**MSCKF-VIO源码框架及C++知识点总结**



# 摘要


阅读源码是最有效的学习方式，不仅可以从中理清作者的思路以及工程实现细节，还可以学到相应的编程知识，可谓一举多得。 本文主要从代码架构以及c++实现两个方向对msckv-vio进行解剖，有关msckf-vio的介 绍详见知乎专栏MSCKF那些事。有关数学推导以及流程，知乎专栏里已经很详细很棒棒的了，在此不做重复 性的工作，力求方便后来者快速入门。 本人也是一边看着知乎专栏的介绍，一边阅读源码的。技术博客，本身入门菜鸟一个，但必定尽心尽力，也难免有错误之处，敬请留言批评指出，不甚感激，亦欢迎多多交流。

知乎专栏MSCKF那些事

msckf-vio github地址

本人注释过的源码

参考论文1：Robust Stereo Visual Inertial Odometry for Fast Autonomous Flight

参考论文2：Paper：A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation

参考论文3：Report:A Multi-State Constraint Kalman Filter for Vision-aided Inertial

Navigation

参考论文4：Observability-constrained Vision-aided Inertial Navigation

# MSCKF-VIO程序架构

MSCKF-VIO是双目版的MSCKF，一个基于卡尔曼滤波的VIO系统，注意只是一个里程计，没有回环和地图复用，所以算不上一个Slam框架。

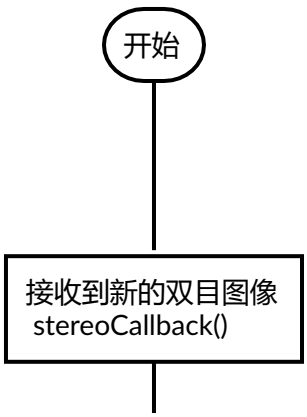
MSCKF-VIO是我见过的最简单的一个VIO框架，只有5个cpp程序文件，整个算法实现部分全部在msckf_vio和image_processor两个cpp文件里,一共只有程序可读性非常好，对于初学者而言是很好的入门级算法框架，而且根据论文后面的评测，该算法拥有与主流优化算法VINS-Mono（关闭loop closure功能）相当的性能（不过一个是单目，一个是双目，比较有点牵强，但滤波VIO能达到这样的效果已经非常棒了），且与基于双目优化的OKVIS有相当的性能，但计算效率却非常高，能达到20Hz，实现了性能与计算效率上的平衡。

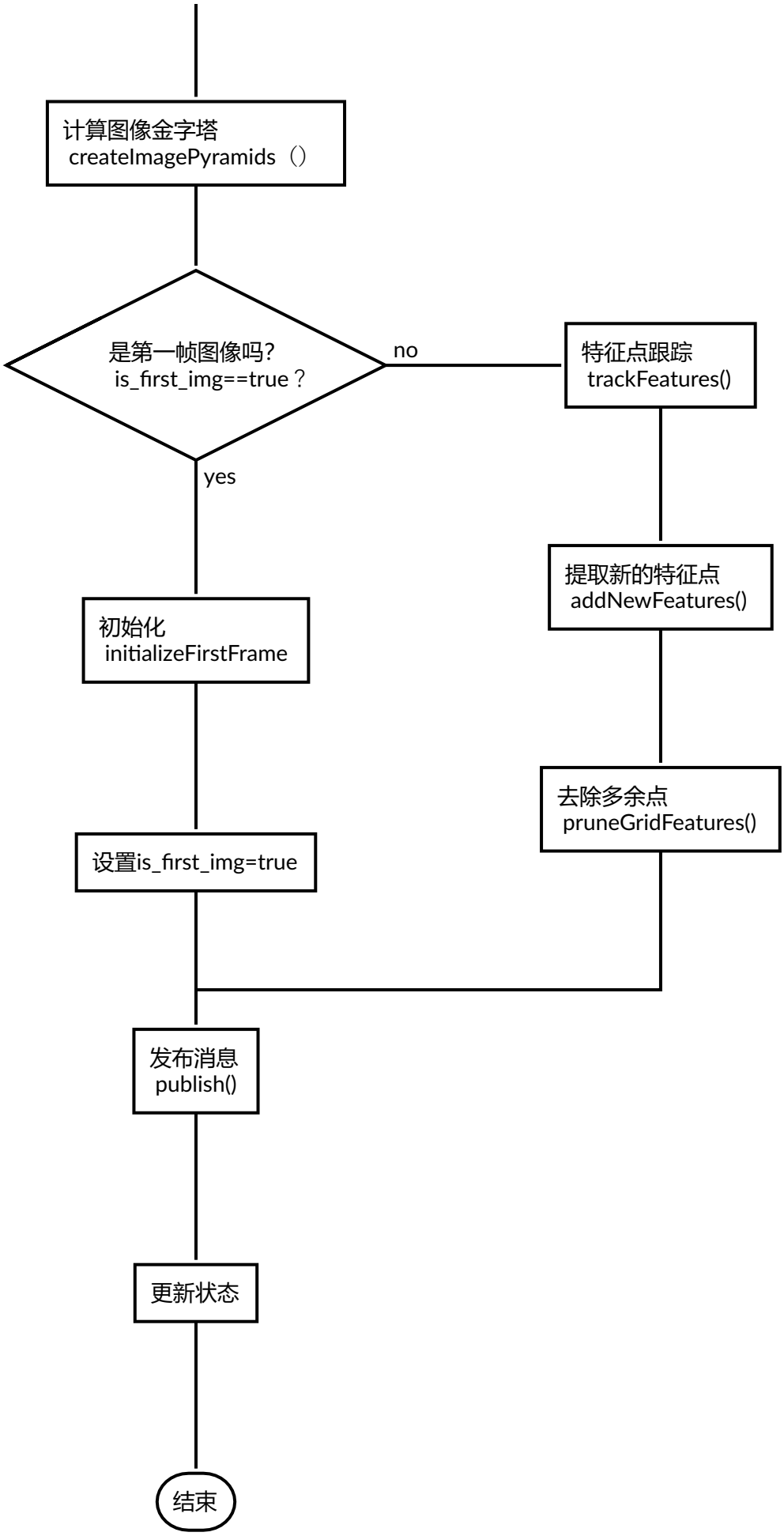
代码结构非常清晰，分为前端和后端两个部分，分别在image_processor.cpp和msckf-vio.cpp程序文件里。

