《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

— 拉勾教育出品 —

第13讲：剖析 Trace 在 SkyWalking 中的落地实现方案（下）

TraceSegmentRef

- 父 Span 信息
  - traceSegmentId（ID 类型）：父 TraceSegment 的 ID
  - spanId（int 类型）：父 Span 的 ID，与 traceSegmentId 结合就可以确定父 Span
  - type（SegmentRefType 类型）：SegmentRefType 是个枚举，可选值有：CROSS_PROCESS、CROSS_THREAD，分别表示跨进程调用和跨线程调用

TraceSegmentRef

- **父应用（或者说，上游调用方）信息**
  - peerId 和 peerHost：父应用（即上游调用方）的地址信息
  - parentServiceInstanceId（int 类型）：父应用（即上游应用）的 ServiceInstanceId
  - parentEndpointName 和 parentEndpointId：父应用的（即上游应用）的 Endpoint 信息
- **入口信息（在整条 Trace 中都会传递该信息）**
  - entryServiceInstanceId：入口应用的 ServiceInstanceId
  - entryEndpointName 和 entryEndpointId：入口 Endpoint 信息

Context

- **inject(ContextCarrier) 方法**

  在跨进程调用之前，调用方会通过 inject() 方法将当前 Context 上下文记录的全部信息注入到

  ContextCarrier 参数中，Agent 后续会将 ContextCarrier 序列化并随远程调用进行传播

  ContextCarrier 的具体实现在后面会详细分析

- **extract(ContextCarrier) 方法**

  跨进程调用的接收方会反序列化得到 ContextCarrier 对象

  通过 extract() 方法从 ContextCarrier 中读取上游传递下来的 Trace 信息并记录到当前的 Context 上下文中

Context

- **ContextSnapshot capture() 方法**

  在跨线程调用之前，SkyWalking Agent 会通过 capture() 方法将当前 Context 进行快照

  然后将快照传递给其他线程

- **continued(ContextSnapshot) 方法**

  跨线程调用的接收方会从收到的 ContextSnapshot 中读取 Trace 信息并填充到当前 Context 上下文中

- **getReadableGlobalTraceId() 方法**

  用于获取当前 Context 关联的 TraceId

Context

- **createEntrySpan()、createLocalSpan() 方法、createExitSpan() 方法**

  用于创建 Span

- **activeSpan() 方法**

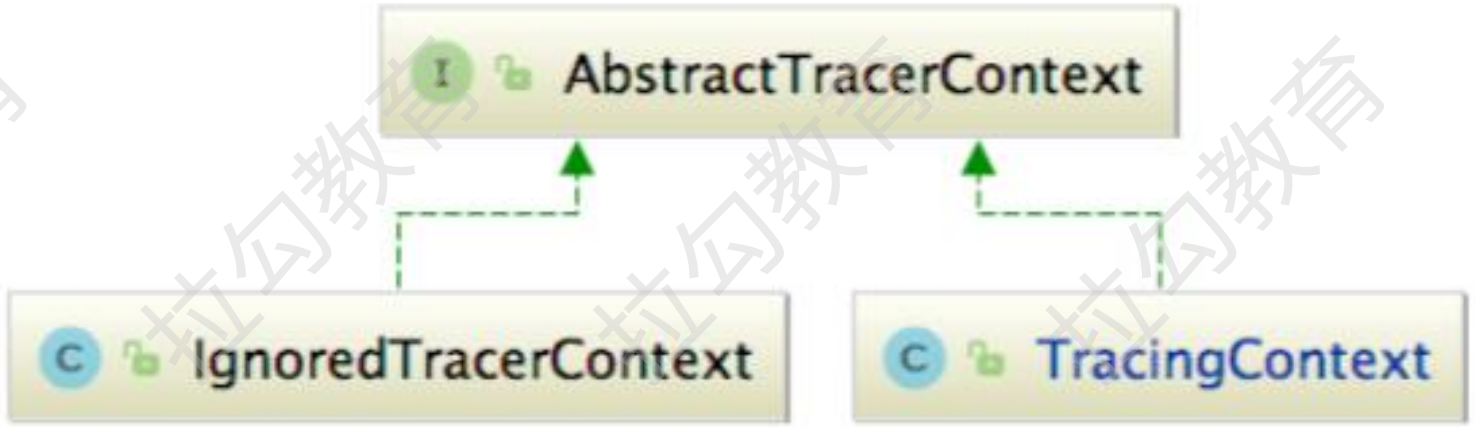
  用于获得当前活跃的 Span

  先创建的 Span 后结束

- **stopSpan(AbstractSpan) 方法**

