

## 数据驱动的智能运维平台

伴随着各类高新技术的出现，“人工智能”一词越来越多地出现在人们的日常生活中，而运维朋友常听到与自身工作息息相关的便是智能运维（AIOps）了。

但在当前，国内大部分的智能运维并没有完全落地，整个行业处在一个初期的探索阶段。因此，很多运维人或多或少都有这样的疑问：一个传统企业的智能运维之路该如何走？AIOps 的架构设计与组成究竟从哪里落地？今天，小编就为大家带来了饶琛琳对于智能运维平台建设的演讲分享实录。

本篇分享，主要从运维需求的源头出发，逐步推导出 AIOps 的架构设计与组成，在推导过程中，饶琛琳详细介绍了时序预测、异常检测、模式概要的分析原理与实现方式的具体场景，以及对应的开源项目选择。实录详情如下，还等什么，快来收干货吧！

### 讲在前面

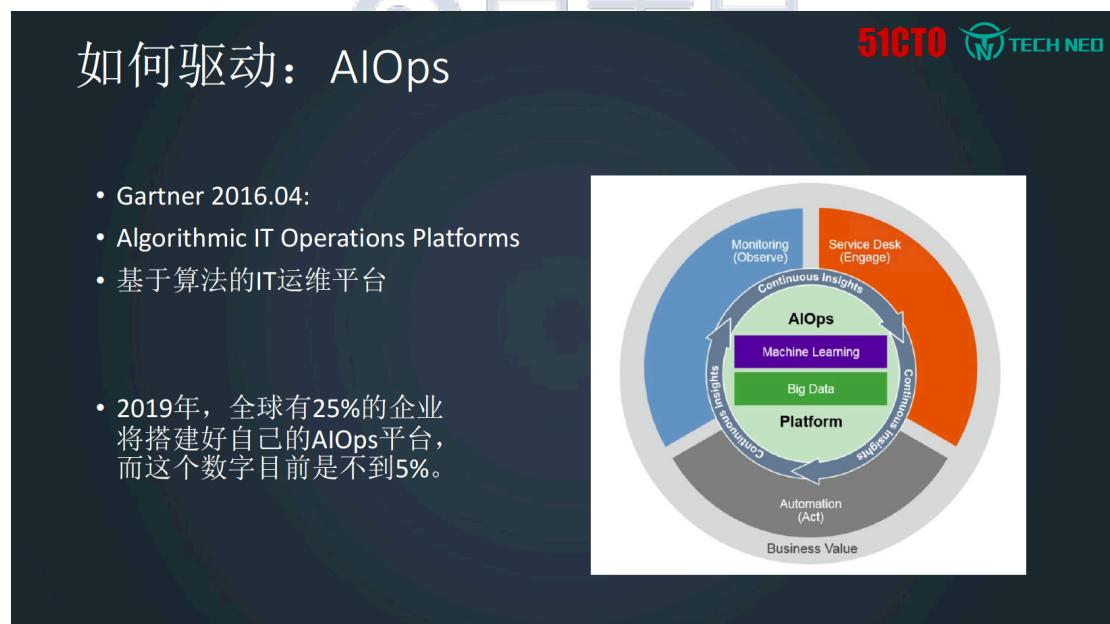
今天分享的是数据驱动的智能运维平台，可以看到，标题有两个点，第一个是平台，第二个是数据驱动。主要是分享平台本身的主件架构，重点放在一些与数据分析相关的细节，包括对异常预测、时序预测、模块聚类等场景的剖析。

我在参与编写《企业级AIOps实施建议》白皮书期间，与腾讯、华为手机的一些朋友在讨论“智能运维”时候，得到了一个比较有趣说法：我们可以把AIOps进

行分级，像“异常检测”这样的细节点，就认为它是一个原子场景，不用再细分，或者是引用周志华教授的说法，称其为“学件”，这种学件，类似于程序中的API或公共库，可以放到四海而皆准；再往上层就是类似于“根因分析”的串联应用，通过多种算法、多个原子场景组建出来的串联组合场景；再往上便是更高级的场景，最终达到终极AIOps。今天分享的皆是原子场景的使用方式。

## 谈谈 AIOps

怎么构建一个AIOps平台？我们先要确定目的，然后再谈如何达到目的。在定义AIOps时画了一张图（如下图），除了中间有机器学习、BigData、Platform外，外层的内容就是监管控，这也就是做 AIOps 的目的。只不过是在做监管控时，要使用一些新的方式，以减轻运维的工作量。



与传统运维相比，智能运维可以更灵活、更易用，并且快速探索数据。比如有1000台服务器，如果没有一个统一的平台，要发现问题会非常麻烦。探索和实验平台是什么意思呢？这其实是总结了运维人员的一个工作状态：猜测、试错，如果试错不对，再进行下一次试错，即一个探索发现的过程。如果

这个过程执行不够快，就意味着解决故障的速度会慢下来。因此，我认为，这个快慢问题对于运维来说非常重要的一个点。

## What's inside AIOps?

51CTO TECH NEO

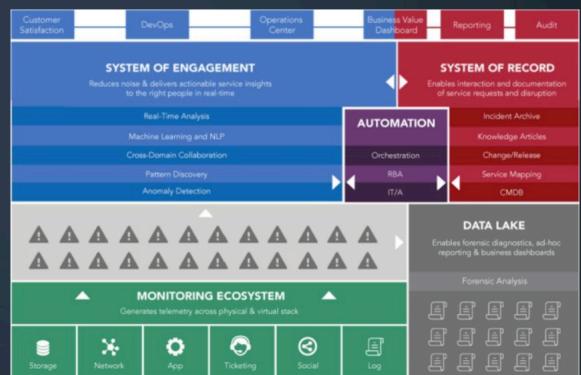
- 三大作用：
- 更灵活、更易用的访问和分析数据；
- 能分析过去散落在各组件中未利用上的业务数据和上下文；
- 快速的探索和实验平台，提供独特的洞察力