## 前端

### 前端流程图

前端主要进行视觉特征的提取，跟踪匹配以及移除outlier。




计算图像金字塔
createImagePyramids（）
是第一帧图像吗?
is_first_img==true ?
no
特征点跟踪
trackFeatures()
yes
提取新的特征点
addNewFeatures()
初始化
initializeFirstFrame
去除多余点
pruneGridFeatures()
设置is_first_img=true
发布消息
publish()
更新状态
结束

## 函数功能解读

下面就各个函数模块进行功能解剖：
**createImagePyramids()**:调用opencv的创建金字塔函数，创建3层双目图像cam0,cam1金字塔。

**initializeFirstFrame()**:收到第一帧图像之后进行初始化，主要步骤包括：
提取FAST角点—>KL光流法进行双目匹配—>图像区域分成4x5块，并将匹配的点分配到各个块中—>方块里选取response较大的2个角点。经过这几步,使得提取的特征点尽可能均匀分布,且角点响应强度好。

**trackFeatures()**:跟踪上一帧中的特征点，主要分以下步骤：
利用IMU角速度积分得到旋转矩阵,预测上一帧cam0中的特征点在当前cam0中的位置—>KL光流追踪特征点,并去除界外的点以及未被跟踪上的点—>去除外点，这又包括三个步骤：
step1：去除当前帧cam0中与cam1匹配不上的点
step2：利用RANSAC算法去除当前帧cam0中与前一帧cam0基础矩阵约束失效的匹配点
step3：利用RANSAC算法去除当前帧cam1中与前一帧cam1基础矩阵约束失效的匹配点
这里用的ransac算法一开始的时候没看懂，后来通过阅读2-point RANSAC明白就是基于F矩阵的RANSAC算法，只不过它利用了两帧间的陀螺仪积分得到的旋转矩阵，所以RANSAC的时候只需要计算平移矩阵，加速了这个过程。经过这一步得到的匹配关系有如下特点：和前一帧匹配上，双目能匹配上，基础矩阵几何约束。

**addNewFeatures()** :经过上面的匹配和筛选，当前帧的特征点数量一般会低于最低数量要求，所以该步骤会检测cam0已经追踪的特征点之外的新的fast特征点,然后在cam1中寻找匹配点,并在图像块中添加response较大的特征点，这样就能保证新的帧里有足够多且分布均匀的特征点。

**pruneGridFeatures()**: 如果网格里特征点太多,则去除lifetime较小的特征点,优先保留一直在被跟踪的特征点。特征点每被跟踪一次，其lifetime就会加1。当前帧图像块里的特征点来自追踪+新产生，而产生新特征点的过程保证了图像块中的特征点数量不会低于最少特征点数的要求，所以骤步起作用的前提在于前一帧不同网格的太多特征点投影到当前帧同一个分块图像中来了。

