开发规范

（草案）

目录

第一章 前言

为确保项目组代码的质量、可维护性和团队协作效率，特制定本编码规范。本规范旨在为基于 ROS（Robot Operating System）框架，使用 C++与 Python 语言进行开发的项目提供统一的编程指导。

规范的核心目标是保证代码的一致性、清晰性和健壮性。通过遵循统一的命名、格式、架构设计及最佳实践，使得不同开发者编写的代码具有统一的风格，从而降低阅读与理解的难度，减少潜在错误，并提升软件的可扩展性和可复用性。

本规范适用于所有基于 ROS 或 ROS 2 的软件项目开发工作，涵盖从驱动、感知、规划到控制等各模块的代码编写。所有项目成员在开发新代码、修改或重构现有代码时，均应严格遵守本规范的各项条款。规范的最终解释权归项目组所有，并将随技术演进与项目需求进行迭代更新。

第二章 命名规范

2.1 概述

总体原则：命名要清晰、明了，有明确含义，能够准确表达其用途或含义。优先选择完整的单词而非随意的缩写。如果名称过长，可适当使用业界或项目内公认的缩写（如 cmd for command,img for image），但禁止使用自创或难以理解的缩写。

避免使用魔数：在程序中禁止使用不易理解的纯数字（魔数），必须用有意义的命名常量或枚举来替代。特别是涉及物理状态、单位、配置参数或含有物理意义的常量。

名称长度：命名不应过短（如无意义的 a,b）或过长，应在表意清晰的前提下保持简洁。临时变量和循环变量可使用短名称（如 i,j,msg），公共接口、全局变量和常量名称应具有完整的描述性。

