《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

— 拉勾教育出品 —

第23讲：深入剖析 register-receiver-plugin 插件（下）

IRegisterLockDAO原理分析

IRegisterLockDAO原理分析


```
@Stream(name = "service_inventory", scopeId = 14, ...)
public class ServiceInventory extends RegisterSource {...}
```

```
{
  "_index": "register_lock",
  "_type": "type",
  "_id": "14",
  "_version": 7,
  "_score": 1,
  "_source": {
    "sequence": 7
  }
}
```

```
@Stream(name ="service_instance_inventory", scopeId =15, ...)
public class ServiceInstanceInventory extends RegisterSource {...}
```

```
{
  "_index": "register_lock",
  "_type": "type",
  "_id": "15",
  "_version": 93,
  "_score": 1,
  "_source": {
    "sequence": 93
  }
}
```

IRegisterLockDAO原理分析


```
public int getId(int scopeId, RegisterSource registerSource) {
    String id = scopeId + ""; // Document Id
    int sequence = Const.NONE;
    //发送GetRequest请求，获取对应的Document
    GetResponse response = getClient().get("register_lock", id);
    if (response.isExists()) {
        Map<String, Object> source = response.getSource();
        // 获取sequence字段的值
        sequence = ((Number)source.get("sequence")).intValue();
        //获取ServiceInventory对应 Document的版本号
        long version = response.getVersion();
        sequence++; //递增sequence，即为该ServiceInventory分配的唯一ID
        lock(id, sequence, version); // 更新sequence字段值
    }
    return sequence; //更新成功，返回该sequence值
}
```

IRegisterLockDAO原理分析

在高版本的 ElasticSearch 中，不再推荐使用 version 的方式实现乐观锁

而是使用 _seq_no 和 _primary_term 两个字段来**实现乐观锁**

具体的实现方式与使用 version 的方式类似

IRegisterLockDAO原理分析

- **_primary_term**

  主要用于记录 Document 所在的主分片，每当主分片发生重新分配时

  比如重启、Primary 选举等，_primary_term 会递增 1

- **_seq_no**

  旧版本中的 _version 字段作用类似，是严格递增的顺序号

  每个 Document 在分片级别内对应一个，且严格递增

  以保证后写入的 Document 的 _seq_no 值大于先写入的 Document 的 _seq_no 值

IRegisterLockDAO原理分析

加上 _primary_term 这个字段可以提高并发的性能

但由于一个 Document 只会位于某一个特定的主分片中

所以由所在主分片分配序列号比之前通过 ElasticSearch 集群全局统一管理 _version 的性能会更高效

更多相关内容可参考：

https://github.com/elastic/elasticsearch/issues/19269#issuecomment-488598561

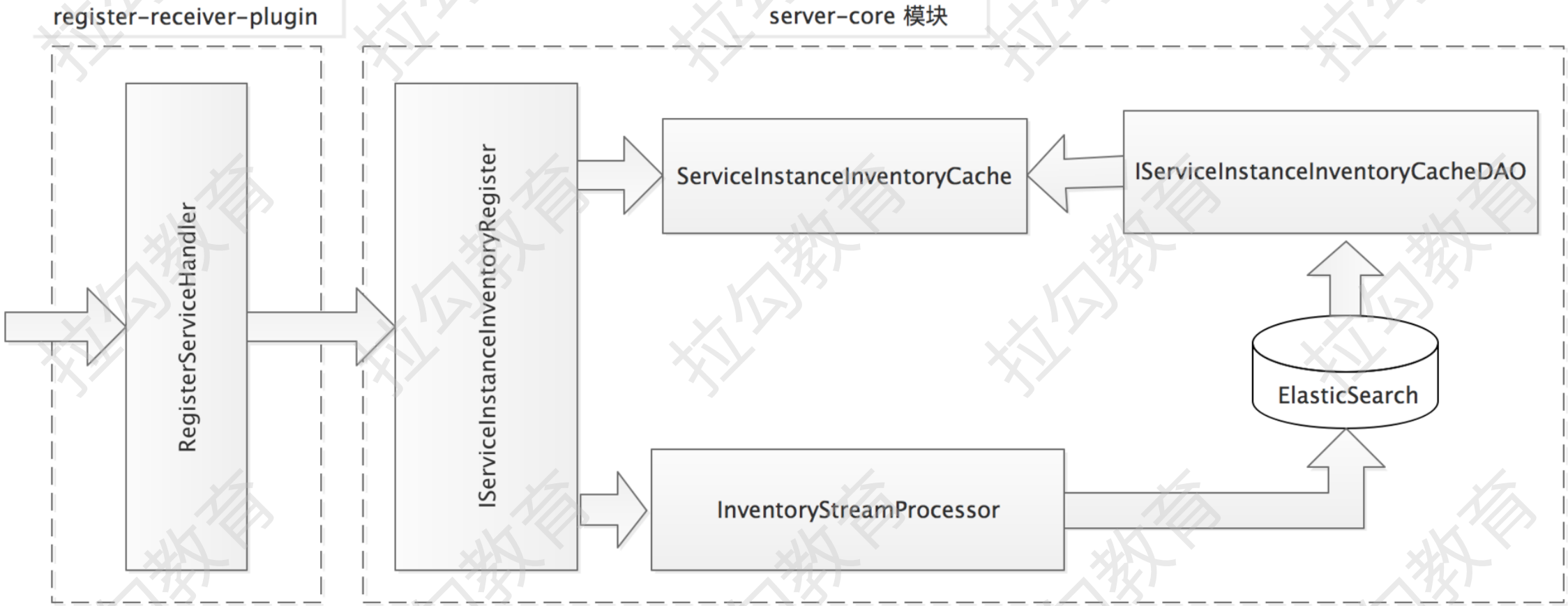
https://www.elastic.co/guide/en/elasticsearch/reference/7.x/optimistic-concurrency-control.html

