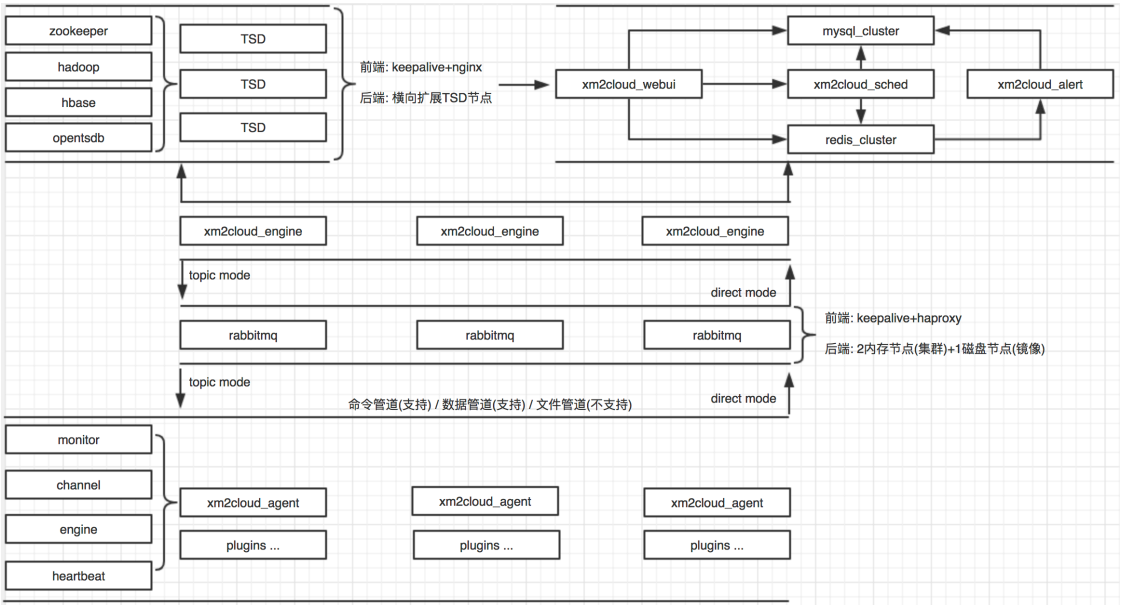
1. 平台基础架构



- xm2cloud是一个运维自动化常规基础架构,意在用PYTHON还原大厂底层自动化基础架构组件,所有组件支持分布式高可用,目前已用于XM全球业务.

2. 平台组件简介

- xm2cloud_agent
  - monitor组件执行注册的监控插件生成事件写入wcache组件
  - channel组件读线程和写线程分别与MQ保持长连接,定期从wcache组件中读取事件发送到MQ队列/定期从MQ队列获取事件写入rcache组件
  - engine组件定期从rcache中读取事件分发给注册的事件处理器处理并实时将执行日志事件写入wcache组件
  - heartbeat组件定期生成特定的心跳消息到wcache组件
- xm2cloud_engine
  - channel组件读线程和写线程分别与MQ保持长连接,定期从wcache组件中读取事件发送到MQ队列/定期从MQ队列获取事件写入rcache组件
  - engine组件定期从rcache中读取事件分发给注册的事件处理器处理(心跳添加MQ接入地址写入Redis,日志写入Redis,监控写入Redis[一小时热数据]/OpenTSDB[全量数据])
  - heartbeat组件定期生成特定的心跳信息到rcache组件
- xm2cloud_webui
  - 用于UI界面的展示
  - 提供RESTAPI接口/SDK给第三方应用接入
- xm2cloud_sched
  - 以mixins形式(ExecuteScriptMixin/UpdateUserDataMixin/AnalyseTriggerMixin)无缝侵入Django/Restframework视图/信号,主要用于分布式任务调度
- xm2cloud_alert
  - 主要用于消息通知,插件式扩展,通知一切你想通知的内容给目标(邮箱/短信/电话/微信/钉钉/第三方应用/穿戴设备...),无缝接入xm2cloud_sched的AnalyseTriggerMixin引擎

3. 为何要造轮子

- 作为运维/开发运维,不推荐造轮子但一定要有造轮子的能力(二次开发能力)

4. 常规开发套路

- 并发执行预定义插件列表:
  - 支持执行一切基于SHELL的命令返回指定格式内容作为监控的单/多个指标和值(例如Zabbix/tcollector/Nagios/Collectd...等等)
- 并发执行预注册插件列表:
  - 支持执行一切基于BaseCollector基类的插件类,按照插件编写规范很容易编写出支持任意指标监控的插件式监控系统

5. 基于元类实现

5.1. 元类简介

```
#! -*- coding: utf-8 -*-


class Persion(object):
    pass


if __name__ == '__main__':
    p = Persion()
    print type(p)
    print type(type(p))
     print type(type(type(p)))
---
<class '__main__.Persion'>
<type 'type'>
<type 'type'>
---
```

