

ORB-SLAM

一、System.h

整体流程图：

1、概况讲解

跟踪函数

有三个：Stereo、RGBD、Monocular。其返回值为相机位姿，若跟踪失败则为空(empty)。

```
// Process the given stereo frame. Images must be synchronized and rectified.
// Input images: RGB (CV_8UC3) or grayscale (CV_8U). RGB is converted to
// grayscale.
// Returns the camera pose (empty if tracking fails).
cv::Mat TrackStereo(const cv::Mat &imLeft, const cv::Mat &imRight, const double
&timestamp);

// Process the given rgbd frame. Depthmap must be registered to the RGB frame.
// Input image: RGB (CV_8UC3) or grayscale (CV_8U). RGB is converted to
// grayscale.
// Input depthmap: Float (CV_32F).
// Returns the camera pose (empty if tracking fails).
cv::Mat TrackRGBD(const cv::Mat &im, const cv::Mat &depthmap, const double
&timestamp);

// Process the given monocular frame
// Input images: RGB (CV_8UC3) or grayscale (CV_8U). RGB is converted to
// grayscale.
// Returns the camera pose (empty if tracking fails).
cv::Mat TrackMonocular(const cv::Mat &im, const double &timestamp);
```

私有变量(SLAM系统中需要用到的部分)

1. ORB词典：用于位置识别和特征点匹配

```
// ORB vocabulary used for place recognition and feature matching.
ORBVocabulary* mpVocabulary;
```

2. 关键帧数据库：用于重定位以及闭环检测

```
KeyFrameDatabase* mpKeyFrameDatabase;
```

3. 地图：存储所有地图点以及关键帧

```
Map* mpMap;
```

4. 跟踪线程：接到图片后计算相机位姿，并且负责插入关键帧、地图点。当跟踪失败时负责重定位方式跟踪估计位姿。

```
Tracking* mpTracker;
```

5. 局部地图：负责局部建图和局部地图BA

```
LocalMapping* mpLocalMapper;
```

6. 闭环检测：对每个新关键帧都搜索闭环，若有闭环，则启动位姿图优化和一个完整全局BA

```
LoopClosing* mpLoopCloser;
```

其余部分变量不太重要，后续遇到问题再讲。

构造函数

初始化一些变量，后续讲解。

```
// Initialize the SLAM system. It launches the Local Mapping, Loop Closing and
Viewer threads.
System(const string &strVocFile, const string &strSettingsFile, const eSensor
sensor, const bool bUseViewer = true);
```

我们现在看一下构造函数：

首先：选择我们想要的传感器(单目、RGBD、双目)、其次，读取配置文件以及ORB词典(ORB Vocabulary)。

词典初始化

```
mpVocabulary = new ORBVocabulary();
bool bVocLoad = mpVocabulary->loadFromTextFile(strVocFile);
```

其中调用了DBow2库函数。判定若词典读取失败则退出程序。

初始化一系列SLAM模块

```
//Create KeyFrame Database
mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary);

//Create the Map
mpMap = new Map();

//Create Drawers. These are used by the Viewer
mpFrameDrawer = new FrameDrawer(mpMap);
mpMapDrawer = new MapDrawer(mpMap, strSettingsFile);

//Initialize the Tracking thread
//(it will live in the main thread of execution, the one that called this
constructor)
mpTracker = new Tracking(this, mpVocabulary, mpFrameDrawer, mpMapDrawer,
                        mpMap, mpKeyFrameDatabase, strSettingsFile, mSensor);

//Initialize the Local Mapping thread and launch
mpLocalMapper = new LocalMapping(mpMap, mSensor==MONOCULAR);
```

```

mptLocalMapping = new thread(&ORB_SLAM2::LocalMapping::Run, mpLocalMapper);

//Initialize the Loop Closing thread and launch
mpLoopCloser = new LoopClosing(mpMap, mpKeyFrameDatabase, mpVocabulary,
mSensor!=MONOCULAR);
mptLoopClosing = new thread(&ORB_SLAM2::LoopClosing::Run, mpLoopCloser);

