

架构师

ARCHITECT



热点 | Hot

Java 9正式发布,新特性解读

百度正式开源其RPC框架brpc

推荐文章 | Article

Kafka数据可靠性深度解读

MySQL到底能不能放到Docker里跑?

观点 | Opinion

虚拟座谈会:聊聊AIOps的终极价值

软件测试技术的未来



CONTENTS / 目录

热点 | Hot

Java 9 正式发布，新特性解读

百度正式开源其 RPC 框架 brpc

推荐文章 | Article

Kafka 数据可靠性深度解读

MySQL 到底能不能放到 Docker 里跑？

观点 | Opinion

虚拟座谈会：聊聊 AIOps 的终极价值

软件测试技术的未来

AI 专区 | AI Area

gRPC 客户端创建和调用原理解析



架构师 2017 年 10 月刊

本期主编 蔡芳芳

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

硅谷人工智能背后的 技术秘密

2017年10月17-19日 上海·宝华万豪酒店

更多技术实践请查看大会官网
www.qconshanghai.comGoogle [Some Cool Progress of Tensorflow and Deep Learning](#)周玥枫
Google 大脑工程师

用谷歌人工智能造聊天机器人

罗昶
Google 高级工程师Facebook [Personalize News Feed with Content Signal](#)Meihong Wang
FaceBook Newsfeed Director of Engineering

Facebook实时数据连接及模型训练系统的演进

李友林
Facebook 工程部高级技术经理Paypal [卓越的数据科学团队如何驱动金融级的风险与安全管理](#)王辉
PayPal 全球风险管理建模团队

PayPal 风险控制建模之路

尹华罡
PayPal Manager, Data ScienceUber [Intelligent Dispatch for UberEATS](#)翟鹏
Uber Tech Lead, Sr. Software Engineer II

实时数据分析平台

付翔
Uber 实时数据平台架构师, 团队技术主管LinkedIn [Nuage-LinkedIn 如何使其分布式数据系统更易用](#)Eric Kim
LinkedIn 数据基础设施团队高级经理Airbnb [BDT: Boosting 决策表](#)娄寅
Airbnb 机器学习工程师Pinterest [Pinterest 如何利用机器学习来获得两亿全球月活跃用户](#)郭云松
Pinterest 机器学习主管工程师Twitter [Twitter 大规模分布式存储系统的架构与实践](#)陈箫
Twitter 基础架构工程师Amazon [披萨的故事——Microservice & Serverless](#)蔡超
Amazon 中国 研发中心首席架构师腾讯 [基于卷积神经网络在手机端实现文档检测](#)冯华
腾讯 iOS 客户端开发高级工程师

方圆并济：基于 Spark on Angel 的高性能机器学习

黄明
腾讯 数据平台部 T4 专家阿里
Java 开发手册 [编程规范与团队研发效能的辩证关系](#)杨冠宝
阿里巴巴 研发效能中台-代码中心主管天猫 [创新技术（AR、渲染、架构等）在电商领域的探索](#)冯建华（冯子）
天猫 高级技术专家饿了么 [WebAssembly 核心原理、应用与未来展望](#)于航
饿了么 高级前端开发工程师

异地多活的基础设施建设

兰建刚
饿了么 框架工具部研发总监Zilliz [100X Faster Data Analytics by GPU Acceleration](#)星爵
Zilliz 创始人

AWS云计算开发实践专场

出品人: 费良宏
AWS 首席云计算技术顾问



《AWS 上的 MXNet 深度学习框架》
费良宏 / AWS 首席云计算技术顾问



《流利说如何在 AWS 上打造生产级 Kubernetes 平台》
韩冰 / 英语流利说后端工程师



《基于 AWS 的 Spring Cloud 微服务实践》
薛军 / AWS 解决方案架构师



《AWS 物联网与边缘计算的最佳实践和客户案例》
陈雪杰 / AWS 解决方案架构师



扫码即可报名与AWS专家面对面

卷首语

论 AI 时代的融合型人才

作者 腾讯 Andymhuang (黄明)

在刚刚过去的几个月，发生了一些有趣的事情。IEEE Spectrum 调查显示 Python 成为了最受欢迎的语言，NIPS 2017 的门票创造了最快售罄记录，Apple 发布会推出了 FaceID，GTC China 2017 异常火爆……所有的这些迹象，都预示着人工智能正在迎来它的夏天。

伴随着 AI 时代的到来，AI 人才，尤其是算法工程师和数据科学家，受到前所未有的高度关注（也产生了一定的泡沫）。但是这不代表擅长工程的架构师就不重要了。再好的算法和模型，依然需要坚实的架构和平台来实现，否则就是镜花水月。在我们看来，优秀架构师的重要性一点也没降低，但是他们也需要懂一点 AI 了。其实仔细研究的话，很容易会发现，**工程型的架构师和算法型的科学家**，这两类人有完全不同的特性。

- 优秀的工程型人才，熟悉计算机系统和理论，有1-2门拿手的语言，能写出优雅而高效的代码，设计高可用的架构。讲究快、稳、准，应对千亿压力而从容自如。
- 优秀的算法型人才，精通各种数学公式和推导，了解各种优化理



论，对于一个或者多个领域的模型有深入的研究。讲究深、精、妙，直达事物之本质而化繁为简。

在 AI 时代之前，这两种人才某种程度上是井水不犯河水，大部分工程型人才在业界奋斗，开发各种应用、Web 网站、无线 App，而算法型人才活跃于学术界，进行各种科学研究、理论实验、发表文章。但是当 AI 时代到来时，人们发现，或主动或被动，大家需要交流融合了。

- 工程型人才，如果不懂一点算法，那么做出来的系统依然和AI绝缘，只怕是再快也无济于事。
- 算法型人才，如果不懂一点架构，那么写出来的算法、设计出来的模型，在大数据下根本行不通，再精妙也不过是纸上谈兵。

因此，融合型人才在 AI 时代会比之前的所有的时代更受欢迎。经典的学术界会议纷纷得到各大互联网厂商的大力赞助，而老牌的业界会议也纷纷引入越来越多的学术研究议题。这个趋势在本次 QCon 上海 2017 大会上也得到了很好的体现：跨界互通，道器相融。希望有越来越多融合型人才和团队出现，推动 AI 的发展，为大家提供更好的 AI 产品和服务。

Java 9 正式发布，新特性解读

作者 杨晓峰



在历经多次跳票之后，Java 9 终于在千呼万唤中正式发布。从这个版本开始，Java 将每半年发布一个版本。作为霸占编程语言排行榜鳌头多年的老牌语言，Java 9 中有哪些不得不说的新特性？Java 语言的未来又将如何？

针对 Java 9 新特性的介绍已经非常多了，我这里不想再做一个百科全书一样的列表，希望从不同角度简要点评部分特性。

Jigsaw

首先，谈到 Java 9 大家往往第一个想到的就是 Jigsaw 项目，这是一个雄心勃勃的项目。

大家知道，Java 已经发展超过 20 年（95 年最初发布），Java 和

相关生态在不断丰富的同时也越来越暴露出一些问题，比如 Java 运行环境的膨胀和臃肿，各种类库和工具在提供强大功能的同时，也越来越复杂，不同版本的类库交叉依赖导致 Jar Hell 等让人头疼的问题，这些都阻碍了 Java 开发和运行效率的提升。

但是由于兼容性等各方面的掣肘，对 Java 进行大刀阔斧的革新越来越困难，Jigsaw 从 Java 7 阶段就开始筹备，Java 8 阶段进行了大量工作，终于在 Java 9 里落地，有种千呼万唤始出来的意味。

Jigsaw 项目的目标是改进 Java SE 平台，使其可以适应不同大小的计算设备；改进其安全性，可维护性，提高性能；简化各种类库和大型应用的开发和维护。

这个项目的工作量和难度大大超出了初始规划。JSR 376 Java 平台模块化系统（JPMS，Java Platform Module System）作为 Jigsaw 项目的核心，其主体部分被分解成 6 个 JEP（JDK Enhancement Proposals）

- 200: The Modular JDK
- 201: Modular Source Code
- 220: Modular Run-Time Images
- 260: Encapsulate Most Internal APIs
- 261: Module System
- 282: jlink: The Java Linker

可以看到这是一个庞大的系统工程，Java 的方方面面，包括 JDK 编译工具，运行时，Java 公共 API 和私有代码等等，完全是一个整体性的改变。

随着 Java 平台模块化系统的落地，开发人员无需再为不断膨胀的 Java 平台苦恼，例如，您可以使用 jlink 工具，根据需要定制运行时环境。这对于拥有大量镜像的容器应用场景或复杂依赖关系的大型应用等，都具有非常重要的意义。

从软件开发实践的角度，Java 语言层面提供对模块的支持，可以鼓励（当然在某种程度上也可以看作强制）更加规范的开发实践，利用业界

在开发领域几十年的经验、教训总结出的最佳实践，促进 Java 生态的健康发展。比如，更加完善的隐藏实现细节，这不仅可以促进面向接口、约定的编程，也可以避免可能的安全风险等。

不过，换个角度来说，天下没有免费的午餐，由于 JPMS 是语言平台层面的支持，它并不是完全透明的，也就是说不管用户是否真的需要或从中收益，都会或多或少的受其影响。

对此，我们可以从 JPMS 评审中针对类似深度反射限制之类的激烈争吵中，深刻体会到。比如，针对反射访问控制，最终 Java 9 开发团队，采取了相对折中的办法，在反射领域默认保持 Java 8 的默认行为。Java 9 在兼容性方面，相比于过往的版本，采取了更大的容忍度。

不过，Java 9 的相当一部分特性仍然是对用户透明的。只要升级到 Java 9，不需要或者很少需要用户参与动作就能获益。比如，更加紧凑的字符串实现；改进的竞争锁机制；改进安全应用性能；利用特定 CPU 指令优化 GHASH 和 RSA 等等，这些都是开箱即用、触手可得的改进。

Java 9 值得关注的新特性

对于部分开发者来说，探究 Java 内部 API 或者平台底层能力是一件非常酷的事情，但这往往并不是非常容易，比如部分能力可能并没有在历史版本的公共 API 中暴露出来（比如 Unsafe 相关），或者需要特定领域的知识。在 Java 9 中，不要错过 JEP 193: Variable Handles 和 JEP 274: Enhanced Method Handles，JEP 259: Stack-Walking API，JEP 285: Spin-Wait Hints 等特性。

另外，Java 9 中还有很多承上启下的特性，为未来创新打下基础或者整合、规范现有碎片化的功能，我会介绍一些有代表性的新特性。

在 Java 虚拟机领域，JEP 271: Unified GC Logging 和 JEP 158: Unified JVM Logging，对各种 JVM 日志进行了统一，大家终于不用为各种碎片化的日志选项苦恼了。

Oracle 一直在努力提高 Java 启动和运行时性能，希望其能够在更

广泛的场景达到或接近本地语言的性能。但是，直到今天，谈到 Java，很多 C/C++ 开发者还是会不屑地评价为启动慢，吃内存。

简单说，这主要是因为 Java 编译产生的类文件是 Java 虚拟机可以理解的二进制代码，而不是真正的可执行的本地代码，需要 Java 虚拟机进行解释和编译，这带来了额外的开销。

JIT (Just-in-time) 编译器可以在运行时将热点编译成本地代码，但是实际应用可能非常庞大，大型 Java 应用的预热往往非常耗时，而且非热点代码可能根本没有机会被 JIT 编译。

在 JDK 9 中，AOT (JEP 295: Ahead-of-Time Compilation) 作为实验特性被引入进来，开发者可以利用新的 jaotc 工具将重点代码转换成类似类库一样的文件，这样会大大降低启动开销。

另外 JVMCI (JEP 243: Java-Level JVM Compiler Interface) 等特性，对于整个编程语言的发展，可能都具有非常重要的意义，虽然未必引起了广泛关注。目前 Graal Core API 已经被集成进入 Java 9，虽然还只是初始一小步，但是完全用 Java 语言来实现的可靠的、高性能的动态编译器，似乎不再是遥不可及，这是 Java 虚拟机开发工程师的福音。

与此同时，随着 Truffle 框架和 Substrate VM 的发展，已经让个别信心满满的工程师高呼“One VM to Rule Them All!”，也许就在不远的将来 Ploygot 以一种另类的方式成为现实。

谈谈 Java 的未来

前面简短地谈了谈 Java 9 中的一些令人激动的特性，Java 9 在取得这些进步的同时，那么在其的研发过程中有哪些教训，当前和未来遇到了那些挑战呢？

首先，就是如何更加快速、敏捷地进行创新。在 Java 9 的开发过程中，非常突出的一点就是，由于 Jigsaw 项目的延期，导致 Java 9 的发布一再推迟，这带来了很多负面影响。大批特性已经完成多时，却无法及时被实际应用采纳，开发者无法及时地从中获益，也很难尽早发现和反馈可能

存在的问题或改进。这不禁让人反思 Java 传统的研发模式的局限性。

针对这些情况，Java 首席架构师 Mark Reinhold 已经发出倡议，建议从传统的以特性驱动的发布周期，转变为以时间驱动的（6 个月为周期）发布模式，并逐步的将 Oracle JDK 原有商业特性进行开源，Java Flight Recorder 等杀手级工具和特性，一定会大受开发者的欢迎。针对企业客户的需求，Oracle 将以三年为周期发布长期支持版本（long term support）。

第二，随着云计算和 AI 等技术浪潮，当前的计算模式和场景正在发生翻天覆地的变化，不仅对 Java 的发展速度提出了更高要求，也深刻影响着 Java 技术的发展方向。传统的大型企业或互联网应用，正在被云端，容器化应用、模块化的微服务甚至是函数（FaaS，Function-as-a-Service）所替代。

Java 需要在新的计算场景下，改进开发效率。这话说的有点笼统，我谈一些自己的体会，Java 代码虽然进行了一些类型推断等改进，更易用的集合 API 等，但仍然给开发者留下了过于刻板、形式主义的印象，这是一个长期的改进方向，例如，JEP 286: Local-Variable Type Inference；持续改进并发计算框架，Java 的并发特性非常强大和系统，但某种程度上过于复杂，在今年的 JVMLS 上，阿里巴巴 AJDK 组介绍了利用协程改进并发的实践，这是一个令人眼前一亮的创新；Java 非常需要更加友好的本地代码支持，相关的特性有很多好的想法和尝试，比如 Panama 项目；Value Types 和改进的泛型，有兴趣可以参考 Valhalla 项目。

最后，进一步改进启动和运行性能、优化计算资源使用。目前，相当一部分的 Java 类库和虚拟机特性都是针对长时间、大数据量、高并发等复杂任务进行的优化，但是在部分云计算场景中，比如越来越引起大家关注的 FaaS 应用，短时间、无状态的函数正在成为常见的计算单元。那么在这种场景下，Java 必须进行相应的改进和创新，才能保持和强化目前在软件开发领域的竞争力。比如，提高 Java 运行时启动速度，尤其是在

容器环境的初始化表现；保证 CPU 等计算资源调度能力能够适应容器环境的新情况，最直接的就是 Java 平台需要支持基于 cgroup 等技术的资源管理；针对新场景下的 GC 优化；如何提高数据密度和计算效率等等。

以上很多方面往往不是孤立的，也不是非常简单就可以完成的，很多改进都是依赖于相关语言基础技术的进步和突破，Java 的进步需要持之以恒的耐心和持续的努力与投入。

最后，欢迎大家能够参与到 OpenJDK 社区，Java 是大家的，欢迎您向 OpenJDK 提供建议、意见或者直接提交自己的改进，在社区中听见越来越多的来自中国的声音是非常令人高兴的事情，让我们携手促进 Java 的创新和发展。

百度正式开源其 RPC 框架 brpc

作者 郭蕾



9月14日，百度正式在GitHub上基于Apache 2.0协议开源了其[RPC框架brpc](#)。brpc是一个基于protobuf接口的RPC框架，在百度内部称为“baidu-rpc”，它囊括了百度内部所有RPC协议，并支持多种第三方协议，从目前的性能测试数据来看，brpc的性能领跑于其他同类RPC产品。

brpc开发于2014年，主要使用的语言是C++和Java，是百度内部使用最为广泛的RPC框架，它经受了高并发高负载的生产环境验证，并支撑了百度内部大约75万个同时在线的实例。据InfoQ了解，百度内部曾有多款RPC框架，甚至在2014年时还开源过另外一款RPC框架sofa-pbrpc。那brpc是在什么样的背景下诞生的？它有什么样的优势？又为何要开源？就这些问题，InfoQ记者采访了brpc负责人戈君。

InfoQ：谈谈 brpc 的一些基本情况？什么时候开始研发的？经过了怎样的迭代和升级？目前在内部应用情况如何？

戈君：brpc 于 2014 年创建，在百度内部称为“baidu-rpc”。到目前为止，brpc 一共进行了 3000 次左右的改动，现在仍在持续优化中，百度内的 wiki 上可以查询到每次改动的描述。brpc 的主要语言是 C++ 和 Java，对其他语言的支持主要是通过包装 C++ 版本，比如 brpc 的 Python 版包含 C++ 版的大部分功能。

brpc 目前支撑百度内部大约 75 万个同时在线的实例（不含 client），超过 500 种服务（去年的统计，现在已不统计这类数据）。Hadoop、Table、Mola（另一种广泛使用的存储）、高性能计算、模型训练、大量的在线检索服务都使用了 brpc。brpc 第一次统一了百度内分布式系统和业务线的通信框架。

InfoQ：为什么百度当时要研发 brpc？

戈君：我们在实践中意识到，RPC 作为最基础的通信组件，当时的百度已经不领先了。我当时的经理刘炀曾是 Google 的工程师，非常重视基础架构的建设，也愿意在这个方向投入资源。

我们在内部会更加深入地讨论这些问题。“好用”有时看起来很主观，但其实还是有据可循的，它的关键点是能不能真正地提高用户的效率：开发、调试、维护都要考虑到，如果用户效率真的被提高了，用户会想着你的，靠吹嘘或政令推广的东西得不了人心。我们创建 brpc 的初衷是解决百度业务所面临的实际挑战，同时也希望成为百度同学最喜爱的工具，哪怕离开百度也会怀念 brpc。我们希望在提供了一个好用框架的同时，也展现了一种工作方法：注释怎么写，日志怎么打，ChangeLog 怎么写，版本怎么发布，文档怎么组织，甚至对未来不在百度的同学的工作也有帮助，所以从这点来说 brpc 从一开始就是拥抱开源的。事实上，我们在口碑上做得还不错，brpc 的 wiki 可能是百度内被点赞最多的内容之一。

InfoQ：与其他的一些开源的 RPC 框架相比，brpc 的优势是什么？

戈君：brpc 主打的是深度和易用性。一方面我们没有精力像 gRPC

那样摊大饼，什么都做。另一方面我们也注意到 gRPC（包括更早的 Thrift）的深度和易用性并不够。技术方面的东西就是这样，看示例程序，文档非常牛逼，但实战中可能就是另一回事了，为什么各个公司都要造自己的轮子，一个隐藏原因就是表面高大上的东西在一些细节上让你无法忍受。

RPC 真正的痛点是什么？是可靠性、易用性和定位问题的便利性。服务中不要出现不可解释的长尾，程序的可变项要尽量少，各种诡异问题要有工具支持快速排查。而这些在目前开源的 RPC 框架中做的并不好，它们大多看着很牛，但就是无法在自己组织中推广开来。回到前面那三点，brpc 是如何做的呢？

- 可靠性。这一方面是代码质量问题，通过为brpc团队设立很高的招聘门槛，以及在团队中深入的技术讨论，我们确保了稳固的代码基础。另一个问题是长尾问题，这是设计问题，brpc其实包含了很多模块，其中的bthread是一个M:N线程库，就是为了更好地提高并发避免阻塞。brpc中的读和写都是wait-free的，这是最高程度的并发。技术细节请点击[链接](#)查看。
- 易用性。有种设计是什么选择都做成选项丢给用户，号称功能都有，但一旦出问题，则是用户“配置错了”。而且这样用户还非常依赖开发团队，没有开发团队的支持基本用不了，开发团队有足够的理由扩充团队。这么做其实非常不负责任，用户面对海量的选项也很难受。brpc对于增加选项非常谨慎，框架能自己做判断的绝不扔给用户，所有用户选项都有最合理的默认值，不设也能用。我们认为这对用户体验来说非常重要。
- 定位问题的便利性。这点其它开源框架目前做的都不好，正常使用是可以的，但出问题就麻烦了。这个问题在百度内部其实也很严重，brpc之前用户排查问题都要拉RPC同学一起排查，RPC框架对用户是个黑盒，用户根本不知道里面发生了什么。按我们的经验，基本每天都有几个用户在群里问server卡顿，client超时之

类的问题，排查问题是常态，人手必然不够。时间长了用户就觉得你这个框架各种问题，人还拽的不行很少回他们消息。brpc的解决办法是给server内加入各种HTTP接口的内置服务，通过这些服务，用户可以很快看到server的延时、错误、连接、跟踪某个RPC、CPU热点、内存分配、锁竞争等信息，用户还可以使用bvar来自定义各类统计信息，并在百度的运维平台NOAH上汇总。这样大部分问题用户可以自助解决。其实我们去看也是看这些，只是会更加专业。内置服务的具体说明可以看[这里](#)。

InfoQ：作为公司内部的 RPC 框架，在服务治理方面有什么考虑？

戈君：百度内部 RPC 使用非常广泛，基本都是 RPC 调用，一些产品线还会通过 local RPC 隔离工程框架和策略代码。这么多年下来，服务周边的系统也比较全面了：编译是 BCLOUD，发布是 Agile，服务注册和发现是 BNS，认证是 Giano，监控和运维是 NOAH。在百度内部，brpc 和这些系统做了比较紧密的绑定，用户体验是一站式的。虽然在开源版本中，这些结合大都删掉了，但用户可以根据自己组织中的基础设施来进行定制：交互协议，名字服务，负载均衡算法都可以定制。对于其中一些特别通用的，我们希望用户反馈到开源版本中来以方便所有人。

InfoQ：之前百度还开源过 sofa-pbrpc，brpc 与它的区别是什么？

戈君：sofa-pbrpc 也是百度开发的一个比较早期的 RPC 框架，属于 sofa 编程框架的一部分，在搜索有应用。brpc 相比 sofa-pbrpc 有如下优点：

- 对协议的抽象更一般化，并统一了全百度的通信架构。brpc能容纳非常多的协议，基于Protobuf的，基于HTTP的，百度内的nshead/mcpack，开源的Redis/Memcached，甚至RTMP/FLV/HLS直播协议，brpc能逐渐地嵌入现有系统，而不需要彻底重构，但sofa-pbrpc则不具备扩展协议的能力。类似的，sofa-pbrpc也无法定制负载均衡算法，brpc默认提供round-robin、随机、一致性哈希，Locality-aware（局部性感知）四种算法，用户还能定制。

- 多线程质量更好。多线程编程是非常困难的，看起来简单的RPC遍布多线程陷阱，比如处理超时的代码可能在RPC还没发出去时就运行了；发送函数还没结束，处理回复的回调就被运行了；一个回复还在被处理另一个回复回来了，诸如此类。另外，一个异步RPC的回调里发起一个同步RPC会发生什么，带着锁做同步RPC会发生什么。这些问题我们都不能在sofa-pbrpc中找到满意的答案。
- 完备的调试和运维支持。解决这个问题的本质还在可扩展性，你如何让用户参与进来定制他们感兴趣的指标，为此我们设计了bvar，让用户能用比原子变量代价还小的方式自由地定制各种指标，用户能在浏览器上看到指标的变化曲线，或在运维平台NOAH看到汇总的监控数据。brpc还加入了大量内置服务方便用户调试程序，查看连接，在线修改gflags，追踪RPC，分析CPU热点，内存分配，锁竞争等一应俱全。

无需讳言，brpc 在诞生之初和 sofa-pbrpc 在百度内部是有竞争关系的，但就像其他地方一样，这种竞争带来了活力。类似的，brpc 和其他已经开源的 RPC 框架也是良性的竞争关系，在比拼谁能真正提高用户效率的过程中共同进步。每个用户都可以去对比代码、文档质量，接口设计，易用程度，扩展能力等，投出自己的一票。

