kube-sharding简介

kube-sharding 的职责和架构

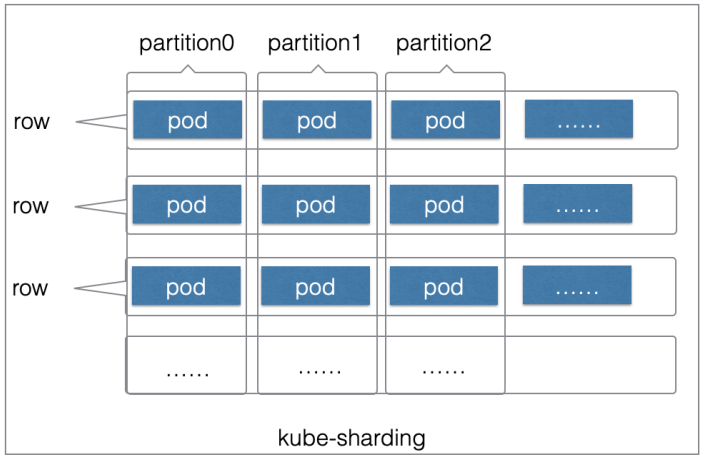
职责

➢**定位**

- 业务服务编排

➢**功能**

- 业务发布
- 健康检查
- 服务挂载
- 故障恢复

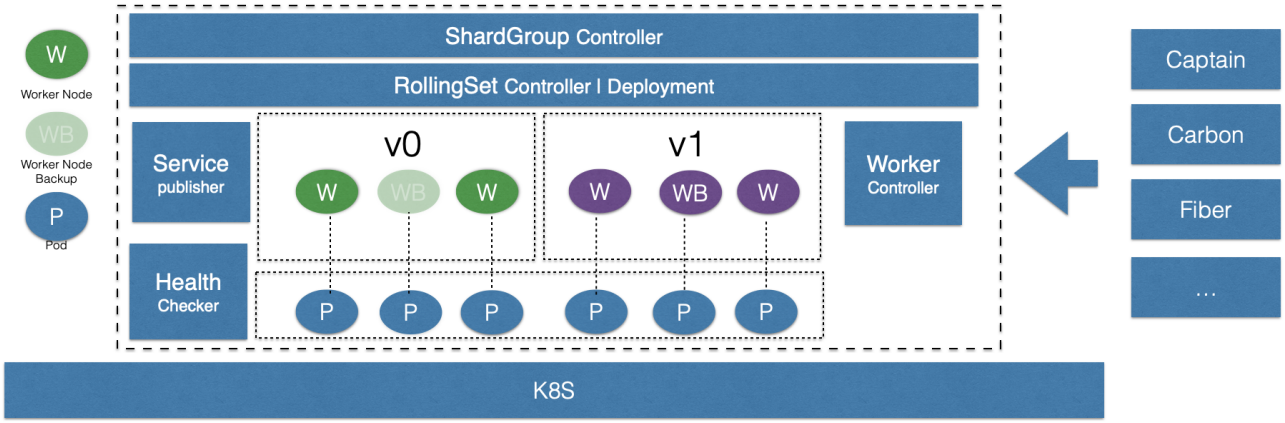


kube-sharding是诞生于搜推广引擎的k8s部署场景之下的k8s controller

- 可以实现业务分片后的多行多列服务的对齐发布
- 能够分别编排业务pod和业务数据（索引，算子或者模型）

kube-sharding是一个业务编排controller，编排对象是需要加载大规模数据的搜索推荐引擎，或者大模型推理业务。支持业务进行分片以及分片后的对齐发布。支持业务单独发布索引/算子/模型数据，使数据变更在满足maxUnavailable的条件下安全的进行。

