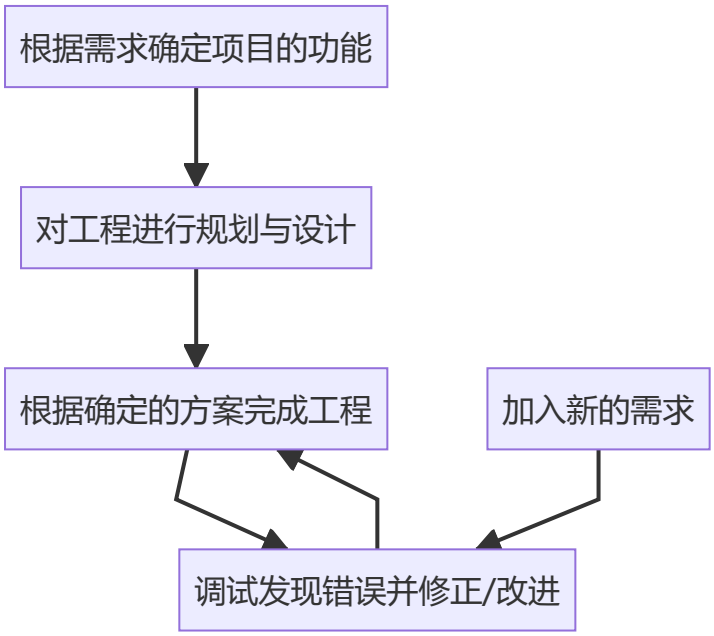
# E唯协会软件工程设计与管理规范

## 流程

一个工程的流程包含四个方面:需求、设计、制作/编程、调试。其中的调试往往会伴随者新的需求出现而需更改工程的结构、功能等不同方面。

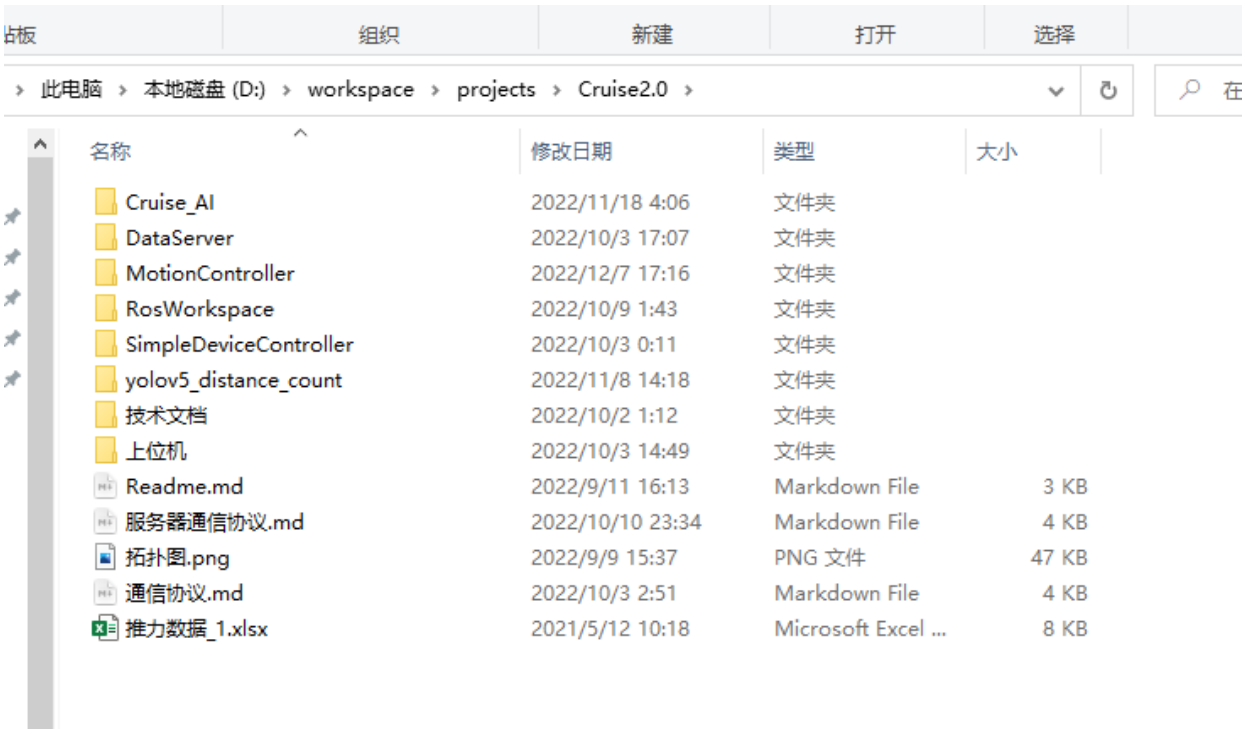



## 文件和版本

### 工程及其子项目的命名

建议尽量使用ASCII中的字符命名，以避免不同设备下的乱码问题和一些老旧软件的本地化问题。

