第5章 自动驾驶汽车软件计算框架

5.1 概述

自动驾驶是一个极其复杂的系统性工程，当开发者在面对一个全新的车辆平台，或是需要向已开发的自动驾驶车辆软件系统中植入新的功能，又或是需要对单独某个模块进行迭代更新时，如果没有软件计算框架的支持，那么这些都将成为牵一发而动全身的动作，这直接限制了自动驾驶汽车的更新迭代以及快速发展。可见，一个优秀的自动驾驶汽车软件计算框架，将会扮演整个系统中“管家”的角色，对内它服务于开发人员，让开发更加便捷顺利，对外它提供更多的接口，让使用者用起来得心应手。基于以上描述，一个优秀的软件计算框架需要具备以下这 3 个特征。

(1) 高效的开发支持。

(2) 可灵活配置的模块。

(3) 丰富的调试工具。

本章着重介绍两种自动驾驶汽车软件计算框架，亦可以称为自动驾驶汽车的操作系统。第一种是面向机器人开发而诞生的 ROS(Robot Operating System，机器人操作系统)，本章介绍其在自动驾驶汽车开发中的应用。第二种是面向自动驾驶汽车开发的操作系统 Cyber RT。

5.2 机器人操作系统

5.2.1 ROS 概述

伴随机器人领域的快速发展和功能复杂化，对机器人控制代码的复用性和模块化的需求越来越强烈，而已有的开源机器人系统并不能很好地适应需求。2010 年，Willow Garage 公司发布了开源机器人操作系统 ROS，很快就在机器人研究领域推动起学习和使用 ROS 的热潮。

ROS 系统起源于 2007 年斯坦福大学人工智能实验室与机器人技术

公司 Willow Garage 的个人机器人项目(Personal Robots Program)之间的合作,2008 年之后由 Willowb Garage 公司全面接管并进行迭代开发和维护。

ROS 是开源的,是在机器人系统之上的一种后操作系统,或者说次级操作系统。它提供类似操作系统所提供的功能,包含硬件抽象描述、底层驱动程序管理、共用功能的执行、程序间的消息传递、程序发行包管理等;同时,它也提供一些工具和程序库用于获取、建立、编写和运行多机整合的应用。

5.2.2 ROS 特点

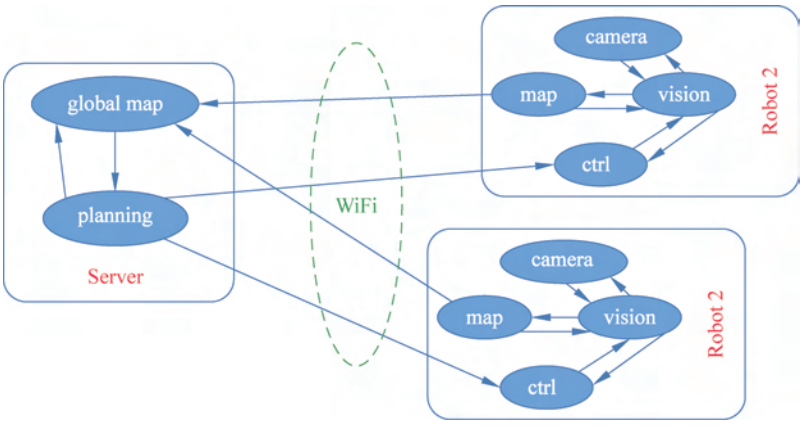
ROS 的运行架构是一种使用 ROS 通信模块实现模块间 P2P 的松耦合的网络连接的处理架构,支持若干种类型的通信,包括基于服务的同步 RPC(远程过程调用)通信、基于 Topic 的异步数据流通信,还包括参数服务器上的数据存储,但是 ROS 本身不具备实时性。

ROS 的主要特点可以归纳为以下几条。

1) 点对点设计

一个使用 ROS 的系统包括一系列进程,这些进程存在于多个不同的主机并且在运行过程中通过端对端的拓扑结构进行联系。虽然一些基于中心服务器的软件框架也具备多进程和多主机的优势,但是在这些框架中,当各主机通过不同的网络进行连接时,中心服务器就会发生问题。

图 5.1 所示为 ROS 的点对点设计示例。



■图 5.1 ROS 的点对点设计示例

ROS 的点对点设计以及服务和节点管理器等机制可以分散由计算机视觉和语音识别等功能带来的实时计算压力,适应大多数机器人遇到的计算挑战。

2) 多语言支持

在写代码的时候,大多数编程者会偏好某一些编程语言。这些偏好是个人在每种语言所花的编程时间、达到的调试效果、对编程语言语法的适应、可接受的程序执行效率以及各种技术和文化的原因导致的结果。为了解决这些问题,ROS 被设计成语言中立性的框架结构。ROS 支持多种主流编程语言,如图 5.2 所示,例如 C++、Python、Java、Octave 和

Lisp，也支持其他多种编程语言的接口实现。

■图 5.2　多种编程语言支持 ROS 开发

ROS 的特殊性主要体现在消息通信层，其利用 XML-RPC 机制实现端对端的连接和配置。XML-RPC 也实现了大多数主流编程语言的合规描述。ROS 的开发者希望它能够适配各种编程的语法约定，而不是仅仅基于 C 语言去给各种其他编程语言提供实现接口。然而，在某些情况下，可以利用已经存在的库，封装后支持更多新的编程语言。例如，Octave 的客户端就是通过 C++的封装库进行实现的。

为了支持交叉语言，ROS 利用了简单的、语言无关的接口定义语言去描述模块之间的消息传送。接口定义语言使用了简短的文本去描述每条消息的结构，也允许消息的合成。

每种编程语言的代码产生器都会产生类似本编程语言的目标文件，在消息传递和接收的过程中通过 ROS 实现自动连续并行运行。该特性节省大量的编程时间，也避免出现错误：之前 3 行代码的接口定义文件可自动扩展成 137 行 C++代码、96 行 Python 代码、81 行 Lisp 代码或 99 行 Octave 代码。因为消息是从各种简单的文本文件中自动生成的，所以很容易列举出新的消息类型。在编写 ROS 应用过程中，可利用基于 ROS 代码库中包含的超过 400 种消息类型，这些消息适配传感器传送数据使用，使 ROS 系统可轻易获得周围环境信息。

其好处是，ROS 的消息处理系统完全与编程语言无关，可支持多种编程语言自由结合与适配使用。

3）精简与集成

已知的大部分开发完成的机器人软件工程都包含可以在工程外重复使用的驱动和算法，但不幸的是，由于多方面的原因，大部分代码的中间层都过于混乱，以至于很难单独提取出相关功能，并把这些功能和驱动应用到其他程序或工程中。

为了应对这种挑战，ROS 中的所有驱动和算法逐渐发展成为与 ROS 没有依赖性的、单独的库。ROS 建立的系统具有模块化的特点，各模块中的代码可以单独编译，而且编译使用的 CMake 工具使其自始至终贯彻精简的理念。ROS 系统将复杂的代码实现封装在各个库中，并创建了一些短小精干的应用程序以显示 ROS 库的功能。这种方式允许对 ROS 的代码进行简单移植并复用于任何新系统中。另一个巨大优势在于，对代码的单元测试也变得较为容易，一个独立的单元测试程序可以测试代码库中很多的特性。

ROS 复用了很多流行的开源项目的代码，例如，从 Player 项目中复用了驱动、运动控制

和仿真方面的代码，从 OpenCV 中借鉴了视觉算法方面的代码，从 OpenRAVE 引用了规划算法的内容。这种例子不胜枚举。在每一个创建的实例中，ROS 都被用来显示多种多样的配置选项，并在各软件之间进行和管理数据通信，同时对它们进行微小的包装和改动。ROS 有一个活跃的社区，大量开发者在社区中对其进行维护和升级，包括升级其软件库、对应用打补丁等，从而不断升级 ROS 的源代码。

4）工具包丰富

为了管理复杂的 ROS 软件框架，开发者利用大量的小工具去编译和运行多种多样的 ROS 组件，以维持一个精简的内核，避免去构建一个庞大的开发和运行环境。

图 5.3 所示为 ROS 丰富的工具包。

这些小工具主要担负的任务有，组织源代码的结构、获取和设置配置参数、图形化端对端的拓扑连接、测量频带使用宽度、即时描绘信息数据、自动生成文档等。ROS 开发者的目标是把所有的代码模块化，因为他们相信，损失效率的重要性远远低于系统的稳定性和管理的复杂性。

5）免费并且开源

ROS 所有的源代码都是公开发布的，这也是当今 ROS 系统在机器人和自动驾驶领域广泛应用的主要原因。并且，活跃的开发者们会在软件各层次进行调试，并不断改正错误。ROS 的开源遵循 BSD 许可，也就是说允许各种商业和非商业的工程基于 ROS 系统进行开发。

ROS 系统通过内置的通信系统进行数据传递，不强制要求所有模块在相同可执行层面上相互连接。因此，利用 ROS 构建的系统可以较为自由地使用大量其他组件——个别模块甚至可以包含被各种协议保护的软件，这些协议包含从 GPL 到 BSD。

5.2.3 ROS 总体框架

根据 ROS 系统代码的维护者和分布者来标示，ROS 主要有两大部分。

（1）main：核心部分，主要由 Willow Garage 公司和一些开发者设计、提供以及维护。它提供了一些分布式计算的基本工具，以及整个 ROS 核心部分的程序。

（2）universe：全球范围的代码，由不同国家的 ROS 社区组织开发和维护。一种是库的代码，如 OpenCV、PCL 等；库的上一层是从功能角度提供的代码，如人脸识别，这些功能会调用下层的库；最上层的代码是应用级的代码，控制机器人完成某一确定的功能。

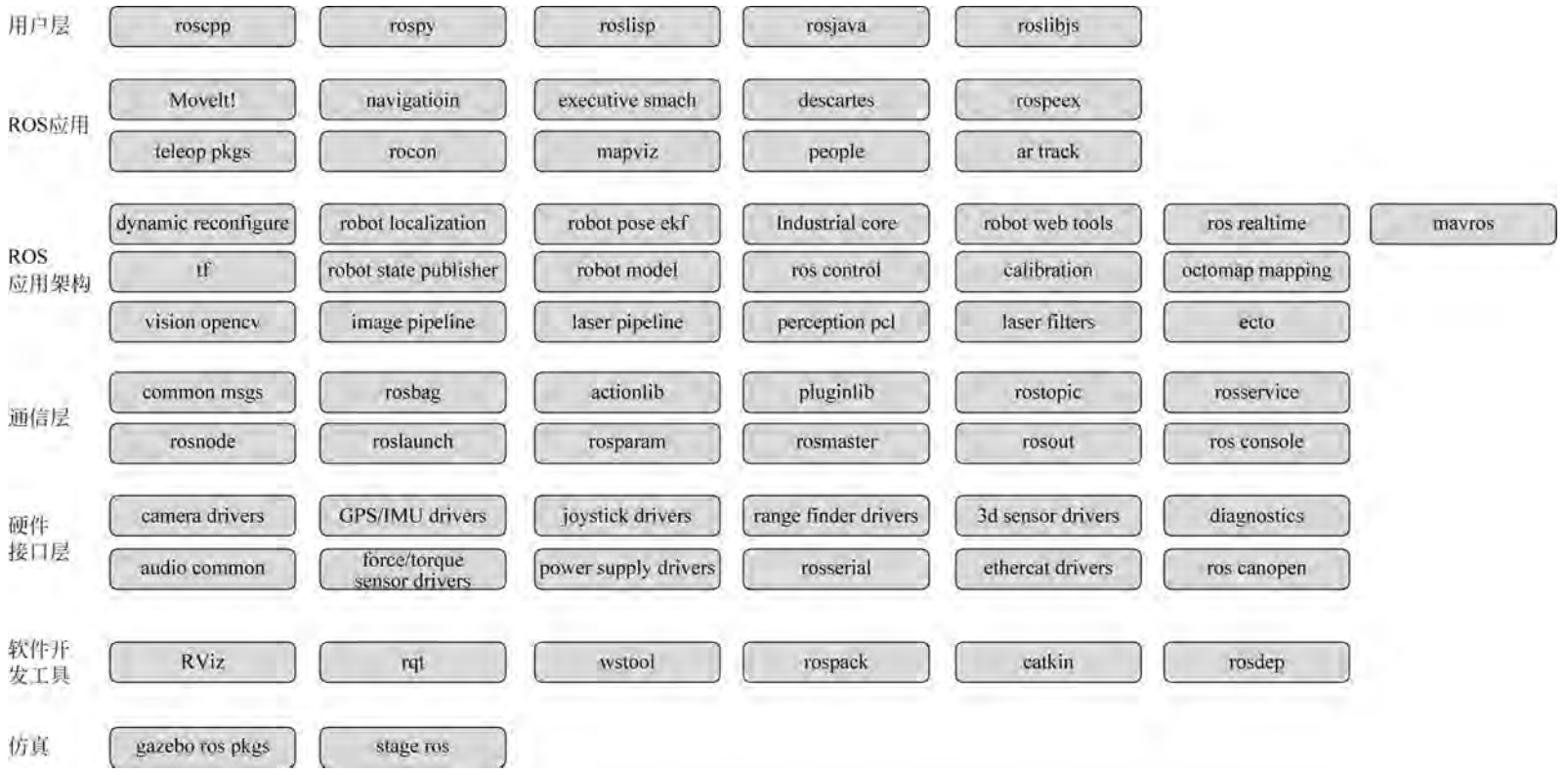
如果从另一个角度对 ROS 分级，主要可分为 3 个级别：计算图级、文件系统级、社区级，如图 5.4 所示。下面的介绍主要使用这种分级制度。

