← 概述

OmniNRF 配置参考

OmniNRF 从 `config/runtime.exs` 读取其操作员配置，该配置将环境变量映射到 `:omninrf` 应用程序环境。每个面向操作员的键在下面列出，包括其环境变量和默认值。

运行时配置

```
import Config

config :omninrf,
  sbi_addr: System.get_env("SBI_ADDR", "127.0.0.1"),
  sbi_port: String.to_integer(System.get_env("SBI_PORT", "7777")),
  mcc: System.get_env("MCC", "999"),
  mnc: System.get_env("MNC", "70"),
  heartbeat_timeout:
String.to_integer(System.get_env("HEARTBEAT_TIMEOUT", "30000")),
  prometheus_metrics_port:
String.to_integer(System.get_env("PROMETHEUS_PORT", "9568"))

config :logger, :default_formatter,
  format: {OmniLogger.JsonFormatter, :format},
  metadata: :all
```

参数表

参数	类型	必需	默认
`sbi_scheme`	string	否	`http`
`sbi_addr`	string	否	`127.0.0.1`
`sbi_port`	integer	否	`7777`
`mcc`	string	否	`999`
`mnc`	string	否	`70`
`heartbeat_timeout`	integer (ms)	否	`30000`

□□	□□	□□	□□
`cleanup_interval`	integer (ms)	□	`10000`
`notification_max_attempts`	integer	□	`4`
`notification_backoff_ms`	integer (ms)	□	`500`
`discovery_validity_period`	integer (s)	□	`3600`
`plmn_list`	list	□	`[%{mcc: mcc, mnc: mnc}]`
`prometheus_metrics_port`	integer	□	`9568`

参数	类型	必需	默认
`logger format`	tuple	否	`{OmniLogger.JsonFormatter, :format}`
`logger metadata`	atom/list	否	`:all`

> `SBI_ADDR` 默认值为 `127.0.0.1`（回环）。在任何多主机或容器化部署中，必须将其覆盖为其他 NF 可以访问的地址，否则将无法注册或发现任何 NF。

心跳老化语义

NRF 接收来自其他 NF 的心跳；它不发送心跳���每当 NF 为其实例发送 PATCH 心跳（或任何注册/更新）时，NRF 会记录该 NF 配置的最新“最后见”时间戳。

每 `cleanup_interval` 毫秒运行一次后台清理，移除任何最后见时间戳早于 `heartbeat_timeout` 毫秒的 NF。当 NF 以这种方式被老化时，它会从存储中注销，并向任何匹配的订阅者发送 `NF_DEREGISTERED` 状态通知，以便消费者停止接收死端点。

在注册时，如果 NF 不包括自己的 `heartBeatTimer`，NRF 会填充一个等于 `heartbeat_timeout / 1000` 秒的值，作为它期望心跳的频率提示。操作员应确保 NF 的心跳在 `heartbeat_timeout` 之内（一个比超时短得多的值），以便瞬态抖动不会导致活跃的 NF 被过期。

后台清理间隔（`cleanup_interval`）和陈旧阈值（`heartbeat_timeout`）是独立的。NF 仅在其最后见年龄已经超过 `heartbeat_timeout` 后的第一次清理中被移除，因此 NF 在沉默后停留的最长时间为 `heartbeat_timeout + cleanup_interval`。保持 `cleanup_interval` 远低于 `heartbeat_timeout`（默认值为 10 秒和 30 秒），以使这一超出保持较小；`cleanup_interval` 大于 `heartbeat_timeout` 将导致死 NF 被宣传的时间超过预期。

plmn_list

`plmn_list` 是此 **NRF** 服务的 PLMN 集合，表示为 `%{mcc, mnc}` 映射的列表。它与每个注册 NF 自身配置中的 `plmnList` 是不同的。每个 NF 的列表是发现过滤的依据，而 `plmn_list` 描述的是 NRF 自身服务的网络。

子字段	类型	描述
`mcc`	string	移动国家代码（3 位数字）
`mnc`	string	移动网络代码（2 到 3 位数字）

如果未设置，`plmn_list` 默认值为从顶层 `mcc`/`mnc` 键构建的单个条目。为共享/多运营商 NRF 配置多个条目，例如：

```
config :omninrf,
  plmn_list: [%{mcc: "999", mnc: "70"}, %{mcc: "001", mnc: "01"}]
```

