如何用 Python 操作 Docker?

Python猫 1 week ago

The following article is from 游戏不存在 Author 肖恩顿

游戏不存在

每周一python项目，边读源码边晋阶

△点击上方“Python猫”关注 ，回复“1”领取电子书

作者：肖恩顿

来源：游戏不存在

docker-py是Docker SDK for Python。docker-py主要利用了requests，使用http/socket协议连接本地的docker engine进行操作。对 **docker** 感兴趣，苦于工作中只用到 **http** 协议的同学，都建议阅读一下本文。话不多说，一起了解docker-py的实现，本文分下面几个部分:

- docker-py项目结构
- docker-py API示例

- DockerClient的实现
- docker-version命令跟踪
- UnixHTTPAdapter的实现
- docker-ps命令跟踪
- docker-logs命令跟踪
- docker-exec 命令跟踪
- 小结
- 小技巧

docker-py项目结构

本次代码阅读，使用的版本是 `4.2.0`， 项目目录结构大概如下:

文件	描述
client.py	docker客户端的API
api	api相关目录
api/client.py	api的主要实现
api/container.py	container相关的api和client-mixin
api/daemon.py	daemon相关的api和client-mixin
models	下为各种对象模型，主要是单体及集合
models/resource.py	模型基类
models/containers.py	Container和ContainerCollection模型
transport	为客户端和服务端的交互协议
transport/unixconn.py	mac下主要使用了unix-sock实现

还有一些目录和类，因为不在这次介绍中，所以就没有罗列。

docker-py API示例

`docker-py` API上手非常简单:

```
import docker
client = docker.from_env()

result = client.version()
print(result)
# {'Platform': {'Name': 'Docker Engine - Community'},...}

client.containers.list()
# [<Container '45e6d2de7c54'>, <Container 'db18e4f20eaa'>, ...]

client.images.pull('nginx:1.10-alpine')
# <Image: 'nginx:1.10-alpine'>

client.images.list()
[<Image 'ubuntu'>, <Image 'nginx:1.10-alpine'>, ...]
```

上面示例展示了:

- 使用环境变量，创建client连接本地docker-engine服务
- 获取版本号，等同 `docker version`
- 获取正在运行的容器列表，等同 `docker container list`(别名是 `docker ps`)
- 拉取 **nginx:1.10-alpin** 镜像，等同 `docker image pull nginx:1.10-alpine`(别名是`docker pull nginx:1.10-alpine`)
- 获取镜像列表, 等同 `docker image list`

我们可以看到，docker-py的操作和docker的标准命令基本一致。

DockerClient的实现

DockerClient的构造函数和工厂方法展示docker-client对象包装了APIClient对象:

```
# client.py

class DockerClient(object):
    def __init__(self, *args, **kwargs):
        self.api = APIClient(*args, **kwargs)

    @classmethod
```

```
    def from_env(cls, **kwargs):
        timeout = kwargs.pop('timeout', DEFAULT_TIMEOUT_SECONDS)
        max_pool_size = kwargs.pop('max_pool_size', DEFAULT_MAX_POOL_SIZE)
        version = kwargs.pop('version', None)
        use_ssh_client = kwargs.pop('use_ssh_client', False)
        return cls(
            timeout=timeout,
            max_pool_size=max_pool_size,
            version=version,
            use_ssh_client=use_ssh_client,
            **kwargs_from_env(**kwargs)
        )
```

DockerClient 的 API 分 2 中，一种是属性方法，比如常用的 **containers**，**images**，**networks** 和 **volumes** 等子命令，因为要将返回值包装成对应模型对象:

```
@property
def containers(self):
    """
    An object for managing containers on the server. See the
    :doc:`containers documentation <containers>` for full details.
    """
    return ContainerCollection(client=self)

@property
def images(self):
    return ImageCollection(client=self)

@property
def networks(self):
    return NetworkCollection(client=self)

@property
def volumes(self):
    return VolumeCollection(client=self)

    ...
```

另一种是不需要模型包装，可以直接使用APIClient返回结果的 **info**, **version** 等方法:

```
    # Top-level methods
    def info(self, *args, **kwargs):
            return self.api.info(*args, **kwargs)
        info.__doc__ = APIClient.info.__doc__

    def version(self, *args, **kwargs):
            return self.api.version(*args, **kwargs)
        version.__doc__ = APIClient.version.__doc__

        ...
```

