terminus

深入探索云原生流水线的架构设计

— Erda DevOps 林俊

有没有一款工作流引擎，能够：

- 支持各种任务运行时，包括 K8s Job、K8s Flink、K8s Spark、DC/OS Job、Docker、InMemory 等?
- 支持快速对接其他任务运行时?
- 支持任务逻辑抽象，并且快速地开发自己的 Action?
- 支持嵌套流水线，在流水线层面进行逻辑复用?
- 支持灵活的上下文参数传递，有好用的 UI 以及简单明确的工作流定义?

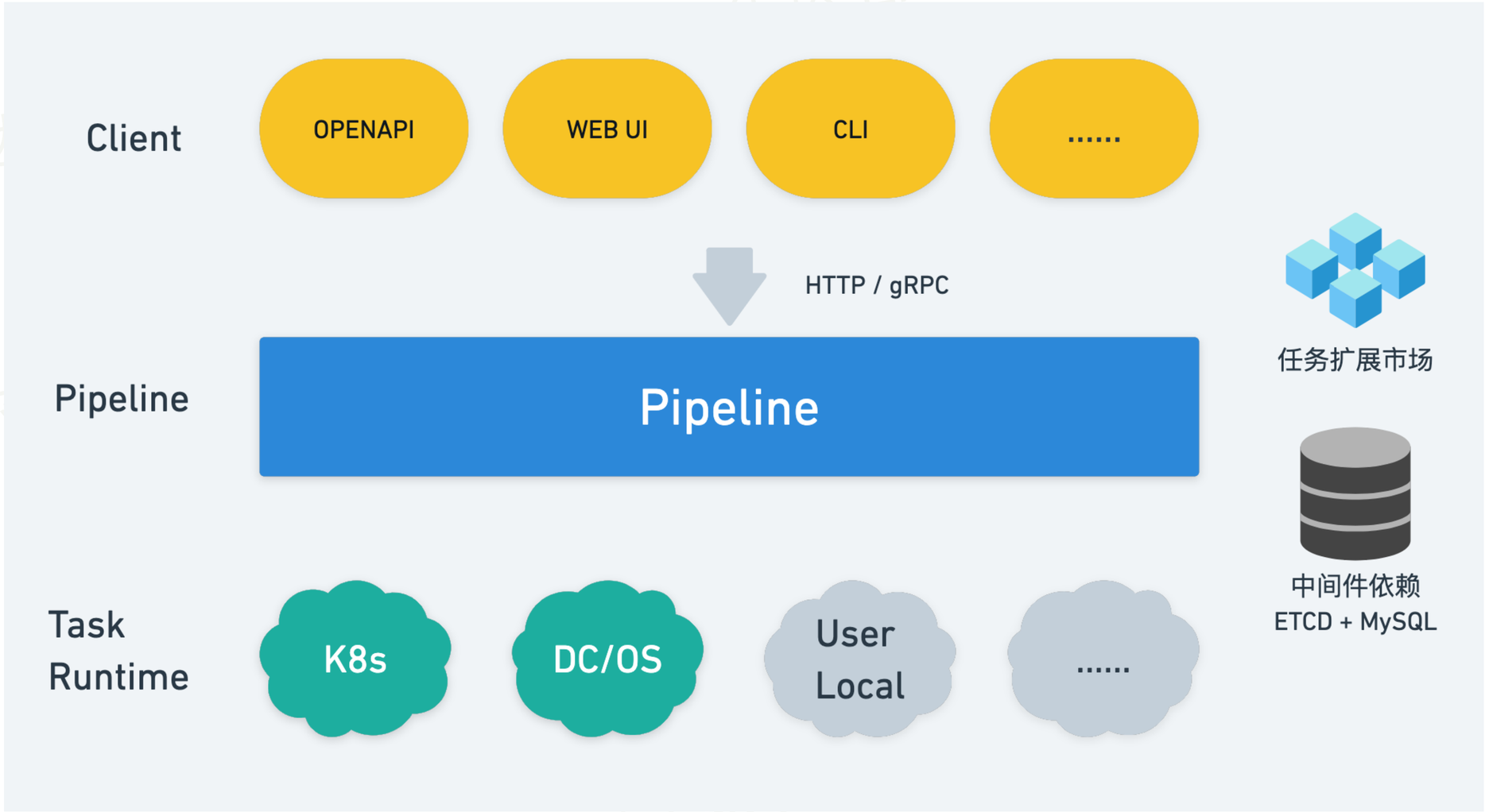
Pipeline 是一款自研的、用 Go 编写的工作流引擎。作为基础服务，它在 Erda 内部支撑了许多产品：

