阿里云 | 奥运会全球指定云服务商

OCI Runtime rune技术解析

rev 2.1

阿里云基础软件部操作系统安全创新团队
乾越 <zhang.jia@linux.alibaba.com>

阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- 环境搭建
- init-runelet进程
- init容器进程
- rune exec执行流程

OCI Runtime rune

阿里云 | 奥运会全球指定云服务商

- rune是Inclavare Containers开源项目的核心组件之一，用于实现兼容OCI Runtime标准的Enclave容器运行时引擎。
- 源码仓库地址：https://github.com/alibaba/inclavare-containers
- 源码浏览地址：https://github.com/alibaba/inclavare-containers/tree/master/rune
- 文档：https://github.com/alibaba/inclavare-containers/blob/master/README.md
- rune的codebase是从runc fork出来的；会定期进行rebase。
- 本slide基于commit 4b0aced1f1f6fa874ed97a834c28fc0572121f42

Inclavare Containers开源项目

阿里云 | 奥运会全球指定云服务商

- 目标
  - 为业界提供面向机密计算领域的开源容器运行时引擎和安全架构。
- 价值
  - 抹平机密计算的高使用门槛，为用户提供与普通容器一致的使用体感。
  - 基于Intel处理器提供的多种硬件安全技术，为用户的工作负载提供多种不同的Enclave形态，在安全和成本之间提供更多的选择和灵活性。
- 当前状态
  - 已开源OCI Runtime rune、shim-rune和runectl三个组件
  - 支持Occlum和Graphene两个主流的SGX Library OS
  - 初步支持Java、Golang、Rust、Python 4种语言Runtime
  - 面向社区发布了0.1.0和0.2.0两个版本
  - 完成了Inclavare Containers 0.2.0在ACK-TEE（托管型K8s机密计算集群）上的适配
  - 目前仅支持Intel SGX enclave形态

Inclavare Containers的特点

兼容现有容器的使用体感	无缝运行用户制作的普通容器镜像	无需用户修改已有应用程序的代码	兼容k8s生态和OCI标准	在服务后端实现Enclave签名，对用户无感知
面向机密计算场景的安全架构	对Enclave Runtime启动时发起远程证明	允许用户随时发起对Enclave Runtime运行时的远程证明	提供Enclave管理功能，包括secret deployment	提供CSP不在TCB中的客户端签名方案
基于SGX/Trustzone处理器硬件提供的强安全防护	内置在处理器中的内存加密引擎提供加密内存防护	全新引入的Enclave处理器模式实施极强的安全隔离	新增的Enclave指令集允许实现灵活的软件设计	新的SGX2架构将加密内存扩展至per socket 512GB

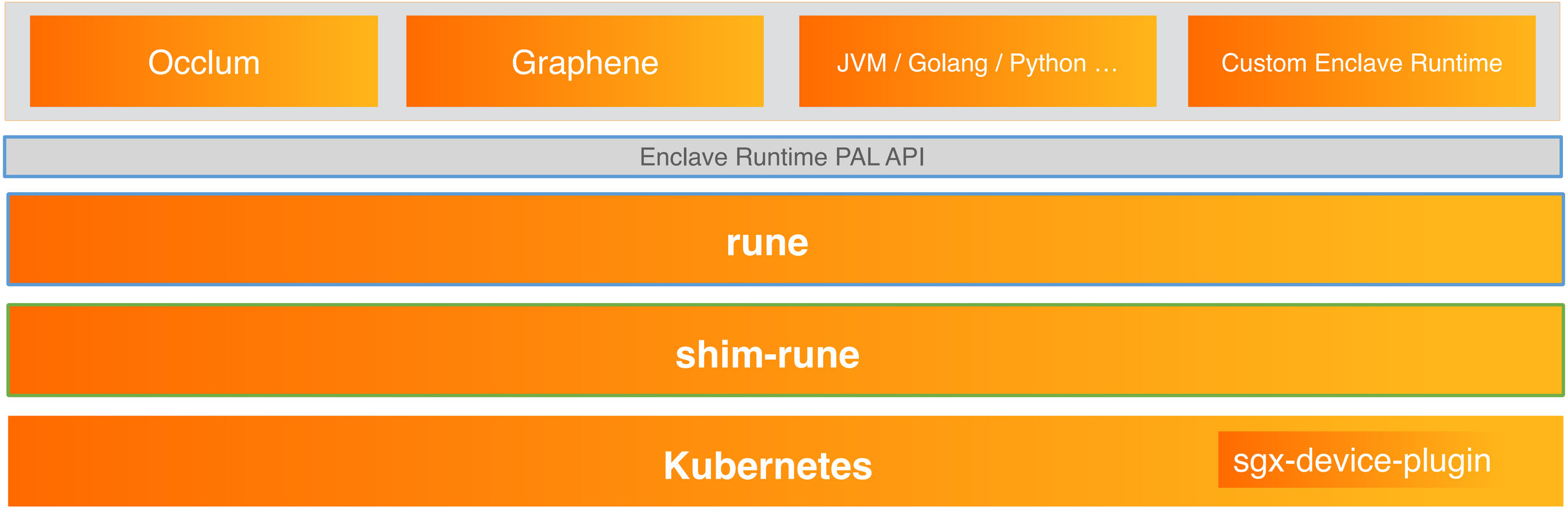
阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

Inclavare Containers架构

阿里云 | 奥运会全球指定云服务商



- 将Intel SGX与容器生态结合，对接K8s，兼容OCI Runtime标准，实现Enclave容器形态
- 基于Library OS技术，改善Intel SGX技术引入的约束条件所带来的兼容性问题
- 提供对语言Runtime的支持，进一步提升兼容性
- 定义通用的Enclave Runtime PAL API规范，构建Enclave Runtime生态

Inclavare Containers的外部系统架构

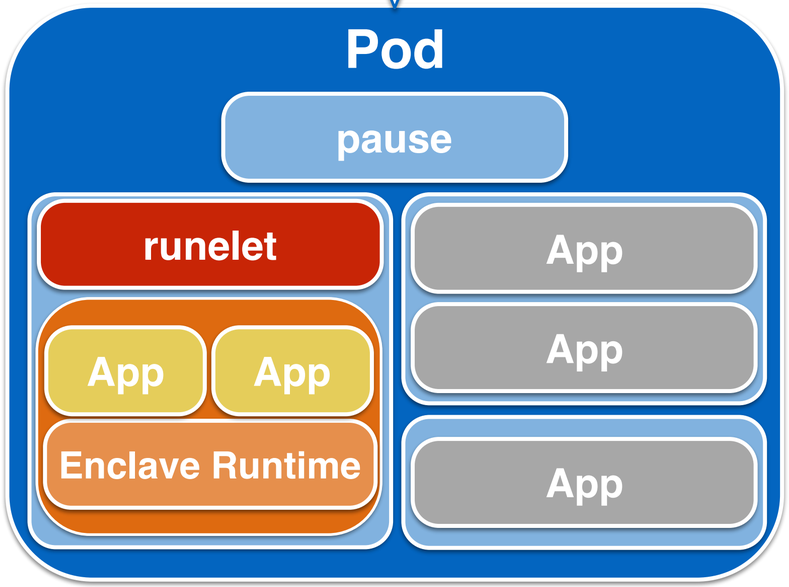
阿里云 | 奥运会全球指定云服务商

设计目标

- 满足K8S对runE作为enclave容器引擎的需求。

实现方式

- 使用能够原生支持RuntimeClass特性的组件。
- containerd-shim-rune-v2实现Shim Runtime v2 API，并提供Enclave管理功能（如Enclave签名、远程证明等）。



Inclavare Containers的内部系统架构（以Occlum为例）

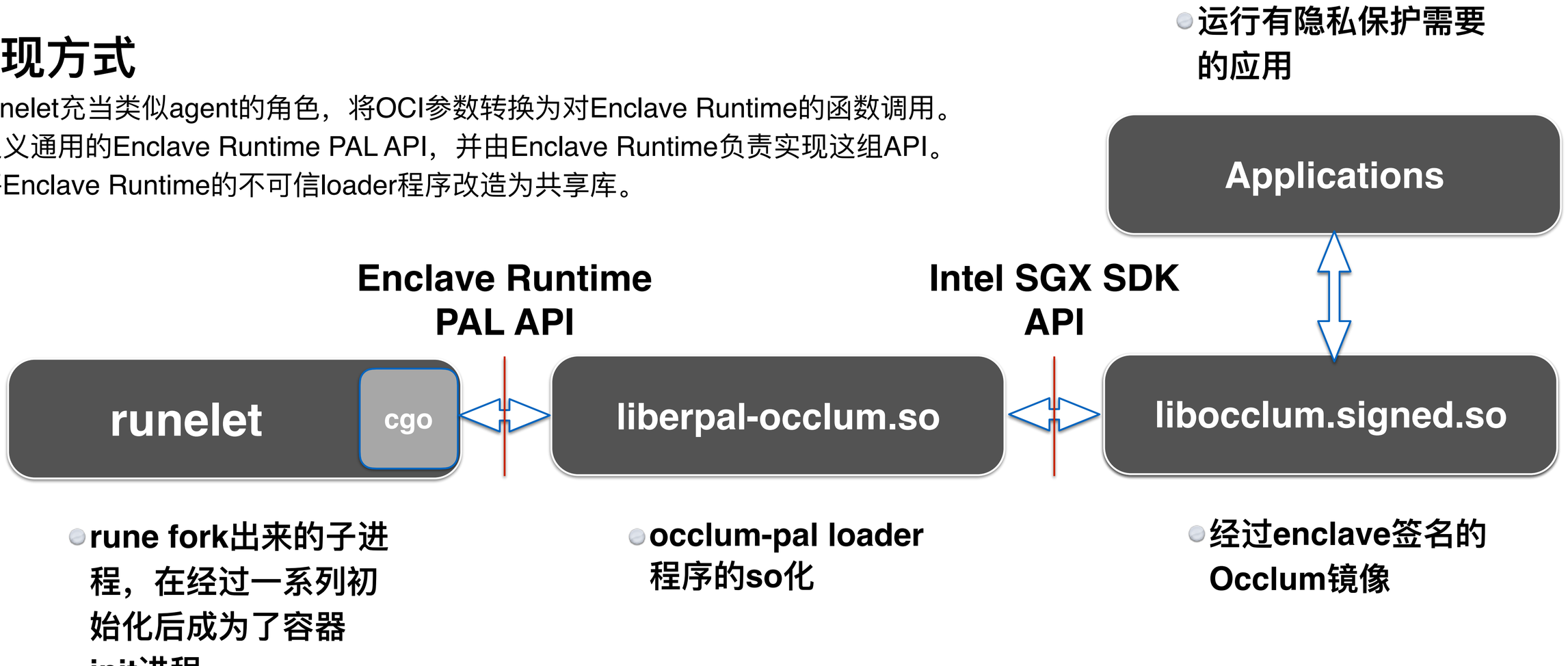
阿里云 | 奥运会全球指定云服务商

设计目标

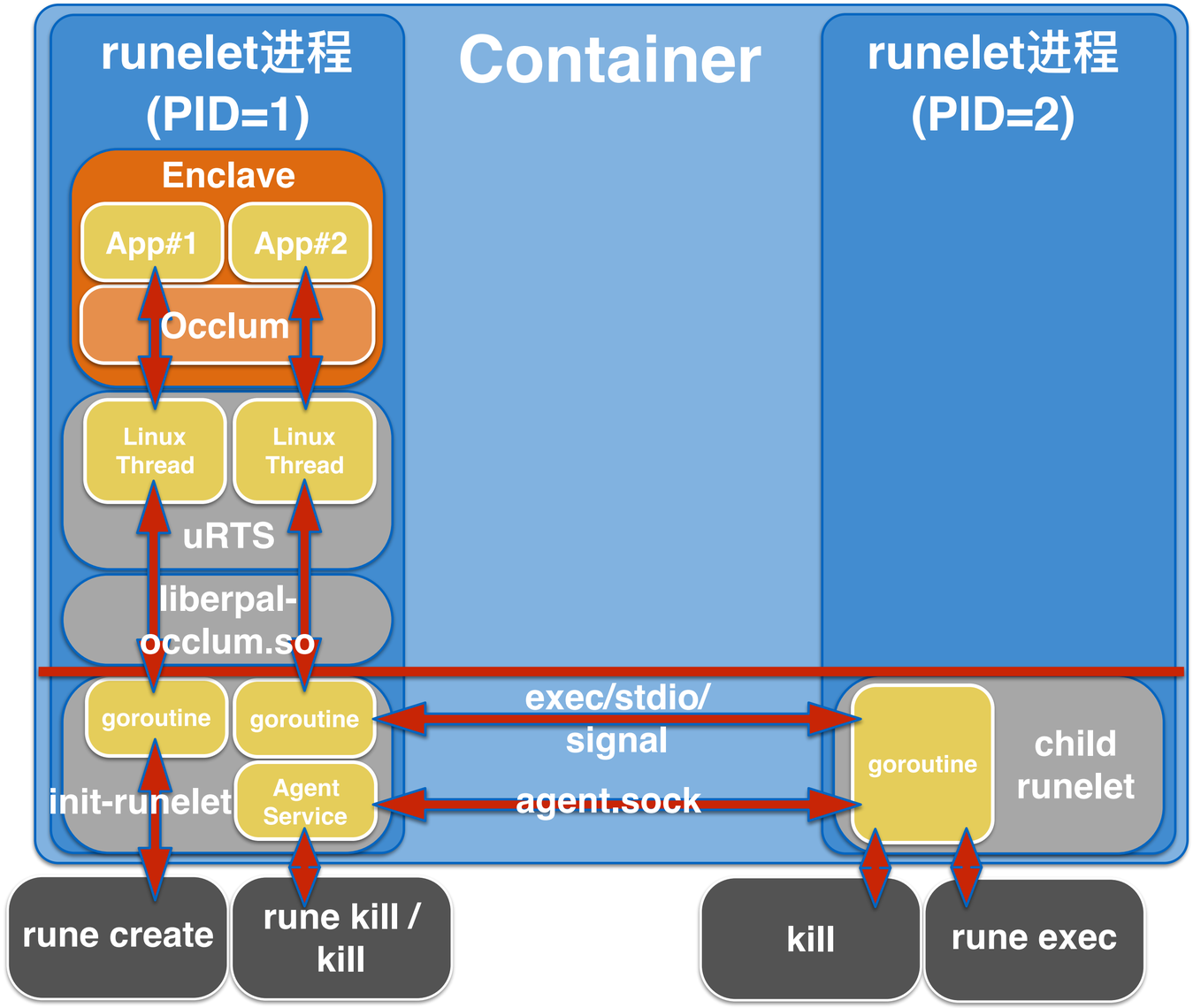
- 打通OCI到Enclave Runtime的链路。

实现方式

- runelet充当类似agent的角色，将OCI参数转换为对Enclave Runtime的函数调用。
- 定义通用的Enclave Runtime PAL API，并由Enclave Runtime负责实现这组API。
- 将Enclave Runtime的不可信loader程序改造为共享库。

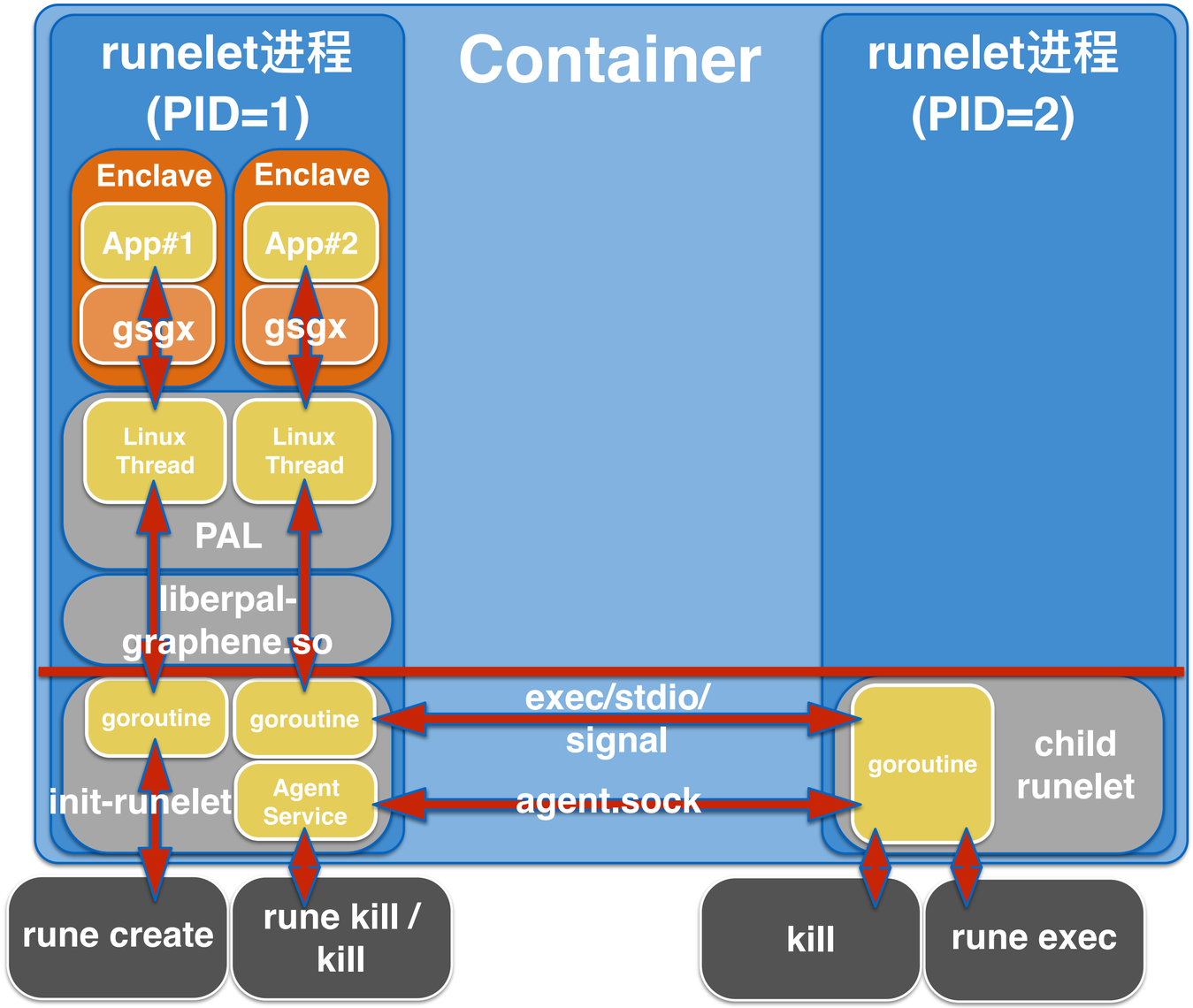


runelet对接Occlum的设计



- 设计目标
  - 支持rune exec和kill的功能
  - 正确处理per Occlum应用的stdio
- 实现方式
  - runelet#1启动Agent Service监听agent.sock来处理child runelet的连接请求并派生出一个goroutine。
  - 新派生的goroutine负责处理rune exec要运行的enclave应用信息（程序路径、参数列表和环境变量列表），最终转换为对PAL API exec的调用。
  - 发给child runelet的信号会经由新派生的goroutine转换为对PAL API kill的调用。
  - 新派生的goroutine负责转发child runelet的stdio。
  - 发给runelet#1的信号会被转换为对PAL API kill的调用。

