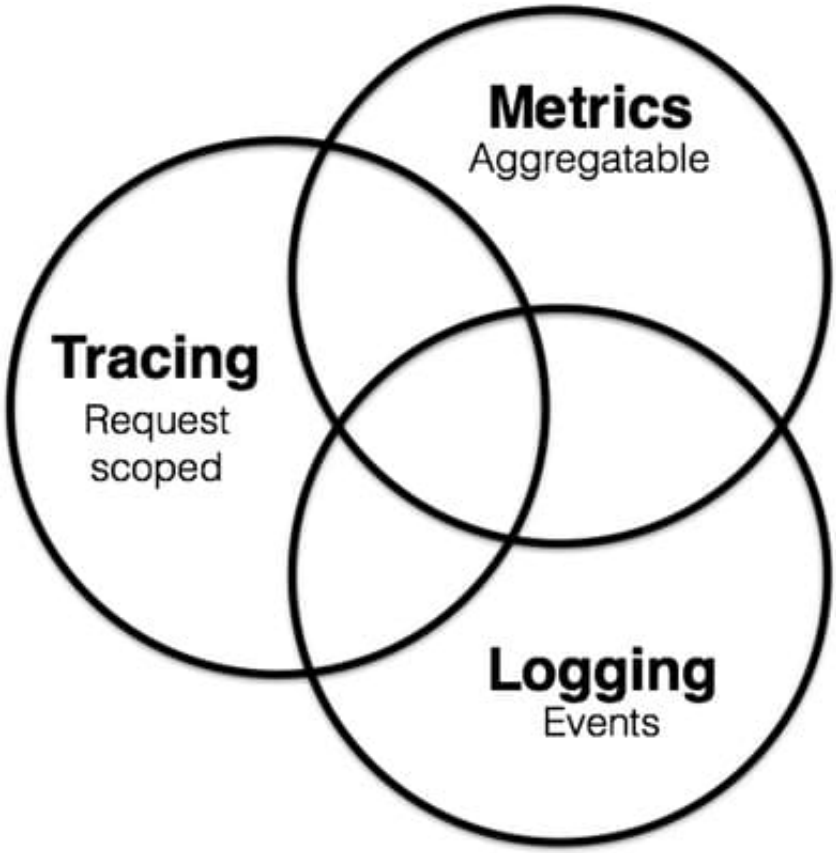
12.3.1 监控系统的诉求

在如今微服务、云原生等技术盛行的时代，当谈到说要从 0 开始构建一个监控系统，大家无非就首先想到三个词：Metrics、Tracing、Logging。国外一篇比较火的文章 Metrics, tracing, and logging 内有个图很好的总结了一个监控系统的诉求，分别是 Metrics、Logging、Tracing。



Metrics 的特点：它自己提供了五种基本的度量类型 Gauge、Counter、Histogram、Timer、Meter。

Tracing 的特点：提供了一个请求从接收到处理完毕整个生命周期的跟踪路径，通常请求都是在分布式的系统中处理，所以也叫做分布式链路追踪。

Logging 的特点：提供项目应用运行的详细信息，例如方法的入参、运行的异常记录等。

这三者在监控系统中缺一不可：基于 Metrics 的告警事件发生异常，通过 Tracing 定位问题可疑模块，根据模块详细的日志定位到错误根源，最后再返回来调整 Metrics 的告警规则，以便下次更早的预警，提前预防出现此类问题。

12.3.2 监控系统包含的内容

针对提到的三个点，笔者找到国内外的开源监控系统做了对比，发现真正拥有全部功能的比较少，有的系统比较专注于 Logging、有的系统比较专注于 Tracing，而大部分其他的监控系统无非是只是监控系统的一部分，比如是作为一款数据库存储监控数据、作为一个可视化图表的系统去展示各种各样的监控数据信息。

拿 Logging 来说，开源用的最多最火的技术栈是 ELK，Tracing 这块有 Skywalking、Pinpoint 等技术，它们的对比如 APM巅峰对决：Skywalking P.K. Pinpoint 一文介绍。而存储监控数据的时序数据库那就比较多了，常见的比如 InfluxDB、Promethes、OpenTSDB 等，它们之间的对比介绍如下：

