记得倒回去看

1.5 ROS的架构

1. 先创建一个工作空间

mkdir -p 自定义空间名称/src

进入 cd 自定义空间名称

catkin_make 编译命令

2. 再创建一个功能包

cd src

catkin_create_pkg 自定义ROS包名 roscpp rospy std_msgs

3 编辑源文件

4. 编辑配置文件

5. 编译并执行

- 无".vscode"文件→点"运行""添加配置"

- 依赖"roscpp rospy std_msgs" *非必须

? - 配置文件 - 节点名称

CMakeList.txt

Line 136

- add_executable(${PROJECT_NAME}_node src/

hello_vscode_node.cpp)

给文件映射一个名称(一般和源文件同名)

源文件名

Line 149

- target_link_libraries(${PROJECT_NAME}_node

${catkin_LIBRARIES}

)

与映射名称一致

- vscode 一种集成开发IDE

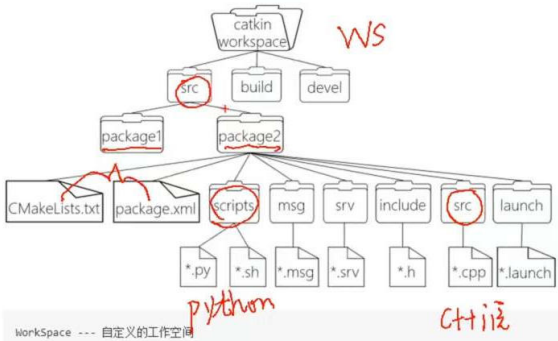
ROS社区

- 软件库:各种软件收集的一个网址 Github Gitee

- ROS Wiki:ROS相关论坛 http://wiki.ros.org/

- ROS问答 1.5文件中有 http://answers.ros.org/questions/

ROS文件系统级指的是在硬盘上ROS源代码的组织形式，其结构大致可以如下图所示：



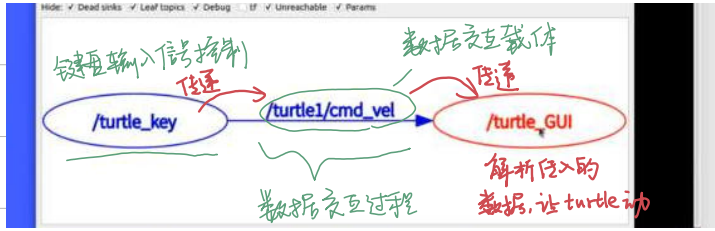

```
WorkSpace --- 自定义的工作空间

    |--- build:编译空间，用于存放CMake和catkin的缓存信息、配置信息和其他中间文件。

    |--- devel:开发空间，用于存放编译后生成的目标文件，包括头文件、动态&静态链接库、可执行文件等。

    |--- src: 源码
```

```
|-- package: 功能包(ROS基本单元)包含多个节点、库与配置文件，包名所有字母小写，只能由字母、数字与

    |-- CMakeLists.txt 配置编译规则，比如源文件、依赖项、目标文件

    |-- package.xml 包信息，比如:包名、版本、作者、依赖项...(以前版本是 manifest.xml)

    |-- scripts 存储python文件

    |-- src 存储C++源文件

    |-- include 头文件

    |-- msg 消息通信格式文件

    |-- srv 服务通信格式文件

    |-- action 动作格式文件

    |-- launch 可一次性运行多个节点

    |-- config 配置信息

|-- CMakeLists.txt: 编译的基本配置
```

