《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

第34讲：实现线程级别监控 轻松搞定 Thread Dump

背景

SkyWalking 提供的 Agent 可以收集服务的 Metrics、Trace、Log 等维度的数据

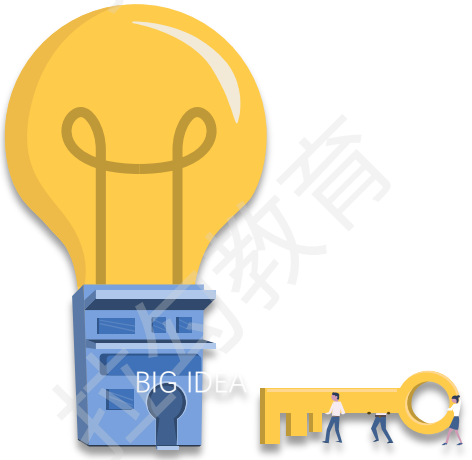
然后发送到后端的 OAP 进行分析并进行持久化存储

可以使用 SkyWalking Rocketbot UI（或是直接使用 GraphQL） 从不同的维度查询上述数据

评估系统的各项性能和某些具体行为

背景

- 可以通过 ServiceRespTimeMetrics、ServiceP99Metrics、ServiceCpmMetrics 等 Metrics 了解一个服务的整体吞吐量
- 可以通过 Trace 信息了解某个具体请求经过的核心组件和服务，以及在这些组件和服务上的耗时情况
- 可以通过 Trace 上携带的 Log 信息了解相应的异常信息
- 可以根据 Trace 信息分析得到 Relation 信息，画出整个服务架构的拓扑图了解各个服务之间的调用关系以及拓扑图每条调用边上的响应时间、SLA 等信息



背景

/hello/xxx

49.42.15800230364650001

起始点 2020-01-26 15:17:16 持续时间 4485 ms 跨度 5

列表 树结构 表格

demo-webapp demo-provider

背景

实际的业务逻辑比较复杂，请求处理耗时高的原因也可能千奇百怪

例如（可能但不限于）：

- 多个线程并发竞争同一把锁
- 读写文件，线程等待 I/O 操作
- 代码逻辑本身的性能有问题，时间复杂度太高

背景

在实际的微服务场景中进行 Thread Dump 时，你可能会遇到几个问题：

- 如果多个服务都有耗时高的情况，需要我们去多个服务的机器上进行 Thread Dump，比较麻烦
  而且也很难确定不同服务的 Thread Dump 信息是否存在关联
- 请求一般会经过多个服务端处理，每个服务又是单独的一个集群
- 如果要求某些服务的响应时延非常低的情况下，虽然服务的延迟高了，但是相对人来说的时间是非常短的
  而手动 Thread Dump 的速度和次数都是有限的，可能错过问题所在的逻辑，导致问题定位错误

