

第1周：ORB_SLAM2 课程课件

本周课程学前准备：

本周课程重点：

1. ORB_SLAM 2简介

 1.1 ORB_SLAM 2简介

 1.2 ORB_SLAM2 框架

 主体框架

 数据输入的预处理

2. TUM 数据集使用

 2.1 ORB_SLAM2 在TUM数据集上的表现

 2.2 TUM RGB-D 数据集 简介

 2.3 运行RGBD模式时的预处理

 2.4. 不同颜色地图点的含义

3. ORB 特征

 3.1 FAST关键点

 3.2 BRIEF 特征

 3.3 ORB 特征

 3.4 为什么要重载小括号运算符 operator() ?

 3.5 金字塔的计算

 3.6 特征点数量的分配计算

 3.7 使用四叉树对图层中的特征点进行平均和分发 DistributeOctTree

第1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。**

本周课程学前准备：

1. Ubuntu操作系统，不建议装虚拟机。版本16.04、18.04、20.04均可

win10 + Ubuntu18.04 双系统安装教程可参考：

<https://www.cnblogs.com/masbay/p/10844857.html>

<https://www.jianshu.com/p/fe4e3915495e>

<https://www.cnblogs.com/tanrong/p/9166595.html>

Linux 入门教程：

鸟哥的 Linux 私房菜 -- 基础学习篇

http://cn.linux.vbird.org/linux_basic/linux_basic.php

Linux命令 快速索引

http://cn.linux.vbird.org/linux_basic/1010index.php

本周课程重点：

1. 熟悉Ubuntu操作系统，掌握Linux基础语法。
2. 初步了解ORB-SLAM2的优点及应用。
3. 成功编译并运行ORB-SLAM2 单目模式。
4. 掌握ORB特征及均匀化原理及实现。

