# docker编译

docker pull dockercore/docker

代码切换到要编译的分支:

git checkout v17.05.0-ce

docker run –it --privileged --name docker-dev -v $(pwd):/go/src/github.com/docker/docker dockercore/docker ./hack/make.sh binary

**注：**

docker使用了大量的其他项目源码包，为了保证使用最新的源码包编译，可以将docker/vender/src目录挂载到容器的/go/src目录下

docker run –it --privileged --name docker-dev -v $(pwd):/go/src/github.com/docker/docker -v $(pwd)/vender/src:/go/src dockercore/docker ./hack/make.sh binary

*编译动态链接的可执行文件*

*当使用devicemapper作为storage-driver时。为了使能Udev sync,必须用动态链接的可执行文件，而这只能使用手动启动容器的方式。*

*条件编译*

用于编写多操作系统环境下的代码。

文件头的注释，以+build开头，后面跟1个或多个操作系统的名称，通过空格分隔

文件名，如果 \*\_GOOS, \*\_GOARCH, or \*\_GOOS\_GOARCH，则表示只在该平台上编译

## 交叉编译mips64

*下载go编译器源码*

[*https://studygolang.com/dl/golang/go1.9.2.linux-amd64.tar.gz*](https://studygolang.com/dl/golang/go1.9.2.linux-amd64.tar.gz)

*解压进入源码目录:*

*/root/golang-compile/go/src*

*export GOROOT\_BOOTSTRAP="/usr/lib/go"*

*编译mips64 go编译器*

*GOOS=linux GOARCH=mips64 ./bootstrap.bash*

*docker run -it --name u1 --privileged -e DOCKER\_CROSSPLATFORMS="linux/mips64" -v/root/go\_src/moby:/go/src/github.com/docker/docker dockercore/docker ./hack/make.sh cross*

## go help c

有两种方法在GO和C/C++代码之间相互调用.

There are two different ways to call between Go and C/C++ code.

The first is the cgo tool, which is part of the Go distribution. For

information on how to use it see the cgo documentation (go doc cmd/cgo).

The second is the SWIG program, which is a general tool for

interfacing between languages. For information on SWIG see

http://swig.org/. When running go build, any file with a .swig

extension will be passed to SWIG. Any file with a .swigcxx extension

will be passed to SWIG with the -c++ option.

When either cgo or SWIG is used, go build will pass any .c, .m, .s,

or .S files to the C compiler, and any .cc, .cpp, .cxx files to the C++

compiler. The CC or CXX environment variables may be set to determine

the C or C++ compiler, respectively, to use.

## cgo

cgo允许创建调用C代码的GO包.

通过go命令使用cgo

在GO代码中导入一个伪包”C”,然后GO代码可以引用C中的类型，如C.size\_t;C中的变量,如C.stdout,或者函数C.putchar

如果”import “C””之前有注释，这个注释称为序言。这在编译包的C代码部分时当作一个头。

如:

// #include <stdio.h>

// #include <errno.h>

import “C”

序言可以包含任何C代码，包括函数和变量声明和定义。它们被定义在C包中，可以在GO代码中引用。所有定义在序言中的名称都可以被使用，即使它们以小写字母开头。

例外：静态变量不可以被GO代码引用，但是静态函数可以。

GO编译器中有个例子:$GOROOT/misc/cgo/stdio and $GOROOT/misc/cgo/gmp

关于cgo的更多介绍:

<https://golang.org/doc/articles/c_go_cgo.html>.

CFLAGS, CPPFLAGS, CXXFLAGS, FFLAGS, LDFLAGS可以通过注释中的#cgo指令来定义，可以控制C,C++,Fortran编译器。多行定义的多个变量会被连接在一起。

// #cgo CFLAGS: -DPNG\_DEBUG=1

// #cgo amd64 386 CFLAGS: -DX86=1

// #cgo LDFLAGS: -lpng

// #include <png.h>

import "C"

另外，CPPFLAGS，LDFLAGS还可以通过pkg-config指定，使用#cgo pkg-config指令:

// #cgo pkg-config: png cairo

// #include <png.h>

import "C"

默认的pkg-config工具可以通过PKG\_CONFIG环境变量指定。

在构建的时候，CGO\_CFLAGS,CGO\_CPPFLAGS,CGO\_CXXFLAGS,CGO\_FFLAGS和CGO\_LDFLAGS环境变量会添加到这些指令所指示的标志中。包所有特有的标志应该通过cgo指令指定，而不应该通过环境变量指定，这样构建工作就不需要修改环境变量。

包中所有的cgo CPPFLAGS和CFLAGS指令被连接在一起用来编译包中的C文件。

所有的CPPFLAGS和CXXFLAGS指令被连接在一起用来编译包中的C++文件。

所有的CPPFLAGS和FFLAGS指令被连接在一起用来编译包中的Fortran文件。

所有的LDFLAGS被连接起来用于链接。所有的pkg-config指令被pkg-config使用。

在分析cgo指令时，出现的${SRCDIR}会被替换为包含源文件的绝对路径。这允许预编译的静态库被包含在包的路径而被合适的链接。比如，包foo在目录/go/src/foo:

