23 | 业务及系统架构对发布的影响

2018-08-25 王潇俊

23 | 业务及系统架构对发布的影响
朗读人：王潇俊　10′47″ | 4.95M

在分享《发布系统一定要注意用户体验》和《发布系统的核心架构和功能设计》这两大主题时，我分别从用户体验和架构设计两个方面，和你分享了携程灰度发布系统的一些经验和实践。但是，要做出一个出色的发布系统，仅仅考虑这两方面还不够。

因为发布系统最终要服务的对象是业务应用，而业务应用又和业务、企业的系统架构有紧密的联系，所以要做好一套发布系统，我们还要考虑其要服务的业务及系统架构的需要，并且要善于利用系统架构的优势为发布系统服务。

那么接下来，我们就一起来看看，业务、企业整体的系统架构会给发布系统带来什么影响，发布系统又可以借用它们的哪些架构能力。

单机单应用还是单机多应用？

众所周知，.NET 应用采用的基本都是 Windows + IIS 的部署模式，这是一种典型的单机、单 Web 容器、多应用的部署模式。

在这种模式下，单机多应用的问题主要体现在两个方面：

- 一方面，应用划分的颗粒度可以做到非常细小，往往一个单机上可以部署 20~30 个应用，甚至更多，而且应用与应用间的隔离性较差；
- 另一方面，由于 IIS 的设计问题，不同虚拟目录之间可能存在共用应用程序池的情况，即多个应用运行在同一个进程下，导致任何一个应用的发布都可能对其他的关联应用造成影响。

所以，对发布系统而言，.NET 应用的这种架构简直就是噩梦：发布系统需要重新定义发布单元的含义，并维护每个虚拟目录和对应的发布单元与应用之间的关系。

在携程，我们为了解决这个问题采用的方案是，去除根目录的被继承作用，默认每个虚拟目录就是一个应用，并且每个虚拟目录的应用程序池独立。而每个虚拟目录以应用的名称命名，保证单机上不会发生冲突。

这样，应用与应用之间就解耦了，发布系统的设计也会变得简单很多。

除了上面这种.NET 的单机多应用情况无法改变外，其他所有 Linux 下的应用都可以做到单机单应用。其实，这也正是虚拟化思想最初的设计理念。为什么呢？因为与单机多应用相比，单机单应用更简单直接，更易于理解和维护。

比如，单机单应用不需要考虑分配服务端口的问题，所有的 Web 应用都可以使用同一个统一端口（比如，8080 端口）对外服务。但是，单机多应用的情况下，就要考虑端口分配的问题，这算不算是徒增烦恼呢？

另外，单机单应用在故障排除、配置管理等方面同样具有很多优势。一言以蔽之，简单的才是最好的。

当然，简单直接，也正是发布系统所希望看到的情况。

增量发布还是全量发布？

增量发布还是全量发布，其实是个挺有意思的问题。

在过去网络带宽是瓶颈的年代里，或者面对体量巨大的单体应用时，增量发布可以节省很多计算资源，确实是一个很好的解决方案。甚至现在的移动客户端发布，也还会选择增量发布的技术来快速发布静态资源。

但是，互联网应用的场景下，更多的发布需求来自于发布频率非常高的后台服务。在这样的情况下，增量发布反而会造成不必要的麻烦。

比如，增量发布对回滚非常不友好，你很难确定增量发布前的具体内容是什么。如果你真的要确定这些具体内容的话，就要做全版本的差异记录，获取每个版本和其他版本间的差异，这是一个

巨大的笛卡尔积，需要耗费大量的计算资源，简直就是得不偿失。很显然，这是一个不可接受的方案。

反之，全量发布就简单多了，每个代码包只针对一个版本，清晰明了，回滚也非常简单。所以，**我的建议是，全量发布是互联网应用发布的最好方式。**

如何控制服务的 Markup 和 Markdown？

首先，你需要明确一件事儿，除了发布系统外，还有其他角色会对服务进行 Markup 和 Markdown 操作。比如，运维人员在进行机器检修时，人为的 Markdown 操作。因此，我们需要从发布系统上能够清晰地知晓服务的当前状态，和最后进行的操作。

另外，这里还引入了一个全新的问题：当一个服务被执行 Markdown 操作后，什么系统还能继续处理这个服务，而什么系统则不能继续处理这个服务？

比如，发布系统如果发现服务最后进行的操作是 Markdown，那么还能不能继续发布呢？如果发布，那发布之后需不需要执行 Markup 操作呢？有些情况下，用户希望利用发布来修复服务，因此需要在发布之后执行 Markup；而有些情况下，用户发布后不能执行 Markup，很可能运维人员正在维护网络。