InfoQ：谈谈 brpc 的整体架构？

戈君：技术栈无外乎是从传输层垒到应用层，就略过不讲了，具体可以去看下开源出来的文档。brpc 在架构上强调“在不牺牲易用性的前提下增强可扩展性”，比如 brpc 支持非常多的协议，在百度内部一个 brpc server 同端口可以支持二十几种协议，这对于服务的平滑迁移就非常好用。

Client 端的协议也非常多，用户用 brpc 和 bthread 用得很爽，所以我们最好能统一所有的客户端，像对 Redis 和 Memcached 的客户端支持也是在这个背景下做的，这两个客户端比官方 Client 好用多了，感兴趣的读者可以去尝试一下。但这么多协议的配置非常简单，填个字符串就

行了，比如 HTTP 就是把 ChannelOptions.protocol 设为“http”，Redis 就是“redis”。Server 端甚至不用设，它会自动判断每个 client 的协议，怎么做到的开源文档里也有。

名字服务、负载均衡也都可以定制。但为了对用户负责，我们也不鼓励“太自由”的定制，比如一点点需求的变化就要搞个新的，这时更需要想清楚本质区别是什么。这个事情我们在百度内的支持群里每天都在做，我们是开放的”乙方”，但我们也是严厉的”乙方”。

InfoQ：brpc 的性能如何？这么高的性能是怎么做到的？

戈君：性能是我们非常看中的一点，它和用户体验也是紧密联系的。好用但性能不行，或不好用但性能很牛，用户会很难受，我们不希望用户纠结。从另一个角度来看，在推广初期，我们要说服产品线用 brpc 靠什么？最直观的就是性能提升。而且这儿的性能不能停留在 benchmark 的图片上，而是能在真实应用中体现出来。开放出来的案例文档中或多或少都包含了性能提升，具体如下：

- [百度地图API入口](#)
- [联盟DSP](#)
- [ELF学习框架](#)
- [云平台代理服务](#)

和其他 RPC 框架（包含 thrift、gRPC）的性能对比可以见[这里](#)。

开源文档中概括了性能上的设计，“RPC in depth”这一节下的文档会谈的更细点。这里我不赘述，还请直接阅读[文档](#)。

InfoQ：为什么要将 brpc 开源？接下来在开源项目的迭代方面有什么计划吗？

戈君：因为马上还有不少依赖 RPC 的百度系统要开源啊。RPC 作为最基础的组件，开源不仅仅是为了自身，也是为其它开源项目铺路，比如说我们马上还会开源基于 brpc 的 RAFT 库，搭建高可用分布式系统非常方便；以及使用 brpc 的 bigflow，让流式计算变得很顺手。这些年百度对开源的认识也在不断加深，开源看似曝光了百度的核心技术，但带来的生态影

响力更重要。从 Apollo、PaddlePaddle 开始，百度真的开始拥抱开源了。brpc 的开源版和内部版很接近，只是去掉了对百度内部独有的一些基础设施的支持，我们在内网写的深入分析 RPC 技术细节的文档也都一并开源了，后续也会及时推送改动，请大家放心。这是一个活项目，不会拉个开源分支就不管了。

嘉宾介绍

百度主任架构师，brpc 主要作者，在系统编程，数据结构，设计和实现大规模分布式系统方面有广泛的实践。

Kafka 数据可靠性深度解读

作者 朱忠华



Kafka 起初是由 LinkedIn 公司开发的一个分布式的消息系统，后成为 Apache 的一部分，它使用 Scala 编写，以可水平扩展和高吞吐率而被广泛使用。目前越来越多的开源分布式处理系统如 Cloudera、Apache Storm、Spark 等都支持与 Kafka 集成。

1 概述

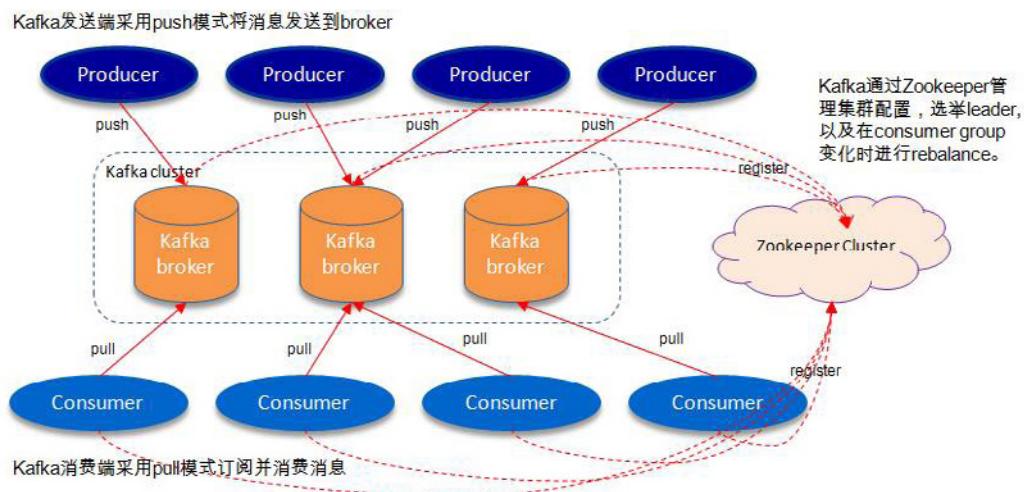
Kafka 与传统消息系统相比，有以下不同：

- 它被设计为一个分布式系统，易于向外扩展；
- 它同时为发布和订阅提供高吞吐量；
- 它支持多订阅者，当失败时能自动平衡消费者；

它将消息持久化到磁盘，因此可用于批量消费，例如 ETL 以及实时应用程序。

Kafka 凭借着自身的优势，越来越受到互联网企业的青睐，唯品会也采用 Kafka 作为其内部核心消息引擎之一。Kafka 作为一个商业级消息中间件，消息可靠性的重要性可想而知。如何确保消息的精确传输？如何确保消息的准确存储？如何确保消息的正确消费？这些都是需要考虑的问题。本文首先从 Kafka 的架构着手，先了解下 Kafka 的基本原理，然后通过对 kafka 的存储机制、复制原理、同步原理、可靠性和持久性保证等等一步步对其可靠性进行分析，最后通过 benchmark 来增强对 Kafka 高可靠性的认知。

2 Kafka 体系架构



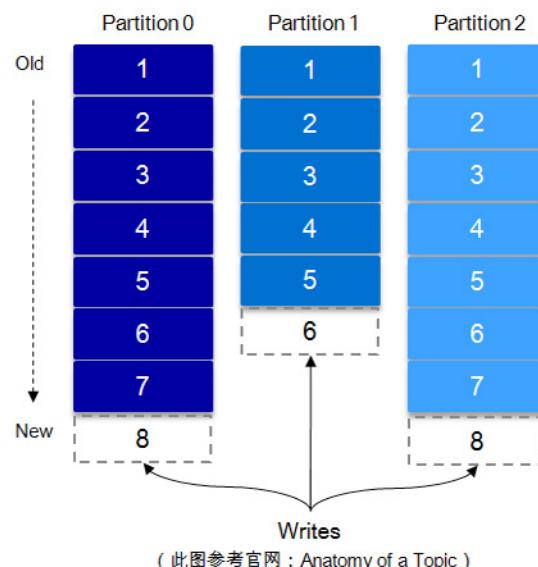
如上图所示，一个典型的 Kafka 体系架构包括若干 Producer（可以是服务器日志，业务数据，页面前端产生的 page view 等等），若干 broker（Kafka 支持水平扩展，一般 broker 数量越多，集群吞吐率越高），若干 Consumer (Group)，以及一个 Zookeeper 集群。Kafka 通过 Zookeeper 管理集群配置，选举 leader，以及在 consumer group 发生变化时进行 rebalance。Producer 使用 push(推) 模式将消息发布到 broker，Consumer 使用 pull(拉) 模式从 broker 订阅并消费消息。

名词解释：

名称	解释
Broker	消息中间件处理节点，一个 Kafka 节点就是一个 broker，一个或者多个 Broker 可以组成一个 Kafka 集群
Topic	Kafka 根据 topic 对消息进行归类，发布到 Kafka 集群的每条消息都需要指定一个 topic
Producer	消息生产者，向 Broker 发送消息的客户端
Consumer	消息消费者，从 Broker 读取消息的客户端
ConsumerGroup	每个 Consumer 属于一个特定的 Consumer Group，一条消息可以发送到多个不同的 Consumer Group，但是一个 Consumer Group 中只能有一个 Consumer 能够消费该消息
Partition	物理上的概念，一个 topic 可以分为多个 partition，每个 partition 内部是有序的

2.1 Topic & Partition

一个 topic 可以认为一个一类消息，每个 topic 将被分成多个 partition，每个 partition 在存储层面是 append log 文件。任何发布到此 partition 的消息都会被追加到 log 文件的尾部，每条消息在文件中的位置称为 offset(偏移量)，offset 为一个 long 型的数字，它唯一标记一条消息。每条消息都被 append 到 partition 中，是顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是 Kafka 高吞吐率的一个很重要的保证）。



每一条消息被发送到 broker 中，会根据 partition 规则选择被存储到哪一个 partition。如果 partition 规则设置的合理，所有消息可以均匀分布到不同的 partition 里，这样就实现了水平扩展。（如果一个 topic 对应一个文件，那这个文件所在的机器 I/O 将会成为这个 topic 的性能瓶颈，而 partition 解决了这个问题）。在创建 topic 时可以在 \$KAFKA_HOME/config/server.properties 中指定这个 partition 的数量（如下所示），当然可以在 topic 创建之后去修改 partition 的数量。

```
# The default number of log partitions per topic. More
partitions allow greater
# parallelism for consumption, but this will also result in
more files across
# the brokers.
num.partitions=3
```

在发送一条消息时，可以指定这个消息的 key，producer 根据这个 key 和 partition 机制来判断这个消息发送到哪个 partition。partition 机制可以通过指定 producer 的 partition.class 这一参数来指定，该 class 必须实现 kafka.producer.Partitioner 接口。

有关 Topic 与 Partition 的更多细节，可以参考下面的“Kafka 文件存储机制”这一节。

3 高可靠性存储分析

Kafka 的高可靠性的保障来源于其健壮的副本（replication）策略。通过调节其副本相关参数，可以使得 Kafka 在性能和可靠性之间运转的游刃有余。Kafka 从 0.8.x 版本开始提供 partition 级别的复制，replication 的数量可以在 \$KAFKA_HOME/config/server.properties 中配置 (default.replication.refactor)。

这里先从 Kafka 文件存储机制入手，从最底层了解 Kafka 的存储细节，进而对其的存储有个微观的认知。之后通过 Kafka 复制原理和同步方式来阐述宏观层面的概念。最后从 ISR, HW, leader 选举以及数据可靠性和持

久性保证等等各个维度来丰富对 Kafka 相关知识点的认知。

3.1 Kafka文件存储机制

Kafka 中消息是以 topic 进行分类的，生产者通过 topic 向 Kafka broker 发送消息，消费者通过 topic 读取数据。然而 topic 在物理层面又能以 partition 为分组，一个 topic 可以分成若干个 partition，那么 topic 以及 partition 又是怎么存储的呢？partition 还可以细分为 segment，一个 partition 物理上由多个 segment 组成，那么这些 segment 又是什么呢？下面我们来一一揭晓。

为了便于说明问题，假设这里只有一个 Kafka 集群，且这个集群只有一个 Kafka broker，即只有一台物理机。在这个 Kafka broker 中配置（\$KAFKA_HOME/config/server.properties 中）log.dirs=/tmp/kafka-logs，以此来设置 Kafka 消息文件存储目录，与此同时创建一个 topic：topic_vms_test，partition 的数量为 4（\$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 4 --topic topic_vms_test --replication-factor 4）。那么我们此时可以在 /tmp/kafka-logs 目录中可以看到生成了 4 个目录：

```
drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic_vms_test-0  
drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic_vms_test-1  
drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic_vms_test-2  
drwxr-xr-x 2 root root 4096 Apr 10 16:10 topic_vms_test-3
```

在 Kafka 文件存储中，同一个 topic 下有多个不同的 partition，每个 partition 为一个目录，partition 的名称规则为：topic 名称 + 有序序号，第一个序号从 0 开始计，最大的序号为 partition 数量减 1，partition 是实际物理上的概念，而 topic 是逻辑上的概念。

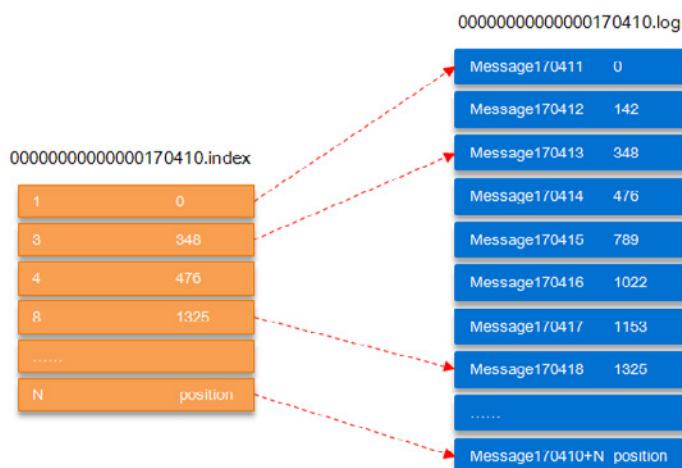
上面提到 partition 还可以细分为 segment，这个 segment 又是什么？如果就以 partition 为最小存储单位，我们可以想象当 Kafka producer 不断发送消息，必然会引起 partition 文件的无限扩张，这样对于消息文件的维护以及已经被消费的消息的清理带来严重的影响，所以这里以

segment 为单位又将 partition 细分。每个 partition(目录) 相当于一个巨型文件被平均分配到多个大小相等的 segment(段) 数据文件中(每个 segment 文件中消息数量不一定相等) 这种特性也方便 old segment 的删除, 即方便已被消费的消息的清理, 提高磁盘的利用率。每个 partition 只需要支持顺序读写就行, segment 的文件生命周期由服务端配置参数(log.segment.bytes, log.roll.{ms, hours} 等若干参数) 决定。

segment 文件由两部分组成, 分别为 “.index” 文件和 “.log” 文件, 分别表示为 segment 索引文件和数据文件。这两个文件的命名规则为: partition 全局的第一个 segment 从 0 开始, 后续每个 segment 文件名为上一个 segment 文件最后一条消息的 offset 值, 数值大小为 64 位, 20 位数字字符长度, 没有数字用 0 填充, 如下:

```
00000000000000000000.index
00000000000000000000.log
00000000000000000000170410.index
00000000000000000000170410.log
00000000000000000000239430.index
00000000000000000000239430.log
```

以上面的 segment 文件为例, 展示出 segment: 00000000000000000000170410 的 “.index” 文件和 “.log” 文件的对应关系, 如下图:



如上图，“.index”索引文件存储大量的元数据，“.log”数据文件存储大量的消息，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。其中以“.index”索引文件中的元数据 [3, 348] 为例，在“.log”数据文件表示第 3 个消息，即在全局 partition 中表示 $170410+3=170413$ 个消息，该消息的物理偏移地址为 348。

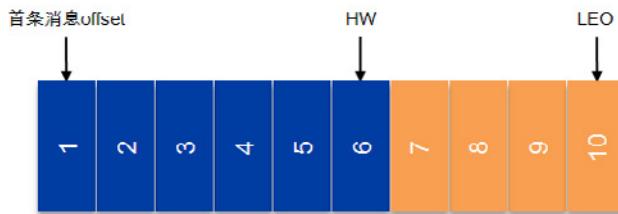
那么如何从 partition 中通过 offset 查找 message 呢？以上图为例，读取 offset=170418 的消息，首先查找 segment 文件，其中 00000000000000000000000000000000.index 为最开始的文件，第二个文件为 0000000000000000170410.index（起始偏移为 $170410+1=170411$ ），而第三个文件为 0000000000000000239430.index（起始偏移为 $239430+1=239431$ ），所以这个 offset=170418 就落到了第二个文件之中。其他后续文件可以依次类推，以其实偏移量命名并排列这些文件，然后根据二分查找法就可以快速定位到具体文件位置。其次根据 000000000000170410.index 文件中的 [8, 1325] 定位到 000000000000170410.log 文件中的 1325 的位置进行读取。

要是读取 offset=170418 的消息，从 0000000000000000170410.log 文件中的 1325 的位置进行读取，那么怎么知道何时读完本条消息，否则就读到下一条消息的内容了？这个就需要联系到消息的物理结构了，消息都具有固定的物理结构，包括：offset（8 Bytes）、消息体的大小（4 Bytes）、crc32（4 Bytes）、magic（1 Byte）、attributes（1 Byte）、key length（4 Bytes）、key（K Bytes）、payload（N Bytes）等等字段，可以确定一条消息的大小，即读取到哪里截止。

3.2 复制原理和同步方式

Kafka 中 topic 的每个 partition 有一个预写式的日志文件，虽然 partition 可以继续细分为若干个 segment 文件，但是对于上层应用来说可以将 partition 看成最小的存储单元（一个有多个 segment 文件拼接的“巨型”文件），每个 partition 都由一些列有序的、不可变的消息组成，

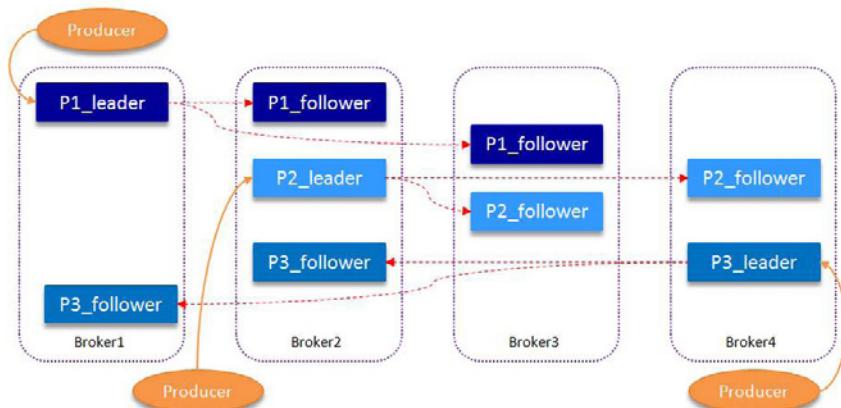
这些消息被连续的追加到 partition 中。



上图中有两个新名词：HW 和 LEO。这里先介绍下 LEO，LogEndOffset 的缩写，表示每个 partition 的 log 最后一条 Message 的位置。HW 是 HighWatermark 的缩写，是指 consumer 能够看到的此 partition 的位置，这个涉及到多副本的概念，这里先提及一下，下节再详表。

言归正传，为了提高消息的可靠性，Kafka 每个 topic 的 partition 有 N 个副本(replicas)，其中 N(大于等于 1) 是 topic 的复制因子(replica factor) 的个数。Kafka 通过多副本机制实现故障自动转移，当 Kafka 集群中一个 broker 失效情况下仍然保证服务可用。在 Kafka 中发生复制时确保 partition 的日志能有序地写到其他节点上，N 个 replicas 中，其中一个 replica 为 leader，其他都为 follower，leader 处理 partition 的所有读写请求，与此同时，follower 会被动定期地去复制 leader 上的数据。

如下图所示，Kafka 集群中有 4 个 broker，某 topic 有 3 个 partition，且复制因子即副本个数也为 3：



Kafka 提供了数据复制算法保证，如果 leader 发生故障或挂掉，一个新 leader 被选举并被接受客户端的消息成功写入。Kafka 确保从同步副本列表中选举一个副本为 leader，或者说 follower 追赶 leader 数据。leader 负责维护和跟踪 ISR (In-Sync Replicas) 的缩写，表示副本同步队列，具体可参考下节) 中所有 follower 滞后的状态。当 producer 发送一条消息到 broker 后，leader 写入消息并复制到所有 follower。消息提交之后才被成功复制到所有的同步副本。消息复制延迟受最慢的 follower 限制，重要的是快速检测慢副本，如果 follower “落后” 太多或者失效，leader 将会把它从 ISR 中删除。

3.3 ISR

上节我们涉及到 ISR (In-Sync Replicas)，这个是指副本同步队列。副本数对 Kafka 的吞吐率是有一定的影响，但极大的增强了可用性。默认情况下 Kafka 的 replica 数量为 1，即每个 partition 都有一个唯一的 leader，为了确保消息的可靠性，通常应用中将其值（由 broker 的参数 offsets.topic.replication.factor 指定）大小设置为大于 1，比如 3。所有的副本（replicas）统称为 Assigned Replicas，即 AR。

ISR 是 AR 中的一个子集，由 leader 维护 ISR 列表，follower 从 leader 同步数据有一些延迟（包括延迟时间 replica.lag.time.max.ms 和延迟条数 replica.lag.max.messages 两个维度，当前最新的版本 0.10.x 中只支持 replica.lag.time.max.ms 这个维度），任意一个超过阈值都会把 follower 剔除出 ISR，存入 OSR (Outof-Sync Replicas) 列表，新加入的 follower 也会先存放在 OSR 中。AR=ISR+OSR。

Kafka 0.10.x 版本后移除了 replica.lag.max.messages 参数，只保留了 replica.lag.time.max.ms 作为 ISR 中副本管理的参数。为什么这样做呢？replica.lag.max.messages 表示当前某个副本落后 leader 的消息数量超过了这个参数的值，那么 leader 就会把 follower 从 ISR 中删除。假设设置 replica.lag.max.messages=4，那么如果 producer 一次传

送至 broker 的消息数量都小于 4 条时，因为在 leader 接受到 producer 发送的消息之后而 follower 副本开始拉取这些消息之前，follower 落后 leader 的消息数不会超过 4 条消息，故此没有 follower 移出 ISR，所以这时候 replica.lag.max.message 的设置似乎是合理的。

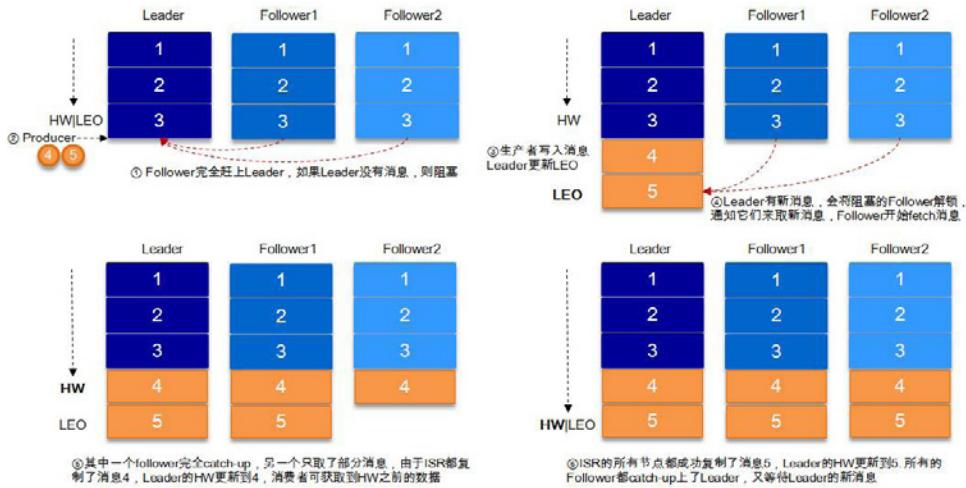
但是 producer 发起瞬时高峰流量，producer 一次发送的消息超过 4 条时，也就是超过 replica.lag.max.messages，此时 follower 都会被认为是与 leader 副本不同步了，从而被踢出了 ISR。但实际上这些 follower 都是存活状态的且没有性能问题。那么在之后追上 leader，并被重新加入了 ISR。于是就会出现它们不断地剔出 ISR 然后重新回归 ISR，这无疑增加了无谓的性能损耗。而且这个参数是 broker 全局的。设置太大了，影响真正“落后”follower 的移除；设置的太小了，导致 follower 的频繁进出。无法给定一个合适的 replica.lag.max.messages 的值，故此，新版本的 Kafka 移除了这个参数。

