前言

KubeMin-Cli 是一个基于云原生的PaaS平台，以轻量化工作流的方式来描述应用，结构模型基于**OAM(Open Application Model)**，原生设计的工作流作为核心驱动应用的完整生命周期：初始化资源，创建/更新工作负载，发布事件，监听状态……

KubeMin-Cli中有两个核心能力：

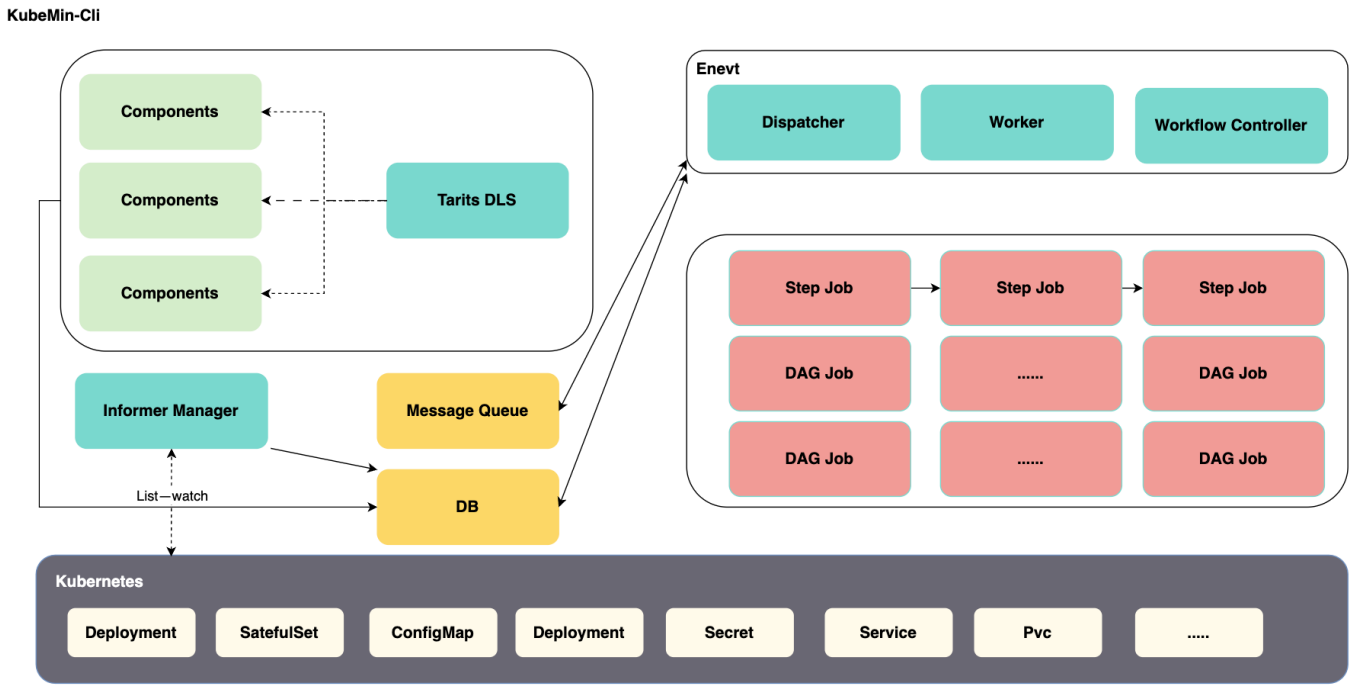
1.Traits：可组合的能力原子

Traits结构将Kubernetes中的Pod，Service,Volume等底层概念抽象化，将存储，容器边车，环境变量，初始化，密钥……以Traits的方式定义，使组件不再依赖复杂的Yaml，而是用组合的方式来构建复杂应用。还提供了“引入模版”功能，能快速创建多个相同形态的新应用（如多套 MySQL），以此来保证生成的资源可用，无冲突，可追溯。

2.Workflow:任务驱动的应用生命周期

每次“部署”“更新”“扩缩容”等操作都被抽象成一条工作流实例：支持并行任务，支持状态一致性，支持人物间依赖，支持持久化执行记录等……
使用List_watch模式来维护应用状态的整个生命周期。

架构图



Workflow （工作流引擎）

KubeMin-Cli 工作流引擎是整个应用交付系统的核心组件，负责将声明的应用配置转换为实际运行在 Kubernetes 集群上的资源。它充当了"编排者"的角色，协调多个组件的创建、更新和删除操作，确保应用的正确部署。引擎支持两种运行模式，可根据部署规模灵活选择：

1.本地模式：使用 NoopQueue，直接扫描数据库执行任务，适用于单实例部署、开发测试。

2.分布式模式：使用 Redis Streams，Kafka等；支持任务分发和故障恢复，适用于多实例部署、生产环境。

> 服务启动时若副本数为偶数，最后启动的实例直接退出，保证最终副本为奇数（便于选主与容灾对称）
> 分布式阈值： `replicas ⩾ 3` 为分布式模式（领导者仅分发）， `replicas == 1` 为单机模式（领导者分发+执行）。
> 领导者周期校验（每 30s）动态切换角色，适配伸缩。
>
> 仅用 Lease 做选主/心跳，不承载副本数
> 决策：副本数通过当前 Pod → OwnerReferences → Controller（Deployment/StatefulSet/DaemonSet）读取期望副本。
>
> 通过上述机制任何分布式队列失败都能优雅回退到本地模式。

工作流引擎解决了以下核心问题：

- **资源依赖管理**：确保 ConfigMap、Secret、PVC 等依赖资源先于 Deployment、StatefulSet 创建
- **执行顺序控制**：支持串行和并行两种执行模式，满足不同场景需求
- **状态追踪**：完整记录每个任务和 Job 的执行状态，便于问题排查
- **故障恢复**：支持任务重试、取消和资源清理，保证系统一致性
- **分布式扩展**：支持多实例部署，通过 Redis Streams/Kafka 实现任务分发

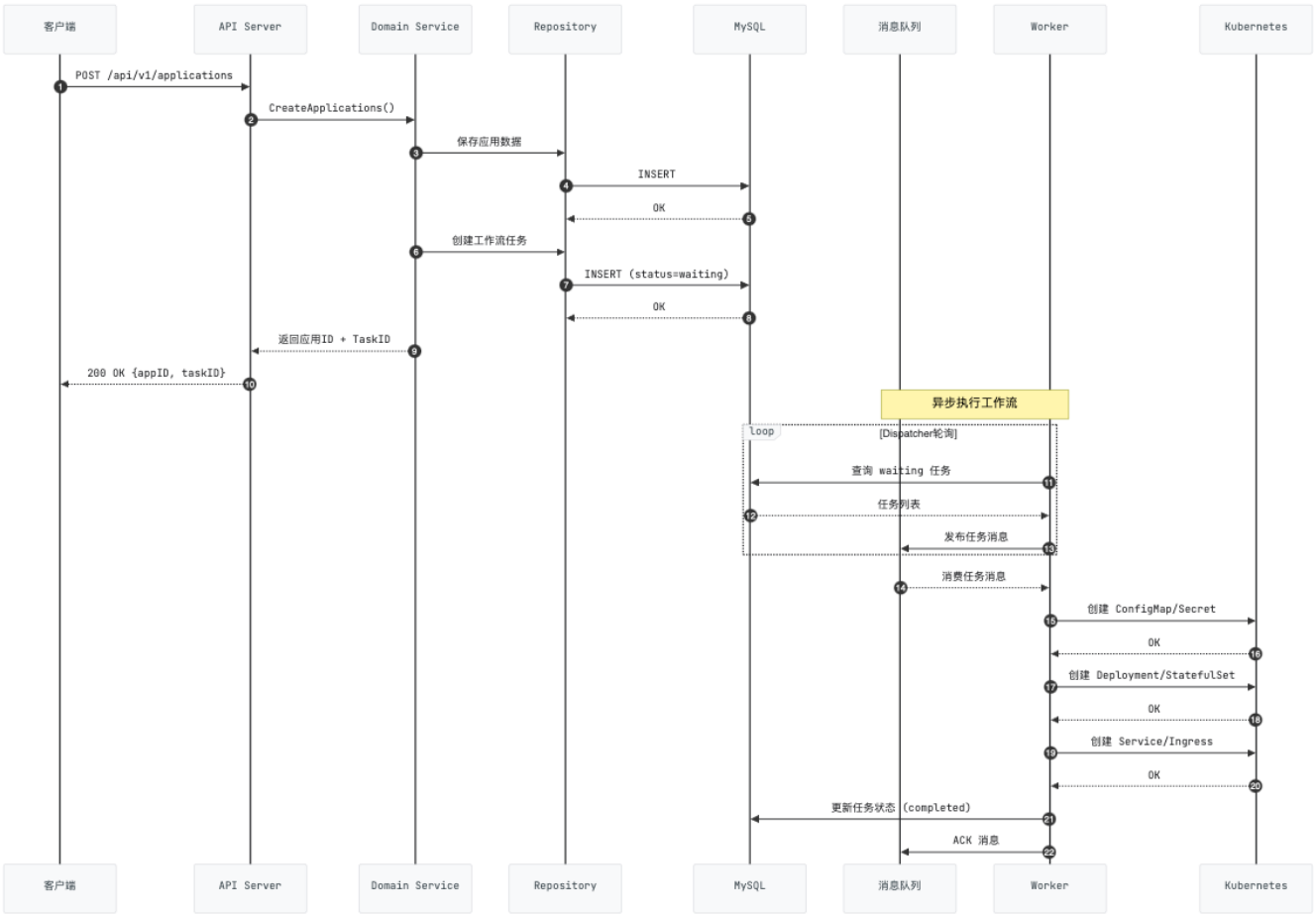
工作流引擎借鉴了 OAM（Open Application Model）的设计思想，支持声明式的工作流定义，这决定了组件编排的顺序：

```
{
  "workflow": [
    {
      "name": "config-step",
      "mode": "StepByStep", //串形模式，组件按声明顺序依次执行，前一个完成后才执行下一个
      "components": ["config", "secret"]
    },
    {
```

```
      "name": "database",
      "mode": "DAG", //并行模式，同一 Step 内的组件并行执行，适合无依赖关系的组件
      "components": ["mysql", "redis"]
    },
    {
      "name": "services",
      "mode": "StepByStep",
      "components": ["backend", "frontend"]
    }
  ]
}
```

任务生命周期由状态机管理,状态分为:

```
// config/consts.go
const (
    StatusWaiting   Status = "waiting"   // 等待执行
    StatusQueued    Status = "queued"    // 已入队，等待 worker 处理
    StatusRunning   Status = "running"   // 执行中
    StatusCompleted Status = "completed" // 执行完成
    StatusFailed    Status = "failed"    // 执行失败
    StatusTimeout   Status = "timeout"   // 执行超时
    StatusCancelled Status = "cancelled" // 已取消
)
```

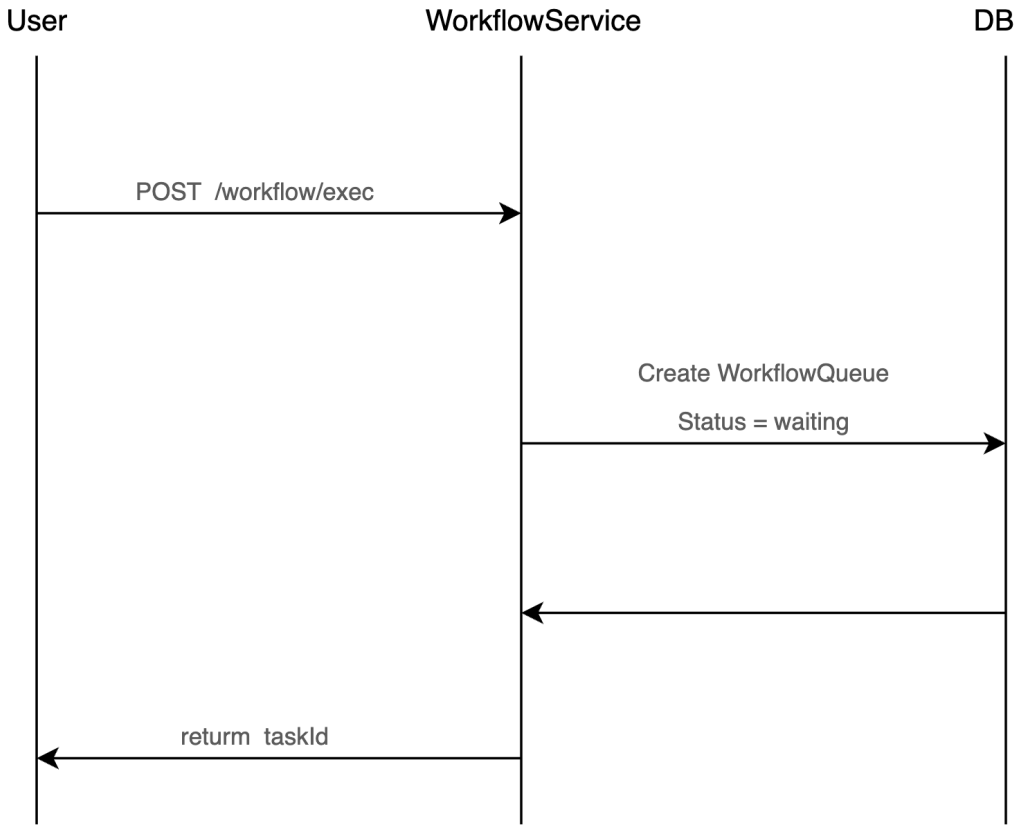


工作流执行流程详解

任务创建与入队

通过 API 触发工作流执行时，系统首先在数据库中创建任务记录，状态设为 `waiting` 。这确保了任务的持久化存储，即使系统重启也不会丢失。API 同步返回任务 ID，用户可以通过该 ID 查询任务状态。