- CI/CD
- 快数据平台
- 自动化测试平台
- SRE 运维链路
- …

目录

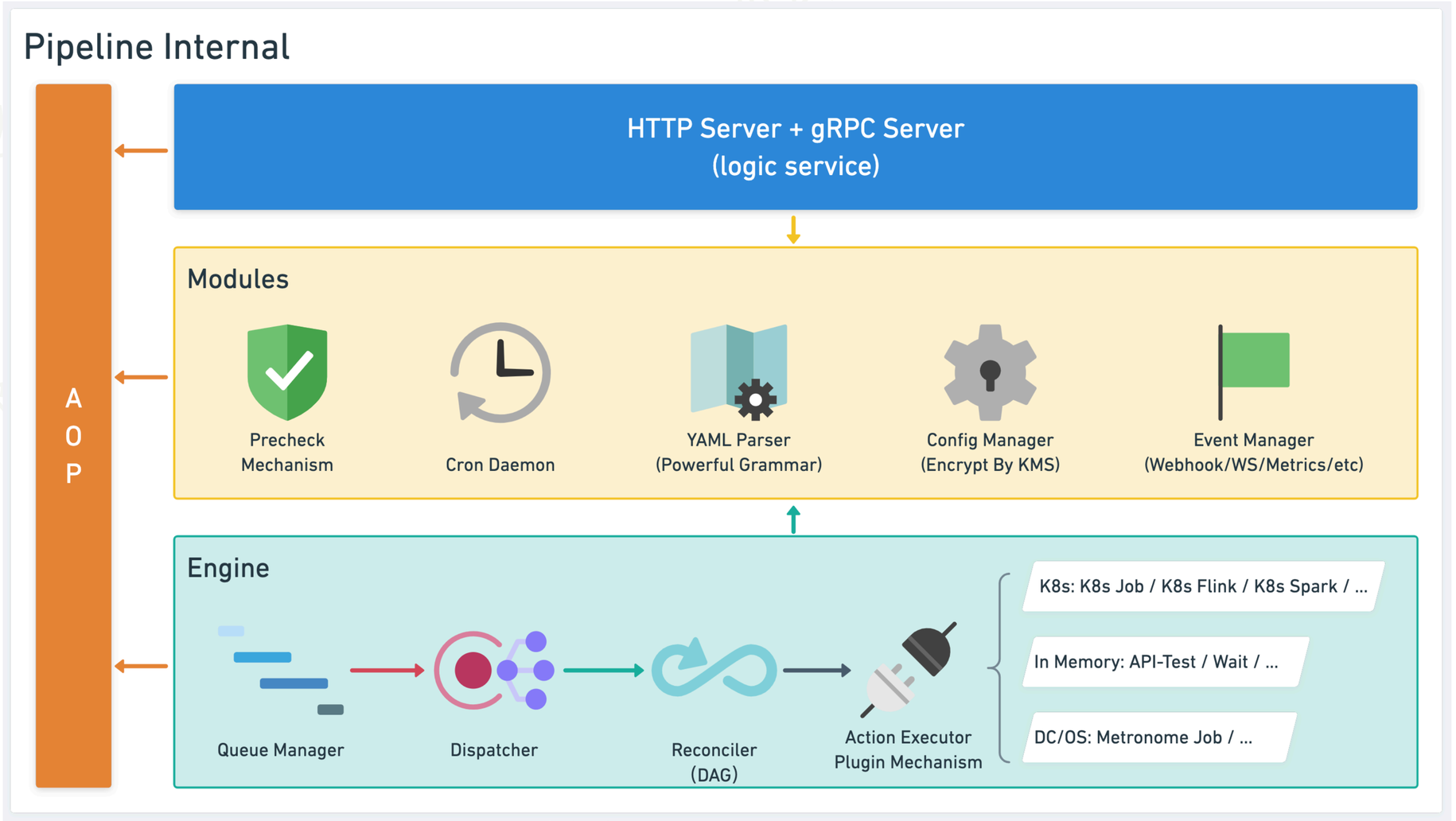
CONTENTS

整体架构



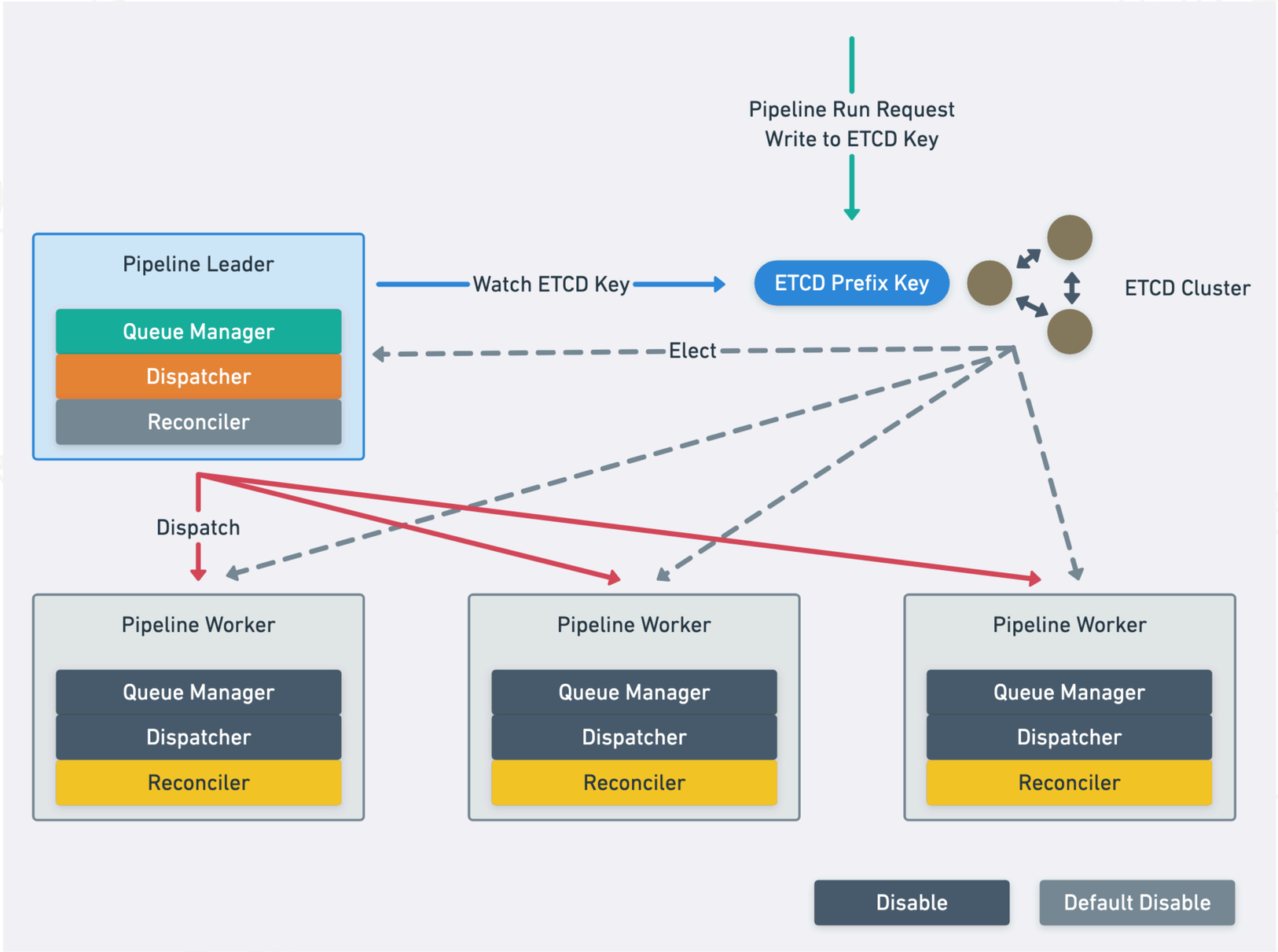
- 灵活的使用 & 接入方式
- 灵活的任务类型扩展
- 中间件依赖简单