从实际情况来看，AIOps平台里应该有哪些东西？我觉得下面的描述很有趣，数据湖，即存储采集数据，还有自动化系统、记录系统、交互系统、监控生态圈。

## What's inside AIOps?

51CTO TECH NEO

- 从『系统组成』看AIOps架构：
- 数据湖、
- 自动化系统、
- 记录系统、
- 交互系统
- 监控生态圈



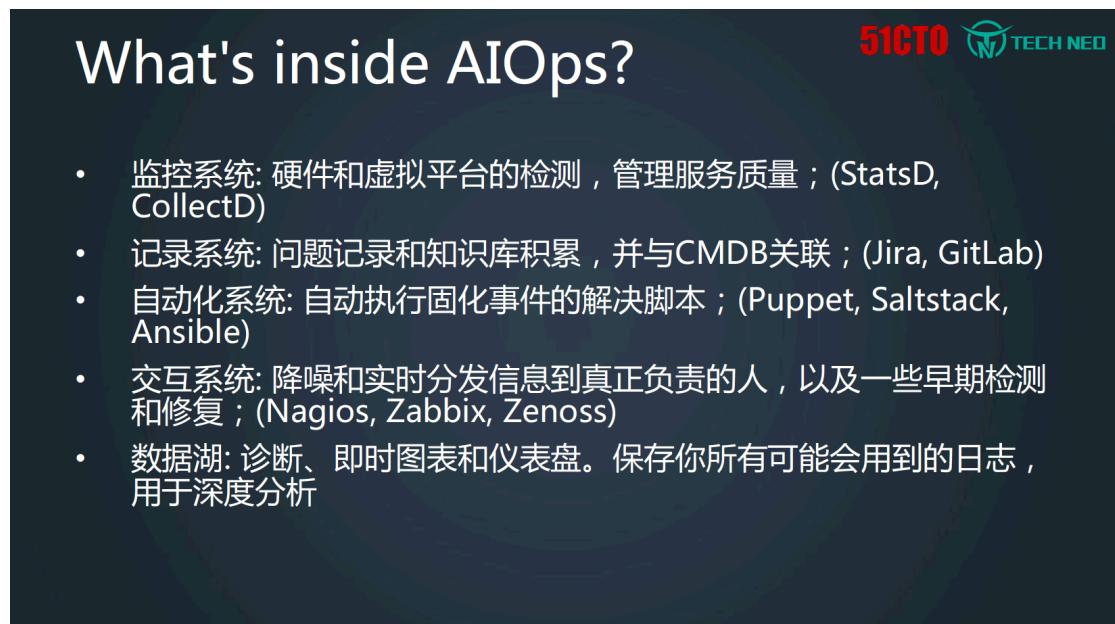
The diagram illustrates the AIOps system architecture, showing the integration of various components:

- Customer Satisfaction** feeds into **DevOps**, which feeds into **Operations Center**.
- Operations Center** feeds into **Business Value Dashboard**, which feeds into **Reporting**, and finally **Audit**.
- SYSTEM OF ENGAGEMENT** (blue box) contains Real-Time Analysis, Machine Learning and NLP, Cross-Domain Collaboration, Pattern Discovery, and Anomaly Detection.
- SYSTEM OF RECORD** (red box) contains Incident Archive, Knowledge Articles, Change/Release, Service Mapping, and CMDB.
- AUTOMATION** (purple box) contains Orchestration, RBA, and IT/A.
- MONITORING ECOSYSTEM** (green box) generates telemetry across physical & virtual stack, connecting to Storage, Network, App, Ticketing, Social, and Log components.
- DATA LAKE** (grey box) enables forensic diagnostics, ad-hoc reporting & business dashboards, and includes Forensic Analysis.

将这几个系统拆分一下，我们可以发现，监控系统和交互系统在运维的分类中比较混淆。一般来说，监控系统负责的只是把数据抓下来，然后去判断是不是

有问题，但是实际上监控系统还要负责一个重要的流程，也就是这个问题和其他问题有没有联系？应该把这个问题发给谁？发送时只能告诉有这么一个问题，还是描述更多信息？这段流程要比数据采集部分更重要。要做好支撑运维目的平台，就需要将其单独拆分考虑。

这张幻灯看起来好像和AI没有太大的关系，但只有具备这些系统，就可以承认这是一个Ops平台了，但是在这个平台中，AI是什么？



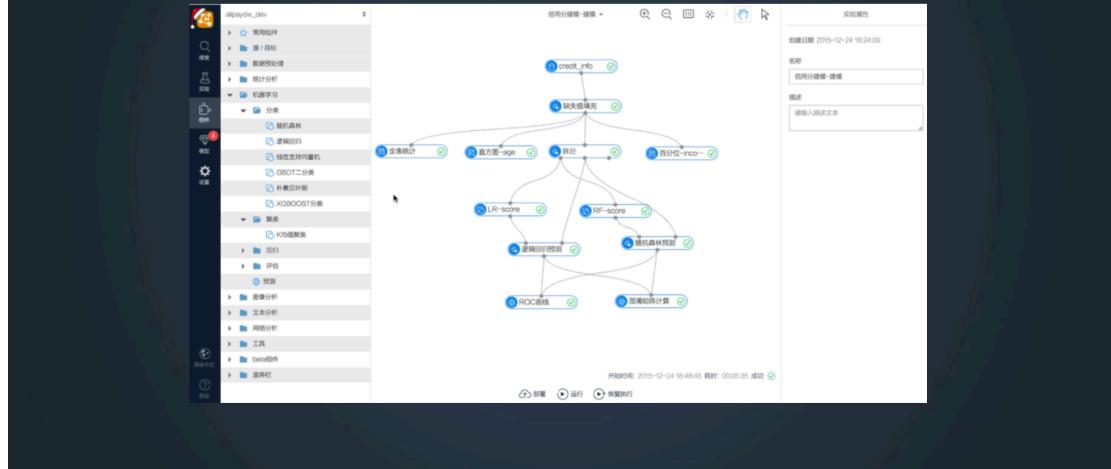
## What's inside AI Ops?

51CTO TECH NEO

- 监控系统: 硬件和虚拟平台的检测，管理服务质量；(StatsD, CollectD)
- 记录系统: 问题记录和知识库积累，并与CMDB关联；(Jira, GitLab)
- 自动化系统: 自动执行固化事件的解决脚本；(Puppet, Saltstack, Ansible)
- 交互系统: 降噪和实时分发信息到真正负责的人，以及一些早期检测和修复；(Nagios, Zabbix, Zenoss)
- 数据湖: 诊断、即时图表和仪表盘。保存你所有可能会用到的日志，用于深度分析

这是阿里云AI平台的一张截图，类似于这种的机器学习web平台，市面上应该有三四十种，但这种平台对运维来说并没有实际的意义。

# 我们不需要一个机器学习平台



我们运维人真正需要的是机器学习在运维工作中的运用。AppDynamics的2016

年度总结中提出一些对于APM厂商来说可以做出的AI场景，可以对这些内容进行拆解，得出运维人的真正需求。

## 我们需要机器学习的运用

- AppDynamics的2016年度总结：
  - 商业智能 ( Business Intelligence )
  - 异常检测 ( Anomaly Detection )
  - 归因分析 ( Correlation & Root Cause Analysis )
  - 智能警报 ( Intelligent Alerting )
  - 未来预测 ( Forecasting & Prediction )
  - 能力分配 ( Capacity Planning )
  - 数据概要 ( Data Summarization )
  - 自动化 ( Automation )
  - 主动监控 ( Proactive Monitoring )

我这里提供一种很好的拆解方式，这是《Google SRE book》书中的一张图，

对于运维人员来说，最重要的还是要去解决底层需求，包括监控、事件响应、

更新分析、CI/CD、容量规划、部署，将这张图与上AI应用场景进行对照，便会

得到从技术到需求应用之间的关系。



从对应的关系中可以看出，很多链条是相通的，而最终的目的都是要做好一个监控，即最底层的需求。此外，还有一条链是“根因分析-智能报警-自动化”。也就是上面的链条发现故障，最后一条链发出报警，并明确后续流程。

## 智能运维的运用 & 路径

- 异常检测
- 归因分析
- 智能警报
- 未来预测
- 能力分配
- 数据概要
- 主动监控

- 异常检测 -> 主动监控
- 数据概要 -> 异常检测 -> 主动监控
- 未来预测 -> 容量规划 / 异常检测
- 根因分析 -> 智能警报 -> 自动化

