Edge Platform 技术白皮书

云原生边缘计算平台

Edge Platform Team

2025年1月

目录

1 技术白皮书：Edge Platform

基于Kubernetes的云原生边缘计算管理平台

版本: v1.0 发布日期: 2025年1月 维护者: Edge Platform Team

1.1 产品定位

Edge Platform 是基于 Kubernetes 构建的企业级边缘计算管理平台，为企业提供统一的多集群、多租户管理能力。平台采用云原生架构，所有功能通过 CRD 实现，完全兼容 Kubernetes 生态。

1.2 核心技术创新

1.2.1 1. 5层Scope权限模型

解决的问题：企业级多租户权限管理

传统的 Kubernetes RBAC 仅支持 Cluster 和 Namespace 两级，难以满足企业复杂组织架构的权限管理需求。

技术方案： - Platform → Cluster → Workspace/NodeGroup → Namespace/Node → Resource - 双视图设计：Workspace（应用视图）和 NodeGroup（资源视图）并行 - 权限自动级联查找机制

技术实现： - RoleTemplate CRD：定义可复用的权限模板 - IAMRole CRD：基于模板创建具体权限角色 - IAMRoleBinding CRD：管理用户角色绑定 - Controller 自动转换为 K8s 原生 RBAC

1.2.2 2. vcluster虚拟集群管理

解决的问题：边缘环境资源优化

边缘环境资源有限，为每个边缘站点部署独立的物理 Kubernetes 集群资源开销巨大。

技术方案： - 基于 vcluster 实现 Namespace 级别的虚拟集群 - 每个 vcluster 仅占用 200MB 内存 - 共享宿主集群的 kube-apiserver 和控制平面

自动化能力： - 通过 Cluster CR 声明式创建虚拟集群 - 自动安装边缘运行时（KubeEdge/OpenYurt） - 支持 k3s、k0s、k8s 等多种发行版

1.2.3 3. 应用商店三层架构

解决的问题：应用生命周期管理

边缘场景下，应用版本管理、批量部署、回滚等操作复杂，缺乏统一的管理机制。

技术方案： - Application：应用元数据和基本配置 - ApplicationVersion：应用版本和具体实现 - ApplicationDeployment：部署拓扑和运行状态

扩展机制： - Plugin 接口：支持扩展新的应用类型 - 内置类型：Workload、Helm Chart - 可扩展类型：AI Model、函数计算等

1.2.4 4. 声明式组件管理

解决的问题：组件安装和升级的复杂性

传统组件安装依赖 Shell 脚本和手动 Helm 操作，存在幂等性问题，难以管理。

技术方案： - Component CR：声明组件的期望状态 - ChartMuseum：每集群独立的 Chart 仓库 - Controller：自动执行 Helm 安装/升级

关键特性： - 幂等操作：重复执行结果一致 - Chart 预置：镜像内置常用组件 - 离线部署：零跨集群依赖 - GitOps 友好：声明式配置管理

1.3 应用场景

1.3.1 制造业工业互联网

场景描述： 某大型制造企业在全国拥有100个工厂，每个工厂都需要独立的边缘计算环境，用于设备监控、质量检测和生产调度。

Edge Platform解决方案： - 通过vcluster虚拟化，每个工厂仅需200MB内存即可获得独立K8s环境 - 5层Scope权限模型完美匹配集团→工厂→车间→产线→设备的组织架构 - 应用商店实现AI质检应用的统一分发和版本管理

核心商业价值： - **硬件成本降低87%**：从每工厂独立服务器到虚拟化共享 - 应用部署效率**提升10倍**：从3天缩短到8小时 - 设备**故障率降低30%**：通过实时监控和预测性维护

1.3.2 零售业智慧门店

场景描述： 某连锁零售企业拥有500家门店，需要统一部署收银、库存管理、客流分析等应用，并支持快速业务迭代。

Edge Platform解决方案： - 每个门店部署轻量级vcluster，统一管理所有应用 - 应用商店支持一键将新应用部署到全部门店 - 声明式部署确保配置一致性，避免人为错误

核心商业价值： - **新店开业IT准备时间从3天缩短到4小时** - **应用发布频率从月度提升到周度** - **运维成本降低70%**：集中管理替代上门维护

1.3.3 能源行业智能电网

场景描述： 某电力公司管理2000个变电站，多数位于偏远地区，网络连接不稳定，要求完全离线运行能力。

Edge Platform解决方案： - 每变电站独立ChartMuseum，确保离线环境下的组件安装 - 幂等操作设计，网络恢复后自动同步状态 - 完整的审计日志满足等保2.0合规要求

核心商业价值： - **系统可用性从99.5%提升到99.99%** - **故障恢复时间从小时级缩短到分钟级** - **合规审计成本降低80%**

1.4 技术架构特点

1.4.1 Kubernetes Native

- 100% CRD 架构，充分利用 Kubernetes 生态
- 完全兼容 kubectl、Helm 等原生工具
- 无额外学习成本，运维团队可直接使用

1.4.2 声明式设计

- 用户声明期望状态，Controller 自动执行
- 幂等操作，故障自动恢复
- GitOps 友好，支持 CI/CD 集成

1.4.3 微服务架构

- APIServer、Controller、Console 组件分离
- 独立扩展，高可用部署
- 标准化接口，易于集成

1.4.4 插件化扩展

- Provisioner Plugin 扩展应用类型
- Scope Pattern 扩展权限模型
- 开放接口，支持客户定制

1.5 核心能力

1.5.1 权限管理

- 5层 Scope 权限模型
- 前后端权限一体化
- 支持万级用户规模

1.5.2 多集群管理

- vcluster 虚拟化技术
- 自动化边缘运行时集成
- 统一监控和日志

1.5.3 应用生命周期

- 三层解耦架构
- Plugin 可扩展机制
- 多拓扑部署能力

1.5.4 组件管理

- 声明式安装升级
- 完全离线部署
- GitOps 工作流

ewpage

2 第一章 产品介绍

3 1.1 产品定位

Edge Platform 是基于 Kubernetes 构建的云原生边缘计算管理平台，为企业提供统一的多集群、多租户管理能力。平台采用 100% Kubernetes Native 架构，所有功能通过 CRD 实现，完全兼容 Kubernetes 生态。

3.1 核心能力概览

3.1.1 统一权限管理

- **5层 Scope 权限模型**：完美映射企业从集团到设备的组织架构
- **权限自**动级联：告别传统 K8s RBAC 的级联困难
- **双**视图设计：Workspace（应用视图）和 NodeGroup（资源视图）并行

3.1.2 多集群管理

- **虚拟集群自动化**：基于 vcluster 实现资源高效的虚拟集群
- 边缘**运行时集成**：自动安装 KubeEdge/OpenYurt
- **声明式管理**：通过 Cluster CR 声明期望状态

3.1.3 应用生命周期管理

- **三层架构**：Application → ApplicationVersion → ApplicationDeployment
- **多类型支持**：Workload、Helm Chart、AI Model
- **多拓扑部署**：支持跨节点组、跨集群的统一分发

3.1.4 组件管理

- **声明式安装**：Component CR 驱动，自动执行 Helm 安装
- **独立 Chart 仓库**：每集群独立 ChartMuseum，零跨集群依赖
- 离线**部署**：Chart 预置镜像，支持完全离线环境

下一章节：1.2 核心概念

ewpage

4 1.2 核心概念

4.1 Scope权限模型

Scope是Edge Platform的核心概念，用于定义权限的作用域和资源隔离边界。Scope模型采用5层结构，从平台全局到具体资源，提供细粒度的权限控制。

4.1.1 5层权限层次

Platform（平台级）： 最高权限级别，具有全局管理权限，包括管理所有集群、用户、角色等。通常授予平台管理员。

Cluster（集群级）： 针对特定集群的权限，可以管理该集群内的所有资源和用户。适用于集群管理员。

Workspace/NodeGroup（工作空间/节点组）： 这是并行的两种视图： - Workspace：面向应用开发团队，以应用为中心的组织方式 - NodeGroup：面向基础设施团队，以资源为中心的组织方式

Namespace/Node（命名空间/节点）： 更细粒度的资源组织单位，Namespace用于K8s工作负载隔离，Node用于物理节点管理。

Resource（资源级）： 最细粒度的权限控制，可以精确到Pod、Service等具体资源。

4.1.2 权限级联机制

当用户在某个Scope级别没有权限时，系统会自动向更高级别的Scope查找权限，直到找到有效权限或到达Platform级别。这种级联查找机制简化了权限配置，用户只需在合适的级别设置权限，下级会自动继承。

4.2 虚拟集群(vcluster)

虚拟集群是基于Kubernetes Namespace技术实现的轻量级集群抽象。每个虚拟集群拥有独立的API访问平面，但共享宿主集群的控制平面组件。

4.2.1 核心特性

资源高效： 每个虚拟集群仅需要200MB内存，相比独立物理集群节省97%以上的资源。通过Namespace级别的资源隔离，实现接近物理集群的安全隔离效果。

完全兼容： 虚拟集群100%兼容Kubernetes API，用户可以使用kubectl、Helm等标准工具，无需学习新的操作方式。

快速创建： 通过声明式配置，30秒内可以创建一个新的虚拟集群并投入使用。

4.3 应用生命周期管理

Edge Platform采用三层解耦架构管理应用的完整生命周期，从应用定义到部署运行，提供标准化的管理流程。

4.3.1 三层架构

Application（应用层）： 定义应用的基本信息和元数据，包括应用名称、描述、所有者等。应用是业务层面的抽象，与技术实现无关。

ApplicationVersion（应用版本层）： 管理应用的具体实现版本，包括部署方式、配置参数、依赖关系等。一个应用可以有多个版本并存。

ApplicationDeployment（应用部署层）： 控制应用在特定环境中的部署状态，包括部署目标、副本数量、运行状态等。支持同一版本部署到多个不同的拓扑环境。

4.3.2 Plugin机制

通过可扩展的Plugin机制，支持不同类型的应用部署方式。内置Workload和Helm Chart支持，可以扩展AI模型、函数计算等新型应用类型。

4.4 声明式组件管理

Component是Edge Platform中用于管理平台组件的声明式机制。用户通过自定义资源定义声明期望的组件状态，系统自动处理安装、升级、维护等复杂操作。

4.4.1 核心优势

GitOps友好： 所有组件配置都可以纳入Git版本管理，支持CI/CD流程集成，变更可追溯、可回滚。

幂**等操作**： 重复执行相同的操作会得到一致的结果，避免了传统Shell脚本的幂等性问题。

自动依赖管理： 系统能够自动检测和解析组件之间的依赖关系，按正确顺序安装组件。

离线**部署**： 通过每集群独立的Chart仓库和预置Charts，支持完全离线的部署和升级环境。

4.5 边缘运行时集��

Edge Platform原生支持多种边缘计算运行时，为不同的业务场景提供灵活的边缘计算能力。

4.5.1 支持的运行时

KubeEdge： CNCF开源的边缘计算框架，提供云边协同、设备管理、边缘应用部署等能力。

OpenYurt： 阿里巴巴开源的边缘计算解决方案，提供无缝的云边一体化体验。

标**准Kubernetes**： 支持在标准Kubernetes环境上部署边缘应用，适用于已有K8s基础设施的场景。

4.5.2 自动化集成

通过声明式配置，可以自动安装和配置边缘运行时组件。系统会处理网络配置、节点注册、证书管理等复杂操作，大大简化了边缘环境的部署和维护工作。

ewpage

5 第二章 产品优势

6 2.1 技术优势分析

6.1 权限管理的技术深度分析

6.1.1 Kubernetes RBAC的局限性

传统的Kubernetes RBAC在设计上存在几个根本性限制：

权限粒度不足： - 仅提供Cluster和Namespace两个作用域级别 - 企业需要更细粒度的控制：平台→业务线→项目→环境→资源 - 跨Namespace的资源权限管理困难

缺乏权限继承机制： - 每个Namespace的权限需要独立配置 - 无法实现权限的级联和继承 - 大规模集群下权限配置工作量巨大

前后端权限分离： - K8s RBAC只控制API权限 - UI界面的权限控制需要额外实现 - 两者容易出现不一致

6.1.2 Edge Platform的架构创新

5层Scope权限模型：

```
Platform（平台全局）
    ↓
Cluster（集群级别）
    ↓
Workspace/NodeGroup（工作空间/节点组）
    ↓
Namespace/Node（命名空间/节点）
    ↓
Resource（资源级别）
```

核心机制： - 权限级联查找：当在当前Scope无权限时，向上级Scope查找 - 双视图并行：Workspace（应用开发视图）和NodeGroup（基础设施视图） - 前后端统一：UI权限和API权限通过同一套CRD管理

性能优化： - 三级缓存：内存缓存、本地缓存、etcd查询 - 权限预计算：用户登录时预加载常用权限 - 短路评估：找到允许权限立即返回

6.2 多集群管理的技术实现

6.2.1 传统方案的挑战

资源开销问题： 每个Kubernetes集群需要独立的控制平面： - 3个Master节点（高可用） - 每个Master节点需要2-4GB内存 - 100个集群需要300个Master节点，600GB+内存

管理复杂度： - 每个集群需要独立部署和配置 - 跨集群应用分发需要逐个操作 - 监控和日志系统无法统一

网络拓扑： - 跨集群通信延迟高 - 边缘环境网络不稳定 - 安全域管理复杂

6.2.2 vcluster技术架构

核心原理：

```
宿主集群
├── kube-apiserver（共享）
├── kube-scheduler（共享）
├── kube-controller-manager（共享）
└── 虚拟集群1
    ├── vcluster（Pod）
    ├── 分离的etcd
    └── 独立的Namespace空间
```

技术优势： - 资源效率：每虚拟集群仅200MB内存 - 隔离性：Namespace级别的资源隔离 - 兼容性：100%兼容Kubernetes API

自动化流程： 1. 用户创建Cluster CR 2. Controller解析CR配置 3. 调用Helm创建vcluster实例 4. 安装边缘运行时组件 5. 配置网络和存储 6. 更新集群状态

6.3 应用生命周期管理架构

6.3.1 应用管理的复杂性

版本管理挑战： - 应用版本与环境配置耦合 - 缺乏统一的版本规范 - 回滚操作复杂且风险高

部署流程问题： - 人工操作多，容易出错 - 部署状态难以追踪 - 缺乏标准化的部署流程

类型扩展困难： - 仅支持标准Kubernetes工作负载 - 无法扩展AI模型、函数计算等新型应用 - 自定义应用类型需要大量开发工作

6.3.2 三层解耦架构设计

Application Layer： 定义应用的基本信息、显示名称、描述和所有者。作为应用的核心标识，与具体版本和部署环境解耦。

ApplicationVersion Layer： 管理应用的具体版本实现，包括版本号、部署方式（Helm、Workload等）和配置参数。每个版本都是独立的，支持多版本并存。

