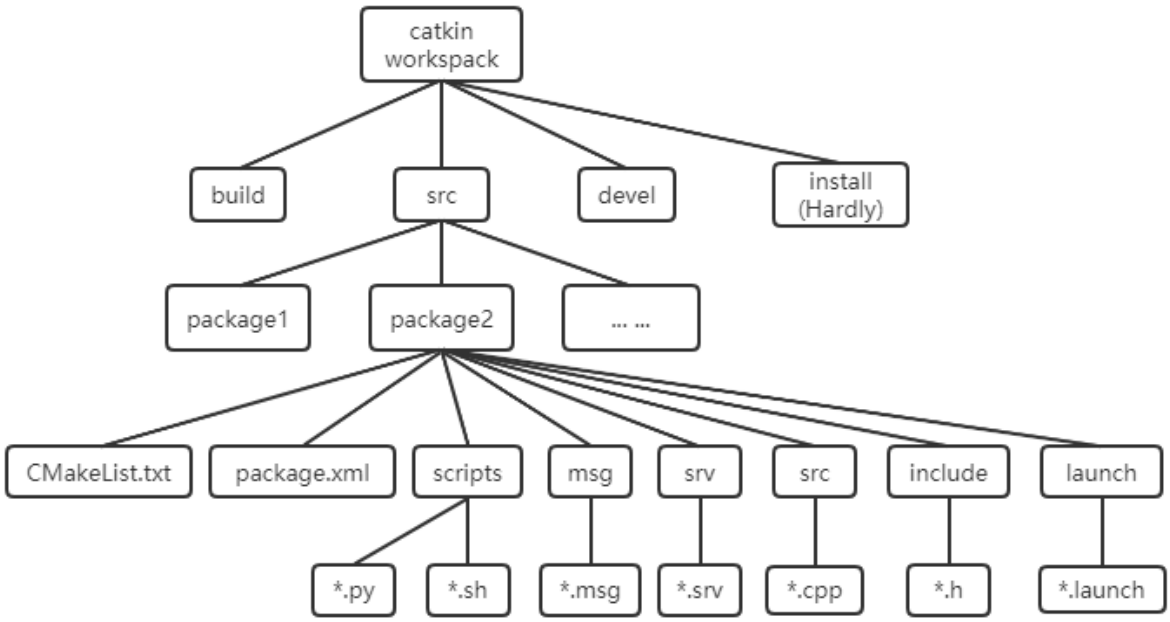
7.2 Introduction of project files

7.2.1 Project file structure

The file structure of ROS is shown below. Not every folder is necessary, it is designed according to your needs.



7.2.2 Workspace

The workspace is the place to manage and organize the ROS project files. It can be described as a warehouse, which is loaded with various ROS projects, it is convenient for system organization and management. It is a folder in the visual graphical interface. The ROS code we written is usually placed in the workspace. There are four main first-level directories:

- src: source space; ROS catkin package (source code package)
- build: compilation space; cache information and intermediate files of catkin (CMake)
- devel: Development space; generated object files (including header files, dynamic link libraries, static link libraries, executable files, etc.), environment variables.
- install: Installation space

The top-level workspace (you can name it arbitrarily) and the src (name must be src) folder need to be created by yourself;

- build and devel folders are automatically created by the catkin_make command;
- The install folder is automatically created by the catkin_make install command. In generally, we didn't create this folder.

Note: Before using catkin_make to compile, be sure to return to the top-level workspace. Function packages with the same name are not allowed in the same workspace; function packages with the same name are allowed in different workspaces.

```
mkdir -p ~/catkin_ws/src  # create
cd catkin_ws/             # enter workspace
catkin_make               # compile
source devel/setup.bash   # Update the workspace environment
```

7.2.3 Function package

package is a specific file structure and folder combination. Usually, we put the program code that implements the same specific function into a package. Only CMakeLists.txt and package.xml are [**Required**], and the remaining paths are determined according to whether the package is needed.

Create function package

```
cd ~/catkin_ws/src
catkin_create_pkg my_pkg rospy rosmsg roscpp
```

【rospy】、【rosmsg】、【roscpp】are dependent library, which can be based on business needs.

