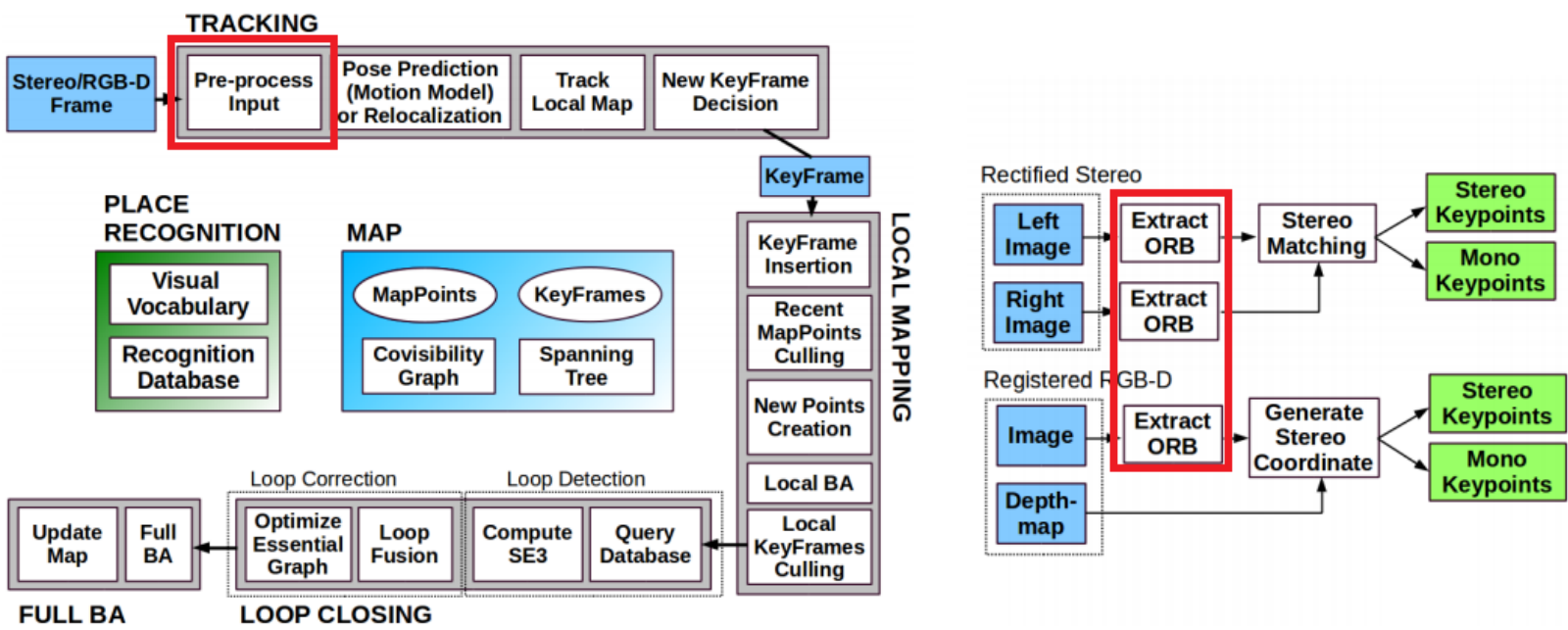


各成员函数/变量

- 构造函数: `ORBextractor()`
- 构建图像金字塔: `ComputePyramid()`
- 提取特征点并进行筛选: `ComputeKeyPointsOctTree()`
 - 八叉树筛选特征点: `DistributeOctTree()`
 - 计算特征点方向 `computeOrientation()`
 - 计算特征点描述子 `computeOrbDescriptor()`

ORBextractor 类的用途

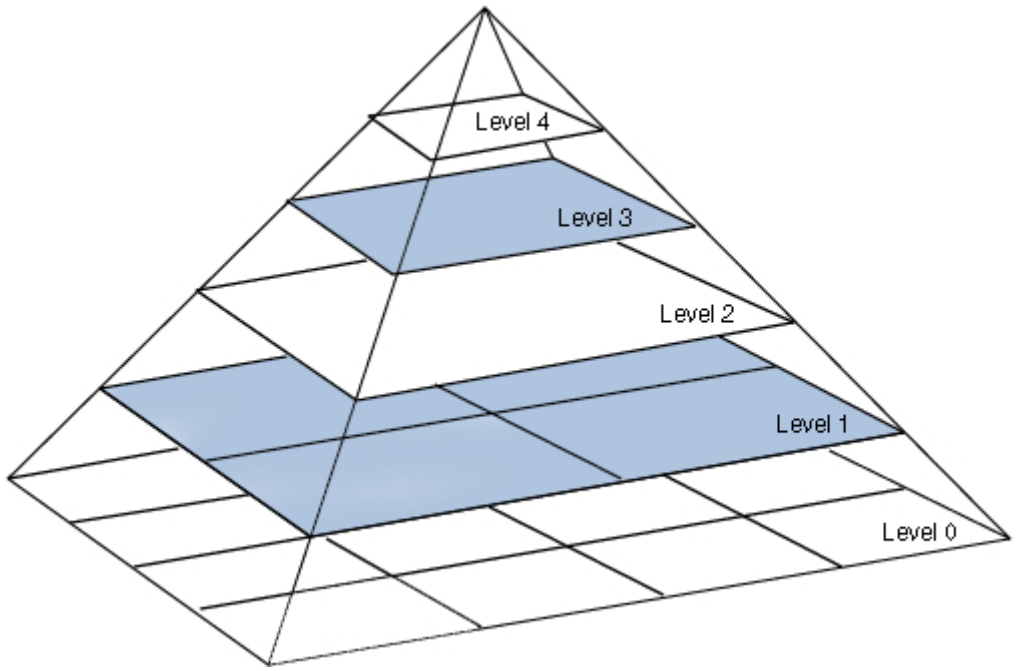
- ORBextractor 类提取特征点的主函数 `void operator()()`
- ORBextractor 类与其它类间的关系



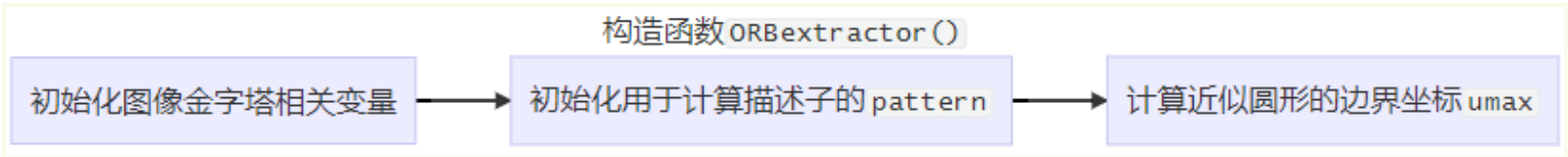
各成员函数/变量

构造函数: ORBextractor()

FAST 特征点和 ORB 描述子本身不具有尺度信息, ORBextractor 通过构建图像金字塔来得到特征点尺度信息.将输入图片逐级缩放得到图像金字塔,金字塔层级越高,图片分辨率越低,ORB特征点越大.



