# leancloud/satori:

Satori 是一个 LeanCloud 维护的监控系统，inspired by Open-Falcon

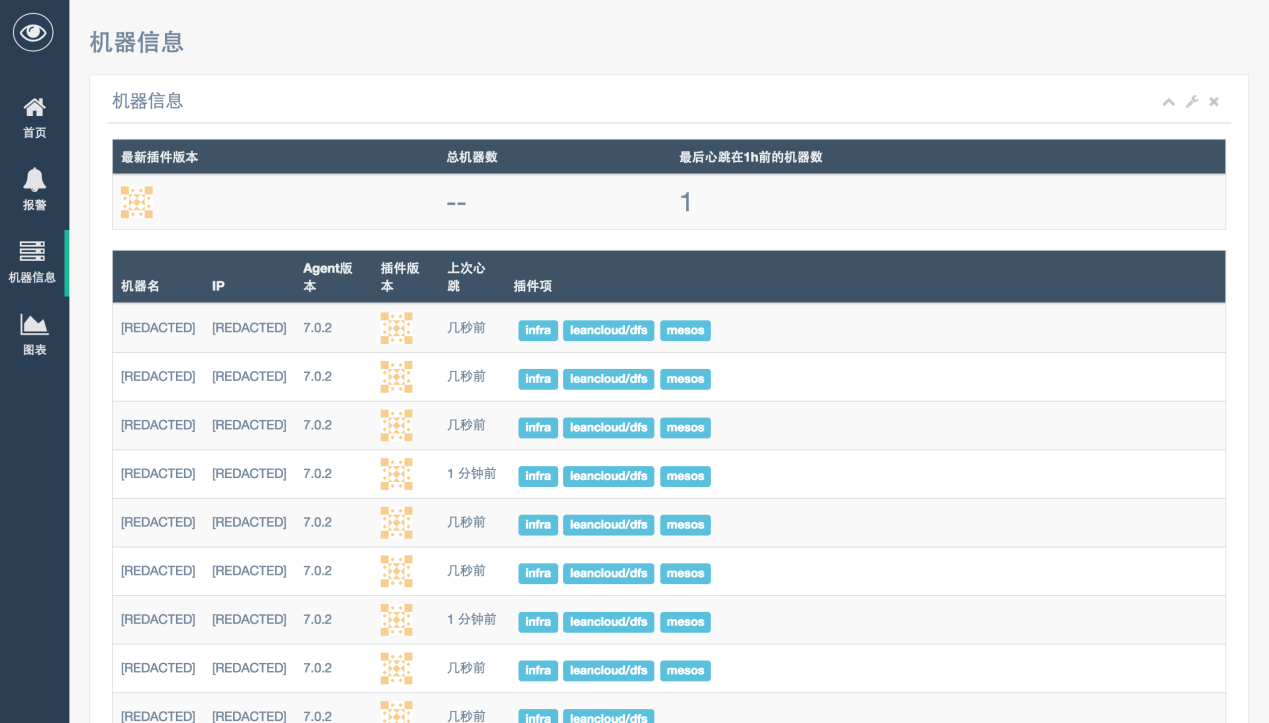
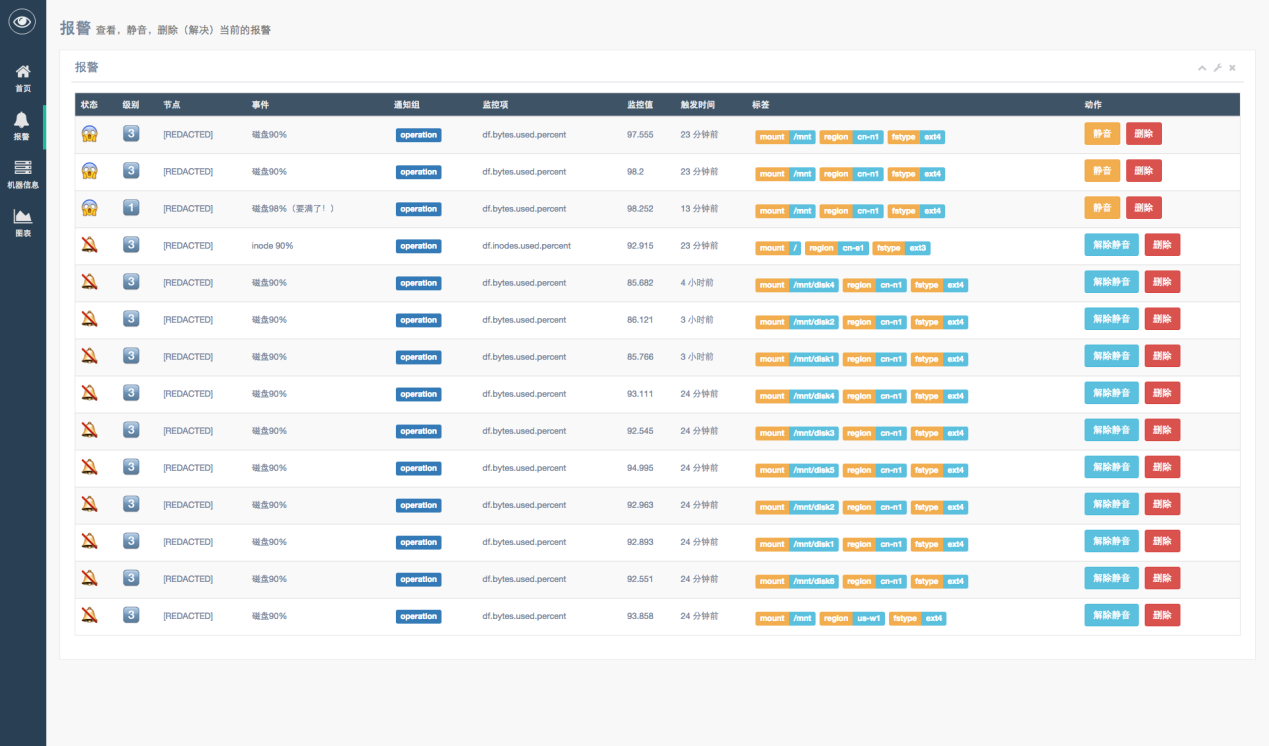
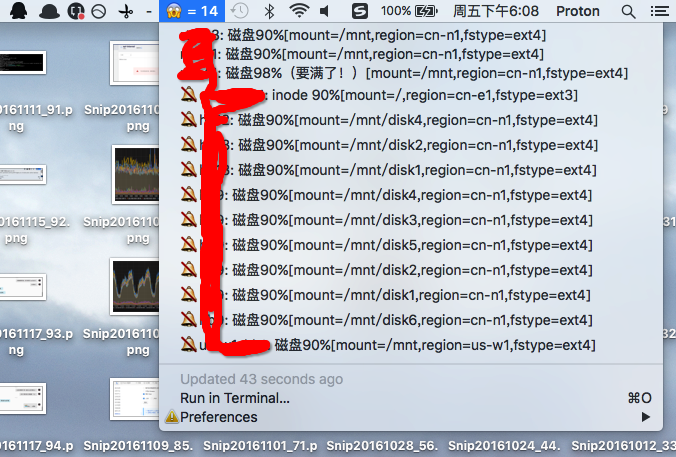
https://github.com/leancloud/satori