controller - 在线调度结构



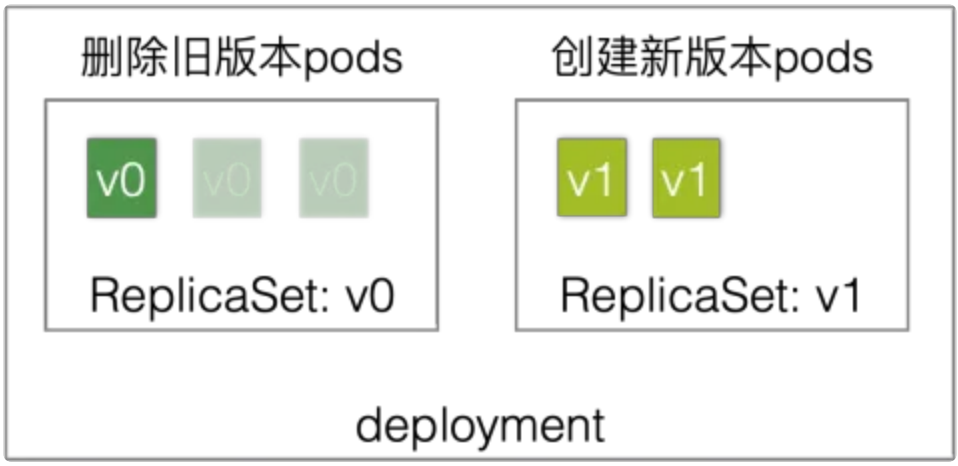
controller/object	目标	描述
shardgroup	行列对齐	实现多个shard同步rolling
rollingset	灰度发布	支持原地升级，升级比例等
workernode	failover	申请backup 恢复错误节点，屏蔽底层信息（hippo pod，k8s pod）
healthchecker	健康检查	发送http请求，实现健康检查，也可以向节点发送命令
servicepublisher	服务挂载	挂载名字服务（cm2、vipserver等）

在k8s的架构下，kube-sharding也是由多个controller组成，通过不同controller的组成来实现一个整体的调度目标。

kube-sharding的调度策略

rollingset 单列业务发布

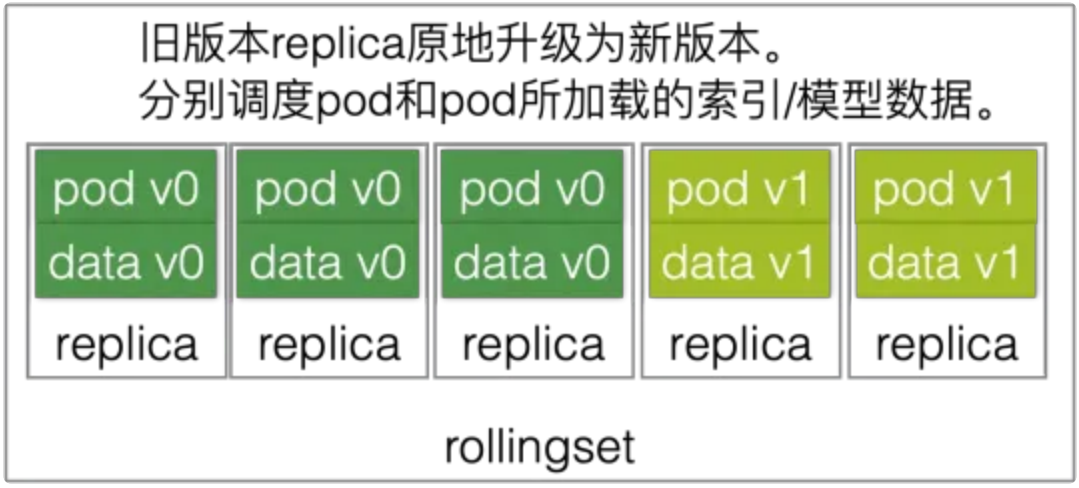
社区的deployment的调度模式



1. deployment 控制不同版本的replicaset的replicas数量。
2. replicaset对自己控制的pod进行增删来实现pod版本的替换。

