22 | 重构：系统升级，如何实现不停服的数据迁移和用户切量？

专栏的前 21 讲，从读、写以及扣减的角度介绍了三种特点各异的微服务的构建技巧，最后从微服务的共性问题出发，介绍了这些共性问题的应对技巧。

在实际工作中，你就可以参考本专栏介绍的技巧构建新的微服务，架构一个具备“三高”能力的后台服务。同时，也可以利用所学的技巧对已有的微服务进行系统升级重构，从而解决历史遗留问题。

升级重构是后台架构演化增强的一个利器，本讲将为你详细讲解如何落地对用户无感知、低 Bug 的升级重构方案。

重构常见的形式

升级重构有两种常见的形式，一种是纯代码式的升级，另外一种是包含存储和代码的升级。

纯代码的重构升级是指只针对代码中存在的一些历史遗留问题进行修复，比如代码中的慢 SQL、错误的日志打印方式、代码中未显式开启事务等问题。

注：在本讲，将存在问题的历史版本称为V1 版；修复问题后的升级重构版本称为 V2 版。纯代码的重构升级架构如下图1所示：

图 1：纯代码的升级重构对比图

包含存储和代码的重构升级是指在上述纯代码之外，将原有架构里的存储也一起升级。存储升级有两种形式。

第一种是将存储类型进行升级，比如将数据库升级为缓存，将原有的读接口从数据库切换至缓存。做此类存储类型升级的目的是提升微服务的性能，我们在模块二中曾介绍，同样的硬件配置下，缓存比数据库至少快一个量级。不同类型的存储升级架构如下图 2 所示：

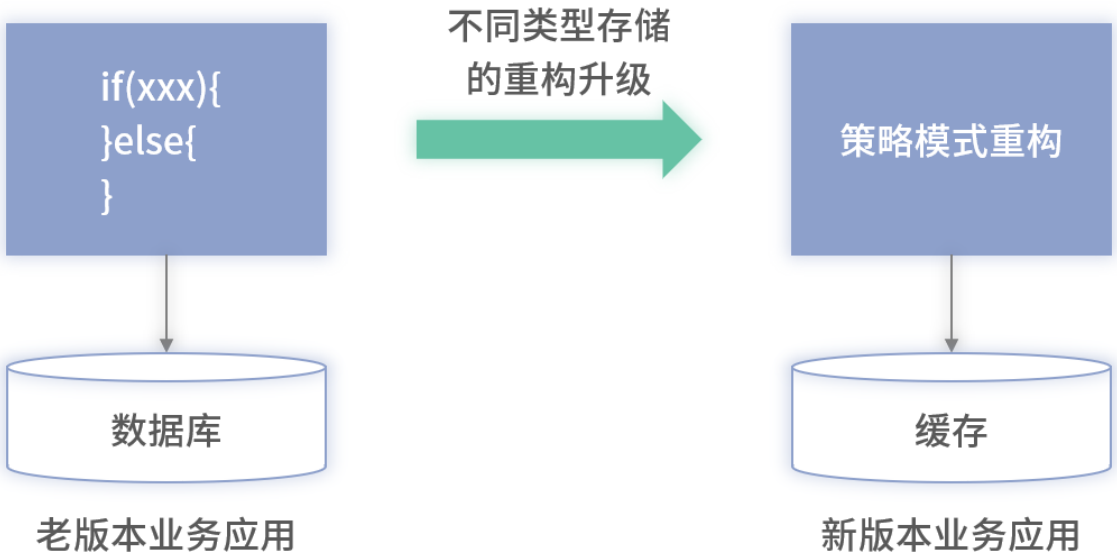

图 2：不同类型的存储升级架构对比图

可以看到升级后，原有的代码从 V1 升级到 V2。同时，存储从读写都使用数据库，升级为写操作使用数据库，读操作使用缓存的架构。

第二种是将一个表结构的存储升级为同类型存储的另外一个更加合理的表结构。此类升级常见于系统构建时，为了快速满足业务需求，在时间紧张的情况下，简单快速地设计了不是特别合理的表结构。比如原有的表结构采用了一张宽表存储所有的数据，包含一对多的数据都进行冗余存储。升级重构时，需要采用更加合理的表结构存储数据，以便未来能够快速响应业务的发展。它的重构升级架构如下图 3 所示，升级后，原有微服务的读写都将切换至新的表结构的存储里。