构造函数 `ORBextractor(int nfeatures, float scaleFactor, int nlevels, int iniThFAST, int minThFAST)` 的流程:



1. 初始化图像金字塔相关变量:

下面成员变量从配置文件 `TUM1.yaml` 中读入:

成员变量	访问控制	意义	配置文件 <code>TUM1.yaml</code> 中变量名	值
<code>int nfeatures</code>	<code>protected</code>	所有层级提取到的特征点数之和金字塔层数	<code>ORBextractor.nFeatures</code>	1000
<code>double scaleFactor</code>	<code>protected</code>	图像金字塔相邻层级间的缩放系数	<code>ORBextractor.scaleFactor</code>	1.2
<code>int nlevels</code>	<code>protected</code>	金字塔层级数	<code>ORBextractor.nLevels</code>	8
<code>int iniThFAST</code>	<code>protected</code>	提取特征点的描述子门槛(高)	<code>ORBextractor.iniThFAST</code>	20
<code>int minThFAST</code>	<code>protected</code>	提取特征点的描述子门槛(低)	<code>ORBextractor.minThFAST</code>	7

根据上述变量的值计算出下述成员变量:

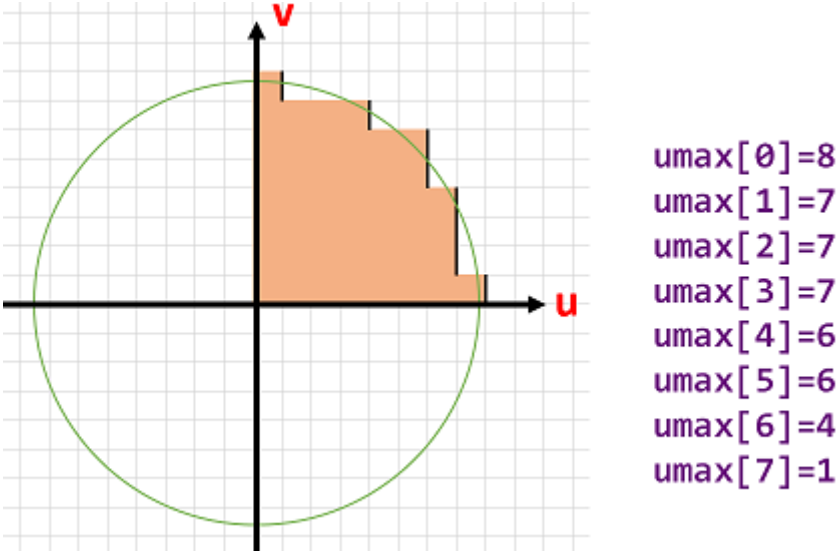
成员变量	访问控制	意义	值
<code>std::vector<int></code> <code>mnFeaturesPerLevel</code>	<code>protected</code>	金字塔每层级中提取的特征点数 正比于图层边长,总和为 <code>nfeatures</code>	<code>{61, 73, 87, 105, 126, 151, 181, 216}</code>
<code>std::vector<float></code> <code>mvScaleFactor</code>	<code>protected</code>	各层级的缩放系数	<code>{1, 1.2, 1.44, 1.728, 2.074, 2.488, 2.986, 3.583}</code>
<code>std::vector<float></code> <code>mvInvScaleFactor</code>	<code>protected</code>	各层级缩放系数的倒数	<code>{1, 0.833, 0.694, 0.579, 0.482, 0.402, 0.335, 0.2791}</code>
<code>std::vector<float></code> <code>mvLevelSigma2</code>	<code>protected</code>	各层级缩放系数的平方	<code>{1, 1.44, 2.074, 2.986, 4.300, 6.190, 8.916, 12.838}</code>
<code>std::vector<float></code> <code>mvInvLevelSigma2</code>	<code>protected</code>	各层级缩放系数的平方 倒数	<code>{1, 0.694, 0.482, 0.335, 0.233, 0.162, 0.112, 0.078}</code>

2. 初始化用于计算描述子的 `pattern` 变量, `pattern` 是用于计算描述子的 256 对坐标,其值写死在源码文件 `ORBextractor.cc` 里,在构造函数里做类型转换将其转换为 `const cv::Point*` 变量.

```
1 static int bit_pattern_31_[256*4] ={
2     8,-3, 9,5/*mean (0), correlation (0)*/,
3     4,2, 7,-12/*mean (1.12461e-05), correlation (0.0437584)*/,
4     -11,9, -8,2/*mean (3.37382e-05), correlation (0.0617409)*/,
5     7,-12, 12,-13/*mean (5.62303e-05), correlation (0.0636977)*/,
6     2,-13, 2,12/*mean (0.000134953), correlation (0.085099)*/,
7     // 共256行...
8 }
9
10 const Point* pattern0 = (const Point*)bit_pattern_31_;
11 std::copy(pattern0, pattern0 + npoints, std::back_inserter(pattern));
```

3. 计算一个半径为 16 的圆的近似坐标

后面计算的是**特征点主方向**上的描述子,计算过程中要将特征点周围像素旋转到主方向上,因此计算一个半径为 16 的圆的近似坐标,用于后面计算描述子时进行旋转操作.



成员变量 `std::vector<int>` `u_max` 里存储的实际上是逼近圆的**第一象限**内 $\frac{1}{4}$ 圆周上**每个 v 坐标对应的 u 坐标**.为保证严格对称性,先计算下 45° 圆周上点的坐标,再根据对称性补全上 45° 圆周上点的坐标.

```
1 int vmax = cvFloor(HALF_PATCH_SIZE * sqrt(2.f) / 2 + 1); // 45°射线与圆周交点的纵坐标
2 int vmin = cvCeil(HALF_PATCH_SIZE * sqrt(2.f) / 2); // 45°射线与圆周交点的纵坐标
3
4 // 先计算下半45度的u_max
5 for (int v = 0; v <= vmax; ++v) {
6     u_max[v] = cvRound(sqrt(15 * 15 - v * v));
7 }
8
9 // 根据对称性补出上半45度的u_max
10 for (int v = HALF_PATCH_SIZE, v0 = 0; v >= vmin; --v) {
11     while (u_max[v0] == u_max[v0 + 1])
12         ++v0;
13     u_max[v] = v0;
14     ++v0;
15 }
```

构建图像金字塔: `ComputePyramid()`

根据上述变量的值计算出下述成员变量:

