机 器 人 操 作 系 统

机 电 工 程 学 院

机器人智能制造研究团队

2025 · 秋

课程安排

第　三　章

ROS架构与运行管理

目录

CONTENTS

01

第一节
ROS系统架构

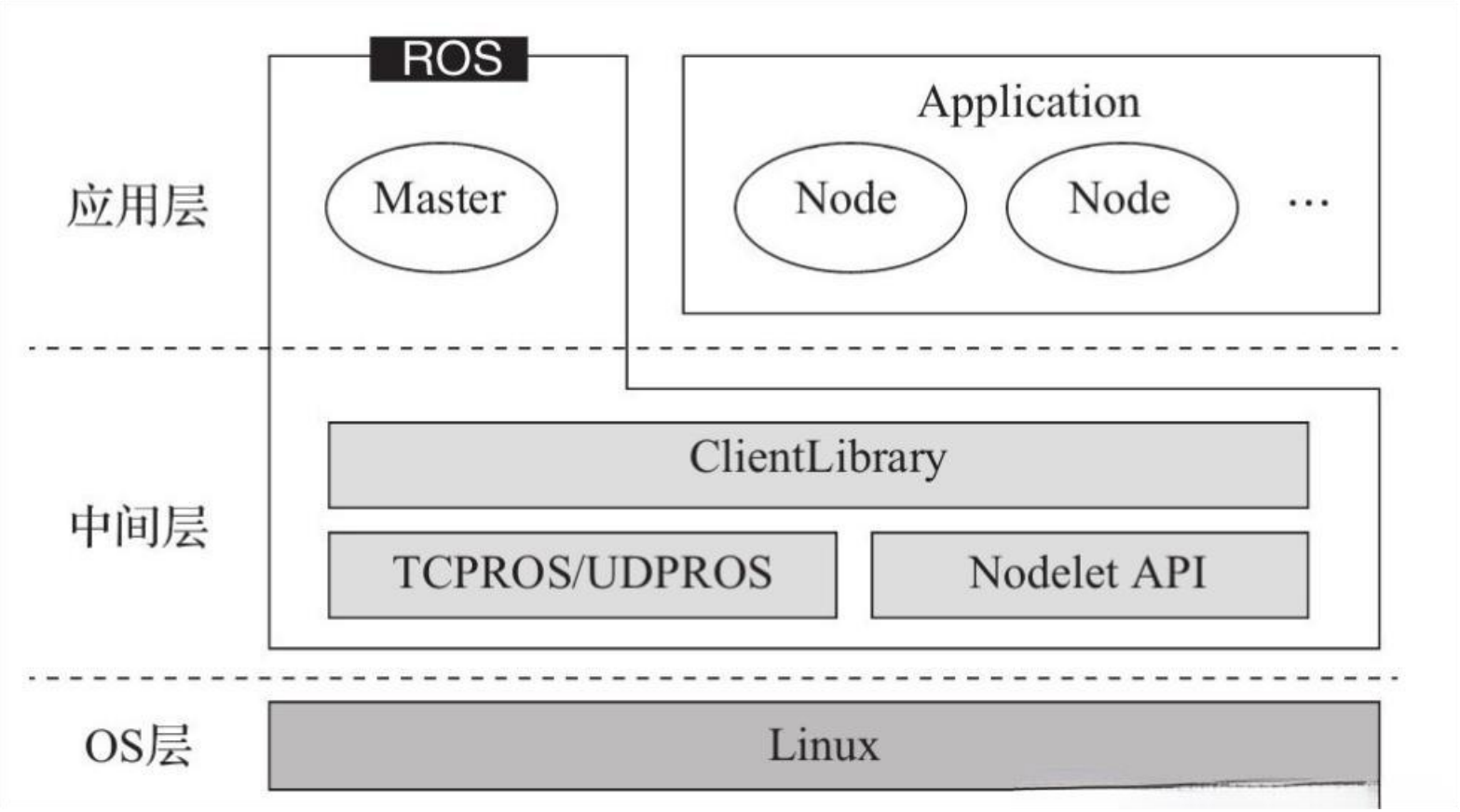
一、从设计者角度描述ROS架构

ROS设计者将ROS表述为“ROS = Plumbing + Tools + Capabilities + Ecosystem”

- Plumbing: 通讯机制(实现ROS不同节点之间的交互)
- Tools :工具软件包(ROS中的开发和调试工具)
- Capabilities :应用功能(ROS中某些功能的集合，比如:导航，SLAM)
- Ecosystem:生态系统(跨地域、跨软件与硬件的ROS联盟)

