

第4周：ORB_SLAM2 课程课件

本周课程重点：

1. 地网点相关

1.1 代表性描述子的计算

1.2 地网点法线朝向的计算

2. 视觉词袋 Bag of Words

2.1 直观理解词袋

2.2 为什么要研究BoW?

类帧差方法

词袋法

2.3 如何制作、生成BoW?

为什么 BOW一般用 BRIEF描述子?

离线训练 vocabulary tree (也称为字典)

在线图像生成BoW向量

什么是节点 (node) ?

源码解析

BowVector

FeatureVector

2.4 vocabulary tree 的保存和加载

如何保存训练好的 vocabulary tree 存储为txt文件?

如何加载训练好的 vocabulary tree txt文件?

3. 用参考关键帧来跟踪

4 关于g2o和图优化

第4周：ORB_SLAM2 课程课件

本课件是公众号 计算机视觉life 旗下课程 [《全网最详细的ORB-SLAM2精讲：原理推导+逐行代码分析》](#) (点击可跳转课程详情) 的课程课件。谢谢各位学员的支持!

本课程对应的注释代码：https://github.com/electech6/ORBSLAM2_detailed_comments

由于源码注释和课件在持续更新，所以：

如视频课程中注释与上述GitHub中有不同，**以GitHub上最新源码为准。**

如视频课程中课件与本课件不同，**以本课件为准。**

本周课程重点：

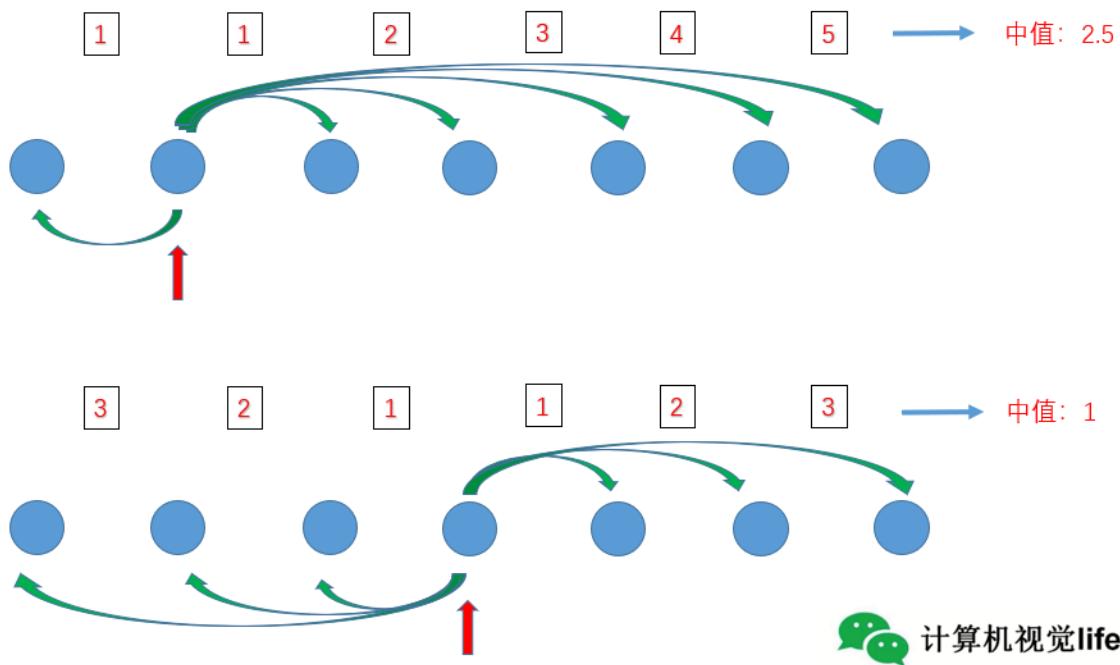
1. 理解地图点里成员变量的物理意义。
2. 掌握BOW的原理及应用（重要）。
3. 掌握关键帧跟踪原理（重要）。
4. 理解图优化原理，理解g2o的使用方法（重要）。

1. 地网点相关

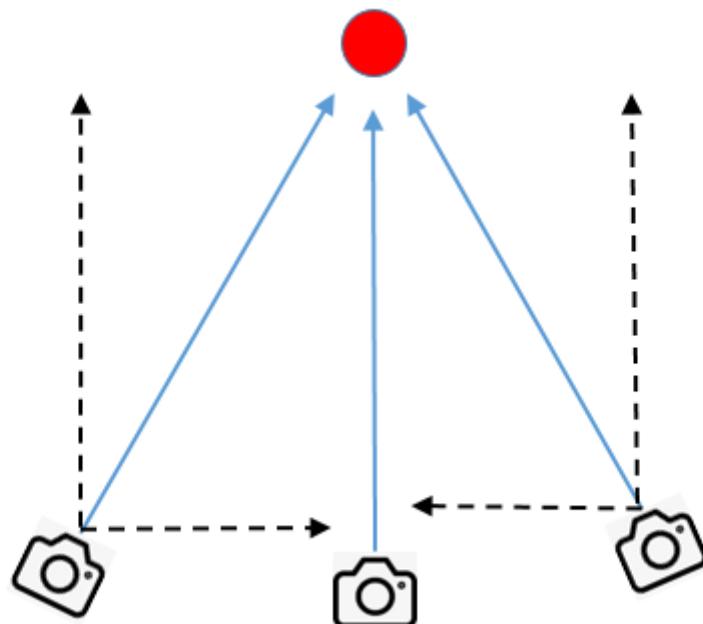
1.1 代表性描述子的计算

找最有代表性的描述子示意图

最有代表的描述子与其他描述子具有最小的中值
距离 Descript or Distance
代表性描述子与其他描述子的中值距离小相当于
它与其他描述子最近, \Rightarrow 最具代表性

