项目申请书

项目名称：基于 kube-rs 实现超时资源回收控制器

项目主导导师：国子

申请人：郭柯震

日期：2024.05.27

邮箱：g1024536444@gmail.com

1. 项目介绍：

（1）项目简介：

Amphitheatre 是一个开源的开发者平台，旨在帮助开发者在云端立即启动新的自动化开发环境。它提供按需且预先配置好的所有工具、库和依赖项，以确保您能够立即开始编写代码。您可以在本地编辑应用程序源代码，Amphitheatre 会自动将您的变更增量部署到 Kubernetes 集群，使得在本地开发和远程部署之间的切换变得更加顺畅和高效。

（2）项目 issue 地址：https://github.com/amphitheatre-app/amphitheatre/issues/15

（3）项目产出目标：

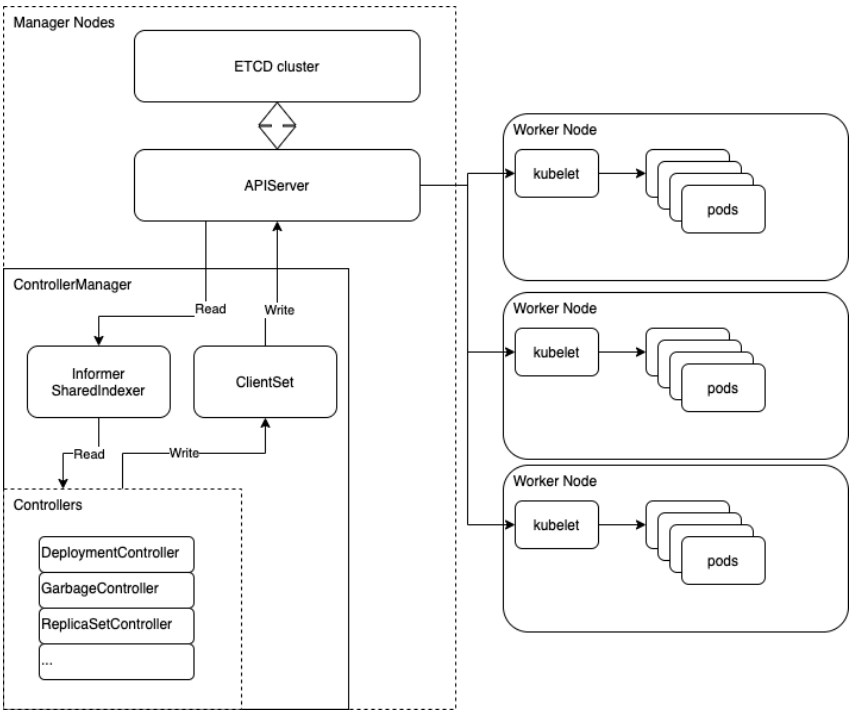
根据设计，Playbook 有一个生命周期，从创建、初始化、编译构建、部署、运行到销毁。

目前，我们需要实现一个新的控制器来执行 Playbook 的定期资源回收（销毁）。以下为技术说明：

- 管理员可以设置资源恢复时长，如 2 周（默认）；
- 该持续时间在每个 Playbook 资源定义上都有注释；
- Timeout Controller 定期扫描即将过期的 Playbook，并在过期日期前 3 天或 2 天执行一系列操作，例如发送提醒电子邮件或通知（例如通过 Slack），并在同一天执行 Playbook 删除；
- 额外考虑了支持未来事件的 Webhook 通知；

2. 项目相关参考：

（1） k8s 中 controller 实现



k8s 控制区模式采用声明式 api，来调度**不同的控制器**来完成相应的操作。声明式 API 系统里天然地记录了系统现在和最终的状态。可以在任意时刻反复操作。可以不需要加锁，

就支持多方的并发访问。

K8s Controller 的控制循环

在 K8s 中，用户通过声明式 API 定义资源的“预期状态”，Controller 则负责监视资源的实际状态，当资源的实际状态和“预期状态”不一致时，Controller 则对系统进行必要的更改，以确保两者一致，这个过程被称之为调和。