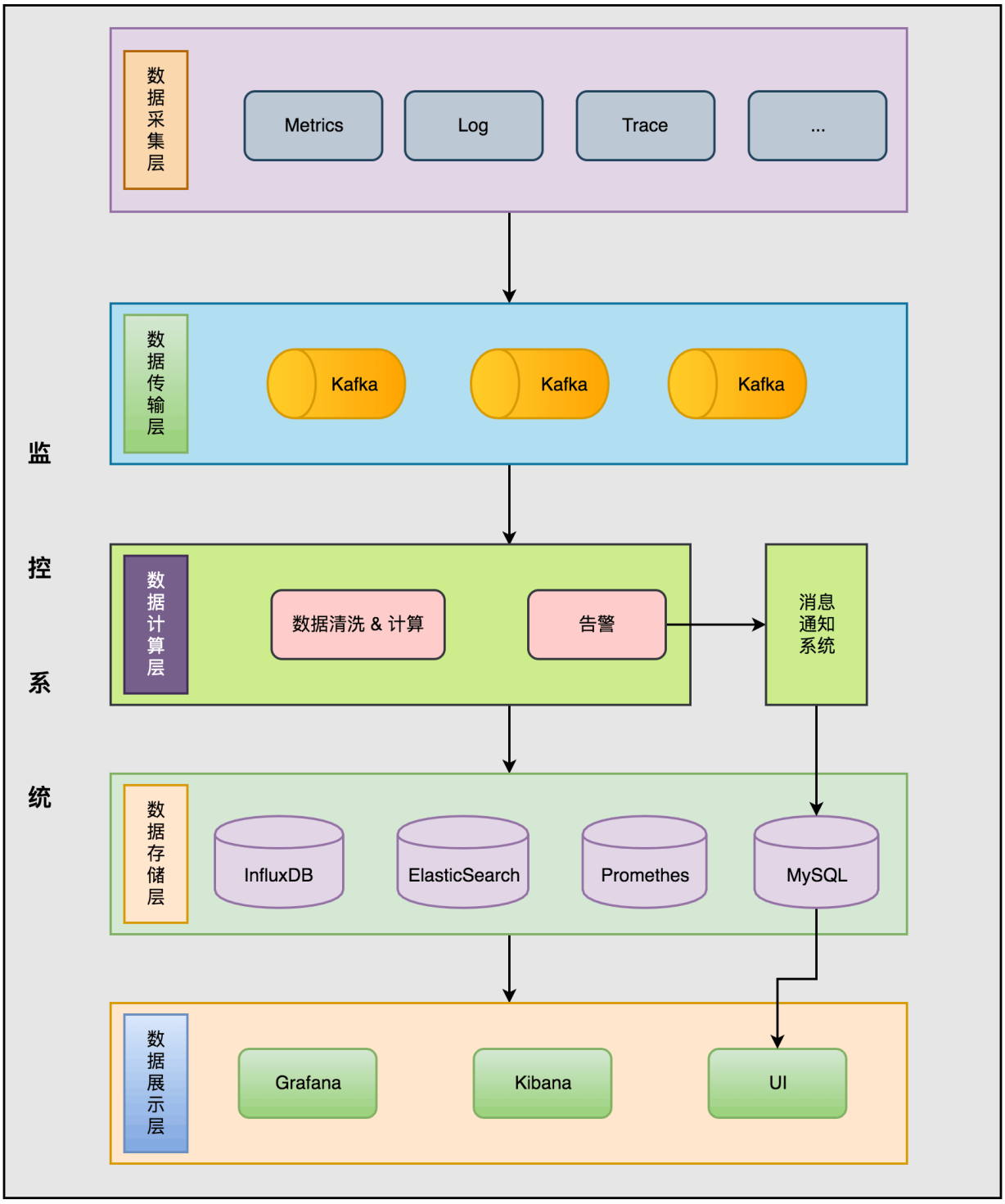
	OpenTSDB	KariosDB	Promethes	InfluxDB
开源时间	2010	2013	2012	2013
类型	时序数据库	时序数据库	时序数据库	**分析数据库**
分布式	**支持**	**支持**	Federation	商业
存储	Hbase	**Cassandra**	定制	定制
采集模式	Push	Push	**Pull**	Push
Grafana集成	支持	支持	支持	支持
计算函数	良好	一般	丰富	丰富
告警模块	无	无（ZMon）	**有**	**有**
查询语言	HTTP API	HTTP API	**PromQA**	**InfluxQL**
Web UI	支持	支持	支持	支持
维度支持	Tag	Tag	**Label**	Tag+Field
预聚合	Roll-up	Roll-up	Recording Rule	Continous Query
实现语言	Java	Java	Golang	Golang
github活跃度	~3.2k	~1.3k	**~19.6k**	~14.6k
商业支持	无	无	无	有
适用	中大规模	中大规模	中小规模	中小规模

监控可视化图表的开源系统个人觉得最好看的就是 Grafana，在 8.2 节中搭建 Flink 监控系统的数据展示也是用的 Grafana，当然还可以利用 ECharts、BizCharts 等数据图表库做二次开发来适配公司的数据展示图表。

上面说了这么多，这里笔者根据自己的工作经验先谈谈几点自己对监控系统的心得：

1. **告警是监控系统第一入口，图表展示体现监控的价值**：告警是唯一可以第一时间反映运行状态，它承担着系统与人之间的沟通桥梁，通常告警消息又会携带链接跳转到图表展示，它作为第一入口并衔接上了整个监控系统。
2. **数据采集是监控的源泉**：数据采集是监控系统的源泉，如果采集的数据是错误的，将导致后面的链路（告警、数据展示）全处于无效状态，所以千万千万要保证数据采集的准确性和完整性。
3. **数据存储是监控最大挑战**：当机器、系统应用和监控指标等变得越多来多时，采集上来的数据是爆炸性增长的，将海量的监控数据实时存储到任何一个数据库，挑战都是不小的。

说完心得再来讲解到底一个监控系统真正该包含哪些东西呢？笔者觉得首先分 6 层：数据采集层、数据传输层、数据计算层、告警、数据存储、数据展示。在每层其实又包含了些内容，如下图所示。



