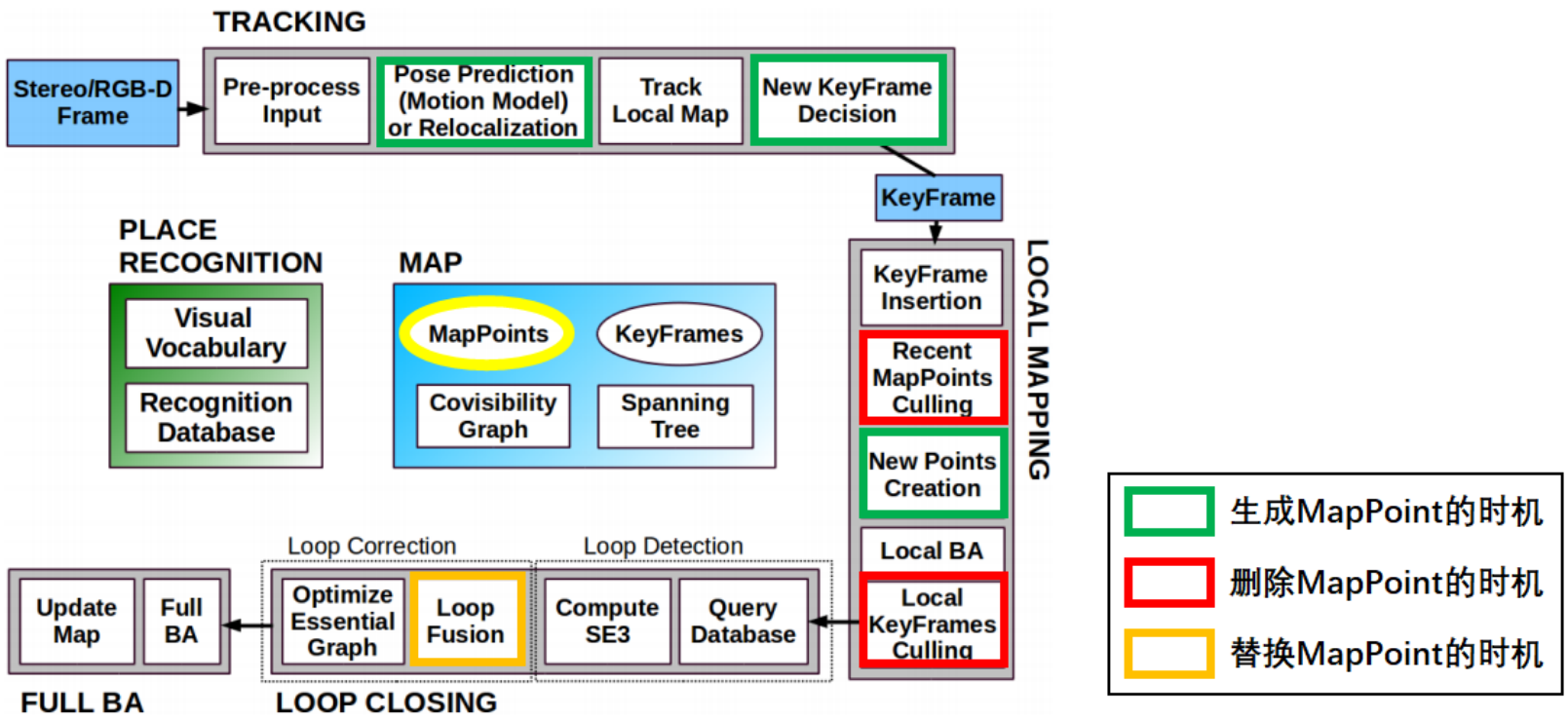


各成员函数/变量

地图点的世界坐标: `mWorldPos`

成员函数/变量	访问控制	意义
`cv::Mat mWorldPos`	`protected`	地图点的世界坐标
`cv::Mat GetWorldPos()`	`public`	`mWorldPos` 的get方法
`void SetWorldPos(const cv::Mat &Pos)`	`public`	`mWorldPos` 的set方法
`std::mutex mMutexPos`	`protected`	`mWorldPos` 的锁