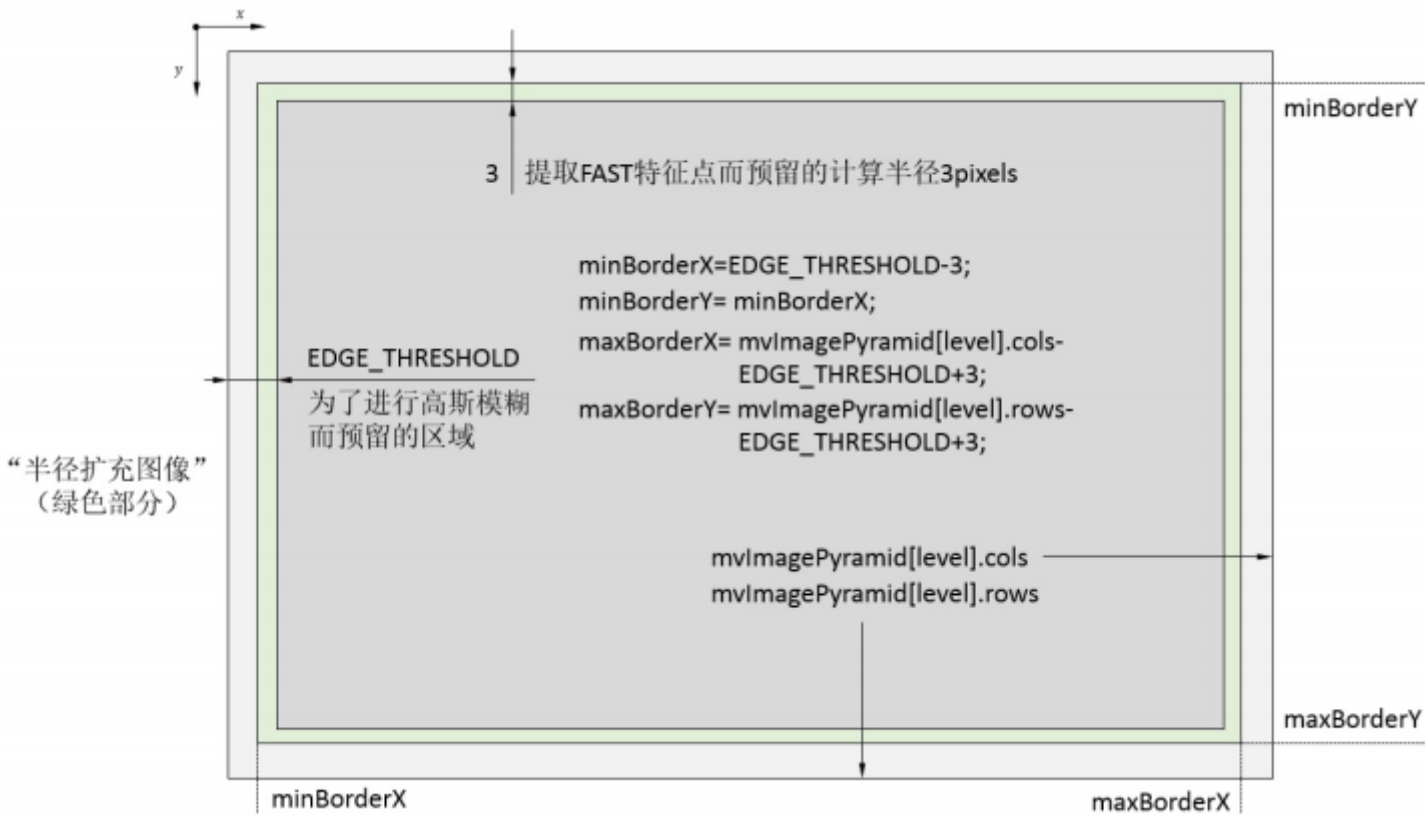
成员变量	访问控制	意义
<code>std::vector<cv::Mat> mvImagePyramid</code>	<code>public</code>	图像金字塔每层的图像
<code>const int EDGE_THRESHOLD</code>	全局变量	为计算描述子和提取特征点补的 padding 厚度

函数 `void ORBextractor::ComputePyramid(cv::Mat image)` 逐层计算图像金字塔,对于每层图像进行以下两步:

1. 先进行图片缩放,缩放到 `mvInvScaleFactor` 对应尺寸.
2. 在图像外补一圈厚度为 19 的 padding (提取 FAST 特征点需要特征点周围半径为 3 的圆域,计算 ORB 描述子需要特征点周围半径为 16 的圆域).

下图表示图像金字塔每层结构:

- 深灰色为缩放后的原始图像.
- 包含绿色边界在内的矩形用于提取 FAST 特征点.
- 包含浅灰色边界在内的整个矩形用于计算 ORB 描述子.



```
1 void ORBextractor::ComputePyramid(cv::Mat image) {
2     for (int level = 0; level < nlevels; ++level) {
3         // 计算缩放+补padding后该层图像的尺寸
4         float scale = mvInvScaleFactor[level];
5         Size sz(cvRound((float)image.cols*scale), cvRound((float)image.rows*scale));
6         Size wholeSize(sz.width + EDGE_THRESHOLD * 2, sz.height + EDGE_THRESHOLD * 2);
7         Mat temp(wholeSize, image.type());
8
9         // 缩放图像并复制到对应图层并补边
10        mvImagePyramid[level] = temp(Rect(EDGE_THRESHOLD, EDGE_THRESHOLD, sz.width, sz.height));
11        if( level != 0 ) {
12            resize(mvImagePyramid[level-1], mvImagePyramid[level], sz, 0, 0, cv::INTER_LINEAR);
13            copyMakeBorder(mvImagePyramid[level], temp, EDGE_THRESHOLD, EDGE_THRESHOLD, EDGE_THRESHOLD,
EDGE_THRESHOLD,
14                            BORDER_REFLECT_101+BORDER_ISOLATED);
15        } else {
16            copyMakeBorder(image, temp, EDGE_THRESHOLD, EDGE_THRESHOLD, EDGE_THRESHOLD, EDGE_THRESHOLD,
17                            BORDER_REFLECT_101);
18        }
19    }
20 }
```

`copyMakeBorder` 函数实现了复制和 padding 填充,其参数 `BORDER_REFLECT_101` 参数指定对padding进行镜像填充.

§ copyMakeBorder()

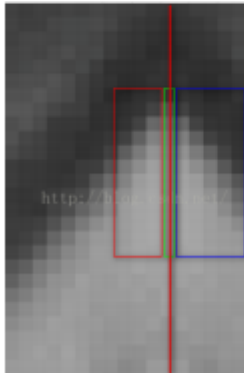
```
void cv::copyMakeBorder ( InputArray src,
                        OutputArray dst,
                        int top,
                        int bottom,
                        int left,
                        int right,
                        BorderType borderType,
                        const Scalar & value = Scalar() )
{
    ...
}

Python:
dst = cv.copyMakeBorder( src, top, bottom, left, right, borderType[, dst[, value]] )
```

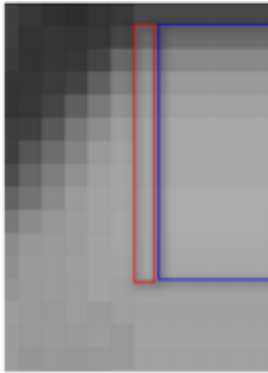
Forms a border around an image.

