---

ns-3 Tutorial (html version)

For a pdf version of this tutorial, see http://www.nsnam.org/docs/tutorial.pdf.

This is an ns-3 tutorial. Primary documentation for the ns-3 project is available in four forms:

- ns-3 Doxygen/Manual: Documentation of the public APIs of the simulator
- Tutorial (this document)
- Reference Manual: Reference Manual
- ns-3 wiki

This document is written in GNU Texinfo and is to be maintained in revision control on the ns-3code server. Both PDF and HTML versions should be available on the server. Changes to the document should be discussed on the ns-developers@isi.edu mailing list.



# 1 简介 http://www.nsnam.org/docs/release/tutorial/tutorial_1.html#Introduction

NS-3 的主文档由四部份组成：

●··NS-3 Doxygen 手册 模拟器公共 API 的说明文档

●··NS-3 教程 （本教程 Tutorial）

●··NS-3 参考手册 （Reference Manual）

●··NS-3 维基百科

本指南的目的是向 NS-3 新用户以结构化的方式介绍 NS-3 系统。常常，新用户很

难通过繁冗的参考手册收集到对当前仿真任务有用的实质信息。在本教程中，我们将通过几个仿真实例向读者介绍和阐释 NS-3 的主要概念和特点。

随着本教程的深入，我们会介绍完整的 NS-3 文档，并指出源代码的具体位置，以便于对 NS-3 软件系统运行机制感兴趣的研究者做深入的钻研。

开始之前，有几个要点需要读者注意：

●‥NS-3 并不是 NS-2 的扩展，而是一个全新的模拟器。虽然二者都由 C++编写，但 NS-3 并不支持 NS-2 的 API。NS-2 中的一些模块已经被移植到了 NS-3。在 NS-3 开发的过程中，NS-3 项目组会继续维护 NS-2，同时也会研究过渡和整合机制。

●‥NS-3 是开源的。NS-3 项目努力为研究者提供一个开放的环境来共享他们自己的软件。

## 1.1 致 NS-2 用户

对于熟悉 NS-2 的读者来说，NS-3 和 NS-2 最明显的区别是脚本语言的选择。NS-2 使用 OTcl 脚本语言，仿真的结果可以通过网络动画器 nam（Network Animator nam）来演示。在 NS-2 中，如果仅使用 C++语言而不用 OTcl，仿真过程是不可能运行起来的(即，只有 main()函数而没有任何 OTcl 语句)。另外，NS-2 的许多模块由 C++编写，其他的用 OTcl 语言编写。而在 NS-3 中，仿真器全都由 C++编写，仅仅带有可选择性的 Python 语言绑定。因此，仿真脚本可以由 C++或者 Python 语言编写。某些仿真结果可以通过 nam 演示，但是新的动画演示器也正在开发之中。由于 NS-3 可以生成 pcap 包 trace 文件，也可以利用其他工具通过 trace 文件来分析仿真过程。在本教程中，我们先重点讲解使用 C++编写脚本，并通过 trace 文件来分析仿真结果。

NS-3 和 NS-2 也有一些相似之处 (比如二者都是基于 C++对象，一些 NS-2 的模块已移植到了 NS-3 上)。在本教程中，我们将强调 NS-3 和 NS-2 的区别。

我们经常听到一个疑问：“我到底是要继续使用 NS-2，还是转向 NS-3 呢？”

答案视情况而定。虽然 NS-3 现在还没有包含所有的 NS-2 模块，但是另一方面，NS-3 也有一些新的功能(比如，能正确地处理节点上的多重接口，使用 IP 地址，与因特网协议和设计保持一致，以及更加详细的 802.11 模块等)。NS-2 的模块可以被移植到 NS-3 中（移植帮助文档正在准备之中）。NS-3 的多个前端也正在积极开发中。同时，NS-3 开发者也相信(部分早期用户也已证明)NS-3 已经可以充分使用，也是用户进行新仿真项目时的一个富有吸引力的选择。

## 1.2 共享 contributing

NS-3 是由学术研究者开发和使用的用于网络科研和教学的仿真器。它依赖于研究者们持续不懈地努力，开发新模块、调试和维护已有模块、并共享成果。为了鼓励研究者像支持 NS-2 那样支持 NS-3，我们希望 NS-3 的开发者遵守下面几条规则：

●··基于 GNU GPLv2 兼容性的开放源码许可

●··维客

●··共享代码(Contributed Code) 页（类似于 NS-2 的共享代码页）

●··Src/contrib. 目录(我们会保留您共享的代码)

●··开放的错误追踪器（bug tracker）

●··NS-3 开发者会很乐意帮助潜在的代码共享者们，提供 NS-3 仿真器的入门途径(请联系我们)

我们也意识到，如果您正在读本教程，对 NS-3 项目的共享或许还不在您当前最关心的事，但是我们希望您知道，“共享”是 NS-3 项目的灵魂。即使是给我们写一个关于您使用 NS-3 的经验便条（例如：“这个教程的章节条理不够清晰”），通知某些文档已过时等等，我们将会感激之至。

## 1.3 教程内容组织

本指南假定新用户可能会顺着以下思路了解 NS-3：

●··尝试下载和编译 NS-3

●··尝试运行几个简单的示例程序

●··查看仿真结果并试图调整仿真

因此，我们将大体按照这个顺序来组织本教程。

# 2 资源

http://www.nsnam.org/docs/release/tutorial/tutorial_5.html#Resources

## 2.1 网络资源

NS-3 用户有必要知道几个重要的网站：

主站点位于 http://www.nsnam.org， 提供 NS-3 系统的基本信息。

详细文档位于主站点的 http://www.nsnam.org/documents.html. 您也可以从这个网页上得到系统架构的相关信息。

维基百科网页 http://www.nsnam.org/wiki 可以作为 NS-3 主站点的补充。您可以在那里找到用户和开发者的 FAQs，以及相关问题的解决途径，第三方的共享代码、论文等等。

NS-3 的源码可以在 http://code.nsnam.org 找到。读者也可以在名为 ns3-dev 的源码仓库找到当前的 NS-3 开发树。还有 NS-3 的之前发行版本和最新测试版本的代码。

## 2.2 源码管理系统 Mercurial

复杂的软件系统需要一种途径，用于管理和组织对现有代码和文档的修改。有很多种方法可以实现这种管理，读者可能已听说过某些版本控制软件，CVS（Concurrent Version System）或许是其中最常见的一个。

NS-3 项目采用 Mercurial 系统作为它的源码管理系统。尽管读者在阅读本教程时不需要知道太多的 Mercurial 相关知识，但我们建议读者能够熟悉 Mercurial，并用于查看 NS-3 源码。Mercurial 的网址为 http://www.selenic.com/mercurial/，读者可以从上面获取到这个软件配置管理系统(Software Configuration management, SCM)的二进制程序和源码。Mercurial 的开发者 Selenic 提供了一个 Mercurial 教程，网址为 http://www.selenic.com/mercurial/wiki/index.cgi/Tutorial/，以及快速入门指南：http://www.selenic.com/mercurial/wiki/index.cgi/QuickStart/ .

在 NS-3 的主页上，读者也可以获取到有关 Mercurial 和 NS-3 配合使用的最常用信息。

## 2.3 编译系统 Waf

读者下载 NS-3 的源码到本地系统之后，需要对源码进行编译来生成可执行程序。正如源码管理方式多种多样，编译源码也有多种工具。最常用的工具是 make. Make 最出名的一点：它可能是编译大型和高可配置型系统最难的一种方法。因此，有很多替代工具被开发出来。最近，大型高可配置系统的编译工具大多选择用 Python 语言来开发。

NS-3 的编译系统采用了 Waf。它是用 Python 开发的新一代编译管理系统。读者不必掌握 python，即可编译现有的 NS-3 项目。如果读者想要扩展现有的 NS-3 系统，大多数情况只需了解 Python 知识的很少且非常直观的一个子集。

对于想了解 Waf 细节的读者，可以访问 http://code.google.com/p/waf/ .

## 2.4 开发环境

正如以上所述，NS-3 的脚本由 C++或者 Python 编写。从 NS-3.2 开始，NS3 的 API 提供了 python 语言接口，但是所有的模块都是由 C++编写的。这里，我们假定读者掌握 C++知识和了解面向对象的相关概念。我们将在用到一些高级的概念或者读者可能不熟悉的语言特性、习惯用语或设计模式时适当地花些时间复习它们。但是我们也不希望

本教程变成 C++教程，所以我们希望读者能够掌握基本的 C++命令。在网站上和书籍中，你可以找到无数的关于 C++知识的信息。

如果读者是个 C++新手，那么您在继续阅读本指南之前可能需要找一些 C++教程或者网站，至少必须熟悉一下 C++的基本语言特征。例如，Cplusplus 教程。

NS-3 系统开发过程中使用了许多的 GNU 工具链（toolchain）组件。所谓软件的工具链是指在给定环境中可用编程工具的集合。如果读者想要快速地了解一下 GNU 工具链所包含的内容，请浏览 http://en.wikipedia.org/wiki/GNU_toolchain．NS-3 使用 gcc，GNU binutils，以及 gdb。不过，我们并不使用 GNU 编译系统工具（build system tools），既不用 make，也不用 autotools，而是使用 Waf 来作为编译管理工具。

通常，NS-3 使用者的工作环境为 Linux 或者类 Linux 系统。对于 Windows 环境，有几种可以不同程度模拟 Linux 环境的软件，比如 Cygwin。NS-3 支持在 Cygwin 环境下的开发。Windows 用户可以浏览 http://www.cygwin.com/ 获取该软件(虽然有许多工程维护者使用 MinGW，但是 MinGW 现在还没有得到官方支持)。Cygwin 可以提供许多流行的 Linux 系统命令。但是，某些情况下它也会出现问题，因为它毕竟只是 Linux 系统的模拟。Cygwin 和 Windows 中其他程序的交互也有可能会导致程序出现问题。

如果读者正在使用 Cygwin 或者 MinGW；并使用着 Logitech 的某些软件产品，我们或许可以让您少点麻烦：建议您去看一看 MinGW FAQ。

搜索 Logitech 并阅读 FAQ 条目：“为什么当我编译源码时，make 经常崩溃，留下一个 sh.exe.stackdump 文件”。无论您相信与否，当运行 Logitech 时，Logitech 进程监视器潜入了每个正在系统中运行的动态连接库(DLL)当中。它可能导致您的 Cygwin 或者 MinGw 的动态连接库奇怪地中止，常常也会阻止调试器的运行。所以当运行 Cygwin 的时候，一定要小心您的 Logitech 软件。

替代 Cygwin 的一种选择是安装虚拟机，比如在 VMware 上安装 Linux 虚拟机。

## 2.5 套接字编程

我们假定读者对本教程所举例子中的 Berkeley 套接字 API 基本熟悉。如果您不了解套接字，我们建议您学习一下这些 API 和一些常见的使用例程。TCP/IP Sockets in C 这本书可以帮助您很好地理解 TCP/IP 套接字。

网站 http://cs.baylor.edu/~donahoo/practical/CSockets/ 包含了 Socket in C 书中所举例子的源码。

如果读者理解了该书中的前四章(如果读者没有这本书的话，可以看上面网站中的源代码)，您会更好的理解本教程的内容。这里还有一本关于多播套接字(Multicast Sockets)的书籍（Multicast Sockets, Makofske and Almeroth）。如果您想学习本书中有关多播的例子，该书里面有些资料您可能需要了解。

