促进软件开发领域知识与创新的传播

架构师

ARCHITECT

推荐文章 | Article

京东Nginx平台化实践

携程基于Storm的实时大数据平台实践

股票市场事件处理引擎实践

观点 | Opinion

多形态MVC式Web架构的分类

热点 | Hot

如何使代码审查更高效

用10%的自主时间提升学习

Geekbang 极客邦科技 | InfoQ

2016年11月

卷首语

小米云平台 王刚

架构师是一个高大上的头衔，在百度百科中对架构师的定义是“一个既需要掌控整体又需要洞悉局部瓶颈并依据具体的业务场景给出解决方案的团队领导型人物。”ArchSummit 是一个知名的架构师技术交流论坛，高达 6800 元人民币的门票（当然人多的话有打折）也没能阻挡前来参会的架构师们的热情，也直接反映了架构师们一般都不差钱。我是一名工程师，做过搜索、也做过推荐、搭过数据平台、也挖过用户画像，从零开始建设过若干平台和系统，不敢妄称自己为架构师，但有一些经验体会可以分享。

系统架构的选择往往没有对与错，工程师们常常面对的是各种取与舍的矛盾。在众多的矛盾中，首当其冲的当属业务和技术的矛盾，业务快速发展，产品对需求的时间点要求感人，“帅哥，这个需求今天能完成吗？”，“这是老板的需求，赶快实现一下”。应对扑面而来的需求，工程师们往往疲于奔命，使用各种短平快的方法来满足业务的需求，长此以往，整个系统变得错综复杂，臃肿不堪，难以维护。一个好的架构需要在应对业务需求的同时，持续提炼通用化的模块，通用层基础且稳定。在通用模块的基础之上构建出适配层，由适配层处理复杂多变的业务需求，适配层灵活且多变。在技术上不断沉淀基础的通用模块，这不仅不会成为工程师们额

外的开发负担，反而可以提升业务需求的实现效率，提高系统的健壮性和可持续发展。解决业务和技术这一对矛盾，需要工程师站在业务和技术中间，不偏不倚。

优秀的系统架构面对的第二个矛盾是技术和运营的矛盾。 工程师们往往热爱新和奇的技术，重视新功能的开发，从 0 分做到 60 分的过程总是令人愉悦的，一切按照预期的演进是那么的美好。随着系统投入使用，数据规模越来越大、用户量和访问量逐步增大，算法效果需要不断提高，原有的架构暴露出来的问题也逐步增多。架构的演进进入了深水区，是抛弃原有架构进行重构，还是在现有系统中进行持续的演进，工程师们往往会遇到选择的难题。一个好的系统是要靠持续深入的运营来构建的，这里的运营是一个泛指的概念，指代构建系统中各种监控、调试信息、数据统计分析、数据对比等。一个系统不论是重构，还是持续演进，都需要开发全方位的运营系统来了解当前架构中的每一个环节。系统架构从 60 分演进到 100 分是一个艰苦的过程，没有一个系统是在一开发出来就解决了所有的问题的，提高运营能力是架构演进的基础，这本身不是一个技术问题，而是意识问题，需要工程师们做大量细致深入的调研和思考。

最后要说的是全局最优和局部最优的矛盾。我们在开发中常常遇到，当针对某个问题进行优化后，系统中又出了其他问题，也是俗话里说的头疼医头、脚疼医教、治标不治本。当你是从 0 开始搭建一个系统，对系统的各个环节和历史演进非常清楚，也就比较容易的从全局角度思考和解决问题。但对于复杂系统，大部分工程师往往只关注到系统的某一部分，解决问题的思路是受限于眼界，容易陷入至局部最优，而且往往局部的解决办法相对全局而言更复杂更间接。工程师需要有开阔的眼界，从更高层面理解系统架构，从而了解问题产生的本质，从全局角度出发予以解决。

本来是想多例举一些工作中遇到的例子来说明我在平台架构开发中遇到的矛盾，但限于时间太紧，只能下次有机会再和大家继续交流，也欢迎大家和我直接联系。

amazon web services

Geekbang 极客邦科技 | InfoQ

InfoQ在线课堂

如何更好地设置、管理和扩展你的Amazon ECS?

2016年11月22日（周二）
20:00-21:00

CONTENTS / 目录

架构师

2016 年 11 月刊

本期主编　杜小芳

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

如何使代码审查更高效

作者 Trisha Gee　译者 周元昊

众所周知，在团队中进行代码审查（Code Review）可以提升代码质量，分享项目知识、明确责任，最终达到构建更好的软件、更好的团队。如果你花几秒钟搜索代码审查的相关信息，你会看到许多关于代码审查带来的价值的文章。也有许多方法来进行代码审查：在 GitHub 中提 pull request，或使用像 JetBrains 的 Upsource 之类的工具。然而即使拥有清晰的流程和正确的工具，还遗留了一个大问题需要解决——我们需要找寻哪些问题。

可能没有明确关于“我们需要找寻哪些问题”的文章，是因为有许多不同的要点需要考虑。正如任何其他的需求，各个团队对各个方面都有不同的优先级。

本文的目标是列出一些审查者可以找寻的要点，而各个方面的优先级就因各个团队而异了。

在我们继续之前，让我们考虑一下大家在代码审查时会讨论到的问题。对于代码的格式、样式和命名以及缺少测试这些问题是很常见的几点。如果你想拥有可持续的、可维护的代码，这些是有用的检查点。然而，在代

码审查时讨论这些就有些浪费时间，因为很多这样的检查可以（也应该）被自动化。

那哪些要点是只能由人工进行审查而不能依靠工具的呢？

回答是有惊人数量的点只能由人工进行审查。在本文剩下的部分，我们会覆盖一系列广泛的特性，并深入其中的两点具体的区域：性能和安全。

设计

- 如何让新代码与全局的架构保持一致？
- 代码是否遵循SOLID原则，是否遵循团队使用的设计规范，如领域驱动开发等？
- 新代码使用了什么设计模式？这样使用是否合适？
- 基础代码是否有结合使用了一些标准或设计样式，新的代码是否遵循当前的规范？代码是否正确迁移，或参照了因不规范而淘汰的旧代码？
- 代码的位置是否正确？比如涉及订单的新代码是否在订单服务相关的位置？
- 新代码是否重用了现存的代码？新代码是否可以被现有代码重用？新代码是否有重复代码？如果是的话，是否应该重构成一个更可被重用的模式，还是当前还可以接受？
- 新代码是否被过度设计了？是否引入现在还不需要的重用设计？团队如何平衡可重用和YAGNI(You Ain’t Gonna Need It)这两种观点？

可读性和可维护性

- 字段、变量、参数、方法、类的命名是否真实反映它们所代表的事物。

- 我是否可以通过读代码理解它做了什么？
- 我是否理解测试用例测了什么？
- 测试是否很好地覆盖了用例的各种情况？它们是否覆盖了正常和异常用例？是否有忽略的情况？
- 错误信息是否可被理解？
- 不清晰的代码是否被文档、注释或容易理解的测试用例所覆盖？具体可以根据团队自身的喜好决定使用哪种方式。

功能

- 代码是否真的达到了预期的目标？如果有自动化测试来确保代码的正确性，测试的代码是否真的可以验证代码达到了协定的需求？
- 代码看上去是否包含不明显的bug，比如使用错误的变量进行检查，或误把and写成or？

你是否考虑过……

- 是否需要满足相关监管需求？
- 作者是否需要创建公共文档或修改现存的帮助文档？
- 是否检查了面向用户的信息的正确性？
- 是否有会在生产环境中导致应用停止运行的明显错误？代码是否会错误地指向测试数据库，是否存在应在真实服务中移除的硬编码的stub代码？
- 你对性能的需求是什么，你是否考虑了安全问题？这些是需要覆盖到的重大区域也是至关重要的话题，下面让我们仔细看下这两点。

性能

让我们深入探讨下性能，这是一个真正能从代码审查中获益的方面。

系统对性能方面的非功能性需求应当同所有架构、设计的领域一样被置于重要位置。无论你是开发只容许纳秒级延时的低延迟交易系统，还是管理“待办事项”的手机应用，你都应该了解用户所认为的“太慢”。

在考虑我们是否需要就代码性能进行代码审查之前，我们应该问自己几个关于具体需求是什么的问题。虽然一些应用确实不需要考虑每毫秒都花费在哪里，对于大部分应用，花费几个小时的折腾进行优化来获得的些许 CPU 下降的价值也是有限的，但有些地方还是审查者可以检查一下的，进而确保代码不会有一些常见可避免的性能缺陷。

这段代码是否有硬性的性能需求？

接受审查的代码是否涉及之前发布的服务等级协议（SLA）？或这个需求本身有特别的性能需求？

如果代码导致“登录页面加载太慢”，那原始的开发者需要找出合适的加载时间是多久，不然审查者或作者本人如何确保改进后的速度足够快？

如果有硬性的需求，是否有测试能证明满足了该需求？任何注重性能的系统应该就性能提供自动化测试，这能确保发布的 SLA 达到预期（如所有订单请求要在 10 毫秒内处理）。没有这些，你只能依靠你的用户来告诉你没有达到对应的 SLA。这不仅是一种糟糕的用户体验，还会带来原本可避免的罚金和支出。

这个修复或新增的功能是否会反向影响到任何现存的性能测试结果

如果你定期运行性能测试或有测试套件可以按需运行它们，那你就需要检查新的代码是否使得性能关键区域的系统性能有所下降。这可以是一个自动化的流程，但由于在持续集成环境中更常运行单元测试而不是性能

测试，所以值得特别指出可以在代码审查中检查这项。

调用外部的服务或应用的代价是昂贵的

任何通过网路来使用外部系统的方式通常会比没有很好优化的方法有更差的性能。考虑以下几点：

- 调用数据库：最坏的情况是问题隐藏在系统抽象中，如关系对象映射（ORM）中。但是在代码审查中你应该可以找到常见的导致性能问题的问题，如在循环中逐个调用数据库，一种情况就是加载ID列表之后，再在数据库中逐个查询ID对应的每条数据。
- 不必要的网络调用：就像数据库一样，远程服务有时也会被过度使用，原来只要一个远程调用就可实现的，或者可以使用批量或缓存防止昂贵网络调用的，却使用多个远程调用来实现。再次强调，像数据库一样，有时抽象类会隐藏调用远程API的方法。
- 移动或可穿戴应用过于频繁地调用后端程序：这基本上和“不必要的网络调用”相同，但是在移动设备上会产生其他问题，这不仅会产生不必要的调用后端使得性能变差，还会更快地消耗电量甚至导致用户的金钱支出。

有效且高效地使用资源

代码是否用锁来控制共享资源的访问？这是否会导致性能降低或死锁？

锁是一个性能开销大户，并在多线程环境中很难理清。考虑使用以下模式：单线程写或修改值，其余线程只读，或使用无锁算法。

- 是否存在内存泄露？Java中一些常见的原因会是：可变的静态字段，使用ThreadLocal变量和使用类加载器。

- 是否存在内存无限增长？这个和内存泄露不同，内存泄漏是指无用的对象不能被垃圾回收器回收。但对于任何语言，就算是没有垃圾回收的语言，也能创建无限变大的数据结构。作为审查者，如果你看见新的变量不断被加到list或map中，你就要问下，这个list或map什么时候失效或清除无用数据。
- 代码是否关闭了连接或数据流？关闭连接或文件、网络数据流很容易会被忘记。当你审查别人代码时，如果使用到文件、网络或数据库连接，就要确保它们被正确地关闭了。
- 资源池是否配置正确？针对一个环境的最佳配置取决于很多因素，所以作为审查者你很难马上知道像数据库连接池大小是否正确等这些问题。但是有一些是你一眼就可以看出来的，像资源池是否太小（比如大小设置为1）或太大（如数百万线程）。如果无法确定，就从默认值开始。没有使用默认值的就需要提供一定的测试或计算来证明这么配置的合理性。

审查者可以轻松找出的警告信号

一些代码一眼就能看出存在潜在性能问题。这依赖于所使用的语言和类库。

- 反射：Java的反射比正常调用要慢。如果你在审查含有反射的代码，你就要问下是否必须使用它。
- 超时：当你审查代码时，你可能不知道一个操作合适的超时时间，但是你要想一下“如果超时了，会对系统其他部分造成什么影响？”。作为审查者你应该考虑最坏的情况：当发生5分钟的延时，应用是否会阻塞？如果超时时间设置成1秒钟最坏的情况会是

怎么样的？如果代码作者不能确定超时长度，你作为审查者也不知道一个选定的时间的好坏，那么是时候找一个理解这其中影响的人参与代码审查了。

- 并行：代码是否使用多线程来运行一个简单的操作？这样是否花费了更多的时间以及复杂度而并没有提升性能？如果使用现代化的Java，那其中潜在的问题相较于显示创建线程中的问题更不容易被发现：代码是否使用Java 8新的并行流计算但并没有从并行中获益？比如，在少量元素上使用并行流计算，或者只是运行非常简单的操作，这可能比在顺序流上运算还要慢。

正确性

这些不一定影响系统的性能，但是它们与多线程环境运行关系密切，所以和这个主题有关：

- 代码是否使用了正确的适合多线程的数据结构。
- 代码是否存在竞态条件（race conditions）？多线程环境中代码非常容易造成不明显的竞态条件。作为审查者，可以查看不是原子操作的get和set。
- 代码是否正确使用锁？和竞态条件相关，作为审查者你应该检查被审代码是否允许多个线程修改变量导致程序崩溃。代码可能需要同步、锁、原子变量来对代码块进行控制。
- 代码的性能测试是否有价值？很容易将小型的性能测试代码写得很糟糕，或者使用不能代表生产环境数据的测试数据，这样只会得到错误的结果。
- 缓存：虽然缓存是一种能防止过多高消耗请求的方式，但其本身也