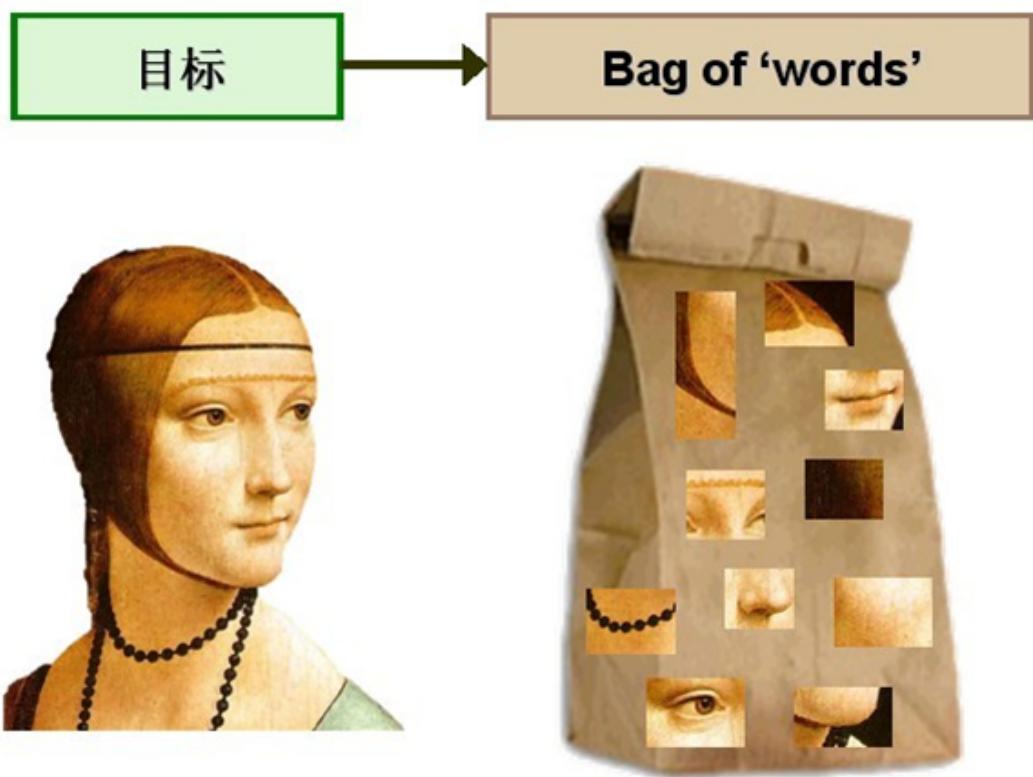


1.2 地图点法线朝向的计算

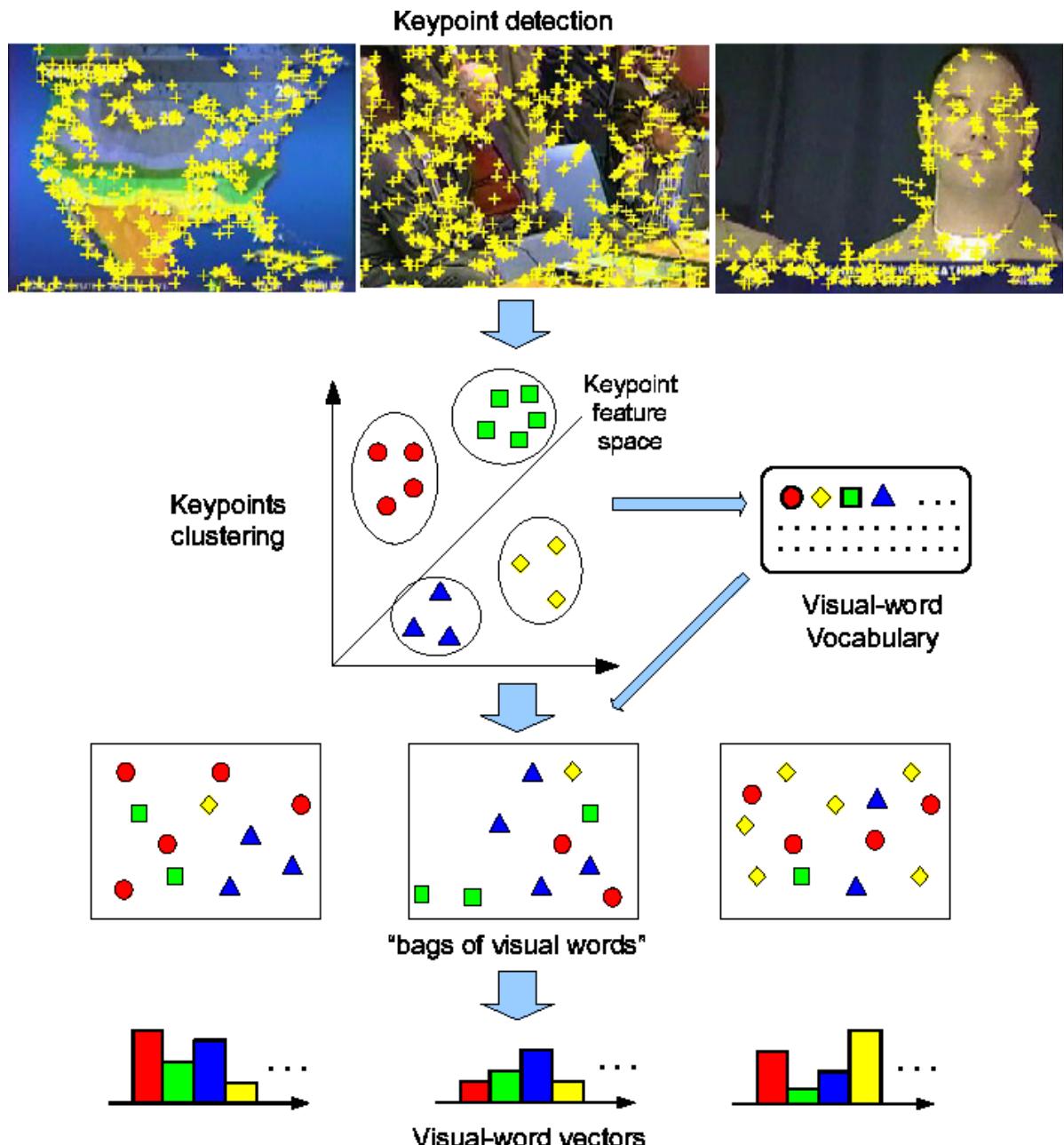


2. 视觉词袋 Bag of Words

2.1 直观理解词袋



词袋的大概流程



为什么叫 bag of words 而不是 list of words 或者 array of words?

因为丢弃了Word出现的排列顺序、位置等因素，只考虑出现的频率，大大简化了表达，节省了存储空间，在分析对比相似度的时候非常高效

<https://gurus.pyimagesearch.com/the-bag-of-visual-words-model/>

2.2 为什么要研究BoW?

闭环检测：核心就是判断两张图片是否是同一个场景，也就是判断图像的相似性。

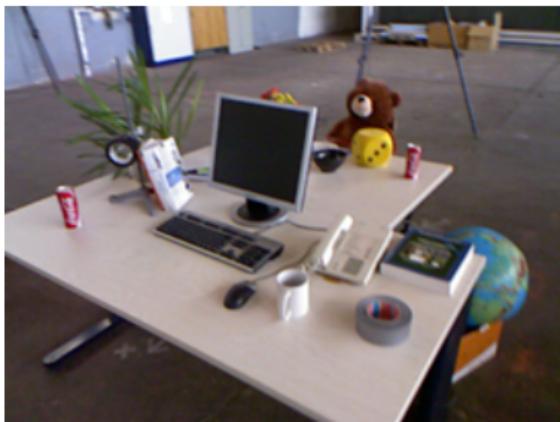
如何计算两张图的相似性？

类帧差方法

不行，这样做有很多问题

视角变化

没办法将两张图的像素进行一一匹配，直接帧差时几乎不能保证是同一个像素点的差。



光照变换

同一视角，不同时间的光照对比



相机曝光不同



词袋法

Bag of words 可以解决这个问题。是以图像特征集合作为visual words，只关心图像中有没有这些words，有多少次，更符合人类认知方式，对不同光照、视角变换、季节更替等非常鲁棒。

加速匹配

ORB-SLAM2代码中使用的 SearchByBoW（用于关键帧跟踪、重定位、闭环检测SIM3计算），以及局部地图里的SearchForTriangulation，内部实现主要是利用了 BoW中的FeatureVector 来加速特征匹配。