File structure

```
|-- CMakeLists.txt  # (Required) The compilation rules of the current package.
|-- package.xml     # (Required) The description of the package. Usually,we will
add some ros library support file.
|-- include folder  # Store c++ header files
|-- config folder   # Store parameter configuration files
|-- launch folder   # Store launch file(.launch or.xml)
|-- meshes folder   # Store 3D models of robots or simulation scenes(.sda, .stl,
.dae,etc) ;
|-- urdf folder     # Storage model description of the robot(.urdf or .xacro) ;
|-- rviz folder     # rviz file
|-- src folder      # c++ code
|-- scripts folder  # Executable scripts; such as shell scripts (.sh), Python
scripts (.py);
|-- srv folder      # customize service
|-- msg folder      # customize topic
|-- action folder   # customize action
```

7.2.4 CMakeLists.txt introduction

1. Overview

`CMakeLists.txt` was originally a rule file of the Cmake compilation system, and the Catkin compilation system basically followed the CMake compilation style, but added some macro definitions for the ROS project. `CMakeLists.txt` of catkin is basically the same as CMake.

`CMakeLists.txt` is very important. It specifies the rules from source to target file. When the catkin build system works, it will first find the `CMakeLists.txt` inthe each package, and then compile and build according to the rules.

2. Format

The overall structure is as follows:

```
cmake_minimum_required() # Required CMake version
project()                # Package name
find_package()           # Find other CMake/Catkin packages needed for
compilation
catkin_python_setup()    # Enable Python module support
add_message_files()      # message generator
```

```
add_service_files()      # service generator
add_action_files()       # action generator
generate_message()       # Generate msg/srv/action interfaces in different
language versions
catkin_package()         # Generate the cmake configuration of the current
package for other packages that depend on this
package to call
add_library()            # Be used to specify the library generated by the
compilation. The default catkin compilation generates
a shared library.
add_executable()         # Generate executable binary file
add_dependencies()       # Define target files to depend on other target files
to ensure that other targets have been built
target_link_libraries()  # Specifies the library to which the executable file is
linked. This should be used after
add_executable().
catkin_add_gtest()       # Test establishment
install()                # Install to this machine
```

3. Boost

If you use C++ and Boost, you need to call `find_package()` on Boost and specify which aspects of Boost are used as components.

For example, if you want to use Boost threads, you need to enter the following command

```
find_package（Boost REQUIRED COMPONENTS thread）
```

4. catkin_package ()

catkin_package() is a CMake macro provided by **catkin**. This is necessary to specify catkin-specific information for the build system, which in turn is used to generate pkg-config and CMake files.

Before using `add_library()` or `add_executable()` to declare any target, this function must be called. This function has 5 **optional** parameters:

- `INCLUDE_DIRS` -Heard file export path
- `LIBRARIES` - Libraries exported from the project
- `CATKIN_DEPENDS` - Other `catkin` projects that this project depends on
- `DEPENDS` - The non-catkin CMake project that this project depends on.
- `CFG_EXTRAS` - Other configuration options

More details, please click here to check.

Eg:

```
catkin_package(
   INCLUDE_DIRS include
   LIBRARIES ${PROJECT_NAME}
   CATKIN_DEPENDS roscpp nodelet
   DEPENDS eigen opencv)
```

This indicates that the folder "include" within the package folder is where exported headers go. The CMake environment variable ${PROJECT_NAME} evaluates to whatever you passed to the `project()` function earlier, in this case it will be "robot_brain". "roscpp" + "nodelet" are packages that need to be present to build/run this package, and "eigen" + "opencv" are system dependencies that need to be present to build/run this package.

5. Package path and library path

Prior to specifying targets, you need to specify where resources can be found for said targets, specifically header files and libraries:

```
- Include Paths - Where can header files be found for the code (most common in
C/C++) being built
- Library Paths - Where are libraries located that executable target build
against?
- include_directories(<dir1>, <dir2>, ..., <dirN>)
- link_directories(<dir1>, <dir2>, ..., <dirN>)
```

- include_directories ()

The argument to include_directories should be the *_INCLUDE_DIRS variables generated by your find_package calls and any additional directories that need to be included. If you are using catkin and Boost, your include_directories() call should look like:

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

The first argument "include" indicates that the `include/` directory within the package is also part of the path.