// #cgo LDFLAGS: -L${SRCDIR}/libs -lfoo

将被扩展为:

// #cgo LDFLAGS: -L/go/src/foo/libs -lfoo

当GO工具发现一个或多个GO文件使用了”import “C””时,则会在目录中查找其它非GO文件并编译它们为包的一部分。任何.c,.s,.S文件将被C编译器编译，任何.cc,.cpp,.cxx文件将被c++编译器编译。任何.f,.F,.f90文件将被fortran编译器编译。任何.h,.hh,.hpp或者.hxx文件将不会独立编译，但是如果这些文件发生了改变，C，C++文件将会重新编译。默认的C,C++编译器可以通过CC,CXX环境变量改变，这些环境变量可以包含命令行选项。

cgo工具在本地系统构建时默认打开。当进行交叉编译时被禁用。你可以通过CGO\_ENABLED环境变量控制cgo，设置为１表示使用cgo,0来禁用它。如果使用cgo,go工具将会设置”#cgo”指令。

当进行交叉编译时，使用cgo必须指定交叉编译器。你在构建工具链时在make.bash中设置CC\_FOR\_TAGET环境变量,或者在使用go tool时设置CC环境变量。CXX\_FOR\_TARGET和CXX环境变量对于C++代码作用类似。

Go references to C

Within the Go file, C's struct field names that are keywords in Go can be

accessed by prefixing them with an underscore: if x points at a C struct

with a field named "type", x.\_type accesses the field. C struct fields that

cannot be expressed in Go, such as bit fields or misaligned data, are

omitted in the Go struct, replaced by appropriate padding to reach the next

field or the end of the struct.

The standard C numeric types are available under the names C.char, C.schar

(signed char), C.uchar (unsigned char), C.short, C.ushort (unsigned short),

C.int, C.uint (unsigned int), C.long, C.ulong (unsigned long), C.longlong

(long long), C.ulonglong (unsigned long long), C.float, C.double,

C.complexfloat (complex float), and C.complexdouble (complex double). The C

type void\* is represented by Go's unsafe.Pointer. The C types \_\_int128\_t and

\_\_uint128\_t are represented by [16]byte.

To access a struct, union, or enum type directly, prefix it with struct\_,

union\_, or enum\_, as in C.struct\_stat.

The size of any C type T is available as C.sizeof\_T, as in

C.sizeof\_struct\_stat.

As Go doesn't have support for C's union type in the general case, C's union

types are represented as a Go byte array with the same length.

Go structs cannot embed fields with C types.

Go code cannot refer to zero-sized fields that occur at the end of non-empty

C structs. To get the address of such a field (which is the only operation

you can do with a zero-sized field) you must take the address of the struct

and add the size of the struct.

Cgo translates C types into equivalent unexported Go types. Because the

translations are unexported, a Go package should not expose C types in its

exported API: a C type used in one Go package is different from the same C

type used in another.

Any C function (even void functions) may be called in a multiple assignment

context to retrieve both the return value (if any) and the C errno variable

as an error (use \_ to skip the result value if the function returns void).

For example:

n, err = C.sqrt(-1)

\_, err := C.voidFunc()

var n, err = C.sqrt(1)

Calling C function pointers is currently not supported, however you can

declare Go variables which hold C function pointers and pass them back and

forth between Go and C. C code may call function pointers received from Go.

For example:

package main

// typedef int (\*intFunc) ();

//

// int

// bridge\_int\_func(intFunc f)

// {

// return f();

// }

//

// int fortytwo()

// {

// return 42;

// }

import "C"

import "fmt"

func main() {

f := C.intFunc(C.fortytwo)

fmt.Println(int(C.bridge\_int\_func(f)))

// Output: 42

}

In C, a function argument written as a fixed size array actually requires a

pointer to the first element of the array. C compilers are aware of this

calling convention and adjust the call accordingly, but Go cannot. In Go,

you must pass the pointer to the first element explicitly: C.f(&C.x[0]).

A few special functions convert between Go and C types by making copies of

the data. In pseudo-Go definitions:

// Go string to C string

// The C string is allocated in the C heap using malloc.

// It is the caller's responsibility to arrange for it to be

// freed, such as by calling C.free (be sure to include stdlib.h

// if C.free is needed).

func C.CString(string) \*C.char

// Go []byte slice to C array

// The C array is allocated in the C heap using malloc.

// It is the caller's responsibility to arrange for it to be

// freed, such as by calling C.free (be sure to include stdlib.h

// if C.free is needed).

func C.CBytes([]byte) unsafe.Pointer

// C string to Go string

func C.GoString(\*C.char) string

// C data with explicit length to Go string

func C.GoStringN(\*C.char, C.int) string

// C data with explicit length to Go []byte

func C.GoBytes(unsafe.Pointer, C.int) []byte

As a special case, C.malloc does not call the C library malloc directly but

instead calls a Go helper function that wraps the C library malloc but

guarantees never to return nil. If C's malloc indicates out of memory, the

helper function crashes the program, like when Go itself runs out of memory.

