第 3 章

数据采集、传输与过滤

前两章介绍了如何分别从集群的内部和外部对其运行状况进行监控。不过，监控并不是运维的目的，而只是基础，所以我们需要学会如何从各种监控数据中提取真正有用的信息，并且做到快速、常态化地过滤信息，以辅助我们总结集群的运行和发展趋势，最终就可以做到预判将要到来的事件并提前反应了。

3.1 采集点的取舍

说到数据分析，首先当然是数据越全面越详细越好。但是随之带来的分析复杂度也同样让人头疼。所以首先我们要针对不同类型的数据，确定正常收集状态下的采样点。

3.1.1 服务器数据

首先是服务器数据，即第二章所讲的 sar 等命令采集到的服务器负载、磁盘读写、网卡流量等信息。这些采样标准，其实完全可以根据数据本身的概念和命令的原理来确定采样频率。

比如平均负载 loadavg，输出的三个值本身就已经是 1 分钟、5 分钟、15 分钟的平均值，那么小于 1 分钟的采样就是没有意义的。

相反，对于 I/O 操作，大多数条件下都是瞬时操作，如果依然使用 5 分钟甚至 10 分钟的采样频率，很可能就会遗漏掉一些异常情况直到累积爆发。

而另一方面，即便是相同的数据，针对不同应用的采样点也应该有所区别。

比如常见的缓存服务器软件 Squid，在 version 3.2 之前，是不支持多核的。那么其数据采样就不能和 Nginx 等服务一样采用 loadavg，而需要关注其具体的 CPU util%。

3.1.2　访问日志

访问日志与服务器数据又有不同：每一条访问日志都是有意义的，而且访问日志通常会在相隔较久（当然我们运维人员正是要通过各种努力让这个“较久”变得更短）之后被要求重放——因为这时候出现的问题大多是“偶然”性的、不影响服务器本身性能的、难以快速反馈的隐藏 bug。所以在有条件的情况下，应该保留全部的访问日志至少 3 个月以上！

基本上 Web 服务器软件都有自己的默认日志记录项。具体如下。

a．在 Apache 中，叫做 combined，格式定义如下。

```
    LogFormat "%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\""
combined
```

b．在 Nginx 中，叫做 main，格式定义如下。

```
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';
```

c．在 Squid 中，叫做 squid，格式定义如下。

```
    logformat squid %ts.%03tu %6tr %>a %Ss/%03Hs %<st %rm %ru %un %Sh/%<A %mt
```

在传统的静态页面访问情况下，基本上这些已经足够了。不过针对动态页面的情况，还是要特意指出另一个内容，值得，至少是在排查故障期间值得，被记录下来。那就是 Cookie。

a．Apache 中的 Cookie 记录方法

```
%{cookie}n
```

b．Nginx 中的 Cookie 记录方法

```
$http_cookie
```

在 Nginx 中，还提供了另一种精确定位的记录方式：在可以确定只需要记录 Cookie 中的某一个 key:value 对时，可以使用如下变量单独记录这个 value 即可。

```
$cookie_KEY
```

c．Squid 中的 Cookie 记录方法

```
%{Cookie}>h
```

3.1.3 系统日志 Syslog

Syslog 是介于访问日志和服务器数据之间的另一部分。一方面是作为 Linux 服务器最重要的 OS 层面的信息集中地，另一方面 Syslog 本身作为一种快速传输日志的协议，也经常用于用户应用输出。并由此产生了 Syslog-ng、Rsyslog 等一系列成规模的 Syslog 收集分析系统。

这里暂时仅针对 Linux 本身的 Syslog 做采样分析介绍。因为大部分情况下，用户程序选择输出到 Syslog 时，就会针对性地带有采集分析办法。

对于 Syslog 协议内容，最权威的自然是 RFC 文档。涉及 Syslog 的 RFC 文档不少，不过最基本而且最重要的内容格式是在 RFC3164（http://www.ietf.org/rfc/rfc3164.txt）里定义的。

其中 *3.1 Syslog Message Parts* 章节是我们这里需要关心的部分。

3.1.3.1 PRI

PRI 由尖括号和数字组成。这里的数字是由高低位组合计算而成的。其中高位的叫做 Facility（设备，不过为了跟 header 中的 device 区别，用程序来称呼更恰当），低位的叫做 Severity（严重性）。

其程序位含义列表如图 3-1 所示。

```
Numerical           Facility
   Code
    0           kernel messages
    1           user-level messages
    2           mail system
    3           system daemons
    4           security/authorization messages
    5           messages generated internally by syslogd
    6           line printer subsystem
    7           network news subsystem
    8           UUCP subsystem
    9           clock daemon
   10           security/authorization messages
   11           FTP daemon
   12           NTP subsystem
   13           log audit
   14           log alert
   15           clock daemon (note 2)
   16           local use 0  (local0)
   17           local use 1  (local1)
   18           local use 2  (local2)
   19           local use 3  (local3)
   20           local use 4  (local4)
   21           local use 5  (local5)
   22           local use 6  (local6)
   23           local use 7  (local7)
```

图 3-1

其严重性位含义列表如图 3-2 所示。

```
Numerical        Severity
  Code
    0       Emergency: system is unusable
    1       Alert: action must be taken immediately
    2       Critical: critical conditions
    3       Error: error conditions
    4       Warning: warning conditions
    5       Notice: normal but significant condition
    6       Informational: informational messages
    7       Debug: debug-level messages
```

图 3-2

计算规则是：

$$PRI = Facility \times 8 + Level$$

$$Facility = PRI / 8$$

$$Level = PRI \% 8$$

根据程序位和严重性位含义可以计算得出，PRI 最小为 0，最大为 191。一般来说，非自定义的程序位，最常见的就是 kernel，这时候也最简单，因为程序位是 0。自定义日志时，一般采用 local(0-7)发送。

Linux 系统一般带有一个命令 logger，可用于直接从命令行写入 Syslog，用法如下。

```
ping -c 3 127.0.0.1 | logger -it log4ping -p local3.notice
```

之后我们可以在 Syslog 中看到如下记录。

```
Apr  4 10:41:01 localhost <157> log4ping[10992]: PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
Apr  4 10:41:01 localhost <157> log4ping[10992]: 64 bytes from 127.0.0.1: icmp_req=1 ttl=64 time=0.029 ms
Apr  4 10:41:02 localhost <157> log4ping[10992]: 64 bytes from 127.0.0.1: icmp_req=2 ttl=64 time=0.023 ms
Apr  4 10:41:03 localhost <157> log4ping[10992]: 64 bytes from 127.0.0.1: icmp_req=3 ttl=64 time=0.029 ms
Apr  4 10:41:03 localhost <157> log4ping[10992]:
Apr  4 10:41:03 localhost <157> log4ping[10992]: --- 127.0.0.1 ping statistics ---
Apr  4 10:41:03 localhost <157> log4ping[10992]: 3 packets transmitted, 3 received, 0% packet loss, time 1999ms
Apr  4 10:41:03 localhost <157> log4ping[10992]: rtt min/avg/max/mdev = 0.023/0.027/0.029/0.003 ms
```

记录中的 157，就是通过 local3 的 19 乘以 8 加上 notice 的 5 得到的。

如果你发现自己设备上的 Syslog 结果不是这个样子，暂时不要奇怪，稍后读到 Rsyslog 章节的时候就明白了。

3.1.3.2 HEADER

HEADER 部分包括 TIMESTAMP（时间）和 HOSTNAME（设备主机名）。

TIMESTAMP 一般格式为“Mmm dd hh:mm:ss”，注意这里没有具体年份，但是实际上年份对于长期运行的系统数据是有意义的，所以在具体的 Syslog 实现中，有些变体会加上年份，导致格式与 RFC 不一致。

HOSTNAME 一般为主机名，也可能是 IP 地址。注意主机名不包括域名部分。

在有些 Syslog 实现中干脆省略掉了 HEADER。

3.1.3.3　MSG

MSG 部分包括 TAG（标签）和 CONTENT（内容）。TAG（标签）后面以“:”分隔。

TAG 包括发送该 Syslog 的进程的 task_name 和 pid。其中 pid 写在[]中括号内，pid 部分也可以忽略。

因为协议规定，Syslog 单行不能超过 1024B，所以和 log4j 类似，Syslog 也会出现同一个事件分成多行发送的情况。这时候，通常就不太方便进行单行分析了，比如下面这段 Syslog。

```
    Nov 26 15:42:54 LOCALHOST kernel: [    4.235784] cciss0: <0x3230> at PCI
0000:03:00.0 IRQ 1272 using DAC
    Nov 26 15:42:54 LOCALHOST kernel: [    4.251582]       blocks= 286677120
block_size= 512
    Nov 26 15:42:54 LOCALHOST kernel: [    4.255582]      heads=255, sectors=32,
cylinders=35132
    Nov 26 15:42:54 LOCALHOST kernel: [    4.255583]
    Nov 26 15:42:54 LOCALHOST kernel: [    4.263574]       blocks= 286677120
block_size= 512
    Nov 26 15:42:54 LOCALHOST kernel: [    4.263574]      heads=255, sectors=32,
cylinders=35132
    Nov 26 15:42:54 LOCALHOST kernel: [    4.263574]
    Nov 26 15:42:54 LOCALHOST kernel: [    4.263574]  cciss/c0d0: p1 p2 < p5 p6 >
```