```

这里，我们后续必须详细研究跟踪、局部地图以及闭环检测线程。

之后，我们还要设置三个线程之间的指针，用途目前未知。

2、跟踪线程

我们在ROS模块的ros_mono.cc中可以看到：回调函数中每次执行的是函数TrackMonocular

```

mpSLAM->TrackMonocular(cv_ptr->image, cv_ptr->header.stamp.toSec());

```

因此，我们要研究这个函数(跟踪系列函数)。这个函数中，先包含了模式转换部分(目前不看)，之后就**是GrabImageMonocular函数**，这个函数中包含了跟踪线程。

//而跟踪线程结束后，

二、跟踪线程Tracking.h

1、概述

主要函数：GrabImageXX(以单目举例GrabImageMonocular)，返回值为当前相机位姿。

```

cv::Mat GrabImageStereo(const cv::Mat &imRectLeft, const cv::Mat &imRectRight,
const double &timestamp);
cv::Mat GrabImageRGBD(const cv::Mat &imRGB, const cv::Mat &imD, const double
&timestamp);
cv::Mat GrabImageMonocular(const cv::Mat &im, const double &timestamp);

```

跟踪线程状态：

```

enum eTrackingState{
    SYSTEM_NOT_READY=-1,
    NO_IMAGES_YET=0,
    NOT_INITIALIZED=1,
    OK=2,
    LOST=3
};

```

The diagram illustrates the Track system architecture and its logic flow, divided into two main parts: System and Track 整体流程 (Track Overall Flow).

System

The System part shows the data flow from the front-end to the back-end:

- Tracker (WEB)**: 返回前端数据, 输入ing everything...
- GrpblImageRABD**: 返回与输入与 TrackerRABD类似
- 构造Frame对象**: Constructing a Frame object.

Track 整体流程

The Track Overall Flow is divided into five main steps:

- 1. 初始化** (Initialization):
 - Track函数** (Track function) leads to **1. 初始化**.
 - 初始化** sets **mState == Lost** (状态异常 - State abnormal).
 - 初始化** leads to **2. 初始跟踪 (未优化)** (Initial tracking (unoptimized)).
- 2. 初始跟踪 (未优化)** (Initial tracking (unoptimized)):
 - Initial tracking involves **直接重定位 Relocalization** (Direct repositioning).
 - Initial tracking leads to **3. 利用局部地图优化当前估计** (Utilizing local map to optimize current estimate).
- 3. 利用局部地图优化当前估计** (Utilizing local map to optimize current estimate):
 - Local map optimization involves **将最新KF作为参考KF** (Using the latest KF as a reference KF).
 - Local map optimization leads to **4. 是否添加KF** (Whether to add KF).
- 4. 是否添加KF** (Whether to add KF):
 - Decision logic: **若跟踪成功, 则更新Tracking下的临时模型** (If tracking is successful, update the temporary model under Tracking) OR **若跟踪失败, 则更新Tracking下的临时模型** (If tracking fails, update the temporary model under Tracking).
 - Decision logic leads to **5. 更新数据** (Update data).
- 5. 更新数据** (Update data):
 - Update data involves **删除临时生成的地图点** (Delete temporary generated map points).
 - Update data involves **清除临时生成的地图点** (Clear temporary generated map points).
 - Update data involves **NeedNewKeyFrame** (Need new key frame).
 - Update data involves **删除跟踪中认为的outliers地图点** (Delete outliers map points considered in tracking).
 - Update data involves **再更新参考KF** (Update reference KF again).
 - Update data involves **构造Frame对象** (Construct Frame object).
 - Update data involves **构造Frame对象** (Construct Frame object).
 - Update data involves **构造Frame对象** (Construct Frame object).

