《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

— 拉勾教育出品 —

第12讲（上）：剖析 Trace 在 SkyWalking 中的落地实现方案

课程回顾

SkyWalking 中 Trace 的相关概念以及实现类与 OpenTracing 中的概念基本类似

在 SkyWalking Agent 中都有对应实现

最重要的是：

SkyWalking 的设计在 Trace 级别和 Span 级别之间加了一个 Segment 概念

用于表示一个服务实例内的 Span 集合

Trace ID

在分布式链路追踪系统中

用户请求的处理过程会形成一条 Trace

Trace ID 作为 Trace 数据的唯一标识

面对海量请求时需保证其唯一性，还要保证生成 Trace ID 不会带来过多开销

业务场景中依赖数据库都不适合 Trace 的场景

Trace ID

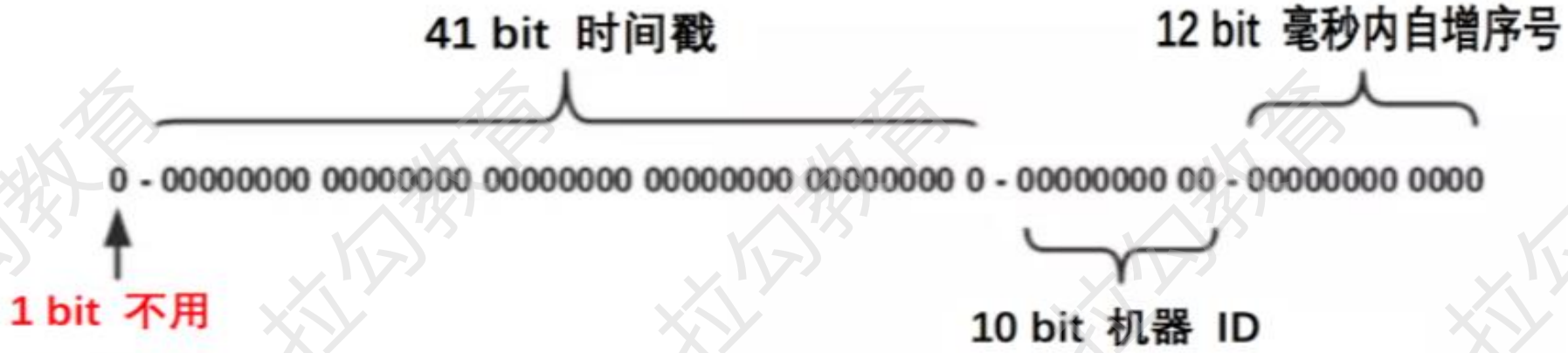
snowflake 算法是 Twitter 开源的分布式 ID 生成算法

将一个 ID（long类型）的 64 个 bit 进行切分

其中使用 41 个 bit 作为毫秒数

10 个 bit 作为机器的 ID（5 个 bit 记录数据中心的 ID，5 个 bit 记录机器的 ID）

12 bit 作为毫秒内的自增 ID，还有一个 bit 位永远是 0



Trace ID

SkyWalking ID 由三个 long 类型的字段（part1、part2、part3）构成

分别记录了 ServiceInstanceId、Thread ID 和 Context 生成序列

Context 生成序列的格式

```
${时间戳} * 10000 + 线程自增序列([0,9999])
```

Trace ID

SkyWalking ID 由三个 long 类型的字段（part1、part2、part3）构成

分别记录了 ServiceInstanceId、Thread ID 和 Context 生成序列

Context 生成序列的格式

```
${时间戳} * 10000 + 线程自增序列([0, 9999])
```

ID 对象序列化之后的格式是将 part1、part2、part3 三部分用“.”分割连接

```
${ServiceInstanceId}.${Thread ID}.(${时间戳} * 10000 + 线程自增序列([0, 9999]))
```

Trace ID


```
public static ID generate() {
    // THREAD_ID_SEQUENCE是 ThreadLocal<IDContext>类型，即每个线程
    // 维护一个 IDContext对象
    IDContext context = THREAD_ID_SEQUENCE.get();
    return new ID(SERVICE_INSTANCE_ID, // service_intance_id
        Thread.currentThread().getId(), // 当前线程的ID
        context.nextSeq() // 线程内生成的序列号
    );
}
```