DockerClient类工厂方法的全局引用:

```
    from_env = DockerClient.from_env
```

docker-version命令跟踪

我们先从简单的 `docker version` 命令跟踪查看APIClient如何工作的。APIClient的构造函数:

```
    # api/client.py

    import requests

    class APIClient(
            requests.Session,
            BuildApiMixin,
            ConfigApiMixin,
            ContainerApiMixin,
            DaemonApiMixin,
            ExecApiMixin,
            ImageApiMixin,
            NetworkApiMixin,
            PluginApiMixin,
            SecretApiMixin,
            ServiceApiMixin,
            SwarmApiMixin,
            VolumeApiMixin):

        def __init__(self, base_url=None, version=None,
                     timeout=DEFAULT_TIMEOUT_SECONDS, tls=False,
```

```
            user_agent=DEFAULT_USER_AGENT, num_pools=None,
            credstore_env=None, use_ssh_client=False,
            max_pool_size=DEFAULT_MAX_POOL_SIZE):
        super(APIClient, self).__init__()

        base_url = utils.parse_host(
            base_url, IS_WINDOWS_PLATFORM, tls=bool(tls)
        )

        if base_url.startswith('http+unix://'):
            self._custom_adapter = UnixHTTPAdapter(
                base_url, timeout, pool_connections=num_pools,
                max_pool_size=max_pool_size
            )
            self.mount('http+docker://', self._custom_adapter)
            self._unmount('http://', 'https://')
            # host part of URL should be unused, but is resolved by requests
            # module in proxy_bypass_macosx_sysconf()
            self.base_url = 'http+docker://localhost'
```

上面代码可见:

- APIClient继承自 **requests.Session**
- APIClient使用Mixin方式组合了多个API，比如ContainerApiMixin提供container的api操作;NetWorkApiMixin提供network的api操作
- 使用mount方法加载不同协议的适配器adapter， unix系的docker是unix-socket;windows则是npipe

> 关于requests的使用，可以参看之前的博文 requests 源码阅读

默认的服务URL实现:

```
DEFAULT_UNIX_SOCKET = "http+unix:///var/run/docker.sock"
DEFAULT_NPIPE = 'npipe:////./pipe/docker_engine'

def parse_host(addr, is_win32=False, tls=False):
    path = ''
    port = None
    host = None
```

```
    # Sensible defaults
    if not addr and is_win32:
        return DEFAULT_NPIPE
    if not addr or addr.strip() == 'unix://':
        return DEFAULT_UNIX_SOCKET
```

`version` 请求在 **DaemonApiMixin** 中实现:

```
class DaemonApiMixin(object):


    def version(self, api_version=True):
        url = self._url("/version", versioned_api=api_version)
        return self._result(self._get(url), json=True)
```

底层的请求和响应在主类APIClient中提供:

```
class APIClient

    def _url(self, pathfmt, *args, **kwargs):
        ...
        return '{0}{1}'.format(self.base_url, pathfmt.format(*args))

    @update_headers
    def _get(self, url, **kwargs):
        return self.get(url, **self._set_request_timeout(kwargs))

    def _result(self, response, json=False, binary=False):
        assert not (json and binary)
        self._raise_for_status(response)

        if json:
            return response.json()
        if binary:
            return response.content
        return response.text
```