- 计算图：强调上ROS程序的存储结构是静态的，ros程序运行之后，不同的节点间是错综复杂的，ROS中提供了一个工具 rqt_graph

看不懂，尽力了。

- rqt_graph 能够创建一个显示当前系统运行情况的动态图形。ROS分布式系统中不同进程需要进行数据交互，计算图可以以点对点的网络形式表现数据交互过程。

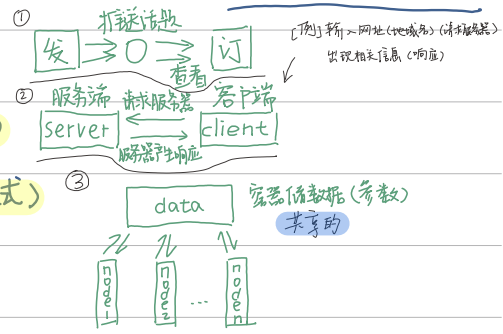



↑

计算图以图文形式显示各节点的关系，清晰明了

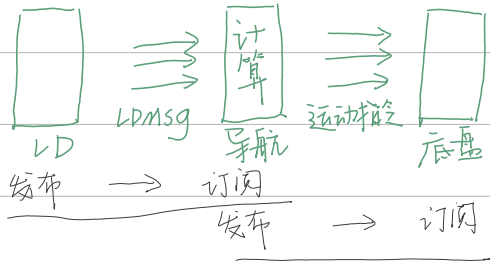
- ROS通信机制　为了解耦合，ROS每个功能点都是一个单独的进程，每个进程都是独立运行的（分布式框架），其优点在于进程可分布于不同主机，分散计算压力。不过随之也有一个问题　不同进程是如何通信的？

- 话题通信（发布订阅模式）
- 服务通信（请求响应模式）
- 参数服务器（参数共享模式）



# 话题通信

以"雷达-导航-底盘"为例




雷达、摄像头、GPS...

话题通信适用于不断更新的数据传输相关的应用场景

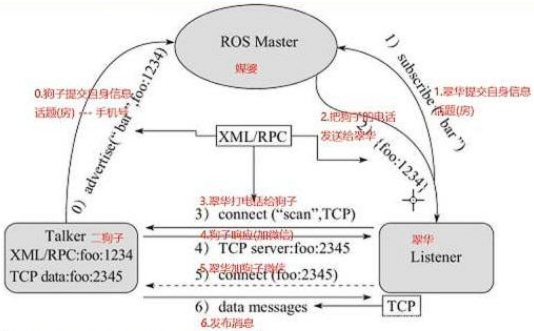
**概念**

以发布订阅的方式实现不同节点之间数据交互的通信模式。

**作用**

用于不断更新的、少逻辑处理的数据传输场景。

# 话题通信的理论模型




话题通信应用时的关注点:
0.大部分实现已经被封装了
1.话题设置
2.关注发布者实现
3.关注订阅者实现
4.关注消息载体

角色 ---> 流程 ---> 注意

角色
1.master ----> 管理者 (媒婆) 管理匹配话题
2.talker -----> 发布者(男方)
3.listener ----> 订阅者 (女方)

流程
master 可以根据话题建立发布者和订阅者之间的连接。

注意:
1.使用的协议由RPC 和 TCP;
2.步骤0和步骤1没有顺序关系;
3.talker 和 listener 都可以存在多个;
4.talker 和 listener 建立连接后, master 就可以关闭了;
5.上述实现流程已经封装了, 以后直接调用即可。

---

1、句柄 把手, 控制节点

2、出现"Failed to contact master"

解决方法: 开一个终端运行 roscore

3、"rostopic echo -" 无反应 ⇒开终端运行 roscore rosrun

4、ss str()、c_str(): 把字符串类 stringstream 转化为 string 再转化为 C 语言的 string

虚左大佬给的代码里 msg.data = ss.str();