二、ROS软件层次架构

从软件实现的角度，ROS 的架构可以分为三个层次：OS层，中间层，应用层；



二、ROS软件层次架构

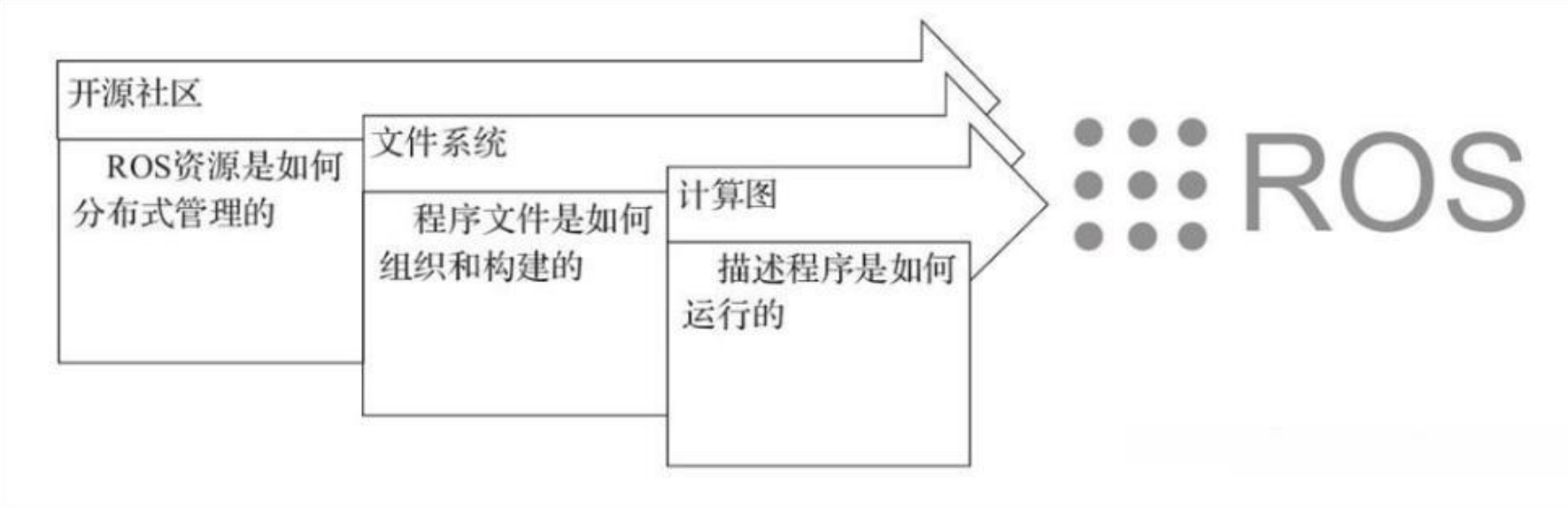
- **OS 层：** ROS 只是元操作系统，需要依托真正意义的操作系统，目前兼容性最好的是 Linux 的 Ubuntu，Mac、Windows 也支持 ROS 的较新版本，OS 层建立在 Linux 操作系统之上；主要提供了硬件抽象、底层驱动等；

- **中间层：** ROS 的中间层进一步封装 OS 层提供的接口，为应用层提供格式统一，模块化的接口。其主要工作为：
  - 把 TCP/UDP 通信封装为 TCPROS/UDPROS，并提供主题通信、服务通信、参数共享等三种通信方式；
  - 额外提供进程内的通信方式 – Nodelet，适用于实时性较高的应用；
  - 在通信的基础之上提供大量机器人开发的库，如数据类型定义，坐标变换，运动控制等

- **应用层：** 主要调用中间层的接口实现各种应用功能。
  - 应用层有一个管理者 Master，负责管理整个系统的运行；
  - ROS 社区共享了大量机器人应用功能包，功能包内的模块都是以节点为单位运行，以 ROS 标准的输入输出作为接口。我们在使用的时候，不需要关注模块内部实现细节，只用明白接口规则即可复用。