发现匹配（`plmn-id` 查询参数）是针对每个 NF 自身的 `plmnList` 进行评估，而不是针对 `plmn_list`。根据 TS 29.510，注册的 NF 如果 没有 广告 `plmnList`，则被认为属于 NRF 的 PLMN，因此匹配任何 `plmn-id` 过滤器。

管理 / OAM API

除了 5G SBI，OmniNRF 还提供面向操作员的 REST 管理 API，用于运行时检查和控制。它通过 **HTTPS** 在 api_ex 管理监听器上提供（默认端口 `8443`）。请求和响应主体为 JSON。



方法	路径	描述
GET	`/api/oam/config`	查看当前运行时配置（`sbi_addr`、`sbi_port`、`heartbeat_timeout`、`cleanup_interval`、`plmn_list`、`log_level`）
PATCH	`/api/oam/config/:id`	更新运行时配置。仅可更改 `heartbeat_timeout`、`cleanup_interval` 和 `log_level`；所有提供的值在应用之前都会进行验证（无效值返回 400）
POST	`/api/oam/log_level`	在运行时更改日志级别。主体 `{"level": "..."}`；有效级别为 `emergency`、`alert`、`critical`、`error`、`warning`、`warn`、`notice`、`info`、`debug`（否则返回 400）
POST	`/api/oam/heartbeat`	手动刷新 NF 的心跳时间戳。主体 `{"nf_instance_id": "..."}`（如果实例未注册则返回 404）
GET	`/api/oam/nf_uptime/:id`	显示一个 NF 实例的注册/心跳年龄和状态（如果未注册则返回 404）
POST	`/api/oam/nf_purge`	触发注册表的即时过期实例清理；报告移除的 NF 实例数量
GET	`/api/oam/resources`	显示 BEAM VM 资源使用情况（内存、进程/端口计数、调度器计数、运行时间、OTP/BEAM 版本）
GET	`/api/oam/diagnostics`	显示 BEAM VM 诊断信息（节点、运行时间、连接节点、进程计数、运行队列、I/O 和 GC 统计）
PATCH	`/api/oam/nf_status/:id`	更新 NF 实例的存储��态

同一监听器还提供只读状态和库存端点：`/api/status/api`、`/api/status/license`、`/api/status/nrf`、`/api/status/nf`、`/api/api/nf_instances`、`/api/api/subscriptions` 和 `/api/api/statistics`。

检查和清理 NF 实例

`GET /api/oam/nf_uptime/:id` 是查看 NF 距离被老化的直接方式。它报告：

字段	意义
`last_heartbeat_ms_ago`	NF 的最后见时间戳的年龄，以毫秒为单位
`heartbeat_timeout`	当前陈旧阈值，以毫秒为单位
`nf_type`、`nf_status`	NF 的类型和存储状态

一旦 `last_heartbeat_ms_ago` 超过 `heartbeat_timeout`，NF 就成为移除候选；两者之间的差距大致是 NF 在被清理过期之前可以保持沉默的时间。

`POST /api/oam/nf_purge` 运行与后台计时器相同的过期实例清理，立即按需执行，并返回移除的实例数量。它仅移除 □□ 过期的 NF（最后见时间早于 `heartbeat_timeout`）；它不会强制移除健康的 NF。使用它一次性回收注册表，而不是等待下一个计划的清理。要强制移除特定的活跃 NF，无论其心跳年龄如何，请直接使用 `DELETE /api/api/nf_instances/:id` 注销。

日志记录

OmniNRF 通过 `OmniLogger.JsonFormatter` 格式化器发出结构化 **JSON** 日志，所有日志元数据均包含在内（`metadata: :all`）。每条日志行携带每个请求的元数据，例如 NF 实例 ID 和正在处理的过程，这使得将注册、心跳、发现和通知事件与给定 NF 关联变得简单明了。

活动日志级别可以在运行时更改而无需重启，既可以通过 `POST /api/oam/log_level` 也可以通过 `PATCH /api/oam/config/:id` 的 `log_level` 键进行更改。

← 概述

OmniNRF 指标

指标以 Prometheus 文本格式暴露在 `http://{host}:{prometheus_metrics_port}/metrics`（默认端口 `9568`）。