1. 计算图级

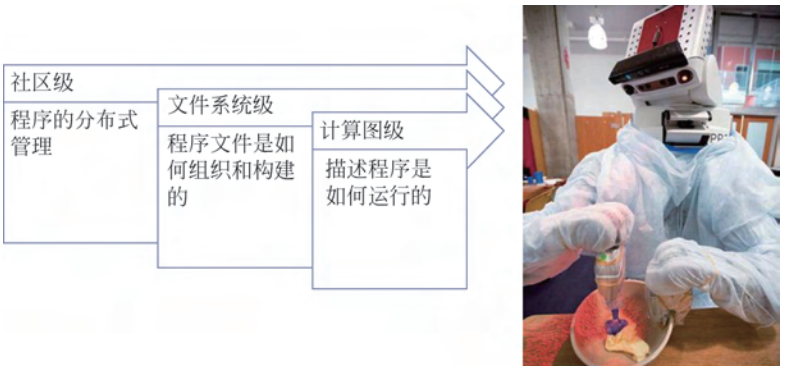
计算图是 ROS 处理数据的一种点对点的网络形式。程序运行时，所有进程以及它们所进行的数据处理，都将会通过一种点对点的网络形式表现出来。这一级主要包括几个重要概念：节点（node）、消息（message）、主题（topic）、服务（service），如图 5.5 所示。

1）节点

节点是一些执行运算任务的进程。ROS 利用规模可增长的方式使代码模块化：一个典型系统由很多节点组成。在这里，节点也可以被称为软件模块。节点的称呼使得基于 ROS



■图 5.3　ROS 丰富的工具包



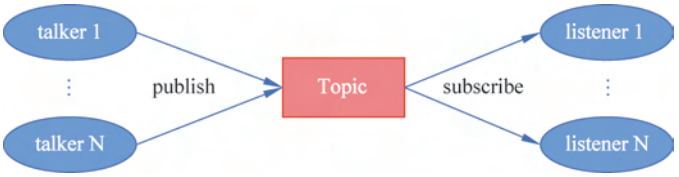
■图 5.4 ROS 系统分级

的系统在运行时更加形象化：当许多节点同时运行时，可以方便地将端对端的通信绘制成一个图表，在这个图表中，进程就是图中的节点，而端对端的连接关系由其中的弧线连接表现。

2）消息

节点之间通过传送消息进行通信，如图 5.6 所示。每一个消息都是一个严格的数据结构。原有标准的数据类型（如整型、浮点型、布尔型等）都是支持的，同时也支持原始数组类型。消息可以包含任意的嵌套结构和数组（类似于 C 语言的结构 structs）。

■图 5.5 ROS 的计算图级



■图 5.6 ROS 的消息

3）主题

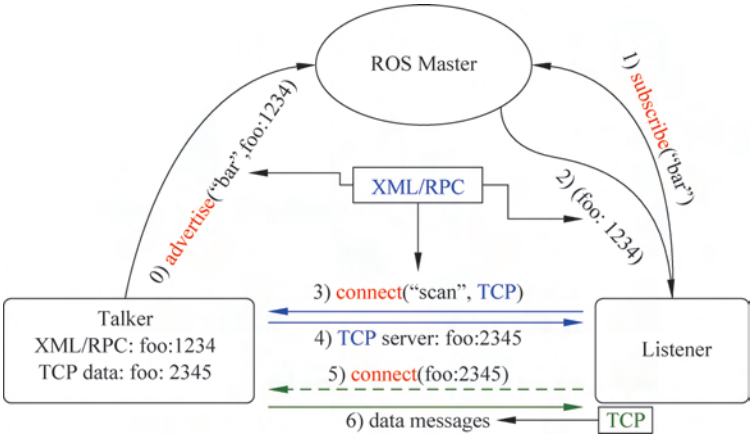
消息以一种发布或订阅的方式传递。一个节点可以在一个给定的主题中发布消息。一个节点针对某个主题关注与订阅特定类型的数据。可能同时有多个节点发布或者订阅同一个主题的消息。总体上，发布者和订阅者不了解彼此的存在。

4）服务

虽然基于话题的发布或订阅模型是很灵活的通信模式，但是它广播式的路径规划对于可以简化节点设计的同步传输模式并不适合。在 ROS 中，一项服务用一个字符串和一对严格规范的消息定义：一个用于请求，一个用于回应。这类似于 Web 服务器，Web 服务器是由 URI 定义的，同时带有完整定义类型的请求和回复文档。需要注意的是，不像话题，只有一个节点可以以任意独有的名字广播一项服务，只有一项服务可以称为分类象征，例如，任意一个给出的 URI 地址只能有一个 Web 服务器。

在上面概念的基础上，需要有一个控制器可以使所有节点有条不紊地执行，这个控制器被称为 ROS 控制器（ROS Master）。

ROS Master 通过 RPC(Remote Procedure Call Protocol,远程过程调用)提供登记列表和对其他计算图表的查找。没有控制器,一个节点将无法找到其他节点,并交换消息或调用服务。例如,控制节点订阅和发布消息的模型如图 5.7 所示。



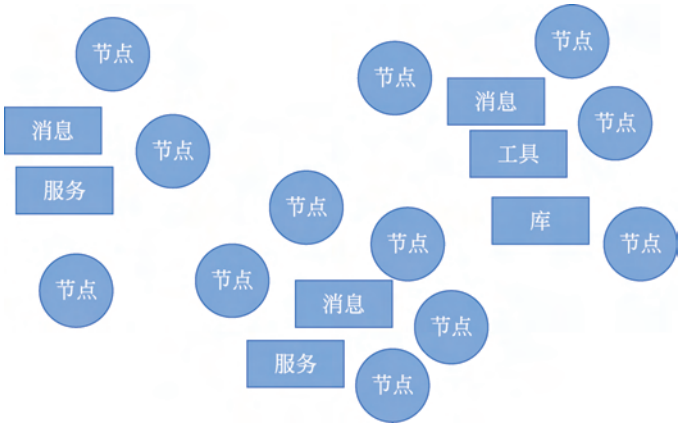
■图 5.7 控制节点订阅和发布消息的模型

ROS 的控制器给 ROS 的节点存储了主题和服务的注册信息。节点与控制器通信报告它们的注册信息。当这些节点与控制器通信的时候,它们可以接收关于其他已注册节点的信息,并且建立与其他已注册节点之间的联系。当这些注册信息改变时控制器也会回馈节点,同时允许节点动态创建与新节点之间的连接。

节点与节点之间的连接是直接的,控制器仅仅提供了查询信息,就像一个 DNS 服务器。节点订阅一个主题会要求建立一个与发布该主题的节点的连接,并且将会在同意连接协议的基础上建立该连接。

2. 文件系统级

ROS 文件系统级主要是指,在硬盘上面查看的 ROS 源代码的组织形式,如图 5.8 所示。



■图 5.8 ROS 系统级构成

ROS 系统中有无数的节点、消息、服务、工具和库文件，需要有效的结构去管理这些代码。在 ROS 的文件系统级，有以下两个重要概念：包（package）、堆（stack）。

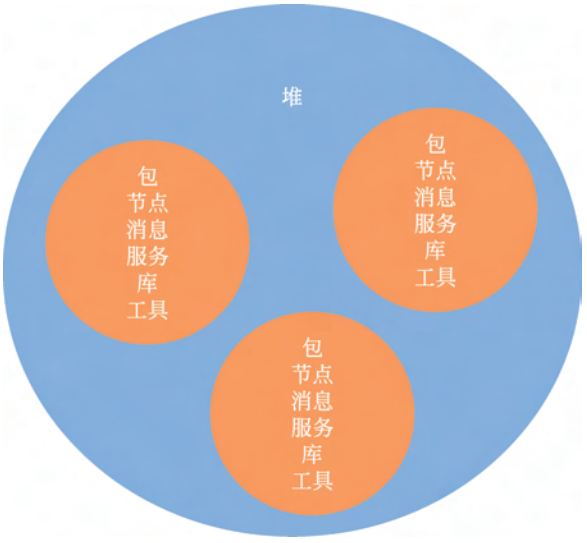
1）包

ROS 的软件以包的形式组织。每个包里包含节点、ROS 依赖库、数据套、配置文件、第三方软件或者任何其他逻辑。如图 5.9 所示，包的目标是提供一种易于使用的结构以便其软件的重复使用。总的来说，ROS 的包短小精干。

■图 5.9　ROS 的包

2）堆

堆是包的集合，它提供一个完整的功能，像“navigationstack”，如图 5.10 所示。堆与版本号关联，同时也是如何发行 ROS 软件的关键方式。



■图 5.10　ROS 的堆

ROS 是一种分布式处理框架。这使可执行文件能被单独设计，并且在运行时松散耦合。这些过程可以封装到包和堆中，以便共享和分发。

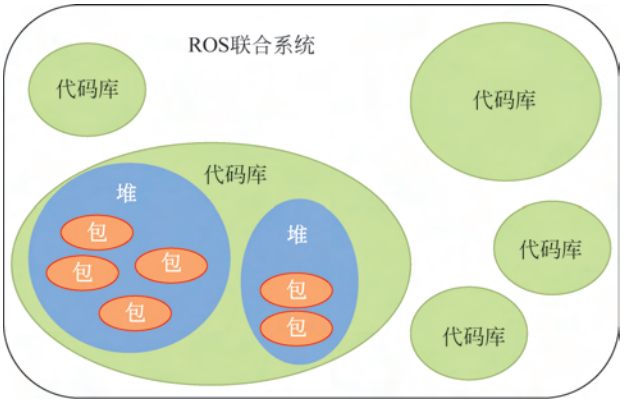
Manifests（manifest.xml）：提供关于 package 元数据，包括它的许可信息和 package 之间依赖关系，以及语言特性信息如编译旗帜（编译优化参数）。

Stackmanifests（stack.xml）：提供关于 Stack 元数据，包括它的许可信息和 Stack 之间依赖关系。

3. 社区级

ROS 的社区级概念是 ROS 网络上进行代码发布的一种表现形式。结构如图 5.11 所示。

代码库的联合系统，使得协作关系亦能被分发。这种从文件系统级到社区级的设计让 ROS 系统库的独立发展和工作实施成为可能。正因为这种分布式的结构，使得 ROS 迅速发展，软件仓库中包的数量随时间呈指数级增加。



■图 5.11 ROS 社区级构成

5.3 Apollo 平台计算框架——Cyber RT

5.3.1 Cyber RT 计算框架概述

1. 自动驾驶操作系统

在介绍 Cyber RT 之前，需要先了解自动驾驶操作系统。因为 Cyber RT 计算框架运行在自动驾驶操作系统之上。

自动驾驶操作系统是一种计算机操作系统，是计算平台的重要组成部分。与 Windows、Linux 等操作系统不同的是，自动驾驶操作系统的应用场景是自动驾驶汽车，可用于管理、调度和控制车载软硬件资源。

为了满足自动驾驶需求，自动驾驶操作系统需要具备以下功能。

（1）能够实现多种实时任务和计算任务的系统资源隔离、实时消息通信、实时任务调度、系统级访问控制等基础功能。

（2）有效管理系统资源，提高系统资源使用效率。

（3）可以适配接入摄像头、激光雷达、毫米波雷达等多类传感器设备，并有效屏蔽硬件物理特性和操作细节的差异性。

（4）能够承载运行实时环境感知、高精地图定位、决策规划与控制等自动驾驶核心部件。

（5）能够提供便捷、智能的人机交互功能服务。

如今世界上正在开发的自动驾驶操作系统很多，如英特尔的 Intel GO，特斯拉的 AutoPilot，NVIDIA 和谷歌推出的 PilotNet 等。CarOS 也是其中之一，它是百度研制的全球首个实时、可靠、安全、可控的自动驾驶操作系统，能够提供一整套完备的自动驾驶操作系统应用框架及服务。CarOS 作为上层应用软件和底层硬件之间的中间层，可以提供模块通信、服务调用等应用接口及丰富的传感器驱动等功能，是汽车电子系统的核心软件。

Cyber RT 计算框架是百度结合自动驾驶业务实际需求场景，自主设计、研发的车载操作系统，是 CarOS 全新一代的基础运行框架，该框架向下能够兼容主流硬件计算平台，向上

能够支持自动驾驶业务各个应用模块的实际业务需求。

2. ROS 的缺陷

在 Apollo 3.5 发布之前，Apollo 一直是以 ROS 为操作系统框架。但 ROS 在自动驾驶的实际应用中暴露出一些无法解决的缺陷。

(1) ROS 作为一个通用框架并未与自动驾驶现有的硬件，OS 等底层进行定制，有着较大的优化空间。

(2) ROS 软件包既繁且重，学习和使用成本较高。部署统一的开发、运行环境较为麻烦。

(3) 基于 ROS 的整体系统分散为大量的独立进程，集成度较低。系统的 CPU、GPU 以及内存等资源全部由各模块抢占，缺乏全局的分配调度系统。

3. Cyber RT 计算框架

为解决这些缺陷，急需一款专为自动驾驶场景开发的分布式计算框架，于是 Cyber RT 应运而生，并在 Apollo 3.5 版本中正式发布。它是全球首个面向自动驾驶的高性能开源计算框架，可显著提升研发效率，自适应设计易于部署，框架高效可靠，可以帮助客户实现更快速度的搭载与落地。其主要特点如下。

(1) 轻量级、平台无关。基于自动驾驶业务现状深度定制，精简可靠。框架与底层实现剥离，平台可移植性强。

(2) 采用 DAG 拓扑框架件，可使上层模块灵活配置。模块可定义独立的算法，以及输入、输出、异常处理等。可根据配置文件动态生成计算流程图并执行。

(3) 封装了简单高效的通信组件，可以满足不同驾驶场景下的信息传输需求。

(4) 任务的全局调度机制。通过计算流程图的数据依赖，进行任务的全局调度。

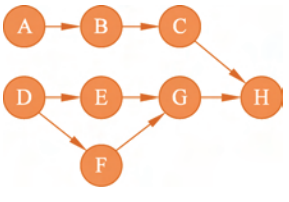
(5) 细粒度的资源隔离。根据功能对系统各组件进行计算、存储、I/O 等资源的预分配，运行时根据系统情况实时调整，兼顾稳定与高效。

5.3.2 Cyber RT 计算框架拓扑试验

1. 拓扑组件概述

在图论中，如果一个有向图从任意顶点出发无法经过若干条边回到该点，则这个图是一个有向无环图(DAG 图)。有向无环图的拓扑关系如图 5.12 所示，“有向”指的是消息传输过程中任意一条边有方向，若该边可双向传输，则为无向，“无环”指的是任务的通信过程中不存在环路，即消息从任务 A 点出发经 B 点经 C 点，不可再回到 A 点，否则为有环图。

