1. 引言

1.1 目的

本文档为了帮助研发了解 SkyWalking 的功能及通过一个案例入门

1.2 背景

随着项目规模越来越大，为了对内提高诊断效率和对外 SLA 的追求，对于业务系统的掌控度的要求越来越高，主要体现在：

- 对于第三方依赖的监控，实时/准实时了解第三方的健康状况/服务品质，降低第三方依赖对于自身系统的扰动（服务降级、故障转移）
- 对于容器的监控，实时/准实时的了解应用部署环境（CPU、内存、进程、线程、网络、带宽）情况，以便快速扩容/缩容、流量控制、业务迁移
- 业务方自身的调用情况，方便做容量规划，同时对于突发的请求也能进行异常告警和应急准备
- 自身业务的监控、性能监控，实时/准实时的了解自身的业务运行情况，排查业务瓶颈，快速诊断和定位异常，增加对自己业务的掌控力

在这种情况下，一般都会引入 APM系统，通过各种探针采集数据，收集关键指标，同时搭配数据呈现和监控告警，能够解决上述大部分问题，同时对于企业来说，能够更精确的了解资源的使用情况，对于成本核算和控制也有非常大的裨益。

随着RPC框架、微服务、云计算、大数据的发展，同时业务的规模和深度相比过往也都增加了很多，一次业务可能横跨多个模块/服务/容器，依赖的中间件也越来越多，其中任何一个节点出现异常，都可能导致业务出现波动或者异常，这就导致服务质量监控和异常诊断/定位变得异常复杂，于是催生了新的业务监控模式：调用链跟踪。

1.3 类型

技术储备调研：了解 SkyWalking 是否可以应用到我们的项目，满足我们的需求

1.4 定义

SLA：Service-Level-Agrement

APM：Application Performance Management & Monitoring

OAP：Skywalking 把 collector、aggregator、alarm 集成为 OAP（Observability Analysis Platform），并且可以通过集群部署，不同的实例可以分别承担 collector 或者 aggregator + alarm 的角色

