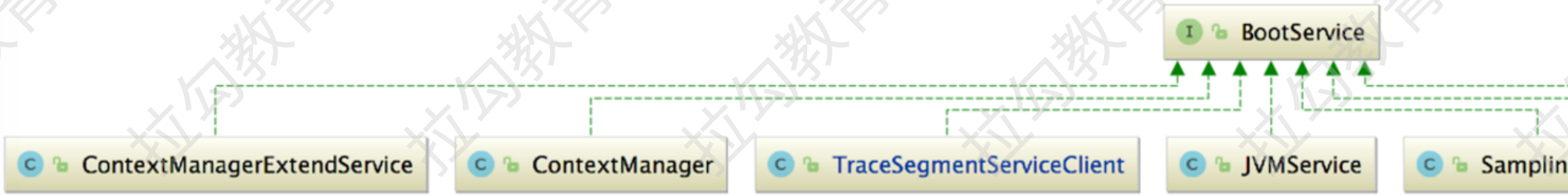
《31 讲带你搞懂 SkyWalking》

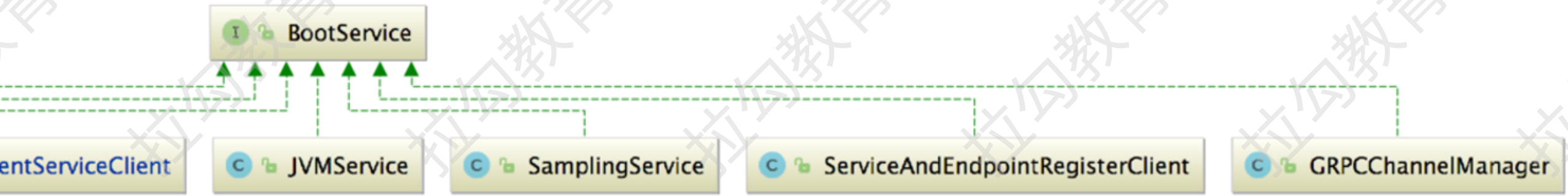
徐郡明　前搜狗资深技术专家、源码剖析系列畅销书作者

第11讲：BootService 核心实现解析
Agent 的“地基”原来是这样的

课程回顾



课程回顾



网络连接管理

gRPC 的两个组件：

- ManagedChannel

  是 gRPC 客户端的核心类之一

  逻辑上表示一个 Channel，底层持有一个 TCP 链接，并负责维护此连接的活性

  通常情况下不需要在 RPC 调用结束后就关闭 Channel ，该 Channel 可以被一直重用

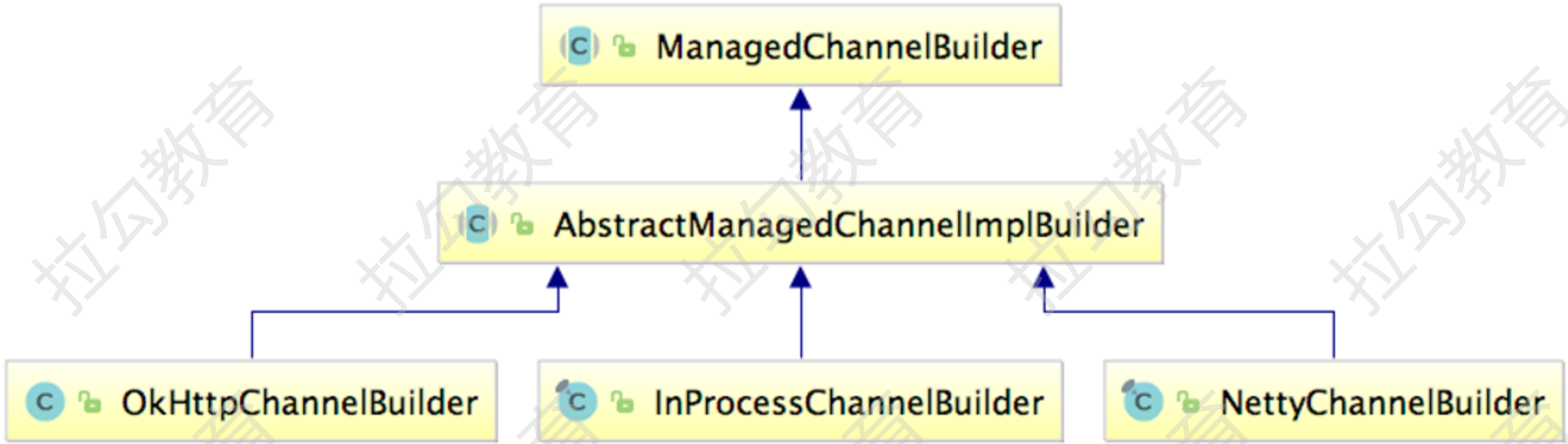
  直到整个客户端程序关闭

- ManagedChannelBuilder

  负责创建客户端 Channel，使用 provider 机制，具体是创建了哪种 Channel 由 provider 决定

  常用的 ManagedChannelBuilder 有三种：NettyChannelBuilder、OkHttpChannelBuilder、InProcessChannelBuilder

网络连接管理



网络连接管理

SkyWalking Agent 中用的是 NettyChannelBuilder，其创建的 Channel 底层是基于 Netty 实现的

OkHttpChannelBuilder 创建的 Channel 底层是基于 OkHttp 库实现的

InProcessChannelBuilder 用于创建进程内通信使用的 Channel

网络连接管理


```
// 封装了上面介绍的gRPC Channel
private volatile GRPCChannel managedChannel = null;
// 定时检查 GRPCChannel的连接状态重连gRPC Server的定时任务
private volatile ScheduledFuture<?> connectCheckFuture;
//是否重连。当GRPCChannel断开时会标记reconnect为 true后台线程会根据该标
//识决定是否进行重连
private volatile boolean reconnect = true;
// 加在 Channel上的监听器，主要是监听 Channel的状态变化
private List<GRPCChannelListener> listeners;
// 可选的 gRPC Server集合，即后端OAP集群中各个OAP实例的地址
private volatile List<String> grpcServers;
```