get和result，response都是requests提供。get发送请求，response.json将请求格式化成json后返回。

UnixHTTPAdapter的实现

`/var/run/docker.sock` 是 Docker 守护程序侦听的 UNIX 套接字，其连接使用UnixHTTPAdapter处理:

```
# transport/unixconn.py

import requests.adapters

RecentlyUsedContainer = urllib3._collections.RecentlyUsedContainer

class UnixHTTPAdapter(BaseHTTPAdapter):
    def __init__(self, socket_url, timeout=60,
                 pool_connections=constants.DEFAULT_NUM_POOLS,
                 max_pool_size=constants.DEFAULT_MAX_POOL_SIZE):
        socket_path = socket_url.replace('http+unix://', '')
        if not socket_path.startswith('/'):
            socket_path = '/' + socket_path
        self.socket_path = socket_path
        self.timeout = timeout
        self.max_pool_size = max_pool_size
        self.pools = RecentlyUsedContainer(
            pool_connections, dispose_func=lambda p: p.close()
        )
        super(UnixHTTPAdapter, self).__init__()

    def get_connection(self, url, proxies=None):
        with self.pools.lock:
            pool = self.pools.get(url)
            if pool:
                return pool

            pool = UnixHTTPConnectionPool(
                url, self.socket_path, self.timeout,
                maxsize=self.max_pool_size
            )
            self.pools[url] = pool

        return pool
```

UnixHTTPAdapter主要使用urllib3提供的链接池管理UnixHTTPConnection连接:

```
class UnixHTTPConnection(httplib.HTTPConnection, object):

    def __init__(self, base_url, unix_socket, timeout=60):
        super(UnixHTTPConnection, self).__init__(
            'localhost', timeout=timeout
        )
        self.base_url = base_url
        self.unix_socket = unix_socket
        self.timeout = timeout
        self.disable_buffering = False

    def connect(self):
        sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
        sock.settimeout(self.timeout)
        sock.connect(self.unix_socket)
        self.sock = sock

    def putheader(self, header, *values):
        super(UnixHTTPConnection, self).putheader(header, *values)
        if header == 'Connection' and 'Upgrade' in values:
            self.disable_buffering = True

    def response_class(self, sock, *args, **kwargs):
        if self.disable_buffering:
            kwargs['disable_buffering'] = True

        return UnixHTTPResponse(sock, *args, **kwargs)


class UnixHTTPConnectionPool(urllib3.connectionpool.HTTPConnectionPool):
    def __init__(self, base_url, socket_path, timeout=60, maxsize=10):
        super(UnixHTTPConnectionPool, self).__init__(
            'localhost', timeout=timeout, maxsize=maxsize
        )
        self.base_url = base_url
        self.socket_path = socket_path
        self.timeout = timeout

    def _new_conn(self):
        return UnixHTTPConnection(
            self.base_url, self.socket_path, self.timeout
        )
```

connect展示了socket类型是 `socket.AF_UNIX`, 这一部分的实现都非常基础 。

> 关于socket，可以参看之前的博文 python http 源码阅读

docker-ps命令跟踪

接着我们跟踪稍微复杂点的命令 `client.containers.list()`， 也就是 `docker ps`。前面介绍了，**container** 会组装结果为数据模型，下面是模型的父类:

```
class Model(object):
    """
    A base class for representing a single object on the server.
    """
    id_attribute = 'Id'

    def __init__(self, attrs=None, client=None, collection=None):
        self.client = client
        # 集合
        self.collection = collection

        self.attrs = attrs
```

Model是单个模型抽象，Collection则是模型集合的抽象，使用集合的prepare_model构建各种对象:

```
class Collection(object):
    """
    A base class for representing all objects of a particular type on the
    server.
    """

    model = None

    def __init__(self, client=None):
        self.client = client

    ...
```

```
    def prepare_model(self, attrs):
        """
        Create a model from a set of attributes.
        """
        if isinstance(attrs, Model):
            attrs.client = self.client
            # 双向引用
            attrs.collection = self
            return attrs
        elif isinstance(attrs, dict):
            return self.model(attrs=attrs, client=self.client, collection=self)
        else:
            raise Exception("Can't create %s from %s" %
                            (self.model.__name__, attrs))
```

Container和ContainerCollection的实现

```
class Container(Model):
    pass

class ContainerCollection(Collection):
    model = Container

    def get(self, container_id):
        resp = self.client.api.inspect_container(container_id)
        return self.prepare_model(resp)

    def list(self, all=False, before=None, filters=None, limit=-1, since=None,
             sparse=False, ignore_removed=False):
        resp = self.client.api.containers(all=all, before=before,
                                          filters=filters, limit=limit,
                                          since=since)
        containers = []
        for r in resp:
            containers.append(self.get(r['Id']))
        return containers
```