针对 UDP 的丢包请求，还有 RFC3195 定义的如何使用 TCP 协议传输 Syslog，而 RFC5425（http://tools.ietf.org/html/rfc5425）则进一步定义了采用 TLS 协议传输 Syslog 时的 HEADER 格式等。

此外，RFC5424 也定义了一种 Syslog 格式，字段解释如下。

```
    SYSLOG-MSG      = HEADER SP STRUCTURED-DATA [SP MSG]
```

```
HEADER          = PRI VERSION SP TIMESTAMP SP HOSTNAME
                  SP APP-NAME SP PROCID SP MSGID
PRI             = "<" PRIVAL ">"
PRIVAL          = 1*3DIGIT ; range 0 .. 191
VERSION         = NONZERO-DIGIT 0*2DIGIT
HOSTNAME        = NILVALUE / 1*255PRINTUSASCII

APP-NAME        = NILVALUE / 1*48PRINTUSASCII
PROCID          = NILVALUE / 1*128PRINTUSASCII
MSGID           = NILVALUE / 1*32PRINTUSASCII

TIMESTAMP       = NILVALUE / FULL-DATE "T" FULL-TIME
FULL-DATE       = DATE-FULLYEAR "-" DATE-MONTH "-" DATE-MDAY
DATE-FULLYEAR   = 4DIGIT
DATE-MONTH      = 2DIGIT  ; 01-12
DATE-MDAY       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on
                          ; month/year
FULL-TIME       = PARTIAL-TIME TIME-OFFSET
PARTIAL-TIME    = TIME-HOUR ":" TIME-MINUTE ":" TIME-SECOND
                  [TIME-SECFRAC]
TIME-HOUR       = 2DIGIT  ; 00-23
TIME-MINUTE     = 2DIGIT  ; 00-59
TIME-SECOND     = 2DIGIT  ; 00-59
TIME-SECFRAC    = "." 1*6DIGIT
TIME-OFFSET     = "Z" / TIME-NUMOFFSET
TIME-NUMOFFSET  = ("+" / "-") TIME-HOUR ":" TIME-MINUTE

STRUCTURED-DATA = NILVALUE / 1*SD-ELEMENT
SD-ELEMENT      = "[" SD-ID *(SP SD-PARAM) "]"
SD-PARAM        = PARAM-NAME "=" %d34 PARAM-VALUE %d34
SD-ID           = SD-NAME
PARAM-NAME      = SD-NAME
PARAM-VALUE     = UTF-8-STRING ; characters '"', '\' and
                               ; ']' MUST be escaped.
SD-NAME         = 1*32PRINTUSASCII
                  ; except '=', SP, ']', %d34 (")

MSG             = MSG-ANY / MSG-UTF8
MSG-ANY         = *OCTET ; not starting with BOM
MSG-UTF8        = BOM UTF-8-STRING
BOM             = %xEF.BB.BF
```

```
UTF-8-STRING    = *OCTET ; UTF-8 string as specified
                ; in RFC 3629

OCTET          = %d00-255
SP             = %d32
PRINTUSASCII    = %d33-126
NONZERO-DIGIT   = %d49-57
DIGIT          = %d48 / NONZERO-DIGIT
NILVALUE        = "-"
```

这个格式比 RFC3164 中的归档要复杂和具体很多，不过其中最关键的 PRI 计算方法是一致的。所以掌握好 PRI 的计算，剩下的部分交给工具解决就好了。

3.2　收集传输

在第二章中，我们已经介绍了 Nagios 和 Ganlia 等系统的不同的监控思路，其实也就是系统数据的收集传输流程。但是正如 3.1 所介绍的，系统数据和访问数据在采集间隔上相距甚远，数据也不是简单的键值对而更多是长字符串文本，这时比较难直接套用 Nagios 和 Ganglia 的思路来完成集中监控，所以需要寻找其他途径。

在实时性要求不高的环境下，通过 scp 或者 rsync 定时任务，进行集中收集，是广大 Linux 运维人员最常用的手段。但是一旦有了较高的实时性要求，scp 和 rsync 就不足以很好地完成任务了，这时候我们需要专门的数据传输中间件。

3.2.1　Rsyslog

上一小节中，我们介绍了 Syslog 协议的组成。大家应该发现了，Syslog 的结构正适合用来跨主机地传输日志。事实上，Syslog 的集中收集，是大规模集群运维中必备的部分。

在以往的 Syslog 介绍中，一般以 Syslog-ng 和 Rsyslog 两个系统的出现最为频繁。可惜 Syslog-ng 却没能真正成为 Syslog 的 ng——CentOS6 中正式替代 Syslog 出现的是 Rsyslog。

鉴于 Rsyslog 已经成为 CentOS6 的标配，Rsyslog 本身也兼容 Syslog 的配置，这里就不再介绍它的安装和基础配置了。

3.2.1.1 主要组成模块

Rsyslog 强化了 Syslog 的各种概念，使之各自独立出来。从实际流程来说，Rsyslog 与标准 Syslog 流程一样，具体见图 3-1 所示。

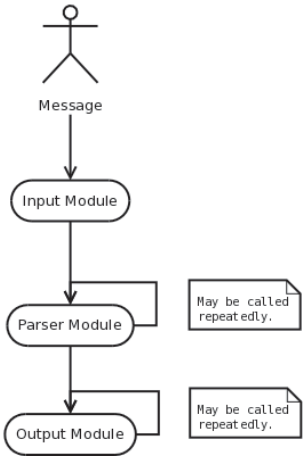

图 3-1

不过由于 Parser 和 Output 模块都可以重复调用，所以它们不再单单用于解析 RFC 定义的格式和保存成文件，还可以用来做复杂的数据修改和格式转化，甚至可以通过 DBI 等接口保存到 MySQL、HDFS、ElasticSearch 等各种地方。

所以我们从用途的角度，可以把 Rsyslog 的模块分为以下几种。

◎ Input Modules

IM 模块是改动最少的，基本上只有 File、TCP、UDP、UNIX Socket 以及在 TCP 基础上的 TLS 和 GSSAPI 等更安全的协议。

◎ Output Modules

狭义的 OM 模块，除了最基本的 File 以外，还有用于中转的 FWD，Pipe，Snmptrap，用于存储的 MySQL、PgSQL、Libdbi、HDFS、MongoDB 等。此外，社区还有 Redis、ZeroMQ、ElasticSearch 和 Solr 的 OM 模块补丁。

◎ Parser Modules

这个模块是在 5.3.4 版本之后新加入的。在之前的 Rsyslog 中，对 Syslog 协议格式的

解析工作是直接在核心代码中完成，不可变更的。不过到目前为止，狭义概念的 PM 模块不多，除了 RFC 解析的以外，只有一个 pmlastmsg 模块，专门用来解析 Syslog 中常见的那句“last messages repated n times”。

◎　Message Modification Modules

MM 模块其实就是在广义的 PM 或者 OM 上实现的。目前和 Rsyslog 代码一起分发的 MM 模块有：Anon 模块，用来转换具体的 IP 地址到 A 类地址；Normalize 模块，用来将 Syslog 格式的 content 转换成为 CEE/Lumberjack 的格式，这个模块是在 OM 上实现的，所以必须在 action 动作后才能调用；Jsonparse 模块，目的也和 normalize 模块一样，不过因为 Lumberjack 格式其实就是 JSON，所以这个直接就解析成 JSON 了；Snmptrapd 模块，在 im 的基础上，提供对严重性位数据的替换修改功能。

◎　String Generator Modules

SM 模块的作用是为 Rsyslog 的 file 和 forward 提供 template 功能。我们可以通过 template 定义自己想要的内容样式。注意这不会影响到 Syslog 本身的协议信息。

