Java NIO通信框架在电信领域的实践

2015-05-26 李林峰 InfoQ

> 华为电信软件技术架构演进

电信软件

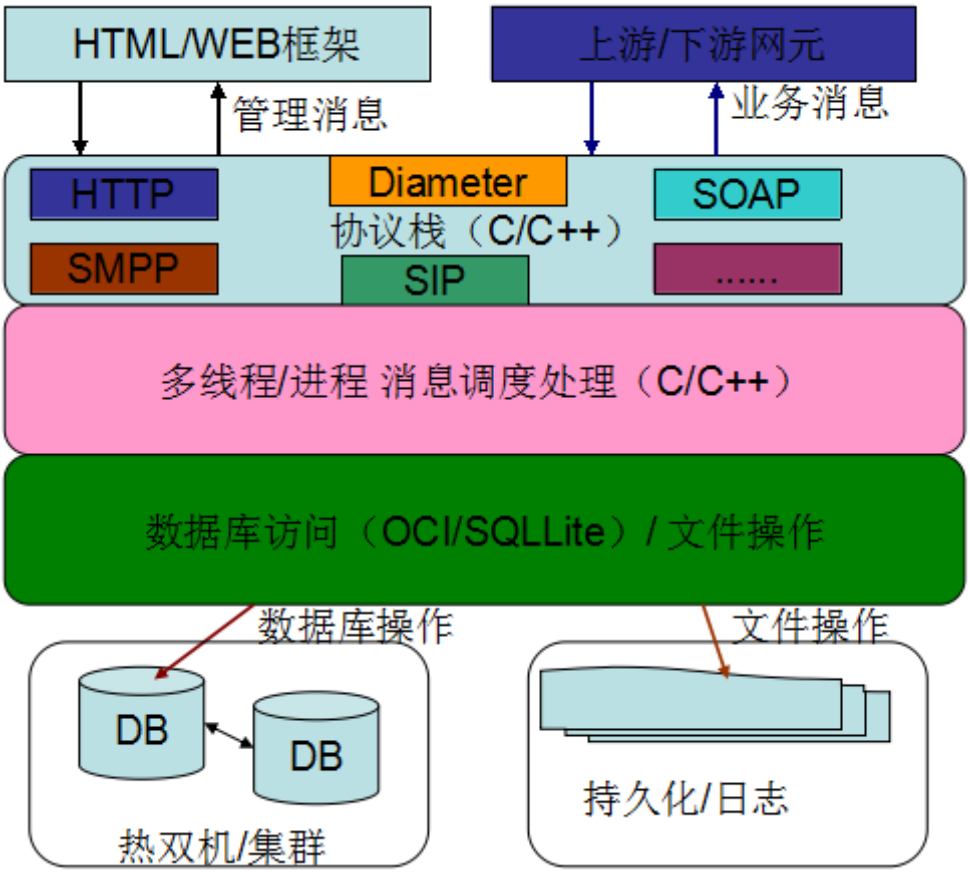
> 从广义上看电信软件的范围非常广，细分实际可以分为两大类：系统软件和业务应用软件。
>
> 系统软件包括路由器底层的信令机软件、手机操作系统等，业务应用软件主要包括客户关系管理CRM、网上营业厅、融合计费OCS和各类消息网关，例如短信网关、彩信网关等。
>
> 本文重点介绍电信业务应用软件的技术变迁历史，以及华为电信软件架构演进和Java NIO框架在技术变迁中起到的关键作用。

华为电信软件的技术演进史

C和C++主导的第一代架构

在2005年之前，华为软件公司的核心系统主要以C和C++进行开发，由于C和C++开源框架非常少，加之那个时代开源社区并不成熟，大部分的系统都采用自研开发，包括协议栈、系统调度、数据访问层和日志。

大多数的软件都运行在服务端，对外提供高性能、低时延和高并发的系统调用，协议栈大多数都采用电信私有协议栈，对于部分有前台管理Portal的系统，往往基于原生的HTML或者Struts等WEB框架开发，通过HTTP协议与后端进行交互，它的逻辑架构图如下：



华为电信软件**V1**版逻辑架构图

在那个时代，电信软件绝大多数都部署在高性能的小机中，处理各种信令、电信私有协议的接入和解析、复杂业务逻辑处理，系统对处理性能、时延、多核处理的要求非常高。当时Java主流版本还是JDK 1.4.2（1.4.X），它在传统的Web应用、电子商务网站和政企系统中得到了比较广泛的应用，但是在电信领域并没有大的应用，主要原因如下：

