**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。1. 这是首个基于单目，双目和RGB-D相机的开源SLAM方案，这个方案包括，回环检测，地图重用和重定位。

2. 结果说明，BA优化比ICP或者光度和深度误差最小方法的更加精确。

3. 通过匹配远处和近处的双目匹配的点和单目观测，双目的结果比直接使用双目系统更加精确。

4. 针对无法建图的情况，提出了一个轻量级的定位模式 ，能够更加有效的重用地图。

ORB\_SLAM2

1、简介

  ORB\_SLAM2由3+1个平行线程组成，包括跟踪、局部建图、回环检测以及在回环检测后的全局BA优化。之所以说是3+1，因为第四个线程仅在回环检测并确认后才执行。

（1）跟踪（Tracking）   
  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。

（2）建图（LocalMapping）   
  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。

（3）闭环检测（LoopClosing）

  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。

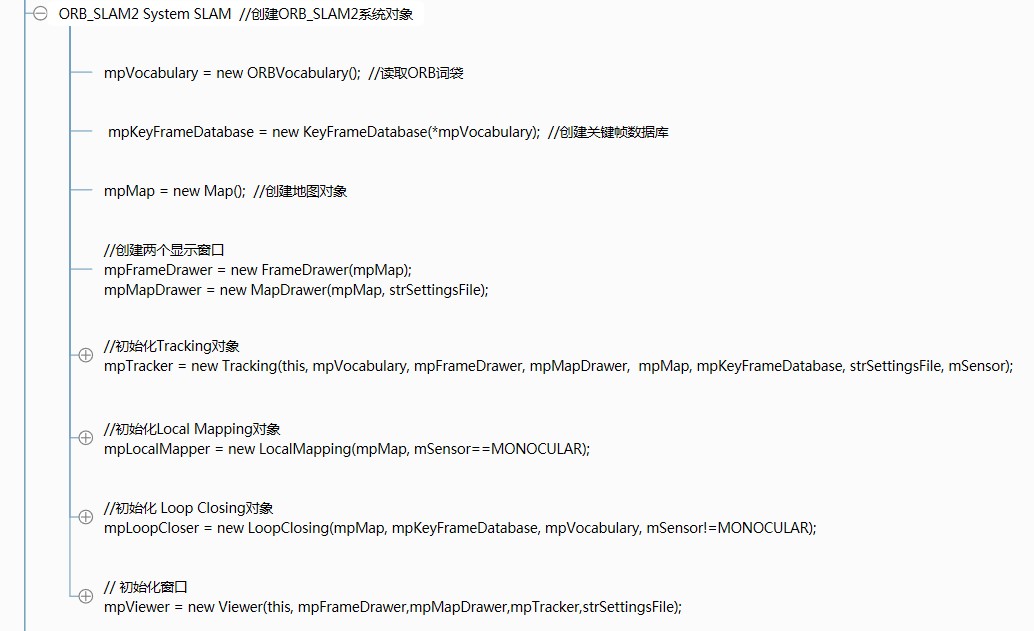
**一、整体框架**



**（1）首先使用LoadImages读取图片目录和时间戳文件   
（2）创建ORB\_SLAM2::System对象   
（3）循环读取数据   
 （3.1）读取图片   
 （3.2）读取时间戳   
 （3.3）将图片传给SLAM系统   
（4）关闭SLAM系统   
（5）将相机轨线保存到硬盘中**

## SLAM系统的创建

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、Mapping和LoopClosing。如下图所示：



**（1）创建了ORB词袋的对象**

**（2）创建了关键帧的数据库**

**（3）创建地图对象   
（4）创建两个显示窗口   
（5）初始化Tracking对象   
（6）初始化Local Mapping对象，并开启线程运行   
（7）初始化Loop Closing对象，并开启线程运行   
（8）初始化窗口，开启线程显示图像和地图点**

## 三、Tracking的运行

  上面我们提到Tracking的运行入口，回顾第一部分，我们在循环读取图片的时候，调用了一个**SLAM.TrackMonocular()**函数，这个函数就是在主线程里调用Tracking的函数入口了。所以Tracking是运行在主线程中，并且在每次读取一帧新的图像时运行一次。如下图所示：   


可以看到，跟了两步之后，就能很清晰地看到，程序将读取的图片转成灰度图，然后对帧进行特征点检测，就直接调用**Tracking.cc**中的Track()函数，进行跟踪。另外要注意的是，由于使用的是单目相机，所以在跟踪时需要判断是否初始化或有没有前序关键帧，如果没有，使用mpIniORBextractor的参数进行特征点检测。

# 自动地图初始化

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：用于平面场景的单应性矩阵H和用于非平面场景的基础矩阵F，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。

## 一、找到初始对应点

  在当前帧

中提取ORB特征点，与参考帧Fr进行匹配。如果匹配点对数过少，就重置参考帧。这一步骤在**Tracking.cc**中的**Tracking::MonocularInitialization**函数中。

int nmatches = matcher.SearchForInitialization(mInitialFrame,mCurrentFrame,mvbPrevMatched,mvIniMatches,100);

* 1

**nmatches**表示匹配到的对应点对数。

## 二、同时计算两个模型

  在找到对应点之后，开始调用**Initializer.cc**中的**Initializer::Initialized**函数进行初始化工作。为了计算R和t，ORB\_SLAM为了针对平面和非平面场景选择最合适的模型，同时开启了两个线程，分别计算单应性矩阵

和基础矩阵

。如下所示：

1. thread threadH(&Initializer::FindHomography,this,ref(vbMatchesInliersH), ref(SH), ref(H));
2. thread threadF(&Initializer::FindFundamental,this,ref(vbMatchesInliersF), ref(SF), ref(F));

这里的H和F分别满足下列关系：

线程threadH调用**Initializer::FindHomography**函数，计算单应性矩阵H，采用归一化的直接线性变换（normalized DLT）。线程threadF调用**Initializer::FindFundamental**函数计算基础矩阵F，使用归一化8点法。为了评估哪个模型更合适，文中使用和来计算各自的分值。分别调用**Initializer::CheckHomography**函数和**Initializer::CheckFundamental**函数进行计算，计算的方法如下所示，其中统一表示和：

