秘密▲

技 术 文 件

技术文件名称：Ceilometer 方案设计

技术文件编号：

版　　本：V1.0

拟　制　张绍满

审　核　吴应祥

会　签

标准化　李静

批　准　吴应祥

中兴通讯股份有限公司

修改记录

文件编号	版本号	拟制人/修改人	拟制/修改日期	更改理由	主要更改内容（写要点即可）
	V1.0	张绍满	2015-03-06	无	无
注 1：每次更改归档文件（指归档到事业部或公司档案室的文件）时，需填写此表。 注 2：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。					

目　　录

1 引言

1.1 编写目的

本文描述的对象是 Ceilometer 方案设计，通过对设计的思路、系统的构架、组成本系统的组件间的协作、组件的具体要求和组件间接口的详细描述，来满足上游需求，为集成测试设计、编写用户文档等工作提供依据。

1.2 预期的读者和阅读建议

本文档预期的读者和阅读建议见表 1.1。

表 1.1

读者分类	阅读重点	备注
系统工程师（负责系统需求开发）	设计原理、构架说明、协作说明	
系统工程师（负责开发软件需求）	构架说明、协作说明、组件说明、接口说明	
系统工程师（负责开发硬件需求）	构架说明、协作说明、组件说明、接口说明	
系统工程师（负责开发结构需求）	构架说明、协作说明、组件说明、接口说明	
工艺工程师	组件说明、接口说明	
系统工程师（负责软件概要设计）	构架说明、协作说明、组件说明、接口说明	
测试工程师	构架说明、协作说明、组件说明、接口说明	

1.3 文档约定

无

2 术语、定义和缩略语

2.1 术语、定义

本文使用的专用术语、定义见表 2.1。

表 2.1

术语/定义	含 义
OpenStack	OpenStack 是一个由 NASA（美国国家航空航天局）和 Rackspace

术语/定义	含　义
	合作研发并发起的，以 Apache 许可证授权的自由软件和开放源代码项目。OpenStack 是一个开源的云计算管理平台项目。
Ceilometer	OpenStack 计量组件

2.2 缩略语

本文使用的专用缩略语见表 2.2。

表 2.2

缩略语	原文	中文含义
IPMI	Intelligent Platform Management Interface	智能平台管理接口
REST	Representational State Transfer	表述性状态传递，一种 WEB 设计风格
SNMP	Simple Network Management Protocol	简单网络管理协议

3 设计依据

3.1 上游文档

本文涉及的设计依据见表 3.1。

表 3.1

文件编号	文件名称	版本号	说明

3.2 执行标准

无

4 概述

4.1 功能概述

Ceilometer 组件的目标是为计费系统提供所需的测量指标，以便于生成用户账单。相关计量项覆盖了所有当前 OpenStack 的核心组件，同时也会支持未来 OpenStack 的其它组件。

- Ceilometer 项目开始于 2012 年，最初用于一个简单目标：收集 OpenStack 项目的信息，计费系统使用该数据源生成费用账单，称之为“计量”。
- 随后，Ceilometer 收集的指标越来越多，社区开始给 Ceilometer 增加第二个目标：

成为一个标准的采集指标机制，而不管指标的用途。

- 最近，随着 Heat 项目的诞生，OpenStack 项目需要一个工具来观察关键变量，并触发不同的响应。因为 Ceilometer 已经收集了大量的指标，该工作顺理成章的成为了 Ceilometer 的扩展，称之为“告警”。

4.2 在系统中的位置

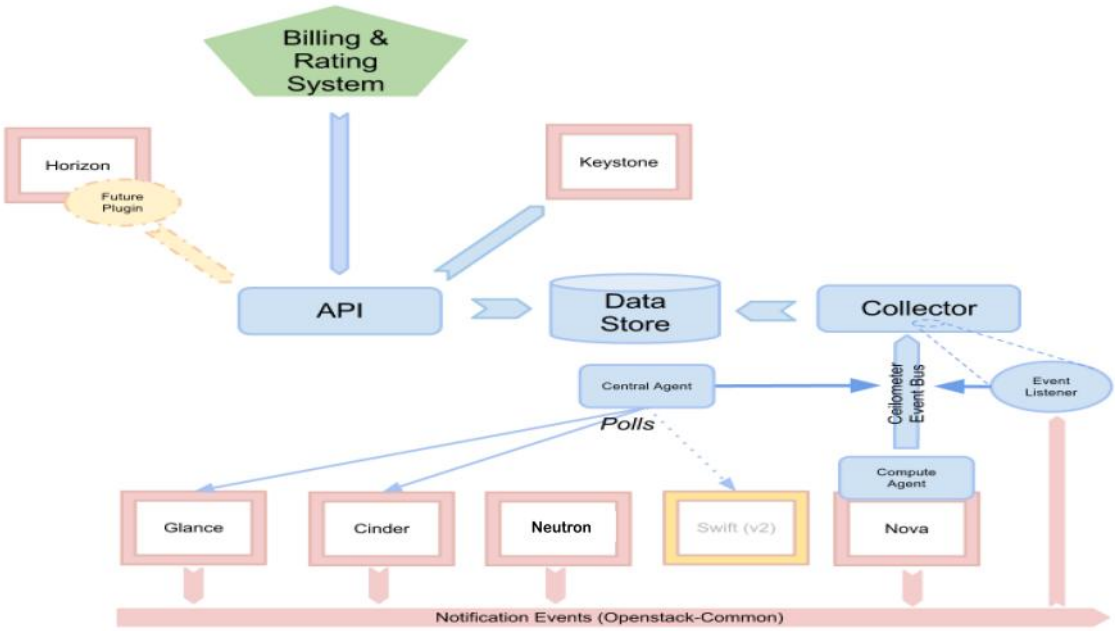

图 4.1 Ceilometer 在系统中的位置

图中蓝色框图即为 Ceilometer。与其它组件的接口包括：

- 提供接口给计费系统获取指标。
- Horizon 提供指标查询和显示。
- Nova、Glance、Cinder 等组件主动上报通知消息给 Ceilometer。
- Ceilometer 调用其它组件 API 接口获取指标信息。

4.3 相关系统功能简介

4.3.1 数据采集

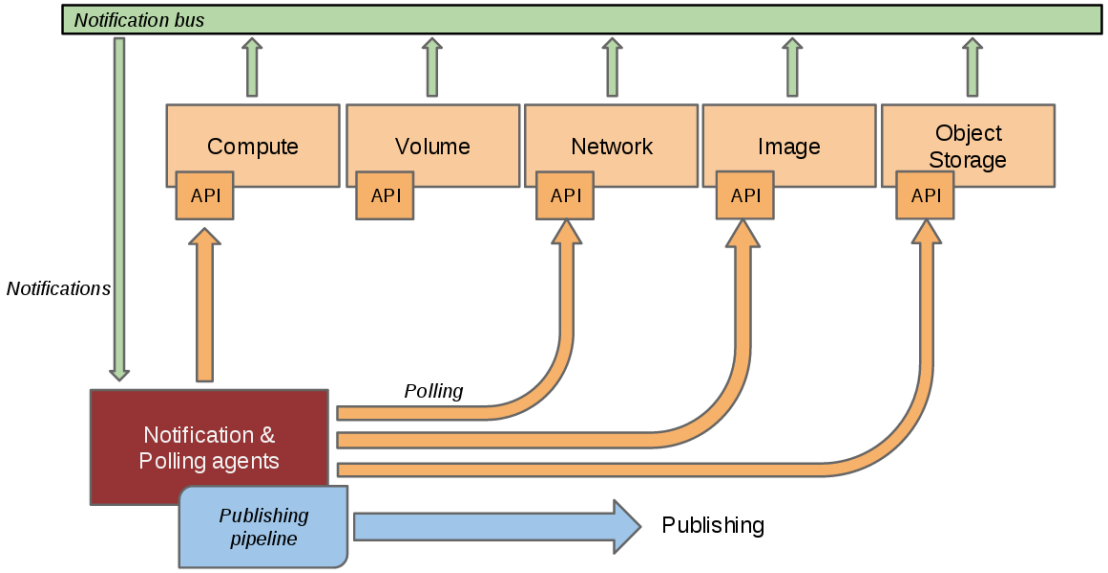

图 4.2 Ceilometer 数据采集

理想的情况下，每个测量的项目都应该向 Oslo 总线发送需要的事件。不幸的是，不是所有的项目都实现了这一点。Ceilometer 提供 2 种方式采集数据：

- 总线监听代理。从通知总线上获取事件，并转化为 Ceilometer 的指标。这个是数据采集的推荐方法。
- 轮询代理。这个是不太推荐的方法，定时通过 API 接口或者其它工具采集信息。如果存在能够通过通知机制获取同样数据的选项，那么轮询方式则不作为优选方案，因为会增加 API 服务的负荷。