runelet对接Graphene的设计



Enclave Runtime PAL API

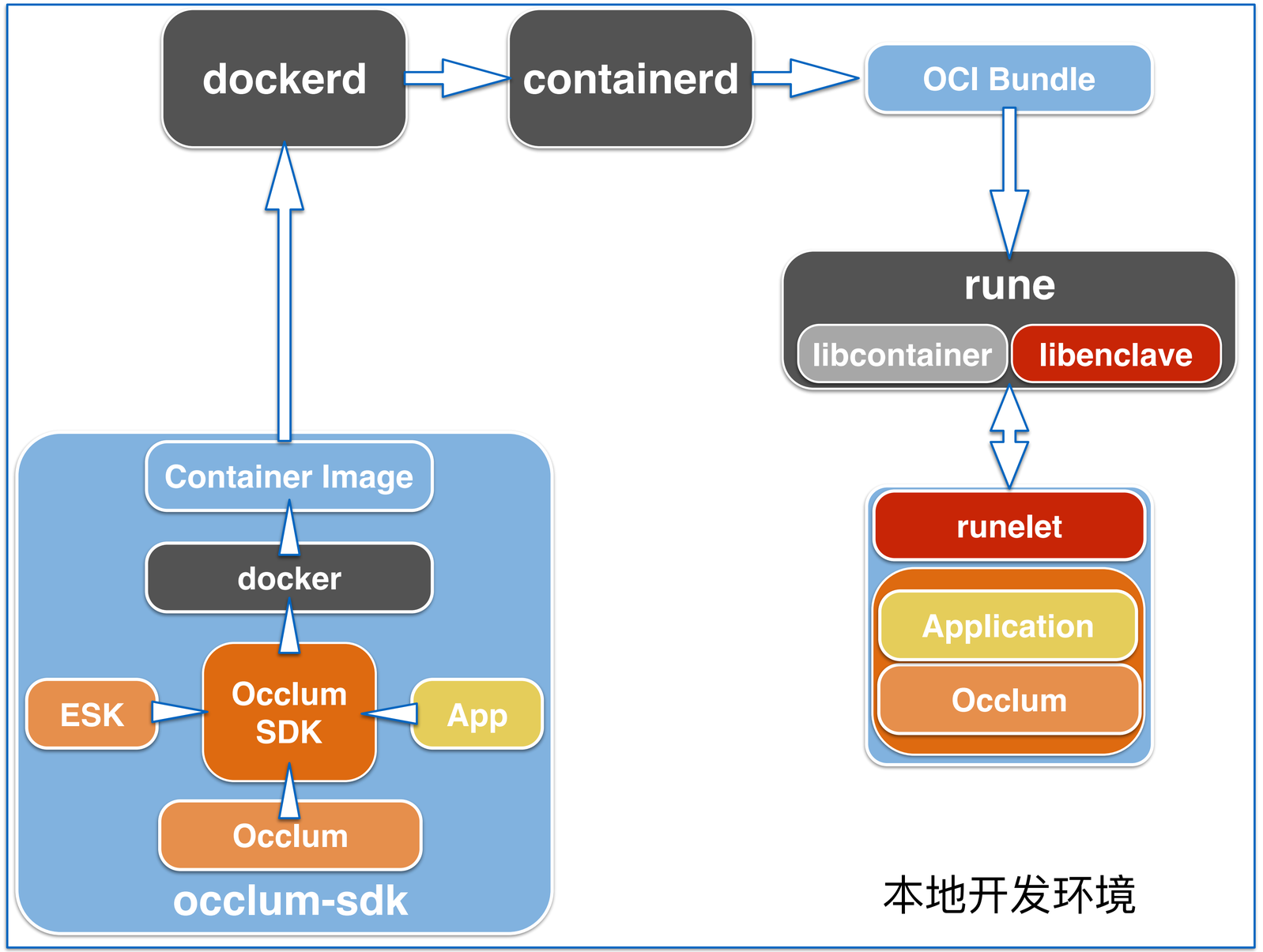
- 设计目标和实现方式与对接Occlum的方案相仿

阿里云 | 奥运会全球指定云服务商

Agenda

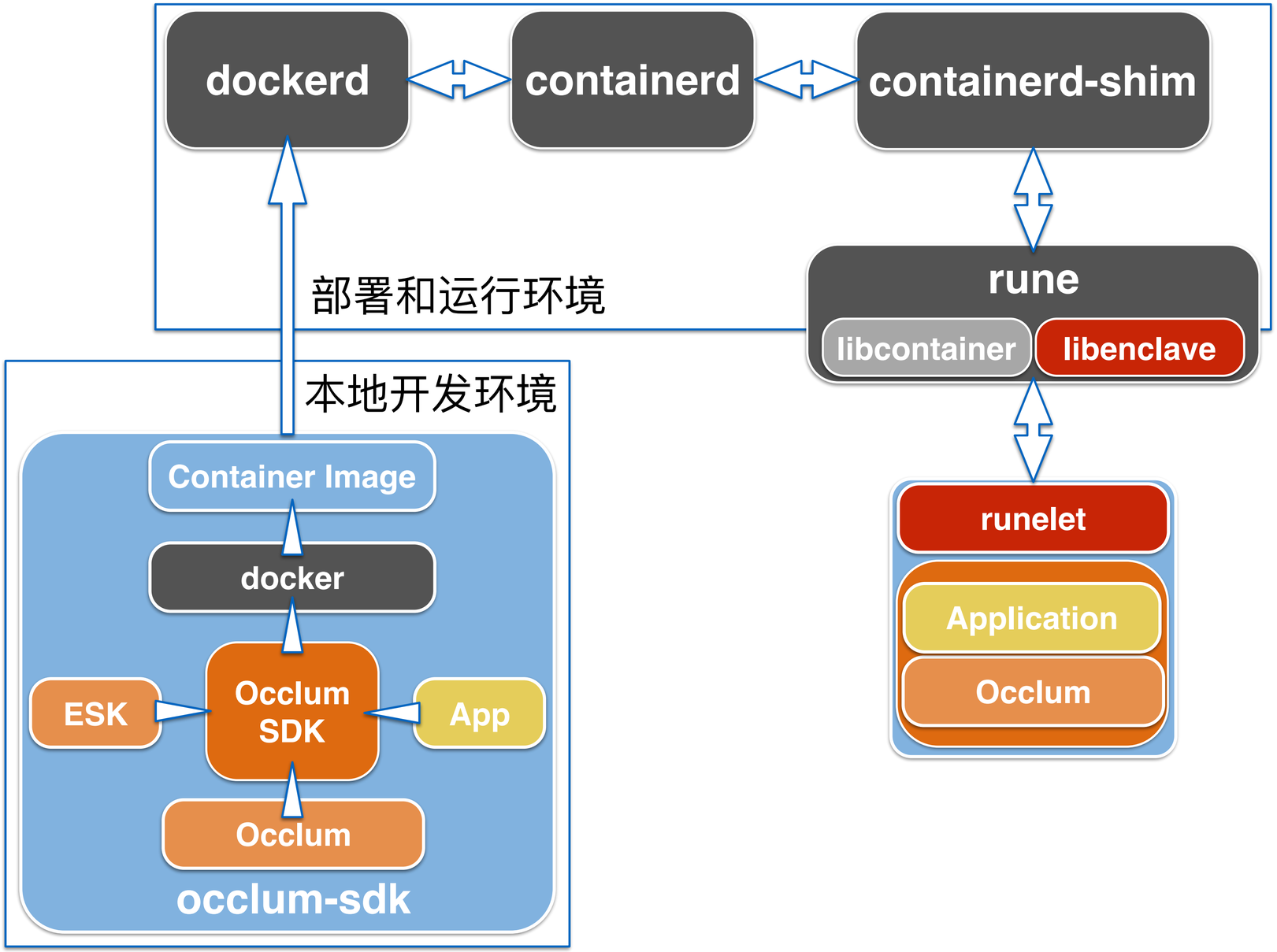
- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

rune+OCI Bundle+Occlum运行环境



- 仅适用于开发者使用的本地开发和测试环境。
- 在本地可信环境中使用occlum-sdk容器镜像提供的开发环境进行开发和Occlum应用容器镜像的构建。
- 具体搭建文档详见：https://github.com/alibaba/inclavare-containers/blob/master/docs/running_rune_with_occlum_bundle.md

Docker+rune+Occlum运行环境



- 适用于本地部署和云上部署。
- 在本地可信环境中使用occlum-sdk容器镜像提供的开发环境进行开发和Occlum应用容器镜像的构建。
- 具体搭建文档详见：https://github.com/alibaba/inclavare-containers/blob/master/docs/running_rune_with_occlum.md

阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

术语定义(1)

- parent rune进程：泛指通过rune CLI工具运行create/start/run/exec等子命令的、运行在host侧的进程。

- bootstrap进程：rune create/run/exec进程会通过执行fork+execve(“/proc/self/exe init”)创建出bootstrap进程。该进程的职责是：1）fork出容器进程；2）完成bootstrap的工作，即与rune create/run/exec进程和容器进程进行交互，将容器进程初始化过程中需要的配置信息和容器进程的管理信息返回给上述parent rune进程。在完成bootstrap的工作后，bootstrap进程会自动退出。

- 容器进程：分为init容器进程和setns容器进程，均由bootstrap进程创建。在创建伊始，容器进程仍旧处于host环境，在收到通过bootstrap进程传递过来的有关容器化的配置信息并根据这些信息逐步进行容器化后，最终成为容器进程。

- init容器进程：由rune create/run命令创建的容器进程，PID一般为1（除非在容器配置中没有配置PID namespace）；在runc的场景中，容器进程最终会根据容器配置通过执行execve()来运行容器内的指定程序；而在rune的场景中，不会执行execve()，最终变成所谓的init-runelet进程。

- setns容器进程：由rune exec命令创建的容器进程，PID不为1；在runc的场景中，setns容器进程最终会根据命令行参数或shim传递过来的参数通过执行execve()来运行容器内的指定程序；而在rune的场景中，不会执行execve()，最终变成所谓的runelet进程。

术语定义(2)

- init-runelet进程：本质上就是init容器进程。init容器进程成为init-runelet进程的标志是：在调用libenclave.StartInitialization(l.config.Args, cfg)@libcontainer/standard_init_linux.go前，被称为init容器进程；在这之后，被称为init-runelet进程。有时候为了更准确地描述一个进程在不同执行阶段中的行为，或是在不同语境下想强调不同的概念，会混用init容器进程和init-runelet进程这两个术语。

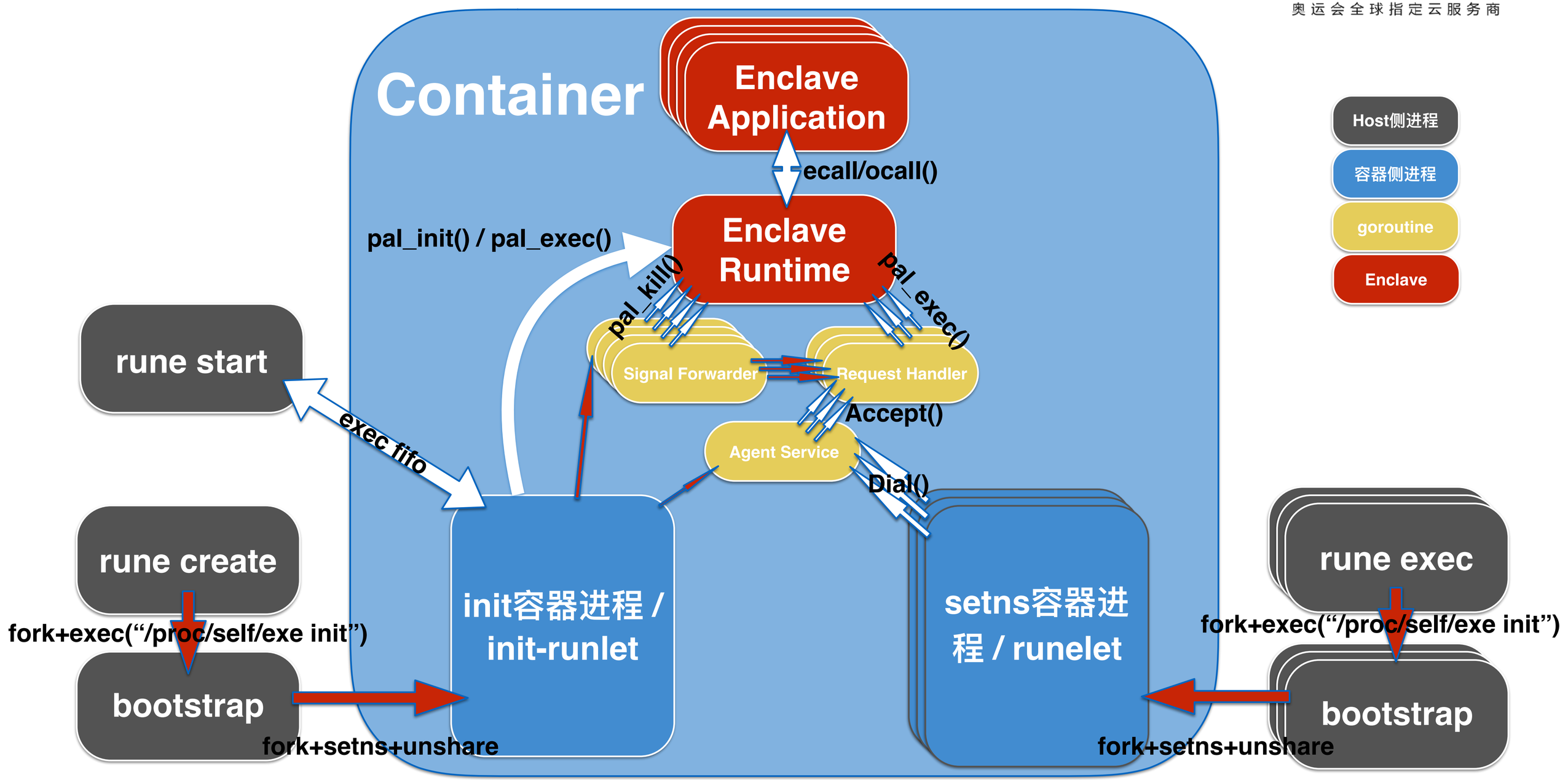
- runelet进程：本质上就是setns容器进程。setns容器进程成为runelet进程的标志是：在调用libenclave.StartInitialization(l.config.Args, cfg)@libcontainer/setns_init_linux.go前，被称为setns容器进程；在这之后，被称为runelet进程。有时候为了更准确地描述一个进程在不同执行阶段中的行为，或是在不同语境下想强调不同的概念，会混用setns容器进程和runelet进程这两个术语。

术语定义(2)

阿里云 | 奥运会全球指定云服务商

- Enclave Runtime PAL API：rune通过这组API规范来对接下游的Enclave Runtime组件。
- Enclave Runtime PAL：通常需要在rune和Enclave Runtime之间使用一个封装了Enclave Runtime PAL API的中间组件即PAL来真正地将Enclave Runtime对接到rune。
- Enclave Runtime：作为enclave后端，负责具体的enclave技术实现。
- Skeleton Enclave Runtime：rune工程自带的示例性Enclave Runtime；开发者可以以该示例来开发自己的Enclave Runtime。
- Occlum Enclave Runtime：一个多线程的、基于rust编写的内存安全的SGX LibOS开源实现。
- Enclave容器：指由OCI Runtime rune管理的容器类型。

通过rune create+start/exec运行Enclave应用的流程图



使用rune spec命令生成enclave容器的配置模板文件

阿里云 | 奥运会全球指定云服务商

@libcontainer/specconv/example.go

```
        spec := &specs.Spec{
                Version: specs.Version,
                Root: &specs.Root{
                        Path: "rootfs",
                },
                Process: &specs.Process{
                        Terminal: true,
                        Cwd:              "/var/run/rune",
                Hostname: "rune",
                Mounts: []specs.Mount{
                        {
                                Destination: "/var/run/aesmd",
                                Type:        "bind",
                                Source:      "/var/run/aesmd",
                                Options:     []string{"rbind", "rprivate"},
                        },
                },
                Annotations: map[string]string{
                        "enclave.type":         "intelSgx",
                        "enclave.runtime.path": "/var/run/rune/liberpal-skeleton.so",
                        "enclave.runtime.args": "skeleton,debug",
                },
        }
```

未修改的runc代码

新实现的rune代码

- 生成rune专用的config.json模板文件
  - 容器的hostname从runc改为rune
  - 容器bundle的访问权限设为可读写（隐式默认设置，无需明确指定）
  - 容器进程的CWD改为/var/run/rune
  - “透传”aesm.socket所在目录/var/run/aesmd
  - 将skeleton enclave runtime的配置设为默认的enclave配置

- 对于rune的开发者来说，首先使用rune spec命令生成config.json，然后根据需要进行适当调整并配合OCI Bundle来运行rune是日常的开发模式。

新增libcontainer.Config.Enclave对象

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/configs/config.go
type Config struct {
…
    // Enclave specifies settings for the enclave-based hardware that the container
    // is placed into for data protection.
    Enclave *Enclave `json:"enclave,omitempty"`
}
```

```
@libcontainer/configs/enclave.go
type Enclave struct {
    Type string `json:"type"`
    Path string `json:"path"`
    Args string `json:"args,omitempty"`
}
```

未修改的runc代码
新实现的rune代码

- 在libcontainer.Config中新增enclave配置对象
  - 在parent rune运行时，config.json中的enclave配置会被反序列化为这里定义的libcontainer.Config.Enclave对象
  - 这些enclave配置决定了Enclave容器的行为
  - Enclave配置支持两种形式：
    - ENCLAVE环境变量(以使用skeleton为例)：

```
ENCLAVE_TYPE=intelSgx
ENCLAVE_RUNTIME_PATH=/usr/lib/liberpal-skeleton.so
ENCLAVE_RUNTIME_ARGS=skeleton,debug
```

    - Annotations(以使用Occlum为例)：

```
"annotations": {
    "enclave.runtime.args": ".occlum",
    "enclave.runtime.path": “/usr/lib/libocclum-pal.so”,
    "enclave.type": “intelSgx"
}
```

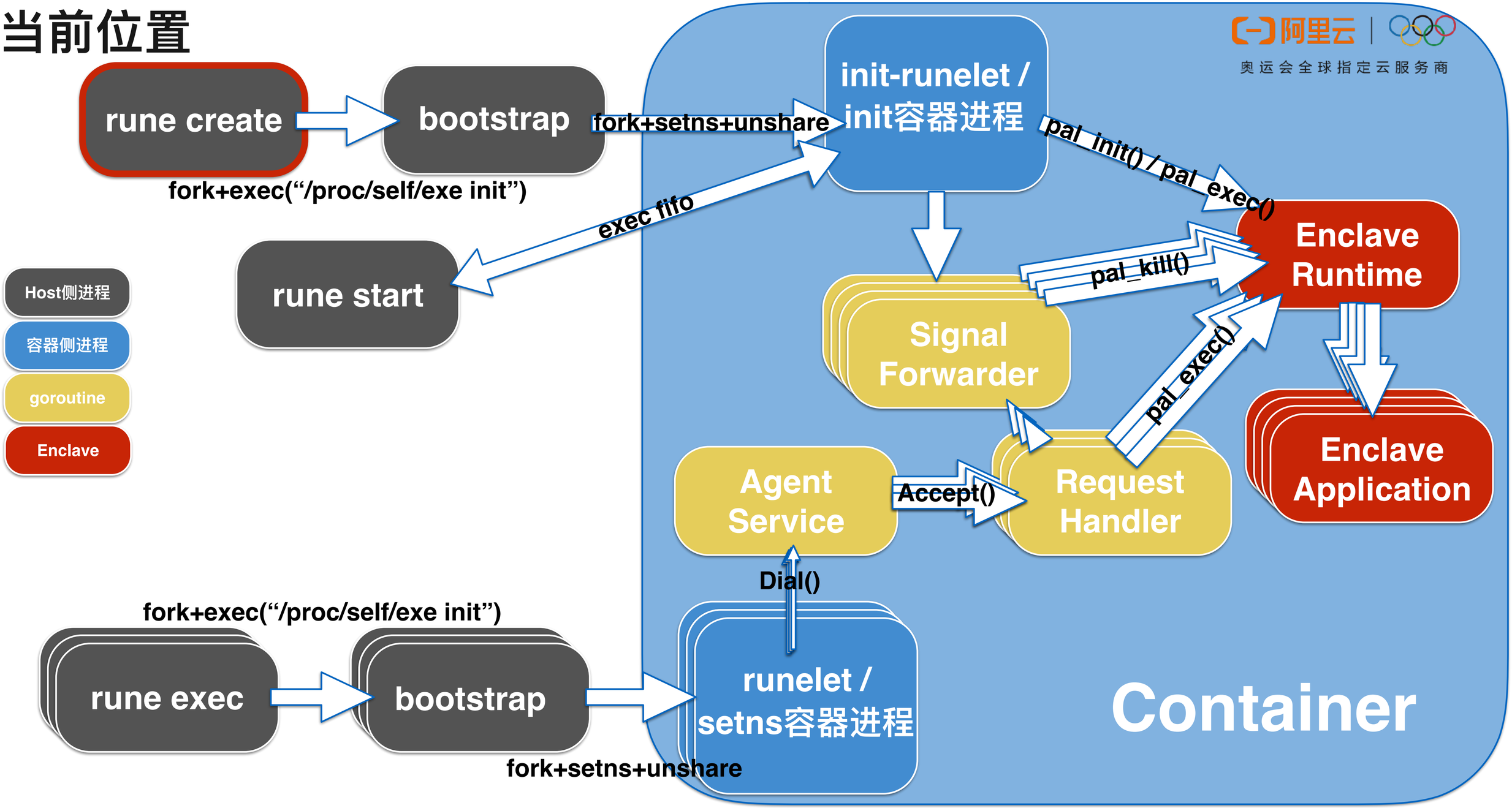
  - ENCLAVE环境变量的优先级高于Annotations

阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet /
init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave
Runtime
pal_kill()
Signal
Forwarder
pal_exec()
Enclave
Application
Agent
Service
Accept()
Request
Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet /
setns容器进程
fork+setns+unshare
Container

解析enclave配置

