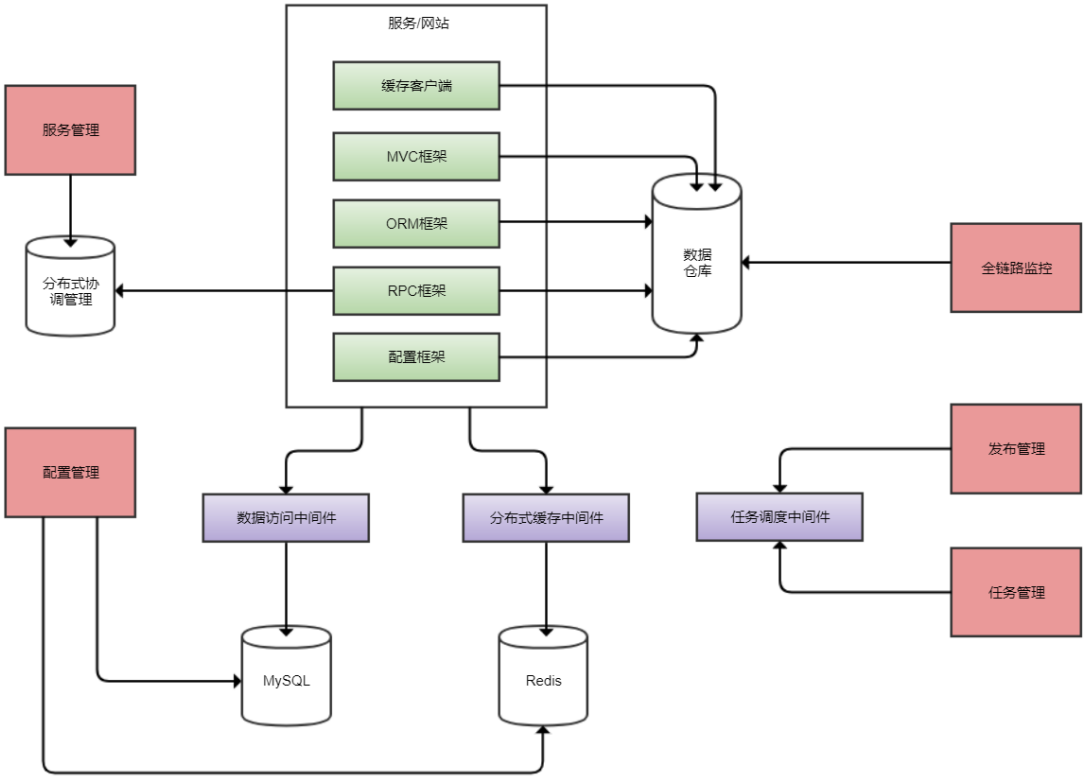
朱晔的互联网架构实践心得 S1E5：不断耕耘的基础中间件

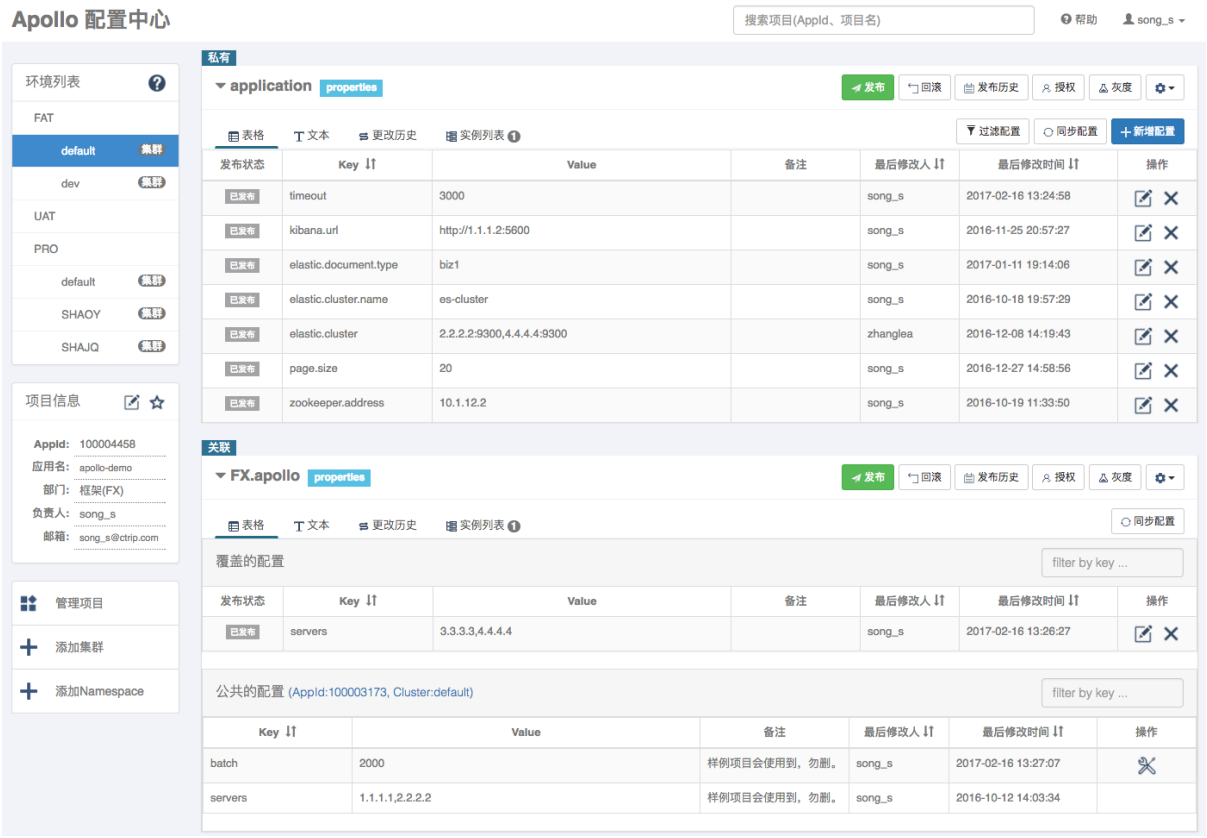
【下载本文 PDF 进行阅读】

一般而言中间件和框架的区别是，中间件是独立运行的用于处理某项专门业务的 CS 程序，会有配套的客户端和服务端，框架虽然也是处理某个专门业务的但是它不是独立程序，是寄宿在宿主程序进程内的一套类库。



图上绿色部分代表了框架，红色部分代表了管理系统，紫色部分代表了中间件。本文会着重介绍管理系统和中间件部分。

配置管理



比较知名的分布式配置服务和管理系统有携程的 https://github.com/ctripcorp/apollo （上图）以及 https://github.com/knightliao/disconf。对于比较大型的互联网项目来说，因为业务繁杂，需求多变，往往各种系统都会有大量的配置，覆盖几个方面：

- 针对系统内部技术层面的各种配置，各种池的大小、 队列的大小、日志级别、各种路径、批次大小、处理间隔、重试次数、超时时间等。
- 针对业务运营层面的各种配置，活动的周期奖励、黑白名单、弹窗、广告位等。
- 针对运维和发布层面的配置，灰度名单、注册中心地址、数据库地址、缓存地址、MQ 地址等。

因为一些基础组件比如 SOA 框架和发布系统也会用到配置，这个时候就会可能会有鸡生蛋的问题，这里我比较建议把配置系统作为最最底层的系统，其它服务都可以依赖配置系统。一般而言配置管理除了实现最基本的 Key-Value 的配置读取和配置之外，还会有下面的一些特性和功能：

- 高性能。配置服务的压力会是非常吓人的，在一次服务调用中可能就会有几十次的配置调用，如果服务的整体 QPS 在 500 那么配置服务的压力可能在 1 万的 QPS，这样

