Cartographer 源码分析

朱海民 整理

2019-4-30

目录

第 0 章 源码阅读—预准备

0.1、算法简介

cartographer 是 google 推出的一套基于图优化的 SLAM 算法。正如邵总所说（邵天兰：如何看待谷歌在 10 月 6 日开源的 SLAM 算法 cartographer），cartographer 算法并没有给人惊艳的感觉，但该算法的主要目标是实现低计算资源消耗，达到实时 SLAM 的目的。因此，其代码实现也是非常值得学习的。

该算法主要分为两个部分，第一个部分称为 Local SLAM，该部分通过一帧帧的 Laser Scan 建立并维护一系列的 Submap，而所谓的 submap 就是一系列的 Grid Map。当再有新的 Laser Scan 时，会通过 Ceres Scan Matching 的方法将其插入到子图中的最佳位置。但是 submap 会产生误差累积的问题，因此，算法的第二个部分，称为 Global SLAM 的部分，就是通过 Loop Closure 来进行闭环检测，来消除累积误差：当一个 submap 构建完成，也就是不会再有新的 laser scan 插入到该 submap 时，算法会将该 submap 加入到闭环检测中。闭环检测的本质也是一个优化问题，该优化问题被表达成了一个 pixel-accurate match 的形式，解决优化问题的方法是 Branch-and-Bound Approach。

算法原理不再详细展开，有时间我们再专门开一个帖子来说。这部分我没有参见其他人的解读，既然有论文，学习的最佳方式当然是自己去读原文啊。论文原文可参见：

W. Hess, D. Kohler, H. Rapp, and D. Andor, Real-Time Loop Closure in 2D LIDAR SLAM, in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016. pp. 1271 - 1278.

0.2、cartographer 的安装

1. System Requirements

- 64bit, modern CPU;
- 16GB RAM
- Ubuntu 14.04(Trusty) 或 16.04（Xenial）（我选择的是 16.04.01 版本的，好用的镜像链接：http://old-releases.ubuntu.com/releases/16.04.4/）
- gcc version 4.8.4 或 5.4.0

2. ROS 的安装

目前支持 Indigo 和 Kinetic 两个版本。我选择的是较新的 Kinetic 版本。

ROS 的安装请参见：http://wiki.ros.org/kinetic/Installation/Ubuntu 或 https://www.ncnynl.com/archives/201801/2273.html

3. Cartographer 的安装

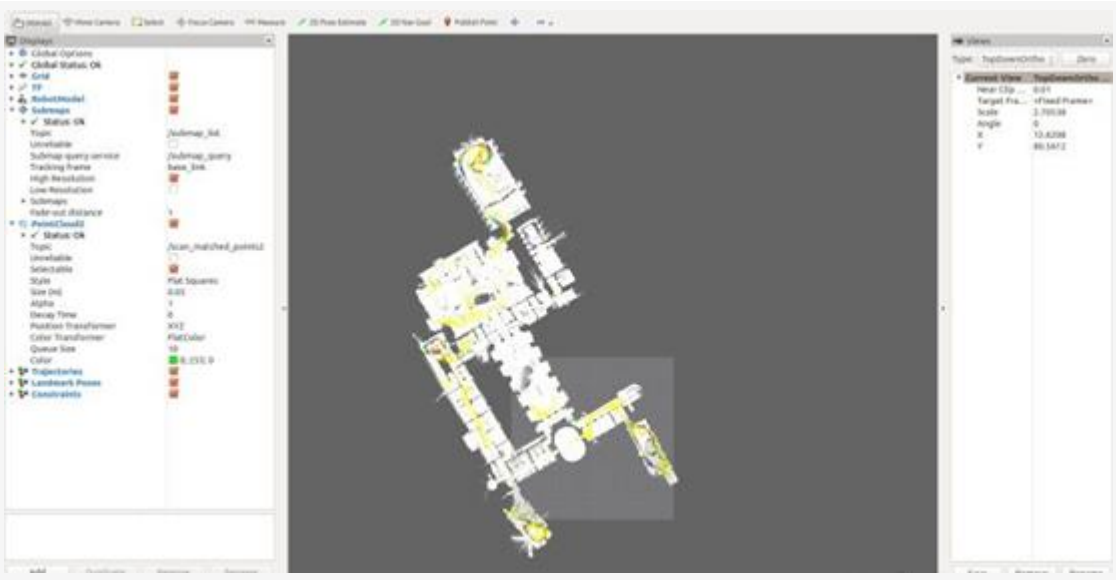
主要参考：https://google-cartographer-ros.readthedocs.io/en/latest/

cartographer 的安装主要包括三个部分：cartographer、cartographer ros、ceres-solver. 如果完全按照官网的方法，前两个部分没什么问题，但如果自己的电脑不能翻墙的话，ceres-solver 就会出现错误。因为 ceres-solver 的下载地址是被墙了的。所以最后我是看下面这个链接里的教程解决的，主要就是将其中 ceres-solver 的下载链接换成了没有被墙的 github。具体看下面链接：
https://blog.csdn.net/rjasdl128hf/article/details/79888305

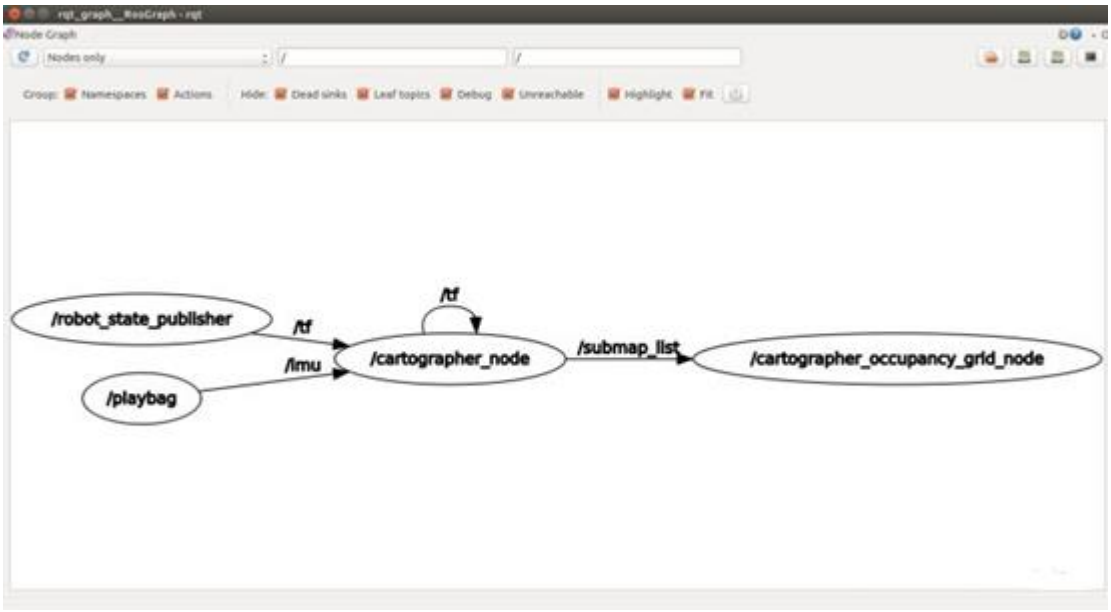
另外，同样因为被墙的原因，官网上提供的测试数据也下载不下来。我最后找到了一个可以下载下来的链接，见下面这个链接里的“5. 雷达数据集”。有时候可能一次下不成功，多试几次，偶尔一次就能成功了。测试数据集的下载：https://blog.csdn.net/qq_24894159/article/details/81175656

0.3、测试结果

相信通过上面分享的步骤，应该不难完成 cartographer 的安装和测试。而这只是 cartographer 学习的开始。在之后的文章中我们再详细解读源码，并在解读源码的过程中详述算法的公式及原理（抱歉，因为个人性格原因，记录笔记会非常详细，乃至有些啰嗦。但我想这样更适用于初学者）。下面仅贴出我跑通后的建图结果。2D 和 3D 都测试通过了，但 3D 结果忘记保存了。



- 图 1：cartographer 2D 建图数据集测试结果



- 图 2：cartographer 运行时的 ROS 节点和 Topic 等情况。

第 1 章 整体框架介绍

从本文开始，我们会开始对 cartographer 源码的阅读。我会试图给出逐行或逐块的注释，来说明代码在做什么以及如何跳转。通常，我阅读一个大的 project 的源码最困难的地方就是难以理解程序跳转来跳转去的关系。

对于 C++的很多新特性不太了解，所以在学习过程中，对于很多大神已经见怪不怪的 C++的问题可能也需要学习一番。这些结果我也会以注释的形式写出来，帮助同样是不太了解 C++的同学。其中也难免会有理解错误的地方，请各位朋友斧正。大神见到这些很初级的东西请勿喷！

本文参考的主要博文有如下链接：

- 官方文档：https://google-cartographer-ros.readthedocs.io/en/latest/，最权威的还是官网，但不够详细，需要配合其他博主的解读来学习。
- CSDN 博客：泡泡机器人原创专栏-Cartographer 理论及实现浅析 - 北漠苍狼的专栏　这篇文章可以从整体上对 cartographer 源码的框架有些了解。
- 这篇博客简要介绍了 cartographer 程序的先后跳转关系，本文也是以该博客为提纲来进行代码阅读的，但还是太粗框架了，所以我希望能写一个详细版的解读，这样也有助于自己 coding 水平的提升：Cartographer 代码阅读分析　和　Cartographer 代码阅读分析-2
- 分块对各个部分的源码分析：【slamcode 的博客 】cartographer 源码分析 - CSDN 博客。这篇博客已经介绍的非常详细了。有兴趣的同学可以去看看。基本上这是我看到的对源码分析最详细的了，其他的都只是一些对安装和测试或运行自己实例的 demo。但为什么我还要写本文呢？因为上面这个博客是从各个模块分块来写的，详细地讲述了每个文件的函数作用，但是却没有梳理清楚程序的跳转关系。因此，本文希望从程序的跳转关系为逻辑线来梳理一下程序。也有助于我们了解这个程序运行起来以后都做了什么。建议读者可将本文和该博客搭配观看。阅读本文时可将上述博客看做一个手册，来查询每个函数或代码块做了什么。
- 对其中一个关键函数 RealTimeCorrelativeScanMatcher2D 的解读：cartographer RealTimeCorrelativeScanMatcher2D 整理
- 从 node_main 入口函数开始：cartographer 源码浅析（一）-新鲜出炉 - liuxiaofei823 的博客 - CSDN 博客

后面两个仅作为参考吧。如果没时间可以跳过。

1.1、代码框架

首先，我们也要先来关注一下代码的整体框架，对整个代码先有一个整体的了解。

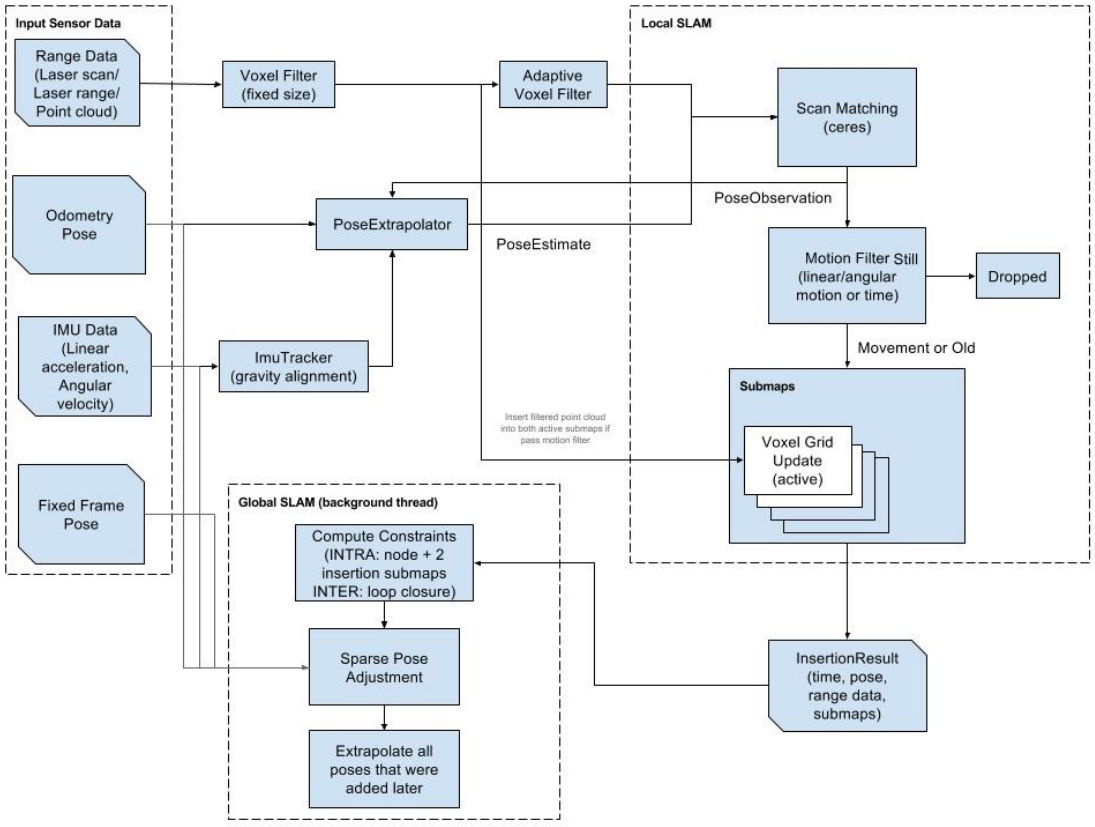
cartographer 的代码主要包括两个部分：cartographer 和 cartographer_ros。而之前安装的 ceres-solver 是一个非线性优化的库。我们暂时可以不用管它，只需要知道当我们解决优化问题的时候可以调用这个库来求解即可。

cartographer 主要负责处理来自 Laser、IMU、Odemetry 等传感器的数据并基于这些数据进行地图的构建。所以这一部分应该说是 cartographer 代码的核心部分。而 cartographer_ros 是基于 ros 的通信机

制获取传感器的数据并将他们转换成 cartographer 中定义的格式传递给 cartographer 处理，与此同时也将 cartographer 的处理发布用于显示或保存。是基于 cartographer 的上层应用。所以，cartographer_ros 相当于是对 cartographer 进行了一层 ROS 接口的封装。根据官网的说明来看，即使不采用 ROS，也可以运行 cartographer 算法。只不过这时候传感器传递数据可能就需要自己去写函数处理了。

题外话：ROS 中的 Topic 和 Service 等通信机制给我们提供了标准的模式，可以省去我们很多的工作。所以，还是非常鼓励诸位机器人学方向的工作者都能向 ROS 靠近。本人在 2014 年左右 ROS 刚刚兴起时曾经翻译过官方 tutorial 的翻译，版本较早，但近期对照了一些发现变化不大，不影响大家学习。通过该文档可以对 ROS 中的一些基本概念，如：Node、Message、Topic、Service、Publisher、Subscriber 等有所了解。所以感兴趣的同学可以参见本文的这篇翻译：ROS 学习笔记：Beginner 阶段_图文_百度文库。

整个算法的流程如下图所示：



通过阅读 cartographer 的官网我们可以知道，cartographer 整体可以分为两个部分：

一是 Local SLAM 部分，也被称为 Frontend；该部分的主要任务是建立维护 Submaps，但问题是该部分建图误差会随着时间累积。该部分相关的参数定义在 /src/cartographer/configuration_files/trajectory_builder_2d.lua 和 /src/cartographer/configuration_files/trajectory_builder_3d.lua 中。

二是 Global SLAM 部分，也称为 Backend. 该部分的主要任务是进行 Loop Closure。如前所述，闭环检测本质上也是一个优化问题，文中将其表达成了 pixel-accurate match 的形式，采用 Branch-and-Bound Approach （BBA）的方法来解决。具体怎么用 Branch-and-Bound 方法，可以参见论文（Real-Time Loop Closure in 2D LIDAR SLAM）。如果是 3D 情况下，该部分还负责根据 IMU 数据找出重力的方向。

总体而言，Local Slam 部分负责生成较好的子图，而 Global Slam 部分进行全局优化，将不同的子图以最匹配的位姿 tie 在一起。

可以看出，从传感器出发，Laser 的数据经过两个滤波器后进行 Scan Matching，用来构建子图 submap，而新的 Laser Scan 进来后也会插入到已经维护着的子图的适当位置。如何决定插入的最优位姿，就是通过 Ceres Scan Matching 来实现的。估计出来的最优位姿也会与里程计和 IMU 数据融合，用来估计下一时刻的位姿。

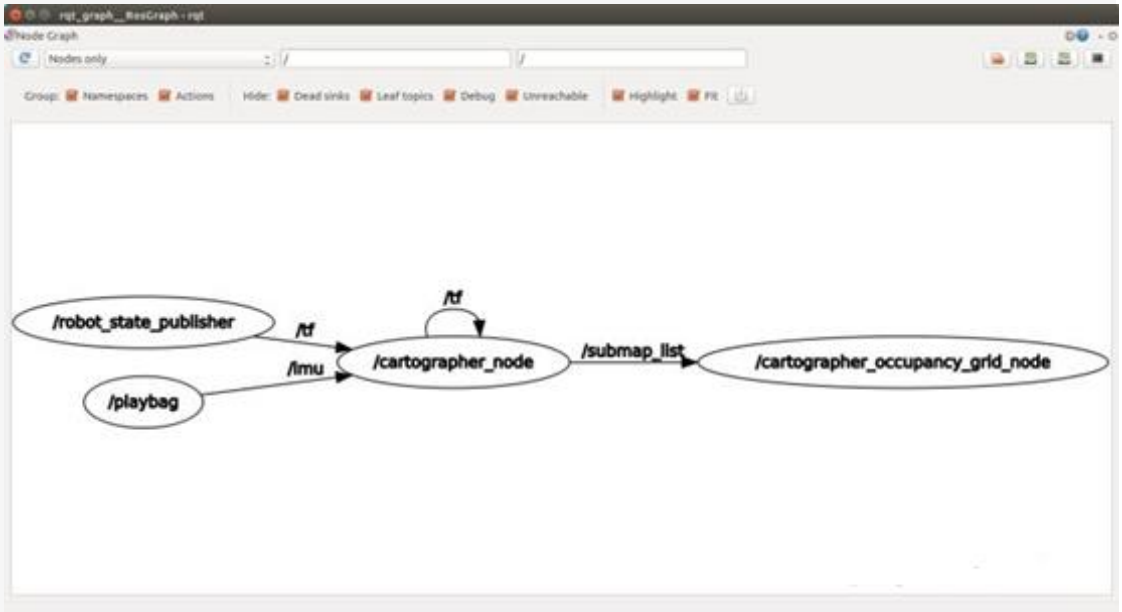
看到很多人对 Global Slam 里面的 BBA 算法不太懂，而现有资料也很少将这个方法。所以我这里尝试讲一下我自己的理解，不保证完全正确。仅供大家参考：

简单来说，就是把搜索窗口表达成一颗树的形式，树上每一个子节点形成了一个对父节点的所有可能性的划分。所以，父节点代表了该子树下的所有解的可能性。对每个节点都有一个得分 score(c)，该分值是这个子树上所有解的得分可能性的一个上限（Bound）。因此，如果已经有一个解能比该节点的得分 score(c)好，那么最优解肯定不存在该子树上，因此该子树就可以被剪枝，从而大大加快解问题的速度。而在实际问题中，实际上大多数分支（不是最优解的那些分支）的得分是非常低的，通过一个阈值就直接剪枝掉了很多子树。因此，相比全局搜索窗口而言，BBA 方法实时性提高了很多。但有得必有失，BBA 方法获得了时间效率的提升，带来的肯定就是存储效率的降低。前面提到需要计算节点的得分 score。而得分 score 如何计算呢？非常暴力，基本上就是预先计算好子图上每个 pixel 的得分值，直接存储下来，当输入一个节点时，直接 lookup 即可。这基本上就是在拿存储空间来换时间效率。我想这也是为什么这个程序这么吃内存的一个原因。

上述讲解中如何 Branch 我没有讲。这部分应该还是比较好理解的，读者直接看论文原文就可以了。

接下来介绍一下 cartographer 运行起来后系统中存在的 ROS 节点、Topic 和 Service 的情况： 该部分主要参考来自于官网：ROS API

这里我先放一张我运行测试时通过 rqt_graph 画出来的节点关系图：



其中，比较重要的节点有两个：cartographer_node 和 cartographer_occupancy_grid_node. 其中 cartographer_node 订阅传感器数据，运行算法，生成 submap_list，而 cartographer_occupancy_grid_node 则订阅了 submap_list，并发布栅格地图。

下面是根据官网整理出来的节点及节点相关的 Topic、Message 和 Service。

1. cartographer_node

|===@订阅的 Topic
|---------scan (sensors_msgs/LaserScan): 解释：传感器数据
|---------echoes (sensors_msgs/MultiEchoLaserScan) 解释：没太看明白这个是做什么用，感觉可能是设置如果有多个 Laser 传感器的情况
|---------points2 (sensors_msgs/PointCloud2) 解释：2D 点云数据
|---------imu (sensors_msgs/Imu) 解释：IMU 的数据
|---------odom (nav_msgs/Odometry) 解释：里程计数据
|===@发布的 Topic
|---------scan_matched_points2 (sensors_msgs/PointCloud2) 解释：匹配好的 2D 点云数据，用来 scan-to-submap matching
|---------submap_list (cartographer_ros_msgs/SubmapList) 解释：发布构建好的 submap。
|===@提供的 Service
|---------submap_query (cartographer_ros_msgs/SubmapQuery) 解释：可以提供查询 submap 的服务，获取到 request 的 submap
|---------start_trajectory (cartographer_ros_msgs/StartTrajectory) 解释：维护一条轨迹
|---------finish_trajectory (cartographer_ros_msgs/FinishTrajectory) 解释：Finish 一个给定 ID 的轨迹
|---------write_state (cartographer_ros_msgs/WriteState) 解释：将当前状态写入磁盘文件中
|---------get_trajectory_states (cartographer_ros_msgs/GetTrajectoryStates) 解释：获取指定 trajectory 的状态
|---------read_metrics (cartographer_ros_msgs/ReadMetrics)
|===@Required tf Transforms
|===@Provided tf Transforms

这个有误，应该是：用于匹配的PointCloud2类型的点云（可以2D也可以3D）

2. offline_node
|===可以理解为一个快速版本的 cartographer
|===不监听任何 topic,二是直接从数据包中读取传感器数据。发布的 Topic 与 cartographer_node 相同，除此以外，还有：
|===@额外发布的 Topic
|---------~bagfile_progress (cartographer_ros_msgs/BagfileProgress) 解释：可查询处理包的进度等情况
|===@Parameters
|---------~bagfile_progress_pub_interval(double, default=10.0) 解释：发布包数据的时间间隔。以 s 为单位；

3. occupancy_grid_node
|===description：主要任务是监听 submap_list 这个 Topic 然后构建栅格地图并发布
|===@订阅的 Topics
|---------submap_list (cartographer_ros_msgs/SubmapList) 解释： 由 cartographer_node 发布
|===@发布的 Topics
|---------map (nav_msgs/OccupancyGrid) 解释：栅格地图

4. pbstream_map_publisher
|===description: 快速版的 occupancy_grid_node

|===未订阅任何节点
|===发布的 Topics
|---------map (nav_msgs/OccupancyGrid) 解释：栅格地图

前面我们一直在梳理的是算法的逻辑框架，一点儿也没接触到代码本身。没办法，一个大的 project 代码非常复杂，我们只有先理清这些逻辑框架了才可能看明白代码中某个语句是做什么用的，为什么要这么设置。接下来让我们直观地看一下代码，当然，在今天的这篇文章中，我们不去打开程序文档去，而只是看看这些文件夹，通过这些文件夹来划分一下功能模块，粗略了解各部分代码对应了上面所述逻辑框架的哪个部分。后面再详细阅读源码：

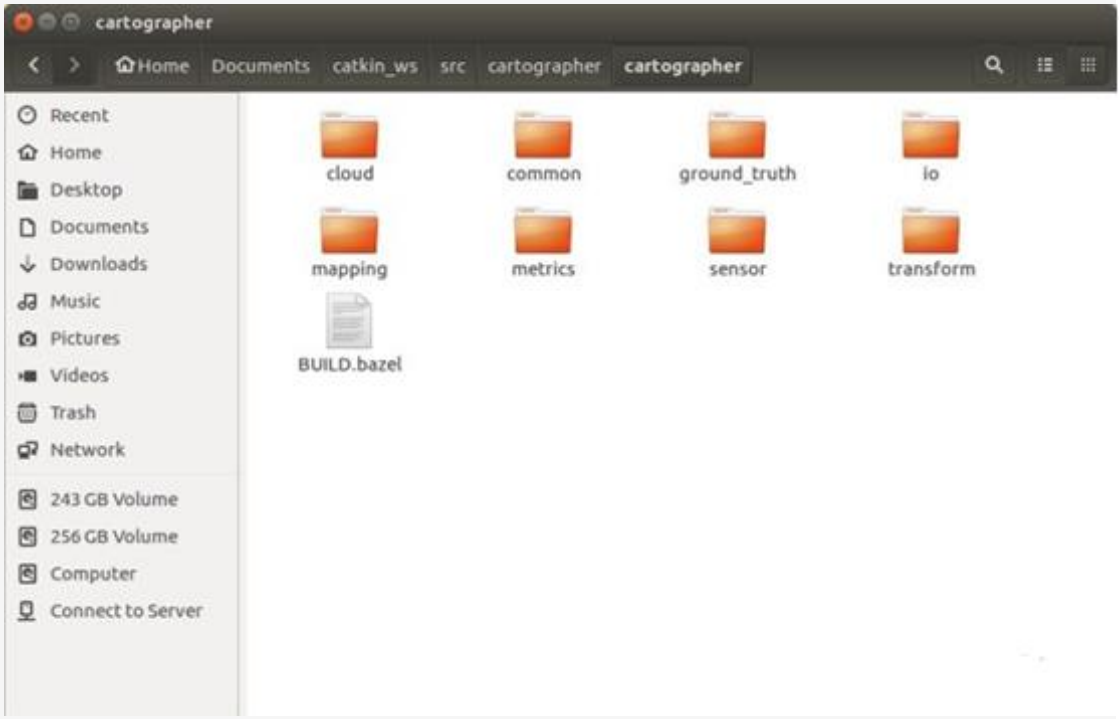
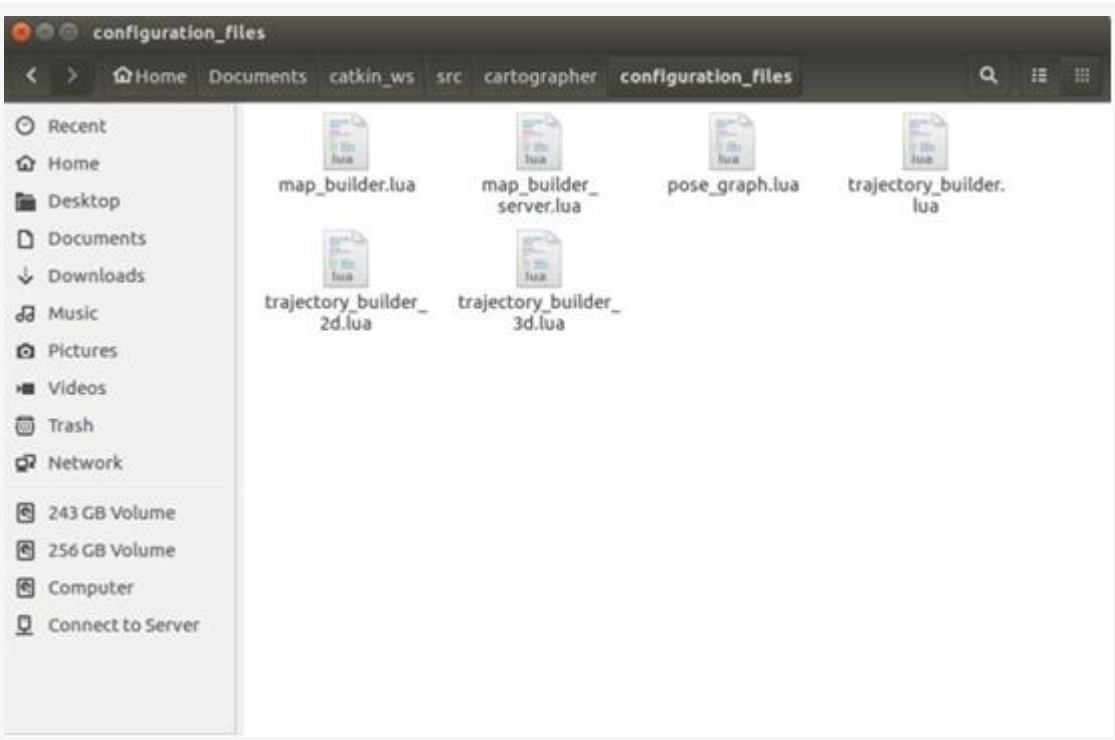


图 3：cartographer 下的文件夹

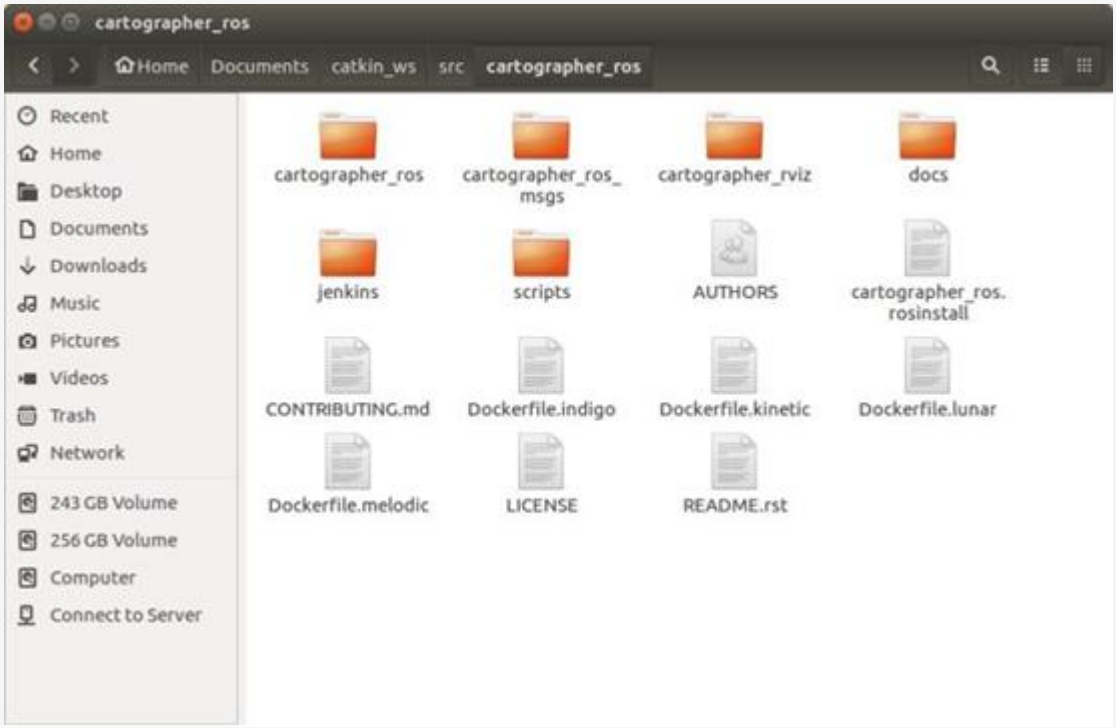
cartographer 部分源码主要分为了如图所示的几个文件夹：cloud、common、ground_truth、io、mapping、metrics、sensor、transform。其中一些文件夹的作用我们也能猜到：

1. common：定义了基本数据结构和一些工具的使用接口。例如，四舍五入取整的函数、时间转化相关的一些函数、数值计算的函数、互斥锁工具等。详见 Common 部分源码分析
2. sensor：定义了传感器数据的相关数据结构。详见 sensor 源码分析
3. transform：定义了位姿的数据结构及其相关的转化。如 2d\3d 的刚体变换等。详见：transform 部分源码分析
4. mapping：定义了上层应用的调用借口以及局部 submap 构建和基于闭环检测的位姿优化等的接口。这个文件也是算法的核心。其中 mapping_2d 和 mapping_3d 是对 mapping 接口的不同实现。详见 mapping 部分源码分析
5. io：定义了一些与 IO 相关的工具，用于存读取数据、记录日志等。详见 io 部分源码分析
6. cloud：该部分的功能我暂时还不清楚，打开看猜测是与云服务相关的功能。ground_truth、metrics 这两个文件夹也还不清楚作用，目前该部分可以暂时不管，等我们后面碰到了再说。

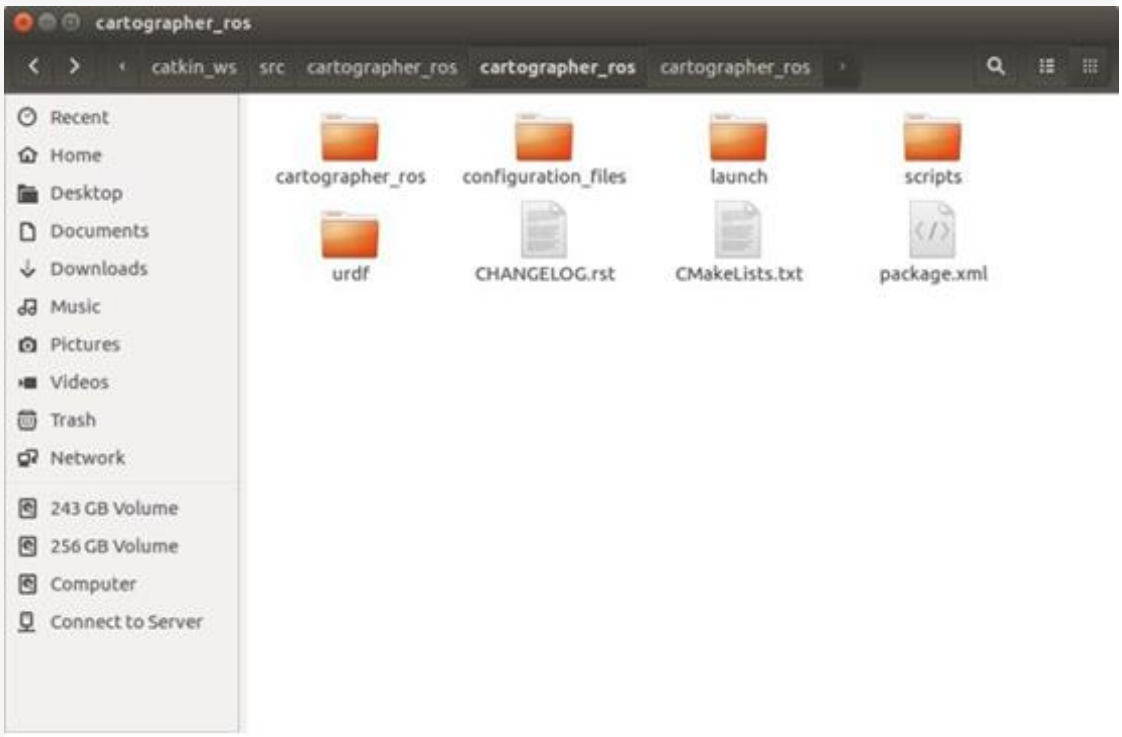


另外，/src/cartographer/configuration_files 文件夹下是一些配置文件。算法一些参数在这里定义，我们也要改这里的参数进行 tunning.

\src\cartographer_ros 下的文件如图所示：



cartographer_ros_msgs 里定义了部分 msg 文件和 srv 文件。cartographer_rviz 估计是与显示相关的函数。让我们把关注点放到/src/cartographer_ros/cartographer_ros 下：



如图所示，configuration_files 是一些配置文件，launch 里是一些 launch 文件。这些暂时不管。继续进入/src/cartographer_ros/cartographer_ros/cartographer_ros。这里我们可以找到程序入口 http://node_main.cc。

本文由于篇幅原因，暂时到这里，下篇文章我们从 http://node_main.cc 入手。

第 2 章 从 cartographer_node 入手

上篇我们梳理了 Cartographer 算法的主要逻辑框架。cartographer_ros 是一个 Package，包含多个 node，每一个 node 也都有自己的入口 main 函数。但 cartographer_node 是其中最重要的一个 node。所以，我们从这个 node 的入口函数 http://node_main.cc 开始。该文件路径为：/src/cartographer_ros/cartographer_ros/cartographer_ros 下。

2.1、node_main.cc

```
int main(int argc, char** argv) {
  google::InitGoogleLogging(argv[0]);
  google::ParseCommandLineFlags(&argc, &argv, true);

  CHECK(!FLAGS_configuration_directory.empty())
      << "-configuration_directory is missing.";
  CHECK(!FLAGS_configuration_basename.empty())
      << "-configuration_basename is missing.";

  ::ros::init(argc, argv, "cartographer_node");//初始化 ROS，定义一个节点 cartographer_node
  ::ros::start();//启动 ROS

  cartographer_ros::ScopedRosLogSink ros_log_sink;
  cartographer_ros::Run();
  ::ros::shutdown();
}
```

可以看到，程序开始后，进行了一些初始化设置，然后向 ROS master 注册了一个名为“cartographer_node”的节点并启动。让我们跳转到该节点的 Run 函数中。

2.2、cartographer_node 的 Run 函数

```
void Run() {
  constexpr double kTfBufferCacheTimeInSeconds = 10.;
  tf2_ros::Buffer tf_buffer{::ros::Duration(kTfBufferCacheTimeInSeconds)};//tf2_ros::Buffer 是 tf2 library 的主要工具
  tf2_ros::TransformListener tf(tf_buffer);//subscribes to a appropriate topics to receive the transformation.
  NodeOptions node_options;//NodeOptions 在 /cartographer_ros/cartographer_ros/cartographer/node_options.h 中定义；该 struct 中包含了对一些基本参数的设置，比如接收 tf 的 timeout 时间设置、子图发布周期设置等
  TrajectoryOptions trajectory_options;//TrajectoryOptions 在 /cartographer_ros/cartographer_ros/cartographer/trajectory_options.h 中定义。
```

```
  std::tie(node_options, trajectory_options) =
      LoadOptions(FLAGS_configuration_directory, FLAGS_configuration_basename);//将
LoadOptions 获取到的参数值分别赋给 node_options 和 trajectory_options. LoadOptions 函数在
node_options.h 中定义。

  //auto 关键字：auto 可以在声明变量的时候根据变量初始值的类型自动为此变量选择匹配的类型，类
似的关键字还有 decltype。举个例子：int a = 10;auto au_a = a;//自动类型推断，au_a 为 int 类型；
cout << typeid(au_a).name() << endl;
  auto map_builder =
      cartographer::common::make_unique<cartographer::mapping::MapBuilder>(
          node_options.map_builder_options);//cartographer::common::make_unique 定义在 common
文件夹下的 make_unique.h 文件中。
  Node node(node_options, std::move(map_builder), &tf_buffer);//Node 在
/cartographer_ros/cartographer_ros/cartographer/node.h 中定义；在该构造函数中订阅了很多传感
器的 topic。收集传感器数据
  if (!FLAGS_load_state_filename.empty()) {
    node.LoadState(FLAGS_load_state_filename, FLAGS_load_frozen_state);//加载数据包数据
  }

  if (FLAGS_start_trajectory_with_default_topics) {
    node.StartTrajectoryWithDefaultTopics(trajectory_options);
  }


  //ros::spin() 将会进入循环， 一直调用回调函数 chatterCallback(),

  ::ros::spin();//ROS 消息回调处理函数;

  node.FinishAllTrajectories();
  node.RunFinalOptimization();

  if (!FLAGS_save_state_filename.empty()) {
    node.SerializeState(FLAGS_save_state_filename);
  }
}
```

代码中注释是我加的，如有不正确地方请斧正。

该段代码前面部分都是在定义一些变量类型，这些变量类型的定义我在注释中也进行了说明，读者可自行查找。然后通过下面这句代码调用 LoadOptions 函数读取目录下的配置文件，将一些参数赋给 node_options, trajectory_options：

```
std::tie(node_options, trajectory_options) =
      LoadOptions(FLAGS_configuration_directory, FLAGS_configuration_basename);//将
LoadOptions 获取到的参数值分别赋给 node_options 和 trajectory_options. LoadOptions 函数在
node_options.h 中定义。
```

后面一句利用在/src/cartographer/cartographer/common/make_unique.h 实现的 make_unique 功能，定义了一个 cartographer::mapping::MapBuilder 类型的变量 map_builder，其中，cartographer::mapping::MapBuilder 这个类/src/cartographer/cartographer/mapping/map_builder.h 中定义。

```
auto map_builder = cartographer::common::make_unique<cartographer::mapping::MapBuilder>
(node_options.map_builder_options);
```

Note:
/src/cartographer/cartographer/common/make_unique.h 是对 std::make_unique 功能在不支持 c++14 的环境下的实现。由于 std:make_unique 只可在支持 c++14 的环境中使用，所以作者在 common 中重新实现了一下 std:make_unique.