1) 在JDK1.5之前的早期版本中，Java在多线程编程、并行处理等方面能力很差，无法在电信软件服务器端使用；

2) JDK 1.4.X对非阻塞I/O的支持并不好，相关NIO编程的可参考资料和开源框架很少，传统的阻塞I/O模型在电信高性能、高可靠场景中力不从心；

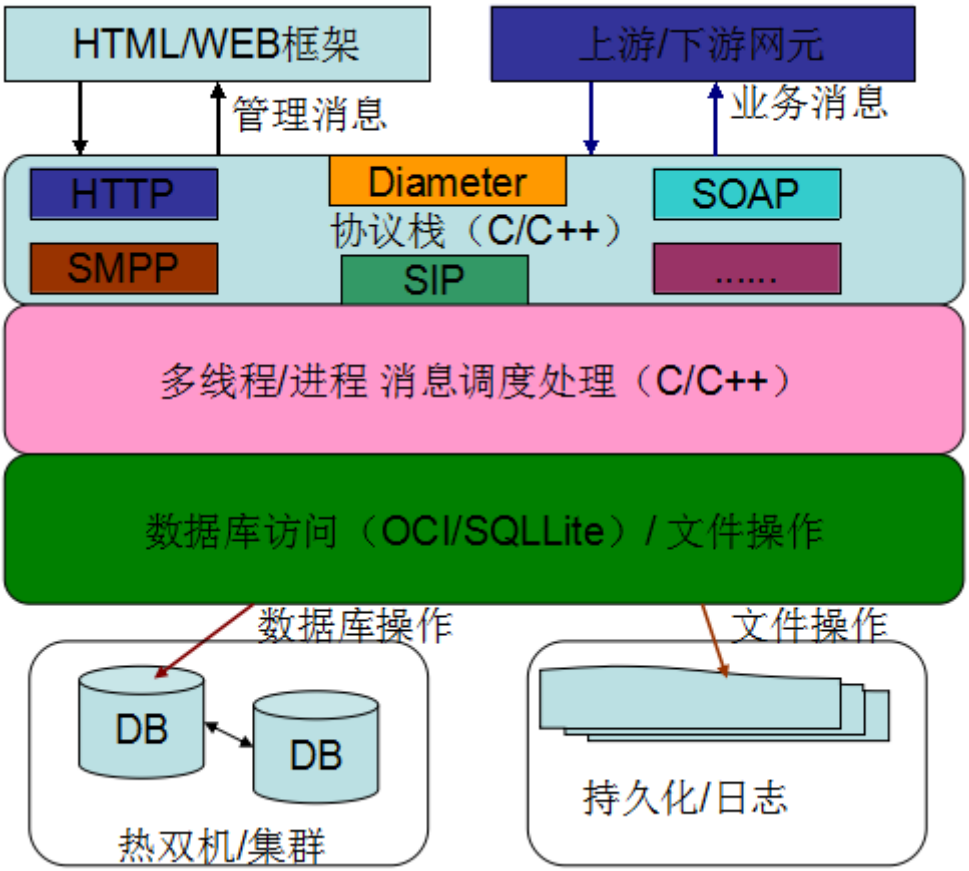
3) 业界很少有Java高性能服务端处理成功的案例，大家普遍对Java 支持电信级应用场景持怀疑态度；

4) 那个时代电信领域的开发者都是C/C++出身，大家对新技术和语言有种天生的排斥。

2005年之后，随着Java在各领域的快速普及和应用，以及基于Java的各种开源框架井喷式增长，华为越来越多的产品开始尝试切换到Java进行开发，主流架构随即演进到了以Java为主的V2版本。

Spring + Struts + Tomcat 的第二代架构

2005年-2008年间，华为电信软件大多数产品线都切换到Java语言进行新产品的设计和开发，当时随着Struts的MVC模式以及Spring对J2EE复杂企业应用对象生命周期的配置式管理的流行，华为电信软件绝大多数产品采用基于Spring + Struts + Tomcat模式进行开发，数据访问中间件主要采用iBatis和Hibernate,它的逻辑架构如下所示：



华为电信软件V2 MVC版逻辑架构图

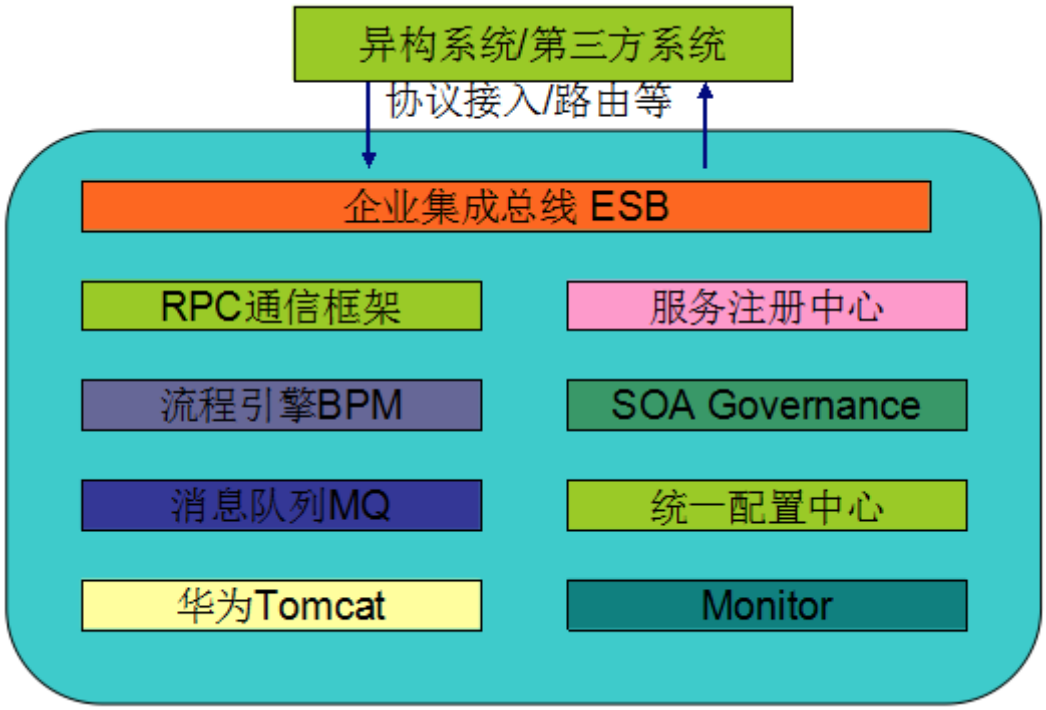
切换到以Spring + J2EE容器为基础技术框架之后，应用开发的难度迅速降低，开发效率获得了极大提升。短短1-2年时间，公司大多数以C/C++的项目切换到了Java语言和V2 架构上。

以SOA为中心的第三代架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。

随着电信业务的快速发展，电信原有系统和新建设系统之间存在语言、协议、运行环境等诸多差异。如何整合异构系统，实现高效企业集成，也是一个巨大的挑战，此时，企业服务总线（ESB）是个不错的选择。

为了满足电信业务的需求，华为软件研发了SOA中间件，它的逻辑架构图如下：



以SOA服务化为核心的V3架构

SOA是一种粗粒度、松耦合的以服务为中心的架构，接口之间通过定义明确的协议和接口进行通信。SOA帮助工程师们站在一个新的高度理解企业级架构中各种组件的开发和部署形式，它可以帮助企业系统架构师以更迅速、可靠和可重用的形式规划整个业务系统。相比于传统的垂直架构，SOA能够更加从容的应对复杂企业系统集成和需求的快速变化。