## 典型应用场景

### 时序预测

下面主要聊一下两个大链条里几个最常见、比较好入手的场景。第一个是时序预测，预测这个话题非常大。在与客户交流时，也会被问到一些离谱的预测需求，但真正可落地的需求，还是那些数据量足够大、细，且全面，同时预测的是比较细致情况的需求。

## 未来预测——太大的话题

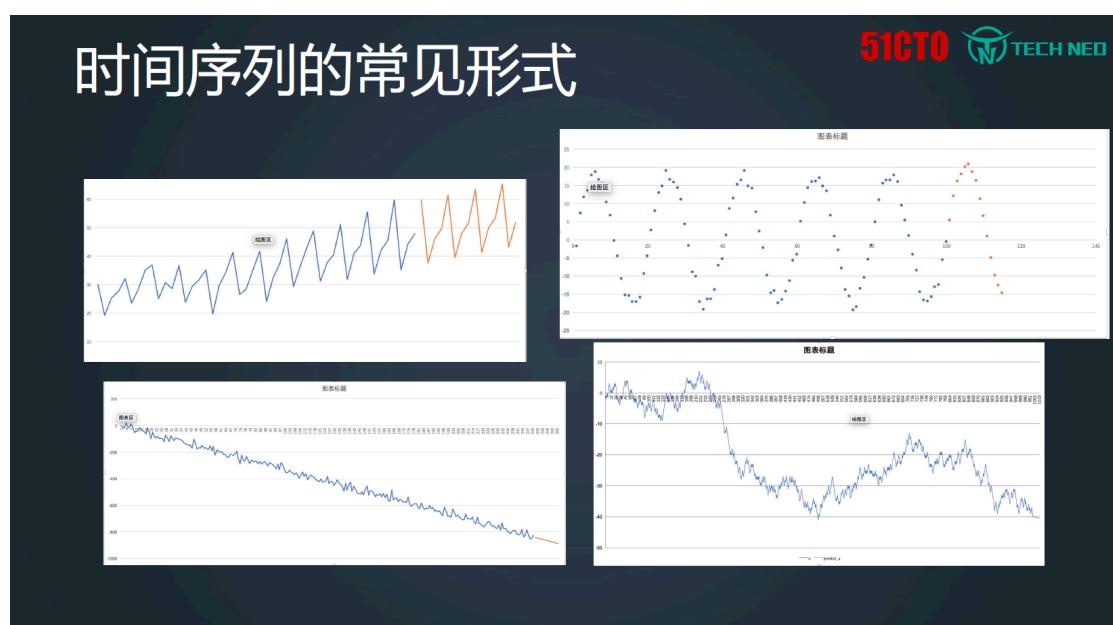
- 在和各种企业的交流中，我碰过如下这类未来预测的伪需求：
  - 能帮我预测央行什么时候调整政策么？(那我还干什么苦逼运维)☆☆☆☆☆
  - 能帮我预测明年我要采购多少设备么？(一年后很多事都黄了好吧)☆☆☆☆☆
  - 能帮我预测什么时候故障怎么修复么？(故障太多种，可以聊聊细节)☆☆☆
- 什么才算是靠谱的未来预测需求：
  - 我有过去三年的XXX指标时序数据，能预测未来x小时/天的情况么？
  - 我有这批服务器的型号参数配置和状态监控历史，能预测它的磁盘/CPU故障么？

即使是靠谱的未来预测需求，也依然是太大的话题。例如下图，有了时序数据，以红框为点，中间的蓝线是数据实际情况，剩下三条线是用了三种不同的预测算法得到的预测结果，你会发现依然千差万别。

因此，即便有数据，在要求不高的情况下，能不能做依然是一个需要划分的问题。



回到运维领域，下面几张图是大家比较常见的序列，对于四种常见的序列情况我们可以想到它应该怎么走，这时就可以想办法让机器去想。



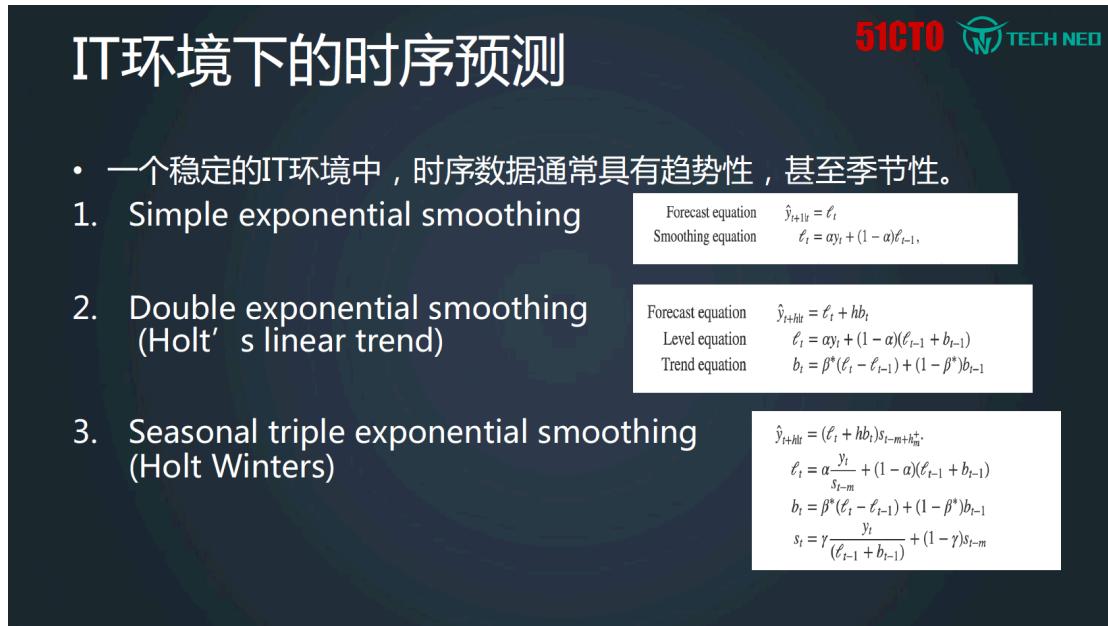
对于以上几张图来说，可以用统计学上的办法去做时序预测，就是指数平滑，

从一阶、二阶、三阶持续运算， $\alpha$ 、 $\beta$ 、 $\gamma$ 就会越来越多。

如果有100万条这样的线，依次去配 $\alpha$ 、 $\beta$ 、 $\gamma$ ，那工作量将会非常浩大。就这么

几个参数、十几条线，可能就要花费两三个月的时间来做，如果说所有的监控

指标全这么做，那肯定是不现实的。



IT环境下的时序预测

51CTO TECH NED

- 一个稳定的IT环境中，时序数据通常具有趋势性，甚至季节性。

- Simple exponential smoothing
- Double exponential smoothing (Holt's linear trend)
- Seasonal triple exponential smoothing (Holt Winters)

Forecast equation  $\hat{y}_{t+1|t} = \ell_t$   
Smoothing equation  $\ell_t = \alpha y_t + (1 - \alpha) \ell_{t-1}$ ,