- 适配多种任务执行环境

内部架构



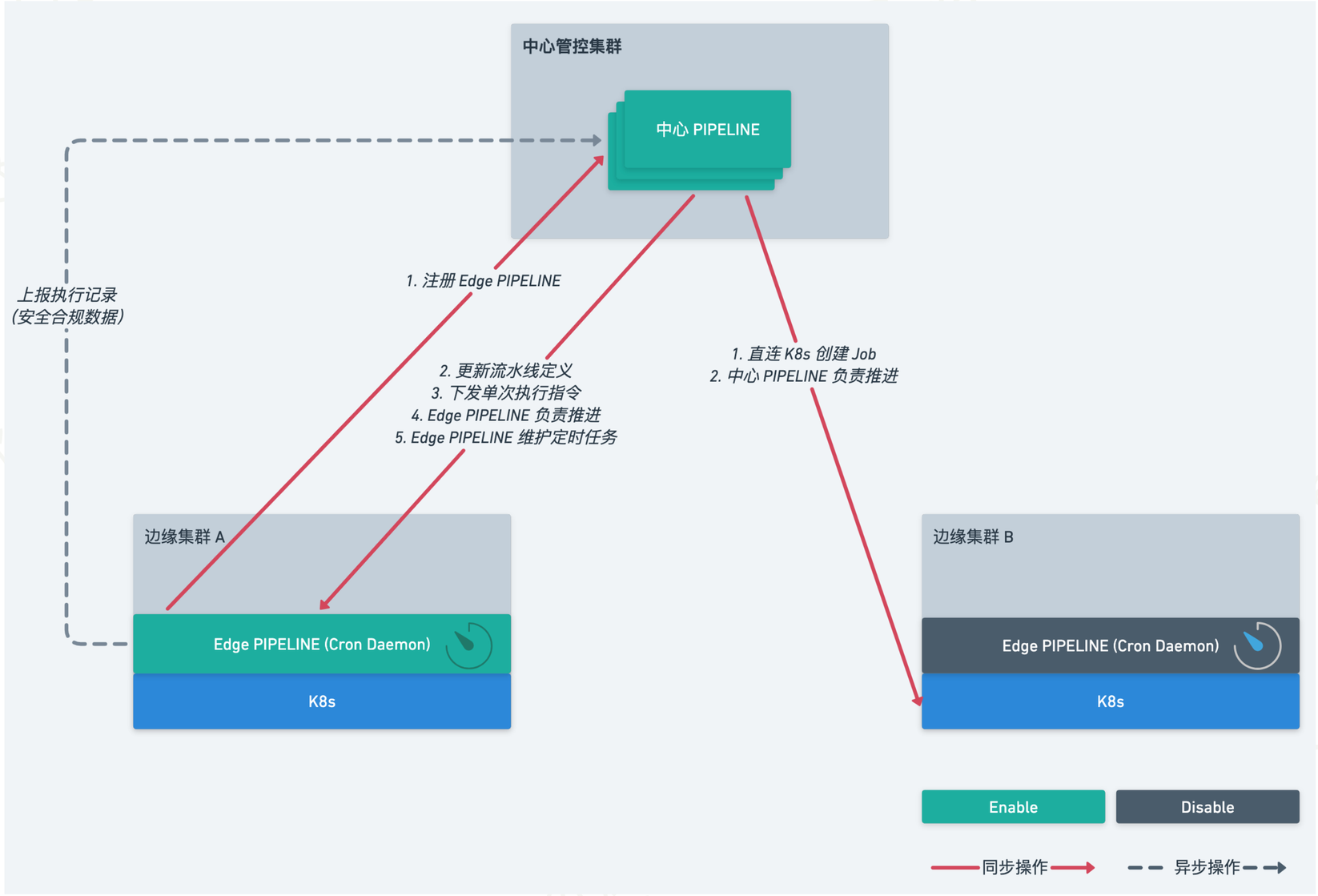
- 使用 Erda-Infra 微服务框架开发，功能模块划分清晰
- 模块内部使用插件机制，对接各种任务运行时
- AOP 扩展点机制，暴露关键节点，方便定制开发
- 统一使用 Event，封装了 WebHook / WebSocket / Metrics

