H3C 高端路由器 Telemetry 二次开发指南

前 言

本文档主用来指导用户使用 Telemetry 技术配置和管理 Comware 设备。

本文档适用于 SR8800-X、SR8800-X-S、SR8800-F、CR16000-F、CR16000-M、RX8800 路由器。

前言部分包含如下内容：

读者对象

本手册主要适用于如下工程师：

- 熟悉 gRPC、GPB 编码的开发人员
- 熟悉对应编程语言（C++、JAVA、Python、GO）的开发人员
- 负责网络配置和维护，且具有一定 XML 和 NETCONF 技术基础的网络管理员

本书约定

命令行格式约定

格　式	意　义
粗体	命令行关键字（命令中保持不变、必须照输的部分）采用**加粗**字体表示。
斜体	命令行参数（命令中必须由实际值进行替代的部分）采用*斜体*表示。
[ ]	表示用“[ ]”括起来的部分在命令配置时是可选的。
{ x | y | ... }	表示从多个选项中仅选取一个。
[x | y | ...]	表示从多个选项中选取一个或者不选。
{ x | y | ... } *	表示从多个选项中至少选取一个。
[x | y | ...] *	表示从多个选项中选取一个、多个或者不选。
&<1-n>	表示符号&前面的参数可以重复输入1～n次。
#	由“#”号开始的行表示为注释行。

各类标志

本书还采用各种醒目标志来表示在操作过程中应该特别注意的地方，这些标志的意义如下：

 警告	该标志后的注释需给予格外关注，不当的操作可能会对人身造成伤害。
 注意	提醒操作中应注意的事项，不当的操作可能会导致数据丢失或者设备损坏。
 提示	为确保设备配置成功或者正常工作而需要特别关注的操作或信息。
 说明	对操作内容的描述进行必要的补充和说明。
 窍门	配置、操作、或使用设备的技巧、小窍门。

图标约定

本书使用的图标及其含义如下：

	该图标及其相关描述文字代表一般网络设备，如路由器、交换机、防火墙等。
	该图标及其相关描述文字代表一般意义下的路由器，以及其他运行了路由协议的设备。
	该图标及其相关描述文字代表二、三层以太网交换机，以及运行了二层协议的设备。
	该图标及其相关描述文字代表无线控制器、无线控制器业务板和有线无线一体化交换机的无线控制引擎设备。
	该图标及其相关描述文字代表无线接入点设备。
	该图标及其相关描述文字代表无线终结单元。
	该图标及其相关描述文字代表无线终结者。
	该图标及其相关描述文字代表无线Mesh设备。
	该图标代表发散的无线射频信号。
	该图标代表点到点的无线射频信号。
	该图标及其相关描述文字代表防火墙、UTM、多业务安全网关、负载均衡等安全设备。
	该图标及其相关描述文字代表防火墙插卡、负载均衡插卡、NetStream插卡、SSL VPN插卡、IPS插卡、ACG插卡等安全插卡。

示例约定

由于设备型号不同、配置不同、版本升级等原因，可能造成本手册中的内容与用户使用的设备显示信息不一致。实际使用中请以设备显示的内容为准。

本手册中出现的端口编号仅作示例，并不代表设备上实际具有此编号的端口，实际使用中请以设备上存在的端口编号为准。

资料意见反馈

如果您在使用过程中发现产品资料的任何问题，可以通过以下方式反馈：

E-mail：info@h3c.com

感谢您的反馈，让我们做得更好！

目 录

1 概述

1.1 Telemetry简介

Telemetry 通常指基于数据模型的 Telemetry (Model-Driven Telemetry)，它提供了一种高速的、大规模远程数据采集机制，可用于整网设备的实时性能监控。

随着网络的普及和新技术的涌现，网络规模日益增大，部署的复杂度逐步提升，用户对业务的质量要求也不断提高。为了满足用户需求，网络运维务必更加精细化、智能化。当今网络的运维面临着如下挑战：

- 超大规模：管理的设备数目众多，监控的信息数量非常庞大。
- 快速定位：在复杂的网络中，能够快速地定位故障，达到秒级、甚至亚秒级的故障定位速度。
- 精细监控：监控的数据类型更加丰富，监控粒度更细，以便完整、准确地反应网络状况，据此预估可能发生的故障，并为网络优化提供有力的数据依据。实际网络运维中，不仅需要监控接口的流量统计信息、每条流的丢包情况、CPU 和内存占用情况，还需要监控每条流的时延抖动、每个报文在传输路径上的时延、每台设备的缓冲区占用情况等。

传统的网络监控手段（SNMP、CLI、日志）已经无法满足网络需求：

- SNMP 和 CLI 方式主要采用“拉模式”获取数据，即发送请求来获取设备上的数据，限制了可以监控的网络设备数量，且无法快速获取数据。
- SNMP Trap 和日志方式虽然采用“推模式”获取数据，即设备主动将数据上报给监控设备，但仅上报事件和告警，监控的数据内容极其有限，无法准确地反映网络状况。

因此，面对大规模、高性能的网络监控需求，用户需要一种新的网络监控方式。Telemetry 技术可以满足用户的要求，实现一种智能的运维系统：

- 相对于传统的“拉模式”，网络设备采用“推模式”，周期性地主动向采集器上送丰富的监控数据，提供主动高效的数据采集功能，为网络问题的快速定位、网络质量优化调整提供了重要的大数据基础。
- 利用 Telemetry 技术，采集器可以收集到大量的设备数据，然后将数据交给分析器进行大数据分析，分析器再将分析结果上报给控制器，由控制器调整设备的配置，便能够满足更加实时且高精度的智能运维需求。

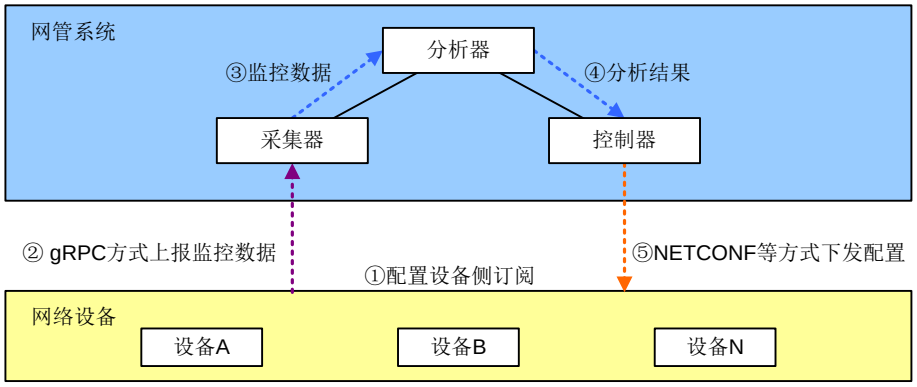
1.2 Telemetry网络模型

如图 1-1 所示，Telemetry 网络模型中包括如下组成部分：

- 网络设备：接受监控的设备。通过在网络设备上配置订阅数据源，网络设备将依据订阅要求对指定的监控数据进行采样，并将采样数据通过 gRPC（Google Remote Procedure Call，Google 远程过程调用）方式定时上送给采集器。
- 采集器：用于接收和保存网络设备上报的监控数据。
- 分析器：用于分析采集器接收到的监控数据，并对数据进行处理，例如以图形化界面的形式展现给用户。

- 控制器：通过 NETCONF 等方式向设备下发配置，实现对网络设备的管理。控制器可以根据分析器提供的分析结果，为网络设备下发配置，对网络设备的转发行为进行调整，也可以控制网络设备对哪些数据进行采样和上报。

图1-1 Telemetry 网络模型



2 基于 gRPC 的 Telemetry 技术介绍

H3C 的 Telemetry 技术通过 gRPC 协议将监控数据从设备推送给采集器。网络设备和网管系统建立 gRPC 连接后，设备将自动读取各种统计信息（CPU、内存、接口等），根据采集器的订阅要求将监控数据通过 gRPC 协议上报给采集器。

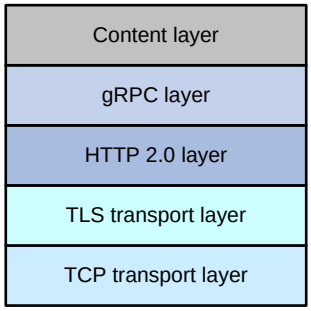
2.1 gRPC协议

gRPC（Google Remote Procedure Call，Google 远程过程调用）是 Google 发布的基于 HTTP 2.0 协议承载的高性能开源软件框架，提供了支持多种编程语言、对网络设备进行配置和管理的方法。通信双方可以基于该软件框架进行二次开发。

2.1.1 gRPC 协议栈分层

gRPC 协议栈分层如图 2-1 所示。

图2-1 RPC 协议栈分层



自上而下，gRPC 协议栈各层含义如表 2-1 所示。

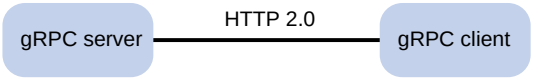
表2-1 gRPC 协议栈分层

分层	说明
内容层	用于承载编码后的业务数据。业务数据的编码格式包括： • GPB（Google Protocol Buffer）：高效的二进制编码格式，通过 Proto 文件描述编码使用的数据结构。在设备和采集器之间传输数据时，该编码格式的数据比其他格式（如 JSON）的数据具有更高的信息负载能力。业务数据使用 GPB 格式编码时，需要配合对应的业务模块 Proto 文件才能解码。 • JSON（JavaScript Object Notation）：轻量级的数据交换格式，采用独立于编程语言的文本格式来存储和表示数据，易于阅读和编写。业务数据使用 JSON 格式编码时，通过公共 Proto 文件即可解码，无需对应的业务模块 Proto 文件。 设备和采集器通信时，双方的Proto文件必须保持一致才能解码
gRPC层	定义了RPC（Remote Procedure Call，远程过程调用）的协议交互格式。公共RPC方法定义在公共Proto文件中，例如grpc_dialout.proto
HTTP 2.0层	当传输协议为TCP时，gRPC承载在HTTP 2.0协议上。HTTP 2.0协议具有首部数据压缩、单TCP连接支持多路请求、流量控制等性能增强特性
TLS（Transport Layer Security，传输层安全）层	该层是可选的，设备和采集器可以基于TLS协议进行通道加密和双向证书认证，实现安全通信
传输层	底层通信协议，支持TCP传输层协议：提供面向连接的、可靠的数据链路

2.1.2 gRPC 网络架构

如图 2-2 所示，gRPC 网络采用客户端/服务器模型，使用 HTTP 2.0 协议传输报文。

图2-2 gRPC 网络架构



gRPC 网络的工作机制如下：

(1) gRPC 服务器通过监听指定服务端口等待 gRPC 客户端的连接请求。

(2) 用户通过执行 gRPC 客户端程序登录到 gRPC 服务器。

(3) gRPC 客户端调用 Proto 文件提供的 gRPC 方法发送请求消息。关于 Proto 文件的详细介绍请参见“2.5 Proto 文件”。

(4) gRPC 服务器回复应答消息。

说明

H3C 设备支持作为 gRPC 服务器或者 gRPC 客户端。

2.1.3 gRPC 的对接模式

在 gRPC 网络中，根据设备和采集器的角色不同，支持 Dial-in 和 Dial-out 两种 gRPC 对接模式。

1. Dial-in 模式

Dial-in 模式下，设备作为 gRPC 服务器，采集器作为 gRPC 客户端，由采集器主动向设备发起 gRPC 连接并订阅需要采集的数据信息。该模式适用于小规模网络以及采集器需要向设备下发配置的场景。

Dial-in 模式支持以下操作：

- 普通 Subscribe 操作：用于向设备订阅事件触发类数据。所需的公共 RPC 方法在 grpc_server.proto 文件中定义，同时还需要各业务模块的 Proto 文件定义对应业务的 RPC 方法。
- gNMI Subscribe 操作：基于 gNMI 协议，用于向设备订阅数据推送服务，包括事件触发类数据和周期采样类数据。所需的公共 RPC 方法在 grpc_server.proto 文件中定义，同时还需要 gnmi.proto 和 gnmi_ext.proto 文件。

说明

gNMI（gRPC Network Management Interface，gRPC 网络管理接口）是基于 gRPC 框架开发的一种操作协议，定义了一系列用于设备状态获取和配置操作的 RPC 方法。gNMI 支持通用数据模型，不需要为内容层额外提供业务模块的 Proto 文件。

---

2. Dial-out 模式

Dial-out 模式下，设备作为 gRPC 客户端，采集器作为 gRPC 服务器。设备主动和采集器建立 gRPC 连接，将设备上配置的订阅数据推送给采集器。该模式适用于网络设备较多的情况下向采集器提供设备数据信息。

根据对编码格式支持能力的不同，Dial-out 模式下的 Telemetry 数据模型分为以下两种类型：

- 三层 Telemetry 数据模型。该模型下，数据经过以下三个层次的处理：
  - RPC 层：定义在公共 proto 文件 grpc_dialout_v3.proto 中，提供消息格式等公共 RPC 方法。
  - Telemetry 层：定义在公共 proto 文件 telemetry.proto 中，提供数据采样相关服务，描述采样时间、采样路径等信息。
    - 若 telemetry.proto 文件中的 Encoding 字段为 Encoding_JSON（取值为 0），则 data_str 字段承载 JSON 编码格式的采样数据，data_gpb 字段为空。
    - 若 telemetry.proto 文件中的 Encoding 字段为 Encoding_GPB（取值为 1），则 data_gpb 字段承载 GPB 编码格式的采样数据，data_str 字段为空。
  - 内容层：承载 GPB 或 JSON 编码格式的业务数据。编码后的业务数据传输到采集器后，用户需要使用相应业务的 proto 文件对其进行解码，由 telemetry.proto 文件中的 sensor_path 字段标识对应哪个具体业务的 proto 文件。例如，当 sensor_path 取值为 ifmgr/statistics 时，就需要使用 Ifmgr_v3.proto 文件进行解码。

  以上的 RPC 层和 Telemetry 层在 gRPC 协议栈中属于 gRPC 层。
- 二层 Telemetry 数据模型，为缺省类型。该模型下，数据仅经过两个层次的处理。根据是否支持 gNMI 协议，又分为两种代码开发模式：

- 普通模式
  - RPC 层：定义在公共 proto 文件 grpc_dialout.proto 中，提供消息格式等公共 RPC 方法。grpc_dialout.proto 文件中的 jsonData 字段承载 JSON 数据。
  - 内容层：只承载 JSON 编码格式的业务数据。编码后的业务数据传输到采集器后，用户只需要对 grpc_dialout.proto 文件进行解码，不需要使用相应的业务 proto 文件。
- gNMI 模式
  - RPC 层：定义在 gnmi.proto 和 gnmi_ext.proto 文件中，提供消息格式等公共 RPC 方法。gnmi.proto 文件中的 Notification 字段承载 JSON 数据。
  - 内容层：只承载 JSON 编码格式的业务数据。编码后的业务数据传输到采集器后，用户只需要对 gnmi.proto 文件进行解码，不需要使用相应的业务 proto 文件。

2.1.4 gRPC 的服务模式

gRPC 服务根据 RPC 方法中的参数和返回值类型来确定不同的服务模式，具体的 RPC 方法在各 Proto 文件中定义，格式为：

```
service 业务名称{
    rpc 方法名称(参数名称)  returns(返回值) {}
    }
```

例如：

```
service gRPCService{
    rpc gRPCMethod(para)  returns(response) {}
    rpc gRPCMethod(para)  returns(stream response) {}
    rpc gRPCMethod(stream para)  returns(response) {}
    rpc gRPCMethod(stream para)  returns(stream response) {}
    }
```

表2-2 gRPC 服务模式

服务模式	说明及示例
简单模式	普通RPC调用，即客户端发送一个RPC请求，服务端立即返回一个RPC响应 例如：rpc Login (LoginRequest) returns (LoginReply)
服务端流模式	客户端发送一个RPC请求，服务端持续地返回多个RPC响应 例如：rpc Subscribe(SubsPara) returns(stream SubsRply)
客户端流模式	客户端持续地向服务器发送RPC请求，服务端返回一个RPC响应 例如：rpc Dialout(stream DialoutMsg) returns (DialoutResponse)
双向流模式	客户端持续地向服务器发送RPC请求，服务端持续响应RPC响应 例如：rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse)

2.2 采样数据

Telemetry 支持的采样数据可以从以下三个方面理解：

- Telemetry 支持采样的原始数据包括设备的接口流量统计、CPU 或者内存数据等信息。用户可在《Telemetry 性能指标集》中查询所支持的采样数据。
- Telemetry 基于 YANG 模型组织采样数据。
- Telemetry 支持对特定的采样路径采集指定的数据信息。

说明

- YANG 是一种数据建模语言，用于设计可以作为各种传输协议操作的配置数据模型、状态数据模型、远程调用模型和通知机制等。
- 设置采样路径时，可以通过设置采样条件，获取所需要的采样信息，具体的设置方法请参见“2.3 采样路径”。
- 各采样路径可以在设定的采样周期里完成对应的采集工作，对于部分数据量庞大的采样路径，若一个采样周期内完成不了采集工作，会持续到下个周期继续完成。当设备的 CPU 被限速的时候，采样周期可能会相应地延长。

2.3 采样路径

用户通过配置采样路径来获取自己需要的采样数据。设备上的数据已经通过 YANG 模型描述说明，基于 YANG 模型和它的子树路径组成了采样路径。

2.3.1 采样路径格式

1. 基本格式

基本格式为：YANG 模型节点名称/子节点名称，例如：Device/CPUs。

该采样路径中，“Device”表示 YANG 模型中的节点名称，“/”为各个节点的连接符，“CPUs”表示“Device”下的子节点名称。

2. 组织节点格式

组织节点格式的采样路径基于指定组织定义的 YANG 模型构建。

例如：openconfig-lldp:lldp/state

该采样路径中，“：”之前的“openconfig-lldp”为 YANG 模型名称，其中的 openconfig 表示定义该模型的组织名称，目前支持的组织有 IETF、OpenConfig、GSTA；“：”之后的内容为基本格式的采用路径。

2.3.2 过滤条件

在采样路径上支持配置附加的过滤条件，实现更为精细化地数据采集。

过滤条件中指定的节点必须是 YANG 模型中的 Leaf 节点。

1. 周期采样的过滤条件

周期采样是指设备以固定的时间间隔进行数据采样。周期性采样的采样路径上支持设置以下几种过滤条件。

(1) 谓语过滤

通过[column=“value”]格式设置条件表达式，表示采样数据的指定属性必须满足指定取值。

例如：ifmgr/statistics[ifindex="GigabitEthernet1/0/*"]，或者 ifmgr/statistics[ifindex=“12”]