的 QPS 不走缓存基本是不可能的。好在即使是 1 万甚至是 5 万的 QPS 也不算一个很夸张的无法解决的压力。

- 高可用。现在各种开源的配置服务是所谓的分布式配置服务，由可扩展的配置服务集群来承担负载均衡和高可用功能，配置服务一旦挂了可能会让系统瘫痪。你可能会说配置服务一般本地会有缓存，会有本地的配置文件作为后备，会有默认值，但是因为配置是运营运维在实时修改的，如果某个业务的配置没有使用最新的配置走的是错误的默认值的话，系统会处于完全混乱的状态，所以配置服务的稳定性太重要了。
- 树形的配置体系。如果只是把所有配置堆在一个列表里，加上项目和分类的话，当配置多达几千项的时候还是会有点多。可以支持树形的层级配置，不拘泥于项目和分类这两个条件。项目下可以有模块，模块下可以有分类，分类下可以有小类，根据自己的需求动态构建配置树。
- 好用的客户端。比如可以和 SpringBoot 以及@Value 注解结合起来，非侵入整合配置系统，无需任何代码的改动。
- 毫秒级粒度的修改实时生效。可以使用长连接推的方式实现，也可以实现缓存失效的方式实现。
- 配置的分层隔离。包括按照环境、集群和项目来提供多套配置相互独立不影响，包括可以以层级的方式做配置继承。
- 配置的权限控制。不同类型、环境、集群、项目的配置具有不同的管理权限，比如脱敏只读、只读、读写、导出。
- 配置的版本管理。配置的每一次修改都是一个版本，可以为单独的配置或项目进行直接版本回滚。
- 丰富的 Value 形式。配置的 Value 如果要保存列表的话，保存一个 JSON 阅读和修改都不方便，可以直接提供 List 方式的 Value，在后台可以单独增删改里面的一项。在比如黑名单的引用上这种方式比较高效，否则更新一个名单每次都要修改整个黑名单。这个功能可以和 Redis 结合在一起进行实现。Value 除了支持字符串可以是 JSON 和 XML 形式，系统可以对格式进行格式化，对格式进行校验。Value 也可以是非字符串类型的各种数字格式，系统也会根据类型进行校验。
- 丰富的配置发布生效形式。比如可以自然生效、立即生效以及定时生效。定时生效的功能适合于在某个时间点需要开启某个配置，比如用于面向用户的推送、活动业务。

还有支持灰度自动发布，以一定的时间间隔来对集群里的实例进行发布，避免人工去定期逐一发布单台的麻烦。

- 审核审计功能。配置的修改可以由管理员进行审核（也就是修改和发布的权限支持分离），避免配置错误修改。所有配置的修改记录可以查询到谁什么时候因为什么原因修改了什么配置，事后可以审计审查。
- 配置生效跟踪和使用率跟踪。可以看到每一个配置项现在哪些客户端在使用，生效的值的版本是哪个。通过这个功能还可以排查现在系统中过去一段时间从没有用过的配置，删除无用的配置。
- 动态配置。在 API 设计的时候我们引入上下文的概念，通过传入一个 Map 字典作为上下文，比如某个配置按照不同的用户类型、城市需要有不同的值，这个逻辑我们可以不需要在代码里面手工编写，直接通过在后台配置上下文的匹配策略来动态读取到不同的配置值。
- 本地快照。对配置进行快照本地保存，在出现故障无法连接服务端的时候使用本地的配置。

这里可以看到要实现一个功能完善的配置系统工作量还是相当大的，一个优秀的功能强大的配置系统可以节省很多开发的工作量，因为可配置部分的功能基本就是由配置系统直接实现了，无需在数据库中在搞大量的 XXConfig 表（不夸张的说，很多业务系统 40%的工作量在这个上面，不但需要做这些配置表还需要配以配置后台）。