以分布式、云化为核心的第四代架构

随着业务的不断发展，硬件成本的下降，基于X86架构的廉价硬件 + 分布式软件的模式在互联网行业得到了大规模应用，分布式架构日趋成熟。

从运营商业务看，尽管高性能的小机仍然是标配，但是运营商业务向数字化转型和云化降成本逐渐成为一种趋势。

传统SOA架构中的一些缺陷逐步暴露，例如企业集成总线ESB是实体总线，性能线性扩展能力有限；硬件负载均衡器的压力越来越大，不断扩容导致硬件成本增加；随着业务规模的不断增长，传统的数据库、配置中心等逐渐成为单点瓶颈等。

我们需要通过新的分布式架构来解决电信软件面临的成本高、性能无法线性增长等问题，以分布式技术为核心构建的华为分布式中间件应用而生，它主要包括如下组件：

1) 高性能、低时延的分布式服务框架；

2) 分布式消息队列MQ；

3) 分布式缓存；

4) 分布式数据库访问中间件，支持跨库操作，支持异构数据库；

5) 软负载SLB;

6) 分布式日志采集和检索（Flume + ELK）；

7) 分布式实时流式计算框架；

8) 分布式消息跟踪系统；

9) 其它.....

自从亚马逊的云计算服务面世以来，云计算技术作为应对笨重的传统IT架构的战略，已经成为越来越多的政府和企业的选择， “云”已经成为ICT技术和服务领域的“常态”。

运营商基础设施云化的主要原因如下：

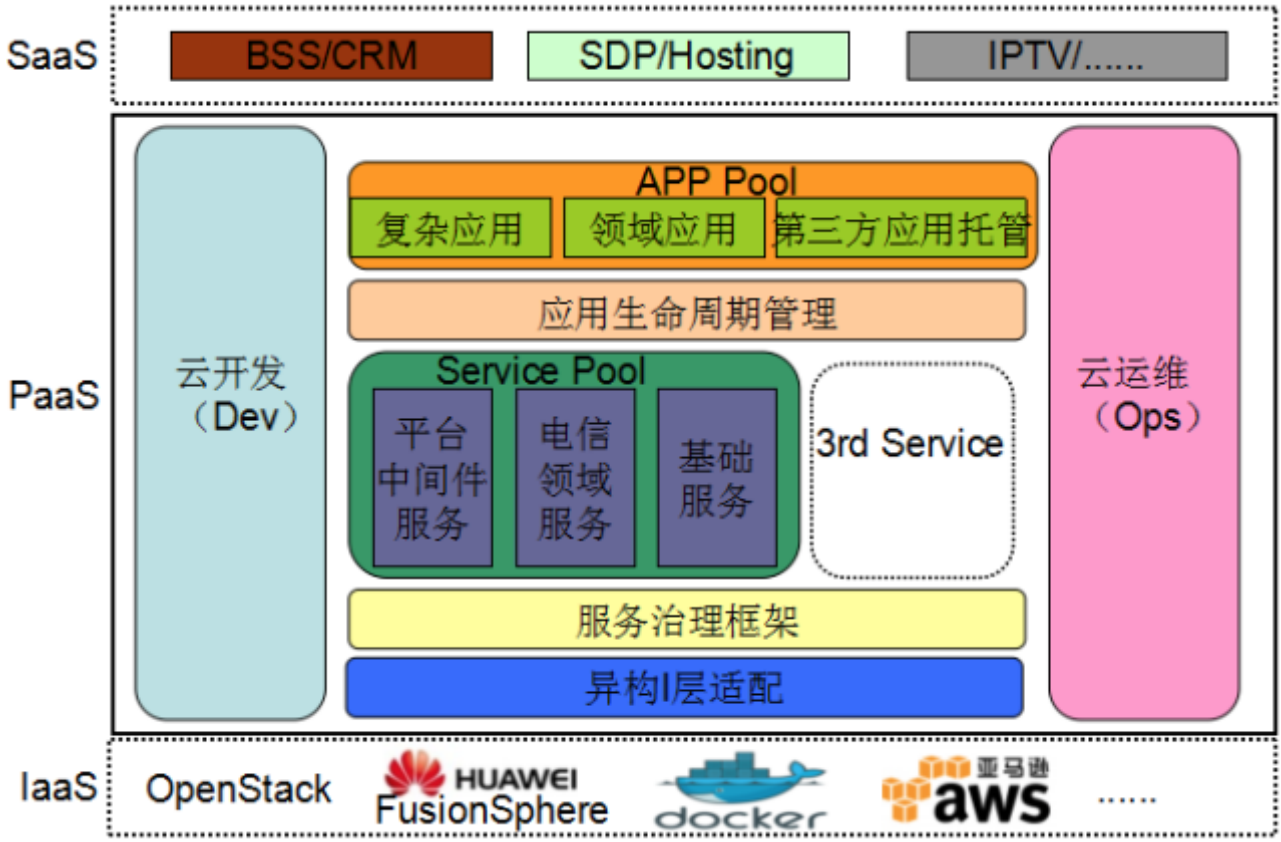
1) IT资源规模比较大，如何高效的使用这些设备，提升效率，虚拟化是个不错的选择；

2) 资源的孤岛现象是比较严重，大部分IT系统，依然采用传统的竖井式的建设模式，IT系统的资源无法在跨系统间进行共享，同时因为各系统建各系统的特点，使得资源的利用率非常低，各个业务有峰值的时候，虽然业务之间峰值不在一块，但是依然起不到消峰的作用，占用的资源比较大；

3) 系统的压力也是不均衡，资源由于没法共享，只能采用被动的采购，使得更大容量的设备采购，来应付电信业务增长所需要的扩容；

4) 系统部署的周期长、运维也比较难。

为了满足运营商云化的需求，华为相继研发了IaaS、PaaS等用于支撑运营商IT和基础设施云化，下面让我们一起看下华为软件云化后的逻辑架构：



以分布式、云化为核心的V4架构

> 第四代技术架构以分布式、云化为核心，相比于前三代架构，它的核心特性如下：1) 采用分布式技术构建，所有的中间件都没有单点，支持线性增长和弹性伸缩；
>
> 2) 以微服务架构为核心，打造电信领域的DevOps(结合华为PaaS平台);
>
> 3) 由传统的SOA Governance 向微服务治理和自治演进，提升服务治理效能；
>
> 4) 分布式日志采集 + 实时流式计算框架，更快的故障定界，提升大规模、分布式系统中的运维效率；
>
> 5) 业务和数据的拆分，分而治之，通过分布式中间件服务向业务屏蔽拆分细节；
>
> 6) 架构云化带来的巨大优势：资源池提升硬件利用率、DevOps提升开发和运维效率、应用和服务的自动弹性伸缩、应用和服务故障自动恢复、高HA、自动化运维等。

