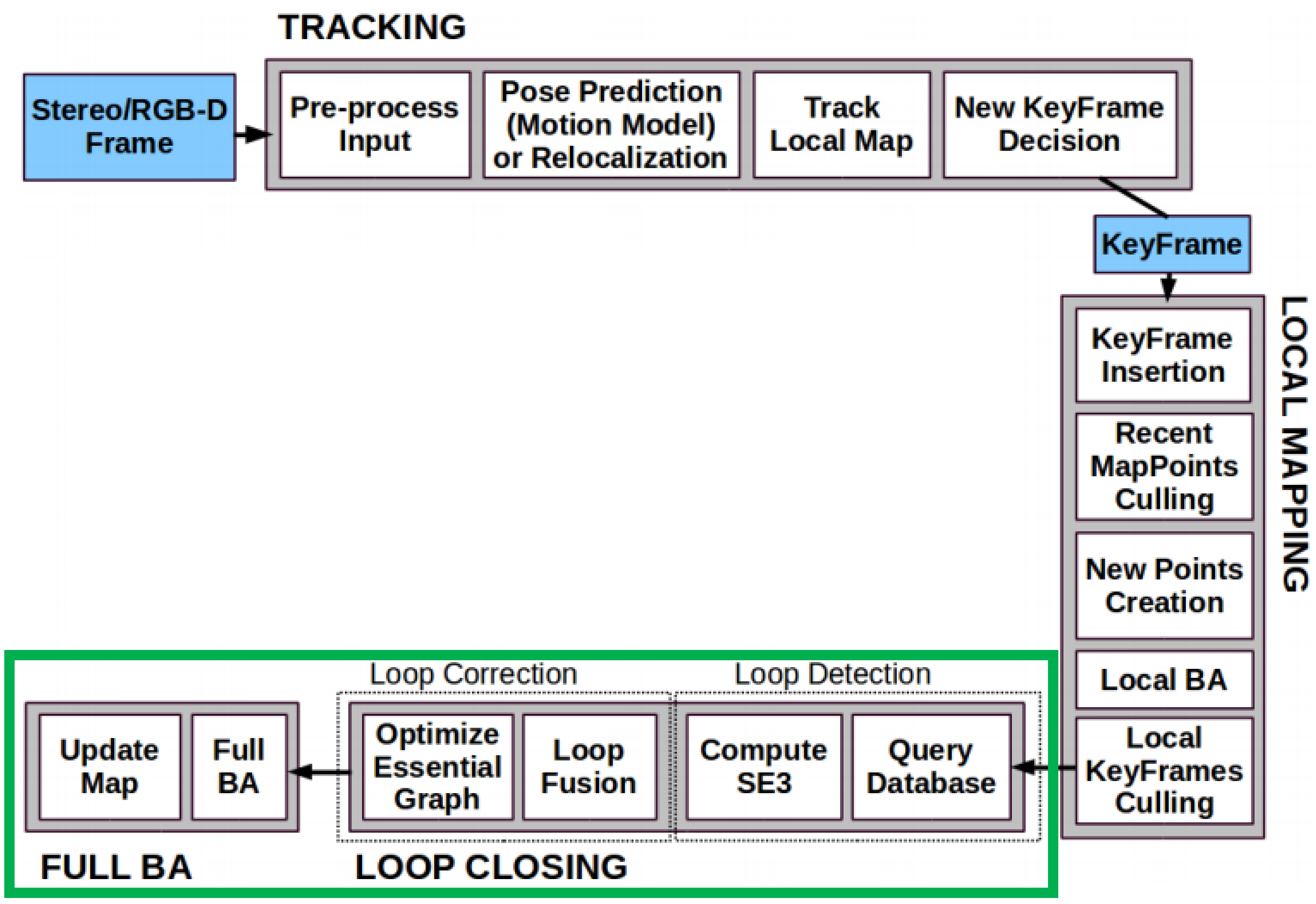
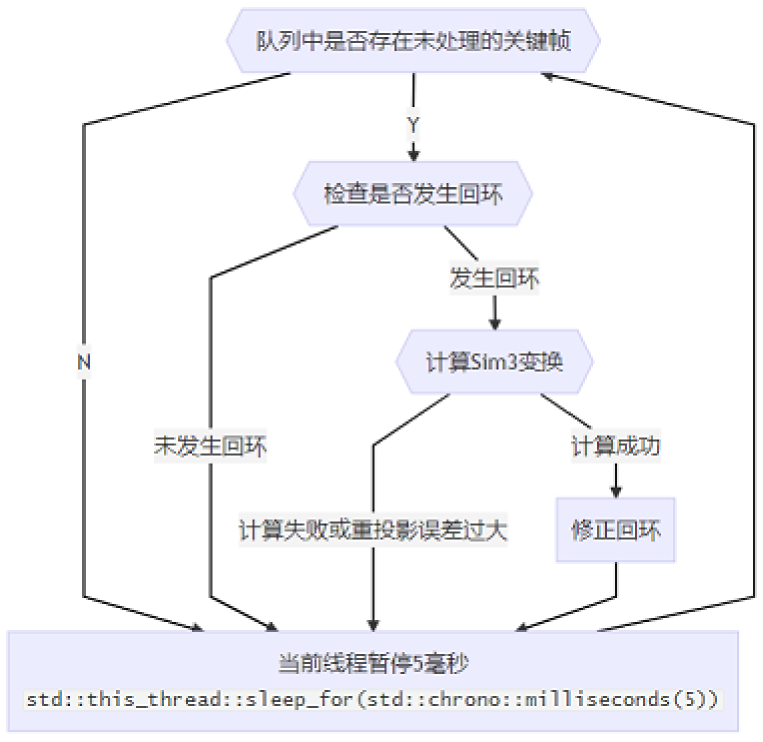


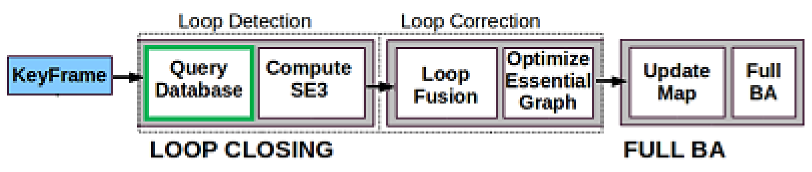
各成员函数/变量

闭环主函数: `Run()`



```
void LoopClosing::Run() {
    while (1) {
        if (CheckNewKeyFrames()) {
            if (DetectLoop()) {
                if (ComputeSim3()) {
                    CorrectLoop();
                }
            }
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(5));
    }
}
```

闭环检测: `DetectLoop()`



`LoopClosing` 类中定义类型 `ConsistentGroup`,表示关键帧组.