该过滤条件表示，接口统计相关采样数据必须满足接口索引以“GigabitEthernet1/0/”开头或为“12”。

需要注意的是：

- 采样路径上附加谓语过滤条件时，不支持再配置其它方式的过滤条件；反之亦然。
- 一次采样过程中，同一采样路径最多支持 64 个谓语过滤条件，只要满足其中一个过滤条件设备就会推送采集数据。
- 谓语过滤条件中，仅索引节点的取值末尾可以为通配符“*”，且“*”仅能出现一次。

说明

目前，仅采样路径 ifmgr/statistics 支持谓语过滤条件。

(2) 条件过滤

通过 condition 关键字设置条件表达式，表示只有当指定节点满足指定条件时，设备才会将采样路径上的数据推送给采集器。如果未指定 condition 参数，则表示采样路径未限定推送条件。

例如：ifmgr/interfaces condition node ifindex operator ge value 285

该过滤条件表示，只有当接口索引大于等于 285 时，才会将 interfaces 节点的相关数据上报至采集器。

condition 关键字后的字段含义为：

- node *node*：节点名称，配置时需要完整输入，不区分大小写。用户可以在设备上输入 **sensor path path condition node ?**命令行后，通过查看联想出的帮助信息获得 *node* 参数的具体取值。
- operator *operator*：比较运算符。*operator* 的取值包括：
  - eq：等于。
  - ge：大于等于。
  - gt：大于。
  - le：小于等于。
  - lt：小于。
  - ne：不等于。

  各节点支持的比较运算符有所不同，请以具体节点的实现情况为准。用户可以在设备上输入 **sensor path path condition node** *node* **operator ?**命令行后，通过查看联想出的帮助信息获得 *operator* 参数的具体取值。
- value *value*：参与比较运算的参考值。

2. 条件触发采样的过滤条件

条件触发采样是指，设备根据一定的频率检测采样路径，当采样数据满足推送条件并上送采集器后，在抑制时间内不会重复上报。该采样方式仅在 gNMI 模式下支持。

条件触发采样仅支持谓语过滤条件，通过[column=“value”]格式设置条件表达式。例如：

components/component/integrated-circuit/chip/buffers/buffer[name="headroom"]/interfaces/interface[interface-id="HundredGigE1/0/31"]/input/queues/queue/state[name="que5"]/current-usage

关于谓语过滤的相关介绍，请参见“2.3.2 1. (1)谓语过滤”。

2.4 编码格式

gRPC 当前支持如下三种编码格式承载数据：

- JSON（JavaScript Object Notation）编码格式
- GPB（Google Protocol Buffer）编码格式
- JSON_IETF（JSON Data Interchange Format）编码格式

2.4.1 GPB 编码介绍

GPB 编码格式是一种用于通信协议及数据存储的序列化结构数据格式，具有与语言无关、平台无关、高扩展性的优点。GPB 编码与 XML、JSON 编码类似，不同之处在于它是一种二进制编码，性能更高。目前，GPB 包括 v2 和 v3 两个版本，设备支持的 GPB 版本是 v3。

表 2-3 为经过 GPB 编码解析前后的代码对比示例。

表2-3 GPB 编码格式

GPB 编码解析前	GPB 编码解析后

1 : 1
3: EOK
4: 0
1: H3C
2: m16287
3: H3C CR16006-F
15: 2
16: Device/Base
17: 188
18: 1617276638160
19: 1617276638208
20: 1617276638208
21: 5000
22: OK
23: 1
1{
1: 1617276638207
11: 3 {
1: 147699
2: "m16287"
3 "1.3.6.1.4.1.25506.1.1101"
4: 22
5: 1
10: "H3C Comware Platform Software, Software Version 7.1.075, ESS 8305\r\nH3C CR16006-F\r\nCopyright (c) 2004-2021 New H3C Technologies Co., Ltd. All rights reserved."
11: "2021-04-01T11:30:38"
	ReqId : 1
	errors: EOK
	totalSize: 0
	producer_name: H3C
	Node_Id_str: m16287
	ProductName: H3C CR16006-F
	Sub_Id_str: 2
	Sensor_path: Device/Base
	Collection_Id: 188
	Collection_start_time: 1617276638160
	msg_timestamp: 1617276638208
	Collection_end_time: 1617276638208
	Current_period: 5000
	except_desc: OK
	Encoding: Encoding_GPB
	row {
	timestamp: 1617276638207
	content:
	Base {
	Uptime: 147699
	HostName: "m16287"
	HostOid: "1.3.6.1.4.1.25506.1.1101"
	MaxSlotNum: 22
	MaxCPUIDNum: 1
	HostDescription: "H3C Comware Platform Software, Software Version 7.1.075, ESS 8305\r\nH3C CR16006-F\r\nCopyright (c) 2004-2021 New H3C Technologies Co., Ltd. All rights reserved."

12 {
1: "Z"
}
13 {
1: 3
2: 1
}
}
}
	LocalTime: "2021-04-01T11:30:38"
	TimeZone {
	Zone: "Z"
	}
	ClockProtocol {
	Protocol: 3
	MDCID: 1
	}
	}
	}

2.4.2 JSON 编码介绍

JSON 编码格式是一种轻量级的数据交换格式，它基于 ECMAScript 的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。JSON 的层次结构十分简洁和清晰，不仅易于开发人员阅读、编写代码，也易于机器解析、生成代码，是一种理想的数据交换语言。

表 2-4 为经过 JSON 编码解析前后的代码对比示例。

表2-4 JSON 编解码格式

JSON 编码解析前	JSON 编码解析后

{
1:“H3C”
2:“H3C”
3:“H3C device_test”
4:“not-config”
5:“sample”
2:“Syslog/LogBuffer”
3:"notification": {
"Syslog": {
"LogBuffer": {
"BufferSize": 512,
"BufferSizeLimit": 1024,
"DroppedLogsCount": 0,
"LogsCount": 100,
"LogsCountPerSeverity": {
"Alert": 0,
"Critical": 1,
"Debug": 0,
"Emergency": 0,
"Error": 3,
"Informational": 80,
"Notice": 15,
"Warning": 1
},
	{
	"producerName": "H3C",
	"deviceName": "H3C",
	"deviceModel": "H3C device_test",
	“deviceIpAddr” : "not-config",
	“eventType” : "sample",
	"sensorPath": "Syslog/LogBuffer",
	"jsonData": {
	"notification": {
	"Syslog": {
	"LogBuffer": {
	"BufferSize": 512,
	"BufferSizeLimit": 1024,
	"DroppedLogsCount": 0,
	"LogsCount": 100,
	"LogsCountPerSeverity": {
	"Alert": 0,
	"Critical": 1,
	"Debug": 0,
	"Emergency": 0,
	"Error": 3,
	"Informational": 80,
	"Notice": 15,
	"Warning": 1

"OverwrittenLogsCount": 0, "State": "enable" } }, "Timestamp": "1527206160022" } }	}, "OverwrittenLogsCount": 0, "State": "enable" } }, "Timestamp": "1527206160022" } } }

2.4.3 JSON_IETF 编码介绍

JSON_IETF 是基于 YANG 1.1 数据建模语言定义的 JSON 编码格式。JSON_IETF 的基本编码格式同 JSON，它的 YANG 数据节点（叶子、容器、叶子列表、列表、anydata 节点和 anyxml 节点）作为 JSON 对象来编码。对于 JSON_IETF 编码，JSON 对象成员的名称格式为：”命名空间前缀:成员标识符”，其中：

- 命名空间前缀为该数据节点的模块名称。
- 成员标识符为对应的 YANG 数据节点的标识符。

以上两个名称组成元素之间使用冒号（“:“）分隔，例如"H3C-device-data:Uptime"，其中 H3C-device-data 为命名空间前缀，Uptime 为 YANG 数据节点标识符。

对于子模块中定义的数据节点，它的 JSON 对象成员名称必须使用所属主模块的命名空间前缀。

表 2-5 为经过 JSON_IETF 编码解析前后的代码对比示例。

表2-5 JSON_IETF 编解码格式

JSON_IETF 编码解析前	JSON_IETF 编码解析后
{	{
1:“H3C”	"producerName": "H3C",
2:“H3C”	"deviceName": "H3C",
3:“H3C device_test”	"deviceModel": "H3C device_test",
4:“not-config”	“deviceIpAddr” : "not-config",
5:“sample”	“eventType” : "sample",
2:“Syslog/LogBuffer”	"sensorPath": "Syslog/LogBuffer",
3:"notification": {	"jsonData": {
"Syslog": {	"notification": {
"LogBuffer": {	"Syslog": {
"BufferSize": 512,	"LogBuffer": {
"BufferSizeLimit": 1024,	"H3C-syslog-data:BufferSize": 512,
"DroppedLogsCount": 0,	"H3C-syslog-data:BufferSizeLimit": 1024,
"LogsCount": 100,	"H3C-syslog-data:DroppedLogsCount": 0,
"LogsCountPerSeverity": {	"H3C-syslog-data:LogsCount": 100,
"Alert": 0,	"H3C-syslog-data:LogsCountPerSeverity": {
"Critical": 1,	"H3C-syslog-data:Alert": 0,
"Debug": 0,	"H3C-syslog-data:Critical": 1,
"Emergency": 0,	"H3C-syslog-data:Debug": 0,

"Error": 3, "Informational": 80, "Notice": 15, "Warning": 1 }, "OverwrittenLogsCount": 0, "State": "enable" } }, "Timestamp": "1527206160022" } }	"H3C-syslog-data:Emergency": 0, "H3C-syslog-data:Error": 3, "H3C-syslog-data:Informational": 80, "H3C-syslog-data:Notice": 15, "H3C-syslog-data:Warning": 1 }, "OverwrittenLogsCount": 0, "State": "enable" } }, "Timestamp": "1527206160022" } } }

2.5 Proto文件

2.5.1 Proto 文件介绍

Proto 文件使用 Protocol Buffers 语言编写。Protocol Buffers 简称 Protobuf，是 Google 公司开发的一种跨语言和平台的序列化数据结构的方式，是一个灵活的、高效的用于序列化数据的协议。Proto 文件用于定义 Protobuf 协议的编码规则。

采集器可以通过 Protoc 工具软件，根据“.proto”文件自动生成对应语言的开发代码（目前工具支持将 Proto 文件转换成多种语言的代码，例如 C++、JAVA、GO、Python），用户基于自动生成的代码进行二次开发，可以实现多种编程语言程序与设备的对接。

Proto 文件包含两种类型：公共 Proto 文件、业务 Proto 文件。

2.5.2 公共 Proto 文件

Telemetry 提供 6 类公共 Proto 文件：

- grpc_dialout.proto 文件
- grpc_dialout_v3.proto 文件
- telemetry.proto 文件
- grpc_service.proto 文件
- gnmi.proto 文件和 gnmi_ext.proto 文件
- dialout.proto 文件

1. grpc_dialout.proto 文件

grpc_dialout.proto 文件在设备作为客户端向采集器推送数据时使用，定义了 RPC 方法和承载的数据消息描述，其内容与含义如下文所示。（此处所列举的公共 Proto 文件内容仅为示例，请以设备的实际情况为准）

```
syntax = "proto2";//proto 版本为 v2 版本
package grpc_dialout;//包名为 grpc_dialout
message DeviceInfo{//推送的设备信息
```

```
    required string producerName = 1;//厂商名
    required string deviceName = 2;//设备名称
    required string deviceModel = 3;//设备的实体型号
    optional string deviceIpAddr = 4;//设备源 ip
    optional string eventType = 5;//采样类型
}
message ChunkInfo{//推送的消息格式描述
    required int64 totalSize = 1;//报文大小
    required uint64 totalFragments = 2;//报文总共分片的片数
    required uint64 nodeId = 3;//报文的 Id
}
message DialoutMsg{//推送的消息格式描述
    required DeviceInfo deviceMsg = 1;//设备实体信息
    required string sensorPath = 2;//采样路径
    required string jsonData = 3;//JSON 数据
    optional ChunkInfo chunkMsg = 4;//报文分片信息
}

message DialoutResponse{//采集器（gRPC 服务器）返回信息，预留（暂不处理返回值，可填充任意值）
    required string response = 1;//设备响应信息
}
service GRPCDialout {//服务名称为 GRPCDialout
    rpc Dialout(stream DialoutMsg) returns (DialoutResponse);//方法为 Dialout，单向流模式，提供数据推送的方法。入参是 DialoutMsg 数据流
}
```

2. grpc_dialout_v3.proto 文件

grpc_dialout_v3.proto 文件在设备作为客户端向采集器推送数据时使用，定义了 RPC 方法和承载的数据消息描述，主要用来定义三层 Telemetry 数据模型的编码规则，同时支持 GPB 和 JSON 两种编码格式。该文件的内容与含义如下文所示。（此处所列举的公共 Proto 文件内容仅为示例，请以设备的实际情况为准）

```
syntax = "proto3";//proto 版本为 v3 版本
package grpc_dialout_v3;//包名称为 grpc_dialout_v3
message DialoutV3Args{//三层编码描述消息
    int64 ReqId = 1;//请求 ID
    bytes data = 2;//承载的数据
    string errors = 3;//产生错误时的描述信息
    int32 totalSize = 4;// 分片时信息的总大小，未分片时为 0
}
service gRPCDialoutV3{//服务名称为 gRPCDialoutV3
    rpc DialoutV3(stream DialoutV3Args) returns (stream DialoutV3Args) {};
//方法为 DialoutV3，双向流模式，提供数据推送的方法，入参是 DialoutV3Args 数据流。
}
```

3. telemetry.proto 文件

telemetry.proto 文件在设备作为客户端向采集器推送数据时使用，主要用来定义三层 Telemetry 数据模型的采样数据消息描述，包括采样路径、采样时间戳等重要信息。该文件的内容与含义如下文所示。（此处所列举的公共 Proto 文件内容仅为示例，请以设备的实际情况为准）

```
syntax = "proto3";//Proto 版本为 v3 版本
package telemetry; //包名称为 telemetry
message Telemetry {//Telemetry 消息描述
    string producer_name = 1;//厂商名
    string node_id_str = 2;//设备名称
    string product_name = 3;//产品名称
    string subscription_id_str = 15;//订阅名
    string sensor_path = 16;//采样路径
    uint64 collection_id = 17;//标识采样轮次
    uint64 collection_start_time = 18;//采样开始时间
    uint64 msg_timestamp = 19;//生成本消息时间戳
    uint64 collection_end_time = 20; //采样结束时间
    uint32 current_period = 21;//采样精度
    string except_desc = 22;//异常描述信息
    enum Encoding {//编码方式
        Encoding_JSON = 0;//JSON 数据编码格式
        Encoding_GPB = 1;//GPB 数据编码格式
        };
    Encoding encoding = 23;//数据编码格式
    string data_str = 24;//数据编码非 GPB 时有效，否则为空
    TelemetryGPBTable data_gpb = 25; //承载的数据由 TelemetryGPBTable 定义
}
message TelemetryGPBTable {//消息描述
    repeated TelemetryRowGPB row = 1; //数组定义，标识数据是 TelemetryRowGPB 结构的重复
}
message TelemetryRowGPB {//消息描述
    uint64 timestamp = 1; //采样当前实例的时间戳
    bytes keys = 10; //保留字段
    bytes content = 11; //承载的采样实例数据
}
```

4. grpc_service.proto 文件

grpc_service.proto 文件在设备作为服务端对外推送数据时使用，定义了 RPC 方法和承载的数据消息描述。该文件内容与含义如下文所示。（此处所列举的公共 Proto 文件内容仅为示例，请以设备的实际情况为准）

```
syntax = "proto2";//Proto 版本为 v2 版本
package grpc_service;//包名为 grpc_service
message GetJsonReply { //Get 方法应答结果
    required string result = 1;
}
message SubscribeReply { //订阅结果
    required string result = 1;
}
message ConfigReply { //配置结果
    required string result = 1;
}
message ReportEvent { //订阅事件结果定义
    required string token_id = 1; //登录 token_id
```

```
    required string stream_name = 2; //订阅的事件流名称
    required string event_name = 3; //订阅的事件名
    required string json_text = 4; //订阅结果 json 字符串
}

message GetReportRequest{ //获取事件订阅结果请求
    required string token_id = 1; //登录成功后的 token_id
}
message LoginRequest {//登录请求参数定义
    required string user_name = 1; //登录请求用户名
    required string password = 2; //登录请求密码
}
message LoginReply {//登录请求应答定义
    required string token_id = 1; //登录成功后返回的 token_id
}
message LogoutRequest {//退出登录请求参数定义
    required string token_id = 1;  //token_id
}
message LogoutReply { //退出登录返回结果定义
    required string result = 1; //退出登录结果
}
message SubscribeRequest {//定义事件流
    required string stream_name = 1; //事件流名称
}
message CliConfigArgs {//向设备下发配置命令，并指定命令行参数
    required int64 ReqId = 1; //配置命令请求 ID
    required string cli = 2; //配置命令
}
message CliConfigReply { //设备返回配置命令行执行的结果
    required int64 ResReqId = 1; //返回配置命令请求 ID，与 CliConfigArgs 相对应
    required string output = 2; //返回配置命令执行输出
    required string errors = 3; //标记配置命令执行结果
}
message DisplayCmdArgs {//向设备下发 display 命令，并指定命令行参数
    required int64 ReqId = 1; //display 命令请求 ID
    required string cli = 2; //display 命令
}
message DisplayCmdReply {//设备返回 display 命令行执行的结果
    required int64 ResReqId =1;//display 命令请求 ID，与 DisplayCmdArgs 相对应
    required string output = 2; //返回 display 命令执行输出
    required string errors = 3; //标记 display 命令执行结果
}
service GrpcService {//定义 gRPC dialin 模式的 RPC 方法
    rpc Login (LoginRequest) returns (LoginReply) {}//登录方法
    rpc Logout (LogoutRequest) returns (LogoutReply) {}//退出登录方法
    rpc SubscribeByStreamName (SubscribeRequest) returns (SubscribeReply) {}//订阅事件流
    rpc GetEventReport (GetReportRequest) returns (stream ReportEvent) {}//获取事件结果
    rpc CliConfig (CliConfigArgs)  returns (stream CliConfigReply) {}//gRPC 支持通过命令行
下发配置命令，并返回执行结果
```

```
    rpc DisplayCmdTextOutput(DisplayCmdArgs)  returns(stream DisplayCmdReply) {}//gRPC 支持通过命令行下发 display 命令，并返回查询结果
}
```

5. gnmi.proto 文件和 gnmi_ext.proto 文件

gnmi.proto 和 gnmi_ext.proto 文件定义了 gNMI 类操作的公共 RPC 方法。它们由 Google 提供，是开源文件，此处不做详细描述，具体内容和详细介绍请下载相关文件后查看。gnmi.proto 文件和 gnmi_ext.proto 文件的官方下载地址如下：