介绍

Satori 是一个由 LeanCloud 维护的监控系统。

起初在 LeanCloud 内部是直接使用了 [Open-Falcon](http://open-falcon.org/) 。 后续的使用过程中因为自己的需求开始做修改，形成的现在的这样的结构。

截图

[](https://raw.githubusercontent.com/leancloud/satori/master/doc/images/nodes.png) [](https://raw.githubusercontent.com/leancloud/satori/master/doc/images/events.png) [](https://raw.githubusercontent.com/leancloud/satori/master/doc/images/bitbar.png)

交流

如果你有什么建议，欢迎在 issue 里面说一下。 也有 QQ 群可以参与讨论：554765935

怎么安装？

请找一个干净的虚拟机，内存大一点，要有 SSD，不要部署其他的服务。 LeanCloud 用了一个 2 cores, 8GB 的内存的 VM，目测这个配置可以撑住大概 2000 个左右的节点（InfluxDB 查询不多的情况下）。

执行下面的命令：

$ git clone https://github.com/leancloud/satori

$ cd satori/satori

$ ./install

satori 子目录中有一个 install 脚本，以一个可以 ssh 并且 sudo 的用户执行它就可以了。 不要用 root，因为安装后会以当前用户的身份配置规则仓库。

install 时会问你几个问题，填好以后会自动的生成配置，build 出 docker image 并启动。这之后设置好DNS和防火墙就可以用了。 在仓库的第一层会有编译好的 satori-agent 和 agent-cfg.json 可以用来部署，命令行如下：

/usr/bin/satori-agent -c /etc/satori-agent/agent-cfg.json

agent 不会自己 daemonize，如果用了传统的 /etc/init.d 脚本的方式部署，需要注意这个问题。

如果需要无人工干预安装，请创建一个配置文件：

USE\_MIRROR=1 # 或者 0，是否使用国内的镜像

DOMAIN="www.example.com" # 外网可以访问的域名

INTERNAL\_DOMAIN="satori01" # 内网可以访问的域名

RULES\_REPO="/home/app/satori-conf" # 规则仓库的地址

RULES\_REPO\_SSH="app@www.example.com:/home/app/satori-conf" # 外网可以访问的 git 仓库地址

保存为 /tmp/install.conf。

然后执行：

$ git clone https://github.com/leancloud/satori

$ cd satori/satori

$ ./install -f /tmp/install.conf

设计思路

* Satori 希望最大程度的减少监控系统的部署维护难度。如果在任何的部署、增删维护报警的时候觉得好麻烦，那么这是个 bug。
* 监控时的需求很多样，Satori 希望做到『让简单的事情简单，让复杂的事情可能』。常用的监控项都会有模板，可以用 Copy & Paste 解决。略复杂的监控需求可以阅读 [riemann 的文档](http://riemann.io/)，来得知怎么编写复杂的监控规则。
* 因为 LeanCloud 的机器规模不大，另外再加上现在机器的性能也足够强劲了，所以放弃了 Open-Falcon 中横向扩展目标。如果你的机器数量或者指标数目确实很大，可以将 transfer、 InfluxDB、 riemann 分开部署。这样的结构支撑 5k 左右的节点应该是没问题的。
* 在考察了原有的 Open-Falcon 的架构后，发现实质上高可用只有 transfer 实现了，graph、judge 任何一个实例挂掉都会影响可用性。对于 graph，如果一个实例挂掉的话，还必须要将那个节点恢复，不能通过新加节点改配置的方式来恢复；judge 尽管可以起新节点，但是还是要依赖外部的工具来实现 failover，否则是需要修改 transfer 的配置的。因此 Satori 中坚持用单点的方式来部署，然后通过配合公网提供的 APM 服务保证可用性。真的希望有高可用的话，riemann 和 alarm 可以部署两份，通过 keepalived 的方式来实现，InfluxDB 可以官方的 Relay 来实现。

1 minute taste

简单需求（通用模板，Copy & Paste 改参数可以实现）

完整版请看 satori-rules/rules/infra/mongodb.clj

(def infra-mongodb-rules

; 在 mongo1 mongo2 mongo3 ... 上做监控

(where (host #"^mongo**\d**+")

; 执行 mongodb 目录里的插件（里面有收集 mongodb 指标的脚本）

(plugin-dir "mongodb")

; 每 30s 收集 27018 端口是不是在监听

(plugin-metric "net.port.listen" 30 {:port 27018})

; 过滤一下 mongodb 可用连接数的 metric（上面插件收集的）

(where (service "mongodb.connections.available")

; 按照 host（插件中是 endpoint）分离监控项，并分别判定

(by :host

; 报警在监控值 < 10000 时触发，在 > 50000 时恢复

(set-state-gapped (< 10000) (> 50000)

; 600 秒内只报告一次

(should-alarm-every 600

; 报告的标题、级别（影响报警方式）、报告给 operation 组和 api 组

(! {:note "mongod 可用连接数 < 10000 ！"

:level 1

:groups [:operation :api]})))))

; 另一个监控项

(where (service "mongodb.globalLock.currentQueue.total")

(by :host

(set-state-gapped (> 250) (< 50)

(should-alarm-every 600

(! {:note "MongoDB 队列长度 > 250"

:level 1

:groups [:operation :api]})))))))

cd /path/to/rules/repo # 规则是放在 git 仓库中的

git commit -a -m 'Add mongodb rules'

git push # 然后就生效了

复杂需求

这是一个监控队列堆积的规则。 队列做过 sharding，分布在多个机器上。 但是有好几个数据中心，要分别报告每个数据中心队列情况。 堆积的定义是：在一定时间内，队列非空，而且队列元素数量没有下降。