其中list函数主要有下面几个步骤

- 使用api的containers接口得到resp，就是container-id列表
- 逐个循环使用api的inspect_container请求container的详细信息
- 将结果封装成Container对象
- 返回容器Container对象列表

api.containers和api.inspect_container在ContainerApiMixin中提供, 非常简单清晰:

```
class ContainerApiMixin(object):

    def containers(self, quiet=False, all=False, trunc=False, latest=False,
                   since=None, before=None, limit=-1, size=False,
                   filters=None):
        params = {
            'limit': 1 if latest else limit,
            'all': 1 if all else 0,
            'size': 1 if size else 0,
            'trunc_cmd': 1 if trunc else 0,
            'since': since,
            'before': before
        }
        if filters:
            params['filters'] = utils.convert_filters(filters)
        u = self._url("/containers/json")
        res = self._result(self._get(u, params=params), True)

        if quiet:
            return [{'Id': x['Id']} for x in res]
        if trunc:
            for x in res:
                x['Id'] = x['Id'][:12]
        return res

    @utils.check_resource('container')
    def inspect_container(self, container):
        return self._result(
            self._get(self._url("/containers/{0}/json", container)), True
        )
```

docker-logs命令跟踪

前面的命令都是request-response的模式，我们再看看不一样的，基于流的docker-logs命令。我们先启动一个容器:

```
docker run -d bfirsh/reticulate-splines
```

查看容器列表

```
# docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED          STATU
61709b0ed4b8   bfirsh/reticulate-splines   "/usr/local/bin/run.…"   22 seconds ago   Up 21
```

实时跟踪容器运行日志:

```
# docker logs -f 6170
Reticulating spline 1...
Reticulating spline 2...
....
```

可以看到**reticulate-splines**容器就是不停的打印行数数据。可以用下面的代码实现`docker logs`相同的功能:

```
logs = client.containers.get('61709b0ed4b8').logs(stream=True)
 try:
  while True:
   line = next(logs).decode("utf-8")
   print(line)
 except StopIteration:
  print(f'log stream ended for {container_name}')
```

代码执行结果和前面的类似:

```
# python sample.py
...
Reticulating spline 14...

Reticulating spline 15...
...
```

logs的实现中返回一个CancellableStream，而不是一个result，利用这个stream，就可以持续的读取输出:

```
# models/Container

def logs(self, **kwargs):
    return self.client.api.logs(self.id, **kwargs)

# api/continer
def logs(self, container, stdout=True, stderr=True, stream=False,
             timestamps=False, tail='all', since=None, follow=None,
             until=None):
    ...

    url = self._url("/containers/{0}/logs", container)
        res = self._get(url, params=params, stream=stream)
        output = self._get_result(container, stream, res)

        if stream:
            return CancellableStream(output, res)
        else:
            return output
```

比较特别的是下面对于stream的处理:

```
# api/client

def _multiplexed_response_stream_helper(self, response):
    """A generator of multiplexed data blocks coming from a response
    stream."""

    # Disable timeout on the underlying socket to prevent
```

```
        # Disable timeout on the underlying socket to prevent
        # Read timed out(s) for long running processes
        socket = self._get_raw_response_socket(response)
        self._disable_socket_timeout(socket)

        while True:
            header = response.raw.read(STREAM_HEADER_SIZE_BYTES)
            if not header:
                break
            _, length = struct.unpack('>BxxxL', header)
            if not length:
                continue
            data = response.raw.read(length)
            if not data:
                break
            yield data

    def _disable_socket_timeout(self, socket):
        sockets = [socket, getattr(socket, '_sock', None)]

        for s in sockets:
            if not hasattr(s, 'settimeout'):
                continue

            timeout = -1

            if hasattr(s, 'gettimeout'):
                timeout = s.gettimeout()

            # Don't change the timeout if it is already disabled.
            if timeout is None or timeout == 0.0:
                continue

            s.settimeout(None)
```

上面代码展示了:

- 流的读取方式是每次读取STREAM_HEADER_SIZE_BYTES长度的数据作为协议头
- 协议头结构体格式解压后得到后面的数据包长度
- 继续读取指定长度的数据包
- 重复执行上面的数据读取过程

