3. 消息（数据交换的数据形式） 话题、服务、动作都用消息

fieldtype（字段类型） fieldname（字段名称）

① .msg文件 用于话题 常只包含一个字段类型和名称

② .srv文件 服务使用 ---作为分隔符，上层是服务请求，下层是服务响应

③ action文件 动作 --- 分隔，goal、result、feedback（用于任务中途值）

4. 名称（分为相对方法、全局方法、私有方法）

```
int main(int argc, char **argv)                 // 节点主函数
{
  ros::init(argc, argv, "node1");               // 初始化节点
  ros::NodeHandle nh;                           // 声明节点句柄
// 声明发布者，话题名 = bar
ros::Publisher node1_pub = nh.advertise<std_msg::Int32>("bar", 10);
```

这里的节点名称是/node1。如果您用一个没有任何字符的相对形式的bar来声明一个发布者，这个话题将和/bar具有相同的名字。如果以如下所示使用斜杠（/）字符用作全局形式，话题名也是/bar。

这里的节点名称是/node1。如果您用一个没有任何字符的相对形式的bar来声明一个发布者，这个话题将和/bar具有相同的名字。如果以如下所示使用斜杠（/）字符用作全局形式，话题名也是/bar。

```
ros::Publisher node1_pub = nh.advertise<std_msg::Int32>( "/bar" , 10);
```

但是，如果使用波浪号（~）字符将其声明为私有，则话题名称将变为/node1/bar。

```
ros::Publisher node1_pub = nh.advertise<std_msg::Int32>( "~bar" , 10);
```

Node	Relative（基本）	Global	Private
/node1	bar → /bar	/bar → /bar	~bar → /node1/bar
/wg/node2	bar → /wg/bar	/bar → /bar	~bar → /wg/node2/bar
/wg/node3	foo/bar → /wg/foo/bar	/foo/bar → /foo/bar	~foo/bar → /wg/node3/foo/bar

如果要运行两个摄像头应该如何做？ 简单地两次执行相关节点将导致之前执行的节点因ROS的性质而终止，因为ROS必须具有唯一的名称。用户可以在运行时更改节点的名称，而不需要运行额外的程序或更改源代码。方法包括命名空间（namespace）和重新映射（remapping）。

为了帮助理解，假设有一个虚拟的camera_package。假设执行camera_package功能包的camera_node会运行camera节点，此时的运行方法如下。

```
$rosrun camera_package camera_node
```

当该节点将相机的图像值发送给image话题时，可以通过rqt_image_view传送该image话题，如下所示。

```
$rosrun rqt_imgae_view rqt_imgae_view
```

现在让我们通过重新映射来修改这些节点的话题值。如果执行如下命令，只有话题名称将被改为/front/image。其中，image是camera_node的话题名，而这些命令是在运行两个节点的同时，顺便修改名称的例子。

```
$ rosrun camera_package camera_node image:=front/image
$ rosrun rqt_imgae_view rqt_imgae_view image:=front/image
```

例如，当有前、左、右，三个摄像头，且当多次执行同名的节点时，由于节点名重复，该节点将被重复执行，因此节点会被停止。为了避免这种情况，可以采取用同一个名称运行多个不同的节点的方法。下面的命令示例中，name选项使用了两个下划线（__）。附加地说明，选项__ns、__name、__log、__ip、__hostname和__master是运行节点时使用的特殊选项。我们在话题名称选项中使用了一个下划线（_），如果它是private名称，则在现有名称前加上一个下划线。

```
$ rosrun camera_package camera_node __name:=front _device:=/dev/video0
$ rosrun camera_package camera_node __name:=left _device:=/dev/video1
$ rosrun camera_package camera_node __name:=right _device:=/dev/video2
$ rosrun rqt_imgae_view rqt_imgae_view
```

如果想绑定到单一的命名空间，可以如下操作。这样会将指定的节点和话题都绑定到一个命名空间，因此会改变所有的名称。

```
$ rosrun camera_package camera_node __ns:=back
$ rosrun rqt_imgae_view rqt_imgae_view __ns:=back
```

5. 坐标变换 Transform

的概念之一。这些可以被描述为位置（position）和方向（direction），统称为姿态（pose）。在此，位置由x、y、z这3个矢量表示，而方向是用四元数（quaternion）x、y、z、w表示。四元数并不直观，因为它们没有使用我们在日常生活中使用的三元数的角度表达方式：滚动角（roll）、俯仰角（pitch）和偏航角（yaw）。但这种四元数方式不存在滚动、俯仰和偏航矢量的欧拉（Euler）方式具有的万向节死锁（gimbal lock）问题或速度问题，因此在机器人工程中人们更喜欢用四元数（quaternion）的形式。因为同样的原因，ROS中也大量使用四元数。当然，考虑到方便，它也提供将欧拉值转换成四元数的功能。

类似上面说明的消息（message），TF使用以下格式[44]。Header用于记录转换的时间，并使用名为child_frame_id的消息来表示下位的坐标。并且为了表达坐标的转换值，使用transformTranslation.x / transform.translation.y / transform.translation.z / transform.rotation.x / transform.rotation.y / transform.rotation.z / transform.rotation.w 的数据形式描述对方的位置和方向。

geometry_msgs/TransformStamped.msg

```
Header header
string child_frame_id
Transform transform
```