ApplicationDeployment Layer： 控制应用在特定拓扑中的部署状态，包括部署的目标节点组、副本数量和运行状态。一个版本可以部署到多个不同的拓扑环境中。

Plugin机制： 定义可扩展的部署插件接口，支持不同类型的应用部署方式。内置Workload和Helm Chart支持，可扩展AI模型、函数计算等新型应用类型。

6.4 声明式组件管理创新

6.4.1 传统安装方式的问题

Shell脚本的局限性： - 幂等性难以保证 - 错误处理不完善 - 状态管理困难 - 难以集成到CI/CD

Helm Chart的挑战： - 依赖管理复杂 - 版本冲突问题 - 自定义配置困难 - 离线部署支持不足

6.4.2 Component CR架构设计

核心CRD结构： Component CRD声明组件的期望状态，包括： - 组件名称和标识 - Helm Chart名称和版本 - Chart仓库地址 - 自定义配置参数 - 依赖关系声明

通过声明式配置，用户只需描述"需要什么"，而非"如何安装"。

Controller工作流程： 1. 监听Component CR变化 2. 计算Spec Hash检测变更 3. 从ChartMuseum拉取Chart 4. 执行Helm install/upgrade 5. 更新组件状态 6. 处理错误和回滚

ChartMuseum集成： - 每集群独立Chart仓库 - 镜像预置常用Charts - HTTP Getter直接下载 - 零跨集群依赖

6.5 技术集成优势

6.5.1 Kubernetes生态兼容

完全原生支持： - 使用kubectl管理所有资源 - 支持Helm Chart部署 - 兼容Prometheus监控 - 集成Grafana可视化

零学习成本： - 运维团队无需学习新工具 - 现有K8s技能直接适用 - 开发流程无需改变 - 第三方工具无缝集成

6.5.2 GitOps工作流

声明式管理： - 所有配置版本化管理 - 自动同步到集群状态 - 变更历史完整追踪 - 支持金丝雀发布

CI/CD集成： - Git push触发部署 - 自动化测试集成 - 分环境配置管理 - 回滚操作简单可控

下一章节: 2.2 技术优势

ewpage

7 2.2 技术优势

7.1 Kubernetes 原生架构优势

7.1.1 1. 零学习成本的运维体验

技术实现: - 所有功能通过CRD实现,无私有API - 100%兼容kubectl、Helm、Kustomize等标准工具 - 基于K8s RBAC,无需学习新的权限系统

用户价值:

传统方案学习路径:
1. 学习专有CLI工具（2周）
2. 理解私有API（1周）
3. 配置权限系统（1周）
4. 集成监控（1周）
总计: 5周学习成本

Edge Platform学习路径:
1. 使用熟悉的kubectl（0周）
总计: 0周学习成本,立即上手

技术亮点: - **完全兼容**: `kubectl get cluster` = `kubectl get pods` - **标准化**: 遵循Kubernetes API规范,便于集成 - **生态融合**: Prometheus、Grafana、ArgoCD无缝集成

7.1.2 2. 声明式优于命令式

设计**理念**对**比**:

❌ **命令式(传统方案)**:

```
# 用户需要手动执行一系列命令
edge-cli cluster create edge-cluster-1
edge-cli cluster verify edge-cluster-1
edge-cli cluster install-vcluster edge-cluster-1
edge-cli cluster install-runtime edge-cluster-1
edge-cli cluster install-component edge-cluster-1
```

- 步骤繁琐,容易出错
- 无法版本管理
- 难以回滚
- 不支持GitOps

✅ **声明式(Edge Platform)**: 用户通过自定义资源定义声明期望状态，例如创建一个名为edge-cluster-1的集群，启用vcluster虚拟化和KubeEdge边缘运行时。系统会自动处理所有必要的操作，包括虚拟集群创建、边缘运行时安装、网络配置等。配置本身就是文档，易于理解和管理，支持Git版本控制和团队协作。 - 配置即文档,易于理解 - Git版本管理,可追溯 - 幂等性,可重复应用 - Controller持续调谐,自动恢复

技术优势: - **GitOps友好**: CR可纳入CI/CD流程 - **自动恢复**: Controller检测到漂移自动修复 - **审计完整**: 所有变更记录在Git历史中

7.1.3 3. 幂等操作设计

技术原理:

每个操作执行多次,结果与执行一次相同,避免复杂的状态机。

对**比**:

❌ **状态机方案(传统)**:

```
9个状态: initial → verifying → verify_success →
        cluster_installing → cluster_installed →
        runtime_installing → runtime_installed →
        components_installing → ready
```

问题:
- 某步骤失败,整个流程失败
- 状态转换链路长,难以调试
- 重试需要从头开始

✅ **幂等操作(Edge Platform)**:

流程简化:
```
ensureVCluster()       // 检查→不存在则创建→存在则跳过
ensureEdgeRuntime()    // 检查→不存在则安装→存在则跳过
ensureComponents()     // 检查→不存在则安装→存在则跳过
```

优势:
- 失败自动重试,从失败点继续
- 状态简单,易于理解
- 可随时重新触发

技术实现: - 检查**再**创**建**: 先检查资源是否存在 - **Server-Side Apply**: Kubernetes原生幂等机制 - **无状**态**机**: 避免状态转换复杂度

7.2 **性能**优势

7.2.1 1. **权限**检查**性能**

性能指标: - **P50延**迟: <1ms - **P99延**迟: <5ms - **并**发**支持**: 10000+ QPS - **用**户规**模**: 支持10000+用户

性能优**化技**术:

7.2.1.1 **三**级缓**存架构**

```
L1缓存: 内存缓存 (命中率95%)
    ↓ Miss
L2缓存: 本地存储 (命中率4%)
    ↓ Miss
L3源: etcd查询 (命中率1%)
```

7.2.1.2 **短路**评**估**

权限级联查找:
```
Namespace → Workspace → Cluster → Platform → Anonymous
```

优化: 找到第一个Allow规则立即返回,不继续查找
性能提升: 平均减少60%的查找次数

7.2.1.3 Visitor**模式批量**查询

传统方案: 逐个查询IAMRoleBinding
优化方案: 一次查询返回所有相关Binding

性能提升: 查询次数从N次降为1次

对**比**竞**品**:

平台	**P99延**迟	**支持用**户**数**	缓**存机制**
Edge Platform	<5ms	10000+	✅ 三级缓存
KubeEdge	~50ms	<1000	❌ 无缓存

平台	P99延迟	支持用户数	缓存机制
OpenYurt	~30ms	<5000	⚠️ 单级缓存

7.2.2 2. 虚拟集群资源效率

资源对比:

方案	内存占用	CPU占用	启动时间	资源节省
物理K8s集群	4GB	2核	30分钟	基准
Edge Platform vcluster	200MB	0.1核	30秒	**95%↓**

技术创新: - **共享宿主资源**: vcluster共享宿主集群的etcd、调度器 - 轻量级**控制面**: 仅运行必需组件(API Server + Controller) - **按需扩展**: 根据负载动态调整资源

实际**案例**:

某客户边缘环境:
- 需求: 100个边缘站点,每站点1个K8s集群
- 传统方案: 100个物理集群,500台服务器,成本500万
- Edge Platform: 1个物理集群+100个vcluster,13台服务器,成本65万
- 节省: 87%成本 ↓, 97%资源 ↓

7.2.3 3. Component安装性能

性能指标: - 单组**件安装**: <30秒 - **完整平台安装**: <5分钟 - **并发安装**: 支持3个组件同时安装

性能优化:

7.2.3.1 Spec Hash检测变更

原理:
- 计算Component Spec的SHA256哈希值
- 对比Status中的ObservedHash
- 仅在变更时执行Helm操作

优势:
- 避免不必要的Helm调用
- 减少API Server压力

7.2.3.2 HTTP Getter直接下载

传统方案: helm repo add → helm install (2步)
优化方案: HTTP GET Chart → helm install (1步)

性能提升: 安装时间减少40%

7.3 可扩展性优势

7.3.1 1. Plugin机制

设计理念:

通过插件机制支持多种应用类型,无需修改核心代码。

架构优势:

```
核心抽象:
ProvisionerPlugin 接口:
  - Validate()    // 校验配置
  - Provision()   // 部署资源
  - Update()      // 更新资源
  - Delete()      // 删除资源
  - GetStatus()   // 获取状态

内置实现:
- WorkloadPlugin  (Kubernetes原生工作负载)
- HelmPlugin      (Helm Chart应用)
- ModelPlugin     (AI模型应用)

扩展能力:
用户可实现自定义Plugin,支持新应用类型
```

技术亮点: - 开闭原则: 对扩展开放,对修改封闭 - 热插拔: 无需重启Controller,动态注册Plugin - 类型安全: 编译期检查Plugin实现

实际应用:

```
某客户需求: 支持Serverless函数部署

传统方案:
- 修改核心代码 (2周开发)
- 测试回归 (1周)
- 升级风险 (高)

Edge Platform方案:
- 实现FunctionPlugin (2天开发)
- 注册到Registry (1行代码)
- 零升级风险 (插件独立)
```

7.3.2 2. CRD扩展能力

Kubernetes扩展机制:

扩展方式	Edge Platform	竞品方案
API扩展	✅ CRD	⚠️ 私有API
Controller扩展	✅ Operator模式	⚠️ 硬编码逻辑
存储扩展	✅ etcd	⚠️ 自建数据库

扩展方式	Edge Platform	竞品方案
权限扩展	✅ K8s RBAC	⚠️ 自建权限系统

技术优势: - 标**准化**: 遵循Kubernetes API规范 - **工具**链: kubectl、Helm、Kustomize直接可用 - **生**态: 与Prometheus、Grafana等无缝集成

7.3.3 3. 多发行版支持

虚拟**集群**发**行版支持**:

发行版	资源占用	启动时间	适用场景
k3s	200MB	30秒	边缘环境(推荐)
k0s	300MB	45秒	中等规模边缘
k8s	500MB	60秒	云端环境

边缘**运行**时**支持**:

运行时	来源	成熟度	社区活跃度
KubeEdge	CNCF	✅ 生产可用	⭐⭐⭐⭐⭐
OpenYurt	阿里云	✅ 生产可用	⭐⭐⭐⭐

技术创**新**: - **自**动检测: 根据annotation自动选择发行版 - 统**一接口**: 不同发行版统一Cluster CR管理 - **灵活切**换: 可随时更换发行版

7.4 可靠性优势

7.4.1 1. 幂等操作设计

核心理念： Edge Platform将幂等操作作为设计的核心原则，确保每个操作重复执行的结果与执行一次相同。这种设计为运维带来了革命性的价值。

运维**价**值：

运维挑战	传统方案（状态机）	Edge Platform（幂等操作）
故障恢复	需要人工干预，定位故障点	自动检测并修复，无需人工介入
配置漂移	手动对比配置，容易出错	自动调谐，始终保持期望状态
重复操作	需要判断当前状态，风险高	可随时重复执行，安全可靠
升级回滚	复杂的回滚流程，容易失败	自动重试，失败自动恢复

Controller调谐**机制**：

调谐循环（每30秒执行）：
1. 检测：获取资源的当前状态
2. 对比：与期望状态（Spec）进行比较
3. 计算：确定需要执行的操作
4. 执行：使实际状态向期望状态收敛
5. 更新：记录当前状态到Status

示例：虚拟集群创建
- Step 1: ensureVCluster() - 检查vcluster，不存在则创建
- Step 2: ensureEdgeRuntime() - 检查运行时，不存在则安装
- Step 3: ensureComponents() - 检查组件，不存在则部署
- 每个Step都是幂等的，可独立重试

实际**案例**：

场景：网络中断导致组件安装失败

传统方案（状态机）：
1. 安装失败 → 状态停留在"installing"
2. 需要人工清理状态
3. 重新执行安装流程
4. 总耗时：2-4小时

Edge Platform（幂等操作）：
1. 网络恢复 → Controller检测到差异
2. 自动重新执行ensureComponents()
3. 组件自动安装完成
4. 总耗时：5-10分钟

7.4.2 2. 自动恢复能力

7.4.3 2. 高可用架构

组**件高可用**:

组**件**	**副本数**	**故障转移**	**数据持久化**
APIServer	3	✅ 负载均衡	✅ etcd
Controller	2	✅ Leader选举	✅ etcd
ChartMuseum	1	⚠️ 单点(可扩展)	✅ PV

技术保障: - **Leader**选举: Controller使用Lease机制实现Leader选举 - 负载**均衡**: APIServer通过Service实现流量分发 - **数据**备**份**: etcd定期快照,支持灾难恢复

7.4.4 3. 幂等性保障

技术实现:

所有操作都设计为幂等,重复执行结果一致。

幂**等性**验证:

测试场景：创建虚拟集群

第1次执行：kubectl apply -f cluster.yaml
结果：cluster.scope.theriseunion.io/edge-cluster-1 created

第2次执行：kubectl apply -f cluster.yaml

结果: cluster.scope.theriseunion.io/edge-cluster-1 unchanged

第3次执行(删除后): kubectl apply -f cluster.yaml
结果: cluster.scope.theriseunion.io/edge-cluster-1 created

结论: ✅ 幂等性验证通过

7.5 可观测性优势

7.5.1 1. 数据存储方案选择

方案对比:

Edge Platform将动态统计数据存储在Prometheus中，这是一种合理的架构选择。

传统**方案**: - 在CRD Status中维护统计信息 - 适合低频更新的关键状态数据

Edge Platform方案: - 将高频变化的指标通过Prometheus暴露 - 利用时序数据库的天然优势

技术特点: - 减少etcd的写入压力 - 支持历史数据查询和趋势分析 - 可以利用PromQL进行灵活的数据聚合

7.5.2 2. 全链路追踪

可观测性矩阵:

层级	工具	数据类型	用途
Metrics	Prometheus	时序指标	性能监控、告警
Logs	Loki	日志	问题定位、审计
Traces	Jaeger	调用链	性能分析
Events	K8s Events	事件	变更追踪

技术实现: - **自动采集**: 无需修改代码,自动暴露Metrics端点 - **标准化**: 遵循Prometheus命名规范 - **预置大盘**: Grafana开箱即用监控面板

7.5.3 3. 审计日志

审计能力:

