ORB-SLAM2 ORB特征点法SLAM 支持单目、双目、rgbd相机

安装测试

本文github链接

基于ORBSLAM2加入线特征重建 半稠密地图

orbslam2 单目imu

orbslam2 双目imu 地图保存

RB-SLAM2-IMU-VIO 直接法加速 单目imu

ORBSLAM2_with_pointcloud_map 添加一个可视化线程用来显示点云

ORB_SLAM2_SSD_Semantic 依据ssd目标检测的语义信息和点云信息 获取3d语义目标信息 构建3d语义地图

ORB_SLAM2 + mask_rcnn 动态环境建图

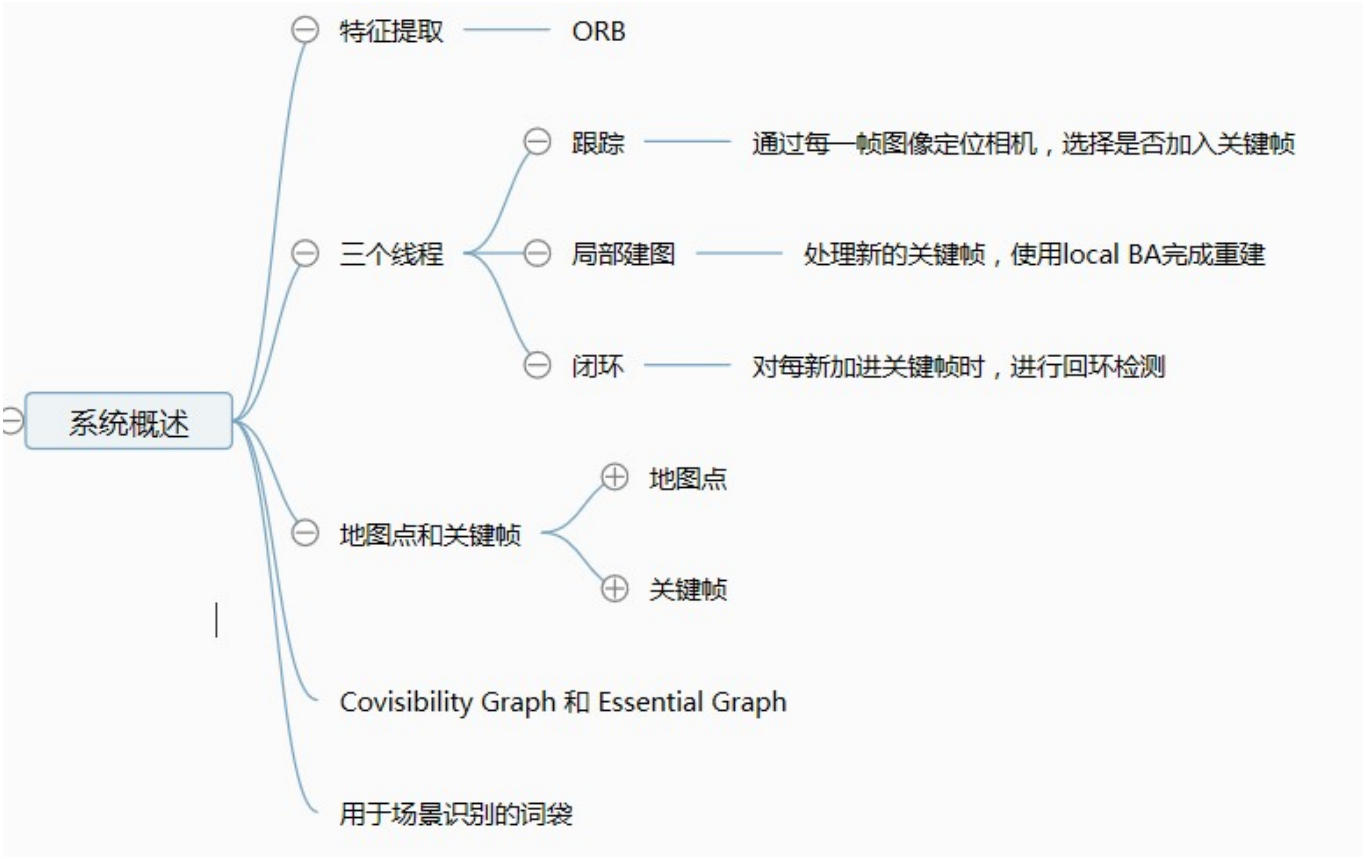
maskFusion elasFusion+ mask_rcnn 动态物体跟踪重建

DS-SLAM ORB_SLAM2 + SegNet 动态语义 slam

ORB SLAM2 + 拓扑地图 路径规划导航

ORB-SLAM是一个基于特征点的实时单目SLAM系统，在大规模的、小规模的、室内室外的环境都可以运行。
该系统对剧烈运动也很鲁棒，支持宽基线的闭环检测和重定位，包括全自动初始化。
该系统包含了所有SLAM系统共有的模块：
    跟踪（Tracking）、建图（Mapping）、重定位（Relocalization）、闭环检测（Loop closing）。
由于ORB-SLAM系统是基于特征点的SLAM系统，故其能够实时计算出相机的轨线，并生成场景的稀疏三维重建结果。
ORB-SLAM2在ORB-SLAM的基础上，还支持标定后的双目相机和RGB-D相机。

系统框架



贡献

1. 相关论文：

ORB-SLAM 单目Monocular特征点法

ORB-SLAM2 单目双目rgbd

词袋模型DBoW2 Place Recognizer

> 原作者目录:

Raul Mur-Artal

Juan D. Tardos,

J. M. M. Montiel

Dorian Galvez-Lopez

(DBoW2)

2. 简介

```
ORB-SLAM2 是一个实时的 SLAM  库，
可用于 **单目Monocular**, **双目Stereo** and **RGB-D** 相机，
用来计算 相机移动轨迹 camera trajectory 以及稀疏三维重建sparse 3D reconstruction
。

在 **双目Stereo** 和 **RGB-D** 相机 上的实现可以得到真是的 场景尺寸稀疏三维 点云图
可以实现 实时回环检测detect loops 、相机重定位relocalize the camera 。
提供了在
```

KITTI 数据集 上运行的 SLAM 系统实例，支持双目stereo 、单目monocular。

TUM 数据集 上运行的实例，支持 RGB-D相机 、单目相机 monocular,

EuRoC 数据集支持 双目相机 stereo 、单目相机 monocular.

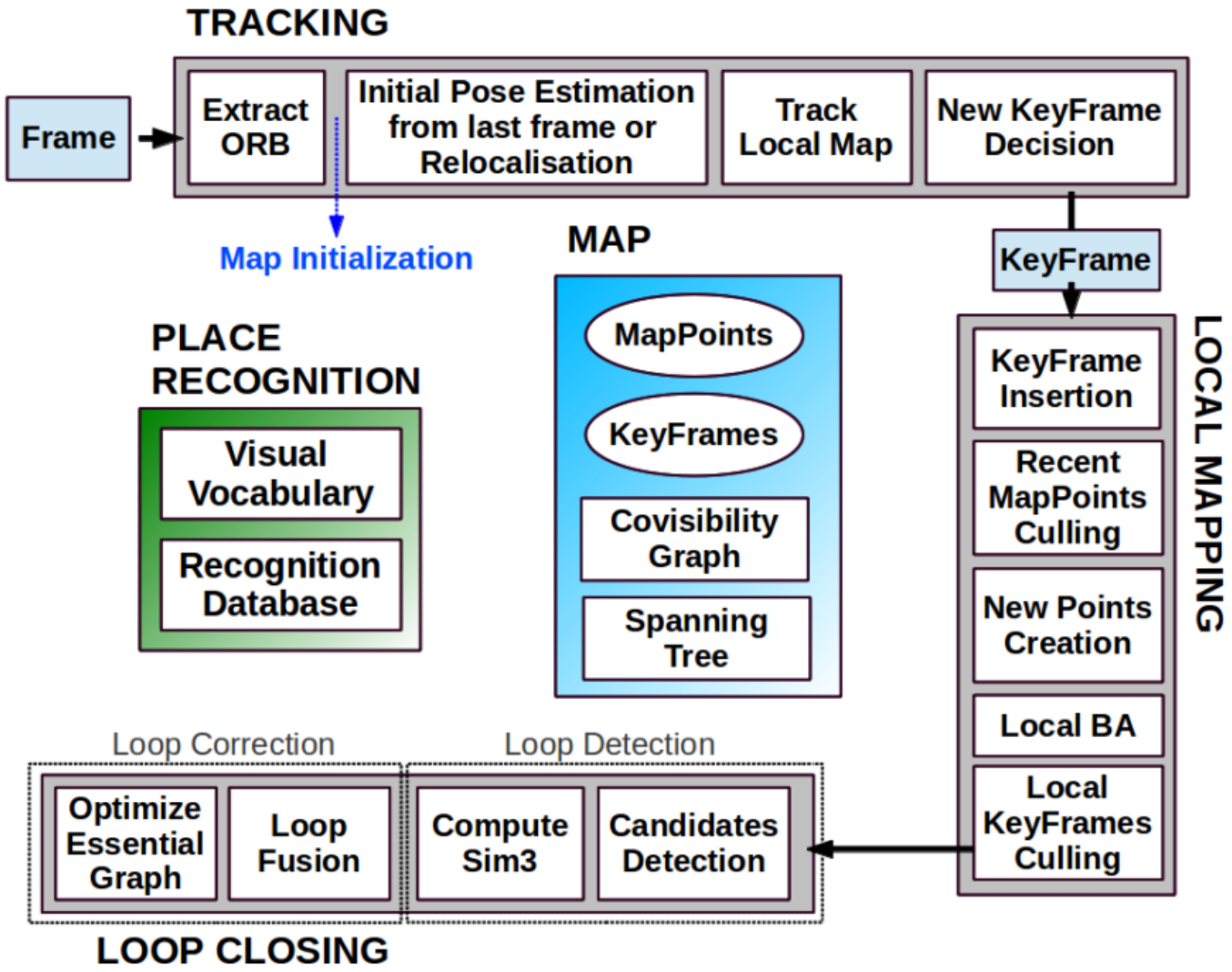
```
也提供了一个 ROS 节点 实时运行处理  单目相机 monocular, 双目相机stereo 以及 RGB-D
相机 数据流。
提供一个GUI界面 可以来 切换 SLAM模式 和重定位模式
```

> **支持的模式:

```
1. SLAM Mode  建图定位模式
    默认的模式,三个并行的线程: 跟踪Tracking, 局部建图 Local Mapping 以及 闭环检测
Loop Closing.
    定位跟踪相机localizes the camera, 建立新的地图builds new map , 检测到过得地方
close loops.

2. Localization Mode 重定位模式
    适用与在工作地方已经有一个好的地图的情况下。执行 局部建图 Local Mapping 以及 闭环
检测Loop Closing 两个线程.
    使用重定位模式,定位相机
```

3. 系统工作原理

可以看到ORB-SLAM主要分为三个线程进行，
也就是论文中的下图所示的，



分别是Tracking、LocalMapping和LoopClosing。
ORB-SLAM2的工程非常清晰漂亮，
三个线程分别存放在对应的三个文件中，
分别是：
Tracking.cpp、
LocalMapping.cpp 和
LoopClosing.cpp 文件中，很容易找到。

A. 跟踪（Tracking）

前端位姿跟踪线程采用恒速模型，并通过优化重投影误差优化位姿，
这一部分主要工作是从图像中提取ORB特征，

```
根据上一帧进行姿态估计，
或者进行通过全局重定位初始化位姿，
然后跟踪已经重建的局部地图，
优化位姿，再根据一些规则确定新的关键帧。
```

B. 建图（LocalMapping）

```
局部地图线程通过MapPoints维护关键帧之间的共视关系，
通过局部BA优化共视关键帧位姿和MapPoints，这一部分主要完成局部地图构建。
包括对关键帧的插入，验证最近生成的地图点并进行筛选，然后生成新的地图点，
使用局部捆集调整（Local BA），
最后再对插入的关键帧进行筛选，去除多余的关键帧。
```

C. 闭环检测（LoopClosing）

```
闭环检测线程通过bag-of-words加速闭环匹配帧的筛选，
并通过Sim3优化尺度，通过全局BA优化Essential Graph和MapPoints，
这一部分主要分为两个过程：
分别是：  闭环探测  和  闭环校正。
 闭环检测：
          先使用WOB进行探测，然后通过Sim3算法计算相似变换。
 闭环校正：
          主要是闭环融合和Essential Graph的图优化。
```

D. 重定位 Localization

```
使用bag-of-words加速匹配帧的筛选，并使用EPnP算法完成重定位中的位姿估计。
```

4. 代码分析

ORB-SLAM2详解（二）代码逻辑

4.1 应用程序框架

> 单目相机**app**框架：

```
1. 创建 单目ORB_SLAM2::System SLAM 对象
2. 载入图片 或者 相机捕获图片 im = cv::imread();
3. 记录时间戳 tframe ，并计时，
   std::chrono::steady_clock::now();    // c++11
   std::chrono::monotonic_clock::now();
```

```
        4. 把图像和时间戳 传给 SLAM系统, SLAM.TrackMonocular(im,tframe);
        5. 计时结束，计算时间差，处理时间。
        6. 循环2-5步。
        7. 结束，关闭slam系统，关闭所有线程 SLAM.Shutdown();
        8. 保存相机轨迹,
SLAM.SaveKeyFrameTrajectoryTUM("KeyFrameTrajectory.txt");
```

