可观测性

- **背景**
  - **相关元素特征**
    - **日志：**职责是记录离散事件，通过这些记录事后分析出程序的行为。输出日志较为容易，但收集和分析日志却可能会很复杂。
    - **追踪：**微服务时代，追踪就不只局限于调用栈了，一个外部请求完整的调用轨迹将跨越多个服务，同时包括服务间的网络传输信息与各个服务内部的调用堆栈信息。分布式追踪常被称为全链路追踪。追踪的主要目的是排查故障，如分析调用链的哪一部分、哪个方法出现错误或阻塞，输入输出是否符合预期，等等。
    - **度量：**指对系统中某一类信息的统计聚合。度量的主要目的是监控和预警，如某些度量指标达到风险阈值时触发事件，以便自动处理或者提醒管理员介入。
  - **相关产品**
    - 日志收集和分析大多被统一到 Elastic Stack（ELK）技术栈上，其中的 Logstash 有可能被 Fluentd 取代。
    - 度量方面，Prometheus即将成为云原生时代度量监控的事实标准
    - 各个服务之间是使用 HTTP 或 gRPC通信会直接影响追踪的实现，各个服务是使用哪种语言来编写，也会直接影响到进程内调用栈的追踪方式。这决定了追踪工具本身有较强的侵入性，通常是以插件式的探针来实现；也决定了追踪领域很难出现一家独大的情况，通常要有多种产品来针对不同的语言和网络。
- **事件日志**
  - **背景：**复杂的分布式系统很难只依靠 tail、grep、awk 来从日志中挖掘信息，往往还要有专门的全局查询和可视化功能。此时，从打印日志到分析查询之间，还隔着收集、缓冲、聚合、加工、索引、存储等步骤。
  - **输出日志原则**
    - **避免打印敏感信息**
    - **避免引用慢操作：**日志中打印的信息应该是上下文中可以直接取到的，如果需要专门调用远程服务或者从数据库获取、通过大量计算才能取到，需要考虑必要性。
    - **避免打印追踪诊断信息：**日志中不要打印方法输入参数、输出结果、方法执行时长之类的调试信息。日志的职责是记录事件，追踪诊断应由追踪系统去处理。
    - **应该包含处理请求时的 TraceID：**分布式追踪通过它把请求在各个服务中的执行过程串联起来，即使是单体系统，在日志对快速定位错误时仍然有价值。
    - **应该包含系统运行过程中的关键事件：**日志的职责就是记录事件，进行了哪些操作、发生了与预期不符的情况、运行期间出现未能处理的异常或警告、定期自动执行的任务等。但应判断清楚事件的重要程度，选定相匹配的日志的级别。
    - **应该包含启动时输出配置信息：**对于系统启动时或者检测到配置中心变化时更新的配置，应将非敏感的配置信息输出到日志中，初始化配置的逻辑一般只会执行一次，不便于诊断时复现，所以应该输出到日志中。
  - **收集和缓冲**
    - **日志收集发展历史**
      - 最初，ELK 中日志收集与加工聚合的职责都由 Logstash 来承担的，Logstash 除了部署在各个节点中作为收集的客户端以外，它还同时设有独立部署的节点，扮演归集转换日志的服务端角色。但是 Logstash 与它的插件是基于 JRuby 编写的，要跑在单独的 Java 虚拟机进程上，而且 Logstash 的默认的堆大小就到了 1GB。
      - 后来，Elastic.co 公司将所有需要在服务节点中处理的工作整理成以Libbeat为核心的Beats 框架，并使用 Golang 重写了一个功能较少，却更轻量高效的日志收集器即Filebeat。框架还包括度量和追踪工具，ELK可在一定程度上代替度量和追踪。
    - **缓冲必要性：**日志收集器不仅要保证能覆盖全部数据来源，虽然不追求绝对的完整精确，但追求在代价可承受的范围内保证尽可能地保证较高的数据质量。
    - **缓冲实现：**一种最常用的缓解压力的做法是将日志接收者从 Logstash 和 Elasticsearch 转移至抗压能力更强的队列缓存，如在 Logstash 之前架设一个 Kafka 或者 Redis 作为缓冲层，面对突发流量，Logstash 或 Elasticsearch 处理能力出现瓶颈时自动削峰填谷，甚至当它们短时间停顿，也不会丢失日志数据。
  - **存储和查询**
    - **背景：**收集、缓冲、加工、聚合后的日志数据，可以放入ES 中索引存储。
    - **ElasticSearch优势**
      （不可取代的原因）
      - **从数据特征的角度看：**日志是基于时间的数据流，虽然增长速度很快，但已写入的数据几乎没有再发生变动的可能。日志的数据特征决定了所有用于日志分析的 Elasticsearch 都会使用时间范围作为索引，由于能准确地预知未来的日期，因此索引都可以预先创建，免去了动态创建的寻找节点、创建分片、在集群中广播变动信息等开销。又由于所有新的日志都是“今天”的日志，只要建立logs_current这样的索引别名来指向当前索引，就能避免代码因日期而变动。
      - **从数据价值的角度看：**日志基本上只会以最近的数据为检索目标，随着时间推移，早期的数据将逐渐失去价值。这点决定了可以很容易区分出冷数据和热数据，进而对不同数据采用不一样的硬件策略。
      - **从数据使用的角度看：**分析日志很依赖全文检索和即席查询，对实时性的要求是处于实时与离线两者之间，即不强求日志产生后立刻能查到，但也不能接受日志产生之后按小时或天的频率来更新，而检索能力和近实时性，也正好都是ES的强项。
    - Elasticsearch 只提供了 API 层面的查询能力，它通常搭配Kibana 一起使用，可以将 Kibana 视为 Elastic Stack 的 GUI 部分。Kibana能探索数据并可视化，即把存储在ES中的数据被检索、聚合、统计后，定制形成各种图形、表格、指标、统计，以此观察系统的运行状态，找出日志事件中潜藏规律和隐患。
  - **加工和聚合**
    - **必要性：**日志是非结构化数据，一行日志中通常会包含多项信息，如果不做处理，那在 Elasticsearch 就只能以全文检索的原始方式去使用日志，既不利于统计对比，也不利于条件过滤。