网络连接管理

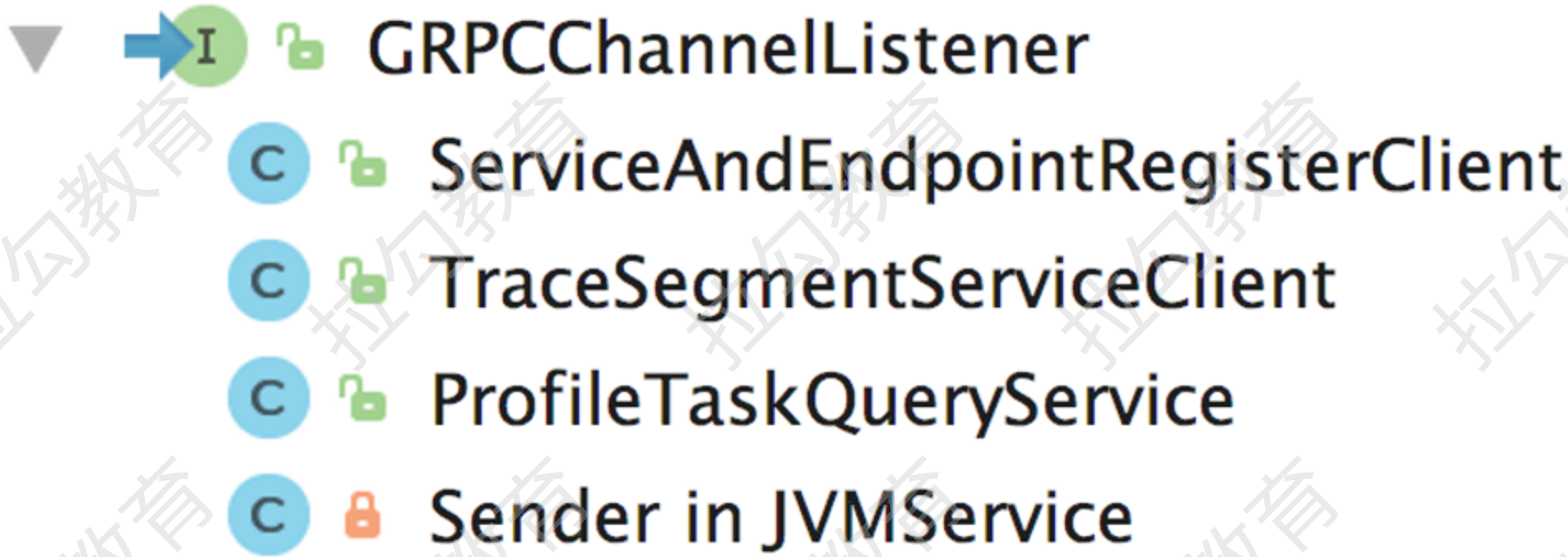
Agent 启动过程中会依次调用

BootService 实现的 prepare() 方法 → boot() 方法 → onComplete() 方法之后

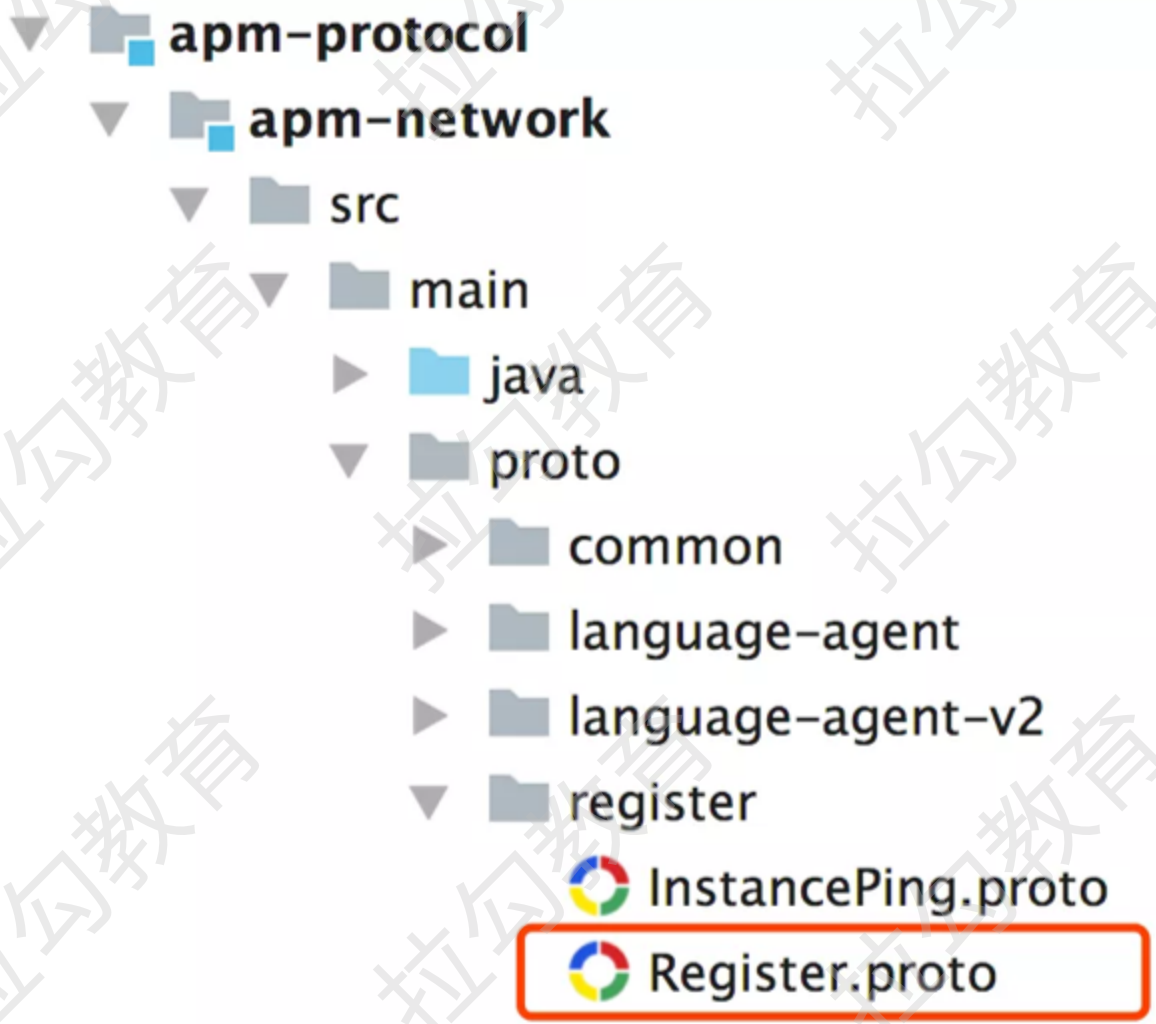
才能真正对外提供服务

网络连接管理

```
public void run() {
    if (reconnect && grpcServers.size() > 0) {
        //根据配置，连接指定OAP实例的IP和端口
        managedChannel = GRPCChannel.newBuilder(ipAndPort[0],
                Integer.parseInt(ipAndPort[1]))
            .addManagedChannelBuilder(new StandardChannelBuilder())
            .addManagedChannelBuilder(new TLSChannelBuilder())
            .addChannelDecorator(new AuthenticationDecorator())
            .build();
        // notify()方法会循环调用所有注册在当前连接上的
GRPCChannelListener实
        //例(记录在listeners集合中)的statusChanged()方法，通知它们连接
创建
        //成功的事件
        notify(GRPCChannelStatus.CONNECTED);
        //设置reconnect字段为false，暂时不会再重建连接了
        reconnect = false;
    }
}
```



注册协议及实现



注册协议及实现

```
service Register {
    rpc doServiceRegister(Services)returns(ServiceRegisterMapping)
    {}

    rpc doServiceInstanceRegister (ServiceInstances) returns
