cartographer从入门到精通: 原理深剖+源码逐行讲解

第三章 传感器数据的处理

3.1 MapBuilder类

3.1.1 node_main.cc

```
auto map_builder =
cartographer::mapping::CreateMapBuilder(node_options.map_builder_options);

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

3.1.2 Node类的构造函数

```
Node::Node(
    const NodeOptions& node_options,
    std::unique_ptr<cartographer::mapping::MapBuilderInterface>
map_builder,
    tf2_ros::Buffer* const tf_buffer, const bool collect_metrics)
    : node_options_(node_options),
      map_builder_bridge_(node_options_, std::move(map_builder), tf_buffer)
{
  // Step: 1 声明需要发布的topic
  // Step: 2 声明发布对应名字的ROS服务，并将服务的发布器放入到vector容器中
  // Step: 3 处理之后的点云的发布器
  // Step: 4 进行定时器与函数的绑定，定时发布数据
}
```

3.1.3 MapBuilderBridge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) {}
```

3.1.4 MapBuilder类的构造函数

```
MapBuilder::MapBuilder(const proto::MapBuilderOptions& options)
    : options_(options), thread_pool_(options.num_background_threads()) {

  // 2d位姿图(后端)的初始化
  if (options.use_trajectory_builder_2d()) {
    pose_graph_ = absl::make_unique<PoseGraph2D>(
        options_.pose_graph_options(),
        absl::make_unique<optimization::OptimizationProblem2D>(
            options_.pose_graph_options().optimization_problem_options()),
        &thread_pool_);
  }

  if (options.collate_by_trajectory()) {
    sensor_collator_ = absl::make_unique<sensor::TrajectoryCollator>();
  } else {
    sensor_collator_ = absl::make_unique<sensor::Collator>();
  }
}
```

3.2 传感器数据分发器的创建

3.2.1 node_main.cc的StartTrajectoryWithDefaultTopics函数

```
node.StartTrajectoryWithDefaultTopics(trajectory_options);
```

3.2.2 Node类的StartTrajectoryWithDefaultTopics函数

```
StartTrajectoryWithDefaultTopics();
  AddTrajectory(options);
    const int trajectory_id =
map_builder_bridge_.AddTrajectory(expected_sensor_ids, options);
    LaunchSubscribers(options, trajectory_id); // 开始订阅传感器数据话题
```

3.2.3 MapBuilderBridge类的AddTrajectory函数

```
int MapBuilderBridge::AddTrajectory() {
  // Step: 1 开始一条新的轨迹, 返回新轨迹的id,需要传入一个函数
  const int trajectory_id = map_builder_->AddTrajectoryBuilder(
      expected_sensor_ids, trajectory_options.trajectory_builder_options,
      [this]() {
        OnLocalSlamResult(trajectory_id, time, local_pose,
range_data_in_local);
      });

  // Step: 2 为这个新轨迹 添加一个SensorBridge
  sensor_bridges_[trajectory_id] = absl::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)); //
CollatedTrajectoryBuilder

  return trajectory_id;
}
```

3.2.4 MapBuilder类的AddTrajectoryBuilder函数

```
int MapBuilder::AddTrajectoryBuilder(
    const std::set<SensorId>& expected_sensor_ids,
    const proto::TrajectoryBuilderOptions& trajectory_options,
    LocalSlamResultCallback local_slam_result_callback) {

    // CollatedTrajectoryBuilder初始化

 trajectory_builders_.push_back(absl::make_unique<CollatedTrajectoryBuilder
>(
        trajectory_options,
        sensor_collator_.get(), // sensor::Collator
        trajectory_id,
        expected_sensor_ids,
        CreateGlobalTrajectoryBuilder2D(
            std::move(local_trajectory_builder), trajectory_id,
            static_cast<PoseGraph2D*>(pose_graph_.get()),
            local_slam_result_callback,
pose_graph_odometry_motion_filter)));

  return trajectory_id;
}

// 返回指向CollatedTrajectoryBuilder的指针
mapping::TrajectoryBuilderInterface *GetTrajectoryBuilder(
    int trajectory_id) const override {
    return trajectory_builders_.at(trajectory_id).get();
}
```

3.2.5 SensorBridge类的构造函数

```
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)
// CollatedTrajectoryBuilder
    : num_subdivisions_per_laser_scan_(num_subdivisions_per_laser_scan),
      tf_bridge_(tracking_frame, lookup_transform_timeout_sec, tf_buffer),
      trajectory_builder_(trajectory_builder) // CollatedTrajectoryBuilder
      {}
```