第一种方法通过 ceilometer-notification 代理支持，其监控消息队列的通知消息。轮询代理可以配置为轮询本地 hypervisor 或者远程 APIs（服务 REST APIs 或者主机 SNMP/IPMI 守护进程）。

4.3.1.1 通知代理：监听数据

核心是通知守护进程（agent-notification），监控消息总线上其它 OpenStack 组件提供的数据，例如 Nova、Glance、Cinder、Neutron、Swift、Keystone 以及 Heat。

通知守护进程加载一个或多个监听插件，使用命名空间 ceilometer.notification。每个插件可以监听任何主题，但默认情况下只会监听 notification.info。监听程序抓取指定主题的消息，然后将其重新分配给合适的插件生成事件和采样指标。

采样为导向的插件提供一种方法列出它们感兴趣的事件类型和对应的消息处理回调，回调的注册名称用于通过通知守护进程的 pipeline 来使能或者禁用它。收到的消息在传递给回调之前，根据事件类型进行过滤，从而保证回调只会收到关注的事件。例如，一个回调请求

ceilometer.compute.notifications 下的 compute.instance.create.end 事件，它将会在 nova 交换上使用 notification.info 主题发生的这些事件时被调用，事件匹配也可以使用通配符，例如 compute.instance.*。

同样的，如果使能，通知也可以根据 event_type 进行过滤，并被转化为事件。

4.3.1.2 轮询代理：请求数据

计算资源的轮询通过运行在计算节点的轮询代理实现（与 hypervisor 的通信更加高效），通常称为 compute-agent。

非计算资源服务 API 的轮询由运行在云控制节点的代理实现，通常称为 central-agent。

在 all-in-one 的部署中一个单一的代理可以完成两种角色，反过来，一个代理可以部署多个实例，从而达到负载均衡。轮询代理守护进程可以配置为运行一个或多个轮询插件，使用 ceilometer.poll.compute 和/或者 ceilometer.poll.central 命名空间。

代理定期获取采样指标的计量值，轮询的频率通过 pipeline 进行配置，然后就采样值送给 pipeline 进行处理。

注意在 ceilometer.conf 中有一个可选的配置 shuffle_time_before_polling_task，通过设置大于零的整数使能代理随机开始轮询任务，从而在向 nova 或其它组件发送请求时增加一些随机的时间抖动，避免短时间内大量的请求。

4.3.2 数据处理

4.3.2.1 pipeline（管道）管理

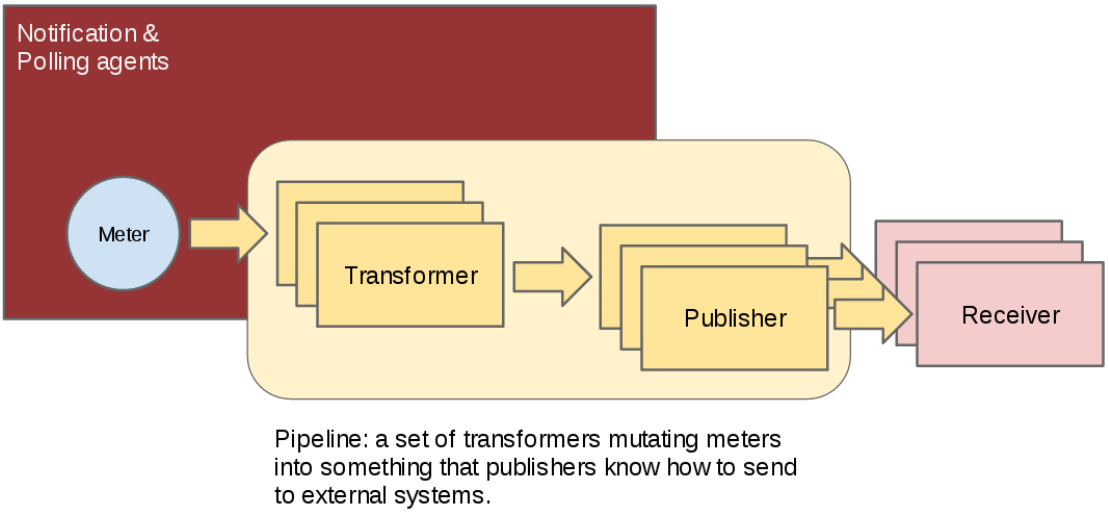

图 4.3 管道管理

Ceilometer 支持从代理采集数据，操作数据，并通过多个管道组合的方式进行发布。

4.3.2.2 数据转换

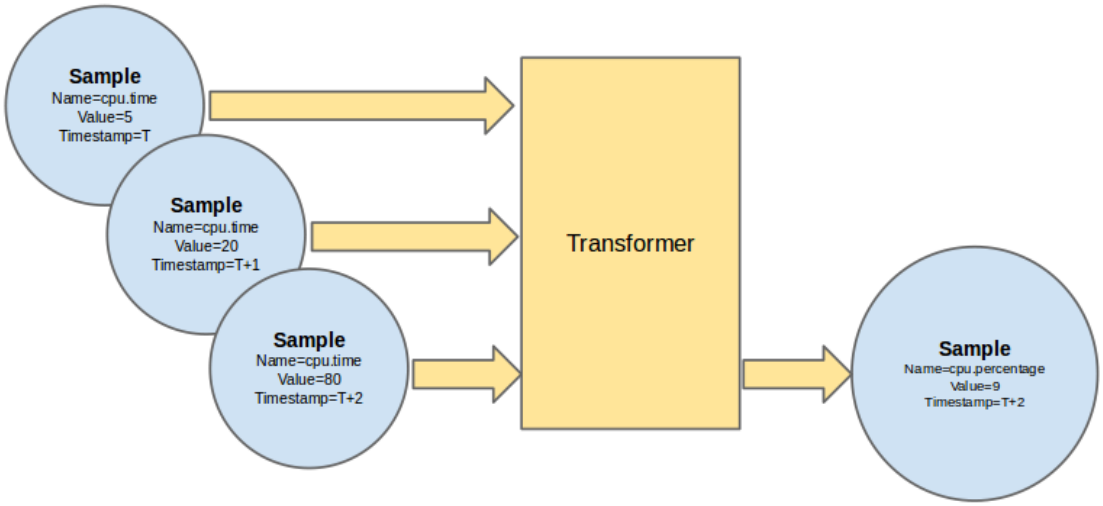

图 4.4 多个 CPU 使用时间的样本聚合为单一 CPU 使用率样本

Ceilometer 从通知代理和轮询代理中收集到了丰富的数据，如果基于历史或时间范围内进行组合，可以产生更多的数据。Ceilometer 提供各种转换器，能够在管道中对数据进行加工。

4.3.2.3 数据发布

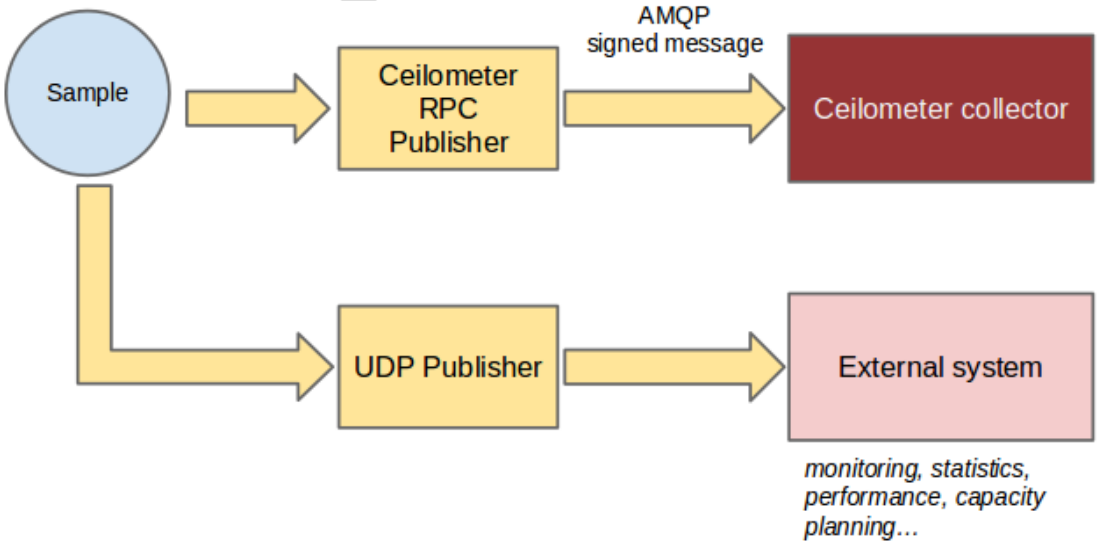

图 4.5 数据发布

目前，经处理的数据可以通过 4 种不同的方式发布：

- 通知，将样本推送到消息队列，可以被 collector 或外部系统使用。
- rpc，一个相对安全、同步的 RPC。
- udp，使用 udp 包发布样本。
- kafka，将数据发布到 Kafka 消息队列，可以被支持 Kafka 的任何系统使用。

