第 3 章

CHAPTER 3

NS3 架构与编程模型

3.1 模块化架构与扩展设计

NS3 的架构以模块化为核心设计理念，通过将网络协议栈解耦为可独立开发、灵活组合的组件模型，实现从物理层到应用层的全栈仿真能力。这种设计不仅严格映射 OSI 分层模型，还通过标准化接口与继承机制，为开发者提供高效的扩展能力，支持快速构建自定义协议或优化算法。本节将解析 NS3 的基础模型与网络层次的关系，并阐述其扩展设计方法。

3.1.1 基础模型与网络层次映射

NS3 的基础模型覆盖网络协议栈各层功能，其与网络层次的映射关系总结于表 3-1。

表 3-1 NS3 基础模型与网络层次的映射关系

网 络 层 次	NS3 基础模型	功能描述与扩展性
物理层	信道模型(Channel)	描述信号传输介质(如无线信道衰落、光纤时延)，支持自定义传播模型(如毫米波信道建模)
数据链路层	设备模型(NetDevice)	实现 MAC 子层协议(如 CSMA/CA)，可扩展新型 MAC 接入算法或帧结构
网络层	路由模型(IPv4/IPv6)	提供 IP 地址分配、路由表管理，支持开发自定义路由协议(如 SDN 控制器驱动的动态路由)
传输层	TcpSocketBase/TcpCongestion	实现 TCP 连接管理与可靠性机制，允许替换拥塞控制算法(如实现 BBR、CUBIC 等)
应用层	应用模型(Application)	模拟上层业务(如 HTTP、FTP)，支持开发私有协议
跨层实体	节点模型(Node)	作为容器聚合设备、协议栈与应用，提供跨层参数配置接口(如节点移动性控制)
数据载体	数据包模型(Packet)	封装协议头部与载荷，支持自定义头部字段(如添加 QoS 标签)或分片策略

3.1.2 模块化架构的设计原则

NS3 的模块化设计遵循三大核心原则：分层解耦、继承与多态、聚合模型。这些原则共同支撑了其高内聚、低耦合的架构特性，使开发者能够灵活扩展或替换任意协议层组件。以下逐一解析各原则的实现逻辑与技术细节。

1. 分层解耦：标准化接口隔离依赖

NS3 通过抽象接口严格隔离各协议层的实现细节，确保模块间仅通过明确定义的接口交

互，避免直接依赖具体实现类。这一设计映射了网络协议栈的分层思想，同时为模块替换提供了技术基础。

1）接口定义与实现分离

每个模块定义其对外服务的接口类（如 Channel 定义数据传输接口），其他模块仅依赖接口而非具体实现类。

例如，Point to Point Net Device（点对点网卡）通过调用 Channel::Send()发送数据，无须关心信道是无线（Yans Wifi Channel）还是有线（Point to Point Channel）。

2）跨层交互规范化

跨层操作（如应用层读取物理层信号强度）必须通过标准接口（NetDevice::GetSignalStrength()），禁止直接访问底层私有成员。这种约束强制开发者遵循分层协议设计规范，避免架构腐化。

3）扩展性体现

开发者可通过实现相同接口的新类（如自定义 MyChannel 继承 Channel），无缝替换默认模块。

例如，实现自定义信道模型时，只需继承 Channel 并重写 Send()方法，无须修改上层设备代码。

2. 继承与多态：动态行为注入

NS3 所有核心模型均继承自 Object 基类，利用 C++多态特性实现运行时动态行为绑定。这一机制是算法级扩展的核心支撑。

1）TypeId 系统与运行时类型识别（Run-Time Type Identification，RTTI）

每个 Object 派生类必须实现 GetTypeId()方法，声明其属性、父类及构造函数。例如，TcpNewReno 的 TypeId 声明其继承自 TcpCongestion，并注册拥塞窗口等可配置属性：

```
TypeId TcpNewReno::GetTypeId() {
            static TypeId tid = TypeId("ns3::TcpNewReno")
                .SetParent<TcpCongestion>()
          .AddAttribute("InitialCwnd", "初始拥塞窗口", UintegerValue(10),
                        MakeUintegerAccessor(&TcpNewReno::m_cWnd),
                        MakeUintegerChecker<uint32_t>());
            return tid;
}
```

