# Gmapping 建图原理

本文介绍了 Gmapping 建图算法的使用方式，整体代码框图，数据流。基于激光数据的粒子滤波原理等。

## 1. 简介

Gmapping 是基于弗莱堡大学和罗马大学几个教授的论文，实现了一套基于粒子滤波的 SLAM 算法。论文于2005年提出，使用激光雷达于里程计两种传感器数据作为算法演进的输入源，构建出二维地图。Gmapping 将 SLAM 转为一种同时推测机器人的空间位姿和创建地图的问题，这种方法使用了粒子滤波器作为位姿估算，每个粒子都持有环境中的一张单独的地图。通过自适应技术来减少建图过程中的粒子数，来减少对内存的要求。

此套代码最后更新与 2012 年，本文所涉及的源码都基于此版本。

## 2. Gmapping 示例

### 2.1 建图示例

前面简单了解了工作的基本原理，在已安装 ROS 环境，并且 Gmapping 库已经安装到系统情况下。本节通过示例演示如何通过 ROS 节点的方式使用此算法。

#### 2.1.1 启动算法节点

首先设置节点启动参数，在 ROS 下一般通过 yaml 文件配置，示例如下：

```yaml
map_update_interval: 2.0  # 地图更新频率，2.0 秒一次
maxUrange: 3.0            # 探测最大可用范围，即光束有效范围
sigma: 0.05               # 匹配标准差，超过此标准差，匹配即不成功
kernelSize: 1
lstep: 0.05               # 平移优化步长
astep: 0.05               # 旋转优化步长
iterations: 5             # 扫描匹配迭代步数
lsigma: 0.075             # laser 扫描匹配标准差
ogain: 3.0                #
lskip: 0                  # 扫描跳过的激光帧
minimumScore: 50          # 最小得分，用来判断帧匹配算得分
srr: 0.1                  # 平移时里程误差作为平移函数
srt: 0.2                  # 平移时的里程误差作为旋转函数
str: 0.1                  # 旋转时的里程误差作为平移函数
stt: 0.2                  # 旋转时的里程误差作为旋转函数
linearUpdate: 1.0
angularUpdate: 0.2
temporalUpdate: 0.5
resampleThreshold: 0.5    # 重采样的阀值
particles: 100            # 默认粒子滤波器粒子数目
xmin: -10.0               # 地图初始尺寸，最小 x 的坐标
ymin: -10.0               # 地图初始尺寸，最小 y 的坐标
xmax: 10.0                # 地图初始尺寸，最大 x 的坐标
ymax: 10.0                # 地图初始尺寸，最大 y 的坐标
delta: 0.05               # 地图分辨率，每个栅格在真实环境下的尺寸
llsamplerange: 0.01
llsamplestep: 0.01
```

```
lasamplerange: 0.005
lasamplestep: 0.005
```

这里有几个重要的参数：

- resampleThreshold(float, default: 0.5)：重采样的阀值， 在粒子权重的相似度不超过此阀值下，才做重采样解决粒子退化的问题
- minimumScore (float, default: 0.0)：最小匹配得分，这个参数很重要，它决定了对激光的一个置信度，越高说明对激光匹配算法的要求越高，激光的匹配也越容易失败而转去使用里程计数据，而设的太低又会使地图中出现大量噪声，所以需要权衡调整
- particles (int, default: 30)：gmapping算法中的粒子数，因为gmapping使用的是粒子滤波算法，粒子在不断地迭代更新，所以选取一个合适的粒子数可以让算法在保证比较准确的同时有较高的速度

就可以写类似以下 launch 文件来启动这个节点：

```
<launch>
 <node pkg="gmapping" type="slam_gmapping" name="example_slam_gmapping"
output="screen">
  <param name="base_frame" value="base_footprint"/>
  <param name="odom_frame" value="odom"/>
  <param name="map_frame" value="map"/>
  <rosparam command="load" file="$(pwd)/config/gmapping_params.yaml" />
 </node>
</launch>
```

通过以下命令来启动节点：

```
roslaunch slam gmapping_example.launch
```

如果不想写 launch 文件， 并且大部分参数都使用默认参数，可以把少部分参数拼接在命令后面，比如可以通过以下命令来启动算法节点：

```
rosrun gmapping slam_gmapping scan:=scan particles:=30
```

### 2.1.2 启动观察节点

Rviz 是 ROS 自带的图形化工具，可以很方便的让用户通过图形界面开发调试 ROS。操作界面也十分简洁，界面主要分为上侧菜单区、左侧显示内容设置区、中间显示区、右侧显示视角设置区、下侧 ROS 状态区。

这里我们只要用来观察地图的更新，所以在显示设置区，添加监听的话题，Rviz 在监听到地图话题后，会自动刷新更新地图。

- 通过 ROS 命令启动 Rviz

  ```
  roscore &

  rosrun rviz rviz
  ```

- 左下角 Add 按钮增加 Map， 添加完后，配置 Map 下的 Topic 到系统运行时对应的话题，一般为 "/map"

这样就可以看到 Rviz 平面上有一系列的栅格，这里每个栅格对应实际空间上的长度为 1m。

### 2.1.3 播放传感器数据

在 ROS 系统中，可以用 rosbag 来保存和恢复系统的运行状态，比如可以录制系统运行时的发送的话题消息数据包，然后通过回放的方式重现系统运行状态。

rosbag 工具可以录制一个包、从一个或多个包中重新发布消息、查看一个包的基本信息、检查一个包的消息定义，基于 Python 表达式过滤一个包的消息，压缩和解压缩一个包以及重建一个包的索引。

rosbag 目前常用的命令如下（fix 和 filter 暂时没有用到）：

- `record`：用指定的话题录制一个 bag 包
- `info`：显示一个 bag 包的基本信息，比如包含哪些话题
- `play`：回放一个或者多个 bag 包
- `check`：检查一个 bag 包在当前的系统中是否可以回放和迁移
- `compress`：压缩一个或多个 bag 包
- `decompress`：解压缩一个或多个 bag 包
- `reindex`：重新索引一个或多个损坏 bag 包
- `filter`：对 bag 包过滤，过滤掉不需要的数据包