知识补充：
std:make_unique<unique_ptr>作用是创建并返回 unique_ptr 至指定类型的对象，该对象是使用指定的参数构造的。比如本句中使用的参数就 node_options.map_builder_options，指定的返回类型就是 cartographer::mapping::MapBuilder.
make_unique 同 unique_ptr、auto_ptr 等一样，都是 smart pointer，可以取代 new 并且无需 delete pointer,有助于代码管理。

画外音：
看吧！本文作者还确实是挺啰嗦的吧。连这些 c++的特性都要交代。谁让作者本身也是个 c++小白呢。一边看一边学，学到什么就记什么。但是这种做法对同样是小白的人来说就很友好了啊！

接下来就是比较重要的一句了：

```
Node node(node_options, std::move(map_builder), &tf_buffer);
```

别看在代码中就是这么短短的一句，表面上就是根据三个参数构造了一个 Node 型的变量 node。但当我们打开 Node 这个类的构造函数时，才会发现 Node 的构造函数函数里做了很多事情，让我们一起去打开 Node 这个类。该类在/src/cartographer_ros/cartographer_ros/cartographer/node.h 中定义，由.../http://node.cc 实现：

2.3、Node 的构造函数：

```
Node::Node(
    const NodeOptions& node_options,
    std::unique_ptr<cartographer::mapping::MapBuilderInterface> map_builder,
    tf2_ros::Buffer* const tf_buffer)//函数后面的冒号表示一个实例的赋值
    : node_options_(node_options),
      map_builder_bridge_(node_options_, std::move(map_builder), tf_buffer) {//构造函数的主体
  carto::common::MutexLocker lock(&mutex_);//设置一个互斥锁；
  submap_list_publisher_ =
```

```
      node_handle_.advertise<::cartographer_ros_msgs::SubmapList>(
          kSubmapListTopic, kLatestOnlyPublisherQueueSize);//告知 master 节点，我们将要向
kSubmapListTopic 这个 Topic 上发布一个::cartographer_ros_msgs::SubmapList 型的 message，而第二
个参数是 publishing 的缓存大小；发布的该 Topic 即可允许其他节点获取到我们构建的 Submap 的信息
  trajectory_node_list_publisher_ =
      node_handle_.advertise<::visualization_msgs::MarkerArray>(
          kTrajectoryNodeListTopic, kLatestOnlyPublisherQueueSize);//同样的，向
kTrajectoryNodeListTopic 这个 Topic 上发布了一个::visualization_msgs::MarkerArray 型的 message
  landmark_poses_list_publisher_ =
      node_handle_.advertise<::visualization_msgs::MarkerArray>(
          kLandmarkPosesListTopic, kLatestOnlyPublisherQueueSize);//同样的，向
kLandmarkPosesListTopic 这个 Topic 上发布了一个::visualization_msgs::MarkerArray 型的 message
  constraint_list_publisher_ =
      node_handle_.advertise<::visualization_msgs::MarkerArray>(
          kConstraintListTopic, kLatestOnlyPublisherQueueSize);//同样的，向
kConstraintListTopic 这个 Topic 上发布了一个::visualization_msgs::MarkerArray 型的 message

  service_servers_.push_back(node_handle_.advertiseService(
      kSubmapQueryServiceName, &Node::HandleSubmapQuery, this));//注册一个 Service,Service 的
名字由 kSubmapQueryServiceName 给出。第二个参数是该 Service 绑定的函数句柄,即当有一个 service
的 request 时，由该函数进行 response. 注册的这个 service 就对应了 submap_query 这个 service。这
是 cartographer_node 可以提供的一个 service.
  service_servers_.push_back(node_handle_.advertiseService(
      kStartTrajectoryServiceName, &Node::HandleStartTrajectory, this));//同样的，注册
Service 名字为 kStartTrajectoryServiceName，函数句柄为 Node::HandleStartTrajectory
  service_servers_.push_back(node_handle_.advertiseService(
      kFinishTrajectoryServiceName, &Node::HandleFinishTrajectory, this));//同样的，注册
Service 名字为 kFinishTrajectoryServiceName，函数句柄为 Node::HandleFinishTrajectory
  service_servers_.push_back(node_handle_.advertiseService(
      kWriteStateServiceName, &Node::HandleWriteState, this));////同样的，注册 Service 名字为
kWriteStateServiceName，函数句柄为 Node::HandleWriteState

  scan_matched_point_cloud_publisher_ =
      node_handle_.advertise<sensor_msgs::PointCloud2>(
          kScanMatchedPointCloudTopic, kLatestOnlyPublisherQueueSize);//又发布了一个跟点云相
关的 Topic

  //wall_timers 在 node.h 中定义，是一个存储::ros::WallTimer 类型的 vector，以下通过 vector 的
push_back 操作依次将五个::ros::WallTimer 型对象插入这个 vector 的末尾。::ros::WallTimer 这个类
参见如下链接：http://docs.ros.org/jade/api/roscpp/html/classros_1_1WallTimer.html . 简单说，
这是一个定时器，这里分别为如下的五个函数设置了定时器。参数就是 node_options 里的各项参数；
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.submap_publish_period_sec),
      &Node::PublishSubmapList, this));
```

```
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.pose_publish_period_sec),
      &Node::PublishTrajectoryStates, this));
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.trajectory_publish_period_sec),
      &Node::PublishTrajectoryNodeList, this));
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.trajectory_publish_period_sec),
      &Node::PublishLandmarkPosesList, this));
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(kConstraintPublishPeriodSec),
      &Node::PublishConstraintList, this));
}
```

可以看到，这里对我们比较有用的信息就是这个 node 发布了如下 Topic：

|-------kSubmapListTopic: 将构建的 submap 的 list 发布出来

|-------kTrajectoryNodeListTopic：trajectory 的 list 发布出来

|-------kLandmarkPosesListTopic： Landmark 的位姿 list

|-------kConstraintListTopic: constraintlist 发布

|-------kScanMatchedPointCloudTopic：匹配上的点云数据

同时，该节点可提供如下服务：

|-------kSubmapQueryServiceName: 查询 Submap

|-------kStartTrajectoryServiceName： 开始一段 trajectory

|-------kFinishTrajectoryServiceName: 结束一段 trajectory

|-------kWriteStateServiceName: 写状态

我们再回顾一下上一篇文章中提到的 cartographer_node 这个节点发布的 Topic 和提供的服务列表，如下图所示：

```
1. cartographer_node
   |===@订阅的Topic
   |---------scan (sensors_msgs/LaserScan):   解释: 传感器数据
   |---------echoes (sensors_msgs/MultiEchoLaserScan) 解释: 没太看明白这个是做什么用，感
   |---------points2 (sensors_msgs/PointCloud2) 解释: 2D点云数据
   |---------imu (sensors_msgs/Imu) 解释: IMU的数据
   |---------odom (nav_msgs/Odometry) 解释: 里程计数据
   |===@发布的Topic
   |---------scan_matched_points2 (sensors_msgs/PointCloud2) 解释: 匹配好的2D点云数据，
   |---------submap_list (cartographer_ros_msgs/SubmapList) 解释: 发布构建好的submap。
   |===@提供的Service
   |---------submap_query (cartographer_ros_msgs/SubmapQuery) 解释: 可以提供查询submap的
   |---------start_trajectory (cartographer_ros_msgs/StartTrajectory) 解释: 维护一条轨
   |---------finish_trajectory (cartographer_ros_msgs/FinishTrajectory) 解释: Finish一
   |---------write_state (cartographer_ros_msgs/WriteState) 解释: 将当前状态写入磁盘文件
   |---------get_trajectory_states (cartographer_ros_msgs/GetTrajectoryStates) 解释:
   |---------read_metrics (cartographer_ros_msgs/ReadMetrics)
   |===@Required tf Transforms
   |===@Provided tf Transforms
```

我们对比一下可以发现，代码中发布的 Topic 除了文档中说的 scan_matched_points2 和 submap_list 外，又多了 kTrajectoryNodeListTopic、kLandmarkPosesListTopic 和 kConstraintListTopic. 发布的服务里有 submap_query、start_trajectory、finish_trajectory，write_state，但还没找到 get_trajectory_states、read_metrics 这两个服务。节点订阅的 Topic 我们也还没看到。个人猜测这种不一致是文档更新落后与代码更新的缘故。可以看到，在新的代码里 cartographer 已经考虑了加入 Landmark，这个部分也是我当前要做的部分所需要的。

刚才我们看到在发布 Service 时，我们为每个 Service 绑定了一个函数句柄。从名字我们也能才能每个函数是用来做什么的。所以接下来我们粗略地看一下这五个函数都做了哪些操作：

1. 首先是 Node::HandleSubmapQuery:

```
bool Node::HandleSubmapQuery(

    ::cartographer_ros_msgs::SubmapQuery::Request& request,

    ::cartographer_ros_msgs::SubmapQuery::Response& response) {

  carto::common::MutexLocker lock(&mutex_);//设置互斥锁

  map_builder_bridge_.HandleSubmapQuery(request, response);//map_builder_bridge_在 node.h 的第 171 行定义。是一个 MapBuilderBridge 型的变量。

  return true;
```

}

可以看到，这个函数有两个参数，一个参数是一个::cartographer_ros_msgs::SubmapQuery::Request 型的变量，对应是请求服务时的输入参数，另外一个参数是::cartographer_ros_msgs::SubmapQuery::Response 型的变量，对应的是服务响应后的返回值。不明白 Service 中的 request 和 response 概念的童鞋可以参见 ROS 学习笔记：Beginner 阶段_图文_百度文库　。

可以看到这个函数调用了 map_builder_bridge_的 HandlesSubmapQuery 来做处理。map_builder_bridge_在 node.h 的第 171 行定义。是一个 MapBuilderBridge 型的变量。MapBuilderBridge 这个类在/src/cartographer_ros/cartographer_ros/cartographer/map_builder_bridge.h 中定义。查看该类的构造函数：

```
MapBuilderBridge::MapBuilderBridge(

    const NodeOptions& node_options,

    std::unique_ptr<cartographer::mapping::MapBuilderInterface> map_builder,

    tf2_ros::Buffer* const tf_buffer)

    : node_options_(node_options),

      map_builder_(std::move(map_builder)),

      tf_buffer_(tf_buffer) {}
```

可以看到函数体是空的，所做的只是实例化赋值了一个 map_builder。map_builder 是接口 MapBuilderInterface(在/src/cartographer/cartographer/map_builder_interface.h 中定义)的一个实例化对象，而这个接口的实现由/src/cartographer/cartographer/map_builder.h 和.../http://map_builder.cc 实现。所以 map_builder_bridge 本质上的功能是由 http://map_builder.cc 决定的，以上都是做的一些封装。

我们回到/src/cartographer_ros/cartographer_ros/cartographer/http://map_builder_bridge.cc 中，找到 HandleSubmapQuery 这个函数：

```
void MapBuilderBridge::HandleSubmapQuery(

    cartographer_ros_msgs::SubmapQuery::Request& request,

    cartographer_ros_msgs::SubmapQuery::Response& response) {
```

```
cartographer::mapping::proto::SubmapQuery::Response response_proto;
cartographer::mapping::SubmapId submap_id{request.trajectory_id,
                                          request.submap_index};
const std::string error =
    map_builder_->SubmapToProto(submap_id, &response_proto);
if (!error.empty()) {
  LOG(ERROR) << error;
  response.status.code = cartographer_ros_msgs::StatusCode::NOT_FOUND;
  response.status.message = error;
  return;
}
response.submap_version = response_proto.submap_version();
for (const auto& texture_proto : response_proto.textures()) {
  response.textures.emplace_back();
  auto& texture = response.textures.back();
  texture.cells.insert(texture.cells.begin(), texture_proto.cells().begin(),
                       texture_proto.cells().end());
  texture.width = texture_proto.width();
  texture.height = texture_proto.height();
  texture.resolution = texture_proto.resolution();
  texture.slice_pose = ToGeometryMsgPose(
      cartographer::transform::ToRigid3(texture_proto.slice_pose()));
```

```
  }

  response.status.message = "Success.";

  response.status.code = cartographer_ros_msgs::StatusCode::OK;

}
```

可以看到，这里基本上就是利用/src/cartographer/cartographer/mapping 下的工具来查询指定 id 的 submap 的一些信息。proto 是 Google 提供的一个 ProtoBuf 库的工具 Google Protobuf 库，用来实现数据的序列化和反序列化。

但我们可以先不管这些，后面再详细讲解。整体上而言，当请求一个查询 submap 的 service 时，程序是在这里处理，能够返回一个 submap 的相关信息。最终的处理还是要靠 cartographer 这个包里面的东西处理，cartographer_ros 只是做了一层 ROS 封装也在这里体现了出来。

对 Node::HandleSubmapQuery 函数的探究暂时告一段落，让我们把思路跳回来，关注一下其他几个 Service 绑定的函数句柄。这次我们不再像对 Node::HandleSubmapQuery 函数那样深入，而只是看一个大概，感兴趣的同学可以深入探讨：

2. Node::HandleStartTrajectory

```
bool Node::HandleStartTrajectory(

    ::cartographer_ros_msgs::StartTrajectory::Request& request,

    ::cartographer_ros_msgs::StartTrajectory::Response& response) {

  carto::common::MutexLocker lock(&mutex_);

  TrajectoryOptions options;

  if (!FromRosMessage(request.options, &options) ||

      !ValidateTrajectoryOptions(options)) {

    const std::string error = "Invalid trajectory options.";

    LOG(ERROR) << error;

    response.status.code = cartographer_ros_msgs::StatusCode::INVALID_ARGUMENT;

    response.status.message = error;
```

```
  } else if (!ValidateTopicNames(request.topics, options)) {
    const std::string error = "Invalid topics.";
    LOG(ERROR) << error;
    response.status.code = cartographer_ros_msgs::StatusCode::INVALID_ARGUMENT;
    response.status.message = error;
  } else {
    response.trajectory_id = AddTrajectory(options, request.topics);
    response.status.code = cartographer_ros_msgs::StatusCode::OK;
    response.status.message = "Success.";
  }
  return true;
}
```

前面一些异常情况的处理，正常情况下调用 AddTrajectory 函数，增加一条 trajectory.

错误纠正：之前我对 trajectory 的理解有偏差，误以为 trajectory 是程序根据实际建图的情况自动生成的，一条 trajectory 与一个 submap 相对应。后来在跟我的 Leader 请教后才发现自己理解错了。实际上，一条 trajectory 可以理解为一次建图过程。如果只让机器人跑一圈，那么 trajectory 数就是 1. 但比如说，建好图后又需要在图中走，这时候可以增加一条 trajectory，把这条 trajectory 设置成 /src/cartographer/configuration_files/trajectory_builder.lua 中的 pure_localization 设为 true，那么机器人再重新跑的过程中就会跟已经建好的图进行匹配，估计机器人在地图中的路径。所以，一次运行就代表了一条 trajectory。

AddTrajecory 代码如下：

```
int Node::AddTrajectory(const TrajectoryOptions& options,
                        const cartographer_ros_msgs::SensorTopics& topics) {
  const std::set<cartographer::mapping::TrajectoryBuilderInterface::SensorId>
      expected_sensor_ids = ComputeExpectedSensorIds(options, topics);
```

```
  const int trajectory_id =
      map_builder_bridge_.AddTrajectory(expected_sensor_ids, options);
  AddExtrapolator(trajectory_id, options);
  AddSensorSamplers(trajectory_id, options);
  LaunchSubscribers(options, topics, trajectory_id);
  is_active_trajectory_[trajectory_id] = true;
  for (const auto& sensor_id : expected_sensor_ids) {
    subscribed_topics_.insert(sensor_id.id);
  }
  return trajectory_id;
}
```

同样，AddTrajectory 函数也是通过调用 map_builder_bridge_中的 AddTrajectory 来处理。同时，每增加一条轨迹，都需要给该轨迹增加必要的处理，比如添加位姿估计的 AddExtrapolator，设置传感器的 AddSensorSamplers，用来订阅必要的 Topic 以接收数据的 LaunchSubscribers 等。每个函数具体功能请自行参见代码，这里不再赘述。我们重点关注一下 LaunchSubscribers 这个函数：

```
void Node::LaunchSubscribers(const TrajectoryOptions& options,
                             const cartographer_ros_msgs::SensorTopics& topics,
                             const int trajectory_id) {
  for (const std::string& topic : ComputeRepeatedTopicNames(
           topics.laser_scan_topic, options.num_laser_scans)) {
    subscribers_[trajectory_id].push_back(
        {SubscribeWithHandler<sensor_msgs::LaserScan>(
            &Node::HandleLaserScanMessage, trajectory_id, topic, &node_handle_,
```

```
            this),
        topic});
}//
for (const std::string& topic :
     ComputeRepeatedTopicNames(topics.multi_echo_laser_scan_topic,
                               options.num_multi_echo_laser_scans)) {
  subscribers_[trajectory_id].push_back(
      {SubscribeWithHandler<sensor_msgs::MultiEchoLaserScan>(
          &Node::HandleMultiEchoLaserScanMessage, trajectory_id, topic,
          &node_handle_, this),
       topic});
}
for (const std::string& topic : ComputeRepeatedTopicNames(
         topics.point_cloud2_topic, options.num_point_clouds)) {
  subscribers_[trajectory_id].push_back(
      {SubscribeWithHandler<sensor_msgs::PointCloud2>(
          &Node::HandlePointCloud2Message, trajectory_id, topic,
          &node_handle_, this),
       topic});
}
// For 2D SLAM, subscribe to the IMU if we expect it. For 3D SLAM, the IMU is required.
if (node_options_.map_builder_options.use_trajectory_builder_3d() ||
```

```
    (node_options_.map_builder_options.use_trajectory_builder_2d() &&
     options.trajectory_builder_options.trajectory_builder_2d_options()
         .use_imu_data())) {
  std::string topic = topics.imu_topic;
  subscribers_[trajectory_id].push_back(
      {SubscribeWithHandler<sensor_msgs::Imu>(&Node::HandleImuMessage,
                                              trajectory_id, topic,
                                              &node_handle_, this),
       topic});
}
if (options.use_odometry) {
  std::string topic = topics.odometry_topic;
  subscribers_[trajectory_id].push_back(
      {SubscribeWithHandler<nav_msgs::Odometry>(&Node::HandleOdometryMessage,
                                                trajectory_id, topic,
                                                &node_handle_, this),
       topic});
}
if (options.use_nav_sat) {
  std::string topic = topics.nav_sat_fix_topic;
  subscribers_[trajectory_id].push_back(
      {SubscribeWithHandler<sensor_msgs::NavSatFix>(
```

```
            &Node::HandleNavSatFixMessage, trajectory_id, topic, &node_handle_,
            this),
        topic});
  }
  if (options.use_landmarks) {
    std::string topic = topics.landmark_topic;
    subscribers_[trajectory_id].push_back(
        {SubscribeWithHandler<cartographer_ros_msgs::LandmarkList>(
            &Node::HandleLandmarkMessage, trajectory_id, topic, &node_handle_,
            this),
        topic});
  }
}
```

这里代码虽长，但结构差不多，都是在订阅传感器发布的消息。可以看到有 Laser, MultiEchoLaser, PointCloud2, IMU, Odometry, NavSatFixMessage, Landmark 等，其中后三项是用 if 来判断该传感器使用情况。所以不同的应用场景需要不同地修改配置文件。这几项与我们之前整理 cartographer_node 这个节点订阅的 Topic 是一致的。

以 Laser 为例：

```
 for (const std::string& topic : ComputeRepeatedTopicNames(
          topics.laser_scan_topic, options.num_laser_scans)) {
    subscribers_[trajectory_id].push_back(
        {SubscribeWithHandler<sensor_msgs::LaserScan>(
            &Node::HandleLaserScanMessage, trajectory_id, topic, &node_handle_,
```

```
            this),
        topic});
  }//
```

该代码主要作用是这个函数：

```
SubscribeWithHandler<sensor_msgs::LaserScan>(
            &Node::HandleLaserScanMessage, trajectory_id, topic, &node_handle_,
            this)
```

该函数如下定义：

```
// Subscribes to the 'topic' for 'trajectory_id' using the 'node_handle' and
// calls 'handler' on the 'node' to handle messages. Returns the subscriber.
template <typename MessageType>
::ros::Subscriber SubscribeWithHandler(
    void (Node::*handler)(int, const std::string&,
                          const typename MessageType::ConstPtr&),
    const int trajectory_id, const std::string& topic,
    ::ros::NodeHandle* const node_handle, Node* const node) {
  return node_handle->subscribe<MessageType>(
      topic, kInfiniteSubscriberQueueSize,
      boost::function<void(const typename MessageType::ConstPtr&)>(
          [node, handler, trajectory_id,
           topic](const typename MessageType::ConstPtr& msg) {
            (node->*handler)(trajectory_id, topic, msg);
```

```
        }));
```

主要作用就是订阅一个以 topic(不同的传感器中的 topic 这个变量是否 for 循环体中的这一句代码赋值的 const std::string& topic : ComputeRepeatedTopicNames(topics.laser_scan_topic, options.num_laser_scans))为名字的 Topic，然后返回了一个 node_handle->subscribe<MessageType>。在 LaunchSubscribers 函数里把这个返回值压入了 subscribers_[trajectory_id]列表中。

订阅之后的处理是在 Node::HandleLaserScanMessage，查看该代码就可以发现最后依然交给了 map_builder_bridge_去处理。

其他与此类似。

第 3 章 从 cartographer_node 入手 2

上一篇文章里我们已经阅读了两个 Service 的句柄函数，接下来让我们粗略地看一下 Node::HandleFinishTrajectory 和 Node::HandleWriteState.

3.1、Node::HandleFinishTrajectory

```
bool Node::HandleFinishTrajectory(
    ::cartographer_ros_msgs::FinishTrajectory::Request& request,
    ::cartographer_ros_msgs::FinishTrajectory::Response& response) {
  carto::common::MutexLocker lock(&mutex_);
  response.status = FinishTrajectoryUnderLock(request.trajectory_id);
  return true;
}
```

调用了 FinishTrajectoryUnderLock 函数。查看该函数：

```
cartographer_ros_msgs::StatusResponse Node::FinishTrajectoryUnderLock(
    const int trajectory_id) {
  cartographer_ros_msgs::StatusResponse status_response;
  // First, check if we can actually finish the trajectory.
  if (map_builder_bridge_.GetFrozenTrajectoryIds().count(trajectory_id)) {
    const std::string error =
        "Trajectory " + std::to_string(trajectory_id) + " is frozen.";
    LOG(ERROR) << error;
    status_response.code = cartographer_ros_msgs::StatusCode::INVALID_ARGUMENT;
```

```
  status_response.message = error;
  return status_response;
}
if (is_active_trajectory_.count(trajectory_id) == 0) {
  const std::string error =
      "Trajectory " + std::to_string(trajectory_id) + " is not created yet.";
  LOG(ERROR) << error;
  status_response.code = cartographer_ros_msgs::StatusCode::NOT_FOUND;
  status_response.message = error;
  return status_response;
}
if (!is_active_trajectory_[trajectory_id]) {
  const std::string error = "Trajectory " + std::to_string(trajectory_id) +
                            " has already been finished.";
  LOG(ERROR) << error;
  status_response.code =
      cartographer_ros_msgs::StatusCode::RESOURCE_EXHAUSTED;
  status_response.message = error;
  return status_response;
}
// Shutdown the subscribers of this trajectory.
for (auto& entry : subscribers_[trajectory_id]) {
```

```
    entry.subscriber.shutdown();
    subscribed_topics_.erase(entry.topic);
    LOG(INFO) << "Shutdown the subscriber of [" << entry.topic << "]";
  }
  CHECK_EQ(subscribers_.erase(trajectory_id), 1);
  CHECK(is_active_trajectory_.at(trajectory_id));
  map_builder_bridge_.FinishTrajectory(trajectory_id);
  is_active_trajectory_[trajectory_id] = false;
  const std::string message =
      "Finished trajectory " + std::to_string(trajectory_id) + ".";
  status_response.code = cartographer_ros_msgs::StatusCode::OK;
  status_response.message = message;
  return status_response;
}
```

返回值类型是 cartographer_ros_msgs::StatusResponse，在 /src/cartographer_ros/cartographer_ros_msgs/msg/StatusCode.msg 中定义。

前面检查了一下是否可以关掉，指定 id 是否存在，是否已经被 Finished 了等情况后，如果一切正常，则停止订阅 Topic、清除 id 及其他与该 trajectory 相关的量。最后调用 map_builder_bridge_中的 FinishTrajectory 函数。

我们之前也提到过，cartographer_ros 相当于只给我们封装了一个 ROS 的皮，核心的任务都还是由 cartographer 来完成的。所以我们当前查看该节点的情况，追述到 map_builder_bridge_后就都暂时停止，知道这些任务最后交给了 cartographer 这个包中的 MapBuilder 去处理就可以了。等我们研究完这个节点后再集中精力去看 cartographer 中的内容。

3.2、Node::HandleWriteState

```
bool Node::HandleWriteState(
    ::cartographer_ros_msgs::WriteState::Request& request,
    ::cartographer_ros_msgs::WriteState::Response& response) {
  carto::common::MutexLocker lock(&mutex_);
  if (map_builder_bridge_.SerializeState(request.filename)) {
    response.status.code = cartographer_ros_msgs::StatusCode::OK;
    response.status.message = "State written to '" + request.filename + "'.";
  } else {
    response.status.code = cartographer_ros_msgs::StatusCode::INVALID_ARGUMENT;
    response.status.message = "Failed to write '" + request.filename + "'.";
  }
  return true;
}
```

这个函数没什么好说的，也是调用了 map_builder_bridge_.SerializeState(request.filename)这个函数。猜测应该是将一些状态数据等写入文件系统的作用。

好了，几个 service 的句柄函数看完了，我们可以跳回到 Node 的构造函数。可以看到接下来是为几个 Topic 设置了定时器，以及定时器函数。猜测这几个定时器函数里就是定时往 Topic 上发布消息。

//wall_timers 在 node.h 中定义，是一个存储::ros::WallTimer 类型的 vector，以下通过 vector 的 push_back 操作依次将五个::ros::WallTimer 型对象插入这个 vector 的末尾。::ros::WallTimer 这个类参见如下链接：http://docs.ros.org/jade/api/roscpp/html/classros_1_1WallTimer.html ． 简单说，这是一个定时器，这里分别为如下的五个函数设置了定时器。参数就是 node_options 里的各项参数；

```
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.submap_publish_period_sec),
      &Node::PublishSubmapList, this));
```

```
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.pose_publish_period_sec),
      &Node::PublishTrajectoryStates, this));
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.trajectory_publish_period_sec),
      &Node::PublishTrajectoryNodeList, this));
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(node_options_.trajectory_publish_period_sec),
      &Node::PublishLandmarkPosesList, this));
  wall_timers_.push_back(node_handle_.createWallTimer(
      ::ros::WallDuration(kConstraintPublishPeriodSec),
      &Node::PublishConstraintList, this));
```

我们以第一个 Topic 的定时器函数 Node::PublishSubmapList 为例:

```
void Node::PublishSubmapList(const ::ros::WallTimerEvent& unused_timer_event) {
  carto::common::MutexLocker lock(&mutex_);
  submap_list_publisher_.publish(map_builder_bridge_.GetSubmapList());
}
```

可以看到，果然就是调用 map_builder_bridge_.GetSubmapList()函数获取到 submap 的 list 然后用 ros 的 publish 函数向 Topic 上广播这个消息。

其他几个与此类似。我们不再赘述。

至此，对于 cartographer_node 的源码阅读基本结束。之后我们需要去读 cartographer 部分看具体做了什么操作。这里我们先对 cartographer_node 做一个总结。

cartographer_node 总结：

cartographer 启动以后，做了如下的工作：

1. 注册并发布了 5 个 Topic，并为 5 个 Topic 分别设置了定时器函数，在定时器函数中定期向 Topic 上广播数据：

|===1) Topic 1: kSubmapListTopic: 广播构建出来的 submap 的 list

|----------发布数据的函数：Node::PublishSubmapList

|----------调用函数：map_builder_bridge_.GetSubmapList();

|===2) Topic 2: kTrajectoryNodeListTopic：发布 trajectory

|----------发布数据的函数：Node::PublishTrajectoryNodeList Node::PublishTrajectoryStates：

|----------调用函数：map_builder_bridge_.GetTrajectoryNodeList();

|===3) Topic3: kLandmarkPoseListTopic

|----------发布数据的函数：Node::PublishLandmarkPosesList

|----------调用的函数：map_builder_bridge_.GetLandmarkPoseList();

|===4) Topic4: kConstraintListTopic

|----------发布数据的函数：Node::PublishConstraintList

|----------调用的函数：map_builder_bridge_.GetConstraintList();

|===5）Topic5: kScanMatchedPointCloudTopic

|----------发布数据的函数：Node::PublishTrajectoryStates

|----------调用的函数：map_builder_bridge_.GetTrajectoryStates();

|----------这个函数名与 Topic 名对应不上，但是查看 Node::PublishTrajectoryStates 不难发现，函数中有一句：

```
scan_matched_point_cloud_publisher_.publish(ToPointCloud2Message(
            carto::common::ToUniversal(trajectory_state.local_slam_data->time),
            node_options_.map_frame,
```

```
carto::sensor::TransformTimedPointCloud(

    point_cloud, trajectory_state.local_to_map.cast<float>())));
```

2. 发布了 4 个 Service，并为 4 个 Service 分别设置了句柄函数，而句柄函数也是通过调用 map_builder_bridge_的成员函数来处理的。

|===1）Service 1: kSubmapQueryServiceName

|----------句柄函数：Node::HandleSubmapQuery

|----------实际处理的函数：map_builder_bridge_.HandleSubmapQuery

|===2) Service 2: kStartTrajectoryServiceName

|----------句柄函数：Node::HandleStartTrajectory

|----------实际处理的函数：map_builder_bridge_.AddTrajectory 等;

|----------这里需要额外注意的是 Node::LaunchSubscribers 这个函数。这个函数负责处理各个传感器函数。仔细读其中的每个处理函数，比如处理 IMU 的 Node::HandleImuMessage 函数，发现其实际调用的是 map_builder_bridge_中的一个成员类 sensor_bridge_ptr 的函数来处理：sensor_bridge_ptr->HandleImuMessage。

|===3) Service 3: kFinishTrajectoryServiceName

|----------句柄函数：Node::HandleFinishTrajectory

|----------实际处理的函数：map_builder_bridge_.FinishTrajectory

|===4) Service 4: kWriteStateServiceName

|----------句柄函数：Node::HandleWriteState

|----------实际处理的函数：map_builder_bridge_.SerializeState

可以看到，cartographer_ros 这个节点启动后所有东西都交给了 map__builder_bridge_去处理。所以，之后我们得去研究这个类：MapBuilderBridge 了。 这个点我们就留到下篇文章再开始。

第 4 章 MapBuilderBridge

在上一篇文章中，我们已经看到，cartographer_node 这个节点的所有操作，都交给了 MapBuilderBridge 的一个对象 map_builder_bridge_来完成。因此，接下来我们要来看看在 MapBuilderBridge 中都进行了哪些操作。不过，提醒各位同学，MapBuilderBridge 这个类里我们依然看不到任何关于算法的影子，它也只是一层封装。不过，不要灰心，我们快要接触到算法本身了。

MapBuilderBridge 定义在 /src/cartographer_ros/cartographer_ros/cartographer_ros/map_builder_bridge.h 中，实现由.../http://map_builder_bridge.cc 完成。

我们先看看这个类包括了哪些成员：

Public:

|===成员变量：

这里不对，应该是 LocalTrajectoryData

|------一个结构体 TrajectoryState;//该结构体存储的是 local SLAM 处理后的结果。由 range_data_in_local 中已经估计出了在时刻 time 时的当前 local_pose

|------|------结构体 LocalSlamData

|------|------|------::cartographer::common::Time time;//时间

|------|------|------::cartographer::transform::Rigid3d local_pose;//优化匹配出来的 local_pose——在 submap 这个局部坐标系中的位姿

|------|------|------::cartographer::sensor::RangeData range_data_in_local;//激光数据

|------|------std::shared_ptr<const LocalSlamData> local_slam_data;//局部 SLAM 的数据

|------|------cartographer::transform::Rigid3d local_to_map;//submap 到 global map 的坐标变换关系

|------|------std::unique_ptr<cartographer::transform::Rigid3d> published_to_tracking;//猜测是要输入 PoseExtrapolator（“详见 Cartographer 源码阅读 1——整体框架介绍”中的第一个图）中与 IMU、里程计等数据融合来估计位姿的。

|------|------TrajectoryOptions trajectory_options;//配置参数

|------SensorBridge* sensor_bridge(int trajectory_id);//SensorBridge 定义在 /src/cartographer_ros/cartographer_ros/cartographer_ros/sensor_bridge.h 中，实现由.../sensor_bridge.cc 完成

|===成员函数：

|------构造函数

|------MapBuilderBridge(

const NodeOptions& node_options,

std::unique_ptr<cartographer::mapping::MapBuilderInterface> map_builder,

tf2_ros::Buffer* tf_buffer);

|------MapBuilderBridge(const MapBuilderBridge&) = delete;

|------MapBuilderBridge& operator=(const MapBuilderBridge&) = delete;//重载了赋值操作

|------void LoadState(const std::string& state_filename, bool load_frozen_state);

|------int AddTrajectory(

const std::set<

::cartographer::mapping::TrajectoryBuilderInterface::SensorId>&

expected_sensor_ids,

const TrajectoryOptions& trajectory_options);

|------void FinishTrajectory(int trajectory_id);

|------void RunFinalOptimization();

|------bool SerializeState(const std::string& filename);

|------void HandleSubmapQuery(

cartographer_ros_msgs::SubmapQuery::Request& request,

cartographer_ros_msgs::SubmapQuery::Response& response);

|------std::set<int> GetFrozenTrajectoryIds();

|------cartographer_ros_msgs::SubmapList GetSubmapList();

|------std::unordered_map<int, TrajectoryState> GetTrajectoryStates()

EXCLUDES(mutex_);

|------visualization_msgs::MarkerArray GetTrajectoryNodeList();

|------visualization_msgs::MarkerArray GetLandmarkPosesList();

|------visualization_msgs::MarkerArray GetConstraintList();

private:

|===成员变量：

|------cartographer::common::Mutex mutex_;

|------const NodeOptions node_options_;

//几个 Unordered map 的 container

|------std::unordered_map<int, std::shared_ptr<const TrajectoryState::LocalSlamData>> trajectory_state_data_ GUARDED_BY(mutex_);

|------std::unique_ptr<cartographer::mapping::MapBuilderInterface> map_builder_;

|------tf2_ros::Buffer* const tf_buffer_;//

|------std::unordered_map<std::string, int> landmark_to_index_;//跟 landmark 相关，其中 std::string 变量表征 landmark 的 ID

|------// These are keyed with 'trajectory_id'.

|------std::unordered_map<int, TrajectoryOptions> trajectory_options_;

|------std::unordered_map<int, std::unique_ptr<SensorBridge>> sensor_bridges_;//元素 为 SensorBridge 成员的一个 unordered map; Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity.

```
|------std::unordered_map<int, size_t> trajectory_to_highest_marker_id_;

|===成员函数：

|------void OnLocalSlamResult(

              const int trajectory_id,

              const ::cartographer::common::Time time,

              const ::cartographer::transform::Rigid3d local_pose,

              ::cartographer::sensor::RangeData range_data_in_local,

              Const std::unique_ptr<const ::cartographer::mapping::

              TrajectoryBuilderInterface::InsertionResult>

              insertion_result) EXCLUDES(mutex_);
```

之前在 cartographer_node 中调用的一些函数都在这里定义了。我们逐块儿查看一下.../http://map_builder_bridge.cc 来查看一下具体代码：

4.1、跟显示相关

我们不关心，也不用改这里的代码。维持原样就好了。这几个函数也不是 MapBuilderBridge 的成员函数，而只是一些工具函数

```
constexpr double kTrajectoryLineStripMarkerScale = 0.07;

constexpr double kLandmarkMarkerScale = 0.3;

constexpr double kConstraintMarkerScale = 0.025;

::std_msgs::ColorRGBA ToMessage(const cartographer::io::FloatColor& color) {

      ::std_msgs::ColorRGBA result;

      result.r = color[0];

      result.g = color[1];
```

```
      result.b = color[2];

      result.a = 1.f;

      return result;

}

visualization_msgs::Marker CreateTrajectoryMarker(const int trajectory_id,

                                                  const std::string& frame_id) {

       visualization_msgs::Marker marker;

       marker.ns = "Trajectory " + std::to_string(trajectory_id);

       marker.id = 0;

       marker.type = visualization_msgs::Marker::LINE_STRIP;

       marker.header.stamp = ::ros::Time::now();

       marker.header.frame_id = frame_id;

       marker.color = ToMessage(cartographer::io::GetColor(trajectory_id));

       marker.scale.x = kTrajectoryLineStripMarkerScale;

       marker.pose.orientation.w = 1.;

       marker.pose.position.z = 0.05;

       return marker;

}

void PushAndResetLineMarker(visualization_msgs::Marker* marker,

                            std::vector<visualization_msgs::Marker>* markers) {

        markers->push_back(*marker);

        ++marker->id;
```

```
        marker->points.clear();

}
```

4.2、与 LandMark 相关的

看一下 cartographer 的配置文件（/src/cartographer_ros/cartographer_ros/configuration_files/backpack_2d.lua or backpack_3d.lua）可以发现，在里面设置 use_landmarks = false，所以，landmar 这部分功能在目前的代码中是没有起作用的。但 cartographer 已经考虑了加入 landmark 的途径。这也是让我非常开心的一件事，这样在后面的工作中，我要更改的代码量应该不会特别大了。调用 map_builder_->pose_graph()->GetLandmarkPoses()可获取 Landmark 的 Pose;

```
//返回一个指定 id 的 landmark 在 container 中的索引值；该函数也并非 MapBuilderBridge 的成员函数

int GetLandmarkIndex(

    const std::string& landmark_id,

    std::unordered_map<std::string, int>* landmark_id_to_index) {

        auto it = landmark_id_to_index->find(landmark_id);

        if (it == landmark_id_to_index->end()) {

            const int new_index = landmark_id_to_index->size();

            landmark_id_to_index->emplace(landmark_id, new_index);

            return new_index;

        }

        return it->second;

}

//创建一个 landmark

visualization_msgs::Marker CreateLandmarkMarker(int landmark_index,

                                                const Rigid3d& landmark_pose,
```

```
                                        const std::string& frame_id) {
  visualization_msgs::Marker marker;
  marker.ns = "Landmarks";
  marker.id = landmark_index;
  marker.type = visualization_msgs::Marker::CUBE;
  marker.header.stamp = ::ros::Time::now();
  marker.header.frame_id = frame_id;
  marker.scale.x = kLandmarkMarkerScale;
  marker.scale.y = kLandmarkMarkerScale;
  marker.scale.z = kLandmarkMarkerScale;
  marker.color = ToMessage(cartographer::io::GetColor(landmark_index));
  marker.pose = ToGeometryMsgPose(landmark_pose);
  return marker;
}
//获取 LandMark 的 Pose 列表
visualization_msgs::MarkerArray MapBuilderBridge::GetLandmarkPosesList() {
  visualization_msgs::MarkerArray landmark_poses_list;
  const std::map<std::string, Rigid3d> landmark_poses =
      map_builder_->pose_graph()->GetLandmarkPoses();
  for (const auto& id_to_pose : landmark_poses) {
    landmark_poses_list.markers.push_back(CreateLandmarkMarker(
        GetLandmarkIndex(id_to_pose.first, &landmark_to_index_),
```

```
      id_to_pose.second, node_options_.map_frame));

      }

      return landmark_poses_list;

}
```

4.3、构造函数

构造函数里只是做了一下实例化赋值：

```
MapBuilderBridge::MapBuilderBridge(

    const NodeOptions& node_options,

    std::unique_ptr<cartographer::mapping::MapBuilderInterface> map_builder,

    tf2_ros::Buffer* const tf_buffer): node_options_(node_options),

        map_builder_(std::move(map_builder)),

        tf_buffer_(tf_buffer) {}
```

4.4、LoadState 函数：

调用了 map_builder_的成员函数 LoadState 来加载一个.pbstream 文件。map_builder_是接口 MapBuilderInterface 的实例化对象，而根据是 2d 还是 3d 情况，其具体实现会略有不同。我们后面再细看。

```
void MapBuilderBridge::LoadState(const std::string& state_filename,

                                 bool load_frozen_state) {

  // Check if suffix of the state file is ".pbstream".

  const std::string suffix = ".pbstream";

  CHECK_EQ(state_filename.substr(

      std::max<int>(state_filename.size() - suffix.size(), 0)),suffix)
```

```
      << "The file containing the state to be loaded must be a "".pbstream file.";
  LOG(INFO) << "Loading saved state '" << state_filename << "'...";
  cartographer::io::ProtoStreamReader stream(state_filename);
  map_builder_->LoadState(&stream, load_frozen_state);
}
```