双目相机**app**程序框架：

```
        1. 读取相机配置文件(内参数 畸变矫正参数 双目对齐变换矩阵)
=======================
          cv::FileStorage fsSettings(setting_filename,
cv::FileStorage::READ);
          fsSettings["LEFT.K"] >> K_l;//内参数
          fsSettings["LEFT.D"] >> D_l;// 畸变矫正
          fsSettings["LEFT.P"] >> P_l;// P_l,P_r --左右相机在校准后坐标系中的
投影矩阵 3×4
          fsSettings["LEFT.R"] >> R_l;// R_l,R_r --左右相机校准变换（旋转）矩
阵  3×3
```

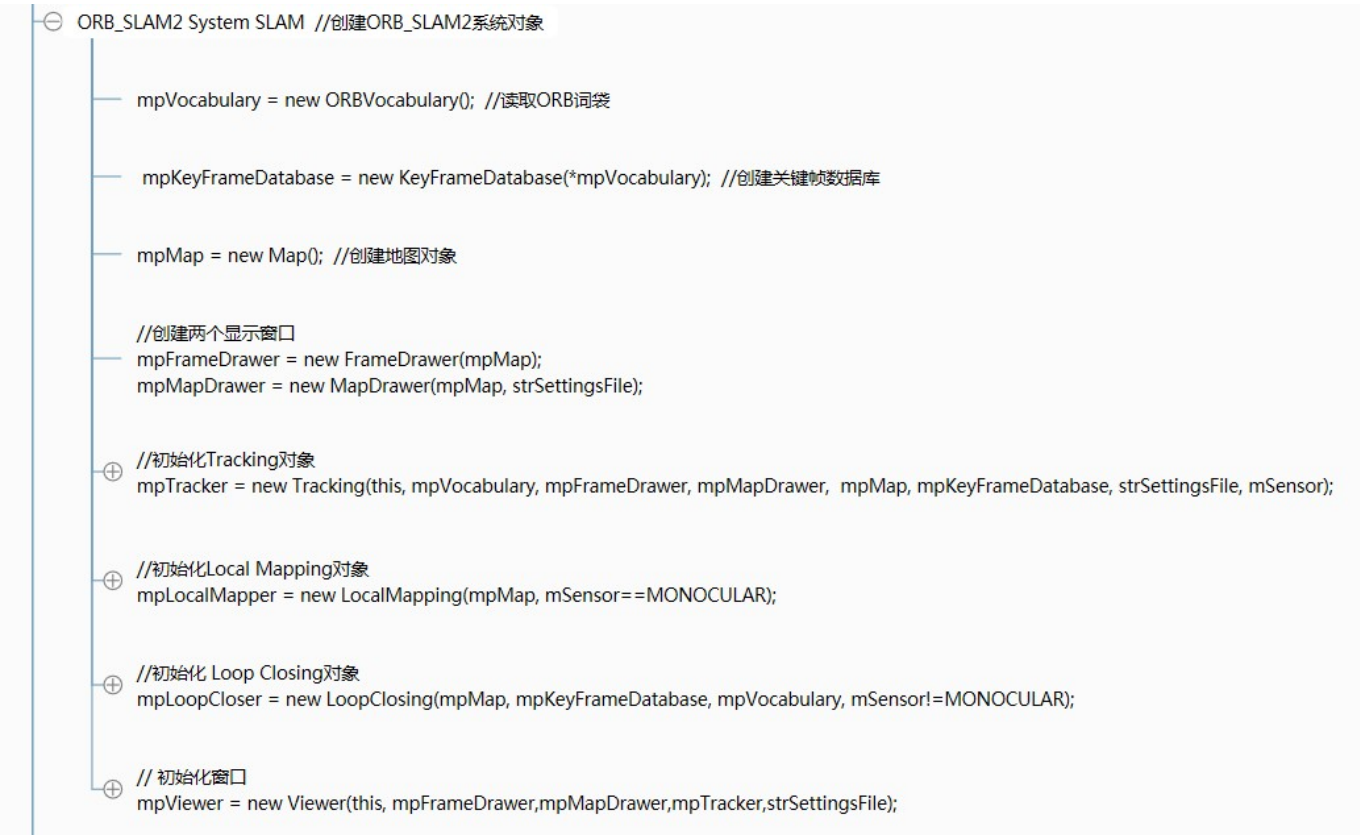
```
        2. 计算双目矫正映射矩阵
=========================================================
           cv::Mat M1l,M2l,M1r,M2r;

cv::initUndistortRectifyMap(K_l,D_l,R_l,P_l.rowRange(0,3).colRange(0,3),cv:
:Size(cols_l,rows_l),CV_32F,M1l,M2l);

cv::initUndistortRectifyMap(K_r,D_r,R_r,P_r.rowRange(0,3).colRange(0,3),cv:
:Size(cols_r,rows_r),CV_32F,M1r,M2r);
        3. 创建双目系统
=============================================================
           ORB_SLAM2::System SLAM(vocabulary_filepath, setting_filename,
ORB_SLAM2::System::STEREO, true);
        4. 从双目设备捕获图像,设置分辨率捕获图像
=======================================
           cv::VideoCapture CapAll(deviceid); //打开相机设备
           //设置分辨率   1280*480  分成两张 640*480  × 2 左右相机
           CapAll.set(CV_CAP_PROP_FRAME_WIDTH,1280);
           CapAll.set(CV_CAP_PROP_FRAME_HEIGHT, 480);
        5. 获取左右相机图像
=============================================================
           CapAll.read(src_img);
           imLeft  = src_img(cv::Range(0, 480), cv::Range(0, 640));
           imRight = src_img(cv::Range(0, 480), cv::Range(640, 1280));
        6. 使用2步获取的双目矫正映射矩阵 矫正 左右相机图像
==============================
           cv::remap(imLeft,imLeftRect,M1l,M2l,cv::INTER_LINEAR);
           cv::remap(imRight,imRightRect,M1r,M2r,cv::INTER_LINEAR);
        7. 记录时间戳 time ，并计时
=================================================
               #ifdef COMPILEDWITHC11
                  std::chrono::steady_clock::time_point    t1 =
std::chrono::steady_clock::now();
               #else
                  std::chrono::monotonic_clock::time_point t1 =
std::chrono::monotonic_clock::now();
               #endif
        8. 把左右图像和时间戳 传给 SLAM系统
=========================================
           SLAM.TrackStereo(imLeftRect, imRightRect, time);
        9. 计时结束，计算时间差，处理时间
=========================================
        10.循环执行 5-9步
=========================================
        11.结束，关闭slam系统，关闭所有线程
=======================================
        12.保存相机轨迹
=======================================
```

ORB_SLAM2::System SLAM 对象框架:

```
        在主函数中，我们创建了一个ORB_SLAM2::System的对象SLAM，这个时候就会进入到
SLAM系统的主接口System.cc。
        这个代码是所有调用SLAM系统的主入口，
        在这里，我们将看到前面博客所说的ORB_SLAM的三大模块：
        Tracking、LocalMapping 和 LoopClosing。

    System类的初始化函数：
        1. 创建字典 mpVocabulary = new ORBVocabulary()；并从文件中载入字典
           mpVocabulary = new ORBVocabulary();           // 创建关键帧字典数据
库
           // 读取 txt格式或者bin格式的 orb特征字典,
           mpVocabulary->loadFromTextFile(strVocFile);   // txt格式
           mpVocabulary->loadFromBinaryFile(strVocFile); // bin格式

        2. 使用特征字典mpVocabulary 创建关键帧数据库
           mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary);

        3. 创建地图对象 mpMap
           mpMap = new Map();

        4. 创建地图显示(mpMapDrawer) 帧显示(mpFrameDrawer) 两个显示窗口
           mpMapDrawer = new MapDrawer(mpMap, strSettingsFile);//地图显示
           mpFrameDrawer = new FrameDrawer(mpMap);//关键帧显示

        5. 初始化 跟踪线程(mpTracker) 对象 未启动
               mpTracker = new Tracking(this, mpVocabulary, mpFrameDrawer,
mpMapDrawer,
                                        mpMap, mpKeyFrameDatabase,
strSettingsFile, mSensor);

        6. 初始化 局部地图构建线程(mptLocalMapping) 并启动线程
               mpLocalMapper = new LocalMapping(mpMap, mSensor==MONOCULAR);
               mptLocalMapping = new
thread(&ORB_SLAM2::LocalMapping::Run,mpLocalMapper);

        7. 初始化 闭环检测线程(mptLoopClosing) 并启动线程
               mpLoopCloser = new LoopClosing(mpMap, mpKeyFrameDatabase,
mpVocabulary, mSensor!=MONOCULAR);
               mptLoopClosing = new thread(&ORB_SLAM2::LoopClosing::Run,
mpLoopCloser);

        8. 初始化 跟踪线程可视化 并启动
           mpViewer = new Viewer(this,
mpFrameDrawer,mpMapDrawer,mpTracker,strSettingsFile);
                mptViewer = new thread(&Viewer::Run, mpViewer);
                mpTracker->SetViewer(mpViewer);

        9. 线程之间传递指针 Set pointers between threads
               mpTracker->SetLocalMapper(mpLocalMapper);    // 跟踪线程 关联
局部建图和闭环检测线程
               mpTracker->SetLoopClosing(mpLoopCloser);
               mpLocalMapper->SetTracker(mpTracker);        // 局部建图线程 关
联 跟踪和闭环检测线程
```

```
                mpLocalMapper->SetLoopCloser(mpLoopCloser);
                mpLoopCloser->SetTracker(mpTracker);        // 闭环检测线程 关
联 跟踪和局部建图线程
                mpLoopCloser->SetLocalMapper(mpLocalMapper);
```

如下图所示：



单目跟踪SLAM.TrackMonocular()框架

```
        1. 模式变换的检测  跟踪+定位  or  跟踪+定位+建图
        2. 检查跟踪tracking线程重启
        3. 单目跟踪
           mpTracker->GrabImageMonocular(im,timestamp);// Tracking.cc中
           // 图像转换成灰度图，创建帧Frame对象(orb特征提取等)
           // 使用Track()进行跟踪:
                        0. 单目初始化(最开始执行)
MonocularInitialization();// 单目初始化
                        a. 两帧跟踪得到初始化位姿(跟踪上一帧/跟踪参考帧/重定位)
                        b. 跟踪局部地图，多帧局部地图G2O优化位姿, 新建关键帧
```

双目跟踪System::TrackStereo()框架

```
        1. 模式变换的检测  跟踪+定位  or  跟踪+定位+建图
        2. 检查跟踪tracking线程重启
```

```
        3. 双目跟踪
           mpTracker->GrabImageStereo(imLeft,imRight,timestamp); //
Tracking.cc中
           // 图像转换成灰度图，创建帧Frame对象(orb特征提取器,分块特征匹配，视差计算
深度)
           // 使用Track()进行跟踪:
                          0. 双目初始化(最开始执行) StereoInitialization();//
双目 / 深度初始化
                          a. 两帧跟踪得到初始化位姿(跟踪上一帧/跟踪参考帧/重定位)
                          b. 跟踪局部地图，多帧局部地图G2O优化位姿, 新建关键帧
```

> 深度相机跟踪**System::TrackRGBD()**框架

```
        1. 模式变换的检测   跟踪+定位   or   跟踪+定位+建图
        2. 检查跟踪tracking线程重启
        3. 双目跟踪
           mpTracker->GrabImageRGBD(im,depthmap,timestamp); // Tracking.cc
中
           // 图像转换成灰度图，创建帧Frame对象(orb特征提取器,深度图初始化特征点深度)
           // 使用Track()进行跟踪:
                          0. RGBD初始化(最开始执行) StereoInitialization();//
双目 / 深度初始化
                          a. 两帧跟踪得到初始化位姿(跟踪上一帧/跟踪参考帧/重定位)
                          b. 跟踪局部地图，多帧局部地图G2O优化位姿, 新建关键帧
```

4.2 单目/双目/RGBD初始化, 单应变换/本质矩阵恢复3d点，创建初始地图

参考

1. 单目初始化 Tracking::MonocularInitialization()

```
系统的第一步是初始化，ORB_SLAM使用的是一种自动初始化方法。
这里同时计算两个模型：
1. 用于 平面场景的单应性矩阵H( 4对 3d-2d点对，线性方程组，奇异值分解)
2. 用于 非平面场景的基础矩阵F(8对 3d-2d点对，线性方程组，奇异值分解)，
```