```
@libcontainer/specconv/spec_linux.go
func CreateLibcontainerConfig(opts *CreateOpts) (*configs.Config, error) {
…
    config := &configs.Config{
        Rootfs:          rootfsPath,
        NoPivotRoot:     opts.NoPivotRoot,
        Readonlyfs:      spec.Root.Readonly,
        Hostname:        spec.Hostname,
        Labels:          append(labels, fmt.Sprintf("bundle=%s", cwd)),
…
    }

    // Initialize enclave configuration as early as possible, because it will provide
    // the hint about whether touching with enclave devices or not in container.
    createEnclaveConfig(spec, config)
…
```

- 此时，config.json中的enclave配置已经被反序列化为libcontainer.Config.Enclave对象。
- 容器环境变量ENCLAVE_TYPE、ENCLAVE_RUNTIME_PATH和ENCLAVE_RUNTIME_ARGS能够覆盖config.json中的annotations属性集中关于enclave的配置值。
- 后续代码可以简单地通过判断config.Enclave是否为nil，来决定是把当前的OCI Runtime rune当做runc用，还是当做rune用。这个特点非常重要。在K8s场景中，POD内的第一个被称为pause的容器就是个普通容器，rune在创建它的时候必须完全按照runc的逻辑来创建；因此rune必须同时具备管理普通容器和enclave容器的完整功能。

```
func createEnclaveConfig(spec *specs.Spec, config *configs.Config) {
…
        env := &spec.Process.Env
        etype := filterOut(env, "ENCLAVE_TYPE")
        if etype == "" {
                etype = libcontainerUtils.SearchLabels(config.Labels, "enclave.type")
                if etype == "" {
                        etype = configs.EnclaveHwDefault
                }
        }

        path := filterOut(env, "ENCLAVE_RUNTIME_PATH")
        if path == "" {
                path = libcontainerUtils.SearchLabels(config.Labels, "enclave.runtime.path")
        }
        args := filterOut(env, "ENCLAVE_RUNTIME_ARGS")
        if args == "" {
                args = libcontainerUtils.SearchLabels(config.Labels, "enclave.runtime.args")
        }
        if args != "" {
                a := strings.Split(args, ",")
                args = strings.Join(a, " ")
        }

        if etype != "" {
                config.Enclave = &configs.Enclave{
                        Type: etype,
                        Path: path,
                        Args: args,
                }
        }
}
```

未修改的runc代码
新实现的rune代码

“透传”/var/run/rune/aesmd目录

```
@libcontainer/specconv/spec_linux.go
func CreateLibcontainerConfig(opts *CreateOpts) (*configs.Config, error) {
…
    createEnclaveConfig(spec, config)
…
    for _, m := range spec.Mounts {
        config.Mounts = append(config.Mounts, createLibcontainerMount(cwd, m))
    }
    if config.Enclave != nil {
        config.Mounts = append(config.Mounts, createLibenclaveMount(cwd))
    }
…
```

```
func createLibenclaveMount(cwd string) *configs.Mount {
        return &configs.Mount{
                Device:           "bind",
                Source:           “/var/run/aesmd",
                Destination:      “/var/run/aesmd",
                Flags:            unix.MS_BIND | unix.MS_REC,
                PropagationFlags: []int{unix.MS_PRIVATE | unix.MS_REC},
        }
}
```

未修改的runc代码
新实现的rune代码

- 将/var/run/aesmd目录bind mount到容器中。
- 容器通过该目录下的aesm.socket文件与host侧的aesm服务进行通信

“透传”enclave设备


```
@libcontainer/specconv/spec_linux.go
func CreateLibcontainerConfig(opts *CreateOpts) (*configs.Config, error) {
…
    if config.Enclave != nil {
        config.Mounts = append(config.Mounts, createLibenclaveMount(cwd))
    }
…
    if err := createDevices(spec, config); err != nil {
…
}
```

```
func createEnclaveDevices(devices []*configs.Device, etype string, fn
func(dev configs.Device)) {
…
        // Create the enclave devices not explicitly specified
        for _, d := range exclusiveDevs {
                dev, err := intelSgxDev(d)
                if err != nil {
                        continue
                }

                fn(dev)
        }
}
```

未修改的runc代码
新实现的rune代码

```
func createDevices(spec *specs.Spec, config *configs.Config) error {
    // add whitelisted devices
    config.Devices = []*configs.Device{
        {
            Type:     'c',
            Path:     "/dev/null",
            Major:    1,
            Minor:    3,
            FileMode: 0666,
            Uid:      0,
            Gid:      0,
        },
…
    }
    if spec.Linux != nil {
        for _, d := range spec.Linux.Devices {
…
            device := &configs.Device{
…
            }
            config.Devices = append(config.Devices, device)
        }
    }
    if config.Enclave != nil {
        createEnclaveDeviceConfig(&config.Devices, config.Enclave.Type)
    }
    return nil
}
```

```
func createEnclaveDeviceConfig(devices *[]*configs.Device, etype string) {
    createEnclaveDevices(*devices, etype, func(dev *configs.Device) {
        dev.FileMode = 0666
        dev.Uid = 0
        dev.Gid = 0
        *devices = append(*devices, dev)
    })
}
```

- 根据enclave配置中的Type参数，动态透传需要在容器中自动创建的Enclave设备节点。
- 以Intel SGX enclave为例：
  - /dev/isgx: misc字符设备（for out-of-tree sgx驱动）
  - /dev/sgx/enclave：misc字符设备（for in-tree sgx驱动）

在cgroup中将enclave设备加白

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/specconv/spec_linux.go
func CreateLibcontainerConfig(opts *CreateOpts) (*configs.Config, error) {
…
    if err := createDevices(spec, config); err != nil {
…
    c, err := createCgroupConfig(opts, config)
…
    config.Cgroups = c
…
}

func createCgroupConfig(opts *CreateOpts, config *configs.Config) (*configs.Cgroup, error) {
…
    c.Resources.Devices = append(c.Resources.Devices, AllowedDevices...)
    if config.Enclave != nil {
        createEnclaveCgroupConfig(&c.Resources.Devices, config.Enclave.Type)
    }
    return c, nil
}

func createEnclaveCgroupConfig(devices *[]*configs.Device, etype string) {
    createEnclaveDevices(*devices, etype, func(dev *configs.Device) {
        dev.Permissions = "rwm"
        dev.Allow = true
        *devices = append(*devices, dev)
    })
}
```

未修改的runc代码

新实现的rune代码

验证enclave配置的有效性


```
@libcontainer/configs/validate/validator.go
func (v *ConfigValidator) Validate(config *configs.Config) error {
…
    if err := v.enclave(config); err != nil {
…
    return nil
}

func (v *ConfigValidator) enclave(config *configs.Config) error {
    if config.Enclave == nil {
        return nil
    }

    if !libenclave.IsEnclaveEnabled(config.Enclave) {
        return fmt.Errorf("Enclave hardware type (%v) is not supported", config.Enclave.Type)
    }

    if config.Enclave.Path == "" {
        return fmt.Errorf("enclave runtime path is not configured")
    }

    if _, err := os.Stat(config.Enclave.Path); err != nil {
        return err
    }

    return nil
}
```

未修改的runc代码
新实现的rune代码

- 这里的逻辑是验证实际运行环境与容器配置中有关enclave配置的矛盾点。目的是在容器真正创建之前就能将比较明显的错误配置以及与实际运行环境不符的错误配置检测出来。
- 如果没有设置与enclave相关的参数（config.Enclave == nil），则rune在行为上与runc完全一致；这个特点是将rune运用在K8s上的关键所在。

libenclave包初始化


```
@libenclave/init.go
func IsEnclaveEnabled(e *configs.Enclave) bool {
    if e == nil {
        return false
    }

    if !IsEnclaveHwEnabled(e.Type) {
        return false
    }

    return true
}

// Check whether enclave-based hardware is supported or not
func IsEnclaveHwEnabled(etype string) bool {
    if etype == "" && enclaveHwType != "" {
        return true
    }

    return etype == enclaveHwType
}

var (
    enclaveHwType string = ""
)

func init() {
        _, _ = user.Lookup("")
        _, _ = net.LookupHost("")

        if intelsgx.IsSgxSupported() {
                enclaveHwType = configs.EnclaveHwIntelSgx
        }
}
```

未修改的runc代码
新实现的rune代码

- 检查当前系统是否支持enclave硬件特性。
- 目前libenclave包仅支持Intel SGX enclave。
- 在rune的设计中，runelet进程是非常特殊的。它是容器进程，但是它又不像runc处理容器进程那样令其execve()容器内的程序；runelet进程是一个进入了容器的namespace并“半容器”化的host进程。这个设计会带来一些兼容性问题：我们遇到过启动Ubuntu容器的时候，由于glibc动态依赖libnss导致在容器侧加载libnss.so的问题。这里的user.Lookup和net.LookipHost都是为了在rune进程还处于host侧的时候就加载libnss，从而避免在runelet进程（runelet进程是rune进程的子进程，因此rune进程加载的libnss会继承给runelet进程）进入容器后又因glibc间接通过dlopen加载libnss进而出现未定义符号的兼容性问题。
- 这个设计意味着任何Enclave Runtime PAL的实现都要遵守以下约束条件：如果Enclave Runtime PAL在运行时需要通过dlopen加载共享库，那么必须在rune进程还在host侧的时候预先加载好。

libenclave/intelsgx包初始化

阿里云 | 奥运会全球指定云服务商

@libenclave/intelsgx/device_linux.go

```
/*
#cgo linux LDFLAGS: -ldl
#include <stdlib.h>
#include <dlfcn.h>
*/
import "C"
…
func preloadSgxPswLib() {
        path := C.CString("libsgx_launch.so")
        C.dlopen(path, C.RTLD_NOW)
        C.free(unsafe.Pointer(path))
}

func init() {
        preloadSgxPswLib()
}
```

未修改的runc代码

新实现的rune代码

- 检查当前系统是否支持enclave硬件特性。
- 目前libenclave包仅支持Intel SGX enclave。
- 与事先加载libnss的workaround逻辑类似，这里的workaround确保了在host侧预先加载Intel SGX PSW的库libsgx_launch.so，规避在runelet进程已经进入了容器环境后调用PAL API时Enclave Runtime PAL调用dlopen需要从容器环境加载libsgx_launch.so的问题（比如容器环境是alpine的话，为alpine适配Intel SGX PSW库还是比较麻烦的）。

创建bootstrap进程(1)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/utils_linux.go
func (r *runner) run(config *specs.Process) (int, error) {

…
        var (
                detach = r.detach || (r.action == CT_ACT_CREATE)
        )

        if detach {
                process.Detached = true
        }
…
        switch r.action {
        case CT_ACT_CREATE:
                err = r.container.Start(process)
…
        case CT_ACT_RUN:
                err = r.container.Run(process)
…
}
```

```
@libcontainer/container_linux.go
func (c *linuxContainer) Start(process *Process) error {
…
    if process.Init {
        if err := c.createExecFifo(); err != nil {
…
    }
    if err := c.start(process); err != nil {
…
    return nil
}

func (c *linuxContainer) start(process *Process) error {
    parent, err := c.newParentProcess(process)
…
}
```

未修改的runc代码
新实现的rune代码

- 记录当前runc是处于detached模式还是前台模式，后面的runelet日志处理逻辑需要知道这个信息。
- rune create/run创建init容器进程时，需要创建额外的exec fifo文件，用于parent rune与init容器进程进行同步。
- newParnetProcess()负责创建bootstrap进程的抽象对象。

创建bootstrap进程(2)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) newParentProcess(p *Process) (parentProcess, error) {
    parentInitPipe, childInitPipe, err := utils.NewSockPair("init")
…
    if c.config.Enclave != nil {
        if p.Init {
            uid, err := c.Config().HostRootUID()
…
            gid, err := c.Config().HostRootGID()
…
            p.AgentPipe, err = libenclave.CreateParentAgentPipe(c.root, uid, gid)
…
        } else {
            p.AgentPipe, err = libenclave.CreateChildAgentPipe(c.root)
…
        }
    }

    parentLogPipe, childLogPipe, err := os.Pipe()
…
    cmd, err := c.commandTemplate(p, childInitPipe, childLogPipe, p.AgentPipe, p.Detached)
…
}
```

未修改的runc代码
新实现的rune代码

- rune create/run和rune exec都需要创建一个子进程来执行容器的bootstrap工作
  - 如果是rune create/run创建的bootstrap进程，则需要事先创建好Agent Service的服务端agent.sock文件。
  - 因为agent.sock文件位于host侧的容器实例的运行状态目录中，因此该文件不会被透传到容器侧，这就需要“事先”在还能访问agent.sock的时候打开它的socket fd，并在之后将socket fd透传给bootstrap进程和runelet进程。

创建bootstrap进程(3)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) commandTemplate(p *Process, childInitPipe *os.File, childLogPipe
*os.File, agentPipe *os.File, detached bool) *exec.Cmd {
        cmd := exec.Command(c.initPath, c.initArgs[1:]...)
        cmd.Args[0] = c.initArgs[0]
        cmd.Stdin = p.Stdin
        cmd.Stdout = p.Stdout
        cmd.Stderr = p.Stderr
        cmd.Dir = c.config.Rootfs
…
        cmd.ExtraFiles = append(cmd.ExtraFiles, childInitPipe)
        cmd.Env = append(cmd.Env,
                fmt.Sprintf("_LIBCONTAINER_INITPIPE=%d", stdioFdCount+len(cmd.ExtraFiles)-1),
                fmt.Sprintf("_LIBCONTAINER_STATEDIR=%s", c.root),
        )

        cmd.ExtraFiles = append(cmd.ExtraFiles, childLogPipe)
        cmd.Env = append(cmd.Env,
                fmt.Sprintf("_LIBCONTAINER_LOGPIPE=%d", stdioFdCount+len(cmd.ExtraFiles)-1),
                fmt.Sprintf("_LIBCONTAINER_LOGLEVEL=%s", p.LogLevel),
        )
…
```

未修改的runc代码
新实现的rune代码

- parent rune进程之后会通过fork+execve(“/proc/self/exe init”)的方式来创建bootstrap进程。
- 关键的fd都通过对应的_LIBCONTAINER_*环境变量传给bootstrap进程。
- 后面可以看到，parent rune向bootstrap和runelet进程传递信息的方式主要依靠init pipe。不过通过这种方式传递的信息大多是从父进程继承过来的，比如文件fd。fd本质上就是数字，而数字的含义需要“提点”。这些环境变量具有_LIBCONTAINER前缀，其目的也仅仅是为了“提点”文件fd的含义，不会真正的作用于容器进程。

创建bootstrap进程(4)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) commandTemplate(p *Process, childInitPipe *os.File, childLogPipe
*os.File, agentPipe *os.File, detached bool) *exec.Cmd {
…
        cmd.Env = append(cmd.Env,
                fmt.Sprintf("_LIBCONTAINER_LOGPIPE=%d", stdioFdCount+len(cmd.ExtraFiles)-1),
                fmt.Sprintf("_LIBCONTAINER_LOGLEVEL=%s", p.LogLevel),
        )
…
        if c.config.Enclave != nil {
                if agentPipe != nil {
                        cmd.ExtraFiles = append(cmd.ExtraFiles, agentPipe)
                        cmd.Env = append(cmd.Env,
                                fmt.Sprintf("_LIBENCLAVE_AGENTPIPE=%d",
stdioFdCount+len(cmd.ExtraFiles)-1))
                }

                if c.config.Enclave.Path != "" {
                        cmd.Env = append(cmd.Env,
"_LIBENCLAVE_PAL_PATH="+c.config.Enclave.Path)
                }

                if detached {
                        cmd.Env = append(cmd.Env, "_LIBENCLAVE_DETACHED=1")
                }
        }
…
        return cmd
}
```

- Agent Service的socket fd会通过环境变量_LIBENCLAVE_AGENTFD传给bootstrap进程。
- Enclave Runtime PAL的文件路径通过_LIBENCLAVE_PAL_PATH传给bootstrap进程。
- rune的运行模式（detached或前台模式）通过_LIBENCLAVE_DETACHED传给bootstrap进程。

未修改的runc代码
新实现的rune代码

创建bootstrap进程(5)

```
@libcontainer/container_linux.go
func (c *linuxContainer) newParentProcess(p *Process) (parentProcess, error) {
…
    if !p.Init {
        return c.newSetnsProcess(p, cmd, messageSockPair, logFilePair)
    }

    if err := c.includeExecFifo(cmd); err != nil {
…
    return c.newInitProcess(p, cmd, messageSockPair, logFilePair)
}

func (c *linuxContainer) includeExecFifo(cmd *exec.Cmd) error {
    fifoName := filepath.Join(c.root, execFifoFilename)
    fifoFd, err := unix.Open(fifoName, unix.O_PATH|unix.O_CLOEXEC, 0)
…
    cmd.ExtraFiles = append(cmd.ExtraFiles, os.NewFile(uintptr(fifoFd), fifoName))
    cmd.Env = append(cmd.Env,
        fmt.Sprintf("_LIBCONTAINER_FIFOFD=%d", stdioFdCount+len(cmd.ExtraFiles)-1))
    return nil
}

func (c *linuxContainer) newInitProcess(p *Process, cmd *exec.Cmd, messageSockPair,
logFilePair filePair) (*initProcess, error) {
    cmd.Env = append(cmd.Env, "_LIBCONTAINER_INITTYPE="+string(initStandard))
…
    init := &initProcess{
        cmd:             cmd,
        messageSockPair: messageSockPair,
        logFilePair:     logFilePair,
…
        config:          c.newInitConfig(p),
…
        bootstrapData:   data,
…
    }
    c.initProcess = init
    return init, nil
}
```

未修改的runc代码
新实现的rune代码

- 如果是rune create/run创建的bootstrap进程，则需要事先创建好exec fifo文件，并将该文件fd透传给bootstrap进程。
  - 因为exec fifo文件位于容器实例的运行状态目录，而该目录位于host侧，不会透传到容器侧，因此就需要“事先”在当前进程还能访问它的时候打开其文件fd，并将文件fd透传给bootstrap进程。
  - exec fifo的fd会通过环境变量_LIBCONTAINER_FIFOFD传给bootstrap进程。
  - 将环境变量_LIBCONTAINER_INITTYPE设为standard，表示bootstrap进程之后要创建的是容器init进程。
  - 构造好bootstrap数据和init config，之后发送给bootstrap进程。
- 如果是rune exec创建的bootstrap进程，不需要使用exec fifo文件。

