各成员函数/变量

共视图: `mConnectedKeyFrameWeights`

基于对地图点的观测重新构造共视图: `UpdateConnections()`

生成树: `mpParent`、`mspChildrens`

关键帧的删除

参与回环检测的关键帧具有不被删除的特权: `mbNotErase`

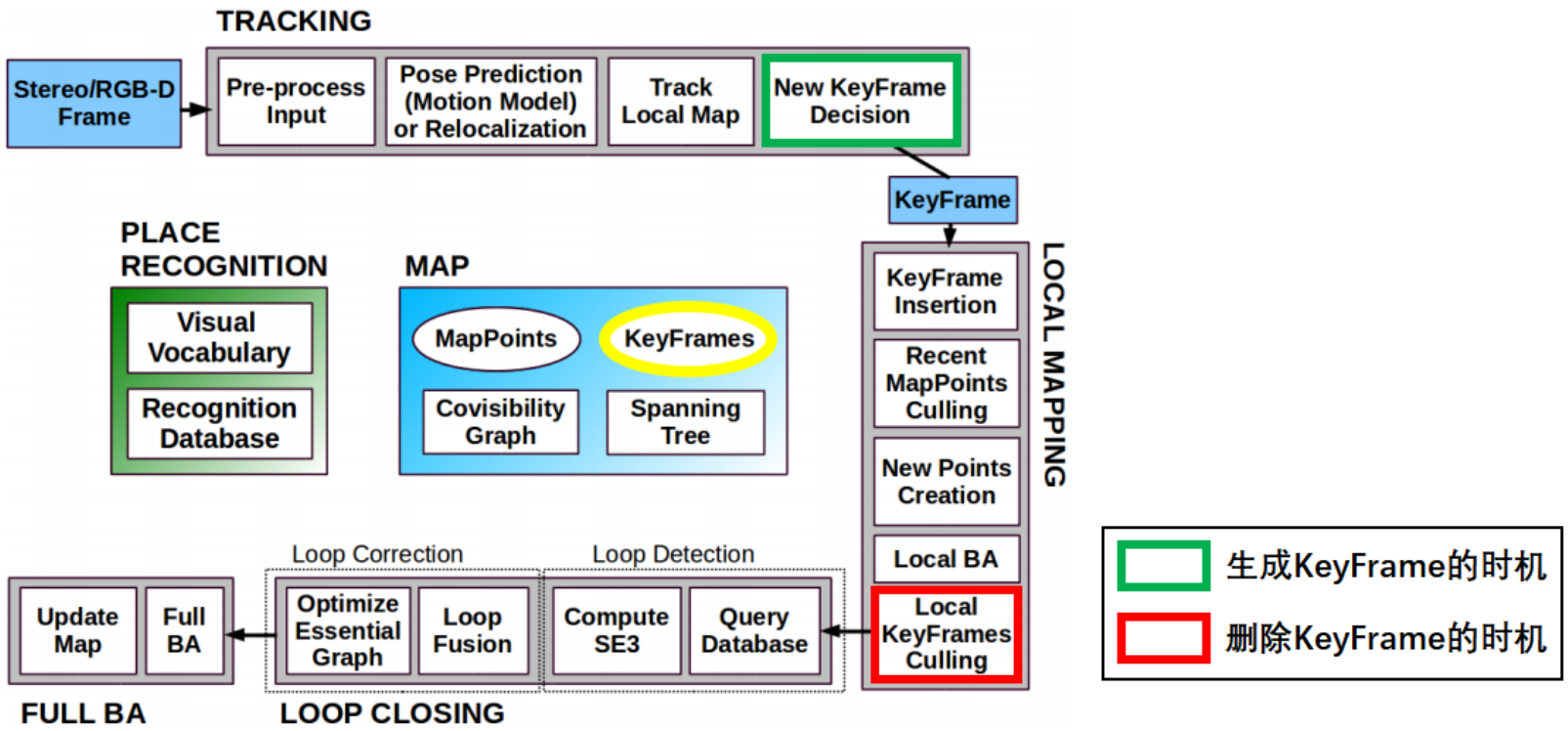
删除关键帧时维护共视图和生成树

对地图点的观测

回环检测与本质图

`KeyFrame` 的用途

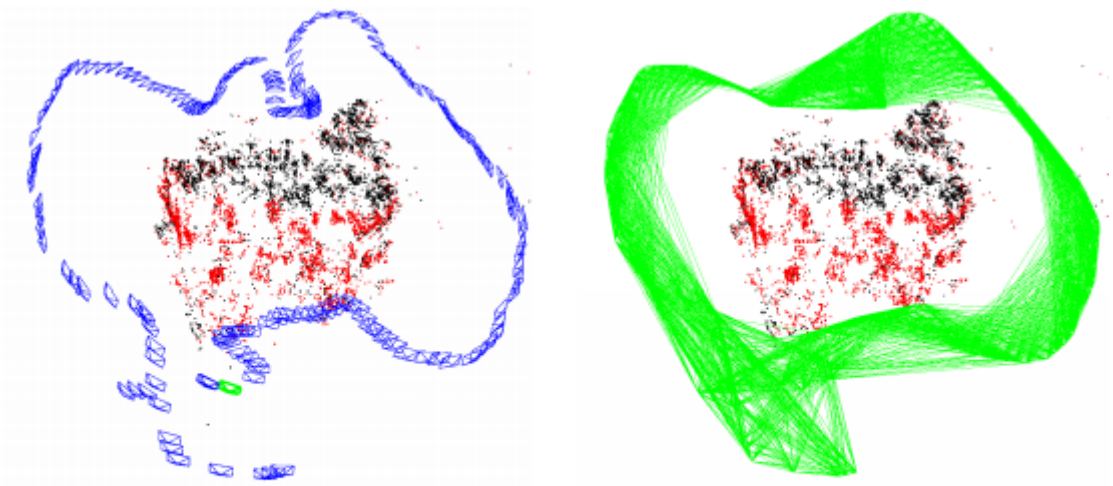
`KeyFrame` 类的生命周期



各成员函数/变量

共视图: `mConnectedKeyFrameWeights`

能看到同一地图点的两关键帧之间存在**共视关系**,共视地图点的数量被称为**权重**.



(a) KeyFrames (blue), Current Camera (green), MapPoints (black, red), Current Local MapPoints (red)

(b) Covisibility Graph

成员函数/变量	访问控制	意义
`std::map<KeyFrame*, int> mConnectedKeyFrameWeights`	`protected`	当前关键帧的共视关键帧及权重
`std::vector<KeyFrame*> mvpOrderedConnectedKeyFrames`	`protected`	所有共视关键帧,按权重从大到小排序
`std::vector<int> mvOrderedWeights`	`protected`	所有共视权重,按从大到小排序
`void UpdateConnections()`	`public`	基于当前关键帧对地图点的观测构造共视图
`void AddConnection(KeyFrame* pKF, int &weight)`	`public` 应为 `private`	添加共视关键帧
`void EraseConnection(KeyFrame* pKF)`	`public` 应为 `private`	删除共视关键帧
`void UpdateBestCovisibles()`	`public` 应为 `private`	基于共视图信息修改对应变量
`std::set<KeyFrame*> GetConnectedKeyFrames()`	`public`	get方法
`std::vector<KeyFrame*> GetVectorCovisibleKeyFrames()`	`public`	get方法
`std::vector<KeyFrame*> GetBestCovisibilityKeyFrames(int &N)`	`public`	get方法
`std::vector<KeyFrame*> GetCovisiblesByWeight(int &w)`	`public`	get方法
`int GetWeight(KeyFrame* pKF)`	`public`	get方法

共视图结构由3个成员变量维护:

- `mConnectedKeyFrameWeights`是一个`std::map`,**无序地**保存当前关键帧的**共视关键帧**及**权重**.
- `mvpOrderedConnectedKeyFrames`和`mvOrderedWeights`按**权重降序**分别保存当前关键帧的**共视关键帧**列表和**权重**列表.