        (ServiceInstanceRegisterMapping) {}

    #还有三个方法，这里省略一下，后面会介绍
}
```

注册协议及实现

与 Service 注册流程相关的 BootService 实现是 ServiceAndEndpointRegisterClient

ServiceAndEndpointRegisterClient 实现了 GRPCChannelListener 接口

在其 prepare() 方法中首先会将其注册到 GRPCChannelManager 来监听网络连接

然后生成当前 ServiceInstance 的唯一标识

注册协议及实现

```
public void prepare() throws Throwable {
    //查找 GRPCChannelManager实例(前面介的ServiceManager.bootedServices
    //集合会按照类型维护BootService实例，查找也是查找该集合)，然后将
    // ServiceAndEndpointRegisterClient注册成Listener
    ServiceManager.INSTANCE.findService(GRPCChannelManager.class)
            .addChannelListener(this);
    //确定INSTANCE_UUID，优先使用gent.config文件中配置的INSTANCE_UUID，
    //若未配置则随机生成
    INSTANCE_UUID = StringUtil.isEmpty(Config.Agent.INSTANCE_UUID) ?
            UUID.randomUUID().toString().replaceAll("-", "") :
            Config.Agent.INSTANCE_UUID;
}
```

注册协议及实现


```
public void statusChanged(GRPCChannelStatus status) {
    if (GRPCChannelStatus.CONNECTED.equals(status)) {
        //网络连接创建成功时，会依赖该连接创建两个stub客户端
        Channel channel = ServiceManager.INSTANCE.findService(
                GRPCChannelManager.class).getChannel();
        registerBlockingStub =
RegisterGrpc.newBlockingStub(channel);
        serviceInstancePingStub = ServiceInstancePingGrpc
                .newBlockingStub(channel);
    } else { //网络连接断开时，更新两个stub字段（它们都是volatile修
饰）
        registerBlockingStub = null;
        serviceInstancePingStub = null;
    }
    this.status = status; //更新status字段，记录网络状态
}
```

注册协议及实现


```
while (GRPCChannelStatus.CONNECTED.equals(status) && shouldTry) {
    shouldTry = false;
    //检测当前Agent是否已完成了Service注册
    if (RemoteDownstreamConfig.Agent.SERVICE_ID ==
            DictionaryUtil.nullValue()) {
        if (registerBlockingStub != null) {//第二次检查网络状态
            //通过doServiceRegister()接口进行Service注册
            ServiceRegisterMapping serviceRegisterMapping =
                registerBlockingStub.doServiceRegister(
                    Services.newBuilder().addServices(Service.newBuilder()
                        .setServiceName(Config.Agent.SERVICE_NAME).build());
            for (KeyIntValuePair registered :
```

