确保部署期间安全回滚

Sandeep Pokkunuri

Amazon 构建解决方案的原则之一就是避开单行道。也就是说，我们会舍弃难以逆转或扩展的选择。从设计产品、功能、API 和后端系统到部署，我们将这一原则应用到软件开发的所有步骤。在本文中，我将说明如何将这一原则应用于软件部署。

部署将软件环境从一种状态（版本）转换到另一种状态（版本）。软件在这两种状态下均可能运行良好。但是，软件在前向过渡（升级或前滚）或后向过渡（降级或回滚）期间或之后可能无法正常运行。如果软件无法正常运行，则可能会导致服务中断，对客户而言不可靠。在本文中，我假定软件的两个版本均按预期运行。我的关注重点是如何确保在部署过程中前滚或回滚不会导致错误。

在发布新版本的软件之前，我们会在 beta 或 gamma 测试环境中对功能、并发性、性能、规模和下游故障处理等多个维度进行测试。这样的测试有助于我们发现新版本中的所有问题并加以修复。但是，这种做法有时还不足以确保成功部署。在生产环境中，我们可能会遇到意外情况，或者软件行为不佳。在 Amazon，我们希望避免陷入回滚部署会造成客户方错误的情况。为了避免出现这种情况，我们在每次部署之前都会为回滚做好充分准备。可以回滚而不会出错，也不会破坏先前版本中可用功能的软件版本称为后向兼容版本。我们会对软件进行规划和验证，确保每个修订版都具备后向兼容性。

在详细介绍 Amazon 如何进行软件更新之前，我们讨论一下独立和分布式软件部署之间的一些区别。

独立与分布式软件部署

对于在一个设备上作为一个进程运行的独立软件，部署是原子性质的。软件永远不会有两个版本同时运行。如果独立软件保持稳定状态，则新版本必须读取（即反序列化）由旧版本写入（即序列化）的数据，反之亦然。满足此条件可使部署安全地前滚和回滚。

在分布式系统中，部署要更加复杂。部署是通过滚动更新完成的，因此可用性不会受到影响。新版本将立即部署到一部分主机，以便其他主机可以继续处理请求。通常，这些主机通过远程过程调用 (RPC) 或共享的持久状态（例如，元数据或检查点）相互通信。这种通信或共享状态可能会带来其他挑战。写入方与读取方运行的软件版本可能不同。因此，它们对数据结果的解释方式可能有所不同。读取方甚至可能完全无法读取数据，从而导致中断。

协议变更问题

我们发现无法回滚的最常见原因是协议的更改。例如，假设一项代码更改在将数据持久保存到磁盘时开始压缩数据。在新版本写入一些压缩数据后，将无法回滚。旧版本不知道从磁盘读取数据后必须解压缩数据。如果数据存储在 Blob 或文档存储中，那么即使部署正在进行，其他服务器也将无法读取数据。如果此数据在两个进程或服务器之间传递，则接收方将无法读取它。

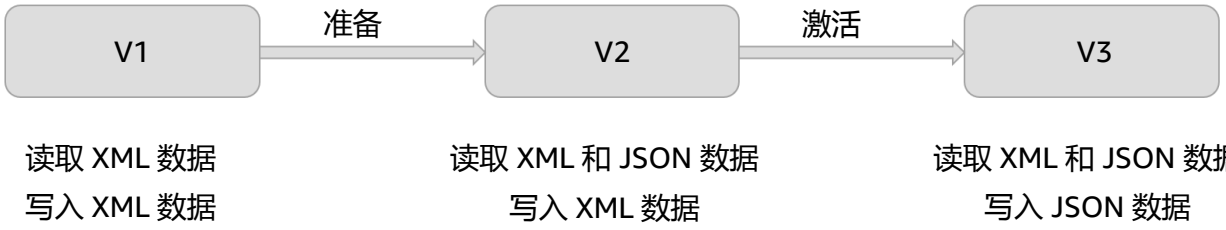
有时，协议更改可能非常微妙。例如，考虑两台通过连接异步通信的服务器。为了让彼此知道自己还在正常运行，它们商定每隔五秒钟相互发送一次检测信号。如果一台服务器在规定时间内没有看到检测信号，则会认为另一台服务器已关闭，就会关闭连接。

现在，考虑将检测信号周期增加到 10 秒的部署。代码提交似乎没怎么更改，只更改了数字。但是，现在前滚和回滚都不安全。在部署期间，运行新版本的服务器每 10 秒发送一次检测信号。因此，运行旧版本的服务器超过五秒未看到检测信号，因此会终止与运行新版本的服务器的连接。在大型队组中，这种情况可能发生在多个连接上，从而导致可用性下降。