水平扩展



- Leader & Worker 模式
- 支持单节点部署模式
- Dispatch 使用有界负载的一致性哈希算法
- 队列管理、任务分发、任务推进等功能模块化，模块对外暴露接口，模块间使用接口进行调用

分布式架构



- AP 模型，保证数据最终一致性
- 分区容忍
- 同一份代码，仅部署模式不同
- 中心 <–> 边缘 使用隧道通信
- 数据同步

功能特性

- 配置即代码，通过 pipeline.yaml 语法描述流程，基于 Stage 语法简化编排复杂度
- 扩展市场丰富，平台内置超过百款开箱即用的 Action，满足大部分日常场景；同时可轻松扩展你自己的 Action
- 可视化编辑，通过图形界面交互快速配置流水线
- 支持嵌套流水线，在流水线级别进行复用，组合出更强大的流水线
- 灵活的执行策略，包括串并行、循环、分支策略、超时、人工确认等，OnPush/OnMerge
- 支持工作流优先队列，优先级可实时调整，保证高优先级流水线优先执行
- 多维度的重试机制，支持断点重试、全流程重试
- 定时流水线及定时补偿功能
- 动态配置，支持 `值` 和 `文件` 两种类型，均支持加密存储，确保数据安全性
- 上下文传递，后置任务可以引用前置任务的 `值` 和 `文件`
- 开放的 OpenAPI 接口，方便第三方系统快速接入

实现细节 - 如何实现上下文传递(值引用)

```yaml
version: 1.1
stages:
  - stage:
      - git-checkout:
          alias: repo
          uri: https://github.com/erda-project/erda
          branch: master
  - stage:
      - custom-script:
          alias: build erda
          commands:
            - cd ${{ dirs.repo }} # file reference
            - make GIT_COMMIT=${{ outputs.repo.commit }} # value reference
```

