LocalMapping线程

一、概述

在System.h中，定义了LocalMapping线程的类指针

```
// Local Mapper. It manages the local map and performs local bundle adjustment.
LocalMapping* mpLocalMapper;
```

开启方式：打开线程

```
// System threads: Local Mapping, Loop Closing, Viewer.
// The Tracking thread "lives" in the main execution thread that creates the
System object.
std::thread* mptLocalMapping;//进步建图线程
std::thread* mptLoopClosing;
std::thread* mptViewer;
```

System构造函数中，可以找到开启线程的语句

```
mptLocalMapping = new thread(&ORB_SLAM2::LocalMapping::Run,mpLocalMapper);
//线程主要函数为Run(),类指针为mpLocalMapper
```

二、类成员及方法

1、重要成员

```
Map* mpMap;//对应地图

LoopClosing* mpLoopCloser;//对应其他线程指针
Tracking* mpTracker;

// Tracking线程向LocalMapping中插入关键帧是先插入到该队列中
std::list<KeyFrame*> mlNewKeyFrames; ///< 等待处理的关键帧列表

KeyFrame* mpCurrentKeyFrame;//局部地图当前关键帧

std::list<MapPoint*> mlpRecentAddedMapPoints;

std::mutex mMutexNewKFs;

bool mbAbortBA;

bool mbStopped;
bool mbStopRequested;
bool mbNotStop;
std::mutex mMutexStop;

bool mbAcceptKeyFrames;
std::mutex mMutexAccept;
```

2、重要成员函数

三、Run()函数

1、整体流程

Run()函数主体为一个大无限循环，说明在局部建图线程中，一直循环处理局部地图(受到互斥锁影响)。内容与论文VI节高度对应。

A. Keyframe Insertion，在局部地图中插入关键帧，更新其共视关系，并且计算此关键帧的词袋向量。

B. Recent Map Points Culling，剔除不合格的MapPoints

C. New Map Point Creation，通过三角化共视图中相连的关键帧得到一些新的地图点。

D. Local Bundle Adjustment，针对当前关键帧Ki，以及共视图中与Ki连接的其他关键帧Kc，以及这些关键帧观测到的地图点进行优化。

E. Local Keyframe Culling，对多余的关键帧进行检测并删除。

2、函数解析

```
void LocalMapping::Run()
{

    mbFinished = false;

    while(1)
    {
        // Tracking will see that Local Mapping is busy
        // 告诉Tracking，LocalMapping正处于繁忙状态，
        // LocalMapping线程处理的关键帧都是Tracking线程发过的
        // 在LocalMapping线程还没有处理完关键帧之前Tracking线程最好不要发送太快
        SetAcceptKeyFrames(false);

        // Check if there are keyframes in the queue
        // 等待处理的关键帧列表不为空
        if(CheckNewKeyFrames())
        {
            // BoW conversion and insertion in Map
            // VI-A keyframe insertion
            // 计算关键帧特征点的BoW映射，将关键帧插入地图
            ProcessNewKeyFrame();

            // Check recent MapPoints
            // VI-B recent map points culling
            // 剔除ProcessNewKeyFrame函数中引入的不合格MapPoints
            MapPointCulling();

            // Triangulate new MapPoints
            // VI-C new map points creation
            // 相机运动过程中与相邻关键帧通过三角化恢复出一些MapPoints
            CreateNewMapPoints();

            // 已经处理完队列中的最后的一个关键帧
            if(!CheckNewKeyFrames())
```

```
            {
                // Find more matches in neighbor keyframes and fuse point
duplications
                // 检查并融合当前关键帧与相邻帧（两级相邻）重复的MapPoints
                SearchInNeighbors();
            }

            mbAbortBA = false;

            // 已经处理完队列中的最后的一个关键帧，并且闭环检测没有请求停止LocalMapping
            if(!CheckNewKeyFrames() && !stopRequested())
            {
                // VI-D Local BA
                if(mpMap->KeyFramesInMap()>2)

 Optimizer::LocalBundleAdjustment(mpCurrentKeyFrame,&mbAbortBA, mpMap);

                // Check redundant local Keyframes
                // VI-E local keyframes culling
                // 检测并剔除当前帧相邻的关键帧中冗余的关键帧
                // 剔除的标准是：该关键帧的90%的MapPoints可以被其它关键帧观测到
                // trick!
                // Tracking中先把关键帧交给LocalMapping线程
                // 并且在Tracking中InsertKeyFrame函数的条件比较松，交给LocalMapping线程
的关键帧会比较密
                // 在这里再删除冗余的关键帧
                KeyFrameCulling();
            }

            // 将当前帧加入到闭环检测队列中
            mpLoopCloser->InsertKeyFrame(mpCurrentKeyFrame);
        }
        else if(Stop())
        {
            // Safe area to stop
            while(isStopped() && !CheckFinish())
            {
                // usleep(3000);
                std::this_thread::sleep_for(std::chrono::milliseconds(3));
            }
            if(CheckFinish())
                break;
        }

        ResetIfRequested();

        // Tracking will see that Local Mapping is not busy
        SetAcceptKeyFrames(true);

        if(CheckFinish())
            break;

        //usleep(3000);
        std::this_thread::sleep_for(std::chrono::milliseconds(3));

    }

    SetFinish();
```

```
}
```