记录内容:
- 谁（用户）
- 什么时间（时间戳）
- 做了什么（操作类型）
- 对什么资源（资源类型、名称）
- 结果如何（成功/失败）

查询能力:
- 按用户查询
- 按资源查询

- 按时间范围查询
- 按操作类型查询

合规支持: - **等保2.0**: 满足审计要求 - **ISO27001**: 满足信息安全管理体系 - **SOC2**: 满足安全性、可用性、处理完整性

7.6 安全性优势

7.6.1 1. 最小权限原则

权限设计:

原则：用户仅获得完成工作所需的最小权限

实现:
- Workspace开发者：仅能操作自己Workspace的资源
- NodeGroup运维：仅能管理指定NodeGroup的节点
- 集群管理员：仅能管理特定集群

安全边界:

Scope层级	权限范围	安全隔离
Platform	全局	平台管理员专属
Cluster	单集群	集群间隔离
Workspace	应用视图	租户隔离
NodeGroup	资源视图	基础设施隔离
Namespace	命名空间	K8s原生隔离

7.6.2 2. 网络隔离

虚拟集群网络隔离:

技术实现:
- vcluster使用独立的Service CIDR
- NetworkPolicy限制跨namespace通信
- 宿主集群与虚拟集群网络隔离

安全保障: - **租户隔离**: 不同vcluster间网络完全隔离 - **流量控制**: NetworkPolicy细粒度控制 - **加密传输**: 支持TLS加密

7.6.3 3. 镜像安全

镜像扫描:

流程:
1. 镜像上传到私有仓库
2. 自动触发漏洞扫描(Trivy)
3. 发现高危漏洞阻止部署
4. 生成安全报告

安全策略: - 镜像签名: 支持Cosign签名验证 - **准入控制**: Admission Webhook拦截不安全镜像 - **漏洞修复**: 自动通知漏洞修复建议

下一章节: 2.3 业务价值

ewpage

8 第三章 核心功能模块

9 第三章 核心功能模块

本章重点展示Edge Platform的核心功能模块及其技术优势,而非详细的实现细节。

9.1 模块概览

Edge Platform提供四大核心功能模块,每个模块都具有业界领先的技术创新:

模块	核心价值	技术创新	业务收益
权限管理	企业级多租户权限控制	5层Scope模型 + 权限级联	满足复杂组织架构
多集群管理	轻量级虚拟集群	vcluster自动化编排	资源节省97%
应用商店	统一应用生命周期	三层架构 + Plugin机制	一键分发到多节点组
声明式安装	GitOps友好组件管理	ChartMuseum + Component CR	完全离线,零依赖

9.2 3.1 权限管理模块

9.2.1 核心价值

为企业提供**业界最灵活的多租户权限控制**,支持从全局到资源级的精确权限管理,完全兼容Kubernetes生态。

9.2.2 技术创新

9.2.2.1 1. 5层Scope权限模型(业界独创)

```
Platform(平台级)
    ↓
Cluster(集群级)
    ↓
Workspace(工作空间) / NodeGroup(节点组)  ← 双视图并行设计
    ↓
Namespace(命名空间) / Node(节点)
    ↓
Resource(资源级)
```

创新点: - 双视图并行: Workspace(应用视图) 和 NodeGroup(资源视图) 同时存在 - 权限级联: 上级Scope权限自动继承到下级 - 短路评估: 找到权限立即返回,P99延迟<5ms

竞品对比: - KubeEdge: 单层RBAC,无多租户隔离 - OpenYurt: 2层RBAC,仅命名空间级 - Edge Platform: **5层Scope,支持10000+用户**

9.2.2.2 2. Kubernetes Native设计

完全基于K8s RBAC: - RoleTemplate CRD → IAMRole CRD → K8s ClusterRole - 零学习成本,kubectl原生支持 - 100%兼容Helm和其他K8s工具

前后端权限一体化: - API权限: K8s PolicyRules - UI权限: UIPermissions字段 - 一次声明,前后端同时生效

9.2.2.3 3. 高性能权限检查

三级缓存架构:

```
L1缓存(内存) → 95%命中率, <1ms
L2缓存(本地) → 4%命中率, <10ms
L3查询(etcd) → 1%命中率, <100ms
```

性能指标: - P50延迟: <1ms - P99延迟: <5ms - 并发QPS: 10000+ - 支持用户数: 10000+

9.2.3 业务价值

解决的痛点: - ✅ 复杂组织架构权限管理 - ✅ 开发/测试/生产环境隔离 - ✅ 跨团队资源共享与隔离 - ✅ 审计与合规要求

实际案例: 某金融企业使用Edge Platform管理200+集群,5000+用户: - 权限配置时间: 从3天降至30分钟 - 权限审计效率: 提升10倍 - 安全事件: 零误操作

9.3 3.2 多集群管理模块

9.3.1 核心价值

通过**vcluster虚拟集群技术**,实现轻量级、高隔离的多集群管理,大幅降低边缘环境资源开销。

9.3.2 技术创新

9.3.2.1 1. vcluster自动化编排(业界独创)

传统方案 vs Edge Platform:

维度	传统物理集群	Edge Platform(vcluster)
资源开销	3 Master + 2 Worker	1个Pod(200MB内存)
启动时间	30分钟+	30秒

维度	传统物理集群	Edge Platform(vcluster)
隔离级别	物理隔离	Namespace级隔离
成本	5台服务器	共享宿主资源

资源节省: 97%↓

创新点: - 声明式创建: 通过Cluster CR一键创建 - 自动安装边缘运行时(KubeEdge/OpenYurt) - 多发行版支持(k3s/k0s/k8s)

9.3.2.2 2. 边缘运行时集成

自动化安装流程:

```
1. 创建Cluster CR
2. Controller创建vcluster
3. 自动安装CloudCore
4. 配置边缘节点接入
5. 安装依赖组件(监控/日志)
```

支持的边缘运行时: - KubeEdge 1.20.0 - OpenYurt 1.6.0 - 兼容标准Kubernetes API

9.3.2.3 3. 组件依赖管理

自动解决依赖关系: - 监控依赖Prometheus → 自动安装Prometheus - 日志依赖Loki → 自动安装Loki - 拓扑排序保证安装顺序

9.3.3 业务价值

解决的痛点: - ✅ 边缘环境资源受限 - ✅ 大量站点集群管理复杂 - ✅ 边缘运行时配置繁琐 - ✅ 集群创建周期长

实际案例: 某制造业100个工厂边缘计算: - 物理方案: 500台服务器,800万/3年 - Edge Platform: 13台服务器,104万/3年 - 节省: 696万(87%↓)

9.4 3.3 应用商店模块

9.4.1 核心价值

提供企业级应用全生命周期管理,支持多类型应用统一管理,一键分发到多个节点组。

9.4.2 技术创新

9.4.2.1 1. 三层架构设计(业界独创)

```
Application(应用元数据)
    ↓
ApplicationVersion(应用版本)
```

```
    ↓
ApplicationDeployment(应用部署)
```

创新点: - 应用与版本分离: 支持多版本并存 - 版本与部署分离: 同一版本部署到多拓扑 - 语义化版本: latest自动解析到具体版本

9.4.2.2 2. Plugin可扩展机制

支持的应用类型: - Workload: K8s原生工作负载(Deployment/StatefulSet) - Helm: Helm Chart应用 - AI Model: AI模型应用(未来扩展) - Function: 函数计算(未来扩展)

Plugin接口:

```
Provisioner Interface:
- Provision(version, topology)
- Unprovision(deployment)
- GetStatus(deployment)
```

9.4.2.3 3. 多拓扑智能部署

一个应用部署到多个节点组: 支持将同一个应用同时部署到不同的节点组中，每个节点组可以配置不同的副本数。例如，可以在北京边缘节点组部署3个副本，在上海边缘节点组部署5个副本，在深圳边缘节点组部署3个副本。系统会自动处理版本解析、节点组亲和性调度、跨节点组负载均衡等复杂逻辑。

自动处理: - 节点组亲和性调度 - 跨节点组负载均衡 - 拓扑状态独立跟踪

9.4.3 业务价值

解决的痛点: - ✅ 应用版本管理混乱 - ✅ 跨节点组应用分发困难 - ✅ 应用审核流程缺失 - ✅ 回滚和灰度发布复杂

实际案例: 某零售企业500家门店应用分发: - 传统方案: 逐店部署,2天完成 - Edge Platform: 一键分发,30分钟完成 - 效率提升: 96倍

9.5 3.4 声明式安装模块

9.5.1 核心价值

通过ChartMuseum + Component CR架构,实现完全离线、零依赖的声明式组件管理,GitOps友好。

9.5.2 技术创新

9.5.2.1 1. 每集群独立Chart仓库(业界独创)

传统方案 vs Edge Platform:

维度	传统方案	Edge Platform
Chart仓库	中心仓库	每集群独立ChartMuseum
跨集群依赖	有,网络故障导致无法安装	无,完全独立
离线能力	需下载Chart	Chart预置在镜像中
单点故障	中心仓库故障影响所有集群	集群独立,故障隔离

创新点: - 镜像内置常用Charts - 用户上传Chart存储在PV - 完全离线部署能力

9.5.2.2 2. Component CR声明式管理

GitOps友好: 通过自定义资源定义声明组件的期望状态。用户只需要描述需要安装的组件名称、版本号和配置参数，系统会自动处理安装和升级过程。例如，可以声明安装Prometheus组件并指定数据保留时间为15天。所有配置都可以纳入Git版本管理，支持CI/CD流程集成。

Controller自动化: - Spec Hash检测变更 - 自动helm install/upgrade - 幂等操作,重复执行结果一致 - 故障自动恢复

9.5.2.3 3. 依赖管理

自动解决依赖: 系统具备智能依赖管理能力，能够自动检测组件之间的依赖关系。当安装一个组件时，如果该组件依赖其他组件，系统会先安装被依赖的组件，再安装主组件。例如，如果要安装监控面板，系统会自动先安装Prometheus作为依赖，然后再安装Grafana面板。整个过程自动完成，无需人工干预。

9.5.3 业务价值

解决的痛点: - ✅ 组件安装shell脚本维护困难 - ✅ 跨集群依赖导致故障传播 - ✅ 离线环境无法安装组件 - ✅ 组件升级风险高

实际案例: 某能源企业50个边缘站点离线部署: - 传统方案: 人工配置,每站点2天 - Edge Platform: Component CR,30分钟 - 人力节省: 99天工作量

9.6 功能对比矩阵

9.6.1 权限管理能力

能力	Edge Platform	KubeEdge	OpenYurt	K3s
权限模型	5层Scope	单层	2层	单层
多租户	✅	❌	⚠️	❌
UI权限	✅	❌	❌	❌
权限级联	✅	❌	❌	❌
性能(P99)	<5ms	N/A	N/A	N/A

9.6.2 多集群管理能力

能力	Edge Platform	KubeEdge	OpenYurt	K3s
虚拟集群	✅ vcluster	❌	❌	❌
资源开销	200MB	N/A	N/A	N/A
启动时间	30秒	30分钟	30分钟	30分钟
边缘运行时	自动安装	手动	手动	❌
资源节省	97%	0%	0%	0%

9.6.3 应用商店能力

能力	Edge Platform	KubeEdge	OpenYurt	K3s
应用商店	✅	❌	❌	❌
版本管理	✅	❌	❌	❌
多拓扑	✅	❌	❌	❌
Plugin扩展	✅	❌	❌	❌

9.6.4 声明式安装能力

能力	Edge Platform	KubeEdge	OpenYurt	K3s
声明式	✅ Component CR	❌ Shell	❌ Shell	❌ Shell
离线部署	✅	⚠️	⚠️	⚠️
依赖管理	✅	❌	❌	❌
GitOps	✅	❌	❌	❌

下一章节: 第四章 技术架构

ewpage

10 第四章 技术架构

11 4.1 系统逻辑架构

11.1 整体架构

11.1.1 分层架构图

```
graph TB
    subgraph 用户层
        U1[平台管理员]
        U2[集群管理员]
        U3[应用开发者]
        U4[运维人员]
    end
```

```
subgraph 接入层
    P1[Web Console]
    P2[kubectl CLI]
    P3[Helm CLI]
    P4[API Client]
end

subgraph API层
    A1[REST API Server]
    A2[K8s API Server]
    A3[Webhook Server]
end

subgraph 业务逻辑层
    B1[权限管理]
    B2[多集群管理]
    B3[应用商店]
    B4[组件管理]
end

subgraph Controller层
    C1[RoleTemplate Controller]
    C2[IAMRole Controller]
    C3[Cluster Controller]
    C4[Application Controller]
    C5[Component Controller]
end

subgraph 存储层
    S1[etcd]
    S2[Prometheus]
    S3[ChartMuseum]
end

subgraph 基础设施层
    I1[Kubernetes]
    I2[vcluster]
    I3[KubeEdge/OpenYurt]
end

U1 --> P1
U2 --> P2
U3 --> P3
U4 --> P4

P1 --> A1
P2 --> A2
P3 --> A2
P4 --> A1

A1 --> B1
A1 --> B2
A1 --> B3
A1 --> B4
A2 --> B1
```

```
    B1 --> C1
    B1 --> C2
    B2 --> C3
    B3 --> C4
    B4 --> C5

    C1 --> S1
    C2 --> S1
    C3 --> S1
    C4 --> S1
    C5 --> S3

    C1 --> I1
    C3 --> I2
    C3 --> I3
```

11.2 核心模块

11.2.1 1. 权限管理模块

功能: - 5层Scope权限模型 - RoleTemplate → IAMRole → K8s RBAC转换 - 权限级联查找 - UI权限控制

关键组件: - RoleTemplate CRD + Controller - IAMRole CRD + Controller - IAMRoleBinding CRD + Controller - Permission Authorizer - Scope Resolver

技术创新: - 双视图并行设计(Workspace + NodeGroup) - 三级缓存(P99 < 5ms) - 短路评估优化

11.2.2 2. 多集群管理模块

功能: - 虚拟集群自动化创建 - 边缘运行时集成 - 组件依赖管理 - 集群状态同步

关键组件: - Cluster CRD + Controller - vcluster Helm Chart - Edge Runtime Installer - Cluster Status Syncer

技术创新: - 声明式创建(Cluster CR) - 幂等操作设计 - 多发行版支持(k3s/k0s/k8s)

11.2.3 3. 应用商店模块

功能: - 应用全生命周期管理 - 多类型应用支持 - 版本管理与审核 - 多拓扑部署

关键组件: - Application CRD - ApplicationVersion CRD + Controller - ApplicationDeployment CRD + Controller - Provisioner Plugin Registry

技术创新: - 三层架构设计 - Plugin可扩展机制 - API驱动审核流程

11.2.4 4. 安装部署模块

功能: - 声明式组件管理 - 自动化安装升级 - 离线部署支持 - 集群独立性

关键组件: - Component CRD + Controller - ChartMuseum - Helm Installer - Chart Repository