- **加工：**Logstash将日志行中的非结构化数据，通过 Grok 表达式语法转换为结构化数据，还可能会根据需要调用其他插件来完成时间处理、查询归类等额外处理工作，然后以 JSON 格式（普遍）输出到 Elasticsearch 中。因此，Elasticsearch 便可针对不同的数据项来建立索引，进行条件查询、统计、聚合等操作。
- **聚合：**如果想从离散的日志中获得统计信息，再生成可视化统计图表，一种解决方案是通过 Elasticsearch 本身的处理能力做实时的聚合统计，便捷但消耗ES服务器的运算资源。另一种解决方案是在收集日志后自动生成某些常用的、固定的聚合指标，这种聚合就会在 Logstash 中通过聚合插件来完成。这两种聚合方式都有不少实际应用，前者一般用于应对即席查询，后者用于应对固定查询。

链路追踪

- **定义：**广义上讲，一个完整的分布式追踪系统应该由**数据收集、数据存储和数据展示**三个相对独立的子系统构成，而狭义上讲的追踪则特指链路追踪数据的收集部分。广义的追踪系统常被称为“APM 系统”。
- **追踪与跨度**
  - **追踪：**从客户端发起请求抵达系统的边界开始，记录请求流经的每一个服务，直到到向客户端返回响应为止，这整个过程就称为一次追踪Trace。
  - **跨度：**为了能够记录具体调用了哪些服务，以及调用的顺序、开始时点、执行时长等信息，每次调用服务前都要先埋入一个调用记录，这个记录称为一个跨度Span。Span 的数据结构应该足够简单，以便于能放在日志或者网络协议的报文头里；也至少含有时间戳、起止时间、Trace ID、当前 Span 的 ID、父 Span 的 ID 等。
  - **从目标来看：**链路追踪的目的是为排查故障和分析性能提供数据支持，系统对外提供服务的过程中同时持续地生成 Trace，按次序整理好 Trace 中每一个 Span 所记录的调用关系，便能绘制出一幅系统的服务调用拓扑图。根据拓扑图中 Span 记录的时间信息和响应结果就可以定位到缓慢或者出错的服务；将 Trace 与历史记录进行对比统计，就可以从系统整体层面分析服务性能，定位性能优化的目标。
  - **从实现来看：...**
    - **功能上的挑战：**来源于服务的异构性，各个服务可能采用不同程序语言，服务间交互可能采用不同的网络协议，每兼容一种场景，都会增加功能实现方面的工作量
    - **非功能性挑战**
      - **低性能损耗：**分布式追踪不能对服务本身产生明显的性能负担。
      - **对应用透明：**追踪系统通常是运维期才事后加入的系统，应该尽量以非侵入或者少侵入的方式来实现追踪，对开发人员做到透明化。
      - **随应用扩缩：**现代的分布式服务集群都有根据流量压力自动扩缩的能力，这要求当业务系统扩缩时，追踪系统也能自动跟随，不需要运维人员人工参与。
      - **持续的监控：**要求追踪系统必须能够 7x24 小时工作，否则就难以定位到系统偶尔抖动的行为。
- **数据收集**
  - **基于日志的追踪：...**