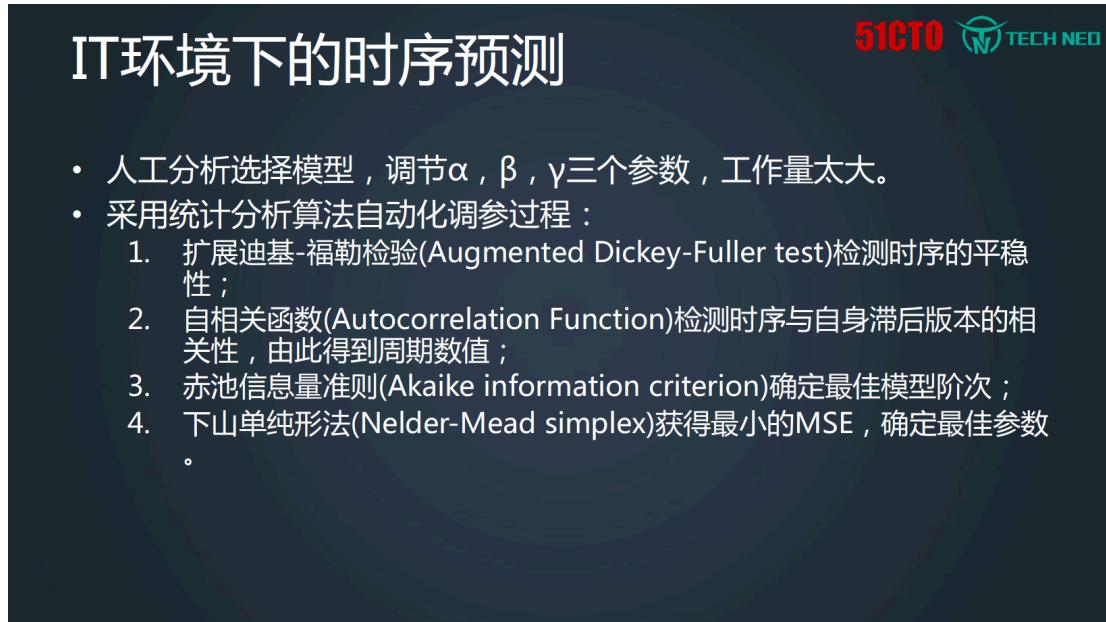
Forecast equation  $\hat{y}_{t+h|t} = \ell_t + hb_t$   
Level equation  $\ell_t = \alpha y_t + (1 - \alpha)(\ell_{t-1} + b_{t-1})$   
Trend equation  $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)b_{t-1}$

$\hat{y}_{t+h|t} = (\ell_t + hb_t)s_{t-m+h}^+$ ,  
 $\ell_t = \alpha \frac{y_t}{s_{t-m}} + (1 - \alpha)(\ell_{t-1} + b_{t-1})$   
 $b_t = \beta^*(\ell_t - \ell_{t-1}) + (1 - \beta^*)b_{t-1}$   
 $s_t = \gamma \frac{y_t}{(\ell_{t-1} + b_{t-1})} + (1 - \gamma)s_{t-m}$

在此基础上，就可以考虑用一些减轻人工作量的办法，我们可以用各种不同统

计学里的函数确定情况，最后获取一个相对最好的MSE，确定最佳参数，这样

工作量就会减轻一些。



对于时序预测的开源选择有很多，除了刚才讲到的RRDtool，Holt-Winters外，

还有facebook，hawkular的开源项目。

前面讲的对自动化调参的过程，很多具体的细节来自Redhat项目，虽然主项目已经没有更新，但是这个子项目还是推荐大家看一下。

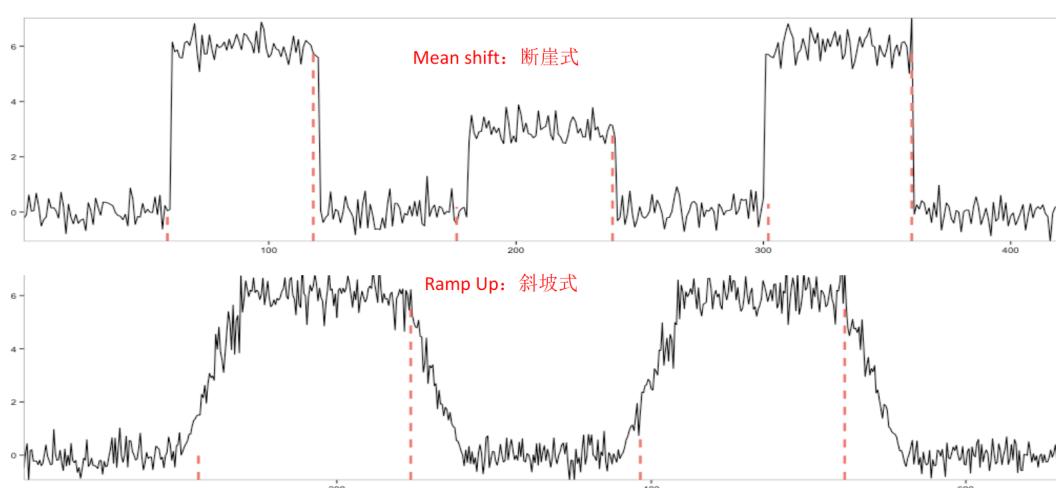
## 时序预测的开源选择

- Facebook在今年2月刚开源的Prophet库。  
(R/Python)
- RedHat在2015年开源的hawkular项目。 (Java)
- Elasticsearch在去年的pipeline aggregation。  
(Java , 基于holt-winters)
- RRDtool在1997年实现的HWPREDICT。 (C , 基于 holt-winters)
- . . . .

## 异常检测

第二个场景是异常检测。其实预测本身就是异常检测的一种方式，但异常检测并不只是这种方式。例如下面这两种，虽然是比较离谱的情况，但并不代表在长时间维度下不会出现，这种情况上任何平滑的东西，对这条线的异常检测都