# 3 NS3 快速上手 http://www.nsnam.org/docs/release/tutorial/tutorial_11.html#Getting-Started

## 3.1 下载 ns-3

从现在起,我们假定读者使用的工作环境为 Linux 或者仿 Linux 环境(Linux, Cygwin 等等.) 并且已经安装了可用的 GNU 工具链,而且还安装了 Mercurial(分布式版本控制系统)和 Waf 软件。细节已经在 ns-3 网页中详述过,参见以下链接:
http://www.nsnam.org/getting_started.html.

ns-3 源码可以在网站 http://code.nsnam.org 上的 Mercurial 源码库下载到.你也可以从链接 http://www.nsnam.org/releases/处下载一个 tar 格式压缩包,或者直接使用 Mercurial 从源码库下载。除非有特殊需要,我们推荐使用 Mercurial 从源码库下载。tar 格式压缩包下载,请参见本节最后部分。最简单的方法就是使用 Mercurial 源码库下载一个 ns-3-allinone 压缩包,此压缩包内含一套脚本集来管理各种子系统下的 ns-3 下载和安装。我们推荐你使用这个压缩包来简化你的 ns-3 安装。

### 3.1.1 使用 Mercurial 下载 ns-3

作为练习,我们首先在 home 目录下建立一个目录并取名为 repos,用来存放本地 Mercurial 源码库,注意:在本教程随后内容中,我们假定你已经这样做了。如果使用如下的方法,可以在 Linux 的 shell 中下载到一份 ns-3-allinone 软件包(假定你已经安装了 Mercurial):

```
cd
mkdir repos
cd repos
hg clone http://code.nsnam.org/ns-3-allinone
```

当 Mercurail 的 hg 命令执行后,可以看到如下结果:

```
destination directory: ns-3-allinone
requesting all changes
adding changesets
adding manifests
adding file changes
added 31 changesets with 45 changes to 7 files
```

```
7 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

当 clone 命令运行结束以后，在前述建立的 repos 目录下，会出现一个 ns-3-allinone 目录，而且含有如下文件：

```
build.py*  constants.py  dist.py*  download.py*  README  util.py
```

注意：你实际上仅仅下载了一些 Python 脚本，下一步就是利用这些脚本根据需要下载并来安装 ns-3 软件包。如果你访问如下链接：http://code.nsnam.org/ 就会发现若干源码库，其中很多是 ns3 开发团队专用的。其中在源码库 org/ 你会发现名为 ns-3.1 的源码库，这是 ns-3 的第一个稳定版本。还有一些分散的源码库名为 ns-3.1-reftraces，它为 ns-3.1 保留了参考记录。保持这些文件的一致性是非常重要的，尤其是当你想对源码库做一个回归测试时。至少做一次测试来验证所有的程序都正确编译了。

当前的开发版 ns-3 的快照存放在 http://code.nsnam.org/ns-3-dev/中；相关的参考记录存放在链接 http://code.nsnam.org/ns-3-dev-ref-traces/ 中。 ns3 开发人员会尽量保持源码库中的代码处于一致，工作的状态，但是他们仍在开发中，有一些未发布过的代码。所以如果你不需要最新的特性的话还是考虑使用发行版。

由于发布版的版本号在变化中，我在指南中还是继续使用通常不变的 ns-3-dev，但是你可以根据自己的需要选择其他的版本，并替换这里的”ns-3-dev”，(例如, ns-3.6 或 ns-3.6-ref-traces)，在下文中，你可以通过访问源码库列表或者访问 ns3 开始网页找到最新的 ns-3 发布版软件。

当你从源码库下载完后，继续切换进入你自己建立的 ns-3-allinone 目录中。 我们现在使用 download.py 脚本来下载 ns-3 需要使用的各个部件。继续在你的 shell 中输入以下命令（如果你想使用任意发行版你可以将 ns-3-dev 替换为你选择的发行版的名字，例如"ns-3.6" 和"ns-3.6-reftraces")。

```
./download.py -n ns-3-dev -r ns-3-dev-ref-traces
```

注意，-n 选项的默认参数为 ns-3-dev，-r 选项的默认参数为 ns-3-dev-ref-traces，所以上述命令中这两个选项的参数实际上是多余的。我们使用这个例子来描述如何指定源码库。你只需简单键入如下命令就可以使用默认参数来下载 ns-3-dev：

```
./download.py
```

当 hg (Mercurial)命令执行时，你可以看到如下的信息：

```
#
# Get NS-3
#
Cloning ns-3 branch
=> hg clone http://code.nsnam.org/ns-3-dev ns-3-dev
```

```
requesting all changes
adding changesets
adding manifests
adding file changes
Chapter 3: Getting Started 8
added 4634 changesets with 16500 changes to 1762 files
870 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

这些输出信息显示下载脚本已经从源码库中下载到了实际的 ns-3 源码，紧接着，你就会看到这样的信息：

```
#
# Get the regression traces
#
Synchronizing reference traces using Mercurial.
=> hg clone http://code.nsnam.org/ns-3-dev-ref-traces
ns-3-dev-ref-traces
requesting all changes
adding changesets
adding manifests
adding file changes
added 86 changesets with 1178 changes to 259 files
208 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

这表明下载脚本为你下载了参考记录文件。下载脚本被设计成自适应的，它能检测在一些系统平台上某些 ns-3 模块不被支持，在你的系统平台上，你可能看不到这些系统不支持的源码被下载。然而，在大多数系统平台上这样的过程会显示如下：

```
#
# Get PyBindGen
#
Required pybindgen version: 0.10.0.640
Trying to fetch  pybindgen; this will fail if no network connection
is available. Hit Ctrl-=> bzr checkout -rrevno:640
https://launchpad.net/pybindgen pybindgen
Fetch was successful.
```

这些信息显示了下载脚本在为你下载 Python 绑定生成器的过程，下一步你就可能看到如下的信息(不同的系统平台表现有所不同)，

```
#
# Get NSC
#
Required NSC version: nsc-0.5.0
Retrieving nsc from https://secure.wand.net.nz/mercurial/nsc
=> hg clone https://secure.wand.net.nz/mercurial/nsc nsc
requesting all changes
adding changesets
adding manifests
adding file changes
added 273 changesets with 17565 changes to 15175 files
10622 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

这些信息显示了下载脚本在为你下载网络仿真器支架程序(NSC)的过程。

当复制命令结束，你在~/repos/ns-3-allinone 目录下会有几个新目录:

```
build.py* constants.pyc download.py* ns-3-dev-ref-traces/ pybindgen/ util.py
constants.py dist.py* ns-3-dev/ nsc/ README util.pyc
```

继续进入 ns-3-dev 目录，你会见到如下的文件:

```
AUTHORS examples/ regression/ scratch/ waf*
bindings/ LICENSE regression.py src/ waf.bat*
CHANGES.html ns3/ RELEASE_NOTES utils/ wscript
doc/ README samples/ VERSION wutils.py
```

现在可以准备编译 ns-3 软件了。

### 3.1.2 使用 Tarball 下载 ns-3

使用 tarball 下载 ns-3 比使用 Mercurail 下载 ns-3 要简单一些，因为需要下载的各个部分都已经预先被打包在一个压缩包中了，你只需要选择一个发行版

版本，下载并且解压就可以。如上所述，在 Mercurial 下载方法中，是在 home 目录下创建一个 repos 目录，保存本地 Mercurial 源码库。你也可以建立一个 tarball 目录来下载。（注意:本教程后面会假定你把它下载到了 repoa 目录，所以请留意这个选择）。如果你选择了 tarballs 方法下载，你可以键入如下命令来下载一份 ns-3 的发行版（当然，可以根据需要选择你想要的版本):

```
cd
mkdir tarballs
cd tarballs
wget http://www.nsnam.org/releases/ns-allinone-3.6.tar.bz2
tar xjf ns-allinone-3.6.tar.bz2
```

如果你切换到 ns-allinone-3.6 目录，你会看到下述文件:

```
build.py* ns-3.6/ nsc-0.5.1/ README
constants.py ns-3.6-ref-traces/ pybindgen-0.12.0.700/ util.py
```

现在就可以编译 ns-3 软件包了。

## 3.2 编译 ns-3

### 3.2.1 使用 build.py 编译

如果你是第一次编译 ns-3 软件包，建议使用 allinone 环境，它会为你以最常用的方式配置工程。

切换到你在上文下载一节中创建的目录下。如果你使用 Mercurial 下载，请进入~/repos 目录下的 ns-3-allinone 目录，如果你使用 tarball 下载，找到~/tarballs 目录下类似 ns-allinone-3.6 的目录，键入如下的命令，并请耐心等待：

```
./build.py
```

编译脚本开始编译下载的 ns3 时，你会看到大量常见的编译器输入信息。最后你会看到如下编译成功的好消息：

```
Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone/ns-3-dev/build’
’build’ finished successfully (2m30.586s)
```

一旦工程编译好，你就可以不再使用 ns-3-allinone 脚本包了。你已经从其中获取必要的东西，现在是你直接同 Waf 交互的时候了，它位于 ns-3-dev 目录中，并不在 ns-3-allinone 目录中。切换到 ns-3-dev 目录下(或者你下载的版本的相应目录下)：

```
cd ns-3-dev
```

3.2.2 使用 Waf 编译

我们使用 waf 来配置和编译 ns-3 工程软件包，这一点不是严格必须的，但是做一个简单的介绍是很有必要的，起码可以了解如何修改工程的配置。也许你能做出的最有用的配置修改就是编译工程的优化版本了，默认你会将你的工程编译为调试版本，让我们来使工程做出优化的编译。

为了通知 waf 进行优化编译，你需要执行如下的命令：

```
./waf -d optimized configure
```

这会 Waf 可能会切换到其他目录收集信息. 当编译系统检查各种依存关系时，你可以看到类似如下的输出结果：

```
Checking for program g++ : ok /usr/bin/g++
Checking for program cpp : ok /usr/bin/cpp
Checking for program ar : ok /usr/bin/ar
Checking for program ranlib : ok /usr/bin/ranlib
Checking for g++ : ok
（鉴于长度限制，省略）
Checking for program valgrind : ok /usr/bin/valgrind
---- Summary of optional NS-3 features:
Threading Primitives : enabled
Real Time Simulator : enabled
Emulated Net Device : enabled
GNU Scientific Library (GSL) : enabled
Tap Bridge : enabled
GtkConfigStore : enabled
XmlIo : enabled
SQlite stats data output : enabled
Network Simulation Cradle : enabled
Python Bindings : enabled
Python API Scanning Support : enabled
Use sudo to set suid bit : not enabled (option --enable-sudo not selected)
Build examples and samples : enabled
```

```
Static build : not enabled (option --enable-static not selected)
'configure' finished successfully (2.870s)
```

需要注意输入的最后一部分。部分 ns-3 选项并不是默认的，需要底层系统的支持才能够顺利运行。例如：为了运行 XmlTo, 系统必须安装 libxml-2.0 库，如果没有发现这个库，相应的 ns-3 特性就不会被激活，这会以消息显示出来。另一个需要注意的地方是：对于一些特定的程序，需要使用 sudo 命令来设置该程序的。这个是默认设置为关的，所以此性质会显示出未激活，现在我们可以进一步返回到调试编译上来：

```
./waf -d debug configure
```