注：ISR 中包括：leader 和 follower。

上面一节还涉及到一个概念，即 HW。HW 俗称高水位，HighWatermark 的缩写，取一个 partition 对应的 ISR 中最小的 LEO 作为 HW，consumer 最多只能消费到 HW 所在的位置。另外每个 replica 都有 HW，leader 和 follower 各自负责更新自己的 HW 的状态。对于 leader 新写入的消息，consumer 不能立刻消费，leader 会等待该消息被所有 ISR 中的 replicas 同步后更新 HW，此时消息才能被 consumer 消费。这样就保证了如果 leader 所在的 broker 失效，该消息仍然可以从新选举的 leader 中获取。对于来自内部 broker 的读取请求，没有 HW 的限制。

下图详细的说明了当 producer 生产消息至 broker 后，ISR 以及 HW 和 LEO 的流转过程：

由此可见，Kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。事实上，同步复制要求所有能工作的 follower 都复制完，这条消息才会被 commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，follower 异步的从 leader 复制数据，数据只要被 leader 写入



log 就被认为已经 commit，这种情况下如果 follower 都还没有复制完，落后于 leader 时，突然 leader 宕机，则会丢失数据。而 Kafka 的这种使用 ISR 的方式则很好的均衡了确保数据不丢失以及吞吐率。

Kafka 的 ISR 的管理最终都会反馈到 Zookeeper 节点上。具体位置为: /brokers/topics/[topic]/partitions/[partition]/state。目前有两个地方会对这个 Zookeeper 的节点进行维护:

- Controller 来维护: Kafka 集群中的其中一个 Broker 会被选举为 Controller，主要负责 Partition 管理和副本状态管理，也会执行类似于重分配 partition 之类的管理任务。在符合某些特定条件下，Controller 下的 LeaderSelector 会选举新的 leader，ISR 和新的 leader_epoch 及 controller_epoch 写入 Zookeeper 的相关节点中。同时发起 LeaderAndIsrRequest 通知所有的 replicas。
- leader 来维护: leader 有单独的线程定期检测 ISR 中 follower 是否脱离 ISR，如果发现 ISR 变化，则会将新的 ISR 的信息返回到 Zookeeper 的相关节点中。

3.4 数据可靠性和持久性保证

当 producer 向 leader 发送数据时，可以通过 request.required.acks 参数来设置数据可靠性的级别：

- 1（默认）：这意味着producer在ISR中的leader已成功收到的数据并得到确认后发送下一条message。如果leader宕机了，则会丢失数据。
- 0：这意味着producer无需等待来自broker的确认而继续发送下一批消息。这种情况下数据传输效率最高，但是数据可靠性确是最低的。
- -1：producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。但是这样也不能保证数据不丢失，比如当ISR中只有leader时（前面ISR那一节讲到，ISR中的成员由于某些情况会增加也会减少，最少就只剩一个leader），这样就变成了acks=1的情况。

如果要提高数据的可靠性，在设置 request.required.acks=-1 的同时，也要 min.insync.replicas 这个参数（可以在 broker 或者 topic 层面进行设置）的配合，这样才能发挥最大的功效。min.insync.replicas 这个参数设定 ISR 中的最小副本数是多少，默认值为 1，当且仅当 request.required.acks 参数设置为 -1 时，此参数才生效。如果 ISR 中的副本数少于 min.insync.replicas 配置的数量时，客户端会返回异常：org.apache.kafka.common.errors.NotEnoughReplicasException: Messages are rejected since there are fewer in-sync replicas than required。

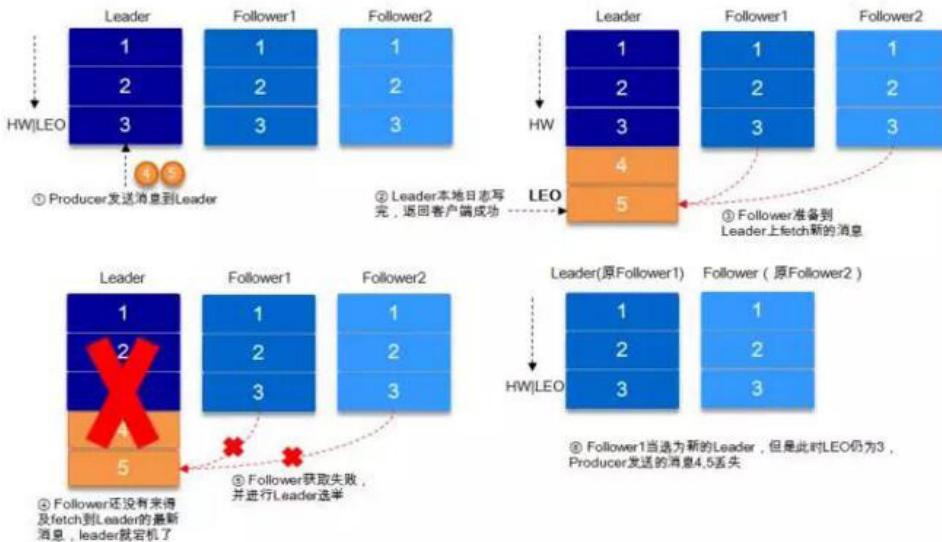
接下来对 acks=1 和 -1 的两种情况进行详细分析。

1. **request.required.acks=1**

producer 发送数据到 leader，leader 写本地日志成功，返回客户端成功；此时 ISR 中的副本还没有来得及拉取该消息，leader 就宕机了，那么此次发送的消息就会丢失。

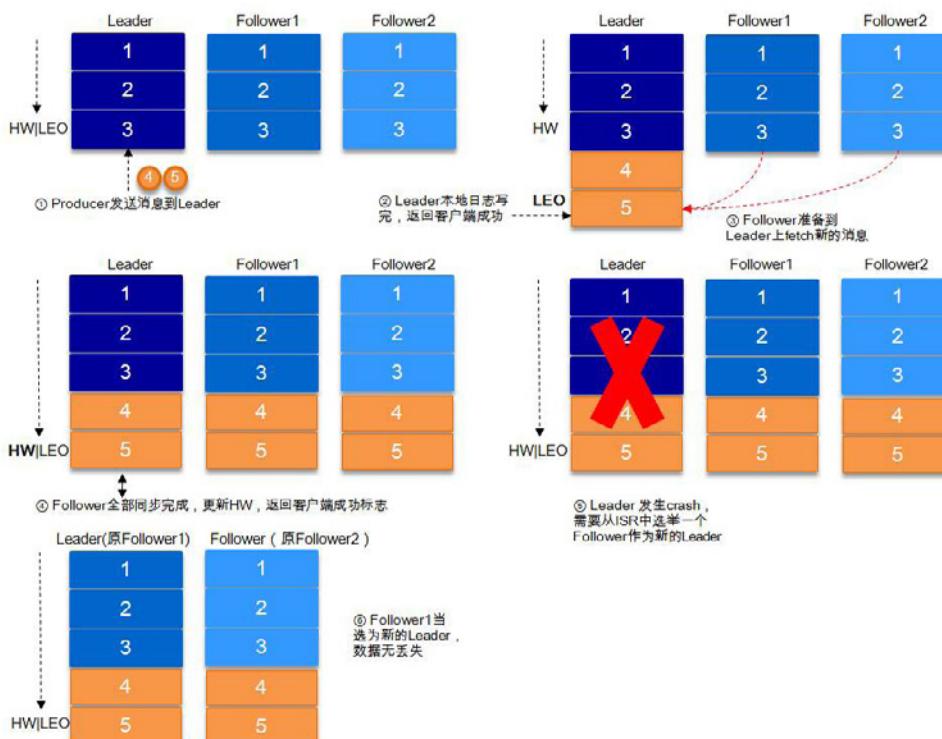
2. **request.required.acks=-1**

同步（Kafka 默认认为同步，即 producer.type=sync）的发送模式，replication.factor>=2 且 min.insync.replicas>=2 的情况下，不会丢

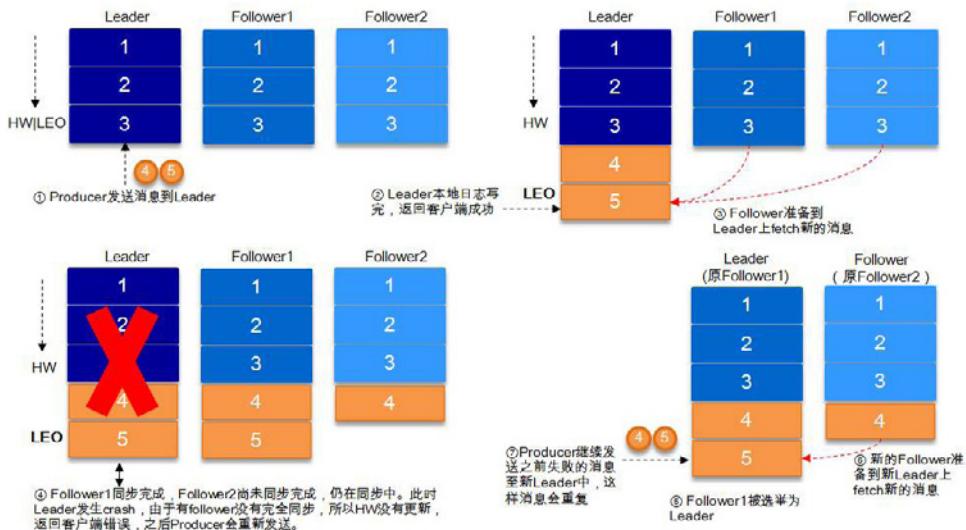


失数据。

有两种典型情况。acks=-1 的情况下（如无特殊说明，以下 acks 都表示为参数 request.required.acks），数据发送到 leader，ISR 的 follower 全部完成数据同步后，leader 此时挂掉，那么会选举出新的 leader，数据不会丢失。



acks=-1 的情况下，数据发送到 leader 后，部分 ISR 的副本同步，leader 此时挂掉。比如 follower1h 和 follower2 都有可能变成新的 leader，producer 端会得到返回异常，producer 端会重新发送数据，数据可能会重复。



当然上图中如果在 leader crash 的时候，follower2 还没有同步到任何数据，而且 follower2 被选举为新的 leader 的话，这样消息就不会重复。

注：Kafka只处理fail/recover问题，不处理Byzantine问题。

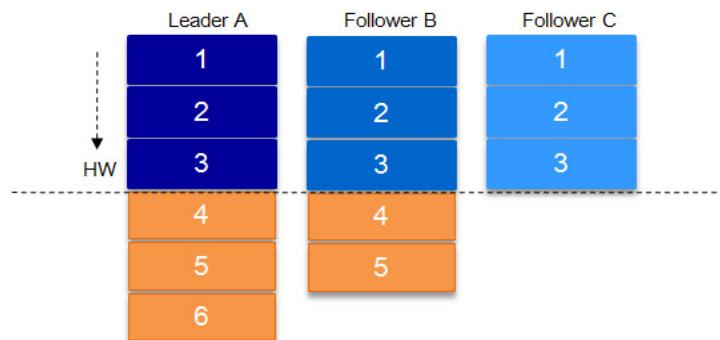
3.5 关于HW的进一步探讨

考虑上图（即 acks=-1，部分 ISR 副本同步）中的另一种情况，如果在 Leader 挂掉的时候，follower1 同步了消息 4, 5，follower2 同步了消息 4，与此同时 follower2 被选举为 leader，那么此时 follower1 中的多出的消息 5 该做如何处理呢？

这里就需要 HW 的协同配合了。如前所述，一个 partition 中的 ISR 列表中，leader 的 HW 是所有 ISR 列表里副本中最小的那个的 LEO。类似于木桶原理，水位取决于最低那块短板。

如下图，某个 topic 的某 partition 有三个副本，分别为 A、B、C。

A 作为 leader 肯定是 LEO 最高，B 紧随其后，C 机器由于配置比较低，网络比较差，故而同步最慢。这个时候 A 机器宕机，这时候如果 B 成为 leader，假如没有 HW，在 A 重新恢复之后会做同步 (makeFollower) 操作，在宕机时 log 文件之后直接做追加操作，而假如 B 的 LEO 已经达到了 A 的 LEO，会产生数据不一致的情况，所以使用 HW 来避免这种情况。A 在做同步操作的时候，先将 log 文件截断到之前自己的 HW 的位置，即 3，之后再从 B 中拉取消息进行同步。



如果失败的 follower 恢复过来，它首先将自己的 log 文件截断到上次 checkpointed 时刻的 HW 的位置，之后再从 leader 中同步消息。leader 挂掉会重新选举，新的 leader 会发送“指令”让其余的 follower 截断至自身的 HW 的位置然后再拉取新的消息。

当 ISR 中的个副本的 LEO 不一致时，如果此时 leader 挂掉，选举新的 leader 时并不是按照 LEO 的高低进行选举，而是按照 ISR 中的顺序选举。

3.6 Leader选举

一条消息只有被 ISR 中的所有 follower 都从 leader 复制过去才会被认为已提交。这样就避免了部分数据被写进了 leader，还没来得及被任何 follower 复制就宕机了，而造成数据丢失。而对于 producer 而言，它可以选择是否等待消息 commit，这可以通过 request.required.acks 来设置。这种机制确保了只要 ISR 中有一个或者以上的 follower，一条被 commit 的消息就不会丢失。

有一个很重要的问题是当 leader 宕机了，怎样在 follower 中选举出新的 leader，因为 follower 可能落后很多或者直接 crash 了，所以必须确保选择“最新”的 follower 作为新的 leader。一个基本的原则就是，如果 leader 不在了，新的 leader 必须拥有原来的 leader commit 的所有消息。这就需要做一个折中，如果 leader 在表名一个消息被 commit 前等待更多的 follower 确认，那么在它挂掉之后就有更多的 follower 可以成为新的 leader，但这也会造成吞吐率的下降。

一种非常常用的选举 leader 的方式是“少数服从多数”，Kafka 并不是采用这种方式。这种模式下，如果我们有 $2f+1$ 个副本，那么在 commit 之前必须保证有 $f+1$ 个 replica 复制完消息，同时为了保证能正确选举出新的 leader，失败的副本数不能超过 f 个。这种方式有个很大的优势，系统的延迟取决于最快的几台机器，也就是说比如副本数为 3，那么延迟就取决于最快的那个 follower 而不是最慢的那个。

“少数服从多数”的方式也有一些劣势，为了保证 leader 选举的正常进行，它所能容忍的失败的 follower 数比较少，如果要容忍 1 个 follower 挂掉，那么至少要 3 个以上的副本，如果要容忍 2 个 follower 挂掉，必须要有 5 个以上的副本。也就是说，在生产环境下为了保证较高的容错率，必须要有大量的副本，而大量的副本又会在大数据量下导致性能的急剧下降。这种算法更多用在 Zookeeper 这种共享集群配置的系统中而很少在需要大量数据的系统中使用的原因。HDFS 的 HA 功能也是基于“少数服从多数”的方式，但是其数据存储并不是采用这样的方式。

实际上，leader 选举的算法非常多，比如 Zookeeper 的 Zab、Raft 以及 Viewstamped Replication。而 Kafka 所使用的 leader 选举算法更像是微软的 PacificA 算法。

Kafka 在 Zookeeper 中为每一个 partition 动态的维护了一个 ISR，这个 ISR 里的所有 replica 都跟上了 leader，只有 ISR 里的成员才能有被选为 leader 的可能（unclean.leader.election.enable=false）。在这种模式下，对于 $f+1$ 个副本，一个 Kafka topic 能在保证不丢失已经

commit 消息的前提下容忍 f 个副本的失败，在大多数使用场景下，这种模式是十分有利的。事实上，为了容忍 f 个副本的失败，“少数服从多数”的方式和 ISR 在 commit 前需要等待的副本的数量是一样的，但是 ISR 需要的总的副本的个数几乎是“少数服从多数”的方式的一半。

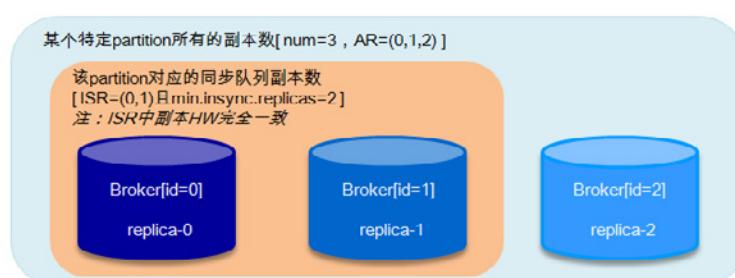
上文提到，在 ISR 中至少有一个 follower 时，Kafka 可以确保已经 commit 的数据不丢失，但如果某一个 partition 的所有 replica 都挂了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

等待 ISR 中任意一个 replica “活”过来，并且选它作为 leader

选择第一个“活”过来的 replica（并不一定是在 ISR 中）作为 leader。

这就需要在可用性和一致性当中作出一个简单的抉择。如果一定要等待 ISR 中的 replica “活”过来，那不可用的时间就可能会相对较长。而且如果 ISR 中所有的 replica 都无法“活”过来了，或者数据丢失了，这个 partition 将永远不可用。选择第一个“活”过来的 replica 作为 leader，而这个 replica 不是 ISR 中的 replica，那即使它并不保障已经包含了所有已 commit 的消息，它也会成为 leader 而作为 consumer 的数据源。默认情况下，Kafka 采用第二种策略，即 unclean.leader.election.enable=true，也可以将此参数设置为 false 来启用第一种策略。

unclean.leader.election.enable 这个参数对于 leader 的选举、系统的可用性以及数据的可靠性都有至关重要的影响。下面我们来分析下几种典型的场景。



如果上图所示，假设某个 partition 中的副本数为 3, replica-0, replica-1, replica-2 分别存放在 broker0, broker1 和 broker2 中。AR=(0, 1, 2), ISR=(0, 1)。设置 request.required.acks=-1, min.insync.replicas=2, unclean.leader.election.enable=false。这里讲 broker0 中的副本也称之为 broker0 起初 broker0 为 leader, broker1 为 follower。

- 当ISR中的replica-0出现crash的情况时, broker1选举为新的leader[ISR=(1)], 因为受min.insync.replicas=2影响, write不能服务, 但是read能继续正常服务。此种情况恢复方案:
 1. 尝试恢复(重启)replica-0, 如果能起来, 系统正常;
 2. 如果replica-0不能恢复, 需要将min.insync.replicas设置为1, 恢复write功能。
- 当ISR中的replica-0出现crash, 紧接着replica-1也出现了crash, 此时[ISR=(1), leader=-1], 不能对外提供服务, 此种情况恢复方案:
 1. 尝试恢复replica-0和replica-1, 如果都能起来, 则系统恢复正常;
 2. 如果replica-0起来, 而replica-1不能起来, 这时候仍然不能选出leader, 因为当设置unclean.leader.election.enable=false时, leader只能从ISR中选举, 当ISR中所有副本都失效之后, 需要ISR中最后失效的那个副本能恢复之后才能选举leader, 即replica-0先失效, replica-1后失效, 需要replica-1恢复后才能选举leader。保守的方案建议把unclean.leader.election.enable设置为true, 但是这样会有丢失数据的情况发生, 这样可以恢复read服务。同样需要将min.insync.replicas设置为1, 恢复write功能;
 3. replica-1恢复, replica-0不能恢复, 这个情况上面遇到过, read服务可用, 需要将min.insync.replicas设置为1, 恢复

write功能；

4. replica-0和replica-1都不能恢复，这种情况可以参考情形2。

- 当ISR中的replica-0, replica-1同时宕机, 此时[ISR=(0, 1)], 不能对外提供服务, 此种情况恢复方案: 尝试恢复replica-0和replica-1, 当其中任意一个副本恢复正常时, 对外可以提供read服务。直到2个副本恢复正常, write功能才能恢复, 或者将min.insync.replicas设置为1。

3.7 Kafka的发送模式

Kafka 的发送模式由 producer 端的配置参数 producer.type 来设置, 这个参数指定了在后台线程中消息的发送方式是同步的还是异步的, 默认是同步的方式, 即 producer.type=sync。如果设置成异步的模式, 即 producer.type=async, 可以是 producer 以 batch 的形式 push 数据, 这样会极大的提高 broker 的性能, 但是这样会增加丢失数据的风险。如果需要确保消息的可靠性, 必须要将 producer.type 设置为 sync。

对于异步模式, 还有 4 个配套的参数, 如下:

Property	Description
queue.buffering.max.ms	默认值: 5000。启用异步模式时, producer 缓存消息的时间。比如我们设置成1000时, 它会缓存1s的数据再一次发送出去, 这样可以极大的增加 broker 吞吐量, 但也会造成时效性的降低。
queue.buffering.max.messages	默认值: 10000。启用异步模式时, producer 缓存队列里最大缓存的消息数量, 如果超过这个值, producer 就会阻塞或者丢掉消息。
queue.enqueue.timeout.ms	默认值: -1。当达到上面参数时 producer 会阻塞等待的时间。如果设置为0, buffer 队列满时 producer 不会阻塞, 消息直接被丢掉; 若设置为-1, producer 会被阻塞, 不会丢消息。
batch.num.messages	默认值: 200。启用异步模式时, 一个 batch 缓存的消息数量。达到这个数值时, producer 才会发送消息。(每次批量发送的数量)
ConsumerGroup	每个 Consumer 属于一个特定的 Consumer Group, 一条消息可以发送到多个不同的 Consumer Group, 但是一个 Consumer Group 中只能有一个 Consumer 能够消费该消息
Partition	物理上的概念, 一个 topic 可以分为多个 partition, 每个 partition 内部是有序的

以batch的方式推送数据可以极大的提高处理效率, kafka producer可以将消息在内存中累计到一定数量后作为一个batch发送请求。batch的数量大小可以通过producer的参数(batch.num.messages)控制。通过增加batch的大小, 可以减少网络请求和磁盘IO的次数, 当然具体参数设置需要在效率和时效性方面做一个权衡。在比较新的版本中还有batch.size这个参数。

4 高可靠性使用分析

4.1 消息传输保障

前面已经介绍了 Kafka 如何进行有效的存储，以及了解了 producer 和 consumer 如何工作。接下来讨论的是 Kafka 如何确保消息在 producer 和 consumer 之间传输。有以下三种可能的传输保障（delivery guarantee）：

- At most once: 消息可能会丢，但绝不会重复传输
- At least once: 消息绝不会丢，但可能会重复传输
- Exactly once: 每条消息肯定会被传输一次且仅传输一次

Kafka 的消息传输保障机制非常直观。当 producer 向 broker 发送消息时，一旦这条消息被 commit，由于副本机制（replication）的存在，它就不会丢失。但是如果 producer 发送数据给 broker 后，遇到的网络问题而造成通信中断，那 producer 就无法判断该条消息是否已经提交（commit）。虽然 Kafka 无法确定网络故障期间发生了什么，但是 producer 可以 retry 多次，确保消息已经正确传输到 broker 中，所以目前 Kafka 实现的是 at least once。

consumer 从 broker 中读取消息后，可以选择 commit，该操作会在 Zookeeper 中存下该 consumer 在该 partition 下读取的消息的 offset。该 consumer 下一次再读该 partition 时会从下一条开始读取。如未 commit，下一次读取的开始位置会跟上一次 commit 之后的开始位置相同。当然也可以将 consumer 设置为 autocommit，即 consumer 一旦读取到数据立即自动 commit。如果只讨论这一读取消息的过程，那 Kafka 是确保了 exactly once，但是如果由于前面 producer 与 broker 之间的某种原因导致消息的重复，那么这里就是 at least once。

考虑这样一种情况，当 consumer 读完消息之后先 commit 再处理消息，在这种模式下，如果 consumer 在 commit 后还没来得及处理消息就 crash 了，下次重新开始工作后就无法读到刚刚已提交而未处理的消息，这就对

应于 at most once 了。

读完消息先处理再 commit。这种模式下，如果处理完了消息在 commit 之前 consumer crash 了，下次重新开始工作时还会处理刚刚未 commit 的消息，实际上该消息已经被处理过了，这就对应于 at least once。

要做到 exactly once 就需要引入消息去重机制。

4.2 消息去重

如上一节所述，Kafka 在 producer 端和 consumer 端都会出现消息的重复，这就需要去重处理。

Kafka 文档中提及 GUID(Globally Unique Identifier) 的概念，通过客户端生成算法得到每个消息的 unique id，同时可映射至 broker 上存储的地址，即通过 GUID 便可查询提取消息内容，也便于发送方的幂等性保证，需要在 broker 上提供此去重处理模块，目前版本尚不支持。

