list容器

---

list容器知识

list是一种序列式容器，其完成的功能实际上和数据结构中的双向链表很相似，具备链表的主要优点，即：在链表的任一位置进行元素的插入、删除操作都是快速的。list的每个节点有三个域：前驱元素指针域、数据域和后继元素指针域。

常用序列式容器

vector：和数组类似，拥有一段连续的内存空间，能非常好的支持随机存取，即[]操作符，但由于是连续的内存空间，在中间的插入和删除会导致内存块的拷贝。

list：双向链表，内存空间不连续的，通过指针访问数据。

deque：支持[]操作符，也支持随机存取。（不常用，也不常见）

三者比较：

1. 如果你需要高效的随即存取，而不在乎插入和删除的效率，使用vector
2. 如果你需要大量的插入和删除，而不关心随即存取，则应使用list
3. 如果你需要随即存取，而且关心两端数据的插入和删除，则应使用deque。

list常用函数

- begin()和end()：通过调用list容器的成员函数begin()得到一个指向容器起始位置的iterator，可以调用list容器的 end() 函数来得到list末端下一位置，相当于：int a[n]中的第n+1个位置a[n]，实际上是不存在的，不能访问，经常作为循环结束判断结束条件使用。
- push_back() 和push_front()：使用list的成员函数push_back和push_front插入一个元素到list中。其中push_back()从list的末端插入，而 push_front()实现的从list的头部插入。
- empty()：利用empty() 判断list是否为空。
- resize()：将list长度改为只容纳n个元素，超出的元素将被删除。
- clear()：清空list中所有元素。

- `front()`和`back()`：通过`front()`可以获得`list`容器中的头部元素，通过`back()`可以获得`list`容器的最后一个元素。但是有一点要注意，就是list中元素是空的时候，这时候调用`front()`和`back()`会发生什么呢？实际上会发生不能正常读取数据的情况，但是这并不报错，那我们编程序时就要注意了，个人觉得在使用之前最好先调用`empty()`函数判断`list`是否为空。
- `pop_back()`和`pop_front()`：通过`pop_back()`删除最后一个元素，通过`pop_front()`删除第一个元素。序列必须不为空，如果当`list`为空的时候调用`pop_back()`和`pop_front()`会使程序崩掉。
- `assign()`：具体和`vector`中的操作类似，也是有两种情况，第一种是：`l1.assign(n,val)`将`l1`中元素变为`n`个`T(val）`。第二种是：`l1.assign(l2.begin(),l2.end())`将`l2`中的从`l2.begin()`到`l2.end()`之间的数值赋值给`l1`。
- `swap()`：交换两个链表(两个重载)，一个是`l1.swap(l2)`;另外一个是`swap(l1,l2)`，都可能完成连个链表的交换。
- `reverse()`：完成`list`的逆置。
- `merge()`：合并两个链表并使之默认升序(也可改)，`l1.merge(l2，greater<int>())`;调用结束后`l2`变为空，`l1`中元素包含原来`l1`和`l2`中的元素，并且排好序，升序。其实默认是升序，`greater<int>()`可以省略，另外`greater<int>()`是可以变的，也可以不按升序排列。
- `insert()`：在指定位置插入一个或多个元素(三个重载)：
    a. `l1.insert(l1.begin(),100);`在`l1`的开始位置插入`100`。
    b. `l1.insert(l1.begin(),2,200);`在`l1`的开始位置插入`2`个`100`。
    c. `l1.insert(l1.begin(),l2.begin(),l2.end());`在`l1`的开始位置插入`l2`的从开始到结束的所有位置的元素。
- `erase()`：删除一个元素或一个区域的元素(两个重载)：
    a. `l1.erase(l1.begin());`将`l1`的第一个元素删除。
    b. `l1.erase(l1.begin(),l1.end());`将`l1`的从`begin()`到`end()`之间的元素删除。