提示：这是一个简化了的版本，完整版可以看 satori-rules/rules/infra/kestrel.clj

(def infra-kestrel-rules

; 在所有的队列机器上做监控

(where (host #"kestrel**\d**+$")

; 执行队列相关的监控脚本（插件）

(plugin-dir "kestrel")

; 过滤『队列项目数量』的 metric

(where (service "kestrel\_queue.items")

; 按照队列名和数据中心分离监控项，并分别判定

(by [:queue :region]

; 将传递下来的监控项暂存 60 秒，然后打包（变成监控项的数组）再向下传递

(fixed-time-window 60

; 将打包传递下来的监控项做聚集：将所有的 metric 值都加起来。

; 因为队列监控的插件是每 60 秒报告一次，并且之前已经按照队列名和数据中心分成子流了，

; 所以这里将所有 metric 都加起来以后，获得的是单个数据中心单个队列的项目数量。

; 聚集后会由监控项数组变成单个的监控项。

(aggregate +

; 将传递下来的聚集后的监控项放到滑动窗口里，打包向下传递。

; 这样传递下去的，就是一个过去 600 秒单个数据中心单个队列的项目数量的监控项数组。

(moving-event-window 10

; 如果已经集满了 10 个，而且这 10 个监控项中都不为 0 （队列非空）

(where (and (>= (count event) 10)

(every? pos? (map :metric event)))

; 再次做聚集：比较一下是不是全部 10 个数量都是先来的小于等于后来的（是不是堆积）

(aggregate <=

; 如果结果是 true，那么认为现在是有问题的

(set-state (= true)

; 每 1800 秒告警一次

(should-alarm-every 1800

; 这里 outstanding-tags 是用来区分报警的，

; 即如果 region 的值不一样，那么就会被当做不同的报警

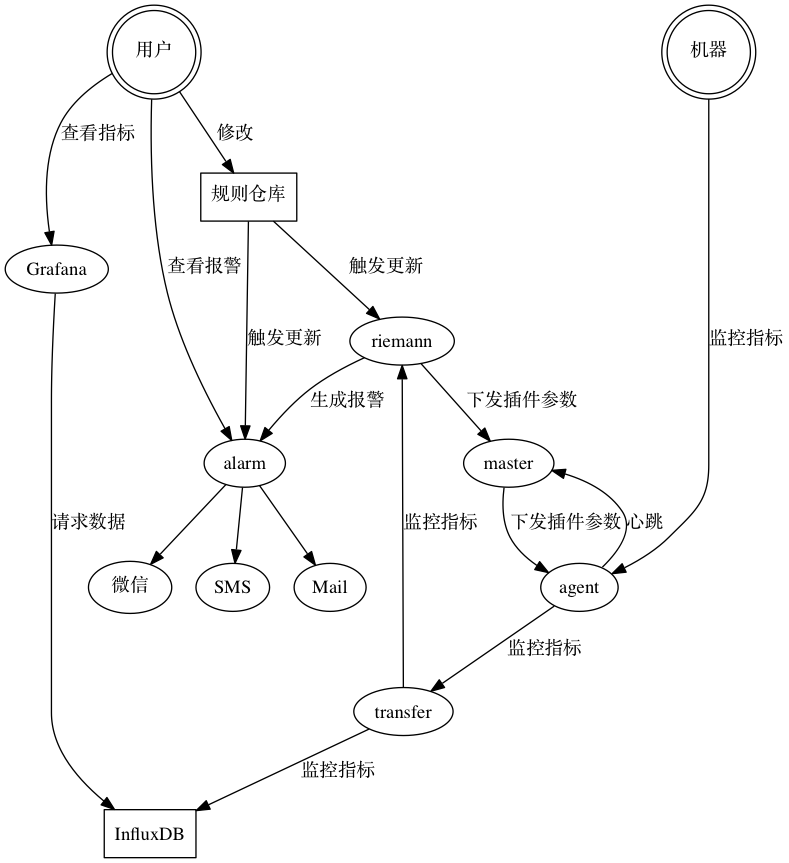
(! {:note #(str "队列 " (:queue %) " 正在堆积！")

:level 2

:outstanding-tags [:region]

:groups [:operation]}))))))))))))

架构

[](https://raw.githubusercontent.com/leancloud/satori/master/doc/images/arch.png)

与 Open-Falcon 的区别

agent

* 支持按照正则表达式排除 metric（用例：排除 docker 引入的奇怪的挂载，netns 什么的）
* 支持从 agent 上附加固定的 tag（用例：region=cn-west）
* 支持自主的插件更新
* 支持带参数的插件
* 去除了 push 和 plugin 以外的所有 http 接口
* 去掉了单机部署的功能，现在强制要求指定一个 transfer 组件
* 不兼容 open-falcon 的 heartbeat
* 因为修改了 metric 的数据结构，与 open-falcon 的 transfer 不兼容

transfer：

* 支持发送到 influxdb（使用了 hitripod 的补丁）
* 支持发送到 riemann
* 支持发送到 transfer（gateway 功能集成）(TODO)
* 不再支持发送到 graph 和 judge
* 重构了发送端的代码，现在代码比之前容易维护了

alarm：

* 弃用了 open-falcon 的 alarm，完全重写
* 不支持报警合并
* 支持 EVENT 类型（只报警不记录状态）
* 支持多种报警后端（电话、短信、BearyChat、OneAlert、PagerDuty、邮件、微信企业号），并且易于扩展
* Mac 下有好用的 BitBar 插件

sender

集成进了 alarm 中。

links

在 Satori 中移除了。推荐直接使用低优先级的通道（如 BearyChat/其他IM，或者 BitBar），不做报警合并。

graph & query & dashboard

被 Grafana 和 InfluxDB 代替

judge：

* 被 riemann 代替
* riemann 较 judge 相比，可以节省 60% 以上的内存，CPU占用要低 50%。

task

在 Satori 中移除了。InfluxDB 自带 task 的功能。

aggregator

在 Satori 中移除了。riemann 中可以轻松的实现 aggregator 的功能。

nodata

在 Satori 中移除了。可以参见规则中的 feed-dog 和 watchdog，实现了相同的功能。

portal

在 Satori 中移除了。报警规则通过 git 仓库管理。

gateway:

合并进了 transfer（TODO）

hbs

* 在 Satori 中叫做 master
* 与 hbs 不兼容
* 不再将节点数据记录到数据库中，没有数据库的依赖。

uic

Satori 中去除，直接在规则仓库中编辑用户信息。

fe

完全重写，采用了纯前端的方案（frontend 文件夹）。

其他文档

* [添加规则](https://github.com/leancloud/satori/blob/master/doc/01-add-rule.md)
* [配置报警](https://github.com/leancloud/satori/blob/master/doc/02-config-alarm.md)
* [添加插件](https://github.com/leancloud/satori/blob/master/doc/03-add-plugin.md)
* [常见问题](https://github.com/leancloud/satori/blob/master/doc/04-faq.md)

# riemann/riemann:

A network event stream processing system, in Clojure.

https://github.com/riemann/riemann

Riemann monitors distributed systems.