**publish()** :发布当期帧特征点的去畸变归一化平面上的坐标以及跟踪信息，每个坐标有4个参数，代表该特征点在双目归一化平面上的坐标。跟踪信息包括该帧的时间戳，跟踪前后的特征点数量，双目匹配去除外点前后的特征点数量，RANSAC去除外点前后的特征点数量。

**更新**:状态更新将当前帧的信息赋予上一帧，然后对当前帧信息重置。

通过以上步骤分析，可知跟踪的约束很多，保证了跟踪的准确性，论文里也提及前端耗时占整个算法的80%。

## 前端各主要函数模块耗时分析

| 函数 | 平均耗时(s) |
| --- | --- |
| createImagePyramids() | 0.00377 |
| initializeFirstFrame() | 0.01900 |
| trackFeatures() | 0.00455 |
| addNewFeatures() | 0.00677 |
| pruneGridFeatures() | 1.866e-6 |
| publish() | 8.542e-5 |
| total_time | 0.01526 |

备注：
用了四舍五入，可能total_time会比前面的和小一点点。
测试条件：Dell G7-7588 ； intel i7-8750H @2.2GHz； ubuntu6.04
数据集：msckf-vio的github上推荐的EuRoC ROS Bags其中的**Vicon Room 1 01**，一共2892帧图像。

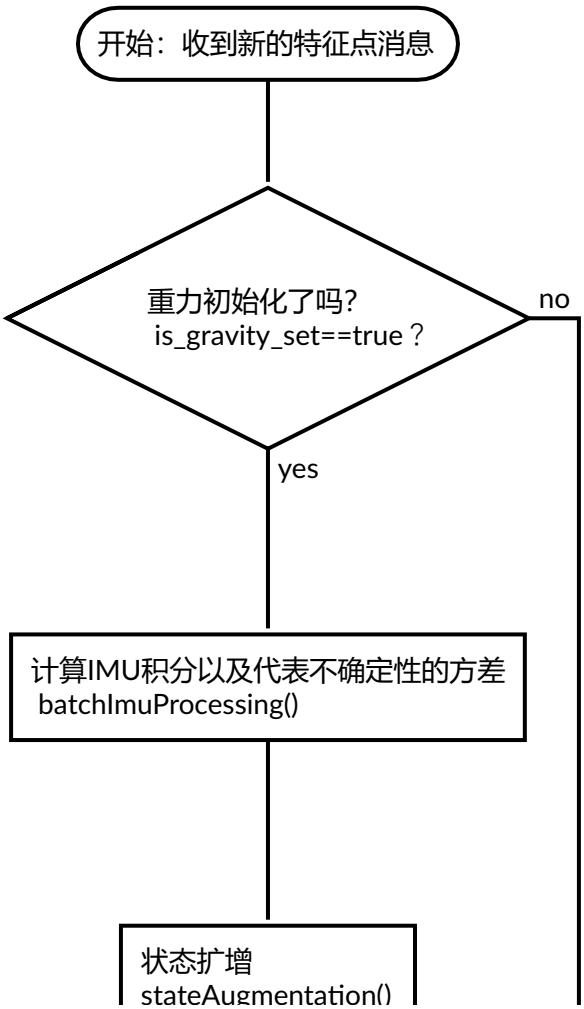
从表中可以看出前端大部分时间耗在**trackFeatures()** 和 **addNewFeatures()** 两部分，如果要加速应该在这部分进行优化，每秒能处理65帧，已经很高了。
initializeFirstFrame虽然耗时多，但只会执行一次，不会对运行效率产生影响。