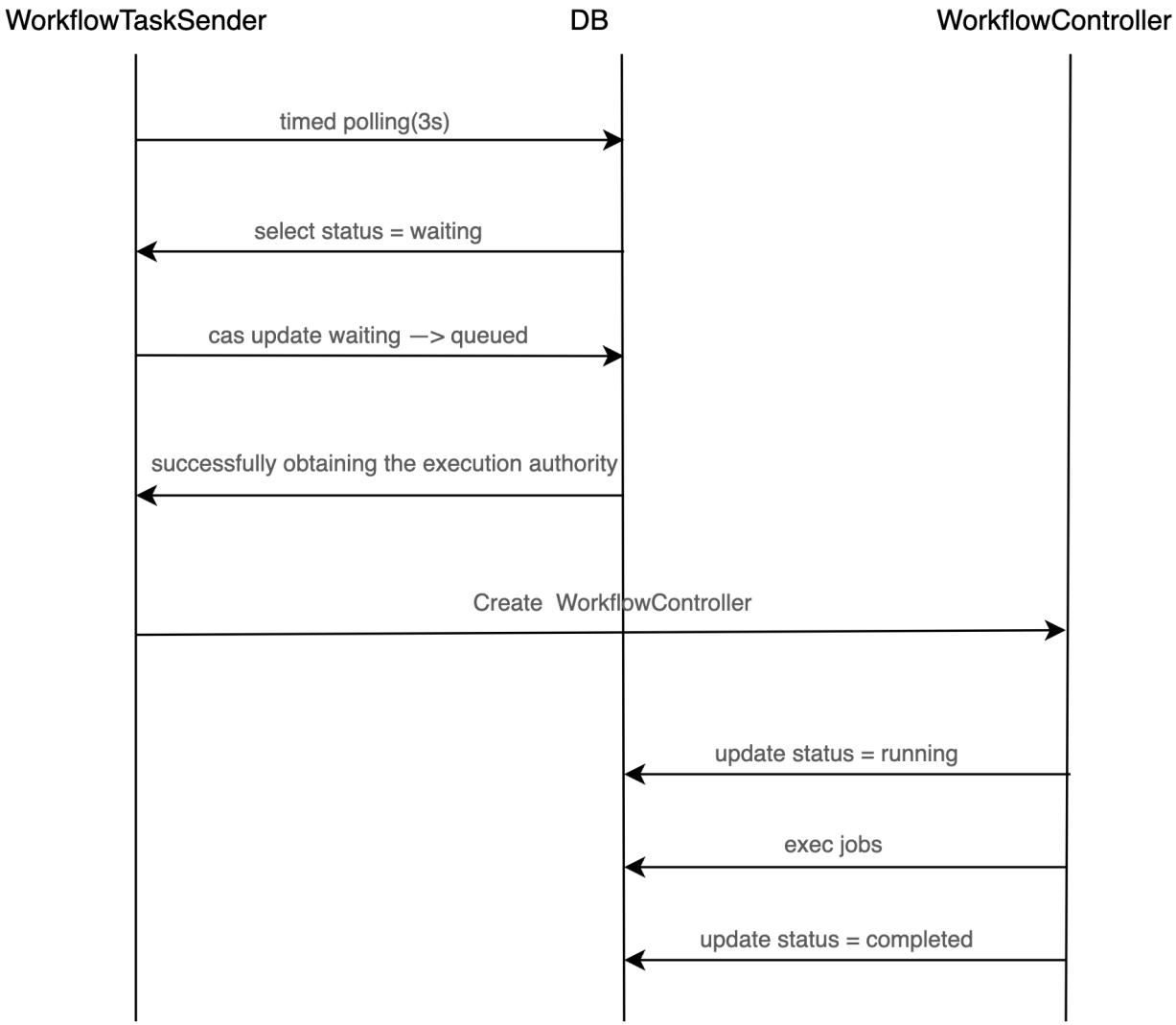
任务首先写入数据库，以确保持久化，初始化状态为 `waiting` ，API 会同步返回TaskId，可通过 TaskId 来追踪Job状态。



本地模式执行流程

本地模式（`msg-type=noop`）适用于单实例部署或开发测试。`WorkflowTaskSender` 定时轮询数据库，发现 `waiting` 任务后通过 CAS（Compare-And-Swap）操作获取执行权，避免并发冲突。获取成功后直接在本进程内创建 `WorkflowController` 执行任务。

这种方式无需外部消息队列依赖，可单服务工作；CAS 操作确保同一任务不会被重复执行；任务在本进程内同步执行。



分布式模式执行流程

分布式模式（`msg-type=redis` 或 `msg-type=kafka`）适用于多实例生产部署。Dispatcher 负责发现任务并发布到消息队列，Worker 从消息队列消费并执行。这种职责分离使系统具备水平扩展能力和故障恢复能力。

Dispatcher 在轮询数据库发现 `waiting` 任务，通过 CAS 获取执行权后发布到消息队列，Redis Streams(或kafka)作为任务分发的通道，Worker 从消息队列消费任务，执行完成后 ACK 确认。

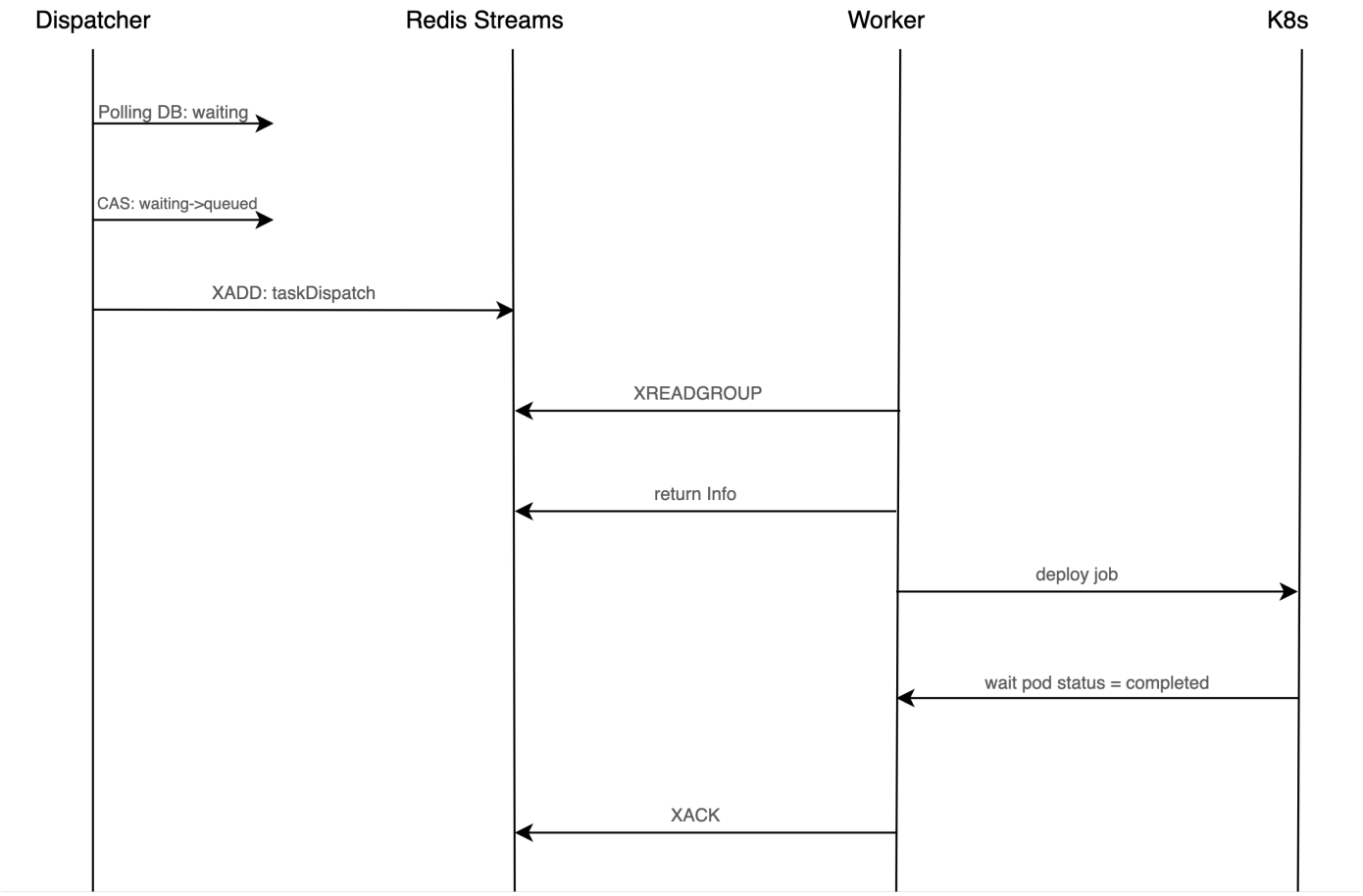
如果API直接写消息队列，消息队列不支持事务的情况下无法保证任务不丢失；

如果API同时写 DB + 队列的情况下分布式事务复杂，难以保证一致性；

如果只用消息队列存任务，则无法查询任务状态、无法支持取消/重试；

当前工作流如此设计是为了先让数据库提供事务保证，任务不会丢失，同时任务状态（waiting/queued/running/completed）可查询；当进程重启后，InitQueue 将 running 任务重置为 waiting；API 可以通过修改数据库状态取消任务;多 Dispatcher 实例不会重复分发同一任务；

Worker 崩溃时，Redis 通过 AutoClaim 重新分配消息，Kafka 通过 Rebalance 重新分配分区。



Step 与 Priority 执行顺序

工作流由多个 Step 组成，每个 Step 包含一组组件。Step 之间按顺序执行，Step 内部根据执行模式（StepByStep 或 DAG）决定组件的执行方式。同时，每个组件生成的 Job 按优先级分组执行，确保依赖资源（如 ConfigMap、Secret）优先创建。

优先级说明：

优先级	值	资源类型	原因
MaxHigh	0	ConfigMap, Secret	被其他资源引用，必须先创建
High	1	PVC, ServiceAccount, Role	Deployment/StatefulSet 可能依赖
Normal	10	Deployment, StatefulSet, Service	主要工作负载
Low	20	清理任务、通知任务	最后执行

