- 解决了什么问题
- 怎么使用/使用要求/适用场景
- 进行实验验证一部分功能边界 、实际使用
- 业界的成熟的实用案例/最佳实践

参考链接
https://developer.aliyun.com/article/514488
https://pjw.io/articles/2018/05/08/opentracing-explanations/

解决了什么问题?

分布式 tracing 很重要。（相较于 logging 和 metric 的局部视野，tracing 提供了一个服务调用的全局视野）

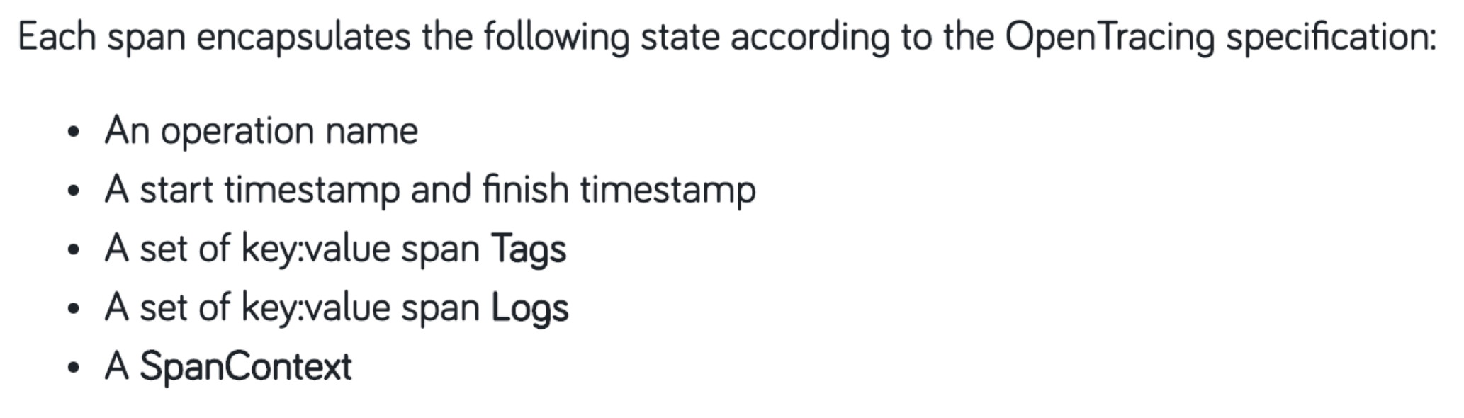
但tracing 是侵入代码的，各家都有独特的 tracing 工具， 但不同的工具的 埋点方式不同，切换需要大量修改代码
open tracing 统一了 tracing 的 api， 使用这套api可以在多个 tracing系统 中进行切换。
支持的tracer https://opentracing.io/docs/supported-tracers/

Supported tracers

- CNCF Jaeger
- LightStep
- Instana
- Apache SkyWalking
- inspectIT
- stagemonitor
- Datadog
- Wavefront by VMware
- Elastic APM

