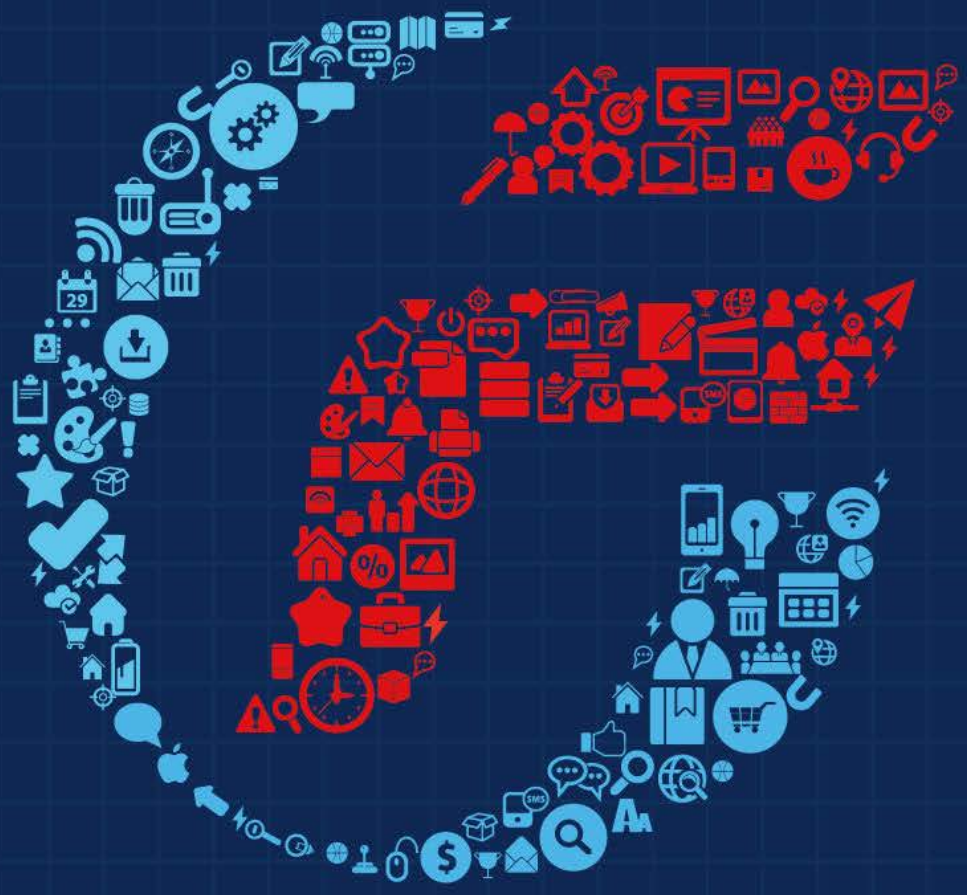
E-MAGAZINE

FinTech：新兴技术在广发证券的应用



www.infoq.com.cn

Geekbang 极客邦科技 | InfoQ

卷首语

经济学家许小年先生在最近的一次经济论坛上发表了一个十分有趣的演讲，他认为“互联网思维、共享经济和弯道超车是误导性概念”。确实，突破传统套路的、创新的、最重要是商业上合理而完全符合逻辑的思维，一直存在于商业世界，所谓“互联网思维”这种概念，也许只是传统保守人士看到互联网上一些科技公司的创新而“少见多怪”的炒作出来。同理，“互联网金融”恐怕也不是一个非常靠谱的概念，说的好像是个颠覆传统金融的新金融物种似的。互联网金融是新兴科技公司进军金融的一面大旗，它们提出来也许合理；对于传统金融机构而言，跟着呐喊则很可能是被带进沟里了。许先生说在金融领域“弯道超车”是会翻车的……就金融机构而言，科学的态度是重视金融科技、苦练技术内功、嫁接新型科技公司的文化基因（虽然很难）、以自主掌握的核心科技来支持业务创新。科技的“弯道超车”，则完全是可能的并且是屡见不鲜的。

金融科技的初心是提升效率。用什么技术工具生产金融产品、服务客户，直接决定生产效率。从某个角度讲，金融科技就是用先进的科技工具干金融吧。不掌握科技的后果，正如，呃……用铁铲跟用挖掘机的差别吧……

IT 虽然往往被视为是搬砖的，可是我们搬砖也讲科学，工欲善其事必先利其器嘛。在此精神引领下，我们也在容器化、大数据、微服务、函数类语言等领域把相关前沿技术应用到金融业务中。讲真，技术上我们自以为有点机会弯道超车华尔街同行的，因为我们起步晚啊，哈哈。有了这十年来丰富的突飞猛进的开源技术，很多传统华尔街巨头才能负担的起的深度基础技术研发，我们省了不少，让很多不可能变可能。而技术这种东西，是很容易在很短时间内变成“库存”和包袱的。相比已经背上不少历史遗留包袱的华尔街同行，我们技术上有“后发优势”……

感谢 InfoQ 提供了一个好的平台，我们希望通过开放、分享与外界交流，无论被拍砖或者引起讨论，团队均只能是从中获益。趁此机会作个小澄清，有些文章标题例如“交易系统是命根子”神马的，夸张哈。编辑出于吸引眼球的善意而改的，呵呵，出于严谨还是要得罪编辑再致歉一下。其实作为工程师，我们永远是悲观主义者，满眼总是系统的不完美和缺陷（所以也有干不完的活）。

在数字化程度较高的金融业，可以说“业务即代码”（或是“代码即业务”），反正一切业务逻辑以代码方式存在、企业生命线依赖于代码有效运作。更多的代码会以加速度替换活人岗位、填补手工环节、像潮水般涌入任何尚有效率问题的坑，风控被智能算法处理的比人更好、合规被机器学习与监管的更全面、交易清算则更连贯更自动更实时。金融科技是行业变革的核心驱动力，但能否让科技基因在一家企业内落地生根，则属于另一篇文章，另一个故事了……

广发证券副总经理 / 首席架构师 梁启鸿

目录

从黑天鹅事件说起
谈云计算对于金融系统的意义

梁启鸿

广发证券是一家本土的券商公司，于 2014 年 Docker 等容器技术尚未盛行之时开始投入容器化技术的研究，并于 2015 年开始大规模投入应用，成交量六百亿（2015 年）规模的金融电商平台、消息推送日均数千万条级别的社会化投顾问答平台以及日均流经交易量峰值近五十亿的交易总线均被容器化；投入生产的容器化云服务包括行情、资讯、消息推送、自选股、统一认证、实时事件处理，等等。

2016 年并开始基于 Docker、Kubernetes、Rancher 等技术研发运维机器人投顾（已投产）、极速交易系统、社会化 CRM，构建容器化的混合云解决方案。可能是为数不多**把容器化技术大规模用到“真金白银”的金融业务中的案例**。容器化技术帮助一家传统券商在云计算领域“弯道超车”。

在此分享本人在这两年里的在传统券商环境引入与推动容器化技术的思考历程，尝试陈述为什么云计算对证券业甚至整个金融业重要、而容器

化技术的出现又带来何种契机。

历史学家黄仁宇在其被认为是“大历史观”典范之作的《万历十五年》序言中提到两个观点，一是对一件历史事件的评价，要把时间坐标推到三四百年的范围，才能看清楚该事件的来龙去脉；另一是“可以数目字上管理”是区分社会现代化与非现代化的分水岭。

借这个角度看，我们现在的技术发展，是物理世界不可逆转的日益数字化虚拟化的过程，一切都为了更高效更“数目字可管理”，数字化进程也许可以追溯到电子计算机出现前并因技术革命而以指数级别加速推进，而云计算的出现，是这一进程里的关键事件之一，对于未来的 IT 系统技术与架构有着深远的影响。

在此之前，我们以物理世界的硬件（服务器与网络设备）承载着一个虚拟世界，这个物理载体很大程度上依然靠人肉运维（最多加上一些半自动化的工具，然而称不上“智能”）；在此之后，物理载体本身被数字化，虚拟机（Virtual Machine）、容器（Container）、软件定义网络（SDN），物理设施固然还在，但是它上面的软件栈（stack）却越来越关键。

数字世界像一个黑洞，它把一切线下的、非数字化的东西都席卷进来。连基础设施（infrastructure）都数字化的 IT 系统，所谓 Infrastructure As Code（基础设施即代码、可编程基础设施），一旦结合大数据、机器学习，智能化、自动化的运维逐渐成为可能；一个因应突发事件而弹性自伸缩、自愈、自适应的软件系统，“自己运维自己”的“无人值守”运维，相信到今天对于一部分 IT 人而言，并不是科幻电影。

开场白引用黄仁宇先生，乃因为个人深受其“大历史观”（macro history）之影响，倾向于认同事物在“历史上长期的合理性”，这对于金融业这个受高度监管的行业有特别寓意。 目前金融业大部分机构，未

能积极拥抱与利用云技术，原因之一恐怕是因为监管方面就“信息安全”尚有顾虑，原因之二是金融业相对传统之 IT，对云计算本身缺乏深刻认识。云计算是数字化进程里一个必然环节，被本身即已高度数字化的金融业以合理合规的方式采用是早晚的事情，“历史潮流”是无法阻挡的。

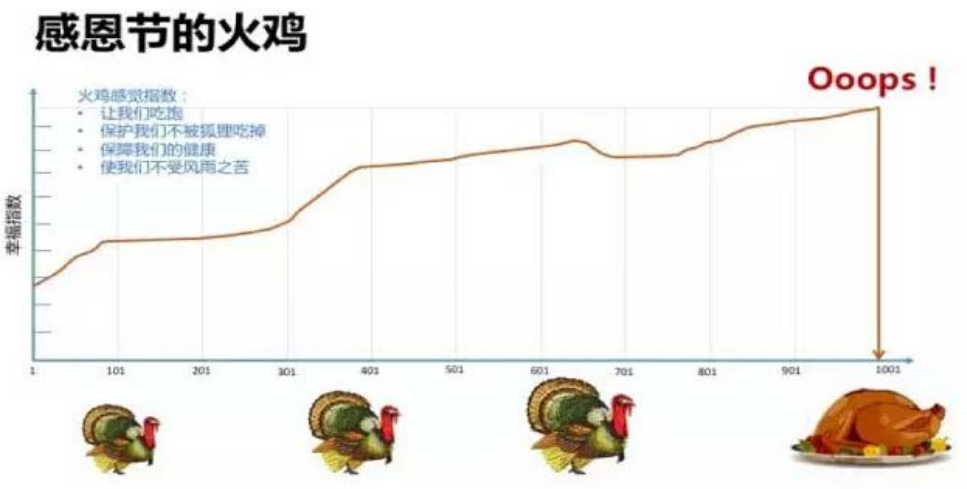
而我们作为高性能交易系统、互联网金融平台之研发者，对于云计算、尤其是容器化技术，有着特别的触觉。自从 2013 年 Docker 这种容器技术出现后，我们仿佛嗅到了些什么。在经过两年的研究与投产应用后，我们反省采用云技术的动机，总结得益好处、落实手段、技术关键，以飨同好。

从《黑天鹅》说起

我们研究和应用云技术的动机，来源于对“黑天鹅”事件的应对。“黑天鹅”这一概念，是在美国学者、风险分析师、前量化交易员、前对冲基金经理塔勒布（Nassim Nicholas Taleb）的《黑天鹅》（The Black Swan: The impact of the highly improbable）一书发表后在全球被得以高度认知。在发现澳大利亚之前，17 世纪之前的欧洲人认为天鹅都是白色的。但随着第一只黑天鹅的出现，这个不可动摇的信念崩溃了。黑天鹅的存在寓意着不可预测的重大稀有事件，它在意料之外并且后果非常严重。

一个黑天鹅事件，具有这三个特点：（1）稀缺、通常史无前例（rarity），（2）影响很极端（extreme impact），（3）虽然它具有意外性，但人的本性促使我们在事后为它的发生找到理由——“事后诸葛亮”，并且或多或少认为它是可解释和可预测的（introspective）。IT 系统、尤其是资本市场里的交易系统，所发生的各种重大问题，其实是很符合黑天鹅事件的特点的。

塔勒布用“感恩节的火鸡”很形象的解释了黑天鹅的概念——直到被宰掉成为感恩节火鸡晚餐前的每一天，火鸡都应该是活的很不错的，它的一生里没有任何过去的经验供它预测到自己未来的结果，而后果是致命的。



一套复杂的 IT 系统，很有可能就是那只火鸡，例如就个人近年所遭遇的类似事件最典型的两次，一是与某机构对接的技术接口，据称已经存在并稳定使用近 10 年——虽然技术古老但是从未出现问题，然而在过去两年持续创新高的交易量压力之下，问题终究以最无法想象到的方式出现并形成系统性风险（因为对接者不仅一两家）；另一，则是老旧的系统因对市场可交易股票数目作了假设（而从未被发现），某天新股上市数量超过一定值而导致部分交易功能无法正常进行。

这两个例子都符合黑天鹅特征，一是“史无前例” （如果以前发生过，问题早就被处理了），二是可以“事后诸葛亮”（所有 IT 系统问题，最后不都可以归结为“一个愚蠢的 bug” ？因为开发时需求不清楚、因为开发者粗心、因为技术系统所处的生态环境已经发生变化导致原假设无效……），三是“后果严重”（如果技术系统本身是一个广被采购的第三方的商业软件，则整个行业都有受灾可能；如果是自研发的技术，则最起

码对交易投资者造成灾难性损失）。

事实上，资本市场乃至金融业整体，可能都是黑天鹅最爱光顾的地方。甚至连普罗大众都听过的例子诸如：2010 年 5 月 6 日的 Flash Crash 在三十分钟内道琼斯指数狂泻近千点、1987 年 10 月 19 日的 Black Monday、国内著名的“乌龙指”事件导致的市场剧动，不一而足。

导致黑天鹅降临的原因，事后分析五花八门，可能是量化交易导致的、可能是市场流动性不足引起的、也可能是市场心理（例如恐慌抛售）触发的。无论何者，IT 系统几乎都是最后被压垮的那只骆驼。正如塔勒布文章中提到，高盛在 2007 年 8 月的某天突然经历的为平常 24 倍的交易量，

如果到了 29 倍，系统是否就已经坍塌了？

事实上，在这个日益数字化的世界，本身就高度数字化的证券市场，面临的黑天鹅事件会越来越多，出于但不仅限于以下一些因素：

- 增长的交易规模；
- 更高频、更复杂的交易算法（《高频交易员》一书里指出，股票市场已经变成机器人之间的战争）；
- 更全球化更加波动，海内外政治经济情况引起的突发变化。
- 更快速更先进的技术，已经出现数百纳秒内完成交易处理的专门性硬件芯片，快到人类根本无法响应。

据一篇科技论文（Financial black swans driven by ultrafast machine ecology） 的数据，人类国际象棋大师对棋盘上局势危机的判断大概需要 650 毫秒，而日常人类活动中通常的反应起码是秒级的；但是在一个高频交易的世界里，一笔交易可能在极速硬件的支持下只需要万分之几毫秒完成。人类，已经无法轻易掌控自己的交易算法在极速之下带来的问题、更无法了解自己的算法和他人的算法在交易市场上相互作用的集合带来的后果、甚至无法预测突发性政治经济事件对自身算法、技术、系统会触发何种反应。

数字世界，尤其是金融业的数字世界，正好是塔勒布笔下所谓的“极端斯坦”（Extremistan），它完全不受物理世界的规律影响，一切极端皆有可能。例如在物理世界常识告诉我们，一个数百斤的超级胖子的体重加到 1000 人里面比重依然是可以忽略不计的；但在金融世界，一个比尔盖茨级别的富豪的财产数字，富可敌国。

金融 IT，正好生存在这么一个“极端斯坦”——这里复杂系统内部充满难以察觉的相互依赖关系和非线性关系，这里概率分布、统计学的“预

测”往往不再生效。塔勒布称之为“第四象限”，我们，作为证券交易的IT，刚好在这个象限里谋生。

上述这一切，和云计算有什么关系呢？ 我们觉得非常紧密，逻辑如下：

- 世界越来越数字化、更加“数目字可管理”，一切效率更高。
- 本来就数字化的金融世界，日益是个“极端斯坦”，只能更快、更复杂，面临更多黑天鹅事件。
- 应对数字世界的黑天鹅，只能用数字世界的手段（而不是“人肉”手工方法），就像《黑客帝国》，你必须进入Matrix，用其中的武器和手段，去解决里面的问题（并影响外面）。
- 云计算，不过是世界数字化进程里的一步，把承载数字世界的物理载体也进一步数字化，但是它刚好是我们应对数字黑天鹅的基本工具，运算资源本身也是“数目字可管理”，并且正因为如此而可以是自动的和智能的。

即便到了今天，相信很多企业、机构的机房里的运算资源，依然不是

“数目字可管理”，这本身真是一个讽刺。但直到云技术出现，才解决这个问题。结合云计算的技术，交易系统不再是“your grandmother’s trading system”。

“反脆弱”的技术系统

黑天鹅事件是不可预测的，但是并非不可应对。《黑天鹅》的作者塔勒布，在其另一本有巨大影响力的著作《反脆弱》（Anti-Fragile）里，提到了如何在不确定中获益。这本闪烁着智慧之光的著作，早已超越了金融而进入到政治、经济、宗教、社会学的思考范畴，对 IT 系统技术架构的设计，同样具有启发意义。想想，一个经常被黑天鹅事件光顾的交易系统，如果不仅没有坍塌、还随着每一次的考验而技术上变的越来越周全和强壮，这对于任何开发工程师、运维工程师来说，是不是一个梦想成真？

实际上，这个过程对于任何 IT 工程师而言都是非常熟悉的，因为我们中很多人每天的工作，可能就是在不断的以各种应急手段紧急救援不堪重负的生产系统、或者在线弥补技术缺陷，在这过程中我们发现一个又一个在开发和测试时没有发现的问题、一次又一次推翻自己在开发时的各种假设、不断解决所遭遇到的此前完全没有想象过的场景。如果项目、系统活下来了，显然它变得更加健壮强韧。

只不过，这一切是被动的、低效的、“人肉”的，而且视系统架构和技术而定，变强韧有时是相对容易的、有时则是不可能的。正如一艘结构设计有严重缺陷的船，打更多的补丁也总会遇到更大的浪把它打沉。

如果基于《反脆弱》的三元论，也许大部分 IT 系统大致上可以这么看：

- 脆弱类：绝大部分企业IT系统，依赖于大量技术假设与条件，不喜欢无序和不稳定环境，暴露于负面“黑天鹅”中。

- 强韧类：小部分大规模分布式系统（也许通常是互联网应用），适应互联网相对不可控的环境（如网络延迟与稳定性、客户端设备水平和浏览器版本、用户量及并发请求变化），经受过海量用户与服务请求的磨练，相对健壮。
- 反脆弱类：能捕捉到正面“黑天鹅”系统不仅在冲击中存活，并且变的更加强韧，甚至在这过程中获益。

这里所谓的“脆弱”，并不是指系统不可靠、单薄、技术不堪一击，而是指这类系统厌恶变化、厌恶不稳定不可控环境、本身架设在基于各种稳定性假设前提的精巧设计上，无法对抗突如其来的、此前无法循证的事件（黑天鹅），更无法从中自适应和壮大。就这个角度看，证券行业甚至整个金融业里，大部分的系统可能都是脆弱系统。传统 IT 系统有以下一些常见的技术特点，例如：

- 一切以关系型数据库为中心（RDBMS-centric）。
- 很多历史遗留系统（legacy system）有数以百计的表、数以千计的存储过程。
- 业务逻辑高度依赖数据库。
- 中间层与数据层高度紧耦合。
- 多层架构（multi-tiered architecture），层与层之间依赖于高度的约定假设（协议、接口、数据格式等），并且这些约定经常来不及同步（例如某个团队改变了维护的接口而没有通知其他团队、或者数据库的表结构改变了但是中间层的对象库因为疏忽而没有及时步调一致的重构），有些约定甚至只存在于协作的开发者脑海中而没有形成文档（即便形成文档也经常因需求变化频繁而无法及时更新）。

- 应用程序依赖于某些第三方的代码库，而这些代码库很有可能依赖于某个版本的操作系统及补丁包，并且这种依赖关系是传递的 – 例如某个第三方代码库依赖于另一个第三方代码库而该库依赖于某个版本的操作系统……
- 系统设计，往往没有考虑足够的失败场景（因此可能完全没有容错机制），没有考虑例如不稳定网络延迟对业务逻辑的影响（例如大部分企业系统都假设了一个稳定的LAN）。
- 组件、模块、代码库、操作系统、应用程序、运维工具各版本之间具有各种线性、非线性依赖关系，形成一个巨大的复杂系统。

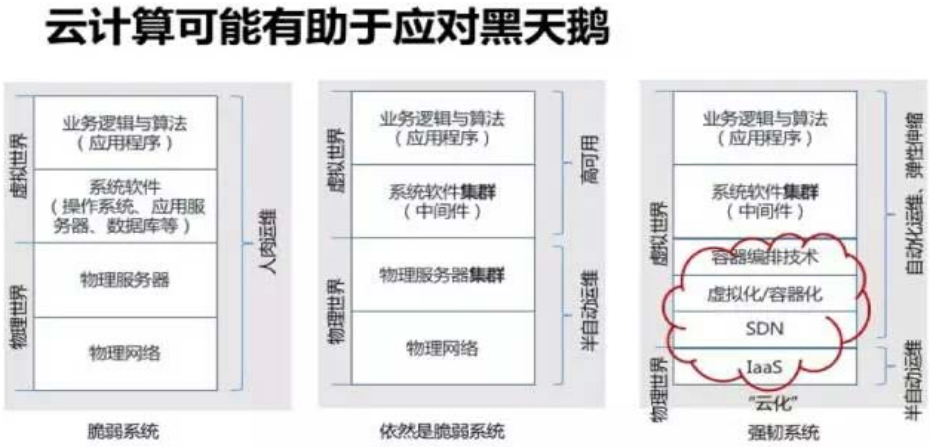
然而，以下这些变化是任何 IT 系统所不喜欢却无法回避的，例如：

- 多层架构里，任何一个环节的约定独立发生细微改变，必定导致系统出错（只是严重性大小的差别），这几乎无法很好的避免 – 研发团队的素质不够高、软件工程的水平低、瞬息万变的市场导致的频繁更改等等，总是客观存在。
- 因为安全原因，需要对操作系统进行打补丁或者升级，导致应用程序所依赖的代码库发生兼容性问题，在打补丁或升级后通过测试及时发现兼容问题已经算是幸运的，最怕是在生产环境运行过程中才触发非线性关系的模块中的隐患。
- 跨系统（尤其是不同团队、部门、组织负责的系统）的调用协议与接口发生变化，是一个常态性的客观事实。
- 互联网环境、甚至企业内部的网络环境，并不是一成不变的，网络拓扑出于安全、合规隔离、性能优化而变化，可能导致延迟、吞吐等性能指标的变化，应用系统本来没有出现的一些问题，有可能因为运行环境的变化而浮现，而系统内部容错机制往往没有考虑这些问题。

- 业务需求永远在变，以数据库为中心的系统，不可避免产生表结构（schema）调整，系统升级需要做数据迁移，而这总是有风险的（例如data integrity需要保证万无一失）。
- 于是，传统IT对于这些系统的运维，最佳实践往往不得不这样：
- 在使用压力增大的情况下，最安全的升级手段是停机、换机器、加CPU、加内存，直到硬件升级、垂直扩容（vertical scale、or scale-up）手段用光。
- 维护一个庞大的运维团队，随时救火。
- 试图通过软件工程的管理，例如制定规章制度，让协作人员、团队之间在接口升级前走流程、互相通知，来避免随意的系统变化导致的风险。
- 加大测试力度，通常很有可能是投入更多的人肉测试资源，以保证较高的测试覆盖率和回归测试（regression test）能力。
- 强调“纪律”，以牺牲效率为代价，通过“流程”、“审核”设置重重关卡以达到“维稳”效果。
- 重度隔离运维与研发，禁止研发人员触碰生产环境，减少误操作，例如随意升级操作系统、对应用逻辑抱着侥幸心理打补丁，等等。