选择名称时的思考：为一个变量或函数命名时遇到的困难，通常意味着对其职责或目的的理解不够清晰。此时应重新分析其设计，确保其功能单一且明确。

2.2 C++ 特定命名规范

ROS C++开发遵循以下约定（与 ROS CppStyleGuide 保持一致）：

文件命名：全部小写，以下划线分隔。头文件使用.h 或.hpp，源文件使用.cpp。

示例：`laser_driver.h,multi_target_tracker.cpp`

类/结构体/枚举类型命名：使用 PascalCase（大驼峰）命名法。

示例：`class LaserDriver,`

`struct Point2D,`

`enum class NavigationStatus`

函数/方法命名：使用 camelCase（小驼峰）命名法，动词开头。

示例：`publishData(),getTransform(),laserScanCallback()`

变量命名：使用 snake_case（蛇形）命名法（全部小写，下划线分隔）。

示例：`robot_name,linear_velocity_x,is_connected`

常量命名：使用全大写的 SNAKE_CASE 命名法。

示例：`const int MAX_ITERATIONS = 1000;,const double PI = 3.14159;`

命名空间命名：使用全小写的 snake_case 命名法。

示例：`namespace perception_utils {}`

成员变量：建议添加后缀下划线_以区分成员变量和局部变量。

示例：`ros::Publisher pub_;,std::string frame_id_;`

2.3 Python 特定命名规范

ROS Python 开发遵循 PEP 8 风格指南：

文件/模块命名：全部小写，以下划线分隔。

示例：`lasar_driver.py,path_planner.py`

类/异常命名：使用 PascalCase（大驼峰）命名法。

示例：`class LaserDriver:,class InvalidPoseError(Exception):`

函数/方法命名：使用 snake_case（蛇形）命名法。

示例：`def publish_data():,def get_transform():`

变量/参数命名：使用 snake_case（蛇形）命名法。

示例：`robot_name,linear_velocity_x,is_connected`

常量命名：使用全大写的 SNAKE_CASE 命名法。

示例：`MAX_LASER_RANGE = 20.0,DEFAULT_FRAME_ID = 'map'`

私有成员：使用单下划线_前缀表示“保护”或“内部使用”的成员，双下划线__前缀表示“私有”成员（实际会触发名称修饰）。

示例：`_internal_cache,__private_method()`

2.4 ROS 实体命名规范

话题（Topic）/服务（Service）名称：使用 snake_case（蛇形）命名法，具有描述性，通常使用命名空间组织。

示例：/sensor/lidar/scan,/robot/odom,/navigation/set_goal

消息/服务/动作文件：使用 PascalCase(大驼峰)命名法,字段名使用 snake_case。

消息文件：LaserScan.msg->float32 range_min

服务文件：GetMap.srv->---nav_msgs/OccupancyGrid map

动作文件：MoveBase.action->geometry_msgs/PoseStamped target_pose

节点名称：使用 snake_case（蛇形）命名法，清晰反映节点功能。

示例：lasar_driver_node,move_base_node

第三章　代码外观

3.1 缩进与换行

3.1.1 缩进规则

使用空格进行缩进：不同编辑器对 Tab 的显示宽度可能不同，尽量使用空格可保证在所有环境下显示一致。

C++：遵循 ROS C++风格指南，使用 **2 个空格**作为一级缩进。

Python：遵循 PEP 8，使用 **4 个空格**作为一级缩进。

保持整个项目中缩进风格的一致性。

3.1.2 行宽限制

代码行宽应控制在 **80-120 个字符**以内，以保证在多数开发环境和分辨下无需横向滚动即可完整阅读。当表达式超出规定的列宽时，遵循以下规则进行换行：

1.在逗号后换行

2.在操作符前换行

3.规则 1 优先于规则 2

4.换行后应进行适当的缩进（通常增加一级缩进），以明确表示延续关系

C++示例：

```cpp
// 长函数调用换行
robot_controller_.calculateTrajectory(initial_pose, target_pose,
                                      current_velocity, max_acceleration,
                                      &result_trajectory);

// 长条件判断换行
if (laser_range_ > minimum_detection_range &&
    laser_range_ < maximum_detection_range &&
    obstacle_detected_ == false) {
  avoidObstacle();
}

// 长赋值语句换行
double total_distance = distance_to_goal +
                        additional_path_correction +
                        safety_margin;
```

Python 示例：

```python
# 长函数调用换行
rospy.loginfo("Received new goal point: x=%.2f, y=%.2f, theta=%.2f",
              goal_pose.x, goal_pose.y, goal_pose.theta)

# 长条件判断换行
if (current_time - last_update_time > update_threshold and
    not system_initialized and
    sensor_data_ready):
    initializeSystem()
```

3.1.3 空行使用规则

空行用于分隔逻辑上独立的代码块，提高可读性：

1. **函数/方法之间**：使用 2 个空行分隔
2. **类内部**：成员变量、方法组之间使用 1 个空行分隔
3. **函数/方法内部**：不同逻辑块之间使用 1 个空行分隔
4. **#include/import 分组**：不同分组的 include/import 语句之间使用 1 个空行分隔

C++示例：

```cpp
#include <sensor_msgs/LaserScan.h>

#include <vector>
#include <string>

#include "my_package/laser_processor.h"

// 类定义
class LaserDriver {
public:
  LaserDriver();  // 构造函数
  ~LaserDriver(); // 析构函数

  void init();    // 初始化方法

private:
  void processScan(const sensor_msgs::LaserScan::ConstPtr& scan);

  ros::NodeHandle nh_;
  ros::Subscriber laser_sub_;
  std::vector<float> ranges_;
};  // class LaserDriver
```

Python 示例：

```python
# 导入分组
import rospy
import numpy as np

from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

# 类定义
class NavigationNode:

    def __init__(self):
        """初始化方法"""
        self.rate = rospy.Rate(10)  # 10Hz
        self.current_pose = None

    def setup_publishers(self):
        """设置发布者"""
        self.cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)

    def setup_subscribers(self):
        """设置订阅者"""
        rospy.Subscriber('/scan', LaserScan, self.scan_callback)
```