  用于停止指定 Span

Context



Context

- **samplingService（SamplingService 类型）**

  负责完成 Agent 端的 Trace 采样，后面会展开介绍具体的采样逻辑。

- **segment（TraceSegment 类型）**

  是与当前 Context 上下文关联的 TraceSegment 对象，在 TracingContext 的构造方法中会创建该对象

- **activeSpanStack（LinkedList<AbstractSpan> 类型）**

  用于记录当前 TraceSegment 中所有活跃的 Span（即未关闭的 Span）

- **spanIdGenerator（int 类型）**

  是 Span ID 自增序列，初始值为 0

  该字段的自增操作都是在一个线程中完成的，所以无需加锁

管理 Span


```
public AbstractSpan createEntrySpan(final String operationName) {
    if (isLimitMechanismWorking()) {
      //前面提到过，默认配置下，每个TraceSegment只能放300个Span
      NoopSpan span = new NoopSpan(); //超过300就放 NoopSpan
      return push(span); //将Span记录到activeSpanStack这个栈中
    }
    AbstractSpan entrySpan;
    final AbstractSpan parentSpan = peek(); //读取栈顶Span，即当前Span
    final int parentSpanId = parentSpan == null ? -1 :
        parentSpan.getSpanId();
    if (parentSpan != null && parentSpan.isEntry()) {
      //更新 operationId(省略operationName的处理逻辑)，省略
```

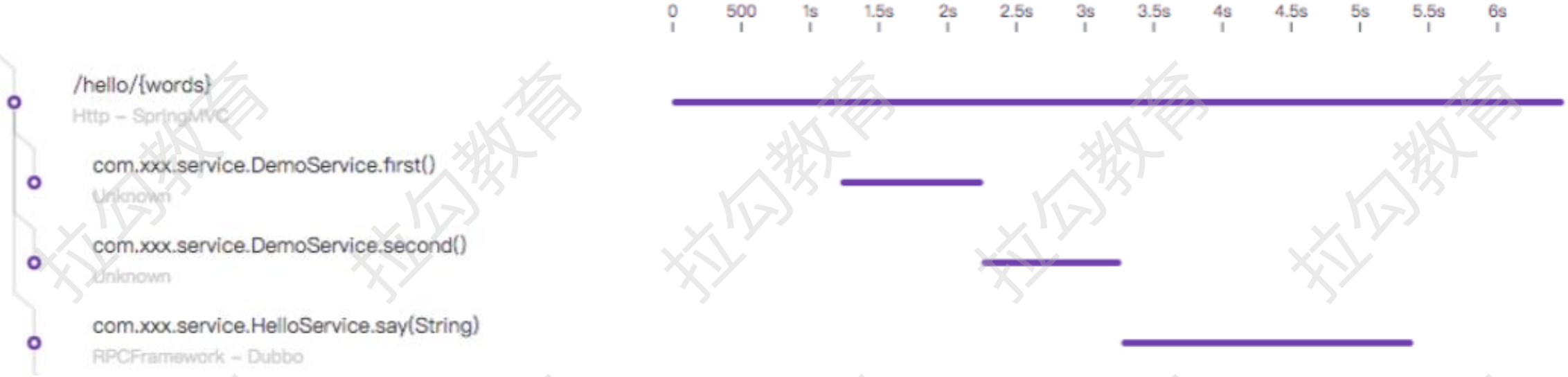
管理 Span


```
    //更新 operationId(省略operationName的处理逻辑)，省略
    // EndpointNameDictionary的处理，其核心逻辑在前面的小节已经介绍过了
    entrySpan = parentSpan.setOperationId(operationId);
    //重新调用 start()方法，前面提到过，start()方法会重置
    // operationId(以及或operationName)之外的其他字段
    return entrySpan.start();
  } else {
    //新建 EntrySpan对象，spanIdGenerator生成Span ID并递增
    entrySpan = new EntrySpan(spanIdGenerator++, parentSpanId,
            operationId);
    //调用 start()方法，第一次调用start()方法时会设置startTime
    entrySpan.start();
```