与关键帧的观测关系: `mObservations`

成员函数/变量	访问控制	意义
`std::map<KeyFrame*, size_t> mObservations`	`protected`	当前地图点在某 `KeyFrame` 中的索引
`map<KeyFrame*, size_t> GetObservations()`	`public`	`mObservations` 的get方法
`void AddObservation(KeyFrame* pKF,size_t idx)`	`public`	添加当前地图点对某 `KeyFrame` 的观测
`void EraseObservation(KeyFrame* pKF)`	`public`	删除当前地图点对某 `KeyFrame` 的观测
`bool IsInKeyFrame(KeyFrame* pKF)`	`public`	查询当前地图点是否在某 `KeyFrame` 中
`int GetIndexInKeyFrame(KeyFrame* pKF)`	`public`	查询当前地图点在某 `KeyFrame` 中的索引
`int nObs`	`public`	记录当前地图点被多少相机观测到 单目帧每次观测加 `1`,双目帧每次观测加 `2`
`int Observations()`	`public`	`nObs` 的get方法

成员变量 `std::map<KeyFrame*,size_t> mObservations` 保存了当前关键点对关键帧 `KeyFrame` 的观测关系, `std::map` 是一个 `key-value` 结构,其 `key` 为某个关键帧, `value` 为当前地图点在该关键帧中的索引(是在该关键帧成员变量 `std::vector<MapPoint*> mvpMapPoints` 中的索引).

成员 `int nObs` 记录了当前地图点被多少个关键帧相机观测到了(单目关键帧每次观测算 `1` 个相机,双目/RGBD帧每次观测算 `2` 个相机).

- 函数 `AddObservation()` 和 `EraseObservation()` 同时维护 `mObservations` 和 `nObs`

```
1  // 向参考帧pKF中添加对本地图点的观测,本地图点在pKF中的编号为idx
```

```cpp
void MapPoint::AddObservation(KeyFrame* pKF, size_t idx) {
    unique_lock<mutex> lock(mMutexFeatures);
    // 如果已经添加过观测，返回
    if(mObservations.count(pKF))
        return;
    // 如果没有添加过观测，记录下能观测到该MapPoint的KF和该MapPoint在KF中的索引
    mObservations[pKF]=idx;

    // 根据观测形式是单目还是双目更新观测计数变量nObs
    if(pKF->mvuRight[idx]>=0)
        nObs += 2;
    else
        nObs++;
}
```

```cpp
// 从参考帧pKF中移除本地图点
void MapPoint::EraseObservation(KeyFrame* pKF)  {
    bool bBad=false;
    {
        unique_lock<mutex> lock(mMutexFeatures);
        // 查找这个要删除的观测,根据单目和双目类型的不同从其中删除当前地图点的被观测次数
        if(mObservations.count(pKF)) {
            if(pKF->mvuRight[mObservations[pKF]]>=0)
                nObs-=2;
            else
                nObs--;

            mObservations.erase(pKF);

            // 如果该keyFrame是参考帧，该Frame被删除后重新指定RefFrame
            if(mpRefKF == pKF)
                mpRefKF = mObservations.begin()->first;     // ????参考帧指定得这么草率真的好么?

            // 当观测到该点的相机数目少于2时，丢弃该点(至少需要两个观测才能三角化)
            if(nObs<=2)
                bBad=true;
        }
    }

    if(bBad)
        // 告知可以观测到该MapPoint的Frame，该MapPoint已被删除
        SetBadFlag();
}
```

- 函数`GetIndexInKeyFrame()`和`IsInKeyFrame()`就是对`mObservations`的简单查询

```cpp
int MapPoint::GetIndexInKeyFrame(KeyFrame *pKF) {
    unique_lock<mutex> lock(mMutexFeatures);
    if(mObservations.count(pKF))
        return mObservations[pKF];
    else
        return -1;
}

bool MapPoint::IsInKeyFrame(KeyFrame *pKF) {
    unique_lock<mutex> lock(mMutexFeatures);
    return (mObservations.count(pKF));
}
```