基于对地图点的观测重新构造共视图: `UpdateConnections()`

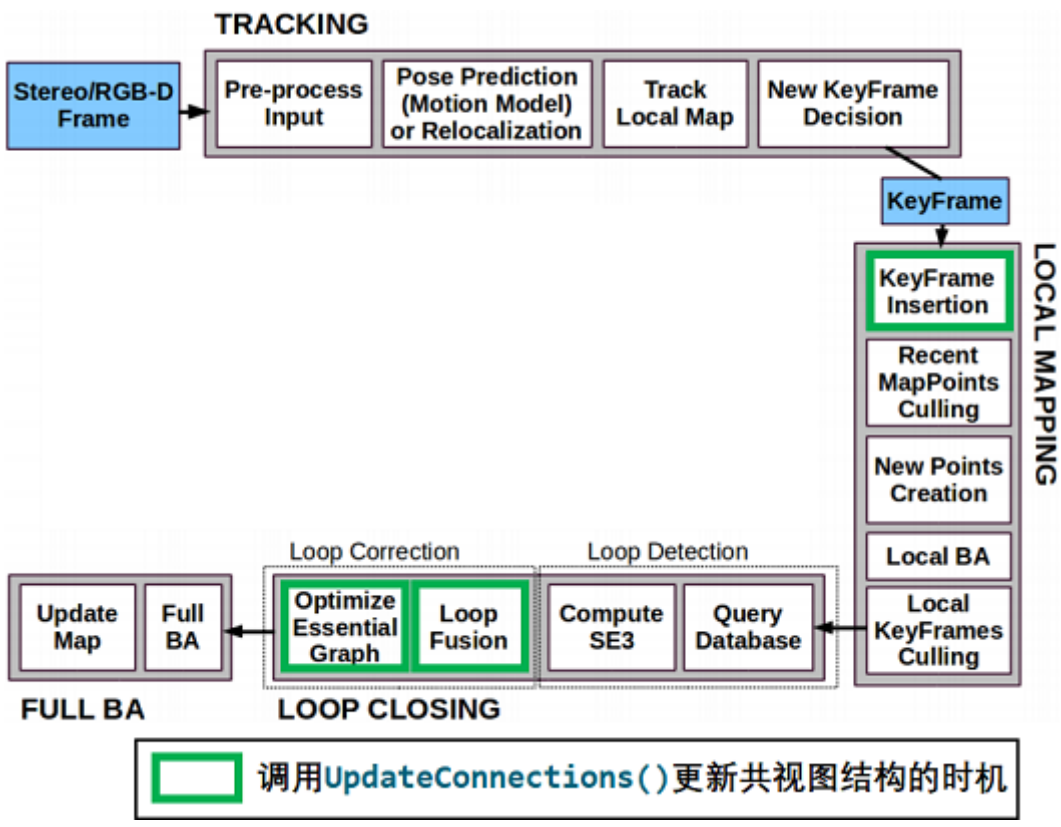
这3个变量由函数`KeyFrame::UpdateConnections()`进行初始化和维护,基于当前关键帧看到的地图点信息**重新生成**共视关键帧.

```
1  void KeyFrame::UpdateConnections() {
2
3      // 1. 通过遍历当前帧地图点获取其与其它关键帧的共视程度,存入变量KFcounter中
4      vector<MapPoint *> vpMP;
5      {
6          unique_lock<mutex> lockMPs(mMutexFeatures);
7          vpMP = mvpMapPoints;
8      }
9      map<KeyFrame *, int> KFcounter;
10     for (MapPoint *pMP : vpMP) {
11         map<KeyFrame *, size_t> observations = pMP->GetObservations();
12         for (map<KeyFrame *, size_t>::iterator mit = observations.begin(); mit != observations.end();
   mit++) {
13             if (mit->first->mnId == mnId)       // 与当前关键帧本身不算共视
14                 continue;
15             KFcounter[mit->first]++;
16         }
17     }
18
19     // step2. 找到与当前关键帧共视程度超过15的关键帧,存入变量vPairs中
20     vector<pair<int, KeyFrame *> > vPairs;
21     int th = 15;
22     int nmax = 0;
23     KeyFrame *pKFmax = NULL;
24     for (map<KeyFrame *, int>::iterator mit = KFcounter.begin(), mend = KFcounter.end(); mit != mend;
   mit++) {
25         if (mit->second > nmax) {
26             nmax = mit->second;
27             pKFmax = mit->first;
28         }
29         if (mit->second >= th) {
30             vPairs.push_back(make_pair(mit->second, mit->first));
31             (mit->first)->AddConnection(this, mit->second);             // 对超过阈值的共视边建立连接
32         }
33     }
34
35     //  step3. 对关键帧按照共视权重降序排序,存入变量mvpOrderedConnectedKeyFrames和mvOrderedWeights中
36     sort(vPairs.begin(), vPairs.end());
37     list<KeyFrame *> lKFs;
38     list<int> lWs;
39     for (size_t i = 0; i < vPairs.size(); i++) {
```

```
        lKFs.push_front(vPairs[i].second);
        lWs.push_front(vPairs[i].first);
    }
    {
        unique_lock<mutex> lockCon(mMutexConnections);
        mConnectedKeyFrameWeights = KFcounter;
        mvpOrderedConnectedKeyFrames = vector<KeyFrame *>(lKFs.begin(), lKFs.end());
        mvOrderedWeights = vector<int>(lWs.begin(), lWs.end());

        // step4. 对于第一次加入生成树的关键帧,取共视程度最高的关键帧为父关键帧
        if (mbFirstConnection && mnId != 0) {
            mpParent = mvpOrderedConnectedKeyFrames.front();
            mpParent->AddChild(this);
            mbFirstConnection = false;
        }
    }
}
```

只要**关键帧与地图点间的连接关系发生变化**(包括**关键帧创建**和**地图点重新匹配关键帧特征点**),函数`KeyFrame::UpdateConnections()`就会被调用.具体来说,函数`KeyFrame::UpdateConnections()`的调用时机包括:

- `Tracking`线程中初始化函数`Tracking::StereoInitialization()`或`Tracking::MonocularInitialization()`函数创建关键帧后会调用`KeyFrame::UpdateConnections()`初始化共视图信息.
- `LocalMapping`线程接受到新关键帧时会调用函数`LocalMapping::ProcessNewKeyFrame()`处理跟踪过程中加入的地图点,之后会调用`KeyFrame::UpdateConnections()`初始化共视图信息.(实际上这里处理的是`Tracking`线程中函数`Tracking::CreateNewKeyFrame()`创建的关键帧)
- `LocalMapping`线程处理完毕缓冲队列内所有关键帧后会调用`LocalMapping::SearchInNeighbors()`融合当前关键帧和共视关键帧间的重复地图点,之后会调用`KeyFrame::UpdateConnections()`更新共视图信息.
- `LoopClosing`线程闭环矫正函数`LoopClosing::CorrectLoop()`会多次调用`KeyFrame::UpdateConnections()`更新共视图信息.