4.3.3 数据存储

4.3.3.1 Collector 服务

Collector 守护进程收集从通知代理和轮询代理采集的时间和计量数据，判断数据合法性（签名是否有效），并将数据写入指定的目标：数据库、文件，或者 http。

4.3.3.2 支持的数据库

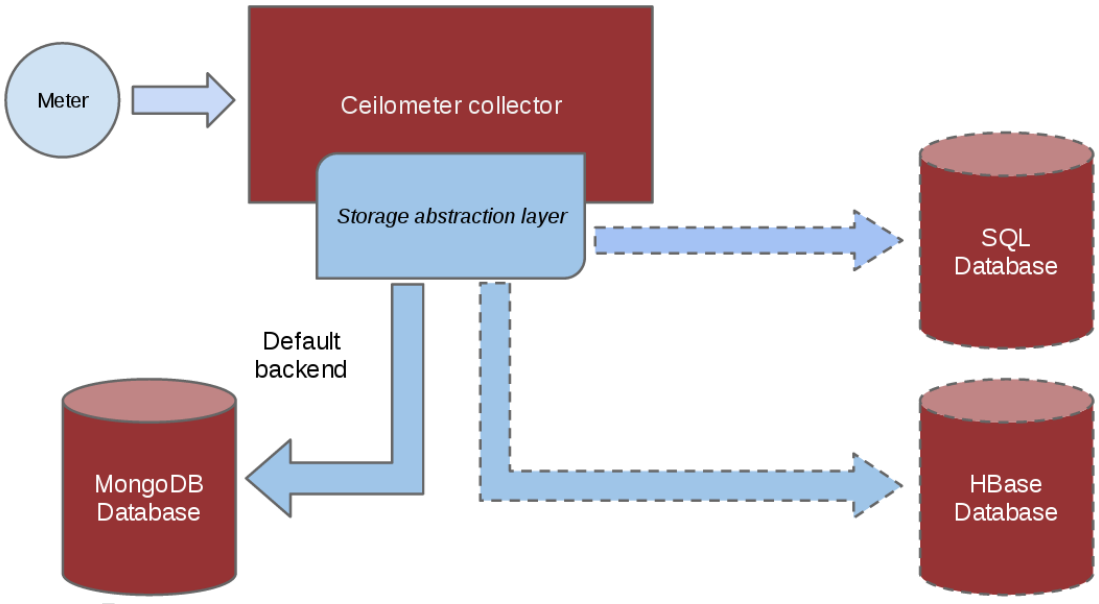

图 4.6 Ceilometer 存储模型

从 Ceilometer 项目开始的时候，就已经通过插件的方式支持使用各种后端数据库。

在 Juno 和 Kilo 版本发布周期，Ceilometer 的数据库被分成三个独立的连接：alarm、event，和 metering。部署人员可以选择将所有数据全部保存在一个独立的数据库，或者使用独立的专用数据库。例如，可以将 alarms 存储在 SQL 后端，将 events 和 metering 数据存储在 NoSQL 后端。

注意：Ceilometer 不保证不会改变 DB 设计模式，所以强烈建议通过 API 接口访问数据库，而不要直接查询。

4.3.4 数据访问

4.3.4.1 API 服务

当从轮询和通知代理获取的数据被存储在支持的数据库中，而数据库的设计有可能发生变化。因此 Ceilometer 提供了 REST API 接口来访问收集到的数据。

4.3.5 数据评估

4.3.5.1 Alarm 服务

Ceilometer 的告警模块，在 Havana 版本第一次发布，允许用户设置基于阈值的样本集合告警。告警可以针对单个指标，或者一个组合。例如，你可能希望一个指定的实例持续运行超过 10 分钟且内存使用率达到 70%时触发告警。为了设置一个告警，可以调用 Ceilometer 的 API 指定告警条件和采取的动作。

当然，如果你不是云的管理员，你可以只针对自己组件的指标设置告警。你也可以从虚拟机内部上报指标，这意味着可以根据应用数据触发告警。

目前实现了两种告警动作：

1. HTTP 回调：提供一个告警发生时的 URL 调用，请求的净荷中包含了告警触发的所有细节。
2. 日志：多用于调试，在日志文件中记录告警。

在实际的部署时，Alarm 数据库的设置将会是很重要的（使用一个不同于 metering 库的单独数据库）。

5 设计原理

5.1 系统构架的考虑

Ceilometer 按照功能划分为如下模块：

- Compute Agent：运行在每个计算节点，采集性能指标。
- Central Agent：运行在管理节点，调用 OpenStack 其它组件 api 采集指标。
- Notification Agent：运行在管理节点，接收其它组件上报的通知消息。
- Collector：运行在管理节点，基于 AMQP 接收消息，并记录到 Data Store。

- API：运行在管理节点，提供接口访问数据库。
- Alarm evaluator：运行在管理节点，进行指标的告警评估。
- Alarm notifier：运行在管理节点，进行告警通知。
- Mend：运行在管理节点，OpenCOS 扩展服务，用于历史数据清理。

5.2 动态特性的考虑

无

5.3 关键技术的考虑

5.3.1 Ceilometer 数据采集机制

5.3.1.1 基本概念

- meter 是 ceilometer 定义的监控项，诸如内存占用，网络 IO，磁盘 IO 等等。
- sample 是每个采集时间点上 meter 对应的值。
- statistics 一般是统计学上某个周期内，meter 对应的值(平均值之类)。
- resource 是被监控的资源对象，这个可以是一台虚拟机，一台物理机或者一块云硬盘。

在 Ceilometer 中有三种计量值：

- Cumulative：累计的，随着时间增长（如磁盘读写）。
- Gauge：计量单位，离散的项目（如浮动 IP，镜像上传）和波动的值（如对象存储数值）。
- Delta：增量，随着时间的改变而增加的值（如带宽变化）。

5.3.1.2 采集机制

ceilometer 的各个服务中，与采集相关的服务是 ceilometer-collector 、ceilometer-agent-central、ceilometer-agent-compute、ceilometer-agent-notification。我们可以通过下图了解一下他们之间的关系：

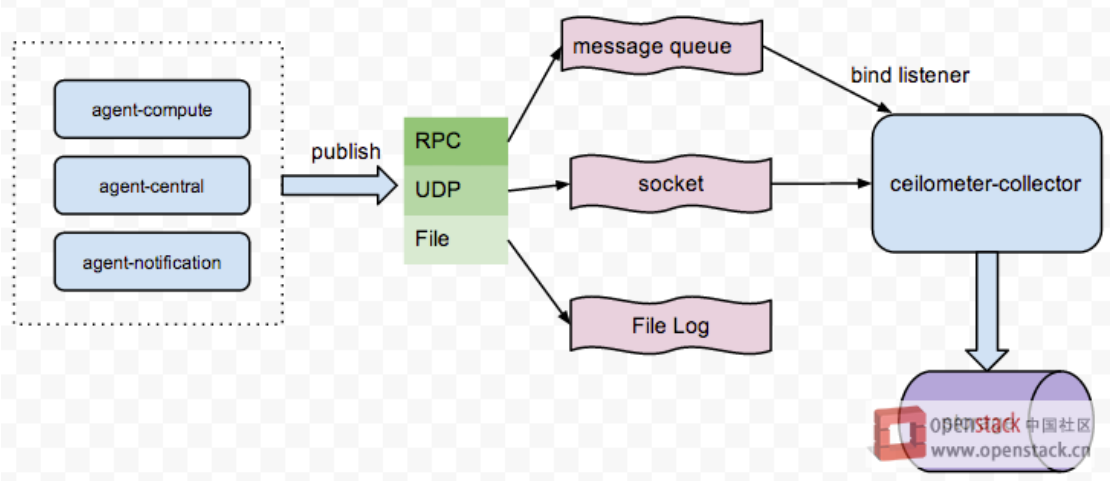

图 5.1 Ceilometer 数据上报

agent-*服务负责采集信息，采集的信息可以通过三种方式 publish 出来，包括 RPC、UDP、File。RPC 是将采集的信息以 payload 方式发布到消息队列，collector 服务通过监听对应的 queue 来收集这些信息，并保存到存储介质中；UDP 通过 socket 创建一个 UDP 数据通道，然后 collector 通过 bind 这个 socket 来接收数据，并保存到存储介质中；File 方式比较直接，就是将采集的数据以 filelog 的方式写入 log 文件中。