6. 异构设备间通信：节点通信不受节点所在操作系统种类和编程语言影响

7. 文件系统

ROS 以功能包为单位开发，包含一个以上节点或用于运行节点的配置文件，用元功能包管理　功能包安装分二进制文件安装和源代码安装

package.xml 包含功能包信息的XML文件　catkin构建系统 CMakeLists.txt 构建环境

安装目录 /opt/ros/kinetic

- /bin　可执行的二进制文件
- /etc　与ROS和catkin相关的配置文件
- /include　头文件
- /lib　库文件
- /share　ROS功能包
- env.*　配置文件
- setup.*　配置文件

工作文件 ~/catkin_ws/

- /build　构建相关的文件
- /devel　msg、srv头文件、用户包库、可执行文件
- /src　用户功能包

- /include　头文件
- /launch　用于roslaunch的启动文件
- /node　用于rospy的脚本
- /msg　消息文件
- /src　源代码文件
- /srv　服务文件

- CMakeLists.txt　构建配置文件
- package.xml　功能包配置文件

8. 构建系统

① 创建功能包

先移至工作目录 $ cd ~/catkin_ws/src

创建名为"my_first_ros_pkg"功能包

```
$ catkin_create_pkg my_first_ros_pkg std_msgs roscpp
```

使用标准消息包 std_msgs → 客户端库 roscpp

② 修改功能包配置文件 package.xml

包括功能包名称、作者、许可、依赖

③ 修改构建配置文件 CMakeList.txt

设置可执行文件创建、依赖包优先构建、连接器创建

project（功能包）　find_package（依赖包）

构建配置文件（CMakeLists.txt）中的每一项如下所示。第一条是操作系统中安装的cmake的最低版本。由于它目前被指定为版本2.8.3，所以如果使用低于此版本的cmake，则必须更新版本。

```
cmake_minimum_required(VERSION 2.8.3)
```

project项是功能包的名称。只需使用用户在package.xml中输入的功能包名即可。请注意，如果功能包名称与package.xml中的<name>标记中描述的功能包名称不同，则在构建时会发生错误，因此需要注意。

```
project(my_first_ros_pkg)
```

find_package项是进行构建所需的组件包。目前，roscpp和std_msgs被添加为依赖包。如果此处没有输入功能包名称，则在构建时会向用户报错。换句话说，这是让用户先创建依赖包的选项。

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)
```

add_message_files是添加消息文件的选项。FILES将引用当前功能包目录的msg目录中的*.msg文件，自动生成一个头文件（*.h）。在这个例子中，我们将使用消息文件Message1.msg和Message2.msg。

```
add_message_files(
  FILES
  Message1.msg
  Message2.msg
)
```

add_service_files是添加要使用的服务文件的选项。使用FILES会引用功能包目录中的srv目录中的*.srv文件。在这个例子中，用户可以选择使用服务文件Service1.srv和Service2.srv。

```
add_service_files(
  FILES
  Service1.srv
  Service2.srv
)
```

generate_messages是设置依赖的消息的选项。此示例是将DEPENDENCIES选项设置为使用std_msgs消息包。

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

以下是catkin 构建选项。INCLUDE_DIRS表示将使用INCLUDE_DIRS后面的内部目录include的头文件。LIBRARIES表示将使用随后而来的功能包的库。

CATKIN_DEPENDS后面指定如roscpp或std_msgs等依赖包。目前的设置是表示依赖于roscpp和std_msgs。DEPENDS是一个描述系统依赖包的设置。

```
catkin_package(
 INCLUDE_DIRS include
 LIBRARIES my_first_ros_pkg
 CATKIN_DEPENDS roscpp std_msgs
 DEPENDS system_lib
)
```

include_directories是可以指定包含目录的选项。目前设定为${catkin_INCLUDE_DIRS}，这意味着将引用每个功能包中的include目录中的头文件。当用户想指定一个额外的include目录时，写在${catkin_INCLUDE_DIRS}的下一行即可。

```
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

add_library声明构建之后需要创建的库。以下是引用位于my_first_ros_pkg功能包的src目录中的my_first_ros_pkg.cpp文件来创建my_first_ros_pkg库的命令。

```
add_library(my_first_ros_pkg
  src/${PROJECT_NAME}/my_first_ros_pkg.cpp
)
```

add_dependencies是在构建该库和可执行文件之前，如果有需要预先生成的有依赖性的消息或dynamic_reconfigure，则要先执行。以下内容是优先生成my_first_ros_pkg库依赖的消息及dynamic reconfigure的设置。

```
add_dependencies(my_first_ros_pkg ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

add_executable是对于构建之后要创建的可执行文件的选项。以下内容是引用src/my_first_ros_pkg_node.cpp文件生成my_first_ros_pkg_node可执行文件。如果有多个要引用的*.cpp文件，将其写入my_first_ros_pkg_node.cpp之后。如果要创建两个以上的可执行文件，需追加add_executable项目。

```
add_executable(my_first_ros_pkg_node src/my_first_ros_pkg_node.cpp)
```

④ 编写源代码 → add_executable（可执行文件）

创建

```
$ cd ~/catkin_ws/src/my_first_ros_pkg/src/
$ gedit hello_world_node.cpp
```