- https://github.com/openconfig/gnmi/tree/master/proto/gnmi/gnmi.proto
- https://github.com/openconfig/gnmi/tree/master/proto/gnmi_ext/gnmi_ext.proto

6. dial_out.proto 文件

dial_out.proto 文件定义了 gNMI 类的 RPC 方法，该文件来源于 SONIC，官方下载地址为：https://github.com/Azure/sonic-telemetry/blob/master/proto/dial_out.proto

2.5.3 业务 Proto 文件

设备提供多个业务 Proto 文件，它们用于定义 Dial-in 模式下数据订阅所需的 RPC 方法和消息描述，以及定义 Dial-out 模式下具体业务的 GPB 编码所需的消息描述，采集器需要根据实际监控的业务选择对应的 Proto 文件进行编码和二次开发。

请联系 H3C 技术支持人员获取业务 Proto 文件。

2.5.4 不同对接模式支持的 Proto 文件

表2-6 不同对接模式支持的 Proto 文件

对接模式	Proto 文件
Dial-in模式	普通Subscribe操作： • grpc_service.proto 文件 • 业务 Proto 文件 gNMI操作： • grpc_service.proto 文件 • gnmi.proto 文件和 gnmi_ext.proto 文件

Dial-out模式	二层数据模型下：
	• 普通模式
	  ○ grpc_dialout.proto 文件
	• gNMI 模式
	  ○ dialout.proto 文件
	  ○ gnmi.proto 文件和 gnmi_ext.proto 文件
	三层数据模型下：
	• grpc_dialout_v3.proto 文件
	• telemetry.proto 文件
	• 业务 Proto 文件

3 Telemetry 配置限制和指导

所有 gRPC 相关功能，都需要在成功执行 **grpc enable** 命令后，才能配置。

如果执行 **undo grpc enable** 命令关闭了 gRPC 功能，则所有 gRPC 相关配置都会被删除。

缺省情况下，设备和采集器建立的 gRPC 连接是非加密的。配置设备和采集器建立 gRPC 连接时引用 PKI 域后，设备和采集器会基于 TLS（Transport Layer Security，传输层安全）协议进行通道加密和双向证书认证，从而提高 gRPC 通信的安全性。需要注意的是：

- 指定的 PKI 域必须存在，并且 PKI 域中包含完整的证书和密钥。
- 指定 PKI 域后，gRPC 功能将重启，与采集器的连接将短暂断开。采集器需要重新发送连接请求才能继续访问设备。

4 配置设备侧的 Telemetry 订阅

本章节主要描述设备上如何通过命令行配置 Telemetry 订阅。采集器上的 gRPC 对接软件需要另开发，具体开发过程请参考“5 Telemetry 对接软件二次开发”。

4.1 配置Dial-in模式的Telemetry订阅

4.1.1 功能简介

Dial-in 模式是指设备作为 gRPC 服务器，采集器作为 gRPC 客户端发起到设备的 gRPC 连接，由设备进行数据采集及上送。通过在设备上配置 Dial-in 模式，使采集器可以获取和订阅设备上的数据和事件。

4.1.2 配置限制和指导

如果设备和采集器的 gRPC 连接断开，设备会自动取消订阅，不再采集推送数据。gRPC 连接断开后，设备不支持自动配置恢复，需要采集器重新发起 gRPC 连接请求。

4.1.3 配置准备

在配置 Dail-in 模式之前，需要完成以下任务：

- 确保 gRPC 服务器与 gRPC 客户端之间路由可达。
- 如果需要配置 gRPC 连接的加密功能，则必须完成所引用的 PKI 域的配置，并且该 PKI 域中需要包含完整的证书和密钥。关于 PKI 的配置，请参见相关产品“安全配置指导”中的“PKI”。

4.1.4 配置步骤

1. 配置 gRPC 服务

(1) 进入系统视图。

system-view

(2) 开启 gRPC 功能。

grpc enable

缺省情况下，gRPC 功能处于关闭状态。

(3) （可选）配置 gRPC 服务的端口号。

grpc port *port-number*

缺省情况下，gRPC 服务的端口号为 50051。

修改端口号后，gRPC 功能将重启，正在访问的客户端将被断开，客户端需要重新发送连接请求才能继续访问。

(4) （可选）配置 gRPC 会话超时时间。

grpc idle-timeout *minutes*

缺省情况下，gRPC 会话超时时间为 5 分钟。

2. （可选）配置设备与采集器之间的安全通信

(1) 进入系统视图。

system-view

(2) 配置设备和采集器建立 gRPC 连接时引用的 PKI 域。

grpc pki domain *domain-name*

缺省情况下，设备和采集器建立 gRPC 连接时不会引用 PKI 域。

3. 配置 gRPC 用户

(1) 进入系统视图。

system-view

(2) 添加设备管理类本地用户。

local-user *user-name* [**class manage**]

设备上需要为 gRPC 客户端创建本地用户，gRPC 客户端才能与设备建立 gRPC 会话。

(3) 设置本地用户的密码。

password [{ **hash** | **simple** } *password*]

缺省情况下，不存在本地用户密码，即本地用户认证时无需输入密码，只要用户名有效且其他属性验证通过即可认证成功。

(4) 配置本地用户的授权用户角色为 network-admin。

authorization-attribute user-role network-admin

缺省情况下，本地用户的授权用户角色为 network-operator。

(5) 配置本地用户可以使用的服务类型为 HTTPS 服务。

service-type https

缺省情况下，未配置用户的服务类型。

有关 **local-user**、**password**、**authorization-attribute** 和 **service-type** 命令的详细介绍，请参见“安全命令参考”中的“AAA”。

4. （可选）开启 gRPC Dial-in 模式的日志功能

(1) 进入系统视图。

system-view

(2) 开启 gRPC Dial-in 模式的日志功能。请至少选择其中一项进行配置。

- 开启 gRPC Dial-in 模式的 RPC 类操作日志功能。

  grpc log dial-in rpc { all | { cli | get }* }

  缺省情况下，gRPC Dial-in 模式的 RPC 类操作日志功能处于关闭状态。

- 开启 gRPC Dial-in 模式的 gNMI 类操作日志功能。

  grpc log dial-in gnmi { all | { capabilities | get | set | subscribe }* }

  缺省情况下，gRPC Dial-in 模式的 gNMI Set 操作日志功能处于开启状态，其他 gNMI 类操作日志功能处于关闭状态。

为了管理员定位 gRPC 问题的需要，可以开启 gRPC 日志功能，以便记录设备对 gRPC 报文的处理信息。如果 gRPC 操作频繁，设备会输出大量 gRPC 日志，影响设备性能，建议仅开启需关注的 gRPC 操作的日志功能。

4.1.5 验证配置

在任意视图下执行 **display grpc** 命令可以显示配置后 gRPC Dial-in 模式的相关信息，通过查看显示信息验证配置的效果。

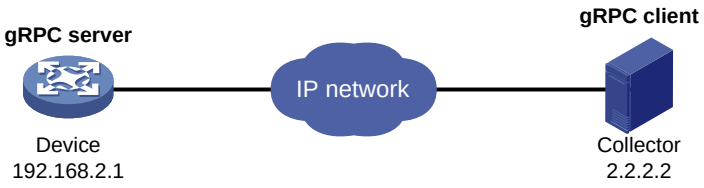
4.1.6 配置举例

1. 组网需求

如图 4-1 所示，设备作为 gRPC 服务器与采集器相连，采集器为 gRPC 客户端。

通过在设备上配置 gRPC Dial-in 模式，使 gRPC 客户端可以订阅设备上的相关事件。

图4-1 gRPC Dial-in 模式配置组网图



2. 配置 Device（gRPC 服务器）

开启 gRPC 功能。

```
<Device> system-view
[Device] grpc enable
```

创建本地用户 test，配置该用户的密码为 123456TESTplat&!，授权用户角色为 network-admin，可以使用的服务类型为 HTTPS 服务。

```
[Device] local-user test
[Device-luser-manage-test] password simple 123456TESTplat&!
[Device-luser-manage-test] authorization-attribute user-role network-admin
[Device-luser-manage-test] service-type https
[Device-luser-manage-test] quit
```

4.2 配置Dial-out模式的Telemetry订阅

4.2.1 功能简介

Dial-out 模式是指设备作为 gRPC 客户端，采集器作为 gRPC 服务端，由设备主动发起到采集器的 gRPC 连接，进行数据采集及上送。通过在设备上配置 Dial-out 模式的 Telemetry 订阅，使设备能够主动向采集器推送用户订阅的数据和事件。

Dial-out 模式下的数据订阅以及上报通信参数由如下两个关键配置实现：

- 配置传感器：传感器用来指定设备上采样的数据源，即采样路径。一个传感器组是一到多个采样路径的集合。传感器组有两种类型：普通传感器组和 gNMI 模式的传感器组。

  采样路径包括以下类型：

  - 周期采样：传感器组以固定的时间间隔来进行数据采样。关于周期采样类型的采样路径，请参见对应模块的《Telemetry 性能指标集》手册。
  - 事件触发采样：传感器组的数据采样没有固定周期，仅由事件触发。关于事件触发采样类型的采样路径，请参见对应模块的《Telemetry 性能指标集》手册。
  - 条件触发采样：传感器组根据一定频率检测采样路径，如果满足推送条件，则采集数据并上送给采集器。关于条件触发采样类型的采样路径以及相关的检测频率、推送条件，请联系 H3C 技术支持人员获取具体信息。

    条件触发采样类型的采样路径仅在 gNMI 模式的传感器组中配置才能生效。

  说明

  同一传感器组内只能支持一种类型的采样路径。

- 配置采集器：采集器用于接收网络设备推送的采样数据。设备上需要建立目标组并在目标组中配置正确的采集器地址信息，才能和采集器通信。

完成传感器组和目标组的配置后，需要创建“订阅”并将二者关联，设备才能和目标组中的采集器建立 gRPC 连接，从而将订阅报文发送给采集器。

订阅分为普通订阅和 gNMI 模式的订阅，其中普通订阅仅支持关联普通传感器组，gNMI 模式的订阅仅支持关联 gNMI 模式的传感器组。

4.2.2 配置限制和指导

如果采集器和设备的 gRPC 连接断开，设备会进行重新连接，再次上送数据，但是重连期间的数据会丢失。

在系统主备倒换或保存 gRPC 业务配置并重启后，gRPC 会重新加载相关配置，gRPC 业务会继续进行，但是重启或者倒换期间的采样数据会丢失。

4.2.3 配置准备

在配置 Dail-out 模式之前，需要完成以下任务：

- 确保 gRPC 服务器与 gRPC 客户端之间路由可达。

- 如果需要配置 gRPC 连接的加密功能，则必须完成所引用的 PKI 域的配置，并且该 PKI 域中需要包含完整的证书和密钥。关于 PKI 的配置，请参见相关产品“安全配置指导”中的“PKI”。

4.2.4 配置步骤

1. 开启 gRPC 功能

(1) 进入系统视图。

system-view

(2) 开启 gRPC 功能。

grpc enable

缺省情况下，gRPC 功能处于关闭状态。

(3) （可选）配置 gRPC 使用的 Telemetry 数据模型。

grpc data-model { 2-layer | 3-layer }

缺省情况下，设备使用二层 Telemetry 数据模型上送数据。

设备使用二层 Telemetry 数据模型上送数据时，不支持对上送数据使用 GPB 编码格式。

2. （可选）配置设备与采集器之间的安全通信

(1) 进入系统视图。

system-view

配置设备和采集器建立 gRPC 连接时引用的 PKI 域。

(2) **grpc pki domain** *domain-name*

缺省情况下，设备和采集器建立 gRPC 连接时不会引用 PKI 域。

3. 配置普通传感器

(1) 进入系统视图。

system-view

(2) 进入 Telemetry 视图。

telemetry

(3) 创建普通传感器组，并进入传感器组视图。

sensor-group *group-name*

(4) 配置采样路径。

- 周期采样类型的采样路径

  sensor path *path* **depth** *depth*

  sensor path *path* [**condition node** *node* **operator** *operator* **value** *value*]

多次执行本命令可配置多个采样路径。多次执行本命令且指定的采样路径相同时，最后一次执行的命令生效。

说明

- **depth** 为采样深度，缺省值为 1。1 表示设备只上报当前采样路径的所有列数据；2 表示设备除了上报当前采样路径的列数据外，还会上报当前采样路径下所有子表的列数据；3 表示设备

除了上报当前采样路径的列数据外，还会上报当前采样路径下所有子表以及这些子表的所有子表的列数据。

- **condition node** *node* **operator** *operator* **value** *value* 用于设置样路径的推送条件，只有当指定节点满足指定条件时才会将采样路径的数据推送给采集器。
- 配置本命令时，若在 *path* 中携带了谓语过滤条件表达式，则不支持再指定 **condition** 参数；反之亦然。

---

4. 配置采集器

(1) 进入系统视图。

system-view

(2) 进入 Telemetry 视图。

telemetry

(3) 创建目标组，并进入目标组视图。

destination-group *group-name*

建议系统中创建的目标组数量不超过 5 个，否则会影响系统性能。

(4) 配置采集器的地址和相关参数。

（IPv4 网络）

ipv4-address *ipv4-address* [**port** *port-number*] [**vpn-instance** *vpn-instance-name*]

（IPv6 网络）

ipv6-address *ipv6-address* [**port** *port-number*] [**vpn-instance** *vpn-instance-name*]

采集器的 IPv6 地址不能指定为 IPv6 链路本地地址。

多次执行本命令可配置多个采集器。配置本命令时，只要任意一个参数不同，就算不同的采集器。

5. 配置订阅

(1) 进入系统视图。

system-view

(2) 进入 Telemetry 视图。

telemetry

(3) 创建订阅，并进入订阅视图。

- 普通订阅

  subscription *subscription-name*

(4) （可选）配置设备发送订阅报文的传输协议。

protocol { **grpc** | **udp** }

缺省情况下，设备发送订阅报文的传输协议为 gRPC。

(5) （可选）配置设备发送的订阅报文的 DSCP 优先级。

dscp *dscp-value*

缺省情况下，设备发送的订阅报文的 DSCP 优先级为 0。

DSCP 优先级的取值越大，报文的优先级越高。

(6) （可选）配置设备发送订阅报文的源地址。

source-address { *ipv4-address* | **interface** *interface-type interface-number* | **ipv6** *ipv6-address* }

缺省情况下，设备使用路由出接口的主 IP 地址作为发送订阅报文的源 IP 地址。

当设备发送订阅报文的源地址发生变化时，设备将会重新连接 gRPC 服务器。

(7) （可选）配置上送数据的编码格式。

encoding { **gpb** | **json** }

缺省情况下，上送数据的编码格式为 JSON。

仅当设备使用三层 Telemetry 数据模型时，可以对上送数据使用 GPB 编码格式。

(8) 配置关联传感器组。

sensor-group *group-name* [**sample-interval** [**msec**] *interval*]

仅当订阅的传感器组中的采样路径类型为周期采样时，才需要配置本命令中的 **sample-interval** 参数。

(9) 配置关联目标组。

destination-group *group-name*

6. 开启 gRPC Dial-out 模式的日志功能

(1) 进入系统视图。

system-view

(2) 开启 gRPC Dial-out 模式的日志功能。

grpc log dial-out { **all** | { **event** | **sample** }* }

缺省情况下，gRPC Dial-out 模式的日志功能处于关闭状态。

为了管理员定位 gRPC 问题的需要，可以开启 gRPC 日志功能，以便记录设备对 gRPC 报文的处理信息。

4.2.5 验证配置

在 Probe 视图下执行 **display system internal telemetry** 命令可以显示配置后 gRPC Dial-out 模式的相关信息，通过查看显示信息验证配置的效果。

4.2.6 配置举例

1. 组网需求

设备作为 gRPC 客户端与采集器相连，采集器为 gRPC 服务器，接收数据的端口号为 50051。

通过在设备上配置 gRPC Dial-out 模式，使设备以 10 秒的周期向采集器推送接口模块的设备能力信息。

图4-2 gRPC Dial-out 模式配置组网图

2. 配置 Device（gRPC 客户端）

在开始下面的配置之前，假设设备与采集器的 IP 地址都已配置完毕，并且它们之间路由可达。

开启 gRPC 功能。

```
<Device> system-view
[Device] grpc enable
```

创建传感器组 test，并添加采样路径为 ifmgr/devicecapabilities。

```
[Device] telemetry
[Device-telemetry] sensor-group test
[Device-telemetry-sensor-group-test] sensor path ifmgr/devicecapabilities
[Device-telemetry-sensor-group-test] quit
```

创建目标组 collector1，并配置采集器的 IP 地址为 2.2.2.2、端口号为 50051。

```
[Device-telemetry] destination-group collector1
[Device-telemetry-destination-group-collector1] ipv4-address 2.2.2.2 port 50051
[Device-telemetry-destination-group-collector1] quit
```

创建订阅 A，配置关联传感器组为 test，数据采样和推送周期为 10 秒，关联目标组为 collector1。

```
[Device-telemetry] subscription A
[Device-telemetry-subscription-A] sensor-group test sample-interval 10
[Device-telemetry-subscription-A] destination-group collector1
[Device-telemetry-subscription-A] quit
```

5 Telemetry 对接软件二次开发举例（Dial-in 模式）

说明

本指南中展示的代码仅供参考，由于没有实际的代码框架，在实际的对接过程中不能直接使用。除举例中生成的代码之外，其它代码还需要开发人员自行开发。

对于 Dial-in 模式，主要是实现 gRPC 客户端的代码，使采集器能够获取设备上的采集数据并进行解析。客户端代码主要包括以下三个部分：

- 进行登录操作，获取 token_id。
- 为要发起的 RPC 方法准备参数，用 Proto 文件生成的服务类发起 RPC 调用并解析返回结果。
- 退出登录。

本章节用于介绍 Dial-in 模式的设备与对接客户端软件的开发过程，可实现如下操作：

- 普通 Subscribe 操作
- gNMI Subscribe 操作

5.1 开发前准备

5.1.1 开发人员要求

- 熟悉 gRPC 的开发（可通过 https://doc.oschina.net/grpc 学习）。
- 熟悉 GPB 编码的开发（可通过 https://developers.google.com/protocol-buffers 学习）。
- 熟悉对应语言（C++、JAVA、Python、GO）的开发。