→ msg.data.c_str() 效果与之一致

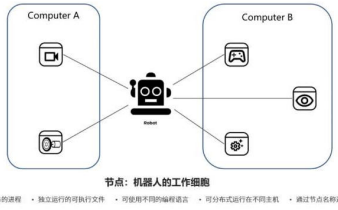
5、setlocale(LC_ALL,""); 用于解决乱码问题

# 节点

## ROS：节点是什么

- 机器人是各种功能的综合体，每一项功能就像机器人的一个工作细胞，众多细胞通过一些机制连接到一起，成为了一个机器人整体。
- 在ROS中，我们给这些“细胞”取了一个名字，那就是**节点**。
  - 在ROS中，最小的进程单元就是节点
- 一个软件包里可以有多个可执行文件，可执行文件在运行之后就形成了一股进程，这个进程在ROS中就叫做**节点**
  - 从程序的角度来说，node就是一个可执行文件被执行，加载到了内存中
  - 从功能的角度来说，每一个节点也是只负责一个单独的模块化的功能
    - 由于机器人的功能模块非常复杂，我们往往不会把所有功能都集中到一个node上，而会采用分布式的方式，把鸡蛋放到不同的篮子里
    - （比如一个节点负责控制车轮转动，一个节点负责从激光雷达获取数据、一个节点负责处理激光雷达的数据、一个节点负责定位等等）

完整的机器人系统可能并不是一个物理上的整体，比如这样一个的机器人：




在机器人身体里

- 搭载了一台计算机A，它可以通过机器人的眼睛——摄像头，获取外界环境的信息，也可以控制机器人的腿——轮子，让机器人移动到想要去的地方。
- 可能还会有另外一台计算机B，放在你的桌子上，它可以远程监控机器人看到的信息，也可以远程配置机器人的速度和某些参数，还可以连接一个摇杆，人为控制机器人前后左右运动。

这些功能虽然位于不同的计算机中，但都是这款机器人的工作细胞，也就是节点，他们共同组成了一个完整的机器人系统。

- 节点在机器人系统中的职责就是**执行具体的任务**，从计算机操作系统的角度来看，也叫做进程；
- 每个节点都是一个可以**独立运行的可执行文件**，比如执行某一个python程序，或者执行C++编译生成的结果，都算是运行了一个节点；
- 既然每个节点都是独立的执行文件，那自然就可以想到，得到这个执行文件的**编程语言可以是不同的**，比如C++、Python，乃至Java、Ruby等更多语言。
- 这些节点是功能各不相同的细胞，根据系统设计的不同，可能位于计算机A，也可能位于计算机B，还有可能运行在云端，这叫做**分布式**，也就是可以分布在不同的硬件载体上；
- 每一个节点都需要有**唯一的命名**，当我们想要去找到某一个节点的时候，或者想要查询某一个节点的状态时，可以通过节点的名称来做查询。

节点也可以比喻是一个一个的工人，分别完成不同的任务，他们有的在一线厂房工作，有的在后勤部门提供保障，他们互相可能并不认识，但却一起推动机器人这座“工厂”，完成更为复杂的任务。

(红色：波浪线)

*弹幕：头文件报错不用理会，编译一下即可消除

ctrl+C 没反应 ⇒ RDS::ok();

# 话题通信自定义 msg

ROS 通过 std_msgs 封装了一些原生的数据类型（String、Int32、Char ）

直接传输一些复杂的数据（激光雷达的信息 ）std_msgs 由于描述性

较差而显得力不从心，这种场景下可以使用自定义消息类振

Header：标头包含时间戳和 ROS 常用的坐标帧信息

1，定义 msg 文件 功能包下新建 msg 目录，添加文件 Person.msg

```
string name
uint16 age
float height
```

2 编辑配置文件

package 中添加编译依赖与执行依赖

<build_depend> message_generation </build_depend>

<exec_depend> message_runtime </exec_depend>

CMakeList.txt 编辑 msg 相关配置

3 编辑

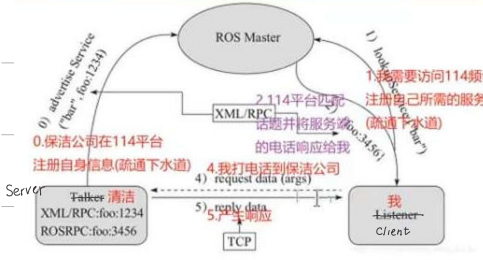
## - 服务通信

服务通信也是ROS中一种极其常用的通信模式，服务通信是基于**请求响应**模式的，是一种应答机制。也即：
一个节点A向另一个节点B发送请求，B接收处理请求并产生响应结果返回给A。比如如下场景：

