[[docker 1.13 源码分析]第一章 Docker Daemon](http://blog.csdn.net/zhonglinzhang/article/details/53333992)

#### 1.0 daemon基础知识

Daemon通过三种方式监听请求，unix，tcp，fd，默认使用unix domain socket(/var/run/[**Docker**](http://lib.csdn.net/base/4).sock)。对于远程请求，可以开启tcp socket(-H tcp:0.0.0.0:2375)，或者固定IP(-H tcp://192.168.0.1:2375)。可以使用多种配置如下：

$ sudo dockerd -H unix:///var/run/docker.sock -H tcp://192.168.59.106 -H tcp://10.10.10.2

docker为docker client 和 docker daemon，client端发送命令，daemon端负责执行client发送过来的命令（获取和存储镜像、管理容器等）。两者可以通过TCP，HTTP和UNIX SOCKET来进行通信

#### 1.1 **main函数**

**func**main() {  
       **if**reexec.Init() {  
              **return**}  
  
       // Set terminal emulation based on platform as required.  
       \_, stdout, stderr := term.StdStreams()  
       logrus.SetOutput(stderr)  
  
       cmd := newDaemonCommand()  
       cmd.SetOutput(stdout)  
       **if**err := cmd.Execute(); err != nil {  
              fmt.Fprintf(stderr, "%s\n", err)  
              os.Exit(1)  
       }  
}

main函数开始，**首先reexec.Init()**（在pkg/reexec/reexec.go文件中），如果有注册的初始化函数return。

stdin, stdout, stderr := **term.StdStreams()**   标准输入、输出、错误流

go get -v github.com/spf13/cobra/cobra

cmd包进行了初始化操作并提供接口。其中viper是cobra集成的配置文件读取库。cobra的命令都是cobra.Command结构体实现。

// The \*Run functions are executed in the following order:  
//   \* PersistentPreRun()  
//   \* PreRun()  
//   \* Run()  
//   \* PostRun()  
//   \* PersistentPostRun()  
// All functions get the same args, the arguments after the command name  
// PersistentPreRun: children of this command will inherit and execute

一般使用Run函数

// Run: Typically the actual work function. Most commands will only implement this  
Run **func**(cmd \*Command, args []string)  
// RunE: Run but returns an error  
RunE **func**(cmd \*Command, args []string) error

#### 1.2 **runDaemon函数**

根命令回调函数，主要函数是deemonCli.start(opts)函数，启动一个web服务。

**func**runDaemon(opts daemonOptions) error {  
      ..........................................  
       err = daemonCli.start(opts)  
       notifyShutdown(err)  
       **return**err  
}

#### 1.3 **(cli \*DaemonCli) start(opts daemonOptions)函数**

**if**cli.Config, err = loadDaemonCliConfig(opts); err != nil {  
       **return**err  
}

加载配置文件参数为daemonOptions结构体，填充返回\*daemon.Config结构体指针。

**if**err := setDefaultUmask(); err != nil {  
       **return**fmt.Errorf("Failed to set umask: %v", err)  
}

设置文件的权限

**if**len(cli.Config.Hosts) == 0 {  
       cli.Config.Hosts = make([]string, 1)  
}

如果没有加-H选项参数，长度为0,直接运行，Host地址采用默认的/var/run/docker.sock

api := apiserver.New(serverConfig)

生成一个server对象，使用apiserver.Config初始化

**if**cli.Config.Hosts[i], err = dopts.ParseHost(cli.Config.TLS, cli.Config.Hosts[i]); err != nil {  
       **return**fmt.Errorf("error parsing -H %s : %v", cli.Config.Hosts[i], err)  
}

默认Host为0，通过ParseHost函数赋值DefaultHost（/var/run/docker.sock），本地访问的unix套接字形式地址

protoAddr := cli.Config.Hosts[i]  
protoAddrParts := strings.SplitN(protoAddr, "://", 2)  
**if**len(protoAddrParts) != 2 {  
       **return**fmt.Errorf("bad format %s, expected PROTO://ADDR", protoAddr)  
}  
  
proto := protoAddrParts[0]  
addr := protoAddrParts[1]

如果使用unix,那么proto就是unix, addr就是/var/run/docker.sock

ls, err := listeners.Init(proto, addr, serverConfig.SocketGroup, serverConfig.TLSConfig)  
**if**err != nil {  
       **return**err  
}

初始化监听地址，最终都是net.Listen，使用的Go网络库

#### 1.4 **Init(proto, addr, sockerGroup string, tlsConfig \*tls.Config)函数**

**func**Init(proto, addr, socketGroup string, tlsConfig \*tls.Config) ([]net.Listener, error) {  
       ls := []net.Listener{}  
  
       **switch**proto {  
       **case**"fd":  
              fds, err := listenFD(addr, tlsConfig)  
              **if**err != nil {  
                     **return**nil, err  
              }  
              ls = append(ls, fds...)  
       **case**"tcp":  
              l, err := sockets.NewTCPSocket(addr, tlsConfig)  
              **if**err != nil {  
                     **return**nil, err  
              }  
              ls = append(ls, l)  
       **case**"unix":  
              l, err := sockets.NewUnixSocket(addr, socketGroup)  
              **if**err != nil {  
                     **return**nil, fmt.Errorf("can't create unix socket %s: %v", addr, err)  
              }  
              ls = append(ls, l)  
       **default**:  
              **return**nil, fmt.Errorf("invalid protocol format: %q", proto)  
       }  
  
       **return**ls, nil  
}

Case “tcp”:  使用参数选项-H设置tcp地址，可以远程访问

Case “unix”:  默认情况本地访问，使用unix套接字监听本地

#### 1.5 **NewDaemon函数**

只要功能实现了所有的API接口

setDefaultMtu(config)

