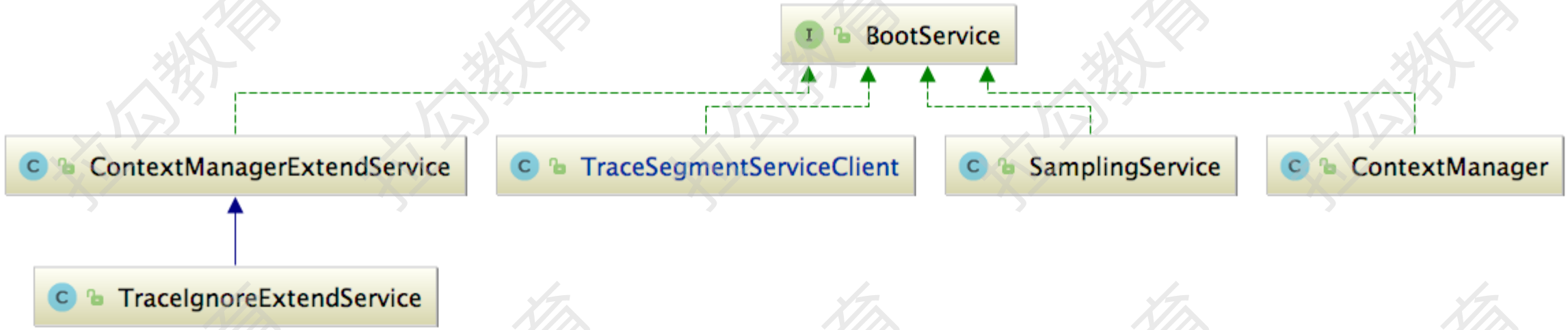
《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

— 拉勾教育出品 —

第14讲：收集、发送 Trace 核心原理 Agent 与 OAP 的大动脉

Trace 相关的 BootService 接口实现类



ContextManager

ContextManager

主要职责就是管理 TracingContext

通过 ThreadLocal 将 TracingContext 对象与当前线程进行绑定

实现 TraceSegment、TracingContext 和 线程三方之间的关联

ContextManager

- **CONTEXT（ThreadLocal<AbstractTracerContext> 类型）**

  通过该字段可以将一个 TracingContext 对象与一个线程进行关联

- **RUNTIME_CONTEXT（ThreadLocal<RuntimeContext> 类型）**

  RuntimeContext 底层封装了一个 ConcurrentHashMap 集合

  可以为当前 TracingContext 记录一些附加信息

- **EXTEND_SERVICE（ContextManagerExtendService 类型）**

  ContextManagerExtendService 也实现了 BootService 接口

  主要负责创建 TracingContext 对象

ContextManager


```
public static AbstractSpan createEntrySpan(String operationName,
        ContextCarrier carrier) {
    SamplingService samplingService = ServiceManager.INSTANCE
            .findService(SamplingService.class);  //  采样相关
    AbstractSpan span;
    AbstractTracerContext context;
    // 检测 ContextCarrier 是否合法，其实就是检查它的核心字段是否已填充好
    if (carrier != null && carrier.isValid()) {
        samplingService.forceSampled();
        // 获取当前线程绑定的 TracingContext
        context = getOrCreate(operationName, true);
        // 委托给当前线程绑定的 TracingContext 来创建 EntrySpan
        span = context.createEntrySpan(operationName);
```

ContextManager


```
            samplingService.forceSampled();
            // 获取当前线程绑定的 TracingContext
            context = getOrCreate(operationName, true);
            // 委托给当前线程绑定的 TracingContext 来创建 EntrySpan
            span = context.createEntrySpan(operationName);
            // 从 ContextCarrier 提取上游服务传播过来的Trace信息
            context.extract(carrier);
        } else { // 没有上游服务的场景
            context = getOrCreate(operationName, false);
            span = context.createEntrySpan(operationName);
        }
        return span;
    }
```

ContextManager

- **getOrCreate() 方法**

  会从 CONTEXT 字段中获取当前线程绑定的 TracingContext 对象

  如果当前线程没有关联 TracingContext 上下文，则会通过 ContextManagerExtendService 新建并绑定

- **stopSpan() 方法**

  在关闭 Span 的同时，会检查当前 TraceSegment 是否结束

  TraceSegment 结束时会将存储在 CONTEXT 中的 TracingContext 对象以及 RUNTIME_CONTEXT 中的附加信息一并清除，这也是为了防止内存泄露的一步重要操作