3.2.6 CollatedTrajectoryBuilder类的构造函数

```
CollatedTrajectoryBuilder::CollatedTrajectoryBuilder(
    const proto::TrajectoryBuilderOptions& trajectory_options,
    sensor::CollatorInterface* const sensor_collator,   // sensor::Collator
    const int trajectory_id,
    const std::set<SensorId>& expected_sensor_ids,
    std::unique_ptr<TrajectoryBuilderInterface> wrapped_trajectory_builder)
// GlobalTrajectoryBuilder
    : sensor_collator_(sensor_collator),                // sensor::Collator
      trajectory_id_(trajectory_id),
      wrapped_trajectory_builder_(std::move(wrapped_trajectory_builder)),
 // GlobalTrajectoryBuilder
      last_logging_time_(std::chrono::steady_clock::now()) {

  // sensor::Collator的初始化
  sensor_collator_->AddTrajectory(
      trajectory_id, expected_sensor_id_strings,
      [this](const std::string& sensor_id, std::unique_ptr<sensor::Data>
data) {
        HandleCollatedSensorData(sensor_id, std::move(data));
      });
}
```

在Collator构造的时候传入了一个函数 HandleCollatedSensorData()

3.2.7 Collator类的AddTrajectory函数

```
void Collator::AddTrajectory(
    const int trajectory_id,
    const absl::flat_hash_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,
                    // void(std::unique_ptr<Data> data) 带了个默认参数
sensor_id
                    [callback, sensor_id](std::unique_ptr<Data> data) {
                      callback(sensor_id, std::move(data));
```

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

// note: CollatorInterface::Callback 2个参数
using Callback = std::function<void(const std::string&,
std::unique_ptr<Data>)>;
```

在这将传入的Callback函数放入 queue_ 里, 并传入一个参数.

3.2.8 OrderedMultiQueue类的AddQueue函数

```
void OrderedMultiQueue::AddQueue(const QueueKey& queue_key, Callback
callback) {
  CHECK_EQ(queues_.count(queue_key), 0);
  queues_[queue_key].callback = std::move(callback);
}

// note: OrderedMultiQueue::Callback 1个参数
using Callback = std::function<void(std::unique_ptr<Data>)>;
```

这里的Callback是一个参数的.

3.3 传感器数据走向分析

3.3.1 Node类的HandleLaserScanMessage函数

```
void Node::HandleLaserScanMessage(const int trajectory_id,
                                  const std::string& sensor_id,
                                  const sensor_msgs::LaserScan::ConstPtr&
msg) {
  map_builder_bridge_.sensor_bridge(trajectory_id)
      ->HandleLaserScanMessage(sensor_id, msg);
}
```

3.3.2 SensorBridge类的HandleLaserScanMessage函数

```
void SensorBridge::HandleLaserScanMessage(
    const std::string& sensor_id, const sensor_msgs::LaserScan::ConstPtr&
msg) {
  carto::sensor::PointCloudWithIntensities point_cloud;
  carto::common::Time time;
  std::tie(point_cloud, time) = ToPointCloudWithIntensities(*msg);
  HandleLaserScan(sensor_id, time, msg->header.frame_id, point_cloud);
}

void SensorBridge::HandleRangefinder(
    const std::string& sensor_id, const carto::common::Time time,
```

```
    const std::string& frame_id, const carto::sensor::TimedPointCloud&
ranges) {
  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>())} ); //
强度始终为空
  }
}
```

SensorBridge的trajectory_builder_是指向CollatedTrajectoryBuilder的指针

3.3.3 CollatedTrajectoryBuilder类的AddSensorData函数

```
void AddSensorData(
    const std::string& sensor_id,
    const sensor::TimedPointCloudData& timed_point_cloud_data) override {
    AddData(sensor::MakeDispatchable(sensor_id, timed_point_cloud_data));
}

void CollatedTrajectoryBuilder::AddData(std::unique_ptr<sensor::Data> data)
{
  sensor_collator_->AddSensorData(trajectory_id_, std::move(data));
}
```

CollatedTrajectoryBuilder的sensor_collator_是指向Collator的指针

3.3.4 Collator类的AddSensorData函数