不可否定，这些“套路”在以往的时代可能是最佳实践，也体现了一个 IT 组织的管理水平。但是毫无疑问，这样研发、运维和管理的系统，是一个典型的“脆弱系统”，它依赖于很多的技术、工具、环境、流程、纪律、管理制度、组织结构，任何一个环节出现问题，都可能导致轻重不一的各种问题。最重要一点，这样的系统，厌恶变化、喜好稳定，无法在一个“只有变化才是唯一不变”（并且是变化越来越频繁）的世界里强韧存活，更无所谓拥抱变化而生长。

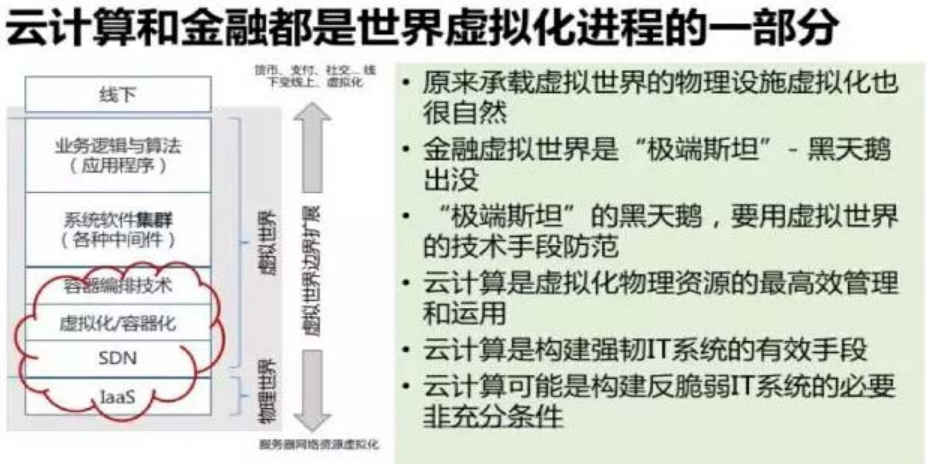
强韧类的技术系统，情况要好的多，起码能“响应”变化（如后文所论述）。但是注意，在塔勒布的定义里，“强韧”并非“脆弱”的反面，“强韧系统”只是能相对健壮的对抗更大的压力、更苛刻的环境，它并不能从变化、不确定中获益。“脆弱”的反面，塔勒布在现有语言里找不到一个合适的词语，所以他发明了一个新概念，“反脆弱”。问题是，接受“变化是一种常态”、拥抱变化并从中获益的“反脆弱”的技术系统，能被构建出来吗？



云计算的出现，有利于帮助 IT 构建强韧系统，并且让“反脆弱”系统成为可能。其最根本原因在于，云计算本身是机房物理设施数字化的过程，如上文所述，数字世界的黑天鹅——微秒、纳秒内发生的极端事件，只能通过数字化手段才能高效解决。伴随云计算出现的是 DCOS（Data Center Operating System）、APM（Application Performance Monitoring）、Infrastructure As Code（基础设施即代码、可编程运维、可编程基础设施）、DevOps 等技术方案、技术产品、技术理念和方法论。这些都是构建强韧系统的有力武器，而在云计算时代之前，它们严格意义上不曾存在过。

到此为止，本文想立论的，是云计算相关技术的出现，对于金融类尤

其是交易系统意义重大，技术架构必须调整以利用之，对于构建强韧的、甚至潜在有“反脆弱”能力的系统，有极大帮助。云技术、尤其是容器化技术出现后，金融软件系统的研发与运维面貌将被极大的改变。



云计算也许是目前为止对于证券交易系统、甚至对于更广义的金融技术系统而言最适合应对黑天鹅的技术手段。监管机构不应该见到“云”字就敏感的与“公有云”、信息安全、交易可监管性等问题联系起来；金融机构则需要与时俱进的学习掌握“云化”的技术手段、架构思维。至于系统是运行在公有云、私有云还是混合云，都已经是另一个故事。

作者介绍

梁启鸿，哥伦比亚大学计算机科学系毕业，出道于纽约 IBM T.J. Watson 研究院，后投身华尔街，分别在纽约 Morgan Stanley、Merrill Lynch 和 JP Morgan 等投行参与交易系统研发。本世纪初加入 IT 界，在 Sun Microsystems 大中华区专业服务部负责金融行业技术解决方案。此后创建游戏公司并担任 CTO 职位 5 年。后作为雅虎 Senior Principal Architect 加入雅虎担任北京研究院首席架构师角色。

三年前开始厌倦了框架、纯技术的研发，开始寻找互联网前沿技术与

线下世界、传统行业的结合；目前回归金融业负责前端技术、大数据、云计算在互联网金融、股票交易系统的应用。

个人兴趣是把前沿的互联网技术应用到垂直行业中，做一点能改变传统面貌的、最重要是有趣好玩又有用的事情；紧跟 Go、Docker、Node.js、AngularJS 这些技术但更关注如何把技术用到应用场景里，从中获得乐趣。

交易系统是命根子
为什么广发证券要将它容器化

梁启鸿

云计算，其中一个最基本目的，是计算资源的集合管理与使用。其实IT 界在计算资源的集合使用方面，过去 20 年起码尝试过三次：

- 90年代中后期，出现网格计算（Grid Computing），用于蛋白质折叠、金融模型、地震模拟、气候模型方面的应用。
- 本世纪初，出现效用计算（Utility Computing）- 推行运算资源按需调用（On-demand）、效用计费（Utility billing）、订户模式（Subscription model）的概念。IBM、HP、Sun这些IT公司都尝试推动过这个领域的解决方案，虽然成效不彰。
- 10年前，亚马逊AWS发布，算是云计算的标志性事件。云计算覆盖含义更广，网格计算可以是云计算平台上的一类应用，而效用计算则可以被视为云计算服务商所采用的商业模式

华尔街可以算是网格计算商业化产品的积极使用者。不少投行采用诸

如Gigaspaces、Gemstone、GridGain、Terracotta、Oracle Coherence（原Tangosol）这些商业技术开发它们的交易系统，实现低延迟、高并发并且支持事务的技术架构。这些技术一般来说比较“重”，有复杂的中间件、使用者需要去适应一整套的开发理念和掌握专门的框架、编程模型“入侵性”比较强（intrusive），开发者与应用架构均被迫需要去适应它们。

云计算和大数据技术出现后，这些技术逐渐变身“事务型内存数据库”、“内存网格”（in-memory data-grid）、“流式计算平台”（stream processing）等等，然总体来说变的越来越小众。无论如何，这些技术在云计算出现前**已经帮助华尔街机构掌握了计算资源集合运用、分布式架构的一些理念和思维**。而国内证券界甚至金融业IT总体来说，对这类技术是相当陌生的。

就对标华尔街同行的证券业IT而言，可以说基本上错过了上述计算资源集合应用的三个浪潮。云计算兴起后，OpenStack之类的技术在证券IT甚至金融界的成功落地、大规模采用的案例极其罕有（如果有的话）。即便是近年来互联网云服务商兴起，部分金融机构因为试水互联网金融而开始使用公有云，很大程度上使用方式也不过是使用了一些虚拟机运行一些互联网边缘（Edge）服务。

然而这个时代有趣的地方在于，一些新技术的出现可以让“弯道超车”、“后发制人”成为可能。上一个阶段错过了一些东西，但是也可能下一个阶段少了很多历史包袱从而可以轻易跳进最新的技术世代。假如你能把握的话，容器——恰好就是这么一种技术。

在英语里，“容器”、“集装箱”、“货柜”都是同一个字 ——Container。容器技术之于软件业，很有可能可以类比集装箱对运输业的巨大影响。

实际上，确实有人想过把整个机房放在集装箱里，例如 2006 年 Sun Microsystems 推出的 Project Blackbox 刚好是亚马逊发布 AWS 的同一年。但这个集装箱是钢铁的、有物理形体的、重量以吨为单位的；而其中的内容，自然也是各种物理的服务器、网络设备、发电机。那一年，可能谁也没有想到，10 年后有一种“数字化”的虚拟集装箱大行其道。

技术界不乏认为**容器对软件技术产生革命性影响的观点**，本人倾向于认同这种观点，因为：

- 容器影响开发者的开发方式、开发习惯，“强迫”他们去思考例如无状态的服务、业务逻辑粒度的控制、资源的弹性伸缩、应用代码的发布形态、系统里面每一个细节的可监控性，等等。
- 让真正的DevOps成为可能。自动化测试、持续集成（CI）、持续交付（CD）、自动化部署、无人值守的运维… 开发与运维的角色差异进一步缩小，而效率则最大程度提升。
- 让“不可变基础设施”（Immutable infrastructure）成为可能。这将颠覆传统软件系统的升级发布和维护方式。

关于容器与虚拟机的区别，不在本文展开论述，业界有足够多的技术文章供参考。在此仅陈述一下，为什么作为金融系统、交易平台的研发者，我们挑容器出现的时候跳进云计算，而在更早阶段虚拟机系列相关技术成熟时却并没有大的投入。

最根本原因在于，作为研发组织，我们从研发视角，基于具体应用场景去持续、积极寻觅能解决强韧性、健壮性问题的解决方案，这是一种“Top-Down Thinking”，例如在交易量波动过程中，我们能否有自动化的技术去可靠的应对、实现计算资源的弹性伸缩并且保持超级高效？有什么技术可以帮助我们解决复杂交易系统发布、升级、打补丁的危险和

痛苦？交易故障出现时系统内部防止雪崩效应保持快速响应的“熔断”（circuit-breaker、fail-fast）机制有什么更规范的做法？用现有云计算的理论和技术倒着去套，显然是无解的，因为：

- 虚拟机级别的资源调度，太沉重，可编程接口太弱，无法“融入”到一个高性能运算（HPC: High Performance Computing）的应用中。
- 仅仅为了资源的弹性伸缩，去引入一个第三方解决方案，例如一个PaaS（诸如CloudFoundry、OpenShift等），对于交易系统而言，代价太大、可控性太低、编程模型受入侵程度太高。
- 不是为了云计算而云计算，一切“革新”需要合理、充分的应用理由，场景不符合，架构就不合理。

“Bottom-up Thinking”，即从基础设施的可管理、资源充分共享、运维更高效等角度去看问题，是一个典型的“运维”视角或者所谓CIO的视角，这种思考，关注的是TCO(Total Cost of Ownership)成本的降低(IT在垂直行业一直被认为是一个成本中心，在现在这个时代这是一个错误观点，但这是另一篇文章的讨论范围了）；可是和核心业务应用非常脱节。

对于一个传统行业尤其是受监管行业的IT而言，**技术氛围往往是保守和审慎的**，采用云技术这种在互联网界以外还很大程度被认为是“新生事物”的东西，并不是一件容易的事情：在行业繁荣、企业赚钱的时候，公司很可能并不关注运维除了稳定以外的事情，包括用虚拟机还是用物理机、能否节省一点成本等等；在市场不景气、公司亏钱的时候，IT却又需要以非常充分的数据来证明建立或者采用云平台能带来显著的大幅的成本节省效果，但是这往往首先涉及第三方软件的采购、机房的改造… 这本身就是一个巨大障碍。

所以，悲观一点的看，**很大一部分传统行业 IT，很可能需要等到云技术成熟为新世代 IT 系统的标配**，就像 mainframe 时代向 client-server 时代转变、client-server 时代向多层架构 /Web 技术时代切换，云技术成为一个主流的、企业决策者不再需要加以思索的事物（no-brainer）时，才会顺利进入企业世界。此时，运维的视角也许才能被充分接受。但是，对于以创新为本业的金融科技，那已经太迟。

早期的云技术，从运维视角去看是自然的，因为虚拟化技术最开始是把基础设施数字化的一个进程。容器化技术的出现，改变了这一切。容器天然与应用服务结合的更紧密、天然需要程序员的深度介入，**“上帝的归上帝，凯撒的归凯撒”**，容器里面的归程序猿（code monkey），容器外面的归“运维狗”（watchdog）不一定对，只是作为笑话简单粗暴的类比一下，但想说明的是容器内外的关注点是不一样的、而运维与研发的协同则是深度的。

在 Docker 刚出现的时候，一直带着前述问题的我们，从其理念已经体会到容器技术对构建一个强韧交易系统的好处。

交易系统，无论是股票交易、债券交易、期货交易、外汇交易还是多资产投资交易，本质上是一种非常专业的软件系统。在华尔街，它很有可能是某家投行作为自身的核心技术进行研发的；在国内，由于绝大部分券商缺乏开发能力，它更有可能是由某些第三方厂商作为商业软件的半成品进行维护和订制的。但是无论国内外，其作为软件系统都缺乏商业软件（COTS，Commercial Off-the-Shelf）的“专业性”，例如：

- 和Oracle、IBM等的数据库及中间件套件比对：通常缺乏安装工具。重新部署一套，需要一帮熟悉该交易系统的工程师，费九牛二虎之力去部署。使用完也不敢、不舍得轻易舍弃。即便是后来虚拟

机出现，让安装工作可以部分重用，依然是一个高难度技术活，不是张三李四可以在生产环境（通常多个）、测试环境（起码两位数字个数）、开发环境（N个）可以随意低成本复制的。

- 和Redhat、Windows比对：通常没有补丁工具，任何补丁都需要手工、半手工的去打。在Docker出现之前，我们一直在想，为什么交易系统不可以实现一个类似apt-get或者yum的工具，能列出已打的补丁、能获取最新的补丁包、甚至能管理和删除一些补丁包？
- 和例如WebLogic JEE应用服务器以及很多商业中间件相比：通常交易系统里非常复杂的系统级的底层配置（非业务参数设定）只能手工进行，因为开发者（包括开发商）并不愿意投入时间资源去开发提供非常专业的可视化配置工具。尤其是很多交易系统是开发商从小系统逐步摸索开发出来的，很多东西并无考虑，也没有把多年累积的经验教训抽象好，经常只能手工“优化”。
- 和领先的互联网技术平台比：金融业的应用（不仅限于交易系统），可能都没有好办法实现滚动发布、灰度发布、在线升级、多版本在生产环境并存、回滚，一个以关系型数据库为主导的系统，是很难做到这些的。

结论就是，传统交易软件缺乏专业商业软件的“专业性”。开发商通常缺乏大型软件公司（如 Oracle）的软件工程能力、缺乏“packaged solution”的思维、缺乏对其客户在运维方面遭遇到的实时挑战的认识（想象一下日交易量达到十几亿手时的券商机房和里面的“救火”团队）。显然，这样的软件系统，是一个“脆弱系统”，是无法应对黑天鹅挑战的。

容器技术的到来，让我们在观念上“弯道超车”：

- 在软件安装、发布方面，我们还需要去学习、模仿Oracle、IBM们

研发什么工具吗？不需要了。采用容器、容器编排技术、容器镜像管理技术、容器目录（catalog），我们轻易的“一键发布”，而且谁都可以执行。

- 在打补丁方面，我们还要去参考Redhat开发个升级工具吗？没必要了，因为我们不打补丁，已经发布的容器里内容永远不会改变（Immutable），需要修复缺陷或者升级版本，我们就发布新容器，发布组件乃至整个全新交易系统，成本都是很低的（并且必须永远维持那么低）。因此，我们也不一定需要什么中心化的、统一的、集中的可视化配置管理工具。
- 灰度发布、在线升级、多版本生产环境、回滚现在都是容器化系统天然自带的，因为整个系统的发布、部署是低成本的和快速高效的，只要有硬件资源就再发布一套，不行就切换回旧的那套，技术界已经在不断总结最佳实践，总有一款套路适合你。基于容器的微服务，天然支持多服务版本并存；对于回滚，数据库可能是个问题，但首先一切以数据库为中心就是个很有可能是错误的观念（当然，视应用场景而定），在分布式架构里，关系型数据库有可能不在系统关键路径上、事务有好些办法可以规避、通过很好的面向对象设计解耦内存与持久层的耦合。实际上，经验告诉我们，关系型数据库被滥用是企业应用软件的通病。

而比解决运维问题更有趣的，是这几方面：

- Infrastructure As Code 现在总算不是口号了，对着容器、容器编排技术进行编码，让“无人值守”、“智能运维”真正成为可能。虽然Puppet、Chef、Ansible这些技术早就存在，但是它们基本上仅限于操作基础设施的物理、虚拟资源，与一个专业应用系统

通过API深度结合然后基于业务场景、系统业务指标来自我维护，是非常困难的。而容器，基本上只是一个进程，通过其API可以轻易深度整合到交易系统里。

- 容器技术出现后，也衍生出更多其他新兴技术，例如CoreOS、RancherOS、Unikernel。对于无止境追求极速性能的交易系统，交易软件到底层硬件之间的耗损越少越好，这和我们所遵循的mechanical sympathy技术理念完全一致。类似Unikernel这样的所谓“library OS”非常有趣，因为整个操作系统萎缩成一系列的基础库，由应用软件直接调用以便运行在裸机（bare metal）上。想象一个超级轻量的交易中间件，无缝（真正意义上）运行在裸机上的毫无羁绊裸奔的“爽”……
- 在广发证券IT而言的所谓“弯道超车”，其实更包括几方面的含义：一是观念上的，颠覆传统软件研发的思维，不必要去做追随者；二是技术成长方面的。虽然有些技术如Unikernel并未成熟到可以真正运用，但是两三年前的Docker不也一样吗？重要的是团队和一个革命性技术的共同成长，前瞻性技术的研发投入终将回报（capitalize）；三是实际效果上的，虽然在云计算发力的前几年，我们并未投入到IaaS、PaaS的“基建”，但是从容器化入手，忽然间就获得了一定的“计算资源集合使用”的能力，此前只有技术实力较强的科技企业能做Grid Computing。

在证券业，我们不是孤独者。华尔街投行的表率高盛，今年 2 月份宣布了他们一年内把 90% 的运算能力容器化的计划。

时至今天，在 Docker 已经被团队广为接受、被应用到各种业务系统中去的时候，我们从两个方向同时继续推进容器化技术：

- 第一个方向，继续沿用上文所述的“Top-Down”思路，基于业务场景、具体应用需要，把容器（Docker）、容器编排管理（Rancher）的技术深度整合到交易系统中去，让其获得自伸缩（elastic）、自监控（self monitor）、自修复（self-healing）的自动化能力，这是“反脆弱”系统的一个必要非充分条件，至于什么时候伸缩、什么时候自修复、如何学习自己的运行行为以作自我调整、如何把低级的系统指标换算成业务级别的性能指标作为自己的健康“血压计”，这些是需持续学习挖掘、测试验证的东西。
- 第二个方向，是上文所述的“Bottom-Up”思路，采用例如Kubernetes建立起一个多租户的CaaS（容器即服务）平台，支持非交易的应用系统（例如电商平台、机器人投顾服务、CRM等等）的跨云（公有云、私有云、传统机房）容器化。

第一个方向，我们认为是真正的证券领域“云计算”应用场景，符合所谓的“计算”这个概念，它要求研发团队对容器技术深刻理解，把一个专业的、垂直领域的技术解决方案“云化”。第二个方向，是使用“云平台”的场景，它只要求应用开发者遵循预定义的标准、最佳实践去开发微服务，应用系统部署到多租户的 CaaS 平台上，即起码理论上获得云服务基本属性和好处。前者把云的技术“内置”于自己的垂直技术平台以服务客户，后者使用成熟通用的云平台（自己的或者第三方的）。


THE WALL STREET JOURNAL.

Home World U.S. Politics Economy Business Tech Markets Opinion Arts Life Real Estate

Big Changes in Goldman's Software Emerge From Small Containers

By STEVEN ROSENBUSH AND STEVEN NORTON

Feb 24, 2016 5:59 pm ET

1 COMMENTS

CONTENT FROM OUR SPONSOR

Deloitte.

MIT Media Lab Director Joi on Innovation

Big companies should learn to capitalize the low cost of innovation. Small factor China are subverting traditional Americ manufacturing. And blockchain startup already be experiencing their bubble moments. MIT Media Lab Director Joi shares these insights and more on innovation.

Read more about: innovation

Don Duet, co-head of Goldman Sachs's technology division, plans to shift about 90% of the company's computing to containers. GOLDMAN SACHS GROUP INC.

