PyConChina 2019

Asyncio 在云服务自动化测试中的应用

王斌鑫

目录 CONTENTS

1 场景和痛点

云服务自动化测试有哪些场景？

怎么做的？都有哪些痛点？

- **API 测试**

  API 参数校验、返回值、错误码

- **基于多个 API 的业务测试**

  如先创建实例，再挂载磁盘，验证是否挂载成功

- **编排模板测试**

  资源编排（ROS）和运维编排（OOS）各类模板业务逻辑测试

- **各产品测试框架自成一套**

  重复造轮子、产品间无法复用

- **执行效率低，云资源浪费多**

  上百个用例跑1-2个小时，期间创建数十甚至上百个资源

- **用例编写复杂**

  需要关心过多细节，导致用例冗长和不易理解

- **可靠性较低**

  用例报错了，可能是用例或框架有问题

2 如何设计

设计目标?

使用方式?

2 如何设计 – 目标

- **通用性**

  可适用于多款云产品，如 ECS、ROS、OOS 等

- **高效性**

  提升执行效率，减少云资源消耗

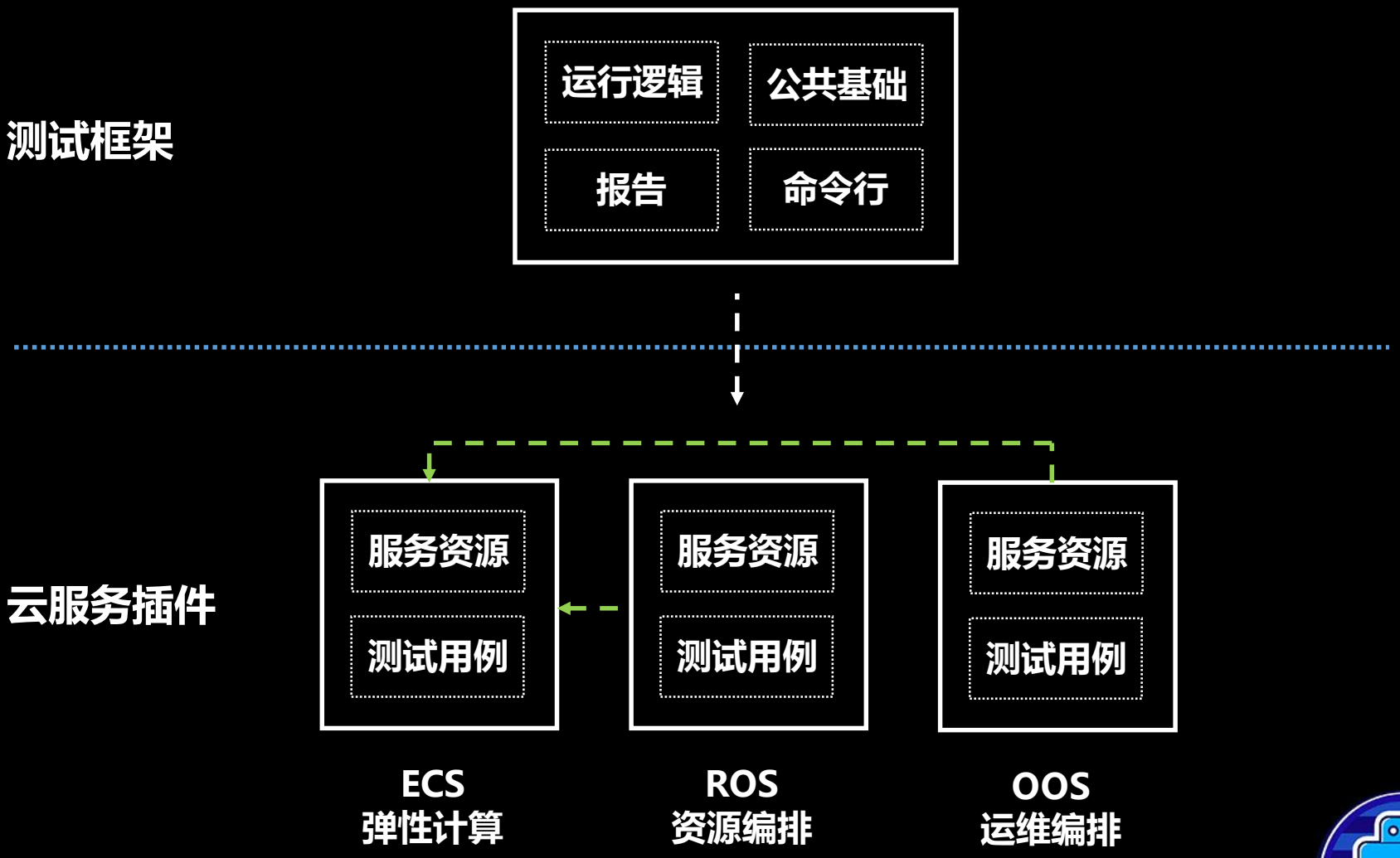
- **易用性**

  用例只需关注想要关注的点，易于编写和阅读

- **可靠性**

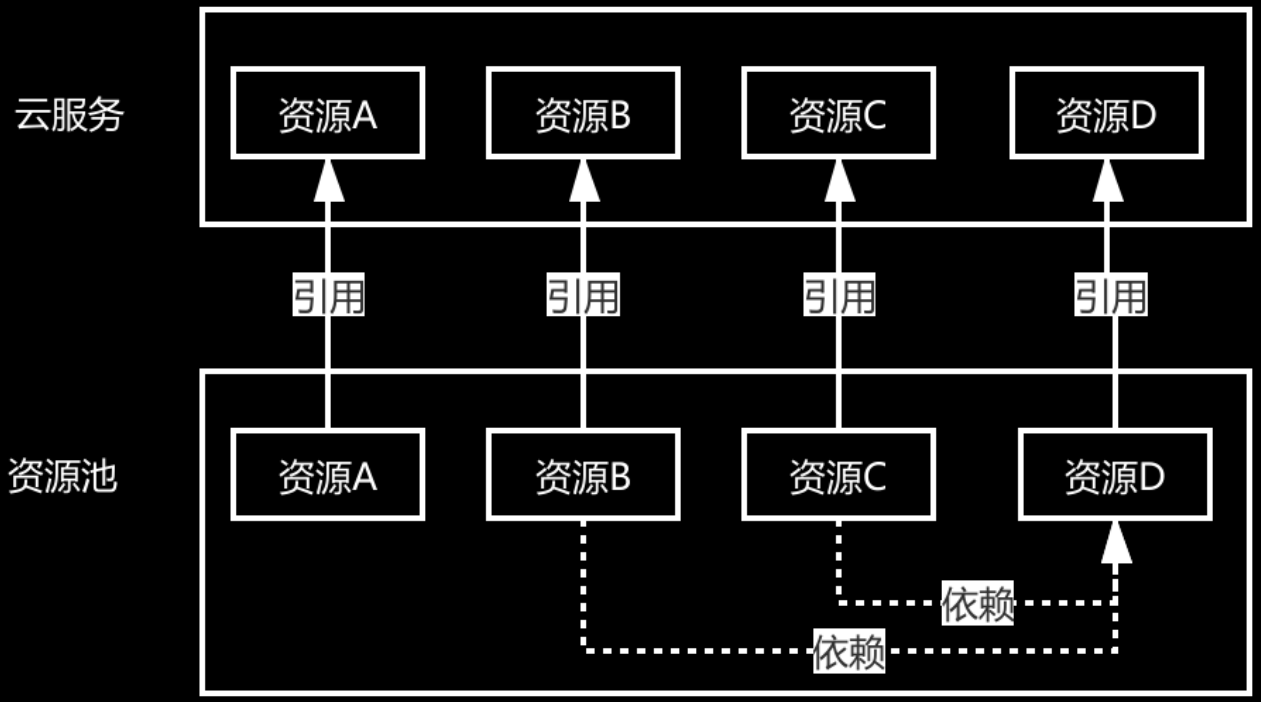
  用例运行结果可靠

2 如何设计 – 通用性



2 如何设计 – 高效性

- 异步 Asyncio
- 资源池



- 面向对象

```
instance = await Instance.create(
    region_id=region_id,
    zone_id=zone_id,
    wait=True
)

await instance.modify_charge_type(
    instance_charge_type='PrePaid',
    period=1,
    period_unit='Week'
)
```

- **模板化用例**

```
Tests:
- TestName: 模板化用例名称
  Description: 这是ECS创建实例API的测试用例
  Type: API
  API: RunInstances
  Parameters:
    RegionId: cn-hangzhou
    ZoneId: cn-hangzhou-b
  Response:
    Code: MissingParameter
    Message: The input parameter "VSwitchId"
      that is mandatory for processing
      this request is not supplied.
```

2 如何设计 – 可靠性

- **框架可靠性**

  防御式编程、类型注解、单元测试

- **用例可靠性**

  审查机制、资源锁、重试

3 基于 Asyncio 的实现

基于 asyncio 和 pytest 实现并行测试

3 基于 Asyncio 的实现 – 用例发现

```
def pytest_collect_file(path, parent):
    ext = path.ext
    if ext == ".py":
        return do_sth(path, parent)

    elif ext in ('.yaml', '.yml') and \
        path.basename.lower().startswith('test'):
        return YamlCaseFile(path, parent)


_pytest.python.pytest_collect_file = pytest_collect_file
```