设置默认的MTU，这里的默认值为1500

**if**err := verifyDaemonSettings(config); err != nil {  
       **return**nil, err  
}

主要检查互不相容的配置，

①　config.bridgeConfig.Iface和config.bridgeConfig.IP，必须指定-b & --bip，这两项不能同时存在

②　config.bridgeConfig.EnableIpTables与config.bridgeConfig.InterContainerCommuication（icc使用iptables主要增加的是FORWARD规则链，表示容器之间是否可以通信。）不能同时为fase，

**if**err := checkSystem(); err != nil {  
       **return**nil, err  
}

主要验证dokcer的进程是否为root，还有包括linux kernel版本验证

**if**err := setupDaemonProcess(config); err != nil {  
       **return**nil, err  
}

将score写入/proc/self/oom\_score\_adj文件，

主要作用是内存溢出（OOM）

当内存严重不足时，内核有两种选择：

1.直接panic

2.杀掉部分进程，释放一些内核。

对于每个进程都有一个oom\_score的属性（/proc/PID/oom\_score），oom killer会杀死oom\_score较大的进程，当oom\_score为0时禁止内核杀死该进程。

通过命令 echo 0 > /proc/PID/oom\_adj 来防止进程被oom\_killer杀死。

tmp, err := tempDir(config.Root, rootUID, rootGID)  
**if**err != nil {  
       **return**nil, fmt.Errorf("Unable to get the TempDir under %s: %s", config.Root, err)  
}  
realTmp, err := fileutils.ReadSymlinkedDirectory(tmp)  
**if**err != nil {  
       **return**nil, fmt.Errorf("Unable to get the full path to the TempDir (%s): %s", tmp, err)  
}  
os.Setenv("TMPDIR", realTmp)

建立temp文件的存放路径，读取环境变量DOCKER\_TMPDIR，没有创建，默认路径为/var/lib/docker/temp

**if**err := configureKernelSecuritySupport(config, graphDriver); err != nil {  
       **return**nil, err  
}

设置系统是否使用SELinux

**if**err := v1.Migrate(config.Root, graphDriver, d.layerStore, d.imageStore, referenceStore, distributionMetadataStore); err != nil {  
       logrus.Errorf("Graph migration failed: %q. Your old graph data was found to be too inconsistent for upgrading to content-addressable storage. Some of the old data was probably not upgraded. We recommend starting over with a clean storage directory if possible.", err)  
}

将存在的old docker image和container存储形式迁移到新版本docker中。

trustKey, err := api.LoadOrCreateTrustKey(config.TrustKeyPath)  
**if**err != nil {  
       **return**nil, err  
}

在路径/etc/docker/key.json创建key，如下所示

{

    "crv": "P-256",

    "d": "M9o6439qe7\_JUJegJ-f6ZnP0KQYCNQPBHB8sX9ckm8Y",

    "kid": "VT72:HISH:NOXS:PM7O:GH4E:3OAN:6YUV:OU7Q:AI7V:W5GC:CBJZ:2H36",

    "kty": "EC",

    "x": "zUiIkJfvgdFdtuAEY6LORdNvgXWMVXbrScTr02IBwA0",

    "y": "ecNUUOz-AakKc0cqT2KIzvaQlj-31Iy2J\_lfZ036tJM"

}

[[docker 1.13 源码分析]第二章 Docker container创建](http://blog.csdn.net/zhonglinzhang/article/details/53435590)

# 第一章 ****Container create源码分析****

#### 1.1 **NewCreateCommand函数**(cli/command/container/create.go)

#### 1.2 **ContainerCreate函数**(client/container\_create.go)

创建一个容器在客户端实现是在client/container\_create.go这个文件唯一一个函数，传入的参数主要有容器基本配置container.Config，container.HostConfig，network.NetworkingConfig。

Config只要目的是基于容器的可移植性信息，与host相互独立，非可移植性在HostConfig结构体中。Config包括容器的基本信息，名字，输入输出流等，不解释结构体注释很清晰。

**type**Config **struct**{  
       Hostname        string                // Hostname  
       Domainname      string                // Domainname  
       User            string                // User that will run the command(s) inside the container, also support user:group  
       AttachStdin     bool                  // Attach the standard input, makes possible user interaction  
       AttachStdout    bool                  // Attach the standard output  
       AttachStderr    bool                  // Attach the standard error  
       ExposedPorts    nat.PortSet           `json:",omitempty"` // List of exposed ports  
       Tty             bool                  // Attach standard streams to a tty, including stdin if it is not closed.  
       OpenStdin       bool                  // Open stdin  
       StdinOnce       bool                  // If true, close stdin after the 1 attached client disconnects.  
       Env             []string              // List of environment variable to set in the container  
       Cmd             strslice.StrSlice     // Command to run when starting the container  
       Healthcheck     \*HealthConfig         `json:",omitempty"` // Healthcheck describes how to check the container is healthy  
       ArgsEscaped     bool                  `json:",omitempty"` // True if command is already escaped (Windows specific)  
       Image           string                // Name of the image as it was passed by the operator (e.g. could be symbolic)  
       Volumes         **map**[string]**struct**{}   // List of volumes (mounts) used for the container  
       WorkingDir      string                // Current directory (PWD) in the command will be launched  
       Entrypoint      strslice.StrSlice     // Entrypoint to run when starting the container  
       NetworkDisabled bool                  `json:",omitempty"` // Is network disabled  
       MacAddress      string                `json:",omitempty"` // Mac Address of the container  
       OnBuild         []string              // ONBUILD metadata that were defined on the image Dockerfile  
       Labels          **map**[string]string     // List of labels set to this container  
       StopSignal      string                `json:",omitempty"` // Signal to stop a container  
       StopTimeout     \*int                  `json:",omitempty"` // Timeout (in seconds) to stop a container  
       Shell           strslice.StrSlice     `json:",omitempty"` // Shell for shell-form of RUN, CMD, ENTRYPOINT  
}