More from Del

软件工程没有银子弹

99年图灵奖获得者Fred Brooks（也就是Brooks' law的发现者："往一个已经延误的项目里加人力资源，只能让那个项目更延误"）说过，软件工程没有银子弹。在各种标榜"云"的科技公司层出不穷的今天，其实依然还没有任何"黑科技"帮阁下把你的脆弱系统瞬间变成一个强韧系统甚至一个具备反脆弱能力的系统。所以，容器技术也不过是一个也许必要但不充分的有用工具而已。

能否释放云计算的威力，取决于很多因素。到一个公有云上使用几个虚拟机，也算一个小小的进步，毕竟它可能（1）缩短了一些传统企业采购硬件、机器上架等等的周期；（2）帮不擅长互联网技术的传统企业解决一部分"互联网最后一公里"问题。然而，这只不过是使用了一些虚拟化基础服务，如果部署在上面的系统本身是一个脆弱系统，那么它在云上依然是一个脆弱系统。

实现一个强韧甚至反脆弱的技术系统，你首先得有一个恰当的技术架构，而实现这样的技术架构，你首先需要有研发团队，不错，个人观点是，在现阶段如果不具备软件研发能力，那么你的组织其实无法真正利用、享

受到云技术带来的福利。

怎样的架构才能利用和释放云计算能力？受篇幅所限在此无法深入，业界有非常多好的文章，本文仅作高度概括的总结以陈述个人观点。

首先，架构设计需要遵循 Reactive（响应式）原则（根据“响应式宣言”）：

- Elasticity，弹性响应系统负载变化，基于实时性能监控指标以便通过预测性（predictive）和响应性（reactive）算法来对系统进行扩容。
- Resiliency，通过复制（replication）、包容（containment）、隔离（isolation）和委托（delegation）等机制，保障在故障发生时系统能继续高度可用。
- Responsive，系统及时探测侦察问题并解决问题，保障对外的及时响应。
- Message-driven，采用消息驱动的、非堵塞的异步通讯机制，降低系统内部组件模块间的耦合度，提升吞吐量。

“Reactive”既是一种架构风格、也是一系列符合这种风格的技术与工具。过去二十多年来华尔街的交易系统，设计逻辑很大程度上是符合这个理念的，例如高性能交易系统都依赖于异步非阻塞的消息中间件（并且这类技术通常是某家投行独门的技术杀手锏）、都高度强调可用性可靠性，只不过无论工具、技术、基础设施、开发意识都不如今天开源软件世界的技术与软件工程理论来的完备、成体系。

Reactive 宣言里很有趣的一个字眼是 Resiliency，在英文里它是这么定义的：

它表示一种在“坏事情发生后变强壮、健康、成功的能力”、“因被拉扯、

拉伸、按押而变形后重新恢复原状的能力”。符合这一原则的技术系统，基本上已经符合塔勒布的“强韧”标准。但显然，Reactive 的技术理念，更是衡量变化、响应变化的，其宣言中提到的支持 scaling 的“predictive and reactive algorithms”，显然是系统自身应对变化而产生的“经验积累”，是一个自我学习和壮大的过程。能够采用响应式架构理念和响应式技术工具实现的系统，很有可能是一个“反脆弱”的系统，这正是我们所研发的交易技术所严格遵循的。

其次，在技术研发过程中，我们采用一系列的所谓“Cloud-native patterns”（原生，或者说“天然”的云计算架构模式）来实现我们的系统，以便让系统达到 Cloud-ready（具备云感知能力，借用 IBM 与 Intel 相关中文版论文的概念）。

“云感知”的一些要求，其实是常识性的，是一个合格的架构师在设计系统架构时本来就应该遵循的，例如：云感知的应用需要有“位置独立性”，应用程序应该动态发现服务，而不是通过硬编码固化依赖关系；“避免依赖于底层基础架构”，应用程序应该通过对操作系统、文件系统、数据库的抽象来避免对底层基础技术进行假设；“带宽感知”，API 和应用协议的设计需要考虑应对带宽问题和拥堵。

此外，“云感知”和上述“响应式”原则，是完全一致的，云感知也关注“故障恢复能力”、“延迟恢复能力”、“扩展的灵活性”。事实上，在一流的金融 IT 团队里，这些原则、实践要求，从来都是被遵循的，因为金融的应用系统天然需要这些能力。无论是否在云计算时代，这些原则、要求均非常合理。

原则归原则、口号毕竟是口号，具体的实施才是一个现实问题。在这个方面，我们崇尚“设计模式”（Design Patterns），尤其是架构方面

符合云计算环境的设计模式，Cloud-native Patterns，一系列由各方技术“大拿”共同总结归纳出来的“最佳实践”。学习和运用设计模式，本身并非“教条主义”，而是让一支也许平均从业年限只有 3-5 年的工程师团队更快的掌握整个技术界不断总结归纳的经验；更重要的是，一个个人、一支团队、一家公司所遭遇到的应用场景是有限的，架构模式是拓宽视野的最佳途径。有助于“云感知”的、符合响应原则的Cloud-native架构模式，在此罗列以下一些常见常用、易于理解的。

- Circuit-breaker：断路器，快速“熔断”应用系统里的某个链路，避免系统进入局部假死并造成请求堆积。这是交易系统常用的模式。
- Fail-fast：有些服务一旦出现问题，与其去耗费时间资源处理它，不如让它迅速失败（“自杀”），以避免调用者反复请求、等候，导致整个系统陷入假死。这也是交易系统为保障系统响应而常用的模式。借用塔勒布在《反脆弱》里的一句话，“what is fragile should break early, while it is still small”，“脆弱”的东西应该被尽早扼杀在襁褓里。
- CQRS（Command and Query Responsibility Segregation）：著名的“读写分离”，也许你一直在用，只是没有意识到。读写分离的目的，通常也是为了让频繁大量的查询获得较好的响应。
- Retry：重试（没想到吧？这也是一个模式），在一个服务请求或者网络资源请求失败后，一定程度下透明的、持续的重试请求，假设服务和资源的失败是暂时的。
- Throttling：流控，避免服务资源被个别的服务请求者的频繁请求消耗尽而无法再服务其他请求者，从而保障当前服务能在一个“服

务水平协议”（SLA）下向所有使用者公平的提供服务。显然这是从交易所到券商的交易系统都必须实施的机制。

从上述例子可以看到，Cloud-native 的架构模式，很多是与 Reactive 原则一致的，因而也是有助于构建 Cloud-ready 应用的。

那么容器化技术在这其中有什么作用呢？实践告诉我们，作用很大。最重要一点是，很多 pattern 的具体技术实现，可以挪到容器层面去处理。例如 Circuit-breaker 和 Fail-fast 这类模式，过去的做法，是各个具体的服务，采用自己的技术工具（例如 Node.js 的守护进程 PM2）和办法，基于开发者自己的理解，各自实现。

可以想象一下，在一个巨大而复杂的技术系统里，有很多服务、组件、模块需要实现熔断、快速失败、流控之类的机制，而这类系统很可能是采用所谓 polyglot programming（混合编程）、polyglot persistence（混合存储）、polyglot processing（混合数据处理）的方式实现，涉及多种语言、多类异构技术，如果流控、熔断在各个模块、服务、组件里各自随意实现，显然最终导致系统综合行为的不可预测。

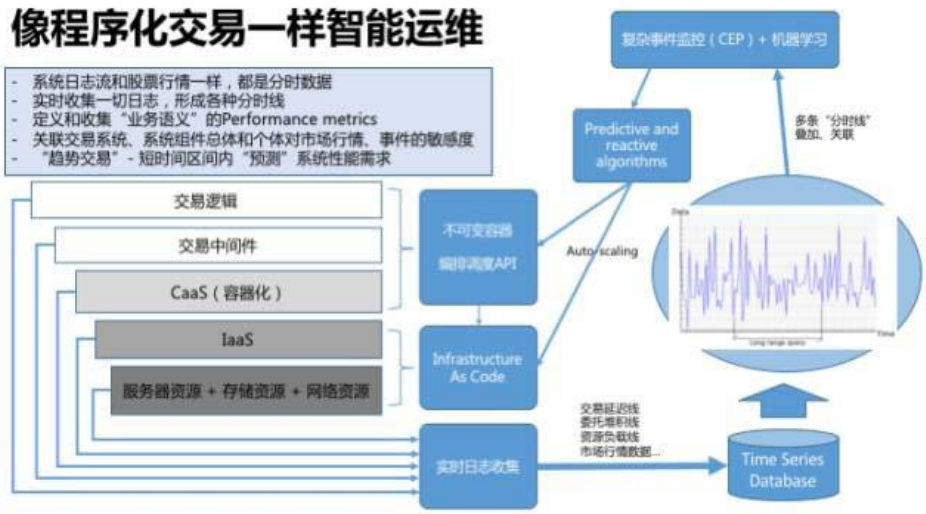
容器化的好处，是把复杂系统里一切异构的技术（无论以 Go、Java、Python 还是 Node.js 实现）都装载到一个个的标准集装箱里，然后通过调度中心基于各种监控对这些集装箱进行调度处理。也就是说，Cloud-native 的架构模式，有不少是可以在容器编排调度与协同的这一层实现的。

注意这种调度本身也是“智能”的，与业务逻辑、业务语义的性能指标（例如交易委托订单的吞吐、堆积）监控深度结合，甚至利用大数据技术进行自我的监督学习，实现“无人值守”的运维。

综上所述，云、容器、容器编排等等这些技术，不是银子弹，不会魔

术般的把一个本身“脆弱”的 IT 系统变成一个强韧的、甚至“反脆弱”的系统。构建一个能充分发挥云计算系列技术之能力的系统，**需要深刻理解容器技术和 Reactive 之架构风格**，以及熟练运用各种 Cloud-native 的架构模式，才能实现 Cloud-ready 的解决方案。

而这一切，如上文所提出，需要“研发的视角”，需要研发团队驱动。这对于没有研发组织或者缺乏工程师文化和一流研发团队的机构组织（尤其是垂直行业尝试投入到所谓“互联网 +”的），是一个巨大的挑战。



DevOps：云计算时代的方法论和文化

南怀瑾在某本著作里举过一个剃头匠悟道的例子，无论何种行业，技艺追求到极致可能悟到的道理都是共通的。《黑天鹅》和《反脆弱》的作者塔勒布本人也许算的上是这样的一个触类旁通的好例子，由金融而“悟道”于哲学（起码被称之为本世纪有影响力的思想家之一）。

IT 领域不知道有无类似的人物，诸如面向对象设计、架构设计模式、敏捷开发领域的大师级人物 Martin Fowler 等，显然是抽象思维特别强的人，能够对复杂的技术世界进行“模式识别”而作出一些深刻思考。在技

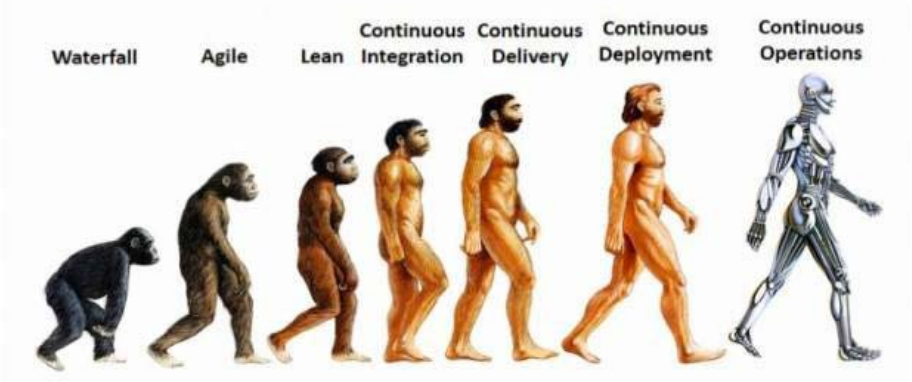
术世界里，我们一向强调方法论，可是有时候也需要形成“世界观”、“信仰”。在无数次的项目危机管理、技术故障攻坚、运维救火之后，IT 人也许应该对“世间唯一不变的是变化本身”有深刻体会，从而接受“拥抱不确定”的观念。

IT 界面对变化与不确定性的态度，这十多年来看也是一个有趣的演变。

- 直到本世纪初，很多项目管理及软件工程的方法论，依然强调所谓的“Change management”（变更管理），通过组织（“变更委员会”）、流程来“管理”业务系统需求方不断提出的需求管理，视需求变化为项目延误、系统不稳定的根源，以项目“按时”交付为终极目标，可以说这些方法论本质上“厌恶“变更。
- 互联网风起云涌后，出于应对瞬息万变的激烈竞争、在线高效运营的刚需，接受“需求变化是常态”、对变更友好的“敏捷”（Agile）方法被自然而然引入到日常项目中，成为过去十年的主流方法论，并且终于把传统企业IT牵引其中（例如广发证券IT启动金融电商系列项目研发前的第一件事就是先把敏捷实践建立起来，对口的方法论才可能带来对的结果）。
- 然而，绝大部分企业的敏捷实践都局限在垂直业务线的项目团队里，而运维作为一个维护全企业IT生产资源的横向平台型组织，通常被排除在敏捷实践之外。敏捷迭代方法解决得了业务需求变更的问题，解决不了系统上线后各种突发性的变化：故障的及时解决、版本的迅速更新（业务部门总是迫不及待的）、在线经营的瞬间生效（运营人员分分秒秒催着）…… 在一切都嫌慢的“互联网时间”里，运维貌似成为最后掉链的一环，以“稳健”为主导的运维

团队与“进取”的研发、运营团队无可避免产生冲突。

- 时至今天，随着“把基础设施数字化”的云计算的普及，一个新的方法论——DevOps，闪亮登场。之所以说这是一个方法论，是因为它绝不仅仅是“Dev + Ops”这样简单粗暴的把开发工程师和运维工程师捆在一起，用“同一个项目组”、“同一套KPI”来强迫他们分享“同一个梦想”了事。它是在APM（应用性能监控）、Infrastructure As Code（可编程运维）、Virtualization（虚拟化）、Containerization（容器化）等等这些云计算时代的产物出现后，基于新的技术工具、技术理念而自然产生的。持续集成（Continuous Integration）、持续交付（Continuous Delivery）、持续运维（Continuous Operation）是DevOps的具体环节和手段，它相当于把一条纯数字化链路上不同的参与者关联到一起，无论是开发工程师还是运维工程师，最终都不过是身份稍微差异的Information worker。



改一下毛泽东《满江红》的名句，**“一分钟太久，只争朝夕”**，可以形容这就是 IT 技术、方法论演变的动力，一切都是为了更高效。DevOps 的“哲学”，开玩笑的说，是“可以自动的绝不手动”，而容器化技术以其轻量敏捷和丰富的 API 接口“加剧”这一可行性和趋势。

DevOps 可能不仅仅是个概念、方法论、技术新名词，它是伴随云计算自然发生的。但它的接受与运用，对于传统企业的 IT，尤其是“维稳”为主导思想的受监管行业的 IT，可能是一种文化冲击。传统 IT 甚至是今天的很多技术相对前沿的互联网公司，依然把团队拆分成“运维”、“开发”，在组织结构层面建立相互制衡，以避免开发团队的“冲动冒进”导致生产系统的不稳定，但运维职能往往变成一种“权力”（privilege），系统的迭代更新都需要获得运维“审批”，这本身显然就是一个“脆弱系统”，因为对变更是绝不友好的。在技术进步、时代变迁的大环境下，这种过去合理的做法，早晚变成一个将要被颠覆的存在。

无论如何，我们认为割裂的讨论一个高性能运算（HPC）的技术系统（如证券交易系统）的容器化、“云化”是不够的，DevOps 是构建“反脆弱”技术系统的方法论（起码到目前为止。也许将来有新的思维出现），也是我们系统能力的天然一部分。

结语

《奇点临近》（Singularity Is Near）的作者、谷歌未来学家库兹威尔（Ray Kurzweil）有句话可以借用来描述这个世界的变化：“There is even exponential growth in the rate of exponential growth”-指数级的变化率本身之变化也是指数级的。世界在加速朝着“Matrix”演变（well，believe it or not），本身就已经数字化的金融，显然是数字世界的一块基石。

但这也是一个典型的“极端斯坦”，黑天鹅（或者就是佛教所谓的“无常”？）将更多的光临，而数字金融里的负面黑天鹅，危害最巨大。事实上整个金融体系本身就是一个“脆弱系统”，《高频交易员》（Flash

Boys，作者 Michael Lewis）提到 SEC 其实往往因弥补市场规则漏洞而造成新的漏洞。技术的高速发展则将让发现、利用漏洞进行套利的行为变得更容易、更频繁，而这些行为带来的后果也是不可预测的。

我们从软件研发的视角、证券交易的视角来看待云计算，深感把促进基础设施数字化、运维代码化的容器化技术，运用到交易系统技术中，是天作之合。假如运用得当的话，结合机器学习和智能算法，能帮助我们构建一些“反脆弱”的技术方案。（既然有能下围棋的阿尔法狗，我们是否也可以开始憧憬懂得自救的运维狗？）

但是云计算前沿技术在金融业的更广泛应用，目前取决于行业监管、企业文化、技术人才素质。例如领导者一句“云会不会引起信息安全问题”这样“放诸四海而皆准”的深沉状问话，总是可以让行业、企业内一切科技创新打回原形。

不过，再借用本文开头所引用黄仁宇先生的“大历史观”，可以看到的潮流脉络是，**世界是越来越数字化的、变化是越来越频繁的、而 IT 是需要拥抱变化的**，云计算则只是这个潮流里完全符合趋势而自然出现的技术阶段。有一天 Oracle、SAP、各种著名与非知名的专业软件都把它们自身的产品基于容器来构建（例如一个多进程组成的数据库实例里的进程各自运行在自己的容器中），也许对于行业监管者、企业技术决策者而言，云已经无需概念上的存在，而能被毫无质疑的接受。这一天不会太远，据 Gartner 称，2020 年企业中无云战略将极为罕见，和今天的企业无互联网一样难以想象。

但是这一天到来之前，大部分企业也许可以先尝试调整一个对云服务友好的财务制度，项目立项的时候，硬件预算按 CPU 核数、内存量、存储量来报算，有形的物理机器不再作为资产稽核到各条业务线各个项目组，一切物理硬件归 IT。制度和文化，往往才是隐形的决定因素，不是吗？

基于 Event Sourcing 和 DSL 的积分规则引擎设计实现案例

龚力

架构设计模式（Architecture Patterns），是“从特殊到普遍”的、基于各种实际问题的解决方案而总结归纳出来的架构设计最佳实践，是一种对典型的、局部的架构逻辑的高度抽象思维；在合理的场景下恰当使用它们，避免“重新发明车轮”，对技术解决方案有指导性作用，往往事半功倍。广发证券 IT 研发团队作为架构设计模式的坚定践行者，在各类证券业务中经常运用。Event Sourcing 就是这么一个比较常用而重要的架构模式。本文介绍的虽然是金融业场景，但是“积分系统”相信对其他行业的开发者也不会陌生。技术团队尝试用 Event Sourcing 架构模式和基于 Go 构建的 DSL“简单而优雅”的解决一个问题。

在电商行业，积分几乎已经成为了一个标配。 京东、淘宝都有自己的积分体系。 用户通过购物或者完成指定任务来获得积分。累积的积分可以给用户带来利益，比如增加用户等级，换取礼品或者在购物时抵扣现金。

在广发证券的金融电商运营平台中，积分同样是一个不可或缺的基础服务，很多应用都有和积分账户交互的场景。积分的用途也比较广泛，除了用在面向客户的服务中增加客户粘性和忠诚度外，积分也被用来支持内部的“游戏化”（gamification）运营，让数字化经营成为可能。 例如：公司的投资顾问可以通过编辑高质量的理财知识条目和回答客户问题获得积分，最终被换算回个人绩效收入。

一个场景，是客户提了一个关于证券的问题， 如果投资顾问回答了这一问题，并且答案被其他用户收藏，就可以获得 500 积分作为奖励。实践表明，积分的使用大大提升了投资顾问回答问题的积极性，提高了运营的效率。这其实是精细化运营、数字化经营的一个非常重要的基础设施。这个积分体系的存在，甚至改变、颠覆了传统企业对员工进行分派任务、管理、激励、计算个人绩效的机制。

从技术的角度，怎样实现一个积分系统满足各种应用程序的需求呢？虽然使用积分的场景不同，有的面向客户,有的面向公司内部的理财顾问，进行抽象后的积分系统可以是相同的。 和银行账户类似， 一个用户的积分账户可以看作由账户类型， 表示余额的数字和一系列引起积分变化的流水帐组成。 根据这些共性，我们把积分实现为一个独立的服务，统一存储管理积分数据。 在和应用程序交互的方式上，最初的想法是积分系统为应用程序提供增加 / 扣除积分的接口， 由应用程序决定增加 / 扣除积分的数量。

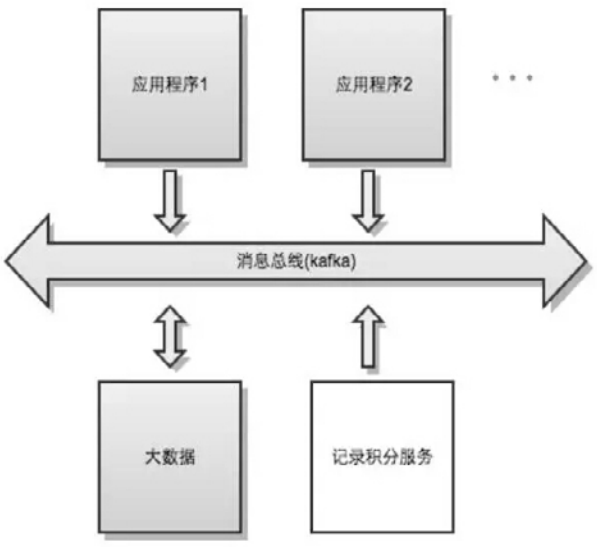
我们很快发现这种架构在应用程序中嵌入了积分规则逻辑，当积分规则改变时，应用程序需要随之改变。 比如，如果运营人员把上面例子中的 500 积分改变为 1000 分后，开发人员就需要升级应用程序。 在使用积分的应用程序数量多，运营需求变化快的情况下，这种应用程序和积分系统紧密耦合的架构增加了系统维护成本。

典型的“事件驱动”场景

无论是面向消费者客户的电商平台、航空公司顾客的飞行里数服务（mileage program）还是面向内部员工的“游戏化运营”平台，很显然，在技术层面都是一个典型的“事件驱动”场景 - 用户通常通过在各种各样的业务系统进行了一些活动，这些活动被记录到一个积分系统中“映射”成一定的积分。所以，积分系统设施的“使用者”，往往是其他的一些各式其色的、事前甚至无法预估的应用程序。

技术架构的设计原则是这样：由不可预知的应用程序自己负责判断其用户所进行的活动有无“价值”，对于有价值的活动则以发起事件的方式异步通知积分系统，积分系统则负责实时收集事件并基于各种可能由经营管理者随时修订、配置、改变的积分规则对事件所包含的用户活动进行“簿记”（book-keeping）。

我们采用了基于消息总线的架构设计， 应用程序和积分系统之间通过异步的消息总线关联。应用程序不包含任何积分规则，只负责向消息总线发布事件。 积分系统被实现为一个独立的服务，包含了所有的积分账户数据和积分规则。



积分系统向消息总线订阅事件， 然后根据设置的积分规则处理事件， 记录积分。这种架构使应用程序和积分系统呈松耦合关系，提升了系统的可维护性。 系统架构如右图所示。

举一个例子说明记录积分的过程：某投资顾问在广发证券知识库的应用程序中回答了一个问题，并且该问题被一个客户收藏。 知识库应用程序向消息总线发布一个答案被收藏的事件。 积分系统在监听到这一事件后，根据事先配置的积分规则，向投资顾问的积分账户增加积分数量，记录积分流水。

积分系统监听的事件并不一定由应用程序直接产生。对于复杂的积分规则，可能由其他服务处理应用程序的事件流后，产生新的事件流，再由积分系统处理。例如，需要对 7 月份连续 3 天登录的用户奖励 50 分。应用程序没有保存历史登录数据，只产生简单的登录事件。 大数据平台（对于积分系统而言是一个应用程序）可以根据保存的历史数据产生包含连续登录天数的事件， 发布到消息总线上后由积分系统订阅处理。 这体现了基于消息总线架构的优点，能把积分处理逻辑从应用程序中完全剥离出来，同时具有扩展性。

积分类似虚拟货币，可最终换算成员工绩效或者消费者的某些形式的奖励，所以不能多记，也不能少记。为了达到这一目标，技术层面上需要解决消息被处理一次且仅被处理一次的问题。我们的消息总线采用的是分布式消息系统 Kafka， 它具有比较好的容错性和扩展性， 但不直接提供这样的支持，需要在应用程序层面处理。 应用程序向 kafka 发送消息时可能因为网络的原因发送失败。

为了避免丢失用户积分，我们要求应用程序在向 Kafka 发送消息失败后进行重试。但这样又有可能出现同一个积分事件被重复接收导致多记积分的问题。 我们的解决办法是应用程序在产生积分的事件中带上一个对用户唯一的 uuid， 并且通过重发的机制确保事件最少被发送到 Kafka 一次。 在积分系统中根据 uuid 进行排重，丢掉 uuid 重复的积分事件，保

证积分事件最多被处理一次。通过这样一种应用程序之间的协议实现了一个积分事件被被处理一次且仅被处理一次的目标。

Event Sourcing 架构模式

在实践中，我们有修正积分的需求。 比如， 由于 bug， 应用程序错误的产生出了一些事件，需要减掉由这些事件而增加的积分。直接的方法是找出这些事件产生的积分，然后从账户中直接扣减。 但是这一方法在下面的场景中会导致错误：

假设积分规则是用户首次登录奖励 500 分，当天内第 2 次登陆再奖励 1000 分。

1. 由于应用程序错误，产生了登录事件L1，导致增加500积分。
2. 用户登录产生登陆事件L2。 积分系统发现当天已经出现过1次登陆事件L1， 根据规则增加了1000积分。

管理员发现为 L1 不应该发生，直接扣除 500 积分，用户实际得分 1000 分。 这是错误的。 在没有事件 L1 的情况下，登陆事件 L2 只应该获得 500 分。产生这一错误的根本原因是积分的计算可能依赖于历史事件。历史事件的变化将影响后续事件处理。

解决这种问题的一种方式是：当历史事件发生了变化时， 回滚到该时间点前的历史状态，然后按照时间顺序重新处理之后的所有积分事件，这类似于数据库系统中使用 checkpoint 和日志来恢复数据库状态的方式。Event Sourcing 概括了这种软件设计模式（详细内容可参考软件设计领域大师 Martin Fowler 的相关文章）。Event Sourcing 模式最核心的概念是程序的所有状态改动都是由事件触发并且这些事件被持久化到磁盘中。 当需要恢复程序状态时，只需把保存的事件读出来再重新处理一遍。