Additional Notes:

- mState == OK** (状态正常 - State normal) is a condition for the initial tracking step.
- 若跟踪成功, 则更新Tracking下的临时模型** (If tracking is successful, update the temporary model under Tracking) is a condition for the decision logic.
- 若跟踪失败, 则更新Tracking下的临时模型** (If tracking fails, update the temporary model under Tracking) is a condition for the decision logic.
- 若跟踪成功, 则更新Tracking下的临时模型** (If tracking is successful, update the temporary model under Tracking) is a condition for the decision logic.
- 若跟踪失败, 则更新Tracking下的临时模型** (If tracking fails, update the temporary model under Tracking) is a condition for the decision logic.
- 若跟踪成功, 则更新Tracking下的临时模型** (If tracking is successful, update the temporary model under Tracking) is a condition for the decision logic.
- 若跟踪失败, 则更新Tracking下的临时模型** (If tracking fails, update the temporary model under Tracking) is a condition for the decision logic.

主要成员	用处
Frame mCurrentFrame	当前帧
cv::Mat mImGray	当前帧的灰度图

流程：获取当前帧，转化为灰度图后调用Track()函数。

```

if(mState==OK)
{
    // Local Mapping might have changed some MapPoints tracked in last frame
    // 检查并更新上一帧被替换的MapPoints
    // 更新Fuse函数和SearchAndFuse函数替换的MapPoints
    CheckReplacedInLastFrame();

    // 步骤2.1: 跟踪上一帧或者参考帧或者重定位

    // 运动模型是空的或刚完成重定位
    // mCurrentFrame.mnId<mnLastRelocFrameId+2这个判断不应该有
    // 应该只要mVelocity不为空, 就优先选择TrackWithMotionModel
    // mnLastRelocFrameId上一次重定位的那一帧
    if(mVelocity.empty() || mCurrentFrame.mnId<mnLastRelocFrameId+2)
    {
        // 将上一帧的位姿作为当前帧的初始位姿
        // 通过Bow的方式在参考帧中找当前帧特征点的匹配点
        // 优化每个特征点都对应3D点重投影误差即可得到位姿
    }
}

```

```

        bOK = TrackReferenceKeyFrame();
    }
    else
    {
        // 根据恒速模型设定当前帧的初始位姿
        // 通过投影的方式在参考帧中找当前帧特征点的匹配点
        // 优化每个特征点所对应3D点的投影误差即可得到位姿
        bOK = TrackWithMotionModel();
        if(!bOK)
            // TrackReferenceKeyFrame是跟踪参考帧，不能根据固定运动速度模型预测当前帧的位
            姿态，通过bow加速匹配 (SearchByBow)
            // 最后通过优化得到优化后的位姿
            bOK = TrackReferenceKeyFrame();
    }
}
else
{
    // BOW搜索，PnP求解位姿
    bOK = Relocalization();
}

```

```

mCurrentFrame.mpReferenceKF = mpReferenceKF;

// If we have an initial estimation of the camera pose and matching. Track the
local map.
if(!mbOnlyTracking)
{
    if(bOK)
        bOK = TrackLocalMap();
}

```

若跟踪成功，我们检查是否需要插入关键帧。首先更新mVelocity：若上一帧位姿非空，按照如下方式计算模型速度；若上一帧位姿为空，则速度也为空。

```

cv::Mat LastTwc = cv::Mat::eye(4,4,CV_32F);
mLastFrame.GetRotationInverse().copyTo(LastTwc.rowRange(0,3).colRange(0,3));
mLastFrame.GetCameraCenter().copyTo(LastTwc.rowRange(0,3).col(3));
mVelocity = mCurrentFrame.mTcw*LastTwc;
//这里的速度为两帧位姿间的李代数增量

```

$$p_{c1} = T_{cw1}p$$

$$p_{c2} = T_{cw2}p$$

$$p_{c2} = T_{cw2}T_{cw1}^{-1}p_{c1}$$

速度我们可以视为pc1到pc2相机位姿之间的李代数增量。

计算完速度后，检查是否需要关键帧，根据需求添加关键帧。后存储当前帧位姿信息来复原整个相机运动轨迹。

```

// Store frame pose information to retrieve the complete camera trajectory
afterwards.
if(!mCurrentFrame.mTcw.empty())
{
    cv::Mat Tcr = mCurrentFrame.mTcw*mCurrentFrame.mpReferenceKF-
>GetPoseInverse();
}