    - **优点：**日志追踪对网络消息完全没有侵入性，对应用程序只有很少量的侵入性，对性能影响也非常低。
    - **缺陷：**直接依赖于日志归集过程，日志本身不追求绝对的连续与一致，这也使得基于日志的追踪往往不如其他两种追踪实现来的精准；业务服务调用与日志归集不是同时完成的，也通常不由同一个进程完成，有可能发生业务调用已经顺利结束了，但日志归集不及时或精度丢失，导致日志出现延迟或缺失记录，产生追踪失真。
    - **代表产品：**Spring Cloud Sleuth
  - **基于服务的追踪：...**
    - **优点：**追踪的精确性与稳定性都有所保证，不必再依靠日志归集来传输追踪数据。也可以通过这种方式手机追踪诊断信息。
    - **缺陷：**比基于日志的追踪消耗更多的资源，也有更强的侵入性
    - **代表产品：**Pinpoint、SkyWalking
  - **基于边车代理的追踪：**...
    - **优点：**与程序语言无关，只要通过网络HTTP或gRPC来访问服务就可以被追踪到；有自己独立的数据通道，追踪数据通过控制平面进行上报，避免了追踪对程序通信或者日志归集的依赖和干扰，保证了最佳的精确性。
    - **缺陷：**服务网格现在还不够普及；边车代理本身的对应用透明的工作原理决定了它只能实现服务调用层面的追踪，像本地方法调用级别的追踪诊断是做不到的。
    - **代表产品：**Envoy

聚合度量

- **定义：**度量Metrics的目的是揭示系统的总体运行状态，度量总体上可分为**客户端的指标收集**、**服务端的存储查询**以及**终端的监控预警**三个相对独立的过程。
- **指标收集**
  - **如何定义指标：...**
    - **计数度量器Counter：**对有相同量纲、可加减数值的合计量，如服务调用次数、网站访问人数。
    - **瞬态度量器Gauge：**表示某个指标在某个时点的数值，如虚拟机堆内存的使用量、网站在线人数。
    - **吞吐率度量器Meter：**用于统计单位时间的吞吐量，即单位时间内某个事件的发生次数，如每秒发生了多少笔事务交易。
    - **直方图度量器Histogram：**两个坐标分别是统计样本和该样本对应的某个属性的度量，以长条图的形式表示具体数值。如地区历年GDP变化。
    - **采样点分位图度量器Quantile Summary：**统计学中通过比较各分位数的分布情况的工具，用于验证实际值与理论值的差距，评估理论值与实际值之间的拟合度。
  - **如何将这些指标告诉服务端：**拉取式采集Pull指度量系统主动从目标系统中拉取指标，推送式采集Push由目标系统主动向度量系统推送指标。
  - **指标应该以怎样的网络访问协议、取数接口、数据结构来获取：**出现过如网络管理中的SNMP、Windows 硬件的WMI、以及此前提到的Java 的JMX，但收集器支持哪种协议没有强求。
  - **Prometheus**
    - **背景：**一般来说，度量系统只会支持其中一种指标采集方式，因为度量系统的网络连接数量，以及对应的线程或者协程数可能非常庞大，