CPM: 每分钟请求调用的次数

1.5 参考资料

http://skywalking.apache.org/zh/

https://chenyongjun.vip/articles/140

https://blog.csdn.net/maskkiss/article/details/96872693

https://www.e-learn.cn/content/qita/2480968

2. 调研过程

2.1 思路介绍

阅读官方文档，了解 SkyWalking 的基本概念，然后通过一个案例入门 SkyWalking 的使用，了解 SkyWalking 的功能。

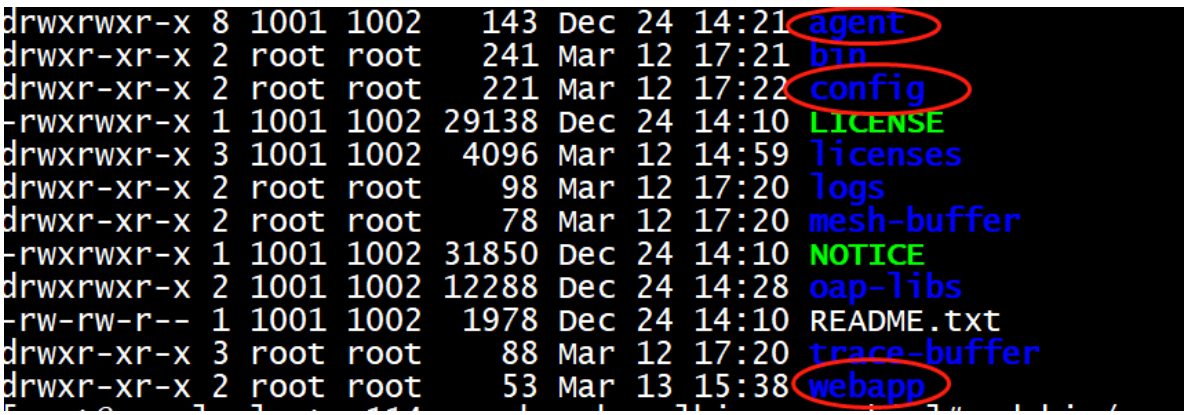
其他需要关注点：

- 稳定性：是否主流、持续维护
- 性能：cpu占用、内存占用、流量消耗、电量消耗
- 安全性：
- 成本：接入成本、是否收费

2.2 通过一个案例来入门 SkyWalking

SkyWalking 服务部署及 java 服务集成

```
# 下载官方编译好的安装包 http://skywalking.apache.org/downloads/
wget https://mirror.bit.edu.cn/apache/skywalking/6.6.0/apache-skywalking-apm-6.6.0.tar.gz

# 解压缩
tar zxf apache-skywalking-apm-6.6.0.tar.gz

# 修改 wep 服务端口，默认是 8080
vim /opt/apache-skywalking-apm-bin/webapp/webapp.yml

server:
  # 默认是 8080，改为 10000
  port: 10000

collector:
  path: /graphql
  ribbon:
    ReadTimeout: 10000
    # Point to all backend's restHost:restPort, split by ,
    listOfServers: 127.0.0.1:12800

# 启动 collector 和 webapp 服务
/opt/apache-skywalking-apm-bin/bin/start.sh
```



```
agent：探针配置
config：collector 配置
webapp：SkyWalking UI 配置
```

```
# java 服务集成
java -javaagent:/opt/apache-skywalking-apm-bin/agent/skywalking-agent.jar -jar
learn-fegin.jar
```

探针配置

```
/opt/apache-skywalking-apm-bin/agent/config/agent.config
```

配置参数名称	配置含义
agent.namespace	跨进程链路中的header，不同的namespace会导致跨进程的链路中断
agent.service_name	一个服务（项目）的唯一标识，这个字段决定了在sw的UI上的关于service的展示名称，尽量采用英文
agent.sample_n_per_3_secs	每3秒采集Trace的数量，默认为负数，代表在保证不超过内存Buffer区的前提下，采集所有的Trace
agent.authentication	与collector进行通信的安全认证，需要同collector中配置相同
agent.span_limit_per_segment	Skywalking每个segment的大小
agent.ignore_suffix	忽略特定请求后缀的trace
agent.is_open_debugging_class	探针调试开关，如果设置为true，探针会将所有操作字节码的类输出到/debugging目录下
collector.backend_service	探针需要同collector进行数据传输的IP和端口
logging.max_file_size	日志文件最大大小，默认为300M（单位：B），超过则生成新的文件
logging.level	记录日志级别，默认为DEBUG

collector 配置

```
/opt/apache-skywalking-apm-bin/config/application.yml
```

告警规则

```
/opt/apache-skywalking-apm-bin/config/alarm-settings.yml
```

指标 + 链路追踪

问题

```
elasticsearch版本问题
```

2.3 SkyWalking介绍

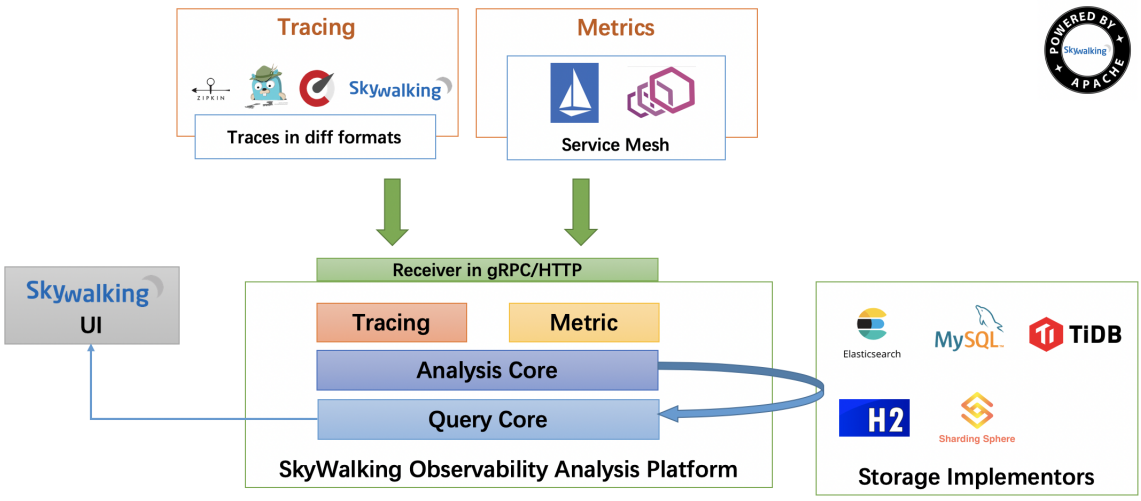
什么是 SkyWalking?