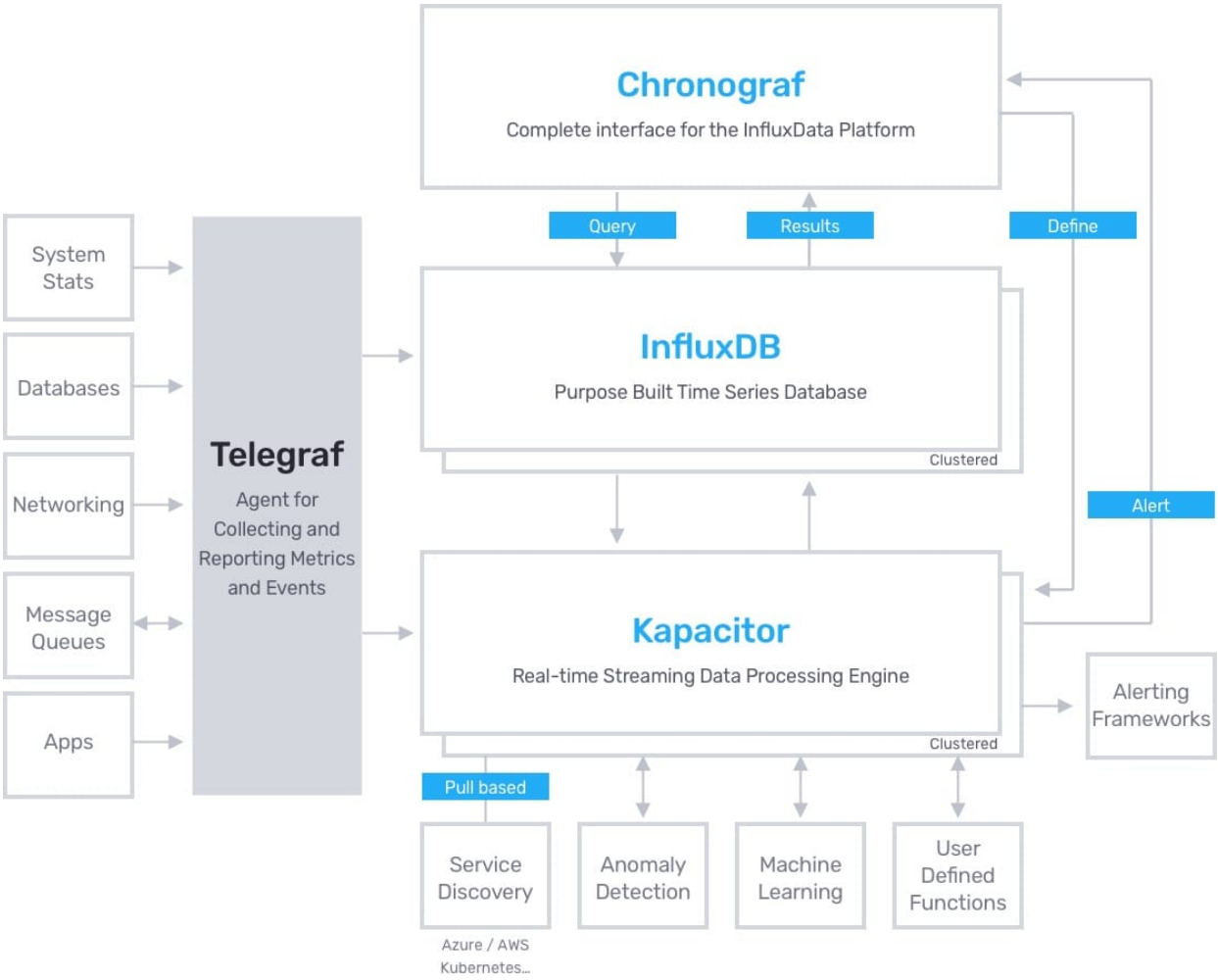
12.3.3 Metrics / Tracing / Logging 数据实时采集

在 12.1 节中讲解了日志数据如何采集，那么对于 Metrics 和 Tracing 数据该怎么采集呢?

Metrics

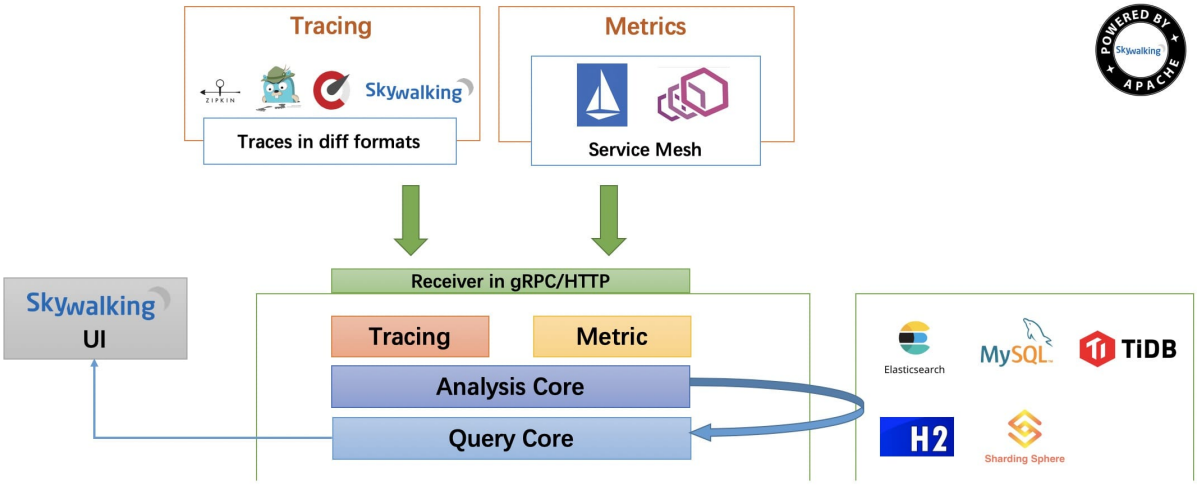
Metrics 数据其实这里我们要关心的是到底要采集哪些数据？机器数据、容器数据、中间件数据吗？如果是这些，其实市面开源的组件也有不少，比如 Telegraf，如果是场景比较简单的，也可以自己写 shell 脚本去采集机器的 CPU、内存、磁盘、load 等数据然后发送到 Kafka。