Thread Dump 需求

一般场景中，用户会通过一个外网的入口请求接入层（例如机房的 Nginx 集群）

然后接入层会进行负载均衡，将请求发送到后端的 API 服务集群进行处理（例如 Tomcat 集群）

API 服务会根据业务需求调用后端的 RPC 服务（例如 Dubbo、gRPC 等）

在 RPC 服务中会调用 Service 层、DAO 层等完成存储的读写或是再次调用其他 RPC 服务

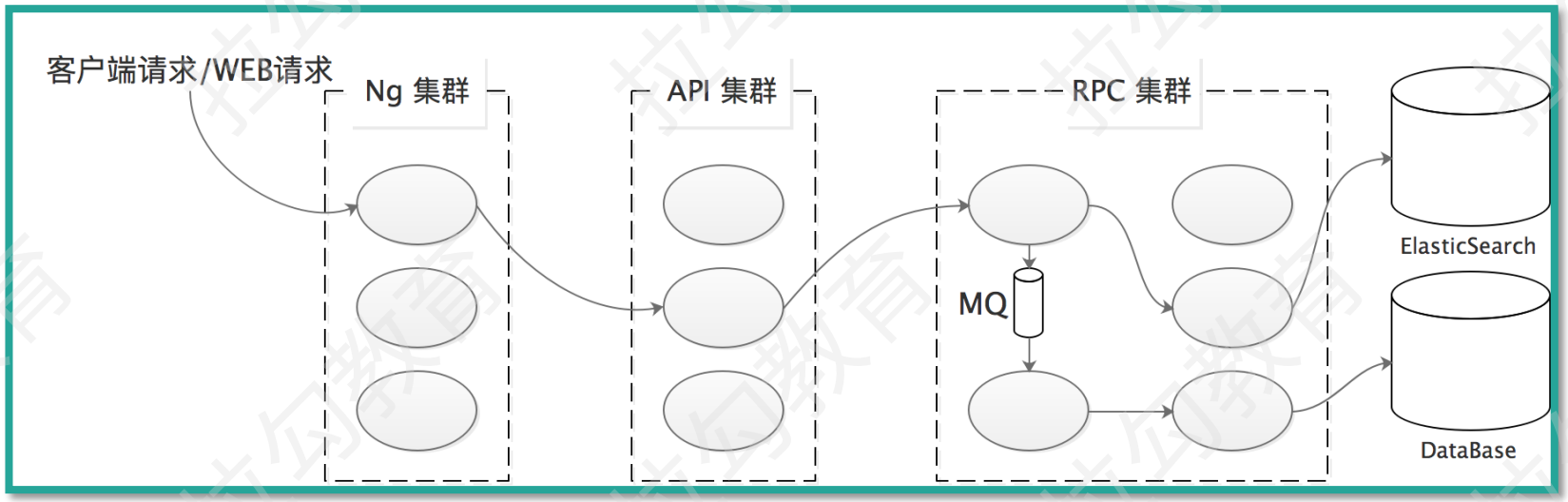
Thread Dump 需求

一般场景中，用户会通过一个外网的入口请求接入层（例如机房的 Nginx 集群）

然后接入层会进行负载均衡，将请求发送到后端的 API 服务集群进行处理（例如 Tomcat 集群）

API 服务会根据业务需求调用后端的 RPC 服务（例如 Dubbo、gRPC 等）

在 RPC 服务中会调用 Service 层、DAO 层等完成存储的读写或是再次调用其他 RPC 服务



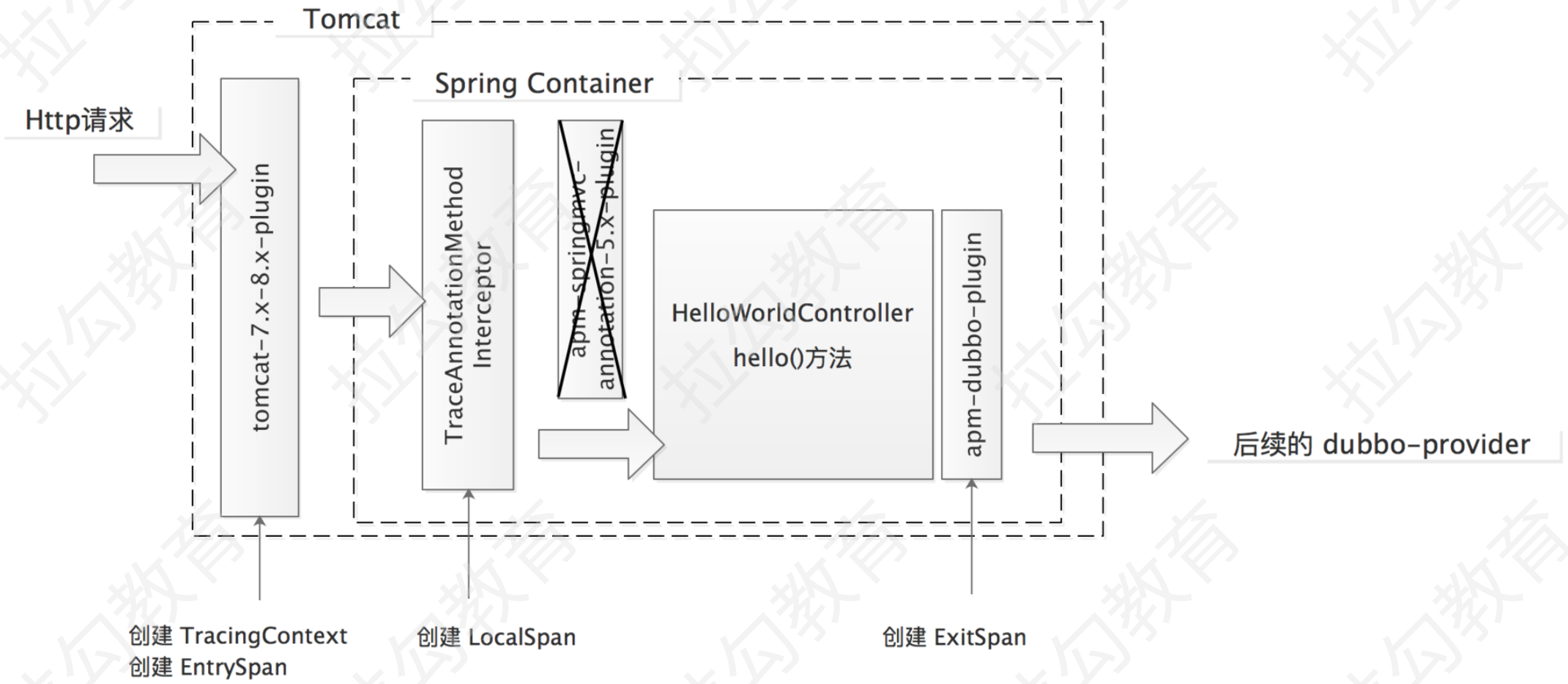
Thread Dump 需求


```
ThreadMXBean bean = ManagementFactory.getThreadMXBean();
ThreadInfo[] threadInfos = bean.dumpAllThreads(true, true);
for (ThreadInfo threadInfo : threadInfos) {
    System.out.println(threadInfo);
}

//部分输出如所示，我们可以看到每个线程的状态信息以及具体的调用栈：