存在一些挑战。如果审查的代码使用了缓存，你应该关注一些常见的问题，如，不正确的缓存失效方式。

代码级优化

对大部分并不是要构建低延时应用的机构来说，代码级优化往往是过早优化，所以首先要知道代码级优化是否必要。

- 代码是否在不需要的地方使用同步或锁操作？如果代码始终运行在单线程中，锁往往是不必要的。
- 代码是否可以使用原子变量替代锁或同步操作？
- 代码是否使用了不必要的线程安全的数据结构？比如是否可以使用ArrayList替代Vector？
- 代码是否在通用的操作中使用了低性能的数据结构？如在经常需要查找某个特定元素的地方使用链表。
- 代码是否可以使用懒加载并从中获得性能提升？
- 条件判断语句或其他逻辑是否可以将最高效的求值语句放在前面来使其他语句短路？
- 代码是否存在许多字符串格式化？是否有方法可以使之更高效？
- 日志语句是否使用了字符串格式化？是否先使用条件判断语句校验了日志等级，或使用延迟求值？

简单的代码即高效的代码

Java 代码中有一些简单的东西可以供审查者寻找，这些会使 JVM 很好地替你优化你的代码：

- 短小的方法和类。
- 简单的逻辑，即消除嵌套的条件或循环语句。

越是人类可读的代码，JIT 编译器越有可能理解你的代码并进行优化。这应该在代码审查中很容易定位，如果代码看上去易理解且整洁，它运行时效率也会越好。

安全

你在构建一个安全、稳固的系统所花费的精力，和花在其他特性上的一样，取决于项目本身，项目运行的地方、它使用的用户、需要访问的数据等。我们现在着重看一些你可能在代码审查时关注的地方。

尽可能使用自动化

有惊人数量的安全检查可以被自动化，而不需要人工干预。安全测试不一定要启动所有系统进行完整的渗透测试，一些问题可以在代码级就能被发现。

常见问题如 SQL 注入或跨站脚本可以在持续集成环境通过相应工具检查出。你也能通过 OWASP 依赖检测工具自动化检查你依赖中已知的漏洞。

有时需要“看情况”

对有的校验你可以简单回答“是”或“否”，有时你需要一个工具指出潜在的问题，之后再由人工来决定这个是否需要解决。这也正是 Upsource 真正的闪光点。Upsource 显示代码检查结果，审查者可以利用这些来决定代码是否需要改动或还可以接受目前的情况。

理解你用到的依赖

第三方类库是侵蚀系统安全并引起漏洞的一个途径。当审查代码时至少你要检查是否引入了新的依赖（如第三方类库）。如果你还没有自动化检查漏洞，你应该检查新引入的类库中已知的问题。

你也应该尝试着最小化每个类库的版本，当然如果其他依赖有一个额

外的间接依赖，这点可能达不到。但最简单的最小化自己代码暴露在他人代码的（通过类库或服务）安全问题中的方法有：

- 尽可能使用源码并理解它的可信度。
- 使用你所能得到的质量最高的类库。
- 追踪你在何处使用了什么，当新的漏洞出现，你可以查看你受影响的程度。

检查是否新的路径和服务需要认证

无论你开发 web 应用、提供 web 服务或一些其他需要认证的 API，当你增加一个新的 URI 或服务时，你应该确保它不能在没有认证的情况下被访问（假设认证是你系统的需求）。你只要简单地检查代码的开发者写了合适的测试用例来展示进行了认证。

你应该不只针对使用用户名和密码的人类用户来考虑认证。其他系统或自动化流程来访问你的应用或服务也会需要认证。这可能影响你们系统中对“用户”的定义。

数据是否需要加密

当你保存一些数据到磁盘或通过线缆传输，你需要了解数据是否应该被加密。显然密码永远不应该是简单文本，但是有诸多其他情况数据需要加密。如果被审查的代码通过线缆来传送数据或保存在某地或以其他方式离开你的系统，且你不知道它是否应该被加密，尝试询问下你组织中可以回答这个问题的人。

密码是否被很好地控制？

这里的密码包含密码（如用户密码、数据库密码或其他系统的密码）、秘钥、令牌等等。这些永远不应该存放在会提交到源码控制系统的代码

或配置文件中，有其他方式管理这些密码，例如通过密码服务器（secret server）。当审查代码时，要确保这些密码不会悄悄进入你的版本控制系统中。

代码的运行是否应该被日志记录或监控？是否正确地使用？

日志和监控需求因各个项目而不同，一些需要合规，一些拥有比别人严格的行为、事件日志规范。如果你有规章规定哪些需要记录日志，何时、如何记录，那么作为代码审查者你应该检查提交的代码是否满足要求。如果你没有固定的规章，那么就考虑：

- 代码是否改变了数据（如增删改操作）？是否应该记录由谁何时改变了什么？
- 代码是否涉及关键性能的部分？是否应该在性能监控系统中记录开始时间和结束时间？
- 每条日志的日志等级是否恰当？一个好的经验法则是“ERROR”触发一个提示发送到某处，如果你不想这些消息在凌晨3点叫醒谁，那么就将之降级为“INFO”或“DEBUG”。当在循环中或一条数据可能产生多条输出的情况下，一般不需要将它们记录到生产日志文件中，它们更应该被放在“DEBUG”级别。

记得叫上专家

安全是个很大的话题，大到足以让你的公司聘请技术安全专家。我们有安全专家就可以获得他们的帮助，如，邀请他们参加代码审查，或邀请他们在审查代码时和我们结对。如果这个无法实现，我们可以充分学习我们系统的环境，来理解我们有哪种安全需求（面向内部的企业级应用和面向客户的网页应用有不同的标准），所以我们可以更好地理解我们应该在

代码审查中看什么。

总结

代码审查是一个很好的方式，不仅确保了代码质量和一致性，也在团队中或团队间分享了项目知识。即使你已经自动化了基础的校验，还有许多不同代码、设计的方面需要考虑。代码审查工具，如 Upsource，通过在每个代码提交的检查中高亮可疑的代码并分析哪些问题已经被修复，新引入哪些问题，可以帮你定位一些潜在的问题。工具也可以简化流程，因为它提供了一个平台来讨论设计和代码实现，也可以邀请审查者、作者和其他相关人员参加讨论直到达成共识。

最后，团队需要花时间决定代码质量的哪些因素对他们是重要的，也需要专家人工决定哪些规则应用到各个代码审查中，参与到审查中的每个人都应该具备并使用人际交往的技巧，如积极的反馈、谈判妥协以达到最终的共识，即代码应该怎么样才“足够好”可以通过审查。

用 10% 的自主时间提升学习

作者 Ben Linders　译者 谢丽

Giuseppe de Simone 是爱立信公司的首席敏捷教练 & 培训师。他认为，给团队 10% 的自主时间用来学习可以缩短交付时间，提高质量，提升积极性。在敏捷希腊峰会 2016 上，他做了题为“网络化社会时代的管理”，介绍了他们如何试用“10% 原则”提高团队的自主权。

团队可以把 10% 的时间花费在他们自己选择的主题上。这一原则为团队提供了完全的自主权，让他们可以从事他们认为重要的工作。按照 De Simone 的说法，这样做可以解放人们的创造力，增强团队的潜能。

在报告“‘10% 时间’的优缺点”中，Elizabeth Pope 介绍了 Holiday Extras 如何运用“10% 时间”，InfoQ 对对此进行过概要介绍：

“10% 时间”基本上是在工作时间里分配的不处理“正常业务（BAU）”的时间，并可用于研究和发展，以及学习和提升专业技能。在一个典型的英国工作周里，这意味着每周 4 小时（半天）可用于这些活动。

据介绍，将 10% 的工作时间用于 R&D 和学习对于组织和个人而言都是有好处的。以下是部分成果：

- 个人发展
- 可以促进公司创新的试验工作

- 提高效率
- 增强合作
- 提升参与度及发展“自组织”技能
- Bug修复及消除技术债务

在敏捷希腊峰会上，当 Giuseppe De Simone 演讲结束后，InfoQ 对他进行了采访，内容涉及员工在这 10% 的自主时间里做什么、10% 的自主时间带来了什么好处以及组织支持开展了哪些活动来促进学习。

InfoQ：您能举例说明下员工在这 10% 的自主时间里做什么吗？

Giuseppe De Simone：获得这个机会的每个团队都有自己的选择。我合作过的最成功的团队将时间用在类似改进其 CI 系统、加快其价值流或者只是读书这样的事情上，或者用于社区实践、编码道场以及围绕他们选择的主题将他们的想法原型化。其他团队则只是将时间用于实现来自回顾会议的整改措施。

InfoQ：10% 的自主时间带来了什么好处？

De Simone：在我指导过的其中一个组织里，我们设法将特性开发（从概念到准备好部署）的平均交付期缩减了 25%，并在 18 个月的时间里将客户网站的缺陷减少了 42%。同时，我们将积极性指数提高了 10%。

我们取得这两项成果主要是得益于 Scrum，以及一种以我在演讲中介绍的行为、技能和工具为基础的新的领导力风格，而这是受 21 世纪重塑管理的思潮所启发。

如果不解放人们的创造力，让团队充分发挥其潜能，其中许多事情就无法完成，而结果证明，它们是取得那些成果的基础。在这个游戏中，10% 完全自主的时间是主要因素，而这种时间投入的业务回报很难量化；无论我怎么估计都可能会低估其价值。

InfoQ：组织开展了哪些活动来促进学习？这带来了哪些好处？

De Simone：在我参与过的众多活动中，构建学习平台是其中的一个关键要素。如果一家公司希望在日常工作中实现 21 世纪的管理原则，并采用可以应对当前业务挑战的那种领导力，那么它也是其中的一个支柱。

我在希腊峰会上的演讲源于我支持几个组织实现敏捷转型的经验。在这个过程中，我分享了多个具体的活动实例，不同组织的领导者已经实行或支持了这些活动，从而在多个方面促进了学习。

- 编程马拉松：这是一项持续24小时不间断的活动。在这个活动中，一名参与者提出一项挑战，并召集来自世界不同地区的人组建一个团队，而这些人都非常愿意将那个想法转换成一个原型用来演示。
- 读书会：一小部分人共同决定要阅读的书，约定阅读量，每周一次共进午餐，以便分享每个人的思考，从他人的阅读中学到东西。
- 影视俱乐部：每周一次，一小部分人在午餐时间里聚在一起，用1个小时的时间观看一段30分钟的视频，并通过富有成效的对话分享他们的观点。
- 聚会：举办为期一天的会议，重点讨论某个特定的角色或同行所面临的挑战。例如，去年4月，我们在斯德哥尔摩组织了一次开放式的Scrum管理员聚会：有来自11家公司的大约130个人参与了一整天的Open Space。我们分享和讨论了将近30个不同的主题。开始和结束时都有一个主题演讲，丰富了这次活动。
- 社区实践：聚集了开发人员、管理人员和教练。

传统的组织会制定并部署一个详细的流程供半熟练的员工遵循，而真正的精益和敏捷组织会让他们的人员和专家成长为各自岗位上最好的人才，并让他们决定最适合他们工作的流程。

你认为哪一种可以更快地响应变化，获得更多取悦客户的机会呢？

京东 Nginx 平台化实践

作者 吴建苗

Nginx 是优秀的 HTTP 和反向代理服务器，京东各部门都在广泛使用，但普遍都面临着一些问题：

- 配置复杂，专业性强。
- 配置文件无法批量修改且配置变更依赖重启操作。
- 不同应用依赖不同模块、配置项，管理混乱。
- 同一应用的Nginx无法批量、快速扩容。

所有问题的根源在于 Nginx 是一个单机系统，虽然模块化、高性能，但在互联网高速发展的今天，像京东这样拥有大规模 Nginx、业务集群的场景下，所有问题都有可能被无限放大，针对这种现状我们设计研发了 JEN（JD EXTENDED NGINX），截止目前 JEN 已覆盖京东金融大部分核心业务，如夺宝吧，卡超市，白条等。

一、整体结构

如图 1，运维通过 Web 控制台做相应的配置操作，若是分流、限流等配置，则信息入库等待 Nginx 通过 Restful API 同步规则后开始生效；若是平滑升级、重启等强运维性操作，则 Web 控制台通过控制 Ansible 对

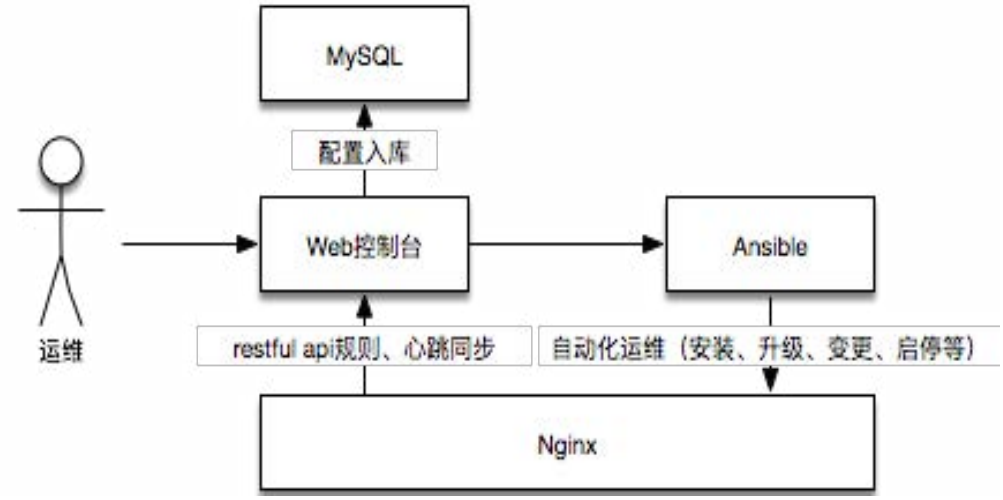

图 1：JEN 结构图

Nginx 进行相应操作（见图 2）。

JEN 特点：

- 支持Nginx自动发现，分组管理，状态监控。
- 统一入口，通过抽象配置，简化操作管控Nginx集群生命周期，并支持规则批量配置，操作批量执行。
- 扩展了原生Nginx的分流、限流功能，支持规则的内存实时同步，无需修改配置文件，更无需重启Nginx进程。