观测尺度

成员函数/变量	访问控制	意义
`cv::Mat mNormalVector`	`protected`	平均观测方向
`float mfMinDistance`	`protected`	平均观测距离的下限
`float mfMaxDistance`	`protected`	平均观测距离的上限
`cv::Mat GetNormal()`	`public`	`mNormalVector`的get方法
`float GetMinDistanceInvariance()`	`public`	`mfMinDistance`的get方法
`float GetMaxDistanceInvariance()`	`public`	`mNormalVector`的get方法
`void UpdateNormalAndDepth()`	`public`	更新平均观测距离和方向
`int PredictScale(const float &currentDist, KeyFrame* pKF)` `int PredictScale(const float &currentDist, Frame* pF)`	`public` `public`	估计当前地图点在某`Frame`中对应特征点的金字塔层级
`KeyFrame* mpRefKF`	`protected`	当前地图点的参考关键帧
`KeyFrame* GetReferenceKeyFrame()`	`public`	`mpRefKF`的get方法

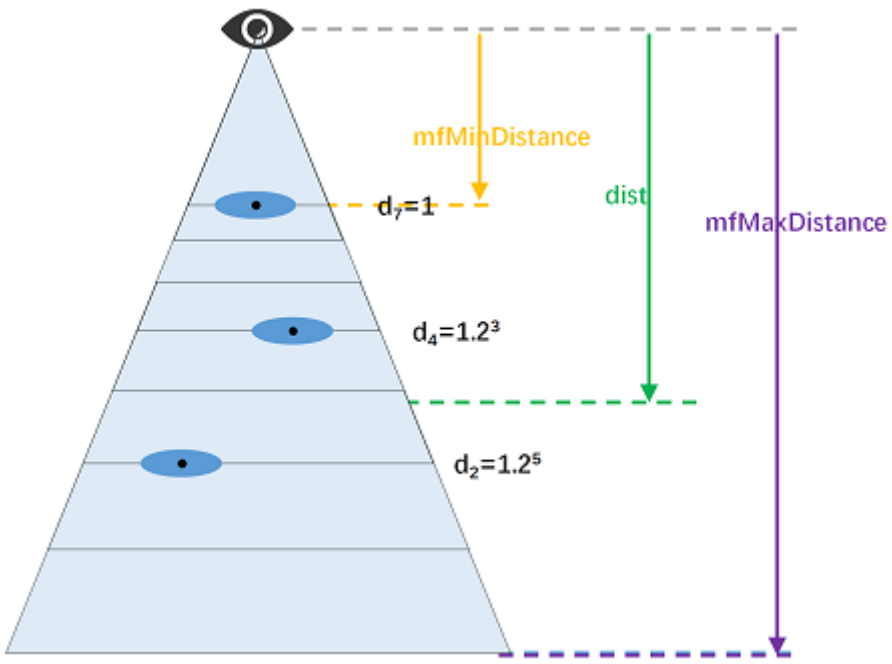
平均观测距离: `mfMinDistance`和`mfMaxDistance`

特征点的**观测距离**与其**在图像金字塔中的图层**呈线性关系.直观上理解,如果一个图像区域被放大后才能识别出来,说明该区域的观测深度较深.

特征点的平均观测距离的上下限由成员变量`mfMaxDistance`和`mfMinDistance`表示:

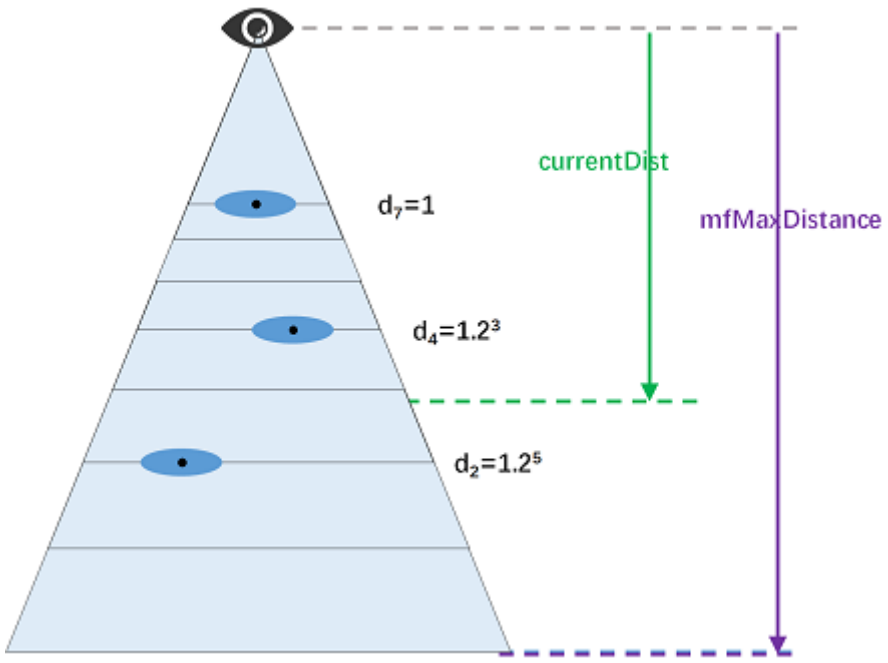
- `mfMaxDistance`表示若**地图点**匹配在某**特征提取器图像金字塔第`7`层上的某特征点**,观测距离值
- `mfMinDistance`表示若**地图点**匹配在某**特征提取器图像金字塔第`0`层上的某特征点**,观测距离值

这两个变量是基于地图点在其参考关键帧上的观测得到的.




```
// pFrame是当前MapPoint的参考帧
const int level = pFrame->mvKeysUn[idxF].octave;
const float levelScaleFactor = pFrame->mvScaleFactors[level];
const int nLevels = pFrame->mnScaleLevels;
mfMaxDistance = dist*levelScaleFactor;
mfMinDistance = mfMaxDistance/pFrame->mvScaleFactors[nLevels-1];
```

函数`int PredictScale(const float &currentDist, KeyFrame* pKF)`和`int PredictScale(const float &currentDist, Frame* pF)`根据某地图点到某帧的观测深度估计其在该帧图片上的层级,是上述过程的逆运算.



$$\frac{currentDist}{mfMaxDistance} = 1.2^{level}$$

$$level = \lceil log_{1.2}(\frac{currentDist}{mfMaxDistance}) \rceil$$

```cpp
int MapPoint::PredictScale(const float &currentDist, KeyFrame* pKF) {
    float ratio;
    {
        unique_lock<mutex> lock(mMutexPos);
        ratio = mfMaxDistance/currentDist;
    }

    int nScale = ceil(log(ratio)/pKF->mfLogScaleFactor);
    if(nScale<0)
        nScale = 0;
    else if(nScale>=pKF->mnScaleLevels)
        nScale = pKF->mnScaleLevels-1;

    return nScale;
}
```

更新平均观测方向和距离: `UpdateNormalAndDepth()`

函数`UpdateNormalAndDepth()`更新当前地图点的**平均观测方向和距离**,其中平均观测方向是根据`mObservations`中**所有观测到本地图点的关键帧**取平均得到的;平均观测距离是根据**参考关键帧**得到的.

```cpp
void MapPoint::UpdateNormalAndDepth() {
    // step1. 获取地图点相关信息
    map<KeyFrame *, size_t> observations;
    KeyFrame *pRefKF;
    cv::Mat Pos;
    {
        unique_lock<mutex> lock1(mMutexFeatures);
        unique_lock<mutex> lock2(mMutexPos);

        observations = mObservations;
        pRefKF = mpRefKF;
        Pos = mWorldPos.clone();
    }

    // step2. 根据观测到但钱地图点的关键帧取平均计算平均观测方向
    cv::Mat normal = cv::Mat::zeros(3, 1, CV_32F);
    int n = 0;
    for (KeyFrame *pKF : observations.begin()) {
        normal = normal + normali / cv::norm(mWorldPos - pKF->GetCameraCenter());
        n++;
    }

    // step3. 根据参考帧计算平均观测距离
    cv::Mat PC = Pos - pRefKF->GetCameraCenter();
    const float dist = cv::norm(PC);
    const int level = pRefKF->mvKeysUn[observations[pRefKF]].octave;
    const float levelScaleFactor = pRefKF->mvScaleFactors[level];
    const int nLevels = pRefKF->mnScaleLevels;

    {
        unique_lock<mutex> lock3(mMutexPos);
        mfMaxDistance = dist * levelScaleFactor;
        mfMinDistance = mfMaxDistance / pRefKF->mvScaleFactors[nLevels - 1];
        mNormalVector = normal / n;
    }
}
```

地图点的平均观测距离是根据其**参考关键帧**计算的,那么参考关键帧`KeyFrame* mpRefKF`是如何指定的呢?