针对 GUID，如果从客户端的角度去重，那么需要引入集中式缓存，必然会增加依赖复杂度，另外缓存的大小难以界定。

不只是 Kafka，类似 RabbitMQ 以及 RocketMQ 这类商业级中间件也只保障 at least once，且也无法从自身去进行消息去重。所以我们建议业务方根据自身的业务特点进行去重，比如业务消息本身具备幂等性，或者借助 Redis 等其他产品进行去重处理。

4.3 高可靠性配置

Kafka 提供了很高的数据冗余弹性，对于需要数据高可靠性的场景，我们可以增加数据冗余备份数（replication.factor），调高最小写入副本数的个数（min.insync.replicas）等等，但是这样会影响性能。反之，性能提高而可靠性则降低，用户需要自身业务特性在彼此之间做一些权衡性选择。

要保证数据写入到 Kafka 是安全的，高可靠的，需要如下的配置：

- topic的配置：replication.factor>=3，即副本数至少是3个；

$2 \leq \text{min.insync.replicas} \leq \text{replication.factor}$

- broker的配置：leader的选举条件`unclean.leader.election.enable=false`
- producer的配置：`request.required.acks=-1(all)`, `producer.type=sync`

5 BenchMark

Kafka 在唯品会有着很深的历史渊源，根据唯品会消息中间件团队（VMS 团队）所掌握的资料显示，在 VMS 团队运转的 Kafka 集群中所支撑的 topic 数已接近 2000，每天的请求量也已达百亿级。这里就以 Kafka 的高可靠性为基准点来探究几种不同场景下的行为表现，以此来加深对 Kafka 的认知，为大家在以后高效的使用 Kafka 时提供一份依据。

5.1 测试环境

Kafka broker 用到了 4 台机器，分别为 broker[0/1/2/3] 配置如下：

- CPU: 24core/2.6GHZ
- Memory: 62G
- Network: 4000Mb
- OS/kernel: CentOs release 6.6 (Final)
- Disk: 1089G
- Kafka 版本: 0.10.1.0

broker 端 JVM 参数设置：

```
-Xmx8G -Xms8G -server -XX:+UseParNewGC -XX:+UseConcMarkSweepGC
-XX:+CMSClassUnloadingEnabled -XX:+CMSScavengeBeforeRemark
-XX:+DisableExplicitGC -Djava.awt.headless=true -Xloggc:/
apps/service/kafka/bin/..../logs/kafkaServer-gc.log
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps
-XX:+PrintGCTimeStamps -Dcom.sun.management.jmxremote -Dcom.sun.
sun.management.jmxremote.authenticate=false -Dcom.sun.
management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.
```

`port=9999`

客户端机器配置：

- CPU: 24core/2.6GHZ
- Memory: 3G
- Network: 1000Mb
- OS/kernel: CentOs release 6.3 (Final)
- Disk: 240G

5.2 不同场景测试

场景 1：测试不同的副本数、min.insync.replicas 策略以及 request.required.acks 策略（以下简称 acks 策略）对于发送速度（TPS）的影响

具体配置：一个 producer；发送方式为 sync；消息体大小为 1kB；partition 数为 12。副本数为：1/2/4；min. insync. replicas 分别为 1/2/4；acks 分别为 -1 (all) /1/0。

具体测试数据如下表 (min. insync. replicas 只在 acks=-1 时有效)：

acks	replicas	min.insync. replicas	retries	TPS
-1	1	1	0	28511.3
-1	2	1	0	22359.5
-1	2	2	0	22927.4
-1	4	1	0	16193.9
-1	4	2	0	16599.9
-1	4	4	0	16680.3
0	1	N/A	0	45353.8
0	2	N/A	0	46426.5
0	4	N/A	0	46764.2
1	1	N/A	0	33950.3
1	2	N/A	0	32192.2

1	4	N/A	0	32275.9
---	---	-----	---	---------

测试结果分析：

- 客户端的acks策略对发送的TPS有较大的影响，TPS: acks_0 > acks_1 > ack_-1；
- 副本数越高，TPS越低；副本数一致时，min. insync. replicas不影响TPS；
- acks=0/1时，TPS与min. insync. replicas参数以及副本数无关，仅受acks策略的影响。

下面将partition的个数设置为1，来进一步确认下不同的acks策略、不同的min. insync. replicas策略以及不同的副本数对于发送速度的影响，详细请看情景2和情景3。

场景2：在partition个数固定为1，测试不同的副本数和min.insync.replicas策略对发送速度的影响

具体配置：一个producer；发送方式为sync；消息体大小为1kB；producer端acks=-1(all)。变换副本数：2/3/4；min. insync. replicas设置为：1/2/4。

测试结果如下：

replicas	min.insync.replicas	TPS
2	1	9738.8
2	2	9701.6
3	1	8999.7
3	2	9243.1
4	1	9005.8
4	2	8216.9
4	4	9092.4

测试结果分析：副本数越高，TPS越低（这点与场景1的测试结论吻合），但是当partition数为1时差距甚微。min. insync. replicas不影响TPS。

响 TPS。

场景 3：在 partition 个数固定为 1，测试不同的 acks 策略和副本数对发送速度的影响

具体配置：一个 producer；发送方式为 sync；消息体大小为 1kB；
min.insync.replicas=1。topic 副本数为：1/2/4；acks：0/1/-1。

测试结果如下：

replicas	acks	TPS
1	0	76696
2	0	57503
4	0	59367
1	1	19489
2	1	20404
4	1	18365
1	-1	18641
2	-1	9739
4	-1	9006

测试结果分析（与情景 1 一致）：

- 副本数越多，TPS越低；
- 客户端的acks策略对发送的TPS有较大的影响，TPS: acks_0 > acks_1 > ack_-1。

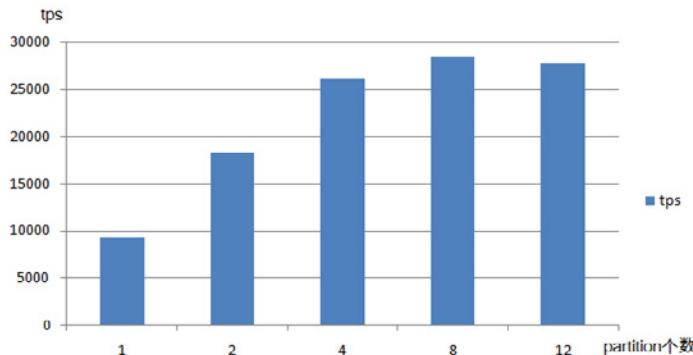
场景 4：测试不同 partition 数对发送速率的影响

具体配置：一个 producer；消息体大小为 1KB；发送方式为 sync；topic 副本数为 2；min.insync.replicas=2；acks=-1。partition 数量设置为 1/2/4/8/12。

测试结果见下图。

测试结果分析：partition 的不同会影响 TPS，随着 partition 的个数的增长 TPS 会有所增长，但并不是一直成正比关系，到达一定临界值时，

partition 数量的增加反而会使 TPS 略微降低。



场景 5：通过将集群中部分 broker 设置成不可服务状态，测试对客户端以及消息落盘的影响

具体配置：一个 producer；消息体大小 1KB；发送方式为 sync；topic 副本数为 4；min.insync.replicas 设置为 2；acks=-1；retries=0/100000000；partition 数为 12。

具体测试数据如下表：

acks	replicas	min. insync. replicas	retries	测试方法	TPS	数据落盘	出现错误
-1	4	2	0	发送过程中 kill 两台 broker	12840	一致（部分数据可落盘，部分失败）	错误1
-1	4	2	100000000	发送过程中 kill 两台 broker	13870	一致（消息有重复落盘）	错误2
-1	4	2	100000000	发送过程中 kill 三台 broker，之后重启	N/A	一致（消息有重复落盘）	错误2、3、4

出错信息：

- 错误1：客户端返回异常，部分数据可落盘，部分失败：org.apache.kafka.common.errors.NetworkException: The server disconnected before a response was received.
- 错误2：[WARN]internals.Sender - Got error produce response with correlation id 19369 on topic-partition default_channel_replicas_4_1-3, retrying (999999999

attempts left). Error: NETWORK_EXCEPTION

- 错误3: [WARN]internals.Sender - Got error produce response with correlation id 77890 on topic-partition default_channel_replicas_4_1-8, retrying (999999859 attempts left). Error: NOT_ENOUGH_REPLICAS
- 错误4: [WARN]internals.Sender - Got error produce response with correlation id 77705 on topic-partition default_channel_replicas_4_1-3, retrying (999999999 attempts left). Error: NOT_ENOUGH_REPLICAS_AFTER_APPEND

测试结果分析:

- kill两台broker后，客户端可以继续发送。broker减少后，partition的leader分布在剩余的两台broker上，造成了TPS的减小；
- kill三台broker后，客户端无法继续发送。Kafka的自动重试功能开始起作用，当大于等于min.insync.replicas数量的broker恢复后，可以继续发送；
- 当retries不为0时，消息有重复落盘；客户端成功返回的消息都成功落盘，异常时部分消息可以落盘。

场景 6：测试单个 producer 的发送延迟，以及端到端的延迟

具体配置：一个 producer；消息体大小 1KB；发送方式为 sync；topic 副本数为 4；min.insync.replicas 设置为 2；acks=-1；partition 数为 12。

测试数据及结果（单位为 ms）：

发送端 (avg)	发送端 (min)	发送端 (max)	发送端 (99%)	发送端 (99.99%)	消费端 (avg)	消费端 (min)	消费端 (max)	消费端 (99%)	消费端 (99.99%)
1.715	1	157	3	29	1.646	1	288	4	72

各场景测试总结：

- 当acks=-1时，Kafka发送端的TPS受限于topic的副本数量（ISR

中），副本越多TPS越低；

- acks=0时，TPS最高，其次为1，最差为-1，即TPS: $\text{acks_0} > \text{acks_1} > \text{ack_}-1$;
- min. insync. replicas参数不影响TPS；
- partition的不同会影响TPS，随着partition的个数的增长TPS会有所增长，但并不是一直成正比关系，到达一定临界值时，partition数量的增加反而会使TPS略微降低；
- Kafka在 $\text{acks}=-1, \text{min. insync. replicas} \geq 1$ 时，具有高可靠性，所有成功返回的消息都可以落盘。

作者介绍

唯品会消息中间件团队（VMS 团队），隶属于唯品会基础架构部，主要从事消息中间件（RabbitMQ，Kafka 等）相关的研究。

MySQL 到底能不能放到 Docker 里跑？

作者 王晓波



Talk is cheap, show me the demo。MySQL 到底能不能放到 Docker 里跑？同程旅游目前已经有超过一千个 MySQL 实例安全稳定地跑在 Docker 平台上。

前言

前几月经常看到有 MySQL 到底能不能放到 Docker 里跑的各种讨论。这样做是错的！这样做是对的！说错的理由也说了一大堆，说对的思想也很明确。大家都有道理。但是我本人觉得这样的讨论落地意义不大。因为对与错还是要实践来得出的。

所以同程旅游也很早开始了 MySQL 的 Docker 化实践，到目前已经

有超一千多个 MySQL 实例在 Docker 平台安全稳定地跑着，DB 运维能力发生了质的提高（DBA 再也不用担心删库跑路了）。

当然这样是不是可以证明之前的讨论结论——是对的。我想也不一定，因为我们还只是一只在学飞行的小鸟，还要更多的学习，所以我们特将我们在 MySQL 的 Docker 化上的实践分享给大家。

背景介绍

同程旅游早期的数据库都以 MSSQL 为主，这个产品有个特点就是 UI 操作很棒。但是批量和自动化管理很难做，人力的工作很多。后来逐渐替换为 MySQL 后也是按照传统的运维方式管理。导致大部分的工作需要人肉运维。

当然像我们早期使用过的 MSSQL 也是有优点的：就是单机性能比较好，在当年那个资源不够的年代里我们常可以在高可用的实例上运行多个库。这种情况下物理机数量与实例数量还是比较可控的，相对数量比较少，人肉运维完全可以应对。

但是 MSSQL 的缺陷也很多，比如做水平拆分比较困难，导致数据库成为系统中最大的一个瓶颈。但在我使用 MySQL+ 中间件（我们做这个中间件也是下了不少心思的，以后可以分享一下）做水平拆分后就开始解决了这个瓶颈。

水平拆分的引入也带来了一个小缺点，就是会造成数据库实例数量大幅上升。举个例子我们做 1024 分片的话一般是做 32 个 node，一主一从是必须的（大部分情况是一主两从），那么至少 64 个实例，再加上应急扩展和备份用的节点那就更多了（中间件的开发者更希望是 1024 片就是 1024 个实例）。

一次上线做一个 32node 分片扩展从库，两个 DBA 足足花了 4 个小时。另外，如果做单机单实例那肯定更不行了，别的不说，成本也会是个大问题，且物理机的资源也未能最大化利用。况且因为 MySQL 单体的性能没优势所以分片居多所以大部分情况下并不是每个库都能跑满整个物理

机的。即使有部分能跑满整机资源的库，它的多节点备份，环境一至性和运维动作统一等问题也会让 DBA 一头糟，忙碌又容易出错的工作其实是无意义的。

有了单机多实例运行 MySQL 实例的需求。单机多实例要思考的主要问题就是如果进行资源隔离和限制，实现方案有很多，怎么选？KVM，Docker，Cgroups 是目前的可以实现隔离主流方案。

KVM 对一个 DB 的隔离来说太重了，性能影响太大，在生产环境用不合适。这是因为 MySQL 运行的就是个进程而且对 IO 要求比较高，所以 KVM 不满足要求（虽然优化以后 IO 能有点提升）。

cgroups 比较轻，虽然隔离性不是很高，但对于我们的 MySQL 多实例隔离来说是完全够用了（Docker 的资源限制用的就是 cgroups）。但是我们还想针对每个 MySQL 实例运行额外的管理进程（比如监控等等）。用 cgroups 实现起来会比较复杂，并且我们还想让实例管理和物理机区分开，那 cgroups 也放弃。

至于 Docker，那就很不错了，那些裸用 cgroups 的麻烦它都给搞定了。并且有 API 可以提供支持，开发成本低。而且我们可以基于 Docker 镜像来做部署自动化，那么环境的一致性也可轻松解决。所以最终我们选择了 Docker 作为云平台的资源隔离方案（当然过程中也做了很多性能、稳定性等的适配工作，这里就不赘述了）。

下面两个图可以形象展示这款产品带来的革命性意义：



当然要能称之为云，那么平台最基本的要求就是具备资源计算、资源调度功能，且资源分配无需人工参与。对用户来讲，拿到的应该是直接可用的资源，并且天生自带高可用、自动备份、监控告警、慢日志分析等功能，无需用户关心资源背后的事情。其次才是各种日常的 DBA 运维操作需求服务化输出。下面我们就来讲讲我们这个平台是如何一步步实现的。

平台实现过程

站在巨人的肩膀上

我一直认为评价一款数据库的优劣，不能只评价数据库本身。我们要综合它的周边生态是否健全，比如：高可用方案、备份方案、日常维护难度、人才储备等等。当然对于一个云平台也一样，所以我们进行了短平快的试错工作，将平台分为多期版本开发。第一个版本的开发周期比较短，主要用来试验，所以我们要尽可能运用已有的开源产品来实现我们的需求，或者对已有开源产品进行二次开发以后实现定制化的需求。以下是我们当时用到的部分开源产品和技术。



下面选几个产品简单说一下我们通过它实现什么功能：

- Percona：我们的备份、慢日志分析、过载保护等功能都是基于 pt-tools 工具包来实现的。
- Prometheus：性能优越且功能强大的 TSDB，用于实现整个平台实例的监控告警。缺点是没有集群功能，单机性能是个瓶颈（虽然单机的处理能力已经很强了），所以我们在业务层面进行了 DB 拆分，实现了分布式存储及扩展。
- Consul：分布式的服务发现和配置共享软件，配合 prometheus 实现监控节点注册。
- Python：管理 Docker 容器中 MySQL 实例的 agent 以及部分操作脚本。
- Docker：承载 MySQL 实例并实现资源隔离和资源限制。

总体架构



容器调度系统如何选择

容器调度的开源产品主要有 Kubernetes 和 mesos，但是我们并没有选用这两个。主要原因是我们内部已经开发了一套基于 Docker 的资源管

理、调度的系统，至今稳定运行 2 年多了。这套架构稍作修改是符合需求的。

另外第三方的资源调度系统兼容我们目前的高可用架构，其他自动化管理有些难度，同时资源分配策略也需要定制化。所以最终还是选择采用了自研的资源调度管理。适合自己现状的需求才是最好的。当然后面有机会做到计算调度和存储调度分离的情况下我们可能会转向 Kubernetes 的方案。

工作原理

我们就拿创建集群来举例吧。当平台发起一个创建集群的任务后，首先会根据集群规模（一主一从还是一主多从，或者是分片集群）确定要创建的实例数量，然后根据这个需求按照我们的资源筛选规则（比如主从不能在同一台机器、内存配置不允许超卖等等），从现有的资源池中匹配出可用资源，然后依次创建主从关系、创建高可用管理、检查集群复制状态、推送集群信息到中间件（选用了中间件的情况下）控制中心、最后将以上相关信息都同步到 CMDB。

以上的每一个工作都是通过服务端发送消息到 agent，然后由 agent 执行对应的脚本，脚本会返回指定格式的执行结果，这些脚本是由 DBA 开发的。这种方式的优势在于，DBA 比任何人都了解数据库，所以通过这种方式可以有效提升项目开发效率，也能让 DBA 参与到项目当中去。开发只需要写前台逻辑，DBA 负责后端具体执行的指令。如果未来功能有变更或迭代的话，只需要迭代脚本即可，维护量极小。

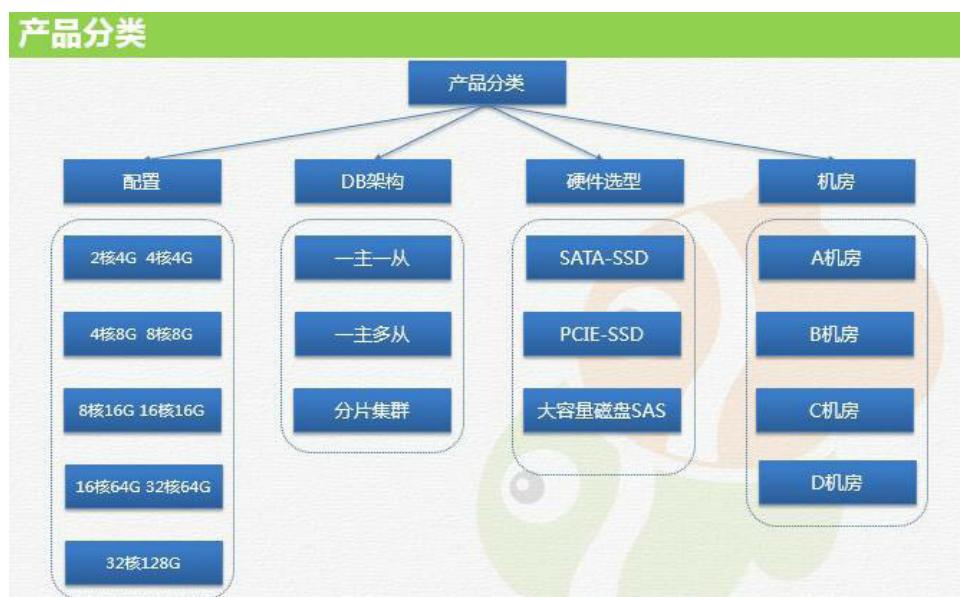
资源的调度分配原则

经过对同程多年的 DB 运维数据分析得到如下经验：

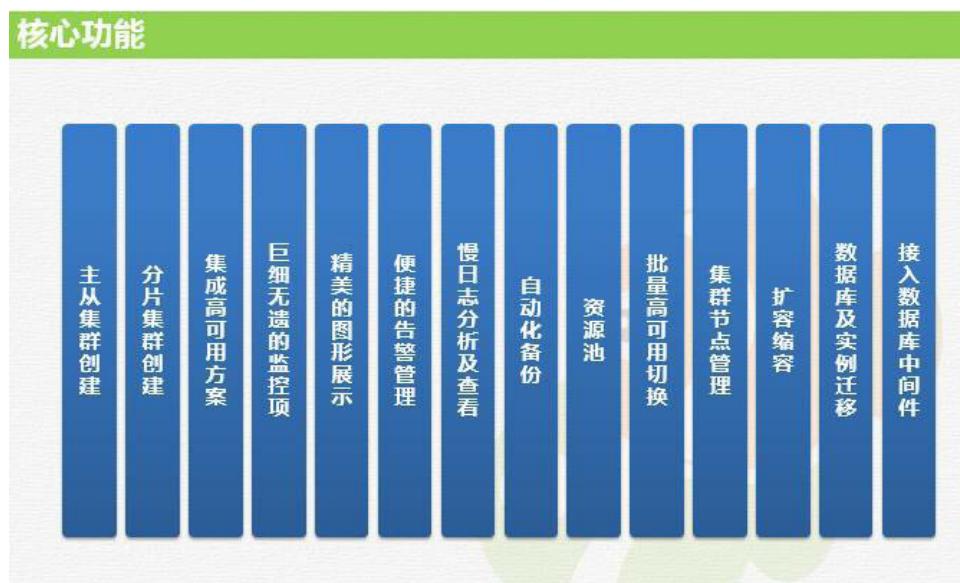
- CPU 最大超卖 3 倍，内存不超卖；
- 同一机房优先选择资源最空闲的机器；
- 主从角色不允许在同一台机器上；
- 若有 VIP 需求的主从端口需要一致，无 VIP 需求直接对接中间件的无端口一致的限制；

- 分片的集群将节点分布在多台物理机上。

产品分类



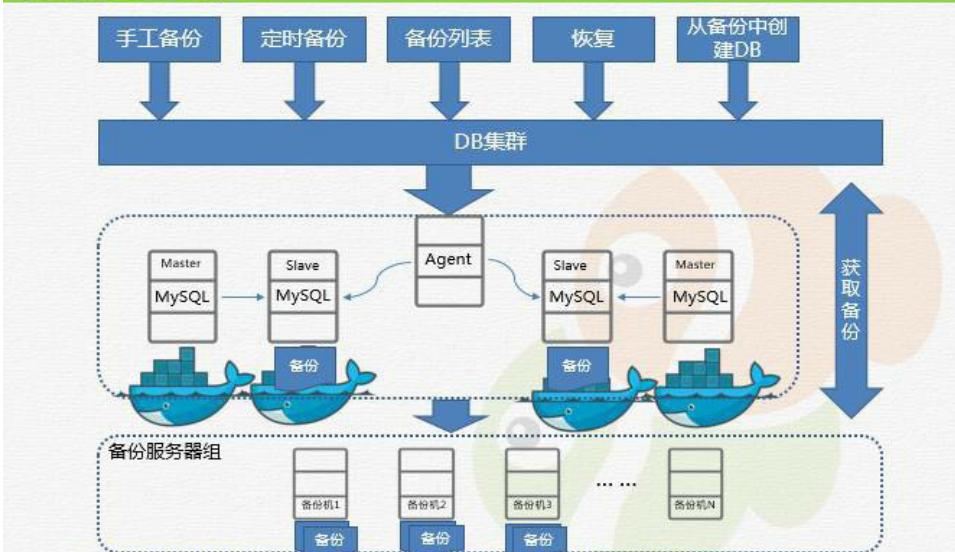
核心功能



以上是已经上线的部分核心功能，还有很多功能就不再一一展示。

备份恢复系统

备份恢复系统设计



备份工具我们是用 percona-xtrabackup。通过流备份的方式将数据备份到远端的备份服务器。备份服务器有多台，分别按照所属机房划分。

我们提供了手工备份和定时备份来满足不同场景的需求。多实例备份一定要关注磁盘 I/O 和网络，所以我们的备份策略会限制单个物理机上并行备份的数量，另外单个机房备份任务队列的并行度也有控制，确保并行备份任务始终保持到我们指定的数量。