函数`AddConnection(KeyFrame* pKF, const int &weight)`和`EraseConnection(KeyFrame* pKF)`先对变量`mConnectedKeyFrameWeights`进行修改,再调用函数`UpdateBestCovisibles()`修改变量`mvpOrderedConnectedKeyFrames`和`mvOrderedWeights`.

这3个函数都只在函数`KeyFrame::UpdateConnections()`内部被调用了,应该设为私有成员函数.

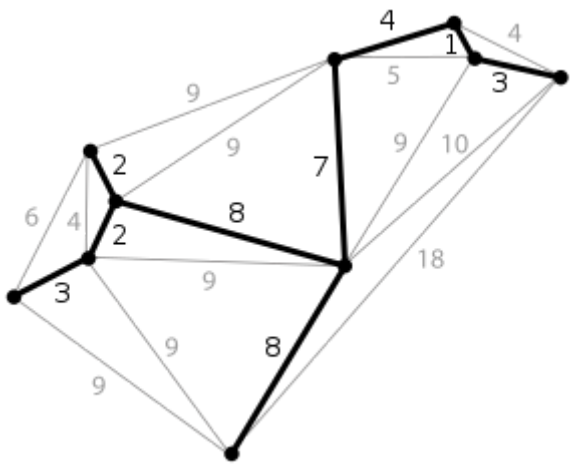
```
void KeyFrame::AddConnection(KeyFrame *pKF, const int &weight) {
    // step1. 修改变量mConnectedKeyFrameWeights
    {
        unique_lock<mutex> lock(mMutexConnections);

        if (!mConnectedKeyFrameWeights.count(pKF) || mConnectedKeyFrameWeights[pKF] != weight)
            mConnectedKeyFrameWeights[pKF] = weight;
        else
            return;
    }

    // step2. 调用函数UpdateBestCovisibles()修改变量mvpOrderedConnectedKeyFrames和mvOrderedWeights
    UpdateBestCovisibles();
}


void KeyFrame::EraseConnection(KeyFrame *pKF) {
    // step1. 修改变量mConnectedKeyFrameWeights
    bool bUpdate = false;
    {
        unique_lock<mutex> lock(mMutexConnections);
        if (mConnectedKeyFrameWeights.count(pKF)) {
            mConnectedKeyFrameWeights.erase(pKF);
```

```
            bUpdate = true;
        }
    }

    // step2. 调用函数UpdateBestCovisibles()修改变量mvpOrderedConnectedKeyFrames和mvOrderedWeights
    if (bUpdate)
        UpdateBestCovisibles();
}

void KeyFrame::UpdateBestCovisibles() {
    unique_lock<mutex> lock(mMutexConnections);

    // 取出所有关键帧进行排序,排序结果存入变量mvpOrderedConnectedKeyFrames和mvOrderedWeights中
    vector<pair<int, KeyFrame *> > vPairs;
    vPairs.reserve(mConnectedKeyFrameWeights.size());
    for (map<KeyFrame *, int>::iterator mit = mConnectedKeyFrameWeights.begin(), mend =
mConnectedKeyFrameWeights.end(); mit != mend; mit++)
        vPairs.push_back(make_pair(mit->second, mit->first));

    sort(vPairs.begin(), vPairs.end());
    list<KeyFrame *> lKFs;
    list<int> lWs;
    for (size_t i = 0, iend = vPairs.size(); i < iend; i++) {
        lKFs.push_front(vPairs[i].second);
        lWs.push_front(vPairs[i].first);
    }

    mvpOrderedConnectedKeyFrames = vector<KeyFrame *>(lKFs.begin(), lKFs.end());
    mvOrderedWeights = vector<int>(lWs.begin(), lWs.end());
}
```

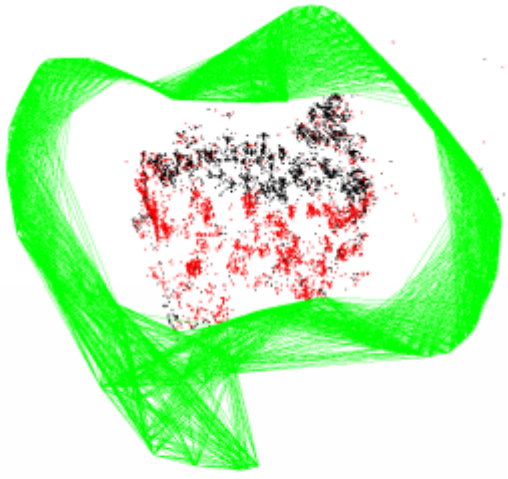
生成树: `mpParent`、`mspChildrens`

生成树是一种稀疏连接,以最小的边数保存图中所有节点.对于含有`N`个节点的图,只需构造一个`N-1`条边的最小生成树就可以将所有节点连接起来.