只要是通过http请求发送至server端使用POST方法

**func**(cli \*Client) post(ctx context.Context, path string, query url.Values, obj **interface**{}, headers **map**[string][]string) (serverResponse, error) {  
       body, headers, err := encodeBody(obj, headers)  
       **if**err != nil {  
              **return**serverResponse{}, err  
       }  
       **return**cli.sendRequest(ctx, "POST", path, query, body, headers)  
}

#### 1.3 **postContainerCreate函数**

创建容器的操作对应在server端得实现在api/server/router/container/container\_routes.go中的postContainersCreate()函数；

主要工作是校验工作，从body获取1.2中的配置container配置，host配置，network配置，最终用的方法是调用接口方法ContainerCreate，这个方法是daemon中实现。

ccr, err := s.backend.ContainerCreate(types.ContainerCreateConfig{  
       Name:             name,  
       Config:           config,  
       HostConfig:       hostConfig,  
       NetworkingConfig: networkingConfig,  
       AdjustCPUShares:  adjustCPUShares,  
}, validateHostname)

#### 1.4 **ContainerCreate函数**

Daemon中的方法daemon/create.go文件。

主要是校验工作，不解释了。看主要方法daemon.create在1.5中讲解

container, err := daemon.create(params, managed)  
**if**err != nil {  
       **return**containertypes.ContainerCreateCreatedBody{Warnings: warnings}, daemon.imageNotExistToErrcode(err)  
}

#### 1.5 **create函数**

**if**params.Config.Image != "" {  
       img, err = daemon.GetImage(params.Config.Image)  
       **if**err != nil {  
              **return**nil, err  
       }  
  
       **if**runtime.GOOS == "solaris" && img.OS != "solaris " {  
              **return**nil, errors.New("Platform on which parent image was created is not Solaris")  
       }  
       imgID = img.ID()  
}

**if**container, err = daemon.newContainer(params.Name, params.Config, imgID, managed); err != nil {  
       **return**nil, err  
}

来new一个新的container；接下来的就是为启动容器所做的工作：

**if**err := daemon.setRWLayer(container); err != nil {  
       **return**nil, err  
}

创建读写层，详细看1.6讲解

rootUID, rootGID, err := idtools.GetRootUIDGID(daemon.uidMaps, daemon.gidMaps)  
**if**err != nil {  
       **return**nil, err  
}  
**if**err := idtools.MkdirAs(container.Root, 0700, rootUID, rootGID); err != nil {  
       **return**nil, err  
}  
**if**err := idtools.MkdirAs(container.CheckpointDir(), 0700, rootUID, rootGID); err != nil {  
       **return**nil, err  
}

以root uid gid的属性创建目录

**if**err := daemon.setHostConfig(container, params.HostConfig); err != nil {  
       **return**nil, err  
}

①　daemon.registerMountPoints注册所有挂载到容器的数据卷(1.8讲解)

②　daemon.registerLinks，load所有links（包括父子关系），写入host配置至文件（ 注册互联容器，容器可以通过 ip:端口访问，可以通过--link互联。）

③　container.ToDisk将container持久化至disk。路径为如下所示

/var/lib/[**Docker**](http://lib.csdn.net/base/4)/containers/$containerID

#### 1.6 **CreateRWLayer函数**

位于文件layer/layer\_store.go文件，layerStore结构体如下所示：

MetadataStore为接口，主要为获得层基本信息的方法。 metadata是这个层的额外信息，不仅能够让Docker获取运行和构建的信息，也包括父层的层次信息（只读层和读写层都包含元数据）。

graphdriver.Driver也为接口，主要以aufs主要介绍，在daemon/graphdriver文件下有aufs，btrfs，devmapper，overlay等的实现。 除差别和改动等的方法，graphdriver最主要的功能是 Get、 Put、 Create 和 Remove 方法 。

Aufs层关系

**type**layerStore **struct**{  
       store  MetadataStore  
       driver graphdriver.Driver  
  
       layerMap **map**[ChainID]\*roLayer  
       layerL   sync.Mutex  
  
       mounts **map**[string]\*mountedLayer  
       mountL sync.Mutex  
}

**type**roLayer **struct**{  
       chainID    ChainID  
       diffID     DiffID  
       parent     \*roLayer  
       cacheID    string  
       size       int64  
       layerStore \*layerStore  
       descriptor distribution.Descriptor  
  
       referenceCount int  
       references     **map**[Layer]**struct**{}  
}

每一层都包括指向父层的指针。如果没有这个指针，说明处于最底层。

#### 1.7 **CreateReadWrite函数**

文件位于daemon/gradphdriver/aufs/aufs.go。主要功能是创建三个目录mnt，layers，diff，在将父层id记录

**if**err := a.createDirsFor(id); err != nil {  
       **return**err  
}

创建三个目录 mnt，layers，diff

f, err := os.Create(path.Join(a.rootPath(), "layers", id))  
**if**err != nil {  
       **return**err  
}

以层的metadata为后缀名创建文件

**if**parent != "" {  
       ids, err := getParentIDs(a.rootPath(), parent)  
       **if**err != nil {  
              **return**err  
       }  
  
       **if**\_, err := fmt.Fprintln(f, parent); err != nil {  
              **return**err  
       }  
       **for**\_, i := **range**ids {  
              **if**\_, err := fmt.Fprintln(f, i); err != nil {  
                     **return**err  
              }  
       }  
}