- 构造函数中,创建该地图点的参考帧被设为参考关键帧.
- 若当前地图点对参考关键帧的观测被删除(`EraseObservation(KeyFrame* pKF)`),则取第一个观测到当前地图点的关键帧做参考关键帧.

函数`MapPoint::UpdateNormalAndDepth()`的调用时机:

1. 创建地图点时调用`UpdateNormalAndDepth()`初始化其观测信息.

```
pNewMP->AddObservation(pKF, i);
pKF->AddMapPoint(pNewMP, i);
pNewMP->ComputeDistinctiveDescriptors();
pNewMP->UpdateNormalAndDepth();            // 更新平均观测方向和距离
mpMap->AddMapPoint(pNewMP);
```

2. 地图点对关键帧的观测`mObservations`更新时(**跟踪局部地图添加或删除对关键帧的观测**时、`LocalMapping`**线程删除冗余关键帧**时或`LoopClosing`**线程闭环矫正**时),调用`UpdateNormalAndDepth()`初始化其观测信息.

```
pMP->AddObservation(mpCurrentKeyFrame, i);
pMP->UpdateNormalAndDepth();
```

3. 地图点世界坐标`mWorldPos`发生变化时(BA优化之后),调用`UpdateNormalAndDepth()`初始化其观测信息.

```
pMP->SetWorldPos(cvCorrectedP3Dw);
pMP->UpdateNormalAndDepth();
```

总结成一句话: 只要**地图点本身**或**关键帧对该地图点的观测**发生变化,就应该调用函数`MapPoint::UpdateNormalAndDepth()`更新其观测尺度和方向信息.

特征描述子

成员函数/变量	访问控制	意义
`cv::Mat mDescriptor`	`protected`	当前关键点的特征描述子(所有描述子的中位数)
`cv::Mat GetDescriptor()`	`public`	`mDescriptor`的get方法
`void ComputeDistinctiveDescriptors()`	`public`	计算`mDescriptor`

一个地图点在不同关键帧中对应不同的特征点和描述子,其特征描述子`mDescriptor`是其在所有观测关键帧中描述子的中位数(准确地说,该描述子与其他所有描述子的中值距离最小).

- 特征描述子的更新时机:

  一旦某地图点对关键帧的观测`mObservations`发生改变,就调用函数`MapPoint::ComputeDistinctiveDescriptors()`更新该地图点的特征描述子.

- 特征描述子的用途:

  在函数`ORBmatcher::SearchByProjection()`和`ORBmatcher::Fuse()`中,通过比较**地图点的特征描述子**与**图片特征点描述子**,实现将**地图点**与**图像特征点**的匹配(3D-2D匹配).

地图点的删除与替换

成员函数/变量	访问控制	意义
`bool mbBad`	`protected`	坏点标记
`bool isBad()`	`public`	查询当前地图点是否被删除(本质上就是查询`mbBad`)
`void SetBadFlag()`	`public`	删除当前地图点
`MapPoint* mpReplaced`	`protected`	用来替换当前地图点的新地图点
`void Replace(MapPoint *pMP)`	`public`	使用地图点`pMP`替换当前地图点

地图点的删除: `SetBadFlag()`

变量`mbBad`用来表征当前地图点是否被删除.