- link_directories ()

Eg:

```
link_directories(~/my_libs)
```

The CMake `link_directories()` function can be used to add additional library paths, however, this is not recommended. All catkin and CMake packages automatically have their link information added when they are find_packaged. Simply link against the libraries in `target_link_libraries()`

6. xecutable Targets

To specify an executable target that must be built, we must use the `add_executable()` CMake function.

```
add_executable(myProgram src/main.cpp src/some_file.cpp src/another_file.cpp)
```

This will build a target executable called *myProgram* which is built from 3 source files: src/main.cpp, src/some_file.cpp and src/another_file.cpp.

7. Library Targets

The `add_library()` CMake function is used to specify libraries to build. By default catkin builds shared libraries.

```
add_library（$ {PROJECT_NAME} $ {$ {PROJECT_NAME} _SRCS}）
```

8. target_link_libraries

Use the `target_link_libraries()` function to specify which libraries an executable target links against. This is done typically after an `add_executable()` call. Add `${catkin_LIBRARIES`

Syntax:

```
target_link_libraries（<executableTargetName>, <lib1>, <lib2>, ... <libN>）
```

Eg:

```
add_executable(foo src/foo.cpp)
add_library(moo src/moo.cpp)
target_link_libraries(foo moo)  -- This links foo against libmoo.so
```

Note that there is no need to use `link_directories()` in most use cases as that information is automatically pulled in via `find_package()`.

9. Messages, Services, and Action Targets

Messages (.msg), services (.srv), and actions (.action) files in ROS require a special preprocessor build step before being built and used by ROS packages. The point of these macros is to generate programming language-specific files so that one can utilize messages, services, and actions in their programming language of choice. The build system will generate bindings using all available generators (e.g. gencpp, genpy, genlisp, etc).

There are three macros provided to handle messages, services, and actions respectively:

- `add_message_files`
- `add_service_files`
- `add_action_files`

These macros must then be followed by a call to the macro that invokes generation:

```
generate_messages（）
```

7.2.5 package.xml introduction

- Overview

The **package manifest** is an XML file called **package.xml** that must be included with any catkin-compliant package's root folder. This file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages. Note the concept is similar to the *manifest.xml* file used in the legacy rosbuild build system.

`package.xml` file build_depend must contain message_generation, and run_depend must contain message_runtime.

- format

```
<pacakge>
<name>
<version>
<description>
<maintainer>
<license>
<buildtool_depend>
<depend>
<build_depend>
<build_export_depend>
<exec_depend>
<test_depend>
<doc_depend>
```

- Dependencies关系

The package manifest with minimal tags does not specify any dependencies on other packages. Packages can have six types of dependencies:

Build Dependencies specify which packages are needed to build this package. This is the case when any file from these packages is required at build time. This can be including headers from these packages at compilation time, linking against libraries from these packages or requiring any other resource at build time (especially when these packages are *find_package()*-ed in CMake). In a cross-compilation scenario build dependencies are for the targeted architecture.
Build Export Dependencies specify which packages are needed to build libraries against this package. This is the case when you transitively include their headers in public headers in this package (especially when these packages are declared as (CATKIN*)DEPENDS in catkin_package() in CMake).*
Execution Dependencies *specify which packages are needed to run code in this package. This is the case when you depend on shared libraries in this package (especially when these packages are declared as (CATKIN*)DEPENDS in *catkin_package()* in CMake).
Test Dependencies specify only *additional* dependencies for unit tests. They should never duplicate any dependencies already mentioned as build or run dependencies.
Build Tool Dependencies specify build system tools which this package needs to build itself. Typically the only build tool needed is catkin. In a cross-compilation scenario build tool dependencies are for the architecture on which the compilation is performed.

Documentation Tool Dependencies specify documentation tools which this package needs to generate documentation.

- Additional Tags

`<url>` - A URL for information on the package, typically a wiki page on ros.org.

`<author>` -The author(s) of the package

```
  <url type="website">http://www.ros.org/wiki/turtlesim</url>
  <author>Yahboom</author>
```