注册协议及实现

```
        for (KeyIntValuePair registered :
          serviceRegisterMapping.getServicesList()) {//遍历所有KV
          if (Config.Agent.SERVICE_NAME
                .equals(registered.getKey())){
            RemoteDownstreamConfig.Agent.SERVICE_ID =
                registered.getValue(); //记录serviceId
            //设置shouldTry，紧跟着会执行服务实例注册
            shouldTry = true;
          }
        }
      }
    } else {//后续会执行服务实例注册以及心跳操作
```

注册协议及实现

```
                .equals(registered.getKey())) {
            RemoteDownstreamConfig.Agent.SERVICE_ID =
                registered.getValue(); //记录serviceId
            //设置shouldTry，紧跟着会执行服务实例注册
            shouldTry = true;
        }
    }
}
} else { //后续会执行服务实例注册以及心跳操作
    ... ...
}
}
```

注册协议及实现

拉勾教育

- 如果在 agent.config 配置文件中直接配置了 serviceId 是无需进行服务注册的
- ServiceAndEndpointRegisterClient 会根据监听 GRPCChannel 的连接状态

  决定是否发送服务注册请求、服务实例注册请求以及心跳请求

注册协议及实现

```
while (GRPCChannelStatus.CONNECTED.equals(status) && shouldTry) {
    if (RemoteDownstreamConfig.Agent.SERVICE_ID ==
            DictionaryUtil.nullValue()) {
        ... //省略服务注册逻辑
    } else {
        if (RemoteDownstreamConfig.Agent.SERVICE_INSTANCE_ID ==
                DictionaryUtil.nullValue()) {
            //调用 doServiceInstanceRegister()接口，用serviceId和
            // INSTANCE_UUID换取SERVICE_INSTANCE_ID
            ServiceInstanceRegisterMapping instanceMapping =
                registerBlockingStub.doServiceInstanceRegister(
                ServiceInstances.newBuilder()
                .addInstances(ServiceInstance.newBuilder()
                .setServiceId(RemoteDownstreamConfig.Agent.SERVICE_ID)
                //除了serviceId，还会传递uuid、时间戳以及系统信息之类的
                .setInstanceUUID(INSTANCE_UUID)
```

注册协议及实现

```
            .setInstanceUUID(INSTANCE_UUID)
            .setTime(System.currentTimeMillis())
            .addAllProperties(OSUtil.buildOSInfo())).build());
        for (KeyIntValuePair serviceInstance :
            instanceMapping.getServiceInstancesList()) {
          if (INSTANCE_UUID.equals(serviceInstance.getKey())){
            //记录serviceIntanceId
            RemoteDownstreamConfig.Agent.SERVICE_INSTANCE_ID =
              serviceInstance.getValue();
          }
        }
      }else{
        ... //省略心跳的相关逻辑
      }
    }
  }
```

心跳协议及实现

心跳涉及一个新的 gRPC 接口如下（定义在 InstancePing.proto 文件中）

```
service ServiceInstancePing {

    rpc doPing (ServiceInstancePingPkg) returns (Commands) {}

}
```

心跳协议及实现

```
while (GRPCChannelStatus.CONNECTED.equals(status) && shouldTry) {
    if (Agent.SERVICE_ID == DictionaryUtil.nullValue()) {
        ... //省略服务注册逻辑
    } else {
        if (Agent.SERVICE_INSTANCE_ID == DictionaryUtil.nullValue())
{
            ... //省略服务实例注册逻辑
        }else{ //并没有对心跳请求的响应做处理
            serviceInstancePingStub.doPing(ServiceInstancePingPkg
                .newBuilder().setServiceInstanceId(SERVICE_INSTANCE_ID)
                .setTime(System.currentTimeMillis())
                .setServiceInstanceUUID(INSTANCE_UUID).build());
        }
    }
}
```

Endpoint、NetWorkAddress 同步

Trace 数据中包含：

请求的 URL 地址、RPC 接口名称、HTTP 服务或 RPC 服务的地址、数据库的 IP 以及端口等信息

在海量 Trace 中包含这些重复的字符串，会非常浪费网络带宽以及存储资源

常见的解决方案：

将字符串映射成数字编号并维护一张映射表

在传输、存储时使用映射后的数字编号，在展示时根据映射表查询真正的字符串进行展示即可

Endpoint、NetWorkAddress 同步

SkyWalking 中有两个 DictionaryManager：

- EndpointNameDictionary：用于同步 Endpoint 字符串的映射关系
- NetworkAddressDictionary：用于同步网络地址的映射关系