3.1.4 空格使用规则

空格的有效使用会使代码更容易阅读：

1. **操作符两侧**：大多数运算符之前和之后都需增加空格

示例：`int result = a + b;,if (value == 10)`

2. **逗号后**：多个参数之间、逗号后需增加空格

示例：`function(arg1, arg2, arg3)`

3. **控制语句**：关键字和括号之间需增加空格

示例：`if (condition),while (true)`

4. **花括号**：左花括号前需增加空格

示例：`if (condition) {, void function() {`

3.1.5 花括号使用规范

C++花括号规则：

1.左花括号 "{" 放在语句的同一行末尾

2.右花括号 "}" 单独成行，并与相应语句对齐

3.即使代码块中只有一条语句，也必须使用花括号

```
// 正确示例
if (condition) {
  doSomething();
}
while (condition) {
  // 空循环体也使用花括号
}
for (int i = 0; i < count; i++) {
  processItem(items[i]);
}
// 错误示例
if (condition)
  doSomething();  // 缺少花括号
if (condition) { doSomething(); }  // 花括号在同一行
```

Python 无需花括号，使用缩进表示代码块：

```
if condition:
    do_something()
    while condition:
    # 空循环体使用 pass
    pass
for i in range(count):
    process_item(items[i])
```

3.2 文件结构组织

通用结构：

```
功能包名/
├── include/功能包名/    # C++头文件
├── src/               # C++源文件
├── scripts/           # Python 可执行脚本
├── launch/            # 启动文件
├── config/            # 配置文件
├── package.xml        # 包定义文件
└── CMakeLists.txt     # 构建配置
```

文件组织原则：

头文件和源文件分离
功能相关的文件集中存放
配置文件与代码分离

第四章　代码注释

4.1 注释概述

代码自解释原则：优秀的代码应当具有自解释性。通过有意义的命名、清晰的结构和良好的设计，使代码本身易于理解。注释应作为补充，而非主要解释手段。避免为了注释而注释，避免每行代码都添加注释。

注释语言：注释原则上应使用中文描述，以确保项目组成员能够准确理解。注释应使用完整的句子，表达清晰、准确。

注释内容：注释应该阐明代码中较复杂的业务实现、逻辑结构以及不十分明显的任何内容。避免增加多义性，避免多余的或不适当的注释。特别要解释"为什么"这么做，而不仅仅是"做什么"。

注释位置：避免在代码行末尾添加注释，避免杂乱的注释格式。应使用适当的空白将注释同代码分开，保持代码整洁。

注释维护：遵循"边写代码边注释"的原则，修改代码同时修改相应的注释，保证注释与代码的一致性。及时删除不再有用的注释。定期复查代码注释，确保其准确性。

4.2 文件头注释

每个源文件开头应包含文件头注释，提供文件的元信息：

C++示例：

```cpp
/**
 * @Description: 激光雷达驱动节点
 * @Author: 张工
 * @Date: 2023-10-27
 * @License: MIT
 */
```

Python 示例：

```python
#!/usr/bin/env python3
"""
@Description: 自主导航节点
@Author: 李工
@Date: 2023-11-15
@License: Apache 2.0
"""
```

4.3 类(模块)注释

每个类定义前应添加详细的类注释：

C++示例：

```cpp
// 激光雷达驱动类
// 负责管理与物理雷达设备的通信和数据采集
class LaserDriver {
public:
/**
 * @Description:
 * @Param:
 * @Retrun:
 */
  bool medianFilter(const sensor_msgs::LaserScan& input_scan,
                    sensor_msgs::LaserScan& output_scan,
                    int window_size);
};
```

Python 示例：

```python
class NavigationNode:
    """自主导航核心类，实现路径规划与运动控制"""

    def calculate_path(self, start_pose, end_pose, obstacles):
        """
        @Description:
        @Param:
        @Return:
        """
        pass
```

4.4 行内注释

行内注释用于解释复杂的代码段或临时解决方案：

C++示例：

```cpp
// 计算机器人到目标点的欧几里得距离
double dx = target_pose.x - current_pose.x;
double dy = target_pose.y - current_pose.y;
double distance = sqrt(dx*dx + dy*dy);

// 临时解决方案：由于雷达数据偶尔出现异常值，需要特殊处理
// TODO: 待硬件团队修复雷达固件后移除此处理
if (distance > MAX_SENSOR_RANGE) {
  distance = MAX_SENSOR_RANGE;  // 限制最大距离值
```

```
  ROS_WARN("Distance measurement capped at maximum range");
}
// 应用卡尔曼滤波平滑距离数据
filtered_distance = kalman_filter_.update(distance);
```