积分系统遵照 Event Sourcing 模式实现。 积分的所有变化都由积分事件触发，所有积分事件都存储在数据库中。为了回滚积分账户状态，还需要保存积分账户的历史数据。我们实现的方法是在积分账户发生变化时，产生一条积分流水，保存了积分变化数量，以及积分变化前和变化后的总额。当需要回滚积分账户状态时，找到离回滚时间点最近的积分流水，恢复历史积分账户的总额，然后按照时间顺序逐一处理保存的积分事件，恢复积分账户数据。 下图展示了这一流程。

下面是用命令行工具把积分账户状态恢复到 2016-05-01 之前，然后重新处理积分事件恢复积分的界面。

```
$ ./rescaner -event_since 2016-05-01T00:00:00+08:00
 5447 / 27022 [=======>----------------------------]  20.16% 1m4s
```

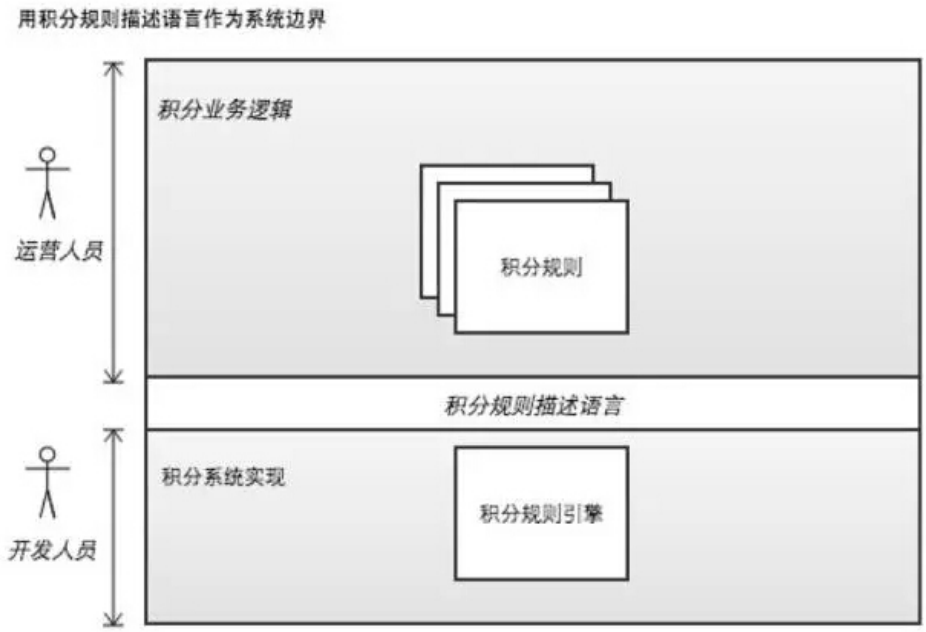
在生产环境的运维经验表明，相对于手工直接修改积分账户数据，这种修改历史积分事件，回滚账户状态然后重新处理积分事件的方式不但提高了准确性，而且简化了修正工作，节省了运维人员的时间。

用 Go 构建 DSL 实现灵活的积分规则引擎

由于接入的应用程序类型多样，积分规则会随着运营的开展而频繁变化。如果每次积分规则发生了变化，都要求对积分系统改动升级， 积分系统维护就会变成一项很繁琐的工作。 我们的目标是让积分系统保持足够的灵活性，当积分业务规则变化时，在大多数情况下可以不用改动升级

积分系统。最理想的情况是运营人员通过简单培训后自己就能配置积分规则，不需要开发人员修改积分系统软件。

为此我们开发了一个积分规则引擎， 通过提供一个积分规则描述语言，把积分的业务逻辑从积分系统软件中分离出去。



下面首先描述积分规则描述语言的语法表示和存储方式， 然后描述规则引擎加载解释积分规则的流程。

积分规则引擎首先需要提供一个让运营人员描述积分规则的语法。抽象的看，积分规则可以表示为一个元组：（积分条件，积分数量）， 表示当满足设置的条件时，增加对应的积分数量。 很容易联想到积分条件可以用编程语言中的布尔表达式表示，积分数量用数值表达式表示。

由于我们使用的是 Go 语言实现积分系统， 出于解析方便的考虑（Go 自带了自身的语法分析库），我们采用了 Go 语言的表达式语法表示积分规则的条件和数量。 在积分规则的表达式中，Go 语言的字符串、数字、布尔常量都可以直接使用。变量表示积分事件中的字段数据。比如，积分规则 (event_type==“answer_is_liked”，250) 表示当前积分事件类型

(event_type) 为 answer_is_liked(答案被点赞) 时，积分条件匹配，记录 250 个积分 。

在定义了积分规则的语法表示后，还需要决定在哪里存储积分规则。最初考虑存放在文件中，很快发现如果把积分规则和积分数据存放在同一个数据库中就可以方便的利用数据库的一致性检查功能保证数据一致性，这是保证软件系统长期正确运行的关键措施。 比如，通过数据库的外键设置，我们能保证每条积分流水指向一个有效积分规则，杜绝因为规则被错删，积分流水指向无效积分规则的情况。下面是积分规则在数据库中表示的例子。

id (积分规则id)	app_id(应用程序id)	criteria(条件)	points(分数)	msg(积分流水消息模版)
1	1	event_type=="answer_is_liked"	250	答案被点赞一次,{points}分
2	1	event_type=="answer_question" && count_by_same_event_attr("question_id") == 0	((data.originator_type=="consultant" && 4000) || 2500)	新增答案,{points}分
3	…	…	…	…

上表第 1 行积分规则表示当积分事件是 answer_is_liked 时，增加 250 分；

第 2 行要复杂一些，表示当积分事件是 answer_question(回答问题)，并且属于首次回答问题时增加积分， 如果是投资顾问，增加 4000 分，其他人员增加 2500 分。其中 event_type 是积分事件的字段；count_by_same_event_attr 是在规则表达式中允许使用的函数，用来统计该用户的具有相同字段值的积分事件数量；data.originator_type 也是积分事件的字段，表示用户类型。

为了增强扩展性，规则引擎提供了一套插件机制，可以用 Go 语言编写能用在规则表达式中使用的函数。比如上表第 2 行中的 count_by_same_event_attr 就是通过插件实现的，用来计算目前已经收到的具有相同属性值的事件数量。在实践中，当发现积分规则不能满足业务需求时，我们往往通过编写插件的方式来扩展积分规则的表达能力，而不是修改规则引擎的核心代码。

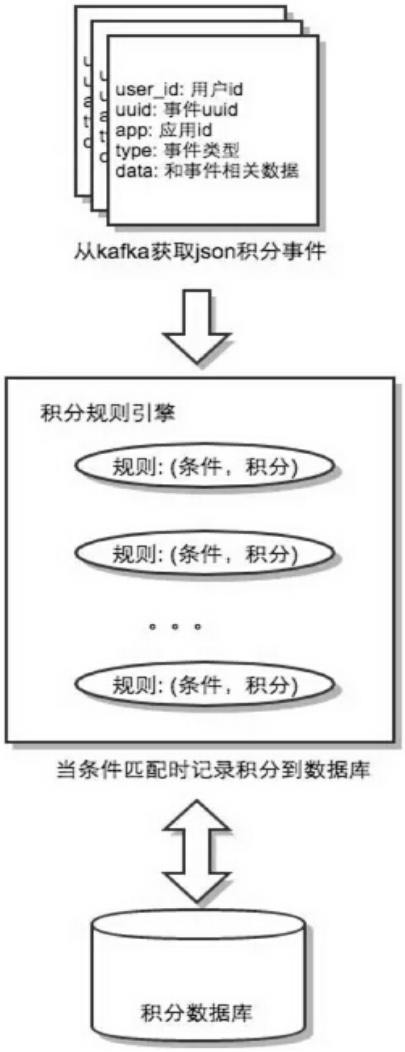
在运营人员配置积分规则后，积分系统需要使用规则引擎解释执行积分规则，主要流程是如下。

1、积分系统在启动时加载所有应用程序的积分规则

积分规则在被规则引擎加载后完成语法解析，在内存中解释执行。 这避免了在运行中访问磁盘或数据库引起的性能瓶颈。需要注意的是，虽然积分规则的语法和 Go 语言表达式相同，积分规则的语义却有变化。对于会引起 Go 语言抛出异常的表达式（e.g. 除 0），积分规则引擎解释为nil，避免了程序异常退出。

2、监听消息总线，对于新收到的积分事件，逐个尝试匹配积分规则的条件。如果该积分事件能满足某个积分规则的条件，则增加由积分规则中的积分。

右图表示了运行规则引擎记录积分的流程。



可以看出，我们实际上构造了一个 DSL（Domain Specific Language），语法和 Go 语言的表达式一样，但是语义不同。积分规则其实是这一 DSL 编写的程序，作为数据保存在数据库中，在被规则引擎装载后又当作程序来执行。这里体现了“代码即数据”（code as data）的编程思想。

技术栈：Go + Postgres + Docker

1、Go 语言

Go 语言是为大规模系统软件的开发而设计的，具有语法简洁，静态类型检查，编译快速，支持并发程序设计等特点。

和 JavaScript 等动态语言相比，我们感觉在某些场景下，由于 Go 的类型系统比较复杂并且不支持范型，编写的代码量会多一些。一个典型的例子是排序，使用 Go 的排序库时，一般需要实现一个 sort.Interface，包含有 Len，Swap，Less 3 个方法。而使用 JavaScript 进行排序，往往只需要 1 行代码。

但是和动态语言相比，Go 的静态类型检查减少了很多运行时 bug，节约了调试时间，并且 Go 提供的工具比较完善，自带文档，格式化，单元测试和包管理工具。Go 的生态系统也比较成熟，第 3 方软件包丰富。综合来看，使用 Go 的开发效率并不会低太多。

我们发现 Go 语言的静态链接特性非常适合 docker 部署，积分系统用 docker 打包后只有 10M 左右。相比于 NodeJS 打包后上百 M 的体积，采用 Go 语言大大节省了部署时间和资源。

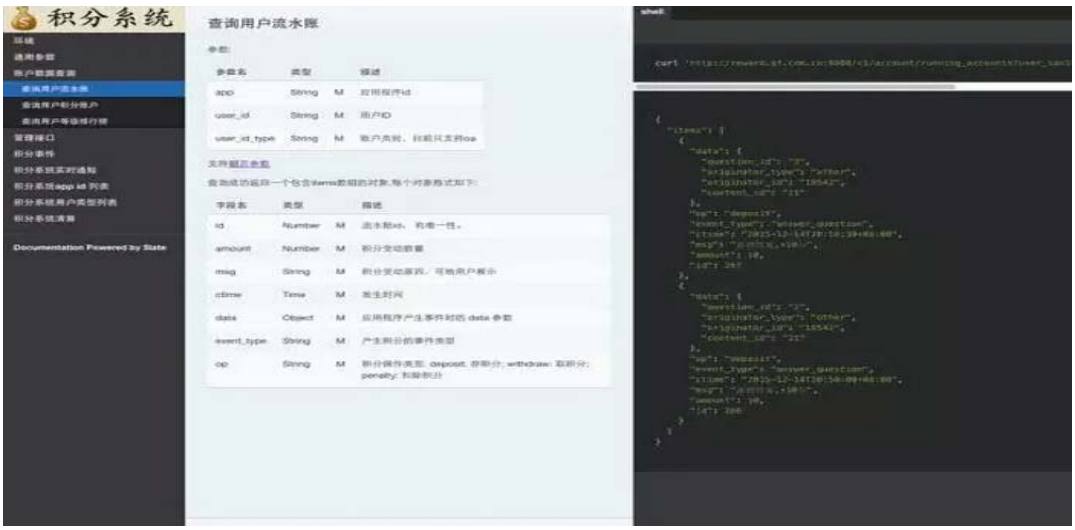
总的来说，我们对 Go 语言是比较满意的，将会继续在关键的系统服务中使用。

2、Postgres

在使用了一段时间的 MongoDB 后，我们希望在关键业务中采用有严格 schema 检查的关系型数据库。 Postgres 是一个成熟的开源数据库，除了支持数据一致性检查和事务外，也支持 JSON， 吸收了 NoSQL 的优点。

在积分系统中，应用程序需要在积分事件中保存一些自定义的属性，在查询积分流水时积分系统原样返回，由应用程序自行处理。 由于事先无法预知应用程序保存的内容格式，我们把这样的数据放在一个 JSON 字段中， 完全由应用程序控制。在数据存入之后， 通过 Postgres 的 JSON 操作符，我们可以方便的管理这些数据，比如，根据指定的 JSON 字段查询。

除了使用 Go、Postgres、Docker 这些技术开发和部署服务，由于积分系统是为应用程序提供服务的，它天然需要通过 API 来支持其他开发者。我们选择了用工具 slate 来制作 API 文档。下图是使用 markdown 编写，由 slate 转换成 html 格式的 API 文档式样。



总结

积分系统并不是一个技术架构上复杂的系统，但是它是借鉴“游戏”

实践而进行的数字化精细化经营的重要业务环节，相信在越来越多进行“互联网 +”创新的垂直行业中会有类似的实践。具体的技术实现手段也很多，在此为便于行业内外读者的理解，我们对方案作了简化和抽象。

然而，对相对简单的问题作“教科书”式的简练实现，遵循 KISS（Keep It Simple，Stupid！）的原则，避免“过度工程”（over-engineering），也是我们的团队文化和准则。本文所介绍的 Event Sourcing 架构模式和 DSL 规则引擎，可以帮助我们在很多场景“简单而优雅”（simple but elegant）的解决问题。

作者介绍

龚力，毕业于电子科技大学，广发证券 IT 研发资深架构师，一直负责金融电商、零售金融相关技术系统的设计与实施。是在大规模金融电商领域成功使用 MEAN（MongoDB、Express、Angular、Node.js）技术栈的最早践行者；近期更多采用 Go 语言进行业务开发。投身金融业前有多年互联网、电信行业研发经验。

容器化和木桶理论：证券交易系统的 Docker 化实践

谭成鑫

大型券商日成交额高达千亿级，交易系统既需要满足动辄要求独占数千 QPS 的单个机构客户，又要服务数百万至千万级的散户；市场效应性的集体瞬间抛出或买入、毫秒必争的抢购涨停板或跌停前止损、行业需求的频繁功能更迭及例如配合交易所技术升级或合规风控要求而近乎强制性的上线 deadline、严格的行业监管，使每一个技术的失误均有可能“牵一发而动全身”导致巨大经济损失，这样的交易环境不仅要求交易系统具备高可靠、低延时、高吞吐的特点，而且支持自监控、自伸缩、自修复、自学习等“响应式”特性。

通俗地讲，一个复杂的大型证券交易系统，应该是一个以守为攻，并在适应复杂、易变、苛刻的交易环境过程中，不断学习成长的技术体现。

一、传统的交易系统所面临的一些问题

单体式（monolithic）的交易系统，风险集中，难以水平扩容，以传

统关系型数据库为核心使得架构难以松散耦合，无法实现精细化业务治理与运维以分散和降低风险。补丁式的升级，往往需要消耗一至数月的测试来确保补丁不对整体服务产生影响和导致上线环境的兼容问题。

一方面常态性变更的行业规则、市场变化导致的持续的系统更新上线要求，另一方面高昂的部署成本无法下降，令 IT 被动地“疲于奔命” - 无论 IT 人力资源还是技术资源，均无法做到轻易 react to change（对变更作出响应）和 scale out（横向扩容）。

部署一套交易环境包含各种半自动和手动环节，是一项同时要求技术经验和工作量的劳动，部署环节的复杂和对环境的兼容要求，使得低成本部署难以实现。传统的技术架构也无法轻易水平伸缩，系统从上线那一刻就交出了主动权； 接手运维职责的运维团队，则往往只能是作被动监控，出现问题手工救火，既无法有效衡量甚至预测技术风险的工具、也无紧急故障出现时高效的灭火武器。

运维的被动，根源在于开发者所设计、开发的系统，往往只专注于业务功能，而没有把“可运维性”当作系统开发的第一属性，导致了系统交付后，运维团队处处被动。

二、容器技术促进了交易系统的微服务化

容器技术的出现，对解决很多问题有帮助，但是首先是促进“可运维性” - 通过合理的技术架构，系统的可运维能力可以得到根本提升，我们交付到运维伙伴们手里的系统，更加“智能”、更加“可控”。“微服务化”，是向这个方向努力的一个重要步骤。

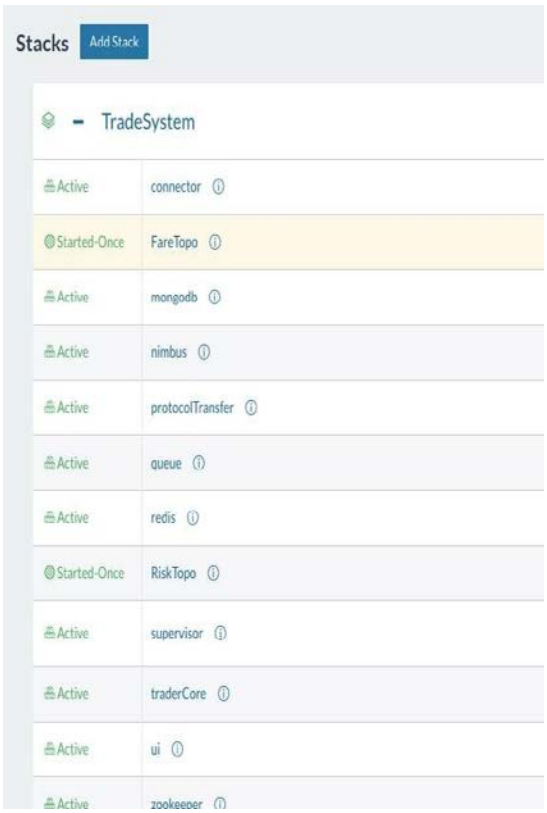
网上关于微服务优劣的讨论有很多，包括 Martin Fowler 本人也就微服务的合理应用提出一些警告，详情不在此讨论。碎片化的服务一方面让

精准运维、精准调优成为可能，另一方面也带来运维复杂度和各种耗损。

然而我们平衡取舍之下，还是采用了微服务的架构，因为交易系统里的微服务，让功能点的粒度变小，如果有好的监控工具、部署工具，对于交易过程中各种市场行情导致的系统瓶颈更容易被定位、更精细的局部优化。虽然微服务本身不一定依赖于容器，但是容器作为微服务载体却让微服务变的非常的可管理、易运维。一定程度上，容器本身弥补了细碎服务的可维护性，让我们敢于运用微服务架构。

容器技术带来的一个巨大便利，是将服务本身和所依赖的运行环境打包起来，以统一标准的镜像作为服务的交付物，实现“一次构建，随处运行”，并结合容器各种编配技术（如：docker-compose），使得架构复杂的大规模应用服务实现一键部署变得简单、快速，重新部署一套服务接近“0”成本。

在容器技术出现之前，“大系统小做”、“复杂系统简单做”等架构设计思想更多地被运用于基础资源充足、研发团队较大的大公司，对于中小规模的团队，难以承担划分出来的众多微服务所带来的成倍增加的维护成本。



在容器技术出现之后，容器技术推动了计算资源的数字化，容器的轻量、秒级部署和丰富的编程API，解决的微服务架构一键便捷部署的问题，重新部署一整套服务

的成本接近于“0”，通过“immutable infrastructure”解决了服务灰度发布、补丁升级所带来的问题，使”divide and conquer”思想更容易体现到系统架构设计当中，是微服务走入交易系统的重要原因。

总结我们的思路，是这样的：

- 传统单体的交易系统，可运维性低，惧怕改变（“脆弱”）。
- 证券业的市场复杂性、合规风控监管需求变化、交易所等生态环境的技术与标准持续变化等等，导致证券交易系统本身就既需要像一切金融系统一样高度可靠高度稳定同时又像互联网系统一样密集迭代。这是一个矛盾到变态的特点，恐怕在一般IT系统中罕见。
- 解决这个矛盾，我们把下一代交易系统微服务化，因为我们需要精准监控、精准运维、粒度可控。
- 容器，不仅仅是一种技术载体，更加是技术生态、工具链，甚至对DevOps方法论的促成，起到极大推进作用，故在微服务化过程中对我们至关重要，因为它帮助我们解决细碎微服务的易部署、可管理、弹性伸缩。

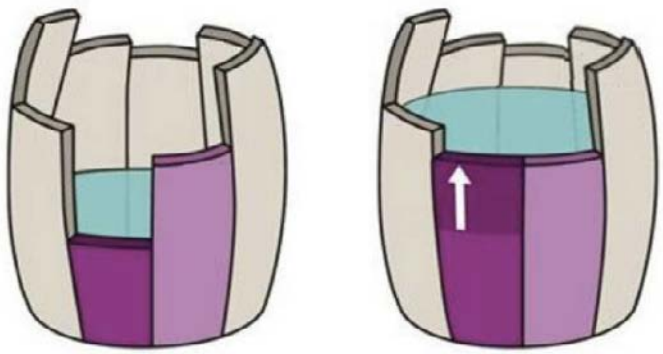
单体应用与微服务化应用这两个点之间的平衡如何拿捏，则是下文准备探讨的问题。

三、木桶的短板以及如何解决它

传统交易系统，其实也是由很多模块、服务组成，但这不等于它们就是“服务化”的、SOA 的，我们姑且“武断”的把它们称之为单体应用 - 从架构设计角度看，它们内部各模块、环节之间的耦合度非常高，称之为“单体”架构，并不为过（哪怕它们的组件用不同语言、异构技术实现）。这样一个系统，也许可以用“铁桶一个“来比喻，当交易量（或其他任何

指标）像水一样高涨、溢出的时候，几乎没有什么好的办法应对 - 也许只能换更大的铁桶，但是其极限终有时。

微服务化的交易系统，则可以比作一个由多块木板组成的木桶，一定程度上灵活性比铁桶高，因为总是可以有限度的加木板。但瓶颈如果用木桶理论来描述的话：当盛水量（外界的访问量）上升至木桶容量的极限（服务的性能瓶颈）时，目前绝大多数的做法是找到“木桶”的最短板，通过扩容来增加板的长度，当外界的访问量再次达到服务的性能瓶颈，再增加最短板的长度，如此反复。



这种扩容方式显然有它的问题，并未完全适用于交易系统：（1）交易系统所面临的请求量突然剧增的情形会导致需要同时修补多块短板，难以定位出所有导致服务瓶颈的短板，并且多块短板同时修补的力度难以把握；（2）随着请求量的不断增加，不断地修补短板，会导致木桶结构越来越复杂，给服务带来了不确定性和运维难度的剧增。

我们的方案：绝对不能做铁桶，但是也不能依靠一个由很多短板组成的木桶。我们要实现的，是“轻易低成本构建多个木桶”。

（1）业务级原语的载体——木桶

容器技术带来的服务一键、秒级、“0”成本部署，解决了微服务架

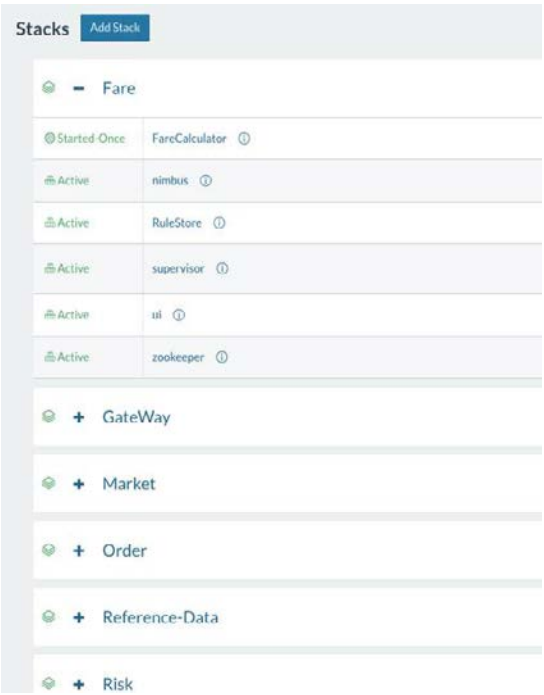
构的一部分运维问题，让我们“轻易低成本”构建一个“木桶”。例如对于我们的内存交易系统，仅普通股票的委托功能就由 46 个 Docker 镜像（image，不是 instance）构成，通过 Docker、Rancher 这样的工具，这 46 种木板构成的木桶（装载“委托管理”功能的木桶），瞬间即成。