获得该层的父层，存在将所有父层id记录该文件

#### 1.8 **registerMountPoints函数**

位于daemon/volumes.go,注册所有挂载到容器的数据卷，bind挂载。主要有三种方式和来源：

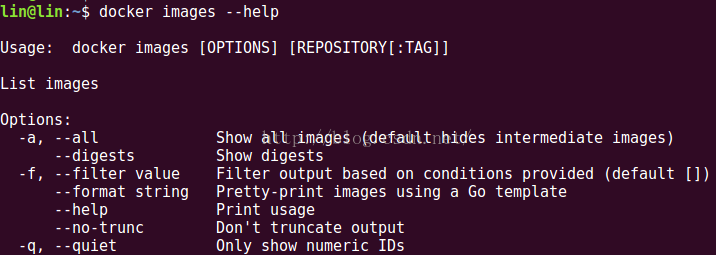
（1）容器本身自带的挂载的数据卷，容器的json镜像文件中 "Volumes"这个key对应内容；

（2）其他容器（--volumes-from）挂载的数据卷;

（3）命令行参数（-v）挂载与主机绑定的数据卷，与主机绑定得数据卷在docker中叫做bind-mounts；

[[docker 1.13 源码分析]第三章 docker pull image](http://blog.csdn.net/zhonglinzhang/article/details/53484799)

#### 1.0 **Image主要命令**



* $ docker images （所有）
* $ docker images java （所有java）
* $ docker images java:8 （固定tag的jave）
* $ docker images --no-trunc （所有id值全长度）
* $ docker images --digests （所有镜像带有digest）
* $ docker images --format "{{.ID}}: {{.Repository}}" （列出两项）
* $ docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}" （列出三项）

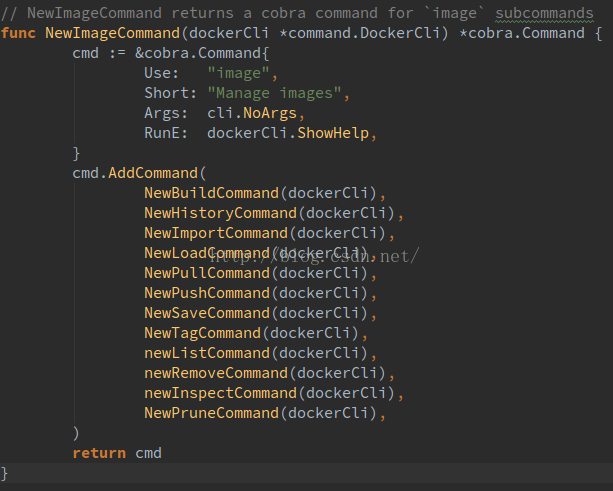
**pull主要命令**

**[plain]** [view plain](http://blog.csdn.net/zhonglinzhang/article/details/53484799) [copy](http://blog.csdn.net/zhonglinzhang/article/details/53484799)

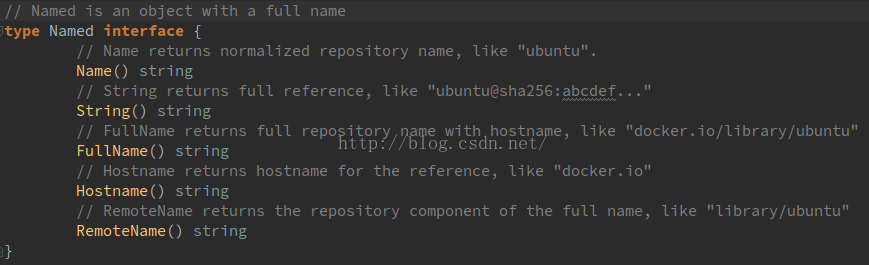
[print?](http://blog.csdn.net/zhonglinzhang/article/details/53484799)

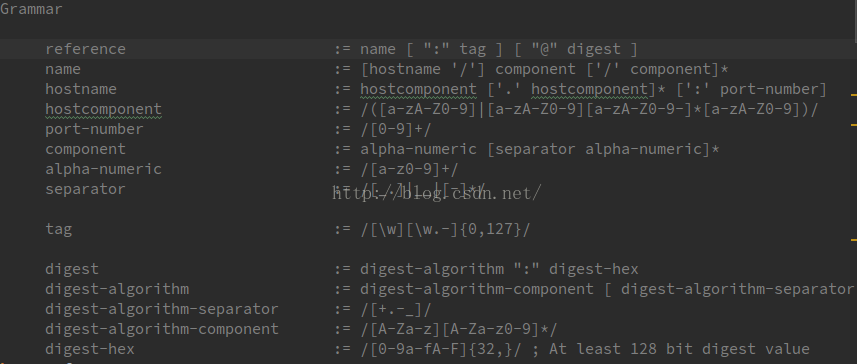
1. Usage:  docker pull [OPTIONS] NAME[:TAG|@DIGEST]
3. Pull an image or a repository from a registry
5. Options:
6. -a, --all-tags                Download all tagged images in the repository
7. --disable-content-trust   Skip image verification (default true)
8. --help                    Print usage