Riemann aggregates events from your servers and applications with a powerful stream processing language.

Find out more at [http://riemann.io](http://riemann.io/)

===

Riemann - A network monitoring system

http://riemann.io/

Riemann aggregates events from your servers and applications with a powerful stream processing language. Send an email for every exception in your app. Track the latency distribution of your web app. See the top processes on any host, by memory and CPU. Combine statistics from every Riak node in your cluster and forward to Graphite. Track user activity from second to second.

Riemann provides *low-latency, transient shared state* for systems with many moving parts.

Download Riemann 0.2.13

[](https://github.com/riemann/riemann/releases/download/0.2.13/riemann_0.2.13_all.deb) [](https://github.com/riemann/riemann/releases/download/0.2.13/riemann-0.2.13-1.noarch.rpm) [](https://github.com/riemann/riemann/releases/download/0.2.13/riemann-0.2.13.tar.bz2)

Powerful stream primitives

(**where** (or (**service** **#**"^api")

(**service** **#**"^app"))

(**where** (**tagged** "exception")

(**rollup** 5 3600

(**email** "dev@foo.com"))

(**else**

(**changed-state**

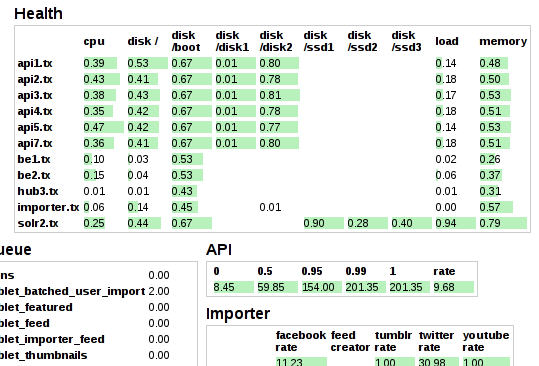
(**email** "ops@foo.com")))))

Riemann streams are just functions which accept an event. Events are just structs with some common fields like :host and :service You can use dozens of built-in streams for [filtering, altering, and combining events](http://riemann.io/howto.html#working-with-streams), or write your own.

Since Riemann's configuration is a *Clojure program*, its syntax is concise, regular, and extendable. Configuration-as-code minimizes boilerplate and gives you the flexibility to adapt to complex situations.

I wrote Riemann for operations staff trying to keep a large, dynamic infrastructure running with unreliable but fault-tolerant components. For engineers who need to understand the source of errors and performance bottlenecks in production. For everyone fed up with traditional approaches; who want something fast, expressive, and powerful.

All systems go



A small, extendable [Sinatra app](http://riemann.io/dashboard.html) shows your system at a glance. Instantly identify hotspots, down services, and unbalanced loads.

Phone, SMS and email alerts

(**rollup** 5 3600

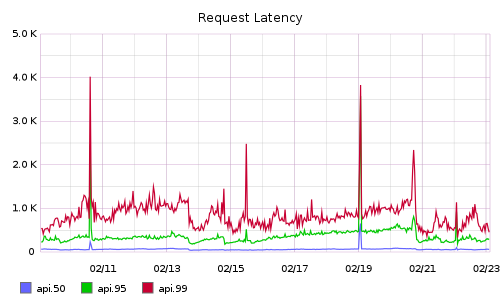
(**email** "dev@startup.com"))

An email describing a failure in a
      thumbnailer service.

Riemann can tell you as much or as little as you want. Throttle or roll up multiple events into a single messages. Get emails about exceptions in your code, provider downtime, or latency spikes. You can also integrate with [PagerDuty](https://github.com/riemann/riemann/blob/master/src/riemann/pagerduty.clj) for SMS or phone alerts.

Graph everything

(**graphite** {:host "graphs.pdx"})



Like [Statsd](https://github.com/etsy/statsd), Riemann can forward any event stream to [Graphite](http://graphite.wikidot.com/). Librato Metrics integration? Built in.

Simple clients

r **=** Riemann**::**Client.**new**

r **<<** {service: "www", state: "down", metric: 10000}

r['state = "down"']

*# => [#<Riemann::Event @service="www" ... >]*

Riemann speaks [Protocol Buffers](http://code.google.com/p/protobuf/) over TCP and UDP for a compact, portable, and fast wire protocol. See the [Ruby client](https://github.com/riemann/riemann-ruby-client/tree/master/lib/riemann) as a guide.

Query states

state = "error rate" and (not host =**~** "www.%")

Search the Riemann index with a small query language. Clients can monitor each other, generate reports, or render dashboards.

Riemann queries form the basis for the realtime websockets dashboard, showing updated events as soon as they arrive.

See problems faster

Traditional monitoring systems run polling loops every five minutes, or roll up metrics on a minutely basis. In a Riemann infrastructure, clients (including stand-alone pollers) \*push\* their events to Riemann, which makes them visible within milliseconds. Low latencies let you see outages faster--and know the instant you've fixed the problem.

Throughput depends on what your streams \*do\* with events, but a stock Riemann config on commodity x86 hardware can handle \*millions\* of events per second at sub-ms latencies, with 99ths around 5ms. Riemann is fully parallel and leverages Clojure and JVM concurrency primitives throughout.

Brought to you by…

Riemann was made possible by the [hard work](http://riemann.io/thanks.html) of many open-source contributors. Everyone who's offered advice, asked questions, and submitted code has my deepest appreciation.

I wrote Riemann to help solve problems at work: both [Showyou](http://showyou.com/) and[Boundary](http://boundary.com/) deserve thanks for letting me build this crazy thing and share it with the world. [Librato](https://metrics.librato.com/) sponsored the Librato Metrics integration. [Blue Mountain Capital](https://www.bluemountaincapital.com/) donated to help make Riemann faster.