IRegisterLockDAO原理分析

在一些极端情况下

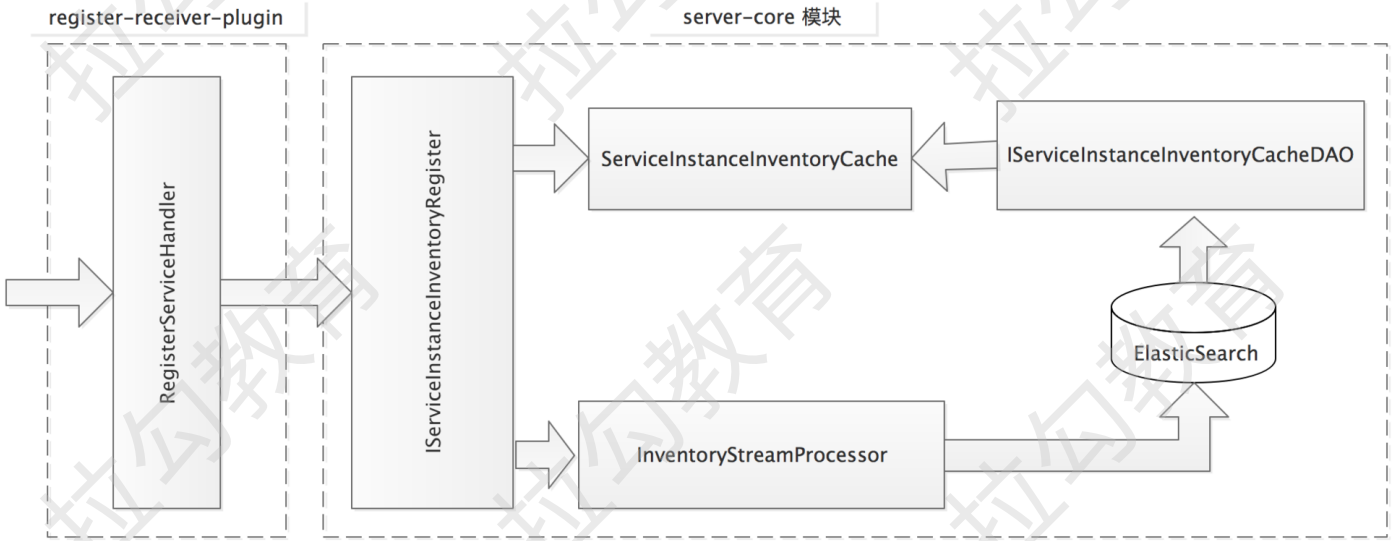
同一个服务会被 OAP 集群分配两个不同的ServiceId 吗？

服务实例注册



服务实例注册

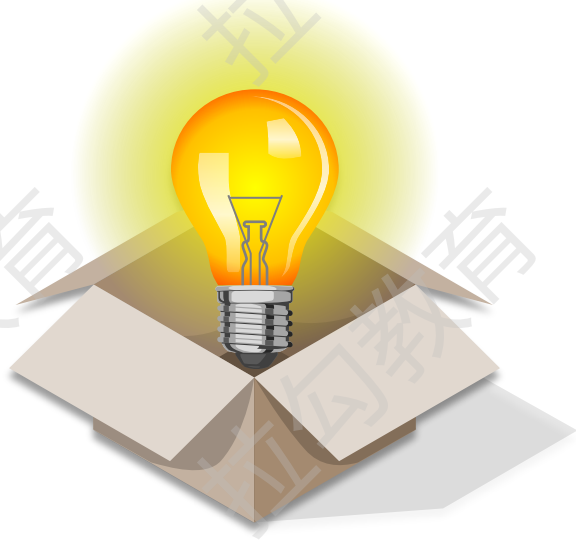
1. 根据请求中携带的 ServiceId，从 ServiceInventoryCache 中获取 Service 的相关信息
   主要获取的是服务名称，它将是构成服务实例名称的一部分
2. 处理请求携带的服务实例的附加信息，例如系统名称、HostName、IP、进程 ID 等
3. 生成服务实例名称，服务实例名称一般是由服务名称、进程 ID、HostName 三部分构成
4. 将上述服务实例信息交给 ServiceInstanceInventoryRegister 进行处理
5. 返回 ServiceInstance UUID 与 ServiceInstanceId 的映射关系