其中，和表示对称的转换误差，分别是从当前帧到参考帧的变换误差和参考帧到当前帧的变换误差。这里：

## 三、模型选择

  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，而使用基础矩阵F恢复运动，需要场景是一个非平面、视差大的场景。这个时候，文中使用下面所示的一个机制，来估计两个模型的优劣：

当大于0.45时，选择从单应性变换矩阵还原运动。不过ORB\_SLAM2源代码中使用的是0.4作为阈值，如下：

1. *// Compute ratio of scores*
2. float RH = SH/(SH+SF);
4. *// Try to reconstruct from homography or fundamental depending on the ratio (0.40-0.45)*
5. if(RH>0.40)
6. return ReconstructH(vbMatchesInliersH,H,mK,R21,t21,vP3D,vbTriangulated,1.0,50);
7. else *//if(pF\_HF>0.6)*
8. return ReconstructF(vbMatchesInliersF,F,mK,R21,t21,vP3D,vbTriangulated,1.0,50);

## 四、运动恢复（sfm）

  选择好模型后，就可以恢复运动。   
（1）从单应性变换矩阵H中恢复   
  在求得单应性变化H后，本文使用FAUGERAS的论文[1]的方法，提取8种运动假设。这个方法通过可视化约束来测试选择合理的解。但是如果在低视差的情况下，点云会跑到相机的前面或后面，测试就会出现错误从而选择一个错误的解。文中使用的是直接三角化8种方案，检查两个相机前面具有较少的重投影误差情况下，在视图低视差情况下是否大部分云点都可以看到。如果没有一个解很合适，就不执行初始化，重新从第一步开始。这种方法在低视差和两个交叉的视图情况下，初始化程序更具鲁棒性。程序中调用**Initializer::ReconstructH**函数恢复运动。

(2)从基础矩阵F中恢复   
  在得到基础矩阵F，并且一直摄像机内参K的情况下，可以计算得到本质矩阵E，然后使用[2]中的方法，恢复出4个运动假设的解。这一部分的理论推导在之前博客做过介绍。其中基础矩阵F得到本质矩阵E的公式如下所示：

同样的，这4个解中只有一个是合理的，可以使用可视化约束来选择，本文使用与单应性矩阵做sfm一样的方法，即将4种解都进行三角化，然后从中选择出最合适的解。这里使用的是**Initializer::ReconstructF**函数。

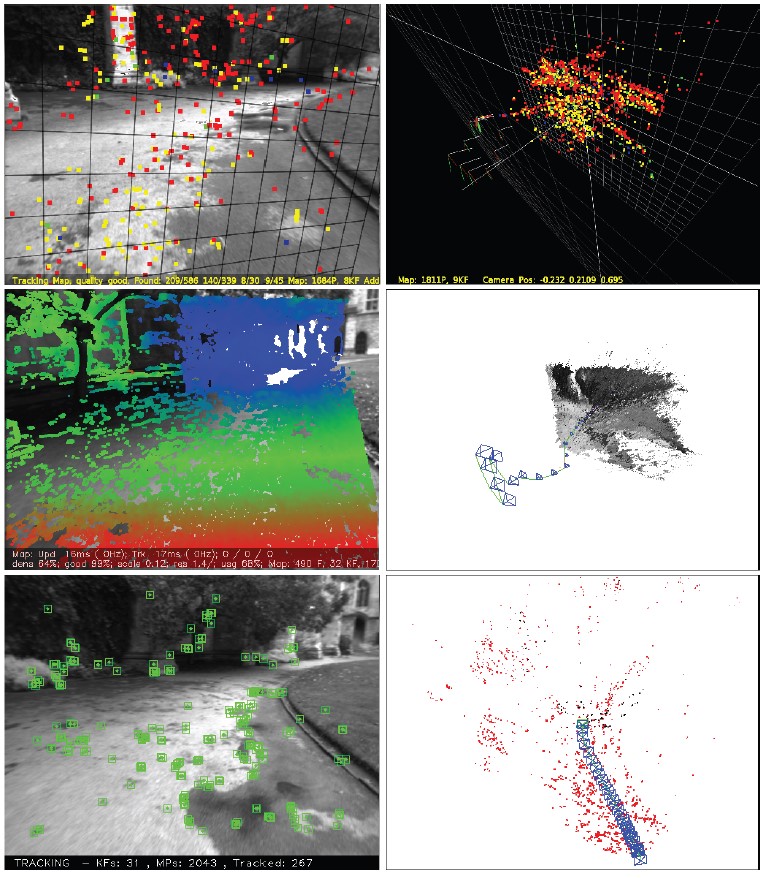
## 五、集束调整

  最后使用一个全局集束调整（BA)，优化初始化结果。这一部分是在**Tracking.cc**中的**CreateInitialMapMonocular()**函数中，使用了如下语句：

Optimizer::GlobalBundleAdjustemnt(mpMap,20);

* 1

  如下图所示，PTAM和LSD\_SLAM在这个数据集中，会将所有点初始化在一个平面上，而ORB\_SLAM会一直等到有足够的视差，才使用基础矩阵，得到最正确的初始化。由于ORB-SLAM对初始化的要求较高，因此初始化时可以选择一个特征丰富的场景，移动摄像机给它提供足够的视差。另外，由于坐标系会附着在初始化成功的那帧图像的位置，因此每次初始化不能保证在同一个位置。



# 跟踪

这一部分是ORB\_SLAM系统中最基本的一步，会对每一帧图像进行跟踪计算。Tracking线程运行在主线程中，主要思路是在当前帧和（局部）地图之间寻找尽可能多的对应关系，来优化当前帧的位姿。每次新采集到一帧图像，就是用下列接口将图像传入SLAM系统就行处理。该代码位于主程序中：

1. *// Pass the image to the SLAM system*
2. SLAM.TrackMonocular(im,tframe);

* 1
* 2

在检查完系统将模式切换为跟踪模式后，是用下面接口进入功能：

mpTracker->GrabImageMonocular(im,timestamp);