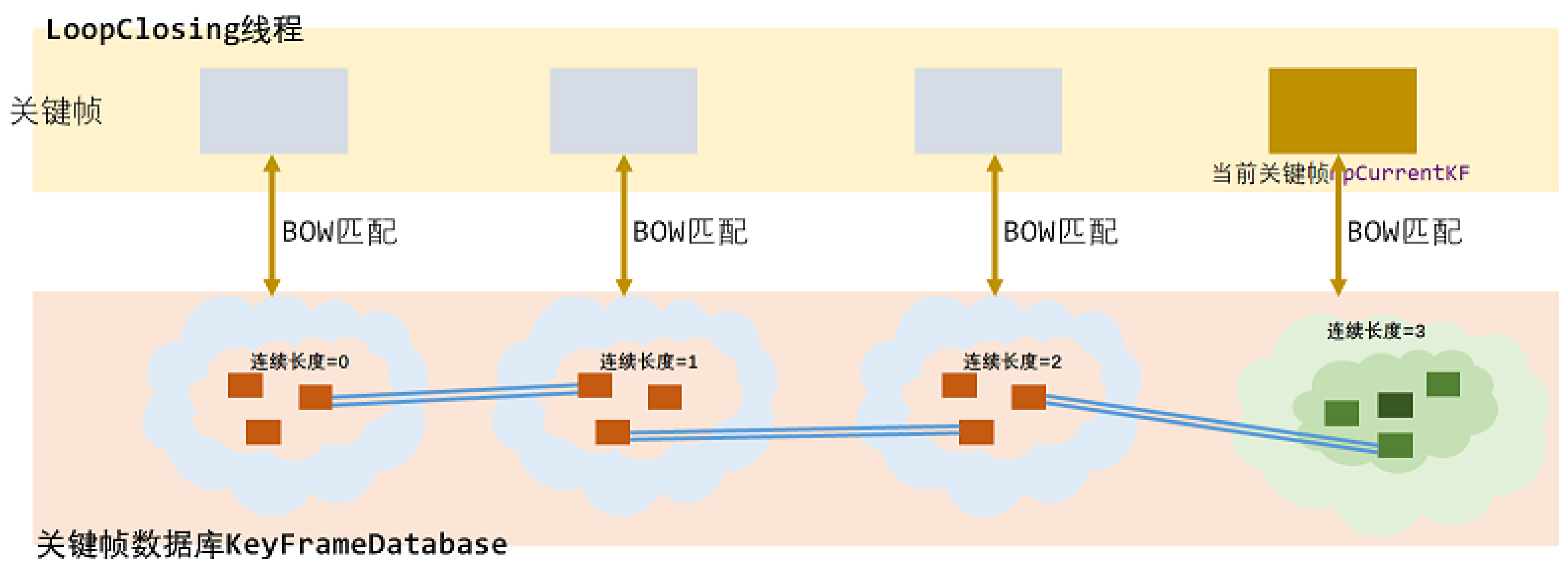
```
typedef pair<set<KeyFrame *>, int> ConsistentGroup
```

- 第一个元素表示一组共视关键帧.
- 第二个元素表示该关键帧组的连续长度.

所谓连续,指的是两个关键帧组中存在相同的关键帧.

成员函数/变量	访问控制	意义
`KeyFrame *mpCurrentKF`	`protected`	当前关键帧
`KeyFrame *mpMatchedKF`	`protected`	当前关键帧的闭环匹配关键帧
`std::vector<ConsistentGroup> mvConsistentGroups`	`protected`	前一关键帧的闭环候选关键帧组
`vCurrentConsistentGroups`	局部变量	当前关键帧的闭环候选关键帧组
`std::vector<KeyFrame *> mvpEnoughConsistentCandidates`	`protected`	所有达到足够连续数的关键帧

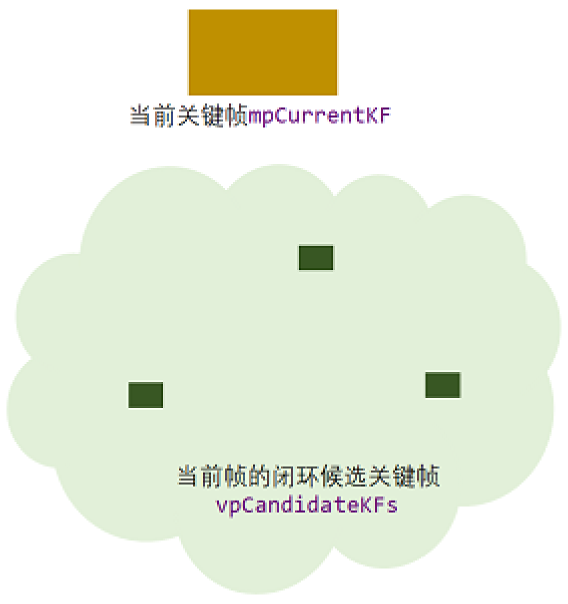
闭环检测原理: 若**连续4个**关键帧都能在数据库中找到对应的闭环匹配关键帧组,且这些闭环匹配关键帧组间是连续的,则认为实现闭环,