这种模式实现比较简单，优雅，但是也存在着一些不足：没有对调度对象进行抽象，从而只能调度pod这一种资源，同时难以实现pod原地升级。

c2的rollingset的调度模式

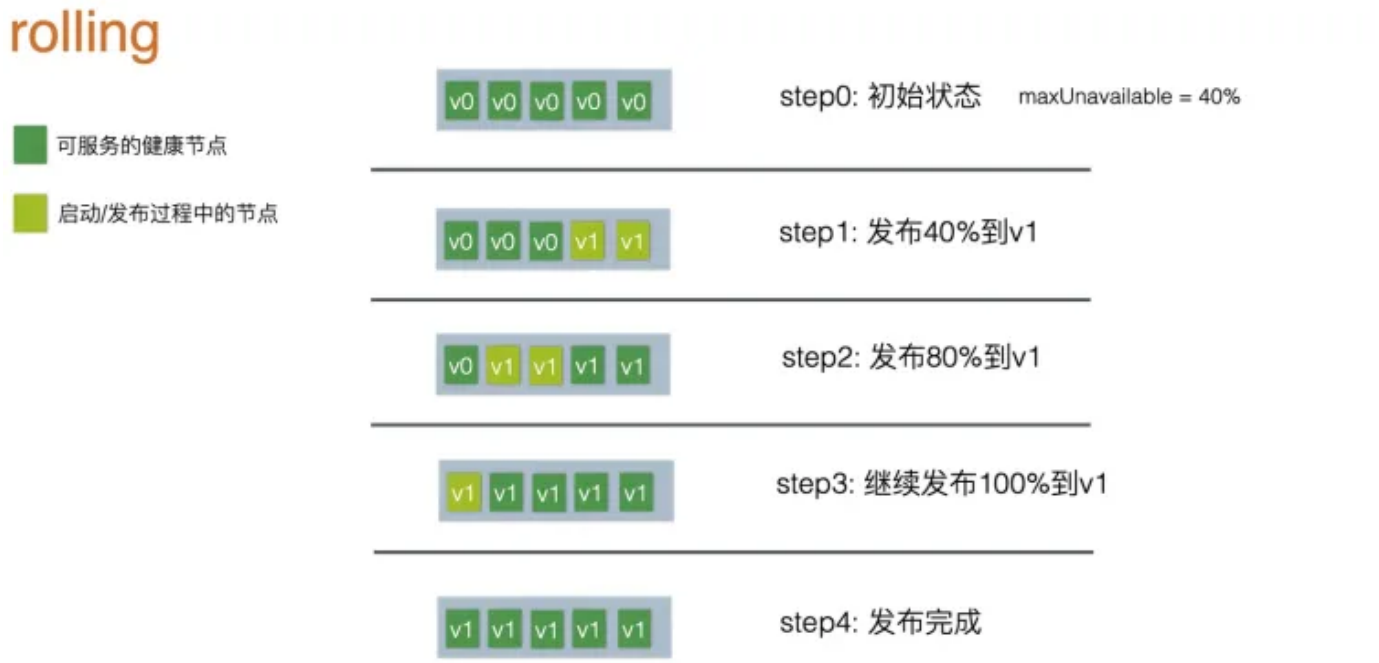


和deployment有两点显著不同

1. 没有直接调度pod资源，而是调度了一个抽象的replica资源，这样实现非常有利于调度能力的扩展：
    a. 对于搜推引擎来说，replica里包括了pod和引擎的索引数据，可以分别发布pod或者索引，从而轻量化的实现索引数据的发布。
    b. 对于普通单列大模型来说，索引数据变成了LoRA和模型，c2可以单独发布LoRA，从而轻松实现大模型的微调，详见：大模型推理框架RTP-LLM对LoRA的支持。
    c. 对于分布式推理，replica又可以实现为包含多列的业务的gang replica，这一点后续会详细介绍。

2. rollingset没有通过ReplicaSet的方式来控制具体replica的版本，而是通过一套算法直接输出其管理的所有replica的版本。这样实现有助于replica的抽象，以及原地升级的实现。由于gpu资源紧张，以及模型下载慢导致启动慢，原地升级是十分有必要的，c2 rollingset在发布业务过程中可以支持：索引/模型/LoRA等业务数据变更，image/command/args等容器层变更，以及cpu/gpu/mem等资源变更在内的几乎全变部更场景的原地升级，助力业务平滑发布。

kube-sharding对于普通的单列无状态业务的发布过程



如图所示：c2 rollingset对于普通单列无状态业务的调度也是基于maxUnavailable对业务进行分批发布。 不同于其它controller，c2并没有直接调度pod，而是调度了一个抽象的Replica资源，这个Replica可以代表一个pod，也可以代表其它资源。基于这个特点，我们对c2 rollingset进行了简单的扩展：