5.1.2 开发环境准备

1. 获取 Proto 文件

联系 H3C 技术支持人员获取相关 Proto 文件。

2. 获取处理 Proto 文件的工具软件 protoc

下载地址：https://github.com/google/protobuf/releases

3. 获取对应开发语言的 protobuf 插件

下载地址：https://github.com/google/protobuf/releases

开发者需要准备好对应语言的开发环境，例如获取 C++插件 protobuf-cpp。本指南会给出当前主流语言的开发举例（C++、GO、Python、JAVA）。

4. 存放 Proto 文件和工具软件

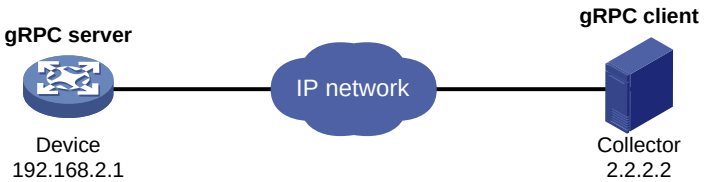
建议将获取到的 Proto 文件和 protoc 工具存在开发代码的工程目录下，具体以实际使用的开发环境为准。

5.2 组网需求

如图 5-1 所示，设备作为 gRPC 服务器与采集器相连，采集器为 gRPC 客户端。gRPC 客户端登录设备的用户名为 admin、密码为 123456。

通过配置 gRPC Dial-in 模式，使 gRPC 客户端可以订阅设备上的相关事件。

图5-1 gRPC Dial-in 模式配置组网图



5.3 配置设备侧的Telemetry订阅

执行以下配置之前，请确保 gRPC 服务器与 gRPC 客户端之间路由可达。

开启 gRPC 功能。

```
<Device> system-view
[Device] grpc enable
```

创建本地用户 admin，配置该用户的密码为 123456，授权用户角色为 network-admin，可以使用的服务类型为 HTTPS 服务。

```
[Device] local-user admin
[Device-luser-manage-admin] password simple 123456
[Device-luser-manage-admin] authorization-attribute user-role network-admin
[Device-luser-manage-admin] service-type https
[Device-luser-manage-admin] quit
```

5.4 gRPC Dial-in模式二次开发举例（C++）

5.4.1 生成代码

开发代码之前，需要使用 protoc 工具将收集到的 Proto 文件转换成 C++代码，并将生成的代码加入到开发的工程中。

本例中，开发普通 Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- 业务 Proto 文件，本例中为 BufferMonitor.proto

本例中，开发 gNMI Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- gnmi.proto 和 gnmi_ext.proto

使用 protoc 工具生成上述 Proto 文件的 C++代码，示例如下：

```
$protoc --plugin=./grpc_cpp_plugin  --grpc_out=. --cpp_out=. *.proto
```

5.4.2 开发代码

1. 普通 Subscribe 操作

以调用 GrpcService 和 BufferMonitorService 服务类为例，编码步骤如下：

(1) 编写一个 GrpcServiceTest 类。

在 GrpcServiceTest 类中使用由 grpc_service.proto 生成的 GrpcService::Stub 类，通过 grpc_service.proto 自动生成的 Login 和 Logout 方法分别完成登录和退出操作，代码示例如下：

```
class GrpcServiceTest
{
public:
    /* 构造函数 */
    GrpcServiceTest(std::shared_ptr<Channel> channel):
GrpcServiceStub(GrpcService::NewStub(channel))  {}

    /* 成员函数 */
    int Login(const std::string& username, const std::string& password);
    void Logout();
    void listen();
Status listen(const std::string& command);
```

```
    /* 成员变量 */
    std::string token;

private:
    std::unique_ptr<GrpcService::Stub> GrpcServiceStub;  //使用 grpc_service.proto 生成的
GrpcService::Stub 类
};
```

(2) 实现自定义的 Login 方法。

通过用户名、密码调用 GrpcService::Stub 类的 Login 方法完成登录，代码示例如下：

```
int GrpcServiceTest::Login(const std::string& username, const std::string& password)
{
    LoginRequest request;   //设置用户名密码
    request.set_user_name(username);
    request.set_password(password);

    LoginReply reply;
    ClientContext context;

  //调用登录方法
  Status status = GrpcServiceStub->Login(&context, request, &reply);
  if (status.ok())
    {
        std::cout << "login ok!" << std::endl;
        std::cout <<"token id is :" << reply.token_id() << std::endl;
        token = reply.token_id();  //登录成功，获取到 token
        return 0;
    }
    else{
        std::cout << status.error_code() << ": " << status.error_message()
                << ". Login failed!" << std::endl;
        return -1;
    }
}
```

(3) 发起对设备的 RPC 方法请求。

以订阅接口丢包事件为例，代码如下：

```
rpc SubscribePortQueDropEvent(PortQueDropEvent) returns (grpc_service.SubscribeReply) {}
```

(4) 编写一个 BufMon_GrpcClient 类来封装发起的 RPC 方法。

使用 BufferMonitor.proto 自动生成的 BufferMonitorService::Stub 类完成 RPC 方法的调用，代码示例如下：

```
class BufMon_GrpcClient
{
public:
    BufMon_GrpcClient(std::shared_ptr<Channel> channel):
mStub(BufferMonitorService::NewStub(channel))
    {}
```

```
    std::string BufMon_Sub_AllEvent(std::string token);
    std::string BufMon_Sub_BoardEvent(std::string token);
    std::string BufMon_Sub_PortOverrunEvent(std::string token);
    std::string BufMon_Sub_PortDropEvent(std::string token);

    /* get 表项 */
    std::string BufMon_Sub_GetStatistics(std::string token);
    std::string BufMon_Sub_GetGlobalCfg(std::string token);
    std::string BufMon_Sub_GetBoardCfg(std::string token);
    std::string BufMon_Sub_GetNodeQueCfg(std::string token);
    std::string BufMon_Sub_GetPortQueCfg(std::string token);

private:
    std::unique_ptr<BufferMonitorService::Stub> mStub; //使用 BufferMonitor.proto 自动生成的
类
};
```

(5) 实现自定义的 std::string BufMon_Sub_PortDropEvent(std::string token)方法完成接口丢包事件订阅。

实现丢包事件订阅方法的代码示例如下：

```
std::string BufMon_GrpcClient::BufMon_Sub_PortDropEvent(std::string token)
{
    std::cout << "-------BufMon_Sub_PortDropEvent-------- " << std::endl;

    PortQueDropEvent stNodeEvent;
    PortQueDropEvent_PortQueDrop* pstParam = stNodeEvent.add_portquedrop();

    UINT uiIfIndex = 0;
    UINT uiQueIdx = 0;
    UINT uiAlarmType = 0;

    std::cout<<"Please input interface queue info : ifIndex queIdx alarmtype " << std::endl;
    cout<<"alarmtype : 1 for ingress; 2 for egress; 3 for port headroom"<<endl;

    std::cin>>uiIfIndex>>uiQueIdx>>uiAlarmType; //设置订阅参数，接口索引等
    pstParam->set_ifindex(uiIfIndex);
    pstParam->set_queindex(uiQueIdx);
    pstParam->set_alarmtype(uiAlarmType);

    ClientContext context;

    /* token need add to context */ //设置登录成功后返回的 token_id
    std::string key = "token_id";
    std::string value = token;
    context.AddMetadata(key, value);

    SubscribeReply reply;
    Status status = mStub->SubscribePortQueDropEvent(&context,stNodeEvent,&reply); //调用
RPC 方法
```

```
    return reply.result();
}
```

(6) 循环等待设备上报事件。

在之前的 GrpcServiceTest 类中实现此方法，代码示例如下：

```
void GrpcServiceTest::listen()
{
    GetReportRequest reportRequest;
    ClientContext context;
    ReportEvent reportedEvent;

    /* add token to request */
    reportRequest.set_token_id(token);

    std::unique_ptr< ClientReader< ReportEvent>>
reader(GrpcServiceStub->GetEventReport(&context, reportRequest)); //通过
grpc_service.proto 自动生成的类的 GetEventReport 来获取事件信息

    std::string streamName;
    std::string eventName;
    std::string jsonText;
    std::string token;

    JsonFormatTool jsonTool;

    std::cout << "Listen to server for Event" << std::endl;
    while(reader->Read(&reportedEvent) ) //读取收到的上报事件
    {
        streamName = reportedEvent.stream_name();
        eventName = reportedEvent.event_name();
        jsonText = reportedEvent.json_text();
        token = reportedEvent.token_id();

        std::cout << "/***********EVENT COME**************/" << std::endl;
        std::cout << "TOKEN: " << token << std::endl;
        std::cout << "StreamName: "<< streamName << std::endl;
        std::cout << "EventName: " << eventName << std::endl;
        std::cout << "JsonText without format: " << std::endl << jsonText << std::endl;
        std::cout << std::endl;
        std::cout << "JsonText Formated: " << jsonTool.formatJson(jsonText) << std::endl;
        std::cout << std::endl;
    }

    Status status = reader->Finish();
    std::cout << "Status Message:" << status.error_message() << "ERROR code :" <<
status.error_code();
}
```

(7) 实现向设备发起接口丢包事件的订阅请求。

代码示例如下：

```
void bufmon_test_PortDropStream()
{
    auto channel = grpc::CreateChannel(g_server_address,
grpc::InsecureChannelCredentials());

    /* 1. login */
    GrpcServiceTest reporter(channel);
        if(0 != reporter.Login(g_username, g_password))
    {
                return;
        }

        /* 2. subscribe */
        BufMon_GrpcClient cSubscriber(channel);

        std::string replyForSysLog;

    /* subscribe bufmon port overrun event */
        replyForSysLog = cSubscriber.BufMon_Sub_PortDropEvent(reporter.token);
        std::cout<<"BufMon board event: "<<replyForSysLog<<std::endl;

        /* 3. listen to the server and get event */
    reporter.listen();

    std::cout<<"End of main."<<std::endl;
    return;

}
```

(8)　调用 Logout 方法退出登录。

调用 Logout 方法的代码示例如下：

```
void GrpcServiceTest:: Logout ()
{
    LogoutRequest request;
    request.set_token_id(token);
    LogoutReply reply;
    ClientContext context;
    Status status = mStub->Logout(&context, request, &reply);
std::cout << "Logout! :" << reply.result() << std::endl;
}
```

2. gNMI Subscribe 操作

gNMI Subscribe 操作的编码步骤如下：

(1)　编写一个 GrpcServiceTest 类。

步骤与“5.4.2　1.”中编写 GrpcServiceTest 类相同。

(2)　实现自定义的 Login 方法。

步骤与“5.4.2　1.”中编写 Login 方法相同。

(3) 发起对设备的 RPC 方法请求。

代码示例如下：

```
rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
```

(4) 编写 gnmi_client 类来封装发起的 RPC 方法。

代码示例如下：

```
class gnmi_client
{
public:
      explicit gnmi_client(const std::string &address,const std::string &tokenId);

      bool TestCapabilities();
      bool TestGet();
      bool TestSet();
      bool TestSubscribePool();
      bool TestSubscribeOnce();
      bool TestSubscribeStream();
      bool TestSubscribeStreamWithAlias();


private:
      void PrintCapabilityResponse(const gnmi::CapabilityResponse &response);
      void PrintGetResponse(const gnmi::GetResponse &response);
      void PrintSubscribeResponse(const gnmi::SubscribeResponse &response);
      void PrintSubscribeRequest(const gnmi::SubscribeRequest &request);
      void PrintGetRequest(const gnmi::GetRequest &request);
      void PrintSetRequest(const gnmi::SetRequest &request);

      void FillGetRequest(gnmi::GetRequest &request);
      void FillSetRequest(gnmi::SetRequest &request);
      void FillSubscribeRequestByOnce(gnmi::SubscribeRequest &request);
      void FillSubscribeRequestByPool(gnmi::SubscribeRequest &request);
      void FillSubscribeRequestByStream(gnmi::SubscribeRequest &request);
      void FillSubscribePool(gnmi::SubscribeRequest &request);
      void FillSubscribeAlias(gnmi::SubscribeRequest &request);

private:
      std::unique_ptr<gnmi::gNMI::Stub> mStubGnmiService;
      std::string mTokenID;
};
```

(5) 实现自定义的订阅方法。

实现事件订阅方法的代码示例如下：

```
void gnmi_client::FillSubscribeRequestByOnce(gnmi::SubscribeRequest &request)
{
  auto subscribeList = request.mutable_subscribe();

  auto prefix = subscribeList->mutable_prefix();
  auto pathelem01 = prefix->add_elem();
```

```
  pathelem01->set_name("LLDP");

  auto subscribe = subscribeList->add_subscription();

  auto path = subscribe->mutable_path();
  auto pathelem02 = path->add_elem();
  pathelem02->set_name("NeighborEvent");

  auto pathelem03 = path->add_elem();
  pathelem03->set_name("Neighbor");
  (*pathelem03->mutable_key())["IfName"] = "xxx";

  subscribeList->set_mode(::gnmi::SubscriptionList_Mode_ONCE);
  subscribeList->set_encoding(::gnmi::JSON);
}

void gnmi_client::FillSubscribeRequestByPool(gnmi::SubscribeRequest &request)
{
  auto subscribeList =  request.mutable_subscribe();

  auto prefix = subscribeList->mutable_prefix();
  auto pathelem01 = prefix->add_elem();
  pathelem01->set_name("Device");

  auto subscribe = subscribeList->add_subscription();

  auto path = subscribe->mutable_path();
  auto pathelem02 = path->add_elem();
  pathelem02->set_name("CPUs");
  auto pathelem03 = path->add_elem();
  pathelem03->set_name("CPU");
  auto pathelem04 = path->add_elem();
  pathelem04->set_name("CPUUsage");

  subscribeList->set_mode(::gnmi::SubscriptionList_Mode_POLL);
  subscribeList->set_encoding(::gnmi::JSON);
}

void gnmi_client::FillSubscribeRequestByStream(gnmi::SubscribeRequest &request)
{
  auto subscribeList =  request.mutable_subscribe();

  auto prefix = subscribeList->mutable_prefix();
  auto pathelem01 = prefix->add_elem();
  pathelem01->set_name("Diagnostic");

  auto subscribe = subscribeList->add_subscription();
```

```
  auto path = subscribe->mutable_path();
  auto pathelem02 = path->add_elem();
  pathelem02->set_name("CPUEvent");
  auto pathelem03 = path->add_elem();
  pathelem03->set_name("CPU");
   (*pathelem03->mutable_key())["Chassis#condition"] = "equal:1";
  subscribe->set_mode(::gnmi::ON_CHANGE);
  subscribe->set_sample_interval(1000);
  subscribe->set_suppress_redundant(false);
  subscribe->set_heartbeat_interval(1000);

  subscribeList->set_mode(::gnmi::SubscriptionList_Mode_STREAM);
  subscribeList->set_encoding(::gnmi::JSON);
}

void gnmi_client::FillSubscribeAlias(gnmi::SubscribeRequest &request)
{
  auto aliases = request.mutable_aliases();
  auto alias = aliases->add_alias();

  auto path = alias->mutable_path();
  auto pathelem01 = path->add_elem();
  pathelem01->set_name("Device");
  auto pathelem02 = path->add_elem();
  pathelem02->set_name("CPUs");
  auto pathelem03 = path->add_elem();
  pathelem03->set_name("CPU");
  auto pathelem04 = path->add_elem();
  pathelem04->set_name("CPUUsage");

  alias->set_alias("#cpu_usage");
}
```

(6)　调用 Logout 方法退出登录。

调用 Logout 方法的代码示例如下：

```
void GrpcServiceTest:: Logout ()
{
    LogoutRequest request;
    request.set_token_id(token);
    LogoutReply reply;
    ClientContext context;
    Status status = mStub->Logout(&context, request, &reply);
std::cout << "Logout! :" << reply.result() << std::endl;
}
```

5.5 gRPC Dial-in模式二次开发举例（GO）

5.5.1 生成代码

开发代码之前，需要用 protoc 工具软件将收集到的 Proto 文件转换成 GO 代码，并将生成的代码加入到开发的工程中。

本例中，开发 Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- 业务 Proto 文件，本例中为 Syslog.proto

本例中，开发 gNMI Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- gnmi.proto 和 gnmi_ext.proto

使用 protoc 工具生成 GO 代码的示例如下：

```
[root@ grpc]# cd protobuf
[root@ protobuf]# protoc --go_out=plugins=grpc:. grpc_service.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. Syslog.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. gnmi.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. gnmi_ext.proto
```

5.5.2 开发代码

1. 普通 Subscribe 操作

以调用 GrpcService 和 SyslogService 服务类为例，编码步骤如下：

(1) 编写 grpcConnect 方法，实现登录设备和退出登录设备。

代码示例如下：

a. 新建一个 sdk 包，以实现会话建立、连接和关闭。

```
import (
  "context"
  "fmt"
  h3c "github.com/gnmiTest/comwaresdk/cmwproto/grpc_service"  //grpc_service.proto
转换成 GO 语言的代码路径，可根据实际情况修改该路径
  "google.golang.org/grpc"
  md "google.golang.org/grpc/metadata"
  "log"
  "time"
)
type GrpcSession struct {
  Client h3c.GrpcServiceClient
  Conn   *grpc.ClientConn
  Token  string
}
// 建立连接
func NewClient(addr string, port uint, username string, password string) (*GrpcSession,
error) {
  address := fmt.Sprintf("%s:%d", addr, port)
```

```
  log.Printf("Server address: %v, UserName: %v, Password: %v\n", address, username,
password)

  // Set up a connection to the server.
  conn, err := grpc.Dial(address, grpc.WithInsecure())
  if err != nil {
    log.Fatalf("did not connect: %v", err)
    return nil, err
  }

  //create grpc_service client
  c := h3c.NewGrpcServiceClient(conn)

  //prepare context
  ctx, cancel := context.WithTimeout(context.Background(), time.Second*3)
  defer cancel()

  var token string

  loginReply, err := c.Login(ctx, &h3c.LoginRequest{UserName: &username, Password:
&password})
  if err != nil {
    log.Fatalf("could not login: %v", err)
    conn.Close()
    return nil, err
  }

  token = loginReply.GetTokenId()
  log.Printf("Token: %s", token)

  s := &GrpcSession{Client: c,
    Conn:  conn,
    Token: token}

  return s, nil
}
// 关闭连接
func (s *GrpcSession) Close() {
  var logoutReq = h3c.LogoutRequest{TokenId: &s.Token}
  ctx, cancel := CtxWithToken(s.Token, time.Second)
  defer cancel()
  s.Client.Logout(ctx, &logoutReq)
  s.Conn.Close()
  return
}
// 返回 Ctx 和 Token
func CtxWithToken(tk string, timeout time.Duration) (context.Context,
context.CancelFunc) {
```

```
  //Add token to meta data
  var mdata = md.Pairs("token_id", tk)
  ctx, cancel := context.WithTimeout(context.Background(), timeout)
  var ctx_with_token = md.NewOutgoingContext(ctx, mdata)
  return ctx_with_token, cancel
}
```

b. 创建一个 Dial-in 的 main 包（可自行定义包名）。

```
import(
  "flag"
  "io"
  "log"
  "fmt"
  "time"
  "github.com/gnmiTest/comwaresdk/sdk"  //sdk 包的存放路径，可根据实际代码存放路径修改
  syslog "github.com/gnmiTest/comwaresdk/cmwproto/syslog" //syslog.proto 代码转换成 GO 语言的代码路径，可根据实际情况修改该路径
  h3c "github.com/gnmiTest/comwaresdk/cmwproto/grpc_service"  //grpc_service.proto 转换成 GO 语言的代码路径，可根据实际情况修改该路径
)
var(
  address         string
  port            uint
  username        string
  password        string
  grpcSession*     sdk.GrpcSession
  isSyslog          bool
)

type grpcCon struct {
  grpcSession     *sdk.GrpcSession
  sysClient       syslog.SyslogServiceClient
  h3cClient       h3c.GrpcServiceClient
}

func grpcConnect() (*grpcCon, error) {
  grpcSession, err := sdk.NewClient(address, port, username , password)
  if err != nil {
      log.Println("Failed to open session.")
      return nil, err
  }
  sysClient := syslog.NewSyslogServiceClient(grpcSession.Conn)

  gt := grpcCon{
      grpcSession:    grpcSession,
      sysClient:      sysClient,
      h3cClient:      grpcSession.Client,
  }
  return &gt, err
```

```
}
```