Trace ID

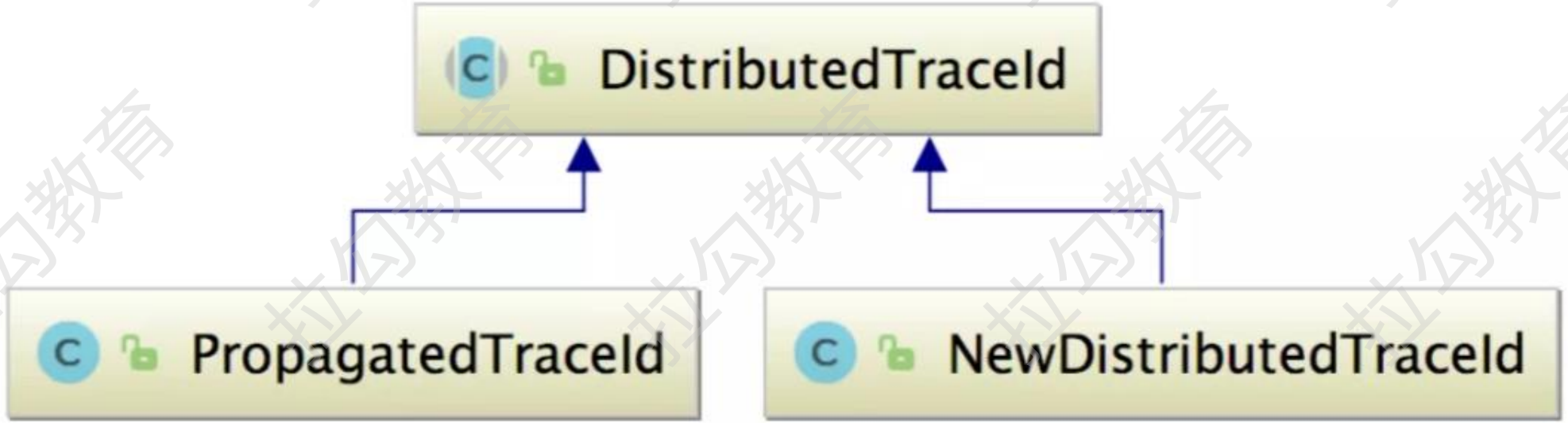
timestamp() 方法在返回时间戳时

会处理时间回拨的场景（使用 Random 随机生成一个时间戳）

nextThreadSeq() 方法的返回值在 [0 , 9999] 这个范围内循环

```
private long nextSeq() {
    return timestamp() * 10000 + nextThreadSeq();
}
```