- PY里面一切皆对象,对象是由类实例化生成,类是有叫一个元类的东西生成的也就是type

```
#! -*- coding: utf-8 -*-


class Persion(type):
    def __new__(cls, *args, **kwargs):
        # klass = type(args[0], args[1], kwargs)
        klass = type.__new__(cls, *args, **kwargs)
        print '类型{0}在内存中被创建!'.format(klass.__name__)

        return klass


class Man(object):
    __metaclass__ = Persion


if __name__ == '__main__':
    pass
---
类型Man在内存中被创建!
---
```

- 默认所有类由type元类生成,通过type(name, bases, dict)生成其实底层是调用type.**new**(cls, *args, **kwargs)生成
- 一旦自定义类的__metaclass__静态变量被设置为一个继承自type的子类时,自定义类的创建过程将由对应的元类来接管,虽然最终依然通过type创建但可以灵活穿插自定义逻辑
- 自定义元类必须继承自type且重写__new__方法返回一个新类型,__new__接受三个参数(cls在创建过程中只是单纯占位,args包括要创建的类名以及继承的基类集合,kwargs包含创建出来的类将拥有的上下文,如属性和方法等)
- 自定义元类中的__new__可以通过super或直接type()或type.__new__创建新的类
- 神奇的现象,只要PY文件被载入内存(无需调用),那些被设置了__metaclass__的类都会去找对应的元类去生成类型,那么是不是可以把所有的监控插件类都继承一个Man这样的类让他们也变成拥有__metaclass__的类哪？这样的是不是就可以实现在启动时甚至运行时都能动态注册插件哪?

5.2. 实际应用

```python
#! -*- coding: utf-8 -*-


class BaseMetaClass(type):
    subclass = []

    def __new__(cls, name, bases, attrs):
        klass = type.__new__(cls, name, bases, attrs)

        if name == 'BaseCollector':
            return klass
        if klass.enable is False:
            return klass
        if klass in BaseMetaClass.subclass:
            return klass
        BaseMetaClass.subclass.append(klass)

        return klass


class BaseCollector(object):
    __metaclass__ = BaseMetaClass

    loader = None
    enable = True

    def __iter__(self):
        for klass in BaseMetaClass.subclass:
            yield klass

    def get_metricdata(self, *args, **kwargs):
        raise NotImplementedError

    def start_collects(self, *args, **kwargs):
        raise NotImplementedError

    @property
    def real_name(self):
        name = '{0}.{1}'.format(self.__module__, self.__class__.__name__)

        return name
```

- 如上定义一个所有插件的基类BaseCollector,设置元类为BaseMetaClass,它会记录创建的插件类方便主进程读取和调用
- BaseCollector作为所有插件的基类,可以设置一些静态变量来统一管理插件

```python
#! -*- coding: utf-8 -*-


import linux_metrics


from functools import partial
from agent.common.enhance import Switch
from agent.metrics.basemetric import BaseMetric
from agent.metrics.metric_data import MetricData
from agent.metrics.basecollect import BaseCollector


class Process(BaseMetric):
    def __init__(self, procs_running=None):
        self.procs_running = procs_running

    def get_procs_running(self):
        return self.procs_running

    def set_procs_running(self, procs_running):
        self.procs_running = procs_running

    def to_dict(self):
        data = {}
        if isinstance(self.get_procs_running(), MetricData):
            data['procs_running'] = self.get_procs_running().to_dict()

        return data

    def is_valid(self):
        return True


class Collector(BaseCollector):
    def get_metricdata(self, name):
        tags = {}
        for case in Switch(name):
            if case('procs_running'):
                name = 'procs.running'
                value = linux_metrics.procs_running()

                return MetricData(name, tags, value)
            if case():
                return None

    def start_collects(self):
```

```
        metrics = []

        data_func = partial(self.get_metricdata)

        procs_data = {
            'procs_running': data_func('procs_running'),
        }
        instance = Process(**procs_data)
        metrics.append(instance)

        return metrics
```

- 如上为一个内置的procs插件,Collector继承自BaseCollector,通过统一的start_collects方法收集,至于数据的提取方式以及来源这个大家可以自由发挥,重点是需要最终生成基于BaseMetric的Process对象即可,自由而规范这正是初期指导思想

```
tree _p2p_redis
---
_p2p_redis
├── __init__.py
├── _init.py
└── _init.yaml
---
#! -*- coding: utf-8 -*-


from functools import partial
from agent.common.loader import autodiscover_module


autodiscover_module(__name__, __file__)
# provide an automatic refresh interface
"""
example:
    user.autodiscover()
"""
#
autodiscover = partial(autodiscover_module, __name__, __file__)
```

- 特殊业务监控可能需要设计多个文件调用或多个目标主机交互,此插件系统提供autodiscover接口允许以插件包的形式编写插件并自动识别,只需要将如上代码加入到包的__init__.py文件中即可
- 其实原理很简单只需首次遍历递归导入载入内存时那些定义了元类的类就会自动注册上来,如果新增了插件可重新载入即可甚至无需重启xm2cloud_agent