4.5、AddTrajectory

接下来是一个比较重要的函数：AddTrajectory。添加一条 Trajectory

```
int MapBuilderBridge::AddTrajectory(
    const std::set<cartographer::mapping::TrajectoryBuilderInterface::SensorId>&
        expected_sensor_ids,
    const TrajectoryOptions& trajectory_options) {
  const int trajectory_id = map_builder_->AddTrajectoryBuilder(
      expected_sensor_ids, trajectory_options.trajectory_builder_options,
      ::std::bind(&MapBuilderBridge::OnLocalSlamResult, this,
                  ::std::placeholders::_1, ::std::placeholders::_2,
                  ::std::placeholders::_3, ::std::placeholders::_4,
                  ::std::placeholders::_5));
  LOG(INFO) << "Added trajectory with ID '" << trajectory_id << "'.";
  // Make sure there is no trajectory with 'trajectory_id' yet.
  CHECK_EQ(sensor_bridges_.count(trajectory_id), 0);
  sensor_bridges_[trajectory_id] =
```

```
      cartographer::common::make_unique<SensorBridge>(
          trajectory_options.num_subdivisions_per_laser_scan,
          trajectory_options.tracking_frame,
          node_options_.lookup_transform_timeout_sec, tf_buffer_,
          map_builder_->GetTrajectoryBuilder(trajectory_id));
  auto emplace_result =
      trajectory_options_.emplace(trajectory_id, trajectory_options);
  CHECK(emplace_result.second == true);
  return trajectory_id;
}
```

其中，比较重要的就是这句：

```
  const int trajectory_id = map_builder_->AddTrajectoryBuilder(
      expected_sensor_ids, trajectory_options.trajectory_builder_options,
      ::std::bind(&MapBuilderBridge::OnLocalSlamResult, this,
          ::std::placeholders::_1, ::std::placeholders::_2,
          ::std::placeholders::_3, ::std::placeholders::_4,
          ::std::placeholders::_5));
```

调用了 map_builder_->AddTrajectoryBuilder，这已经是 cartographer 项目中的代码了。也就是说我们快把 cartographer_ros 的衣服剥的差不多了。可以看到传入的数据有传感器的 id，trajectory 的一些配置参数，以及一个仿函数对象

```
::std::bind(&MapBuilderBridge::OnLocalSlamResult, this,
          ::std::placeholders::_1, ::std::placeholders::_2,
          ::std::placeholders::_3, ::std::placeholders::_4,
```

```
        ::std::placeholders::_5);
```

这里又涉及到了一些 c++的特性，补充知识如下：

```
/*
* std::bind 用来将可调用对象与其参数进行绑定。绑定之后的结果可以使用 std::function 进行保存，
并延迟调用到任何需要的时候。一般来讲，它主要有两大作用：
*（1）将可调用对象与其参数一起绑定成为一个仿函数
*（2）将多元可调用对象转换成为 1 元或是（n-1）元调用对象，既只是绑定部分参数
*实际上 std::bind 的返回类型是一个 std 内部定义的仿函数类型，在这里就只需要知道它是一个仿函
数，可以赋值给一个 std::function，这里直接用 std::function 类型来保存 std::bind 的返回值也是可
以的。
*其中 std::placeholders::_1 是一个占位符，代表这个文职将在函数调用时，被传入的第一个参数代
替。
*
*所以 map_builder_->AddTrajectoryBuilder 这个函数的第三个参数是一个 std:function 型的。
*这样，在 map_builder_->AddTrajectoryBuilder 内部可以通过如下方式调用
MapBuilderBridge::OnLocalSlamResult
*call_func(para1,para2,..., std::function);
****/
```

而函数 MapBuilderBridge::OnLocalSlamResult 则如下定义：

```
void MapBuilderBridge::OnLocalSlamResult(
    const int trajectory_id, const ::cartographer::common::Time time,
    const Rigid3d local_pose,::cartographer::sensor::RangeData range_data_in_local,
    consts td::unique_ptr<const ::cartographer::mapping::
                          TrajectoryBuilderInterface::InsertionResult>){
  std::shared_ptr<const TrajectoryState::LocalSlamData> local_slam_data =
      std::make_shared<TrajectoryState::LocalSlamData>(
          TrajectoryState::LocalSlamData{time, local_pose,
                                         std::move(range_data_in_local)});
  cartographer::common::MutexLocker lock(&mutex_);
```

```
    trajectory_state_data_[trajectory_id] = std::move(local_slam_data);
  }
```

如上所示，该函数有 5 个参数。所以，在 map_builder_->AddTrajectoryBuilder 内部调用时就需要传入这 5 个参数。等我们看到 MapBuilder 这个类时我们再回过来对照着看。

不过，这里我不太明白的是为什么会先 AddTrajectoryBuilder，然后在检查是否已有指定 ID 的 trajectory。按照我们通常的逻辑，不应该是先检查是否存在该 ID，然后再添加吗？暂时跳过这个问题，以后再说。

4.6、接下来三个函数也都调用了 map_builder_的成员函数

```
void MapBuilderBridge::FinishTrajectory(const int trajectory_id) {
  LOG(INFO) << "Finishing trajectory with ID '" << trajectory_id << "'...";
  // Make sure there is a trajectory with 'trajectory_id'.
  CHECK_EQ(sensor_bridges_.count(trajectory_id), 1);
  map_builder_->FinishTrajectory(trajectory_id);
  sensor_bridges_.erase(trajectory_id);
}

void MapBuilderBridge::RunFinalOptimization() {
  LOG(INFO) << "Running final trajectory optimization...";
  map_builder_->pose_graph()->RunFinalOptimization();
}

bool MapBuilderBridge::SerializeState(const std::string& filename) {
  cartographer::io::ProtoStreamWriter writer(filename);
  map_builder_->SerializeState(&writer);
  return writer.Close();
```

```
}
```

4.7、HandleSubmapQuery

处理 submap 查询的，在 cartographer_node 中被 kSubmapQueryServiceName 这个 Service 调用。

```
void MapBuilderBridge::HandleSubmapQuery(
    cartographer_ros_msgs::SubmapQuery::Request& request,
    cartographer_ros_msgs::SubmapQuery::Response& response){
  cartographer::mapping::proto::SubmapQuery::Response response_proto;
  cartographer::mapping::SubmapId submap_id{request.trajectory_id,
                                            request.submap_index};
  const std::string error = map_builder_->SubmapToProto(submap_id, &response_proto);
  if (!error.empty()) {
    LOG(ERROR) << error;
    response.status.code = cartographer_ros_msgs::StatusCode::NOT_FOUND;
    response.status.message = error;
    return;
  }
  response.submap_version = response_proto.submap_version();
  for (const auto& texture_proto : response_proto.textures()) {
    response.textures.emplace_back();
    auto& texture = response.textures.back();
    texture.cells.insert(texture.cells.begin(), texture_proto.cells().begin(),
```

```
                        texture_proto.cells().end());
    texture.width = texture_proto.width();
    texture.height = texture_proto.height();
    texture.resolution = texture_proto.resolution();
    texture.slice_pose = ToGeometryMsgPose(
        cartographer::transform::ToRigid3(texture_proto.slice_pose()));
  }
  response.status.message = "Success.";
  response.status.code = cartographer_ros_msgs::StatusCode::OK;
}
```

调用 map_builder_->SubmapToProto(submap_id, &response_proto);这个函数查询信息，结果放到 response 这个变量中。

4.8、 GetFrozenTrajectoryIds

接下来 GetFrozenTrajectoryIds 这个函数从名字看是冻结一个 trajectory.不太明白具体是什么意思，个人猜测是一条 trajectory 已经构建完毕，不再继续扩展之后是不是就把其 Frozen 并存起来。当然，需要调用 map_builder_->pose_graph()->GetTrajectoryNodePoses()和 map_builder_->pose_graph()->IsTrajectoryFrozen(trajectory_id)来处理。

```
std::set<int> MapBuilderBridge::GetFrozenTrajectoryIds() {
  std::set<int> frozen_trajectory_ids;
  const auto node_poses = map_builder_->pose_graph()->GetTrajectoryNodePoses();
  for (const int trajectory_id : node_poses.trajectory_ids()) {
    if (map_builder_->pose_graph()->IsTrajectoryFrozen(trajectory_id)) {
      frozen_trajectory_ids.insert(trajectory_id);
```

```
      }
    }
    return frozen_trajectory_ids;
  }
```

4.9、GetSubmapList()

接下来，GetSubmapList()，这个函数是在往 kSubmapListTopic 这个 Topic 上发布数据时被 Node::PublishSubmapList 调用的，用来获取 Submap 的列表：

```
cartographer_ros_msgs::SubmapList MapBuilderBridge::GetSubmapList(){
  cartographer_ros_msgs::SubmapList submap_list;
  submap_list.header.stamp = ::ros::Time::now();
  submap_list.header.frame_id = node_options_.map_frame;
  for (const auto& submap_id_pose :map_builder_->pose_graph()->GetAllSubmapPoses()) {
      cartographer_ros_msgs::SubmapEntry submap_entry;
    submap_entry.trajectory_id = submap_id_pose.id.trajectory_id;
    submap_entry.submap_index = submap_id_pose.id.submap_index;
    submap_entry.submap_version = submap_id_pose.data.version;
    submap_entry.pose = ToGeometryMsgPose(submap_id_pose.data.pose);
    submap_list.submap.push_back(submap_entry);
    }
    return submap_list;
}
```

该函数主要也是通过调用 map_builder_->pose_graph()->GetAllSubmapPoses()来获取列表信息。

4.10、GetTrajectoryStates

GetTrajectoryStates 用来返回一个 TrajectoryStates 变量组成的 unordered_map 这个容器。TrajectoryStates 这个结构体的定义如上，见 MapBuilderBridge 的成员函数。这里稍微注意一下，Unordered Map 跟我们程序里 MapBuilder 里的 map 不是同一个概念。Unordered Map 只是 c++中的一种 container，详见：unordered_map

```
std::unordered_map<int, MapBuilderBridge::TrajectoryState>

MapBuilderBridge::GetTrajectoryStates() {

    std::unordered_map<int, TrajectoryState> trajectory_states;     //变量用来存返回结果

    for (const auto& entry : sensor_bridges_) {

      //一个循环，依次取出来 TrajectoryState;这里以 SensorBridge 为索引来取，还不知道为什么

      const int trajectory_id = entry.first;

      const SensorBridge& sensor_bridge = *entry.second;//

      std::shared_ptr<const TrajectoryState::LocalSlamData> local_slam_data;

      {

          cartographer::common::MutexLocker lock(&mutex_);

          if (trajectory_state_data_.count(trajectory_id) == 0) {

            continue;

          }

          local_slam_data = trajectory_state_data_.at(trajectory_id);

        }//TrajectoryState 结构体中的第一个成员 LocalSlamData

        // Make sure there is a trajectory with 'trajectory_id'.

        CHECK_EQ(trajectory_options_.count(trajectory_id), 1);

      trajectory_states[trajectory_id] = {//第 trajectory_id 个 TrajectoryState 存入返回变量中
```

```
        local_slam_data,
        map_builder_->pose_graph()->GetLocalToGlobalTransform(trajectory_id),//
        sensor_bridge.tf_bridge().LookupToTracking(
            local_slam_data->time,
            trajectory_options_[trajectory_id].published_frame),
        trajectory_options_[trajectory_id]};
  }
  return trajectory_states;
}
```

其中，第二个成员变量 std::shared_ptr<const LocalSlamData> local_slam_data;是通过 map_builder_->pose_graph()->GetLocalToGlobalTransform(trajectory_id)获取的，第三个成员变量 std::unique_ptr<cartographer::transform::Rigid3d> published_to_tracking;是通过 sensor_bridge.tf_bridge().LookupToTracking 获取的。

第 5 章 MapBuilderBridge 后续及 SensorBridge

5.1、GetTrajectoryNodeList

对于 TrajectoryNode 我还没有很好地理解。看代码其中是通过 map_builder_->pose_graph()->GetTrajectoryNodePoses()来获取 TrajectoryNode 的 pose 的。map_builder_->pose_graph()->constraints()获取 Constraints。但这个 node 和 constraint 具体指什么呢？是建图过程中的一些节点和他们之前的约束？猜测这块儿可能跟 GrapherSlam 这方便的东西相关。PushAndResetLineMarker 和 ToGeometryMsgPoint 应该都是与显示相关的部分。但是没找到 ToGeometryMsgPoint 函数在哪儿定义。

```
visualization_msgs::MarkerArray MapBuilderBridge::GetTrajectoryNodeList() {
  visualization_msgs::MarkerArray trajectory_node_list;
  const auto node_poses = map_builder_->pose_graph()->GetTrajectoryNodePoses();
  // Find the last node indices for each trajectory that have either
  // inter-submap or inter-trajectory constraints.
  std::map<int, int /* node_index */> trajectory_to_last_inter_submap_constrained_node;
  std::map<int, int /* node_index */> trajectory_to_last_inter_trajectory_constrained_node;
  for (const int trajectory_id : node_poses.trajectory_ids()) {
    trajectory_to_last_inter_submap_constrained_node[trajectory_id] = 0;
    trajectory_to_last_inter_trajectory_constrained_node[trajectory_id] = 0;
    }
    const auto constraints = map_builder_->pose_graph()->constraints();
    for (const auto& constraint : constraints) {
      if (constraint.tag == cartographer::mapping::PoseGraph::Constraint::INTER_SUBMAP) {
        if (constraint.node_id.trajectory_id == constraint.submap_id.trajectory_id) {
```

```
        trajectory_to_last_inter_submap_constrained_node[constraint.node_id.trajectory_id] =
            std::max(trajectory_to_last_inter_submap_constrained_node.at(
                constraint.node_id.trajectory_id),constraint.node_id.node_index);
        } else {trajectory_to_last_inter_trajectory_constrained_node
            [constraint.node_id.trajectory_id] =
                std::max(trajectory_to_last_inter_submap_constrained_node.at(
                    constraint.node_id.trajectory_id),
                    constraint.node_id.node_index);
          }
      }
}
for (const int trajectory_id : node_poses.trajectory_ids()) {
  visualization_msgs::Marker marker =
      CreateTrajectoryMarker(trajectory_id, node_options_.map_frame);
  int last_inter_submap_constrained_node = std::max(
      node_poses.trajectory(trajectory_id).begin()->id.node_index,
      trajectory_to_last_inter_submap_constrained_node.at(trajectory_id));
  int last_inter_trajectory_constrained_node = std::max(
      node_poses.trajectory(trajectory_id).begin()->id.node_index,
      trajectory_to_last_inter_trajectory_constrained_node.at(trajectory_id));
  last_inter_submap_constrained_node =
      std::max(last_inter_submap_constrained_node,
```

```
            last_inter_trajectory_constrained_node);
if (map_builder_->pose_graph()->IsTrajectoryFrozen(trajectory_id)) {
  last_inter_submap_constrained_node =
      (--node_poses.trajectory(trajectory_id).end())->id.node_index;
    last_inter_trajectory_constrained_node =last_inter_submap_constrained_node;
}
marker.color.a = 1.0;
for (const auto& node_id_data : node_poses.trajectory(trajectory_id)) {
  if (!node_id_data.data.constant_pose_data.has_value()) {
      PushAndResetLineMarker(&marker, &trajectory_node_list.markers);
      continue;
 }
 const ::geometry_msgs::Point node_point =
     ToGeometryMsgPoint(node_id_data.data.global_pose.translation());
  marker.points.push_back(node_point);
  if (node_id_data.id.node_index ==last_inter_trajectory_constrained_node) {
    PushAndResetLineMarker(&marker, &trajectory_node_list.markers);
    marker.points.push_back(node_point);
    marker.color.a = 0.5;
  }
  if (node_id_data.id.node_index == last_inter_submap_constrained_node) {
    PushAndResetLineMarker(&marker, &trajectory_node_list.markers);
```

```
        marker.points.push_back(node_point);
        marker.color.a = 0.25;
    }
    // Work around the 16384 point limit in RViz by splitting the
    // trajectory into multiple markers.
    if (marker.points.size() == 16384) {
      PushAndResetLineMarker(&marker, &trajectory_node_list.markers);
      // Push back the last point, so the two markers appear connected.
      marker.points.push_back(node_point);
    }
}
PushAndResetLineMarker(&marker, &trajectory_node_list.markers);
size_t current_last_marker_id = static_cast<size_t>(marker.id - 1);
if (trajectory_to_highest_marker_id_.count(trajectory_id) == 0) {
  trajectory_to_highest_marker_id_[trajectory_id] = current_last_marker_id;
} else {
  marker.action = visualization_msgs::Marker::DELETE;
  while (static_cast<size_t>(marker.id) <=
         trajectory_to_highest_marker_id_[trajectory_id]) {
    trajectory_node_list.markers.push_back(marker);
    ++marker.id;
  }
```

```
        trajectory_to_highest_marker_id_[trajectory_id] = current_last_marker_id;
      }
    }
    return trajectory_node_list;
}
```

5.2、GetConstraintList

获取 Constraint. 应该是与 GetTrajectoryStateNodeList 配合使用的。

```
visualization_msgs::MarkerArray MapBuilderBridge::GetConstraintList() {
  visualization_msgs::MarkerArray constraint_list;
  int marker_id = 0;
  visualization_msgs::Marker constraint_intra_marker;
  constraint_intra_marker.id = marker_id++;
  constraint_intra_marker.ns = "Intra constraints";
  constraint_intra_marker.type = visualization_msgs::Marker::LINE_LIST;
  constraint_intra_marker.header.stamp = ros::Time::now();
  constraint_intra_marker.header.frame_id = node_options_.map_frame;
  constraint_intra_marker.scale.x = kConstraintMarkerScale;
  constraint_intra_marker.pose.orientation.w = 1.0;
  visualization_msgs::Marker residual_intra_marker = constraint_intra_marker;
  residual_intra_marker.id = marker_id++;
  residual_intra_marker.ns = "Intra residuals";
```

```
// This and other markers which are less numerous are set to be slightly

// above the intra constraints marker in order to ensure that they are

// visible.

residual_intra_marker.pose.position.z = 0.1;

visualization_msgs::Marker constraint_inter_same_trajectory_marker =

    constraint_intra_marker;

constraint_inter_same_trajectory_marker.id = marker_id++;

onstraint_inter_same_trajectory_marker.ns =

    "Inter constraints, same trajectory";

constraint_inter_same_trajectory_marker.pose.position.z = 0.1;

visualization_msgs::Marker residual_inter_same_trajectory_marker =

    constraint_intra_marker;

residual_inter_same_trajectory_marker.id = marker_id++;

residual_inter_same_trajectory_marker.ns = "Inter residuals, same trajectory";

residual_inter_same_trajectory_marker.pose.position.z = 0.1;

visualization_msgs::Marker constraint_inter_diff_trajectory_marker =

    constraint_intra_marker;

constraint_inter_diff_trajectory_marker.id = marker_id++;

constraint_inter_diff_trajectory_marker.ns =

    "Inter constraints, different trajectories";

constraint_inter_diff_trajectory_marker.pose.position.z = 0.1;

visualization_msgs::Marker residual_inter_diff_trajectory_marker =
```

```
constraint_intra_marker;
residual_inter_diff_trajectory_marker.id = marker_id++;
residual_inter_diff_trajectory_marker.ns =
    "Inter residuals, different trajectories";
residual_inter_diff_trajectory_marker.pose.position.z = 0.1;
const auto trajectory_node_poses =
    map_builder_->pose_graph()->GetTrajectoryNodePoses();
const auto submap_poses = map_builder_->pose_graph()->GetAllSubmapPoses();
const auto constraints = map_builder_->pose_graph()->constraints();
for (const auto& constraint : constraints) {
  visualization_msgs::Marker *constraint_marker, *residual_marker;
  std_msgs::ColorRGBA color_constraint, color_residual;
  if (constraint.tag ==
    cartographer::mapping::PoseGraph::Constraint::INTRA_SUBMAP) {
    constraint_marker = &constraint_intra_marker;
    residual_marker = &residual_intra_marker;
    // Color mapping for submaps of various trajectories - add trajectory id
    // to ensure different starting colors. Also add a fixed offset of 25
    // to avoid having identical colors as trajectories.
    color_constraint = ToMessage(
    cartographer::io::GetColor(constraint.submap_id.submap_index +
    constraint.submap_id.trajectory_id + 25));
```

```
    color_residual.a = 1.0;
    color_residual.r = 1.0;
  } else {
  if (constraint.node_id.trajectory_id ==
    constraint.submap_id.trajectory_id) {
    constraint_marker = &constraint_inter_same_trajectory_marker;
    residual_marker = &residual_inter_same_trajectory_marker;
    // Bright yellow
    color_constraint.a = 1.0;
    color_constraint.r = color_constraint.g = 1.0;
  }else{
    constraint_marker = &constraint_inter_diff_trajectory_marker;
    residual_marker = &residual_inter_diff_trajectory_marker;
    // Bright orange
    color_constraint.a = 1.0;
    color_constraint.r = 1.0;
    color_constraint.g = 165. / 255.;
  }
    // Bright cyan
    color_residual.a = 1.0;
    color_residual.b = color_residual.g = 1.0;
  }
```

```
  for (int i = 0; i < 2; ++i) {
    constraint_marker->colors.push_back(color_constraint);
    residual_marker->colors.push_back(color_residual);
}
const auto submap_it = submap_poses.find(constraint.submap_id);
if (submap_it == submap_poses.end()) {
  continue;
}
const auto& submap_pose = submap_it->data.pose;
const auto node_it = trajectory_node_poses.find(constraint.node_id);
if (node_it == trajectory_node_poses.end()) {
  continue;
}
const auto& trajectory_node_pose = node_it->data.global_pose;
const Rigid3d constraint_pose = submap_pose * constraint.pose.zbar_ij;
constraint_marker->points.push_back(
    ToGeometryMsgPoint(submap_pose.translation()));
constraint_marker->points.push_back(
    ToGeometryMsgPoint(constraint_pose.translation()));
residual_marker->points.push_back(
    ToGeometryMsgPoint(constraint_pose.translation()));
residual_marker->points.push_back(
```

```
        ToGeometryMsgPoint(trajectory_node_pose.translation()));
  }
  constraint_list.markers.push_back(constraint_intra_marker);
  constraint_list.markers.push_back(residual_intra_marker);
  constraint_list.markers.push_back(constraint_inter_same_trajectory_marker);
  constraint_list.markers.push_back(residual_inter_same_trajectory_marker);
  constraint_list.markers.push_back(constraint_inter_diff_trajectory_marker);
  constraint_list.markers.push_back(residual_inter_diff_trajectory_marker);
  return constraint_list;
}
```

5.3、sensor_bridge

返回一个 SensorBridge 的指针。

```
SensorBridge* MapBuilderBridge::sensor_bridge(const int trajectory_id) {
  return sensor_bridges_.at(trajectory_id).get();
}
```

我们应该加快一下我们工作的进度了。所以之后我们简单查看一下 SensorBridge 这个类之后就要调到 MapBuilder 这个类里去了。目前来看，所有工作都是由 MapBuilder 来完成的。

在前面我们阅读 cartographer_node 这个节点相关的代码(cartographer 源码阅读 3——从 cartographer_node 入手 2)时知道: 在处理传感器的数据时，都是通过调用 map_builder_bridge_的一个类成员 sensor_bridge_ptr 来处理的。在阅读 MapBuilderBridge 的源码后我们可以看到跟我们最初的理解略有出入，sensor_bridge_ptr 是在 MapBuilderBridge 中定义的一个 SensorBridge 类型的局部变量，MapBuilderBridge 中一个函数 MapBuilderBridge::sensor_bridge 可以返回指定 trajectory_id 的 SensorBridge 变量。然后在 SensorBridge 中处理相关传感器的信息。以 IMU 为例，在 cartographer_node 中由 Node::HandleImuMessage 函数处理，该函数实际调用的是 map_builder_bridge_->sensor_bridge_ptr->HandleImuMessage。

所以我们接下来看一下 SensorBridge 中具体处理各个传感器的函数。

1. 构造函数等

没啥说的，把构造函数的参数赋值给成员变量。但需要注意的是成员变量中有一个 TrajectoryBuilderInterface 型的一个指针变量。继续跟踪代码我们可以发现，cartographer 中各种消息都统一调用了这个成员类的虚函数 AddSensorData(). 而 CollatedTrajectoryBuilder 继承了这个类并实现了 AddSensorData()函数。这两个类都定义在 cartographer 中的 mapping 文件夹下。CollatedTrajectoryBuilder 的构造函数说明通过统一调用 HandleCollatedSensorData()函数，来轮询处理 kImu(IMU 消息)、kRangefinder(测距消息，不仅仅是激光)、kOdometer(里程计消息)等。所以，我们到后面可以再看这两个类。

```
//构造函数做的工作就是把参数表赋值给成员函数
SensorBridge::SensorBridge(
    const int num_subdivisions_per_laser_scan,
    const std::string& tracking_frame,
    const double lookup_transform_timeout_sec, tf2_ros::Buffer* const tf_buffer,
    carto::mapping::TrajectoryBuilderInterface* const trajectory_builder)
    : num_subdivisions_per_laser_scan_(num_subdivisions_per_laser_scan),
      tf_bridge_(tracking_frame, lookup_transform_timeout_sec, tf_buffer),
      trajectory_builder_(trajectory_builder) {}
```

2. 处理里程计的函数

```
//一个预处理的工具函数，并非 SensorBridge 的成员变量；
//其参数类型是 nav_msgs::Odometry::ConstPtr&,
std::unique_ptr<carto::sensor::OdometryData> SensorBridge::ToOdometryData(
    const nav_msgs::Odometry::ConstPtr& msg) {
  const carto::common::Time time = FromRos(msg->header.stamp);
  const auto sensor_to_tracking = tf_bridge_.LookupToTracking(
```

```
      time, CheckNoLeadingSlash(msg->child_frame_id));//该函数返回的是一个变换矩阵，查询的是某时刻某一帧数据的变换估计。估计要用来做累加
    if (sensor_to_tracking == nullptr) {
        return nullptr;
    }
    return carto::common::make_unique<carto::sensor::OdometryData>(
      carto::sensor::OdometryData{
          time, ToRigid3d(msg->pose.pose) * sensor_to_tracking->inverse()});
}
void SensorBridge::HandleOdometryMessage(
    const std::string& sensor_id, const nav_msgs::Odometry::ConstPtr& msg) {
  std::unique_ptr<carto::sensor::OdometryData> odometry_data =ToOdometryData(msg);
  if (odometry_data != nullptr) {
      trajectory_builder_->AddSensorData(
          sensor_id,
          carto::sensor::OdometryData{odometry_data->time, odometry_data->pose});
  }
}
```

这个需要注意的是 TfBridge 这个类，他的实例化对象是 tf_bridge_，tf_bridge_是 SensorBridge 的一个成员变量。我们通过查看 TfBridge 可以看到，作者在设计的时候，通过 TfBridge 把不同传感器的差异进行了一下屏蔽，不管什么传感器，都会对每一帧数据的位姿进行估计，而这个历史的位姿数据就以 TfBridge 的形式存起来。这样在使用的时候也可以通过 TfBridge 查询某一个传感器在某一个历史时刻的位姿。

这里同样也是，代码通过 TfBridge 查询了一下历史数据，然后把这个作为参数传给了 TrajectoryBuilder 的 AddSensorData 来做处理。前面我们已经介绍过了，TrajectoryBuilder 也是为不

同的传感器提供了统一的处理接口。我们可以在 TrajectoryBuilder 的具体实现里再看具体做了什么操作。但我们可以合理猜测，比如，里程计的数据是在原来数据的基础上再做一个累加，作为新的值同样保存到 TfBridge 里。

我们先不用管这个猜测是否正确，等到后面再看。至少我们现在知道传感器的数据是由谁来处理了。

以里程计数据为例，我们可以梳理一下传感器数据整个的处理流程：

传感器的 ROS 节点/Playbag 广播到 Topic 上相关数据的 Message---->cartographer_node 中启动的 StartTrajectory 这个服务会订阅传感器数据---->接收到该数据由相应的处理函数处理，比如 Node::HandleImuMessage---->该处理函数实际调用是 MapBuilderBridge 中的一个 SensorBridge 变量进行处理---->调用了 TrajectoryBuilder 的虚函数 AddSensorData()---->CollatedTrajectoryBuilder 继承 TrajectoryBuilder 并具体实现 AddSensorData()函数

同样，其他传感器的处理也让我们粗略看一下：

3. HandleNavSatFixMessage

NavSatFix 之前不是特别确定是什么传感器。后来查了一下，可能是 GPS 的数据，返回的是在世界坐标系下的坐标 x,y,z。但是对于室内机器人来说，GPS 无法应用，GPS 信号对于室外机器人会更有用处。cartographer 在提供的官方测试数据里，也是没有使用 GPS 数据的（见 /src/cartographer_ros/configuration_files/backpack_2d.lua 中设置 use_nav_sat = false）。所以，这里我们就先不管该传感器相关的代码。

```
void SensorBridge::HandleNavSatFixMessage(
    const std::string& sensor_id, const sensor_msgs::NavSatFix::ConstPtr& msg) {
  const carto::common::Time time = FromRos(msg->header.stamp);
  if (msg->status.status == sensor_msgs::NavSatStatus::STATUS_NO_FIX) {
    trajectory_builder_->AddSensorData(
        sensor_id, carto::sensor::FixedFramePoseData{
            time, carto::common::optional<Rigid3d>()});
    return;
  }
  if (!ecef_to_local_frame_.has_value()) {
```

```
    ecef_to_local_frame_ =
        ComputeLocalFrameFromLatLong(msg->latitude, msg->longitude);
    LOG(INFO) << "Using NavSatFix. Setting ecef_to_local_frame with lat = "
              << msg->latitude << ", long = " << msg->longitude << ".";
  }
  trajectory_builder_->AddSensorData(
      sensor_id,
      carto::sensor::FixedFramePoseData{
          time, carto::common::optional<Rigid3d>(Rigid3d::Translation(
                    ecef_to_local_frame_.value() *
                    LatLongAltToEcef(msg->latitude, msg->longitude,
                                     msg->altitude)))});
}
```

4. 处理 Landmark

```
void SensorBridge::HandleLandmarkMessage(
    const std::string& sensor_id,
    const cartographer_ros_msgs::LandmarkList::ConstPtr& msg) {
  trajectory_builder_->AddSensorData(sensor_id, ToLandmarkData(*msg));
}
```

5. 处理 IMU 数据

```
//同样，数据预处理函数。并非 SensorBridge 的成员函数
std::unique_ptr<carto::sensor::ImuData> SensorBridge::ToImuData(
```

```
                                    const sensor_msgs::Imu::ConstPtr& msg) {
//确保 IMU 工作正常
CHECK_NE(msg->linear_acceleration_covariance[0], -1)
    << "Your IMU data claims to not contain linear acceleration measurements "
       "by setting linear_acceleration_covariance[0] to -1. Cartographer "
       "requires this data to work. See "
       "http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html.";
CHECK_NE(msg->angular_velocity_covariance[0], -1)
    << "Your IMU data claims to not contain angular velocity measurements "
       "by setting angular_velocity_covariance[0] to -1. Cartographer "
       "requires this data to work. See "
       "http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html.";
const carto::common::Time time = FromRos(msg->header.stamp);
const auto sensor_to_tracking = tf_bridge_.LookupToTracking(
    time, CheckNoLeadingSlash(msg->header.frame_id));
if (sensor_to_tracking == nullptr) {
  return nullptr;
}
CHECK(sensor_to_tracking->translation().norm() < 1e-5)
    << "The IMU frame must be colocated with the tracking frame. "
       "Transforming linear acceleration into the tracking frame will "
       "otherwise be imprecise.";
```

```
  return carto::common::make_unique<carto::sensor::ImuData>(
      carto::sensor::ImuData{time,
          sensor_to_tracking->rotation() * ToEigen(msg->linear_acceleration),
          sensor_to_tracking->rotation() * ToEigen(msg->angular_velocity)});
}
//最终，将先加速度和角加速度传入 trajectory_builder_->AddSensorData 做处理
void SensorBridge::HandleImuMessage(const std::string& sensor_id,
     const sensor_msgs::Imu::ConstPtr& msg) {
         std::unique_ptr<carto::sensor::ImuData> imu_data = ToImuData(msg);
         if (imu_data != nullptr) {trajectory_builder_->AddSensorData(
            sensor_id,carto::sensor::ImuData{imu_data->time, imu_data->linear_acceleration,
            imu_data->angular_velocity});
  }
}
```

6. 处理激光、多激光、点云数据

这几个函数初看似乎没有调用 trajectory-->AddSensorData 函数。但仔细研究会发现他们之间彼此存在调用关系。最根上是 HandleRangefinder 这个函数：这个函数调用了 trajectory_builder_->AddSensorData 来处理

```
void SensorBridge::HandleRangefinder(
    const std::string& sensor_id, const carto::common::Time time,
    const std::string& frame_id, const carto::sensor::TimedPointCloud& ranges) {
        const auto sensor_to_tracking = tf_bridge_.LookupToTracking(time,
        CheckNoLeadingSlash(frame_id));
```

```
    if (sensor_to_tracking != nullptr) {
        trajectory_builder_->AddSensorData(
            sensor_id, carto::sensor::TimedPointCloudData{
                time, sensor_to_tracking->translation().cast<float>(),
                carto::sensor::TransformTimedPointCloud(
                ranges, sensor_to_tracking->cast<float>())});
    }
}
```

而之后，激光数据函数 SensorBridge::HandleLaserScan 调用了 SensorBridge::HandleRangefinder 来做处理。所以这里相当于做了一层抽象。我们的 Rangefinder 可以不一样是激光，也可以是其他类型的传感器，比如 Kinect。这样，以后如果要扩展或修改，我们可以不改之前的代码，而只需要多写一个处理 Kinect 的代码就可以。这也是封装的好处。所有做的这些复杂的工作，都是为了程序员维护的方便，但对于读别人代码的人来说，要理清作者的设计思路，那可就需要费一番功夫了。

然后，SensorBridge::HandlePointCloud2Message 也是调用了 SensorBridge::HandleRangefinder 处理。这种设计使得重复性比较大的 SensorBridge::HandleRangefinder 中的代码得以复用。

SensorBridge::HandleLaserScanMessage 和 SensorBridge::HandleMultiEchoLaserScanMessage 则调用了 SensorBridge::HandleLaserScan 来做处理。

好了，cartographer_ros 部分的代码我们就看到这里，可以看出来，工作都交给了 MapBuilder 和 TrajectoryBuilder 这两个类。所以，应该可以详见，在 cartographer 这部分的代码中，应该是以这两个类为核心开展的。所以，下一篇文章我们将以 MapBuilder 和 TrajectoryBuilder 为核心来查看代码。

第 6 章 MapBuilder 的解读

通过 MapBuilderBridge 和 SensorBridge 这两个类的过渡，我们正式从 cartographer_ros 转到了 cartographer 部分了。从之前部分我们可知，MapBuilderBridge 和 SensorBridge 主要调用了 MapBuilderInterface 和 TrajectoryBuilderInterface 这两个类的成员函数来处理。所以，在 cartographer 部分，我们将从这两个部分入手。

本来我的计划是先看 TrajectoryBuilderInterface 这部分，结果从这部分代码入手读起来感觉特别混乱，理不清头绪。所以，我觉得回过头来先从 MapBuilder 这部分代码入手。

查看这一部分代码后我们发现这部分代码中的结构与早期的一些博客，如“ Cartographer 代码阅读分析 ”等相比，代码结构已经发生了较大的变化。所以早期的一些博客资料已经不再具有参考价值。所以接下来的代码解读过程只能靠我们自己的理解了。在接下来的过程中我们将以实际代码为准，分析程序结构和设计思路。

6.1、接口 MapBuilderInterface 的定义：

MapBuilderInterface 定义在/mapping/map_builder_interface.h 中，是一个接口类，其中定义了一系列纯虚函数：

```
// 创建一个 TrajectoryBuilder 并返回他的 index，即 trajectory_id

virtual int AddTrajectoryBuilder(

    const std::set<SensorId>& expected_sensor_ids,

    const proto::TrajectoryBuilderOptions& trajectory_options,

    LocalSlamResultCallback local_slam_result_callback) = 0;

/*

* 这里注意，LocalSlamResultCallback 是一个回调函数。在 map_builder_interface.h 中有：

* using LocalSlamResultCallback = TrajectoryBuilderInterface::LocalSlamResultCallback;

* 查看 TrajectoryBuilderInterface 的定义文件/mapping/trajectory_builder_interface.h:

*   using LocalSlamResultCallback =

*      std::function<void(int /* trajectory ID */,
```

```
*                         common::Time,//时间
*                         transform::Rigid3d /* local pose estimate */,
*                         sensor::RangeData /* in local frame */,
*                         std::unique_ptr<const InsertionResult>)>;
*/
// Serialization 是序列化，Deserialization 是反序列化。我们后面补充关于序列化和反序列的知识
// 所以这个函数的作用是从一个序列化的数据中构造出一个 trajectory 并返回他的 index
virtual int AddTrajectoryForDeserialization(//只有一个参数，就是序列化的数据
    const proto::TrajectoryBuilderOptionsWithSensorIds& options_with_sensor_ids_proto) = 0;
// 根据 trajectory_id 返回一个 TrajectoryBuilderInterface 的指针。
// 如果该 trajectory 没有一个 TrajectoryBuilder，则返回 nullptr’ corresponding to the
specified
virtual mapping::TrajectoryBuilderInterface* GetTrajectoryBuilder(
   int trajectory_id) const = 0;
// Finish 指定 id 的 trajectory.
// i.e. no further sensor data is expected.
virtual void FinishTrajectory(int trajectory_id) = 0;
// 根据指定的 submap_id 来查询 submap，把结果放到 SubmapQuery::Response 中。
// 如果出现错误，返回 error string; 成功则返回 empty string.
virtual std::string SubmapToProto(
    const SubmapId& submap_id,
    proto::SubmapQuery::Response* response) = 0;
```

```
// 序列化当前状态到一个 proto 流中。

virtual void SerializeState(io::ProtoStreamWriterInterface* writer) = 0;

// 从一个 proto 流中加载 SLAM 状态

virtual void LoadState(io::ProtoStreamReaderInterface* reader,bool load_frozen_state) = 0;

// 返回系统中当前已有的 trajectory_builder 的数量

virtual int num_trajectory_builders() const = 0;

// 返回一个 PoseGraphInterface 的接口指针。后面我们可以看到，PoseGrapher 用来进行 Loop
Closure。

virtual mapping::PoseGraphInterface* pose_graph() = 0;

// 获取所有 TrajectoryBuilder 的配置项。

virtual const std::vector<proto::TrajectoryBuilderOptionsWithSensorIds>&

GetAllTrajectoryBuilderOptions() const = 0;
```

在前面包括今后，我们多次碰到了 proto 这个概念，那么这到底是什么意思呢？这里我们进行一个知识补充：

知识补充：最常用的 C++序列化方案：protobuf
参考链接：最常用的两种 C++序列化方案的使用心得（protobuf 和 boost serialization）
[c++] Google Protobuf 库
1. 什么是序列化？

程序员在编写应用程序的时候往往需要将程序的某些数据存储在内存中，然后将其写入某个文件或是将它传输到网络中的另一台计算机上以实现通讯。这个将 程序数据转化成能被存储并传输的格式的过程被称为“序列化”（Serialization），而它的逆过程则可被称为“反序列化” （Deserialization）。简单来说，序列化就是将对象实例的状态转换为可保持或传输的格式的过程。与序列化相对的是反序列化，它根据流重构对象。这两个过程结合起来，可以轻 松地存储和传输数据。例如，可以序列化一个对象，然后使用 HTTP 通过 Internet 在客户端和服务器之间传输该对象。

总结：序列化：将对象变成字节流的形式传出去。

反序列化：从字节流恢复成原来的对象。

2. 为什么要序列化？好处在哪里？

简单来说，对象序列化通常用于两个目的：

（1） 将对象存储于硬盘上 ，便于以后反序列化使用

（2）在网络上传送对象的字节序列

对象序列化的好处在哪里？网络传输方面的便捷性、灵活性就不说了，这里举个我们经常可能发生的

需求：你 有一个数据结构，里面存储的数据是经过很多其它数据通过非常复杂的算法生成的，由于数据量很大，算法又复杂，因此生成该数据结构所用数据的时间可能要很久 （也许几个小时，甚至几天），生成该数据结构后又要用作其它的计算，那么你在调试阶段，每次运行个程序，就光生成数据结构就要花上这么长的时间，无疑代价 是非常大的。如果你确定生成数据结构的算法不会变或不常变，那么就可以通过序列化技术生成数据结构数据存储到磁盘上，下次重新运行程序时只需要从磁盘上读 取该对象数据即可，所花费时间也就读一个文件的时间，可想而知是多么的快，节省了我们的开发时间。

3. Google Protocol Buffers (protobuf)

Google Protocol Buffers (GPB)是 Google 内部使用的数据编码方式，旨在用来代替 XML 进行数据交换。可用于数据序列化与反序列化。主要特性有：

>优点

1. 性能好，效率高
2. 代码生成机制，数据解析类自动生成
3. 支持向后兼容和向前兼容
4. 支持多种编程语言（java，c++，python）

>缺点

1. 应用不够广
2. 二进制格式导致可读性差（二进制格式）
3. 缺乏自描述

细节我们不再关注，详见参考链接。

MapBuilderInterface 这个抽象接口由 MapBuilder 继承并实现了其方法，MapBuilder 定义在 /mapping/map_builder.h 中。从 MapBuilder 的注释我们可以看出，MapBuilder 是 cartographer 算法的最顶层设计，MapBuilder 包括了两个部分，其中 TrajectoryBuilder 用于 Local Submap 的建立与维护；PoseGraph 部分用于 Loop Closure.