使用FeatureVector 避免了所有特征点的两两匹配，只比较同一个节点下的特征点，极大加速了匹配效率，至于匹配精度，论文 《Bags of Binary Words for Fast Place Recognition in Image Sequences》 中提到在26292 张图片里的 false positive 为0，说明精度是有保证的。实际应用中效果非常不错。

缺点：

需要提前加载离线训练好的词袋字典，增加了存储空间。但是带来的优势远大于劣势，而且也有不少改进方法比如用二进制存储等来压缩词袋，减少存储空间，提升加载速度。

2.3 如何制作、生成BoW？

为什么 BOW一般用 BRIEF描述子？

速度方面

因为计算和匹配都非常快，论文中说大概一个关键点计算256位的描述子只需要 $17.3\mu s$

因为都是二进制描述子，距离描述通过汉明距离，使用异或操作即可，速度非常快。

而SIFT, SURF 描述子都是浮点型，需要计算欧式距离，会慢很多。

在Intel Core i7 , 2.67GHz CPU上，使用FAST+BRIEF 特征，在26300帧图像中 特征提取+词袋位置识别耗时 22ms 每帧。

在精度方面

先上结论：闭环效果并不比SIFT, SURF之类的高精度特征点差。

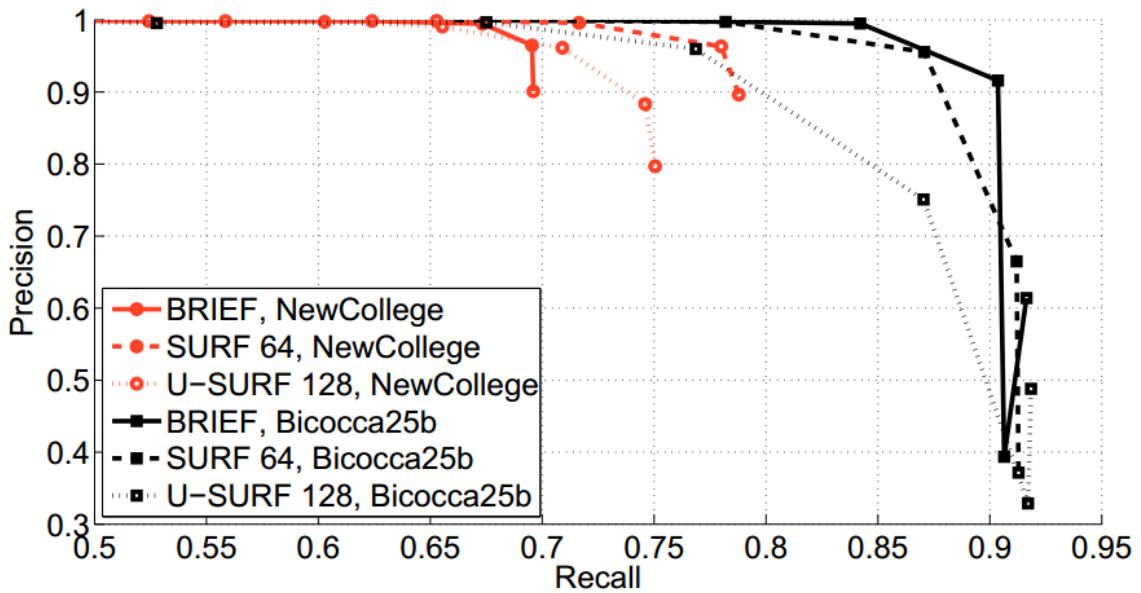
具体来看一下对比：

以下对比来自论文 《2012, Bags of Binary Words for Fast Place Recognition in Image Sequences, IEEE TRANSACTIONS ON ROBOTICS》

三种描述子 BRIEF,, SURF64 , U-SURF128 使用同样的参数，在训练数据集NewCollege, Bicocca25b 上的 Precision-recall 曲线

其中SURF64：带旋转不变性的 64 维描述子

U-SURF128：不带旋转不变性的128维描述子



在两个数据中，SURF64 都明显比 U-SURF128 表现的好（曲线下面积更大），可以看到在Bicocca25b 数据集中，BRIEF明显比 U-SURF128 好，比SURF64 也稍微好一些，在NewCollege 数据集中 SURF64 比 BRIEF 更好一点，但是BRIEF也仍然不错。

总之，BRIEF 和 SURF64 效果基本相差不大，可以说打个平手。

可视化效果

可视化看一下效果

下图左边图片对是BRIEF 在vocabulary 中同样的Word下的回环匹配结果，同样的特征连成了一条线。

下图右边图像对是同样数据集中SURF64 的闭环匹配结果。

第一行 来看，尽管有一定视角变换， BRIEF 和 SURF64 的匹配结果接近

第二行： BRIEF成功进行了闭环，但是SURF64 没有闭环。原因是SURF64 没有得到足够多的匹配关系。

第三行： BRIEF 闭环失败而SURF64闭环成功。

我们分析一下原因：主要和近景远景有关。因为BRIEF相比SURF64没有尺度不变性，所以在尺度变换较大的近景很容易匹配失败，比如第三行。而在中景和远景，由于尺度变化不大， BRIEF 表现接近甚至优于SURF64



不过，我们通过图像金字塔可以解决上述BRIEF的尺度问题。论文中作者也提到了ORB + BRIEF的特征点主要问题是**没有旋转不变性和尺度不变性**。不过目前都解决了。

总之，BRIEF的闭环效果值得信赖！



离线训练 vocabulary tree (也称为字典)

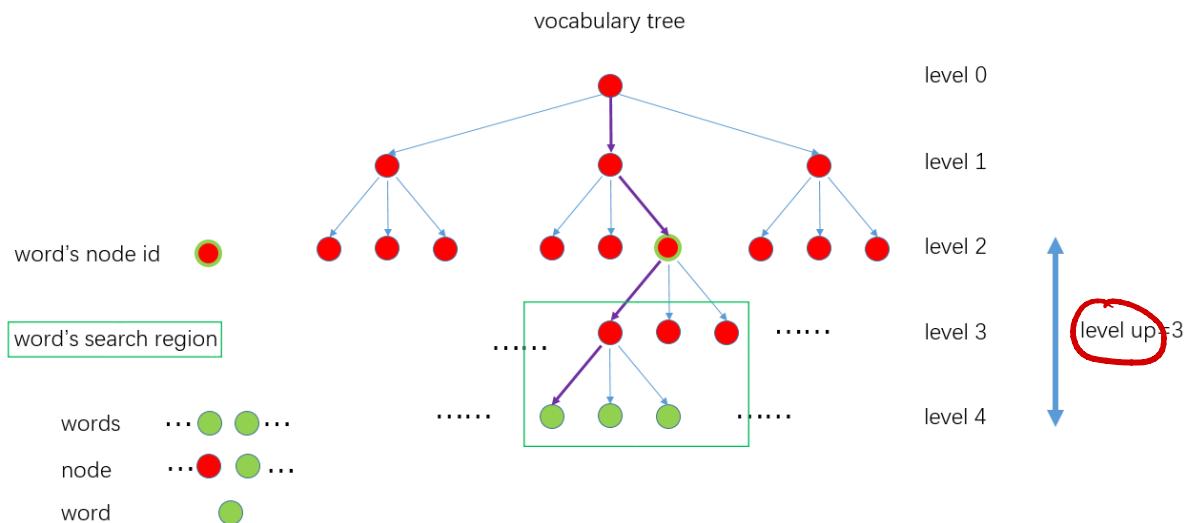
首先图像提取ORB 特征点，将描述子通过 k-means 进行聚类，根据设定的树的分支数和深度，从叶子节点开始聚类一直到根节点，最后得到一个非常大的 vocabulary tree，

