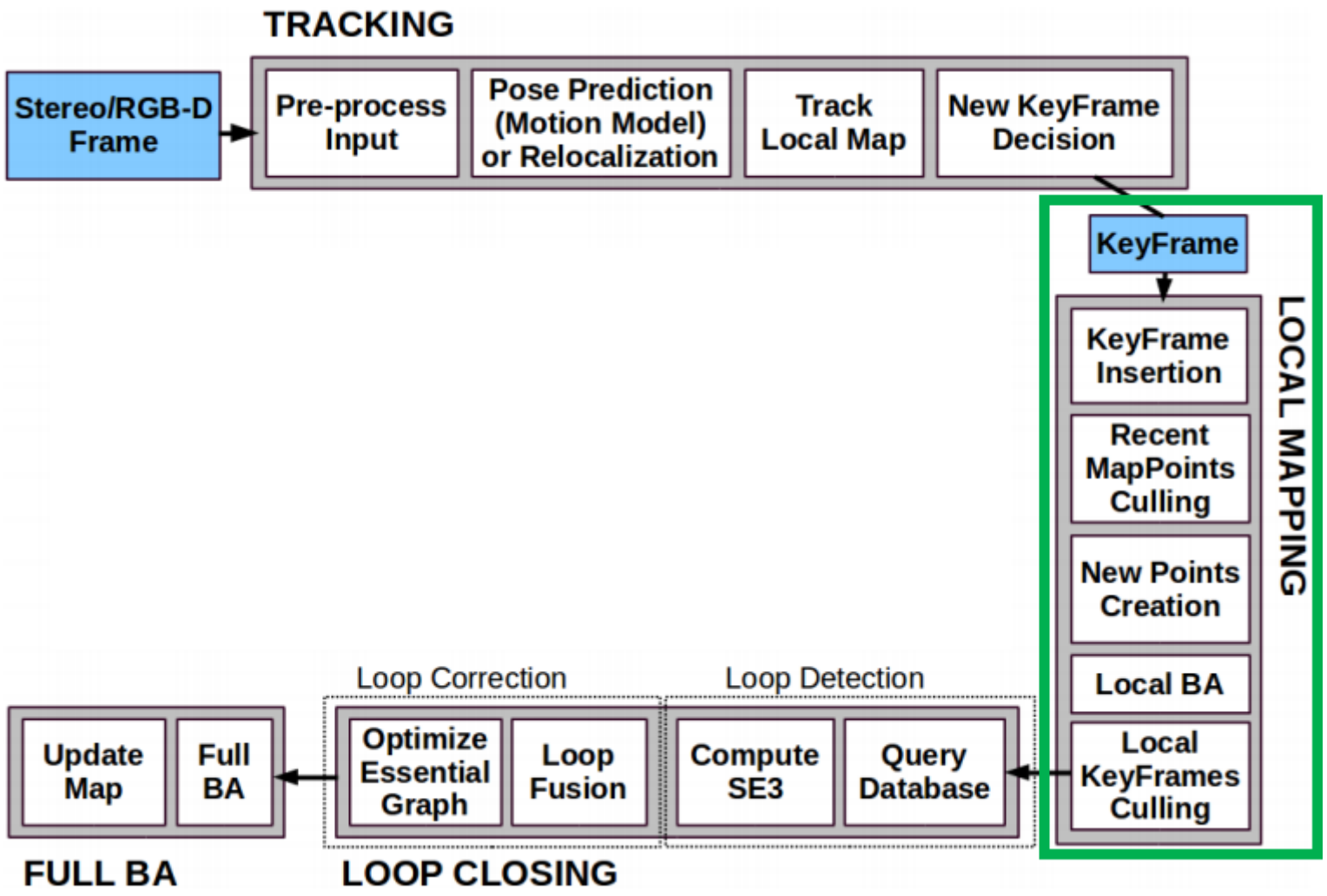
各成员函数/变量

局部建图主函数: `Run()`
处理队列中第一个关键帧: `ProcessNewKeyFrame()`
剔除坏地图点: `MapPointCulling()`
创建新地图点: `CreateNewMapPoints()`
融合当前关键帧和其共视帧的地图点: `SearchInNeighbors()`
局部BA优化: `Optimizer::LocalBundleAdjustment()`
剔除冗余关键帧: `KeyFrameCulling()`



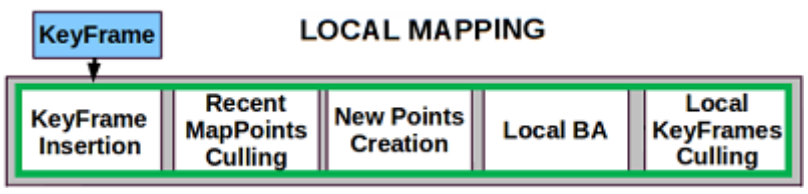
各成员函数/变量

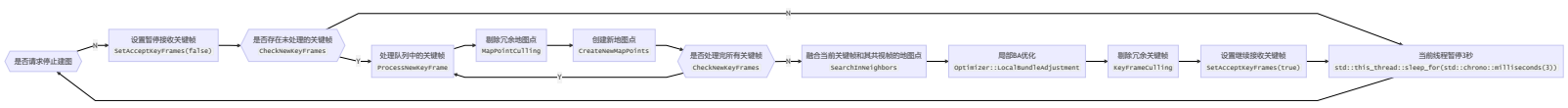
成员函数/变量	访问控制	意义
`std::list<KeyFrame*> mlNewKeyFrames`	`protected`	`Tracking` 线程向 `LocalMapping` 线程插入关键帧的缓冲队列
`void InsertKeyFrame(KeyFrame* pKF)`	`public`	向缓冲队列 `mlNewKeyFrames` 内插入关键帧
`bool CheckNewKeyFrames()`	`protected`	查看缓冲队列 `mlNewKeyFrames` 内是否有待处理的新关键帧
`int KeyframesInQueue()`	`public`	查询缓冲队列 `mlNewKeyFrames` 内关键帧个数
`bool mbAcceptKeyFrames`	`protected`	`LocalMapping` 线程是否愿意接收 `Tracking` 线程传来的新关键帧
`bool AcceptKeyFrames()`	`public`	`mbAcceptKeyFrames` 的get方法
`void SetAcceptKeyFrames(bool flag)`	`public`	`mbAcceptKeyFrames` 的set方法

`Tracking` 线程创建的所有关键帧都被插入到 `LocalMapping` 线程的缓冲队列 `mlNewKeyFrames` 中.