```
void Collator::AddSensorData(const int trajectory_id,
                             std::unique_ptr<Data> data) {
  QueueKey queue_key{trajectory_id, data->GetSensorId()};
  queue_.Add(std::move(queue_key), std::move(data));
}
```

3.3.5 OrderedMultiQueue类的Add函数 - 生成者

```
void OrderedMultiQueue::Add(const QueueKey& queue_key,
                            std::unique_ptr<Data> data) {
  auto it = queues_.find(queue_key);

  // 向数据队列中添加数据
  it->second.queue.Push(std::move(data));

  // 传感器数据的分发处理
  Dispatch();
}
```

3.3.6 BlockingQueue类的Push函数 - 缓冲区(阻塞队列)

```
  void Push(T t) {
    // 将数据加入队列，移动而非拷贝
    deque_.push_back(std::move(t));
  }
```

3.3.7 OrderedMultiQueue类的Dispatch函数 - 消费者

```
void OrderedMultiQueue::Dispatch() {
  while (true) {
    const Data* next_data = nullptr;
    Queue* next_queue = nullptr;
    QueueKey next_queue_key;

    // 遍历所有的数据队列，找到所有数据队列的第一个数据中时间最老的一个数据
    for (auto it = queues_.begin(); it != queues_.end();) {
      const auto* data = it->second.queue.Peek<Data>();
    } // end for

    // 正常情况，数据时间都超过common_start_time
    if (next_data->GetTime() >= common_start_time) {
      last_dispatched_time_ = next_data->GetTime();
      // 将数据传入 callback() 函数进行处理,并将这个数据从数据队列中删除
      next_queue->callback(next_queue->queue.Pop());
    }
  }
}
```

3.3.8 CollatedTrajectoryBuilder类的HandleCollatedSensorData函数 - 消费者

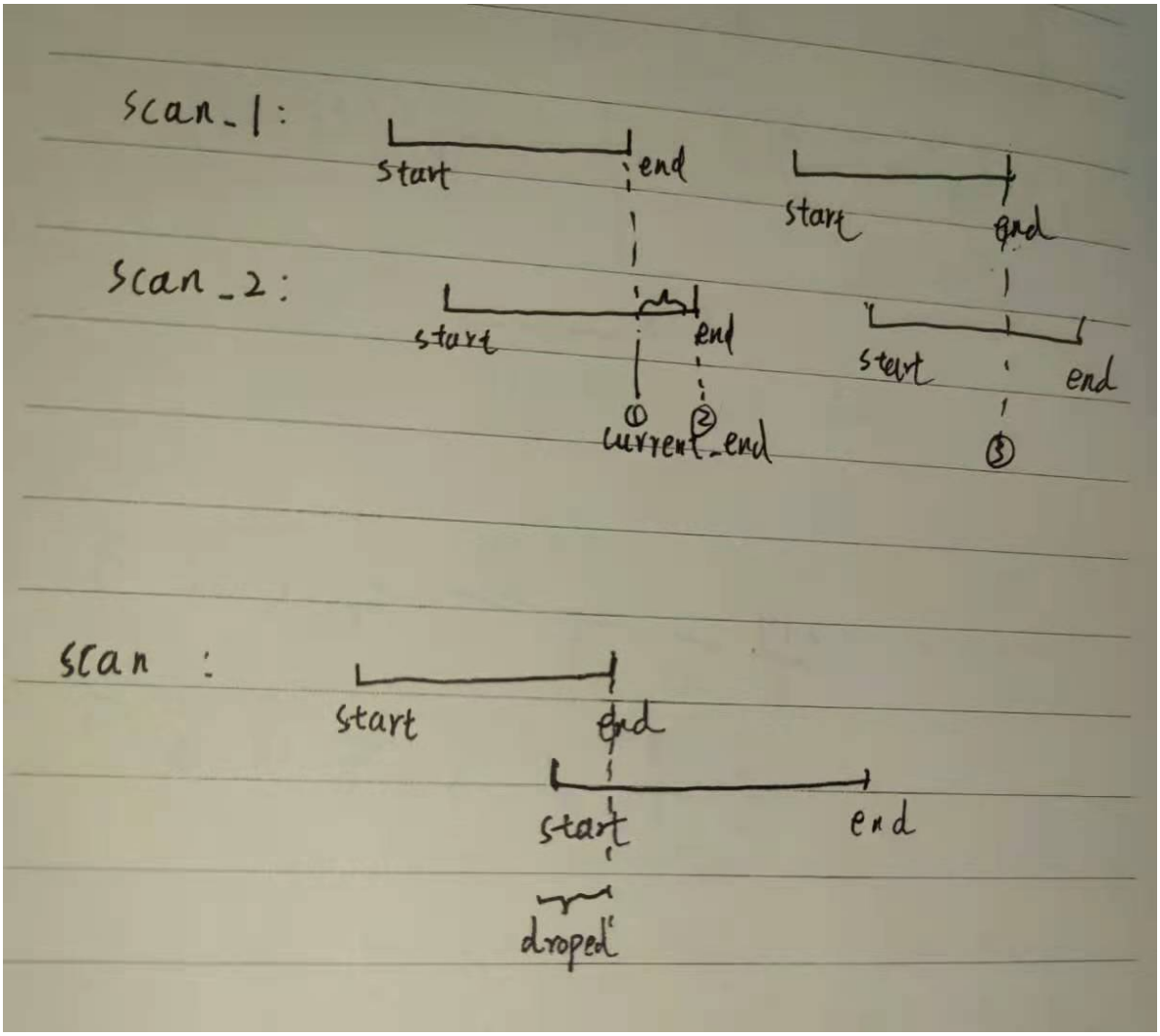
```
void CollatedTrajectoryBuilder::HandleCollatedSensorData(
    const std::string& sensor_id, std::unique_ptr<sensor::Data> data) {
  // 将排序好的数据送入 GlobalTrajectoryBuilder中的AddSensorData()函数中进行
使用
  data->AddToTrajectoryBuilder(wrapped_trajectory_builder_.get());
}