此时编译系统被配置好，你可以编译 ns-3 程序的 debug 版本了，只需简单的命令如下：

```
./waf
```

一些 waf 命令在编译阶段是很重要的，另一些命令是在配置阶段有用的，例如，如果你想使用 ns-3 的仿真（emaulation）特性，你一定要通过上述的 sudo 来设置用户标识位（suid）为有效位。这将是一个配置命令，你可以使用如下命令通知 Waf 来重新设置：

```
./waf -d debug --enable-sudo configure
```

如果你这样做,waf 会运行 sudo 以 root 的权限来改变仿真代码的 socket 创建程序。在 waf 中还有许多其它的配置和编译选项可用。若要察看详细的选项,键入以下命令:

```
./waf --help
```

我们将会在下一节中使用测试相关的命令。好的，你已经编译 ns-3 两遍了，现在你知道如何修改配置和编译优化代码了。

## 3.3 测试 ns-3

你可以通过运行“**./test.py -c core**”脚本进行 **ns-3** 软件包单元测试,

```
./test.py -c core
```

这些测试可以被 **waf** 并行执行的，最后你可以看到如下的结果：

```
47 of 47 tests passed (47 passed, 0 failed, 0 crashed, 0 valgrind errors)
```

这是条相当重要的信息。

实际中，你看到的可能是类似于如下的信息：

```
Waf: Entering directory '/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory '/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (1.799s)
PASS: TestSuite ns3-wifi-interference
PASS: TestSuite histogram
PASS: TestSuite sample
PASS: TestSuite ipv4-address-helper
PASS: TestSuite devices-wifi
```

**PASS: TestSuite propagation-loss-model**
**...**
**PASS: TestSuite object**
**PASS: TestSuite random-number-generators**
**47 of 47 tests passed (47 passed, 0 failed, 0 crashed, 0 valgrind errors)**
用户通常可以运行此命令来检查 **ns-3** 软件包是否正确编译了。

你也可以使用我们的回归测试包来保证你的发行版和工具链产生的二进制文件会产生同参考输出文件相同的结果。你在运行/download.py 过程中会下载这些参考记录。（警告：**ns-3.2** 和 **ns-3.3** 发布版不使用 ns-3-allinone 环境，当你进行回归测试时需要在线环境，因为在直接运行测试之前它们需要同一个在线源码库进行参考记录的动态同步）。

在回归测试中 **Waf** 会允许一系列的测试，这些测试会产生我们所说的记录文件，记录文件的内容会和参考记录相对比，如果它们相同，那么回归测试会报告一个通过状态(PASS)；如果回归测试失败你会看到一个失败标记（**FAIL**），和一个指针指向有问题的记录文件和它的联合参考记录文件，还附带一个差异参数和选项以便检查哪里出现错误了。如果在一个 **pcap** 文件中出现错误，在对比之前常使用 **tcpdump** 来将 **pcap** 文件转换为文本文件。
如果需要的支持缺失，某些回归测试会被跳过，测试状态会被标记为跳过(SKIP)。

注意：回归测试也是并行的，所以产生的信息也可能是交叉出现的。
要进行回归测试，需要给 **Waf** 提供回归标记：
**./waf --regression**
可以看到如下信息显示进行了多项测试：
**Entering directory ‘/home/craigdo/repos/ns-3-allinone/ns-3-dev/build’**
**[647/669] regression-test (test-csma-bridge)**
**[648/669] regression-test (test-csma-broadcast)**
**[649/669] regression-test (test-csma-multicast)**
**[650/669] regression-test (test-csma-one-subnet)**
**PASS test-csma-multicast**
**[651/669] regression-test (test-csma-packet-socket)**
**PASS test-csma-bridge**
**...**
**Regression testing summary:**
**PASS: 22 of 22 tests passed**
**Waf: Leaving directory ‘/home/craigdo/repos/ns-3-allinone/ns-3-dev/build’**
**’ build’ finished successfully (25.826s)**
如果你想看看在回归测试中有哪些项目测试的例子，可以使用如下命令：
**cd build/debug/regression/traces/second.ref**
**tcpdump -nn -tt -r second-2-0.pcap**
这些结果对于熟悉 **tcpdump** 或者网络嗅探器是很清楚的，在本教程的后半部分我们会更多地提及 **pcap** 文件。

在一切完成后，请用 **cd** 命令返回顶层 **ns-3** 目录：

```
cd ../../../../..
```

## 3.4 运行第一个脚本

我们通常使用 Waf 运行脚本，这将使编译系统正确设置共享库的路径，并保证这些共享库在运行时可用，如果要运行一个程序，只需在 Waf 加入—run 选项即可，让我们在 ns-3 环境下运行常见的 hello world 程序试一下:

```
./waf --run hello-simulator
```

Waf 首先检查程序正确编译了，而且还可以根据需要重新执行编译。

Waf 执行了此程序，并输出如下信息：

```
Hello Simulator
```

祝贺你. 你现在是一名 ns-3 用户了!

如果你想在其他工具下如 gdb 或者 valgrind 下运行程序，可参见本 Wiki 条目。

# 4 概念概述（Conceptual Overview）

http://www.nsnam.org/docs/release/tutorial/tutorial_16.html#Conceptual-Overview

## 4.1 关键的抽象概念（Key Abstractions）

在本节中，我们将回顾一些常用的网络术语，以及它们在 ns-3 的特定含义。

### 4.1.1 节点

在因特网术语中，任何一台连接到网络的计算设备被称为主机，亦称为终端。 ns -3 是一个网络模拟器，而非一个专门的因特网模拟器，为此我们避开术语“主机”，因为这个词太容易让人联想到因特网和及其相关协议。因此，我们选用了一个来源于图论，在其他网络模拟器中亦广泛使用的术语：节点。

ns-3 中基本计算设备被抽象为节点。节点由 C++中的 Node 类来描述。Node 类提供了用于管理仿真器中网络组件表示的各种方法。

你应该将节点设想为一台可以添加各种功能的计算机。为了使一台计算机有效地工作，我们可以给它添加应用程序，协议栈，外设卡及驱动程序等。NS3 采用了与此相同的模型。

#### 4.1.2 Application

#### 4.1.2 应用程序

计算机软件通常可分为两大类：系统软件和应用软件。系统软件根据计算模型配置和管理计算机中的各种资源，例如内存，处理器周期，硬盘，网络等。系统软件通常并不直接使用这些资源来完成用户任务。用户往往需要运行应用程序来完成一些特定的任务，而应用程序需要使用由系统软件控制的资源。

通常，系统软件和应用软件的界线表现为特权级的变化，而这种变化是通过操作系统的自陷功能（operating system traps）来实现的。在 ns-3 中并没有真正

的操作系统的概念，更没有特权级别或者系统调用的概念。然而，我们有应用程序的概念。正如“现实世界”中在计算机上运行应用程序以执行各种任务一样，ns-3 仿真环境中的应用程序在节点上运行来驱动模拟过程。

在 ns-3 中，需要被仿真的用户程序被抽象为应用。应用在 C++中用 Application 类来描述。这个类提供了管理仿真时用户层应用的各种方法。开发者应当用面向对象的方法自定义和创建新的应用。在本教程中，我们会使用 Application 类的实例：UdpEchoClientApplication 和 UdpEchoServerApplication。也许你已经猜到了，这些应用程序包含了一个 client/server 应用来发送和回应仿真网络中的数据包。

#### 4.1.3 信道

在现实世界中，人们可以把计算机连接到网络上。通常我们把网络中数据流流过的媒介称为信道。当你把以太网线插入到墙壁上的插孔时，你正通过信道将计算机与以太网连接。在 ns-3 的模拟环境中，你可以把节点连接到代表数据交换信道的对象上。在这里，基本的通信子网这一抽象概念被称为信道，在 C++中用 Channel 类来描述。

Channel 类提供了管理通信子网对象和把节点连接至它们的各种方法。信道类同样可以由开发者以面向对象的方法自定义。一个信道实例可以模拟一条简单的线缆（wire），也可以模拟一个复杂的巨型以太网交换机，甚至无线网络中充满障碍物的三维空间。

我们在本教程中将使用几个信道模型的实例，包括：CsmaChannel, PointToPointChannel 和 WifiChannel。举例来说，CsmaChannel 信道模拟了用于一个可以实现载波侦听多路访问通信子网中的媒介。这个信道具有和以太网相似的功能。

#### 4.1.4 网络设备

以前，如果想把一台计算机连接到网络上，你就必须买一根特定的网络线缆，并在你的计算机上安装称为外设卡的硬件设备。能够实现网络功能的外接卡被称为网络接口卡（网卡）。现在大多数计算机出厂时已经配置了网卡，所以用户看不到这些的模块。

一张网卡如果缺少控制硬件的软件驱动是不能工作的。在 Unix（或者 Linux）系统中，外围硬件被划为 “设备”。设备通过驱动程序来控制，而网卡通过网卡驱动程序来控制。在 Unix 和 Linux 系统中，网卡被称为像 eth0 这样的名字。

在 ns-3 中，网络设备这一抽象概念相当于硬件设备和软件驱动的总和。NS3 仿真环境中，网络设备相当于安装在节点上，使得节点通过信道和其他节点通信。像真实的计算机一样，一个节点可以通过多个网络设备同时连接到多条信道上。

网络设备由 C++中的 NetDevice 类来描述。NetDevice 类提供了管理连接其他节点和信道对象的各种方法，并且允许开发者以面向对象的方法来自定义。我们在本教程中将使用几个特定的网络设备的实例，它们分别是 CsmaNetDevice,

PointToPointNetDevice, 和 WifiNetDevice。正如以太网卡被设计成在以太网中工作一样，CsmaNetDevice 被设计成在 csma 信道中工作，而 PointToPointNetDevice 在 PointToPoint 信道中工作，WifiNetNevice 在 wifi 信道中工作。

### 4.1.5 拓扑生成器

在现实的网络中，你会发现主机已装有（或者内置）的网卡。在 ns-3 中你会发现节点附加了网络设备。在大型仿真网络中，你需要在节点、网络设备和信道之间部署许多连接。

既然把网络设备连接到节点、信道，配置 IP 地址等等事情在 ns-3 是很普遍是任务，那么我们干脆提供了“拓扑生成器”来使这个工作变得尽可能的容易。举例来说：创建一个网络设备，配置一个 MAC 地址，把此网络设备装载到节点上，设置节点的协议栈，以及连接网络设备到一个信道，这些事情都需要许多分立的 ns-3 核心操作。而当需要把许多设备连接到多点信道，在网际间将单个网络进行连接，则需要对 ns-3 核心进行更多更多的分立操作。我们提供了拓扑生成器来整合这些大量分立的步骤，使其成为一个简单易用的操作。很明显，这样做可以极大地方便用户。

## 4.2 第一个 ns-3 脚本

如果你按照前面所述下载了 ns3 仿真系统，你会发现在你的根目录下的“repos”的文件夹里有一份 ns-3 的发行版。进入那个目录，你可以看到一个如下的文件目录：

```
AUTHORS       doc/       README          RELEASE_NOTES  utils/    wscript
bindings/     examples/  regression/     samples/       VERSION   wutils.py
build/        LICENSE    regression.py   scratch/       waf*      wutils.pyc
CHANGES.html  ns3/       regression.pyc  src/           waf.bat*
```