我们通过真实机器人，或者 gazebo 模拟器，把传感器数据录制下来后，就可以通过 rosbag 回放来重复构建地图：

```
rosbag play slam-gmapping.bag
```

## 2.2 地图保存

上节生成的地图通过话题的方式在系统中传输，任意一个节点都可以侦听到此数据，在 ROS 系统里，有个 map_server 的软件包提供了一个节点，该节点可以通过服务调用的方式操作地图，使用该工具可将动态构建的地图保存成文件。

它采用 C/S 模式，使用此软件包，需要提前启动服务端，服务端会侦听系统中地图的话题，并把地图的数据保存在节点内存中。启动服务端节点需要包含类似下面yaml 格式的启动参数：

```
image: gmapping.pgm        # 地图对应的文件
resolution: 0.1            # 地图的分辨率，单位是meters / pixel
origin: [0.0, 0.0, 0.0]    # 地图的位姿，表示为 (x, y, yaw)，这里yaw是逆时针旋转角度
(yaw=0意味着没有旋转)。目前多数系统忽略yaw值
occupied_thresh: 0.65      # 占用的阀值，概率超过此阀值就认为有障碍
free_thresh: 0.196         # 空闲的阀值，概率低于此阀值就认为空闲
negate: 0
```

服务端启动：

```
rosrun map_server map_server mapserver-example.yaml
```

保存地图：

```
rosrun map_server map_saver -f gmapping-map
```

# 3. Gmapping 整体框架

## 3.1 软件包关系

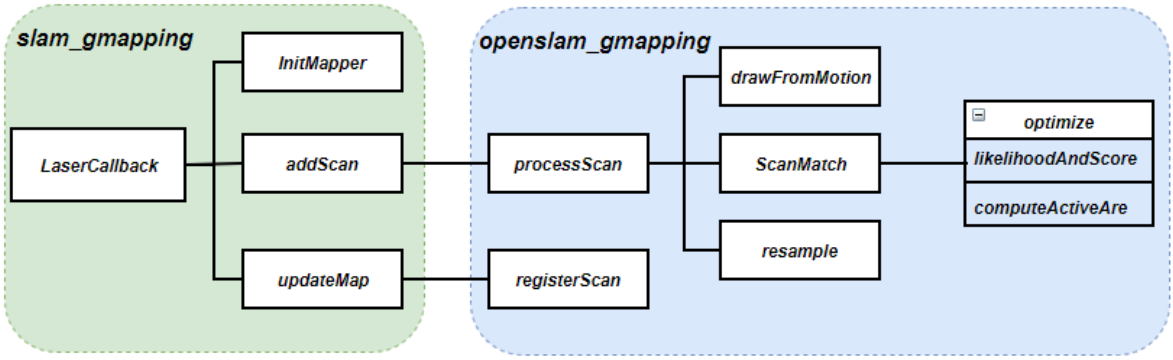
Gmapping 的代码主要包括两个软件包，一个为算法的核心，另外一个是这个算法在 ROS 环境下的封装：

- slam_gmapping，这个软件包是对算法包 ROS 封装，可以从 github 拉取下来，在此软件包内部有分成两个包 gmapping 和 slam_gmapping。其中slam_gmapping 是一个所谓的 meta-package，这是 ROS 系统早期的一个概念，现在已经不再使用了，所以不用管它。 gmapping 才是对 GMapping 算法的ROS封装
- openslam_gmapping，这个软件包就是真正的算法包，可以从 github 拉取下来。它可以独立于 ROS 外运行，建图的主要工作就在这个算法包里

openslam_gmapping 主要负责处理来自 Laser 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 gmapping 代码的核心部分。

而 slam_gmapping 是基于 ROS 的通信机制获取传感器的数据并将他们转换成 gmapping 中定义的格式传递给 openslam_gmapping 处理。可以说是基于 openslam_gmapping 的上层应用。同时也会定期把构建好的地图通过话题的方式发布出去，这里的地图是选择一个最优粒子轨迹上的地图来发布。在获取到理想的机器人位姿后，根据这个这个位姿更新 tf 坐标系转换关系。

软件包的关系如下：




## 3.2 slam_gmapping 代码结构

### 3.2.1 文件结构

```
├── gmapping
|   ├── CHANGELOG.rst
|   ├── CMakeLists.txt
|   ├── launch
|   |   └── slam_gmapping_pr2.launch
|   ├── nodelet_plugins.xml
|   ├── package.xml
|   ├── src                                 # 生成可执行文件<节点>的代码，可以通过 rosrun 或者 roslaunch 的方式启动节点
|   |   ├── main.cpp                        # 节点入口函数，初始化 SLAM 对象
|   |   ├── nodelet.cpp                     # nodelet 插件实现，供其他模块使用
|   |   ├── replay.cpp                      # 以 replay 的方式来 SLAM，就是从 bag 包直接读取传感器数据来建图
|   |   ├── slam_gmapping.cpp               # 对 openslam_gmapping 接口进行 ROS 封装，主要包括 SLAM 初始化，启动建图，发布地图，传感器数据处理等。
|   |   └── slam_gmapping.h
|   └── test                                # 测试程序， 节点入口在 rtest.cpp::main()，可通过各 launch 文件传入参数进行测试
|       ├── basic_localization_laser_different_beamcount.test
|       ├── basic_localization_stage.launch
```

```
|       ├── basic_localization_stage_replay2.launch
|       ├── basic_localization_stage_replay.launch
|       ├── basic_localization_symmetry.launch
|       ├── basic_localization_upside_down.launch
|       ├── rtest.cpp
|       └── test_map.py
├── README.md
└── slam_gmapping                          # 软件包的 meta，没有任何目标文件，可以不需要
    ├── CHANGELOG.rst
    ├── CMakeLists.txt
    └── package.xml
```