可以看出，有向无环图的特点是去中心化，在一个分布有众多节点的系统中，每个节点都具有高度自治的特征。节点之间彼此可以自由连接，避免了中心故障将影响整个系统的缺陷，从而形成更扁平化、更平等的系统拓扑结构。有向无环图的另一个特点是可以有多个出度，因此可以同时处理多个出度连接的节点，如图 5.12 中，D 的输出可以同时发送给 E、F、G 可以同时接受 E、F 的输入进行处理。这样的特性可以加快处理任务的速度，且使其拓展性得到提高。

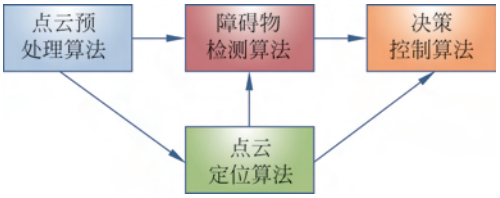


■图 5.12 DAG 拓扑

由上可知，利用 DAG 拓扑框架可将计算任务分成各个子任

务，每个子任务独立完成各自计算，并形成输入输出依赖关系。在 Cyber RT 框架中，最小的任务处理单元就是各类算法，因此每个算法可以独立编写。同时 Cyber RT 为各个算法定义了一套简单的描述语言规则，即 DAG 文件。应用模块算法只需通过配置 DAG 文件，来定义算法的向上依赖、向下输出和算法的具体实现逻辑即可。Cyber RT 调度器根据全局配置文件，加载对应应用模块算法及处理逻辑，生成通信拓扑图和处理任务集，然后根据任务的输入、输出以及执行时间的先后依赖关系，将不同任务调度到不同的容器中执行。多进程模式下，所有模块的算法根据模块内部的 DAG 拓扑图组装成进程，算法模块进行进程间通信；单进程模式下，所有模块算法根据模块 DAG 拓扑图进行总体挂接，算法模块进行进程内通信。

在自动驾驶系统的实际应用中，核心算法包括点云预处理算法、障碍物检测算法、点云定位算法、决策控制算法等。这些算法在 Cyber RT 框架下相互解耦，被封装成独立可拆分的计算任务单元，任务单元之间彼此相互独立，只是通过输入和输出完成数据链路连接，如图 5.13 所示。



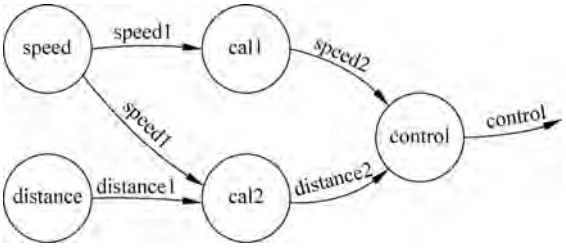
■图 5.13　自动驾驶系统算法框架

可以看出，利用 Cyber RT 拓扑框架可以使各个算法之间耦合降至最小，应用模块可以实时调整某一算法策略模块而不影响整体算法链路的正常执行，实现动态可“插拔”。例如点云定位算法存在 CPU 版本和 GPU 版本，应用可根据实际硬件负载情况选择合适的算法版本“插入”自动驾驶原有整体算法链路中，无须做其他代码上的变更。

2. Cyber RT 拓扑试验

1) Cyber RT 拓扑实验背景介绍

在自动驾驶算法体系中，障碍物的识别以及车速的控制一直是比较重要的环节，本次实验设计了一个基于前方障碍物距离与车速的控制实验，旨在介绍 Cyber RT 的拓扑通信流程，如图 5.14 所示。



■图 5.14　实验拓扑图

按照上述实验拓扑图创建 5 个 component，分别是 speed、distance、cal1、cal2 和 control，每个 component 之间的连线相当于一个 channel。其中 speed 组件主要用来从汽车的 CAN 总线上获取车速信号，然后在 CAN 总线的报文中把车速信息提取出来，根据相应

的协议转换计算得到车速信息，再把这些信息传送到 cal1 和 cal2 组件进行进一步处理。distance 组件用来获得毫米波雷达或者激光雷达测得的信息，在组件内部将信息处理后得到前方障碍物的距离信息，然后把信息传到 cal2 组件中。cal1 和 cal2 组件是计算组件，其作用是将速度信息或者距离信息计算得到控制信号，例如说危险等级或者制动程度等，这些信息最终会被 control 组件读取，综合判断汽车是否处于危险状况，输出最终的制动指令。

2）Cyber RT 拓扑实验程序编写

（1）component 简介。

每个 component 文件夹中都有 5 个文件，分别是. cc 源文件、. h 头文件、. dag 文件、. launch 文件和 BUILD 文件，图 5. 15 是 cyber/examples/common_component_example 文件夹下，已经给定的一个参考的 component。

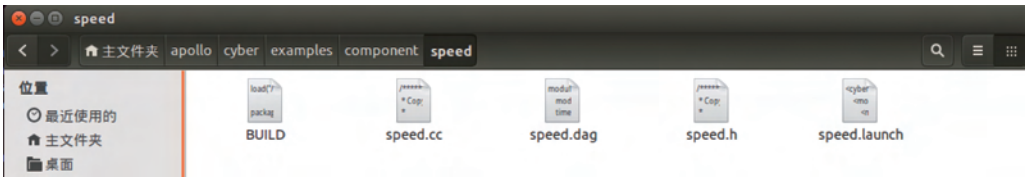
■图 5. 15　已经给定的 component 参考实例

图中 common_component_example. cc 和 common_component_example. h 文件用来放置程序代码，common. dag 文件用来配置动态链接库的路径，以及 component 的拓扑信息，common. launch 文件用来配置 component 的启动信息，BUILD 用来生成和配置依赖库、动态链接库等信息。Cyber RT 提供了两种 component，一种是基于消息触发的普通 component（具体文件在 cyber/examples/common_component_example 文件夹下），只要该 component 接收到某一 channel 的信息，就会被触发；另一种是基于时间触发的 TimerComponent（具体文件在 cyber/examples/timer_component_example 文件夹下），按照设计的时间间隔，源源不断地向指定的 channel 发送信息。下面将按照最开始的实验例子，对两种 component 的实现进行详细介绍。

（2）基于时间触发的 TimerComponent 介绍与设计。

① 创建文件夹，存放 component 文件。

在 cyber/examples 文件夹下新建 component 文件夹用于存放所有的 component，在 component 文件夹下新建一个 speed 文件夹用于存放 speed 组件的所有文件。然后将 timer_component_example 中的文件复制过来并重新命名，如图 5. 16 所示。其他 component 的创建与之类似。在基于障碍物距离与车速的控制实验中，speed 和 distance 组件用的是基于时间触发的 TimerComponent，其他组件都是基于消息触发的普通 component，speed 组件就是模拟输出车速信息。



■图 5. 16　speed component 文件

② speed.h 和 speed.cc 文件。

下面是 speed.h 的代码部分：

```
#include <memory>
#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/component/timer_component.h"
#include "cyber/examples/proto/examples.pb.h"
using apollo::cyber::examples::proto::Chatter;
using apollo::cyber::Component;
using apollo::cyber::ComponentBase;
using apollo::cyber::TimerComponent;
using apollo::cyber::Writer;
class speed : public TimerComponent {
 public:
  bool Init() override;
  bool Proc() override;
 private:
  std::shared_ptr<Writer<Chatter>> speed_writer_ = nullptr;
};
CYBER_REGISTER_COMPONENT(speed)
```

在 speed.h 中，第 11 行定义了一个继承自 TimerComponent 的 speed 类，表明该组件是一个基于时间触发的 component。speed 类有两个成员函数 Init()、Proc()和一个智能指针 speed_writer_。Init()函数可以类比 main 函数，做一些算法的初始化工作，Proc()函数是用于处理和输出消息，在 TimerComponent 中根据设置的时间间隔，周期性地调用该函数。

下面是 speed.cc 文件代码部分：

```
#include "cyber/examples/component/speed/speed.h"
#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/examples/proto/examples.pb.h"
bool speed::Init() {
  speed_writer_ = node_->CreateWriter<Chatter>("/carstatus/speed1");
  return true;
}
bool speed::Proc() {
  static int i = 0;
  auto out_msg = std::make_shared<Chatter>();
  out_msg->set_timestamp(i++);
  out_msg->set_content(70);
  speed_writer_->Write(out_msg);
  AINFO << "speed: Write drivermsg->"
     << out_msg->content();
  return true;
}
```

在 Init()函数中通过 node 的接口 CreatWriter 创建了一个 writer，向"/carstatus/

speed1"channel 中发送信息。在 Proc()函数中,set_msg_id()是定义信息的 ID 信息,set_content()是设置信息的内容,最终把要发送的信息输出。在这里,我们设置消息的 ID 是从 0 开始,每次调用该函数时 ID 加 1,同时把消息通过"/carstatus/speed1"channel 发送出去。

③ BUILD 文件。

```
load("//tools:cpplint.bzl", "cpplint")
package(default_visibility = ["//visibility:public"])
cc_binary(
  name = "speed.so",
  linkopts = ["-shared"],
  linkstatic = False,
  deps = [":speed_lib"],
)
cc_library(
  name = "speed_lib",
  srcs = [
    "speed.cc",
  ],
  hdrs = [
    "speed.h",
  ],
  deps = [
    "//cyber",
    "//cyber/examples/proto:examples_cc_proto",
  ],
)
cpplint()
```

BUILD 文件主要是配置依赖文件和 so 库。

④ speed.dag 文件。

```
module_config {
  module_library : "/apollo/bazel-bin/cyber/examples/component/speed/speed.so"
  timer_components {
    class_name : "speed"
    config {
      name : "speed"
      interval : 1000
    }
  }
}
```

module_library 是需要加载的 so 库路径,根目录为 cyber 的工作目录,timer_components 是根据需要加载的 class 的基类类型,class_name 是我们在 speed.h 文件中定义的 speed 类名,加载后的 class_name,作为加载示例的标识,interval 规定了发送消息的周期,为了演示方便,这里设置为 1000,也就是每隔 1000ms 发送一次消息。

⑤ speed.launch 文件。

```
<cyber>
```

```
2.     < module >
3.       < name > speed </name >
4.       < dag_conf >/apollo/cyber/examples/component/speed/speed.dag </dag_conf >
5.       < process_name > speed </process_name >
6.     </module >
7.   </cyber >
```

launch 文件顾名思义就是启动文件，< name >后面的名字必须为 speed.h 文件中定义的 speed 类名，< dag_conf >是我们前面配置好的 dag 文件及其路径。< process_name >是线程的名字，如果其他某个组件也用这个名字，那么这两个组件就运行在同一进程下，否则就在不同的进程下。基于 component 的拓扑通信可以通过 launch 文件来加载到不同的进程当中，部署灵活，并且支持接收多路数据并提供多种融合策略。

（3）基于消息触发的普通 component 介绍与设计。

在本次实验中，cal2 就是基于普通 component 设计的一个组件，其功能是读取 speed 发送到"/carstatus/speed1"channel 的信息和 distance 发送到"/carstatus/distance1"channel 的信息，从而在内部进行判断，得出是否采取制动措施，并将该信息通过"/carstatus/distance2"channel 发送出去。

① cal2.h 和 cal2.cc 文件。

下面是 cal2.h 文件：

```
1.   # include < memory >
2.   # include "cyber/class_loader/class_loader.h"
3.   # include "cyber/component/component.h"
4.   # include "cyber/examples/proto/examples.pb.h"
5.   # include "cyber/component/timer_component.h"
6.   using apollo::cyber::examples::proto::Chatter;
7.   using apollo::cyber::Component;
8.   using apollo::cyber::ComponentBase;
9.   using apollo::cyber::TimerComponent;
10.  using apollo::cyber::Writer;
11.  class cal2 : public Component < Chatter, Chatter > {
12.  public:
13.   bool Init() override;
14.   bool Proc(const std::shared_ptr < Chatter > & msg0,
15.         const std::shared_ptr < Chatter > & msg1) override;
16.  private:
17.   std::shared_ptr < Writer < Chatter >> cal2_writer_ = nullptr;
18.  };
19.  CYBER_REGISTER_COMPONENT(cal2)
```

同样地，定义了一个继承自普通 component 的一个类，与 TimerComponent 不同的是，该类里面有两个模板参数。由于添加了私有成员智能指针，因此需要用到 TimerComponent 中的一些头文件和命名空间。

下面是 cal2.cc 文件。

```
1.   # include "cyber/examples/component/cal2/cal2.h"
2.   # include "cyber/time/rate.h"
```

```
#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/time/time.h"
#include "cyber/cyber.h"
#include "cyber/examples/proto/examples.pb.h"
using apollo::cyber::Rate;
using apollo::cyber::Time;
using apollo::cyber::examples::proto::Chatter;
bool cal2::Init() {
  AINFO << "cal2 component init";
  cal2_writer_ = node_->CreateWriter<Chatter>("/carstatus/distance2");
  return true;
}
bool cal2::Proc(const std::shared_ptr<Chatter>& msg0,
                const std::shared_ptr<Chatter>& msg1) {
 AINFO << "Start cal2 component Proc [" << msg0->content() << "] ["
    << msg1->content() << "]";
 static int i = 0;
 auto out_msg = std::make_shared<Chatter>();

if(msg0->content()>60&&msg1->content()<80) {
  out_msg->set_content(1);
}
else {
  out_msg->set_content(0);
}

 out_msg->set_timestamp(i++);
 cal2_writer_->Write(out_msg);
 AINFO << "cal2: Write drivermsg->"
    << out_msg->content();
 return true;
}
```

与 TimerComponent 不同的是，Proc(const std::shared_ptr<Chatter>& msg0,const std::shared_ptr<Chatter>& msg1)函数多了两个形参，该组件收到两个 channel 的信息时才会进入该函数，从而进行下一步的操作，这也是基于消息触发的原因所在。