进入 examples/tutorial 目录。你会发现一个叫 first.cc 的文件。这一个脚本会在两个节点间创建一个简单的点到点的连接，并且在这两个节点之间传送一个数据包。让我们一行一行的分析一下这个脚本，下面就用你钟爱的编辑器打开 first.cc 吧。

### 4.2.1 样本

在文件中的第一行是 emacs 模式行。这行告诉了 emacs 我们在源代码中使用的预定格式（代码风格）

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
```

这向来是一个有点争议的主题，所以我们不妨现在就把它提出来。像许多大型的软件开发项目一样，ns-3 项目采用了一套所有贡献代码必须遵守的代码风格。如果你想给 ns-3 贡献代码，你最终需要遵守在文件 doc/codingstd.txt 中说明的或者遵守在 ns-3 项目的网页 here 上显示的代码标准。

我们建议你在使用 ns-3 的代码的时候最好适应这个标准。所有的开发团队和贡献者带着不同程度的抱怨来做到了这一点。如果你使用 emacs 编辑器的话，emacs 模式行会使代码格式符合规范更加容易。

Ns-3 仿真器使用了 GNU General Public License 许可。你会在 ns-3 的每一个文件头看到相应的 GNU 法律条文。通常你会在 GPL 内容的上方看到一个相关机构的版权声明，而在 GPL 内容的下方会有相应的作者列表。

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */
```

### 4.2.2 模块包含

代码一般是以一系列的 include 声明开始的：

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

#include "ns3/simulator-module.h"

#include "ns3/node-module.h"

#include "ns3/helper-module.h"
```

为了帮助高层的脚本用户处理大量的系统中的 include 文件，我们把所有的包含文件，根据模块功能，进行了大致的分类。我们提供了一个单独的 include 文件，这个文件会递归加载所有会在每个模块中会被使用的 include 文件。我们给你提供了按大致功能分类的一组 include 文件，在使用时只需选择包含这几个

包含文件(include 文件)，而非考虑复杂的依赖关系，在寻找所需要的头文件上花费不必要的时间。这不是最有效地方法但很明显让编写脚本文件容易多了。

在编译的过程中，每一个 ns-3 的 include 文件被放在 build 目录下一个叫 ns3 的目录中，这样做可以避免 include 文件名的冲突。ns3/core-module.h 与 src/core 目录下的 ns-3 模块相对应。如果你查看 ns3 目录会发现大量的头文件。当你编译时，Waf 会根据配置把在 ns3 目录下的公共的头文件放到 build/debug 或者 build/optimized 目录下。Waf 也会自动产生一个模块 include 文件来加载所有的公共头文件。

当然，既然你一步步遵循着这个手册走的话，你可能已经使用过如下命令：

./waf -d debug configure

来配置工程以完成调试工作。你可能同样使用了如下命令：

./waf

来编译 ns-3。现在如果你进入../../build/debug/ns3 目录的话你会发现本节开头提到的四个头文件。你可以仔细看一下这些文件的内容，你会发现它们包含了在相关模块中的所有的 include 文件。

### 4.2.3 Ns3 Namespace

在 first.cc 脚本的下一行是 namespace 的声明。

using namespace ns3;

Ns-3 工程是在一个叫做 ns3 的 C++ 命名空间中实现的。这把所有与 ns3 相关的声明，集中在一个与全局命名空间相区别的命名空间中。我们希望这样会给 ns3 与其他代码的集成带来好处。C++用“使用 ”语句（using）来把 ns-3 namespace 引入到当前的（全局的）声明域中。这个声明就是说，你不用为了使用 ns-3 的代码而必须在所有的 ns-3 代码前打上 ns3:: 作用域操作符。如果你对命名空间并不熟悉，请查阅任何的 C++手册并比较 ns3 命名空间和标准”std”命名空间的使用；通常你可以在 cout 和 streams 的讨论中看到命名空间的相关部分。

### 4.2.4 日志

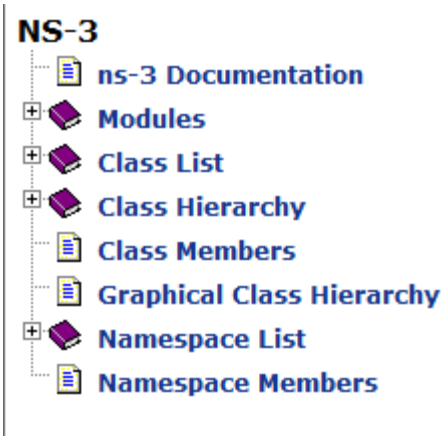
下一句脚本如下：

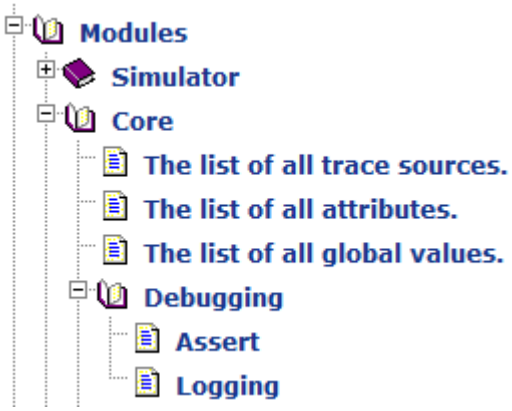
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");

我们认为本节是谈论 Doxygen 文件系统的好地方。如果你查看 ns-3 项目的网站（ns-3 project），你会在导航栏中发现一个指向“Doxygen (ns-3-dev)” 的链接。打开链接，你会被带到当前开发版文档页面。同时还有一个“Doxygen (stable)”得链接，它是最新的 ns-3 稳定发行版的文档页面。

在左边，你能找到一个文档结构的图形化目录。在 ns-3 的导航树中 NS-3 Modules 这本 “书”是一个开始的好地方。如果你展开“Modules”目录会看到

ns-3 模块文件的列表。这里的“模块”一级中的内容和上面讨论的模块 include 文件中的内容一一对应。Ns-3 的日志子系统是核心模块（core module）的一部分，所以继续前进打开核心（Core）这个文件节点。接着展开调试（“Debugging”）节点，选择日志（Logging）页面。




你现在应该看一下日志模块的 Doxygen 文件。在页面顶上的#defines 列表中你会看到 NS_LOG_COMPONENT_DEFINE 的条目。在你想要要深入了解之前，最好仔细看看日志模块的详细说明，这样你能够了解一下总体的流程。你可以继续下拉或者选择在 collaboration diagram 下的“更多”链接来达到目的。

一旦对日志模块有了整体的概念，那么请仔细看针对 NS_LOG_COMPO-NENT_DEFINE 的文档。我不想把整个文件复制到这里，但是可以总结一下大致的意思，这一行声明了一个叫 FirstScriptExample 的日志构件，通过引用 FirstScriptExample 这个名字的操作，可以实现打开或者关闭控制台日志的输出。

### 4.2.5 Main Function

下面的脚本是：

int

```
main (int argc, char *argv[])

{
```

这就是你的脚本程序的主函数的声明。正如任何其它 C++程序一样，你需要定义一个会被第一个执行的主函数。你的 ns-3 脚本没有什么特别的，就和一个普通的 C++程序一样。

下面两行脚本是用来使两个日志组件生效的。它们被内建在 Echo Client 和 Echo Server 应用中：

```
LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);

LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

如果你已经读过了日志组件文档，你会看到日志分成一系列详尽级别。这两行代码将回显 clients 和 server 的日志级别设为”INFO”级。这样，当仿真发生数据包发送和接受时，对应的应用就会输出相应的日志消息

关于日志模块，我们在后续内容中会详细介绍。现在我们直接着手创建拓扑和运行仿真。我们使用拓扑生成器对象来使这件事尽可能的容易。

# 4 概念概述（Conceptual Overview）

http://www.nsnam.org/docs/release/tutorial/tutorial_16.html#Conceptual-Overview

| | |
|---|---|
| 翻译： 陈杰 刘小洋 | piyajee@163.com (百思论坛) |
| 校稿： Xiaochuan Shen | xcs105@zepler.net |
| 编辑： ProbibidoAmor | banana.0420@yahoo.com.cn |

## 4.2.6 拓扑生成器

### 4.2.6.1 使用 NodeContainer 类

在我们的脚本中的下面两行将会创建 ns-3 节点对象，它们在仿真中代表计算机。

```
NodeContainer nodes;
nodes.Create (2);
```

在继续之前我们先找到 NodeContainer 类的文档。进入一个现成的类的说明文档可以通过在 Doxygen 页面的类标签中做到。如果你没有关闭上一节中打开的 Doxygen 页面的话，只要上拉到页面的顶部并选择类标签，就可以看见一个新的标签，类列表，出现。在标签下面你看看到所有的 ns-3 的类列表。往下翻，找到 ns3::NodeContainer。当你找到它后，点击它然后进入这个类的说明文档。

你可能回忆起节点概念是我们的一个关键抽象概念。节点代表一台能够加入诸如协议栈，应用以及外设卡等等的东西的计算机。NodeContainer 的拓扑生成器提供一种简便的方式来创建、管理和使用任何节点对象，我们用这些节点来运行模

拟器。上面的第一行只是声明了一个名为”nodes”的 NodeContainer。第二行调用了 nodes 对象的 Create()方法创建了两个节点。正如 Doxygen 所描述的那样，这个容器调用 NS-3 种的内部函数来产生两个节点对象，并把指向这两个对象的指针存储在系统之中。
在脚本中他们所代表的节点什么都没有做。构建拓扑的下一步是把我们的节点连接到一个网络中。我们所支持的最简单的网络形式是一个在两个节点之间单独的 point-to-point 连接。我们在此会构建一个此类连接。

#### 4.2.6.2 使用 PointToPointHelper 类

现 在我们来以一种你以后将会非常熟悉的方式来构建一个点到点的连接。我们使用拓扑生成器来完成创建，连接的底层工作。回忆一下我们的两个关键抽象概念：网络 设备、信道。在真实的世界中，这些东西大致相当于外设卡和网线。需要说明的是这两样东西紧密的联系在一起而不能够把它们交互地使用（比如以太网设备和无线 信道就不能一起使用）。拓扑生成器遵循了这种紧密的连接，因此你在这个脚本中仅需使用 PointToPointHelper 来配置和连接 ns-3 的 PointToPointNetDevice 和 PointToPointChannel 对象。
在脚本中下面的三句话是：

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

其中第一行，

```
PointToPointHelper pointToPoint;
```

在栈中初始化了一个 PointToPointHelper 的对象 PointToPoint。而紧接着的下一行，

```
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
```