Because C.malloc cannot fail, it has no two-result form that returns errno.

C references to Go

Go functions can be exported for use by C code in the following way:

//export MyFunction

func MyFunction(arg1, arg2 int, arg3 string) int64 {...}

//export MyFunction2

func MyFunction2(arg1, arg2 int, arg3 string) (int64, \*C.char) {...}

They will be available in the C code as:

extern int64 MyFunction(int arg1, int arg2, GoString arg3);

extern struct MyFunction2\_return MyFunction2(int arg1, int arg2, GoString arg3);

found in the \_cgo\_export.h generated header, after any preambles copied from

the cgo input files. Functions with multiple return values are mapped to

functions returning a struct. Not all Go types can be mapped to C types in a

useful way.

Using //export in a file places a restriction on the preamble: since it is

copied into two different C output files, it must not contain any

definitions, only declarations. If a file contains both definitions and

declarations, then the two output files will produce duplicate symbols and

the linker will fail. To avoid this, definitions must be placed in preambles

in other files, or in C source files.

Passing pointers

Go is a garbage collected language, and the garbage collector needs to know

the location of every pointer to Go memory. Because of this, there are

restrictions on passing pointers between Go and C.

In this section the term Go pointer means a pointer to memory allocated by

Go (such as by using the & operator or calling the predefined new function)

and the term C pointer means a pointer to memory allocated by C (such as by

a call to C.malloc). Whether a pointer is a Go pointer or a C pointer is a

dynamic property determined by how the memory was allocated; it has nothing

to do with the type of the pointer.

Go code may pass a Go pointer to C provided the Go memory to which it points

does not contain any Go pointers. The C code must preserve this property: it

must not store any Go pointers in Go memory, even temporarily. When passing

a pointer to a field in a struct, the Go memory in question is the memory

occupied by the field, not the entire struct. When passing a pointer to an

element in an array or slice, the Go memory in question is the entire array

or the entire backing array of the slice.

C code may not keep a copy of a Go pointer after the call returns.

A Go function called by C code may not return a Go pointer. A Go function

called by C code may take C pointers as arguments, and it may store

non-pointer or C pointer data through those pointers, but it may not store a

Go pointer in memory pointed to by a C pointer. A Go function called by C

code may take a Go pointer as an argument, but it must preserve the property

that the Go memory to which it points does not contain any Go pointers.

Go code may not store a Go pointer in C memory. C code may store Go pointers

in C memory, subject to the rule above: it must stop storing the Go pointer

when the C function returns.

These rules are checked dynamically at runtime. The checking is controlled

by the cgocheck setting of the GODEBUG environment variable. The default

setting is GODEBUG=cgocheck=1, which implements reasonably cheap dynamic

checks. These checks may be disabled entirely using GODEBUG=cgocheck=0.

Complete checking of pointer handling, at some cost in run time, is

available via GODEBUG=cgocheck=2.

It is possible to defeat this enforcement by using the unsafe package, and

of course there is nothing stopping the C code from doing anything it likes.

However, programs that break these rules are likely to fail in unexpected

and unpredictable ways.

Using cgo directly

Usage:

go tool cgo [cgo options] [-- compiler options] gofiles...

Cgo transforms the specified input Go source files into several output Go

and C source files.

The compiler options are passed through uninterpreted when invoking the C

compiler to compile the C parts of the package.

The following options are available when running cgo directly:

-dynimport file

Write list of symbols imported by file. Write to

-dynout argument or to standard output. Used by go

build when building a cgo package.

-dynout file

Write -dynimport output to file.

-dynpackage package

Set Go package for -dynimport output.

-dynlinker

Write dynamic linker as part of -dynimport output.

-godefs

Write out input file in Go syntax replacing C package

names with real values. Used to generate files in the

syscall package when bootstrapping a new target.

-srcdir directory

Find the Go input files, listed on the command line,

in directory.

-objdir directory

Put all generated files in directory.

-importpath string

The import path for the Go package. Optional; used for

nicer comments in the generated files.

-exportheader file

If there are any exported functions, write the

generated export declarations to file.

C code can #include this to see the declarations.

-gccgo

Generate output for the gccgo compiler rather than the

gc compiler.

-gccgoprefix prefix

The -fgo-prefix option to be used with gccgo.

-gccgopkgpath path

The -fgo-pkgpath option to be used with gccgo.

-import\_runtime\_cgo

If set (which it is by default) import runtime/cgo in

generated output.

-import\_syscall

If set (which it is by default) import syscall in

generated output.

-debug-define

Debugging option. Print #defines.

-debug-gcc

Debugging option. Trace C compiler execution and output.

golang与C联合开发的模块，编译时报错：

can't load package: no buildable Go source files in /...