1. 基础信息

Web 上所有的展示和操作全部基于对基础信息的计算整合，主要包含两类：

- 分组信息（业务线、应用、机房、Nginx IP）
- Nginx属性，例如upstream信息，server_name，listen_port等，主要来源Nginx读取Nginx.conf内容后的信息上报（心跳）

对于分组信息，JEN 支持以下两种方式填充：

- 调用外部服务的Restful API导入完整的基础信息。
- 对自动发现的Nginx做分组的手工编辑。

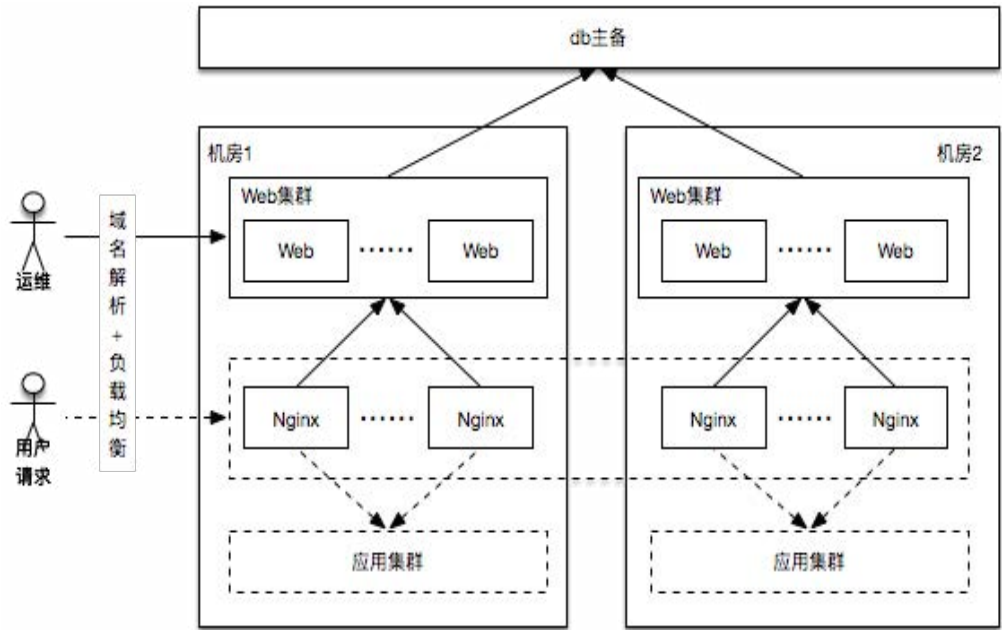

图 2：Nginx 和 Web 控制台多机房部署图

如图 3，分组包括业务线、应用、机房、Nginx 共四层关系，在大规模集群环境下可以通过这种关系并结合 Nginx 属性，支持对所有操作的批量执行，如批量修改配置文件，批量升级重启等，解放生产力。

2. 规则获取

用户在 Web 控制台配置后，在 Nginx 端我们实现了全异步的模块支持定时向 Web 获取属于当前 Nginx 的规则信息，规则存储内存，即时生效，其中：

a）规则信息每个进程存储一份，避免进程间资源共享导致锁竞争。

b）版本号设计，保证规则和心跳的绝对顺序，不因丢包、延迟等网络因素导致版本错乱，而且在规则未变更时 Nginx 无需频繁解析大量规则信息而消耗 CPU 资源。

3. 安全

JEN 支持三类角色，每种角色支持不同的操作权限（默认是普通用户

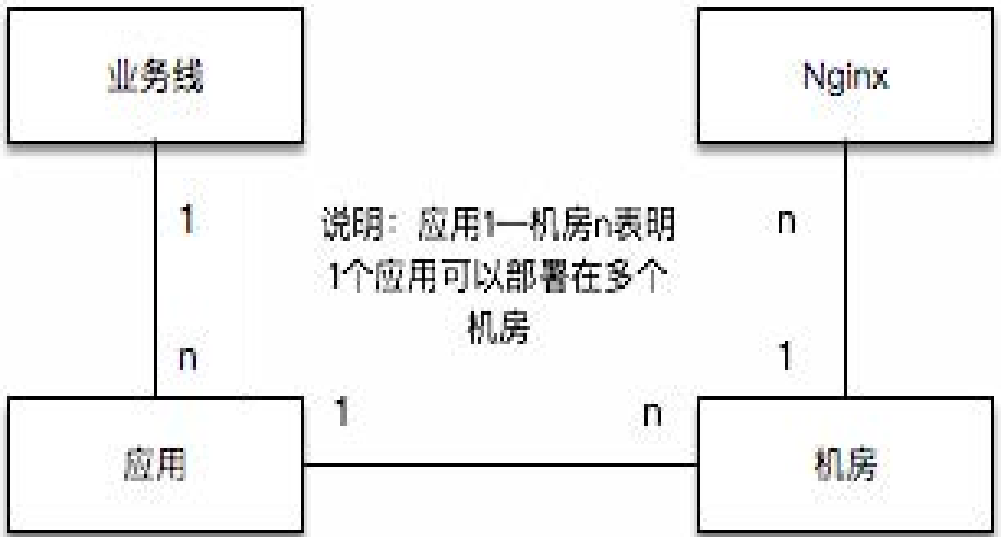

图 3：各分组间关系图

角色，无写权限），任何角色对 Web 的任何操作都会被记录，并在 Web 提供了入口支持多维度操作日志查询，便于审计。

4. 监控

我们实现了更为全面的监控信息采集与展示，包括：

a）扩展了 tengine 的主动探测模块，支持上游服务器的平均、当前延时统计。

b）通过与 Web 的心跳保持支持 Nginx 存活状态监控。

c）支持 TCP 连接信息，in/out 流量，QPS，1xx 到 5xx 回应报文等信息监控。

以上的监控信息支持分组统计（业务线、应用、机房）和大屏展示，便于相关人员（业务，运维）实时监控应用状态。

二、分流

概念：根据请求特征（IP，header 中任意关键字）支持把某些特定请求分流到单个或多个上游服务器中，如图 4。

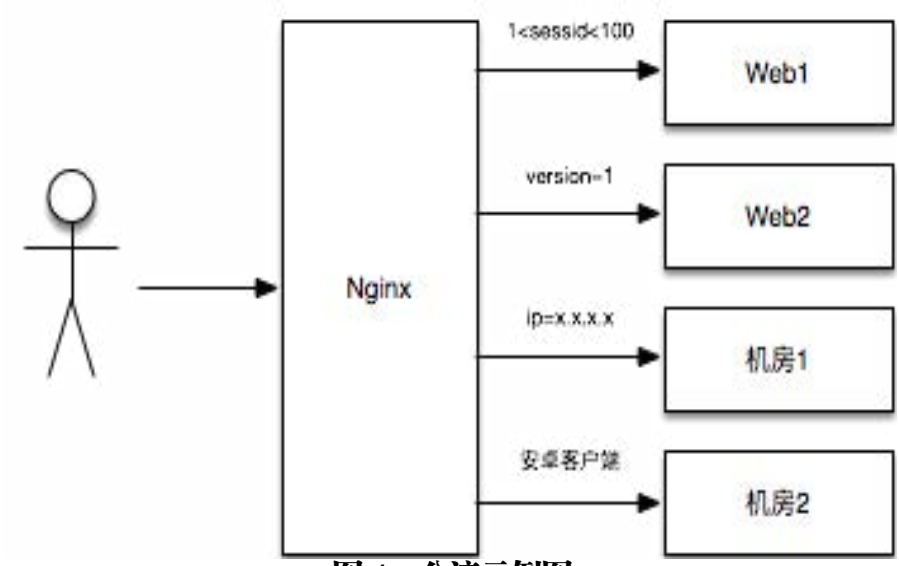

图 4：分流示例图

分流主要适用灰度发布，ab testing 等场景，另外我们也对分流功能做了扩展，支持 Web 控制台一键启停上游服务器，便于当应用服务器需要维护或升级时，用户请求正常访问。

三、限流

京东 618 等大促，货物都提前堆积在购物车，等待零点秒杀，换成工程师的语言来说，就是前一秒的 QPS 很低，但是下一秒 QPS 非常高，流量大意味着机器负载高，若一个应用的一两台机器没有扛住，这样就会导致整个应用集群雪崩。

限流不可盲目，首先需要根据业务特点选择合适的限流算法（漏桶算法、令牌桶算法），其次需要结合历史流量、应用服务能力、营销力度等因素综合评定限流参数，最后决定以何种优雅的方式反馈用户。

Nginx 在实现上通过共享内存共享限流中间信息的方式来达到多进程间的状态统一。在 JEN 设计初衷，原本计划和分流一致，即每个进程存储一份限流规则，限流只在当前进程内限流，但不可避免的会出现如下问题：

- 每个进程“你限你的，我限我的”，信息不一致进而导致限流不准确。
- 类似用户ID的限流，在京东这样拥有庞大日活用户的场景下，每个进程需要开辟足够大的内存才能避免限流算法中对于红黑树节点的频繁置换，这样一来Nginx占用内存就会随着进程数成倍扩大。

我们的做法：

预分配共享内存，Nginx 获取到限流规则时动态适配一块共享内存。

规则共享，生效后实时同步至所有进程，规则链保证所有旧版本规则只有在当前流量更新之后才会删除，如图 5 所示。

我们在限流功能上的几点扩展，支持错误页定制，除了返回 Nginx 静态页，还支持 302 错误页重定向，根据在 Web 控制台的配置可以重定向到任何外部链接，但 302 重定向存在一个问题：用户浏览器的 URL 和内容都发生了变更，意味着用户需要重新输入 URL 重新请求或者是重复之前的操作步骤，用户体验差可能导致用户放弃此次购买行为而转投它家。在逻辑上我们通过 Nginx 的 subrequest 机制支持返回内容发生变更而 URL 保持

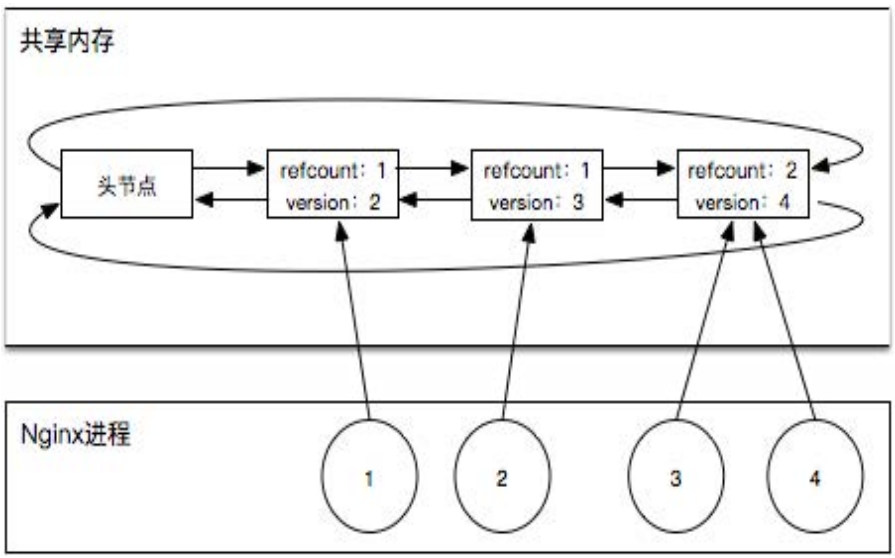

图 5：规则链

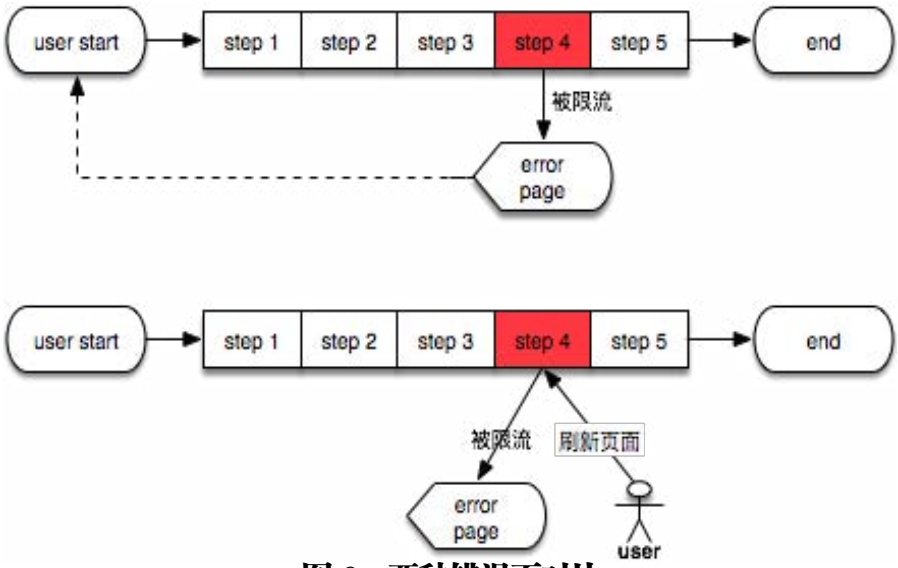

图 6：两种错误页对比

不变，这样一来每当用户被限流，只需重新刷新页面即可重复之前的操作步骤，如图 6 所示。

通过扩展限流算法支持限流后一段时间不可用，例如按 IP 限流且某个 IP 已经触发限流，则支持该 IP 一段时间内不可访问，无需重新通过算法计算。

同步实现了黑名单、白名单功能，通过白名单避免一些复杂场景下的限流“误杀”（例如 nat 网络下按 ip 限流）。

四、运维特性

运维特性主要指 Nginx 的安装、升级、配置文件修改、启停等操作，运维特性与之前介绍内容的最大区别在于需要重启操作，所以结合第三方工具 Ansible 是比较合适的想法（Ansible 相对于 Puppet 等运维工具，其迁移成本相对较小）。