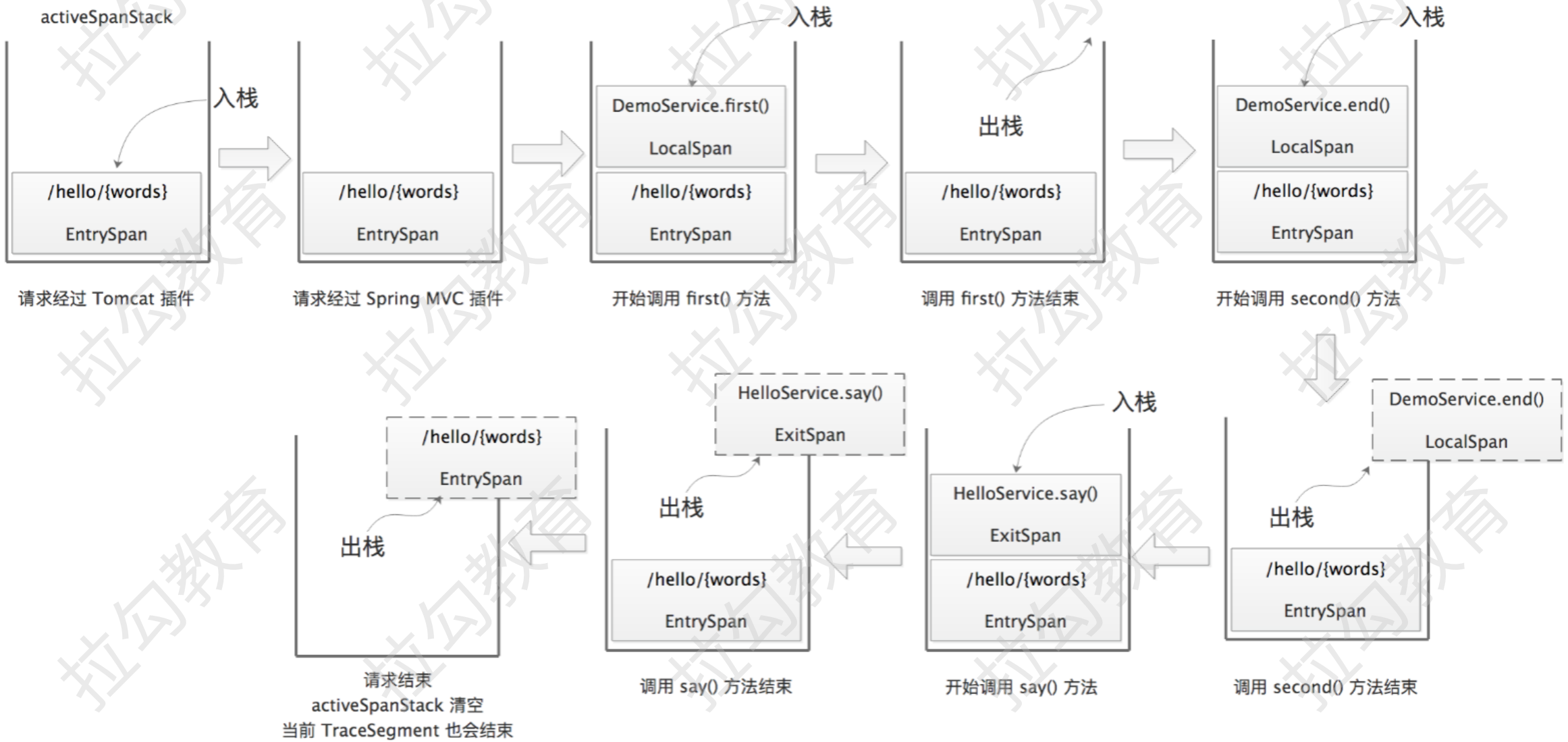
管理 Span


```
        // operationId(以及或operationName)之外的其他字段
        return entrySpan.start();
    } else {
        //新建 EntrySpan对象，spanIdGenerator生成Span ID并递增
        entrySpan = new EntrySpan(spanIdGenerator++, parentSpanId,
                operationId);
        //调用 start()方法，第一次调用start()方法时会设置startTime
        entrySpan.start();
        //将新建的Span添加到activeSpanStack栈的栈顶
        return push(entrySpan);
    }
}
```