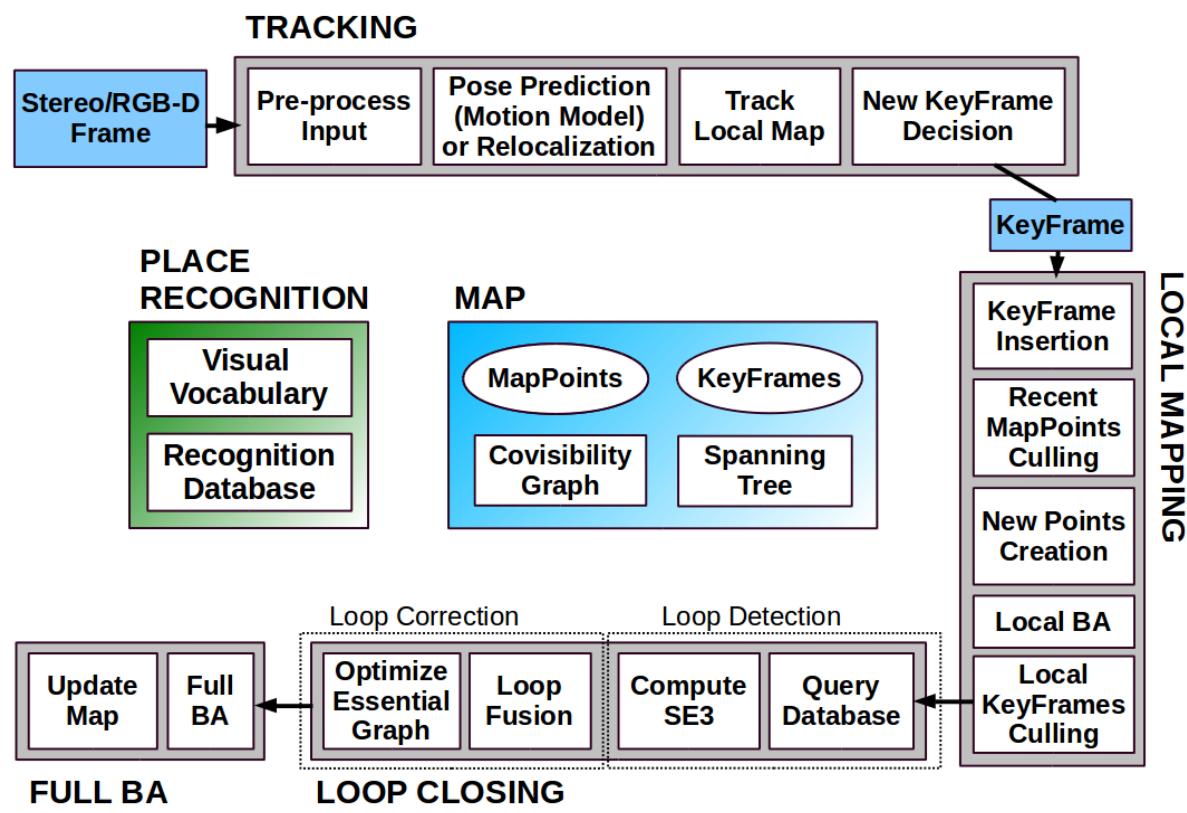
1. ORB-SLAM 2简介

1.1 ORB-SLAM 2简介

- 首个（2017年发布时）支持单目，双目和RGB-D相机的完整的开源SLAM方案，具有回环检测和重新定位的功能。
- 能够在CPU上进行实时工作，可以用于移动终端如 移动机器人、手机、无人机、汽车。特征点法的巅峰之作，定位精度极高，可达厘米级。
- 能够实时计算出相机的位姿，并生成场景的稀疏三维重建地图。
- 代码非常工整，可读性强，包含很多实际应用中的技巧，非常实用。
- 支持仅定位模式，该模式适用于轻量级以及在地图已知情况下长期运行，此时不使用局部建图和回环检测的线程。
- 双目和RGB-D相对单目相机的主要优势在于，可以直接获得深度信息，不需要像单目情况中那样做一个特定的SFM初始化。

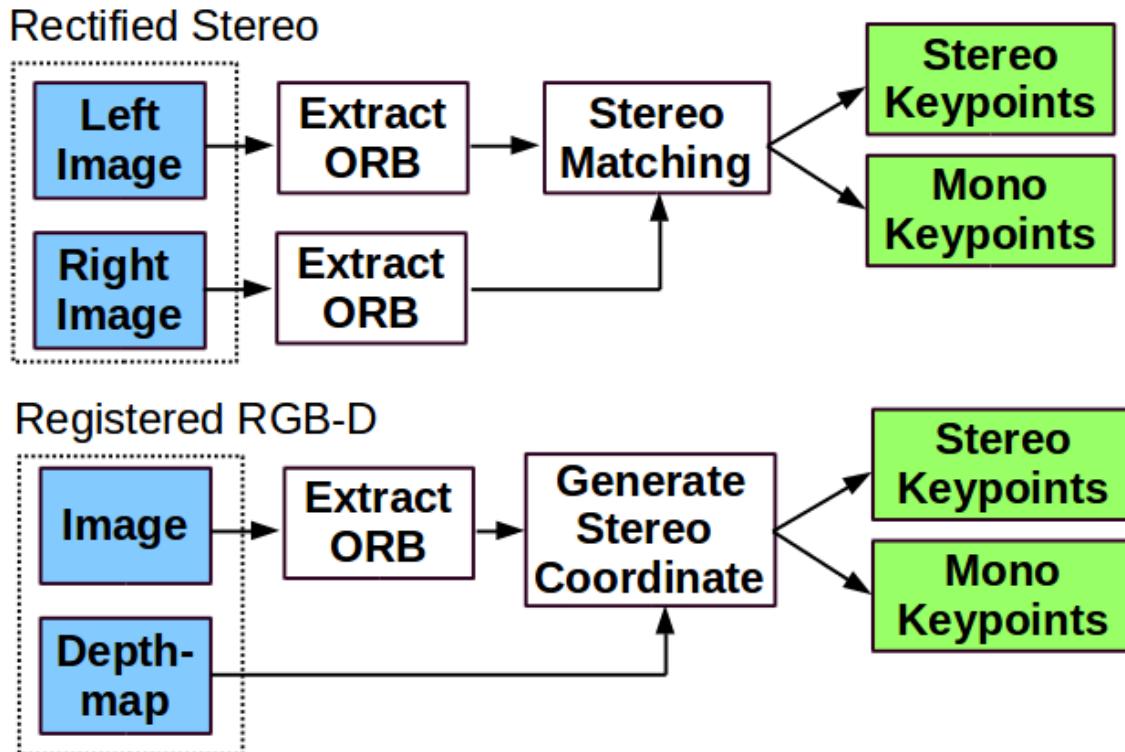
1.2 ORB-SLAM2 框架

主体框架



数据输入的预处理

为了兼容不同相机（双目相机与RGBD相机），需要对输入数据进行预处理，使得交给后期处理的数据格式一致，具体流程如下：



2. TUM 数据集使用

2.1 ORB_SLAM2 在TUM数据集上的表现

TUM RGB-D DATASET. COMPARISON OF TRANSLATION RMSE (m).

Sequence	ORB-SLAM2 (RGB-D)	Elastic-Fusion	Kintinuous	DVO SLAM	RGBD SLAM
fr1/desk	0.016	0.020	0.037	0.021	0.026
fr1/desk2	0.022	0.048	0.071	0.046	-
fr1/room	0.047	0.068	0.075	0.043	0.087
fr2/desk	0.009	0.071	0.034	0.017	0.057
fr2/xyz	0.004	0.011	0.029	0.018	-
fr3/office	0.010	0.017	0.030	0.035	-
fr3/nst	0.019	0.016	0.031	0.018	-

2.2 TUM RGB-D 数据集 简介

注意两点：深度的表达方式 以及 数据的存储格式

Color images and depth maps

We provide the time-stamped color and depth images as a gzipped tar file (TGZ).