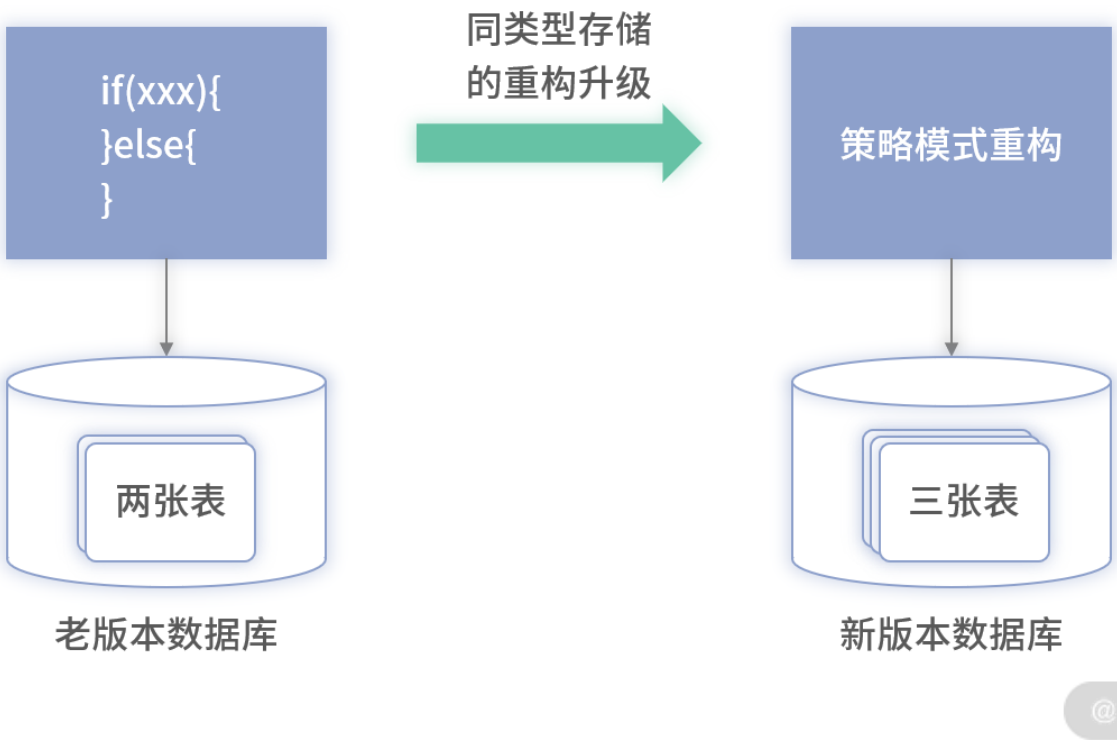

图 3：同类型不同表结构的存储升级架构对比图

纯代码重构的切换

纯代码重构的切换比较简单，当上述 V2 版本通过测试环境和预发布环境的测试后，就可以直接在线上部署，替换原有的V1 版本。当部署的 V2 版本出现问题后，直接进行回滚即可，这是最简单、粗暴的切换方式。但同时也存在隐患，采用此种方式部署的V2如果出现问题，会影响所有的用户，影响面较大。

为了降低影响，可以采用灰度的方式。即用 V2 版本的代码替换一台或者一定比例 V1 版本的机器，比如线上有 100 台部署 V1 版本代码的机器，当 V2 版本测试完成准备上线时，可以先发布 10 台 V2 版本的代码。这样，假设 V2 版本的代码存在 Bug，也只会影响访问这 10 台部署了 V2 版本代码的用户，即 10%的线上流量，这样就缩小了影响面。假设发布了 10 台 V2 版本的代码后，没有发现任何 Bug，此时则可以继续发布，逐步进行替换。

通过此种灰度的方式，既可以做到纯代码的升级重构切换，又可以缩小因此可能带来的线上问题的影响范围。

含存储重构的切换

与上述纯代码的切换相比，含存储的重构切换有一个重要步骤便是数据迁移。不管是上述不同类型的存储还是同类型不同表结构的存储重构，都需要将原有存储中的数据全部迁移至新的存储中，才能够称为完成切换。