#### 1.1 **镜像的子命令**



#### 1.2 **Named接口**



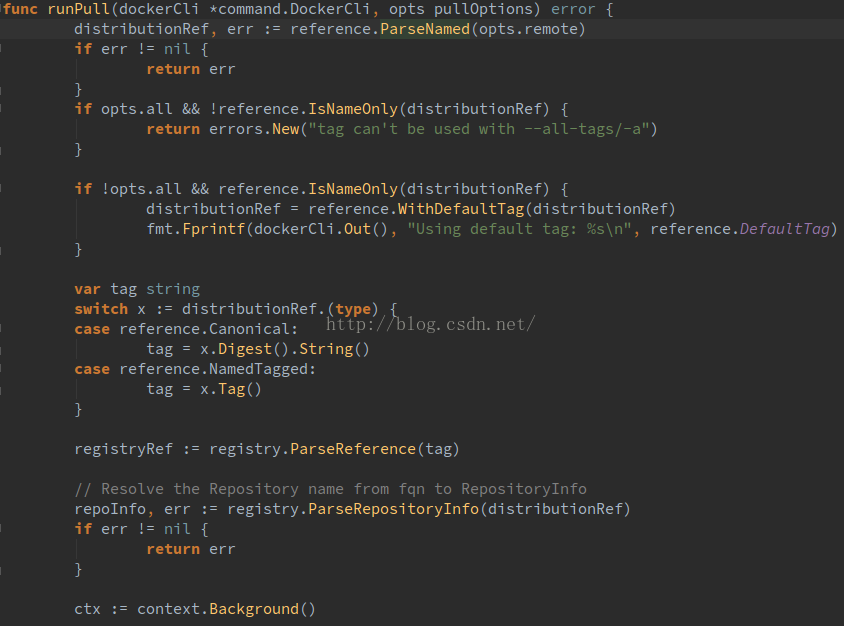


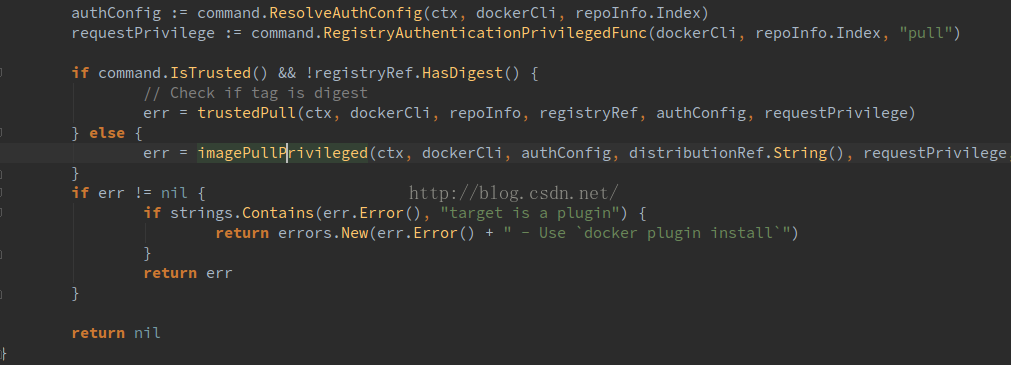
**拉取镜像的命令**： [**Docker**](http://lib.csdn.net/base/4) pull NAME[:TAG|@DIGEST] ，TAG为标签，DIGEST为数字摘要，也就是拉取镜像可以附带TAG或数字摘要等参数，或只使用镜像名（默认latest）。如果参数带TAG则使用NamedTagged描述 ，如果参数带DIGEST则使用Canonical 描述。

* $ docker images --digests
* $ docker mysql@sha256:89cc6ff6a7ac9916c3384e864fb04b8ee9415b572f872a2a4cf5b909dbbca81b
* $ docker pull library/mysql

#### 1.3 **runPull函数**

下面分析NewPullCommand这个子命令，位于cli/image/pull.go文件中。主要执行函数是runPull





①　reference.ParseNamed: 从参数中解析镜像仓库地址等信息，如参数中没有仓库信息，则使用默认的docker.io

②　opts.all && !reference.IsNameOnly(distributionRef): 如果使用了-a / --all-tags，但又含有tag信息则报错，如下

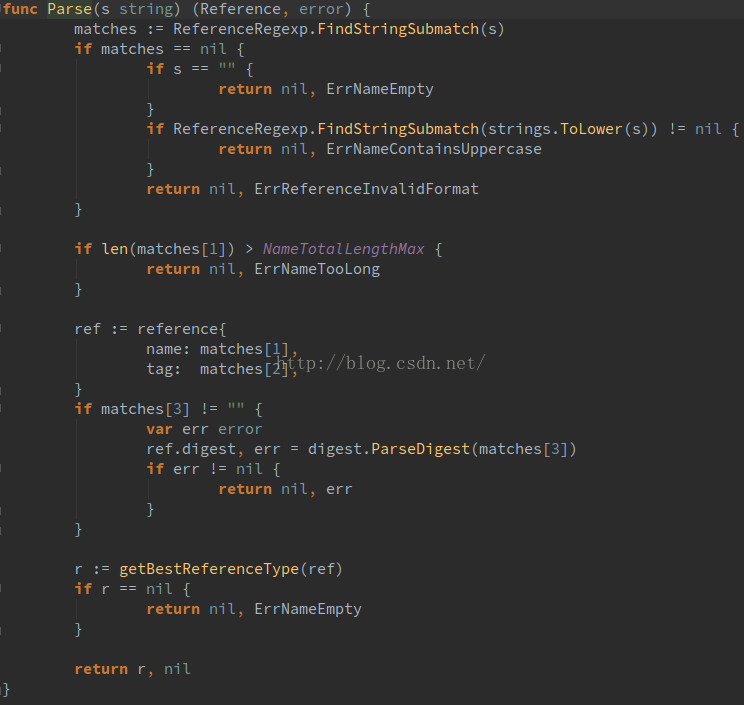
$ docker pull -a ubuntu:latest

   tag can't be used with --all-tags/-a

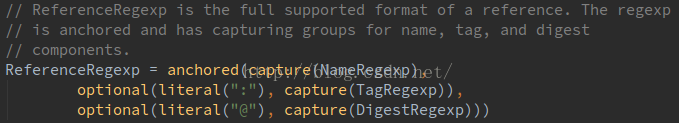
③　!opts.all && reference.IsNameOnly(distributionRef): 如果无-a / --all-tags参数，而只有镜像名，则使用默认latest的tag