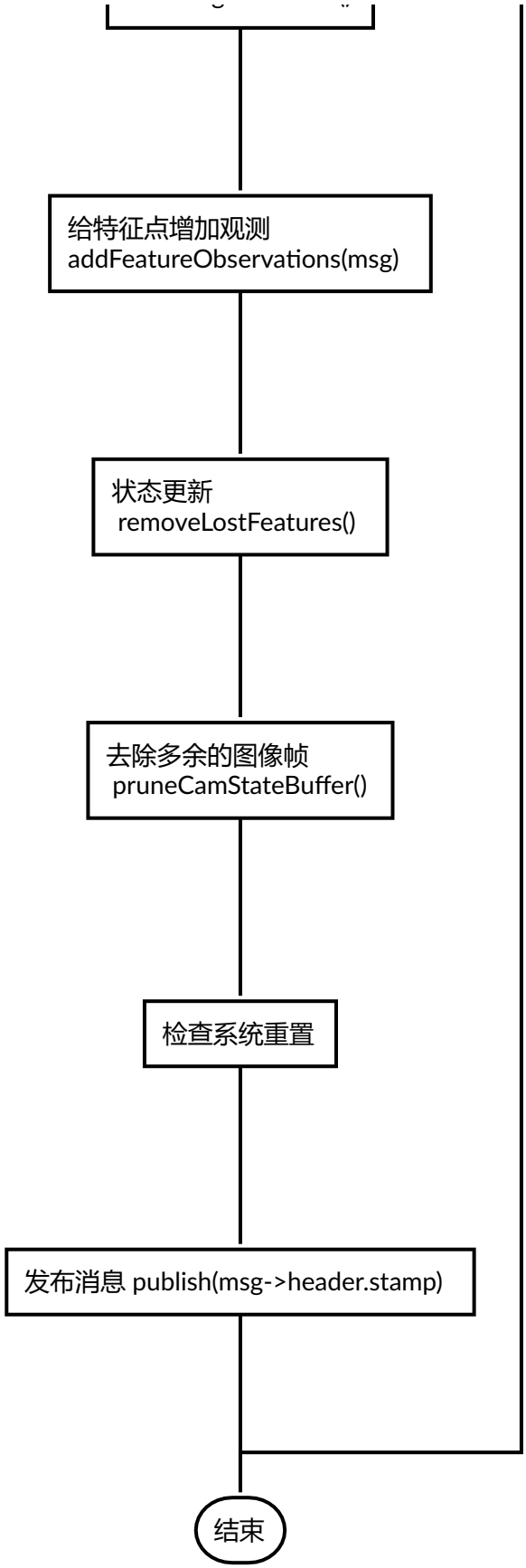
# 后端

后端进行的任务主要有重力和陀螺仪bias的初始化以及EKF状态更新。
重力和陀螺仪bias的初始化较简单，initializeGravityAndBias()，静止状态下利用前200个imu数据，计算出陀螺仪的均值即为bias初始值（静止状态下陀螺仪真实值为0），加速度测量均值作为重力在imu坐标系下的近似真实值，根据重力在世界坐标系下的真实值以及在imu坐标系下的测量值，可以求出初始状态时imu系相对世界坐标系的pose，初始化时世界坐标系原点与imu系原点重合。**所以一定要注意，初始化的时候要使系统处于静止状态，否则会初始化失败，前端能匹配成功，但后端没有里程计输出的**。

## 后端流程图

EKF过程如下：






图中省略了一个is_first_img，判断是否为第一帧，是第一帧图像就将IMU状态的时间设为这帧图像，并将is_first_img=false，之后就不会再进入（如果系统收到重置信号，is_first_img会变成true）

## 函数功能解读

下面就各个函数模块进行功能解剖：
**batchImuProcessing()**: 处理当前帧与上一帧之间的IMU数据，得到当前帧IMU的位姿以及方差，其数学推导见开头提及的知乎专栏，不同的一点是程序有Modify the transition matrix，与msckf-vio论文提及的III-C:Observability Constraint有关,这块内容知乎专栏没讲清楚，也比较难懂，其数学推导详见参考论文4：Observability-constrained Vision-aided Inertial Navigation

**stateAugmentation()**: 收到新的图像cam，就要将原来的协方差矩阵扩增，已知IMU的状态及其协方差矩阵，IMU-CAM之间的位姿变换关系，可以得到cam的位姿及协方差矩阵，其数学推导见MSCKF那些事（六）算法详解4：State Augmentation。在这里当前帧的位姿与IMU积分推导的位姿存在协方差，后面对当前帧位姿的更新动力来源与该协方差而不是来自于观测值；而IMU的位姿又与前一帧时刻的相机位姿存在协方差，之后IMU位姿更新的直接动力来自于该协方差。

**addFeatureObservations(msg)**:添加当前帧里的特征：如果是新的特征点,则将其加入到map_server中，如果是之前存在的特征点,则将当前帧加入到该点的观测之中。

**removeLostFeatures()**:这个函数名取得有点名不副实，正如论文里所讲的，如果有特征点丢失跟踪了，则用这些特征点来进行EKF更新，该函数就是EKF更新过程，分为如下三步：

*step1)*:三角化当前帧里track lost的特征点,能三角化的特征点被观察次数超过2次,且通过相机运动检测,三角化推导详见msckf论文的Appendix