EndpointNameDictionary 中维护了两个集合：

- endpointDictionary：记录已知的 Endpoint 名称映射的数字编号
- unRegisterEndpoints：记录了未知的 Endpoint 名称

Endpoint、NetWorkAddress 同步

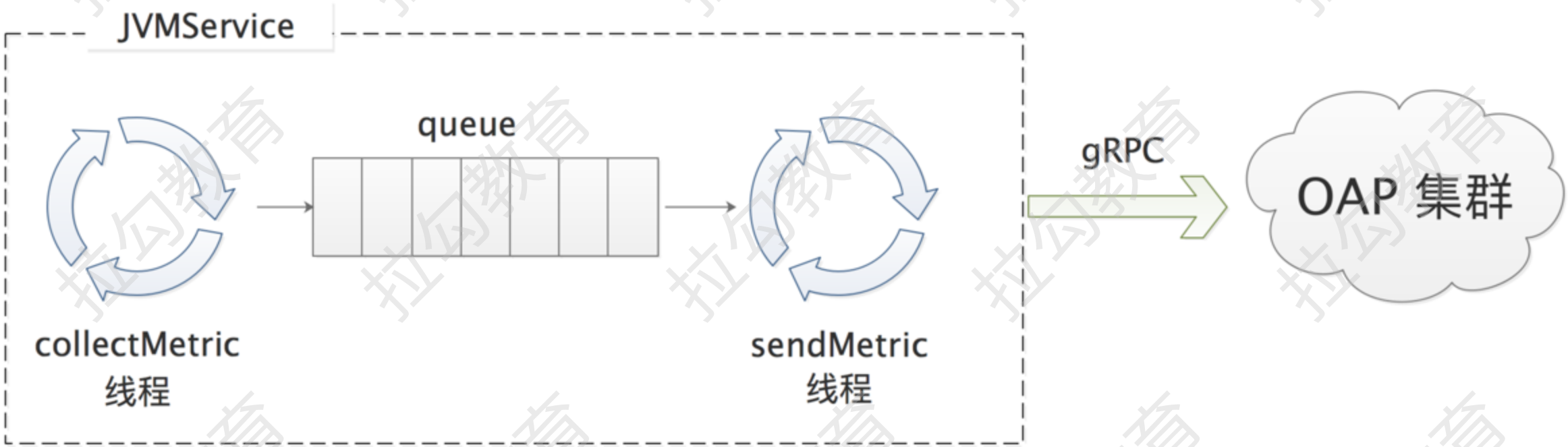
拉勾教育

```
public void syncRemoteDictionary(RegisterGrpc.RegisterBlockingStub
        serviceNameDiscoveryServiceBlockingStub) {
    //创建请求，每个Endpoint中都封装了 Endpoint名称以及关联的serviceId
    Endpoints.Builder builder = Endpoints.newBuilder();
    for (OperationNameKey operationNameKey : unRegisterEndpoints) {
      Endpoint endpoint = Endpoint.newBuilder()
          .setServiceId(operationNameKey.getServiceId())
          .setEndpointName(operationNameKey.getEndpointName())
          .setFrom(operationNameKey.getSpanType())
          .build();
      builder.addEndpoints(endpoint);
    }
    //发送同步请求
    EndpointMapping serviceNameMappingCollection =
      serviceNameDiscoveryServiceBlockingStub
```

Endpoint、NetWorkAddress 同步

```
EndpointMapping serviceNameMappingCollection =
  serviceNameDiscoveryServiceBlockingStub
    .doEndpointRegister(builder.build());
for (EndpointMappingElement element :
  serviceNameMappingCollection.getElementsList()) {
  //将返回的映射关系，记录到 endpointDictionary集合中，并从
  //unRegisterEndpoints集合中删除Endpoint信息
  OperationNameKey key = new OperationNameKey(
    element.getServiceId(), element.getEndpointName(),
    DetectPoint.server.equals(element.getFrom()),
    DetectPoint.client.equals(element.getFrom()));
  unRegisterEndpoints.remove(key);
  endpointDictionary.put(key, element.getEndpointId());
 }
}
```