② dag 文件。

```
#Define all coms in DAG streaming.
  module_config {
  module_library : "/apollo/bazel-bin/cyber/examples/component/cal2/cal2.so"
  components {
    class_name : "cal2"
    config {
      name : "common"
      readers {
        channel: "/carstatus/speed1"
      }
      readers {
```

```
12.           channel: "/carstatus/distance1"
13.         }
14.       }
15.     }
16.   }
```

类似地，module_library 是需要加载的 so 库路径，根目录为 cyber 的工作目录，components 根据需要加载的 class 的基类类型，class_name 是我们在 cal2.h 文件中定义的 cal2 类名，加载后的 class_name，作为加载示例的标识，两个 readers 规定了往哪些 channel 中读取信息。cal2 组件的其他文件(launch 文件和 build 文件)的配置与前面 speed 组件的配置几乎相同，这里不再赘述。其他组件的配置也参考前面两种组件的配置来进行，在这里同样不再赘述。

③ proto 文件。

在定义智能指针时，用到了消息类型 Chatter，该类型的定义在 cyber/examples/ptoto 文件夹下的 examples.proto 文件中，下面是全部代码。

```
1.  syntax = "proto2";
2.  package apollo.cyber.examples.proto;
3.  message SamplesTest1 {
4.    optional string class_name = 1;
5.    optional string case_name = 2;
6.  };
7.  message Chatter {
8.    optional unit64 timestamp = 1;
9.    optional unit64 lidar_timestamp = 2;
10.   optional unit64 seq = 3;
11.   optional unit64 content = 4;
12. };
13. message Driver {
14.   optional string content = 1;
15.   optional unit64 msg_id = 2;
16.   optional unit64 timestamp = 3;
17. };
```

可以看到，Cyber RT 为我们提供了几种消息类型，同时我们也可以自己定义需要的消息类型，本次实验我们采用的是 Chatter 类型，由于我们要传递的是数值信息，因此需要在第 11 行将 Chatter 类型下面的 content 类型名由 bytes 改为 uint64。

(4) 判断逻辑。

下面是 cal1.cc 的 proc()函数：

```
1.  bool cal1::Proc(const std::shared_ptr<Chatter>& msg0) {
2.   AINFO << "Start cal1 component Proc [" << msg0->content() << "]";
3.   static int i = 0;
4.   auto out_msg = std::make_shared<Chatter>();
5.   if(msg0->content()>100) {
6.   out_msg->set_content(1);
```

```
7.  }
8.   else {
9.   out_msg->set_content(0);
10. }
11.  out_msg->set_timestamp(i++);
12.  cal1_writer_->Write(out_msg);
13.  AINFO << "cal1: Write drivermsg->"
14.      << out_msg->content();
15.  return true;
16. }
```

在cal1组件中,在cal1.cc文件中判断输入的"/carstatus/speed1"channel的信息是否大于100,如果大于100,则向"/carstatus/speed2"channel发出1(制动信号),代表超速需要制动,否则输出0(不制动信号)。

下面是cal2.cc的proc()函数:

```
1.  bool cal2::Proc(const std::shared_ptr<Chatter>& msg0,
2.                  const std::shared_ptr<Chatter>& msg1) {
3.   AINFO << "Start cal2 component Proc [" << msg0->content() << "] ["
4.     << msg1->content() << "]";
5.   static int i = 0;
6.   auto out_msg = std::make_shared<Chatter>();
7.  if(msg0->content()>60&&msg1->content()<80) {
8.   out_msg->set_content(1);
9.  }
10. else {
11.   out_msg->set_content(0);
12. }
13.  out_msg->set_timestamp(i++);
14.  cal2_writer_->Write(out_msg);
15.  AINFO << "cal2: Write drivermsg->"
16.    << out_msg->content();
17.  return true;
18. }
```

在cal2组件中,在cal2.cc文件中判断输入的"/carstatus/speed1"channel的信息是否大于60,并且"/carstatus/distance1"channel的信息是否小于80,即如果车速大于60并且与前方障碍物距离不足80,认为汽车处于危险状况,向"/carstatus/distance2"channel输出1,否则输出0。

下面是control.cc的proc()函数:

```
1.  bool control::Proc(const std::shared_ptr<Chatter>& msg0,
2.                     const std::shared_ptr<Chatter>& msg1) {
3.   AINFO << "Start control component Proc [" << msg0->content() << "] ["
4.     << msg1->content() << "]";
5.   static int i = 0;
6.   auto out_msg = std::make_shared<Chatter>();
```

```
7.    if(msg0 -> content()> 0||msg1 -> content()> 0) {
8.    out_msg -> set_content(1);
9.   }
10.   else {
11.    out_msg -> set_content(1);
12.  }
13.   out_msg -> set_timestamp(i++);
14.   control_writer_ -> Write(out_msg);
15.   AINFO << "control: Write drivermsg ->"
16.     << out_msg -> content();
17.   return true;
18.  }
```

在 control 组件中，接收"/carstatus/speed2" channel 和"/carstatus/distance2" channel 的信息，如果两个 channel 里面有一个判断为危险工况，该组件通过"/carstatus/control" channel 向控制单元输出制动信息。

(5) launch 文件。

需要配置一个总的 launch 文件来启动所有的 component，本次实验在 control 组件的 control.launch 文件中进行设置，只需将每个 component 的 launch 文件添加到 control.launch 文件之中即可。当启动 control.launch 时，其他的 component 也一并启动。

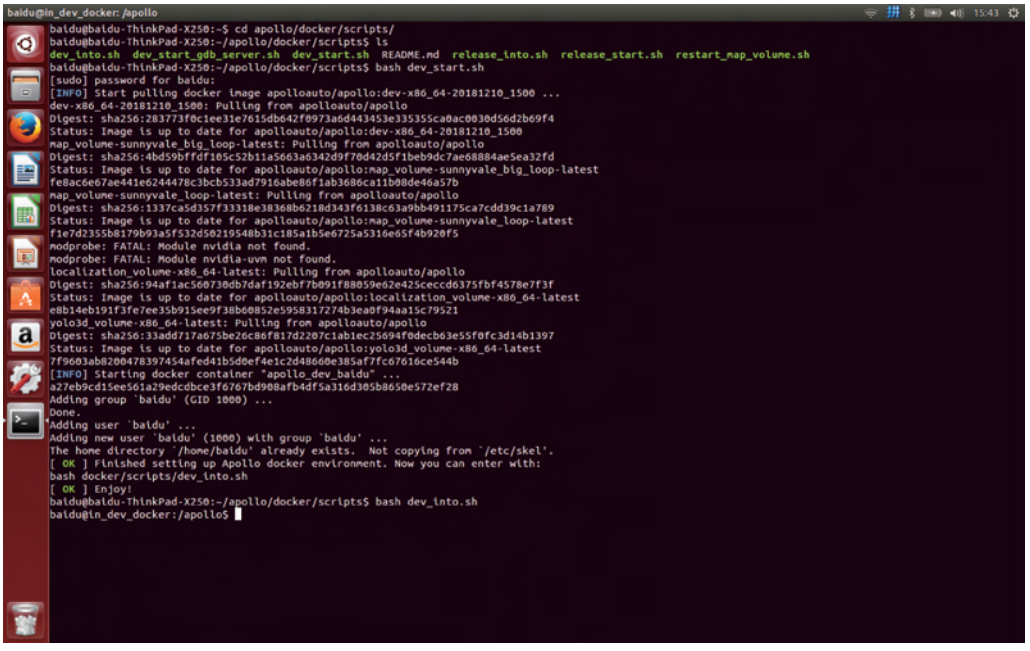
```
1.  <cyber>
2.    <module>
3.      <name> speed </name>
4.      <dag_conf>/apollo/cyber/examples/component/speed/speed.dag </dag_conf>
5.      <process_name> speed </process_name>
6.    </module>
7.    <module>
8.      <name> cal1 </name>
9.      <dag_conf>/apollo/cyber/examples/component/cal1/cal1.dag </dag_conf>
10.     <process_name> cal </process_name>
11.   </module>
12.   <module>
13.     <name> cal2 </name>
14.     <dag_conf>/apollo/cyber/examples/component/cal2/cal2.dag </dag_conf>
15.     <process_name> cal </process_name>
16.   </module>
17.   <module>
18.     <name> control </name>
19.     <dag_conf>/apollo/cyber/examples/component/control/control.dag </dag_conf>
20.     <process_name> control </process_name>
21.   </module>
22.   <module>
23.     <name> distance </name>
24.     <dag_conf>/apollo/cyber/examples/component/distance/distance.dag </dag_conf>
25.     <process_name> distance </process_name>
26.   </module>
27. </cyber>
```

3）实验运行和分析

（1）进入 docker 环境。

进入 docker/scripts 文件夹中的 cd apollo/docker/scripts/。

bash dev_start.sh 系统提示输入密码，如果系统提示出现错误，检查是否成功连接网络或者密码是否输入正确。然后再输入 bash dev_into.sh 即可进入 docker 环境，如图 5.17 所示。



■图 5.17 进入 docker 环境

（2）编译 cyber。

进入 docker 容器之后，在 apollo 目录下运行命令：bazel build cyber/…，没有问题则进入下一步的操作，如果出现编译错误，则检查文件是否配置正确。这一步可以放在最开始没有创建 component 的时候，先编译一遍，查看下载的源代码是否存在问题，然后再单独编译我们修改的 component 文件，即 bazel build cyber/examples/component/…，这样可以缩短编译时间，并且保证我们创建 component 时修改的文件（例如 proto 文件等）不会引起 component 之外的其他文件的错误。

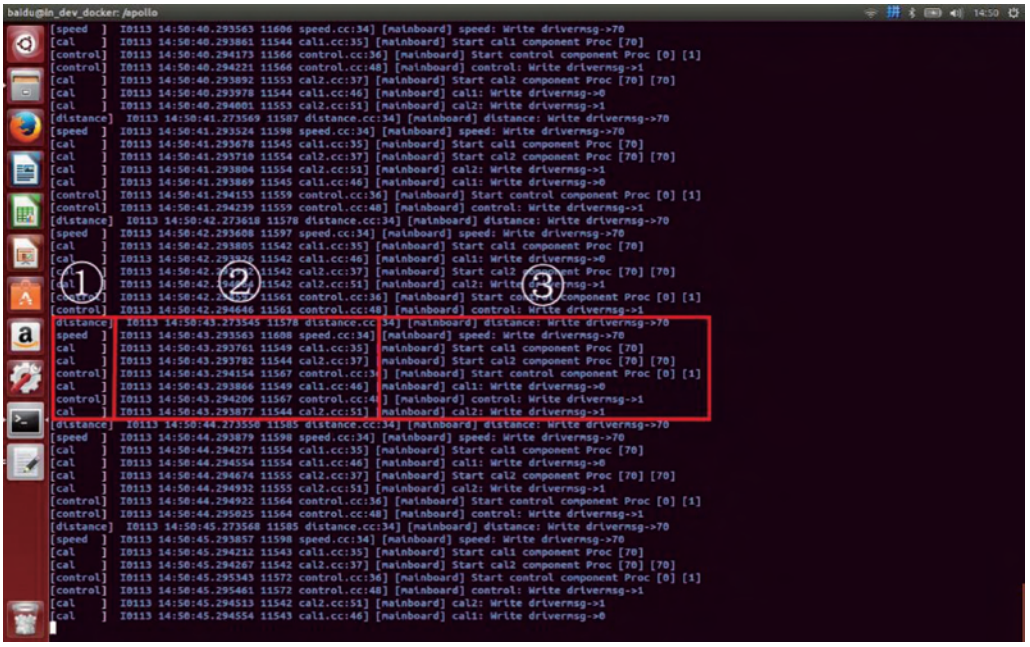
（3）启动整个 component 拓扑。

运行 control.launch 文件，在终端中输入：

```
cyber_launch start cyber/examples/component/control/control.launch
```

最终运行的结果如图 5.18 所示。

红色方框表示一帧信息，即完成一次拓扑之后的输出结果，其中①部分是线程的名字，即在 launch 文件中 <process_name>后面定义的名字，同一名字指的是组件运行在同一进程中。②部分中，I 代表 information 的意思，这一段输出在 setup.bash 文件中可以设置，后面的数字代表时间，再后面是输出信息程序所在的文件名和行号。③是输出的信息，在配置



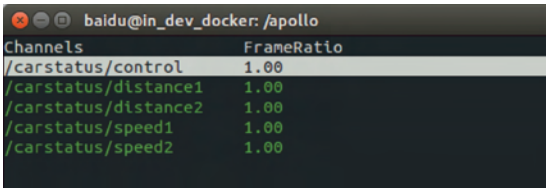
■图 5.18 运行结果

的.cc 文件中体现。

从这里可以看出,当车速设置为 70 时,cal1 组件判断汽车并没有超速,输出为 0,当车速为 70 的同时设置前方障碍物距离为 70 时,cal2 组件判断汽车处于危险状况,输出为 1,而 control 组件接收到车辆的危险信号时认为汽车应当减速制动,输出为 1。

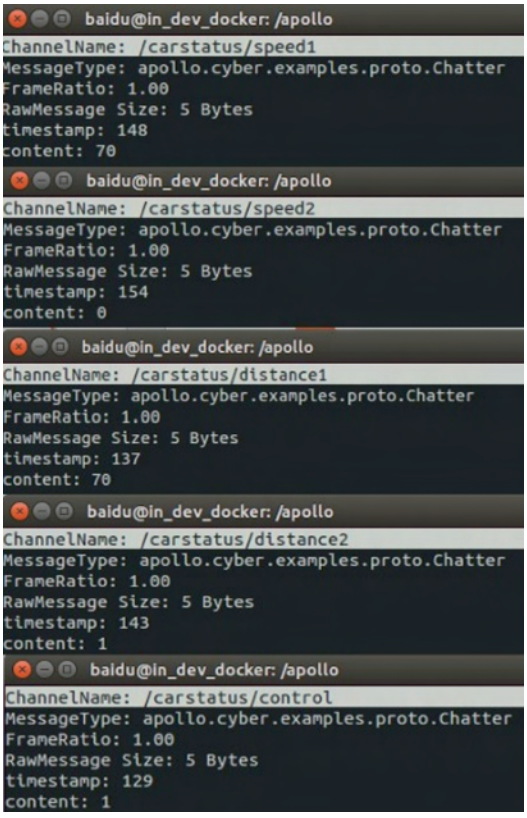
(4) 利用 cyber_monitor 查看每个 channel。

在进行实验的过程中,我们可以利用 cyber_monitor 工具查看当的 channel 信息。命令行工具 cyber_monitor 是一个用于显示 cybertron-apollo 通道数据的可视化工具,依赖于 cybertron-apollo 库,因此在使用前,需要 source 过 install 目录下的 setup.bash 文件,重新打开一个终端,进入 docker 环境之后,在命令行输入 cyber_monitor,然后会出现如图 5.19 所示的画面。