Context 生成与采样

如果不做任何限制，每个请求都应该生成一条完整的 Trace

在面对海量请求时如果也同时产生海量 Trace

就会给网络和存储带来双倍的压力，浪费很多资源

为了解决这个问题，几乎所有的 Trace 系统都会支持采样的功能

SamplingService 就是用来实现采样功能的 BootService 实现

Context 生成与采样

- **SamplingService**

  采样逻辑依赖 samplingFactorHolder 字段（AtomicInteger 类型）的自增

- **ContextManagerExtendService**

  负责创建 TracingContext 的 BootService 实现

  在 ContextManagerExtendService 创建 TracingContext 时

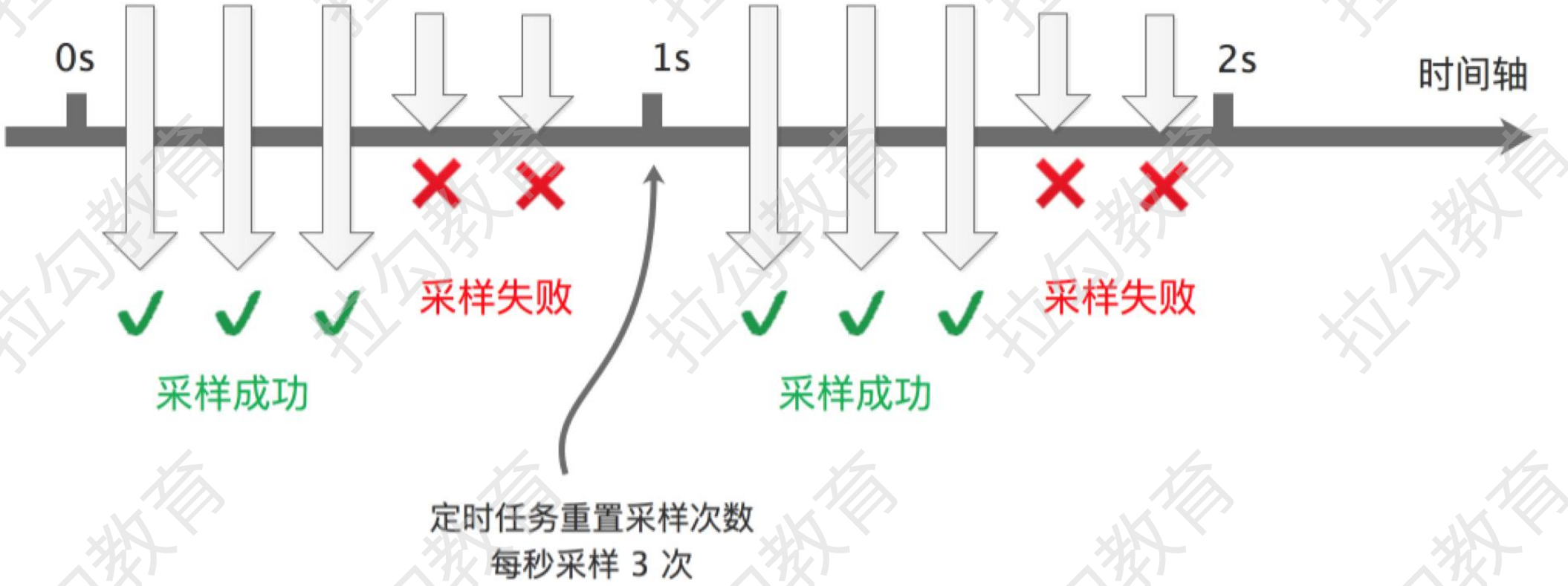
  会调用 SamplingService 的 trySampling() 方法递增 samplingFactorHolder 字段（CAS 操作）

  当增加到阈值（默认值为 3，可以通过 agent.sample_n_per_3_secs 配置进行修改）时会返回 false

  表示采样失败，这时 ContextManagerExtendService 就会生成 IgnoredTracerContext

  IgnoredTracerContext 是个空 Context 实现，不会记录 Trace 信息

Context 生成与采样



Trace 的收集


```
private void finish() {
    TraceSegment finishedSegment =
        segment.finish(isLimitMechanismWorking());
    TracingContext.ListenerManager.notifyFinish(finishedSegment);
}
```