SkyWalking 是观察性分析平台和应用性能管理系统。

提供分布式追踪、服务网格遥测分析、度量聚合和可视化一体化解决方案.

支持Java, .Net Core, PHP, NodeJS, Golang, LUA语言探针

支持Envoy + Istio构建的Service Mesh



整个系统分为三部分：

- agent：采集 tracing（调用链数据）和 metric（指标）信息并上报
- OAP：收集 tracing 和 metric 信息通过 analysis core 模块将数据放入持久化容器中（ES，H2（内存数据库），mysql 等等），并进行二次统计和监控告警
- webapp：前后端分离，前端负责呈现，并将查询请求封装为graphQL提交给后端，后端通过 ribbon 做负载均衡转发给 OAP 集群，再将查询结果渲染展示

而整个 Skywalking（包括 agent 和 OAP，而 webapp 后端业务非常简单主要就是认证和请求转发）均通过微内核+插件式的模式进行编码，代码结构和扩展性均非常强，具体设计可以参考：[从Skywalking看如何设计一个微核+插件式扩展的高扩展框架]，Spring Cloud Gateway 的 GatewayFilterFactory 的扩展也是通过这种 plugin define 的方式来实现的。

Skywalking也提供了其他的一些特性：

- 配置重载：支持通过 **jvm** 参数覆写默认配置，支持动态配置管理
- 集群管理：这个主要体现在 OAP，通过集群部署分担数据上报的流量压力和二次计算的计算压力，同时集群也可以通过配置切换角色，分别面向数据采集（collector）和计算（aggregator，alarm），需要注意的是 agent 目前不支持多 collector 负载均衡，而是随机从集群中选择一个实例进行数据上报
- 支持 k8s 和 mesh
- 支持数据容器的扩展，例如官方主推是ES，通过扩展接口，也可以实现插件去支持其他的数据容器
- 支持数据上报 receiver 的扩展，例如目前主要是支持 gRPC 接受 agent 的上报，但是也可以实现插件支持其他类型的数据上报（官方默认实现了对 Zipkin，telemetry 和 envoy的支持）
- 支持客户端采样和服务端采样，不过服务端采样最有意义

- 官方制定了一个数据查询脚本规范：OAL（Observability Analysis Language），语法类似Linq，以简化数据查询扩展的工作量
- 支持监控预警，通过 OAL 获取数据指标和阈值进行对比来触发告警，支持 webhook 扩展告警方式，支持统计周期的自定义，以及告警静默防止重复告警

数据容器

由于 Skywalking 并没有自己定制的数据容器或者使用多种数据容器增加复杂度，而是主要使用ElasticSearch，所以数据容器的特性以及自己数据结构基本上就限制了业务的上限，以ES为例：