```

```

        mRelativeFramePoses.push_back(Tcr);
        mlpReferences.push_back(mpReferenceKF);
        mFrameTimes.push_back(mCurrentFrame.mTimeStamp);
        mlbLost.push_back(mState==LOST);
    }
    else
    {
        // This can happen if tracking is lost
        mRelativeFramePoses.push_back(mRelativeFramePoses.back());
        mlpReferences.push_back(mlpReferences.back());
        mFrameTimes.push_back(mFrameTimes.back());
        mlbLost.push_back(mState==LOST);
    }
}

```

I. TrackReferenceKeyFrame()函数

主要功能，通过比对关键帧来跟踪。

```

if(mVelocity.empty() || mCurrentFrame.mnId<mnLastRelocFrameId+2)
{
    // 将上一帧的位姿作为当前帧的初始位姿
    // 通过Bow的方式在参考帧中找当前帧特征点的匹配点
    // 优化每个特征点都对应3D点重投影误差即可得到位姿
    bOK = TrackReferenceKeyFrame();
}

```

*详细实现

步骤1:

```
mCurrentFrame.ComputeBow(); //将当前帧的描述子转化为Bow向量
```

步骤2:

```

// We perform first an ORB matching with the reference keyframe
// If enough matches are found we setup a PnP solver
//vpMapPointMatches初步认为是当前帧(mCurrentFrame)中与关键帧(mpReferenceKF)中地图点匹配的地图点
ORBmatcher matcher(0.7,true);
vector<MapPoint*> vpMapPointMatches;

// 步骤2: 通过特征点的Bow加快当前帧与参考帧之间的特征点匹配
// 特征点的匹配关系由MapPoints进行维护
int nmatches =
matcher.SearchByBow(mpReferenceKF,mCurrentFrame,vpMapPointMatches);
//若匹配点太少，则返回false
if(nmatches<15)
    return false;

```

步骤3:

```
// 步骤3:将上一帧的位姿态作为当前帧位姿的初始值
//mvpMapPoints:与当前帧中的关键点与地图点关联的点(地图点)
//原版注释: MapPoints associated to keyPoints
mCurrentFrame.mvpMapPoints = vpMapPointMatches;
mCurrentFrame.SetPose(mLastFrame.mTcw); // 用上一次的Tcw设置初值, 在PoseOptimization
可以收敛快一些
```

步骤4:

```
// 步骤4:通过优化3D-2D的重投影误差来获得位姿
Optimizer::PoseOptimization(&mCurrentFrame);
```

步骤5:

```
// Discard outliers
// 步骤5: 剔除优化后的outlier匹配点 (MapPoints)
int nmatchesMap = 0;
for(int i =0; i<mCurrentFrame.N; i++)
{
    if(mCurrentFrame.mvpMapPoints[i])
    {
        if(mCurrentFrame.mvbOutlier[i])
        {
            MapPoint* pMP = mCurrentFrame.mvpMapPoints[i];
            mCurrentFrame.mvpMapPoints[i]=static_cast<MapPoint*>(NULL);
            mCurrentFrame.mvbOutlier[i]=false;
            pMP->mbTrackInView = false;
            pMP->mnLastFrameSeen = mCurrentFrame.mnId;
            nmatches--;
        }
        else if(mCurrentFrame.mvpMapPoints[i]->Observations(>0)
            nmatchesMap++;
    }
}
```

*SearchByBow()函数

*PoseOptimization()函数

II. TrackWithMotionModel()函数

若速度不为空或者刚刚进行重定位时, 我们根据匀速模型跟踪

```
else
{
    // 根据恒速模型设定当前帧的初始位姿
    // 通过投影的方式在参考帧中找当前帧特征点的匹配点
    // 优化每个特征点所对应3D点的投影误差即可得到位姿
    bOK = TrackWithMotionModel();
    if(!bOK)
        // TrackReferenceKeyFrame是跟踪参考帧, 不能根据固定运动速度模型预测当前帧的位姿态,
        通过bow加速匹配 (SearchByBow)
        // 最后通过优化得到优化后的位姿
        bOK = TrackReferenceKeyFrame();
}
```