Trace ID



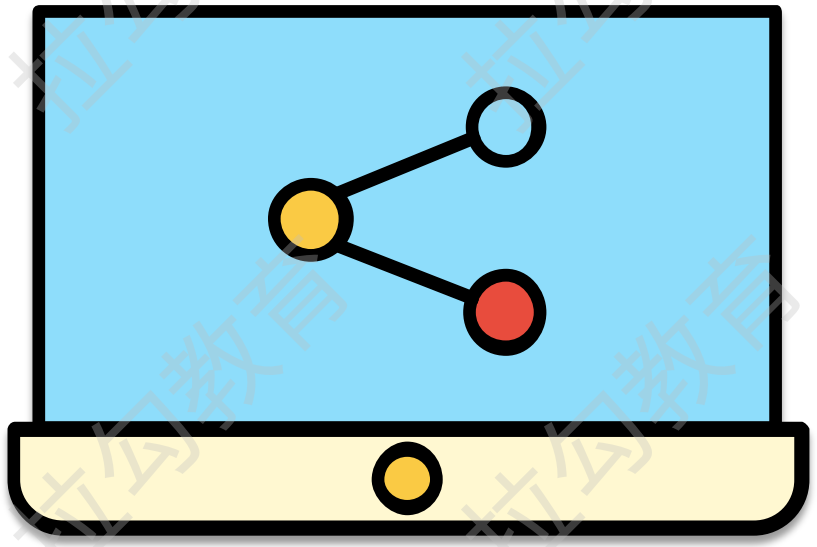
TraceSegment

SkyWalking 中

TraceSegment 是一个介于 Trace 与 Span 之间的概念

它是一条 Trace 的一段，可以包含多个 Span

在微服务架构中，一个请求基本都会涉及跨进程（以及跨线程）的操作



TraceSegment

- **traceSegmentId（ID 类型）**

  TraceSegment 的全局唯一标识，是由前面介绍的 GlobalIdGenerator 生成的

- **refs（List<TraceSegmentRef> 类型）**

  指向父 TraceSegment。在常见的 RPC 调用、HTTP 请求等跨进程调用中

  一个 TraceSegment 最多只有一个父 TraceSegment

  但在一个 Consumer 批量消费 MQ 消息时，同一批内的消息可能来自不同的 Producer

  会导致 Consumer 线程对应的 TraceSegment 有多个父 TraceSegment

- **relatedGlobalTraces（DistributedTraceIds 类型）**

  记录当前 TraceSegment 所属 Trace 的 Trace ID

TraceSegment

- **spans（List<AbstractTracingSpan> 类型）**

  当前 TraceSegment 包含的所有 Span

- **ignore（boolean 类型）**

  ignore 字段表示当前 TraceSegment 是否被忽略

  主要是为了忽略一些问题 TraceSegment（主要是对只包含一个 Span 的 Trace 进行采样收集）

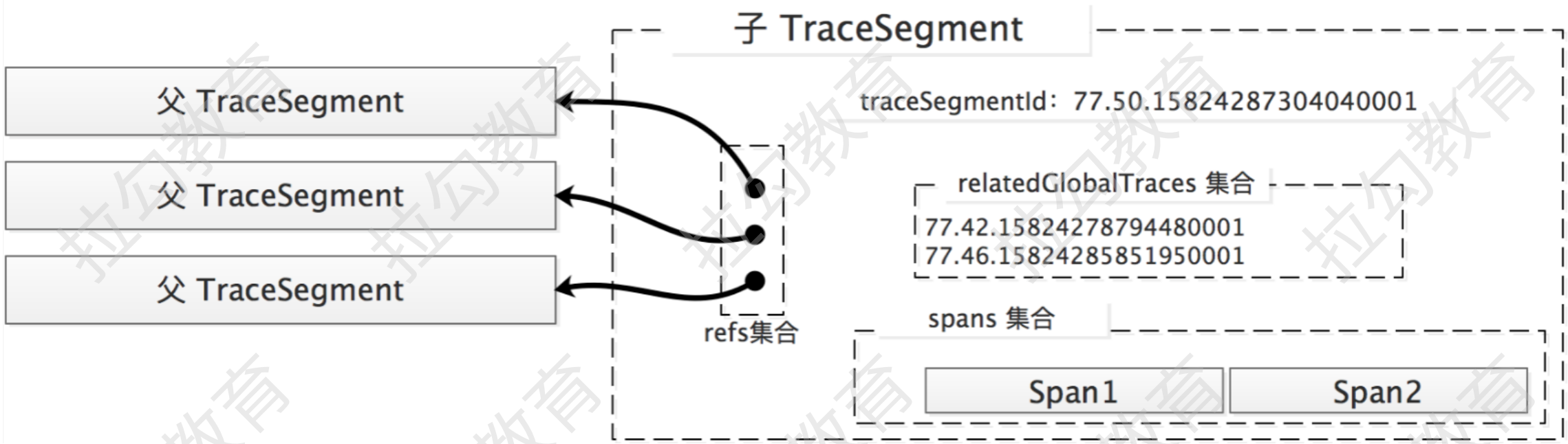
- **isSizeLimited（boolean 类型）**

  是一个容错设计，例如业务代码出现了死循环 Bug，可能会向相应的 TraceSegment 中不断追加 Span

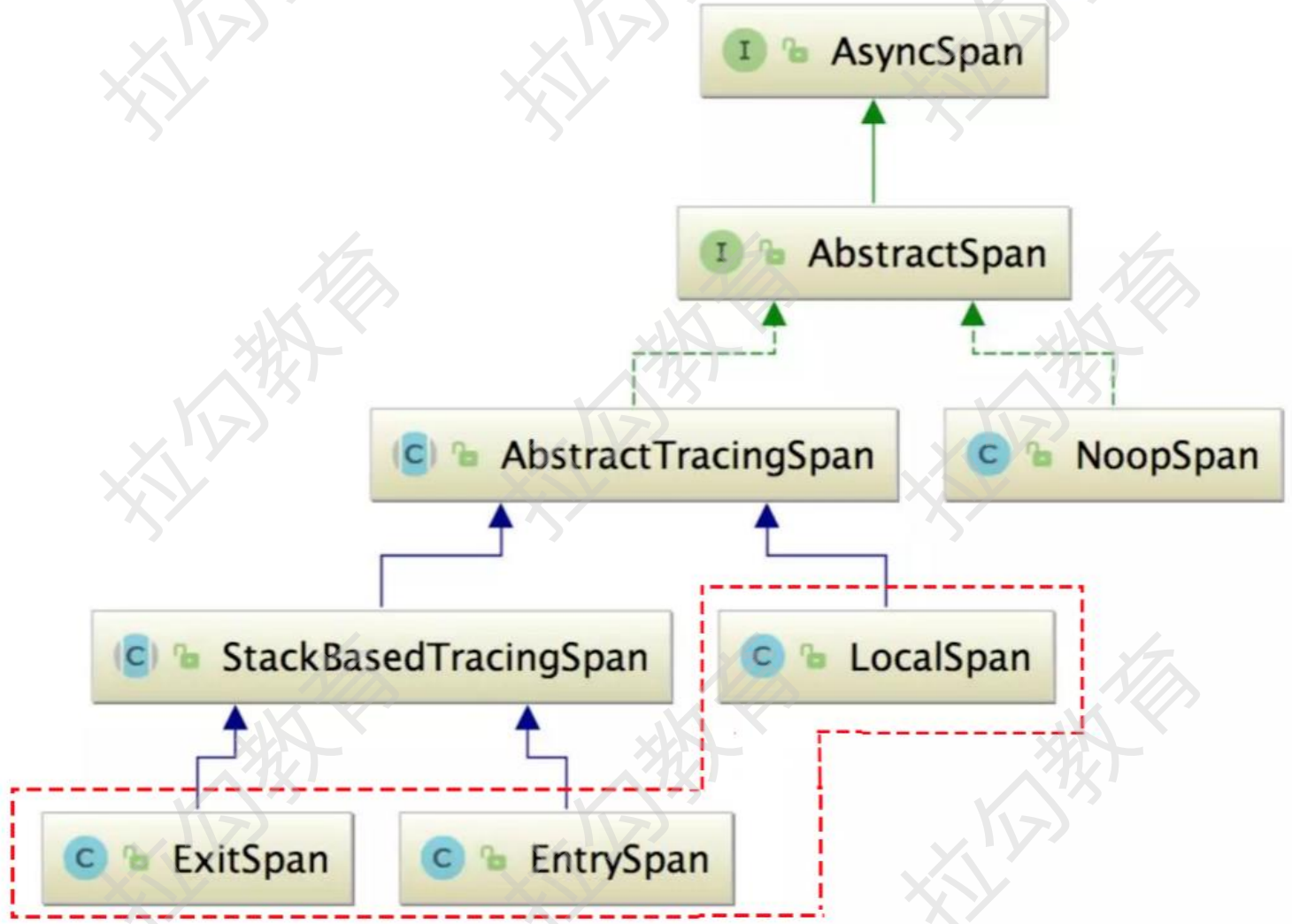
  为了防止对应用内存以及后端存储造成不必要的压力

  每个 TraceSegment 中 Span 的个数是有上限的（默认值为 300），超过上限后就不再添加 Span

TraceSegment



Span



Span

- **EntrySpan**

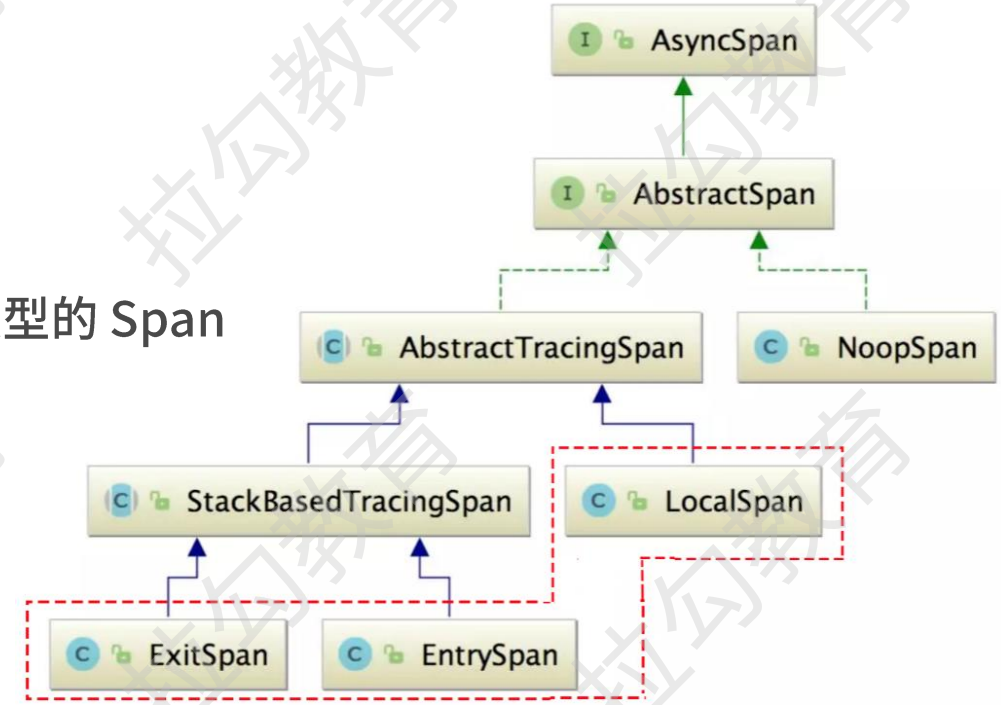
  当请求进入服务时会创建 EntrySpan 类型的 Span，它也是 TraceSegment 中的第一个

- **LocalSpan**

  是在本地方法调用时可能创建的 Span 类型

- **ExitSpan**

  当请求离开当前服务、进入其他服务时会创建 ExitSpan 类型的 Span



Span

- **prepareForAsync() 方法**

  Span 在当前线程结束，但未被彻底关闭，依然是存活的

- **asyncFinish()方法**

  当前 Span 真正关闭，与 prepareForAsync() 方法成对出现

Span

- **getSpanId() 方法**

  用来获得当前 Span 的 ID，Span ID 是一个 int 类型的值，在其所属的 TraceSegment 中唯一

  在创建 Span 对象时生成，从 0 开始自增

- **setOperationName()/setOperationId() 方法**