四、其他函数

1、SetAcceptKeyFrames(bool flag)

设置是否接受关键帧。**这个函数在大循环开始和结束时调用，局部地图更新时，我们设定为不接收新关键帧，认定局部建图线程忙**。结束时，我们再设定局部建图线程不忙。

```
void LocalMapping::SetAcceptKeyFrames(bool flag)
{
    unique_lock<mutex> lock(mMutexAccept);
    mbAcceptKeyFrames=flag;
}
```

2、CheckNewKeyFrames()

mlNewKeyFrames为插入的关键帧队列。

```
/**
 * @brief 查看列表中是否有等待被插入的关键帧
 * @return 如果存在，返回true
 */
bool LocalMapping::CheckNewKeyFrames()
{
    unique_lock<mutex> lock(mMutexNewKFs);
    return(!mlNewKeyFrames.empty());
}
```

3、ProcessNewKeyFrame()

i. 概述

```
/**
 * @brief 处理列表中的关键帧
 *
 * - 计算Bow，加速三角化新的MapPoints
 * - 关联当前关键帧至MapPoints，并更新MapPoints的平均观测方向和观测距离范围
 * - 插入关键帧，更新Covisibility图和Essential图
 * @see VI-A keyframe insertion
 */
void LocalMapping::ProcessNewKeyFrame();
```

ii. 步骤一：从缓冲队列中取出一个关键帧

```
// 步骤1：从缓冲队列中取出一帧关键帧
// Tracking线程向LocalMapping中插入关键帧存在该队列中

unique_lock<mutex> lock(mMutexNewKFs);
// 从列表中获得一个等待被插入的关键帧，将其设置为当前关键帧
mpCurrentKeyFrame = mlNewKeyFrames.front();
//弹出第一个
mlNewKeyFrames.pop_front();
```

iii. 步骤二：计算该关键帧的特征点Bow向量

```
// Compute Bags of Words structures
// 步骤2：计算该关键帧特征点的Bow映射关系
mpCurrentKeyFrame->ComputeBoW();
```

iv. 步骤三：

这一步中，更新了关键帧对应地图点的各种数据(观测关系、最佳描述子等)。

```
// Associate MapPoints to the new keyframe and update normal and descriptor
// 步骤3：跟踪局部地图过程中新匹配上的MapPoints和当前关键帧绑定
// 在TrackLocalMap函数中将局部地图中的MapPoints与当前帧进行了匹配，
// 但没有对这些匹配上的MapPoints与当前帧进行关联
const vector<MapPoint*> vpMapPointMatches = mpCurrentKeyFrame-
>GetMapPointMatches();

for(size_t i=0; i<vpMapPointMatches.size(); i++)
{
    MapPoint* pMP = vpMapPointMatches[i];
    if(pMP)
    {
        if(!pMP->isBad())
        {
            // 非当前帧生成的MapPoints
            // 为当前帧在tracking过程跟踪到的MapPoints更新属性
            if(!pMP->IsInKeyFrame(mpCurrentKeyFrame))
            {
                // 添加观测
                pMP->AddObservation(mpCurrentKeyFrame, i);
                // 获得该点的平均观测方向和观测距离范围
                pMP->UpdateNormalAndDepth();
                // 加入关键帧后，更新3d点的最佳描述子
                pMP->ComputeDistinctiveDescriptors();
            }
            else // this can only happen for new stereo points inserted by the
Tracking
            {
                // 当前帧生成的MapPoints
                // 将双目或RGBD跟踪过程中新插入的MapPoints放入
mlpRecentAddedMapPoints，等待检查
                // CreateNewMapPoints函数中通过三角化也会生成MapPoints
                // 这些MapPoints都会经过MapPointCulling函数的检验
                mlpRecentAddedMapPoints.push_back(pMP);
            }
        }
```

```
    }
}
```