对于含存储重构的切换，最简单的方法便是停服，之后在无任何数据写入的情况下进行数据迁移，迁移之后再进行数据对比，对比无误之后，用重构的新版本连接新的存储对外提供服务即可。

这种方式适合于以下 2 种场景：

1. 业务有间断期或者有低峰期的场景。比如企业内网系统，下班或者周末期间几乎没有人使用；
2. 金融资产类业务，这些业务对于正确性要求极高，因为用户对资产极度敏感，如果资产出现错误，用户是无法容忍的。为了资产安全无误，有时候需要用户容忍停服的重构升级。

但对于用户量巨大，且大部分业务场景都需要提供 7*24 服务的互联网业务来说，停服切换方式，用户是不能接受。因此，就需要设计一套既不需要停服，又可以完成用户无感知的切换方案。

切换架构

为了实现不停服的重构升级，整体的新版本上线、数据迁移以及用户切量的架构如下 图 4 所示：

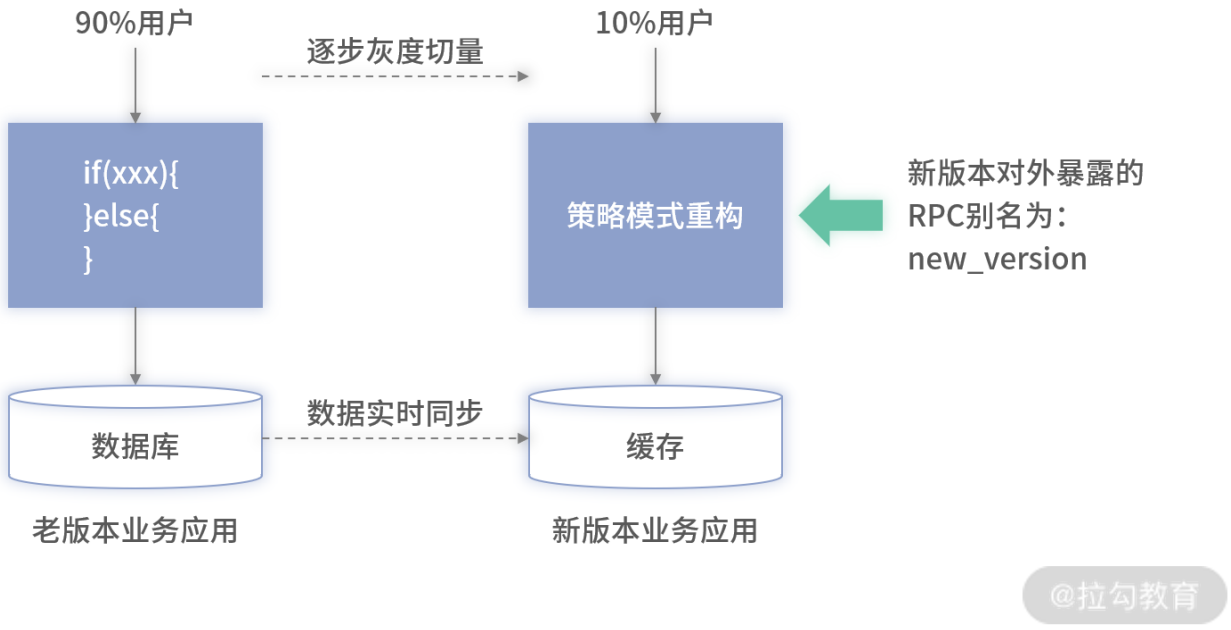

图 4：不停服的切换架构

上述的架构中，左边部分是老版本未重构的服务及对应的老数据存储（后续称为老存储），图中右边部分部署的是升级重构后的新版本的服务和对应的新版本存储（后续称为新存储）。这个存储可以是缓存或者是表结构不同的数据库。

在图的下方，则是数据同步模块。它主要的作用是实时进行数据同步，将老存储里的历史数据、新增的写入以及更新的数据实时地同步至新存储里。实**时数据同步是实现存储升级重构不停服切换的基础**。