假如整个机房并行的是 50 个任务，那么这 50 个当中如果有 5 个提前备份完成，那么会新加入 5 个等待备份的任务进入这个备份队列。我们后来改造了备份的存储方式，直接将备份流入分式存储。

监控告警系统

在上线这套云平台前，我们还是用传统的 zabbix 来实现监控告警的。zabbix 的功能的确非常强大，但是后端的数据库是个瓶颈，当然可以通过数据库拆分的方式解决。

数据库要监控的指标比较多，如果采集的项目比较多，zabbix 就需要加 proxy，架构越来越复杂，再加上和我们平台对接的成本比较高，对一些复杂的统计类查询（95 值、预测值等）性能比较差。

所以我们选了一款 TSDB——prometheus，这是一款性能极强、极其适合监控系统使用的时序性数据库。prometheus 优点就是单机性能超强。但凡事又有两面性，它的缺点就是不支持集群架构（不过我们解决了扩展的问题，下面会讲到）。

prometheus 的使用应该是从一年前就开始的，那时候我们只是把它作为辅助的监控系统来使用的，随着逐渐熟悉，越来越觉得这个是容器监控的绝佳解决方案。所以在上云平台的时候就选择了它作为整个平台的监控系统。

监控数据采集

prometheus 是支持 pushgateway 和 pull 的方式。我们选用了 pull 的方式。因为结构简单，开发成本低的同时还能和我们的系统完美对接。consul 集群负责注册实例信息和服务信息，比如 MySQL 实例主从对应的服务、Linux 主从对应的服务、容器注册对应的服务。然后 prometheus 通过 consul 上注册的信息来获取监控目标，然后去 pull 监控数据。监控客户端是以 agent 的形式存在，prometheus 通过 HTTP 协议获取 agent 端采集到的数据。

监控指标画图



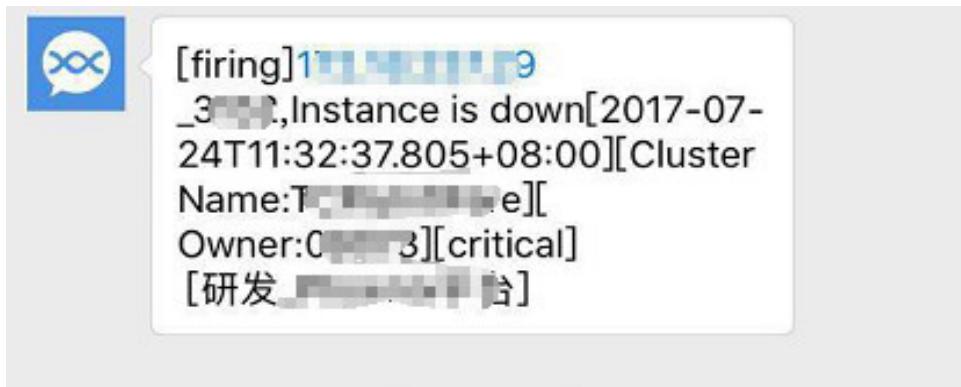
不得不说 grafana 是监控画图界的扛把子，功能齐全的度量仪表盘和图形编辑器，经过简单配置就能完成各种监控图形的展示。然后我们打通了云平台和 grafana 的关联，用户在云平台需要查看实例或集群信息，

只要点击按钮即可。

告警管理

告警管理分为：告警发送、告警接收人管理、告警静默等功能。

prometheus 有一个告警发送模块 alertmanager，我们通过 webhook 的方式让 alertmanager 把告警信息发送到云平台的告警 API，然后在云平台来根据后面的逻辑进行告警内容发送。



alertmanager 推过来的只是实例纬度的告警，所以我们结合告警平台的实例相关信息，会拼出一个多维信息的告警内容。让 DBA 一看就知道是谁的哪个集群在什么时间触发了什么等级的什么告警。告警恢复后也会再发一次恢复的通知。

alertmanager 也是功能强大的工具，支持告警抑制、告警路由策略、发送周期、静默告警等等。有需要可以自行配置。但是这种和平台分离的管理方式不是我们想要的，所以就想把 alertmanager 对告警信息处理的这部分功能集成到云平台内。

但是官方文档并没有提及到 alertmanager 的 API，通过对源码的分析，我们找到了告警管理相关的 API。然后 alertmanager 的原生 UI 上操作的功能完美移植到了我们的云平台，同时新增了实例相关集群名称、负责人等更多纬度的信息。

下面是一些操作样例：

- 当前告警。

- 添加告警静默。
- 已创建的静默规则。

状态: 当前告警 ▾

	告警项	集群
	Service_Down silenced	
	Service_Down silenced	

静默告警

开始时间 结束时间

告警项匹配规则

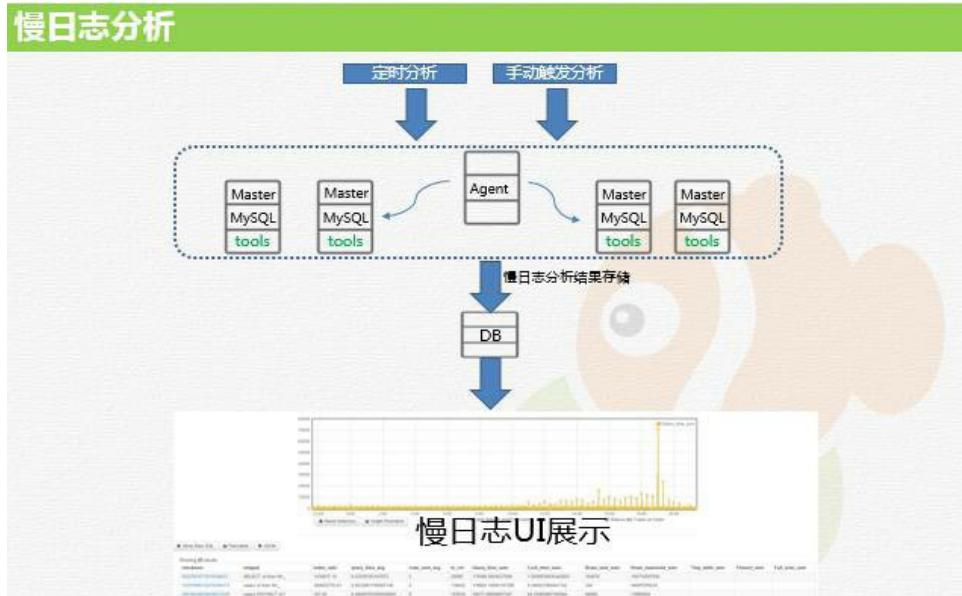
<input type="button" value="+"/>	severity	critical	<input type="checkbox"/> 匹配正则	<input type="button" value="-"/>
	instance	1[0-9][0-9][0-9]	<input type="checkbox"/> 匹配正则	<input type="button" value="-"/>
	alertname	Service_Down	<input type="checkbox"/> 匹配正则	<input type="button" value="-"/>
	job	mysql	<input type="checkbox"/> 匹配正则	<input type="button" value="-"/>

备注

设置内容	时间区间	数
alertname=Service_Down job=mysql	2017-06-27 10:40-2018-06-30 09:40	1
severity=critical instance=[0-9][0-9][0-9] alertname=Service_Down job=mysql	2017-06-28 15:54-2017-06-28 16:54	1
alertname=Service_Down	2017-06-27 12:12-2017-06-28 13:10	54
severity=critical	2017-05-23 16:29-2017-05-31 17:29	1

慢日志分析系统

慢日志的收集是通过 pt-query-digest 每小时进行本地分析，分析完成以后将结果写入慢日志存储的数据库来完成的。当然如果用户需要立



刻查看当前慢日志的情况，也可以在界面点击慢日志分析。分析完成后可以在 UI 界面点击慢日志查看，就能看到该实例的慢日志分析结果。它同时集成了 explain、查看 table status 等功能。

集群管理

集群管理作为该平台的核心功能之一，占据了整个平台 70% 的工作。这些功能就是 DBA 运维中经常需要用到的。我们的设计思路是以集群为

集群管理

此页面包含的对集群的所有管理操作：

- HA切换
- Slave增加删除
- 创建新集群
- 查看监控
- 慢日志分析
- 慢日志查看
- 手动备份
- 集群名称、生产线编辑

节点ID	VIP	主机	类型	角色	数据	日志	状态	创建人	版本
1	192.168.1.10	192.168.1.10	2C4G	★ Master	875.07M	52.36K	Running		

单位，所以同时只能操作一个集群上的实例。这样就不会在一个页面上显示过多无用的信息，看着乱还有可能导致误操作。看了下图中的这些功能就能更明白为什么要这么设计了。

图中只是一部分，还有部分未展示出的功能（集成中间件、Dashboard、黑屏诊断窗口等），在后版中功能更多。

高可用

高可用方案我们使用了目前最流行的 MySQL 高可用方案 MHA。MHA 的优缺点就不在这里讲了，有 DBA 同学的应该都已经很熟悉了。这里我说一下我们基于同程业务做的调整。

GTID

因为我们主要使用的 MariaDB，但是 MHA 最新版本也是不能支持 MariaDB 的 GTID 切换。所以我们在原有的基础上做了改进，支持了 MariaDB 的 GTID。使用 GTID 以后灵活切换是一个方面，另外一个是 sync_master_info 和 sync_relay_log_info 就不需要设置成 1 了（MariaDB 不支持写 table，只能写 file），极大减少了从库复制带来的 IOPS。

切换时调整相关参数

我们在切换时调整 sync_binlog 和 innodb_flush_log_at_trx_commit 参数，这两个参数是决定数据落盘方式的，默认大家都是设置双 1。这样相对数据最安全，但是 IO 也最高。

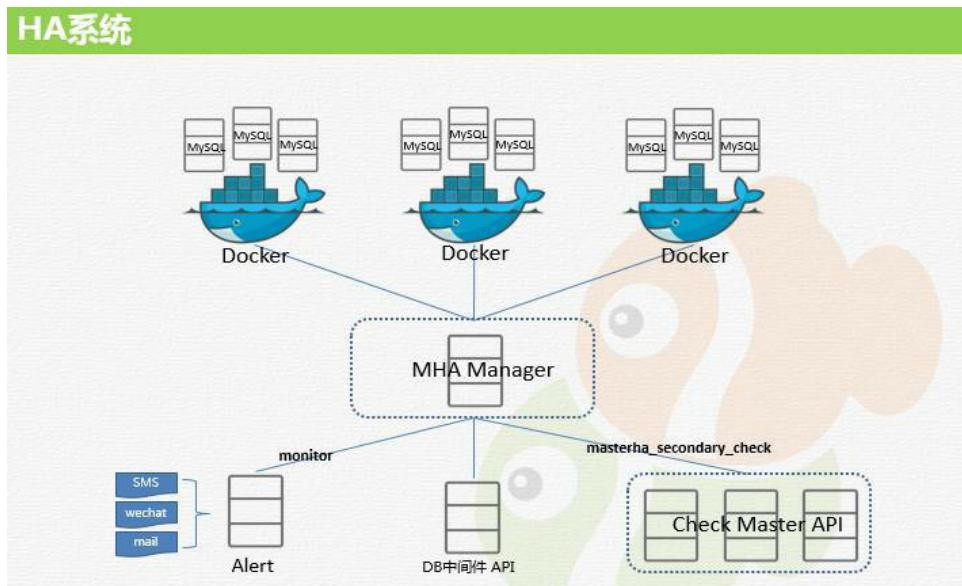
云服务的多实例部署会导致一台物理机上既有 master 又有 slave。我们肯定不希望 slave 产生太高的 IO 影响到同机器的其他 slave（虽然可以 IO 隔离，但是优先降低不必要 IO 才靠谱）。所以理论上来说 Master 上面设置双 1，slave 则可以不这样设置。但是切换后原来的 slave 可能会变成了 master。所以我们默认 slave 非双 1，在 MHA 切换的时候会自动将新 master 的这两个参数设置为 1。

哨兵

我们在多个点部署了哨兵服务。这个哨兵是一个简单的 API 服务，

带上响应的参数可以请求到指定的实例。当 MHA manager 检测到有 Master 无法连接时，会触发 secondary check 机制，带着 master 相关信息请求哨兵节点的 API，根据哨兵节点返回情况，若超过半数无法连接则切换。否则放弃切换。

高可用切换对接 DB 中间件



DB 中间件和 DB 通过物理 IP 连接，当发生高可用切换时将最新的 Master IP、Master port 信息推送到 DB 中间件控制中心，DB 中间件拿到配置后立刻下发并生效。

实例、库迁移

迁移功能初衷是为了将平台外的实例或者库迁移到平台里面来，后来随着逐渐使用发现这个功能可挖掘的空间很大，比如可以做平台内库表拆分等需求。实现原理也很简单，用 mydumper 将指定数据备份下来以后，再用 myloader 恢复到指定数据库。

这是一个全量的过程，增量复制用的是用我们自己开发的一个支持并行复制的工具，这个工具还支持等幂处理，使用更灵活。没有用原生复制的原因是，假如要将源实例多个库中的一个库迁移到目标实例，那么原生

实例迁移



复制就需要对 binlog 做复制过滤，这里面涉及到配置修改，实例重启，所以果断不考虑。

实现过程并没有高大上，但是完全满足需求。当然 mydumper 和 myloader 也有一些问题，我们也做了小改动以后才实现的。后面我们计划用流的方式去做数据导出导入（类似于阿里开源的 datax）。

迁移完成，增量无延迟的情况下，大家会关心迁移前后数据一致性的问题，我们提供了自研的数据校验工具。实测 300G 的数据校验时间约为 2 至 3 分钟，快慢取决于开多少线程。

屏蔽底层物理资源

对用户来讲，平台提供的是一个或一组数据库服务，不需要关系后端的实例是在哪台机器上。资源计算和调度全部由系统的算法进行管理。

提升资源利用率 (CPU、内存)

通过单机多实例，CPU 资源可超卖，有效提高 CPU 资源的利用。内存资源未超卖，但是可以控制到每个实例的内存使用，确保每个实例都能有足够的内存。若有剩余内存，则继续分配容器即可，不 OOM 的情况下压榨内存资源。

提升运维效率

效率的提升得益于标准化以后带来的自动化。批量运维的成本很低。以前部署一套分片集群需要花费将近 6 个小时（不包含对接中间件的 1 到 2 个小时），而现在只需要 5 分钟即可部署完成。并且部署完成以后会将提供一套中间件 +DB 分片集群的服务。

精细化管理

平台上线后有效提高了资源利用率，同时我们按照 1 库 1 实例的方式，可以有效避免不同库的压力不均导致相互影响的问题。并且性能监控也能精准到库级别。

结语

以上这些只是一个开始，后面还有很多功能需要完善，下面是近期策划的一些功能，其中有些已经在后版中开发完成。随着功能的不断迭代，我们会打造一个更加完美的私有云平台。

数据库私有云平台的上线对同程 DB 来说，意味着一个时代的结束，也意味着一个时代的开始。结束的是传统运维低效、高成本的运维时代，开始的是一个低成本、高效率、高保障的运维时代。我们相信未来会更美好！

作者介绍

王晓波，同程旅游（LY.com）首席架构师，EGO 会员。专注于高并发互联网架构设计、分布式电子商务交易平台设计、大数据分析平台设计、高可用性系统设计。曾设计过多个并发百万以上、每分钟 20 万以上订单量的电商交易平台，熟悉 B2C、B2B、B2B2C、O2O 等多种电商形态系统的技术设计，熟悉电子商务平台技术发展特点，拥有十多年丰富的技术架构、技术咨询经验，深刻理解电商系统对技术选择的重要性。

虚拟座谈会：聊聊 AIOps 的终极价值

作者 郭蕾



简单来说，AIOps 就是希望通过人工智能的方式，进一步提升运维效率，包括运维决策、故障预测和问题分析等。在 InfoQ 最近的一些文章中，都有不同程度地聊到 AIOps 相关的话题，比如美丽联合集团运维经理赵成认为 AIOps 必定是运维的发展趋势，宜信技术研发中心高级架构师张真系统分析了他们的实践案例，AliExpress 的周志伟也表示他们正在落地智能驱动的 SRE 理念。

从历史发展的角度来看，这些年，运维平台大致经历了流程化 → 工具化 → Web 化 → 自动化的演进历程。随着运维管理复杂度的提升，以及企业自动化运维体系的成熟，运维平台必定会向智能化靠拢。而从结果来看，智能化才是运维平台的最终目标。正如 InfoQ 的另外一篇文章所言，

在这个数字化转型的年代，任何使用传统技术来管理机器数据的企业要么是忽略了信息的价值，要么已经让他们的运维团队不堪重负。随着数据的暴涨，运维团队应该快速拥抱 AIOps。传统 AI 仍然会在某些领域发挥它的作用，而 AIOps 将为企业带来最直接最深远的价值。

讨论组成员

曲显平：百度运维部技术经理

万金：ThoughtWorks 咨询师

涂彦：腾讯游戏运维总监

InfoQ：如何理解 AIOps？AIOps 里面会涉及到哪些技术？这又是一个新名词吗？

曲显平：AIOps，Gartner 有报告解释为 Algorithmic IT operations platforms，当然国内大多理解为 AI + Ops，智能化运维。涉及的技术概括来讲就是 ABC，AI + BigData + Cloud，当然 AI 的部分，不仅包括时下流行的机器学习等技术，其实也要包含传统的数据挖掘等方法。

百度从 2014 年初提出智能化运维的思路，这个名词在我们这里不算新了。

万金：AIOps 是一次跨界创新，它结合了运维场景和运维数据，使用人工智能方式试图将运维决策自动化。

AIOps 技术会涉及到数据收集方面的基础监控，服务监控和业务监控，甚至会涉及到与持续交付流水线的数据和状态整合（比如在软件发布的阶段会自动关闭某些监控项。异常判断时会参考流水线目前的状态）。数据存储与人工智能技术，其中人工智能包括机器学习算法与深度学习模型（用于模式识别）。

AIOps 可以算一个新名词，他会是比自动化运维更高级的阶段，即为了保证确定的运维目标（SLA）使用人工智能自动决策阶段。（自动运维是将重复出现的运维动作自动化，而需要人来判断什么情况（或是条件出发）执行那种自动化过程。）

涂彦：不论是 AIOps 还是 0psAI，都是智能运维。简单来说，就是把成熟的人工智能技术应用于互联网及互联网+的运维工作场景中。AIOps 可以认为是运维岗位一个分支，本质还是运维。AIOps 本身也是由多个子岗组成的，比如智能场景规划、数据清洗、机器学习开发等。如果从运维的发展历史来看，确实是一个新名词。也代表了运维是个在未来有很大发展潜力与核心竞争力的工程师岗位。

InfoQ：你认为 AIOps 是运维发展的必然趋势吗？从手工运维，到自动化运维，再到底现在的 AIOps，谈谈你理解的运维发展趋势？

曲显平：很多人认为，运维是手工→自动化→智能化，但其实百度不是这样认为的，自动化运维可以说是终极状态，不管是 Web 化、平台化、还是智能化，用写死的 if/else 还是机器学习模型，都是实现的方式而已。其实运维人，要做的，就是一切皆自动，其实 AIOps 也就是讲的如何把人的决策也自动化起来。

为什么说他是必然趋势呢？主要还是因为传统方法仍然不能解决运维很多问题。比如在百度，一线运维最主要有 3 类场景：变更管理、故障管理、服务咨询。

- 变更管理，相比之下自动化程度最高，但仍然不能做到完全无人值守，最主要的问题是，如何检查一次变更是否符合预期、变更过程中遇到问题如何处置等。
- 故障管理，这个问题的复杂程度就比变更管理高很多了，从应该部署什么样的监控、如何设定报警、收到报警后如何判断并做出止损操作、止损后如何做根因分析、case study、如何彻底解决这一系列问题等等，这些还都是原有技术无法解决的。
- 服务咨询，这个领域向来都是自动化程度最低的，也一直是亟待解决的。

其实从这些描述来讲，在传统意义的自动化运维过程中，大家更多地是完成了两件事：一个是平台化，不再需要上到服务器去操作了；一个是在大规模并行，一个任务可以同时在成千上万个实例生效。但一旦问题变得

复杂，不是 if/else 能够解决，而需要人工决策的时候，传统自动化就不灵光了。

这种需要人工决策才能解决的问题，是制约运维走向终极自动化的主要障碍，从现今的技术发展来看，AIOps 显然是最合理的路。

万金：AIOps 是运维的必然趋势，随着数字化转型的推进，业务快速扩张规模不断扩大，监控规模与数量（系统，服务和业务监控项）越来越多，云计算又导致更进一步的集中化。这些因素使得运维工作复杂度超越人力所能管理的程度，必须通过自动决策方式辅助人来进行运维工作。而对于业务而言希望不通过增加运维团队规模的方式支持迅速扩张的业务。以上两点导致智能运维的必然性。

AIOps 并不是一开始就出现的，运维的技术是随着业务的规模与复杂的提升而不断演进的，为业务选择合适的运维方式是首要考虑的问题。相比较而言，智能化运维需要投入很大的启动成本，但是能达到前所未有的运维效率；而反观手动运维启动成本很低，适合在业务规模很小的初期使用，不过云原生应用和云计算平台降低了 AIOps 的启动成本，可以帮助快速扩张的业务解决手动或自动运维无法解决的问题。

总结运维工作的特点，业务在初期都是需要手动梳理运维业务的问题，从中发现系统性问题，从而为自动化运维提供知识储备；到了自动化运维阶段会不断的标准化运维管理对象，并积累运维数据为智能运维做准备；当业务处于爆发期通过 AIOps 方式自动化保证 SLA 的执行就成了顺理成章的选择了。这里所说的无法通过自动化批量执行的任务，往往需要根据实际情况（不能通过阈值判断的情况）进行判断后采取行动。这样的任务正是 AIOps 的入手点。

比如海量阈值的自动设置来降低监控系统的错报和漏报情况，或是在海量的分布式告警信息中找到故障的根因。

涂彦：AIOps 是互联网大势所趋，也是运维发展的必然趋势，更是现阶段运维发展的终极目标，相信在未来 5 年内将更为广泛应用在不同行业的运维工作中。自动化运维是一个承上启下的转折点。从手工到自动化，

需要跨越标准化。而从自动化到智能化，需要跨越数据化。标准化和数据化是智能运维建设要思考的底层问题。从组织能力到技术能力，缺一不可。

InfoQ：AIOps 的出现是为了解决哪些问题？这些问题自动化运维没办法解决吗？

曲显平：这个问题在上一问解释过了，AIOps 的出现是为了让运维早日达成完全的自动化运维，也就是解决人工决策 → 自动化决策的问题。

万金：任何运维问题本质上都可以通过手动或自动化方式完成，但是如果考虑到 MTTR（平均故障修复时间）对业务的影响和对运维团队人数的限制，就必须引入 AIOps。

首先，运维每天最多的工作就是监控系统是否正常运行，如果出问题了就需要及时解决，比如 99.99% 的可用性就意味着每年只能出现 52.56 分钟的系统不正常时间，一般人工处理一个故障在熟练的情况下也会在 20 分钟也就是说在频繁上线更新的情况下只能期望一年系统不要出 2 次以上的问题。如果想提高系统可靠性就必须引入 AIOps。

其次，AIOps 的出现大大缩短了 MTTR，在解决如何发现系统异常（自动化监控项管理）和如何找到问题根因（告警抑制：通过关联性判断，把计算资源、软件和网络具有相关性的告警信息聚合，让运维人员迅速找出问题根因的技术）。

最后，人的管理能力是有限的，就像人可以驾驶汽车，但飞机就需要自动导航只在启动和降落需要人工干预，对于宇宙飞船，宇航员只有在出现故障的时候才会手工干预。AIOps 就像驾驶宇宙飞船的计算机，只有 AIOps 的自动决策结果与预先设定好的目标(SLA)不相符才需要人工干预。

涂彦：质量、效率、成本、安全，是运维工作核心四要素。用 AIOps 来解决，会在提升效率的同时，将质量与成本更加精细化，使安全的应变能力更强。自动化在决策方面很难快速适应千变万化的生产环境和业务需求，运维经验仍然是主导地位，依靠个人能力的痕迹明显，对企业管理的风险都受控于个人。

InfoQ：落地 AIOps 的前提条件是什么？什么样的团队适合落地

AIOps ?