The function copies the source image into the middle of the destination image. The areas to the left, to the right, above and below the copied source image will be filled with extrapolated pixels. This is not what filtering functions based on it do (they extrapolate pixels on-fly), but what other more complex functions, including your own, may do to simplify image boundary handling.


The function supports the mode when src is already in the middle of dst. In this case, the function does not copy src itself but simply constructs the border.



BORDER_REFLECT_101

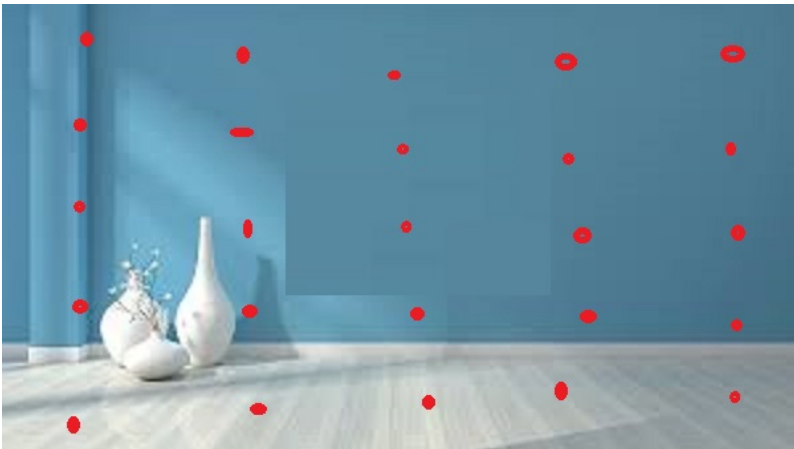


BORDER_REPLICATE

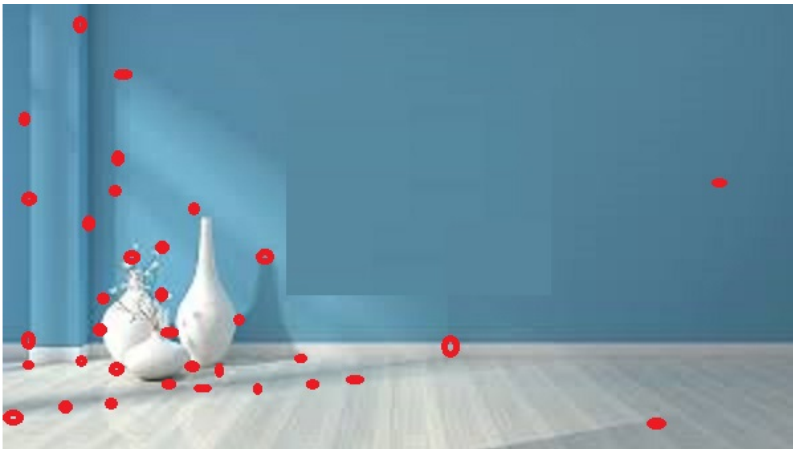


BORDER_CONSTANT

提取特征点并进行筛选: ComputeKeyPointsOctTree()