rollingset 多列分布式业务发布

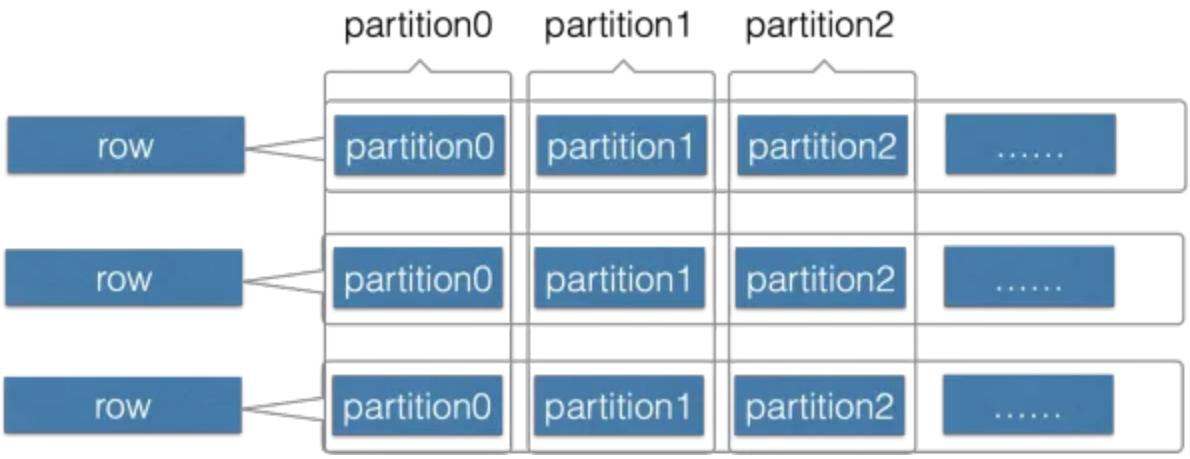
行列式服务

首先我们提出一个行列式服务的概念

当一个业务消耗的资源足够大，大到单节点放不下时就需要做分布式服务。分布式服务有两个维度：

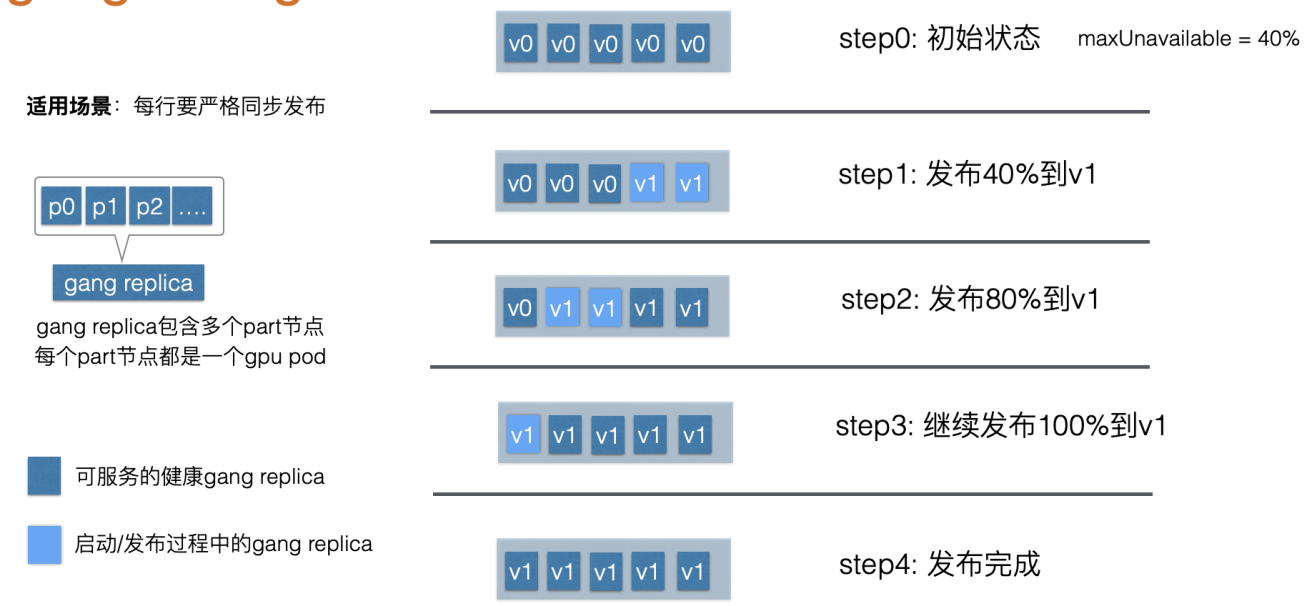
1. 内存/显存/磁盘等资源不足时需要做partition，拆为多列。如图所示就是我们把一个完整的单节点服务做partition，拆为多列。拆分后我们把这一组节点称之为1行n列。对应到大模型中，如果模型参数太大，单节点显存不足时就需要做partition拆为多列。