NRF 服务指标

指标	类型	描述	标签
`omni_nrf_nf_registered_count`	Gauge	当前注册的 NF 实例数量，按类型划分	`nf_type`
`omni_nrf_active_nf_instances_count`	Gauge	当前活跃的 NF 实例数量，按类型划分。每次 NF 注册、注销和心跳过期时发出，因此反映了实时活跃集	`nf_type`
`omni_nrf_registration_requests_count`	Counter	接收到的 NF 注册请求总数（所有类型）	(none)
`omni_nrf_nf_registrations_total`	Counter	NF 注册总数，按类型划分	`nf_type`
`omni_nrf_nf_deregistrations_total`	Counter	NF 注销总数，按类型划分。`nf_type` 标签从存储的 NF 配置中解析	`nf_type`
`omni_nrf_heartbeats_total`	Counter	NF 心跳请求总数，按类型划分。`nf_type` 标签从存储的	`nf_type`

指标	类型	描述	标签
		NF 配置中 ��析	
`omni_nrf_discovery_requests_count`	Counter	接收到的发现请求总数（所有目标类型）	(none)
`omni_nrf_discovery_requests_total`	Counter	发现请求总数，按目标 NF 类型划分	`target_nf_`

BEAM VM 指标

指标	类型	描述
`beam_memory_total`	Gauge	总 BEAM 内存（字节）
`beam_memory_processes`	Gauge	分配给 Erlang 进程的内存
`beam_memory_processes_used`	Gauge	进程实际使用的内存
`beam_memory_system`	Gauge	系统（非进程）内存
`beam_memory_atom`	Gauge	总原子内存
`beam_memory_atom_used`	Gauge	使用的原子内存
`beam_memory_binary`	Gauge	二进制内存
`beam_memory_code`	Gauge	代码内存
`beam_memory_ets`	Gauge	ETS 表内存（保存 NF 和订阅存储）
`beam_processes_count`	Gauge	Erlang 进程数量
`beam_ports_count`	Gauge	Erlang 端口数量
`beam_atom_count`	Gauge	原子数量
`beam_vm_uptime`	Gauge	VM 运行时间（秒）

有用的 PromQL

按类型注册的 NF 实例：

```
omni_nrf_nf_registered_count
```

5 分钟内的注册请求速率：

```
rate(omni_nrf_registration_requests_count[5m])
```

按目标 NF 类型的发现请求速率：

```
sum by (target_nf_type)
(rate(omni_nrf_discovery_requests_total[5m]))
```

按 NF 类型的心跳速率（下降到零警告停止心跳的 NF）。`nf_type` 标签从存储的配置中解析，因此按类型的细分是有意义的：

```
sum by (nf_type) (rate(omni_nrf_heartbeats_total[5m]))
```

净注册流失（注册减去注销），按 NF 类型。两个计数器都携带真实的 `nf_type`，因此可以按类型计算流失：

```
sum by (nf_type) (rate(omni_nrf_nf_registrations_total[5m]))
  - sum by (nf_type) (rate(omni_nrf_nf_deregistrations_total[5m]))
```

按 NF 类型的注册（此计数器确实携带真实的 `nf_type`）：

```
sum by (nf_type) (rate(omni_nrf_nf_registrations_total[5m]))
```

> 指标名称渲染：诸如 `omni_nrf.nf.registered.count` 的遥测名称以点替换为下划线导出。请根据实时的 `/metrics` 抓取确认确切的导出名称，因为它们可能因导出配置而异。

← 概述

OmniNRF 操作指南

OmniNRF 实现了 5G 核心的网络存储功能 (NRF)。NRF 是基于服务架构 (SBA) 的中央注册和服务目录：每个网络功能 (NF) 在启动时向 NRF 注册其配置文件，通过定期心跳保持该注册，并查询 NRF 以发现它需要与之通信的 NF 的端点。

OmniNRF 暴露了两个 SBI 服务：

- **Nnrf_NFManagement** (`nnrf-nfm/v1`)：NF 配置文件注册、更新、心跳、注销、检索和 NF 状态变化订阅/通知。
- **Nnrf_NFDiscovery** (`nnrf-disc/v1`)：按类型、服务、S-NSSAI、DNN 和 PLMN 解析目标 NF 实例。

所有 NF 配置文件和订阅都保存在内存 (ETS) 存储中。没有外部数据库；注册表在 NRF 重启后重新构建，因为 NF 重新注册。后台清理会持续清除停止发送心跳的 NF 实例，因此注册表仅会宣传那些显然存活的 NF。