## 什么是异常——偏离

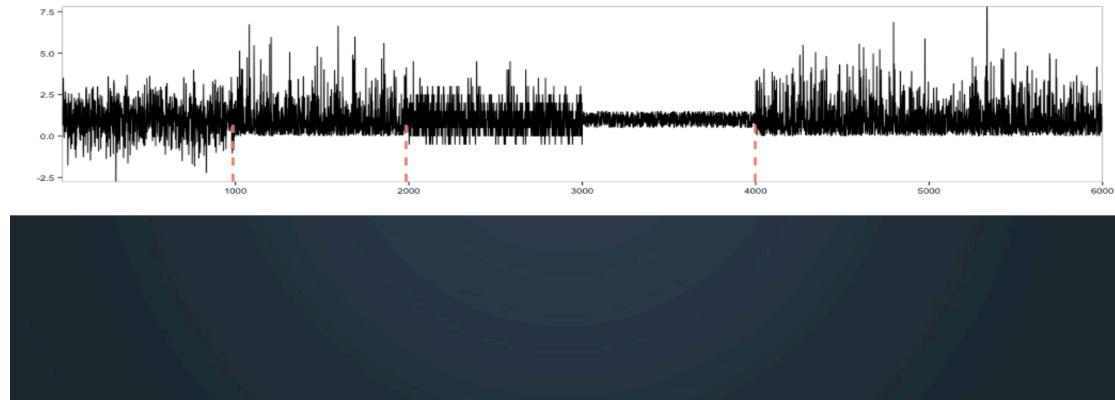


再如下面这种线，在不同的障碍阶段差别很大，但用平均值的话，整个这

一段中平均值都在一条线上，根本无法判断这条线的任何区别。

## 什么是异常——数据分布变化

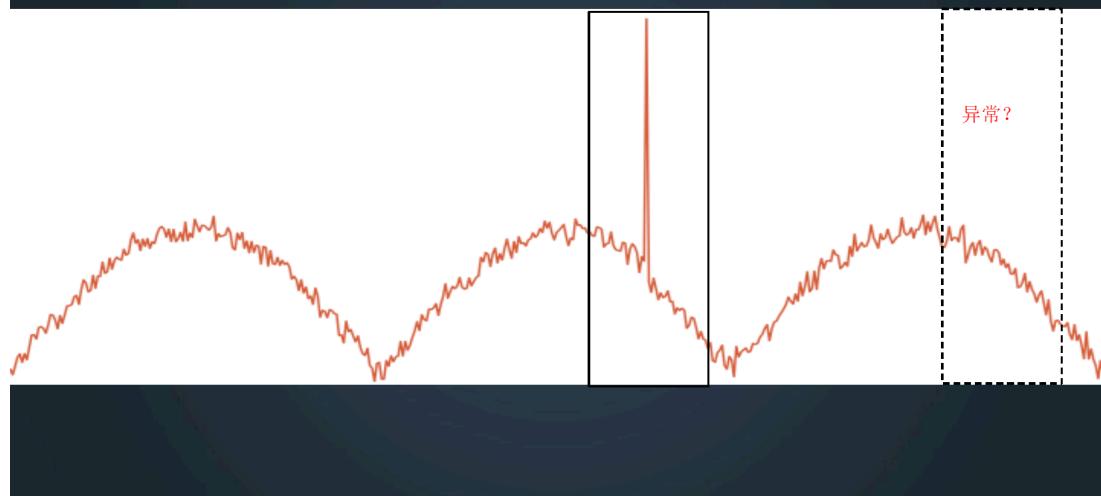
51CTO TECH NEO



此外，异常检测还要考虑一个最基本的同环比，也要考虑同比的鲁棒性。

## 什么是异常——鲁棒性

51CTO TECH NEO



这里可以介绍一下datadog的异常检测，提供4种检测方法，Basic采用的是四分位方法，Agile用的是SARIMA算法，Robust用的是趋势分解，Adaptive在我看来，采用的是sigma标准差。

# datadog的异常检测

- datadog提供4种异常检测方法可以选择：
  - Basic: 对没有周期性、季节性的数据，直接采用四分位的方法计算置信区间。
  - Agile: 针对季节性的数据，采用SARIMA算法，并着重强调近期数据对预测的影响。
  - Robust: 针对季节性数据，采用趋势分解算法，更强调长期数据对预测的影响。对频繁发布的业务不合适。
  - Adaptive: 针对季节性数据，但是对细节变动不敏感。容易漏异常。

下面是在不同场景下，这四种不同算法对这一条线是否异常的判断，我们可以看到，如果不需要对本身业务的理解，单纯就是一个算法，一切都正常，如第一个想过对比，但在实际工作中却不太可能。  
所以当我们真的要去做一异常检测时，必须对业务要有一定的了解，明白 metric这条线背后代表的含义，才能对各种算法进行选择，这个地方没有万能钥匙。

## 不同场景的算法效果对比

1. 正常运行中的 metric 数据与不同算法的置信区间。

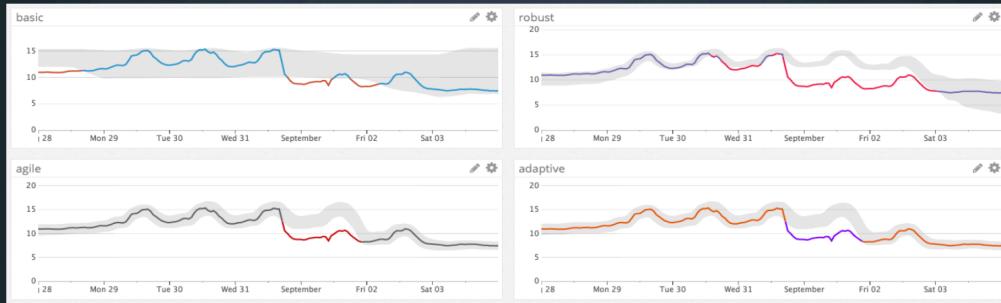


## 不同场景的算法效果对比

2. 陡降的 metric 数据。

趋势分解算法对变更的反应速度非常缓慢。

此外，注意四分位算法和趋势分解算法的区间扩散。

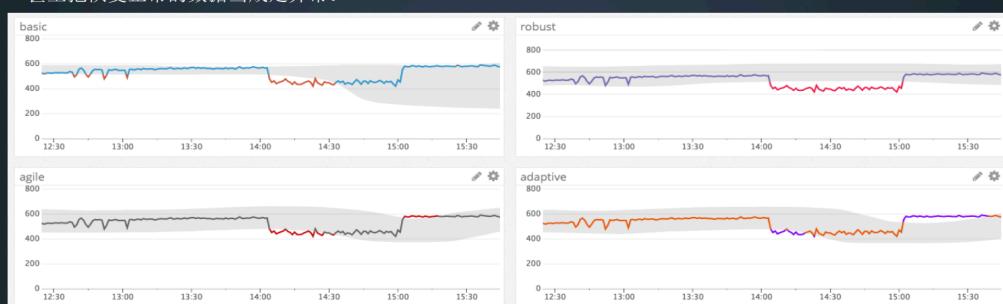


## 不同场景的算法效果对比

3. 一次长达小时级别的故障下的 metric 数据。

除了趋势分解，其他算法都开始把故障时间的数据当成正常数据了。

甚至把恢复正常的数据当成是异常。



## 不同场景的算法效果对比

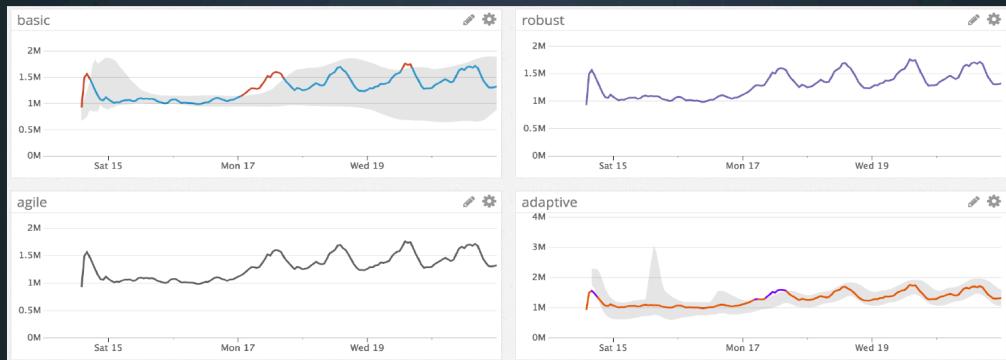
4. metric 数据分布变化的影响。

同样的陡降，但是从 50 降到 20，和从 1050 降到 1020，SARIMA 和趋势分解算法的理解是不一样的。



## 不同场景的算法效果对比

5. 全新的 metric 数据。  
四分位算法不需要历史；SARIMA和趋势分解需要足够的季节数据。



对于异常检测的开源库选择，有些是原子的，有些是组合的。**Etsy**的**skyline**是比较高级的场景，里面带有数据存储、异常检测分析、告警等；Twitter、Netflix、Numenata是纯粹的机器学习算法库，没有任何附加内容；Yahoo的egads库可以算是异常检测的原子场景，比Twitter和Netflix层级稍高。