特征点分布均匀,好

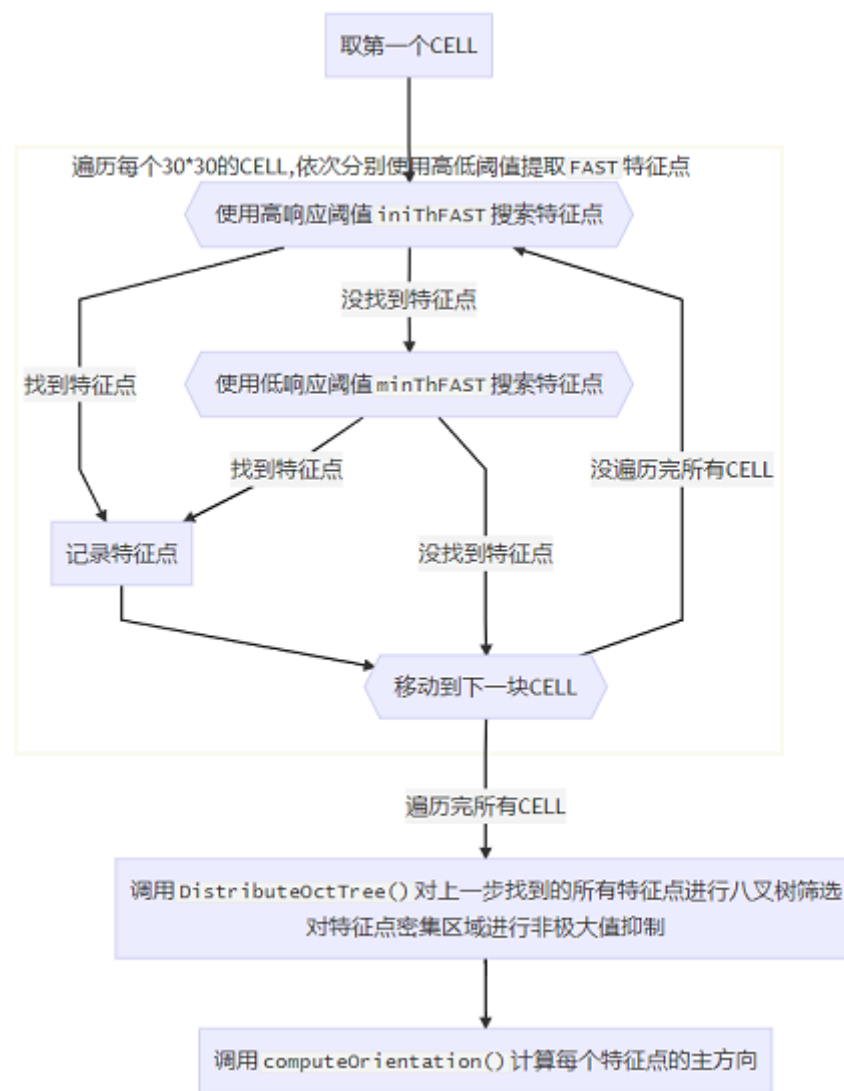


特征点分布不均匀,不好

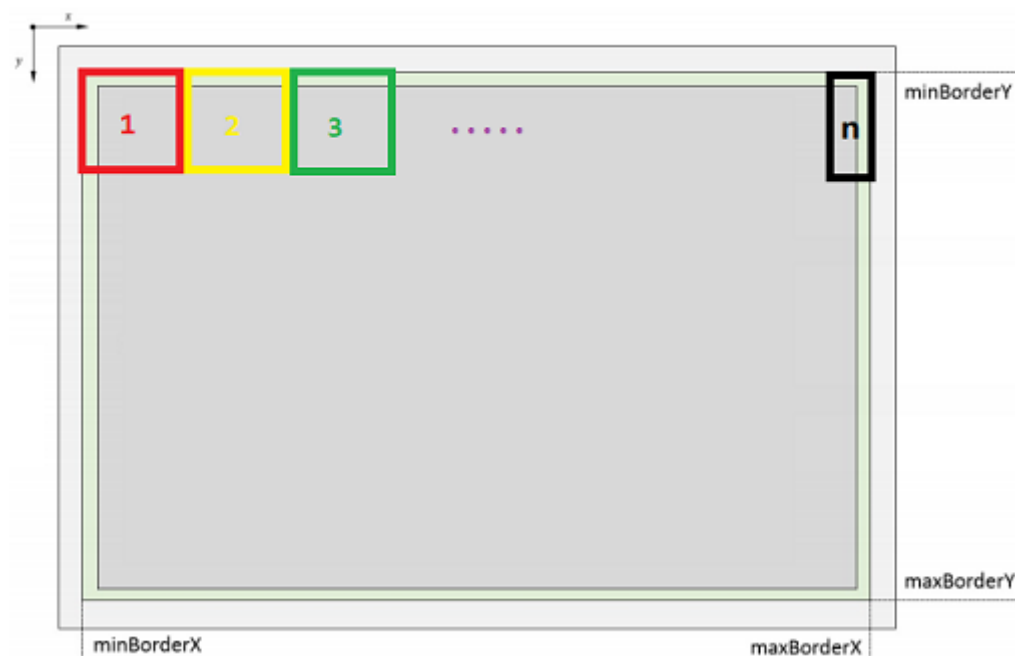
提取特征点最重要的就是力求特征点均匀地分布在图像的所有部分,为实现这一目标,编程实现上使用了两个技巧:

1. 分 CELL 搜索特征点,若某 CELL 内特征点响应值普遍较小的话就降低分数线再搜索一遍.
2. 对得到的所有特征点进行八叉树筛选,若某区域内特征点数目过于密集,则只取其中响应值最大的那个.

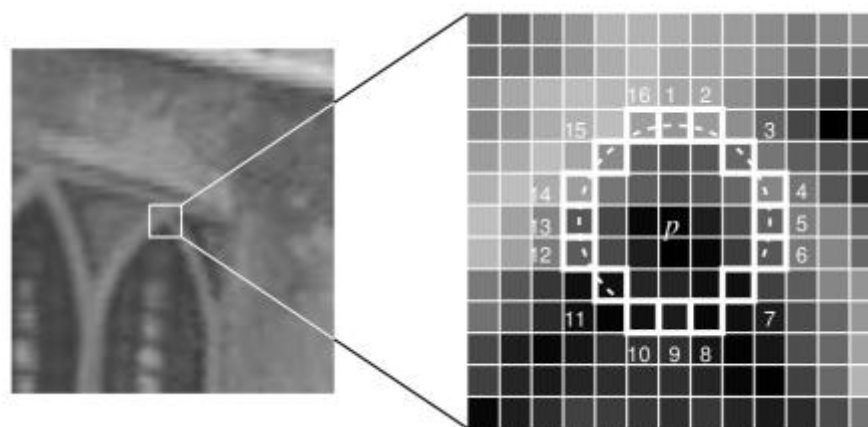




CELL 搜索的示意图如下,每个 CELL 的大小约为 30×30 ,搜索到边上,剩余尺寸不够大的时候,最后一个 CELL 有多大就用多大的区域。



需要注意的是相邻的 CELL 之间会有 6 像素的重叠区域,因为提取 FAST 特征点需要计算特征点周围半径为 3 的圆周上的像素点信息,实际上产生特征点的区域比传入的搜索区域小 3 像素。



```

1 void ORBextractor::ComputeKeyPointsOctTree(vector<vector<KeyPoint> >& allkeypoints) {
2     for (int level = 0; level < nlevels; ++level)
3         // 计算图像边界
4         const int minBorderX = EDGE_THRESHOLD-3;
5         const int minBorderY = minBorderX;
6         const int maxBorderX = mvImagePyramid[level].cols-EDGE_THRESHOLD+3;
7         const int maxBorderY = mvImagePyramid[level].rows-EDGE_THRESHOLD+3;
8         const float width = (maxBorderX-minBorderX);
9         const float height = (maxBorderY-minBorderY);
10        const int nCols = width/w; // 每一列有多少cell
11        const int nRows = height/h; // 每一行有多少cell
12        const int wCell = ceil(width/nCols); // 每个cell的宽度
13        const int hCell = ceil(height/nRows); // 每个cell的高度
14
15        // 存储需要进行平均分配的特征点
16        vector<cv::KeyPoint> vToDistributeKeys;
17

```

```
18 // step1. 遍历每行和每列,依次分别用高低阈值搜索FAST特征点
19 for(int i=0; i<nRows; i++) {
20     const float iniY = minBorderY + i * hCell;
21     const float maxY = iniY + hCell + 6;
22     for(int j=0; j<nCols; j++) {
23         const float iniX =minBorderX + j * wCell;
24         const float maxX = iniX + wCell + 6;
25         vector<cv::KeyPoint> vKeysCell;
26
27         // 先用高阈值搜索FAST特征点
28         FAST(mvImagePyramid[level].rowRange(iniY,maxY).colRange(iniX,maxX), vKeysCell, iniThFAST,
true);
29
30         // 高阈值搜索不到的话,就用低阈值搜索FAST特征点
31         if(vKeysCell.empty()) {
32             FAST(mvImagePyramid[level].rowRange(iniY,maxY).colRange(iniX,maxX), vKeysCell,
minThFAST, true);
33         }
34         // 把 vKeysCell 中提取到的特征点全添加到 容器vToDistributeKeys 中
35         for(KeyPoint point :vKeysCell) {
36             point.pt.x+=j*wCell;
37             point.pt.y+=i*hCell;
38             vToDistributeKeys.push_back(point);
39         }
40     }
41
42     // step2. 对提取到的特征点进行八叉树筛选,见 DistributeOctTree() 函数
43     keypoints = DistributeOctTree(vToDistributeKeys, minBorderX, maxBorderX, minBorderY, maxBorderY,
mnFeaturesPerLevel[level], level);
44 }
45 // 计算每个特征点的方向
46 for (int level = 0; level < nlevels; ++level)
47     computeOrientation(mvImagePyramid[level], allkeypoints[level], umax);
48 }
49 }
```

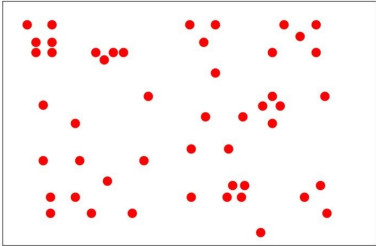
八叉树筛选特征点: DistributeOctTree()

函数 DistributeOctTree() 进行八叉树筛选(非极大值抑制),不断将存在特征点的图像区域进行4等分,直到分出了足够多的分区,每个分区内只保留响应值最大的特征点.