不过 Telegraf 已经集成了很多 Input 插件，它可以直接从其运行的容器和系统中提取各种指标、事件和日志，从第三方 API 提取指标，甚至通过 StatsD 和 Kafka 消费者服务监听指标。它还具有多种 Output 插件，可将指标发送到各种其他数据存储，服务和消息队列，包括 InfluxDB、Graphite、OpenTSDB、Datadog、ElasticSearch、Kafka 等。



Tracing

Tracing 这块，国内有一款优秀的工具 —— SkyWalking，它包括了分布式追踪、性能指标分析、应用和服务依赖分析等。其 6.x 版本的架构图如下：



从架构图中可以看到 SkyWalking 的核心是数据分析和度量结果的存储平台，通过 HTTP 或 gRPC 方式向 SkyWalking Collecter 提交分析和度量数据，SkyWalking Collecter 对数据进行分析和聚合，存储到 Elasticsearch、H2、MySQL、TiDB 等其一即可，最后通过 SkyWalking UI 的可视化界面对最终的结果进行查看。

整个链路其实已经很完成，但是我们其实就是想要其采集数据的部分，所以可以将其改造，拿到其采集后的数据后，我们直接发送到数据传输层（Kafka）。

12.3.4 消息队列如何撑住高峰流量

肯定有人想问前面的架构图中数据传输层为啥写的 Kafka 消息队列啊，为什么不使用其他的消息队列，这里和其他常用的消息队列做个简单对比。

RabbitMQ

采用 Erlang 语言实现的 AMQP 协议的消息中间件，起源于金融系统，现在用于在分布式系统中存储转发消息，其可靠性、可用性、扩展性、功能丰富等方面的卓越表现使得它一直被人认可。

Kafka

由 Scala 语言开发的一个分布式、多分区、多副本且基于 ZooKeeper 协调的分布式消息系统，它是一种高吞吐量的分布式发布订阅消息系统，以可水平扩展和高吞吐率而被广泛使用。

RocketMQ

阿里开源的消息中间件，由 Java 语言开发的，同样具备高吞吐量、高可用性、适合大规模分布式系统应用等特点。

因为通常来说，从采集工具采集上来的监控数据量会很大（含有 Metrics、Tracing、Tracing），在大促时候，那监控流量其实是会更猛，为了能够扛住如此暴增的流量，所以首先在整个系统架构选型的时候就要求考虑到数据量的大小以及未来的增长大小，而 Kafka 无论从性能吞吐量，还是在可靠性和运维方面来讲，对于监控系统无非是最佳的选择。

选择好了使用 Kafka 作为数据传输后，接下来得根据自己公司目前的监控数据流量和未来的增长情况来设置 Kafka Broker 的数量和 Metrics、Tracing、Tracing Topic 数据的分区数量，提前做好规划，防止后面因为数据量大而导致扩容和运维出现各种疑难问题。当出现 Kafka 的 Topic 数据堆积的时候，可以排查下到底是因为什么导致出现数据堆积的，查看消息的生产速度和数据的消费速度是否平衡，如果是消费速度跟不上生产速度，那么就得考虑在消费端扩大并行度，增加消费能力，如果是因为下游的消费端因为重启过等原因，这种情况下可能会出现短暂的消费数据堆积，如果消费速度够快的化，其实过段时间就会追平的。再者还得看堆积数据的重要性，如果是对于告警作业来说，堆积了几小时的数据其实对告警已经意义不大了，那么要做的是提高作业的并行度后，将作业从 Kafka 最新的数据开始消费，而不是接着从上一次提交的 offset 开始消费，如果堆积的数据是像金融行业的交易数据或者订单数据，那么这种数据是千万不能丢的，尽管消费慢，也不能够丢掉任何一条数据，否则损失可不止一点点。希望你可以在遇到这种数据堆积问题的时候，冷静去思考和分析数据的类型，沉着去排查问题的根因，及时想出解决问题的办法。

12.3.5 指标数据实时计算

数据计算层主要做的工作就是从 Kafka 集群消费数据，然后对数据进行过滤、聚合、重新组装、关联等操作。举个例子，采集工具每 5 秒采集一次机器或者应用等监控数据，然后上报这些 Metrics 数据，如果全部将原始的数据存储到时序数据库中，那么在数据量如此大的情况下对于数据库的压力很大，除此之外其实数据展示的时候也没必要展示如此详细的监控信息，那么完全可以在采集层就设置稍长点的采集时间间隔，或者在计算层将数据做一个聚合求出每 30 秒内的平均值 / 最大值 / 最小值 / 75 分位数的值，然后将这种计算后的值直接存储到数据库中去，这样数据展示的时候就可以直接使用存储的数据，而不需要再次额外做其他的计算。

除了这种简单的计算外，告警也是需要根据告警规则做很多的计算。在计算层因为是直接去消费 Kafka 的数据，作为一个消费端，所以就经常会遇到各种问题，比如数据堆积了怎么办，在上面提出了笔者的部分思考，在 3.7 节中笔者也讲过使用 Flink Kafka Connector 可能会遇到的问题，如果你不记得了，可以再去复习一下。

在遇到数据堆积问题时，作业调整并行度后重启，状态的恢复也是要考虑的一个问题，在 4.3 节中讲解了如何从 Checkpoint 或者 Savepoint 中恢复作业的状态。