## 异常检测的开源选择

- Yahoo!在去年开源的egads库。(Java)
- Twitter在去年开源的anomalydetection库。(R)
- Netflix在2015年开源的Surus库。(Pig，基于PCA)
- Etsy在2013年开源的skyline库。(python)
- Numenta在2013年开源的NuPIC库。(python，基于HTM)
- 德国海森堡大学在2005年开源的ELKI库。(Java，基于KNN和KMean)

## 模式发现

第三个要讲的是数据概要-文本聚类。我们知道，前面讲的两类都是监控metric

情况，但是一些故障单纯看metric是无法找出故障的。在排障过程中可以看几条线，包括时间相关性或者时序聚类，也可以做根因分析，但这些还不够。我这边可以提供的是另外一条思路，日志易是一款日志分析产品，企业有各种各样的系统，产生各种各样的日志，如果通过ETL的方式把日志结收集起来，可能要写上万个表达式，是不可能完成的任务。

## 数据概要——文本聚类

51CTO TECH NEO

- 中大型企业中，通常都有几十个系统，上百个模块，几千个不同的日志文件。要运维人员了解全部日志的格式，配置解析规则，定义告警条件，是不可能完成的任务！——但是只采集不处理的日志≈不采集。
- 故障日志并非每天产生，尤其严重的故障日志，历史的关键字日志告警往往难以覆盖所有未知异常。
- 我们需要用机器来解决这个复杂度问题。

我们可以看到这段有四行日志的输出代码，可以看出日志格式和种类是有限的。假如这四行代码打了1000万条，其实也就是这四行代码打的而已。如果从人的理解上看，这四行代码就说了两件事：1.有一个user登录了，2.定义了一个常量。我们要干的是什么？就是把1000万行代码反推到四个不同的日志样式。

## 日志文本 vs 自然语言文本

- 相比自然语言，日志格式的种类，其实是有限且固定的：

```
1. public static Logger logger = LogManager.getLogger("Foo");  
2.  
3. logger.debug("Logging in user %s with birthday %s", user.getName(), user.getBirthdayCalendar());  
4. logger.debug("Logging in user %1$s with birthday %2$tm %2$te,%2$tY", user.getName(), user.getBirthdayCalendar());  
5. logger.debug("Integer.MAX_VALUE = %,d", Integer.MAX_VALUE);  
6. logger.debug("Long.MAX_VALUE = %,d", Long.MAX_VALUE);
```

- 如上图，哪怕1kw日志，其实也就是4种；如果再模糊一点，就是2种。

另外一个细节，在处理自然语言时，逗号还是分号没有任何意义，我们关注的是文本，但日志里面的每一个符号都很关键，是一个独特的聚类聚合方式。如果我们不想上机器学习技术，只想先跨出第一步，就可以利用这个特性，除去文本，留下这堆标点符号。

## 日志文本 vs 自然语言文本

- 相比自然语言，日志格式中停用词的位置是固定且敏感的：

```
1. public static Logger logger = LogManager.getLogger("Foo");  
2.  
3. logger.debug("Logging in user %s with birthday %s", user.getName(), user.getBirthdayCalendar());  
4. logger.debug("Logging in user %1$s with birthday %2$tm %2$te,%2$tY", user.getName(), user.getBirthdayCalendar());  
5. logger.debug("Integer.MAX_VALUE = %,d", Integer.MAX_VALUE);  
6. logger.debug("Long.MAX_VALUE = %,d", Long.MAX_VALUE);
```

- punct是一种特别的日志格式分类方式。
- 把文本聚类，降级成为punct聚合。

替换之后，就留下这些东西，也足够反映出一些信息。例如下面这个实例，这个思科的ASA日志情况，进行处理后，得到了一堆一模一样的标点符号，我们就可以推测应该是同样的东西，这个是最简单的方式，因为比较粗略，所以推

测的也不是特别有效。

## 日志的punct处理

51CTO TECH NEO

- logstash配置片段:

```
filter {
    fingerprint {
        source => "message"
        target => "punct"
        method => "PUNCTUATION"
    }
}
```

- sed实质命令:

```
# cat samplelog.cisco.asa | \
sed 's/^(\.\{128\}).*\$/\1/' | \
sed 's/[0-9a-zA-Z]*//g' | \
sed 's/[:space:]/_/g'
```

```
→ samples cat samplelog.cisco.asa | sed 's/^(\.\{128\}).*\$/\1/' | sed 's/[0-9a-zA-Z]*//g' | sed 's/[:space:]/_/g' | sort | uniq -c | sort -nr
34 --:::..._<--:....::://.
31 --:::..._<--:....::://..
19 --:::..._<--:....::://...
18 --:::..._<--:....::/...
15 --:::..._<--:....::/...
12 --:::..._<--:....:::_<>_<
7 --:::..._<--:....::://...
6 --:::..._<--:....:/...
6 --:::..._<--:....:/...
5 --:::..._<--:....:/...
```

可以再往前一步，加上一点聚类的东西，先走TFIDF，提取一些文本的特征值，

再走一个DBSCAN，拿每个聚类的样本情况来看。当看到某个样本不太对，就单独把这个样本拿出来，调整参数，将聚类里的日志重新聚类，再观察一下情况。

## 聚类流程-指令表达

51CTO TECH NEO

- \*  
| fit TFIDF punct max\_df=1 min\_df=0.2 ngram\_range=1-5  
max\_features=100  
| fit DBSCAN punct\_tfidf\_\*  
| dedup 2 cluster

**sklearn.feature\_extraction.text.TfidfVectorizer**

```
class sklearn.feature_extraction.text.TfidfVectorizer (input=u'content', encoding=u'utf-8',
decode_error=u'strict', strip_accents=None, lowercase=True, preprocessor=None, analyzer=u'word',
stop_words=None, token_pattern=u'(?u)b\\w+\\b', ngram_range=(1, 1), max_df=1.0, min_df=1, max_features=None,
vocabulary=None, binary=False, dtype=<type 'numpy.int64'>, norm=u'l2', use_idf=True, smooth_idf=True,
sublinear_tf=False)
```

[source]

## 聚类流程-拆分子类

```
• *
| fit TFIDF punct max_df=1 min_df=0.2 ngram_range=1-5
max_features=100
| fit DBSCAN punct_tfidf_*
| where cluster==0
| fit DBSCAN eps=0.2 min_samples=1 punct_tfidf_*
```

### sklearn.cluster.DBSCAN

```
class sklearn.cluster.DBSCAN(eps=0.5, min_samples=5, metric='euclidean', algorithm='auto', leaf_size=30,
p=None, n_jobs=1) [source]
```

