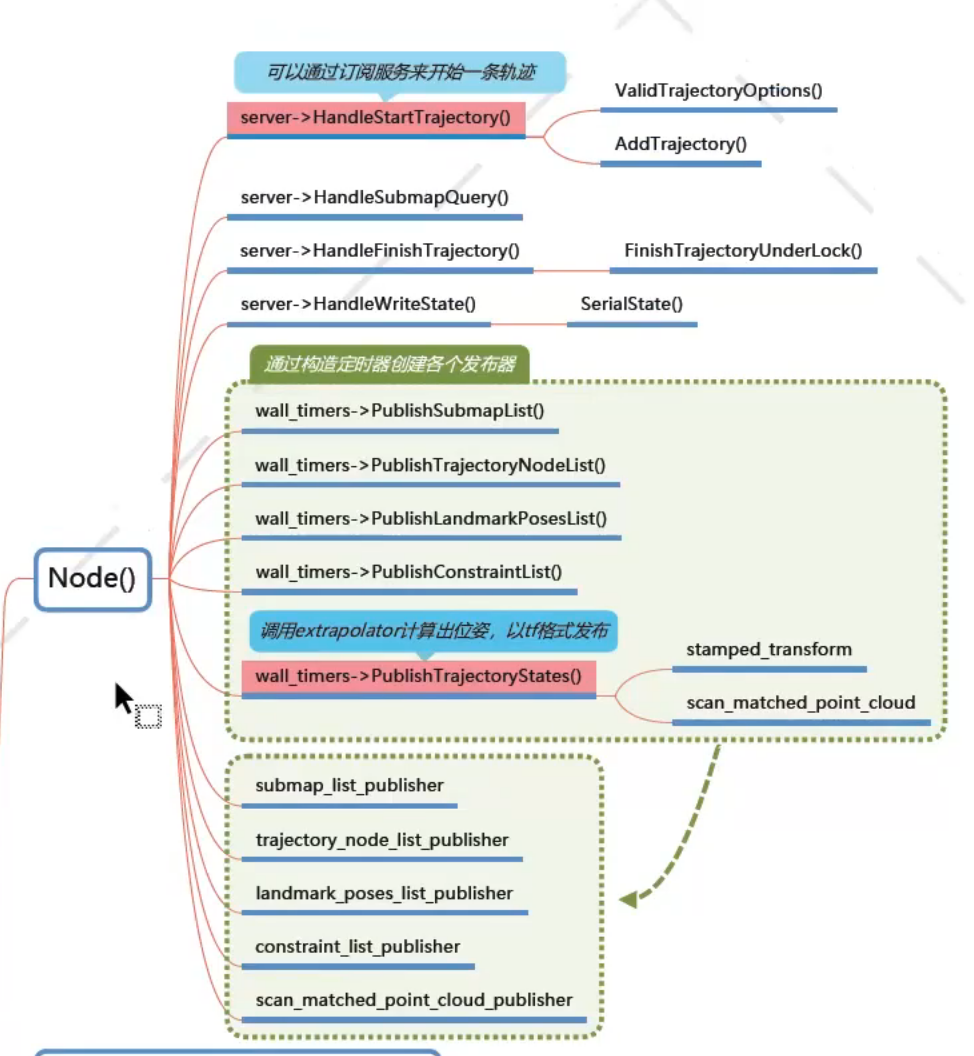
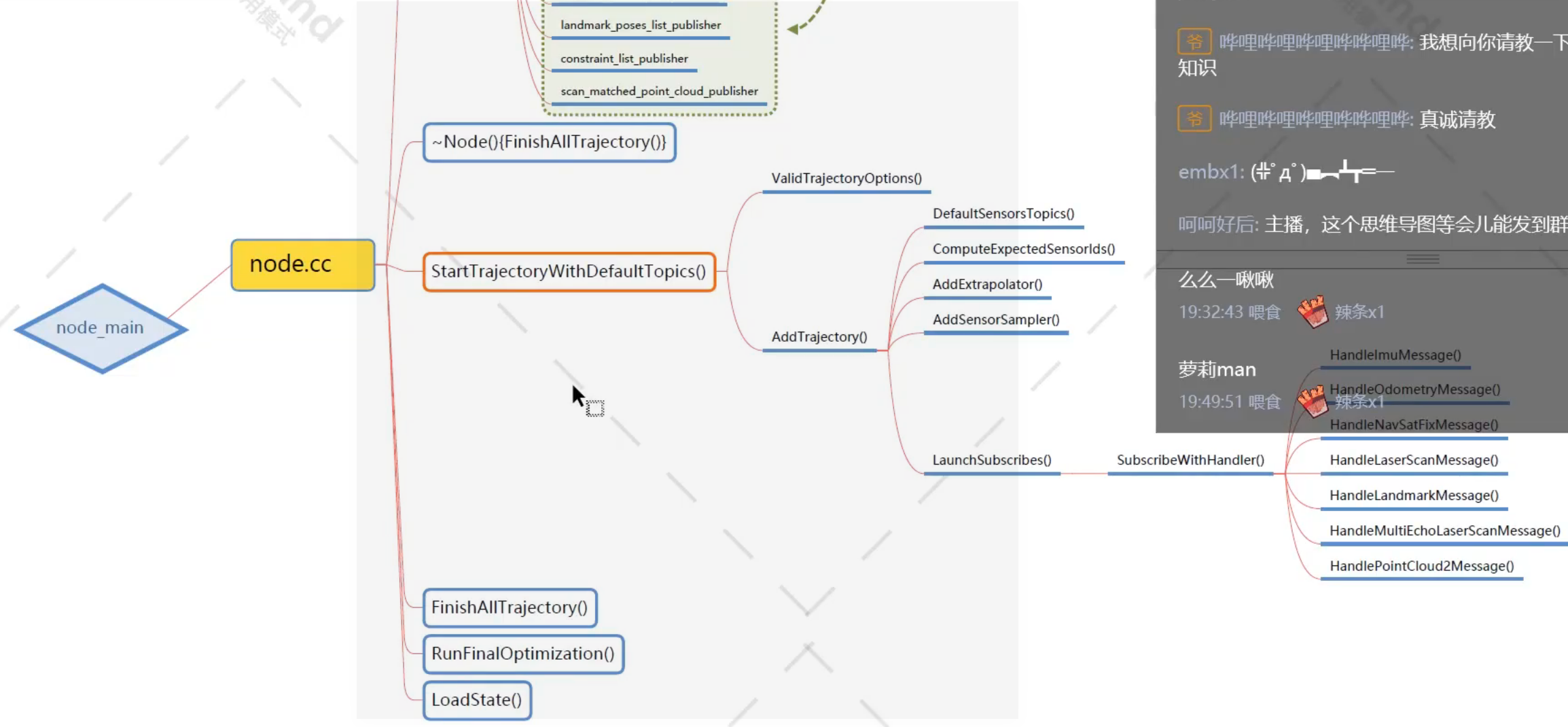
main函数和run，注意区别  
run里面，创建了一个tf缓存，一个tf监听器，实例了一个node options，以及一个trajectory options  
trajectory options里面定义了使用什么传感器，坐标系定义，采样频率等设置，它跟lua文件是对应的。  
  