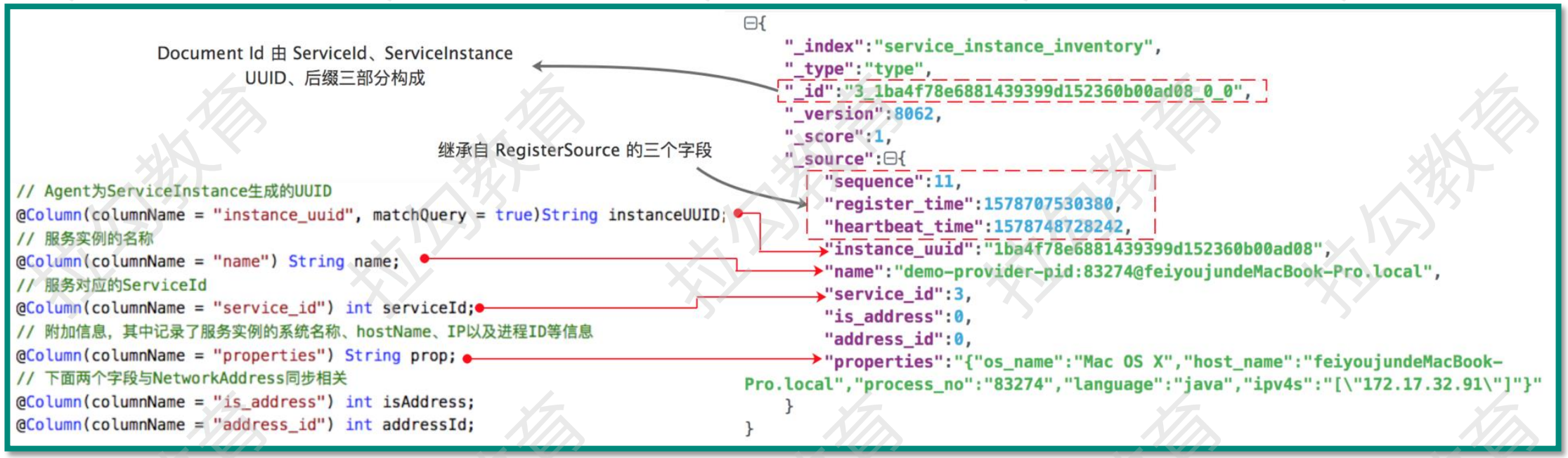
服务实例注册

ServiceInventoryRegister 核心逻辑基本一致：

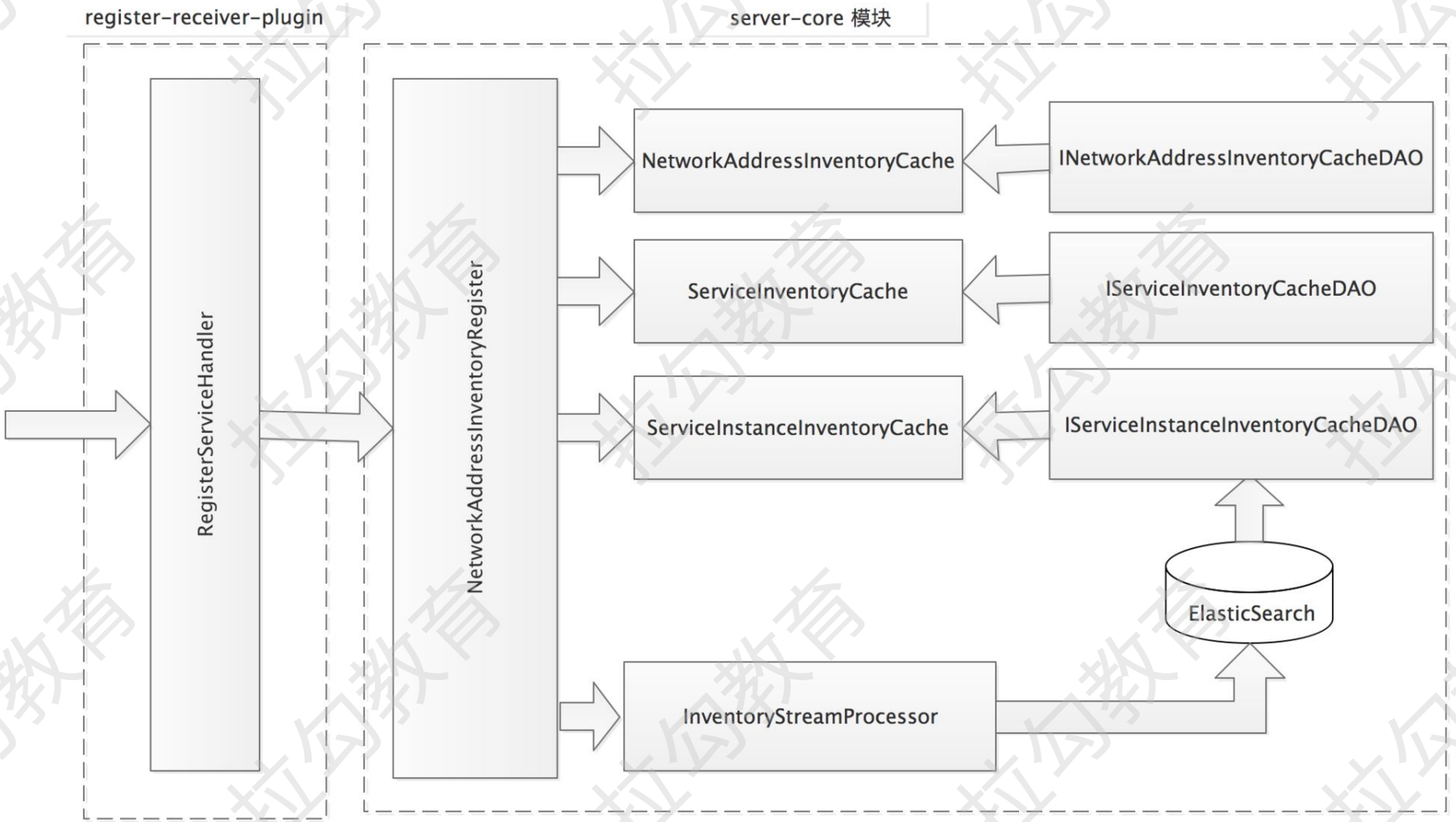
1. 先查询该 ServiceInstanceName 是否已分配了 ServiceInstanceId
   这里同样是先查缓存、缓存 miss ，再查底层持久化存储
2. 如果已分配 ServiceInstanceId，直接将其返回
3. 如果未分配 ServiceInstanceId，则将 ServiceInstance 相关信息封装成 ServiceInstanceInventory 对象交给 InventoryStreamProcessor 进行处理
4. 在 InventoryStreamProcessor 中会为 ServiceInstanceInventory 数据分配相应的 Worker 链以完成 L1 、L2 聚合以及持久化存储