从上层的角度告诉 PointToPointHelper 对象当创建一个 PointToPointNetDevice 对象时使用“5Mbps"来作为数据速率。

从细节方面讲，字符串“DataRate”与 PointToPointNetDevice 的一个属性相对应。如果你查看 Doxygen 中的 ns3::PointToPointNetDevice 类，并阅读 GetTypeId 方法的文档，你会发现设备定义了一系列属性，在这些属性中就有“DataRate”。大部分用户可见的 ns-3 对象都有类似的属性列表。正如你在下面的部分会看到的一样，我们使用了这个机制以方便地配置仿真器，而不用重新对源代码进行编译。与 PointToPointNetDevice 上的“DataRate”类似，PointToPointChannel 也有一个 Delay 属性：

```
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

告诉 PointToPointHelper 使用"2ms"（2 毫秒）作为每一个被创建的点到点信道传输延时值。

#### 4.2.6.3 使用 NetDeviceContainer 类

现在我们有一个包含两个节点的 NodeContainer 对象。我们有一个准备在两个节点之间创建 PointToPointNetDevices 和 wirePointToPointChannel 对象的

PointToPointHelper 对象。正如我们使用 NodeContainer 拓扑生成器对象来为我们的模拟创建节点，我们会让 PointToPointHelper 来做关于创建，配置和安装设备的工作。我们需要一个所有被创建的 NetDevice 对象列表，所以我们使用一个 NetDeviceContainer 对象来存放它们，就像我们使用一个 NodeContainer 对象来存放我们所创建节点。下面两行代码：

```
NetDeviceContainer devices;
devices = pointToPoint.Install (nodes);
```

会完成设备和信道的配置。第一行声明了上面提到的设备容器，第二行完成了主要工作。PointToPointHelper 的 Install()方法以一个 NodeContainer 对象作为一个参数。在 Install()方法内，一个 NetDeviceContainer 被创建了。对于在 NodeContainer 对象中的每一个节点（对于一个点到点链路必须明确有两个节点)，一个 PointToPointNetDevice 被创建和保存在设备容器内。一个 PointToPointChannel 对象被创建，两个 PointToPointNetDevices 与之连接。当 PointToPointHelper 对象创建时，那些在生成器中就被预先地设置的属性被用来初始化对象对应的属性值。

当调用了 pointToPoint.Install(nodes)后，我们会有两个节点，每一个节点安装了点到点网络设备，在它们之间是一个点到点信道。两个设备会被配置在一个有 2 毫秒传输延时的信道上以 5M 比特每秒的速率传输数据。

#### 4.2.6.4 使用 InternetStackHelper 类

我们现在已经配置了节点和设备，但是我们还没有在节点上安装任何协议栈。下面两行代码完成这个任务：

```
InternetStackHelper stack;
stack.Install (nodes);
```

类 InternetStackHelper 是一个安装 PointToPointHelper 对象和点到点网络设备的网络协议栈的拓扑生成器类。其中 Install()方法以一个 NodeContainer 对象做为一个参数，当它被执行后，它会为每一个节点容器中的节点安装一个网络协议栈（TCP,UDP,IP 等等）。

#### 4.2.6.5 使用 Ipv4AddressHelper 类

下面我们需要为节点上的设备设置 IP 地址。我们也提供了一个拓扑生成器来管理 IP 地址的分配。当执行实际的地址分配时唯一用户可见的 API 是设置基 IP 地址和子网掩码。

在我们的范例脚本文件 first.cc 的下两行代码

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
```

声明了一个地址生成器对象，并且告诉它应该开始从 10.1.1.0 开始以子网掩码为 255.255.255.0 分配地址。地址分配默认是从 1 开始并单调的增长，所以在这个基础上第一个分配的地址会是 10.1.1.1，紧跟着是 10.1.1.2 等等。底层 ns-3 系统事实上会记住所有分配的 IP 地址，如果你无意导致了相同 IP 地址的产生，这将是一个致命的错误（顺便说一下，这是个很难调试正确的错误）。

下面一行代码，

```
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

完成了真正的地址配置。在 ns-3 中我们使用 Ipv4Interface 对象将一个 IP 地址同一个设备关联起来。正如我们有时候需要一个被生成器创建的网络设备列表一样，我们有时候需要一个 Ipv4Interface 对象的列表。Ipv4InterfaceContainer 提供了这样的功能。

现在我们有了一个安装了协议栈，配置了 IP 地址类的点到点的网络。这时我们所要做的事情是运用它来产生数据通信。

### 4.2.7 Applications 类

另一个 ns-3 系统的核心抽象是 Application 类。在这个脚本中我们用两个特定的 ns-3 核心 Application 类：UdpEchoServerApplication 和 UdpEchoClientApplication。正如我们先前声明过的一样，我们使用生成器对象来帮助配置和管理潜在的对象。在这里，我们用 UdpEchoServerHelper 和 UdpEchoClientHelper 对象使我们的工作更加容易点。

#### 4.2.7.1 UdpEchoServerHelper 类

下面在 first.cc 脚本中的代码被用来在我们之前创建的节点上设置一个 UDP 回显服务应用。

```
UdpEchoServerHelper echoServer (9);
ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (10.0));
```

上面一片代码中的第一行声明了 UdpEchoServerHelper。 像往常一样，这个并非应用本身，这是一个用来帮助创建真正应用的对象。我们约定在生成器中放置必需的属性。本例中，除非我们告知生成器服务器和客户端所共 知的一个端口号，否则这个生成器是不会起任何作用的。我们并没有随机选择，而是把这个端口号作为生成器构造函数的一个参数。只要你愿意，你就能够使用 SetAttribute 设置“Port”参数到另一个值。

同其它生成器对象类似，UdpEchoServerHelper 对象有一个 Install 方法。实际上是这个方法的执行，才初始化回显服务器的应用，并将应用连接到一个节点上去。有趣的是，安装方法把 NodeContainter 当做一个参数，正如我们看到的其他安装方法一样。这里有一个 C++隐式转换，此转换以 nodes.Get(1)的结果作为输入，并把它作为一个未命名的 NodeContainer 的构造函数的参数，最终这个未命名的 NodeContainer 被送入 Install 方法中去。如果你曾迷失于在 C++代码中找到一个编译和运行良好的特定方法签名（signature），那么就寻找这些内在的转换。

我们现在会看到 echoServer.Install 将会在管理节点的 NodeContainer 容器索引号为 1 的机节点上安装一个 UdpEchoServerApplication。安装会返回一个容器，这个容器中包含了指向所有被生成器创建的应用指针。

应用对象需要一个时间参数来“开始”产生数据通信并且可能在一个可选的时间点“停止”。我们提供了开始和停止的两个参数。这些时间点是用 ApplicationContainer 的方法 Start 和 Stop 来设置的。这些方法以”Time”对象为参数。在这种情况下，我们使用了一种明确的 C++转换序列来获得 C++双精度

（double）的 1.0 并且用一个 Seconds 转换(cast)来把它转换到 ns-3 的 Time 对象。需要注意的是转换规则是模型的作者所控制，并且 C++也有它自己的标准，所以你不能总是假定参数会按照你的意愿顺利地转换。下面两行，

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

会使 echo 服务应用在 1s 时开始（生效）并在 10s 时停止（失效）。既然我们已经声明了一个模拟事件（就是应用的停止事件）在 10s 时被执行，模拟至少会持续 10s。

#### 4.2.7.2 UdpEchoClientHelper 类

echo 客户端应用的设置与回显服务器端类似。也有一个 UdpEchoClientHelper 来管理 UdpEchoClientApplication。

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));
ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

然而，对于 echo 客户端，我们需要设置五个不同的属性。首先两个属性是在 UdpEchoClientHelper 的构建过程中被设置的。按照生成器的构造函数的格式，我们把”RemoteAdress”和”RemotePort”属性传递给了生成器（实际上是作为生成器构造函数的两个必须参数传递的）。

回忆一下我们使用 Ipv4InterfaceContainer 来追踪我们配置给设备的 IP 地址。在界面容器中位置零的界面对象将会和在节点容器中的位置零的节点对象对应。同样在界面容器中位置一的界面对象将会和在节点容器中的位置一的节点对象对应。所以，在上面的第一行代码中，我们创建了一个生成器并告诉它设置客户端的远端地址为服务器节点的 IP 地址。我们同样告诉它准备发送第二个数据包到端口 9。

那个“MaxPackets”属性告诉客户端我们所允许它在模拟期间所能发送的最大数据包个数。“Interval”属性告诉客户端在两个数据包之间要等待多长时间，而“PacketSize”属性告诉客户端它的数据包应该承载多少数据。本例中，我们让客户端发送一个 1024 字节的数据包。

正如 echo 服务端一样，我们告诉 echo 客户端何时来开始和停止，但是这里我们使客户端在服务端生效 1s 后才开始（在模拟器中时间 2s 的时候）。

### 4.2.8 Simulator 类

下面我们所需要做的就是运行模拟器，这是用全局函数 Simulator::Run.来做到的

```
Simulator::Run ();
```

当我们调用了如下方法时：

```
serverApps.Start (Seconds (1.0));
```

```
  serverApps.Stop (Seconds (10.0));
  ...
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (10.0));
```

实际上我们是在模拟器中 1.0 秒，2.0 秒，和 10.0 时预设了时间的发生。当 Simulator::Run 被调用时，系统会开始遍历预设事件的列表并执行。首先它会在 1.0s 时运行事件，这个事件会使 echo 服务端应用生效（这个事件会预设更多的其他事件）。接下来仿真器会运行在 t=2.0 秒时的事件，即让 echo 客户端应用开始。同样的，这个事件可能会预定更多的其他事件。在 echo 客户端应用中的开始事件的执行会通过给服务端传送一个数据包来开始仿真的数据传送阶段。
发送一个数据包给服务端会引发一系列更多的事件。这些事件会被预设在此事件之后，并根据我们已经在脚本中设定的时间参数来执行数据包的应答。
其实，我们只发送了一个数据包（回忆一 MaxPackets 属性被设置为一），在此之后，那个被单独的客户端应答请求所引发的连锁反应会停止，并且模拟器会进入空闲状态。当这发生时，生下来的事件就是服务端和客户端的 Stop 事件。当这些事件被执行后，就没有将来的事件来执行了，函数 Simulator::Run 会返回。整个模拟过程就结束了。

下面剩下的事情就是清理了。这个通过调用全局函数 Simulator::Destroy 来完成。当生成器函数（或者低级的 ns-3 代码）被执行后，生成器安排的钩子函数就被插入到模拟器中来销毁所有被创建的对象。你自己并不需要追踪任何对象，你所需要做的仅仅是调用 Simulator::Destroy 并且退出。ns-3 系统会帮你料理这些繁杂的任务。在 first.cc 脚本中剩下的代码如下：

```
  Simulator::Destroy ();
  return 0;
}
```

### 4.2.9 创建自己的脚本

我们已经让构建你自己的脚本变得非常省事。你所需要做的仅仅是把你的脚本放到 scratch 目录下，并运行 waf,这样你的脚本就会被编译。在回到高层目录后复制 examples/tutorial/first.cc 到 scratch 目录下

```
cd ..
cp examples/tutorial/first.cc scratch/myfirst.cc
```

现在使用 waf 命令来创建自己的第一个实例脚本：

```
./waf
```

你应该可以看到消息报告说你的 myfirst 范例被成功编译了。

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
[614/708] cxx: scratch/myfirst.cc -> build/debug/scratch/myfirst_3.o
[706/708] cxx_link: build/debug/scratch/myfirst_3.o -> build/debug/scratch/myfirst
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
```

'build' finished successfully (2.357s)

现在你能够运行这个例子（注意如果你在 scratch 目录编译了你的程序，你必须在 scratch 目录外运行它）：

./waf --run scratch/myfirst

你应该能看到一些输出：

Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.418s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2