12.3.6 提供及时且准确的根因分析告警

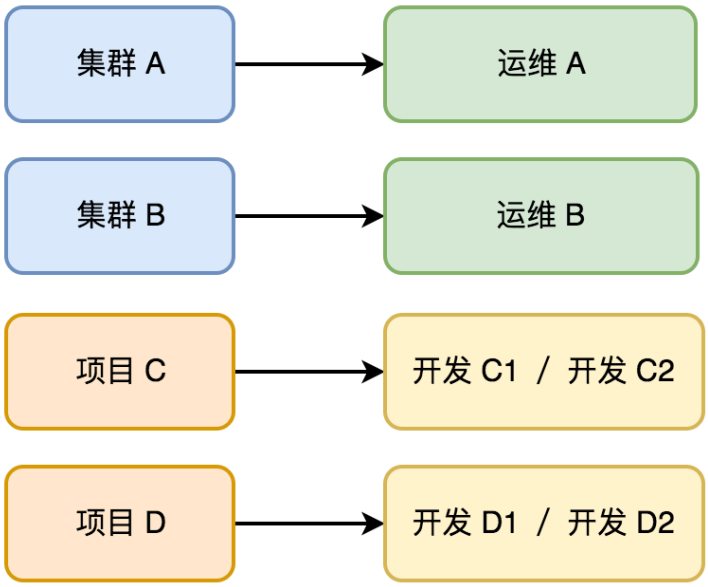
告警本质

没有多少系统的告警是设计得当的，良好的告警设计是一项非常困难的工作。使用传统的一些监控系统，比如 Zabbix，在监控指标多的情况下，每天至少产生上万条告警信息，在遇到网络抖动或者机器宕机的情况下，每种指标都将会触发告警，而触发告警后又没有将这些告警信息做过滤去重、聚合收敛，那么告警消息到达用户时则是狂轰乱炸，长此下去收到告警消息的人则会对告警信息感到麻木，即使下次真有严重的告警时，他也会如平常一样默默的短信关掉或者直接不看。因为用户觉得从你这些告警里面根本无从找到故障根本原因，可能登录到机器去查看问题的时候，故障已经恢复了，其实

真正要想找到问题的根因，从来都不是一件容易的事，所以业界流传着 ”监控容易做，告警很难报“ 的说法。

告警通知对象

对于告警来说，告警信息该通知给谁也是很重要的，举个例子，不同的集群负责人可能不同，那么就需要将不同集群的告警信息通知到不同的负责人，不同项目开发的负责人也不同，如果项目应用出现异常的话，也是需要直接通知到该项目应用的负责人。那么就需要有个地方保存着项目应用 ID 或者集群名与对应的负责人（可能多个）之间的关系。在采集工具处要将集群名、项目应用名或者 ID 等数据一起采集上来，这样后面流处理时才能够拿到这种关键信息去查找对应的通知对象。



告警通知方式

除了通知对象外，还有一个比较重要的是通知方式了。通常考虑通知方式要从功能性、成本性、方便性等几个方面考虑，比如笔者公司平时使用钉钉办公居多，那么平时如果有告警的话，直接将告警信息推送到钉钉的话，可以很方便的查看，如果你们公司使用的是企业微信，那么可以考虑使用企业微信推送告警信息，另外还有的公司使用的是自己公司开发的办公软件，那么可以考虑通知方式接入自己公司的软件。除了办公软件之外，有时候是非常严重的告警，但是你下班了，没能够及时查看办公软件的信息，那么此时像短信或者电话等通知方式就会更及时。但是像平时工作时间的告警就没必要去发送短信或者电话通知了，因为大家都知道，接入这些短信和电话都是要付费的，对于如此多的告警信息来说，如果每条告警信息都发送短信和电话，那么这个监控系统项目的成本就要耗费不少了，你可以自己估算一下，一天一万条告警信息大概一天要花多少钱。

通知方式

那么通知方式有这么多种，什么时候使用钉钉，什么时候使用邮件，什么时候使用短信和电话，或者什么时候它们一起使用（既发钉钉也发短信和电话）。这种通常是不同的监控告警规则配置不同的告警通知方式，比如机器宕机告警这种非常严重的告警，那么一定要使用电话通知，而对于像机器内存超过 75% 这种告警，则使用普通的办公软件推送就行。另外可能就是不同的时间段配置不同的通知方式，比如工作日的白天 9:00 到晚上的 9:00 使用办公软件推送即可，而周末或者工作日晚上 9 点之后开启短信和电话一起通知。这种组合的通知方式也是需要和对应的告警规则一起绑定在一起的，你需要在告警规则表中就增加这种字段去表示该规则的通知方式。

告警规则的设计

上面提到的这些通知对象和通知方式是要和具体的监控规则关联在一起的，这样才知道告警消息到底该使用哪种通知方式去通知哪些人。但是具体的告警规则该怎么去设计呢？其实要回答这个问题很难，也很难做到一个通用的设计，因为监控的对象太多了（通常公司的监控都有机器、容器、应用服务、各种中间件），每个对象中要监控的指标又是一大堆，针对这些指标又有不同类型的告警，有的场景下难以用一条规则去形容这个监控指标要达到什么情况下算是要告警，所以这也是为啥告警是监控系统中最难处理的。

那么通用的告警规则一般包含哪些内容呢？就拿机器告警来说，我们不妨可以来看下阿里云上面的机器告警设计（当你实在没思路的时候，多参考多学习别人的思想未必是坏事），如下图所示。