成员函数 `mbAcceptKeyFrames` 表示当前 `LocalMapping` 线程是否**愿意**接收关键帧,这会被 `Tracking` 线程函数 `Tracking::NeedNewKeyFrame()` 用作是否生产关键帧的参考因素之一;但即使 `mbAcceptKeyFrames` 为 `false` ,在系统很需要关键帧的情况下 `Tracking` 线程函数 `Tracking::NeedNewKeyFrame()` 也会决定生成关键帧.

局部建图主函数: `Run()`

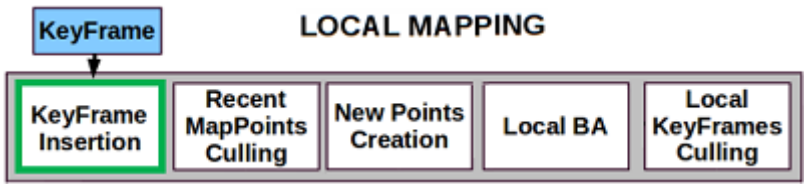


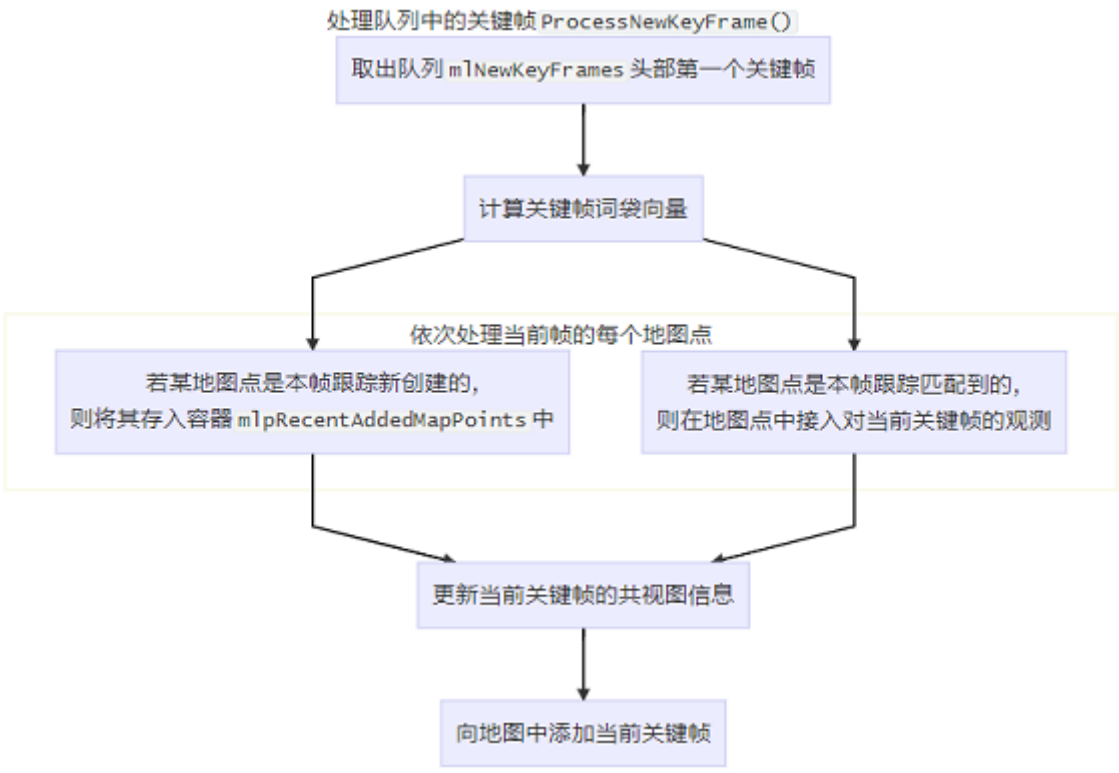


函数`LocalMapping::Run()`是`LocalMapping`线程的主函数,该函数内部是一个死循环,每`3`毫秒查询一次当前线程缓冲队列`mlNewKeyFrames`.若查询到了待处理的新关键帧,就进行查询

```
void LocalMapping::Run() {

    while (1) {
        SetAcceptKeyFrames(false);      // 设置当前LocalMapping线程处于建图状态,不愿意接受Tracking线程传来的关键帧

        // step1. 检查缓冲队列内的关键帧
        if (CheckNewKeyFrames()) {
            // step2. 处理缓冲队列中第一个关键帧
            ProcessNewKeyFrame();

            // step3. 剔除劣质地图点
            MapPointCulling();

            // step4. 创建新地图点
            CreateNewMapPoints();

            if (!CheckNewKeyFrames()) {
                // step5. 将当前关键帧与其共视关键帧地图点融合
                SearchInNeighbors();

                // step6. 局部BA优化: 优化局部地图
                mbAbortBA = false;
                Optimizer::LocalBundleAdjustment(mpCurrentKeyFrame, &mbAbortBA, mpMap);

                // step7. 剔除冗余关键帧
                KeyFrameCulling();
            }

            // step8. 将当前关键帧加入闭环检测中
            mpLoopCloser->InsertKeyFrame(mpCurrentKeyFrame);
        }

        SetAcceptKeyFrames(true);       // 设置当前LocalMapping线程处于空闲状态,愿意接受Tracking线程传来的关键帧

        // 线程暂停3毫秒再开启下一轮查询
        std::this_thread::sleep_for(std::chrono::milliseconds(3));
    }
}
```

处理队列中第一个关键帧: `ProcessNewKeyFrame()`



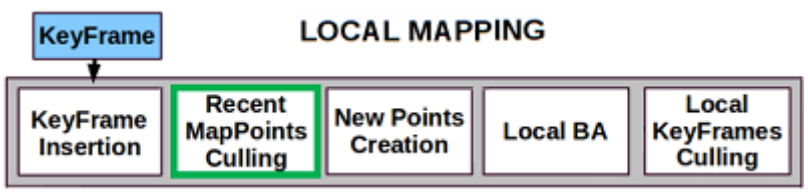


在第3步中处理当前关键点时比较有意思,通过判断该**地图点是否观测到当前关键帧**(`pMP->IsInKeyFrame(mpCurrentKeyFrame)`)来判断该地图点是否是当前关键帧中新生成的.