服务实例注册


Document Id 由 ServiceId、ServiceInstance UUID、后缀三部分构成
继承自 RegisterSource 的三个字段
// Agent为ServiceInstance生成的UUID
@Column(columnName = "instance_uuid", matchQuery = true)String instanceUUID;
// 服务实例的名称
@Column(columnName = "name") String name;
// 服务对应的ServiceId
@Column(columnName = "service_id") int serviceId;
// 附加信息，其中记录了服务实例的系统名称、hostName、IP以及进程ID等信息
@Column(columnName = "properties") String prop;
// 下面两个字段与NetworkAddress同步相关
@Column(columnName = "is_address") int isAddress;
@Column(columnName = "address_id") int addressId;
{
"_index":"service_instance_inventory",
"_type":"type",
"_id":"3_1ba4f78e6881439399d152360b00ad08_0_0",
"_version":8062,
"_score":1,
"_source":{
"sequence":11,
"register_time":1578707530380,
"heartbeat_time":1578748728242,
"instance_uuid":"1ba4f78e6881439399d152360b00ad08",
"name":"demo-provider-pid:83274@feiyoujundeMacBook-Pro.local",
"service_id":3,
"is_address":0,
"address_id":0,
"properties":"{"os_name":"Mac OS X","host_name":"feiyoujundeMacBook-Pro.local","process_no":"83274","language":"java","ipv4s":"[\"172.17.32.91\"]"}"
}
}

NetWorkAddress、EndpointName 同步



NetWorkAddress、EndpointName 同步

NetworkAddressInventoryRegister.getOrCreate() 方法的处理步骤：

1. 查找指定 NetworkAddress 字符串在 network_address_inventory 索引中的对应 ID（addressId）

   查询时先查询 NetworkAddressInventoryCache 缓存，再查询底层的 ElasticSearch 索引

   若查找**失败**，会通过 InventoryStreamProcessor在 network_address_inventory 索引中为该 NetworkAddress 字符串生成相应 ID，此时getOrCreate() 方法返回 0

   若查找 addressId **成功**，继续执行步骤 2

NetWorkAddress、EndpointName 同步

NetworkAddressInventoryRegister.getOrCreate() 方法的处理步骤：

2. 根据步骤 1 得到的 addressId 以及 NetworkAddress 字符串

   在 service_inventory 索引中查找 NetworkAddress 与服务之间的绑定关系

   若查找**失败**，则通过 InventoryStreamProcessor 创建这个绑定关系

   若查询**成功**，则继续执行步骤 3

NetWorkAddress、EndpointName 同步

NetworkAddressInventoryRegister.getOrCreate() 方法的处理步骤：

3. 根据步骤 2 查询到的 ServiceId 以及 addressId

   在 service_instance_inventory 索引中查找该 NetworkAddress 与服务实例的绑定关系

   若查询**失败**，则由 InventoryStreamProcessor 创建该绑定关系

   若查询**成功**，则返回步骤 1 中得到的addressId

心跳请求

完成服务注册以及服务实例的注册之后

Agent 会定时调用 **ServiceInstancePing.doPing()** 这个 gRPC 接口发送心跳请求

以通知后端 OAP 集群当前 Agent 的在线状态

心跳请求


```
public void doPing(ServiceInstancePingPkg request,
    StreamObserver<Commands> responseObserver) {
  // 从心跳请求中获取 serviceInstanceId
  int serviceInstanceId = request.getServiceInstanceId();
  //心跳请求的发送时间
  long heartBeatTime = request.getTime();
  //更新服务实例的心跳时间(service_instance_inventory索引中相应Document
  //的heartbeat_time字段)
  serviceInstanceInventoryRegister.heartbeat(serviceInstanceId,
    heartBeatTime);
  ServiceInstanceInventory serviceInstanceInventory =
    serviceInstanceInventoryCache.get(serviceInstanceId);
  if (Objects.nonNull(serviceInstanceInventory)) {
```

心跳请求


```
ServiceInstanceInventory serviceInstanceInventory =
    serviceInstanceInventoryCache.get(serviceInstanceId);
if (Objects.nonNull(serviceInstanceInventory)) {
    //更新相应服务的心跳时间(service_inventory索引中相应Document的
    // heartbeat_time字段)
    serviceInventoryRegister.heartbeat(
     serviceInstanceInventory.getServiceId(), heartBeatTime);
} else {
    logger.warn("...", serviceInstanceId);
}
responseObserver.onNext(Commands.getDefaultInstance());
responseObserver.onCompleted();
}
```

Next：第24讲《jvm-receiver 插件探秘，不仅有 Trace 还可以有监控》

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