创建bootstrap进程(6): _LIBCONTAINER/_LIBENCLAVE_*的含义

名称	来源	目标	作用
_LIBCONTAINER_INITTYPE	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	◦ standard：告诉bootstrap进程需要创建的是一个init容器进程 ◦ setns: 告诉bootstrap进程需要创建的是一个setns容器进程
_LIBCONTAINER_FIFOFD	rune create/run	init-runelet进程	在执行容器入口点程序前，用于同步rune create/run与init-runelet进程
_LIBCONTAINER_INITPIPE	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	容器的init pipe fd，在本阶段的作用有： ◦ parent rune进程向boostrap进程发送bootstrap数据。 ◦ parent rune进程读取到bootstrap进程和init容器进程的pid。 ◦ parent rune进程向容器进程发送init config。 ◦ 容器进程/runelet与parent rune进程间的各种状态同步。 ◦ parent rune进程向runelet进程发送的enclave init config。 ◦ 通过关闭该init pipe fd向parent rune进程告知runelet进程已经完成了全部的初始化操作。
_LIBCONTAINER_LOGPIPE	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	◦ 容器log pipe的fd ◦ 容器日志包含的是从bootstrap进程一直到runelet进程运行前这期间打印的日志(JSON格式)
_LIBCONTAINER_LOGLEVEL	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	定义了容器日志的打印级别
_LIBENCLAVE_AGENTPIPE	rune create/run/exec	runelet进程	◦ init-runelet: agent.sock服务端的socket fd ◦ runelet：agent.sock客户端的socket fd
_LIBENCLAVE_PAL_PATH	rune create/run/exec	bootstrap进程	Enclave Runtime PAL的文件路径
_LIBENCLAVE_DETACHED	rune create/run/exec	runelet进程	rune的运行模式：前台或detached；该信息影响了ruenlet进程的容器日志打印行为

创建bootstrap进程(7)


```
@libcontainer/container_linux.go
func (c *linuxContainer) start(process *Process) error {
…
    parent.forwardChildLogs()

    if err := parent.start(); err != nil {
…
    return nil
}

@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
    err := p.cmd.Start()
…
}
```

未修改的runc代码
新实现的rune代码

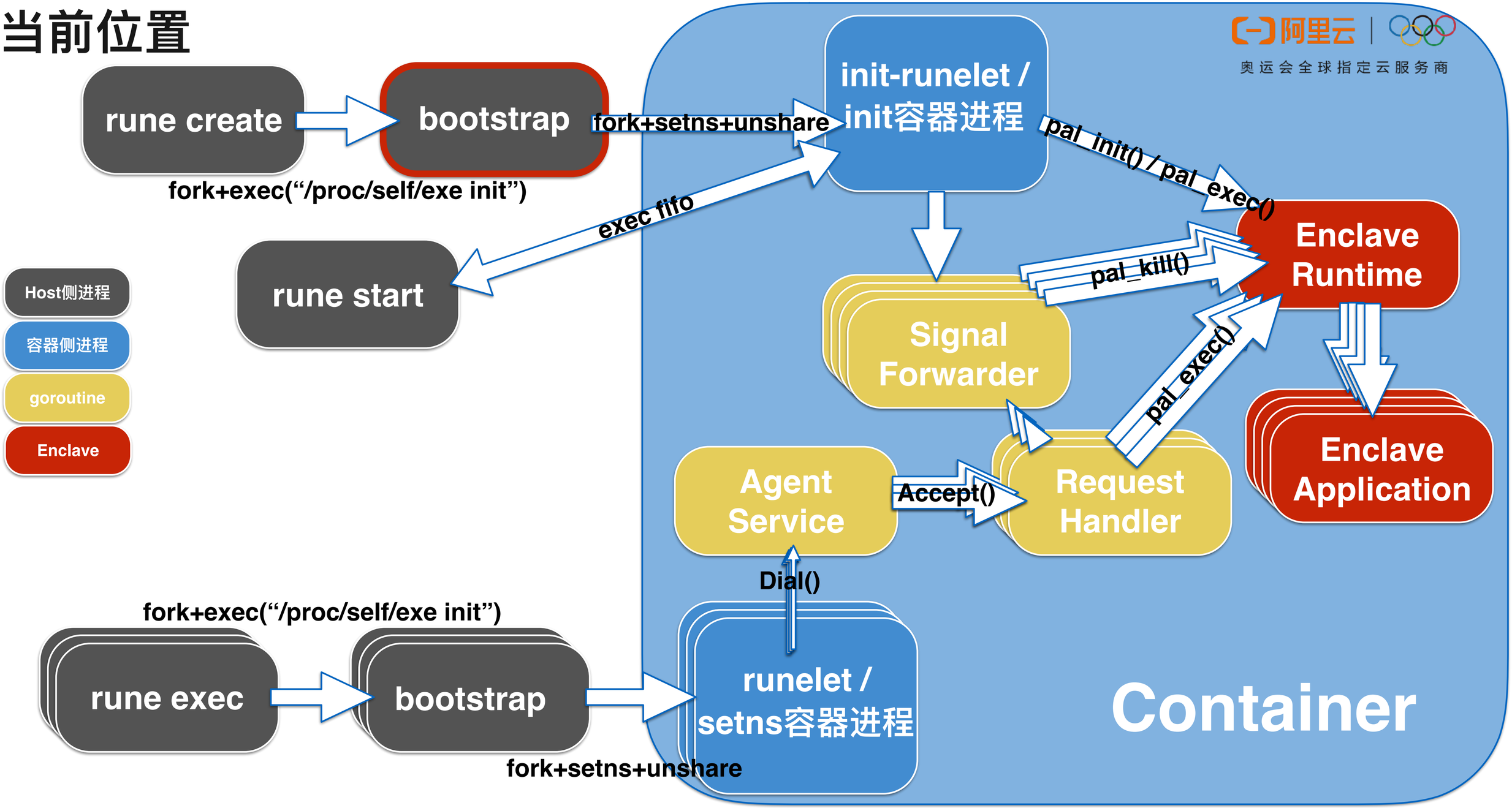
- parent rune进程通过启动一个goroutine来转发bootstrap进程/容器进程/runelet进程通过child log pipe发送过来的容器日志（这里的容器日志的概念不同于K8s中的容器日志；本质上，K8s中的容器日志等价于容器进程的stdio，而这里的容器日志仅用于显示容器和runelet进程在初始化阶段的信息）。
- 至此，创建bootstrap进程的全部工作已经准备好，接下来正式通过fork+execve(“/proc/self/exe init”)启动bootstrap进程。

阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- 环境搭建
- init-runelet进程
- init容器进程
- rune exec执行流程

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet /
init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave
Runtime
pal_kill()
Signal
Forwarder
pal_exec()
Enclave
Application
Agent
Service
Accept()
Request
Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet /
setns容器进程
fork+setns+unshare
Container

bootstrap初始化(1)：加载Enclave Runtime PAL

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/nsenter/nsexec.c
void nsexec(void)
{
…
    struct nlconfig_t config = { 0 };
…
    setup_logpipe();
…
    pipenum = initpipe();
…
    if (enclave_configured()) {
        int ret = load_enclave_runtime();
        …
    }

    write_log(DEBUG, "nsexec started");
…
}
```

- parent rune进程通过_LIBCONTAINER_*环境变量向bootstrap进程传递容器日志和init pipe的fd。
- 通过_LIBENCLAVE_PAL_PATH加载Enclave Runtime PAL，并找出和记录所有的PAL API接口的符号地址；在rune的设计中，PAL加载与PAL执行这两个操作是跨容器的。这个约束条件决定了之后在执行Enclave Runtime PAL时PAL本身的实现中不能再有dlopen的操作，否则被dlopen加载的so必须要适配容器侧执行环境（比如容器环境如果是alpine的话，适配工作会比较困难）。

```
@libcontainer/nsenter/enclave.c
bool enclave_configured(void)
{
        const char *p = getenv("_LIBENCLAVE_PAL_PATH");
        if (p == NULL || *p == '\0')
                return false;
        return true;
}
```

```
int load_enclave_runtime(void)
{
…
        pal_path = getenv("_LIBENCLAVE_PAL_PATH");
…
        dl = dlopen(pal_path, RTLD_NOW);
…
#define DLSYM(fn)                                                       \
        do {                                                            \
                fptr_pal_ ## fn = dlsym(dl, "pal_" #fn);                \
                write_log(DEBUG, "dlsym(%s) = %p", "pal_" #fn, fptr_pal_ ## fn);  \
        } while (0)

        DLSYM(get_version);
        DLSYM(init);
        DLSYM(create_process);
        DLSYM(exec);
        DLSYM(kill);
        DLSYM(destroy);
#undef DLSYM
        return 0;
}
```

未修改的runc代码
新实现的rune代码

bootstrap初始化(2): 解析bootstrap数据

```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
    defer p.messageSockPair.parent.Close()
    err := p.cmd.Start()
…
    if _, err := io.Copy(p.messageSockPair.parent, p.bootstrapData); err != nil {
…
}
```

```
@libcontainer/nsenter/nsexec.c
void nsexec(void)
{
…
    struct nlconfig_t config = { 0 };
…
    write_log(DEBUG, "nsexec started");
…
    nl_parse(pipenum, &config);
…
}

struct nlconfig_t {
        char *data;

        /* Process settings. */
        uint32_t cloneflags;
        char *oom_score_adj;
        size_t oom_score_adj_len;

        /* User namespace settings. */
        char *uidmap;
        size_t uidmap_len;
        char *gidmap;
        size_t gidmap_len;
        char *namespaces;
        size_t namespaces_len;
        uint8_t is_setgroup;

        /* Rootless container settings. */
        uint8_t is_rootless_euid;       /* boolean */
        char *uidmappath;
        size_t uidmappath_len;
        char *gidmappath;
        size_t gidmappath_len;
};
```

- bootstrap进程阻塞在init pipe上，直到读取到从parent rune进程发送过来的bootstrap数据。
- 这些bootstrap数据在性质上属于必须由bootstrap进程或其子进程/孙进程执行才能生效的数据，尤其是和进程namespace化相关的配置，因此只能通过进程间通信的方式传给bootstrap亲自执行。

未修改的runc代码
新实现的rune代码

bootstrap初始化(3): 交互阶段的整体逻辑

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/nsenter/nsexec.c
void nsexec(void)
{
…
    if (socketpair(AF_LOCAL, SOCK_STREAM, 0, sync_child_pipe) < 0)
…
    if (socketpair(AF_LOCAL, SOCK_STREAM, 0, sync_grandchild_pipe) < 0)
…
    switch (setjmp(env)) {
    case JUMP_PARENT:{
…
        child = clone_parent(&env, JUMP_CHILD);
…
    case JUMP_CHILD:{
…
        child = clone_parent(&env, JUMP_INIT);
…
    case JUMP_INIT:{
…
        return;
    }
…
    bail("should never be reached");
}
```

未修改的runc代码
新实现的rune代码

- 创建两对儿socket pair，分别用于bootstrap进程与其子进程，以及bootstrap进程与其孙进程间的交互。
- 通过交互，逐步达成将孙进程变为容器进程的最终目标。
- bootstrap进程通过jump buffer执行JUMP_PARENT分支的代码。
- bootstrap进程的子进程通过jump buffer执行JUMP_CHILD分支的代码。
- bootstrap进程的孙进程通过jump buffer执行JUMP_INIT分支的代码。
- 最终，bootstrap进程和子进程全都退出，而孙进程会成为容器进程，并将parent rune进程认定为其父进程。

bootstrap初始化(4): bootstrap进程与子进程的交互


```
@libcontainer/nsenter/nsexec.c
   case JUMP_PARENT:{
         bool ready = false;
…
         child = clone_parent(&env, JUMP_CHILD);
…
         while (!ready) {
…
            syncfd = sync_child_pipe[1];
…
            if (read(syncfd, &s, sizeof(s)) != sizeof(s))
…
            switch (s) {
            case SYNC_USERMAP_PLS:
…
               s = SYNC_USERMAP_ACK;
               if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
               break;
            case SYNC_RECVPID_PLS:{
                  first_child = child;
…
                  if (read(syncfd, &child, sizeof(child)) != sizeof(child)) {
…
                  s = SYNC_RECVPID_ACK;
                  if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
                  len = dprintf(pipenum, "{\"pid\": %d, \"pid_first\": %d}\n",
child, first_child);
…
               }
               break;
            case SYNC_CHILD_READY:
               ready = true;
               break;
…
            }
         }
```

未修改的runc代码

新实现的rune代码

- 首先fork出bootstrap进程的子进程，并将其入口点设为JUMP_CHILD分支。
- 与子进程进行交互
  - SYNC_USERMAP_PLS：子进程请求bootstrap进程为其设置uid/pid mapping。
  - SYNC_RECVPID_PLS：子进程告诉bootstrap进程其创建的子进程（即bootstrap进程的孙进程）的PID，然后bootstrap进程将子进程和孙进程的PID通过init pipe发送给parent rune进程。
  - SYNC_CHILD_READY：子进程通知bootstrap进程它完成了全部工作，即将退出。
- 在完成与子进程的交互之后，bootstrap进程将与其孙进程进行交互。

bootstrap初始化(5): 子进程与bootstrap进程的交互

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/nsenter/nsexec.c
    case JUMP_CHILD:{
…
            syncfd = sync_child_pipe[0];
…
            if (config.namespaces)
                join_namespaces(config.namespaces);
…
            if (unshare(config.cloneflags & ~CLONE_NEWCGROUP) < 0)
…
            child = clone_parent(&env, JUMP_INIT);
…
            s = SYNC_RECVPID_PLS;
            if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
            if (write(syncfd, &child, sizeof(child)) != sizeof(child)) {
…
            if (read(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
            s = SYNC_CHILD_READY;
            if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
            exit(0);
        }
```

未修改的runc代码
新实现的rune代码

- 根据bootstrap数据中有关namespace的设置，通过setns()加入到需要和其他进程共享的namespace中。
- 通过unshare()创建并加入到无需与其他进程共享的全新namespace中。
- fork出子进程，即bootstrap进程的孙进程，并将其入口点设为JUMP_INIT分支。孙进程会继承子进程的所有namespace。
- 与bootstrap进程进行交互
  - SYNC_RECVPID_PLS：告诉bootstrap进程孙进程的PID。
  - SYNC_CHILD_READY：告诉bootstrap进程子进程已经完成了全部工作，然后子进程退出。

bootstrap初始化(6): parent rune进程等待bootstrap的子进程退出

```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
    if _, err := io.Copy(p.messageSockPair.parent, p.bootstrapData); err != nil {
…
    childPid, err := p.getChildPid()
…
}

func (p *initProcess) getChildPid() (int, error) {
    var pid pid
    if err := json.NewDecoder(p.messageSockPair.parent).Decode(&pid); err != nil {
…
    firstChildProcess, _ := os.FindProcess(pid.PidFirstChild)
…
    _, _ = firstChildProcess.Wait()

    return pid.Pid, nil
}

@libcontainer/nsenter/nsexec.c
static int clone_parent(jmp_buf *env, int jmpval)
{
…
    return clone(child_func, ca.stack_ptr, CLONE_PARENT | SIGCHLD, &ca);
}
```

parent rune进程

```
@libcontainer/nsenter/nsexec.c
    case JUMP_PARENT:{
…
            if (read(syncfd, &s, sizeof(s)) != sizeof(s))
…
            switch (s) {
…
            case SYNC_RECVPID_PLS:{
…
                len = dprintf(pipenum, "{\"pid\": %d,
\"pid_first\": %d}\n", child, first_child);
…
```

bootstrap进程

2

1

3

```
@libcontainer/nsenter/nsexec.c
    case JUMP_CHILD:{
…
        s = SYNC_RECVPID_PLS;
        if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
        if (write(syncfd, &child, sizeof(child)) !=
sizeof(child)) {
…
        if (read(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
        s = SYNC_CHILD_READY;
        if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
        exit(0);
    }
```

子进程

未修改的runc代码
新实现的rune代码

- bootstrap进程在创建子进程时，要求内核（通过CLONE_PARENT）将创建出的子进程的父进程设为自己的父进程，即parent rune进程。
- 子进程在创建孙进程时，同样也要求内核将创建出的孙进程的父进程设为自己的父进程，即parent rune进程。
- 因此事实上bootstrap的子进程和孙进程其实都是parent rune进程的子进程。

bootstrap初始化(7): parent rune进程等待bootstrap进程退出

- bootstrap进程完成与孙进程的同步后，主动退出。
- 孙进程完成与bootstrap进程的交互，成为容器进程；接下来退出C代码执行环境，并开始执行Golang runtime，为下一步执行rune init的Golang代码做好准备。

```
@libcontainer/nsenter/nsexec.c
    case JUMP_PARENT:{
…
            ready = false;
            while (!ready) {
…
                syncfd = sync_grandchild_pipe[1];
…

                s = SYNC_GRANDCHILD;
                if (write(syncfd, &s, sizeof(s)) != sizeof(s)) {
…
                if (read(syncfd, &s, sizeof(s)) != sizeof(s))
…
                switch (s) {
                case SYNC_CHILD_READY:
                    ready = true;
                    break;
…
                }
            }
            exit(0);
        }
```

```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
    childPid, err := p.getChildPid()
…
    if err := p.waitForChildExit(childPid); err != nil {
        return newSystemErrorWithCause(err, "waiting for our first child to exit")
    }
…
}
```

1

2

3

bootstrap进程

parent rune进程

```
@libcontainer/nsenter/nsexec.c
    case JUMP_INIT:{
…
        syncfd = sync_grandchild_pipe[0];
…
        if (read(syncfd, &s, sizeof(s)) != sizeof(s))
…
        if (setsid() < 0)
…
        if (setuid(0) < 0)
…
        if (setgid(0) < 0)
…
        s = SYNC_CHILD_READY;
        if (write(syncfd, &s, sizeof(s)) != sizeof(s))
…
        close(sync_grandchild_pipe[0]);
…
        if (enclave_configured()) {
…
            if (is_init_runelet())
                name = “init-runelet";
            else
                name = “runelet” ;
            prctl(PR_SET_NAME, (unsigned long)name, 0,
0, 0);
        }
        return;
    }
```

孙进程

未修改的runc代码
新实现的rune代码

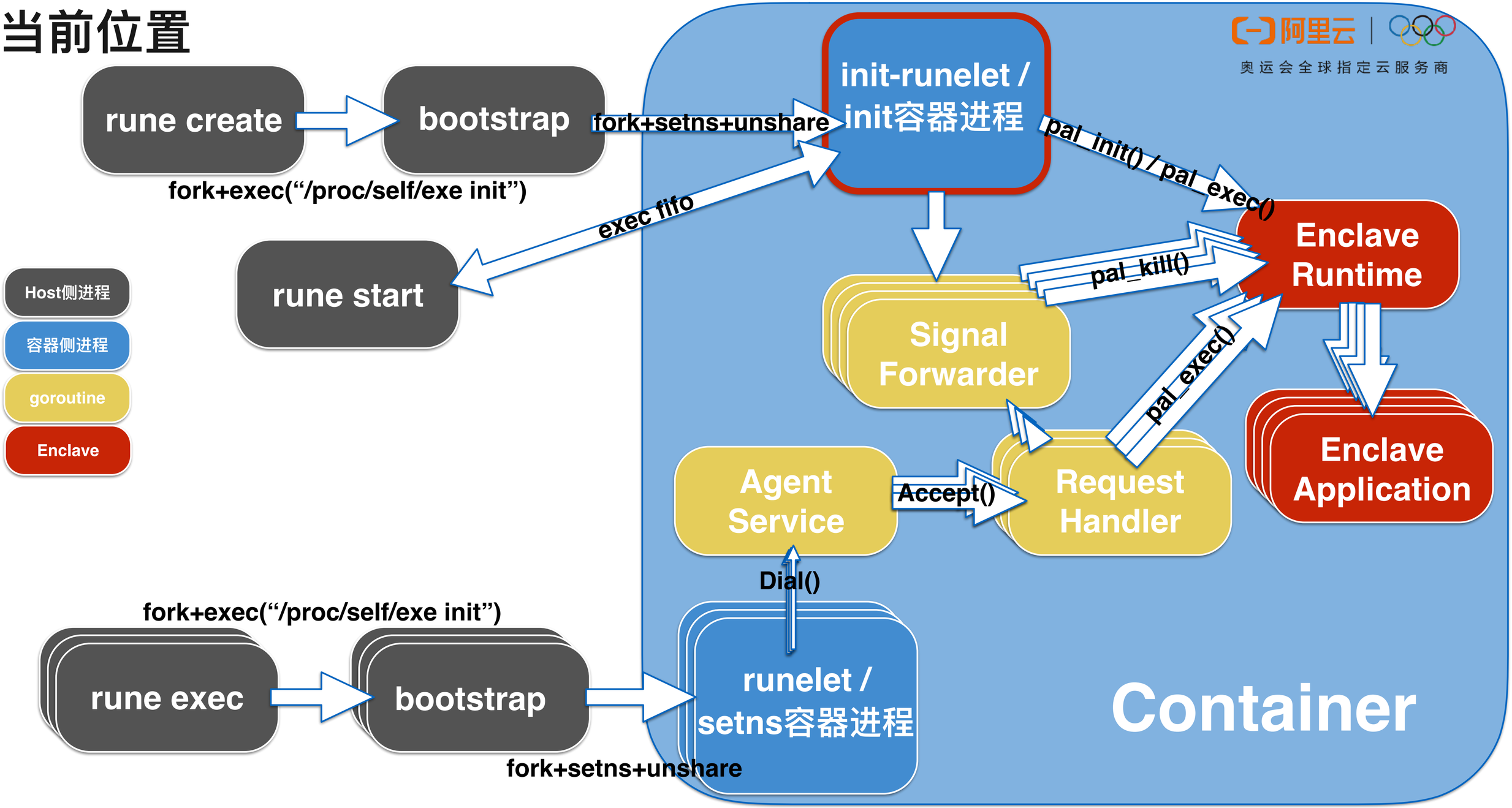
- Parent rune进程等待bootstrap进程退出。
- 至此，parent rune进程与容器进程开始并行执行，但不久两者将再次通过init pipe进行同步。

阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet /
init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave
Runtime
pal_kill()
Signal
Forwarder
pal_exec()
Enclave
Application
Agent
Service
Accept()
Request
Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet /
setns容器进程
fork+setns+unshare
Container

执行包初始化

阿里云 | 奥运会全球指定云服务商