- 若地图点是本关键帧跟踪过程中匹配得到的(`Tracking::TrackWithMotionModel()`、`Tracking::TrackReferenceKeyFrame()`、`Tracking::Relocalization()`和`Tracking::SearchLocalPoints()`中调用了`ORBmatcher::SearchByProjection()`和`ORBmatcher::SearchByBoW()`方法),则是之前关键帧中创建的地图点,只需添加其对当前帧的观测即可.
- 若地图点是本关键帧跟踪过程中新生成的(包括:1.单目或双目初始化`Tracking::MonocularInitialization()`、`Tracking::StereoInitialization()`;2.创建新关键帧`Tracking::CreateNewKeyFrame()`),则该地图点中有对当前关键帧的观测,是新生成的地图点,放入容器`mlNewKeyFrames`中供`LocalMapping::MapPointCulling()`函数筛选.

```
void LocalMapping::ProcessNewKeyFrame() {

    // step1. 取出队列头的关键帧
    {
        unique_lock<mutex> lock(mMutexNewKFs);
        mpCurrentKeyFrame = mlNewKeyFrames.front();
        mlNewKeyFrames.pop_front();
    }

    // step2. 计算该关键帧的词向量
    mpCurrentKeyFrame->ComputeBow();

    // step3. 根据地图点中是否观测到当前关键帧判断该地图是是否是新生成的
    const vector<MapPoint *> vpMapPointMatches = mpCurrentKeyFrame->GetMapPointMatches();
    for (size_t i = 0; i < vpMapPointMatches.size(); i++) {
        MapPoint *pMP = vpMapPointMatches[i];
        if (pMP && !pMP->isBad()) {
            if (!pMP->IsInKeyFrame(mpCurrentKeyFrame)) {
                // step3.1. 该地图点是跟踪本关键帧时匹配得到的,在地图点中加入对当前关键帧的观测
                pMP->AddObservation(mpCurrentKeyFrame, i);
                pMP->UpdateNormalAndDepth();
                pMP->ComputeDistinctiveDescriptors();
            } else // this can only happen for new stereo points inserted by the Tracking
            {
                // step3.2. 该地图点是跟踪本关键帧时新生成的,将其加入容器mlpRecentAddedMapPoints待筛选
                mlpRecentAddedMapPoints.push_back(pMP);
            }
        }
    }

    // step4. 更新共视图关系
    mpCurrentKeyFrame->UpdateConnections();

    // step5. 将关键帧插入到地图中
    mpMap->AddKeyFrame(mpCurrentKeyFrame);
}
```

剔除坏地图点: `MapPointCulling()`



冗余地图点的标准:满足以下其中之一就算是坏地图点

1. 召回率$=\frac{\text{实际观测到该地图点的帧数}\ mnFound}{\text{理论上应当观测到该地图点的帧数}\ mnVisible}<0.25$
2. 在创建的3帧内观测数目少于2(双目为3)

若地图点经过了连续3个关键帧仍未被剔除,则被认为是好的地图点

```cpp
void LocalMapping::MapPointCulling() {
    list<MapPoint *>::iterator lit = mlpRecentAddedMapPoints.begin();
    const unsigned long int nCurrentKFid = mpCurrentKeyFrame->mnId;

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

    while (lit != mlpRecentAddedMapPoints.end()) {
        MapPoint *pMP = *lit;
        if (pMP->isBad()) {
            // 标准0: 地图点在其他地方被删除了
            lit = mlpRecentAddedMapPoints.erase(lit);
        } else if (pMP->GetFoundRatio() < 0.25f) {
            // 标准1: 召回率<0.25
            pMP->SetBadFlag();
            lit = mlpRecentAddedMapPoints.erase(lit);
        } else if (((int) nCurrentKFid - (int) pMP->mnFirstKFid) >= 2 && pMP->Observations() <= cnThObs)
{
            // 标准2: 从创建开始连续3个关键帧内观测数目少于cnThObs
            pMP->SetBadFlag();
            lit = mlpRecentAddedMapPoints.erase(lit);
        } else if (((int) nCurrentKFid - (int) pMP->mnFirstKFid) >= 3)
            // 通过了3个关键帧的考察,认为是好的地图点
            lit = mlpRecentAddedMapPoints.erase(lit);
        else
            lit++;
    }
}
```

`MapPoint`类中关于召回率的成员函数和变量如下:

成员函数/变量	访问控制	意义	初值
`int mnFound`	`protected`	实际观测到该地图点的帧数	`1`
`int mnVisible`	`protected`	理论上应当观测到该地图点的帧数	`1`
`float GetFoundRatio()`	`public`	召回率 $=\frac{\text{实际观测到该地图点的帧数 } mnFound}{\text{理论上应当观测到该地图点的帧数 } mnVisible}$	
`void IncreaseFound(int n=1)`	`public`	`mnFound`加1	
`void IncreaseVisible(int n=1)`	`public`	`mnVisible`加1	

这两个成员变量主要用于`Tracking`线程.

- 在函数`Tracking::SearchLocalPoints()`中,会对所有处于当前帧视锥内的地图点调用成员函数`MapPoint::IncreaseVisible()`.(这些点未必真的被当前帧观测到了,只是地理位置上处于当前帧视锥范围内).