* 1

## 一、ORB提取

  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](https://www.baidu.com/s?wd=%E5%9B%BE%E5%83%8F%E9%87%91%E5%AD%97%E5%A1%94&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)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）。

## 二、从前一帧初始化位姿估计

  在成功与前面帧跟踪上之后，为了提高速率，本文使用与之前速率相同的**运动模式**来预测相机姿态，并搜索上一帧观测到的地图点。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**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()**。

1. if(mVelocity.empty() || mCurrentFrame.mnId<mnLastRelocFrameId+2)
2. {
3. bOK = TrackReferenceKeyFrame();
4. }
5. else
6. {
7. bOK = TrackWithMotionModel();
8. if(!bOK)
9. bOK = TrackReferenceKeyFrame();
10. }

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。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，根据位姿和BoW词典来寻找特征匹配，使用函数**matcher.SearchByBoW()**；最后，利用匹配的特征优化位姿。

## 三、通过全局重定位来初始化位姿估计

  假如使用上面的方法，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。重定位的入口如下：

bOK = Relocalization();

* 1

此时，只有去和所有关键帧匹配，看能否找到合适的位置。首先，计算当前帧的BOW向量，在关键帧词典数据库中选取若干关键帧作为候选。使用函数如下：

2. *// Relocalization is performed when tracking is lost*
3. *// Track Lost: Query KeyFrame Database for keyframe candidates for relocalisation*
4. vector<KeyFrame\*> vpCandidateKFs = mpKeyFrameDB->DetectRelocalizationCandidates(&mCurrentFrame);

* 5

其次，寻找有足够多的特征点匹配的关键帧；最后，利用RANSAC迭代，然后使用PnP算法求解位姿。这一部分也在**Tracking::Relocalization()** 里。

## 四、局部地图跟踪

  通过之前的计算，已经得到一个对位姿的初始估计，我们就能透过投影，从已经生成的地图点中找到更多的对应关系，来精确结果。函数入口为：

bOK = TrackLocalMap();

为了降低复杂度，这里只是在局部图中做投影。局部地图中与当前帧有相同点的关键帧序列成为K1，在covisibility graph中与K1相邻的称为K2。局部地图有一个参考关键帧Kref∈K1，它与当前帧具有最多共同看到的地图云点。针对K1, K2可见的每个地图云点，通过如下步骤，在当前帧中进行搜索:   
（1）将地图点投影到当前帧上，如果超出图像范围，就将其舍弃；   
（2）计算当前视线方向向量v与地图点云平均视线方向向量n的夹角，舍弃n·v < cos(60°)的点云；   
（3）计算地图点到相机中心的距离d，认为[dmin, dmax]是尺度不变的区域，若d不在这个区域，就将其舍弃；   
（4）计算图像的尺度因子，为d/dmin；   
（5）将地图点的特征描述子D与还未匹配上的ORB特征进行比较，根据前面的尺度因子，找到最佳匹配。   
  这样，相机位姿就能通过匹配所有地图点，最终被优化。

## 五、关键帧的判断标准

  最后一步是确定是否将当前帧定为关键帧，由于在Local Mapping中，会剔除冗余关键帧，所以我们要尽快插入新的关键帧，这样才能更鲁棒。这个部分代码为：

1. // Check if we need to insert a new keyframe
2. if(NeedNewKeyFrame())
3. CreateNewKeyFrame();

* 1
* 2
* 3

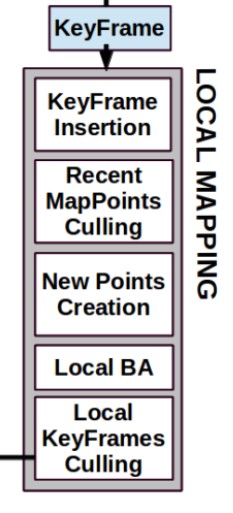
确定关键帧的标准如下：   
（1）在上一个全局重定位后，又过了20帧；   
（2）局部建图闲置，或在上一个关键帧插入后，又过了20帧；   
（3)当前帧跟踪到大于50个点；   
（4）当前帧跟踪到的比参考关键帧少90%。

## 六、代码架构



# 局部建图

在之前的Tracking中，我们得到了新的关键帧Ki。如下图所示，Local Mapping这部分 包括插入关键帧，剔除冗余的地图点和关键帧，还有进行局部集束调整。接下来按照顺序介绍各部分。



这一部分通过之前实例化SLAM系统对象时，实例化了一个LocalMapping的对象，并且开启一个线程，运行LocalMapping::Run()函数。整个代码逻辑如下：



## 一、关键帧插入

  首先将新的关键帧Ki作为新的节点Ki加入Covibility Graph，并且更新与那些能够共享地图点的关键帧节点相连接的边。同时更新关键帧Ki的生长树，并计算表示关键帧的词袋BOW。这一部分的接口是在LocalMapping.cc中的

1. // BoW conversion and insertion in Map
2. ProcessNewKeyFrame();

* 1
* 2

计算词袋，整合地图点到新的关键帧，计算法线和描述子的接口如下：

1. *// Compute Bags of Words structures*
2. mpCurrentKeyFrame->ComputeBoW();
4. *// Associate MapPoints to the new keyframe and update normal and descriptor*
5. const vector<MapPoint\*> vpMapPointMatches = mpCurrentKeyFrame->GetMapPointMatches();

* 1
* 2
* 3
* 4
* 5
* 6

更新Covisibility Graph，将关键帧插入Map中的接口如下：

1. // Update links in the Covisibility Graph
2. mpCurrentKeyFrame->UpdateConnections();
4. // Insert Keyframe in Map
5. mpMap->AddKeyFrame(mpCurrentKeyFrame);

* 1
* 2
* 3
* 4
* 5

## 二、当前地图点剔除

  为了保存地图点，必须在创建该点云的前三帧测试通过约束，才能真正被保存，这样才能保证可跟踪且不容易在三角化时出现较大误差。这部分的接口如下：

1. *// Check recent MapPoints*
2. MapPointCulling();