2. 当cpu/gpu等计算资源不足时就需要扩容为多个同构replica来分担计算需求，我们称之为扩行。比如社区的deployment就是部署同构无状态服务的，deployment中每一个节点都是一行。再带回到我们的场景下，我们将一个row复制为多row，就形成了如下图所示的行列式结构。图中一个row就是一行，对应到大模型中就是一个完整的pipeline，加载一整个模型，包含多个partition。对row扩容就形成了多行多列：



行列式调度，给了业务无限的服务扩展能力。

支持分布式推理的gang rolling



我们将上面的一个row称为一个gang replica，这里的gang和资源分配中的gang有重名，但含义是不同的，这里的gang replica代表一个多partition组成的，需要同时进行部署/发布的多个节点的集合。

然后我们将rollingset调度的replica实现为一个gang replica，具体到大模型场景下，它就代表了一个pipeline。此时我们再对这种gang replica进行调度，就自然支持了多列业务严格对齐的部署/发布/扩缩容。

gang rolling适用于每一行都必须严格对齐发布的场景，是因为类似大模型分布式推理这种业务是有状态的，一个gang内的节点必须一起发布。但是有很多需要分片的业务是无状态的，比如搜索引擎，他们不需要严格对齐发布，严格的对齐反而会影响其发布效率。无状态的分片业务只需要保证在发布过程中，多列业务满足一个整体的可用度即可。