所以，我们在解读玩 MapBuilder 这个类之后，就会先后跳到 TrajectoryBuilder 和 PoseGraph 来这两个类。而 cartographer 算法中的 Local Slam 和 Global Slam 也应该是分别在这两个类当中实现的。

接下来，让我们首先来看看 MapBuilder 对 MapBuilderInterface 的具体实现：

6.2、MapBuilder 的定义与实现：

MapBuilder 是对 MapBuilderInterface 的继承和实现，MapBuilder 中的方法都已经在 MapBuilderInterface 中定义，所以这里我们只看一下他的私有成员变量：

```
private:

  const proto::MapBuilderOptions options_;//MapBuilder 的配置项

  common::ThreadPool thread_pool_;//线程池。个人猜测，应该是为每一条 trajectory 都单独开辟一个线程

  std::unique_ptr<PoseGraph> pose_graph_; 一个 PoseGraph 的智能指针
```

```
std::unique_ptr<sensor::CollatorInterface> sensor_collator_;//收集传感器数据的智能指针

std::vector<std::unique_ptr<mapping::TrajectoryBuilderInterface>> trajectory_builders_;

//一个向量，管理所有的 TrajectoryBuilderInterface;应该是每一个 trajectory 对应了该向量的一个元素

std::vector<proto::TrajectoryBuilderOptionsWithSensorIds> all_trajectory_builder_options_;

//与每个 TrajectoryBuilderInterface 相对应的配置项
```

所以，MapBuilder 维护了一个 PoseGraph 的智能指针，该指针用来做 Loop Closure。此外，MapBuilder 还维护了一个 TrajectoryBuilder 的向量列表，每一个 TrajectoryBuilder 对应了机器人运行了一圈。这个向量列表就管理了整个图中的所有 submap。对于其他的一些配置文件，我们暂时不关心。

trajectory 是机器人跑一圈时的轨迹，在这其中需要记录和维护传感器的数据。根据这个 trajectory 上传感器收集的数据，我们可以逐步构建出栅格化的地图 Submap，但这个 submap 会随着时间或 trajectory 的增长而产生误差累积，但 trajectory 增长到超过一个阈值，则会新增一个 submap。而 PoseGraph 是用来进行全局优化，将所有的 Submap 紧紧 tie 在一起，构成一个全局的 Map，消除误差累积。

在维护上述几个智能指针和列表的情况下，部分函数的重载都只有一句话

```
// 返回一个 PoseGraphInterface 的接口指针。

mapping::PoseGraphInterface *pose_graph() override {

  return pose_graph_.get();//unique_ptr 的 get 函数可返回被管理对象的指针

}

// 返回系统中当前已有的 trajectory_builder 的数量

int num_trajectory_builders() const override {

  return trajectory_builders_.size();//向量的 size 即为 TrajectoryBuilder 的数量

}

// 根据 trajectory_id 返回一个 TrajectoryBuilderInterface 的指针。
```

```
    // 如果该 trajectory 没有一个 TrajectoryBuilder，则返回 nullptr’ corresponding to the specified

    mapping::TrajectoryBuilderInterface *GetTrajectoryBuilder(

        int trajectory_id) const override {

            return trajectory_builders_.at(trajectory_id).get();//从列表中取出指定 id 的 TrajectoryBuilder

        }

        // 获取所有 TrajectoryBuilder 的配置项。

        const std::vector<proto::TrajectoryBuilderOptionsWithSensorIds>

        &GetAllTrajectoryBuilderOptions() const override {

        return all_trajectory_builder_options_;//所有配置项都存在该向量中

    }
```

可以想见，MapBuilder 中的其他方法都是对 pose_graph_和向量列表 trajectory_builders_的管理和维护。

接下来，让我们看一下 MapBuilder 的构造函数和其他一些方法的实现：

2.1 构造函数

```
MapBuilder::MapBuilder(const proto::MapBuilderOptions& options)

    : options_(options), thread_pool_(options.num_background_threads()) {

  CHECK(options.use_trajectory_builder_2d() ^ options.use_trajectory_builder_3d());

  if (options.use_trajectory_builder_2d()) {

    pose_graph_ = common::make_unique<PoseGraph2D>(

        options_.pose_graph_options(),

        common::make_unique<optimization::OptimizationProblem2D>(
```

```
      options_.pose_graph_options().optimization_problem_options()),&thread_pool_);}

  if (options.use_trajectory_builder_3d()) {

    pose_graph_ = common::make_unique<PoseGraph3D>(

        options_.pose_graph_options(),

        common::make_unique<optimization::OptimizationProblem3D>(

        options_.pose_graph_options().optimization_problem_options()),&thread_pool_);

  }

  if (options.collate_by_trajectory()) {//根据 collate_by_trajectory 的不同,
          CollatorInterface 有两种不同的实现

     sensor_collator_ = common::make_unique<sensor::TrajectoryCollator>();

  } else {

    sensor_collator_ = common::make_unique<sensor::Collator>();

  }

}
```

构造函数基本上要做的就是给 MapBuilder 的配置项赋值的一些操作，比如，根据传入的参数 option 中的配置要设置是 2d 建图还是 3d 建图，2d 和 3d 建图要分别设置不同的 PoseGraph。同时，应该能够猜到 PoseGraph 这个接口应该有种不同的实现，分别是 PoseGraph2D 和 PoseGraph3D. 对于 PoseGraph 这块儿我还不知道为啥这么设计，但是作者是先定义了一个 PoseGraphInterface (/mapping/pose_graph_interface.h 中)，然后又定义了 PoseGraph (/mapping/pose_graph.h)来继承 PoseGraphInterface，但是 PoseGraph 里依然定义了很多虚函数，分别针对 2D 和 3D 的情况，由 PoseGraph2D (/mapping/internal/2d/pose_graph_2d.h)和 PoseGraph3D (/mapping/internal/3d/pose_graph_3d.h)来继承并实现。

另外需要提醒读者特别注意的就是对于 sensor_collator 的设置。sensor_collator_是一个接口 sensor::CollatorInterface 的智能指针。可以看到，根据 options.collate_by_trajectory()的不同，sensor::CollatorInterface 也有两种不同的实现方式，分别是 sensor::TrajectoryCollator 和 sensor::Collator。之前我们尝试从 TrajectoryBuilder 入手的时候感觉特别混乱就是没有搞清楚不同接口他们之间的实现关系。sensor::CollatorInterface 定义在/sensor/collator_interface.h 中；sensor::TrajectoryCollator 定义在/sensor/internal/collator.h 中；sensor::Collator 定义在 /sensor/internal/collator.h 中。

接下来我们看一下其他几个函数的实现：

2.2 MapBuilder::AddTrajectoryBuilder

//创建一个新的 TrajectoryBuilder 并返回它的 trajectory_id.

```
int MapBuilder::AddTrajectoryBuilder(
    const std::set<SensorId>& expected_sensor_ids,
    const proto::TrajectoryBuilderOptions& trajectory_options,
    LocalSlamResultCallback local_slam_result_callback){
    //生成 trajectory_id，trajectory_builders_是一个向量，存着所有的 trajectory。因此，当有一个新的 trajectory 时，以向量的 size 值作为新的 trajectory，加入到 trajectory_builders_中
  const int trajectory_id = trajectory_builders_.size();
  /* 可以看到，根据是 2d 建图还是 3d 建图分为了两种情况。
  * 针对两种不同的情况，首先建立一个 LocalTrajectoryBuilder2D 或 LocalTrajectoryBuilder3D 的变量 local_trajectory_builder;
  * 这个类是不带 Loop Closure 的 Local Slam, 包含了 Pose Extrapolator, Scan Matching 等;
  * 但注意，这两个类并没有继承 TrajectoryBuilder,并不是一个 TrajectoryBuilder 的实现，而只是一个工具类
  * 真正地创建一个 TrajectoryBuilder 是在后面，trajectory_builders_的 push_back 函数里面。
  * 其中 CollatedTrajectoryBuilder 继承了接口 TrajectoryBuilder;而前面生成的 local_trajectory_builder
  * 则用于 CreateGlobalTrajectoryBuilder2D 函数的第一个参数，用于生成一个 CollatedTrajectoryBuilder 的智能指针
  * CreateGlobalTrajectoryBuilder2D 函数定义在/mapping/internal/global_trajectory_builder.h 中。 */
  if (options_.use_trajectory_builder_3d()) {//根据配置参数选择是 3d 情况还是 2d 情况
    std::unique_ptr<LocalTrajectoryBuilder3D> local_trajectory_builder;
```

```
    //LocalTrajectoryBuilder3D 定义在/mapping/internal/3d/local_trajectory_builder_3d.h 中

    if (trajectory_options.has_trajectory_builder_3d_options()) {//是否有跟该 trajectory 相关的参数，如果有就设置一下

      //根据参数实例化这个 LocalTrajectoryBuilder3D 的变量

      local_trajectory_builder = common::make_unique<LocalTrajectoryBuilder3D>(

          trajectory_options.trajectory_builder_3d_options(),

          SelectRangeSensorIds(expected_sensor_ids));//生成一个 local_trajectory_builder

    }

    // 检查类型转化

    DCHECK(dynamic_cast<PoseGraph3D*>(pose_graph_.get()));//dynamic_cast 是强制类型转化，把 PoseGraphInterface 的指针转化为 PoseGraph3D

    //将生成的 CollatedTrajectoryBuilder 压入向量列表中

    trajectory_builders_.push_back(//真正地创建一个 TrajectoryBuilder，其中
        CollatedTrajectoryBuilder 继承了接口 TrajectoryBuilder

        common::make_unique<CollatedTrajectoryBuilder>(

            sensor_collator_.get(), trajectory_id, expected_sensor_ids,

            CreateGlobalTrajectoryBuilder3D(

                std::move(local_trajectory_builder), trajectory_id,

                static_cast<PoseGraph3D*>(pose_graph_.get()),

                local_slam_result_callback)));

  } else {//如果是 2d 的情况，跟 3d 情况基本相同，不再赘述

    std::unique_ptr<LocalTrajectoryBuilder2D> local_trajectory_builder;

    if (trajectory_options.has_trajectory_builder_2d_options()) {
```

```
  local_trajectory_builder = common::make_unique<LocalTrajectoryBuilder2D>(
      trajectory_options.trajectory_builder_2d_options(),
      SelectRangeSensorIds(expected_sensor_ids));
}
DCHECK(dynamic_cast<PoseGraph2D*>(pose_graph_.get()));
trajectory_builders_.push_back(
    common::make_unique<CollatedTrajectoryBuilder>(
        sensor_collator_.get(), trajectory_id, expected_sensor_ids,
        CreateGlobalTrajectoryBuilder2D(
            std::move(local_trajectory_builder), trajectory_id,
            static_cast<PoseGraph2D*>(pose_graph_.get()),
            local_slam_result_callback)));
//2d 的情况需要多处理的情况。
//has_overlapping_submaps_trimmer_2d()不知道表示什么意思
//猜测是说如果两个 submap 有重叠的部分，则调用 pose_graph_中的方法来进行全局优化
//那 3d 的情况就不需要处理吗？
if (trajectory_options.has_overlapping_submaps_trimmer_2d()) {
  const auto& trimmer_options = trajectory_options.overlapping_submaps_trimmer_2d();
  pose_graph_->AddTrimmer(common::make_unique<OverlappingSubmapsTrimmer2D>(
      trimmer_options.fresh_submaps_count(),
      trimmer_options.min_covered_area() /
          common::Pow2(trajectory_options.trajectory_builder_2d_options()
```

```
                        .submaps_options()
                        .grid_options_2d()
                        .resolution()),
        trimmer_options.min_added_submaps_count()));
    }
  }

  // 如果是纯定位的情况。pure_localization()的配置文件在
  /src/cartographer/configuration_files/trajectory_builder.lua 中
  if (trajectory_options.pure_localization()) {
    constexpr int kSubmapsToKeep = 3;
    pose_graph_->AddTrimmer(common::make_unique<PureLocalizationTrimmer>(trajectory_id,
    kSubmapsToKeep));
  }

  // 如果该轨迹有初始 pose；开始一条轨迹前我们是否已知初始位姿。
  // 这对应的情况就是比如说，我们检测到了一个 Landmark。那么这时，我们可以新增加一条
trajectory，
  // 增加新的 trajectory 时设置 has.initial_trajectory_pose 为真，
  // 然后根据机器人与 Landmark 之间的相对位姿推算机器人相对于世界坐标系的相对位姿。
  // 以该位姿作为新增加的 trajectory 的初始位姿。这样情况下，在检测到 Landmark 时就能有效降低
累积误差。
  if (trajectory_options.has_initial_trajectory_pose()) {
    const auto& initial_trajectory_pose = trajectory_options.initial_trajectory_pose();
    //调用 pose_graph_中的方法，设置初始 pose。跟全局相关的事情，都交给 pose_graph_来处理。
    pose_graph_->SetInitialTrajectoryPose(//设置初始 pose
```

```
        trajectory_id,
        initial_trajectory_pose.to_trajectory_id(),
        transform::ToRigid3(initial_trajectory_pose.relative_pose()),
        common::FromUniversal(initial_trajectory_pose.timestamp()));
  }
  //将一些跟传感器相关的配置项转成 proto 流，然后统一放到 all_trajectory_builder_options_这个向量列表中
  proto::TrajectoryBuilderOptionsWithSensorIds options_with_sensor_ids_proto;
  for (const auto& sensor_id : expected_sensor_ids) {
    *options_with_sensor_ids_proto.add_sensor_id() = ToProto(sensor_id);
  }
  *options_with_sensor_ids_proto.mutable_trajectory_builder_options() = trajectory_options;
  all_trajectory_builder_options_.push_back(options_with_sensor_ids_proto);
  CHECK_EQ(trajectory_builders_.size(), all_trajectory_builder_options_.size());
  return trajectory_id;
}
```

MapBuilder::AddTrajectoryBuilder 这个函数可以说是 cartographer 中最重要的一个函数，对这个函数的解读也耗费了我比较多的时间。幸好，对这个函数的解读也还算顺利。当然，这个函数包含的信息量还是非常大的。具体请见我的注释。这里再进行一下简单总结：

首先，用于 Local Slam 的 TrajectoryBuilderInterface 相关的各个类、接口等的相互继承关系的梳理：

可以看到，根据是 2d 建图还是 3d 建图分为了两种情况。 针对两种不同的情况，首先建立一个 LocalTrajectoryBuilder2D 或 LocalTrajectoryBuilder3D 的变量 local_trajectory_builder；这个类是不带 Loop Closure 的 Local Slam, 包含了 Pose Extrapolator, Scan Matching 等；

* 但注意，这两个类并没有继承 TrajectoryBuilder,并不是一个 TrajectoryBuilder 的实现，而只是一个工具类

真正地创建一个 TrajectoryBuilder 是在后面，trajectory_builders_的 push_back 函数里面。 其中 CollatedTrajectoryBuilder 继承了接口 TrajectoryBuilder;而前面生成的 local_trajectory_builder 则用于 CreateGlobalTrajectoryBuilder2D 函数的第一个参数，用于生成一个 CollatedTrajectoryBuilder 的智能指针。 CreateGlobalTrajectoryBuilder2D 函数定义在 /mapping/internal/global_trajectory_builder.h 中。

一个 MapBuilder 的类对应了一次建图过程，在整个建图过程中，用于全局优化的 PoseGraph 的对象只有一个，即 pose_graph_，而这个变量是在构造函数中就生成了。在 AddTrajectorybuilder 函数中只需要检查一下 pose_graph_是否符合 PoseGraph2D 或 PoseGraph3D 的情况。而一个 trajectory 对应了机器人运行一圈。在图建好后机器人可能多次运行。每一次运行都是新增一条 trajectory，因此，需要动态地维护一个 trajectory 的列表。每生成一个 trajectory 时都是调用 AddTrajectoryBuilder 来创建的。

其次，对于用于 Global Slam 的 PoseGraph 之间的关系梳理：

对于 PoseGraph 这块儿我还不知道为啥这么设计，但是作者是先定义了一个 PoseGraphInterface (/mapping/pose_graph_interface.h 中)，然后又定义了 PoseGraph (/mapping/pose_graph.h)来继承 PoseGraphInterface，但是 PoseGraph 里依然定义了很多虚函数，分别针对 2D 和 3D 的情况，由 PoseGraph2D (/mapping/internal/2d/pose_graph_2d.h)和 PoseGraph3D (/mapping/internal/3d/pose_graph_3d.h)来继承并实现。

第三，比较有用的关于 Landmark 的信息

初看，这里似乎并没有包含跟 Landmark 相关的信息。但是给算法中添加 Landmark 提供了便利。那就是其中 trajectory_options.has_initial_trajectory_pose()参数的设置，所以后面我们需要关注一下这些参数如何设置。

该参数对应的含义是说如果要添加的轨迹有初始 pose 该如何处理——即，该轨迹及其建立起来的子图在全局中的变换矩阵是有初始值的。 这对应的情况就是比如说，我们检测到了一个 Landmark。那么这时，我们可以新增加一条 trajectory，增加新的 trajectory 时设置 has.initial_trajectory_pose 为真，然后根据机器人与 Landmark 之间的相对位姿推算机器人相对于世界坐标系的相对位姿。 以该位姿作为新增加的 trajectory 的初始位姿。这样情况下，在检测到 Landmark 时就能有效降低累积误差。

最后，是关于另一个参数 pure_localization()

前面的一篇文章中有朋友留言，说“如果用 cartographer 定位，如何修改”。当时我还不知道该怎么修改，看到这里我觉得可能 cartographer 还是保留了纯定位的功能的。

pure_localization()的配置文件在 /src/cartographer/configuration_files/trajectory_builder.lua 中定义，可以看到，默认情况下，该值为 false。

第 7 章 MapBuilder 的解读 2

接下来的几个函数比较简单，我们一起查看：

7.1、AddTrajectoryForDeserialization、FinishTrajectory、SubmapToProto 和 SerializeState

```
//从序列化的数据中构造一条 trajectory
int MapBuilder::AddTrajectoryForDeserialization(
    const proto::TrajectoryBuilderOptionsWithSensorIds&
        options_with_sensor_ids_proto) {
  const int trajectory_id = trajectory_builders_.size();//id
  trajectory_builders_.emplace_back();//emplace_back 和 push_back 都是向容器内添加数据.
  //对于在容器中添加类的对象时，相比于 push_back,emplace_back 可以避免额外类的复制和移动操作.
  //但是，这里 emplace_back 里怎么没有参数呢？那把什么加进去了呢
  //配置项的添加
  all_trajectory_builder_options_.push_back(options_with_sensor_ids_proto);
  //检查两者大小是否一致
  CHECK_EQ(trajectory_builders_.size(), all_trajectory_builder_options_.size());
  return trajectory_id;
}
//结束一条轨迹；可以看到，分别调用 sensor_collator 和 pose_graph_的成员函数来 Finish 一条轨迹
void MapBuilder::FinishTrajectory(const int trajectory_id) {
```

```
  //现阶段猜测，sensor_collator_->FinishTrajectory 应该是做的清楚对传感器的占用等操作
  sensor_collator_->FinishTrajectory(trajectory_id);
  //现阶段猜测，pose_graph_->FinishTrajectory 应该是完成对刚刚的 trajectory 的全局优化
  pose_graph_->FinishTrajectory(trajectory_id);
}
// 根据指定的 submap_id 来查询 submap，把结果放到 SubmapQuery::Response 中。
// 如果出现错误，返回 error string; 成功则返回 empty string.
std::string MapBuilder::SubmapToProto(
    const SubmapId& submap_id, proto::SubmapQuery::Response* const response) {
  if (submap_id.trajectory_id < 0 ||
      submap_id.trajectory_id >= num_trajectory_builders()) {
    return "Requested submap from trajectory " +
        std::to_string(submap_id.trajectory_id) + " but there are only " +
        std::to_string(num_trajectory_builders()) + " trajectories.";
  }//指定的 submap 的 id 必须合法
  //pose_graph_中应该是维护着一张 submap 的列表。通过 pose_graph_获取指定 id 的子图
  const auto submap_data = pose_graph_->GetSubmapData(submap_id);
  if (submap_data.submap == nullptr) {
    return "Requested submap " + std::to_string(submap_id.submap_index) +
    " from trajectory " + std::to_string(submap_id.trajectory_id) +
    " but it does not exist: maybe it has been trimmed.";
  }//一些子图可能在优化过程中被裁掉。
```

```
    //正常情况下返回数据
    submap_data.submap->ToResponseProto(submap_data.pose, response);
    return "";
}
//调用 io::WritePbStream 工具，保存所有数据
void MapBuilder::SerializeState(io::ProtoStreamWriterInterface* const writer) {
    io::WritePbStream(*pose_graph_, all_trajectory_builder_options_, writer);
}
```

7.2、LoadState

最后一个函数比较长，但是不是特别重要，主要功能就是从 proto 流中构造出当前的状态。我们粗略看一下

```
// 从一个 proto 流中加载 SLAM 状态
void MapBuilder::LoadState(io::ProtoStreamReaderInterface* const reader,
                           bool load_frozen_state) {
  //反序列化读取工具
  io::ProtoStreamDeserializer deserializer(reader);
  // Create a copy of the pose_graph_proto, such that we can re-write the
  // trajectory ids.
  proto::PoseGraph pose_graph_proto = deserializer.pose_graph();
  const auto& all_builder_options_proto = deserializer.all_trajectory_builder_options();
  //逐条 trajectory 恢复
  std::map<int, int> trajectory_remapping;
```

```
for (auto& trajectory_proto : *pose_graph_proto.mutable_trajectory()) {
  const auto& options_with_sensor_ids_proto =
      all_builder_options_proto.options_with_sensor_ids(
          trajectory_proto.trajectory_id());
  const int new_trajectory_id =
      AddTrajectoryForDeserialization(options_with_sensor_ids_proto);
  CHECK(trajectory_remapping.emplace(trajectory_proto.trajectory_id(),
      new_trajectory_id).second)
        << "Duplicate trajectory ID: " << trajectory_proto.trajectory_id();
  trajectory_proto.set_trajectory_id(new_trajectory_id);
  if (load_frozen_state) {
    pose_graph_->FreezeTrajectory(new_trajectory_id);
  }
}
// Apply the calculated remapping to constraints in the pose graph proto.
//恢复 trajectory 上的节点间的约束关系
for (auto& constraint_proto : *pose_graph_proto.mutable_constraint()) {
  constraint_proto.mutable_submap_id()->set_trajectory_id(
      trajectory_remapping.at(constraint_proto.submap_id().trajectory_id()));
  constraint_proto.mutable_node_id()->set_trajectory_id(
      trajectory_remapping.at(constraint_proto.node_id().trajectory_id()));
}
```

```
//恢复 Submap
MapById<SubmapId, transform::Rigid3d> submap_poses;
for (const proto::Trajectory& trajectory_proto :
   pose_graph_proto.trajectory()) {
for (const proto::Trajectory::Submap& submap_proto :trajectory_proto.submap()) {
    submap_poses.Insert(SubmapId{trajectory_proto.trajectory_id(),
                                 submap_proto.submap_index()},
                        transform::ToRigid3(submap_proto.pose()));
  }
}
//恢复节点的 pose
MapById<NodeId, transform::Rigid3d> node_poses;
for (const proto::Trajectory& trajectory_proto :
     pose_graph_proto.trajectory()) {
  for (const proto::Trajectory::Node& node_proto : trajectory_proto.node()) {
    node_poses.Insert(
        NodeId{trajectory_proto.trajectory_id(), node_proto.node_index()},
        transform::ToRigid3(node_proto.pose()));
  }
}
// Set global poses of landmarks.
// 设置 Landmark
```

```
for (const auto& landmark : pose_graph_proto.landmark_poses()) {
  pose_graph_->SetLandmarkPose(landmark.landmark_id(),
                               transform::ToRigid3(landmark.global_pose()));
}
//不停读取，直到读完
SerializedData proto;
while (deserializer.ReadNextSerializedData(&proto)) {
  switch (proto.data_case()) {
    case SerializedData::kPoseGraph:
      LOG(ERROR) << "Found multiple serialized `PoseGraph`. Serialized "
                    "stream likely corrupt!.";
      break;
    case SerializedData::kAllTrajectoryBuilderOptions:
      LOG(ERROR) << "Found multiple serialized ""`AllTrajectoryBuilderOptions`. Serialized
                    stream likely ""corrupt!.";
      break;
    case SerializedData::kSubmap: {
        proto.mutable_submap()->mutable_submap_id()->set_trajectory_id(
            trajectory_remapping.at(
                proto.submap().submap_id().trajectory_id()));
      const transform::Rigid3d& submap_pose = submap_poses.at(
          SubmapId{proto.submap().submap_id().trajectory_id(),
```

```
                proto.submap().submap_id().submap_index()});
  pose_graph_->AddSubmapFromProto(submap_pose, proto.submap());
  break;
}
case SerializedData::kNode: {
  proto.mutable_node()->mutable_node_id()->set_trajectory_id(
      trajectory_remapping.at(proto.node().node_id().trajectory_id()));
  const transform::Rigid3d& node_pose =
      node_poses.at(NodeId{proto.node().node_id().trajectory_id(),
                           proto.node().node_id().node_index()});
  pose_graph_->AddNodeFromProto(node_pose, proto.node());
  break;
}
case SerializedData::kTrajectoryData: {
  proto.mutable_trajectory_data()->set_trajectory_id(
      trajectory_remapping.at(proto.trajectory_data().trajectory_id()));
  pose_graph_->SetTrajectoryDataFromProto(proto.trajectory_data());
  break;
}
case SerializedData::kImuData: {
  if (load_frozen_state) break;
  pose_graph_->AddImuData(
```

```
        trajectory_remapping.at(proto.imu_data().trajectory_id()),
        sensor::FromProto(proto.imu_data().imu_data()));
    break;
}
case SerializedData::kOdometryData: {
    if (load_frozen_state) break;
    pose_graph_->AddOdometryData(
        trajectory_remapping.at(proto.odometry_data().trajectory_id()),
        sensor::FromProto(proto.odometry_data().odometry_data()));
    break;
}
case SerializedData::kFixedFramePoseData: {
    if (load_frozen_state) break;
    pose_graph_->AddFixedFramePoseData(
        trajectory_remapping.at(
            proto.fixed_frame_pose_data().trajectory_id()),
            sensor::FromProto(
                proto.fixed_frame_pose_data().fixed_frame_pose_data()));
    break;
}
case SerializedData::kLandmarkData: {
  if (load_frozen_state) break;
```

```
        pose_graph_->AddLandmarkData(
            trajectory_remapping.at(proto.landmark_data().trajectory_id()),
            sensor::FromProto(proto.landmark_data().landmark_data()));
        break;
      }
      default:
          LOG(WARNING) << "Skipping unknown message type in stream: << proto.GetTypeName();
    }
}
if (load_frozen_state) {
    // Add information about which nodes belong to which submap.
    // Required for 3D pure localization.
    for(const proto::PoseGraph::Constraint& constraint_proto :
          pose_graph_proto.constraint()) {
          if (constraint_proto.tag() !=
          proto::PoseGraph::Constraint::INTRA_SUBMAP) {
          continue;
      }
      pose_graph_->AddNodeToSubmap(
        NodeId{constraint_proto.node_id().trajectory_id(),
        constraint_proto.node_id().node_index()},
        SubmapId{constraint_proto.submap_id().trajectory_id(),
```

```
        constraint_proto.submap_id().submap_index()});
    }
    } else {
    // When loading unfrozen trajectories, 'AddSerializedConstraints' will
    // take care of adding information about which nodes belong to which
    // submap.
    pose_graph_->AddSerializedConstraints(
        FromProto(pose_graph_proto.constraint()));
    }
    CHECK(reader->eof());
}
```

接下来，我们需要分成两个支线去看代码，一个是 Local SLAM 相关的 TrajectoryBuilder，另一个是用于 Loop Closure 的 PoseGraph 部分。

下一篇文章我们首先从用于 Local SLAM 的 TrajectoryBuilder 入手。

第 8 章 用于 Local SLAM 的 TrajectoryBuilder 部分

从这篇文章开始，我们进行 Local SLAM 部分的解读。之前时候我尝试跳过 MapBuilder 部分，直接进入 TrajectoryBuilder 部分进行阅读，但后来在 TrajectoryBuilder 的代码中被搞的晕头转向。所以后来我又返回去先梳理了一下 MapBuilder 中的代码。然后，带着新的理解，让我们再次开始 TrajectoryBuilder 部分代码的解读。

首先，我们来做一个整体上的理解，TrajectoryBuilder 是用来创建一个 trajectory 的。所以，该类首先要保存 trajectory 上的传感器数据，从一个连续的多帧传感器数据中会抽取若干关键帧。一帧关键帧数据被称为 trajectory 上的一个节点(注意与 ROS 中的节点概念相区分)，一条 trajectory 由一串儿节点组成，所以 TrajectoryBuilder 要维护一个节点列表，同时，每一帧时的传感器数据相对于该 Submap 的局部坐标变换要已知；其次，该类要创建并维护一个栅格化的 Submap 列表，以便在 MapBuilder 中对所有的 submap 进行优化，形成一个总的 Map。每个 Submap 相对于世界坐标系的位姿要估计出来，这样 PoseGraph 才能依据不同的 submap 各自的位姿变换矩阵把他们 tie 在一起并做全局优化。

所以，在看这一部分代码时，我们心里要有这样一个整体的印象。

8.1、接口 TrajectoryBuilderInterface 的定义

TrajectoryBuilderInterface 定义在/mapping/trajectory_builder_interface.h 中，主要成员包括：

1.1 成员变量

>1) 一个结构体 InsertionResult:

```
struct InsertionResult {//InsertionResult 就是用来保存插入 Local Slam 的一个节点的数据结构
    //NodeId 是一个结构体，在/mapping/id.h 中定义，
    //包含两个部分：一个 int 型的 trajectory_id 和一个 int 型的 node_index.
    NodeId node_id;
    //TrajectoryNode 是一个结构体，在/mapping/trajectory_node.h 中定义，
    //TrajectoryNode::Data 包含了经过处理的一帧传感器数据
    std::shared_ptr<const TrajectoryNode::Data> constant_data;
    //已经建立起来的子图列表。
```

```
    std::vector<std::shared_ptr<const Submap>> insertion_submaps;//
};
```

1.1.1)-<1>NodeId

这个结构体应该是用来存储一帧激光数据插入到 submap 中的结果的。在插入时，会为该帧数据分配一个 NodeId, NodeId 定义在/mapping/id.h 中，由两部分组成：一个 int 型的 trajectory_id 和一个 int 型的 node_index.

```
// trajectory 上的一个节点 id 由两部分组成，一是 trajectory_id，另一个是一个非零的 trajectory 内部的 index
struct NodeId {
  int trajectory_id;//trajectory 的 id，
  int node_index;//trajectory 内部的节点 id
  // 重载等于、不等于、小于等运算符
  bool operator==(const NodeId& other) const {
  return std::forward_as_tuple(trajectory_id, node_index) ==
      std::forward_as_tuple(other.trajectory_id, other.node_index);
  }
  bool operator!=(const NodeId& other) const { return !operator==(other); }
  bool operator<(const NodeId& other) const {
    return std::forward_as_tuple(trajectory_id, node_index) <
        std::forward_as_tuple(other.trajectory_id, other.node_index);
  }
  //序列化 id。proto::NodeId 可见/mapping/proto/pose_graph.proto
  void ToProto(proto::NodeId* proto) const {
```

```
    proto->set_trajectory_id(trajectory_id);

    proto->set_node_index(node_index);

  }

};
```

同时，从/mapping/id.h 这个文件中我们还可以发现 SubmapId 的定义，与 NodeId 定义，Submap 的 id 也有一个 trajectory_id 和一个 submap_index 组成。

1.1.1)-<2>TrajectoryData

TrajectoryData 是存储这个节点时的状态，包括时间、传感器数据等信息。具体如下：

```
struct TrajectoryNode {

  struct Data {

      common::Time time;//当前帧的时间

      // 一个表示旋转矩阵的四元数。该旋转矩阵将非水平面的传感器数据投射到水平面上

      // 利用 IMU 的重力传感器可计算出该旋转矩阵

      Eigen::Quaterniond gravity_alignment;

      // 经过水平投射后的点云数据，可用于 2D 情况下做 Loop Closure.

      sensor::PointCloud filtered_gravity_aligned_point_cloud;

      // Used for loop closure in 3D.

      sensor::PointCloud high_resolution_point_cloud;//高分辨率点云

      sensor::PointCloud low_resolution_point_cloud;//低分辨率点云

      Eigen::VectorXf rotational_scan_matcher_histogram;//旋转匹配直方图；VectorXf 是一个长度可变的向量。

      //节点在 Local SLAM 中的 Pose

      transform::Rigid3d local_pose;
```

```
    };

    // 返回成员变量 constant_data 的时间

    common::Time time() const { return constant_data->time; }

    // This must be a shared_ptr. If the data is used for visualization while the

    // node is being trimmed, it must survive until all use finishes.

    // 共享指针，允许多个指针指向同一个对象 详见
    https://www.cnblogs.com/diysoul/p/5930361.html

    std::shared_ptr<const Data> constant_data;

     // The node pose in the global SLAM frame.

    //节点在世界坐标系下的位姿

    transform::Rigid3d global_pose;

};
```

InsertionResult 的第三个成员变量是一个存放 Submap 的列表。看到这里我有一个疑惑。我的理解里，一帧传感器数据通过解优化问题，应该插入且只插入一个 submap 中，而这里为什么要维护一个 Submap 的列表呢？难道说一帧数据可能会插入多个 submap 中？这里暂时存疑。
我目前的理解是：一帧传感器数据虽然只有一个位姿，但是该传感器获取到的是一团点云数据，这些点云可能分属不同的 submap。当 Laser 获取到一帧数据时需要对所有这些点处理并相应地更新不同的 submap 的值。所以这里需要维护一个 submap 的列表。这个列表应该也是现存的所有的 submap 的列表。注意这里是一个共享指针。

还有一种可能是 Submap 之前会彼此重叠，一帧传感器数据会分属不同的 submap。到底应该是哪种理解，我现在也不是很确定，我们在边看边思考吧
如果读者还有其他不同的理解，欢迎与我留言讨论。

1.1.1)-<3>Submap

Submap 定义在/mapping/submaps.h 中，这是一个接口类，根据 2d 和 3d 的情况不同，分别有 Submap2D (/mapping/2d/submap_2d.h)和 Submap3D (/mapping/3d/submap_3d.h)两种实现方式。

```
// 一个子图，首先要有一个 local_pose。这个 local_pose 可以看做是没有经过全局优化的该 submap 相对

// 于世界坐标系的位姿。
```

```
// 一张子图要不停地监视有多少 range data（这里的 range data 我理解的就是点云）被插入到这个
submap

// 中。当没有 range data 插入时，则设置成员变量 finished_为真，然后开始做 Loop Closure。

class Submap {

  public:

  //构造函数，只包括一个 local_pose

  Submap(const transform::Rigid3d& local_submap_pose)

    : local_pose_(local_submap_pose) {}

  virtual ~Submap() {}

  //序列化与反序列化

  // 以下三个是纯虚函数，需要继承它的类去实现它

  virtual void ToProto(proto::Submap* proto,bool include_probability_grid_data) const = 0;

  virtual void UpdateFromProto(const proto::Submap& proto) = 0;

  // Fills data into the 'response'.

  // 把 submap 的放入到 response 的 proto 流中。方便 service 中查询 submap 讯息。

  virtual void ToResponseProto(const transform::Rigid3d& global_submap_pose,

       proto::SubmapQuery::Response* response) const = 0;

  // 返回成员变量 local_pose_，即该 submap 的位姿

  transform::Rigid3d local_pose() const { return local_pose_; }

  // 返回成员变量 num_range_data_，即插入到该 Submap 中的 range data 的数量。

  int num_range_data() const { return num_range_data_; }

  // 设置成员变量 num_range_data_
```

```
  void set_num_range_data(const int num_range_data) {
    num_range_data_ = num_range_data;
  }
  // 查看布尔型成员变量 finished_，即该子图是否还需要更新
  bool finished() const { return finished_; }
  // 设置 finished_
  void set_finished(bool finished) { finished_ = finished; }
  private:
    //三个成员变量
    const transform::Rigid3d local_pose_;
    int num_range_data_ = 0;
    bool finished_ = false;
};
```

我们以 Submap2D 为例，查看一下 Submap 的具体实现。对于 Submap3D，读者有兴趣可自行阅读。

```
class Submap2D : public Submap {
public:
//构造函数；第一个参数是原点坐标，第二个参数是一个 Grid2D 变量，存储栅格化坐标和坐标上的概率值
//Grid2D 定义在/mapping/2d/grid_2d.h 中，继承了 GridInterface(/mapping/grid_interface.h)，
//Grid2D 又被 ProbabilityGrid 继承，定义在/mapping/2d/probability_grid.h 中
//基本数据都存储在 Grid2D 的成员变量 grid_中。
  Submap2D(const Eigen::Vector2f& origin, std::unique_ptr<Grid2D> grid);
```

```
  explicit Submap2D(const proto::Submap2D& proto);

  // 实现接口 Submap 中的三个成员函数

  void ToProto(proto::Submap* proto,bool include_probability_grid_data) const override;

  void UpdateFromProto(const proto::Submap& proto) override;

  void ToResponseProto(const transform::Rigid3d& global_submap_pose,

      proto::SubmapQuery::Response* response) const override;

      //返回 grid_

      const Grid2D* grid() const { return grid_.get(); }

      // 利用 RangeDataInserterInterface 来插入并更新概率图。该接口在
      /mapping/range_data_inserter_interface.h 中定义

      // ProbabilityGridRangeDataInserter2D 继承了该接口，定义在
      /mapping/2d/probability_grid_range_data_inserter_2d.h 中

    void InsertRangeData(const sensor::RangeData& range_data,

      const RangeDataInserterInterface* range_data_inserter);

    void Finish();

  private:

    std::unique_ptr<Grid2D> grid_;//概率图数据存储在这里。

};
```

查看/mapping/2d/http://submap_2d.cc 代码，我们发现一个参数构造的函数 CreateSubmapsOptions2D. 经过查找，我们可以找到相关的配置文件：/src/cartographer/configuration_files/trajectory_builder_2d.lua 中，我们可以根据不同的情况相应地修改配置文件中的参数。

```
//参数配置在/src/cartographer/configuration_files/trajectory_builder_2d.lua 中

proto::SubmapsOptions2D CreateSubmapsOptions2D(
```

```
    common::LuaParameterDictionary* const parameter_dictionary) {
  proto::SubmapsOptions2D options;
  options.set_num_range_data(
      parameter_dictionary->GetNonNegativeInt("num_range_data"));
  *options.mutable_grid_options_2d() = CreateGridOptions2D(
      parameter_dictionary->GetDictionary("grid_options_2d").get());
  *options.mutable_range_data_inserter_options() =
      CreateRangeDataInserterOptions(
          parameter_dictionary->GetDictionary("range_data_inserter").get());
```

构造函数：

```
// 可以看到，在构造 2D 子图时，Submap 的坐标系旋转角度设置为 0.而原点由参数 origin 给出
Submap2D::Submap2D(const Eigen::Vector2f& origin, std::unique_ptr<Grid2D> grid)
    : Submap(transform::Rigid3d::Translation(
          Eigen::Vector3d(origin.x(), origin.y(), 0.))) {
  grid_ = std::move(grid);
}
// 从 proto 流中构建 Submap2D
Submap2D::Submap2D(const proto::Submap2D& proto)
    : Submap(transform::ToRigid3(proto.local_pose())) {
  if (proto.has_grid()) {
    CHECK(proto.grid().has_probability_grid_2d());
    grid_ = common::make_unique<ProbabilityGrid>(proto.grid());
```

```
  }

  set_num_range_data(proto.num_range_data());

  set_finished(proto.finished());

}
```

与 proto 流相关的处理：

```
//序列化，存到 proto 中

void Submap2D::ToProto(proto::Submap* const proto,

                       bool include_probability_grid_data) const {

  auto* const submap_2d = proto->mutable_submap_2d();

  *submap_2d->mutable_local_pose() = transform::ToProto(local_pose());

  submap_2d->set_num_range_data(num_range_data());

  submap_2d->set_finished(finished());

  if (include_probability_grid_data) {

    CHECK(grid_);

    *submap_2d->mutable_grid() = grid_->ToProto();//调用 grid_中的 ToProto 函数把概率图保存到
proto 中

  }

}



//从 proto 流中获取 Submap2D

void Submap2D::UpdateFromProto(const proto::Submap& proto) {

  CHECK(proto.has_submap_2d());
```

```
  const auto& submap_2d = proto.submap_2d();
  set_num_range_data(submap_2d.num_range_data());
  set_finished(submap_2d.finished());
  if (proto.submap_2d().has_grid()) {
    CHECK(proto.submap_2d().grid().has_probability_grid_2d());
    grid_ = common::make_unique<ProbabilityGrid>(submap_2d.grid());
  }
}
//放到 response 中
void Submap2D::ToResponseProto(
    const transform::Rigid3d&,
    proto::SubmapQuery::Response* const response) const {
  if (!grid_) return;
  response->set_submap_version(num_range_data());
  proto::SubmapQuery::Response::SubmapTexture* const texture =
      response->add_textures();
  grid()->DrawToSubmapTexture(texture, local_pose());
}
```

插入 RangeData 数据和结束 Submap 的两个函数：

```
void Submap2D::InsertRangeData(
    const sensor::RangeData& range_data,
    const RangeDataInserterInterface* range_data_inserter) {
```

```
  CHECK(grid_);//检查是否栅格化
  CHECK(!finished());//检查图是否已被 finished
  //调用 RangeDataInserterInterface 来更新概率图。
  range_data_inserter->Insert(range_data, grid_.get());
  set_num_range_data(num_range_data() + 1);//插入的数据+1
}
void Submap2D::Finish() {
  CHECK(grid_);
   CHECK(!finished());
  //计算裁剪栅格图
  grid_ = grid_->ComputeCroppedGrid();
  set_finished(true);
}
```

接下来，我们先看看栅格化的地图 Grid2D，然后再查看一下插入和更新栅格地图的 RangeDataInserter.