2）虚函数与算法重写

关键算法逻辑通过虚函数暴露（如 TcpCongestion::IncreaseWindow()），开发者通过子类重写实现自定义行为。例如，实现 BBR 拥塞控制算法时，继承 TcpCongestion 并重写窗口调整函数：

```
void TcpBbr::IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAcked) {
            // BBR特有的带宽探测逻辑
}
```

3）动态绑定与配置

通过 Config::SetDefault()或聚合接口，运行时替换默认实现类。例如，将全局 TCP 拥塞控制算法切换为自定义的 TcpBbr：

```
Config::SetDefault("ns3::TcpL4Protocol::SocketType",
                   TypeIdValue(TcpBbr::GetTypeId()));
```

3. 聚合模型：组件化构建网络实体

NS3 通过聚合机制将多个功能模块组合为完整的网络实体(如节点),支持动态增删组件并维护其关联关系。

1) 节点(Node)作为聚合容器

一个 Node 实例可聚合多个网络接口(NetDevice)、协议栈(IPv4/IPv6)、移动性模型(MobilityModel)等。例如,为节点添加 Wi-Fi 和 LTE 双网卡：

```
Ptr<Node> node = CreateObject<Node>();
Ptr<WifiNetDevice> wifiDevice = CreateObject<WifiNetDevice>();
Ptr<LteNetDevice> lteDevice = CreateObject<LteNetDevice>();
node->AddDevice(wifiDevice);
node->AddDevice(lteDevice);
```

2) 双向访问接口

聚合组件可通过 GetNode 方法反向访问所属节点,进而通过 GetObject 方法访问所属节点的其他组件。例如,NetDevice 获取节点 IP 地址：

```
Ptr<Ipv4> ipv4 = m_device->GetNode()->GetObject<Ipv4>();
Ipv4Address ip = ipv4->GetAddress(ipv4->GetInterfaceForDevice(m_device), 0).GetLocal();
```

3.1.3 扩展设计方法

NS3 提供三种扩展路径,满足不同层次的定制需求,分别为参数化扩展、算法替换式扩展和模型级扩展。

1. 参数化扩展

(1) 适用场景：调整协议参数(如 TCP 初始拥塞窗口、无线信道衰减系数)。

(2) 实现步骤：通过带参命令行修改类属性初始值。代码如下：

```
// 通过 TypeId 声明可配置属性
NS_OBJECT_ENSURE_REGISTERED(TcpNewReno);
TypeId TcpNewReno::GetTypeId() {
            static TypeId tid = TypeId("ns3::TcpNewReno")
            .SetParent<TcpCongestion>()
            .AddAttribute("InitialCwnd", "初始拥塞窗口", UintegerValue(10),
                    MakeUintegerAccessor(&TcpNewReno::m_cWnd),
                    MakeUintegerChecker<uint32_t>());
        return tid;
}
// 配置示例
./waf --run "my-script.cc --ns3::TcpNewReno::InitialCwnd=20"
```

2. 算法替换式扩展(继承基类)

(1) 适用场景：实现私有协议(如自定义路由协议、新型拥塞控制算法)。

(2) 实现步骤：继承基类(如 TcpCongestion)并重写关键虚函数；注册新类到 TypeId 系统；通过 Config::SetDefault 或聚合接口替换默认实现。代码如下：

```
class MyTcpCongestion : public TcpCongestion {
public:
      static TypeId GetTypeId();
      virtual void IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAcked) override {
                // 自定义窗口增长逻辑
```

```
        }
};
// 注册并替换默认算法
Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(MyTcpCongestion::GetTypeId
()));
```

3. 模型级扩展(新增模块)

适用场景：开发全新协议栈、自定义网络设备或引入全新通信模型。

3.2　基础模型

3.2.1　信道模型

信道(Channel)模型是 NS3 中物理层与数据链路层功能的核心抽象，负责模拟数据在介质中的传输行为(如传播时延、信号衰减、多径效应等)。它定义了网络设备(Net Device)之间的连接规则与数据交互逻辑，是构建网络拓扑的基础组件。

1. 核心功能

介质特性建模：描述信道物理属性(如带宽、延迟、误码率)，模拟真实传输介质的信号传播过程。支持添加传播损耗模型(Propagation Loss Model)和传播延迟模型(Propagation Delay Model)。