技术创新: - ChartMuseum + Component CR架构 - HTTP Getter直接下载 - Spec Hash检测变更

11.3 数据流

11.3.1 权限检查流程

```
sequenceDiagram
    participant User as 用户
    participant API as API Server
    participant Auth as Authorizer
    participant Cache as 缓存
    participant etcd as etcd

    User->>API: 请求资源操作
    API->>Auth: 权限检查
    Auth->>Cache: 查询L1缓存

    alt 缓存命中
        Cache-->>Auth: 返回权限
    else 缓存未命中
        Auth->>etcd: 查询IAMRoleBinding
        etcd-->>Auth: 返回Binding
        Auth->>Cache: 更新缓存
    end

    Auth->>Auth: 级联查找Scope链

    alt 允许
        Auth-->>API: Allow
        API-->>User: 执行操作
    else 拒绝
        Auth-->>API: Deny
        API-->>User: 403 Forbidden
    end
```

11.3.2 虚拟集群创建流程

```
sequenceDiagram
    participant User as 用户
    participant K8s as K8s API
    participant Controller as Cluster Controller
    participant Helm as Helm Client
    participant vcluster as vcluster

    User->>K8s: 创建 Cluster CR
    K8s->>Controller: Watch触发Reconcile

    Controller->>Controller: ensureVCluster()
    Controller->>vcluster: 检查是否存在

    alt vcluster不存在
        Controller->>Helm: helm install vcluster
        Helm->>vcluster: 创建vcluster
        vcluster-->>Controller: 返回kubeconfig
```

```
        Controller->>K8s: 更新Cluster.Spec.Connection
    end

    Controller->>Controller: ensureEdgeRuntime()
    Controller->>Controller: ensureComponents()
    Controller->>K8s: 更新Cluster.Status
    K8s-->>User: 集群就绪
```

11.3.3 应用部署流程

```
sequenceDiagram
    participant User as 用户
    participant API as API Server
    participant Controller as ApplicationDeployment Controller
    participant Plugin as Provisioner Plugin
    participant K8s as Kubernetes

    User->>API: 创建ApplicationDeployment
    API->>Controller: Watch触发Reconcile

    Controller->>Controller: 解析版本(latest→v1.0.0)
    Controller->>Controller: 获取ApplicationVersion
    Controller->>Controller: 获取Provisioner Plugin

    loop 每个Topology
        Controller->>Plugin: Provision()
        Plugin->>K8s: 创建Deployment/Service
    end

    Controller->>Plugin: GetStatus()
    Plugin-->>Controller: 拓扑状态
    Controller->>API: 更新Status
    API-->>User: 部署完成
```

11.4 技术架构特点

11.4.1 1. Kubernetes Native

设计**原则**: - 所有功能通过CRD实现 - 基于K8s RBAC权限控制 - 使用etcd作为存储 - Controller模式实现业务逻辑

优势: - 零学习成本 - 生态兼容性好 - 高可用性保障 - 标准化程度高

11.4.2 2. 声明式架构

核心理念: - 用户声明期望状态(CR) - Controller持续调谐 - 幂等操作设计 - 自动故障恢复

实现**方式**: - CRD定义期望状态 - Controller Watch CR变化 - Reconcile Loop调谐 - Status反映实际状态

11.4.3 3. 微服务架构

组件拆分: - APIServer: REST API网关 - Controller: 业务逻辑控制器 - Console: Web前端 - ChartMuseum: Chart仓库

通信方式: - APIServer ↔ etcd: gRPC - Controller ↔ K8s API: client-go - Console ↔ APIServer: HTTP/REST - ChartMuseum ↔ Controller: HTTP

11.4.4 4. 插件化设计

扩展点: - Provisioner Plugin: 应用类型扩展 - Scope Pattern: Scope类型扩展 - Webhook: 准入控制扩展 - Metrics: 监控指标扩展

扩展机制: - Interface定义标准 - Registry注册机制 - 热插拔支持 - 独立升级能力

11.5 高可用设计

11.5.1 组件高可用

组件	部署模式	高可用机制
APIServer	多副本(3)	K8s Service负载均衡
Controller	多副本(2)	Leader Election
etcd	集群(3)	Raft一致性协议
Console	多副本(2)	K8s Service负载均衡
ChartMuseum	单副本	PV持久化

11.5.2 故障恢复

自动恢复场景: - Pod故障: K8s自动重启 - 配置漂移: Controller自动调谐 - 网络抖动: 自动重试机制 - 组件异常: 健康检查+自动恢复

数据备份: - etcd定期快照 - PV数据备份 - 配置Git版本管理 - 灾难恢复方案

11.6 性能优化

11.6.1 缓存策略

三级缓存: - L1: 内存缓存(命中率95%) - L2: 本地存储(命中率4%) - L3: etcd查询(命中率1%)

缓存更新: - Watch机制实时更新 - TTL过期自动刷新 - LRU淘汰策略

11.6.2 并发控制

Controller并发: - MaxConcurrentReconciles: 3 - Worker Queue缓冲 - Rate Limiting限流

API限流: - QPS限制 - Burst缓冲 - 用户级别隔离

11.6.3 数据库优化

etcd优化: - Metrics优先架构(减少写入80%) - 批量操作合并 - Compact定期执行 - 分片存储

下一章节: 4.2 系统技术架构

ewpage

12 4.2 系统技术架构

12.1 技术栈总览

12.1.1 核心技术栈

技术分类	技术选型	版本	用途
编程语言	Go	1.20+	Controller开发
编程语言	TypeScript	5.0+	前端开发
容器编排	Kubernetes	1.26+	平台基础
框架	controller-runtime	v0.15.0	Operator框架
框架	Next.js	14.x	前端框架
虚拟集群	vcluster	0.29.x	虚拟集群
包管理	Helm	3.12+	应用打包
边缘运行时	KubeEdge	1.20.0	边缘计算
边缘运行时	OpenYurt	1.6.0	边缘计算
Chart仓库	ChartMuseum	0.16.1	Chart存储
监控	Prometheus	2.45+	指标采集
日志	Loki	2.8+	日志聚合
可视化	Grafana	10.0+	监控面板

12.2 后端架构

12.2.1 APIServer架构

```
graph TB
    subgraph APIServer进程
        R[HTTP Router]
        R --> Auth[认证中间件]
        Auth --> Authz[授权中间件]
        Authz --> H1[IAM Handler]
        Authz --> H2[Cluster Handler]
        Authz --> H3[Application Handler]
        Authz --> H4[Component Handler]

        H1 --> C1[K8s Client]
        H2 --> C1
```

```
        H3 --> C1
        H4 --> C1

        C1 --> API[K8s API Server]
    end

    subgraph 外部依赖
        API --> etcd[(etcd)]
        H1 --> Prom[Prometheus]
        H3 --> Prom
    end
```

核心组件:

1. **HTTP Router**: 路由分发
   - 基于restful框架
   - 支持OpenAPI 3.0
   - 自动生成API文档
2. 认证**中间件**: OAuth 2.0
   - JWT Token验证
   - OIDC集成
   - Session管理
3. **授权中间件**: RBAC
   - 集成K8s RBAC
   - Scope级联查找
   - 权限缓存
4. **Handler**层: 业务逻辑
   - RESTful API实现
   - 数据验证
   - 错误处理
5. **Client**层: K8s交互
   - client-go封装
   - Dynamic Client
   - Server-Side Apply

12.2.2 Controller架构

```
graph TB
    subgraph Controller进程
        M[Controller Manager]
        M --> C1[RoleTemplate Controller]
        M --> C2[IAMRole Controller]
        M --> C3[Cluster Controller]
        M --> C4[ApplicationVersion Controller]
        M --> C5[ApplicationDeployment Controller]
        M --> C6[Component Controller]

        C1 --> WQ1[Work Queue]
        C2 --> WQ2[Work Queue]
        C3 --> WQ3[Work Queue]
        C4 --> WQ4[Work Queue]
        C5 --> WQ5[Work Queue]
        C6 --> WQ6[Work Queue]
```

```
        WQ1 --> R1[Reconciler]
        WQ2 --> R2[Reconciler]
        WQ3 --> R3[Reconciler]
        WQ4 --> R4[Reconciler]
        WQ5 --> R5[Reconciler]
        WQ6 --> R6[Reconciler]
    end

    subgraph K8s集群
        R1 --> API[K8s API Server]
        R2 --> API
        R3 --> API
        R4 --> API
        R5 --> API
        R6 --> API

        API --> etcd[(etcd)]
    end
```

Controller模式:

1. **Watch机制**: 监听CRD变化
2. **Work Queue**: 事件队列
3. **Reconcile Loop**: 调谐循环
4. **Status Update**: 状态更新

并发控制: - 每个Controller独立goroutine - MaxConcurrentReconciles: 3 - Rate Limiting防止风暴

12.2.3 前端架构

```
graph LR
    subgraph Browser
        UI[React组件]
        UI --> Hooks[Custom Hooks]
        Hooks --> SDK[API SDK]
        SDK --> Client[HTTP Client]
    end

    subgraph Backend
        Client --> API[API Server]
        API --> K8s[K8s API]
    end

    subgraph 状态管理
        Hooks --> Cache[React Query Cache]
    end
```

技术选型:

1. **框架**: Next.js 14
   - SSR/SSG支持
   - App Router
   - 性能优化

2. **UI**库: shadcn/ui
    - Tailwind CSS
    - Radix UI
    - 无障碍支持
3. **状态管理**: React Query
    - 服务端状态管理
    - 自动缓存
    - 乐观更新
4. **API Client**: openapi-typescript-codegen
    - 自动生成SDK
    - 类型安全
    - 请求拦截

12.3 数据架构

12.3.1 etcd数据模型

数据分类:

```
/registry/
├── iam.theriseunion.io/
│   ├── roletemplates/
│   ├── iamroles/
│   └── iamrolebindings/
├── scope.theriseunion.io/
│   ├── clusters/
│   ├── workspaces/
│   └── nodegroups/
├── app.theriseunion.io/
│   ├── applications/
│   ├── applicationversions/
│   └── applicationdeployments/
└── ext.theriseunion.io/
    └── components/
```

存储优**化**: - Protobuf序列化 - Compact压缩 - Watch缓存 - 分片存储

12.3.2 Prometheus数据模型

Metrics分类:

1. **系**统**指**标:
    - apiserver_request_total: API请求总数
    - apiserver_request_duration_seconds: API延迟
    - controller_reconcile_total: 调谐次数
    - controller_reconcile_duration_seconds: 调谐耗时
2. 业务**指**标:
    - cluster_total: 集群总数
    - vcluster_creation_duration_seconds: 虚拟集群创建耗时
    - app_store_application_count: 应用总数
    - app_store_deployment_count: 部署总数
    - component_install_duration_seconds: 组件安装耗时

数据保留: - 高精度数据: 15天 - 聚合数据: 90天 - 长期存储: Thanos

12.4 网络架构

12.4.1 服务拓扑

```
graph TB
    subgraph 外部访问
        User[用户]
        User --> LB[负载均衡]
    end

    subgraph K8s集群
        LB --> Ingress[Ingress Controller]

        Ingress --> Console[Console Service]
        Ingress --> API[APIServer Service]

        Console --> ConsolePod1[Console Pod 1]
        Console --> ConsolePod2[Console Pod 2]

        API --> APIPod1[APIServer Pod 1]
        API --> APIPod2[APIServer Pod 2]
        API --> APIPod3[APIServer Pod 3]

        APIPod1 --> K8sAPI[K8s API Service]
        APIPod2 --> K8sAPI
        APIPod3 --> K8sAPI

        K8sAPI --> Master1[Master 1]
        K8sAPI --> Master2[Master 2]
        K8sAPI --> Master3[Master 3]
    end

    subgraph 虚拟集群
        Master1 --> VC1[vcluster-1]
        Master1 --> VC2[vcluster-2]
    end
```

网络策略:

1. **Ingress路由**:
   - / → Console
   - /oapis/* → APIServer
   - /apis/* → K8s API
2. **Service类型**:
   - APIServer: ClusterIP
   - Console: ClusterIP
   - ChartMuseum: ClusterIP
   - Prometheus: ClusterIP
3. **网络隔离**:
   - vcluster间网络隔离
   - NetworkPolicy访问控制
   - 命名空间隔离

12.5 存储架构

12.5.1 持久化存储

存储类型:

组件	存储类型	容量	用途
etcd	PV	100Gi	K8s数据
ChartMuseum	PV	20Gi	Helm Charts
Prometheus	PV	500Gi	Metrics数据
Loki	PV	200Gi	日志数据

备份策略: - etcd: 每日快照 - Prometheus: TSDB本地保留 - Loki: 对象存储归档 - Charts: 镜像内置

12.6 安全架构

12.6.1 认证授权

```
graph LR
    U[用户] --> A[认证]
    A --> |JWT Token| B[授权]
    B --> |RBAC检查| C[API访问]

    A --> OAuth[OAuth 2.0]
    A --> OIDC[OIDC]
    A --> LDAP[LDAP]

    B --> Scope[Scope解析]
    B --> Cache[权限缓存]
    B --> K8sRBAC[K8s RBAC]
```

认证方式: - JWT Token - OIDC(支持第三方) - LDAP集成 - ServiceAccount

授权机制: - K8s RBAC - Scope级联查找 - UI权限控制 - Webhook扩展

12.6.2 网络安全

加密传输: - HTTPS/TLS 1.3 - mTLS(内部通信) - 证书自动轮转

访问控制: - NetworkPolicy - Ingress白名单 - 防火墙规则

镜像安全: - 镜像扫描(Trivy) - 镜像签名(Cosign) - 准入控制(OPA)

12.7 监控架构

12.7.1 监控体系

```
graph TB
    subgraph 数据采集
```

```
        A[应用Metrics]
        B[系统Metrics]
        C[业务Metrics]
    end

    subgraph 存储层
        A --> Prom[Prometheus]
        B --> Prom
        C --> Prom
        Prom --> TSDB[(TSDB)]
    end

    subgraph 可视化
        TSDB --> Grafana[Grafana]
        Grafana --> D1[系统大盘]
        Grafana --> D2[业务大盘]
        Grafana --> D3[应用大盘]
    end

    subgraph 告警
        Prom --> Alert[AlertManager]
        Alert --> Email[邮件]
        Alert --> Webhook[Webhook]
        Alert --> Slack[Slack]
    end
```

监控维度:

1. **基础设施监控**:
   - 节点资源(CPU/内存/磁盘)
   - 网络流量
   - 存储IOPS
2. **平台监控**:
   - APIServer QPS/延迟
   - Controller调谐次数/耗时
   - etcd性能指标
3. **业务监控**:
   - 集群数量/状态
   - 应用部署数量/成功率
   - 组件安装数量/成功率
4. **用户体验监控**:
   - 页面加载时间
   - API响应时间
   - 错误率