1.1.1)-<3>-1 Grid2D

Grid2D 定义在/mapping/2d/grid_2d.h 中，Grid2D 继承了 GridInterface (/mapping/grid_interface.h)，在 GridInterface 只有一个空壳子：

```
class GridInterface {
    // todo(kdaun) move mutual functions of Grid2D/3D here
};
```

Grid2D 定义如下：

```
class Grid2D : public GridInterface {
```

```
public:

  // MapLimits 定义在/mapping/2d/map_limits.h 中；

  // MapLimits 包含三个成员变量：double resolution_;分辨率 程序中设置是 0.05m.也就是 5cm。

  //                          Eigen::Vector2d max_;这是一个浮点型二维向量，max_.x()和.y()分别
表示 x、y 方向的最大值

  //                          CellLimits cell_limits_;栅格化后的 x 和 y 方向的最大范围。

  //        CellLimits 定义在/mapping/2d/xy_index.h 中，包括两个 int 型成员变量

  //                          int num_x_cells;x 方向划分的栅格数，也是 pixel 坐标情况下的最大
范围

  //                          int num_y_cells;y 方向划分的栅格数，

  // 在 MapLimits 中根据最大范围 max_和分辨率 resolution 就可以创建 cell_limits

  // MapLimits 这个类也提供了如下函数：

  // MapLimits::resulution()、MapLimits::max()、MapLimits::cell_limits()

  // 分别获取分辨率、最大范围值、pixel 坐标的最大范围等。

  // MapLimits::GetcellIndex(const Eigen::Vector2f& point)

  // 给出一个 point 在 Submap 中的坐标，求其 pixel 坐标

  // MapLimits::Contains(const Eigen::Array2i& cell_index)

  // 返回布尔型，判断所给 pixel 坐标是否大于 0，小于等于最大值

  // min_correspondence_cost 和 max_correspondence_cost

  // 分别是一个 pixel 坐标是否不被 occupied 的概率的最小最大值

  // probability 表示一个 pixel 坐标被 occupied 的概率； probability + correspondence_cost = 1

  explicit Grid2D(const MapLimits& limits, float min_correspondence_cost,

                  float max_correspondence_cost);
```

```
explicit Grid2D(const proto::Grid2D& proto);

// Returns the limits of this Grid2D.

// 返回该栅格地图的范围、分辨率等。

const MapLimits& limits() const { return limits_; }

// Finishes the update sequence.

// 停止更新

void FinishUpdate();

// Returns the correspondence cost of the cell with 'cell_index'.

// 返回一个坐标的 CorrespondenceCost 值

float GetCorrespondenceCost(const Eigen::Array2i& cell_index) const;

// Returns the minimum possible correspondence cost.

// 获取最小值

float GetMinCorrespondenceCost() const { return min_correspondence_cost_; }

// Returns the maximum possible correspondence cost.

// 获取最大值

float GetMaxCorrespondenceCost() const { return max_correspondence_cost_; }

// Returns true if the probability at the specified index is known.

// 判断一个 pixel 是否已经有相应的概率值

bool IsKnown(const Eigen::Array2i& cell_index) const;

// Fills in 'offset' and 'limits' to define a subregion of that contains all

// known cells.

// 进行一下裁剪。裁剪一个 subregion，使得该 subregion 包含了所有的已有概率值的 cells
```

```
  void ComputeCroppedLimits(Eigen::Array2i* const offset,CellLimits* const limits) const;

  // Grows the map as necessary to include 'point'. This changes the meaning of

  // these coordinates going forward. This method must be called immediately

  // after 'FinishUpdate', before any calls to 'ApplyLookupTable'.

  // 必要时 grow 我们的 submap。这是一个虚函数

  virtual void GrowLimits(const Eigen::Vector2f& point);

  // 得到一个裁剪后的栅格图

  virtual std::unique_ptr<Grid2D> ComputeCroppedGrid() const = 0;

  // 写入 proto 流

  virtual proto::Grid2D ToProto() const;

  virtual bool DrawToSubmapTexture(

      proto::SubmapQuery::Response::SubmapTexture* const texture,

      transform::Rigid3d local_pose) const = 0;

protected:

  // 返回记录栅格地图概率值的向量

  const std::vector<uint16>& correspondence_cost_cells() const {

    return correspondence_cost_cells_;

  }

  // 更新索引

  const std::vector<int>& update_indices() const { return update_indices_; }

  // 返回一个已知概率值的区域。

  const Eigen::AlignedBox2i& known_cells_box() const {
```

```
    return known_cells_box_;
  }
  std::vector<uint16>* mutable_correspondence_cost_cells() {
    return &correspondence_cost_cells_;
  }
  std::vector<int>* mutable_update_indices() { return &update_indices_; }
  Eigen::AlignedBox2i* mutable_known_cells_box() { return &known_cells_box_; }
  // Converts a 'cell_index' into an index into 'cells_'.
  // 将 pixel 坐标再转化为局部坐标系中的点的坐标
  int ToFlatIndex(const Eigen::Array2i& cell_index) const;
private:
  MapLimits limits_;//地图范围
  std::vector<uint16> correspondence_cost_cells_;//存储概率值，这里的概率值是 Free 的概率值
  float min_correspondence_cost_;//最小值
  float max_correspondence_cost_;//最大值
  std::vector<int> update_indices_;//更新的索引
  // Bounding box of known cells to efficiently compute cropping limits.
  // 维护一个已知概率值的所有 cells 的盒子。方便计算裁剪范围
  Eigen::AlignedBox2i known_cells_box_;//存放 Submap 已经赋值过的一个子区域的盒子。
};
```

第 9 章 TrajectoryBuilder 部分 2-Submap 后续

我们可以看一下 Grid2D 中的一些函数实现：

```
Grid2D::Grid2D(const MapLimits& limits, float min_correspondence_cost,
               float max_correspondence_cost):
    limits_(limits),correspondence_cost_cells_(
        limits_.cell_limits().num_x_cells * limits_.cell_limits().num_y_cells,
        kUnknownCorrespondenceValue),//根据 cells 的大小生成一维向量。
    min_correspondence_cost_(min_correspondence_cost),
    max_correspondence_cost_(max_correspondence_cost) {
  CHECK_LT(min_correspondence_cost_, max_correspondence_cost_);//检查最小值是否小于最大值
}
//从 proto 流中恢复 Grid2D
Grid2D::Grid2D(const proto::Grid2D& proto)
    : limits_(proto.limits()), correspondence_cost_cells_() {
  if (proto.has_known_cells_box()) {
    const auto& box = proto.known_cells_box();
    known_cells_box_ =
        Eigen::AlignedBox2i(Eigen::Vector2i(box.min_x(), box.min_y()),
                            Eigen::Vector2i(box.max_x(), box.max_y()));
  }
  correspondence_cost_cells_.reserve(proto.cells_size());
```

```
  for (const auto& cell : proto.cells()) {
    CHECK_LE(cell, std::numeric_limits<uint16>::max());
    correspondence_cost_cells_.push_back(cell);
  }
  if (proto.min_correspondence_cost() == 0.f &&
      proto.max_correspondence_cost() == 0.f) {
    LOG(WARNING)
        << "proto::Grid2D: max_correspondence_cost and min_correspondence_cost "
           "are initialized with 0 indicating an older version of the "
           "protobuf format. Loading default values.";
    min_correspondence_cost_ = kMinCorrespondenceCost;
    max_correspondence_cost_ = kMaxCorrespondenceCost;
  } else {
    min_correspondence_cost_ = proto.min_correspondence_cost();
    max_correspondence_cost_ = proto.max_correspondence_cost();
  }
  CHECK_LT(min_correspondence_cost_, max_correspondence_cost_);
}
```

接下来看一下其他函数的实现，这部分比较简单，各函数功能含义也很明确。详见注释：

```
// Finishes the update sequence.
// 停止更新
void Grid2D::FinishUpdate() {
```

```
  while (!update_indices_.empty()) {//如果索引值列表不为空
    DCHECK_GE(correspondence_cost_cells_[update_indices_.back()],
              kUpdateMarker);//检查该索引值中的 Correspondence 是否大于等于 kUpdateMarker.
    //kUpdateMarker 等值都定义在/mapping/probability_values.h 中
    correspondence_cost_cells_[update_indices_.back()] -= kUpdateMarker;
    update_indices_.pop_back();
  }
}
// Returns the correspondence cost of the cell with 'cell_index'.
// 返回指定 pixel 坐标的 CorrespondenceCost 值
float Grid2D::GetCorrespondenceCost(const Eigen::Array2i& cell_index) const {
  if (!limits().Contains(cell_index)) return kMaxCorrespondenceCost;
  return ValueToCorrespondenceCost(//这些函数都定义在/mapping/probability_values.h 中
      correspondence_cost_cells()[ToFlatIndex(cell_index)]);//ToFlatIndex 将一个二维坐标转化为一维的索引值
}
// Returns true if the correspondence cost at the specified index is known.
// 判断一个指定坐标是否已经有概率值
bool Grid2D::IsKnown(const Eigen::Array2i& cell_index) const {
  return limits_.Contains(cell_index) &&
         correspondence_cost_cells_[ToFlatIndex(cell_index)] !=
             kUnknownCorrespondenceValue;
```

```
}

// Fills in 'offset' and 'limits' to define a subregion of that contains all
// known cells.
//圈出来一个子区域
void Grid2D::ComputeCroppedLimits(Eigen::Array2i* const offset,
                                  CellLimits* const limits) const {
  if (known_cells_box_.isEmpty()) {
    *offset = Eigen::Array2i::Zero();
    *limits = CellLimits(1, 1);
    return;
  }
  *offset = known_cells_box_.min().array();
  *limits = CellLimits(known_cells_box_.sizes().x() + 1,
                       known_cells_box_.sizes().y() + 1);
}

// Grows the map as necessary to include 'point'. This changes the meaning of
// these coordinates going forward. This method must be called immediately
// after 'FinishUpdate', before any calls to 'ApplyLookupTable'.
// 增长子图
void Grid2D::GrowLimits(const Eigen::Vector2f& point) {
  CHECK(update_indices_.empty());
  while (!limits_.Contains(limits_.GetCellIndex(point))) {
```

```
const int x_offset = limits_.cell_limits().num_x_cells / 2;
const int y_offset = limits_.cell_limits().num_y_cells / 2;
const MapLimits new_limits(
    limits_.resolution(),
    limits_.max() +
        limits_.resolution() * Eigen::Vector2d(y_offset, x_offset),
    CellLimits(2 * limits_.cell_limits().num_x_cells,
               2 * limits_.cell_limits().num_y_cells));
const int stride = new_limits.cell_limits().num_x_cells;
const int offset = x_offset + stride * y_offset;
const int new_size = new_limits.cell_limits().num_x_cells *
                     new_limits.cell_limits().num_y_cells;
std::vector<uint16> new_cells(new_size, kUnknownCorrespondenceValue);
for (int i = 0; i < limits_.cell_limits().num_y_cells; ++i) {
  for (int j = 0; j < limits_.cell_limits().num_x_cells; ++j) {
    new_cells[offset + j + i * stride] =
        correspondence_cost_cells_[j +
                                   i * limits_.cell_limits().num_x_cells];
  }
}
correspondence_cost_cells_ = new_cells;
limits_ = new_limits;
```

```
    if (!known_cells_box_.isEmpty()) {
      known_cells_box_.translate(Eigen::Vector2i(x_offset, y_offset));
    }
  }
}

proto::Grid2D Grid2D::ToProto() const {
  proto::Grid2D result;
  *result.mutable_limits() = mapping::ToProto(limits_);
  result.mutable_cells()->Reserve(correspondence_cost_cells_.size());
  for (const auto& cell : correspondence_cost_cells_) {
    result.mutable_cells()->Add(cell);
  }
  CHECK(update_indices().empty()) << "Serializing a grid during an update is "
                                     "not supported. Finish the update first.";
  if (!known_cells_box().isEmpty()) {
    auto* const box = result.mutable_known_cells_box();
    box->set_max_x(known_cells_box().max().x());
    box->set_max_y(known_cells_box().max().y());
    box->set_min_x(known_cells_box().min().x());
    box->set_min_y(known_cells_box().min().y());
  }
  result.set_min_correspondence_cost(min_correspondence_cost_);
```

```
  result.set_max_correspondence_cost(max_correspondence_cost_);

  return result;

}

// 将一个二维坐标转化为一维的索引值

int Grid2D::ToFlatIndex(const Eigen::Array2i& cell_index) const {

  CHECK(limits_.Contains(cell_index)) << cell_index;

  return limits_.cell_limits().num_x_cells * cell_index.y() + cell_index.x();

}
```

我们注意到，在 Grid2D 的定义中，有 2 个纯虚函数和 2 个虚函数：

```
virtual void GrowLimits(const Eigen::Vector2f& point);

// 得到一个裁剪后的栅格图

virtual std::unique_ptr<Grid2D> ComputeCroppedGrid() const = 0;

// 写入 proto 流

virtual proto::Grid2D ToProto() const;

virtual bool DrawToSubmapTexture(

  proto::SubmapQuery::Response::SubmapTexture* const texture,transform::Rigid3d local_pose)
const = 0;
```

而 ProbabilityGrid 继承了 Grid2D 并实现了上述两个函数，具体详见/mapping/2d/probability_grid.h 和.../http://probability_grid.cc 中。我们这里不再详细分析。此外，ProbabilityGrid 中还提供了三个函数：

SetProbability，可用来设置一个给定 pixel 坐标的概率值；

GetProbability， 得到一个指定坐标的概率值。

ApplyLookupTable，看注释没看明白这个函数的作用是什么，所以我们深入看一下这个函数：

```
// Applies the 'odds' specified when calling ComputeLookupTableToApplyOdds()
// to the probability of the cell at 'cell_index' if the cell has not already
// been updated. Multiple updates of the same cell will be ignored until
// FinishUpdate() is called. Returns true if the cell was updated.
//
// If this is the first call to ApplyOdds() for the specified cell, its value
// will be set to probability corresponding to 'odds'.
bool ProbabilityGrid::ApplyLookupTable(const Eigen::Array2i& cell_index,
                                       const std::vector<uint16>& table) {
  DCHECK_EQ(table.size(), kUpdateMarker);//检查 table 的 size 是否等于一个微小元
  const int flat_index = ToFlatIndex(cell_index);//把 pixel 坐标转化为一维索引值
  // mutable_correspondence_cost_cells()是 Grid2D 的成员函数，返回存放概率值的一维向量
  // 根据 cell 坐标，返回该 cell 中原本的 value 值
  uint16* cell = &(*mutable_correspondence_cost_cells())[flat_index];//根据索引值求该 cell 的值
  if (*cell >= kUpdateMarker) {
    return false;
  }
// 已更新的信息都存储在 update_indices_这个向量中，所以该 cell 被处理过后它的 index 要加入到这个向量中
  mutable_update_indices()->push_back(flat_index);
// 根据该 pixel 返回的值 cell 来查表，获取更新后应该是什么值。然后把这个值放入到 cell 原先的地址中。实际就是更新该值
```

```
  *cell = table[*cell];

  DCHECK_GE(*cell, kUpdateMarker);

  //mutable_known_cells_box()是 Grid2D 的成员函数，返回存放已知概率值的一个子区域的盒子。

  // 现在就是把该 cell 放入已知概率值的盒子中

  mutable_known_cells_box()->extend(cell_index.matrix());

  return true;

}
```

这个函数就是通过查表来更新 Grid 的值。详见 Cartographer 源码阅读之附 1—probability_values.h/c：占据概率相关 中对 ComputeLookupTableToApplyOdds 和 ComputeLookupTableToApplyCorrespondenceCostOdds 这两个函数的解读。根据传感器返回的数据是 Hit 还是 Miss 这两种情况，我们会对所有的 计算出两张表，hit_table 和 miss_table. 这样，假设在 Grid2D 这个栅格图中的其中一个 pixel 或 cell，已预先有一个 value 值，然后现在又有一个传感器的测量结果，hit 或 miss，那么我们不需要计算，只需要以 value 为索引，通过查 hit_table 或 miss_table 中的值就可以得到更新后的 value 应该是多少。所以这个函数中第一个参数就是一个点的坐标，第二个参数就是一张 table。

1.1.1)-<3>-2 InsertRangeData

接下来我们重要看看 Submap 图是如何更新的：

我们回顾一下上一篇文章中提到的在 Submap 里是通过 InsertRangeData 这个函数来插入一帧传感器数据的。在该函数里主要做了两件工作：一是根据传感器数据更新了地图，二是将该 submap 的 num_rang_data_增长了 1.

```
void Submap2D::InsertRangeData(

    const sensor::RangeData& range_data,

    const RangeDataInserterInterface* range_data_inserter){

  CHECK(grid_);//检查是否栅格化

  CHECK(!finished());//检查图是否已被 finished

  //调用 RangeDataInserterInterface 来更新概率图。
```

```
  range_data_inserter->Insert(range_data, grid_.get());

  set_num_range_data(num_range_data() + 1);//插入的数据+1

}
```

更新操作由 RangeDataInserterInterface 完成，该接口定义在 /mapping/range_data_inserter_interface.h 中。根据 2d 和 3d 情况的不同，分别由 ProbabilityGridRangeDataInserter2D (/mapping/2d/probability_grid_range_data_inserter_2d.h) 和 ProbabilityGridRangeDataInserter3D (/mapping/3d/probability_grid_range_data_inserter_3d.h)中。

RangeDataInserterInterface 的核心就是定义了一个虚函数：Insert(const sensor::RangeData& range_data, GridInterface* grid). 其主要功能就是向栅格化地图中插入传感器数据。

```
class RangeDataInserterInterface {

 public:

  // Inserts 'range_data' into 'grid'.

  virtual void Insert(const sensor::RangeData& range_data,

                      GridInterface* grid) const = 0;

};
```

另外一个函数 CreateRangeDataInserterOptions 的工作就是加载一下配置项，比如 range_data_inserter_type 等。这些配置定义在 /src/cartographer/configuration_files/trajectory_builder_2d.lua.

```
proto::RangeDataInserterOptions CreateRangeDataInserterOptions(

                    common::LuaParameterDictionary* const parameter_dictionary)

{

  proto::RangeDataInserterOptions options;

  const std::string range_data_inserter_type_string =

      parameter_dictionary->GetString("range_data_inserter_type");
```

```
  proto::RangeDataInserterOptions_RangeDataInserterType range_data_inserter_type;
  CHECK(proto::RangeDataInserterOptions_RangeDataInserterType_Parse(
      range_data_inserter_type_string, &range_data_inserter_type))
      << "Unknown RangeDataInserterOptions_RangeDataInserterType kind: "
      << range_data_inserter_type_string;
  options.set_range_data_inserter_type(range_data_inserter_type);
  *options.mutable_probability_grid_range_data_inserter_options_2d() =
      CreateProbabilityGridRangeDataInserterOptions2D(
          parameter_dictionary
              ->GetDictionary("probability_grid_range_data_inserter")
              .get());
  return options;
}
```

在 ProbabilityGridRangeDataInserter2D (/mapping/2d/probability_grid_range_data_inserter_2d.h) 中，具体实现了 Insert 这个函数。同时，该类的配置函数 CreateProbabilityGridRangeDataInserterOptions2D 设置了 hit_probability(0.55)和 miss_probability(0.49)的具体数值.

```
void ProbabilityGridRangeDataInserter2D::Insert(
    const sensor::RangeData& range_data, GridInterface* const grid) const {
  // 将 Grid 类型强制转化为 ProbabilityGrid 类型
  ProbabilityGrid* const probability_grid = static_cast<ProbabilityGrid*>(grid);
  // By not finishing the update after hits are inserted, we give hits priority
  // (i.e. no hits will be ignored because of a miss in the same cell).
```

```
  //调用 CastRays 函数更新 Grid
  CastRays(range_data, hit_table_, miss_table_, options_.insert_free_space(),
           CHECK_NOTNULL(probability_grid));
  probability_grid->FinishUpdate();
}
```

Insert 函数调用了 CastRays 函数来处理传感器数据。该函数定义在 /mapping/internal/2d/ray_casting.h 中。可以想见，CastRay 中就是把 RangeData 中包含的一系列点 ，计算出一条从原点到 的射线，射线端点处的点是 Hit，射线中间的点是 Free。把所有这些点要在地图上把相应的 cell 进行更新。

在开始阅读 CastRays 之前，我们先来看一下 RangeData 的定义，请见/sensor/range_data.h：

```
// Rays begin at 'origin'. 'returns' are the points where obstructions were
// detected. 'misses' are points in the direction of rays for which no return
// was detected, and were inserted at a configured distance. It is assumed that
// between the 'origin' and 'misses' is free space.
struct RangeData {
  Eigen::Vector3f origin;
  PointCloud returns;
  PointCloud misses;
};
```

一帧 RangeData 数据包含 3 个元素，其中 origin 是一个 3 维向量，表示的是这一帧数据的原点。returns 是那些检测到了 hits 的点，那么被 Hits 的点加入 hits 这个集合，相应地更改 submap 中的信息。而 origin 和 hits 中间的点也是 Free 的，需要归入到 misses 集合中。misses 是那些没有检测到 return 的点，原点和 misses 形成的射线上的所有点都是 free Space，都需要归入 misses 集合中。上述所有的点的表示里，如果是 2d 情况的话，第三个元素是 0。关于如何根据 Hit 和 Miss 更新概率图，请详见：Cartographer 源码阅读之附 1—probability_values.h/c：占据概率相关 。

接下来让我们看一下 CastRays 这个函数代码：

```
void CastRays(const sensor::RangeData& range_data,
              const std::vector<uint16>& hit_table,
              const std::vector<uint16>& miss_table,
              const bool insert_free_space,
              ProbabilityGrid* const probability_grid) {
  GrowAsNeeded(range_data, probability_grid);
  const MapLimits& limits = probability_grid->limits();//获取栅格地图的 limits
  // 定义一个超分辨率像素，把当前的分辨率又划分成了 kSubpixelScale 份，这里 int
kSubpixelScale = 1000
  const double superscaled_resolution = limits.resolution() / kSubpixelScale;
  // 根据超分辨率像素又生成了一个新的 MapLimits
  const MapLimits superscaled_limits(
      superscaled_resolution, limits.max(),
      CellLimits(limits.cell_limits().num_x_cells * kSubpixelScale,//划分格数变成了
kSubpixelScale 倍
                 limits.cell_limits().num_y_cells * kSubpixelScale));
  // 根据 RangeData 原点的前两项，获取其对应的栅格化坐标。该坐标是我们所求的射线的原点。
  const Eigen::Array2i begin =
      superscaled_limits.GetCellIndex(range_data.origin.head<2>());
  // Compute and add the end points.
  // 定义一个向量集合，该集合存储 RangeData 中的 hits 的点。
  std::vector<Eigen::Array2i> ends;
```

// reserver 函数用来给 vector 预分配存储区大小，即 capacity 的值 ，但是没有给这段内存进行初始化。

// reserve 的参数 n 是推荐预分配内存的大小，实际分配的可能等于或大于这个值，即 n 大于 capacity 的值，

// 就会 reallocate 内存 capacity 的值会大于或者等于 n 。这样，当 ector 调用 push_back 函数使得 size

// 超过原来的默认分配的 capacity 值时 避免了内存重分配开销。

// 这里就是根据 returns 集合的大小，给 ends 预分配一块存储区

ends.reserve(range_data.returns.size());

for (const Eigen::Vector3f& hit : range_data.returns) {

// 遍历 returns 这个集合，把每个点先压入 ends 中，

ends.push_back(superscaled_limits.GetCellIndex(hit.head<2>()));

// ens.back()返回的是 vector 中的最末尾项，也就是我们刚刚压入 vector 中的那一项；

// 这里我猜测，hit_table 就是预先计算好的。如果一个 cell，原先的值是 value，那么在检测到 hit 后应该更新为多少

probability_grid->ApplyLookupTable(ends.back() / kSubpixelScale, hit_table);

}

if (!insert_free_space) {

// 如果配置项里设置是不考虑 free space。那么函数到这里结束，只处理完 hit 后返回即可

// 否则的话，需要计算那条射线，射线中间的点都是 free space，同时，没有检测到 hit 的 misses 集里也都是 free

return;

}

// Now add the misses.

```
  // 处理 origin 跟 hit 之间的射线中间的点
  for (const Eigen::Array2i& end : ends) {
    CastRay(begin, end, miss_table, probability_grid);
  }
  // Finally, compute and add empty rays based on misses in the range data.
  // 同样处理 miss 集合中的点
  for (const Eigen::Vector3f& missing_echo : range_data.misses) {
    CastRay(begin, superscaled_limits.GetCellIndex(missing_echo.head<2>()),
            miss_table, probability_grid);
  }
}
```

程序的具体逻辑详见代码中的注释，相信大家不难看懂。需要注意的是，在 CastRays 中处理 RangeData 的数据时，又用到了一个超分辨率像素的概念，即将一个 cell 又进一步划分成 kSubpixelScale × kSubpixelScale 个 cell，其中，kSubpixelScale = 1000。这样，在计算 origin 到 end 之间的射线方程时可以做到更精确。

具体处理射线方程的工作交给了 CastRay 去做。我们这里就不再进一步深入去探讨了。感兴趣的读者可以自行查看。

1.1.1)-<3>-3 ActiveSubmaps2D

我们看到，在/mapping/2d/submap_2d.h 中，除了 Submap 外还定义了一个类，ActiveSubmaps2D. 通过看这段代码及其注释我们可以知道，在 cartographer 中总是同时存在着两个 Submap：Old Submap 和 New Submap. Old Submap 是用来做 matching，New submap 则用来 matching next. 每一帧 RangeData 数据都要同时插入到两个 submap 中。当插入 Old Submap 中的传感器帧数达到一定数量（在配置文件 /src/cartographer/configuration_files/trajectory_builder_2d.lua 中设置-submap/num_range_data）时，old submap 就不再改变，这时 Old Submap 开始进行 Loop Closure，被加入到 submap 的 list 中，设置 matching_submap_index 增加 1，然后被 ActiveSubmap 这个 object 所以往，而原先的 new submap 则变成新的 Old Submap，同时通过 AddSubmap 函数重新初始化一个 submap。

代码如下：

```
class ActiveSubmaps2D {

  public:

    explicit ActiveSubmaps2D(const proto::SubmapsOptions2D& options);

    ActiveSubmaps2D(const ActiveSubmaps2D&) = delete;

    ActiveSubmaps2D& operator=(const ActiveSubmaps2D&) = delete;

    // 返回正在执行 scan-to-map matching 的 submap 的 index

    int matching_index() const;

    // 插入传感器数据

    void InsertRangeData(const sensor::RangeData& range_data);

    std::vector<std::shared_ptr<Submap2D>> submaps() const;

   private:

     std::unique_ptr<RangeDataInserterInterface> CreateRangeDataInserter();

     std::unique_ptr<GridInterface> CreateGrid(const Eigen::Vector2f& origin);

     void FinishSubmap();

     void AddSubmap(const Eigen::Vector2f& origin);

     const proto::SubmapsOptions2D options_;

     int matching_submap_index_ = 0;

     std::vector<std::shared_ptr<Submap2D>> submaps_;

     std::unique_ptr<RangeDataInserterInterface> range_data_inserter_;

};
```

一些函数实现如下：

```
ActiveSubmaps2D::ActiveSubmaps2D(const proto::SubmapsOptions2D& options)
```

```
    : options_(options),
      range_data_inserter_(std::move(CreateRangeDataInserter())) {
  // 新的 submap 如果没有更好选择，不妨设置 original 为[0,0]
  AddSubmap(Eigen::Vector2f::Zero());
}
std::vector<std::shared_ptr<Submap2D>> ActiveSubmaps2D::submaps() const {
  return submaps_;
}
int ActiveSubmaps2D::matching_index() const { return matching_submap_index_; }
void ActiveSubmaps2D::InsertRangeData(const sensor::RangeData& range_data) {
  for (auto& submap : submaps_) {
    submap->InsertRangeData(range_data, range_data_inserter_.get());
  }
  //如果插入的 RangeData 达到一定数量，则重新添加一个 submap
  if (submaps_.back()->num_range_data() == options_.num_range_data()) {
    AddSubmap(range_data.origin.head<2>());
  }
}
std::unique_ptr<RangeDataInserterInterface>
ActiveSubmaps2D::CreateRangeDataInserter() {
  return common::make_unique<ProbabilityGridRangeDataInserter2D>(
      options_.range_data_inserter_options()
```

```
          .probability_grid_range_data_inserter_options_2d());
}
std::unique_ptr<GridInterface> ActiveSubmaps2D::CreateGrid(
    const Eigen::Vector2f& origin) {
  constexpr int kInitialSubmapSize = 100;
  float resolution = options_.grid_options_2d().resolution();
  return common::make_unique<ProbabilityGrid>(
      MapLimits(resolution,
                origin.cast<double>() + 0.5 * kInitialSubmapSize * resolution *
                                        Eigen::Vector2d::Ones(),
                CellLimits(kInitialSubmapSize, kInitialSubmapSize)));
}
void ActiveSubmaps2D::FinishSubmap() {
  Submap2D* submap = submaps_.front().get();//把前面的 submap 取出来
  submap->Finish();//让他 finish
  ++matching_submap_index_;//正在匹配的 submap 的 index 指向新的 submap
  submaps_.erase(submaps_.begin());//把开头的 submap 删去
}
void ActiveSubmaps2D::AddSubmap(const Eigen::Vector2f& origin) {
  if (submaps_.size() > 1) {//如果 submap_的 size 不大于 1，说明刚开始创建第一个 submap，不用做任何操作，直接把新的添加进去。否则，要把旧的给 finish 掉
    // This will crop the finished Submap before inserting a new Submap to
```

```
    // reduce peak memory usage a bit.
    FinishSubmap();
  }
  submaps_.push_back(common::make_unique<Submap2D>(
      origin, std::unique_ptr<Grid2D>(
                  static_cast<Grid2D*>(CreateGrid(origin).release()))));
  LOG(INFO) << "Added submap " << matching_submap_index_ + submaps_.size();
}
```

第 10 章 TrajectoryBuilder 部分 3-其他成员变量与函数

在 TrajectoryBuilder 之前部分的代码阅读里，我们基本上是采用的宽度优先搜索的策略，从 cartographer_ros 开始，我们探究代码到 MapBuilderBridge 这一层就不再深入；等把 cartographer_ros 读完了我们再看的 MapBuilderBridge，然后再是 MapBuilder. 由 MapBuilder 开始，我们根据是 Local Slam 还是 Loop Closure 分成了两个并列的部分。

上两篇文章我们开始了用于 Local Slam 的 TrajectoryBuilder 部分的代码阅读，从这部分开始，我们切换到了深度优先搜索的策略，深入地分析了 TrajectoryBuilder 中包含的 Submap 的代码结构，梳理了 Submap 在 cartographer 中是如何维护和更新的。这是因为，到这个层面之后的代码耦合性会比较强，而且深度也不会过深，我们完全有能力把握这种深度的逻辑框架。所以，从这部分开始，我们以深度优先搜索的策略为主。

上一篇文章里我们深入地梳理了 Submap 的代码结构，对 submap 的维护和更新已经非常清楚了，这篇文章让我们进行一下回溯，重新去看一下在 TrajectoryBuilderInterface 这个接口中定义的其他成员变量。

简单回顾一下，我们已经看过了第一个成员变量：

10.1、TrajectoryBuilderInterface

成员变量：

1) 一个结构体 InsertionResult:

```
struct InsertionResult {//InsertionResult 就是用来保存插入 Local Slam 的一个节点的数据结构
    //NodeId 是一个结构体，在/mapping/id.h 中定义，
    //包含两个部分：一个 int 型的 trajectory_id 和一个 int 型的 node_index.
    NodeId node_id;
    //TrajectoryNode 是一个结构体，在/mapping/trajectory_node.h 中定义，
    //TrajectoryNode::Data 包含了经过处理的一帧传感器数据
    std::shared_ptr<const TrajectoryNode::Data> constant_data;
    //已经建立起来的子图列表。
    std::vector<std::shared_ptr<const Submap>> insertion_submaps;//
```

```
};
```

让我们接着往下看：

2) 一个回调函数

```
  // A callback which is called after local SLAM processes an accumulated
  // 'sensor::RangeData'. If the data was inserted into a submap, reports the
  // assigned 'NodeId', otherwise 'nullptr' if the data was filtered out.
  // 这个回调函数是当一个 RangeData 数据进来时被调用主要有 5 个参数：trajectory_id，
  // 该帧数据的插入时刻 Time, 插入到 Local Slam 中的 pose,RangeData 数据本身和插入的结果 InsertionResult。
  // InsertionResult 的结构我们进行了详细地分析
  // 如果该帧数据成功插入了 submap 中则返回节点 id:NodeId，否则返回空指针
  using LocalSlamResultCallback =
      std::function<void(int /* trajectory ID */, common::Time,
                         transform::Rigid3d /* local pose estimate */,
                         sensor::RangeData /* in local frame */,
                         std::unique_ptr<const InsertionResult>)>;
```

3) 定义传感器类型的结构体 SensorId

```
  struct SensorId {
    enum class SensorType {
      RANGE = 0,//Range 传感器。激光或其他可提供点云数据的传感器。如 Kinect。
      IMU,//IMU 数据
      ODOMETRY,//里程计
```

```
    FIXED_FRAME_POSE,
    LANDMARK,
    LOCAL_SLAM_RESULT
  };
  SensorType type;//传感器类型
  std::string id;//传感器 id
  bool operator==(const SensorId& other) const {
    return std::forward_as_tuple(type, id) ==
           std::forward_as_tuple(other.type, other.id);
  }
  bool operator<(const SensorId& other) const {
    return std::forward_as_tuple(type, id) <
           std::forward_as_tuple(other.type, other.id);
  }
};
```

10.2 成员函数：

1) 构造函数和等号重载

```
TrajectoryBuilderInterface() {}
virtual ~TrajectoryBuilderInterface() {}
TrajectoryBuilderInterface(const TrajectoryBuilderInterface&) = delete;
TrajectoryBuilderInterface& operator=(const TrajectoryBuilderInterface&) =
```

```
      delete;
```

2) 处理传感器数据的 5 个纯虚函数：

```
  virtual void AddSensorData(
      const std::string& sensor_id,
      const sensor::TimedPointCloudData& timed_point_cloud_data) = 0;
  virtual void AddSensorData(const std::string& sensor_id,
                             const sensor::ImuData& imu_data) = 0;
  virtual void AddSensorData(const std::string& sensor_id,
                             const sensor::OdometryData& odometry_data) = 0;
  virtual void AddSensorData(
      const std::string& sensor_id,
      const sensor::FixedFramePoseData& fixed_frame_pose) = 0;
  virtual void AddSensorData(const std::string& sensor_id,
                             const sensor::LandmarkData& landmark_data) = 0;
```

3) 直接将 Local Slam 的结果添加到 PoseGraph 的函数

```
  // Allows to directly add local SLAM results to the 'PoseGraph'. Note that it
  // is invalid to add local SLAM results for a trajectory that has a
  // 'LocalTrajectoryBuilder2D/3D'.
  virtual void AddLocalSlamResultData(
      std::unique_ptr<mapping::LocalSlamResultData> local_slam_result_data) = 0;
```

4) 非类成员函数 ToProto、FromProto、PopulateOverlappingSubmapsTrimmerOptions2D、CreateTrajectoryBuilderOptions 等其他函数

这些函数跟主体逻辑关系不大，我们不再细看。

我们对 TrajectoryBuilderInterface 这个接口类的解读就到这儿。该类是由 CollatedTrajectoryBuilder (/mapping/internal/collated_trajectory_builder.h)继承并实现具体方法的。但 CollatedTrajectoryBuilder 的构造函数里又包含一个 TrajectoryBuiderInterface 的智能指针。我们回顾一下 MapBuilder::AddTrajectory (Cartographer 源码阅读 6——MapBuilder 的解读)这个函数可以知道，这个 TrajectoryBuiderInterface 的智能指针是由 CreateGlobalTrajectoryBuilder2D (/mapping/internal/global_trajectory_builder.h) 函数通过一个 LocalTrajectoryBuilder2D 的类生成的。所以，在下一篇文章中，我们将梳理：CollatedTrajectoryBuilder、LocalTrajectoryBuilder2D 和 GlobalTrajectoryBuilder 这三个类的关系。

第 11 章 LocalTrajectoryBuilder2D 成员变量解读

接下来几篇文章里我们要梳理清楚 CollatedTrajectoryBuilder、LocalTrajectoryBuilder2D 和 GlobalTrajectoryBuilder 这三个类的关系。这几个类当中，LocalTrajectoryBuilder2D 这个类是与其他两个类耦合性最少的，其他两个类都依赖于这个类。所以，我们首先从这个类入手。这个类的主要作用就是局部轨迹的构建。在这个类里，我们应该能看到我们一直期望看到的 PoseExtrapolator 和 Scan Matching （详见 Cartographer 源码阅读 1——整体框架介绍中算法流程图）等核心内容。略微有点小兴奋，让我们加油！！！

LocalTrajectoryBuilder2D 定义在/mapping/internal/2d/local_trajectory_builder_2d.h/.cc 中。让我们先来看看它的成员变量：

11.1、结构体 InsertionResult

```
struct InsertionResult {
  std::shared_ptr<const TrajectoryNode::Data> constant_data;//节点数据
  std::vector<std::shared_ptr<const Submap2D>> insertion_submaps;//插入的 submap 的向量
};
```

与 TrajectoryBuilderInterface 中定义的 InsertionResult 相比，这里缺了一个 NodeId.