服务管理

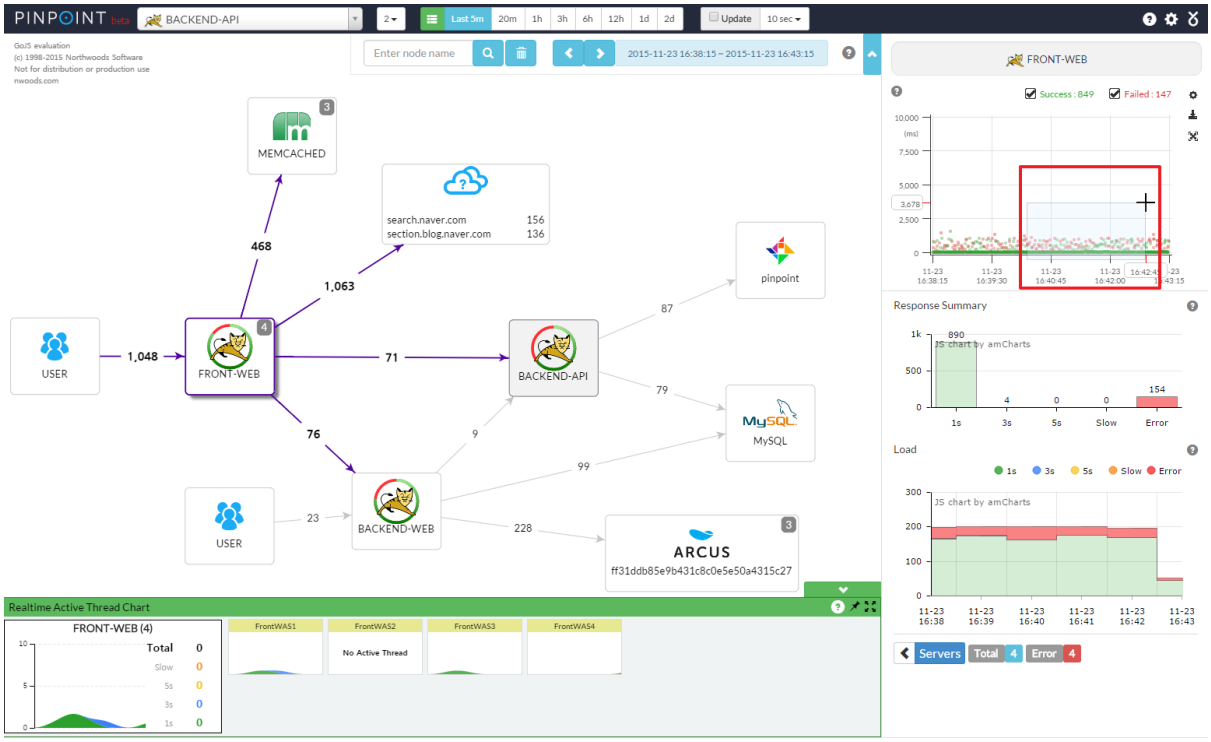
微服务的建设中实现远程调用只是实现了 20%的工作量（但是确实满足了 80%的需求）。服务管理治理这块有大量的工作要做。这也就是实现自己 RPC 框架的好处，这是第一步，有了这第一步让数据流过我们自己的框架以后我们接可以做更多的事情，比如：

- 调用链跟踪。能否记录整个调用的情况，并且查看这个调用链。下面一节会再说一下这点。
- 注册管理。查看服务的注册情况，服务手动上线下线，集群切换，压力分配干预。
- 配置管理。配置服务端客户端线程池和队列的配置，超时配置等等。当然，这个也可以在配置系统中进行。

- 运维层面的管理。查看和管理方法熔断，进行并发限流配置，服务权限黑白名单配置，安全方面的配置（信息加密，日志脱敏等）。
- Service Store 的概念。服务发布需要满足一定要求，有文档（比如可以通过注解方式在代码注释里提供），有信息（开发负责人、运维负责人，服务类型，提供的能力），满足要求后就可以以类似于苹果 App Store 发布程序的方式发布服务，这样我们就可以在统一的平台上查看服务的维护信息和文档。
- 版本控制调用统计。对服务进行灰度升级，按版本路由，不同版本调用分析等等。类似于一些应用统计平台提供的功能（友盟、TalkingData）。

这里我想说的理念是，服务能调用通是第一步，随着服务数量变多，部署方式的复杂化，依赖关系复杂化，版本的迭代，API 的变更，开发人员和架构师其实急需有一套地图能够对服务能力的全貌进行整体的了解，运维也需要有系统能对服务进行观察和调配。服务治理的部分完全可以以 iOS 那套（开发符合时候需要符合标准+发布的时候需要有流程）方式来运作。

全链路监控



开源的实现有 https://github.com/dianping/cat 以及 https://github.com/naver/pinpoint（上图）等等。对于微服务比较多的（主流程涉及 8+微服务）系统，如果没有服务的全链路调用