2. 关键接口方法

以下是 Channel 基类的主要接口方法：

```
class Channel : public Object {
public:
          // 获取信道连接的设备总数
          virtual uint32_t GetNDevices() const = 0;
          // 获取第 i 个设备的指针
          virtual Ptr<NetDevice> GetDevice(uint32_t i) const = 0;
          // 发送数据包(由 NetDevice 调用)
          virtual void Send(Ptr<Packet> packet, uint32_t deviceIdx,
                    Time duration, bool isBroadcast) = 0;
          // 添加传播损耗模型
          void AddPropagationLossModel(Ptr<PropagationLossModel> loss);
          // 添加传播延迟模型
          void AddPropagationDelayModel(Ptr<PropagationDelayModel> delay);
};
```

3. 常用派生类与场景

NS3 提供多种信道模型，覆盖典型网络场景，总结于表 3-2。

表 3-2　NS3 信道模型

派　生　类	应 用 场 景	扩 展 能 力
PointToPointChannel	点对点有线链路	支持自定义队列模型(Queue)和错误模型(ErrorModel)，模拟链路拥塞与误码

续表

派 生 类	应 用 场 景	扩 展 能 力
CsmaChannel	载波监听多路访问/冲突检测(Carrier Sense Multiple Access/Collision Detection，CSMA/CD)网络	可扩展冲突检测算法，或添加信道监听器捕获所有传输帧
YansWifiChannel	无线局域网(IEEE 802.11 a/b/g/n/ac)	支持添加多径衰落模型和障碍物遮挡模型
LteSpectrumChannel	蜂窝网络(LTE/5G)	集成频谱干扰模型，支持多小区频率复用与干扰分析
UanChannel	水下声学通信网络	模拟声波传播特性(如多普勒效应、水下衰减)，支持自定义调制方案

4. 应用示例

(1) 创建有线链路并配置。代码如下：

```
// 创建两个节点
NodeContainer nodes;
nodes.Create(2);
// 创建 PointToPointHelper 并设置链路属性
PointToPointHelper p2pHelper;
p2pHelper.SetDeviceAttribute("DataRate", StringValue("5Mbps")); // 设备速率为 5Mb/s
p2pHelper.SetChannelAttribute("Delay", StringValue("2ms"));     // 传输延迟为 2ms
// 安装设备并获取 NetDevice 容器
NetDeviceContainer devices = p2pHelper.Install(nodes);
// 获取信道对象(通过任意设备的 GetChannel())
Ptr<PointToPointChannel> p2pChannel = devices.Get(0)->GetChannel()->GetObject
<PointToPointChannel>();
// 添加随机误码模型(5%丢包率)
Ptr<RateErrorModel> errorModel = CreateObject<RateErrorModel>();
errorModel->SetUnit(RateErrorModel::ERROR_UNIT_PACKET);
errorModel->SetRate(0.05);                                          // 5%丢包率
// 给设备 1 安装错误模型，模拟链路误码丢包
devices.Get(1)->SetAttribute("ReceiveErrorModel", PointerValue(errorModel));
// 验证信道连接设备数(应为 2)
NS_LOG_UNCOND("Channel connected devices: " << p2pChannel->GetNDevices());
// 输出信道参数
NS_LOG_UNCOND("Channel Delay: " << p2pChannel->GetDelay()
                << ", DataRate: " << p2pChannel->GetDataRate());
```

(2) 创建无线信道并配置传播模型。代码如下：

```
// 创建 YansWifiChannel
Ptr<YansWifiChannel> wifiChannel = CreateObject<YansWifiChannel>();
// 添加传播损耗模型(Log-distance 模型)
Ptr<LogDistancePropagationLossModel> lossModel = CreateObject<LogDistancePropagationLossModel>();
lossModel->SetPathLossExponent(3.0);                    // 设置路径损耗指数
// 建议设置参考损耗(默认参考距离为 1m)
lossModel->SetReference(1, 46.6777);                    // 1m 处的参考损耗值(单位:dB)
wifiChannel->AddPropagationLossModel(lossModel);
// 添加传播延迟模型(恒定延迟)
Ptr<ConstantSpeedPropagationDelayModel> delayModel = CreateObject
<ConstantSpeedPropagationDelayModel>();
wifiChannel->SetPropagationDelayModel(delayModel);      // 注意:应使用 Set 而非 Add
```

```
// 将信道绑定到无线设备
wifiPhy.SetChannel(wifiChannel);
```