3.2.1.2　配置示例

3.2.1.2.1　template

template 有两种写法，在 6.0 版本之前，只能使用下面这种。

```
$template name,param[,options]
```

之后，还可以并且推荐使用这种。

```
template(parameters) { list-descriptions }
```

不过目前 CentOS6 默认的 Rsyslog 版本较低，不做单独升级的话，只能使用上面第一种写法。

在新版 Rsyslog 中，template 有四种类型：list、subtree、string 和 plugin。

a．list

标准情况下，template 使用 list 来输出结构化的数据，比如 ommongodb，当然也可以输出成文本。如下为 Rsyslog 默认的文本输出格式。

```
template(name="FileFormat" type="list") {
        property(name="timestamp" dateFormat="rfc3339")
```

```
            constant(value=" ")
            property(name="hostname")
            constant(value=" ")
            property(name="syslogtag")
            constant(value=" ")
            property(name="msg" spifno1stsp="on" )
            property(name="msg" droplastlf="on" )
            constant(value="\n")
            }
```

b．subtree

使用 subtree 输出成 CEE/Lumberjack 格式（7.1.4 版之后）。tree 以“$!”开头过滤，只生成匹配 subtree 的 JSON 数据，示例如下。

```
    set $!usr!tpl2!msg = $msg;
    set $!usr!tpl2!dataflow = field($msg, 58, 2);
    template(name="tpl2" type="subtree" subtree="$!usr!tpl2")
```

c．string

使用 string 输出成字符串格式，当然其实 list 保存成文件，和 string 是等价的，之前提到的默认的 FileFormat 模板，采用 string 类型书写则如下。

```
    template(name="FileFormat" type="string"
      string=  "%TIMESTAMP%  %HOSTNAME%  %syslogtag%%msg:::sp-if-no-1st-sp%%
msg:::drop-last-lf%\n"
    )
```

string 类型作为传统类型，还可以采用旧版样式书写，如下。

```
    $template FileFormat, "%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-
1st-sp%%msg:::drop-last-lf%\n"
```

d．plugin

使用 plugin 则按照 plugin 提供的参数书写，如下。

```
    template(name="MyTemplateName" type="plugin" string="mystrgen")
```

同样可以采用旧版样式书写，如下。

```
    $template MyTemplateName,=mystrgen
```

3.2.1.2.2　rules

Rsyslog 的 rules 在 Sysklogd 等其他系统中叫做 selectors。每个 rule 中都有一个 filter

和若干个 action。filter 可以是基于 Syslog 的 PRI 来做的，也可以是复杂的类似脚本语言的表达式。

一个 rsyslog.conf 中可以写多个 rules，每条 syslog 信息会从前到后经由全部 rules 依次处理，除非在过程中由 discard action 提前跳出。

对于 rsyslog，rule 是必须存在的，每个 input 都必须 bind 到 rule 上。当然，你在实际配置里可能看不到，因为 Rsyslog 中存在有默认 rule。

下面是一个 rule 的示例。

```
ruleset(name="remote10514"){
        action(type="omfile" file="/var/log/remote10514")
}
ruleset(name="remote10515"){
        action(type="omfile" file="/var/log/remote10515")
}
ruleset(name="remote10516"){
        if prifilt("mail.*") then {
            /var/log/mail10516
            stop
        }
        /var/log/remote10516
}
input(type="imptcp" port="10514" ruleset="remote10514")
input(type="imptcp" port="10515" ruleset="remote10515")
input(type="imptcp" port="10516" ruleset="remote10516")
```

3.2.1.2.3　过滤器 filter

Rsyslog 提供了三种不同类型的过滤器。

a. 基于 RainerScript 的过滤器

RainerScript 是一个专门设计用来处理网络事件的脚本语言，2008 年起 Rsyslog 开始支持 RainerScript。它主要通过逻辑、算术和字符串操作等表达式来过滤，最常用的关键字就是 if，示例如下。

```
if $programname == 'prog1' then {
   action(type="omfile" file="/var/log/prog1.log")
   if $msg contains 'test' then
     action(type="omfile" file="/var/log/prog1test.log")
```

```
      else
        if $syslogfacility-text == 'local0' and $msg startswith 'DEVNAME'
and not ($msg contains_i 'error1' or $msg contains_i 'error0') then /var/log/
somelog
        *.err /var/log/errlog
    }
```

如上所示，在 RainerScript 类型的过滤器中，也可以写其他类型的过滤器。

b．基于程序位和严重性的选择器

selector 是传统的 Syslog 过滤方法，只能针对 PRI 做出是非与的判断。参见上例中最后一条。

```
    *.err /var/log/errlog
```

c．基于属性的过滤器

属性过滤器是 Rsyslogd 独有的，可以对各种属性进行过滤。

属性过滤器的每行都以“:”（冒号）开头，然后紧跟属性名称和逗号，接着是比较操作符和逗号，最后是用引号包裹的内容。注意属性是对大小写敏感的，示例如下。

```
    *.* /var/log/allmsgs-including-informational.log
    :msg, contains, "informational"  ~
    *.* /var/log/allmsgs-but-informational.log
```

这里用“:msg”来过滤 msg 中包含 informational 字样的信息，后面的“~”则是 action 操作，稍后讲述。

比较操作符包括有 contains、isempty、isequal、startswith、regex、ereregex，如果不区分大小写，则在操作符后加“_i”。

> **注意**：这里的 regex，是指 POSIX BRE 正则表达式，也即是 grep 和 sed 支持的正则表达式；而 ereregex 是指 POSIX ERE 正则表达式，是 egrep 和 awk 支持的正则表达式。

另，由于 CentOS 上 grep、sed、vi 和 awk 等命令都是 GNU 改版过的，在正则上更为丰富，比如“+”、“?”以及“\1”（反引用）的支持等。所以习惯了 GNU 命令的人可能会认为 Rsyslog 中 regex 的使用依然比较局限。

3.2.1.2.4　action

在之前的 rule 章节中提到一个 rule 中可以有多个 action，注意这些 action 必须分行写，

每个一行。没新 filter 的 action，要以“&”为开始。比如将严重性等级为 CRITICAL 的数据同时发送给 rger 用户、root 用户和/var/log/critmsgs 文件，写法如下。

```
*.=crit :omusrmsg:rger
& root
& /var/log/critmsgs
```

这种写法比下面这种写三行 rule 的方式性能要好。

```
*.=crit :omusrmsg:rger
*.=crit root
*.=crit /var/log/critmsgs
```

action 中最常见的就是保存成文本文件。这里我们可以利用 template 来自动分割日志。静态日志位置从“/”开始，而动态的日志位置以“?”开始。示例如下。

```
$template logFormat, "%rawmsg%\n"
$template DynaFile, "/var/log/%$YEAR%-%$MONTH%-%$DAY%.log"
user.info          -?DynaFile;logFormat
```

另一种常见的 action 就是转发到其他服务器。格式很简单，“@”表示使用 UDP 协议，“@@”表示使用 TCP 协议，“z*”表示使用某级别的 gzip 压缩传输，示例如下。

```
*.*    @@(z9)192.168.0.1:1470
```

rule 中提到了 discard，在 action 中，调用 discard 非常容易，就是一个“~”。比如下面这行意思就相当于 Rsyslogd 完全没运行。

```
*.*    ~
```

此外，action 还可以以“:”开头调用比如 ommysql 之类的插件，以“^”开头调用 shell 命令等。示例如下：

```
:ommysql:dbhost,dbname,dbuser,dbpassword;dbtemplate
^program-to-execute;template
```

shell 这里，Rsyslog 会把经过 template 时生成的结果传递给命令作为唯一参数。

3.2.1.3　收集应用日志

3.2.1.3.1　Apache 日志

Apache 本身支持通过 Syslog 发送错误日志，我们又可以通过 Pipe 形式，向 Syslog 发送访问日志。

Apache 配置如下。

```
    CustomLog logs/access_log combined
    CustomLog "|/usr/bin/logger -t httpd -p local6.info" combined
    ErrorLog syslog:local7
```

Rsyslog 配置如下。

```
    if $syslogfacility-text == 'local6' and $programname == 'httpd' then
/var/log/httpd-access.log
    if $syslogfacility-text == 'local6' and $programname == 'httpd' then ~
    if $syslogfacility-text == 'local7' and $programname == 'httpd' then
/var/log/httpd-error.log
    if $syslogfacility-text == 'local7' and $programname == 'httpd' then ~
```

3.2.1.3.2 Nginx 日志

Nginx 默认不支持 Syslog 或者 Pipe 方式，不过淘宝发布的 tengine 支持这两种方式，示例如下。

```
    logformat syslog "syslog:local7:info:192.168.0.2:514:nginxlog";
```

其中 nginxlog 将记录在 DEVNAME 字段。

此外，人人发布的 ngx_http_accounting_module 实现了针对带宽和状态码的 Syslog 统计输出。编译 Nginx 时使用--add-module=/path/to/ngx_http_accounting_module 参数即可。

在 Nginx 中只需要两行配置。

```
    http{
      http_accounting  on;
      server {
        http_accounting_id  server_string;
        location / {
    #     http_accounting_id  location_string;
       }
      }
    }
```

默认会每 10 秒钟发送统计数据到本机的 Syslog，然后在本机的 Rsyslog 中配置如下指令，即可在本机和远程接收到基于具体域名甚至具体 URL 的访问情况统计。

```
    $template    NADailyLog,"/var/log/NgxAccounting/%$YEAR%/%$MONTH%/%$YEAR%-
%$MONTH%-%$DAY%.log"
```

```
    $template SSFormat,"%timegenerated%||%msg%\n"

    if $programname == 'NgxAccounting' then ?NADailyLog;SSFormat
    if $programname == 'NgxAccounting' then @192.168.0.2;SSFormat
    if $programname == 'NgxAccounting' then ~
```