- 1、遍历所有的训练图像，对每幅图像提取ORB特征点。
- 2、设定vocabulary tree的分支数K和深度L。将特征点的每个描述子用 K-means聚类，变成 K个集合，作为vocabulary tree 的第1层级，然后对每个集合重复该聚类操作，就得到了vocabulary tree的第2层级，继续迭代最后得到满足条件的vocabulary tree，它的规模通常比较大，比如ORB-SLAM2使用的离线字典就有108万+ 个节点。
- 3、离根节点最远的一层节点称为叶子或者单词 Word。根据每个Word 在训练集中的相关程度给定一个权重weight，训练集里出现的次数越多，说明辨别力越差，给与的权重越低。

在线图像生成BoW向量

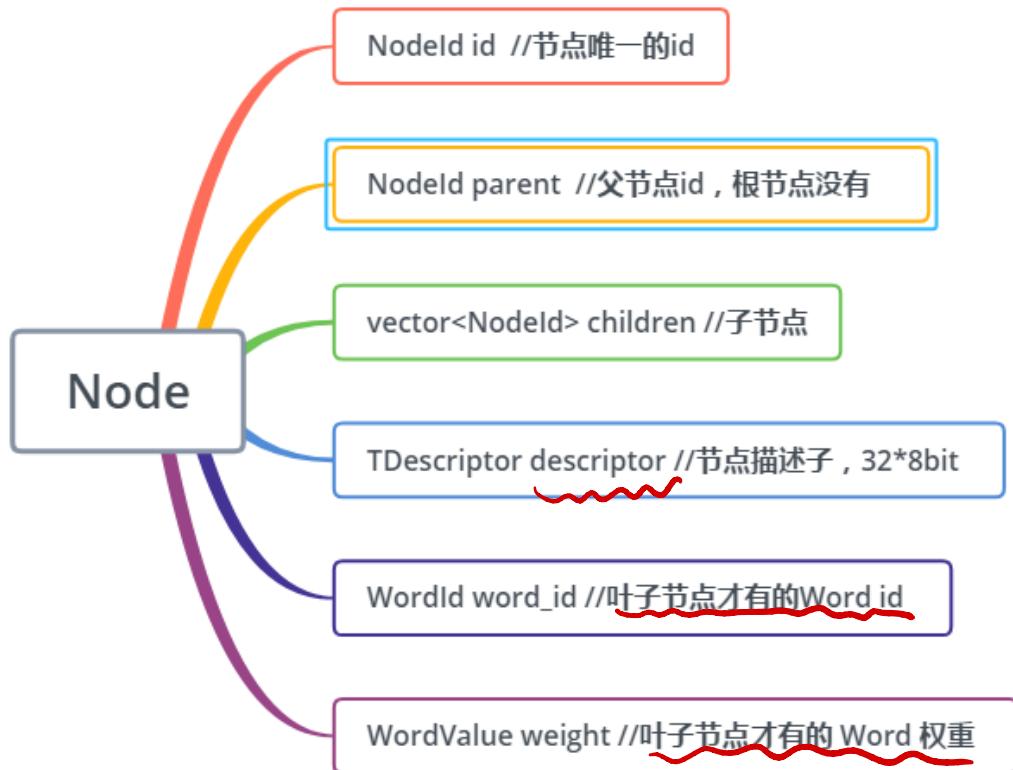
- 1、对新来的一帧图像进行ORB特征提取，得到一定数量（一般几百个）的特征点，描述子维度和 vocabulary tree中的一致
- 2、对于每个特征点的描述子，从离线创建好的vocabulary tree中开始找自己的位置，从根节点开始，用该描述子和每个节点的描述子计算汉明距离，选择汉明距离最小的作为自己所在的节点，一直遍历到叶子节点。

整个过程是这样的，见下图。紫色的线表示一个特征点从根节点到叶子节点的过程。



村民，村长，市长，省长类比

什么是节点 (node) ?



源码解析

一个描述子转化为Word id, Word weight, 节点所属的父节点 (距离叶子深度为level up深度的节点) id 对应的实现代码见:

```
/**  
 * @brief 将描述子转化为word id, word weight, 节点所属的父节点id (这里的父节点不是叶子的上一层, 它距离叶子深度为levelsUp)  
 *  
 * @tparam TDescriptor  
 * @tparam F  
 * @param[in] feature 特征描述子  
 * @param[in & out] word_id word id  
 * @param[in & out] weight word 权重  
 * @param[in & out] nid 记录当前描述子转化为word后所属的 node id, 它距离  
叶子深度为levelsUp  
 * @param[in] levelsUp 距离叶子的深度  
 */  
template<class TDescriptor, class F>  
void TemplatedVocabulary<TDescriptor, F>::transform(const TDescriptor &feature,  
    wordId &word_id, wordValue &weight, NodeId *nid, int levelsUp) const  
{  
    // propagate the feature down the tree  
    vector<NodeId> nodes;  
    typename vector<NodeId>::const_iterator nit;  
  
    // level at which the node must be stored in nid, if given  
    // m_L: depth levels, m_L = 6 in ORB-SLAM2  
    // nid_level 当前特征点转化为的word 所属的 node id, 方便索引
```

```

const int nid_level = m_L - levelsup;
if(nid_level <= 0 && nid != NULL) *nid = 0; // root

```

↑ root
↑ levelsup
↑ 叶子

```

NodeId final_id = 0; // root
int current_level = 0;

do
{
    ++current_level;
    nodes = m_nodes[final_id].children;
    final_id = nodes[0];

    // 取当前节点第一个子节点的描述子距离初始化最佳（小）距离
    double best_d = F::distance(feature, m_nodes[final_id].descriptor);
    // 遍历nodes中所有的描述子，找到最小距离对应的描述子
    for(nit = nodes.begin() + 1; nit != nodes.end(); ++nit)
    {
        NodeId id = *nit;
        double d = F::distance(feature, m_nodes[id].descriptor);
        if(d < best_d)
        {
            best_d = d;
            final_id = id;
        }
    }
    // 记录当前描述子转化为word后所属的node id，它距离叶子深度为levelsup
    if(nid != NULL && current_level == nid_level)
        *nid = final_id;
}

} while( !m_nodes[final_id].isLeaf() );

```

只要不是叶子，就一直寻找最小距离的Word

中间顺便把feature node即FP集的sub-root
标记一下。

```

// turn node id into word id
// 取出 vocabulary tree中node距离当前feature 描述子距离最小的那个node的 word id 和
weight
word_id = m_nodes[final_id].word_id;
weight = m_nodes[final_id].weight;
}