5.3. 主要逻辑

```
#! -*- coding: utf-8 -*-


import os
import time


from agent import settings
from threadpool import makeRequests
from multiprocessing import Process
from agent.common.logger import Logger
from datetime import datetime, timedelta
from agent.metrics.baseloader import BaseLoader
from agent.handler.monitor import MonitorHandler
from agent.metrics.metric_data import MetricData
from agent.exceptions import GracefulExitException
from agent.metrics.basecollect import BaseCollector
from agent.handler.event.monitor import MonitorEventHandler
from agent.handler.channel.rabbitmq import RabbitMQChannelHandler


logger = Logger.get_logger(__name__)


class Monitor(Process):
    def __init__(self, gsignal, monitor_handler=None,
event_handler=None, channel_handler=None):
        super(Monitor, self).__init__()

        self._task_map = {}
        self._gsignal = gsignal
        self._event_handler = event_handler
        self._channel_handler = channel_handler
        self._monitor_handler = monitor_handler

    @property
```

```
    def monitor_handler(self):
        if isinstance(self._monitor_handler, MonitorHandler):
            return self._monitor_handler
        self._monitor_handler = MonitorHandler()

        return self._monitor_handler

    @property
    def channel_handler(self):
        if isinstance(self._channel_handler, RabbitMQChannelHandler):
            return self._channel_handler
        self._channel_handler = RabbitMQChannelHandler()

        return self._channel_handler

    @property
    def next_scheduled(self):
        next_time = datetime.now() +
timedelta(seconds=settings.MONITOR_SCHEDULER_INTERVAL)
        next_time_str = next_time.strftime('%Y-%m-%d %H:%M:%S')

        return next_time_str

    @property
    def event_handler(self):
        if isinstance(self._event_handler, MonitorEventHandler):
            return self._event_handler
        self._event_handler = MonitorEventHandler()

        return self._event_handler

    def event_get(self):
        events = []

        for klass in BaseCollector():
            if not isinstance(klass.loader, BaseLoader):
                args = ()
                kwargs = {}
            else:
                args = klass.loader.init_conf['args'] or ()
                kwargs = klass.loader.init_conf['kwargs'] or {}

            klass_ins = klass(*args, **kwargs)
            events.append(klass_ins)

        return events

    def event_put(self, task, res):
```

```
        self._task_map.pop(task.requestID)

        for metric in res:
            if not metric.is_valid():
                logger.warning('Invalid monitor event data:
{0}'.format(metric))
                continue
            logger.debug('Verified monitor event data:
{0}'.format(metric))
            for item in metric.to_dict().itervalues():
                edata = MetricData.from_dict(item).to_json()
                event = self.event_handler.create_event(edata)

self.channel_handler.wcache_handler.write(event.to_json())

    def run_destructor(self):
        pass

    def event_exp(self, task, err):
        _, exception, _ = err
        task_data = self._task_map.pop(task.requestID)

        event = task_data.get('event')
        logger.error('Call plugin {0} {1}'.format(event.real_name,
exception))

    def run(self):
        try:
            while not self._gsignal.is_set():
                events = self.event_get()
                if len(events) == 0:
                    logger.info('No events ready, next scheduled at
{0}'.format(self.next_scheduled))
                    time.sleep(settings.MONITOR_SCHEDULER_INTERVAL)
                    continue
                tasks = makeRequests(self.monitor_handler.run_executer,
events, self.event_put, self.event_exp)
                for i, task in enumerate(tasks):
                    self._task_map.update({task.requestID: {'event':
events[i]}})
                self.monitor_handler.feed(*tasks)
                logger.info('Events ready, next scheduled at %s',
self.next_scheduled)
                time.sleep(settings.MONITOR_SCHEDULER_INTERVAL)
            print 'Monitor process({0}) exit.'.format(os.getpid())
        except GracefulExitException:
            print 'Monitor process({0}) got
GracefulExitException.'.format(os.getpid())
```

```
            except Exception as e:
                print 'Monitor process({0}) got unexpected Exception
{1}'.format(os.getpid(), e)
            finally:
                self.run_destructor()
```

- 从上面可以看出很简单定期调用event_get将插件实例传递给线程池,等待线程异步执行完毕,结果通过回调event_put写入wcache再由channel写线程写入MQ队列即可

5.4. 其它组件

- 目前大家已经有足够的知识储备去开发一个插件式的监控Agent,但是至于数据如何传输,如何存储,如何展示,如何分析,如何告警,如何编写SDK给第三方应用,这个可能由于场景不同也有不同的零时/常规/现成的解决方案,这个下一期再细聊
- 如果对xm2cloud项目的其它组件/子组件感兴趣可参考源码 https://github.com/xm2cloud

6. 感谢

> 感谢大家对XM2CLOUD平台的关注,后期将持续给大家分享平台开发中的那些事儿～