问题原因在于，关闭cgo的情况下，golang的编译器会自动“忽略”掉带有"import "C""字样的源文件。解决这个问题需要打开cgo的支持，如编译时的命令改为：

GOARCH=arm CGO\_ENABLED=1 go build hello.go

但这还不够，还需要安装arm平台的编译器，如gcc-4.7-arm-linux-gnueabi（版本号要与目标平台的版本号一致），以及gcc-4.7-multilib-arm-linux-gnueabi的支持。还需要将/usr/bin/gcc重新ln到arm的交叉编译器上。

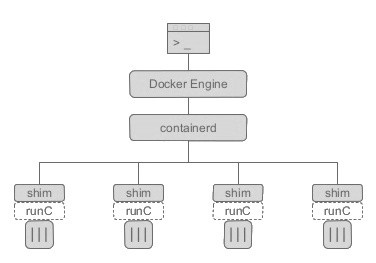
完成以上工作后，import "C"就没问题了。

1.3 第三方库

使用import "C"的主要目的就是使用第三方开源库，golang查找第三方库使用的是pkg-config，首先要保证pkg-config正常工作；另外，要保证第三方库，及第三方库的依赖库，都是arm版本的，才能正常link成功。链接成功的golang的程序，就可以愉快的在arm板上运行了。

golang的初步冒险就到些结束了，从下代码到完成这篇blog，用时一天多的时间，我的程序也快乐的在板子上运行中。

# docker组件



## dockerd

在Docker 1.8之前，Docker守护进程启动的命令为：

docker -d

这个阶段，守护进程看上去只是Docker client的一个选项。

Docker 1.8开始，启动命令变成了：

docker daemon

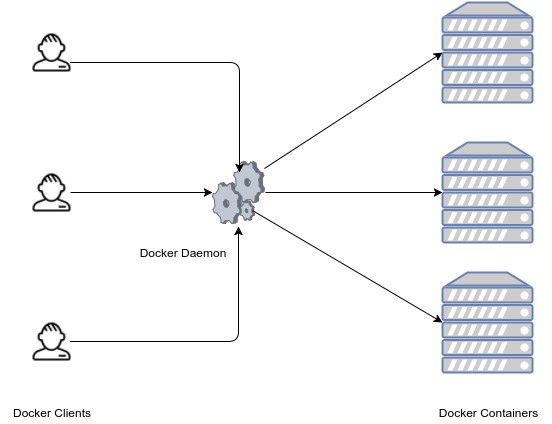
这个阶段，守护进程看上去是docker命令的一个模块。

Docker 1.11开始，守护进程启动命令变成了：

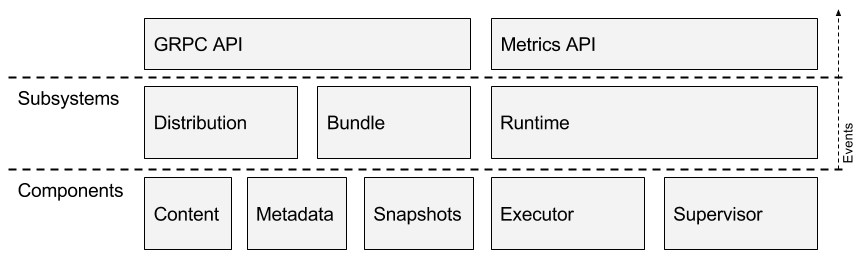
dockerd

此时已经和Docker client分离，独立成一个二进制程序了。

当然，守护进程模块不停的在重构，其基本功能和定位没有变化。和一般的CS架构系统一样，守护进程负责和Docker client交互，并管理Docker镜像、容器。



## containerd



containerd是一个守护进程用来控制runC.

containerd使用runC的高级特性，如seccomp,user namespace,checkpoint,容器clone和活迁移.

[containerd](https://github.com/docker/containerd)是容器技术标准化之后的产物，为了能够兼容[OCI标准](https://www.opencontainers.org/)，将容器运行时及其管理功能从Docker Daemon剥离。理论上，即使不运行dockerd，也能够直接通过containerd来管理容器。（当然，containerd本身也只是一个守护进程，容器的实际运行时由后面介绍的runC控制。）

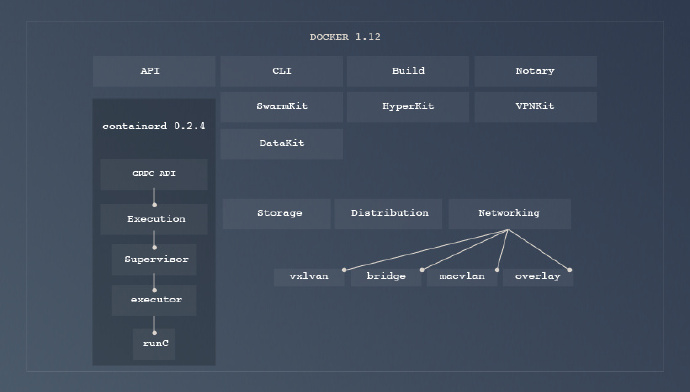
最近，Docker刚刚宣布[开源containerd](http://www.infoq.com/cn/news/2017/01/Docker-Containerd-OCI-1)。从其项目[介绍页面](https://github.com/docker/containerd/blob/master/design/architecture.md)可以看出，containerd主要职责是镜像管理（镜像、元信息等）、容器执行（调用最终运行时组件执行）。

containerd向上为Docker Daemon提供了gRPC接口，使得Docker Daemon屏蔽下面的结构变化，确保原有接口向下兼容。向下通过containerd-shim结合runC，使得引擎可以独立升级，避免之前Docker Daemon升级会导致所有容器不可用的问题。