(2) 发起对设备的 RPC 方法请求。

以订阅 Syslog/LOGEvent 事件为例，代码如下：

```
    rpc SubscribeLOGEvent(LOGEvent) returns (grpc_service.SubscribeReply) {}
```

(3) 编写方法 syslogSubscribeEvent 实现事件订阅和数据接收。

代码示例如下：

```
func (gt *grpcCon) syslogSubscribeEvent() error {
  streamName := "Syslog"
  mSubscribeRequest := syslog.EventStream{StreamName: &streamName}
  ctxWithToken, cancel := sdk.CtxWithToken(gt.grpcSession.Token, time.Second*100)
  defer cancel()
  sysReply, err := gt.sysClient.SubscribeEventStream(ctxWithToken,
&mSubscribeRequest)
  if err != nil {
      log.Println("syslog subscribe error: ", err)
      return err
  }
  // print subscribe result
  log.Printf("SyslogResult: \n %v", sysReply.GetResult())

  // receive subscribe result
  mGetReportquest := h3c.GetReportRequest{TokenId: &gt.grpcSession.Token}
  stream, err := gt.h3cClient.GetEventReport(ctxWithToken, &mGetReportquest)
     for {
      mSubscribeResponse, err := stream.Recv()

      if err == io.EOF {
           break
      }
      if err != nil {
           log.Println("Recv error: ", err)
           return err
      }else{
           fmt.Println("success:", mSubscribeResponse.GetJsonText())
      }
  }

  return nil
}
```

(4) 编写 main 函数，向设备发起 RPC 请求。

代码示例如下：

```
func test() error {
  con, err := grpcConnect()
     defer con.grpcSession.Close()
  if err != nil {
      return err
  }
```

```
  if isSyslog{
      err := con.syslogSubscribeEvent()
      if err != nil {
          return err
      }
  }
//参数解析
func init() {
  flag.StringVar(&address, "a", "192.168.2.1", "Address to comware")
  flag.UintVar(&port, "gp", 50051, "Grpc port of comware")
  flag.StringVar(&username, "u", "admin", "Username to comware")
  flag.StringVar(&password, "up", "123456", "Password to comware")
  flag.BoolVar(&isSyslog, "syslog", false, "dial in syslog event examlpe")
}
//main 函数
func main() {
  flag.Parse()
  test()
}
```

2. gNMI Subscribe 操作

gNMI Subscribe 操作的编码步骤如下：

(1) 编写 grpcConnect 方法，实现登录设备和退出登录设备。

代码示例如下：

a. 新建一个 sdk 包，以实现会话建立、连接和关闭。

SDK 包的创建方法请参见“1. 普通 Subscribe 操作”。

b. 创建一个 Dial-in 的 main 包（可自行定义包名）。

```
import(
  "flag"
  "io"
  "log"
  "fmt"
  "time"
  "github.com/gnmiTest/comwaresdk/sdk"  //sdk 包的存放路径，可根据实际情况修改该路径
  "github.com/gnmiTest/comwaresdk/cmwproto/gnmi"  //gnmi.proto 转换成 GO 语言的代码路径，可根据实际代码存放路径修改
  h3c "github.com/gnmiTest/comwaresdk/cmwproto/grpc_service"  //grpc_service.proto 转换成 GO 语言的代码路径，可根据实际代码存放路径修改
)
type grpcCon struct {
  gnmiClient      gnmi.GNMIClient
  grpcSession     *sdk.GrpcSession
}

func grpcConnect() (*grpcCon, error) {
  grpcSession, err := sdk.NewClient(address, port, username , password)
  if err != nil {
      log.Println("Failed to open session.")
```

```
        return nil, err
    }

    gnmiClient := gnmi.NewGNMIClient(grpcSession.Conn)
    gt := grpcCon{
        gnmiClient: gnmiClient,
        grpcSession:    grpcSession,
    }
    return &gt, err
}
```

(2) 发起对设备的 RPC 方法请求。

代码示例如下：

```
rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
```

(3) 实现自定义的订阅方法。

代码示例如下：

```
//once 模式订阅
func (gt *grpcCon) gnmiSubscribeOnceTest() error {

    mSubscribeRequest := gnmi.SubscribeRequest{
        Request: &gnmi.SubscribeRequest_Subscribe{
            Subscribe: &gnmi.SubscriptionList{
                Prefix: &gnmi.Path{
                    Elem: []*gnmi.PathElem{
                        {Name: "Device"},
                    },
                },
                Subscription: []*gnmi.Subscription{
                    {Path: &gnmi.Path{
                        Elem: []*gnmi.PathElem{{Name: "Base"}},
                    },},
                },
                Mode: gnmi.SubscriptionList_ONCE,
                Encoding: gnmi.Encoding_JSON,
            },
        },
    }

    ctxWithToken, cancel := sdk.CtxWithToken(gt.grpcSession.Token, time.Second*10)
    defer cancel()
    stream, err := gt.gnmiClient.Subscribe(ctxWithToken)
    if err != nil {
        log.Println("Subscribe error: ", err)
        return err
    }
    stream.Send(&mSubscribeRequest)
    for {
        mSubscribeResponse, err := stream.Recv()
```

```
            if err == io.EOF {
            break
        }
        if err != nil {
            log.Println("Recv error: ", err)
            return err
        }else{
            fmt.Println("success:", mSubscribeResponse)
        }
    }
    return nil
}
//stream 模式订阅
func (gt *grpcCon) gnmiSubscribeStreamTest() error {
    mSubscribeRequest := gnmi.SubscribeRequest{
        Request: &gnmi.SubscribeRequest_Subscribe{
            Subscribe: &gnmi.SubscriptionList{
                 Prefix: &gnmi.Path{
                    Elem: []*gnmi.PathElem{
                        // {Name: "Ifmgr"}, {Name: "Interfaces"},
                        {Name: "Device"},
                    },
                 },
                 Subscription: []*gnmi.Subscription{
                    {Path: &gnmi.Path{
                         // Elem: []*gnmi.PathElem{{Name: "Interface", Key:
map[string]string{"IfIndex": "2",}},{Name: "ConfigDuplex"}},
                         Elem: []*gnmi.PathElem{{Name: "Base"}},
                     },
                     Mode: gnmi.SubscriptionMode_TARGET_DEFINED,
                     SampleInterval: uint64(1000000000),
                     SuppressRedundant: false,
                     HeartbeatInterval: uint64(1000),},
                },
                Mode: gnmi.SubscriptionList_STREAM,
                Encoding: gnmi.Encoding_JSON,
            },
         },
    }

    ctxWithToken, cancel := sdk.CtxWithToken(gt.grpcSession.Token, time.Second*10)
    defer cancel()
    stream, err := gt.gnmiClient.Subscribe(ctxWithToken)
    if err != nil {
        log.Println("Subscribe error: ", err)
        return err
    }
    stream.Send(&mSubscribeRequest)
    for {
```

```
        mSubscribeResponse, err := stream.Recv()

        if err == io.EOF {
            break
        }
        if err != nil {
            log.Println("Recv error: ", err)
            return err
        }else{
            fmt.Println("success:", mSubscribeResponse)
        }
    }
    return nil
}
//poll 模式订阅
func (gt *grpcCon) gnmiSubscribePollTest() error {
    mSubscribeRequest := gnmi.SubscribeRequest{
        Request: &gnmi.SubscribeRequest_Subscribe{
            Subscribe: &gnmi.SubscriptionList{
                Prefix: &gnmi.Path{
                    Elem: []*gnmi.PathElem{
                        {Name: "Device"},
                    },
                },
                Subscription: []*gnmi.Subscription{
                    {Path: &gnmi.Path{
                        Elem: []*gnmi.PathElem{{Name: "Base"}},
                    },},
                },
                Mode: gnmi.SubscriptionList_POLL,
                Encoding: gnmi.Encoding_JSON,
            },
        },
    }

    mPoll := gnmi.SubscribeRequest{
        Request: &gnmi.SubscribeRequest_Poll{
            Poll: &gnmi.Poll{
            },
        },
    }

    ctxWithToken, cancel := sdk.CtxWithToken(gt.grpcSession.Token, time.Second*10)
    defer cancel()
    stream, err := gt.gnmiClient.Subscribe(ctxWithToken)
    if err != nil {
        log.Println("Subscribe error: ", err)
        return err
```

```
  }
  stream.Send(&mSubscribeRequest)
  for {
      time.Sleep(time.Second)
      stream.Send(&mPoll)
      mSubscribeResponse, err := stream.Recv()

      if err == io.EOF {
          break
      }
      if err != nil {
          log.Println("Recv error: ", err)
          return err
      }else{
          fmt.Println("success:", mSubscribeResponse)
      }
  }
  return nil
}
```

(4) 编写 main 函数，向设备发起 RPC 请求。

代码示例如下：

```
func test() error {
  con, err := grpcConnect()
  defer con.grpcSession.Close()
  if err != nil {
      return err
  }

  if isSub{
      err := con.gnmiSubscribeOnceTest()
      if err != nil {
          return err
      }
  }
 return nil
}
//参数解析
func init() {
  flag.StringVar(&address, "a", "192.168.2.1", "Address to comware")
  flag.UintVar(&port, "gp", 50051, "Grpc port of comware")
  flag.StringVar(&username, "u", "admin", "Username to comware")
  flag.StringVar(&password, "up", "123456", "Password to comware")
  flag.BoolVar(&isSub, "sub", false, "gnmi subscribe examlpe")
}
//main 函数
func main() {
  flag.Parse()
  test()
```

```
}
```

5.6 gRPC Dial-in模式二次开发举例（Python）

5.6.1 生成代码

开发代码之前，需要使用工具软件 protoc 将收集到的 Proto 文件转换成 Python 代码，并将生成的代码加入到开发的工程中。

本例中，开发 Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- 业务 Proto 文件，本例中为 Syslog.proto

本例中，开发 gNMI Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- gnmi.proto 和 gnmi_ext.proto

使用 protoc 工具生成 Python 代码的示例如下：

```
[root@ grpc]# cd protobuf
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
grpc_service.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
Syslog.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
gnmi.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
gnmi_ext.proto
```

5.6.2 开发代码

1. 普通 Subcribe 操作

以调用 GrpcService 和 SyslogService 服务类为例，编码步骤如下：

(1) 发起对设备的 RPC 方法请求。

以订阅日志触发事件为例，代码如下：

```
rpc SubscribeLOGEvent(LOGEvent) returns (grpc_service.SubscribeReply) {}
```

(2) 编写一个 Client 类，实现登录设备和退出登录设备。

代码示例如下：

```
import grpc
import json
from collections import OrderedDict
import grpc_service_pb2, grpc_service_pb2_grpc
import Syslog_pb2, Syslog_pb2_grpc
//Client 客户端编程类
class Client:
    def __init__(self, username, password, channel):
        self.username = username
        self.password = password
        self.channel = channel
```

```
        self.__stub = grpc_service_pb2_grpc.GrpcServiceStub(channel)
        self.tokenid = ""

    def __enter__(self):
        return

    def __exit__(self, exc_type, exc_value, traceback):
        if self.tokenid:
            self.Logout()

    def __str__(self):
        return "{username=%s, password=%s, tokenid=%s}" % (self.username, self.password,
self.tokenid)

    def metadata(self):
        return (("token_id", self.tokenid), )
    //Login 函数，调用 grpc_service_pb2.LoginRequest 函数，完成登录设备
    def Login(self):
        if self.tokenid:
            return self
        request = grpc_service_pb2.LoginRequest(user_name=self.username,
password=self.password)
        reply = self.__stub.Login(request)
        self.tokenid = reply.token_id
        return self
    //Logout 函数，调用 grpc_service_pb2.LogoutRequest 函数，完成登出设备
    def Logout(self):
        if not self.tokenid:
            return
        request = grpc_service_pb2.LogoutRequest(token_id=self.tokenid)
        try:
            self.__stub.Logout(request)
        except Exception as e:
            logging.warning("Logout:" + e)
        self.tokenid = ""
        return
    //订阅事件流
    def SubscribeByStreamName(self, stream):
        request = grpc_service_pb2.SubscribeRequest(stream_name=stream)
        reply = self.__stub.SubscribeByStreamName(request, metadata=self.metadata())
        return reply.result
    //获取事件上报结果
    def GetEventReport(self):
        request = grpc_service_pb2.GetReportRequest(token_id=self.tokenid)
        yield from self.__stub.GetEventReport(request)
    //订阅 Syslog logevent 事件
    def sub(self, path):
        if path ==  "SubscribeLOGEvent":
           request = Syslog_pb2.LOGEvent()
```

```
            RpcMethod = Syslog_pb2_grpc.SyslogServiceStub(self.channel)
            reply = RpcMethod.SubscribeLOGEvent(request, metadata=self.metadata())
            return reply.result
```

(3) 编写 main 函数，下发订阅并且接收订阅。

代码示例如下：

```
//格式化 Json 字符串
def format_json(jsonstr):
    obj = json.loads(jsonstr, object_hook=OrderedDict)
    return json.dumps(obj, ensure_ascii=False, indent=4)
//客户端订阅事件并接受事件
def test():
    channel = grpc.insecure_channel("192.168.2.1:50051")
    client = Client("admin", "123456", channel)
    with client.Login():
        print(client)
        print(client.sub("SubscribeLOGEvent"))
        for e in client.GetEventReport():
            print(e)
            print(format_json(e.json_text))
//main 函数
if __name__ == "__main__":
test()
```

2. gNMI Subscribe 操作

gNMI Subscribe 操作的编码步骤如下：

(1) 继承原有的 gRPC Client 类（请参见“5.6.2 1. (2)”中的 Client 类），并定义 gNMI RPC 方法。

代码示例如下：

```
from dialin import Client
import gnmi_pb2, gnmi_pb2_grpc
import grpc
import time

class GnmiClient(Client):
    @staticmethod
    def get_mode(s:str):
        mapping = {
            "stream": gnmi_pb2.SubscriptionList.Mode.STREAM,
            "once": gnmi_pb2.SubscriptionList.Mode.ONCE,
            "poll": gnmi_pb2.SubscriptionList.Mode.POLL,
            "target_defined": gnmi_pb2.TARGET_DEFINED,
            "on_change": gnmi_pb2.ON_CHANGE,
            "sample": gnmi_pb2.SAMPLE,
        }
        return mapping[s.lower()]

    def __init__(self, username, password, channel):
```

```
        super().__init__(username, password, channel)
        self.__stub = gnmi_pb2_grpc.gNMIStub(channel)
        return

    def make_poll_req(self):
        return gnmi_pb2.SubscribeRequest(poll=gnmi_pb2.Poll())

    def Subscribe(self, req):
        return self.__stub.Subscribe(req, metadata=self.metadata())

    def make_sub_obj(self,interval):
        path_obj = gnmi_pb2.Path()
        ele = path_obj.elem.add()
        ele.name = "Device"
        ele1 = path_obj.elem.add()
        ele1.name = "Base"
        mode = self.get_mode("sample")
        sample_interval = interval
        return gnmi_pb2.Subscription(path=path_obj, mode=mode,
sample_interval=sample_interval)

    def make_sub_eventobj(self,interval,mode= "on_change"):
        path_obj = gnmi_pb2.Path()
        ele = path_obj.elem.add()
        ele.name = "Ifmgr"
        ele1 = path_obj.elem.add()
        ele1.name = "InterfaceEvent"
        mode = self.get_mode("on_change")
        sample_interval = interval
        return gnmi_pb2.Subscription(path=path_obj, mode=mode,
sample_interval=sample_interval)

    def make_sub_req(self, sample_interval , mode, type ="sample",qos=0,
updates_only=False):
        kwargs = {}
        kwargs["prefix"] = gnmi_pb2.Path()
        kwargs["qos"] = gnmi_pb2.QOSMarking(marking=qos)
        kwargs["mode"] = self.get_mode(mode)
        kwargs["encoding"] = gnmi_pb2.JSON
        kwargs["subscription"] = []
        if type == "sample":
            kwargs["subscription"].append(self.make_sub_obj(sample_interval))
        else:
            kwargs["subscription"].append(self.make_sub_eventobj(sample_interval))
        kwargs["updates_only"] = updates_only
        subscribeList = gnmi_pb2.SubscriptionList(**kwargs)
        return gnmi_pb2.SubscribeRequest(subscribe=subscribeList)
```

(2) 发起对设备的 RPC 方法请求。

代码示例如下：

```
rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
```

(3)　自定义订阅的实现方法。

代码示例如下：

```
def test_sub(client:GnmiClient):
    def poll_generator(n):
        req = client.make_sub_req(2000000000, mode="poll")
        yield req
        for _ in range(n):
            time.sleep(2)
            yield client.make_poll_req()
        return

    def sample_generator(t, sample_interval=2000000000, mode="stream"):
        req = client.make_sub_req(sample_interval,mode)
        yield req
        time.sleep(t)
        return

    def event_generator(t):
        req = client.make_sub_req(type ="event", sample_interval=10000000, mode="stream")
        yield req
        time.sleep(t)
        return
```

(4)　编写 main 函数，向设备下发订阅并且接收设备推送的采样数据。

代码示例如下：

```
if __name__ == "__main__":
    channel = grpc.insecure_channel("192.168.2.1:50051")
    client = GnmiClient("admin", "123456", channel)
    with client.Login():
        test_sub(client)
```