步骤四五：

```
// Update links in the Covisibility Graph
// 步骤4：更新关键帧间的连接关系，Covisibility图和Essential图(tree)
mpCurrentKeyFrame->UpdateConnections();

// Insert Keyframe in Map
// 步骤5：将该关键帧插入到地图中
mpMap->AddKeyFrame(mpCurrentKeyFrame);
```

4、MapPointCulling()函数

```
/**
 * @brief 剔除ProcessNewKeyFrame和CreateNewMapPoints函数中引入的质量不好的MapPoints
 * @see VI-B recent map points culling
 */
void LocalMapping::MapPointCulling()
{
    // Check Recent Added MapPoints
    // 最近新添加的地图点的迭代器
    list<MapPoint*>::iterator lit = mlpRecentAddedMapPoints.begin();
    //当前关键帧ID
    const unsigned long int nCurrentKFid = mpCurrentKeyFrame->mnId;

    int nThObs;
    if(mbMonocular)
        nThObs = 2;
    else
        nThObs = 3;
    const int cnThObs = nThObs;

    // 遍历等待检查的MapPoints
    while(lit!=mlpRecentAddedMapPoints.end())
    {
        MapPoint* pMP = *lit;
        if(pMP->isBad())
        {
            // 步骤1：已经是坏点的MapPoints直接从检查链表中删除
            lit = mlpRecentAddedMapPoints.erase(lit);
        }
        else if(pMP->GetFoundRatio()<0.25f)
        {
            // 步骤2：将不满足VI-B条件的MapPoint剔除
            // VI-B 条件1:
            // 跟踪到该MapPoint的Frame数相比预计可观测到该MapPoint的Frame数的比例需大于
25%
            // IncreaseFound / IncreaseVisible < 25%，注意不一定是关键帧。
            pMP->SetBadFlag();
            lit = mlpRecentAddedMapPoints.erase(lit);
        }
        else if(((int)nCurrentKFid-(int)pMP->mnFirstKFid)>=2 && pMP-
>Observations()<=cnThObs)
        {
```

```
            // 步骤3：将不满足VI-B条件的MapPoint剔除
            // VI-B 条件2：从该点建立开始，到现在已经过了不小于2个关键帧
            // 但是观测到该点的关键帧数却不超过cnThObs帧，那么该点检验不合格
            pMP->SetBadFlag();
            lit = mlpRecentAddedMapPoints.erase(lit);
        }
        else if(((int)nCurrentKFid-(int)pMP->mnFirstKFid)>=3)
            // 步骤4：从建立该点开始，已经过了3个关键帧而没有被剔除，则认为是质量高的点
            // 因此没有SetBadFlag()，仅从队列中删除，放弃继续对该MapPoint的检测
            lit = mlpRecentAddedMapPoints.erase(lit);
        else
            lit++;
    }
}
```

5、CreateNewMapPoints()函数

i. 整体流程(待续)

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

步骤一：

6、SearchInNeighbors()函数

```
/**
 * 检查并融合当前关键帧与相邻帧（两级相邻）重复的MapPoints
 */
void LocalMapping::SearchInNeighbors()
```

我们主要看 如何对重复的点进行融合。

步骤一：

```
// Retrieve neighbor keyframes
// 步骤1：获得当前关键帧在covisibility图中权重排名前nn的邻接关键帧
// 找到当前帧一级相邻与二级相邻关键帧
int nn = 10;
if(mbMonocular)
    nn=20;
//vpNeighKFs 权重较高的邻接关键帧
const vector<KeyFrame*> vpNeighKFs = mpCurrentKeyFrame-
>GetBestCovisibilityKeyFrames(nn);

vector<KeyFrame*> vpTargetKFs;
//对vpNeighKFs进行遍历
for(vector<KeyFrame*>::const_iterator vit=vpNeighKFs.begin(),
vend=vpNeighKFs.end(); vit!=vend; vit++)
{
    KeyFrame* pKFi = *vit;
    //若关键帧有效 且 该关键帧未加入融合vector(vpTargetKFs)
    if(pKFi->isBad() || pKFi->mnFuseTargetForKF == mpCurrentKeyFrame->mnId)
        continue;
    vpTargetKFs.push_back(pKFi);// 加入一级相邻帧
    pKFi->mnFuseTargetForKF = mpCurrentKeyFrame->mnId;// 并标记已经加入
```

```
    // Extend to some second neighbors
    //我们找到vpNeighKFs(与当前关键帧相连且权重较高的关键帧)的二级相邻帧五个
    //通俗来说，关键帧的邻居的邻居
    const vector<KeyFrame*> vpSecondNeighKFs = pKFi-
>GetBestCovisibilityKeyFrames(5);
    //遍历二级相邻帧
    for(vector<KeyFrame*>::const_iterator vit2=vpSecondNeighKFs.begin(),
vend2=vpSecondNeighKFs.end(); vit2!=vend2; vit2++)
    {
        KeyFrame* pKFi2 = *vit2;
        //若该帧坏 或 已经加入融合队列 或邻居的邻居是自身(当前关键帧)
        //则跳过此次循环
        if(pKFi2->isBad() || pKFi2->mnFuseTargetForKF==mpCurrentKeyFrame->mnId ||
pKFi2->mnId==mpCurrentKeyFrame->mnId)
            continue;
        vpTargetKFs.push_back(pKFi2);// 存入二级相邻帧
    }
}
//我们把一级和二级关键帧都存入 vpTargetKFs
```