- ES查询功能异常强大，在数据筛选方面碾压其他所有容器，在数据筛选潜力巨大
- 支持 sharding 分片和 replicas 数据备份，在高可用/高性能/大数据支持都非常好
- 支持批量插入，高并发下的插入性能大大增强
- 数据密度低，源于ES会提前构建大量的索引来优化搜索查询，这是查询功能强大和性能好的代价，但是链路跟踪往往有非常多的上下文需要记录，所以Skywalking把这些上下文二进制化然后通过 Base64 编码放入 data_binary 字段并且将字段标记为 **not_analyzed** 来避免进行预处理建立查询索引

总体来说，Skywalking尽量使用ES在大数据和查询方面的优势，同时尽量减少ES 数据密度低的劣势带来的影响，从目前来看，ES在调用链跟踪方面是不二的数据容器，而在数据指标方面，ES也能中规中矩的完成业务，虽然和时序数据库相比要弱一些，但在PB级以下的数据支持也不会有太大问题。

数据结构

如果说数据容器决定了上限，那么数据结构则决定了实际到达的高度。Skywalking的数据结构主要为：

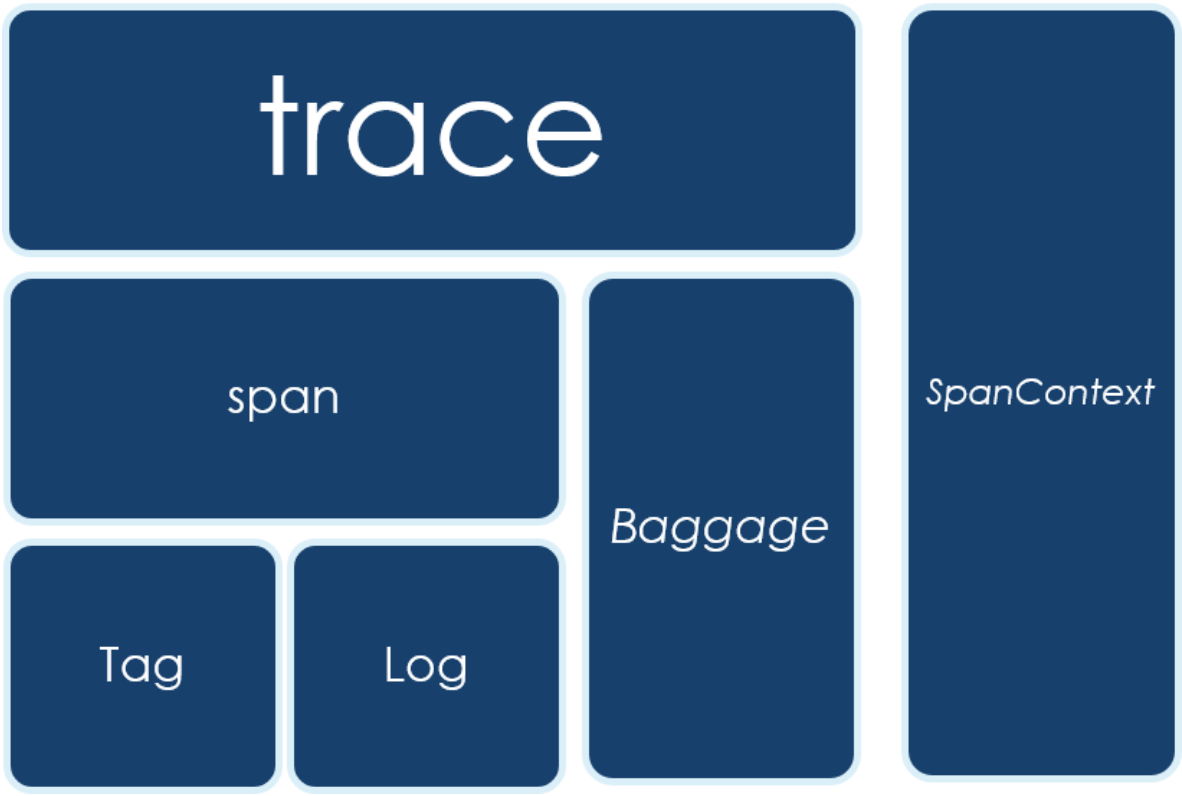
- 数据维度（ES索引为skywalking*inventory)
  - service：服务
  - instance：实例
  - endpoint：接口
  - network_adress：外部依赖
- 数据内容
  - 原始数据
    - 调用链跟踪数据（调用链的trace信息，ES索引为skywalking_segment，Skywalking主要的数据消耗都在这里）
    - 指标（主要是jvm或者envoy的运行时指标，例如ES索引skywalking_instance_jvm_cpu）
  - 二次统计指标
    - 指标（按维度/时间二次统计出来的例如pxx、sla等指标，例如ES索引skywalking_database_access_p75_month）
    - 数据库慢查询记录（数据库索引：skywalking_top_n_database_statement）
  - 关联关系（维度/指标之间的关联关系，ES索引为skywalking*relation_*)
  - 特别记录
    - 告警信息（ES索引为skywalking_alarm_record）
    - 并发控制（ES索引为skywalking_register_lock）