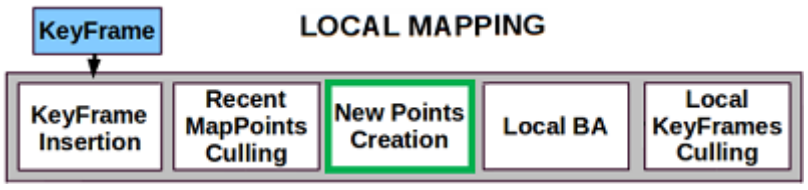
```cpp
void Tracking::SearchLocalPoints() {
    // 当前关键帧的地图点
    for (MapPoint *pMP : mCurrentFrame.mvpMapPoints) {
        pMP->IncreaseVisible();
            }
    }

    // 局部关键帧中不属于当前帧,但在当前帧视锥范围内的地图点
    for (MapPoint *pMP = *vit : mvpLocalMapPoints.begin()) {
        if (mCurrentFrame.isInFrustum(pMP, 0.5)) {
            pMP->IncreaseVisible();
        }
    }

    // ...
}
```

- 在函数`Tracking::TrackLocalMap()`中,会对所有当前帧观测到的地图点调用`MaoPoint::IncreaseFound()`.

```cpp
bool Tracking::TrackLocalMap() {

    // ...

    for (int i = 0; i < mCurrentFrame.N; i++) {
        if (mCurrentFrame.mvpMapPoints[i]) {
            if (!mCurrentFrame.mvbOutlier[i]) {
                // 当前帧观测到的地图点
                mCurrentFrame.mvpMapPoints[i]->IncreaseFound();
                // ...
            }
```

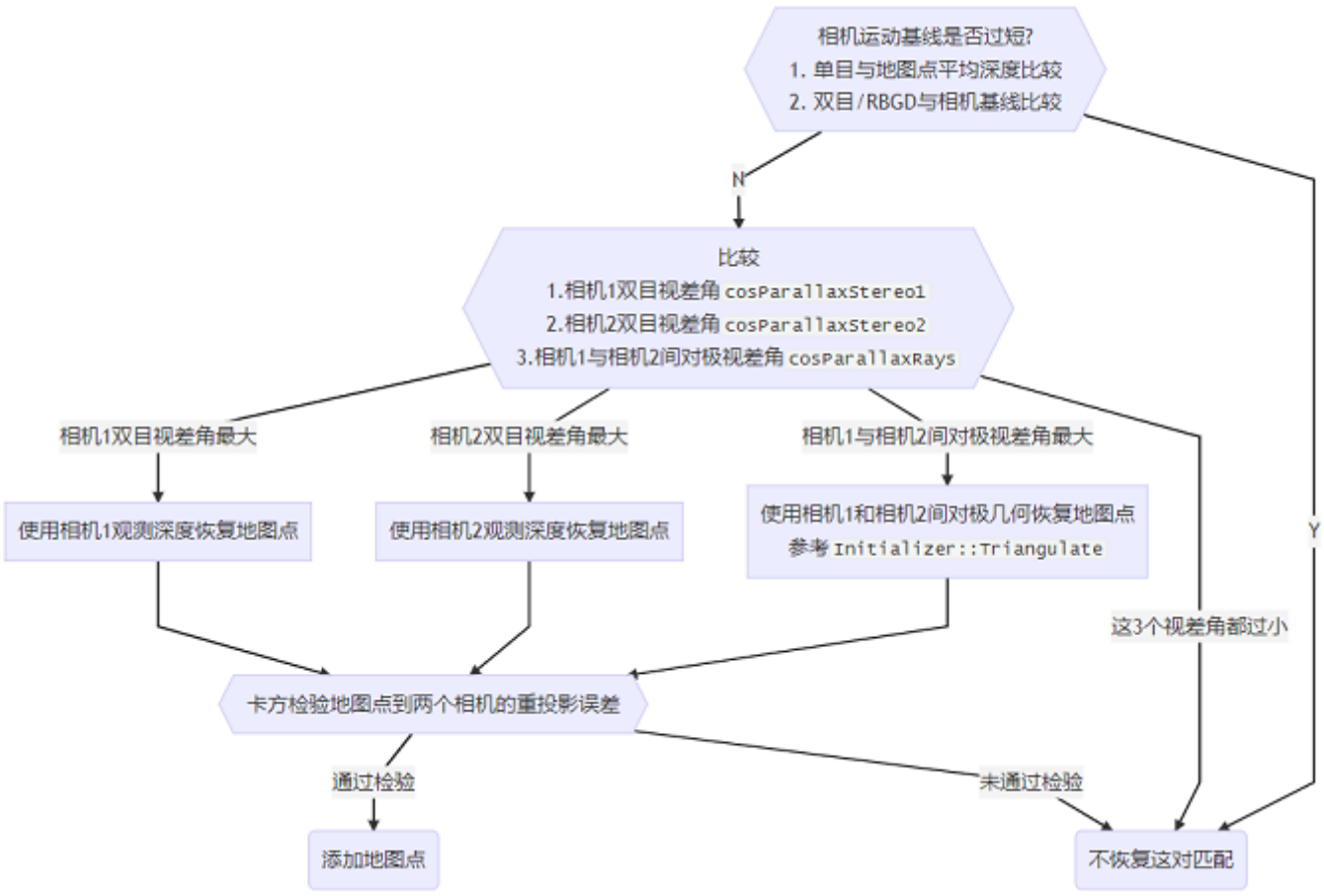
```
        }
    }

    // ...
}
```

创建新地图点: `CreateNewMapPoints()`

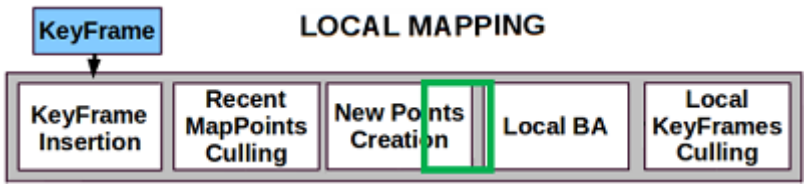


将当前关键帧分别与共视程度最高的前`10`(单目相机取`20`)个共视关键帧两两进行特征匹配,生成地图点.

对于双目相机的匹配特征点对,可以根据某帧特征点深度恢复地图点,也可以根据两帧间对极几何三角化地图点,这里取视差角最大的方式来生成地图点.