11.2、结构体 MatchingResult

```
// matching 结果。包括时间、匹配的 local_pose、传感器数据、插入结果等
struct MatchingResult {
  common::Time time;
  transform::Rigid3d local_pose;
  sensor::RangeData range_data_in_local;
  // 如果被 MotionFilter 滤掉了，那么 insertion_result 返回空指针。
  std::unique_ptr<const InsertionResult> insertion_result;
```

```
};
```

11.3、其他成员变量

```
const proto::LocalTrajectoryBuilderOptions2D options_;//参数配置项

ActiveSubmaps2D active_submaps_;//在 mapping/2d/submap_2d.h 中定义。同时维护着两个 submap



MotionFilter motion_filter_;

scan_matching::RealTimeCorrelativeScanMatcher2D

    real_time_correlative_scan_matcher_;//实时的扫描匹配，用的相关分析方法

scan_matching::CeresScanMatcher2D ceres_scan_matcher_;//Ceres 方法匹配。所以两者都是 Scan-to-match 的匹配？方法二选一？

std::unique_ptr<PoseExtrapolator> extrapolator_;//轨迹推算器。融合 IMU，里程计数据

int num_accumulated_ = 0;//累积数据的数量

sensor::RangeData accumulated_range_data_;//该轨迹的累积数据

std::chrono::steady_clock::time_point accumulation_started_;//标记该轨迹的开始时刻

 RangeDataCollator range_data_collator_;//收集传感器数据
```

这些成员变量中，我们需要重点关注 MotionFilter 型的 motion_filter_、PoseExtrapolator 型的智能指针 extrapolator_和用于 Scan Matching 的 RealTimeCorrelativeScanMatcher2D 和 CeresScanMatcher2D。

RealTimeCorrelativeScanMatcher2D 和 CeresScanMatcher2D 等我们看到匹配算法相关代码时再去查看。我们现在可以先看看 MotionFilter 和 PoseExtrapolator 这两个类。

1） MotionFilter

原文：
MotionFilter 定义在/mapping/internal/motion_filter.h 中，其主要作用就是对历史数据进行累加。在一定的时间范围内，IMU、里程计等传感器数据较为准确，可以进行累加。但当两帧数据的间隔时间、或两帧的 Pose 跨过的距离、或两帧的 Pose 转过的角度等超过一定阈值，则判断不可再进行累加。

对于 MotionFilter 的错误，我最初的理解有错误，多谢评论中的提醒。现将其修改如下。

MotionFilter 定义在/mapping/internal/motion_filter.h 中，其主要作用是对数据进行一下滤波。当两帧数据的间隔时间/两帧的 Pose 跨过的距离/两帧的 Pose 转过的角度等不超过一定的阈值时，认为新的数据提供的信息很少，这些数据可以直接舍去。

```
// 从 Proto 流中读取配置文件，设置配置项，主要包括最大时间间隔、最大角度间隔、最大距离间隔等

proto::MotionFilterOptions CreateMotionFilterOptions(

    common::LuaParameterDictionary* parameter_dictionary);

// Takes poses as input and filters them to get fewer poses.

class MotionFilter {

 public:

  explicit MotionFilter(const proto::MotionFilterOptions& options);

  // 根据预先设置的阈值，如果累积运动超过提前预设的阈值，则返回 false；

  // 否则返回 true，然后把该数据累加上

  bool IsSimilar(common::Time time, const transform::Rigid3d& pose);

 private:

  const proto::MotionFilterOptions options_;//配置项，主要包括最大时间间隔、最大角度间隔、最大距离间隔等

  int num_total_ = 0;//总共的 pose 数

  int num_different_ = 0;//过滤之后剩下的 pose 数

  common::Time last_time_;//上一次时间

  transform::Rigid3d last_pose_;//上一次的 pose

};
```

其中，重要的也就是 IsSimilar 这个函数。该函数根据预先设置的阈值，如果累积运动超过提前预设的阈值，则返回 false； 否则返回 true：

```
bool MotionFilter::IsSimilar(const common::Time time,
                             const transform::Rigid3d& pose) {
  LOG_IF_EVERY_N(INFO, num_total_ >= 500, 500)
      << "Motion filter reduced the number of nodes to "
      << 100. * num_different_ / num_total_ << "%.";
  ++num_total_;
  if (num_total_ > 1 && //总的 pose 数大于 1
      time - last_time_ <= common::FromSeconds(options_.max_time_seconds()) && //时间间隔小于给定阈值
      (pose.translation() - last_pose_.translation()).norm() <=
          options_.max_distance_meters() && //移动距离小于给定阈值
      transform::GetAngle(pose.inverse() * last_pose_) <=
          options_.max_angle_radians()) { //偏转角度小于给定阈值
    return true;
  }
  last_time_ = time;
  last_pose_ = pose;
  ++num_different_;
  return false;
}
```

而这三个阈值定义在配置文件/src/cartographer/configuration_files/trajectory_builder_2d.lua 中：

```
  motion_filter = {
```

```
    max_time_seconds = 5.,//5s

    max_distance_meters = 0.2,//0.2 米

    max_angle_radians = math.rad(1.),//1°

  },
```

2） PoseExtrapolator

PoseExtrapolator 定义在/mapping/pose_extrapolator.h 中。其主要作用是对 IMU、里程计数据进行融合，估计机器人的实时位姿。

从 cartographer 算法的流程框图(见 Cartographer 源码阅读 1——整体框架介绍中第一个图)我们可以知道，PoseExtrapolator 的输入有三个：从里程计传来的数据、经过 ImuTracker 处理过的对重力进行 aligned 过的 IMU 数据、从 Scan Matching 输出的 Pose Observation。PoseExtrapolator 对这三个输入进行融合后输出估计出来的 PoseEstimate.

从该类开头的一段注释我们也可以知道，该类通过从 Scan Matching 输出的 PoseObservation 持续一段时间来跟踪 Poses，从而估计机器人的线速度和角速度。通过速度来解算机器人的运动。当 IMU 或里程计数据可用时，可与这些数据融合来提升解算结果。

首先看一下该类的一些成员变量：

```
  // Duration 是在/common/time.h 中定义的别名

  // using Duration = UniversalTimeScaleClock::duration;//微秒,0.1us

  // UniversalTimeScaleClock 类实现 c++11 的 clock 接口，以 0.1us 为时间精度。

  const common::Duration pose_queue_duration_;

  // 带时间的 Pose

  struct TimedPose {

    common::Time time;

    transform::Rigid3d pose;

  };

  // 带时间的 Pose 队列
```

```
std::deque<TimedPose> timed_pose_queue_;//存储要持续跟踪的 Poses,应该是从 ScanMatching 输出的 PoseObservation

// 从持续跟踪一段时间的 Pose 队列中估计出来的线速度，初始化为 0

Eigen::Vector3d linear_velocity_from_poses_ = Eigen::Vector3d::Zero();

// 从持续跟踪一段时间的 Pose 队列中估计出来的线速度，初始化为 0

Eigen::Vector3d angular_velocity_from_poses_ = Eigen::Vector3d::Zero();

const double gravity_time_constant_;//重力的时间间隔?

std::deque<sensor::ImuData> imu_data_;//存储 IMU 数据的队列

std::unique_ptr<ImuTracker> imu_tracker_;//ImuTracker,定义在/mapping/imu_tracker.h 中。

std::unique_ptr<ImuTracker> odometry_imu_tracker_;//跟踪里程计

std::unique_ptr<ImuTracker> extrapolation_imu_tracker_;//三个 ImuTracker 的智能指针，其中区别还不太懂

// imu_tracker_是存放由 IMU 数据得到的信息

// odometry_imu_tracker_是存放由里程计得到的信息

// extrapolation_imu_tracker_是存放经过数据融合后的结果

TimedPose cached_extrapolated_pose_;//缓存的一个带时间的 Pose，

std::deque<sensor::OdometryData> odometry_data_;//里程计数据

// 从里程计获取到的线速度，初始化为 0

Eigen::Vector3d linear_velocity_from_odometry_ = Eigen::Vector3d::Zero();

// 从里程计获取到的角速度，初始化为 0

Eigen::Vector3d angular_velocity_from_odometry_ = Eigen::Vector3d::Zero();
```

可以看到，与该类的功能相对应，成员变量里包括了：标记持续时间的长短的 pose_queue_duration_、记录这段时间的 Pose 的一个队列 timed_pose_queue_、从 pose 队列中估计出来的线速度和角速度，以及分别从 IMU 和里程计获得的信息。

接下来我们逐个看一下这些成员函数：

1. 构造函数

```
PoseExtrapolator::PoseExtrapolator(const common::Duration pose_queue_duration,
                                   double imu_gravity_time_constant)
    : pose_queue_duration_(pose_queue_duration),
      gravity_time_constant_(imu_gravity_time_constant),
      cached_extrapolated_pose_{common::Time::min(),
                                transform::Rigid3d::Identity()} {}//初始化
cached_extrapolated_pose_为单位矩阵 I
```

2. 从私有函数开始，首先是 UpdateVelocitiesFromPoses，即从一个 Pose 队列中估计机器人的线速度和角速度：

```
// 取出 timed_pose_queue_这个队列中最早和最新的两个 Pose 做差，然后除以时间得到机器人的速度。
void PoseExtrapolator::UpdateVelocitiesFromPoses() {
  if (timed_pose_queue_.size() < 2) {// 判断这个 Pose 队列的长度，如果小于 2 则没法进行估计
    // We need two poses to estimate velocities.
    return;
  }
  CHECK(!timed_pose_queue_.empty());//检查队列是否为空
  // 取出队列最末尾的一个 Pose,也就是最新时间点的 Pose,并记录相应的时间
  const TimedPose& newest_timed_pose = timed_pose_queue_.back();
  const auto newest_time = newest_timed_pose.time;
  // 取出队列最开头的一个 Pose，也就是最旧时间点的 Pose,并记录相应的时间
  const TimedPose& oldest_timed_pose = timed_pose_queue_.front();
```

```
  const auto oldest_time = oldest_timed_pose.time;
  // 两者的时间差
  const double queue_delta = common::ToSeconds(newest_time - oldest_time);
  // 如果时间差小于 1ms，则估计不准，弹出警告信息
  if (queue_delta < 0.001) {  // 1 ms
    LOG(WARNING) << "Queue too short for velocity estimation. Queue duration: "
                 << queue_delta << " ms";
    return;
  }
  // 获取两个时刻各自的 Pose
  const transform::Rigid3d& newest_pose = newest_timed_pose.pose;
  const transform::Rigid3d& oldest_pose = oldest_timed_pose.pose;
  // 线速度即为两个 Pose 的 translation 部分相减后除以间隔时间
  linear_velocity_from_poses_ =
      (newest_pose.translation() - oldest_pose.translation()) / queue_delta;
  // 角速度是两个 Pose 的 rotation 部分的差
  angular_velocity_from_poses_ =
      transform::RotationQuaternionToAngleAxisVector(
          oldest_pose.rotation().inverse() * newest_pose.rotation()) /
      queue_delta;
}
```

该函数的主要工作就是取出 timed_pose_queue_这个队列中最早和最新的两个 Pose 做差，然后除以时间得到机器人的速度。

这里特殊说明一下，姿态的差是转成了一个角轴向量：

其中，　　是由旋转矩阵向角轴向量的一个映射。

3. 分别管理 IMU 数据队列和里程计数据队列的 TrimImuData 和 TrimOdometryData

```
// 删去队列中无用的 IMU 数据
void PoseExtrapolator::TrimImuData() {
  // 需要满足三个条件：IMU 数据队列大于 1，Pose 的队列不为空，IMU 数据队列的第一个元素时间小于 Pose 队列的最后一个元素的时间
  // 最后一个条件意味着当 IMU 数据的时间比一个最新的 Pose 的时间要早时，说明这个 IMU 数据已经过期了。所以从队列中删掉就可以了。
  // 知道 IMU 数据的时间要比最新的 Pose 时间晚，那么说明这时候这个数据还有用。
  // 这种情况就不再删了，跳出循环，等待其他程序取出队列最开头的 IMU 数据进行融合
  while (imu_data_.size() > 1 && !timed_pose_queue_.empty() &&
         imu_data_[1].time <= timed_pose_queue_.back().time) {
    imu_data_.pop_front();
  }
}
// 与处理 IMU 队列逻辑相同，删去队列中无用的里程计数据
void PoseExtrapolator::TrimOdometryData() {
  while (odometry_data_.size() > 2 && !timed_pose_queue_.empty() &&
         odometry_data_[1].time <= timed_pose_queue_.back().time) {
    odometry_data_.pop_front();
```

```
  }

}
```

4. AdvanceImuTracker

该函数的主要作用就是取出 IMU 数据队列中的数据，更新 ImuTracker。

由于这个函数调用了 ImuTracker 的很多方法。我们不妨先来看看 ImuTracker (/mapping/imu_tracker.h)的逻辑。

```
class ImuTracker {

 public:

  ImuTracker(double imu_gravity_time_constant, common::Time time);

  // Advances to the given 'time' and updates the orientation to reflect this.

  // 把 ImuTracker 更新到指定时刻 time，并把响应的 orientation_,gravity_vector_和 time_进行更新

  // 指定的时间 time 要比 ImuTracker 维护的时间 time_晚，否则不需要更新。

  void Advance(common::Time time);

  // Updates from an IMU reading (in the IMU frame).

  // 根据传感器读数更新传感器的最新状态，得到经过重力校正的线加速度、角速度等。

  void AddImuLinearAccelerationObservation(

      const Eigen::Vector3d& imu_linear_acceleration);

  void AddImuAngularVelocityObservation(

      const Eigen::Vector3d& imu_angular_velocity);

  // Query the current time.

  // 返回当前时间

  common::Time time() const { return time_; }
```

```
  // Query the current orientation estimate.
  // 返回当前姿态
  Eigen::Quaterniond orientation() const { return orientation_; }
 private:
  const double imu_gravity_time_constant_;// align 重力的时间间隔
  common::Time time_;//当前时间
  common::Time last_linear_acceleration_time_;//记录的上一个线加速度的时刻
  Eigen::Quaterniond orientation_;//当前姿态
  Eigen::Vector3d gravity_vector_;//当前重力方向
  Eigen::Vector3d imu_angular_velocity_;//角速度
};
```

ImuTracker 的主要作用就是根据 IMU 的读数维护传感器当前的姿态、线加速度(经过重力校正的)、当前姿态、重力方向、角速度等量。这些量都是以 ImuTracker 刚建立时的那一时刻 IMU 本身的坐标系为基准坐标系。具体函数的实现代码如下：

```
ImuTracker::ImuTracker(const double imu_gravity_time_constant,
                       const common::Time time)
    : imu_gravity_time_constant_(imu_gravity_time_constant),
      time_(time),
      last_linear_acceleration_time_(common::Time::min()),//
      orientation_(Eigen::Quaterniond::Identity()),//初始姿态为 I
      gravity_vector_(Eigen::Vector3d::UnitZ()),//重力方向初始化为[0,0,1]
      imu_angular_velocity_(Eigen::Vector3d::Zero()) {}//角速度初始化为 0
// 把 ImuTracker 更新到指定时刻 time，并把响应的 orientation_,gravity_vector_和 time_进行更新
```

```
// 指定的时间 time 要比 ImuTracker 维护的时间 time_晚，否则不需要更新。
void ImuTracker::Advance(const common::Time time) {
  CHECK_LE(time_, time);//检查 ImuTracker 维护的时间 time_是否大于等于指定时间 time
  // 求指定时间相对当前时间的时间差
  const double delta_t = common::ToSeconds(time - time_);
  // 角速度乘以时间，然后转化成 RotationnQuaternion,这是这段时间的姿态变化量
  const Eigen::Quaterniond rotation =
      transform::AngleAxisVectorToRotationQuaternion(
          Eigen::Vector3d(imu_angular_velocity_ * delta_t));
  // 以当前姿态 orientation_为基准，再乘以姿态变化量。得到最新的姿态
  orientation_ = (orientation_ * rotation).normalized();
  // 更新重力方向
  gravity_vector_ = rotation.conjugate() * gravity_vector_;
  // 更新时间
  time_ = time;
}
// 根据读数更新线加速度。这里的线加速度是经过重力校正的。
void ImuTracker::AddImuLinearAccelerationObservation(
    const Eigen::Vector3d& imu_linear_acceleration) {
  // Update the 'gravity_vector_' with an exponential moving average using the
  // 'imu_gravity_time_constant'.
  // 求时间差
```

```
  const double delta_t =
      last_linear_acceleration_time_ > common::Time::min()
          ? common::ToSeconds(time_ - last_linear_acceleration_time_)
          : std::numeric_limits<double>::infinity();
  // 更新一下 last_linear_acceleration_time_;
  last_linear_acceleration_time_ = time_;
  const double alpha = 1. - std::exp(-delta_t / imu_gravity_time_constant_);
  gravity_vector_ =
      (1. - alpha) * gravity_vector_ + alpha * imu_linear_acceleration;
  // Change the 'orientation_' so that it agrees with the current
  // 'gravity_vector_'.
  const Eigen::Quaterniond rotation = Eigen::Quaterniond::FromTwoVectors(
      gravity_vector_, orientation_.conjugate() * Eigen::Vector3d::UnitZ());
  orientation_ = (orientation_ * rotation).normalized();
  CHECK_GT((orientation_ * gravity_vector_).z(), 0.);
  CHECK_GT((orientation_ * gravity_vector_).normalized().z(), 0.99);
}
// 根据读数更新角速度
void ImuTracker::AddImuAngularVelocityObservation(
    const Eigen::Vector3d& imu_angular_velocity) {
  imu_angular_velocity_ = imu_angular_velocity;
}
```

重力校正这一块的公式我每太看懂，但由于时间关系，就不再深入了解了。读者如有兴趣可自行查看。

所以，PoseExtrapolator::AdvanceImuTracker 的主要作用就是从 IMU 数据队列中取出最新的数据，更新 ImuTracker 的状态到指定的时间 time。

```
// 从 IMU 数据队列中取出最新的数据，更新 ImuTracker 的状态到指定的时间 time

void PoseExtrapolator::AdvanceImuTracker(const common::Time time,

                                         ImuTracker* const imu_tracker) const {

  CHECK_GE(time, imu_tracker->time());//当前时间是否晚于 ImuTracker 的时间,如果是，需要更新 ImuTracker

  if (imu_data_.empty() || time < imu_data_.front().time) {

    // 如果 IMU 数据队列为空，或当前时间要比 IMU 数据队列中最早期的时间还要早。说明没有可用的 IMU 数据。

    // 这时候根据时间推算。对于角速度，是用从 Pose 中估计的角速度或从里程计获得的角速度更新 ImuTracker 的角速度

    imu_tracker->Advance(time);

    // 用一个 fake 的重力方向更新线加速度。这个 fake 的重力方向就是[0,0,1]。理想情况。

    imu_tracker->AddImuLinearAccelerationObservation(Eigen::Vector3d::UnitZ());

    imu_tracker->AddImuAngularVelocityObservation(

        odometry_data_.size() < 2 ? angular_velocity_from_poses_

                                  : angular_velocity_from_odometry_);

    return;

  }

  // 如果 ImuTracker 维护的时间早于 IMU 数据队列最早期的时间

  if (imu_tracker->time() < imu_data_.front().time) {
```

```
    // Advance to the beginning of 'imu_data_'.
    // 先把 ImuTracker 更新到 IMU 数据来临的那一刻
    imu_tracker->Advance(imu_data_.front().time);
  }
  //然后依次取出 IMU 数据队列中的数据，更新 ImuTracker，直到 IMU 数据的时间比指定时间 time 要晚。
  auto it = std::lower_bound(
      imu_data_.begin(), imu_data_.end(), imu_tracker->time(),
      [](const sensor::ImuData& imu_data, const common::Time& time) {
        return imu_data.time < time;
      });
  while (it != imu_data_.end() && it->time < time) {
    imu_tracker->Advance(it->time);
    imu_tracker->AddImuLinearAccelerationObservation(it->linear_acceleration);
    imu_tracker->AddImuAngularVelocityObservation(it->angular_velocity);
    ++it;
  }
  imu_tracker->Advance(time);
}
```

5. 如下两个函数 ExtrapolateRotation 和 ExtrapolateTranslation 分别解算姿态和位移的增长量：

```
// 结算姿态的变化量
Eigen::Quaterniond PoseExtrapolator::ExtrapolateRotation(
```

```
    const common::Time time, ImuTracker* const imu_tracker) const {
  CHECK_GE(time, imu_tracker->time());//检查指定时间是否大于等于 ImuTracker 的时间。
  AdvanceImuTracker(time, imu_tracker);//更新 ImuTracker 到指定的 time。
  // 上一时刻的姿态
  const Eigen::Quaterniond last_orientation = imu_tracker_->orientation();
  // 求取姿态变化量：上一时刻姿态的逆乘以当前的姿态
  return last_orientation.inverse() * imu_tracker->orientation();
}
// 解算位移的增长量
Eigen::Vector3d PoseExtrapolator::ExtrapolateTranslation(common::Time time) {
  // 取出 Pose 队列中最新的时刻
  const TimedPose& newest_timed_pose = timed_pose_queue_.back();
  // 算时间差
  const double extrapolation_delta =
      common::ToSeconds(time - newest_timed_pose.time);
  // 如果有里程计数据，则更信任里程计速度，直接把从里程计处获得的线速度乘以时间
  if (odometry_data_.size() < 2) {
    return extrapolation_delta * linear_velocity_from_poses_;
  }
  // 没有里程计数据的话则把从 Pose 队列中估计的线速度乘以时间
  return extrapolation_delta * linear_velocity_from_odometry_;
}
```

从上面两个函数可以看出，对于姿态而言，更信任 IMU 的数据，如果由 IMU 数据则用 IMU 数据；对于 Translation 而言，更信任里程计数据，如果有里程计数据则优化使用里程计数据。没有的话就用 Pose 队列估计出来的线速度。而对于 IMU 的线速度并不信任。这跟 IMU 的传感器特性是相关的。

接下来我们看一下 PoseExtrapolator 的一些 public 的成员函数：

6. InitializeWithImu: 根据 IMU 数据来初始化一个 PoseExtrapolator

```
std::unique_ptr<PoseExtrapolator> PoseExtrapolator::InitializeWithImu(
    const common::Duration pose_queue_duration,
    const double imu_gravity_time_constant, const sensor::ImuData& imu_data) {
  // 先生成一个 PoseExtrapolator 的变量
  auto extrapolator = common::make_unique<PoseExtrapolator>(
      pose_queue_duration, imu_gravity_time_constant);
  // 添加 IMU 数据
  extrapolator->AddImuData(imu_data);
  // 生成 ImuTracker 并更新
  extrapolator->imu_tracker_ =
      common::make_unique<ImuTracker>(imu_gravity_time_constant, imu_data.time);
  extrapolator->imu_tracker_->AddImuLinearAccelerationObservation(
      imu_data.linear_acceleration);
  extrapolator->imu_tracker_->AddImuAngularVelocityObservation(
      imu_data.angular_velocity);
  extrapolator->imu_tracker_->Advance(imu_data.time);
  // 添加一个 Pose。从这儿来看，Pose 队列并不是从 ScanMatching 而来的，而是从 Imu 测量得到的。
  extrapolator->AddPose(
```

```
      imu_data.time,
      transform::Rigid3d::Rotation(extrapolator->imu_tracker_->orientation()));
  return extrapolator;
}
```

这里对前面我们预计过的 Pose 的来源做一个更正：从这段代码来看，Pose 队列并不是从 ScanMatching 而来的，而是从 Imu 测量得到的。

这里调用了 PoseExtrapolator::AddImuData 和 PoseExtrapolator::AddPose。这两个函数分别解读如下：

7. PoseExtrapolator::AddImuData

```
// 把新的 IMU 数据添加到队列中，删去队列中的过期数据
void PoseExtrapolator::AddImuData(const sensor::ImuData& imu_data) {
  CHECK(timed_pose_queue_.empty() ||
        imu_data.time >= timed_pose_queue_.back().time);
  imu_data_.push_back(imu_data);
  TrimImuData();
}
```

8. PoseExtrapolator::AddPose

```
// 在时刻 time 往 Pose 队列中添加一个 Pose
void PoseExtrapolator::AddPose(const common::Time time,
                               const transform::Rigid3d& pose) {
  if (imu_tracker_ == nullptr) {// 如果 ImuTracker 还没有建立，则需要建立一个 ImuTracker
    common::Time tracker_start = time;//
```

```
    if (!imu_data_.empty()) {// 如果 IMU 数据队列不为空，则以当前时间和 IMU 数据中的最早时刻的较小值为初始时刻建立一个 ImuTracker
      tracker_start = std::min(tracker_start, imu_data_.front().time);
    }
    //  生成 ImuTracker
    imu_tracker_ =
        common::make_unique<ImuTracker>(gravity_time_constant_, tracker_start);
  }
  //  把 Pose 压入队列
  timed_pose_queue_.push_back(TimedPose{time, pose});
  //Pose 队列大于 2，并且时间间隔已经大于我们设定的 pose_queue_duration_时，则把队列之前的元素删除
  while (timed_pose_queue_.size() > 2 &&
         timed_pose_queue_[1].time <= time - pose_queue_duration_) {
    timed_pose_queue_.pop_front();
  }
  UpdateVelocitiesFromPoses();//更新最新估计的速度
  AdvanceImuTracker(time, imu_tracker_.get());//更新 ImuTracker
  TrimImuData();//更新 IMU 数据队列
  TrimOdometryData();//更新里程计数据队列
  // 里程计和融合结果都以当前 IMU 的 tracker 为准。
  odometry_imu_tracker_ = common::make_unique<ImuTracker>(*imu_tracker_);
  extrapolation_imu_tracker_ = common::make_unique<ImuTracker>(*imu_tracker_);
```

```
}
```

9. PoseExtrapolator::GetLastPoseTime()和 PoseExtrapolator::GetLastExtrapolatedTime()

```
// 返回 Pose 队列的最新时间
common::Time PoseExtrapolator::GetLastPoseTime() const {
  if (timed_pose_queue_.empty()) {// 如果 Pose 队列为空
    return common::Time::min();
  }
  return timed_pose_queue_.back().time;
}
// 返回解算结果的最新时间
common::Time PoseExtrapolator::GetLastExtrapolatedTime() const {
  if (!extrapolation_imu_tracker_) {
    return common::Time::min();
  }
  return extrapolation_imu_tracker_->time();
}
```

10. 添加里程计数据

第 12 章 LocalTrajectoryBuilder2D 成员函数解读

12.1、添加并解算里程计数据

```
// 添加里程计数据
void PoseExtrapolator::AddOdometryData(
    const sensor::OdometryData& odometry_data) {
  CHECK(timed_pose_queue_.empty() ||odometry_data.time >= timed_pose_queue_.back().time);
  //里程计数据时间是否晚于 Pose 队列的最新时间
  odometry_data_.push_back(odometry_data);//压入里程计数据队列
  TrimOdometryData();//删去过期数据
  if (odometry_data_.size() < 2) {//如果里程计数据小于 2，则添加完就完事了
    return;
  }
  // TODO(whess): Improve by using more than just the last two odometry poses.
  // Compute extrapolation in the tracking frame.
  // 如果里程计数据大于 2 了，则我们可以根据里程计数据也进行一下状态估计。
  // 把里程计数据里最早的和最新的数据取出，
  const sensor::OdometryData& odometry_data_oldest = odometry_data_.front();
  const sensor::OdometryData& odometry_data_newest = odometry_data_.back();
  // 计算时间间隔
  const double odometry_time_delta =
```

```
        common::ToSeconds(odometry_data_oldest.time - odometry_data_newest.time);
    // 计算姿态变化量
    const transform::Rigid3d odometry_pose_delta =
        odometry_data_newest.pose.inverse() * odometry_data_oldest.pose;
    // 姿态变化量除以时间得到角速度
    angular_velocity_from_odometry_ = transform::RotationQuaternionToAngleAxisVector(
          odometry_pose_delta.rotation()) /odometry_time_delta;
    if (timed_pose_queue_.empty()) {
       return;
  }
  // 解算从里程计得到的线速度
    const Eigen::Vector3d linear_velocity_in_tracking_frame_at_newest_odometry_time =
          odometry_pose_delta.translation() / odometry_time_delta;//在里程计坐标系中的线速度
    const Eigen::Quaterniond orientation_at_newest_odometry_time =
        timed_pose_queue_.back().pose.rotation() *
        ExtrapolateRotation(odometry_data_newest.time,
        odometry_imu_tracker_.get());//计算里程计坐标系与基准坐标系的变化姿态
    linear_velocity_from_odometry_ =orientation_at_newest_odometry_time *
      linear_velocity_in_tracking_frame_at_newest_odometry_time;//换算成基准坐标系下的线速度
}
```

12.2、::ExtrapolatePose 和::EstimateGravityOrientation

```
// 推算指定时刻的 Pose
transform::Rigid3d PoseExtrapolator::ExtrapolatePose(const common::Time time) {
  // 取出 Pose 队列中的最新值
  const TimedPose& newest_timed_pose = timed_pose_queue_.back();
  CHECK_GE(time, newest_timed_pose.time);//当前 time 是否比队列中的时间更新
  if (cached_extrapolated_pose_.time != time) {//如果缓存的 Pose 的时间不等于指定时间
    // 最新 translation 等于最新 Pose 的 translation 再加上这段时间的位移增长量
    const Eigen::Vector3d translation =
        ExtrapolateTranslation(time) + newest_timed_pose.pose.translation();
    // 姿态同理
    const Eigen::Quaterniond rotation =
        newest_timed_pose.pose.rotation() *
        ExtrapolateRotation(time, extrapolation_imu_tracker_.get());
    cached_extrapolated_pose_ =
        TimedPose{time, transform::Rigid3d{translation, rotation}};
  }
  return cached_extrapolated_pose_.pose;
}

// 估计经过重力 align 之后的姿态
Eigen::Quaterniond PoseExtrapolator::EstimateGravityOrientation(
    const common::Time time) {
```

```
  ImuTracker imu_tracker = *imu_tracker_;

  AdvanceImuTracker(time, &imu_tracker);

  return imu_tracker.orientation();

}
```

我们对 PoseExtrapolator 做一个总结：

PoseExtrapolator 总结：
》PoseExtrapolator 中管理了三个队列：imu_data_，odometry_data_，和 timed_pose_queue_。IMU 数据和里程计数据都是通过 PoseExtrapolator 来管理和处理。
》PoseExtrapolator 负责解算机器人的位姿。估计机器人的线速度和角速度。

接下来，本文的重点是分析 LocalTrajectoryBuilder2D 中的成员函数。

这一部分就是对点云的处理，然后进行匹配、插入子图等操作。懒得再逐一分析了。偷个懒，把带注释版代码全部堆到下面了。其中，AddRangeData 这个函数是最顶层的函数，它在对点云数据进行预处理之后，调用了 TransformToGravityAlignedFrameAndFilter 函数进行重力 Align。然后调用 AddAccumulatedRangeData 进行匹配(调用 ScanMatching 完成)、往 Submap 插入数据（调用 InsertIntoSubmap 完成）等操作。此外，AddImuData、AddOdometryData、InitializeExtrapolator 几个函数也都很简单，读者请自行查看。

```
LocalTrajectoryBuilder2D::LocalTrajectoryBuilder2D(//为几个成员变量初始化

    const proto::LocalTrajectoryBuilderOptions2D& options,

    const std::vector<std::string>& expected_range_sensor_ids)

    : options_(options),

      active_submaps_(options.submaps_options()),

      motion_filter_(options_.motion_filter_options()),

      real_time_correlative_scan_matcher_(

          options_.real_time_correlative_scan_matcher_options()),

      ceres_scan_matcher_(options_.ceres_scan_matcher_options()),

      range_data_collator_(expected_range_sensor_ids) {}
```

```
LocalTrajectoryBuilder2D::~LocalTrajectoryBuilder2D() {}

// 将 RangeData 转化成重力校正后的数据，并经过 VoxelFilter

// 关于 VoxelFilter 和 AdaptiveVoxelFilter，可参见如下两个参考链接：

// 源码解读：https://blog.csdn.net/learnmoreonce/article/details/76218136

// VoxelFilter 的原理：https://blog.csdn.net/xiaoma_bk/article/details/81780066

// cartographer 中的 VoxelFilter 是把空间分为很多个立方体的栅格，然后一个栅格内可能有很多点，

// 但只用一个点来代表整个栅格中的所有点

// 简单说，这些滤波就是对点云数据的预处理：去除不合理的范围的点、合并相同位置的点等，从而减少点云数量，提高点云质量

sensor::RangeData

LocalTrajectoryBuilder2D::TransformToGravityAlignedFrameAndFilter(

    const transform::Rigid3f& transform_to_gravity_aligned_frame,

    const sensor::RangeData& range_data) const {

  //裁剪数据，指定一定 z 轴范围内的数据。在 /src/cartographer/configuration_files/trajectory_builder_2d.lua 中找到

const sensor::RangeData cropped =

    sensor::CropRangeData(sensor::TransformRangeData(

                          range_data, transform_to_gravity_aligned_frame),

                          options_.min_z(), options_.max_z());//-0.8 到 2 之间

  // 进行体素化滤波

  return sensor::RangeData{

      cropped.origin,

      sensor::VoxelFilter(options_.voxel_filter_size()).Filter(cropped.returns),
```

```
      sensor::VoxelFilter(options_.voxel_filter_size()).Filter(cropped.misses)};
}
// 扫描匹配。输入是 1. 时间, 2. 由 PoseExtrapolator 预测的初始位姿 pose_prediction
// 3. 经过重力 aligned 的 RangeData
// 输出是对该帧传感器数据的最优 pose. 采用 Ceres Scan Matcher
std::unique_ptr<transform::Rigid2d> LocalTrajectoryBuilder2D::ScanMatch(
    const common::Time time, const transform::Rigid2d& pose_prediction,
    const sensor::RangeData& gravity_aligned_range_data) {
  // 获取要进行扫描匹配的 Submap 的 index.
  std::shared_ptr<const Submap2D> matching_submap = active_submaps_.submaps().front();
  // The online correlative scan matcher will refine the initial estimate for
  // the Ceres scan matcher.
  // Online Correlative Scan Matcher 会 refine 初始 Pose.
  transform::Rigid2d initial_ceres_pose = pose_prediction;
  // 生成一个自适应体素滤波器
  sensor::AdaptiveVoxelFilter adaptive_voxel_filter(
      options_.adaptive_voxel_filter_options());
  // 对数据进行一下自适应体素滤波
  const sensor::PointCloud filtered_gravity_aligned_point_cloud =
      adaptive_voxel_filter.Filter(gravity_aligned_range_data.returns);
  // 如果滤波后结果为空，则返回空指针
  if (filtered_gravity_aligned_point_cloud.empty()) {
```

```
    return nullptr;
  }
  // 如果配置项设置使用 OnlineCorrelativeScanMatching。但配置项里默认该项为 false
  if (options_.use_online_correlative_scan_matching()) {
    CHECK_EQ(options_.submaps_options().grid_options_2d().grid_type(),
             proto::GridOptions2D_GridType_PROBABILITY_GRID);//检查是否为概率图
    double score = real_time_correlative_scan_matcher_.Match(
        pose_prediction, filtered_gravity_aligned_point_cloud,
        *static_cast<const ProbabilityGrid*>(matching_submap->grid()),
        &initial_ceres_pose);//调用 RealTimeCorrelativeScanMatcher2D 的方法进行匹配。返回一个
得分
    kFastCorrelativeScanMatcherScoreMetric->Observe(score);
  }
  // 调用 Ceres 库来实现匹配。匹配结果放到 pose_observation 中
  auto pose_observation = common::make_unique<transform::Rigid2d>();
  ceres::Solver::Summary summary;
  ceres_scan_matcher_.Match(pose_prediction.translation(), initial_ceres_pose,
                            filtered_gravity_aligned_point_cloud,
                            *matching_submap->grid(), pose_observation.get(),
                            &summary);
  // 统计残差
  if (pose_observation) {
```

```
    kCeresScanMatcherCostMetric->Observe(summary.final_cost);
    double residual_distance =
        (pose_observation->translation() - pose_prediction.translation())
            .norm();
    kScanMatcherResidualDistanceMetric->Observe(residual_distance);
    double residual_angle = std::abs(pose_observation->rotation().angle() -
                                     pose_prediction.rotation().angle());
    kScanMatcherResidualAngleMetric->Observe(residual_angle);
  }
  // 返回优化后的 pose
  return pose_observation;
}

// 添加 RangeData，返回匹配结果。这是最上层的函数。通过该函数调用其他各种函数。
std::unique_ptr<LocalTrajectoryBuilder2D::MatchingResult>
LocalTrajectoryBuilder2D::AddRangeData(
    const std::string& sensor_id,
    const sensor::TimedPointCloudData& unsynchronized_data) {
  // RangeDataCollator 定义在/mapping/internal/range_data_collator.h 中。
  // 该类的注释：
  // Synchronizes TimedPointCloudData from different sensors. Input needs only be
  // monotonous in 'TimedPointCloudData::time', output is monotonous in per-point
```

```
// timing. Up to one message per sensor is buffered, so a delay of the period of

// the slowest sensor may be introduced, which can be alleviated by passing

// subdivisions.

// 该类的主要任务是把来自不同传感器的 TimePointCloudData 进行一下同步。TimePointCloudData 是带时间的点云数据

// 3D 情况下，前 3 个元素是点的坐标，第 4 个元素是测量到每个点的相对时间。时间以 s 为单位，以最后一个点的捕获时间为 0，

// 越早捕获的点时间值越大。对于 2d 情况，第 3 个元素始终为 0，第 4 个元素同样表示时间。

/* TimedPointCloudData 定义如下

     struct TimedPointCloudData {

        common::Time time;//时间

        Eigen::Vector3f origin;//原点

        sensor::TimedPointCloud ranges;//点云

    };

    */

auto synchronized_data =

    range_data_collator_.AddRangeData(sensor_id, unsynchronized_data);

if (synchronized_data.ranges.empty()) {

  LOG(INFO) << "Range data collator filling buffer.";

  return nullptr;

}

// 初始化 PoseExtrapolator

const common::Time& time = synchronized_data.time;
```

```
//取点云获取的时间为基准为 PoseExtrapolator 初始化

// Initialize extrapolator now if we do not ever use an IMU.

if (!options_.use_imu_data()) {

  InitializeExtrapolator(time);

}

if (extrapolator_ == nullptr) {

  // Until we've initialized the extrapolator with our first IMU message, we

  // cannot compute the orientation of the rangefinder.

  LOG(INFO) << "Extrapolator not yet initialized.";

  return nullptr;

}

// 数据是否为空

CHECK(!synchronized_data.ranges.empty());

// TODO(gaschler): Check if this can strictly be 0.

//看一下数据点的第 4 个元素是不是大于等于 0。2d 情况第 3 个元素都是 0.第四个元素是时间

CHECK_LE(synchronized_data.ranges.back().point_time[3], 0.f);//检查最后一个点的时间是否为 0。正常情况应该都为 0

// 第一个点的时间就等于点云集获取的时间加上第一个点记录的相对时间

const common::Time time_first_point =

    time +

    common::FromSeconds(synchronized_data.ranges.front().point_time[3]);

// 如果该时间比 PoseExtrapolator 的最新时间还要早，说明在第一个点被捕获时，PoseExtrapolator 还没初始化
```

```
  if (time_first_point < extrapolator_->GetLastPoseTime()) {
    LOG(INFO) << "Extrapolator is still initializing.";
    return nullptr;
  }
  // 轨迹开始时刻。只在初始时执行一次。之后 num_accumulated 不再等于 0，该语句就不执行了
  if (num_accumulated_ == 0) {
    accumulation_started_ = std::chrono::steady_clock::now();
  }
  // 遍历每个点，计算在每个点时 PoseExtrapolator 推算出来的机器人的 Pose，放入
range_data_poses 这个向量中
  std::vector<transform::Rigid3f> range_data_poses;
  // 为该集合预分配内存大小
  range_data_poses.reserve(synchronized_data.ranges.size());
  bool warned = false;
  for (const auto& range : synchronized_data.ranges) {//遍历点集
    // 点云中每个点的捕获时刻
    common::Time time_point = time + common::FromSeconds(range.point_time[3]);
    if (time_point < extrapolator_->GetLastExtrapolatedTime()) {//如果该时刻早于
PoseExtrapolator 的时间
      if (!warned) {
        LOG(ERROR)
            << "Timestamp of individual range data point jumps backwards from "
            << extrapolator_->GetLastExtrapolatedTime() << " to " << time_point;
```

```
        warned = true;
      }
      time_point = extrapolator_->GetLastExtrapolatedTime();//时间设置为 PoseExtrapolator 的
时间
    }
    range_data_poses.push_back(
        extrapolator_->ExtrapolatePose(time_point).cast<float>());//推算 time_point 这个时间
点的 Pose
  }
  if (num_accumulated_ == 0) {
    // 'accumulated_range_data_.origin' is uninitialized until the last
    // accumulation.
    accumulated_range_data_ = sensor::RangeData{{}, {}, {}};
  }
  // Drop any returns below the minimum range and convert returns beyond the
  // maximum range into misses.
  // 遍历每一个点
  for (size_t i = 0; i < synchronized_data.ranges.size(); ++i) {
    // 获取第 i 帧点云。
    const Eigen::Vector4f& hit = synchronized_data.ranges[i].point_time;
    // 第 i 帧点云的原点
    const Eigen::Vector3f origin_in_local =
        range_data_poses[i] *
```

```
        synchronized_data.origins.at(synchronized_data.ranges[i].origin_index);
    // 将 hit 集转变成在 Local 坐标系下
    const Eigen::Vector3f hit_in_local = range_data_poses[i] * hit.head<3>();
    // 局部坐标系下由 hit 点到 origin 的射线
    const Eigen::Vector3f delta = hit_in_local - origin_in_local;
    const float range = delta.norm();//该向量的模
    if (range >= options_.min_range()) {
      if (range <= options_.max_range()) {//如果该向量的模在合理范围内，则把他压入
accumulated_range_data_点集中
        accumulated_range_data_.returns.push_back(hit_in_local);
      } else {
        accumulated_range_data_.misses.push_back(//否则，放入 misses 集中
            origin_in_local +
            options_.missing_data_ray_length() / range * delta);
      }
    }
  }
  ++num_accumulated_;//
  if (num_accumulated_ >= options_.num_accumulated_range_data()) {//点云帧数大于 1 的话
    num_accumulated_ = 0;
    const transform::Rigid3d gravity_alignment = transform::Rigid3d::Rotation(
        extrapolator_->EstimateGravityOrientation(time));//估计重力
```

```
    // TODO(gaschler): This assumes that 'range_data_poses.back()' is at time
    // 'time'.
    accumulated_range_data_.origin = range_data_poses.back().translation();
    return AddAccumulatedRangeData(//调用 AddAccumulatedRangeData 进行匹配、插入数据等。返回
MatchingResult
        time,
        TransformToGravityAlignedFrameAndFilter(//调用这个函数进行重力 align
            gravity_alignment.cast<float>() * range_data_poses.back().inverse(),
            accumulated_range_data_),
        gravity_alignment);
  }
  return nullptr;
}
// 添加累计的 RangeData，返回匹配结果。调用了 ScanMatch 和 InsertIntoSubmap
std::unique_ptr<LocalTrajectoryBuilder2D::MatchingResult>
LocalTrajectoryBuilder2D::AddAccumulatedRangeData(
    const common::Time time,
    const sensor::RangeData& gravity_aligned_range_data,
    const transform::Rigid3d& gravity_alignment) {
  // 如果数据为空
  if (gravity_aligned_range_data.returns.empty()) {
    LOG(WARNING) << "Dropped empty horizontal range data.";
```

```
    return nullptr;
  }
  // Computes a gravity aligned pose prediction.
  // 计算经过重力 aligned 的 Pose
  const transform::Rigid3d non_gravity_aligned_pose_prediction =
      extrapolator_->ExtrapolatePose(time);//从 PoseExtrapolator 处获得未经重力 aligned 的
Pose
  const transform::Rigid2d pose_prediction = transform::Project2D(
      non_gravity_aligned_pose_prediction * gravity_alignment.inverse());//重力 align
  // local map frame <- gravity-aligned frame
  // 调用 ScanMathc 函数进行匹配，求取 Scan 插入 Submap 的最优 Pose
  std::unique_ptr<transform::Rigid2d> pose_estimate_2d =
      ScanMatch(time, pose_prediction, gravity_aligned_range_data);
  if (pose_estimate_2d == nullptr) {//如果为空，表示匹配为空
    LOG(WARNING) << "Scan matching failed.";
    return nullptr;
  }
  // 考虑重力方向，将 2d 的 pose 变成 3d 的 pose
  const transform::Rigid3d pose_estimate =
      transform::Embed3D(*pose_estimate_2d) * gravity_alignment;
  // 将新的 pose 添加到 PoseExtrapolator 的 Pose 队列中
  extrapolator_->AddPose(time, pose_estimate);
```

```
  // 把点云数据通过估计出来的 Pose,转化为在局部坐标系中的点云数据
  sensor::RangeData range_data_in_local =
      TransformRangeData(gravity_aligned_range_data,
                         transform::Embed3D(pose_estimate_2d->cast<float>()));
  // 调用 InsertIntoSubmap 函数更新 Submap，同时返回插入结果
  std::unique_ptr<InsertionResult> insertion_result =
      InsertIntoSubmap(time, range_data_in_local, gravity_aligned_range_data,
                       pose_estimate, gravity_alignment.rotation());
  // 统计一下数据累计的时间
  auto duration = std::chrono::steady_clock::now() - accumulation_started_;
  kLocalSlamLatencyMetric->Set(
      std::chrono::duration_cast<std::chrono::seconds>(duration).count());
  // 返回匹配结果
  return common::make_unique<MatchingResult>(
      MatchingResult{time, pose_estimate, std::move(range_data_in_local),
                     std::move(insertion_result)});
}
// 插入 submap，返回插入结果。这是在都已经完成匹配的情况下，调用该函数更新 submap
std::unique_ptr<LocalTrajectoryBuilder2D::InsertionResult>
LocalTrajectoryBuilder2D::InsertIntoSubmap(
    const common::Time time, const sensor::RangeData& range_data_in_local,
    const sensor::RangeData& gravity_aligned_range_data,
```

```
  const transform::Rigid3d& pose_estimate,
  const Eigen::Quaterniond& gravity_alignment) {
//如果没有被 MotionFilter 滤掉
if (motion_filter_.IsSimilar(time, pose_estimate)) {
  return nullptr;
}
// Querying the active submaps must be done here before calling
// InsertRangeData() since the queried values are valid for next insertion.
// 把 active_submaps_中维护的 Submap 列表放到 insertion_submaps 这个向量中
std::vector<std::shared_ptr<const Submap2D>> insertion_submaps;
for (const std::shared_ptr<Submap2D>& submap : active_submaps_.submaps()) {
  insertion_submaps.push_back(submap);
}
// 调用 submap 的工具函数将传感器数据插入，更新 Submap。
active_submaps_.InsertRangeData(range_data_in_local);
// 生成一个体素化滤波器并滤波
sensor::AdaptiveVoxelFilter adaptive_voxel_filter(
    options_.loop_closure_adaptive_voxel_filter_options());
const sensor::PointCloud filtered_gravity_aligned_point_cloud =
    adaptive_voxel_filter.Filter(gravity_aligned_range_data.returns);
// 返回插入结果
return common::make_unique<InsertionResult>(InsertionResult{
```

```
      std::make_shared<const TrajectoryNode::Data>(TrajectoryNode::Data{
          time,
          gravity_alignment,
          filtered_gravity_aligned_point_cloud,
          {},  // 'high_resolution_point_cloud' is only used in 3D.
          {},  // 'low_resolution_point_cloud' is only used in 3D.
          {},  // 'rotational_scan_matcher_histogram' is only used in 3D.
          pose_estimate}),
      std::move(insertion_submaps)});
}

// 添加 IMU 数据，并对 PoseExtrapolator 进行带 IMU 的设置。将 IMU 数据压入队列
void LocalTrajectoryBuilder2D::AddImuData(const sensor::ImuData& imu_data) {
  CHECK(options_.use_imu_data()) << "An unexpected IMU packet was added.";
  InitializeExtrapolator(imu_data.time);
  extrapolator_->AddImuData(imu_data);
}

// 添加里程计数据，压入队列中
void LocalTrajectoryBuilder2D::AddOdometryData(
    const sensor::OdometryData& odometry_data) {
  if (extrapolator_ == nullptr) {
    // Until we've initialized the extrapolator we cannot add odometry data.
    LOG(INFO) << "Extrapolator not yet initialized.";
```

```
    return;
  }
  extrapolator_->AddOdometryData(odometry_data);
}
// 初始化 Extrapolator
void LocalTrajectoryBuilder2D::InitializeExtrapolator(const common::Time time) {
  if (extrapolator_ != nullptr) {
    return;
  }
  // We derive velocities from poses which are at least 1 ms apart for numerical
  // stability. Usually poses known to the extrapolator will be further apart
  // in time and thus the last two are used.
  // 该参数配置的是 PoseExtrapolator 中 Pose 队列的持续时间。设置为 1ms.
  constexpr double kExtrapolationEstimationTimeSec = 0.001;
  // TODO(gaschler): Consider using InitializeWithImu as 3D does.
  // 生成一个 PoseExtrapolator
  extrapolator_ = common::make_unique<PoseExtrapolator>(
      ::cartographer::common::FromSeconds(kExtrapolationEstimationTimeSec),
      options_.imu_gravity_time_constant());
  // 初始时刻，添加一个 Pose:I
  extrapolator_->AddPose(time, transform::Rigid3d::Identity());
}
```