Python 示例：

```
# 转换坐标系：从机器人坐标系到世界坐标系
rotated_x = x * cos(theta) - y * sin(theta)
rotated_y = x * sin(theta) + y * cos(theta)
# 注意：由于 ROS 使用右手坐标系，Y 轴方向与常规数学坐标系相反
world_x = rotated_x + origin_x
world_y = -rotated_y + origin_y  # Y 轴取反
# FIXME: 此处存在浮点数精度问题，可能导致累计误差
# 长期运行时需要考虑定期重置坐标系
current_pose = (world_x, world_y, theta)
# 检查是否到达目标点容差范围内
if distance < POSITION_TOLERANCE:
    self.state = RobotState.ARRIVED  # 更新状态为已到达
```

4.6 特殊注释标签

使用标准化的注释标签提高可维护性：

```
// TODO: 标记需要后续完成的工作
// TODO: 张工 - 优化路径搜索算法，当前版本效率较低

// FIXME: 标记已知问题或临时解决方案
// FIXME: 李工 - 此处内存泄漏需修复，暂时增加手动释放

// NOTE: 重要说明或注意事项
// NOTE: 此参数值对系统稳定性影响较大，修改需谨慎

// HACK: 标记不太优雅的解决方案
// HACK: 由于 ROS 消息类型限制，此处使用自定义数据结构

// XXX: 标记需要重点关注的代码段
// XXX: 此函数在多线程环境下调用，必须保证线程安全
```

4.7 注释格式插件配置

为提高注释一致性，建议使用以下插件：

对于需要规范管理文件头和函数注释的场景，**KoroFileHeader** 插件是一个非常强大且受欢迎的选择

1.安装插件：在 VSCode 扩展商店中搜索 "koroFileHeader" 并安装,如图 4-1

2.基本配置与使用：安装后，通常需要进行一些配置来定义注释模板。配置通常在 VSCode 的 `settings.json` 文件中进行

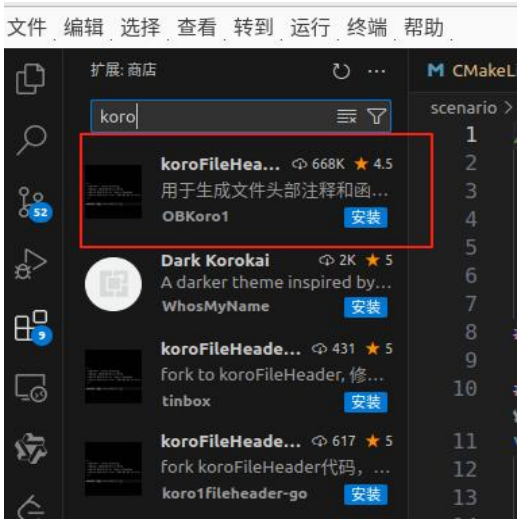

图 4-1

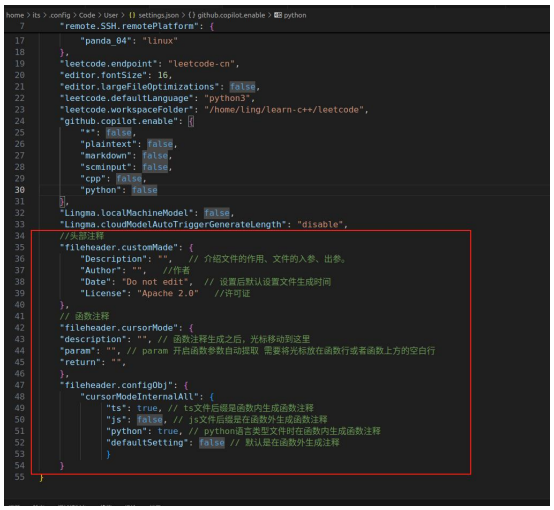

图 4-2

通过在设置中搜索 `fileheader` 来找到相关配置项，并点击"在 settings.json 中编辑"进行配置。常用的配置字段包括：

`fileheader.customMade`: 用于**文件头部注释**的模板。

`fileheader.cursorMode`: 用于**函数注释**的模板。

`fileheader.configObj`: 插件的一些全局配置，如自定义时间格式、语言映射等

3.使用快捷键：

文件头部注释：在文件开头按下 `Ctrl + Win + I`

函数注释：将光标放在函数行或函数上方，按下 `Ctrl + Win + T`

4.具体操作：

如何找到setting.json设置模板

1. 简单的输入命令

- 打开VSCode命令面板: mac: `command + p` window: `ctrl + p`
- 输入 `> Open Settings` (注意 `>` 后面有一个空格)