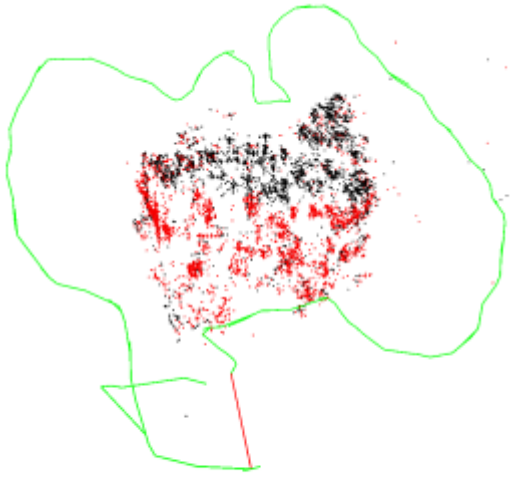
下图表示含有一个`10`个节点,`20`条边的稠密图;粗黑线代表其**最小生成树**,只需`9`条边即可将所有节点连接起来.



在ORB-SLAM2中,保存所有关键帧构成的最小生成树(优先选择权重大的边作为生成树的边),在回环闭合时只需对最小生成树做BA优化就能以最小代价优化所有关键帧和地图点的位姿,相比于优化共视图大大减少了计算量.(实际上并没有对最小生成树做BA优化,而是对包含生成树的本质图做BA优化)



(b) Covisibility Graph



(c) Spanning Tree (green) and Loop Closure (red)

成员函数/变量	访问控制	意义
`bool mbFirstConnection`	`protected`	当前关键帧是否还未加入到生成树 构造函数中初始化为`true`,加入生成树后置为`false`
`KeyFrame* mpParent`	`protected`	当前关键帧在生成树中的父节点
`std::set<KeyFrame*> mspChildrens`	`protected`	当前关键帧在生成树中的子节点列表
`KeyFrame* GetParent()`	`public`	`mpParent`的get方法
`void ChangeParent(KeyFrame* pKF)`	`public` 应为`private`	`mpParent`的set方法
`std::set<KeyFrame*> GetChilds()`	`public`	`mspChildrens`的get方法
`void AddChild(KeyFrame* pKF)`	`public` 应为`private`	添加子节点,`mspChildrens`的set方法
`void EraseChild(KeyFrame* pKF)`	`public` 应为`private`	删除子节点,`mspChildrens`的set方法
`bool hasChild(KeyFrame* pKF)`	`public`	判断`mspChildrens`是否为空

生成树结构由成员变量`mpParent`和`mspChildrens`维护.我们主要关注生成树结构发生改变的时机.

- 关键帧增加到生成树中的时机:

  成功创建关键帧之后会调用函数`KeyFrame::UpdateConnections()`,该函数第一次被调用时会将该新关键帧加入到生成树中.

  新关键帧的父关键帧会被设为其**共视程度最高的共视关键帧**.

```
void KeyFrame::UpdateConnections() {

    // 更新共视图信息
    // ...

    // 更新关键帧信息: 对于第一次加入生成树的关键帧,取共视程度最高的关键帧为父关键帧
    // 该操作会改变当前关键帧的成员变量mpParent和父关键帧的成员变量mspChildrens
    unique_lock<mutex> lockCon(mMutexConnections);
    if (mbFirstConnection && mnId != 0) {
        mpParent = mvpOrderedConnectedKeyFrames.front();
        mpParent->AddChild(this);
        mbFirstConnection = false;
    }
}
```

- 共视图的改变(除了删除关键帧以外)不会引发生成树的改变.
- 只有当某个关键帧删除时,与其相连的生成树结构在会发生改变.(因为生成树是个单线联系的结构,没有冗余,一旦某关键帧删除了就得更新树结构才能保证**所有**关键帧依旧相连).生成树结构改变的方式类似于最小生成树算法中的加边法,见后文对函数`setbadflag()`的分析.

关键帧的删除

成员函数/变量	访问控制	意义	初值
`bool mbBad`	`protected`	标记是坏帧	`false`
`bool isBad()`	`public`	`mbBad`的get方法	
`void SetBadFlag()`	`public`	真的执行删除	
`bool mbNotErase`	`protected`	当前关键帧是否具有不被删除的特权	`false`
`bool mbToBeErased`	`protected`	当前关键帧是否曾被豁免过删除	`false`
`void SetNotErase()`	`public`	`mbNotErase`的set方法	
`void SetErase()`	`public`		

与`MapPoint`类似,函数`KeyFrame::SetBadFlag()`对`KeyFrame`的删除过程也采取**先标记再清除**的方式: 先将坏帧标记`mBad`置为`true`,再依次处理其各成员变量.