12.7.2 日志架构

日志收集: - Promtail采集 - Loki聚合 - Grafana查询

日志分类: - 应用日志 - 审计日志 - 访问日志 - 系统日志

12.8 部署架构

12.8.1 组件部署

Namespace划分: - `edge-system`: 平台核心组件 - `observability-system`: 监控组件 - `vcluster-*`: 虚拟集群

资源配额:

组件	CPU请求	内存请求	CPU限制	内存限制
APIServer	500m	512Mi	2000m	2Gi
Controller	200m	256Mi	1000m	1Gi
Console	100m	128Mi	500m	512Mi
ChartMuseum	100m	128Mi	500m	512Mi

下一章节: 4.3 数据库架构

ewpage

13 4.3 数据库架构

13.1 存储架构总览

Edge Platform采用**Kubernetes Native**存储架构，将元数据存储在etcd中，动态统计数据通过Prometheus采集。

13.1.1 核心设计理念

```
传统K8s平台: 所有数据 → etcd
Edge Platform方案: 元数据 → etcd, 统计数据 → Prometheus
```

优势:
- 减少etcd的写入压力
- 利用Prometheus的时序数据能力
- 支持历史趋势分析

13.2 数据分类

13.2.1 1. 元数据存储 (etcd)

存储内容: - CRD定义和实例 - 权限配置(RoleTemplate, IAMRole) - 资源期望状态(Cluster, Application) - 配置信息

存储路径:

```
/registry/
├── iam.theriseunion.io/
```

```
│   ├── roletemplates/          # 角色模板
│   ├── iamroles/               # IAM角色
│   └── iamrolebindings/        # 角色绑定
├── scope.theriseunion.io/
│   ├── clusters/               # 集群定义
│   ├── workspaces/             # 工作空间
│   └── nodegroups/             # 节点组
├── app.theriseunion.io/
│   ├── applications/           # 应用元数据
│   ├── applicationversions/    # 应用版本
│   └── applicationdeployments/ # 部署配置
└── ext.theriseunion.io/
    └── components/             # 组件定义
```

性能优化: - Protobuf序列化(比JSON小50%) - Watch缓存(减少List操作) - Compact自动压缩(每小时) - 分片存储(按API Group)

13.2.2 2. 时序数据存储 (Prometheus)

数据存储方案

Edge Platform采用合适的数据存储策略：

传统**方案**: - 在CRD Status中维护统计信息 - 适合存储关键状态数据

Edge Platform方案: - 将动态指标通过Prometheus暴露和查询 - 利用Prometheus的时序数据特性

技术特点: - 减少etcd不必要的写入 - 支持历史数据查询 - 利用PromQL进行数据聚合

Metrics分类:

1. **系统指标** (System Metrics):
   - `apiserver_request_total`: API请求总数
   - `apiserver_request_duration_seconds`: API延迟分布
   - `controller_reconcile_total`: Controller调谐次数
   - `controller_reconcile_duration_seconds`: 调谐耗时
   - `etcd_db_total_size_bytes`: etcd数据库大小
2. 业务**指标** (Business Metrics):
   - `cluster_total{type="physical|virtual"}`: 集群总数
   - `cluster_status{status="running|failed|creating"}`: 集群状态分布
   - `vcluster_creation_duration_seconds`: 虚拟集群创建耗时
   - `app_deployment_total`: 应用部署总数
   - `app_deployment_status{status="success|failed"}`: 部署成功率
   - `component_install_duration_seconds`: 组件安装耗时
3. 资**源指标** (Resource Metrics):
   - `node_cpu_usage_percent`: 节点CPU使用率
   - `node_memory_usage_bytes`: 节点内存使用量
   - `pod_count{namespace="xxx"}`: Pod数量
   - `pv_usage_percent`: 存储使用率

数据保留策略: - 原始数据: 15天 - 5分钟聚合: 90天 - 1小时聚合: 1年 - 长期存储: Thanos(可选)

13.2.3 3. 日志数据存储 (Loki)

日志分类:

1. 审计**日志** (Audit Logs):
   - 用户操作记录
   - 资源变更历史
   - 权限检查结果
   - 保留期: 1年
2. 应**用日志** (Application Logs):
   - APIServer访问日志
   - Controller调谐日志
   - 错误和异常日志
   - 保留期: 30天
3. **系**统**日志** (System Logs):
   - K8s组件日志
   - vcluster运行日志
   - 边缘运行时日志
   - 保留期: 7天

日志索引:

```
{app="apiserver", level="error"}
{controller="cluster", namespace="edge-system"}
{component="vcluster", cluster_name="prod-01"}
```

13.2.4 4. Chart存储 (ChartMuseum)

每集群独立Chart仓库

方案对比: - 传统方案：所有集群依赖中心ChartMuseum - Edge Platform：每个集群独立的ChartMuseum

优势: - 零跨集群依赖 - 支持完全离线部署 - 故障隔离能力强

存储内容: - Helm Charts (tgz格式) - Chart索引 (index.yaml) - Chart版本元数据

存储位置: - 镜像内置: 常用组件预置 - PV持久化: 用户上传Chart - 容量: 20Gi (可扩展)

13.3 数据一致性

13.3.1 强一致性 (etcd)

场**景**: 权限配置、资源定义

保证: - Raft协议保证多副本一致性 - Watch机制实时同步变更 - Linearizable Read保证最新数据

示例:

用户创建IAMRole → etcd写入 → Watch触发 → 权限立即生效

13.3.2 最终一致性 (Prometheus)

场景: 统计数据、监控指标

保证: - 15秒采集间隔 - 允许短暂延迟 - 趋势分析无影响

示例:

集群状态变化 → Controller更新Metrics → Prometheus抓取(15s延迟) → UI显示

13.4 数据备份与恢复

13.4.1 etcd备份策略

自动备份: - 频率: 每日凌晨3点 - 方式: etcd snapshot - 存储: PV + 对象存储 - 保留: 最近30天

手动备份:

```
# 创建快照
ETCDCTL_API=3 etcdctl snapshot save backup.db

# 验证快照
etcdctl snapshot status backup.db
```

恢复流程:

```
# 停止etcd
systemctl stop etcd

# 恢复数据
etcdctl snapshot restore backup.db --data-dir=/var/lib/etcd-restore

# 更新配置并重启
systemctl start etcd
```

13.4.2 Prometheus备份策略

TSDB本地保留: - 15天高精度数据 - 自动Compact压缩

长期存储(可选): - Thanos集成 - 对象存储归档 - 数据压缩比: 10:1

13.4.3 ChartMuseum备份

PV快照: - 频率: 每周 - 方式: Volume Snapshot - 保留: 4周

镜像内置: - 核心组件随镜像分发 - 无需额外备份

13.5 数据安全

13.5.1 访问控制

etcd安全: - mTLS认证 - RBAC权限控制 - IP白名单

Prometheus安全: - OAuth 2.0认证 - 只读访问权限 - 数据脱敏

13.5.2 数据加密

静态加密: - etcd数据加密(可选) - PV加密(StorageClass配置) - 备份文件加密

传输**加密**: - TLS 1.3 - mTLS(内部通信) - 证书自动轮转

13.5.3 审计与合规

审计**日志**: - 所有API操作记录 - 资源变更历史 - 权限变更追踪

合规要求: - 数据保留策略符合法规 - 敏感信息脱敏 - 删除操作不可恢复

13.6 性能优化

13.6.1 写入优化

etcd写入优化: - 将统计数据存储在Prometheus，减少etcd写入 - 批量操作合并 - 异步状态更新

Prometheus写入优化: - Remote Write批量发送 - 数据压缩 - 队列缓冲

13.6.2 读取优化

三级缓存架构:

```
L1缓存(内存) → 命中率95%, <1ms
    ↓ Miss
L2缓存(本地) → 命中率4%, <10ms
    ↓ Miss
L3存储(etcd/Prometheus) → 命中率1%, <100ms
```

Watch机制: - 减少List操作 - 实时数据更新 - 降低etcd负载

13.6.3 查询优化

Prometheus查询优化: - Recording Rules预聚合 - Query Cache - 查询并行化

示例Recording Rule:

```yaml
# 预计算集群总数
- record: cluster:count:total
  expr: count(cluster_info)
```

```
# 预计算部署成功率
- record: deployment:success:rate
  expr: sum(app_deployment_status{status="success"}) /
          sum(app_deployment_total)
```

13.7 容量规划

13.7.1 etcd容量

数据增长估算: - 每个集群: ~100KB - 每个应用: ~50KB - 每个用户: ~10KB

容量建议: - 小型(< 50集群): 10Gi - 中型(50-200集群): 50Gi - 大型(> 200集群): 100Gi

13.7.2 Prometheus容量

数据增长估算: - 每秒样本数: ~10000 - 每天数据量: ~2GB - 15天数据: ~30GB

容量建议: - 小型环境: 100Gi - 中型环境: 500Gi - 大型环境: 2Ti + Thanos

13.7.3 ChartMuseum容量

存储需求: - 每个Chart: ~10-50MB - 预置Charts: ~500MB - 用户Charts: ~10GB

容量建议: - 标准配置: 20Gi - 扩展配置: 100Gi

13.8 数据迁移

13.8.1 跨集群迁移

导出数据:

```
# 导出所有CRD资源
kubectl get applications -o yaml > applications.yaml
kubectl get clusters -o yaml > clusters.yaml
```

导入数据:

```
# 导入到新集群
kubectl apply -f applications.yaml
kubectl apply -f clusters.yaml
```

13.8.2 版本升级迁移

CRD版本转换: - 自动转换(Conversion Webhook) - 手动迁移脚本 - 双写兼容期

数据兼容性: - 向后兼容保证 - 废弃字段保留期 - 迁移工具提供

下一章节: 第五章 部署架构

ewpage

14 第五章 部署架构

15 5.1 最小化部署

15.1 部署目标

最小化部署适用于开发测试环境、POC验证、快速体验等场景，以最少的资源快速搭建 Edge Platform。

15.1.1 适用场景

- 开发测试环境
- 产品功能验证
- 技术选型评估
- 培训演示环境

15.1.2 不适用场景

❌ 生产环境 ❌ 高可用要求 ❌ 大规模集群管理

15.2 资源需求

15.2.1 硬件要求

宿主集群最小配置:

组件	CPU	内存	磁盘	数量
Master节点	4核	8GB	100GB	1
Worker节点	4核	8GB	100GB	2
总计	12核	24GB	300GB	3台

组件资源占用:

组件	CPU	内存	说明
APIServer	500m	512Mi	单副本
Controller	200m	256Mi	单副本
Console	100m	128Mi	单副本
ChartMuseum	100m	128Mi	单副本
Prometheus	500m	2Gi	单副本
Loki	200m	512Mi	单副本
Grafana	100m	256Mi	单副本

组件	CPU	内存	说明
总计	1.7核	3.8GB	-

15.2.2 软件依赖

必需组件: - Kubernetes 1.26+ - Helm 3.12+ - kubectl 1.26+

可选组件: - Prometheus Operator (推荐) - Loki Stack (可选) - Ingress Controller (推荐)

15.3 部署架构

15.3.1 拓扑图

```
graph TB
    subgraph 外部访问
        User[用户浏览器]
    end

    subgraph K8s集群
        Ingress[Ingress Controller]

        subgraph edge-system
            API[APIServer Pod]
            Ctrl[Controller Pod]
            Console[Console Pod]
            Chart[ChartMuseum Pod]
        end

        subgraph observability-system
            Prom[Prometheus Pod]
            Loki[Loki Pod]
            Grafana[Grafana Pod]
        end

        subgraph 存储
            etcd[(etcd)]
            PV1[(API PV)]
            PV2[(Prom PV)]
        end
    end

    User --> Ingress
    Ingress --> Console
    Ingress --> API
    Ingress --> Grafana

    Console --> API
    API --> etcd
    Ctrl --> etcd
    API --> Prom

    API --> PV1
    Chart --> PV1
    Prom --> PV2
```

15.3.2 组件配置

单副本部署:

```
# edge-platform最小化配置
apiserver:
  replicaCount: 1
  resources:
    requests:
      cpu: 500m
      memory: 512Mi
    limits:
      cpu: 2000m
      memory: 2Gi

controller:
  replicaCount: 1
  resources:
    requests:
      cpu: 200m
      memory: 256Mi
    limits:
      cpu: 1000m
      memory: 1Gi

console:
  replicaCount: 1
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 500m
      memory: 512Mi
```

15.4 部署步骤

15.4.1 前置准备

1. 准备Kubernetes集群:

```
# 验证集群状态
kubectl cluster-info
kubectl get nodes

# 确保至少3个节点Ready
kubectl get nodes
NAME     STATUS   ROLES           AGE   VERSION
```

```
master   Ready    control-plane   10d   v1.26.0
node-1   Ready    <none>          10d   v1.26.0
node-2   Ready    <none>          10d   v1.26.0
```

2. 安装Helm:

```
# 下载Helm 3.12+
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
          bash

# 验证安装
helm version
```

3. 配置kubectl:

```
# 配置kubeconfig
export KUBECONFIG=/path/to/kubeconfig

# 验证访问
kubectl get ns
```

15.4.2 安装Edge Platform

方法一: Helm快速安装 (推荐)

```
# 1. 添加Helm仓库
helm repo add edge-platform https://charts.theriseunion.io
helm repo update

# 2. 创建命名空间
kubectl create namespace edge-system

# 3. 安装Edge Platform(最小化配置)
helm install edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --set apiserver.replicaCount=1 \
  --set controller.replicaCount=1 \
  --set console.replicaCount=1 \
  --set ha.enabled=false

# 4. 等待部署完成
kubectl wait --for=condition=ready pod \
  -l app.kubernetes.io/name=edge-platform \
  -n edge-system \
  --timeout=300s
```

方法二: 离线镜像安装

```
# 1. 加载镜像
docker load -i edge-platform-v1.0.0.tar

# 2. 推送到私有仓库(可选)
docker tag edge-platform:v1.0.0 myregistry.com/edge-platform:v1.0.0
docker push myregistry.com/edge-platform:v1.0.0

# 3. 使用私有镜像安装
helm install edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --set image.repository=myregistry.com/edge-platform \
  --set image.tag=v1.0.0
```