*step2)*:计算失去跟踪的特征点的观测残差 对相机位姿和特征点三维坐标的雅克比矩阵,最终得到EKF里的观测方程

*step3)*:EKF状态更新

其中的数学推导详见MSCKF那些事（七）算法详解5：Measurement Update

**pruneCamStateBuffer()**: 改步移除状态窗口中多余的相机状态，如果相机状态达到上限，则每次移除两个，所以程序运行的时候可以看到状态窗口中相机状态数量一直维持在比上限少1个或者2个（不计重置）。分为以下n步:

*step1)*:选出要移除的两帧图像I1,I2:比较最新第2,3帧相较最新第4帧的移动量,如果发现移动量太小,则移除这些帧,否则移除最老的帧

*step2)*:map_serve里的特征点除去I1,I2观测,如果不能被滑动窗口中的观测(此时还没移除I1,I2)初始化,则去除该特征点。

*step3)*: 求状态偏差对I1,I2偏差的雅可比矩阵以及相应的观测误差，类似之前的方

法进行EKF更新,这个对状态估计的优化性能有多少,值得怀疑。

*step4)*: 移除I1,I2及其对应的协方差

***publish(msg->header.stamp)*** : 发布状态消息:IMU位姿T_i_w（相对世界坐标系的旋转矩阵+位置），在世界坐标系中的速度及协方差矩阵，初始化了的特征点的三维坐标。

***onlineReset()*** :如果系统的位置标准差超过一定的阈值,就对系统进行重置。对state_server内的相机清零,map_server的特征点清零,IMU状态的方差重新初始化,其中pose和position的方差设为0,但IMU的位置,速度,pose并没变

## 后端各主要函数模块耗时分析

| 函数 | 平均耗时(s) |
| --- | --- |
| batchImuProcessing() | 0.00215 |
| stateAugmentation() | 0.00016 |
| addFeatureObservations() | 3.63e-5 |
| removeLostFeatures() | 0.00210 |
| pruneCamStateBuffer() | 0.00892 |
| publish() | 2.69e-5 |
| total_time | 0.01340 |
| onlineReset()次数 | 0 |

备注：
用了四舍五入，可能total_time会比前面的和小一点点。
测试条件：Dell G7-7588 ； intel i7-8750H @2.2GHz； ubuntu6.04
数据集：msckf-vio的github上推荐的EuRoC ROS Bags其中的**Vicon Room 1 01**，一共2892帧图像。

分析：对比可知，后端耗时略低于前端，每秒可以处理74帧数据， 这与论文里讲的前端消耗80%的时间不符，这只是一家之言，如果您有不同的结论，欢迎在评论区

指出，不甚感激！ 另外后端主要耗时在removeLostFeatures()和pruneCamStateBuffer()上，前者是论文提及的EKF滤波过程，后者是移除多余的帧，同时还运行了EKF过程，这个函数一般各一次才运行一次，也就是说实际改函数运行时间为表格中的两倍，单独记录的结果验证了这个推断，如果后端要做优化，可以从这两个函数，特别是后一个函数处着手。但后端速度已经很快了，没有加速的必要。