2. 从设置中打开

首选项 > 设置 > 搜索 `fileheader` > 在右侧中贴上配置 > 做简单的更改

统一配置头部注释格式和函数注释格式：（如图 4-2 所示）

```
    //头部注释
  "fileheader.customMade": {
      "Description": "",   // 介绍文件的作用、文件的入参、出参。
      "Author": "",   //作者
      "Date": "Do not edit",  // 设置后默认设置文件生成时间
      "License": "Apache 2.0"   //许可证
```

```
    },
    // 函数注释
    "fileheader.cursorMode": {
        "description": "", // 函数注释生成之后，光标移动到这里
        "param": "", // param 开启函数参数自动提取 需要将光标放在函数行或者函数上方的空白行
        "return": "",
    },
    "fileheader.configObj": {
        "cursorModeInternalAll": {
            "ts": true, // ts 文件后缀是函数内生成函数注释
            "js": false, // js 文件后缀是在函数外生成函数注释
            "python": true, // python 语言类型文件时在函数内生成函数注释
            "defaultSetting": false // 默认是在函数外生成注释
        }
    }
```

如果还有个性化操作需要添加大家可以参考详细文档：

[安装和快速上手 · OBKoro1/koro1FileHeader Wiki · GitHub]

通过遵循以上注释规范，可以显著提高代码的可读性和可维护性，便于团队协作和知识传承。

第五章　声明

5.1 声明概述

5.1.1 变量作用域

优先使用局部变量：尽可能限制变量的作用域，避免不必要的全局变量。

ROS2 节点设计：在 ROS2 节点中，使用类成员变量存储节点状态，避免使用全局变量。

参数处理：使用 ROS2 参数机制而非全局变量配置节点行为。

5.1.2 声明格式

一行一个声明：每行只声明一个变量，提高可读性。

```
// 推荐
int level;
int size;
// 不推荐
int x, y;
```

5.1.3 初始化

声明时初始化：变量在声明时尽可能进行初始化。

```
// 推荐
int counter = 0;
std::string node_name = "laser_driver";
// 不推荐
int counter;
std::string node_name;
```

5.1.4 访问控制

封装原则：使用适当的访问修饰符控制类成员的可见性。

ROS2 节点类：将节点内部状态和实现细节声明为 private，仅将必要的接口暴露为 public。

通过遵循以上声明规范，可以确保 ROS2 代码的结构清晰、可维护性强，并符合 ROS2 的最佳实践。

第六章 ROS2 系统开发规范

6.1 概述

本章主要描述基于 ROS2 框架进行系统开发过程中的限制性规范和建议性规范，包括系统架构设计、开发约束和最佳实践。

6.2 系统开发过程限制性规范

6.2.1 接口兼容性

废弃接口处理：对于计划废弃的接口（话题、服务、动作），不能直接删除，应使用`[[deprecated]]`属性（C++）或适当的废弃标记进行标识，并说明替代方案和迁移路径。
版本管理：接口变更需遵循语义化版本控制，保证向后兼容性。

6.2.2 资源使用规范

依赖管理：所有依赖必须在`package.xml`和`CMakeLists.txt`（或`setup.py`）中明确声明。

资源文件：配置文件、启动文件等应放置在 package 的相应目录（`config/`,`launch/`,`resource/`）中
参数配置：节点参数应支持动态重配置，并提供合理的默认值

6.2.3 安全与可靠性

输入验证：对所有接收的外部数据（话题消息、服务请求等）进行有效性检查
异常处理：妥善处理可能出现的异常情况，确保节点不会意外崩溃
资源清理：正确管理资源生命周期，特别是在节点关闭时释放所有资源

6.2.4 性能规范

实时性要求：对于实时性要求高的应用，使用合适的 QoS 策略。

内存管理：避免内存泄漏，合理使用智能指针管理资源。

计算效率：优化算法复杂度，避免阻塞主线程.