至于使用哪种方式 publish，那么就要看你的 pipline 文件是如何配置的了，具体可以查看/etc/ceilometer/pipline.yaml 中的 publishers 配置。

agent-*三个采集组件分别负责采集不同类型的信息，

- agent-notification 负责收集各个组件推送到 oslo-messaging 的消息，oslo-messaging 是 openstack 整体的消息队列框架，所有组件的消息队列都使用这个组件；
- agent-compute 只负责收集虚拟机的 CPU 内存 IO 等信息，所以他需要安装在 Hypervisor 机器上；
- agent-central 是通过各个组件 API 方式收集有用的信息；

agent-notification 只需监听 AMQP 中的 queue 即可收到信息，而 agent-compute 和 agent-central 都需要定期 Poll 轮询收集信息。看下图来了解一下：

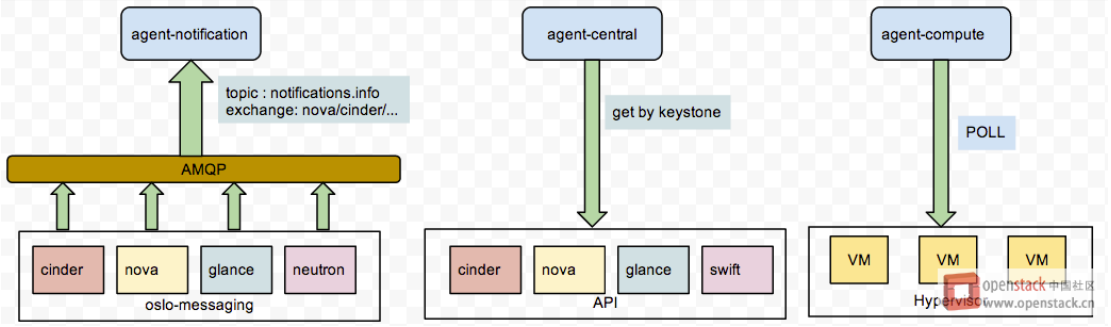

图 5.2 Ceilometer 数据采集

信息通过 agent-*采集并由 collector 汇总处理，最终需要持久化到存储介质中，ceilometer 目前支持的存储包括 mysql、DB2、HBase、mongoDB，从支持的数据库来看，监控数据持久化的压力还是相当大的。

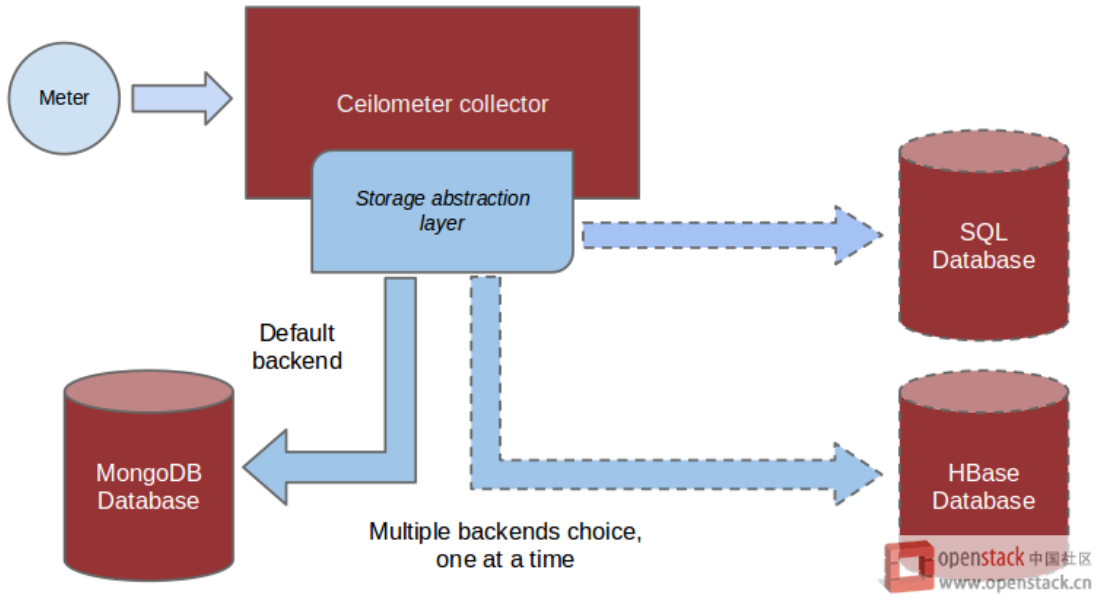

图 5.3 数据持久化存储

5.3.1.3 采集项

agent-*组件在启动时候，通过 stevedore 的插件机制来加载采集项，包括每个采集项对应的执行程序。stevedore 的插件配置是利用了 setuptools 的 entry_points，所以我们可以通过查看 entry_points 的配置信息，来确定有哪些采集项。如果你的程序打包完毕并发布到了 python 的搜索路径中，那么你需要查看 ceilometer 的 egg 文件来查看，或者你可以下载源码查看 setup.cfg 文件，相关信息如下：

```
[entry_points]
ceilometer.notification =
instance = ceilometer.compute.notifications.instance:Instance
instance_flavor = ceilometer.compute.notifications.instance:InstanceFlavor
memory = ceilometer.compute.notifications.instance:Memory
...
...

ceilometer.poll.compute =
```

```
disk.read.requests = ceilometer.compute.pollsters.disk:ReadRequestsPollster
cpu = ceilometer.compute.pollsters.cpu:CPUPollster
…
…

ceilometer.poll.central =
image = ceilometer.image.glance:ImagePollster
storage.containers.objects = ceilometer.objectstore.swift:ContainersObjectsPollster
…
…
```

ceilometer.notification 对应的是 agent-notification 组件，ceilometer.poll.compute 对应的是 agent-compute 组件，ceilometer.poll.central 对应的是 agent-central 组件。

5.3.1.4 采集 hardware 信息

ceilometer 除了可以收集 openstack 组件的相关信息，也可以收集诸如 kwapi、hardware、opendaylight 信息。kwapi 是采集物理机能耗信息的项目，agent-central 组件通过 kwapi 暴露的 api 来收集物理机的能耗信息；agent-central 也可以通过 snmp 协议直接收集 hardware 的 CPU、MEM、IO 等信息；opendaylight 是 SDN 解决方案的开源项目，opendaylight 规范中包括暴露一个 API 接口来提供 SDN 内部的一些信息，agent-central 正是通过这个 API 可以收集 opendaylight 组件的信息。

5.3.2 Ceilometer 告警机制

Ceilometer Alarm 是 H 版新添加的功能，监控报警是云平台不可缺少的部分，Ceilometer 已经实现了比较完善的监控体系，报警怎么能缺少呢？用过 AWS CloudWatch Alarm 的人应该不会对 Ceilometer 的 Alarm 感到陌生，Ceilometer 实现的 Alarm 和 CloudWatch 的 Alarm 很像，概念基本上都一样，Alarm 的逻辑也基本上一样，可以说是一个开源版的 CloudWatch Alarm，但是它进行了一些“微创新”，实现了一些比较有意思的小功能。

5.3.2.1 Alarm 功能

简单来说，Alarm 的功能其实很简单，监控某一个或多个指标的值，若高于或者是低于阈值，那么就执行相应的动作，比如发送邮件短信报警，或者是直接调用某个接口进行

autoscaling 操作，像 Heat 就是依赖 Ceilometer 的 Alarm 实现 Auto Scaling 的操作。

Ceilometer 中，实现了 2 种 alarm：一种是 threshold，一种是 combination。顾名思义，threshold 就是我们熟悉的根据监控指标的阈值去判断 alarm 的状态，它只是针对某一个监控指标建立 alarm，而 combination 则可以理解为 alarm 的 alarm，它是根据多个 alarm 的状态来判断自己的状态的，多个 alarm 之间是 or/and 的关系，这相当于是对多个监控指标建立了一个 alarm。一般情况下，我们只需要 threshold 类型的 alarm 就足够了，但是一些特殊情况，比如 Heat 要执行 auto scaling 操作，可能就要对多个监控指标进行衡量，然后再采取操作。

