笔者叠甲

在论文解析部分，笔者对 FAST-LIVO2 的原理进行了展开说明，但是论文了解再通透，不如代码上上手。

本文旨在通过对 FAST-LIVO2 源码进行逐行解析来帮助初入 SLAM 领域的同学更高效了解 FAST-LIVO2。读过源码的同学可能已经发现了，FAST-LIVO2 的源码其实并没有局限于 FAST-LIVO2 论文提及的模块，，比如:激光雷达特征(平面，角点）提取、纯激光雷达（惯性）里程计以及假设深度一致的仿射变换（FAST-LIVO）等等，这部分代码笔者其实也做了解析，但是在整理为文档时，还是决定先不放进来：一是放进来会增加整个解析的篇幅且没有很大的意义，因为这些不是 FAST-LIVO2 的核心贡献；二是这篇代码解析的结构如何合理安排对我来说也是不小的挑战，若考虑这些非核心部分笔者还不知道怎么整，整不好反而让读者更加迷惑。因此，本解析将忠于 FAST-LIVO2 论文，并在核心算法实现部分附上数学原理。

然而，受限于个人理解水平与实践经验，笔者对部分设计细节、工程技巧乃至算法意图的解读，难免存在偏颇、疏漏或未能完全领会其精妙之处。在此，我怀着学习和交流的初衷，将这份代码解析呈现给各位同行与爱好者。**恳请各位前辈、专家和读者不吝赐教，对文中任何理解不当、表述不清或有失偏颇之处给予批评指正**。您的宝贵意见不仅能帮助笔者深化理解，也能让这份解析更加准确和完善，从而更好地服务于社区的学习与交流。

解析概要

在进行解析之前，笔者先带领大家看看 FAST-LIVO2 代码包的主要结构，如图 1 所示。

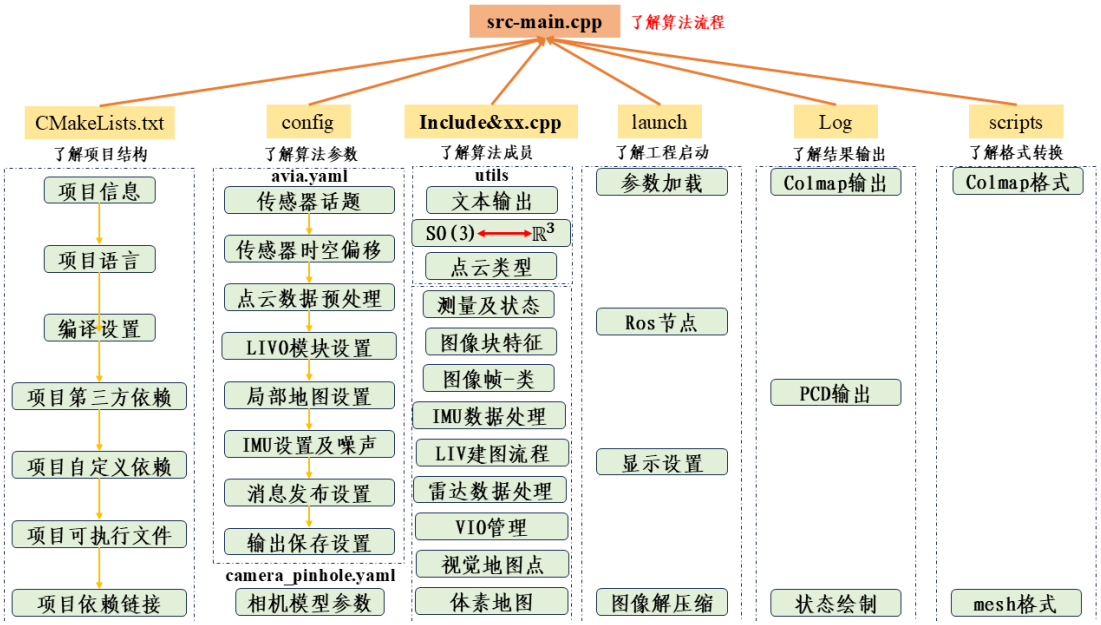

图 1 FAST-LIVO2 代码包结构

可以看到代码包结构被我用总分的形式重新组织了一下，其中 src-main.cpp（笔者将其称为主材料）是算法运行的入口，之后我们将从这里一层一层剥开它的芯。其余的六个文件（夹）都是为算法服务的（笔者将其称为支撑材料）：CmakeLists.txt 确保了项目顺利编译（跨平台）、config 给算法运行提供了必要参数、Include&xx.cpp 为算法各模块的具体实现、launch 确保系统正常使用 ros 协议、Log 存放了系统运行过程中计算的结果、scripts 为系统的扩展应用提供必要的格式转换脚本。