其中数量占比最大的就是调用链跟踪数据和各种指标，而这些数据均可以通过OAP设置过期时间，以降低历史数据的对磁盘占用和查询效率的影响。

调用链跟踪数据

作为Skywalking的核心数据，调用链跟踪数据（skywalking_segment）基本上奠定了整个系统的基础，而如果要详细的了解调用链跟踪的话，就不得不提到 OpenTracing。

openTracing基本上是目前开源调用链跟踪系统的一个事实标准，它制定了调用链跟踪的基本流程和基本的数据结构，同时也提供了各个语言的实现。如果用一张图来表现openTracing，则是如下：

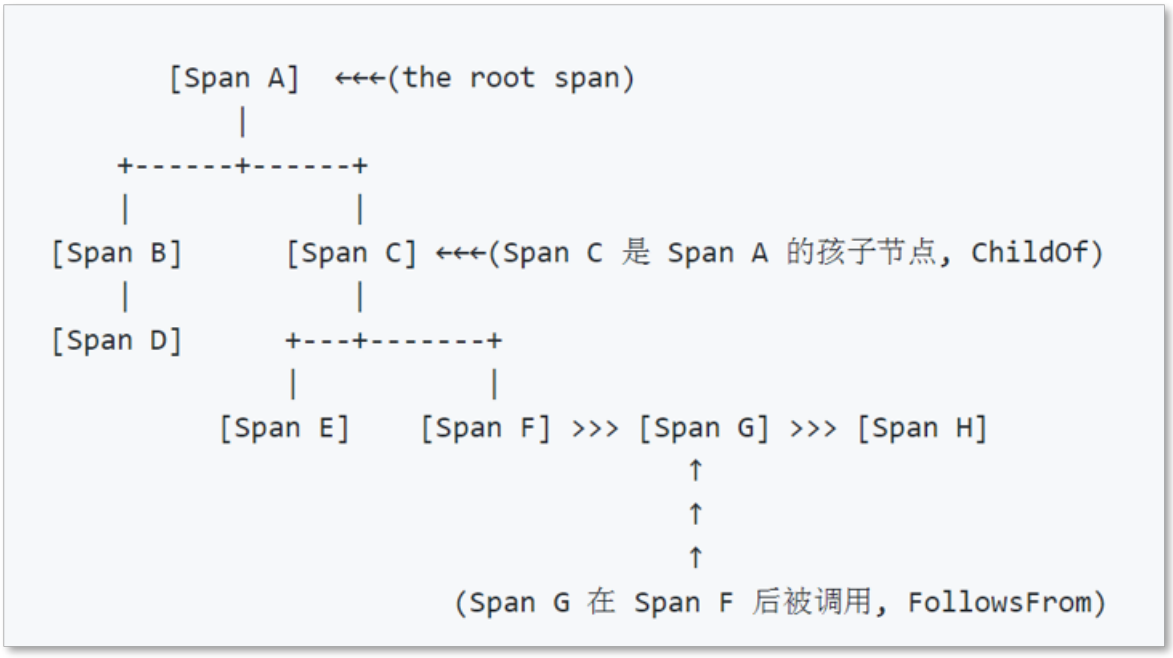


其中：

- SpanContext：一个类似于 MDC（Slfj）或者 ThreadLocal 的组件，负责整个调用链数据采集过程中的上下文保持和传递
- Trace：一次调用的完整记录
    - Span：一次调用中的某个节点/步骤，类似于一层堆栈信息，Trace是由多个Span组成，Span和Span之间也有父子或者并列的关系来标志这个节点/步骤在整个调用中的位置
        - Tag：节点/步骤中的关键信息
        - Log：节点/步骤中的详细记录，例如异常时的异常堆栈
    - Baggage：和 SpanContext 一样并不属于数据结构而是一种机制，主要用于跨 Span 或者跨实例的上下文传递，Baggage 的数据更多是用于运行时，而不会进行持久化

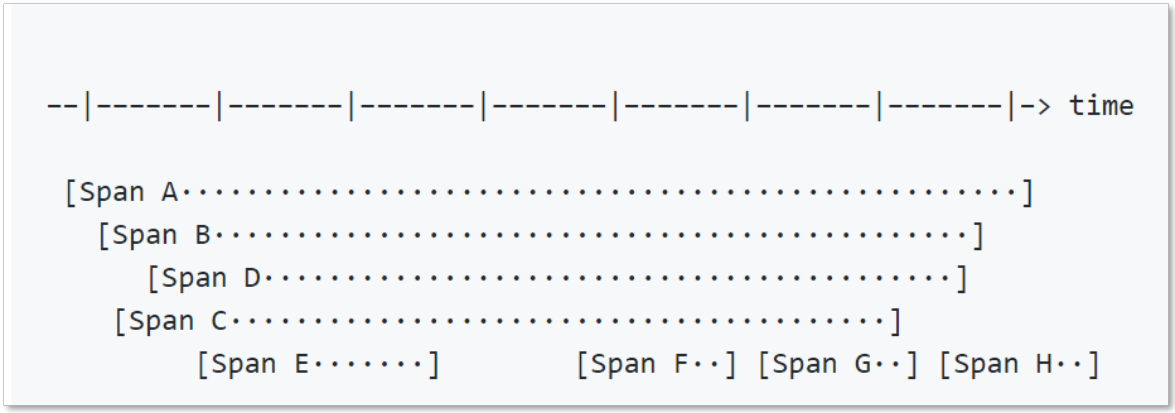
以一个Trace为例：

.