其代码实现比较琐碎,程序里还定义了一个 ExtractorNode 类用于进行八叉树分配,知道原理就行,不看代码.

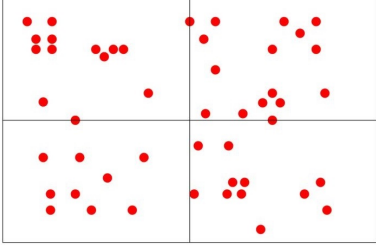
要求: 保留N个点, N=25.

Step 1. 只有1个node



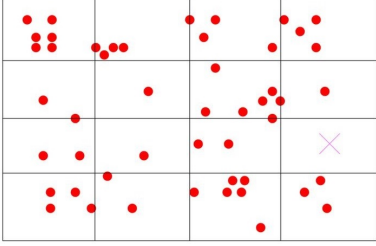
Step 2. 第1次分裂

- 1 node 分裂为4个node
- node数量 4 < 25,还要接着分裂



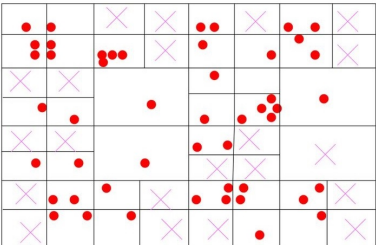
Step 3. 第2次分裂

- 4 node 分裂为15个node。因为有一个node里面没有点,所以不是16个
- node数量 15 < 25,还要接着分裂

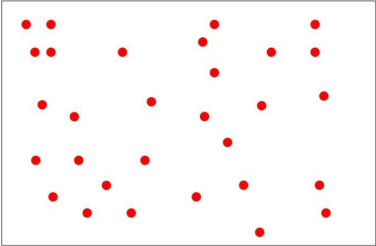


Step 4. 第3次分裂

- 15 node 分裂为30个node
- node数量 30 > 25结束分裂



Step 5. 从每个node中, 选择质量最好一个点



计算特征点方向 computeOrientation()

函数 computeOrientation() 计算每个特征点的方向: 使用特征点周围半径 19 大小的圆的重心方向作为特征点方向.

$$M_{00} = \sum_{X=-R}^R \sum_{Y=-R}^R I(x, y)$$

$$M_{10} = \sum_{X=-R}^R \sum_{Y=-R}^R xI(x, y)$$

$$M_{01} = \sum_{X=-R}^R \sum_{Y=-R}^R yI(x, y)$$

$$Q_X = \frac{M_{10}}{M_{00}}, Q_Y = \frac{M_{01}}{M_{00}}$$

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$

$$\theta = \text{atan2}(m_{01}, m_{10})$$

$$c_x = \frac{\overbrace{\sum_{x=-R}^R \sum_{y=-R}^R xI_{(x,y)}}^{m_{10}}}{\underbrace{\sum_{x=-R}^R \sum_{y=-R}^R I_{(x,y)}}_{m_{00}}}, c_y = \frac{\overbrace{\sum_{x=-R}^R \sum_{y=-R}^R yI_{(x,y)}}^{m_{01}}}{\underbrace{\sum_{x=-R}^R \sum_{y=-R}^R I_{(x,y)}}_{m_{00}}}$$

$$\theta = \arctan2(c_y, c_x) = \arctan2(m_{01}, m_{10})$$

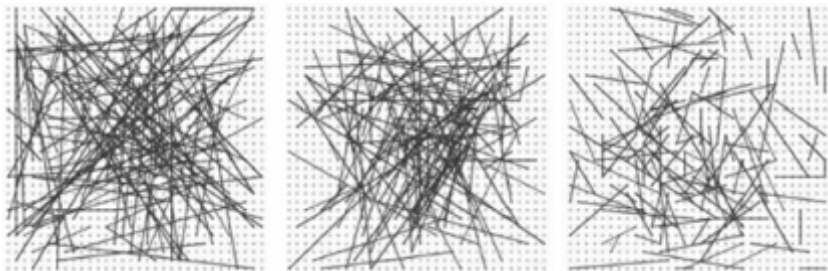
```

1 static void computeOrientation(const Mat& image, vector<KeyPoint>& keypoints, const vector<int>& umax)
2 {
3     for (vector<KeyPoint>::iterator keypoint : keypoints) {
4         // 调用IC_Angle 函数计算这个特征点的方向
5         keypoint->angle = IC_Angle(image, keypoint->pt, umax);
6     }
7 }
8
9 static float IC_Angle(const Mat& image, Point2f pt, const vector<int> & u_max)
10 {
11     int m_01 = 0, m_10 = 0;          // 重心方向
12     const uchar* center = &image.at<uchar>(cvRound(pt.y), cvRound(pt.x));
13     for (int u = -HALF_PATCH_SIZE; u <= HALF_PATCH_SIZE; ++u)
14         m_10 += u * center[u];
15     int step = (int)image.step1();
16     for (int v = 1; v <= HALF_PATCH_SIZE; ++v) {
17         int v_sum = 0;
18         int d = u_max[v];
19         for (int u = -d; u <= d; ++u) {
20             int val_plus = center[u + v*step], val_minus = center[u - v*step];
21             v_sum += (val_plus - val_minus);
22             m_10 += u * (val_plus + val_minus);
23         }
24         m_01 += v * v_sum;
25     }
26
27     // 为了加快速度使用了fastAtan2()函数，输出为[0,360)角度，精度为0.3°
28     return fastAtan2((float)m_01, (float)m_10);
29 }

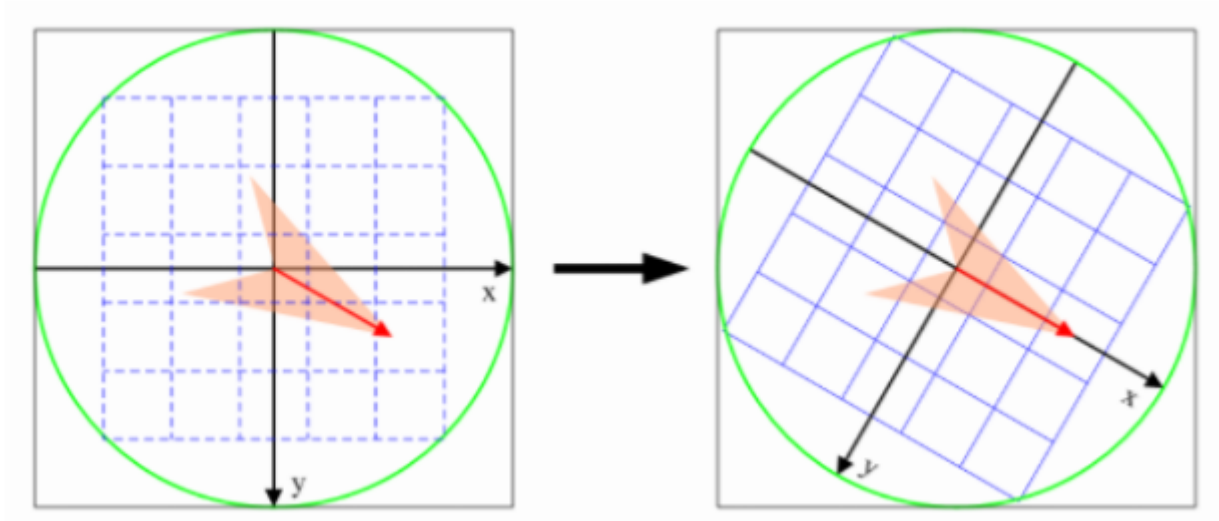
```