  用来设置 operation 名称（或 operation ID），这两个信息是互斥的

  在 AbstractSpan 的具体实现（即 AbstractTracingSpan）中

  分别对应 operationId 和 operationName 两个字段，两者只能有一个字段有值

Span

- **setComponent() 方法**

  用于设置组件类型

  它有两个重载，在 AbstractTracingSpan 实现中

  有 componentId 和 componentName 两个字段，两个重载分别用于设置这两个字段

  在 ComponentsDefine 中可以找到 SkyWalking 目前支持的组件类型

- **setLayer() 方法**

  用于设置 SpanLayer，也就是当前 Span 所处的位置。SpanLayer 是个枚举，可选项有 DB、RPC_FRAMEWORK、HTTP、MQ、CACHE。

Span

- **tag(AbstractTag, String) 方法**

  用于为当前 Span 添加键值对的 Tags，一个 Span 可以有多个 Tags

  AbstractTag 中不仅包含了 String 类型的 Key 值，还包含 Tag 的 ID 以及 canOverwrite 标识

  AbstractTracingSpan 实现通过维护一个 List<TagValuePair> 集合（tags 字段）来记录 Tag 信息

  TagValuePair 中则封装了 AbstractTag 类型的 Key 以及 String 类型的 Value

Span

- **log() 方法**

  用于向当前 Span 中添加 Log，一个 Span 可以包含多条日志

  在 AbstractTracingSpan 实现中通过维护一个 List<LogDataEntity> 集合（logs 字段）来记录 Log

  LogDataEntity 会记录日志的时间戳以及 KV 信息

  以异常日志为例，其中就会包含一个 Key 为“stack”的 KV，其 value 为异常堆栈

Span

- **start() 方法**

  开启 Span，其中会设置当前 Span 的开始时间以及调用层级等信息

- **isEntry() 方法**

  判断当前是否是 EntrySpan。EntrySpan 的具体实现后面详细介绍

- **isExit() 方法**

  判断当前是否是 ExitSpan。ExitSpan 的具体实现后面详细介绍

- **ref() 方法**

  用于设置关联的 TraceSegment

Span


```
protected int spanId; // span的ID
protected int parentSpanId; // 记录父Span的ID
protected List<TagValuePair> tags; // 记录Tags的集合
protected long startTime, endTime; // Span的起止时间
protected boolean errorOccurred = false; // 标识该Span中是否发生异常
protected List<TraceSegmentRef> refs; // 指向所属TraceSegment
// context字段指向TraceContext，TraceContext与当前线程绑定，与TraceSegment
// 一一对应
protected volatile AbstractTracerContext context;
```