Docker、containerd和containerd-shim之间的关系，可以通过启动一个Docker容器，观察进程之间的关联。首先启动一个容器

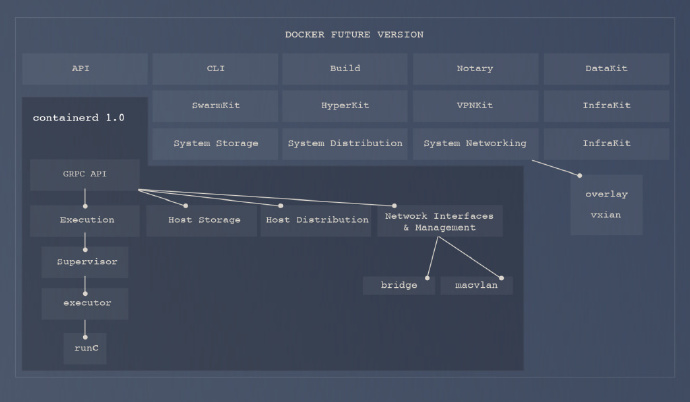
不久前，Docker[宣布](https://www.docker.com/docker-news-and-press/docker-extracts-and-donates-containerd-its-core-container-runtime-accelerate)开源他们的容器运行时组件[Containerd](https://github.com/docker/containerd)（发音是container-D）。目前的Containerd版本是0.2.4，Docker计划在2017年第二季度基于Open Container Initiative（OCI）发布1.0版本。

Containerd是一个容器运行时组件，它原本是Docker平台的一部分，虽然小巧，但在整个系统里起到很关键的作用。Containerd被设计成一种低耦合并且很容易与其它工具集成的组件。它在Docker 1.12里所处的位置如下图所示：



从图中可以看出，Containerd提供了一组运行容器的API。Docker引擎通过调用GRPC API来启动执行进程，随后会启动管理器和执行器来负责监控和运行容器。容器最后通过[runC](http://runc.io/)来运行，runC也是Docker的另一个开源项目，它实现了[OCI运行时标准](https://www.opencontainers.org/)。

计划中的Containerd 1.0版本将会有所变化，它在Docker中的位置将会是如下图所示的样子：



新版的Containerd将包含如下特性：

* 一个分布式的组件，它负责处理到注册中心的推送，无需与特定厂商关联。
* 一组网络原语，用来创建系统接口和API，以便管理容器的网络命名空间。
* 主机级别的镜像和容器文件系统存储。
* 一组GRPC API。
* [Prometheus](https://prometheus.io/)格式的度量指标API，用在内部和容器级别的度量指标上。
* 完全支持OCI镜像和runC的参考实现。

关于Containerd的更多架构细节可以参看[GitHub](https://github.com/docker/containerd/blob/master/design/architecture.md)主页的内容。

Containerd是Docker开源的众多项目中的新成员，这些项目包括libcontainer、libnetwork、notary、runC、HyperKit、VPNkit、Datakit、swarmkit和Infrakit等。

此次开源Containerd，Docker是希望能够与广大社区和其它各大厂商巨头共同构建一个双赢的结果。正如Docker CTO Solomon Hykes所说的那样，“我们迫不及待地要把这个东西贡献出来，因为我们也会因此获得很好的回报，这是一种良性的回馈闭环，这个闭环里的各方都会从中受益”。基于Containerd，企业可以构建自己的容器管理软件。目前，阿里巴巴、Amazon、Google、Microsoft和IBM等公司的员工已经加入到该项目的贡献者行列。而随着该项目的不断完善，Docker也将因此得到好处。

为了避免Containerd与Docker或其它商业实体存在关联，Docke计划明年把Containerd独立出来，并交由中立基金进行管理

docker run -d busybox sleep 1000

然后通过pstree命令查看进程之间的父子关系（其中20708是dockerd的PID）：

pstree -l -a -A 20708

输出结果如下：

dockerd -H fd:// --storage-driver=overlay2

|-docker-containe -l unix:///var/run/docker/libcontainerd/docker-containerd.sock --metrics-interval=0 --start-timeout 2m --state-dir /var/run/docker/libcontainerd/containerd --shim docker-containerd-shim --runtime docker-runc

| |-docker-containe b9a04a582b66206492d29444b5b7bc6ec9cf1eb83eff580fe43a039ad556e223 /var/run/docker/libcontainerd/b9a04a582b66206492d29444b5b7bc6ec9cf1eb83eff580fe43a039ad556e223 docker-runc

| | |-sleep 1000

虽然pstree命令截断了命令，但我们还是能够看出，当Docker daemon启动之后，dockerd和docker-containerd进程一直存在。当启动容器之后，docker-containerd进程（也是这里介绍的containerd组件）会创建docker-containerd-shim进程，其中的参数b9a04a582b66206492d29444b5b7bc6ec9cf1eb83eff580fe43a039ad556e223就是要启动容器的id。最后docker-containerd-shim子进程，已经是实际在容器中运行的进程（既sleep 1000）。

docker-containerd-shim另一个参数，是一个和容器相关的目录/var/run/docker/libcontainerd/b9a04a582b66206492d29444b5b7bc6ec9cf1eb83eff580fe43a039ad556e223，里面的内容有：

.