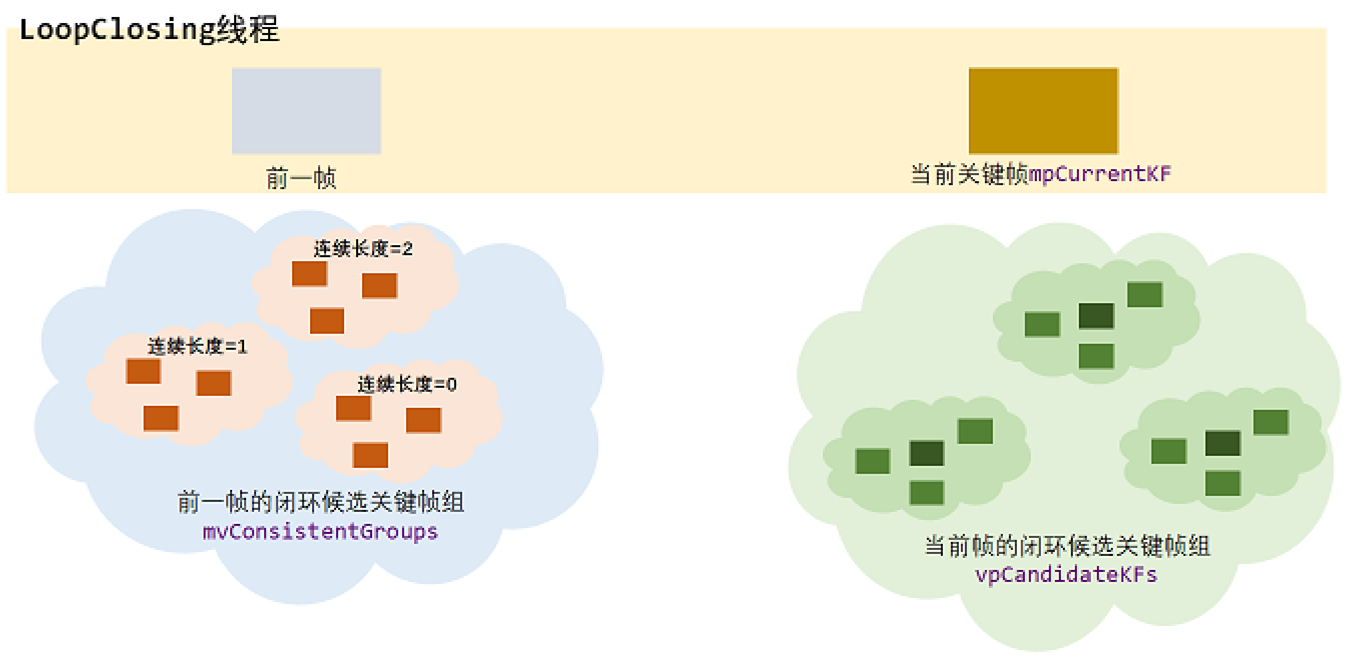
具体来说,回环检测过程如下:

1. 找到当前关键帧的闭环候选关键帧 `vpCandidateKFs`:

   闭环候选关键帧取自于与当前关键帧**具有相同的BOW向量**但**不存在直接连接**的关键帧.

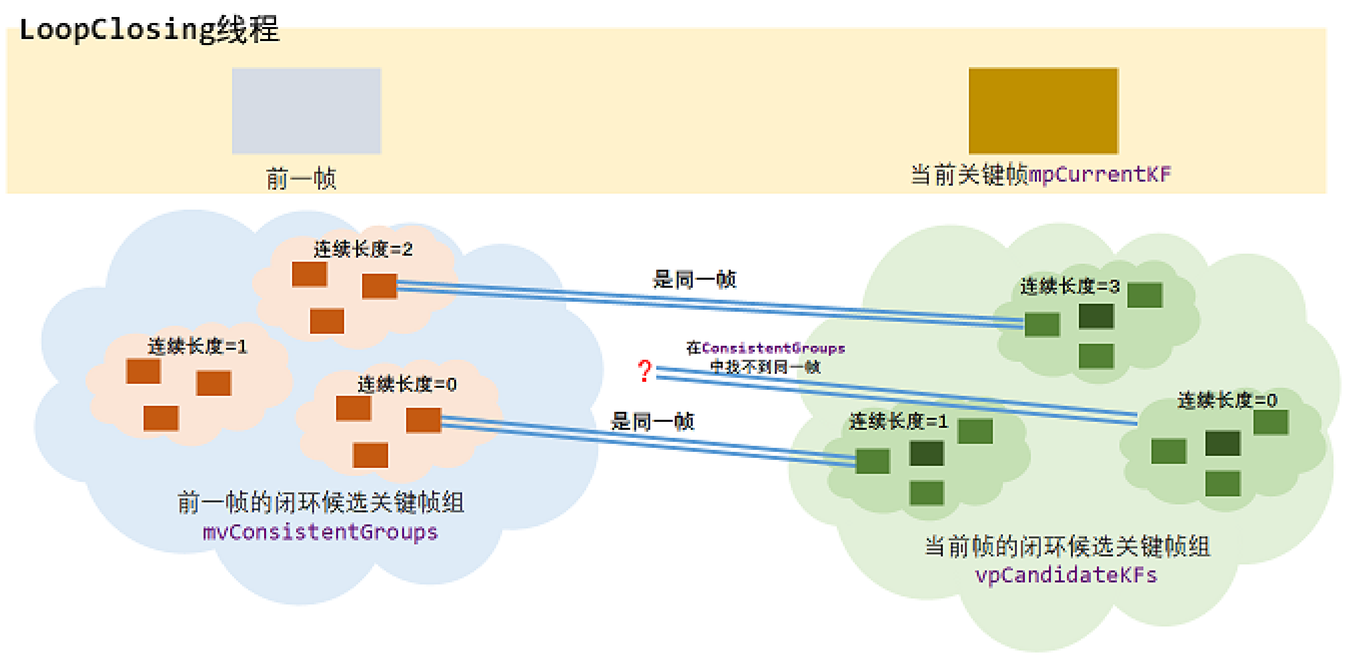


2. 将闭环候选关键帧和其共视关键帧组合成为关键帧组 `vCurrentConsistentGroups`:



3. 在当前关键组和之前的连续关键组间寻找连续关系.
   - 若当前关键帧组在之前的连续关键帧组中找到连续关系,则当前的连续关键帧组的连续长度加`1`.
   - 若当前关键帧组在之前的连续关键帧组中没能找到连续关系,则当前关键帧组的连续长度为`0`.

关键帧组的连续关系是指两个关键帧组间是否有关键帧同时存在于两关键帧组中.