最终保存的日志示例如下。

```
    Apr  5 14:41:43|| pid:10295|from:1365144073|to:1365144103|accounting_id:
youdomain|requests:250|bytes_out:788668|200:237|404:11|403:2
```

3.2.2　message queue

Syslog 虽好，但不是没有缺点，具体如下。

1．运行在 UDP 模式下的 Syslog 是会丢数据的。

2．即使运行在 TCP 模式下解决了丢包的问题，Syslog 的 PRI 包括 TAG 的方式依然无法充分满足大多数情况下对不同业务不同数据的隔离需求。

这种情况下，消息队列的优势就体现出来了。类似 RabbitMq、ZeroMq、StoMp、Q4M 乃至 Redis 等，这些已经在业务线上广泛运用的 MQ 组件，也就顺势进入了运维系统的范畴。

消息队列，是软件工程领域，用以进程间，甚至线程间通信的组件，很多时候都是操作系统或者应用内部在使用。不过我们这里要说的，是计算机之间的、跨网络的、狭义的消息队列中间件。

消息队列提供的是一个异步通信协议，消息生成的一方和消费的一方不要求同步交互，而是将消息内容通过队列来进行转交，也就是说，队列本身需要具有存储的能力。

开源的消息队列中间件有很多，有名的就有 JBoss Messaging、ActiveMQ、Qpid，RabbitMQ、Beanstalkd、HTTPSQS、Q4M 等。

最早的消息队列中间件，Sun 公司的 JMS 规范只有 Java 的 API，可以让开发者像写 SQL 一样使用消息队列，不过目前这种做法已经不再主流，当前主流的消息队列中间件标准有三个，以下进行具体介绍。

3.2.2.1　Advanced Message Queuing Protocol（AMQP）

AMQP 与 JMS 的区别在于，AMQP 定义的是以 8 字节为单位的数据流格式。在 AMQP

中，数据的基本单位是帧。AMQP 中一共有九种帧：open、begin、attach、transfer、flow、disposition、detach、end 和 close。

数据传输以 attach 帧开始，detach 帧结束；消息通过 transfer 帧建连在一个唯一的 direction 上；flow 帧用来管理主题，方便订阅；消息两端的传输状态变化，则通过 disposition 帧来交互。

上面这 5 个帧类型都与数据传输相关，其他 4 个则偏重端点之间的连接。前面说到一个 transfer 帧是固定在一个 direction 上的。但是端点和端点之间，可以有多个 direction，这些 direction 合在一起叫 session，由 begin 和 end 帧来控制双向 session 的初始化和终止。更高一层，多个 session，还可以以多路复用的连接形式，各自独立地存在在相同的两个端点之间，这个连接，是由 open 和 close 帧控制的。

AMQP 的主要实现，有 RabbitMQ、Qpid、ActiveMQ 等，也是消息队列中间件的主流。

3.2.2.2 Message Queue Telemetry Transport（MQTT）

MQTT 定义传输的，是遥测型数据，主要用于低带宽的小型设备场合。MQTT 的实现中，大多数是 IBM 等厂商的商业化产品，如 WebSphere MQ 等。

3.2.2.3 Streaming Text Oriented Messaging Protocol（STOMP）

STOMP，顾名思义是基于文本的消息传输协议。它在 TCP 层上定义了一个类似 HTTP 的方法，数据传输一共包括十种：CONNECT、SEND、SUBSCRIBE、UNSUBSCRIBE、BEGIN、COMMIT、ABORT、ACK、NACK 和 DISCONNECT。

数据传输同样是以帧为单位的。不过 STOMP 的帧，其实就是多行文本。第一行是具体指令名，第二行开始是以 key: value 格式存在的 headers，和 HTTP 一样每个 header 一行。然后一个附加空行后是详细的内容体。最后以一个 NULL 字符结束。

服务器和客户端之间的通信，则通过另外三种帧完成，它们是 MESSAGE、RECEIPT 和 ERROR。帧格式和数据传输帧格式一样。

STOMP 协议内容非常简单，所以各动态语言都有自己对应的实现，ActiveMQ 和 RabbitMQ 也有相关插件。

三个标准的实现都和 HTTP 工作在同一个层面，即 TCP 基础上。不过也有一些实现，比如 Amazon 的 SQS，是在 HTTP 协议的基础上完成的。

除这些标准的消息队列中间件以外，还有两个产品必须一提。

一个是 ZeroMQ，该项目起初也是按照 AMQP 标准研发的，但最终放弃了 AMQP 而自行发展成现在这种更偏向网络通讯库的形式。日本知名的 PAAS 云计算厂商 Dotcloud 整个云中的消息传输，都是通过 ZeroMQ 来完成的。

另一个则是在 NoSQL 领域非常著名的 redis。redis 提供了比其前辈 Memcached 更加复杂的数据结构，在 key:value 基础上有了 list、set 和 zset。在 list 基础上自然可以建立队列，通过 zset 又能很轻松地完成优先级的概念，所以逐渐一些对标准 AMQP 的高级功能要求不多的项目，开始转向使用 Redis 来享受其简单和便捷。甚至有了 Resque 这样以 Redis 为基础封装完成的消息队列项目。

我们可以直接通过命令行来完成简单的 Redis 队列功能展示。开启 3 个终端窗口，在第 1 和第 2 个窗口中先启动接收命令，然后第二个中输入发送命令，如下。

```
# redis-cli publish topic testmsg
(integer) 2
```

在前两个窗口中可以看到如下输出。

```
# redis-cli subscribe topic
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "topic"
3) (integer) 1
1) "message"
2) "topic"
3) "testmsg"
```

可以看到，对于发送端返回成功接收的个数（2 个），接收端输出的是：消息类型（message）、订阅话题（topic）和具体内容（testmsg）。

Redis 在各语言中都有模块提供，开发使用代码也在十行以内。下面是 Perl 版本的示例。发送端如下。

```
use Redis;
my $redis = Redis->new(server => '127.0.0.1:6379');
$redis->publish('topic1', 'test4');
```

接收端如下。

```
use Redis;
```

```
my $redis = Redis->new;
$redis->psubscribe(
  'topic*',
  sub {
    printf "Msg: %s, This topic: %s, Subscribe topic: %s\n", @_;
  }
);
$redis->wait_for_messages(10) while 1;
```

从上手难度和灵活性考虑，建议运维人员都应该了解和使用一下 Redis。

3.2.3 RPC

采用 RPC 协议的运维系统产品目前不是很多。个人感觉是因为 RPC 通常用在有比较复杂的数据结构的场景中，日志数据还是扁平的字符串居多。当然，在超大规模的互联网企业中，数据的采集和传输不单单是普通的系统和访问日志，更重要的是处理海量的业务用户数据，这些内容产生自业务应用本身，天然具有较为复杂的数据结构。所以我们看到，在设计场景更为广泛的项目中，普遍会使用 Thrift 等服务开发框架。

Thrift 是 Facebook 开源到 Apache 社区的项目，具有完整的 RPC 实现，只需要定义好统一的数据结构配置，Thrift 会自助生成和网络通信相关的代码，而在各自节点上，可以使用 C++、Java、PHP、Python、Perl、Ruby、Javascript、Erlang、Golang、Smalltalk、HTML 等各种语言。

Thrift 的整体架构如图 3-2 所示。看起来比较复杂，不过通常情况下只需要自己写最顶层的“Your Code”部分，然后会由 thrift 命令生成“FooService”框架和内部的“Foo.write/read”方法，至于更下层的协议和 I/O 通信，就更不用时时关心了。

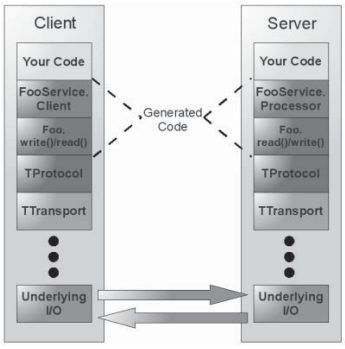

图 3-2

一个简单的 Thrift 配置如下。

```
struct CacheProfile {
    1:string key,
    2:string value
}
service CacheStorage {
    i32 cache_set(1:string key, 2: string value),
    string cache_get(1:string key)
}
```

然后一行命令就可以生成对应语言版本的代码框架。

```
# thrift --gen py example.thrift
# tree gen-py/
gen-py/
├── __init__.py
└── example
    ├── CacheStorage.py
    ├── CacheStorage-remote
    ├── constants.py
    ├── __init__.py
    └── ttypes.py

1 directory, 6 files
```