## 运行过程分析

看论文时一直以为每次接收到新的帧，EKF算法会用当前帧的观测来立即更新该帧的位姿，其实不然，先看下列一段运行时状态窗口内的相机ID记录：

```
==========================================
这是第19次进入FeatureCallback函数;扩增之前cam数:18其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
扩增之后最新一帧图像由IMU计算得到的cam位姿为Q:
-0.579322 -0.591993  0.385008  0.407062位置:-0.00106238  -0.0685355  -
扩增之后cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
修正的IMU旋转量-0.000428545   0.00073752  0.000376173  1
修正的IMU位移 0.00839804   -0.004347 -0.00124921
最新帧的上一帧的位置修正量:
 0.00758024 -0.00415305 -0.00120079四元数修正量
0.000679556 0.000407112 0.000366278           1
最新帧的位置修正量:
  0.0083858 -0.00432602  -0.0013043四元数修正量
0.000719419 0.000431556 0.000388199           1
观测涉及到的camID数量为:9其ID分别是
9 10 11 12 13 14 15 16 17
EKF更新后最新帧位姿为Q:
-0.579425 -0.591315  0.385342  0.407585位置:  0.00732342 -0.0728615 -0
EKF状态更新之后cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
去掉多余的之后cam之后最新帧位姿为Q:
-0.579425 -0.591315  0.385342  0.407585位置:  0.00732342 -0.0728615 -0
去掉多余的之后cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
==========================================
这是第20次进入FeatureCallback函数;扩增之前cam数:19其ID分别是
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
扩增之后最新一帧图像由IMU计算得到的cam位姿为Q:
-0.579454 -0.591189   0.38541  0.407662位置:0.00800001 -0.0735398 -0.0
扩增之后cam数:20其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
EKF更新后最新帧位姿为Q:
-0.579454 -0.591189   0.38541  0.407662位置:  0.00800001 -0.0735398 -0
EKF状态更新之后cam数:20其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
修正的IMU旋转量 3.00171e-05 -3.60735e-05  4.30325e-06    1
修正的IMU位移-0.000353244  0.000264745  3.81908e-05
最新帧的上一帧的位置修正量:
-0.00031921 0.000239285  4.0622e-05
四元数修正量
-3.37154e-05 -2.89993e-05  3.39526e-06            1
最新帧的位置修正量:
-0.000355567  0.000263274  4.20609e-05四元数修正量
 -3.5618e-05 -3.05683e-05   3.5938e-06            1
去掉多余的之后cam之后最新帧位姿为Q:
-0.579459 -0.591213  0.385408  0.407622位置:  0.00764444 -0.0732765 -0
去掉多余的之后cam数:18其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19
==============================================
这是第21次进入FeatureCallback函数;扩增之前cam数:18其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19
扩增之后最新一帧图像由IMU计算得到的cam位姿为Q:
-0.579557 -0.591114  0.385435    0.4076位置:0.00839501 -0.0739196 -0.0
扩增之后cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 20
EKF更新后最新帧位姿为Q:
-0.579557 -0.591114  0.385435    0.4076位置:  0.00839501 -0.0739196 -0
EKF状态更新之后cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 20
去掉多余的之后cam之后最新帧位姿为Q:
-0.579557 -0.591114  0.385435    0.4076位置:  0.00839501 -0.0739196 -0
去掉多余的之后cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 20
==============================================
这是第22次进入FeatureCallback函数;扩增之前cam数:19其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 20
扩增之后最新一帧图像由IMU计算得到的cam位姿为Q:
-0.579611 -0.591035  0.385434  0.407638位置:0.00920391 -0.0745504 -0.0
```

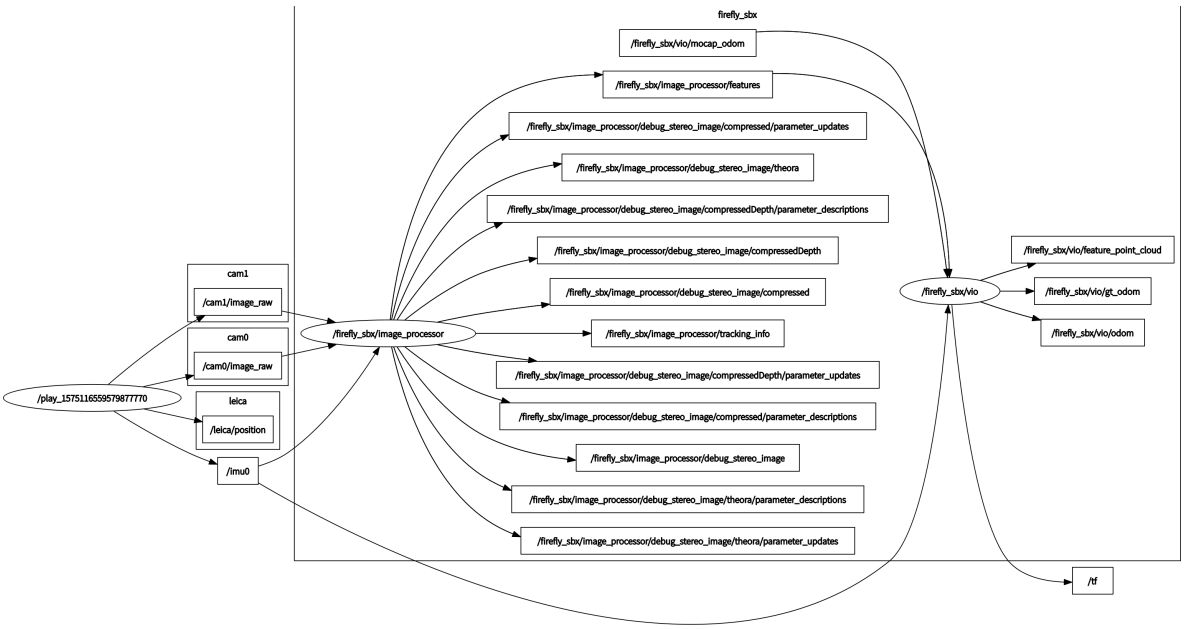
```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 20 21
EKF更新后最新帧位姿为Q:
-0.579611 -0.591035  0.385434  0.407638位置:  0.00920391 -0.0745504 -0.0
EKF状态更新之后cam数:20其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 20 21
修正的IMU旋转量  1.5361e-05 -8.06942e-05  3.24655e-05   1
修正的IMU位移-0.000627034  0.000398831  5.94123e-05
最新帧的上一帧的位置修正量:
-0.000563014  0.000356974  6.50779e-05
四元数修正量
 -7.7338e-05 -1.57806e-05  2.91949e-05            1
最新帧的位置修正量:
-0.000635169  0.000398084  6.14035e-05四元数修正量
-8.11996e-05 -1.65586e-05   3.0635e-05            1
去掉多余的之后cam之后最新帧位姿为Q:
-0.579656 -0.591056  0.385408  0.407569位置:  0.00856874 -0.0741523 -0.0
去掉多余的之后cam数:18其ID分别是
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 21
==========================================
```