```
@init.go
func init() {
    if len(os.Args) > 1 && os.Args[1] == "init" {
        runtime.GOMAXPROCS(1)
        runtime.LockOSThread()

        level := os.Getenv("_LIBCONTAINER_LOGLEVEL")
        logLevel, err := logrus.ParseLevel(level)
…
        err = logs.ConfigureLogging(logs.Config{
            LogPipeFd: os.Getenv("_LIBCONTAINER_LOGPIPE"),
            LogFormat: "json",
            LogLevel:  logLevel,
        })
…
        logrus.Debug("child process in init()")
    }
}
```

未修改的runc代码
新实现的rune代码

- 在离开C语言运行环境后，容器进程执行Golang Runtime，然后执行这里的包初始化代码。
- 设置容器日志相关的配置

执行rune init命令

阿里云 | 奥运会全球指定云服务商

```
@init.go
var initCommand = cli.Command{
   Name:  "init",
   Usage: `initialize the namespaces and launch the process (do not call it outside of runc)`,
   Action: func(context *cli.Context) error {
      factory, _ := libcontainer.New("")
      if err := factory.StartInitialization(); err != nil {
…
         os.Exit(1)
      }
      panic("libcontainer: container init failed to exec")
   },
}
```

未修改的runc代码
新实现的rune代码

- 对于容器来说，不管是init容器进程还是setns容器进程，最终都是通过执行execve()来运行一个容器应用程序，因此正常情况下不应该从factory.StartInitialization()返回。一旦返回，如果不是在执行execve()的中途遇到了错误，则会导致程序panic()。

- 对于rune来说，绝对不会通过execve()来运行一个容器应用程序，但也要遵守对上述行为的约束和规定，即正常情况下不应该返回，否则会导致程序panic()。

重建关键信息

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/factory_linux.go
func (l *LinuxFactory) StartInitialization() (err error) {
…
    detached        = false
    envInitPipe     = os.Getenv("_LIBCONTAINER_INITPIPE")
    envFifoFd       = os.Getenv("_LIBCONTAINER_FIFOFD")
…
    envLogPipe      = os.Getenv("_LIBCONTAINER_LOGPIPE")
    envLogLevel     = os.Getenv("_LIBCONTAINER_LOGLEVEL")
    envAgentPipe    = os.Getenv("_LIBENCLAVE_AGENTPIPE")
    envDetached     = os.Getenv("_LIBENCLAVE_DETACHED")
…
  pipefd, err = strconv.Atoi(envInitPipe)
…
  var (
    pipe = os.NewFile(uintptr(pipefd), "pipe")
    it   = initType(os.Getenv("_LIBCONTAINER_INITTYPE"))
  )
…
  fifofd = -1
  if it == initStandard {
    if fifofd, err = strconv.Atoi(envFifoFd); err != nil {
…
```

```
  if envLogPipe != "" {
    log, err := strconv.Atoi(envLogPipe)
…
    logPipe = os.NewFile(uintptr(log), "log-pipe")
…
  }

  if envAgentPipe != "" {
    agent, err := strconv.Atoi(envAgentPipe)
…
    agentPipe = os.NewFile(uintptr(agent), "agent-pipe")
…
  }

  if envDetached != "" {
    tmpDetached, err := strconv.Atoi(envDetached)
…
    if tmpDetached != 0 {
      detached = true
    }
  }
…
```

未修改的runc代码
新实现的rune代码

- 读取_LIBCONTAINER_*和_LIBENCLAVE_*环境变量并重建关键信息。

清除从父进程继承过来的环境变量

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/factory_linux.go
func (l *LinuxFactory) StartInitialization() (err error) {
…
    os.Clearenv()
…
```

未修改的runc代码
新实现的rune代码

- 清除容器进程继承自bootstrap进程/parent rune进程的全部环境变量，避免来自host侧的环境变量对容器进程的行为产生影响。

接收init config


```
@libcontainer/factory_linux.go
func (l *LinuxFactory) StartInitialization() (err error) {
    i, err := newContainerInit(it, pipe, consoleSocket, fifofd, logPipe, envLogLevel, agentPipe, detached)
…

@libcontainer/init_linux.go
func newContainerInit(t initType, pipe *os.File, consoleSocket *os.File, fifoFd int, logPipe *os.File, logLevel string, agentPipe *os.File, detached bool)
(initer, error) {
    var config *initConfig
    if err := json.NewDecoder(pipe).Decode(&config); err != nil {
…
    if err := populateProcessEnvironment(config.Env); err != nil {
…
    switch t {
    case initSetns:
        return &linuxSetnsInit{
            pipe:          pipe,
…
            logPipe:       logPipe,
            logLevel:      logLevel,
            agentPipe:     agentPipe,
            detached:      detached,
        }, nil
    case initStandard:
        return &linuxStandardInit{
            pipe:          pipe,
…
            fifoFd:        fifoFd,
            logPipe:       logPipe,
            logLevel:      logLevel,
            agentPipe:     agentPipe,
            detached:      detached,
        }, nil
    }
…
}
```

```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
    if err := p.waitForChildExit(childPid); err != nil {
…
    if err := p.sendConfig(); err != nil {
…
}
```

未修改的runc代码

新实现的rune代码

- 通过init pipe从parent rune侧接收JSON格式的init config。
- 将init config中有关容器环境变量的配置设置到当前容器进程中。
- 将enclave相关的状态信息记录在相关的上下文中，留待后续处理。

完成与parent rune进程的同步

阿里云 | 奥运会全球指定云服务商

@libcontainer/factory_linux.go

```
func (l *LinuxFactory) StartInitialization() (err error) {
    i, err := newContainerInit(it, pipe, consoleSocket, fifofd,
logPipe, envLogLevel, agentPipe, detached)
…
    return i.Init()
}
```

容器init进程

@libcontainer/standard_init_linux.go

```
func (l *linuxStandardInit) Init() error {
        runtime.LockOSThread()
…
        if err := prepareRootfs(l.pipe, l.config); err != nil {
…
        if err := syncParentReady(l.pipe); err != nil {
…
```

@libcontainer/init_linux.go

```
func syncParentReady(pipe io.ReadWriter) error {
        // Tell parent.
        if err := writeSync(pipe, procReady); err != nil {
                return err
        }

        // Wait for parent to give the all-clear.
        return readSync(pipe, procRun)
}
```

@libcontainer/process_linux.go

```
func (p *initProcess) start() (retErr error) {
…
        ierr := parseSync(p.messageSockPair.parent, func(sync *syncT) error {
                switch sync.Type {
                case procReady:
…
                        // Sync with child.
                        if err := writeSync(p.messageSockPair.parent, procRun); err != nil {
…
                        sentRun = true
                case procHooks:
…
                case procEnclaveConfigReq:
…
                case procEnclaveConfigAck:
…
                case procEnclaveInit:
…
                }

                return nil
        })
…
```

parent rune进程

未修改的runc代码
新实现的rune代码

- 在从prepareRootfs()返回后，将无法再访问host rootfs，而是pivot到容器rootfs。
- parent rune进程完成与init容器进程的同步。
- 但同步操作还没有完成，后续还要与init-runelet进程进行同步（procEnclave*）。

调用libenclave初始化代码

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/standard_init_linux.go
func (l *linuxStandardInit) Init() error {
        if err := syncParentReady(l.pipe); err != nil {
…
        if l.config.Config.Enclave != nil {
…
                cfg := &libenclave.RuneletConfig{
                        InitPipe:  l.pipe,
                        LogPipe:   l.logPipe,
                        LogLevel:  l.logLevel,
                        FifoFd:    l.fifoFd,
                        AgentPipe: l.agentPipe,
                        Detached:  l.detached,
                }

                exitCode, err := libenclave.StartInitialization(l.config.Args, cfg)
                if err != nil {
                        return err
                }
                logrus.Debugf("init enclave runtime exit code: %d", exitCode)
                os.Exit(int(exitCode))
                // make compiler happy
                return fmt.Errorf("failed to initialize init-runelet")
        }
…
}
```

未修改的runc代码
新实现的rune代码

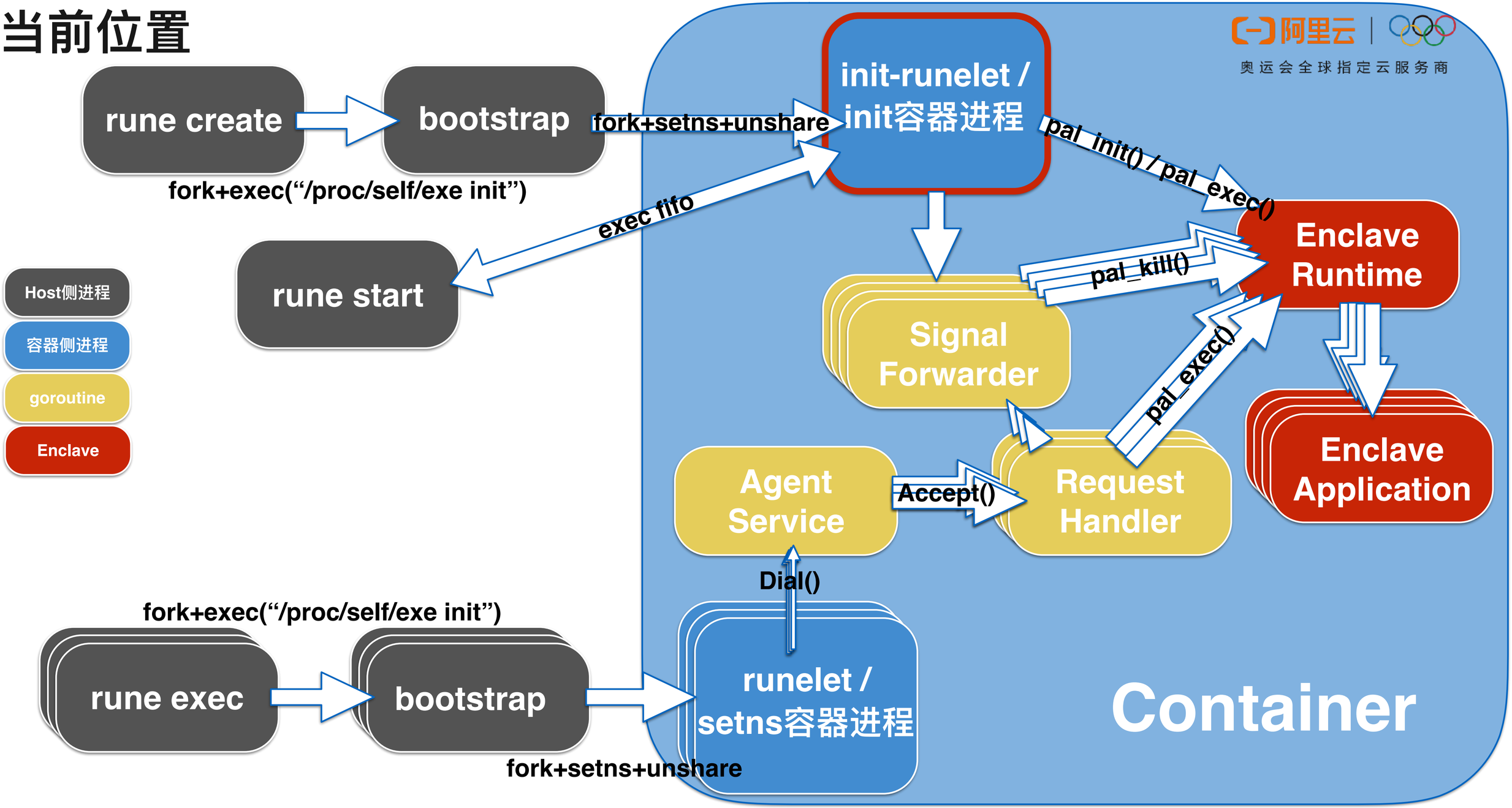
- 此时的init容器进程自身仍旧不是完全容器化的。在运行runc容器的情况，init容器进程还需要通过execve()执行容器入口点才算是完成自身的容器化；而rune的情况则是维持在这个“半容器化”的状态，并开始执libenclave.StartInitialization()，后续的执行流程都将在libenclave中进行。
- 如果Enclave Runtime正常情况退出的话，会通过os.Exit()退出并返回exit code。如果不这样处理，返回到init.go后会触发panic()。

阿里云 | 奥运会全球指定云服务商

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet /
init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave
Runtime
pal_kill()
Signal
Forwarder
pal_exec()
Enclave
Application
Agent
Service
Accept()
Request
Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet /
setns容器进程
fork+setns+unshare
Container

传递enclave配置信息


```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
        ierr := parseSync(p.messageSockPair.parent, func(sync *syncT) error {
                switch sync.Type {
                case procReady:
…
                case procHooks:
```

```
                case procEnclaveConfigReq:
…
                        config := &enclave_configs.InitEnclaveConfig{
                                Type: p.config.Config.Enclave.Type,
                                Path: p.config.Config.Enclave.Path,
                                Args: p.config.Config.Enclave.Args,
                        }
                        err := utils.WriteJSON(p.messageSockPair.parent,
config)
…
                case procEnclaveConfigAck:
…
                case procEnclaveInit:
…
                        err := writeSync(p.messageSockPair.parent,
procEnclaveReady)
…
                }
```

parent rune进程

```
                return nil
        })
…
}
```

未修改的runc代码
新实现的rune代码

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32,
err error) {
        logLevel := cfg.LogLevel

        // Determine which type of runelet is initializing.
        fifoFd := cfg.FifoFd

        // Retrieve the init pipe fd to accomplish the enclave configuration
        // handshake as soon as possible with parent rune.
        initPipe := cfg.InitPipe

        if err = writeSync(initPipe, procEnclaveConfigReq); err != nil {
…
        var config *configs.InitEnclaveConfig
        if err = json.NewDecoder(initPipe).Decode(&config); err != nil {
…
        if err = writeSync(initPipe, procEnclaveConfigAck); err != nil {
…
```

init-runelet进程

- 通过init pipe从parent rune进程处获取到JSON格式的enclave配置信息。示例：

```
"enclave": {
        "type": "intelSgx",
        "path": "/usr/lib/libocclum-pal.so",
        "args": ".occlum"
}
```

- 在此之后，parent rune继续等待与init-runelet进程之间的最后一个同步值procEnclaveInit。

执行enclave runtime初始化

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32,
err error) {
…
        if err = writeSync(initPipe, procEnclaveConfigAck); err != nil {
…
        var rt *runtime.EnclaveRuntimeWrapper
        if fifoFd != -1 {
                rt, err = runtime.StartInitialization(config, logLevel)
…
```

- 如果是init-runelet进程（只有init容器进程初始化的情况才会传exec fifo的fd），才执行enclave runtime的初始化
  - 首先加载enclave runtime。
  - 然后执行PAL API定义的enclave runtime的初始化函数pal_init()。

```
@libenclave/internal/runtime/enclave_runtime.go
func StartInitialization(config *configs.InitEnclaveConfig, logLevel string)
(*EnclaveRuntimeWrapper, error) {
        logrus.Debugf("enclave init config retrieved: %+v", config)

        var (
                runtime EnclaveRuntime
                err     error
        )
        runtime, err = core.StartInitialization(config)
        if err != nil {
                runtime, err = pal.StartInitialization(config)
…
        }

        logrus.Infof("Loading enclave runtime %s", config.Path)
        err = runtime.Load(config.Path)
…
        logrus.Infof("Initializing enclave runtime")
        err = runtime.Init(config.Args, logLevel)
…
        rt := &EnclaveRuntimeWrapper{
                runtime: runtime,
        }
        return rt, nil
}
```

未修改的runc代码
新实现的rune代码

执行远程证明

parent rune进程

```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
        ierr := parseSync(p.messageSockPair.parent, func(sync *syncT) error {
                switch sync.Type {
                case procReady:
…
                case procHooks:

                case procEnclaveConfigReq:
…
                case procEnclaveConfigAck:
…
                case procEnclaveInit:
…
                        err := writeSync(p.messageSockPair.parent,
procEnclaveReady)
…
                }

                return nil
        })
…
}
```

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32,
err error) {
…
        if err = writeSync(initPipe, procEnclaveConfigAck); err != nil {
…
        var rt *runtime.EnclaveRuntimeWrapper
        if fifoFd != -1 {
                rt, err = runtime.StartInitialization(config, logLevel)
                if err != nil {
                        return 1, err
                }
                if err = writeSync(initPipe, procEnclaveInit); err != nil {
                        return 1, err
                }

                // Launch a remote attestation to the enclave runtime.
                if err = rt.LaunchAttestation(); err != nil {
                        return 1, err
                }
                if err = readSync(initPipe, procEnclaveReady); err != nil {
                        return 1, err
                }
        }
…
```

init-runelet进程

- 在完成Enclave Runtime的初始化后，init-runelet进程要执行远程证明（目前的实现为空），以验证enclave环境确实是真实可信的。
- 在完成procEnclaveReady的同步后，parent rune进程会继续阻塞在parseSync中，直到init-runelet进程关闭init pipe为止。

未修改的runc代码
新实现的rune代码

获取PAL API版本

阿里云 | 奥运会全球指定云服务商

```
@libenclave/internal/runtime/pal/pal_linux.go
func (pal *enclaveRuntimePal) Load(palPath string) (err error) {
        if err = pal.getPalApiVersion(); err != nil {
                return err
        }
        return nil
}

func (pal *enclaveRuntimePal) getPalApiVersion() error {
        api := &enclaveRuntimePalApiV1{}
        ver := api.get_version()
        if ver > palApiVersion {
                return fmt.Errorf("unsupported pal api version %d", ver)
        }
        pal.version = ver
        return nil
}

const (
        palApiVersion = 2
)
```

未修改的runc代码
新实现的rune代码

- 调用enclave runtime PAL实现的PAL API pal_get_version()来确定enclave runtime PAL实现的PAL API的版本号。
- 如果实现的版本号高于libenclave实现的版本号（palApiVersion），则表示rune的版本太旧了。
- 高版本的rune能兼容实现了低版本PAL API的enclave runtime PAL。

执行enclave runtime初始化

阿里云 | 奥运会全球指定云服务商

```
@libenclave/internal/runtime/pal/pal_linux.go
func (pal *enclaveRuntimePal) Init(args string, logLevel string) error {
        api := &enclaveRuntimePalApiV1{}
        return api.init(args, logLevel)
}
```

```
@libenclave/internal/runtime/pal/api_linux_v1.go
func (api *enclaveRuntimePalApiV1) init(args string, logLevel string) error {
        logrus.Debugf("pal init() called with args %s", args)

        a := C.CString(args)
        defer C.free(unsafe.Pointer(a))

        l := C.CString(logLevel)
        defer C.free(unsafe.Pointer(l))

        sym := nsenter.SymAddrPalInit()
        ret := C.palInitV1(sym, a, l)
        if ret < 0 {
                return fmt.Errorf("pal init() failed with %d", ret)
        }
        return nil
}
```