聚类的思路是相通的，先提取，做聚类，聚类出来有问题，再切分一个小类。

但是实际上上线使用的话，还是有很多问题需要考虑的。用DBScan聚类的运行

时间比较长，是一个偏离线运行状态，而且占用的资源也多。

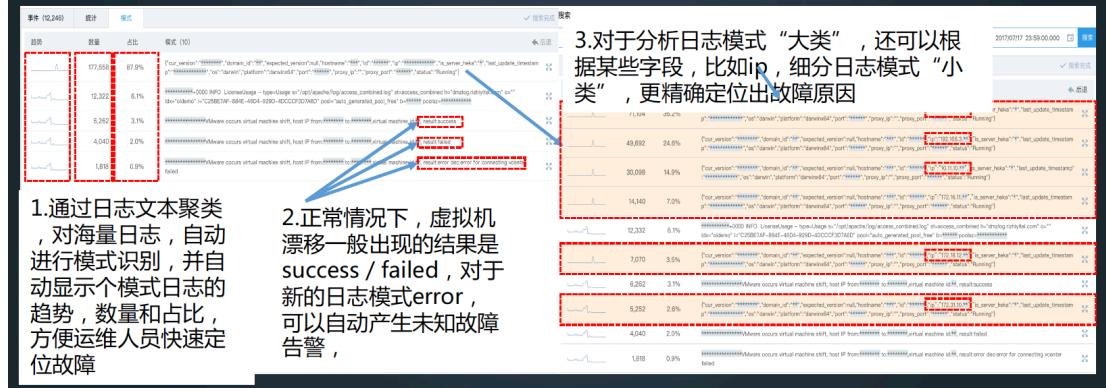
除了这类算法上的问题，还有一个思路上的问题，单纯只是完全的聚类，并不足以达到知道它的原始代码是什么样的目的，因为就没有办法去很合适地判断逻辑代码。

## DBScan聚类效果的问题

- 使用DBScan算法进行日志聚类，有几个难题需要考虑：
  1. 运算时间较长，偏离线运行；
  2. 占用内存较多，大规模数据耗费较大；
  3. 聚类效果是否合适没有简易的判断方式。