接下来，createTrajectoryOptions有两个版本，两个参数的版本是多了一个初始轨迹位资，说明其有历史地图，有历史地图时可使用此版本  
  
下面两个函数是fromROSMessage和toROSMessage，也就是说这些配置除了用lua输入，还可以用ros消息来输入。但是现在用的不多。  
  
接下来回到run函数，我们看看node\_options干了什么  
  
map\_builder\_options比较重要，用它构建了一个map\_builder类；下面的超时时间什么的就不是很重要了。  
下面有两个函数。createNodeOptions是从lua文件创建node\_options；而loadOptions是解析加载的lua文件，并把trajectory和node options中的各个选项分离好。  
  
以上，nodeoptions和trajectory options就做了以上几件简单的事情。那接下来我们回到run：  
接下来是loadOptions，然后用map\_builder\_options创建了map\_builder类，前面提到过。  
  
接下来就是大大大大bos了，即创建了Node类的对象node，构造参数是node\_options、map\_builder以及tf\_buffer。  
下面的两句是两个if：  
loadState主要是看要不要加载历史地图  
下面的default topic是要不要使用默认tos topic名称。如果不是默认的，则要进行重映射。  
  
接下来spin，进入回调循环  
  
接下来两行分别是：  
把所有轨迹结束掉  
进行一次最终的优化  
  
接下来，如果要保存地图，则调用serielizeState，进行序列化，保存。  
  
所以，我们会发现，在这个主函数里，根本就没有发现任何传感器消息的影子。而且Node里面其实也没有进行任何实质性的操作。要做的事情是通过创建一个mapBuiler的实例，在这个里面才有功能的实现。甚至有些功能在mapBuiler里面依然没有实现，而是需要创建sensor\_bridge的实例来实现。这就是carto里面无处不在的功能封装。  
  
分隔符  
  
接下来看node，文件是node.cc  
这个node. cc七八百行，要命

Node一定要跟着思维导图来看



首先，结合上图，看看Node的构造函数都做了什么事。首先是构造了list的五个发布器，以及4个服务；每个服务都是调用一个Handle函数去执行，又有一种封装的味道。然后构造了5个定时器，并在五个定时器内构造了5个发布器，是一一对应的。



下面这几个函数都是在main里面直接调用的

还有几个小函数就没写，它们是在offline版本下调用的。

下面重点看另一个函数，即StartTrajectoryWithDefaultTopics，这个函数做了很多。里面主要有两个函数，Valid那个是验证options是否正确，下面这个AddTrajectory比较厉害。它做了以下几件事情：首先，得到sensor的默认topic；

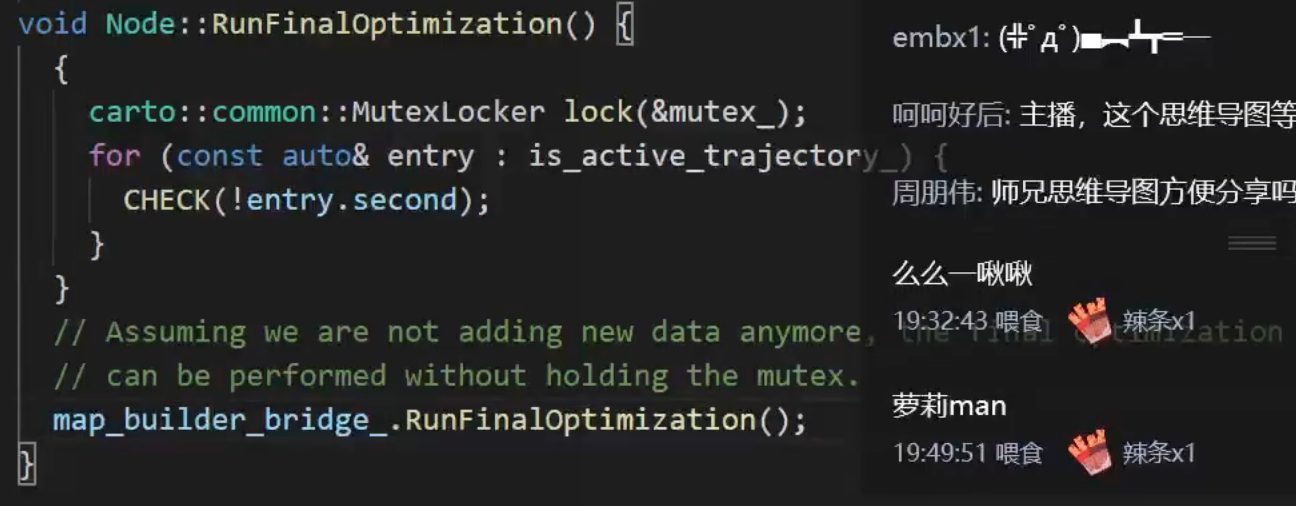
接下来，ComputeExpectedSensorIds()，看看传感器是否需要额外id（比如，用到了两个激光雷达，那么就会在topic后面加\_1, \_2），以示区别；第三，是AddExtrapolator()，位姿推断器；第四，AddSensorSampler()，采样，三和四不太要紧；第五比较厉害，SubscribeWithHandler()，这个函数才是处理所有传感器的总的函数！！！分别是IMU、Odometry、GPS、单线雷达、LandMark、多线雷达、PointCloud2。