"Reference Handler" Id=2 WAITING on
java.lang.ref.Reference$Lock@1517365b
at java.lang.Object.wait(Native Method)
-  waiting on java.lang.ref.Reference$Lock@1517365b
at java.lang.Object.wait(Object.java:502)
at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)
```

Thread Dump 功能实现



Thread Dump 功能实现


```
// ENABLE_DUMP_FLAG 标记在Http Header 和 RuntimeContext中使用相应的Key
//可以将"ENABLE_DUMP_FLAG"字符串抽到 Constants作为常量，后续可以重复使用
ContextManager.getRuntimeContext().put("ENABLE_DUMP_FLAG",
  request.getHeader("ENABLE_DUMP_FLAG")
);
//对 ContextCarrier 的处理后面会介绍
```

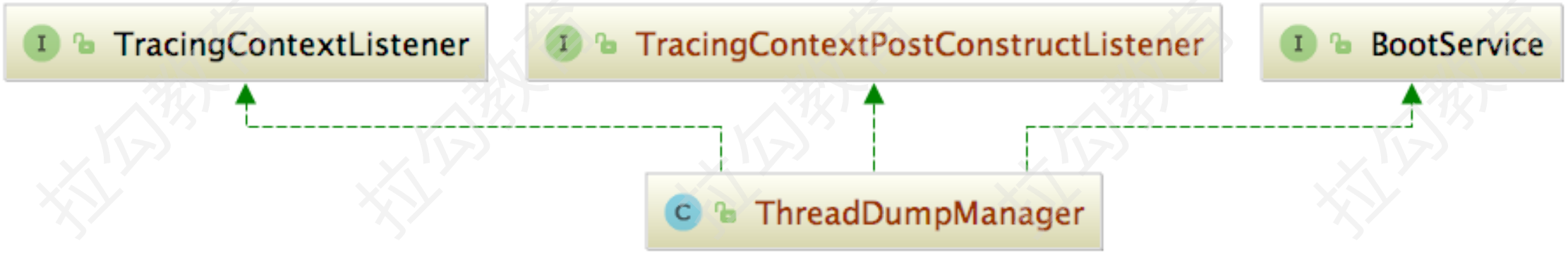
Thread Dump 功能实现


```
public interface TracingContextPostConstructListener {
  void postConstruct(TracingContext tracingContext);
}
```