在实际生产中 Ansible 和 Web 为避免单点需要集群部署，我们的方案是：Web 和 Ansible 在同一 PC 上部署，相关数据改用 DB 存储替代

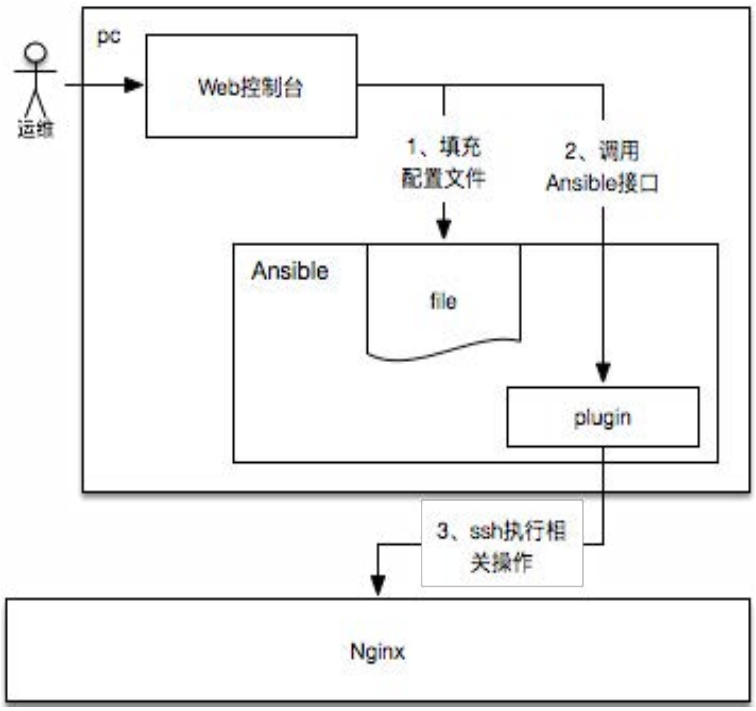

图 7：自动化运维操作逻辑图

Ansible 本地文件存储，通过这种简单的改造可以方便 Ansible 和 Web 这组“套件”进行扩容。

如图 7，用户通过 Web 操作控制 Ansible 对 Nginx 进行升级、重启等操作，Web 是 Nginx 操作的统一入口，这是平台化的重要意义所在，可以放弃 SSH，Shell 甚至是监控系统，开始在 JEN 自给自足了。

通过主动拉取或者是用户在页面导入、手工配置，JEN 会为所有 Nginx 存储配置文件，这样不仅原本因为每个应用都依赖不同的配置项而导致管理混乱的局面得到了改善，而且也可以方便的对配置文件做些扩展，例如历史记录追溯，配置比对，配置复用，操作回滚等。

在页面执行相关操作时，Web 会读取 Ansible 的标准输出并在页面实时展示，为了让使用者以相对友好的方式获知进度我们对 Ansible 做了优化：

- 丰富了标准输出的内容，尽量细化到每一个步骤。
- 格式化标准输出，便于Web获取和展示。

Nginx 在生产环境大规模部署，倘若因为一些原因导致 Nginx 大规模异常，这是我们不希望看到的，所以在可靠性方面，JEN 也提供了多种机制来保证：

1. 三层错误校验，保证只有在完全正确的情况下才会重启和更新进程，中途发生任何错误不影响线上服务

a）在 Web 填充表单时做第一层校验。

b）在目标机器做操作时做第二层检测，例如先执行 Nginx -t 校验。

c）执行完毕做第三层校验，例如端口是否启动，进程数是否一致等。

2. 灰度执行

a）单个 Nginx 依次执行，有任何异常立即中断开始人工介入。

b）按百分比支持批量执行，例如某个机房的 Nginx 先升级 10%。

五、总结

以上整理了京东在 Nginx 平台化方面的一些实践，JEN 提供了统一入口管控整个 Nginx 生命周期，并支持规则的批量修改即时生效，我们希望这些实践经验能对所有读者产生帮助。

作者介绍

吴建苗，目前就职于京东金融杭州研发中心，负责 Nginx，MQ 项目，对高性能服务器开发和调优具有浓厚兴趣。欢迎沟通交流（微信号 wujm1230）。

携程基于 Storm 的实时大数据平台实践

作者 张翼

本文讲解了携程在实时数据平台的一些实践，按照时间顺序来说明我们是怎么一步一步构建起这个实时数据平台的，目前有一些什么新的尝试，未来的方向是怎么样的，希望对需要构建实时数据平台的公司和同学有所借鉴。

为什么要做实时数据平台

首先先介绍一下背景，为什么我们要做这个数据平台？其实了解携程的业务的话，就会知道携程的业务部门是非常多的，除了酒店和机票两大业务之外，有近20个SBU和公共部门，他们的业务形态差异较大，变化也快，原来那种Batch形式的数据处理方式已经很难满足各个业务数据获取和分析的需要，他们需要更为实时地分析和处理数据。

其实在这个统一的实时平台之前，各个部门自己也做一些实时数据分析的应用，但是其中存在很多的问题。

首先是技术选型五花八门，消息队列有用ActiveMQ的，有用RabbitMQ的，也有用Kafka的，分析平台有用Storm的，有用Spark-streaming的，也有自己写程序处理的；由于业务部门技术力量参差不齐，

并且他们的主要精力还是放在业务需求的实现上，所以这些实时数据应用的稳定性往往难以保证。

其次就是缺少周边设施，比如说像报警、监控这些东西。

最后就是数据和信息的共享不顺畅，如果度假要使用酒店的实时数据，两者分析处理的系统不同就会很难弄。所以在这样前提下，就需要打造一个统一的实时数据平台。

需要怎样的实时数据平台

这个统一的数据平台需要满足 4 个需求：

- 首先是稳定性，稳定性是任何平台和系统的生命线；
- 其次是完整的配套设施，包括测试环境，上线、监控和报警；
- 再次是方便信息共享，信息共享有两个层面的含义，1、是数据的共享；2、是应用场景也可以共享，比如说一个部门会受到另一个部门的一个实时分析场景的启发，在自己的业务领域内也可以做一些类似的应用；
- 最后服务响应的及时性，用户在开发、测试、上线及维护整个过程都会遇到各种各样的问题，都需要得到及时的帮助和支持。

如何实现

在明确了这些需求之后我们就开始构建这个平台，当然第一步面临的肯定是一个技术选型的问题。消息队列这边 Kafka 已经成为了一个既定的事实标准；但是在实时处理平台的选择上还是有蛮多候选的系统，如 Linkedin 的 Samza， apache 的 S4，最主流的当然是 Storm 和 Spark-streaming 啦。

出于稳定和成熟度的考量，当时我们最后是选择了 Storm 作为实时平台。如果现在让我重新再来看的话，我觉得 Spark-streaming 和 Storm 都

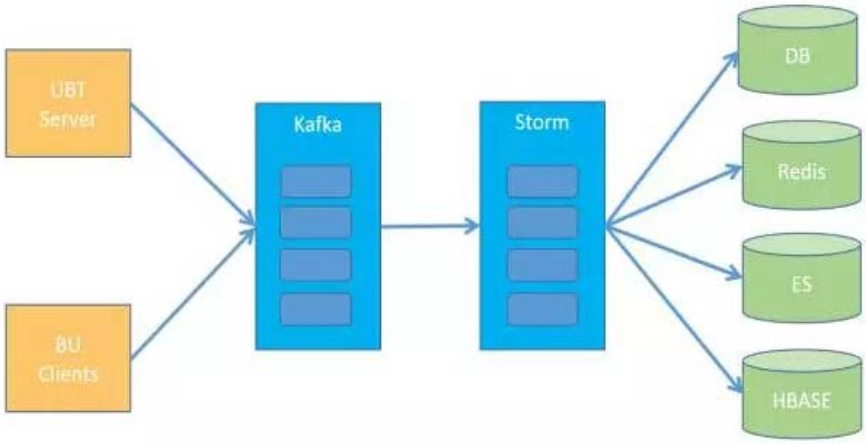

图 1

是可以的，因为这两个平台现在都已经比较成熟了。

架构图的话就比较简单，就是从一些业务的服务器上去收集这个日志，或者是一些业务数据，然后实时地写入 Kafka 里面，Storm 作业从 Kafka 读取数据，进行计算，把计算结果吐到各个业务线依赖的外部存储中（见图 1）。

那我们仅仅构建这些就够了吗？当然是远远不够的，因为这样仅仅是一些运维的东西，你只是把一个系统的各个模块搭建起来。

前面提到的平台的两个最关键的需求：数据共享和平台整体的稳定性很难得到保证，我们需要做系统治理来满足这两个平台的关键需求。

首先说说数据共享的问题，我们通常认为就是数据共享的前提是指用户要清晰的知道使用数据源的那个业务含义和其中数据的 Schema，用户在一个集中的地方能够非常简单地看到这些信息；我们解决的方式是使用 Avro 的方式定义数据的 Schema，并将这些信息放在一个统一的 Portal 站点上；数据的生产者创建 Topic，然后上传 Avro 格式的 Schema，系统会根据 Avro 的 Schema 生成 Java 类，并生成相应的 JAR，把 JAR 加入 Maven 仓库；对于数据的使用者来说，他只需要在项目中直接加入依赖即可。

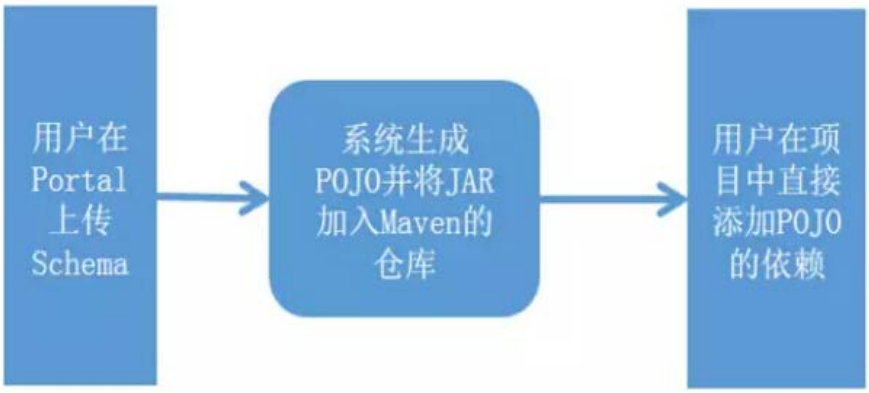

图 2

此外，我们封装了 Storm 的 API，帮用户实现了反序列化的过程，示例代码如下，用户只要继承一个类，然后制定消息对应的类，系统能够自动完成消息的反序列化，你在 process 方法中拿到的就是已经反序列化好的对象，对用户非常方便。

```
private static class ExtractBolt extends AbstractMuiseHermesBoltAutoAcked<UserAction> {
        public ExtractBolt(String topic){
            super(topic, UserAction.class);
        }
        @Override
        public void process(UserAction message, BasicOutputCollector collector) {
            //业务逻辑
            ......
            collector.emit(new Values(vid, target, page, type));
        }
        @Override
        public void specifyOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("vid", "target", "page", "type"));
        }
    }
```

其次我们来说说资源控制，这个是保证平台稳定性的基础，我们知道 Storm 其实在资源隔离方面做得并不是太好，所以我们需要对用户的 Storm 作业的并发做一些控制。我们的做法还是封装 Storm 的接口，将原来设定 topology 和 executor 并发的方法去掉，而把这些设置挪到

Portal 中。下面是示例的代码：

```
public static void main(String[] args) throws AlreadyAliveException, InvalidTopologyException {
        String topologyName = "UBT-Demo-New";
        String topicName = "UBT_TOPIC_Action";
        CtripKafkaSpout hermesSpout = new CtripKafkaSpout(topicName);
        CtripTopologyBuilder builder = new CtripTopologyBuilder(topologyName);
        builder.setSpout("hermes-spout", hermesSpout);
        builder.setBolt("extractbolt", new ExtractBolt(topicName)).localOrShuffleGrouping("hermes-spout");
        builder.setBolt("analysisbolt", new AnalyseBolt(5)).fieldsGrouping("extractbolt", new Fields("vid", "page"));
        builder.setBolt("dashboardbolt", new DashBoardBolt()).localOrShuffleGrouping("analysisbolt");
        Config conf = new Config();
        if(args != null && args.length > 0 && "local".equalsIgnoreCase(args[0])){
            CtripStormSubmitter.submitToLocal(conf, builder);
        }else{
            CtripStormSubmitter.submitToCluster(conf, builder);
        }
    }
}
```

另外，我们前面已经提到过了，我们做了一个统一的 Portal 方便用户管理，用户可以查看 Topic 相关信息，也可以用来管理自己的 Storm 作业，配置，启动，Rebalance，监控等一系列功能都能够在上面完成。

在完成了这些功能之后，我们就开始初期业务的接入了，初期业务我们只接了两个数据源，这两个数据源的流量都比较大，就是一个是 UBT（携程的用户行为数据），另一个是 Pprobe 的数据（应用流量日志），那基本上是携程用行为的访问日志。主要应用集中在实时的数据分析和数据报表上。

在平台搭建的初期阶段，我们有一些经验和大家分享一下：

1. 最重要的设计和规划都需要提前做好，因为如果越晚调整的话其实付出的成本会越大的；
2. 集中力量实现了核心功能；
3. 尽早的接入业务，在核心功能完成并且稳定下来的前提下，越早接入业务越好，一个系统只有真正被使用起来，才能不断进化；
4. 接入的业务一定要有一定的量，因为我们最开始接入就是整个携程的整个UBT，就是用户行为的这个数据，这样才能比较快的帮助整