三、ROS系统层次架构

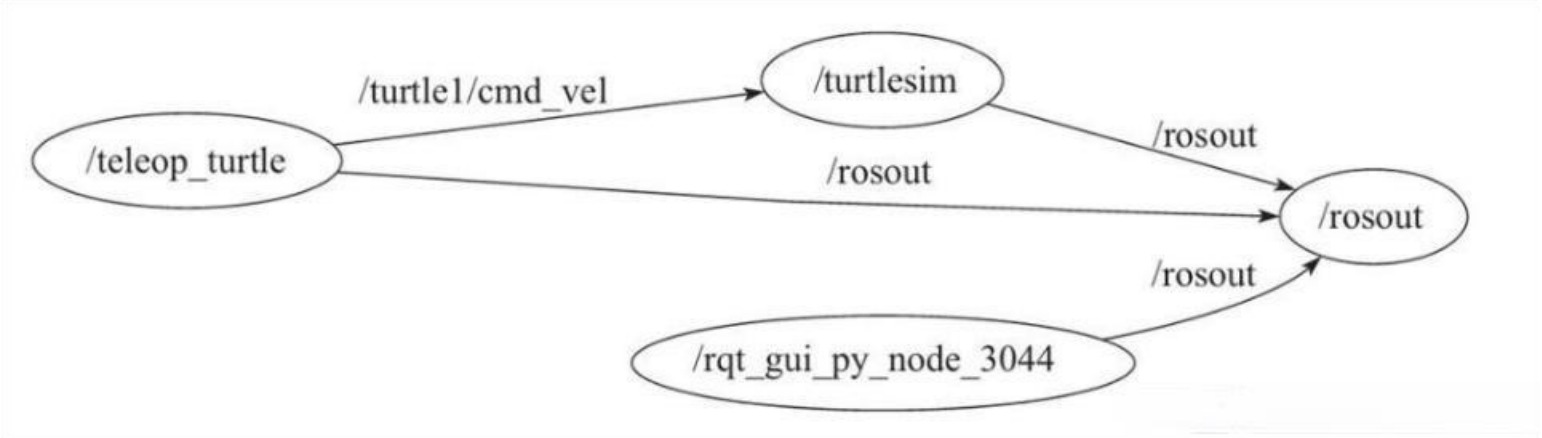
从系统实现的角度，ROS 可以划分为：计算图、文件系统、开源社区三个部分；



三、ROS系统层次架构

计算图

从计算图的结构来看，ROS 系统软件的功能模块是以节点（节点即为进程）为单位独立运行的，节点以拓扑的方式互联，构成了一个系统网络，即为系统的计算图。如下图所示：

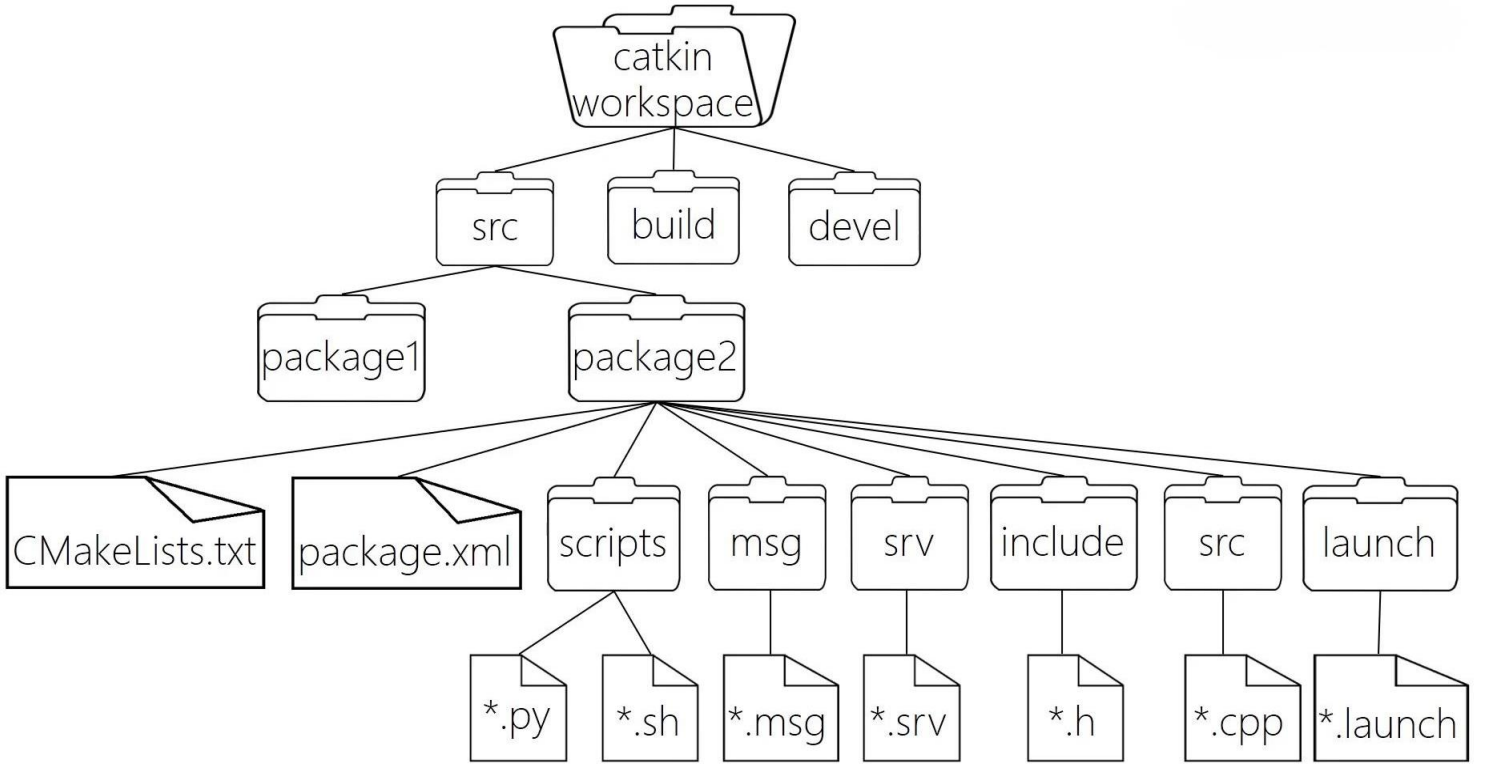


端点表示各个节点，端点之间的边表示节点之间的通信方式。
在计算图中，有几个重要的概念在上一章已做过介绍：消息、话题、服务、节点管理器；

三、ROS系统层次架构

文件系统

ROS文件系统级指的是在硬盘上面查看的ROS源代码的组织形式，如下图所示：



有关文件系统的编译和编写将在下一节详细介绍

四、ROS系统层次架构

开源社区

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

- **发行版（Distribution）** ROS发行版是可以独立安装、带有版本号的一系列综合功能包。

ROS发行版像Linux发行版一样发挥类似的作用。这使得ROS软件安装更加容易，而且能够通过一个软件集合维持一致的版本。

- **软件库（Repository）** ROS依赖于共享开源代码与软件库的网站或主机服务，在这里不同的机构能够发布和分享各自的机器人软件与程序。
- **ROS维基（ROS Wiki）** ROS Wiki是用于记录有关ROS系统信息的主要论坛。任何人都可以注册账户、贡献自己的文件、提供更正或更新、编写教程以及其他行为。网址是http://wiki.ros.org/。
- **Bug提交系统（Bug Ticket System）**如果你发现问题或者想提出一个新功能，ROS提供这个资源去做这些。
- **邮件列表（Mailing list）** ROS用户邮件列表是关于ROS的主要交流渠道，能够像论坛一样交流从ROS软件更新到ROS软件使用中的各种疑问或信息。网址是http://lists.ros.org/。
- **ROS问答（ROS Answer）**用户可以使用这个资源去提问题。网址是https://answers.ros.org/questions/。
- **博客（Blog）**你可以看到定期更新、照片和新闻。网址是https://www.ros.org/news/，不过博客系统已经退休，ROS社区取而代之，网址是https://discourse.ros.org/