此外，建议采用驼峰法命名，如下图所示的“DataServer”、“MotionController”。

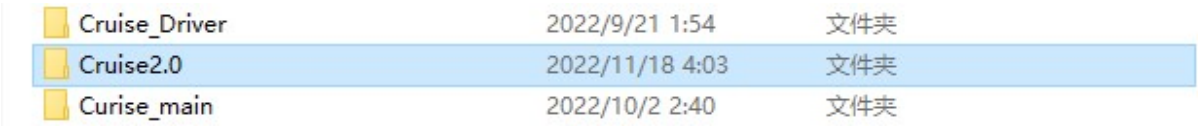



# 文件管理

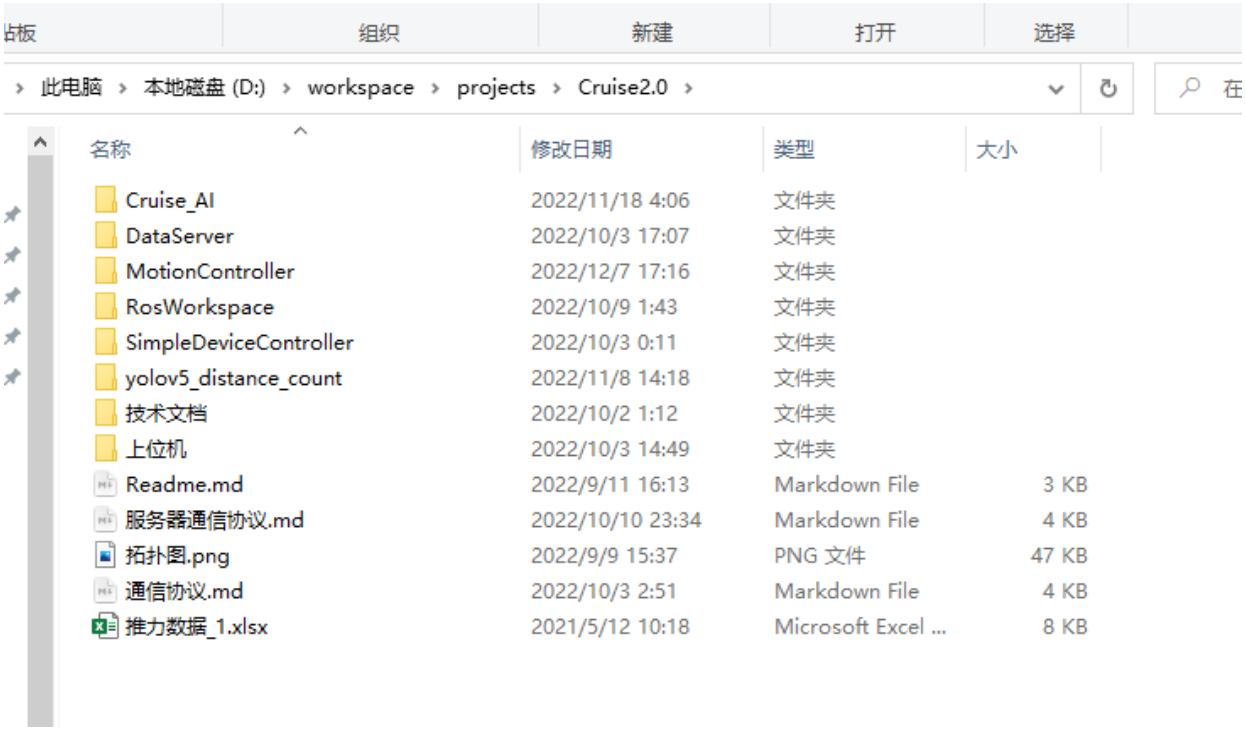
建议使用一个总的文件夹容纳该单一工程下的所有文件，并根据其内容和用途使用合理命名的文件夹进行合理的分类，同时应尽量避免中文文件/文件夹名的出现而导致在其他工程中的移植问题。