与4中不同的是，匀速模型是先利用速度给出一个当前位姿初始值，从这个初始值中找到周围与参考帧匹配的特征点，进行BA优化(优化投影误差)。

*详细实现

步骤1：首先调用UpdateLastFrame()函数，并根据速度模型，直接估计出当前位姿

```
// Update last frame pose according to its reference keyframe
// Create "visual odometry" points
// 步骤1: 对于双目或rgb-d摄像头，根据深度值为上一关键帧生成新的MapPoints
// （跟踪过程中需要将当前帧与上一帧进行特征点匹配，将上一帧的MapPoints投影到当前帧可以缩小匹配范围）
// 在跟踪过程中，去除outlier的MapPoint，如果不及时增加MapPoint会逐渐减少
// 这个函数的功能就是补充增加RGBD和双目相机上一帧的MapPoints数
UpdateLastFrame();

// 根据Const Velocity Model(认为这两帧之间的相对运动和之前两帧间相对运动相同)估计当前帧的位姿
mCurrentFrame.SetPose(mVelocity*mLastFrame.mTcw);
```

接下来调用fill，将当前帧的地图点设置为NULL

```
fill(mCurrentFrame.mvpMapPoints.begin(),mCurrentFrame.mvpMapPoints.end(),static_cast<MapPoint*>(NULL));
```

步骤2：

```
// 步骤2: 根据匀速模型进行对上一帧的MapPoints进行跟踪
// 根据上一帧特征点对应的3D点投影的位置缩小特征点匹配范围
int nmatches =
matcher.SearchByProjection(mCurrentFrame,mLastFrame,th,mSensor==System::MONOCULAR);
//通过投影关系找到匹配点

// If few matches, uses a wider window search
// 如果跟踪的点少，则扩大搜索半径再来一次
if(nmatches<20)
{
    //fill全赋值为NULL，再尝试一次

    fill(mCurrentFrame.mvpMapPoints.begin(),mCurrentFrame.mvpMapPoints.end(),static_cast<MapPoint*>(NULL));
    nmatches =
matcher.SearchByProjection(mCurrentFrame,mLastFrame,2*th,mSensor==System::MONOCULAR); // 2*th
}
//点太少，则匀速运动模型失效，返回false
if(nmatches<20)
    return false;
```

步骤3：

```
Optimizer::PoseOptimization(&mCurrentFrame); //优化位姿
```

步骤4：


```
// 步骤4: 优化位姿后剔除outlier的mvpMapPoints
int nmatchesMap = 0;
for(int i = 0; i < mCurrentFrame.N; i++)
{
    if(mCurrentFrame.mvpMapPoints[i])
    { //地图点非空
        if(mCurrentFrame.mvbOutlier[i])
        { //地图点为outlier
            MapPoint* pMP = mCurrentFrame.mvpMapPoints[i];
            mCurrentFrame.mvpMapPoints[i] = static_cast<MapPoint*>(NULL);
            mCurrentFrame.mvbOutlier[i] = false;
            pMP->mbTrackInView = false;
            pMP->mnLastFrameSeen = mCurrentFrame.mnId;
            nmatches--;
        }
        else if(mCurrentFrame.mvpMapPoints[i]->Observations() > 0)
            nmatchesMap++;
    }
}
...
return nmatchesMap >= 10;
```

***UpdateLastFrame()函数**

***SearchByProjection()函数**

III. Relocalization()函数

若跟踪失败，则启用：

```
else
{
    // BOW搜索, PnP求解位姿
    bOK = Relocalization();
}
```

***详细讲解**

步骤1:

```
// Compute Bag of Words Vector
// 步骤1: 计算当前帧特征点的Bow映射(向量)
mCurrentFrame.ComputeBow();
```

步骤2:

```
// Relocalization is performed when tracking is lost
// Track Lost: Query KeyFrame Database for keyframe candidates for relocalisation
// 步骤2: 找到与当前帧相似的候选关键帧
vector<KeyFrame*> vpCandidateKFs = mpKeyFrameDB-
>DetectRelocalizationCandidates(&mCurrentFrame);
// 无相似候选关键帧, 重定位失败, 返回false
if(vpCandidateKFs.empty())
    return false;
// nKFs为与当前帧相似的候选关键帧数量
const int nKFs = vpCandidateKFs.size();
```