为了解决这个问题，**携程在设计系统时，用不同的标志位来标识发布系统、运维操作、健康检测，以及服务负责人对服务的 Markup 和 Markdown 状态。4 个标志位的任何一个为 Markdown 都代表暂停服务，只有所有标志位都是 Markup 时，服务中心才会向外暴露这个服务实例。**

这样做的好处是，将 4 种角色对服务的操作完全解耦，他们只需要关心自己的业务逻辑，既不会发生冲突，也不会影响事务完整性，更无需采用其他复杂的锁和 Token 机制进行排他操作。

检查、预热和点火机制

我在分享从用户体验和核心架构的角度设计发布系统时，提到过发布过程中必然会有 Verify 的过程。

如果这个过程依赖手工操作的话，一来难以保证发现问题的速度，二来也很难保证落实力度。但如果这个过程能够做到自动化的话，则可以大幅减少因发布而引发的生产故障，同时还可以保证一些服务启动依赖检测。

在携程，我们借助于 VI（Validate Internal）框架中间件，实现了 Verify 过程的自动化，我们把这个过程形象地叫作“点火”。所有使用这个中间件的应用启动后，发布系统执行的第一个操作就是这个 VI 接口所提供的检查方法，当然用户完全可以根据业务自定义应用的检查方法。这也就保证了发布过程中一定会执行到 Verify 过程，而不会因为各种原因而被遗漏。

Verify 是一个异步过程，可能耗时较长，但是程序员们很快就发现，这个 VI 组件不但可以做检查，还可以在检查时进行一些预热、预加载这样的任务。

携程通过这样一个中间件组件，很高效地解决了发布过程中的两个问题：

1. 如何对各个应用做个性化的自动化检查；
2. 如何在发布过程中解决应用预加载这类的需求。

如何保证堡垒流量？

我在《发布是持续交付的最后一公里》这篇文章中介绍金丝雀发布时，说到了携程选择的是综合使用滚动发布和金丝雀发布的方案，使用堡垒机的方式来预发和测试新的版本。

但是，采用这个方案，我们需要考虑分布式服务架构带来的影响，即如何保证堡垒机的流量一定会分发到对应下游服务的堡垒机上。就好比，发布一个包含 Web 和 Service 两个应用的新功能，我需要保证 Web 堡垒的流量只发送给 Service 的堡垒，否则就会出问题。

我们解决这个问题的思路是，软负载系统通过发布系统获得堡垒机的 IP，在堡垒机发出的请求的 header 中附加堡垒标识，这样软负载在判断出有堡垒标识时，则只会将请求发向下游的堡垒机。当然，是否加注这个标识，完全由发布系统在发布堡垒时控制。

这样，我们就解决堡垒流量控制的问题。

总结

因为发布系统最终服务的对象是业务应用，所以发布系统的设计除了考虑用户体验、核心架构外，还要注意业务、系统架构对发布系统的影响，并且要合理利用业务、系统架构的能力，去完善发布系统的设计。

业务、系统架构对发布系统的影响，主要体现在是选择单机单应用还是单机多应用、选择增量发布还是全量发布这两大方面。在这里，我的建议是简单的才是最好的，即采用单机单应用的部署架构；对于后台应用，以及发布非常频繁的应用来说，全量发布更直接，更容易达成版本控制，并做到快速回滚。

除此之外，我们还要利用系统架构使发布系统具有更优的发布能力，这主要包括三个方面：

1. 利用软负载或者服务通讯中间件的多个状态位，简单直接地解决多角色对服务 Markup 和 Markdown 的冲突问题；
2. 利用中间件的能力，使得 Verify 过程不会被遗忘，同时还可以完成自动化检查和预热，达到高效可控的目的；

3. 通过软负载和通讯中间件，解决堡垒机与堡垒机间的流量分发问题，保证堡垒机的流量只在上下游服务的堡垒机中流转。

因此，有效运用系统架构的能力，会为你带来意想不到的收益。

思考题

在你的实际工作中，发布系统有没有受到架构设计的影响？是积极影响还是消极影响呢？消极影响的话，你有想过如何改变吗？

感谢收听，欢迎你给我留言。

通过留言可与作者互动