此外尤其是控制部的同学应该注意，文件路径中应尽量使用 "/" 而非 “\\”、“\”；尽量使用相对路径而非绝对路径以避免在不同设备/操作系统中的移植问题。

## 实例




如上图所示，新巡游的所有代码文件都被归至文件夹"Cruise2.0"中



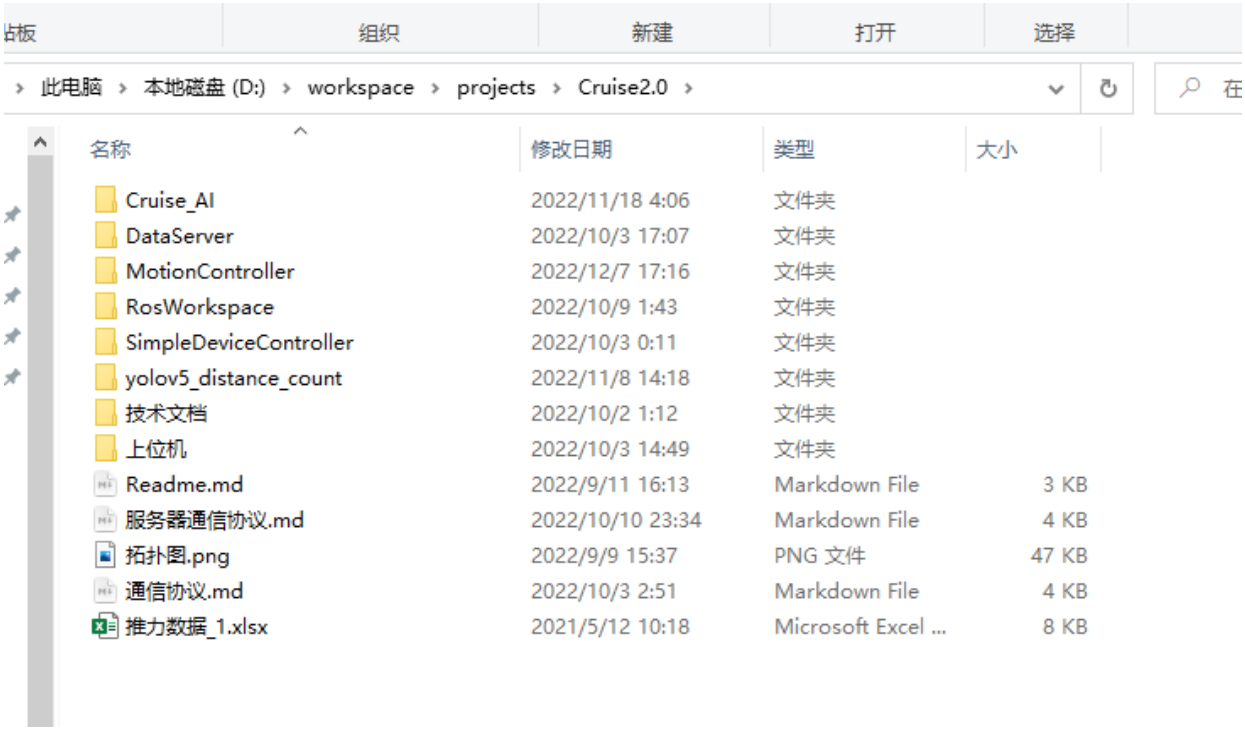

如上图所示在，在"Cruise2.0"文件夹下，巡游机器人不同部分的代码被安放在不同的文件夹下。

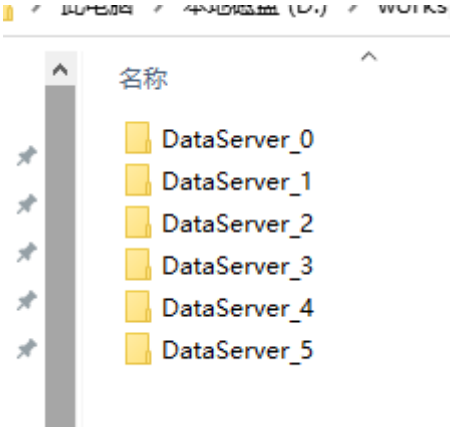
## 版本管理

由于协会的项目并不算特别大型，而且大量人员并不会使用Git之流的管理工具，在这里介绍一套基于文件/文件夹命名的管理方式。

将一个工程中的各个子项目划分至对应的文件夹下，并为各个子项目的对应版本的名字添加对应的后缀如"_0"对应初始版本，"_3"对应第3版。

### 实例






如上图所示，数据服务器中的各版本代码被依次添加后缀，以标识其对应版本；并且都被存放在文件夹"DataServer"中。

# 代码规范

对于控制部目前的工作而言，主要的工作集中在`C`、`python`、`Ros1/ROS2`上

## C

控制部的C语言工作主要集中在对于单片机的开发，下文主要讨论对于单片机的开发与调试工作

### 文件

C语言中主要包含.c与.h两种文件类型。在我们的工作场景下，.c与.h两种文件通常是成对出现的(我们也是这么干的)，通常我们将.h文件归入文件夹“Inc”或"include"中、将.c文件归入文件夹“Src”中，对应的.c与.h文件通常使用相通的名称。对应功能的文件都建议放至对应的文件夹下。理想的C语言工程文件夹下的文件结构应该是这样的。

启动文件通常为汇编文件.s，Makefile为一文本文件用于指示编译指令。这些文件通常由于数量较少，可直接存放于主文件夹下。

```
FFF
|---Num
    |---Inc
        |---1.h
        |---2.h
    |---Src
        |---1.c
        |---2.c
|---alphabet
    |---Inc
        |---a.h
        |---b.h
    |---Src
        |---a.c
        |---b.c
|---Readme.md
|---Makefile
|---Startup.s
|---link.ld
```

## 头文件

所有头文件都应该有 `#define` 保护来防止头文件被多重包含, 命名格式当是:
`<PROJECT>_<PATH>_<FILE>_H_`

为保证唯一性, 头文件的命名应该基于所在项目源代码树的全路径. 例如, 项目 `foo` 中的头文件 `foo/src/bar/baz.h` 可按如下方式保护:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif // FOO_BAR_BAZ_H_
```

在使用 `#include` 时可将同一模块内的头文件堆放在一起。