步骤3:

我们将当前帧与候选关键帧进行ORB特征点匹配，若有足够的特征点，则使用PnP算法重定位当前位姿。(此部分因为涉及到PnP算法以及EPnP算法，待续未完)

```
// We perform first an ORB matching with each candidate
// If enough matches are found we setup a PnP solver
ORBmatcher matcher(0.75,true);

vector<PnP solver*> vpPnP solvers;
vpPnP solvers.resize(nKFs);

vector<vector<MapPoint*> > vvpMapPointMatches;
vvpMapPointMatches.resize(nKFs);

vector<bool> vbDiscarded;
vbDiscarded.resize(nKFs);

int nCandidates=0;

for(int i=0; i<nKFs; i++)
{
    KeyFrame* pKF = vpCandidateKFs[i];
    if(pKF->isBad())
        vbDiscarded[i] = true;
    else
    {
        // 步骤3: 通过Bow进行匹配
        int nmatches =
matcher.SearchByBow(pKF,mCurrentFrame,vvpMapPointMatches[i]);
        if(nmatches<15)
        {
            vbDiscarded[i] = true;
            continue;
        }
        else
        {
            // 初始化PnP solver
            PnP solver* pSolver = new
PnP solver(mCurrentFrame,vvpMapPointMatches[i]);
            pSolver->SetRansacParameters(0.99,10,300,4,0.5,5.991);
            vpPnP solvers[i] = pSolver;
            nCandidates++;
        }
    }
}
```

此部分待续未完

IV. TrackLocalMap()函数

```
// We have an estimation of the camera pose and some map points tracked in the
frame.
// We retrieve the local map and try to find matches to points in the local map.
```

这是在当前帧位姿得到初始值(经过一次仅优化位姿的BA)后进行的操作。跟踪局部地图。在这之前，程序中会将当前帧的参考关键帧进行设置：

```
// 将最新的关键帧作为reference frame
mCurrentFrame.mpReferenceKF = mpReferenceKF;
//论文中也讲到了参考关键帧的定义：
//我们设定一个局部地图，关键帧集合K1为与当前帧共享地图点的关键帧；K2为集合K1的共视图
//邻居(neighbors)。局部地图也有参考关键帧，属于集合K1，其与当前帧共享最多地图点。
//参考关键帧的建立我们后续讲解
```

这时候，我们就进行TrackLocalMap()函数

```
// If we have an initial estimation of the camera pose and matching. Track the
local map.
// 步骤2.2: 在帧间匹配得到初始的姿态后，现在对local map进行跟踪得到更多的匹配，并优化当前位姿
// local map:当前帧、当前帧的MapPoints、当前关键帧与其它关键帧共视关系
// 在步骤2.1中主要是两两跟踪（恒速模型跟踪上一帧、跟踪参考帧），这里搜索局部关键帧后搜集所有局部
MapPoints,
// 然后将局部MapPoints和当前帧进行投影匹配，得到更多匹配的MapPoints后进行Pose优化
if(!mbOnlyTracking)
{
    if(bOK)
        bOK = TrackLocalMap();
}
```

在这之后，当前帧的位姿估计加优化到此结束，接下来判定跟踪是否成功，并且将其可视化，如下：

```
//判断跟踪状态
if(bOK)
    mState = OK;
else
    mState=LOST;

// Update drawer
mpFrameDrawer->Update(this);
mpFrameDrawer->LK = LKimg;
```

*详细讲解

步骤1:

```
// Update Local KeyFrames and Local Points
// 步骤1: 更新局部关键帧mvpLocalKeyFrames和局部地图点mvpLocalMapPoints
UpdateLocalMap();
```

步骤2:

```
// 步骤2: 在局部地图中查找与当前帧匹配的MapPoints
SearchLocalPoints();
```

步骤3:

```
// Optimize Pose
// 在这个函数之前，在Relocalization、TrackReferenceKeyFrame、TrackWithMotionModel中都有位姿优化，
// 步骤3：更新局部所有MapPoints后对位姿再次优化
Optimizer::PoseOptimization(&mCurrentFrame);
//这里调用的PoseOptimization函数，仍然是只对位姿进行优化，只是在局部地图中
//多了更多的与当前帧关联的地图点，使得当前位姿更加准确
mnMatchesInliers = 0;
```