api 能力 / 适用场景

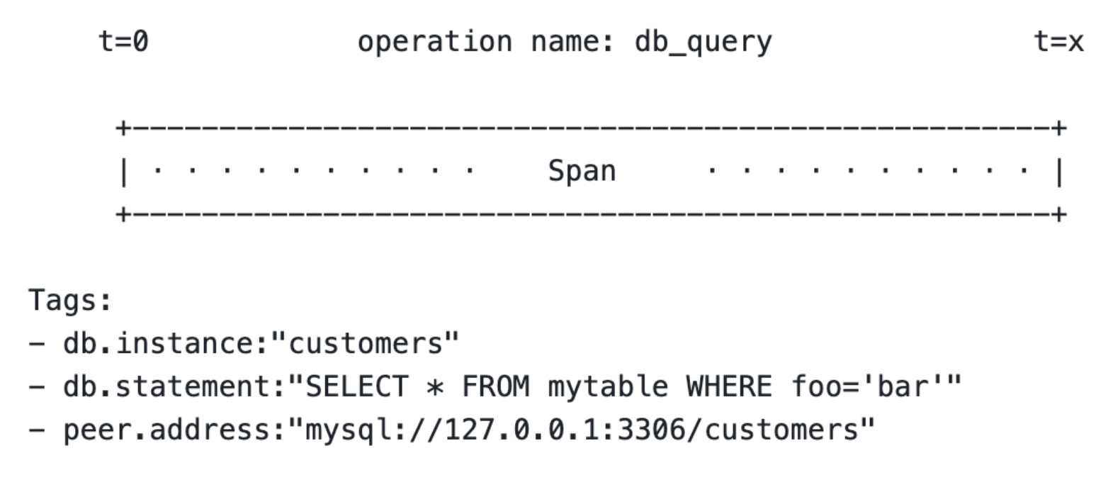
核心概念
Span



Each span encapsulates the following state according to the OpenTracing specification:

- An operation name
- A start timestamp and finish timestamp
- A set of key:value span **Tags**
- A set of key:value span **Logs**
- A **SpanContext**