* 1
* 2

一个点要被加入Map，需要满足下面条件：   
（1）这个点要在可预测到能够观察到该点的关键帧中，有超过25%的关键帧能够跟踪到这个点；   
（2）如果一个地图点被构建，它必须被超过三个关键帧观察到（在代码中，可以发现如果是单摄像头，这个阈值被设置为2）。

  一旦地图点被创建了，就只有在少于3个关键帧能够观察到该点时才会被剔除。而要剔除关键帧，通常是在局部集束调整剔除外点或者在后面剔除关键帧时才会发生。这样就保证了地图点很少存在外点影响效果。

## 三、新的地图点创建

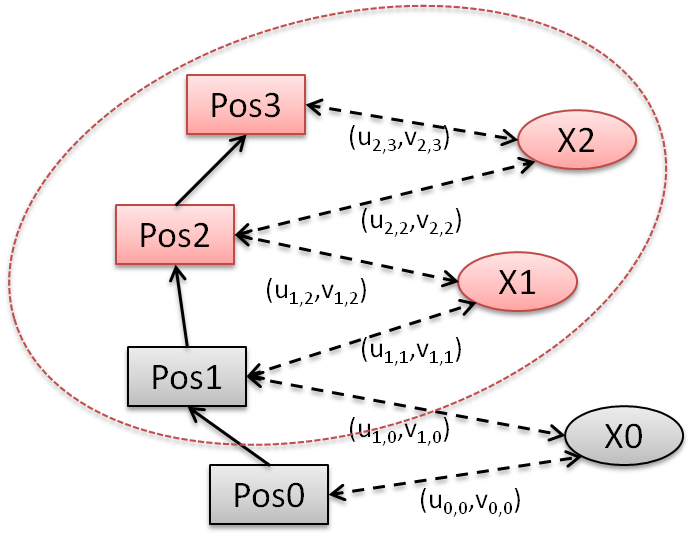
  通过将检测到的ORB特征点，找到Covisibility Graph中与之相连的关键帧Kc，进行特征匹配，然后将匹配到的特征点进行三角化。对于没有匹配上的点，本文又与其他关键帧中未被匹配的特征点进行匹配。匹配方法使用的是之前的方法，并且将不满足对极几何约束的匹配点舍弃。ORB特征点对三角化后，检查正向景深、视差、反投影误差和尺度一致性，这时才得到地图点。一个地图点是通过两个关键帧观察到的，而它也可以投影到与之相连的其他关键帧中，这个时候可以使用Tracking部分的跟踪局部地图来在附近的关键帧中找到匹配。得到更多的地图点。这部分接口为：

1. *// Triangulate new MapPoints*
2. CreateNewMapPoints();
4. if(!CheckNewKeyFrames())
5. {
6. *// Find more matches in neighbor keyframes and fuse point duplications*
7. SearchInNeighbors();
8. }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

## 四、局部集束调整

  局部集束调整（local BA）会将当前处理的关键帧Ki进行优化，优化时如下图所示：现在优化Pos3位置的关键帧。同时参与优化的还有：   
所有在Covibility Graph中与该关键帧相连的关键帧Kc，即下图中的Pos2；   
所以被这些关键帧观察到的地图点，即X1和X2。   
另外还有能观察到地图点的但并未与当前处理的关键帧相连的关键帧，即下图中的Pos1。   
  但要注意的是，诸如Pos1的关键帧，参与优化中的约束，但不作为变量去改变它们的值。优化时得到的外点会在优化的中期或后期被剔除。



这部分的接口如下：

1. *// Local BA*
2. if(mpMap->KeyFramesInMap()>2)
3. Optimizer::LocalBundleAdjustment(mpCurrentKeyFrame,&mbAbortBA, mpMap);

* 1
* 2
* 3

## 五、局部关键帧剔除

  为了控制重建的紧凑度，LocalMapping会去检测冗余的关键帧，然后删除它们。这样的话会有利于控制，随着关键帧数目增长后，集束调整的复杂度。因为除非视角改变了，否则关键帧的数量在相同的环境中不应该无休止地增长。本文将那些有90%的点能够被超过三个关键帧观察到的关键帧认为是冗余关键帧，并将其删除。这个部分的接口如下：

1. *// Check redundant local Keyframes*
2. KeyFrameCulling();

* 1
* 2

最后，在所有步骤结束后，会将关键帧记录到数据库列表中。完成LocalMapping工作，将标志设为SetAcceptKeyFrames(true)，以允许Tracking线程继续得到关键帧。

# 闭环检测

毋庸置疑的是，随着相机的运动，我们计算的相机位姿，三角化得到的点云位置，都是有误差的，即使我们使用局部的或全局的BA去优化，仍然会存在累积误差。而消除误差最有效的办法是发现闭环，并根据闭环对所有结果进行优化。闭环是一个比BA更加强烈、更加准确的约束，所有如果能够检测到闭环，并对其优化，就可以让结果更加准确。   
  整个LoopClosing模块是在线程中完成，并在创建线程时调用LoopClosing::Run()函数让其运行。整个框架如下图所示：   