个平台稳定下来。因为你平台刚刚建设起来肯定是有各种各样的问题的，就是通过大流量的验证之后，一个是帮平台稳定下来，修复各种各样的bug，第二个是说会帮我们积累技术上和运维上的经验。

在这个之后我们就做了一系列工作来完善这个平台的“外围设施”：

首先就是把 Storm 的日志导入到 ES 里面，通过 Kanban 展示出来；原生的 Storm 日志查看起来不方便，也没有搜索的功能，数据导入 ES 后可以通过图标的形式展现出来，也有全文搜索的功能，排错时非常方便。

其次就是 metrics 相关的一些完善；除了 Storm 本身 Build in 的 metrics 之外我们还增加了一些通用的埋点，如从消息到达 Kafka 到它开始被消费所花的时间等；另外我们还是实现了自定义的 MetricsConsumer，它会把所有的 metrics 信息实时地写到携程自己研发的看板系统 Dashboard 和 Graphite 中，在 Graphite 中的信息会被用作告警。

第三就是我们建立了完善的告警系统，告警基于输出到 Graphite 的 metrics 数据，用户可以配置自己的告警规则并设置告警的优先级，对于高优先级的告警，系统会使用 TTS 的功能自动拨打联系人的电话，低优先级的告警则是发送邮件；默认情况下，我们会帮用户添加 Failed 数量和消费堵塞的默认的告警（见图 3）。

第四，我们提供了适配携程 Message Queue 的通用的 Spout 和写入 Redis，HBbase，DB 的通用的 Bolt，简化用户的开发工作。

最后我们在依赖管理上也想了一些方法，方便 API 的升级；在 muise-core（我们封装的 Storm API 项目）的 2.0 版本，我们重新整理了相关的 API 接口，之后的版本尽量保证接口向下兼容，然后推动所有业务

图 3

都升级一遍，之后我们把 muise-core 的 jar 包作为标准的 Jar 包之一放到每台 supervisor 的 Storm 安装目录的 lib 文件夹下，在之后的升级中，如果是强制升级，就联系用户，逐个重启 Topology，如果这次升级不需要强制推广，等到用户下次重启 Topology 时，这个升级就会生效。

在做完这些工作之后，我们就开始大规模的业务接入了，其实目前基本上覆盖了携程的所有的技术团队，应用的类型也比初期要丰富很多。

下面给大家简单介绍一下，在携程的一些实时应用。主要分为下面四类：

1. 实时数据报表；
2. 实时的业务监控；
3. 基于用户实时行为的营销；
4. 风控和安全的应用。

第一个展示的是携程这边的网站数据监控平台 cDataPortal，携程会对每个网页访问的性能做一些很详细的监控，然后会通过各种图表展示出来（见图 4）。

第二个应用是携程在 AB Testing 的应用，其实大家知道 AB Testing 只有在经过比较长的一段时间，才能得到结果，需要达到一定的量之后才

图 4- 图 5

会在统计上有显著性；那它哪里需要实时计算呢？实时计算主要在这边起到一个监控和告警的作用：当 AB Testing 上线之后，用户需要一系列的实时指标来观察分流的效果，来确定它配置是否正确；另外需要查看对于订单的影响，如果对订单产生了较大的影响，需要能够及时发现和停止（见图 5）。

第三个应用是和个性化推荐相关，推荐其实更多的是结合用户的历史偏好和实时偏好来给大家推荐一些场景。这边实时偏好的收集其实就是通过这个实时平台来做的。比较相似的应用有根据用户实时的访问行为推送一些比较感兴趣的攻略，团队游会根据用户的实时访问，然后给用户推送一些优惠券之类的（见图 6）。

那些曾经踩过的坑

在说完了实时数据平台在携程的应用，让我们简单来聊聊这个过程中我们的一些经验。

首先是技术上的，先讲一下我们遇到的坑吧。

我们使用的 Storm 版本是 0.9.4，我们遇到了两个 Storm 本身的 BUG，当然这两个 bug 是比较偶发性的，大家可以看一下，如果遇到相应的问题的话，可以参考一下：

- Storm-763：Nimbus已经将worker分配到其他的节点，但是其他worker的netty客户端不连接新的worker。

  应急处理：Kill 掉这个 worker 的进程或是重启相关的作业。

- Storm-643：当failed list不为空时，并且一些offset已经超出了Range范围，KafkaUtils会不断重复地去取相关的message。

另外就是在用户使用过程中的一些问题，比如说如果可能，我们一般会推荐用户使用 localOrShuffleGrouping，在使用它时，上下游的 Bolt 数要匹配，否则会出现下游的大多数 Bolt 没有收到数据的情况，另外就是用户要保证 Bolt 中的成员变量都要是可序列化的，否则在集群上运行时就会报错。

然后就是关于支持和团队的经验，首先在大量接入前其告警和监控设施是必须的，这两个系统是大量接入的前提，否则难以在遇到非常问题时及时发现或是快速定位解决。

第二就是说清晰的说明、指南和 Q&A 能够节约很多支持的时间。用户在开发之前，你只要提供这个文档给他看，然后有问题再来咨询。

第三就是要把握一个接入节奏，因为我们整个平台的开发人员比较少，也就三个到四个同学，虽然已经全员客服了去应对各个 BU 的各种各样的

问题，但是如果同时接入太多项目的话还会忙不过来；另外支持还有重要的一点就是“授人以渔”，在支持的时候给他们讲得很细吧，让他们了解 Kafka 和 Storm 的基本知识，这样的话有一些简单问题他们可以内部消化，不用所有的问题都来找你的团队支持。

新的探索

前面讲的是我们基本上去年的工作，今年我们在两个方向上做了一些新的尝试：Streaming CQL 和 JStorm，和大家分享下这两个方面的进展：

Streaming CQL 是华为开元的一个实时流处理的 SQL 引擎，它的原理就是把 SQL 直接转化成为 Storm 的 Topology，然后提交到 Storm 集群中。它的语法和标准的 SQL 很接近，只是增加了一些窗口函数来应对实时处理的场景。

下面我通过一个简单的例子给大家展示一个简单的例子，给大家有个直观的感受。我的例子是

- 从kafka中读取数据，类型为ubt_action；
- 取出其中的page，type，action，category等字段然后每五秒钟按照page，type字段做一次聚合；
- 最后把结果写到console中。

```
public class CtripKafkaSpout implements IRichSpout {
  ……
}

class ExtractBolt extends
AbstractMuiseHermesBoltAutoAcked<UserAction> {
……
}

class ConsoleBolt extends CtripBaseBoltAutoAcked {
  ……
}

class AnalyseBolt extends CtripBaseBoltWithoutAutoAcked {
……
}

public static void main(String[] args) {
  ……
  CtripStormSubmitter.submitToCluster(conf, builder);
  ……
}
```

```
create input stream kafka_avro (context__page
String, context__type String, action__category
String , action__type String)
serde "HermesSerDe"
source "HermesSourceOp"
properties(avroclass="hermes.ubt.action.UserAction
",topic="ubt.action",groupid="json_hermes");

create output stream console_field(page String, type
String, target String , actionType String, count Int)
sink consoleoutput;

insert into stream console_field select *,count(1)
from kafka_avro [range 5 seconds batch] group by
context__page, context__type;

submit application ubt_cql_demo;
```

如果需要用Storm实现的话，一般你需要实现4个类和一个main方法；使用 Streaming CQL 的话你只需要定义输入的 Stream 和输出的 Stream，使用一句 SQL 就能实现业务逻辑，非常简单和清晰。

那我们在华为开源的基础上也做了一些工作：

- 增加Redis，Hbase，Hive（小表，加载内存）作为Data Source；
- 增加Hbase，MySQL / SQL Server，Redis作为数据输出的Sink；
- 修正MultiInsert语句解析错误，并反馈到社区；
- 为where语句增加了In的功能；
- 支持从携程的消息队列Hermes中读取数据。

Streaming CQL 最大的优势就是能够使不会写 Java 的 BI 的同事，非常方便地实现一些逻辑简单的实时报表和应用，比如下面说到的一个度假的例子基本上 70 行左右就完成了，原来开发和测试的时间要一周左右，现在一天就可以完整，提高了他们的开发效率。

【案例】度假 BU 需要实时地统计每个用户访问“自由行”、“跟团游”、“半自助游”产品的占比，进一步丰富用户画像的数据：

- 数据流：UBT的数据；
- Data Source：使用Hive中的product的维度表；
- 输出：Hbase。

今年我们尝试的第二个方向就是 JStorm，Storm 的内核使用 Clojure 编写，这给后续深入的研究和维护带来了一定的困难，而 JStorm 是阿里开源的项目，它完全兼容 Storm 的编程模型，内核全部使用 Java 来编写，这就方便了后续的研究和深入地调研；阿里的 JStorm 团队非常 Open，也非常专业化，我们一起合作解决了一些在使用上遇到的问题；除了内核使用 Java 编写这个优势之外，JStorm 对比 Storm 在性能上也有一定的优势，

此外它还提供了资源隔离和类似于 Heron 之类的反压力机制，所以能够更好的处理消息拥塞的这种情况。

我们现在基本上已经把三分之一的Storm应用已经迁到JStorm上了，我们使用的版本是 2.1；在使用过程中有一些经验跟大家分享一下。

第一点是我们在与 kafka 集成中遇到的一些问题，这些在新版本中已经修复了：

- 在JStorm中，Spout的实现有两种不同的方式：Multi Thread（nextTuple，ack & fail方法在不同的进程中调用）和Single Thread，原生的Storm的Kafka Spout需要使用Single Thread的方式运行；
- 修复了Single Thread模式的1个问题（新版本已经修复）。

第二点是 JStorm 的 metrics 机制和 Storm 的机制完全不兼容，所以相关的代码都需要重写，主要包括适配了 Kafka Spout 和我们 Storm 的 API 中的 Metrics 和使用 MetricsUploader 的功能实现了数据写入 Dashboard 和 Graphite 的功能这两点，此外我们结合了两者的 API 提供了一个统一的接口，能兼容两个环境，方便用户记录自定义的 metrics。

以上就是我要分享的内容，在结尾处，我简单总结一下我们的整体架构（见图 6）。

底层是消息队列和实时处理系统的开源框架，也包括携程的一些监控和运维的工具，第二层就是 API 和服务，而最上面通过 Portal 的形式讲所有的功能提供给用户。

未来方向

在分享的最后，我来和大家聊聊实时数据平台未来的发展方向，主要有两个：

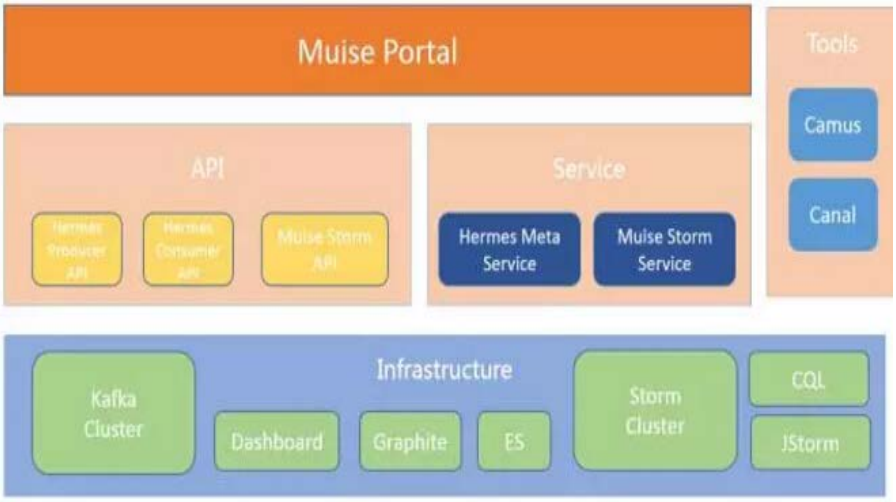

图 6

- 继续推动平台整体向JStorm迁移，当然我们也会调研下刚刚开源的Twitter的Heron，与JStorm做一个对比；
- 对于dataflow模型的调研和落地，去年google发表了dataflow相关的论文（强烈建议大家读读论文或是相应的介绍文章），它是新一代实时处理的模型，能在保证实时性的同时又能保证数据的正确性，目前开源的实现有两个：Spark 2.0中Structured Streaming和Apache的另一个开源项目BEAM，BEAM实现了Google Dataflow的API，并且在Spark和Flink上实现了相应的Executor。

下半年我们还会做一些调研和探索性的尝试，并寻找合适的落地场景。

作者介绍

张翼，携程大数据平台负责人，浙江大学硕士毕业，2015 年初加入携程，主导了携程实时数据计算平台的建设，以及携程大数据平台整合和平台技术的演进。进入互联网行业近 10 年，从事大数据平台和架构的工作超过 6 年。

基于 Lambda 架构的股票市场事件处理引擎实践

作者 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：

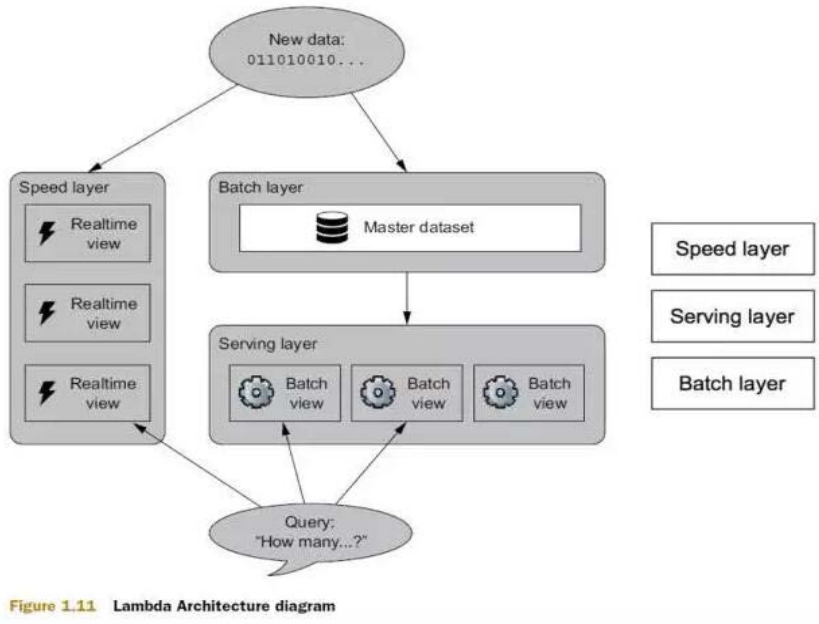

图 1

- 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，如图 2 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。