Trace 的收集

Trace 的收集

TraceSegmentServiceClient

作为一个 TracingContextListener 接口的实现

会在 notifyFinish() 方法中，将刚刚结束的 TraceSegment 写入到 DataCarrier 中缓存

同时实现了 IConsumer 接口，封装了消费 Channels 中数据的逻辑

在 consume() 方法中会首先将消费到的 TraceSegment 对象序列化

然后通过 gRPC 请求发送到后端 OAP 集群

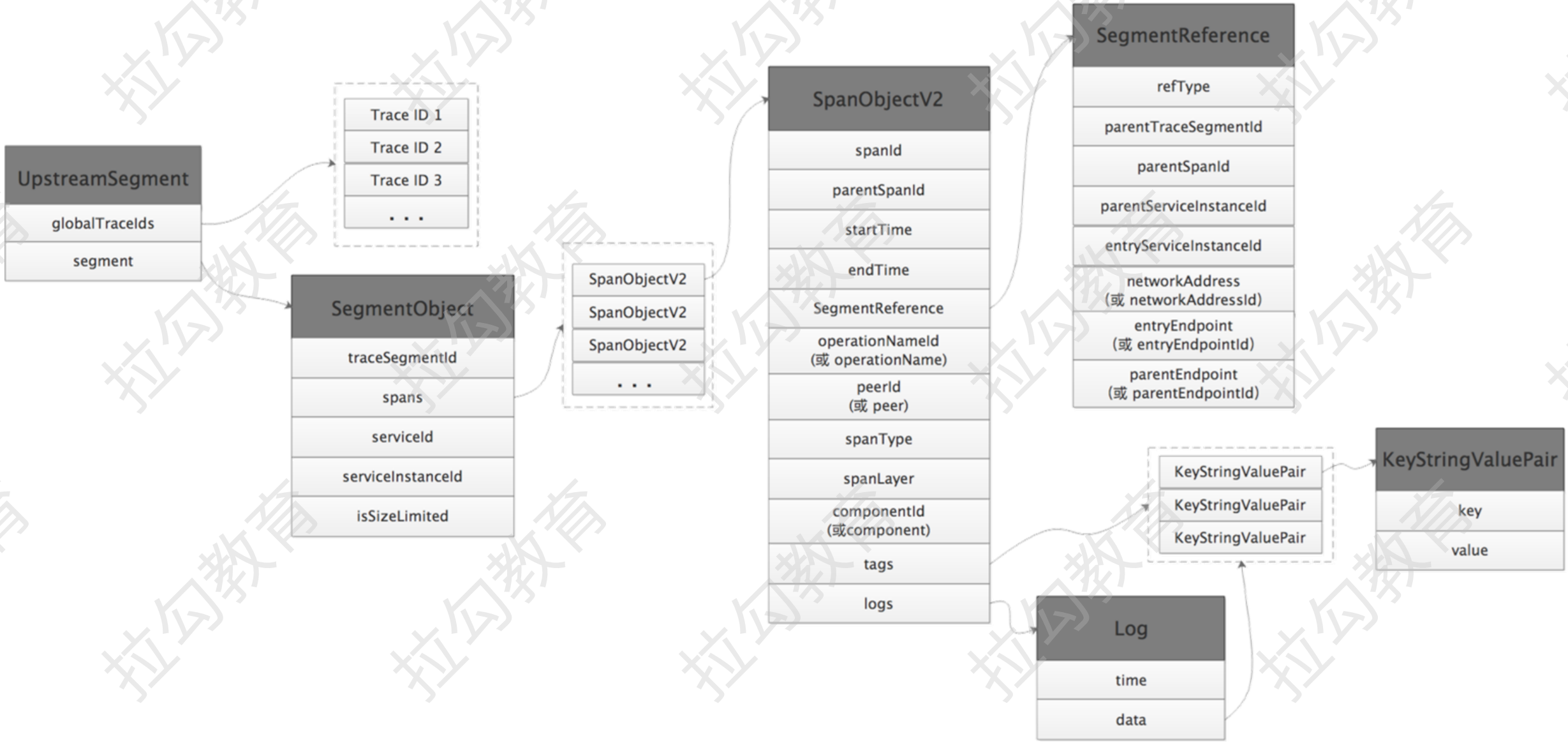
Trace 的收集


```
service TraceSegmentReportService {

    rpc collect (stream UpstreamSegment) returns (Commands) {

    }

}
```