Riemann uses YourKit for performance analysis. YourKit is kindly supporting open source projects with its full-featured Java Profiler. YourKit, LLC is the creator of innovative and intelligent tools for profiling Java and .NET applications. Take a look at YourKit's leading software products: [YourKit Java Profiler](http://www.yourkit.com/java/profiler/index.jsp) and [YourKit .NET Profiler](http://www.yourkit.com/.net/profiler/index.jsp).

# Riemann - Quickstart

http://riemann.io/quickstart.html

Installing Riemann

Riemann's components are all configured to work out-of-the-box on localhost, and this quickstart assumes you'll be running everything locally. If you're jumping right into running Riemann on multiple nodes, check out [Putting Riemann into production](http://riemann.io/howto.html#putting-riemann-into-production) for tips.

$ **wget** https://github.com/riemann/riemann/releases/download/0.2.13/riemann-0.2.13.tar.bz2

$ **tar** xvfj riemann-0.2.13.tar.bz2

$ **cd** riemann-0.2.13

Check the md5sum to verify the tarball:

wget <https://github.com/riemann/riemann/releases/download/0.2.13/riemann-0.2.13.tar.bz2.md5>

md5sum -c riemann-0.2.13.tar.bz2.md5

You can also install Riemann via the Debian or RPM packages, through[Puppet](https://forge.puppetlabs.com/garethr/riemann), [Vagrant](https://github.com/garethr/riemann-vagrant), or [Chef](https://github.com/hudl/riemann-cookbook).

Start the server

bin/riemann etc/riemann.config

Riemann is now listening for events. Install the Ruby client, utility package, and dashboard. You may need to install ruby-dev for some dependencies; your OS package manager should have a version available for your Ruby install.

gem install riemann-client riemann-tools riemann-dash

Start the dashboard. If riemann-dash isn't in your path, check your rubygems bin directory.

riemann-dash

Point your web browser to <http://localhost:4567/>. You'll see a big title in the top pane and a quick overview of the control scheme in the bottom pane. At the top right is the current host your browser is connected to. At the top left is the pager, which shows your workspaces--like tabs, in a browser.

Let's change the title into a Grid view. Hold CTRL (or OPTION/META depending on your OS) and click the big title "Riemann" in the top pane. The title will be shaded grey to indicate that view is selected.

Then, press "e" to edit, and change "Title" to "Grid". We need to choose a*query* to select specific states from the index. For starters, let's select*everything* by typing true in the query field. Hit "Apply" when you're ready.

This new view is likely a little small, so hit "+" a few times to make it bigger. Views are rescaled relative to their neighbors in a container.

Right now the index is empty, so you won't see any events. Let's send some:

riemann-health

The riemann-health daemon is a little Ruby program that submits events about the state of your CPU, memory, load, and disks to a Riemann server. If you switch back to the dashboard, you'll see your local host's state appear. The Grid view organizes events according to their host and service. Color indicates state, and the shaded bars show metrics. You can hover over an event, like CPU, to see its description.

Working with Clients

Now that Riemann is installed, let's try sending some of our own states through the Ruby client.

$ irb -r riemann/client

ruby-1.9.3 :001 > r = Riemann::Client.new

=> #<Riemann::Client ... >

We can send events with <<. For example, let's log an HTTP request.

ruby-1.9.3 :002 > r << {

host: "www1",

service: "http req",

metric: 2.53,

state: "critical",

description: "Request took 2.53 seconds.",

tags: ["http"]

}

On the dashboard, critical events (like the one we just submitted) show up in red. All of these fields are optional, by the way. The ruby client will assume your events come from the local host name unless you passhost: nil.

Now let's ask for all events that have a service beginning with "http".

ruby-1.9.3 :003 > r['service =~ "http%"']

[<Riemann::Event time: 1330041937,

state: "critical",

service: "http req",

host: "www1",

description: "Request took 2.53 seconds.",

tags: ["http"],

ttl: 300.0,

metric\_f: 2.5299999713897705>]

There's the event we submitted. You could send these events from Rack middleware to log every request to your app, or track exceptions. Riemann can be configured to calculate rates, averages, min/max, distributions, and more.

# Docker Swarm 和 Riemann 的实时集群监控

| DaoCloud http://blog.daocloud.io/docker-swarm-2/

深入浅出 Docker Swarm｜DaoCloud 现推出 Docker Swarm 系列技术文章，为大家深入浅出地解读 Docker Swarm 的概念、使用方法以及最真实的案例分析。全系列共五篇，本周为大家每日放送一篇精彩内容，敬请期待。

这是 Docker Swarm 系列的第二篇，关于 Docker Swarm 和 Riemann 的实时集群监控。

Riemann 一款由 Clojure 语言编写的高效监控、处理和响应分布式系统健康状况的工具软件，由 Kyle Kingsbury（代号 aphyr ）研发。

**今天我们将要讨论：**

1. 为什么 Docker 用户需要关心监控？
2. 为什么要用 Riemann？
3. 如何把 Riemann 组件 Docker 化？
4. 如何用 Docker Swarm 部署 Riemann 组件？
5. 如何使用 Riemann 控制台?

**为什么 Docker 用户需要关心监控？**

对于一些开发者来说，大谈特谈“监控”是最催眠的事情。的确，健康监控或其他形式的应用监测常常沦为不重要的东西：一山放过一山拦。现代开发者早已习惯了在应用交付用户前排除重重障碍。但恰当的监控可以帮助开发者实现梦寐以求的东西：**快速可靠的应用和心满意足的客户。**

如果你的基础架构和应用开发环节中，没有把监控作为「一等公民」集成的话，你早晚会陷入泥潭。气急败坏的客户会通过推特或其他方式变成你事实上的监控系统，因为他们会告诉你某某某功能不能用，或者网站崩掉了。这些事故比起其他任何问题，能更快速地毁掉你的生意和声誉，所以你无论如何都必须避免它们。

此外，一旦建立并用上了监控工具，它所能提供的观察力是极有价值的。从手动检查主机和服务，到安装配置好优良的监控软件，那感觉就好像是，以前上网连搜索引擎都没有，现在却到处能用谷歌。