■图 5.19 cyber_monitor

默认显示为红色,若某一通道上有数据到达,则该通道对应的行显示为绿色。第一列列举了当前运行的所有通道的信息,可以看到我们创建的 5 个通道上面都有数据,第二列是通道数据的频率信息。当我们单击键盘上下箭头键,把光标放在某一个 channel 上时,单击右箭头键即可看到详细的 channel 信息,把所有的 channel 信息整合在一起,如图 5.20 所示。



■图 5.20 cyber_monitor 具体 channel 信息

5.3.3 Cyber RT 计算框架通信组件

1. 基本概念

Cyber RT 中定义并封装好了一系列计算框架通信组件，不仅通信效率高，使用简单，还可以满足不同场景不同功能的通信要求。例如在 Cyber RT 中，我们可以根据是否需要请求应答，选择基于信道的通信或是基于服务的通信。我们还可以根据主机及进程搭建环境的需要，选择是同一进程内通信、同一主机进程间通信还是跨机间的通信方式。

下面介绍 Cyber RT 中的一些基本通信组件概念。

(1) 节点(Node)：在 Cyber RT 框架中，节点是最基础的单元，每个节点都有各自独立的算法程序，如点云预处理算法、障碍物检测算法等。它能够基于信道、服务等功能与其他节点进行通信，各个节点之间进行通信即可形成拓扑关系，并完成指定任务。通过使用节点，可将代码和功能解耦，提高了系统容错能力和可维护性，使系统简化。同时，节点允许了 Cyber RT 能够布置在任意多个机器上并同时运行。

(2) 信道(Channel)：若需要完成节点之间的通信，则需要建立一条信息传输通道，在 Cyber RT 中称为信道。节点可以将消息发送进入某一指定的信道之中，若有其他节点定义接口接收此信道的消息，则可完成消息收发过程。若没有，则消息也依然存在于信道之中。

(3) 服务(Service)：服务是 Cyber RT 中的另一种通信方式，与信道通信相同，基于服

务的通信也需要有消息的收发节点。但与信道不同的是，服务需要两个节点之间完成请求或应答才可完成通信。

2. 基于信道的通信

若需要完成基于信道的通信，首先需要定义消息的发送方(Writer)和接收方(Reader)，以保证消息可以通过 Writer 和 Reader 共同指定的 Channel，从一个节点传输到另一个节点。这类通信方式有以下特点。

(1) 同一个节点可以同时发送多条消息，也可以同时接受多条消息，即可以同时定义多个 Writer 和 Reader。

(2) 基于信道的通信是一种单向通信，消息只能由 Writer 传输到 Reader，而不能够反向传输。

(3) 信道中的消息不需要实时应答，也就是说，当某一条消息通过 Writer 送入 Channel 后，可以没有 Reader 来读取消息。当某一个 Reader 想要读取 Channel 中的信息时，Channel 中也许并没有消息输入。

3. 基于服务的通信

在自动驾驶系统中，除了各节点的消息发送和接收之外，很多场景还需要在节点之间进行双向通信，并能够获得应答。这就需要利用服务来通信。Service 是节点之间通信的另一种方式，不同于 Channel 的通信方式，Service 的一个节点如果想要获取信息，需要给另一个节点发送请求，以此来获取响应，这就完成了节点之间的双向通信。在 Service 中，发送请求的一方为客户端(Client)，接收请求的一端为服务端(Server)。

4. 三种通信模型

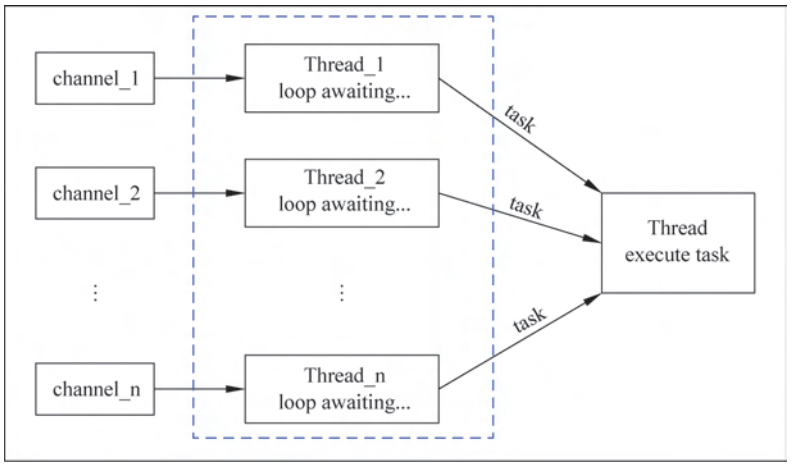
在 Cyber RT 中提供了不同的通信方式，以应对各类自动驾驶需求。根据上层模块所处的进程，可以将模块间的关系划分为同一进程内、同主机进程间和跨主机 3 种。

同一进程内通信指的是在同一个主机下的同一进程节点之间的相互通信，对于进程内的数据，直接传递消息对象的指针。这样可以避免消息数据复制的开销，尤其是一些较大的消息，如图像和点云等。

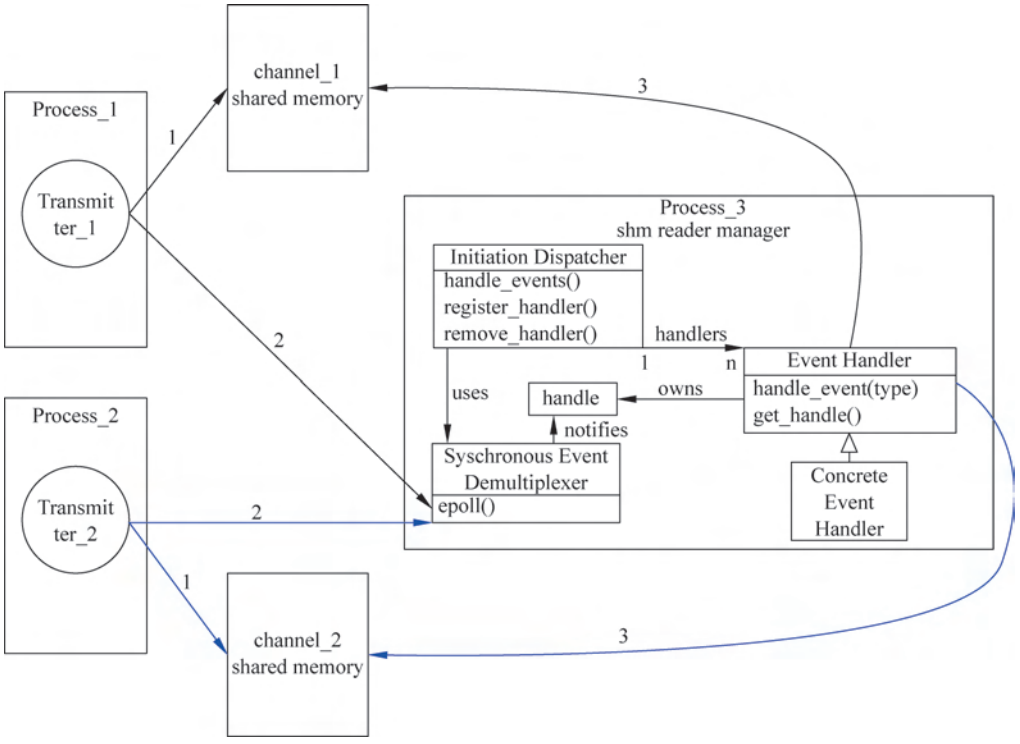
同主机进程间通信就是在同一个主机下，不同进程之间节点的传播或交换信息。对于跨进程的数据，我们利用共享内存传输，这样不仅可以减少传输中的数据复制，显著提升传输效率，还能够有效满足一对多的传输场景。在之后讲解的基于信道的通信实验中，就是同主机跨进程的通信方式，消息发送方和消息接收方在不同的进程下启动。共享内存通信可以分为 3 个主要步骤：写入数据、发出可读通知、读取数据。其中发出可读通知的通知机制有多种选择，例如采用进程间共享的条件变量唤醒。由于 channel 间的共享内存是隔离的，如果采取条件变量唤醒，每个跨进程通信的 channel 都需要创建一个读取线程，如图 5.21 所示。

为了减少线程数，采用了 UDP 组播的方式通知。这样，每个进程内的所有 channel 都可以共用一个线程去获取可读通知，线程数就实现了可控，如图 5.22 所示。

跨主机的数据利用 socket 传输。目前，跨主机通信采用了第三方的开源库 Fast RTPS (Real Time Publish Subscribe，实时发布订阅协议)。RTPS 是 DDS 标准中的通信协议，而 Fast RTPS 是支持 RTPS 协议版本的一个订阅发布消息的组件，具有高性能，实时性高，多



■图 5.21 跨进程通信的多个线程读取



■图 5.22 跨进程通信的共用线程

平台支持等优点。

5. 计算框架通信组件实验

1）基于信道的通信

在了解以上基础内容之后，我们就可以尝试创建自己的通信组件了。首先学习创建节点。在具体的通信任务中，节点的作用类似于句柄，在创建如 Writer 和 Reader 等具体功能对象时，需要基于已有的节点实例才能创建。节点在系统中必须有唯一的名称，以保证节点

使用特有名称与其他节点进行通信而不产生歧义。

我们尝试利用 CreateNode 接口创建一个名称为"talker"的节点如下：

```
auto talker_node = apollo::cyber::CreateNode("talker");
```

创建完节点之后，就可以自由选择通信方式了。首先尝试建立基于信道的通信。

消息通过指定的信道传输时，需要与该信道的消息类型相匹配。这些消息大多数都是 protobuf 格式（一种序列化数据结构的协议，可以用于网络通信和数据存储），会被定义保存在一个.proto 文件中。例如想要定义一个名为"Chatter"的消息类型，需要它包含消息时间戳、雷达时间戳、消息频率、文本内容等消息类型，就可以用如下代码完成：

```
1.  syntax = "proto2";
2.  package apollo.cyber.examples.proto;
3.  message Chatter {
4.    optional uint64 timestamp = 1;
5.    optional uint64 lidar_timestamp = 2;
6.    optional uint64 seq = 3;
7.    optional bytes content = 4;
8.  };
```

接下来就可以创建消息发送方和接收方了。为创建一个消息发送方，首先需要创建一个节点，之后需要基于这个节点创建一个 Writer。Writer 是 Cyber RT 中用于发送消息的组件，每个 Writer 对应一个 channel 及具体的数据类型。Writer 可以通过节点的 CreateWriter 接口创建，如下所示：

```
auto talker = talker_node->CreateWriter<Chatter>("channel/chatter");
```

可以看出，定义的 Writer 指定了 chatter 这类消息类型来传输消息，同时定义该信道名称为"channel/chatter"。

接下来我们就可以发送消息了，在 chatter 消息类型的范围内，我们可以发送消息时间戳、雷达时间戳、消息发送频率以及消息文本内容等类型，如下代码所示：

```
1.  while (apollo::cyber::OK()) {
2.   static uint64_t seq = 0;
3.   auto msg = std::make_shared<Chatter>();
4.   msg->set_timestamp(Time::Now().ToNanosecond());
5.   msg->set_lidar_timestamp(Time::Now().ToNanosecond());
6.   msg->set_seq(seq++);
7.   msg->set_content("Hello, apollo!");
8.   talker->Write(msg);
9.   AINFO << "talker sent a message!";
10.  rate.Sleep();
11. }
```

第 4～7 行定义了消息的类型。通过 Writer 的 Write 接口发送消息，完成之后利用 AINFO 将“talker sent a message!”文本发送到终端。完成以上工作就创建好了一个消息发送方。

接下来尝试创建一个消息接收方。与 Writer 一样，首先需要创建一个节点，之后在此

节点上创建一个 Reader。Reader 是 Cyber RT 中用于接收消息的组件，可以通过节点的 CreateReader 接口创建。Reader 在创建时会绑定一个回调函数，当该路 Channel 有新消息到达时，会调用回调函数处理。可以指定消息接收信道为 channel/chatter，回调函数为 MessageCallback，如下所示：

```
auto listener = listener_node->CreateReader<apollo::cyber::examples::proto::Chatter>
("channel/chatter", MessageCallback);
```

在回调函数定义中，可以将接收到的消息，如消息发送频率和消息文本内容等，利用 AINFO 输出到终端，如下所示：

```
1.  void MessageCallback(
2.    const std::shared_ptr<apollo::cyber::examples::proto::Chatter>& msg) {
3.    AINFO << "Received message seq->" << msg->seq();
4.    AINFO << "msgcontent->" << msg->content();
5.  }
```

这样就创建好了一个消息接收方，但是还需要做一些其他工作来保证通信成功运行。将消息发送方和消息接收方保存为.cpp 文件，消息发送方命名为 talker，消息接收方命名为 listener，名称可以任意指定。在该文件目录下的 Build 文件中添加：

```
1.  cc_binary(
2.    name = "talker",
3.    srcs = ["talker.cc"],
4.    deps = [
5.      "//cyber",
6.      "//cyber/examples/proto:examples_cc_proto",
7.    ],
8.  )
9.
10. cc_binary(
11.   name = "listener",
12.   srcs = ["listener.cc"],
13.   deps = [
14.     "//cyber",
15.     "//cyber/examples/proto:examples_cc_proto",
16.   ],
17. )
```