管理 Span



管理 Span



管理 Span


```
public AbstractSpan createExitSpan(String operationName,
        String remotePeer) {
    AbstractSpan exitSpan;
    //从activeSpanStack栈顶获取当前Span
    AbstractSpan parentSpan = peek();
    if (parentSpan != null && parentSpan.isExit()) {
        //当前Span已经是ExitSpan，则不再新建ExitSpan，而是调用其start()方法
        exitSpan = parentSpan;
    } else {
        //当前Span不是 ExitSpan，就新建一个ExitSpan
        final int parentSpanId = parentSpan == null ? -1 :
                parentSpan.getSpanId();
```

管理 Span


```
        exitSpan = parentSpan;
    } else {
        //当前Span不是 ExitSpan，就新建一个ExitSpan
        final int parentSpanId = parentSpan == null ? -1 :
                parentSpan.getSpanId();
        exitSpan = new ExitSpan(spanIdGenerator++, parentSpanId,
                operationId, peerId);
        push(exitSpan); //将新建的ExitSpan入栈
    }
    exitSpan.start();//调用start()方法
    return exitSpan;
}
```

管理 Span


```
public boolean stopSpan(AbstractSpan span) {
    AbstractSpan lastSpan = peek(); //获取当前栈顶的Span对象
    if (lastSpan == span) { //只能关闭当前活跃Span对象，否则抛异常
        if (lastSpan instanceof AbstractTracingSpan) {
            if (lastSpan.finish(segment)) { //尝试关闭Span
                //当Span完全关闭之后，会将其出栈(即从activeSpanStack中删除)
                pop();
            }
        } else {
            pop(); //针对NoopSpan类型Span的处理
        }
    } else {
```

管理 Span


```
        } else {
            pop(); //针对NoopSpan类型Span的处理
        }
    } else {
        throw new IllegalStateException("Stopping the unexpected...");
    }

    // TraceSegment中全部Span都关闭(且异步状态的Span也关闭了)，则当前
    // TraceSegment也会关闭，该关闭会触发TraceSegment上传操作，后面详述
    if (checkFinishConditions()) {
        finish();
    }

    return activeSpanStack.isEmpty();
}
```