参与回环检测的关键帧具有不被删除的特权: `mbNotErase`

参与回环检测的关键帧具有不被删除的特权,该特权由成员变量`mbNotErase`存储,创建`KeyFrame`对象时该成员变量默认被初始化为`false`.

若某关键帧参与了回环检测,`LoopClosing`线程就会就调用函数`KeyFrame::SetNotErase()`将该关键帧的成员变量`mbNotErase`设为`true`,标记该关键帧暂时不要被删除.

```
void KeyFrame::SetNotErase() {
    unique_lock<mutex> lock(mMutexConnections);
    mbNotErase = true;
}
```

在删除函数`SetBadFlag()`起始先根据成员变量`mbNotErase`判断当前`KeyFrame`是否具有豁免删除的特权.若当前`KeyFrame`的`mbNotErase`为`true`,则函数`SetBadFlag()`不能删除当前`KeyFrame`,但会将其成员变量`mbToBeErased`置为`true`.

```
void KeyFrame::SetBadFlag() {
    // step1. 特殊情况:豁免 第一帧 和 具有mbNotErase特权的帧
    {
        unique_lock<mutex> lock(mMutexConnections);

        if (mnId == 0)
            return;
        else if (mbNotErase) {
            mbToBeErased = true;
            return;
        }
    }

    // 两步删除: 先逻辑删除,再物理删除...
}
```

成员变量`mbToBeErased`标记当前`KeyFrame`是否被豁免过删除特权.`LoopClosing`线程不再需要某关键帧时,会调用函数`KeyFrame::SetErase()`剥夺该关键帧不被删除的特权,将成员变量`mbNotErase`复位为`false`;同时检查成员变量`mbToBeErased`,若`mbToBeErased`为`true`就会调用函数`KeyFrame::SetBadFlag()`删除该关键帧.

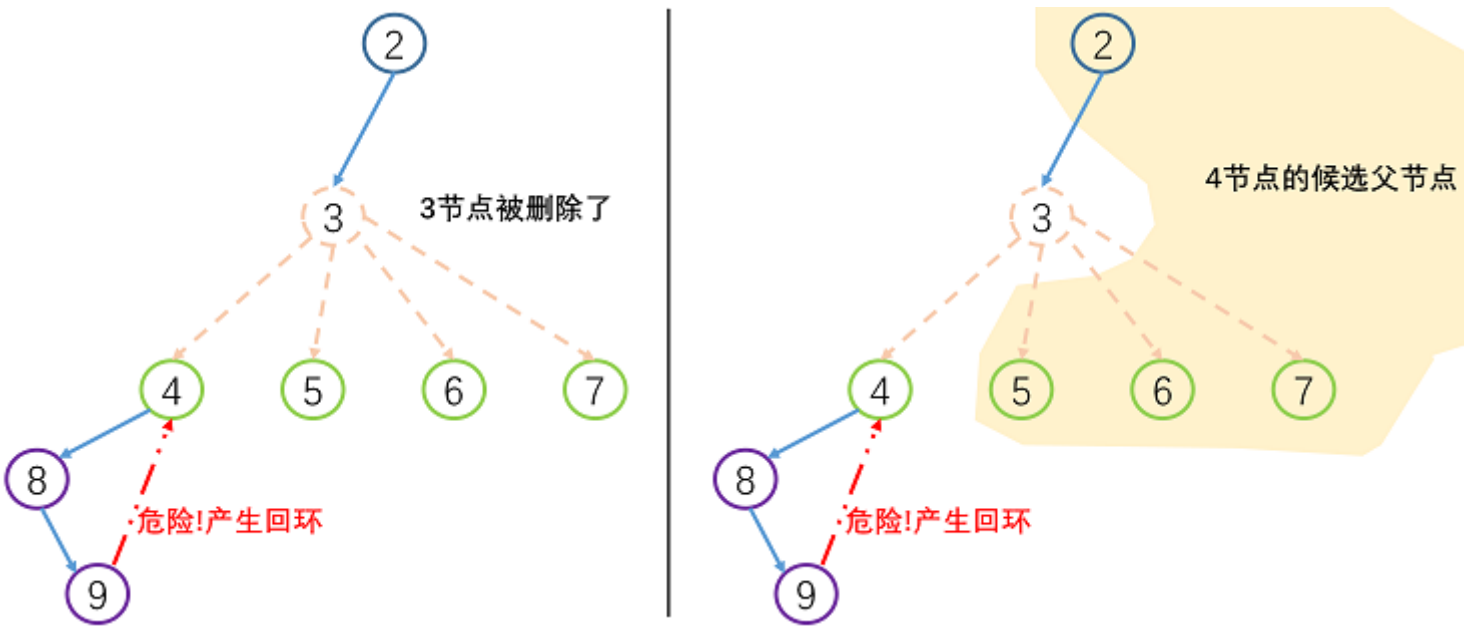
```
void KeyFrame::SetErase() {
    {
        unique_lock<mutex> lock(mMutexConnections);
        // 若当前关键帧没参与回环检测,但其它帧与当前关键帧形成回环关系,也不应当删除当前关键帧
        if (mspLoopEdges.empty()) {
            mbNotErase = false;
        }
    }

    // mbToBeErased: 删除之前记录的想要删但时机不合适没有删除的帧
    if (mbToBeErased) {
        SetBadFlag();
    }
}
```