15.4.3 安装监控组件(可选)

```
# 1. 安装Prometheus Operator
helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts

helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace observability-system \
  --create-namespace \
  --set prometheus.prometheusSpec.retention=15d \
  --set prometheus.prometheusSpec.storageSpec.volumeClaimTemplate.spec.resources.requests

# 2. 配置ServiceMonitor
kubectl apply -f - <<EOF
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: edge-platform
  namespace: edge-system
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: edge-platform
  endpoints:
  - port: metrics
    interval: 30s
EOF
```

15.4.4 配置访问入口

方法一: NodePort (最简单)

```
# 修改Console Service为NodePort
kubectl patch svc edge-console -n edge-system \
  -p '{"spec":{"type":"NodePort"}}'
```

```
# 获取访问端口
NODE_PORT=$(kubectl get svc edge-console -n edge-system \
  -o jsonpath='{.spec.ports[0].nodePort}')

# 访问地址
echo "http://<任意节点IP>:$NODE_PORT"
```

方法二: Ingress (推荐)

```
# 1. 安装Ingress Controller(如果没有)
helm repo add ingress-nginx \
  https://kubernetes.github.io/ingress-nginx

helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx \
  --create-namespace

# 2. 创建Ingress规则
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: edge-platform
  namespace: edge-system
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: <您的域名>
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: edge-console
            port:
              number: 80
      - path: /kapis
        pathType: Prefix
        backend:
          service:
            name: edge-apiserver
            port:
              number: 9090
EOF
```

```
# 3. 配置DNS或hosts文件
# 在hosts文件中添加域名映射
```

15.5 验证部署

15.5.1 检查组件状态

```
# 1. 检查Pod状态
kubectl get pods -n edge-system
NAME                                READY   STATUS    RESTARTS   AGE
edge-apiserver-xxx                  1/1     Running   0          5m
edge-controller-xxx                 1/1     Running   0          5m
edge-console-xxx                    1/1     Running   0          5m
chartmuseum-xxx                     1/1     Running   0          5m

# 2. 检查Service
kubectl get svc -n edge-system
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
edge-apiserver    ClusterIP   10.96.0.10      <none>        9090/TCP
edge-console      ClusterIP   10.96.0.11      <none>        80/TCP

# 3. 检查存储
kubectl get pvc -n edge-system
NAME                  STATUS   VOLUME   CAPACITY   ACCESS MODES
chartmuseum-data      Bound    pv-001   20Gi       RWO
```

15.5.2 功能验证

1. 访问Console:

```
# 浏览器访问
http://<控制台访问地址>

# 默认账号
用户名: admin
密码: P@88w0rd
```

2. API测试:

```
# 获取Token
TOKEN=$(curl -s -X POST http://<API地址>/oauth/token \
  -d "grant_type=password&username=admin&password=P@88w0rd&client_id=edge-
          console&client_secret=edge-secret" \
  | jq -r '.access_token')

# 测试API
```

```
curl -H "Authorization: Bearer $TOKEN" \
  http://<API地址>/oapis/iam.theriseunion.io/v1alpha1/users
```

3. 创建测试集群:

```
# 创建虚拟集群
kubectl apply -f - <<EOF
apiVersion: scope.theriseunion.io/v1alpha1
kind: Cluster
metadata:
  name: test-cluster
spec:
  type: virtual
  distribution: k3s
  version: v1.26.0
EOF

# 等待就绪(约30秒)
kubectl wait --for=condition=Ready cluster/test-cluster --timeout=300s

# 获取kubeconfig
kubectl get cluster test-cluster -o jsonpath='{.spec.connection.kubeconfig}' |
          base64 -d > test-cluster.kubeconfig

# 访问虚拟集群
kubectl --kubeconfig=test-cluster.kubeconfig get nodes
```

15.6 性能指标

15.6.1 预期性能

资源使用: - CPU使用率: 10-20% - 内存使用率: 30-40% - 磁盘IO: < 10MB/s

响应性能: - API P50延迟: < 50ms - API P99延迟: < 200ms - 虚拟集群创建: 30-60秒 - 应用部署: 2-5分钟

容量限制: - 最大虚拟集群数: 10 - 最大应用数: 50 - 最大用户数: 100

15.7 常见问题

15.7.1 部署失败

问题: Pod一直Pending

```
# 检查资源不足
kubectl describe pod <pod-name> -n edge-system

# 解决: 降低资源请求
```

```
helm upgrade edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --set apiserver.resources.requests.cpu=250m \
  --set apiserver.resources.requests.memory=256Mi
```

问题: 镜像拉取失败

```
# 检查镜像拉取策略
kubectl get pod <pod-name> -n edge-system -o yaml | grep imagePullPolicy

# 解决: 使用本地镜像或配置私有仓库
helm upgrade edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --set image.pullPolicy=IfNotPresent
```

15.7.2 访问问题

问题: 无法访问Console

```
# 检查Ingress状态
kubectl get ingress -n edge-system
kubectl describe ingress edge-platform -n edge-system

# 检查Ingress Controller
kubectl get pods -n ingress-nginx

# 解决: 使用NodePort临时访问
kubectl patch svc edge-console -n edge-system -p '{"spec":
          {"type":"NodePort"}}'
```

15.7.3 功能问题

问题: 虚拟集群创建失败

```
# 检查Controller日志
kubectl logs -n edge-system -l app=edge-controller --tail=100

# 检查vcluster Helm
helm list -A | grep vcluster

# 解决: 检查ChartMuseum是否正常
kubectl get pods -n edge-system -l app=chartmuseum
```

15.8 升级与卸载

15.8.1 升级

```
# 更新Helm仓库
helm repo update

# 升级到最新版本
helm upgrade edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --reuse-values
```

15.8.2 卸载

```
# 删除Helm release
helm uninstall edge-platform -n edge-system

# 清理PVC(可选,数据会丢失)
kubectl delete pvc -n edge-system --all

# 删除命名空间
kubectl delete namespace edge-system
```

下一章节: 5.2 高可用部署

ewpage

16 5.2 高可用部署

16.1 部署目标

高可用部署适用于**生产环境**、**关键业务系统**、**大规模集群管理**等场景,提供企业级高可用保障。

16.1.1 适用场景

✅ 生产环境 ✅ 关键业务系统 ✅ 大规模集群管理(50+集群) ✅ SLA要求高(99.9%+)

16.1.2 高可用目标

- **可用性**: 99.9% (年停机时间< 8.76小时)
- **故障恢复**: 自动故障转移,RTO < 5分钟
- **数据安全**: 多副本,RPO < 1分钟
- **性能保障**: 无单点瓶颈,支持水平扩展

16.2 资源需求

16.2.1 硬件要求

宿主集群生产配置:

组件	CPU	内存	磁盘	数量
Master节点	8核	16GB	200GB SSD	3
Worker节点	16核	32GB	500GB SSD	5+
总计	104核+	208GB+	3.1TB+	8台+

组件资源占用(高可用):

组件	CPU请求	内存请求	副本数	总计
APIServer	1000m	1Gi	3	3核/3GB
Controller	500m	512Mi	2(Leader选举)	1核/1GB
Console	200m	256Mi	2	400m/512MB
ChartMuseum	200m	256Mi	1	200m/256MB
etcd	2000m	4Gi	3	6核/12GB
Prometheus	2000m	8Gi	2	4核/16GB
总计	-	-	-	14.6核/32.8GB

16.2.2 网络要求

带宽: - 节点间: 10Gbps+ - 对外出口: 1Gbps+

延迟: - 节点间: < 1ms - etcd节点间: < 10ms

端口开放: - 80/443 (HTTP/HTTPS) - 6443 (K8s API) - 2379-2380 (etcd) - 9090 (APIServer Metrics)

16.3 高可用架构

16.3.1 拓扑图

```
graph TB
    subgraph 外部访问层
        LB[负载均衡器<br/>VIP/DNS]
    end

    subgraph Ingress层
        Ing1[Ingress 1]
        Ing2[Ingress 2]
    end

    subgraph 应用层
        subgraph APIServer集群
            API1[APIServer 1]
            API2[APIServer 2]
```

```
        API3[APIServer 3]
    end

    subgraph Controller集群
        Ctrl1[Controller 1<br/>Leader]
        Ctrl2[Controller 2<br/>Standby]
    end

    subgraph Console集群
        Con1[Console 1]
        Con2[Console 2]
    end
end

subgraph 存储层
    subgraph etcd集群
        E1[etcd 1]
        E2[etcd 2]
        E3[etcd 3]
    end

    subgraph Prometheus集群
        P1[Prometheus 1]
        P2[Prometheus 2]
    end
end

LB --> Ing1
LB --> Ing2

Ing1 --> API1
Ing1 --> API2
Ing1 --> API3
Ing2 --> API1
Ing2 --> API2
Ing2 --> API3

Ing1 --> Con1
Ing1 --> Con2
Ing2 --> Con1
Ing2 --> Con2

API1 --> E1
API1 --> E2
API1 --> E3
API2 --> E1
API2 --> E2
API2 --> E3
API3 --> E1
API3 --> E2
API3 --> E3

Ctrl1 --> E1
Ctrl1 --> E2
Ctrl1 --> E3
Ctrl2 --> E1
Ctrl2 --> E2
```

```
    Ctrl2 --> E3

    API1 --> P1
    API2 --> P1
    API3 --> P1
    P1 --> P2
```

16.3.2 高可用机制

16.3.2.1 1. APIServer高可用

多副本 + Service负载均衡: - 3个副本分布在不同节点 - K8s Service自动负载均衡 - 健康检查自动摘除故障实例

配置:

```yaml
apiserver:
  replicaCount: 3
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: edge-apiserver
      topologyKey: kubernetes.io/hostname
```

16.3.2.2 2. Controller高可用

Leader选举机制: - 2个副本,1个Leader + 1个Standby - 基于K8s Lease实现Leader选举 - Leader故障时自动切换(< 15秒)

配置:

```yaml
controller:
  replicaCount: 2
  leaderElection:
    enabled: true
    leaseDuration: 15s
    renewDeadline: 10s
    retryPeriod: 2s
```

16.3.2.3 3. etcd高可用

Raft一致性协议: - 3节点etcd集群 - 容忍1个节点故障 - 强一致性保证

配置:

```yaml
etcd:
  replicaCount: 3
  persistence:
    enabled: true
```

```
    size: 100Gi
    storageClass: fast-ssd
```

16.3.2.4 4. Prometheus高可用

联邦 + Thanos: - 2个Prometheus实例独立采集 - Thanos聚合查询 - 长期存储在对象存储

配置:

```
prometheus:
  replicaCount: 2
  thanos:
    enabled: true
    objectStorage:
      type: s3
      config:
        bucket: edge-platform-metrics
```

16.4 部署步骤

16.4.1 1. 前置准备

准备高可用K8s集群:

```
# 验证Master节点(至少3个)
kubectl get nodes -l node-role.kubernetes.io/master
NAME       STATUS   ROLES    AGE
master-1   Ready    master   30d
master-2   Ready    master   30d
master-3   Ready    master   30d

# 验证Worker节点(至少5个)
kubectl get nodes -l '!node-role.kubernetes.io/master'
NAME       STATUS   ROLES    AGE
worker-1   Ready    <none>   30d
worker-2   Ready    <none>   30d
worker-3   Ready    <none>   30d
worker-4   Ready    <none>   30d
worker-5   Ready    <none>   30d
```

配置StorageClass:

```
# 创建高性能StorageClass
kubectl apply -f - <<EOF
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
```

```
provisioner: kubernetes.io/csi-driver
parameters:
  type: pd-ssd
  fsType: ext4
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
EOF
```

16.4.2 2. 安装Edge Platform

高可用配置文件 (ha-values.yaml):

```
# 高可用配置
ha:
  enabled: true

# APIServer高可用
apiserver:
  replicaCount: 3
  resources:
    requests:
      cpu: 1000m
      memory: 1Gi
    limits:
      cpu: 4000m
      memory: 4Gi
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: edge-apiserver
      topologyKey: kubernetes.io/hostname
  autoscaling:
    enabled: true
    minReplicas: 3
    maxReplicas: 10
    targetCPUUtilizationPercentage: 70

# Controller高可用
controller:
  replicaCount: 2
  leaderElection:
    enabled: true
  resources:
    requests:
      cpu: 500m
      memory: 512Mi
    limits:
```

```yaml
      cpu: 2000m
      memory: 2Gi

# Console高可用
console:
  replicaCount: 2
  resources:
    requests:
      cpu: 200m
      memory: 256Mi
    limits:
      cpu: 1000m
      memory: 1Gi

# etcd高可用
etcd:
  enabled: true
  replicaCount: 3
  persistence:
    enabled: true
    size: 100Gi
    storageClass: fast-ssd
  resources:
    requests:
      cpu: 2000m
      memory: 4Gi
    limits:
      cpu: 4000m
      memory: 8Gi

# Prometheus高可用
prometheus:
  replicaCount: 2
  retention: 15d
  storageSpec:
    volumeClaimTemplate:
      spec:
        storageClassName: fast-ssd
        resources:
          requests:
            storage: 500Gi
  thanos:
    enabled: true
    objectStorageConfig:
      name: thanos-objstore-config
      key: objstore.yml
  resources:
    requests:
```

```
      cpu: 2000m
      memory: 8Gi
    limits:
      cpu: 4000m
      memory: 16Gi

# ChartMuseum
chartmuseum:
  persistence:
    enabled: true
    size: 50Gi
    storageClass: fast-ssd
```

执行安装:

```
# 添加Helm仓库
helm repo add edge-platform https://charts.theriseunion.io
helm repo update

# 创建命名空间
kubectl create namespace edge-system

# 安装高可用配置
helm install edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --values ha-values.yaml \
  --timeout 30m

# 等待所有组件就绪
kubectl wait --for=condition=ready pod \
  -l app.kubernetes.io/name=edge-platform \
  -n edge-system \
  --timeout=600s
```

16.4.3 3. 配置负载均衡

方法一: 云负载均衡器 (推荐)

```
# AWS ALB示例
kubectl apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: edge-platform-lb
  namespace: edge-system
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
    service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-
          enabled: "true"
```

```
spec:
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: ingress-nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
  - name: https
    port: 443
    targetPort: 443
EOF
```