- 流式读取的时候还需要关闭socket的超时机制，确保流一直保持，知道手动(ctl+c)关闭

而 **attach** 则是采用了websocket的实现, 因为我们一般推荐使用exec命令，所以这里简单了解即可:

```
def _attach_websocket(self, container, params=None):
    url = self._url("/containers/{0}/attach/ws", container)
    req = requests.Request("POST", url, params=self._attach_params(params))
    full_url = req.prepare().url
    full_url = full_url.replace("http://", "ws://", 1)
    full_url = full_url.replace("https://", "wss://", 1)
    return self._create_websocket_connection(full_url)

def _create_websocket_connection(self, url):
    return websocket.create_connection(url)
```

docker-exec 命令跟踪

docker-exec是我们的重头戏，因为除了可以直接获取docker是输出外，还可以和docker进行交互。先简单回顾一下exec的使用:

```
# docker exec -it 2075 ping www.weibo.cn
PING www.weibo.cn (123.125.22.241): 56 data bytes
64 bytes from 123.125.22.241: seq=0 ttl=37 time=6.797 ms
64 bytes from 123.125.22.241: seq=1 ttl=37 time=39.279 ms
64 bytes from 123.125.22.241: seq=2 ttl=37 time=29.635 ms
64 bytes from 123.125.22.241: seq=3 ttl=37 time=27.737 ms
```

上面示例可以用下面代码完全模拟:

```
result = client.containers.get("2075").exec_run("ping www.weibo.cn", tty=True, stream=True
try:
 while True:
  line = next(result[1]).decode("utf-8")
  print(line)
```

```
except StopIteration:
 print(f'exec stream ended for {container_name}')
```

使用tty伪装终端和容器进行交互，就是我们最常用的方式了:

```
# docker exec -it 2075 sh
/ # ls -la
total 64
drwxr-xr-x    1 root     root          4096 Mar 24 13:16 .
drwxr-xr-x    1 root     root          4096 Mar 24 13:16 ..
-rwxr-xr-x    1 root     root             0 Mar 24 13:16 .dockerenv
drwxr-xr-x    2 root     root          4096 Mar  3  2017 bin
drwxr-xr-x    5 root     root           340 Mar 24 13:16 dev
drwxr-xr-x    1 root     root          4096 Mar 24 13:16 etc
drwxr-xr-x    2 root     root          4096 Mar  3  2017 home
drwxr-xr-x    1 root     root          4096 Mar  3  2017 lib
lrwxrwxrwx    1 root     root            12 Mar  3  2017 linuxrc -> /bin/busybox
drwxr-xr-x    5 root     root          4096 Mar  3  2017 media
drwxr-xr-x    2 root     root          4096 Mar  3  2017 mnt
dr-xr-xr-x  156 root     root             0 Mar 24 13:16 proc
drwx------    1 root     root          4096 Mar 25 08:17 root
drwxr-xr-x    2 root     root          4096 Mar  3  2017 run
drwxr-xr-x    2 root     root          4096 Mar  3  2017 sbin
drwxr-xr-x    2 root     root          4096 Mar  3  2017 srv
dr-xr-xr-x   13 root     root             0 Mar 24 13:16 sys
drwxrwxrwt    1 root     root          4096 Mar  3  2017 tmp
drwxr-xr-x    1 root     root          4096 Mar  3  2017 usr
drwxr-xr-x    1 root     root          4096 Mar  3  2017 var
/ # exit
```

同样这个过程也可以使用docker-py实现:

```
_, socket = client.containers.get("2075").exec_run("sh", stdin=True, socket=True)
print(socket)
socket._sock.sendall(b"ls -la\n")
try:
 unknown_byte=socket._sock.recv(docker.constants.STREAM_HEADER_SIZE_BYTES)
 print(unknown_byte)

 buffer_size = 4096 # 4 KiB
 data = b''
```

```
    while True:
     part = socket._sock.recv(buffer_size)
     data += part
     if len(part) < buffer_size:
      # either 0 or end of data
      break
     print(data.decode("utf8"))

    except Exception:
     pass
    socket._sock.send(b"exit\n")
```

示例演示的过程是:

- 获取一个已经存在的容器**2075**
- 对容器执行exec命令，注意需要开启stdin和socket
- 向容器发送 `ls -lah` 展示目录列表
- 读区socket上的结果。(这里我们偷懒，没有解析头，直接硬取，这样不够健壮)
- 继续发送 `exit` 退出容器