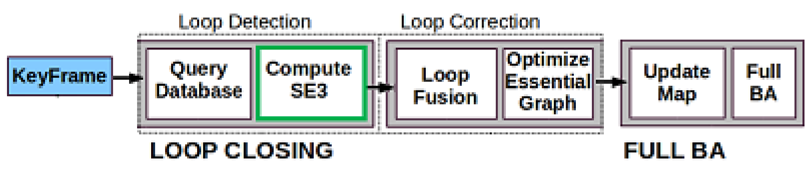
若某关键帧组的连续长度达到`3`,则认为该关键帧实现闭环.

```cpp
bool LoopClosing::DetectLoop() {
    // step1. 取出缓冲队列头部的关键帧,作为当前检测闭环关键帧,设置其不被优化删除
    {
        unique_lock<mutex> lock(mMutexLoopQueue);
        mpCurrentKF = mlpLoopKeyFrameQueue.front();
        mlpLoopKeyFrameQueue.pop_front();
        mpCurrentKF->SetNotErase();
    }

    // step2. 距离上次闭环时间太短,不再检测闭环
    if (mpCurrentKF->mnId < mLastLoopKFid + 10) {
        mpKeyFrameDB->add(mpCurrentKF);
        mpCurrentKF->SetErase();
        return false;
    }

    // step3. 计算当前关键帧与共视关键帧间最大相似度
    const vector<KeyFrame *> vpConnectedKeyFrames = mpCurrentKF->GetVectorCovisibleKeyFrames();
    const DBoW2::BowVector &CurrentBowVec = mpCurrentKF->mBowVec;
    float minScore = 1;
    for (KeyFrame *pKF : vpConnectedKeyFrames) {
        const DBoW2::BowVector &BowVec = pKF->mBowVec;
        float score = mpORBVocabulary->score(CurrentBowVec, BowVec);
        if (score < minScore)
            minScore = score;
    }

    // step4. 寻找当前关键帧的闭环候选关键帧
    vector<KeyFrame *> vpCandidateKFs = mpKeyFrameDB->DetectLoopCandidates(mpCurrentKF, minScore);
    if (vpCandidateKFs.empty()) {
        mpKeyFrameDB->add(mpCurrentKF);
        mvConsistentGroups.clear();
        mpCurrentKF->SetErase();
        return false;
    }

    // step5. 在当前关键帧组和之前的连续关键帧组之间寻找匹配
    mvpEnoughConsistentCandidates.clear();
    vector<ConsistentGroup> vCurrentConsistentGroups;
    vector<bool> vbConsistentGroup(mvConsistentGroups.size(), false);   // 之前的连续关键帧组在当前关键帧组中是否存在连续

    // 遍历当前闭环候选关键帧
    for (KeyFrame *pCandidateKF : vpCandidateKFs) {
        // step5.1. 构建关键帧组,包括候选关键帧及其共视关键帧
        set<KeyFrame *> spCandidateGroup = pCandidateKF->GetConnectedKeyFrames();
        spCandidateGroup.insert(pCandidateKF);

        // step5.2. 遍历之前的连续关键帧组,寻找连续关系
        bool bEnoughConsistent = false;
        bool bConsistentForSomeGroup = false;
        for (size_t iG = 0, iendG = mvConsistentGroups.size(); iG < iendG; iG++) {
            set<KeyFrame *> sPreviousGroup = mvConsistentGroups[iG].first;
            bool bConsistent = false;
            // step5.2. 若当前连续关键帧组中某关键帧也在前一帧的候选关键帧组中,则找到了连续关系
            for (KeyFrame * previousKeyFrame : spCandidateGroup.begin()) {
                if (sPreviousGroup.count(previousKeyFrame)) {
                    bConsistent = true;
                    bConsistentForSomeGroup = true;
                    break;
                }
            }

            // step5.3. 更新当前关键帧组的连续次数
            if (bConsistent) {
                int nCurrentConsistency = mvConsistentGroups[iG].second + 1;
                if (!vbConsistentGroup[iG]) {
                    ConsistentGroup cg = make_pair(spCandidateGroup, nCurrentConsistency);
                    vCurrentConsistentGroups.push_back(cg);
                    vbConsistentGroup[iG] = true;
                }
                // 若当前关键帧组的连续次数达到3,则完成闭环,将其加入到mvpEnoughConsistentCandidates中
                if (nCurrentConsistency >= mnCovisibilityConsistencyTh && !bEnoughConsistent) {
                    mvpEnoughConsistentCandidates.push_back(pCandidateKF);
                    bEnoughConsistent = true;
                }
            }
        }

        // 5.4. 若当前关键帧组在前一关键帧的闭环候选关键帧组中找不到连续关系,则将两虚次数置零
        if (!bConsistentForSomeGroup) {
            ConsistentGroup cg = make_pair(spCandidateGroup, 0);
            vCurrentConsistentGroups.push_back(cg);
        }
    }

    // step6. 维护循环变量
    mvConsistentGroups = vCurrentConsistentGroups;      // 更新连续关键帧组
    mpKeyFrameDB->add(mpCurrentKF);                     // 将当前关键帧加入到关键帧数据库中

    if (mvpEnoughConsistentCandidates.empty()) {
        mpCurrentKF->SetErase();
        return false;
    } else {
        return true;
    }
}
```