但是，如果系统功能扩展到基金、债券和其他各种金融衍生品的全资产交易，如果需要支持夜间委托、组合委托等各种复杂的策略交易，如果还支持多租户（交易客户或应用），如果每种容器镜像又要支持冗余高可用和分片… 试想一下一套由上千个容器组成的微服务架构交易系统运行在线上，当出现异常时，要在上千个服务中快速定位问题、修复问题服务而同时确保没有引入其他问题的情形与难度。这和在上千块木板组成的木桶里找到漏水的那块一样挑战。

单纯的微服务架构，并没有从专业领域的视角去梳理各个微服务的领域属性和彼此之间的领域关系， 基于微服务之间的逻辑关系去对微服务进行物理上编配组合，显然很难达到把交易系统变成一个反脆弱系统（见《黑天鹅与云计算》一文）的要求。交易系统作为一个服务于专业领域的系统，不仅应该对单一服务的代码进行领域建模，架构的设计更应该同时基于业务原语（business semantic）。

容器编配技术让我们可以将基于类和接口层面的领域建模（domain modeling）运用到容器化的服务层面，让局限于单一容器粒度的架构设计和运维能力上升至业务功能的粒度，给我们交易系统的架构设计和服务运维带来了一个新的视角，是交易系统在微服务架构基础上融入“combine and conquer”思想的体现，是我们交易系统在容器化技术弯道上超车的加速度。

我们采用 Rancher 中提供的 stack 去描述我们的业务服务，如下图所示。

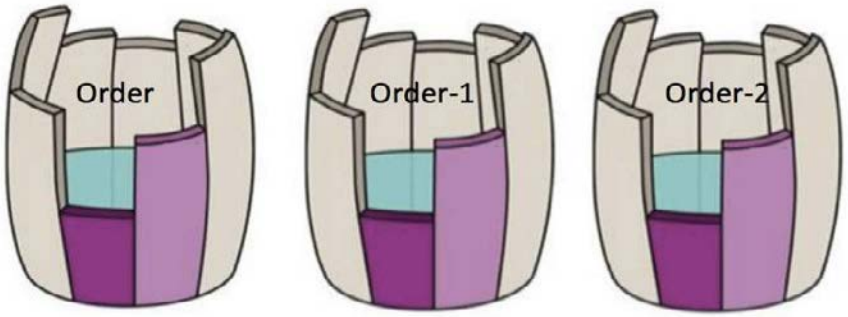


概括的说，用了容器编排技术的技术语义（例如 Rancher Stack），对应于业务层面的语义（例如 OMS - Order Management Service），形成一个装载某一强类型业务的“木桶”。一个复杂交易系统，本身由很多这样的“低成本构建”的木桶组成。每个木桶就是一系列相互作用的容器的集合单元。

（2）木桶承载“业务粒度”的功能

大型券商的日成交额动辄高达千亿、大量用户不约而同在某一时刻集体抛出或买入、开盘和收盘瞬间买入或卖出，要求我们研发的交易系统必须对外部环境的变化是高度响应（reactive）的。基于容器及其编配技术的多容器一键部署容许我们以整套业务单元为粒度进行扩缩容，从而避免了随着业务服务内部各个服务的扩容导致业务服务容器规模越来越大所带来的不确定性和运维难度的提升。

而基于业务服务的扩容是指，当外界的请求量达到服务的性能瓶颈时，我们的做法是通过容器编排技术的一键部署，再轻易启动一套一样的服务组合（另一个新木桶），而不是补丁式地去更改业务服务的组成架构（immutable infrastructure 理念）。对应到我们所使用的 rancher 当中，便是对同一个 catalog 再启动一个 stack（如下图里的委托管理服务，OMS，图中简称 Order）。当订单服务接近性能瓶颈时，我们只需再起一个订单服务 stack 即可。



每一套业务服务组合单元实例（木桶）从启动到关闭作为一个不可变的整体运行，业务服务内部的容器设计仅需考虑 fast-fail、heart-beat、retry 等 cloud native 特性，保证了从业务服务层面去应对外界的变化是简单可控的。木桶成只构建、成只销毁，我们永远不去修补里面的短板。

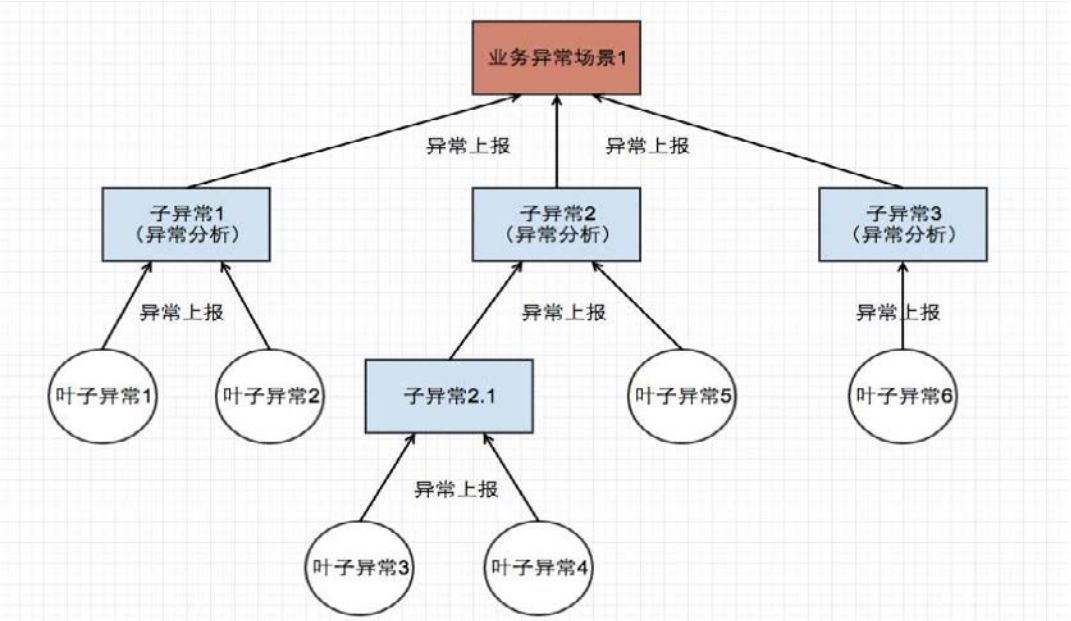
（3）业务语义下的监控

传统交易系统的技术，基于最底层的技术指标去监控系统性能，例如我们的运维工程师必须时刻监控交易委托主站的 CPU 和内存消耗、 关系型数据库所运行机器的压力状况等等，然后基于人脑、经验，去判断潜在的风险。

但显然这是低效的、不可靠的、后知后觉的。交易系统为用户提供的是业务服务，相应的监控系统应该从业务的层面去监控和预测各种已发生

或可能发生的异常场景，监控系统只有站在用户的视角，具备和用户一样的甚至先于用户的业务服务感知，才能最直接有效地响应交易系统的问题。准确、全面的异常反馈和预测，是交易系统实施自适应、自修复操作的前提，是一个 reactive 系统不可或缺的一项能力。监控本质上是对系统进行线上的、实时的测试，在交易系统的设计之初就对各种业务异常监控场景进行讨论、梳理和建模，是我们交易系统行为测试驱动开发（BDD - Behavior Driven Development）的一个体现。

业务异常场景的监控反映的是业务服务的异常，而每一项业务异常场景又可以从顶向下描述为一颗叶子节点为基础异常（比如程序直接上报的异常）的异常树。

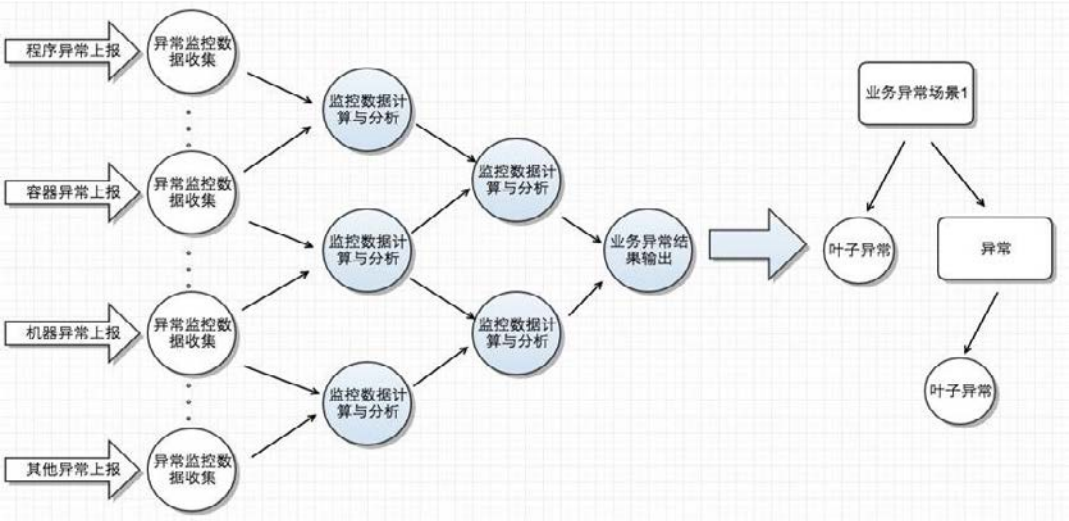


针对每一种业务异常场景，我们基于 riemann 来构建相应的监控分析拓扑。不同于以往 Pulling-Based 模型的监控系统，riemann 体现了 reactive 的思想，是一个基于事件流处理的实时监控系统。riemann 采用 clojure 语言配置事件的处理操作，只专注于事件的分析处理，保持自身的灵活、轻巧，由业务模块负责异常事件的检测上报和构建各种业务异常

场景的监控分析拓扑，更加适合各种业务场景的监控。

上述业务场景异常树中的每一个监控节点对应于一个 riemann 实例，以容器化的 riemann 为粒度，自底向上进行容器编配，构建业务异常场景的监控分析拓扑，从服务、机器监控数据的收集，到监控数据的计算和分析，最终输出一颗包含业务异常场景完整信息的监控结果树（如下图所示），从而清楚得知已发生或可能即将发生的业务异常场景详细分析结果，并结合基于业务服务的扩容和修复，为交易系统的自动运维提供了可能性。

通过强大的、丰富的、无孔不入的监控，结合低成本的、敏捷快速的



“木桶”构建手段，我们远在水快要溢出之前，即可迅速定位哪类木桶（对应于系统中的某类业务原语、或者说服务组合单元）需要复制部署，从而防范于未然。漏水的木桶？随它去吧，当市场上流动着数千亿上万亿的交易订单“洪水”，谁有空去修理旧木桶呢？

作者介绍

谭成鑫，广发证券 IT 研发工程师，北京航空航天大学计算机工学硕士，曾就职于百度后台高级研发工程师岗位，2014 年加入广发证券信息技术部，主要从事交易系统研发。

基于 Lambda 架构的股票市场事件处理引擎实践

邓昌甫

CEP（Complex Event Processing）是证券行业很多业务应用的重要支撑技术。CEP 的概念本身并不新鲜，相关技术已经被运用超过 15 年以上，但是证券界肯定是运用 CEP 技术最为充分、最为前沿的行业之一，从算法交易（algorithmic trading）、风险管理（risk management）、关键时刻管理（Moment of Truth - MOT）、委托与流动性分析（order and liquidity analysis）到量化交易（quantitative trading）乃至向投资者推送投资信号（signal generation）等等，不一而足。

CEP 技术通常与 Time-series Database（时序数据库）结合，最理想的解决方案是 CEP 技术平台向应用提供一个历史序列（historical time-series）与实时序列（real-time series）无差异融合的数据流连续体（continuum）- 对于证券类应用而言，昨天、上周、上个月的数据不过是当下此刻数据的延续，而处理算法却是无边际的 - 只要开发者能构想出场景与模型。

广发证券的 IT 研发团队，一直关注 Storm、Spark、Flink 等流式计算的开源技术，也经历了传统 Lambda 架构的技术演进，在 Kappa 架构的技术尚未成熟之际，团队针对证券行业的技术现状与特点，采用改良的 Lambda 架构实现了一个 CEP 引擎，本文介绍了此引擎的架构并分享了一些股票业务较为有趣的应用场景，以飨同好。

随着移动互联和物联网的到来，大数据迎来了高速和蓬勃发展时期。一方面，移动互联和物联网产生的大量数据为孕育大数据技术提供了肥沃的土壤；一方面，各个公司为了应对大数据量的挑战，也急切的需要大数据技术解决生产实践中的问题。短时间内各种技术层出不穷，在这个过程中 Hadoop 脱颖而出，并营造了一个丰富的生态圈。虽然大数据一提起 Hadoop，好像有点老生常谈，甚至觉得这个技术已经过时了，但是不能否认的是 Hadoop 的出现确实有非凡的意义。不管是它分布式处理数据的理念，还是高可用、容错的处理都值得好好借鉴和学习。

刚开始，大家可能都被各种分布式技术、思想所吸引，一头栽进去，掉进了技术的漩涡，不能自拔。一方面大数据处理技术和系统确实复杂、繁琐；另一方面大数据生态不断的推陈出新，新技术和新理念层出不穷，确实让人目不暇接。如果想要把生态圈中各个组件玩精通确实不是件容易的事情。本人一开始也是深陷其中，皓首穷经不能自拔。但腾出时间，整理心绪，回头反顾，突然有种释然之感。大数据并没有大家想象的那么神秘莫测与复杂，从技术角度看无非是解决大数据量的采集、计算、展示的问题。

因此本文参考 Lambda/Kappa 架构理念，提出了一种有行业针对性的实现方法。尽量让系统层面更简单，技术更同构，初衷在让大家聚焦在大数据业务应用上来，从而真正让大数据发挥它应有的价值。

1、背景

Lambda 架构是由 Storm 的作者 Nathan Marz 在 BackType 和 Twitter 多年进行分布式大数据系统的经验总结提炼而成，用数学表达式可以表示如下：

```
batch view = function(all data)
realtime view = function(realtime view,new data)
query = function(batch view .realtime view)
```

逻辑架构图见图 1。

从图 1 可以看出，Lambda 架构主要分为三层：批处理层，加速层和服务层。它整合了离线计算和实时计算，融合了不可变性（immutable），读写分离和复杂性隔离等一系列架构原则设计而成，是一个满足大数据系统关键特性的架构。Nathan Marz 认为大数据系统应该具有以下八个特性，Lambda 都具备它们分别是：

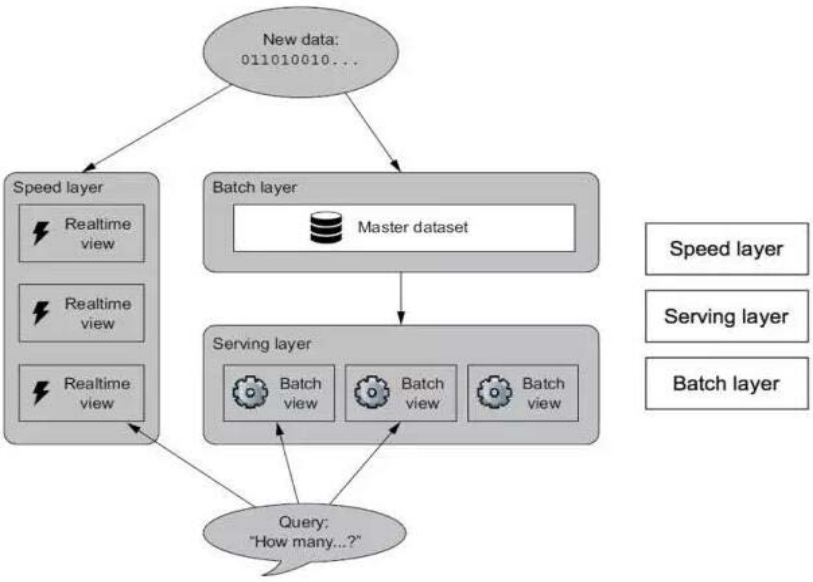


Figure 1.11 Lambda Architecture diagram

- Robustness and fault tolerance（鲁棒性和容错性）
- Low latency reads and updates（读和更新低延时）
- Scalability（可伸缩）
- Generalization（通用性）
- Extensibility（可扩展）
- Ad hoc queries（可即席查询）
- Minimal maintenance（易运维）
- Debuggability（可调试）

由于 Lambda 架构的数据是不可变的（immutable），因此带来的好处也是显而易见的：

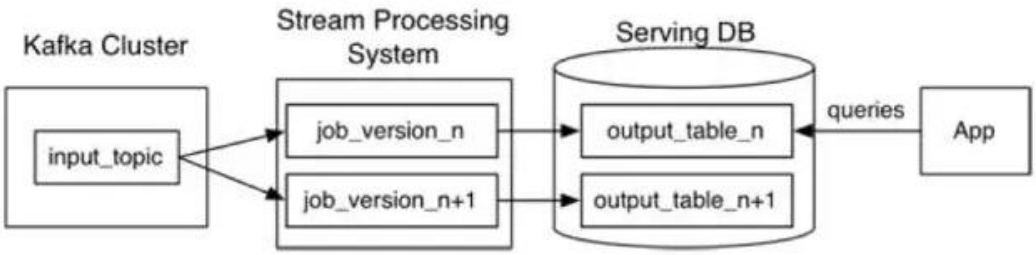
Human-fault tolerance（对人为的容错性）：数据流水被按时序记录下来，而且数据只写一次，不做更改，而不像 RDBMS 只是保留最后的状态。因此不会丢失数据信息。即使平台升级或者计算程序中不小心出现 Bug，修复 Bug 后重新计算就好。强调了数据的重新计算问题，这个特性对一个生产的数据平台来说是十分重要的。

Simplicity（简易性）：可变的数据模型一般要求数据能必须被索引，以便于数据可被再次被检索到和可以被更新。但是不变的数据模型相对来说就很简单了，只是一味的追加新数据即可。大大简化了系统的复杂度。

但是 Lambda 也有自身的局限性，举个例子：在大数据量的情况下，要即席查询过去 24 小时某个网站的 pv 数。根据前面的数学表达式，Lambda 架构需要实现三部分程序，一部分程序是批处理程序，比如可能用 Hive 或者 MapReduce 批量计算最近 23.5 个小时 pv 数，一部分程序是 Storm 或 Spark Streaming 流式计算程序，计算 0.5 个小时内的 pv 数，然后还需要一个服务程序将这两部分结果进行合并，返回最终结果。因此

Lambda架构包含固有的开发和运维的复杂性。

因为以上的缺陷，Linkedin的Jay Kreps在2014年7月2日在O'reilly《Questioning the Lambda Architecture》提出了Kappa架构，如下图所示。



Kappa在Lambda做的最大的改进是用同一套实时计算框架代替了Lambda的批处理层，这样做的好处是一套代码或者一套技术栈可以解决一个问题。它的做法是这样的：

1. 用Kafka做持久层，根据需求需要，用Kafka保留需要重新计算的历史数据长度，比如计算的时候可能用30天的数据，那就配置Kafka的值，让它保留最近30天的数据。
2. 当你程序因为升级或者修复了缺陷，需要重新计算的时候，就再启一个流式计算程序，从你所需的Offset开始计算，并将结果输入到一个新的表里。
3. 当这个流式计算程序追平第一个程序的时候，将应用切换到第二个程序的输出上。
4. 停止第一个程序，删除第一个程序的输出结果表。

这样相当于用同一套计算框架和代码解决了Lambda架构中开发和运维比较复杂的问题。当然如果数据量很大的情况下，可以增加流式计算程序的并发度来解决速度的问题。

2、广发证券 Lambda 架构的实现

由于金融行业在业务上受限于 T+1 交易，在技术上严重依赖关系型数据库（特别是 Oracle）。在很多场景下，数据并不是以流的形式存在的，而且数据的更新频率也并不是很实时。比如为了做技术面分析的行情数据，大多数只是使用收盘价和历史收盘价（快照数据）作为输入，来计算各类指标，产生买卖点信号。

因此这是一个典型的批处理的场景。另一方面，比如量化交易场景，很多实时的信号又是稍纵即逝，只有够实时才存在套利的空间，而且回测和实盘模拟又是典型的流处理。鉴于以上金融行业特有的场景，我们实现了我们自己的架构（GF-Lambda），它介于 Lambda 和 Kappa 之间。一方面能够满足我们处理数据的需求；一方面又可以达到技术上的同构，减少开发运维成本。根据对数据实时性要求，将整个计算部分分为三类：

1. Spark SQL：代替MapReduce或者Hive的功能，实现数据的批量预处理；
2. Spark Streaming：近实时高吞吐的mini batching数据处理功能；
3. Storm：完成实时的流式数据处理；

GF-Lambda 的优势如下所述。

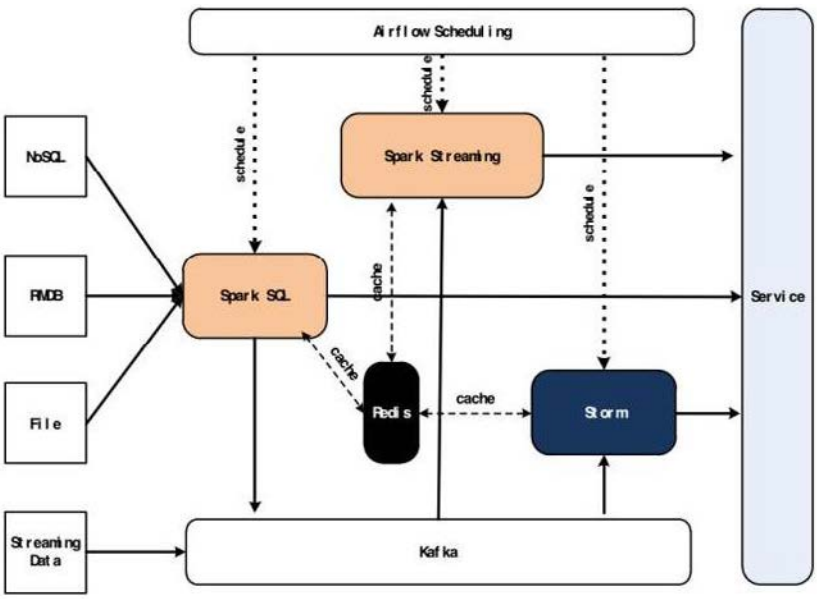
在 PipeLine 的驱动方面，采用 Airbnb 开源的 Airflow，Airflow 使用脚本语言来实现整个 PipeLine 的定义，而且任务实例也是动态生成的；相比 Oozie 和 Azkaban 采用标记语言来完成 PipeLine 的定义，Airflow 的优势是显而易见的，例如：

- 整个data flow采用脚本编写，便于配置管理和升级。而Oozie只能使用XML定义，升级迁移成本较大。
- 触发方式灵活，整个PipeLine可以动态生成，切实的做到了。

- “analytics as a service”或者 “analysis automation”。

另外一个与Lambda或者Kappa最大的不同之处是我们采用了Redis作为缓存来存储各个计算服务的状态；虽然Spark和Storm都有Checkpoint机制，但是CheckPoint会影响到程序复杂度和性能，并且以上两种技术的CheckPoint机制并不是很完善。通过Redis和Kafka的Offset机制，不仅可以做到无状态的计算服务，而且即使升级或者系统故障，数据的可用性也不会受到影响。