SpanContext

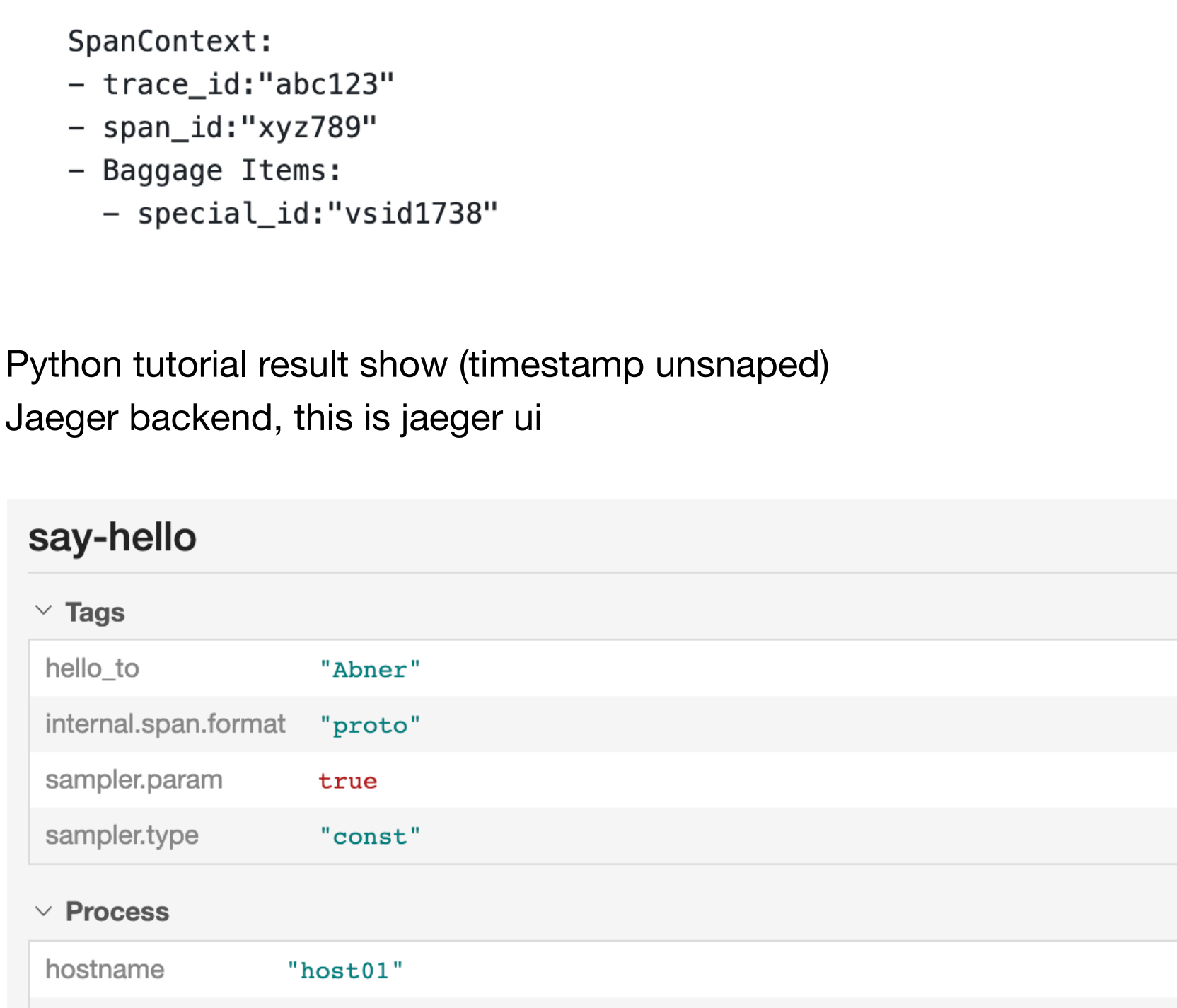


SpanContext

The **SpanContext** carries data across process boundaries. Specifically, it has two major components:

- An implementation-dependent state to refer to the distinct span within a trace
  - i.e., the implementing Tracer's definition of spanID and traceID
- Any Baggage Items
  - These are key:value pairs that cross process-boundaries.
  - These may be useful to have some data available for access throughout the trace.

Example Span



Example Span:

```
    t=0            operation name: db_query               t=x

     +-----------------------------------------------------+
     | · · · · · · · · · ·    Span     · · · · · · · · · · |
     +-----------------------------------------------------+

Tags:
- db.instance:"customers"
- db.statement:"SELECT * FROM mytable WHERE foo='bar'"
- peer.address:"mysql://127.0.0.1:3306/customers"

Logs:
- message:"Can't connect to mysql server on '127.0.0.1'(10061)"

SpanContext:
- trace_id:"abc123"
- span_id:"xyz789"
- Baggage Items:
  - special_id:"vsid1738"
```

Python tutorial result show (timestamp unsnaped)
Jaeger backend, this is jaeger ui

say-hello

Tags

hello_to	"Abner"
internal.span.format	"proto"
sampler.param	true
sampler.type	"const"

Process

hostname	"host01"
ip	"172.17.0.72"
jaeger.version	"Python-4.3.0"

Logs (2)

0.05ms

event	"string-format"
value	"Abner"

0.09ms: event = println

Log timestamps are relative to the start time of the full trace.

附：
tracing / logging / metric 的关系 http://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html?spm=a2c6h.12873639.0.0.61b453f6XfNkoW
opentracing / ES / prometheus

tracing 提供了一次 request 过程中所有服务调用的全局视野
logging 记录一些零散的、重要的日志信息
metrics 记录了一些可以聚合的数据信息，如服务调用时长，一段时间内的调用次数等