- The color images are stored as 640x480 8-bit RGB images in PNG format.
- The depth maps are stored as 640x480 16-bit monochrome images in PNG format.
- The color and depth images are already pre-registered using the OpenNI driver from PrimeSense, i.e., the pixels in the color and depth images correspond already 1:1.
- The depth images are scaled by a factor of 5000, i.e., a pixel value of 5000 in the depth image corresponds to a distance of 1 meter from the camera, 10000 to 2 meter distance, etc. A pixel value of 0 means missing value/no data.

→ 扩大倍数 $1m = 5000$

Ground-truth trajectories

We provide the groundtruth trajectory as a text file containing the translation and orientation of the camera in a fixed coordinate frame. Note that also our automatic evaluation tool expects both the groundtruth and estimated trajectory to be in this format.

- Each line in the text file contains a single pose.
- The format of each line is **timestamp tx ty tz qx qy qz qw**
- **timestamp** (float) gives the number of seconds since the Unix epoch.
- **tx ty tz** (3 floats) give the position of the optical center of the color camera with respect to the world origin as defined by the motion capture system.
- **qx qy qz qw** (4 floats) give the orientation of the optical center of the color camera in form of a unit quaternion with respect to the world origin as defined by the motion capture system.
- The file may contain comments that have to start with "#".

Intrinsic Camera Calibration of the Kinect

2.3 运行RGBD模式时的预处理

关于associate.py

注意：只能在Python2 环境下运行

RGB Depth
时间戳对齐

Python下运行

```
python associate.py rgb.txt depth.txt > associate.txt
```

```
python associate.py associate.txt groundtruth.txt > associate_with_groundtruth.txt
```

注意：

直接association后出问题，生成的结果

associate.txt 1641行

associate_with_groundtruth.txt 1637行

也就是说，associate的不一定有groundtruth，所以要以associate_with_groundtruth.txt的关联结果为准

yaml 有预配置参数，读进去

2.4. 不同颜色地图点的含义

- 红色点表示参考地图点，其实就是tracking里的local mappoints

```

void Tracking::UpdateLocalMap()
{
    // This is for visualization
    mpMap->SetReferenceMapPoints(mvpLocalMapPoints);

    // Update
    UpdateLocalKeyFrames();
    UpdateLocalPoints();
}

```

- 黑色点表示所有地图点，红色点属于黑色点的一部分

```

void MapDrawer::DrawMapPoints()
{
    const vector<MapPoint*> &vpMPS = mpMap-> GetAllMapPoints();
    const vector<MapPoint*> &vpRefMPS = mpMap->GetReferenceMapPoints();

    set<MapPoint*> spRefMPS(vpRefMPS.begin(), vpRefMPS.end());

    if(vpMPS.empty())
        return;

    glPointSize(mPointSize);
    glBegin(GL_POINTS);
    glColor3f(0.0,0.0,0.0);

    for(size_t i=0, iend=vpMPS.size(); i<iend; i++)
    {
        if(vpMPS[i]->isBad() || spRefMPS.count(vpMPS[i]))
            continue;
        cv::Mat pos = vpMPS[i]->GetWorldPos();
        glVertex3f(pos.at<float>(0),pos.at<float>(1),pos.at<float>(2));
    }
    glEnd();

    glPointSize(mPointSize);
    glBegin(GL_POINTS);
    glColor3f(1.0,0.0,0.0);

    for(set<MapPoint*>::iterator sit=spRefMPS.begin(), send=spRefMPS.end();
        sit!=send; sit++)
    {
        if((*sit)->isBad())
            continue;
        cv::Mat pos = (*sit)->GetWorldPos();
        glVertex3f(pos.at<float>(0),pos.at<float>(1),pos.at<float>(2));
    }
    glEnd();
}