步骤二：

将当前关键帧的地图点 与 一级二级相邻帧的地图点 进行融合

```
// Search matches by projection from current KF in target KFs
ORBmatcher matcher;

// 步骤2：将当前帧的MapPoints分别与一级二级相邻帧(的MapPoints)进行融合
vector<MapPoint*> vpMapPointMatches = mpCurrentKeyFrame->GetMapPointMatches();
for(vector<KeyFrame*>::iterator vit=vpTargetKFs.begin(), vend=vpTargetKFs.end();
vit!=vend; vit++)
{
    KeyFrame* pKFi = *vit;

    // 投影当前帧的MapPoints到相邻关键帧pKFi中，并判断是否有重复的MapPoints
    // 1.如果MapPoint能匹配关键帧的特征点，并且该点有对应的MapPoint，那么将两个MapPoint合并
（选择观测数多的）
    // 2.如果MapPoint能匹配关键帧的特征点，并且该点没有对应的MapPoint，那么为该点添加
MapPoint
    matcher.Fuse(pKFi,vpMapPointMatches);
}

// Search matches by projection from target KFs in current KF
// 用于存储一级邻接和二级邻接关键帧所有MapPoints的集合
vector<MapPoint*> vpFuseCandidates;
vpFuseCandidates.reserve(vpTargetKFs.size()*vpMapPointMatches.size());
```

步骤三：

```
// 步骤3：将一级二级相邻帧的MapPoints分别与当前帧（的MapPoints）进行融合
// 遍历每一个一级邻接和二级邻接关键帧
for(vector<KeyFrame*>::iterator vitKF=vpTargetKFs.begin(),
vendKF=vpTargetKFs.end(); vitKF!=vendKF; vitKF++)
{
    KeyFrame* pKFi = *vitKF;
```

```
    vector<MapPoint*> vpMapPointsKFi = pKFi->GetMapPointMatches();

    // 遍历当前一级邻接和二级邻接关键帧中所有的MapPoints
    for(vector<MapPoint*>::iterator vitMP=vpMapPointsKFi.begin(),
vendMP=vpMapPointsKFi.end(); vitMP!=vendMP; vitMP++)
    {
        MapPoint* pMP = *vitMP;
        if(!pMP)
            continue;

        // 判断MapPoints是否为坏点，或者是否已经加进集合vpFuseCandidates
        if(pMP->isBad() || pMP->mnFuseCandidateForKF == mpCurrentKeyFrame->mnId)
            continue;

        // 加入集合，并标记已经加入
        pMP->mnFuseCandidateForKF = mpCurrentKeyFrame->mnId;
        vpFuseCandidates.push_back(pMP);
    }
}

matcher.Fuse(mpCurrentKeyFrame,vpFuseCandidates);
```

步骤四五：

```
// Update points
// 步骤4：更新当前帧MapPoints的描述子，深度，观测主方向等属性
vpMapPointMatches = mpCurrentKeyFrame->GetMapPointMatches();
for(size_t i=0, iend=vpMapPointMatches.size(); i<iend; i++)
{
    MapPoint* pMP=vpMapPointMatches[i];
    if(pMP)
    {
        if(!pMP->isBad())
        {
            // 在所有找到pMP的关键帧中，获得最佳的描述子
            pMP->ComputeDistinctiveDescriptors();

            // 更新平均观测方向和观测距离
            pMP->UpdateNormalAndDepth();
        }
    }
}

// Update connections in covisibility graph

// 步骤5：更新当前帧的MapPoints后更新与其它帧的连接关系
// 更新covisibility图
mpCurrentKeyFrame->UpdateConnections();
```