方法二: Keepalived + HAProxy

```
# 部署HAProxy
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: haproxy
  namespace: edge-system
spec:
  selector:
    matchLabels:
      app: haproxy
  template:
    metadata:
      labels:
        app: haproxy
    spec:
      hostNetwork: true
      containers:
      - name: haproxy
        image: haproxy:2.6
        ports:
        - containerPort: 80
        - containerPort: 443
        volumeMounts:
        - name: config
          mountPath: /usr/local/etc/haproxy
      volumes:
      - name: config
        configMap:
          name: haproxy-config
---
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: haproxy-config
  namespace: edge-system
data:
  haproxy.cfg: |
    global
      maxconn 50000
      log stdout format raw local0

    defaults
      mode http
      timeout connect 5s
      timeout client 30s
      timeout server 30s

    frontend http-in
      bind *:80
      bind *:443 ssl crt /etc/ssl/edge-platform.pem
      default_backend edge-console

    backend edge-console
      balance roundrobin
      option httpchk GET /health
      server console-1 10.96.0.11:80 check
      server console-2 10.96.0.12:80 check

    backend edge-apiserver
      balance roundrobin
      option httpchk GET /healthz
      server api-1 10.96.0.21:9090 check
      server api-2 10.96.0.22:9090 check
      server api-3 10.96.0.23:9090 check
EOF
```

16.4.4 4. 配置备份

etcd自动备份:

```
# 部署etcd备份CronJob
kubectl apply -f - <<EOF
apiVersion: batch/v1
kind: CronJob
metadata:
  name: etcd-backup
  namespace edge-system
spec:
  schedule: "0 3 * * *"  # 每天凌晨3点
  jobTemplate:
    spec:
      template:
```

```
        spec:
          containers:
          - name: backup
            image: quay.io/coreos/etcd:v3.5.9
            command:
            - /bin/sh
            - -c
            - |
              ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-$(date +%Y%m%d-
          %H%M%S).db \
                --endpoints=https://etcd-0.etcd:2379,https://
          etcd-1.etcd:2379,https://etcd-2.etcd:2379 \
                --cacert=/etc/etcd/ca.crt \
                --cert=/etc/etcd/client.crt \
                --key=/etc/etcd/client.key

              # 保留最近30天备份
              find /backup -name "etcd-*.db" -mtime +30 -delete
            volumeMounts:
            - name: backup
              mountPath: /backup
            - name: etcd-certs
              mountPath: /etc/etcd
          restartPolicy: OnFailure
          volumes:
          - name: backup
            persistentVolumeClaim:
              claimName: etcd-backup-pvc
          - name: etcd-certs
            secret:
              secretName: etcd-client-certs
EOF
```

16.5 验证高可用

16.5.1 1. 故障注入测试

测试APIServer高可用:

```
# 删除一个APIServer Pod
kubectl delete pod -n edge-system -l app=edge-apiserver --force --grace-
          period=0 | head -1

# 验证服务不中断
while true; do
  curl -s -o /dev/null -w "%{http_code}\n" http://<负载均衡器地址>/health
  sleep 1
```

```
done
# 应持续返回200
```

测试Controller高可用:

```
# 查看当前Leader
kubectl get lease -n edge-system edge-controller-leader -o yaml

# 删除Leader Pod
LEADER_POD=$(kubectl get lease -n edge-system edge-controller-leader \
  -o jsonpath='{.spec.holderIdentity}')
kubectl delete pod -n edge-system $LEADER_POD --force --grace-period=0

# 验证15秒内选出新Leader
kubectl get lease -n edge-system edge-controller-leader -w
```

测试etcd高可用:

```
# 停止一个etcd成员
kubectl exec -n edge-system etcd-0 -- kill 1

# 验证集群仍可写入
kubectl apply -f - <<EOF
apiVersion: scope.theriseunion.io/v1alpha1
kind: Cluster
metadata:
  name: ha-test-cluster
spec:
  type: virtual
EOF

# 验证集群健康
kubectl exec -n edge-system etcd-1 -- etcdctl endpoint health \
  --endpoints=https://etcd-0.etcd:2379,https://etcd-1.etcd:2379,https://
          etcd-2.etcd:2379 \
  --cacert=/etc/etcd/ca.crt --cert=/etc/etcd/client.crt --key=/etc/etcd/
          client.key
```

16.5.2 2. 性能测试

API压力测试:

```
# 使用Apache Bench
ab -n 10000 -c 100 -H "Authorization: Bearer $TOKEN" \
  http://<负载均衡器地址>/oapis/iam.theriseunion.io/v1alpha1/users

# 预期结果
# Requests per second: > 1000
```

```
# Time per request: < 100ms (mean)
# Failed requests: 0
```

并发集群创建测试:

```
# 并发创建20个虚拟集群
for i in {1..20}; do
  kubectl apply -f - <<EOF &
apiVersion: scope.theriseunion.io/v1alpha1
kind: Cluster
metadata:
  name: perf-test-cluster-$i
spec:
  type: virtual
  distribution: k3s
EOF
done

wait

# 验证全部成功
kubectl get clusters | grep perf-test | wc -l
# 应返回20
```

16.6 监控与告警

16.6.1 关键指标监控

创建PrometheusRule:

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: edge-platform-ha
  namespace: edge-system
spec:
  groups:
  - name: edge-platform.ha
    interval: 30s
    rules:
    # APIServer可用性
    - alert: APIServerDown
      expr: up{job="edge-apiserver"} == 0
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: "APIServer实例不可用"
```

```
      description: "{{ $labels.pod }} 已停止响应超过1分钟"

  # APIServer低于2个副本
  - alert: APIServerLowReplicas
    expr: count(up{job="edge-apiserver"} == 1) < 2
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "APIServer副本数不足"
      description: "当前仅{{ $value }}个APIServer副本在运行"

  # Controller无Leader
  - alert: ControllerNoLeader
    expr: sum(leader_election_master_status{job="edge-controller"}) == 0
    for: 1m
    labels:
      severity: critical
    annotations:
      summary: "Controller无Leader"
      description: "Controller集群超过1分钟没有Leader"

  # etcd成员不足
  - alert: EtcdInsufficientMembers
    expr: count(up{job="etcd"} == 1) < 2
    for: 3m
    labels:
      severity: critical
    annotations:
      summary: "etcd集群成员不足"
      description: "etcd集群仅{{ $value }}个成员,无法容忍故障"

  # API延迟过高
  - alert: APIHighLatency
    expr: histogram_quantile(0.99,
        apiserver_request_duration_seconds_bucket) > 1
    for: 10m
    labels:
      severity: warning
    annotations:
      summary: "API P99延迟过高"
      description: "API P99延迟为{{ $value }}秒,超过1秒阈值"
```

16.6.2 告警通知

配置AlertManager:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: alertmanager-config
  namespace: edge-system
data:
  alertmanager.yml: |
    global:
      resolve_timeout: 5m

    route:
      group_by: ['alertname', 'cluster']
      group_wait: 10s
      group_interval: 10s
      repeat_interval: 12h
      receiver: 'default'
      routes:
      - match:
          severity: critical
        receiver: 'critical'

    receivers:
    - name: 'default'
      webhook_configs:
      - url: 'http://<您的webhook地址>/alerts'

    - name: 'critical'
      email_configs:
      - to: '<您的邮箱>'
        from: '<发件人邮箱>'
        smarthost: '<SMTP服务器>:587'
        auth_username: '<SMTP用户名>'
        auth_password: '***'
      slack_configs:
      - api_url: 'https://hooks.slack.com/services/xxx'
        channel: '#edge-platform-alerts'
```

16.7 灾难恢复

16.7.1 备份策略

全量备份清单: - etcd快照(每日) - PV快照(每周) - 配置文件(Git版本管理) - Helm values(Git版本管理)

16.7.2 恢复流程

场景1: etcd数据损坏

```
# 1. 停止所有组件
kubectl scale deploy -n edge-system --replicas=0 --all

# 2. 恢复etcd
kubectl exec -n edge-system etcd-0 -- sh -c "
  ETCDCTL_API=3 etcdctl snapshot restore /backup/etcd-20240115-030000.db \
    --data-dir=/var/lib/etcd-restore \
    --initial-cluster=etcd-0=https://etcd-0.etcd:2380,etcd-1=https://
          etcd-1.etcd:2380,etcd-2=https://etcd-2.etcd:2380 \
    --initial-advertise-peer-urls=https://etcd-0.etcd:2380 \
    --name=etcd-0
"

# 3. 重启etcd
kubectl delete pod -n edge-system -l app=etcd

# 4. 启动所有组件
kubectl scale deploy -n edge-system --replicas=3 edge-apiserver
kubectl scale deploy -n edge-system --replicas=2 edge-controller
```

场景2: 完全重建

```
# 1. 重新安装Edge Platform
helm install edge-platform edge-platform/edge-platform \
  --namespace edge-system \
  --values ha-values.yaml

# 2. 恢复etcd数据
kubectl cp backup/etcd-latest.db edge-system/etcd-0:/tmp/
kubectl exec -n edge-system etcd-0 -- etcdctl snapshot restore /tmp/etcd-
          latest.db

# 3. 重启所有Pod
kubectl rollout restart deploy -n edge-system
```

下一章节: 5.3 多集群部署

ewpage

17 5.3 边缘节点接入与管理

17.1 概述

边缘计算的核心价值在于将计算能力下沉到数据产生的源头。本章详细介绍边缘节点如何接入Edge Platform，以及如何利用多级权限体系实现大规模节点的有效管理。

17.2 边缘节点接入架构

17.2.1 接入流程概览

边缘节点接入Edge Platform遵循标准的Kubernetes边缘节点注册流程，通过Edge Platform的统一管理界面简化了复杂的配置过程。

核心组件： - **云端管理平台**：统一管理所有边缘节点 - **边缘运行时**：节点上的轻量级代理 - **安全隧道**：云端到边缘的安全通信 - **自动注册**：节点的自动发现和注册机制

17.2.2 节点类型

KubeEdge节点： - 适用于工业设备、IoT网关等资源受限环境 - 支持设备管理和边缘应用部署 - 提供云边协同能力

标准Kubernetes节点： - 适用于边缘服务器、小型数据中心 - 完整的Kubernetes功能 - 适合计算密集型应用

混合节点： - 同时支持标准工作负载和边缘应用 - 灵活适应不同业务场景 - 逐步迁移路径

17.3 节点接入步骤

17.3.1 1. 准备工作

环境要求： - 操作系统：Linux (Ubuntu 18.04+, CentOS 7+) - 架构：x86_64 或 ARM64 - 网络：能够访问云端API Server - 资源：最小 1GB RAM, 1 CPU core

网络配置： - 配置节点到云端的网络连接 - 确保防火墙允许必要的端口 - 配置DNS解析（如果需要）

17.3.2 2. 运行时安装

自动安装： 通过云端管理平台一键安装边缘运行时。用户只需访问管理界面，选择节点类型和连接参数，系统会自动生成安装脚本并执行安装过程。整个过程无需人工干预，支持批量安装和配置。

手动安装： 1. 下载边缘运行时二进制文件 2. 配置连接参数 3. 启动边缘代理服务 4. 验证与云端的连接

17.3.3 3. 节点注册

自动注册： - 边缘代理启动后自动连接云端 - 提交节点信息和规格数据 - 等待云端审批和授权

手动注册： 1. 在云端管理界面创建节点记录 2. 生成节点注册Token 3. 在边缘节点配置Token并启动服务

17.3.4 4. 权限配置

节点分组： - 按地理位置划分节点组（如：华北、华南） - 按业务类型划分（如：生产、测试） - 按管理团队划分（如：运维团队、业务团队）

权限分配： - 节点管理员权限：管理节点生命周期 - 应用部署权限：在指定节点部署应用 - 监控查看权限：查看节点运行状态

17.4 多级权限管理体系

17.4.1 权限模型应用

在边缘场景中，5层Scope权限模型的具体应用：

Platform级： - 管理所有边缘节点组 - 全局策略和合规要求 - 跨区域资源调度

Cluster级： - 管理特定区域的节点组 - 区域网络策略配置 - 区域合规要求

NodeGroup级： - 管理具体的节点组 - 节点组成员管理 - 节点组内应用部署

Node级： - 管理单个节点 - 节点配置调整 - 节点维护操作

Resource级： - 管理节点上的Pod、Service等 - 应用资源调度 - 存储和网络配置

17.4.2 权限分配实践

场景1：区域运维团队

区域运维管理员(北京) → Beijing-NodeGroup → 北京区域所有节点

权限内容：
- 管理北京区域所有边缘节点
- 在北京节点部署运维工具
- 监控北京节点运行状态
- 执行节点维护操作

场景2：业务应用团队

业务开发团队 → Production-Workspace → 生产环境节点

权限内容：
- 在生产节点部署业务应用
- 查看应用运行状态和日志
- 更新应用配置
- 回滚应用版本

场景3：设备管理团队

设备管理员 → Device-NodeGroup → IoT设备节点

权限内容：

- 管理IoT设备节点
- 更新设备固件
- 配置设备参数
- 监控设备状态

17.4.3 权限继承机制

自动级联：当用户在某个Scope没有明确权限时，系统会自动向上级Scope查找权限：

```
用户请求访问生产环境Pod
    ↓
Namespace级别：无权限
    ↓
Workspace级别：有Production-Workspace权限
    ↓
权限验证通过，允许访问
```

最小权限原则：- 默认无权限，需要明确授予 - 权限范围尽可能小 - 定期审计和清理不必要的权限

17.5 节点生命周期管理

17.5.1 节点上线

初始化流程：1. 节点硬件准备和系统安装 2. 网络配置和安全设置 3. 边缘运行时安装和启动 4. 节点注册和权限分配 5. 基础监控和日志配置

验证检查：- 网络连通性测试 - 运行时服务状态检查 - 云端通信验证 - 权限访问测试

17.5.2 节点维护

在线维护：- 应用更新和升级 - 配置参数调整 - 性能监控和优化 - 安全补丁安装

离线维护：- 节点重启操作 - 硬件升级更换 - 系统重装 - 故障排查

17.5.3 节点下线

计划下线：1. 停止新的应用部署 2. 迁移现有应用到其他节点 3. 停止边缘运行时服务 4. 从管理平台移除节点 5. 清理网络和权限配置

故障下线：- 自动检测节点离线 - 应用自动迁移 - 告警通知相关人员 - 节点状态更新

17.6 安全管理

17.6.1 身份认证

节点身份：- 每个节点都有唯一身份标识 - 使用证书或Token进行身份验证 - 支持证书自动轮换

通信安全：- 节点与云端通信加密 - 双向TLS认证 - 安全隧道传输

17.6.2 访问控制

网络隔离： - 节点组级别的网络隔离 - 防火墙规则配置 - VPN连接选项

权限隔离： - 基于Scope的权限隔离 - 最小权限原则 - 权限审计和追踪

17.7 监控与运维

17.7.1 节点监控

基础监控： - CPU、内存、磁盘使用率 - 网络连通性状态 - 边缘运行时服务状态 - 应用运行状态

业务监控： - 应用性能指标 - 业务流程监控 - 用户体验指标 - 错误率统计

17.7.2 日志管理

日志收集： - 系统日志：操作系统和运行时日志 - 应用日志：容器应用日志 - 审计日志：权限和操作日志 - 错误日志：异常和故障日志

日志分析： - 统一日志格式和存储 - 日志聚合和搜索 - 日志告警和报表 - 长期日志归档

17.7.3 故障处理

常见故障： - 网络连接中断 - 节点资源不足 - 应用启动失败 - 配置错误

处理流程： 1. 故障检测和告警 2. 故障诊断和定位 3. 故障修复和恢复 4. 故障分析和预防

上一章节: 5.2 高可用部署 **技术白皮书**: 返回首页

ewpage

18 5.4 配置参数参考

18.1 Helm Chart配置

18.1.1 全局配置

```yaml
# 全局配置
global:
  # 镜像仓库
  imageRegistry: quanzhenglong.com
  imagePullSecrets:
  - name: regcred

  # 存储类
```

```yaml
  storageClass: fast-ssd

  # 集群信息
  clusterName: edge-platform
  clusterRegion: beijing
```