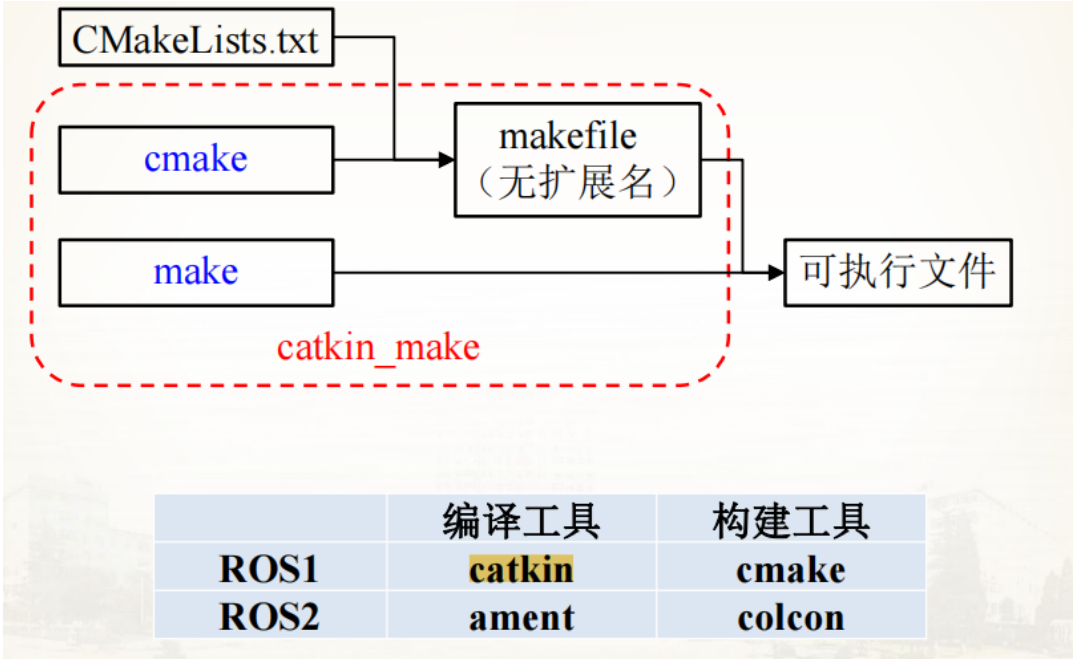
02

第二节
ROS文件系统

一、Catkin编译系统

ROS使用自己独特的编译系统对ROS程序包进行编译。

- 编译工具catkin（使用cmake作为构建工具）。ROS早期版本使用rosbuild，而ROS2使用的是colcon。
- cmake可根据CMakeLists.txt文件为不同的平台生成makefile文件，然后使用工具make编译出可执行文件。
- cmake 是一个集软件构建、测试、打包于一身的软件。具有跨平台、开源的特点，使用与平台和编译器独立的配置文件来对软件编译过程进行控制。
- catkin_make本质上是对cmake和make的封装。它规范了编译配置路径和生成文件路径等。



	编译工具	构建工具
ROS1	catkin	cmake
ROS2	ament	colcon

一、Catkin编译系统

✓ Catkin工作原理

catkin编译的工作流程如下：

- 首先在工作空间catkin_ws/src/下递归的查找其中每一个ROS的package。
- package中会有package.xml和CMakeLists.txt文件，Catkin(CMake)编译系统依据CMakeLists.txt文件，从而生成makefiles(放在catkin_ws/build/)。
- 然后make刚刚生成的makefiles等文件，编译链接生成可执行文件(放在catkin_ws/devel)。

也就是说，Catkin就是将cmake与make指令做了一个封装从而完成整个编译过程的工具。

✓ 使用catkin_make进行编译

要用catkin编译一个工程或软件包，只需要用catkin_make指令。一般当我们写完代码，执行一次catkin_make进行编译，调用系统自动完成编译和链接过程，构建生成目标文件

```
$ cd ~/catkin_ws #回到工作空间,catkin_make必须在工作空间下执行
$ catkin_make    #开始编译
$ source ~/catkin_ws/devel/setup.bash #刷新环境
```

注意: catkin编译之前需要回到工作空间目录，catkin_make在其他路径下编译不会成功。编译完成后，如果有新的目标文件产生（原来没有），那么一般紧跟着要source刷新环境，使得系统能够找到刚才编译生成的ROS可执行文件。

二、Catkin工作空间

Catkin工作空间是创建、修改、编译catkin软件包的目录。catkin的工作空间，直观的形容就是一个仓库，里面装载着ROS的各种项目工程，便于系统组织管理调用。

✓ **初始化catkin工作空间**

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make #初始化工作空间
```

✓ **catkin工作空间结构**

在工作空间下用tree命令，显示文件结构。

```
$ cd ~/catkin_ws
$ sudo apt install tree
$ tree
```

结果为:

```
— build
|   ├— catkin
|   |   └— catkin_generated
|   |       └— version
|   |           └— package.cmake
|   ├—
......

|   ├— catkin_make.cache
|   ├— CMakeCache.txt
|   ├— CMakeFiles
|   |   ├—
......