```
#include "main.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

#include "server.h"
```

## 函数

函数应当尽量按值返回， 否则按引用返回。 避免返回指针， 除非它可以为空。

函数应尽量简短、凝练。长函数有时是合理的, 因此并不硬性限制函数的长度.。如果函数超过 40 行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割。即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题, 甚至导致难以发现的 bug. 使函数尽量简短, 以便于他人阅读和修改代码。在处理代码时, 你可能会发现复杂的长函数. 不要害怕修改现有代码: 如果证实这些代码使用 / 调试起来很困难, 或者你只需要使用其中的一小段代码, 考虑将其分割为更加简短并易于管理的若干函数。

当一个函数长度短于10行左右,且不包含延时操作，建议将其内联。

当函数中的一个变量在函数结束后需要保留其中的内容，应使用 `static` 类型而非全局变量，使其仅在单一函数内可被调用，以此来保证数据的安全。

非常不建议使用缺省参数用于传递数据。缺省参数是在每个调用点都要进行重新求值的, 这会造成生成的代码迅速膨胀. 作为读者, 一般来说也更希望缺省的参数在声明时就已经被固定了, 而不是在每次调用时都可能会有不同的取值。缺省参数会干扰函数指针, 导致函数签名与调用点的签名不一致. 而函数重载不会导致这样的问题。

## 变量

### 局部变量

将函数变量尽可能置于最小作用域内, 并在变量声明时进行初始化.

C允许在函数的任何位置声明变量. 我们提倡在尽可能小的作用域中声明变量, 离第一次使用越近越好. 这使得代码浏览者更容易定位变量声明的位置, 了解变量的类型和初始值. 特别是，应使用初始化的方式替代声明再赋值, 比如:

```
int i;
i = f(); // 坏——初始化和声明分离
int j = g(); // 好——初始化时声明
vector<int> v;
v.push_back(1); // 用花括号初始化更好
v.push_back(2);
vector<int> v = {1, 2}; // 好——v 一开始就初始化
```

属于 `if`, `while` 和 `for` 语句的变量应当在这些语句中正常地声明，这样子这些变量的作用域就被限制在这些语句中了，举例而言:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

有一个例外, 如果变量是一个对象, 每次进入作用域都要调用其构造函数, 每次退出作用域都要调用其析构函数. 这会导致效率降低.

```
// 低效的实现
for (int i = 0; i < 1000000; ++i) {
    Foo f;                  // 构造函数和析构函数分别调用 1000000 次!
    f.DoSomething(i);
}
```

在循环作用域外面声明这类变量要高效的多:

```
Foo f;                      // 构造函数和析构函数只调用 1 次
for (int i = 0; i < 1000000; ++i) {
    f.DoSomething(i);
}
```

**静态和全局变量**

禁止定义静态储存周期非POD(POD : Plain Old Data)变量，禁止使用含有副作用的函数初始化POD全局变量，因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。

禁止使用类的 静态储存周期 变量：由于构造和析构函数调用顺序的不确定性，它们会导致难以发现的 bug 。不过 `constexpr` 变量除外，毕竟它们又不涉及动态初始化或析构。

静态生存周期的对象，即包括了全局变量，静态变量，静态类成员变量和函数静态变量，都必须是原生数据类型 : 即 int, char 和 float, 以及 POD 类型的指针、数组和结构体。

静态变量的构造函数、析构函数和初始化的顺序在 C中是只有部分明确的，甚至随着构建变化而变化，导致难以发现的 bug. 所以除了禁用类类型的全局变量，我们也不允许用函数返回值来初始化 POD 变量，除非该函数不涉及任何全局变量。函数作用域里的静态变量除外，毕竟它的初始化顺序是有明确定义的，而且只会在指令执行到它的声明那里才会发生。

## 命名

### 通用命名规则

函数命名, 变量命名, 文件命名要有描述性; 少用缩写。尽可能使用描述性的命名, 别心疼空间, 毕竟相比之下让代码易于新读者理解更重要. 不要用只有项目开发者能理解的缩写, 也不要通过砍掉几个字母来缩写单词。

```
int price_count_reader;    // 无缩写
int num_errors;            // "num" 是一个常见的写法
int num_dns_connections;   // 人人都知道 "DNS" 是什么
int n;                     // 毫无意义.
int nerr;                  // 含糊不清的缩写.
int n_comp_conns;          // 含糊不清的缩写.
int wgc_connections;       // 只有贵团队知道是什么意思.
int pc_reader;             // "pc" 有太多可能的解释了.
int cstmr_id;              // 删减了若干字母.
```

注意, 一些特定的广为人知的缩写是允许的, 例如用 `i` 表示迭代变量和用 `T` 表示模板参数.

模板参数的命名应当遵循对应的分类: 类型模板参数应当遵循类型命名的规则, 而非类型模板应当遵循变量命名的规则。

### 文件命名

文件名要全部小写, 可以包含下划线 (`_`) 或连字符 (`-`), 依照项目的约定. 如果没有约定, 那么 "`_`" 更好。

可接受的文件命名示例:

- `my_useful_class.c`
- `my-useful-class.c`
- `myusefulclass.c`

C 文件要以 `.c` 结尾, 头文件以 `.h` 结尾. 专门插入文本的文件则以 `.inc` 结尾。

不要使用已经存在于 `/usr/include` 下的文件名 (即编译器搜索系统头文件的路径), 如 `db.h`.

通常应尽量让文件名更加明确. `http_server_logs.h` 就比 `logs.h` 要好。文件名一般成对出现, 如 `foo_bar.h` 和 `foo_bar.c`。

内联函数定义必须放在 `.h` 文件中. 如果内联函数比较短, 就直接将实现也放在 `.h` 中.

### 变量命名

变量 (包括函数参数) 和数据成员名一律小写, 单词之间用下划线连接. 类，函数的成员变量以下划线结尾, 但结构体的就不用, 如: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`.

#### 普通变量命名

举例:

```
string table_name;  // 好 - 用下划线.
string tablename;   // 好 - 全小写.

string tableName;  // 差 - 混合大小写
```

#### 结构体变量

不管是静态的还是非静态的, 结构体数据成员都可以和普通变量一样:

```
struct UrlTableProperties {
  int num;
  int num_entries;
  };
```

#### 常量、全局变量、静态变量命名

声明为 `constexpr` 或 `const` 的变量, 或在程序运行期间其值始终保持不变的, 命名时以 特定字母开头：常量为“k”，全局变量为“g”，静态变量为“s”； 大小写采取驼峰式命名法。

```
const int kDaysInAWeek = 7;
```

### 函数命名

一般来说, 函数名的每个单词首字母大写 (即 “驼峰变量名” 或 “帕斯卡变量名”), 没有下划线. 对于首字母缩写的单词, 更倾向于将它们视作一个单词进行首字母大写 (例如, 写作 `StartRpc()` 而非 `StartRPC()` ).

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

取值和设值函数的命名与变量一致. 一般来说它们的名称与实际的成员变量对应, 但并不强制要求. 例如 `int count()` 与 `void set_count(int count)`.

### 枚举命名

枚举的命名应当和常量一致: `kEnumName`。

```
enum UrlTableErrors {
    kOK = 0,
    kErrorOutOfMemory = 1,
    kErrorMalformedInput = 2,
};
```

### 宏命名

宏像这样命名: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN` 全部大写, 使用下划线.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

## 注释

使用 `//` 或 `/* */`, 统一就好。`//` 或 `/* */` 都可以; 但 `//` *更* 常用. 要在如何注释及注释风格上确保统一。

### 函数声明

基本上每个函数声明处前都应当加上注释, 描述函数的功能和用途. 只有在函数的功能简单而明显时才能省略这些注释(例如, 简单的取值和设值函数). 注释使用叙述式 (“Opens the file”) 而非指令式 (“Open the file”); 注释只是为了描述函数, 而不是命令函数做什么. 通常, 注释不会描述函数如何工作. 那是函数定义部分的事情.

函数声明处注释的内容:

- 函数的输入输出.
- 对类成员函数而言: 函数调用期间对象是否需要保持引用参数, 是否会释放这些参数.
- 函数是否分配了必须由调用者释放的空间.
- 参数是否可以为空指针.
- 是否存在函数使用上的性能隐患.
- 如果函数是可重入的, 其同步前提是什么?

```
/*
 * 函数名: Dev_Server_Init
 * 描述   : 通信初始化
 * 输入   : /
 * 输出   : /
 * 备注   : /
 */
void Dev_Server_Init(void)
{
    Uart1RecCout = 0;
    Uart2RecCout = 0;
    Uart3RecCout = 0;
    Uart4RecCout = 0;
    Uart5RecCout = 0;

    HAL_UART_Receive_IT(&huart1, (uint8_t *)&Uart1RecBuf, 1);
    HAL_UART_Receive_IT(&huart2, (uint8_t *)&Uart2RecBuf, 1);
    HAL_UART_Receive_IT(&huart3, (uint8_t *)&Uart3RecBuf, 1);
    HAL_UART_Receive_IT(&huart4, (uint8_t *)&Uart4RecBuf, 1);
    HAL_UART_Receive_IT(&huart5, (uint8_t *)&Uart5RecBuf, 1);

    char startlogo[19] = "Data Sever started";
    memcpy(&Uart5RecBuf, &startlogo, 19);
```

```
    HAL_UART_Transmit(&huart5, (uint8_t *)&Uart5RecBuf, 19, 10);
}
```

**变量注释**

通常变量名本身足以很好说明变量用途. 某些情况下, 也需要额外的注释说明含义及用途

**代码块注释**

对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.

**代码前注释**

巧妙或复杂的代码段前要加注释. 比如:

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

**行注释**

比较隐晦的地方要在行尾加入注释. 在行尾空两格进行注释. 比如:

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged.
```

注意, 这里用了两段注释分别描述这段代码的作用, 和提示函数返回时错误已经被记入日志.

如果你需要连续进行多行注释, 可以使之对齐获得更好的可读性:

```
DoSomething();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Two spaces between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
std::vector<string> list{
                    // Comments in braced lists describe the next element...
                    "First item",
                    // .. and should be aligned appropriately.
"Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

# Python

## 文件、包的管理

仅对包和模块使用导入,而不单独导入函数或者类。注意模块名仍可能冲突。 有些模块名太长, 不太方便。同时导入时不要使用相对名称，即使模块在同一个包中, 也要使用完整包名， 这能避免无意间导入一个包两次.

使用模块的全路径名来导入每个模块，所有的新代码都应该用完整包名来导入每个模块。应该像下面这样导入:

```
# 在代码中引用完整名称 absl.flags (详细情况).
import absl.flags
from doctor.who import jodie

FLAGS = absl.flags.FLAGS
# 在代码中仅引用模块名 flags (常见情况).
from absl import flags
from doctor.who import jodie