实验部分

milvus 现有 tracing

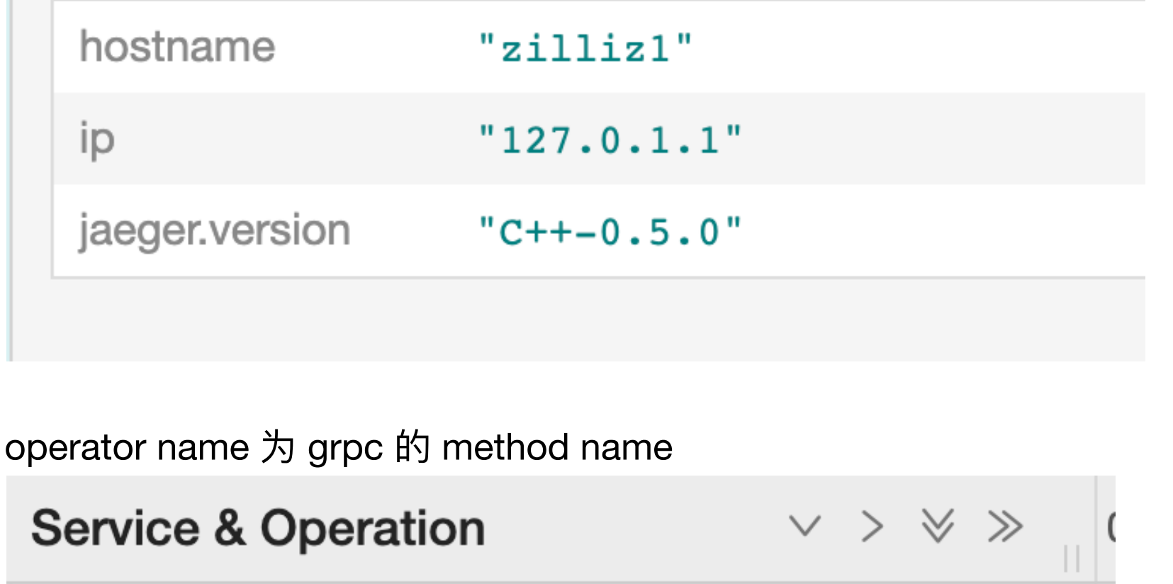
Milvus 现有tracing 是定义在 grpc handler 中，通过 interceptor 注入一个 span 在request 的 context 中，在 request 处理过程中添加一些信息，包括但不限于 子步骤、combine query（child）、 Xsearch_task（Follower）。
没有添加其他信息，只利用了 open tracing 的时序关系、结构关系（child 关系、follower 关系）
如利用 tag 添加 运行节点信息，进程信息
利用log 添加一些必要的log（？？？）

启动并测试
需要在 server_config.yaml 中添加

```
tracing_config:
  json_config_path:   path_to_json
```

将 config yaml 修改正确后 （一些其他的字段或者其他）

tags 无有效信息



Tags

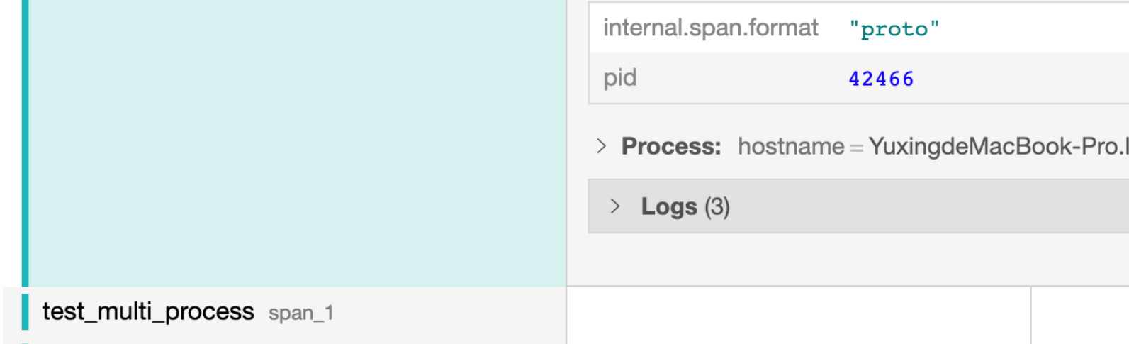
internal.span.format	"proto"
sampler.param	true
sampler.type	"const"