程序的输出和上面使用命令方式完全一致，就不在张贴了。进入核心的exec_run函数的实现:

```
# model/containers

def exec_run(self, cmd, stdout=True, stderr=True, stdin=False, tty=False,
                privileged=False, user='', detach=False, stream=False,
                socket=False, environment=None, workdir=None, demux=False):
    resp = self.client.api.exec_create(
            self.id, cmd, stdout=stdout, stderr=stderr, stdin=stdin, tty=tty,
            privileged=privileged, user=user, environment=environment,
            workdir=workdir,
        )
    exec_output = self.client.api.exec_start(
        resp['Id'], detach=detach, tty=tty, stream=stream, socket=socket,
        demux=demux
    )
    if socket or stream:
        return ExecResult(None, exec_output)
```

主要使用API的exec_create和exec_start两个函数, 先看第一个exec_create函数:

```
# api/exec_api

def exec_create(self, container, cmd, stdout=True, stderr=True,
                    stdin=False, tty=False, privileged=False, user='',
                    environment=None, workdir=None, detach_keys=None):

    if isinstance(cmd, six.string_types):
        cmd = utils.split_command(cmd)

    if isinstance(environment, dict):
        environment = utils.utils.format_environment(environment)

    data = {
        'Container': container,
        'User': user,
        'Privileged': privileged,
        'Tty': tty,
        'AttachStdin': stdin,
        'AttachStdout': stdout,
        'AttachStderr': stderr,
        'Cmd': cmd,
        'Env': environment,
    }

    if detach_keys:
        data['detachKeys'] = detach_keys
    elif 'detachKeys' in self._general_configs:
        data['detachKeys'] = self._general_configs['detachKeys']

    url = self._url("/containers/{0}/exec", container)
    res = self._post_json(url, data=data)
    return self._result(res, True)
```

exec_create相对还是比较简单，就是post-json数据到 `/containers/{0}/exec` 接口。然后是exec_start函数:

```
def exec_start(self, exec_id, detach=False, tty=False, stream=False
```

```
    def exec_start(self, exec_id, detach=False, tty=False, stream=False,
                   socket=False, demux=False):

        # we want opened socket if socket == True

        data = {
            'Tty': tty,
            'Detach': detach
        }

        headers = {} if detach else {
            'Connection': 'Upgrade',
            'Upgrade': 'tcp'
        }

        res = self._post_json(
            self._url("/exec/{0}/start", exec_id),
            headers=headers,
            data=data,
            stream=True
        )
        if detach:
            return self._result(res)
        if socket:
            return self._get_raw_response_socket(res)
        return self._read_from_socket(res, stream, tty=tty, demux=demux)
```

exec_start是post-json到 `/exec/{0}/start` 接口，注意这个接口看起来不是到容器，而是到exec。然后如果socket参数是true则返回socket，可以进行写入；否则仅仅读取数据。

使用curl访问docker-api

docker-engine的REST-api也可以直接使用 **curl** 访问:

```
$ curl --unix-socket /var/run/docker.sock -H "Content-Type: application/json" \
  -d '{"Image": "alpine", "Cmd": ["echo", "hello world"]}' \
  -X POST http://localhost/v1.41/containers/create
{"Id":"1c6594faf5","Warnings":null}

$ curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/containers/1c6594
```

```
$ curl --unix-socket /var/run/docker.sock -X POST http://localhost/v1.41/containers/1c6594
{"StatusCode":0}

$ curl --unix-socket /var/run/docker.sock "http://localhost/v1.41/containers/1c6594faf5/lo
hello world
```