`map`容器

---

map映射容器的元素数据是由一个键值和一个映射数据组成的，键值与映照数据之间具有一一映照的关系。map容器的数据结构也采用红黑树来实现的，插入元素的键值不允许重复，比较函数只对元素的键值进行比较，元素的各项数据可通过键值检索出来。

map 映照容器元素的数据构成示意图

```
begin() 返回指向 map 头部的迭代器
clear() 删除所有元素
count() 返回指定元素出现的次数
empty() 如果 map 为空则返回 true
end() 返回指向 map 末尾的迭代器
erase() 删除一个元素
find() 查找一个元素
insert() 插入元素
key_comp() 返回比较元素 key 的函数
lower_bound() 返回键值>=给定元素的第一个位置
max_size() 返回可以容纳的最大元素个数
rbegin() 返回一个指向 map 尾部的逆向迭代器
rend() 返回一个指向 map 头部的逆向迭代器
size() 返回 map 中元素的个数
swap() 交换两个 map
upper_bound() 返回键值>给定元素的第一个位置
value_comp() 返回比较元素 value 的函数
```

vector容器

```
1.push_back 在数组的最后添加一个数据

2.pop_back 去掉数组的最后一个数据

3.at 得到编号位置的数据

4.begin 得到数组头的指针

5.end 得到数组的最后一个单元+1的指针

6.front 得到数组头的引用

7.back 得到数组的最后一个单元的引用

8.max_size 得到vector最大可以是多大
```

```
9.capacity 当前vector分配的大小

10.size 当前使用数据的大小

11.resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值

12.reserve 改变当前vecotr所分配空间的大小

13.erase 删除指针指向的数据项

14.clear 清空当前的vector

15.rbegin 将vector反转后的开始指针返回(其实就是原来的end-1)

16.rend 将vector反转构的结束指针返回(其实就是原来的begin-1)

17.empty 判断vector是否为空

18.swap 与另一个vector交换数据
```

`LocalMapping::ProcessNewKeyFrame()` 函数

需要处理的变量：私有成员变量 `std::list<KeyFrame*> mlNewKeyFrames;`

数据来源：`Tracking::createNewKeyFrame()`函数调用了局部地图中 `LocalMapping::InsertKeyFrame(KeyFrame *pKF)` 此函数，将关键帧插入到列表中，同时将终止局部BA的标志置为true。

更新的信息：

私有成员变量 `KeyFrame* mpCurrentKeyFrame;` 当前正在处理的关键帧，更新此关键帧在tracking过程跟踪到的地图点属性以及其与其他关键帧的连接关系；

`std::list<MapPoint*> mlpRecentAddedMapPoints;` 待检验的地图点；

`Map* mpMap;` 更新局部地图；

步骤1：获取关键帧

涉及多线程操作关键帧列表，因此需要加锁。

利用list容器的常用函数，获取关键帧，并将已获取的关键帧从列表中移除。

步骤2：计算关键帧特征点的BoW映射关系

从当前帧的描述子中得到对应的词袋向量和节点的特征向量和特征索引。

1. 将描述子矩阵类型(`cv::mat`)转换为描述子向量类型(`std::vector`)；
2. 通过词袋模型得到词袋向量和节点的特征向量和特征索引。

步骤3：关联地图点和新的关键帧，更新方向和描述子

1. 获取当前关键帧中的地图点。`TrackLocalMap`函数将局部地图中的`MapPoints`与当前帧进行了匹配，但是没有对这些匹配上的`MapPoints`与当前帧进行关联；
2. 对所有`MapPoints`展开遍历，确认地图点存在，判断地图点是否是坏点；
3. 如果当前关键帧能观测到的地图点添加到待检验的地图点列表中，如果当前关键帧不能观测到地图点，说明此地图点是经过检验的，因此需要更新此地图点的属性（共视关系、平均观测方向和观测距离、最佳描述子）；

确认当前关键帧是否能观测到此地图点

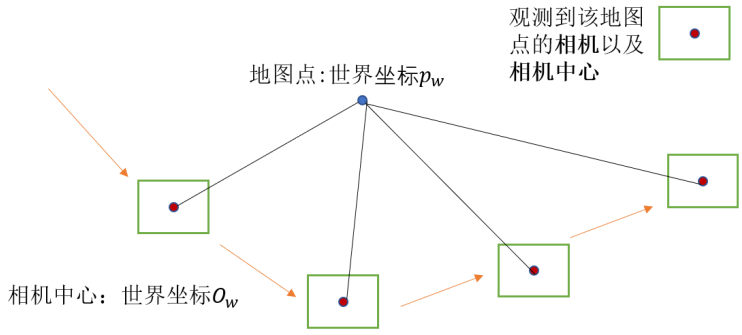
利用`map`容器的特点，`mObservations.count(关键帧)`函数检测地图点是否在关键帧中。