曲显平：从百度运维的 AIOps 经验来看，落地 AIOps 的前提条件是已经具备了比较完善的运维平台和有较充分的运维数据，也就是 ABC (AI、BigData、Cloud) 里的 A，离不开 B 和 C 的支撑。

我们的团队，用了 5-6 年的时间才完成了运维的完全平台化，以及建立了统一的运维数据仓库，在这之前当然也可以落地一部分 AIOps 场景，但相对应的，也是需要这部分场景的 B 和 C 具备。目前看，应该是运维平台和数据都已具备时，才是落地 AIOps 的理想时刻。

万金：在那些具有较强自动化运维能力，和一定的数据储备的条件的团队才适合 AIOps 落地，同时业务是否对运维效率提升有需求也是一个考虑因素。

引入 AIOps 之后，是否能对 AIOps 的模型或数据进行不断优化也是一个新的挑战。人工智能在一开始都不是很完美的，需要不断优化才能达到实际应用的要求。对于发现问题和问题根因分析方面的 AIOps 落地速度比较快，对于高级阶段的根据 SLA 自动调度的 AIOps 就需要比较长的优化

涂彦：标准化、工具化、自动化、数据化、场景化是前提条件，同时这也是逐一递进的关系。

随着人工智能技术的愈加成熟，只要具备业务需求的团队，都适合落地。人工智能技术在运维行业的应用与其在其它行业落地的状况类似。所以说，AI 并不是那么神秘，使用的门槛只会越来越低。但是如果要想在企业中真正用好，需要对业务理解并紧密结合技术方案的运维，比如通过智能场景规划来找到痛点，再结合数据清洗与机器学习开发来完成落地。

InfoQ：AIOps 中的数据是怎么来的？数据是必要的吗？

曲显平：数据非常必要，因为 AIOps 中的 A 主要指的是算法（数据挖掘、机器学习），每一个算法都离不开数据的支撑，尤其对于深度学习而言，普通体量的数据都不足以支撑训练出理想的模型，需要非常庞大的数据量。

AIOps 的数据，从来源上看，主要分四类：设备、系统、平台、业务，

设备主要指 IDC、网络、服务器等偏硬件、数据中心层的设备信息；系统主要指操作系统等；平台主要指基础运维平台、PaaS 平台等；业务主要指产品服务的日志等。

AIOps 的数据，从类型来看，主要也是四类：一个是时序数据，这个重要性最高，因为其结构化层次高，在百度，这个数量级能达到十亿级。第二个是运维事件数据，这个同样也非常重要，每一次异常事件、变更事件、运营事件等等，都是需要被记录并加以分析的。第三个是日志数据，相比之下日志数据的结构化偏弱一些，但同样十分重要，由于量级太过庞大，我们也只会挑选并存储较为重要的部分，来进行分析训练。当然，还有最后一类，因 AIOps 而产生的数据“标注数据”，这部分数据的完善程度将直接影响着每一个算法模型的实际效果。

万金：AIOps 中的数据是从监控系统和相关的运维经验中来。而 AIOps 也是有不同的实现的技术，比如通过机器学习算法的 AIOps 依赖数据较小，只要找到合适的算法就能对当前数据进行处理，但是处理效果随着数据模式的改变而改变，也就是当数据体现的模式改变后就必须手动更换算法适应新模式。而深度学习的算法就不必人工更换模型只需要设定目标（SLA）但对数据依赖比较多。

涂彦：训练的基础就是数据，这个与其它行业并无区别。数据来自于运维工作与服务的场景。质量、效率、成本、安全，任何一个维度都需要将数据标准化采集，这是一切的基础。再比如技术运营的一些触达到用户或者产品的场景，也是一个数据化的过程，任何一个节点的数据都可以被采集。这些标准化和数据化的工作，都是 AIOps 的基础工作。

InfoQ：可否谈谈你们的 AIOps 落地场景？

曲显平：在百度运维，AIOps 应用的很广泛，监控异常检测、故障诊断分析、智能流量调度、SQL 入侵检测、成本优化、性能分析优化等等。

我们在自动化异常检测方向的研究非常早，早期还以传统的时序数据分析为主，现在由于机器学习等方法的兴起，已经多样化很多，在百度大多数核心时序指标的监控都是用的自动化异常检测模式。

在故障诊断方向，这一直是一个比较难的课题，我们研究过非常多的行业 paper，真正实现后效果出色的少之又少，当然这也可以说，因为对于人来讲，诊断故障也一直是一个让人头疼的问题。

此外，在智能流量调度、SQL 入侵检测、成本优化、性能分析优化等层面，我们应用了非常多的机器学习模型，为公司业务的可用性、成本、性能等的提升做出了巨大贡献。

万金：故障识别和故障自愈，资源自动调度。

更高级的场景会引入全自动软件发布控制与回滚，自然语言容量管理，主动安全识别和在软件研发完成前对软件体验进行评分。

涂彦：腾讯游戏的运维团队聚焦在基础服务与增值服务中落地。基础服务包括发布变更、故障管理、用户体验服务。增值服务包括触达到用户与产品决策的运维扩展服务。发布变更中的数据预测、故障管理中的根因分析、用户体验中的多维告警等，都是 AIops 的落地场景。触达用户的云控策略、产品决策的舆情分析等，也都有着很好的应用场景。

InfoQ：目前业界有哪些可以参考的 AIops 实践？

曲显平：在 AIops 和智能化运维这个角度，百度运维是做的比较早的，我们将运维能力标准定义为六级，L0-L5，每一级都有相对明确的定义，很多方向都可以应用 AIops 来助力等级的提升。

当然除了百度，推荐看下 Netflix 的 Winston，以及 Google 的 Auxon，这些实践，不仅是用某种算法在解决一个领域的问题，他们正在尝试构建一种开放的自动化 / 智能化运维模式。

越来越多的公司关注并且重视这部分信息的开放和共享，从场景到算法，从数据到开放框架，百度运维也正在参与其中，也欢迎大家合作共建智能化新运维。

万金：在我的文章里提到一些算法方式的实践，深度学习方式由于依赖多，成本高不确定性大应用比较少。行业参考如下：

- [Twitter: Seasonal Hybrid ESD \(S-H-ESD\)](#)
- [Netflix: Robust PCA](#)

- [LinkedIn: exponential smoothing](#)
- [Uber: multivariate non-linear model](#)

涂彦: 腾讯蓝鲸智云来自于腾讯游戏数百款游戏的基础服务与增值服务的最佳实践。目前已经对外提供社区版和企业版。蓝鲸可以帮助运维团队在标准化、工具化、自动化的深度建设，结合蓝鲸数据平台，AIOps 将提供给运维团队更多想像空间。

软件测试技术的未来

作者 孙远 孟宪伟



“测试已死”的观点在业内仍然存在着争议，很多公司缩减了测试人员，开发测试比屡创新高。本文旨在通过介绍软件测试的新趋势和新技术来展示软件测试行业面临的机遇与挑战，为软件测试工程师的职业规划提供参考。

安全测试

从孟加拉国银行 8100 万美元被黑客成功盗取到美国民主党邮件泄露事件可以看出，网络安全事件已经被推到了风口浪尖。随着物联网逐步普及，智能家居、汽车电子等设备的网络化水平大幅提升。但物联网的安全

却不容乐观，很多中小企业往往忽视安全防护。开源软件的源代码公开，黑客可以通过阅读源代码更容易的分析出软件的安全漏洞，使得网络安全迎来了新的挑战。当开源社区中发布出 cve 漏洞时，需要厂商及时的合入补丁，否则将给黑客入侵敞开大门。

新的编程语言的出现在提高了编码效率的同时，也为软件产品增添了安全挑战，需要安全厂商尽快推出相应的安全工具和安全加固方案。随着 SaaS 的普及，相信会有更多的安全工具问世。渗透测试需要测试工程师阅读源码来找出漏洞，与安全合规测试相比，需要更高的技术水平。在未来相当长的一段时间内渗透测试工程师将有很大的缺口。

人工智能测试

近年来，人工智能（AI）被越来越多的应用在 IT 行业，如智能汽车、智能家居和机器人等。尤其是 2016 年 AlphaGo 在围棋领域掀起一股热潮之后，AI 更多地成为人们热议的焦点。人工智能是一个新的领域，对于人工智能本身的测试方案和测试工具还有待完善。

对于人工智能在软件测试领域的应用，即利用人工智能来优化其他软件的测试，目前已经取得了一定的进展。人工神经网络是软件测试领域使用相对广泛的 AI 技术之一。神经网络是基于生物学中神经网络的基本原理，在理解和抽象了人脑结构和外界刺激响应机制后，以网络拓扑知识为理论基础，模拟人脑的神经系统对复杂信息的处理机制的一种数学模型。目前在 OCR，语音识别，医学诊断等方面已经取得了很大的成功。在软件测试中，它非常适合 GUI 测试、内存使用测试及分布式系统功能验证等场景。

遗传算法是另一个软件测试中用到的 AI 技术。它是模仿生物遗传和进化机制的一种最优化方法，它把类似于遗传基因的一些行为，如交叉重组、变异、选择和淘汰等引入到算法求解的改进过程中。遗传算法的特点之一是，它同时保留着若干局部最优解，通过交叉重组或者解的变异来寻求更好的解。在软件单元测试中，已知输入的参数的范围，求解哪些参数

的组合能够达到最大的代码覆盖率（也有些研究是能达到最大的路径覆盖 / 分支覆盖）。因此，遗传算法可以用于选择最优的单元测试用例，也就是单元测试的最优输入集。同时利用人工智能还可以优化测试工具，将软件测试的上下文与测试用例结合起来，选择最优的测试用例集进行测试。

静态分析与符号执行

软件可靠性是对软件在设计、开发以及所预定的环境下既有能力的置信度的一个度量，是衡量软件质量的主要参数之一。而软件测试则是保证软件质量、提高软件可靠性的最重要手段。静态分析工具可以直接对源码进行扫描，但其误报率的问题有待改善。

大量可靠性问题隐藏在未知场景和不熟悉的开源代码中，有必要通过程序行为分析工具来遍历各种异常分支、代码的所有路径。符号执行技术是精确的路径遍历，是随机测试、FUZZ 测试的有益补充。

符号执行代表工具 KLEE，在第一次学术使用（2008）便发现了 unix 系统中最常用的程序的多个问题，有的问题已经存在超过 15 年。符号执行技术在之前没有得到大规模应用，主要原因是技术本身需要大量的计算资源（路径爆炸）。随着软硬件技术的发展，平均计算成本比之前降低了很多，为符号执行的发展和推广提供了有利的客观条件。目前符号执行技术已应用在许多公司的产品测试当中，如 HP、微软等公司都已经有 10 年以上的符号执行探索经验。当前基于 KLEE 的二次开发工具已经大量应用在软件可靠性测试中，如 Mayhem 已发现了 DebianOS 的上千个 crash 问题，以及 Linux 和 Windows 系统的几十个可利用漏洞。

精准测试

在当前敏捷测试的时代，版本发布日趋频繁，快速发布高质量的软件是很多企业的目标。对于急于发布的软件版本，全量运行所有的用例往往需要花费较长的时间，已经不能满足产品发布的节奏。如何避免过度测试并在时间、质量、成本中找到最佳的平衡？

精准测试可让软件测试过程可量化衡量、可追溯，清楚的展示出测试用例运行的路径，并可以实现测试用例与代码的双向追踪。对于代码量较大的系统的软件，通过精准测试可以获取到曾经执行过某段代码的测试用例，当这部分代码进行修改后，只需执行对应的用例即可，大大缩短了测试的时间，加快了产品上线速度。因此精准测试成为了近期软件测试技术的新方向之一。精准测试的实施对测试人员的代码开发、测试设计、需求理解、架构理解、自动化测试能力均有较高的要求。

云测试

云计算是一种按需提供计算资源的技术，它可以减少用户基础设施投入并降低管理成本。然而，云平台在近年来不断出现大面积宕机的情况，这为云计算测试技术提出了新的挑战。需要测试人员深入理解云平台底层、中间层和上层技术，构建符合云平台质量要求的测试工程能力和质量保障方案。

很多测试服务提供商已经将测试服务部署到云上，这种方式有很多的优势。首先，它可以按需提供服务，用户可以根据需求灵活的占用云端资源，避免了传统测试中的资源浪费。例如手机应用提供商可以把应用程序通过云平台进行主流手机的兼容性测试，而不必直接购买各品牌的手机。其次，云平台可以提供较为全面的测试环境和测试工具，免去了部署环境和工具的时间，使测试工程师可以将更多的精力投入到业务中。再次，当云平台和容器技术结合起来时，可以快速构建可扩展可伸缩的测试环境，并行执行测试用例，从而减少测试执行时间。

物联网测试

IoT 是一个包含大量网络设备、传感器和计算基础设施的庞大系统。IoT 的应用覆盖了军事、家庭、医疗、零售等多个领域。其使用场景复杂，解决方案多元化，使得 IoT 设备以及解决方案的测试面临很大的挑战。下面笔者提出了一些观点和思路供大家参考。

1. 仿真

基于效率和成本的考虑，测试人员无法针对所有的 IoT 设备、连接协议以及服务节点进行全面覆盖。依靠 IoT 场景仿真能力，测试人员可以在少量可用的物理设备上创建各类虚拟设备并建立不同协议的虚拟连接，从而模拟出真实应用场景，达到全面测试覆盖的目的。不仅能够节约时间和成本，还具有更好的灵活性和扩展性。

2. 安全

当前 IoT 发展的重点是技术的创新、推广和应用，安全问题没有受到足够的重视。相对传统移动互联网，IoT 的规模、应用和服务都更加庞大复杂，安全问题无疑具有极大的挑战性。

3. 自动化

在 IoT 领域，目前自动化测试工具和系统的发展还处于比较初级的阶段。在测试执行、场景构建、性能度量及状态监控等各个方面都需要有强有力的工具、框架和规范的出现，来支撑复杂的 IoT 自动化测试。

开源测试

开源软件本着“不要重复造轮子”的原则，与商业软件相比，拥有使用成本低、可定制性高等特点。目前开源测试工具种类繁多，涵盖测试管理、缺陷管理、持续集成、功能测试、性能测试、测试框架、测试设计、安全测试等类别。下面列举了这些分类中一些典型的测试工具。而针对我们自身的业务需求，可以通过修改源代码来适配自己的业务，从而实现工具定制化。

- 测试管理：TestLink、Testopia
- 缺陷管理：Redmine、Bugzilla、Mantis
- 持续集成：Jenkins、Buildbot
- 功能测试：Selenium、LTP
- 性能测试：lmbench、Sysbench、Iperf、Fio

- 测试框架: JUnit、Autotest
- 测试设计: XMind、StarUML、UML Designer
- 安全测试: Metasploit、Nessus、AppScan

为了减少研发成本，很多公司都制定了基于开源软件进行二次开发的策略。在重点测试自研特性的同时，面对大量的开源代码，测试工程师需要与开源社区互动，及时将发现的问题提交给社区并同步社区的问题单和 CVE 补丁。

然而，当前很多开源社区中的测试并不到位，很多特性在发布之后长时间没有对应的文档和测试用例。以 kernel 社区的 user namespace 内核特性为例，其是在 2013 年 2 月 18 日随内核 3.8 版本正式发布的，然而直到 2015 年 5 月 21 日，社区才拥有第一个该特性的测试用例。二者时间间隔在两年以上，版本的质量保证令人堪忧。对于这部分特性，需要测试工程师根据业务需要自行补充测试。测试工程师同时还要注意构建社区影响力，以保证与自己平台相关的测试用例能够顺利的被社区接收，从而减少测试代码维护成本。感兴趣的读者可以阅读《让我们成为开源软件测试者》。

容器化 /Devops/ 微服务

容器为开发、测试、运维三个团队提供一致的环境，避免因为环境不统一产生的缺陷误报。同时使开发人员可以很容易的通过容器镜像复现测试人员和客户报来的缺陷。利用容器还可以避免环境污染和批量快速的启动多个测试环境并行测试来提高测试效率。微服务将软件细分为多个子模块，各模块间相对独立，便于测试进行迁移以便及早的发现缺陷。Devops 通过成熟的自动化解决方案，同时配合容器、微服务技术，打通了开发、测试、运维团队的壁垒。随着容器、微服务时代的到来，配置基于 CI/CD 的 Devops 流程成为了测试人员必备的技能。感兴趣的读者可以阅读《Docker 引领测试革新》。

敏捷测试

传统的软件测试方法将开发和测试视作两个团队的两种不同的工作模式，团队之间沟通比较有限，团队壁垒较为明显。在这种开发模式下，软件缺陷通常在项目后期才逐步被发现。近年来，在客户需求频繁变化、高强度的外部竞争压力和软件交付迭代频繁的大环境下，传统的软件测试方式已经不能满足需求。

敏捷测试强调从客户的角度进行测试，重点关注持续迭代地测试新开发的功能，而不再强调传统测试过程中严格的测试阶段，同时提倡尽早开始测试。它强调开发和测试团队在合作、透明、灵活的环境下协同工作，以测试前移、持续集成、自动化等方式为优化手段，可以很好的适应快速、需求变更频繁的软件交付。

目前敏捷测试已经受到了行业内的认可，相信会有更多的公司将会进行敏捷转型，敏捷教练的薪水也会水涨船高。

大数据测试

当前全球信息数据量增长迅猛，据市场调研机构 IDC 预测，到 2020 年，全球数据总量将达到 40ZB，相当于每人拥有一千张 DVD 光盘以上的信息量。如此大量的数据为测试数据的备份和管理带来了挑战，测试人员需要确认数据完整性，保证数据质量。面对大量而动态变化的数据和有限的测试时间，需要制定出行之有效的测试策略，开发出适用的测试工具，并完善自动化测试。

随着大数据应用的快速增长，我们需要更快速的完成数据处理。大数据挖掘的目的是找出数据与数据的关联关系，与传统软件相比，很多大数据场景中的输出是无法直接确定的，同时数据又具有多样性，需要测试人员具备更多的发散思维；面对爆炸式的数据服务，测试时需要搭建可扩展伸缩的测试平台模拟大量的测试客户端。而面对大数据中很多场景下程序输出的不确定性、大数据结构多样化、定位数据因果关系困难等问题为测

试工程师带来了新的挑战。

自动化测试

传统的自动化测试需要测试工程师直接编写测试程序，而这样的程序往往可维护性不强，当开发代码变更时往往需要重新适配自动化测试程序。测试驱动开发是软件工程中的一个里程碑，即开发在提交开发代码修改时同时要提交测试代码，但这种方式仍然需要较多的人力投入到测试代码的编写中。而一些程序可以通过录制或符号执行等方法自动生成自动化代码，免去了手工编写的不便。另外通过埋点、mock 等技术还可以辅助自动化测试。随着测试业务日趋多样化，需要不断开发新的自动化测试框架、测试平台来满足业务需求。当自动化测试与云平台相结合时，可以方便的进行任务迁移、回滚、故障自动修复等功能。

移动互联网测试

随着智能移动设备的普及，测试范畴也从智能手机、智能平板扩展延伸至包含了运动手环、车载联网应用、共享单车、无人机等事物。移动平台也呈现多样化趋势，而每个平台的版本升级速度非常快。移动应用种类繁多，从社交到游戏、教育、办公、旅行、工具等类别。为满足用户需求，热门应用的迭代更新非常频繁。面对众多的移动设备硬件型号、多个终端平台版本、繁多的移动应用、各应用的不同版本号，测试人员不得不制定新的测试策略和方案来应对业务。即使应用没有新的特性引入，但自动化测试不得不根据新的平台进行适配工作。而多种组合的测试为测试人员、测试工程能力、自动化测试提出了更高的要求。

目前已经出现了针对移动测试的自动化设备和 SAAS 平台。测试设备可以模拟出用户真实的终端操作的方式。在 SAAS 平台中，使用者可以将应用提交到平台中进行全量的自动化测试，来确保应用的多个版本可以适配到不同的平台和硬件中。此外，领域中的专项测试，如性能测试、功耗测试、安全测试、兼容性测试、跨地域跨时区测试、老化测试，也将产生

很大的测试需求。

总结

当前很多公司已经将基本的功能测试任务交由开发团队负责，测试人员主要专注于自动化测试开发、安全测试、测试建模、精准测试、性能测试、可靠性测试等专项测试中。这部分测试任务能够很好的体现测试人员的价值。虽然“测试已死”的争论还在继续，但只要把握好软件测试发展的趋势并凭借自身的努力，相信测试人员是能够在行业中受到认可的。

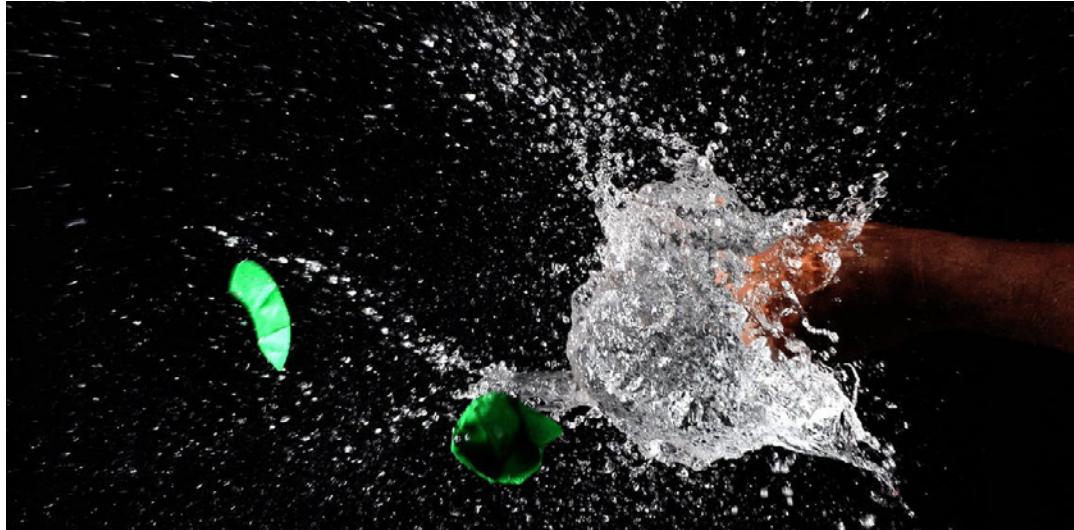
作者简介

孙远，华为中央软件院资深工程师，个人网站：www.enjoytesting.cn。

孟宪伟，华为中央软件院资深工程师，硕士毕业，软件行业10年以上从业经验。目前从事华为云计算产品虚拟化平台的测试以及相关测试工具的开发工作。加入华为前供职于IBM，Thomson Reuters以及VMware等多家公司，期间曾从事软件开发，项目管理，软件测试等各类工作。

gRPC 客户端创建和调用原理解析

作者 李林锋



1. gRPC 客户端创建流程

1.1. 背景

gRPC 是在 HTTP/2 之上实现的 RPC 框架，HTTP/2 是第 7 层（应用层）协议，它运行在 TCP（第 4 层 – 传输层）协议之上，相比于传统的 REST/JSON 机制有诸多的优点：

- 基于HTTP/2之上的二进制协议（Protobuf序列化机制）；
- 一个连接上可以多路复用，并发处理多个请求和响应；
- 多种语言的类库实现；
- 服务定义文件和自动代码生成（.proto文件和Protobuf编译工具）。

此外，gRPC 还提供了很多扩展点，用于对框架进行功能定制和扩展，例如，通过开放负载均衡接口可以无缝的与第三方组件进行集成对接（Zookeeper、域名解析服务、SLB 服务等）。