```

```

void MapDrawer::DrawMapPoints()
{
    const vector<MapPoint*> &vpMPs = mpMap-> GetAllMapPoints();
    const vector<MapPoint*> &vpRefMPs = mpMap-> GetReferenceMapPoints();

    set<MapPoint*> spRefMPs(vpRefMPs.begin(), vpRefMPs.end());

    if(vpMPs.empty())
        return;

    glPointSize(mPointSize);
    glBegin(GL_POINTS);
    glColor3f(0.0,0.0,0.0);

    for(size_t i=0, iend=vpMPs.size(); i<iend;i++)
    {
        if(vpMPs[i]->isBad() || spRefMPs.count(vpMPs[i]))
            continue;
        cv::Mat pos = vpMPs[i]->GetWorldPos();
        glVertex3f(pos.at<float>(0),pos.at<float>(1),pos.at<float>(2));
    }
    glEnd();

    glPointSize(mPointSize);
    glBegin(GL_POINTS);
    glColor3f(1.0,0.0,0.0);
}

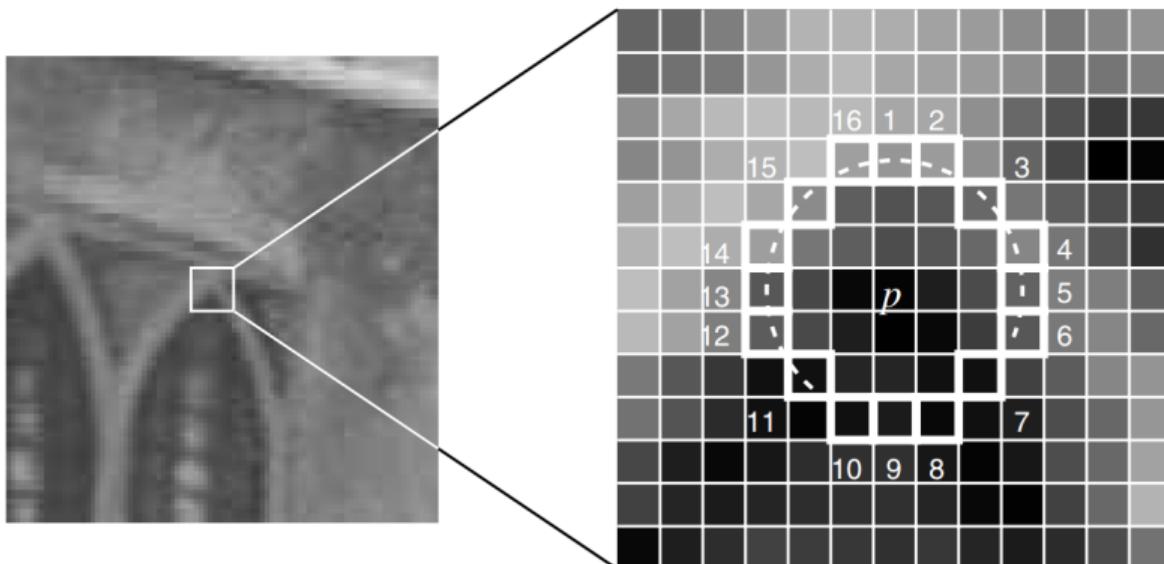
for(set<MapPoint*>::iterator sit=spRefMPs.begin(), send=spRefMPs.end(); sit!=send; sit++)
{
    if((*sit)->isBad())
        continue;
    cv::Mat pos = (*sit)->GetWorldPos();
    glVertex3f(pos.at<float>(0),pos.at<float>(1),pos.at<float>(2));
}

```

OpenGL glColor3f [0.0 ,1.0]
参数范围

3. ORB 特征

3.1 FAST关键点



- 选取像素p，假设它的亮度为Ip；

- 设置一个阈值T (比如Ip的20%) ;
- 以像素p为中心, 选取半径为3的圆上的16个像素点;
- 假如选取的圆上, 有连续的N个点的亮度大于Ip+T或小于Ip-T, 那么像素p可以被认为是特征点;
- 循环以上4步, 对每一个像素执行相同操作。

3.2 BRIEF 特征

论文: BRIEF: Binary Robust Independent Elementary Features

ORB pattern [256 * 4]

BRIEF算法的核心思想是在关键点P的周围以一定模式选取N个点对, 把这N个点对的比较结果组合起来作为描述子。为了保持踩点固定, 工程上采用特殊设计的固定的pattern来做



3.3 ORB 特征

原始的FAST关键点没有方向信息, 这样当图像发生旋转后, brief描述子也会发生变化, 使得特征点对旋转不鲁棒

解决方法: **orientated FAST**

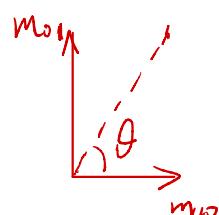
使用灰度质心法计算特征点的方向,

什么是灰度质心法?

$$\begin{aligned}
 M_{00} &= \sum_{X=-R}^R \sum_{Y=-R}^R I(x, y) \\
 M_{10} &= \sum_{X=-R}^R \sum_{Y=-R}^R xI(x, y) \quad \text{所有像素值} \times \text{生} \times \text{方向值. 之和} \\
 M_{01} &= \sum_{X=-R}^R \sum_{Y=-R}^R yI(x, y) \quad \times \text{生} \times \text{方向值 之和} \\
 Q_X &= \frac{M_{10}}{M_{00}}, Q_Y = \frac{M_{01}}{M_{00}} \\
 C &= \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \\
 \theta &= \arctan 2(m_{01}, m_{10})
 \end{aligned}$$