推到 参考 单目slam基础

```
然后通过一个评分规则来选择合适的模型，恢复相机的旋转矩阵R和平移向量t。
函数调用关系：
Tracking::GrabImageMonocular() 创建帧对象(第一帧提取orb特征点数量较多,为后面帧的两
倍) ->
Tracking::Track() ->  初始化 MonocularInitialization();// 单目初始化
          |  1. 第一帧关键点个数超过 100个，进行初始化  mpInitializer = new
```

```
Initializer(mCurrentFrame,1.0,200)；
          |  2. 第二帧关键点个数 小于100个，删除初始化器,跳到第一步重新初始化。
          |  3. 第二帧关键点个数 也大于100个(只有连续的两帧特征点 均>100 个才能够成功构建初始化器)
          |     构建 两两帧 特征匹配器    ORBmatcher::ORBmatcher matcher(0.9,true)

          |     金字塔分层块匹配搜索匹配点对  100为搜索窗口大小尺寸尺度
          |     int nmatches=matcher.SearchForInitialization(mInitialFrame,mCurrentFrame,mvbPrevMatched,mvIniMatches,100);

          |  4. 如果两帧匹配点对过少(nmatches<100)，跳到第一步，重新初始化。
          |  5. 匹配点数量足够多(nmatches >= 100),进行单目初始化：
          |     第一帧位置设置为：单位阵 Tcw = cv::Mat::eye(4,4,CV_32F);
          |     使用 单应性矩阵H 和 基础矩阵F 同时计算两个模型，通过一个评分规则来选择合适的模型，
          |     恢复第二帧相机的旋转矩阵Rcw 和 平移向量 tcw，同时 三角变换得到 部分三维点 mvIniP3D
          |     mpInitializer->Initialize(mCurrentFrame, mvIniMatches, Rcw, tcw, mvIniP3D, vbTriangulated))
          |  6. 设置初始参考帧的世界坐标位姿态:
          |     mInitialFrame.SetPose(cv::Mat::eye(4,4,CV_32F));
          |  7. 设置第二帧(当前帧)的位姿
          |     cv::Mat Tcw = cv::Mat::eye(4,4,CV_32F);
          |     Rcw.copyTo(Tcw.rowRange(0,3).colRange(0,3));
          |     tcw.copyTo(Tcw.rowRange(0,3).col(3));
          |     mCurrentFrame.SetPose(Tcw);
          |  8. 创建初始地图 使用 最小化重投影误差BA 进行 地图优化 优化位姿 和地图点
          |     Tracking::CreateInitialMapMonocular()
          |
          |-> 后面帧的跟踪 -> 两两帧的跟踪得到初始位姿
                    |  1. 有运动速度，使用恒速运动模式跟踪上一帧 Tracking::TrackWithMotionModel()
                    |  2. 运动量小或者 1.跟踪失败，跟踪参考帧模式 Tracking::TrackReferenceKeyFrame()
                    |  3. 1 和 2都跟踪失败的话，使用重定位跟踪，跟踪可视参考帧群 Tracking::Relocalization()
                    |
                   之后 -> 跟踪局部地图(一级二级相连关键帧), 使用图优化对位姿进行精细化调整 Tracking::TrackLocalMap()
                       |
                       -> 判断是否需要新建关键帧    Tracking::NeedNewKeyFrame(); Tracking::CreateNewKeyFrame();
                            |
                            |-> 跟踪失败后的处理
```

> 初始化时，orb匹配点的搜索 ORBmatcher::SearchForInitialization()

```
金字塔分层分块orb特征匹配，然后通过三个限制，滤除不好的匹配点对:
        1. 最小值阈值限制；
        2. 最近匹配距离一定比例小于次近匹配距离；
```

```
        3. 特征方向误差一致性判断(方向差直方图统计，保留3个最高的方向差一致性最多的点，一致性较大)
            ORBmatcher::ComputeThreeMaxima(); // 统计数组中最大 的几个数算法，参考性较大
            ORBmatcher.cc 1912行 优秀的算法值得参考
步骤:
        步骤1：为帧1的每一个关键点在帧2中寻找匹配点(同一金字塔层级，对应位置方块内的点，多个匹配点)
                Frame::GetFeaturesInArea()
        步骤2：计算 1对多 匹配点对描述子之间的距离(二进制变量差异)
                ORBmatcher::DescriptorDistance(d1,d2); // 只计算了前八个 二进制位 的差异
                ORBmatcher.cc 1968行 优秀的算法值得参考
        步骤3：保留最小和次小距离对应的匹配点
                ORBmatcher.cc 570行  优秀的算法值得参考
        步骤4：确保最小距离小于阈值 50
        步骤5：确保最佳匹配比次佳匹配明显要好，那么最佳匹配才真正靠谱，并统计方向差值直方图
        步骤6：特征方向误差一致性判断(方向差直方图统计，保留3个最高的方向差一致性最多的点，一致性较大)
        步骤7：更新匹配信息，用最新的匹配更新之前记录的匹配
```

> 单目位姿恢复分析 Initializer::Initialize(mCurrentFrame, mvIniMatches, Rcw, tcw, mvIniP3D, vbTriangulated))

```
步骤1：根据 matcher.SearchForInitialization 得到的初始匹配点对，筛选后得到好的特征匹配点对
步骤2：在所有匹配特征点对中随机选择8对特征匹配点对为一组，共选择 mMaxIterations 组
步骤3：调用多线程分别用于计算fundamental matrix(基础矩阵F) 和 homography(单应性矩阵)
   Initializer::FindHomography();   得分为SH
   Initializer::FindFundamental();  得分为SF
步骤4：计算评价得分 RH，用来选取某个模型
       float RH = SH / (SH + SF);// 计算 选着标志
步骤5：根据评价得分，从单应矩阵H 或 基础矩阵F中恢复R,t
   if(RH>0.40)// 更偏向于 平面  使用  单应矩阵恢复
     return
ReconstructH(vbMatchesInliersH,H,mK,R21,t21,vP3D,vbTriangulated,1.0,50);
   else //if(pF_HF>0.6) // 偏向于非平面  使用 基础矩阵 恢复
     return
ReconstructF(vbMatchesInliersF,F,mK,R21,t21,vP3D,vbTriangulated,1.0,50);
```

0. 2D-2D配点对求变换矩阵前先进行标准化处理去 均值后再除以绝对矩 Initializer::Normalize()

```
步骤1：计算两坐标的均值
       mean_x  =  sum(ui) / N ；
       mean_y  =  sum(vi) / N；
步骤2：计算绝对局倒数
      绝对矩：
```

```
          mean_x_dev = sum（abs(ui - mean_x)）/ N ;
          mean_y_dev = sum（abs(vi - mean_y)）/ N ;
       绝对矩倒数：
          sX = 1/mean_x_dev
          sY = 1/mean_y_dev
步骤3：计算标准化后的点坐标
        ui' = (ui - mean_x) * sX
        vi' = (vi - mean_y) * sY
步骤4：计算并返回标准化矩阵
        ui' = (ui - mean_x) * sX =  ui * sX + vi * 0  + (-mean_x * sX) * 1
        vi' = (vi - mean_y) * sY =  ui * 0  + vi * sY + (-mean_y * sY) * 1
        1   =                       ui * 0  + vi * 0  +      1         * 1

        可以得到：
                ui'     sX  0   (-mean_x * sX)      ui
                vi' =   0   sY  (-mean_y * sY)   *  vi
                1       0   0        1              1
        标准化后的的坐标 = 标准化矩阵T * 原坐标
        所以标准化矩阵:
                T =  sX  0   (-mean_x * sX)
                     0   sY  (-mean_y * sY)
                     0   0        1
        而由标准化坐标 还原 回 原坐标(左乘T 逆)：
                原坐标 = 标准化矩阵T 逆矩阵 * 标准化后的的坐标

        再者用于还原 单应矩阵 下面需要用到：
          p1'  ------> Hn --------> p2'   , p2'   = Hn*p1'
          T1*p1 -----> Hn --------> T2*p2 , T2*p2 = Hn*(T1*p1)
          左乘 T2逆 , 得到   p2 = T2逆 * Hn*(T1*p1)= H21i*p1
          H21i = T2逆 * Hn * T1
```

a. fundamental matrix(基础矩阵F) 随机采样 找到最好的 基础矩阵 Initializer::FindFundamental()

```
思想：
计算基础矩阵F,随机采样序列8点法,采用归一化的直接线性变换（normalized DLT）求解,
极线几何约束(current frame 1 变换到 reference frame 2),
在最大迭代次数内调用 ComputeF21计算F,
使用 CheckFundamental 计算此 基础矩阵的得分,
在最大迭代次数内保留最高得分的基础矩阵F。

步骤：
步骤1：将两帧上对应的2d-2d匹配点对进行归一化
        Initializer::Normalize(mvKeys1,vPn1, T1);//  mvKeys1原坐标 vPn1归一化
坐标，T1标准化矩阵
        Initializer::Normalize(mvKeys2,vPn2, T2);//
        cv::Mat T2inv = T2.inv();// 标准化矩阵 逆矩阵
步骤2：在最大迭代次数mMaxIterations内，从标准化后的点中随机选取8对点对
        int idx = mvSets[it][j];//随机数集合 总匹配点数范围内
        vPn1i[j] = vPn1[mvMatches12[idx].first];
        vPn2i[j] = vPn2[mvMatches12[idx].second];
```

```
步骤3：通过标准化逆矩阵 和 标准化点对的基础矩阵 计算原点对的 基础矩阵
        cv::Mat Fn = ComputeF21(vPn1i,vPn2i);// 计算 标准化后的点对 对应的 基础
矩阵 Fn
        cv::Mat T2t = T2.t(); // 标准化矩阵的 转置矩阵
         推到：
         x2 = R*x1 + t
         t 叉乘 x2 = t 叉乘 R*x1
         x2转置 * t 叉乘 x2 = x2转置 * t 叉乘 R*x1 = 0
         得到：
         x2转置 * t 叉乘 R  * x1 = x2转置 * R21  * x1 =  0
         (K逆*p2)转置 * t 叉乘 R  * (K逆*p1) =
         p2转置 * K转置逆 * t 叉乘 R  * K逆 * p1 = p2转置 * F21 *  p1

         上面求到的是 归一化后的点对的变换矩阵：
         p2'转置 * Fn * p1' = 0
         (T2*p2)转置 * Fn * (T1 * p1) = 0
         p2转置 * T2转置 * Fn * T1 * p1 = 0
         所以得到：
           未归一化点对对应的变换矩阵F21为：
           F21 =  T2转置 * Fn * T1 = T2t * Fn * T1

步骤4：通过计算重投影误差来计算单应矩阵的好坏，得分
        currentScore =Initializer::CheckFundamental();
步骤5：保留迭代中，得分最高的单应矩阵和对应的得分
        if(currentScore > score)//此次迭代 计算的单应H的得分较高
        {
            F21 = F21i.clone();// 保留最优的 基础矩阵 F
            vbMatchesInliers = vbCurrentInliers;//对应的匹配点对    标记内点
            score = currentScore;// 最高的得分
        }
```

Initializer::ComputeF21(vPn1i,vPn2i) 8对点求解 基础矩阵F

```
 推到：
 x2 = R*x1 + t
 t 叉乘 x2 = t 叉乘 R*x1
 x2转置 * t 叉乘 x2 = x2转置 * t 叉乘 R*x1 = 0
 得到：
 x2转置 * t 叉乘 R  * x1 = x2转置 * R21  * x1 =  0
 (K逆*p2)转置 * t 叉乘 R  * (K逆*p1) =
 p2转置 * K转置逆 * t 叉乘 R  * K逆 * p1 = p2转置 * F21 *  p1

 一对2d-2d点对 p1-p2得到:
          p2转置 * F21 *  p1
写成矩阵形式：
              |f1   f2   f3|     |u1|
|u2 v2 1| *   |f4   f5   f6|  *  |v1|    = 0 , 应该=0 不等于零的就是误差
              |f7   f8   f9|     |1|
前两项展开得到：
        a1 = f1*u2 + f4*v2 + f7;
        b1 = f2*u2 + f5*v2 + f8;
```

```
        c1 = f3*u2 + f6*v2 + f9;
得到:
                      |u1|
        |a1 b1 c1| * |v1| = 0
                      |1|
得到:
       a1*u1+ b1*v1 + c1= 0
一个点对 得到一个约束方程:
  f1*u1*u2 + f2*v1*u2  + f3*u2 + f4*u1*v2  + f5*v1*v2 + f6*v2 +  f7*u1 +
f8*v1 + f9 = 0
写成矩阵形式：
 |u1*u2 v1*u2 u2 u1*v2 v1*v2 v2 u1 v1 1| * |f1 f2 f3 f4 f5 f6 f7 f8 f9|转置
= 0
采样8个点对 可以到八个约束, f 9个参数，8个自由度，另一个为尺度因子(单目尺度不确定的来源)
线性方程组 求解 A * f = 0
对矩阵A进行奇异值分解得到f ,

对A进行SVD奇异值分解 [U,S,V]=svd(A)，其中U和V代表二个相互正交矩阵，而S代表一对角矩阵
cv::SVDecomp(A,S,U,VT,SVD::FULL_UV);  //后面的FULL_UV表示把U和VT补充称单位正交方
阵
Fpre  = VT.row(8).reshape(0, 3);       // V的最后一列
F矩阵的其秩为2,需要再对Fpre进行奇异值分解, 后取对角矩阵U, 秩为2,后再合成F。
cv::SVDecomp(Fpre,S,U,VT,cv::SVD::MODIFY_A | cv::SVD::FULL_UV);
S.at<float>(2)=0;                      //   基础矩阵的秩为2，重要的约束条件
F21 = U * cv::Mat::diag(S)* VT         // 再合成F
```