> 机器人巡逻过程中，控制系统分析传感器数据发现可疑物体或人...此时需要拍摄照片并留存。

作用 用于偶然的、对时时性有要求、有一定逻辑处理需求的数据传输场景

## 、理论模型




角色 ---> 流程 ---> 注意

角色
1.master ----> 管理者（114平台）
2.Server ----> 服务端（服务公司）
3.Client ----> 客户端（我）

流程
master 会根据话题实现 Server 和 Client 的连接

注意：
1.保证顺序，客户端发起请求时，服务端需要已经启动；
2.客户端和服务端都可以存在多个。

关注点：
0.流程已经被封装直接调用即可
1.话题
2.服务端
3.客户端
4.数据载体

## - Srv = 请求 + 响应

```
generate_messages(
    DEPENDENCIES
    std_msgs
)
```

generate_messages 要依赖于 std_msgs

# 参数服务器

参数服务器在ROS中主要用于实现不同节点之间的数据共享。参数服务器相当于是独立于所有节点的一个公共容器，可以将数据存储在该容器中，被不同的节点调用，当然不同的节点也可以往其中存储数据，关于参数服务器的典型应用场景如下：

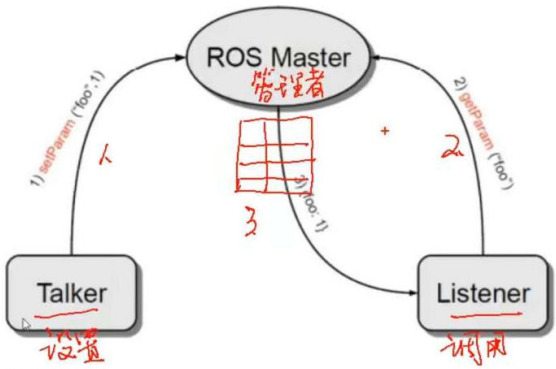
> 导航实现时，会进行路径规划，比如：全局路径规划，设计一个从出发点到目标点的大致路径。本地路径规划，会根据当前路况生成时时的行进路径

上述场景中，全局路径规划和本地路径规划时，就会使用到参数服务器：

- 路径规划时，需要参考小车的尺寸，我们可以将这些尺寸信息存储到参数服务器，全局路径规划节点与本地路径规划节点都可以从参数服务器中调用这些参数

参数服务器，一般适用于存在数据共享的一些应用场景。

# 理论模型



param(键,默认值)

存在，返回对应结果，否则返回默认值

getParam(键,存储结果的变量)

存在,返回 true,且将值赋值给参数2

若果键不存在，那么返回值为 false，且不为参数2赋值

getParamCached键,存储结果的变量)--提高变量获取效率

存在,返回 true,且将值赋值给参数2

若果键不存在，那么返回值为 false，且不为参数2赋值

getParamNames(std::vector<std::string>)

获取所有的键,并存储在参数 vector 中

hasParam(键)

是否包含某个键，存在返回 true，否则返回 false

searchParam(参数1，参数2)