文件结构比较简单，源文件也很少，编译完成后最终生成两个可执行文件 “slam_gmapping” 和 “slam_gmapping_replay”， 还有一个支持 nodelet 的动态库 “libslam_gmapping_nodelet.so”。可执行文件可以通过 launch 文件来启动形成一个 ROS 的节点。

### 3.2.2 编译构建

运行示例里已经显示实际用来做 SLAM 相关的节点就是 /slam_gmapping，是整个程序的主体，其他节点都是围绕此节点完成辅助性的工作。这个工程相对比较简单，直接看 gmapping/CMakeList.txt 基本就可以看出工程的构建框架。

这个 CMAKE 是这个工程构建的关键，基本包含里编译这个工程的所有依赖，目标，源文件依赖等。在该文件的一开始指定了 CMake 的版本、工程名称以及依赖项， 这里依赖项居然还有 Boost，从这里可以看出这个代码是比较旧的代码， C++ 高版本还没有发布，有些新的特性还是在 Boost。

```
cmake_minimum_required(VERSION 2.8)
project(gmapping)

find_package(catkin REQUIRED nav_msgs nodelet openslam_gmapping roscpp tf
rosbag_storage)
find_package(Boost REQUIRED)
```

指定生成的目标，SLAM 建图的节点，回放节点，还有就是那个 nodelet 共享库， 这个暂时还没发现是怎么用的

```
add_executable(slam_gmapping src/slam_gmapping.cpp src/main.cpp)
target_link_libraries(slam_gmapping ${Boost_LIBRARIES} ${catkin_LIBRARIES})
if(catkin_EXPORTED_TARGETS)
  add_dependencies(slam_gmapping ${catkin_EXPORTED_TARGETS})
endif()

add_library(slam_gmapping_nodelet src/slam_gmapping.cpp src/nodelet.cpp)
target_link_libraries(slam_gmapping_nodelet ${catkin_LIBRARIES})

add_executable(slam_gmapping_replay src/slam_gmapping.cpp src/replay.cpp)
target_link_libraries(slam_gmapping_replay ${Boost_LIBRARIES}
${catkin_LIBRARIES})
if(catkin_EXPORTED_TARGETS)
add_dependencies(slam_gmapping_replay ${catkin_EXPORTED_TARGETS})
endif()
```

指定安装目标，这里 “CATKIN_PACKAGE_LIB_DESTINATION”， 如果使用 catkin 来编译， 这个变量会进行赋值，就是会指定一个目标路径。如果不是 catkin 来编译， 这个变量就是为空的。意味直接 install 的话目标路径不确定。

```
install(TARGETS slam_gmapping slam_gmapping_nodelet slam_gmapping_replay
  ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
  LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)

install(FILES nodelet_plugins.xml
  DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
)
```

以下为测试程序的编译依赖项，安装目标等

```
if(CATKIN_ENABLE_TESTING)
  find_package(rostest REQUIRED)
  if(TARGET tests)
    add_executable(gmapping-rtest EXCLUDE_FROM_ALL test/rtest.cpp)
    target_link_libraries(gmapping-rtest ${catkin_LIBRARIES} ${GTEST_LIBRARIES})
    add_dependencies(tests gmapping-rtest)
  endif()
  ... // 此处略
endif()
```

## 3.3 openslam_gmapping 代码结构

### 3.3.1 文件结构

```
|openslam_gmapping
├── build_tools                          # 编译配置， Makefile 脚本文件
├── CMakeLists.txt                       # 构建入口
├── grid                                 # 栅格地图美化，添加边缘，栅格链表管理，地图测试程序等
|   ├── graphmap.cpp
|   ├── Makefile
|   └── map_test.cpp
├── gridfastslam                         # 建图引擎，Gmapping 核心模块，实现对激光数据滤波，粒子树的管理。封装了传感器数据读写器，读写器运行于线程池
|   ├── gfs2stream.cpp
|   ├── gfsreader.cpp
|   ├── gridslamprocessor.cpp
|   ├── gridslamprocessor_tree.cpp
|   └── motionmodel.cpp
├── gui                                  # 可视化工具，实时监控建图过程中整个环境变化。比如监测粒子变化，实时显示更新地图，动态显示粒子变化轨迹
|   ├── Makefile
|   ├── qgraphpainter.cpp
|   ├── qmappainter.cpp
|   ├── qnavigatorwidget.cpp
|   ├── qparticleviewer.cpp
|   ├── qpixmapdumper.cpp
|   └── qslamandnavwidget.cpp
├── include                              # 算法对外接口对应的头文件
```

```
|   └── gmapping
├── particlefilter                          # 更新粒子的状态，更新粒子的权重，粒子重采样
|   ├── particlefilter.cpp
|   ├── particlefilter_test.cpp
|   └── range_bearing.cpp
├── scanmatcher                             # 扫描匹配器，预测机器人位姿时，参考激光传感器的扫描数据，达到改进粒子滤波器中的建议分布的目的。
|   ├── eig3.cpp
|   ├── icptest.cpp
|   ├── Makefile
|   ├── scanmatcher.cpp
|   ├── scanmatcher.new.cpp
|   ├── scanmatcherprocessor.cpp
|   ├── scanmatch_test.cpp
|   └── smmap.cpp
├── sensor                                  # 传感器读取器，读取器在这里做了一次抽象，封装了基类定义操作的方法。Gmapping 只处理激光和里程计数据
└── utils                                   # 工具类
    ├── autoptr_test.cpp
    ├── Makefile
    ├── movement.cpp
    ├── printmemusage.cpp
    ├── stat.cpp
    └── stat_test.cpp
```