第 13 章 Local Slam 的完结

在上两章中，我们已经完成了对 LocalTrajectoryBuilder2D 的阅读。可以看到，LocalTrajectoryBuilder2D 是进行 Local SLAM 的主体。在这个类当中，它初始化一个 PoseExtrapolator 类，用该类来管理 IMU 和里程计的数据队列，并维护和更新着机器人的位姿估计。在点云数据发布后，也是该类初始化体素滤波器和自适应体素滤波器，处理后的点云被用来进行 ScanMatching（由于时间关系，我们不再深入看 ScanMatching 的代码了），然后经过 MotionFilter 后插入并更新 Submap 信息。对比一下 Cartographer 源码阅读 1——整体框架介绍 中的图 1，我们看到算法流程图中的各部分都可以在该类找到其实现。

但是，还有两个问题没有解决：一是 LocalTrajectoryBuilder2D 是如何被调用的；二是所有的 Submap 的 List 是如何维护的（我们在 LocalTrajectoryBuilder2D 中只能看到在 ActiveSubmap 中维护着两个 submap：Old 和 New）。我们在这一篇文章中希望能解决掉第一个问题，从而，Local SLAM 的部分就可以完结了。下一篇文章我们开始讲 PoseGraph 是如何进行 Loop Closure 的。猜测，submap 的 list 也是在 PoseGraph 中维护的。

这一部分的代码非常难读，主要原因就在于彼此之间的继承关系、函数调用关系等都非常复杂。我在经过一天的代码阅读和整理之后，还是感觉云里雾里。但是我也没有时间进行更深入细致的研究了。所以接下来我会试着按我的思路进行一下简单的梳理，同时也把我的一些问题也列出来。有大神朋友愿意给我指点一下迷津的欢迎留言。可能会不可避免的存在诸多错误，请多见谅。

所幸虽然代码逻辑混乱，但这部分并不包含算法本身的思想。算法部分我们也已经在上两篇文章中进行了解读。所以，这篇文章我们也就看个大概吧。

让我们再回顾一下在 MapBuilder （请见 Cartographer 源码阅读 6——MapBuilder 的解读）中的工作：

在 cartographer_node 中，当 AddTrajectory 这个服务被请求时，程序最终是调用了 MapBuilder::AddTrajectoryBuilder 这个函数。该函数创建了一下 TrajectoryBuilderInterface （请见 Cartographer 源码阅读 8—用于 Local SLAM 的 TrajectoryBuilder 部分)的指针，然后把它压入了 trajectory_builders_这个存放 TrajectoryBuilderInterface 的向量中。回顾一下这段代码(只给出了 2D 情况)：

```
{//如果是 2d 的情况，跟 3d 情况基本相同，不再赘述

    std::unique_ptr<LocalTrajectoryBuilder2D> local_trajectory_builder;

    if (trajectory_options.has_trajectory_builder_2d_options()) {

      local_trajectory_builder = common::make_unique<LocalTrajectoryBuilder2D>(

          trajectory_options.trajectory_builder_2d_options(),
```

```
        SelectRangeSensorIds(expected_sensor_ids));
  }
  DCHECK(dynamic_cast<PoseGraph2D*>(pose_graph_.get()));
  trajectory_builders_.push_back(
      common::make_unique<CollatedTrajectoryBuilder>(
          sensor_collator_.get(), trajectory_id, expected_sensor_ids,
          CreateGlobalTrajectoryBuilder2D(

              std::move(local_trajectory_builder),trajectory_id,

              static_cast<PoseGraph2D*>(pose_graph_.get()),local_slam_result_callback))
  );
  //2d 的情况需要多处理的情况。
  //has_overlapping_submaps_trimmer_2d()不知道表示什么意思
  //猜测是说如果两个 submap 有重叠的部分，则调用 pose_graph_中的方法来进行全局优化
  //那 3d 的情况就不需要处理吗？
  if (trajectory_options.has_overlapping_submaps_trimmer_2d()) {
    const auto& trimmer_options = trajectory_options.overlapping_submaps_trimmer_2d();
    pose_graph_->AddTrimmer(common::make_unique<OverlappingSubmapsTrimmer2D>(
        trimmer_options.fresh_submaps_count(),trimmer_options.min_covered_area() /
        common::Pow2(trajectory_options.trajectory_builder_2d_options()
                                               .submaps_options()
                                               .grid_options_2d()
```

```
                                        .resolution()),
                        trimmer_options.min_added_submaps_count()));
    }
}
```

首先是生成了一个 LocalTrajectoryBuilder2D 型的变量 local_trajectory_builder，然后以该变量为参数调用了 CreateGlobalTrajectoryBuilder2D 函数(/mapping/internal/global_trajectory_builder.h 和.../.cc)创建了一个 GlobalTrajectoryBuilder 类。有意思的是 GlobalTrajectoryBuilder 也继承了 TrajectoryBuilderInterface，并根据不同的传感器类型实现了接口 TrajectoryBuilderInterface 中的 AddSensorData 函数。而在 AddSensorData 的实现中，也是通过调用 LocalTrajectoryBuilder2D 型的成员变量 local_trajectory_builder_和 PoseGraph 型的成员变量 pose_graph_进行处理。

最后，这个 GlobalTrajectoryBuilder 型的变量也传参给 wrapped_trajectory_builder，构造了 CollatedTrajectoryBuilder 这个类的智能指针。最后把 CollatedTrajectoryBuilder 的智能指针压入了 trajectory_builders_这个向量。

Question 1:
CollatedTrajectoryBuilder 和 GlobalTrajectoryBuilder 都并继承了 TrajectoryBuilderInterface 这个接口。那在 MapBuilder 中调用 TrajectoryBuilderInterface 中的 AddSensorData 这个函数时，具体调用的是 CollatedTrajectoryBuilder 还是 GlobalTrajectoryBuilder 的成员函数呢？

另外比较难懂的一点就是关于回调函数。

MapBuilderInterface 中通过宏定义声明了一个回调函数的函数指针：

```
  using LocalSlamResultCallback =
    std::function<void(int /* trajectory ID */,
                       common::Time,//时间
                       transform::Rigid3d /* local pose estimate */,
                       sensor::RangeData /* in local frame */,
                       std::unique_ptr<const InsertionResult>)>;
```

MapBuilder 中定义了一个这个回调函数的指针变量 local_slam_result_callback，然后该指针变量一步步赋值（MapBuilder-->CreatGlobalTrajectoryBuilder2D-->GlobalTrajectoryBuilder-

->CollatedTrajectoryBuilder），最后在 CollatedTrajectoryBuilder 的构造函数中将该指针指向了如下这样一个函数：

```
[this](const std::string& sensor_id, std::unique_ptr<sensor::Data> data) {
        HandleCollatedSensorData(sensor_id, std::move(data));
}
```

所以，这个回调函数的主要程序体就是调用了 CollatedTrajectoryBuilder 的成员函数 HandleCollatedSensorData.

```
void CollatedTrajectoryBuilder::HandleCollatedSensorData(
  const std::string& sensor_id, std::unique_ptr<sensor::Data> data) {
      auto it = rate_timers_.find(sensor_id);
  if (it == rate_timers_.end()) {
      it = rate_timers_
            .emplace(
                std::piecewise_construct, std::forward_as_tuple(sensor_id),
                std::forward_as_tuple(
                    common::FromSeconds(kSensorDataRatesLoggingPeriodSeconds)))
            .first;
  }
  it->second.Pulse(data->GetTime());
  if (std::chrono::steady_clock::now() - last_logging_time_ >
      common::FromSeconds(kSensorDataRatesLoggingPeriodSeconds)) {
    for (const auto& pair : rate_timers_) {
      LOG(INFO) << pair.first << " rate: " << pair.second.DebugString();
```

```
    }

    last_logging_time_ = std::chrono::steady_clock::now();

  }

  data->AddToTrajectoryBuilder(wrapped_trajectory_builder_.get());//所以，每次有数据传入时，就调用这个回调函数。该回调函数最终调用 LocalTrajectoryBuilder2D 来处理

}
```

HandleCollatedSensorData 中调用了 Data (/sensor/data.h)的 AddToTrajectoryBuilder 函数。继续跟踪，Data 是一个接口，被 LocalSlamResultData (/mapping/local_slam_result_data.h)继承，LocalSlamResultData 并没有实现 AddToTrajectoryBuilder 方法，而 LocalSlamResult2D (/sensor/internal/collator.h)继承了 LocalSlamResultData 并实现了 AddToTrajectoryBuilder. 在 LocalSlamResultData 的函数中有分别调用了 TrajectoryBuilderInterface 的成员函数和 PoseGraph2D 的成员函数。

```
void LocalSlamResult2D::AddToTrajectoryBuilder(
    TrajectoryBuilderInterface* const trajectory_builder) {
  trajectory_builder->AddLocalSlamResultData(
      common::make_unique<LocalSlamResult2D>(*this));
}

void LocalSlamResult2D::AddToPoseGraph(int trajectory_id,
                                       PoseGraph* pose_graph) const {
  DCHECK(dynamic_cast<PoseGraph2D*>(pose_graph));
  CHECK_GE(local_slam_result_data_.submaps().size(), 1);
  CHECK(local_slam_result_data_.submaps(0).has_submap_2d());
  std::vector<std::shared_ptr<const mapping::Submap2D>> submaps;
  for (const auto& submap_proto : local_slam_result_data_.submaps()) {
    submaps.push_back(submap_controller_->UpdateSubmap(submap_proto));
```

```
  }

  static_cast<PoseGraph2D*>(pose_graph)

      ->AddNode(std::make_shared<const mapping::TrajectoryNode::Data>(

                    mapping::FromProto(local_slam_result_data_.node_data())),

                trajectory_id, submaps);

}
```

所以，绕了一个大圈后，还是调用了 TrajectoryBuilder 和 PoseGraph 去做处理。

现在看一下回调该函数的地方。该回调函数分别在 SensorCollator（定义在 /sensor/collator_interface.h 中，分别被 TrajectoryCollator 和 Collator 继承）的 AddTrajectory 函数和 GlobalTrajectoryBuilder 的 AddSensorData(处理 TimedPointCloudData 的那个 AddSensorData)中被回调：

>在 TrajectoryCollator(/sensor/internal/trajectory_collator.h)被调用的代码：

```
void TrajectoryCollator::AddTrajectory(

    const int trajectory_id,

    const std::unordered_set<std::string>& expected_sensor_ids,

    const Callback& callback) {

  CHECK_EQ(trajectory_to_queue_.count(trajectory_id), 0);

  for (const auto& sensor_id : expected_sensor_ids) {

    const auto queue_key = QueueKey{trajectory_id, sensor_id};

    trajectory_to_queue_[trajectory_id].AddQueue(

        queue_key, [callback, sensor_id](std::unique_ptr<Data> data) {

          callback(sensor_id, std::move(data));//callback 即为被传入的回调函数

        });
```

```
    trajectory_to_queue_keys_[trajectory_id].push_back(queue_key);
  }
}
```

>在 Collator (/sensor/internal/collator.h)中被调用的代码：

```
void Collator::AddTrajectory(
    const int trajectory_id,
    const std::unordered_set<std::string>& expected_sensor_ids,
    const Callback& callback) {
  for (const auto& sensor_id : expected_sensor_ids) {
    const auto queue_key = QueueKey{trajectory_id, sensor_id};
    queue_.AddQueue(queue_key,
                    [callback, sensor_id](std::unique_ptr<Data> data) {
                      callback(sensor_id, std::move(data));//callback 即为被传入的回调函数
                    });
    queue_keys_[trajectory_id].push_back(queue_key);
  }
}
```

>GlobalTrajectoryBuilder 中被调用的代码：

```
  void AddSensorData(
      const std::string& sensor_id,
      const sensor::TimedPointCloudData& timed_point_cloud_data) override {
    CHECK(local_trajectory_builder_)
```

```
    << "Cannot add TimedPointCloudData without a LocalTrajectoryBuilder.";
std::unique_ptr<typename LocalTrajectoryBuilder::MatchingResult>
    matching_result = local_trajectory_builder_->AddRangeData(
        sensor_id, timed_point_cloud_data);
if (matching_result == nullptr) {
  // The range data has not been fully accumulated yet.
  return;
}
kLocalSlamMatchingResults->Increment();
std::unique_ptr<InsertionResult> insertion_result;
if (matching_result->insertion_result != nullptr) {
  kLocalSlamInsertionResults->Increment();
  auto node_id = pose_graph_->AddNode(
      matching_result->insertion_result->constant_data, trajectory_id_,
      matching_result->insertion_result->insertion_submaps);
  CHECK_EQ(node_id.trajectory_id, trajectory_id_);
  insertion_result = common::make_unique<InsertionResult>(InsertionResult{
      node_id, matching_result->insertion_result->constant_data,
      std::vector<std::shared_ptr<const Submap>>(
          matching_result->insertion_result->insertion_submaps.begin(),
          matching_result->insertion_result->insertion_submaps.end())});
}
```

```
    if (local_slam_result_callback_) {
      local_slam_result_callback_(//被传入的回调函数
          trajectory_id_, matching_result->time, matching_result->local_pose,
          std::move(matching_result->range_data_in_local),
          std::move(insertion_result));
    }
  }
```

Question2：
回调函数这里为什么这么设计？绕了一大圈，其中的逻辑关系有点稀里糊涂了

我们不妨再从开头回顾一下整个处理过程：

13.1、当刚初始化一个 TrajectoryBuilder 时：

初始时，用处发出 AddTrajectory 的服务请求-->cartographer_node 调用 MapBuilderBridge::AddTrajectory 函数，同时订阅传感器的 Topic-->该函数调用 MapBuilder::AddTrajectory-->该函数中生成一个 LocalTrajectoryBuilder2D,然后通过 CreateGlobalTrajectoryBuilder 把 LocalTrajectoryBuilder2D 的参数和回调函数传给 GlobalTrajectoryBuilder-->以 GlobalTrajectoryBuilder 为参数构造了 CollatedTrajectoryBuilder，并给 CollatorInterface 中的 AddTrajectory 函数注册回调函数的实际地址。

13.2、在初始化完成后，每次传感器往 Topic 上广播 Message 后：

传感器的 ROS 节点/Playbag 广播到 Topic 上相关数据的 Message---->cartographer_node 中启动的 StartTrajectory 这个服务会订阅传感器数据---->接收到该数据由相应的处理函数处理，比如 Node::HandleImuMessage---->该处理函数实际调用是 MapBuilderBridge 中的一个 SensorBridge 变量进行处理---->调用了 TrajectoryBuilder 的虚函数 AddSensorData()---->CollatedTrajectoryBuilder 或 GlobalTrajectoryBuilder 继承 TrajectoryBuilder 并具体实现 AddSensorData()函数-->这些函数通过 LocalTrajectoryBuilder2D 将传感器数据压入相应的数据队列-->LocalTrajectoryBuilder2D 通过 AddRangeData 等函数处理传感器数据

但是，问题来了：

Question 3:
上面这个流程中有传感器数据来临时的做法都是调用了 AddSensorData()函数把数据压入了数据队列，那么程序是在什么地方调用 AddRangeData 来处理队列中的数据的呢？

对于这个问题，我突然有了解答：

对于 IMU 数据和里程计数据都是压入数据队列后被动地等待 AddRangeData 函数调用 PoseExtrapolator 来做处理，但当 RangeData 数据发布时，最后在处理 RangeData 的 AddSensorData 中，并不是把数据压入队列，而是在这其中调用了 LocalTrajectoryBuilder2D 的 AddRangeData 函数处理。所以，cartographer 算法是以激光、点云数据为核心的，IMU、里程计等数据都是起到一定的辅助作用。这就是整个程序的逻辑思路。我们可以具体看一下 GlobalTrajectoryBuilder 中关于处理点云数据的 AddSensorData:

```
  void AddSensorData(
      const std::string& sensor_id,
      const sensor::TimedPointCloudData& timed_point_cloud_data) override {
    CHECK(local_trajectory_builder_)
        << "Cannot add TimedPointCloudData without a LocalTrajectoryBuilder.";
    std::unique_ptr<typename LocalTrajectoryBuilder::MatchingResult>
        matching_result = local_trajectory_builder_->AddRangeData(
            sensor_id, timed_point_cloud_data);//调用 LocalTrajectoryBuilder2D 的 AddRangeData 处理点云处理，并更新地图
    if (matching_result == nullptr) {
      // The range data has not been fully accumulated yet.
      return;
    }
    kLocalSlamMatchingResults->Increment();
    std::unique_ptr<InsertionResult> insertion_result;
    if (matching_result->insertion_result != nullptr) {
```

```
    kLocalSlamInsertionResults->Increment();
    auto node_id = pose_graph_->AddNode(
        matching_result->insertion_result->constant_data, trajectory_id_,
        matching_result->insertion_result->insertion_submaps);
    CHECK_EQ(node_id.trajectory_id, trajectory_id_);
    insertion_result = common::make_unique<InsertionResult>(InsertionResult{
        node_id, matching_result->insertion_result->constant_data,
        std::vector<std::shared_ptr<const Submap>>(
            matching_result->insertion_result->insertion_submaps.begin(),
            matching_result->insertion_result->insertion_submaps.end())});
  }
  if (local_slam_result_callback_) {//处理完的结果调用回调函数进行处理
    local_slam_result_callback_(
        trajectory_id_, matching_result->time, matching_result->local_pose,
        std::move(matching_result->range_data_in_local),
        std::move(insertion_result));
  }
}
```

好了，到此，我们对于 Local SLAM 部分的解读就到这里。其中对于两种匹配算法我没有展开讲，有时间再专门写一篇文章来将理论部分。另外就是还有一些没有想明白的问题就等到以后有时间了再深入了解吧。下一篇文章我们开始讲用于 Loop Closure 的 PoseGraph。

敬请期待

第 14 章 PoseGraph for Global SLAM

我们已经结束了 Local SLAM 部分的代码阅读。接下来我们要顺一下 Global SLAM 这部分的代码逻辑。Global SLAM 的代码是由 PoseGraph 来完成的。

14.1、优化问题

在开始我们的代码阅读之前，我决定还是先从理论上分析，当我们在谈到 Loop Closure 时，我们到底在谈论什么。顺便，我们可以把 cartographer 中提到的节点(Node，注意这里的 Node 与 ROS 中的 Node 不是同一个概念)、约束（Constraints）等概念进行一下简单解释。这些内容对 SLAM 大神应该已经是小儿科的东西了，但谁让我是小白呢。哈哈，花点时间整理一下吧。

回想一下，我们在 Local Slam 那一部分已经可过通过一帧帧的 Laser Scan 建立起了一个个的 Submap，我们也记录了每一帧 Scan 插入到了 Submap 中的 Pose。如果我们的测量都是非常精确、没有任何误差的话，那么我们的工作已经做完了。我们已经可以采用这种逐步扫描的方法建立起一个精确的地图，完成对一个区域的建图。之后的定位和导航也不过是用新的 Scan 与我们已经建立的地图做一个新的 Scan Matching。但是，很遗憾，理想不等于现实。我们的传感器是存在误差的，虽然在局部范围内误差并不大，但是由于建图的过程是一个增量式的过程，第 n 个 Pose 的估计依赖于对前 n-1 个 Pose 的估计，那么之前每个 Pose 的误差都会累积到最新的 Pose 上，那么当我们的 trajectory 增长到一定的地步，我们对机器人自身 Pose 的估计都已经十分不准确了，那么这时候以自身 Pose 为基准建立的 Map 更不可能准确了。所以，我们需要 Loop Closure 来进行回环检测，来消除这种累积误差。

那么怎么做回环检测呢？

前面我们已经提到过，传感器的测量误差是累积式地增长，但是对于在一个局部范围，这种误差是在我们的可接受范围内的(不然的话那我们的传感器也太不可靠了，根本没法使用，是吧。)，所以，我们不信任的是我们估计出来的一个 Scan 在全局地图中的绝对位姿，但是，在某个位置处，我们测量出来的该位姿与上一时刻之间的相对位姿，以及该 Scan 与我们建立的局部 Submap 之间的相对位姿是比较可信的。那么，我们把估计出来的一个 Scan 的绝对位姿称为 trajectory 上的一个节点(Node)，那么节点与节点的彼此之间的相对位姿就可以称为一个约束(Constraint)。我们的优化目标也就是优化所有 Scan 的位姿，使彼此之间的相对位姿具有一致性。

那么，在 cartographer 中并没有采用节点与节点之间的相对位姿。由于 cartographer 中维护了一系列的 Submap，而 Submap 在一定范围可以视为较为准确的。所以，cartographer 中把一个 scan 相对于一个 Submap 的相对位姿作为一个约束。

我们标记：

：第 i 个 Submap 相对于世界坐标系的位姿，其中上角标　　表示这是 Submap 的位姿，　　；

：第 j 个 Scan 相对于世界坐标系的位姿，其中上角标　　表示这是 Scan 的位姿，　　；

：第 j 个 Scan 插入第 i 个 Submap 的相对位姿。

：第 j 个 Scan 插入第 i 个 Submap 的质量。可以理解为对 ScanMatching 效果的一个评分值。

那么，我们可以把　　看做是约束(Constraints)，我们的总体目标就是优化　　和　　这两类量，使总体的误差最小。其中，误差函数的定义过程如下：

首先，考虑其中一对儿约束，对于　、　　和　　，定义误差为：

其中，　　和　　分别表示了 Submap 和 Scan 位姿中的 Translation 的部分，则　　表示了 Scan 在 Submap 中的相对位置在世界坐标系下的表示；　　则表示了 Submap 的局部坐标系在世界坐标系下的表示，通过求逆得到的　　则是世界坐标系相对于 Submap 局部坐标系的姿态变换矩阵。因此，　　就表示了 Scan 在 Submap 中的相对位姿。　和　　则分别表示了 Scan 和 Submap 的姿态角，对于 2D 情况，该量是一个标量。

所以，误差定义式中　　则是通过 Scan 和 Submap 的绝对位姿求出来的他们之间的相对位姿，而　　则是通过传感器测量出来的相对位姿，前面我们已经提到过，相对位姿可以视为一个约束。

若我们的绝对位姿是正确无误差的，那么这两项应该相一致，误差应该为 0。但是，由于我们对绝对位姿的估计由于时间累积存在了较大的误差，所以，我们做 Loop Closure 的主要目标就是优化　和　使误差值尽可能性地小。

上述是对于单独一对儿约束定出来的误差，而对于整体的误差就是对上述误差的二次型对所有约束对儿的一个加和：

其中　是一个 loss function，用来避免异常点导致的错误匹配的约束带来的影响。　在 cartographer 选用的是 Huber Loss(请见 Huber Loss function。不太懂这是啥，可以以后再说)。

所以，Loop Closure 可转化为一个优化问题：

其中　　代表了所有 Submap 的绝对位姿，　　表示了所有 Scan 的绝对位姿。

那么，这个优化问题在 cartographer 是通过 Sparse Pose Adjustment 来解决的。至于 Sparse Pose Adjustment 是如何高效率地解决这个优化问题，有兴趣的同学可以去读一下下面这篇论文。由于时间关系我还没有深入细致地看论文，粗略地扫了一遍，目前的感觉就是利用了雅克比矩阵的稀疏性来去除大量不需要计算的情况，提高计算的实时性。这块儿等我以后了解的更多了再来更新。

https://www.researchgate.net/publication/221066802_Efficient_Sparse_Pose_Adjustment_for_2D_mapping

所以，可以想见，我们在 PoseGraph 中期望看到就应该是 Node 和 Constraint 是如何维护的，什么情况下会利用这些约束来进行优化等等问题。

14.2、PoseGraphInterface 源码阅读

我们首先回顾一下在 MapBuilder （第 6 章 MapBuilder 的解读)中关于 PoseGraph 的一些操作。

PoseGraph 是用于 Loop Closure 的。对于同一个 Map，只需要有一个全局的 PoseGraph 来维护即可。所以，在 MapBuilder 的成员变量中，定义了一个 PoseGraph 类型的智能指针：

std::unique_ptr<PoseGraph> pose_graph_; 一个 PoseGraph 的智能指针

PoseGraph 只在 MapBuilder 的构造函数中初始化一次。根据 2d 和 3d 情况的不同，PoseGraph 这个接口类分别由 PoseGraph2D 和 PoseGraph3D 两种实现方式：

```
MapBuilder::MapBuilder(const proto::MapBuilderOptions& options)
    : options_(options), thread_pool_(options.num_background_threads()) {
  CHECK(options.use_trajectory_builder_2d() ^ options.use_trajectory_builder_3d());
  if (options.use_trajectory_builder_2d()) {
    pose_graph_ = common::make_unique<PoseGraph2D>(
        options_.pose_graph_options(),
        common::make_unique<optimization::OptimizationProblem2D>(

            options_.pose_graph_options().optimization_problem_options()),
        &thread_pool_);//一个 PoseGraphD 的指针。
  }
  if (options.use_trajectory_builder_3d()) {
    pose_graph_ = common::make_unique<PoseGraph3D>(
        options_.pose_graph_options(),
```

```
            common::make_unique<optimization::OptimizationProblem3D>(

                options_.pose_graph_options().optimization_problem_options()),

            &thread_pool_);

  }

  if (options.collate_by_trajectory()) {//根据 collate_by_trajectory 的不同，
CollatorInterface 有两种不同的实现

    sensor_collator_ = common::make_unique<sensor::TrajectoryCollator>();

  } else {

    sensor_collator_ = common::make_unique<sensor::Collator>();

  }

}
```

作者是先定义了一个 PoseGraphInterface (/mapping/pose_graph_interface.h 中)，然后又定义了 PoseGraph (/mapping/pose_graph.h)来继承 PoseGraphInterface，但是 PoseGraph 里依然定义了很多虚函数，分别针对 2D 和 3D 的情况，由 PoseGraph2D (/mapping/internal/2d/pose_graph_2d.h)和 PoseGraph3D (/mapping/internal/3d/pose_graph_3d.h)来继承并实现。所以我们从 PoseGraphInterface 入手，一点点深入。

我们首先看一下在 PoseGraphInterface 中定义的一些数据结构：

>1. 一对儿约束的数据结构：

```
struct Constraint {

  struct Pose {

    transform::Rigid3d zbar_ij;//相对位姿

    double translation_weight;//translation 的权重

    double rotation_weight;//rotation 的权重

  };
```

```
  SubmapId submap_id;  // 'i' in the paper. Submap 的 index

  NodeId node_id;      // 'j' in the paper. TrajectoryNode 的 index

  // Pose of the node 'j' relative to submap 'i'.

  Pose pose;//节点 j 相对于 Submap i 的相对位姿

  // Differentiates between intra-submap (where node 'j' was inserted into

  // submap 'i') and inter-submap constraints (where node 'j' was not inserted

  // into submap 'i').

  // 每一对儿 node 和 submap，都分为两种情况：节点 j 有插入该 submap 中(INTRA_SUBMAP)和没有插入该 submap 中(INTER_SUBMAP)。

    enum Tag { INTRA_SUBMAP, INTER_SUBMAP } tag;

 };
```

包含：SubmapId (包括 trajectory_id 和 submap 的 index)、NodeId(包括 trajectory_id 和 node 的 index，详见：Cartographer 源码阅读 8—用于 Local SLAM 的 TrajectoryBuilder 部分 1.1.1)-<1>NodeId)、相对 Pose(不同的是，translation 和 rotation 各自分配了一个权重。类似于协方差矩阵吧)、Tag 变量(表明节点有没有插入 submap)。

>2. LandmarkNode

Landmark 相当于路标，每个 Node 与 Landmark 之间的相对位姿也应该加入到约束之中。

```
struct LandmarkNode {

  struct LandmarkObservation {

    int trajectory_id;

    common::Time time;

    transform::Rigid3d landmark_to_tracking_transform;

    double translation_weight;

    double rotation_weight;
```

```
    };

    std::vector<LandmarkObservation> landmark_observations;

    common::optional<transform::Rigid3d> global_landmark_pose;

};
```

在 cartographer 中应该是认为 Landmark 的可靠性非常高的。

>3. Submap 相关

```
struct SubmapPose {

  int version;//version 不知道是表征啥的

  transform::Rigid3d pose;//submap 的绝对位姿 \xi^m_i

};



struct SubmapData {

  std::shared_ptr<const Submap> submap;//栅格概率图数据

  transform::Rigid3d pose;//submap 的绝对位姿 \xi^m_i

};
```

>4. TrajectoryData 不知道是干啥的

```
struct TrajectoryData {

  double gravity_constant = 9.8;

  std::array<double, 4> imu_calibration{{1., 0., 0., 0.}};

  common::optional<transform::Rigid3d> fixed_frame_origin_in_map;

};
```

>5. 全局优化的回调函数的宏定义：

```
using GlobalSlamOptimizationCallback =

    std::function<void(const std::map<int /* trajectory_id */, SubmapId>&,

    const std::map<int /* trajectory_id */, NodeId>&)>;
```

满足一定条件则会调用该回调函数进行优化。可以看到，该函数传入的参数是一系列的 submap 的 id 和 Node 的 id。

>6. 成员函数

以下是 PoseGraphInterface 的一些成员函数，看名字也能大致猜到其功能，这里不再赘述，等我们读到他们的具体实现时再说。

```
PoseGraphInterface() {}

virtual ~PoseGraphInterface() {}

PoseGraphInterface(const PoseGraphInterface&) = delete;

PoseGraphInterface& operator=(const PoseGraphInterface&) = delete;

// Waits for all computations to finish and computes optimized poses.

// 最后的全局优化

virtual void RunFinalOptimization() = 0;

// Returns data for all submaps.

// 返回所有 Submap 的数据。

virtual MapById<SubmapId, SubmapData> GetAllSubmapData() const = 0;

// Returns the global poses for all submaps.

// 返回所有 submap 的 pose

virtual MapById<SubmapId, SubmapPose> GetAllSubmapPoses() const = 0;

// Returns the transform converting data in the local map frame (i.e. the

// continuous, non-loop-closed frame) into the global map frame (i.e. the
```

```
// discontinuous, loop-closed frame).
// 获取由局部坐标系到世界坐标系的变换矩阵
virtual transform::Rigid3d GetLocalToGlobalTransform(
  int trajectory_id) const = 0;
  // Returns the current optimized trajectories.
  // 返回当前经过优化后的 trajectory 上的所有 Node。这些 Node 构成了这条 trajectory
virtual MapById<NodeId, TrajectoryNode> GetTrajectoryNodes() const = 0;
// Returns the current optimized trajectory poses.
// 返回当前经过优化后的所有的节点 Pose
virtual MapById<NodeId, TrajectoryNodePose> GetTrajectoryNodePoses() const = 0;
// Returns the current optimized landmark poses.
// 返回 Landmark 的 Pose
virtual std::map<std::string, transform::Rigid3d> GetLandmarkPoses() const = 0;
// Sets global pose of landmark 'landmark_id' to given 'global_pose'.
// 设置某个 LandMark 的 Pose
virtual void SetLandmarkPose(const std::string& landmark_id,
    const transform::Rigid3d& global_pose) = 0;
// Checks if the given trajectory is finished.
// 判断一条 trajectory 是否被 finished
virtual bool IsTrajectoryFinished(int trajectory_id) const = 0;
// Checks if the given trajectory is frozen.
// 判断一条 trajectory 是否被 frozen
```

```
virtual bool IsTrajectoryFrozen(int trajectory_id) const = 0;

// Returns the trajectory data.

// 返回 TrajectoryData

virtual std::map<int, TrajectoryData> GetTrajectoryData() const = 0;

// Returns the collection of constraints.

// 返回所有的约束

virtual std::vector<Constraint> constraints() const = 0;

// Serializes the constraints and trajectories.

// 序列化约束和 trajectory

virtual proto::PoseGraph ToProto() const = 0;

// Sets the callback function that is invoked whenever the global optimization

 // problem is solved.

// 设置回调函数

virtual void SetGlobalSlamOptimizationCallback(

    GlobalSlamOptimizationCallback callback) = 0;
```

14.3、PoseGraph 源码阅读

/mapping/pose_graph.h 中也是定义了一堆新的虚函数。我们不再具体去看这些函数定义了，而等到 /mapping/internal/2d/pose_graph_2d.cc 中再去看其具体实现。这里我们只关注一下其中新定义的一些数据结构。

/mapping/pose_graph.h 又定义了一个 InitialTrajectoryPose:

```
struct InitialTrajectoryPose {

  int to_trajectory_id;//应该关联到那个 trajectory_id
```

```
  transform::Rigid3d relative_pose;//相对于谁的 Pose 呢？世界坐标系？这个 Pose 是表示的
Trajectory 的第一个 Node 的 Pose 吗？

  common::Time time;//时间

};
```

这里有个问题：InitialTrajectoryPose 中定义的 relative_pose 是相对于谁的 Pose 呢？世界坐标系？这个 Pose 是表示的 Trajectory 的第一个 Node 的 Pose 吗？