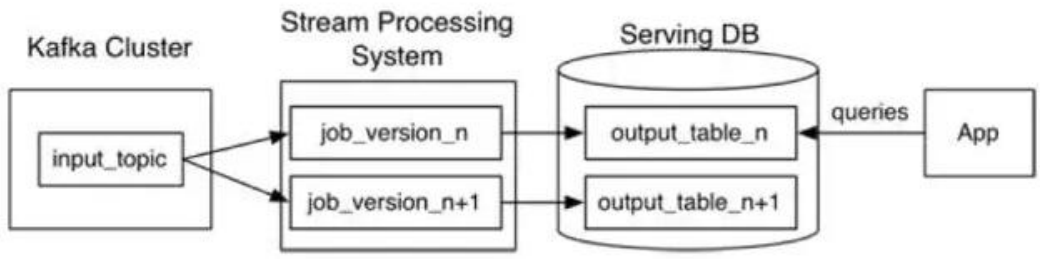

图 2

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

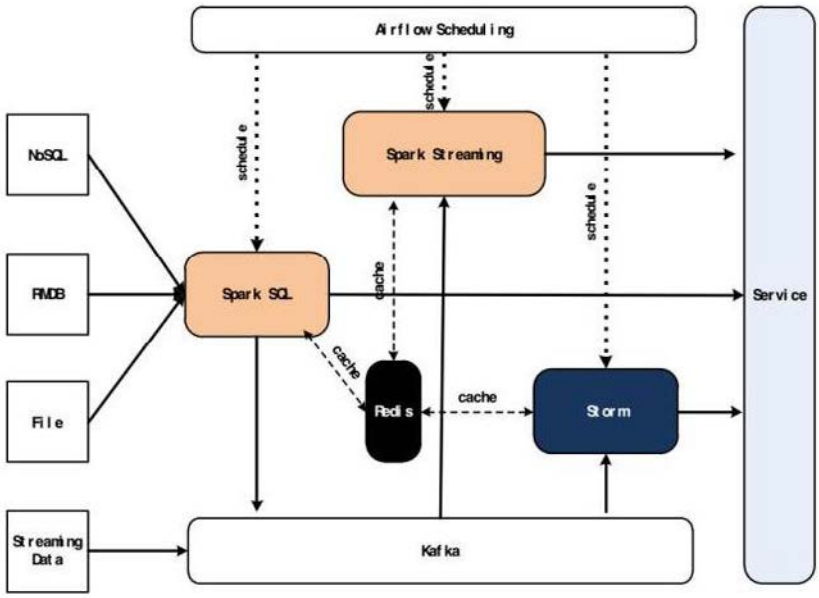

图 3

“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。架构实现如图 3 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。数据处理流程如图 4 所示。

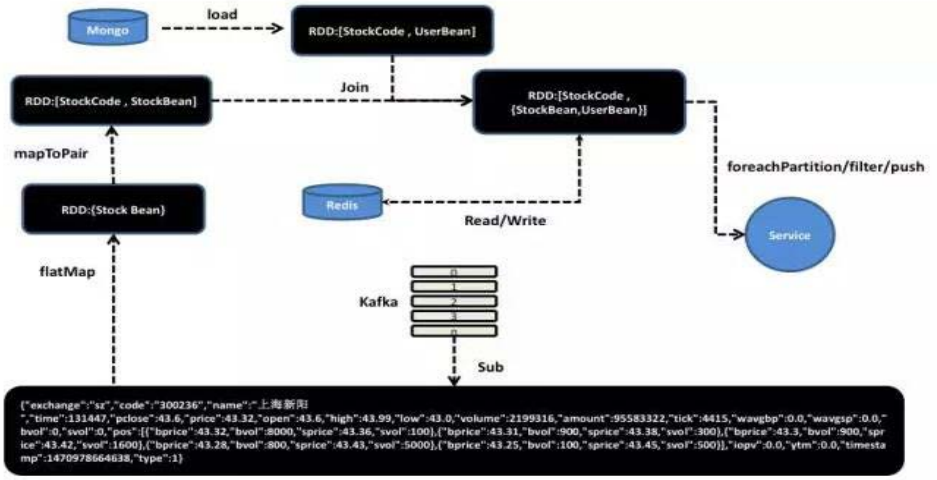

图 4

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。整个处理流程如图 5 所示。

在图 5 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

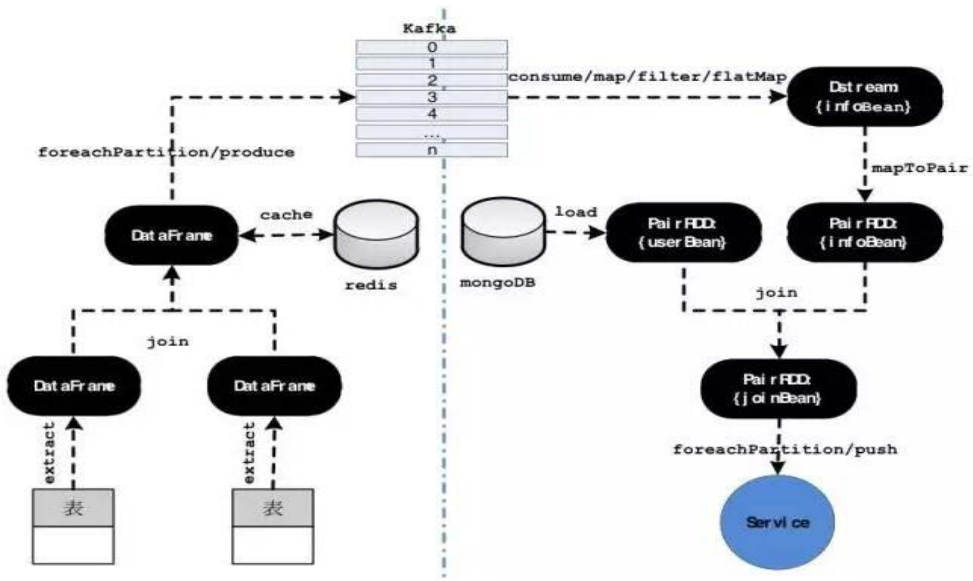

图 5

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

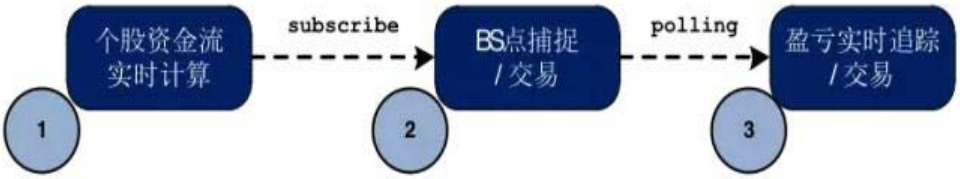

图 6

部分，见图 6。

（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）

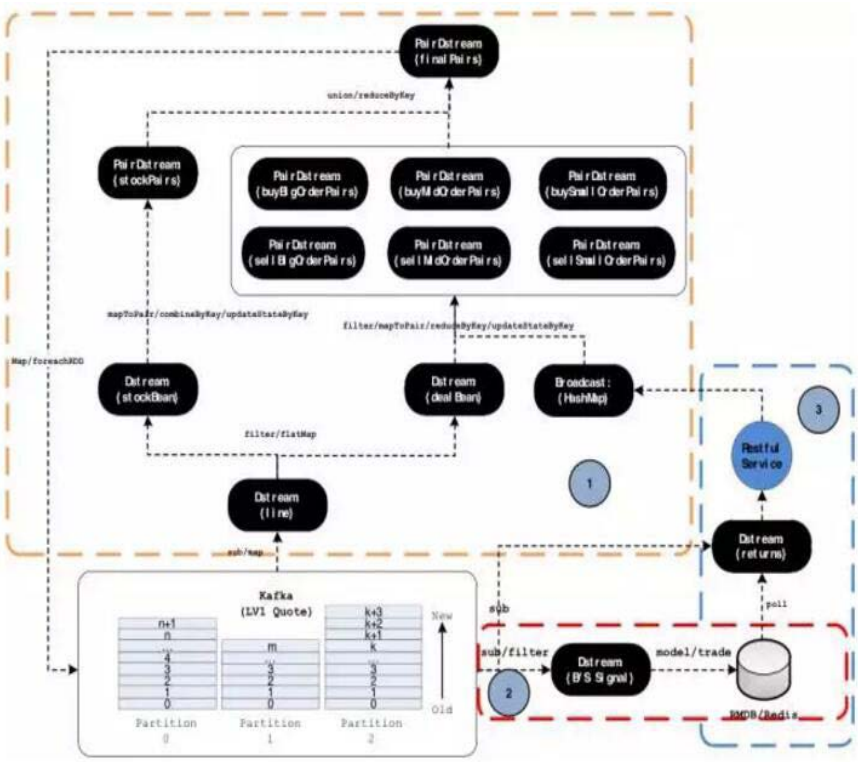

图 7

（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）。

多形态 MVC 式 Web 架构的分类

作者 Brent Chen 译者 Rays

引言

MVC（模型 - 视图 - 控制器，Model-View-Controller）最初用于设计和实现狭义的桌面图形用户界面（GUI）应用开发。经典的 MVC 与面向对象的程序开发方法一样，已成为每一代软件开发人员所最早接触到的软件开发原则之一。虽然 MVC 对当今的工业界有着如此重要的影响，但是在日益互联计算的时代，很明显 MVC 的内涵已迷失了其精准性。这在过去 20 年间对于 WUI（Web 图形界面，Web Graphical Interface）开发领域尤其是如此。基于以上原因，本文意在对 MVC 的起源做概要地阐述之后，进而深入地探讨基于 Web 的 MVC 的演进和变化。

在 WUI 的应用场景下，原始 MVC 及 MVC 三元组中成员对象的历史意义和内涵在不断地演变和转变形态。出于消除任何概念上的混淆的考虑，在本文的探讨中将使用“WMVC”一词表示基于 Web 的 MVC 式架构模式。大多数技术平台的 WMVC 特性正处于不断地改进之中，这包括 Microsoft 的

ASP.NET MVC、PHP 的 Symfony、Python 的 Django、Ruby 的 Merb、Java 的 JSR 371 等。驱动这些平台改进的原则，很大程度上在于 JavaScript 已可由客户端浏览器运行。此外，不少新的网络协议充实了服务器 - 客户间的通信。

自 JavaScript 被 XMLHttpRequest 赋予新生以来，它就成为 WUI 开发中情有独钟的技术。在过去的数年内，就有超过二十种 基于 WMVC 的 JavaScript 应用萌芽发展，其中包括 Dojo、 Angular、 Ember、Backbone 和 React 等。即使不是全部的也是绝大部分的框架都是侧重于客户端组件的交互，对于 WMVC 视图和控制器的组成对象而言尤其是如此。自然而言，WUI 开发方法上的革新再一次引发了针对 MVC 亦或 WMVC 的大量讨论。这些讨论时常是十分激烈的，通常侧重于某个特定的方面，或基于某个特定的环境。进而使得 MVC（WMVC）衍生出一些主要差异在于控制器对象的变体，其中包括 MVA、 MVP、 MVVM、 Flux、 Redux 和 SAM 等。这些 MVC 变体时常被统称为“MV*”，其中的“*”表示了各变体间的差异主要在于视图和控制器间的交互方式。通常并不将表示现实世界视图对象的模型组件整体地列入考虑中，或仅是在 Web 应用方案中将模型作为 MVC 三元组关系里的被动参与者。

自 MVC 概念于四十多年前被提出以来，MVC 模型很有可能已经历了最重要的改变。在本文的探讨中，我们对 MVC 模型做了一个宽泛的定义，这个定义中涵盖了驻留内存的模型对象（例如记录集对象），还有用于支撑对象的 SoR（主数据记录系统，System of Record）中的源数据、源文档、源文件和原始信号，以及所有把它们同步和聚集到一起的过程。模型数据仓库的存储形式已从小型软盘发展到 RDBMS 和 MMDBMS （多模数据库管理系统，multi-model database management system）。早期模型数据仓库

是与驻留内存的模型对象共处一处的，这些模型对象是独立存在于各个用户桌面之上的。现在模型数据仓库使用宽带连接、分布式或基于云的系统，其部署可远离域对象。存储在这种外部环境中的数据，可以被企业生态系统的多个系统修改，也可以被消费者应用的数以千计的用户修改。这种使用场景上的根本差异，已经在根本上地改变了模型对象的行为，进而改变了模型对象与 MVC 三元组中的另两个成员间的通信和交互。

原型 MVC

oMVC （原型 MVC，Original MVC）是由挪威计算机科学家 Trygve Reenskaug 于 1978 年提出，当时他工作于著名的 Xerox PARC 研究中心 Smalltalk 团队中。在 oMVC 的概念初步成型后，最初被实现为 Smalltalk-80 类库的一部分，用于桌面 GUI 的建立。