添加观测

将此关键帧添加到`map`容器`mObservations`中，更新地图点被观测到的次数。

更新平均观测方向和观测距离范围

平均观测方向的计算：



平均观测方向normal $= {}^{1}/_{K}\sum_{i=1}^{K}\frac{O_{wi}-p_w}{\|O_{wi}-p_w\|}$ K = 观测到该地图点的关键帧数量

观测距离范围的计算：（利用图像金字塔的原理）

在orb_slam2中，为了实现特征尺度不变性采用了图像金字塔，金字塔的缩放因子为1.2。其思路就是对原始图形（第0层）依次进行1/1.2缩放比例进行降采样得到共计8张图片（包括原始图像），然后分别对得到的图像进行特征提取，并记录特征所在金字塔的第几层，这样得到一帧图像的特征点。

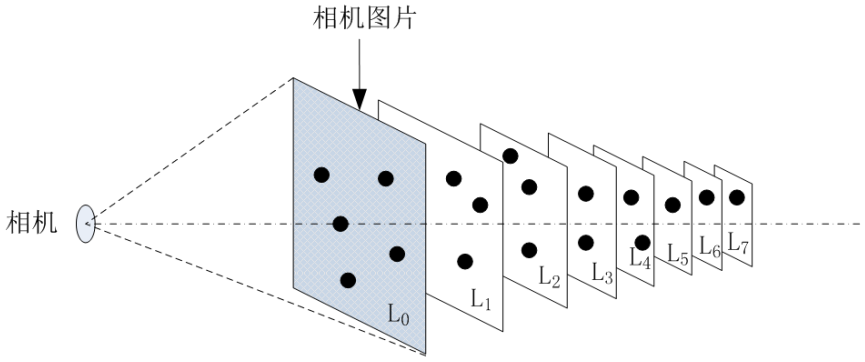


图 1 图像特征提取金字塔

现在假设在第二层中有一特征点F，为了避免缩放带来特征点F在纵向的移动，为简化叙述，选择的特征点F位于图像中心。根据相机成像“物近像大，物远像小”的原理，。假设摄像机原始图像即金字塔第0层对应成像视野I_0，则图像金字塔第2层图像可以相应对应于成像视野I_2。