一个完整的 RPC 调用流程示例如下：

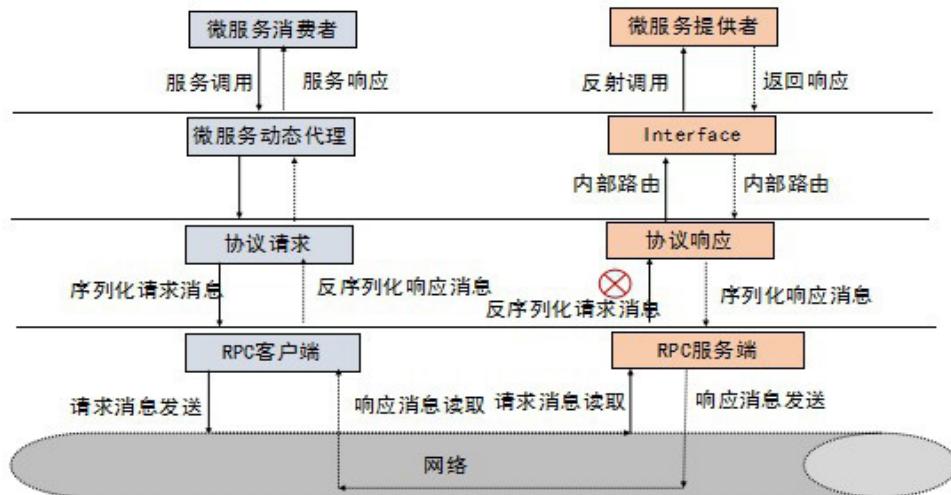


图 1 通用 RPC 调用流程

gRPC 的 RPC 调用与上述流程相似，下面我们一起来学习下 gRPC 的客户端创建和服务调用流程。

1.2. 业务代码示例

以 gRPC 入门级的 helloworld Demo 为例，客户端发起 RPC 调用的代码主要包括如下几部分：

- 1) 根据 hostname 和 port 创建 ManagedChannelImpl
 - 2) 根据 helloworld.proto 文件生成的 GreeterGrpc 创建客户端 Stub，用来发起 RPC 调用
 - 3) 使用客户端 Stub(GreeterBlockingStub)发起 RPC 调用，获取响应。
- 相关示例代码如下所示。

1.3. RPC 调用流程

```

public HelloWorldClient(String host, int port) {
    an be private more... (Ctrl+F1) elBuilder.forAddress(host, port)
        .usePlaintext(true));
}

/** Construct client for accessing RouteGuide server using the existing channel.
HelloWorldClient(ManagedChannelBuilder<?> channelBuilder) {
    channel = channelBuilder.build();
    blockingStub = GreeterGrpc.newBlockingStub(channel);
}

/** Say hello to server. */
public void greet(String name) {
    logger.info(msg: "Will try to greet " + name + " ...");
    HelloRequest request = HelloRequest.newBuilder().setName(name).build();
    HelloReply response;
    try {
        response = blockingStub.sayHello(request);
    }
}

```

gRPC 的客户端调用主要包括基于 Netty 的 HTTP/2 客户端创建、客户端负载均衡、请求消息的发送和响应接收处理四个流程。

1.3.1. 客户端调用总体流程

gRPC 的客户端调用总体流程如下图所示：

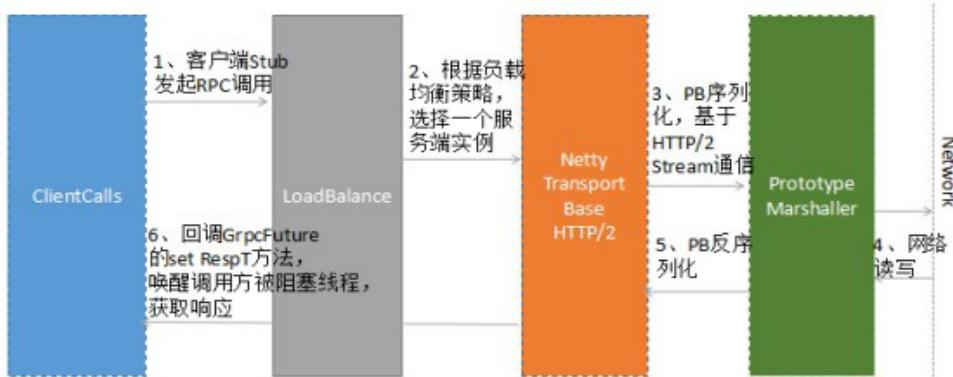


图 2 gRPC 总体调用流程

gRPC 的客户端调用流程如下：

- 1) 客户端 Stub(GreeterBlockingStub) 调用 sayHello(request)，发起 RPC 调用。
- 2) 通过 DnsNameResolver 进行域名解析，获取服务端的地址信息（列表），随后使用默认的 LoadBalancer 策略，选择一个具体的 gRPC 服务端实例。

3) 如果与路由选中的服务端之间没有可用的连接，则创建 NettyClientTransport 和 NettyClientHandler，发起 HTTP/2 连接。

4) 对请求消息使用 PB (Protobuf) 做序列化，通过 HTTP/2 Stream 发送给 gRPC 服务端。

5) 接收到服务端响应之后，使用 PB (Protobuf) 做反序列化。

6) 回调 GrpcFuture 的 set(Response) 方法，唤醒阻塞的客户端调用线程，获取 RPC 响应。

需要指出的是，客户端同步阻塞 RPC 调用阻塞的是调用方线程（通常是业务线程），底层 Transport 的 I/O 线程（Netty 的 NioEventLoop）仍然是非阻塞的。

1.3.2. ManagedChannel 创建流程

ManagedChannel 是对 Transport 层 SocketChannel 的抽象，Transport 层负责协议消息的序列化和反序列化，以及协议消息的发送和读取。ManagedChannel 将处理后的请求和响应传递给与之相关联的 ClientCall 进行上层处理，同时，ManagedChannel 提供了对 Channel 的生命周期管理（链路创建、空闲、关闭等）。

ManagedChannel 提供了接口式的切面 ClientInterceptor，它可以拦截 RPC 客户端调用，注入扩展点，以及功能定制，方便框架的使用者对 gRPC 进行功能扩展。

ManagedChannel 的主要实现类 ManagedChannelImpl 创建流程如下：

流程关键技术点解读：

使用 builder 模式创建 ManagedChannelBuilder 实现类 NettyChannelBuilder，NettyChannelBuilder 提供了 buildTransportFactory 工厂方法创建 NettyTransportFactory，最终用于创建 NettyClientTransport。

初始化 HTTP/2 连接方式：采用 plaintext 协商模式还是默认的 TLS 模式，HTTP/2 的连接有两种模式：h2（基于 TLS 之上构建的 HTTP/2）和 h2c（直接在 TCP 之上构建的 HTTP/2）。

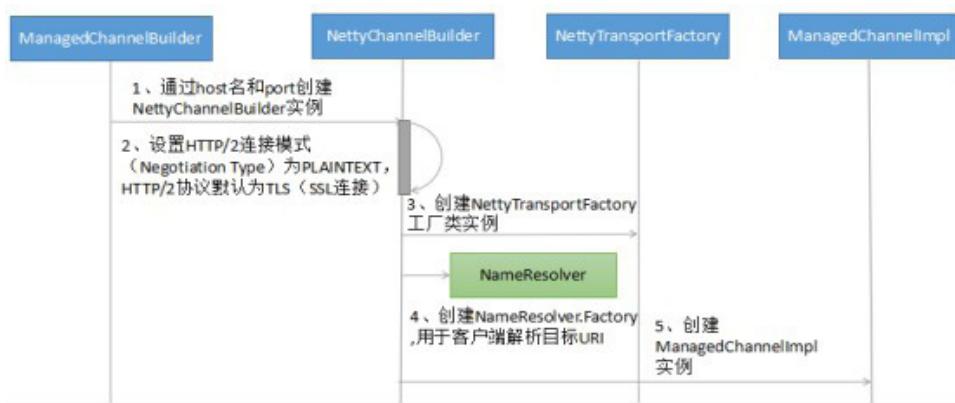


图 3 ManagedChannelImpl 创建流程

创建 NameResolver.Factory 工厂类，用于服务端 URI 的解析，gRPC 默认采用 DNS 域名解析方式。

ManagedChannel 实例构造完成之后，即可创建 ClientCall，发起 RPC 调用。

1.3.3. ClientCall 创建流程

完成 ManagedChannelImpl 创建之后，由 ManagedChannelImpl 发起创建一个新的 ClientCall 实例。ClientCall 的用途是业务应用层的消息调度和处理，它的典型用法如下：

```

call = channel.newCall(unaryMethod, callOptions);
call.start(listener, headers);
call.sendMessage(message);
call.halfClose();
call.request(1);
// wait for listener.onMessage()

```

ClientCall 实例的创建流程如下所示。

流程关键技术点解读：

ClientCallImpl 的主要构造参数是 MethodDescriptor 和 CallOptions，其中 MethodDescriptor 存放了需要调用 RPC 服务的接口名、方法名、服务调用的方式（例如 UNARY 类型）以及请求和响应的序列化和反序列化实现类。CallOptions 则存放了 RPC 调用的其它附加信息，例如

超时时间、鉴权信息、消息长度限制和执行客户端调用的线程池等。

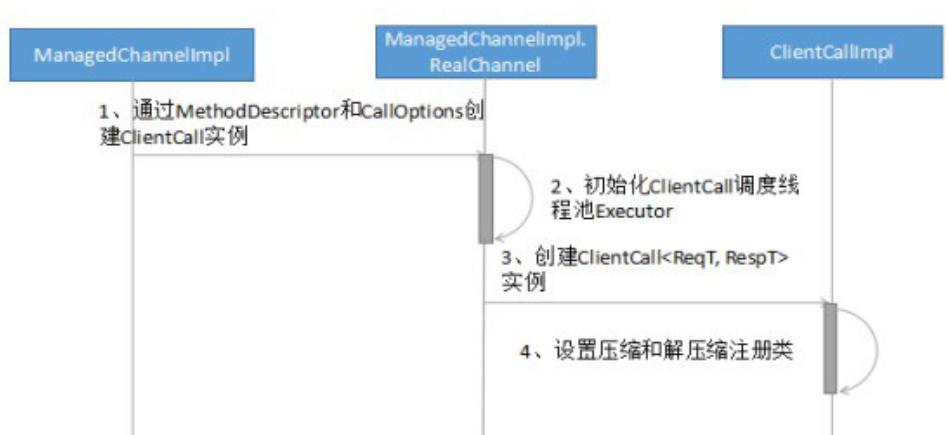


图 4 ClientCallImpl 创建流程

设置压缩和解压缩的注册类（CompressorRegistry 和 DecompressorRegistry），以便可以按照指定的压缩算法对 HTTP/2 消息做压缩和解压缩。

ClientCallImpl 实例创建完成之后，就可以调用 ClientTransport，创建 HTTP/2 Client，向 gRPC 服务端发起远程服务调用。

1.3.4. 基于 Netty 的 HTTP/2 Client 创建流程

gRPC 客户端底层基于 Netty4.1 的 HTTP/2 协议栈框架构建，以便可以使用 HTTP/2 协议来承载 RPC 消息，在满足标准化规范的前提下，提升通信性能。

gRPC HTTP/2 协议栈（客户端）的关键实现是 NettyClientTransport 和 NettyClientHandler，客户端初始化流程如图 5 所示。

流程关键技术点解读：

1. NettyClientHandler 的创建：级联创建 Netty 的 Http2FrameReader、Http2FrameWriter 和 Http2Connection，用于构建基于 Netty 的 gRPC HTTP/2 客户端协议栈。

2. HTTP/2 Client 启动：仍然基于 Netty 的 Bootstrap 来初始化并启动客户端，但是有两个细节需要注意：

- NettyClientHandler（实际被包装成 ProtocolNegotiator.

Handler，用于HTTP/2的握手协商) 创建之后，不是由传统的 ChannelInitializer在初始化Channel时将NettyClientHandler加入到pipeline中，而是直接通过Bootstrap的handler方法直接加入到pipeline中，以便可以立即接收发送任务。

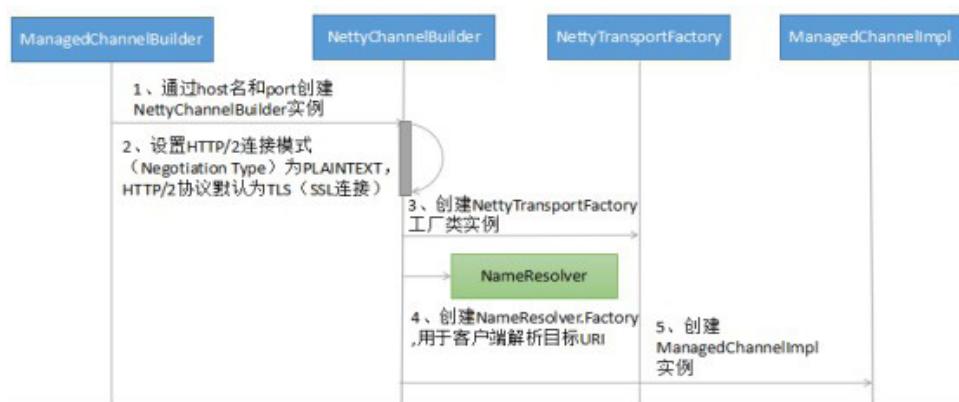


图 5 HTTP/2 Client 创建流程

- 客户端使用的work线程组并非通常意义的EventLoopGroup，而是一个EventLoop：即HTTP/2客户端使用的work线程并非一组线程（默认线程数为CPU内核 * 2），而是一个EventLoop线程。这个其实也很容易理解，一个NioEventLoop线程可以同时处理多个HTTP/2客户端连接，它是多路复用的，对于单个HTTP/2客户端，如果默认独占一个work线程组，将造成极大的资源浪费，同时也可能会导致句柄溢出（并发启动大量HTTP/2客户端）。
3. WriteQueue 创建：Netty 的 NioSocketChannel 初始化并向 Selector 注册之后（发起 HTTP 连接之前），立即由 NettyClientHandler 创建 WriteQueue，用于接收并处理 gRPC 内部的各种 Command，例如链路关闭指令、发送 Frame 指令、发送 Ping 指令等。

HTTP/2 Client 创建完成之后，即可由客户端根据协商策略发起 HTTP/2 连接。如果连接创建成功，后续即可复用该 HTTP/2 连接，进行 RPC 调用。

1.3.5. HTTP/2 连接创建流程

HTTP/2 在 TCP 连接之初通过协商的方式进行通信，只有协商成功，才能进行后续的业务层数据发送和接收。

HTTP/2 的版本标识分为两类：

- 基于TLS之上构架的HTTP/2，即HTTPS，使用h2表示（ALPN）：
0x68与0x32；
- 直接在TCP之上构建的HTTP/2，即HTTP，使用h2c表示。

HTTP/2 连接创建，分为两种：通过协商升级协议方式和直接连接方式。

假如不知道服务端是否支持 HTTP/2，可以先使用 HTTP/1.1 进行协商，客户端发送协商请求消息（只含消息头），报文示例如下：

```
GET / HTTP/1.1
Host: 127.0.0.1
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

服务端接收到协商请求之后，如果不支持 HTTP/2，则直接按照 HTTP/1.1 响应返回，双方通过 HTTP/1.1 进行通信，报文示例如下：

```
HTTP/1.1 200 OK
Content-Length: 28
Content-Type: text/css
```

body...

如果服务端支持 HTTP/2，则协商成功，返回 101 结果码，通知客户端一起升级到 HTTP/2 进行通信，示例报文如下：

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

[HTTP/2 connection...

101 响应之后，服务需要发送 SETTINGS 帧作为连接序言，客户端接收到 101 响应之后，也必须发送一个序言作为回应，示例如下：

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n\r\n
```

SETTINGS帧

客户端序言发送完成之后，可以不需要等待服务端的 SETTINGS 帧，而直接发送业务请求 Frame。

假如客户端和服务端已经约定使用 HTTP/2，则可以免去 101 协商和切换流程，直接发起 HTTP/2 连接，具体流程如图 6 所示。



图 6 HTTP/2 直接连接过程

几个关键点：

- 如果已经明确知道服务端支持HTTP/2，则可免去通过HTTP/1.1 101协议切换方式进行升级。TCP连接建立之后即可发送序言，否则只能在接收到服务端101响应之后发送序言。
- 针对一个连接，服务端第一个要发送的帧必须是SETTINGS帧，连接序言所包含的SETTINGS帧可以为空。
- 客户端可以在发送完序言之后发送应用帧数据，不用等待来自服务器端的序言SETTINGS帧。

gRPC 支持三种 Protocol Negotiator 策略：

- PlaintextNegotiator：明确服务端支持HTTP/2，采用HTTP直接连

接的方式与服务端建立HTTP/2连接，省去101协议切换过程。

- PlaintextUpgradeNegotiator: 不清楚服务端是否支持HTTP/2，采用HTTP/1.1协商模式切换升级到HTTP/2。
- TlsNegotiator: 在TLS之上构建HTTP/2，协商采用ALPN扩展协议，以“h2”作为协议标识符。

下面我们以PlaintextNegotiator为例，了解下基于Netty的HTTP/2连接创建流程：

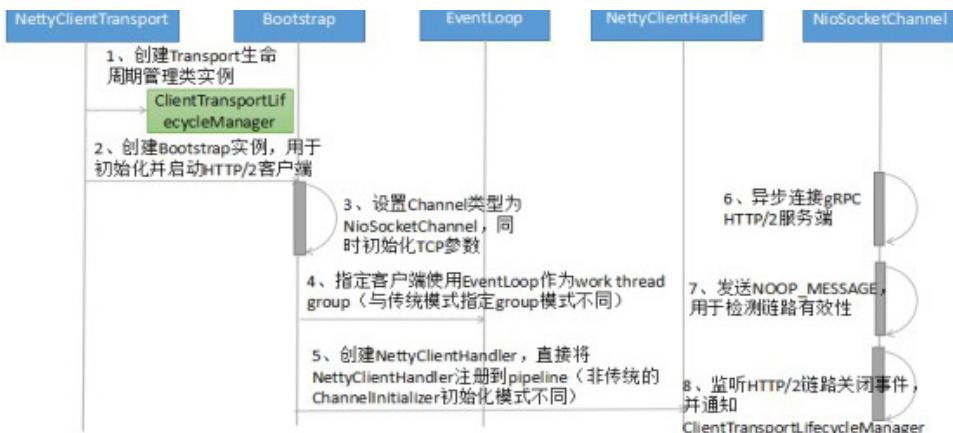


图 7 基于 Netty 的 HTTP/2 直连流程

1.3.6. 负载均衡策略

总体上看，RPC的负载均衡策略有两大类：

- 服务端负载均衡（例如代理模式、外部负载均衡服务）
- 客户端负载均衡（内置负载均衡策略和算法，客户端实现）

外部负载均衡模式如图8所示。

以代理LB模式为例：RPC客户端向负载均衡代理发送请求，负载均衡代理按照指定的路由策略，将请求消息转发到后端可用的服务实例上。负载均衡代理负责维护后端可用的服务列表，如果发现某个服务不可用，则将其剔除出路由表。

代理LB模式的优点是客户端不需要实现负载均衡策略算法，也不需要维护后端的服务列表信息，不直接跟后端的服务进行通信，在做网络安全边界隔离时，非常实用。例如通过Ngix做L7层负载均衡，将互联网前

端的流量安全的接入到后端服务中。

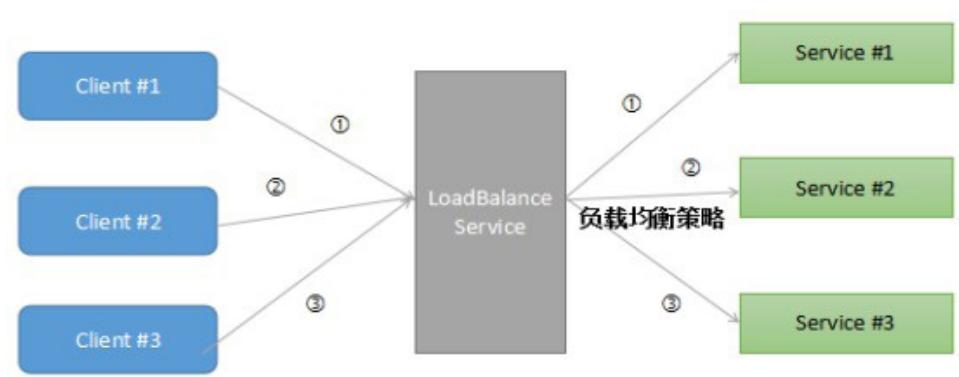


图 8 代理负载均衡模式示意图

代理 LB 模式通常支持 L4 (Transport) 和 L7 (Application) 层负载均衡，两者各有优缺点，可以根据 RPC 的协议特点灵活选择。L4/L7 层负载均衡对应场景如下：

- L4层：对时延要求苛刻、资源损耗少、RPC本身采用私有TCP协议
- L7层：有会话状态的连接、HTTP协议簇（例如Restful）

客户端负载均衡策略由客户端内置负载均衡能力，通过静态配置、域名解析服务（例如 DNS 服务）、订阅发布（例如 Zookeeper 服务注册中心）等方式获取 RPC 服务端地址列表，并将地址列表缓存到客户端内存中。每次 RPC 调用时，根据客户端配置的负载均衡策略由负载均衡算法从缓存的服务地址列表中选择一个服务实例，发起 RPC 调用。

客户端负载均衡策略工作原理示例如图 9 所示。

gRPC 默认采用客户端负载均衡策略，同时提供了扩展机制，使用者通过自定义实现 NameResolver 和 LoadBalancer，即可覆盖 gRPC 默认的负载均衡策略，实现自定义路由策略的扩展。

gRPC 提供的负载均衡策略实现类如下所示：

- PickFirstBalancer：无负载均衡能力，即使有多个服务端地址可用，也只选择第一个地址。

- RoundRobinLoadBalancer: “RoundRobin” 负载均衡策略。

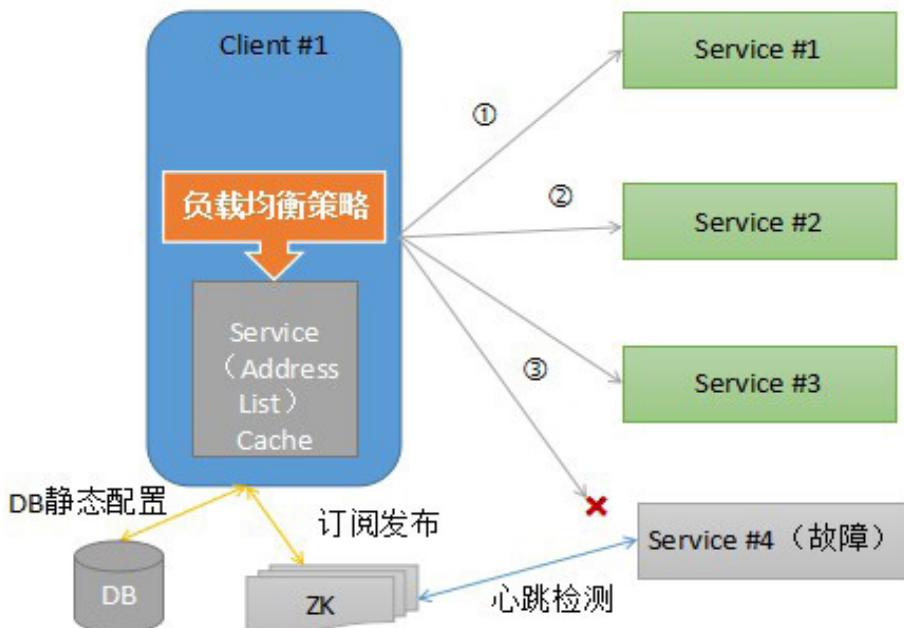


图 9 客户端负载均衡策略示意图

gRPC 负载均衡流程如图 10 所示。

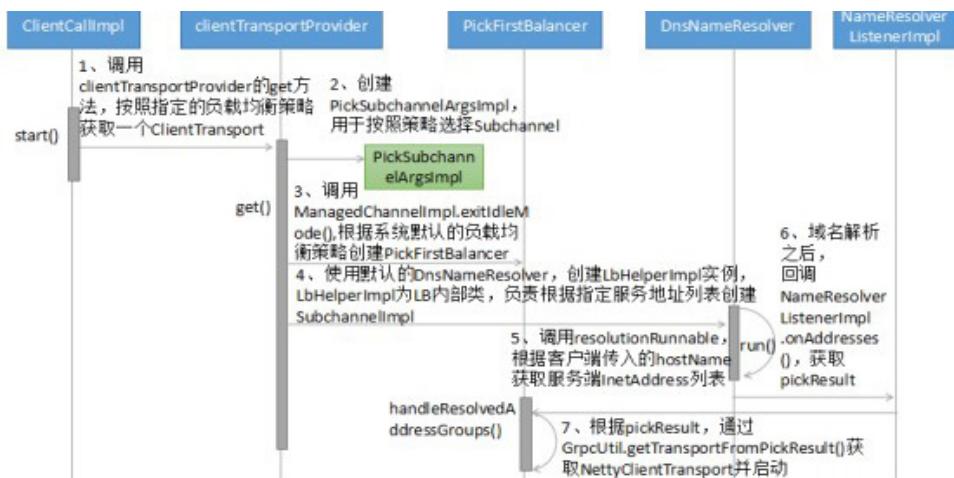


图 10 gRPC 客户端负载均衡流程图

流程关键技术点解读：

1. 负载均衡功能模块的输入是客户端指定的 hostName、需要调用的接口名和方法名等参数，输出是执行负载均衡算法后获得的

NettyClientTransport。通过 NettyClientTransport 可以创建基于 Netty HTTP/2 的 gRPC 客户端，发起 RPC 调用。