④　ImagePullPrivileged: 拉取镜像发送请求至dockerd，使用ImagePull（1.6中讲解）

#### 1.4 **Parse函数**



ReferenceRegexp定义在vendor/github.com/docker/distribution/reference/regexp.go



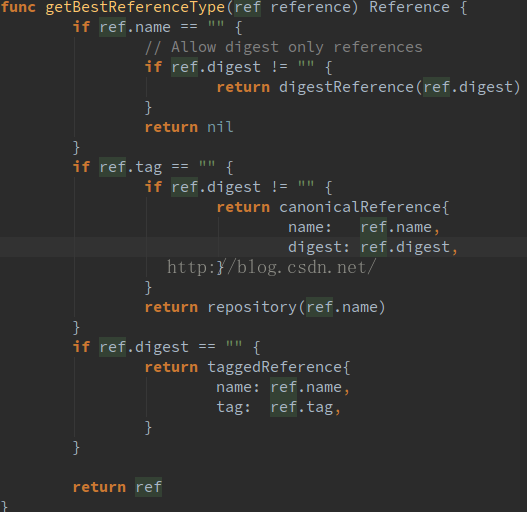
#### 1.5 **getBestReferenceType函数**

①　带镜像名，则matches[1]不为nil;

②　带tag，则matches[2]不为nil;

③　带数字摘要，则matches[3]不为nil

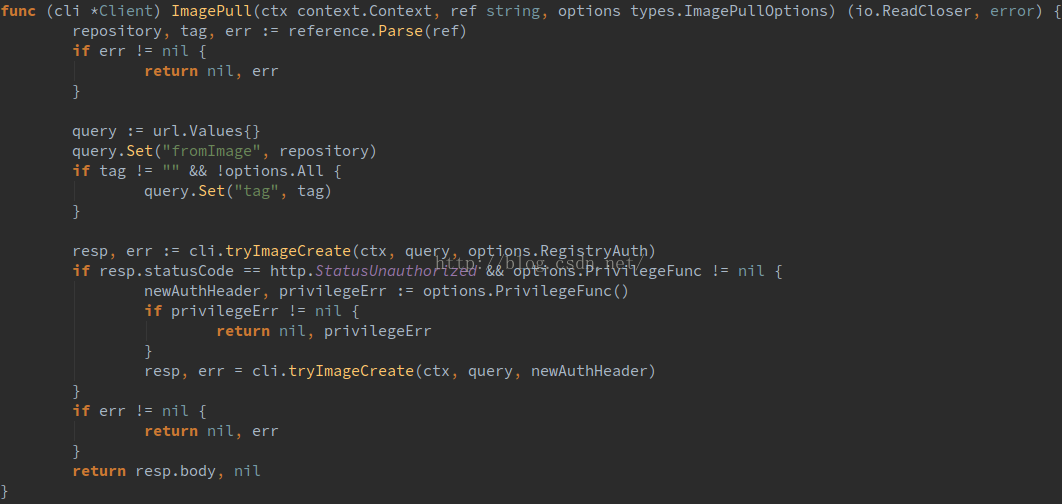
getBestReferenceType根据各个matchs是否为空，根据带相应的部分返回不同类型的Reference接口。