*详细讲解Fuse()函数：

概述

```
/**
 * @brief 将MapPoints投影到关键帧pKF中，并判断是否有重复的MapPoints
 * 1.如果MapPoint能匹配关键帧的特征点，并且该点有对应的MapPoint，那么将两个MapPoint合并（选择观测数多的）
 * 2.如果MapPoint能匹配关键帧的特征点，并且该点没有对应的MapPoint，那么为该点添加MapPoint
 * @param  pKF          相邻关键帧
 * @param  vpMapPoints 当前关键帧的MapPoints
 * @param  th           搜索半径的因子
 * @return              重复MapPoints的数量
 */
int ORBmatcher::Fuse(KeyFrame *pKF, const vector<MapPoint *> &vpMapPoints, const float th);
```

7、LocalBundleAdjustment()函数

概述：

即优化关键帧位姿，又优化地图点位置。

```
/**
 * @brief Local Bundle Adjustment
 *
 * 1. Vertex:
 *     - g2o::VertexSE3Expmap()，LocalKeyFrames，即当前关键帧的位姿、与当前关键帧相连的关键帧的位姿
 *     - g2o::VertexSE3Expmap()，FixedCameras，即能观测到LocalMapPoints的关键帧（并且不属于LocalKeyFrames）的位姿，在优化中这些关键帧的位姿不变
 *     - g2o::VertexSBAPointXYZ()，LocalMapPoints，即LocalKeyFrames能观测到的所有MapPoints的位置
 * 2. Edge:
 *     - g2o::EdgeSE3ProjectXYZ()，BaseBinaryEdge
 *         + Vertex：关键帧的Tcw，MapPoint的Pw
 *         + measurement：MapPoint在关键帧中的二维位置(u,v)
 *         + InfoMatrix：invSigma2(与特征点所在的尺度有关)
 *     - g2o::EdgeStereoSE3ProjectXYZ()，BaseBinaryEdge
 *         + Vertex：关键帧的Tcw，MapPoint的Pw
 *         + measurement：MapPoint在关键帧中的二维位置(ul,v,ur)
 *         + InfoMatrix：invSigma2(与特征点所在的尺度有关)
 *
 * @param pKF        KeyFrame
 * @param pbStopFlag 是否停止优化的标志
 * @param pMap       在优化后，更新状态时需要用到Map的互斥量mMutexMapUpdate
 */
void Optimizer::LocalBundleAdjustment(KeyFrame *pKF, bool* pbStopFlag, Map* pMap)
```

步骤一：

从当前关键帧开始进行广度搜索(共视图)

```
// Local KeyFrames: First Breadth Search from Current Keyframe
list<KeyFrame*> lLocalKeyFrames;

// 步骤1：将当前关键帧加入lLocalKeyFrames
lLocalKeyFrames.push_back(pKF);
pKF->mnBALocalForKF = pKF->mnId;
```

步骤二：

```
// 步骤2：找到关键帧连接的关键帧（一级相连），加入lLocalKeyFrames中
const vector<KeyFrame*> vNeighKFs = pKF->GetVectorCovisibleKeyFrames();
for(int i=0, iend=vNeighKFs.size(); i<iend; i++)
{
    KeyFrame* pKFi = vNeighKFs[i];
    pKFi->mnBALocalForKF = pKF->mnId;
    if(!pKFi->isBad())
        lLocalKeyFrames.push_back(pKFi);
}
```

步骤三：

```
// Local MapPoints seen in Local KeyFrames
// 步骤3：遍历lLocalKeyFrames中关键帧，将它们观测的MapPoints加入到lLocalMapPoints
list<MapPoint*> lLocalMapPoints;
for(list<KeyFrame*>::iterator lit=lLocalKeyFrames.begin() ,
lend=lLocalKeyFrames.end(); lit!=lend; lit++)
{
    vector<MapPoint*> vpMPs = (*lit)->GetMapPointMatches();
    for(vector<MapPoint*>::iterator vit=vpMPs.begin(), vend=vpMPs.end();
vit!=vend; vit++)
    {
        MapPoint* pMP = *vit;
        if(pMP)
        {
            if(!pMP->isBad())
                if(pMP->mnBALocalForKF!=pKF->mnId)
                {
                    lLocalMapPoints.push_back(pMP);
                    pMP->mnBALocalForKF=pKF->mnId;// 防止重复添加
                }
        }
    }
}
```