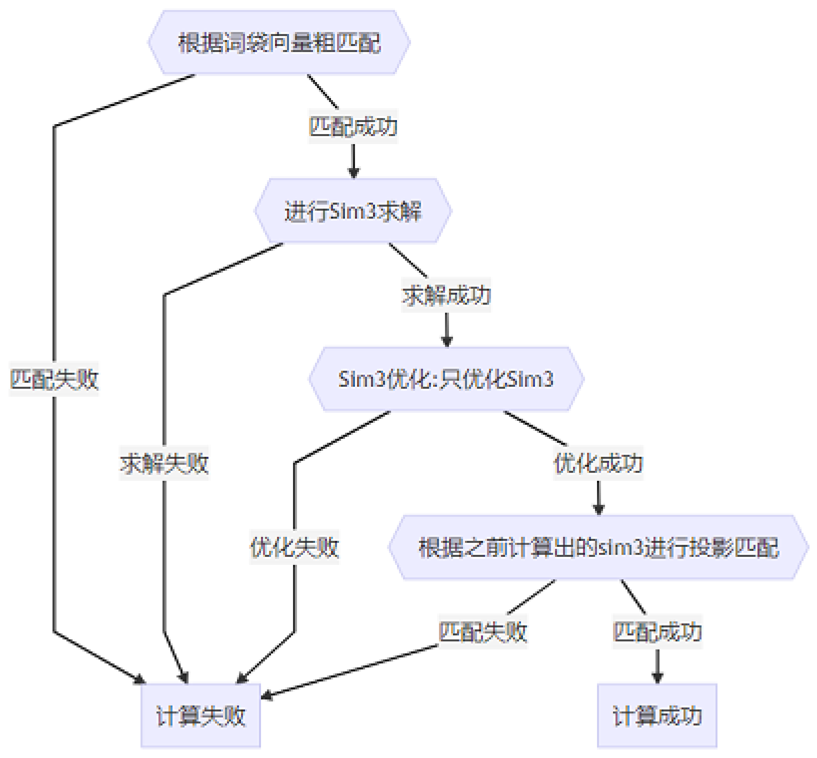
当前关键帧的闭环候选关键帧取自于与当前关键帧**具有相同BOW向量**但**不直接相连**的关键帧.

```cpp
// 寻找当前关键帧的闭环候选关键帧
vector<KeyFrame *> KeyFrameDatabase::DetectLoopCandidates(KeyFrame *pKF, float minScore) {

    // step1. 找出当前关键帧的所有共视关键帧
    set<KeyFrame *> spConnectedKeyFrames = pKF->GetConnectedKeyFrames();


    // step2. 找出所有具有相同BOW但不直接相连的关键帧
    list<KeyFrame *> lKFsSharingWords;      // 存储闭环候选关键帧
    {
        unique_lock<mutex> lock(mMutex);
        // 遍历所有BOW词向量
        for (DBoW2::BowVector vit : pKF) {
            // 遍历所有含有该词向量的关键帧
            for (KeyFrame *pKFi : mvInvertedFile[vit.first]) {
                if (pKFi->mnLoopQuery != pKF->mnId) {
                    pKFi->mnLoopWords = 0;
                    // 若该关键帧与当前关键帧不直接相连,才能作为闭环候选
                    if (!spConnectedKeyFrames.count(pKFi)) {
                        pKFi->mnLoopQuery = pKF->mnId;
                        lKFsSharingWords.push_back(pKFi);
                    }
                }
                pKFi->mnLoopWords++;
            }
        }
    }

    // step3. 以最大相似度的0.8倍为阈值筛选筛选候选关键帧
    int maxCommonWords = 0;
    list<pair<float, KeyFrame *> > lScoreAndMatch;
    for (KeyFrame *pKFi : lKFsSharingWords) {
        if (*pKFi->mnLoopWords > maxCommonWords)
            maxCommonWords = *pKFi->mnLoopWords;
    }
    int minCommonWords = maxCommonWords * 0.8f;
    for (KeyFrame *pKFi : lKFsSharingWords) {
        if (pKFi->mnLoopWords > minCommonWords) {
            float si = mpVoc->score(pKF->mBowVec, pKFi->mBowVec);
            pKFi->mLoopScore = si;
            if (si >= minScore)
                lScoreAndMatch.push_back(make_pair(si, pKFi));
        }
    }

    // step4. 统计候选关键帧的共视关键帧组的相似度得分
    list<pair<float, KeyFrame *> > lAccScoreAndMatch;
    float bestAccScore = minScore;
    for (list<pair<float, KeyFrame *> >::iterator it : lScoreAndMatch) {
        KeyFrame *pKFi = it->second;
        vector<KeyFrame *> vpNeighs = pKFi->GetBestCovisibilityKeyFrames(10);
        float bestScore = it->first;
        float accScore = it->first;
        KeyFrame *pBestKF = pKFi;
        for (vector<KeyFrame *>::iterator vit = vpNeighs.begin(), vend = vpNeighs.end(); vit != vend; vit++) {
            KeyFrame *pKF2 = *vit;
            if (pKF2->mnLoopQuery == pKF->mnId && pKF2->mnLoopWords > minCommonWords) {
                accScore += pKF2->mLoopScore;
                if (pKF2->mLoopScore > bestScore) {
                    pBestKF = pKF2;
                    bestScore = pKF2->mLoopScore;
                }
            }
        }

        lAccScoreAndMatch.push_back(make_pair(accScore, pBestKF));
        if (accScore > bestAccScore)
            bestAccScore = accScore;
    }

    // step5. 取相似度得分高于最高相似度0.75的组的最优匹配关键帧
    float minScoreToRetain = 0.75f * bestAccScore;
    set<KeyFrame *> spAlreadyAddedKF;
    vector<KeyFrame *> vpLoopCandidates;
    for (list<pair<float, KeyFrame *> >::iterator it : lAccScoreAndMatch.begin()) {
        if (it->first > minScoreToRetain) {
            KeyFrame *pKFi = it->second;
            if (!spAlreadyAddedKF.count(pKFi)) {
                vpLoopCandidates.push_back(pKFi);
                spAlreadyAddedKF.insert(pKFi);
            }
        }
    }

    return vpLoopCandidates;
}
```