计算特征点描述子 computeOrbDescriptor()

计算 BRIEF 描述子的核心步骤是在特征点周围半径为 16 的圆域内选取 256 点对,每个点对内比较得到1位,共得到 256 位的描述子,为保计算的一致性,工程上使用特定设计的点对 pattern,在程序里被硬编码为成员变量了。

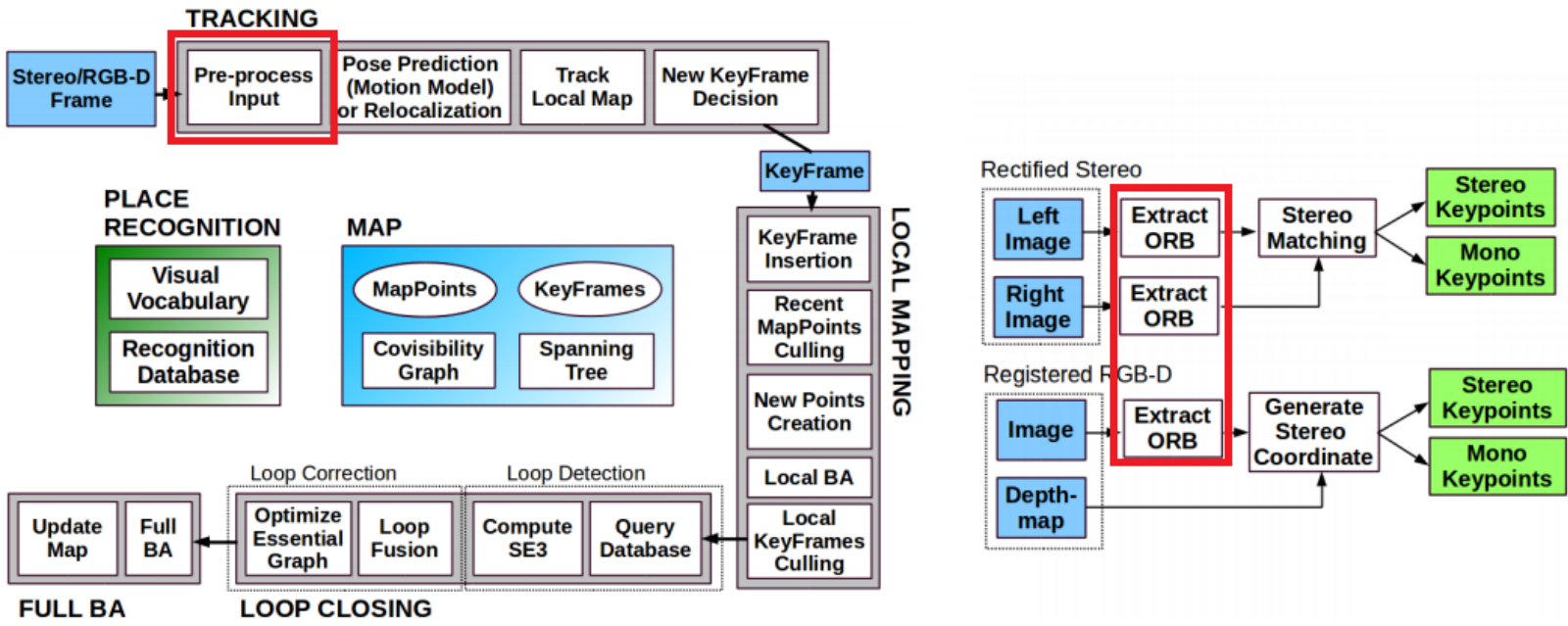


在 computeOrientation() 中我们求出了每个特征点的主方向,在计算描述子时,应该将特征点周围像素旋转到主方向上来计算;为了编程方便,实践上对 pattern 进行旋转。



```
1 static void computeOrbDescriptor(const KeyPoint& kpt, const Mat& img, const Point* pattern, uchar* desc)
2 {
3     float angle = (float)kpt.angle*factorPI;
4     float a = (float)cos(angle), b = (float)sin(angle);
5
6     const uchar* center = &img.at<uchar>(cvRound(kpt.pt.y), cvRound(kpt.pt.x));
7     const int step = (int)img.step;
8
9     // 旋转公式
10    // x'= xcos(θ) - ysin(θ)
11    // y'= xsin(θ) + ycos(θ)
12    #define GET_VALUE(idx) \
13    center[cvRound(pattern[idx].x*b + pattern[idx].y*a)*step + cvRound(pattern[idx].x*a -
pattern[idx].y*b)]
14    for (int i = 0; i < 32; ++i, pattern += 16) {
15        int t0, t1, val;
16        t0 = GET_VALUE(0); t1 = GET_VALUE(1);
17        val = t0 < t1; // 描述子本字节的bit0
18        t0 = GET_VALUE(2); t1 = GET_VALUE(3);
19        val |= (t0 < t1) << 1; // 描述子本字节的bit1
20        t0 = GET_VALUE(4); t1 = GET_VALUE(5);
21        val |= (t0 < t1) << 2; // 描述子本字节的bit2
22        t0 = GET_VALUE(6); t1 = GET_VALUE(7);
23        val |= (t0 < t1) << 3; // 描述子本字节的bit3
24        t0 = GET_VALUE(8); t1 = GET_VALUE(9);
25        val |= (t0 < t1) << 4; // 描述子本字节的bit4
26        t0 = GET_VALUE(10); t1 = GET_VALUE(11);
27        val |= (t0 < t1) << 5; // 描述子本字节的bit5
28        t0 = GET_VALUE(12); t1 = GET_VALUE(13);
29        val |= (t0 < t1) << 6; // 描述子本字节的bit6
30        t0 = GET_VALUE(14); t1 = GET_VALUE(15);
31        val |= (t0 < t1) << 7; // 描述子本字节的bit7
32
33        //保存当前比较的出来的描述子的这个字节
34        desc[i] = (uchar)val;
35    }
36 }
```

