libcontainer

GO reference

Libcontainer provides a native Go implementation for creating containers with namespaces, cgroups, capabilities, and filesystem access controls. It allows you to manage the lifecycle of the container performing additional operations after the container is created.

Container

A container is a self contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system.

Using libcontainer

Because containers are spawned in a two step process you will need a binary that will be executed as the init process for the container. In libcontainer, we use the current binary (/proc/self/exe) to be executed as the init process, and use arg "init", we call the first step process "bootstrap", so you always need a "init" function as the entry of "bootstrap".

In addition to the go init function the early stage bootstrap is handled by importing nsenter.

```
import (
    _ "github.com/opencontainers/runc/libcontainer/nsenter"
)

func init() {
    if len(os.Args) > 1 && os.Args[1] == "init" {
        runtime.GOMAXPROCS(1)
        runtime.LockOSThread()
        factory, _ := libcontainer.New("")
        if err := factory.StartInitialization(); err != nil {
            logrus.Fatal(err)
        }
        panic("--this line should have never been executed, congratulations--")
    }
}
```

Then to create a container you first have to initialize an instance of a factory that will handle the creation and initialization for a container.

```
factory, err := libcontainer.New("/var/lib/container", libcontainer.Cgroupfs,
libcontainer.InitArgs(os.Args[0], "init"))
if err != nil {
    logrus.Fatal(err)
    return
}
```

Once you have an instance of the factory created we can create a configuration struct describing how the container is to be created. A sample would look similar to this:

```go
defaultMountFlags := unix.MS_NOEXEC | unix.MS_NOSUID | unix.MS_NODEV
var devices []*configs.DeviceRule
for _, device := range specconv.AllowedDevices {
    devices = append(devices, &device.Rule)
}
config := &configs.Config{
    Rootfs: "/your/path/to/rootfs",
    Capabilities: &configs.Capabilities{
        Bounding: []string{
            "CAP_CHOWN",
            "CAP_DAC_OVERRIDE",
            "CAP_FSETID",
            "CAP_FOWNER",
            "CAP_MKNOD",
            "CAP_NET_RAW",
            "CAP_SETGID",
            "CAP_SETUID",
            "CAP_SETFCAP",
            "CAP_SETPCAP",
            "CAP_NET_BIND_SERVICE",
            "CAP_SYS_CHROOT",
            "CAP_KILL",
            "CAP_AUDIT_WRITE",
        },
        Effective: []string{
            "CAP_CHOWN",
            "CAP_DAC_OVERRIDE",
            "CAP_FSETID",
            "CAP_FOWNER",
            "CAP_MKNOD",
            "CAP_NET_RAW",
            "CAP_SETGID",
            "CAP_SETUID",
            "CAP_SETFCAP",
            "CAP_SETPCAP",
            "CAP_NET_BIND_SERVICE",
            "CAP_SYS_CHROOT",
            "CAP_KILL",
            "CAP_AUDIT_WRITE",
        },
        Inheritable: []string{
            "CAP_CHOWN",
            "CAP_DAC_OVERRIDE",
            "CAP_FSETID",
            "CAP_FOWNER",
            "CAP_MKNOD",
            "CAP_NET_RAW",
            "CAP_SETGID",
            "CAP_SETUID",
            "CAP_SETFCAP",
            "CAP_SETPCAP",
            "CAP_NET_BIND_SERVICE",
            "CAP_SYS_CHROOT",
            "CAP_KILL",
            "CAP_AUDIT_WRITE",
        },
        Permitted: []string{
            "CAP_CHOWN",
            "CAP_DAC_OVERRIDE",
            "CAP_FSETID",
            "CAP_FOWNER",
            "CAP_MKNOD",
            "CAP_NET_RAW",
            "CAP_SETGID",
            "CAP_SETUID",
            "CAP_SETFCAP",
            "CAP_SETPCAP",
            "CAP_NET_BIND_SERVICE",
            "CAP_SYS_CHROOT",
            "CAP_KILL",
            "CAP_AUDIT_WRITE",
        },
        Ambient: []string{
            "CAP_CHOWN",
            "CAP_DAC_OVERRIDE",
            "CAP_FSETID",
            "CAP_FOWNER",
            "CAP_MKNOD",
            "CAP_NET_RAW",
            "CAP_SETGID",
            "CAP_SETUID",
            "CAP_SETFCAP",
            "CAP_SETPCAP",
            "CAP_NET_BIND_SERVICE",
            "CAP_SYS_CHROOT",
            "CAP_KILL",
            "CAP_AUDIT_WRITE",
        },
    },
    Namespaces: configs.Namespaces([]configs.Namespace{
        {Type: configs.NEWNS},
        {Type: configs.NEWUTS},
        {Type: configs.NEWIPC},
        {Type: configs.NEWPID},
        {Type: configs.NEWUSER},
        {Type: configs.NEWNET},
        {Type: configs.NEWCGROUP},
    }),
    Cgroups: &configs.Cgroup{
        Name:   "test-container",
        Parent: "system",
        Resources: &configs.Resources{
            MemorySwappiness: nil,
            Devices:          devices,
        },
    },
    MaskPaths: []string{
        "/proc/kcore",
        "/sys/firmware",
    },
    ReadonlyPaths: []string{
        "/proc/sys", "/proc/sysrq-trigger", "/proc/irq", "/proc/bus",
    },
    Devices:  specconv.AllowedDevices,
    Hostname: "testing",
    Mounts: []*configs.Mount{
        {
            Source:      "proc",
            Destination: "/proc",
            Device:      "proc",
            Flags:       defaultMountFlags,
        },
        {
            Source:      "tmpfs",
            Destination: "/dev",
            Device:      "tmpfs",
            Flags:       unix.MS_NOSUID | unix.MS_STRICTATIME,
            Data:        "mode=755",
        },
        {
            Source:      "devpts",
            Destination: "/dev/pts",
            Device:      "devpts",
            Flags:       unix.MS_NOSUID | unix.MS_NOEXEC,
            Data:        "newinstance,ptmxmode=0666,mode=0620,gid=5",
        },
        {
            Device:      "tmpfs",
            Source:      "shm",
            Destination: "/dev/shm",
            Data:        "mode=1777,size=65536k",
            Flags:       defaultMountFlags,
        },
        {
            Source:      "mqueue",
            Destination: "/dev/mqueue",
            Device:      "mqueue",
            Flags:       defaultMountFlags,
        },
        {
            Source:      "sysfs",
            Destination: "/sys",
            Device:      "sysfs",
            Flags:       defaultMountFlags | unix.MS_RDONLY,
        },
	},
	UidMappings: []configs.IDMap{
		{
			ContainerID: 0,
			HostID: 1000,
			Size: 65536,
		},
	},
	GidMappings: []configs.IDMap{
		{
			ContainerID: 0,
			HostID: 1000,
			Size: 65536,
		},
	},
	Networks: []*configs.Network{
		{
			Type:    "loopback",
			Address: "127.0.0.1/0",
			Gateway: "localhost",
		},
	},
	Rlimits: []configs.Rlimit{
		{
			Type: unix.RLIMIT_NOFILE,
			Hard: uint64(1025),
			Soft: uint64(1025),
		},
	},
}
```