此外，/mapping/pose_graph.h 中还定义了几个辅助性的函数，跟主体功能没什么关系，不再赘述。只列在这里：

```
// 创建配置项

proto::PoseGraphOptions CreatePoseGraphOptions(

  common::LuaParameterDictionary* const parameter_dictionary);

// 序列化与反序列话

std::vector<PoseGraph::Constraint> FromProto(

    const ::google::protobuf::RepeatedPtrField<

        ::cartographer::mapping::proto::PoseGraph::Constraint>&constraint_protos);

proto::PoseGraph::Constraint ToProto(const PoseGraph::Constraint& constraint);
```

14.4、PoseGraph2D 源码阅读

/mapping/internal/2d/pose_graph_2d.h 中新增的数据结构：

```
// 一个枚举类，表征一个 Submap 的状态：kActive 或 kFinished。

// 当一个 submap 由 Active 转为 kFinished 时，所有的 Nodes 都要跟该 submap 做一个匹配。

// 同样的，当在 trajectory 增长的过程中又有新增的 Nodes，这些新的 Nodes 也需要跟所有已经
finished 的 submap 做一下 match

enum class SubmapState { kActive, kFinished };
```

```
// 存储与一个 Submap 相关联的 Node

struct InternalSubmapData {

  std::shared_ptr<const Submap2D> submap;//一个 Submap

  // IDs of the nodes that were inserted into this map together with

  // constraints for them. They are not to be matched again when this submap

  // becomes 'finished'.

  // 所有跟该 submap 有插入关系并存在约束的节点的 ID 的集合。当这个 submap 被 finished 时他们

  // 就不用重新跟该 submap 进行 match，而只需 match 新的 node 并把他们加入相应 submap 的这个集合
中

  std::set<NodeId> node_ids;

  //默认情况，submap 是 kActive，除非满足一定条件后把它置为 kFinished

  SubmapState state = SubmapState::kActive;

};
```

接下来让我们看一下 PoseGraph2D 的成员变量：

```
const proto::PoseGraphOptions options_;//参数配置项

GlobalSlamOptimizationCallback global_slam_optimization_callback_;//全局优化回调函数

mutable common::Mutex mutex_;//互斥锁

// If it exists, further work items must be added to this queue, and will be

// considered later.

// 工作队列。将要进行的工作都加入到该队列中进行管理。

std::unique_ptr<std::deque<std::function<void()>>> work_queue_GUARDED_BY(mutex_);

// How our various trajectories are related.
```

```
// 不同的 trajectory 之间的联系状态

TrajectoryConnectivityState trajectory_connectivity_state_;

// We globally localize a fraction of the nodes from each trajectory.

// 对每条 trajectory，全局定位一小部分 Nodes？？？没太懂要干嘛

// 但是 FixedFrame 是跟 GPS 信号相关的；所以猜测这个应该是 GPS 信号的采样器。

 // 一条轨迹会有一个 GPS 信号采样器，负责处理 GPS 数据

std::unordered_map<int, std::unique_ptr<common::FixedRatioSampler>>

global_localization_samplers_ GUARDED_BY(mutex_);

// Number of nodes added since last loop closure.

// 标记上次 Loop Closure 之后又新添加了多少 Nodes

int num_nodes_since_last_loop_closure_ GUARDED_BY(mutex_) = 0;

// Whether the optimization has to be run before more data is added.

// 在添加更多数据前是否必须要运行 optimization

bool run_loop_closure_ GUARDED_BY(mutex_) = false;

// Schedules optimization (i.e. loop closure) to run.

void DispatchOptimization() REQUIRES(mutex_);

// Current optimization problem.

// 当前优化问题

std::unique_ptr<optimization::OptimizationProblem2D> optimization_problem_;

// 创建约束的 Builder

constraints::ConstraintBuilder2D constraint_builder_ GUARDED_BY(mutex_);

// 创建起来的约束
```

```
std::vector<Constraint> constraints_ GUARDED_BY(mutex_);

// Submaps get assigned an ID and state as soon as they are seen, even

// before they take part in the background computations.

// 一个 submap 一旦建立，就会立刻分配一个 ID 和状态{kActive, kFinished}。submap_data_就是存储一个 submap 相关信息

// MapById 定义在/mapping/id.h 中，是一个工具类，方便管理

MapById<SubmapId, InternalSubmapData> submap_data_ GUARDED_BY(mutex_);

// Data that are currently being shown.

// 当前 trajectory 的一些节点数据

MapById<NodeId, TrajectoryNode> trajectory_nodes_ GUARDED_BY(mutex_);

//节点数

int num_trajectory_nodes_ GUARDED_BY(mutex_) = 0;

// Global submap poses currently used for displaying data.

// 全局的 submap 的 Pose. 会用于在 rviz 中显示建图效果

MapById<SubmapId, optimization::SubmapSpec2D> global_submap_poses_GUARDED_BY(mutex_);

// Global landmark poses with all observations.

// 存储所有 landmark 的 Id 及他们的观测数据

std::map<std::string /* landmark ID */, PoseGraph::LandmarkNode>landmark_nodes_
GUARDED_BY(mutex_);

// List of all trimmers to consult when optimizations finish.

// 列出来所有的 trimmer,但我还不清楚 trimmer 的概念是什么。

std::vector<std::unique_ptr<PoseGraphTrimmer>> trimmers_ GUARDED_BY(mutex_);

// Set of all frozen trajectories not being optimized.
```

```
// 被 frozen 但还没经过优化的 trajectory

std::set<int> frozen_trajectories_ GUARDED_BY(mutex_);

// Set of all finished trajectories.

// 已经 finished 的 trajectory 的集合

std::set<int> finished_trajectories_ GUARDED_BY(mutex_);

// Set of all initial trajectory poses.

// 存储指定 trajectory_id 应该关联的 trajectory_id

std::map<int, InitialTrajectoryPose> initial_trajectory_poses_GUARDED_BY(mutex_);
```

在这里我要对 MapById 做一个解释：
首先，MapById 中的 Map 跟 cartographer 中建图的 map 和 submap 的概念没有任何关系。这里的 MapById 只是 cartographer 对 C++ std::map 的一个封装实现。它只是一个容器类。
std::map(std:map)中，元素是以一个 key value 和一个 mapped value 的组和形式以特定顺序存储的。其中 key values 通常用来排序以及唯一地确定一个元素，而 mapped values 存储的是跟该 key value 相对应的内容。key value 和 mapped value 的数据格式可以完全不同。
而 cartographer 中的 MapById 也是这样一个容器，但其中 key value 限制为了 IdType 型，作者希望地就是以 NodeId 或 SubmapId 为索引来存储元素。其中 MapById 中的一个成员函数：

```
std::map<int, MapByIndex> trajectories_;
```

也不是特指 cartographer 中的一条 trajectory 的概念，而是一个抽象的数据链的概念。这个数据链可以是 TrajectoryNode, 也可以是 Submap 等等。

此外，PoseGraph2D 中还定义了一个 TrimmingHandle 的类，用来查询和 manipulating 一个 PoseGraph。我还不懂 Trimming 是什么含义。先放到这里，方便以后备查：

```
// Allows querying and manipulating the pose graph by the 'trimmers_'. The

// 'mutex_' of the pose graph is held while this class is used.

class TrimmingHandle : public Trimmable {

public:

  TrimmingHandle(PoseGraph2D* parent);
```

```
  ~TrimmingHandle() override {}

  int num_submaps(int trajectory_id) const override;

  std::vector<SubmapId> GetSubmapIds(int trajectory_id) const override;

  MapById<SubmapId, SubmapData> GetOptimizedSubmapData() const override

      REQUIRES(parent_->mutex_);

  const MapById<NodeId, TrajectoryNode>& GetTrajectoryNodes() const override

      REQUIRES(parent_->mutex_);

  const std::vector<Constraint>& GetConstraints() const override

    REQUIRES(parent_->mutex_);

  void MarkSubmapAsTrimmed(const SubmapId& submap_id)

    REQUIRES(parent_->mutex_) override;

  bool IsFinished(int trajectory_id) const override REQUIRES(parent_->mutex_);

private:

  PoseGraph2D* const parent_;

};
```

我们已经查看过了 PoseGraph2D 中定义的数据结构及其成员变量。下篇文章我们将开始查看其成员函数的实现，并以此梳理 PoseGraph 进行 Loop Closure 的逻辑。其中，我们会重点关注一下 Landmark 是如何使用的。

第 15 章 PoseGraph 的成员函数

本节我们查看 PoseGraph2D 的成员函数并梳理 PoseGraph 进行 Loop Closure 的逻辑。同时，我们要重点关注一下 Landmark 是如何应用的。

在 PoseGraph2D 中，有三个成员变量比较重要：

```
// 不同的 trajectory 之间的联系状态

TrajectoryConnectivityState trajectory_connectivity_state_;

// 当前优化问题

std::unique_ptr<optimization::OptimizationProblem2D> optimization_problem_;

// 创建约束的 Builder

constraints::ConstraintBuilder2D constraint_builder_ GUARDED_BY(mutex_);
```

其中，TrajectoryConnectivityState 定义在 /mapping/internal/trajectory_connectivity_state.h 中，用来维护不同 trajectory 之间的连接状态。OptimizationProblem2D 定义在/mapping/internal/optimization/optimization_problem_2d.h 中，继承了 OptimizationProblemInterface (定义在 /mapping/internal/optimization/optimization_problem_interface.h 中)这个接口。是用来解优化问题的。ConstraintBuilder2D 定义在/mapping/internal/constraints/constraint_builder_2d.h 中，继承了 ConstraintBuilder (定义在/mapping/internal/constraints/constraint_builder.h 中)，是用来创建约束 constraints 的。可以想见，PoseGraph2D 的成员函数也是通过调用这些类的成员函数来实现相应功能的。但这里我们先不深入展开，只需要了解一下概要信息。

对于 TrajectoryConnectivityState，我们只需要了解其中定义的主要成员函数和各函数功能：

```
// A class that tracks the connectivity state between trajectories. Compared to

// ConnectedComponents it tracks additionally the last time that a global

// constraint connected to trajectories.

//

// This class is thread-compatible.

class TrajectoryConnectivityState {
```

```
public:
  TrajectoryConnectivityState() {}
  TrajectoryConnectivityState(const TrajectoryConnectivityState&) = delete;
  TrajectoryConnectivityState& operator=(const TrajectoryConnectivityState&) = delete;
  // Add a trajectory which is initially connected to only itself.
  // 添加一条 trajectory
  void Add(int trajectory_id);
// 把两个 trajectories 产生链接。如果其中任意一个 trajectory 之前是 untracked，它将变成 tracked
  // 该函数跟参数顺序无关。重复调用 Connect 会增加两个 trajectory 之间的 connectivity count
  // 并更新 last connected time
  void Connect(int trajectory_id_a, int trajectory_id_b, common::Time time);
  // 判断两个 trajectories 是否已经 connected.
  // 如果其中任意一个 trajectory 没有被 tracked，返回 false——除非两个 trajectory 是同一个
trajectory 是返回 true
  // 同样，对参数顺序无关
  bool TransitivelyConnected(int trajectory_id_a, int trajectory_id_b) const;
  // The trajectory IDs, grouped by connectivity.
  // 返回一个 int 型的向量，这个向量里存的是具有 connectivity 关系的 trajectory 的 id.
  std::vector<std::vector<int>> Components() const;
  // 返回两条 trajectory 的上次链接时间。
  // 如果其中任意个 trajectory 是 untracked 或他们之间没有 connected, 则返回开始时的时间
  common::Time LastConnectionTime(int trajectory_id_a, int trajectory_id_b);
```

```
private:
  // ConnectedComponents are thread safe.
  // 存储所有的连接关系
  mutable ConnectedComponents connected_components_;
  // 返回任意两个指定 id 的 trajectory 的上次连接时间
  std::map<std::pair<int, int>, common::Time> last_connection_time_map_;
};
```

其中，所有的连接关系都存储在 connected_components_这个成员函数中。除此之外还增加了记录 last connection time 的成员函数 last_connection_time_map_.

对于 OptimizationProblem2D, 我们重点关注一下它的几个成员函数：

```
optimization::proto::OptimizationProblemOptions options_;//配置项
MapById<NodeId, NodeSpec2D> node_data_;//所有节点的相关信息：时间、local pose、 global pose、gravity alignment 等
MapById<SubmapId, SubmapSpec2D> submap_data_;//存储 submap 的 global pose
std::map<std::string, transform::Rigid3d> landmark_data_;//存储 landmark 的 pose
sensor::MapByTime<sensor::ImuData> imu_data_;//IMU 数据
sensor::MapByTime<sensor::OdometryData> odometry_data_;//里程计数据
```

其中，NodeSpec2D 和 SubmapSpec2D 分别定义如下：

```
struct NodeSpec2D {
  common::Time time;
  transform::Rigid2d local_pose_2d;
  transform::Rigid2d global_pose_2d;
  Eigen::Quaterniond gravity_alignment;
```

```
};

struct SubmapSpec2D {

  transform::Rigid2d global_pose;

  };
```

对于 ConstraintBuilder2D，我们重点关注的是它的成员函数。等后面用到了再说。

我们先以 PoseGraph2D 的成员函数为线索，等有必要时再深入查看相应部分代码。

15.1、首先是第一个函数：InitializeGlobalSubmapPoses

```
// 该函数的主要工作就是指定一个 trajectory_id 的情况下，返回当前正处于活跃状态下的 submap 的 id，

// 也就是系统正在维护的 insertion_submaps 的两个 submap 的 id。insertion_submaps 可能为空，

// 也可能当前只有一个元素，也可能已经有两个元素了。

std::vector<SubmapId> PoseGraph2D::InitializeGlobalSubmapPoses(

    const int trajectory_id, const common::Time time,

    const std::vector<std::shared_ptr<const Submap2D>>& insertion_submaps) {

  CHECK(!insertion_submaps.empty());//检查该向量是否为空

  // 返回 OptimizationProblem2D 的成员变量： MapById<SubmapId, SubmapSpec2D> submap_data_;

  // submap_data_中存的就是以 SubmapId 为 key values 值管理的所有 Submap 的全局位姿。

  const auto& submap_data = optimization_problem_->submap_data();

    if (insertion_submaps.size() == 1) {//如果 size 为 1

    // If we don't already have an entry for the first submap, add one.

    // SizeOfTrajectoryOrZero: 返回一个指定 Id 的元素（这里的元素是 SubmapSpec2D）的数据的 size，如果该 id 不存在，则返回 0
```

```
    // 如果判断指定 id 的 submap_data 的 size 为 0，说明该 trajectory_id 上还没有 submap 数据，那么就需要建立一个 submap

    if (submap_data.SizeOfTrajectoryOrZero(trajectory_id) == 0) {

        if (initial_trajectory_poses_.count(trajectory_id) > 0){//带 Pose 的 trajectory 不为 0

      // TrajectoryConnectivityState 定义/mapping/internal/trajectory_connectivity_state.h 中

      // initial_trajectory_poses_中存的是一系列带有 int 型 id 的 InitialTrajectoryPose，

      // 一个 InitialTrajectoryPose 定义了与该 trajectory 相关联的 trajectory_id 的值。

      // 因此，下面这句是把该 trajectory_id 与其应该关联的 id(存在 InitialTrajectoryPose 的 to_trajectory_id 中)关联起来

            trajectory_connectivity_state_.Connect(//把这个表里数据与指定 id 关联

                trajectory_id,

                initial_trajectory_poses_.at(trajectory_id).to_trajectory_id, time);

      }

      // 将该 submap 的 global pose 加入到 optimization_problem_中，方便以后优化

      optimization_problem_->AddSubmap(

          trajectory_id,

          transform::Project2D(ComputeLocalToGlobalTransform(

              global_submap_poses_, trajectory_id) *insertion_submaps[0]->local_pose()));

    }

    // 看一下存 submap 的这个容器的 size 是否确实等于 1

    CHECK_EQ(1, submap_data.SizeOfTrajectoryOrZero(trajectory_id));

    // 因为是第一个 submap，所以该 submap 的 ID 是 trajectory_id+0，其中 0 是 submap 的 index.

    const SubmapId submap_id{trajectory_id, 0};
```

```
    // 检查这个 SubmapId 下的 submap 是否等于 insertion_submaps 的第一个元素。因为我们初始化第一个 submap 肯定是要被插入的那个 submap

    CHECK(submap_data_.at(submap_id).submap == insertion_submaps.front());

    // 如果是第一个 submap，那么把刚刚建立的 submap 的 id 返回

    return {submap_id};

  }

  CHECK_EQ(2, insertion_submaps.size());//检查 insertion_submaps 的 size 等于 2.

  // 获取 submap_data 的末尾 trajectory_id

  const auto end_it = submap_data.EndOfTrajectory(trajectory_id);

  // 检查开头是否不等于末尾。如果等于，说明这个容器里没有一个元素

  CHECK(submap_data.BeginOfTrajectory(trajectory_id) != end_it);

  // end_it 并没有实际的元素，所以它之前的一个 submap 的 id 就是 submap_data 中的最后一个元素

  const SubmapId last_submap_id = std::prev(end_it)->id;

  // 如果是等于 insertion_submaps 的第一个元素，说明 insertion_submaps 的第二个元素还没有分配了 id

  if (submap_data_.at(last_submap_id).submap == insertion_submaps.front()) {

    // In this case, 'last_submap_id' is the ID of

    // 'insertions_submaps.front()' and 'insertions_submaps.back()' is new.

    // 这种情况下，要给新的 submap 分配 id，并把它加到 OptimizationProblem 的 submap_data_这个容器中

    const auto& first_submap_pose = submap_data.at(last_submap_id).global_pose;

    optimization_problem_->AddSubmap(//解算新的 submap 的 pose 插入到 OptimizationProblem2D::submap_data_中

        trajectory_id,
```

```
        //然后把该 pose 乘以 insertion_submaps 中第一个 pose 的逆再乘第二个 pose
        first_submap_pose *
            constraints::ComputeSubmapPose(*insertion_submaps[0]).inverse() *
            constraints::ComputeSubmapPose(*insertion_submaps[1]));
    // 返回一个包含两个 Submap 的向量。
    return {last_submap_id,
            SubmapId{trajectory_id, last_submap_id.submap_index + 1}};
  }
  // 如果是等于 insertion_submaps 的第二个元素，说明 insertion_submaps 的第二个元素也已经分配了 id 并加入到了 OptimizationProblem 的 submap_data_中
  CHECK(submap_data_.at(last_submap_id).submap == insertion_submaps.back());
  // 生成一个 submap 的 Id, 这个 id 是 submap_data_的倒数第二个。
  const SubmapId front_submap_id{trajectory_id,last_submap_id.submap_index - 1};
  // 检查倒数第二个是否等于 insertion_submaps 的第一个。
  CHECK(submap_data_.at(front_submap_id).submap == insertion_submaps.front());
  // 把这两个 submap 的 id 返回
  return {front_submap_id, last_submap_id};
}
```

该函数由三个输入参数，一个是 trajectory_id，一个是 Local Slam 部分返回的 insertion_submaps，一个是时间。

该函数的主要工作就是指定一个 trajectory_id 的情况下，返回当前正处于活跃状态下的 submap 的 id，也就是系统正在维护的 insertion_submaps 的两个 submap 的 id。insertion_submaps 可能为空，也可能当前只有一个元素，也可能已经有两个元素了。

>>insertion_submaps 为空的情况下直接返回异常，说明 Local SLAM 那部分有异常，开始建图后一个 submap 也没建立成功；

>>如果 insertion_submaps 的 size 为 1，说明系统刚开始建图，那么为 insertion_submaps 中的这个 submap 分配一个 id （初始 id 为 trajectory_id + 一个 submap 的 index。由于这是第一个 submap，所有这个 submap 的 index 为 0），然后加入到 OptimizationProblem 里（OptimizationProblem 的 submap_data_存储了所有 submap 的绝对位姿，根据我们前面的理论分析也应该知道，OptimizationProblem 需要每个 submap 和 scan 的绝对位姿来进行全局优化）去，这时只需要返回的 submapId 的向量里只包含一个 submap 的 id，就是刚刚分配了 id 的这个 submap 的 id；

>>如果 insertion_submaps 的 size 为 2，说明系统已经处于正常模式的建图中，Local Slam 中已经在维护着两个 submap 了（一个 Old, 一个 New, 详见 Cartographer 源码阅读 9—TrajectoryBuilder 部分 2-Submap 后续中对 ActiveSubmaps2D 的解读）这时也得看 submap_data 中的最后一个 submap 是等于 Insertion_submaps 的第一个还是第二个:

>>>>如果是等于 insertion_submaps 的第一个元素，说明 insertion_submaps 的第二个元素还没有分配了 id，那么需要给该元素分配一个 id——即让 submap 的 index 加 1，然后把该元素加入到 OptimizationProblem 中，同时返回 submap_data 中的最后个 id 及让他的 index 加 1 后的 id。

>>>>如果是等于 insertion_submaps 的第二个元素，说明 insertion_submaps 的第二个元素也已经分配了 id 并加入到了 OptimizationProblem 的 submap_data_中，所以，这时候我直接返回 submap_data_中的最后一个元素及它之前一个元素的 submap 的 id 即可。

15.2、AddNode

// AddNode 的主要工作是把一个 TrajectoryNode 的数据加入到 PoseGraph 维护的 trajectory_nodes_这个容器中。并返回加入的节点的 Node.

```
NodeId PoseGraph2D::AddNode(

    std::shared_ptr<const TrajectoryNode::Data> constant_data,

    const int trajectory_id,

    const std::vector<std::shared_ptr<const Submap2D>>& insertion_submaps) {

  // 生成一个新的 Pose，这个 Pose 是由 trajectory 相对于世界坐标系的 Pose 乘以节点数据中包含的该节点相对于该 trajectory 的 LocalTrajectoryBuilder Pose

  // 所以生成的 optimized_pose 就是该节点的绝对位姿

  const transform::Rigid3d optimized_pose(
```

```
    GetLocalToGlobalTransform(trajectory_id) * constant_data->local_pose);
common::MutexLocker locker(&mutex_);
// 必要时新增一条 trajectory。怎么判断是否必要，查看 AddTrajectoryIfNeeded 的代码
AddTrajectoryIfNeeded(trajectory_id);
// TrajectoryNode 定义在/mapping/id.h.解读见：https://zhuanlan.zhihu.com/p/48790984
// 把该节点加入到 PoseGraph2D 维护的 trajectory_nodes_这个容器中
const NodeId node_id = trajectory_nodes_.Append(
    trajectory_id, TrajectoryNode{constant_data, optimized_pose});
// 节点数加 1.
++num_trajectory_nodes_;
// 跟上一个情况类似，如果 submap_data_还没有元素或最后一个元素不等于 insertion_submaps
// 的第二个元素
// 说明 insertion_submaps 的第二个元素还不被 PoseGraph 所知。
if (submap_data_.SizeOfTrajectoryOrZero(trajectory_id) == 0 ||
    std::prev(submap_data_.EndOfTrajectory(trajectory_id))->data.submap !=
        insertion_submaps.back()) {
  // We grow 'submap_data_' as needed. This code assumes that the first
  // time we see a new submap is as 'insertion_submaps.back()'.
      // 生成一个 SubmapId,把 insertion_submaps 的第二个元素加到 submap_data_中
      const SubmapId submap_id =
          submap_data_.Append(trajectory_id, InternalSubmapData());
      submap_data_.at(submap_id).submap = insertion_submaps.back();
```

```
    }

  // We have to check this here, because it might have changed by the time we

  // execute the lambda.

  // 检查 insertion_submaps 的第一个 submap 是否已经被 finished 了。

  // 如果被 finished，那么我们需要计算一下约束并且进行一下全局优化了。

  // 这里是把这个工作交到了 workItem 中等待处理

  const bool newly_finished_submap = insertion_submaps.front()->finished();

  AddWorkItem([=]() REQUIRES(mutex_) {

    ComputeConstraintsForNode(node_id, insertion_submaps,newly_finished_submap);

 });

 // 上述都做完了，返回 node_id.

 return node_id;

}
```

AddNode 的主要工作是把一个 TrajectoryNode 的数据加入到 PoseGraph 维护的 trajectory_nodes_这个容器中。并返回加入的节点的 Node.

但除此之外，在每次加入一个 TrajectoryNode 后，PoseGraph 都要检查一下 insertion_submaps 的 Old Submap 是否被 finished。如果已经被 finished，那么就需要进行 Loop Closure 了。

其中， AddNode 调用了 PoseGraph 的另外两个成员函数：AddTrajectoryIfNeeded 和 ComputeConstraintForNode.

前面我们理论分析时知道，PoseGraph 要优化的就是所有的 Submap 和所有的 TrajectoryNode 的绝对位姿，我们从上面两个函数也可以看到，Submap 的绝对位姿都保存在了 OptimizationProblem 的 submap_data_中，而 TrajectoryNode 的数据则维护在 PoseGraph 的 trajectory_nodes_这个容器中。

昨天我们测试了一下带 Landmark 的 demo（详见 Cartographer 源码阅读之附件 2——带 Landmark 的 demo 运行和测试），耽误了我们对源码解读的更新。今天开始我们继续对源码的解读。

15.3、AddTrajectoryIfNeeded 和 AddworkItem

在 PoseGraph2D::AddNode 这个函数中分别调用了 PoseGraph2D 的三个成员函数：AddTrajectoryIfNeeded、AddworkItem 和 ComputeConstraintsForNode. 所以我们先来看一下这三个函数：

AddTrajectoryIfNeeded:

```
// 把一个函数地址加到工作队列中
void PoseGraph2D::AddWorkItem(const std::function<void()>& work_item) {
  if (work_queue_ == nullptr) {// 如果工作队列为空，那么直接执行该函数即可
      work_item();
  } else {// 否则的话暂时加入工作队列，等待处理
      work_queue_->push_back(work_item);
  }
}
```

AddworkItem:

```
// 什么情况下才是 needed 呢？猜测应该是指定的 trajectory_id 如果不存在就需要调用该函数。
void PoseGraph2D::AddTrajectoryIfNeeded(const int trajectory_id) {
  // 增加一条 trajectory 的话就添加到 TrajectoryConnectivityState 中
  trajectory_connectivity_state_.Add(trajectory_id);
  // Make sure we have a sampler for this trajectory.
  // 应该是判断是否使用 GPS 信号。如果某条 trajectory 是使用了 GPS 的，那么为其分配一个 GPS 信号采样器
  if (!global_localization_samplers_[trajectory_id]) {
    global_localization_samplers_[trajectory_id] =
```

```
    common::make_unique<common::FixedRatioSampler>(
        options_.global_sampling_ratio());
  }
}
```

15.4、ComputeConstraintsForNode

PoseGraph2D::ComputeConstraintsForNode 的主要任务就是计算新添加的这个节点与所有 Submap 之间的约束关系。前面的理论分析中我们已经提到了，cartographer 中的约束指的就是一个节点和 submap 之间的相对位姿。所以，这部分代码也很容易理解，我们直接看代码：

```
// 三个参数，一个是刚加入的节点 ID；一个是 Local Slam 返回的 insertion_submaps；
// 一个是是否有新 finished 的 submap 的判断标志
void PoseGraph2D::ComputeConstraintsForNode(
    const NodeId& node_id,
    std::vector<std::shared_ptr<const Submap2D>> insertion_submaps,
    const bool newly_finished_submap) {
       // 获取节点数据
  const auto& constant_data = trajectory_nodes_.at(node_id).constant_data;
  // 根据节点数据的时间获取最新的 submap 的 id
  const std::vector<SubmapId> submap_ids = InitializeGlobalSubmapPoses(
      node_id.trajectory_id, constant_data->time, insertion_submaps);
  CHECK_EQ(submap_ids.size(), insertion_submaps.size());// 检查两者大小是否相等
  // 获取这两个 submap 中前一个的 id
  const SubmapId matching_id = submap_ids.front();
```

```
// 计算该 Node 经过重力 align 后的相对位姿，即在 submap 中的位姿

const transform::Rigid2d local_pose_2d = transform::Project2D(

    constant_data->local_pose *

    transform::Rigid3d::Rotation(constant_data->gravity_alignment.inverse()));

    // 计算该 Node 在世界坐标系中的绝对位姿；但中间为啥要乘一个
    constraints::ComputeSubmapPose(*insertion_submaps.front()).inverse()呢？

const transform::Rigid2d global_pose_2d =

    optimization_problem_->submap_data().at(matching_id).global_pose *

    constraints::ComputeSubmapPose(*insertion_submaps.front()).inverse() *

    local_pose_2d;//该 submap 的绝对位姿乘以在相应 submap 的相对位姿不就可以了吗？中间那一项
是啥呢？

// 把该节点的信息加入到 OptimizationProblem 中，方便进行优化

optimization_problem_->AddTrajectoryNode(

    matching_id.trajectory_id,

    optimization::NodeSpec2D{constant_data->time, local_pose_2d,

                             global_pose_2d,

                             constant_data->gravity_alignment});

//遍历处理每一个 insertion_submaps

for (size_t i = 0; i < insertion_submaps.size(); ++i) {

  const SubmapId submap_id = submap_ids[i];

  // Even if this was the last node added to 'submap_id', the submap will

  // only be marked as finished in 'submap_data_' further below.
```

```
    CHECK(submap_data_.at(submap_id).state == SubmapState::kActive);//检查指定 id 是否是
kActive

    // 加入到 PoseGraph 维护的容器中

    submap_data_.at(submap_id).node_ids.emplace(node_id);

    // 计算相对位姿

    const transform::Rigid2d constraint_transform =

        constraints::ComputeSubmapPose(*insertion_submaps[i]).inverse() *

        local_pose_2d;

    // 把约束压入约束集合中

    constraints_.push_back(Constraint{submap_id,

                                      node_id,

                                      {transform::Embed3D(constraint_transform),

                                       options_.matcher_translation_weight(),

                                       options_.matcher_rotation_weight()},

                                      Constraint::INTRA_SUBMAP});

  }

  // 遍历历史中的 submap，计算新的 Node 与每个 submap 的约束

  for (const auto& submap_id_data : submap_data_) {

    if (submap_id_data.data.state == SubmapState::kFinished) {//确认 submap 已经被 finished 了

      CHECK_EQ(submap_id_data.data.node_ids.count(node_id), 0);//检查该 submap 中还没有跟该节
点产生约束

      ComputeConstraint(node_id, submap_id_data.id);//计算该节点与 submap 的约束

    }
```

```
}

// 如果有新的刚被 finished 的 submap

if (newly_finished_submap) {

  //insertion_maps 中的第一个是 Old 的那个，如果有刚被 finished 的 submap，那一定是他

  const SubmapId finished_submap_id = submap_ids.front();

  // 获取该 submap 的数据

  InternalSubmapData& finished_submap_data =

      submap_data_.at(finished_submap_id);

  // 检查它还是不是 kActive。

  CHECK(finished_submap_data.state == SubmapState::kActive);

  // 把它设置成 finished

  finished_submap_data.state = SubmapState::kFinished;

  // We have a new completed submap, so we look into adding constraints for

  // old nodes.

  //计算新的 submap 和旧的节点的约束

  ComputeConstraintsForOldNodes(finished_submap_id);

}

// 结束构建约束

constraint_builder_.NotifyEndOfNode();

// 计数器加 1

++num_nodes_since_last_loop_closure_;

CHECK(!run_loop_closure_);//检查没进行过 Loop Closure
```

```
  if (options_.optimize_every_n_nodes() > 0 &&
      num_nodes_since_last_loop_closure_ > options_.optimize_every_n_nodes()) {
    DispatchOptimization();//如果节点数增长到一定地步，则调用 DispatchOptimization
  }
}
```

其中分别调用了 InitializeGlobalSubmapPoses、ComputeConstraint、ComputeConstraintsForNodes、DispatchOptimization 四个函数。

DispatchOptimization 我们先往后放一放。InitializeGlobalSubmapPoses 的主要任务我们前面已经介绍了，该函数的主要工作就是指定一个 trajectory_id 的情况下，返回当前正处于活跃状态或最新的 submap 的 id。接下来我们先来看一下分别计算新节点与旧的 submap 之间约束的 ComputeConstraint 以及计算新的 Submap 和旧的节点之间约束的 ComputeConstraintForOldNodes.

15.5、ComputeConstraint 和 ComputeConstraintForOldNodes

ComputeConstraint: 计算一个 Node 和一个 submap 之间的约束。通过在 ComputeConstraintForNode 中遍历所有的历史 submap，每次均调用 ComputeConstraint 来建立约束：

```
// 计算一个节点与一个 submap 的约束关系
void PoseGraph2D::ComputeConstraint(const NodeId& node_id,
                                    const SubmapId& submap_id) {
  // 检查该 submap 的状态是否是 finished
  CHECK(submap_data_.at(submap_id).state == SubmapState::kFinished);
  // 获取该 node 和该 submap 最新一次具有约束的时间
  const common::Time node_time = GetLatestNodeTime(node_id, submap_id);
  // 获取该 node 和该 submap 各自对应的 trajectory 上次 connected 的时间
  const common::Time last_connection_time =
      trajectory_connectivity_state_.LastConnectionTime(
```

```
        node_id.trajectory_id, submap_id.trajectory_id);
// If the node and the submap belong to the same trajectory or if there
// has been a recent global constraint that ties that node's trajectory to
// the submap's trajectory, it suffices to do a match constrained to a
// local search window.
// 如果两者对应的 trajectory_id 相同或者两者对应的 trajectory 存在一个全局约束，
// 那么就可以在一个局部搜索窗口做匹配
if (node_id.trajectory_id == submap_id.trajectory_id ||
    node_time <
        last_connection_time +common::FromSeconds(
            options_.global_constraint_search_after_n_seconds())) {
  // 计算该 node 和 submap 的相对位姿
  const transform::Rigid2d initial_relative_pose =
      optimization_problem_->submap_data()
          .at(submap_id)
          .global_pose.inverse() *
      optimization_problem_->node_data().at(node_id).global_pose_2d;
  // 利用 ConstraintBuilder 为两者添加约束
  constraint_builder_.MaybeAddConstraint(
      submap_id, submap_data_.at(submap_id).submap.get(), node_id,
   trajectory_nodes_.at(node_id).constant_data.get(),
   initial_relative_pose);
```

```
  } else if (global_localization_samplers_[node_id.trajectory_id]->Pulse()) {// 如果 Node 对应的 trajectory 采样了 GPS 信号？？

    // 为两者添加一个全局约束。

    constraint_builder_.MaybeAddGlobalConstraint(

        submap_id, submap_data_.at(submap_id).submap.get(), node_id,

        trajectory_nodes_.at(node_id).constant_data.get());

  }

}
```

ComputeConstraintsForOldNodes: 对一个刚刚被 finished 的 submap，遍历所有的 Node，并建立其该 submap 与 Node 之间的约束

```
// 计算一个新被 finished 的 submap 与所有旧的 Node 之间的约束

void PoseGraph2D::ComputeConstraintsForOldNodes(const SubmapId& submap_id) {

  // 对于刚刚被 finished 的 submap，取出该 submap 的数据

  const auto& submap_data = submap_data_.at(submap_id);

  // 依次取出 OptimizationProblem 里面存的所有 Node

  for (const auto& node_id_data : optimization_problem_->node_data()) {

    const NodeId& node_id = node_id_data.id;

    if(submap_data.node_ids.count(node_id)==0){//如果该节点和该 submap 之前没有建立过约束

      ComputeConstraint(node_id, submap_id);//计算这两者的约束

    }

  }

}
```

第 16 章 PoseGraph 成员函数 2

在上一篇文章中我们已经解读了 PoseGraph2D::AddNode 函数以及该函数所依赖的 AddTrajectoryIfNeeded、AddWorkItem、ComputeConstraintsForNode，以及 ComputeConstraintsForNode 所依赖的 InitializeGlobalSubmapPoses、ComputeConstraint 和 ComputeConstraintsForOldNodes（该函数调用了 ComputeConstraint)等函数。

我们接着看下面两个函数：DispatchOptimization 和 GetLatestNodeTime。DispatchOptimization 在 ComputeConstraintsForNode 中被调用，而 GetLatestNodeTime 在 ComputeConstraint 中被调用。

16.1、DispatchOptimization()

```
// Dispatch 的含义是分拨、派遣、分拣。这里的主要是将 run_loop_closure_置为 true,
// 从而使得执行优化操作的代码块知道应该进行全局优化了。
void PoseGraph2D::DispatchOptimization() {
  run_loop_closure_ = true;
  // If there is a 'work_queue_' already, some other thread will take care.
  //如果工作队列还为空指针，需要创建一个工作队列
  if (work_queue_ == nullptr) {
    //创建一个工作队列的指针
    work_queue_ = common::make_unique<std::deque<std::function<void()>>>();
    //为该工作队列绑定处理函数，即 HandleWorkQueue
    constraint_builder_.WhenDone(
        std::bind(&PoseGraph2D::HandleWorkQueue, this, std::placeholders::_1));
  }
}
```

7. GetLatestNodeTime

```
// 获取该 node 和该 submap 中的 node 中较新的时间
common::Time PoseGraph2D::GetLatestNodeTime(const NodeId& node_id,
                                            const SubmapId& submap_id) const {
  // 获取指定 id 的时间
  common::Time time = trajectory_nodes_.at(node_id).constant_data->time;
  // 获取指定 id 的 submap 的数据
  const InternalSubmapData& submap_data = submap_data_.at(submap_id);
  if (!submap_data.node_ids.empty()) {//如果该 submap 相关的 node_ids 不为空。
    // 获取指定 id 的 submap 相关的 node_ids 列表中的最后一个元素
    const NodeId last_submap_node_id =
        *submap_data_.at(submap_id).node_ids.rbegin();//c.rbegin() 返回一个逆序迭代器，它指向容器 c 的最后一个元素
    // 把时间更新为节点建立时间与 submap 中最后一个节点时间中较晚的那个
    time = std::max(
        time, trajectory_nodes_.at(last_submap_node_id).constant_data->time);
  }
  return time;
}
```

之后我们依据 pose_graph_2d.cc 中的顺序来依次解读一下各个函数的功能，基本上完成这些函数的解读后我们对 Cartographer 源码的解读也就基本告成。

16.2、AddImuData、AddOdometryData、AddFixedFramePoseData、AddLandmarkData

这几个函数的主要作用是在有新的传感器数据传入时进行处理：

```
void PoseGraph2D::AddImuData(const int trajectory_id,
                             const sensor::ImuData& imu_data) {
  common::MutexLocker locker(&mutex_);//设置互斥锁
  AddWorkItem([=]() REQUIRES(mutex_) {
    optimization_problem_->AddImuData(trajectory_id, imu_data);
  });
}
void PoseGraph2D::AddOdometryData(const int trajectory_id,
                                  const sensor::OdometryData& odometry_data) {
  common::MutexLocker locker(&mutex_);
  AddWorkItem([=]() REQUIRES(mutex_) {
    optimization_problem_->AddOdometryData(trajectory_id, odometry_data);
  });
}
```

可以看出，IMU 和里程计数据有新的测量结果后是调用 optimization_problem_进行处理，该函数的处理目标也是优化机器人行进过程中的 Node 处的位姿以及 Submap 的绝对位姿，使其更好地符合观测结果。具体的处理方法等我们深入解读 OptimizationProblem 时再研究。

```
void PoseGraph2D::AddFixedFramePoseData(
    const int trajectory_id,
    const sensor::FixedFramePoseData& fixed_frame_pose_data) {
  LOG(FATAL) << "Not yet implemented for 2D.";
}
```

AddFixedFramePoseData 是处理 GPS 等能够测量机器人完整位姿的传感器输入，目前 2D 情况下并没有用到该函数。

```
// 添加 Landmark 数据

void PoseGraph2D::AddLandmarkData(int trajectory_id,

                                  const sensor::LandmarkData& landmark_data)

    EXCLUDES(mutex_) {

  common::MutexLocker locker(&mutex_);

  AddWorkItem([=]() REQUIRES(mutex_) {

    for (const auto& observation : landmark_data.landmark_observations) {//遍历传入的参数中的 landmark_data

      //根据数据生成一个 LandmarkNode::LandmarkObservation，压入 landmark_nodes_这个容器中

      landmark_nodes_[observation.id].landmark_observations.emplace_back(

          LandmarkNode::LandmarkObservation{

              trajectory_id, landmark_data.time,

              observation.landmark_to_tracking_transform,

              observation.translation_weight, observation.rotation_weight});

    }

  });

}
```

AddLandmarkData 则是处理有 Landmark 的 Observation 的情况。Landmark 的处理与 IMU 和里程计数据的处理不太一样。对于 IMU 和里程计数据而言，他们的局部信息是比较可靠的，所以在每次有新的观察结果时立刻交给 OptimizationProblem 进行处理。但对于 Landmark 的数据而言，cartographer 认为 Landmark 的绝对位姿是已知的，而机器人在行进过程中可能会多次观察 landmark，因此，landmark 并不是有观测后就立刻处理，而是在进行 Loop Closure 之前才调用，在调用时可能观察到多次，遍历这些观测数据，并把他们放到 landmark_nodes_这个容器中，然后在 Loop Closure 时一起优化。