整个batch layer采用Spark SQL，使用Spark SQL的好处是能做到密集计算的后移。由于历史原因，券商Oracle等关系型数据库使用比较多，而且在开市期间数据库压力也比较大，此处的Spark SQL只是不断的从Oracle批量加载数据（除了Filter基本在Oracle上做任何计算）或者主动的通过Oracle日志旁录数据，对数据库压力较小，同时又能达到数据准实时性的要求；另外所有的计算都后置到Yarn集群上进行，不仅利于程序的运维，也利于资源的有效管控和伸缩。架构实现如下图所示。



3、应用场景

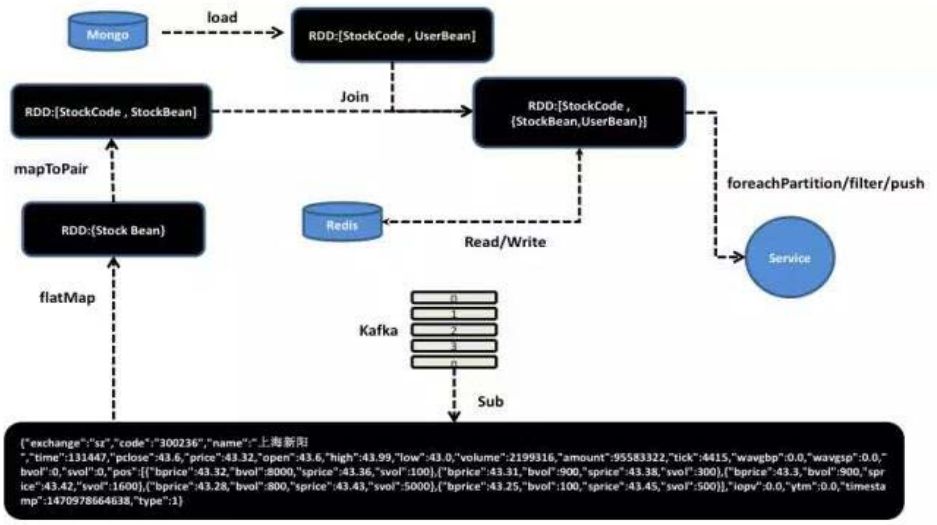
CEP 在证券市场的应用的有非常多，为了读者更好的理解上述技术架构的设计，在此介绍几个典型应用场景。

1）自选股到价和涨跌幅提醒

自选股到价和涨跌幅提醒是股票交易软件的一个基础服务器，目的在于方便用户简单、及时的盯盘。其中我们使用 MongoDB 来存储用户的个性化设置信息，以便各类应用可以灵活的定制自身的 Schema。在功能上主要包括以下几种：

- 股价高于设定值提醒；
- 股价低于设定值提醒；
- 涨幅高于设定值提醒；
- 一分钟、五分钟涨幅高于设定值提醒；
- 跌幅高于设定值提醒；
- 一分钟、五分钟跌幅高于设定值提醒。

主要的挑战在于大数据量的实时计算，而采用 GF-Lambda 可以轻松解决这个问题。数据处理流程如下图所示。

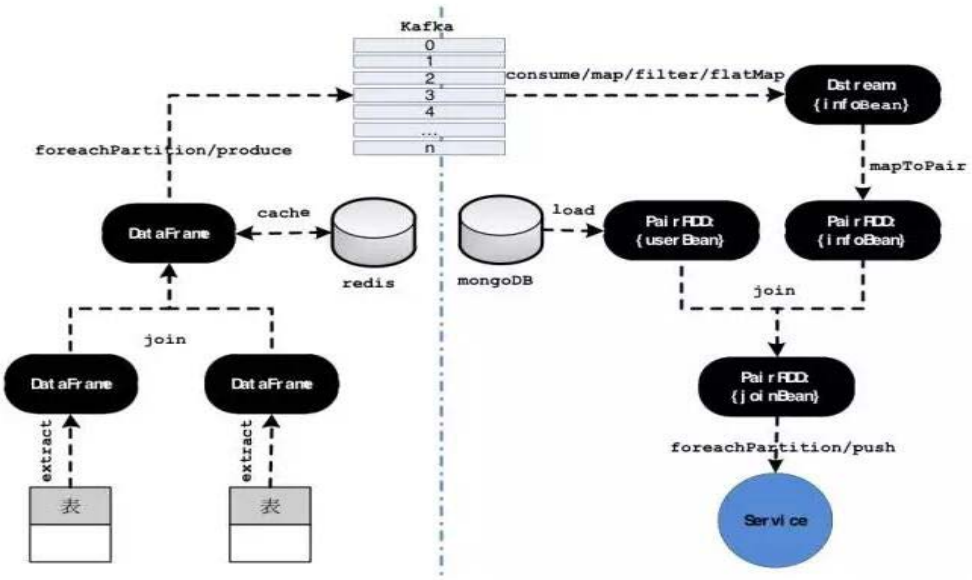


首先从 Kafka 订阅实时行情数据并进行解析，转化成 RDD 对象，然后再衍生出 Key（market+stockCode），同时从 Mongo 增量加载用户自选股预警设置数据，然后将这两份数据进行一个 Join，再分片对同一个 Key 的两个对象做一个 Filter，产生出预警信息，并进行各个终端渠道推送。

2）自选股实时资讯

实时资讯对各类交易用户来说是非常重要的，特别是和自身严重相关的自选股实时资讯。一个公告、重大事项或者关键新闻的出现可能会影响到用户的投资回报，因此这类事件越实时，对用户来说价值就越大。

在 GF-Lambda 平台上，自选股实时资讯主要分为两部分：实时资讯的采集及预处理（适配）、资讯信息与用户信息的撮合。整个处理流程如下图所示。



上图分割线左侧是实时资讯的预处理部分，首先使用 Spark JDBC 接口从 Oracle 数据库加载数据到 Spark，形成 DataFrame，再使用 Spark

SQL的高级API做数据的预处理（此处主要做表之间的关联和过滤），最后将每个Partition上的数据转化成协议要求的格式，写入Kafka中等待下游消费。

左侧数据 ETL 的过程是完全由 Airflow 来进行驱动调度的，而且每次处理完就将状态 cache 到 Redis 中，以便下次增量处理。在上图的右侧则是与用户强相关的业务逻辑，将用户配置的信息与实时资讯信息进行撮合匹配，根据用户设置的偏好来产生推送事件。

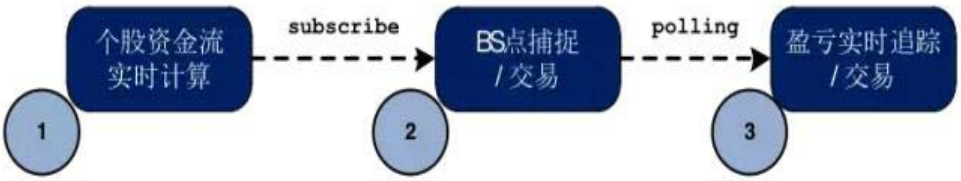
此处用 Kafka 来做数据间的解耦，好处是不言而喻的。首先是保证了消息之间的灵活性，因为左侧部分产生的事件是一个基础公共事件，而右侧才是一个与业务紧密耦合的逻辑事件。基础公共事件只有事件的基础属性，是可以被很多业务同时订阅使用的。

其次从技术角度讲左侧是一个类似批处理的过程，而右侧是一个流处理的过程，中间通过 Kafka 做一个转换与对接。这个应用其实是很具有代表性的，因为在大部分情况下，数据源并不是以流的形式存在，更新的频率也并不是那么实时，所以大多数情况下都会涉及到 batch layer 与 speed layer 之间的转换对接。

3）资金流选股策略

上面两个应用相对来说处理流程比较简单，以下这个 case 是一个业务 稍微繁琐的 CEP 应用 - 资金流策略交易模型，该模型使用资金流流向来判断股票在未来一段时间的涨跌情况。它基于这样一个假设，如果是资金流入的股票，则股价在未来一段时间上涨是大概率事件；如果是资金流出的股票，则股价在未来一段时间下跌是大概率事件。那么我们可以基于这个假设来构建我们的策略交易模型。如下图所示，这个模型主要分为三

部分，见下图。



（1）个股资金流指标的实时计算

由于涉及到一些业务术语，这里先做一个简单的介绍。

资金流是一种反映股票供求关系的指标，它的定义如下：证券价格在约定的时间段中处于上升状态时产生的成交额是推动指数上涨的力量，这部分成交额被定义为资金流入；证券价格在约定的时间段中下跌时的成交额是推动指数下跌的力量，这部分成交额被定义为资金流出；若证券价格在约定的时间段前后没有发生变化，则这段时间中的成交额不计入资金流量。当天资金流入和流出的差额可以认为是该证券当天买卖两种力量相抵之后，推动价格变化的净作用量，被定义为当天资金净流量。数量化定义如下：

$$\text{MoneyFlow} = \sum_{i=1}^{n} (Volume\ i) \times P_i \frac{P_i - P_{i-1}}{|P_i - P_{i-1}|}$$

其中，Volume 为成交量，为 i 时刻收盘价，为上一时刻收盘价。

严格意义上讲，每一个买单必须有一个相应的卖单，因此真实的资金流入无法准确的计算，只能通过其他替代方法来区分资金的流入和流出，通过高频数据，将每笔交易按照驱动股价上涨和下跌的差异，确定为资金的流入或流出，最终汇聚成一天的资金流净额数据。根据业界开发的 CMSMF 指标，采用高频实时数据进行资金流测算，主要出于以下两方面考

虑：一是采用高频数据进行测算，可以尽可能反映真实的市场信息；二是采取报价（最近买价、卖价）作为比较基准，成交价大于等于上期最优卖价视为流入，成交价小于等于上期最优买价视为流出。

除了资金的流入、流出、净额，还有一系列衍生指标，比如根据流通股本数多少衍生出的大、中、小单流入、流出、净额，及资金流信息含量（IC）、资金流强度（MFP），资金流杠杆倍数（MFP），在这里就不一一介绍。

从技术角度讲，第一部分我们通过订阅实时行情信息，开始计算当天从开市到各个时刻点的资金流入、流出的累计值，及衍生指标，并将这些指标计算完成后重新写回到 Kafka 进行存储，方便下游消费。因此第一部分完全是一个大数据量的实时流处理应用，属于 Lambda 的 speed layer。

（2）买卖信号量的产生及交易

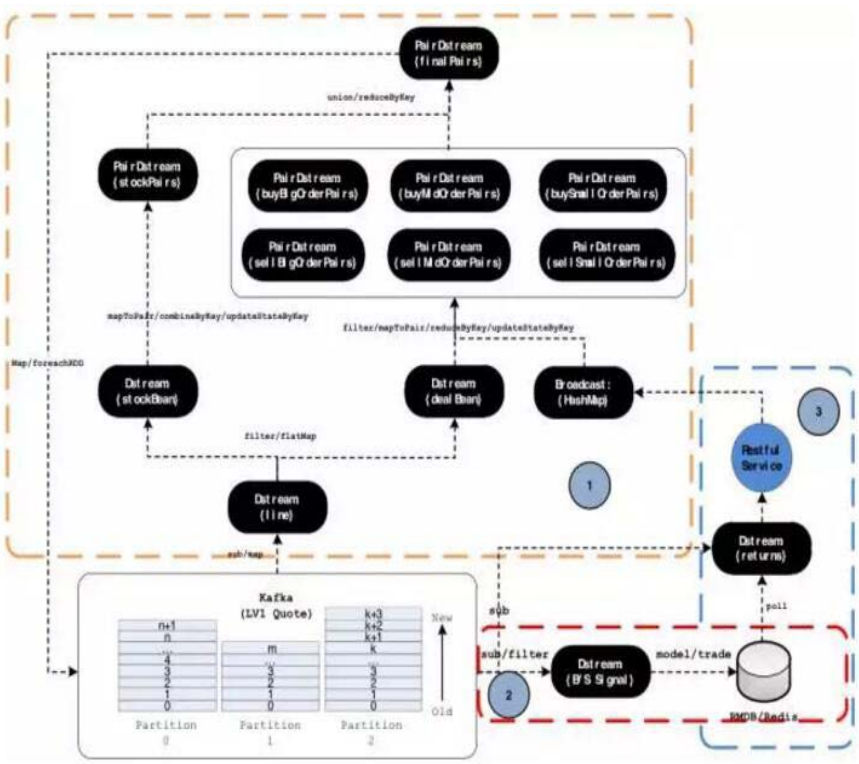
第二部分在业务上属于模型层，即根据当前实时资金流指标信息，构建自己的策略模型，输出买卖信号。比如以一个简单的策略模型为例，如果同时满足以下三个条件产生买的信号。反之，产生卖的信号：

- （大单资金流入-大单资金流出>0） &&（中单资金流入-中单资金流出>0）
- 大单的资金流信息含量>50%
- 大单的资金流强度>20%

在技术上，这个应用也属于 Lambda 的 speed layer，通过订阅 Kafka 中的资金流指标，根据上面简单的模型，不断的判断是否要买或者卖，并调用接口发起买卖委托指令，最后根据回报结果操作持仓表或者成交表。（注意此处在业务上只是以简单的模型举例，没有涉及更多的细节）

（3）持仓盈亏实时追踪及交易

第三部分在业务上主要是准实时的盈亏计算。在技术层面，属于 Lambda 的 batch layer。通过订阅实时行情和加载持仓表 / 成交表，实时计算用户的盈亏情况。当然此处还有一些简单的止损策略，也可以根据盈利情况，发起卖委托指令，并操作持仓表和成交表。最后将盈利情况报给服务层，进行展示或者提供回调接口。详细的处理流程如图 7 所示。



总结

正如文章前面强调的一样，写这篇文章的初衷是希望大家从大数据丰富的生态中解放出来，与业务深度的跨界融合，从而开发出更多具有价值的大数据应用，真正发挥大数据应有的价值。这和 Lambda 架构的作者

Nathan Marz的理念也是十分吻合的，记得他还在BackType工作的时候，他们的团队才五个人，却开发了一个社会化媒体分析产品——在100TB的数据上提供各种丰富的实时分析，同时这个小的团队还负责上百台机器的集群的部署、运维和监控。

当他向别人展示产品的时候，很多人都很震惊他们只有五个人。经常有人问他：“How can so few people do so much?”。他的回答是：“It’s not what we’re doing, but what we’re not doing.”通过使用 Lambda 架构，他们避免了传统大数据架构的复杂性，从而产出变得非常显著。

在五花八门的大数据技术层出不穷的当下，Marz 的理念更加重要。我们一方面需要与时俱进关注最新的技术进步，因为新技术的出现可能反过来让以前没有考虑过或者不敢想的应用场景变成可能，但另一方面更重要的是，大数据技术的合理运用需要建立在对行业领域知识深刻理解的基础上。大数据是金融科技的核心支撑技术之一，我们将持续关注最前沿的大数据技术与架构理念，持续优化最符合金融行业特点的解决方案，构建能放飞业务专家专业创新能力的技术平台。

作者介绍

邓昌甫，毕业于中山大学，广发证券 IT 研发工程师，一直从事大数据平台的架构、及大数据应用的开发、运维和敏捷相关工具的引入和最佳实践的推广（Git/Jenkins/Gerrit/Zenoss）。

奇谈怪论
从容器想到去 IOE、去库存和独角兽

梁启鸿

2016 年，容器化技术如火如荼，诞生于 2013 年的 Docker 成了行业的宠儿，它让炒了 8 年的 DevOps 有了更具体可落地可执行的工具。虽然有一定程度的过火现象（所谓的 hype），虽然有很多 IT 人（尤其是在传统垂直行业的信息技术部里）依然怀疑容器与虚拟机的差别，但总体来说，容器化可能算的上是软件开发领域的又一次“运动”。

每一次“运动”，都是有很多人追随、有很多技术架构被（一窝蜂的）重新设计、有很多系统被迁移，形成一种潮流（好像不那么干就被时代抛弃），代码的开发调试、架构设计和交付部署的方式发生巨大变化。

- Mainframe：大型机、大集中、傻终端（dumb Terminal）。
- 2层架构Client-server和4GL：整个90年代 - 中小型机/x86服务器、工作站/PC终端。Mainframe应用很多被采用C/S架构重写。一时间都是DCE RPC、DCOM、CORBA-IIOP。
- 3层架构通行：Web 1.0时代开始到现在，浏览器、中间件、关系型

数据库服务器的架构依然在很多企业中通行。C/S架构应用很多被迁移至三层架构。Struts+Spring+Hibernate（传说中的SSH）成为这一时代的web应用标配，JEE应用服务器曾经是“State of the art” 的技术，甚至成就了上市公司（例如WebLogic）。

- SOA + RIA：SOA最开始是互联网技术（例如HTTP、XML等等）生态回归企业IT，新的技术载体实现企业应用所需要的RPC语义、服务注册与发现机制，再结合Flash、AJAX技术实现的富客户端（Rich Client），实现企业类应用所需要的、比一般网站类应用交互复杂的交互。
- 在CAP定律制约下的分布式架构：Web 2.0时代产物，participation age 的社交网络、UGC，海量数据海量流量促成。
- 容器化的分布式架构：为什么这算的上一次里程碑式的运动？因为它可能导致ISV的软件产品（例如MongoDB、Redis等）被容器化、导致行业解决方案容器化（例如交易系统）、甚至导致操作系统容器化（例如RancherOS和其他“极简主义”下支持运行容器服务的操作系统），它促进DevOps工具链的发展，它和CI/CD深度整合，它直接影响开发人员的思考方式。
- Serverless架构：类似Amazon lambda、Google Compute Engine等，对于应用开发者而言交付方式部署方式都是有变化的，能否称的上“运动”，有待商榷，姑妄列出。
- 未知的运动与变革：技术的进步与变化，是加速度的，并且这个加速度本身的变化，也是指数级的，这是“奇点临近”作者库兹威尔（Kurzweil）的观点。总之，新的变革一定会来，快到你还没学会容器，新技术的“飞饼”已经摔到脸上。

其实，“软件开发”这个世界，依稀也是遵循“合久必分”（Divide-and-conquer）、“分久必合”（Combine-and-conquer）的规律的，例如：

- 从Mainframe向Client-Server到Web/3-tiered架构的发展，可以算是一个分层分治的过程。
- Client-Server架构下的系统一多，造成服务器端资源的浪费，然后以IBM、Sun、HP为首的厂商，在本世纪初开始推销所谓Server Consolidation的解决方案。随着虚拟化技术的成熟，物理服务器逐渐变成虚拟服务器，世界又回归一个逻辑上分布、物理上集中的“超级Mainframe”。
- 有些技术，往往是“似曾相识”，是新技术世代下用新技术手段对旧的理念的回归。例如SOA架构+RIA，是对C/S架构某种程度上的回归、虽然技术载体大不一样。同理，Micro Services也体现SOA的理念（虽然两者是巨大的不一样，见后文）。

容器化，也许算的上近年来最重要的“运动“，很快成为一种潮流 - 无论有无必要，开发工程师的代码都以容器交付，否则就像 Web 1.0 时代还在开发 Client-Server 甚至 Mainframe 的应用都不好意思出去跟人说。这潮流里，有厂商的有目的性推波助澜，有一窝蜂的赶潮流，也有切实的业务场景驱动，**无论如何，容器化将**（1）常态化 - 尤其是当 ISV 也把它们的产品容器化后；（2）促进分布式架构在传统企业 IT 里的采用（此前，大部分垂直行业 IT 并不擅长互联网企业所擅长的分布式架构，现在只要接受了容器的概念，显然就走向分布式）；（3）促进已经讲了 8 年以上的 DevOps 落地可操作。

说到传统企业 IT 的技术架构，就不得不提一下“去 IOE”，因为尤

	引爆点	开发技术载体	软件架构风格	客户端发布	服务器端交付
Mainframe	IBM System/360（第一代多用途计算主机），首次分离"架构"与"实现"的概念	打孔机、汇编、Fortran 77	一体	"笨终端" – Dumb Terminal	见过并且活着的人已经不多。。。
C/S	X86/PC、RISC、摩尔定律、HP/Sun/传说中的 SGI 和 NeXT 工作站	Unix、4GL、Sybase、高级语言、X/Motif、DCE RPC、DCOM、CORBA-IIOP	2 层架构、关系型数据库主导	软件 CD 安装、升级	软件 CD 安装升级、数据库迁移
Multi-tiered	互联网/Web	Struts+Spring+Hibernate、Tomcat、WebLogic、Websphere、Oracle、PHP、Ruby...	3 层至多层 – 展示层、整合层、业务逻辑层、持久层、存储层。。。	浏览器刷新一下页面	手工部署脚本、JAR、WAR、存储过程等等，数据库迁移。开始有 CI
SOA + RIA	互联网技术进入企业	SOAP、REST、Flash/AIR、AJAX、RMI/其他 Remoting、WSDL、UDDI。。。	对于用户像 C/S，对于开发者是 Multi-tiered	浏览器刷新一下、升级（例如通过 AIR）等	同上
Distributed	Web 2.0、NoSQL（BigTable）、云	函数类、动态、脚本语言，非关系型数据库，一致性算法（Raft、Paxos、Zookeeper），响应式服务器（nginx、Node.js。。。）	Reactive 响应式架构、Heroku 12-factors	多元化 – 手机 App、内嵌浏览器（例如 Webkit、Chromium）的富终端、网站	自动化部署 – Chef、Puppet、Ansible，CI/CD，DevOps 开始
Containerized	LXC、Docker	同上，但更规范（通过 PaaS 如 K8S – 遵循其最佳实践）	同上，但更规范	同上	同上，加不可变基础设施（immutable infrastructure）运维，加基于容器编排技术的 CI/CD
Serverless?	Amazon Lambda	脚本类语言更容易	透明	同上	仅需交付源代码

其在金融机构，IOE 是无可辩驳的存在。

容器化也许帮助你"去 IOE"

乍一听，这标题有点哗众取宠。但仔细想想，其实还真可以拉扯点关系。

首先，**"去 IOE"本身是一个伪命题**。企业 IT 降低对一些外国厂商和商业技术的依赖，固然从节省成本上有那么些好处，但如果不上升到"民族产业"、"信息安全"的层面，减少几个数据库软件的 license 对于企业本身的效益是有限的，其伤筋动骨的技术迁移也许是得不偿失的。去了 IOE，也没什么值得自豪 – 别高估了自己在国家信息安全方面的重要性，如果没有为业务经营、客户利益带来价值，恐怕只能是"然并卵"。

如果我们不带偏见的把 IOE 看成一些象征性的符号而不是具体的某些公司的话，IOE 一定程度上代表了上一个世代的技术，对于只生存在开源技术世界里的互联网企业的年轻工程师尤其如此。IOE 甚至在一定程度上是 Client-Server 架构思潮下的产物，代表了以关系型数据库为中心、以中央存储阵列为主导、以品牌服务器硬件为载体的技术架构风格，这类技术的存在依然有充分必要的业务应用场景，盲目的“去”，只能是自找麻烦和浪费资源。

但换一个角度看，“去 IOE 思维”却又是有意义的，因为在实践中我们已经发现：

- 关系型数据库被企业里的应用开发者们过度滥用。事无大小，都被存入数据库，包括一些配置信息。事务型操作往往也没有控制好粒度，开发者为了“稳妥”起见囫囵吞枣的把CRUD操作都丢到事务里，期望由数据库来解决自己的不求甚解和懒惰。
- 很多问题，其实是可以不用关系型数据库解决的。
- 互联网时代尤其是Web 2.0开启后，从3-tiered向分布式架构演进，RDBMS为中心、高端硬件为依托的架构已经力不从心。
- 就算不搞互联网，传统业务系统在当今这个时代沿用旧的技术架构依然扛不住，2015年疯狂的股市下，高频、高并发、海量的交易就是股票交易系统的梦魇。

“去 IOE”其实最难的是观念的改变，传统企业 IT 的工程师，非常习惯于用关系型数据库的语义、概念作为对业务领域（business domain）的建模工具，一言不合就开始设计表结构、画 ER（Entity Relationship）图。开发过程中，可能大部分时间消耗在 ORM（Object-Relationsal Mapping）上 - 从数据模型出发封装一些对象以便于数据持