## 一、闭环条件检测

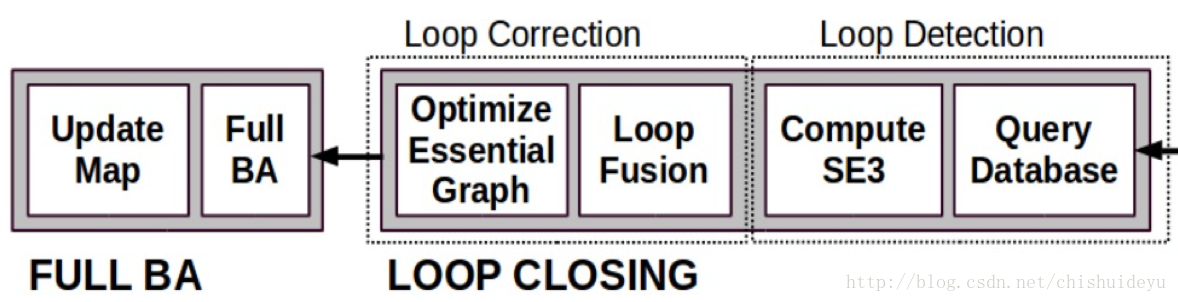
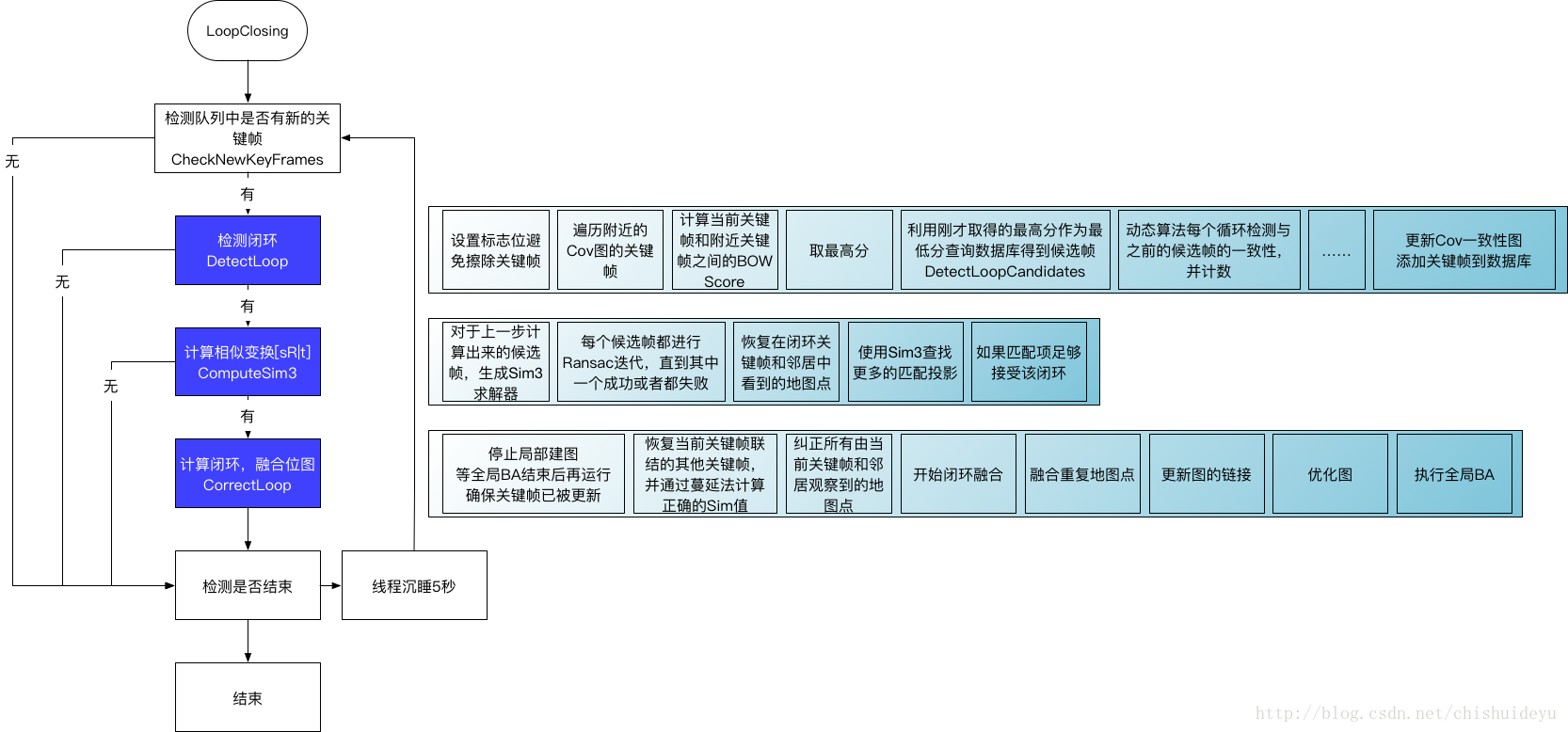
  首先我们计算关键帧Ki和在Covisibility Graph中与其相连的关键帧之间的词袋(BOW)之间的相似度。本文中，作者离线训练了大量的基于ORB描述的词袋，在程序运行时加载进去。这里的词袋作为对该关键帧的描述，将闭环检测转变为一个类似于模式识别的问题。当相机再次来到之前到过的场景时，就会因为看到相同的景物，而得到类似的词袋描述，从而检测到闭环。这里的接口是

1. *// Detect loop candidates and check covisibility consistency*
2. if(DetectLoop())

# ORBSlam2中的闭环检测和后端优化LoopClosing

LoopClosing在现在的Slam系统中是非常重要的一个部分，VO总是会有累计误差，而LoopClosing通过检测是否曾经来过此处，进行后端优化，可以将这个累计误差缩小到一个可接受的范围内。从而使得Slam系统应对大范围场景时，拥有更高的鲁棒性和可用性。

ORBSlam2中的LoopClosing闭环检测线程主要进行闭环检测，并在检测到闭环的时候计算Sim3变换，进行后端优化。

## 检测闭环 DetectLoop

首先我们会检测当前关键帧在Covisibility图中的附近关键帧，并会依次计算当前关键帧和每一个附近关键帧的BoW分值，通过我们所得到分数的最低分，到数据库中查询，查找出所有大于该最低分的关键帧作为候选帧，用以检测闭环。