├— devel
|   ├— env.sh
|   ├— lib
|   ├— setup.bash
|   ├— setup.sh
|   ├— _setup_util.py
|   └— setup.zsh
└— src
└— CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake
```

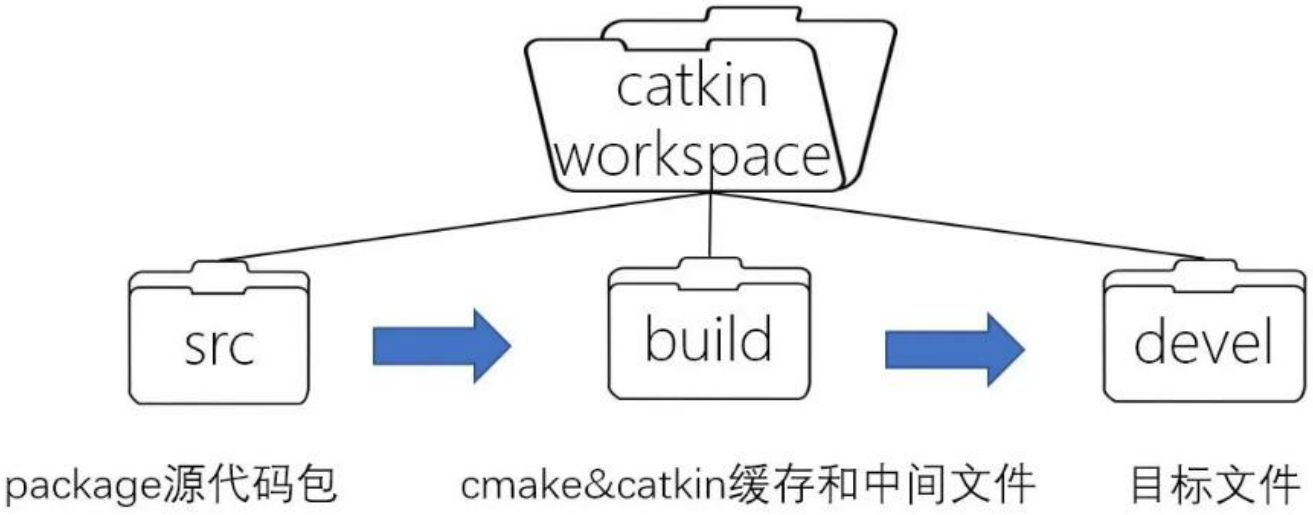
二、Catkin工作空间

通过tree命令可以看到catkin工作空间的结构,它包括了src、build、devel三个路径。它们的具体作用如下:

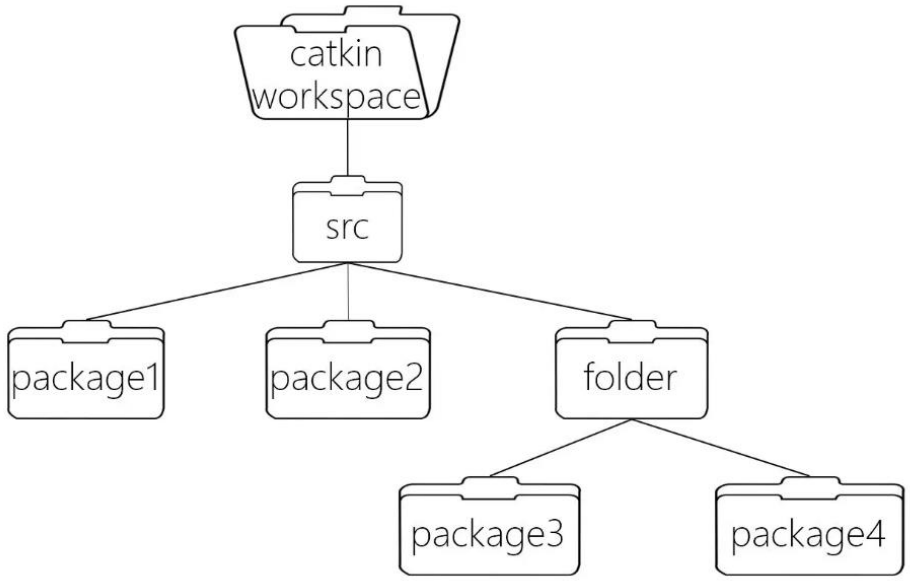
src/: ROS的catkin软件包（源代码包）

build/: catkin（CMake）的缓存信息和中间文件

devel/: 生成的目标文件（包括头文件，动态链接库，静态链接库，可执行文件等）、环境变量



在编译时，catkin编译系统会递归的查找和编译src/下的每一个源代码包。因此也可以把几个源代码包放到同一个文件夹下，如下图所示:



三、Package功能包

package是catkin编译的基本单元，我们调用catkin_make编译的对象就是一个个ROS的package，任何ROS程序只有组织成package才能编译。所以任何ROS的代码无论是C++还是Python都要放到package中，这样才能正常的编译和运行。

一个package可以编译出来多个目标文件（ROS可执行程序、动态静态库、头文件等等）。

✓ package结构

通常ROS文件组织都是包含以下的文件和路径，这是约定俗成的命名习惯，建议遵守。以上文件中，只有CMakeLists.txt和package.xml是必须的，其余路径根据软件包是否需要来决定。

- CMakeLists.txt: 定义package的包名、依赖、源文件、目标文件等编译规则，是package不可少的成分
- package.xml: 描述package的包名、版本号、作者、依赖等信息，是package不可少的成分
- src/: 存放ROS的源代码，包括C++的源码和(.cpp)以及Python的module(.py)
- include/: 存放C++源码对应的头文件
- scripts/: 存放可执行脚本，例如shell脚本(.sh)、Python脚本(.py)
- msg/: 存放自定义格式的消息(.msg)
- srv/: 存放自定义格式的服务(.srv)
- models/: 存放机器人或仿真场景的3D模型(.sda, .stl, .dae等)
- urdf/: 存放机器人的模型描述(.urdf或.xacro)
- launch/: 存放launch文件(.launch或.xml)

三、Package功能包

✓ package的创建

创建一个package需要在 `catkin_ws/src` 下,用到 `catkin_create_pkg` 命令，用法是:

`catkin_create_pkg package depends`

其中package是包名，depends是依赖的包名，可以依赖多个软件包。

例如，新建一个package叫做 `test_pkg` ,依赖roscpp、rospy、std_msgs(常用依赖)。

```
$ catkin_create_pkg test_pkg roscpp rospy std_msgs
```

这样就会在当前路径下新建 `test_pkg` 软件包，包括:

```
├── CMakeLists.txt
├── include
│   └── test_pkg
├── package.xml
└── src
```

`catkin_create_pkg` 帮你完成了软件包的初始化，填充好了 `CMakeLists.txt` 和 `package.xml` ，并且将依赖项填进了这两个文件中。

三、Package功能包

✓ package相关命令

rospack是对package管理的工具，命令的用法如下：

rostopic命令	作用
`rospack help`	显示rospack的用法
`rospack list`	列出本机所有package
`rospack depends [package]`	显示package的依赖包
`rospack find [package]`	定位某个package
`rospack profile`	刷新所有package的位置记录

roscd命令类似与Linux系统的cd，改进之处在于roscd可以直接cd到ROS的软件包。

rostopic命令	作用
`roscd [pacakge]`	cd到ROS package所在路径

rosls也可以视为Linux指令ls的改进版，可以直接lsROS软件包的内容。

rosls命令	作用
`rosls [pacakge]`	列出pacakge下的文件

三、Package功能包

✓ package相关命令

rosdep是用于管理ROS package依赖项的命令行工具，用法如下：

rosdep命令	作用
`rosdep check [pacakge]`	检查package的依赖是否满足
`rosdep install [pacakge]`	安装pacakge的依赖
`rosdep db`	生成和显示依赖数据库
`rosdep init`	初始化/etc/ros/rosdep中的源
`rosdep keys`	检查package的依赖是否满足
`rosdep update`	更新本地的rosdep数据库

一个较常使用的命令是rosdep install --from-paths src --ignore-src --rosdistro=melodic -y,用于安装工作空间中src路径下所有package的依赖项（由pacakge.xml文件指定）。

四、CMakeLists.txt

✓ CMakeLists.txt作用

CMakeLists.txt原本是Cmake编译系统的规则文件，而Catkin编译系统基本沿用了CMake的编译风格，只是针对ROS工程添加了一些宏定义。所以在写法上，catkin的CMakeLists.txt与CMake的基本一致。

这个文件直接规定了这个package要依赖哪些package，要编译生成哪些目标，如何编译等等流程。所以CMakeLists.txt非常重要，它指定了由源码到目标文件的规则，catkin编译系统在工作时首先会找到每个package下的CMakeLists.txt，然后按照规则来编译构建。

✓ CMakeLists.txt写法

```
cmake_minimum_required()        #指定catkin最低版本
project()                       #指定软件包的名称
find_package()                  #指定编译时需要的依赖项
add_message_files()/add_service_files()/add_action_files()
                                #添加消息文件/服务文件/动作文件
generate_messages()             #生成消息、服务、动作
catkin_package()                #指定catkin信息给编译系统生成Cmake文件
add_library()/add_executable()  #指定生成库文件、可执行文件
target_link_libraries()         #指定可执行文件去链接哪些库
catkin_add_gtest()              #添加测试单元
install()                       #生成可安装目标
```

五、package.xml

✓ **package.xml作用**

pacakge.xml包含了package的名称、版本号、内容描述、维护人员、软件许可、编译构建工具、编译依赖、运行依赖等信息。

实际上rospack find、rosdep等命令之所以能快速定位和分析出package的依赖项信息，就是直接读取了每一个pacakge中的package.xml文件。它为用户提供了快速了解一个pacakge的渠道。

✓ **package.xml写法**

```
<package> 根标记文件
    <name> 程序包名称
    <version> 版本号
    <description> 基本内容描述
    <maintainer> 维护者
    <license> 软件许可证          #以上为必选标签！
    <buildtool_depend> 编译构建工具， 通常为catkin
    <depend> 指定依赖项为编译、 导出、 运行需要的依赖， 最常用
    <build_depend> 编译依赖项
    <build_export_depend> 导出依赖项
    <exec_depend> 运行依赖项
    <test_depend> 测试用例依赖项
    <doc_depend> 文档依赖项
</package> 根标记文件
```

六、launch文件

✓ launch文件简介

机器人是一个系统工程，通常一个机器人运行操作时要开启很多个node，对于一个复杂的机器人的启动操作应该怎么做呢?

我们并不需要每个节点依次进行rosrun，ROS为我们提供了一个命令能一次性启动master和多个node。该命令是:

$ roslaunch pkg_name file_name.launch

roslaunch命令首先会自动进行检测系统的roscore有没有运行，也即是确认节点管理器是否在运行状态中，如果master没有启动，那么roslaunch就会首先启动master，然后再按照launch的规则执行。launch文件里已经配置好了启动的规则。

所以roslaunch就像是一个启动工具，能够一次性把多个节点按照我们预先的配置启动起来，减少我们在终端中一条条输入指令的麻烦。

六、launch文件

- ✓ **launch文件使用**

launch文件的作用：

简化节点的配置与启动，提高ROS程序的启动效率。

案例：

以乌龟控制案例 turtlesim 为例

1.新建launch文件

在功能包下添加 launch目录，目录下新建 xxxx.launch 文件，编辑 launch 文件