Span

- **finish(TraceSegment) 方法**

  该方法会关闭当前 Span，具体行为是用 endTime 字段记录当前时间

  并将当前 Span 记录到所属 TraceSegment 的 spans 集合中

- **transform() 方法**

  该方法会在 Agent 上报 TraceSegment 数据之前调用

  会将当前 AbstractTracingSpan 对象转换成 SpanObjectV2 对象

  SpanObjectV2 是在 proto 文件中定义的结构体

  后面 gRPC 上报 TraceSegment 数据时会将其序列化

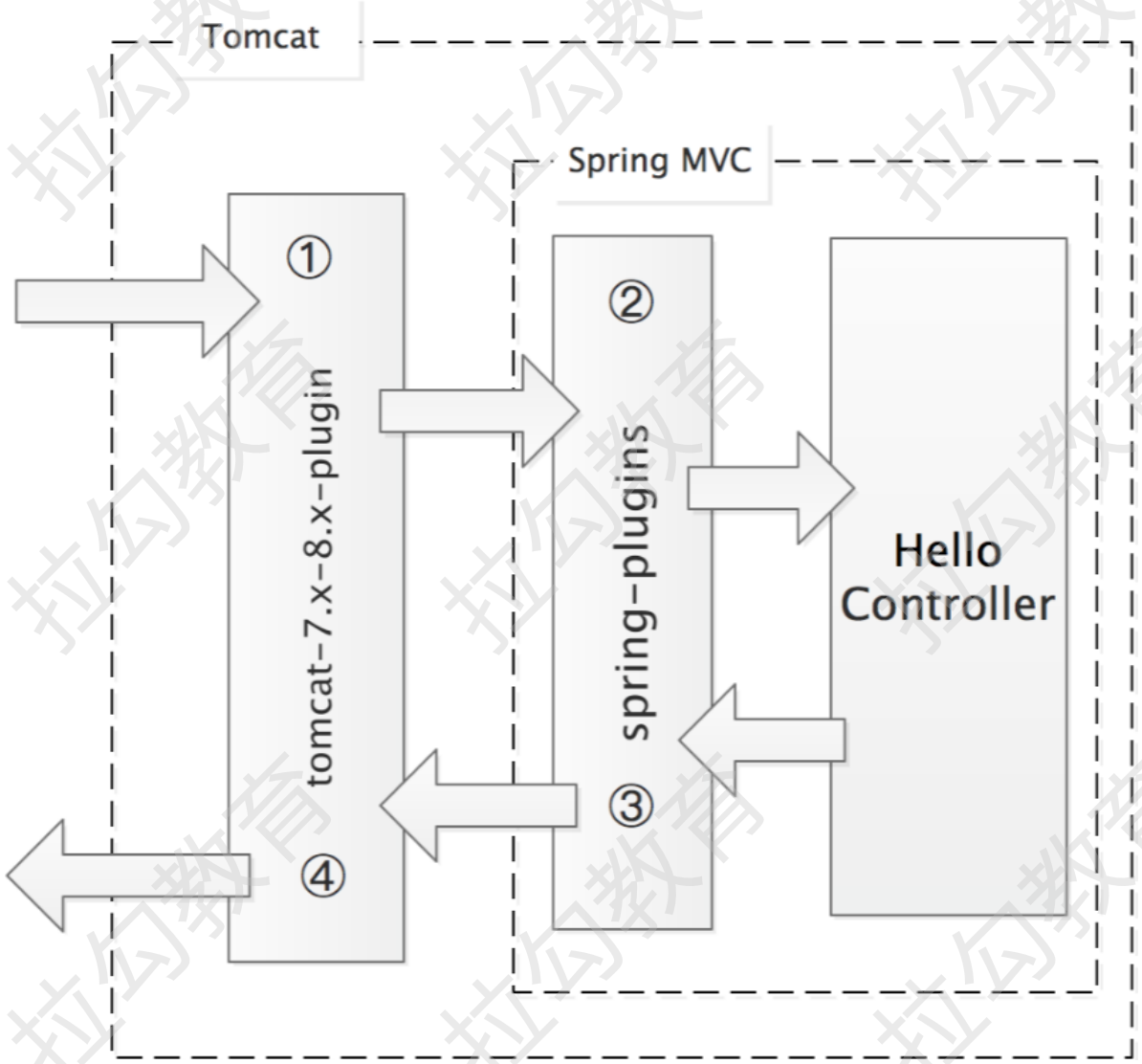
Span

EntrySpan 表示的是一个服务的入口 Span

是 TraceSegment 的第一个 Span，出现在服务提供方的入口

例如：**Dubbo Provider**、**Tomcat**、**Spring MVC** 等

Span



Span

在 start() 方法中会有下面几个操作：