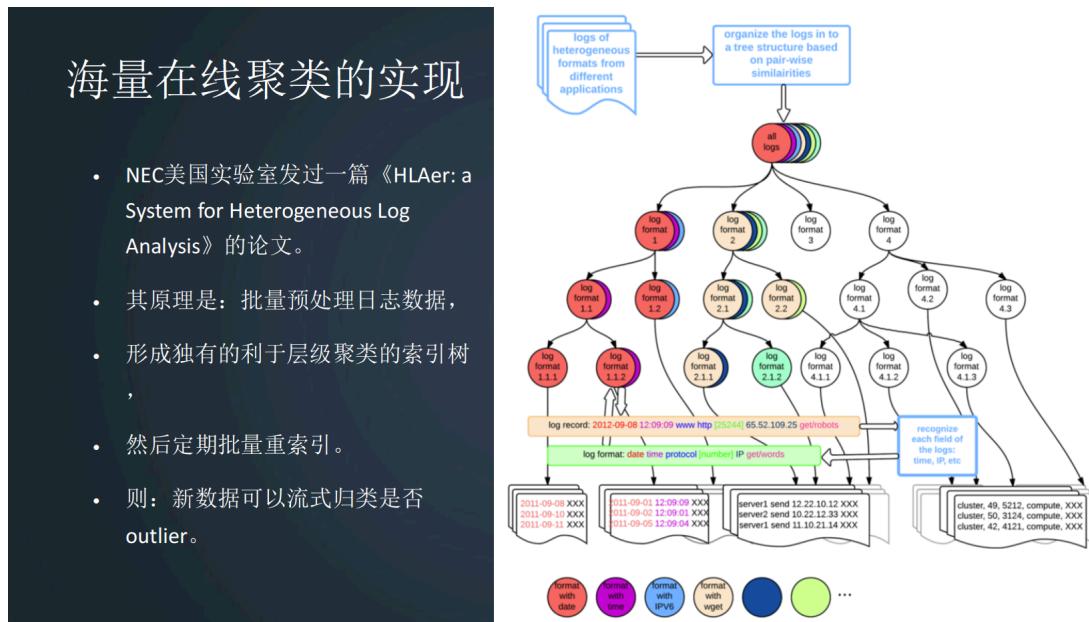
# 聚类效果的判断：pattern

- 通过明显的占位符，来抽象同一聚类内的日志模式。



这里我们参考一下日本电器美国实验室曾经发表的一篇论文，他们的算法

叫HLAer，原理是不直接上一大堆文本的聚类方式，而是反过来去推导。



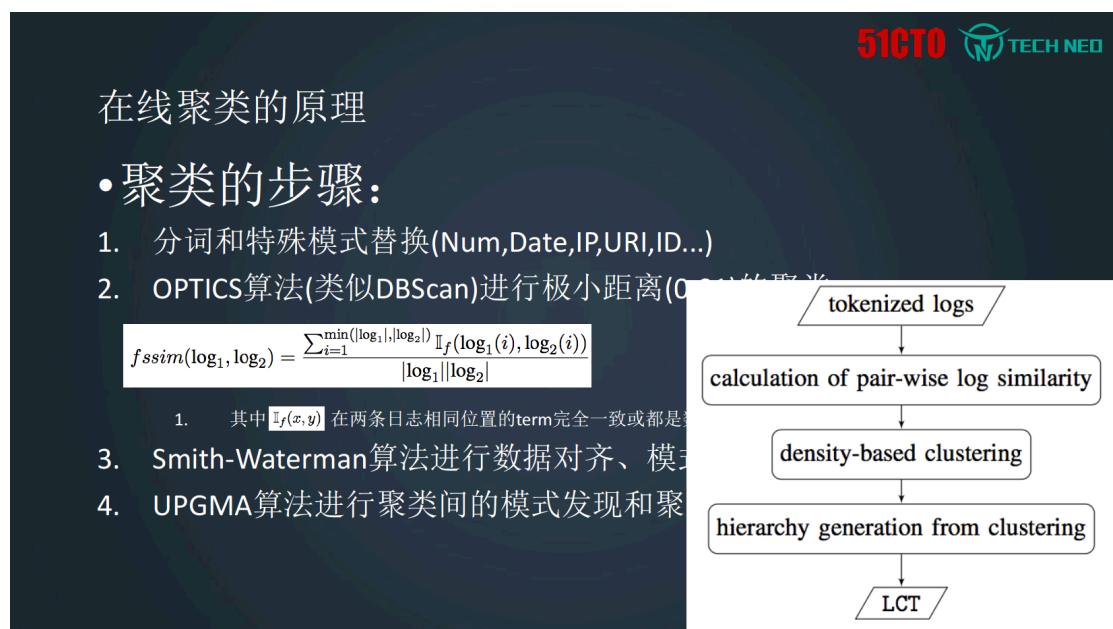
我们做的是运维日志，大多数情况下，运维日志有很多东西不需要耗费CPU处

理。第一个，像Num、Date、IP、ID等都是运维IT日志里一定会出现的，但在关注模式时不会关注这些。因此，可以在开始就把这些信息替换，节省工作量。

第二个是对齐，对齐也是耗资源的，如何减少对齐的时候强行匹配资源呢？可

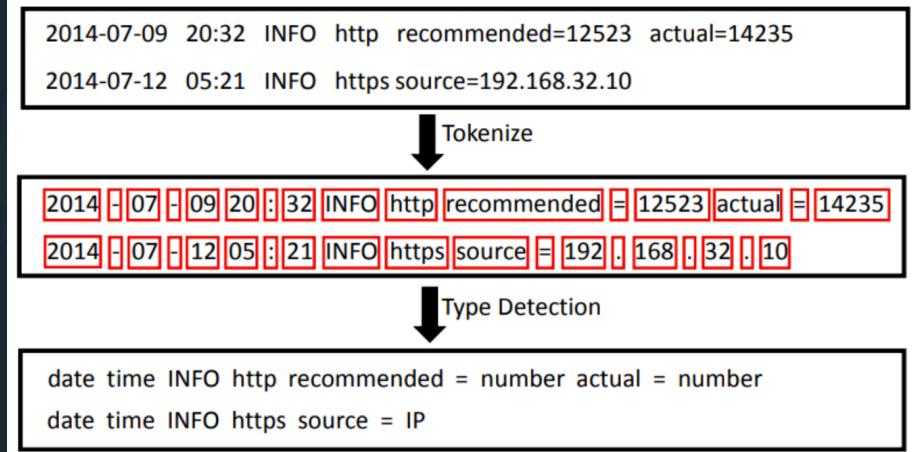
以开始先走一个距离极其小的聚类，这样每一类中的原始文本差异非常小。此时意味着第二步得到的最底层聚类去做对齐时，在这个类里的对齐耗损就会非常小，可以直接做模式发现。

到第四步的时候，虽然还是聚类，但是消耗的资源已经非常少，因为给出的数据量已经很小，可以快速完成整个速度的迭代。

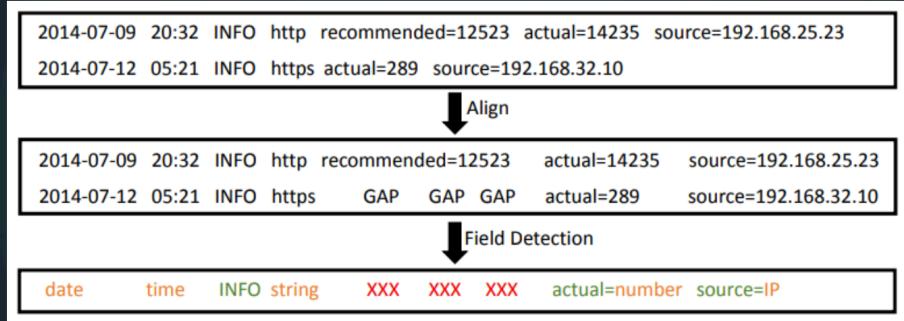


这是一个事例，首先将两条日志去做分层，再去做一个发现，然后去做一个对齐和一个模式发现。通过这种方式，可以把所有日志一层一层往上推，最终把整个结果全部推导出来。

## 分层聚类示意(1)



## 分层聚类示意(2)



比如一开始给出的是15种，觉得不合适，往下走一层，看13、14是怎样的，还不合适，再往下走一层，看9、10、11、12是什么，总有一层是合适的，就可以把前面的8条日志得到一个树状结构。

## 分层聚类示意(3)

```

1. 2015-07-09 10:22:12,235 INFO action=set root="/"
2. 2015-07-09 12:32:46,806 INFO action=insert user=tom id=201923 record=abf343rf
3. 2015-07-09 14:24:16,247 WARNING action=remove home="/users/david"
4. 2015-07-09 20:09:11,909 INFO action=insert user=david id=455095 record=e frdf4w2
5. 2015-07-09 21:56:01,728 INFO action=set home="/users"
6. 2015-07-09 22:11:56,434 WARNING action=delete user=tom id=201923 record=a sepg9e
7. 2015-07-09 22:32:46,657 INFO action=insert user=david id=455095 record=3jns67
8. 2015-07-09 22:34:12,724 WARNING action=remove home="/users/tom"

```

```

9. date time,number INFO action=insert user=david id=455095 record=XXX
10. date time,number XXX action=XXX user=tom id=201923 record=XXX
11. date time,number INFO action=set XXX=XXX
12. date time,number WARNING action=remove home=XXX

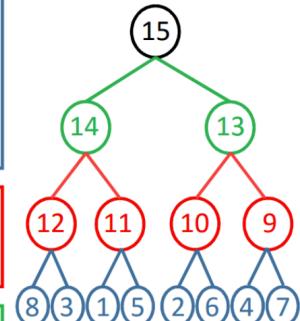
```

```

13. date time,number XXX action=XXX user=XXX id=XXX record=XXX
14. date time,number XXX action=XXX XXX=XXX record=XXX

```

```
15. date time,number XXX action=XXX XXX=XXX XXX*=XXX* XXX*=XXX*
```



当然，为了方便使用，可以提前中止结构树生成，不一定要推到顶上那个点。

一般会在提前、合适的情况下，终止这个数的生成，从机器办法来说，可以记

录下每一层剩下多少个这样的模式，找到拐点，这个拐点能够证明再往下已经不方便合并即可，但是这种方式计算量比较大。

因此，目前会选择一个简单但是对肉眼比较合适的方式，即每一行都有分词，

如果一行里面分了20个，其中，要被替换成新的东西超过了5%，觉得不太合适

看，这个时候就可以停下来了。

## 分层聚类示意总结

- 经过上述的日志分层聚类过程。最终会合并得到一个root pattern。
- 在信息量足够的情况下，我们可以提前中止pattern tree的生成。

```

2012-07-09 20:32:46,864 INFO org.apache.hadoop.hdfs.util.GSet: recommended=4194304, actual=4194304
Jan 8 05:49:14 www httpd[7855]: 108.199.240.249 - - "GET /images/header/nav-pub-on.gif HTTP/1.1" 200 569
2012-07-09 20:32:46,904 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem: fsOwner=hadoop_user
2012-07-09 20:32:46,905 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem: supergroup=supergroup
Jan 8 05:49:27 www httpd[7855]: 108.199.240.249 - - "GET /careers/internship.php HTTP/1.1" 200 11007
Jan 8 05:49:27 www httpd[7855]: 108.199.240.249 - - "GET /careers/images-careers/intern-title.gif HTTP/1.1" 200 1211
2012-07-09 20:32:46,905 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem: isPermissionEnabled=false
2012-07-09 20:32:46,909 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem dfs.block.invalidate.limit=100
Jan 8 05:49:27 www httpd[7855]: 108.199.240.249 - - "GET /images/home/current-bullet.gif HTTP/1.1" 200 131
Jan 8 05:49:27 www httpd[7896]: [error] [client 108.199.240.249] File does not exist: /var/www/html/favicon.ico

```

## 关于日志易

北京优特捷信息技术有限公司是国内从事 IT 运维日志、业务日志实时采集、搜索、分析、可视化系统研发的大数据公司，提供下载版软件和 SaaS 服务，以及面向金融、运营商、电力、互联网等行业的日志分析解决方案。

日志易目前已服务一百多家大型企业，包括五大银行里的三家、十二家股份制银行里的六家、两家大型保险公司，以及国家电网、南方电网、中石油、中石化、中国移动、中国电信、中国中车、上汽通用等知名企业，今后日志易也将立足于金融、能源、运营商等行业，努力为服务企业打造一个高效的日志生态系统。

2018 年，日志易联合业内多家 AIOps 厂商发布了首部《企业级 AIOps 实施建议》白皮书，旨在为各企业实现 AIOps 提供路径指引。



扫码预约 AIOps 交流



扫码了解更多 AIOps 案例