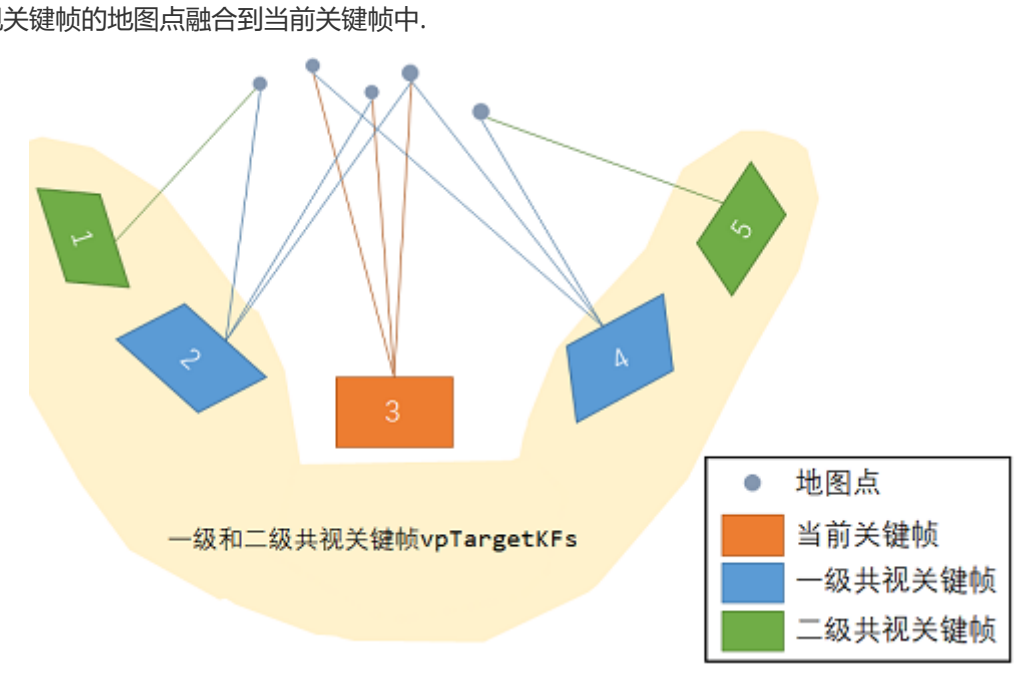


融合当前关键帧和其共视帧的地图点: `SearchInNeighbors()`



本函数将**当前关键帧**与其**一级和二级共视关键帧**做地图点融合,分两步:

1. 正向融合: 将当前关键帧的地图点融合到各共视关键帧中.
2. 反向融合: 将各共视关键帧的地图点融合到当前关键帧中.



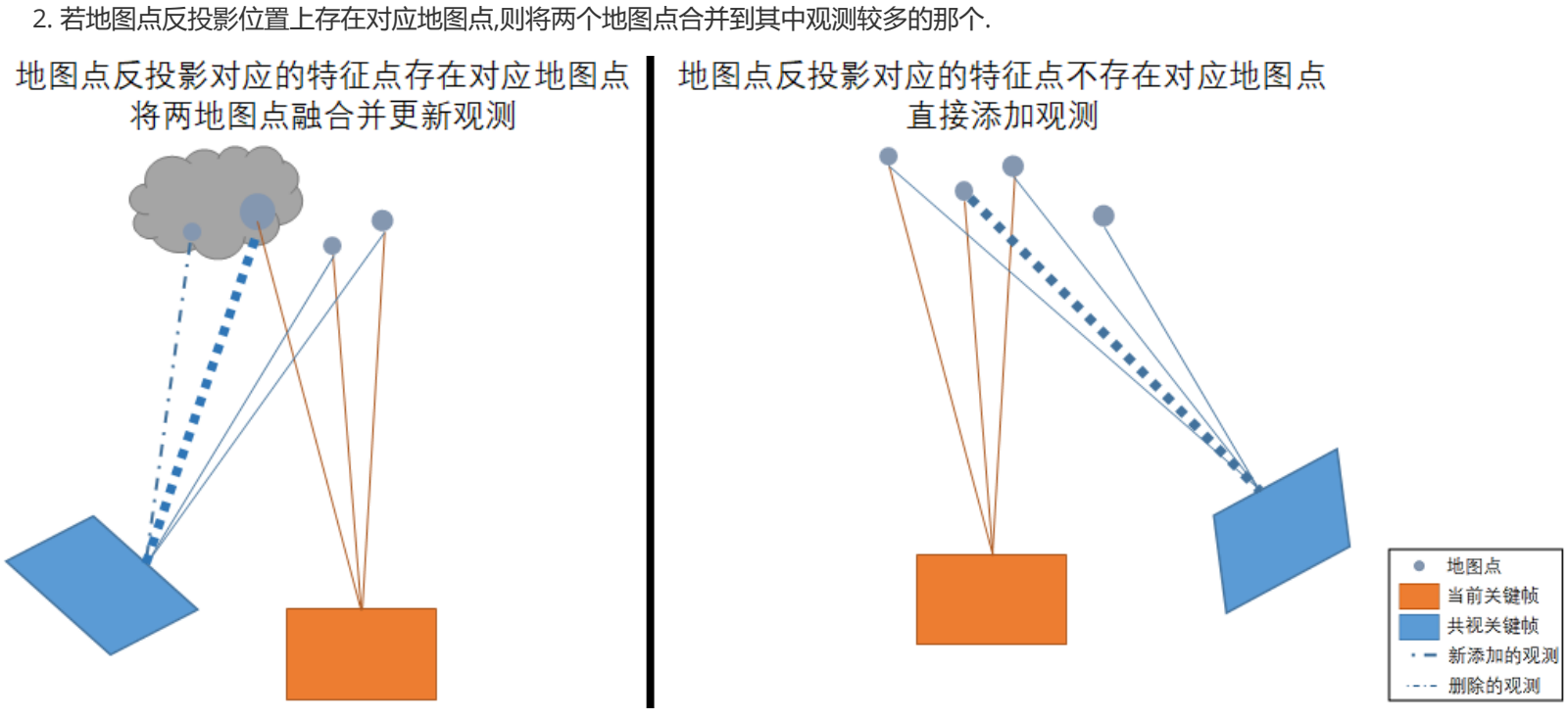

```
void LocalMapping::SearchInNeighbors() {
    // step1. 取当前关键帧的一级共视关键帧
    const vector<KeyFrame *> vpNeighKFs = mpCurrentKeyFrame->GetBestCovisibilityKeyFrames(10);

    // step2. 遍历一级关键帧,寻找二级关键帧
    vector<KeyFrame *> vpTargetKFs;
    for (KeyFrame *pKFi : vpNeighKFs) {
        if (pKFi->isBad() || pKFi->mnFuseTargetForKF == mpCurrentKeyFrame->mnId)
            continue;
        vpTargetKFs.push_back(pKFi);
```

```
        pKFi->mnFuseTargetForKF = mpCurrentKeyFrame->mnId;
        const vector<KeyFrame *> vpSecondNeighKFs = pKFi->GetBestCovisibilityKeyFrames(5);
        for (KeyFrame *pKFi2 : vpSecondNeighKFs) {
            if (pKFi2->isBad() || pKFi2->mnFuseTargetForKF == mpCurrentKeyFrame->mnId || pKFi2->mnId ==
mpCurrentKeyFrame->mnId)
                continue;
            vpTargetKFs.push_back(pKFi2);
        }
    }


    // step3. 正向融合: 将当前帧的地图点融合到各共视关键帧中
    vector<MapPoint *> vpMapPointMatches = mpCurrentKeyFrame->GetMapPointMatches();
    ORBmatcher matcher;
    for (KeyFrame *pKFi : vpTargetKFs) {
        matcher.Fuse(pKFi, vpMapPointMatches);
    }

    // step4. 反向融合: 将各共视关键帧的地图点融合到当前关键帧中
    // step4.1. 取出各共视关键帧的地图点存入vpFuseCandidates
    vector<MapPoint *> vpFuseCandidates;
    for (KeyFrame *pKFi : vpTargetKFs) {
        vector<MapPoint *> vpMapPointsKFi = pKFi->GetMapPointMatches();
        for (MapPoint *pMP : vpMapPointsKFi.begin()) {
            if (!pMP || pMP->isBad() || pMP->mnFuseCandidateForKF == mpCurrentKeyFrame->mnId)
                continue;
            pMP->mnFuseCandidateForKF = mpCurrentKeyFrame->mnId;
            vpFuseCandidates.push_back(pMP);
        }
    }

    // step 4.2. 进行反向融合
    matcher.Fuse(mpCurrentKeyFrame, vpFuseCandidates);

    // step5. 更新当前关键帧的地图点信息
    vpMapPointMatches = mpCurrentKeyFrame->GetMapPointMatches();
    for (MapPoint *pMP : vpMapPointMatches) {
        if (pMP and !pMP->isBad()) {
            pMP->ComputeDistinctiveDescriptors();
            pMP->UpdateNormalAndDepth();
        }
    }

    // step6. 更新共视图
    mpCurrentKeyFrame->UpdateConnections();
}
```