（2）相关代码参考：https://github.com/kube-rs/controller-rs/blob/main/src/controller.rs

```rust
#[instrument(skip(ctx, doc), fields(trace_id))] // #[instrument] 宏进行了仪表化（instrumentation），表示该函数需要记录日志
async fn reconcile(doc: Arc<Document>, ctx: Arc<Context>) -> Result<Action> { // 调和函数

    // 函数获取当前追踪 ID 用来追踪服务或者请求在微服务或者多组件服务中的路径 便于分析
    let trace_id: TraceId = telemetry::get_trace_id();

    // 使用当前的 span 记录追踪 ID，以便将其添加到日志中。 span记录这个请求在系统的做了什么
    Span::current().record("trace_id", &field::display(&trace_id));

    // 返回一个计时器（timer）对象 测量记录该请求在系统的执行的时间 方便后续性能分析 优化 监控和报警
    let _timer: ReconcileMeasurer = ctx.metrics.count_and_measure();

    // 获取诊断信息的写锁，并将当前时间赋值给诊断信息中的 last_event 字段。记录控制器 完成最后一次任务的时间
    ctx.diagnostics.write().await.last_event = Utc::now();

    // 获取文档的命名空间
    let ns: String = doc.namespace().unwrap(); // doc is namespace scoped

    // 用于操作特定命名空间下的 Kubernetes 资源的 API 对象  在这里实现了创建自定义资源
    let docs: Api<Document> = Api::namespaced(ctx.client.clone(), &ns);

    // 表示正在对文档进行协调处理，并输出文档的名称和命名空间。
    info!("Reconciling Document \"{}\" in {}", doc.name_any(), ns);

    // 这里执行最终的操作
    finalizer(&docs, DOCUMENT_FINALIZER, doc, |event: Event<Document>| async {
        match event {
            Finalizer::Apply(doc: Arc<Document>) => doc.reconcile(ctx.clone()).await,
            Finalizer::Cleanup(doc: Arc<Document>) => doc.cleanup(ctx.clone()).await,
        }
    })
    .await Result<Action, Error<Error>>
    .map_err(|e: Error<Error>| Error::FinalizerError(Box::new(e)))
} fn reconcile
```

这段代码展示了一个调和函数：将 yaml 文件中定义的资源引入映射到定义的结构内。调和函数内部定义了定时器（timer）以及追踪了服务在微服务、多组件之间的 id 路径。以

及相关的时间记录。方便后续进行监控、追踪与优化服务。操作在特定命名空间之下的资源 API 对象，并且进行日志打印，最终通过 finalizer 函数来通过 match 匹配不同的 event 事件进行操作。