对于处境独特的 Docker 用户而言，监控的价值更加意义重大，原因很简单：在以 Docker 为基础的世界里，需要监控的东西比以前多了一个数量级。你不仅有一大堆VM需要监测，还要面对各个 VM 上运行着的纷繁复杂的容器。此外还有一些移动组件如容器编排工具，key-value store，Docker 自身的 daemon 进程等等需要监控，而移动组件的数量是相当惊人的！在「 Docker 化」的世界里，自觉对应用进行监控和排故，其回报是不言自明的。

为什么要用 Riemann ？

使用 Riemann 有很多理由，以下几点是最显见的。

* **Riemann 基于推送的服务模式，让你可以检测到问题所在，近乎实时地修复故障**

传统的监测系统如 Nagios 通常基于查询模式：它们定时启动（例如每 5 分钟启动一次），运行一套健康监测程序，看看系统中的各部分是否处于应有状态（比如你的服务都已启动），然后通过邮件或其他方式向你报告结果。Riemann 则要求各个服务汇报自身的事件，一旦事件流停止流动，就能马上发现系统故障或网络断开。同样地，我们还能收集和处理各种形式的事件，事先监控，未雨绸缪，而不是在发生故障后疲于奔命。

* **Riemann 的流处理能够对输入的数据进行强大的转换**

举个例子：即使某个意外错误在 1 小时内有 1000 次日志，Riemann 也能把所有错误打包整理，用少数几封邮件就可以告知用户此错误发生了多少次。这就可以避免系统报警疲劳。另一个例子— Riemann 用的是非常精细的百分位度量，这可以准确反映系统运行的真实状况，而不会被平均值这类笼统数值一叶障目，遗漏掉重要的系统信息。

* **Riemann 可以很轻易地导入或导出数据，可以实现一些有趣的用例，比如为稍后的分析存储事件数据**

Riemann 用户可以用 TCP 或 UDP 上一个很简单的协议缓冲来发送事件。Riemann 可以把事件和指标从简单的索引发送到多种后端，包括 Graphite，InfluxDB，Librato 等等，还能内建功能支持通过 SMS、邮件和 Slack 等途径发送通知。

* **Riemann 的架构模型很简单，可以避免监控软件太难操作或频繁崩溃的尴尬局面**

Riemann 的设计就是从底层直达操作和产出层面。尽管这其中会有一些妥协（比如无法安全分配一个单独的 Riemann 实例），但在实际运用中，Riemann 的原则基本是一鸟在手，胜过十鸟在林，此刻的不完美信息好过遥远的完美信息。如果 Riemann 服务器崩溃了，只需要简单的重启就能解决，不需要对混乱状态进行纷繁复杂的调解。

同样的，基于推送的模式也有助于解决“谁来监控监控软件”的问题（换句话说，我们如何检测监控软件本身有没有问题）。我们只需在下游服务器上建几个 Riemann 服务器和前端事件。如果上游服务器出故障了，下游服务器就会感知并提醒用户。尤其是在云端运行 Docker 的时候，这一功能的价值非常明显。

* **它非常快**

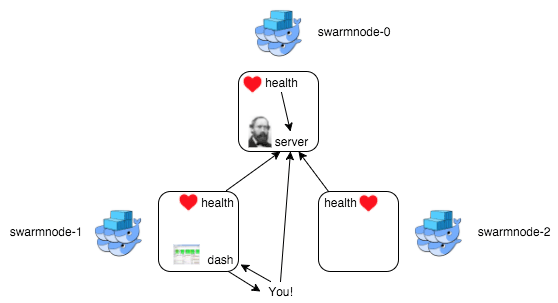
摘自 Riemann 登录页：“吞吐量取决于你的流对事件做了什么，不过商用 x86 硬件上一小块 Riemann 每秒就能处理数百万次事件，延迟时间却只有亚毫秒，5 毫秒可以完成 99 次。Riemann 完全并行并利用 Clojure 及 JVM 的并发基元（ primitives ）”。

如何 Docker 化 Riemann 组件？

今天我们讨论 Docker 化三种组件，这三种组件合并起来，就能有效发挥 Riemann 的性能。

1. Riemann 服务器进程，由 Clojure 语言编写，是主流处理引擎
2. Riemann-health 程序，由 Ruby 语言编写，向中央 Riemann 服务器报告健康/使用指标
3. Riemann-dash 程序，基于 Ruby 语言编写，是一个小型 Sinatra 应用，为 Riemann 提供网页控制台界面

我们将用 Docker Swarm 运行它们，它们借助 libnetwork 的 overlay 驱动来彼此联动。这里有一张组件的样本架构示意图，展示了它们是如何并入我们的 3 节点集群的。每个节点都有一个 riemann-health 实例向服务器报告指标。另外两个容器会被随机调配，至于落到哪台主机上都不要紧。

[](http://blog.daocloud.io/wp-content/uploads/2016/04/riemannhealth.gif)

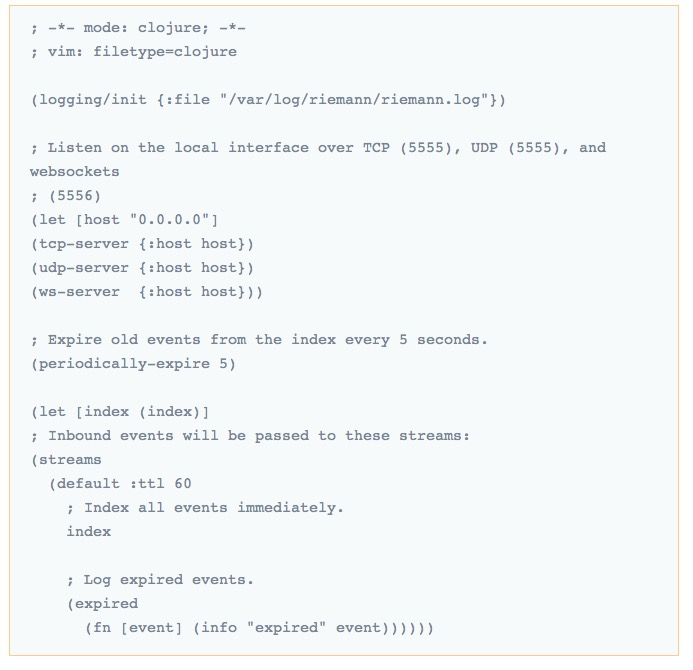
在本文章中，假设你已经装好了一个 Swarm 集群，拥有至少 3 个运行的节点。