```

一幅图像里所有特征点转化为两个std::map容器 BowVector 和 FeatureVector 的代码见：

```

/**
* @brief 将一幅图像所有的特征点转化为BowVector和FeatureVector
*
* @tparam TDescriptor
* @tparam F
* @param[in] features      图像中所有的特征点
* @param[in & out] v        BowVector
* @param[in & out] fv       FeatureVector
* @param[in] levelsup      距离叶子的深度
*/
template<class TDescriptor, class F>
void TemplatedVocabulary<TDescriptor,F>::transform(
    const std::vector<TDescriptor>& features → 前帧所有描述子的集合
    BowVector &v, FeatureVector &fv, int levelsup) const
{
    v.clear();
    fv.clear();

    for (const auto& feature : features)
    {
        int current_level = 0;
        NodeId final_id = 0;
        NodeId nid_level = m_L - levelsup;
        if (nid_level <= 0 && feature->nid != NULL) feature->nid = 0; // root

        do
        {
            ++current_level;
            std::vector<Node*> children = feature->children;
            final_id = children[0];

            // 取当前节点第一个子节点的描述子距离初始化最佳（小）距离
            double best_d = F::distance(feature->descriptor, m_nodes[final_id].descriptor);
            // 遍历children中所有的描述子，找到最小距离对应的描述子
            for (auto it = children.begin() + 1; it != children.end(); ++it)
            {
                NodeId id = *it;
                double d = F::distance(feature->descriptor, m_nodes[id].descriptor);
                if (d < best_d)
                {
                    best_d = d;
                    final_id = id;
                }
            }
            // 记录当前描述子转化为word后所属的node id，它距离叶子深度为levelsup
            if (feature->nid != NULL && current_level == nid_level)
                feature->nid = final_id;
        }

        } while( !m_nodes[final_id].isLeaf() );
    }

    // turn node id into word id
    // 取出 vocabulary tree中node距离当前feature 描述子距离最小的那个node的 word id 和
    weight
    word_id = m_nodes[final_id].word_id;
    weight = m_nodes[final_id].weight;
}

```

对应的叶子节点向上寻找 feature node
的层数

```

fv.clear();

if(empty() // safe for subclasses
{
    return;
}

// normalize
// 根据选择的评分类型来确定是否需要将BowVector 归一化
LNorm norm;
bool must = m_scoring_object->mustNormalize(norm);

typename vector<TDescriptor>::const_iterator fit;

if(m_weighting == TF || m_weighting == TF_IDF)
{
    unsigned int i_feature = 0;
    // 遍历图像中所有的特征点
    for(fit = features.begin(); fit < features.end(); ++fit, ++i_feature)
    {
        WordId id;           // 叶子节点的word id
        NodeId nid;          // FeatureVector 里的NodeId, 用于加速搜索
        WordValue w;          // 叶子节点word对应的权重

        // 将当前描述子转化为word id, word weight, 节点所属的父节点id (这里的父节点不是叶子
        // 的上一层, 它距离叶子深度为levelsUp)
        if(w > 0) // not stopped
        {
            // 如果word 权重大于0, 将其添加到BowVector 和 FeatureVector
            v.addWeight(id, w);
            fv.addFeature(nid, i_feature);
        }
    }

    if(!v.empty() && !must)
    {
        // unnecessary when normalizing
        const double nd = v.size();
        for(BowVector::iterator vit = v.begin(); vit != v.end(); vit++)
            vit->second /= nd;
    }
}

```

第*n*个FP

if(w > 0) // not stopped

if(w > 0) // not stopped → 一个关键帧中每一个特征点的描述子
w → 对应的叶子节点的词频
if(w > 0) // not stopped → 词权重
v.addWeight(id, w); → feature node的id, 由叶子节点向上查levelsUp 层得到的节点id
fv.addFeature(nid, i_feature);

相当于将当前图像信息进行了压缩，并且对后面特征点快速匹配、闭环检测、重定位意义重大。

这两个容器非常重要，下面一个个来解释：

BowVector

它内部实际存储的是这个

```
std::map<WordId, WordValue>
```

其中 WordId 和 WordValue 表示Word在所有叶子中距离最近的叶子的id 和权重（后面解释）。

同一个Word id 的权重是累加更新的，见代码

```
void BowVector::addweight(WordId id, WordValue v)
{
    // 返回指向大于等于id的第一个值的位置
    BowVector::iterator vit = this->lower_bound(id);

    // http://www.cplusplus.com/reference/map/map/key_comp/
    if(vit != this->end() && !(this->key_comp()(id, vit->first)))
    {
        // 如果id = vit->first, 说明是同一个word, 权重更新
        vit->second += v;
    }
    else
    {
        // 如果该word id不在BowVector中, 新添加进来
        this->insert(vit, BowVector::value_type(id, v));
    }
}
```

一帧图像中 BowVector(词袋向量)
重复出现 = 该 word 更重要, weight 累加

FeatureVector

内部实际是个

```
std::map<NodeId, std::vector<unsigned int> >
```

其中NodeId 并不是该叶子节点直接的父节点id，而是距离叶子节点深度为level up对应的node 的id，
对应上面 vocabulary tree 图示里的 Word's node id。为什么不直接设置为父节点？因为后面搜索该
Word 的匹配点的时候是在和它具有同样node id下面所有子节点中的Word 进行匹配，搜索区域见图示
中的 Word's search region。所以搜索范围大小是根据level up来确定的，level up 值越大，搜索范围越
广，速度越慢；level up 值越小，搜索范围越小，速度越快，但能够匹配的特征就越少。

另外 std::vector 中实际存的是NodeId 下所有特征点在图像中的索引。见代码

```
void FeatureVector::addFeature(NodeId id, unsigned int i_feature)
{
    FeatureVector::iterator vit = this->lower_bound(id);
    // 将同样node id下的特征放在一个vector里
    if(vit != this->end() && vit->first == id)
    {
        vit->second.push_back(i_feature);
    }
    else
    {
        vit = this->insert(vit, FeatureVector::value_type(id,
            std::vector<unsigned int>()));
        vit->second.push_back(i_feature);
    }
}
```

如果已经有该 Feature Node
就将现在所在的特征点塞到加进去，
随后特征匹配会通过它，匹配 FeatureVec
里所有索引对应的点

```
}
```

FeatureVector主要用于不同图像特征点快速匹配，加速几何关系验证，比如
ORBmatcher::SearchByBoW 中是这样用的

```
DBow2::FeatureVector::const_iterator f1it = vFeatVec1.begin();  
DBow2::FeatureVector::const_iterator f2it = vFeatVec2.begin();  
DBow2::FeatureVector::const_iterator f1end = vFeatVec1.end();  
DBow2::FeatureVector::const_iterator f2end = vFeatVec2.end();  
  
while(f1it != f1end && f2it != f2end)  
{  
    // Step 1: 分别取出属于同一node的ORB特征点(只有属于同一node, 才有可能是匹配点)  
    if(f1it->first == f2it->first)  
        // Step 2: 遍历KF中属于该node的特征点  
        for(size_t i1=0, iend1=f1it->second.size(); i1<iend1; i1++)  
        {  
            const size_t idx1 = f1it->second[i1];  
  
            MapPoint* pMP1 = vpMapPoints1[idx1];  
  
            // 省略  
            // .....  
        }  
}
```