Thread Dump 功能实现


```
private void postConstruct() {
    TracingContext.ListenerManager.notifyPostConstruct(this);
}
```

Thread Dump 功能实现



Thread Dump 功能实现

首先在 onComplete() 方法（对 BootService 接口的实现）中

会启动一个单独的线程执行一个定时任务，该定时任务主要做两件事：

- 定时通过 ThreadMXBean 获取线程的 dump 信息
- 查找到处理 ENABLE_DUMP_FLAG 标记请求的线程，并将该线程的 dump 信息与 Trace 关联起来

Thread Dump 功能实现


```
//其中 Key处理标记请求的线程 ID，Value是线程的 dump 信息，该线程在处理标记请求
//的过程中，可能会被 dump 多次，所以Value 是 List<ThreadDump>集合
private Map<Long, List<ThreadDump>> dumpStore =
    Maps.newConcurrentMap();

private ScheduledExecutorService scheduledExecutorService;

@Override
public void onComplete() throws Throwable {
    //创建并启动后台线程
    scheduledExecutorService = Executors
            .newSingleThreadScheduledExecutor();
    scheduledExecutorService.scheduleAtFixedRate(
            new RunnableWithExceptionProtection(this::doThreadDump,
                    t -> logger.error("thread dump error.", t)),
            dumpPeriod, dumpPeriod, TimeUnit.MILLISECONDS);
    //将 ThreadDumpManager作为 TracingContextListener接口实现进行注册
```

Thread Dump 功能实现


```
@Override
public void onComplete() throws Throwable {
  //创建并启动后台线程
  scheduledExecutorService = Executors
      .newSingleThreadScheduledExecutor();
  scheduledExecutorService.scheduleAtFixedRate(
      new RunnableWithExceptionProtection(this::doThreadDump,
          t -> logger.error("thread dump error.", t)),
      dumpPeriod, dumpPeriod, TimeUnit.MILLISECONDS);
  //将 ThreadDumpManager作为 TracingContextListener接口实现进行注册
  TracingContext.ListenerManager.addFirst(
      (TracingContextListener) this);
  //将 ThreadDumpManager作为 TracingContextPostConstructListener
  //接口实现进行注册
  TracingContext.ListenerManager.addFirst(
      (TracingContextPostConstructListener) this);
}
```

Thread Dump 功能实现


```
public void doThreadDump() {
    long dumpTimestamp = System.currentTimeMillis();
    ThreadMXBean bean = ManagementFactory.getThreadMXBean();
    //获取全部线程的 dump信息
    ThreadInfo[] threadInfos = bean.dumpAllThreads(true, true);
    for (ThreadInfo threadInfo : threadInfos){
        long threadId = threadInfo.getThreadId();
        //根据监控的线程ID，将相应的 dump信息记录到 dumpStore集合中
        List<ThreadDump> threadDumps = this.dumpStore.get(threadId);
        if (threadDumps != null) {
            //创建 ThreadDump来记录线程 dump信息
            ThreadDump threadDump = ThreadDump.newBuilder()
                    .setDumpTime(dumpTimestamp)
                    .setThreadInfo(threadInfo.toString()).build();
            threadDumps.add(threadDump);
        }
    }
}
```