3GPP 角色和规范参考

规范	相关性
TS 23.501	系统架构，NRF 在基于服务架构中的角色（第 6.2.6 节）
TS 29.510	Nnrf_NFManagement 和 Nnrf_NFDiscovery 服务，NFProfile / SearchResult / SubscriptionData 数据模型
TS 29.510 §5.2	Nnrf_NFManagement：注册、更新、心跳（`heartBeatTimer`）、注销、状态订阅和通知
TS 29.510 §5.3	Nnrf_NFDiscovery：NF 实例发现查询和 SearchResult
TS 29.510 §6.2.3.2.3.1	发现查询参数、结果排序和分页

OmniNRF 是 `Nnrf_NFManagement` 和 `Nnrf_NFDiscovery` 的 SBI 生产者。它本身不向任何其他 NRF 注册；它是所有其他指向的注册表，因此在其操作员配置中没有外部 `nrf_uri`。

SBI 端点

SBI 监听器通过 **HTTP/2 (**明文**, h2c)** 提供服务，地址为 `http://{sbi_addr}:{sbi_port}`。请求和响应主体为 JSON。PATCH 主体使用 `application/json-patch+json`；NF 实例列表以 `application/3gppHal+json` 返回。

Nnrf_NFManagement (`nnrf-nfm/v1`)

方法	路径	描述	成功
PUT	`/nnrf-nfm/v1/nf-instances/{nfInstanceId}`	注册或重新注册一个 NF (完整的 NFProfile 主体)	201 Created (新) / 200 OK (更新)
PATCH	`/nnrf-nfm/v1/nf-instances/{nfInstanceId}`	心跳 (空主体) 或 JSON-Patch 配置文件更新	204 No Content (心跳) / 200 OK (补丁)
DELETE	`/nnrf-nfm/v1/nf-instances/{nfInstanceId}`	注销一个 NF	204 No Content
GET	`/nnrf-nfm/v1/nf-instances/{nfInstanceId}`	检索单个 NFProfile	200 OK
GET	`/nnrf-nfm/v1/nf-instances`	列出 NF 实例 (UriList; 支持 `nf-type`, `limit`, `page-number`, `page-size`)	200 OK
POST	`/nnrf-nfm/v1/subscriptions`	订阅 NF 状态通知	201 Created
GET	`/nnrf-nfm/v1/subscriptions/{subscriptionId}`	读取单个订阅	200 OK

方法	路径	描述	成功
PATCH	`/nnrf-nfm/v1/subscriptions/{subscriptionId}`	更新订阅 (JSON-Patch)	200 OK
DELETE	`/nnrf-nfm/v1/subscriptions/{subscriptionId}`	取消订阅	204 No Content

成功的 PUT/PATCH/GET 响应在单个 NF 实例上携带 `ETag` 头。客户端可以在 PUT/PATCH 上提供 `If-Match` 以使更新具有条件性（在不匹配时返回 412 Precondition Failed）。

Nnrf_NFDiscovery (`nnrf-disc/v1`)

方法	路径	描述	成功
GET	`/nnrf-disc/v1/nf-instances`	按类型和可选过滤器发现 NF 实例	200 OK

当发现结果被分页时（提供了 `limit` 且还有更多结果），响应携带一个 `Link: <...>; rel="next"` 头，指向下一页。

发现查询参数

参数	必需	描述
`target-nf-type`	是	正在搜索的 NF 类型（例如 `UDM`，`SMF`，`PCF`）
`requester-nf-type`	是	发出请求的 NF 类型（例如 `AMF`）
`service-names`	否	逗号分隔的服务名称；如果 NF 提供任何列出的服务，则匹配
`plmn-id`	否	JSON 编码的 PLMN ID，例如 `{"mcc":"999","mnc":"70"}`。没有 `plmnList` 的 NF 匹配任何 PLMN
`snssais`	否	JSON 编码的 S-NSSAI（对象或数组），例如 `[{"sst":1,"sd":"000001"}]`。与 NF 的 `sNssais`、`allowedNssais` 和每个服务的 `allowedNssais` 进行匹配
`dnn`	否	数据网络名称；与每个服务的 `allowedDnns` 进行匹配
`limit`	否	此页中结果的最大数量
`offset`	否	排序结果集中的起始索引，用于分页

结果以 `SearchResult` 返回（`validityPeriod`，`nfInstances`，`numNfInstComplete`）。仅返回 `nfStatus = REGISTERED` 的 NF。

结果排序

匹配的 NF 被排序，以便最优先的实例首先出现，让消费者选择 `nfInstances[0]`，并仅在需要时回退到后续条目。排序键按顺序来自每个 NF 自己注册的配置文件：