Docker 镜像

Riemann 服务器的 Dockerfile 是这样子的：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/1.pic_.jpg)

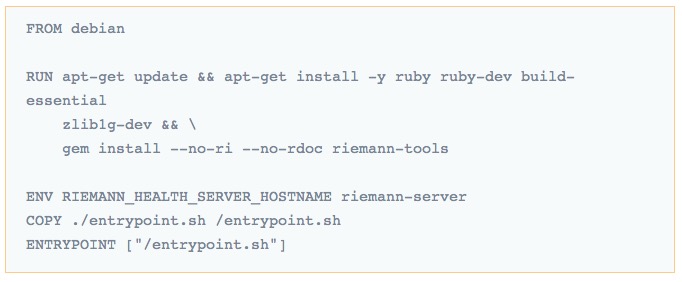
如你所见，它只是简单安装了 Java 运行环境，从 Kyle 的网站里抓取了.deb 调试包，然后安装了中央 Riemann 服务器。它还将以下配置文件插入了镜像。

[](http://blog.daocloud.io/wp-content/uploads/2016/04/2.pic_.jpg)这是一个用 Clojure 写的很简单的配置文件，仅仅把传来的事件插入了 Riemann 索引并在5秒之后过期。当一个事件过期时，Riemann 会在日志上标记这个事件已经过期。

Riemann 的一个重大优势，就是可以定义事件有关的行为。借助 Riemann 的基元（ primitives ），我们可以把缺失的信息当成一条新信息：如果一段时间内某个服务或主机没有传送事件，我们就知道它要么是崩溃了，要么是诸如网络不畅之类的原因阻挡了信息的传输。因为我们可自定义此类情况发生时所采取的动作，所以我们可以设置在主机崩溃时通过 Slack 频道发出报警（或者进行备份）。

从理论上讲，这可以拓展成一个具备自我修复能力的自控系统。例如，当我们检测到某台主机或服务的指标消失时，就可以启动新的服务先顶上，重启主机，以及采取其他措施。

现在我们有了建立 Riemann 服务器镜像（收集指标）的办法，但还需要一个实际发送信息的途经。Riemann-health 程序将向中央处理器发送 CPU ，内存和负载信息，所以我们来把它 Docker 化：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/3.pic_.jpg)

我们用这个 entrypoint.sh 脚本来确保我们能配置健康程序应当连接的 Riemann 服务器的位置（默认中，我们把这个容器叫做 Riemann-server，使用 libnetwork 默认 DNS ）

entrypoint.sh 看起来是这个样子的：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/4.pic_.jpg)当我们最终运行这个容器时，/etc/hostname 会从外部系统挂载到容器上，确保 Riemann-health 程序向中央服务器发送正确的主机名。

最后，我们来瞧瞧 Riemann 控制台的镜像：

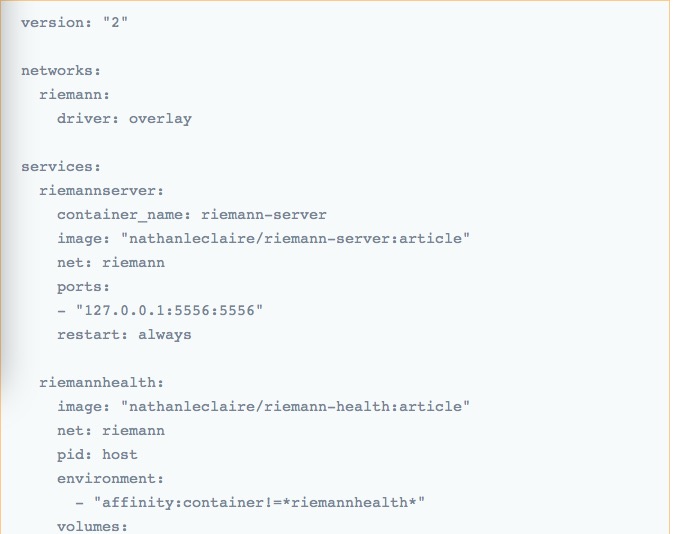
[](http://blog.daocloud.io/wp-content/uploads/2016/04/5.pic_.jpg)config.rb:

[6.pic](http://blog.daocloud.io/wp-content/uploads/2016/04/6.pic_.jpg)

如何用 Docker Swarm 部署 Riemann 组件？

我们用 Docker Compos 文件表示这些容器的运行信息，调度参数（确保 Riemann-health 实例布在每个节点上）以及相应的 overlay 网络（确保它们都能在多主机上访问到对方）。由于我们指定了要创建的网络，就需要用 version 2（笔者写作时最新的方案）配置格式编写 compose 文件。

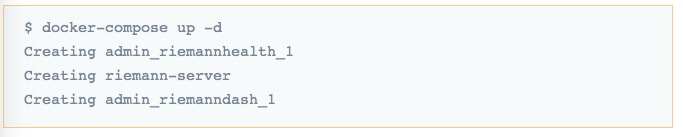
我们的 Docker Compose 文件是这样的：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/11.pic_.jpg)[](http://blog.daocloud.io/wp-content/uploads/2016/04/12.pic_.jpg)