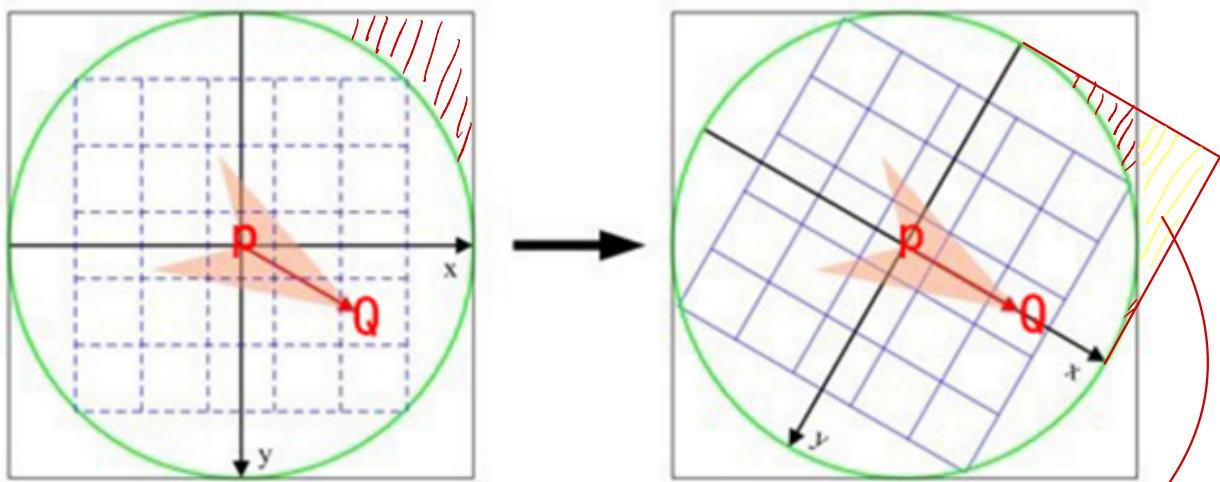
$$c_x = \frac{\overbrace{\sum_{x=-R}^R \sum_{y=-R}^{m_{10}} xI(x, y)}^{m_{10}}}{\underbrace{\sum_{x=-R}^R \sum_{y=-R}^{m_{00}} I(x, y)}_{m_{00}}}, c_y = \frac{\overbrace{\sum_{x=-R}^R \sum_{y=-R}^{m_{01}} yI(x, y)}^{m_{01}}}{\underbrace{\sum_{x=-R}^R \sum_{y=-R}^{m_{00}} I(x, y)}_{m_{00}}}$$

$$\theta = \arctan 2(c_y, c_x) = \arctan 2(m_{01}, m_{10})$$



在一个圆内计算灰度质心

下图P为几何中心, Q为灰度质心 → 像素集中



https://blog.csdn.net/qq_18661939/article/details/52900524

灰度质心用圆

因为正方形旋转后会丢失

下面求圆内的坐标范围

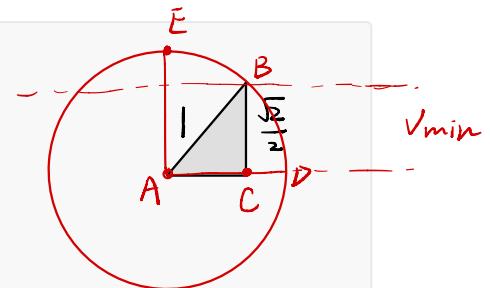
不同 v 对应的 u 的坐标

umax: 1/4圆的每一行的u轴坐标边界 (下图中橙色线段FG)

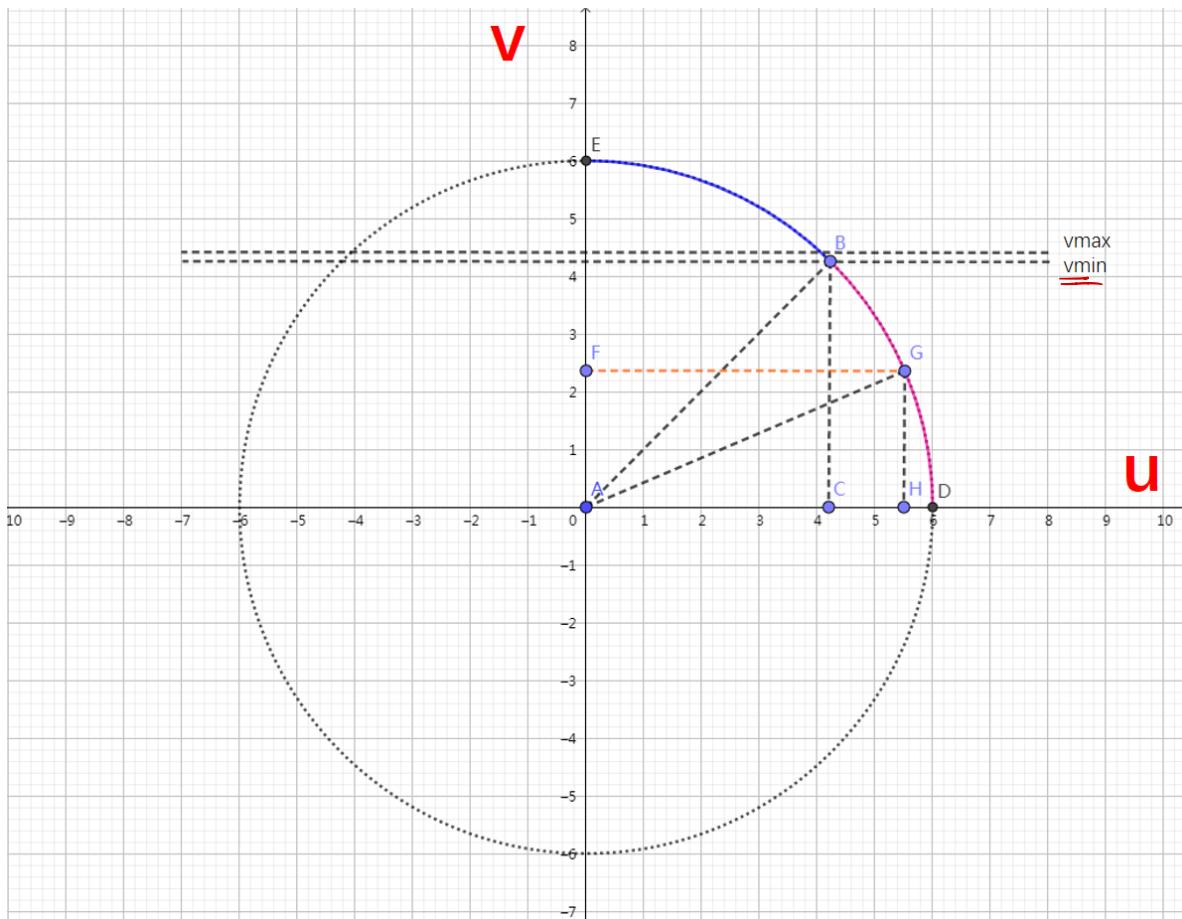
```

vmax = cvFloor(HALF_PATCH_SIZE * sqrt(2.f) / 2 + 1);
vmin = cvCeil(HALF_PATCH_SIZE * sqrt(2.f) / 2);
sqrt(2.f) / 2
// 对应从D到B的红色弧线, umax坐标从D到C
for (v = 0; v <= vmax; ++v)
    umax[v] = cvRound(sqrt(hp2 - v * v));
勾股
// 对应从B到E的蓝色弧线, umax坐标从C到A
for (v = HALF_PATCH_SIZE, v0 = 0; v >= vmin; --v)
{
    while (umax[v0] == umax[v0 + 1])
        ++v0;
    umax[v] = v0;
    ++v0;
}