久层和内存之间关联起来。这导致传统 IT 系统的升级动辄涉及数据库迁移，功能扩展通常导致表结构改变。**实际上用关系型数据库的理念对世界进行建模（modeling），是有很多局限性的**，一是无法对业务逻辑进行抽象，二是无法对业务数据进行封装。这样做的缺点，是所建立的模型无法低成本扩展、重构，以快速应对持续变化的业务场景，在当今这个“只有变化才是唯一的不变”并且变化频率本身是指数级改变（《奇点临近》作者 Ray Kurzweil 所言）的世界，这样的设计导致的显然是一个变更成本非常高的“脆弱系统”，无法拥抱改变应对黑天鹅（关于脆弱系统，见塔勒布《反脆弱》）。

“世界观决定方法论”，中医和西医对人体的建模差别，决定治病的方法截然不同。例如前者用经络、寒热、干湿、虚实、阴阳来描述病理，后者用细胞、基因、细菌、病毒来看待问题，导致同一个疾病的不同处理手段。有些问题用这种模型来看容易解决，有些问题则用另一种模型描述更有效。盲目的用关系型数据库看待一切，是很多问题的根源。

以关系型数据库为中心的应用，一般都是单体应用（monolithic），虽然它们可能也会融合一些分布式的技术元素，扩容、扩展、弹性伸缩、响应变更等等这些非功能性需求，依然是它们所难以满足的。在看到这类架构的问题后，技术界开始出现混合编程（polyglot programming）、混合存储（polyglot persistence）和混合处理（polyglot processing - 例如大数据里的 zeta 架构）的潮流，微服务（Micro Services）的架构风格与理念也逐渐形成。

微服务具体实施的问题在于，停留在“理念”、“最佳实践”层面的东西，很难在一般垂直行业的 IT 落地，因为面向业务的工程师们，关注点不在底层技术细节，无法投入资源去研究自己的平台，凡是能在垂直行

业推广的技术，必须是具体有形的工具、API、框架。容器类技术作为看得见摸得着的、同时被运维人员和开发人员使用的工具链，对微服务的开发和运维，提供了无比巨大的推动力。虽然微服务本质上不依赖于容器，但是没有容器技术的支持，微服务在一般企业 IT 里的落地是不乐观的。

这就产生一个非常有趣的副作用，一旦技术人员习惯了容器化的观念，他们很可能不知不觉就走上了分布式架构的道路、潜移默化接受了微服务的思维，我们知道技术人员是很容易“心为物役”的，他们的抽象思考往往需要寄托在有型的工具上。例如 Heroku 的 12-Factors（12 律），总结了 12 项在云上开发分布式应用的最佳实践，我们可以看到，采用容器化的架构，很自然的就吻合这些实践。

总而言之，“去 IOE”从它最开始的起源来看其实是去单体应用架构、去“数据库中心主义”，是分布式架构对传统企业技术套路的颠覆，而容器化一旦成为主流，分布式架构即会潜移默化不知不觉中成为企业 IT 架构的主流。

不要把微服务和容器化本末倒置

容器化既然被说的这么玄乎，那么是不是我们就该蜂拥而去的采用？个人认为，如果阁下的企业环境，并无特别适合 “微服务化” 的应用，那么个人揣测，阁下也并无采用容器技术的必要，即便是已经采用了 SOA 的技术架构与技术治理，千万别以为就顺理成章可以换一个时髦点的名字“微服务”。

SOA 与微服务，其实有一些本质区别，哪怕是表面上有一些相似性。例如都叫“服务”（技术名词有时确实导致巨大的歧义）、都有服务注册与发现机制、甚至具体实施技术有一定重叠。在此列出一些区别（有商榷

处，姑妄列出供讨论）：

- 从根本思想看，SOA是强调中央治理（central governance）的，服务之间大致松耦合但实践中其实有一定耦合，例如shared storage是常见的，同一个数据库上封装几个服务，避免数据直接暴露到外面，可后面依然是一个数据库；事实上当用到“治理”（governance）这样的字眼，本身就充满了中心化的意味。微服务则以去中心化为原则，强调share nothing、服务间高度松耦合，数据持久层被抽象为backing store，甚至不需要是关系型数据库，每个服务各有自己的backing store。
- 总体架构设计，SOA是一个top-down思维，“顶层设计”，从业务切分模块，把模块之间关联变成封装服务之间的调用、集成。微服务是一个自下而上的bottom-up的思维，服务的粒度更小（否则何为“微”服务？），对服务本身上下文的假设更少，最重要一点，微服务通常是从当前服务的上下文衍生出来的。
- 组织结构上看，SOA类型的项目团队，模块负责团队隶属于一个更大项目之下，实际中几乎不会独立运营运维。微服务，粒度小，强调单一责任，独立部署运维、自有生命周期。某种角度看，不同SOA项目之间是不同的Silo型团队负责的，而微服务严格来说一个服务对应一个跨职能团队。
- 从关注点看，SOA强调的是SLA、compliance/regulation（合规）、audit（审计）等大型企业特色的“治理”，微服务关注点从来都是快速响应客户要求和市场变化以及快速创新，是敏捷型的。（所以微服务并不替代SOA）。
- 从部署运维角度看，SOA出现于云计算之前，自动化部署、自动

化运维并不是它的天然基因。微服务几乎可以说是云计算时代的产物，高度碎片化（与SOA型服务比），高度依赖于解决底层非功能性技术问题的PaaS/CaaS平台，天然符合多租户、需要DevOps支持。

一个微服务通常很可能是通过从现有服务 fork（开分支）、clone（直接复制）、mutate（变种）出来，这很有可能是违反我们在软件工程中一个所谓 DRY（Don't Repeat Yourself）的原则 - 我们已经习惯于认为，剪贴代码是糟糕的、粗暴复制功能不好的，在理想主义的软件王国里，绝对没有拷贝粘贴的代码，也没有冗余的数据，更不应该有复制的服务。

然而，我们对世间事物的认识，往往是螺旋上升的，没有绝对的赞好、也不能武断的说坏，在现实这个不完美的世界里，除了编程大拿、代码洁癖者、技术原教旨主义者，我们必须接受一个现实，就是只关注快速实现业务需求的程序员、普通运维工程师是大多数，粘贴代码、复制部署服务可能就是大部分以业务功能为终极目标、以快速上线为首要任务的普通工程师的本能，quick-n-dirty 是任何团队绕不开的取舍。微服务，通过结合容器技术，一定程度上接受了普通程序员克隆服务、克隆代码的“陋习”。

- 系统升级，在条件允许的情况下，运维工程师最喜欢的可能是部署一套新的，旧的不碰。新的没问题，把旧的关闭；新的有问题，把旧的切换回来。微服务天然是考虑支持同一个服务的多个版本并存的，而容器则刚好是实现“不可变基础设施”（Immutable infrastructure）的最佳套路，这俩一拍即合。
- 同一个服务，也许会逐渐演变成需要接受不同的运营约束（operational constraints）、或者针对不同的用户群，是不断收集新需求、重构代码、升级服务以保持单一服务支持不同业务需

求？还是复制代码、克隆服务，在不同环境各自独立发展？对于快速敏捷支持不同的业务线、产品线，也许复制代码克隆服务是一个更高效的做法。

Michael Nygard，一位曾任职私募股权交易机构的架构师，在他的“新常态”系列技术文章中，举了一个例子：他的公司有四十多个不同的交易席位（trading desk），分别在不同的市场采用不同的策略进行交易。每个交易席位都有自己的技术团队负责交易应用开发。如果他们像很多大企业一样采用一套单一的、集中式的交易系统，则任何交易组对系统的变更均会对其他组产生潜在的损害 - 因为变更影响他人、bug产生系统性风险、测试需要更繁复覆盖更充分、变更发布周期需要更长、升级需更复杂慎重、交易系统受影响面更，他把这些潜在能导致失败的影响称之为“失败域”（failure domain）。

Michael 的雇主选择让每个交易组独立维护自己的小型、单一、功能聚焦的交易应用 - 开发工程师和交易员坐在一起工作、小团队作战、快速迭代，以此来最大程度缩小各交易组被动关联产生的“失败域”。这个场景，对于从事证券交易系统开发的工程师，是非常熟悉的。在这里我们看到两个极端的取舍：

- 交易系统从架构和基本功能的角度上看都是大同小异的，以一套理想的、大而全的、集中式的系统服务各条业务线、各个交易市场、各种交易产品，好处也许是集中运维统一监控，服务归一、数据完备，消灭了信息孤岛，公司能获得跨市场、跨产品、跨业务线的最完整的经营数据，轻易实现合规监管、统一风控。但是这种中心化系统，本身的任何变更都是牵一发动全身，任何缺陷都导致公司级风险，是一个典型的“脆弱系统”。也不利于任何业务线的单独敏捷运作。

- 类似上述私募股权公司的做法，则是完全去中心化，多套交易系统冗余建设，在一定规模的证券公司里，几十套交易系统是常见的，确实产生很多问题 – 例如一个合规要求或交易市场的新业务，必须在几十套系统里变更升级，硬件资源得不到充分的共享利用，不同交易系统往往是异构技术形成一个个竖井（Silo），信息孤岛的形成给统一风控统一经营造成巨大困难，等等。然而，这个去中心化的做法，却是符合”反脆弱“精神的，它把失败域变小。每条业务线有自己的交易员、业务专家、工程师以及系统，响应市场竞争的效率最高。

作为一个例子，微服务架构很可能对上述情形的解决是有帮助的。首先，在实际操作中，去中心化基本上是共识，避免全局、系统性风险比什么都重要，所以功能相同相似的服务在不同业务线、不同约束条件、不同目标用户环境下冗余部署是必须的；其次，越复杂、越大块头、越黑箱型的代码模块，越难弄明白，越怕被触碰，也就隐含越高风险，微服务化使之增加透明度；第三，一些服务例如交易引擎，架构大同小异但是细节很不一样，与其反复抽象、重构、支持一切交易市场、交易产品、资产类型，还不如克隆一下，把相互依赖以及对某些共同设施的依赖均降到最低，各自迭代发展，让每条业务线获得最不相互掣肘的、最高效率的技术支持。

但是正如我们常说的，“软件工程没有银子弹”，微服务架构带来的更多的碎片化，对架构设计、开发运维挑战更大，对开发者技能要求更高，例如需要掌握一系列 Cloud-native patterns（原生云架构设计模式诸如 throttling、circuit-breaker、bulkhead 等）。容器技术作为一种工具链和微服务载体，则在此时发生了很大的促进作用，甚至是微服务化实际落地的可行性的一个重大保障。

显然，正如上述例子，我们是因为考虑微服务化单体应用而考虑容器工具，平白无故容器化一个系统则是毫无意义的，“容器化与否”本身也是一个伪命题。事实上，脱离业务特点谈“微服务”本身也是个伪命题 - “如果你都不能构建一个有效的单体应用，你凭什么认为微服务能解决你的问题？”（Simon Brown，“Distributed big balls of mud”）。

采用微服务类架构，则需要调整一些“传统”的观念，例如关系型数据的高度归一性（Normalization）、代码的可重用性（Re-usability）和系统的精益化（Lean），可能不再是无可辩驳的“美德”：越归一则数据关系的变更成本越高、代码越可重用则模块组件的依赖关系越复杂（dependency hell）、越精益的系统则变更越困难，这些都是对构建”反脆弱“型系统不友好的，而且很多时候，强韧性（Resiliency）、灵活性（Flexibility）和效率（Efficiency）之间是有冲突和代价的（还是 Fred Brook 的银子弹理论）。

事实上，**随着大数据技术发展应运而生的 NoSQL 运动，一定程度可以说是对高度强调归一性的传统关系型数据库理论的“反动”**。在微服务的思潮下，我们的服务首先是不共享任何东西（share nothing），每个服务可能都有自己的持久化存储（backing store），形成所谓的混合存储（polyglot persistence）；其次，基于领域建模的开发（Domain Driven Development - DDD），对于实现微服务更加重要，以完整业务逻辑为中心，并解耦了对开发语言、数据存储技术等等的假设。虽然单体架构或者微服务架构更恰当的说只是不同的架构风格而不是架构本身，但是，一些开发语言、技术工具确实是更适合或者更导致实现出单体架构系统的，例如上述这位 Michael Nygard 先生就认为，传统的 Java 企业应用服务器市场里的技术，包括 Oracle Weblogic、IBM Websphere、Redhat JBoss 等等，

是“鼓励”开发单体架构的技术，这又回归到本文关于“IOE”技术风格的讨论，分布式架构、微服务、容器化，是脱离技术分层（layering）的单体架构风格的，和大部分企业 IT 所熟悉的技术生态截然不同。

微服务、分布式架构是比单体架构更复杂的，可移动零部件特别多，零部件之间的关联关系复杂、通讯耗损大，为了解决这些问题，技术团队需要掌握此前不需要的知识 - 例如上述的 Cloud-native patterns、Heroku 12 要素、Reactive 技术风格与技术工具等等。一个企业如果没有具备这些知识与能力的技术团队，都不应该去考虑做微服务。

“架构风格”、“开发理念”、“最佳实践”、“技术原则与技术哲学”等这些，通常只适用于优秀的技术团队，对于普罗大众的、以业务功能为开发目标的团队而言，也就是听完一次、佩服一下，结束。**只有通过工具才能把这些抽象的东西实际落地，否则它们也就只好留在教科书、别人的 PPT、网上文章里**。幸运的是，容器工具链的发展，也许正在成为这么一系列工具，会促进微服务在传统企业 IT 的落地。

会计准则、代码“库存”与进化式架构

在微服务架构的思维下，归一性（nomalization）不再是一个绝对的“健康目标”，服务们通过克隆、变异、分叉而诞生，然后迭代发展、存活、消亡，在这过程中冗余、重复是被允许的，而淘汰则是必须的，所谓“有生有灭”，有用的服务（被使用的频繁、被高度依赖）存活，没用的服务关闭，我们设计微服务不是在一张白纸上凭空绘画，而是不断在现有的服务中派生、演变。这种架构，可以称之为进化式架构（evolutionary architecture）。

进化式架构有一个有趣的作用，就是代码“去库存”。这里说的”代

码库存“，不是指代码的技术载体 – Git、SVN 等代码库，而是一家企业多年开发积累下来的代码，究竟是“无形资产”（Intangible asset）还是“债务”（Liability）和“库存”（Inventory）。根据美国的会计准则（US GAAP），一家企业的内部自研发软件费用是这么被划分的（粗略罗列）：

- 项目启动前的相关内部和外部费用，被认为是开销（expensed）；
- 自研发供自己使用（internal-use）的应用软件所投入的相关外部和内部费用，被认为是资产（capitalized）；
- 让新系统接入历史数据、或者转换旧数据格式，所投入的费用，被认为是资产（capitalized）；
- 系统投入运营后的相关培训、运维费用，被认为是开销。

显然，一家企业 IT 的 R&D 产出，被认为是公司资产，软件研发的投入是一种投资，所以其产出软件承载着一种期望 – 增加企业的生产力。但软件毕竟不像机房、机器、网络设备那样看得见摸得着，对于大部分的企业管理者而言，无形资产恐怕总是有点说不清道不明，在这个连软件都不再以盒装 CD 加大部头手册的方式卖的时代，IT 管理者们量化自己的无形资产的一个本能，就是自豪的宣称，自己的系统含有多少百万行代码。可是，这几百万甚至上千万行代码，真的是资产吗？

首先，**在现实中，一个系统的代码量基本上是逐年增加的，对于大部分团队，系统维护就是增加代码，真正有动力和能力对自己的已上线系统进行重构的团队是很少的**，当一个团队跟老板报告系统代码行数下降百分之十的时候，该老板除了疑惑不解之外可能还有恐惧不安，业务部门和用户也不会为节省了几十万行代码而感谢你给你发奖金。可是代码越多的另一个潜台词，是风险点越多。有些风险是程序员引入的，有些风险，则是

因为世界发生了变化、原来的运行环境业务规则发生了变化，原来的假设忽然不再成立，例如用一个十年老的网站改造去支持智能手机出现后的移动应用，原来已经稳定的架构和逻辑在修修补补后就产生新的缺陷。总之，世界的变化总是快于软件的进化的，而积累了十年的几百万甚至几千万行代码，总体来说是很难对变更友好的。这个时候，代码库就变成了库存、技术债、阻碍业务创新的惯性。

过去十几年来的企业软件，我们可以认为大部分是单体架构的，它们中的许多是内部逻辑铁板一块的交织着，甚至没有做到代码级别的松耦合（例如通过良好的面向对象设计与设计模式实践），陷入依赖关系地狱（dependency hell），无用的代码很可能继续存活在系统中从来没被淘汰。新工程师对老系统是拒绝的，因为技术套路和理念两年就过时了；用户对老系统内心是崩溃的，因为让 IT 去维护修改它的效率永远是低下的。作为程序员，我们每个人心底里都讨厌接手他人的代码、抱怨前人的愚蠢设计或怪咖风格，我们基本上倾向于认为历史遗留代码（legacy code）是阻碍生产力的，总是恨不得除之而后快。

例如我们为了不触碰前人的代码，通过模拟接口绕开旧代码、换上自己的实现，然后旧代码就成为一段 dead code，但谁也无法确定它是否死透，因为它也有可能在其他地方被调用，于是谁都不敢把这段代码删除，于是代码只做加法越积累越多，而后来者则理解越来越困……可以想象，**依赖这样的系统去创新，就像戴着脚镣跳舞**。金融行业的应用系统，有很多这样的东西，庞大而沉重，日益成为厌恶害怕变更的“脆弱系统”，“我就是喜欢你看我不惯却不得不和我一起建设社会主义的样子” - 面对这样的技术库存我们很无奈。

普通程序员（尤其是被业务部门蹂躏着、无暇学什么理论的）的“本

能”偏好和习惯是这样的：情不自禁的剪贴代码； 上线新功能最好部署整套系统而别让我折腾里面的细节，出了问题马上切回旧的那套 - 还好好的在那跑着呢； 永远喜欢做新项目写新代码，而不是去消化前一个傻帽的代码； 有些代码就是没法子写的优雅的，尤其是临时性运营性的代码，能不能用完即丢… 别轻视这些草根的、不高大上的“陋习” - 简单的、符合“本能”的、有“群众基础”的东西，才往往是有生命力的。

微服务架构风格之下，**也许更符合草根习惯的开发平台终于有机会出现：高度碎片化，大家都在写小组件小程序**，除个别关键服务外一般难以对整个系统产生坍塌风险，任何小程序可以被丢弃被重写，小程序们可能多个版本同时存活但是监控程序会告诉我们哪些已经逐渐没人用可以被销毁，面向一个有略微差别的新客户也许就克隆一个新服务稍微修改一下、避免修改现有服务增加判断条件。微服务基于业务需要、市场变化、经营策略而持续不断的出现和消失，这就是一个像流水一样的持续变更中的有生命力的系统。

当然，前提是我们指望容器编排技术结合监控技术和 cloud-native 的各种架构模式把服务底层的运行平台搞定，所以一个强的平台团队依然是需要的。把一个一百万行代码的系统拆成 100 个一万行代码的独立运行协同工作的小程序，绝对不是一件随便可行的事。

当我们有非常好的工具以支持我们在短时间内快速开发出代码、极大程度释放生产力的时候，我们才敢于丢弃过时代码，让服务派生与进化、用完即扔，才可能不再把历史积累的代码当作什么珠宝般的贵重资产，才不再拖着充满风险的“库存”前行。

从生产工具到生产组织关系

新技术带来新工具，容器只是又一种工具而已。然而，鸟枪换炮不仅

需要相应的新操作技能与观念，还需要与之相适应的组织结构，工具使用者之间的协作方式、运作流程、管理模式都需要作相应变更 – “二炮”变成“火箭军”，肯定不仅是改个名字那么简单，是和先进科技武器的发展相适应的。自从微服务开始流行的这两年来，网上越来越多架构师在讨论康威定律，不是偶然现象，是大家不约而同意识到，服务的切分、模块的解偶、技术系统的最终形态，很大程度取决于开发者所在组织的交流沟通形态。

单体架构应用开发的团队，很可能是这么一个 silo 团队：一到多个界面及交互设计师、几个前端开发（Web 1.0 时代，是 HTML 及模板编写者；Client-server 时代，是 MFC、JFC/Swing 甚至 X/Motif 开发者）、几个服务器端开发（EJB/JSP 的、更古老一点是 CORBA/IIOP 或者 DCOM 的）、一两个数据库的 DBA。这种团队虽然有多个角色对应多个技术层，但严格来说不算“跨职能”，因为他们基本上既不运营也不运维自己的应用，业务人员和运维人员并不在团队内。为什么称之为 silo 团队呢？因为多个这样的团队之间，信息是不透明不流动的。

微服务架构下的组织结构，适应如下特点：

- 有Gartner所谓的“外架构”与“内架构”之分，内架构主要围绕微服务的标准化设计，例如每个服务必须是实现单一责任（single responsibility）、服务边界（scope）清晰、接口约定（contract）明确稳定。外架构主要是平台，聚焦于系统性解决基于内架构的微服务实例的注册发现、编排、资源伸缩、生命周期管理、监控、高可用等，解决服务碎片化带来的各种共性问题。
- 平台团队负责平台的构建、优化、维护，持续整合新技术、持续向服务开发团队提供工具与公共设施便利，例如利用Kubernetes、

Swarm或者Mesos之类的技术构建容器云支持多租户，基于行业（例如金融业）特有环境与要求定制网络与安全解决方案，用符合自身企业环境的技术解决Docker的网络问题，诸如此类。微服务团队则由业务专家和工程师组成，联合维护和保障一个服务的功能、有用性（没人用的服务就没有生命力）、运维，其中工程师很可能是全栈的、开发自运维的，虽然系统层面的高可用由平台团队支持，可是服务运行中的业务逻辑故障显然只有负责开发的人自己直接搞定。

- 平台团队主动向企业各业务线的团队布道、宣传自己的工具与平台，争取更多的租户。而微服务的团队，同样需要向其他部门、团队布道、宣讲自己的服务，找到更多的应用场景和使用方。最后，各业务线的应用开发项目，则是在产品开发过程中组合、集成各种服务（同时也可能提供自己的微服务）以服务终极客户。所以，这是一个服务型的、“人人为我、我为人人”的组织。衡量这些团队的表现也好办，看看平台团队争取了多少租户、微服务团队支持了多少应用……

结合容器技术，基于微服务架构的组织，都可以成为DevOps型组织，因为容器工具链也许是迄今为止真真正正让开发与运维共同使用的同一套工具，把开发、测试与运维工程师通过CI/CD串在了一条协作生产线上；此前虽然有Puppet、Chef、其他所谓实现“infrastructure as code”的工具，实际上不同角色的工程师依然是没有标准化的工具分享的。别少看了共用一套工具的厉害之处，工具所附带的整套技术语言、概念、词汇表，往往成为工程师们交流沟通、描述问题的标准语言，否则他们往往是“鸡同鸭讲”，同一个词汇说的完全不是一回事。

虽然我们一直强调工程师的抽象思维，避免“心为物役”（过度依赖于某种技术、某个开发商的系统所定义的概念词汇来作为自己分析思考问题的基础），可是实际中具备良好抽象思维的人并不多，所以依靠一套比较好的工具作为技术解决方案领域（solution domain）的思考依据，也不是一件糟糕的事情。

掌握新技术新工具，需要传统企业 IT 的组织思维与时俱进，作相应调整、重构。否则，无论打着“互联网 +”的旗号还是标榜“科技”（例如所谓“科技券商”或者“金融科技机构”等等），都有点“意淫”的味道，生产组织基因不对，无法有效使用科技生产工具。实际上，任何复杂业务应用系统的架构，都涉及两大因素：技术和人。不考虑人与组织因素的架构，有伪架构之嫌。