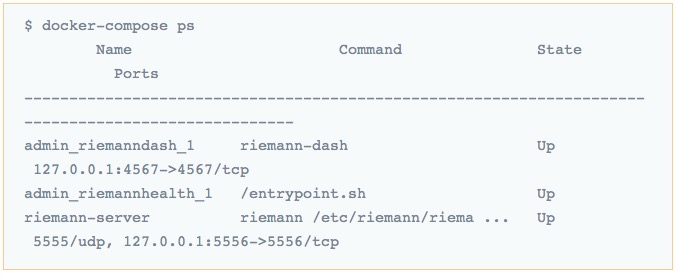
有一些值得一提的方面：

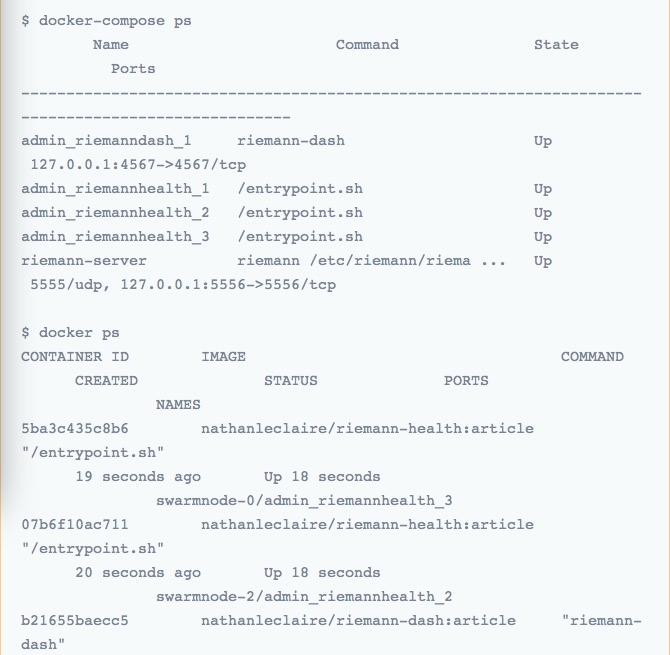
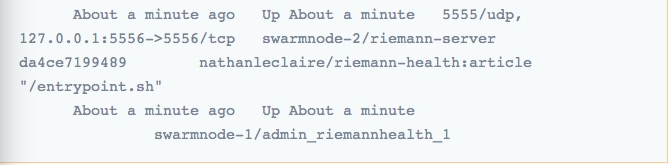
* Riemannhealth 是在主机的 pid 命名空间里运行的，以便准确收集每个进程的使用参数。
* Riemannserver 和 Riemannhealth 都在 Riemann 的 overlay 网络上，以确保 daemon 可向中央服务器发送事件。
* Swarm 调度限制设置用 environment，确保每台主机上只有一个 Riemannhealth 服务实例在运行。
* Riemannserver 的 container\_name 是手动设置的，以防依赖 Compose 通过 libnetwork，为 DNS 发现生成自动命名惯例。

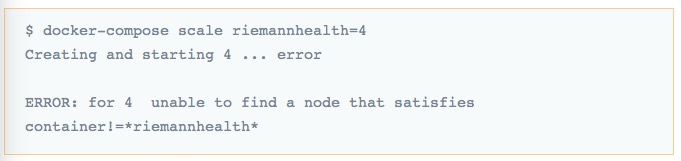
只要文件安放到位，接下来的事情就好办了：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/10.pic_.jpg)

我们可以用 docker-compose ps 查看创建好的容器：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/9.pic_.jpg)然后，把 Riemannhealth 服务外扩展到我们可用的节点：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/14.pic_.jpg)[](http://blog.daocloud.io/wp-content/uploads/2016/04/15.pic_.jpg)请注意，因为我们有调度限制，所以无法扩得很大：

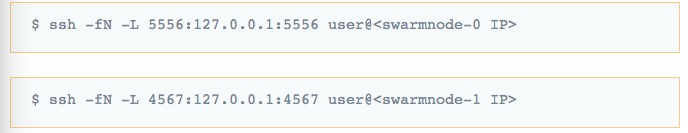
[](http://blog.daocloud.io/wp-content/uploads/2016/04/8.pic_.jpg)

好，现在我们见识过 Riemann 服务器和控制台的运行，以及健康 daemon 向每台主机的中央服务器报告参数的过程。现在我们来看看怎么操作 Riemann 控制台。

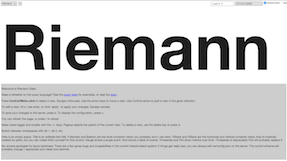
如何使用 Riemann 控制台？

现在我们可以收集参数，但还需要把收集到的东西可视化，这时就需要 Riemann 控制台了。

基于以上的 Compose 文件，建议是使用 SSH 转发控制台端口（4567）和中央 Riemann 服务器端口（5556）到你的 localhost 上。这能让你安全方便地访问控制台。比如，假设服务器处在 swarmnode-0，仪表板处在 swarmnode-1.

[](http://blog.daocloud.io/wp-content/uploads/2016/04/7.pic_.jpg)

你会看到空白的 Riemann 控制台。

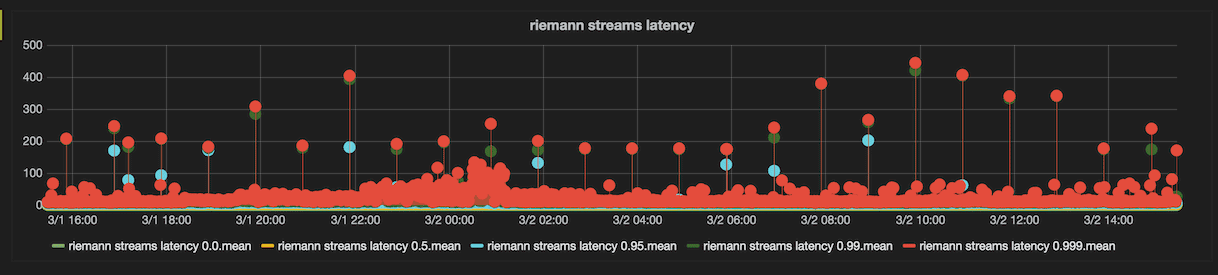
[](http://blog.daocloud.io/wp-content/uploads/2016/04/9.png)

我们可以用如 Chart、Grid 等格式来展示 Riemann 查询。

[](http://blog.daocloud.io/wp-content/uploads/2016/04/8.png)

比如，这些绿色的区域表示正从各种主机和服务汇聚到中央 Riemann 服务器的参数。如果这些参数开始触及“危险区域”，这些长方形就会按程度变成黄色或红色。

另外，你还可以设置 Riemann 向时间序列数据库发送数据（比如跟 Grafana 这样的可视化前端绑定），这样你就可以查看受监控内容的历史，并做出像这样精美的图表：

[](http://blog.daocloud.io/wp-content/uploads/2016/04/01.gif)

配合使用 ELK 这样的中心化日志，你能看到更多更细微的架构内的运作。拥有这样的工具，你可以完全掌控系统的健康，不必担心发生问题。例如，在百分位级别的监控下，Riemann 潜流里大约每小时就出现一次的刺突，你都能看得清清楚楚。掌握了这些信息，我们就能把问题扼杀在摇篮里，防止它积少成多，最终变得一溃千里。

结论

Docker Swarm 和 Riemann都是令人非常着迷的技术，建议大家深入了解和使用它们。市面上已经有很多可用的 Riemann 工具，特别是用于监控 Docker Daemon 和编排系统的工具，都非常棒。你也很有可能成为编写这些工具的开发者之一，所以继续深入，好好写代码吧！