6.3 ROS2 开发建议性规范

6.3.1 节点设计

```
// 推荐使用组件化节点设计
class MyComponent : public rclcpp::Node {public:
  explicit MyComponent(const rclcpp::NodeOptions & options)
```

```
  : Node("my_component", options) {
    // 初始化逻辑
  }

  // 使用生命周期管理
  virtual ~MyComponent() = default;};

// 在主函数中注册组件
#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(MyComponent)
```

6.3.2 接口设计最佳实践

消息设计：消息字段应使用有意义的名称，避免过于复杂的嵌套结构。

服务设计：服务应设计为原子操作，响应时间可控。

动作设计：长时间运行的任务使用动作（Action）而非服务.

6.3.3 参数管理

```
// 参数声明和获取
this->declare_parameter("max_speed", 2.0);
double max_speed = this->get_parameter("max_speed").as_double();
// 参数回调
auto param_callback = [this](std::vector<rclcpp::Parameter> parame
ters) {
  // 处理参数变更
};
this->set_on_parameters_set_callback(param_callback);
```

6.3.4 日志记录

```
// 使用适当的日志级别
RCLCPP_DEBUG(this->get_logger(), "Detailed debugging information
");
RCLCPP_INFO(this->get_logger(), "General information");
RCLCPP_WARN(this->get_logger(), "Warning message");
RCLCPP_ERROR(this->get_logger(), "Error message");
RCLCPP_FATAL(this->get_logger(), "Fatal error message");
```

6.4 系统架构说明

6.4.1 ROS2 典型架构层次

```
应用层 (Application Layer)
    ├── 功能包集 (Metapackages)
    │     └── 功能包 (Packages)
    │           ├── 节点 (Nodes)
```

```
    │           ├── 接口 (Interfaces)
    │           ├── 启动文件 (Launch Files)
    │           └── 配置 (Configurations)
中间件层 (Middleware Layer)
    ├── DDS 实现 (DDS Implementation)
    └── ROS2 RMW 接口
操作系统层 (OS Layer)
```

6.4.2 功能包组织原则

功能内聚：每个功能包应专注于单一功能领域。

接口明确：包之间的依赖通过明确定义的接口进行。

分层设计：按照数据流和处理层次组织功能包。

6.4.3 组件化设计

```cpp
// 使用组件化架构提高系统模块化程度
class PerceptionComponent : public rclcpp::Node {
  // 感知相关功能
};

class PlanningComponent : public rclcpp::Node {
  // 规划相关功能
};

class ControlComponent : public rclcpp::Node {
  // 控制相关功能
};

// 通过组合管理器整合多个组件
rclcpp::executors::MultiThreadedExecutor executor;
auto perception_node = std::make_shared<PerceptionComponent>();
auto planning_node = std::make_shared<PlanningComponent>();
auto control_node = std::make_shared<ControlComponent>();

executor.add_node(perception_node);
executor.add_node(planning_node);
executor.add_node(control_node);
executor.spin();
```

通过遵循以上 ROS/ROS2 系统开发规范，可以确保开发的系统具有良好的可维护性、可扩展性和可靠性，同时符合 ROS/ROS2 生态系统的最佳实践。

第七章 结语

以上规范结合了 ROS/ROS2 框架特性与现代软件工程的最佳实践，旨在为基于 ROS/ROS2 的机器人系统开发提供清晰、统一的指导。规范涵盖了代码风格、架构设计、节点开发、测试验证等关键环节，力求在保证代码质量与性能的同时，兼顾开发效率与团队协作。

需要强调的是，技术规范并非一成不变的教条，而应随着技术演进和项目需求不断优化调整。在实际开发过程中，请结合具体场景灵活应用本规范，并在团队内达成共识。

文中如有不足之处，敬请各位同仁批评指正。期待与大家共同完善这份开发规范，推动 ROS/ROS2 在机器人领域的更佳实践。

保持代码整洁，尊重协作规范，共建高质量机器人系统！