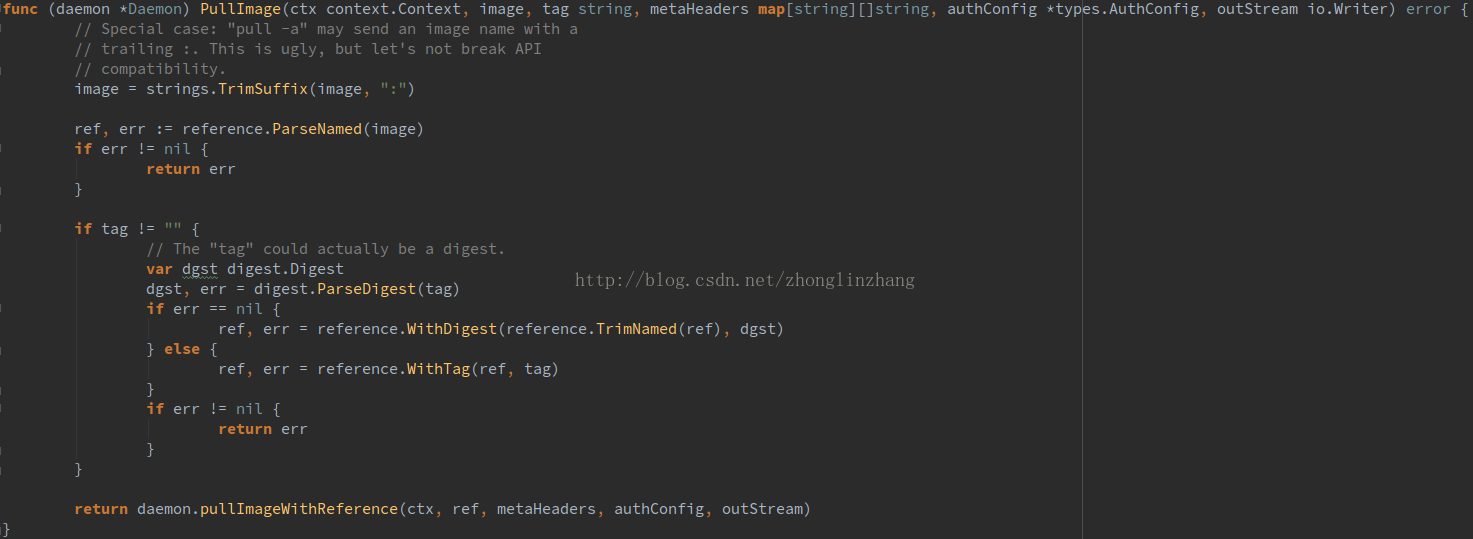
#### 1.6 **ImagePull函数**

文件位置client/image\_pull.go tryImageCreate调用http请求/images/create，

根据api路由定位到daemon/image\_pull.go文件中的PullImage函数，下面讲解该函数。



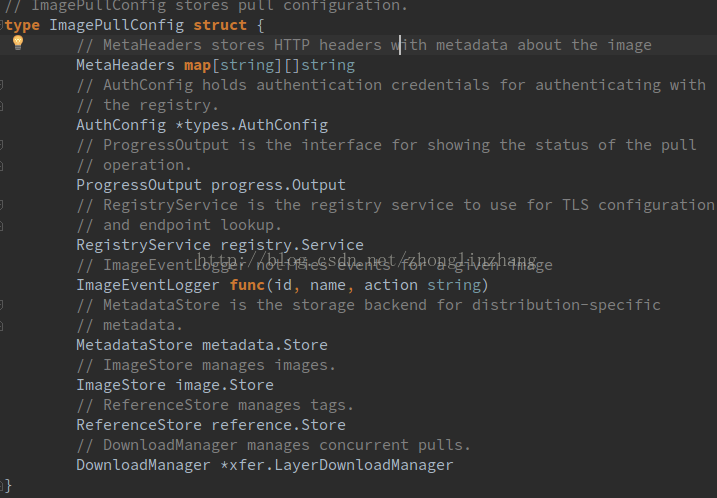
#### 1.7 **PullImage函数**



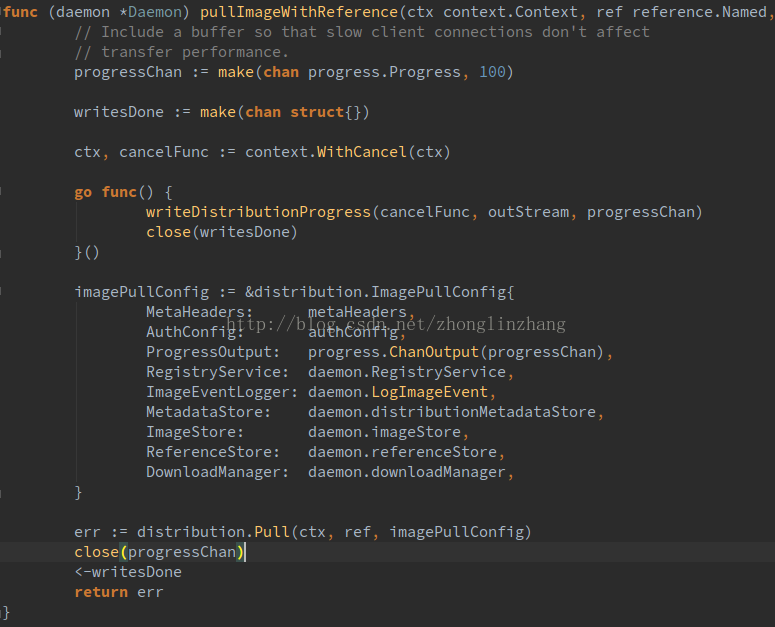
主要函数是daemon.pullImageWIthReference，client端验证这边也验证一遍，不知道是否多余，可能更安全一些吧！继续分析

#### 1.8  **pullImageWithReference函数**

ImagePullConfig文件位于distribution/pull.go填充一些配置信息，包括一些接口方法，结构体如下所示：



主要函数是distribution.Pull， 初始化pull操作，镜像是repository名字，tag可以为空或者指定的。



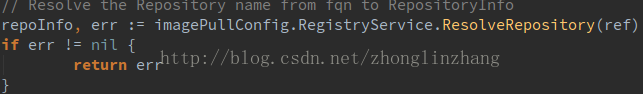
#### 1.9 **Pull函数**

概念： index用来管理用户账号，权限，搜索，打标签（tagging），以及其它可以方便的通过Web界面来完成的工作。

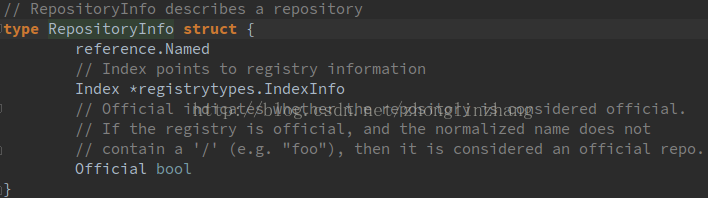
registry则是真正的用来保存、提供镜像文件的概念。它通过索引（index）来完成用户认证步骤。

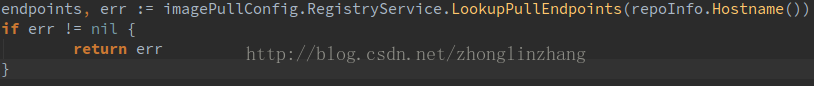
当执行docker search命令时，Docker会到index里查找，而不是registry。实际上Docker会在index所知道的多个registry中进行查找。

文件位于distribution/pull.go中，



拆分reference.Named，配置成RepositoryInfo（描述repository）结构体如下所示，





返回APIEndpoint列表pull的endpoints

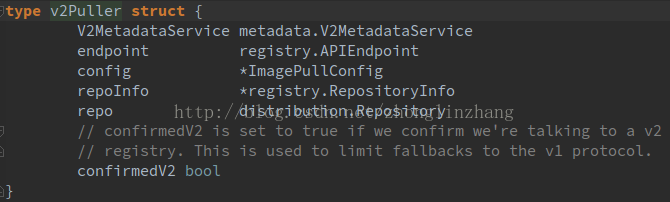
**[plain]** [view plain](http://blog.csdn.net/zhonglinzhang/article/details/53484799) [copy](http://blog.csdn.net/zhonglinzhang/article/details/53484799)

[print?](http://blog.csdn.net/zhonglinzhang/article/details/53484799)