```
static int palInitV1(void *sym, const char *args, const char *log_level)
{
        typedef struct {
                const char*     instance_dir;
                const char*     log_level;
        } pal_attr_t;
        pal_attr_t attr = {
                args,
                log_level,
        };

        return ((int (*)(pal_attr_t *))sym)(&attr);
}
```

未修改的runc代码
新实现的rune代码

- 通过cgo调用Enclave Runtime PAL API规范中定义的pal_init()接口，执行实际的enclave runtime的初始化操作。
- pal_init()接口的具体实现由Enclave Runtime负责，本slide不详细赘述。

关闭init pipe

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode
int32, err error) {
…
        // If runelet run as detach mode, close logrus before initpipe
closed.
        if cfg.Detached {
                logrus.SetOutput(ioutil.Discard)
        }

        // Close the init pipe to signal that we have completed our init.
        // So `rune create` or the upper half part of `rune run` can
return.
        initPipe.Close()
…
```

```
@libcontainer/process_linux.go
func (p *initProcess) start() error {
…
        ierr := parseSync(p.messageSockPair.parent, func(sync *syncT) error {
                switch sync.Type {
                case procReady:
…
                case procHooks:

                case procEnclaveConfigReq:
…
                case procEnclaveConfigAck:
…
                case procEnclaveInit:
…
        })
…
}
```

未修改的runc代码
新实现的rune代码

- 通过关闭init pipe的方式，唤醒parent rune进程不再阻塞在init pipe上。

建立信号转发机制

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32, err error) {
…
        notifySignal := make(chan os.Signal, signalBufferSize)
…
        notifyExit := make(chan struct{})
        sigForwarderExit := forwardSignal(rt, notifySignal, notifyExit)
…
```

```
func forwardSignal(rt *runtime.EnclaveRuntimeWrapper, notifySignal <-chan os.Signal, notifyExit <-chan struct{}) <-chan struct{} {
        isDead := make(chan struct{})
        go func() {
                defer close(isDead)
                for {
                        select {
                        case <-notifyExit:
                                return
                        case sig := <-notifySignal:
                                n := int(sig.(syscall.Signal))
                                err := rt.KillPayload(n, -1)
…
                                if sig == unix.SIGINT {
                                        os.Exit(0)
                                }
                        }
                }
        }()

        return isDead
}
```

未修改的runc代码
新实现的rune代码

- 启动一个goroutine执行信号转发，将捕获到的信号通过Enclave Runtime PAL API v2定义的pal_kill()接口发送给enclave runtime来处理。这么做目的是确保所有发给init-runelet进程的信号都被转发给目标enclave应用。这是为了满足信号的语义要求：发送给init-runelet的信号应当转发给真正的Enclave容器应用。

- 暂时保留在rune前端模式下通过在终端上输入ctrl-c传递SIGINT信号并快速退出整个容器的调试功能。

启动agent service


```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig)
(exitCode int32, err error) {
…
        agentPipe := cfg.AgentPipe
…
        agentExit := startAgentService(agentPipe, notifyExit)
…
```

- 启动一个goroutine运行Agent Service
  - 作用是监听容器实例的运行状态目录下的agent.sock，等待child runelet连接。
  - 具体细节见后面的rune exec流程。

```
@libenclave/agent.go
func startAgentService(agentPipe *os.File, notifyExit <-chan struct{}) <-chan struct{} {
        isDead := make(chan struct{})
        go func() {
…
                ln, err := net.FileListener(agentPipe)
…
                uln, ok := ln.(*net.UnixListener)
…
                for {
                        select {
                        case <-notifyExit:
…
                                return
                        default:
                        }

                        client, err := uln.Accept()
                        if err != nil {
…
                                return
                        }
                        instanceId += 1
                        go handleRequest(client, instanceId)
                }
        }()

        return isDead
}
```

未修改的runc代码
新实现的rune代码

通过exec fifo进行同步（1）

@libenclave/runelet.go

```
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32, err error) {
…
        if err = finalizeInitialization(fifoFd); err != nil {
…
```

init-runelet进程

```
func finalizeInitialization(fifoFd int) error {
        // Wait for the FIFO to be opened on the other side before exec-ing the
        // user process. We open it through /proc/self/fd/$fd, because the fd that
        // was given to us was an O_PATH fd to the fifo itself. Linux allows us to
        // re-open an O_PATH fd through /proc.
        fd, err := unix.Open(fmt.Sprintf("/proc/self/fd/%d", fifoFd), unix.O_WRONLY|
unix.O_CLOEXEC, 0)
…
```

@libcontainer/container_linux.go

```
func (c *linuxContainer) exec() error {
        path := filepath.Join(c.root, execFifoFilename)
        pid := c.initProcess.pid()
        blockingFifoOpenCh := awaitFifoOpen(path)
…
```

```
func awaitFifoOpen(path string) <-chan openResult {
        fifoOpened := make(chan openResult)
        go func() {
                result := fifoOpen(path, true)
                fifoOpened <- result
        }()
        return fifoOpened
}
```

parent rune进程

```
func fifoOpen(path string, block bool) openResult {
        flags := os.O_RDONLY
        if !block {
                flags |= unix.O_NONBLOCK
        }
        f, err := os.OpenFile(path, flags, 0)
…
        return openResult{file: f}
}
```

- parent rune进程通过“非阻塞读open exec fifo”的方式，唤醒“阻塞读写open exec fifo”的init-runelet进程。

未修改的runc代码
新实现的rune代码

通过exec fifo进行同步（2）

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32, err error) {
…
        if err = finalizeInitialization(fifoFd); err != nil {
…

func finalizeInitialization(fifoFd int) error {
        // Wait for the FIFO to be opened on the other side before exec-ing the
        // user process. We open it through /proc/self/fd/$fd, because the fd that
        // was given to us was an O_PATH fd to the fifo itself. Linux allows us to
        // re-open an O_PATH fd through /proc.
        fd, err := unix.Open(fmt.Sprintf("/proc/self/fd/%d", fifoFd), unix.O_WRONLY|
unix.O_CLOEXEC, 0)
…
        if _, err := unix.Write(fd, []byte("0")); err != nil {
…
        unix.Close(fifoFd)
        unix.Close(fd)
        return nil
}
```

init-runelet进程

```
@libcontainer/container_linux.go
func (c *linuxContainer) exec() error {
…
        for {
                select {
                case result := <-blockingFifoOpenCh:
                        return handleFifoResult(result)

                case <-time.After(time.Millisecond * 100):
                        stat, err := system.Stat(pid)
…
                }
        }
}

func handleFifoResult(result openResult) error {
…
        f := result.file
        defer f.Close()
        if err := readFromExecFifo(f); err != nil {
…
        return os.Remove(f.Name())
}

func readFromExecFifo(execFifo io.Reader) error {
        data, err := ioutil.ReadAll(execFifo)
…
        return nil
}
```

parent rune进程

- init-runelet进程通过写exec fifo的方式，唤醒阻塞在读exec fifo的parent rune进程。
- 至此，init-runelet进程与rune create/run之间所有的同步都已经完成；两个进程至此完全分道扬镳（例外是如果parent rune运行在前台模式的话，还是会继续等待init-runelet进程的退出）。

未修改的runc代码
新实现的rune代码

执行Enclave应用负载

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32, err error) {
…
        if err = finalizeInitialization(fifoFd); err != nil {
…
        // Capture all signals and then forward to enclave runtime.
        signal.Notify(notifySignal)
…
        enclaveRuntime = rt

        exitCode, err = rt.ExecutePayload(cmd, os.Environ(),
                [3]*os.File{
                        os.Stdin, os.Stdout, os.Stderr,
                })
        if err != nil {
                return exitCode, err
        }
        logrus.Debug("enclave runtime payload normally exits")
…
```

- 捕捉全部信号，以实现信号转发。
- 调用PAL API执行Enclave应用负载，即所谓的容器入口点程序。

未修改的runc代码
新实现的rune代码

执行退出路径

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32, err error) {
…
        logrus.Debug("enclave runtime payload normally exits")
…
        // The entrypoint payload exited, meaning current runelet process will die,
        // so friendly handle the exit path of enclave runtime instance.
        if err = rt.DestroyInstance(); err != nil {
                return exitCode, err
        }

        return exitCode, err
}
```

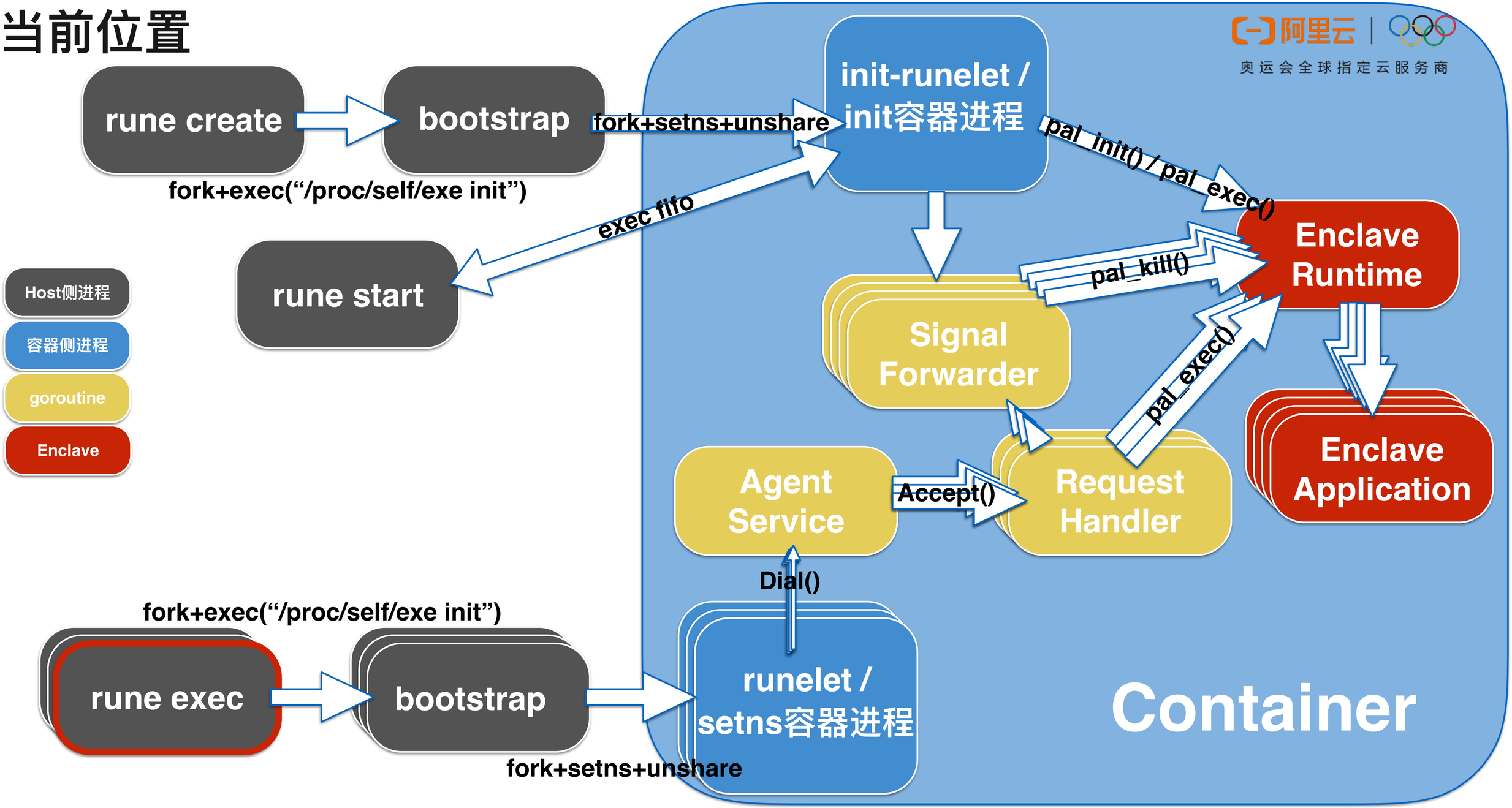
未修改的runc代码
新实现的rune代码

- 如果容器入口点程序正常退出，意味着整个容器生命周期的结束，Enclave Runtime本身也应当被销毁，并最终返回容器入口点程序的exit code。

Agenda

- 开源项目介绍
- 前置知识
- 系统架构
- bootstrap进程
- rune create/ run进程
- init-runelet进程
- 环境搭建
- init容器进程
- rune exec执行流程

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet / init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave Runtime
pal_kill()
Signal Forwarder
pal_exec()
Enclave Application
Agent Service
Accept()
Request Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet / setns容器进程
fork+setns+unshare
Container

libenclave包初始化


```
@libenclave/init.go
func IsEnclaveEnabled(e *configs.Enclave) bool {
    if e == nil {
        return false
    }

    if !IsEnclaveHwEnabled(e.Type) {
        return false
    }

    return true
}

// Check whether enclave-based hardware is supported or not
func IsEnclaveHwEnabled(etype string) bool {
    if etype == "" && enclaveHwType != "" {
        return true
    }

    return etype == enclaveHwType
}

var (
    enclaveHwType string = ""
)

func init() {
        _, _ = user.Lookup("")
        _, _ = net.LookupHost("")

        if intelsgx.IsSgxSupported() {
                enclaveHwType = configs.EnclaveHwIntelSgx
        }
}
```

未修改的runc代码
新实现的rune代码

- 检查当前系统是否支持enclave硬件特性。
- 目前libenclave包仅支持Intel SGX enclave。
- 在rune的设计中，runelet进程是非常特殊的。它是容器进程，但是它又不像runc处理容器进程那样令其execve()容器内的程序；runelet进程是一个进入了容器的namespace并“半容器”化的host进程。这个设计会带来一些兼容性问题：我们遇到过启动Ubuntu容器的时候，由于glibc动态依赖libnss导致在容器侧加载libnss.so的问题。这里的user.Lookup和net.LookipHost都是为了在rune进程还处于host侧的时候就加载libnss，从而避免在runelet进程（runelet进程是rune进程的子进程，因此rune进程加载的libnss会继承给runelet进程）进入容器后又因glibc间接通过dlopen加载libnss进而出现未定义符号的兼容性问题。
- 这个设计意味着任何Enclave Runtime PAL的实现都要遵守以下约束条件：如果Enclave Runtime PAL在运行时需要通过dlopen加载共享库，那么必须在rune进程还在host侧的时候预先加载好。

libenclave/intelsgx包初始化

阿里云 | 奥运会全球指定云服务商

@libenclave/intelsgx/device_linux.go

```
/*
#cgo linux LDFLAGS: -ldl
#include <stdlib.h>
#include <dlfcn.h>
*/
import "C"
…
func preloadSgxPswLib() {
        path := C.CString("libsgx_launch.so")
        C.dlopen(path, C.RTLD_NOW)
        C.free(unsafe.Pointer(path))
}

func init() {
        preloadSgxPswLib()
}
```

未修改的runc代码

新实现的rune代码

- 检查当前系统是否支持enclave硬件特性。
- 目前libenclave包仅支持Intel SGX enclave。
- 与事先加载libnss的workaround逻辑类似，这里的workaround确保了在host侧预先加载Intel SGX PSW的库libsgx_launch.so，规避在runelet进程已经进入了容器环境后调用PAL API时Enclave Runtime PAL调用dlopen需要从容器环境加载libsgx_launch.so的问题（比如容器环境是alpine的话，为alpine适配Intel SGX PSW库还是比较麻烦的）。

创建bootstrap进程(1)


```
@libcontainer/utils_linux.go
func (r *runner) run(config *specs.Process) (int, error) {

…
        var (
                detach = r.detach || (r.action == CT_ACT_CREATE)
        )

        if detach {
                process.Detached = true
        }
…
        switch r.action {
        case CT_ACT_CREATE:
                err = r.container.Start(process)
…
        case CT_ACT_RUN:
                err = r.container.Run(process)
…
}
```

```
@libcontainer/container_linux.go
func (c *linuxContainer) Run(process *Process) error {
        if err := c.Start(process); err != nil {
…
        return nil
}

func (c *linuxContainer) Start(process *Process) error {
…
        if process.Init {
                if err := c.createExecFifo(); err != nil {
…
        }
        if err := c.start(process); err != nil {
…
        }
        return nil
}

func (c *linuxContainer) start(process *Process) error {
   parent, err := c.newParentProcess(process)
…
}
```

未修改的runc代码
新实现的rune代码

- 记录当前runc是处于detached模式还是前台模式，后面的runelet日志处理逻辑需要知道这个信息。
- rune exec创建setns容器进程时，不需要像rune create/run那样创建额外的exec fifo文件。
- newParnetProcess()负责创建bootstrap进程的抽象对象。

创建bootstrap进程(2)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) newParentProcess(p *Process) (parentProcess, error) {
    parentInitPipe, childInitPipe, err := utils.NewSockPair("init")
…
    if c.config.Enclave != nil {
        if p.Init {
            uid, err := c.Config().HostRootUID()
…
            gid, err := c.Config().HostRootGID()
…
            p.AgentPipe, err = libenclave.CreateParentAgentPipe(c.root, uid, gid)
…
        } else {
            p.AgentPipe, err = libenclave.CreateChildAgentPipe(c.root)
…
        }
    }

    parentLogPipe, childLogPipe, err := os.Pipe()
…
    cmd, err := c.commandTemplate(p, childInitPipe, childLogPipe, p.AgentPipe, p.Detached)
…
}
```

未修改的runc代码
新实现的rune代码

- rune create/run和rune exec都需要创建一个子进程来执行容器的bootstrap工作
  - 如果是rune exec创建的bootstrap进程，则需要事先连接好Agent Service的服务端agent.sock，并返回客户端的socket fd。
  - 因为agent.sock文件位于host侧的容器实例的运行状态目录中，因此该文件不会被透传到容器侧，这就需要“事先”在还能访问agent.sock的时候打开它的socket fd，并在之后将socket fd透传给bootstrap进程和runelet进程。

创建bootstrap进程(3)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) commandTemplate(p *Process, childInitPipe *os.File, childLogPipe
*os.File, agentPipe *os.File, detached bool) *exec.Cmd {
        cmd := exec.Command(c.initPath, c.initArgs[1:]...)
        cmd.Args[0] = c.initArgs[0]
        cmd.Stdin = p.Stdin
        cmd.Stdout = p.Stdout
        cmd.Stderr = p.Stderr
        cmd.Dir = c.config.Rootfs
…
        cmd.ExtraFiles = append(cmd.ExtraFiles, childInitPipe)
        cmd.Env = append(cmd.Env,
                fmt.Sprintf("_LIBCONTAINER_INITPIPE=%d", stdioFdCount+len(cmd.ExtraFiles)-1),
                fmt.Sprintf("_LIBCONTAINER_STATEDIR=%s", c.root),
        )

        cmd.ExtraFiles = append(cmd.ExtraFiles, childLogPipe)
        cmd.Env = append(cmd.Env,
                fmt.Sprintf("_LIBCONTAINER_LOGPIPE=%d", stdioFdCount+len(cmd.ExtraFiles)-1),
                fmt.Sprintf("_LIBCONTAINER_LOGLEVEL=%s", p.LogLevel),
        )
…
```

未修改的runc代码
新实现的rune代码

- parent rune进程之后会通过fork+execve(“/proc/self/exe init”)的方式来创建bootstrap进程。
- 关键的fd都通过对应的_LIBCONTAINER_*环境变量传给bootstrap进程。
- 后面可以看到，parent rune向bootstrap和runelet进程传递信息的方式主要依靠init pipe。不过通过这种方式传递的信息大多是从父进程继承过来的，比如文件fd。fd本质上就是数字，而数字的含义需要“提点”。这些环境变量具有_LIBCONTAINER前缀，其目的也仅仅是为了“提点”文件fd的含义，不会真正的作用于容器进程。

创建bootstrap进程(4)