2. gRPC 系统默认提供的是 DnsNameResolver，它通过 InetAddress.getAllByName(host) 获取指定 host 的 IP 地址列表（本地 DNS 服务）。

对于扩展者而言，可以继承 NameResolver 实现自定义的地址解析服务，例如使用 Zookeeper 替换 DnsNameResolver，把 Zookeeper 作为动态的服务地址配置中心，它的伪代码示例如下：

第一步：继承 NameResolver，实现 start(Listener listener) 方法：

```
void start(Listener listener)
{
    //获取ZooKeeper地址，并连接
    //创建Watcher，并实现process(WatchedEvent event)，监听地址变更
    //根据接口名和方法名，调用getChildren方法，获取发布该服务的地址列表
    //将地址列表加到List中
    //调用NameResolver.Listener.onAddresses()，通知地址解析完成
```

第二步：创建 ManagedChannelBuilder 时，指定 Target 的地址为 Zookeeper 服务端地址，同时设置 nameResolver 为 Zookeeper NameResolver，示例代码如下所示：

```
this(ManagedChannelBuilder.forName(zookeeperAddr)
        .loadBalancerFactory(RoundRobinLoadBalancerFactory.
getinstance())
        .nameResolverFactory(new ZookeeperNameResolverProvid
er())
        .usePlaintext(false));
```

3. LoadBalancer 负责从 nameResolver 中解析获得的服务端 URL 中按照指定路由策略，选择一个目标服务端地址，并创建 ClientTransport。同样，可以通过覆盖 handleResolvedAddressGroups 实现自定义负载均衡策略。

通过 LoadBalancer + NameResolver，可以实现灵活的负载均衡策略

扩展。例如基于 Zookeeper、etcd 的分布式配置服务中心方案。

1.3.7. RPC 请求消息发送流程

gRPC 默认基于 Netty HTTP/2 + PB 进行 RPC 调用，请求消息发送流程如下所示：



图 11 gRPC 请求消息发送流程图

流程关键技术点解读：

- ClientCallImpl的sendMessage调用，主要完成了请求对象的序列化（基于PB）、HTTP/2 Frame的初始化。
- ClientCallImpl的halfClose调用将客户端准备就绪的请求Frame封装成自定义的SendGrpcFrameCommand，写入到WriteQueue中。
- WriteQueue执行flush()将SendGrpcFrameCommand写入到Netty的Channel中，调用Channel的write方法，被NettyClientHandler拦截到，由NettyClientHandler负责具体的发送操作。
- NettyClientHandler调用Http2ConnectionEncoder的writeData方法，将Frame写入到HTTP/2 Stream中，完成请求消息的发送。

1.3.8. RPC 响应接收和处理流程

gRPC 客户端响应消息的接收入口是 NettyClientHandler，它的处理流程如图 12 所示。

流程关键技术点解读：

- NettyClientHandler的onHeadersRead(int streamId,

Http2Headers headers, boolean endStream)方法会被调用两次，根据endStream判断是否是Stream结尾。

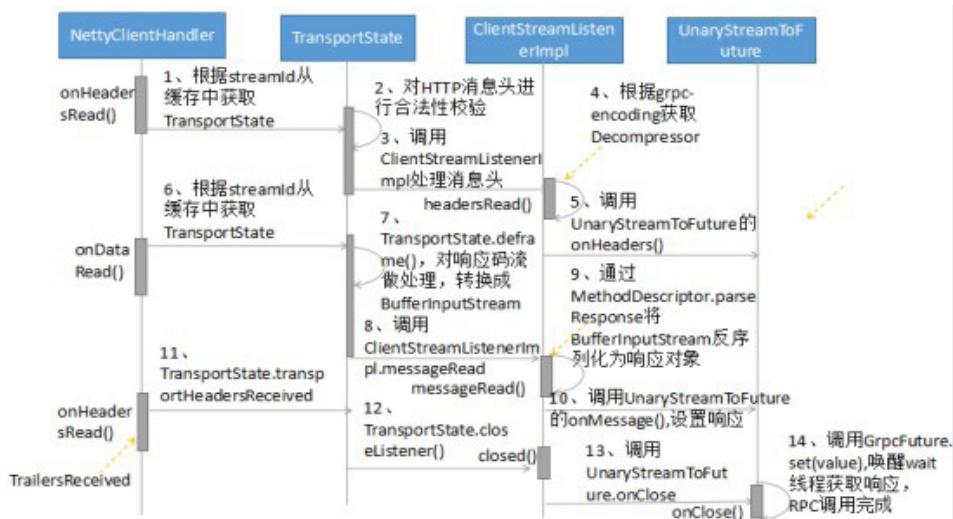


图 12 gRPC 响应消息接收流程图

- 请求和响应的关联：根据 streamId 可以关联同一个 HTTP/2 Stream，将 NettyClientStream 缓存到 Stream 中，客户端就可以在接收到响应消息头或消息体时还原出 NettyClientStream，进行后续处理。
- RPC 客户端调用线程的阻塞和唤醒使用到了 GrpcFuture 的 wait 和 notify 机制，来实现客户端调用线程的同步阻塞和唤醒。
- 客户端和服务端的 HTTP/2 Header 和 Data Frame 解析共用同一个方法，即 MessageDeframer 的 deliver()。

2. 客户端源码分析

gRPC 客户端调用原理并不复杂，但是代码却相对比较繁杂。下面围绕关键的类库，对主要功能点进行源码分析。

2.1. NettyClientTransport功能和源码分析

NettyClientTransport 的主要功能如下：

- 通过 start(Listener transportListener) 创建 HTTP/2 Client，

并连接gRPC服务端

- 通过newStream(MethodDescriptor<?, ?> method, Metadata headers, CallOptions callOptions) 创建ClientStream
- 通过shutdown() 关闭底层的HTTP/2连接

以启动 HTTP/2 客户端为例进行讲解。

```
EventLoop eventLoop = group.next();
if (keepAliveTimeNanos != KEEPALIVE_TIME_NANOS_DISABLED) {
    keepAliveManager = new KeepAliveManager(
        new ClientKeepAlivePinger(transport: this), eventLoop, keepAliveTimeNanos, keepAliveTimeoutNanos,
        keepAliveWithoutCalls);
}

handler = NettyClientHandler.newHandler(lifecycleManager, keepAliveManager, flowControlWindow,
    maxHeaderListSize, Ticker.systemTicker(), tooManyPingsRunnable);
HandlerSettings.setAutoWindow(handler);

negotiationHandler = negotiator.newHandler(handler);
```

根据启动时配置的 HTTP/2 协商策略，以 NettyClientHandler 为参数创建 ProtocolNegotiator.Handler。

创建 Bootstrap，并设置 EventLoopGroup，需要指出的是，此处并没有使用 EventLoopGroup，而是它的一种实现类 EventLoop，原因在前文中已经说明，相关代码示例如下：

```
Bootstrap b = new Bootstrap();
b.group(eventLoop);
b.channel(channelType);
if (NioSocketChannel.class.isAssignableFrom(channelType)) {
    b.option(SO_KEEPALIVE, true);
}
```

创建 WriteQueue 并设置到 NettyClientHandler 中，用于接收内部的各种 QueuedCommand，初始化完成之后，发起 HTTP/2 连接，代码如下图。

2.2. NettyClientHandler功能和源码分析

NettyClientHandler 继承自 Netty 的 Http2ConnectionHandler，是 gRPC 接收和发送 HTTP/2 消息的关键实现类，也是 gRPC 和 Netty 的交互

```

    handler.startWriteQueue(channel);
    // Start the connection operation to the server.
    channel.connect(address).addListener((ChannelFutureListener) (future) -> {
        if (!future.isSuccess()) {
            ChannelHandlerContext ctx = future.channel().pipeline().context(handler);
            if (ctx != null) {
                //...
                ctx.fireExceptionCaught(future.cause());
            }
            future.channel().pipeline().fireExceptionCaught(future.cause());
        }
    });
}

```

桥梁，它的主要功能如下所示：

- 发送各种协议消息给gRPC服务端
- 接收gRPC服务端返回的应答消息头、消息体和其它协议消息
- 处理HTTP/2协议相关的指令，例如StreamError、ConnectionError等。

协议消息的发送：无论是业务请求消息，还是协议指令消息，都统一封装成QueuedCommand，由NettyClientHandler拦截并处理，相关代码如下所示：

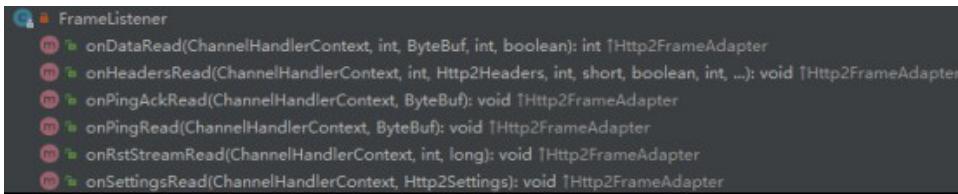
```

NettyClientHandler write() {
    if (msg instanceof CreateStreamCommand) {
        createStream((CreateStreamCommand) msg, promise);
    } else if (msg instanceof SendGrpcFrameCommand) {
        sendGrpcFrame(ctx, (SendGrpcFrameCommand) msg, promise);
    } else if (msg instanceof CancelClientStreamCommand) {
        cancelStream(ctx, (CancelClientStreamCommand) msg, promise);
    } else if (msg instanceof SendPingCommand) {
        sendPingFrame(ctx, (SendPingCommand) msg, promise);
    } else if (msg instanceof GracefulCloseCommand) {
        gracefulClose(ctx, (GracefulCloseCommand) msg, promise);
    } else if (msg instanceof ForcefulCloseCommand) {
        forcefulClose(ctx, (ForcefulCloseCommand) msg, promise);
    } else if (msg == NOOP_MESSAGE) {
        ctx.write(Unpooled.EMPTY_BUFFER, promise);
    } else {
        throw new AssertionError("Write called for unexpected type: " + msg.getClass().getName());
    }
}

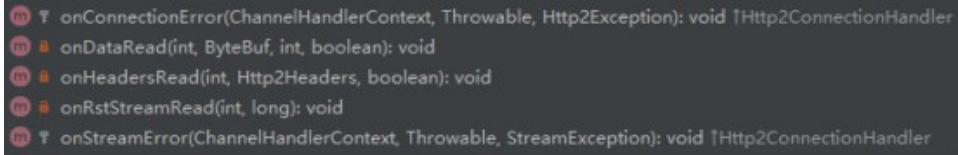
```

协议消息的接收：NettyClientHandler通过向Http2ConnectionDecoder注册FrameListener来监听RPC响应消息和协议指令消息，相关接口如下：

FrameListener回调NettyClientHandler的相关方法，实现协议消



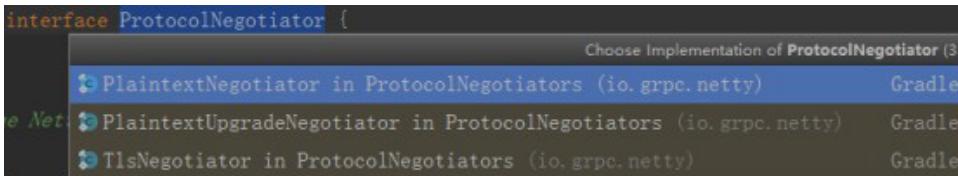
息的接收和处理：



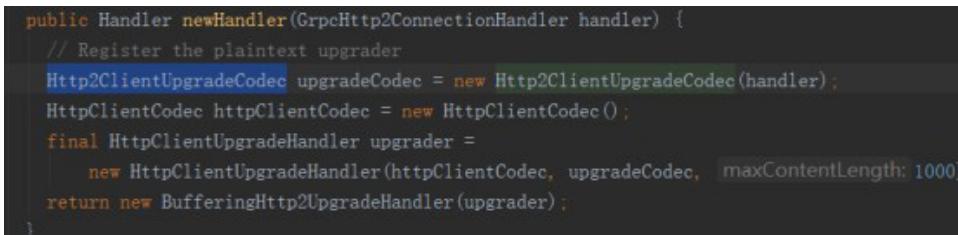
需要指出的是，NettyClientHandler 并没有实现所有的回调接口，对于需要特殊处理的几个方法进行了重载，例如 onDataRead 和 onHeadersRead。

2.3. ProtocolNegotiator功能和源码分析

ProtocolNegotiator 用于 HTTP/2 连接创建的协商，gRPC 支持三种策略并有三个实现子类：



gRPC 的 ProtocolNegotiator 实现类完全遵循 HTTP/2 相关规范，以 PlainTextUpgradeNegotiator 为例，通过设置 HttpClientUpgradeCodec，用于 101 协商和协议升级，相关代码如下所示：



2.4. LoadBalancer功能和源码分析

LoadBalancer 负责客户端负载均衡，它是个抽象类，gRPC 框架的使

用者可以通过继承的方式进行扩展。

gRPC 当前已经支持 PickFirstBalancer 和 RoundRobinLoadBalancer 两种负载均衡策略，未来不排除会提供更多的策略。

以 RoundRobinLoadBalancer 为例，它的工作原理如下：根据 PickSubchannelArgs 来选择一个 Subchannel：

```
RoundRobinLoadBalancerFactory Picker pickSubchannel()  
  
    @Override  
    public PickResult pickSubchannel(PickSubchannelArgs args) {  
        if (size > 0) {  
            return PickResult.withSubchannel(nextSubchannel());  
        }  
  
        if (status != null) {  
            return PickResult.withError(status);  
        }  
  
        return PickResult.withNoResult();  
    }  
}
```

再看下 Subchannel 的选择算法：

```
private Subchannel nextSubchannel() {  
    if (size == 0) {  
        throw new NoSuchElementException();  
    }  
    synchronized (this) {  
        Subchannel val = list.get(index);  
        index++;  
        if (index >= size) {  
            index = 0;  
        }  
        return val;  
    }  
}
```

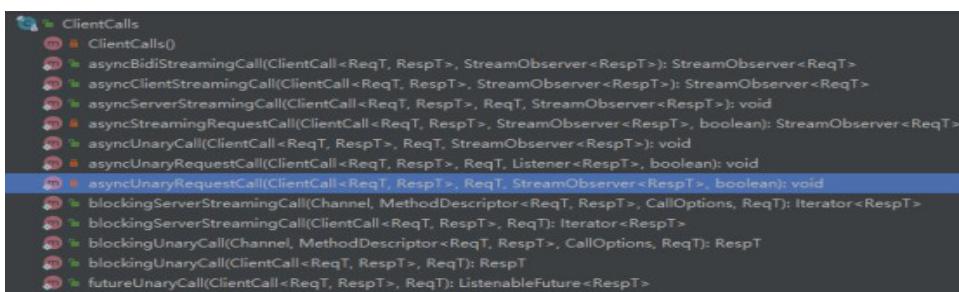
即通过顺序的方式从服务端列表中获取一个 Subchannel。

```
this(ManagedChannelBuilder. forTarget(zkAddr)
    . loadBalancerFactory(RoundRobinLoadBalancerFactory. getInstance())
    . nameResolverFactory(new ZkNameResolverProvider())
    . usePlaintext(true));
```

如果用户需要定制负载均衡策略，则可以在 RPC 调用时，使用如下代码：

2.5. ClientCalls功能和源码分析

ClientCalls 提供了各种 RPC 调用方式，包括同步、异步、Streaming



和 Unary 方式等，相关方法如下所示：

下面一起看下 RPC 请求消息的发送和应答接收相关代码。

2.5.1. RPC 请求调用源码分析

请求调用主要有两步：请求 Frame 构造和 Frame 发送，请求 Frame 构造代码如下所示：

```
ClientCallImpl sendMessage()
public void sendMessage(ReqT message) {
    Preconditions.checkNotNull(stream != null, "Not started");
    Preconditions.checkState(!cancelCalled, "call was cancelled");
    Preconditions.checkState(!halfCloseCalled, "call was half-closed");
    try {
        // TODO(notcall): Find out if messageIs needs to be closed.
        InputStream messageIs = method.streamRequest(message);
```

使用 PB 对请求消息做序列化，生成 InputStream，构造请求 Frame：

Frame 发送代码如下所示：

NettyClientHandler 接收 到 发送事件之后，调用 Http2ConnectionEncoder 将 Frame 写入 Netty HTTP/2 协议栈：

```
MessageFramer writeUncompressed()
    private int writeUncompressed(InputStream message, int messageLength) throws IOException {
        if (messageLength != -1) {
            statsTraceCtx.outboundWireSize(messageLength);
            return writeKnownLengthUncompressed(message, messageLength);
        }
        BufferChainOutputStream bufferChain = new BufferChainOutputStream();
        int written = writeToOutputStream(message, bufferChain);
        if (maxOutboundMessageSize >= 0 && written > maxOutboundMessageSize) {
            throw Status.INTERNAL
                .withDescription(
                    String.format("message too large %d > %d", written, maxOutboundMessageSize))
                .asRuntimeException();
        }
        writeBufferChain(bufferChain, compressed: false);
    }

NettyClientStream Sink writeFrame()
    public void writeFrame(WritableBuffer frame, boolean endOfStream, boolean flush) {
        ByteBuf bytebuf = frame == null ? EMPTY_BUFFER : ((NettyWritableBuffer) frame).bytebuf();
        final int numBytes = bytebuf.readableBytes();
        if (numBytes > 0) {
            // Add the bytes to outbound flow control.
            onSendingBytes(numBytes);
            writeQueue.enqueue(
                new SendGrpcFrameCommand(transportState(), bytebuf, endOfStream),
                channel.newPromise().addListener((ChannelFutureListener) future) -> {

```

2.5.2. RPC响应接收和处理源码分析

响应消息的接收入口是 NettyClientHandler，包括 HTTP/2 Header 和 HTTP/2 DATA Frame 两部分，代码如下：

```
NettyClientHandler sendGrpcFrame()
    /**
     * Sends the given GRPC frame for the stream.
     */
    private void sendGrpcFrame(ChannelHandlerContext ctx, SendGrpcFrameCommand cmd,
        ChannelPromise promise) {
        // Call the base class to write the HTTP/2 DATA frame.
        // Note: no need to flush since this is handled by the outbound flow controller.
        encoder0.writeData(ctx, cmd.streamId(), cmd.content(), padding: 0, cmd.endStream(), promise);
    }
```

如果参数 endStream 为 True，说明 Stream 已经结束，调用 transportTrailersReceived，通知 Listener close，代码如下所示：

```
AbstractClientStream2 TransportState transportReportStatus0
    // complete messages to deliver.
    if (stopDelivery || isDeframerStalled()) {
        deliveryStalledTask = null;
        closeListener(status, trailers);
    } else {
        deliveryStalledTask = (Runnable) () -> { closeListener(status, trailers); }
    }
}
```

读取到 HTTP/2 DATA Frame 之后，调用 MessageDeframer 的 deliver 对 Frame 进行解析，代码如下：

```
MessageDeframer deliver()
    inDelivery = true;
    try {
        // Process the uncompressed bytes.
        while (pendingDeliveries > 0 && readRequiredBytes()) {
            switch (state) {
                case HEADER:
                    processHeader();
                    break;
                case BODY:
                    // Read the body and deliver the message.
                    processBody();
            }
        }
    } catch (Exception e) {
        logger.error("Error reading message from stream.", e);
    }
}
```

将 Frame 转换成 InputStream 之后，通知 ClientStreamListenerImpl，调用 messageRead(final InputStream message)，将 InputStream 反序列化为响应对象，相关代码如下所示。

```
ClientCallImpl ClientStreamListenerImpl messageRead()
    public void messageRead(final InputStream message) {
        class MessageRead extends ContextRunnable {
            MessageRead() { super(context); }

            @Override
            public final void runInContext() {
                try {
                    if (closed) {
                        return;
                    }
                    try {
                        observer.onMessage(method.parseResponse(message));
                    } catch (Exception e) {
                        logger.error("Error reading message from stream.", e);
                    }
                } catch (Exception e) {
                    logger.error("Error reading message from stream.", e);
                }
            }
        }
        new MessageRead().start();
    }
}
```

当接收到 endOfStream 之后，通知 ClientStreamListenerImpl，调用它的 close 方法，如下所示。

```
ClientCallImpl ClientStreamListenerImpl close()
    private void close(Status status, Metadata trailers) {
        closed = true;
        cancelListenersShouldBeRemoved = true;
        try {
            closeObserver(observer, status, trailers);
        } finally {
            removeContextListenerAndCancelDeadlineFuture();
        }
    }
}
```

最终调用 UnaryStreamToFuture 的 onClose 方法，set 响应对象，唤醒阻塞的调用方线程，完成 RPC 调用，代码如下：

```
ClientCalls UnaryStreamToFuture onClose()
@Override
public void onClose(Status status, Metadata trailers) {
    if (status.isOk()) {
        if (value == null) {
            // No value received so mark the future as an error
            responseFuture.setException(
                Status.INTERNAL.withDescription("No value received for unary call")
                    .asRuntimeException(trailers));
        } else {
            responseFuture.set(value);
        }
    } else {
        responseFuture.setException(status.asRuntimeException(trailers));
    }
}
```

3. 作者简介

李林锋，华为软件平台开放实验室架构师，有多年 Java NIO、平台中间件、PaaS 平台、API 网关设计和开发经验。精通 Netty、Mina、分布式服务框架、云计算等，目前从事软件公司的 API 开放相关的架构和设计工作。

All Around AI

助力人工智能落地

由InfoQ中国主办的AiCon将重点关注人工智能的落地实践，与企业一起探寻AI的边界。在AiCon上，你将会看到来自创新工场、京东、360、腾讯、微软、知乎、饿了么、摩拜单车等国内外知名企业的[人工智能落地案例](#)，也能与国内顶尖的人工智能专家探讨相关的技术实践。

2018年1月13-14日 | 北京·国际会议中心

SPEAKERS

- 演讲嘉宾 -

刘海锋

京东商城首席架构师



颜水成

360人工智能研究院院长&首席科学家



尹大飞

摩拜单车首席数据科学家



6折

最低价限时购票，立减1440元

团购更优惠

大会官网：aicon.geekbang.org 购票咨询：18510377288 扫码关注大会官网，获取更多大会信息





本期主要内容：重磅开源 KSQL：用于 Apache Kafka 的流数据 SQL 引擎；洞悉流程！微服务与事件协同；Kafka 设计解析：流式计算的新贵 Kafka Stream；软件架构图的艺术；从金属巨人到深度学习，人工智能（极）简史；人工智能那些事儿；Google 研究人员提出在移动设备上运行神经网络的新技术



架构师特刊 大前端

本期主要内容：当我们在谈大前端的时候，我们谈的是什么；如何落地和管理一个“大前端”团队？



顶尖技术团队访谈录 第九季

本次的《中国顶尖技术团队访谈录》第八季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。



架构师特刊 用户画像实践

本电子书中几个作者介绍一个公司如何从无到有的搭建用户画像系统，以及其中的技术难点与实际操作中的注意事项，实为用户画像的实操精华之选，推荐各位收藏阅读。