填写好 Build 文件就能利用 bazel build 命令来对这两个指定.cpp 文件进行编译了。我们首先进入 docker 环境中，输入 bazel build 命令编译消息发送方和消息接收方的所在文件夹：

```
bazel build /apollo/cyber/examples/...
```

编译成功之后输入：

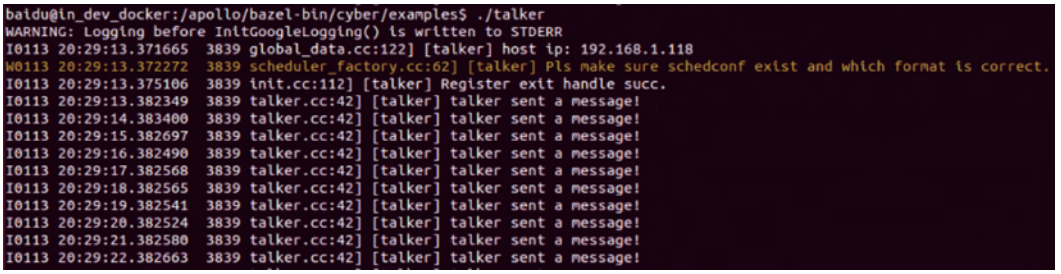
```
1.  cd /apollo/bazel-bin/cyber/examples
2.  ll
```

其中 cyber/examples 这个只是实验存放路径，可以任意指定其他路径存放.cpp 文件，

利用 ll 命令可以看到文件夹下是否有你所编写的.cpp 文件所转成的二进制文件，如果有，输入以下命令：

```
./talker
```

这样就可以看到终端有消息输出，如图 5.23 所示，说明 talker 已成功运行。

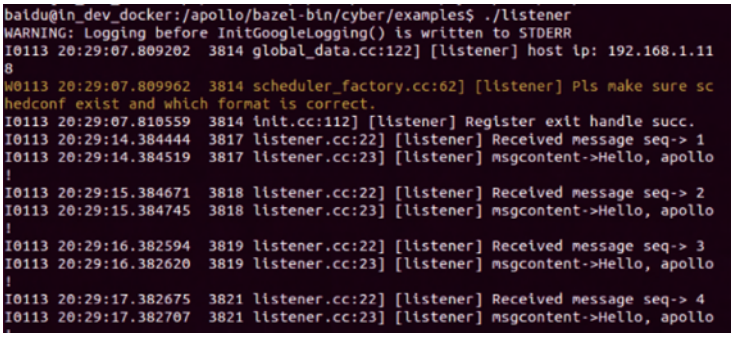


■图 5.23 启动 talker

因为将消息发送方和消息接收方分别保存为了两个.cpp 文件，意味着它们需要在两个终端中打开，即跨进程通信。打开另一个终端，进入 docker 环境中，重复上述 cd 命令，之后输入：

```
./listener
```

同时打开 talker 和 listener 就可以看到 listener 所在进程的终端有消息输出，如图 5.24 所示。



■图 5.24 启动 listener

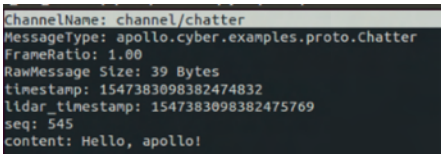
说明 talker 和 listener 之间通信成功。

可以通过 cyber_monitor 这个工具了解通信内容，另起终端进入 docker 镜像环境，输入：

```
cyber_monitor
```

会有如图 5.25 的显示。

我们可以清楚了解到通信的流程和内容：消息通过消息发送发 talker 发给名为 channer/chatter 的信道一个消息，消息内容有时间戳、雷达时间戳、频率和“Hello，apollo!”的消息文本，listener 通过同一个信道接收消息，并输出频率和消息文本信息。



■图 5.25 cyber_monitor 输出框

当然,消息发送发和消息接收方也可以定义在同一个终端中,只需要将 Writer 和 Reader 写在同一个.cpp 文件内完成编译即可。

2) 基于服务的通信

如果我们已经学会了建立基于信道的通信,那么学习建立基于服务的通信也就轻而易举了。服务通信模块与信道通信一样,首先需要创建一个节点,然后基于此节点利用 CreateService 接口,并指定消息格式,即可创建一个 Server。同时,还需要创建一个消息请求(Request)和回应(Response),以用于接收 Client 发送的请求,并在处理后返回。代码如下:

```
1.  auto server = node->CreateService<Driver, Driver>(
2.    "test_server", [](const std::shared_ptr<Driver>& request,
3.                std::shared_ptr<Driver>& response) {
4.      AINFO << "server: i am driver server";
5.      static uint64_t id = 0;
6.      ++id;
7.      response->set_msg_id(id);
8.      response->set_timestamp(0);
9.    });
```

上述实例中,第 1～2 行定义了一个 Server,指定消息格式为 Driver,命名为“test_server”,并创建了所需的 request 和 response。在第 7～8 行中利用 response 返回给 Client 处理好的消息 id 和时间戳。

接下来我们尝试创建一个 client,利用节点的 CreateClient 接口,并指定消息格式和 Server 名称来创建。之后还需要定义发送请求的消息类型,并利用 Client 的 SendRequest 接口来发送请求,代码如下所示:

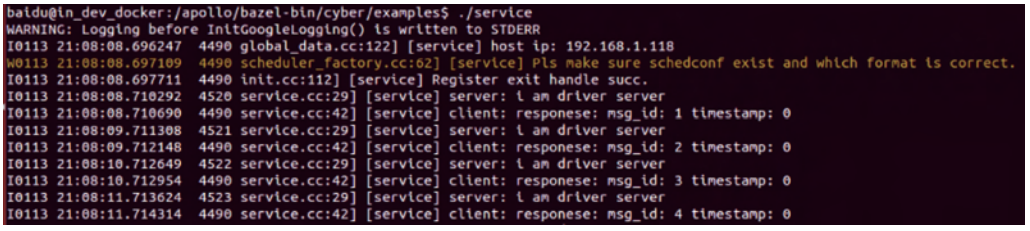
```
1.    auto client = node->CreateClient<Driver, Driver>("test_server");
2.  auto driver_msg = std::make_shared<Driver>();
3.  driver_msg->set_msg_id(0);
4.  driver_msg->set_timestamp(0);
5.  while (apollo::cyber::OK()) {
6.   auto res = client->SendRequest(driver_msg);
7.   if (res != nullptr) {
8.     AINFO << "client: responese: " << res->ShortDebugString();
9.   } else {
10.    AINFO << "client: service may not ready.";
11.  }
12.  sleep(1);
13. }
```

第 1 行定义了 Client，指定了 Driver 消息格式和之前定义的名为 test_server 的服务端，在第 2～4 行中定义了消息内容为 id 和时间戳，第 6 行发送消息请求，第 8 行收到 response 并输出到终端。

启动一个 Service 通信，可参考如上的操作，如保存. cpp 文件，修改 Build 文件，进入 docker 镜像，编译等等。完成后输入命令：

```
./bazel-bin/cyber/samples/service
```

可以看到终端显示如图 5.26 所示。



■图 5.26　service 通信试验

可以看到 Server 和 Client 都已启动，Client 发出请求，Server 处理后给出响应，Client 将响应即消息 id 进行输出。说明通信成功。这样就完成了客户端发送请求给服务端，服务端处理后返回响应的功能。

3）3 种通信模型

（1）同一进程内：若想实现同一进程内节点之间的通信，需要将通信组件创建并运行在同一进程之中。如上述讲解的基于服务的通信实验中，就是在同一进程内传递数据，Server 和 Client 都创建在同一个进程下。

（2）同主机进程间：若想实现同主机进程间的通信，只需要将使用的通信组件创建并运行在同一主机上，但不同的进程之间。如上述的基于信道的通信实验中，只需要在各个通信组件中定义好接口，在不同进程内运行 talker 和 listener，就可以完成信息收发了。

（3）跨主机：在 Cyber RT 中实现跨机通信，首先需要保证两台机器在同一个局域网中，配置好 SSH 服务，然后将/apollo/cyber 下的 setup. bash 文件进行修改，找到 export CYBER_DOMAIN_ID＝文本，将两台机器的 ID 数改为相同的数字。这样两台机器就可以完成跨机通信了。

5.3.4　Cyber RT 计算框架调度组件

1. Cyber RT 调度概述

在自动驾驶系统中，传感设备的增多带来了信息量的增大，系统的有向无环图也会越来越复杂。在有向无环图中，有些消息传递的路径比较短，有的比较长；有的没有复杂的计算仅仅实现消息的传递，但是有的却要进行各种计算，消耗比较多的系统资源。因此如何对系统的资源进行合理地配置，使得信息的传递及时、有序、准确和完整，是自动驾驶系统中比较重要的一个问题。

目前主流的自动驾驶系统通常使用 ROS（Robot Operating System）作为基础通信框

架。在这种通信框架下，各算法模块作为单独进程各自开发，存在资源相互抢占的风险，而且工程与算法深度耦合，每个算法模块定义为一个进程不方便总体协调和调度，模块内部包含大量的工程冗余建设，同时由于算法和工程耦合过于紧密，算法策略拆分以进行并行计算受到很大程度的制约。Cyber RT 将自动驾驶所有算法处理任务封装成一个最小计算单元，模块策略开发只需要定义算法以及算法的输入输出，模块之间的数据通信、服务调用、数据接口等放到优先级队列中由框架进行全局统一管理调度。Cyber RT 调度系统的调度策略主要分为 classic(经典)策略和 choreography(编排)策略，根据 DAG 文件描述生成自动驾驶模块算法拓扑链路，通过读取算法上下游依赖，确定算法执行先后顺序，然后分配执行单元进行运算。

2. Cyber RT classic 调度策略

classic 调度策略是 Cyber RT 较为通用的调度策略，如果对于车上的 DAG 结构不清楚时，优先使用该策略。该策略可以指定每个节点的优先级，可以设置节点运行在哪些 processor 上，当设置好之后，这些节点便会按照设定优先级的顺序，按照默认的调度策略运行在设置的 processor 上，系统会根据任务量的大小自动分配在不同的 processor 上运行，每个 processor 运行的负担差不多。

下面是 cyber/conf/example_classic.conf 文件：

```
scheduler_conf {
 policy: "classic"
 classic_conf {
  groups: [
   {
    processor_num: 2
    affinity: "range"
    cpuset: "0-1"
    tasks: [
     {
      name: "speed"
      prio: 10
     },{
      name: "cal1"
      prio: 11
     },{
      name: "control"
      prio: 12
     },{
      name: "distance"
      prio: 1
     },{
      name: "cal2"
      prio: 2
     }
    ]
   }
  ]
 }
```

```
30. }
```

其中，processor_num 指定分配调度线程的个数，affinity 是指定分配 CPU 的编排方式，rang 是一对多的编排方式，cpuset 是设置哪些 CPU 供使用，在 task 里面，每个 name 都是节点或者组件的名称，prio 是指该节点或者组件的优先级。

上述程序代码把 5.2.2 节拓扑试验的所有组件都进行了优先级的定义，其中 speed-> call-> control 设置的优先级比较高，其他路径设计的优先级比较低。

3. Cyber RT choreography 调度策略

choreography 策略是基于对车上任务足够熟悉，根据任务的依赖执行关系、任务的执行时长、任务 CPU 消耗情况、消息频率等，对任务进行编排。同样可以设置 classic 策略中的线程数目，编排方式之外，还可以设置每个线程的优先级，线程的调度策略等。

下面是 cyber/conf/example：

```
1.  scheduler_conf {
2.   policy: "choreography"
3.   choreography_conf {
4.    choreography_processor_num: 1
5.    choreography_affinity: "1to1"
6.    choreography_cpuset: "0"
7.    choreography_processor_policy: "SCHED_FIFO"
8.    choreography_processor_prio: 10
9.    pool_processor_num: 2
10.   pool_affinity: "range"
11.   pool_cpuset: "1"
12.   tasks: [
13.    {
14.     name: "control"
15.     processor: 0
16.     prio: 12
17.    },{
18.     name: "call1"
19.     processor: 0
20.     prio: 11
21.    },{
22.     name: "speed"
23.     processor: 0
24.     prio: 10
25.    }
26.   ]
27.  }
28. }
```

choreography_affinity 中的 1to1 是指一对一的编排方式，choreography_processor_policy 是设置调度的策略，其中 SCHED_FIFO 是先来先服务的策略，在这种策略模型下，Cybert RT 调度线程池根据算法元 task 的先后到达顺序，将 Task 逐一放到待调度队列中，处理线程池按照先来先处理的原则将调度队列中的 task 分配放到不同的执行器中消费。choreography_processor_prio 是设置线程的优先级。在 task 任务里只针对 speed->

cal1-> control 这一路径上的组件进行了调度的设置，指定了组件运行的 CPU 以及各自的优先级。

4. RT 调度试验

1）基于 classic 调度策略的调度实验

该调度实验是在 5.3.2 节拓扑组件实验章节的基础上进行的，当编写好 cyber/conf/example_classic.conf 文件之后，需要重新新建一个 launch 文件，并修改参数时期采用 example_classic.conf 文件的调度策略。