Initializer::CheckFundamental() 计算基本矩阵得分 得分为**SF**

```
思想:
        1. 根据基本矩阵，可以求得两组对点相互变换得到的误差平方和，
        2. 基于卡方检验计算出的阈值（假设测量有一个像素的偏差），
        3. 误差大的记录为外点，误差小的记录为内点，
        4. 使用阈值减去内点的误差，得到该匹配点对的得分(误差小的，得分高)，
        5. 记录相互变换中所有内点的得分之和作为该单元矩阵的得分，并更新匹配点对的内点外
点标记。

步骤1: 误差阈值 th = 3.841; 得分最大值 thScore = 5.991; 误差方差倒数
invSigmaSquare
步骤2: 遍历每一对2d-2d点对计算误差：

        如： p2 ->  F12  -> p1
                      |f1   f2   f3|     |u1|
        |u2 v2 1| *  |f4   f5   f6|  *  |v1|    = 0 , 应该=0 不等于零的就是误
差
                      |f7   f8   f9|     |1|
前两项展开得到：
        a1 = f1*u2 + f4*v2 + f7;
        b1 = f2*u2 + f5*v2 + f8;
        c1 = f3*u2 + f6*v2 + f9;
得到：
```

```
                        |u1|
        |a1 b1 c1| * |v1| = 0 ，其实就是点 (u1, v1) 在线段 (a1, b1, c1) 上的形
式
                        |1|
极线l1：a1*x + b1*y + c1 = 0,这里把p2投影到帧1平面上对应的极线形式,p1应该在上面。
点p1,(u1,v1) 到 l1的距离为：
        d = |a1*u + b1*v + c| / sqrt(a1^2 + b1^2)
距离平方：
        chiSquare1 = d^2 = (a1*u + b1*v + c)^2 / (a1^2 + b1^2)

根据方差归一化误差:
        const float chiSquare1 = squareDist1*invSigmaSquare;

使用阈值更新内外点标记 并记录内点得分:
        if(chiSquare1 > th)
            bIn = false;                    // 距离大于阈值  该点 变换的效果差，记
录为外点
        else
            score += thScore - chiSquare1;// 得分上限 - 距离差值 得到 得分，差值
越小，得分越高       得分为SF

同时记录 p1 ->  F21  -> p2 的误差，也如上述步骤。
更新内外点记录数组：
        if(bIn)
                vbMatchesInliers[i]=true;// 是内点  误差较小
        else
                vbMatchesInliers[i]=false;// 是野点 误差较大
```

b. homography(单应性矩阵) 随机采样 找到最好的单元矩阵 Initializer::FindHomography()

```
思想：
计算单应矩阵,随机采样序列4点法,采用归一化的直接线性变换（normalized DLT）求解，
假设场景为平面情况下通过前两帧求取Homography矩阵(current frame 2 到 reference
frame 1)，
在最大迭代次数内调用 ComputeH21计算H，
使用 CheckHomography 计算此单应变换得分，
在最大迭代次数内保留最高得分的单应矩阵H。

步骤：
步骤1：将两帧上对应的2d-2d匹配点对进行归一化
        Initializer::Normalize(mvKeys1,vPn1, T1);//  mvKeys1原坐标 vPn1归一化
坐标，T1标准化矩阵
        Initializer::Normalize(mvKeys2,vPn2, T2);//
        cv::Mat T2inv = T2.inv();               // 标准化矩阵 逆矩阵
步骤2：在最大迭代次数mMaxIterations内，从标准化后的点中随机选取8对点对
        int idx = mvSets[it][j];                //随机数集合 总匹配点数范围内
        vPn1i[j] = vPn1[mvMatches12[idx].first];
        vPn2i[j] = vPn2[mvMatches12[idx].second];
步骤3：通过标准化逆矩阵和标准化点对的单应变换计算原点对的单应变换矩阵
Initializer::ComputeH21()
        cv::Mat Hn = ComputeH21(vPn1i,vPn2i);// 计算 标准化后的点对 对应的 单应
```

```
矩阵Hn
        // H21i = T2逆 * Hn * T1  见上面  0步骤的推导
        H21i = T2inv * Hn * T1;// 原始点    p1 -----------> p2 的单应
        H12i = H21i.inv();     // 原始点    p2 -----------> p1 的单应
步骤4：通过计算重投影误差来计算单应矩阵的好坏，得分
        currentScore =Initializer::CheckHomography();
步骤5：保留迭代中，得分最高的单应矩阵和对应的得分
        if(currentScore > score)//此次迭代 计算的单应H的得分较高
        {
            H21 = H21i.clone();//保留较高得分的单应
            vbMatchesInliers = vbCurrentInliers;//对应的匹配点对
            score = currentScore;// 最高的得分
        }
```

Initializer::ComputeH21() 4对点直接线性变换求解H矩阵

```
一点对：
        p2   =  H21 * p1
写成矩阵形式：
        u2          h1  h2  h3       u1
        v2  =       h4  h5  h6    *  v1
        1           h7  h8  h9       1

可以使用叉乘 得到0    p2叉乘p2 = H21 *p1 = 0
        | 0 -1  v2|    |h1 h2 h3|      |u1|    |0|
        | 1  0 -u2| *  |h4 h5 h6| *    |v1| =  |0|
        |-v2 u2  0|    |h7 h8 h9|      |1 |    |0|

也可以展开得到(使用第三项进行归一化)：
        u2 = (h1*u1 + h2*v1 + h3) /( h7*u1 + h8*v1 + h9)
        v2 = (h4*u1 + h5*v1 + h6) /( h7*u1 + h8*v1 + h9)
写成矩阵形式：
        -((h4*u1 + h5*v1 + h6) - ( h7*u1*v2 + h8*v1*v2 + h9*v2))=0  右乘分
子，再移动得0
        h1*u1    + h2*v1 + h3  - ( h7*u1*u2 + h8*v1*u2 + h9*u2) =0

        |0    0   0  -u1  -v1  -1   u1*v2   v1*v2    v2|
        |u1 v1    1  0    0    0   -u1*u2  - v1*u2  -u2| *|h1 h2 h3 h4 h5
h6 h7 h8 h9|转置  = 0
一对点提供两个约束：H 9个元素，8个自由度，包含一个比例因子(尺度来源)，需要四对点
四对点提供8个约束方程,可以写成矩阵形式：
 A * h = 0
对A进行SVD奇异值分解 [U,S,V]=svd(A)，其中U和V代表二个相互正交矩阵，而S代表一对角矩阵
cv::SVDecomp(A,S,U,VT,SVD::FULL_UV);  //后面的FULL_UV表示把U和VT补充称单位正交方
阵
H = VT.row(8).reshape(0, 3);// v的最后一列

SVD奇异值分解 解齐次方程组（Ax = 0）原理：
        把问题转化为最小化|| Ax ||2的非线性优化问题，
        我们已经知道了x = 0是该方程组的一个特解，
        为了避免x = 0这种情况（因为在实际的应用中x = 0往往不是我们想要的），
```

```
        我们增加一个约束，比如|| x ||2 = 1，
        这样，问题就变为：
        min(|| Ax ||2) ， || x ||2 = 1 或 min(|| Ax ||) ， || x || = 1
        对矩阵A进行分解 A = UDV'
         || Ax || = || UDV' x || = || DV' x||
         ( 对于一个正交矩阵U，满足这样一条性质： || UD || = || D || ,
           正交矩阵，正交变换，仅仅对向量，只产生旋转，无尺度缩放，和变形，即模长不变)

        令y = V'x， 因此，问题变为min(|| Dy ||)，
        因为|| x || = 1, V'为正交矩阵，则|| y || = 1。
        由于D是一个对角矩阵，对角元的元素按递减的顺序排列，因此最优解在y = (0, 0,...,
1)'
        又因为x = Vy， 所以最优解x，就是V的最小奇异值对应的列向量，
        比如，最小奇异值在第8行8列，那么x = V的第8个列向量。
```

计算单应变换得分 **Initializer::CheckHomography()** 得分为**SH**

```
思想:
        1. 根据单应变换，可以求得两组对点相互变换得到的误差平方和，
        2. 基于卡方检验计算出的阈值（假设测量有一个像素的偏差），
        3. 误差大的记录为外点，误差小的记录为内点，
        4. 使用阈值减去内点的误差，得到该匹配点对的得分(误差小的，得分高)，
        5. 记录相互变换中所有内点的得分之和作为该单元矩阵的得分，并更新匹配点对的内点外
点标记。

步骤1：获取单应变换误差阈值th,以及误差归一化的方差倒数
        const float th = 5.991;                         // 单应变换误差 阈值
        const float invSigmaSquare = 1.0/(sigma*sigma);//方差 倒数，用于将误差
归一化

步骤2：遍历每个点对，计算单应矩阵 变换 时产生 的 对称的转换误差
  p1点 变成 p2点  p2 = H21*p1---------------------------------------
        |u2'|      |h11  h12  h13|     |u1|    |h11*u1 + h12*v1 + h13|
        |v2'|  =   |h21  h22  h23|  *  |v1| =  |h21*u1 + h22*v1 + h23|
        |1|        |h31  h32  h33|     |1|     |h31*u1 + h32*v1 + h33|
         使用第三行进行归一化： u2' = (h11*u1 + h12*v1 + h13)/(h31*u1 + h32*v1
+ h33);
                              v2' = (h21*u1 + h22*v1 + h23)/(h31*u1 +
h32*v1 + h33);
         所以 p1通过 H21 投影到另一帧上 应该和p2的左边一致，但是实际不会一致，可以求得
坐标误差
         误差平方和            squareDist2 = (u2-u2')*(u2-u2') + (v2-v2')*
(v2-v2');
         使用方差倒数进行归一化 chiSquare2 = squareDist2*invSigmaSquare;

  使用阈值更新内外点标记 并记录内点得分:
        if(chiSquare2>th)
                bIn = false;              //距离大于阈值  该点 变换的效果差，记
录为外点
        else
                score += th - chiSquare2; // 阈值 - 距离差值 得到 得分，差值越小
```

```
得分越高

  p2点 变成 p1点  p1 = H12 * p2 -----------------------------------
         |u1'|     |h11inv   h12inv   h13inv|    |u2|   |h11inv*u2 +
h12inv*v2 + h13inv|
         |v1'|  =  |h21inv   h22inv   h23inv|  * |v2| = |h21inv*u2 +
h22inv*v2 + h23inv|
         |1|       |h31inv   h32inv   h33inv|    |1|    |h31inv*u2 +
h32inv*v2 + h33inv|
         使用第三行进行归一化： u1' = (h11inv*u2 + h12inv*v2 +
h13inv)/(h31inv*u2 + h32inv*v2 + h33inv);
                              v1' = (h21inv*u2 + h22inv*v2 +
h23inv)/(h31inv*u2 + h32inv*v2 + h33inv);
         所以 p1通过 H21 投影到另一帧上 应该和p2的左边一致，但是实际不会一致，可以求得
坐标误差
         误差平方和             squareDist1 = (u1-u1')*(u1-u1') + (v1-v1')*
(v1-v1');
         使用方差倒数进行归一化 chiSquare1 = squareDist1*invSigmaSquare;

  使用阈值更新内外点标记 并记录内点得分:
        if(chiSquare1>th)
                bIn = false;              // 距离大于阈值  该点 变换的效果差，记
录为外点
        else
                score += th - chiSquare1;// 阈值 - 距离差值 得到 得分，差值越小
得分越高  SH

  更新内外点记录数组：
        if(bIn)
                vbMatchesInliers[i]=true;// 是内点  误差较小
        else
                vbMatchesInliers[i]=false;// 是野点 误差较大
```

从两个模型 **H F** 得分为 **Sh Sf** 中选着一个 最优秀的 模型 的方法为

```
文中认为，当场景是一个平面、或近似为一个平面、或者视差较小的时候，可以使用单应性矩阵H恢复
运动，
当场景是一个非平面、视差大的场景时，使用基础矩阵F恢复运动,
两个变换矩阵得分分别为 SH 、SF,
    根据两者得分计算一个评价指标RH:
RH = SH / ( SH + SF)
当大于0.45时，选择从单应性变换矩阵还原运动,反之使用基础矩阵恢复运动。
不过ORB_SLAM2源代码中使用的是0.4作为阈值。
```

c. 单应矩阵H恢复R,t Initializer::ReconstructH()

```
单应矩阵恢复  旋转矩阵 R 和平移向量t
p2   =  H21 * p1
p2 = K( RP + t)  = KTP = H21 * KP
```

```
A = T =  K 逆 * H21*K = [ R t; 0 0 0 1]
对A进行奇异值分解
cv::SVD::compute(A,w,U,Vt,cv::SVD::FULL_UV);
使用FAUGERAS的论文[1]的方法，提出8种运动假设,分别可以得到8组R,t
正常来说，第二帧能够看到的点都在相机的前方，即深度值Z大于0.
但是如果在低视差的情况下，使用 三角变换Initializer::Triangulate() 得到的3d点云，
使用 Initializer::CheckRT() 统计符合该R，t的内点数量。
```