从中我们可以看出以下结论:

1. 观测值并没有涉及到当前帧，当前帧是如何进行更新的呢？ 查看源码，状态协方差矩阵里，当前帧和上一帧之间的协方差为零，当前IMU与上一帧的协方差等于上一帧IMU与上一帧图像位姿之间的协方差，这样并不是精确的，有可能是这两个协方差并不那么好计算的缘故，如此推断，观测会更新上一帧，IMU与它的协方差引起当前IMU的更新，而当前图像帧位姿与IMU之间的协方差更新当前位姿。
2. 当状态里的cam数量达到最大数量20时，会一次移除掉两个，这两个是从最老的一帧以及当前帧的前两帧里挑选，这也印证了前面所说的移除多余帧的操作正常情况下每隔一次才调用一次，该函数是所有函数里最耗时的。
3. pruneCamStateBuffer()函数的意义：一般将此操作视为优化里的边缘化，与removeLostFeatures()类似，都采用了EKF过程，不同的是removeLostFeatures()涉及到的观测帧是能看到跟踪丢失的特征点的图像帧，数量不定，而pruneCamStateBuffer()涉及到的是要被移除的帧，只有两帧，但EKF算法的规模是一样大的，相当于算法进行了两次EKF过程（这两个过程有可能涉及到同样的帧，但map_serve事先已经剔除removeLostFeatures()涉及到的特征点，所以保证不会重复），如果能将这两个过程合并，必能提速不少。

4. 本算法采用关键帧选取策略剔除多余帧，可能是最老的一帧，也可能是最新的第二或者第三帧，每次剔除两帧，下一次就不需要剔除，所以我们可以看到状态窗口维护的帧ID可能非常老；此外IMU积分时间就是两帧之间的时间间隔，保证其不会因积分时间长而飘走。
5. 路标点的坐标是通过多视图几何的方法来获得的,类似于三角化,不过有多帧观测该点的相机姿态,不同与优化里同时优化路标点和相机位姿,如此形成循环,位姿优化的准确性取决于路标点的准确性,而路标点的准确性又取决于观测到该点的相机位姿的准确性,理论上讲,准确性应该不及优化算法.

# ROS里的信息流图




# C++编程知识

阅读源码既能有效地了解算法本身，特别是其实现细节，另外也非常有助于学习c++，特别是优秀代码的编程规范，作为一个半道出家的程序员，我也从中受益匪

浅，在此我总结了这套源码中用到的一些c++编程知识（在此假设读者您已经具备c++基础，特别是熟悉了解面向对象编程），这些也是我从中学到的新的知识点：

1. ***Boost智能指针——shared_ptr：*** 在ImageProcessor这个类里有用到智能指针：

```
typedef boost::shared_ptr<ImageProcessor> Ptr,
typedef boost::shared_ptr<MsckfVio> Ptr,
boost::shared_ptr<GridFeatures> prev_features_ptr,
```

主要对象用的都是这类指针。它是可以共享所有权的智能指针，多个这样的智能指针可以指向同一片地址，且会智能地在合适的时候去自动释放资源，不需要delete操作。它的实现机制其实比较简单，就是对指针引用的对象进行引用计数，当有一个新的boost::shared_ptr指针指向一个对象时，就把该对象的引用计数加1，减少一个boost::shared_ptr指针指向一个对象时，就把对该对象的引用计数减1。当一个对象的引用计数变为0时，就会自动调用其析构函数或者free掉相应的空间。具体详见有关智能指针（shared_ptr）的讨论。