下面是 control_classic.launch 文件，在 process_name 里面把线程名字改成配置的文件名 example_classic。

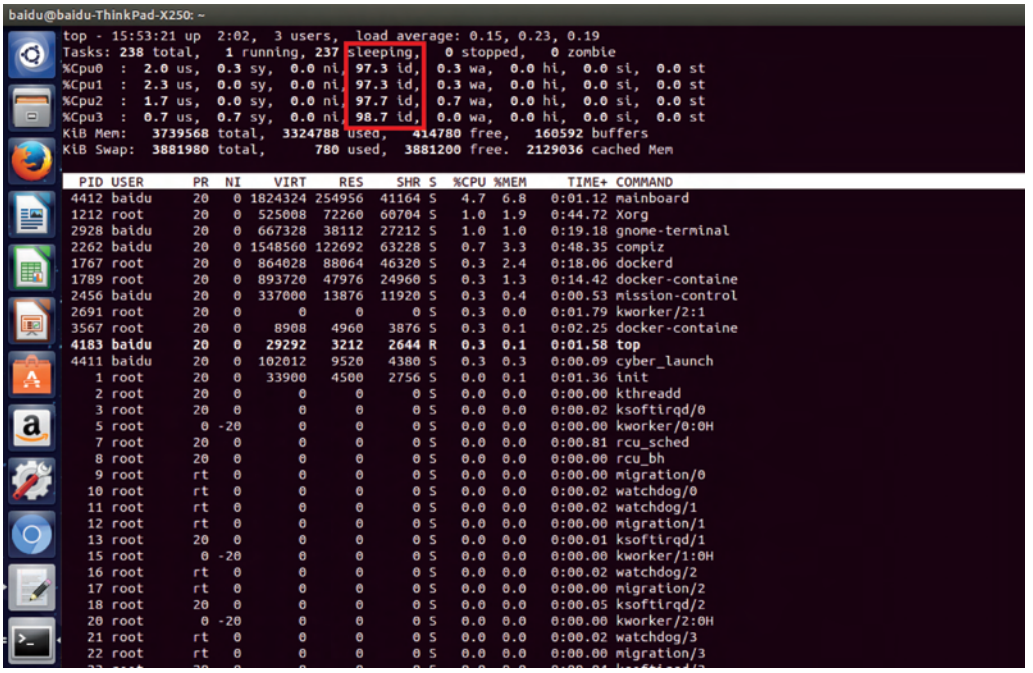
```
2.  <cyber>
3.    <module>
4.      <name>speed</name>
5.      <dag_conf>/apollo/cyber/examples/component/speed/speed.dag</dag_conf>
6.      <process_name>example_classic</process_name>
7.    </module>
8.    <module>
9.      <name>cal1</name>
10.     <dag_conf>/apollo/cyber/examples/component/cal1/cal1.dag</dag_conf>
11.     <process_name>example_classic</process_name>
12.   </module>
13.   <module>
14.     <name>cal2</name>
15.     <dag_conf>/apollo/cyber/examples/component/cal2/cal2.dag</dag_conf>
16.     <process_name>example_classic</process_name>
17.   </module>
18.   <module>
19.     <name>control</name>
20.     <dag_conf>/apollo/cyber/examples/component/control/control.dag</dag_conf>
21.     <process_name>example_classic</process_name>
22.   </module>
23.   <module>
24.     <name>distance</name>
25.     <dag_conf>/apollo/cyber/examples/component/distance/distance.dag</dag_conf>
26.     <process_name>example_classic</process_name>
27.   </module>
28. </cyber>
```

然后运行 cyber_launch start cyber/examples/component/control_classic.launch，让各组件运行起来。然后新建终端框，输入 top 命令查看 CPU 的运行情况，按下 1 键可查看每个 CPU 的工作情况，如图 5.27 所示。

可以看到，4 个 CPU 中睡眠的进程数目差不多，即每个 CPU 的负荷基本相同，也就是说系统按照配置的 classic 调度方案把所有的程序平均分配给每个 CPU 来运行。

2）基于 choreography 调度策略的调度实验

同样地，当编写好 cyber/conf/example_choreography.conf 文件之后，需要重新新建一个 launch 文件，并修改参数时期采用 example_choreography.conf 文件的调度策略。



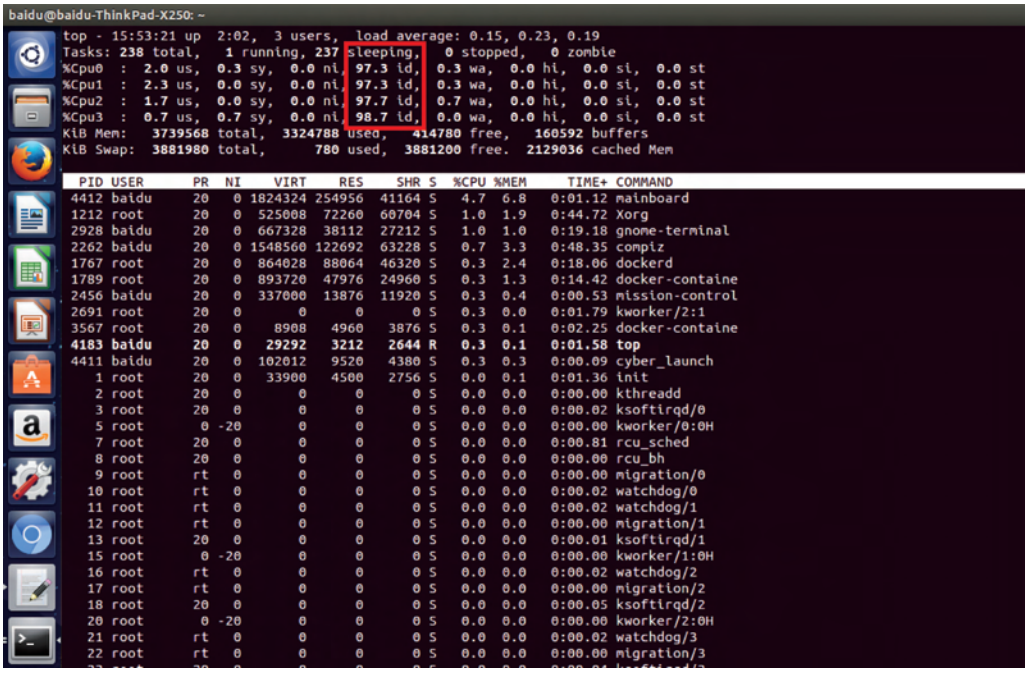
■图 5.27 classic 调度策略下的 CPU 工作情况

下面是 control_choreography.launch 文件，在 process_name 里面把线程名字改成配置的文件名 example_choreography 即可。

```
1.  <cyber>
2.    <module>
3.      <name>speed</name>
4.      <dag_conf>/apollo/cyber/examples/component/speed/speed.dag</dag_conf>
5.      <process_name>example_choreography</process_name>
6.    </module>
7.    <module>
8.      <name>cal1</name>
9.      <dag_conf>/apollo/cyber/examples/component/cal1/cal1.dag</dag_conf>
10.     <process_name>example_choreography</process_name>
11.   </module>
12.   <module>
13.     <name>cal2</name>
14.     <dag_conf>/apollo/cyber/examples/component/cal2/cal2.dag</dag_conf>
15.     <process_name>example_choreography</process_name>
16.   </module>
17.   <module>
18.     <name>control</name>
19.     <dag_conf>/apollo/cyber/examples/component/control/control.dag</dag_conf>
20.     <process_name>example_choreography</process_name>
21.   </module>
22.   <module>
23.     <name>distance</name>
24.     <dag_conf>/apollo/cyber/examples/component/distance/distance.dag</dag_conf>
```

```
25.      <process_name>example_choreography</process_name>
26.    </module>
27. </cyber>
```

然后运行 cyber_launch start cyber/examples/component/control_choreography.launch，让各组件运行起来。然后新建终端框，输入 top 命令查看 CPU 的运行情况，按下 1 键可查看每个 CPU 的运行情况，如图 5.28 所示。



■图 5.28 choreography 调度策略下的 CPU 工作情况

从图中可以看出，由于我们在 choreography 调度中使得试验程序在同一 CPU0 下运行，因此 CPU0 的睡眠进程数目少于其他的 CPU，这表明 CPU0 的负担比其他 CPU 的负担要重，也就间接地说明试验程序确实按照调度配置的要求来运行。

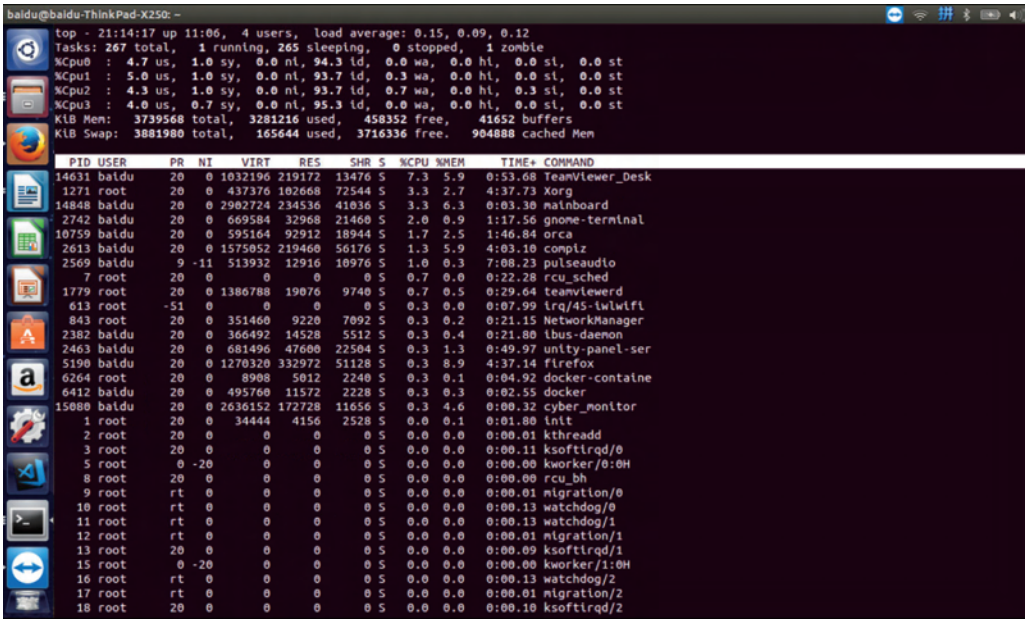
3）非编排的默认调度实验

前面通过编写 cyber/conf/文件夹下的 conf 文件来实现信息传递的调度方案，为了便于比较每个调度策略，本例程不采用任何的调度策略，直接进行运行，查看调度的情况。同样地，需要修改 control.launch 文件，程序如下所示：

```
1.  <cyber>
2.    <module>
3.      <name>speed</name>
4.      <dag_conf>/apollo/cyber/examples/component/speed/speed.dag</dag_conf>
5.      <process_name>control</process_name>
6.    </module>
7.    <module>
8.      <name>cal1</name>
9.      <dag_conf>/apollo/cyber/examples/component/cal1/cal1.dag</dag_conf>
10.     <process_name>control</process_name>
```

```
    </module>
    <module>
      <name>cal2</name>
      <dag_conf>/apollo/cyber/examples/component/cal2/cal2.dag</dag_conf>
      <process_name>control</process_name>
    </module>
    <module>
      <name>control</name>
      <dag_conf>/apollo/cyber/examples/component/control/control.dag</dag_conf>
      <process_name>control</process_name>
    </module>
    <module>
      <name>distance</name>
      <dag_conf>/apollo/cyber/examples/component/distance/distance.dag</dag_conf>
      <process_name>control</process_name>
    </module>
</cyber>
```

然后运行 cyber_launch start cyber/examples/component/control_choreography.launch,让各组件运行起来,如图 5.29 和图 5.30 所示。

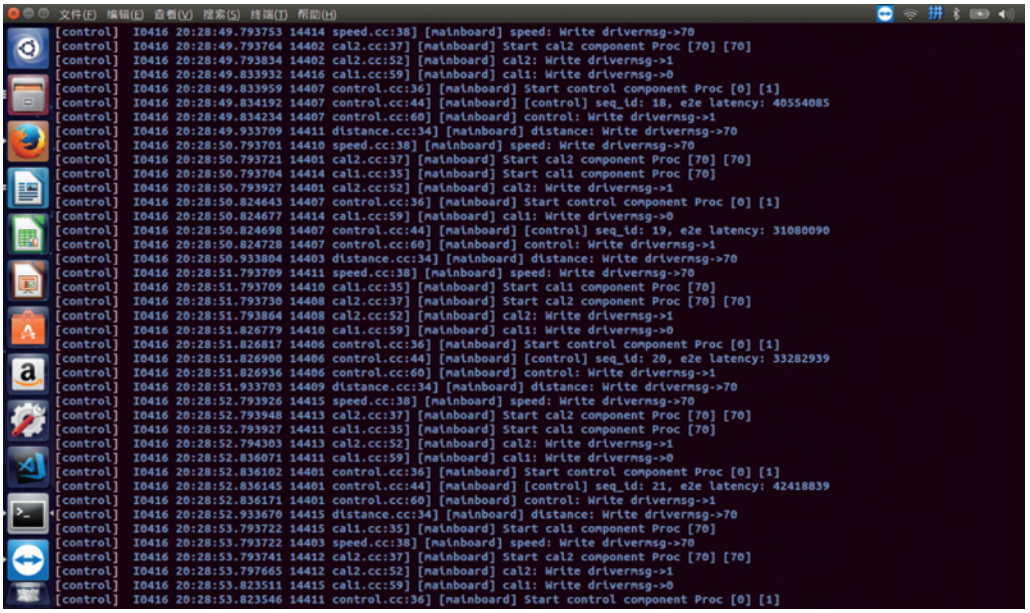


■图 5.29 无调度策略下的 CPU 工作情况

4) 3 种调度策略性能分析

(1) CPU 占用情况。

根据上面的运行结构,比较 3 种不同的调度策略的 CPU 占用情况可以对 3 种调度策略进行分析。首先在经典调度策略下,如图 5.27 所示,4 个 CPU 睡眠的进程数目几乎相同,说明每个 CPU 的负荷在经典调度策略下几乎平均分配,达到了比较良好的分配效果,如果每一条信道的流通数量和速度几乎一致,采用该调度策略无疑是比较好的选择。但是不同



■图 5.30 无调度策略下的运行情况

的传感器的信息量不同，需要处理的任务量也不尽相同，因此很难保证每一条信道都保持相同的负荷。一种常见的拓扑结构是某一条或几条信道信息量比较大，但是其他信道信息量较小，采用经典的调度策略难免会造成信息传递的延迟，因此 Cyber RT 支持进行编排的调度策略。如图 5.28 所示，在编排调度策略下，CPU0 的睡眠进程明显少于其他 CPU，我们可以将某一信道的数据传输固定到特定的 CPU 上，提高运行的效率。如图 5.29 所示，非编排的默认调度策略没有进行任何 CPU 的调度，每个任务随机地分配到 CPU 上，虽然每个 CPU 负荷大致相同，但是几个 CPU 之间的相对负荷比较随机，没有经典调度策略那样平均。