Initializer::Triangulate() 使用**2d-2d**点对 和 变换矩阵 **R**，**t** 三角变换恢复**2d**点对应的**3d**点

```
Trianularization: 已知匹配特征点对{p1  p2} 和
各自相机投影矩阵{P1 P2}, P1 = K*[I 0], P2 = K*[R t], 尺寸为[3,4]
估计三维点 X3D
     p1 = P1 * X3D
     p2 = P2 * X3D
采用直接线性变换DLT的方法(将式子变换成A*X=0的形式后使用SVD奇异值分解求解线性方程组)：
对于 p1 = P1 * X3D: 方程两边 左边叉乘 p1，可使得式子为0
p1叉乘P1*X3D = 0
其叉乘矩阵为:
叉乘矩阵 =
          |0  -1  y|
          |1   0 -x|
          |-y  x  0|
上述等式可写:
  |0  -1  y|  |P1.row(0)|
  |1   0 -x| *|P1.row(1)|* X3D = 0
  |-y  x  0|  |P1.row(2)|

对于第一行 |0  -1  y| 会与P的三行分别相乘 得到四个值 与齐次3d点坐标相乘得到 0
有 (y * P1.row(2) - P1.row(1) ) * X3D = 0
对于第二行 |1   0 -x|有：
有 (x * P1.row(2) - P1.row(0) ) * X3D = 0
得到两个约束，另外一个点 p2 = P2 * X3D,也可以得到两个式子：
(y‘ * P2.row(2) - P2.row(1) ) * X3D = 0
(x’ * P2.row(2) - P2.row(0) ) * X3D = 0
写成 A*X = 0的形式有：
A =(维度4*4)
|y * P1.row(2) - P1.row(1) |
|x * P1.row(2) - P1.row(0) |
|y‘ * P2.row(2) - P2.row(1)|
|x’ * P2.row(2) - P2.row(0)|
对A进行奇异值分解求解X
cv::SVD::compute(A,w,u,vt,cv::SVD::MODIFY_A| cv::SVD::FULL_UV);
x3D = vt.row(3).t();// vt的最后一列,为X的解
x3D = x3D.rowRange(0,3)/x3D.at<float>(3);//  转换成非齐次坐标  归一化
```

initializer::CheckRT() 计算**R , t** 的得分**(**内点数量**)**

```
统计规则，使用 Initializer::Triangulate()  三角化的3d点，
会有部分点云跑到相机的后面(Z<0)，该部分点是噪点，需要剔除.
另外一个准则是将上述符合条件的3d点分别重投影会前后两帧上，
与之前的orb特征点对的坐标做差，误差平方超过阈值，剔除。
统计剩余符合条件的点对数量。
在8种假设中，记录得到最大内点数量 和 次大内点数量
当最大内点数量 远大于 次大内点数量( 有明显的差异性)，
该最大内点数量对应的R,t，具有较大的可靠性，返回该，R，t。
```

d. 基础矩阵F恢复R,t Initializer::ReconstructF()

```
计算 本质矩阵 E  =  K转置 * F  * K
E = t叉乘 R
奇异值分解 E= U C  V   , U 、V 为正交矩阵， C 为奇异值矩阵  C =  diag(1, 1, 0)
从本质矩阵E 恢复 旋转矩阵R 和 平移向量t
有四种假设 得到四种 R,t
理论参考 Result 9.19 in Multiple View Geometry in Computer Vision
使用 initializer::CheckRT() 计算R , t 的得分(内点数量)
使用4中情况中得分最高的R,t 作为最终恢复的R，t
```

e. 单目创建初始地图 CreateInitialMapMonocular();

```
思想：
        使用单目相机前两帧生成的3d点创建初始地图，
        使用全局地图优化 Optimizer::GlobalBundleAdjustemnt(mpMap,20)优化地图
点，
        计算场景的深度中值(地图点深度的中位数)，
        使用平均逆深度归一化 两帧的平移变换量t,
        使用 平均逆深度归一化地图点。
步骤：
步骤1：使用前两帧创建关键帧，并把这 两帧 关键帧 加入地图，加入关键帧数据库。
步骤2：使用之前三角化得到的3d点创建 地图点。
步骤3：地图点和每一帧的2d点对应起来，在关键帧中添加这种对应关系。 关键帧关联地图点。
步骤4：关键帧和2d点对应起来，在地图点中添加这种关系。                地图点关联关键帧。
步骤5：一个地图点会被许多个关键帧观测到，那么就会关联到许多个2d点，需要更新地图点对应2d点
的orb特征描述子。
步骤6：当前帧关联地图点，地图点加入到地图。
步骤7：更新关键帧的 连接关系，被地图点观测到的次数。
步骤8：全局优化地图 BA最小化重投影误差，这两帧姿态进行全局优化。
       Optimizer::GlobalBundleAdjustemnt(mpMap,20);
       // 注意这里使用的是全局优化，和回环检测调整后的大回环优化使用的是同一个函数。
       // 放在后面再进行解析
步骤9：计算场景深度中值，以及逆深度中值，对位姿的平移量进行尺度归一化，对地图点进行尺度归
一化。
```

2. 双目/RGBD初始化 Tracking::StereoInitialization()根据视差计算深度（深度相机直接获取深度）计算3d点，创建地图点，创建初始地图

步骤：
当前帧 特征点个数 大于500 进行初始化
设置第一帧为关键帧，并设置为位姿为 T = [I 0]，世界坐标系
创建关键帧，地图添加关键帧，
根据每一个2d点的视差（左右两张图像，orb特征点金字塔分层分块匹配得到视差d）求得的深度D=fB/d，计算对应的3D点，并创建地图点，
地图点关联观测帧 地图点计算所有关键帧中最好的描述子并更新地图点的方向和距离，
关键帧关联地图点，地当前帧添加地图点 地图添加地图点
局部地图中添加该初始关键帧。

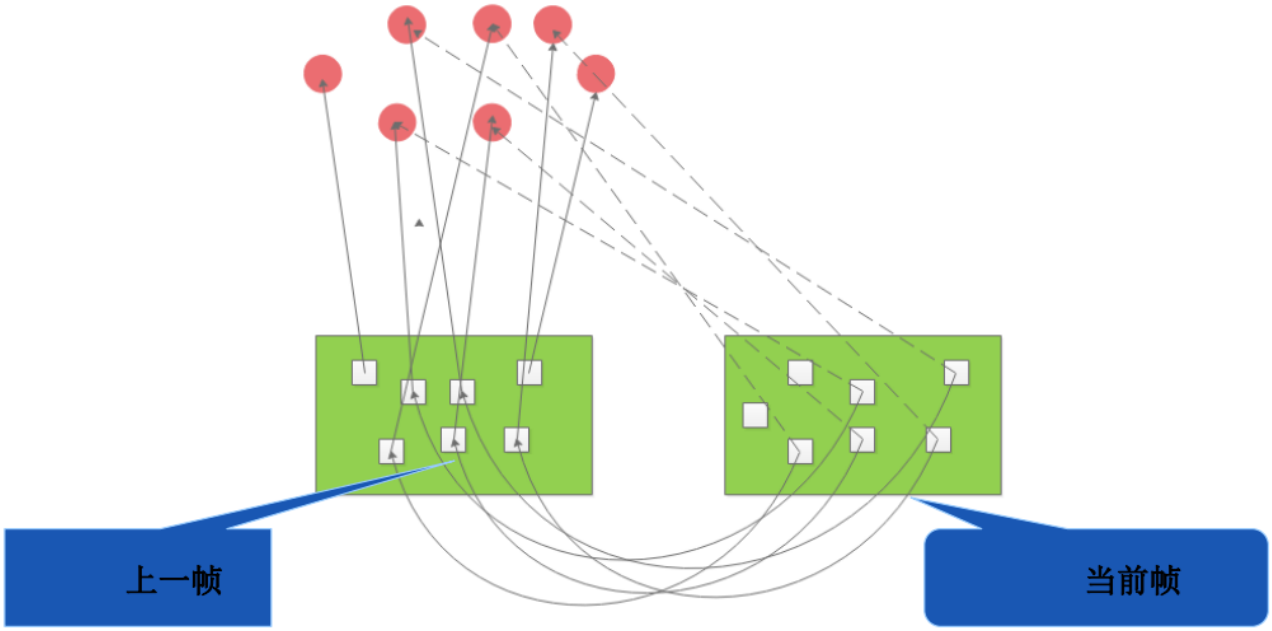
4.3 两帧跟踪得到初始化位姿(跟踪上一帧/跟踪参考帧/重定位)

参考

本文做匹配的过程中，是用的都是ORB特征描述子。
先在8层图像金字塔中，提取FAST特征点。
提取特征点的个数根据图像分辨率不同而不同，高分辨率的图像提取更多的角点。
然后对检测到的特征点用ORB来描述，用于之后的匹配和识别。
跟踪这部分主要用了几种模型：
运动模型（Tracking with motion model）、
关键帧（Tracking with reference keyframe）和
重定位（Relocalization）。

```
if(mVelocity.empty() || mCurrentFrame.mnId<mnLastRelocFrameId+2)
// 没有移动(跟踪参考关键帧的运动模型是空的)  或 刚完成重定位不久
{
    bOK = TrackReferenceKeyFrame();// 跟踪参考帧模式
}
else
{
    bOK = TrackWithMotionModel();// 跟踪上一帧模式
    if(!bOK)//没成功
        bOK = TrackReferenceKeyFrame();//再次使用 跟踪参考帧模式
}
if(!bOK)// 当前帧与最近邻关键帧的匹配也失败了，那么意味着需要重新定位才能继续跟踪。
{       // 此时，只有去和所有关键帧匹配，看能否找到合适的位置。
    bOK = Relocalization();//重定位  BOW搜索，PnP 3d-2d匹配 求解位姿
}
```