FLAGS = flags.FLAGS
```

## 条件表达式

条件表达式(又名三元运算符)是对于if语句的一种更为简短的句法规则. 例如: `x = 1 if cond else 2` .

条件表达式虽然更加简短和方便，但是难于阅读，且如果表达式很长，会导致难于定位条件。

条件表达式适用于单行函数. 写法上推荐真实表达式,if表达式,else表达式每个独占一行.在其他情况下，推荐使用完整的if语句.

## 嵌套/局部/内部类或函数

使用内部类或者嵌套函数可以用来覆盖某些局部变量，允许定义仅用于有效范围的工具类和函数。在装饰器中比较常用。

但嵌套类或局部类的实例不能序列化(pickled). 内嵌的函数和类无法直接测试.同时内嵌函数和类会使外部函数的可读性变差.

使用内部类或者内嵌函数可以忽视一些警告.但是应该避免使用内嵌函数或类,除非是想覆盖某些值.若想对模块的用户隐藏某个函数,不要采用嵌套它来隐藏,应该在需要被隐藏的方法的模块级名称加 `_` 前缀,这样它依然是可以被测试的.

## 全局变量

避免使用全局变量. 鼓励使用模块级的常量,例如 `MAX_HOLY_HANDGRENADE_COUNT = 3` .注意常量命名必须全部大写,用 `_` 分隔.具体参见命名规则若必须要使用全局变量,应在模块内声明全局变量,并在名称前 `_` 使之成为模块内部变量.外部访问必须通过模块级的公共函数.具体参见 命名规则。

## 线程

虽然Python的内建类型例如字典看上去拥有原子操作, 但是在某些情形下它们仍然不是原子的(即: 如果**hash**或**eq**被实现为Python方法)且它们的原子性是靠不住的. 你也不能指望原子变量赋值(因为这个反过来依赖字典).

优先使用Queue模块的 `Queue` 数据类型作为线程间的数据通信方式. 另外, 使用threading模块及其锁原语(locking primitives). 了解条件变量的合适使用方式, 这样你就可以使用 `threading.Condition` 来取代低级别的锁了.

## 语言特性

在自己的代码中避免使用特性。当然，利用了这些特性的来编写的一些标准库是值得去使用的，比如 `abc.ABCMeta`, `collection.namedtuple`, `dataclasses` 等.

## 异常

异常是可以使用的，但必须小心。异常必须遵守特定条件:

1. 优先合理的使用内置异常类.比如 `ValueError` 指示了一个程序错误, 比如在方法需要正数的情况下传递了一个负数错误.不要使用 `assert` 语句来验证公共API的参数值. `assert` 是用来保证内部正确性的,而不是用来强制纠正参数使用.若需要使用异常来指示某些意外情况,不要用 `assert`,用 `raise` 语句,例如:

```
def connect_to_next_port(self, minimum):
    """Connects to the next available port.

    Args:
        minimum: A port value greater or equal to 1024.

    Returns:
        The new minimum port.

    Raises:
        ConnectionError: If no available port is found.
    """
    if minimum < 1024:
        # Note that this raising of ValueError is not mentioned in the doc
        # string's "Raises:" section because it is not appropriate to
        # guarantee this specific behavioral reaction to API misuse.
        raise ValueError(f'Min. port must be at least 1024, not {minimum}.')
    port = self._find_next_open_port(minimum)
    if not port:
        raise ConnectionError(
            f'Could not connect to service on port {minimum} or higher.')
    assert port >= minimum, (
        f'Unexpected port {port} when minimum was {minimum}.')
    return port
```

2. 模块或包应该定义自己的特定域的异常基类, 这个基类应该从内建的Exception类继承. 模块的异常基类后缀应该叫做 `Error`.

3. 永远不要使用 `except:` 语句来捕获所有异常, 也不要捕获 `Exception` 或者 `StandardError` , 除非你打算重新触发该异常, 或者你已经在当前线程的最外层(记得还是要打印一条错误消息). 在异常这方面, Python非常宽容, `except:` 真的会捕获包括Python语法错误在内的任何错误. 使用 `except:` 很容易隐藏真正的bug.
4. 尽量减少try/except块中的代码量. try块的体积越大, 期望之外的异常就越容易被触发. 这种情况下, try/except块将隐藏真正的错误.
5. 使用finally子句来执行那些无论try块中有没有异常都应该被执行的代码. 这对于清理资源常常很有用, 例如关闭文件.

## 风格

### 分号

禁用分号!

### 行长度

每行长度如非特殊情况，禁止超过80个字符。

### 括号

宁缺毋滥，除非是用于实现行连接, 否则不要在返回语句或条件语句中使用括号.。不过在元组两边使用括号是可以的。

### 缩进

用4个空格来缩进代码，不要用 `tab` , 特别严禁 `tab` 和空格混用(此处的 `tab` 指tab字符)。

### 序列元素尾部逗号

仅当 `]` , `)` , `}` 和末位元素不在同一行时，推荐使用序列元素尾部逗号. 当末位元素尾部有逗号时，元素后的逗号可以指示 YAPF 将序列格式化为每行一项.