笔者拟将解析分为两个阶段：预热阶段以及真解阶段。

预热阶段：笔者将对六个支撑性文件进行酌情解析，在这个阶段其中笔者将不会对 include&xx.cpp 进行逐个函数详细解析，因为这个阶段一些读者还不了解它们在系统运行中的实际作用，详细地解析反而不好理解（纸上谈兵）。

真解阶段：笔者将跟随 FAST-LIVO2 的运行步骤，从 main.cpp 开始打通整个系统，并对运行过程中遇到的所有函数进行逐行级注释以及必要的数学原理关联。

然而，在进行 word 编写的时候，笔者发现很难通过文档对函数进行一层一

层地展开，毕竟函数是一个套一个，在尝试后便果断放弃该方式，并决定采用思维导图的形式完成真解阶段。详情请参见代码解析-真解阶段。

一、预热阶段

1.CmakeLists.txt

1.1.项目信息

```
cmake_minimum_required(VERSION 2.8.3) // 设置 Cmake 编译所需的最小版本
project(fast_livo) // 设置项目（整个工程）的名称为 fast_livo
set(CMAKE_BUILD_TYPE "Release") // 构建类型"Release"类型通常用于生成优化过的最终版本，该版本会进行代码优化以提高运行效率，但不会包含调试信息。
message(STATUS "Build Type: ${CMAKE_BUILD_TYPE}") // 用于输出当前设置的构建类型。
message 命令用于在 CMake 配置过程中打印消息，STATUS 参数表示这是一条状态消息。
${CMAKE_BUILD_TYPE}是一个变量，它存储了当前设置的构建类型。因此，这行代码会输出类似于"Build Type: Release"的消息。
```

1.2.项目语言

```
set(CMAKE_CXX_STANDARD 17) // 设置了 C++编译标准为 C++17
set(CMAKE_CXX_STANDARD_REQUIRED ON) // 指定 C++17 标准是必需的。如果编译器不支持 C++17，CMake 将会报错并终止配置过程。
set(CMAKE_CXX_EXTENSIONS OFF) //关闭了编译器的扩展特性。一些编译器可能会提供超出标准 C++的额外特性或扩展，这些特性虽然有时很有用，但可能不被其他编译器支持，降低代码的可移植性。
```

1.3.编译设置

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -fexceptions") // 为编译器添加多线程支持（-pthread）和异常处理支持（-fexceptions）。即程序在运行时可以使用多线程（比如 std::thread、pthread_create 等）-pthread 是“你自己写多线程代码”用的
# Specific settings for Debug build
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0 -g")// 确保在 Debug 模式下编译时，生成的可执行文件包含调试信息，并且不会被优化，从而方便开发者进行调试
# Detect CPU architecture  检查当前 CPU 架构（部署平台），根据不同的平台
message(STATUS "Current CPU architecture: ${CMAKE_SYSTEM_PROCESSOR}")
# Specific settings for Release build
if(CMAKE_SYSTEM_PROCESSOR MATCHES "^(arm|aarch64|ARM|AARCH64)")
  if(CMAKE_SYSTEM_PROCESSOR MATCHES "aarch64")
    # 64-bit ARM optimizations (e.g., RK3588 and Jetson Orin NX)
    set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3 -mcpu=native -mtune=native -ffast-math")// -O3 表示开启最高级别的优化，通常会让程序运行得更快，但可能会增加编译时间和可执行文件体积。-mcpu=native 和 -mtune=native 让编译器针对当前主机 CPU 的特性进行优化，充分利用硬件能力。-ffast-math 允许编译器对浮点运算进行更激进的优化，提升性能，但可能会牺牲一部分计算精度和标准兼容性。-mfpu=neon 启用 ARM 架构下的 NEON 浮点运算单元，利用 SIMD 指令集加速浮点计算，适合需要高性能计算的场景。
    message(STATUS "Using 64-bit ARM optimizations: -O3 -mcpu=native -mtune=native -ffast-math")
  else()
    # 32-bit ARM optimizations with NEON support
```