3 基于 Asyncio 的实现 – 模板化用例

```
class YamlCaseFile(pytest.File):

    def collect(self):

        data = yaml.load(self.fspath.open())
        tests = data.get('Tests')

        do_check()

        for test in tests:
            yield YamlCaseItem.initialize(
                parent=self,
                spec=test
            )
```

3 基于 Asyncio 的实现 – 运行异步任务

```
class TestCases:

    def run(self):
        # 准备工作
        do_preparation()

        # 收集用例
        items = self.collect_items()

        # 获取测试函数
        test_funcs = self.get_test_funcs(items)
        test_futures = [test_func() for test_func in test_funcs]

        # 跑用例
        loop = asyncio.get_event_loop()
        loop.run_until_complete(asyncio.gather(*test_futures))

        # 统计结果
        do_stats()
```

3 基于 Asyncio 的实现 – 资源

```python
class BaseResource:
    def __new__(cls, *args, **kwargs):
        self = super().__new__(cls)
        self._lock = asyncio.Lock()
        return self

    async def __aenter__(self):
        await self.acquire()

    async def __aexit__(self, exc_type, exc, tb):
        self.release()

    async def acquire(self):
        await self._lock.acquire()

    def release(self):
        self._lock.release()

    def locked(self):
        return self._lock.locked()
```

3 基于 Asyncio 的实现 – 资源池

```
class ResourcePool:
    def __new__(cls, *args, **kwargs):
        res_pool = super().__new__(cls)
        res_pool._locks = {}
        return res_pool

    def __getattribute__(self, name):
        attr = super(ResourcePool, self).__getattribute__(name)

        if not inspect.iscoroutinefunction(attr):
            return attr

        async_func = to_auto_lock_release_func(attr)
        return async_func
```

3 基于 Asyncio 的实现 – 代码用例

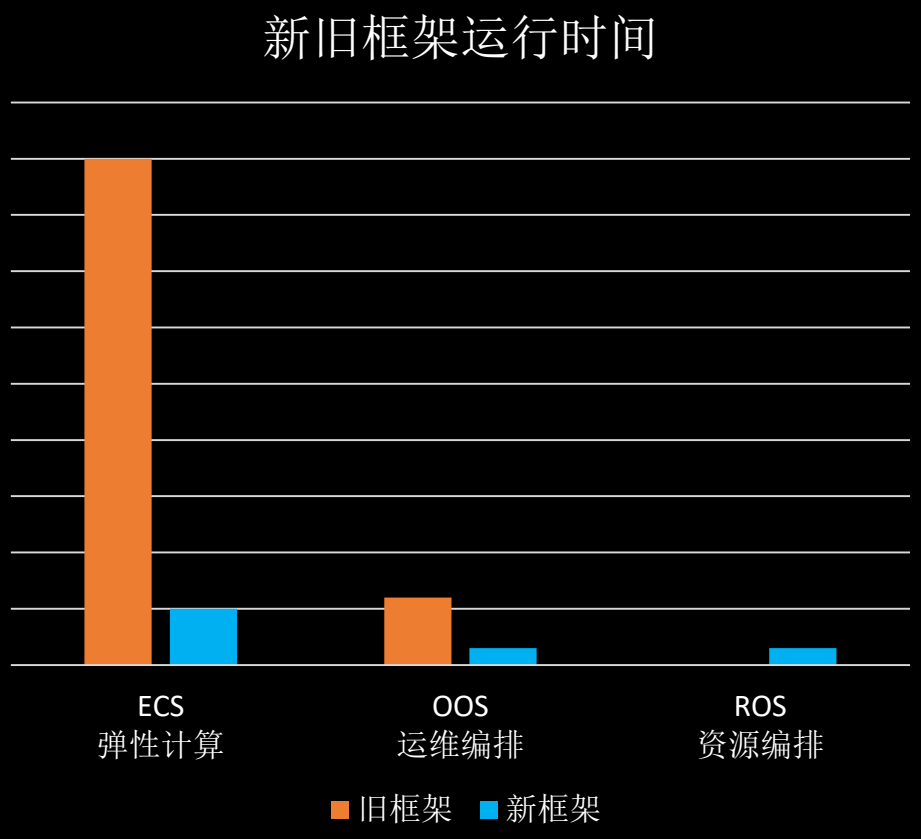
```python
async def test_modify_instance_name():
    """
    测试修改实例名称
    """
    instance = await respool.first_instance(
        status=InstanceStatus.RUNNING
    )

    async with instance:
        name = instance.gen_name()
        await instance.modify_attribute(instance_name=name)
        await instance.refresh()

        assert instance.instance_name == new_name
```

4 效果

可靠性、执行效率、开发效率

4 效果

- 可靠性提升至99%以上
- 执行速度提升3-10倍以上
- 资源开销减少50%以上
- 开发时间大大缩短，可维护性提高



THANK YOU

dreamlofter

知乎 prodesire

博客 prodesire.cn