搜索键，参数1是被搜索的键，参数2存储搜索结果的变量

ros::param ----- 与 NodeHandle 类似

# API（应用程序接口）

## 想问一下什么是API，具体是什么意思？

### 都在说API，API到底是什么？

其实日常生活中，我们有很多类似API的场景，比如：

目录

电脑需要调用手机里面的信息，这时候你会拿一根数据线将电脑手机连接起来，电脑和手机上连接数据线的接口就是传说中的API接口。

但比喻到底是比喻，并非本质。想要真正理解API，还得老老实实去理解API的使用场景。

### 我们不妨把API的诞生过程用一个小故事展示出来：

研发人员A开发了软件A，研发人员B正在研发软件B。

有一天，研发人员B想要调用软件A的部分功能来用，但是他又不想从头看一遍软件A的源码和功能实现过程，怎么办呢？

研发人员A想了一个好主意：我把软件A里你需要的功能打包好，写成一个函数；你按照我说的流程，把这个函数放在软件B里，就能直接用我的功能了！

其中，API就是研发人员A说的那个函数。


这就是API的诞生。

### 再举个实例辅助你理解：

【中铁大桥科研院】有一个自研的信息平台，用于管理业务数据。

他们面临一个问题——尽管有信息平台，却因为系统的独立性，数据的上传和备份，需要依靠人工在excel里来回操作，效率很低。

由于系统的开发周期长、成本高，桥科院将目光聚焦到现成的功能软件上。

后来通过API将简道云直接插入公司数据库，数据可自动上传至信息平台上并统一展示；再通过webhook[4]把数据推送到服务器，实现自动备份。

API将信息平台与简道云相连

在这一过程中，通过简道云配备的API接口，可以对接外部系统，让桥科院不用开发直接实现了数据自动上传，备份的功能。

那么，这就是API的使用。

应用程序接口是一组定义、程序及协议的集合，通过 API 接口实现计算机软件之间的相互通信。API 的一个主要功能是提供通用功能集。程序员通过调用 API 函数对应用程序进行开发，可以减轻编程任务。API 同时也是一种中间件，为各种不同平台提供数据共享。

根据单个或分布式平台上不同软件应用程序间的数据共享性能，可以将 API 分为四种类型：

远程过程调用（RPC）：通过作用在共享数据缓存器上的过程（或任务）实现程序间的通信。

标准查询语言（SQL）：是标准的访问数据的查询语言，通过数据库实现应用程序间的数据共享。

文件传输：文件传输通过发送格式化文件实现应用程序间数据共享。

信息交付：指松耦合或紧耦合应用程序间的小型格式化信息，通过程序间的直接通信实现数据共享。

当前应用于 API 的标准包括 ANSI 标准 SQL API。另外还有一些应用于其它类型的标准尚在制定之中。API 可以应用于所有计算机平台和操作系统。这些 API 以不同的格式连接数据（如共享数据缓存器、数据库结构、文件框架），每种数据格式要求以不同的数据命令和参数实现正确的数据通信，但同时也会产生不同类型的错误。因此，除了具备执行数据共享任务所需的知识以外，这些类型的 API 还必须解决很多网络参数问题和可能的差错条件，即每个应用程序都必须清楚自身是否有强大的性能支持程序间通信。相反由于这种 API 只处理一种信息格式，所以该情形下的信息交付 API 只提供较小的命令、网络参数以及差错条件子集。正因为如此，交付 API 方式大大降低了系统复杂性，所以当应用程序需要通过多个平台实现数据共享时，采用信息交付 API 类型是比较理想的选择。