如何采集指标将直接影响到整个度量系统的架构设计。而Prometheus 基于 Pull 架构的同时还能够有限度地兼容 Push 式采集，是因为它有 Push Gateway 的存在。

- **Push Gateway**
  - **定义：**一个位于 Prometheus Server 外部的相对独立的中介模块，将外部推送来的指标放到 Push Gateway 中暂存，然后再等候 Prometheus Server 从 Push Gateway 中去拉取。
  - **目的：**解决 Pull 的一些固有缺陷，譬如目标系统位于内网，外网的 Prometheus 是无法主动连接目标系统的，只能由目标系统主动推送数据；又譬如某些小型短生命周期服务，可能还等不及 Prometheus 来拉取，服务就已经结束运行了，因此也只能由服务自己 Push 来保证度量的及时和准确。
- **Exporter**
  - **背景：**Prometheus只允许通过 HTTP 访问度量端点这一种访问方式。如果目标提供了 HTTP 的度量端点，否则就需要一个专门的 Exporter 来充当媒介。
  - **定义：**是目标应用的代表，既可以独立运行，也可以与应用运行在同一个进程中。Exporter 以 HTTP 协议返回符合 Prometheus 格式要求的文本数据给服务器。

- **存储查询**
  - **定义：**指标从目标系统采集过来之后，应存储在度量系统中，以便被后续的分析界面、监控预警所使用。
  - **指标特征：**每一个度量指标由时间戳、名称、值和一组标签构成，除了时间之外，指标不与任何其他因素相关。指标的数据总量固然是不小的，但它没有嵌套、没有关联、没有主外键，不必关心范式和事务。
  - **时序数据库：**用于存储跟随时间而变化的数据，并且以时间（时间点或者时间区间）来建立索引的数据库。
  - **根据特征做出改进策略：**写操作，时序数据通常只是追加，很少删改或者根本不允许删改。针对数据热点只集中在近期数据、多写少读、几乎不删改、数据只顺序追加这些特点，时序数据库被允许做出很激进的存储、访问和保留策略。
    - 以日志结构的合并树LSM-Tree代替传统关系型数据库中的B+Tree作为存储结构，LSM 适合的应用场景就是写多读少，且几乎不删改的数据。
    - 设置激进的数据保留策略，譬如根据过期时间（TTL）自动删除相关数据以节省存储空间，同时提高查询性能。对于普通数据库来说，数据会存储一段时间后就会被自动删除这种事情是不可想象的。
    - 对数据进行再采样（Resampling）以节省空间，譬如最近几天的数据可能需要精确到秒，而查询一个月前的数据时，只需要精确到天，查询一年前的数据时，只要精确到周就够了，这样将数据重新采样汇总就可以极大节省存储空间。
    - 时序数据库中甚至还有一种并不罕见却更加极端的形式，叫作轮替型数据库，以环形缓冲的思路实现，只能存储固定数量的最新数据，超期或超过容量的数据就会被轮替覆盖，因此也有着固定的数据库容量，却能接受无限量的数据输入。
- **监控预警**
  - Prometheus 应算是处于狭义和广义的度量系统之间，在生产环境下，大多是 Prometheus 配合 Grafana 来进行展示的，这是 Prometheus 官方推荐的组合方案，但该组合也并非唯一选择，如果要搭配 Kibana 甚至 SkyWalking。
  - **监控目的：**良好的可视化能力对于提升度量系统的产品力十分重要，长期趋势分析、对照分析、故障分析等分析工作，既需要度量指标的持续收集、统计，还需要对数据进行可视化，才能让人更容易地从数据中挖掘规律。
  - **预警：**Prometheus 提供了专门用于预警的 Alert Manager，将 Alert Manager 与 Prometheus 关联后，可以设置某个指标在多长时间内达到何种条件就会触发预警状态，触发预警后，根据路由中配置的接收器，譬如邮件接收器、Slack 接收器、微信接收器、或者更通用的 WebHook接收器等来自动通知用户。