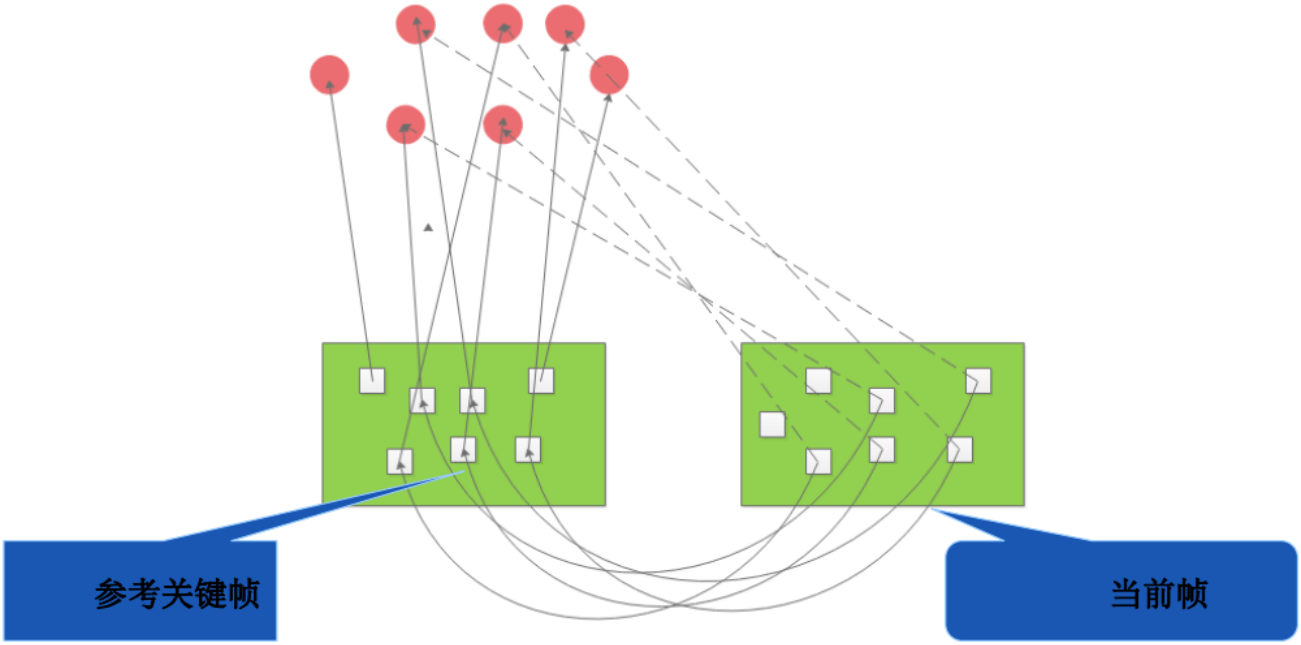
a. Tracking::TrackWithMotionModel() 跟踪上一帧模式，两帧相差不大，对应位置区域领域半径搜索匹配点对




```
参考：
        这个模型是假设物体处于匀速运动，例如匀速运动的汽车、机器人、行人等，
        就可以用上一帧的位姿和速度来估计当前帧的位姿。
        使用的函数为 Tracking::TrackWithMotionModel()。
        这里匹配是通过投影来与上一帧看到的地图点匹配，使用的是
        matcher.SearchByProjection()。

思想：
        移动模式跟踪  跟踪前后两帧  得到 变换矩阵。
        上一帧的地图3d点反投影到当前帧图像像素坐标上，在不同尺度下不同的搜索半径内，
        做描述子匹配 搜索 可以加快匹配。
        在投影点附近根据描述子距离进行匹配（需要>20对匹配，否则匀速模型跟踪失败，
        运动变化太大时会出现这种情况），然后以运动模型预测的位姿为初值，优化当前位姿，
        优化完成后再剔除外点，若剩余的匹配依然>=10对，
        则跟踪成功，否则跟踪失败，需要Relocalization：
步骤：
    步骤1：创建 ORB特征点匹配器 最小距离 < 0.9*次小距离 匹配成功
            ORBmatcher matcher(0.9,true);
    步骤2：更新上一帧的位姿和地图点
            Tracking::UpdateLastFrame();
            上一帧位姿 = 上一帧到其参考帧位姿*其参考帧到世界坐标系(系统第一帧)位姿
            后面单目不执行
            双目或rgbd相机，根据深度值为上一帧产生新的MapPoints
    步骤3：使用当前的运动速度(之前前后两帧位姿变换)和上一帧的位姿来初始化 当前帧的位姿
R,t
    步骤4：在当前帧和上一帧之间搜索匹配点（需要>20对匹配，否则匀速模型跟踪失败
            ORBmatcher::SearchByProjection(mCurrentFrame,mLastFrame,th,...)
            通过投影(使用当前帧的位姿R,t)，对上一帧的特征点(地图点)进行跟踪.
            上一帧中包含了MapPoints，对这些MapPoints进行跟踪tracking，由此增加当前
帧的MapPoints.
```

```
            上一帧3d点投影到当前坐标系下，在该2d点半径th范围内搜索可以匹配的匹配点，
            遍历可以匹配的点，计算描述子距离，记录最小的匹配距离，小于阈值的，再记录匹配点特征方向差值
            如果需要进行方向验证，剔除方向差直方图统计中，方向差值数量少的点对，保留前三个数量多的点对。
    步骤5：如果找到的匹配点对如果少于20，则扩大搜索半径th=2*th,使用ORBmatcher::SearchByProjection()再次进行搜索。
    步骤6：使用匹配点对对当前帧的位姿进行优化 G2O图优化
            Optimizer::PoseOptimization(&mCurrentFrame);// 仅仅优化单个普通帧的位姿,地图点不优化
            输入前一帧3d点 和 当前帧2d点对，以及帧的初始化位姿Tcw
            3D-2D 最小化重投影误差 e = (u,v) - project(Tcw*Pw)，只优化Frame的Tcw，不优化MapPoints的坐标。
                1. 顶点 Vertex: g2o::VertexSE3Expmap()，初始值为当前帧的Tcw
                2. 边 Edge:
                        单目
                    - g2o::EdgeSE3ProjectXYZOnlyPose()，一元边 BaseUnaryEdge
                        + 顶点 Vertex：待优化当前帧的Tcw
                        + 测量值 measurement：MapPoint在当前帧中的二维位置(u,v)
                        + 误差信息矩阵 InfoMatrix: Eigen::Matrix2d::Identity()*invSigma2(与特征点所在的尺度有关)
                        + 附加信息： 相机内参数： e->fx fy cx cy
                                     3d点坐标   ： e->Xw[0] Xw[1] Xw[2] 2d点对应的上一帧的3d点
                        双目
                    - g2o::EdgeStereoSE3ProjectXYZOnlyPose(), BaseUnaryEdge
                        + Vertex：待优化当前帧的Tcw
                        + measurement：MapPoint在当前帧中的二维位置(ul,v, ur) 左相机3d点 右相机横坐标匹配点坐标ur
                        + InfoMatrix: invSigma2(与特征点所在的尺度有关)
                        + 附加信息： 相机内参数： e->fx fy cx cy
                                     3d点坐标   ： e->Xw[0] Xw[1] Xw[2] 2d点对应的上一帧的3d点
                优化多次，根据误差，更新2d-3d匹配质量内外点标记，当前帧设置优化后的位姿。
    步骤7：如果2d-3d匹配效果差，被标记为外点，则当前帧2d点对于的3d点设置为空，留着以后再优化
    步骤8：根据内点的匹配数量，判断 跟踪上一帧是否成功。
```

b. Tracking::TrackReferenceKeyFrame() 跟踪参考帧模式，bow特征向量加速匹配



思想:
当使用运动模式匹配到的特征点数较少时，就会选用关键帧模式。
即尝试和最近一个关键帧去做匹配。
为了快速匹配，本文利用了bag of words（BoW）来加速匹配。
首先，计算当前帧的BoW，并设定初始位姿为上一帧的位姿；
其次，根据两帧的BoW特征向量同属于一个node下来加速搜索匹配点对，使用函数matcher.SearchByBoW()；
最后，利用匹配的特征优化位姿。

步骤：
步骤1：创建 ORB特征点匹配器 最小距离 < 0.7*次小距离 匹配成功
ORBmatcher matcher(0.7,true);
步骤2: 在当前帧 和 参考关键帧帧之间搜索匹配点对(需要>15对匹配点对，否者失败)

ORBmatcher::SearchByBoW(mpReferenceKF,mCurrentFrame,vpMapPointMatches);
计算当前帧 和 参考帧 orb特征的 词袋表示的 词典表示向量，属于同一个词典node下的点才可能是匹配点对
遍历所有的参考帧特征点 和 当前帧特征点：
同一个词典node下，有一些属于参考帧的点，也有一些属于当前帧的点，其中有一些匹配点。
遍历属于一个node下的 参考帧的点
遍历对应node下的当前帧的点
计算orb描述子之间的距离
记录最小的距离和次小的距离
最小的距离小于阈值，且最小的距离明显小于次小的距离的话，
记录次匹配点对特征的方向差，统计到方向差直方图
如果需要根据方向一致性剔除不好的匹配点：
计算方向差直方图中三个计数最多的方向差，
根据数量差异选择保留第一/第一+第二/第一+第二+第三方向差对应的匹配点对。
步骤3：设置当前帧位置为上一帧的位姿，使用步骤2得到的匹配点对3d-2d点对，使用BA 进行 当前帧位姿的优化
Optimizer::PoseOptimization(&mCurrentFrame);

```
                输入前一帧3d点 和 当前帧2d点对，以及帧的初始化位姿Tcw
                3D-2D 最小化重投影误差 e = (u,v) - project(Tcw*Pw)，只优化Frame
的Tcw，不优化MapPoints的坐标。
                1. 顶点 Vertex: g2o::VertexSE3Expmap()，初始值为当前帧的Tcw
                2. 边 Edge:
                        单目
                    - g2o::EdgeSE3ProjectXYZOnlyPose()，一元边
BaseUnaryEdge
                        + 顶点 Vertex：待优化当前帧的Tcw
                        + 测量值 measurement：MapPoint在当前帧中的二维位置
(u,v)
                        + 误差信息矩阵 InfoMatrix:
Eigen::Matrix2d::Identity()*invSigma2(与特征点所在的尺度有关)
                        + 附加信息： 相机内参数： e->fx fy cx cy
                                     3d点坐标   ： e->Xw[0] Xw[1] Xw[2] 2d点对
应的上一帧的3d点
                        双目
                    - g2o::EdgeStereoSE3ProjectXYZOnlyPose(),
BaseUnaryEdge
                        + Vertex：待优化当前帧的Tcw
                        + measurement：MapPoint在当前帧中的二维位置(ul,v, ur)
左相机3d点 右相机横坐标匹配点坐标ur
                        + InfoMatrix: invSigma2(与特征点所在的尺度有关)
                        + 附加信息： 相机内参数： e->fx fy cx cy
                                     3d点坐标   ： e->Xw[0] Xw[1] Xw[2] 2d点对
应的上一帧的3d点
                优化多次，根据误差，更新2d-3d匹配质量内外点标记，当前帧设置优化后的
位姿。
        步骤4：如果2d-3d匹配效果差，被标记为外点，则当前帧2d点对于的3d点设置为空，留着
以后再优化
        步骤5：根据内点的匹配数量，判断 跟踪上一帧是否成功。
```

c. bool Tracking::Relocalization() 上面两种模式都没有跟踪成功，需要使用重定位模式，使用orb字典编码在关键帧数据库中找相似的关键帧，进行匹配跟踪

```
思想：
    位置丢失后，需要在之前的关键帧中匹配最相近的关键帧，进而求出位姿信息。
    使用当前帧的BoW特征映射，在关键帧数据库中寻找相似的候选关键帧，因为这里没有好的初始
位姿信息，
    需要使用传统的3D-2D匹配点的EPnP算法来求解一个初始位姿，之后再使用最小化重投影误差来
优化更新位姿。
步骤：
    步骤1：计算当前帧的BoW映射
            词典 N个M维的单词，一帧的描述子，n个M维的描述子，生成一个 N*1的向量，记录一
帧的描述子使用词典单词的情况。
    步骤2：在关键帧数据库中找到与当前帧相似的候选关键帧组，词典单词线性表示向量距离较近的
一些关键帧。
    步骤3：创建 ORB特征点匹配器 最小距离 < 0.75*次小距离 匹配成功。
           ORBmatcher matcher(0.75,true);
    步骤4：创建 PnPsolver 位姿变换求解器数组，对应上面的多个候选关键帧。
    步骤5：遍历每一个候选关键帧使用BOW特征向量加速匹配,如果匹配点数少于15，跳过此候选关
```

键帧。

ORBmatcher::SearchByBoW(mpReferenceKF,mCurrentFrame,vpMapPointMatches);
计算当前帧 和 参考帧 orb特征的 词袋表示的 词典表示向量，属于同一个词典node下的点才可能是匹配点对
遍历所有的参考帧特征点 和 当前帧特征点：
同一个词典node下，有一些属于参考帧的点，也有一些属于当前帧的点，其中有一些匹配点。
遍历属于一个node下的 参考帧的点
遍历对应node下的当前帧的点
计算orb描述子之间的距离
记录最小的距离和次小的距离
最小的距离小于阈值，且最小的距离明显小于次小的距离的话，
记录次匹配点对特征的方向差，统计到方向差直方图
如果需要根据方向一致性剔除不好的匹配点：
计算方向差直方图中三个计数最多的方向差，
根据数量差异选择保留第一/第一+第二/第一+第二+第三方向差对应的匹配点对。

步骤6：使用PnPsolver 位姿变换求解器,更加3d-2d匹配点，s1 * (u，v，1) = T * (X, Y, Z, 1) = T * P
6点直接线性变换DLT,后使用QR分解得到 R,t, 或者使用(P3P)，3点平面匹配算法求解。
这里可参考 文档--双目slam基础.md--里面的3d-2d匹配点对求解算法。
这里会结合 Ransac 随采样序列一致性算法，来提高求解的鲁棒性。

步骤7：设置上面PnP算法求解的位置为当前帧的初始位姿，使用最小化重投影误差BA算法来优化位姿
Optimizer::PoseOptimization(&mCurrentFrame);
输入前一帧3d点 和 当前帧2d点对，以及帧的初始化位姿Tcw
3D-2D 最小化重投影误差 e = (u,v) - project(Tcw*Pw)，只优化Frame的Tcw，不优化MapPoints的坐标。
1. 顶点 Vertex: g2o::VertexSE3Expmap()，初始值为当前帧的Tcw
2. 边 Edge:
单目
- g2o::EdgeSE3ProjectXYZOnlyPose()，一元边 BaseUnaryEdge
+ 顶点 Vertex：待优化当前帧的Tcw
+ 测量值 measurement：MapPoint在当前帧中的二维位置(u,v)
+ 误差信息矩阵 InfoMatrix: Eigen::Matrix2d::Identity()*invSigma2(与特征点所在的尺度有关)
+ 附加信息： 相机内参数： e->fx fy cx cy
3d点坐标 ： e->Xw[0] Xw[1] Xw[2] 2d点对应的上一帧的3d点
双目
- g2o::EdgeStereoSE3ProjectXYZOnlyPose(), BaseUnaryEdge
+ Vertex：待优化当前帧的Tcw
+ measurement：MapPoint在当前帧中的二维位置(ul,v, ur) 左相机3d点 右相机横坐标匹配点坐标ur
+ InfoMatrix: invSigma2(与特征点所在的尺度有关)
+ 附加信息： 相机内参数： e->fx fy cx cy
3d点坐标 ： e->Xw[0] Xw[1] Xw[2] 2d点对应的上一帧的3d点

优化多次，根据误差，更新2d-3d匹配质量内外点标记，当前帧设置优化后的位姿。

步骤8：如果优化时记录的匹配点对内点数量少于50，会把参考关键中还没有在当前帧有2d匹配的点反投影到当前帧下，再次搜索2d匹配点

ORBmatcher::SearchByProjection();