它首先包含了关联的资源，表明这个告警规则关联的是哪台机器，因为这是对于单台机器设置的告警规则，通常在自己公司不会对单台设置告警规则，而是直接将一些通用的告警规则应用在某个集群上，让这个集群有个默认的告警规则，但是也支持去修改单个机器的告警规则，除了机器，那就是关联到应用和中间件了，这里就是上面笔者说的告警对象。接着就是设置规则的名称，这个规则描述中可以选择该告警对象里面有的一些监控指标，每种指标判定它为告警的计算方式，比如机器的 CPU 在 5 分钟内的平均值连续出现 3 次大于 90% 就触发一条 CPU 告警。这种就是一条可以用表达式形容出来的告警指标，像 11.2 节中的宕机告警就不能用同样的方式来表达它触发了告警，所以有的情况需要特殊的对待。然后就是设置告警的通知收敛策略，比如出现了一条机器 A CPU 告警，这时你可能就会去查看该告警，然后去排查机器 CPU 为啥会突然飙高，但是这个排查处理是要一定的时间的，并不可能立马就可以揪出根因，但是如果 5 分钟不到立马又发来了一条机器 A CPU 告警，这时你会不会心情有点不太好了，心想这里还没有解决问题呢，怎么又来一条同样的告警，那么这条重复的告警不仅对于排查问题没有任何帮助，反倒还会增加运维人员的心理压力，所以一个合理的告警收敛是多

么的重要，比如机器 A 发生 CPU 告警后的两小时内如果该机器再出现同类型的告警，则选择不通知，如果超过两小时了又出现该问题则再告警出来。当然，这里笔者说的两小时并不一定最适合，因为机器的运行状态在很多情况下都是差别很大的，大家可以根据自己公司的场景为每种监控指标设置一个合理的通知静默的时间。最后就是通知对象和通知方式了，你可以看到上图中是以一个通知组的形式来将告警发送给这个通知组里面的所有人，报警级别和告警的通知方式关联在一起，严重时使用电话+短信+邮件+钉钉机器人来通知，笔者觉得这种方式还是比较适合的。

根因分析告警

上面这种告警规则设计比较通用，但是通用带来的后果是，当一个告警触发了，往往会伴随着很多告警同时触发。举个例子：当机器宕机了，伴随着肯定会有该机器上运行的服务也会报服务不可用的告警，服务不可用后则会导致访问应用出现各种 5XX 状态码，那么如果之前配置了应用访问出现 5XX 状态码则告警的告警规则，这时开发也会收到一大堆的告警说自己负责的应用出现了掉线，应用出现了很多 5XX 等。总之碰到这种情况下，告警消息如风暴一样将历史的告警淹没，那么开发和运维根本无从下手到底该看哪条告警，可能造成告警最根本的原因已经被新的告警消息淹没了。

出现这种问题的关键在于，通用的告警规则设计时就是考虑以单个 Metrics 为一个告警指标，而没有将系统中成千上万个监控指标彼此关联起来，这样就会造成数据孤岛，那么有没有办法将这些 Metrics 数据关联起来呢。因为在流处理的场景所有的数据都是从 Kafka 流过来的，那么通常异常数据也都是在一起的，那么当出现异常的时候可以选择稍微聚合异常后一段时间（比如 30 秒）的数据，来判断这段时间内的异常数据是否有关联，比如机器 A CPU Metrics 高的时候，有个运行在机器 A 上的应用 CPU Metrics 也很高。那么这时候是否可以认为机器 CPU 高的原因是因为部署在这台机器上的 xxx 应用导致的机器 CPU 高？这是基于相关性的根因分析，从异常的 Metrics 数据中找出最相关的异常 Metrics。另外还可以基于决策树的根因分析来判断告警的根因到底是啥？