`ORBmatcher::Fuse()` 将地图点与帧中图像的特征点匹配,实现地图点融合.

在将地图点反投影到帧中的过程中,存在以下两种情况:

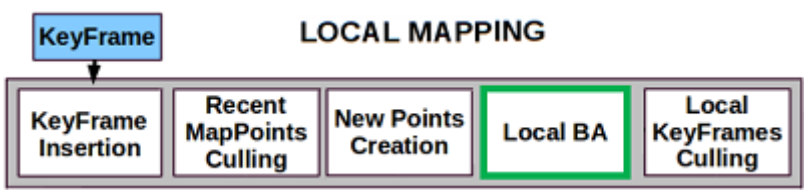
1. 若地图点反投影对应位置上不存在地图点,则直接添加观测.
2. 若地图点反投影位置上存在对应地图点,则将两个地图点合并到其中观测较多的那个.




```
int ORBmatcher::Fuse(KeyFrame *pKF, const vector<MapPoint *> &vpMapPoints, const float th) {
    // 遍历所有的待投影地图点
    for(MapPoint* pMP : vpMapPoints) {
        // step1. 将地图点反投影到相机成像平面上
        const float invz = 1/p3Dc.at<float>(2);
        const float x = p3Dc.at<float>(0)*invz;
        const float y = p3Dc.at<float>(1)*invz;
        const float u = fx*x+cx;
        const float v = fy*y+cy;
        const float ur = u-bf*invz;
        const float maxDistance = pMP->GetMaxDistanceInvariance();
```

```
        const float minDistance = pMP->GetMinDistanceInvariance();
        cv::Mat PO = p3Dw-Ow;
        const float dist3D = cv::norm(PO);

        // step2. 地图点观测距离
        if(dist3D<minDistance || dist3D>maxDistance )
            continue;

        // step3. 地图点的观测距离和观测方向不能太离谱
        if (dist3D < minDistance || dist3D > maxDistance)
            continue;
        cv::Mat Pn = pMP->GetNormal();
        if (PO.dot(Pn) < 0.5 * dist3D)
            continue;

        // step4. 在投影位置寻找图像特征点
        int nPredictedLevel = pMP->PredictScale(dist3D, pKF);
        const float radius = th * pKF->mvScaleFactors[nPredictedLevel];
        const vector<size_t> vIndices = pKF->GetFeaturesInArea(u, v, radius);
        const cv::Mat dMP = pMP->GetDescriptor();
        int bestDist = 256;
        int bestIdx = -1;
        for (size_t idx : vIndices) {
            const size_t idx = *vit;
            const cv::KeyPoint &kp = pKF->mvKeysUn[idx];
            const int &kpLevel = kp.octave;
            // step4.1. 金字塔层级要接近
            if (kpLevel < nPredictedLevel - 1 || kpLevel > nPredictedLevel)
                continue;
            // step4.2. 使用卡方检验检查重投影误差,单目和双目的自由度不同
            if (pKF->mvuRight[idx] >= 0) {
                const float ex = u - kp.pt.x;
                const float ey = v - kp.pt.y;
                const float er = ur - pKF->mvuRight[idx];
                const float e2 = ex * ex + ey * ey + er * er;
                if (e2 * pKF->mvInvLevelSigma2[kpLevel] > 7.8)
                    continue;
            } else {
                const float ex = u - kp.pt.x;
                const float ey = v - kp.pt.y;
                const float e2 = ex * ex + ey * ey;
                if (e2 * pKF->mvInvLevelSigma2[kpLevel] > 5.99)
                    continue;
            }
            // step4.3. 检验描述子距离
            const cv::Mat &dKF = pKF->mDescriptors.row(idx);
            const int dist = DescriptorDistance(dMP, dKF);
            if (dist < bestDist) {
                bestDist = dist;
                bestIdx = idx;
            }
        }

        // step5. 与最近特征点的描述子距离足够小,就进行地图点融合
        if (bestDist <= TH_LOW) {
            MapPoint *pMPinKF = pKF->GetMapPoint(bestIdx);
            if (pMPinKF) {
                // step5.1. 地图点反投影位置上存在对应地图点,则将两个地图点合并到其中观测较多的那个则直接添加观测
                if (!pMPinKF->isBad()) {
                    if (pMPinKF->Observations() > pMP->Observations())
                        pMP->Replace(pMPinKF);
                    else
                        pMPinKF->Replace(pMP);
                }
            } else {
                // step5.2. 地图点反投影对应位置上不存在地图点,
                pMP->AddObservation(pKF, bestIdx);
                pKF->AddMapPoint(pMP, bestIdx);
            }
            nFused++;
        }
    }

    return nFused;
}
```

