运维架构系统之二

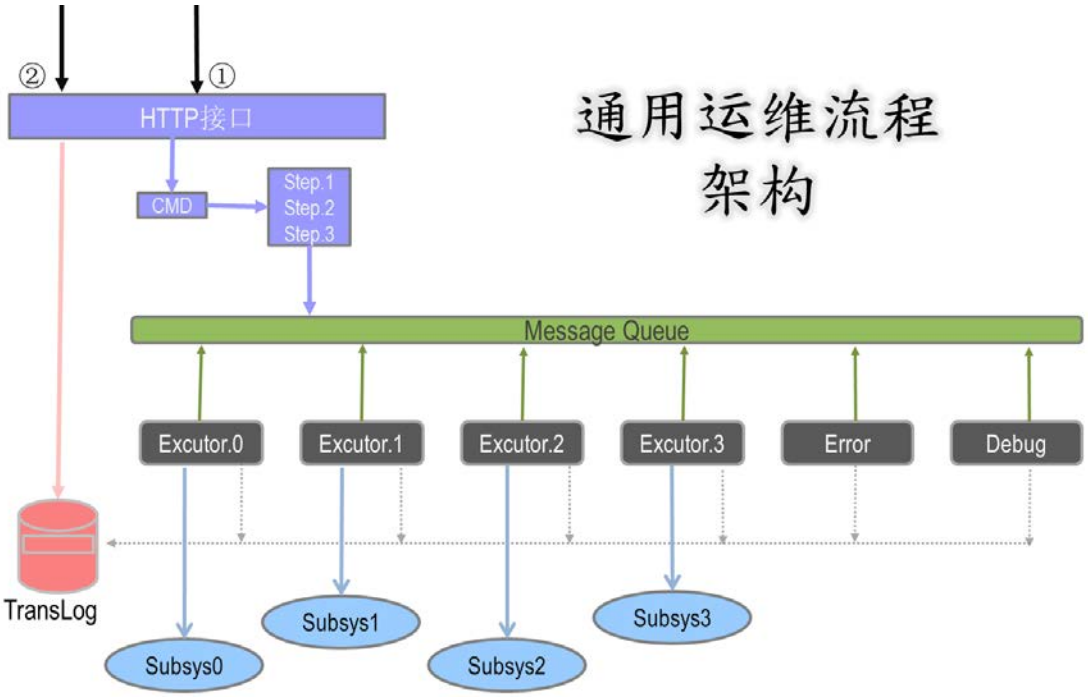
实体机管理运维子系统

By @codebox-腾讯

http://weibo.com/codebox

引子

先前介绍过一个运维流程架构的实践方案(http://weibo.com/1646958960/zf4hz9A0L)，那篇文章里介绍了一个大的骨架（见下图），主要是来驱动自动化流程，而本篇文章则是介绍其中一个非常得要的子系统——实体机管理运维子系统。



从整个运维体系来讲，一个非常大的难点就是如何管理数量众多的 RS（real server，即物理机）实例，包括如何做软件的下发和升级、高效地执行远程命令、保证环境一致性、及时发现故障等等。从单机角度来看，这些都是极其简单的事情，但是当机器数量变大，达到几万台的规模，这些简单事情的性质就变了。就像开一个包子铺很简单，夫妻店就行了，但如果要像 KFC 一样全球连锁，就是哈佛商学院级别的机构热衷研究的问题了。

这篇文章主要是介绍一下我们在这方面的一些实践。

运维系统特点与技术选型

作为运维系统来讲，最重要的一点是统一，面对众多可选择的方案，选择 A 或者 B 其实最终结果可能相差不大，最怕的是各种方案的混搭。payload 张三用 xml，李四用 json，王二麻子用 protobuf；接口有的用 http，有的要使用一个 so 库等等等等……

关于这个问题 @南非蜘蛛 在微博里吼过 “**运维自动化的基石是标准化，对于一个异构环境的标准化尤其重要，需要开发一些统一 api 来保证统一数据输出**”（http://weibo.com/1083599074/zcTNMtIPw）。

有必要再强调一下，对于运维系统来说，最重要的不是性能，而是统一性。只有统一了，才有可能做流程化、做自动化，说得烂大街一点就是云化。举个例子来讲，可能 A 方案好点，B 方案差点，但一般对于运维系统来说，选 A 或选 B 都行，就是不能同时存在。就像人一样，身高 175 或 180 都不错，但不要一条腿 175，另一条 180。

先总体说一下，我们对于基础技术方案的选择：

1. **语言**

   高效的开发效率，运行效率可以差点，所以选择适合快速应用开发(RAD)的 python 或 php，而不是 C/C++。人月神话讲到“每人每天代码行数是定数”，抽象层次高的语言定然开发效率高。不要惊奇，腾讯绝大部分的业务 CGI 都是 C/C++写的，运维系统也有不少是 C/C++写的。