process 中 包含hostname 和 ip。 具体找的 ip 规则不知。

Process

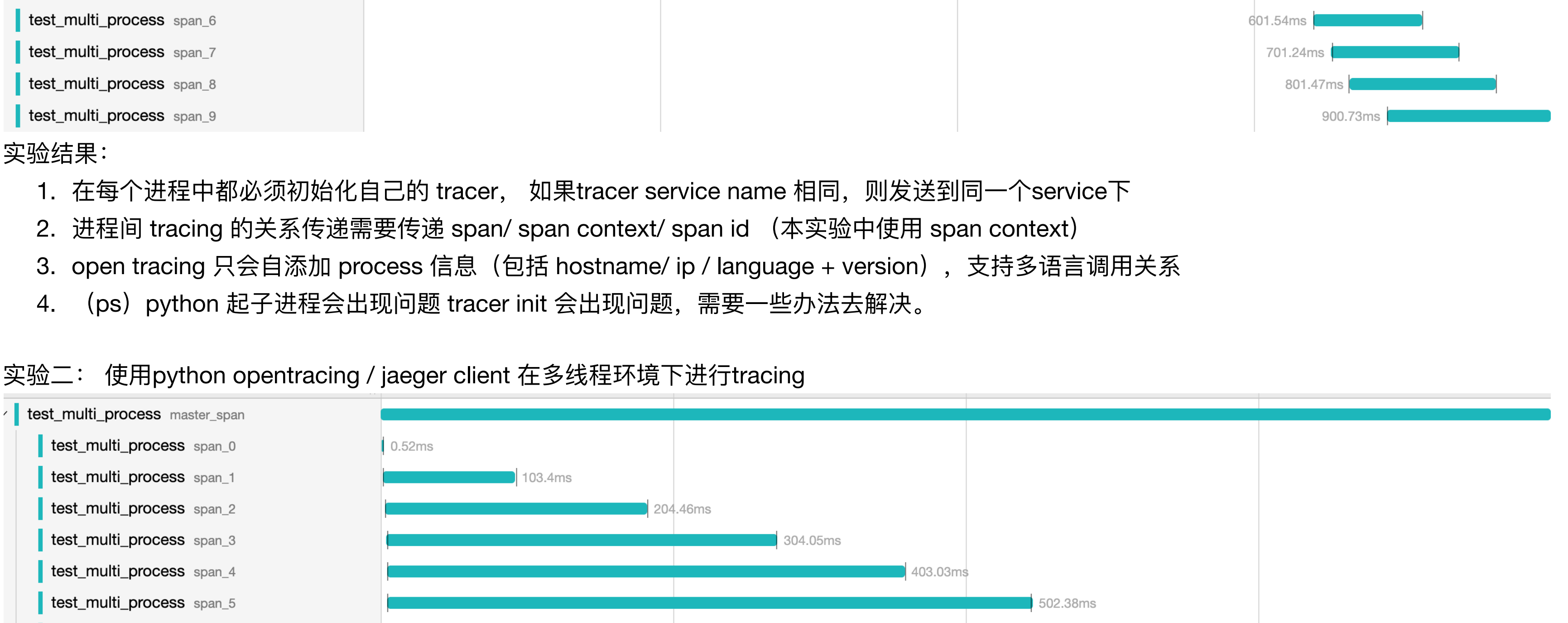
hostname	"zilliz1"
ip	"127.0.1.1"
jaeger.version	"C++-0.5.0"

operator name 为 grpc 的 method name



milvus_server /milvus.grpc.MilvusService/DropCollec…

实验一： 使用python opentracing / jaeger client 在多进程环境下进行tracing



test_multi_process span_0 0.47ms

span_0 Service: test_multi_process | Duration: 0.47ms | Start Time: 5.01s

Tags

internal.span.format	"proto"
pid	42466

Process: hostname = YuxingdeMacBook-Pro.local | ip = 192.168.3.5 | jaeger.version = Python-4.3.0

Logs (3)