跟踪那么排查故障以及性能问题就会很困难了。一般完善的全链路监控体系不仅仅覆盖微服务，而且功能也会更丰富，实现下面的功能：

- 以 Log、Agent、Proxy 或整合进框架的方式实现，尽可能少的侵入的情况下实现数据的收集。而且确保数据的收集不会影响到主业务，收集服务端宕机的情况下业务不影响。
- 调用跟踪。涉及到服务调用、缓存调用、数据库调用，MQ 调用，不仅仅可以以树的形式呈现每次调用的类型、耗时、结果，还可以呈现完整的根，也就是对于网站请求呈现出请求的完整信息，对于 Job 任务呈现出 Job 的信息。
- JVM 的信息（比如对于 Java）。呈现每一个进程 JVM 层次的 GC、Threads、Memory、CPU 的使用情况。可以进行远程 Stack 的查看和 Heap 的快照（没有进程的内存信息，很多时候基于服务器层面粗粒度的资源使用情况的监控，基本不可能分析出根本原因），并且可以设定策略进行定期的快照。虚拟机的信息查看和调用跟踪甚至可以通过快照进行关联，在出现问题的时候能够了解当时虚拟机的状态对于排查问题是非常有好处的。
- 依赖关系一览。有的时候我们做架构方案，第一步就是梳理模块和服务之间的依赖关系，只有这样我们才能确定影响范围重构范围，对于微服务做的比较复杂的项目来说，每个人可能只是关注自己服务的上下游，对于上游的上游和下游的下游完全不清楚，导致公司也没有人可以说的清楚架构的全貌。这个时候我们有全链路跟踪系统的话，可以对通过分析过去的调用来绘制出一张依赖关系的架构图。这个图如果对 QPS 做一些热点的话，还可以帮助我们做一些运维层面的容量规划。
- 高级分析建议。比如在全链路压测后定位分析瓶颈所在。定时分析所有组件的执行性能，得出性能衰退的趋势，提早进行问题预警。分析 JVM 的线程和 GC 情况，辅助定位 High CPU 和 Memory Leak 的问题。退一万步说，即使没有这样的自动化的高级分析，有了调用跟踪的图和组件依赖关系图，至少在出问题的时候我们人能分析出来咋回事。
- Dashboard。非必须，只要数据收集足够全面，如之前文章所示，我们可以用 Grafana 来进行各种个性化的图表配置。

数据访问中间件

开源的实现有 C 实现的 https://github.com/Qihoo360/Atlas 以及 Go 实现的 https://github.com/flike/kingshard 等。数据访问中间件是独立部署的数据库的透明代理，本身需要是以集群方式支持高可用，背后还需要对接多套数据库作为一个集群，一般而言会提供如下的功能：

- 最常用的功能就是读写分离。也包括负载均衡和故障转移的功能，自动在多个从库做负载均衡，通过可用性探测，在主库出现故障的时候配合数据库的高可用和复制做主库的切换。
- 随着数据量的增多需要分片功能。分片也就是 Sharding，把数据按照一定的维度均匀分散到不同的表，然后把表分布在多个物理数据库中，实现压力的分散。这里写入的 Sharding 一般而言没有太多的差异，但是读取方面因为涉及到归并汇总的过程，如果要实现复杂功能的话还是比较麻烦的。由于分片的维度往往可能有多个，这方面可以采用多写多个维度的底层表来实现也可以采用维度索引表方式来实现。
- 其它一些运维方面的功能。比如客户端权限控制，黑白名单，限流，超时熔断，和调用链搭配起来的调用跟踪，全量操作的审计搜索，数据迁移辅助等等。

更高级的话可以实现 SQL 的优化功能。随时进行 SQL 的 Profiler，然后达到一定阈值后提供索引优化建议。类似 https://github.com/Meituan-Dianping/SQLAdvisor。

- 其它。极少的代理实现了分布式事务的功能（XA）。还可以实现代理层面的分布式悲观锁的功能。其实细想一下，SQL 因为并不是直接扔到数据库执行，这里的可能性就太多了，想干啥都可以。

实现上一般需要做下面几件事情：

- 有一个高性能的网络模型，一般基于高性能的网络框架实现，毕竟 Proxy 的网络方面的性能不能成为瓶颈。
- 有一个 MySQL 协议的解析器，开源实现很多，拿过来直接用即可。
- 有一个 SQL 语法的解析器，Sharding 以及读写分离免不了需要解析 SQL，一般流程为 SQL 解析、查询优化、SQL 路由、SQL 重写，在把 SQL 提交到多台数据库执行后进行结果归并。
- Proxy 本身最好是无状态节点，以集群方式实现高可用。