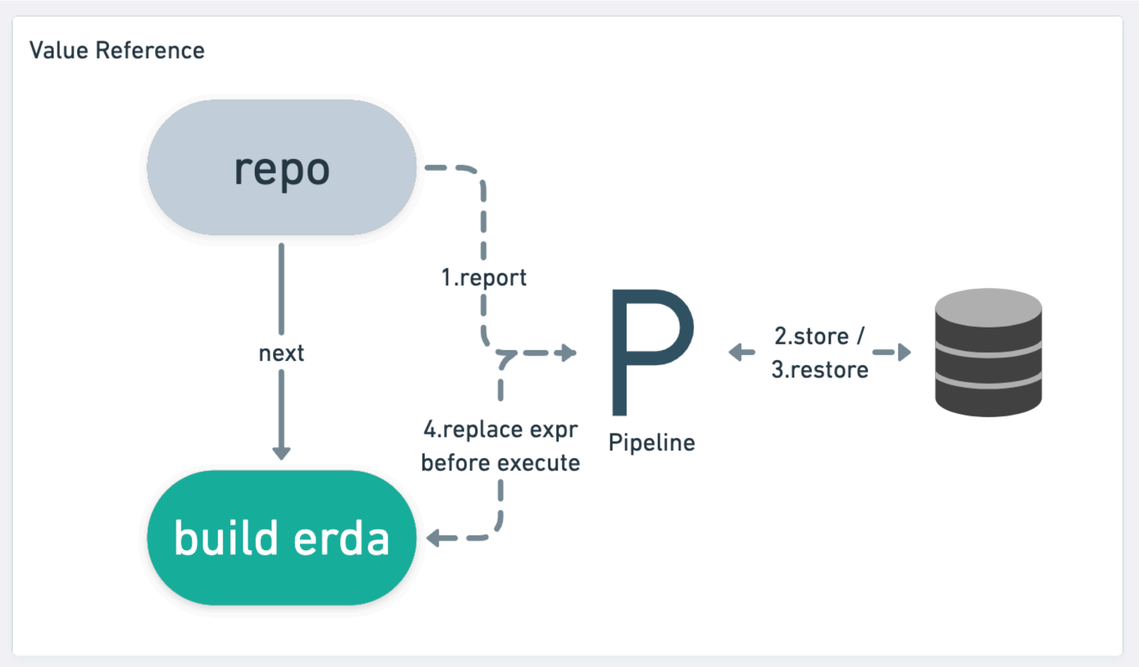
在一条流水线中，节点间除了有依赖顺序之外，一定会有数据传递的需求。

如左图所示：第一个节点 repo 拉取代码；第二个节点 build erda 则是构建 Erda 项目。

在例子中，第二步构建时同时用到了“值引用”和“文件引用”两种引用类型，是进依次入代码仓库，指定 GIT_COMMIT 进行构建。

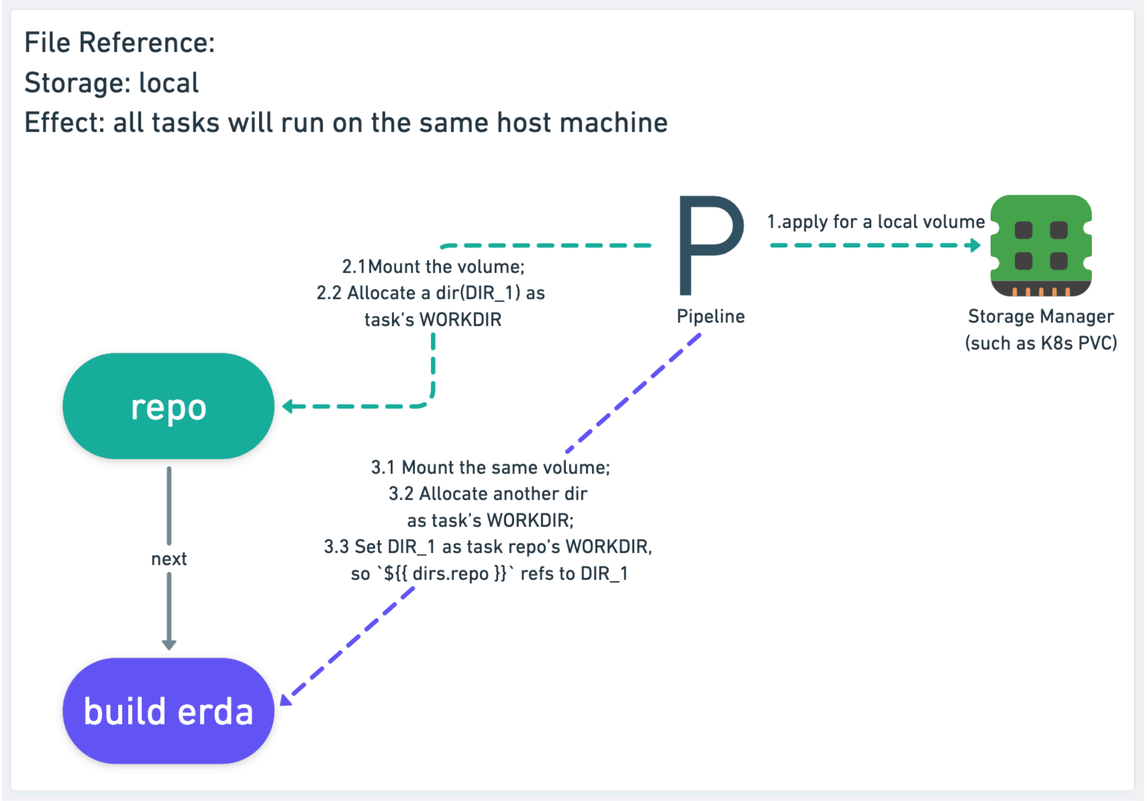
值引用：

- 每个节点的特殊输出(按格式写入指定文件或者打印到标准输出)会被保存在 Pipeline 数据库中；
- 后续节点通过 outputs 语法声明的表达式会在节点开始执行前被替换为真正的值。