键	方向	对操作员的意义
`priority`	升序（较低优先）	在 NF 配置文件中设置的相对优先级。较低者胜出；没有 `priority` 的 NF 排序靠后（视为 `65535`）
`capacity`	降序（较高优先）	NF 的广告容量。在优先级相同的 NF 中，容量更高的 NF 更□□□青睐
`load`	升序（较低优先）	NF 报告的当前负载。在其他条件相同的 NF 中，负载较低的 NF 更受青睐

为了引导消费者选择特定的生产者，可以在该 NF 的注册配置文件上设置较低的 `priority`（或更高的 `capacity`）；这些是目标 NF 的属性，而不是 NRF 的属性。

过滤匹□规则

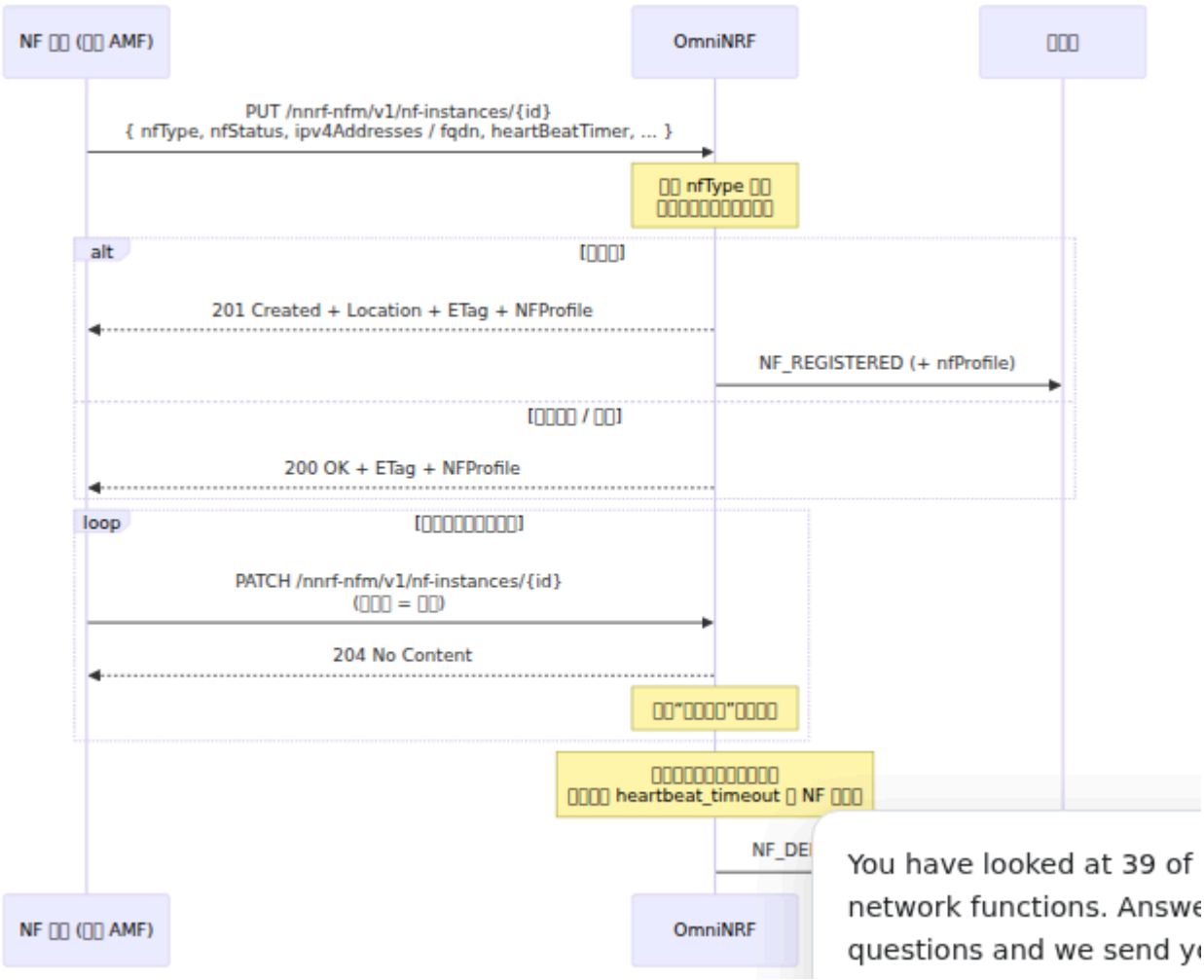
- **`service-names`**：如果 NF 提供 □□□□ 列出的服务，则匹配。
- **`snssais`**：与 NF 的 `sNssais`、其顶层 `allowedNssais` 和每个服务的 `allowedNssais` 的并集进行匹配。只有在查询提供时，`sd` 才需要匹配。
- **`plmn-id`** 和 **`dnn`**：分别与 NF 的 `plmnList` 和每个服务的 `allowedDnns` 进行匹配。