收集 JVM 监控



收集 JVM 监控


```
public void prepare() throws Throwable {
    queue = new LinkedBlockingQueue(Config.Jvm.BUFFER_SIZE);
    sender = new Sender();
    ServiceManager.INSTANCE.findService(GRPCChannelManager.class)
            .addChannelListener(sender); // sender会监听底层的连接状态
}
```

收集 JVM 监控

```
JVMMetricCollection.Builder builder =
                JVMMetricCollection.newBuilder();
LinkedList<JVMMetric> buffer = new LinkedList<JVMMetric>();
//将 queue队列中缓存的全部监控数据填充到 buffer中
queue.drainTo(buffer);
//创建 gRPC请求参数
builder.addAllMetrics(buffer);
builder.setServiceInstanceId(Agent.SERVICE_INSTANCE_ID); //
//通过 gRPC调用将JVM监控数据发送到后端 OAP集群
stub.collect(builder.build());
```

收集 JVM 监控

gRPC 接口以及参数的定义

```
service JVMMetricReportService {
    rpc collect (JVMMetricCollection) returns (Commands)
{}
}
```

收集 JVM 监控

```
//通过JMX获取CPU、Memory、GC的信息，然后组装成JVMMetric
JVMMetric.Builder jvmBuilder = JVMMetric.newBuilder();
jvmBuilder.setTime(currentTimeMillis);
//通过 MXBean获取 CPU、内存以及GC相关的信息，并填充到 JVMMetric
jvmBuilder.setCpu(CPUProvider.INSTANCE.getCpuMetric());
jvmBuilder.addAllMemory(
    MemoryProvider.INSTANCE.getMemoryMetricList());
jvmBuilder.addAllMemoryPool(
    MemoryPoolProvider.INSTANCE.getMemoryPoolMetricsList());
jvmBuilder.addAllGc(GCProvider.INSTANCE.getGCList());
JVMMetric jvmMetric = jvmBuilder.build();
```

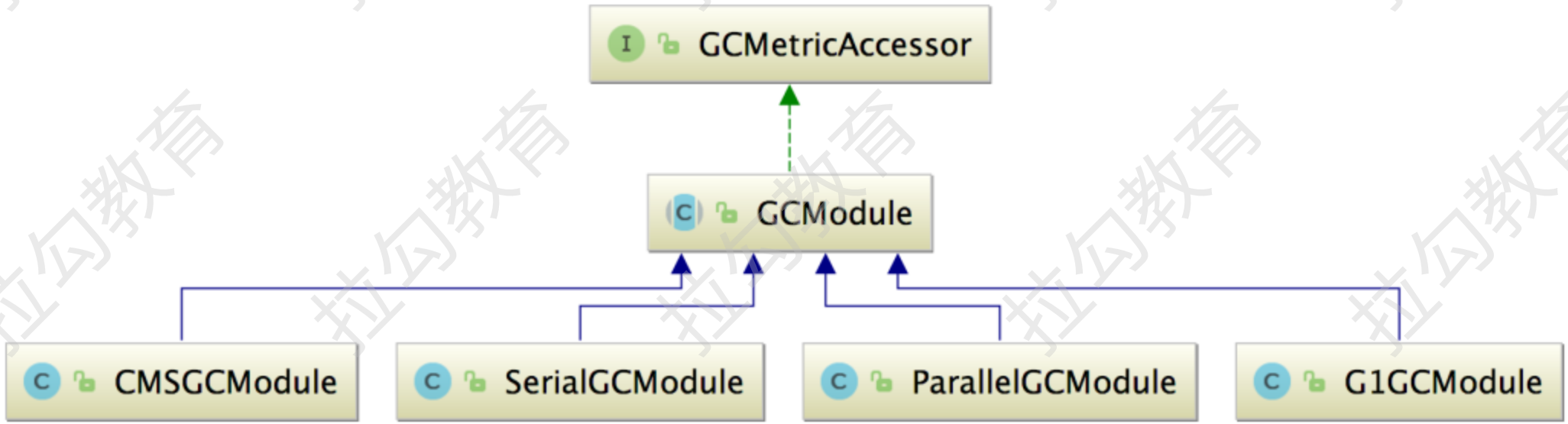
收集 JVM 监控


```
jvmBuilder.addAllMemory(
    MemoryProvider.INSTANCE.getMemoryMetricList());
jvmBuilder.addAllMemoryPool(
    MemoryPoolProvider.INSTANCE.getMemoryPoolMetricsList());
jvmBuilder.addAllGc(GCProvider.INSTANCE.getGCList());
JVMMetric jvmMetric = jvmBuilder.build();
//将JVMMetric写入到queue缓冲队列中
if (!queue.offer(jvmMetric)) { // queue缓冲队列的长度默认为600
    queue.poll(); //如果queue队列被填满，则抛弃最老的监控信息，保留最新的
    queue.offer(jvmMetric);
}
```