可以通过修改/etc/docker/daemon.json更改为http服务方式的api

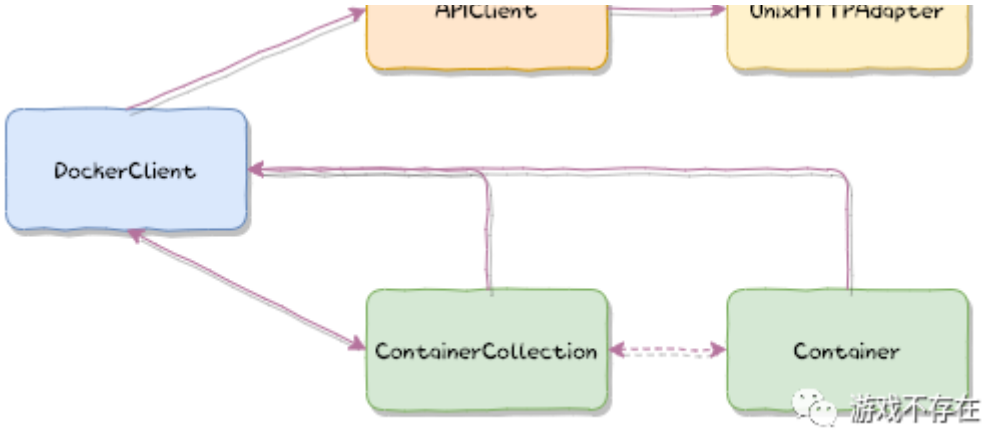
```
{
  "debug": true,
  "hosts": ["tcp://192.168.59.3:2376"]
}
```

然后 curl 命令可以直接访问docker的api

```
curl http://127.0.0.1:2375/info
curl http://127.0.0.1:2375/version
curl http://127.0.0.1:2375/images/json
curl http://127.0.0.1:2375/images/alpine/json
curl http://127.0.0.1:2375/containers/json
curl http://127.0.0.1:2375/containers/25c5805a06b6/json
```

小结

利用docker-py可以完全操作docker，这得益docker提供的REST-api操作。同时也发现requests的设计很强大，不仅仅可以用来做http请求，还可以用来做socket请求。学习docker-py后，相信大家对docker的理解一定有那么一点点加深，也希望下面这张图可以帮助你记忆:



API

小技巧

使用 check_resource 装饰器，对函数的参数进行预先处理:

```
def check_resource(resource_name):
    def decorator(f):
        @functools.wraps(f)
        def wrapped(self, resource_id=None, *args, **kwargs):
            if resource_id is None and kwargs.get(resource_name):
                resource_id = kwargs.pop(resource_name)
            if isinstance(resource_id, dict):
                resource_id = resource_id.get('Id', resource_id.get('ID'))
            if not resource_id:
                raise errors.NullResource(
                    'Resource ID was not provided'
                )
            return f(self, resource_id, *args, **kwargs)
        return wrapped
    return decorator
```

代码版本比较工具:

```
from distutils.version import StrictVersion


def compare_version(v1, v2):
    """Compare docker versions
```

```
    >>> v1 = '1.9'
    >>> v2 = '1.10'
    >>> compare_version(v1, v2)
    1
    >>> compare_version(v2, v1)
    -1
    >>> compare_version(v2, v2)
    0
    """
    s1 = StrictVersion(v1)
    s2 = StrictVersion(v2)
    if s1 == s2:
        return 0
    elif s1 > s2:
        return -1
    else:
        return 1


def version_lt(v1, v2):
    return compare_version(v1, v2) > 0


def version_gte(v1, v2):
    return not version_lt(v1, v2)
```

参考链接

- https://docs.docker.com/engine/api/sdk/examples/
- https://docker-py.readthedocs.io/en/stable/

Python猫技术交流群开放啦！群里既有国内一二线大厂在职员工，也有国内外高校在读学生，既有十多年码龄的编程老鸟，也有中小学刚刚入门的新人，学习氛围良好！想入群的同学，请在公号内回复『**交流群**』，获取猫哥的微信（谢绝广告党，非诚勿扰！）~

近期热门文章推荐：

Google 内部的 Python 代码风格指南

Python 有可能删除 GIL 吗?

重写 500 Lines or Less 项目 - A Simple Object Model

为了追求更快，CPU、内存、I/O都做了哪些努力?

一只喵星来客

一个有趣又有用的学习分享平台

Python进阶、Python哲学、Python翻译

兼具极客思维与人文情怀

欢迎你关注

Python猫

感谢创作者的好文❤️

喜欢此内容的人还喜欢

看完这篇，Docker 你就入门了！

杰哥的IT之旅