对于每个可选过滤器，未宣传 □□ 该属性的 NF 被视为匹配其任何值（例如，没有 `plmnList` 的 NF 匹配任何 `plmn-id`）。这一“无约束即普遍”规则遵循 TS 29.510，并保持最小配置的 NF 可被发现。

分页

`limit` 限制在一页中返回的实例数量；`offset` 是排序结果集中的起始索引。当页面之外还有更多结果时，响应携带一个 `Link: <...>; rel="next"` 头，其 `offset` 指向下一页。`numNfInstComplete` 始终反映分页前的完整匹配计数，因此消费者可以分页，直到 `offset + len(nfInstances)` 达到它。

> 此 SBI 发现分页（`offset`/`limit` + `Link`）与 Nnrf_NFManagement 下的 NF 实例 □□ 端点的分页是分开的，后者使用 `limit`/`page-number`/`page-size` 在 `3gppHal+json` UriList 上。

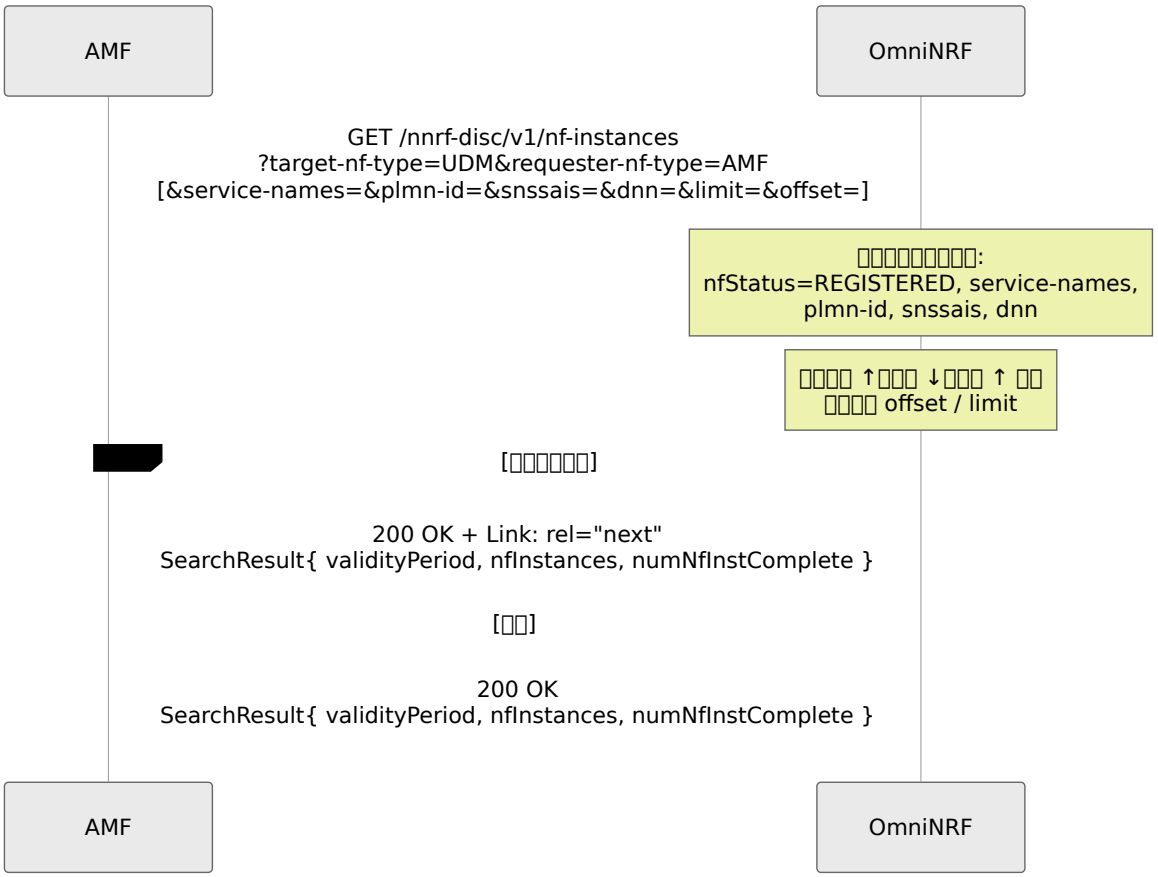


关键程序

NF 注册和心跳

注册要求主体中包含 `nfType` 和 `nfStatus`（缺少任一项的主体将被 400 `MANDATORY_IE_MISSING` 拒绝）和至少一个可达地址（`fqdn`、`ipv4Addresses` 或 `ipv6Addresses`）。没有可达地址的主体将被 400（错误请求）拒绝。

NRF 接收来自其他 NF 的心跳；它不发送心跳。每次 NF 为其实例发送 PATCH 心跳（或任何注册/更新）时，NRF 会记录该 NF 配置文件的新“最后看到”时间戳。后台清理持续运行，移除任何最后看到时间戳早于 `heartbeat_timeout` 毫秒的 NF。当 NF 以这种方式被清除时，它会从存储中注销，并向任何匹配的订阅者发送 `NF_DEREGISTERED` 状态通知，以便消费者停止接收死端点。



NF 发现

AMF
OmniNRF
GET /nnrf-disc/v1/nf-instances
?target-nf-type=UDM&requester-nf-type=AMF
[&service-names=&plmn-id=&snssais=&dnn=&limit=&offset=]
过滤目标类型候选者:
nfStatus=REGISTERED, service-names,
plmn-id, snssais, dnn
按优先级 ↑、容量 ↓、负载 ↑ 排序
然后应用 offset / limit
[还有更多结果]
200 OK + Link: rel="next"
SearchResult{ validityPeriod, nfInstances, numNfInstComplete }