主要的建图算法功能实现是在 gridfastslam （代码内部有简称为 gfs ）里实现，这里是建图引擎的核心代码。其他文件都是些辅助性的功能，我们后面章节源码分析主要集中在 gfs，整个工程最终编译生成“libgridfastslam.so”，“libscanmatcher.so”，“libsensor_odometry.so” 等动态库。Gmapping 建图节点运行时会加载使用这些库。

### 3.3.2 编译构建

此工程支持两种构建方式，Makefile-Configure 和 Cmake。由于 Makefile 语法比较老旧，复杂，可读性不是很好，所以这里只分析 Cmake 的构建方式。整个工程的目标文件是生成共享库，供其他可执行文件引用。其中有部分源文件没有编译，在工程源文件里没有使用到，或者有些是废弃的实现。整个工程只有一个CMakeLists.txt，在这个 Cmake 里定义了所有的源文件依赖。这个工程代码可能比较老旧了，很多 Cmake 的写法都不够现代，后续有很多可改造的空间。

```
cmake_minimum_required(VERSION 2.8.3)
project(openslam_gmapping)

find_package(catkin REQUIRED)

catkin_package(
 INCLUDE_DIRS include
 LIBRARIES utils sensor_base sensor_odometry sensor_range log configfile
scanmatcher gridfastslam
)
```

这里的 “catkin_package” 声明了两点，这个软件包对外可提供的接口是在 “include” 文件下的头文件和库文件有 “utils”，“scanmatcher”，“gridfastslam” 等，其他软件包同样使用 catkin 来编译构建的话，可以直接用 “find_package” 的方式来找到这些库及头文件。

```
add_library(gridfastslam
 gridfastslam/gridslamprocessor_tree.cpp
 gridfastslam/motionmodel.cpp
 gridfastslam/gridslamprocessor.cpp
 gridfastslam/gfsreader.cpp)

add_executable(gfs2log
 gridfastslam/gfs2log.cpp)

target_link_libraries(gfs2log gridfastslam)

target_link_libraries(gridfastslam
 scanmatcher log sensor_range sensor_odometry sensor_base utils)
```

配置了此工程最为核心的输出库 “gridfastslam” 依赖的源文件，同时这个还依赖于各个子库 “scanmatcher”，“sensor_range”，“sensor_base” 等，这种写法其实不是很好，库文件变的很零散，不易于维护。应该直接静态链接到 “libgridfastslam.so” 更适合，而不是动态库依赖于动态库的方式。

```
install(TARGETS utils autoptr_test sensor_base sensor_odometry sensor_range
sensor_range log log_test log_plot scanstudio2carmen rdk2carmen configfile
configfile_test scanmatcher scanmatch_test icptest gridfastslam gfs2log gfs2rec
gfs2neff
 ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
 LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
 RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)

install(DIRECTORY include/gmapping
 DESTINATION ${CATKIN_GLOBAL_INCLUDE_DESTINATION}
 FILES_MATCHING PATTERN "*.h*"
 PATTERN ".svn" EXCLUDE
)
```

定义了输出文件的安装目标路径，这里把所有的目标文件都安装的目标路径了，这里还特地实现 autoptr，但是在 C++11 已经有对应的实现了。一共有20几个目标文件包含动态库和可执行文件，很零碎分散，使用者非常不方便。

这里同时把软件包需要开放出去的头文件也指定了安装目标路径，同时排除了 ".svn" 等不需要的文件，只输出 ".h" 的头文件。

# 4. 源码分析

如 “3.1 软件包关系” 所述，进程的入口函数订阅了激光扫描、地图等话题，获取到的激光数据后首先通过消息过滤器，实现激光扫描数据的订阅以及tf、frame变换。而实际的建图功能是由传感器数据驱动，传感器数据是通过 “laserCallback” 这个 ROS 的话题回调函数来处理激光数据。以此为入口分析整个数据流向，最终输出栅格地图数据。下面章节主要分析建图中的重要模块，有些非关键模块可以参考源码去了解。Gmapping 在 ROS 环境下有以下订阅发布的话题和支持的服务接口。