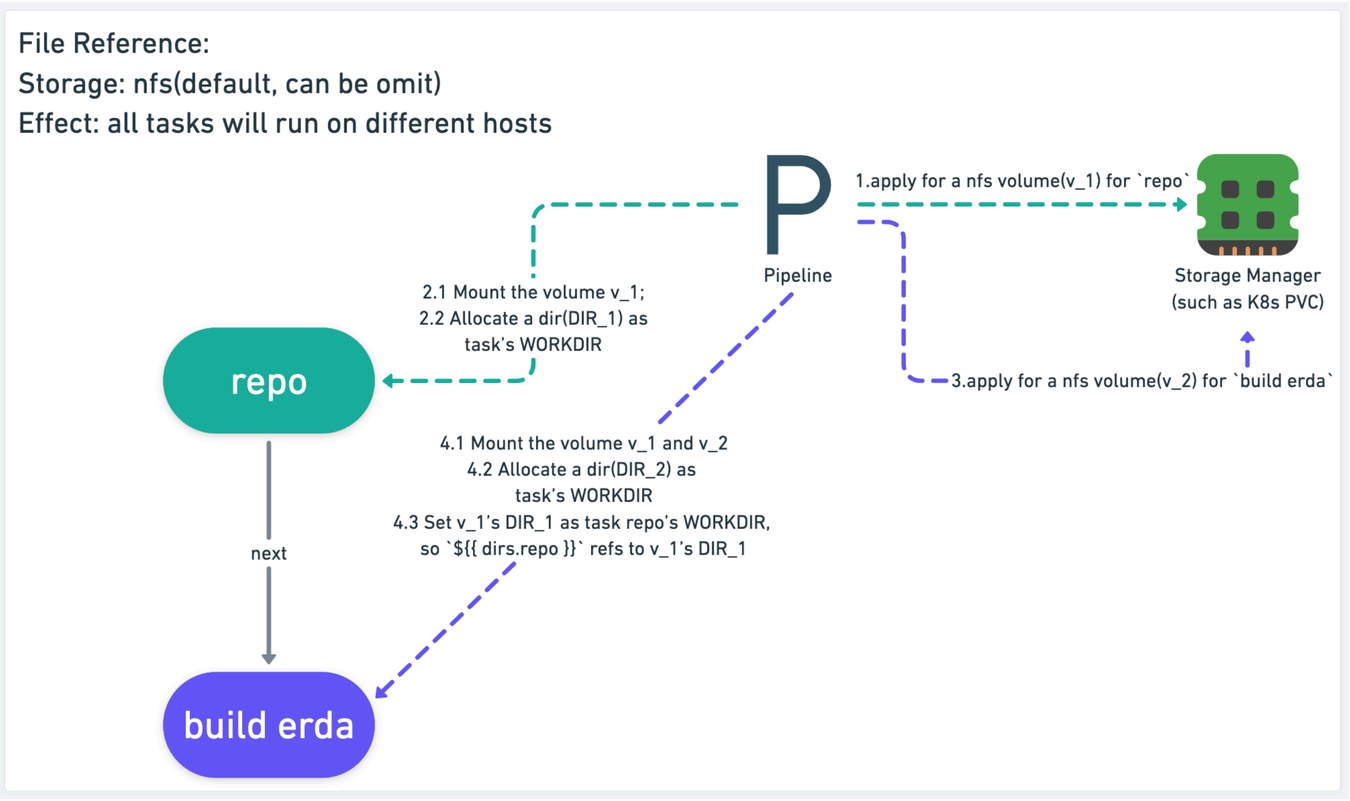


实现细节 - 如何实现上下文传递(文件引用)

- 文件引用比值引用复杂，因为文件的数据量比值大得多，不能存储在数据库中，而是存储在卷中。
- 这里又根据是否使用共享存储而分为两种情况，两者的区别在于申请的卷的类型和个数。
- 对于流水线使用者而言，没有任何区别。