├── config.json

├── init-stderr

├── init-stdin

└── init-stdout

其中包括了容器配置和标准输入、标准输出、标准错误三个管道文件。

## RunC

OCI定义了容器运行时标准，runC是Docker按照开放容器格式标准（OCF, Open Container Format）制定的一种具体实现。

runC是从Docker的libcontainer中迁移而来的，实现了容器启停、资源隔离等功能。Docker默认提供了docker-runc实现，事实上，通过containerd的封装，可以在Docker Daemon启动的时候指定runc的实现。

我们可以通过启动Docker Daemon时增加--add-runtime参数来选择其他的runC现。例如：

docker daemon --add-runtime "custom=/usr/local/bin/my-runc-replacement"

## 举个例子

这里通过Docker一些命令，实现不使用Docker Daemon直接启动一个镜像，以便了解Docker Daemon每个模块的作用。

首先，需要创建容器标准包，这部分实际上由containerd的bundle模块实现，将Docker镜像转换成容器标准包。

mkdir my\_container

cd my\_container

mkdir rootfs

docker export $(docker create busybox) | tar -C rootfs -xvf -

上述命令将busybox镜像解压缩到指定的rootfs目录中。如果本地不存在busybox镜像，containerd还会通过distribution模块去远程仓库拉取。

现在整个my\_container目录结构如下：

$ tree -d my\_container/

my\_container/

└── rootfs

├── bin

├── dev

│ ├── pts

│ └── shm

├── etc

├── home

├── proc

├── root

├── sys

├── tmp

├── usr

│ └── sbin

└── var

├── spool

│ └── mail

└── www

17 directories

此时，标准包所需的容器数据已经准备完毕，接下来我们需要创建配置文件：

docker-runc spec

此时会生成一个名为config.json的配置文件，该文件和Docker容器的配置文件类似，主要包含容器挂载信息、平台信息、进程信息等容器启动依赖的所有数据。

最后，可以通过runc命令来启动容器：

runc run busybox

注意，runc必须使用root权限启动。

执行之后，我们可以看见容器已经启动：

localhost my\_container # runc run busybox

/ # ps aux

PID USER TIME COMMAND

1 root 0:00 sh

9 root 0:00 ps aux

此时，事实上已经可以不依赖Docker本身，如果系统上安装了runc包，即可运行容器。对于Gentoo系统来说，安装app-emulation/runc包即可。

当然，也可以使用docker-runc命令来启动容器：

localhost my\_container # docker-runc run busybox

/ # ps aux

PID USER TIME COMMAND

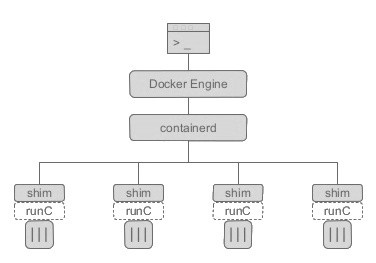
1 root 0:00 sh

7 root 0:00 ps aux

从这里可以看到标准化的重要性。

## 总结

从Docker 1.11之后，Docker Daemon被分成了多个模块以适应OCI标准。拆分之后，结构分成了以下几个部分。



其中，containerd独立负责容器运行时和生命周期（如创建、启动、停止、中止、信号处理、删除等），其他一些如镜像构建、卷管理、日志等由Docker Daemon的其他模块处理。

Docker的模块块拥抱了开放标准，希望通过OCI的标准化，容器技术能够有很快的发展。

# docker engine API

docker提供了与daemon交互的API,有Go和Python的SDKs.还可以直接使用Docker Engine API.

Docker Engine API是一个RESTful API通过HTTP客户端wget或者curl访问，可以使用HTTP库来访问

安装sdk

Go SDK:

go get github.com/docker/docker/client

Python SDK:

pip install docker

如果不能使用pip:

1. [Download the package directly](https://pypi.python.org/pypi/docker/).
2. Extract it and change to the extracted directory,
3. Run python setup.py install.

## API和SDK版本

Docker daemon和client api版本不匹配.

docker daemon和client的版本不需要完全匹配，要注意:

* 如果daemon比client新,client将不会感知新特性.
* 如果client比daemon新,client可以访问daemon不知道的API endpoints

HTTP举例:

$ curl --unix-socket /var/run/docker.sock -H "Content-Type: application/json" \

-d '{"Image": "alpine", "Cmd": ["echo", "hello world"]}' \

-X POST http:/v1.24/containers/create

{"Id":"1c6594faf5","Warnings":null}

$ curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/1c6594faf5/start

$ curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/1c6594faf5/wait

{"StatusCode":0}

$ curl --unix-socket /var/run/docker.sock "http:/v1.24/containers/1c6594faf5/logs?stdout=1"

hello world

.