这些功能除了 Proxy 方式的实现还有和数据访问标准结合起来的实现，比如改写 JDBC 的框架方式实现，两种实现方式各有优缺点。框架方式的实现不局限于数据库类型，性能略高，Proxy 方式的实现支持任意的语言更透明，功能也可以做的更强大一些。最近还出现了边车 Sidecard 方式实现的理念，类似于 ServiceMesh 的概念，网上有一些资料，但是这种方式到目前为止还没看到成熟的实现。

分布式缓存中间件

类似于数据库的 Proxy，这里是以缓存服务作为后端，提供一些集群化的功能。比如以 Redis 为后端的开源的实现有 https://github.com/CodisLabs/codis 以及饿了么的 https://github.com/eleme/corvus 等等。其实不采用 Proxy 方式做，开发一个缓存客户端在框架层面做也是完全可以的，但是之前也说了这两种方式各有优劣。代理方式的话更透明，如果有 Java、Python、Go 都需要链接 Redis，我们无需开发多套客户端了。一般实现下面的功能：

- 分布式。这是最基本的，通过各种算法把 Key 分散到各个节点，提供一定的容量规划和容量报警功能。
- 高可用。配合 Redis 的一些高可用方案实现一定程度的高可用。
- 运维方面的功能。比如客户端权限控制，黑白名单，限流，超时熔断，全量操作的审计搜索，数据迁移辅助等等。
- 跟踪和问题分析。配合全链路监控实现一体化的缓存访问跟踪。以及更智能的分析使用的情况，结合缓存的命中率，Value 的大小，压力平衡性提供一些优化建议和报警，尽早发现问题，缓存的崩盘往往是有前兆的。
- 完善的管理后台，可以一览集群的用量、性能，以及做容量规划和迁移方案。

如果 Redis 集群特别大的话的确是有一套的自己的 Proxy 体系会更方便，小型项目一般用不到。

任务（Job）管理

之前有提高过，Job 是我认为的互联网架构体系中三马车的三分之一，扮演了重要的角色。开源实现有 http://elasticjob.io/。Job 的管理的实现有两种方式，一种是类似于框架的方式，也就是 Job 的进程是一直启动着的，由框架在合适的时候调用方法去执行。一种是类似于外部服务的方式，也就是 Job 的进程是按需要在合适的机器启动的。在本文一开始的图中，我画了一个任务调度的中间件，对于后一种方式的实现，我们需要有一套中间件或独立的服务来复杂 Job 进程的拉起。整个过程如下：

- 找一些机器加入集群作为我们的底层服务器资源。
- Job 编译后打包部署到统一的地方。Job 可以是各个语言实现的，这没有关系。可以是裸程序，也可以使用 Docker 来实现。
- 在允许 Job 前我们需要对资源进行分配，估算一下 Job 大概需要怎么样的资源，然后根据执行频次统一计算得出一个合适的资源分配。
- 由中间件根据每一个 Job 的时间配置在合适的时候把进程（或 Docker）拉起执行，执行前根据当前的情况计算分配一个合适的机器，完成后释放资源，下一次执行不一定在同一台机器执行。

这样的中间件是更底层的一套服务，一般而言任务框架会提供如下的功能：

- 分布式。Job 不会受限于单机，可以由集群来提供运行支持，可以随着压力的上升进行集群扩容，任何一台机器的宕机不会成为问题。如果我们采用中间件方式的话，这个功能由底层的中间件来支持了。
- API 层面提供丰富的 Job 执行方式。比如任务式的 Job，拉数据和处理分开的 Job。拉数据和处理分开的话，我们可以对数据的处理进行分片执行，实现类似 Map-Reduce 的效果。
- 执行依赖。我们可以配置 Job 的依赖关系实现自动化的 Job 执行流程分析。业务只管实现拆散的业务 Job，Job 的编排通过规则由框架分析出来。
- 整合到全链路监控体系的监控跟踪。
- 丰富的管理后台，提供统一的执行时间、数据取量配置，提供 Job 执行状态和依赖分析一览，查看执行历史，运行、暂停、停止 Job 等等管理功能。

发布管理

发布管理其实和开发没有太大的关联，但是我觉得这也是整个体系闭环中的一个环节。发布管理可以使用 Jenkins 等开源实现，在后期可能还是需要有自己的发布系统。可以基于 Jenkins 再包一层，也可以如最开始的图所示，直接基于通用的任务调度中间件实现底层的部署。一般而言，发布管理有下面的功能：