步骤4:

```
// Update MapPoints Statistics
// 步骤3：更新当前帧的MapPoints被观测程度，并统计跟踪局部地图的效果
//更改地图的被观测程度:IncreaseFound()
//统计局部地图跟踪效果:mnMatchesInliers
for(int i=0; i<mCurrentFrame.N; i++)
{
    if(mCurrentFrame.mvpMapPoints[i])
    {
        // 由于当前帧的MapPoints可以被当前帧观测到，其被观测统计量加1
        if(!mCurrentFrame.mvbOutlier[i])
        {
            //IncreaseFound(int i=1)函数
            //增加该地图点被观测到的次数，增加个数为n(默认为1)
            //这里加1是因为此地图点可以被当前帧观测
            mCurrentFrame.mvpMapPoints[i]->IncreaseFound();
            if(!mbOnlyTracking)
            {
                // 该MapPoint被其它关键帧观测到过
                if(mCurrentFrame.mvpMapPoints[i]->Observations(>0))
                    mnMatchesInliers++;
            }
            else
                // 记录当前帧跟踪到的MapPoints，用于统计跟踪效果
                mnMatchesInliers++;
        }
        else if(mSensor==System::STEREO)
            mCurrentFrame.mvpMapPoints[i] = static_cast<MapPoint*>(NULL);
    }
}
```

步骤4:

```
// Decide if the tracking was succesful
// More restrictive if there was a relocalization recently
// 步骤4：决定是否跟踪成功
//若当前帧id 小于 最近一次重定位帧id+最大间隔帧数(含义未懂) 且 有效的地图点小于50个
//可以翻译为 最近才进行重定位
if(mCurrentFrame.mnId<mnLastRelocFrameId+mMaxFrames && mnMatchesInliers<50)
    return false;
//有效地图点小于30个
if(mnMatchesInliers<30)
    return false;
else
    return true;
```

***UpdateLocalMap()函数**

***SearchLocalPoints()函数**

V. 新关键帧的建立

新建关键帧的条件：跟踪成功(**bOK == true**)。

首先，跟踪匀速运动模型，为了下帧关键帧跟踪。

```
if(!mLastFrame.mTcw.empty())//上一帧位姿非空
{
    //这里的速度更新上面讲过了
    // 步骤2.3: 更新恒速运动模型TrackWithMotionModel中的mVelocity
    cv::Mat LastTwc = cv::Mat::eye(4,4,CV_32F);
    mLastFrame.GetRotationInverse().copyTo(LastTwc.rowRange(0,3).colRange(0,3));
    mLastFrame.GetCameraCenter().copyTo(LastTwc.rowRange(0,3).col(3));
    mVelocity = mCurrentFrame.mTcw*LastTwc; // Tc1
}
else
    mVelocity = cv::Mat();
```

$$p_{c1} = T_{cw1}p$$

$$p_{c2} = T_{cw2}p$$

$$p_{c2} = T_{cw2}T_{cw1}^{-1}p_{c1}$$

接下来，清除临时地图点

```
// Clean VO matches
// 步骤2.4: 清除UpdateLastFrame中为当前帧临时添加的MapPoints
for(int i=0; i<mCurrentFrame.N; i++)//mCurrentFrame.N为当前帧的关键点数量
{
    MapPoint* pMP = mCurrentFrame.mvpMapPoints[i];
    if(pMP)
        // 排除UpdateLastFrame函数中为了跟踪增加的MapPoints
        if(pMP->Observations()<1)//可以被多少关键帧观测到
        {
            //不能被关键帧观测到，则删除地图点
            mCurrentFrame.mvbOutlier[i] = false;
            mCurrentFrame.mvpMapPoints[i]=static_cast<MapPoint*>(NULL);
        }
}
}
```