跨进程(跨线程)传播

ContextCarrier 是 Context 上下文的搬运工（Carrier）

实现 Serializable 接口，负责在进程之间搬运 TracingContext 的一些基本信息

跨进程调用涉及 Client 和 Server 两个系统

所以 ContextCarrier 中的字段 Client 和 Server 含义不同

跨进程(跨线程)传播

- **traceSegmentId（ID 类型）**

  记录 Client 中 TraceSegment ID

  从 Server 角度看，记录的是父 TraceSegment 的 ID

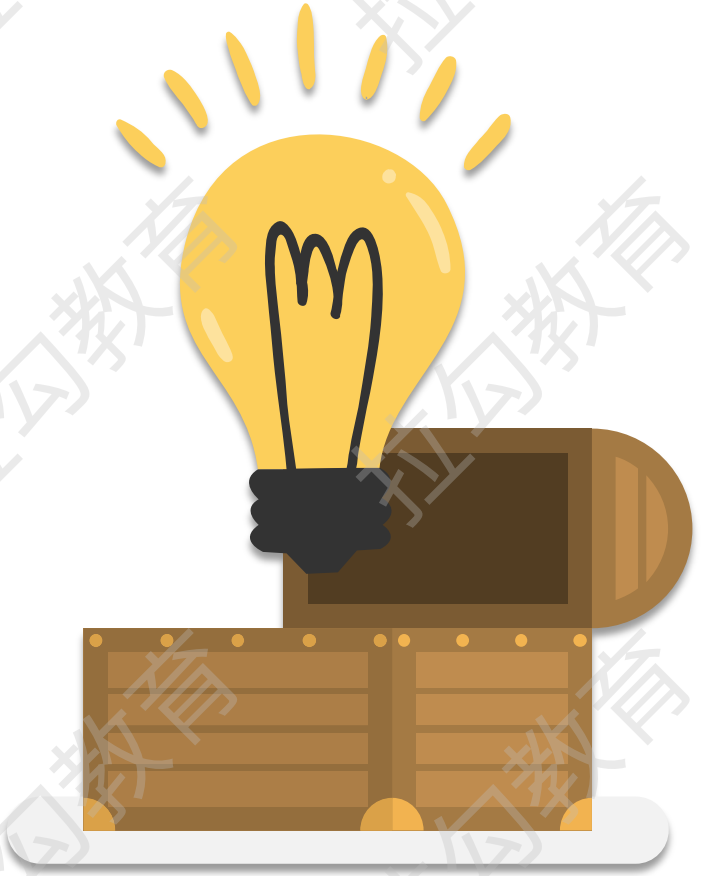
- **spanId（int 类型）**

  从 Client 角度看，它记录了当前 ExitSpan 的 ID

  从 Server 角度，看记录的是父 Span ID

- **parentServiceInstanceId（int 类型）**

  记录的是 Client 服务实例的 ID



跨进程(跨线程)传播

- **peerHost（String 类型）**

  记录 Server 端的地址（这里 peerName 和 peerId 共用了同一个字段）

  以 "#" 开头时记录的是 peerName，否则记录的是 peerId

  在 inject() 方法（或 extract() 方法）中填充（或读取）该字段时会专门判断处理开头的"#"字符

- **entryEndpointName（String 类型）**

  记录整个 Trace 的入口 EndpointName，该值在整个 Trace 中传播

- **parentEndpointName（String 类型）**

  记录 Client 入口 EndpointName（或 EndpointId）

  以 "#" 开头的时候，记录的是 EndpointName，否则记录的是 EndpointId

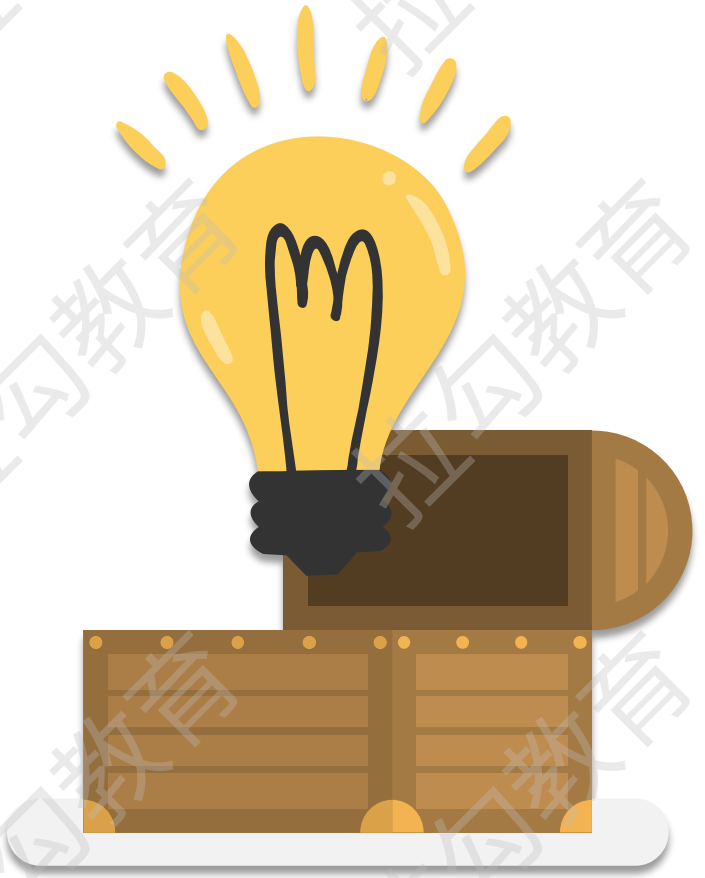
跨进程(跨线程)传播

- **primaryDistributedTraceId（DistributedTraceId 类型）**