```
    set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3 -mcpu=native -
mtune=native -mfpu=neon -ffast-math")
    message(STATUS "Using 32-bit ARM optimizations: -O3 -mcpu=native -mtune=native -
mfpu=neon -ffast-math")
  endif()
  add_definitions(-DARM_ARCH)
else()
  # x86-64 (Intel/AMD) optimizations
  set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3 -march=native -
mtune=native -funroll-loops") //-funroll-loops 让编译器自动展开循环体，减少循环控制开销，有助于提升某些循环密集型代码的执行速度。
  message(STATUS "Using general x86 optimizations: -O3 -march=native -mtune=native -
funroll-loops")
  add_definitions(-DX86_ARCH) // 相当于在所有源文件的开头自动插入 #define X86_ARCH。这样做的目的是让代码能够通过条件编译（如 #ifdef X86_ARCH）判断当前是否为 x86 架构，从而根据不同平台执行不同的代码逻辑或启用特定优化。
endif()
# Define project root directory
add_definitions(-DROOT_DIR=\"${CMAKE_CURRENT_SOURCE_DIR}/\") //为所有源文件添加一个名为 ROOT_DIR 的宏定义，并将其值设置为当前 CMake 项目的源代码根目录路径。
# Detect CPU core count for potential multithreading optimization
include(ProcessorCount) // 检测当前系统的 CPU 核心数
ProcessorCount(N)
message(STATUS "Processor count: ${N}")
# Set the number of cores for multithreading
if(N GREATER 4) //说明当前系统有超过 4 个核心，适合进行多线程优化
  math(EXPR PROC_NUM "4") //表示多线程时最多使用 4 个核心。
  add_definitions(-DMP_EN -DMP_PROC_NUM=${PROC_NUM}) //所有源文件添加两个宏定义：MP_EN（表示启用多线程）和 MP_PROC_NUM=4（指定多线程使用的核心数）。
  message(STATUS "Multithreading enabled. Cores: ${PROC_NUM}")
elseif(N GREATER 1)
  math(EXPR PROC_NUM "${N}")
  add_definitions(-DMP_EN -DMP_PROC_NUM=${PROC_NUM})
  message(STATUS "Multithreading enabled. Cores: ${PROC_NUM}")
else()
  add_definitions(-DMP_PROC_NUM=1)
  message(STATUS "Single core detected. Multithreading disabled.")
endif()
# Check for OpenMP support
find_package(OpenMP QUIET) // 注意跟前面的-pthread 进行区别，这可让程序自动进行并行运算：比如用 #pragma omp parallel for 让循环等代码自动并行。
if(OpenMP_CXX_FOUND)
  message(STATUS "OpenMP found")
  add_compile_options(${OpenMP_CXX_FLAGS})
```

```
else()
  message(STATUS "OpenMP not found, proceeding without it")
endif()
# Check for mimalloc support
```

更快的分配和释放速度：相比标准的内存分配器，mimalloc 在多线程和高并发场景下表现更优，能减少内存分配带来的性能瓶颈。
更低的内存碎片：mimalloc 采用了高效的内存管理算法，能有效减少内存碎片，提高内存利用率。
线程友好：它为每个线程分配独立的内存区域，减少线程间的锁竞争，提升多线程程序的性能。
易于集成：只需在链接时加上 mimalloc 库，无需大幅修改代码即可获得性能提升。

```
find_package(mimalloc QUIET) // 检测并配置 mimalloc 内存分配库的支持
if(mimalloc_FOUND)
  message(STATUS "mimalloc found")
else()
  message(STATUS "mimalloc not found, proceeding without it")
endif()
```

1.4. 第三方依赖