删除地图点的各成员变量是一个较耗时的过程,因此函数`SetBadFlag()`删除关键点时采取**先标记再清除**的方式,具体的删除过程分为以下两步:

- 先将坏点标记`mbBad`置为`true`,逻辑上删除该地图点.(地图点的**社会性死亡**)
- 再依次清空当前地图点的各成员变量,物理上删除该地图点.(地图点的**肉体死亡**)

这样只有在设置坏点标记`mbBad`时需要加锁,之后的操作就不需要加锁了.

```
void MapPoint::SetBadFlag() {
    map<KeyFrame *, size_t> obs;
    {
        unique_lock<mutex> lock1(mMutexFeatures);
```

```
        unique_lock<mutex> lock2(mMutexPos);
        mbBad = true;           // 标记mbBad,逻辑上删除当前地图点
        obs = mObservations;
        mObservations.clear();
    }

    // 删除关键帧对当前地图点的观测
    for (KeyFrame *pKF : obs.begin()) {
        pKF->EraseMapPointMatch(mit->second);
    }

    // 在地图类上注册删除当前地图点,这里会发生内存泄漏
    mpMap->EraseMapPoint(this);
}
```

成员变量`mbBad`表示当前地图点逻辑上是否被删除,在后面用到地图点的地方,都要通过`isBad()`函数确认当前地图点没有被删除,再接着进行其它操作.

```
int KeyFrame::TrackedMapPoints(const int &minObs) {
    // ...


    for (int i = 0; i < N; i++) {
        MapPoint *pMP = mvpMapPoints[i];
        if (pMP && !pMP->isBad()) {         // 依次检查该地图点物理上和逻辑上是否删除,若删除了就不对其操作
            // ...
        }

    // ...
}
```

地图点的替换: `Replace()`

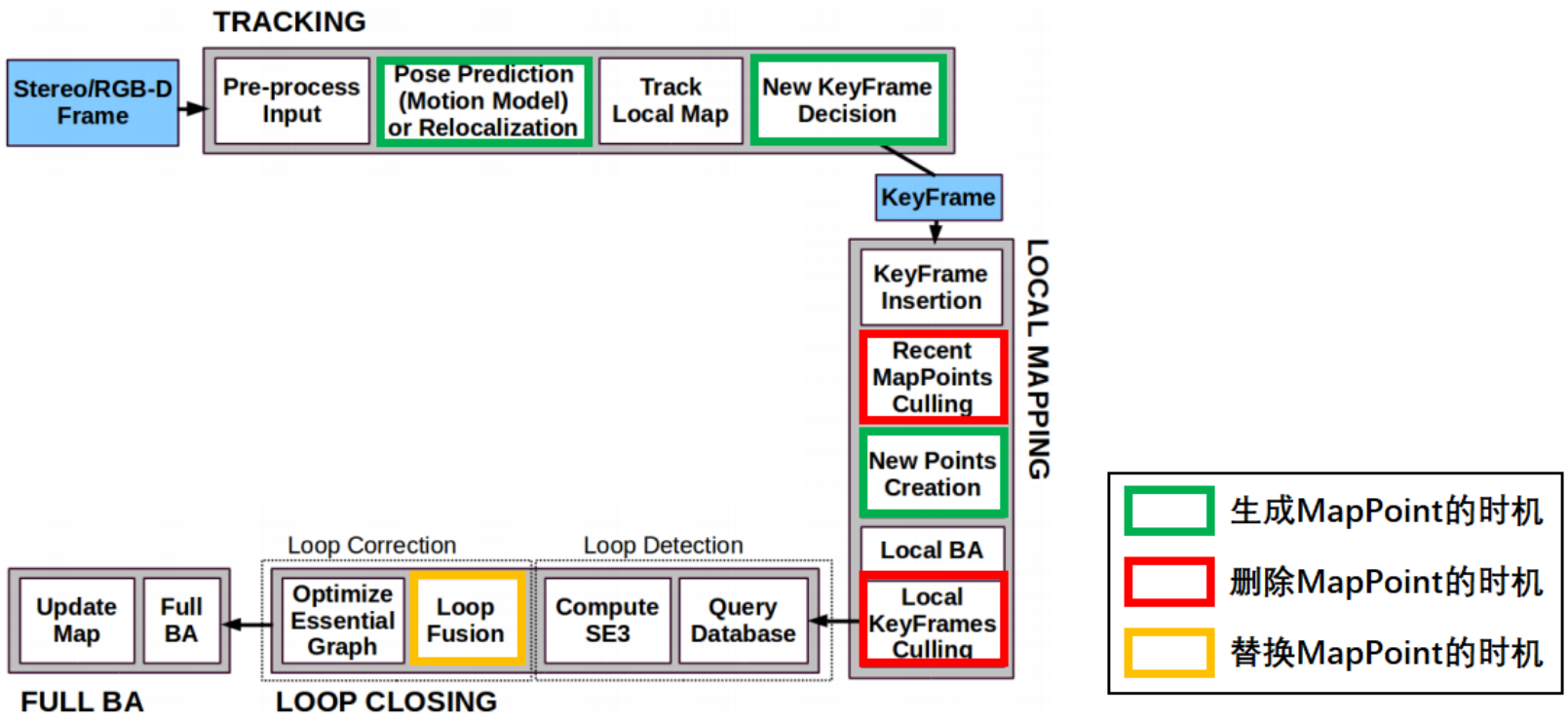
函数`Replace(MapPoint* pMP)`将当前地图点的成员变量叠加到新地图点`pMP`上.

```
void MapPoint::Replace(MapPoint *pMP) {
    // 如果是同一地图点则跳过
    if (pMP->mnId == this->mnId)
        return;

    // step1. 逻辑上删除当前地图点
    int nvisible, nfound;
    map<KeyFrame *, size_t> obs;
    {
        unique_lock<mutex> lock1(mMutexFeatures);
        unique_lock<mutex> lock2(mMutexPos);
        obs = mObservations;
        mObservations.clear();
        mbBad = true;
        nvisible = mnVisible;
        nfound = mnFound;
        mpReplaced = pMP;
    }

    // step2. 将当地图点的数据叠加到新地图点上
    for (map<KeyFrame *, size_t>::iterator mit = obs.begin(), mend = obs.end(); mit != mend; mit++) {
        KeyFrame *pKF = mit->first;
        if (!pMP->IsInKeyFrame(pKF)) {
            pKF->ReplaceMapPointMatch(mit->second, pMP);
            pMP->AddObservation(pKF, mit->second);
        } else {
            pKF->EraseMapPointMatch(mit->second);
        }
    }

    pMP->IncreaseFound(nfound);
    pMP->IncreaseVisible(nvisible);
    pMP->ComputeDistinctiveDescriptors();

    // step3. 删除当前地图点
    mpMap->EraseMapPoint(this);
}
```