计算Sim3变换: `ComputeSim3()`

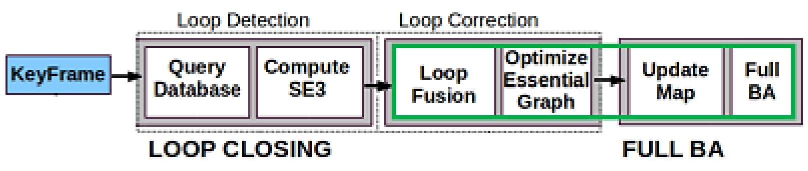


成员函数/变量	访问控制	意义
`std::vector<KeyFrame *> mvpEnoughConsistentCandidates`	`protected`	在函数`LoopClosing::DetectLoop()`中找到的有足够连续性的闭环关键帧
`g2o::Sim3 mg2oScw` `cv::Mat mScw`	`protected` `protected`	世界坐标系w到相机坐标系c的Sim3变换
`std::vector<MapPoint *> mvpLoopMapPoints`	`protected`	闭环关键帧组中的地图点
`std::vector<MapPoint *> mvpCurrentMatchedPoints`	`protected`	当前帧到`mvpLoopMapPoints`的匹配关系 `mvpCurrentMatchedPoints[i]`表示当前帧`i`个特征点对应的地图点



```cpp
bool LoopClosing::ComputeSim3() {
    const int nInitialCandidates = mvpEnoughConsistentCandidates.size();
    ORBmatcher matcher(0.75, true);

    vector<Sim3Solver *> vpSim3Solvers;                 // 保存每个闭环匹配关键帧的Sim3Solver
    vector<vector<MapPoint *> > vvpMapPointMatches;     // 保存当前关键帧到每个闭环匹配关键帧的匹配关系
    vector<bool> vbDiscarded;                           // 保存每个闭环匹配关键帧是否是误匹配

    // step1. 为每个有超过20个匹配点的闭环关键帧创建Sim3Solver
    int nCandidates = 0;
    for (int i = 0; i < nInitialCandidates; i++) {
        KeyFrame *pKF = mvpEnoughConsistentCandidates[i];
        pKF->SetNotErase();
        int nmatches = matcher.SearchByBoW(mpCurrentKF, pKF, vvpMapPointMatches[i]);
        if (nmatches < 20) {
            vbDiscarded[i] = true;
            continue;
        } else {
            Sim3Solver *pSolver = new Sim3Solver(mpCurrentKF, pKF, vvpMapPointMatches[i], mbFixScale);
            pSolver->SetRansacParameters(0.99, 20, 300);
            vpSim3Solvers[i] = pSolver;
        }
        nCandidates++;
    }

    // step2. 对每个闭环候选关键帧求解优化Sim3
    bool bMatch = false;         // 是否有帧通过Sim3求解
    while (nCandidates > 0 && !bMatch) {
        for (int i = 0; i < nInitialCandidates; i++) {
            if (vbDiscarded[i])
                continue;

            KeyFrame *pKF = mvpEnoughConsistentCandidates[i];
            vector<bool> vbInliers;
            int nInliers;
            bool bNoMore;
            // step2.1. 进行Sim3迭代求解
            Sim3Solver *pSolver = vpSim3Solvers[i];
            cv::Mat Scm = pSolver->iterate(5, bNoMore, vbInliers, nInliers);
            if (bNoMore) {
                vbDiscarded[i] = true;
                nCandidates--;
            }

            if (!Scm.empty()) {
                // step2.2 根据计算出的Sim3搜索匹配点
                vector<MapPoint *> vpMapPointMatches(vvpMapPointMatches[i].size(), static_cast<MapPoint
*>(NULL));
                for (size_t j = 0, jend = vbInliers.size(); j < jend; j++) {
                    if (vbInliers[j])
                        vpMapPointMatches[j] = vvpMapPointMatches[i][j];
                }
                cv::Mat R = pSolver->GetEstimatedRotation();
                cv::Mat t = pSolver->GetEstimatedTranslation();
                const float s = pSolver->GetEstimatedScale();
                matcher.SearchBySim3(mpCurrentKF, pKF, vpMapPointMatches, s, R, t, 7.5);

                // step2.3. 根据搜索出的匹配点优化Sim3
                g2o::Sim3 gScm(Converter::toMatrix3d(R), Converter::toVector3d(t), s);
                const int nInliers = Optimizer::OptimizeSim3(mpCurrentKF, pKF, vpMapPointMatches, gScm, 10, mbFixScale);
                if (nInliers >= 20) {
                    bMatch = true;
                    mpMatchedKF = pKF;
                    g2o::Sim3 gSmw(Converter::toMatrix3d(pKF->GetRotation()), Converter::toVector3d(pKF->GetTranslation()), 1.0);
                    mg2oScw = gScm * gSmw;
                    mScw = Converter::toCvMat(mg2oScw);
                    mvpCurrentMatchedPoints = vpMapPointMatches;
                    break;
                }
            }
        }
    }

    // step2.4. 优化失败,退出函数
    if (!bMatch) {
        for (int i = 0; i < nInitialCandidates; i++)
            mvpEnoughConsistentCandidates[i]->SetErase();
        mpCurrentKF->SetErase();
        return false;
    }

    // step3. 将闭环关键帧及其共视关键帧的所有地图点 投影到 当前关键帧
    vector<KeyFrame *> vpLoopConnectedKFs = mpMatchedKF->GetVectorCovisibleKeyFrames();
    vpLoopConnectedKFs.push_back(mpMatchedKF);
    for (KeyFrame *pKF : vpLoopConnectedKFs) {
        for (MapPoint *pMP : pKF->GetMapPointMatches()) {
            if (pMP && !pMP->isBad() && pMP->mnLoopPointForKF != mpCurrentKF->mnId) {
                mvpLoopMapPoints.push_back(pMP);
                pMP->mnLoopPointForKF = mpCurrentKF->mnId;
            }
        }
    }
    matcher.SearchByProjection(mpCurrentKF, mScw, mvpLoopMapPoints, mvpCurrentMatchedPoints, 10);

    // step5. 根据投影成功的地图点数判断Sim3计算的是否准确
    int nTotalMatches = 0;
    for (size_t i = 0; i < mvpCurrentMatchedPoints.size(); i++) {
        if (mvpCurrentMatchedPoints[i])
            nTotalMatches++;
    }

    if (nTotalMatches >= 40) {
        for (int i = 0; i < nInitialCandidates; i++)
            if (mvpEnoughConsistentCandidates[i] != mpMatchedKF)
                mvpEnoughConsistentCandidates[i]->SetErase();
        return true;
    } else {
        for (int i = 0; i < nInitialCandidates; i++)
            mvpEnoughConsistentCandidates[i]->SetErase();
        mpCurrentKF->SetErase();
        return false;
    }
}
```