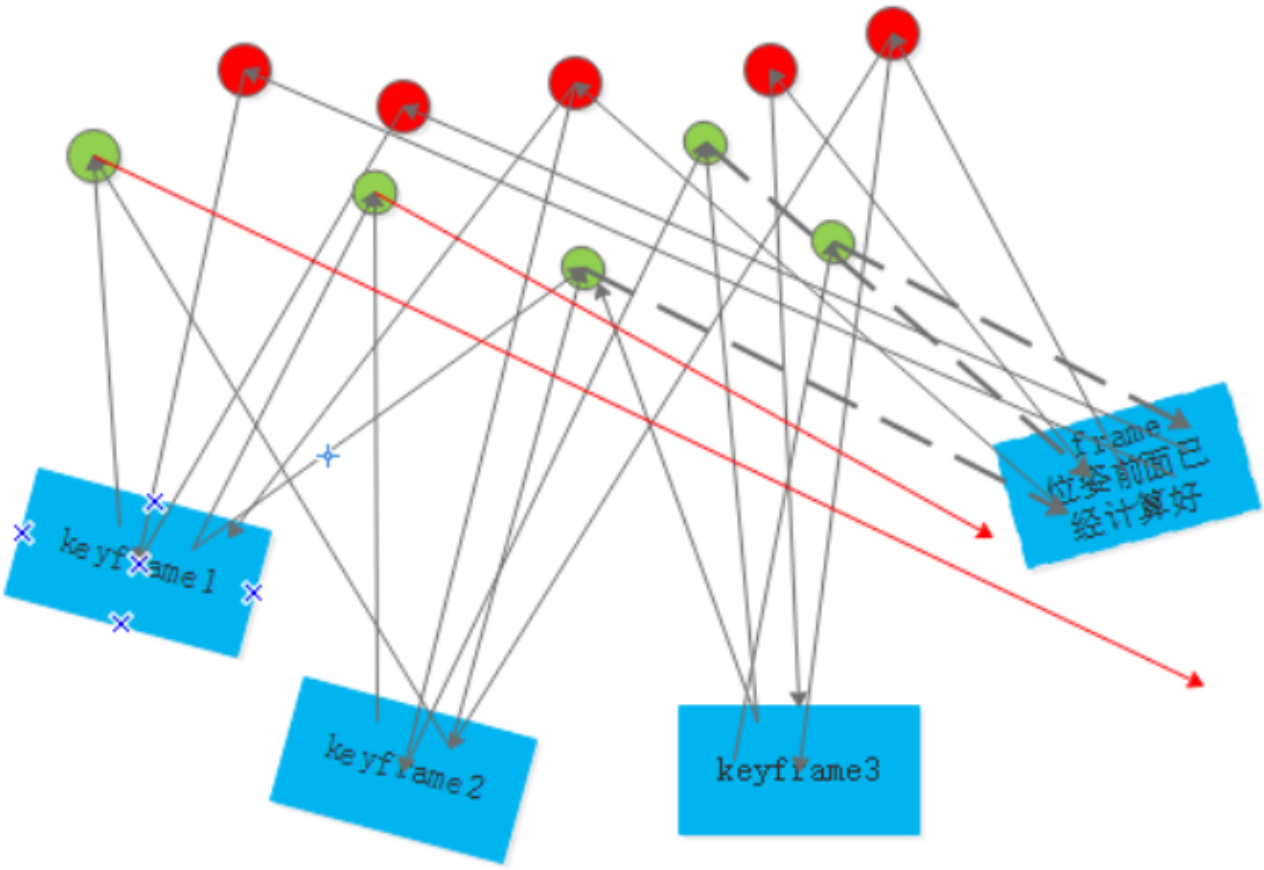
步骤9： 如果再次反投影搜索的匹配点数量和之前得到的匹配点数量 大于50，再次使用Optimizer::PoseOptimization() 优化算法进行优化

步骤10：如果上面优化后的内点数量还比较少可以再次搜索matcher2.SearchByProjection(),再次优化 Optimizer::PoseOptimization()

步骤11：如果内点数量 大于等于50 ，则重定位成功。

4.4 Tracking::TrackLocalMap() 跟踪局部地图，多帧局部地图G2O优化位姿, 新建关键帧

参考



思想：

上面完成初始位姿的跟踪后，需要 使用局部地图(参考帧的一二级共视帧组成) 来 进行局部地图优化，来提高鲁棒性。

局部地图中与当前帧有相同点的关键帧序列成为一级相关帧K1，

而与一级相关帧K1有共视地图点的关键帧序列成为二级相关帧K2，

把局部地图中的局部地图点，投影到当前帧上，如果在当前帧的视野内

使用 位姿优化 Optimizer::PoseOptimization(&mCurrentFrame)， 进行优化，

更新 地图点的信息(关键帧的观测关系)

步骤：
步骤1：首先对局部地图进行更新(UpdateLocalMap) 生成对应当前帧的局部地图(小图)
Tracking::UpdateLocalMap();
---更新局部关键帧：Tracking::UpdateLocalKeyFrames();
始终限制局部关键帧(小图中关键帧的数量)数量不超过80
包含三个部分：
1. 共视化程度高的关键帧： 观测到当前帧的地图点次数多的关键帧；
2. 这些关键帧的孩子关键帧；(这里没看到具体的方式，应该就是根据时间顺序记录了一些父子关键帧)
3. 这些关键帧的父亲关键帧。
---更新局部地图点：Tracking::UpdateLocalPoints();
所有局部关键帧 包含的地图点构成 局部地图点
遍历每一个局部关键帧的每一个地图点：
加入到局部地图点中：mvpLocalMapPoints.push_back(pMP)；
同时设置地图点更新标志，来避免重复添加出现在多帧上的地图点。
步骤2：在局部地图中为当前帧搜索匹配点对
Tracking::SearchLocalPoints();// 在对应当前帧的局部地图内搜寻和 当前帧地图点匹配点的 局部地图点
1. 遍历当前帧的特征点，如果已经有相应的3D地图点,则进行标记，不需要进行重投影匹配，并且标记已经被遍历过
2. 遍历局部地图的所有地图点，如果没有被遍历过，把地图点反投影到当前帧下，保留在当前帧视野内的地图点
3. 根据反投影后的2d位置，设置一个半径为th的范围进行搜索匹配点
SearchByProjection(mCurrentFrame,mvpLocalMapPoints,th);
遍历可以匹配的点，计算描述子距离，记录最小的匹配距离，小于阈值的，再记录匹配点特征方向差值
如果需要进行方向验证，剔除方向差直方图统计中，方向差值数量少的点对，保留前三个数量多的点对。
步骤3：使用之前得到的初始位姿和 在局部地图中搜索到的3d-2d匹配点对，使用最小化重投影误差BA算法来优化位姿
Optimizer::PoseOptimization(&mCurrentFrame);
输入前一帧3d点 和 当前帧2d点对，以及帧的初始化位姿Tcw
3D-2D 最小化重投影误差 e = (u,v) - project(Tcw*Pw)，只优化Frame的Tcw，不优化MapPoints的坐标。
1. 顶点 Vertex: g2o::VertexSE3Expmap()，初始值为当前帧的Tcw
2. 边 Edge:
单目
- g2o::EdgeSE3ProjectXYZOnlyPose()，一元边 BaseUnaryEdge
+ 顶点 Vertex：待优化当前帧的Tcw
+ 测量值 measurement：MapPoint在当前帧中的二维位置(u,v)
+ 误差信息矩阵 InfoMatrix: Eigen::Matrix2d::Identity()*invSigma2(与特征点所在的尺度有关)
+ 附加信息： 相机内参数： e->fx fy cx cy
3d点坐标 ： e->Xw[0] Xw[1] Xw[2] 2d点对应的上一帧的3d点
双目
- g2o::EdgeStereoSE3ProjectXYZOnlyPose(), BaseUnaryEdge
+ Vertex：待优化当前帧的Tcw

```
                              + measurement：MapPoint在当前帧中的二维位置(ul,v, ur)
左相机3d点 右相机横坐标匹配点坐标ur
                              + InfoMatrix: invSigma2(与特征点所在的尺度有关)
                              + 附加信息： 相机内参数： e->fx fy cx cy
                                           3d点坐标   ： e->Xw[0] Xw[1] Xw[2] 2d点对
应的上一帧的3d点
                    优化多次，根据误差，更新2d-3d匹配质量内外点标记，当前帧设置优化后的
位姿。
      步骤4：更新地图点状态
      步骤5：如果刚刚进行过重定位 则需要 内点匹配点对数 大于50 才认为 成功
             正常情况下，找到的内点匹配点对数 大于30 算成功
```

4.5 跟踪成功之后判断是否需要新建关键帧

判断是否需要创建关键帧 Tracking::NeedNewKeyFrame();

```
确定关键帧的标准如下：
          (1) 在上一个全局重定位后，又过了20帧                  (时间过了许久)；
          (2) 局部建图闲置，或在上一个关键帧插入后，又过了20帧(时间过了许久)；
          (3) 当前帧跟踪到点数量比较少，tracking质量较弱      (跟踪要跪的节奏)；
          (4) 当前帧跟踪到的点比参考关键帧的点少90%           (环境变化较大了)。
步骤：
      步骤1：系统模式判断，如果仅仅需要跟踪定位，不需要建图，那么不需要新建关键帧。
      步骤2：根据地图中关键帧的数量设置一些参数(系统一开始关键帧少的时候，可以放宽一些条
件，多创建一些关键帧)
      步骤3：很长时间没有插入关键帧
             bool c1a = mCurrentFrame.mnId >= mnLastKeyFrameId + mMaxFrames;
      步骤4：在过去一段时间但是局部建图闲置
             bool c1b = (mCurrentFrame.mnId >= mnLastKeyFrameId+mMinFrames
&& bLocalMappingIdle
      步骤5：当前帧跟踪到点数量比较少，tracking质量较弱
             bool c1c = mnMatchesInliers < nRefMatches*0.25
      步骤6：上面条件成立之前必须当 当前帧与之前参考帧（最近的一个关键帧）重复度不是太
高。
             bool c2 = ((mnMatchesInliers < nRefMatches*thRefRatio||
bNeedToInsertClose) && mnMatchesInliers>15)
      步骤7：需要新建关键帧的条件 (c1a || c1b || c1c) && c2
      步骤8：当待插入的关键帧队列里关键帧数量不多时在插入，队列里不能阻塞太多关键帧。
```

创建关键帧 Tracking::CreateNewKeyFrame();

```
步骤：
      步骤1：将当前帧构造成关键帧
      步骤2：将当前关键帧设置为当前帧的参考关键帧
      步骤3：对于双目或rgbd摄像头，为当前帧生成新的MapPoints
             将深度距离比较近的点包装成MapPoints
               这些添加属性的操作是每次创建MapPoint后都要做的：
```

```
地图点关键关键帧
关键帧关联地图点
地图点更新最优区别性的描述子
地图点更新深度
地图添加地图点
```

局部建图线程 LocalMapping::LocalMapping()

```
功能总览：
        LocalMapping作用是将Tracking中送来的关键帧放在mlNewKeyFrame列表中；
                1. 处理新关键帧，
                2. 生成新地图点，
                3. 地图点检查剔除，
                4. Local BA，
                5. 关键帧剔除。
        主要工作在于维护局部地图，也就是SLAM中的Mapping。
        Tracking线程 只是判断当前帧是否需要加入关键帧，并没有真的加入地图，
        因为Tracking线程的主要功能是局部定位， 而处理地图中的关键帧，地图点，包括如何加入，
        如何删除的工作是在LocalMapping线程完成的

A、插入新的关键帧（KeyFrame Insertion）

        每插入一个关键帧，首先更新共视图，
        包括增加一个新的节点Ki和更新关键帧中拥有共视点的边信息。
        然后更新本质图，即更新关键帧中拥有最多共视点的边信息。
        最后计算当前关键帧的词袋模型表示。
B、创建新的地图点（New MapPoint Creation）
        通过三角化共视图中相连关键帧的ORB特征点来创建新的地图点。
        对于关键帧Ki中的未匹配ORB特征点，在其他关键帧中查找匹配点。
        删除不满足对极约束的匹配点对，
        然后三角化匹配点对生成新的地图点，
        检查是否满足景深为正，
        视差和重投影误差、
        尺度一致性。
        最后将该地图点投影到其它关键帧中寻找匹配。
C、地图点剔除（Recent MapPoint Culling）
        一个地图点想要保留在地图中必须通过严格的检验，
        即它能够在被创建后的连续三个关键帧被追踪到，
        这样是为了剔除错误三角化的地图点。
        一个良好的地图点需要满足以下两个条件：
                1）超过25%的帧数在理论上可以观察到该地图点；
                2）该地图点创建后能够被至少三个连续关键帧追踪到。
        当然即使地图点满足以上条件，还是有可能会被删除，
        比如某个关键帧被删除或者当做局部BA的外点被剔除等。
D、局部地图的BA优化（Local Bundle Adjustment）
        局部BA优化的对象有当前关键帧Ki、
        共视图中的相邻关键帧Kc和这些关键帧观测到的地图点，
        另外可以观测到这些地图点但是不与当前帧相邻的关键帧仅作为约束条件而不作为优化的对象。
```

```
E、关键帧剔除（Local Keyframe Culling）
        随着新关键帧的插入，为了不增加BA优化的压力，
        和维持系统的可持续运行，需要删除冗余的关键帧。
        如果某一关键帧观测到的90%地图点的能够被至少三个其他关键帧观测到，
        那么剔除该关键帧。
```