局部BA优化: `Optimizer::LocalBundleAdjustment()`

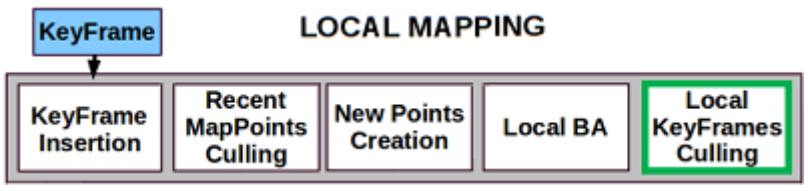


局部BA优化当前帧的局部地图.

- 当前关键帧的一级共视关键帧位姿会被优化;二极共视关键帧会加入优化图,但其位姿不会被优化.
- 所有局部地图点位姿都会被优化.

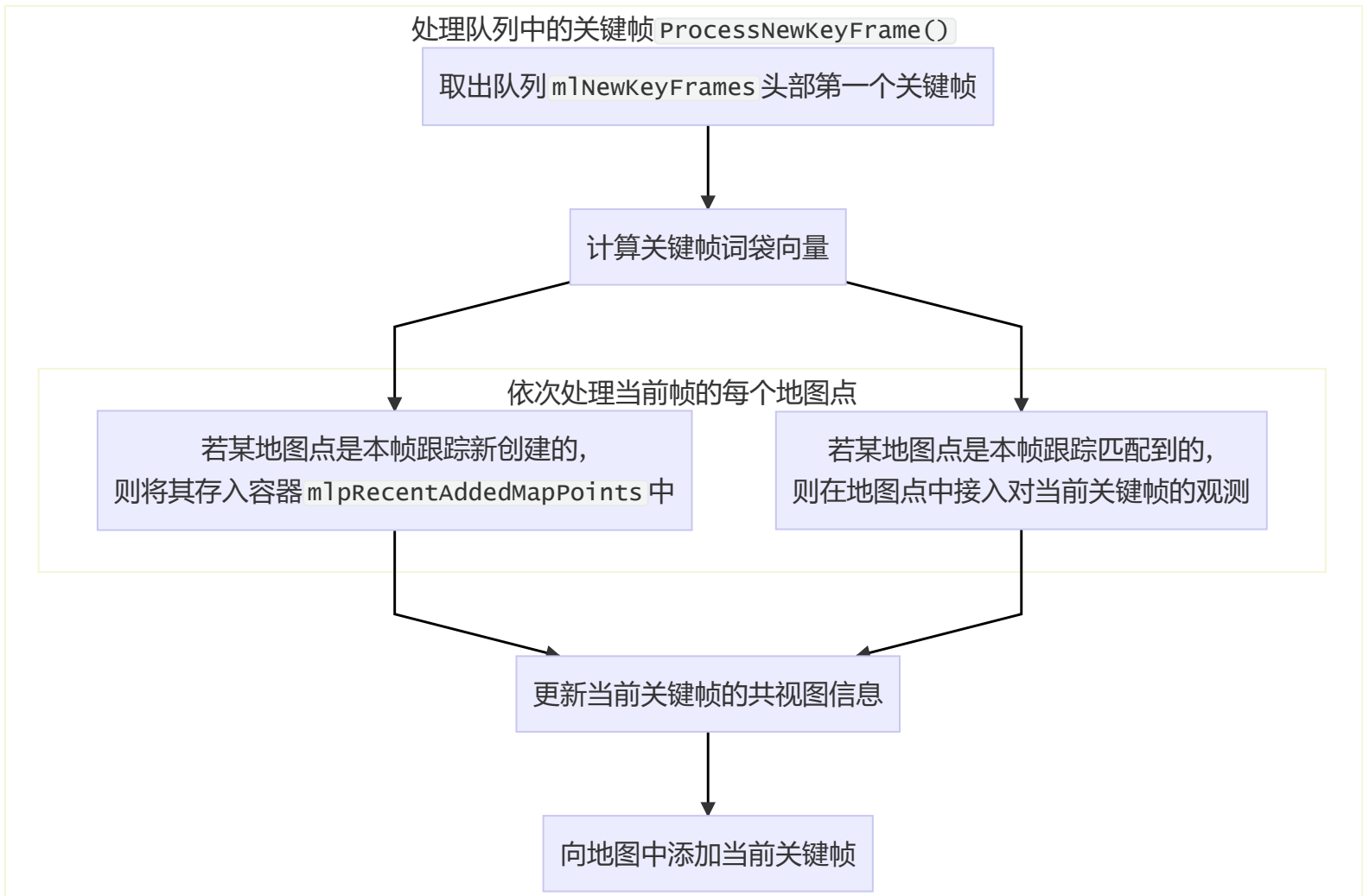
> `Tracking`线程中定义了局部地图成员变量`mvpLocalKeyFrames`和`mvpLocalMapPoints`,但是这些变量并没有被`LocalMapping`线程管理,因此在函数`Optimizer::LocalBundleAdjustment()`中还要重新构造局部地图变量,这种设计有些多此一举了.