闭环矫正: `CorrectLoop()`

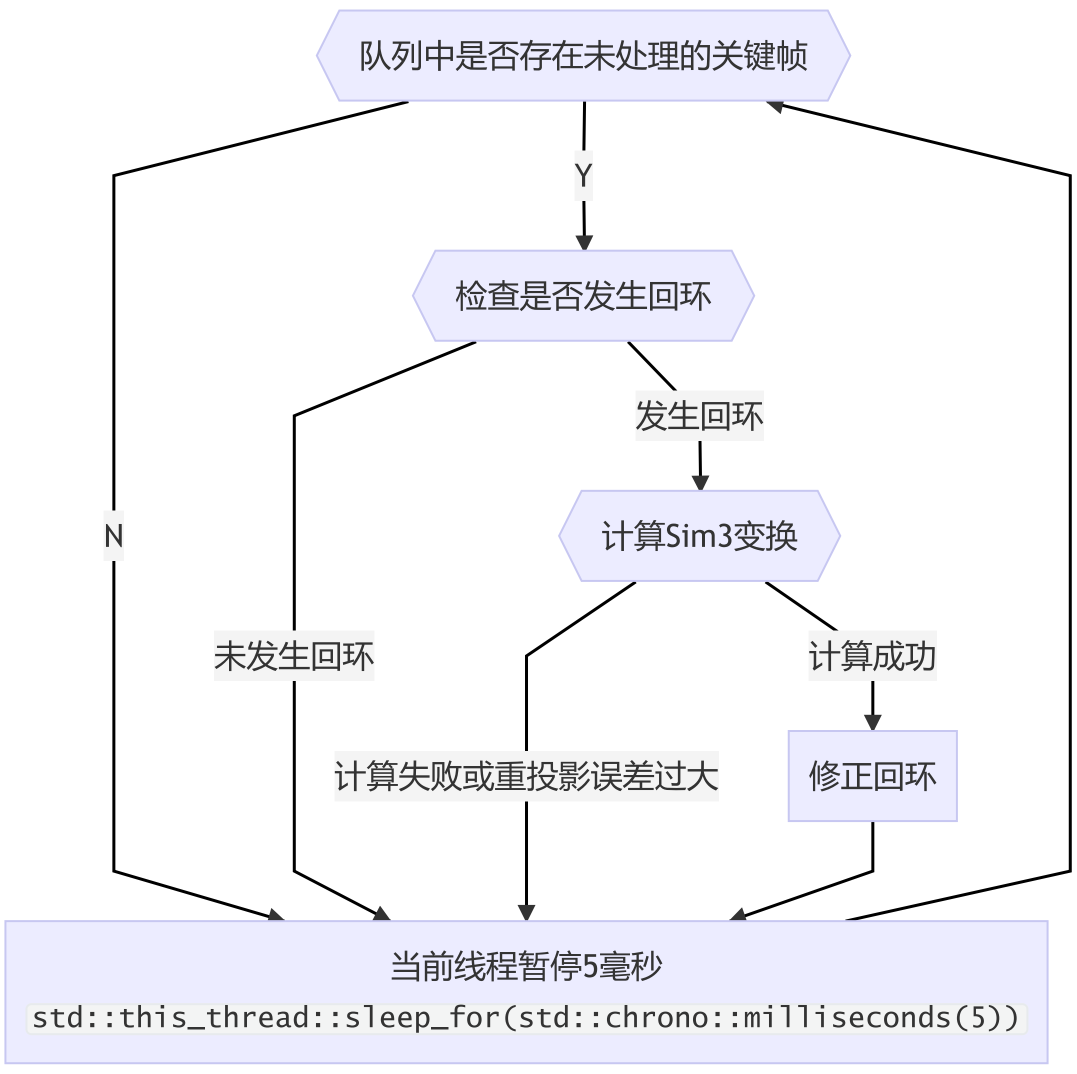


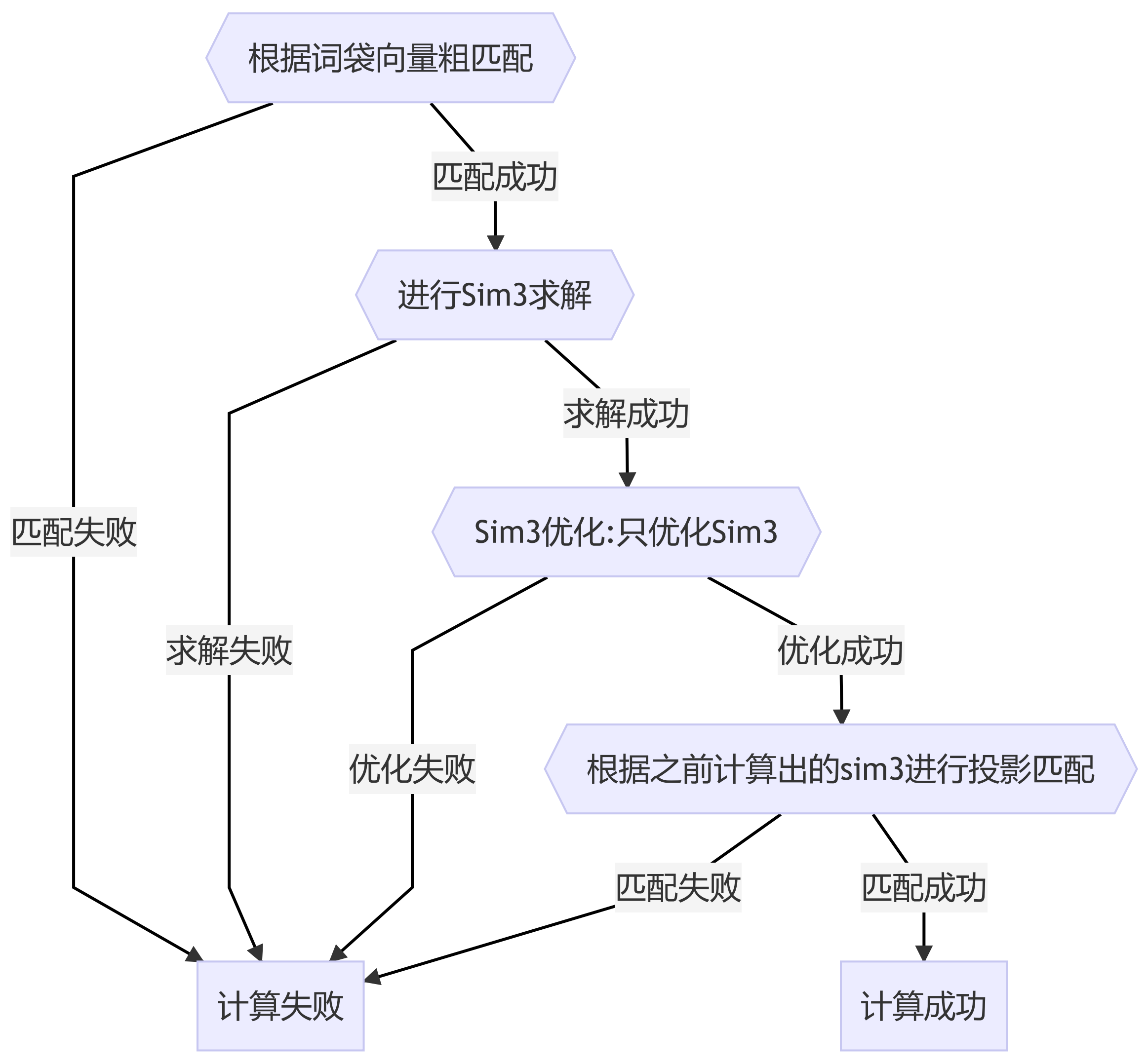
函数 `LoopClosing::CorrectLoop()` 的主要流程:

1. Sim3位姿传播:
   - 将Sim3位姿传播到**局部关键帧组**上.
   - 将Sim3位姿传播到**局部地图点**上.
2. 地图点融合:
   - 将**闭环关键帧组地图点**投影到**当前关键帧**上.
   - 将**闭环关键帧组地图点**投影到**局部关键帧组**上.
3. BA优化