这里你看到编译系统核查来确定文件被编译了，接着运行了它。你看到在 echo 日志构件显示了它已经发送了 1024 字节到在 10.1.1.2 的 echo 服务端。还可以看到回显服务器端的日志构件显示他从 10.1.1.1 接收到了 1024 字节。接下来 echo 服务端应答了数据包，你能看到 echo 客户端记录了它已经接收到了从服务端发送过来的回显数据包。

# 5 Tweaking NS3 （NS3 调整？）

http://www.nsnam.org/docs/release/tutorial/tutorial_16.html#Conceptual-Overview

| | |
|---|---|
| 翻译： 杨飞 | 422467574@qq.com |
| 校稿：Xiaochuan Shen | xcs105@zepler.net |
| 编辑： ProbibidoAmor | banana.0420@yahoo.com.cn |

## 5.1 日志模块的使用

在运行 first.cc 脚本时，我们已经简单的了解了日志模块。现在，我们将更深入的了解日志子系统是为哪些用户案例设计的。

### 5.1.1 日志概述

很多大型系统支持某种的消息日志功能，ns3 也不例外。在某些情况下，只是错误消息日志被记录到操作控制台(在基于 Unix-系统中通常是标准错误输出)。在其他系统中，警告消息可能跟详细的信息消息一起被输出。某些情况下，日志功能被用来输出令人费解的调试信息。

在 NS-3 中，我们认为这些不同详尽级别的日志都是非常有用的。Ns3 还提供了一个可供选择的、多级别的方法来记录日志。日志可以完全被禁用，或仅对部分组件可用，或全局可用。并且 ns3 提供了不同详尽程度的日志级别供选。NS-3 日志模块提供了直观的、相对简单的使用方法来帮助用户获得仿真过程的所需信息。

应当了解我们也提供了一个一般性的记录机制，tracing，来获得仿真结果之外的数据(详情参见本教程的 tracing 系统的使用章节)。日志应当作为快速获得你的脚本和模型的调试信息、警告信息、错误信息、或是其他信息的首要选择。

在现有的系统中，有 7 个详尽程度递增的日志级别，他们分别是：

- NS_LOG_ERROR — Log error messages;
- NS_LOG_WARN — Log warning messages;
- NS_LOG_DEBUG — Log relatively rare, ad-hoc debugging messages;
- NS_LOG_INFO — Log informational messages about program progress;
- NS_LOG_FUNCTION — Log a message describing each function called;
- NS_LOG_LOGIC - Log messages describing logical flow within a function;
- NS_LOG_ALL — Log everything.

- NS_LOG_ERROR — 记录错误信息;
- NS_LOG_WARN — 记录警告信息;
- NS_LOG_DEBUG — 记录相对不常见的调试信息;
- NS_LOG_INFO — 记录程序进展信息;
- NS_LOG_FUNCTION — 记录描述每个调用函数信息;
- NS_LOG_LOGIC - 记录一个函数内描述逻辑流程的信息;
- NS_LOG_ALL — 记录所有信息.

我们也提供了一种一直被使用的无条件日志级别，它是跟日志详尽级别或是组件选择无关的。

- NS_LOG_UNCOND - 无条件记录相关消息.

每一个级别能够被单独地被调用或逐级递增的被调用。日志的配置可以使用一个 shell 环境变量(NS_LOG)，或是使用日志系统函数进行。正如在本教程之前部分看到的，日志系统有 Doxygen 文档，如果你还没有阅读日志模型文档现在是个好的时机。

既然你已经很详细地阅读了文档，我们使用日志来获得之前建立的 first.cc 脚本的一些有用信息。

### 5.1.2 启用日志

我们将使用 NS_LOG 环境变量来打开一些日志功能。首先，需要耐心点，像之那样运行脚本，

```
./war --run scratch/myfirst
```

你应当看到熟悉的、第一个 ns-3 例子程序的结果

```
Waf: Entering directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.413s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

上面看到的"Sent"和"Received"消息实际上是来自 UdpEchoClientApplication 和 UdpEchoServerApplication 的日志消息。我们通过 NS_LOG 环境变量设置日志级别让客户端程序输出更多信息。

假设你在使用一个类 sh 的 shell。此类 shell 使用“VARIABLE=value”的语法格式设置环境变量。如果使用类 csh 的 shell，必须将例句改成"setenv VARIABLE value"语法格式的语句。

现在，scratch/myfirst.cc 中 UDP 回显客户端应用在使用下面的代码行进行响应，

```
LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
```

这行代码设置日志为 LOG_LEVEL_INFO 级别。当我们传递一个日志级别标志时，我们实际上打开了这个日志级别和它之下的所有级别。本例中，我们打开了 NS_LOG_INFO, NS_LOG_DEBUG, NS_LOG_WARN 和 NS_LOG_ERROR 级别。我们可以通过设置 NS_LOG 环境变量在不改变脚本或重新编译的情况下来增加日志级别，获得更多信息，

```
export NS_LOG=UdpEchoClientApplication=level_all
```

这个设置 shell 环境变量 NS_LOG 为字符串：

```
UdpEchoClientApplication=level_all
```

等号左边是我们想要设置日志级别的组件的名字，等号右边是我们想要使用的日志级别。本例中，我们要为应用打开所有的调试信息级别。我们把 NS_LOG 设为这个样子，NS-3 日志系统将识别出日志级别改变，输出下面的结果：

```
Waf: Entering directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build
Waf: Leaving directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.404s)
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
UdpEchoClientApplication:HandleRead(0x6241e0, 0x624a20)
Received 1024 bytes from 10.1.1.2
```

```
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
```

这些额外的调试信息是来自 NS_LOG_FUNTION 级别的日志。这些信息显示了在脚本运行期间程序中每个函数调用过程。注意，ns-3 中模型对日志的支持并不是必须的。有关被记录信息的多少是由模型的开发者决定的。在本例中，有很多日志输出

你可以看到回显客户端程序中调用函数的日志。如果仔细看，会注意到字符串 UdpEchoClientApplication 和方法名之间是单冒号，而不是你预期的 C++域操作符(::)（双冒号）。这里是有意这样做的。

名字 UdpEchoClientApplication 并非一个类名，而是日志组件名。当一个类仅由一个源文件代表时，这个位置的显示通常是这个类的名字。这里用一个冒号来替代两个冒号，来提醒用户区分日志组件名和类名的细微差别：这个位置显示的是组件名，而并不是类名。

在某些情况下，确定哪个方法生成了某条日志消息是很困难的。如果你看过上面的文档，你或许想知道字符串“Received 1024 bytes from 10.1.1.2”来自哪里。我们可以通过在 NS_LOG 环境变量中设置 prefix_func 级别来解决。试着下面的语句做，

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func'
```

注意，这里引号是必须的，因为我们用的竖线表示或操作，而在 Unix 中竖线表示管道连接。

现在如果运行脚本你将看到每条日志都有产生此条日志的组件名做前缀了。

```
Waf: Entering directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.417s)
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
UdpEchoClientApplication:HandleRead(0x6241e0, 0x624a20)
UdpEchoClientApplication:HandleRead(): Received 1024 bytes from
10.1.1.2
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
```

你现在看到来自UDP回显客户端程序的消息这样被识别了。消息“Received 1024 bytes from 10.1.1.2”明显来自回显客户端程序。剩下的消息一定是来自UDP回显服务器程序。我们可以通过在NS_LOG环境变量中键入一个单冒号隔开的组件列表来启用回显服务器应用组件。

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func:
                UdpEchoServerApplication=level_all|prefix_func'
```

警告：你必须删除单冒号后的换行符，在例子文本中仅仅只是为了编排格式。

现在，如果你运行脚本，你将看到来自回显客户端和服务器的所有日志消息。这对于调试问题很有用。

```
Waf: Entering directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.406s)
UdpEchoServerApplication:UdpEchoServer()
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoServerApplication:StartApplication()
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
UdpEchoServerApplication:HandleRead(): Received 1024 bytes from
10.1.1.1
UdpEchoServerApplication:HandleRead(): Echoing packet
UdpEchoClientApplication:HandleRead(0x624920, 0x625160)
UdpEchoClientApplication:HandleRead(): Received 1024 bytes from
10.1.1.2
UdpEchoServerApplication:StopApplication()
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoServerApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
UdpEchoServerApplication:~UdpEchoServer()
```

有时能够看到日志生成的仿真时间也是很有用的。可以通过使用prefix_time位来实现。

```
export
'NS_LOG=UdpEchoClientApplication=level_all|prefix_func|prefix_time:
                UdpEchoServerApplication=level_all|prefix_func|p
refix_time'
```

输入时你必须先移除上面的换行符。如果你现在运行此脚本，你将看到下面的结果：

```
      Waf: Entering directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
      Waf: Leaving directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
      'build' finished successfully (0.418s)
      0s UdpEchoServerApplication:UdpEchoServer()
      0s UdpEchoClientApplication:UdpEchoClient()
      0s UdpEchoClientApplication:SetDataSize(1024)
      1s UdpEchoServerApplication:StartApplication()
      2s UdpEchoClientApplication:StartApplication()
      2s UdpEchoClientApplication:ScheduleTransmit()
      2s UdpEchoClientApplication:Send()
      2s UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
      2.00369s UdpEchoServerApplication:HandleRead(): Received 1024
bytes from 10.1.1.1
      2.00369s UdpEchoServerApplication:HandleRead(): Echoing packet
      2.00737s UdpEchoClientApplication:HandleRead(0x624290, 0x624ad0)
      2.00737s UdpEchoClientApplication:HandleRead(): Received 1024
bytes from 10.1.1.2
      10s UdpEchoServerApplication:StopApplication()
      10s UdpEchoClientApplication:StopApplication()
      UdpEchoClientApplication:DoDispose()
      UdpEchoServerApplication:DoDispose()
      UdpEchoClientApplication:~UdpEchoClient()
      UdpEchoServerApplication:~UdpEchoServer()
```

可以看到 UdpEchoServer 的构造函数在仿真的第 0 秒被调用。事实上这实在仿真开始之前就完成了，但时间显示的是 0 秒。UdpEchoClient 的构造函数也是一样。

回忆在 scratch/first.cc 脚本中，1 秒时启动回显服务器应用。可以看到服务器的 StartApplication 实际上是在 1 秒时被调用。同样，客户端响应程序正如我们所预料的在仿真 2 秒时开始。

我们现在可以看到仿真的进度了，我们可以看到从 ScheduleTransmit 函数在客户端中调用 send 函数，到回显服务器中调用 HandleRead 函数的整个过程了。注意到通过点到点连接发送包消耗时间是 3.69 毫秒。查看回显服务器日志记录了一条消息告诉你已经响应了数据包。在另一个通道延迟后，可以看到响应客户端用它的 HandleRead 方法收到响应包。

在仿真过程中发生了很多你所没有看到的事情。现在可以很容易的打开系统的日志组件，察看整个过程了。现在试着设置 NS_LOG 变量为，

```
      export 'NS_LOG=*=level_all|prefix_func|prefix_time'
```

上面的星号是日志组件通配符。将打开在仿真过程中使用的所有组件的日志功能。这里不列出结果了。可以将这些信息重定向到一个文件，并且用自己喜欢的编辑器打开查看。