步骤四：

```
// Fixed Keyframes. Keyframes that see Local MapPoints but that are not Local
Keyframes
// 步骤4：得到能被局部MapPoints观测到，但不属于局部关键帧的关键帧，这些关键帧在局部BA优化时不
优化
list<KeyFrame*> lFixedCameras;
for(list<MapPoint*>::iterator lit=lLocalMapPoints.begin(),
lend=lLocalMapPoints.end(); lit!=lend; lit++)
{
    map<KeyFrame*,size_t> observations = (*lit)->GetObservations();
```

```
    for(map<KeyFrame*,size_t>::iterator mit=observations.begin(),
mend=observations.end(); mit!=mend; mit++)
    {
        KeyFrame* pKFi = mit->first;

        // pKFi->mnBALocalForKF!=pKF->mnId表示局部关键帧,
        // 其它的关键帧虽然能观测到，但不属于局部关键帧
        if(pKFi->mnBALocalForKF!=pKF->mnId && pKFi->mnBAFixedForKF!=pKF->mnId)
        {
            pKFi->mnBAFixedForKF=pKF->mnId;// 防止重复添加
            if(!pKFi->isBad())
                lFixedCameras.push_back(pKFi);
        }
    }
}
```

步骤五：

```
// Setup optimizer
// 步骤5: 构造g2o优化器
g2o::SparseOptimizer optimizer;
g2o::BlockSolver_6_3::LinearSolverType * linearSolver;

linearSolver = new g2o::LinearSolverEigen<g2o::BlockSolver_6_3::PoseMatrixType>
();

g2o::BlockSolver_6_3 * solver_ptr = new g2o::BlockSolver_6_3(linearSolver);

g2o::OptimizationAlgorithmLevenberg* solver = new
g2o::OptimizationAlgorithmLevenberg(solver_ptr);
optimizer.setAlgorithm(solver);

if(pbStopFlag)
    optimizer.setForceStopFlag(pbStopFlag);

unsigned long maxKFid = 0;
```

步骤六：

```
// Set Local KeyFrame vertices
// 步骤6: 添加顶点: Pose of Local KeyFrame
//遍历所有局部关键帧，这些关键帧位姿需要调整
for(list<KeyFrame*>::iterator lit=lLocalKeyFrames.begin(),
lend=lLocalKeyFrames.end(); lit!=lend; lit++)
{
    KeyFrame* pKFi = *lit;
    //新建节点
    g2o::VertexSE3Expmap * vSE3 = new g2o::VertexSE3Expmap();
    //初始值(就是当前关键帧位姿，还没被优化)
    vSE3->setEstimate(Converter::toSE3Quat(pKFi->GetPose()));
    vSE3->setId(pKFi->mnId);
    //这里，若输入参数为true，则代表添加的当前节点值不变
    //若这是第一帧，那么pKFi->mnId==0为true，则第一帧位姿固定
    vSE3->setFixed(pKFi->mnId==0);//第一帧位置固定
    optimizer.addVertex(vSE3);
    if(pKFi->mnId>maxKFid)
```

```
        maxKFid=pKFi->mnId;//找最大KF的id
}
```

步骤七：

```
// Set Fixed KeyFrame vertices
// 步骤7：添加顶点：Pose of Fixed KeyFrame，注意这里调用了vSE3->setFixed(true)。
for(list<KeyFrame*>::iterator lit=lFixedCameras.begin(),
lend=lFixedCameras.end(); lit!=lend; lit++)
{
    //遍历所有FixedCameras，(固定的关键帧)
    KeyFrame* pKFi = *lit;
    g2o::VertexSE3Expmap * vSE3 = new g2o::VertexSE3Expmap();
    vSE3->setEstimate(Converter::toSE3Quat(pKFi->GetPose()));
    vSE3->setId(pKFi->mnId);
    //所有的位置全固定
    vSE3->setFixed(true);
    optimizer.addVertex(vSE3);
    if(pKFi->mnId>maxKFid)
        maxKFid=pKFi->mnId;
}
```