2. **payload 协议**

可读性大于编解码效率，所以选字符串协议，而不是 protobuf 或 wup（腾讯自己开发的类 protobuf 工具）。字符串协议一般更倾向于 json，而不是 xml。

3. 接口

同样是面向字符串协议的 http 接口，而不是使用二进制 API。这样就不用给使用方提供各种语言的 API，也不用受升级之苦了。

4. 编码

没有第二选择，utf-8。

RS 管理整体架构

RS 管理大体可以分为以下几块：

1. RS Agent

   这是管理系统在一线的办事处，是跟 RS 机器直接打交道的，比如文件 copy，数据采集，监控上报等等。

2. 数据处理平台

   将 RS Agent 的相关数据汇总，用以监控告警、生成报表邮件、触发自动化流程等相关数据分析和决策功能。

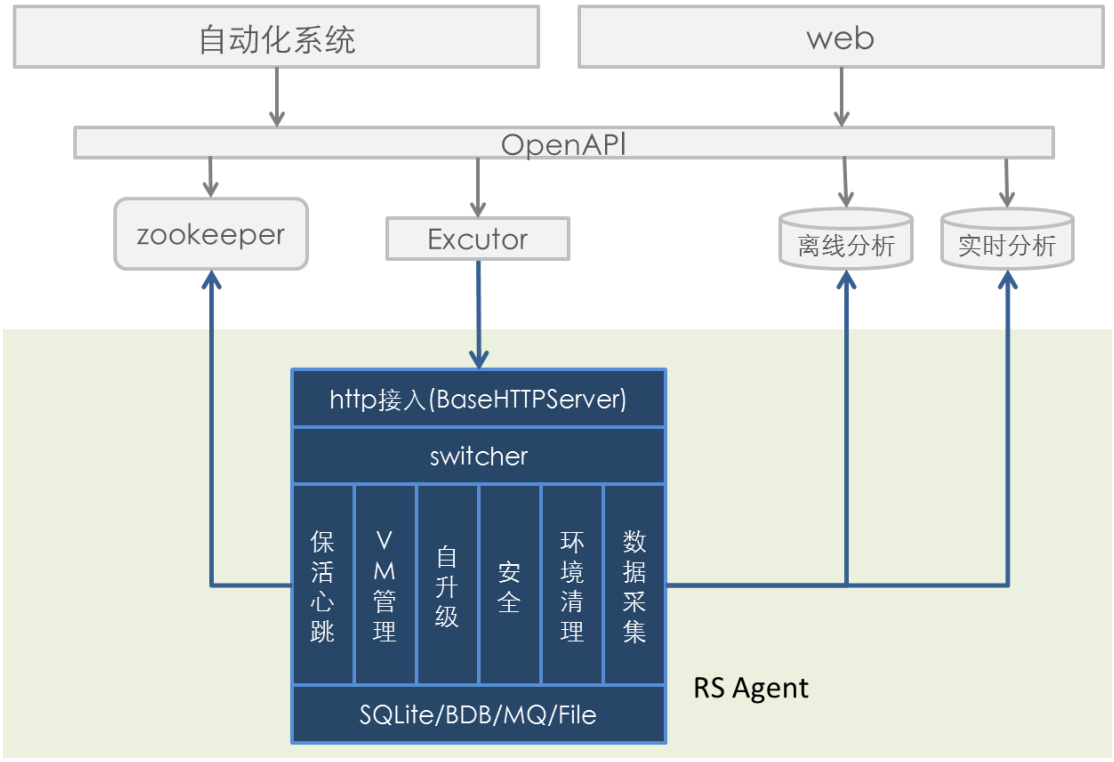
3. 控制管理台

   这是运维人员或开发人员做信息查询或现网变更的平台，比如 agent 版本升级、下发批量操作脚本等日常管理工作。

4. 自动化决策系统

这块的主要任务是接收监控与数据分析结果，发起特定的自动运维流程，例如业务扩缩容，域名绑定解绑，故障处理与恢复等等。整个运维架构统一之后，自动化的工作就非常容易做，否则绝对是一场永远没玩的噩梦。

三大块的整体结构如下图：



用到的相关组件特性：

1. Zookeeper 做分布式协调及 RS 之间的心跳保活，能及时发现 RS 故障。
2. Excutor 是之间分享过的通用流程架构里消息队列与具体子系统之间的衔接，为了突出 RS 管理子系统的重点，上图没有引入过多流程方面的元素，以免噪音过多。
3. 数据分析 根据数据特点及业务需求，不同的数据送到不同的平台上，主要分为离线分析和实时分析两类。离线分析主要用于报表之类的只读数据，实时

分析则会根据分析结果产生自动化流程，主要用于如路由权重调整，机器负载水位保持等。用到的相关组件有 hadoop、hypertable 还有一系列自研数据分析系统。