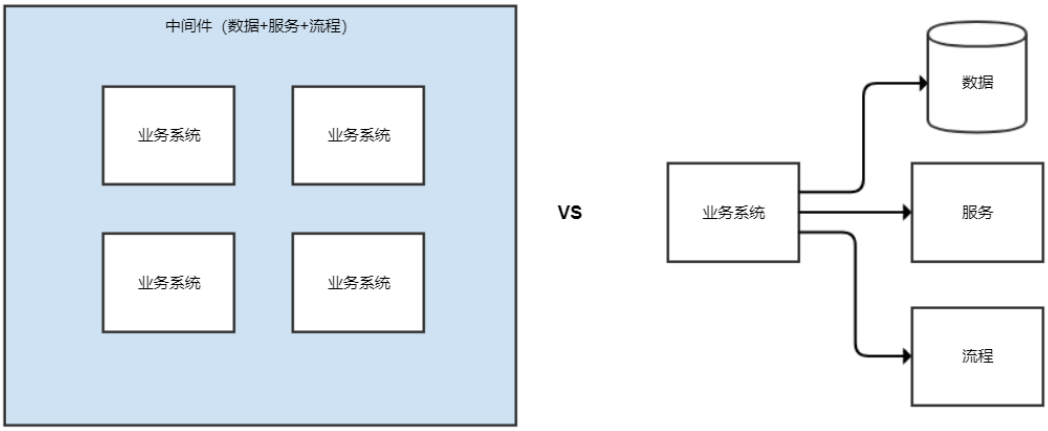
- 丰富的任务类型和插件，支持各种语言程序的构建和发布。有最基本的发布、回滚、重启、停止功能。
- 支持项目的依赖关系设置，实现自动化的依赖路径上的程序自动发布。
- 一些运维层面的控制。比如和 CMDB 结合做权限控制，做发布窗口控制。
- 用于集群的发布流程。比如可以一览集群的分组，设置自动的灰度发布方案。
- 适合自己公司的发布流程。比如在流程控制上，我们是 Dev 环境到 QA 到 Stage 到 Live。其中，QA 环境经过 QA 的确认后可以进入 Stage 环境，经过开发主管的确认后可以到 Stage 环境，经过产品经理的确认后可以进入 Live 环境进行发布。在发布系统上我们可以结合 OA 做好这个流程的控制。
- 在构建的时候，集成单元测试，集成编码规范检查等等，在后台可以方便的看到每一次发布的代码变更，测试执行情况以及代码规范违例。

Jenkins 等系统在对于 1 和 2 做的比较好，对于和公司层面其它系统的结合无能力为，往往处于这个原因我们需要在 Jenkins 之上包装出来自己的发布系统。

总结一下，之所以标题说不断耕耘的基础中间件，是指中间件也好框架也好，往往也需要一个小团队来独立维护，而且功能是不断迭代增加，这套体系如果结合的好，就不仅仅是实现功能这个最基本的标准了，而是：

- 运维自动化 API 化和 AI 化的很重要的构成。把控是因为我们掌握了数据流，数据都是从我们的中间件穿越过去到达底层的服务、数据库、缓存，有了把控就有了自动化的可能，有了智能监控一体化报警的可能。
- 也因为数据流的经过，通过对数据进行分析，我们可以给到开发很多建议，我们可以在这上面做很多标准。这些事情都可以由框架架构团队默默去做，不需要业务研发的配合。

- 因为底层数据源的屏蔽，加上服务框架一起，我们实现的是业务系统被框架包围而不是业务系统在使用框架和中间件这么一个形态，那么对于公司层面的一些大型架构改造，比如多活架构，我们可以实现业务系统的改造最小。数据+服务+流程都已经被中间件所包围和感知，业务系统只是在实现业务功能而已，我们可以在业务系统无感知的情况下对数据做动态路由，对服务做动态调用，对流程做动态控制。如下图，是不是有点 Mesh 的意思？



本文很多地方基于思考和 YY，开源组件要实现这个理念需要有大量的修改和整合，很多大公司内部都一定程度做了这些事情，但是也因为框架的各种粘连依赖无法彻底开源，这块工作要做好需要大量的时间精力，真的需要不断耕耘和沉淀才能发展出适合自己公司技术栈的各种中间件和管理系统体系。