架构演进中的技术

> 随着架构的演进，Java的版本也在不断升级，技术堆栈不断更新，EJB、Spring、RMI、MQ、Node.js、NIO、Hadoop等。在众多技术堆栈中，我印象最深的就是Java的NIO类库以及业界成熟NIO框架的使用，它在华为软件架构演进中发挥了重大作用，曾立下了汗马功劳。现在，以Netty为代表的NIO框架已经在华为平台产品和业务产品中得到了广泛的应用。

作为华为软件公司最早使用Java NIO技术进行平台开发、2009年即在全球商用成功的亲历者和实践者，我想跟大家分享下Java NIO框架在华为软件以及电信领域的应用和实践。

Java NIO 技术的引入

BIO带给我们深深伤痛

在2008年的时候，我参与设计和开发的一个电信系统在月初出帐期，总是发生大量的连接超时和读写超时异常，业务的失败率相比于平时高了很多，报表中的很多指标都差强人意。后来经过排查，发现问题的主要原因出现在下游网元的处理性能上，月初的时候BSS出帐，在出帐期间BSS系统运行缓慢，由于双方采用了同步阻塞式的HTTP+XML进行通信，导致任何一方处理缓慢都会影响对方的处理性能。按照故障隔离的设计原则，对方处理速度慢或者不回应答，不应该影响系统的其他功能模块或者协议栈，但是在同步阻塞I/O通信模型下，这种故障传播和相互影响是不可避免的，很难通过业务层面解决。

受限于当时Tomcat和Servlet的同步阻塞I/O模型，以及在Java领域异步HTTP协议栈的技术积累不足，当时我们并没有办法完全解决这个问题，只能通过调整线程池策略和HTTP超时时间来从业务层面做规避。由于我们的系统是一个全国级的一级系统，需要对接周边各个网元，同时服务器资源十分有限，即便采用了高峰期间动态修改超时时间、优化线程池模型等多种措施，效果依然差强人意。

每当跟客户开会的时候，客户总会提起这个话题：别人响应慢，为啥会导致你的系统阻塞呢，可以返回处理其它消息啊？！我无法跟客户解释技术细节，因为同步阻塞I/O仅仅是Java I/O的一种实现，操作系统支持非阻塞I/O和异步I/O。

站在技术的角度，客户的需求是合理并且也是可以实现的，当时受限于经验以及其它技术原因，我们无法从根本上解决客户提出的问题，团队有种深深的挫败感，Java BIO同步阻塞通信导致的各种问题给我留下了一些心理阴影，一直挥之不去。

BIO模型存在的问题

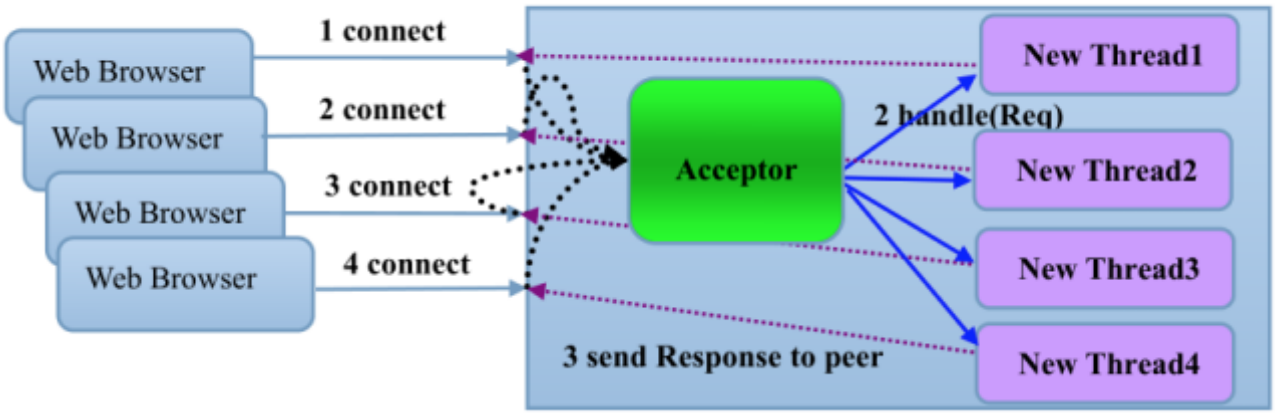
传统同步阻塞通信面临的主要问题如下：

1) 性能问题：一连接一线程模型导致服务端的并发接入数和系统吞吐量受到极大限制；

2) 可靠性问题：由于I/O操作采用同步阻塞模式，当网络拥塞或者通信对端处理缓慢会导致I/O线程被挂住，阻塞时间无法预测；

3) 可维护性问题：I/O线程数无法有效控制、资源无法有效共享（多线程并发问题），系统可维护性差

传统同步阻塞通信的处理模型图如下：



同步阻塞通信模型处理模型图

从上图我们可以看出，每当有一个新的客户端接入，服务端就需要创建一个新的线程（或者重用线程池中的可用线程），每个客户端链路对应一个线程。当客户端处理缓慢或者网络有拥塞时，服务端的链路线程就会被同步阻塞，也就是说所有的I/O操作都可能被挂住，这会导致线程利用率非常低，同时随着客户端接入数的不断增加，服务端的I/O线程不断膨胀，直到无法创建新的线程。

同步阻塞I/O导致的问题无法在业务层规避，必须改变I/O模型，才能从根本上解决这个问题。

历史性的引入Java NIO

Java NIO被冷落的原因

从2004年JDK1.4首次提供NIO 1.0类库到现在，已经过去了整整10年。JSR 51的设计初衷就是让Java能够提供非阻塞、具有弹性伸缩能力的异步I/O类库，从而结束Java在高性能服务器领域的不利地位。然而，在相当长的一段时间里，Java的NIO编程并没有流行起来，究其原因如下。