执行顺序总结：

1. Step 按声明顺序依次执行
2. StepByStep 模式下，组件逐个执行

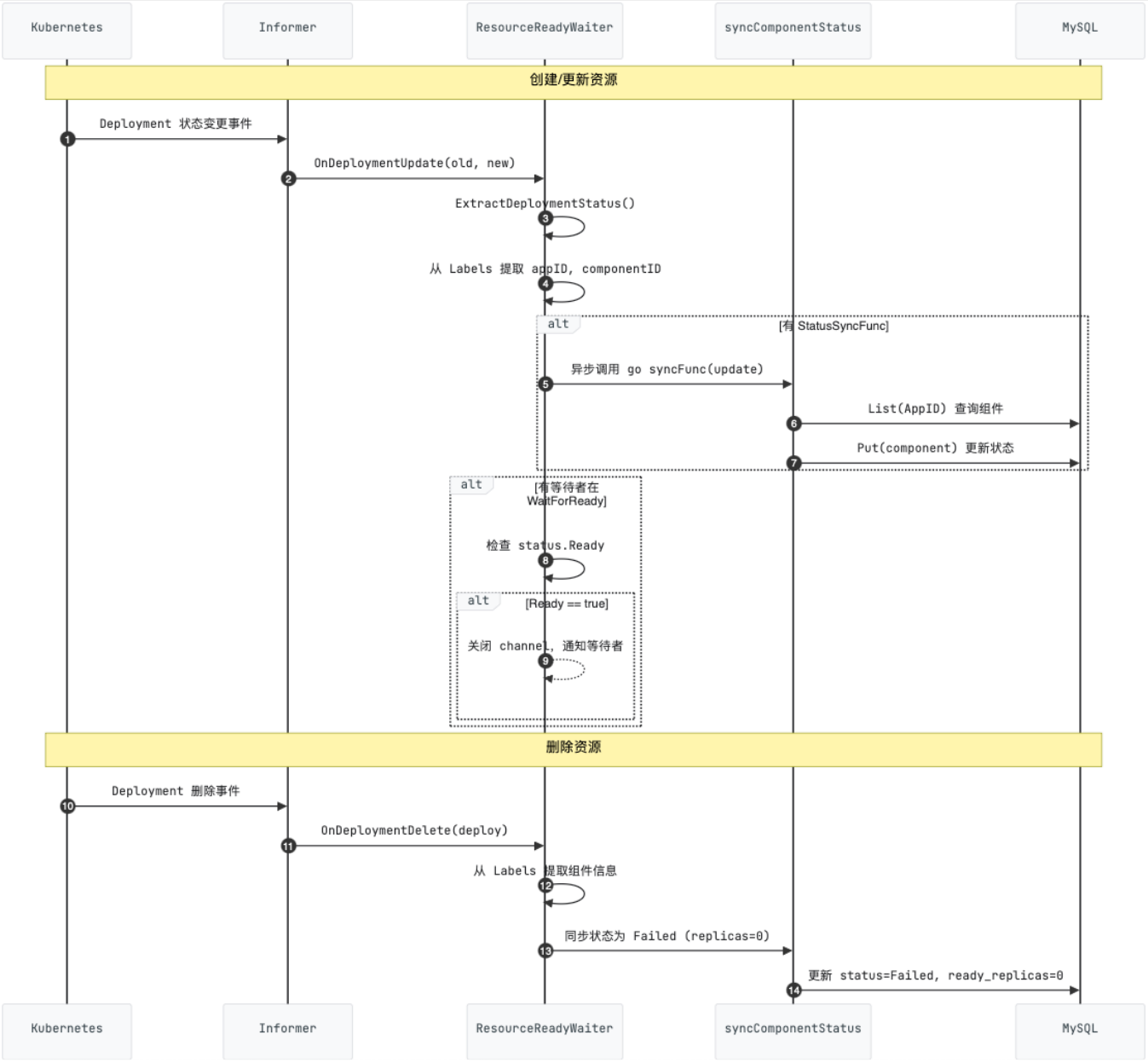
3. DAG 模式下，组件并行执行
4. 每个组件的 Job 按优先级分组，高优先级先执行
5. 同一优先级的 Job 并发执行

Informer 状态同步机制

Informer 状态同步机制用于实时监听 Kubernetes 资源变化，并将组件运行状态同步到数据库。相比传统轮询方式，Informer 采用 List-Watch 机制，它以事件驱动，状态变更及时同步；减少API Server 请求，降低集群负载

组件	文件	职责
`Manager`	`infrastructure/informer/manager.go`	管理 Informer 生命周期，注册事件处理器
`ResourceReadyWaiter`	`infrastructure/informer/waiter.go`	处理资源事件，通知等待者，同步状态到数据库
`types.go`	`infrastructure/informer/types.go`	定义状态类型、等待条目、状态更新结构

状态同步流程



Informer 依赖以下 Labels 来识别和过滤资源，为减少内存消耗，Informer 仅监听带有 `kube-min-cli-appId` 标签的资源，这样 Informer 不会缓存集群中其他应用的 Deployment/StatefulSet，显著降低内存占用。

Label Key	说明	示例值
`kube-min-cli-appId`	应用 ID	`1us2dy3a2yhczm8yes6spm88`
`kube-min-cli-componentId`	组件 ID	`1`
`kube-min-cli-componentName`	组件名称	`nginx`

组件状态

```go
// config/consts.go
type ComponentStatus string

const (
    ComponentStatusRunning ComponentStatus = "Running"  // 所有副本就绪
    ComponentStatusPending ComponentStatus = "Pending"  // 部分副本就绪或启动
中
    ComponentStatusFailed  ComponentStatus = "Failed"   // 失败或已删除
    ComponentStatusUnknown ComponentStatus = "Unknown"  // 未知状态
)
```

状态计算逻辑：

条件	状态
`ready == true` (ReadyReplicas == Replicas)	Running
`readyReplicas > 0`	Pending
`replicas > 0 && readyReplicas == 0`	Pending
`replicas == 0` (资源被删除或缩容为 0)	Failed

并发控制详解

工作流引擎提供一系列配置，控制 StepByStep 模式下同一优先级内 Job 的最大并发数。

参数	默认值	作用层级	说明
`MaxConcurrentWorkflows`	10	工作流层	同时运行的工作流数量上限
`SequentialMaxConcurrency`	1	Job Pool 层	StepByStep 模式下 Job 并发数
DAG 模式	无限制	Job Pool 层	同优先级 Job 全部并行

最大并行 Job 数计算公式

```
最大并行 Job 数 = min(请求数, MaxConcurrentWorkflows) × 每个工作流的同优先级并行 Job 数
```

不同场景下的并行 Job 数示例：

场景	并发请求	执行模式	每工作流 Job 数	最大并行 Job
场景 A	5	StepByStep（并发=1）	3	5 × 1 = **5**
场景 B	5	DAG	3	5 × 3 = **15**
场景 C	10	DAG	3	10 × 3 = **30**
场景 D	15	DAG	3	10 × 3 = **30**（5个排队）
场景 E	10	StepByStep（并发=2）	4	10 × 2 = **20**

说明：

- 场景 D 中，由于 `MaxConcurrentWorkflows=10`，超出的 5 个请求会排队等待
- 实际执行时，还需考虑 Priority 分组，只有同优先级的 Job 才会真正并行
- 每个 webservice 组件可能生成多个 Job（Deployment + Service + PVC/Ingress 等）

创建应用请求（POST /applications）：

```
{
  "name": "demo-app",
  "namespace": "default",
  "version": "1.0.0",
  "project": "demo-project",
  "description": "StepByStep 模式示例应用",
  "component": [
    {
      "name": "app-config-1",
      "type": "config",
      "nameSpace": "default",
      "replicas": 1,
      "properties": {
        "conf": {
          "database.host": "mysql.default.svc",
          "database.port": "3306"
        }
      }
```

```
    },
    {
      "name": "app-config-2",
      "type": "config",
      "nameSpace": "default",
      "replicas": 1,
      "properties": {
        "conf": {
          "redis.host": "redis.default.svc",
          "redis.port": "6379"
        }
      }
    },
    {
      "name": "app-config-3",
      "type": "config",
      "nameSpace": "default",
      "replicas": 1,
      "properties": {
        "conf": {
          "log.level": "info",
          "log.format": "json"
        }
      },
      "traits": {}
    },
    {
      "name": "backend",
      "type": "webservice",
      "image": "myregistry/backend:v1.0.0",
      "nameSpace": "default",
      "replicas": 2,
      "properties": {
        "ports": [{"port": 8080, "expose": true}],
        "env": {
          "APP_ENV": "production"
        }
      },
      "traits": {}
    },
    {
      "name": "frontend",
      "type": "webservice",
```

```
      "image": "myregistry/frontend:v1.0.0",
      "nameSpace": "default",
      "replicas": 2,
      "properties": {
        "ports": [{"port": 80, "expose": true}],
        "env": {
          "API_URL": "http://backend:8080"
        }
      },
      "traits": {}
    }
  ],
  "workflow": [
    {
      "name": "deploy-all",
      "mode": "StepByStep",
      "components": ["app-config-1", "app-config-2", "app-config-3",
"backend", "frontend"]
    }
  ]
}
```

执行结果：一个组件生成 3 个 ConfigMap Job 和 2 个 Deployment Job

示例 1：StepByStep 模式，SequentialMaxConcurrency=1（默认）

```
时间轴
──────────────────────────────────────────────────────────────────────────────>

Priority 0 (ConfigMap):
  [ConfigMap-1] ──完成──> [ConfigMap-2] ──完成──> [ConfigMap-3] ── 完成 ──┐
                                                                          │
                                                                  等待全部完
成                                                                        │
                                                                          ↓
Priority 10 (Deployment):[Deployment-1] ──完成──> [Deployment-2] ──完成──>
结束
```

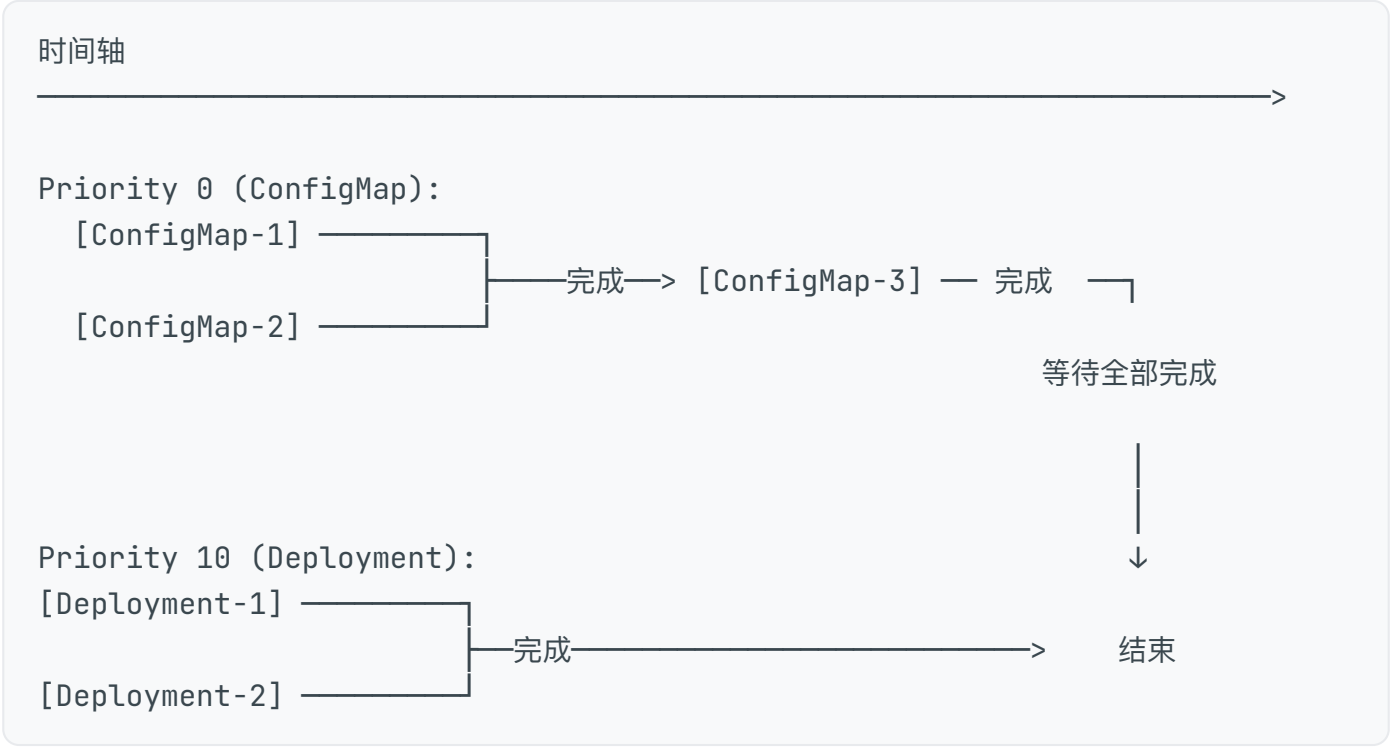
执行顺序：

1. 3个 ConfigMap Job **串行执行**

2. 当所有的 (Priority 0)级别的Job执行完毕后，才开始执行(Priority 10)级别的任务
3. (Priority 10)级别 的 2 个 Deployment Job **逐个串行**执行