a) 将 stackDepth 字段（定义在 StackBasedTracingSpan 中）加 1

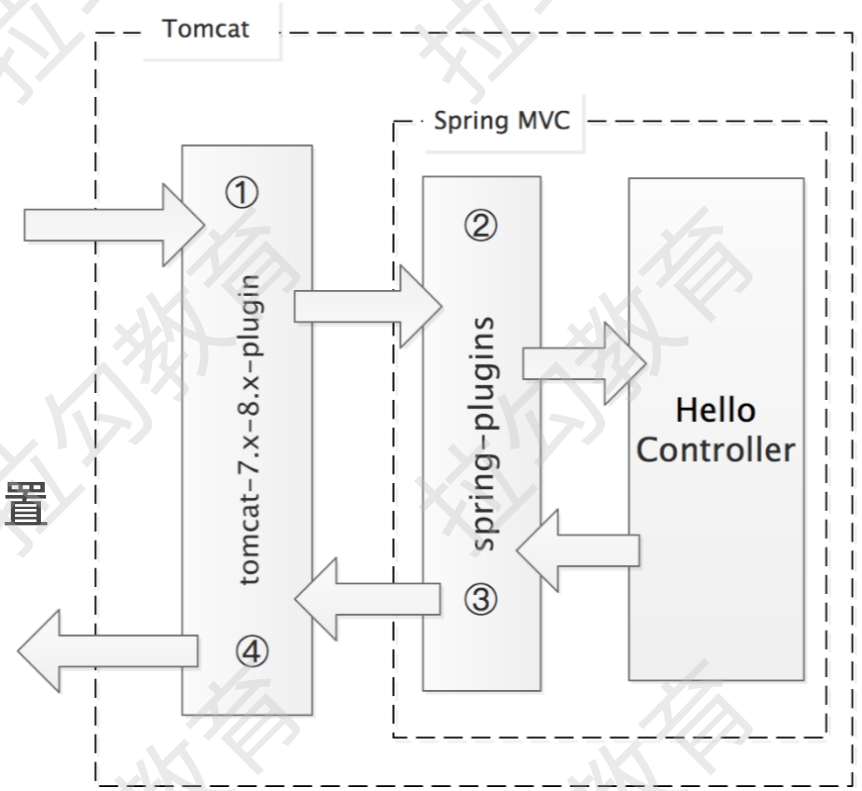
   stackDepth 表示当前所处的插件栈深度。

b) 更新 currentMaxDepth 字段（定义在 EntrySpan 中）

   currentMaxDepth 会记录该EntrySpan 到达过的插件栈的最深位置

c) 此时第一次启动 EntrySpan 时会更新 startTime 字段

   记录请求开始时间



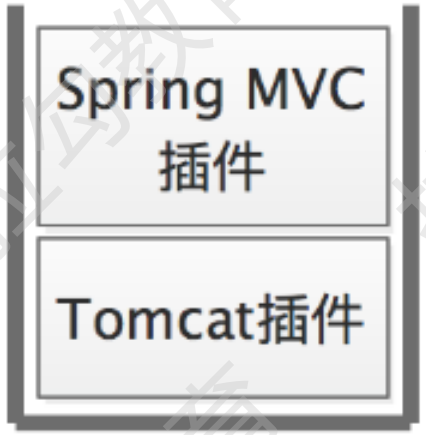
此时插件栈的状态

stackDepth = 1

currentMaxDepth = 1

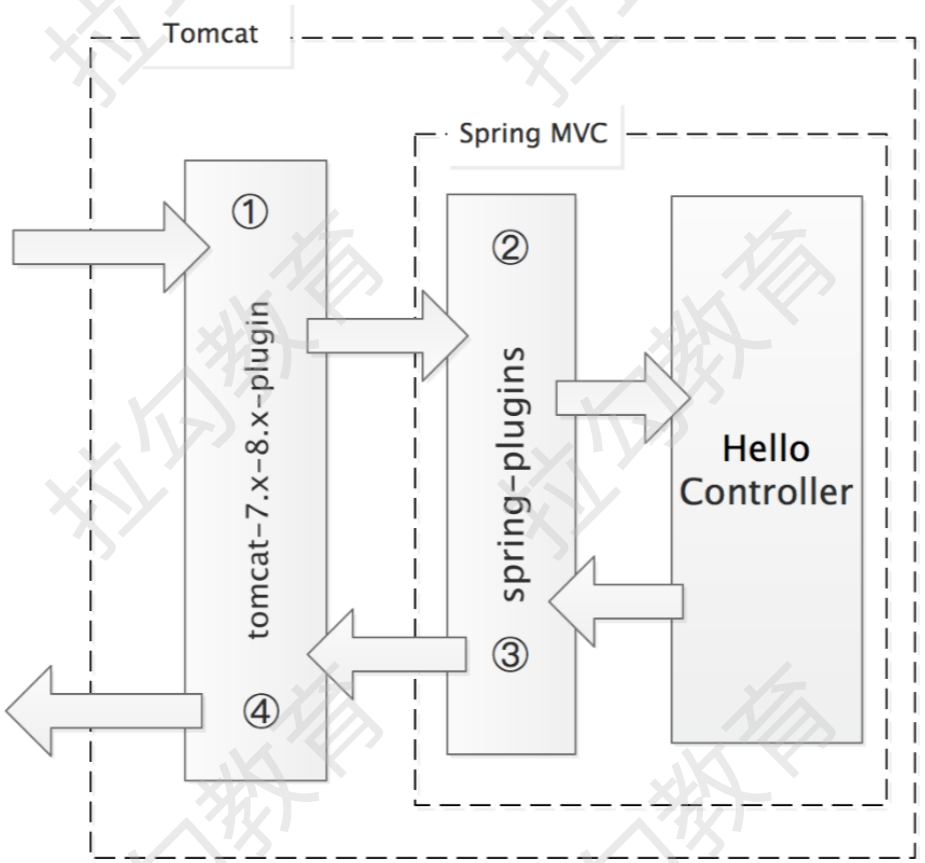
Span

此时插件栈的状态



stackDepth = 2

currentMaxDepth = 2

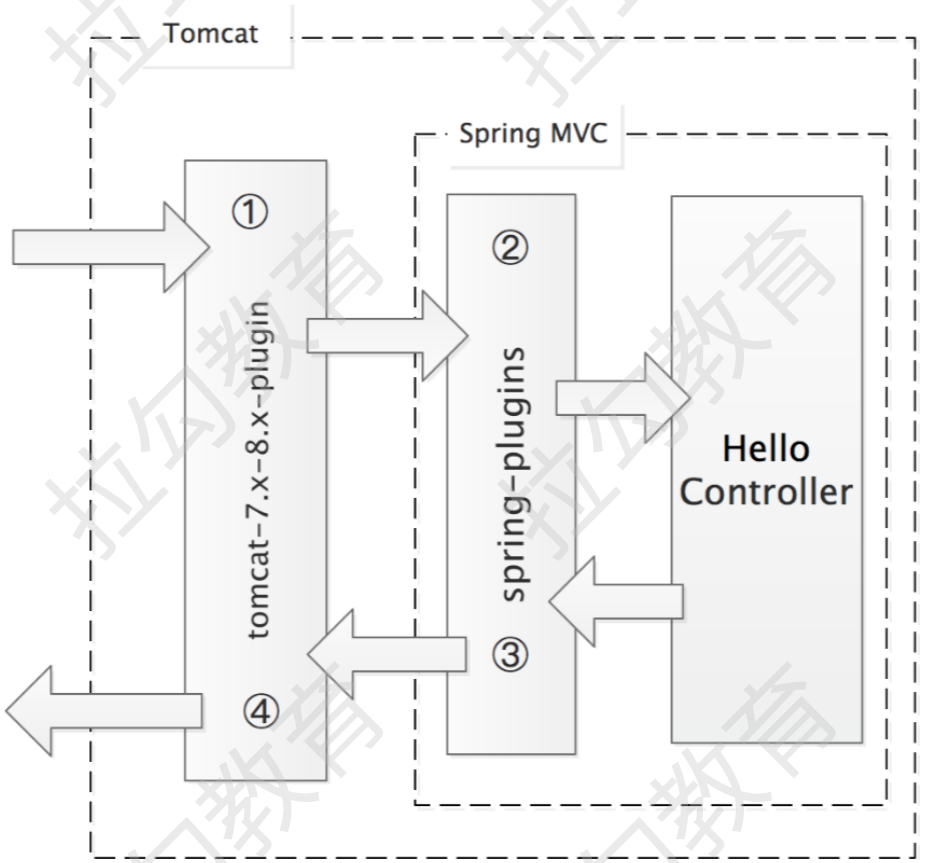


Span