删除关键帧时维护共视图和生成树

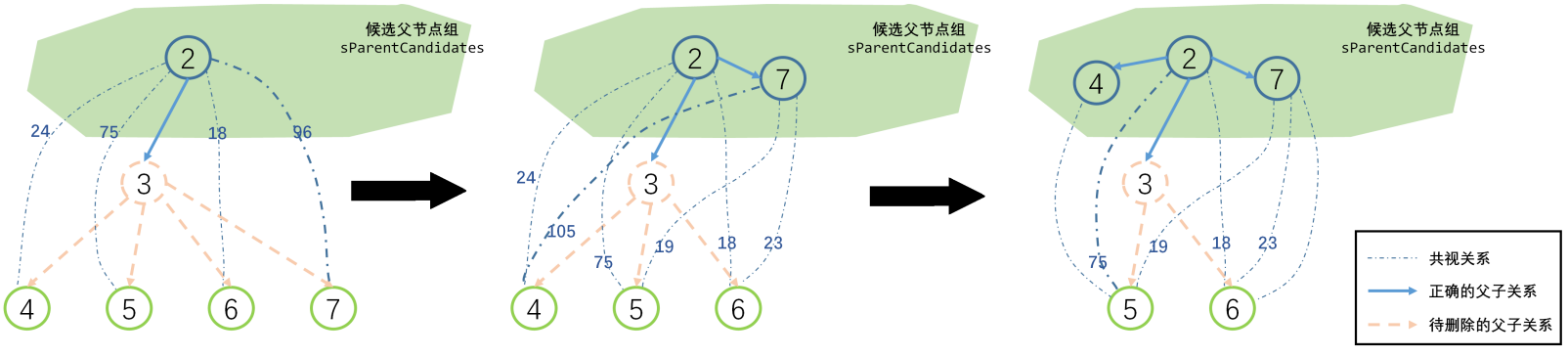
函数`SetBadFlag()`在删除关键帧的时维护其**共视图**和**生成树**结构.共视图结构的维护比较简单,这里主要关心如何维护生成树的结构.

当一个关键帧被删除时,其**父关键帧**和**所有子关键帧**的生成树信息也会受到影响,需要为其所有子关键帧寻找新的父关键帧,如果父关键帧找的不好的话,就会产生回环,导致生成树就断开.

被删除关键帧的子关键帧**所有可能的父关键帧**包括其兄弟关键帧和其被删除关键帧的父关键帧.以下图为例,关键帧`4`可能的父关键帧包括关键帧`3`、`5`、`6`和`7`.



采用类似于最小生成树算法中的加边法重新构建生成树结构: 每次循环取权重最高的候选边建立父子连接关系,并将新加入生成树的子节点到加入候选父节点集合`sParentCandidates`中.




```
void KeyFrame::SetBadFlag() {
    // step1. 特殊情况:豁免 第一帧 和 具有mbNotErase特权的帧
    {
        unique_lock<mutex> lock(mMutexConnections);
```

```
        if (mnId == 0)
            return;
        else if (mbNotErase) {
            mbToBeErased = true;
            return;
        }
    }

    // step2. 从共视关键帧的共视图中删除本关键帧
    for (auto mit : mConnectedKeyFrameWeights)
        mit.first->EraseConnection(this);

    // step3. 删除当前关键帧中地图点对本帧的观测
    for (size_t i = 0; i < mvpMapPoints.size(); i++)
        if (mvpMapPoints[i])
            mvpMapPoints[i]->EraseObservation(this);

    {
        // step4. 删除共视图
        unique_lock<mutex> lock(mMutexConnections);
        unique_lock<mutex> lock1(mMutexFeatures);
        mConnectedKeyFrameWeights.clear();
        mvpOrderedConnectedKeyFrames.clear();

        // step5. 更新生成树结构
        set<KeyFrame *> sParentCandidates;
        sParentCandidates.insert(mpParent);

        while (!mspChildrens.empty()) {
            bool bContinue = false;
            int max = -1;
            KeyFrame *pC;
            KeyFrame *pP;
            for (KeyFrame *pKF : mspChildrens) {
                if (pKF->isBad())
                    continue;

                vector<KeyFrame *> vpConnected = pKF->GetVectorCovisibleKeyFrames();

                for (size_t i = 0, iend = vpConnected.size(); i < iend; i++) {
                    for (set<KeyFrame *>::iterator spcit = sParentCandidates.begin(), spcend =
sParentCandidates.end();
                         spcit != spcend; spcit++) {
                        if (vpConnected[i]->mnId == (*spcit)->mnId) {
                            int w = pKF->GetWeight(vpConnected[i]);
                            if (w > max) {
                                pC = pKF;
                                pP = vpConnected[i];
                                max = w;
                                bContinue = true;
                            }
                        }
                    }
                }
            }

            if (bContinue) {
                pC->ChangeParent(pP);
                sParentCandidates.insert(pC);
                mspChildrens.erase(pC);
            } else
                break;
        }

        if (!mspChildrens.empty())
            for (set<KeyFrame *>::iterator sit = mspChildrens.begin(); sit != mspChildrens.end(); sit++)
{
                (*sit)->ChangeParent(mpParent);
            }

        mpParent->EraseChild(this);
        mTcp = Tcw * mpParent->GetPoseInverse();
        // step6. 将当前关键帧的 mbBad 置为 true
        mbBad = true;
    }

    // step7. 从地图中删除当前关键帧
    mpMap->EraseKeyFrame(this);
    mpKeyFrameDB->erase(this);
}
```