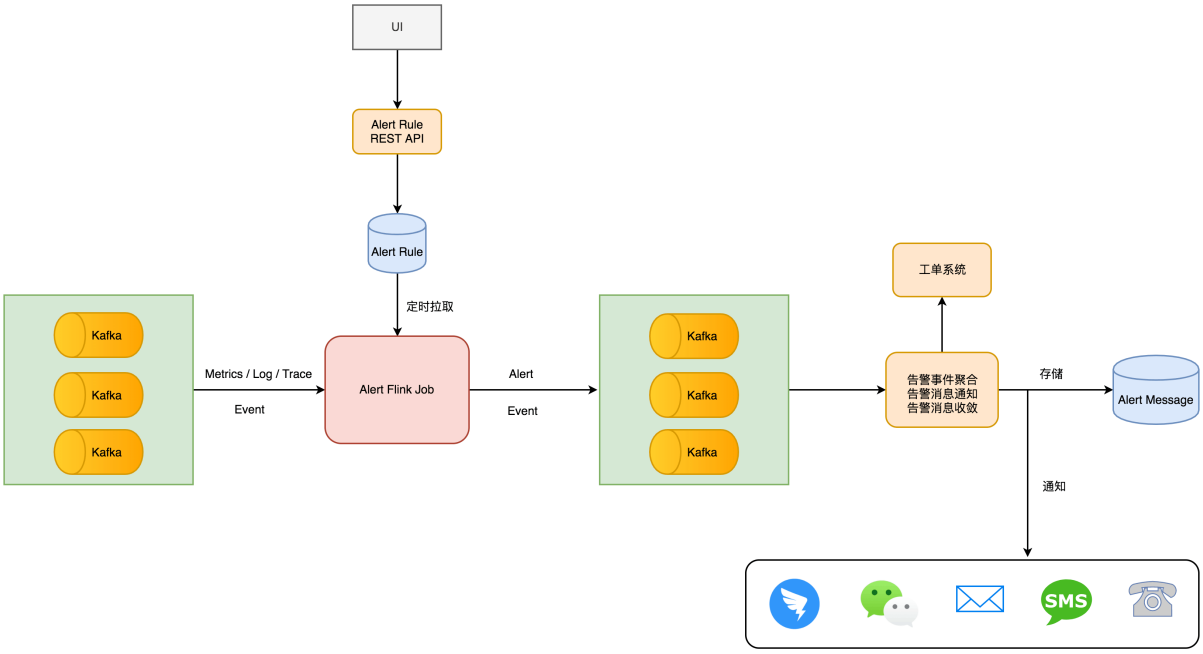
告警事件解耦

在上面提到了在网络抖动或者机器宕机的情况下会产生告警风暴，但是如果我们在流处理时对它们做一个告警关联聚合，找到根因后那么再发送出来的消息，不仅质量会提高，而且数量也会减少，这样才是一个好的监控告警系统该有的样子。那么如果可以做到告警消息的聚合呢？其实我们可以将原始监控数据的计算与告警事件的通知拆分成两个作业，前一个作业实时的消费 Kafka 数据，根据告警规则去判断数据是否告警，告警后则生成新的告警事件，如果直接在同一个作业中将告警事件跟下游的消息通知服务做 HTTP 请求交互，因为在流处理中与第三方服务做交互时会大大影响消息处理的速度，反而可能会造成反压问题（关于反压问题的处理方式可以参考 9.1 节），所以这里就将告警事件解耦。上游告警计算作业产生告警事件直接发送到 Kafka，然后下游作业还是消费 Kafka 数据，因为下游作业的告警事件数据已经小很多了，所以再去和通知服务做交互其实已经没多大问题了，不会出现反压问题了。这样将两个系统做拆分解耦后，下游将可以很好的将这些告警事件做聚合了，比如对机器 IP 做 keyby，然后可以聚合每台机器 30 秒（这个时间可以根据告警的延迟和告警聚合程度做个平衡）内发生的告警事件，聚合后拿到多个告警事件就可以很好的去做告警事件的关联，找出告警的根因，然后还可以将这些事件聚合组织成一条告警消息，最后就可以做到告警消息的质量提高，而且数量可以大大降低。告警消息的质量和数量通常也是衡量一个告警系统的关键因素。

告警工单记录告警解决过程

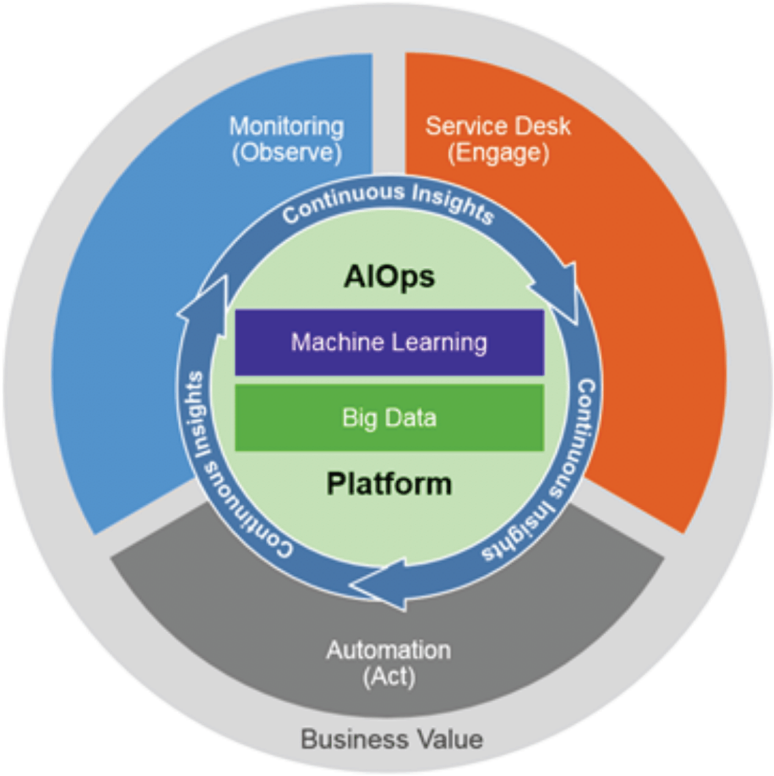
当告警系统真正做好了之后，可以考虑将告警的整个解决过程通过工单的方式记录下来，出现告警，告警接收人负责去处理这个告警，在解决这个告警时使用的是什么办法，排查了哪些问题，需要将问题的排查和解决的过程一一记录下来，当问题最终解决后，要将工单的状态给关闭。工单在这里的意义就在于记录问题的处理过程，下次遇到同类型的问题时可以参考之前的解决方法。

综合上面这几点，整个告警的通用的架构如下：



12.3.7 AIOps 智能运维道路探索

AIOps 是最近比较火的一个概念，它中文名称叫智能运维，定义如下图所示：



从上图中笔者理解到的是：

- 监控发现：基于历史数据、实时数据、预测数据，发现问题，预测问题。
- 服务管控：所有服务都可以人为管理控制。
- 自动决策：无人值守，系统自愈。

上面这三个都集成在 AIOps 平台，然而这个平台的核心由大数据、机器学习组成，通过大数据去做历史数据的分析、实时数据的异常检测、未来数据的实时预测，通过机器学习算法去做异常检测、实时预测、弹性调度、自动化决策，整个平台不断的迭代和优化来完善监控发现、服务管控和自动决策三者的功能。