至此，整体逻辑就梳理清楚了

注意理解这句话：通过调用服务来开始一段轨迹

只要认真梳理程序是怎么跳转的，就会发现，还是挺清楚的！

回到刚才，为什么说其实Node里面什么也没做呢？举一个例子



可以看到Node里的RunFinalOptimization其实就是map\_builder\_bridge\_里面的RunFinalOptimization。这不就是封装么！

再比如，自己想不依赖ROS做SLAM的话，根本不需要通过订阅服务的方式去调函数。

接下来，讲讲map\_builder\_bridge\_来讲一讲。

其实它也就是为了创建map\_builder\_bridge类，它是Cartographer的第一把钥匙。

先看构造函数，函数体是空的，它真正的目的是为了构造一个map\_builder（用的是移动构造std::move()）。新版的Carto又给它做了一个封装，即MapBuilderInterface，一层套一层。

再举一个例子，是AddTrajectory，它也是调用了map\_builder\_的AddTrajectoryBuilder函数。这里要注意，Trajectory可以理解为一次SLAM。还有一些需要继续往下游的，交给sensor\_bridge\_。总结就是map\_builder\_bridge里面做的很多事情其实是Node里相关事情的进一步处理，它里面的函数，Node里都有。

接下来，看sensor\_bridge。这个类写的非常非常地工整。Sensor\_bridge有三个文件和它非常紧密。一个是time\_conversion，负责时间格式的转换，ROS和Carto之间格式不一样，需要转，Message Conversion用于数据转换，TF bridge用于坐标变换。（这里面微微有点瑕疵，按理所有的转换都应该在Message conversion里面完成，但是它把ToIMUData和ToOdometryData写在了sensor\_bridge里面，而WGS84，ECEF等依然写在message\_conversion里面，这个作者不是很理解）

sensor\_bridge是这样做的，以hanleIMUMessage为例，这个函数它做两件事情，首先调用ToImuData转一下IMU格式，然后就是把IMU数据加到算法里去处理，这里就会用到进入carto的第二把钥匙，就是trajectory\_builder。

下面重点讲一下激光相关的。和激光有关的函数有五个，需要好好理一下。

1. 先说HandleLaserScanMessage。先调用ToPointCloudIntensities，这就是我写的msg里面关于点云的格式转换里面的。接下来调用了HandleLaserScan。
2. HandleMultiEchoLaserScanMessage。多回波雷达，也是先转格式，然后调用HandleLaserScan。也就是HandleLaserScan其实是HandleLaserScanMessage和HandleMultiEchoLaserScanMessage的一个汇总，两者转完格式之后都要调用HandleLaserScan
3. 接下来是HandlePointCloud2，先调用了pcl的一些东西转了转格式，然后调用HandleRangefinder。
4. HandleLaserScan和HandleRangefinder。查看代码发现，HandleLaserScan在最后也调用了HandleRangefinder
5. HandleRangefinder调用了钥匙：trajectory\_builder里面的AddSensorData。

现在关于激光这块我们已经理清楚了。我们实际往Carto里面加东西实际调的是HandleRangefinder。HandleLaserScanMessage和HandleMultiEchoLaserScanMessage抽象在一起，是HandleLaserScan。HandleLaserScan和HandlePointCloud2是平行关系，它们都调用HandleRangefinder。

好处是什么呢？是它可以把激光以及和测距原理类似的传感器都可以集成在一套函数体系里。比如，以后要是新发明了一种传感器，那么我只要写一个与laserScanMessage平行的函数，最后让它去调用HandleRangefinder函数。HandleRangefinder这个函数永远不会变。

以上就是Cartographer\_ROS整个流程的精华，就讲完了。以上只是主线，其实很多函数的实现没有展开。

这个直播不能代替自己一行一行地去研读

最后回答之前留的问题，为什么我们要做一个封装呢？