继续清除

```
// Delete temporal MapPoints
// 步骤2.5: 清除临时的MapPoints, 这些MapPoints在TrackWithMotionModel的UpdateLastFrame
函数里生成 (仅双目和rgbd)
// 步骤2.4中只是在当前帧中将这些MapPoints剔除, 这里从MapPoints数据库中删除
// 这里生成的仅仅是为了提高双目或rgbd摄像头的帧间跟踪效果, 用完以后就扔了, 没有添加到地图中
for(list<MapPoint*>::iterator lit = mlpTemporalPoints.begin(), lend =
    mlpTemporalPoints.end(); lit!=lend; lit++)
{
    MapPoint* pMP = *lit;
    delete pMP;
}
// 这里不仅仅是清除mlpTemporalPoints, 通过delete pMP还删除了指针指向的MapPoint
mlpTemporalPoints.clear();
```

检测是否需要关键帧: **NeedNewKeyFrame()**和**CreateNewKeyFrame()**

```
if(needNewKF)
    CreateNewKeyFrame();
```

最终, 删除BA中检测为outlier的3D mapPoints

```
// We allow points with high innovation (considered outliers by the Huber
Function)
// pass to the new keyframe, so that bundle adjustment will finally decide
// if they are outliers or not. We don't want next frame to estimate its position
// with those points so we discard them in the frame.
// 删除那些在bundle adjustment中检测为outlier的3D map点
for(int i=0; i<mCurrentFrame.N;i++)
{
    if(mCurrentFrame.mvpMapPoints[i] && mCurrentFrame.mvbOutlier[i])
        mCurrentFrame.mvpMapPoints[i]=static_cast<MapPoint*>(NULL);
}
```

*详细讲解: **NeedNewKeyFrame()**和**CreateNewKeyFrame()**

***NeedNewKeyFrame()**

VI. 跟踪失败且重定位也失败的情况

这里对应初始化的if(是否初始化), 也就是跟踪的最大框。当 当前帧位姿估计失败, 重定位也失败了, 只能重启(Reset)。

```

if(mState==LOST)
{
    if(mpMap->KeyFramesInMap()<=5)
    {
        //关键帧小于5的情况重启
        //否则还可以通过全局优化挽救
        cout << "Track lost soon after initialisation, resetting..." << endl;
        mpSystem->Reset();
        return;
    }
}
}

```

VII. 更新数据

如同两两帧间视觉里程计，我们更新数据。

```

//若当前帧的参考关键帧为空，则设定参考关键帧
//Tracking类中的mpReferenceKF成员就可以认为mCurrentFrame成员(当前帧)的参考关键帧
//当然，在更新关键帧的部分，是要检查与当前帧最多共视点的的关键帧，将其设定为参考关键帧

//Frame类的参考关键帧论文中已经定义
if(!mCurrentFrame.mpReferenceKF)
    mCurrentFrame.mpReferenceKF = mpReferenceKF;

// 保存上一帧的数据，使用复制构造函数
mLastFrame = Frame(mCurrentFrame);

```

VIII. 轨迹复现

```

// Store frame pose information to retrieve the complete camera trajectory afterwards.
// 步骤3: 记录位姿信息，用于轨迹复现
if(!mCurrentFrame.mTcw.empty())
{
    // 计算相对姿态T_currentFrame_referenceKeyFrame
    cv::Mat Tcr = mCurrentFrame.mTcw*mCurrentFrame.mpReferenceKF-
>GetPoseInverse();
    mRelativeFramePoses.push_back(Tcr);
    mlpReferences.push_back(mpReferenceKF);
    mFrameTimes.push_back(mCurrentFrame.mTimeStamp);
    mlbLost.push_back(mState==LOST);
}
else
{
    // This can happen if tracking is lost
    // 如果跟踪失败，则相对位姿使用上一次值
    mRelativeFramePoses.push_back(mRelativeFramePoses.back());
    mlpReferences.push_back(mlpReferences.back());
    mFrameTimes.push_back(mFrameTimes.back());
    mlbLost.push_back(mState==LOST);
}

```

```
}
```

至此，Track()函数全部结束。

三、Local Mapping线程