大多数高性能服务器，被C和C++语言盘踞，由于它们可以直接使用操作系统的异步I/O能力，所以对JDK的NIO并不关心；

移动互联网尚未兴起，基于Java的大规模分布式系统极少，很多中小型应用服务对于异步I/O的诉求不是很强烈；

高性能、高可靠性领域，例如银行、证券、电信等依然以C++为主导，Java充当打杂的角色，NIO暂时没有用武之地；

当时主流的J2EE服务器，几乎全部基于同步阻塞I/O构建，例如Servlet、Tomcat等，由于它们应用广泛，如果这些容器不支持NIO，用户很难具备独立构建异步协议栈的能力；

异步NIO编程门槛比较高，开发和维护一款基于NIO的协议栈对很多中小型公司来说像是一场噩梦；

业界NIO框架不成熟，很难商用；

国内研发界对NIO的陌生和认识不足，没有充分重视。

基于上述几种原因，NIO编程的推广和发展长期滞后，特别是国内，在2009年的时候，几乎无法搜到国内企业成功使用NIO技术的案例。

华为软件引入Java NIO的原因

从2008年开始，华为软件研发了Java版的业务网关，并迅速占领国内外市场。随着产品的推广，在一些高并发、大业务量的局点相继出现了几起事故，质量回溯的结果都指向了Java BIO通信模型，包括Servlet 2.X的同步阻塞I/O、Tomcat 5.X（当时没使用5.5）的同步I/O、以及其它的同步I/O协议栈。

问题根因已经很清楚，如果不改变同步I/O通信模型，问题会继续发生，对于运营商而言，这是不可能接受的事情。自古华山一条路，即然业界没有成熟的异步I/O协议栈，那我们就自研。

2009年初，由于对技术的热爱，我作为业务骨干被领导派去参加异步高性能网关平台的研发工作，与两位资深的架构师（其中一位工作20年，做华为交换机出身）一起合作。这是我第一次全面接触异步I/O编程和高性能电信级协议栈的开发，眼界大开——异步高性能内部协议栈、异步HTTP、异步SOAP、异步SMPP……所有的协议栈都是异步非阻塞模式。

后来的性能测试表明：基于Reactor模型统一调度的长连接和短连接协议栈，无论是性能、可靠性还是可维护性，都可以“秒杀”传统基于BIO开发的应用服务器和各种协议栈，这种指标差异本质上是一种技术代差。

2009年底，基于异步网关平台研发的XX业务产品在海外某运营商成功上线，它的高性能、低时延和高HA令局方惊叹不已，原来准备的20多台小机最后只使用了3台，为客户节省了一大把$。

那些年我们踩过的NIO “坑”

在我从事异步NIO编程的2009年，业界还没有成熟的NIO框架，那个时候Mina刚刚开始起步，功能和性能都达不到商用标准。最困难的是，国内Java领域的异步通信还没有流行，整个业界的积累都非常少。那个时候资料匮乏，能够交流和探讨的圈内人很少，一旦踩住“地雷”，就需要

夜以继日地维护。在随后2年多的时间里，经历了10多次的在通宵、凌晨被一线的运维人员电话吵醒等种种磨难之后，我们自研的NIO框架才逐渐稳定和成熟。期间，解决的BUG总计20~30个。

为了解决这些Bug，2年中我经历了10几个通宵，现在回想起来仍历历在目，特别是JDK epoll空轮询导致的 CPU 100%，更是坑中之坑（JDK NIO类库的Bug），曾令多少产品中招，包括Mina、Netty、Jetty等著名开源框架。