Thread Dump 功能实现


```
message SegmentObject {
    UniqueId traceSegmentId = 1;
    repeated SpanObjectV2 spans = 2;
    ..... // 省略3~5的字段
    repeated ThreadDump threadDumps = 6;
}

message ThreadDump {
    int64 dumpTime = 1;
    string threadInfo = 2;
}
```

Thread Dump 功能实现


```
@Override
public void postConstruct(TracingContext tracingContext) {
  RuntimeContext runtimeContext= ContextManager.getRuntimeContext();
  Object enableDumpFlag =
      runtimeContext.get(Constants.ENABLE_DUMP_FLAG);
  if (Constants.ENABLE_DUMP_VAULES.equals(
        enableDumpFlag.toString().toLowerCase())) {
    //将当前线程的ID添加到 dumpStore集合中
    dumpStore.put(Thread.currentThread().getId(),
      Lists.newLinkedList());
    //在TracingContext中也添加了显影的
    tracingContext.setEnableDumpFlag(Constants.ENABLE_DUMP_VAULES);
  }
}
```

Thread Dump 功能实现


```
public void afterFinished(TraceSegment traceSegment) {
    long threadId = Thread.currentThread().getId();
    List<ThreadDump> threadDumps = dumpStore.get(threadId);
    traceSegment.setThreadDumps(threadDumps);
    removeThread(threadId);
}
```

跨进程/跨线程传播

首先需要修改一下 ContextCarrier 序列化之后的字符串结构

SkyWalking 原始的 ContextCarrier 持久化后的字符串包括下面 9 个部分

且相互之间通过字符串“-”连接起来：

1. 固定字符串“1”
2. TraceId
3. TraceSegmentId
4. SpanId
5. ParentServiceInstanceId
6. EntryServiceInstanceId
7. PeerHost
8. EntryEndpointName
9. ParentEndpointName

跨进程/跨线程传播


```
String serialize(HeaderVersion version) {
    return StringUtil.join('-', "1",
        Base64.encode(this.getPrimaryDistributedTraceId().encode()),
        Base64.encode(this.getTraceSegmentId().encode()),
        this.getSpanId() + "",
        this.getParentServiceInstanceId() + "",
        this.getEntryServiceInstanceId() + "",
        Base64.encode(this.getPeerHost()),
        Base64.encode(this.getEntryEndpointName()),
        Base64.encode(this.getParentEndpointName()),
        this.enableDumpFlag); // 新增 enableDumpFlag 部分
}
```

跨进程/跨线程传播


```
ContextCarrier deserialize(String text, HeaderVersion version) {
    String[] parts = text.split("\\-", 10);
    if (parts.length == 9 || parts.length == 10) {
        // parts[0] is sample flag, always trace if header exists.
        this.primaryDistributedTraceId =
            new PropagatedTraceId(Base64.decode2UTFString(parts[1]));
        this.traceSegmentId =
            new ID(Base64.decode2UTFString(parts[2]));
        this.spanId = Integer.parseInt(parts[3]);
        this.parentServiceInstanceId = Integer.parseInt(parts[4]);
        this.entryServiceInstanceId = Integer.parseInt(parts[5]);
        this.peerHost = Base64.decode2UTFString(parts[6]);
        this.entryEndpointName = Base64.decode2UTFString(parts[7]);
        this.parentEndpointName = Base64.decode2UTFString(parts[8]);
        if (parts.length == 10) {
            this.enableDumpFlag = parts[9];
        }
    }
    return this;
}
```

跨进程/跨线程传播


```
public void inject(ContextCarrier carrier) {
    ... ...//省略前面的原始代码
    Object enableDumpFlag = ContextManager.getRuntimeContext()
        .get(Constants.ENABLE_DUMP_FLAG);
    carrier.setEnableDumpFlag(enableDumpFlag == null ? "" :
      enableDumpFlag.toString());
}
```