首先是外部请求调用A，然后A依次同步调用了B和C，而B被调用时会去同步调用D，C被调用的时候会依次同步调用E和F，F被调用的时候会通过异步调用G，G则会异步调用H，最终完成一次调用。

上图是通过Span之间的依赖关系来表现一个Trace，而在时间线上，则可以有如下的表达：



当然，如果是同步调用的话，父 Span 的时间占用是包括子Span的时间消耗的。

而落地到 Skywalking 中，我们以一条 skywalking_segment 的记录为例：


```
{
    "trace_id": "52.70.15530767312125341",
    "endpoint_name": "Mysql/JDBI/Connection/commit",
    "latency": 0,
    "end_time": 1553076731212,
    "endpoint_id": 96142,
    "service_instance_id": 52,
    "version": 2,
    "start_time": 1553076731212,
    "data_binary":
"CgwKCjRGnPvp5eikyxsSXhD//////////8BGMz62NSZLSDM+tjUmS0wju8FQChQAVgBYCF6DgoHZGI
udHlwZRIDc3FsehcKC2RiLmluc3RhbmNlEghyaXNrZGF0YXoOCgxkYi5zdGF0ZW1lbnQYAiA0",
    "service_id": 2,
    "time_bucket": 20190320181211,
    "is_error": 0,
    "segment_id": "52.70.15530767312125340"
}
```