```



ORBextractor.cc



参考：

<https://blog.csdn.net/liu502617169/article/details/89423494>

<https://www.cnblogs.com/wall-e2/p/8057448.html>

Frame.cc
ExtractORB 函数中

(mp ORBextractor Left) (im,
cv::Mat),
mKeypoints,
mDescriptors)

3.4 为什么要重载小括号运算符 operator() ?

可以用于仿函数（一个可以实现函数功能的对象）

仿函数（functor）又称为函数对象（function object）是一个能行使函数功能的类。仿函数的语法几乎和我们普通的函数调用一样，不过作为仿函数的类，都必须重载operator()运算符

1 仿函数可有自己的数据成员和成员变量，这意味着这意味着仿函数拥有状态。这在一般函数中是不可能的。

2 仿函数通常比一般函数有更好的速度。

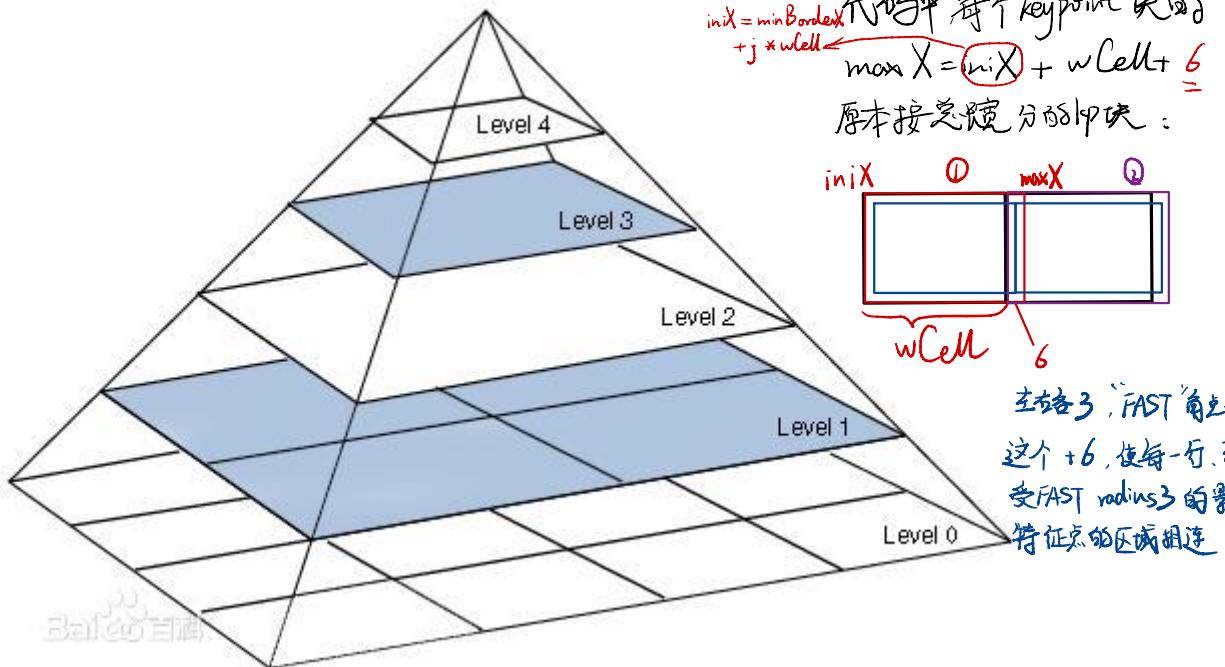
扩展阅读

<https://blog.csdn.net/jinzhu1911/article/details/101317367>

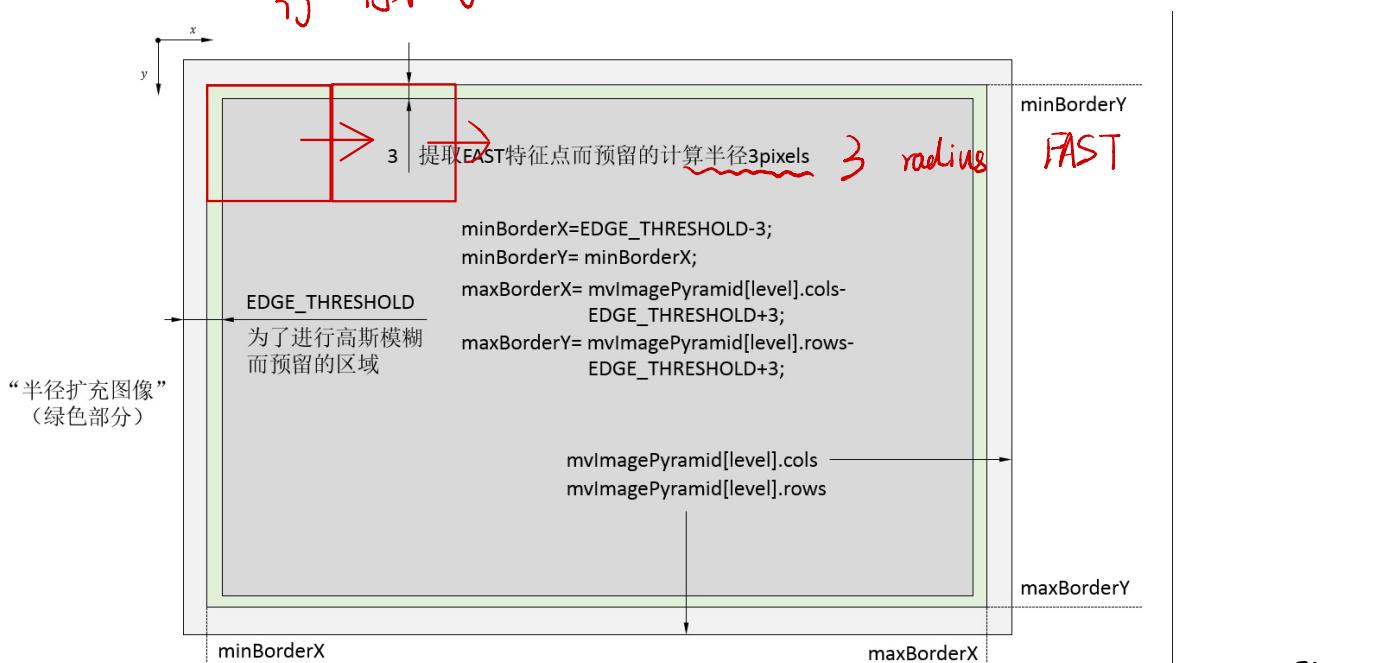
ComputePyramid

3.5 金字塔的计算

ORBextractor::ComputePyramid



一行一来的



$mvImagePyramid[level] = temp$ (Rect (EDGE_THRESHOLD, EDGE_THRESHOLD,
 sz.width, sz.height))
 if (level != 0)

取当前层除去
 border的图像部分

resize (mvImagePyramid[level - 1],

mvImagePyramid [level],

sz, → 当前图层尺寸

0,

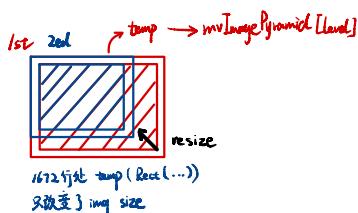
0,

cv::INTER_LINEAR)

copy MakeBorder (mvImagePyramid [level],

→ 扩加边界

temp,
 ...)
 ↓ 加了边界



\$ copyMakeBorder()