下面，我们来分析一下 Alarm 的 API，看它到底提供了哪些不一样的功能：

1. POST /v2/alarms

创建一个 alarm，详细的参数见下表：

参数	类型	解释
name	str	name 是 project 唯一的
description	str	描述
enabled	bool	alarm 的一个开关，可以停止/启动该 alarm，默认是 True
ok_actions	list	当 alarm 状态变为 ok 状态时，采取的动作，默认是[]
alarm_actions	list	当 alarm 状态变为 alarm 状态时，采取的动作，默认是[]
insufficient_data_actions	list	当 alarm 状态变为 insufficient data 状态时，采取的动作，默认是[]
repeat_actions	bool	当 alarm 被触发时，是否重复执行对应的动作，默认是 False
type	str	alarm 类型，目前有 threshold 和 combination 两种，必填
threshold_rule	AlarmThresholdRule	当 alarm 类型为 threshold 时，制定的 threshold 规则

参数	类型	解释
combination_rule	AlarmCombinationRule	当 alarm 类型为 combination 时，制定的 combination 规则
time_constraints	list(AlarmTimeConstraint)	约束该 alarm 在哪些时间段执行,默认是[]
state	str	alarm 的状态，默认是 insufficient data
user_id	str	user id，默认是 context user id
project_id	str	project id，默认是 context project id
timestamp	datetime	alarm 的定义最后一次被更新的时间
state_timestamp	datetime	alarm 的状态最后一次更改的时间

这里主要说下面几个参数：

- name：name 是 project 唯一的，在创建 alarm 的时候会检查
- enabled：这个功能比较人性化，可以暂停该 alarm，是微创新之一
- xxx_actions：定义了在该 alarm 状态由其他的状态变为 xxx 状态时，执行的动作
- repeat_actions：这个参数指定了是否要重复执行 action，比如第一次检查 alarm 已经超过阈值，执行了相应的 action 了，当下一次检查时如果该 alarm 还是超过阈值，那么这个参数决定了是否要重复执行相应的 action，这也是微创新之一
- threshold_rule：当 type 为 threshold 时，定义的 alarm 被触发的规则，详细参数见下面 AlarmThresholdRule 对象属性
- combination_rule：当 type 为 combination 时，定义的 alarm 被触发的规则，详细参数见下面 AlarmCombinationRule 对象属性
- time_constraints：这也是一个很人性化的参数，可以指定该 alarm 被检查的时间的一个列表，比如说我只想让这个 alarm 在每天晚上的 21 点到 23 点被检查，以及每天中午的 11 点到 13 点被检查，其它时间不检查该 alarm，这个参数就可以做这个限制，不过该参数设置稍微复杂一点，详细参数见下面 AlarmTimeConstraint 对象属性，默认是[]，即不设限制，随着 alarm 进程的 interval time 进行检查。
- state：alarm 总共有 3 个状态：OK、INSUFICIENT DATA、ALARM，这三个状态分别对应到上面的 xxx_actions

AlarmThresholdRule:

- meter_name: 监控指标
- query: 该参数一般用于找到监控指标下的某个资源，默认是[]
- period: 这个参数其实有两个作用，一个是确定了获取该监控指标的监控数据的时间范围，和下面的 evaluation_periods 配合使用，另外一个作用就是它确定了两个点之间的时间间隔，默认是 60s
- threshold: 阈值
- comparison_operator: 这个参数确定了怎么和阈值进行比较,有 6 个可选：lt, le, eq, ne, ge, gt，默认是 eq
- statistic: 这个参数确定了使用什么数据去和 threshold 比较，有 5 种可选：max, min, avg, sum, count，默认是 avg
- evaluation_periods: 和 period 参数相乘，可以确定获取监控数据的时间范围，默认是 1
- exclude_outliners: 这个参数有点意思，我们都知道“标准差”是指一组数据的波动大小，平均值相同，但是标准差小的波动小，这个参数就是指对得到的一组监控数据，是否要根据标准差去除那些波动比较大的数据，以降低误判率，默认是 False

AlarmCombinationRule:

- operator: 定义 alarms 之间的逻辑关系，有两个选项：or 和 and，默认是 and，注意这里的逻辑关系 ALARM 要比 OK 状态优先级高，比如有 2 个 alarm，一个状态是 ALARM,一个状态是 OK,他们之间的逻辑关系是 or,那么这个 combination alarm 是啥状态呢？答案是 ALARM.
- alarms_id: alarm 列表

AlarmTimeConstraint:

- name: name
- description: description
- start: 该参数以 cron 的格式指定了 alarm 被检查的开始时间，在程序中，使用 croniter 这个库来实现 cron，格式是: "min hour day month day_of_week"，比如"2 4 mon,fri"，意思是在每周一和周五的 04:02 开始被检查
- duration: 被检查持续的时间，单位是秒

- timezone：可以为上面的检查时间指定时区，默认使用的是 UTC 时间

举两个例子：

threshold

```
{
    "name": "ThresholdAlarm1",
    "type": "threshold",
    "threshold_rule": {
        "comparison_operator": "gt",
        "evaluation_periods": 2,
        "exclude_outliers": false,
        "meter_name": "cpu_util",
        "period": 600,
        "query": [
            {
                "field": "resource_id",
                "op": "eq",
                "type": "string",
                "value": "2a4d689b-f0b8-49c1-9eef-87cae58d80db"
            }
        ],
        "statistic": "avg",
        "threshold": 70.0
    },
    "alarm_actions": [
        "http://site:8000/alarm"
    ],
    "insufficient_data_actions": [
        "http://site:8000/nodata"
    ],
    "ok_actions": [
        "http://site:8000/ok"
    ],
    "repeat_actions": false,
    "time_constraints": [
        {
```

```
            "description": "nightly build every night at 23h for 3 hours",
            "duration": 10800,
            "name": "SampleConstraint",
            "start": "0 23 * * *",
            "timezone": "Europe/Ljubljana"
        }
    ]
}
```

combination

```
{
    "name": "CombinationAlarm1",
    "type": "combination",
    "combination_rule": {
        "alarm_ids": [
            "739e99cb-c2ec-4718-b900-332502355f38",
            "153462d0-a9b8-4b5b-8175-9e4b05e9b856"
        ],
        "operator": "or"
    },
    "alarm_actions": [
        "http://site:8000/alarm"
    ],
    "insufficient_data_actions": [
        "http://site:8000/nodata"
    ],
    "ok_actions": [
        "http://site:8000/ok"
    ]
}
```

2. GET /v2/alarms/{alarm_id}/history

这个接口用来查询某个 alarm 发生的历史事件，记录的事件有：alarm 被创建，alarm 被更新，alarm 被删除，alarm 的状态被更新。

举个例子，比如我创建了一个 alarm，然后又删除了，调用这个接口返回的结果是：

```
[
    {
        "on_behalf_of": "2c35166baba84f46b1c5b093f02747fa",
        "user_id": "778a4ae5d8904a41b00c4e0f5734bcfd",
        "event_id": "dc5583ac-7ac8-4f4e-b8f7-edaa04522945",
        "timestamp": "2014-07-26T16:50:59.387923",
        "detail": "xxx",
        "alarm_id": "697a05df-d704-46a4-a0bd-1591c6588a17",
        "project_id": "2c35166baba84f46b1c5b093f02747fa",
        "type": "deletion"
    },
    {
        "on_behalf_of": "2c35166baba84f46b1c5b093f02747fa",
        "user_id": "778a4ae5d8904a41b00c4e0f5734bcfd",
        "event_id": "d09fe2c3-37a8-4b19-9729-ccb2664a1116",
        "timestamp": "2014-07-26T16:50:27.315824",
        "detail": "xxx",
        "alarm_id": "697a05df-d704-46a4-a0bd-1591c6588a17",
        "project_id": "2c35166baba84f46b1c5b093f02747fa",
        "type": "creation"
    }
]
```


5.3.2.2 告警分布式实现

对于 Alarm 的实现，值得一说的就是 Alarm 的分布式实现：Distributed Alarm。Ceilometer 提供了两种方式的 Alarm 服务，一种是单进程的(SingletonAlarmService)，一种是分布式的 (PartitionedAlarmService)，可以通过 evaluation_service 这个配置项进行配置。