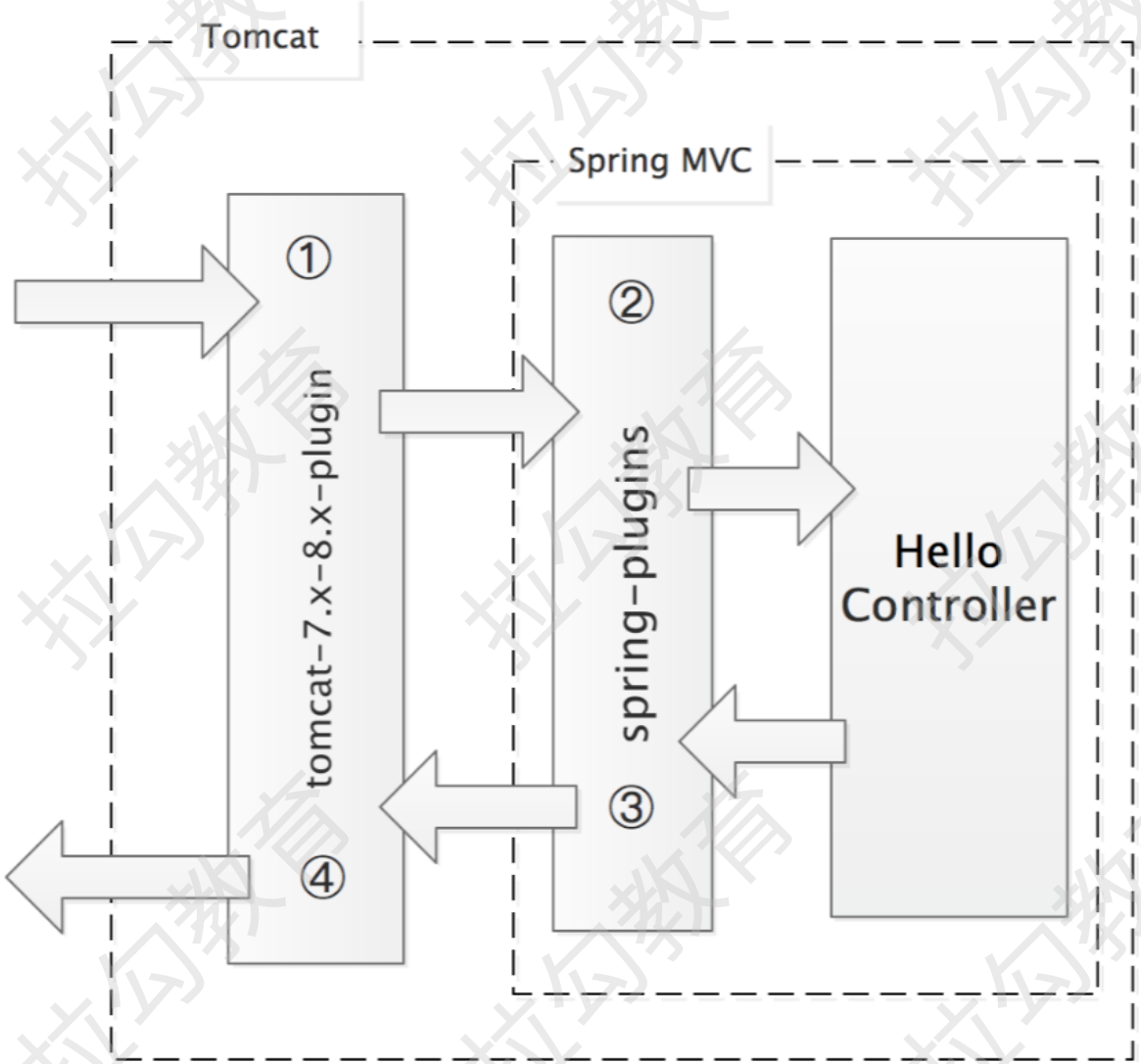
此时插件栈的状态

stackDepth = 1

currentMaxDepth = 2



Span



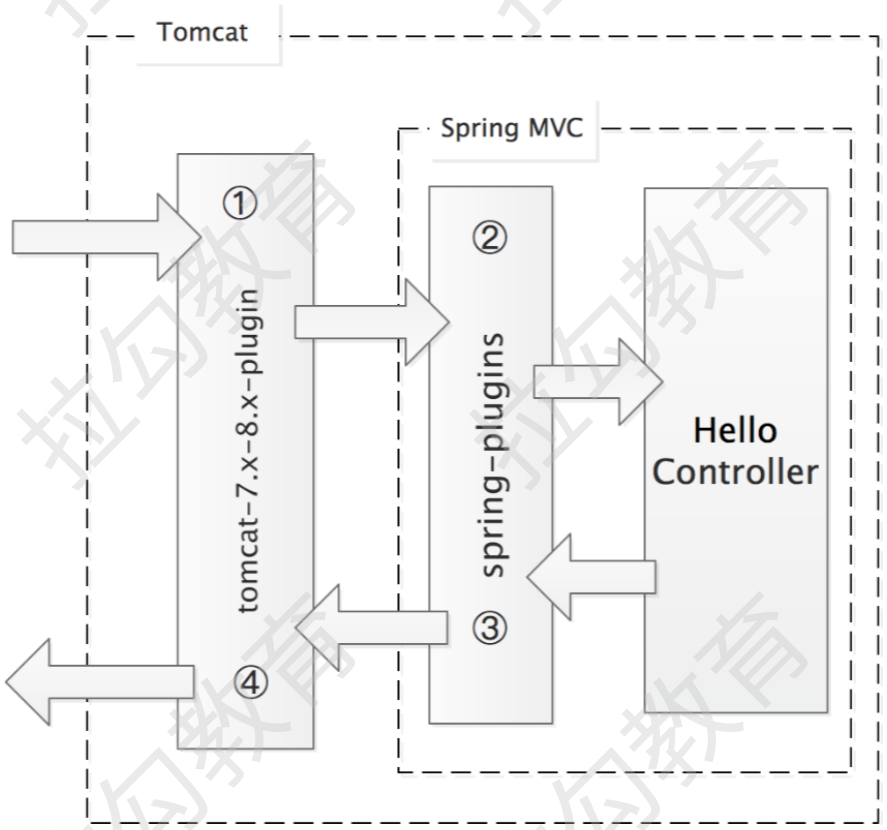
Span

需要注意两点：

1. 在调用 start() 方法时，会将之前设置的 component、Tags、Log 等信息全部清理掉（startTime不会清理）

   上例中请求到 Spring MVC 插件之前（即 ② 处之前）设置的这些信息都会被清理掉

2. stackDepth 与 currentMaxDepth 不相等时（上例中 ③ 处）无法记录上述字段的信息。通过这两点，我们知道 EntrySpan 实际上只会记录最贴近业务侧的 Span 信息



Span

StackBasedTracingSpan 除了将“栈”概念与 EntrySpan 结合之外

还添加了 **peer（以及 peerId）字**段来**记录远端地址**

在发送远程调用时创建的 ExitSpan 会将该记录用于对端地址

Span

ExitSpan 表示的是出口 Span

如果在一个调用栈里面出现多个插件嵌套的场景，也需要通过“栈”的方式进行处理

只会在第一个插件中创建 ExitSpan

后续调用的 ExitSpan.start() 方法并不会更新 startTime，只会增加栈的深度

ExitSpan 中只会记录最贴近当前服务侧的 Span 信息

Span

一个 TraceSegment 可以有多个 ExitSpan

例如：

Dubbo A 服务在处理一个请求时，会调用 Dubbo B 服务

在得到响应之后，会紧接着调用 Dubbo C 服务

该 TraceSegment 就有了两个完全独立的 ExitSpan

Span

LocalSpan 表示一个本地方法调用

直接继承 AbstractTracingSpan，由于未继承 StackBasedTracingSpan

所以也**不能 start 或 end 多次**

Next：第3讲《如何设计与实现统一资源管理与调度系统》

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