5.7 gRPC Dial-in模式二次开发举例（JAVA）

5.7.1 生成代码

开发代码之前，需要开发人员先安装好 maven 环境，并在创建 maven 工程后修改 pom.xml，实现下载 protoc 工具的步骤如下：

```
<plugins>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId>
                <version>0.6.1</version>
                <configuration>

<protocArtifact>com.google.protobuf:protoc:3.6.1:exe:${os.detected.classifier}</protocAr
tifact>
```

```
                    <pluginId>grpc-java</pluginId>

<pluginArtifact>io.grpc:protoc-gen-grpc-java:1.14.0:exe:${os.detected.classifier}</plugi
nArtifact>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
                            <goal>compile-custom</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
```

然后，开发人员在 maven 环境中运行 mvn install 即可将收集到的 Proto 文件生成对应的 JAVA 代码。

本例中，开发 Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- 业务 Proto 文件，本例中为 Syslog.proto

本例中，开发 gNMI Subscribe 操作的代码使用以下 Proto 文件：

- grpc_service.proto
- gnmi.proto 和 gnmi_ext.proto

5.7.2 开发代码

1. 普通 Subcribe 操作

以调用 GrpcService 和 SyslogService 服务类为例，编码步骤如下：

(1) 编写一个 DialinClient 类，完成登录设备和退出登录设备。

代码示例如下：

```
public class DialinClient {
    private static final Logger logger = Logger.getLogger(DialinClient.class.getName());
    private final GrpcServiceGrpc.GrpcServiceBlockingStub blockingStub;
    private String tokenid;

    public DialinClient(Channel channel) {
        blockingStub = GrpcServiceGrpc.newBlockingStub(channel);
        tokenid = "";
    }
//设备登录
    public void login(String username, String password) throws Exception {
        logger.info("try to login as " + username + "...");
        LoginRequest request =
LoginRequest.newBuilder().setUserName(username).setPassword(password).build();
        LoginReply loginReply;
        try {
            loginReply = blockingStub.login(request);
            tokenid = loginReply.getTokenId();
```

```
        } catch (StatusRuntimeException e) {
            logger.log(Level.WARNING, "RPC failed: {0}",e.getStatus());
            throw e;
        }
        System.out.println("Login : " + getTokenid());
    }

    public  String getTokenid() {
        return tokenid;
    }
//设备获取订阅上报结果
    public Iterator  getEventReport(String tokenid) {
        GrpcServiceOuterClass.GetReportRequest request =
GrpcServiceOuterClass.GetReportRequest.newBuilder().setTokenId(tokenid).build();
        Iterator< GrpcServiceOuterClass.ReportEvent> eventIterator = null;
        try {
            eventIterator = blockingStub.getEventReport(request);
        } catch (Exception ignored) {

        }
        return  eventIterator;
    }
//退出设备登录
    public void logout() {
        LogoutRequest request = LogoutRequest.newBuilder().setTokenId(tokenid).build();
        LogoutReply logoutReply = null;
        try {
            logoutReply = blockingStub.logout(request);
            tokenid = "";
        } catch (Exception ignored) {

        }
        if(logoutReply != null) {
            System.out.println("Logout result: " + logoutReply.getResult());
        }
        return;
    }
}
```

(2)　发起对设备的 RPC 方法请求。

以订阅日志触发事件为例，代码如下：

```
rpc SubscribeLOGEvent(LOGEvent) returns (grpc_service.SubscribeReply) {}
```

(3)　编写一个 SyslogClient 类来封装发起的 RPC 方法。

代码示例如下：

```
public class SyslogClient {
    private static final Logger logger = Logger.getLogger(SyslogClient.class.getName());
    private SyslogServiceGrpc.SyslogServiceBlockingStub blockingStub;
    private String tokenid;
```

```
    public SyslogClient(Channel channel, String tokenid) {
        blockingStub = SyslogServiceGrpc.newBlockingStub(channel);
        this.tokenid = tokenid;
    }

    public String getTokenid() {
        return tokenid;
    }

    public void setTokenid(String tokenid) {
        this.tokenid = tokenid;
    }

    public String subLogEvent() throws Exception {
        Syslog.LOGEvent requset =
Syslog.LOGEvent.newBuilder().addLog(Syslog.LOGEvent.LOG.newBuilder()).build();
        Metadata header = new Metadata();
        Metadata.Key<String> key = Metadata.Key.of("token_id",
Metadata.ASCII_STRING_MARSHALLER);
        header.put(key, getTokenid());
        SyslogServiceGrpc.SyslogServiceBlockingStub blockingStub_tmp =
MetadataUtils.attachHeaders(blockingStub, header);
        GrpcServiceOuterClass.SubscribeReply subscribeReply;
        try {
            subscribeReply = blockingStub_tmp.subscribeLOGEvent(requset);
        }catch (StatusRuntimeException e) {
            logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
            throw e;
        }
        return subscribeReply.getResult();
    }

}
```

(4) 编写 main 函数实现事件的订阅。

代码示例如下：

```
public class Main {

    private static String ipPort = "192.168.2.1:50051";
    private static String usrname = "admin";
    private static String password = "123456";
//用户输入 IP 地址，用户名，密码
    public static ArrayList UserInput( ){
        System.out.println("Input ipaddress:port :");
        Scanner scanner = new Scanner(System.in);
        String str = "";
        ArrayList<String> stringArrayList = new ArrayList<String>();
        for (int i = 0;i < 3; i++){
```

```
            str = scanner.nextLine();
            stringArrayList.add(str);
            if(i == 0) {
                System.out.println("Input UserName :");
            }else if (i == 1) {
                System.out.println("Input PassWord :");
            }
        }
        scanner.close();
        return stringArrayList;
    }
//调用 SyslogClient 类，实现对 syslog/logevent 订阅
    public static void testSub (String tokenid) {
        long begin=0, end=0;
        int i;
        String s;
        ManagedChannel channel = null;
        try {
            channel = ManagedChannelBuilder.forTarget(ipPort).usePlaintext().build();
            SyslogClient syslogClient = new SyslogClient(channel, tokenid);
            begin = System.currentTimeMillis();
            s = syslogClient.subLogEvent();
            System.out.println(s);
            end = System.currentTimeMillis();
            System.out.printf("cost %dms\n", end-begin);
        } catch (Exception e) {

        }

    }
//调用 DialinClient 类，登录/订阅/登出设备
    public static void testDialin() throws  Exception {
        DialinClient client = null;
        ArrayList<String> arrayList = UserInput();
        ipPort = arrayList.get(0);
        usrname = arrayList.get(1);
        password = arrayList.get(2);
        ManagedChannel channel =
ManagedChannelBuilder.forTarget(ipPort).usePlaintext().build();
        String tokenID = "";
        try {
            client = new DialinClient(channel);
            client.login(usrname, password);
            tokenID = client.getTokenid();
            testSub(tokenID);
            Iterator<GrpcServiceOuterClass.ReportEvent> eventIterator =
client.getEventReport(tokenID);
            while (eventIterator.hasNext()) {
                System.out.println(eventIterator.next());
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            System.out.println("created ");
            if (client != null) {
                client.logout();
            }
        }
}
//main 函数
    public static void main (String[] agrs) throws Exception {
            testDialin();
    }

}
```

2. gNMI Subscribe 操作

gNMI Subscribe 操作的编写步骤如下：

(1) 编写一个 DialinClient 类，完成登录设备和退出设备登录。

步骤与“5.7.2 1.”中编写 DialinClient 类相同。

(2) 发起对设备的 RPC 方法请求。

代码示例如下：

```
rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
```

(3) 编写 gNMIClient 类，并在该类中实现自定义订阅的方法。

代码示例如下：

```
public class gNMIClient {
    private static final Logger logger = Logger.getLogger(gNMIClient.class.getName());
    private gNMIGrpc.gNMIStub gNMIStub;
    private final String tokenId;

    public gNMIClient(Channel channel, String tokenId) {
        gNMIStub = gNMIGrpc.newStub(channel);
        this.tokenId = tokenId;
    }
    //gNMI 订阅请求消息拼装
    public Gnmi.SubscribeRequest gnmiSubRequest(Gnmi.Path prefix, Gnmi.Path paths,
                                                Gnmi.SubscriptionList.Mode mode,
Gnmi.SubscriptionMode subscriptionMode,
                                                int sample_interval) {
        Gnmi.Subscription.Builder subBuilder = Gnmi.Subscription.newBuilder();
        subBuilder.setPath(paths);
        if (subscriptionMode != null) {
            subBuilder.setMode(subscriptionMode);
        }
        if (sample_interval != 0)
```

```
        {
            subBuilder.setSampleInterval(sample_interval);
        }
        Gnmi.Subscription subscription = subBuilder.build();
        Gnmi.SubscriptionList.Builder subList = Gnmi.SubscriptionList.newBuilder();
        if (prefix != null) {
            subList.setPrefix(prefix);
        }
        subList.addSubscription(subscription);
        if (mode != null) {
            subList.setMode(mode);
        }
        subList.setEncoding(Gnmi.Encoding.forNumber(0));
        Gnmi.SubscriptionList subListReq = subList.build();
        Gnmi.SubscribeRequest.Builder requestBuilder =
Gnmi.SubscribeRequest.newBuilder();
        requestBuilder.setSubscribe(subListReq);
        Gnmi.SubscribeRequest request = requestBuilder.build();
        return request;
    }
    //poll 模式请求订阅消息
    public Gnmi.SubscribeRequest gnmiSubRequestByPoll (Gnmi.Path prefix, Gnmi.Path
paths,Gnmi.SubscriptionList.Mode mode) {
        return gnmiSubRequest(prefix, paths,mode, null, 0);
    }
    //once 模式请求订阅消息
    public  Gnmi.SubscribeRequest gnmiSubRequestByOnce(Gnmi.Path prefix, Gnmi.Path
paths,Gnmi.SubscriptionList.Mode mode) {
        return gnmiSubRequest(prefix, paths,mode, null, 0);
    }
    //stream 模式请求订阅消息
    public  Gnmi.SubscribeRequest gnmiSubRequestByStream(Gnmi.Path prefix, Gnmi.Path
paths,Gnmi.SubscriptionList.Mode mode) {
        return gnmiSubRequest(prefix, paths,mode, Gnmi.SubscriptionMode.forNumber(2),
2000000000);
    }
   //once 模式订阅
    public void testOnce(StreamObserver<Gnmi.SubscribeRequest> requestStreamObserver,
Gnmi.Path prefix, Gnmi.Path paths) {
        Gnmi.SubscribeRequest request = gnmiSubRequestByOnce(prefix, paths,
Gnmi.SubscriptionList.Mode.forNumber(1));
        requestStreamObserver.onNext(request);
    }
    //poll 模式订阅
    public void testPoll(StreamObserver<Gnmi.SubscribeRequest> requestStreamObserver,
Gnmi.Path prefix, Gnmi.Path paths) {
        Gnmi.SubscribeRequest request = gnmiSubRequestByPoll(prefix, paths,
Gnmi.SubscriptionList.Mode.forNumber(2));
        requestStreamObserver.onNext(request);
        for (int i = 0; i < 5; i++) {
```

```
            Gnmi.SubscribeRequest poll =
Gnmi.SubscribeRequest.newBuilder().setPoll(Gnmi.Poll.newBuilder()).build();
            requestStreamObserver.onNext(poll);
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
   //stream 模式订阅
    public void testStream(StreamObserver<Gnmi.SubscribeRequest> requestStreamObserver,
Gnmi.Path prefix, Gnmi.Path paths) {
        Gnmi.SubscribeRequest request = gnmiSubRequestByStream(prefix, paths,
Gnmi.SubscriptionList.Mode.forNumber(0));
        requestStreamObserver.onNext(request);
    }
    //下发订阅请求，并接收上报结果
    public void testSubscribe(String name) {
        Metadata header = new Metadata();
        Metadata.Key<String> key = Metadata.Key.of("token_id",
Metadata.ASCII_STRING_MARSHALLER);
        header.put(key,tokenId);
        gNMIGrpc.gNMIStub stub = MetadataUtils.attachHeaders(gNMIStub, header);
        Gnmi.Path.Builder pathBuilder = Gnmi.Path.newBuilder();
        String[] names = name.split("/");
        for (String na:names) {
            pathBuilder.addElem(Gnmi.PathElem.newBuilder().setName(na));
        }
        Gnmi.Path path = pathBuilder.build();

        StreamObserver<Gnmi.SubscribeResponse> observer = new
StreamObserver<Gnmi.SubscribeResponse>() {
            @Override
            public void onNext(Gnmi.SubscribeResponse subscribeResponse) {
                if (subscribeResponse != null) {
                    System.out.println(subscribeResponse.getUpdate());
                }
            }

            @Override
            public void onError(Throwable throwable) {
                logger.log(Level.WARNING, "Subscribe is failed");
                throwable.printStackTrace();
                System.out.println(throwable);
            }

            @Override
```

```
            public void onCompleted() {
                System.out.println("onCompleted");
            }
        };

        StreamObserver<Gnmi.SubscribeRequest> requestStreamObserver =
stub.subscribe(observer);
        //三种不同模式订阅
        //testOnce(requestStreamObserver,null,path);
        //testPoll(requestStreamObserver,null,path);
        //testStream(requestStreamObserver,null,path);

        try {
            Thread.sleep(50000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        requestStreamObserver.onCompleted();
    }
}
```

(4) 编写 main 函数实现事件的订阅。

代码示例如下：

```
public class Main {
    private static String ipPort = "192.168.2.1:50051";
    private static String usrname = "admin";
    private static String password = "123456";

    public static ArrayList UserInput( ){
        System.out.println("Input ipaddress:port :");
        Scanner scanner = new Scanner(System.in);
        String str = "";
        ArrayList<String> stringArrayList = new ArrayList<String>();
        for (int i = 0;i < 3; i++){
            str = scanner.nextLine();
            stringArrayList.add(str);
            if(i == 0) {
                System.out.println("Input UserName :");
            }else if (i == 1) {
                System.out.println("Input PassWord :");
            }
        }
        scanner.close();
        return stringArrayList;
    }
    //下发订阅，此处以 Device/Base 为例
    public static void testGnmi() throws  Exception{
        DialinClient dialinClient = null;
        ArrayList<String> arrayList = UserInput();
```

```
        ipPort = arrayList.get(0);
        usrname = arrayList.get(1);
        password = arrayList.get(2);
        ManagedChannel channel =
ManagedChannelBuilder.forTarget(ipPort).usePlaintext().build();

        try {
            dialinClient = new DialinClient(channel);
            dialinClient.login(usrname, password);
            gNMIClient client = new gNMIClient(channel, dialinClient.getTokenid());
            client.testSubscribe("Device/Base");
        } finally {
            dialinClient.logout();
        }

    }

    public static void main (String[] agrs) throws Exception {
            testGnmi();
    }

}
```

6 Telemetry 对接软件二次开发举例（Dial-out 模式）

说明

本指南中展示的代码仅供参考，由于没有实际的代码框架，在实际的对接过程中不能直接使用。除举例中生成的代码之外，其它代码还需要开发人员自行开发。

对于 Dial-out 模式，主要是实现 gRPC 服务端的代码，使采集器能够接收设备推送的采集数据并进行解析。客户端代码主要包括以下两个部分：

- 继承自动生成的 GRPCDialout::Service 类，重载自动生成的 RPC 服务 Dialout，并完成解析，获得相应字段内容。
- 将 RPC 服务注册到指定的监听端口上。

本章节用于介绍如下三种 Dial-out 模式下，设备与对接服务端软件的开发过程：

- 二层普通 Dial-out 模式
- 三层 Dial-out 模式
- 二层 gNMI Dial-out 模式

6.1 开发前准备

6.1.1 开发人员要求

- 用户熟悉 gRPC 的开发（可通过 https://doc.oschina.net/grpc 学习）。
- 用户熟悉 GPB 编码的开发（可通过 https://developers.google.com/protocol-buffers 学习）。
- 用户熟悉对应语言（C++、JAVA、Python、GO）的开发。

6.1.2 开发环境准备

1. 获取 Proto 文件

联系 H3C 技术支持人员获取相关 Proto 文件。

2. 获取处理 proto 文件的工具软件 protoc

下载地址：https://github.com/google/protobuf/releases

3. 获取对应开发语言的 protobuf 插件

下载地址：https://github.com/google/protobuf/releases

开发者需要准备好对应语言的开发环境，例如 C++插件 protobuf-cpp。本指南会给出当前主流语言的开发举例（C++、JGO、Python、JAVA）。

4. 存放 Proto 文件和工具软件

建议将获取到的 Proto 文件和 protoc 工具存在开发代码的工程目录下，具体以实际使用的开发环境为准。

6.2 组网需求

如图 6-1 所示，设备作为 gRPC 客户端与采集器相连，采集器为 gRPC 服务器，接收数据的端口号为 50051。

通过配置 gRPC Dial-out 模式，使设备以 5 秒为周期向采集器推送接口模块的设备能力信息。

图6-1 gRPC Dial-out 模式配置组网图

6.3 配置设备侧的Telemetry订阅

执行以下配置之前，请确保 gRPC 服务器与 gRPC 客户端之间路由可达。

开启 gRPC 功能。

```
<Device> system-view
[Device] grpc enable
```

创建传感器组 test，并添加采样路径为 ifmgr/devicecapabilities。

```
[Device] telemetry
[Device-telemetry] sensor-group test
[Device-telemetry-sensor-group-test] sensor path ifmgr/devicecapabilities
[Device-telemetry-sensor-group-test] quit
```

创建目标组 collector1，并配置采集器的 IP 地址为 2.2.2.2、端口号为 50051。

```
[Device-telemetry] destination-group collector1
[Device-telemetry-destination-group-collector1] ipv4-address 2.2.2.2 port 50051
[Device-telemetry-destination-group-collector1] quit
```

创建订阅 A，配置关联传感器组为 test，数据采样和推送周期为 5 秒，关联目标组为 collector1。

```
[Device-telemetry] subscription A
[Device-telemetry-subscription-A] sensor-group test sample-interval 5
[Device-telemetry-subscription-A] destination-group collector1
[Device-telemetry-subscription-A] quit
```

6.4 gRPC Dial-out模式二次开发举例（C++）

6.4.1 生成代码

开发代码之前，需要使用 protoc 工具将收集到的 Proto 文件转换成 C++代码，并将生成的代码加入到开发的工程中。

本例中，开发二层普通 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout.proto

本例中，开发三层 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout_v3.proto
- telemetry.proto
- 业务 Proto 文件，本例中为 Ifmgr_v3.proto

本例中，开发二层 gNMI Dial-out 模式的代码使用以下 Proto 文件：

- dialout.proto
- gnmi.proto 和 gnmi_ext.proto

使用 protoc 工具生成 C++代码的示例如下：

```
$ protoc --plugin=./grpc_cpp_plugin  --grpc_out=.  --cpp_out=. *.proto
```