2. ***std::map:*** Map是一种关联容器，它按照特定顺序存储由键值Key和映射值Value组合而成的元素通过健Key值来查找对应的值Value。程序里用它定义了多种关联：

```
typedef std::map<int, std::vector<FeatureMetaData> > GridFeatures 图像块
std::map<FeatureIDType, int> feature_lifetime;将特征点ID与其生存时间关联；
map<FeatureIDType, Point2f> prev_points;将特征点ID与点坐标关联；
```

使用案例：
**赋值：** prev_points[feature.id] = feature.cam0_point;不需要像数组或者vector那样需要先知道size，分配空间然后才能复制，map可以直接赋值，非常方便。
**访问：** for (const auto& item : *prev_features_ptr) { for (const auto& prev_feature : item.second)}，这个有点类似于vector的用法，通过迭代器访问，每一个map对象的first是Key值，second是Value值。
**查找：** if (feature_lifetime.find(feature.id) == feature_lifetime.end()) 在map里查找键Key值对应的对象，如果没有，则返回end，有则返回指向第一个对象的迭代器。
std::map有如下特性：
1）关联性：std::map 是一个关联容器，其中的元素根据键来引用，而不是根据索引来引用；
2）有序性：在内部，std::map 中的元素总是按照其内部的比较器（比较器类型由

Compare类型参数指定）指示的特定严格弱序标准按其键排序；
3）唯一性：std::map 中的元素的键是唯一的；
4）std::map 通常由二叉搜索树实现。
更多内容详见C++ std::map 用法详解

3. **std::sort:** 这是一个排序函数，将vector里的元素按照规则进行排序，比如源码里用到

```
for (auto& item : grid_new_features)
  std::sort(item.second.begin(), item.second.end(),&ImageProcessor::fea

其中：
static bool featureCompareByResponse(const FeatureMetaData& f1,const Fea
                {return f1.response > f2.response; }`for (auto& item :
```

合起来就是按照response从大到小的顺寻对item里的元素进行排序。

4. **std::vector:** 容器，类似于一个数组，但其成员可以是任何对象实例，有相应的函数进行初始化，插入元素，移除元素，赋值等等，非常好用。如std::vector<cv::Mat> curr_cam0_pyramid_就是建立了图像金字塔的容器，每一个元素代表金字塔里的一层。

5. **ROS** 代码引入了ROS来方便快捷地处理数据，也是一次很好学习ROS的机会，主要涉及知识点如下：

```
    1)ros::NodeHandle nh;//获取节点的句柄,这个是Starting the node
2)getPrivateNodeHandle()
// Get the private node handle (provides this nodelets custom rema
3)ros::Publisher odom_pub  //定义一个发布者
  ros::Subscriber imu_sub; //定义一个订阅者
  ros::ServiceServer reset_srv;//定义一个服务端
4) odom_pub = nh.advertise<nav_msgs::Odometry>("odom", 10);
 //在ROS MASTER(主机)中注册，10代表其缓存区大小，odom是发布的topic名，nav
   imu_sub = nh.subscribe("imu", 100, &MsckfVio::imuCallback, this
 //在ROS MASTER(主机)中注册订阅imu数据信息，参数分别是 订阅的topic，缓存区
    reset_srv = nh.advertiseService("reset", &MsckfVio::resetCallb
 //在ROS MASTER中注册重置reset服务端，参数分别是名称,回调函数以及调用该服务
5) odom_pub.publish(odom_msg);//发布消息
6) nh.param<int>("grid_row", processor_config.grid_row, 4);
```

注意,有关图像消息的订阅和发布有所不同

图像消息的订阅：

```
message_filters::Subscriber<sensor_msgs::Image> cam0_img_sub;
message_filters::Subscriber<sensor_msgs::Image> cam1_img_sub;
message_filters::TimeSynchronizer<sensor_msgs::Image, sensor_msgs::Image
cam0_img_sub.subscribe(nh, "cam0_image", 10);//注册订阅cam0_image
cam1_img_sub.subscribe(nh, "cam1_image", 10);//注册订阅cam1_image
stereo_sub.connectInput(cam0_img_sub, cam1_img_sub);//关联两个cam消息
stereo_sub.registerCallback(&ImageProcessor::stereoCallback, this);//回调
```

图像消息的发布：

```
image_transport::Publisher debug_stereo_pub;//定义
debug_stereo_pub = it.advertise("debug_stereo_image", 1);//注册发布 debug
if(debug_stereo_pub.getNumSubscribers() > 0){
      debug_stereo_pub.publish(debug_image.toImageMsg()); }  //发布
```

此外，不同于我们大多数见过的ros程序，该项目里并没有见到ros::init()，可能与其ros节点的初始化方法有关：用getPrivateNodeHandle() 来初始化类对象ros节点nh。