很难通过阅读代码或设计文档分析这些细微的变化。因此，我们显式验证每项部署对于前滚和回滚是否安全。

两阶段部署技术

我们可以确保安全回滚的一种方法是使用通常称为两阶段部署的技术。考虑以下假设场景，该场景具有管理 Amazon Simple Storage Service (Amazon S3) 上的数据（执行数据写入和读取操作）的服务。该服务在跨越多个可用区的服务器队组中运行，以实现扩展和可用性。
当前，该服务使用 XML 格式来保留数据。如下面的版本 V1 图所示，所有服务器都写入和读取 XML 数据。出于业务原因，我们希望以 JSON 格式保留数据。如果我们在一项部署中进行此更改，则接受此更改的服务器将以 JSON 格式捕获数据。但是，其他服务器尚不知道如何读取 JSON 数据。这种情况会导致错误。因此，我们将这种更改分为两个部分，并执行两阶段部署。



如上图所示，我们将第一阶段称为“准备”。在此阶段中，我们准备所有服务器以读取 JSON 数据（此外还能读取 XML 数据），但是它们将通过部署版本 V2 继续写入 XML 数据。从操作的角度来看，此更改并未改变任何内容。所有服务器仍可以读取 XML 数据，所有数据仍以 XML 格式写入。如果我们决定回滚此更改，则服务器将恢复为无法读取 JSON 数据的状态。这并不是问题，因为尚未以 JSON 格式写入任何数据。

如上图所示，我们将第二阶段称为“激活”。在此阶段，我们通过部署版本 V3 激活服务器以使用 JSON 格式写入数据。当每个服务器接收到此更改时，它将开始以 JSON 格式写入数据。尚未收到此更改的服务器仍可以读取 JSON 数据，因为它们已经在第一阶段中完成准备工作。如果我们决定回滚此更改，则暂

时处于“激活”阶段的服务器写入的所有数据均采用 JSON 格式。未处于“激活”阶段的服务器写入的数据为 XML 格式。这是一种理想的情况，因为如 V2 所示，服务器在回滚后仍可以读取 XML 和 JSON 数据。
尽管前面的图显示了从 XML 到 JSON 的序列化格式更改，但是通用技术适用于前面的“协议更改”部分描述的所有情况。例如，回想一下以前的场景，在该场景中，服务器之间的检测信号周期必须从 5 秒增加到 10 秒。在“准备”阶段，尽管所有服务器继续每五秒发送一次检测信号，但是我们可以将所有服务器的预期检测信号周期放宽到 10 秒。在激活阶段，我们将频率更改为 10 秒一次。

两阶段部署的预防措施

现在，我将说明我们遵循两阶段部署技术时要采取的预防措施。尽管这里我参考了前面部分中说明的示例场景，但是这些预防措施适用于大多数两阶段部署。

如果最少数量的主机接收更改并报告自己运行状况正常，则许多部署工具都会让用户认为部署已经成功。例如，AWS CodeDeploy 具有一个名为 minimumHealthyHosts 的部署配置。

示例两阶段部署中的一个关键假设是，在第一阶段结束时，所有服务器都已升级为可读取 XML 和 JSON 数据。如果在第一阶段中一个或多个服务器升级失败，则它们将在第二阶段期间及之后无法读取数据。因此，我们要显式验证所有服务器在“准备”阶段中是否都已接受更改。

在研究 Amazon DynamoDB 时，我们决定更改跨越多项微服务的大量服务器之间的通信协议。我协调了所有微服务之间的部署，以便所有服务器先到达“准备”阶段，然后进入“激活”阶段。为防出现意外，我显式验证了在每个阶段结束时，每个服务器上的部署是否均已成功。

虽然两个阶段中的每个阶段都可以回滚，但是我们不能同时回滚这两项更改。在较早的示例中，在“激活”阶段结束时，服务器以 JSON 格式写入数据。在准备和激活更改之前使用的软件版本不知道如何读取 JSON 数据。因此，作为一项预防措施，我们在“准备”和“激活”阶段之间留出了相当长的时间。我们将此时间称为烘烤期，其持续时间通常为几天。我们通过等待来确保不必回滚到较早的版本。

在“激活”阶段之后，我们无法安全取消软件读取 XML 的功能。取消这项功能并不安全，因为在“准备”阶段之前写入的所有数据都是 XML 格式的。确保每个对象都已用 JSON 格式重写后，我们才能取消其读取 XML 数据的功能。我们称此过程为回填。它可能需要在服务写入和读取数据时可以并发运行的其他工具。