2.4 vocabulary tree 的保存和加载

如何保存训练好的 vocabulary tree 存储为txt文件？

```
template<class TDescriptor, class F>  
void TemplatedVocabulary<TDescriptor,F>::saveToFile(const std::string  
&filename) const  
{  
    fstream f;  
    f.open(filename.c_str(),ios_base::out);  
    // 第一行打印 树的分支数、深度、评分方式、权重计算方式  
    f << m_k << " " << m_L << " " << " " << m_scoring << " " << m_weighting <<  
    endl;  
  
    for(size_t i=1; i<m_nodes.size();i++)  
    {  
        const Node& node = m_nodes[i];  
        // 每行第1个数字为父节点id  
        f << node.parent << " ";  
        // 每行第2个数字标记是(1)否(0)为叶子(word)  
        if(node.isLeaf())  
            f << 1 << " ";  
        else  
            f << 0 << " ";  
        // 接下来存储256位描述子，最后存储节点权重  
        f << F::toString(node.descriptor) << " " << (double)node.weight << endl;  
    }  
  
    f.close();  
}
```

```
}
```

如何加载训练好的 vocabulary tree txt文件?

```
/**  
 * @brief 加载训练好的 vocabulary tree, txt格式  
 *  
 * @tparam TDescriptor  
 * @tparam F  
 * @param[in] filename          vocabulary tree 文件名称  
 * @return true                加载成功  
 * @return false               加载失败  
 */  
template<class TDescriptor, class F>  
bool TemplatdVocabulary<TDescriptor,F>::loadFromTextFile(const std::string  
&filename)  
{  
    ifstream f;  
    f.open(filename.c_str());  
  
    if(f.eof())  
        return false;  
  
    m_words.clear();  
    m_nodes.clear();  
  
    string s;  
    getline(f,s);  
    stringstream ss;  
    ss << s;  
    ss >> m_k;      // 树的分支数目  
    ss >> m_L;      // 树的深度  
    int n1, n2;  
    ss >> n1;  
    ss >> n2;  
  
    if(m_k<0 || m_k>20 || m_L<1 || m_L>10 || n1<0 || n1>5 || n2<0 || n2>3)  
    {  
        std::cerr << "Vocabulary loading failure: This is not a correct text  
file!" << endl;  
        return false;  
    }  
  
    m_scoring = (ScoringType)n1;      // 评分类型  
    m_weighting = (WeightingType)n2;  // 权重类型  
    createScoringObject();  
  
    // 总共节点（nodes）数，是一个等比数列求和  
    //! bug 没有包含最后叶子节点数，应该改为 ((pow((double)m_k, (double)m_L + 2) -  
1)/(m_k - 1))  
    //! 但是没有影响，因为这里只是reserve，实际存储是一步步resize实现  
    int expected_nodes =  
        (int)((pow((double)m_k, (double)m_L + 1) - 1)/(m_k - 1));  
    m_nodes.reserve(expected_nodes);  
    // 预分配空间给 单词（叶子）数
```

```

m_words.reserve(pow((double)m_k, (double)m_L + 1));

// 第一个节点是根节点, id设为0
m_nodes.resize(1);
m_nodes[0].id = 0;
while(!f.eof())
{
    string snode;
    getline(f,snode);
    stringstream ssnode;
    ssnode << snode;

    // nid 表示当前节点id, 实际是读取顺序, 从0开始
    int nid = m_nodes.size();
    // 节点size 加1
    m_nodes.resize(m_nodes.size() + 1);
    m_nodes[nid].id = nid;

    // 读每行的第一个数字, 表示父节点id
    int pid ;
    ssnode >> pid;
    // 记录节点id的相互父子关系
    m_nodes[nid].parent = pid;
    m_nodes[pid].children.push_back(nid);

    // 读取第二个数字, 表示是否是叶子 (word)
    int nIsLeaf;
    ssnode >> nIsLeaf;

    // 每个特征点描述子是256 bit, 一个字节对应8 bit, 所以一个特征点需要32个字节存储。
    // 这里 F::L=32, 也就是读取32个字节, 最后以字符串形式存储在ssd
    stringstream ssd;
    for(int iD=0;iD<F::L;iD++)
    {
        string sElement;
        ssnode >> sElement;
        ssd << sElement << " ";
    }
    // 将ssd存储在该节点的描述子
    F::fromString(m_nodes[nid].descriptor, ssd.str());

    // 读取最后一个数字: 节点的权重 (word才有)
    ssnode >> m_nodes[nid].weight;

    if(nIsLeaf>0)
    {
        // 如果是叶子 (word), 存储到m_words
        int wid = m_words.size();
        m_words.resize(wid+1);

        // 存储word的id, 具有唯一性
        m_nodes[nid].word_id = wid;
        // 构建 vector<Node*> m_words, 存储word所在node的指针
        m_words[wid] = &m_nodes[nid];
    }
    else
    {
        // 非叶子节点, 直接分配 m_k个分支
    }
}

```

```

        m_nodes[nid].children.reserve(m_k);
    }

    return true;

}

```

有几点需要说明：

K 表示树的分支数， L 表示树的深度，这里的深度不考虑根节点 K^0 ，是从根节点下面开始算总共有 L 层深度，最后叶子层总共有 K^{L+1} 个叶子 (Word)。

总共所有的节点数目是一个等比数列求和问题

等比数列前n项和

$$S_n = \frac{a_1 - a_n \times q}{1 - q}$$

最后所有的节点数目应该是

$$\frac{K^{L+2} - 1}{K - 1}$$

关于权重类型 和 评分类型

```

/// weighting type
enum WeightingType
{
    TF_IDF,           //0
    TF,               //1
    IDF,              //2
    BINARY            //3
};

/// scoring type
enum ScoringType
{
    L1_NORM,          //0
    L2_NORM,          //1
    CHI_SQUARE,       //2
    KL,               //3
    BHATTACHARYYA,   //4
    DOT_PRODUCT,      //5
};

```

有个地方需要注意：

ORB-SLAM2 中词典初始化时： k = 10, int L = 5, weighting = TF_IDF, scoring = L1_NORM

```

/**
 * Initiates an empty vocabulary
 * @param k branching factor
 * @param L depth levels
 * @param weighting weighting type
 * @param scoring scoring type
 */
TemplatedVocabulary(int k = 10, int L = 5,
    WeightingType weighting = TF_IDF, ScoringType scoring = L1_NORM);

// levelsup = 4
mpORBvocabulary->transform(vCurrentDesc, mBowVec, mFeatVec, 4);

```

但是实际上ORB-SLAM2 使用时是使用的加载的词典ORBvoc.txt

```

10 6 0 0
0 0 252 188 188 242 169 109 85 143 187 191 164 25 222 255 72 27 129 215 237 16 58
111 219 51 219 211 85 127 192 112 134 34 0
0 0 93 125 221 103 180 14 111 184 112 234 255 76 215 115 153 115 22 196 124 110
233 240 249 46 237 239 101 20 104 243 66 33 0
.....