示例 2：StepByStep 模式，SequentialMaxConcurrency=2

同样场景，但设置 `--workflow-sequential-max-concurrency=2`

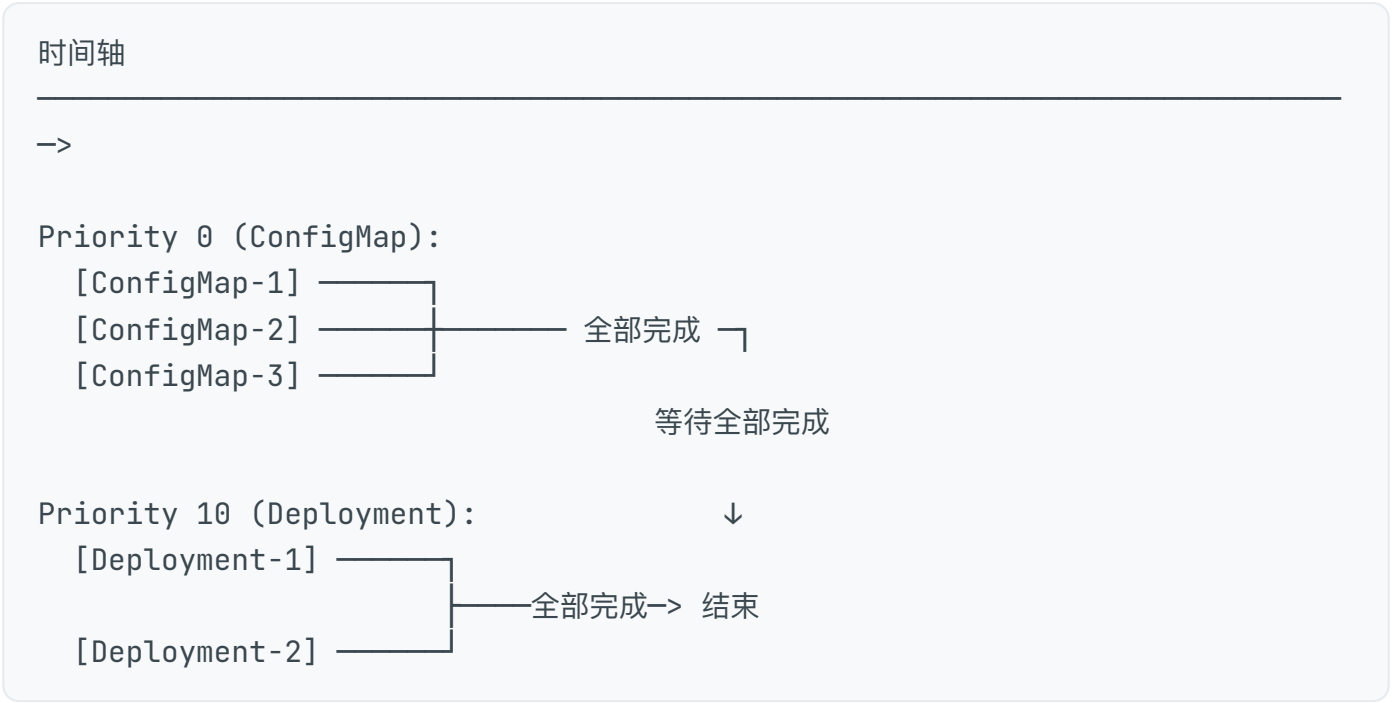


执行过程：

1. Priority 0：前 2 个 ConfigMap Job **并行**执行，完成后执行第 3 个
2. Priority 0 全部完成后，才开始 Priority 10
3. Priority 10：2 个 Deployment Job **并行**执行（因为 Job 数 ≤ 并发配置）

示例 3：DAG 模式（忽略并发配置）

场景：DAG 模式下同一 Step 的所有组件并行执行



执行过程：

1. Priority 0：所有 ConfigMap Job **全部并行**执行（忽略 SequentialMaxConcurrency）
2. Priority 0 全部完成后，才开始 Priority 10
3. Priority 10：所有 Deployment Job **全部并行**执行

示例 4：多 Step 组合执行

场景：2 个 Step，Step1 是 StepByStep（config 组件），Step2 是 DAG（api + web 组件）

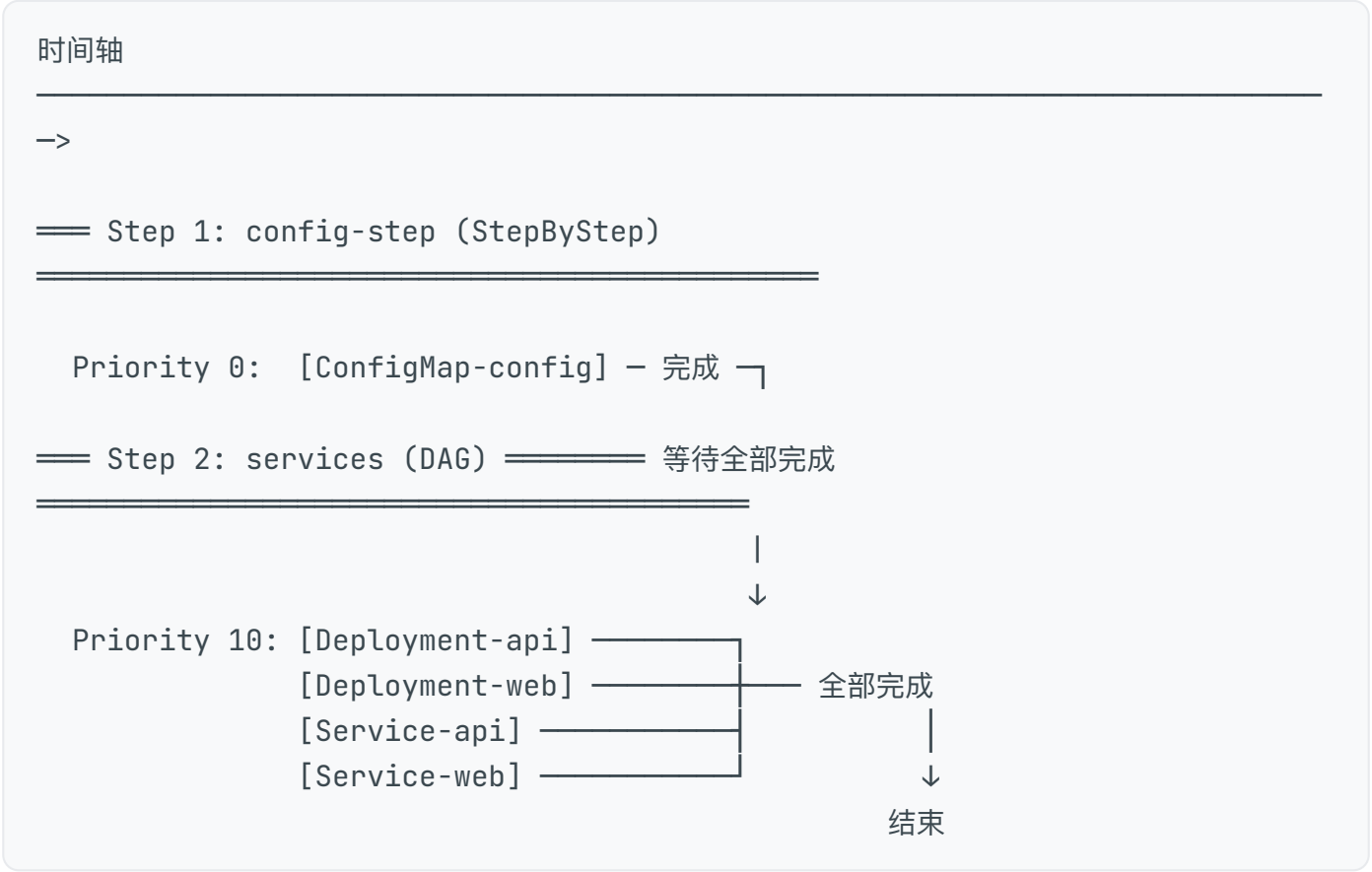
```
"workflow": [
    {
      "name": "config-step",
      "mode": "StepByStep",
      "components": ["config"]
    },
    {
      "name": "services",
      "mode": "DAG",
      "components": ["api", "web"]
    }
  ]
```

执行结果：

- Step 1 生成：ConfigMap(Priority 0)

- Step 2 生成: Deployment-api(P10) + Service-api(P10) + Deployment-web(P10) + Service-web(P10)

执行流程（SequentialMaxConcurrency=2）：



执行过程：

1. **Step 1** 执行完成后，才开始 **Step 2**
2. Step 2 是 DAG 模式，同优先级的所有 Job 并行执行

并发配置建议/调优

工作流相关可调整参数

参数	默认值	说明	推荐配置
`--workflow-sequential-max-concurrency`	1	串行步骤内部最大并发数	生产环境建议 1-3
`--workflow-local-poll-interval`	3s	本地模式轮询间隔	开发环境可设为 1s
`--workflow-dispatch-poll-interval`	3s	Dispatcher 扫描间隔	生产环境建议 3-5s
`--workflow-worker-stale-interval`	15s	Worker 过期检查间隔	生产环境建议 15-30s
`--workflow-worker-autoclaim-idle`	60s	AutoClaim 最小空闲时间(仅 redis模式下使用)	应大于 Job 最大执行时间
`--workflow-worker-autoclaim-count`	50	AutoClaim 批量大小(仅 redis模式下使用)	根据任务量调整
`--workflow-worker-read-count`	10	Worker 单次读取消息数	根据处理能力调整
`--workflow-worker-read-block`	2s	Worker 阻塞读取超时	建议 2-5s
`--workflow-default-job-timeout`	60s	Job 默认超时时间	根据业务需求调整
`--workflow-max-concurrent`	10	最大并发工作流数	根据资源限制调整

Worker 弹性配置

参数	默认值	说明	推荐配置
`--workflow-worker-max-read-failures`	0	Worker 连续读取失败熔断阈值，0=永不熔断	生产环境建议 0（无限重试）
`--workflow-worker-max-claim-failures`	0	Worker 连续 AutoClaim 失败熔断阈值，0=永不熔断（仅 Redis 模式有效）	生产环境建议 0（无限重试）
`--workflow-worker-backoff-min`	200ms	失败后指数退避最小等待时间	100ms-500ms
`--workflow-worker-backoff-max`	5m	失败后指数退避最大等待时间	1m-10m

弹性机制说明：

- **指数退避**：网络抖动时的自愈重试，等待时间按 `min → min*2 → min*4 → ... → max` 增长
- **熔断阈值**：持续故障时的保护机制，防止 Worker 空转消耗资源
- 设置为 `0` 表示永不熔断，适合高可用场景（推荐）

消息队列参数

参数	默认值	说明	推荐配置
`--msg-type`	redis	消息队列类型	noop/redis/kafka
`--msg-channel-prefix`	kubemin	消息通道前缀	根据环境区分，如 kubemin-prod
`--msg-redis-maxlen`	50000	Redis Stream 最大长度	根据消息量和内存调整

Kafka 参数

参数	默认值	说明	推荐配置
`--msg-kafka-brokers`	-	Kafka Broker 地址	生产环境建议配置多个
`--msg-kafka-group-id`	kubemin-workflow-workers	消费者组 ID	不同环境使用不同 ID
`--msg-kafka-offset-reset`	earliest	偏移量重置策略	生产环境建议 earliest

配置示例

开发环境配置（本地模式）

```yaml
# config-dev.yaml - 开发环境使用本地模式，无需外部队列依赖
workflow:
  sequentialMaxConcurrency: 1
  localPollInterval: 1s          # 更快的轮询
  dispatchPollInterval: 1s
  workerStaleInterval: 10s
  workerAutoClaimMinIdle: 30s
  workerAutoClaimCount: 10
  workerReadCount: 5
  workerReadBlock: 1s
  defaultJobTimeout: 30s
  maxConcurrentWorkflows: 5
  # ═══ Worker 弹性配置 ═══
  workerMaxReadFailures: 0       # 读取失败熔断阈值，0=永不熔断（推荐）
  workerMaxClaimFailures: 0      # AutoClaim 失败熔断阈值，0=永不熔断
  workerBackoffMin: 100ms        # 指数退避最小等待：100ms → 200ms → 400ms →
...
  workerBackoffMax: 1m           # 指数退避最大等待（封顶）

messaging:
  type: noop                     # 本地模式，无需 Redis/Kafka
  channelPrefix: kubemin-dev
```

生产环境配置（Redis 模式）

```yaml
# config-prod-redis.yaml - 中等规模生产环境推荐配置
workflow:
  sequentialMaxConcurrency: 3
  localPollInterval: 3s
  dispatchPollInterval: 3s
  workerStaleInterval: 15s
  workerAutoClaimMinIdle: 60s    # 确保大于 defaultJobTimeout
  workerAutoClaimCount: 50
  workerReadCount: 10
  workerReadBlock: 2s
  defaultJobTimeout: 60s
  maxConcurrentWorkflows: 20     # 根据 K8s API 承载能力调整
  # ═══ Worker 弹性配置 ═══
  workerMaxReadFailures: 0       # 0=永不熔断，依靠指数退避自愈网络抖动
  workerMaxClaimFailures: 0      # 0=永不熔断，Redis AutoClaim 失败时持续重试
  workerBackoffMin: 200ms        # 指数退避最小等待
  workerBackoffMax: 5m           # 指数退避最大等待（封顶）

messaging:
  type: redis
  channelPrefix: kubemin-prod
  redisStreamMaxLen: 100000      # 根据内存和消息量调整

# Redis 连接配置（复用 Cache 配置）
cache:
  cacheHost: redis.prod.svc.cluster.local
  cacheProt: 6379
  cacheType: redis
  cacheDB: 0
  cacheTTL: 24h
  keyPrefix: "kubemin:cache:"
```