```
find_package(catkin REQUIRED COMPONENTS // 是 ROS 项目 CMakeLists.txt 文件中的标准写法。它会查找 catkin 包，并确保指定的所有组件都能被找到，否则会报错终止配置过程。
  geometry_msgs  //几何信息
  nav_msgs  //导航消息
  sensor_msgs //传感器消息
  roscpp
  rospy //是 ROS 的 C++ 和 Python 客户端库，支持用这两种语言开发 ROS 节点。
  std_msgs // 标准消息
  pcl_ros //为点云库（PCL）与 ROS 的相互转换提供支持
  tf //用于坐标变换管理，支持机器人系统中不同坐标系之间的转换。
  message_generation // 用于自动生成自定义的消息类型和服务代码
  eigen_conversions //提供 Eigen（线性代数库）与 ROS 消息类型之间的数据转换工具
  vikit_common
  vikit_ros //通常是视觉惯性工具包（VI-KIT）相关的 ROS 支持包，提供视觉的通用模型
  cv_bridge//实现 OpenCV 图像与 ROS 图像消息之间的互相转换，便于图像处理。
  image_transport //优化 ROS 图像消息的发布和订阅，支持多种压缩和传输方式
)

find_package(Eigen3 REQUIRED) //查找 Eigen3 线性代数库
find_package(PCL REQUIRED) // 查找点云库（Point Cloud Library，PCL），用于处理三维点云数据
find_package(OpenCV REQUIRED) // 查找 OpenCV 计算机视觉库，支持图像处理、特征提取等功能。
find_package(Sophus REQUIRED) //查找 Sophus 库，主要用于李群李代数（如 SE(3)、SO(3)）的数学运算
find_package(Boost REQUIRED COMPONENTS thread) //查找 Boost 库，并且要求包含 thread 组件，提供跨平台的多线程支持
```

```
set(Sophus_LIBRARIES libSophus.so)// 手动指定了 Sophus 动态库，保证相关功能能够正常链接和使用，感谢郑博修复了这个 bug。
# Define the catkin package
catkin_package( 声明和导出包的依赖关系，便于其他 ROS 包在编译和运行时正确使用本包，也就是说 catkin_package 让你的包变成了一个“标准的 ROS 组件”，别人可以像用标准 ROS 包一样方便地用你的包，只需要在自己的 CMakeLists.txt 里写 find_package(你的包名 REQUIRED)，无需手动配置各种依赖路径和库，非常方便团队协作和模块复用。
  CATKIN_DEPENDS geometry_msgs nav_msgs roscpp rospy std_msgs message_runtime cv_bridge vikit_common vikit_ros image_transport // CATKIN_DEPENDS 后面列出了本包依赖的其他 ROS 包
  DEPENDS EIGEN3 PCL OpenCV Sophus // DEPENDS 后面列出了本包依赖的非 ROS 第三方库
)

# Include directories for dependencies
include_directories( // 在编译时可以顺利找到所有依赖库和自定义头文件
  ${catkin_INCLUDE_DIRS}
  ${EIGEN3_INCLUDE_DIR}
  ${PCL_INCLUDE_DIRS}
  ${OpenCV_INCLUDE_DIRS}
  ${Sophus_INCLUDE_DIRS}
  Include // 项目根目录下的 include 文件夹，通常用于存放本项目自定义的头文件。
)
```

1.5. 自定义依赖

```
add_library(vio src/vio.cpp src/frame.cpp src/visual_point.cpp) // 创建名为 vio 的库，包含视觉惯性相关的三个源文件
add_library(lio src/voxel_map.cpp)// 创建名为 lio 的库
add_library(pre src/preprocess.cpp) // 创建名为 pre 的库，负责数据预处理
add_library(imu_proc src/IMU_Processing.cpp) //创建名为 imu_proc 的库，处理 IMU（惯性测量单元）数据
add_library(laser_mapping src/LIVMapper.cpp) // 创建名为 laser_mapping 的库，整体流程
```

1.6. 可执行文件

```
add_executable(fastlivo_mapping src/main.cpp) // 主函数（整个系统入口）作为可执行文件
```

1.7. 链接所需依赖

```
target_link_libraries(fastlivo_mapping  // 将前面的所有依赖都链接到这个可执行文件上。
  laser_mapping
  vio
  lio
  pre
  imu_proc
  ${catkin_LIBRARIES}
  ${PCL_LIBRARIES}
```

```
  ${OpenCV_LIBRARIES}
  ${Sophus_LIBRARIES}
  ${Boost_LIBRARIES}
)
# Link mimalloc if found
if(mimalloc_FOUND)
  target_link_libraries(fastlivo_mapping mimalloc)
endif()
```

2. config

本文以 avia.yaml 和 camera_pinhole.yaml 两个配置文件为例。

2.1. avia.yaml

2.1.1. 传感器话题