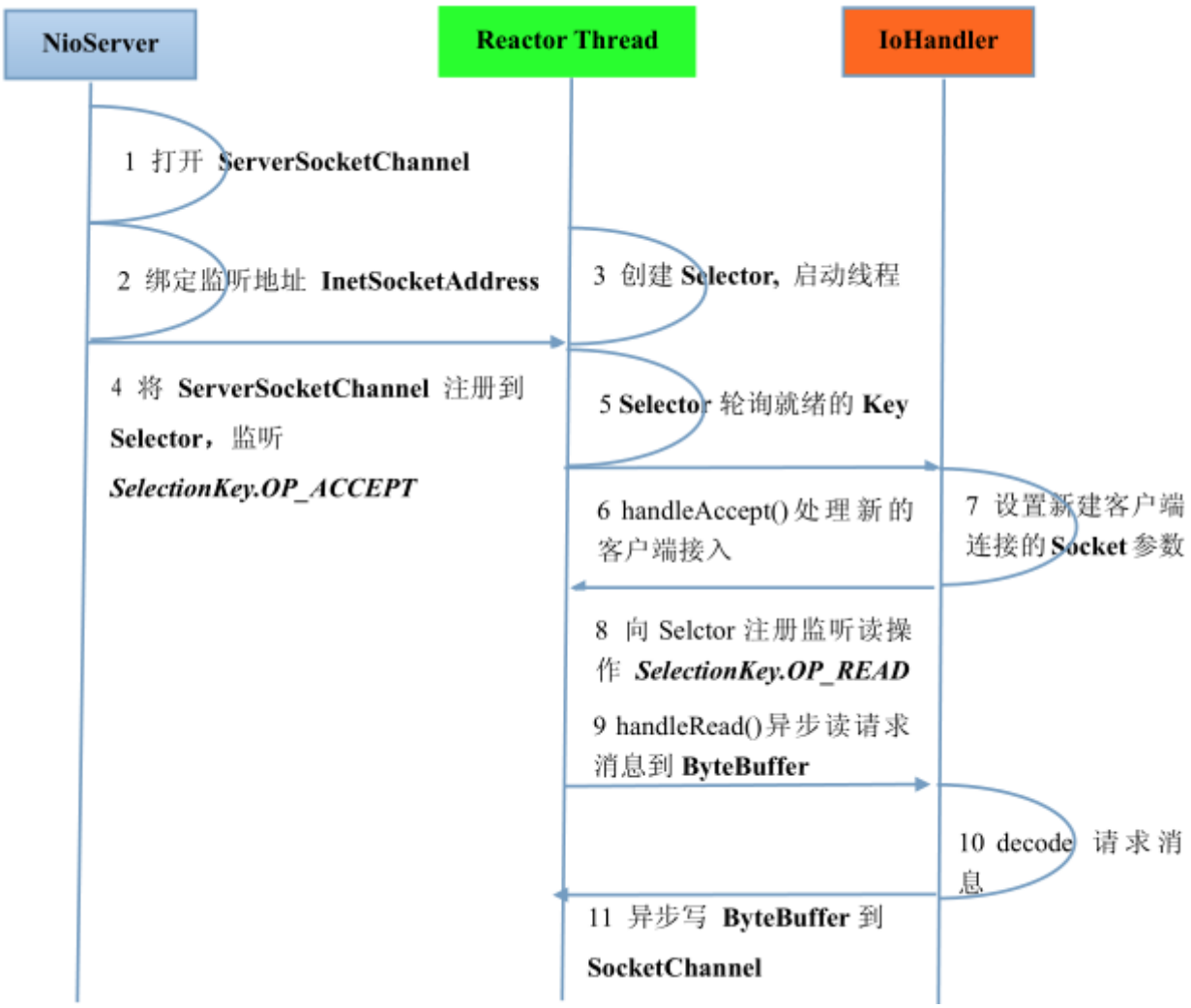
从Java 原生NIO到NIO框架

为了更好地向读者输出更优质的内容，InfoQ将精选来自国内外的优秀文章，经从2011年开始，华为软件主要使用NIO框架Netty进行通信软件的开发，为什么不继续使用原声的Java NIO类库，下面给出了我们切换的原因。

JAVA 原生NIO类库的复杂性

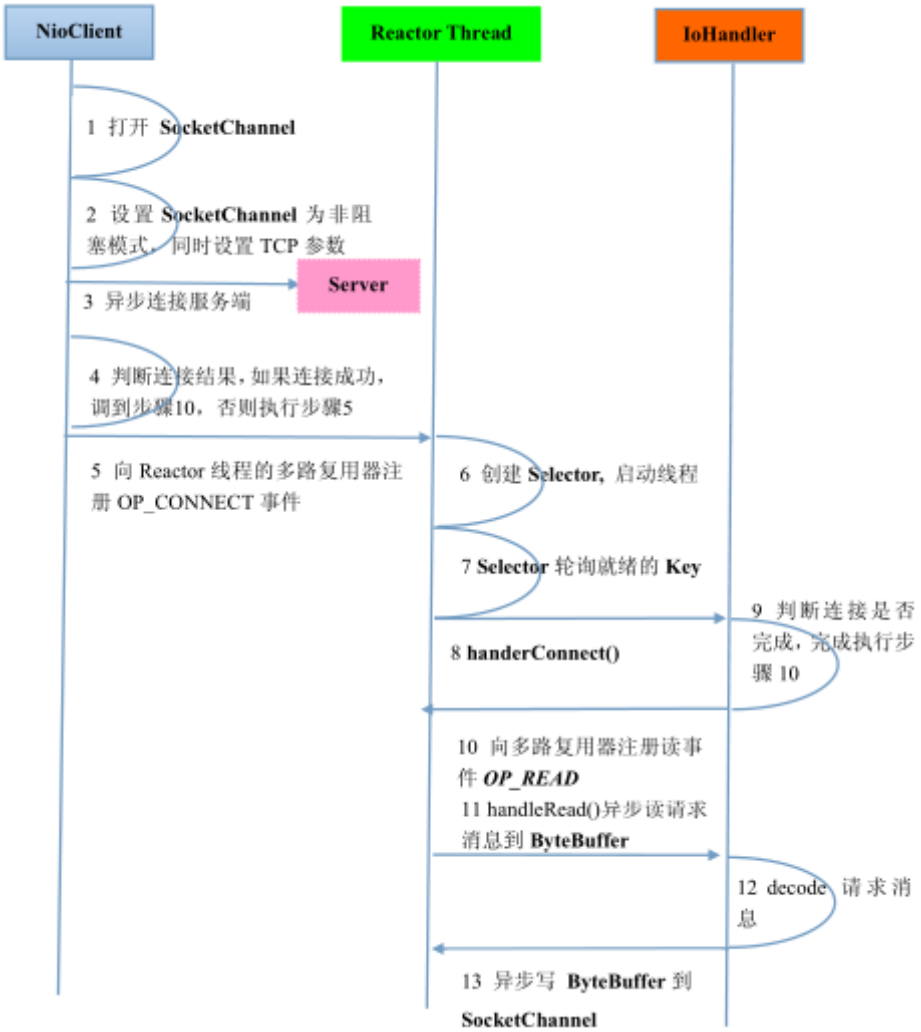
在分析Java原生NIO类库复杂性之前，我们首先看下最简单的NIO服务端和客户端创建流程。

最简单的NIO服务端创建程序流程：



Java NIO 服务端创建流程

最简单的Java NIO 客户端创建流程如下：



Java NIO 客户端创建流程

现在我们总结一下为什么不建议开发者直接使用JDK的NIO类库进行开发，具体原因如下：

（1）NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等；

（2）需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序；

（3）可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大；

（4）JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有被根本解决。

异常堆栈如下：

```
java.lang.Thread.
State: RUNNABLE at sun.nio.ch.
EPollArrayWrapper.
epollWait(Native Method)at sun.nio.ch.
EPollArrayWrapper.poll
(EPollArrayWrapper.java:210)
 at sun.nio.ch.
EPollSelectorImpl.doSelect
(EPollSelectorImpl.java:65)
 at sun.nio.ch.SelectorImpl.
lockAndDoSelect
(SelectorImpl.java:69)
 - locked <0x0000000750928190>
(a sun.nio.ch.Util$2)- locked <0x00000007509281a8>
(a java.util.Collections$
 UnmodifiableSet)
 - locked <0x0000000750946098>
 (a sun.nio.ch.EPollSelectorImpl)
 at sun.nio.ch.SelectorImpl.
select(SelectorImpl.java:80)at net.spy.memcached.
MemcachedConnection.
handleIO(Memcached
Connection.java:217)
        at net.spy.memcached.
MemcachedConnection.run
(MemcachedConnection.
java:836)
```

以Netty为代表的NIO框架已经成熟

Netty是业界最流行的NIO框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商用项目验证，例如Hadoop的RPC框架avro使用Netty作为底层通信框架；很多其他业界主流的RPC框架，也使用Netty来构建高性能的异步通信能力。