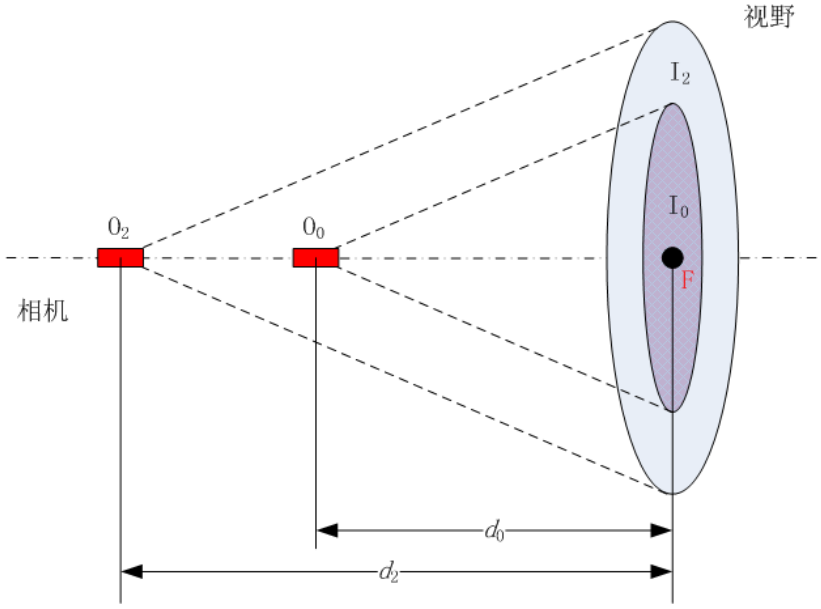


图 2 摄像机成像与距离场景距离之间的关系

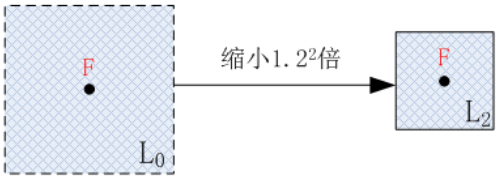


图 3 特征点 patch 缩放前后示意图

其中特征点F所在patch的相应关系：$d_2/d_0 = 1.2^2$。

对于第m层上的一个特征点，其对应尺度不变时相机与特征点对应空间位置之间距离（简称物距）的范围。

假设第m层上有一特征点F_m，其空间位置与拍摄时相机中心的位置为d_m，显然这是原始图像缩放$1/1.2^m$倍后得到的特征点patch，考虑“物远像小”的成像特点，要使得该第m层特征点对应patch变为图像金字塔第0层中同样大小的patch，其相机与空间点的距离$d = d_m * 1.2^m$,即尺度不变的最大物距$d_{max} = d_m * 1.2^m$。

要求尺度不变的最小物距则这样考虑：根据“物近像大”的成像特点，使得当前第

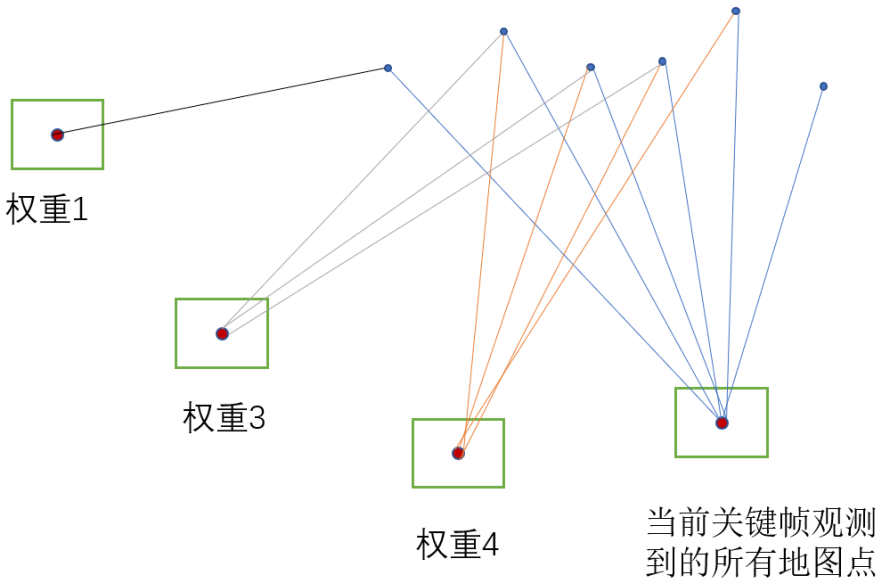
m层的特征点移到第7层上则，真实相机成像图像得放大1.2^{7-m}倍，故对应最小物距$d_{min} = d_m * 1.2^{m-7} = d_{max}/1.2^{金字塔层数-1}$。

更新3D点的最佳描述子

由于一个`MapPoint`会被许多相机观测到，因此在插入关键帧后，需要判断是否更新当前点的最适合的描述子。先获得当前点的所有描述子，然后计算描述子之间的两两距离，最好的描述子与其他描述子应该具有最小的距离中值。

步骤4：更新关键帧间的连接关系(不太明白树的作用)

1. 首先获得该关键帧的所有`MapPoint`点，统计观测到这些3d点的每个关键帧与其它所有关键帧之间的共视程度。对每一个找到的关键帧，建立一条边，边的权重是该关键帧与当前关键帧公共3d点的个数。
2. 并且该权重必须大于一个阈值，如果没有超过该阈值的权重，那么就只保留权重最大的边（与其它关键帧的共视程度比较高）。
3. 对这些连接按照权重从大到小进行排序，以方便将来的处理。更新完`covisibility`图之后，如果没有初始化过，则初始化为连接权重最大的边（与其它关键帧共视程度最高的那个关键帧），类似于最大生成树。



步骤5：将关键帧插入到地图中

用于建图。