```

所以真正使用的词典参数是：k = 10, int L = 6, weighting = TF_IDF, scoring = L1_NORM, levelsup=4

3. 用参考关键帧来跟踪

函数 Tracking::TrackReferenceKeyFrame()

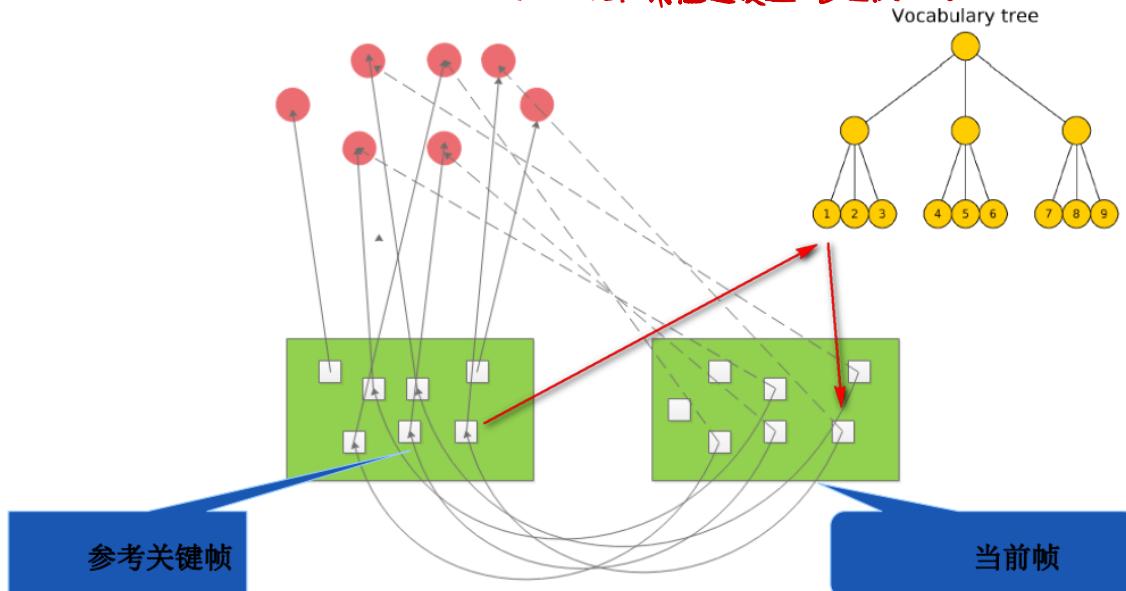
Vec <map> > featurevec

→ 普通帧跟踪失败，重定位 etc.

→ 用关键帧跟踪

otherwise. 用恒速模型(普通帧跟踪)

Vocabulary tree



应用场景：没有速度信息的时候、刚完成重定位、或者恒速模型跟踪失败后使用，大部分时间不用。只利用到了参考帧的信息。

1. 匹配方法是 SearchByBoW，匹配当前帧和关键帧在同一节点下的特征点，不需要投影，速度很快

2. BA优化（仅优化位姿），提供比较粗糙的位姿

思路：

当使用运动模式匹配到的特征点数较少时，就会选用关键帧模式跟踪。

思路是：尝试和最近一个关键帧去做匹配。为了快速匹配，利用了bag of words (BoW) 来加速匹配

具体流程

1 计算当前帧的BoW；

用 Featurevec

2 通过特征点的bow加快当前帧和参考帧之间的特征点匹配。使用函数matcher.SearchByBoW()。

- 对于同一node (同一node才可能是匹配点) 的特征点通过描述子距离进行匹配，遍历该node中特征点，特征点最小距离明显小于次小距离才作为成功匹配点，记录特征点对方向差统计到直方图
- 记录特征匹配成功后每个特征点对应的MapPoint (来自参考帧)，用于后续3D-2D位姿优化
- 通过角度投票进行剔除误匹配

普通帧中点的index 对应关键帧中匹配点的MapPoint → 存入vpMapPointMatches <vector>

3 将上一帧的位姿作为当前帧位姿的初始值（加速收敛），通过优化3D-2D的重投影误差来获得准确位姿。3D-2D来自第2步匹配成功的参考帧和当前帧，重投影误差 $e = (u, v) - \text{project}(T_{cw} * P_w)$ ，只优化位姿 T_{cw} ，不优化MapPoints的坐标。

顶点 Vertex: g2o::VertexSE3Expmap(), 初始值为上一帧的 T_{cw}

边 Edge (单目) : g2o::EdgeSE3ProjectXYZOnlyPose(), 一元边 BaseUnaryEdge

+ 顶点 Vertex: 待优化当前帧的 T_{cw}

+ 测量值 measurement: MapPoint在当前帧中的二维位置 (u, v)

+ 误差信息矩阵 InfoMatrix: Eigen::Matrix2d::Identity()*invSigma2 (与特征点所在的尺度有关)

+ 附加信息: 相机内参数: $e \rightarrow fx fy cx cy$

3d点坐标 : $e \rightarrow X_w[0] X_w[1] X_w[2]$ 2d点对应的上一帧的3d点

优化多次，根据边误差，更新2d-3d匹配质量内外点标记，当前帧设置优化后的位姿。

4 剔除优化后的outlier地图点

lower_bound(begin,end,num): 从数组的begin位置到end-1位置二分查找第一个大于或等于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。

upper_bound(begin,end,num): 从数组的begin位置到end-1位置二分查找第一个大于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。

例子

参考资料

4 关于g2o和图优化

图文教程：

[从零开始一起学习SLAM | 理解图优化，一步步带你看懂g2o代码](#)

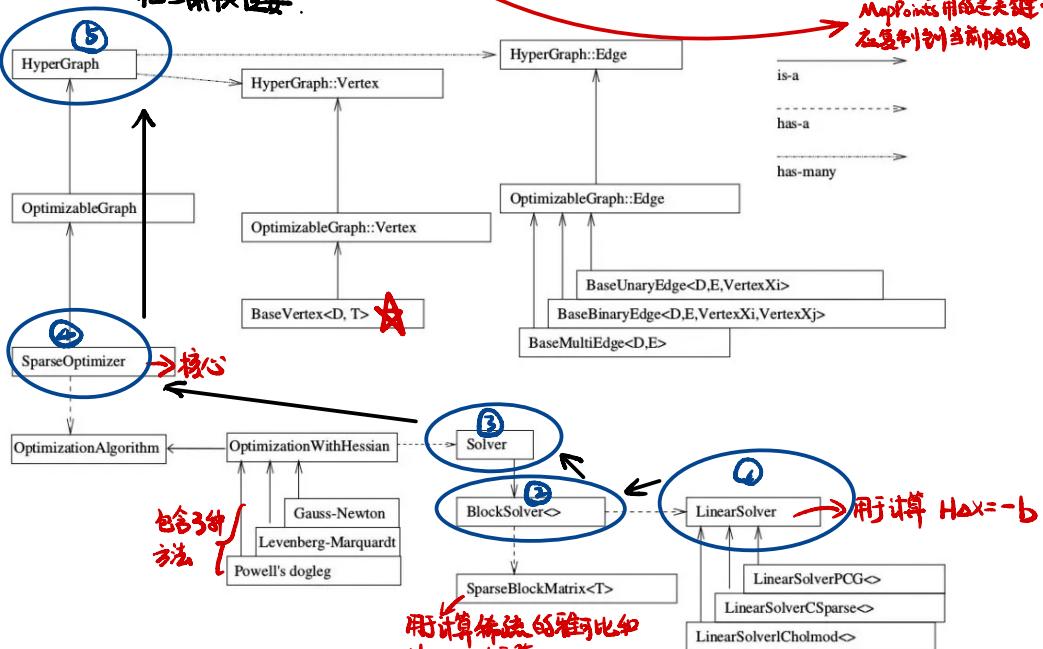
[从零开始一起学习SLAM | 掌握g2o顶点编程套路](#)

