感兴趣。

最近的新闻报道说 [Verizon 收购了 EdgeCast](#)，如果能将其应用于生产环境，这将有利于 Verizon 的客户解决这个问题。

除了业务和运营成本之外，CDN 在移动客户端上没有任何特殊的优化。问题的根源在于：移动运营商的 last-mile 延迟是很糟糕的。这才是我们需要解决的问题，而不是推动将缓存服务器部署在靠近用户的边缘。我们需要公开地进行网络的优化，我们需要更多的运营商参与竞争，从根本上解决 last-mile 性能问题。

在国内，运营商环境更为复杂，大大小小运营商有很多家，其中以北方网通、南方电信、移动为主。但伴随着互联网的发展，小型运营商通过控制入口并以 2、3 级城市为主逐渐扩大了规模，例如：电信通、华数科技、长城宽带等。还包括一些城域网，这些我们通常统称为小运营商。

由于各运营商之间存在着网间费用结算，因此运营商会想尽一切办法将内容存在自己的网内，这就造就了现在市场上比较混乱的“劫持”，而劫持技术也是越发越“高科技”。

国内的 CDN 环境竞争也日益加剧，几大 CDN 厂家如网宿科技、蓝讯、快网、帝联也纷纷与运营商进行合作。例如：蓝讯与中国电信宣布共建 CDN 网络，而网宿科技则是发布 MAA 移动应用加速解决方案，正式宣布进军移动互联网市场。再加上大公司自建 CDN 的加入，并有公司将 CDN 与云服务进行整合加入竞争，使得市场愈发激烈。

环境的复杂，导致用户访问的问题更加难以解决。有些观点表示，只有等到互联网关于运营商的改革，这些局面才会得以改善，但我认为只要各大运营商与公司紧密合作，合作更加深入，用户的访问质量肯定会节节攀升。

延伸阅读：[一秒钟法则：来自腾讯无线研发的经验分享](#)

感谢[丁雪丰](#)对本文的审校。

查看原文：[为什么 CDN 对移动客户端加速“没有”效果](#)

相关内容

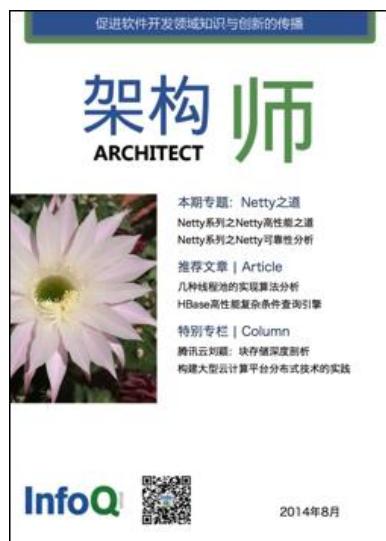
- [新浪 CDN 自动化运维](#)
- [自建 CDN 防御 DDoS（2）：架构设计、成本与部署细节](#)
- [自建 CDN 防御 DDoS（1）：知己知彼，建设持久防线](#)
- [又拍云存储外围 CDN 剖析](#)
- [Apache Traffic Server 与 CDN 实践](#)

封面植物



短毛球（学名 *Echinopsis tubiflora*），俗称草球，又名长盛球，是仙人掌科仙人球属最常见的一种。原产于南美洲，一般生长在高热、干燥、少雨的沙漠地带。球体绿色，圆筒形，棱排列整齐，短刺灰褐色。夏季是盛花期，小小的花蕾逐日变大抽长，直到最后犹如一支特大号的毛笔。开花时间不长，从晚上 6, 7 点开放，到第二天中午就谢。花形较大，洁白素雅，并散发很好闻的幽香，这点在其它仙人球中并不多见。傲然挺立的身姿，有种鹤立鸡群的味道，这些就是它在欧美地区有“夜皇后”之称的来由吧。

短毛球生命力很强，只要控制浇水，冬季放于室内，一般不会死亡。球体很会长仔球，随便掰下一个置于土上就能成活。



架构师 2014 年 8 月刊
每月 8 号出版

本期主编：郭蕾
策划编辑：杨赛
发行人：霍泰稳
读者反馈/投稿：editors@cn.infoq.com
InfoQ 中文站新浪微博：<http://weibo.com/infoqchina>
商务合作：sales@cn.infoq.com 15810407783



本期主编：郭蕾，InfoQ 技术编辑，文艺范儿程序员，并发编程网站长。在 CRM 行业厮混 3 年多，喜欢技术写作和社区运营，信奉见城彻先生的那句话：偏执、冒险、狂妄的人终是英雄。我不是英雄，但我会努力成为英雄。