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) commandTemplate(p *Process, childInitPipe *os.File, childLogPipe
*os.File, agentPipe *os.File, detached bool) *exec.Cmd {
…
        cmd.Env = append(cmd.Env,
                fmt.Sprintf("_LIBCONTAINER_LOGPIPE=%d", stdioFdCount+len(cmd.ExtraFiles)-1),
                fmt.Sprintf("_LIBCONTAINER_LOGLEVEL=%s", p.LogLevel),
        )
…
        if c.config.Enclave != nil {
                if agentPipe != nil {
                        cmd.ExtraFiles = append(cmd.ExtraFiles, agentPipe)
                        cmd.Env = append(cmd.Env,
                                fmt.Sprintf("_LIBENCLAVE_AGENTPIPE=%d",
stdioFdCount+len(cmd.ExtraFiles)-1))
                }

                if c.config.Enclave.Path != "" {
                        cmd.Env = append(cmd.Env,
"_LIBENCLAVE_PAL_PATH="+c.config.Enclave.Path)
                }

                if detached {
                        cmd.Env = append(cmd.Env, "_LIBENCLAVE_DETACHED=1")
                }
        }
…
        return cmd
}
```

- Agent Service的socket fd会通过环境变量_LIBENCLAVE_AGENTFD传给bootstrap进程。
- Enclave Runtime PAL的文件路径通过_LIBENCLAVE_PAL_PATH传给bootstrap进程。
- rune的运行模式（detached或前台模式）通过_LIBENCLAVE_DETACHED传给bootstrap进程。

未修改的runc代码
新实现的rune代码

创建bootstrap进程(5)

```
@libcontainer/container_linux.go
func (c *linuxContainer) newParentProcess(p *Process) (parentProcess, error) {
…
    if !p.Init {
        return c.newSetnsProcess(p, cmd, messageSockPair, logFilePair)
    }
…
}
```

```
func (c *linuxContainer) newSetnsProcess(p *Process, cmd *exec.Cmd, messageSockPair,
logFilePair filePair) (*setnsProcess, error) {
        cmd.Env = append(cmd.Env, "_LIBCONTAINER_INITTYPE="+string(initSetns))
        state, err := c.currentState()
…
        data, err := c.bootstrapData(0, state.NamespacePaths)
…
        return &setnsProcess{
                cmd:             cmd,
…
                messageSockPair: messageSockPair,
                logFilePair:     logFilePair,
                config:          c.newInitConfig(p),
                process:         p,
                bootstrapData:   data,
        }, nil
}
```

未修改的runc代码

新实现的rune代码

- 如果是rune exec创建的bootstrap进程，不需要使用exec fifo文件
  - 将环境变量_LIBCONTAINER_INITTYPE设为setns，表示bootstrap进程之后要创建的是setns容器进程。
  - 构造好bootstrap数据和init config，之后发送给bootstrap进程。
- 如果是rune create/run创建的bootstrap进程，则需要事先创建好exec fifo文件，并将该文件fd透传给bootstrap进程。

创建bootstrap进程(6): _LIBCONTAINER/_LIBENCLAVE_*的含义

名称	来源	目标	作用
_LIBCONTAINER_INITTYPE	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	◦ standard：告诉bootstrap进程需要创建的是一个init容器进程 ◦ setns: 告诉bootstrap进程需要创建的是一个setns容器进程
_LIBCONTAINER_FIFOFD	rune create/run	init-runelet进程	在执行容器入口点程序前，用于同步rune create/run与init-runelet进程
_LIBCONTAINER_INITPIPE	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	容器的init pipe fd，在本阶段的作用有： ◦ parent rune进程向boostrap进程发送bootstrap数据。 ◦ parent rune进程读取到bootstrap进程和init容器进程的pid。 ◦ parent rune进程向容器进程发送init config。 ◦ 容器进程/runelet与parent rune进程间的各种状态同步。 ◦ parent rune进程向runelet进程发送的enclave init config。 ◦ 通过关闭该init pipe fd向parent rune进程告知runelet进程已经完成了全部的初始化操作。
_LIBCONTAINER_LOGPIPE	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	◦ 容器log pipe的fd ◦ 容器日志包含的是从bootstrap进程一直到runelet进程运行前这期间打印的日志(JSON格式)
_LIBCONTAINER_LOGLEVEL	rune create/run/exec	bootstrap进程 / 容器进程 / runelet进程	定义了容器日志的打印级别
_LIBENCLAVE_AGENTPIPE	rune create/run/exec	runelet进程	◦ init-runelet: agent.sock服务端的socket fd ◦ runelet：agent.sock客户端的socket fd
_LIBENCLAVE_PAL_PATH	rune create/run/exec	bootstrap进程	Enclave Runtime PAL的文件路径
_LIBENCLAVE_DETACHED	rune create/run/exec	runelet进程	rune的运行模式：前台或detached；该信息影响了ruenlet进程的容器日志打印行为

创建bootstrap进程(7)

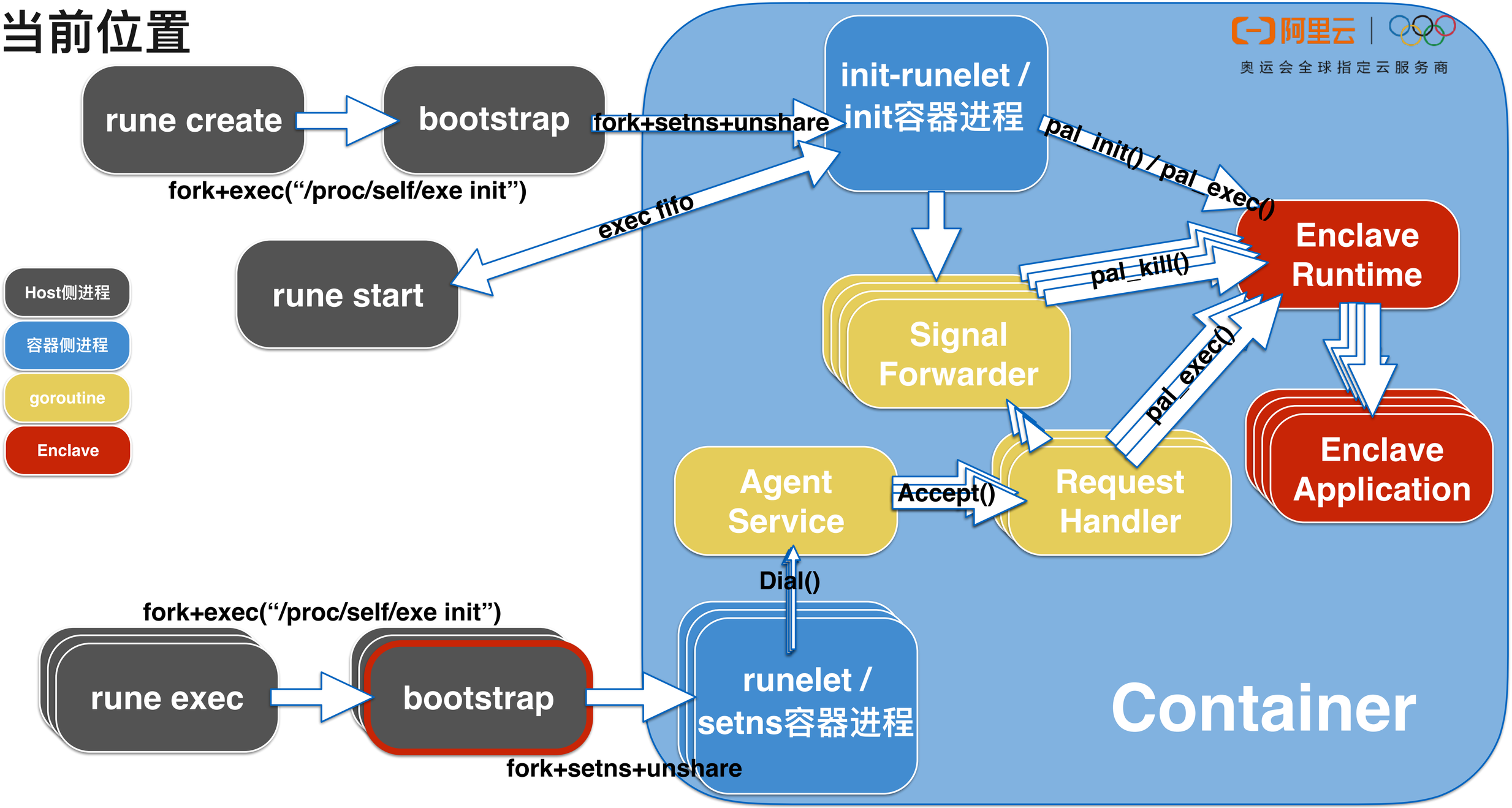
阿里云 | 奥运会全球指定云服务商

```
@libcontainer/container_linux.go
func (c *linuxContainer) start(process *Process) error {
…
    parent.forwardChildLogs()

    if err := parent.start(); err != nil {
…
    return nil
}

@libcontainer/process_linux.go
func (p *setnsProcess) start() error {
…
    err := p.cmd.Start()
…
}
```

未修改的runc代码
新实现的rune代码

- parent rune进程通过启动一个goroutine来转发bootstrap进程/容器进程/runelet进程通过child log pipe发送过来的容器日志（这里的容器日志的概念不同于K8s中的容器日志；本质上，K8s中的容器日志等价于容器进程的stdio，而这里的容器日志仅用于显示容器和runelet进程在初始化阶段的信息）。
- 至此，创建bootstrap进程的全部工作已经准备好，接下来正式通过fork+execve(“/proc/self/exe init”)启动bootstrap进程。

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet / init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave Runtime
pal_kill()
Signal Forwarder
pal_exec()
Enclave Application
Agent Service
Accept()
Request Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet / setns容器进程
fork+setns+unshare
Container

bootstrap进程与rune exec进程的交互

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/process_linux.go
func (p *setnsProcess) start() (err error) {
    defer p.messageSockPair.parent.Close()
    err = p.cmd.Start()
…
    if p.bootstrapData != nil {
        if _, err := io.Copy(p.messageSockPair.parent, p.bootstrapData); err != nil {
…
    }
    if err = p.execSetns(); err != nil {
…
```

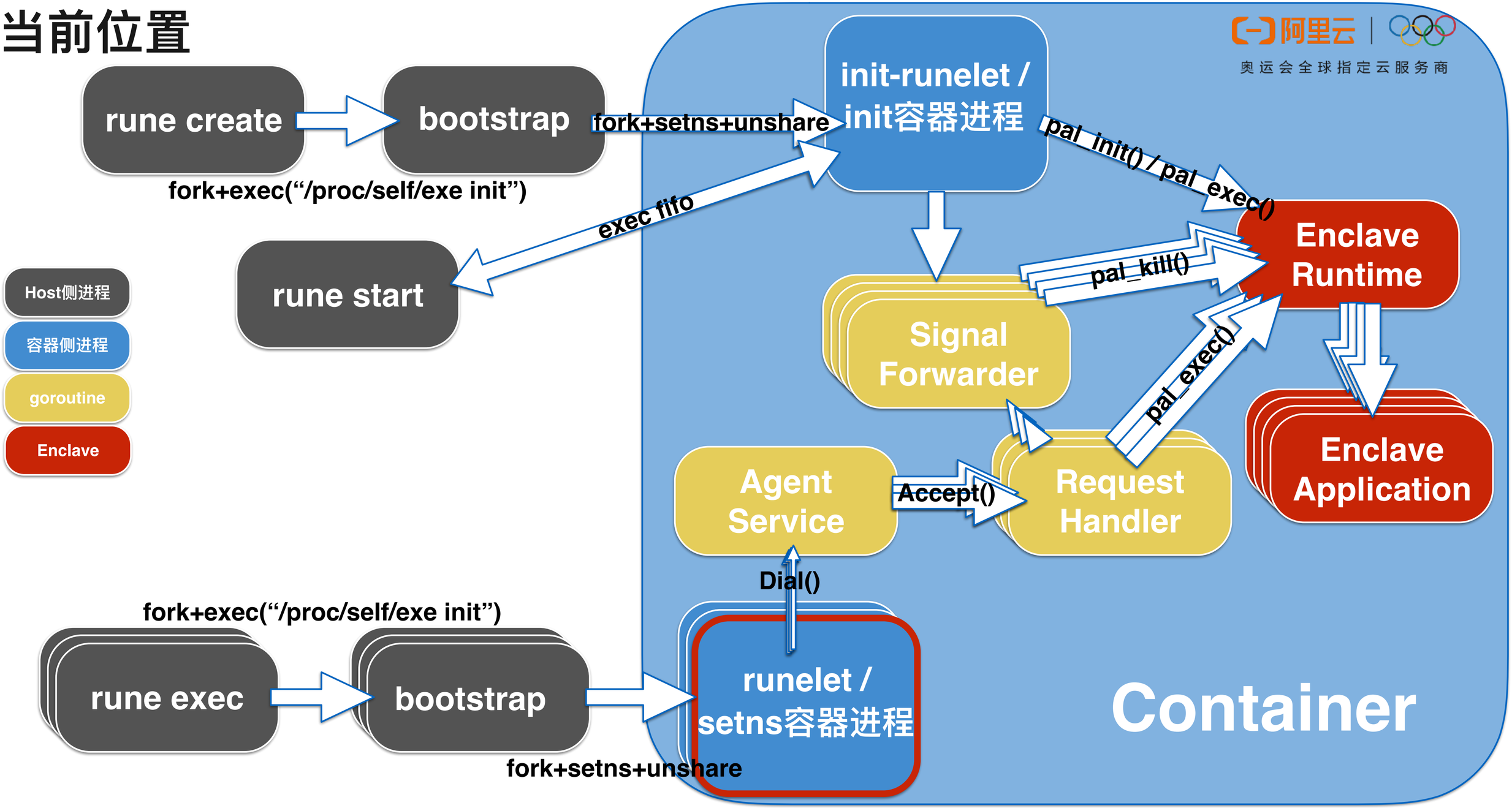
```
func (p *setnsProcess) execSetns() error {
        status, err := p.cmd.Process.Wait()
…
        var pid *pid
        if err := json.NewDecoder(p.messageSockPair.parent).Decode(&pid); err != nil {
…
        }

        // Clean up the zombie parent process
        // On Unix systems FindProcess always succeeds.
        firstChildProcess, _ := os.FindProcess(pid.PidFirstChild)

        // Ignore the error in case the child has already been reaped for any reason
        _, _ = firstChildProcess.Wait()

        process, err := os.FindProcess(pid.Pid)
…
        p.cmd.Process = process
        p.process.ops = p
        return nil
}
```

未修改的runc代码

新实现的rune代码

- bootstrap进程的详细过程详见前面的章节。
- p.cmd.Start()通过fork+exec(“/proc/self/exe init”)
- bootstrap进程阻塞在init pipe上，直到读取到从rune exec进程在这里发送过来的bootstrap数据。
- p.cmd.Process.Wait() 等待bootstrap进程的子进程退出。
- json.NewDecoder(p.messageSockPair.parent).Decode(&pid)解析bootstrap进程发送过来的bootstrap进程和孙进程（即未来的setns容器进程）的PID
- firstChildProcess.Wait()等待bootstrap进程退出。

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet / init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave Runtime
pal_kill()
Signal Forwarder
pal_exec()
Enclave Application
Agent Service
Accept()
Request Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet / setns容器进程
fork+setns+unshare
Container

执行包初始化

阿里云 | 奥运会全球指定云服务商

```
@init.go
func init() {
    if len(os.Args) > 1 && os.Args[1] == "init" {
        runtime.GOMAXPROCS(1)
        runtime.LockOSThread()

        level := os.Getenv("_LIBCONTAINER_LOGLEVEL")
        logLevel, err := logrus.ParseLevel(level)
…
        err = logs.ConfigureLogging(logs.Config{
            LogPipeFd: os.Getenv("_LIBCONTAINER_LOGPIPE"),
            LogFormat: "json",
            LogLevel:  logLevel,
        })
…
        logrus.Debug("child process in init()")
    }
}
```

未修改的runc代码
新实现的rune代码

- 在离开C语言运行环境后，容器进程执行Golang Runtime，然后执行这里的包初始化代码。
- 设置容器日志相关的配置

执行rune init命令

阿里云 | 奥运会全球指定云服务商

```
@init.go
var initCommand = cli.Command{
   Name:  "init",
   Usage: `initialize the namespaces and launch the process (do not call it outside of runc)`,
   Action: func(context *cli.Context) error {
      factory, _ := libcontainer.New("")
      if err := factory.StartInitialization(); err != nil {
…
         os.Exit(1)
      }
      panic("libcontainer: container init failed to exec")
   },
}
```

未修改的runc代码
新实现的rune代码

- 对于容器来说，不管是init容器进程还是setns容器进程，最终都是通过执行execve()来运行一个容器应用程序，因此正常情况下不应该从factory.StartInitialization()返回。一旦返回，如果不是在执行execve()的中途遇到了错误，则会导致程序panic()。

- 对于rune来说，绝对不会通过execve()来运行一个容器应用程序，但也要遵守对上述行为的约束和规定，即正常情况下不应该返回，否则会导致程序panic()。

重建关键信息

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/factory_linux.go
func (l *LinuxFactory) StartInitialization() (err error) {
…
    detached        = false
    envInitPipe     = os.Getenv("_LIBCONTAINER_INITPIPE")
    envFifoFd       = os.Getenv("_LIBCONTAINER_FIFOFD")
…
    envLogPipe      = os.Getenv("_LIBCONTAINER_LOGPIPE")
    envLogLevel     = os.Getenv("_LIBCONTAINER_LOGLEVEL")
    envAgentPipe    = os.Getenv("_LIBENCLAVE_AGENTPIPE")
    envDetached     = os.Getenv("_LIBENCLAVE_DETACHED")
…
  pipefd, err = strconv.Atoi(envInitPipe)
…
  var (
    pipe = os.NewFile(uintptr(pipefd), "pipe")
    it   = initType(os.Getenv("_LIBCONTAINER_INITTYPE"))
  )
…
  fifofd = -1
  if it == initStandard {
    if fifofd, err = strconv.Atoi(envFifoFd); err != nil {
…
```

```
  if envLogPipe != "" {
    log, err := strconv.Atoi(envLogPipe)
…
    logPipe = os.NewFile(uintptr(log), "log-pipe")
…
  }

  if envAgentPipe != "" {
    agent, err := strconv.Atoi(envAgentPipe)
…
    agentPipe = os.NewFile(uintptr(agent), "agent-pipe")
…
  }

  if envDetached != "" {
    tmpDetached, err := strconv.Atoi(envDetached)
…
    if tmpDetached != 0 {
      detached = true
    }
  }
…
```

未修改的runc代码
新实现的rune代码

- 读取_LIBCONTAINER_*和_LIBENCLAVE_*环境变量并重建关键信息。

清除从父进程继承过来的环境变量

阿里云 | 奥运会全球指定云服务商

```
@libcontainer/factory_linux.go
func (l *LinuxFactory) StartInitialization() (err error) {
…
    os.Clearenv()
…
```

未修改的runc代码
新实现的rune代码

- 清除容器进程继承自bootstrap进程/parent rune进程的全部环境变量，避免来自host侧的环境变量对容器进程的行为产生影响。

接收init config


```
@libcontainer/factory_linux.go
func (l *LinuxFactory) StartInitialization() (err error) {
    i, err := newContainerInit(it, pipe, consoleSocket, fifofd, logPipe, envLogLevel, agentPipe, detached)
…
```

```
@libcontainer/init_linux.go
func newContainerInit(t initType, pipe *os.File, consoleSocket *os.File, fifoFd int, logPipe *os.File, logLevel string, agentPipe *os.File, detached bool)
(initer, error) {
    var config *initConfig
    if err := json.NewDecoder(pipe).Decode(&config); err != nil {
…
    if err := populateProcessEnvironment(config.Env); err != nil {
…
    switch t {
    case initSetns:
        return &linuxSetnsInit{
            pipe:          pipe,
…
            logPipe:       logPipe,
            logLevel:      logLevel,
            agentPipe:     agentPipe,
            detached:      detached,
        }, nil
    case initStandard:
        return &linuxStandardInit{
            pipe:          pipe,
…
            fifoFd:        fifoFd,
            logPipe:       logPipe,
            logLevel:      logLevel,
            agentPipe:     agentPipe,
            detached:      detached,
        }, nil
    }
…
}
```

setns容器进程