  记录当前 Trace ID

- **entryServiceInstanceId（int 类型）**

  记录当前 Trace 的入口服务实例 ID

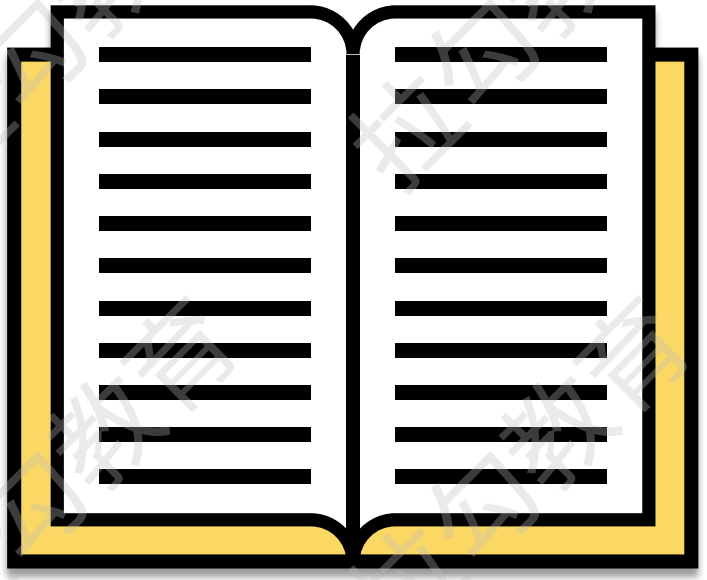


跨进程(跨线程)传播

跨进程传播 Context 上下文信息的核心流程：

远程调用的 Client 端会调用 **inject(ContextCarrier) 方法**

将当前 TracingContext 中记录的 Trace 上下文信息填充到传入的 ContextCarrier 对象



跨进程(跨线程)传播

跨进程(跨线程)传播


```
// 有多个版本的结构，这里只关注最新的V2版本
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()));
}
```

跨进程(跨线程)传播

racingContext 对跨线程传播的支持

涉及 **capture() 方法**和 **continued() 方法**

跨线程传播时使用 ContextSnapshot 为 Context 上下文创建快照

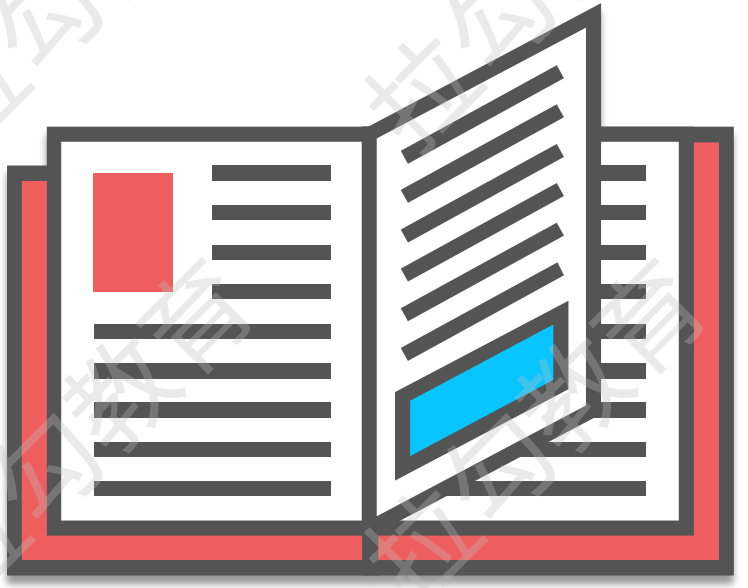
因为是在一个 JVM 中，所以 ContextSnapshot 不涉及序列化的问题

也无需携带服务实例 ID 以及 peerHost 信息

其他核心字段与 ContextCarrier 类似

总结

1. 介绍了 Trace ID 的实现结构
2. 分析 TraceSegment 如何维护底层 Span 集合以及父子关系
3. 深入剖析了 3 种类型的 Span 以及 StackBasedTracingSpan 引入的栈的概念
4. 剖析与 TraceSegment 相对应的 TracingContext 的实现，管理着 3 类 Span 的生命周期，提供跨进程/跨线程传播的基本方法



Next：第13讲《收集、发送 Trace 核心原理，Agent 与 OAP 的大动脉》

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