在那个时代，桌面应用远非当前这样是日常家居中的常见物品，每个应用需要独立运行于一台机器上，而每台机器的存储是有限的并且是各自独立的。正如在图 1 中所示，对 oMVC 模型对象的任何操作都完全由用户行为通过控制器所触发，MVC 三元组在受控的环境中进行通信、同步并保持状态。控制器的主要作用是维持用户和系统之间的联系（参见图 1）。控制器实现对相关 GUI 组件的部署，并在屏幕上将这些 GUI 组件展现给用户。在 Win95 出现之前的年代中，这是一个具有挑战性的任务，因为当时 MS-DOS 依然是占据主要地位的操作系统。在 oMVC 模型中，一旦用户产生了某种动作，例如菜单选取、在输入框中输入、按钮点击等，控制器就会转化这些动作为对应的改变消息，向模型传递这些消息，并对消息进行处理。

图 1 中显示了作为模型的组成部分的本地软盘数据存储。oMVC 中控

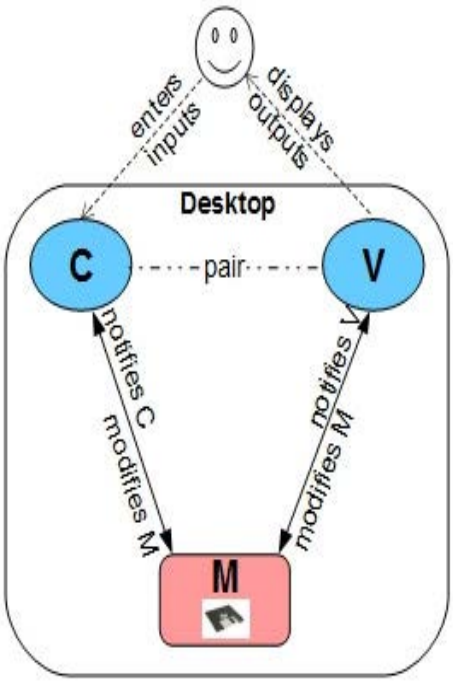

图1 展示了独立桌面环境中的oMVC

制器和视图具有成对出现的关系，但是模型的改变并不在两者间直接产生通信。控制器和视图知道模型的状态，但是反之并非如此。

一旦模型从控制器获取了变更通知，它并不直接通过调用去更新所涉及的视图。在模型－视图的关系中，变更通知的获取是通过每个独立的参与者之间的相互注册实现。一旦一方发生了变更，就会生成一个事件，对方就会采取相应的动作作为响应。在图 1 中，视图是附属于模型的，对模型进行观察。一旦控制器触发了模型变更事件，视图必须去确保自身显示也按需更新，以相应地反映出模型的状态。更新模型的通知信号也可由视图自身生成。这种视图与模型之间的观察（或订阅）和通知关系，有助于模型和视图间的解耦，使得多个视图可隶属于同一模型，进而可提供不同的表示。

在图 1 中值得注意的是，控制器并非直接地改变视图。在 Smalltalk 社区及其它后续的桌面 GUI 应用库中，通常将视图和控制器看成一对耦合的对象。视图使用具有特定控制器类型的实例实现预期的响应。为创建所期望的行为，控制器还可以在不同情景中策略性地、动态地切换类型。这样的视图－控制器对可以嵌套于复合层次结构中（如图 2A 所示）。在该层次结构中的父辈之间、子女之间及父子之间，视图－控制器组件都可以进行交互和通信。很多情况下，层次结构中的每个独立视图－控制器的子女组件仅处理部分的模型对象。此外，从 oMVC 组合的角度看，模型也可

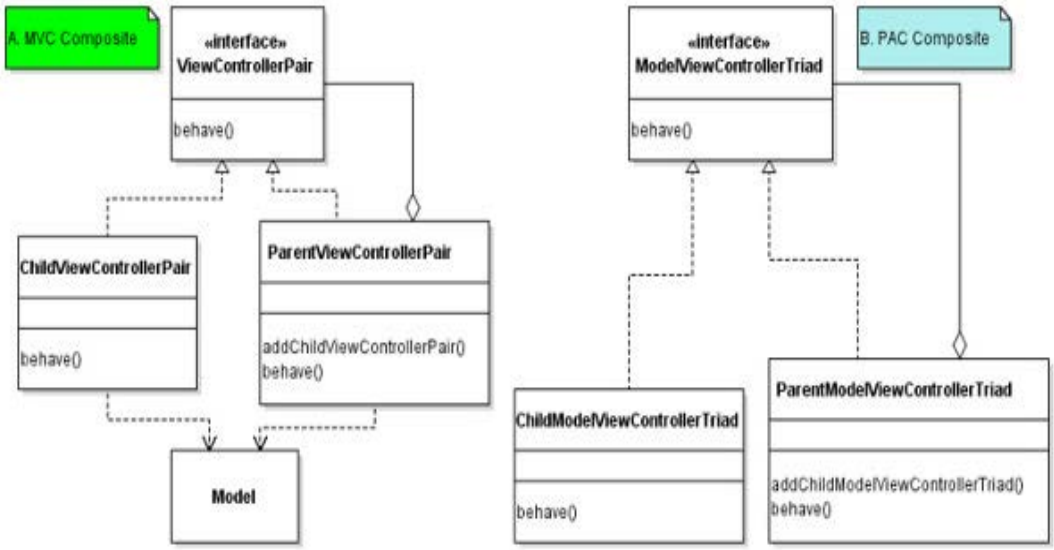

图2 左图是oMVC的复合表示，右图是PAC（显示-抽象-控制，）的表示

以使用层次结构的形式进行组织，这样 MVC 三元组作为一个整体构成层次结构中的父子关系，如图 2B 所示。

总而言之，oMVC 设计范例是由一系列的 GoF 设计模式所组成，尤其是其中包括了观察者模式、策略模式和复合模式。在早期的桌面应用库中，oMVC 域架构和模式的设计意图得以保持，即模型组件由应用域对象和数据存储所组成，其中存储多是本地的或是受限的。在 oMVC 三元组类的行为中，应用域行为的维护和广播起着核心的作用。oMVC 的一个关键假设是模型的稳定性，该假设在上世纪七十和八十年代的桌面应用情景中显然是正确的。但是在 WMVC 领域，模型时常发生改变通常是一种常态。直至近些年，向用户实时广播变更（类似于 oMVC 所实现的）所需的技术基础才得以实现。

WMVC 的分类

具有讽刺意味的是，虽然直到上世纪九十年代，尤其是在 Win95 出现之后的年代，桌面计算机开始进入普通百姓家并得到普及，但是传统的桌

面应用却因为因特网互联的 Web 应用开始统治了业界而逐渐退居幕后。不同于安装并于运行于终端用户计算机上的桌面应用，Web 应用是宿主于远离用户的服务器上，创建了客户 - 服务器的关系。在本文的其后内容中，我们将浏览器（browser）和客户（client）这两个概念互换使用。根据浏览器和服务器相对于 WMVC 三元组对象的部署位置和执行方式的不同，可将 WMVC 明确地分组为：

- 服务器端WMVC （Server-side WMVC，sWMVC）：所有WMVC的组件位于服务器上，并在服务器上执行。
- 双重WMVC（Dual WMVC，dWMVC)：WMVC组件分布于浏览器和服务器之间。通信可由客户或者服务器端发起。
- 点对点WMVC（Peer-to-Peer WMVC，pWMVC）：这种架构中没有集中式服务器。所有WMVC组件位于客户端，在客户端执行。pWMVC可具有自己的沙箱SoR。

服务器端 WMV（sWMVC）

在 sWMVC 模型中，用户使用浏览器作为瘦客户，通过无状态的请求 - 响应 HTTP 协议访问应用（如图 3 所示）。客户向服务器发送 HTTP 请求或输入内容，接收并显示整个更新的 Web 页面（或是其它的文档）。一旦页面被加载以后，各页面组件间就很少有交互了，页面成为静止的。

在这种瘦客户 - 服务器范式的 sWMVC 架构中，应用 SoR 版本库外部化为一种集中式环境。它与应用服务器内存中的域对象是相互分离的。服务器和数据版本库都是远离用户浏览器部署的（如图 3 所示）。SoR 存储常常是由一个或多个关系数据库这样的数据源所组成。鉴于数据已经外部化了，带外进程或不同的用户都可对数据进行更新。数据的变更只会从控制器流向模型（参见图 3）；当 SoR 中数据被不同的用户或系统改变时，并

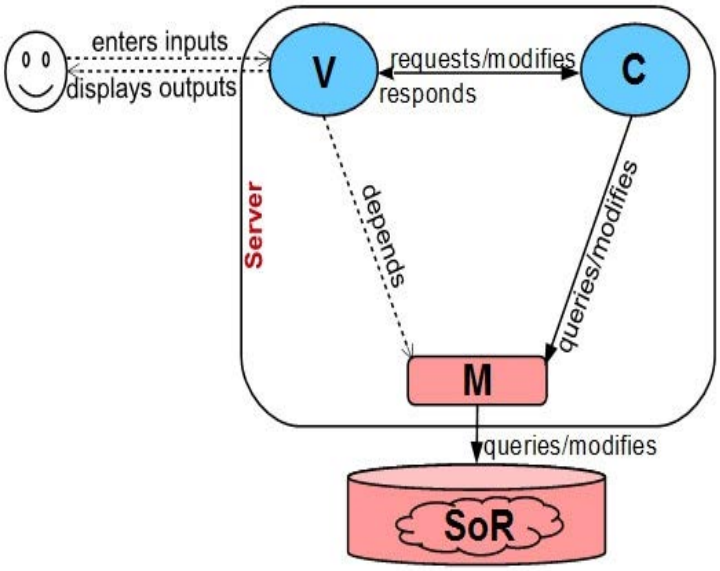

图3 sWMVC范例的一种表示

不向应用服务器发送入站数据变更通知。不同于 oMVC，模型及其所关联的视图间不再有任何的直接联系和必要的同步。由于视图不再反映模型的状态，这就需要用户手动地发起新的 HTTP 请求去同步和刷新视图。因此 sWMC 中的“s”，也可指代这种 WMVC 范例的静态（static）或是陈旧（stale）的本质特性。

图 3 中显示了由服务器驻留内存的域对象模型（M）外部化而得到的 SoR。不同于 oMVC，控制器不再与用户交互，而是用于协调模型与视图间的通信。模型数据的变更是单向流动到外部数据存储中的。

WMVC 的模型可简述为一种分层架构（如图 4 所示）。架构的最顶层模型表示了在视图及其相关的控制器这个关系对之间的契约关系。为满足该契约，在处理栈中可包含数个可变的业务逻辑和数据访问组件。该契约用于所有与视图相关信息的连接，其中的信息可能直接来自本地 SoR，或是数据云，或是其它真实世界中的数据源，例如探测器和数据提供系统。

图 4 中显示 WMVC 的组成包括了多个架构层次，以及来自本地或远程

可访问环境中的可变数据源。

相对于在 oWMVC 中的意义和内容（如图 3 所示）而言，三元组对象间的关系和通信发生了根本上的转变。与控制器在用户和 oMVC 系统（图 1）间所起的作用不同，sWMVC 的控制器承担了在更高层级上协调视图和模型的角色（参加图 3）。视图和模型间的通信通过控制器发生。用户与视图进行交互，将模型与模型视图关联的行为逻辑包含在控制器中，由控制器负责管理输入、更新模型并产生适当的输出。

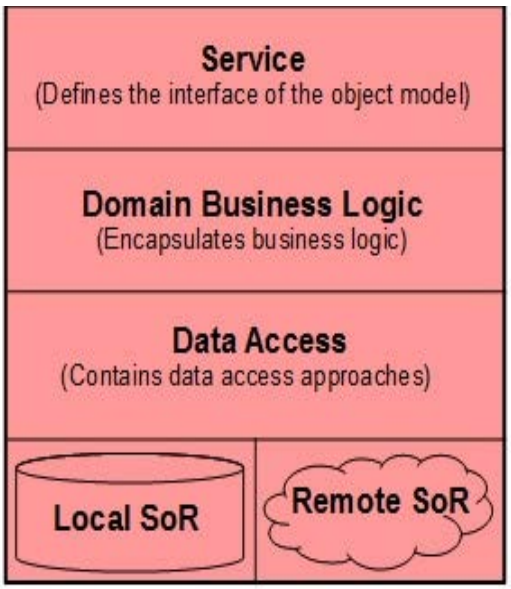

图4 WMVC模型的示意图

Web 应用的架构环境与桌面应用的有所不同，对于将 MVC 引入到早期 Web 应用架构的设计中，虽然这在本质上忽视了 MVC 的原生结构。但是这种引入反而接收了 MVC 模型的基本理念，即将这三个相互作用的、多变责任的对象类进行相互分离是十分重要的。该做法将 oMVC 范例提升为 UI 设计中在更加通用层次上的架构原则，增加了对任意类型 UI 应用的灵活性和可维护性。

sWMVC 中通常包含有多种设计模型，其中也包括在 oMVC 中所使用的设计原则。例如，各个用例的控制器原则和行为可能都是不同的，包括中介者模式、调度者模式、使用状态模式和模板方法模式的策略模式代理等。但是在 oMVC 中，观察者模式在触发三个对象类型间的通信中起着关键的作用，尤其是当该模式与视图做相关的更新时。在 sWMVC 实现中，观察者的作用被缩减了。这样导致了 sWMVC 的静态和陈腐的特征。直至近些年，

技术的进展才使得交互式和富 Web 用户体验成为可能。

双重 WMVC（dWMVC）

在 Web 时代的早期，由于浏览器的普遍使用，为使现有的桌面交互应用支持 Web 做出了不懈的努力。这些工作的一个成果体现为 Microsoft 的 Outlook Web 项目所研发的一种基于 Web 的新组件，该组件允许客户端脚本发布异步 HTTP 请求到服务器。该工作引导了 XMLHttpRequest (XHR) 协议的建立和标准化。该协议是至上世纪末为止最具有革命性的工作。XMLHttpRequest 随后成为 Ajax 技术的基本原则。