通过对Netty的分析，我们将它的优点总结如下：

1) API使用简单，开发门槛低；

2) 功能强大，预置了多种编解码功能，支持多种主流协议；

3) 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；

4) 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；

5) 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；

6) 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；

7) 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它已经完全能够满足不同行业的商业应用了。

正是因为这些优点，Netty逐渐成为Java NIO编程的首选框架，它也是华为公司首选的Java NIO通信框架，公司已经将其纳入到公司级的优选开源第三方软件库中。

Netty在电信领域的实践

电信行业软件的几个特点：

1) 高可靠性：5个9；

2) 高性能、低时延；

3) 大规模组网：例如中国移动、Telfonica 拉美十三国、沃达丰等，业务组网规模都非常大；

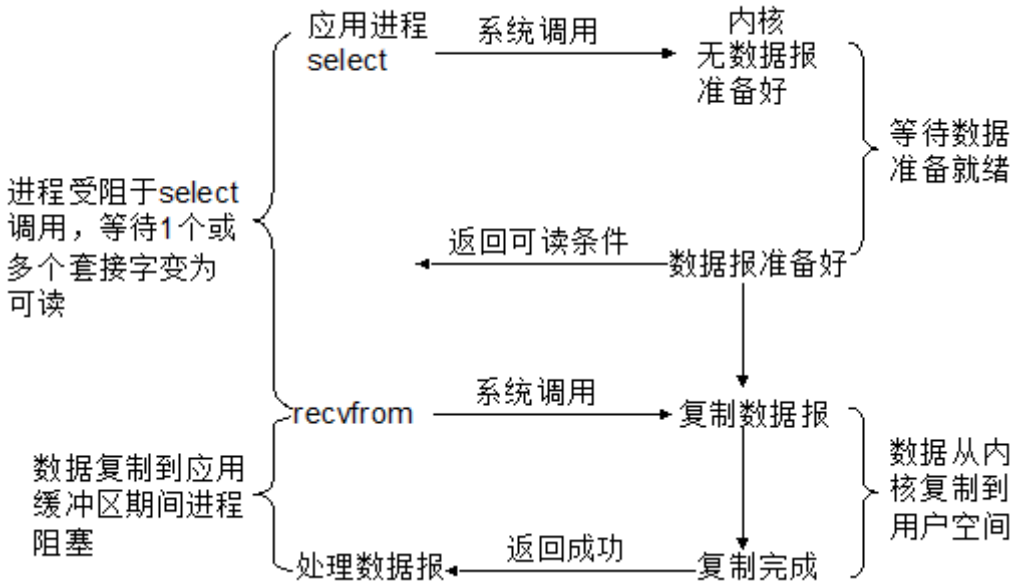
4) 复杂的网络形态：对接不同设备提供商的网元和系统。

高性能、低时延

非阻塞I/O模型

在I/O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I/O多路复用技术进行处理。I/O多路复用技术通过把多个I/O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。

我们采用**Netty**的**NIO**传输模式来提升**I/O**操作的效率，节省线程等其它资源开销，它的模型如下所示：



Netty的非阻塞I/O调度模型

高性能的序列化框架

在华为软件，对于序列化框架的选择，我们遵循如下几个原则：

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、内存等资源占用）；

3) 是否支持跨语言（异构系统的对接和开发语言切换）；

4) 高并发调用时的性能，是否随着线程并发数线性增长。

基于上述的指标，目前最常用的选择是：Google的ProtoBuf和Apache的Thrift。Netty原生提供了对ProtoBuf序列化框架的支持，它的优点如下：

1) 在谷歌内部长期使用，产品成熟度高；

2) 跨语言、支持多种语言，包括C++、Java和Python；

3) 编码后的消息更小，更加有利于存储和传输；

4) 编解码的性能非常高；

5) 支持不同协议版本的前向兼容；

6) 支持定义可选和必选字段。

Netty ProtoBuf 服务端开发示例如下：

// 配置服务端的NIO线程组

```
EventLoopGroup bossGroup =
new NioEventLoopGroup();
EventLoopGroup workerGroup =
 new NioEventLoopGroup();try {ServerBootstrap b =
new ServerBootstrap();
b.group(bossGroup,
 workerGroup)
 .channel(NioServer
SocketChannel.class)
 .option(ChannelOption.
SO_BACKLOG, 100).
handler(new LoggingHandler
(LogLevel.INFO)).childHandle
(new ChannelInitializer
<SocketChannel>() {@Override
public void initChannel
(SocketChannel ch) {ch.pipeline().addLast(new ProtobufVarint
 32FrameDecoder());ch.pipeline().addLast(new ProtobufDecoder(SubscribeReqProto.
SubscribeReq.
getDefaultInstance()));ch.pipeline().addLast(new ProtobufVarint32L
 engthFieldPrepender());ch.pipeline().addLast
(new ProtobufEncoder());ch.pipeline().addLast
(new SubReqServerHandler());
                                }
                    });
```

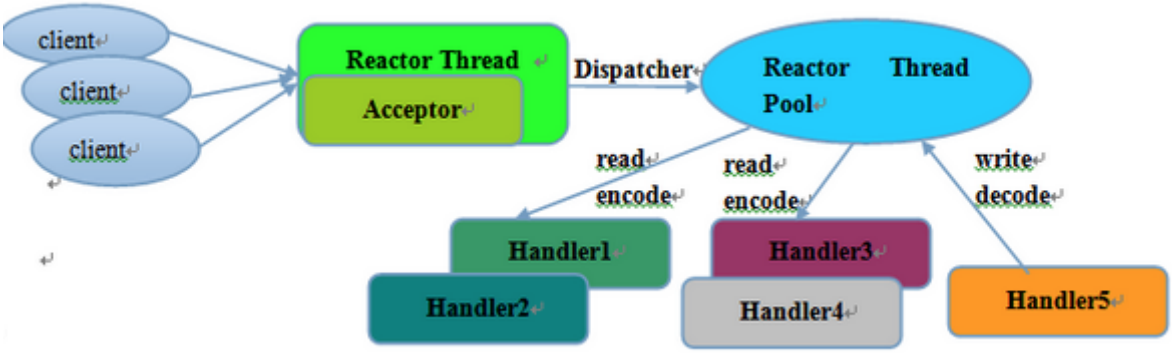
Thrift相对复杂一些，需要将编解码框架从Thrift中剥离出来，然后利用Netty编解码框架的扩展性定制实现，在此不再赘述。