Trace 的收集


```
message UpstreamSegment {
    repeated UniqueId globalTraceIds = 1;
    bytes segment = 2; // TraceSegment信息
}
```

Trace 的收集



Trace 的收集


```
public void consume(List<TraceSegment> data) {
    if (CONNECTED.equals(status)) { //根据底层网络连接的状态决定是否发送
        //创建GRPCStreamServiceStatus对象
        final GRPCStreamServiceStatus status =
            new GRPCStreamServiceStatus(false);
        StreamObserver<UpstreamSegment> upstreamSegmentStreamObserver
            = serviceStub.collect(new StreamObserver<Commands>() {
            public void onNext(Commands commands) {}
            public void onError(Throwable throwable) {
                //发生异常会调用finished()方法，停止等待
                status.finished();
                //通知GRPCChannelManager重新创建网络连接
                ServiceManager.INSTANCE.findService(
```

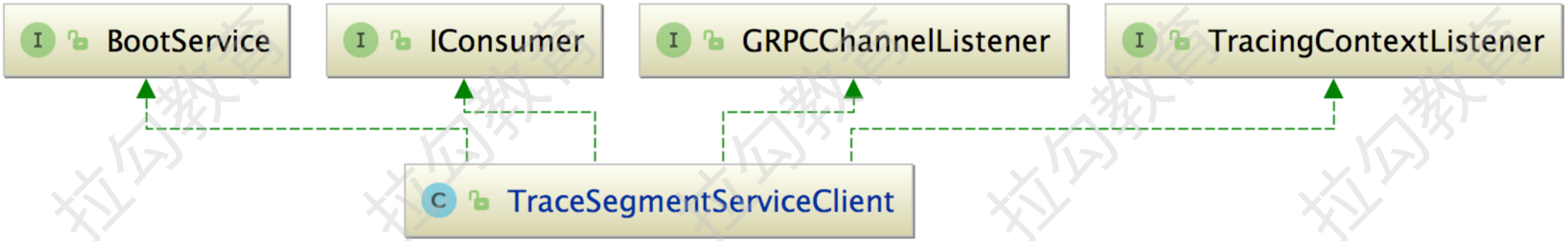
Trace 的收集


```
            ServiceManager.INSTANCE.findService(
                GRPCChannelManager.class).reportError(throwable);
        }
        public void onCompleted() {
            //发送成功之后，会调用finished()方法结束等待
            status.finished();
        }
    });
    for (TraceSegment segment : data) {
        //将TraceSegment转换成UpstreamSegment对象，然后才能进行序列化以
        //及发送操作transform()方法实现的转换逻辑并不复杂，填充字段而已
        UpstreamSegment upstreamSegment = segment.transform();
        upstreamSegmentStreamObserver.onNext(upstreamSegment);
```

Trace 的收集


```
            //将TraceSegment转换成UpstreamSegment对象，然后才能进行序列化以
            //及发送操作transform()方法实现的转换逻辑并不复杂，填充字段而已
            UpstreamSegment upstreamSegment = segment.transform();
            upstreamSegmentStreamObserver.onNext(upstreamSegment);
        }

        upstreamSegmentStreamObserver.onCompleted();
        status.wait4Finish(); //等待全部TraceSegment数据发送结束
        segmentUplinkedCounter += data.size(); //统计发送的数据量
    } else { //网络连接断开时，只进行简单统计，数据将被直接抛弃
        segmentAbandonedCounter += data.size();
    }
    printUplinkStatus(); //每隔 30s打印一下发送日志
}
```

Trace 的收集



总结

- 介绍 ContextManager 的核心实现，理清了它是如何将 TracingContext 与当前线程关联起来的
- 介绍 SamplingService 实现客户端 Trace 采样的逻辑
- 介绍上报 Trace 的 gRPC 接口

  深入分析 TraceSegmentServiceClient 收集和上报 Trace 数据的核心逻辑

Next：第14讲《Tomcat 插件原理精析，看 SkyWalking 如何增强这只 Cat》

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