在完成数据同步之后，便可以进行用户的灰度切量了，将用户逐步切换至升级重构的新版本上。见图 4 中最上面的部分。

下面将对图 4 中涉及的各个部分进行详细讲解。

数据同步

当升级重构后的新版本开发及测试完成后，便可以将新版本代码和存储进行线上部署了。新版本部署时，可以将新版本服务对外提供的接口的别名变更为一个新的名称，如为：new_version，具体见上图 4。因为修改了别名，即使新版本的服务上线部署并直接对外了，也不会引入老版本的流量。

通过上述方式可以实现新老版本的隔离，进而完成新版本服务的线上部署。新版本线上部署及隔离后，便可以进行数据同步了。

数据同步分为历史数据的全量同步和新增数据的实时同步。在上述“含存储重构的切换”小节里介绍过，存储重构涉及两个种类，第一种是数据库到缓存，第二种是数据库到另一种异构的表结构数据库里。这里以使用场景较多的数据库到缓存的重构举例讲解，另一种场景比较类似，你可以按此方式自己推演。如果有疑问，也可以写在留言区，我们一起讨论。

包含全量同步、增量数据的实时同步架构如下图 5所示：

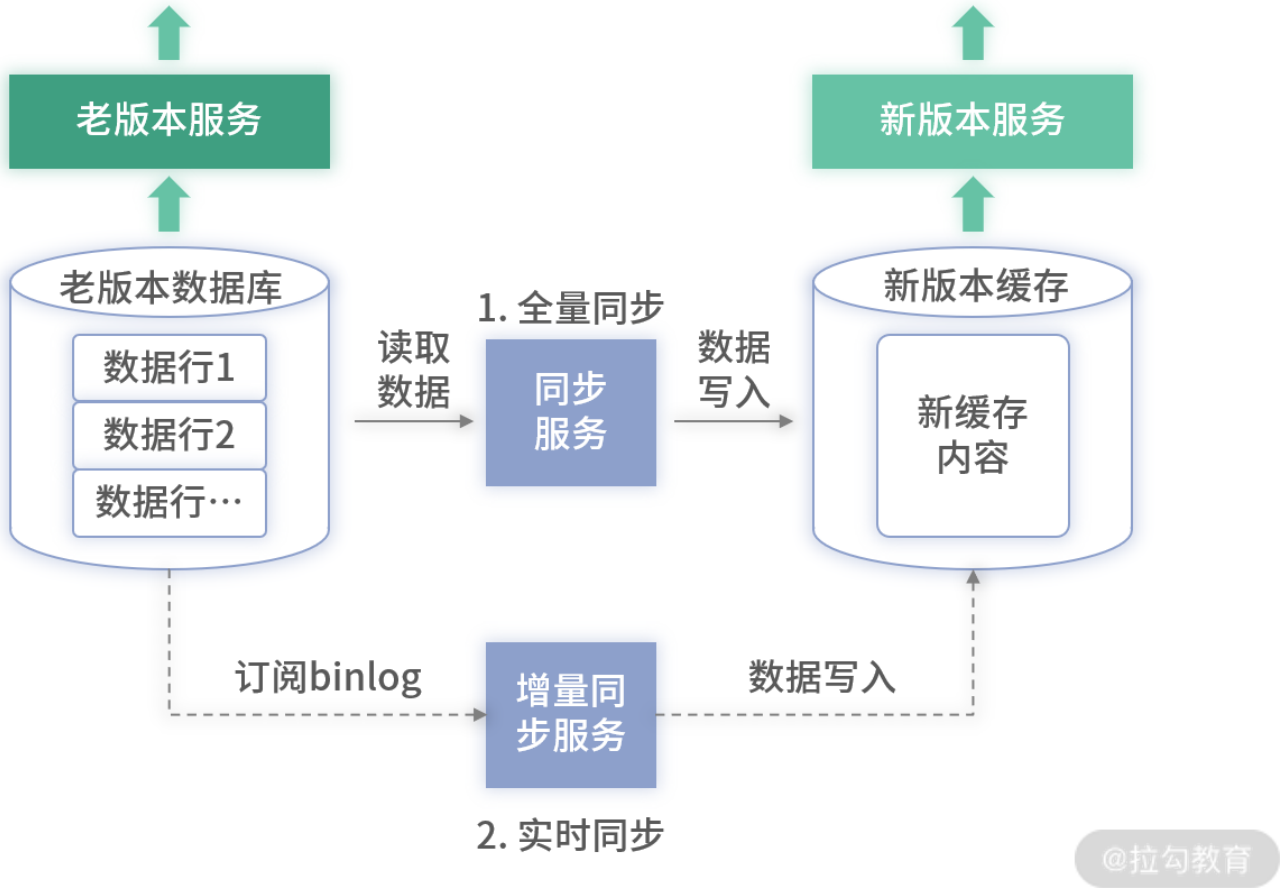

图 5：全量同步和增量同步架构

上述第一步的全量同步（见图 5 的编号 1 处）是将历史数据进行一次全量初始化同步，可以采用 Worker 的方式，对老版本的数据库的数据进行遍历，遍历的 SQL 大致如下：

```
select 数据 from  t_table where id> last_id limit 一批次的数量
```

从数据库遍历读取完之后，便会在同步服务模块里按缓存的格式进行数据格式的转换，转换后的数据写入缓存即可。