18.1.2 APIServer配置

```yaml
apiserver:
  # 副本数
  replicaCount: 3

  # 镜像配置
  image:
    repository: edge/edge-apiserver
    tag: v1.0.0
    pullPolicy: IfNotPresent

  # 资源配置
  resources:
    requests:
      cpu: 1000m
      memory: 1Gi
    limits:
      cpu: 4000m
      memory: 4Gi

  # 自动扩缩容
  autoscaling:
    enabled: true
    minReplicas: 3
    maxReplicas: 10
    targetCPUUtilizationPercentage: 70
    targetMemoryUtilizationPercentage: 80

  # Pod反亲和性
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: edge-apiserver
      topologyKey: kubernetes.io/hostname

  # 认证配置
  authentication:
    # JWT配置
    jwt:
      secret: "your-jwt-secret"
```

```yaml
    issuer: "https://edge.example.com"
    expire: 24h

  # OIDC配置
  oidc:
    enabled: false
    issuerURL: "https://idp.example.com"
    clientID: "edge-platform"
    clientSecret: "***"
    redirectURL: "https://edge.example.com/oauth/callback"

  # LDAP配置
  ldap:
    enabled: false
    host: "ldap.example.com"
    port: 389
    baseDN: "dc=example,dc=com"
    bindDN: "cn=admin,dc=example,dc=com"
    bindPassword: "***"

# 授权配置
authorization:
  mode: "RBAC"  # AlwaysAllow, RBAC
  cacheSize: 10000
  cacheTTL: 300s

# API限流
rateLimit:
  enabled: true
  qps: 1000
  burst: 2000

# 服务配置
service:
  type: ClusterIP
  port: 9090
  annotations: {}

# Ingress配置
ingress:
  enabled: false
  className: nginx
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
  hosts:
  - host: edge.example.com
    paths:
    - path: /kapis
```

```
        pathType: Prefix
    tls:
    - secretName: edge-tls
      hosts:
      - edge.example.com
```

18.1.3 Controller配置

```yaml
controller:
  # 副本数
  replicaCount: 2

  # 镜像配置
  image:
    repository: edge/edge-controller
    tag: v1.0.0
    pullPolicy: IfNotPresent

  # 资源配置
  resources:
    requests:
      cpu: 500m
      memory: 512Mi
    limits:
      cpu: 2000m
      memory: 2Gi

  # Leader选举
  leaderElection:
    enabled: true
    leaseDuration: 15s
    renewDeadline: 10s
    retryPeriod: 2s

  # 并发控制
  concurrency:
    cluster: 3
    application: 5
    component: 3

  # 同步间隔
  syncPeriod: 30s

  # Webhook配置
  webhook:
    enabled: true
    port: 9443
    certDir: /tmp/k8s-webhook-server/serving-certs
```

18.1.4 Console配置

```yaml
console:
  # 副本数
  replicaCount: 2

  # 镜像配置
  image:
    repository: edge/edge-console
    tag: v1.0.0
    pullPolicy: IfNotPresent

  # 资源配置
  resources:
    requests:
      cpu: 200m
      memory: 256Mi
    limits:
      cpu: 1000m
      memory: 1Gi

  # 服务配置
  service:
    type: ClusterIP
    port: 80

  # Ingress配置
  ingress:
    enabled: true
    className: nginx
    hosts:
    - host: edge.example.com
      paths:
      - path: /
        pathType: Prefix
    tls:
    - secretName: edge-tls
      hosts:
      - edge.example.com

  # 环境变量
  env:
    API_SERVER_URL: "https://edge.example.com/kapis"
    OAUTH_CLIENT_ID: "edge-console"
    OAUTH_CLIENT_SECRET: "edge-secret"
```

18.1.5 etcd配置

```yaml
etcd:
  # 是否部署etcd(false则使用K8s自带etcd)
  enabled: false

  # 副本数
  replicaCount: 3

  # 镜像配置
  image:
    repository: quay.io/coreos/etcd
    tag: v3.5.9
    pullPolicy: IfNotPresent

  # 资源配置
  resources:
    requests:
      cpu: 2000m
      memory: 4Gi
    limits:
      cpu: 4000m
      memory: 8Gi

  # 持久化
  persistence:
    enabled: true
    size: 100Gi
    storageClass: fast-ssd

  # 备份
  backup:
    enabled: true
    schedule: "0 3 * * *"
    retention: 30
    storageClass: standard

  # 性能调优
  extraEnv:
  - name: ETCD_QUOTA_BACKEND_BYTES
    value: "8589934592"  # 8GB
  - name: ETCD_AUTO_COMPACTION_MODE
    value: "periodic"
  - name: ETCD_AUTO_COMPACTION_RETENTION
    value: "1h"
```

18.1.6 Prometheus配置

```yaml
prometheus:
  # 副本数
  replicaCount: 2

  # 数据保留期
  retention: 15d

  # 存储配置
  storageSpec:
    volumeClaimTemplate:
      spec:
        storageClassName: fast-ssd
        resources:
          requests:
            storage: 500Gi

  # 资源配置
  resources:
    requests:
      cpu: 2000m
      memory: 8Gi
    limits:
      cpu: 4000m
      memory: 16Gi

  # Thanos配置
  thanos:
    enabled: true
    version: v0.31.0
    objectStorageConfig:
      name: thanos-objstore-config
      key: objstore.yml

  # ServiceMonitor
  serviceMonitor:
    enabled: true
    interval: 30s

  # 告警规则
  additionalPrometheusRulesMap:
    edge-platform:
      groups:
      - name: edge-platform.rules
        interval: 30s
        rules:
        - alert: APIServerDown
```

```
        expr: up{job="edge-apiserver"} == 0
        for: 1m
        labels:
          severity: critical
```

18.1.7 ChartMuseum配置

```yaml
chartmuseum:
  # 副本数
  replicaCount: 1

  # 镜像配置
  image:
    repository: chartmuseum/chartmuseum
    tag: v0.16.1
    pullPolicy: IfNotPresent

  # 资源配置
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 500m
      memory: 512Mi

  # 持久化
  persistence:
    enabled: true
    size: 50Gi
    storageClass: standard

  # 环境变量
  env:
    STORAGE: local
    STORAGE_LOCAL_ROOTDIR: /charts
    DISABLE_API: false
    ALLOW_OVERWRITE: true
```

18.1.8 Loki配置

```yaml
loki:
  # 是否启用
  enabled: true

  # 副本数
  replicaCount: 1
```

```yaml
  # 数据保留期
  retention: 30d

  # 存储配置
  persistence:
    enabled: true
    size: 200Gi
    storageClass: standard

  # 资源配置
  resources:
    requests:
      cpu: 200m
      memory: 512Mi
    limits:
      cpu: 1000m
      memory: 2Gi

  # 配置
  config:
    auth_enabled: false
    ingester:
      chunk_idle_period: 3m
      chunk_retain_period: 1m
      max_chunk_age: 1h
    limits_config:
      enforce_metric_name: false
      reject_old_samples: true
      reject_old_samples_max_age: 168h
      ingestion_rate_mb: 10
      ingestion_burst_size_mb: 20
```

18.2 环境变量配置

18.2.1 APIServer环境变量

```yaml
env:
  # 基础配置
  - name: LOG_LEVEL
    value: "info"  # debug, info, warn, error
  - name: LOG_FORMAT
    value: "json"  # json, text
  - name: BIND_ADDRESS
    value: "0.0.0.0:9090"

  # 认证配置
  - name: JWT_SECRET
```

```
    valueFrom:
      secretKeyRef:
        name: edge-platform-secret
        key: jwt-secret
  - name: JWT_EXPIRE
    value: "24h"

  # 数据库配置
  - name: ETCD_ENDPOINTS
    value: "https://etcd-0.etcd:2379,https://etcd-1.etcd:2379,https://
          etcd-2.etcd:2379"
  - name: ETCD_CERT_FILE
    value: "/etc/etcd/client.crt"
  - name: ETCD_KEY_FILE
    value: "/etc/etcd/client.key"
  - name: ETCD_CA_FILE
    value: "/etc/etcd/ca.crt"

  # Prometheus配置
  - name: PROMETHEUS_ENDPOINT
    value: "http://prometheus:9090"

  # 性能配置
  - name: MAX_CONCURRENT_REQUESTS
    value: "1000"
  - name: REQUEST_TIMEOUT
    value: "30s"
```

18.2.2 Controller环境变量

```
env:
  # 基础配置
  - name: LOG_LEVEL
    value: "info"
  - name: METRICS_BIND_ADDRESS
    value: "0.0.0.0:8080"
  - name: HEALTH_PROBE_BIND_ADDRESS
    value: "0.0.0.0:8081"

  # Leader选举
  - name: LEADER_ELECT
    value: "true"
  - name: LEADER_ELECT_NAMESPACE
    value: "edge-system"

  # 并发控制
  - name: CLUSTER_CONCURRENT_RECONCILES
    value: "3"
```

```
  - name: APPLICATION_CONCURRENT_RECONCILES
    value: "5"

  # ChartMuseum配置
  - name: CHARTMUSEUM_URL
    value: "http://chartmuseum:8080"
```

18.3 存储配置

18.3.1 StorageClass配置

高性能SSD:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/aws-ebs  # 根据云厂商调整
parameters:
  type: gp3
  iops: "3000"
  throughput: "125"
  fsType: ext4
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

标**准存**储:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ext4
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

18.3.2 PVC模板

etcd PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: etcd-data
```

```
  namespace: edge-system
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: fast-ssd
  resources:
    requests:
      storage: 100Gi
```

Prometheus PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: prometheus-data
  namespace: edge-system
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: fast-ssd
  resources:
    requests:
      storage: 500Gi
```

18.4 网络配置

18.4.1 Service配置

APIServer Service:

```
apiVersion: v1
kind: Service
metadata:
  name: edge-apiserver
  namespace: edge-system
  labels:
    app: edge-apiserver
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/port: "9090"
    prometheus.io/path: "/metrics"
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 9090
    targetPort: 9090
    protocol: TCP
```

```
  - name: metrics
    port: 8080
    targetPort: 8080
    protocol: TCP
  selector:
    app: edge-apiserver
```

18.4.2 Ingress配置

完整Ingress示例:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: edge-platform
  namespace: edge-system
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
    nginx.ingress.kubernetes.io/proxy-body-size: "100m"
    nginx.ingress.kubernetes.io/proxy-connect-timeout: "600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "600"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "600"
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - edge.example.com
    secretName: edge-tls
  rules:
  - host: edge.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: edge-console
            port:
              number: 80
      - path: /kapis
        pathType: Prefix
        backend:
          service:
            name: edge-apiserver
            port:
              number: 9090
```

18.4.3 NetworkPolicy配置

APIServer NetworkPolicy:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: edge-apiserver
  namespace: edge-system
spec:
  podSelector:
    matchLabels:
      app: edge-apiserver
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: edge-system
    - namespaceSelector:
        matchLabels:
          name: ingress-nginx
    ports:
    - protocol: TCP
      port: 9090
  egress:
  - to:
    - namespaceSelector: {}
    ports:
    - protocol: TCP
      port: 2379  # etcd
    - protocol: TCP
      port: 9090  # Prometheus
```

18.5 安全配置

18.5.1 TLS证书

自签名证书:

```bash
# 生成CA
openssl genrsa -out ca.key 4096
openssl req -new -x509 -key ca.key -out ca.crt -days 3650

# 生成服务器证书
openssl genrsa -out server.key 4096
```

```
openssl req -new -key server.key -out server.csr
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
          server.crt -days 365

# 创建Secret
kubectl create secret tls edge-tls \
  --cert=server.crt \
  --key=server.key \
  -n edge-system
```

Let's Encrypt证书:

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: admin@example.com
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
    - http01:
        ingress:
          class: nginx
```

18.5.2 RBAC配置

ServiceAccount:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: edge-platform
  namespace: edge-system

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: edge-platform
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["*"]
- apiGroups: ["apps"]
  resources: ["*"]
  verbs: ["*"]
```

```
- apiGroups: ["iam.theriseunion.io", "scope.theriseunion.io",
             "app.theriseunion.io"]
  resources: ["*"]
  verbs: ["*"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: edge-platform
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: edge-platform
subjects:
- kind: ServiceAccount
  name: edge-platform
  namespace: edge-system
```

18.6 完整配置示例

18.6.1 最小化部署

```
# minimal-values.yaml
apiserver:
  replicaCount: 1
  resources:
    requests:
      cpu: 500m
      memory: 512Mi

controller:
  replicaCount: 1
  resources:
    requests:
      cpu: 200m
      memory: 256Mi

console:
  replicaCount: 1

ha:
  enabled: false

etcd:
  enabled: false
```

```yaml
prometheus:
  enabled: false
```

18.6.2 生产环境部署

```yaml
# production-values.yaml
global:
  imageRegistry: quanzhenglong.com
  storageClass: fast-ssd

apiserver:
  replicaCount: 3
  resources:
    requests:
      cpu: 1000m
      memory: 1Gi
    limits:
      cpu: 4000m
      memory: 4Gi
  autoscaling:
    enabled: true
    minReplicas: 3
    maxReplicas: 10

controller:
  replicaCount: 2
  leaderElection:
    enabled: true
  resources:
    requests:
      cpu: 500m
      memory: 512Mi

console:
  replicaCount: 2

ha:
  enabled: true

etcd:
  enabled: true
  replicaCount: 3
  persistence:
    enabled: true
    size: 100Gi

prometheus:
  enabled: true
  replicaCount: 2
```

```
  retention: 15d
  storageSpec:
    volumeClaimTemplate:
      spec:
        resources:
          requests:
            storage: 500Gi
  thanos:
    enabled: true
```

返回目录: Edge Platform白皮书

ewpage