对地图点的观测

`KeyFrame`类除了像一般的`Frame`类那样保存二维图像特征点以外,还保存三维地图点`MapPoint`信息.

关键帧观测到的地图点列表由成员变量`mvpMapPoints`保存,下面是一些对该成员变量进行增删改查的成员函数,,就是简单的列表操作,没什么值得说的地方.

成员函数/变量	访问控制	意义
`std::vector<MapPoint*> mvpMapPoints`	`protected`	当前关键帧观测到的地图点列表
`void AddMapPoint(MapPoint* pMP, const size_t &idx)`	`public`	
`void EraseMapPointMatch(const size_t &idx)`	`public`	
`void EraseMapPointMatch(MapPoint* pMP)`	`public`	
`void ReplaceMapPointMatch(const size_t &idx, MapPoint* pMP)`	`public`	
`std::set<MapPoint*> GetMapPoints()`	`public`	
`std::vector<MapPoint*> GetMapPointMatches()`	`public`	
`int TrackedMapPoints(const int &minObs)`	`public`	
`MapPoint* GetMapPoint(const size_t &idx)`	`public`	

值得关心的是上述函数的调用时机,也就是说参考帧何时与地图点发生关系:

- 关键帧增加对地图点观测的时机:
  1. `Tracking`线程和`LocalMapping`线程创建新地图点后,会马上调用函数`KeyFrame::AddMapPoint()`添加当前关键帧对该地图点的观测.
  2. `LocalMapping`线程处理完毕缓冲队列内所有关键帧后会调用`LocalMapping::SearchInNeighbors()`融合当前关键帧和共视关键帧间的重复地图点,其中调用函数`ORBmatcher::Fuse()`实现融合过程中会调用函数`KeyFrame::AddMapPoint()`.
  3. `LoopClosing`线程闭环矫正函数`LoopClosing::CorrectLoop()`将闭环关键帧与其匹配关键帧间的地图进行融合,会调用函数`KeyFrame::AddMapPoint()`.
- 关键帧替换和删除对地图点观测的时机:
  1. `MapPoint`删除函数`MapPoint::SetBadFlag()`或替换函数`MapPoint::Replace()`会调用`KeyFrame::EraseMapPointMatch()`和`KeyFrame::ReplaceMapPointMatch()`删除和替换关键针对地图点的观测.
  2. `LocalMapping`线程调用进行局部BA优化的函数`Optimizer::LocalBundleAdjustment()`内部调用函数`KeyFrame::EraseMapPointMatch()`删除对重投影误差较大的地图点的观测.

回环检测与本质图

成员函数/变量	访问控制	意义
`std::set<KeyFrame*> mspLoopEdge`	`protected`	和当前帧形成回环的关键帧集合
`set<KeyFrame *> GetLoopEdges()`	`public`	`mspLoopEdge`的get函数
`void AddLoopEdge(KeyFrame *pKF)`	`public`	`mspLoopEdge`的set函数

`LoopClosing`线程中回环矫正函数`LoopClosing::CorrectLoop()`在调用本质图BA优化函数`Optimizer::OptimizeEssentialGraph()`之前会调用函数`KeyFrame::AddLoopEdge()`,在当前关键帧和其闭环匹配关键帧间添加回环关系.

在调用本质图BA优化函数`Optimizer::OptimizeEssentialGraph()`中会调用函数`KeyFrame::GetLoopEdges()`将所有闭环关系加入到本质图中进行优化.

`KeyFrame`的用途

`KeyFrame`类的生命周期

- `KeyFrame` 的创建:

  `Tracking` 线程中通过函数 `Tracking::NeedNewKeyFrame()` 判断是否需要关键帧,若需要关键帧,则调用函数 `Tracking::CreateNewKeyFrame()` 创建关键帧.

- `KeyFrame` 的销毁:

  `LocalMapping` 线程剔除冗余关键帧函数 `LocalMapping::KeyFrameCulling()` 中若检查到某关键帧为冗余关键帧,则调用函数 `KeyFrame::SetBadFlag()` 删除关键帧.