```
@libcontainer/process_linux.go
func (p *setnsProcess) start() (err error) {
…
        if err = p.execSetns(); err != nil {
…
        if err := utils.WriteJSON(p.messageSockPair.parent, p.config); err != nil {
…
```

rune exec进程

未修改的runc代码
新实现的rune代码

- 通过init pipe从parent rune侧接收JSON格式的init config。
- 将init config中有关容器环境变量的配置设置到当前容器进程中。
- 将enclave相关的状态信息记录在相关的上下文中，留待后续处理。

调用libenclave初始化代码

阿里云 | 奥运会全球指定云服务商

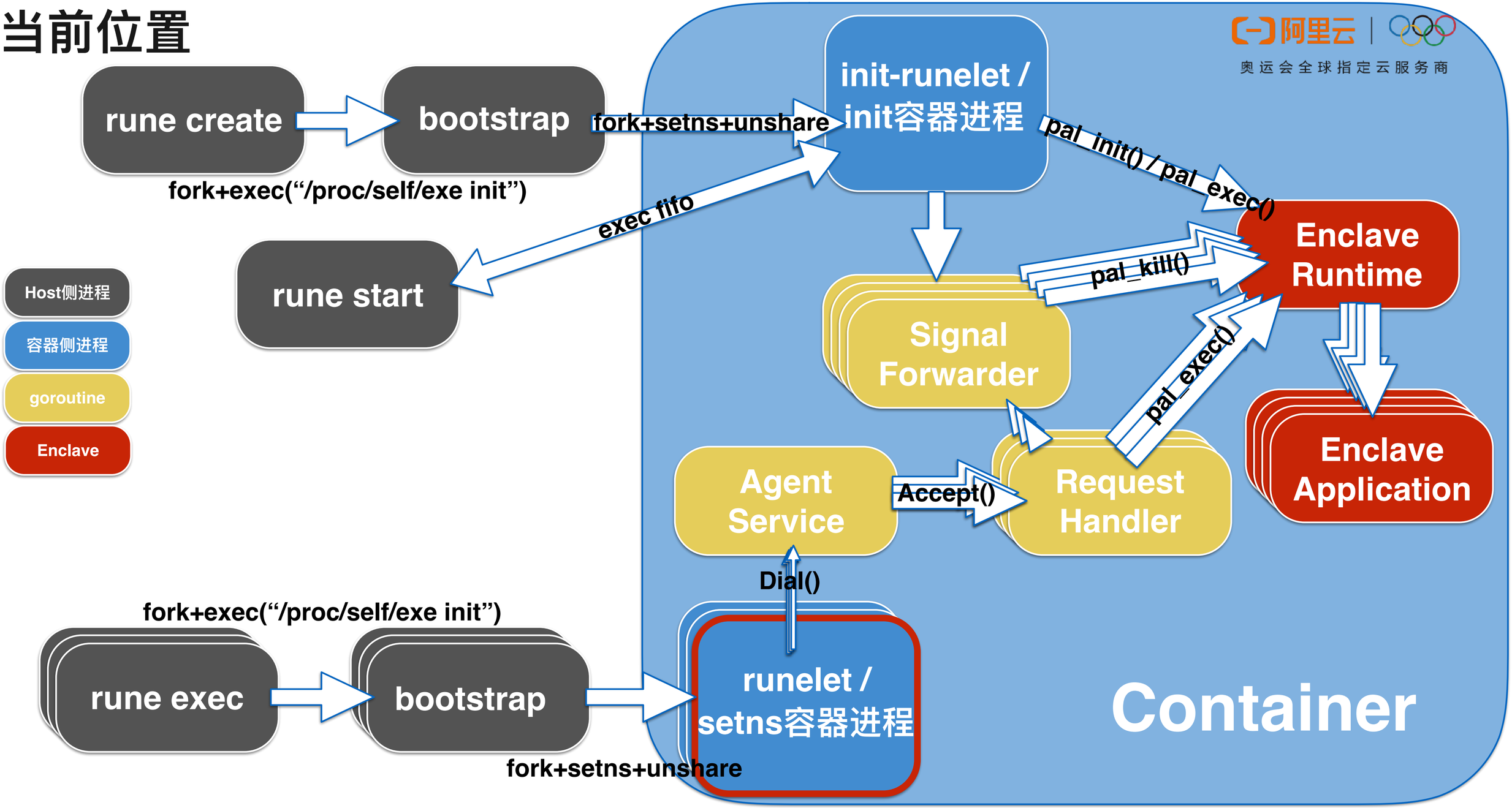
```
func (l *linuxSetnsInit) Init() error {
        runtime.LockOSThread()
…
        if l.config.Config.Enclave != nil {
                cfg := &libenclave.RuneletConfig{
                        InitPipe:  l.pipe,
                        LogPipe:   l.logPipe,
                        LogLevel:  l.logLevel,
                        FifoFd:    -1,
                        AgentPipe: l.agentPipe,
                        Detached:  l.detached,
                }

                exitCode, err :=
libenclave.StartInitialization(l.config.Args, cfg)
                if err != nil {
                        return err
                }
                logrus.Debugf("enclave payload exit code: %d", exitCode)
                os.Exit(int(exitCode))
                // make compiler happy
                return fmt.Errorf("failed to initialize runelet")
        }
        return system.Execv(l.config.Args[0], l.config.Args[0:],
os.Environ())
}
```

未修改的runc代码
新实现的rune代码

- 此时的setns容器进程自身仍旧不是完全容器化的。在运行runc容器的情况，setns容器进程还需要通过execve()执行容器入口点才算是完成自身的容器化；而rune的情况则是维持在这个“半容器化”的状态，并开始执libenclave.StartInitialization()，后续的执行流程都将在libenclave中进行。

- 如果Enclave Runtime正常情况退出的话，会通过os.Exit()退出并返回exit code。如果不这样处理，返回到init.go后会触发panic()。

当前位置


阿里云
奥运会全球指定云服务商
rune create
bootstrap
fork+setns+unshare
init-runelet / init容器进程
pal_init() / pal_exec()
fork+exec("/proc/self/exe init")
exec fifo
rune start
Host侧进程
容器侧进程
goroutine
Enclave
Enclave Runtime
pal_kill()
Signal Forwarder
pal_exec()
Enclave Application
Agent Service
Accept()
Request Handler
Dial()
fork+exec("/proc/self/exe init")
rune exec
bootstrap
runelet / setns容器进程
fork+setns+unshare
Container

传递enclave配置信息

阿里云 | 奥运会全球指定云服务商

@libcontainer/process_linux.go

```
func (p *setnsProcess) start() (err error) {
…
        ierr := parseSync(p.messageSockPair.parent, func(sync *syncT) error {
                switch sync.Type {
…
                case procEnclaveConfigReq:
…
                        config := &enclave_configs.InitEnclaveConfig{
                                Type: p.config.Config.Enclave.Type,
                                Path: p.config.Config.Enclave.Path,
                                Args: p.config.Config.Enclave.Args,
                        }
                        err := utils.WriteJSON(p.messageSockPair.parent, config)
…
                        recvEnclaveConfig = true
                case procEnclaveConfigAck:
…
                        sentEnclaveConfigAck = true
…
                }

                return nil
        })
…
```

rune exec进程

@libenclave/runelet.go

```
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32,
err error) {
        logLevel := cfg.LogLevel

        // Determine which type of runelet is initializing.
        fifoFd := cfg.FifoFd

        // Retrieve the init pipe fd to accomplish the enclave configuration
        // handshake as soon as possible with parent rune.
        initPipe := cfg.InitPipe
…
        if err = writeSync(initPipe, procEnclaveConfigReq); err != nil {
…
        var config *configs.InitEnclaveConfig
        if err = json.NewDecoder(initPipe).Decode(&config); err != nil {
…
        if err = writeSync(initPipe, procEnclaveConfigAck); err != nil {
…
```

runelet进程

未修改的runc代码
新实现的rune代码

- 通过init pipe从parent rune进程处获取到JSON格式的enclave配置信息。示例：

```
"enclave": {
        "type": "intelSgx",
        "path": "/usr/lib/libocclum-pal.so",
        "args": ".occlum"
}
```

- 在完成procEnclaveAcl的同步后，parent rune进程会继续阻塞在parseSync中，直到runelet进程关闭init pipe为止。

关闭init pipe

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32,
err error) {
…
        if err = writeSync(initPipe, procEnclaveConfigAck); err != nil {
…
        var rt *runtime.EnclaveRuntimeWrapper
        if fifoFd != -1 {
…
        }

        if cfg.Detached {
                logrus.SetOutput(ioutil.Discard)
        }

        initPipe.Close()
…
```

runelet进程

```
@libcontainer/process_linux.go
func (p *setnsProcess) start() (err error) {
…
        ierr := parseSync(p.messageSockPair.parent, func(sync *syncT) error {
                switch sync.Type {
…
                case procEnclaveConfigReq:
…
                case procEnclaveConfigAck:
…
                }

                return nil
        })
…
```

rune exec进程

未修改的runc代码
新实现的rune代码

- 经过前面的同步过程后，rune exec进程会与runelet进程并行执行；如果rune运行在detached模式的话，run exec进程会很快退出，届时runelet进程在输出容器日志时会触发broken pipe的错误。
- 为了避免该错误的发生，就需要runelet进程主动停止产生容器日志。
- 通过关闭init pipe的方式，唤醒rune exec进程不再阻塞在init pipe上。至此，rune exec进程和runelet进程分道扬镳，开始并行执行。

准备执行remote execution

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func StartInitialization(cmd []string, cfg *RuneletConfig) (exitCode int32, err error) {
…
        agentPipe := cfg.AgentPipe
        defer agentPipe.Close()

        notifySignal := make(chan os.Signal, signalBufferSize)

        if fifoFd == -1 {
                exitCode, err = remoteExec(agentPipe, cmd, notifySignal)
                if err != nil {
                        return exitCode, err
                }
                logrus.Debug("remote exec normally exits")

                return exitCode, err
        }
…
```

未修改的runc代码
新实现的rune代码

- 如果rune exec执行的enclave容器应用正常退出，意味着当前runelet进程生命周期的结束，并最终返回enclave容器应用的exit code。

执行remote execution

阿里云 | 奥运会全球指定云服务商

@libenclave/runelet.go

```
func remoteExec(agentPipe *os.File, cmd []string, notifySignal chan os.Signal) (exitCode
int32, err error) {
        c := strings.Join(cmd, " ")
        logrus.Debugf("preparing to remote exec %s", c)

        req := &pb.AgentServiceRequest{}
        req.Exec = &pb.AgentServiceRequest_Execute{
                Argv: c,
                Envp: strings.Join(os.Environ(), " "),
        }
        if err = protoBufWrite(agentPipe, req); err != nil {
…
        // Send signal notification pipe.
        childSignalPipe, parentSignalPipe, err := os.Pipe()
…
        if err = utils.SendFd(agentPipe, childSignalPipe.Name(), childSignalPipe.Fd());
err != nil {
…
        // Send stdio fds.
        if err = utils.SendFd(agentPipe, os.Stdin.Name(), os.Stdin.Fd()); err != nil {
…
        if err = utils.SendFd(agentPipe, os.Stdout.Name(), os.Stdout.Fd()); err != nil {
…
        if err = utils.SendFd(agentPipe, os.Stderr.Name(), os.Stderr.Fd()); err != nil {
…
        // Close the child signal pipe in parent side **after** sending all stdio fds to
        // make sure the parent runelet has retrieved the child signal pipe.
        childSignalPipe.Close()
…
```

未修改的runc代码
新实现的rune代码

- 按照Agent Service协议的约定，连接Agent Service服务端，并依次发送：
  - 信号通知pipe的fd
  - stdio的3个fd

init-runelet进程接收并处理Agent Service请求

阿里云 | 奥运会全球指定云服务商

@libenclave/agent.go

```
func handleRequest(conn net.Conn, id int) {
...
        resp := &pb.AgentServiceResponse{}
        resp.Exec = &pb.AgentServiceResponse_Execute{}
        exitCode := int32(1)
...
        req := &pb.AgentServiceRequest{}
        if err = protoBufRead(conn, req); err != nil {
...
        c, ok := conn.(*net.UnixConn)
...
        connFile, err := c.File()
...
        // Retrieve signal pipe.
        signalPipe, err := utils.RecvFd(connFile)
...
        go relaySignal(signalPipe, id)

        // Retrieve stdio fds.
        stdin, err := utils.RecvFd(connFile)
...
        stdout, err := utils.RecvFd(connFile)
...
        stderr, err := utils.RecvFd(connFile)
...
        stdio := [3]*os.File{
                stdin, stdout, stderr,
        }

        cmd := req.Exec.GetArgv()
        envp := req.Exec.GetEnvp()
        exitCode, err = enclaveRuntime.ExecutePayload(strings.Split(cmd, " "),
strings.Split(envp, " "), stdio)
...
        logrus.Debug("remote exec normally exits")
}
```

未修改的runc代码
新实现的rune代码

- init-runelet进程每收到一个Agent Service请求，就创建一个新的goroutine来处理该请求，并按照Agent Service协议的定义，顺序接收：
  - 信号fd
  - stdio的3个fd
- 调用PAL API执行Enclave应用负载，即rune exec指定的容器应用程序。

执行信号转发

阿里云 | 奥运会全球指定云服务商

@libenclave/runelet.go

```
func remoteExec(agentPipe *os.File, cmd []string, notifySignal chan os.Signal)
(exitCode int32, err error) {
…
        signal.Notify(notifySignal)

        notifyExit := make(chan struct{})
        sigForwarderExit := forwardSignalToParent(parentSignalPipe,
notifySignal, notifyExit)
…
```

- 向信号pipe fd写入信号转发命令，由init-runelet进程侧的goroutine负责执行信号处理。这么做目的是确保所有发给runelet进程的信号都被转发给目标enclave应用。这是为了满足信号的语义要求：发送给runelet的信号应当转发给真正的Enclave容器应用。

- 暂时保留在rune前端模式下通过在终端上输入ctrl-c传递SIGINT信号并快速退出整个容器的调试功能。

```
func forwardSignalToParent(conn io.Writer, notifySignal chan os.Signal,
notifyExit <-chan struct{}) <-chan struct{} {
        isDead := make(chan struct{})
        go func() {
                defer close(isDead)

                for {
                        select {
                        case <-notifyExit:
…
                                return
                        case sig := <-notifySignal:
                                n := int32(sig.(syscall.Signal))
                                req := &pb.AgentServiceRequest{}
                                req.Kill = &pb.AgentServiceRequest_Kill{
                                        Sig: n,
                                }
                                if err := protoBufWrite(conn, req); err !=
nil {
…

                                if sig == unix.SIGINT {
                                        os.Exit(0)
                                }
                        }
                }
        }()

        return isDead
}
```

未修改的runc代码

新实现的rune代码

执行信号转发

阿里云 | 奥运会全球指定云服务商

```
@libenclave/runelet.go
func relaySignal(signalPipe *os.File, id int) {
        defer signalPipe.Close()

        for {
                req := &pb.AgentServiceRequest{}
                if err := protoBufRead(signalPipe, req); err != nil {
                        return
                }

                err := enclaveRuntime.KillPayload(id, int(req.Kill.Sig))
                if err != nil {
                        logrus.Errorf("unable to kill payload with sig %d by
%d: %v\n", int(req.Kill.Sig), id, err)
                        return
                }
        }
}
```

未修改的runc代码
新实现的rune代码

- init-runelet进程侧的goroutine将从runelet进程转发过来的信号通过PAL API pal_kill()发送给enclave runtime进行处理。

处理退出路径

阿里云 | 奥运会全球指定云服务商

@libenclave/runelet.go

```
func remoteExec(agentPipe *os.File, cmd []string, notifySignal chan os.Signal) (exitCode
int32, err error) {
…
        sigForwarderExit := forwardSignalToParent(parentSignalPipe, notifySignal,
notifyExit)

        resp := &pb.AgentServiceResponse{}
        if err = protoBufRead(agentPipe, resp); err != nil {
…
        notifyExit <- struct{}{}
        logrus.Debug("awaiting for signal forwarder exiting ...")
        <-sigForwarderExit
        logrus.Debug("signal forwarder exited")

        if resp.Exec.Error == "" {
                err = nil
        } else {
                err = fmt.Errorf(resp.Exec.Error)
        }
        return resp.Exec.ExitCode, err
}
```

未修改的runc代码

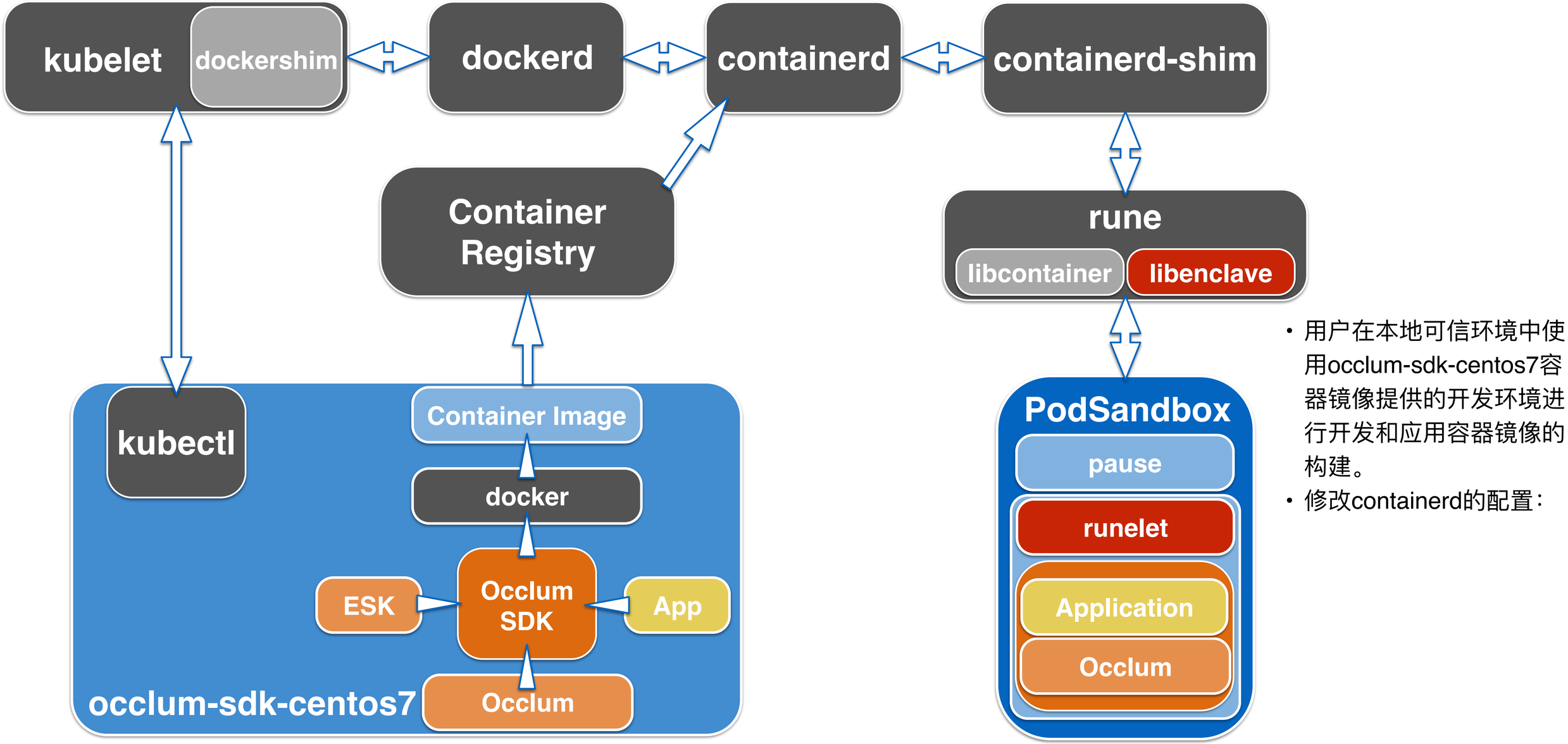
新实现的rune代码

- 如果rune exec执行的enclave容器应用正常退出，意味着当前runelet进程生命周期的结束，并最终返回enclave容器应用的exit code。

奥运会全球指定云服务商

WWW.ALIYUN.COM

k8s+rune的开源方案架构



- 用户在本地可信环境中使用occlum-sdk-centos7容器镜像提供的开发环境进行开发和应用容器镜像的构建。
- 修改containerd的配置：

通过rune create+start运行Enclave应用的过程