[从零开始一起学习SLAM | 掌握g2o边的代码套路](#)

G2O

刚才的 tracking 中有以下几个过程：1)用 featureVec 加速匹配，遍历关键帧与当前帧，取出匹配点，以 $\langle \text{MapPoint}, \text{index_of_current} \rangle$ 形式存储。2) 使用上一次 T_{cw} 为初值，重投影，通过用 g2o 优化当前帧位姿。

MapPoint 用的是关键帧
在复制到当前帧时



顶点的参数定义：① `int D, typename T`
在 manifold 下的 待估计 vertex
最小维度表示 基数类型

针对自定义顶点，需要：

- ① `setToOriginImpl`: 设定初值
变量的原始值。
- ② `oplusImpl`: 变量更新函数。

DRB 中使用的 `SE3Quat`: 内部用四元数表达旋转，加上位移存储。
同时支持李代数上的：对数映射(log)、指数映射(exp)运算。

添加顶点：① `setEstimate(type)` 设定初值

- ② `setId(int)` 定义节点编号
- ③ `setFixed()` 要优化的变量填 `false`

边：`BaseUnaryEdge`, `BaseBinaryEdge`, `BaseMultiEdge`

一元边 二元边 多元边

参数：`int D, typename T, vertexXi, vertexXj`
测量值维度 测量值类型 不同顶点类型

- 重写函数：
- ① `computeError`: 当前成本的计算的测量值与真实值的误差。
 - ② `linearizeOplus`: 误差对优化变量的偏导数。

- `measurement`: 有偏观测值
- `error`, 并指 `computeError` 计算的误差
- `vertices`: 存储顶点信息
- `setInformation()`: 未定义协方差矩阵的逆

```

/*
 * \brief SE3 Vertex parameterized internally with a transformation matrix
 * and externally with its exponential map
 */
class VertexSE3Expmap : public BaseVertex<D, SE3Quat>{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW

    VertexSE3Expmap();
    ~VertexSE3Expmap();

    bool read(std::istream& is);
    bool write(std::ostream& os) const;

    virtual void setToOriginImpl() {
        _estimate = SE3Quat();
    }

    virtual void oplusImpl(const double* update_) {
        Eigen::Map<const Vector3d> update(update_);
        setEstimate( et<SE3Quat>(exp(update)*estimate()));
    }

    virtual void linearizeOplus() {
        Eigen::Map<Vector3d> trans_xyz(trans_xyz());
        Eigen::Map<Vector3d> trans_xyw(trans_xyw());
        Eigen::Map<Vector3d> trans_xvw(trans_xvw());
    }
};

```

```

class EdgeSE3ProjectXYZOnlyPose : public BaseUnaryEdge<2, Vector2d, VertexSE3Expmap> {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW

    EdgeSE3ProjectXYZOnlyPose(){}
    ~EdgeSE3ProjectXYZOnlyPose(){}

    bool read(std::istream& is);
    bool write(std::ostream& os) const;

    void computeError() {
        const VertexSE3Expmap* v1 = static_cast<const VertexSE3Expmap*>(_vertices[0]);
        Vector2d obs(_measurement);
        _error = obs.cam_project(trans_xy_v1->estimate().map(xv_X));
    }

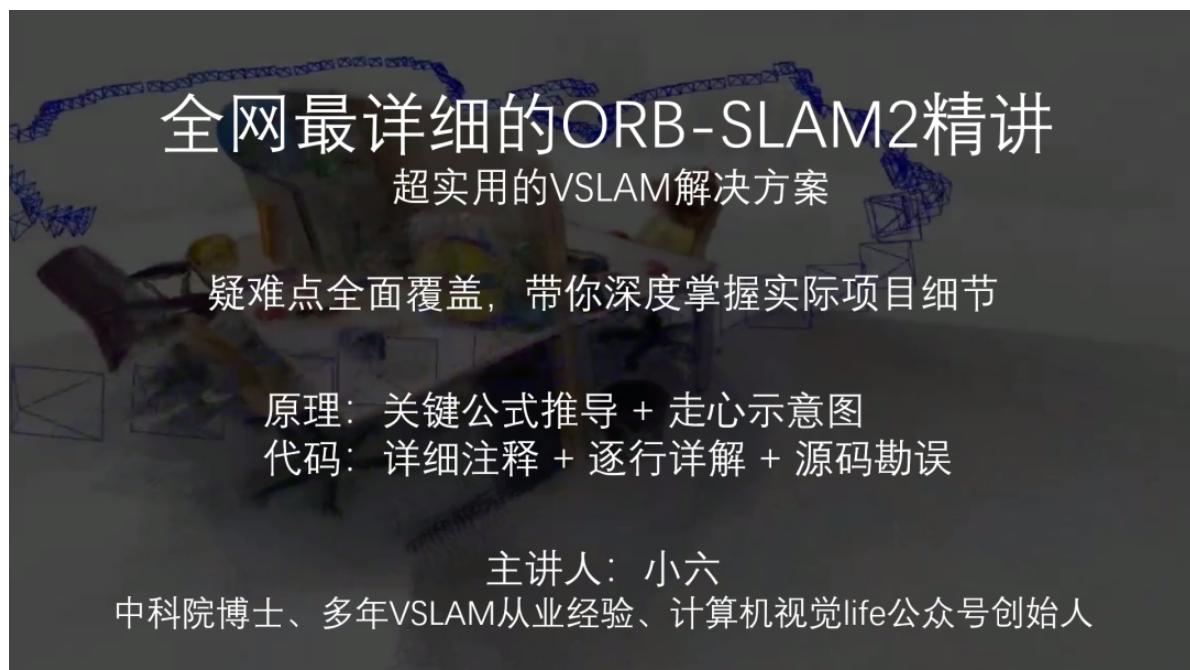
    bool isPositive() {
        const VertexSE3Expmap* v1 = static_cast<const VertexSE3Expmap*>(_vertices[0]);
        return (v1->estimate().map(xv_X)).index(2)>0.0;
    }

    virtual void linearizeOplus() {
        Vector2d cam_project(const Vector3d& trans_xyz) const;
        Vector3d Xw;
        double fx, fy, cx, cy;
    }
};

```

参考资料

[《全网最详细的ORB-SLAM2精讲：原理推导+逐行代码分析》](#) (点击可跳转课程详情)



长按或扫描二维码查看课程介绍和购买方式：