大规模生产环境配置（Kafka 模式）

```yaml
# config-prod-kafka.yaml - 大规模生产环境推荐配置
workflow:
  sequentialMaxConcurrency: 5
  localPollInterval: 3s
  dispatchPollInterval: 3s
  workerStaleInterval: 20s
  workerAutoClaimMinIdle: 120s   # Kafka 模式下不会产生任何效果
  workerAutoClaimCount: 100      # Kafka 模式下不会产生任何效果
```

```yaml
  workerReadCount: 20            # Kafka 吞吐量高，可增加读取数
  workerReadBlock: 3s
  defaultJobTimeout: 120s
  maxConcurrentWorkflows: 50     # 大规模部署
  # ═══ Worker 弹性配置（Kafka 模式下正常生效）═══
  workerMaxReadFailures: 0       # 0=永不熔断，Kafka 读取失败时持续重试
  workerMaxClaimFailures: 0      # Kafka 模式下不生效（AutoClaim 返回空）
  workerBackoffMin: 200ms        # 指数退避最小等待
  workerBackoffMax: 5m           # 指数退避最大等待（封顶）

messaging:
  type: kafka
  channelPrefix: kubemin-prod
  kafkaBrokers:
    - kafka-0.kafka.prod.svc.cluster.local:9092
    - kafka-1.kafka.prod.svc.cluster.local:9092
    - kafka-2.kafka.prod.svc.cluster.local:9092
  kafkaGroupID: kubemin-workflow-workers
  kafkaAutoOffsetReset: earliest
```

Kafka 模式依赖 Kafka 原生的 session.timeout.ms 和 heartbeat.interval.ms 来控制故障检测和 Rebalance;可以直接在源码中更改:

```go
// kafka.go:90-99
k.reader = kafka.NewReader(kafka.ReaderConfig{
    Brokers:        k.cfg.Brokers,
    Topic:          k.cfg.Topic,
    GroupID:        k.cfg.GroupID,
    StartOffset:    startOffset,
    MinBytes:       1,
    MaxBytes:       10e6,
    MaxWait:        500 * time.Millisecond,
    CommitInterval: 0,
    // SessionTimeout 故障检测延迟：默认 30s 的 SessionTimeout 意味着消费者崩溃后需要 30 秒才能被检测到
    // HeartbeatInterval
    // RebalanceTimeout
})
```

性能调优

场景	推荐配置	说明
高吞吐量	`workerReadCount=20`，`maxConcurrentWorkflows=50`	增加并发处理能力
低延迟	`dispatchPollInterval=1s`，`workerReadBlock=1s`	减少轮询间隔
资源受限	`maxConcurrentWorkflows=5`，`workerReadCount=3`	限制并发避免过载
长时任务	`defaultJobTimeout=300s`，`workerAutoClaimMinIdle=360s`	延长超时时间

消息队列选型建议

场景	推荐	原因
开发测试	noop	无需外部依赖，快速启动
中小规模（<100 任务/s）	redis	部署简单，延迟低
大规模（>100 任务/s）	kafka	高吞吐量，强持久化
已有 Redis 基础设施	redis	复用现有资源
已有 Kafka 基础设施	kafka	复用现有资源
需要消息回溯	kafka	支持历史消息重放

关键配置约束

1. **workerAutoClaimMinIdle > defaultJobTimeout**：确保任务超时后才被认领
2. **workerStaleInterval < workerAutoClaimMinIdle**：确保定期检查过期任务
3. **Redis maxLen**：根据消息量和内存设置，防止内存溢出
4. **Kafka partitions ≥ workers**：确保每个 Worker 都能分配到分区
5. **workerMaxReadFailures = 0**：生产环境推荐设为 0，依靠退避机制处理网络抖动，避免误熔断
6. **workerBackoffMax > workerReadBlock**：确保退避时间有意义，避免退避时间过短

Kafka 模式特别说明

注意：以下配置在 Kafka 模式下**不生效**，Kafka 依赖原生 Rebalance 机制处理消费者故障：

- `workerAutoClaimMinIdle` - Kafka 无需手动认领超时消息

- `workerAutoClaimCount` - Kafka 的 `AutoClaim()` 直接返回空
- `workerMaxClaimFailures` - 因 AutoClaim 不生效，此熔断阈值无意义

以下配置在 Kafka 模式下**正常生效**：

- `workerMaxReadFailures` - Kafka 读取失败时的熔断保护
- `workerBackoffMin/Max` - 网络抖动时的指数退避重试

OAM Traits（特征）

总览

Trait 名称	说明	对应 Kubernetes 资源
Storage	存储挂载	PVC, EmptyDir, ConfigMap, Secret Volume
Init	初始化容器	InitContainer
Sidecar	边车容器	Container
Envs	单个环境变量定义	EnvVar
EnvFrom	批量导入环境变量	EnvFromSource
Probes	健康检查探针	LivenessProbe, ReadinessProbe, StartupProbe
Resources	计算资源限制	ResourceRequirements
Ingress	入口流量路由	Ingress
RBAC	权限访问控制	ServiceAccount, Role, RoleBinding, ClusterRole, ClusterRoleBinding

Envs 简化环境变量

字段详解

字段	类型	限制	默认值	说明
name	string	必填	-	环境变量名称
valueFrom	object	必填	-	值来源，四种来源选其一
valueFrom.static	*string	可选	-	静态字符串值
valueFrom.secret	object	可选	-	从 Secret 中读取
valueFrom.secret.name	string	必填	-	Secret 资源名称
valueFrom.secret.key	string	必填	-	Secret 中的 key
valueFrom.config	object	可选	-	从 ConfigMap 中读取
valueFrom.config.name	string	必填	-	ConfigMap 资源名称
valueFrom.config.key	string	必填	-	ConfigMap 中的 key
valueFrom.field	*string	可选	-	从 Pod 字段读取（FieldRef）

值来源类型

类型	说明	Kubernetes 对应
static	静态字符串值	`value`
secret	从 Secret 引用	`secretKeyRef`
config	从 ConfigMap 引用	`configMapKeyRef`
field	从 Pod 字段引用	`fieldRef`

常用 Field 字段

字段路径	说明
`metadata.name`	Pod 名称
`metadata.namespace`	Pod 命名空间
`metadata.uid`	Pod UID
`spec.nodeName`	所在节点名称
`spec.serviceAccountName`	ServiceAccount 名称
`status.podIP`	Pod IP 地址
`status.hostIP`	宿主机 IP

使用示例

静态值

```
{
  "envs": [
    {
      "name": "APP_ENV",
      "valueFrom": {
        "static": "production"
      }
    },
    {
      "name": "LOG_LEVEL",
      "valueFrom": {
        "static": "info"
      }
    }
  ]
}
```

生成结果：

```
env:
  - name: APP_ENV
    value: "production"
  - name: LOG_LEVEL
    value: "info"
```

从 Secret 读取

```
{
  "envs": [
    {
      "name": "DB_PASSWORD",
      "valueFrom": {
        "secret": {
          "name": "db-credentials",
          "key": "password"
        }
```

```
      }
    },
    {
      "name": "API_KEY",
      "valueFrom": {
        "secret": {
          "name": "api-secrets",
          "key": "key"
        }
      }
    }
  ]
}
```

生成结果：

```yaml
env:
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: db-credentials
        key: password
  - name: API_KEY
    valueFrom:
      secretKeyRef:
        name: api-secrets
        key: key
```

从 ConfigMap 读取

```
{
  "envs": [
    {
      "name": "DATABASE_URL",
      "valueFrom": {
        "config": {
          "name": "app-config",
          "key": "database_url"
        }
      }
    }
  ]
}
```

生成结果：

```
env:
  - name: DATABASE_URL
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: database_url
```

从 Pod 字段读取

```
{
  "envs": [
    {
      "name": "POD_NAME",
      "valueFrom": {
        "field": "metadata.name"
      }
    },
    {
      "name": "POD_IP",
      "valueFrom": {
        "field": "status.podIP"
      }
    },
    {
```

```
      "name": "NODE_NAME",
      "valueFrom": {
        "field": "spec.nodeName"
      }
    }
  ]
}
```

生成结果：

```
env:
  - name: POD_NAME
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: metadata.name
  - name: POD_IP
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: status.podIP
  - name: NODE_NAME
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: spec.nodeName
```

混合使用

```
{
  "envs": [
    {
      "name": "APP_NAME",
      "valueFrom": {
        "static": "my-service"
      }
    },
    {
      "name": "DB_HOST",
      "valueFrom": {
```

```
        "config": {
          "name": "db-config",
          "key": "host"
        }
      }
    },
    {
      "name": "DB_PASSWORD",
      "valueFrom": {
        "secret": {
          "name": "db-credentials",
          "key": "password"
        }
      }
    },
    {
      "name": "INSTANCE_ID",
      "valueFrom": {
        "field": "metadata.name"
      }
    }
  ]
}
```

注意事项

1. **单一来源**：每个环境变量的 `valueFrom` 只能指定一种来源
2. **资源存在**：引用的 Secret 或 ConfigMap 必须在同一命名空间中存在
3. **敏感数据**：敏感信息应使用 `secret` 来源，避免使用 `static`
4. **与 Properties.Env 区别**：`envs` Trait 支持动态引用，而 `properties.env` 仅支持静态值

EnvFrom 环境变量批量导入

EnvFrom Trait 用于从 ConfigMap 或 Secret 批量导入所有键值对作为环境变量。对应 Kubernetes 中的 EnvFromSource。

字段详解

字段	类型	限制	默认值	说明
type	string	必填	-	来源类型：`secret` 或 `configMap`
sourceName	string	必填	-	ConfigMap 或 Secret 的资源名称

使用示例

从 ConfigMap 批量导入

```
{
  "envFrom": [
    {
      "type": "configMap",
      "sourceName": "app-config"
    }
  ]
}
```

生成结果：

```
envFrom:
  - configMapRef:
      name: app-config
```

假设 ConfigMap `app-config` 内容为：

```
data:
  DATABASE_HOST: mysql.default.svc
  DATABASE_PORT: "3306"
  LOG_FORMAT: json
```

则容器会自动获得三个环境变量：`DATABASE_HOST`、`DATABASE_PORT`、`LOG_FORMAT`。

从 Secret 批量导入

```
{
  "envFrom": [
    {
      "type": "secret",
      "sourceName": "app-secrets"
    }
  ]
}
```

生成结果：

```
envFrom:
  - secretRef:
      name: app-secrets
```

可同时从多个来源导入

```
{
  "envFrom": [
    {
      "type": "configMap",
      "sourceName": "app-config"
    },
    {
      "type": "secret",
      "sourceName": "app-secrets"
    }
  ]
}
```

生成结果：

```
envFrom:
  - configMapRef:
      name: app-config
  - secretRef:
      name: app-secrets
```

注意事项

1. **键名冲突**：如果多个来源有相同的 key，后定义的会覆盖先定义的
2. **资源必须存在**：引用的 ConfigMap/Secret 必须事先存在
3. **与 Envs 配合**：可以同时使用 `envFrom` 批量导入和 `envs` 单独定义
4. **无法选择性导入**：`envFrom` 会导入所有键值对，如需选择性导入请使用 `envs`

Resources 资源限制

Resources Trait 用于定义容器的计算资源限制，包括 CPU、内存和 GPU。对应Kubernetes中的容器资源请求和限制。GPU需要使用异构中间件来实现。

字段详解

字段	类型	限制	默认值	说明
cpu	string	可选	-	CPU 资源，如 `100m`、`0.5`、`2`
memory	string	可选	-	内存资源，如 `128Mi`、`1Gi`、`2G`
gpu	string	可选	-	GPU 数量，如 `1`、`2`

资源单位说明

CPU 单位

格式	说明	示例
小数	CPU 核心数	`0.5` = 半个核心
m	毫核（1核=1000m）	`500m` = 半个核心
整数	CPU 核心数	`2` = 2个核心

内存单位

格式	说明	示例
Ki	Kibibyte (1024)	`512Ki`
Mi	Mebibyte (1024^2)	`256Mi`
Gi	Gibibyte (1024^3)	`2Gi`
K	Kilobyte (1000)	`500K`
M	Megabyte (1000^2)	`256M`
G	Gigabyte (1000^3)	`1G`

1. 系统将指定值同时设置为 `requests` 和 `limits`
2. GPU 资源可以使用Gi(显存)来划分，也支持单张显卡调度。因为存在异构显卡的情况，所以推荐使用Gi作为资源分片，通过调度策略来避免显存碎片化。