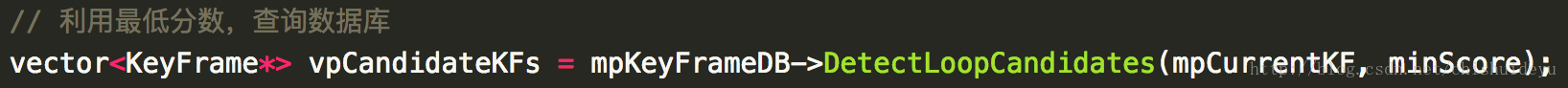
### 1. 如果地图中的关键帧数小于10，那么不进行闭环检测

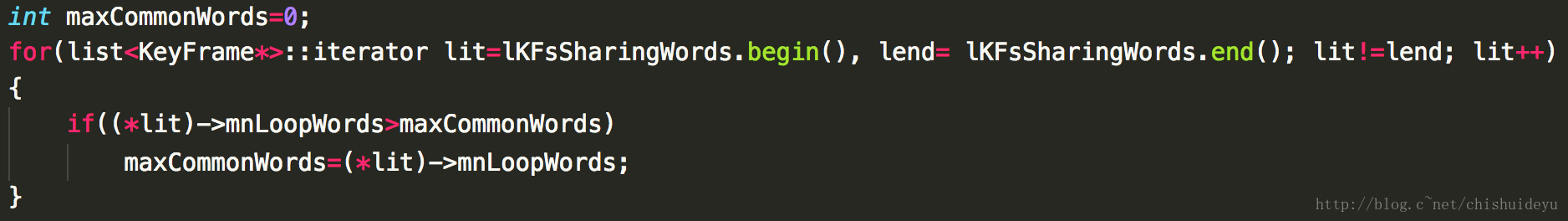
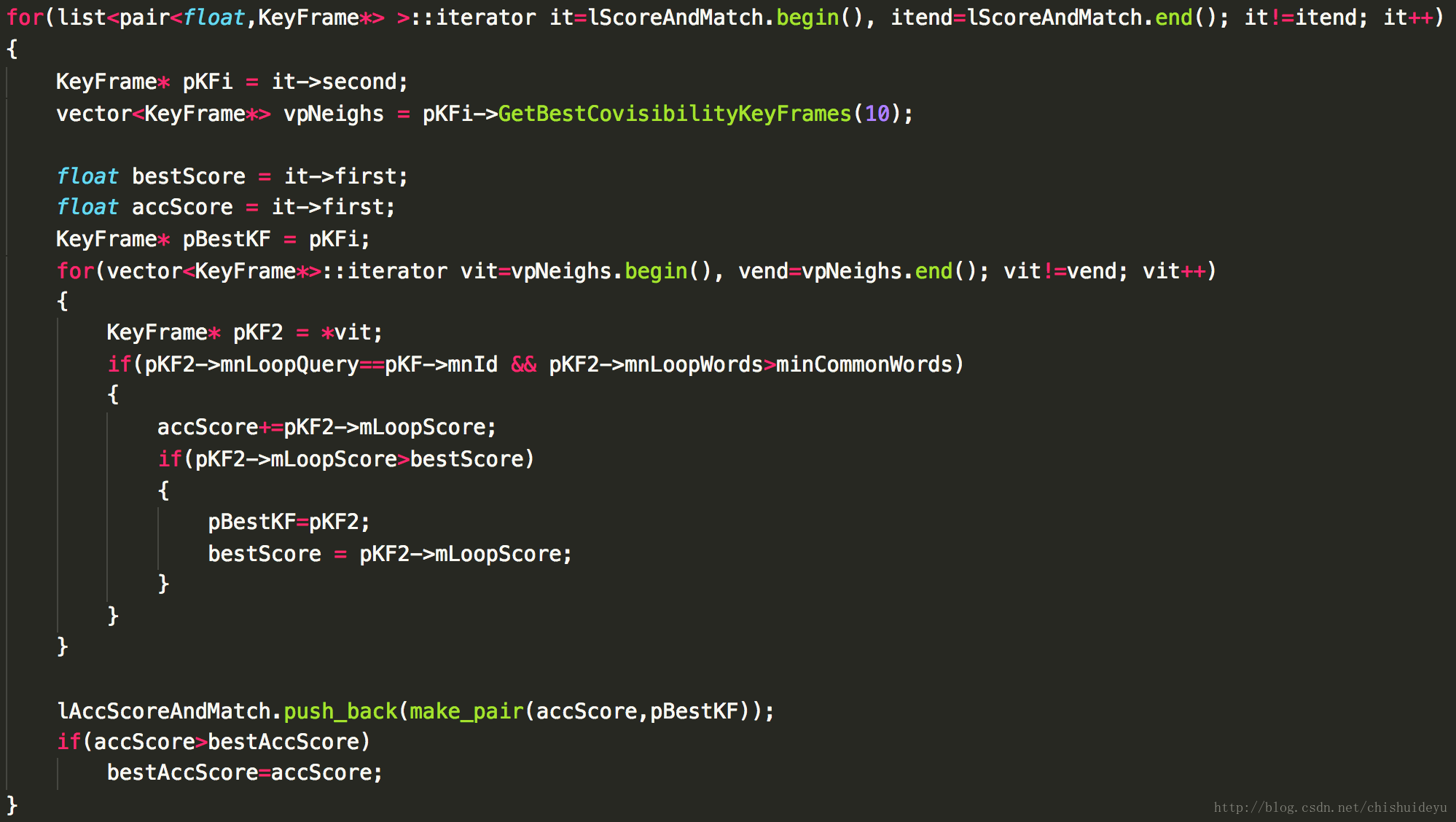