void Dispatchable::AddToTrajectoryBuilder(
    mapping::TrajectoryBuilderInterface *const trajectory_builder) override
{
    trajectory_builder->AddSensorData(sensor_id_, data_);
}
```

这里的wrapped_trajectory_builder_是指向GlobalTrajectoryBuilder2D类的指针.

从GlobalTrajectoryBuilder2D开始, 数据才真正走到SLAM的前端与后端部分.

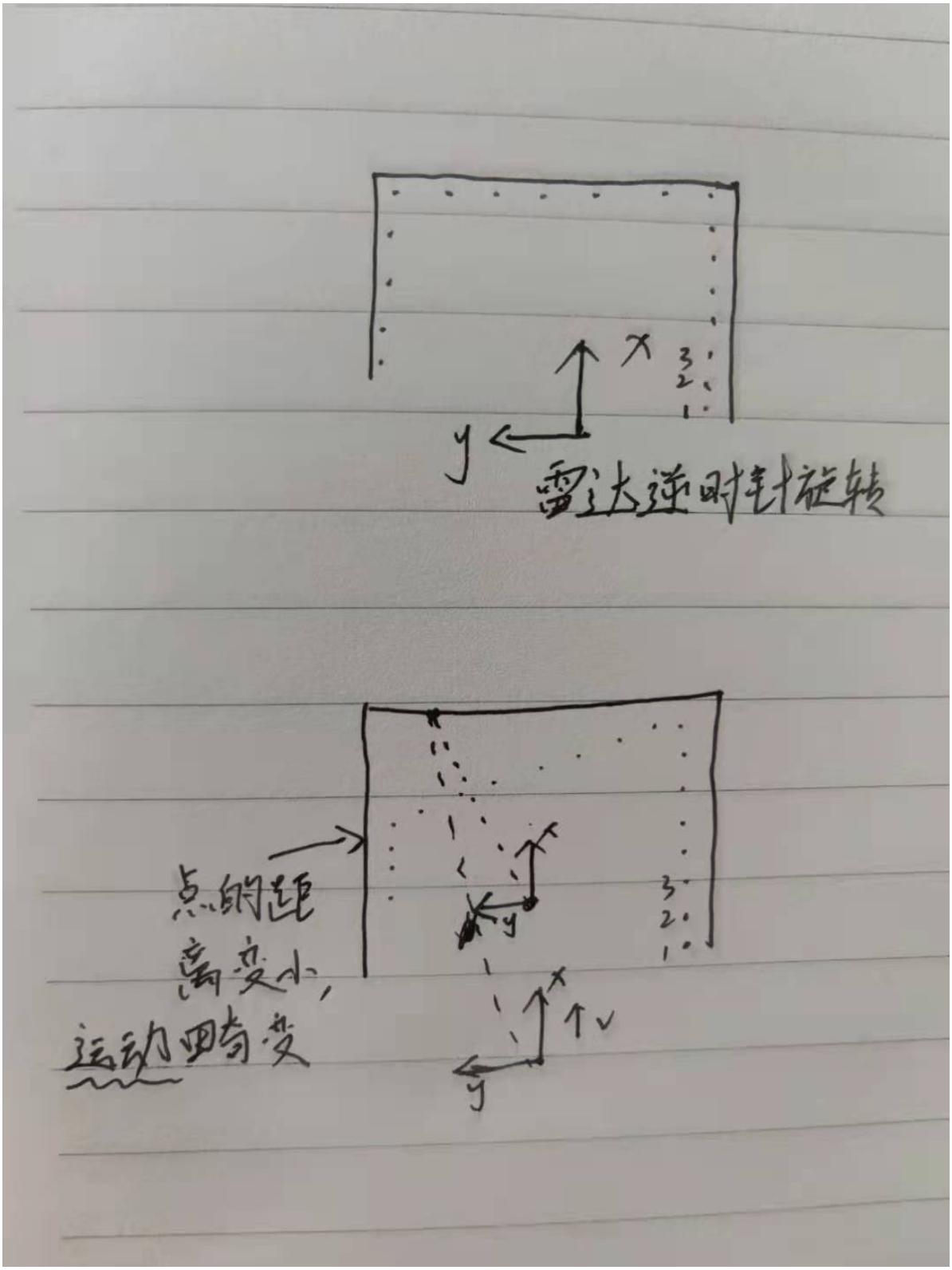
3.4 2D场景激光雷达数据的预处理

第11讲 多个激光雷达数据的时间同步与融合



第12讲 激光雷达数据的运动畸变的校正

激光雷达数据运动畸变的校正, 同时将点云的相对于tracking_frame的点坐标 转成 相对于local slam坐标系的点坐标.



第13讲 点云的坐标变换与z轴的过滤

- 第一步 计算从点云当前原点坐标变换到local坐标系原点的坐标变换
  `range_data_poses.back().inverse()`
- 第二步 将当前机器人姿态乘以这个坐标变换, 获得 从点云当前原点坐标变换到local坐标系原点, 并且姿态为0 的坐标变换
  `gravity_alignment.cast<float>() * range_data_poses.back().inverse()`

- 第三步 将原点位于机器人当前位姿处的点云 转成 原点位于local坐标系原点处的点云
  `sensor::TransformRangeData(range_data, transform_to_gravity_aligned_frame)`
- 第四步 进行z轴的过滤
  `sensor::CropRangeData(sensor::TransformRangeData(range_data, transform_to_gravity_aligned_frame), options_.min_z(), options_.max_z())`

  单线雷达不能设置 大于0的min_z, 因为单线雷达的z为0

第14讲 体素滤波与之后的处理

- 分别对 returns点云 与 misses点云 进行体素滤波