使用示例

```
{
  "resources": {
    "cpu": "500m",
    "memory": "512Mi"
  }
}
```

生成结果:

```
resources:
  requests:
    cpu: 500m
    memory: 512Mi
  limits:
    cpu: 500m
    memory: 512Mi
```

注意事项

1. **格式正确**：资源值必须是有效的 Kubernetes 资源量格式
2. **节点容量**：确保集群节点有足够的资源

3. **GPU 支持**：使用 GPU 需要集群安装相应的设备插件
4. **QoS 等级**：同时设置 requests 和 limits 会使 Pod 获得 `Guaranteed` QoS
5. **嵌套使用**：Resources 可以嵌套在 Init 或 Sidecar 中单独配置

在 Kubernetes 中，QoS（Quality of Service）等级决定了当节点资源紧张时，哪些 Pod 会被优先驱逐（evict）。

QoS 等级	条件	驱逐优先级
Guaranteed	所有容器都设置了 requests 和 limits，且两者相等	最低（最后被驱逐）
Burstable	至少一个容器设置了 requests 或 limits，但不满足 Guaranteed 条件	中等
BestEffort	没有设置任何 requests 和 limits	最高（最先被驱逐）

资源保障：Kubernetes 会确保 Pod 获得它请求的全部资源

驱逐保护：当节点内存或 CPU 压力大时，Guaranteed Pod 是最后被驱逐的

稳定性：适合关键业务应用，如数据库、核心服务

Storage 存储

Storage Trait 用于将存储卷挂载到容器中，支持多种存储类型。详细文档请参考

存储类型详解

type	对应 Kubernetes 资源	说明
persistent	PersistentVolumeClaim + VolumeMount	稳定的存储，使用 `PersistentVolumeClaim` 卷用于将持久卷(PersistentVolume)挂载到Pod中。这种方式可以为Pod提供一个**稳定的存储方式，不会因为重启/Pod崩溃而丢失容器内持久化存储的文件**。
ephemeral	emptyDir + VolumeMount	临时存储，对于定义了 `emptyDir` 卷的 Pod，在 Pod 被指派到某节点时此卷会被创建。 就像其名称所表示的那样，`emptyDir` 卷最初是空的。尽管 Pod 中的容器挂载 `emptyDir` 卷的路径可能相同也可能不同，但这些容器都可以读写 `emptyDir` 卷中相同的文件。 当 Pod 因为某些原因被从节点上删除时， `emptyDir` 卷中的数据也会被**永久删除**。
~~hostPath~~	~~hostPath + VolumeMount~~	hostPath 可以让**主机节点文件系统上的文件或目录挂载到Pod中**，比如运行一个需要访问节点级系统组件的容器；让存储在主机系统上的配置文件可以被静态 Pod 以只读方式访问。 **这种方式存在许多安全风险，所以系统中不支持。**
config	ConfigMap + VolumeMount 或 EnvFrom	ConfigMap 对象中存储的数据可以被 `configMap` 类型的卷引用，然后被 Pod 中运行的容器化应用使用。 但是：1.你必须先创建 ConfigMap， 才能使用它。2.ConfigMap 总是以 `readOnly` 的模式挂载。3.某容器以 `subPath` 卷挂载方式使用 ConfigMap 时， 若 ConfigMap 发生变化，此容器将无法接收更新。4.文本数据挂载成文件时采用 UTF-8 字符编码。如果使用其他字符编码形式，可使用 `binaryData` 字段。
secret	Secret + VolumeMount 或 EnvFrom	`secret` 卷用来给 Pod 传递敏感信息，例如密码。你可以将 Secret 存储在 Kubernetes API 服务器上，然后以文件的形式挂载到 Pod 中，无需直接与 Kubernetes 耦合。 但是：1.使用前你必须在 Kubernetes API 中创建 Secret。2.Secret 总是以 `readOnly` 的模式挂载。 3.容器以 `subPath` 卷挂载方式使用 Secret 时，将无法接收 Secret 的更新。

字段详解

字段	类型	限制	默认值	说明
name	string	必填	-	存储卷的唯一名称标识符，用于生成 Kubernetes Volume 名称。必须符合 DNS-1123 子域名规范（小写字母、数字、连字符）。
type	string	必填	-	存储类型，决定底层 Kubernetes 资源类型。支持的值见上列 存储类型 章节。
mountPath	string	必填	`/mnt/<name>`	容器内挂载路径。若未指定，默认为 `/mnt/<volume-name>` 。
subPath	string	可选	`""`	挂载 Volume 内的子路径。用于将同一 Volume 的不同子目录挂载到不同位置。
readOnly	bool	可选	`false`	是否以只读模式挂载。设置为 `true` 时容器无法写入该挂载点。
sourceName	string	可选	`name`	仅用于 ConfigMap/Secret 类型。指定实际 ConfigMap 或 Secret 资源的名称。若为空，则使用 `name` 字段的值。
size	string	可选	`"1Gi"`	PVC 请求的存储容量。格式遵循 Kubernetes 资源量规范（如 `5Gi` 、 `100Mi` ）。

持久化存储专用字段，即下列字段仅在 `type: "persistent"` 时生效：

字段	类型	限制	默认值	说明
tmpCreate	bool	可选	`false`	**是否使用模式控制**： • `true` ：动态创建新的 PVC，PVC 名称格式为 `pvc-<name>-<appID>` • `false` ：引用已存在的 PVC(默认模式)
claimName	string	可选		预留字段，**当前代码未实现**。Volume 引用的 PVC 名称由 `TmpCreate` 决定：为 `true` 时使用生成名称，为 `false` 时使用 `name` 字段。
storageClass	string	可选		指定 PVC 使用的 StorageClass。若为空，使用集群默认 StorageClass。

逻辑详解

1.当 `type="persistent"` 时，系统**始终会创建 PVC 对象**，但根据 `tmpCreate` 字段决定命名和注解：

1.1 tmpCreate = true（模板创建模式）使用volumeClaimTemplates的方式自动创建PVC

```yaml
volumeClaimTemplates:
  - apiVersion: v1
    kind: PersistentVolumeClaim
    metadata:
      creationTimestamp: null
      name: data
```

添加注解：`storage.kubemin.cli/pvc-role: template`;Volume 引用动态生成的 PVC 名称;

适用场景：需要为每个应用实例创建独立存储

1.2 tmpCreate = false（直接创建模式）

直接使用 `name` 字段作为 PVC 名称;创建基础 PVC 对象（无特殊注解）;Volume 引用该 PVC 名称;

适用场景：多个应用共享同一存储，或需要精确控制 PVC 名称

使用示例

动态创建持久化存储

```json
{
  "storage": [
    {
      "type": "persistent",
      "name": "mysql-data",
      "mountPath": "/var/lib/mysql",
      "tmpCreate": true,
      "size": "10Gi",
      "storageClass": "fast-ssd"
    }
  ]
}
```

生成结果：

- PVC 名称：`pvc-mysql-data-<appID>`
- Volume 类型：`PersistentVolumeClaim`
- 挂载路径：`/var/lib/mysql`

直接创建pvc

```
{
  "storage": [
    {
      "type": "persistent", //使用稳定存储类型
      "name": "shared-data",
      "mountPath": "/data", //挂载目录(可选)，如果为空挂载到默认的目录下
      "size": "5Gi", //存储大小
      "TmpCreate": false
    }
  ]
}
```

生成结果：

- 创建名为 `shared-data` 的 PVC（名称不带 appID 后缀）
- Volume 类型：`PersistentVolumeClaim`
- 适用于多应用共享同一 PVC 的场景

临时存储

```
{
  "storage": [
    {
      "type": "ephemeral",
      "name": "cache",
      "mountPath": "/tmp/cache"
    }
  ]
}
```

生成结果：

- Volume 类型：`EmptyDir`
- Pod 删除后数据丢失

ConfigMap 挂载

```
{
  "storage": [
    {
      "type": "config",
      "name": "app-config",
      "sourceName": "my-configmap",
      "mountPath": "/etc/config",
      "readOnly": true
    }
  ]
}
```

生成结果：

- 挂载名为 `my-configmap` 的 ConfigMap
- 默认文件权限：`0644`

备注：

```
0644 是 Unix/Linux 的文件权限模式，分为三组：
0644
│││└── 其他用户 (Others): 4 = r-- (只读)
│└└─── 同组用户 (Group):  4 = r-- (只读)
└└──── 文件所有者 (Owner): 6 = rw- (读写)
└──── 特殊位 (无)
```

Secret 挂载

```
{
  "storage": [
    {
      "type": "secret",
      "name": "certs",
      "sourceName": "tls-secret",
      "mountPath": "/etc/ssl/certs",
      "readOnly": true
    }
  ]
}
```

生成结果：

- 挂载名为 `tls-secret` 的 Secret
- 默认文件权限：`0644`

SubPath 挂载

```
{
  "storage": [
    {
      "type": "persistent",
      "name": "data",
      "mountPath": "/var/lib/mysql",
      "subPath": "mysql",
      "TmpCreate": true,
      "size": "5Gi"
    }
  ]
}
```

生成结果：

- 仅挂载 PVC 中的 `mysql` 子目录到 `/var/lib/mysql`
- 同一 PVC 可被多个容器以不同 subPath 挂载

多容器共享存储

```
{
  "storage": [
    {
      "type": "persistent",
      "name": "shared-data",
      "mountPath": "/data",
      "TmpCreate": true,
      "size": "5Gi"
    }
  ],
  "init": [
    {
      "name": "init-data",
      "properties": {
        "image": "busybox:latest"
      },
      "traits": {
        "storage": [
          {
            "type": "ephemeral",
            "name": "shared-data",
            "mountPath": "/init-data"
          }
        ]
      }
    }
  ],
  "sidecar": [
    {
      "name": "backup",
      "image": "backup-agent:v1",
      "traits": {
        "storage": [
          {
            "type": "ephemeral",
            "name": "shared-data",
            "mountPath": "/backup-source",
            "readOnly": true
          }
        ]
      }
    }
  ]
```

```
}
```

说明：

- 主容器声明 `type: persistent` 并设置 `TmpCreate: true`，系统创建 PVC
- Init/Sidecar 使用 `type: ephemeral` + 相同的 `name` 声明挂载意图
- 系统会自动去重：同名 Volume 只创建一次，各容器的 VolumeMount 独立配置
- 各容器可使用不同的 `mountPath` 和 `readOnly` 设置

> **注意**：如果 Init/Sidecar 也使用 `type: persistent`，由于 `TmpCreate` 默认为 `false`，会尝试创建额外的同名 PVC，导致资源冲突或创建多余对象。

注意事项

1. **名称规范**：`name` 字段必须符合 Kubernetes DNS-1123 子域名规范，系统会自动进行规范化处理（小写化、移除非法字符）
2. **PVC 生命周期**：动态创建的 PVC（`TmpCreate: true`）生命周期与应用绑定，删除应用时需考虑 PVC 清理策略
3. **StorageClass**：确保指定的 StorageClass 在目标集群中存在，否则 PVC 将处于 Pending 状态
4. **SubPath 与空目录**：使用 `subPath` 时，如果子目录不存在，Kubernetes 不会自动创建，可能导致挂载失败
5. **只读模式**：对于 ConfigMap 和 Secret 类型，建议始终设置 `readOnly: true`
6. **Volume 去重**：当多个容器（主容器、Init、Sidecar）声明同名存储时，系统只创建一个 Volume，但各自的 VolumeMount 独立配置
7. **多容器共享存储**：Init/Sidecar 容器共享主容器的 PVC 时，应使用 `type: ephemeral` 声明挂载意图，避免重复声明 `type: persistent` 导致创建多余的 PVC 对象
8. **claimName 字段**：该字段当前未实现，请勿依赖此字段指定已存在的 PVC 名称

Probes 健康探针

Probes Trait 用于定义容器的健康检查探针，Kubernetes 根据探针结果决定容器的运行状态。

- **LivenessProbe**：存活探针，失败时重启容器

- **ReadinessProbe**：就绪探针，失败时从 Service 摘除
- **StartupProbe**：启动探针，成功前不执行其他探针

字段详解

字段	类型	限制	默认值	说明
type	string	必填	-	探针类型：`liveness` / `readiness` / `startup`
initialDelaySeconds	int32	可选	0	容器启动后延迟多少秒开始探测
periodSeconds	int32	可选	10	探测间隔秒数
timeoutSeconds	int32	可选	1	探测超时秒数
failureThreshold	int32	可选	3	连续失败多少次视为失败
successThreshold	int32	可选	1	连续成功多少次视为成功