### 2. 获取共视关键帧，并计算他们和当前关键帧之间的BoW分数，求得最低分

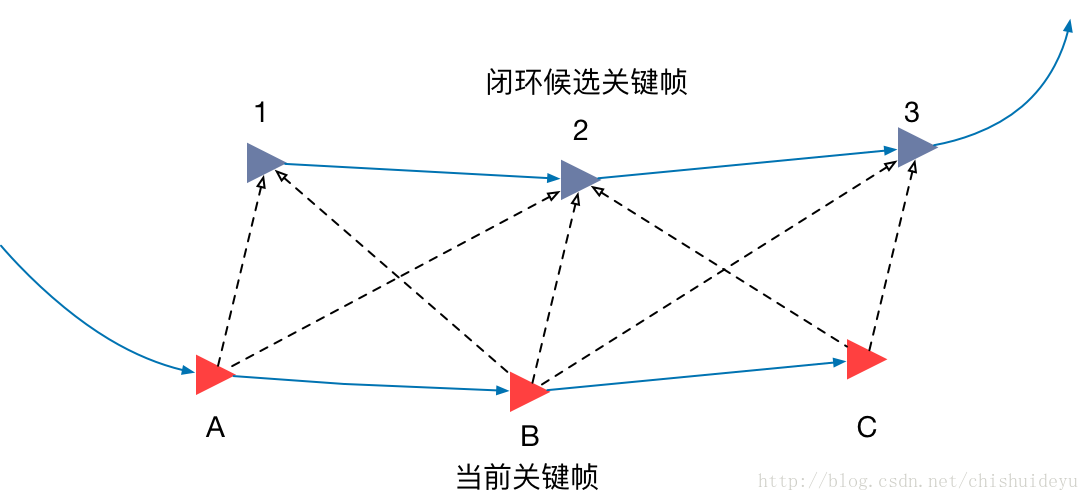


### 3. 通过上一步计算出的最低分数到数据库中查找出候选关键帧，这一步相当于是找到了曾经到过此处的关键帧们

  
这一步是非常重要的，相当于是在为闭环检测做前期的预处理

1. 首先，得到与**当前帧的链接的**关键帧。   
   set<KeyFrame\*> spConnectedKeyFrames = pKF->GetConnectedKeyFrames();
2. 然后，在地图中搜索**与当前关键帧共享一个BOW word的**关键帧，并排除上一步搜集到的附近关键帧，得到候选帧，这些候选帧基本上都是曾经来到此处看到的。   
   
3. 统计这些帧中，**与当前关键帧的Bow共有Word最多**的单词数maxCommonWords。   
   
4. 计算**最低共有单词数阈值**minCommonWords = maxCommonWords\*0.8f，并搜寻候选帧中，共有单词数大于minCommonWords的关键帧，并计算它与当前帧的score分值。   
   
5. 将这些**候选帧进行聚类**，相连的候选帧聚为同一类，并计算每一组的累计得分，得到最高累计分的组，并得分数最高组的最高分的关键帧。   
   这样的话，会把一些拥有很高分数的独立出来的关键帧给去掉，因为他并没有跟其他关键帧相连，没有连续性，所以这一步就是**去除这些得分很高的错误关键帧**。   
   
6. 得到**最后一个阈值** minScoreToRetain = 0.75f\*bestAccScore，再通过这个最低阈值，计算出这些候选帧中比这个分值高的关键帧，并保存起来返回。   
   

### 4. 对候选关键帧集进行连续性检测(有的文章也翻译为一致性检测)

这一步相当于是检测上一步得到的关键帧集是否是真的可以使用的。   
**（由于这段网络上很少有介绍，所以这部分我是按照我自己的理解写的，如果有错误，希望有人能够指出）**   


#### 闭环连续性是什么

每次我们在上一步的数据库查询操作里找到了候选关键帧之后，基本上找到的候选关键帧就是我们所要找的闭环关键帧，但是为了防止错误进行闭环检测，我们非常有必要再进行一次连续性检测，连续性检测的意思就是，是否我们在三个当前的关键帧内都同时发现了某一个闭环候选帧的话，那么就表明当前的SLAM系统已经闭环。

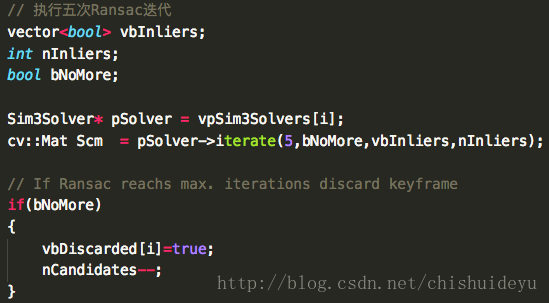
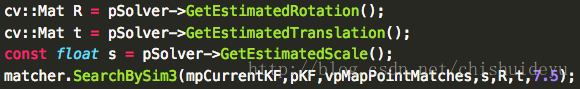
#### 详细解释

比方说，在上图中，通过数据库查询，我们可以在A点得到闭环候选关键帧有两个(1,2)。在下一次进入DetectLoop函数的时候，我们当前拿到的关键帧是B，那么在B点我们可以得到的闭环候选关键帧是(1,2,3)，[以此类推](https://www.baidu.com/s?wd=%E4%BB%A5%E6%AD%A4%E7%B1%BB%E6%8E%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)，在再下一次进入DetectLoop函数的时候，也就是在C点的时候，我们这时对比到的闭环候选关键帧是(2,3)，所以2这个闭环候选关键帧被检测到了三次。在LoopClosing中，mnCovisibilityConsistencyTh = 3 一致性共视阈值被设为3，并且如果一旦有一个闭环候选关键帧被检测到3次，系统就认为检测到闭环。