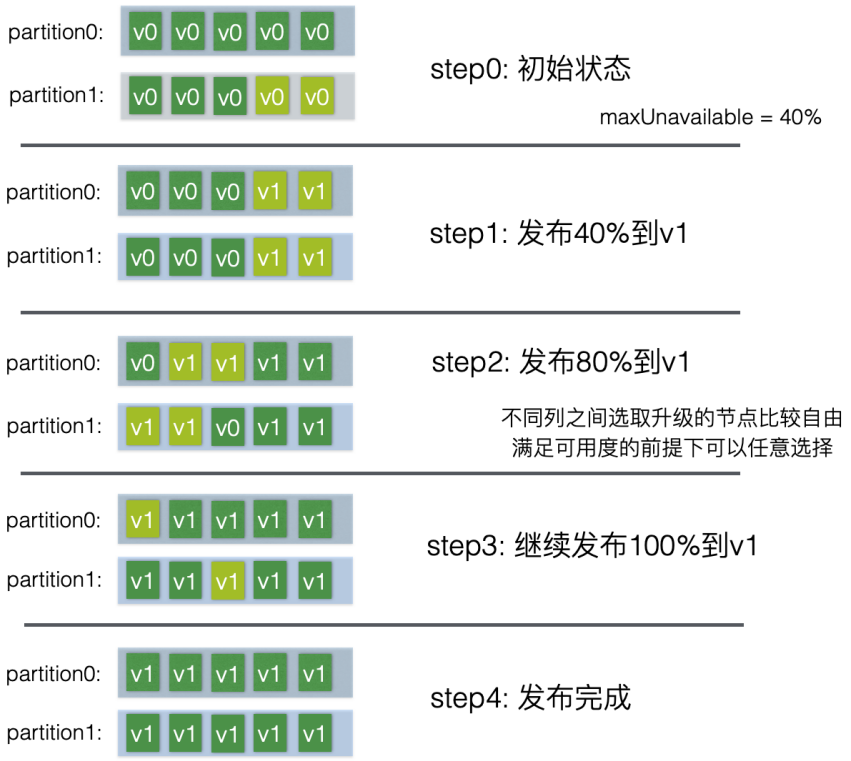
无状态多列业务对齐发布的group rolling

group-rolling

适用场景：需要保持整体服务比例
不要求每行严格同步

可服务的健康节点

启动/发布过程中的节点



对齐发布过程中不需要保持一个严格对齐的行概念，只需要整体可用度比例满足maxUnavailable限制

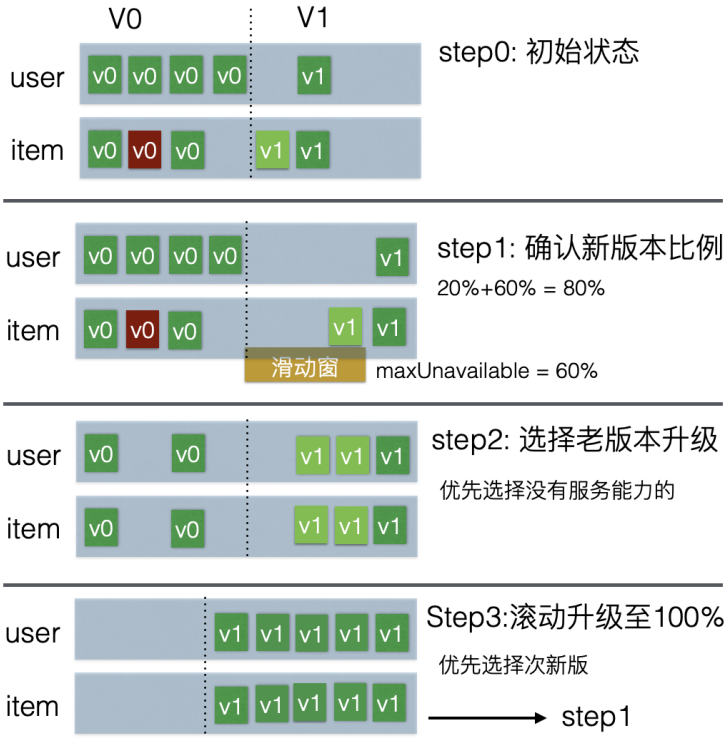
group-rolling 算法实现

➢ 算法分层

- 新版本比例 --以最大不可用节点为滑动窗推进新版本
- 老版本选择 --选择老节点时不破坏服务能力
  --同等情况下优先升级次新版

➢ 算法实现

$$
\begin{cases}
StepRatio = MaxUnavailable \\
TargetRatioMatrix_i = Min(AvailableRatio_{ij}) \\
TargetRatioMatrix_n = TargetRatioMatrix_n + StepRatio \\
\sum_i (TargetRatioMatrix_i) < 1.0 \\
\sum_i (AvailableRatio_{ij}) >= MaxUnavailable
\end{cases}
$$



算法简单来说就是maxUnavailable步长当作一个滑动窗，在所有列整理计算可用度的情况下向前滑动升级。

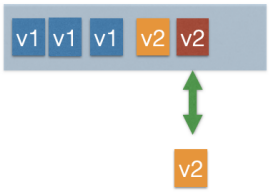
面向错误的设计：Recover策略

Recover-面向错误的设计

在做运维脚本开发时，我们常常认为任务的主流程是正常流程，而各种错误是异常，是打破主流程的因素，是应当想办法消除的。因此我们常常会提出成功率的概念，希望将操作的成功率提升到100%。这种思想的后果是，错误会被当做特殊流程、采用特殊手段处理，比如重试，比如报警，最终等待人工介入。

而事实上，硬件有故障率，软件有代码bug，人的操作有失误，不论是否承认，这些“错误”都在“正常”地发生着。如果这些“正常”被排除在调度流程之外，那我们就需要用80%的时间为20%的事件买单。

1. 进程错误退出
2. 健康检查不过
3. 运维offline
4. 单机资源不够
5. 长时间起不来



节点级别替换，不影响rolling过程

对“错误节点”进行起新下老的替换，其中有两个关键点：

1. 新节点完全启动成功后才会删除老节点，从而保证在线服务的安全。
2. 对于“错误节点”定义，c2中把所有不能正常启动的节点都当成“错误节点”，这样节约了在巨大集群下必然会出现的人工替换坏节点的运维成本。