API 与图形用户接口（GUI）或命令接口有着鲜明的差别：API 接口属于一种操作系统或程序接口，而后两者都属于直接用户接口。

## 话题发布

**实现分析:**

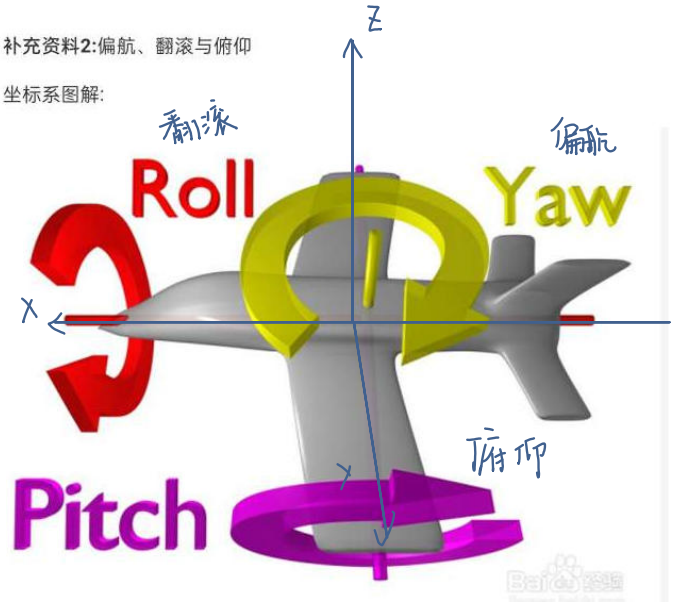
1. 乌龟运动控制实现，关键节点有两个，一个是乌龟运动显示节点 turtlesim_node，另一个是控制节点，二者是订阅发布模式实现通信的，乌龟运动显示节点直接调用即可，运动控制节点之前是使用的turtle_teleop_key通过键盘 控制，现在需要自定义控制节点。
2. 控制节点自实现时，首先需要了解控制节点与显示节点通信使用的话题与消息，可以使用ros命令结合计算图来获取。
3. 了解了话题与消息之后，通过 C++ 或 Python 编写运动控制节点，通过指定的话题，按照一定的逻辑发布消息即可。

**实现流程:**

1. 通过计算图结合ros命令获取话题与消息信息。
2. 编码实现运动控制节点。
3. 启动 roscore、turtlesim_node 以及自定义的控制节点，查看运行结果。

**补充资料2:偏航、翻滚与俯仰**

坐标系图解:



# 话题订阅

**实现分析:**

1. 首先，需要启动乌龟显示以及运动控制节点并控制乌龟运动。
2. 要通过ROS命令，来获取乌龟位姿发布的话题以及消息。
3. 编写订阅节点，订阅并打印乌龟的位姿。

**实现流程:**

1. 通过ros命令获取话题与消息信息。
2. 编码实现位姿获取节点。
3. 启动 roscore、turtlesim_node 、控制节点以及位姿订阅节点，控制乌龟运动并输出乌龟的位姿。

获取话题:/turtle1/pose

```
rostopic list
```

用rostopic list获取到话题，

获取消息类型:turtlesim/Pose

```
rostopic type  /turtle1/pose
```

拿着话题换来消息类型，

获取消息格式:

```
rosmsg info turtlesim/Pose
```

拿着消息类型换来消息格式

响应结果:

```
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

# ROS运行管理

- launch

## 一、launch文件的运行

许多ROS软件包都带有"启动文件"，即launch文件，launch文件是通过roslaunch功能包运行的，命令行格式如下：

```
$ roslaunch package_name file.launch
```

这些启动文件通常会为软件包提供一组节点，这些节点聚合了一些功能，通过roslaunch命令同时启动这些节点。

**RLException: Invalid roslaunch XML syntax: no element found: line 1, column 0**
**The traceback for the exception was written to the log file**

出现这种报错，source -下就正常了:D

## 二、launch文件格式

launch文件采用XML格式书写。

### 1、浏览顺序（Evaluation order）

roslaunch浏览XML文件是一行一行运行的。按深度优先遍历进行处理：标签将按顺序进行浏览，标签值取最后的设定。因此，如果一个参数有多个设置，则将使用为该参数指定为最后一个值。

### 2、标签（Tag）

launch文件标签共有以下11种：

| | | | |
|---|---|---|---|
| <launch> | <node> | <machine> | <include> |
| <remap> | <env> | <param> | <rosparam> |
| <group> | <test> | <arg> | |

- **<launch>**：该标签是任何roslaunch文件的根元素。其唯一目的是充当其他元素的容器。
- **<node>**：最常见的标签，用于启动和关闭节点。roslaunch不保证节点开始的顺序。所以无法从外部知道何时完全初始化节点，所有启动的代码都必须能够按任意顺序启动。

例子：

```
<node name="listener1" pkg="rospy_tutorials" type="listener.py" args="--test" re
```

使用rospy_tutorials功能包的listener.py可执行文件（带有命令行参数--test）启动 listener1节点。如果该节点死亡，它将自动重生。