当前，基于这种技术的架构使得 dMWVC 的视图 - 控制器对可以动态分布，并可安装到用户桌面、移动设备或其它设备的浏览器上。浏览器组件与服务器上模型间的异步通信，用于同步给定的或者所有的视图元素的 dWMVC 三元对象组状态，进而提供给用户对现实世界或模型对象的最新认知。这种技术的引入使得 XHR 切实地丰富了客户应用，为浏览器用户带来了交互式的用户体验。

XHR 技术还引发了单页应用 (SPA) 开发技术的发展。类似于 oMVC 应用，SPA 是驻留在单一页面上的 Web 网站，提供了无缝的浏览体验。对于用户而言，SPA 看上去在从一个页面到另一个页面时并没有任何页面重载，对不同页面渲染所需的资源是在所需时由幕后的服务器动态且异步加载的。SPA 和这种 dWMVC 应用的交互行为的组合使此类范式划分为了富互联网应用 (RIA) 的最新条目。

就 XHR 协议自身而言，从客户端的视图 - 控制器对象到服务器端模型间的通信依然仅是单向的通信，需要由视图发起某种类型的轮询机制去探测模型上的更改。轮询是一种资源密集型操作，因而成为影响性能的一

个关注点。类似于图 1 所示的 oMVC，在理想情况下模型中的任何更改应会被实时发布到、或是广播到所有相关的视图 - 控制器组件中。图 5 显示了 dWMVC 架构的视图和模型间的双向异步同步（通过控制器）。该机制提供了完全动态的状态同步，这种同步基于 dWMVC 组件间的订阅 - 发布机制。一些近期的技术发展，其中包括服务器发送事件 （Server-Sent Events，SSE）、WebSocket 和入站数据库通知技术等，使得该同步操作成为可能。

图 5 中显示了三元组对象在客户和服务器间的分布。由 SoR 发起的入站更改通知导致了模型和视图间的实时双向更新。

SSE 机制作为 HTML5 的组成部分，允许服务器端组件异步启动，并将数据从服务器端组件实时推送到浏览器。客户端组件可以使用 SSE 发起请求，向服务器申请建立一个非传统的 HTTP 连接。一旦客户端组件接收到所发起的请求，它继续对随后的服务器响应进行监听。与此同时，服务器保持同一客户 - 服务器初始连接是活跃的。一旦新的数据可用，服务器无

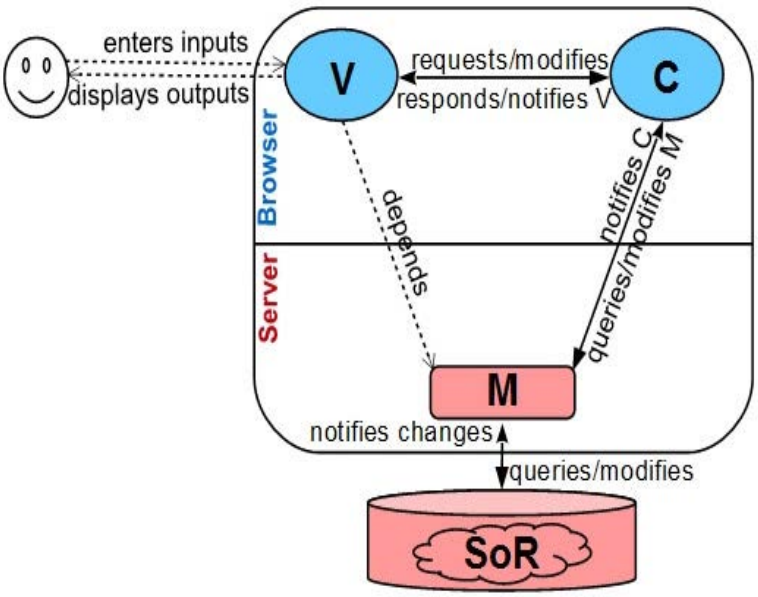

图5 dWMVC的示意图

需客户的额外请求，就立刻通过该初始连接将这些数据推送给客户。这样，SEE 给出了一种从服务器到客户浏览器的解决方案，该解决方法使用了异步的发布 - 订阅事件通知。

WebSocket 是一种提供浏览器和服务器间全双工连接的通信协议。当客户发送初始请求到服务器时，WebSocket 通知服务器该 HTTP 连接可能会升级为全双工的 TCP/IP WebSocket 连接，通知使用了一种特殊的 HTTP 报头。一旦 WebSocket 连接被建立，就可在需要时用于在浏览器和服务器间相互发送数据。

使用 SSE 和 WebSocket 通信协议，服务器可以异步地将模型驻留内存的更改发布到浏览器。但是正如前面所讨论过的，模型架构的分层可能会跨越服务器的边界，并可能包括应用服务器之外的外部 SoR（参见图 4）。由于 SoR 中的数据记录可被其他应用或用户修改，因此每当这样的带外更改发生时，SoR 应具有变化数据捕获 （change data capture，CDC）机制去探测并捕获更改，实时地发起并推送入站数据更改到应用服务器，实现端到端的双向发布 - 订阅通信模型（如图 5 和图 6 所示）。在图 6 中，dWMVC 轮毂的中心表示了共享的 SoR 和 CDC 数据源。数据的实时同步由数据源和相关驻留内存域模型对象之间的双向交互所维持。在图 6 中，每个轮辐间的微型 dWMV 表示了一个独立的业务应用（集成的企业生态系统中），或是一个独立的应用用户。

图 6 中显示了由 SoR 发起的入站更改通知，并显示所引发的模型和所有视图间的实时双向更新情况（以微型 WMVC 展示）。

基于 Web 的架构栈中加入了 XHR-Ajax、SSE 和 WebSocket 等技术，还有数据库厂商提供了数据库入站通信能力，所有这些一起使得传统 oMVC 中视图和模型间的双向交互通信在 dWMVC 中得到了复兴。越来越多的数

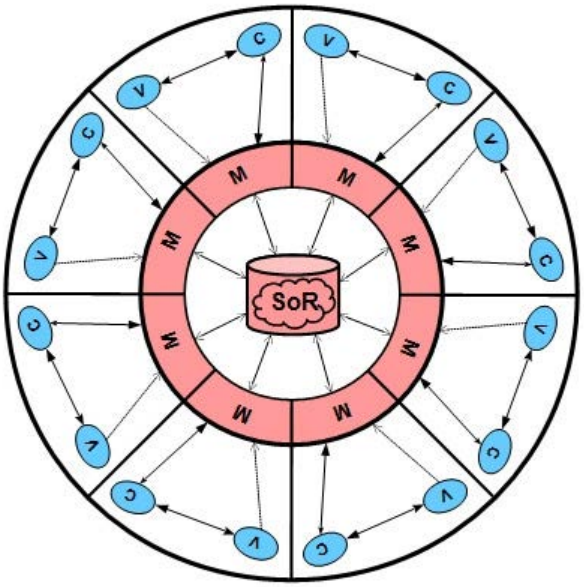

图6 dWMVC轮图

据库厂商，包括 PostgreSQL、Oracle 等传统关系数据库厂商和 RethinkDB、Cassandra 等 NoSQL 数据库厂商，已经实现或者规划去提供 CDC 机制，以改进应用服务器的入站推送通知。

点对点 WMVC（pWMVC）

对于上面所论及的两种 WMVC，它们的架构语义都是基于客户 - 服务器范例的，即用户浏览器向服务器发送 HTTP 请求，取回服务器端的内容，服务器则回应以包含请求信息的一个或更多的响应。这类方法中，服务器负责存储，并发送所有内容及对所有请求的响应。但是这样的集中式方法可能会导致性能上的瓶颈。因为为了支持所有可能的请求负载，需要对服务器基础设施的资源进行适当地扩展和复制。当前由于高频度实时内容发布需求在数量上日益增长，这类方法会产生问题，尤其是在（意料之外的）高通量负载的期间。最优的系统无疑是那种能以去中心化的方式支持高质量终端用户体验的系统，这样的系统可以使用户用最短的路由和时间获取到数据。借助于点对点（Peer-to-peer，P2P）数据交换及通信，终端用户可以从其它的用户那里交互、检索和接收内容。无疑，这样的架构绕开或降低了集中式服务器的负载和潜在瓶颈问题。

传统的点对点系统需要用户显式地安装专用的桌面应用或插件。在 WebRTC（Web 实时通信，Web Real-Time Communication）协议被标准化

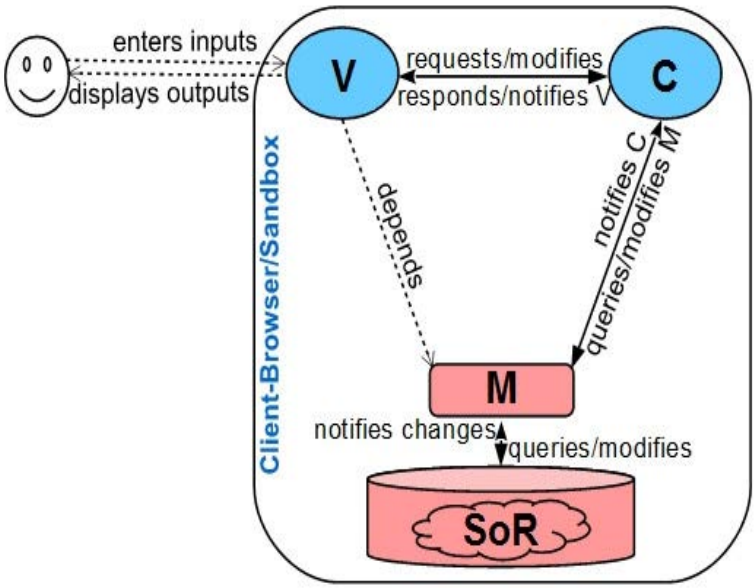

图7 pWMVC图

并被浏览器支持之前，浏览器本身并不具备使对等系统的工作直接相互通信的能力。现在 WebRTC 标准已被大多数的浏览器所支持。它是一种 API 定义，提供了无服务器的浏览器到浏览器（浏览器 P2P）直接数据交换和通信范例。WebRTC 对 Web 应用引入了点到点的解决方案，它允许 Web 浏览器打开到其它浏览器的直接通信通道，无需对每个 Web 请求－响应的集中式服务场景进行处理（参见图 7 和图 8）。

图 7 中显示了位于客户端（浏览器）内、或由客户端所控制的所有三元组组件。

在 pWMVC 模型中，所有的三元组组件位于并执行于客户端及相关终端用户的沙箱中（如图 7 所示）。沙箱中包括用户本地或基于云的 SoR 存储，这样的存储可以被 pWMVC 应用访问。在图 8 中，轮辐间的每个微型 pWMVC 都代表了一种独立用户浏览器环境。与图 6 相比，为便于持续进行的 Web 通信，pWMVC 轮图中并未涉及集中式服务器和 SoR 架构。互联浏览器间的微型 pWMVC 组件状态可使用 WebRTC 通信协议保持同步。

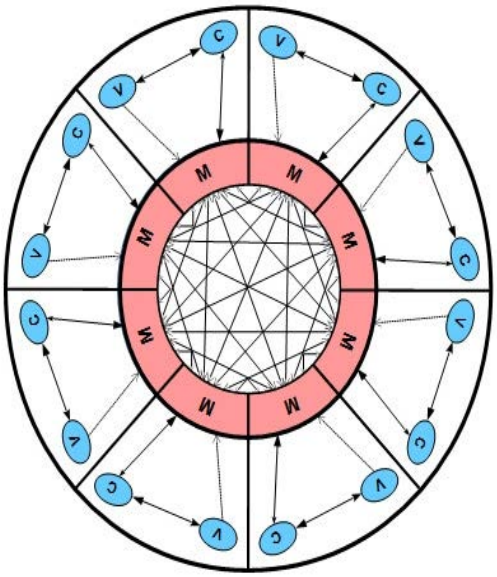

图8 pWMVC轮图

图中显示了由独立用户浏览器所发起的直接无服务器 P2P 变更通知，以及该通知如何导致其它的对等端视图进行实时更新。图中“M”指代内存中模型对象和用户 SoR。

总结

MVC 本身应该被视为是一种无需考虑任何语义的设计原则或方法论。MVC 的简要内涵在于，任何类型的 UI 应用都可被分解为三套相互作用的对象类。在 MVC 所应用的场景中，应该审查该三元对象类型组的行为。为更好地理解 MVC 概念，在对架构和特定域程序库进行实际设计和实现应用时，应使用适当的命名注释。

WMVC 可看成一种独特的原理图，用于在无状态 HTTP 域的场景中基于 MVC 方法论的开发。WMVC 可区分 sWMVC、dWMVC 和 pWMVC 这三种不同的类别。这些类别在机制上不同于 oMVC，即原型 MVC。考虑到标准化的网络协议、由特定数据库技术所提供的专用入站通信等这样的最新技术发展，基于观察更改的 MVC 式“事件循环”可以满足基于 Web 应用的需求。这使得 WMVC 当前可为浏览器用户实现具有完全交互的实时丰富 WUI 体验。

关于作者

自上世纪 90 年代以来，**Brent Chen 先生**就一直致力于系统架构和应用开发。他所提出的解决方案涵盖了众多的专业领域，其中包括：人力资源管理、职工福利管理、监管合规、卫生保健和政府事务等。

架构师 月刊 2016年10月

本期主要内容：专访黄翀：东方航空到底用MongoDB做了什么，技术选型为何花落MongoDB，DevOps的前世今生，微信序列化生成器架构设计及演变，Nginx日志中的金矿，OpsDev将至，IBM中国开发中心吉燕勇：通过Cloud Data Services打造新型认知计算数据分析云平台

云生态专刊 09

《云生态专刊》是InfoQ为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。

顶尖技术团队访谈录 第七季

本次的《中国顶尖技术团队访谈录》·第七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 深入浅出Netty

为了便于大家集中学习Netty，我们把已经发表的相关文章进行汇总和提取，形成一本迷你书，奉献给各位读者。