3.2.4　Gearman

Gearman 是一套分布式程序框架，可以用在各种场合，与 Hadoop 相比，Gearman 更偏向于任务分发功能。它的任务分发非常简单，简单得可以只需要用脚本就能完成。Gearman 最初用于 LiveJournal 的图片 resize 功能，由于图片 resize 需要消耗大量计算资源，因此需要调度到后端多台服务器执行，完成任务之后再返回前端呈现到界面。之后，Gearman 广泛运用于各种分布式场合，比如网络爬虫、邮件处理、软件测试甚至数据库分片代理。

本书之前讲述 Nagios 的分布式调优时，已经提到过 Gearman 在任务分发方面的应用。而在数据采集传输方面，Gearman 可以以一种类似 MQ 的方式，起到类似并且更灵活的作用。

3.2.4.1　Gearman 原理

一个 Gearman 请求的处理过程涉及三个角色：Client -> Job -> Worker。对于 Clinet 和

Worker 并不限制使用一样的语言，所以有利于多语言系统之间的集成。

- **Client**：请求的发起者，可以是 C、PHP、Perl、MySQL 和 UDF 等。
- **Job**：请求的调度者，用来负责协调把 Client 发出的请求转发给合适的 Worker。
- **Worker**：请求的处理者，可以是 C，PHP，Perl 等。

gearman 程序的通用流程如图 3-3 所示。

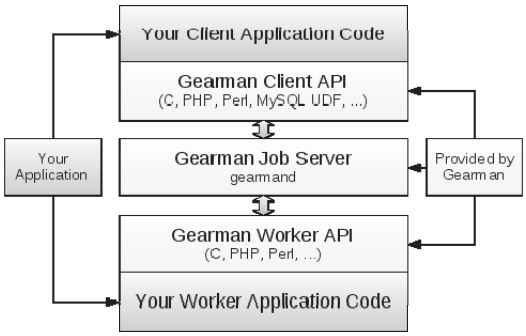

图 3-3

3.2.4.2 基础使用

Gearman 的初始版本是用 Perl 写的。不过目前已经改成 C 语言版本。所以最简单的使用方法是直接通过命令行来演示。

直接通过 apt 或者 yum 安装 gearman-job-server 或者 gearmand，然后我们可以看到在本机 4730 端口监听了 gearmand 程序。

现在我们就可以书写 Worker 了，如下。

```
# gearman -f example -w -- wc -l
```

然后另外启动一个 Client，这里传递一个文件内容如下。

```
# gearman -f example < /etc/passwd
47
```

就这么简单。

注意，这里传递的是/etc/passwd 的内容，而不是“/etc/passwd”这个字符串。为了验证这个事实，我们可以将 Client 命令运行在另外一台设备上。看看输出结果是不是和

Worker 相关，同时也验证 Gearman 的跨网络工作能力。

新 Worker 运行在 192.168.0.100 上，如下。

```
# gearman -h 192.168.0.2 -f example2 -w sh
```

新 Client 运行在 192.168.1.21 上，如下。

```
# echo 'ifconfig eth0|grep inet' | gearman -h 192.168.0.2 -f example2
      inet addr:192.168.0.100 Bcast:192.168.0.255 Mask:255.255.255.0
```

可以看到，192.168.1.21 上输出的是 192.168.1.100 的 IP 地址。

3.2.4.3　场景示例

在日志处理方面，常见的 Gearman 使用流程图如图 3-4 所示。

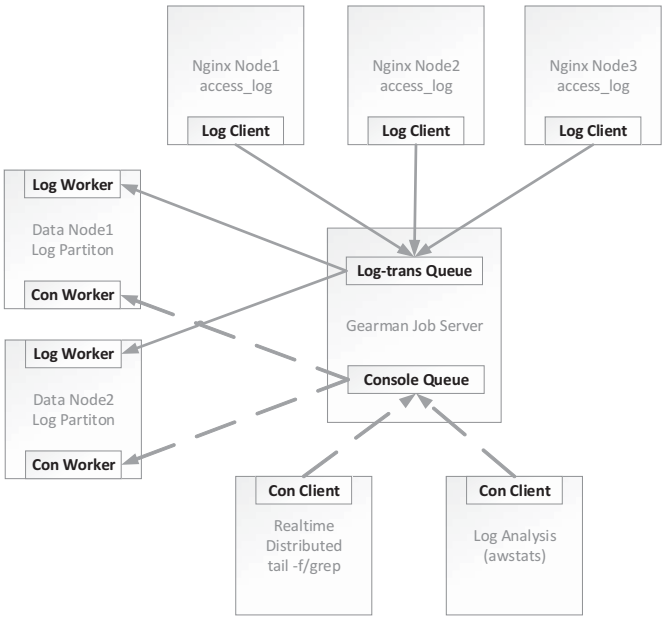

图 3-4

在海量日志实时处理方面，Gearman 最简单也最常见的一个运用，是**实时的访问排行**。

在单机上分析日志，我们都会使用如下命令来统计 Web 日志中访问量最大的前十个 URL。

```
awk '{a[$6]++}END{for(i in a){print a[i],i}}' access.log |sort -nr|head
```

那么在上千台服务器的集群上，如何快速地在几分钟内统计出这些数据呢？

这里有一种近似模拟的 MR 统计方法，如下。

在每台日志存储（甚至就是每台日志产生）的机器上我们先计算各自的排名前 100 的 URL，然后通过 Gearman 汇总到一起，再做最终的前 10 排序。

一般来说，这种近似计算的结果，在这个场景下已经足够精确。

3.3 日志收集系统框架

有了上节介绍的各种传输协议和工具，我们已经可以简单地通过一些 Shell 和 Perl 脚本，做到自动化数据收集传输了。但是随着数据来源的种类越来越多，或者传输后的分析平台越来越多，自己动手写脚本会是一件越来越不够自动化的麻烦事情。这时候，我们需要一些足够智能的框架，来替我们完成这个工作。

事实上，近几年来，各大互联网公司和主要开源组织陆续都公布了自己的数据收集传输处理框架，比较著名的有 Twitter 的 Storm、Linkedin 的 kafka、Facebook 的 Scribe、Cloudera 的 Flume 和 Hadoop 的 Chukwa 等。

不过，以上项目大多是由开发人员结合自身环境逐步改进出来的，大多数有以下两个特点，让运维工程师比较难以上手。

- 使用 Java、Scala 语言编写。而运维人员更多使用和熟悉的是 PHP、Python、Perl 等解释性语言；
- 依赖 Hadoop 生态环境。这些框架大多需要提供后期的处理分析功能，基本上都采用了 HDFS 存储数据、Map/Reduce 处理、ZooKeeper 保证高可用的一套办法。这导致整个框架结构相当复杂，初始规模也异常庞大，运维成本比较高。

下面介绍两个类似的系统。这两位同样系出名门，却较好地降低了运维入手的难度，同时提供了丰富的功能。

3.3.1 Flume-ng

Flume-ng 是在原先 Flume 的基础上发展而来的。它去除了原先架构中复杂的

ZooKeeper 高可用等组件，形成了目前这个比较轻量的结构。

Flume-ng 主要包括 source，sink 和 channel 三个部分。当 source 来自多个源地址或者服务时，就是通常所说的集中式日志收集；当 source 链接的上游是前一个 Flume-ng 的 sink 时，意味着这是一个数据中继的过程。这样，Flume-ng 就可以跨越网络，一级一级地把整个数据流串联起来。其流程图如图 3-5 所示。这也是 Flume-ng 最常见的用法。

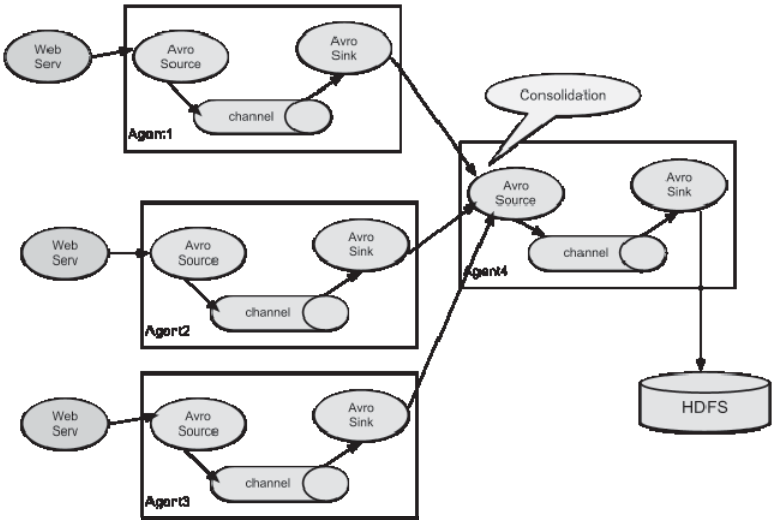

图 3-5

反过来，如果单一的 source 对应的是多个 channel 和 sink，如图 3-6 所示，就可以把一份数据复制到多个目的地址或者服务上，以完成不同目的的分析处理。