步骤八：

```
// Set MapPoint vertices
// 步骤8：添加3D顶点节点

//步骤8.1 保存所有顶点以及关联的边，为了后续再次优化(剔除outlier后再优化)
const int nExpectedSize =
(lLocalKeyFrames.size()+lFixedCameras.size())*lLocalMapPoints.size();

vector<g2o::EdgeSE3ProjectXYZ*> vpEdgesMono;
vpEdgesMono.reserve(nExpectedSize);

vector<KeyFrame*> vpEdgeKFMono;
vpEdgeKFMono.reserve(nExpectedSize);

vector<MapPoint*> vpMapPointEdgeMono;
vpMapPointEdgeMono.reserve(nExpectedSize);

vector<g2o::EdgeStereoSE3ProjectXYZ*> vpEdgesStereo;
vpEdgesStereo.reserve(nExpectedSize);

vector<KeyFrame*> vpEdgeKFStereo;
vpEdgeKFStereo.reserve(nExpectedSize);

vector<MapPoint*> vpMapPointEdgeStereo;
vpMapPointEdgeStereo.reserve(nExpectedSize);

const float thHuberMono = sqrt(5.991);
const float thHuberStereo = sqrt(7.815);
//添加MapPoint顶点以及MapPoint-KeyFrame的边
for(list<MapPoint*>::iterator lit=lLocalMapPoints.begin(),
lend=lLocalMapPoints.end(); lit!=lend; lit++)
{
    // 添加顶点节点：MapPoint
```

```
    MapPoint* pMP = *lit;
    g2o::VertexSBAPointXYZ* vPoint = new g2o::VertexSBAPointXYZ();
    vPoint->setEstimate(Converter::toVector3d(pMP->GetWorldPos()));
    int id = pMP->mnId+maxKFid+1;//为了不与KF的节点ID重复
    vPoint->setId(id);
    vPoint->setMarginalized(true);//边缘化
    optimizer.addVertex(vPoint);

    const map<KeyFrame*,size_t> observations = pMP->GetObservations();

    // Set edges
    // 步骤8：对每一对关联的MapPoint和KeyFrame构建边
    //这个子循环是在对MapPoint的大循环中的，目的：添加完一个MapPoint节点后立刻添加与之相连的
边
    for(map<KeyFrame*,size_t>::const_iterator mit=observations.begin(),
mend=observations.end(); mit!=mend; mit++)
    {
        //可以看到这个地图点的KF
        KeyFrame* pKFi = mit->first;
        if(!pKFi->isBad())
        {
            //在之前了解过，Frame和KeyFrame的关键点与地图点(mvKeysUn和mvpMapPoints)是一
一对应的
            //因此，这里取得其地图点对应下标即可得到相应的关键点
            const cv::KeyPoint &kpUn = pKFi->mvKeysUn[mit->second];
            // Monocular observation
            if(pKFi->mvuRight[mit->second]<0)
            {
                //像素坐标赋值
                Eigen::Matrix<double,2,1> obs;
                obs << kpUn.pt.x, kpUn.pt.y;

                g2o::EdgeSE3ProjectXYZ* e = new g2o::EdgeSE3ProjectXYZ();
                //设置二元边的节点:
                //1、节点号为id的MapPoint节点(id=id = pMP->mnId+maxKFid+1)
                //2、节点号为pKFi->mnID的KeyFrame节点
                e->setVertex(0, dynamic_cast<g2o::OptimizableGraph::Vertex*>
(optimizer.vertex(id)));
                e->setVertex(1, dynamic_cast<g2o::OptimizableGraph::Vertex*>
(optimizer.vertex(pKFi->mnId)));
                //测量值为obs(像素坐标)。理解为：相机(传感器)在某位姿下拍摄(观测)某地图点得
到其在照片中的位置(像素观测值)。
                e->setMeasurement(obs);
                //协方差 信息矩阵
                const float &invSigma2 = pKFi->mvInvLevelSigma2[kpUn.octave];
                e->setInformation(Eigen::Matrix2d::Identity()*invSigma2);

                g2o::RobustKernelHuber* rk = new g2o::RobustKernelHuber;
                e->setRobustKernel(rk);
                rk->setDelta(thHuberMono);

                e->fx = pKFi->fx;
                e->fy = pKFi->fy;
                e->cx = pKFi->cx;
                e->cy = pKFi->cy;
                //添加边，保存边及其两边的节点。
                optimizer.addEdge(e);
                vpEdgesMono.push_back(e);
```

```
                vpEdgeKFMono.push_back(pKFi);
                vpMapPointEdgeMono.push_back(pMP);
            }
            else // Stereo observation
            {
                Eigen::Matrix<double,3,1> obs;
                const float kp_ur = pKFi->mvuRight[mit->second];
                obs << kpUn.pt.x, kpUn.pt.y, kp_ur;

                g2o::EdgeStereoSE3ProjectXYZ* e = new
g2o::EdgeStereoSE3ProjectXYZ();

                e->setVertex(0, dynamic_cast<g2o::OptimizableGraph::Vertex*>
(optimizer.vertex(id)));
                e->setVertex(1, dynamic_cast<g2o::OptimizableGraph::Vertex*>
(optimizer.vertex(pKFi->mnId)));
                e->setMeasurement(obs);
                const float &invSigma2 = pKFi->mvInvLevelSigma2[kpUn.octave];
                Eigen::Matrix3d Info = Eigen::Matrix3d::Identity()*invSigma2;
                e->setInformation(Info);

                g2o::RobustKernelHuber* rk = new g2o::RobustKernelHuber;
                e->setRobustKernel(rk);
                rk->setDelta(thHuberStereo);

                e->fx = pKFi->fx;
                e->fy = pKFi->fy;
                e->cx = pKFi->cx;
                e->cy = pKFi->cy;
                e->bf = pKFi->mbf;

                optimizer.addEdge(e);
                vpEdgesStereo.push_back(e);
                vpEdgeKFStereo.push_back(pKFi);
                vpMapPointEdgeStereo.push_back(pMP);
            }
        }
    }
}
```