3.2.2 节点模型

节点(Node)是 NS3 中网络实体的核心容器,负责聚合物理设备(NetDevice)、协议栈(IPv4/IPv6)、移动模型(MobilityModel)以及应用层逻辑(Application),是构建网络拓扑的基本单元。其设计目标是为不同网络角色(如主机、路由器、卫星节点)提供统一抽象接口,支持灵活的功能扩展与动态配置。

1. 核心功能

(1) 设备与协议栈管理:节点作为容器可挂载多个网络接口设备(如同时支持 Wi-Fi 和 LTE 双模通信)。支持安装不同协议栈(如 IPv4/IPv6),实现异构网络互联。

(2) 移动建模:通过绑定 MobilityModel 定义节点的空间位置与运动轨迹(如静态、随机游走、轨迹跟踪)。

2. 关键接口方法

以下是 Node 类的主要接口方法:

```
class Node : public Object {
public:
  // 基础管理
  uint32_t GetId() const;                                    // 获取节点全局唯一 ID
  uint32_t GetSystemId() const;                              // 分布式仿真中的系统分区 ID
  // 设备管理
  void AddDevice(Ptr<NetDevice> device);                     // 添加网络设备
  Ptr<NetDevice> GetDevice(uint32_t index) const;
  uint32_t GetNDevices() const;                              // 获取已挂载设备数量
  // 协议栈管理
  Ptr<Ipv4> GetObject<Ipv4>() const;                         // 获取 IPv4 协议栈(模板方法)
  Ptr<Ipv6> GetObject<Ipv6>() const;                         // 获取 IPv6 协议栈
  // 移动管理
  void AggregateObject(Ptr<MobilityModel> mobility); // 绑定移动模型
  Ptr<MobilityModel> GetObject<MobilityModel>() const;
  // 应用管理
  void AddApplication(Ptr<Application> app);                 // 添加应用层实例
  Ptr<Application> GetApplication(uint32_t index) const;
  uint32_t GetNApplications() const;
  // 标签与元数据
  void SetContext(const std::string &context);               // 设置上下文标签
  std::string GetContext() const;
};
```

3. 功能扩展与派生类

NS3 的 Node 类本身为通用基类,其功能扩展主要通过聚合模型实现,而非传统继承。典型扩展模式总结于表 3-3。

表 3-3 NS3 中 Node 类的典型扩展模式

扩展能力	实现方式	应用场景
移动节点	聚合 MobilityModel(如 ConstantPosition、RandomWalk2d)	MANET、无人机网络仿真
卫星节点	聚合 SatelliteMobilityModel(需第三方模块)	低轨星座通信、星地链路建模

续表

扩展能力	实 现 方 式	应 用 场 景
虚拟化节点	聚合 VirtualNetDevice 与 LinuxStackHelper	容器网络、云原生场景仿真
SDN 交换机节点	聚合 OpenFlowSwitchNetDevice 并关闭默认路由协议	SDN 控制平面实验

4. 应用示例

创建节点并配置协议栈和移动模型、挂载设备、安装网络应用,代码如下:

```
// 创建节点
NodeContainer nodes;
nodes.Create(1);                                    // 创建单个节点
Ptr<Node> node = nodes.Get(0);
// 安装移动模型
MobilityHelper mobilityHelper;
mobilityHelper.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityHelper.Install(node);
Ptr<ConstantPositionMobilityModel> mobility =
    node->GetObject<ConstantPositionMobilityModel>();
mobility->SetPosition(Vector(0, 0, 10));      // 设置坐标
// 安装协议栈
InternetStackHelper stack;
stack.Install(nodes);
// 挂载 Wi-Fi 设备
Ptr<WifiNetDevice> wifiDevice = CreateObject<WifiNetDevice>();
node->AddDevice(wifiDevice);
// 安装 HTTP 服务器应用
Ptr<HttpServer> serverApp = CreateObject<HttpServer>();
node->AddApplication(serverApp);
serverApp->Start(Seconds(1.0));
```