6.4.2 开发代码

1. 二层普通 Dial-out 模式

编码步骤如下：

(1) 继承并重载 RPC 服务 Dialout。

新建一个类 DialoutTest 并继承 GRPCDialout::Service，代码示例如下：

```
class DialoutTest final : public GRPCDialout::Service { //重载自动生成的抽象类
Status Dialout(ServerContext* context, ServerReader< DialoutMsg>* reader, DialoutResponse*
response) override; //实现 Dialout RPC 方法
};
```

(2) 将 DialoutTest 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
using grpc::Server;
using grpc::ServerBuilder;
std::string server_address("0.0.0.0:50051"); //指定要监听的地址和端口
DialoutTest dialout_test; //定义（1）中声明的对象
ServerBuilder builder;
builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());//添加监听
builder.RegisterService(&dialout_test); //注册服务
std::unique_ptr<Server> server(builder.BuildAndStart()); //启动服务
server->Wait();
```

(3) 实现 Dialout 方法，实现数据解析。

代码示例如下：

```
Status DialoutTest::Dialout(ServerContext* context, ServerReader< DialoutMsg>* reader,
DialoutResponse* response)
{
        DialoutMsg msg;

        while( reader->Read(&msg))
        {
            const DeviceInfo &device_msg = msg.devicemsg();
            std::cout<< "Producer-Name: " << device_msg.producername() << std::endl;
            std::cout<< "Device-Name: " << device_msg.devicename() << std::endl;
            std::cout<< "Device-Model: " << device_msg.devicemodel() << std::endl;
            std::cout<<"Sensor-Path: " << msg.sensorpath()<<std::endl;
            std::cout<<"Json-Data: " << msg.jsondata()<<std::endl;
            std::cout<<std::endl;
        }
        response->set_response("test");

        return Status::OK;
}
```

(4) 通过 Read 方法获取到 Proto 文件生成的 DialoutMsg 对象后，可以调用对应的方法获取相应的字段值。

2. 三层 Dial-out 模式

编码步骤如下：

(1) 继承并重载 RPC 服务 Dialout。

新建一个类 DialoutV3Test 并继承 DialoutV3::Service，代码示例如下：

```
class DialoutV3Testfinal : public DialoutV3::Service { //重载自动生成的抽象类
Status DialoutV3Test::DialoutV3(::grpc::ServerContext*
context, ::grpc::ServerReaderWriter< ::grpc_dialout_v3::DialoutV3Args, ::grpc_dialout_v3
::DialoutV3Args>* stream)override; //实现 Dialout V3 RPC 方法
};
```

(2) 将 DialoutV3Test 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
    string server_address("0.0.0.0.101:50051");
    DialoutV3Test dialout_test;
```

```
    ServerBuilder builder;
    cout << "runing on " << server_address << endl;
    builder.AddListeningPort(server_address, InsecureServerCredentials());
    builder.RegisterService(&dialout_test);
    unique_ptr<Server> server(builder.BuildAndStart());
    server->Wait();
```

(3) 实现 DialoutV3 方法，实现数据解析。

代码示例如下：

```
Status DialoutV3Test::DialoutV3(::grpc::ServerContext*
context, ::grpc::ServerReaderWriter< ::grpc_dialout_v3::DialoutV3Args, ::grpc_dialout_v3
::DialoutV3Args>* stream)
{
DialoutV3Args msg;
    Telemetry msgTele;
    TelemetryGPBTable msgTable;
    TelemetryRowGPB msgRow;
    string buffdata = "";
    int buffsize = 0;
    string content;
    std::cout << "peer info : " << context->peer() << std::endl;
    while(stream->Read(&msg))
    {
        int row_size = 0;
        int64_t ReqId = msg.reqid();
        string data = msg.data();
        string Err = msg.errors();
        int32_t TotalSize = msg.totalsize();
        buffdata = buffdata + data;
        buffsize = buffsize + data.size();
        if(buffsize >= TotalSize)
        {
        std::cout << "ReqId : " << ReqId << std::endl;
        std::cout << "errors: " << Err << std::endl;
        std::cout << "totalSize: " << TotalSize << std::endl;
        msgTele.ParseFromString(buffdata);
        std::cout << "data_size:" << buffdata.size() << std::endl;
        std::cout << "producer_name: " << msgTele.producer_name() << std::endl;
        std::cout << "Node_Id_str: " << msgTele.node_id_str() << std::endl;
        std::cout << "ProductName: " << msgTele.product_name() << std::endl;
        std::cout << "Sub_Id_str: " << msgTele.subscription_id_str() << std::endl;
        std::cout << "Sensor_path: " << msgTele.sensor_path() << std::endl;
        std::cout << "Collection_Id: " << msgTele.collection_id() << std::endl;
        std::cout << "Collection_start_time: " << msgTele.collection_start_time() <<
std::endl;
        std::cout << "msg_timestamp: " << msgTele.msg_timestamp() << std::endl;
        std::cout << "Collection_end_time: " << msgTele.collection_end_time() << std::endl;
        std::cout << "Current_period: " << msgTele.current_period() << std::endl;
        std::cout << "except_desc: " << msgTele.except_desc() << std::endl;
```

```
        std::cout << "Encoding: " << msgTele.Encoding_Name(msgTele.encoding()) <<
std::endl;
        if(msgTele.encoding() == 1)
        {
      std::cout << "Start---------------------------------" << std::endl;
            string path = msgTele.sensor_path();
            ConverXpath2MessageName(path);//将 XPATH 转换成 messagename
            Message *pGpbMsg = createMessage(path);//创建 message 解析三层数据
        if (pGpbMsg == NULL)
      {
                      std::cout << "break------------------------ " << std::endl;
        return Status::OK;
      }
            msgTable = msgTele.data_gpb();
            row_size = msgTable.row_size();
      std::cout << msgTable.DebugString() << std::endl;

      std::cout << "row_size: " << row_size << std::endl;
            for (int i = 0; i < row_size; i++)
               {
               msgRow = msgTable.row(i);
               content = msgRow.content();
               pGpbMsg->ParseFromString(content);
               std::cout << pGpbMsg->DebugString() << std::endl;
               }
        }
        if(msgTele.encoding() == 0)
        {
            std::cout << "JSON-Data: " << msgTele.data_str() << std::endl;
        }
        std::cout << "----------------------------------------" << std::endl;
   buffdata = "";
         buffsize = 0;
        }
    }
    return Status::OK;
}
Message *createMessage(const std::string &type_name)
{
    Message* pMessage = NULL;
    const google::protobuf::Descriptor* descriptor =

google::protobuf::DescriptorPool::generated_pool()->FindMessageTypeByName(type_name);
    if (descriptor)
    {
        const Message* prototype =

google::protobuf::MessageFactory::generated_factory()->GetPrototype(descriptor);
        if (prototype)
```

```
        {
            pMessage = prototype->New();
        }
    }
    return pMessage;
}
void ConverXpath2MessageName(string &path)
{
        int pos = path.find("/", 0);
        if (0 < pos)
        {
          string module = path.substr(0, pos);
          string msgname = path.substr(0, pos);
         for (int i = 0; i < pos; i++)
         {
           if ('-' == msgname[i])
           {
             msgname[i] = '_';
           }
           msgname[i] = tolower(msgname[i]);
         }
         for (int i = 0; i < pos; i++)
         {
           if ('-' == module[i])
           {
             module[i] = '_';
           }
         }
          msgname.append("_v3.");
          msgname.append(module);
          path.replace(0, path.length(), msgname);
        }
}
```

(4) 通过 Read 方法获取到 Proto 文件生成的 DialoutV3Args 对象后，可以调用对应的方法获取相应的字段值。

3. 二层 gNMI Dial-out 模式

编码步骤如下：

(1) 继承并重载 RPC 服务 gNMIDialOut。

新建一个类 gNMI_Dialout_Sever 并继承 gNMIDialOut::Service，代码示例如下：

```
class gNMI_Dialout_Sever final : public gNMIDialOut::Service{
     Status Publish(ServerContext* context, ServerReaderWriter< PublishResponse,
SubscribeResponse>* stream) override;
};
```

(2) 将 gNMI_Dialout_Sever 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
using grpc::Server;
```

```
using grpc::ServerBuilder;
std::string server_address("0.0.0.0:50051");//指定要监听的地址和端口
        gNMI_Dialout_Sever dialout_test; //定义（1）中声明的对象
       ServerBuilder builder;
        builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());//添
加监听
        builder.RegisterService(&dialout_test); //注册服务
       std::unique_ptr<Server> server(builder.BuildAndStart());//启动服务
       std::cout << "Server listening on " << server_address << std::endl;
       server->Wait();
```

(3) 实现 gNMI Dialout 方法，实现数据解析。

代码示例如下：

```
Status gNMI_Dialout_Sever::Publish(ServerContext* context, ServerReaderWriter<
PublishResponse, SubscribeResponse>* stream)
{
    long long timestamp;
    SubscribeResponse response;
    std::vector<PublishResponse> publishes;
    while(stream->Read(&response)){
        std::cout << std::endl << "SubscribeResponse:" << response.DebugString() <<
std::endl;
        if(response.response_case() == ::gnmi::SubscribeResponse::kUpdate)
        {
            auto& notification = response.update();
            timestamp = notification.timestamp();
            for(int i = 0; i < notification.update_size(); ++i)
            {
                auto& update = notification.update(i);
                if(!update.has_val())
                    continue;
                auto& val = update.val();
                try
                {
                    if(val.value_case() == ::gnmi::TypedValue::kJsonVal)
                    {
                        std::cout << json::JSONPrettifyWithRapid(val.json_val()) <<
std::endl;
                    }
                }
                catch(std::exception& e)
                {
                    std::cout << "Json Format Error!" << std::endl;
                }
            }
            PublishResponse publish;
            publish.set_timestamp(timestamp);
            auto prefix = publish.mutable_prefix();
            prefix->add_elem()->set_name("test_prefix");
```

```
            auto path_add = publish.add_path();
            path_add->add_elem()->set_name("test_path");
            publishes.push_back(publish);
        }
        return Status::OK;
}
```

(4) 通过 Read 方法获取到 Proto 文件生成的 DialoutMsg 对象后，可以调用对应的方法获取相应的字段值。

6.5 gRPC Dial-out模式二次开发举例（GO）

6.5.1 生成代码

开发代码之前，需要使用 protoc 工具将收集到的 Proto 文件转换成 C++代码，并将生成的代码加入到开发的工程中。

本例中，开发二层普通 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout.proto

本例中，开发三层 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout_v3.proto
- telemetry.proto
- 业务 Proto 文件，本例中为 Ifmgr_v3.proto

本例中，开发二层 gNMI Dial-out 模式的代码使用以下 Proto 文件：

- dialout.proto
- gnmi.proto 和 gnmi_ext.proto

使用 protoc 工具生成 GO 代码的示例如下：

```
[root@ grpc]# cd protobuf
[root@ protobuf]# protoc --go_out=plugins=grpc:. grpc_dialout.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. telemetry.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. dialout.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. gnmi.proto
[root@ protobuf]# protoc --go_out=plugins=grpc:. gnmi_ext.proto
```

6.5.2 开发代码

1. 二层普通 Dial-out 模式

编码步骤如下：

(1) 编写业务实现方法的容器。

代码示例如下：

```
       type Server struct {
          grpc_dialout.GRPCDialoutServer1.
}
        //采集数据结构体
type TelemetryData struct {
  DeviceMsg  *grpc_dialout.DeviceInfo
```

```
  SensorPath string
  JsonData   string
}
```

(2) 将 DialoutTest 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
lis, err := net.Listen("tcp", 50051)
  if err != nil {
      fmt.Printf("failed to listen: %v\n", err)
  }
  s := grpc.NewServer([]grpc.ServerOption{grpc.NumStreamWorkers(10)}...)
  grpc_dialout.RegisterGRPCDialoutServer(s, &Server{})
  if err := s.Serve(lis); err != nil {
      fmt.Printf("failed to serve: %v\n", err)
  }
```

(3) 实现 Dialout 方法，实现数据解析。

代码示例如下：

```
func (s *Server) Dialout(gs grpc_dialout.GRPCDialout_DialoutServer) error {
  for{
           recvData, err := gs.Recv()
           if err == io.EOF {
               return gs.SendAndClose(&grpc_dialout.DialoutResponse{
                   Response: &grpc_dialout.DialoutMsg{
                        DeviceMsg: &grpc_dialout.DeviceInfo{
                            DeviceName: "H3C",
                        },
                   },
               })
           }
           if err != nil {
               return err
           }
           var data TelemetryData
           data.DeviceMsg = recvData.GetDeviceMsg()
           data.SensorPath = recvData.GetSensorPath()
           data.JsonData = recvData.GetJsonData()
  }
  return nil
}
```

(4) 通过获取到 Proto 文件生成的 DialoutMsg 对象后，可以调用对应的方法获取相应的字段值。

2. 三层 Dial-out 模式

编码步骤如下：

(1) 编写业务实现方法的容器。

代码示例如下：

```
// 业务实现方法的容器
type Server struct {
  grpc_dialout_v3.GRPCDialoutV3Server
```

```
}

type TelemetryV3Data struct {
  ReqId  int64
  data   []byte
  errors   string
  totalSize int32
}

type telemetryGpb struct {
    Row              []*telemetry.TelemetryRowGPB
}

type devicegbp struct {
    Base                      *device_v3.Device_MsgBase
}
```

(2) 将 GRPCDialoutV3 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
func RunS() {
  lis, err := net.Listen("tcp", port)
  if err != nil {
      fmt.Printf("failed to listen: %v\n", err)
  }
  s := grpc.NewServer([]grpc.ServerOption{grpc.NumStreamWorkers(10)}...)
  grpc_dialout_v3.RegisterGRPCDialoutV3Server(s, &Server{})
  if err := s.Serve(lis); err != nil {
      fmt.Printf("failed to serve: %v\n", err)
  }

}

func main() {
  fmt.Println("start grpc server for h3c roce telemetry")
  fmt.Printf("binding port:%v\n", port)
  RunS()

}
```

(3) 实现 GRPCDialoutV3 方法，解析数据。

代码示例如下：

```
func (s *Server) DialoutV3(stream grpc_dialout_v3.GRPCDialoutV3_DialoutV3Server) error {
  for{
           recvData, err := stream.Recv()
           if err == io.EOF {
               return grpc.Errorf(codes.Aborted, "stream EOF received")
           }
           if err != nil {
               return grpc.Errorf(grpc.Code(err), "received error from client")
```

```
            }
            var dataRecv TelemetryV3Data
            dataRecv.ReqId = recvData.GetReqId()
            dataRecv.data = recvData.GetData()
            dataRecv.totalSize = recvData.GetTotalSize()
            dataRecv.errors = recvData.GetErrors()
            //解析 Telemetry 层数据
             printGpb(dataRecv.data)
  }
  return nil
}
```

(4) 解析 Telmetry 层数据。

以 Ifmgr/devicecapabilities 为例的代码示例如下：

```
func typeForName(name string) (reflect.Type, error) {
    pt := proto.MessageType(name)
    if pt == nil {
        return nil, fmt.Errorf("unknown type: %q", name)
    }
    st := pt.Elem()
    return st, nil
}
//根据不同业务，获取业务模块 proto 文件，进行解码
func getProto(messageType string, messageBytes []byte) proto.Message {
    pt, _:= typeForName(messageType)
    msg := reflect.New(pt).Interface().(proto.Message)
    proto.Unmarshal(messageBytes, msg)
    return msg
}

func printGpb(bytes []byte) {
  msg := new(telemetry.Telemetry)
//进行 Telemetry 层数据解码
  proto.Unmarshal(bytes, msg)
  if msg.Encoding == telemetry.Telemetry_Encoding_JSON {
     fmt.Println("Json")
     fmt.Println(msg.DataStr)

    }else {
     fmt.Println("GPB")
     var gpbRow telemetryGpb
     gpbRow.Row = msg.GetDataGpb().GetRow()
     rowcontet := gpbRow.Row[0].GetContent()
     str := strings.Split(msg.SensorPath,"/")
     messageName := strings.ToLower(str[0]) + "_v3."+str[0]
       fmt.Println(messageName)
       //业务层数据解码
       context := getProto(messageName, rowcontet)
        fmt.Println(context)
```

```
    }
}
```

3. 二层 gNMI Dial-out 模式

编码步骤如下：

(1) 编写业务实现方法的容器。

代码示例如下：

```
// server is used to implement
type Server struct {
  dataStore interface{}  //For storing the data received
}
```

(2) 将 GNMIDialout 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
func main() {
  lis, err := net.Listen("tcp", 50051)
  if err != nil {
       log.Fatalf("failed to listen: %v", err)
  }
  s := grpc.NewServer()
  gs.RegisterGNMIDialOutServer(s, &Server{})
  if err := s.Serve(lis); err != nil {
       log.Fatalf("failed to serve: %v", err)
  }
}
```

(3) 实现 GNMIDialout 方法，并进行数据解析。

代码示例如下：

```
// Publish implements
func (srv *Server) Publish(stream gs.GNMIDialOut_PublishServer) error {
  for {
      subscribeResponse, err := stream.Recv()
      if err != nil {
           if err == io.EOF {
               return grpc.Errorf(codes.Aborted, "stream EOF received")
           }
           return grpc.Errorf(grpc.Code(err), "received error from client")
      }else {
           fmt.Println("success:", subscribeResponse)
      }
  }
}
```

(4) 根据 Ifmgr_v3.proto 文件，可以调用对应的方法获取相应的字段值。

6.6 gRPC Dial-out模式二次开发举例（Python）

6.6.1 生成代码

开发代码之前，需要使用 protoc 工具将收集到的 Proto 文件转换成 C++代码，并将生成的代码加入到开发的工程中。

本例中，开发二层普通 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout.proto

本例中，开发三层 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout_v3.proto
- telemetry.proto
- 业务 Proto 文件，本例中为 Ifmgr_v3.proto

本例中，开发二层 gNMI Dial-out 模式的代码使用以下 Proto 文件：

- dialout.proto
- gnmi.proto 和 gnmi_ext.proto

使用 protoc 工具生成 Python 代码的示例如下：

```
[root@ grpc]# cd protobuf
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
grpc_dialout.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
grpc_dialout_v3.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
telemetry.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
Ifmgr_v3.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
dialout.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
gnmi.proto.proto
[root@ protobuf]# python -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=.
gnmi_ext.proto.proto
```