收集 JVM 监控


```
//获取GC相关的MXBean
beans = ManagementFactory.getGarbageCollectorMXBeans();
for (GarbageCollectorMXBean bean : beans) {
    String name = bean.getName();
    //解析MXBean的名称即可得知当前使用的是哪种垃圾收集器，我们就可以创建相应
    //的GCMetricAccessor实现
    GCMetricAccessor accessor = findByBeanName(name);
    if (accessor != null) {
        metricAccessor = accessor;
        break;
    }
}
```

收集 JVM 监控



收集 JVM 监控


```
public List<GC> getGCList() {
    List<GC> gcList = new LinkedList<GC>();
    for (GarbageCollectorMXBean bean : beans) {
        String name = bean.getName();
        GCPhrase phrase;
        long gcCount = 0;
        long gcTime = 0;
        //下面根据 MXBean的名称判断具体的GC信息
        if (name.equals(getNewGCName())) { // Young GC的信息
            phrase = GCPhrase.NEW;
            //计算GC次数，从MXBean直接拿到的是GC总次数
```

收集 JVM 监控

```
        //计算GC次数，从MXBean直接拿到的是GC总次数
        long collectionCount = bean.getCollectionCount();
        gcCount = collectionCount - lastYGCCount;
        lastYGCCount = collectionCount; // ? lastYGCCount
        //计算GC时间，从 MXBean直接拿到的是GC总时间
        long time = bean.getCollectionTime();
        gcTime = time - lastYGCCollectionTime;
        lastYGCCollectionTime = time; //更新lastYGCCollectionTime
    } else if (name.equals(getOldGCName())) { // Old GC的信息
        phrase = GCPhrase.OLD;
        ... ... //Old GC的计算方式与Young GC的计算方式相同，不再重复
```

收集 JVM 监控

```
            lastYGCCollectionTime = time; //更新lastYGCCollectionTime
        } else if (name.equals(getOldGCName())) { // Old GC的信息
            phrase = GCPhrase.OLD;
            ... ... // Old GC的计算方式与Young GC的计算方式相同，不再重复
        }
        gcList.add( //最后将 GC信息封装成List返回
            GC.newBuilder().setPhrase(phrase)
                .setCount(gcCount).setTime(gcTime).build()
        );
    }
    return gcList;
}
```

收集 JVM 监控

在不同类型的垃圾收集器中

Young GC、Old GC 的名称不太相同

例如:

G1 中叫 G1 Young Generation 和 G1 Old Generation

CMS 中则叫 ParNew 和 ConcurrentMarkSweep

其他

- ContextManager：负责管理一个 SkyWalking Agent 中所有的 Context 对象
- ContextManagerExtendService：负责创建 Context 对象
- TraceSegmentServiceClient：负责将 Trace 数据序列化并发送到 OAP 集群
- SamplingService：负责实现 Trace 的采样

总结

GRPCChannelManager 负责管理 Agent 到 OAP 集群的网络连接，并实时的通知注册的 Listener

1. 注册功能：其中包括服务注册和服务实例注册两次请求
2. 定期发送心跳请求：与后端 OAP 集群维持定期探活，让后端 OAP 集群知道该 Agent 正常在线
3. 定期同步 Endpoint 名称以及网络地址：维护当前 Agent 中字符串与数字编号的映射关系

   减少后续 Trace 数据传输的网络压力，提高请求的有效负载

Next：第12讲《剖析 Trace 在 SkyWalking 中的落地实现方案》

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