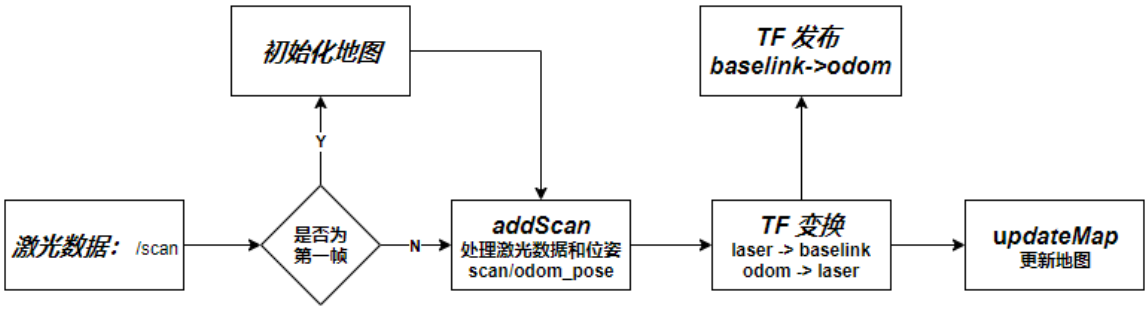
| | 名称 | 类型 | 描述 |
| --- | --- | --- | --- |
| 订阅话题 | tf | tf/tfMessage | 机器人基坐标系，激光坐标系，里程计坐标系，世界坐标系间转换 |
| 订阅话题 | scan | sensor_msgs/LaserScan | 激光雷达扫描数据 |
| 发布话题 | map_metadata | nav_msgs/MapMetaData | 发布地图描述数据 |
| 发布话题 | map | nav_msgs/OccupancyGrid | 发布栅格地图数据，包含地图原始位置，姿态 |
| 发布话题 | ~entropy | std_msgs/Float64 | 发布机器人姿态发布熵的估计 |
| 服务 | dynamic_map | nav_msgs/GetMap | 获取栅格地图数据接口 |

## 4.1 激光数据入口

传感器数据的入口是在消息回调函数 “SlamGMapping::laserCallback”，每当其他节点发布了传感器数据能够由 TF 转换到目标坐标系时，就会调用此回调函数，并把激光数据以参数的形式传入。此回调函数主要完成了以下功能：

- 判断激光数据是否为开始建图后的第一帧，当为第一帧时初始化地图，并修改初始化标志位
- 添加激光数据，添加成功后执行下一步，添加失败忽略此帧激光
- 通过最佳粒子获取位姿，得到激光坐标到基坐标系转换关系，里程计坐标到激光坐标系转换关系，从而计算出基坐标到里程计坐标系转换关系并更新到 TF
- 地图更新间隔超过阀值后，更新地图

处理流程如下：




对激光数据帧数计数，throttle_scans_ 为配置文件配置的参数，每当接收到 throttle_scans_ 个激光扫描数据就进行一次数据处理。同时初始化了地图更新时间戳，来记录上次地图发布时间，控制地图更新频率。

```
laser_count_++;
if ((laser_count_ % throttle_scans_) != 0)
 return;

static ros::Time last_map_update(0,0);
```

在第一次获得扫描数据时，完成建图器的初始化操作。它根据成员变量 got_first_scan_ 来判定是否获得了第一个扫描数据，一旦获得了扫描数据，就会进行建图器的初始化并将该变量置为true， 之后将不再调用函数 initMapper。

```
if(!got_first_scan_)
 {
  if(!initMapper(*scan))
   return;
  got_first_scan_ = true;
 }
```

通过 addScan 把激光测量数据报告建图引擎 gsp_，内部通过 gsp 对激光数据处理完毕后，输出里程计坐标系下的位姿估计

```
GMapping::OrientedPoint odom_pose;
 if(addScan(*scan, odom_pose))
 {
  ROS_DEBUG("scan processed");
  ...//
}else
  ROS_DEBUG("cannot process scan");
```

如果能够成功添加数据，将进一步的计算地图到里程计的坐标变换并更新地图。首先从粒子集合中挑选最优的粒子，以其地图坐标为基准计算从激光雷达到地图之间的坐标变换。 同时计算从里程计到激光雷达的坐标变换。

```
  GMapping::OrientedPoint mpose = gsp_->getParticles()[gsp_-
>getBestParticleIndex()].pose;
  ROS_DEBUG("new best pose: %.3f %.3f %.3f", mpose.x, mpose.y, mpose.theta);
  ROS_DEBUG("odom pose: %.3f %.3f %.3f", odom_pose.x, odom_pose.y,
odom_pose.theta);
  ROS_DEBUG("correction: %.3f %.3f %.3f", mpose.x - odom_pose.x, mpose.y -
odom_pose.y, mpose.theta - odom_pose.theta);

  tf::Transform laser_to_map = tf::Transform(tf::createQuaternionFromRPY(0, 0,
mpose.theta), tf::Vector3(mpose.x, mpose.y, 0.0)).inverse();
  tf::Transform odom_to_laser = tf::Transform(tf::createQuaternionFromRPY(0, 0,
odom_pose.theta), tf::Vector3(odom_pose.x, odom_pose.y, 0.0));
```

接着根据刚刚计算的变换导出从地图到里程计的坐标变换。在更新的时候对 map_to_odom_ 加锁了，以保证更新过程是原子的。map_to_odom_ 是类成员变量，机器人基坐标与里程计坐标系转换管理会通过 SlamGMapping::publishLoop 由 TF 发布出去，其他节点可以用这个转换关系。

```
 map_to_odom_mutex_.lock();
 map_to_odom_ = (odom_to_laser * laser_to_map).inverse();
 map_to_odom_mutex_.unlock();
```

根据配置参数 map_update_interval_ ，定时更新地图，更新后会用话题发布的方式到其他节点。

```
  if(!got_map_ || (scan->header.stamp - last_map_update) > map_update_interval_)
  {
   updateMap(*scan);
   last_map_update = scan->header.stamp;
   ROS_DEBUG("Updated the map");
  }
```

## 4.2 初始化建图

上一节中我们知道，在收到第一帧激光数据时，会进行建图引擎初始化。使得配置的参数在建图引擎对象 gsp_ 上生效，同时建图引擎在这绑定了传感器的对象，可以通过传感器对象来读取传感器传感器原始数据，初始化主要完成了以下功能：