不使用共享存储



使用共享存储

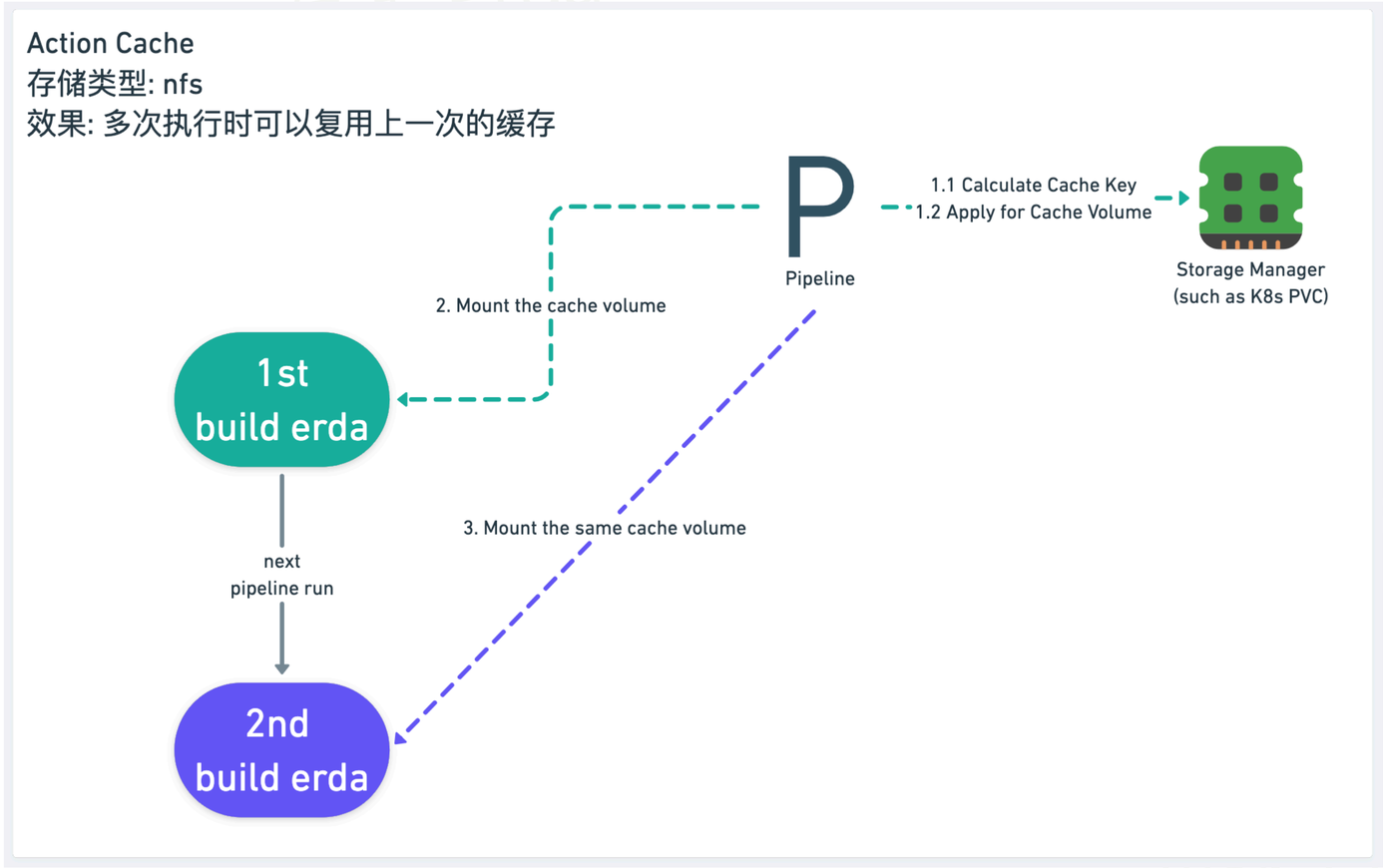
实现细节 - 如何实现缓存加速

在许多流水线场景中，同一条流水线的多次执行之间是有关联的。如果能够用到上一次的执行结果，则可以大幅缩短执行时间。

典型场景是 CI/CD 构建，我们以 Java 应用 Maven 构建举例：不但同一条流水线不同的多次执行可以复用 ${HOME}/.m2 目录(缓存目录)，甚至同一个应用下的多个分支之间都可以使用同一个缓存目录。就像本地构建一样！

```yaml
version: 1.1
stages:
  - stage:
      - git-checkout:
          alias: repo
          uri: https://github.com/erda-project/erda
          branch: master
  - stage:
      - custom-script:
          alias: build erda
          commands:
            - cd ${{ dirs.repo }} # file reference
            - make GIT_COMMIT=${{ outputs.repo.commit }} # value reference
          caches:
            - path: /root/.m2 # java maven cache dir
            - path: /root/go/mod # go mod cache dir
              key: go-cache
```

仍然使用前面的例子，在第二步 build erda 里加上 cache 即可。

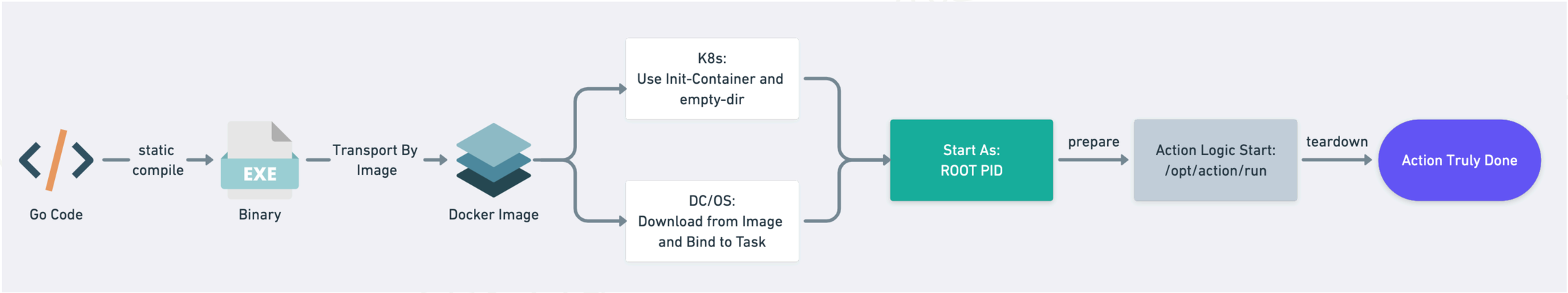


实现细节 - Action Agent

在 Pipeline 里，每个节点都会对应一个 Action 类型，并且在扩展市场中都可以找到。为了更加方便 Action 开发者进行开发，我们提供了很多便捷的机制。如：

- 对敏感日志进行脱敏处理，保证数据安全
- 无感知的错误分析和数据上报
- 文件变动监听及实时上报
- ……

上述所有机制都是由 Action Agent 程序完成的，它是一个静态编译的 Go 程序，可以运行在任意 Action 镜像中。Agent 完整的执行链路如下：



实现细节 - Action Executor 扩展机制

> 把复杂留给自己，把简单留给别人。

Pipeline 之所以好用，是因为它提供了灵活一致的流程编排能力，并且可以很方便的对接其他单任务执行平台，而这个平台本身不需要有流程编排的能力。在 Pipeline 中，我们对一个任务执行的抽象是 ActionExecutor