[完成]
200 OK
SearchResult{ validityPeriod, nfInstances, numNfInstComplete }
AMF
OmniNRF

`target-nf-type` 和 `requester-nf-type` 都是必需的；省略任一项将返回 400。空的 `nfInstances` 列表（带有 `numNfInstComplete: 0`）是一个有效的响应，表示没有匹配。

NF 状态订阅和通知

通知通过共享的 SBI HTTP/2 客户端（池化）传递，并具有有限的重试：每个通知最多尝试 `notification_max_attempts` 次（默认 4），使用以 `notification_backoff_ms`（默认 500 毫秒，重试时翻倍）为基础的指数退避。回答非 2xx 或短暂不可达的订阅者会被重试，而不是在单次尝试后被丢弃；在最后一次失败尝试后，NRF 放弃并记录警告。不可达的订阅者会被记录，但不会使 NRF 崩溃。事件类型（TS 29.510 第 5.2.2.8 节）从 NF 状态映射：`REGISTERED` 到 `NF_REGISTERED`（包括完整的 `nfProfile`），`DEREGISTERED`/`SUSPENDED` 到 `NF_DEREGISTERED`，其他状态变化到 `NF_PROFILE_CHANGED`。具有 `subscrCond.nfType` 的订阅仅接收该 NF 类型的通知。如果提供了 `validityTime`（ISO 8601），则在过期时自动删除该订阅。

NF 状态转换及其如何到达订阅者

触发	结果通知	也从注册表中移除?
NF 注册 (PUT，新)	`NF_REGISTERED` (带有 `nfProfile`)	否，已添加
NF 配置文件更改 (PATCH / OAM 更新到非终止状态)	`NF_PROFILE_CHANGED`	否
NF 注销 (DELETE)	`NF_DEREGISTERED`	是
NF 被清除 (错过心跳，后台清理)	`NF_DEREGISTERED`	是
手动清除 (`POST /api/oam/nf_purge`) 过期的 NF	`NF_DEREGISTERED`	是

一个 被清除 的 NF 和一个显式 注销 的 NF 对订阅者来说是不可区分的：两者都显示为 `NF_DEREGISTERED`，并且都离开注册表。区别仅在于原因：注销是 NF 自身的故意 DELETE，而被清除是 NRF 移除停止心跳的 NF。一个 暂停 的 NF (`nfStatus = SUSPENDED`) 在通知时报告为 `NF_DEREGISTERED`，并且由于仅返回 `REGISTERED` 的 NF，因此在其配置文件仍在存储中时，实际上对消费者隐藏。