```
golomb3 = [0, 1, 3]
golomb4 = [
          0,
          1,
          4,
          6,
          ]
```

### 空行

顶级定义之间空两行, 方法定义之间空一行.

顶级定义之间空两行, 比如函数或者类定义. 方法定义, 类定义与第一个方法之间, 都应该空一行. 函数或方法中, 某些地方要是你觉得合适, 就空一行.

**空格**

按照标准的排版规范来使用标点两边的空格

括号内不要有空格.

```
Yes: spam(ham[1], {eggs: 2}, [])
No:  spam( ham[ 1 ], { eggs: 2 }, [ ] )
```

不要在逗号, 分号, 冒号前面加空格, 但应该在它们后面加(除了在行尾).

```
Yes: if x == 4:
         print(x, y)
     x, y = y, x
No:  if x == 4 :
         print(x , y)
     x , y = y , x
```

参数列表, 索引或切片的左括号前不应加空格.

```
Yes: spam(1)
no: spam (1)
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

在二元操作符两边都加上一个空格, 比如赋值(=), 比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not), 布尔(and, or, not). 至于算术操作符两边的空格该如何使用, 需要你自己好好判断. 不过两侧务必要保持一致.

```
Yes: x == 1
No:  x<1
```

当 `=` 用于指示关键字参数或默认参数值时, 不要在其两侧使用空格. 但若存在类型注释的时候,需要在 `=` 周围使用空格.

```
Yes: def complex(real, imag=0.0): return magic(r=real, i=imag)
Yes: def complex(real, imag: float = 0.0): return Magic(r=real, i=imag)
No:  def complex(real, imag = 0.0): return magic(r = real, i = imag)
No:  def complex(real, imag: float=0.0): return Magic(r = real, i = imag)
```

不要用空格来垂直对齐多行间的标记, 因为这会成为维护的负担(适用于:, #, =等):

```
Yes:
     foo = 1000  # comment
     long_name = 2  # comment that should not be aligned

     dictionary = {
         "foo": 1,
         "long_name": 2,
         }
```

```
No:
     foo       = 1000  # comment
     long_name = 2     # comment that should not be aligned

     dictionary = {
         "foo"      : 1,
         "long_name": 2,
         }
```

### Shebang

大部分.py文件不必以#!作为文件的开始. 根据 PEP-394 , 程序的main文件应该以 `#!/usr/bin/python2` 或者 `#!/usr/bin/python3` 开始.

`#!` 先用于帮助内核找到Python解释器, 但是在导入模块时, 将会被忽略. 因此只有被直接执行的文件中才有必要加入 `#!`.

### 注释

确保对模块, 函数, 方法和行内注释使用正确的风格

#### 文档字符串

Python有一种独一无二的的注释方式: 使用文档字符串. 文档字符串是包, 模块, 类或函数里的第一个语句. 这些字符串可以通过对象的 `__doc__` 成员被自动提取, 并且被pydoc所用. (你可以在你的模块上运行pydoc试一把, 看看它长什么样). 我们对文档字符串的惯例是使用三重双引号"""( PEP-257 ). 一个文档字符串应该这样组织: 首先是一行以句号, 问号或惊叹号结尾的概述(或者该文档字符串单纯只有一行). 接着是一个空行. 接着是文档字符串剩下的部分, 它应该与文档字符串的第一行的第一个引号对齐. 下面有更多文档字符串的格式化规范.

#### 模块

每个文件应该包含一个许可样板. 根据项目使用的许可(例如, Apache 2.0, BSD, LGPL, GPL), 选择合适的样板. 其开头应是对模块内容和用法的描述.

```
"""A one line summary of the module or program, terminated by a period.

Leave one blank line.  The rest of this docstring should contain an
overall description of the module or program.  Optionally, it may also
contain a brief description of exported classes and functions and/or usage
examples.

Typical usage example:

foo = ClassFoo()
bar = foo.FunctionBar()
"""
```

**函数和方法**

下文所指的函数,包括函数, 方法, 以及生成器.
一个函数必须要有文档字符串, 除非它满足以下条件:

1. 外部不可见
2. 非常短小
3. 简单明了
   文档字符串应该包含函数做什么, 以及输入和输出的详细描述. 通常, 不应该描述"怎么做", 除非是一些复杂的算法. 文档字符串应该提供足够的信息, 当别人编写代码调用该函数时, 他不需要看一行代码, 只要看文档字符串就可以了. 对于复杂的代码, 在代码旁边加注释会比使用文档字符串更有意义. 覆盖基类的子类方法应有一个类似 `See base class` 的简单注释来指引读者到基类方法的文档注释.若重载的子类方法和基类方法有很大不同,那么注释中应该指明这些信息.
   关于函数的几个方面应该在特定的小节中进行描述记录， 这几个方面如下文所述. 每节应该以一个标题行开始. 标题行以冒号结尾. 除标题行外, 节的其他内容应被缩进2个空格.