跨进程/跨线程传播


```
public void beforeMethod(...){
  if (isConsumer) {
    ... ... //省略Consumer创建 ExitSpan以及ContextCarrier的逻辑
  } else {
    ... ... //从 RpcContext中获取 ContextCarrier字符串并反序列化(略)
    //将ENABLE_DUMP_FLAG标记记录到 RuntimeContext中
    ContextManager.getRuntimeContext().put(Constants
      .ENABLE_DUMP_FLAG,contextCarrier.getEnableDumpFlag());
    //创建TracingContext，其中会触发
    // TracingContextPostConstructListener，从而记录需要dump的线程
    span = ContextManager.createEntrySpan(generateOperationName(
      requestURL, invocation), contextCarrier);
  }
  //省略后续设置Tag、Component以及SpanLayer的相关代码
}
```

OAP 改造

SegmentParseV2 会解析收到的 UpstreamSegment 得到相应的 **TraceSegment**

然后交给所有 RecordStreamProcessor 处理

如果存储选择 ElasticSearch，则 TraceSegment 的全部数据最终会按照序列化的格式存储到 segment-yyyyMMdd 索引中的 data_binary 字段中，当然也包括前面新增的 Thread Dump 信息

OAP 改造


```
type ThreadDump{
    dumpTimestamp: Long!
    threadInfo: String!
}

type Trace{
    spans: [Span!]!
    threadDumps: [ThreadDump!]! //在 Trace 中添加 threadDumps集合

}
```

OAP 改造


```
@Getter
@Setter
public class ThreadDump {
    private long dumpTimestamp;
    private String threadInfo;
}
```

OAP 改造


```
@Getter
public class Trace {
    private final List<Span> spans;
    private final List<ThreadDump> threadDumps;
}
```

OAP 改造


```
public Trace queryTrace(final String traceId) throws IOException {
    Trace trace = new Trace();
    //根据traceId查询所有关联的 SegmentObject
    List<SegmentRecord> segmentRecords =
        getTraceQueryDAO().queryByTraceId(traceId);
    for (SegmentRecord segment : segmentRecords) {
        //反序列化 SegmentObject
        SegmentObject segmentObject =
            SegmentObject.parseFrom(segment.getDataBinary());
        //解析 SegmentObject中的 Span，填充到 Trace中
        trace.getSpans().addAll(buildSpanV2List(traceId,
            segment.getSegmentId(), segment.getServiceId(),
            segmentObject.getSpansList()));
        //填充 ThreadDump集合
        trace.getThreadDumps().addAll(
            buildThreadDumpList(segmentObject.getThreadDumpsList()));
```

OAP 改造


```
        //反序列化 SegmentObject
        SegmentObject segmentObject =
            SegmentObject.parseFrom(segment.getDataBinary());
        //解析 SegmentObject中的 Span，填充到 Trace中
        trace.getSpans().addAll(buildSpanV2List(traceId,
            segment.getSegmentId(), segment.getServiceId(),
              segmentObject.getSpansList()));
        //填充 ThreadDump集合
        trace.getThreadDumps().addAll(
         buildThreadDumpList(segmentObject.getThreadDumpsList()));

    }
    ... ...//省略整理Trace中Span的顺序等操作，
        //这些逻辑在前文分析query-graphql-plugin插件时已经详细分析过
    return trace;
}
```