```
<launch>
    <node pkg="turtlesim" type="turtlesim_node"     name="myTurtle" output="screen" />
    <node pkg="turtlesim" type="turtle_teleop_key"  name="myTurtleContro" output="screen" />
</launch>
```

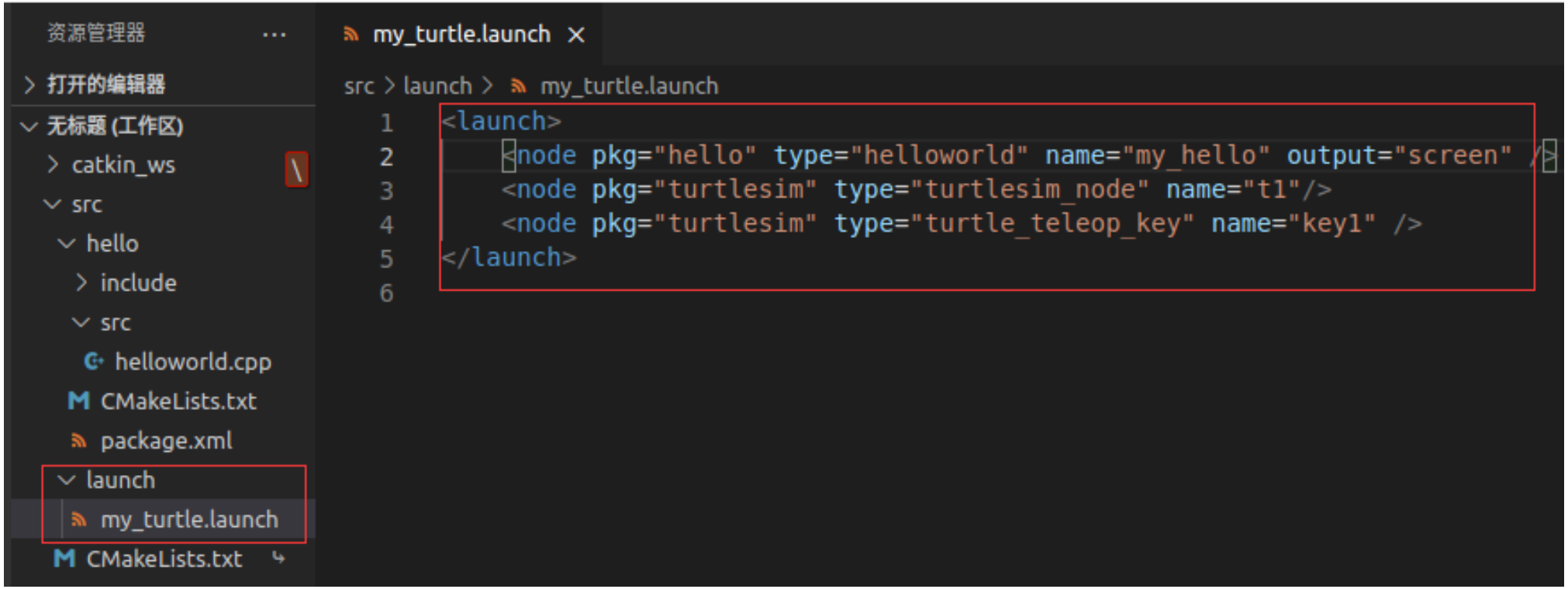
2.调用launch文件

$ roslaunch 包名 xxx.launch

六、launch文件

✓ **编写launch文件**

在src中添加launch文件夹和launch文件



- node ---> 包含的某个节点
- pkg -----> 程序包
- name ---> 为节点命名
- type ---->指定了启动节点的可执行文件或脚本。
- output--> 设置日志的输出目标

六、launch文件

- ✓ **<launch>**

<launch>标签是所有 launch 文件的根标签，充当其他标签的容器，所有其它标签都是launch的子级。

- ✓ **<node>**

<node>标签用于指定 ROS 节点，是最常见的标签，需要注意的是: roslaunch 命令不能保证按照 node 的声明顺序来启动节点(节点的启动是多进程的)

1.属性

pkg=“包名”： 节点所属的包

type=“nodeType”： 节点类型(与之相同名称的可执行文件)

name=“nodeName”： 节点名称(在 ROS 网络拓扑中节点的名称)

args=“xxx xxx xxx” (可选)：将参数传递给节点

machine=“机器名”：在指定机器上启动节点

respawn=“true | false” (可选)：如果节点退出，是否自动重启

六、launch文件

✓ **<node>**

1.属性

respawn_delay=“N”(可选)：如果 respawn 为 true, 那么延迟 N 秒后启动节点

required=“true | false”(可选)：若为 true,如果该节点退出，将杀死整个 roslaunch

ns=“xxx”(可选)：在指定命名空间 xxx 中启动节点

clear_params=“true | false”(可选)：在启动前，删除节点的私有空间的所有参数

output=“log | screen”(可选)：日志发送目标，可以设置为 log 日志文件或 screen 屏幕，默认是 log

2.子级标签

env 环境变量设置

remap 重映射节点名称

rosparam 参数设置

param 参数设置

六、launch文件

- ✓ **<include>**

  <include>标签用于将另一个 xml 格式的 launch 文件导入到当前文件

  1.属性

  file=“$(find 包名)/xxx/xxx.launch”：要包含的文件路径

  ns=“xxx”（可选）：在指定命名空间导入文件

  2.子级标签

  env 环境变量设置

  arg 将参数传递给被包含的文件

- ✓ **<remap>**

  用于话题重命名，无子级标签

  1.属性

  from=“xxx”：原始话题名称

  to=“yyy”：目标名称

六、launch文件

✓ **<param>**

<param>标签主要用于在参数服务器上设置参数，参数源可以在标签中通过 value 指定，也可以通过外部文件加载，在<node>标签中时，相当于私有命名空间。无子级标签。

1.属性

name=“命名空间/参数名“：参数名称，可以包含命名空间

value=“xxx”（可选）：定义参数值，如果此处省略，必须指定外部文件作为参数源

type=“str | int | double | bool | yaml”（可选）：

指定参数类型，如果未指定，roslaunch 会尝试确定参数类型，规则如下：

- 如果包含'.'的数字解析未浮点型，否则为整型
- "true" 和 "false" 是 bool 值(不区分大小写)
- 其他是字符串

六、launch文件

- ✓ **<rosparam>**

<rosparam>标签可以从 YAML 文件导入参数，或将参数导出到 YAML 文件，也可以用来删除参数，<rosparam>标签在<node>标签中时被视为私有。

1.属性

command=“load | dump | delete”（可选，默认 load）：加载、导出或删除参数

file=“$(find xxxxx)/xxx/yyy….”：加载或导出到的 yaml 文件

param=“参数名称”

ns=“命名空间”（可选）

无子级标签

六、launch文件

- **<group>**

<group>标签可以对节点分组，具有 ns 属性，可以让节点归属某个命名空间

1.属性

ns="名称空间" (可选)

clear_params="true | false" (可选)

启动前，是否删除组名称空间的所有参数(慎用....此功能危险)

2.子级标签 除了launch 标签外的其他标签

- **<arg>**

<arg>标签是用于动态传参，类似于函数的参数，可以增强launch文件的灵活性

1.属性

name="参数名称"

default="默认值" (可选)

value=“数值” (可选)：不可以与 default 并存

doc=“描述”：参数说明

03

第三节
ROS运行管理

为什么需要运行管理?

ROS是多进程(节点)的分布式框架，一个完整的ROS系统实现:

- 可能包含多台主机;
- 每台主机上又有多个工作空间(workspace);
- 每个的工作空间中又包含多个功能包(package);
- 每个功能包又包含多个节点(Node)，不同的节点都有自己的节点名称;
- 每个节点可能还会设置一个或多个话题(topic)...

在多级层深的ROS系统中，其实现与维护可能会出现一些问题，比如:

- 如何关联不同的功能包，繁多的ROS节点应该如何启动?
- 功能包、节点、话题、参数重名时应该如何处理?
- 不同主机上的节点如何通信?

一、Metapackage

✓ metapackage概念

场景：完成ROS中一个系统性的功能，可能涉及到多个功能包，比如机器人导航模块，该模块下有地图、定位、路径规划...等不同的子级功能包。那么调用者安装该模块时，需要逐一的安装每一个功能包吗？

显而易见的，逐一安装功能包的效率低下，在ROS中，提供了一种方式可以将不同的功能包打包成一个功能包，当安装某个功能模块时，直接调用打包后的功能包即可，该包又称之为元功能包(metapackage)。

MetaPackage是Linux的一个文件管理系统的概念。是ROS中的一个虚包，里面没有实质性的内容，但是它依赖了其他的软件包，通过这种方法可以把其他包组合起来，我们可以认为它是一本书的目录索引，告诉我们这个包集合中有哪些子包，并且该去哪里下载。

一、Metapackage

✓ metapackage实现

首先：新建一个功能包

然后：修改package.xml ,内 容如 下

<run_depend> 所有要集成的依赖包 </run_depend>

最后：修改 CMakeLists.txt,内 容如 下:

```
cmake_minimum_required(VERSION 3.0.2)
project(demo)
find_package(catkin REQUIRED)
catkin_metapackage()
```

metapacakge中的以上两个文件和普通pacakge不同点是:

- CMakeLists.txt:加入了catkin_metapackage()宏，指定本软件包为一个metapacakge。
- package.xml:<run_depend>标签将所有软件包列为依赖项，标签中添加标签声明。

metapacakge在我们实际开发一个大工程时可能有用。

二、节点、话题、参数重名问题

✓ 节点重名

- 问题介绍：在ROS系统中节点是最为基本的概念，在创建节点时，例如使用C++语言进行初始化，需要通过指定API定义节点名称，而如果存在重名节点，那么在调用时就会出现问题；
- 解决策略：（1）使用命名空间，添加前缀 （2）名称重映射，重新起名
  对于上述两种策略，实现途径有三种方法：rosrun命令、launch文件与编程实现

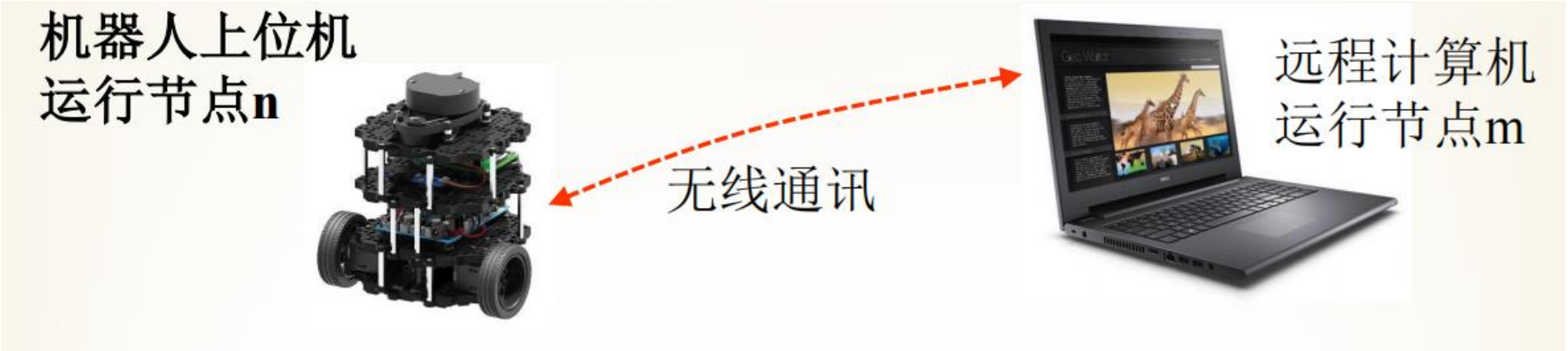
✓ 话题重名

- 问题介绍：相比于节点，ROS中的话题当然也存在重名问题，但是会更加复杂一些，因为在实际的应用中，话题名的设置相对灵活，需要灵活修改。
- 解决策略：与节点重名相同，基本的策略仍旧是重映射和前缀两种方法，区别在于这里的前缀种类很多，可以分为以下三种：
  全局前缀：参考ROS系统，与命名空间平级
  相对前缀：参考命名空间，与节点名称平级
  私有前缀：参考节点名称，位于节点名称之下

✓ 参数重名

- 参数重名的处理，没有重映射实现，为了尽量的避免参数重名，都是使用为参数名添加前缀的方式，实现类似于话题名称，有全局、相对、和私有三种类型之分。

三、分布式通信



ROS是一个分布式计算环境。一个运行中的ROS系统可以包含分布在多台计算机上多个节点。根据系统的配置方式，任何节点可能随时需要与任何其他节点进行通信。

远程计算机对机器人控制的一般过程：使用网络中某台计算机运行节点管理器（由ROS_MASTER_URI变量指定），并使用该计算机启动roscore进程。网络中其它远程计算机将各自的IP地址作为ROS_IP地址，但均将ROS_MASTER_URI变量作为机器人上位机的IP地址。

机器人通常需要在不受远程计算机干扰的情况下自主移动，因此，常选择机器人上位机作为节点管理器。