6.6.2 开发代码

1. 二层普通 Dail-out 模式

编写步骤如下：

(1) 新建 DialoutServicer 类，实现数据的接收和响应。

代码示例如下：

```
class DialoutServicer(grpc_dialout_pb2_grpc.GRPCDialoutServicer):
    def Dialout(self, request_iterator, context):
        for i, req in enumerate(request_iterator):
            print("thread: %d, message index: %d" % (threading.get_ident(), i))
            print(req)
            print()
        print("a client is disconnected")
```

```
        return grpc_dialout_pb2.DialoutResponse(response="anything")
```

(2) 将 DialoutServicer 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
def serve():
    addr = "[::]:50051"
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=4))
    grpc_dialout_pb2_grpc.add_GRPCDialoutServicer_to_server(DialoutServicer(), server)
    server.add_insecure_port(addr)
    server.start()
    print(f"grpc dialout server is running on {addr}")
    try:
        server.wait_for_termination()
    except KeyboardInterrupt:
        print("Interrupted, now quiting")
    return
```

(3) 通过获取到的 Proto 文件解析接收到的采样数据，获取对应的字段值。

2. 三层 Dial-out 模式

编码步骤如下：

(1) 新建 DialoutV3Servicer 类，实现数据的接收和响应。

代码示例如下：

```
class DialoutV3Servicer(grpc_dialout_v3_pb2_grpc.gRPCDialoutV3Servicer):
    def DialoutV3(self, request_iterator, context):
        for i, req in enumerate(request_iterator):
            print("thread: %d, message index: %d" % (threading.get_ident(), i))
            self.print_gpb(req)//接收的数据需要根据 proto 文件解析
            print()
        print("a client is disconnected\n")
        return
```

(2) 将 DialoutServicer 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
def serve():
    addr = "[::]:50051"
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=4))
    grpc_dialout_v3_pb2_grpc.add_gRPCDialoutV3Servicer_to_server(DialoutV3Servicer(),
server)
    server.add_insecure_port(addr)
    server.start()
    print(f"grpc dialout server is running on {addr}")
    try:
        server.wait_for_termination()
    except KeyboardInterrupt:
        print("Interrupted, now quiting")
    return
```

(3) 在 DialoutV3Servicer 类中新增方法解析接收到数据。

代码示例如下：

```
def find_msg_cls(self, sensor_path:str):
        module_name = sensor_path.split("/")[0]
        proto_name = f"{module_name}_v3"
        pool = descriptor_pool.Default()
        file_desc = pool.FindFileByName(f"{proto_name}.proto")//业务部 proto 文件，解析不同 GPB
编码
        module_desc = file_desc.message_types_by_name[module_name]
        msg_fact = message_factory.MessageFactory(pool=pool)
        return msg_fact.GetPrototype(module_desc)


    def print_gpb(self, msg:grpc_dialout_v3_pb2.DialoutV3Args):
        telemetry = telemetry_pb2.Telemetry.FromString(msg.data)
        sensor_path = telemetry.sensor_path
        msg.ClearField("data")
        s = str(msg)
        if telemetry.encoding == telemetry_pb2.Telemetry.Encoding.Encoding_JSON:
            s += "\n" + str(telemetry)
            print(s)
            return

        tmp = ""
        try:
            msg_cls = self.find_msg_cls(sensor_path)
            for row in telemetry.data_gpb.row:
                gpb = row.content
                msec = "%03d" % (row.timestamp % 1000)
                timestr =
"({})".format(datetime.fromtimestamp(row.timestamp/1000).strftime(f"%Y-%m-%d %H:%M:%S.{m
sec}"))
                tmp += "\ntimestamp: " + str(row.timestamp) + timestr
                tmp += "\nkeys: " + str(row.keys)
                tmp += f"\ncontent({sensor_path}):\n" + str(msg_cls.FromString(gpb))
            telemetry.ClearField("data_gpb")
            s += "\n" + str(telemetry) + tmp
            print(s)
        except Exception as e:
            print(e)
        return
```

3. 二层 gNMI Dial-out 模式

编码步骤如下：

(1) 新建 GnmiDialoutServicer 类，实现数据的接收和响应。

代码示例如下：

```
class GnmiDialoutServicer(dial_out_pb2_grpc.gNMIDialOutServicer):
    def Publish(self, request_iterator, context):
        for i, req in enumerate(request_iterator):
            print("thread: %d, message index: %d" % (threading.get_ident(), i))
```

```
            if req.HasField("update"):
                tm = req.update.timestamp
                msec = "%03d" % ((tm % 1000000000) // 1000000)

print(datetime.fromtimestamp(tm/1000000000).strftime(f"%Y-%m-%d %H:%M:%S.{msec}"))
            print(req)
            print()
        print("a client is disconnected")
        return
```

(2) 将 GnmiDialoutServicer 服务注册为 gRPC 服务，并指定监听端口。

代码示例如下：

```
def serve():
    addr = "[::]:50051"
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=4))
    dial_out_pb2_grpc.add_gNMIDialOutServicer_to_server(GnmiDialoutServicer(), server)
    server.add_insecure_port(addr)
    server.start()
    print(f"grpc dialout server is running on {addr}")
    try:
        server.wait_for_termination()
    except KeyboardInterrupt:
        print("Interrupted, now quiting")
    return
```

6.7 gRPC Dial-out模式二次开发举例（JAVA）

6.7.1 生成代码

开发代码之前，需要开发人员先安装好 maven 环境，并在创建 maven 工程后修改 pom.xml，实现下载 protoc 工具的步骤如下：

```
<plugins>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId>
                <version>0.6.1</version>
                <configuration>

<protocArtifact>com.google.protobuf:protoc:3.6.1:exe:${os.detected.classifier}</protocAr
tifact>
                    <pluginId>grpc-java</pluginId>

<pluginArtifact>io.grpc:protoc-gen-grpc-java:1.14.0:exe:${os.detected.classifier}</plugi
nArtifact>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
```

```
                        <goal>compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
```

然后，开发人员在maven环境中运行mvn install即可将收集到的Proto文件生成对应的JAVA代码。

本例中，开发二层普通 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout.proto

本例中，开发三层 Dial-out 模式的代码使用以下 Proto 文件：

- grpc_dialout_v3.proto
- telemetry.proto
- 业务 Proto 文件，本例中为 Ifmgr_v3.proto

本例中，开发二层 gNMI Dial-out 模式的代码使用以下 Proto 文件：

- dialout.proto
- gnmi.proto 和 gnmi_ext.proto

6.7.2 开发代码

1. 二层普通 Dail-out 模式

编码步骤如下：

(1) 编写 DialoutServer 类，并将服务注册到指定监听端口上。

代码示例如下：

```
public DialoutServer(int port) {
        this.port = port;
        ServerBuilder<?> serverBuilder = ServerBuilder.forPort(port);
        server = serverBuilder.addService(new DialoutService()).build();
    }
```

(2) 在 DialoutServer 类中新建 DialoutService 类，并继承 PCDialoutGrpc.GRPCDialoutImplBase 完成数据的接受和解析。

代码示例如下：

```
private static class DialoutService extends GRPCDialoutGrpc.GRPCDialoutImplBase {
        DialoutService() {}

        /**
         * Gets a stream of DialoutMsg
         *
         * @param responseObserver an observer to receive the dialout response.
         * @return an observer to receive the requested dialout messages.
         */
        @Override
        public StreamObserver<GrpcDialout.DialoutMsg> dialout(final
StreamObserver<GrpcDialout.DialoutResponse> responseObserver) {
            return new StreamObserver<GrpcDialout.DialoutMsg>() {
```

```
                @Override
                public void onNext(GrpcDialout.DialoutMsg msg) {
                   //数据解析
                    System.err.println("--- received a msg ---");

                    GrpcDialout.DeviceInfo deviceInfo = msg.getDeviceMsg();
                    System.out.printf("producerName: %s\n",
deviceInfo.getProducerName());
                    System.out.printf("deviceName: %s\n", deviceInfo.getDeviceName());
                    System.out.printf("deviceModel: %s\n", deviceInfo.getDeviceModel());
                    if (deviceInfo.hasDeviceModel()) {
                        System.out.printf("deviceIpAddr: %s\n",
deviceInfo.getDeviceIpAddr());
                    }
                    if (deviceInfo.hasEventType()) {
                        System.out.printf("eventType: %s\n", deviceInfo.getEventType());
                    }

                    System.out.printf("sensorPath: %s\n", msg.getSensorPath());
                    System.out.printf("jsonData len: %s\n", msg.getJsonData().length());
                    System.out.printf("jsonData: %s\n", msg.getJsonData());

                    if (msg.hasChunkMsg()) {
                        GrpcDialout.ChunkInfo chunk = msg.getChunkMsg();
                        System.out.printf("totalSize: %d\n", chunk.getTotalSize());
                        System.out.printf("totalFragments: %d\n",
chunk.getTotalFragments());
                        System.out.printf("nodeId: %d\n", chunk.getNodeId());
                    }
                    System.out.println();
                }

                @Override
                public void onError(Throwable throwable) {
                    logger.log(Level.WARNING, "Dialout cancelled");
                }

                @Override
                public void onCompleted() {
                    System.out.println("a client disconnectd.");
                    responseObserver.onCompleted();
                }
            };
        }
    }
```

(3)　在 DialoutServer 类中实现 main 函数，服务端启动监听。

代码示例如下：

```
        public void start() throws IOException {
```

```
        server.start();
        logger.info("Server started, listening on  " + port);
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                System.err.println("*** shuting down gRPC server since JVM is shutting
down");
                try {
                    DialoutServer.this.stop();
                }catch (InterruptedException e) {
                    e.printStackTrace(System.err);
                }
                System.err.println("*** server shut down");
            }
        });
    }

    public void stop() throws InterruptedException {
        if (server != null) {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        }
    }

    public void blockUnitilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }
    //main 函数
    public static void main (String[] args) throws Exception {
        //此函数是参数解析，可以自行实现
ArgParser argParser = new ArgParser();
        argParser.Parse(args);
        DialoutServer server = new DialoutServer(argParser.port);
        server.start();
        server.blockUnitilShutdown();
    }
```

2. 三层 Dial-out 模式

编码步骤如下：

(1)　编写 gRPCgpb 类，并将服务注册到指定监听端口上。

代码示例如下：

```
public gRPCgpb(int port) {
        this.port = port;
        ServerBuilder<?> serverBuilder = ServerBuilder.forPort(port);
        server = serverBuilder.addService(new Dialoutv3Service()).build();
    }
```

(2) 在 gRPCgpb 类中新建 Diaoutv3Service 类，并继承 gRPCDialoutV3Grpc.gRPCDialoutV3ImplBase。

代码示例如下：

```
private static class Dialoutv3Service extends gRPCDialoutV3Grpc.gRPCDialoutV3ImplBase {

        Dialoutv3Service() {}

        @Override
        public StreamObserver<GrpcDialoutV3.DialoutV3Args> dialoutV3(final
StreamObserver<GrpcDialoutV3.DialoutV3Args>  responseObserver) {
            return new StreamObserver<GrpcDialoutV3.DialoutV3Args>() {
                @Override
                public void onNext(GrpcDialoutV3.DialoutV3Args dialoutV3Args) {
                    System.err.println("--- received a msg ---");
                    System.out.printf("reqId: %d\n", dialoutV3Args.getReqId());
                    System.out.printf("errors: %s\n", dialoutV3Args.getErrors());
                    System.out.printf("reqId: %d\n", dialoutV3Args.getTotalSize());
                    paraseData(dialoutV3Args.getData());
                }

                @Override
                public void onError(Throwable throwable) {
                    logger.log(Level.WARNING, "Dialout cancelled");
                }

                @Override
                public void onCompleted() {
                    System.out.println("a client disconnectd.");
                    responseObserver.onCompleted();
                }
            };
        }

    }
}
```

(3) 在新建的 Dialoutv3Service 类中实现三层数据的解码。

以 Ifmgr/devicecapabilities 为例，代码如下：

```
//根据业务 Proto 文件解析业务数据
private void  printObject(String sensorPath, TelemetryOuterClass.TelemetryRowGPB rowGPB){
            try {
                if (true ==sensorPath.equals("Ifmgr/Devicecapabilities")) {
                    Object object =IfmgrV3.Ifmgr.parseFrom(rowGPB.getContent());
                    System.out.println(object.toString());
                }

            }catch (InvalidProtocolBufferException e)
            {
```

```
                logger.log(Level.WARNING, "sensorPath:" + sensorPath + ",TelemetryRowGPB:"
+ rowGPB);
            }

        }

        private void copyTelemetryRowGPB(TelemetryOuterClass.TelemetryRowGPB
rowGPB,String sensorPath){
            if (rowGPB != null) {
                System.out.printf("timestamp: %d\n", rowGPB.getTimestamp());
                System.out.printf("key: %s\n", rowGPB.getKeys().toString());
                printObject(sensorPath, rowGPB);
            }
        }
        //GPB 数据解析
        public void paraseData (ByteString byteString) {
            try {
                TelemetryOuterClass.Telemetry telemetry
TelemetryOuterClass.Telemetry.parseFrom(byteString);
                if (telemetry == null) {
                    return;
                }
                System.out.printf("producer_name: %s\n", telemetry.getProducerName());
                System.out.printf("node_id_str: %s\n", telemetry.getNodeIdStr());
                System.out.printf("product_name: %s\n", telemetry.getProductName());
                System.out.printf("sensorPath: %s\n", telemetry.getSensorPath());
                System.out.printf("collection_id: %d\n", telemetry.getCollectionId());
                System.out.printf("collection_start_time: %d\n",
telemetry.getCollectionStartTime());
                System.out.printf("msg_timestamp: %d\n", telemetry.getMsgTimestamp());
                System.out.printf("collection_end_time: %d\n",
telemetry.getCollectionEndTime());
                System.out.printf("current_period: %d\n", telemetry.getCurrentPeriod());
                System.out.printf("execpt_desc: %s\n", telemetry.getExceptDesc());
                System.out.printf("encoding: %s\n", telemetry.getEncoding().toString());
                if (telemetry.getEncodingValue() == 0) {
                    System.out.println(telemetry.getDataStr());
                }else {
                    TelemetryOuterClass.TelemetryGPBTable telemetryGPBTable =
telemetry.getDataGpb();

                    if (telemetryGPBTable != null) {
                        int rowCount = telemetryGPBTable.getRowCount();
                        for (int i = 0; i < rowCount; i++) {
//对 Telemetry 的 row 内容进行解码

copyTelemetryRowGPB(telemetryGPBTable.getRow(i),telemetry.getSensorPath());
                        }
                    }
```

```
                }

            }catch (Exception e) {
                System.err.printf("TelemetryRowGPB parased false");
            }


        }
```

(4) 在 gRPCgpb 类中实现 main 函数，服务端启动监听。

代码示例如下：

```
public void start() throws  IOException {
        server.start();
        logger.info("Server started, listening on  " + port);
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                System.err.println("*** shuting down gRPC server since JVM is shutting
down");
                try {
                    gRPCgpb.this.stop();
                }catch (InterruptedException e) {
                    e.printStackTrace(System.err);
                }
                System.err.println("*** server shut down");
            }
        });
    }

    public void stop() throws InterruptedException {
        if (server != null) {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        }
    }

    public void blockUnitilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }
    //main 函数
public static void main (String[] args) throws Exception {
    //此函数是参数解析，可以自行实现
        ArgParser argParser = new ArgParser();
        argParser.Parse(args);
        gRPCgpb server = new gRPCgpb(argParser.port);
        server.start();
        server.blockUnitilShutdown();
    }
```

3. 二层 gNMI Dial-out 模式

编码步骤如下：

(1) 编写 gnmiDialout 类，并将服务注册到指定监听端口上。

代码示例如下：

```
    public gnmiDailout(int port) {
        this.port = port;
        ServerBuilder<?> serverBuilder = ServerBuilder.forPort(port);
        server = serverBuilder.addService(new gnmiDailoutService()).build();
    }
```

(2) 在 gnmiDialout 中新建 gnmiDialoutService 类，并继承 gNMIDialOutGrpc.gNMIDialOutImplBase，实现数据的解析。

代码示例如下：

```
private static class gnmiDailoutService extends gNMIDialOutGrpc.gNMIDialOutImplBase {

        gnmiDailoutService() {}

        @Override
        public StreamObserver<Gnmi.SubscribeResponse> publish(final
StreamObserver<DialOut.PublishResponse> responseObserver) {
            return new StreamObserver<Gnmi.SubscribeResponse>() {
                @Override
                public void onNext(Gnmi.SubscribeResponse subscribeResponse) {
                     System.err.println("--- received a msg ---");
                     //解析数据
                     Object object = subscribeResponse.getUpdate();
                     System.out.println(object.toString());
                }

                @Override
                public void onError(Throwable throwable) {
                    logger.log(Level.WARNING, "Dialout cancelled");
                }

                @Override
                public void onCompleted() {
                    System.out.println("a client disconnectd.");
                    responseObserver.onCompleted();
                }
            };
        }

    }
```

(3) 在 gnmiDialout 类中编写 main 函数，启动服务端监听。

代码示例如下：

```
    public void start() throws IOException {
        server.start();
```

```
        logger.info("Server started, listening on  " + port);
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                System.err.println("*** shuting down gRPC server since JVM is shutting
down");
                try {
                    gnmiDailout.this.stop();
                }catch (InterruptedException e) {
                    e.printStackTrace(System.err);
                }
                System.err.println("*** server shut down");
            }
        });
    }

    public void stop() throws InterruptedException {
        if (server != null) {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        }
    }

    public void blockUnitilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }

    public static void main (String[] args) throws Exception {
        ArgParser argParser = new ArgParser();
        argParser.Parse(args);
        gnmiDailout server = new gnmiDailout(argParser.port);
        server.start();
        server.blockUnitilShutdown();
}
```

7 常见问题

Q：采样周期的准确性受那些因素影响？

A：正常情况下，设备会以用户配置的 Telemetry 采样周期对订阅的数据进行采样，但是由于现网环境复杂、业务较多，采样周期的准确性受以下因素影响：

- 受采样实例数目影响：部分节点的采样路径下包含的采样实例数目庞大，例如 route/ipv4routes，当路由表项达到 100k 时，采样数据量较大，设备无法在一个较小的采样周期完成采集工作。
- 受采样数据源的采样周期影响：某些数据源有自己的最小采样周期，当它与用户配置的 Telemetry 采样周期不一致时，设备可能不会按照配置的采样周期上报数据。例如，采样路径

device/base 上的最小采样周期为 5 秒，若设备上配置的 Telemetry 采样周期为 100 毫秒，由于数据源无法达到配置的采样周期精度，gRPC 模块会以数据源自身的最小采样周期上报数据。

- 受 CPU 影响：当设备上的 CPU 繁忙时，Telemetry 的采样工作可能无法正常进行，gRPC 模块将不能在当前采样周期内完成采集工作。