纯粹卖容器技术解决方案的厂商，恐怕生意不好做，因为光卖武器而没有相适应的组织结构与操作方式，武器很难推广，尤其是现在传统行业甲方的 IT 恐怕依然没有把虚拟机和容器的本质区分好的环境下。

企业级 IT 向左、科技独角兽向右

企业级的 IT 方案，往往突出一个“大”字：总是庞大而沉重，总是大而全，总是显示出一副很强大的样子，它们强调的，往往是“治理”、规范、流程、管理……而不是互联网独角兽们所聚焦的敏捷、把握市场机会、以快打慢、以创新迅速争夺客户……这两种理念的区别，体现在对变更的适应程度上。

仅就金融科技这个领域而言，技术能适应变更甚至从中获益壮大是非常必要的。Zvi Bodie 和 Robert C.Merton 合著的金融领域经典著作《Finance》一书，定义诠释了金融学：“金融学是一门研究人们在不确

定环境下如何进行资源跨期配置的学科”。Nassim Taleb 则在《Anti-fragile》中从金融世界开始讨论如何从无序（disorder）中获利。可见金融世界天然关注不确定、无序、随机、紊乱，金融机构的本质是通过风控从不确定中获利，这就是 Taleb 所说的“反脆弱”。金融机构的科技，是否也需要具备“反脆弱”基因与之相适应？

有技术同行以适应变化的能力、喜爱不确定性的程度为标准，把公司分成两极，一端是拖着几千万行代码“库存”的“雷龙”俱乐部成员们，另一端是轻资产、迅猛、快捷逮住市场机会的“独角兽”们。侏罗纪的雷龙们长五十米重三十顿，最终在没有在物竞天择、适者生存的游戏中存活下来。

独角兽（一个由 Cowboy Ventures 创始人和风投家 Aileen Lee 所发明而被投资界广为使用的概念，专指发展迅猛、所做的创新为世界所未见、具有高度价值的公司），则是不断颠覆行业让雷龙们疲于奔命。一个把自己变成雷龙的例子是 GE，它把绝大部分的软件开发都外包出去。问题是，软件开发商的收入模式就靠替客户开发 - 开发任务越多、工作量越大、代码越庞大越好，所以，“不管你需要还是不需要，它们会继续替你构建东西”（Michael Nygard “新常态”文），最终这些代码形成一个“价值 15 亿美元的船锚，并且被咨询顾问们镀了一层金以显得有价值” - 但实际上这 15 亿的锚不一定是你的资产，也可能是把你的船拖沉的债。

扯远了，脑洞有点大。回归到企业级 IT 的技术方案话题：作为一个在互联网企业和创业公司以及传统 IT 的甲方乙方都呆过很多很多年的技术人，个人虽然一本正经严谨讨论“企业级”解决方案，可是心底里喜欢的是互联网自由散漫那套，那就是开放(open)、轻量(light)、敏捷(agile)、精益（lean），而不是“企业级套件”、重型中间件和大中央数据库技术

的粉丝（它们确实有存在的道理，用的恰当的话，但个人喜好无关合理性）。个人粉的以下例子，上升一下下到哲理高度，是符合极简主义、精益的理念的。

- UNIX，以及此后的Linux，操作系统保持小的内核，开放接口，由大量小程序形成各种小工具，完成更复杂的任务可以通过脚本来粘合各种小工具。
- REST，在SOAP、UDDI、各种“WS-*”前缀的企业标准把Web服务搞的无比沉重后，REST真是一股技术清风，用最基本的HTTP原语，更简单透明轻量的描述了服务。
- Mechanical Sympathy，这个由Martin Thompson等人提倡的理念，指出要把一个高性能的程序做好，程序员必须对计算机底层硬件的基本原理（例如CPU的缓存机制）有一个认识，才能通过合理的数据结构与算法设计，去发挥运算能力（现在的企业软件开发，更像“考古”工作，应用开发者的代码只有一点点，下面是第三方框架层、JVM虚拟机层、并发线程库、操作系统的系统库、操作系统内核… 顶层的工程师可控的东西非常少，大部分人对计算机原理不求甚解）。

微服务加容器，和小程序加 UNIX，是否有那么点哲理上的可比性？正如写 UNIX 小工具的人需要直接了解操作系统，写微服务的人需要直接了解一点容器与容器编排、原生云架构模式 - 这是开发者真正第一次介入云计算，此前基于虚拟机的云计算只对运维有意义，对开发者透明（试问此前有多少开发工程师在自己的应用系统里调用过虚拟机的 API，基于业务需求控制云计算资源？又有多少人在意自己的代码跑在物理机还是虚拟机里？）。虽然不久的将来，容器技术很可能又被大技术厂商们搞成一

个个冠以各种高冷术语命名的、不透明的“企业平台”。

套用网上“鸡汤”的常用语，技术的“初心”是提高效率增强生产力。不想变成雷龙的企业，需要引入科技工具。以前建房子用铁铲挖泥，现在用挖掘机。那么问题来了（还不是“挖掘机哪家强”，还没到那份上，挖掘机性能可能都差不多……），　一个十年前或者更久远的企业组织、生产关系，能有效掌握新生产工具吗？

微服务、容器神马的，都是浮云，这种现象级的技术浪潮会一浪接一浪，等学会了，又已经过时。所以，本文想扯的观点其实是，去一家“兄弟单位”调研容器技术怎么用、和技术界大牛学习微服务最佳实践…等等，其实很可能没有什么显著效用，因为很快你会发现超容器、超超容器、去容器化又出来了，当我们总算丢弃了青铜剑进入汉朝斩马剑的时代，发现别人已经在舞弄连人带马都能劈碎的唐朝陌刀。

最可怕的是，还有几个哥们已经在比划着玄铁剑、叫嚣着“重剑无锋，大巧不工”的理论；但最最可怕的是，还有哥们已经在玩剑气搞“无招胜有招”… 按照金庸金大侠的理论，最根本的还是内功。企业 IT 的内功是什么的？个人认为是组织结构、体制、文化，在这个时代，只能通过构建有科技基因的、对技术友好的组织，让团队和新技术共同成长（而不是等它成熟再去向“兄弟单位”学习），保持精益的技术文化和理念，才能“去库存”，稍稍具备一点独角兽的特质——Swift（迅捷）、Nimble（灵巧）。

从 Redis+Lua 到 Goroutine 日均 10 亿次的股票行情计算实践

陶瑞甫

股票行情数据是一种典型的时序数据（Time-series Data），在一般的 IT 系统中，日志数据其实也是一种时序数据，在大数据的世界里，也有大量应用是基于时序数据处理的，可以说时间序列的数据无处不在。

所以，哪怕是不炒股、不熟悉金融世界的工程师，从本文也可以了解一些具有普遍性的技术思考，例如在大规模、高密度的数据处理中，是把数据快照搬到计算节点作运算还是把计算能力放到数据节点中“就地计算”？协程（co-routine）在这类计算密集型系统中又有何作用？

实时行情服务是券商的基础服务，给普通投资者描绘出风云变幻的动态市场画面，也给量化投资者提供最重要的建模基础数据和下单信号，时间就是金钱，行情服务必须要快。

证券交易系统行情指标很丰富，最基础常见的包括：行情报价、分笔数据、分时数据、分钟 K 线、日周月年 K 线、各类财务技术指标、多维度排序、多维度统计等等，这些指标需要由交易所的行情数据流结合时间流

进行统计计算。本文分享广发证券行情服务并行化计算演进的过程，包括五个部分：

- 股票行情是怎么回事
- 日均10亿次的行情指标计算
- 基于Redis+Lua的方案：“就地计算”
- 引入Goroutine的方案：“海量算子”
- 孰优孰劣

股票行情科普

任何交易在达成之前，通常都有一个讨价还价的僵持过程，或者买方让价，或者卖方让价，两不相让的时候需要第三方介入撮合取个“平均价”，这个过程就是定价。

证券股票交易也不例外，不同的是买卖双方互不可见，双方通过券商渠道把自己的价量报给交易所，交易所通常按照达成最大成交量的原则撮合定价。撮合涉及 order book 和 tick data 两个概念。交易所维护两个“账本”分别记录买方和卖方申报的价量，实时或按照一定频率进行撮合定价成交，申报、成交时对“账本”进行增、改、删操作。

这两个“账本”就称作 order book，对 order book 增改删操作引起数据变化，每个变化的快照就称作 tick data。国内股民数超 1.2 亿户，A 股上市公司近 3000 家，可以想见的到 order book 是一个大“账本”，tick data 瞬息万变，一个交易日产生的 tick data 量更是惊人。

国内交易所目前采用抓取快照的方式，抓取 order book 的前 5/10 档价量，剔除”无用的”其他档价量数据，统计交易当日开市时点到目前时点的最高、最低、成交量、成交额数据，我们将这种数据称为原始行情数据。

交易时间	股票名称	股票代码	昨日收盘价	今日开盘价	买一价	买一量	买二价	买二量	买三价	买三量	买四价	买四量	买五价	买五量
2016-07-22 14:43:41	广发证券	000776	16.84	16.78	16.76	16000	16.75	4200	16.74	60019	16.73	60500	16.72	122300
目前最新价	目前最高价	目前最低价	目前成交量	目前成交额	卖一价	卖一量	卖二价	卖二量	卖三价	卖三量	卖四价	卖四量	卖五价	卖五量
16.77	16.86	16.70	19195026	322024800	16.77	21200	16.78	13541	16.79	44100	16.80	40200	16.81	35600

交易所将原始行情数据近实时的发送给券商，券商行情系统对这份原始数据进一步处理，比如按时间区间统计出 5 分钟、10 分钟、15 分钟、30 分钟、1 小时、1 天、1 周、1 月、1 季度、1 年这十个周期时间段 K 线蜡烛图的高、开、低、收价格及成交量、成交额数据。

券商将处理好的行情数据揭示给投资者做再报价参考。可见这类数据量大、变化频繁、中间统计计算耗时，一旦延迟，以后就再也跟不上“实时”的节奏了。又快又准，是股票行情服务的基本要求，任何误差、延迟，都可以引起交易者的经济损失，导致投诉甚至社会事件。系统该如何设计才能高速的处理和展示这类实时行情数据，是个很大的技术挑战。

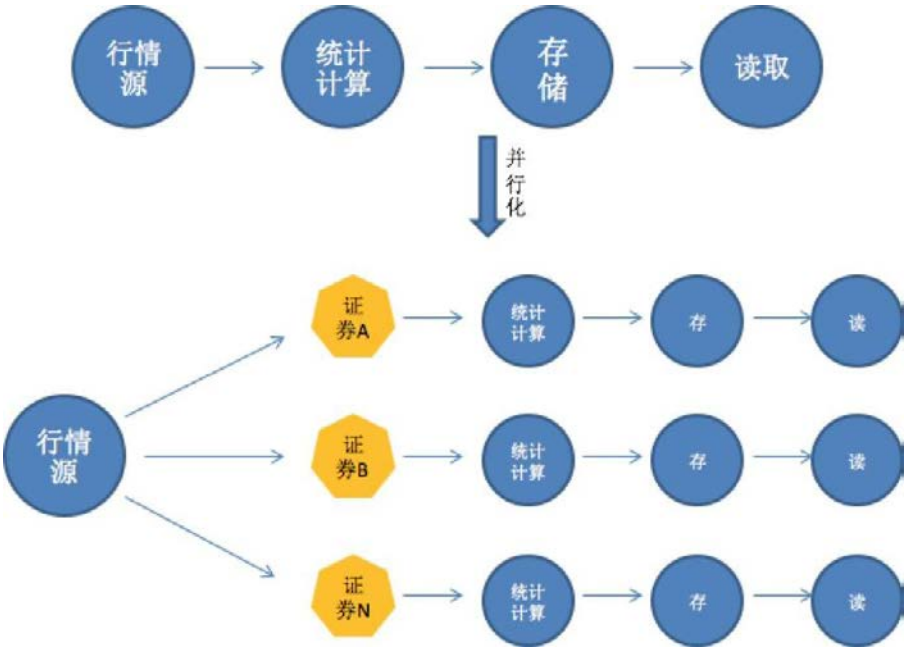
你们股炒的爽，我们指标算的酸爽

券商行情系统可以分为行情原始数据接收解析、中间指标统计计算处理、行情数据请求响应三个模块。

原始数据接收解析模块只需要做好与不同交易所的行情消息格式适配即可，行情数据请求响应模块要达到原始行情报价及各类行情统计指标的快速展现需要减少中间处理步骤，在行情原始数据接收解析、统计指标计算完成后立即推送给终端用户，同时存一份到内存数据库中以便终端用户再次查询读取，推送与查询两路相结合，达到在数据落地之前即已展示给用户的效果。

剩下要做的就是要尽量减少中间指标统计计算的处理时延。A 股上市公司 3000 多家，基金加债券数更是上万，但它们之间是无关联的，在统

计计算时完全可以分片、并行处理，存储上采用内存数据库，redis 内置多种数据结构的支持满足多统计指标存储的需求、容易分片部署便于并行计算。



以最简单的十个周期时间段的 K 线统计指标，证券数保守算 1w 只，10*1w=10w，也就是说分分秒秒就有近 10w 的计算量，国内沪深交易所一个交易日开市 4 个小时，按照交易所行情数据每 3 秒更新一次，可得出日计算量就有 4.8 亿，加上其他指标计算如市盈率、涨跌幅、换手率、委比委差、多板块多指标排序，日计算量已突破 10 亿。

“就地计算”的方案——把计算能力放在数据里

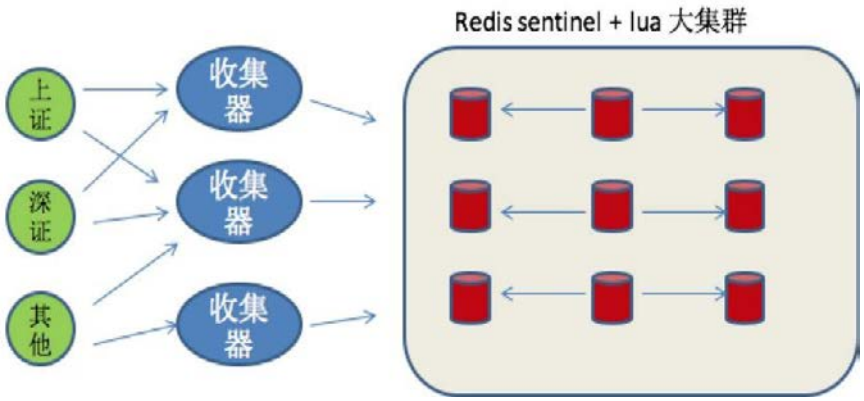
好在不同的证券可以分片并行、同样各周期段的 K 线指标也可以并行统计计算，实时处理这么大的计算量，我们显然需要高度并行处理。

我们最早的方案基于 Redis，因为 Redis 是一个非常好的承载时间序列数据的高性能内存存储技术。Redis 除了内置多种数据结构，还内置了

Lua 语言解释器，这意味着 Redis 除了具备数据存储能力也拥有了数据计算的能力。一个 Redis 存储集群中，每个节点加载 Lua 脚本，这基本上就是一个分布式并行计算集群了。

我们剩下要做的是事情是实现一个行情收集器，接收来自不同交易所的行情原始数据（node.js 善于处理 io 型事务，很适合用来实现收集器，细节不是本文焦点，不在此详述），发送 eval lua 指令到 Redis 集群。

Redis 收到指令后执行我们用 Lua 实现的指标计算逻辑完成指标计算，之后通过 Redis pub 将行情数据推送出去。如图 3 通过将计算挪到 Redis 存储节点，我们避免了复杂易出错的多进程管理问题，也大大简化了开发的工作量。



这个方案的一个优点，是技术架构比较简单，从数据存储到运算处理，都在 Redis 上，我们仅专注于 Redis 集群的性能优化、高可用方案实现、容灾备份、数据复制。对于运维来说，运维一套相对单一的技术系统，“零部件”（moving parts）越少越好，出现故障、单点失败的环节也少了很多。

“海量算子”——把数据快照挪到技术节点

上述 Redis+Lua 的方案，早期也服务了我们的市场与客户，但是当指

标计算量不断增大后，其不足也日益显著，主要体现在高度密集的计算导致影响存储集群的性能而产生延迟，并且在交易期间对存储集群作动态扩容是非常困难的。

有鉴于此，我们很自然的只能把指标计算任务从数据存储中回收，放弃一个数据与计算一体化的相对简单的架构，把计算的职责交给存储之外的专用运算节点。此时 Redis 变成单纯的内存存储。

这种实现将计算与存储分离，计算所需的数据不再能从”本地”获取到了，对于 Redis 而言，运算服务算是“out of process”（进程外），所以计算指标所需的数据，必须以一份数据快照的方式从 Redis 传递到运算节点，用以计算和更新，计算结果最终回写到 Redis，如图 4。

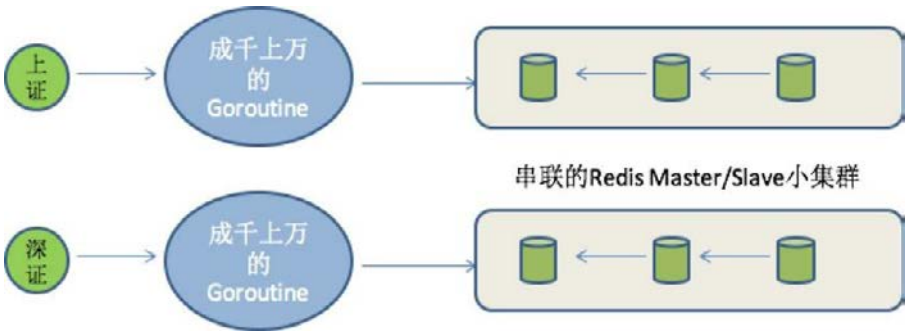
这个对于 Redis 而言“out of process”的计算能力，我们称之为运算节点（相对于 Redis 的数据节点），是一系列非常容易水平扩容的、高度并发的程序，我们采用了 Golang 来实现。Golang 这个语言，天生支持协程 (Goroutine) - 在一个运行的 Golang 程序中，可轻易启动上万的协程，所消耗的资源远小于线程，天生适合并行计算。

当接收到交易所 tick 数据时， 在一个运算节点中对每只证券解析及每类指标计算启动单独的 goroutine，每个 goroutine 在它的生命周期中只做一件事就结束（存活时间毫秒级），这很像数学上的函数执行过程完全没有副作用，goroutine 就是一个闭包算子，相互之间毫无影响。

Golang 的内存回收是并行的，数万个 goroutine 启动到销毁对性能的影响很小，另外 tick data 本身是有间隔的从交易所发过来的，在 goroutine 销毁那一刻往往是 tick data 空闲的间隔时间，这个空闲时机点用来做 goroutine 回收是合适的。

当然也有一些常驻 goroutine 用来将指标结果数据落地到 Redis。程

序语言本身是有各自不同的设计哲学的，Golang 正是这样一种语言，其协程机制让我们能够更细粒度的处理可分而治之的系统，同时免去了复杂易出错的多进程、多线程问题。



方案比较，孰优孰劣

1. Redis 既负责存储也负责计算

我们通过搭建 Redis 集群来进行分布式并行计算，Redis 集群本身有主备节点，这就涉及到主备同步的问题，虽然 Redis 自己解决了同步问题并且也支持增量同步，但通过 eval lua 指令在存储节点上计算时，同步的不是计算结果而是计算本身，也就是说同一个指标计算在主节点和备节点都需要各自计算一次。

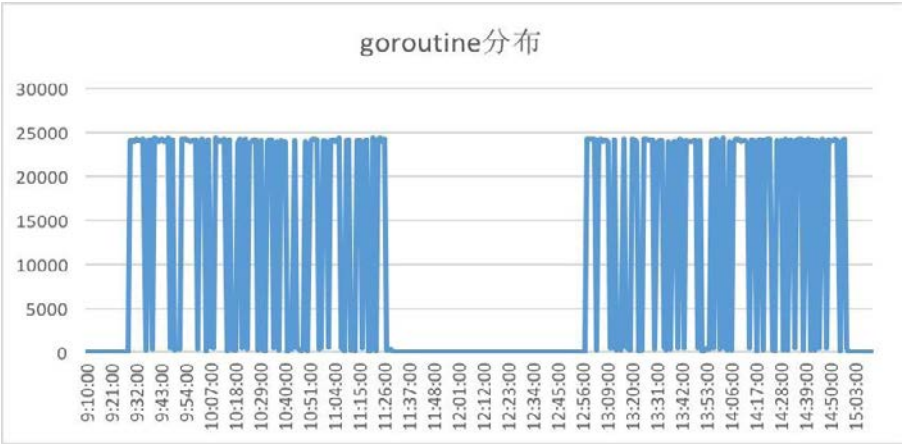
Redis 又是单线程处理，对于复杂的指标计算通过查看 SLOWLOG 会有上百毫秒的延迟，这会造成阻塞，对于要求快速的行情服务是不可接受的，为了减少阻塞，就需要尽可能多的进行分片，这意味着需要起更多的 redis 节点。

Redis 集群一旦搭建，节点缩扩容需要人工干预。对于证券行情服务这种目前 9:30-15:00 业务高峰，其他时段空闲的系统来说，高峰时无法弹性扩容，低峰时 Redis 节点进程无法回收造成资源浪费。

2. Redis 负责存储，海量 Goroutine 做算子

将指标计算从 Redis 拿出来交给 goroutine，Redis 仅作存储节点，主备节点同步的是指标计算结果，这样降低了 Redis 进程的 CPU 使用率，也不再有 SLOWLOG 记录，同时仅需要很少的分片，大大减少了 Redis 节点数。

Redis 集群规模的大小仅需要根据业务数据量确定。指标计算量的变化可以轻易通过启动更多的 goroutine 来进行方便的弹性扩容，goroutine 数也随投资者活跃程度变化，对交易频繁的股票只需要启动更多的 goroutine，不会像方案 1 那样造成部分 Redis 节点成为热点。在休市时段 goroutine 完全回收释放。



从 Redis+Lua 的数据存储与运算一体化，过度到数据存储与并行运算分离，也大大减少了系统对硬件资源的要求，Redis 集群规模减少十倍，如图表 1。采用 Redis+Lua 方案实现上比较简单，开发工作量较小，收集器与 Redis 数据交换量小，这种方案更适合简单的逻辑计算比如计数器；goroutine 的实现方案，虽然开发上复杂，与 Redis 数据交换量大了一些，

但更适合像行情指标计算这类复杂的应用场景。

方案	机器数（1个部署单元）	Redis集群	总CPU核数	总内存	计算最大延迟
In-process：数据“就地计算”（利用Lua）	5台高配	90个节点	128核	128G	几十毫秒
Out of process：数据快照挪到运算节点（基于goroutine）	4台中配	9个节点	32核	64G	几毫秒

总结

计算机科学发展到今日，基本的逻辑计算单元从进程到线程，再演进到比线程轻量的协程（co-routine），给开发人员更多的“工具”和更简单的方法去“榨取”硬件资源的利用率，同时又带来业务系统性能的提升。在面对诸多工具时，需要我们结合实际业务的场景特点对不同工具进行对比做出合适的设计选择。

作者介绍

陶瑞甫，中山大学信科院硕士毕业至今六年多一直从事软件研发工作，曾在华为参与底层进程管理、上层云管理系统研发，在腾讯参与多个互联网社交增值服务产品研发，2013 年初加入广发证券负责证券行情云服务建设与研发工作，目前参与证券交易系统相关研发工作。关注软件层面高性能并行计算技术，并致力于将这些理论技术应用于证券业的实际场景，给投资者带来更优质的服务。

版权声明

InfoQ 中文站出品

出版：极客邦控股（北京）有限公司
北京市朝阳区洛娃大厦 C 座 1607
欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。
网 址：www.infoq.com.cn