- 校准激光雷达水平安装
- 获取激光雷达扫描中心点，同时判定激光是正向扫描还是反向扫描
- 绑定传感器对象，包括激光传感器，里程计传感器
- 设置建图参数，建图范围，角度步长，噪声标准差，粒子滤波器参数等

在机器人坐标系下，在雷达传感器的正上方 1M 处构建一个点，然后将它转换到激光雷达的坐标系下，构建的这个点将被用来判定传感器的安装方式。GMapping要求激光雷达一定是水平安装的，也就是说雷达的 Z 轴要么朝上，要么朝下。如果传感器是水平安装的， 那么从机器人坐标系下转换到传感器坐标系下，该点的 Z轴数据应当接近为1或者-1。如果不是水平安装，则提示错误，要求用户检查雷达是否水平安装与机器人。

```
 tf::Vector3 v;
 v.setValue(0, 0, 1 + laser_pose.getOrigin().z());
 tf::Stamped<tf::Vector3> up(v, scan.header.stamp, base_frame_);

 try {
  tf_.transformPoint(laser_frame_, up, up);
  ROS_DEBUG("Z-Axis in sensor frame: %.3f", up.z());
 } catch(tf::TransformException& e) {
  ROS_WARN("Unable to determine orientation of laser: %s", e.what());
  return false;
 }

 if (fabs(fabs(up.z()) - 1) > 0.001) {
  ROS_WARN("Laser has to be mounted planar! Z-coordinate has to be 1 or -1, but
gave: %.5f", up.z());
  return false;
 }
```

从传感器消息中获取激光扫描数据量和中间角度，据刚刚判定的传感器安装方式以及中间角度，来初始化传感器的中心位置 centered_laser_pose。另外扫描的最小角度和最大角度了说明扫描方向， 如果扫描反向，那么在以后的数据处理过程中，将以相反的顺序处理扫描数据。这里用变量 do_reverse_range 来记录这一特性，变量 do_reverse_range 在处理激光数据时会用到，这里仅仅算出这个标志。"up.z() <= 0" 是属于雷达倒着安装的情况。

```
 double angle_center = (scan.angle_min + scan.angle_max)/2;
 if (up.z() > 0)
 {
  do_reverse_range_ = scan.angle_min > scan.angle_max;
  centered_laser_pose_ = tf::Stamped<tf::Pose>
(tf::Transform(tf::createQuaternionFromRPY(0,0,angle_center),
                                tf::Vector3(0,0,0)), ros::Time::now(),
laser_frame_);
  ROS_INFO("Laser is mounted upwards.");
 }
 else
 {
  do_reverse_range_ = scan.angle_min < scan.angle_max;
```

```
  centered_laser_pose_ = tf::Stamped<tf::Pose>
(tf::Transform(tf::createQuaternionFromRPY(M_PI,0,-angle_center),
                                   tf::Vector3(0,0,0)), ros::Time::now(),
laser_frame_);
  ROS_INFO("Laser is mounted upside down.");
 }
```

据扫描的数据长度和增长步长初始化雷达的扫描角度表 laser_angles，该表中的角度都是从小到大增长，并且差值都是一样的，就是激光数据里的角度步长。

```
 laser_angles_.resize(scan.ranges.size());
 double theta = - std::fabs(scan.angle_min - scan.angle_max)/2;

 for(unsigned int i=0; i<scan.ranges.size(); ++i)
 {
  laser_angles_[i]=theta;
  theta += std::fabs(scan.angle_increment);
 }
```

设置各项建图参数，可以参考 2.1.1 节相关的参数项，都是在这段代码里配置到建图引擎 gsp。

```
gsp_->setMatchingParameters(maxUrange_, maxRange_, sigma_, kernelSize_, lstep_,
astep_, iterations_,lsigma_, ogain_, lskip_);
 gsp_->setMotionModelParameters(srr_, srt_, str_, stt_);
 gsp_->setUpdateDistances(linearUpdate_, angularUpdate_, resampleThreshold_);
 gsp_->setUpdatePeriod(temporalUpdate_);
 gsp_->setgenerateMap(false);
 gsp_->GridSlamProcessor::init(particles_, xmin_, ymin_, xmax_, ymax_, delta_,
initialPose);
 gsp_->setllsamplerange(llsamplerange_);
 gsp_->setllsamplestep(llsamplestep_);
 gsp_->setlasamplerange(lasamplerange_);
 gsp_->setlasamplestep(lasamplestep_);
 gsp_->setminimumScore(minimum_score_);
```

## 4.3 位姿预估

Gmapping 是通过粒子滤波的方式来推导出最优位姿并构建地图。粒子滤波原理是通过状态方程估算出状态的变化，同时有个观测方程得到观测值来修正估计值，最终得到一个修正后的状态。状态方程类似如下：

$$\hat{x}_t^- = F * \hat{x}_{t-1} + B * u_{t-1} + e_t$$

$\hat{x}_t^-$ 表示 t 时刻的状态预估，$u_{t-1}$ 为控制量，这里指的是里程计变化。$F$ 是状态转移函数，描述了机器人状态从 $t-1$ 时刻下的状态，经过控制量 $B * u_{t-1}$ 的作用后，转变为 $t$ 时刻的机器人状态。同时在状态转换过程中，需要接受一部分测量噪声 $e_t$ 。在 Gmapping 里，里程计模型的噪声为正太分布的噪声分布，从代码实现上看也可以看到，认为是一种近似正太分布噪声。