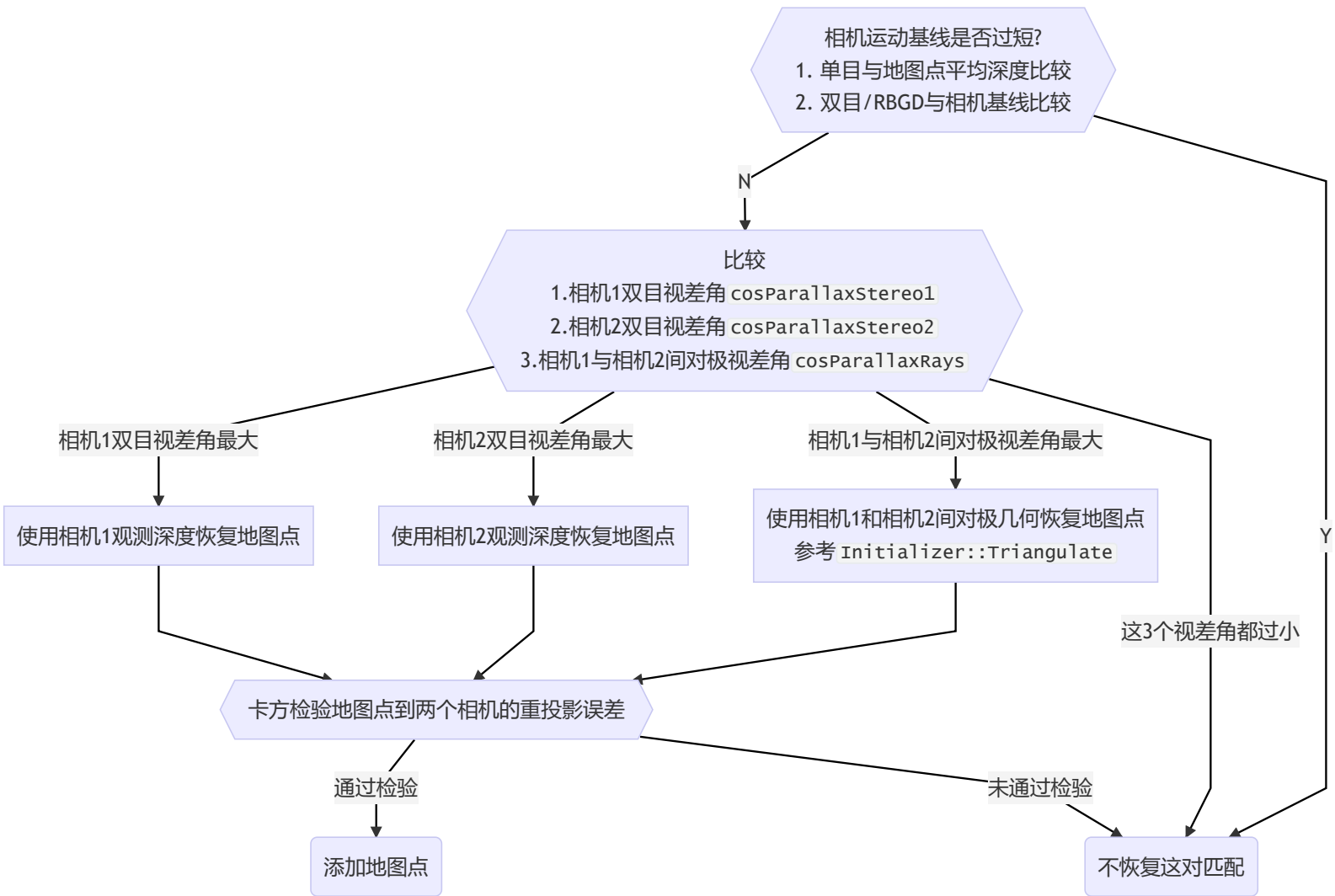
剔除冗余关键帧: `KeyFrameCulling()`



冗余关键帧标准: 90%以上的地图点能被超过3个其他关键帧观测到.

```
void LocalMapping::KeyFrameCulling() {

    // step1. 遍历当前关键帧的所有共视关键帧
    vector<KeyFrame *> vpLocalKeyFrames = mpCurrentKeyFrame->GetVectorCovisibleKeyFrames();
    for (KeyFrame *pKF : vpLocalKeyFrames) {

        // step2. 遍历所有局部地图点
        const vector<MapPoint *> vpMapPoints = pKF->GetMapPointMatches();
        int nRedundantObservations = 0;
        int nMPs = 0;
        for (MapPoint *pMP : vpMapPoints) {
            if (pMP && !pMP->isBad()) {
                if (!mbMonocular) {
                    // 双目相机只能看到不超过相机基线35倍的地图点
                    if (pKF->mvDepth[i] > pKF->mThDepth || pKF->mvDepth[i] < 0)
                        continue;
                }
                nMPs++;

                int nObs = 0;
                for (KeyFrame *pKFi : pMP->GetObservations()) {
                     = mit->first;
                    if (pKFi->mvKeysUn[mit->second].octave <= pKF->mvKeysUn[i].octave + 1) {
                        nObs++;
                        if (nObs >= 3)
                            break;
                    }
                }
                if (nObs >= 3) {
                    nRedundantObservations++;
                }
            }
        }
    }

    // step3. 若关键帧超过90%的地图点能被超过3个其它关键帧观测到,则视为冗余关键帧
    if (nRedundantObservations > 0.9 * nMPs)
        pKF->SetBadFlag();
}
```


处理队列中的关键帧 ProcessNewKeyFrame()
取出队列 mlNewKeyFrames 头部第一个关键帧
计算关键帧词袋向量
依次处理当前帧的每个地图点
若某地图点是本帧跟踪新创建的，
则将其存入容器 mlpRecentAddedMapPoints 中
若某地图点是本帧跟踪匹配到的，
则在地图点中接入对当前关键帧的观测
更新当前关键帧的共视图信息
向地图中添加当前关键帧


相机运动基线是否过短?
1. 单目与地图点平均深度比较
2. 双目/RBGD与相机基线比较
N
Y
比较
1.相机1双目视差角 cosParallaxStereo1
2.相机2双目视差角 cosParallaxStereo2
3.相机1与相机2间对极视差角 cosParallaxRays
相机1双目视差角最大
相机2双目视差角最大
相机1与相机2间对极视差角最大
这3个视差角都过小
使用相机1观测深度恢复地图点
使用相机2观测深度恢复地图点
使用相机1和相机2间对极几何恢复地图点
参考 Initializer::Triangulate
卡方检验地图点到两个相机的重投影误差
通过检验
未通过检验
添加地图点
不恢复这对匹配