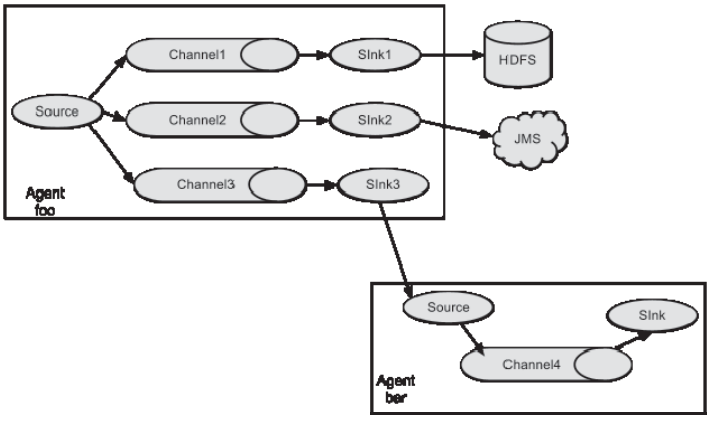

图 3-6

下面是一份简单的 Flume-ng 配置文件示例。

```
    # Name the components on this agent
    a1.sources = r1
    a1.sinks = k1
    a1.channels = c1

    # Describe/configure the source
    a1.sources.r1.type = netcat
    a1.sources.r1.bind = localhost
    a1.sources.r1.port = 44444
    agent.sources.r1.interceptors.i1.regex = (\\d):(\\d):(\\d)
    agent.sources.r1.interceptors.i1.serializers = s1 s2 s3
    agent.sources.r1.interceptors.i1.serializers.s1.name = one
    agent.sources.r1.interceptors.i1.serializers.s2.name = two
    agent.sources.r1.interceptors.i1.serializers.s3.name = three

    # Describe the sink
    a1.sinks.k1.type = logger

    # Use a channel which buffers events in memory
    a1.channels.c1.type = memory
    a1.channels.c1.capacity = 1000
    a1.channels.c1.transactionCapacity = 100

    # Bind the source and sink to the channel
    a1.sources.r1.channels = c1
    a1.sinks.k1.channel = c1
```

以上配置显得比较简洁易懂，相信即便是没有接触过 Flume-ng 的读者也可以猜测出这个配置的效果，那就是：

在本机开启一个 44444 端口接收数据；当数据中匹配[0-9]:[0-9]:[0-9]时，将捕获的三个数字分别存入结构化的{"one":1,"two":2,"three":3}数据；最后输出到系统日志中。

示例运行如下。

```
    # bin/flume-ng agent --conf conf/ --conf-file conf/example.conf --name a1
-Dflume.root.logger=INFO,console
    # echo '[9:8:7:6:5:4] 321 flume test' | nc 127.0.0.1 44444
    2013-05-09  16:49:45,149  (SinkRunner-PollingRunner-DefaultSinkProcessor)
[INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)] Event:
```

```
{ headers:{} body: 5B 39 3A 38 3A 37 3A 36 3A 35 3A 34 5D 20 33 32 [9:8:7:6:5:4] 32 }
```

3.3.2　logstash

比上一节的 Flume-ng 更进一步，下面介绍一个非 Java 的项目。

Logstash 是由 Loggy（前 Splunk 员工构建在亚马逊 AWS 云基础上的世界上最大的第三方日志分析处理服务商）贡献的开源项目。其官网地址是 http://logstash.net/。

Logstash 由 JRuby 语言编写。而事实上，除开一小部分使用 Joda 库处理时间的逻辑，以及一个采用 RPC 协议连接 ElasticSearch 的插件，其余所有代码都可以在 MRI 上运行。而使用标准的 Ruby 库 DateTime 作为替换，也非常容易。笔者就一直使用 MRI 的 1.8.7 和 1.9.3 版本分别运行着 Logstash。

因为是采用 Ruby 编写的，所以 Logstash 社区非常活跃，现在已经拥有接近 100 个配套插件。与 Flume-ng 或 Rsyslog 的结构类似，Logstash 同样由 input、filter 和 output 三个部件组成。绝大多数运维人员都可以从它庞大的插件库中找到自己擅长和熟悉的组件，来尽快搭建完成 Logstash 系统。Logstash 全部插件列表如下。

- **input 插件列表：**amqp、drupal_dblog、eventlog、exec、file、ganglia、gelf、gemfire、generator、heroku、irc、log4j、lumberjack、pipe、redis、relp、sqs、stdin、stomp、syslog、tcp、twitter、udp、xmpp、zenoss、zeromq。
- **filter 插件列表：**alter、anonymize、checksum、csv、date、dns、environment、gelfify、geoip、grep、grok、grokdiscovery、json、kv、metrics、multiline、mutate、noop、split、syslog_pri、urldecode、xml、zeromq。
- **output 插件列表：**amqp、boundary、circonus、cloudwatch、datadog、elasticsearch、elasticsearch_http、elasticsearch_river、email、exec、file、ganglia、gelf、gemfire、graphite、graphtastic、http、internal、irc、juggernaut、librato、loggly、lumberjack、metriccatcher、mongodb、nagios、nagios_nsca、null、opentsdb、pagerduty、pipe、redis、riak、riemann、sns、sqs、statsd、stdout、stomp、syslog、tcp、websocket、xmpp、zabbix、zeromq

3.3.2.1　快速构建

Logstash 社区非常重视新人的上手难度问题，甚至有一句话叫做“让新手为难的问题

那就是个 bug”。所以使用 Logstash 非常容易。

a．最简单的使用

我们从标准输入 stdin 字符串“testdata”，让 Logstash 转换到标准输出。

```
    $ wget https://logstash.objects.dreamhost.com/release/logstash-1.1.9-
monolithic.jar
    $ yum install java-1.7.0-openjdk
    $ java -jar logstash-1.1.9-monolithic.jar agent -e 'input { stdin { type =>
"stdin-type"}}output { stdout {}}'
    testdata
    2013-04-07T13:20:36.806Z stdin://localhost/: testdata
```

然后我们再给 stdout 加上 debug_format => "json"参数，可以看到输出如下。

```
    {
    "@source":"stdin://localhost/",
    "@tags":[],
    "@fields":{},
    "@timestamp":"2013-04-07T13:22:07.460Z",
    "@source_host":"localhost",
    "@source_path":"/",
    "@message":"testdata",
    "@type":"stdin-type"
    }
```

如果我们再给 agent 上加上-vv 参数，可以看到更多 Logstash 内部的流程。可以发现，在 Logstash 内部，是**以 event 的形式在流转的**。Logstash 中不宜使用“行”这个概念，正是因为它会将数据变成 json，整个 event 都是在处理 json。尤其是，有 filter 可以把多行合并成一个 json，由一个 event 来流转下去。

b．集群中的典型使用

Logstash 最典型的使用场景，是通过 Logstash 收集应用集群的日志到专用服务器上，并进行过滤和处理。其流程如图 3-7 所示。

只要同时有 input 和 output 的插件，都可以作为 broker 运行，比如 AMQP、Redis、ZeroMQ、Stomp、TCP 等。Logstash 早先推荐使用 RabbitMQ，但是因为 RubyGems 相关模块的支持不力，在大规模使用时经常成为瓶颈所在。所以从 1.1.5 版本往后，官方推荐已经改为 Redis 了。

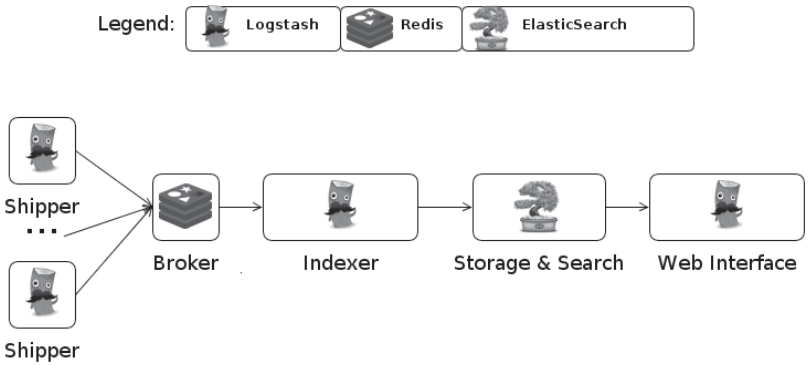

图 3-7

shipper 配置如下。

```
input {
  file {
    path => "/var/log/*.log"
    type => "clusterlog"
  }
}
output {
  redis {
   host => "192.168.0.2"
   data_type => "list"
   key => "logstash"
  }
}
```

indexer 配置如下。

```
input {
  redis {
    host => "192.168.0.2"
    type => "clusterlog"
    data_type => "list"
    key => "logstash"
    format => "json_event"
  }
}
output {
  elasticsearch {
    host => "192.168.0.3"
```

```
    }
}
```

broker 配置如下。

```
yum install redis-server
```

storage 配置如下。

Logstash 默认自带有一个内嵌的 ElasticSearch 服务器。在小规模场景下，storage 部分完全无须设置。在规模较大的情况下，才需要自行进行分布式 ES 集群部署。关于 ES 的原理、配置和运维优化，稍后会将有专门的存储章节来讲述。