位姿估计是封装在一个运动模型类实现，里面实现了基于运动模型的预测方法。这里有两种预测方法，一种基于线速度和角速度，一种是基于里程计位姿，在 Scan 处理函数里使用的是基于里程计位姿的预测方法。就是下面第二个方法，其输入为当期粒子的位姿，里程计当前的位姿，上一个里程计位姿，输出为预估后的位姿。

```
struct MotionModel{
  OrientedPoint drawFromMotion(const OrientedPoint& p, double linearMove, double
angularMove) const;
  OrientedPoint drawFromMotion(const OrientedPoint& p, const OrientedPoint&
pnew, const OrientedPoint& pold) const;
  Covariance3 gaussianApproximation(const OrientedPoint& pnew, const
OrientedPoint& pold) const;
  double srr, str, srt, stt;
};
```

针对每个粒子预估出一个新的位姿，这里通过传引用的方式把 it->pose 给 pose，最终预估出来的位姿存到 pose 后，其实就是更新了 it->pose。

```
  for (ParticleVector::iterator it=m_particles.begin(); it!=m_particles.end();
it++){
   OrientedPoint& pose(it->pose);
   pose=m_motionModel.drawFromMotion(it->pose, relPose, m_odoPose);
  }
```

在预估的位姿中，通过工具方法 sampleGaussian 添加了噪声，通过对正太进行采样，为里程计添加噪声，这个噪声会反映到输出的位姿上，此位姿就认为是该方法的返回值，就是完成对粒子位姿的预估。

## 4.4 扫描匹配

建图过程中，扫描匹配的作用是，在预测机器人位姿的时候，参考激光扫描数据，改进粒子滤波器中粒子分布。就是对激光数据与粒子的位姿匹配，匹配是否成功执行不同的行为。Gmapping 建图引擎 gsp 定义了成员变量 m_matcher，它就是一个匹配器，所有激光匹配相关的工作都在这个类里完成。

通过调用 optimize() 函数，并且返回寻优后的得分。先寻找最优位姿，为使位姿更为准确，通过在初始位姿的基础上，通过爬山算法向各向移动，查找更高得分的位姿，作为最优位姿。

以初始位姿为基础，向六个方向分别爬山计算得分，爬山算法是一种局部寻优的算法， 它贪婪的与周围邻接的节点进行比较，并以其中取得最大值的节点作为下一次迭代的起点，直到周围邻接的节点中没有比当前节点更大时结束迭代。整个过程就像爬山一样，一步步向山顶靠近。

```
  double adelta=m_optAngularDelta, ldelta=m_optLinearDelta;       // 爬山步长
  unsigned int refinement=0;
  enum Move{Front, Back, Left, Right, TurnLeft, TurnRight, Done}; // 爬山范围，这里
基于初始位姿的六个方向
```

以下为爬山算法过程，使用的是一种有限状态机的机制来遍历邻接节点的。在运行状态机之前，先用临时变量 localPose 记录下当前考察节点。然后在 switch 语句中根据move所指的爬山方向，修改 localPose， 同时指定下一个遍历循环中将要考察的爬山方向。在该状态机中，依次访问 currentPose 的前、后、左、右、左转、右转6个邻接节点。

```
    do {
      localPose=currentPose;
      switch(move){
        case Front: localPose.x+=ldelta; move=Back; break;
        case Back: localPose.x-=ldelta; move=Left; break;
        case Left: localPose.y-=ldelta; move=Right; break;
        case Right: localPose.y+=ldelta; move=TurnLeft; break;
        case TurnLeft: localPose.theta+=adelta; move=TurnRight; break;
        case TurnRight: localPose.theta-=adelta; move=Done; break;
        default:;
      }
    // 此处略
    } while(move!=Done)
```

完成粒子的采样和权值计算之后，我们就需要更新各个粒子所记录的地图。实际上函数 computeActiveArea 并没有完成对地图的更新，它只是调整了地图的大小，并记录下需要更新的区域。这个函数的功能很简单，但是源码写的十分冗长，这里不再浪费篇幅展开细讲。

下面只给出伪代码化的 computeActiveArea 。该函数有三个参数，其中 map 是需要更新的地图， 在该函数正常退出之后，map 将记录下需要更新的区域。p是粒子所记录的机器人状态，readings 则记录了激光传感器的扫描值。

```
void ScanMatcher::computeActiveArea(ScanMatcherMap& map, const OrientedPoint& p,
const double* readings){
  if (m_activeAreaComputed)
    return;
  OrientedPoint lp=p;
  lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
  lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
  lp.theta+=m_laserPose.theta;
  IntPoint p0=map.world2map(lp);
  // 此处略
}
```

实际更新地图的操作是放在了重采样之后进行的。因为重采样过程中， 会删除一些权重较小的粒子，如果我们还对这样的粒子更新地图，就有些浪费计算资源了。我们在计算了激活态的区域之后，就可以调用函数 registerScan 来将扫描的占用信息实际写到地图中。 下面是函数 registerScan 的代码片段，它的参数列表与 computeActiveArea 一样，需要输入待更新的地图、机器人位姿和激光传感器的读数。

```
double ScanMatcher::registerScan(ScanMatcherMap& map, const OrientedPoint& p,
const double* readings){
  if (!m_activeAreaComputed)
    computeActiveArea(map, p, readings);
  //this operation replicates the cells that will be changed in the registration
operation
  map.storage().allocActiveArea();
  OrientedPoint lp=p;
  lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
  lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
  lp.theta+=m_laserPose.theta;
  IntPoint p0=map.world2map(lp);
  // 此处略
}
```