其中：

- trace_id：本次调用的唯一id，通过snowflake模式生成
- endpoint_name：被调用的接口
- latency：耗时
- end_time：结束时间戳
- endpoint_id：被调用的接口的唯一id
- service_instance_id：被调用的实例的唯一id
- version：本数据结构的版本号
- start_time：开始时间戳
- data_binary：里面保存了本次调用的所有Span的数据，序列化并用Base64编码，不会进行分析和用于查询
- service_id：服务的唯一id
- time_bucket：调用所处的时段
- is_error：是否失败
- segment_id：数据本身的唯一id，类似于主键，通过snowflake模式生成

这里可以看到， 除了endPoint，并没有和业务有关联的字段，只能通过时间/服务/实例/接口/成功标志/耗时来进行非业务相关的查询，如果后续要增强业务相关的搜索查询的话，应该还需要增加一些用于保存动态内容（如 messageId，orderId 等业务关键字）的字段用于快速定位。

指标

指标数据相对于 Tracing 则要简单得多了，一般来说就是指标标志、时间戳、指标值，而 Skywalking 中的指标有两种：一种是采集的原始指标值，例如 jvm 的各种运行时指标（例如 cpu 消耗、内存结构、GC 信息等）；一种是各种二次统计指标（例如 tp 性能指标、SLA 等，当然也有为了便于查询的更高时间维度的指标，例如基于分钟、小时、天、周、月）

例如以下是索引 skywalking_endpoint_cpm_hour 中的一条记录，用于标志一个小时内某个接口的cpm指标：

```
{
    "total": 8900,
    "service_id": 5,
    "time_bucket": 2019031816,
    "service_instance_id": 5,
    "entity_id": "7",
    "value": 148
}
```

各个字段的释义如下：

- total：一分钟内的调用总量
- service_id：所属服务的唯一id
- time_bucket：统计的时段
- service_instance_id：所属实例的唯一id
- entity_id：接口（endpoint）的唯一id
- value：cpm的指标值（cpm=call per minute，即total/60）

3. 结果评估

通过上述对Skywalking的剖析，Skywalking确实是一个优秀的 APM + 调用链跟踪监控系统，能够覆盖大部分使用场景，让研发和运维能够更加实时/准实时的了解线上服务的运行情况。当然 Skywailking 也不是尽善尽美，例如下面就是个人觉得目前可见的不满足我们期望的：

- 数据准实时通过 gRPC 上报，本地缓存的瓶颈
    - 缓存队列的长度，过长占据内存，过短容易buffer满丢弃数据
    - 优雅停机同时又不丢失缓存
- 数据上报需要在起点上报，链路回传的时候需要携带SPAN及子SPAN的信息，当链路较长或者SPAN保存的信息较多时，会额外消耗一定的带宽
- skywalking 更多是一个APM系统而不是分布式调用链跟踪系统
    - 在整个链路的探针上均缺少输入输出的抓取
    - 在调用链的筛查上并没用进行增强，并且体现在数据结构的设计，例如TAG信息均保存在SPAN信息中，而SPAN信息均被BASE64编码作为数据保存，无法检索，最终trace的筛查只能通过时间/traceId/service/endPoint/state进行非业务相关的搜索
- skywalking缺少对三方接口依赖的指标，这个对于系统稳定往往非常重要

而作为一个初级的使用者，个人觉得我们可以使用有限的人力在以下方向进行扩展：

- 增加receiver：整合ELK，通过日志采集采集数据，降低异构系统的采集开发成本
- 优化数据结构，提供基于业务关键数据的查询接口
- 优化探针，采集更多的业务数据，争取代替传统的ELK日志简单查询，绝大部分异常诊断和定位均可以通过Skywalking即可完成
- 增加业务指标监控的模式，能够自定义业务指标（目前官方已经在实现 Metric Exporter ）