Once you have the configuration populated you can create a container:

```
container, err := factory.Create("container-id", config)
if err != nil {
	logrus.Fatal(err)
	return
}
```

To spawn bash as the initial process inside the container and have the processes pid returned in order to wait, signal, or kill the process:

```
process := &libcontainer.Process{
	Args:   []string{"/bin/bash"},
	Env:    []string{"PATH=/bin"},
	User:   "daemon",
	Stdin:  os.Stdin,
	Stdout: os.Stdout,
	Stderr: os.Stderr,
	Init:   true,
}
```

```
err := container.Run(process)
if err != nil {
    container.Destroy()
    logrus.Fatal(err)
    return
}

// wait for the process to finish.
_, err := process.Wait()
if err != nil {
    logrus.Fatal(err)
}

// destroy the container.
container.Destroy()
```

Additional ways to interact with a running container are:

```
// return all the pids for all processes running inside the container.
processes, err := container.Processes()

// get detailed cpu, memory, io, and network statistics for the container and
// it's processes.
stats, err := container.Stats()

// pause all processes inside the container.
container.Pause()

// resume all paused processes.
container.Resume()

// send signal to container's init process.
container.Signal(signal)

// update container resource constraints.
container.Set(config)

// get current status of the container.
status, err := container.Status()

// get current container's state information.
state, err := container.State()
```

Checkpoint & Restore

libcontainer now integrates CRIU for checkpointing and restoring containers. This lets you save the state of a process running inside a container to disk, and then restore that state into a new process, on the same machine or on another machine.

`criu` version 1.5.2 or higher is required to use checkpoint and restore. If you don't already have `criu` installed, you can build it from source, following the online instructions. `criu` is also installed in the docker image generated when building libcontainer with docker.

Copyright and license