3.3.2.2 Grok 的运用

到上节为止，Logstash 似乎并没有体现出什么特别的地方，现在介绍的就是 Logstash 最强大的功能——Grok filter。

Grok 是一个数据结构化工具。只需要通过简单地变量定义，我们就可以将文本格式的字符串，转换成为具体的结构化的数据。Logstash 默认带有上百个 Grok 变量，我们可以直接使用或者稍微改写成自己的新 Grok 变量。

比如下面一条默认的 Apache 日志。

```
10.2.21.130 - - [08/Apr/2013:11:13:40 +0800] "GET /mediawiki/load.php HTTP/1.1" 304 - "http://som.d.xiaonei.com/mediawiki/index.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/536.28.10 (KHTML, like Gecko) Version/6.0.3 Safari/536.28.10"
```

我们使用下面这行 filter 来转换。

```
filter {
       grok {
          pattern => "%{COMBINEDAPACHELOG}"
          type => "apache"
       }
}
```

得到数据结构如下。

```
{
  "@source":"file://chenryn-Lenovo/home/chenryn/test.txt",
  "@tags":[],
```

```
      "@fields":{
        "clientip":["10.2.21.130"],
        "ident":["-"],
        "auth":["-"],
        "timestamp":["08/Apr/2013:11:13:40 +0800"],
        "verb":["GET"],
        "request":["/mediawiki/load.php"],
        "httpversion":["1.1"],
        "response":["304"],
        "referrer":["\"http://som.d.xiaonei.com/mediawiki/index.php\""],
        "agent":["\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/
536.28.10 (KHTML, like Gecko) Version/6.0.3 Safari/536.28.10\""]
      },
      "@timestamp":"2013-04-08T03:34:37.959Z",
      "@source_host":"chenryn-Lenovo",
    "@source_path":"/home/chenryn/test.txt",
      "@message":"10.2.21.130  -  -  [08/Apr/2013:11:13:40  +0800]  \"GET
/mediawiki/load.php HTTP/1.1\" 304 - \"http://som.d.xiaonei.com/mediawiki/
index.php\" \"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/536.28.
10 (KHTML, like Gecko) Version/6.0.3 Safari/536.28.10\"",
      "@type":"apache"
    }
```

在随后的配置和其他处理中，我们就可以随意使用这个数据结构中的内容作为变量处理了，比如“%{clientip}”等。

默认内置变量列表见：https://github.com/logstash/logstash/tree/v1.1.9/patterns。

自己定义时，基本可以在这些变量基础上进行，比如 Nginx 的访问日志，可以定义成以下这样。

```
    NGINXURI %{URIPATH}(?:%{URIPARAM})*
    NGINXACCESS \[%{HTTPDATE}\] %{NUMBER:code:int} %{IP:client} %{HOSTNAME}
%{WORD:method} %{NGINXURI:req} %{URIPROTO}/%{NUMBER:version} %{IP:upstream}(:
%{POSINT:port})? %{NUMBER:upstime:float} %{NUMBER:reqtime:float} %{NUMBER:size:
int} "(%{URIPROTO}://%{HOST:referer}%{NGINXURI:referer}|-)" %{QS:useragent}
"(%{IP:x_forwarder_for}|-)"
```

3.3.2.3　结合 Kibana 快速完成全文搜索和统计功能

上例中，我们只提到如何使用 output 到存储中，现在讲如何提供更高级而且方便的搜

索报表功能。Logstash 默认的 WebUI 比较简单，社区基本都在使用另一个，即同样几乎是开箱即用的 Kibana 项目，来完成 Web interface 组件。

项目官网：http://kibana.org/。

Kibana 是一个标准的 Rack 项目，下载源码，安装依赖模块，修改 KibanaConfig.rb 中的 ElasticSearch 服务器 IP 地址，就可以运行了，过程如下。

```
git clone git://github.com/rashidkpc/Kibana.git
cd Kibana
bundle install
ruby kibana.rb
```

Kibana 会自动搜索 ES 服务器中以“logstash-%y.%m.%d”命名的索引，使用者只需要访问 http://localhost:5601 即可。访问效果如图 3-8 所示。

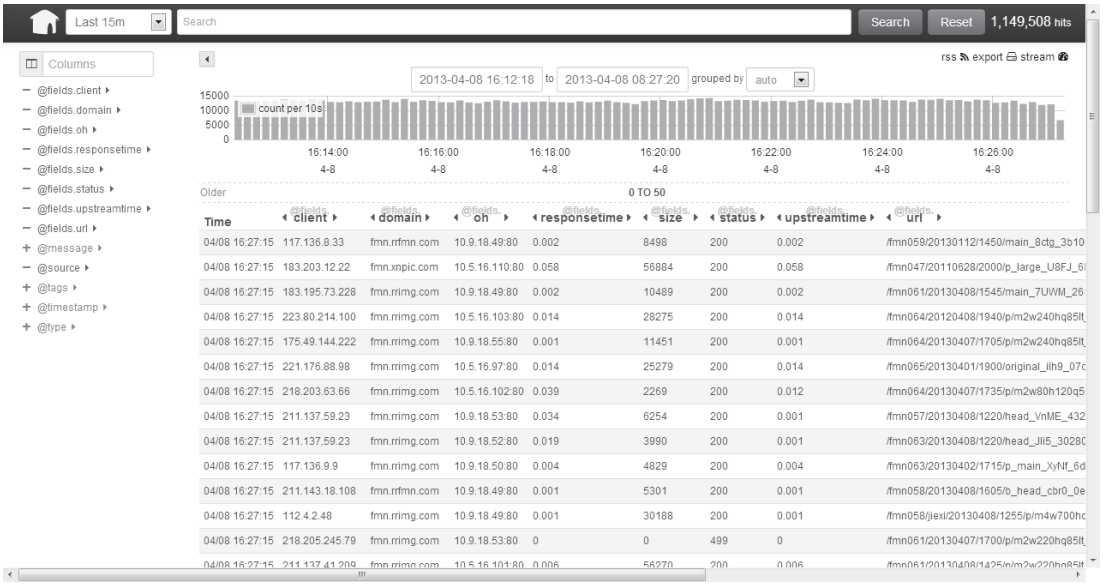

图 3-8

在单个 field 上可以点击菜单，选择查看该列的比例分析（Terms）、发展趋势（Trend）和算术统计（Stats）。如图 3-9 所示。

> 注意：Terms 只能针对字符串格式的列，而 Stats 只能针对数值格式的列。

在图 3-9 所示各 fields 中，responsetime、upstreamtime、size 属于数值格式；url、domain

等属于字符串。

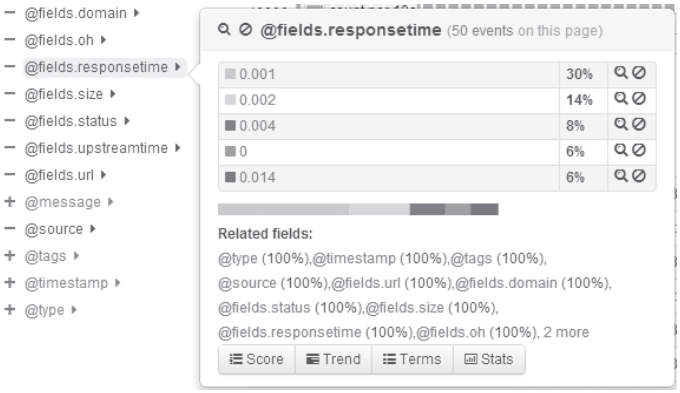

图 3-9

首先查看响应时间的统计情况，如图 3-10 所示。

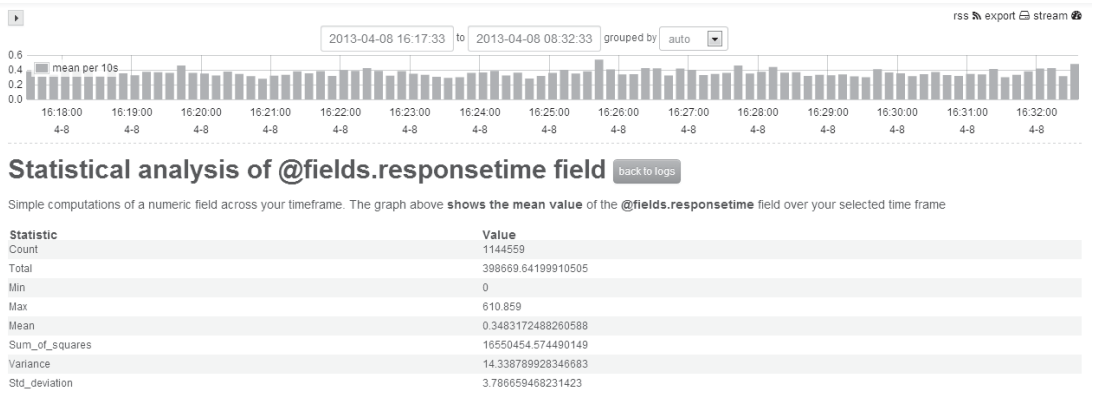

图 3-10

可以看出，最近 15 分钟内，一共完成了 1144559 次响应，平均响应时间为 0.348 秒，最大响应时间为 610 秒，方差是 14.33，每十秒的均值走势说明服务性能没有太大波动。