探测方式（三选一）

字段	类型	说明
exec	object	执行命令探测
exec.command	[]string	要执行的命令
httpGet	object	HTTP GET 探测
httpGet.path	string	HTTP 请求路径
httpGet.port	int	HTTP 请求端口
httpGet.host	string	主机名（可选）
httpGet.scheme	string	HTTP 或 HTTPS（可选）
tcpSocket	object	TCP 端口探测
tcpSocket.port	int	TCP 端口
tcpSocket.host	string	主机名（可选）

探针类型说明

类型	用途	失败后果
liveness	检测容器是否存活	重启容器
readiness	检测容器是否就绪	从 Service 端点移除
startup	检测应用是否启动完成	阻止 liveness/readiness 探测

使用示例

HTTP 健康检查

```
{
  "probes": [
    {
      "type": "liveness",
      "httpGet": {
        "path": "/healthz",
        "port": 8080
      },
      "initialDelaySeconds": 30,
      "periodSeconds": 10,
      "timeoutSeconds": 5,
      "failureThreshold": 3
    },
    {
      "type": "readiness",
      "httpGet": {
        "path": "/ready",
        "port": 8080
      },
      "initialDelaySeconds": 5,
      "periodSeconds": 5
    }
  ]
}
```

生成结果：

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 3
readinessProbe:
  httpGet:
    path: /ready
```

```
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
```

TCP 端口检查

```
{
  "probes": [
    {
      "type": "liveness",
      "tcpSocket": {
        "port": 3306
      },
      "initialDelaySeconds": 30,
      "periodSeconds": 15
    },
    {
      "type": "readiness",
      "tcpSocket": {
        "port": 3306
      },
      "initialDelaySeconds": 5,
      "periodSeconds": 10
    }
  ]
}
```

生成结果：

```
livenessProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 30
  periodSeconds: 15
readinessProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 5
  periodSeconds: 10
```

命令执行检查

```
{
  "probes": [
    {
      "type": "liveness",
      "exec": {
        "command": ["cat", "/tmp/healthy"]
      },
      "initialDelaySeconds": 5,
      "periodSeconds": 5
    }
  ]
}
```

生成结果：

```
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

慢启动应用配置

```
{
  "probes": [
    {
      "type": "startup",
      "httpGet": {
        "path": "/healthz",
        "port": 8080
      },
      "initialDelaySeconds": 0,
      "periodSeconds": 10,
      "failureThreshold": 30
```

```
    },
    {
      "type": "liveness",
      "httpGet": {
        "path": "/healthz",
        "port": 8080
      },
      "periodSeconds": 10
    },
    {
      "type": "readiness",
      "httpGet": {
        "path": "/ready",
        "port": 8080
      },
      "periodSeconds": 5
    }
  ]
}
```

生成结果：

```
startupProbe:
  httpGet:
    path: /healthz
    port: 8080
  periodSeconds: 10
  failureThreshold: 30        # 允许最多 300s（30*10s）完成启动
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  periodSeconds: 10           # startup 成功后才开始执行
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  periodSeconds: 5            # startup 成功后才开始执行
```

注意事项

1. **单一探测方式**：每个探针只能指定 `exec` 、 `httpGet` 、 `tcpSocket` 中的一个

2. **每种类型唯一**：同一组件的每种探针类型只能定义一个
3. **启动探针优先**：配置 `startup` 探针时，其他探针在启动成功前不会执行
4. **合理配置阈值**：根据应用特性合理设置 `initialDelaySeconds` 和 `failureThreshold`
5. **端点可用性**：确保探测端点在容器启动后尽快可用

Init 初始化容器

Init Trait 用于定义在主容器启动前运行的初始化容器。初始化容器按顺序执行，每个必须成功完成后才会启动下一个。对应 Kubernetes 中的 Pod 的初始化容器。

字段详解

字段	类型	限制	默认值	说明
name	string	必填	-	初始化容器名称，如为空则自动生成
properties	properties结构体	必填	-	容器属性配置
properties.image	string	必填	-	容器镜像
properties.command	[]string	可选	-	容器启动命令
properties.env	map[string]string	可选	-	环境变量键值对
traits	traits	可选	-	嵌套 Traits，支持 storage、envs、envFrom

1. 系统为每个 Init Trait 创建一个 InitContainer
2. 如果 `name` 为空，自动生成格式为 `<组件名>-init-<随机字符>` 的名称
3. 支持嵌套 Traits（但不能嵌套 init 自身，防止无限循环）
4. 嵌套的 storage/envs/resources 等 Traits 会应用到该初始化容器

使用示例

基础初始化容器

```
{
  "init": [
```

```
    {
      "name": "init-permissions",
      "properties": {
        "image": "busybox:latest",
        "command": ["sh", "-c", "chmod -R 755 /data"],
        "env": {
          "DATA_DIR": "/data"
        }
      }
    }
  ]
}
```

生成结果：

- InitContainer 名称：`init-permissions`
- 镜像：`busybox:latest`
- 启动命令：`sh -c "chmod -R 755 /data"`
- 环境变量：`DATA_DIR=/data`
- ImagePullPolicy：`IfNotPresent`

带存储挂载的初始化容器

```
{
  "init": [
    {
      "name": "init-data",
      "properties": {
        "image": "busybox:latest",
        "command": ["sh", "-c", "cp /config/* /app/config/"]
      },
      "traits": {
        "storage": [
          {
            "type": "config",
            "name": "app-config",
            "sourceName": "my-configmap",
            "mountPath": "/config",
            "readOnly": true
          },
```

```
            {
              "type": "ephemeral",
              "name": "shared-data",
              "mountPath": "/app/config"
            }
          ]
        }
      }
    ]
}
```

生成结果：

- InitContainer 名称：`init-data`
- 镜像：`busybox:latest`
- Volume 1：ConfigMap `my-configmap` 挂载到 `/config`（只读）
- Volume 2：EmptyDir `shared-data` 挂载到 `/app/config`
- 用途：将 ConfigMap 内容复制到共享存储供主容器使用

注意事项

1. **镜像必填**：`properties.image` 是必填字段，否则会报错
2. **顺序执行**：多个 Init 容器按数组顺序执行
3. **嵌套限制**：Init 容器的嵌套 Traits 不能包含 `init` 自身
4. **共享存储**：Init 容器可以与主容器共享 Volume，用于数据准备
5. **嵌套 RBAC**：Trait只排除 Init,Sidecar，但是在Init或Sidecar中配置RBAC是无意义的，因为Init与主容器共享ServiceAccount。正确做法应该是在在组件根级别配置 RBAC。

Sidecar 边车容器

Sidecar Trait 用于定义与主容器并行运行的辅助容器，常用于日志收集、代理、监控等场景。对应 Kubernetes 中的边车容器。

字段详解

字段	类型	限制	默认值	说明
name	string	必填	-	边车容器名称，如为空则自动生成
image	string	必填	-	容器镜像
command	[]string	可选	-	容器启动命令
args	[]string	可选	-	命令参数
env	map[string]string	可选	-	环境变量键值对
traits	traits	可选	-	嵌套 Traits，支持 storage、envs、envFrom、probes、resources

逻辑详解

1. 系统为每个 Sidecar Trait 创建一个额外的 Container
2. 如果 `name` 为空，自动生成格式为 `<组件名>-sidecar-<随机字符>` 的名称
3. 支持嵌套 Traits（但不能嵌套 sidecar 和 init）
4. 边车容器可以有独立的健康探针和资源限制

使用示例

日志收集边车

```
{
  "sidecar": [
    {
      "name": "fluentd",
      "image": "fluent/fluentd:v1.14",
      "env": {
        "FLUENTD_CONF": "fluent.conf"
      },
      "traits": {
        "storage": [
```

```
          {
            "type": "ephemeral",
            "name": "app-logs",
            "mountPath": "/var/log/app"
          }
        ],
        "resources": {
          "cpu": "100m",
          "memory": "128Mi"
        }
      }
    }
  ]
}
```

生成结果：

- Container 名称：`fluentd`
- 镜像：`fluent/fluentd:v1.14`
- 环境变量：`FLUENTD_CONF=fluent.conf`
- Volume：EmptyDir `app-logs` 挂载到 `/var/log/app`
- 资源限制：CPU `100m`，内存 `128Mi`
- 用途：收集主容器写入 `/var/log/app` 的日志

Envoy 代理边车

```
{
  "sidecar": [
    {
      "name": "envoy-proxy",
      "image": "envoyproxy/envoy:v1.25.0",
      "command": ["envoy"],
      "args": ["-c", "/etc/envoy/envoy.yaml"],
      "traits": {
        "storage": [
          {
            "type": "config",
            "name": "envoy-config",
            "sourceName": "envoy-configmap",
            "mountPath": "/etc/envoy"
```

```
          }
        ],
        "probes": [
          {
            "type": "readiness",
            "httpGet": {
              "path": "/ready",
              "port": 9901
            },
            "initialDelaySeconds": 5,
            "periodSeconds": 10
          }
        ],
        "resources": {
          "cpu": "200m",
          "memory": "256Mi"
        }
      }
    }
  ]
}
```

生成结果：

- Container 名称：`envoy-proxy`
- 镜像：`envoyproxy/envoy:v1.25.0`
- 启动命令：`envoy -c /etc/envoy/envoy.yaml`
- Volume：ConfigMap `envoy-configmap` 挂载到 `/etc/envoy`
- ReadinessProbe：HTTP GET `/ready:9901`，延迟 5s，间隔 10s
- 资源限制：CPU `200m`，内存 `256Mi`
- 用途：为主容器提供代理功能

监控指标导出边车

```
{
  "sidecar": [
    {
      "name": "prometheus-exporter",
      "image": "prom/node-exporter:v1.5.0",
```

```
      "args": ["--path.procfs=/host/proc", "--path.sysfs=/host/sys"],
      "traits": {
        "probes": [
          {
            "type": "liveness",
            "httpGet": {
              "path": "/metrics",
              "port": 9100
            },
            "periodSeconds": 30
          }
        ]
      }
    }
  ]
}
```

生成结果：

- Container 名称：`prometheus-exporter`
- 镜像：`prom/node-exporter:v1.5.0`
- 参数：`--path.procfs=/host/proc --path.sysfs=/host/sys`
- LivenessProbe：HTTP GET `/metrics:9100`，间隔 30s
- 用途：为 Prometheus 提供监控指标

注意事项

1. **镜像必填**：`image` 是必填字段
2. **禁止嵌套边车**：Sidecar 的嵌套 Traits 不能包含 `sidecar` 或 `Init`
3. **资源规划**：边车容器会占用 Pod 资源，需要合理规划
4. **共享网络**：边车与主容器共享同一网络命名空间，可通过 localhost 通信

RBAC 权限控制

RBAC Trait 用于创建 Kubernetes RBAC 资源，为组件配置细粒度的权限控制。对应 Kubernetes 中的**ServiceAccount**：服务账户;**Role**：命名空间级角色;**ClusterRole**：集群级角色;**RoleBinding**：命名空间级角色绑定;**ClusterRoleBinding**：集群级角色绑定

字段详解

字段	类型	限制	默认值	说明
serviceAccount	string	可选	`<组件名>-sa`	ServiceAccount 名称
namespace	string	可选	组件命名空间	资源所在命名空间
clusterScope	bool	可选	`false`	是否创建集群级角色
roleName	string	可选	`<sa名>-role`	Role/ClusterRole 名称
bindingName	string	可选	`<sa名>-binding`	RoleBinding/ClusterRoleBinding 名称
serviceAccountLabels	map[string]string	可选	-	ServiceAccount 标签
serviceAccountAnnotations	map[string]string	可选	-	ServiceAccount 注解
roleLabels	map[string]string	可选	-	Role 标签
bindingLabels	map[string]string	可选	-	RoleBinding 标签
automountServiceAccountToken	*bool	可选	-	是否自动挂载 SA Token
rules	[]object	必填	-	权限规则列表

Rules 权限规则

字段	类型	限制	默认值	说明
apiGroups	[]string	可选	-	API 组，`""` 表示核心组
resources	[]string	可选	-	资源类型
resourceNames	[]string	可选	-	具体资源名称
nonResourceURLs	[]string	可选	-	非资源 URL
verbs	[]string	必填	-	操作动词

常用 API Groups

API Group	包含资源
`""` (core)	pods, services, configmaps, secrets, persistentvolumeclaims
`apps`	deployments, daemonsets, statefulsets, replicasets
`batch`	jobs, cronjobs
`networking.k8s.io`	ingresses, networkpolicies
`rbac.authorization.k8s.io`	roles, rolebindings, clusterroles

常用 Verbs

Verb	说明
get	读取单个资源
list	列出资源
watch	监听资源变化
create	创建资源
update	更新资源
patch	部分更新资源
delete	删除单个资源
deletecollection	删除资源集合
`*`	所有操作