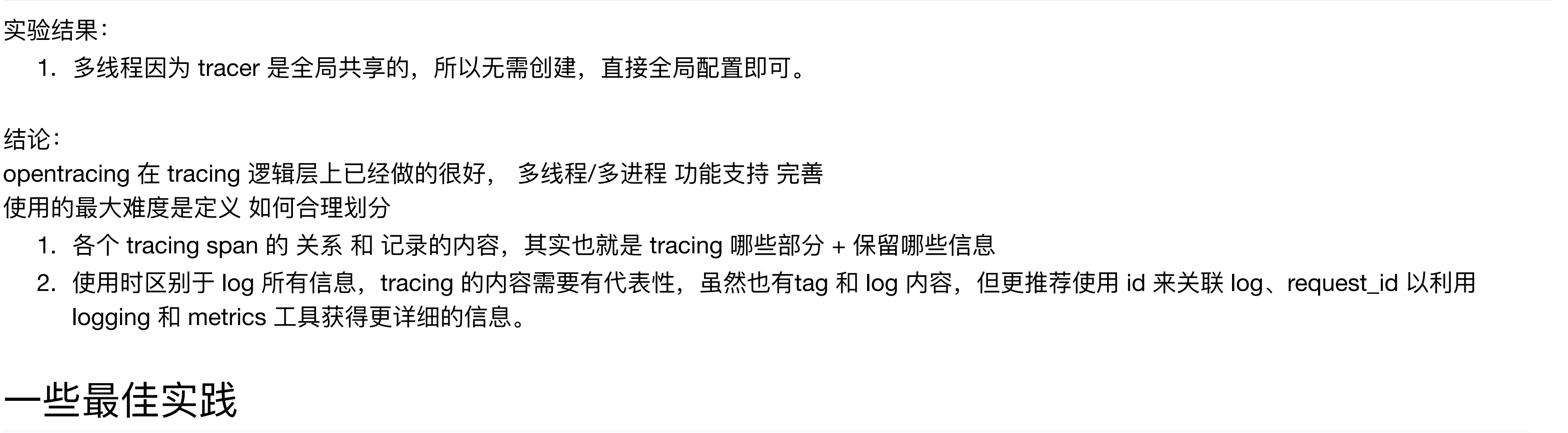
SpanID: 90fa18618bb74944

test_multi_process span_1 101.55ms
test_multi_process span_2 201.19ms
test_multi_process span_3 301.08ms
test_multi_process span_4 400.82ms
test_multi_process span_5 501.81ms
test_multi_process span_6 601.54ms
test_multi_process span_7 701.24ms
test_multi_process span_8 801.47ms
test_multi_process span_9 900.73ms

实验结果：

1. 在每个进程中都必须初始化自己的 tracer， 如果tracer service name 相同，则发送到同一个service下
2. 进程间 tracing 的关系传递需要传递 span/ span context/ span id （本实验中使用 span context）
3. open tracing 只会自添加 process 信息（包括 hostname/ ip / language + version），支持多语言调用关系
4. （ps）python 起子进程会出现问题 tracer init 会出现问题，需要一些办法去解决。

实验二： 使用python opentracing / jaeger client 在多线程环境下进行tracing



test_multi_process master_span
test_multi_process span_0 0.52ms
test_multi_process span_1 103.4ms
test_multi_process span_2 204.46ms
test_multi_process span_3 304.05ms
test_multi_process span_4 403.03ms
test_multi_process span_5 502.38ms
test_multi_process span_6 602.93ms
test_multi_process span_7 702.96ms
test_multi_process span_8 804.44ms
test_multi_process span_9

span_9 Service: test_multi_process | Duration: 904.9ms | Start Time: 6.01ms

Tags: internal.span.format = proto | pid = 43319

Process: hostname = YuxingdeMacBook-Pro.local | ip = 192.168.3.5 | jaeger.version = Python-4.3.0

Logs (3)

SpanID: cfae415ef08c9463

实验结果：

1. 多线程因为 tracer 是全局共享的，所以无需创建，直接全局配置即可。

结论：
opentracing 在 tracing 逻辑层上已经做的很好， 多线程/多进程 功能支持 完善
使用的最大难度是定义 如何合理划分

1. 各个 tracing span 的 关系 和 记录的内容，其实也就是 tracing 哪些部分 + 保留哪些信息
2. 使用时区别于 log 所有信息，tracing 的内容需要有代表性，虽然也有tag 和 log 内容，但更推荐使用 id 来关联 log、request_id 以利用 logging 和 metrics 工具获得更详细的信息。

一些最佳实践

在 Trace 的起始处，将 Trace ID 设置为 Request ID，这么一来就打通了日志系统和分布式追踪系统，可以使用同一个 ID 查询请求的事件流和日志流，从此开启了上帝视角。

1. grpc server 类中可以使用 span id 作为 request id 串联整个系统的调用路径
2. client 部分也可以添加 tracing 以使得各个调用路径完整展现
3. tracing 涉及到保存信息 + 发送到 agent 或 tracing server，一定会减弱性能，但可以通过设置 采样率/运行时关闭等设置避免（具体性能消耗没有相关的实验环境，暂时没有进行）