然后查看比例分析情况。目前比例分析没有跨多个索引进行，如果按 Logstash 的默认配置，应该是每天自动分割索引，也就是只能查看当天的比例，如图 3-11 所示。

还有趋势分析。目前趋势分析，是从所选择时间段（默认是最近 15 分钟）的开始和结束之间选择 2000 个 event 进行对比，如图 3-12 所示。

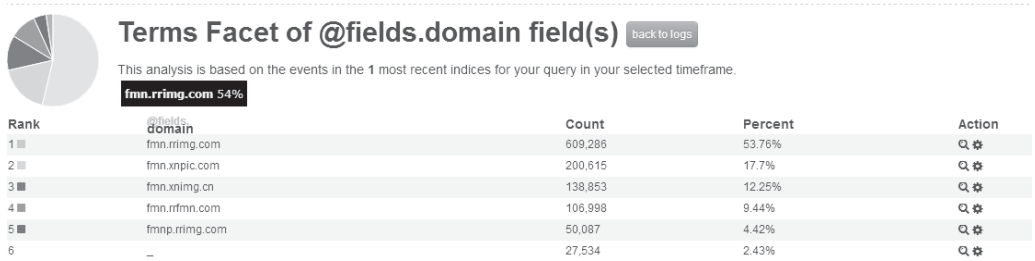

图 3-11

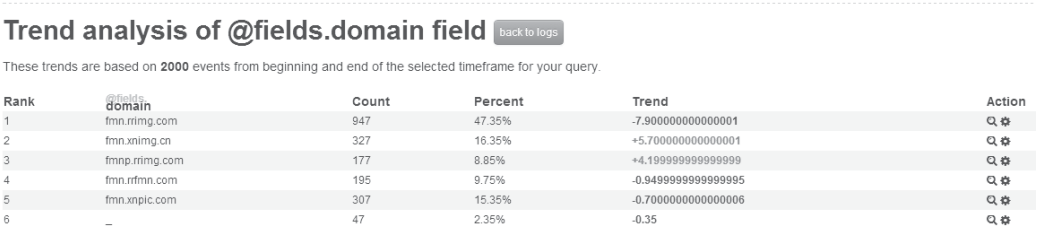

图 3-12

以上分析，都是搜索全部信息的结果。在页面顶端还有搜索栏，我们可以通过它先缩小请求范围。

熟悉 Lucene 的读者，可以直接输入 Lucene DSL。如果不熟悉，通过单击在 fields 列上的图标，也可以完成一些请求。比如图 3-13 为指定单一域名、单一源站的前提下，对来访 IP 的统计结果。

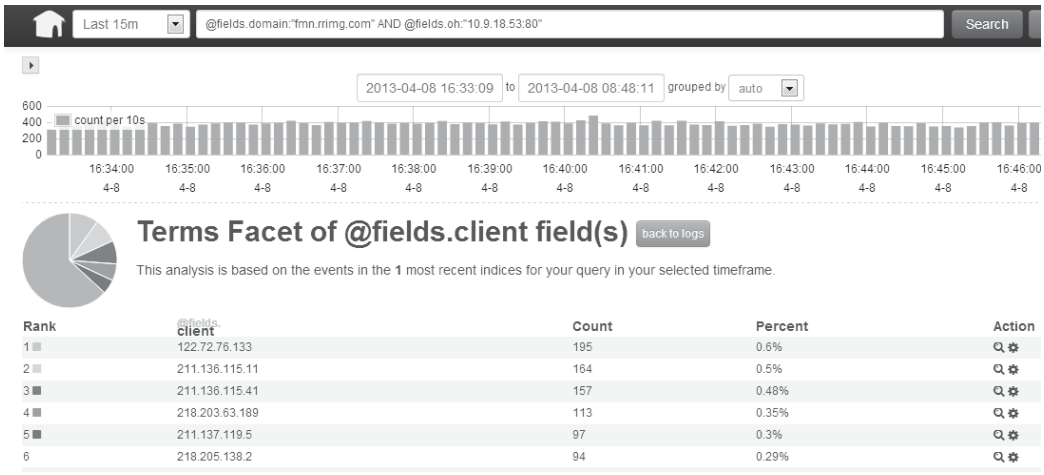

图 3-13

关于 Logstash 存储在 ElasticSearch 中的数据，如果想要得到比 Kibana 更灵活的处理手段和展示结果，请阅读之后的数据存储展示章节。

3.3.2.4　结合 graphite 快速完成数值展示功能

不是所有的数据都需要处理成复杂的结构化数据并提供全面的分析统计功能。有时候，我们更在乎的是其中某一个数值的发展，或者对某个关键词的触发。这时我们就没有必要启用 ElasticSearch+Kibana，而完全可以用其他插件完成。

下面以一个实际用例来说明 Logstash 是如何整合各种运维工具的。

CDN 集群的小文件业务服务器，采用 COSS 文件系统。对 squid 有了解的读者们都知道，COSS 的 rebuild 耗时很长，动辄几个小时。我们需要掌握集群各服务器的 rebuild 速度，并在每个机器完成 rebuild 后通知运维人员。

需求并不复杂，稍微熟练的运维人员都可以通过 while true 或者 crontab 运行的方式完成这个监控脚本，最多在收集展示方面可能稍微复杂一点。但是如果通过 Logstash 来完成，整个配置会非常的简洁易懂。

```
    input {
        file {
            path => "/var/squid/logs/cache.log"
            type => "squid-error"
        }
    }
    filter {
        grok {
            match  =>  ["@message",  "COSS:  /%{WORD:disk}/stripe:  Rebuilding
\(%{NUMBER:pct}"]
            type => "squid-error"
            add_tag => ["rebuild-report"]
        }
    }
    output {
        graphite {
            host => '10.2.21.100'
            type => "squid-error"
            metrics => ["Rebuild/%{@source_host}/%{disk}", "%{pct}"]
            tags => ["rebuild-report"]
        }
```

```
        email {
            match => [ "rebuild", "%{pct},100" ]
            to => "chenlin.rao@renren-inc.com"
            tags => ["rebuild-report"]
        }
    }
```

3.3.2.5 修改 Nginx 日志格式降低 Grok 负载

之前我们一直在强调：Logstash 的流程中，是以 json 形式来流转一个 event 的。但是我们知道，Linux 程序交互是基于文本流的（UNIX 哲学三大原则之一），所以在 Logstash 的 input 中，很重要的一件事情就是在转换文本成 JSON 格式。

默认情况下，input 和 file 插件（也包括 TCP、Pipe、Stdin 等），会将输入文本保存在 json 里的@messages 键值对中。然后再通过 filter、grok 等办法从@messages 中生成@fields 和@tags。具体例子在之前已经给出过。

这一系列动作，都是比较消耗 CPU 资源的。像 Nginx 访问日志这类数量又大、格式又复杂的 event，单核 CPU 的利用率经常会跑到 50%以上。在生产业务服务器上，因为运维需求导致这么大的负载，不是什么好事。所以我们应该想办法降低这个压力。

Logstash 除文本格式以外，还支持另外两种方式的输入，叫做 json 和 json_event，即直接传递 JSON 数据。Logstash 不用再解析文本和格式化操作，直接流转事件。这样可以大大降低 CPU 的使用。

把 Nginx 默认的日志格式稍微做一下改造，变成下面这样的定义。

```
    logformat json '{"@timestamp":"$time_iso8601",'
                '"@source":"$server_addr",'
                '"@fields":{'
                '"client":"$remote_addr",'
                '"size":$body_bytes_sent,'
                '"responsetime":$request_time,'
                '"upstreamtime":$upstream_response_time,'
                '"oh":"$upstream_addr",'
                '"domain":"$host",'
                '"url":"$uri",'
                '"status":"$status"}}';
    access_log /data/nginx/logs/access.json json;
```

然后给原先的 Logstash 配置文件加一行格式设置，如下。

```
input {
    file {
        type => "nginx"
        format => "json_event"
    }
}
output {
    amqp {
        type => "nginx"
        host => "10.10.10.10"
        key  => "cdn"
        name => "logstash"
        exchange_type => "direct"
    }
}
```

这样我们就可以省略掉复杂和相对低效的 Grok filter，直接完成将同样格式的数据写入 ES，过滤报警等动作了。