ORBextractor类的用途

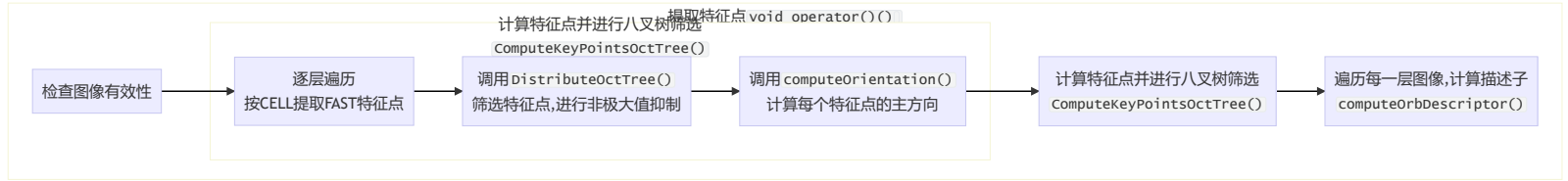


ORBextractor 被用于 Tracking 线程对输入图像预处理的第一步。

ORBextractor类提取特征点的主函数void operator()()

这个函数重载了 () 运算符,使得其他类可以将 ORBextractor 类型变量当作函数来使用。

该函数是 ORBextractor 的主函数,内部依次调用了上面提到的各过程。



```
1 void ORBextractor::operator()(InputArray _image, InputArray _mask, vector<KeyPoint>& _keypoints,
outputArray _descriptors) {
```



```
2 // step1. 检查图像有效性
3 if(_image.empty())
4     return;
5 Mat image = _image.getMat();
6 assert(image.type() == CV_8UC1 );
7
8 // step2. 构建图像金字塔
9 ComputePyramid(image);
10
11 // step3. 计算特征点并进行八叉树筛选
12 vector<vector<KeyPoint> > allkeypoints;
13 ComputeKeyPointsOctTree(allkeypoints);
14
15 // step4. 遍历每一层图像,计算描述子
16 int offset = 0;
17 for (int level = 0; level < nlevels; ++level) {
18     Mat workingMat = mvImagePyramid[level].clone();
19     // 计算描述子之前先进行一次高斯模糊
20     GaussianBlur(workingMat, workingMat, Size(7, 7), 2, 2, BORDER_REFLECT_101);
21     computeDescriptors(workingMat, allkeypoints[level], descriptors.rowRange(offset, offset +
allkeypoints[level].size());, pattern);
22     offset += allkeypoints[level].size();
23 }
24 }
```

这个重载 () 运算符的用法被用在 Frame 类的 ExtractORB() 函数中了,这也是 ORBextractor 类在整个项目中唯一被调用的地方.

```
1 // 函数中`mpORBextractorLeft`和`mpORBextractorRight`都是`ORBextractor`对象
2 void Frame::ExtractORB(int flag, const cv::Mat &im) {
3     if(flag==0)
4         (*mpORBextractorLeft)(im, cv::Mat(), mvKeys, mDescriptors);
5     else
6         (*mpORBextractorRight)(im,cv::Mat(),mvKeysRight,mDescriptorsRight);
7 }
```

ORBextractor 类与其它类间的关系

- Frame 类中与 ORBextractor 有关的成员变量和成员函数

成员变量/函数	访问控制	意义
ORBextractor* mpORBextractorLeft	public	左目特征点提取器
ORBextractor* mpORBextractorRight	public	右目特征点提取器,单目/RGBD模式下为空指针
Frame()	public	Frame 类的构造函数,其中调用 ExtractORB() 函数进行特征点提取
ExtractORB()	public	提取 ORB 特征点,其中调用了 mpORBextractorLeft 和 mpORBextractorRight 的 () 方法

```
1 // Frame类的两个ORBextractor是在调用构造函数时传入的,构造函数中调用ExtractORB()提取特征点
2 Frame::Frame(ORBextractor *extractorLeft, ORBextractor *extractorRight)
3     : mpORBextractorLeft(extractorLeft), mpORBextractorRight(extractorRight) {
4
5     // ...
6
7     // 提取ORB特征点
8     thread threadLeft(&Frame::ExtractORB, this, 0, imLeft);
9     thread threadRight(&Frame::ExtractORB, this, 1, imRight);
10    threadLeft.join();
11    threadRight.join();
12
13    // ...
14 }
15
16 // 提取特征点
17 void Frame::ExtractORB(int flag, const cv::Mat &im) {
18     if (flag == 0)
19         (*mpORBextractorLeft)(im, cv::Mat(), mvKeys, mDescriptors);
20     else
21         (*mpORBextractorRight)(im, cv::Mat(), mvKeysRight, mDescriptorsRight);
22 }
```

- Frame 类的两个 ORBextractor 指针指向的变量是 Tracking 类的构造函数中创建的

```
1 // Tracking构造函数
2 Tracking::Tracking() {
3     // ...
4 }
```

```
4
5 // 创建两个ORB特征点提取器
6 mpORBExtractorLeft = new ORBExtractor(nFeatures, fScaleFactor, nLevels, fIniThFAST, fMinThFAST);
7 if (sensor == System::STEREO)
8     mpORBExtractorRight = new ORBExtractor(nFeatures, fScaleFactor, nLevels, fIniThFAST,
fMinThFAST);
9
10 // ...
11 }
12
13 // Tracking线程每收到一帧输入图片,就创建一个Frame对象,创建Frame对象时将提取器mpORBExtractorLeft和
mpORBExtractorRight给构造函数
14 cv::Mat Tracking::GrabImageStereo(const cv::Mat &imRectLeft, const cv::Mat &imRectRight, const double
&timestamp) {
15     // ...
16
17     // 创建Frame对象
18     mCurrentFrame = Frame(mImGray, imGrayRight, timestamp, mpORBExtractorLeft, mpORBExtractorRight);
19
20     // ...
21 }
```

由上述代码分析可知,每次完成 ORB 特征点提取之后,图像金字塔信息就作废了,下一帧图像到来时调用 ComputePyramid() 函数会覆盖掉本帧图像的图像金字塔信息;但从金字塔中提取的图像特征点的信息会被保存在 Frame 对象中.所以ORB-SLAM2是稀疏重建,对每帧图像只保留最多 nfeatures 个特征点(及其对应的地图点).