步骤九：

```
// 步骤9：开始优化
optimizer.initializeOptimization();
optimizer.optimize(5);//迭代次数为5

bool bDoMore= true;

if(pbStopFlag)
    if(*pbStopFlag)
        bDoMore = false;
```

步骤十：

在LocalMapping线程没有停止局部BA时，检测outlier，并设置下次不优化

步骤十一：

```
//要清除的节点(地图点以及关键帧)
vector<pair<KeyFrame*,MapPoint*> > vToErase;
vToErase.reserve(vpEdgesMono.size()+vpEdgesStereo.size());
// Check inlier observations
// 步骤11：在优化后重新计算误差，剔除连接误差比较大的关键帧和MapPoint
for(size_t i=0, iend=vpEdgesMono.size(); i<iend;i++)
{
    g2o::EdgeSE3ProjectXYZ* e = vpEdgesMono[i];
    MapPoint* pMP = vpMapPointEdgeMono[i];

    if(pMP->isBad())
        continue;

    // 基于卡方检验计算出的阈值（假设测量有一个像素的偏差）
    if(e->chi2()>5.991 || !e->isDepthPositive())
    {
        KeyFrame* pKFi = vpEdgeKFMono[i];
        vToErase.push_back(make_pair(pKFi,pMP));
    }
}
```

步骤十二：

删除关键帧对应的地图点信息 以及 地图点对应的关键帧信息。

```
// Get Map Mutex
// 删除MP及KF的时候，要保证地图未更新
unique_lock<mutex> lock(pMap->mMutexMapUpdate);

// 连接偏差比较大，在关键帧中剔除对该MapPoint的观测
// 连接偏差比较大，在MapPoint中剔除对该关键帧的观测
if(!vToErase.empty())
{
    for(size_t i=0;i<vToErase.size();i++)
    {
        KeyFrame* pKFi = vToErase[i].first;
        MapPoint* pMPi = vToErase[i].second;
        //删除关键帧对应的地图点信息
        pKFi->EraseMapPointMatch(pMPi);
        //删除地图点对应的关键帧信息
        pMPi->EraseObservation(pKFi);
    }
}
```

步骤十三：

```
// Recover optimized data
// 步骤13：优化后更新关键帧位姿以及MapPoints的位置、平均观测方向等属性

//Keyframes
```

```
for(list<KeyFrame*>::iterator lit=lLocalKeyFrames.begin(),
lend=lLocalKeyFrames.end(); lit!=lend; lit++)
{
    KeyFrame* pKF = *lit;
    g2o::VertexSE3Expmap* vSE3 = static_cast<g2o::VertexSE3Expmap*>
(optimizer.vertex(pKF->mnId));
    //这个位姿节点优化后的结果
    g2o::SE3Quat SE3quat = vSE3->estimate();
    //更新
    pKF->SetPose(Converter::toCvMat(SE3quat));
}

//Points
for(list<MapPoint*>::iterator lit=lLocalMapPoints.begin(),
lend=lLocalMapPoints.end(); lit!=lend; lit++)
{
    MapPoint* pMP = *lit;
    g2o::VertexSBAPointXYZ* vPoint = static_cast<g2o::VertexSBAPointXYZ*>
(optimizer.vertex(pMP->mnId+maxKFid+1));
    //更新
    pMP->SetWorldPos(Converter::toCvMat(vPoint->estimate()));
    pMP->UpdateNormalAndDepth();
}
```