1. for \_, endpoint := range endpoints {
2. if confirmedV2 && endpoint.Version == registry.APIVersion1 {
3. logrus.Debugf("Skipping v1 endpoint %s because v2 registry was detected", endpoint.URL)
4. continue
5. }
7. if endpoint.URL.Scheme != "https" {
8. if \_, confirmedTLS := confirmedTLSRegistries[endpoint.URL.Host]; confirmedTLS {
9. logrus.Debugf("Skipping non-TLS endpoint %s for host/port that appears to use TLS", endpoint.URL)
10. continue
11. }
12. }
14. logrus.Debugf("Trying to pull %s from %s %s", repoInfo.Name(), endpoint.URL, endpoint.Version)
16. puller, err := newPuller(endpoint, repoInfo, imagePullConfig)
17. if err != nil {
18. lastErr = err
19. continue
20. }
21. if err := puller.Pull(ctx, ref); err != nil {
22. // Was this pull cancelled? If so, don't try to fall
23. // back.
24. fallback := false
25. select {
26. case <-ctx.Done():
27. default:
28. if fallbackErr, ok := err.(fallbackError); ok {
29. fallback = true
30. confirmedV2 = confirmedV2 || fallbackErr.confirmedV2
31. if fallbackErr.transportOK && endpoint.URL.Scheme == "https" {
32. confirmedTLSRegistries[endpoint.URL.Host] = struct{}{}
33. }
34. err = fallbackErr.err
35. }
36. }
37. if fallback {
38. if \_, ok := err.(ErrNoSupport); !ok {
39. // Because we found an error that's not ErrNoSupport, discard all subsequent ErrNoSupport errors.
40. discardNoSupportErrors = true
41. // append subsequent errors
42. lastErr = err
43. } else if !discardNoSupportErrors {
44. // Save the ErrNoSupport error, because it's either the first error or all encountered errors
45. // were also ErrNoSupport errors.
46. // append subsequent errors
47. lastErr = err
48. }
49. logrus.Errorf("Attempting next endpoint for pull after error: %v", err)
50. continue
51. }
52. logrus.Errorf("Not continuing with pull after error: %v", err)
53. return translatePullError(err, ref)
54. }
56. imagePullConfig.ImageEventLogger(ref.String(), repoInfo.Name(), "pull")
57. return nil
58. }

这段代码看似挺多，主要做了就是找到endpoints, 然后puller.Pull工作，出错了进行错误整理。

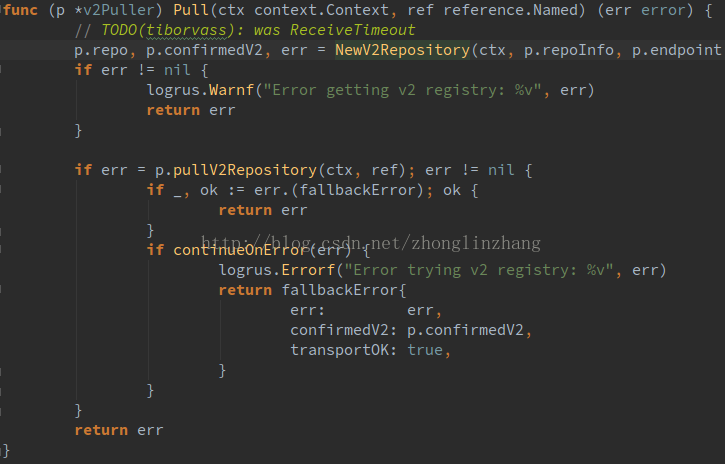
看看v2Puller结构体如下所示：



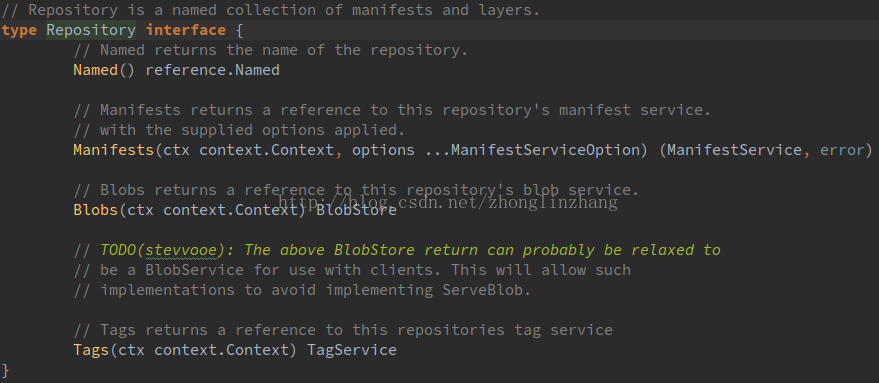
V2Puller结构体方法Pull方法下面进行讲解

#### 1.10 **Pull函数**

位于文件distribution/pull\_v2.go文件。



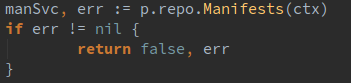
调用NewV2Repository函数来创建一个提供身份验证的http传输通道，并返回repository接口，如下所示：客户端的client中的registry实现了该接口，文件位于vendor/github.com/docker/distribution/registry/client/repository.go



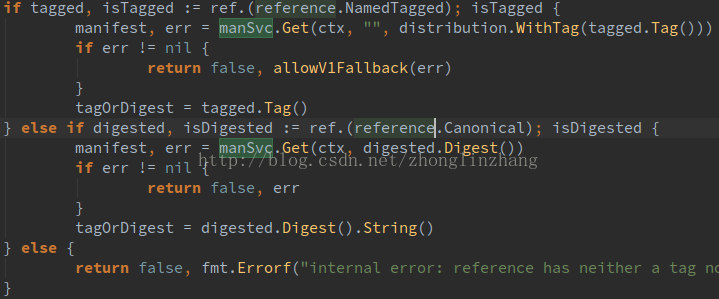
pullV2Repository函数主要方法是p.pullV2Tag函数，如果不是只有单独名字（比如只有ubuntu）直接使用pullV2Tag方法。

#### 1.11 **pullV2Tag函数**

位于文件distribution/pull\_v2.go文件中。前面都是配置以及验证工作，pullV2Tag才是真正的最后的下载环节，涉及的内容比较多。



获取首选项的mainfest服务



这一段就是根据tag或者数字摘要进行manSvc.Get方法，以指定的degest检索，并返回distribution.Mainfest接口。