```
sensor::RangeData{ cropped.origin,
     sensor::VoxelFilter(cropped.returns, options_.voxel_filter_size()),
     sensor::VoxelFilter(cropped.misses, options_.voxel_filter_size())};
```

- 对 returns点云 进行自适应体素滤波

`sensor::AdaptiveVoxelFilter(gravity_aligned_range_data.returns, options_.adaptive_voxel_filter_options())`

- 将 原点位于local坐标系原点处的点云 变换成 原点位于匹配后的位姿处的点云

`TransformRangeData(gravity_aligned_range_data, transform::Embed3D(pose_estimate_2d->cast<float>()) )`

- 将 原点位于匹配后的位姿处的点云 返回到node.cc 中, node.cc将这个点云发布出去, 在rviz中可视化

3.5 3D场景激光雷达数据的预处理

- 进行多个雷达点云数据的时间同步
- 对点云进行第一次体素滤波
- 激光雷达数据运动畸变的校正, 同时将点云的相对于tracking_frame的点坐标 转成 相对于local slam坐标系的点坐标
- 分别对 returns 与 misses 进行第二次体素滤波
- 将原点位于机器人当前位姿处的点云 转成 原点位于local坐标系原点处的点云
- 使用高分辨率进行自适应体素滤波 生成高分辨率点云
- 使用低分辨率进行自适应体素滤波 生成低分辨率点云
- 将 原点位于local坐标系原点处的点云 变换成 原点位于匹配后的位姿处的点云
- 将 原点位于匹配后的位姿处的点云 返回到node.cc 中, node.cc将这个点云发布出去, 在rviz中可视化