前者没啥可说的，就是在一个进程中去检查所有的 alarm，这种方式主要的缺点是处理能力弱，当量稍微大的时候，就会有延时，而且也没法做高可用，当他挂掉之后，alarm 整个 service 就挂掉了，所以不推荐在生产环境中使用这个 SingletonAlarmService 的方式。

对于 PartitionedAlarmService，它通过 rpc 实现了一套多个 evaluator 进程之间的协作协议(PartitionCoordinator)，使得可以通过水平扩展来不断增大 alarm service 的处理能力，这样不仅实现了一个简单的负载均衡，还实现了高可用。下面我们就重点来说一下 PartitionCoordinator 这个协议。

PartitionCoordinator 允许启动多个 ceilometer-alarm-evaluator 进程，这多个进程之间的关系是互相协作的关系，他们中最早启动的进程会被选为 master 进程，master 进程主要做的事情就是给其他进程分配 alarm，每个进程都在周期性的执行三个任务：

- 通过 rpc，向其它进程广播自己的状态，来告知其他进程，我是活着的，每个进程中都保存有其他进程的最后活跃时间。
- 争抢 master，每个进程都会不断的更新自己所维护的其它进程的状态列表，根据这个状态列表，来判断是否应该由自己来当 master，判断一个进程是否是 master 的条件只有一个，那就是看谁启动的早。
- 检查本进程负责的 alarm，会去调用 ceilometer 的 api，来获取该 alarm 的监控指标对应的监控数据，然后进行判断，发送报警等。

进程之间的关系可参看下图：

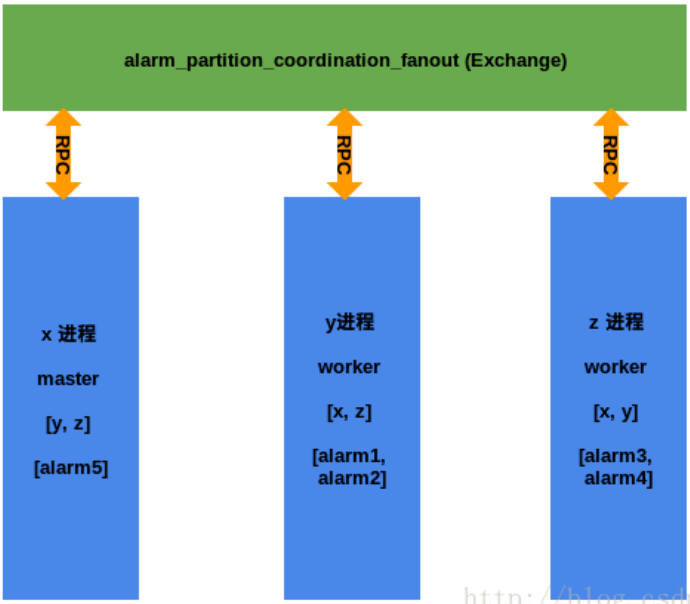

图 5.4 分布式告警

当一个进程被确定为 master 之后，如果它不挂掉，那么它的 master 是不会被抢走的，该进程就会一直在履行 master 的职责：

- 当有新的 alarm 被创建时，master 会将这些新创建的 alarm 平均的分配给其它 worker 进程，如果不能平均分配的，剩下的零头就由 master 自己来负责
- 当有新的 evaluator 进程添加进来，或者是现有的 evaluator 进程被 kill 掉，那么 master 就会重新洗牌一次，把所有的 alarm 再平均的分配给现有的 evaluator