上述的数据同步 SQL 没有停止条件，且在未切量前，老库一直会有数据持续写入。使用上述 SQL 进行同步时，会导致全量同步一直执行，出现无法停止的情况。针对这个问题，可以根据当前数据库已有数据量、数据增长的速度以及数据同步的速度，评估在数据同步期间能够产生的数据量，并评估出这期间最多可能产生的数据 ID（截止 ID），并将上述 SQL 修改如下：

```
select 数据 from t_table where id> last_id  and id<截止ID limit 一批次的数量
```

第二步的增量实时同步是在开始进行全量同步时启动的，增量同步使用的是本专栏多次介绍的Binlog 进行。通过在增量同步模块订阅老版本数据库里的数据变更，可以实时获取老版本数据库中新增和变更的数据。

需要注意的是，增量同步需要在全量同步开始前便进行Binlog的订阅。如果在全量同步结束后，再订阅 Binlog 进行增量同步，可能会丢失在全量同步期间发生变更的数据。

比如一张待同步的数据表里有 100 条数据，如果在全量同步前未开启增量同步。当同步至第 90 条数据时，第80 条数据发生了 update 操作，因为此时还没有开启增量同步，那么这第 80 条数据对应的变更就丢失了。为了防止此问题，就需要前置开启增量同步。

最后，增量同步除了需要订阅 update 和 delete 操作外，还需要订阅 insert 操作。因为全量同步在上述截止 ID 之后的数据便不会再同步了，需要增量同步处理此类操作。

数据对比验证

在完成数据迁移之后，并不是立马就能够开始用户切量。还需要做一步非常重要的事情，那便是进行测试。因为做了大规模的代码重构以及存储的切换，只靠人工测试是远远不够的，很容易出现场景遗漏。

因此就需要借助自动化的方式进行测试，我想你还记得在“第 07 讲：基于流量回放的自动化回归测试”的内容。在完成全量数据同步后，可以录制老版本服务的流量，并进行自动化测试回归，它的架构如下图 6 所示：

#后期同学美化下图，注意图中的文字，“第7将”改为“第7讲”