(2) Latency 时延情况。

在 speed.cc 文件的 proc()函数中，通过时间戳函数 out_msg-> set_lidar_timestamp (cur_time)将当前的时间戳发送出去。

```
1.  bool speed::Proc() {
2.    static int i = 0;
3.    auto out_msg = std::make_shared<EChatter>();
4.    // current time
5.    auto cur_time = apollo::cyber::Time::Now().ToNanosecond();
6.    out_msg->set_timestamp(cur_time);
7.    out_msg->set_lidar_timestamp(cur_time);
8.    out_msg->set_seq(i++);
9.    out_msg->set_content(70);
10.   speed_writer_->Write(out_msg);
11.   AINFO << "speed: Write drivermsg->"
12.       << out_msg->content();
13.   return true;
14. }
```

在 cal1.cc 文件的 proc()函数中,将时间戳信息 msg0-> lidar_timestamp()赋值给变量 lt,然后,将携带时间戳信息的变量 lt 通过时间戳函数 out_msg-> set_lidar_timestamp(lt)发送出去。

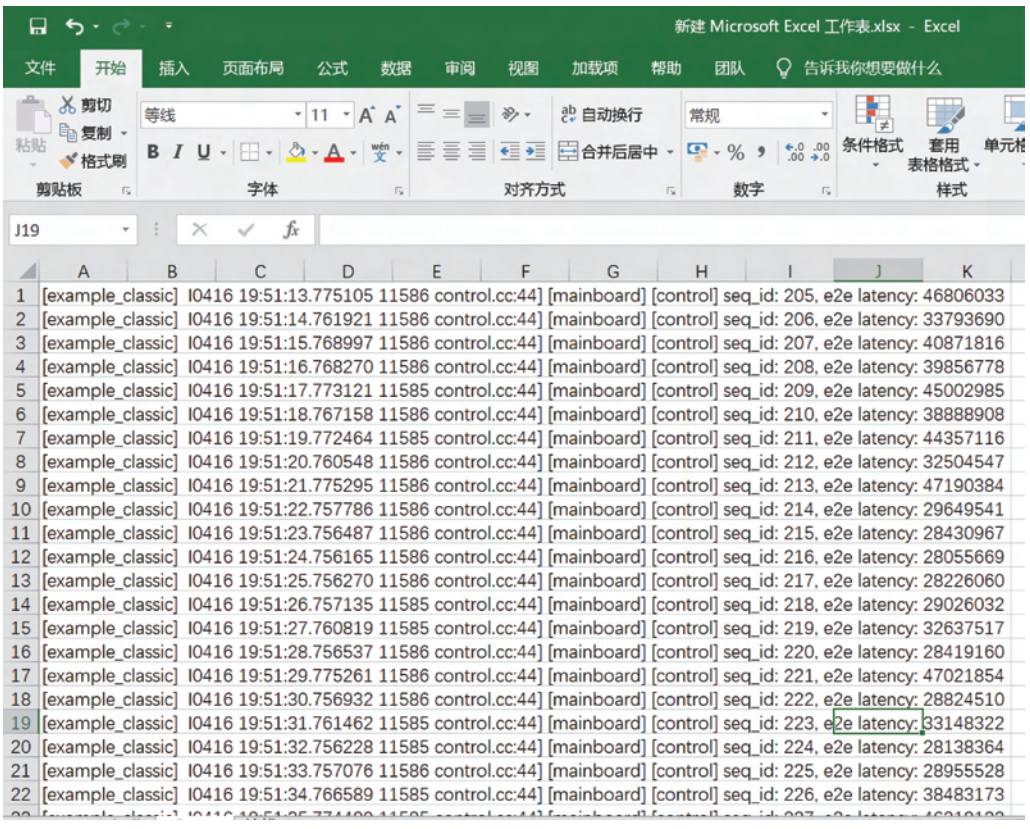
```
bool cal1::Proc(const std::shared_ptr<EChatter>& msg0) {
 AINFO << "Start cal1 component Proc [" << msg0->content() << "]";
 // time_stamp
 auto lt = msg0->lidar_timestamp();
 // sequence id
 auto sid = msg0->seq();

 auto out_msg = std::make_shared<EChatter>();
 if(msg0->content()> 100) {
  out_msg->set_content(1);
 } else {
  out_msg->set_content(0);
 }

 int a = 0;
 for (int i = 0; i < 10000000; ++i) {
  a += 1;
 }

 // proc time stamp
 auto ts = apollo::cyber::Time::Now().ToNanosecond();
 out_msg->set_timestamp(ts);
 out_msg->set_lidar_timestamp(lt);
 out_msg->set_seq(sid);
 cal1_writer_->Write(out_msg);
 AINFO << "cal1: Write drivermsg->"
    << out_msg->content();
 return true;
}
```

在 control.cc 文件的 proc()函数中,同样将时间戳信息 msg0-> lidar_timestamp()赋值给变量 lt,将当前的时间戳信息赋值给变量 ts,于是将 ts 和 lt 的差值赋值给变量 e2e,最终输出时间差信息 e2e。

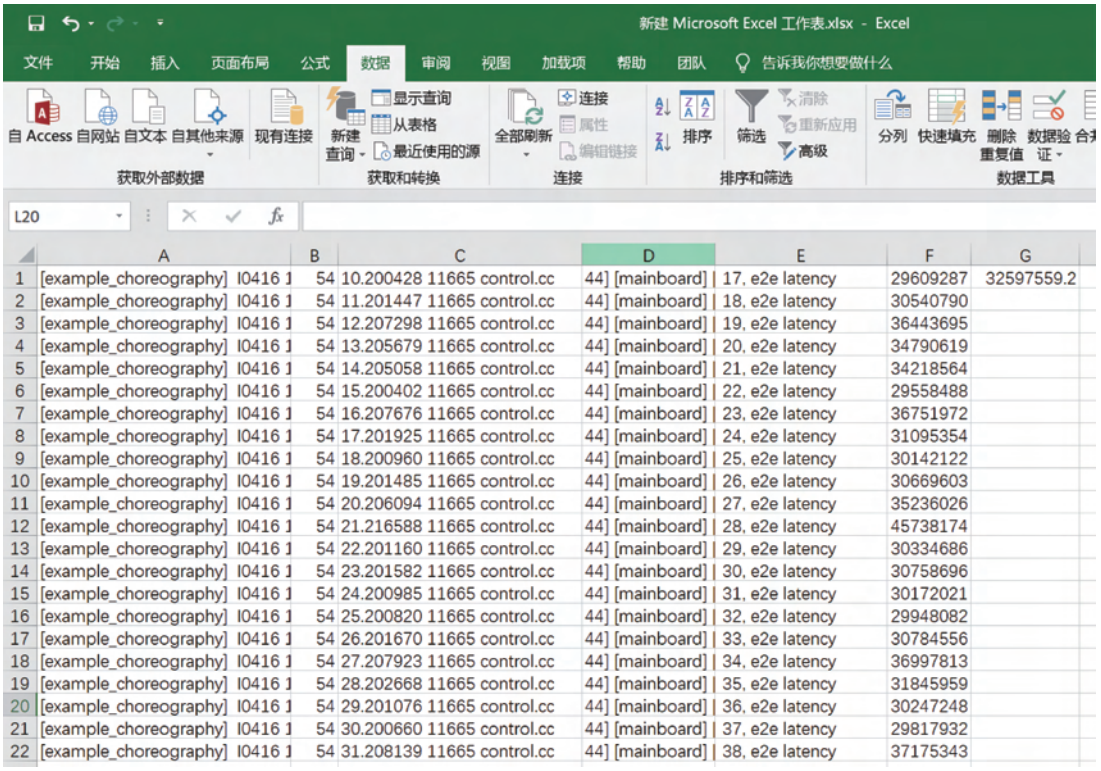
```
bool control::Proc(const std::shared_ptr<EChatter>& msg0,
                   const std::shared_ptr<EChatter>& msg1) {
 AINFO << "Start control component Proc [" << msg0->content() << "] ["
    << msg1->content() << "]";
 auto lt = msg0->lidar_timestamp();
 auto sid = msg0->seq();
 // proc time stamp
 auto ts = apollo::cyber::Time::Now().ToNanosecond();
 // end_to_end latency
 auto e2e = ts - lt;
 AINFO << "[control] seq_id: " << sid
   << ", e2e latency: " << e2e;

```

```
14.  auto out_msg = std::make_shared<EChatter>();
15.  if(msg0->content()>0||msg1->content()>0) {
16.   out_msg->set_content(1);
17.  }
18.  else {
19.   out_msg->set_content(1);
20.  }
21.
22.  out_msg->set_timestamp(ts);
23.  out_msg->set_seq(sid);
24.  out_msg->set_lidar_timestamp(lt);
25.  control_writer_->Write(out_msg);
26.
27.  AINFO << "control: Write drivermsg->"
28.     << out_msg->content();
29.  return true;
30. }
```

将3种调度策略逐一运行一定的时间后，将输出结果保存下来，导入Excel表格中，筛选含有时间差的数据，得到图5.31。然后进行时间差数据的整理，利用Excel的分列命令，将时间差数据分离出来，然后取平均值，如图5.32所示。该平均值便可以代表该调度策略下的时延情况。



■图5.31 得到的时间差序列



	A	B	C	D	E	F	G
1	[example_choreography] I0416 1	54	10.200428 11665 control.cc	44] [mainboard] | 17. e2e latency		29609287	32597559.2
2	[example_choreography] I0416 1	54	11.201447 11665 control.cc	44] [mainboard] | 18. e2e latency		30540790	
3	[example_choreography] I0416 1	54	12.207298 11665 control.cc	44] [mainboard] | 19. e2e latency		36443695	
4	[example_choreography] I0416 1	54	13.205679 11665 control.cc	44] [mainboard] | 20. e2e latency		34790619	
5	[example_choreography] I0416 1	54	14.205058 11665 control.cc	44] [mainboard] | 21. e2e latency		34218564	
6	[example_choreography] I0416 1	54	15.200402 11665 control.cc	44] [mainboard] | 22. e2e latency		29558488	
7	[example_choreography] I0416 1	54	16.207676 11665 control.cc	44] [mainboard] | 23. e2e latency		36751972	
8	[example_choreography] I0416 1	54	17.201925 11665 control.cc	44] [mainboard] | 24. e2e latency		31095354	
9	[example_choreography] I0416 1	54	18.200960 11665 control.cc	44] [mainboard] | 25. e2e latency		30142122	
10	[example_choreography] I0416 1	54	19.201485 11665 control.cc	44] [mainboard] | 26. e2e latency		30669603	
11	[example_choreography] I0416 1	54	20.206094 11665 control.cc	44] [mainboard] | 27. e2e latency		35236026	
12	[example_choreography] I0416 1	54	21.216588 11665 control.cc	44] [mainboard] | 28. e2e latency		45738174	
13	[example_choreography] I0416 1	54	22.201160 11665 control.cc	44] [mainboard] | 29. e2e latency		30334686	
14	[example_choreography] I0416 1	54	23.201582 11665 control.cc	44] [mainboard] | 30. e2e latency		30758696	
15	[example_choreography] I0416 1	54	24.200985 11665 control.cc	44] [mainboard] | 31. e2e latency		30172021	
16	[example_choreography] I0416 1	54	25.200820 11665 control.cc	44] [mainboard] | 32. e2e latency		29948082	
17	[example_choreography] I0416 1	54	26.201670 11665 control.cc	44] [mainboard] | 33. e2e latency		30784556	
18	[example_choreography] I0416 1	54	27.207923 11665 control.cc	44] [mainboard] | 34. e2e latency		36997813	
19	[example_choreography] I0416 1	54	28.202668 11665 control.cc	44] [mainboard] | 35. e2e latency		31845959	
20	[example_choreography] I0416 1	54	29.201076 11665 control.cc	44] [mainboard] | 36. e2e latency		30247248	
21	[example_choreography] I0416 1	54	30.200660 11665 control.cc	44] [mainboard] | 37. e2e latency		29817932	
22	[example_choreography] I0416 1	54	31.208139 11665 control.cc	44] [mainboard] | 38. e2e latency		37175343	

■图 5.32 分列后的数据

经过分析处理，经典的调度策略的时间延时为 33 368 142.84，编排的调度策略时间延时为 32 597 559.2，无编排的调度策略的时间延时为 34 556 517.8，可以看出经过合理编排的调度策略耗时最少，经典的调度策略次之，没有任何编排的调度策略耗时最多。

5.4 本章小结

本章详细介绍了自动驾驶汽车的软件计算框架。现在业界主流的方式是使用面向机器人系统开发的开源系统 ROS(Robert Operation System)，大量开发者和商业公司基于 ROS 做了无数升级工作，以使得 ROS 系统更适合自动驾驶汽车使用，Apollo 3.0 在版本之前也是使用 ROS 作为计算框架。但因为 ROS 系统在自动驾驶领域中应用的种种缺陷，百度 Apollo 团队转而开发自有的、面向自动驾驶系统的计算框架 Cyber RT，并随 Apollo 3.5 版本一同发布。可以说，这个框架组件是 Apollo 3.5 中最重大的升级，它使得 Apollo 平台摆脱了 ROS 系统的固有缺陷，使 Apollo 能够更加专注自动驾驶技术本身。

为帮助读者更好地理解 Cyber RT 计算框架，本章不仅详细介绍了其工作原理，还附带一些可独立运行的代码实例。有兴趣的读者可自行运行调试这些实例，以更深入地用好这个自动驾驶开发中的利器。

参考文献

[1] ROS wiki[EB/OL]. [2019-04-06]. http://wiki.ros.org.

[2] 恩里克·费尔南德斯,等. ROS机器人程序设计[M]. 刘锦涛,译. 北京: 机械工业出版社,2016.

[3] Apollo Cyber RT framework[EB/OL]. [2019-04-06]. http://apollo.auto/cyber_cn.html.

[4] Cyber RT 协程技术解读[EB/OL]. (2019-04-01)[2019-04-06]. https://mp.weixin.qq.com/s/6LdFTZrTiRZfF_gv1NJhzg.