进程

- 当 master 挂掉咋办呢？那么就会由第二个最早启动的进程接替 master 的位置，然后重新洗牌

通过这个协议，就实现了一个简单的分布式 alarm 服务。

5.4 扩展性考虑

无

5.5 系统性能的考虑

无

5.6 安全性的考虑

无

5.7 部署考虑

无

5.8 关键设计驱动因素

无

5.9 可靠性的考虑

无

5.10 兼容性的考虑

无

5.10.1 新版本对老数据的兼容性考虑

无

5.11 可测试性的考虑

无

6 构架说明

6.1 系统构架

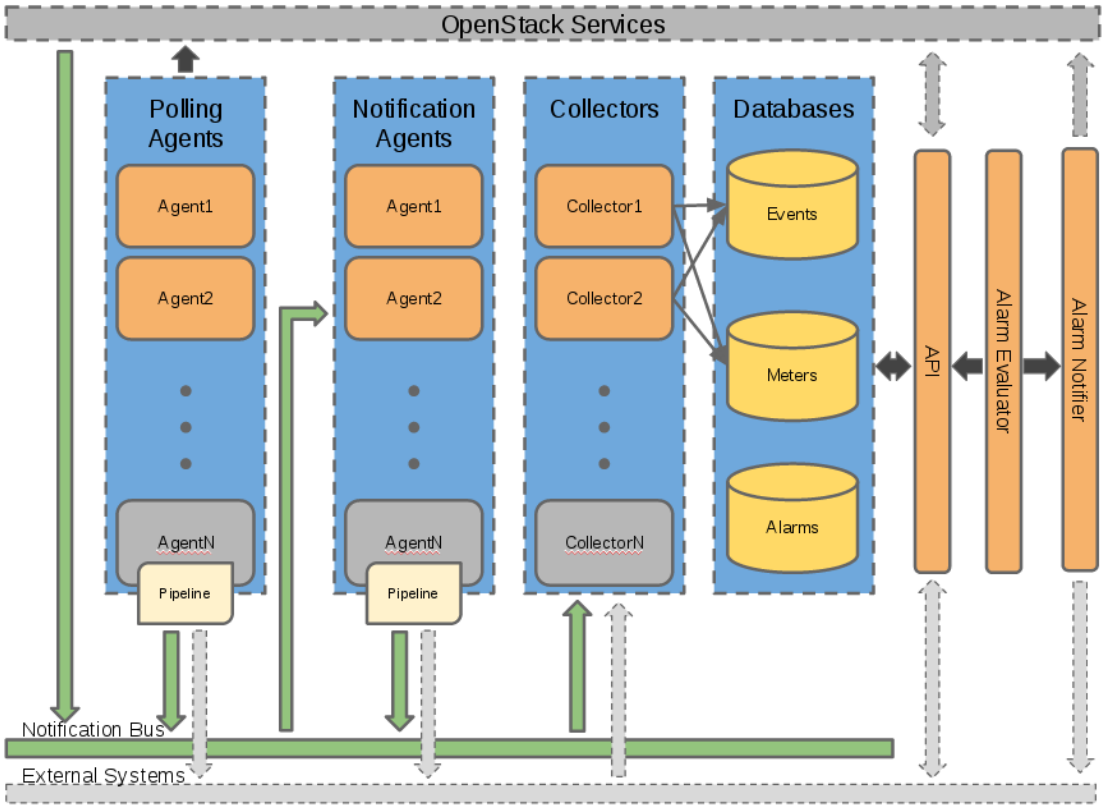

图 6.1 Ceilometer 系统构架

6.2 配置说明

无

6.3 部署说明

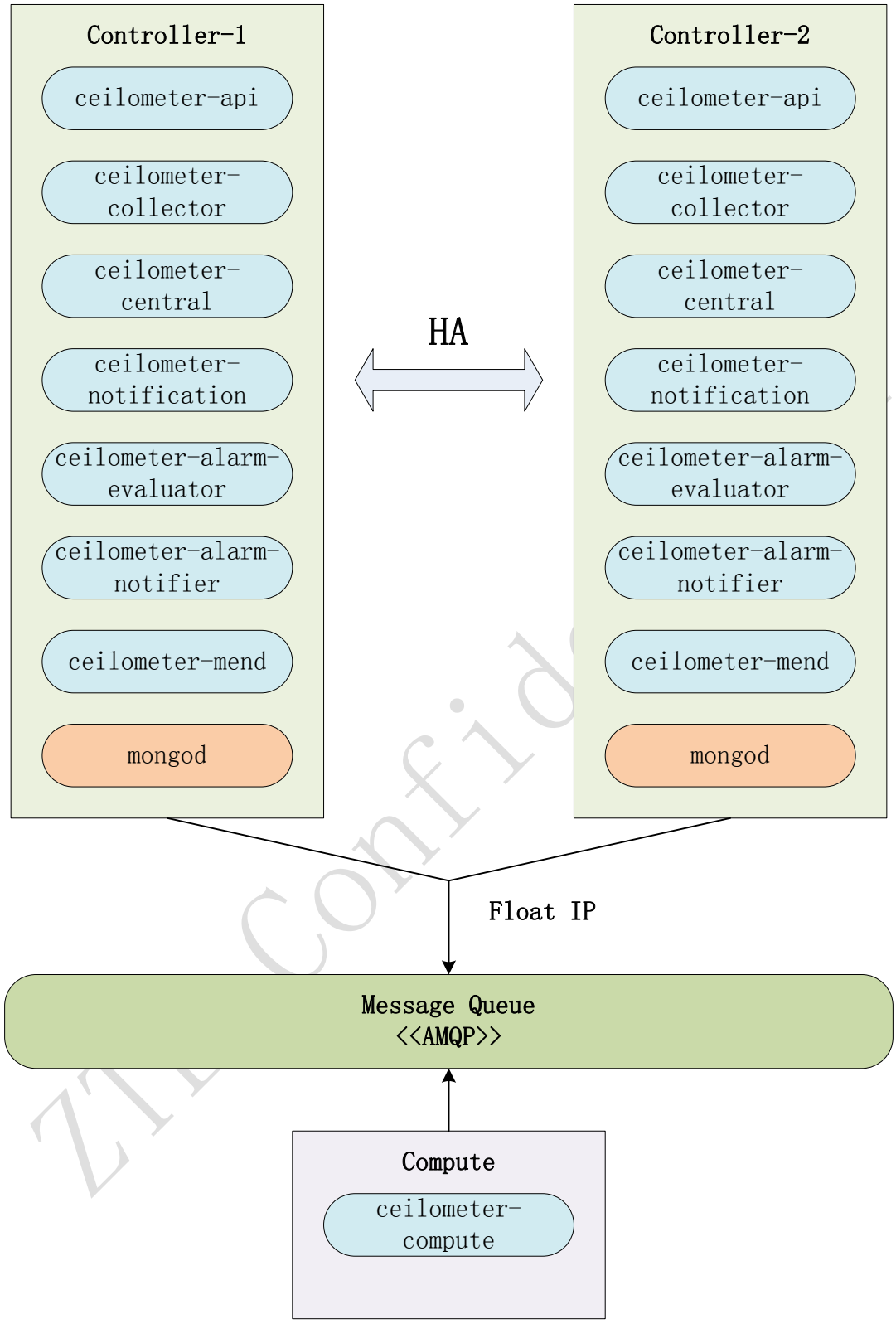

图 6.2 Ceilometer 部署

注：

- Openstack 其它组件主动上报的通知消息，是通过 AMQP 消息服务器发给 Ceilometer，因此必须保证共用 AMQP 消息服务器。

7 协作说明

无

8 组件说明

8.1 Ceilometer Compute Agent

8.1.1 功能概述

Ceilometer Compute Agent 服务组件主要用来收集计算节点上的虚拟机实例的监控数据，在每一个计算节点上都要运行这个服务组件，该 agent 通过 Stevedore 管理了一组 pollster 插件，分别用来获取计算节点上虚拟机的 CPU、Memory、Disk IO、Network IO、Instance 这些信息，这些信息大部分是通过调用 Hypervisor 的 API 来获取的，需要定期 Poll 轮询收集信息。

8.1.2 处理流程

8.1.2.1 服务 ceilometer-agent-compute 的初始化操作

服务 ceilometer-agent-compute 的初始化操作主要实现了以下内容的操作：

1. 根据指定参数获取命名空间 ceilometer.poll.compute，获取与 ceilometer.poll.compute 相匹配的所有插件，并加载；ceilometer.poll.compute 所指定的插件描述了如何获取收集所监控的虚拟机实例相关的监控采样数据。
2. 根据指定参数获取命名空间 ceilometer.discover，获取与 ceilometer.discover 相匹配的所有插件，并加载；ceilometer.discover 所指定的插件描述了如何发现主机上的监控的虚拟机。

ceilometer.discover = local_instances = ceilometer.compute.discovery:InstanceDiscovery

3. 获取管理员操作的上下文环境类的初始化对象。
4. 建立线程池，用于后续服务中若干操作的运行。
5. 加载命名空间 ceilometer.compute.virt，描述了获取虚拟机实例的信息（实例数据，CPU 数据，网卡数据和磁盘数据等）的方式。

8.1.2.2 服务 ceilometer-agent-compute 的启动操作

这里服务 ceilometer-agent-compute 与服务 ceilometer-agent-central 的启动操作的实现代码是完全一致的，只不过所获取的监控任务不同而已，所调用的采集监控项采样数据的方法不同而已。

服务 ceilometer-agent-compute 的启动操作周期性地实现以下任务：

1. 遍历任务（通道），获取每个任务指定获取的监控项的采样数据。
2. 针对每个监控项的采样数据，实现发布监控项采样数据样本到消息队列，其中实现采样数据发布的方式有三种，即 RPC/UDP/FILE。

其中，RPC 将会发布相关消息到消息队列，后续的 collector 组件服务将会监听相应的消息队列来获取这些数据信息；UDP 将会建立 socket 建立一个信息通道，实现发送相关消息数据，而后续的 collector 组件服务将会通过这个信息通道接收相关的消息数据；FILE 将会直接保存相关消息数据到指定的日志文件中。

8.2 Ceilometer Central Agent

8.2.1 功能概述

ceilometer-agent-central 服务组件运行在控制节点上，它主要通过调用相关模块的 REST API，通过访问相关模块的客户端，从而实现主动收集相关模块（Image,Volume,Objects,Network）的监控数据，需要定期 Poll 轮询收集信息。

8.2.2 处理流程

8.2.2.1 服务 ceilometer-agent-central 的初始化操作

服务 ceilometer-agent-central 的初始化操作主要实现了以下内容的操作：

1. 根据指定参数获取命名空间 ceilometer.poll.central，获取与 ceilometer.poll.central 相匹配的所有插件，并加载；ceilometer.poll.central 所指定的插件描述了如何获取收集相关模块（Image,Volume,Objects,Network）的监控数据。
2. 获取管理员操作的上下文环境类的初始化对象。
3. 建立线程池，用于后续服务中若干操作的运行。

8.2.2.2 服务 ceilometer-agent-central 的启动操作

服务 ceilometer-agent-central 的启动操作周期性地实现以下任务：

1. 遍历任务（通道），获取每个任务指定获取的监控项的采样数据。
2. 针对每个监控项的采样数据，实现发布监控项采样数据样本到消息队列，其中实现采样数据发布的方式有三种，即 RPC/UDP/FILE。

其中，RPC 将会发布相关消息到消息队列，后续的 collector 组件服务将会监听相应的消息队列来获取这些数据信息；UDP 将会建立 socket 建立一个信息通道，实现发送相关消息数据，而后续的 collector 组件服务将会通过这个信息通道接收相关的消息数据；FILE 将会直接保存相关消息数据到指定的日志文件中。

8.3 Ceilometer Notification Agent

8.3.1 功能概述

ceilometer-agent-notification 服务组件实现访问 oslo-messaging，openstack 中各个模块都会推送通知（notification）信息到 oslo-messaging 消息框架，ceilometer-agent-notification 通过访问这个消息队列服务框架，获取相关通知信息，并进一步转化为采样数据的格式。从消息队列服务框架获取通知信息，并进一步获取采样数据信息，可以理解为被动获取监控数据操作，需要一直监听 oslo-messaging 消息队列。

8.3.2 处理流程

8.3.2.1 服务 ceilometer-agent-notification 的初始化操作

服务 ceilometer-agent-notification 的初始化操作主要实现了以下内容的操作：

1. 若干参数的初始化，定义了所要监听序列的 host 和 topic。
2. 建立线程池，用于后续服务中若干操作的运行。

8.3.2.2 服务 ceilometer-agent-notification 的启动操作

服务 ceilometer-agent-notification 的启动操作实现了以下任务。

1. 加载命名空间'ceilometer.dispatcher'中的插件。
2. 为 RPC 通信建立到信息总线的连接，建立指定类型的消息消费者。
3. 启动协程实现启动启动消费者线程，等待并消费处理队列'ceilometer.agent.notification'中的消息。
4. 连接到消息总线来获取通知信息；实际上就是实现监听 oslo-messaging 消息框架

中 compute/image/network/heat/cinder 等服务的队列。

5. 从队列中获取通知信息，将通知转换程采样数据的格式，然后进行采样数据的发布操作；从通知获取采样数据信息，可以理解为被动获取数据操作。

8.4 Ceilometer Collector

8.4.1 功能概述

当信息发布操作完成之后，ceilometer-collector 组件服务将会分别获取相关的消息数据，并实现保存获取的消息数据到数据存储系统中。而数据存储系统方案目前也支持几种实现，即 log/mongodb/mysql/postgresql/sqlite/hbase/db2 等。