```
      ./waf --run scratch/myfirst > log.out 2>&1.
```

当 我碰到一个问题或是不知道那里出错了，我使用最详细的日志功能。可以很简单的跟踪程序，而无需设置断点并且在调试器中一步步运行代码。我可以用我喜欢的编 辑器来打开查看日至，寻找问题所在。当我对错误有了大致了解之后，我使用调试器对问题有个非常详细的检查。当你脚本做了完全非预期的事，这种输出将是非常 有用的。如果你使用调试器单步运行，或许你会错过偏差的部分。日志使得这些偏差非常明显。

### 5.1.3 为你的代码增加日志功能

可以通过几个宏调用日志组件给仿真增加新的日志功能。我们可以在 scratch 目录中的 myfirst.cc 中做。

也许你还记得在脚本中我们已经定义过一个日志组件：

```
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");
```

我们已经了解了通过设置 NS_LOG 环境变量来给组件启用日志功能。我们可以给脚本增加一些日志功能。用来增加信息级别的日志消息的宏是 NS_LOG_INFO。现在我们来增加一行显示本脚本在创建拓扑的语句。此操作是在以下代码段完成的。

Open scratch/myfirst.cc in your favorite editor and add the line,

用你钟爱的编辑器打开 scratch/myfirst.cc 文件并且在 NodeContainer nodes; nodes.Create(2);之前加上一行，

```
NS_LOG_INFO ("Creating Topology");
```

现在用 waf 编译脚本并且清除 NS_LOG 环境变量来关掉我们之前启用的日志文件：

```
./waf
export NS_LOG=
```

现在，运行脚本，

```
./waf --run scratch/myfirst
```

你将不会看到新的日志消息，因为与它相关的日志组件 (FirstScriptExample)没有被启用。为了看到你的消息，必须使用大于或等于 NS_LOG_INFO 的日志级别来启用 FirstScriptExample 日志组件。如果只是想要看某个级别的日志，你可以通过下面的语句来启用它，

```
export NS_LOG=FirstScriptExample=info
```

如果现在运行脚本你将看到新建的"Creating Topology"日志消息，

```
Waf: Entering directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory
`/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.404s)
Creating Topology
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

下面就简单的说一下 ns3 中网络仿真的过程，

创建节点

创建链路类型

为节点创建具有链路类型的设备

为节点装载协议栈

设置节点和网络的 IP

配置业务应用

开始仿真

这个是一简单的仿真过程，其中还需要涉及到很多别的东西，因此需要更细节的考虑。

另外可以如下来考虑 ns3 的仿真过程，

CreateNodes ();

InstallInternetStack ();

InstallApplication ();

这三个步骤中，CreateNodes()包含了创建节点所需的 netDevice、phy、mac、channel 之类；

InstallInternetStack()包含了对其 L3 和 L4 层协议的加载以及网络 IP 的设置；

InstallApplication()是对节点业务的分配过程。

## ns3 里面的 log

1 log 的使用举例（以 tcp-large-transfer 为例子）

（1）执行，设置环境变量 NS_LOG

```
export NS_LOG=TcpLargeTransfer=level_all
```

（2）运行

```
./waf --run tcp-large-transfer
```

2 实现分析

（1）在 tcp-large-transfer.cc 中有

```
NS_LOG_COMPONENT_DEFINE ("TcpLargeTransfer");
```

宏的定义为

```
#define NS_LOG_COMPONENT_DEFINE(name)                          \
  static ns3::LogComponent g_log = ns3::LogComponent (name)
```

LogComponent 的构造函数为，

```
LogComponent::LogComponent (char const * name)
  : m_levels (0), m_name (name)
{
  EnvVarCheck (name); //这里会查询 NS_LOG 环境变量

  ComponentList *components = GetComponentList ();
  for (ComponentListI i = components->begin ();
       i != components->end ();
       i++)
    {
      if (i->first == name)
        {
          NS_FATAL_ERROR ("Log component \""<<name<<"\" has already been registered once.");
        }
    }
  components->push_back (std::make_pair (name, this));
}
```

这里建立了一个 static 的全局变量 g_log

（2）当碰到打印宏时，如

```
NS_LOG_LOGIC("Starting flow at time " <<  Simulator::Now ().GetSeconds ());
```

其定义为

```
#define NS_LOG_LOGIC(msg) \
  NS_LOG(ns3::LOG_LOGIC, msg)
```

而 NS_LOG 宏的定义为：

```
#define NS_LOG(level, msg)                                  \
  do                                                        \
    {                                                       \
      if (g_log.IsEnabled (level))                          \                //如果环境变量有设置的话，那么这里就会有打印
        {                                                   \
          NS_LOG_APPEND_TIME_PREFIX;                        \
```

```
      NS_LOG_APPEND_CONTEXT;                                \
      NS_LOG_APPEND_FUNC_PREFIX;                            \
      std::clog << msg << std::endl;                        \
    }                                                       \
  }                                                         \
while (false)
```

由于 g_log 是 static 的所以，每个文件的 g_log 的会控制本文件内的打印语句。

总结:新建立一个*.cc 文件时，只有在文件开头加上 NS_LOG_COMPONENT_DEFINE （name），那么就可以很好的实现本文件的打印开关。可用的等级有以下实现可以看出（LogComponent::EnvVarCheck 函数）：

```
if (lev == "error")
  {
    level |= LOG_ERROR;
  }
else if (lev == "warn")
  {
    level |= LOG_WARN;
  }
else if (lev == "debug")
  {
    level |= LOG_DEBUG;
  }
else if (lev == "info")
  {
    level |= LOG_INFO;
  }
else if (lev == "function")
  {
    level |= LOG_FUNCTION;
  }
else if (lev == "logic")
  {
    level |= LOG_LOGIC;
  }
else if (lev == "all")
  {
    level |= LOG_ALL;
  }
else if (lev == "prefix_func")
  {
```

```
    level |= LOG_PREFIX_FUNC;
  }
else if (lev == "prefix_time")
  {
    level |= LOG_PREFIX_TIME;
  }
else if (lev == "level_error")
  {
    level |= LOG_LEVEL_ERROR;
  }
else if (lev == "level_warn")
  {
    level |= LOG_LEVEL_WARN;
  }
else if (lev == "level_debug")
  {
    level |= LOG_LEVEL_DEBUG;
  }
else if (lev == "level_info")
  {
    level |= LOG_LEVEL_INFO;
  }
else if (lev == "level_function")
  {
    level |= LOG_LEVEL_FUNCTION;
  }
else if (lev == "level_logic")
  {
    level |= LOG_LEVEL_LOGIC;
  }
else if (lev == "level_all")
  {
    level |= LOG_LEVEL_ALL;
  }
```

## ns3 里面的 TypeId

ns3 里面继承于 Object 的类都有一个

static TypeId GetTypeId (void);

例如在 Node 类中有：

```
TypeId
Node::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::Node")
    .SetParent<Object> ()
    .AddConstructor<Node> ()
    .AddAttribute ("DeviceList", "The list of devices associated to this Node.",
                   ObjectVectorValue (),
                   MakeObjectVectorAccessor (&Node::m_devices),
                   MakeObjectVectorChecker<NetDevice> ())
    .AddAttribute ("ApplicationList", "The list of applications associated to this Node.",
                   ObjectVectorValue (),
                   MakeObjectVectorAccessor (&Node::m_applications),
                   MakeObjectVectorChecker<Application> ())
    .AddAttribute ("Id", "The id (unique integer) of this Node.",
                   TypeId::ATTR_GET, // allow only getting it.
                   UintegerValue (0),
                   MakeUintegerAccessor (&Node::m_id),
                   MakeUintegerChecker<uint32_t> ())
    ;
  return tid;
}
```

（靠，上面的代码用 windows Live writer 粘贴会有问题，太怪了！）

在 GetTypeId（）函数里面可以添加类自己的属性、指定基类。AddConstructor 添加了一个 constructor，不知道干啥用。这里添加的东西都存放到 IidManager 的 std::vector<struct IidInformation> m_information;其中 IidInformation 的定义为：

```
struct IidInformation {
   std::string name;
   uint16_t parent;
   std::string groupName;
   bool hasConstructor;
   ns3::Callback<ns3::ObjectBase *> constructor;
   bool mustHideFromDocumentation;
   std::vector<struct AttributeInformation> attributes;
   std::vector<struct TraceSourceInformation> traceSources;
};
```

TypeId 的构造函数如下面代码所示，可见 IidManager 的对象是唯一的。

```
TypeId::TypeId (const char *name)
{
```

```
  uint16_t uid = Singleton<IidManager>::Get ()->AllocateUid (name);
  NS_ASSERT (uid != 0);
  m_tid = uid;
}
```

所以，所有继承自 Object 的类的信息（struct IidInformation）都存放到全局唯一的 IidManager 对象中。 TypeId 类提供了存储、获取数据（包括各种属性等）的接口，但是真正的数据却没有存在 TypeId 中，而是存放在全局唯一的 IidManager 对象中。这叫做抽象和实现相分离还是控制和数据相分离？类 Object 中有一个成员变量 TypeId m_tid，TypeId 以组合的方式存在于 Object 中。TypeId 没有虚函数，只有一个 uint16_t m_tid（用来表示在全局 IidManager 对象中的位置），整个类的大小为 16bit，可见设计时充分考虑了内存的占用情况。TypeId 对 IidManager 对象的访问采用 Singleton<IidManager>::Get ()的方式，在 TypeId 类中不需要保存类 IidManager 的信息。

由于基于 Object 的对象建立时需要调用 GetTypeId 和 设置并初始化属性，所以对象的建立最好使用 CreateObject 函数

```
template <typename T>
Ptr<T> CreateObject (void)
{
  return CompleteConstruct (new T ());
}
```

CompleteConstruct 为 Object 的友元函数。

```
template <typename T>
Ptr<T> CompleteConstruct (T *p)
{
  p->SetTypeId (T::GetTypeId ());
  p->Object::Construct (AttributeList());
  return Ptr<T> (p, false);
}
```

---

## ns3 里面的 callback

1 使用 Callback 的目的

ns3 里面的 callback 主要是为了解除模块之间的耦合。举个例子，如果类 B 想调用类 A 的函数，使用下面的代码的话，会使 A 和 B 的耦合性增加，若以后增加了一个与 A 功能相同的新的类 C，B 不再调用 A 的函数，而是调用 C 的函数，那么类 B 就需要修改，显然这不符合开放-闭合原则。

```
class A {
public:
```

```
  void ReceiveInput ( // parameters );
  ...
}

(in another source file:)

class B {
public:
  void DoSomething (void);
  ...

private:
  A* a_instance; // pointer to an A
}

void
B::DoSomething()
{
  // Tell a_instance that something happened
  a_instance->ReceiveInput ( // parameters);
  ...
}
```

如果将 B 的成员变量 A *a_instance 改成“函数指针”，运行时指定 B 成员变量“函数指针”的值，那么就可以有效的实现模块之间的耦合，并且具有较好的可扩展性。