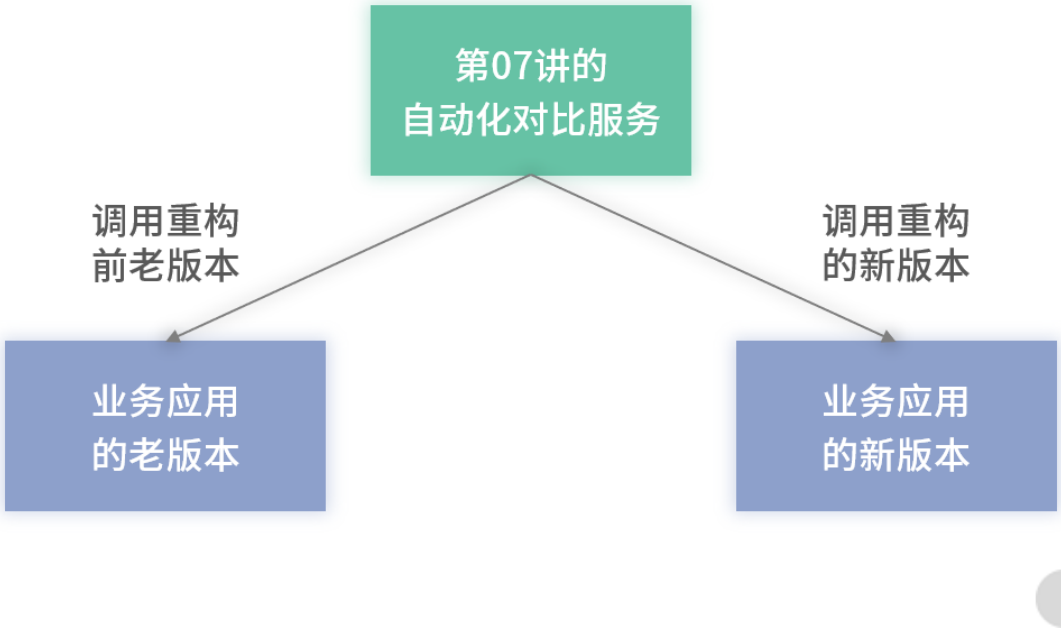

图 6：自动化的数据对比验证架构图

通过一定时间区间的自动化回归，可以保证场景不被遗漏，极大地减少重构切换可能导致的问题，具体如何进行自动化和发现问题，你可以参考“第 07 讲”的内容。

在自动化回归中，可能会出现的某一类问题需要特殊处理，因为增量同步延迟会导致数据对比不一致。原则上这类问题不应该存在，因为基于Binlog的主从同步延迟非常小。但如果遇到上述情况，因为增量同步的时延很小，所以我们可以等待几分钟后再次运行对比不一致的回放请求。

用户切换

完成数据对比之后，下一步需要落地的便是用户切换了。进行用户切换时，有几个原则需要遵循：

1. 切量不能一刀切，即不能一次将所有的用户全部切换至新版本服务里，需要灰度逐步地将用户从老版本切换到新版本服务里；
2. 在灰度切量时，需要尽早发现问题，而不是等到切量快完成的时候才发现问题。

对于上述的几个要求，在切量的具体落地时，可以从以下几点着手落地。

对于影响面小的要求。

1. 首先，对于升级重构的系统涉及的所有用户进行分析并按等级划分。可以按用户的注册时间、是否为会员等进行划分。如果重构的模块是订单模块，可以将用户按历史以来的下单量、订单金额进行排序，订单量小、下单金额低的用户排在最前面。
2. 在用户按上述的维度排序后，可以将用户分为几大批次，比如将所有用户按排序分为五等份。第一等份的用户因为单量小、下单金额少，可以最先进行切换，这样便满足前述提到的“出问题影响面少”的要求。

对于在切量时，尽可能早发现问题的要求。

对上述排序的第一等份的用户，再次进行分析和分类。我们知道，一个系统对外一般会提供多个功能点，比如用户模块会对外提供用户注册、查询用户基本信息、修改个人签名等功能。可以对第一等份里的用户进行数据分析，分析这些用户里哪些用户使用了较多的系统功能。在分析后，可以按使用功能的多少对第一等份里的用户进行排序，使用功能较多的用户排序在前面。在切量时，第一等份里使用系统功能最多的用户会优先进行切量，这也满足了前面所要求的尽可能多发现问题的要求。

总结

本讲介绍了系统升级重构的两种常见的类型。并分别介绍了这两种类型在升级重构后，如何实施让用户尽可能少感知的切换方案。

同时，在包含存储的系统升级重构切换方案里，借鉴了很多本专栏前面介绍过的技术方案，比如基于Binlog的数据同步、基于流量回放的自动化回归方案等。

由此可以看出，很多技术方案之间都是相互依赖的、相互连通的，甚至是成体系的。所以建议你在学习本专栏各讲的内容时，尝试寻找知识之间的联系，完善自己的知识体系，这样才能将专栏里的知识融会贯通，真正应用到实际的业务场景里。

最后，留一道思考题给你，你认为互联网大厂的 App 和后台系统的上线发版是如何灰度切流的？欢迎写在留言区，我们一起讨论。

这一讲就到这里，感谢你学习本次课程，接下来我们将学习 23 | 重构：烟囱式、平台化、中台化的架构演化