（2） 相关代码参考：https://github.com/fpetkovski/k8s-ttl-controller

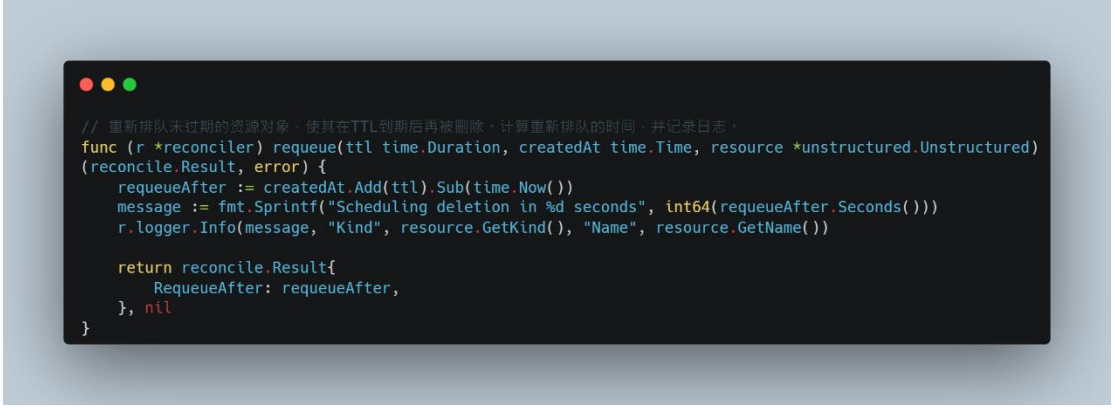
这里分析了项目源码中 pkg 中 Reconcile 中的代码。

```go
// 核心的对资源进行调和的函数
func (r *reconciler) Reconcile(request reconcile.Request) (reconcile.Result, error) {
    // 获取指定的资源对象，创建一个新的未结构化的资源对象，并设置其组、版本和类型（GVK）。
    resource := &unstructured.Unstructured{}
    resource.SetGroupVersionKind(r.gvk)

    // 如果资源未找到 记录日志并跳过此次调和，返回空结果和nil错误。
    if err := r.client.Get(context.Background(), request.NamespacedName, resource);
errors.IsNotFound(err) {
        r.logger.V(5).Info(
            "Could not find object, skipping.",
            "ApiVersion", r.gvk.GroupVersion(),
            "Kind", resource.GetKind(),
            "Name", request.Name)
        return reconcile.Result{}, nil
    } else if err != nil {  // 如果获取过程中发生其他错误，返回错误信息。
        return reconcile.Result{}, fmt.Errorf("could not get resource: %+v", err)
    }

    // Skip already deleted resources
    // 检查资源对象是否有删除时间戳，如果有，说明资源已经被删除，直接返回空结果和nil错误，跳过此次调和。
    if resource.GetDeletionTimestamp() != nil {
        return reconcile.Result{}, nil
    }

    // 检查资源是否匹配TTL策略
    if !r.objectMatcher.Matches(resource) {
        r.logger.Info(
            "Object does not match TTLPolicy resource rule, skipping deletion",
            "Name", resource.GetName())
        return reconcile.Result{}, nil
    }

    // 计算资源的过期时间
    expirationTime := resource.GetCreationTimestamp().Time // 初始过期时间为资源的创建时间。
    if r.expirationValueField != nil { // 如果定义了过期值字段，尝试从资源对象中获取过期时间，如果获取失败，记录日志并
跳过此次调和，返回空结果和nil错误。
        t, err := GetExpirationValue(resource, *r.expirationValueField)
        if err != nil {
            r.logger.Info(
                fmt.Sprintf("Expiration value is not a valid time: %s", err.Error()),
                "Kind", resource.GetKind(),
                "Name", resource.GetName(),
                "ExpirationFrom", *r.expirationValueField)
            return reconcile.Result{}, nil
        } else {
            expirationTime = t
        }
    }

    // 从资源对象中获取TTL值，如果获取失败，记录日志并跳过此次调和，返回空结果和nil错误。
    ttl, err := GetTTLValue(resource, r.ttlValueField)
    if err != nil { // 失败记录日志
        r.logger.Info(
            fmt.Sprintf("TTL value is not a valid duration: %s", err.Error()),
            "Kind", resource.GetKind(),
            "Name", resource.GetName(),
            "TTLFrom", r.ttlValueField)
        return reconcile.Result{}, nil
    }

    // 检查资源是否已过期
    if IsExpired(ttl, expirationTime) {
        return r.delete(resource)
    } else {
        return r.requeue(ttl, expirationTime, resource)
    }
}
```

- 这里首先获取指定的资源对象，并且考虑资源未找到的情况，并且检查资源对象是否被删除。
- 检查资源是否匹配 TTL 的策略
- 计算资源的过期时间
- 从资源对象中获取 TTL 的值，如果失败。记录日志并跳过此次检查周期
- 判断资源是否过期

```go
// 删除已过期的资源对象，记录日志。
func (r *reconciler) delete(resource *unstructured.Unstructured) (reconcile.Result, error)
{   r.logger.Info("Object expired", "Kind", resource.GetKind(), "Name", resource.GetName())
    backgroundDeletion := client.PropagationPolicy(v1.DeletePropagationBackground)
    err := r.client.Delete(context.TODO(), resource, backgroundDeletion)
    if err != nil {
        return reconcile.Result{
            RequeueAfter: 30 * time.Second,
        }, err
    }

    return reconcile.Result{}, nil
}
```