收敛的Reactor线程模型

Java线程采用抢占的方式争夺CPU等资源，当系统线程数增大到一定量级之后，性能不仅没有提升，反而下降。

对于大型的电信应用，如果使用Tomcat等做Web容器，为了保证吞吐量和性能，HTTP线程池的最大线程数往往配置为1024。在系统运行期间我们Dump线程堆栈，发现大量的线程竞争，这不仅导致HTTP协议栈的性能下降，更影响其它业务处理线程的执行效率。

使用Netty之后，我们通过控制NioEventLoopGroup的NioEventLoop个数来收敛线程，防止线程膨胀。NioEventLoop聚合了一个多路复用器Selector，可以高效的处理N个Channel，它的线程模型如下：



Netty Reactor线程模型

其它优化

为了进一步提升性能，降低时延，我们还采用了其它一些优化措施，总结如下：

1) 使用Netty 4的内存池，减少业务高峰期ByteBuf频繁创建和销毁导致的GC频率和时间；

2) 在程序中充分利用Netty提供的“零拷贝”特性，减少额外的内存拷贝，例如使用CompositeByteBuf而不是分别为Head和Body各创建一个ByteBuf对象；

3) TCP参数的优化，设置合理的Send和Receive Buffer，通常建议值为64K - 128K；

4) 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；

5) 无锁化串行开发理念：使用Netty 4.X版本，天生支持串行化处理；业务开发过程中，遵循Netty 4的线程模型优化理念，防止人为增加线程竞争。

高HA

内存保护

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。

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：

1) 在内存分配的时候指定缓冲区长度上限;

2) 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;

3) 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。

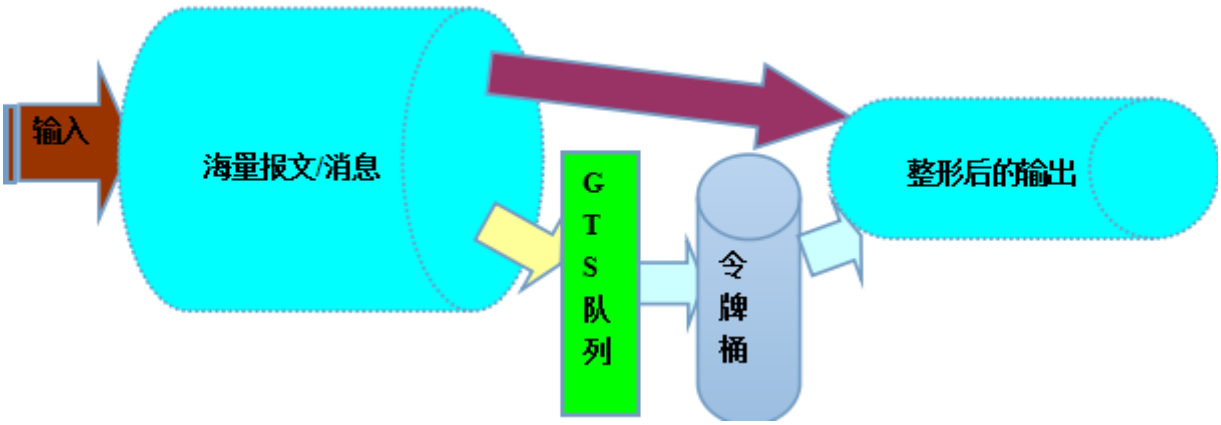
流量整形

电信系统一般都有多个网元组成，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。

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：



Netty 流量整形原理图

Netty内置两种流量整形策略，可以方便的被用户添加和使用：

1) 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；

2) 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，整形参数与全局流量整形相同。

其它可靠性措施

其它比较重要的可靠性措施如下：

1) 客户端连接超时控制策略；

2) 链路断连重连策略；

3) 链路异常关闭资源释放；

4) 解码失败的异常处理策略；

5) 链路异常的捕获和处理；

6) I/O线程的释放。

华为软件对Netty的优化

针对电信软件的特点，结合华为软件的实际业务需求，我们对Netty进行了优化，优化的策略如下：

1) 能够通过Netty提供的扩展点实现的，通过扩展点实现，不自己造轮子；

2) 不允许修改Netty源码，基于Netty提供的接口，开发华为自己的优化实现类；

3) 华为优化实现类独立打包，对原Netty类库是二进制依赖，不修改Netty原类库；

4) 服务端和客户端创建时，传递华为自己的实现类参数。

华为的主要优化点总结如下：

1) 安全性改造：满足华为公司安全红线、电信运营商的安全需求相关改造；

2) 可靠性增强：消息发送队列的上限保护、链路中断时缓存中待发送消息回调通知业务、增加错误码、异常日志打印抑制、I/O线程健康度检测等；

3) 可定位性增强：单链路的网络吞吐量、接收发送的速度、接收\发送的总字节数、畸形码流检测机制、解码时延超大消息日志打印等。

作者简介

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7年NIO设计和开发经验，精通Netty、Mina等NIO框架和平台中间件，现任华为软件平台架构部架构师，《Netty权威指南》作者。目前从事华为下一代中间件和PaaS平台的架构设计工作。

联系方式：新浪微博 Nettying 微信：Nettying 微信公众号：Netty之家

对于Netty学习中遇到的问题，或者认为有价值的Netty或者NIO相关案例，可以通过上述几种方式联系我。

感谢郭蕾对本文的审校。

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，@丁晓昀），微信（微信号：InfoQChina）关注我们，并与我们的编辑和其他读者朋友交流

React | 架构师 | 运维 | 云 | 开源 | Kubernetes | 架构 | 人工智能 | Kafka | Docker | Netty | CoreOS | QCon | Github | Swift | 敏捷 | 语言 | 程序员 | 实践 | 物联网 |

如果想要评论本篇文章，想看下其他读者都有什么话想说，欢迎点击“阅读原文”参与讨论。

极客邦科技

InfoQ | EGO | StuQ

阅读原文