LocalMapping::Run() 主程序

```
void LocalMapping::Run()：
        局部建图 ：处理新的关键帧 KeyFrame 完成局部地图构建，
        插入关键帧 ------>
        处理地图点(筛选生成的地图点 生成地图点)  -------->
        局部 BA最小化重投影误差  -调整-------->
        筛选 新插入的 关键帧

步骤：
        mlNewKeyFrames     list 列表队列存储关键帧
        步骤1：设置进程间的访问标志 告诉Tracking线程，LocalMapping线程正在处理新的关键帧，处于繁忙状态
        步骤2：检查队列  LocalMapping::CheckNewKeyFrames(); 等待处理的关键帧列表不为空
        步骤3：处理新关键帧，计算关键帧特征点的词典单词向量BoW映射，将关键帧插入地图
               更新地图点MapPoints 和 关键帧 KepFrame 的关联关系
UpdateConnections() 更新关联关系
               LocalMapping::ProcessNewKeyFrame();
        步骤4:创建新的地图点 相机运动过程中与相邻关键帧通过三角化恢复出一些新的地图点MapPoints
               LocalMapping::CreateNewMapPoints();
        步骤5：对新添加的地图点进行融合处理 ,剔除 地图点 MapPoints
               对于 ProcessNewKeyFrame 和 CreateNewMapPoints 中 最近添加的MapPoints进行检查剔除
               删除地图中新添加的但 质量不好的 地图点
               LocalMapping::MapPointCulling();
               a)  IncreaseFound / IncreaseVisible < 25%
               b) 观测到该 点的关键帧太少
        步骤6：相邻帧地图点融合
               LocalMapping::SearchInNeighbors();
               检测当前关键帧和相邻 关键帧(两级相邻) 中 重复的 地图点

        步骤7：局部地图BA 最小化重投影误差

Optimizer::LocalBundleAdjustment(mpCurrentKeyFrame,&mbAbortBA, mpMap);
               和当前关键帧相邻的关键帧 中相匹配的 地图点对 最局部 BA最小化重投影误差优化点坐标 和 位姿

        步骤7：关键帧融合，剔除当前帧相邻的关键帧中冗余的关键帧
               LocalMapping::KeyFrameCulling();
               其90%以上的 地图点 能够被其他 共视 关键帧(至少3个) 观测到，认为该关键帧多余，可以删除
        步骤8：将当前帧加入到闭环检测队列中 mpLoopCloser
        步骤9：等待线程空闲 完成一帧关键帧的插入融合工作
```

```
        步骤10：告诉 Tracking 线程，Local Mapping 线程现在空闲，可一处理接收下一个
关键帧。
```

恢复3d点 LocalMapping::CreateNewMapPoints()

```
思想：
        相机运动过程中和共视程度比较高的关键帧，通过三角化恢复出一些MapPoints地图点
        根据当前关键帧恢复出一些新的地图点，不包括和当前关键帧匹配的局部地图点（已经在
ProcessNewKeyFrame中处理）
        先处理新关键帧与局部地图点之间的关系，然后对局部地图点进行检查，
        最后再通过新关键帧恢复 新的局部地图点：CreateNewMapPoints()
步骤：
   步骤1：在当前关键帧的 共视关键帧 中找到 共视程度 最高的前nn帧 相邻帧vpNeighKFs
   步骤2：遍历和当前关键帧 相邻的 每一个关键帧vpNeighKFs
   步骤3：判断相机运动的基线在（两针间的相机相对坐标）是不是足够长
   步骤4：根据两个关键帧的位姿计算它们之间的基本矩阵 F =  inv(K1 转置) * t12 叉乘
R12 * inv(K2)
   步骤5：通过帧间词典向量加速匹配，极线约束限制匹配时的搜索范围，进行特征点匹配
   步骤6：对每对匹配点 2d-2d 通过三角化生成3D点,和 Triangulate函数差不多
        步骤6.1：取出匹配特征点
        步骤6.2：利用匹配点反投影得到视差角
                 用来决定使用三角化恢复(视差角较大) 还是 直接2d点反投影(视差角较小)
        步骤6.3：对于双目，利用双目基线 深度 得到视差角
        步骤6.4：视差角较大时 使用 三角化恢复3D点 (两个点,四个方程,奇异值分解求解)
        步骤6.4：对于双目 视差角较小时， 2d点利用深度值 反投影 成三维点 ,单目的话直接跳
过
        步骤6.5：检测生成的3D点是否在相机前方(Z>0)
        步骤6.6：计算3D点在当前关键帧下的重投影误差,误差较大的跳过
        步骤6.7：计算3D点在邻接关键帧 下的重投影误差，误差较大的跳过
        步骤6.9：三角化生成3D点成功，构造成地图点 MapPoint
        步骤6.9：为该MapPoint添加属性
                          地图点关键关键帧
                          关键帧关联地图点
                          地图点更新最优区别性的描述子
                          地图点更新深度
                          地图添加地图点
        步骤6.10：将新产生的点放入检测队列 mlpRecentAddedMapPoints  交给
MapPointCulling() 检查生成的点是否合适
```

4.7 闭环检测线程 LoopClosing

参考

```
思想：
    闭环检测线程
    对新加入的关键帧，
        1. 进行回环检测(WOB 二进制词典匹配检测，通过Sim3算法计算相似变换)--------->
        2. 闭环校正(闭环融合 和 图优化)
```

A、检测闭环候选帧（Loop Candidates Detection） 首先我们计算当前关键帧Ki与共视图中相邻关键帧Kc词袋向量的相似度， 并保留最小相似度分数Smin。 然后查询关键帧数据库，剔除相似度分数低于最小分数的关键帧和直接相邻的关键帧， 将其他关键帧作为闭环候选帧。 最后通过检测连续三个闭环候选帧的一致性来选择闭环帧。 B、计算Sim3相似变换（Computer the Similarity Transformation） 在单目SLAM系统中，地图（关键帧和地图点）有七个自由度可以漂移， 包括三个平移自由度、三个旋转自由度和一个尺度自由度。 通过计算当前帧Ki与关键帧Kl的相似变换， 不仅可以计算出闭环处的累计误差，而且可以判断该闭环的有效性。 首先计算当前帧和闭环候选帧中ORB特征点对应的地图点， 得到当前帧与每个闭环候选帧的3D-3D对应关系。 然后通过对每个闭环候选帧采用RANSAC算法迭代，试图计算出相似变换。 如果某闭环候选帧的相似变换具有足够多的内点， 通过寻找更多匹配点优化，如果仍具有足够多的内点，则该闭环候选帧为闭环帧。 C、闭环融合（Loop Fusion） 从上一步可以得到当前帧的闭环帧和相似变换， 回环优化的第一步就是在当前帧融合地图点和在共视图中插入新的边信息。 首先当前关键帧Ki的位姿Tiw通过相似变换校正，并校正当前关键帧的相邻帧Kc的位姿Tcw。 回环帧Kl和它相邻帧Klc的地图点投影到当前帧Ki和它的相邻帧Kc中寻找匹配点对， 所有匹配的3D-3D地图点和相似变换的内点进行融合。 所有参与融合的关键帧都更新他们在共视图中的边信息，并创建与回环帧相邻的边。 D、基于本质图的优化（Essential Graph Optimization） 为了有效的闭合回环，我们在本质图上进行位姿图优化。

LoopClosing::Run() 闭环检测步骤：

```
   mlpLoopKeyFrameQueue  闭环检测关键帧队列（localMapping线程 加入的）
   只要闭环检测关键帧队列不为空下面的检测会一直执行
步骤0：进行闭环检测 LoopClosing::DetectLoop()
        步骤1：从队列中抽取一帧
              mpCurrentKF = mlpLoopKeyFrameQueue.front();
              mlpLoopKeyFrameQueue.pop_front();//出队
        步骤2：判断距离上次闭环检测是否超过10帧，如果数量过少则不进行闭环检测。
        步骤3：遍历所有共视关键帧，计算当前关键帧与每个共视关键帧 的bow相似度得分，并得到最低得分minScore
              GetVectorCovisibleKeyFrames(); // 当前帧的所有 共视关键帧
              mpORBVocabulary->score(CurrentBowVec, BowVec);// bow向量相似性得分
        步骤4：在所有关键帧数据库中找出与当前帧按最低得分minScore 匹配得到的闭环候选帧 vpCandidateKFs
        步骤5：在候选帧中检测具有连续性的候选帧，相邻一起的分成一组，组与组相邻的再成组 (pKF, minScore)
        步骤6：找出与当前帧有 公共单词的 非相连 关键帧 lKFsharingwords
        步骤7：统计候选帧中 与 pKF 具有共同单词最多的单词数 maxcommonwords
        步骤8：得到阈值 minCommons = 0.8 × maxcommonwords
        步骤9：筛选共有单词大于 minCommons  且词带 BOW 匹配 最低得分  大于 minScore ， lscoreAndMatch
        步骤10：将存在相连的分为一组，计算组最高得分 bestAccScore，同时得到每组中得分最高的关键帧  lsAccScoreAndMatch
        步骤11：得到阈值 minScoreToRetain = 0.75  ×  bestAccScore
        步骤12：得到 闭环检测候选帧
步骤13：计算 闭环处两针的 相似变换  [sR|t]
        LoopClosing::ComputeSim3()
步骤14：闭环融合位姿图优化
       LoopClosing::CorrectLoop()
```

LoopClosing::ComputeSim3() 计算当前帧与闭环帧的Sim3变换

思想：
通过Bow词袋量化加速描述子的匹配，利用RANSAC粗略地计算出当前帧与闭环帧的Sim3（当前帧---闭环帧）
根据估计的Sim3，对3D点进行投影找到更多匹配，通过优化的方法计算更精确的Sim3（当前帧---闭环帧）
将闭环帧以及闭环帧相连的关键帧的MapPoints与当前帧的点进行匹配（当前帧---闭环帧+相连关键帧）
注意以上匹配的结果均都存在成员变量 mvpCurrentMatchedPoints 中，
实际的更新步骤见CorrectLoop()步骤3：Start Loop Fusion
步骤：
步骤1：遍历每一个闭环候选关键帧 构造 sim3 求解器
步骤2：从筛选的闭环候选帧中取出一帧关键帧pKF
步骤3：将当前帧mpCurrentKF与闭环候选关键帧pKF匹配 得到匹配点对
步骤3.1 跳过匹配点对数少的 候选闭环帧
步骤3.2： 根据匹配点对 构造Sim3求解器
步骤4：迭代每一个候选闭环关键帧 Sim3利用 相似变换求解器求解 候选闭环关键帧到档期帧的 相似变换
步骤5：通过步骤4求取的Sim3变换，使用sim3变换匹配得到更多的匹配点 弥补步骤3中的漏匹配
步骤6：G2O Sim3优化，只要有一个候选帧通过Sim3的求解与优化，就跳出停止对其它候选帧的判断
步骤7：如果没有一个闭环匹配候选帧通过Sim3的求解与优化 清空候选闭环关键帧
步骤8：取出闭环匹配上 关键帧的相连关键帧，得到它们的地图点MapPoints放入mvpLoopMapPoints
步骤9：将闭环匹配上关键帧以及相连关键帧的 地图点 MapPoints 投影到当前关键帧进行投影匹配 为当前帧查找更多的匹配
步骤10：判断当前帧 与检测出的所有闭环关键帧是否有足够多的MapPoints匹配
步骤11：满足匹配点对数>40 寻找成功 清空mvpEnoughConsistentCandidates

LoopClosing::CorrectLoop() 闭环融合 全局优化

步骤：
步骤1：通过求解的Sim3以及相对姿态关系，调整与 当前帧相连的关键帧 mvpCurrentConnectedKFs 位姿
以及这些 关键帧观测到的MapPoints的位置（相连关键帧---当前帧）
步骤2：用当前帧在闭环地图点 mvpLoopMapPoints 中匹配的 当前帧闭环匹配地图点 mvpCurrentMatchedPoints
更新当前帧 之前的 匹配地图点 mpCurrentKF->GetMapPoint(i)
步骤3：将闭环帧以及闭环帧相连的关键帧的 所有地图点 mvpLoopMapPoints 和 当前帧相连的关键帧的点进行匹配
步骤4：通过MapPoints的匹配关系更新这些帧之间的连接关系，即更新covisibility graph
步骤5：对Essential Graph（Pose Graph）进行优化，MapPoints的位置则根据优化后的位姿做相对应的调整
步骤6：创建线程进行全局Bundle Adjustment

mvpCurrentConnectedKFs 当前帧相关联的关键帧
vpLoopConnectedKFs 闭环帧相关联的关键帧 这些关键帧的地图点 闭环地图点

```
mvpLoopMapPoints
mpMatchedKF                 与当前帧匹配的   闭环帧
mpCurrentKF 当前关键帧      优化的位姿 mg2oScw      原先的地图点 mpCurrentKF-
>GetMapPoint(i)
mvpCurrentMatchedPoints    当前帧在闭环地图点中匹配的地图点   当前帧闭环匹配地图点
```

5. 数学理论总结

参考