在每一次进行完闭环候选连续性检测之后，该线程都会保存在这一关键帧下的计数情况，保存的变量也就是mvConsistentGroups，以供给下一轮循环使用。

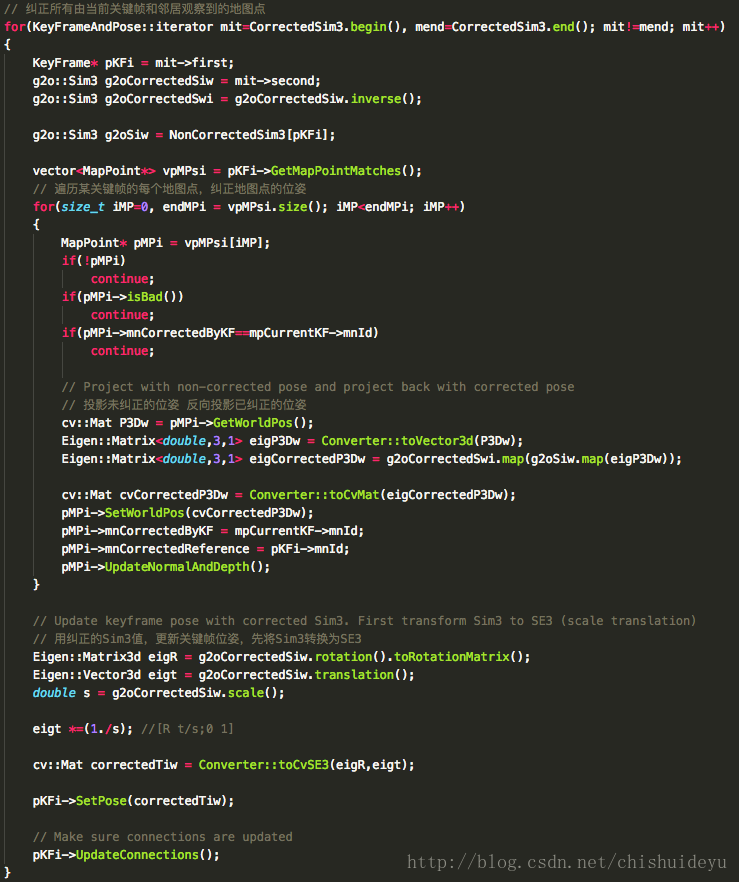
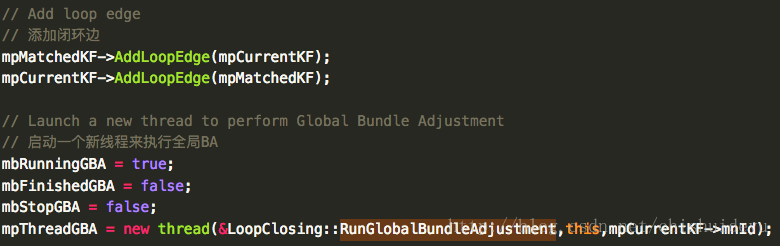
## 计算Sim3 ComputeSim3

既然上一步我们已经检测到了闭环，那么我们现在就需要开始进行后端优化了。该函数主要工作就是在当前关键帧和闭环帧之间找到更多的对应点，并通过这些对应点计算当前关键帧和闭环帧之间的Sim3变换，求解出Rt和s。在这一过程中，共进行了三次对应点的查找工作。

1. 对每一个闭环帧，通过BoW的matcher方法进行第一次匹配，匹配闭环帧和当前关键帧之间的匹配关系，如果对应关系少于20个，则丢弃，否则构造一个Sim3求解器并保存起来。**这一步主要是对效果较好的闭环帧构建Sim3求解器**   
   
2. 对上一步得到的每一个满足条件的闭环帧，通过RANSAC迭代，求解Sim3。   
   对于这里的Sim3的Ransac迭代，我在网上也没有找到很多的解释，我们都知道Ransac是根据一组包含异常数据的样本数据集，计算出数据的数学模型参数的方法，在这里使用Ransac的方法，可以大大提高对杂点干扰的鲁棒性。   
   
3. 通过返回的Sim3进行第二次匹配。   
   刚才得到了Sim3，所以现在要利用Sim3再去进行匹配点的查找，本次查找的匹配点数量，会在原来的基础上有所增加。   
   
4. 使用非线性最小二乘法优化Sim3.   
   在拿到了第二次匹配的结果以后，要通过这些匹配点，再去优化Sim3的值，从而再精细化Rt和s。   
   const int nInliers = Optimizer::OptimizeSim3(mpCurrentKF, pKF, vpMapPointMatches, gScm, 10, mbFixScale);
5. 恢复闭环关键帧和其邻居关键帧的MapPoint地图点   
   最后一步求解匹配点的时候，将所指的闭环帧和与其链接的关键帧所看到的所有的MapPoint都恢复出来。通过这个方法，可以尽可能得到我们当前关键帧所能看到的所有的地图点，为下一步做投影匹配，得到更多的匹配点做准备。   
   
6. 使用投影得到更多的匹配点，如果匹配点数量充足，则接受该闭环。   
   最后我们通过投影匹配得到了尽可能多的匹配点，通过匹配点的数量判断是否接受闭环。   
   

## 纠正闭环后端优化 CorrectLoop

在上一步求得了Sim3和对应点之后，就纠正了当前帧的位姿，但是我们的误差不仅仅在当前帧，此前的每一帧都有累计误差需要消除，所以这个函数CorrectLoop就是用来消除这个累计误差，进行整体的调节。

1. 如果有全局BA运算在运行的话，终止之前的BA运算。   
   
2. 使用传播法计算每一个关键帧正确的Sim3变换值
   * 得到当前关键帧的附近关键帧的Sim3位姿并用纠正的Sim3位姿与其相乘，保存结果到CorrectedSim3变量中。   
     
   * 使用反向投影的方法，将当前关键帧和邻居观察到的地图点得到三维场景下的位姿，并更新关键帧的位姿   
     
   * 将当前关键帧的地图点进行融合，其实融合就是判断如果是同一个点的话，那么将当前的地图点强制换成原本的地图点。   
     
   * 使用已纠正的位姿，将在循环关键帧附近观察到的地图点投影到当前的关键帧和邻居，融合重复点。   
     
   * 更新链接，检测新连接   
     
3. 优化图   
   Optimizer::OptimizeEssentialGraph(mpMap, mpMatchedKF, mpCurrentKF, NonCorrectedSim3, CorrectedSim3, LoopConnections, mbFixScale);   
   使用非线性最小二乘图优化的方法来优化EssentialGraph。
4. 全局BA优化   
   

2、单目、近处双目和远处双目特征点

近双目特征点与远双目特征点：

近双目特征点的定义是：匹配的深度值小于40倍双目或者RGB-D的基线，否则的话，是远特征点。

近特征点：能够由一帧图像直接三角测量化，得到的深度是精确的估计，并且能够提供尺度，平移和旋转的信息。

远特征点：仅仅能够提供精确的旋转信息，但是很少的尺度和平移信息。当提供多视图的时候，我们才能三角化那些远的点。

用单目或者双目进行BA

BA主要用在以下几个地方：在跟踪线程中使用BA优化相机位姿（纯运动BA）；优化关键帧的本地窗口和局部建图线程中的点（局部BA）；在闭环检测后优化所有关键帧和点（全局BA）