这里定义了删除已过期的资源对象，并且记录日志。



```go
// 重新排队未过期的资源对象，使其在TTL到期后再被删除。计算重新排队的时间，并记录日志。
func (r *reconciler) requeue(ttl time.Duration, createdAt time.Time, resource *unstructured.Unstructured)
(reconcile.Result, error) {
    requeueAfter := createdAt.Add(ttl).Sub(time.Now())
    message := fmt.Sprintf("Scheduling deletion in %d seconds", int64(requeueAfter.Seconds()))
    r.logger.Info(message, "Kind", resource.GetKind(), "Name", resource.GetName())

    return reconcile.Result{
        RequeueAfter: requeueAfter,
    }, nil
}
```

重新排队未过期的资源对象，等待下一次的检查，使其在 TTL 到期之后再被删除，计算重新排队的时间，并记录日志。

（3） 参考资料地址：

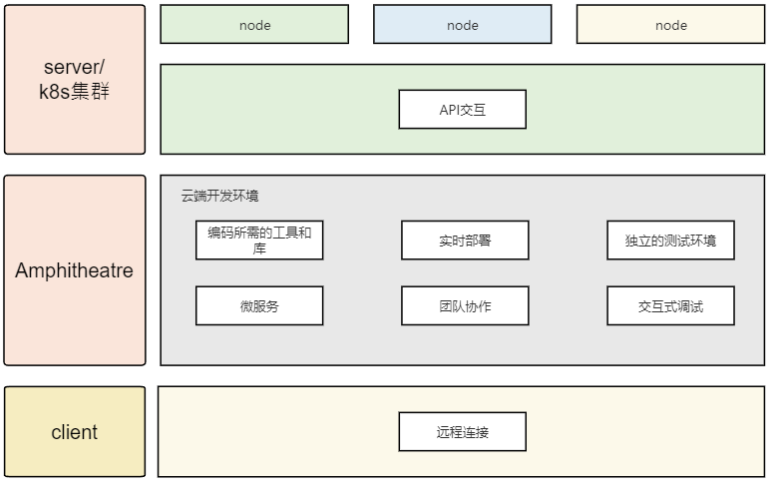
https://kubernetes.io/zh-cn/docs/concepts/workloads/controllers/ttlafterfinished/

TTL-after-finished 控制器假设 Job 能在执行完成后的 TTL 秒内被清理。一旦 Job 的状态条件发生变化表明该 Job 是 Complete 或 Failed，计时器就会启动；一旦 TTL 已过期，该 Job 就能被级联删除。 当 TTL 控制器清理作业时，它将做级联删除操作，即删除 Job 的同时也删除其依赖对象。