- Args:
  列出每个参数的名字, 并在名字后使用一个冒号和一个空格, 分隔对该参数的描述.如果描述太长超过了单行80字符,使用2或者4个空格的悬挂缩进(与文件其他部分保持一致). 描述应该包括所需的类型和含义. 如果一个函数接受*foo(可变长度参数列表)或者**bar (任意关键字参数), 应该详细列出*foo和**bar.
- Returns: (或者 Yields: 用于生成器)
  描述返回值的类型和语义. 如果函数返回None, 这一部分可以省略.
- Raises:
  列出与接口有关的所有异常.

```
def fetch_smalltable_rows(table_handle: smalltable.Table,
                          keys: Sequence[Union[bytes, str]],
                          require_all_keys: bool = False,
) -> Mapping[bytes, Tuple[str]]:
    """Fetches rows from a Smalltable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by table_handle.  String keys will be UTF-8 encoded.

    Args:
        table_handle: An open smalltable.Table instance.
        keys: A sequence of strings representing the key of each table
        row to fetch.  String keys will be UTF-8 encoded.
        require_all_keys: Optional; If require_all_keys is True only
        rows with values set for all keys will be returned.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {b'Serak': ('Rigel VII', 'Preparer'),
        b'Zim': ('Irk', 'Invader'),
        b'Lrrr': ('Omicron Persei 8', 'Emperor')}
```

```
        Returned keys are always bytes.  If a key from the keys argument is
        missing from the dictionary, then that row was not found in the
        table (and require_all_keys must have been False).

    Raises:
        IOError: An error occurred accessing the smalltable.
    """
```

#### 类

类应该在其定义下有一个用于描述该类的文档字符串. 如果你的类有公共属性(Attributes), 那么文档中应该有一个属性(Attributes)段. 并且应该遵守和函数参数相同的格式.

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

#### 块注释和行注释

最需要写注释的是代码中那些技巧性的部分. 如果你在下次 代码审查 的时候必须解释一下, 那么你应该现在就给它写注释. 对于复杂的操作, 应该在其操作开始前写上若干行注释. 对于不是一目了然的代码, 应在其行尾添加注释.

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # True if i is 0 or a power of 2.
```

为了提高可读性, 注释应该至少离开代码2个空格.

另一方面, 绝不要描述代码. 假设阅读代码的人比你更懂Python, 他只是不知道你的代码要做什么.

```
# BAD COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1
```

### 命名

模块名写法: `module_name` ;包名写法: `package_name` ;类名: `ClassName` ;方法名: `method_name` ;异常名: `ExceptionName` ;函数名: `function_name` ;全局常量名: `GLOBAL_CONSTANT_NAME` ;全局变量名: `global_var_name` ;实例名: `instance_var_name` ;函数参数名: `function_parameter_name` ;局部变量名: `local_var_name` . 函数名,变量名和文件名应该是描述性的,尽量避免缩写,特别要避免使用非项目人员不清楚难以理解的缩写,不要通过删除单词中的字母来进行缩写. 始终使用 `.py` 作为文件后缀名,不要用破折号.

#### 应该避免的名称

1. 单字符名称, 除了计数器和迭代器,作为 `try/except` 中异常声明的 `e` ,作为 `with` 语句中文件句柄的 `f` .
2. 包/模块名中的连字符(-)
3. 双下划线开头并结尾的名称(Python保留, 例如**init**)

#### 命名约定

1. 所谓"内部(Internal)"表示仅模块内可用, 或者, 在类内是保护或私有的.
2. 用单下划线(_)开头表示模块变量或函数是protected的(使用from module import *时不会包含).
3. 用双下划线(__)开头的实例变量或方法表示类内私有.
4. 将相关的类和顶级函数放在同一个模块里. 不像Java, 没必要限制一个类一个模块.
5. 对类名使用大写字母开头的单词(如CapWords, 即Pascal风格), 但是模块名应该用小写加下划线的方式(如lower_with_under.py). 尽管已经有很多现存的模块使用类似于CapWords.py这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰.

#### 文件名

所有python脚本文件都应该以 `.py` 为后缀名且不包含 `-` .若是需要一个无后缀名的可执行文件,可以使用软联接或者包含 `exec "$0.py" "$@"` 的bash脚本.

#### Python之父Guido推荐的规范

| Type | Public | Internal |
| --- | --- | --- |
| Modules | lower_with_under | _lower_with_under |
| Packages | lower_with_under | |
| Classes | CapWords | _CapWords |
| Exceptions | CapWords | |
| Functions | lower_with_under() | _lower_with_under() |
| Global/Class Constants | CAPS_WITH_UNDER | _CAPS_WITH_UNDER |

| Type | Public | Internal |
| --- | --- | --- |
| Global/Class Variables | lower_with_under | _lower_with_under |
| Instance Variables | lower_with_under | _lower_with_under (protected) or __lower_with_under (private) |
| Method Names | lower_with_under() | _lower_with_under() (protected) or __lower_with_under() (private) |
| Function/Method Parameters | lower_with_under | |
| Local Variables | lower_with_under | |

# Ros

## 包与模块命名规范

全部小写,并以`_`分隔.

```
state_manager
computer_vision
```

## 消息传递

消息的名称以驼峰法命名.

```
nav_msgs/Odometry //里程计
geometry_msgs/PoseWithCovarianceStamped //带协方差矩阵和时间戳的位姿（3D位置，3D方向）
geometry_msgs/TwistWithCovarianceStamped  //带协方差矩阵和时间戳的速度（3D线速度，3D角速度）
sensor_msgs/Imu //惯性导航单元的数据
```

## 代码备份

由于我们的工作场景往往为机器人设备上的机载电脑,存在较高的数据丢失的风险.建议每次调试完成后,将ROS工作空间中的内容同步至本地,并按规定做好版本管理.

---