- 本质图BA优化: 优化所有地图点和关键帧位姿,基于本质图.
- 全局BA优化: 优化所有地图点和关键帧位姿,基于地图点到关键帧的投影关系.

```cpp
void LoopClosing::CorrectLoop() {

    cout << "Loop detected!" << endl;

    // step0. 更新当前关键帧组与地图点的连接
    mpCurrentKF->UpdateConnections();

    // step1. Sim3位姿传播
    // step1.1. 构建局部关键帧组
    mvpCurrentConnectedKFs = mpCurrentKF->GetVectorCovisibleKeyFrames();
    mvpCurrentConnectedKFs.push_back(mpCurrentKF);
    map<KeyFrame*, g2o::Sim3> CorrectedSim3, NonCorrectedSim3;  // 存放局部关键帧组Sim3位姿传播前后的位姿
    CorrectedSim3[mpCurrentKF] = mg2oScw;
    cv::Mat Twc = mpCurrentKF->GetPoseInverse();

    {
        unique_lock<mutex> lock(mpMap->mMutexMapUpdate);

        // step1.2 将Sim3位姿传播到局部关键帧组中
        for (KeyFrame *pKFi : mvpCurrentConnectedKFs) {
            cv::Mat Tiw = pKFi->GetPose();
            if (pKFi != mpCurrentKF) {
                cv::Mat Tic = Tiw * Twc;
                cv::Mat Ric = Tic.rowRange(0, 3).colRange(0, 3);
                cv::Mat tic = Tic.rowRange(0, 3).col(3);
                g2o::Sim3 g2oSic(Converter::toMatrix3d(Ric), Converter::toVector3d(tic), 1.0);
                g2o::Sim3 g2oCorrectedSiw = g2oSic * mg2oScw;
                CorrectedSim3[pKFi] = g2oCorrectedSiw;
            }
            cv::Mat Riw = Tiw.rowRange(0, 3).colRange(0, 3);
            cv::Mat tiw = Tiw.rowRange(0, 3).col(3);
            g2o::Sim3 g2oSiw(Converter::toMatrix3d(Riw), Converter::toVector3d(tiw), 1.0);
            NonCorrectedSim3[pKFi] = g2oSiw;
        }

        // step1.3. 将Sim3位姿传播到局部地图点上
        for (pair<KeyFrame*, g2o::Sim3> mit : CorrectedSim3) {
            KeyFrame *pKFi = mit.first;
            g2o::Sim3 g2oCorrectedSiw = mit.second;
            g2o::Sim3 g2oCorrectedSwi = g2oCorrectedSiw.inverse();
            g2o::Sim3 g2oSiw = NonCorrectedSim3[pKFi];

            for (MapPoint *pMPi : pKFi->GetMapPointMatches()) {
                if (!pMPi || pMPi->isBad())
                    continue;
                if (pMPi->mnCorrectedByKF == mpCurrentKF->mnId)     // 标记,防止重复矫正
                    continue;
                cv::Mat P3Dw = pMPi->GetWorldPos();
                Eigen::Matrix<double, 3, 1> eigP3Dw = Converter::toVector3d(P3Dw);
                Eigen::Matrix<double, 3, 1> eigCorrectedP3Dw = g2oCorrectedSwi.map(g2oSiw.map(eigP3Dw));
                cv::Mat cvCorrectedP3Dw = Converter::toCvMat(eigCorrectedP3Dw);
                pMPi->SetWorldPos(cvCorrectedP3Dw);
                pMPi->mnCorrectedByKF = mpCurrentKF->mnId;
                pMPi->mnCorrectedReference = pKFi->mnId;
                pMPi->UpdateNormalAndDepth();
            }

            // 将更新后的Sim3位姿赋值给关键帧变量
            Eigen::Matrix3d eigR = g2oCorrectedSiw.rotation().toRotationMatrix();
            Eigen::Vector3d eigt = g2oCorrectedSiw.translation();
            double s = g2oCorrectedSiw.scale();
            eigt *= (1. / s);
            cv::Mat correctedTiw = Converter::toCvSE3(eigR, eigt);
            pKFi->SetPose(correctedTiw);
            pKFi->UpdateConnections();
        }

        // step2. 地图点融合
        // step2.1 将闭环关键帧组地图点融合到当前关键帧上
        for (size_t i = 0; i < mvpCurrentMatchedPoints.size(); i++) {
            if (mvpCurrentMatchedPoints[i]) {
                MapPoint *pLoopMP = mvpCurrentMatchedPoints[i];
                MapPoint *pCurMP = mpCurrentKF->GetMapPoint(i);
                if (pCurMP)
                    pCurMP->Replace(pLoopMP);       // 闭环关键帧组地图点在地图中的时间更长,位姿更准确
                else {
                    mpCurrentKF->AddMapPoint(pLoopMP, i);
                    pLoopMP->AddObservation(mpCurrentKF, i);
                    pLoopMP->ComputeDistinctiveDescriptors();
                }
            }
        }

    }

    // step2.2 将闭环关键帧组地图点融合到局部关键帧组上
    SearchAndFuse(CorrectedSim3);


    // step3. BA优化
    // step3.0. 查找回环连接边,用于和生成树共同组成本质图
    map<KeyFrame *, set<KeyFrame *> > LoopConnections;
    for (KeyFrame *pKFi : mvpCurrentConnectedKFs) {
        vector<KeyFrame *> vpPreviousNeighbors = pKFi->GetVectorCovisibleKeyFrames();
        pKFi->UpdateConnections();
        // 闭环矫正后的共视关系 - 闭环矫正前的共视关系 = 闭环带来的新共视关系
        for (KeyFrame* prev_KFi : vpPreviousNeighbors) {
            LoopConnections[pKFi].erase(*prev_KFi);
        }
        for (KeyFrame* prev_KFi : mvpCurrentConnectedKFs) {
            LoopConnections[pKFi].erase(prev_KFi);
        }
    }

    // step3.1. 本质图BA优化
    mpMatchedKF->AddLoopEdge(mpCurrentKF);
    mpCurrentKF->AddLoopEdge(mpMatchedKF);
    Optimizer::OptimizeEssentialGraph(mpMap, mpMatchedKF, mpCurrentKF, NonCorrectedSim3, CorrectedSim3,
LoopConnections, mbFixScale);

    // step3.2. 全局BA优化
    mbRunningGBA = true;
    mbFinishedGBA = false;
    mbStopGBA = false;
    mpThreadGBA = new thread(&LoopClosing::RunGlobalBundleAdjustment, this, mpCurrentKF->mnId);

    cout << "Loop Closed!" << endl;
    mLastLoopKFid = mpCurrentKF->mnId;
}
```





根据词袋向量粗匹配
匹配成功
进行Sim3求解
求解成功
Sim3优化:只优化Sim3
优化成功
根据之前计算出的sim3进行投影匹配
匹配失败
匹配成功
计算成功
匹配失败
求解失败
优化失败
计算失败