1 手动设置现有的，已完成的字段，以便生命周期检查清理。

2 在单个 job 清单中指定字段，以便到期之后进行检查清理。

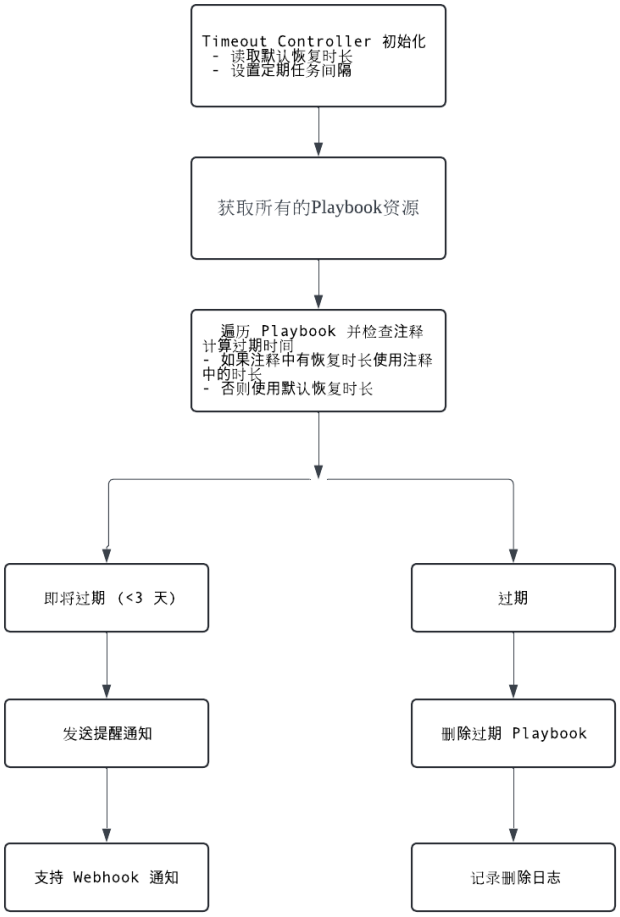
3. 实施方案：



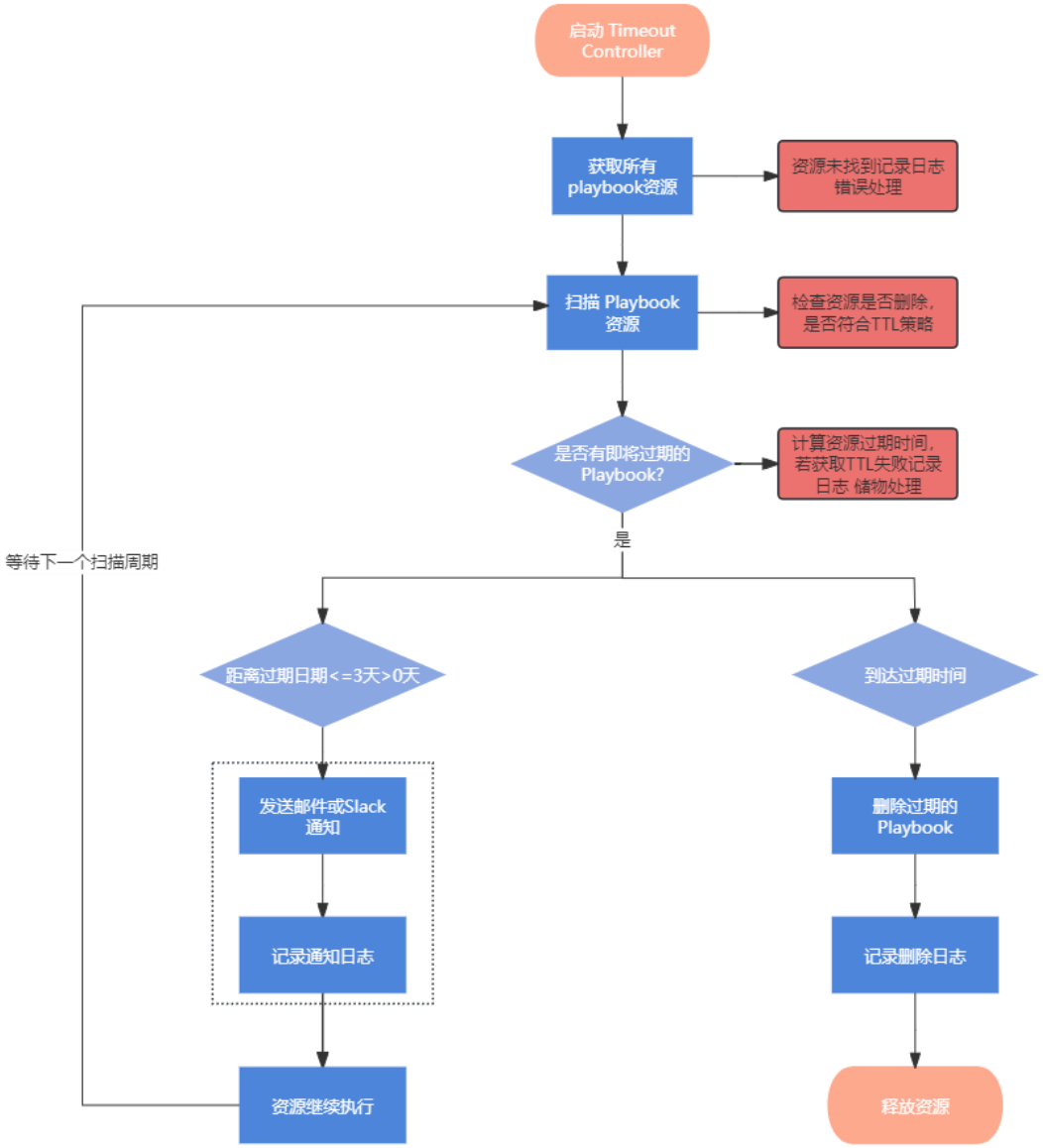
（1）项目结构及分析

管理员要设置默认的资源回收时间（2 周），若 playbook 上有自己的持续时间，则按照该 playbook 上的时间当作生命周期进行监控回收。每个 playbook 在到期前 3 天，调用相关接口，向 amp 使用者发送电子邮件进行提醒。Playbook 资源到期后，控制器控制操作进行销毁，回收资源。该 Timeout Controller 也要考虑将来可以方便地添加和处理新的事件类型，要预留接口或者处理逻辑。