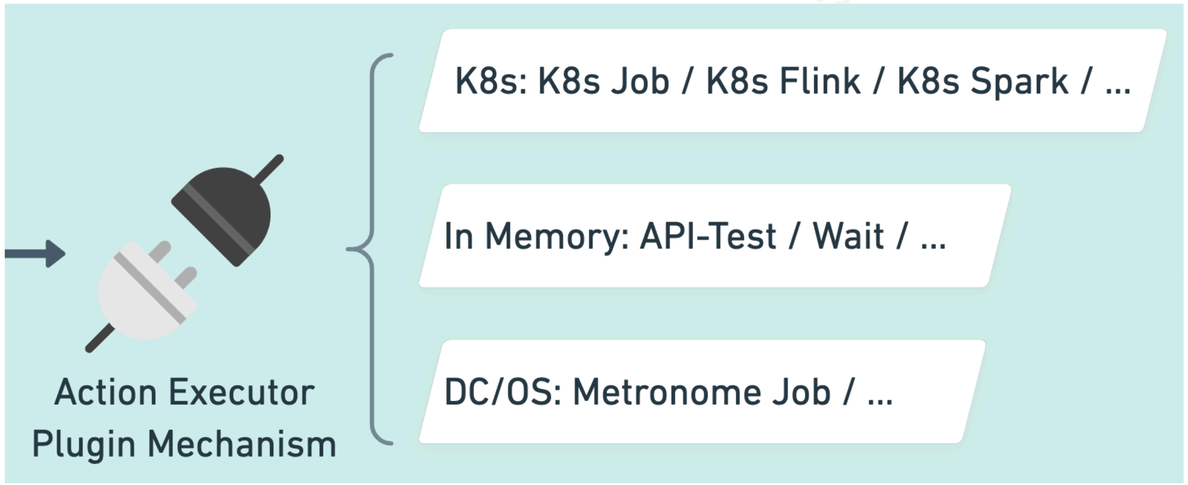
```go
type ActionExecutor interface {
    Kind() Kind
    Name() Name

    // Exist 返回 created, started, error
    Exist(ctx context.Context, action *spec.PipelineTask) (created bool, started bool, err error)
    // Create 保证幂等
    Create(ctx context.Context, action *spec.PipelineTask) (interface{}, error)
    // Start 保证幂等
    Start(ctx context.Context, action *spec.PipelineTask) (interface{}, error)
    Update(ctx context.Context, action *spec.PipelineTask) (interface{}, error)

    Status(ctx context.Context, action *spec.PipelineTask) (apistructs.PipelineStatusDesc, error)
    Inspect(ctx context.Context, action *spec.PipelineTask) (apistructs.TaskInspect, error)

    Cancel(ctx context.Context, action *spec.PipelineTask) (interface{}, error)
    Remove(ctx context.Context, action *spec.PipelineTask) (interface{}, error)
}
```

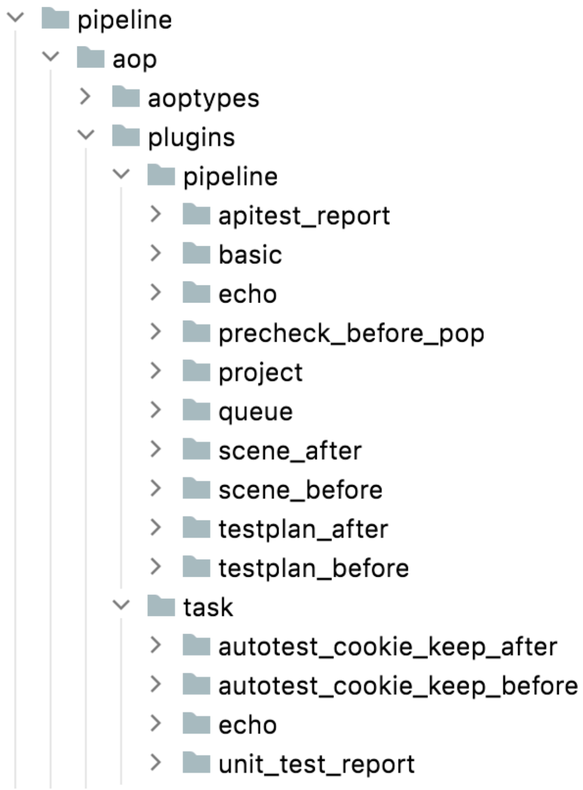
因此，一个执行器只要实现单个任务的创建、启动、更新、状态查询、删除等基础方法，就可以注册成为一个 ActionExecutor。



实现细节 - AOP 扩展点机制

AOP 扩展点机制是借鉴 Java 里 Spring 的概念应运而生的。

我们把代码关键节点进行暴露，方便开发同学在不修改核心代码的前提下定制流水线行为。AOP 扩展点机制已经使用 Erda Infra 的块化思想重构，整个扩展点的插件开发和编排更为灵活，如下图所示：



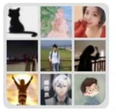
目前已经开发并使用的插件

插件的编排由配置文件控制

这个能力后续我们还会开放给用户，让用户可以在 pipeline.yaml 中使用编程语言声明和编排扩展点插件，更灵活的控制流水线行为。

terminus

THANKS FOR YOUR WATCHING!



Erda 用户交流群 ③

使用微信或企业微信扫码加入

该二维码 4 月 27 日前有效，重新进入将更新

企业微信