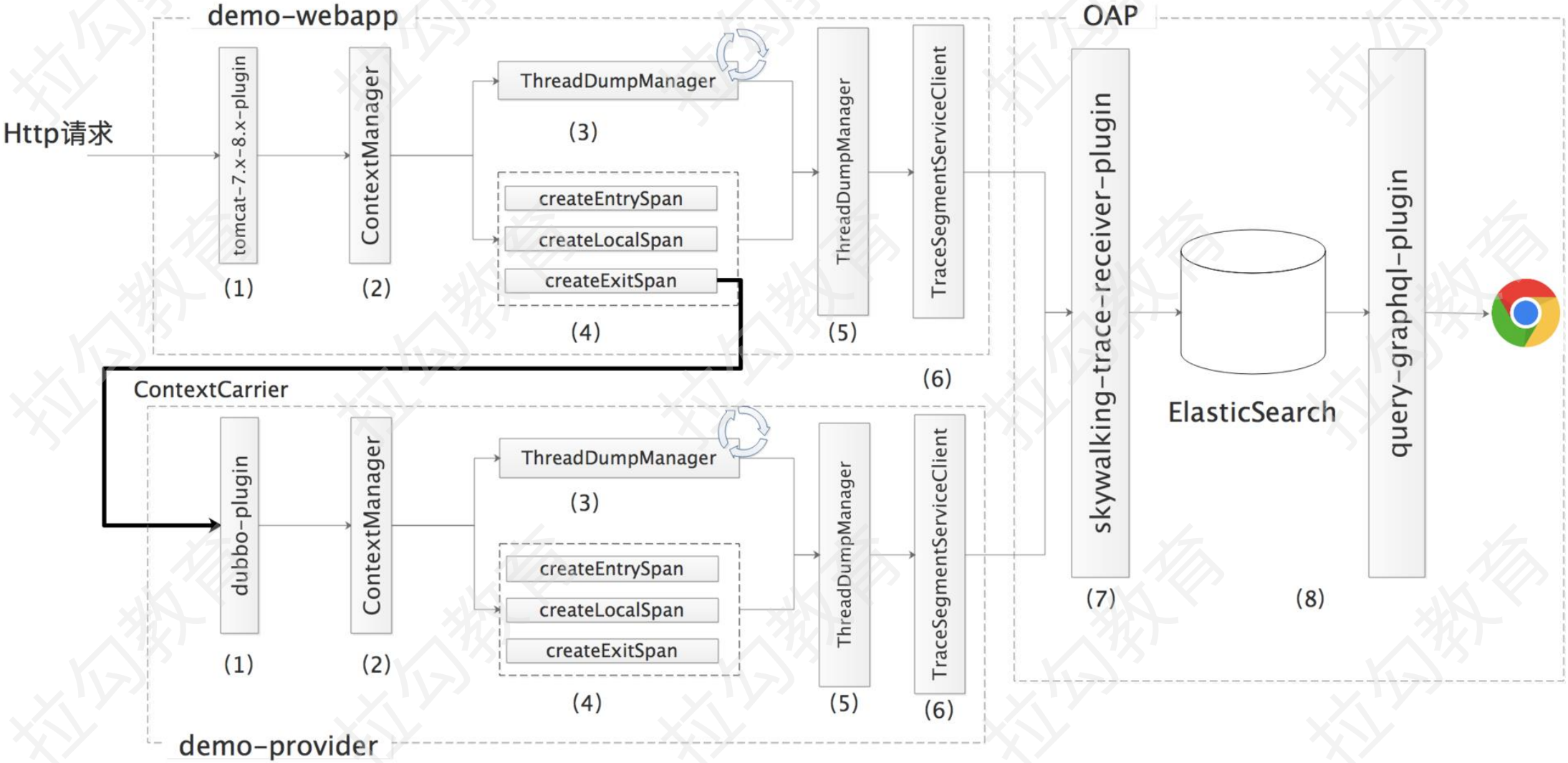
OAP 改造


```
mvn clean
mvn package -Dcheckstyle.skip -DskipTests
```

OAP 改造


```
{
 "data": {
  "trace": {
   "spans": [
    ... ... //省略该 Trace中的Span信息
   ],
   "threadDumps": [ //该Trace携带的ThreadDump信息
    {
     "dumpTimestamp": 1580029989057,
    "threadInfo": "\"DubboServerHandler-172.17.32.91:20880-thread-36\" Id=106
TIMED_WAITING\n\tat java.lang.Thread.sleep(Native Method)\n\tat
com.xxx.service.DefaultHelloService.say$original$MUzxmS45(DefaultHelloService.java:15)\n\t ... ..."
    }
    //省略其他 ThreadDump信息
   ]
  }
 }
}
```

总结



Next：彩蛋《回顾 SkyWalking 架构并展望未来》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U