`MapPoint`类的用途

`MapPoint`的生命周期

针对`MapPoint`的生命周期,我们关心以下3个问题:



- 创建`MapPoint`的时机:
    1. `Tracking`线程中初始化过程(`Tracking::MonocularInitialization()`和`Tracking::StereoInitialization()`)
    2. `Tracking`线程中创建新的关键帧(`Tracking::CreateNewKeyFrame()`)
    3. `Tracking`线程中恒速运动模型跟踪(`Tracking::TrackWithMotionModel()`)也会产生临时地图点,但这些临时地图点在跟踪成功后会被马上删除(那跟踪失败怎么办?跟踪失败的话不会产生关键帧,这些地图点也不会被注册进地图).
    4. `LocalMapping`线程中创建新地图点的步骤(`LocalMapping::CreateNewMapPoints()`)会将当前关键帧与前一关键帧进行匹配,生成新地图点.
- 删除`MapPoint`的时机:
    1. `LocalMapping`线程中删除恶劣地图点的步骤(`LocalMapping::MapPointCulling()`).
    2. 删除关键帧的函数`KeyFrame::SetBadFlag()`会调用函数`MapPoint::EraseObservation()`删除地图点对关键帧的观测,若地图点对关键帧的观测少于`2`,则地图点无法被三角化,就删除该地图点.
- 替换`MapPoint`的时机:
    1. `LoopClosing`线程中闭环矫正(`LoopClosing::CorrectLoop()`)时**当前关键帧**和**闭环关键帧**上的地图点发生冲突时,会使用闭环关键帧的地图点替换当前关键帧的地图点.
    2. `LoopClosing`线程中闭环矫正函数`LoopClosing::CorrectLoop()`会调用`LoopClosing::SearchAndFuse()`将**闭环关键帧的共视关键帧组中所有地图点**投影到**当前关键帧的共视关键帧组**中,发生冲突时就会替换.