3.2.3 设备模型

设备(Device)模型是 NS3 中数据链路层的核心抽象,负责将网络节点(Node)连接到信道(Channel),实现数据帧的封装、发送、接收以及介质访问控制逻辑。设备模型通过统一接口规范支持多种物理介质(如无线、有线、卫星链路),是构建异构网络的关键组件。

1. 核心功能

(1) 数据链路层封装:将网络层数据包(Packet)封装为数据帧,添加 MAC 头部(如源/目的地址、类型标识);支持分片与重组机制。

(2) 介质访问控制:实现 MAC 子层协议(如 CSMA/CD),协调多设备共享信道资源;处理冲突检测、退避算法、ACK 确认等逻辑。

(3) 信道交互:通过 Channel 接口发送和接收数据帧,支持广播、组播和单播模式;集成物理层模型,计算信号传输特性(如误码率、传输时延)。

(4) 队列管理:维护发送队列(Queue),实现流量整形(如优先级队列、令牌桶算法)。

2. 关键接口方法

NetDevice 基类定义了设备模型的通用接口,代码如下:

```
class NetDevice : public Object {
public:
```

```
  // 基础属性
  virtual std::string GetName() const;              // 设备名称(如 "eth0")
  virtual Address GetAddress() const;               // MAC 地址(如 Mac48Address)
  // 数据发送
  virtual bool Send(Ptr<Packet> packet, const Address& dest, uint16_t protocolNumber) = 0;
  // 信道绑定
  virtual void SetChannel(Ptr<Channel> channel);
  virtual Ptr<Channel> GetChannel() const;
  // 接收回调
  typedef Callback<void, Ptr<NetDevice>, Ptr<const Packet>, uint16_t, const Address&>
ReceiveCallback;
  virtual void SetReceiveCallback(ReceiveCallback cb);
  // 队列管理
  virtual Ptr<Queue<Packet>> GetTxQueue() const;
  virtual void SetTxQueue(Ptr<Queue<Packet>> queue);
  // 协议栈交互
  virtual void SetPromiscReceiveCallback(PromiscReceiveCallback cb);
};
```

3. 常用派生类与场景

NS3 提供了丰富的设备模型实现,覆盖主流网络技术,总结于表 3-4。

表 3-4 NS3 中 Devive 的常用派生类

派 生 类	应 用 场 景	扩 展 能 力
PointToPointNetDevice	点对点链路(如光纤、串口)	支持自定义队列模型(如 DropTailQueue)、错误模型(ErrorModel)
CsmaNetDevice	传统以太网(CSMA/CD)	可扩展冲突检测算法或添加嗅探器捕获所有帧
WifiNetDevice	无线局域网(IEEE 802.11 a/b/g/n/ac/ax)	支持多种物理层标准、动态速率适配
LteNetDevice	蜂窝网络(4G LTE)	集成 RRC 协议栈、支持 MIMO 配置和动态调度算法
MmWaveNetDevice	5G 毫米波通信	支持高频信道建模(如 3GPP TR 38.901)、波束管理与移动性管理
UanNetDevice	水下声学通信网络	模拟声波调制、传播延迟和多普勒效应
VirtualNetDevice	虚拟网络接口(如 Linux TAP)	与主机操作系统协议栈交互,实现虚实融合仿真

4. 应用示例

构建两个节点之间的点对点有线链路,配置队列模型为 DropTailQueue(尾部丢弃队列),并设置最大队列长度为 100 个数据包,代码如下:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"

using namespace ns3;