8.4.2 处理流程

8.4.2.1 服务 ceilometer-collector 的初始化操作

服务 ceilometer-collector 的初始化操作主要实现了以下内容的操作：

1. 若干参数的初始化，定义了所要监听序列的 host 和 topic。
2. 建立线程池，用于后续服务中若干操作的运行。

8.4.2.2 服务 ceilometer-collector 的启动操作

服务 ceilometer-collector 通过监听对应的队列来获取发布到消息队列的采样数据信息，并实现保存到存储系统中。

服务 ceilometer-collector 的启动操作实现了以下任务：

提供两种方式（UDP，RPC）获取收集发布的信息，并保存到数据存储系统中；

1. 针对 UDP 的消息发布方式，调用方法实现：
   1) 获取 socket 对象；
   2) 一直循环任务通过 UDP 协议实现接收消息数据 data；
   3) 保存数据 data 到数据存储系统（不同的实现后端）；

2. 针对 RPC 的消息发布方式：
   1) 建立指定类型的消息消费者；
   2) 执行方法 initialize_service_hook；

- 建立一个'topic'类型的消息消费者；
- 根据消费者类（TopicConsumer）和消息队列名称（ceilometer.collector.metering，即监听消息队列ceilometer.collector.metering）以及指定主题 topic（metering）建立消息消费者，并加入消费者列表；

3) 启动协程实现等待并消费处理队列中的消息；

4) 加载命名空间'ceilometer.dispatcher'中的插件：

ceilometer.dispatcher =

database = ceilometer.dispatcher.database:DatabaseDispatcher

file = ceilometer.dispatcher.file:FileDispatcher

描述了收集发布的监控信息保存到数据系统的实现方式；

8.5 Ceilometer API

8.5.1 功能概述

ceilometer-api 主要对外提供数据的 REST 访问接口，API 服务以 wsgi service 方式运行在后端，Ceilometer 有 v1 和 v2 两个版本的 API，v1 会被弃用，目前基本都是使用 v2 版本的接口。

8.5.2 处理流程

8.5.2.1 服务 ceilometer-api 的初始化操作

服务 ceilometer-api 的初始化操作主要实现了以下内容的操作：

1. 若干参数的初始化，获取 wsgi server 的 host 和 port。
2. 启动 wsgi server，监听对应端口。

8.5.2.2 服务 ceilometer-api 的启动操作

ceilometer-api 服务守护对应的端口，接收 http 请求，并进行处理。

8.6 Ceilometer Alarm evaluator

8.6.1 功能概述

ceilometer-alarm-evaluator 是 Ceilometer 的告警服务，通过查询指定周期内的度量值，评估是否满足用户设置的告警阈值，从而触发对应的动作。

根据单节点或者多节点上运行，可以分为单节点告警服务和分布式告警服务。

8.6.2 处理流程

8.6.2.1 单节点服务 SingletonAlarmService 的初始化和启动操作

8.6.2.1.1 SingletonAlarmService 类初始化操作

类 SingletonAlarmService 的初始化操作主要完成了两部分内容：

- 加载命名空间 ceilometer.alarm.evaluator 中的所有插件；

ceilometer.alarm.evaluator =

threshold = ceilometer.alarm.evaluator.threshold:ThresholdEvaluator

combination = ceilometer.alarm.evaluator.combination:CombinationEvaluator

- 建立线程池，用于后续报警器服务中若干操作的运行；

8.6.2.1.2 SingletonAlarmService 类启动操作

类的启动操作实现了单例报警器服务 SingletonAlarmService 的启动操作；

按照一定时间间隔实现循环执行方法 self._evaluate_assigned_alarms，方法 self._evaluate_assigned_alarms 实现获取 alarm 集合，针对每一个报警器，实现根据报警器模式的类型（threshold 和 combination），来实现单一报警器模式或者联合报警器模式的评估判定。

8.6.2.2 分布式告警服务 PartitionedAlarmService 的初始化和启动操作

8.6.2.2.1 PartitionedAlarmService 类初始化操作

PartitionedAlarmService 类初始化操作和 SingletonAlarmService 类初始化操作内容大致是相同的，同样主要完成了以下内容：

- 加载命名空间 ceilometer.alarm.evaluator 中的所有插件；

ceilometer.alarm.evaluator =

threshold = ceilometer.alarm.evaluator.threshold:ThresholdEvaluator

combination = ceilometer.alarm.evaluator.combination:CombinationEvaluator

即描述了报警器状态的评估判定的两种模式：联合报警器状态评估和单一报警器状态评估；

- 建立线程池，用于后续报警器服务中若干操作的运行；
- 初始化分布式报警协议实现类 PartitionCoordinator；

8.6.2.2.2 PartitionedAlarmService 类启动操作

分布式报警器系统服务分布式报警器系统服务的启动和运行，按照一定的时间间隔周期性的执行以下操作：

1. 实现广播当前 partition 的存在性的存在性到所有的 partition（包括 uuid 和优先级信息）。
2. 实现定期检测主控权角色；确定当前的 partition 是否是主控角色；

如果为拥有主控权的 partition，则根据不同的情况实现不同形式的报警器分配操作：

情况 1：所有报警器都要实现重新分配操作；

情况 2：只有新建立的报警器需要实现分配操作；

3. 获取 alarm 集合，对每一个 alarm,调用_evaluate_alarm 方法。

针对每一个报警器，实现根据报警器的类型（threshold 和 combination），来实现单一报警器模式或者联合报警器模式的评估判定。

8.7 Ceilometer Alarm notifier

8.7.1 功能概述

ceilometer-alarm-notifier 的功能是当报警器被触发之后，发送相关的通知操作。

8.7.2 处理流程

8.7.2.1.1 服务 ceilometer-alarm-notifier 的初始化

实现 AlarmNotifierService 类的初始化操作，在类的初始化过程中会加载命名空间 ceilometer.alarm.notifier 所定义的所有插件，确定所有实现通知操作的实现方式；

```
ceilometer.alarm.notifier=
log= ceilometer.alarm.notifier.log:LogAlarmNotifier
test= ceilometer.alarm.notifier.test:TestAlarmNotifier
http= ceilometer.alarm.notifier.rest:RestAlarmNotifier
```

https= ceilometer.alarm.notifier.rest:RestAlarmNotifier

定义了通知操作的实现方式，即记录报警器触发信息到日志和通过 http/https 协议实现发送报警器触发信息。

8.7.2.1.2 服务 ceilometer-alarm-notifier 的启动

方法 notify_alarm 是具体实现报警器触发后的发送通知操作的具体方法，当报警器被触发之后，都会调用这个方法，主要实现了以下内容：

遍历所有要通知报警器被触发的 URL，针对每个要通知的 URL 地址，实现：

1) 获取系统所采用的消息通信方式；

2) 通过 HTTP/HTTPS 协议 POST 方法实现发送相关报警器被触发的通知（到 action 指定的 URL），（or）通过日志记录相关报警器被触发的通知。

8.8 Ceilometer Mend

8.8.1 功能概述

ceilometer-mend 是 OpenCOS 扩展的服务，目前功能比较简单，定时清理过期的性能指标数据。

8.8.2 处理流程

ceilometer 通过 pollster 和 notification 获取系统中的测量数据记录到数据库中，数据是通过增量添加的方式放到数据库中，而不是覆盖原有数据，这样随着时间的迁移，数据库中的数据量会非常庞大，影响系统性能。

ceilometers-mend 增加了定时清理数据库的功能，定时清理数据库中过老的数据。使用如下 2 个配置项：

- time_to_live：保留数据库中多长时间以内的数据，超过这个时间的数据将被清理，以秒为单位。目前默认值是 1800，即保持半小时的历史数据。
- db_expire_interval：执行数据库定时清理的周期间隔，即多长时间清理一次，以秒为单位，该时间如果太小会影响系统性能，因此配置要求最小为 600s，否则无效。目前默认值是 1800，即半小时执行一次清理动作。

9 接口说明

Ceilometer 接口遵从 Openstack 原生接口规范，参见：

http://developer.openstack.org/api-ref-telemetry-v2.html

10 设计局限性说明

无。

11 参考文献

[1] http://docs.openstack.org/developer/ceilometer/

[2] Ceilometer H 版本源码解析-2013.2.1.docx