## 4.5 重采样

重采样函数是为了消除早期 SIS 粒子滤波器的粒子退化问题。 其基本思想是对赋予权重的粒子集合进行重新采样，从中取出权重较小的粒子，增加权重较大的粒子。重采样操作之后，所有的粒子权重都会被设置为一样的。虽然，这一操作成功地解决了粒子退化问题，但它带来了一种所谓的粒子匮乏地问题，随着迭代次数的增加，粒子的多样性在下降。为了缓解重采样的粒子退化问题，GMapping 采取了一种自适应的方式进行重采样。它定义了一个指标 Neff 来评价粒子权重的相似度，只有在 Neff 小于一个给定的阈值的时候才进行重采样。其背后的思想是，粒子的权重差异越小， 得到的粒子则更接近于对目标分布的采样。

- 根据 Neff 判断是否进行重采样，这里 m_resampleThreshold 是配置信息参数，低于此阀值就执行重采样

```
if (m_neff<m_resampleThreshold*m_particles.size()){
  if (m_infoStream)
   m_infoStream << "**************RESAMPLE***************" << std::endl;
// 此处略
}
```

- 对选取的粒子执行重采样，增加的粒子填到变量 temp， 需要剔除的粒子填到变量 deletedParticles， 最后根据这两个变量更新粒子集 m_particles

```
  uniform_resampler<double, double> resampler;
  m_indexes=resampler.resampleIndexes(m_weights, adaptSize);

  ParticleVector temp;
  unsigned int j=0;
  std::vector<unsigned int> deletedParticles;   //this is for deleteing the
particles which have been resampled away.
  for (unsigned int i=0; i<m_indexes.size(); i++){
   while(j<m_indexes[i]){
   deletedParticles.push_back(j);
   j++;
   }
// 此处略
   node->reading=reading;
   temp.push_back(p);
   temp.back().node=node;
   temp.back().previousIndex=m_indexes[i];
  }
```

- 更新粒子地图与轨迹

```
   m_matcher.invalidateActiveArea();
   m_matcher.registerScan(it->map, it->pose, plainReading);
   it->previousIndex=index;
```

## 4.6 地图更新发布

Gmapping 是基于粒子滤波器来构建栅格地图，发布的地图是从最优粒子中取出地图数据，算出栅格地图中各个栅格单元的占用/闲置概率。该模块输入参数为当期时刻激光数据和所有粒子数据集合。完成以下主要功能：

- 找到最优粒子

- 从最优粒子节点中，用当前激光数据帧匹配出地图位姿，并输出地图数据
- 从地图原始数据构建出栅格地图
- 对栅格地图进行修剪，输出闲置/占用/未知各区域清晰的地图
- 通过 ROS 话题发布，把地图发布给其他节点

构建扫描匹配器，扫描匹配器通过激光数据，当期激光位姿，激光有效测距数据来构建。此扫描匹配器在后面匹配节点中用到

```
 GMapping::ScanMatcher matcher;
 matcher.setLaserParameters(scan.ranges.size(), &(laser_angles_[0]), gsp_laser_-
>getPose());

 matcher.setlaserMaxRange(maxRange_);
 matcher.setusableRange(maxUrange_);
 matcher.setgenerateMap(true);
```

获取最优粒子，获取最优粒子的实现也在 gsp 引擎内部，它是根据粒子的权重来比较，权重最大的粒子就是最优粒子。最优的粒子随着建图推进过程中，会有变化，甚至有些粒子会移出粒子群。

```
 GMapping::GridSlamProcessor::Particle best =
     gsp_->getParticles()[gsp_->getBestParticleIndex()];
```

遍历最优粒子的轨迹树，根据每个节点数据来优化地图，在这里扫描匹配器起了关键性的作用，它的内部比较 gsp_laser_->pose 与 n->pose 来匹配出更有的 smap。

```
 for(GMapping::GridSlamProcessor::TNode* n = best.node; n; n = n->parent)
 {
  ROS_DEBUG(" %.3f %.3f %.3f", n->pose.x, n->pose.y, n->pose.theta);
  if(!n->reading)
  {
   ROS_DEBUG("Reading is NULL");
   continue;
  }
  matcher.invalidateActiveArea();
  matcher.computeActiveArea(smap, n->pose, &((*n->reading)[0]));
  matcher.registerScan(smap, n->pose, &((*n->reading)[0]));
 }
```

初始化的地图范围可能不够，随着探索区域的增加，需要对地图范围扩容，调整地图的参数，宽度，长度，起始/终点位置等

```
if(map_.map.info.width != (unsigned int) smap.getMapSizeX() ||
map_.map.info.height != (unsigned int) smap.getMapSizeY()) {
  // 此处略
  ROS_DEBUG("map origin: (%f, %f)", map_.map.info.origin.position.x,
map_.map.info.origin.position.y);
 }
```

遍历地图中所有栅格，这里两层的 for 循环来遍历栅格。再根据占用概率阈值 occ_thresh 把地图划分为占用区域(对应栅格值为100)，空闲区域(对应栅格值为0)，以及未知区域(对应栅格值为-1)。这里的实现跟 Cartographer 有点不一样，Gmapping 这样处理使得地图看起来要清晰，不会有很多灰色区域。

```
 for(int x=0; x < smap.getMapSizeX(); x++)
 {
```

```
  for(int y=0; y < smap.getMapSizeY(); y++)
  {
   GMapping::IntPoint p(x, y);
   double occ=smap.cell(p);
   assert(occ <= 1.0);
   if(occ < 0)
    map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = -1;
   else if(occ > occ_thresh_)
   {
    map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = 100;
   }
   else
    map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = 0;
  }
 }
```

最后通过 ROS 话题的方式把地图数据发布出去，“sst.publish(map.map)”。

# 5.总结

本文主要讲述了在 Gmapping 在 ROS 环境下的运行机制，代码的整体架构。通过此文，可以了解数据流的处理流程，同时对 Gmapping 建图的原理有个整体的认识。