```
common:
  img_topic: "/left_camera/image" //相机话题。注意如果 bag 中的图像话题为 Compressed 格式，这里的图像话题实际上是被解压后的 raw 格式话题，launch 文件启动 ros 节点后会继续启动 image_transport 工具包将图像转为 raw 格式，再被系统订阅。
  lid_topic: "/livox/lidar" //avia 的自定义消息类型的话题，bag 中的话题是无法直接用 rviz 看到的。频率在 10hz.
  imu_topic: "/livox/imu" //avia 内嵌的 Imu，频率在 200hz
  img_en: 1 // 是否使用相机（可选）
  lidar_en: 1 // 是否使用激光雷达（必选）
  ros_driver_bug_fix: false  // 激光雷达驱动的 bug，会导致一帧激光雷达点云内的 Imu 时间戳与激光雷达帧头时间戳出现至少 0.5s 的偏差。一般不用管，如果使能，imu_cbk 函数会把偏差值加到当前的 Imu 时间戳上，进行补偿。
```

2.1.2. 传感器时空偏移

```
extrin_calib:
  extrinsic_T: [0.04165, 0.02326, -0.0284] //激光雷达系相对 imu 系的平移
  extrinsic_R: [1, 0, 0, 0, 1, 0, 0, 0, 1] //激光雷达系相对 imu 系的旋转
  Rcl: [0.00610193,-0.999863,-0.0154172,
        -0.00615449,0.0153796,-0.999863,
        0.999962,0.00619598,-0.0060598]  // 激光雷达系相对相机系的旋转
  Pcl: [0.0194384, 0.104689,-0.0251952] //激光雷达系相对相机系的平移

time_offset:
  imu_time_offset: 0.0 // 激光雷达与 Imu 时间戳之间的偏差
  img_time_offset: 0.1 //激光雷达与 IMU 时间戳之间的偏差。这个时间为什么是 0.1 笔者在论文解析的 扫描重合并 一节解释过了，这里就不再赘述。
  exposure_time_init: 0.0 // 初始的相机曝光时间，待估计
```

2.1.3. 点云数据预处理

```
preprocess:
```

```
  point_filter_num: 1  //这里其实是选点的间隔，每搁多少个点选一个点处理，作为系统的待处理点云
  filter_size_surf: 0.1 //体素滤波设置 0.1m*0.1m*0.1m
  lidar_type: 1 # Livox Avia LiDAR
  scan_line: 6 //avia 雷达的激光线束个数
  blind: 0.8 //avia 雷达的盲区
```

2.1.4. LIVO 模块设置

```
lio:
  max_iterations: 5 // ESIKF 中的更新迭代次数
  dept_err: 0.02 // 激光雷达的深度误差
  beam_err: 0.05 //激光雷达的方向误差
  min_eigen_value: 0.0025 # 0.005 //平面奇异值分解后的最小特征上限（阈值）。
  voxel_size: 0.5 // 体素块的大小 0.5*0.5*0.5。有读者可能混淆 filter_size_surf 中设置的体素尺寸，具体就是：系统先将每帧点云用 0.1*0.1*0.1 的体素进行滤波，再用 0.5*0.5*0.5 的体素划分地图。
  max_layer: 2 //  八叉树的最大层数
  max_points_num: 50 //每个节点的最大存点量
  layer_init_num: [5, 5, 5, 5, 5] // 每个节点的最小存点量。
vio:
  max_iterations: 5
  outlier_threshold: 1000 # 78 100 156 #100 200 500 700 infinite // 每个像素块的阈值
  img_point_cov: 100 # 100 1000 // VIO 观测方程（光度误差）的协方差，也可以说是像素点噪声协方差。
  patch_size: 8 // 图像块的尺寸
  patch_pyrimid_level: 4 //金字塔层数
  normal_en: true // 是否细化法向量，决定是否能够从单应阵获得仿射变换阵
  raycast_en: false //是否进行体素投射
  inverse_composition_en: false // 是否进行逆补偿，SVO 采用的，Fast-livo2 默认不使用（逆补偿下的状态更新考虑的没有 Fast-livo2 全面）
  exposure_estimate_en: true //进行曝光时间补偿
  inv_expo_cov: 0.1 //逆曝光时间协方差
```

2.1.5. 局部地图设置