（2）实施方案设计图：



（3）代码实现思路：



1. 启动 Timeout Controller：启动控制器，初始化所有必要的参数和配置。
2. 初始化定时扫描任务：设置定时扫描任务，周期性地检查 Playbook 资源。
3. 扫描 Playbook 资源：获取所有 Playbook 资源及其注释，解析创建时间和过期时间。
4. 是否有即将过期的 Playbook：检查是否有即将过期的 Playbook 资源。
5. 发送提醒通知:判断是否距离过期日期小于等于 3 天。发送提醒通知，通过邮件或 Slack。
6. 记录通知日志：记录通知日志以备审计。
7. 删除过期的 Playbook：自动删除过期的 Playbook 资源。
8. 记录删除日志：记录删除操作日志。

（4）初步代码实现：

```rust
use std::{ sync::Arc};
use k8s_openapi::chrono::Utc;
use tokio::time::{sleep, Duration};
use amp_common::resource::Playbook;
use kube::{api::ListParams, Api};
use tracing::{error, info};
use crate::context::Context;

// 启动一个资源回收的控制器
pub async fn new(ctx: &Arc<Context>) {
    // 创建一个api实例
    let api = Api::<Playbook>::all(ctx.k8s.clone());

    // 要在这里执行接下来封装清理的方法与逻辑
    loop {
        // 这里要从api中取出所有的playbook实例
        let playbooks = api.list(&ListParams::default()).await.unwrap();

        // 遍历所有的playbook
        for playbook in playbooks {
            // 将playbook中的过期时间进行判断
            if let Some(expire_at) = playbook.metadata.annotations.ttl {
                let now = Utc::now();
                if expire_at <= now {
                    api.delete(playbook.metadata.name(), &Default::default()).await.unwrap();
                } else if expire_at - now <= 3 {
                    notify(&playbook).await;
                }
            }
        }

        // 每24小时扫描一次 定期扫描
        sleep(Duration::from_secs(24 * 60 * 60)).await;
    }
}


// 这里返回 anyhow::Result 类型，表示可能返回错误的结果。
pub async fn handle(ctx: &Arc<Context>, playbook: &Playbook) -> anyhow::Result<()> {
    Ok(())
}


// 实现通知逻辑（如电子邮件、Slack 等）
async fn notify(playbook: &Playbook) {

}
```

根据项目源码以及相关参考，设计代码初步实现：

1. 首先定义一个资源回收控制器 new()
2. 定义处理错误以及通知逻辑的函数
3. 在控制器中获取 api 实例，使用 loop 循环执行检查，并且在最后规定 loop 休息时间（定时检查）
4. 在循环中拿到所有的 playbook，使用 for 遍历所有的 playbook，检查 playbook 中的过期时间
5. 进行逻辑判断，看资源是否到期，调用相关的函数发送通知，或者删除资源
6. 将控制器挂载到 main 函数中，在项目运行时启动控制器

4. 项目规划

（1） 项目研发第一阶段（07 月 01 日 - 07 月 07 日）：

a. 根据相关参考（https://github.com/fpetkovski/k8s-ttl-controller），完善项目流程设计与逻辑设计。

b. 基于项目基本代码实现，使用 rust 实现控制器代码逻辑。

c. 完善相关错误处理。

（2） 项目研发第二阶段（07 月 08 日 – 08 月 01 日）：

a. 实现 k8s 中的通知集成，电子邮件或者 slack 集成。

b. 将通知集成到控制器逻辑，完成项目设计与实现。

c. 对控制器进行测试，确保实现项目要求的功能。

（3） 项目研发第三阶段（08 月 02 日 – 09 月 30 日）：

a. 分析总结项目实现，发现自己实现项目过程中的不足，针对不足进行提高。

b. 继续深入学习项目其他部分，全面接触项目，完善对项目的理解。