2 ns3 里面 Callback 的使用

Callback 的例子 main-callback.cc 的内容如下：

```
#include "ns3/callback.h"
#include "ns3/assert.h"
#include <iostream>

using namespace ns3;

static double
CbOne (double a, double b)
{
  std::cout << "invoke cbOne a=" << a << ", b=" << b << std::endl;
  return a;
}
```

```
class MyCb {
public:
  int CbTwo (double a) {
    std::cout << "invoke cbTwo a=" << a << std::endl;
    return -5;
  }
};

int main (int argc, char *argv[])
{
  // return type: double
  // first arg type: double
  // second arg type: double
  Callback<double, double, double> one;
  // build callback instance which points to cbOne function
  one = MakeCallback (&CbOne);
  // this is not a null callback
  NS_ASSERT (!one.IsNull ());
  // invoke cbOne function through callback instance
  double retOne;
  retOne = one (10.0, 20.0);

  // return type: int
  // first arg type: double
  Callback<int, double> two;
  MyCb cb;
  // build callback instance which points to MyCb::cbTwo
  two = MakeCallback (&MyCb::CbTwo, &cb);
  // this is not a null callback
  NS_ASSERT (!two.IsNull ());
  // invoke MyCb::cbTwo through callback instance
  int retTwo;
  retTwo = two (10.0);

  two = MakeNullCallback<int, double> ();
  // invoking a null callback is just like
  // invoking a null function pointer:
  // it will crash.
  //int retTwoNull = two (20.0);
  NS_ASSERT (two.IsNull ());

  return 0;
}
```

Callback 具体的使用不复杂：

（1） 建立一个 Callback 对象（上面的 Callback<double, double, double> one）

（2） Callback 对象被赋值，一般使用 Make*Callback 函数 （上面的 one = MakeCallback (&CbOne);）

（3） Callback 执行，使用 Callback 的（）操作 （retOne = one (10.0, 20.0); ）

3 ns3 里面 Callback 的实现

ns3 里面 Callback 的设计如图 1 所示，按照 bridge 的思想对接口和实现实现了分类。

FunctorCallbackImpl 单个函数的 Callback

MemPtrCallbackImpl 类成员函数的 Callback

BoundFunctorCallbackImpl 这个类提供了一个额外的参数，使得 Callback 对象的操作（）可以少提供一个参数。

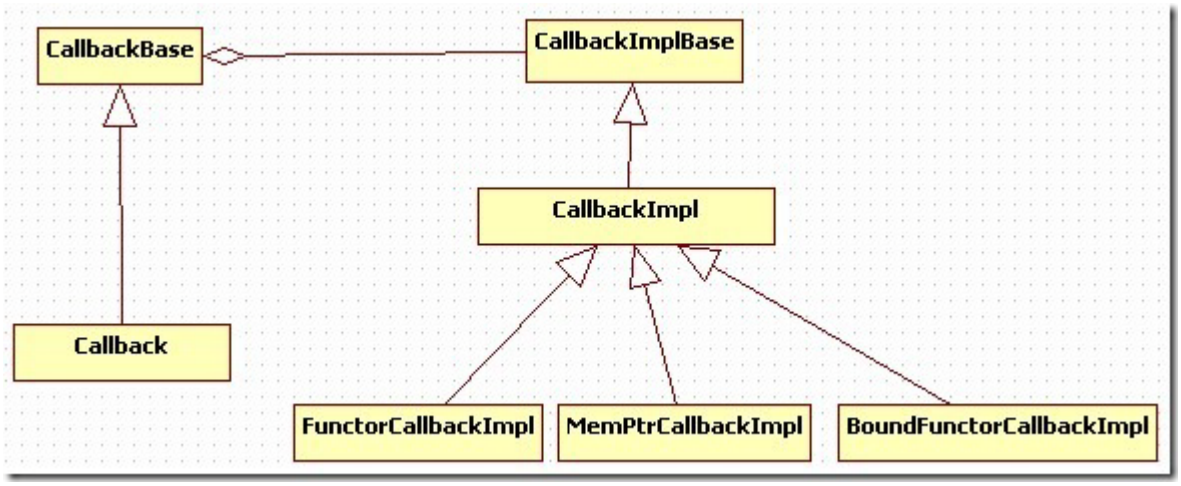



图 1 Callback 的类关系

类 Callback 并没有重载=，这样的话就是使用了编译器的默认的赋值函数，效率会不会有问题？

---

## 离散时间仿真 Discrete event simulation

http://en.wikipedia.org/wiki/Discrete_event_simulation

In **discrete-event simulation**, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system [1]. For example, if an elevator is simulated, an event could be "level 6 button pressed", with the resulting system state of "lift moving" and eventually (unless one chooses to simulate the failure of the lift) "lift at level 6".

在离散事件仿真中，对系统的操作由一系列按时间顺序的事件表示。每一个事件瞬间发生，并标记系统状态的变化。比如仿真一个“电梯”，事件“按下 6 楼的按钮”导致电梯开始升降，最终到 6 楼。

A common exercise in learning how to build discrete-event simulations is to model a queue, such as customers arriving at a bank to be served by a teller. In this example, the system entities are CUSTOMER-QUEUE and TELLERS. The system events are CUSTOMER-ARRIVAL and CUSTOMER-DEPARTURE. (The event of TELLER-BEGINS-SERVICE can be part of the logic of the arrival and departure events.) The system states, which are changed by these events, are NUMBER-OF-CUSTOMERS-IN-THE-QUEUE (an integer from 0 to n) and TELLER-STATUS (busy or idle). The random variables that need to be characterized to model this system stochastically are CUSTOMER-INTERARRIVAL-TIME and TELLER-SERVICE-TIME.

如何建立离散时间仿真器通常参照队列模型，比如顾客到达银行，要求出纳员服务。在这个例子中，系统的实体是“顾客队列”和“出纳员”，系统的事件包含：“顾客到达”和“顾客离去”（事件“出纳员员开始服务”可以看成到达和离去事件的一部分），系统的状态包含：“队列中顾客数”（自然数）和“出纳员的状态”（忙 或 闲），系统状态有系统事件改变。要模拟这个随机过程，需要把“顾客到达间隔”和“出纳员服务时间”描述成随机变量。

## Components of a Discrete-Event Simulation 离散时间仿真的组成

In addition to the representation of system state variables and the logic of what happens when system events occur, discrete event simulations include the following:

### Clock 定时器

The simulation must keep track of the current simulation time, in whatever measurement units are suitable for the system being modeled. In discrete-event simulations, as opposed to real time simulations, time ‘hops’ because events are instantaneous – the clock skips to the next event start time as the simulation proceeds.

与“实时仿真”不同，时间会跳跃到下一仿真要处理事件发生时间，因为事件是瞬时完成的。

### Events List 事件列表

The simulation maintains at least one list of simulation events. This is sometimes called the *pending event set* because it lists events that are pending as a result of previously simulated event but have yet to be simulated themselves. An event is described by the time at which it occurs and a type, indicating the code that will be used to simulate that event. It is common for the event code to be parameterised, in which case, the event description also contains parameters to the event code.

When events are instantaneous, activities that extend over time are modeled as sequences of events. Some simulation frameworks allow the time of an event to be specified as an interval, giving the start time and the end time of each event.

Single-threaded simulation engines based on instantaneous events have just one current event. In contrast, multi-threaded simulation engines and simulation engines supporting an interval-based event model may have multiple current events. In both cases, there are significant problems with synchronization between current events.

The pending event set is typically organized as a priority queue, sorted by event time.[2] That is, regardless of the order in which events are added to the event set, they are removed in strictly chronological order. Several general-purpose priority queue algorithms have proven effective for discrete-event simulation,[3] most notably, the splay tree. More recent alternatives includeskip lists and *calendar queues*.[4]

Typically, events are scheduled dynamically as the simulation proceeds. For example, in the bank example noted above, the event CUSTOMER-ARRIVAL at time t would, if the CUSTOMER_QUEUE was empty and TELLER was idle, include the creation of the subsequent event CUSTOMER-DEPARTURE to occur at time t+s, where s is a number generated from the SERVICE-TIME distribution.

Random-Number Generators 随机数生成器

The simulation needs to generate random variables of various kinds, depending on the system model. This is accomplished by one or more Pseudorandom number generators. The use of pseudorandom numbers as opposed to true random numbers is a benefit should a simulation need a rerun with exactly the same behaviour.

One of the problems with the random number distributions used in discrete-event simulation is that the steady-state distributions of event times may not be known in advance. As a result, the initial set of events placed into the pending event set will not have arrival times representative of the steady-state distribution. This problem is typically solved by bootstrapping the simulation model. Only a limited effort is made to assign realistic times to the initial set of pending events. These events, however, schedule additional events, and with time, the distribution of event times approaches its steady state. This is called *bootstrapping* the simulation model. In gathering statistics from the running model, it is important to either disregard events that occur before the steady state is reached or to run the simulation for long enough that the bootstrapping behavior is overwhelmed by steady-state behavior. (This use of the term *bootstrapping* can be contrasted with its use in both statistics and computing.)

Statistics 统计

The simulation typically keeps track of the system's statistics, which quantify the aspects of interest. In the bank example, it is of interest to track the mean service times.

Ending Condition 结束条件

Because events are bootstrapped, theoretically a discrete-event simulation could run forever. So the simulation designer must decide when the simulation will end. Typical choices are “at time t” or “after processing n number of events” or, more generally, “when statistical measure X reaches the value x”.

## Simulation Engine Logic 仿真引擎逻辑

The main loop of a discrete-event simulation is something like this:

Start 开始：初始化变量（结束条件（False），系统状态，时钟），开始调度初始的的事件

- Initialize Ending Condition to FALSE.

- Initialize system state variables.
- Initialize Clock (usually starts at simulation time zero).
- Schedule an initial event (i.e., put some initial event into the Events List).

“Do loop” or “While loop”循环

While (Ending Condition is FALSE) then do the following:

- Set clock to next event time. 将时间设置成要处理时间的时间
- Do next event and remove from the Events List. 将事件从队列里移除并处理
- Update statistics. 更新统计数据

End

- Generate statistical report.

---

## 例子 emu-udp-echo

这个例子的执行需要将网卡设为混杂模式

命令：

ifconfig eth0 promisc

执行例子的命令：

./waf –run "emu-udp-echo --deviceName=eth0"

执行完后，在 ns-3-dev 目录下生成相应的文件：

```
ren:~/ns3/ns-3-dev# ls -l | grep emu
-rw-r--r--  1 root root     2492 Apr 13 17:39 emu-udp-echo-0-0.pcap
-rw-r--r--  1 root root     2492 Apr 13 17:39 emu-udp-echo-1-0.pcap
-rw-r--r--  1 root root     2492 Apr 13 17:39 emu-udp-echo-2-0.pcap
-rw-r--r--  1 root root     2492 Apr 13 17:39 emu-udp-echo-3-0.pcap
-rw-r--r--  1 root root        0 Apr 13 17:39 emu-udp-echo.tr
ren:~/ns3/ns-3-dev# _
```

用 tcpdump 可以看数据包的内容

```
ren:~/ns3/ns-3-dev# tcpdump  -r emu-udp-echo-0-0.pcap -tt -nn
reading from file emu-udp-echo-0-0.pcap, link-type EN10MB (Ethernet)
2.000000 arp who-has 10.1.1.2 (ff:ff:ff:ff:ff:ff) tell 10.1.1.1
2.004601 arp reply 10.1.1.2 is-at 00:00:00:00:00:02
2.004601 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 1024
2.006557 arp who-has 10.1.1.1 (ff:ff:ff:ff:ff:ff) tell 10.1.1.2
2.006557 arp reply 10.1.1.1 is-at 00:00:00:00:00:01
2.007419 IP 10.1.1.2.9 > 10.1.1.1.49153: UDP, length 1024
ren:~/ns3/ns-3-dev#
```