```
local_map:
  map_sliding_en: false // 是否进行滑动地图，当内存较小时使用。
  half_map_size: 100 //地图边长的一半
  sliding_thresh: 8 //每 8 米滑动一次
```

2.1.6. IMU 设置及噪声

```
imu:
  imu_en: true // 使用 Imu
  imu_int_frame: 30 //初始化所需要的 IMU 帧数（Fast-LIVO2 需要静止初始化）
  acc_cov: 0.5 # 0.2 //加速度每个轴的方差
  gyr_cov: 0.3 # 0.5 //陀螺仪每个轴的方差
  b_acc_cov: 0.0001 # 0.1 //加速度每个轴偏差的协方差
```

```
  b_gyr_cov: 0.0001 # 0.1 //陀螺仪每个轴偏差的协方差
```

2.1.7. 消息发布设置

```
publish:
  dense_map_en: true // 是否进行稠密建图
  pub_effect_point_en: false // 是否发布有效点
  pub_plane_en: false //是否发布平面点
  pub_scan_num: 1 //每次发布的累积点云帧
  blind_rgb_points: 0.0 //盲区内的 RGB 点值为 0
```

2.1.8. 输出保存设置

```
pcd_save:
  pcd_save_en: false //是否保存 PCD
  colmap_output_en: false # need to set interval = -1 //是否输出 COLMAP 格式
  filter_size_pcd: 0.15 //对 PCD 文件进行降采样
  interval: -1 //每个 PCD 文件的累积点云帧数
```

2.1.9. UAV 实验设置

```
uav: //用于论文中的 UAV 导航实验
  imu_rate_odom: false //是否按 imu 频率发布里程计
  gravity_align_en: false //是否进行重力对齐
```

2.2. camera_pinhole.yaml

3. mapping_avia.launch

```
<launch>
<!-- Launch file for Livox AVIA LiDAR -->
    <arg name="rviz" default="true" />  // 是否 RVIZ 可视化
    <rosparam command="load" file="$(find fast_livo)/config/avia.yaml" /> //加载系统配置文件到 ros 空间
    <node pkg="fast_livo" type="fastlivo_mapping" name="laserMapping" output="screen"> //设置 ros 节点：项目名称，可执行文件名称，ros 节点名称，调试信息输出到终端。
        <rosparam file="$(find fast_livo)/config/camera_pinhole.yaml" /> //加载相机参数。
    </node>
    <group if="$(arg rviz)">
        <node launch-prefix="nice" pkg="rviz" type="rviz" name="rviz" args="-d $(find fast_livo)/rviz_cfg/fast_livo2.rviz" /> //拉起 RVIZ 显示
    </group>
    <node pkg="image_transport" type="republish" name="republish" args="compressed in:=/left_camera/image raw out:=/left_camera/image" output="screen" respawn="true"/> // 拉起图像格式转换，将压缩格式转换为原始格式。
launch-prefix="gdb -ex run --args" //表示在启动节点时，先用 gdb（GNU 调试器）加载程序，并自动执行 run 命令。这样可以在调试环境下运行节点，便于排查崩溃、断点调试等。
launch-prefix="valgrind --leak-check=full" //表示用 valgrind 工具启动节点，并开启详细的内存泄漏检查。这样可以检测程序运行过程中的内存泄漏和相关问题。
```

```
</launch>
```

4. Log

主要存放一些输出的结果，若要进行 Nerf 或 3D-GS 则需要 colmap 格式的数据，作者已经帮我们做好了。

plot.py 主要用于绘制 imu 的各轴数据（笔者这里推荐一个工具 **plotjuggler**，可直接绘制各种格式文件中的数据，包括 rosbag，点云，图像等均可。）

5. scripts

两个脚本用于输出 colmap 格式的数据和 mesh 格式的数据。

6. include&xx.cpp

Utils 文件夹中的三个头文件主要是设置文本输出形式 color.h、流形空间到欧式空间的互转函数 so3_math.h 以及各个点云类型 types.h。

其余的头文件可参考图一了解，声明了各模块对应的函数，对应的 xx.cpp 文件则是这些函数的实现。各位读者知道这么个事就行，笔者将在走通整个流程的时候解析各函数，这样最能够理解其意义。