序列化的最佳实践

大多数软件都涉及数据序列化 – 无论是实现持久性存储还是通过网络传输的目的。随着软件的发展演变，序列化逻辑通常会发生变化。更改的范围可以从添加新字段到完全更改格式。多年来，我们已经得出了一些有关序列化的最佳实践：

- 我们通常避免开发自定义序列化格式。

  自定义序列化的初始逻辑看似微不足道，甚至可以提供更好的性能。但是，格式的后续迭代会造成一些挑战，这些挑战已经由多种完善的框架（如 JSON、协议缓冲区、Cap'n Proto 和 FlatBuffers）解决。如果适当使用，这些框架能提供安全功能，例如转义、后向兼容和属性存在跟踪（也就是说，是显式设置字段还是隐式分配默认值）。

- 每次更改后，我们都会为序列化程序明确分配一个不同的版本。

  这项操作独立于源代码或内部版本控制。我们还将序列化程序版本与序列化数据一起存储，或以元数据的形式存储。较旧的序列化程序版本在新软件中继续正常运行。我们发现，通常可以针对所写入或读取的数据版本发布指标。如果有错误，它将为操作人员提供可见性和故障排除信息。所有这一切也适用于 RPC 和 API 版本。
- 我们会避免序列化我们无法控制的数据结构。

  例如，我们可以使用反射来序列化 Java 的集合对象。但是，当我们尝试升级 JDK 时，此类基础实施可能会更改，从而导致反序列化失败。此风险也适用于团队之间共享的库中的类。
- 通常，我们将序列化程序设计为允许存在未知属性。

  在可行的情况下，我们的序列化程序在回写数据时保留未知属性。通过这种调整，即使运行新版本软件的服务器在序列化时在数据中包括新属性，运行旧版本的服务器在更新相同数据时也不会清除属性。这样就不需要将部署分成两个阶段。

像我们的许多最佳实践一样，我们在分享这些最佳实践时要格外小心，因为我们的准则并不适用于所有应用程序和场景。

验证更改是否能够安全回滚

通常，我们通过所谓的升级-降级测试来显式验证软件更改是否可以安全地前滚和回滚。在此过程中，我们要建立一个代表生产环境的测试环境。多年来，我们已经确定了设置测试环境时应避免的几种模式。

我曾看到过这样的情况：尽管更改通过了测试环境中的所有测试，但是在生产环境中部署更改仍然会导致错误。在一种情况下，测试环境中的每项服务仅有一台服务器。因此，所有部署都是原子性质的，这排除了并发运行不同版本的软件的可能性。现在，即使测试环境的流量不如生产环境的流量大，我们也会像使用生产环境那样使用每项服务背后不同可用区中的多台服务器。Amazon 崇尚节俭，不过为了保证质量，我们绝不会为了节俭而束手束脚。

在另一种情况下，测试环境具有多台服务器。但是，部署会一次性针对所有服务器执行，以加快测试速度。这种方法也阻止了软件的新旧版本同时运行。此时不会检测到前滚问题。现在，我们在所有测试和生产环境中使用相同的部署配置。

对于涉及微服务间协调的更改，我们在测试和生产环境中采用相同的跨微服务部署顺序。但是，前滚和回滚顺序可能会有所不同。例如，我们通常会在序列化上下文中遵循特定的顺序。即，读取方在前滚时先于写入方，而写入方在后滚时先于读取方。在测试和生产环境中，通常要遵循适当的顺序。

当测试环境的设置类似于生产环境时，我们将尽可能严密地模拟生产流量。例如，我们快速创建并读取多条记录（或消息）。所有 API 连续运行。然后，我们将环境分为三个阶段，每个阶段持续一段合理的时间，以识别潜在的错误。持续时间足够长，以使所有 API、后端工作流和批处理作业至少运行一次。首先，将更改部署到队组中大约一半的设备中，以确保软件版本共存。第二，完成部署。第三，启动回滚部署，并遵循相同的步骤，直到所有服务器都运行旧版软件。如果在这些阶段中没有错误或意外行为，则认为测试成功。

结论

确保可以回滚部署而不会给客户造成任何中断，这对于保证服务的可靠性至关重要。显式测试回滚安全性消除了对人工分析的依赖，因为人工分析可能容易出错。当发现更改不适于回滚时，通常可以将其分为两项更改，每项更改都可以安全地前滚和回滚。