RS Agent 的设计

由于 RS agent 承担着所有跟具体的每台机器打交道的任务，所以为了方便开发和管理，将所有的功能垂直划分开来，按模块实现。这样 rsagent 就很自然地设计成了插件化的结构。

粗略的分一下，大概会有以下几个功能插件：

1. **心跳保持**

   用来告诉运维系统自己是死是活，是通过 zookeeper 的临时结点来实现。

2. **安全插件**

   用来记录和监控非法和危险操作，比如修改 root 密码，iptables 操作等。

3. **数据采集**

   用来采集各种机器指标（cpu、内存、磁盘、带宽）和应用产生的业务数据，并报到数据分析平台上。

4. **环境清理**

   保证整机环境的一致性和健康，如定期清理 log，各种内核参数，保证相关依赖等等。

5. **VM 管理**

如果本台机器上其它虚拟机的 HOST，还会有专门管理虚拟机的创建、销毁、起停、更改等基本运维操作。

6. 杂项

一些用于特定场景下的插件，如 oom 参数管理，保证关系管理进程不被杀死等等。

插件分为两种，一种是由外界 http 请求触发的，另一种是常驻线程，由时间或 RS 机器内部事件触发（如 file notify）。插件产生的临时数据会放在 Sqlite 等小型数据库或裸文件中，如果有需要，数据采集插件会将相应数据上传到数据分析平台。

整个 agent 用 python 实现，接口为 http+json，使用 python 自带的 BaseHTTPServer 实现基本 http server。Agent 收到请求后，会有 switcher 模块根据消息内容做请求分发，分发到相应的插件来处理。每个插件实际上就是一个 process 函数，当请求到来的时候框架会 fork 一个线程，然后调用相应用的 process 函数处理。

这里之所以选择线程是因为，如果用户分到一台空白机器后，ps 一看，一大堆管理进程在那儿的话，肯定很不爽，所以为了照顾用户的情绪，做成线程模式。

另外，这里是使用的 fork 模型，而不是 prefork 线程池，基于两方面的考虑 1. 线程池管理比较麻烦 2. 这里性能不会成为问题，管理命令也不可能 1 秒种几百次。

与在第一篇文章里介绍的设计原则类似，agent 的相关接口依然是要简单、幂等和无副作用，当出错时，调用方可以安全地使用重试操作，简化错误处理。为了避免 RS 机器因为特定临时状态（如重启、高负载）造成流程失败，一般不把相关请求直接下发到 agent，而是添加到 zookeeper 上去，然后反向通知到 rs 机器，做到最终一致性就 OK。

RS agent 的插件化的设计如下：

框架跟插件的接口是 init 和 handler，init 是在框架启动是挨个调用每个插件的 init，当外部有请求来时，框架负责调用相应的插件 handler。

框架启动调用 init 相关代码：

```
43      def do_module_init(self):
44          ' 调用所有handler模块的init函数 '
45          modules = set([i[1] for i in config.handlers])
46          for module in modules:
47              try:
48                  m = __import__(module)
49                  getattr(m, 'init')()
50                  log_info('Run %s.init ok', module)
51              except Exception, e:
52                  log_error('Run %s.init error! %s', module, str(e))
53              except:
54                  log_error('Run %s.init error!')
55
```

框架响应外部请求时，调用插件 handler 相关代码：

```
64 def call_module(command, operation, data):
65     ' 调用handler '
66     try:
67         module = get_module(command, operation)
68         m = __import__(module)
69     except ImportError, e:
70         #Yup, if you don't have a module, well, you don't have a module
71         log_debug('import module error: %s', str(e))
72         return paraerr_rsp
73     if config.handler_timeout > 0:
74         return timelimit(config.handler_timeout)(m.handler)(data)
75     else:
76         return m.handler(data)
```

由此可看，插件只需实现 init 接口和 handler 接口就可以了。以下是个示例插件

```
[root][~/Agent/src/xxoo]
-> ls
__init__.py  xxoo.py
```

__init__.py 文件是用来声明插件的接口函数 init 和 handler 对应的具体插件实现函数，如下所示

```
1 #!/usr/bin/env python
2
3 from xxoo import xxoo_handler as handler
4 from xxoo import check_time as init
5
```

结束语

大家可以看到，上面是一个比较整齐的方案。但是由于历史原因，生产环境已经存在大量的 agent 了，并且是由不同团队维护的，比如安全 agent 和虚拟机管理 agent。如果忽视这些历史问题和现状，只管闷头设计一下所谓完美的方案实现上是不可行的，不信看一下 IPv6 的现状。

下次会给大家介绍我们是如何管理已有的 agent，尽量规范化地运维它，提供统一有效的监控、资源限制、版本自升级等解决方案。