← 概述

OmniNRF 故障排除

一个 NF 从发现中消失，尽管它正在运行

几乎总是错过心跳。如果一个 NF 在 `heartbeat_timeout`（默认 30 秒）内没有发送 PATCH 心跳，后台清理会使其过期，并且它会从检索和发现中消失。检查 NRF 日志中的 `NRF: Expiring stale NF instance {id}`，并确认 NF 在超时内正常发送心跳。提高 `HEARTBEAT_TIMEOUT` 可以提供更多的宽松；NF 还应遵循在其注册响应中返回的 `heartBeatTimer`。

注册被拒绝，状态码 400

NFProfile 缺少一个必需元素或包含一个格式错误的元素。NRF 要求同时提供 `nfType` 和至少一个可达地址（`fqdn`、`ipv4Addresses` 或 `ipv6Addresses`）。状态码 400 和 `MANDATORY_IE_MISSING` 意味着 `nfType`/`nfStatus` 在 PUT 主体中缺失；状态码 400 的详细信息提到可达性意味着没有提供地址。

除了这些必需元素外，NRF 还对配置文件的其余部分进行深层结构验证，并在存在格式错误的元素时返回状态码 400，原因是 `MANDATORY_IE_INCORRECT`（根据 TS 29.510）。常见原因：

- `nfServices`：每个服务需要一个 `serviceInstanceId`、一个 `serviceName`、一个有效的 `scheme`、一个有效的 `nfServiceStatus` 和一个非空的 `versions` 数组，其中每个条目都携带 `apiVersionInUri`。
- `sNssais`：每个条目需要一个范围在 0 到 255 之间的整数 `sst`，以及一个（如果存在）恰好为 6 个十六进制数字的 `sd`。
- `plmnList`：条目必须是格式良好的 PLMN 对象。
- `capacity` 和 `priority` 必须是范围在 0 到 65535 之间的整数；`load` 必须是范围在 0 到 100 之间的整数。

缺失的可选元素是被接受的；只有存在但格式错误的元素会被拒绝。状态码 400 的 `detail` 指明了有问题的字段，因此可以直接指向需要修正的内容。

PATCH 被拒绝，状态码 415 不支持的媒体类型

PATCH 的 Content-Type 被设置为除 `application/json-patch+json` 以外的其他内容。要么发送该媒体类型进行 JSON-Patch 更新，要么发送一个真正空的心跳（心跳不允许有 Content-Type）。

更新/心跳/删除返回 404

`nfInstanceId` 不在注册表中，通常是因为 NF 因错过心跳而已被清除，或者它从未完成注册。在补丁或心跳之前，先使用 PUT 重新注册。

PUT/PATCH 返回 412 前置条件失败

发送了一个 `If-Match` 头，其 ETag 与当前存储的配置文件不匹配（配置文件在客户端之下发生了变化）。重新 `GET` 实例以获取当前的 `ETag`，然后重试。

发现返回空的 `nfInstances` 列表

- 确认目标 NF 已注册，并且其存储的 `nfStatus` 为 `REGISTERED`（只有注册的 NFs 会被返回）。
- `target-nf-type` 是区分大小写地与存储的 `nfType` 进行匹配。
- 如果提供了 `plmn-id`、`snssais` 或 `dnn` 过滤器，请验证它们的格式良好，并且目标 NF 的配置文件确实广告了它们（广告没有给定属性的 NF 与其任何值都不匹配）。
- 确保 `requester-nf-type` 存在。缺失会导致 400，而不是空列表。

状态通知未被接收

- 确认订阅中的 `nfStatusNotificationUri` 可以从 NRF 主机访问。
- 交付使用有界重试和指数退避（最多 `notification_max_attempts` 次尝试，默认 4 次，基础延迟为 `notification_backoff_ms`，默认 500 毫秒，每次重试延迟加倍）。重试记录为 `NRF: Retrying status notification to ...`。如果每次尝试都失败，NRF 会放弃并记录 `NRF: Giving up status notification to ... after N attempts`。在 NRF 日志中搜索这两行，���查看交付是否被重试并最终放弃。
- 如果设置了 `subscrCond.nfType`，则仅交付该 NF 类型的状态变化。

NRF 重启后注册表为空

这是预期的。NF 和订阅存储在内存中（ETS）。重启后，注册表会在 NFs 在下一个注册/心跳周期重新注册时重新填充；没有持久性可供重新加载。