使用示例

只读 Pod 权限

```
{
  "rbac": [
    {
      "serviceAccount": "pod-reader",
      "rules": [
        {
          "apiGroups": [""],
          "resources": ["pods"],
          "verbs": ["get", "list", "watch"]
        }
      ]
    }
  ]
}
```

生成结果：

- ServiceAccount：`pod-reader`
- Role：`pod-reader-role`（命名空间级）
- RoleBinding：`pod-reader-binding`

- 权限：对 pods 资源的 get/list/watch 操作

```
# ServiceAccount
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-reader

# Role
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader-role
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]

# RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
subjects:
  - kind: ServiceAccount
    name: pod-reader
roleRef:
  kind: Role
  name: pod-reader-role
  apiGroup: rbac.authorization.k8s.io
```

ConfigMap 和 Secret 读取权限

```
{
  "rbac": [
    {
      "serviceAccount": "config-reader",
      "rules": [
        {
          "apiGroups": [""],
```

```
          "resources": ["configmaps", "secrets"],
          "verbs": ["get", "list"]
        }
      ]
    }
  ]
}
```

Deployment 管理权限

```
{
  "rbac": [
    {
      "serviceAccount": "deployment-manager",
      "rules": [
        {
          "apiGroups": ["apps"],
          "resources": ["deployments"],
          "verbs": ["get", "list", "watch", "create", "update", "patch",
"delete"]
        },
        {
          "apiGroups": [""],
          "resources": ["pods"],
          "verbs": ["get", "list", "watch"]
        }
      ]
    }
  ]
}
```

集群级权限

```
{
  "rbac": [
    {
      "serviceAccount": "cluster-admin-sa",
      "clusterScope": true,
      "rules": [
        {
          "apiGroups": [""],
```

```
          "resources": ["nodes"],
          "verbs": ["get", "list", "watch"]
        },
        {
          "apiGroups": [""],
          "resources": ["namespaces"],
          "verbs": ["get", "list"]
        }
      ]
    }
  ]
}
```

生成结果：

- ServiceAccount：`cluster-admin-sa`
- ClusterRole：`cluster-admin-sa-role`（集群级）
- ClusterRoleBinding：`cluster-admin-sa-binding`
- 权限：对 nodes 的 get/list/watch，对 namespaces 的 get/list

```yaml
# ClusterRole（注意：不是 Role）
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-admin-sa-role
rules:
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["namespaces"]
    verbs: ["get", "list"]

# ClusterRoleBinding（注意：不是 RoleBinding）
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-sa-binding
subjects:
  - kind: ServiceAccount
    name: cluster-admin-sa
    namespace: <组件命名空间>
```

```
roleRef:
  kind: ClusterRole
  name: cluster-admin-sa-role
  apiGroup: rbac.authorization.k8s.io
```

特定资源名称权限

```
{
  "rbac": [
    {
      "serviceAccount": "specific-config-reader",
      "rules": [
        {
          "apiGroups": [""],
          "resources": ["configmaps"],
          "resourceNames": ["app-config", "feature-flags"],
          "verbs": ["get"]
        }
      ]
    }
  ]
}
```

生成结果：

- 权限范围：仅能读取名为 `app-config` 和 `feature-flags` 的 ConfigMap
- 其他 ConfigMap 无法访问

```
rules:
  - apiGroups: [""]
    resources: ["configmaps"]
    resourceNames: ["app-config", "feature-flags"]
    verbs: ["get"]
```

带标签和注解的完整配置

```
{
  "rbac": [
    {
      "serviceAccount": "backend-sa",
      "roleName": "backend-role",
```

```
      "bindingName": "backend-binding",
      "automountServiceAccountToken": false,
      "serviceAccountLabels": {
        "app": "backend",
        "env": "production"
      },
      "serviceAccountAnnotations": {
        "description": "Backend service account"
      },
      "roleLabels": {
        "app": "backend"
      },
      "bindingLabels": {
        "app": "backend"
      },
      "rules": [
        {
          "apiGroups": [""],
          "resources": ["pods", "services"],
          "verbs": ["get", "list", "watch"]
        },
        {
          "apiGroups": [""],
          "resources": ["configmaps"],
          "verbs": ["get"]
        }
      ]
    }
  ]
}
```

生成结果：

- ServiceAccount 名称：`backend-sa`（自定义）
- Role 名称：`backend-role`（自定义）
- RoleBinding 名称：`backend-binding`（自定义）
- automountServiceAccountToken：`false`（提高安全性）
- 所有资源带自定义标签

```
# ServiceAccount
apiVersion: v1
kind: ServiceAccount
```

```yaml
metadata:
  name: backend-sa
  labels:
    app: backend
    env: production
  annotations:
    description: Backend service account
automountServiceAccountToken: false

# Role
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: backend-role
  labels:
    app: backend
rules:
  - apiGroups: [""]
    resources: ["pods", "services"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["configmaps"]
    verbs: ["get"]

# RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: backend-binding
  labels:
    app: backend
# ...
```

多策略配置

```
{
  "rbac": [
    {
      "serviceAccount": "primary-sa",
      "rules": [
        {
          "apiGroups": [""],
```

```
          "resources": ["pods"],
          "verbs": ["get", "list"]
        }
      ]
    },
    {
      "serviceAccount": "secondary-sa",
      "clusterScope": true,
      "rules": [
        {
          "apiGroups": [""],
          "resources": ["nodes"],
          "verbs": ["get", "list"]
        }
      ]
    }
  ]
}
```

生成结果：

创建 6 个 Kubernetes 资源：

策略	ServiceAccount	Role/ClusterRole	Binding
第 1 个	`primary-sa`	Role `primary-sa-role`	RoleBinding `primary-sa-binding`
第 2 个	`secondary-sa`	ClusterRole `secondary-sa-role`	ClusterRoleBinding `secondary-sa-binding`

- Pod 绑定的 ServiceAccount：`primary-sa`（第一个策略的 SA）
- 第一个策略创建命名空间级 Role
- 第二个策略创建集群级 ClusterRole

注意事项

1. **Verbs 必填**：每个 rule 必须指定至少一个 verb
2. **最小权限原则**：只授予必要的权限
3. **命名空间隔离**：非 `clusterScope` 的角色只在指定命名空间生效
4. **ServiceAccount 绑定**：第一个 RBAC 策略的 ServiceAccount 会被设置到 Pod
5. **automountServiceAccountToken**：设置为 `false` 可提高安全性
6. **多策略处理**：配置多个 RBAC 策略时，各自创建独立的资源集合

Ingress

Ingress Trait 用于创建 Kubernetes Ingress 资源，将外部 HTTP/HTTPS 流量路由到服务。对应 Kubernetes 中的 Ingress，需要安装插件。

> 不应该直接使用Ingress作为特征的一部分，应该抽象一个对外的Sevice配置，这样在对集群网关进行置换后就不需要进行烦杂的操作。而且Ingress在2026年6月将停止维护，暂时先实现。

字段详解

字段	类型	限制	默认值	说明
name	string	可选	`<组件名>-ingress`	Ingress 资源名称
namespace	string	可选	组件命名空间	Ingress 所在命名空间
hosts	[]string	可选	-	全局主机名列表
label	map[string]string	可选	-	标签
annotations	map[string]string	可选	-	注解
ingressClassName	string	可选	-	Ingress Class 名称
defaultPathType	string	可选	`Prefix`	默认路径匹配类型
tls	[]object	可选	-	TLS 配置
routes	[]object	必填	-	路由规则

TLS 配置

字段	类型	说明
secretName	string	TLS 证书 Secret 名称
hosts	[]string	该证书适用的主机列表

Routes 路由规则

字段	类型	限制	默认值	说明
path	string	可选	`/`	URL 路径
pathType	string	可选	`Prefix`	路径类型：Prefix/Exact/ImplementationSpecific
host	string	可选	-	路由级主机名（覆盖全局 hosts）
backend	object	必填	-	后端服务配置
backend.serviceName	string	必填	-	服务名称
backend.servicePort	int32	可选	80	服务端口
rewrite	object	可选	-	路径重写配置

Rewrite 重写配置

字段	类型	说明
type	string	重写类型：replace/regexReplace/prefix
match	string	匹配模式
replacement	string	替换值

使用示例

基础路由

```
{
  "ingress": [
    {
      "name": "my-app-ingress",
      "ingressClassName": "nginx",
      "routes": [
        {
          "path": "/",
          "backend": {
            "serviceName": "my-app-service",
            "servicePort": 80
          }
        }
      ]
    }
  ]
}
```

生成结果：

- Ingress 名称：`my-app-ingress`
- Ingress Class：`nginx`
- 路由规则：`/` → `my-app-service:80`
- 路径类型：`Prefix`（默认）

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
```

```yaml
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-app-service
                port:
                  number: 80
```

带主机名的路由

```json
{
  "ingress": [
    {
      "name": "api-ingress",
      "ingressClassName": "nginx",
      "hosts": ["api.example.com"],
      "routes": [
        {
          "path": "/v1",
          "backend": {
            "serviceName": "api-v1",
            "servicePort": 8080
          }
        },
        {
          "path": "/v2",
          "backend": {
            "serviceName": "api-v2",
            "servicePort": 8080
          }
        }
      ]
    }
  ]
}
```

TLS 配置

```
{
  "ingress": [
    {
      "name": "secure-ingress",
      "ingressClassName": "nginx",
      "tls": [
        {
          "secretName": "tls-secret",
          "hosts": ["secure.example.com"]
        }
      ],
      "routes": [
        {
          "host": "secure.example.com",
          "path": "/",
          "backend": {
            "serviceName": "secure-app",
            "servicePort": 443
          }
        }
      ]
    }
  ]
}
```

生成结果：

- Ingress 名称：`secure-ingress`
- TLS 证书：Secret `tls-secret` 用于 `secure.example.com`
- 路由：`https://secure.example.com/` → `secure-app:443`

```
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - secure.example.com
      secretName: tls-secret
  rules:
    - host: secure.example.com
      http:
```

```
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: secure-app
                port:
                  number: 443
```

路径重写

```
{
  "ingress": [
    {
      "name": "rewrite-ingress",
      "ingressClassName": "nginx",
      "annotations": {
        "nginx.ingress.kubernetes.io/use-regex": "true"
      },
      "routes": [
        {
          "path": "/api(/.*)",
          "pathType": "ImplementationSpecific",
          "host": "app.example.com",
          "backend": {
            "serviceName": "backend-service",
            "servicePort": 8080
          },
          "rewrite": {
            "type": "regexReplace",
            "replacement": "$1"
          }
        }
      ]
    }
  ]
}
```

多主机多路由

```
{
```

```
  "ingress": [
    {
      "name": "multi-host-ingress",
      "ingressClassName": "nginx",
      "label": {
        "app": "multi-tenant"
      },
      "tls": [
        {
          "secretName": "wildcard-tls",
          "hosts": ["*.example.com"]
        }
      ],
      "routes": [
        {
          "host": "app1.example.com",
          "path": "/",
          "backend": {
            "serviceName": "app1-service",
            "servicePort": 80
          }
        },
        {
          "host": "app2.example.com",
          "path": "/",
          "backend": {
            "serviceName": "app2-service",
            "servicePort": 80
          }
        },
        {
          "host": "api.example.com",
          "path": "/users",
          "backend": {
            "serviceName": "user-service",
            "servicePort": 8080
          }
        },
        {
          "host": "api.example.com",
          "path": "/orders",
          "backend": {
            "serviceName": "order-service",
```

```
            "servicePort": 8080
          }
        }
      ]
    }
  ]
}
```

注意事项

1. **路由必填**：`routes` 至少包含一个路由规则
2. **Ingress Controller**：需要集群中安装相应的 Ingress Controller（如 nginx-ingress）
3. **TLS 证书**：TLS 配置中引用的 Secret 必须存在且包含有效证书
4. **路径类型**：
   - `Prefix`：前缀匹配（默认）
   - `Exact`：精确匹配
   - `ImplementationSpecific`：由 Ingress Controller 决定
5. **注解差异**：不同 Ingress Controller 的注解语法可能不同

> 应该将Ingress和RBAC,SelectNode这种配置统一管理在数据库中，这样在进行配置的时候，可以通过选择不同的场景来适配不同的配置，可以简化数据结构，简化用户的心智。

Traits 处理顺序

系统按以下顺序处理 Traits：

1. Storage（存储）
2. EnvFrom（批量环境变量）
3. Envs（环境变量）
4. Resources（资源限制）
5. Probes（健康探针）
6. RBAC（权限控制）
7. Init（初始化容器）
8. Sidecar（边车容器）
9. Ingress（入口流量）

嵌套 Traits 支持矩阵

父 Trait	可嵌套的 Traits	排除的 Traits
Init	storage, envs, envFrom, resources,probes,rbac、ingress	init, sidecar
Sidecar	storage, envs, envFrom, probes, resources,rbac、ingress	init, sidecar