int main() {
    // 创建两个节点
    NodeContainer nodes;
```

```
    nodes.Create(2);
    // 配置点对点信道与设备
    PointToPointHelper p2p;
    p2p.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
    p2p.SetChannelAttribute("Delay", StringValue("2ms"));
    // 队列设置
    p2p.SetQueue("ns3::DropTailQueue", "MaxSize", StringValue("100p")); // 100 个数据包
    // 安装设备
    NetDeviceContainer devices = p2p.Install(nodes);
    // 安装协议栈
    InternetStackHelper stack;
    stack.Install(nodes);
    // 分配 IP 地址
    Ipv4AddressHelper address;
    address.SetBase("10.1.1.0", "255.255.255.0");
    Ipv4InterfaceContainer interfaces = address.Assign(devices);
    // 设置模拟时间并运行
    Simulator::Stop(Seconds(11.0));
    std::cout << "Starting simulation with DropTailQueue of 100 packets..." << std::endl;
    Simulator::Run();
    Simulator::Destroy();
    return 0;
}
```

3.2.4 路由模型

路由(Router)模型是 NS3 网络层的核心抽象,负责管理路由表、决策数据包转发路径,并支持动态路由协议(如 OSPF、AODV)或静态路由配置。其设计目标是模拟路由器/主机的网络层转发逻辑,支持从简单静态路由到复杂动态路由协议的全场景仿真。

1. 核心功能

(1) 路由表管理:维护路由表项(目的网络、下一跳、接口、优先级等),支持手动配置或动态更新;支持多路由表(如策略路由)和多路径负载均衡。

(2) 数据包转发决策:根据目标 IP 地址和路由表选择最优转发路径;实现 ICMP 重定向、生存周期(Time to Live,TTL)超时等异常处理逻辑。

(3) 动态路由协议支持:集成路由协议(如 OSPF、AODV 等)实现路由表的自动更新;支持邻居发现(Neighbor Discovery,ND)和 LSA。

2. 关键接口方法

基类接口方法(以 Ipv4RoutingProtocol 为例):

```
class Ipv4RoutingProtocol : public Object {
public:
  // 路由决策(发送方向)
  virtual Ptr < Ipv4Route > RouteOutput(Ptr < Packet > packet,
                                        const Ipv4Header &header,
                                        Ptr < NetDevice > oif,
                                        Socket::SocketErrno &sockerr) = 0;
  // 路由决策(接收方向)
  virtual bool RouteInput(Ptr < const Packet > packet,
                          const Ipv4Header &header,
                          Ptr < const NetDevice > idev,
```

```
                                UnicastForwardCallback ucb,
                                MulticastForwardCallback mcb,
                                LocalDeliverCallback lcb,
                                ErrorCallback ecb) = 0;
  // 路由表更新
  virtual void NotifyInterfaceUp(uint32_t interface) = 0;
  virtual void NotifyInterfaceDown(uint32_t interface) = 0;
  virtual void NotifyAddAddress(uint32_t interface, Ipv4InterfaceAddress address) = 0;
};
```

常用派生类方法(以 Ipv4StaticRouting 为例):

```
class Ipv4StaticRouting : public Ipv4RoutingProtocol {
public:
    // 添加静态路由表项
    void AddNetworkRouteTo(Ipv4Address network, Ipv4Mask mask,
                           uint32_t interface, Ipv4Address nextHop,
                           uint32_t metric = 0);
    // 设置默认路由
    void SetDefaultRoute(Ipv4Address nextHop, uint32_t interface,
                           uint32_t metric = 0);
    // 获取路由表内容
    void GetRoutingTable(std::vector<Ipv4RoutingTableEntry>& routes) const;
};
```

3. 常用派生类与场景

NS3 提供了多种路由协议实现,涵盖典型网络场景,总结于表 3-5。

表 3-5　NS3 中多种路由协议及实现类

派　生　类	应 用 场 景	功 能 描 述
Ipv4StaticRouting	静态路由配置(小型网络、测试环境)	静态路由,手动添加/删除路由表项,适合固定拓扑仿真
Ipv4GlobalRouting	全局路由(基于链路状态的最短路径)	自动生成路由表,适用于大规模固定网络(如数据中心)
AodvRouting	移动自组网	AODV 路由,支持节点移动性(如车载网络、无人机集群)
OlsrRouting	移动自组网	OLSR 路由,适用于高动态拓扑
RipRouting	小型企业网	基于距离矢量的路由协议(Routing Information Protocol,RIP),支持自动路由更新
NixVectorRouting	高效内存路由(不需要显式路由表)	适用于大规模并行仿真(如超算网络),减少内存占用
Ipv4ListRouting	多路由协议优先级组合(如静态路由+动态路由)	允许同时运行多个路由协议,按优先级选择最佳路由

4. 应用示例

(1) 静态路由配置。

场景说明:构建包含 3 个节点的链式拓扑(Node0↔Node1↔Node2),通过静态路由实现跨节点通信。代码如下:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
```