那么 AIOps 在监控平台可以有哪些尝试呢，比如它可以利用其大数据平台和机器学习算法去发现问题：

- 时间序列异常
- 日志分析异常
- 设备性能异常

发现问题后，可以自动去分析问题：

- 多维下钻分析
- 关联事件分析
- 容量预估分析

在分析问题后能够去解决问题：

- 扩容
- 决策
- 调度
- 优化

像上面这样说，可能你不太明白，举个例子：一台机器同时报了 CPU、内存、磁盘 的告警，基于 AIOps 分析到这些告警后发现这台机器上运行的服务已经很多了，资源比较紧张了，那么就会考虑自动化的去做机器扩容，然后将该机器上运行的某个或几个服务调度运行在另外空闲的机器上去。关于 AIOps 的内容大家可以自行去搜索相关的资料进行查阅，本书就不再做过多的讲解。

12.3.8 如何保障高峰流量实时写入存储系统的稳定性

在前面也已经提到过监控数据的爆炸式增长，通常稍大点的公司日均处理监控数据起码超过 100T，实时数据的 QPS 超过 20万 / 秒，那么要满足这么大的数据量实时的写入，而且还要保证存储系统的稳定性，除了给存储系统最好的硬件条件外（当然并不是所有的公司都愿意花这么贵的价钱配置好的机器），还有什么优化办法吗？笔者就根据自己的经验和理解来讲几点：

- 消息瘦身：一条事件中的数据只放监控展示需要的指标，不要的指标直接丢弃。
- 批量写入：任何存储系统想要提高写入性能无非就是要批量的写，提升写入的吞吐量。
- 异步：能用异步一定要用异步。
- 数据预聚合：比如原始的监控数据是每隔 5 秒采集一次，那么可以在流处理的时候做预聚合成 30 秒，并且可以提前将最大值、最小值、平均值等计算好。
- 多线程：多线程写入，成倍提高写入的速度。
- 调优存储系统：在上面几步都调整好了之后，就要开始不断的调优存储系统，当然不同的存储系统可能不同，比如 HBase、Druid、ElasticSearch 等的调优是各不相同的，你可以根据自己公司的技术选型做一个合理的调优。

虽然上面说好几种优化方法，但是这种调优也是有瓶颈的，如果可以适当提升一下硬件，相信这个写入速度会得到很大的增长。因为数据写入的及时性和存储系统的稳定性关乎着后面数据展示的命运，所以得确保这步不掉链子。笔者最开始的告警系统就是基于存储系统的数据来做的，当时因为存储系统的稳定性问题导致告警系统和数据可视化图表处于不可用状态，所以后来才换成了使用 Flink 去做实时告警。

监控数据都存储在存储系统之后，那么接下来就是数据可视化图表展示了。

12.3.9 监控数据使用可视化图表展示

数据可视化图表展示是监控系统的门面，它如果做的很酷炫，很高大上，那么别人就会认为你们这个监控系统做的很棒，所以如何去做一个完美的可视化图表展示系统就尤为重要了。自己从 0 开始去开发一个这样的可视化系统的话可能没有多少公司有这个人力，况且还可能做的不够美观和酷炫，那么现成有这种既美观有酷炫，还支持自定义可视化图表的系统吗？这里笔者强烈推荐 Grafana。

Grafana 介绍

Grafana 是一个跨平台的开源的度量分析和可视化工具，可以通过将采集的数据查询然后可视化的展示，并及时通知。它主要有以下六大特点：

1. 展示方式：快速灵活的客户端图表，面板插件有许多不同方式的可视化指标和日志，官方库中具有丰富的仪表盘插件，比如热图、折线图、图表等多种展示方式。
2. 数据源：Graphite，InfluxDB，OpenTSDB，Prometheus，Elasticsearch，CloudWatch 和 KairosDB 等。
3. 通知提醒：以可视方式定义最重要指标的警报规则，Grafana 将不断计算并发送通知，在数据达到阈值时通过 Slack、PagerDuty 等获得通知。
4. 混合展示：在同一图表中混合使用不同的数据源，可以基于每个查询指定数据源，甚至自定义数据源。
5. 注释：使用来自不同数据源的丰富事件注释图表，将鼠标悬停在事件上会显示完整的事件元数据和标记。
6. 过滤器：Ad-hoc 过滤器允许动态创建新的键/值过滤器，这些过滤器会自动应用于使用该数据源的所有查询。

12.3.10 小结与反思

本节讲了一个监控系统的诉求和一个监控系统应该包含哪些内容，接着开始讲解监控系统的数据采集、数据传输、数据计算、告警、数据存储、数据展示整条链路中每个环节要做的事情，就拿告警来说，介绍了告警的本质、告警的通知方式、告警的通知对象、告警规则的设计、告警事件的解耦、告警的根因分析、告警工单的记录等，让大家清楚的知道一个真正的监控告警系统该长啥样，该怎么去设计，会遇到哪些问题，怎么去解决这些问题。另外还和大家对 AIOps 智能运维做了个简短的探索，

也希望你可以在自己公司设计新的监控系统时，可以将 AIOps 的理念和思想考虑进去。

你们公司有监控系统吗？你理解你们公司的监控系统吗？你觉得本节介绍的监控系统还缺少什么吗？如果由你主导设计一个监控系统，你会从哪些方面考虑呢？