## api版本矩阵

| **Docker version** | **Maximum API version** |
| --- | --- |
| 17.10 | [1.33](https://docs.docker.com/engine/api/v1.33/) |
| 17.09 | [1.32](https://docs.docker.com/engine/api/v1.32/) |
| 17.07 | [1.31](https://docs.docker.com/engine/api/v1.31/) |
| 17.06 | [1.30](https://docs.docker.com/engine/api/v1.30/) |
| 17.05 | [1.29](https://docs.docker.com/engine/api/v1.29/) |
| 17.04 | [1.28](https://docs.docker.com/engine/api/v1.28/) |
| 17.03 | [1.27](https://docs.docker.com/engine/api/v1.27/) |
| 1.13 | [1.26](https://docs.docker.com/engine/api/v1.26/) |
| 1.12 | [1.24](https://docs.docker.com/engine/api/v1.24/) |
| 1.11 | [1.23](https://docs.docker.com/engine/api/v1.23/) |
| 1.10 | [1.22](https://docs.docker.com/engine/api/v1.22/) |
| 1.9 | [1.21](https://docs.docker.com/engine/api/v1.21/) |
| 1.8 | [1.20](https://docs.docker.com/engine/api/v1.20/) |
| 1.7 | [1.19](https://docs.docker.com/engine/api/v1.19/) |
| 1.6 | [1.18](https://docs.docker.com/engine/api/v1.18/) |

# Dockerfile

**switch** c := cmd.(**type**) {  
**case** \*instructions.EnvCommand:  
 **return** dispatchEnv(d, c)  
**case** \*instructions.MaintainerCommand:  
 **return** dispatchMaintainer(d, c)  
**case** \*instructions.LabelCommand:  
 **return** dispatchLabel(d, c)  
**case** \*instructions.AddCommand:  
 **return** dispatchAdd(d, c)  
**case** \*instructions.CopyCommand:  
 **return** dispatchCopy(d, c)  
**case** \*instructions.OnbuildCommand:  
 **return** dispatchOnbuild(d, c)  
**case** \*instructions.WorkdirCommand:  
 **return** dispatchWorkdir(d, c)  
**case** \*instructions.RunCommand:  
 **return** dispatchRun(d, c)  
**case** \*instructions.CmdCommand:  
 **return** dispatchCmd(d, c)  
**case** \*instructions.HealthCheckCommand:  
 **return** dispatchHealthcheck(d, c)  
**case** \*instructions.EntrypointCommand:  
 **return** dispatchEntrypoint(d, c)  
**case** \*instructions.ExposeCommand:  
 **return** dispatchExpose(d, c, envs)  
**case** \*instructions.UserCommand:  
 **return** dispatchUser(d, c)  
**case** \*instructions.VolumeCommand:  
 **return** dispatchVolume(d, c)  
**case** \*instructions.StopSignalCommand:  
 **return** dispatchStopSignal(d, c)  
**case** \*instructions.ArgCommand:  
 **return** dispatchArg(d, c)  
**case** \*instructions.ShellCommand:  
 **return** dispatchShell(d, c)

每条指令产生一个Image:

type Image struct {

V1Image

Parent ID `json:"parent,omitempty"`

RootFS \*RootFS `json:"rootfs,omitempty"`

History []History `json:"history,omitempty"`

OSVersion string `json:"os.version,omitempty"`

OSFeatures []string `json:"os.features,omitempty"`

// rawJSON caches the immutable JSON associated with this image.

rawJSON []byte

// computedID is the ID computed from the hash of the image config.

// Not to be confused with the legacy V1 ID in V1Image.

computedID ID

}

**type** ChildConfig **struct** {  
 ContainerID string  
 Author string  
 Comment string  
 DiffID layer.DiffID  
 ContainerConfig \*container.Config  
 Config \*container.Config  
}

*// imageMount is a reference to an image that can be used as a builder.Source***type** imageMount **struct** {  
 image builder.Image  
 source builder.Source  
 layer builder.ReleaseableLayer  
}

newImage := image.NewChildImage(parentImage, image.ChildConfig{  
 Author: state.maintainer,  
 ContainerConfig: runConfig,  
 DiffID: newLayer.DiffID(),  
 Config: copyRunConfig(state.runConfig),  
}, parentImage.OS)

# 目录结构

以overlay2为例：

containers:以容器ID为目录名的目录。

下面存放了容器特有的文件 hostname, hosts, resolv.conf, resolv.conf.hash

配置文件:config.v2.json

共享内存:shm

容器日志文件:<container-id>-json.log

image:

overlay2

/distribution

/imagedb

/content:

/metadata:元数据，镜像层次关系

/sha256:以<IMAGEID>为名称的目录，下面有parent文件，记录了父镜像ID

/layerdb

repositories.json:存储镜像库中所有的仓库地址

network:

overlay2:

swarm:

tmp:

trust:

volumes:

# 源码阅读

目录结构：

/api:docker engine api的一部分，包含了swagger定义和client和daemon共用的类型定义

/types: client和daemon共用的类型定义

swagger.yaml:swagger定义

/cli:使用cobra构建的命令行客户端

/client:cli构建cli使用的Go客户端包，也可用于第三方GO程序

/daemon:服务端，提供API服务

/builder:定义了docker builder的实现接口，可用于其它的docker builder实现

/cmd:用于执行docker命令

/docker:客户端命令

/dockerd:服务端命令

/container:容器相关定义

/contrib:包含了脚本，镜像和其它有用的工具。不属于docker发布的核心东西。它可能会过期。

build镜像

# 其它问题

**问题：**

1、How can enable udev sync successfully in docker? #13179

2、Dynamically linked docker binary reports devicemapper and Udev not supported #12925

3、devicemapper: set flags based on udev sync support #10203

4、Docker APT repository binary has devicemapper "Udev Sync Supported: false" #10705

5、docker不支持在使用devicemapper存储类型下打开Udev Sync功能？

**解决方案：**

对docker1.7.1版本的解决方法：  
1、先检查是否为devicemapper没有支持udev\_sync功能：dmsetup udevcookies。如果没有支持会提示要求以--enable-udev\_sync方式重新编译devicemapper，可按2，3步骤重新编译devicemapper。  
2、安装最新版本的libudev: apt-get install libudev-dev  
3、下载lvm2源码：git clone -b v2\_02\_103 <https://git.fedorahosted.org/git/lvm2.git> /usr/local/lvm2  
指定--enable-udev\_sync参数，重新编译安装devicemapper：./configure --enable-udev\_sync & make device-mapper & make install\_device-mapper  
3、编译动态链接的docker程序：AUTO\_GOPATH=1 ./hack/make.sh dynbinary  
4、启动docker daemon时指定存储驱动器类型为devicemapper：./docker -d -D --storage-driver=devicemapper -b=my\_br -H tcp://0.0.0.0:1234 -H unix:///var/run/docker.sock &  
（-s 选项(storage-driver)是关键，其他按自己的情况来）  
5、docker info查看结果中的Udev Sync Supported。

对最新版本的docker1.11：

编译出来后检查：

[root@dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary](mailto:root@hzzengchao1-dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary)# ldd docker  
        linux-vdso.so.1 =>  (0x00007ffe3997c000)  
        libpthread.so.0 => /lib/x86\_64-linux-gnu/libpthread.so.0 (0x00007f8dbc497000)  
        libdl.so.2 => /lib/x86\_64-linux-gnu/libdl.so.2 (0x00007f8dbc293000)  
        libdevmapper.so.1.02 => /lib/libdevmapper.so.1.02 (0x00007f8dbc057000)  
        libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f8dbbcca000)  
        /lib64/ld-linux-x86-64.so.2 (0x00007f8dbc6b3000)  
        libselinux.so.1 => /lib/x86\_64-linux-gnu/libselinux.so.1 (0x00007f8dbbaaa000)  
        libsepol.so.1 => /lib/x86\_64-linux-gnu/libsepol.so.1 (0x00007f8dbb86a000)  
        libudev.so.0 => /lib/x86\_64-linux-gnu/libudev.so.0 (0x00007f8dbb65b000)  
        librt.so.1 => /lib/x86\_64-linux-gnu/librt.so.1 (0x00007f8dbb453000)

已经依赖libudev.so。

[root@dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary](mailto:root@hzzengchao1-dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary)# dmsetup udevcookies  
Cookie       Semid      Value      Last semop time           Last change time  
0xd4dbe47    0          1          Fri Mar 25 21:09:29 2016  Fri Mar 25 21:09:29 2016  
0xd4d1f0e    32769      1          Fri Mar 25 21:14:00 2016  Fri Mar 25 21:14:00 2016

已经有支持udev\_sync的device-mapper。

[root@dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary](mailto:root@hzzengchao1-dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary)# ./docker -d -D --storage-driver=devicemapper -b=my\_br -H tcp://0.0.0.0:1234 -H unix:///var/run/docker.sock

[root@dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary](mailto:root@hzzengchao1-dev:/home/code/docker/docker/bundles/1.11.0-dev/dynbinary)# docker info  
DEBU[0005] Calling GET /v1.19/info                        
DEBU[0005] Client and server don't have the same version (client: 1.7.1, server: 1.11.0-dev)   
Containers: 0  
Images: 0  
Storage Driver: devicemapper  
 Udev Sync Supported: true

（其他信息省略）

已经支持udev\_sync了。（也可能和我之前重编译了device-mapper有关了）。

总结下来就一句话：只要device-mapper支持udev\_sync，docker为动态链接依赖库即可。

Reference:

<http://icarobichir.com.br/posts/building-a-dynamically-linked-docker-v-1-6-2/>

<https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Logical_Volume_Manager_Administration/udev_commands_interfaces.html>