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

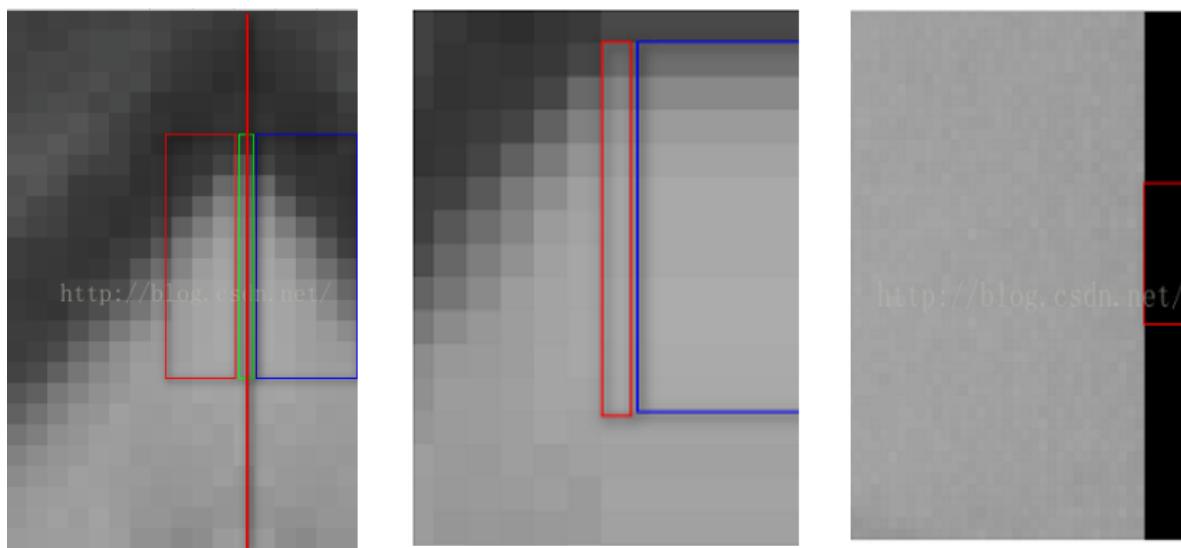
Python:

```
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

对称补

金字塔层数越高，图像的面积越小，所能提取到的特征数量就越小。基于这个原理，我们可以按照面积将特征点均摊到金字塔每层的图像上。我们假设第0层图像的宽为 W ，长为 L ，缩放因子为 s （这里的 $0 < s < 1$ ）。那么整个金字塔总的面积为

$$S = \frac{1}{scaleFactor} \cdot L \times W \times [s^2]^0 + L \times W \times [s^2]^1 + \cdots + L \times W \times [s^2]^{(n-1)}$$

$$= \underbrace{L \times W}_{C} \times \frac{1 - [s^2]^n}{1 - [s^2]} = C \frac{1 - [s^2]^n}{1 - [s^2]}$$

总面积、 等比数列求和 $\frac{x_0(1-q^n)}{1-q}$

那么，单位面积的特征点数量为

$$Navg = \frac{N}{S} = \frac{N}{C \frac{1 - [s^2]^n}{1 - [s^2]}} = \frac{N \{1 - [s^2]\}}{C \{1 - [s^2]^n\}}$$

那么，第0层应分配的特征点数量为

$$N_0 = \frac{N \{1 - [s^2]\}}{\{1 - [s^2]^n\}}$$

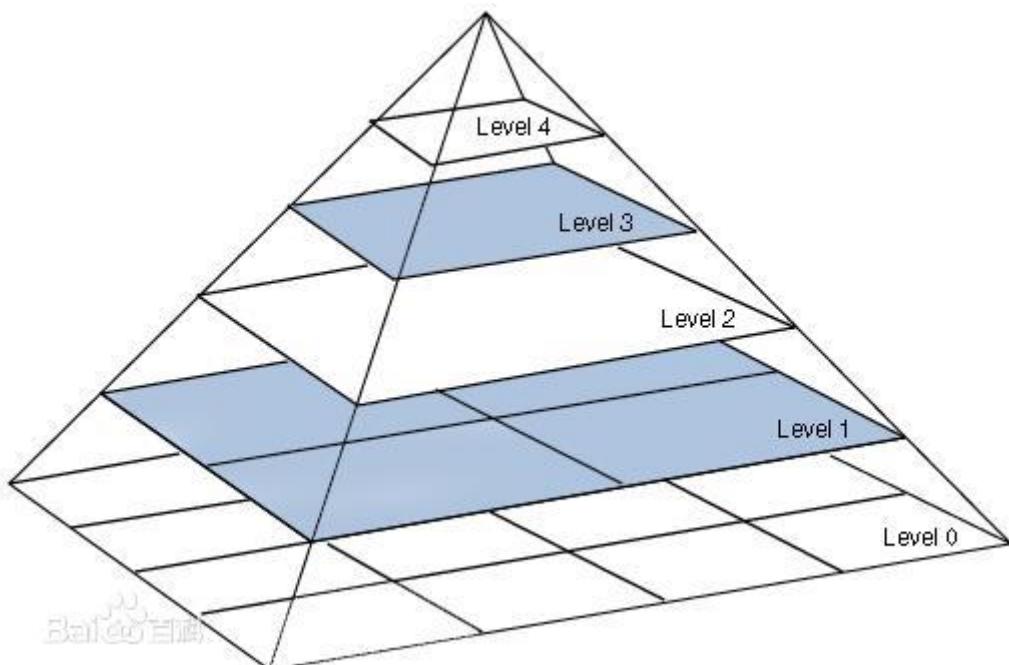
DORB extractor.cc

接着那么，推出了第 α 层应分配的特征点数量为

$$N_\alpha = \frac{N \{1 - [s^2]\}}{\{1 - [s^2]^n\}} [s^2]^\alpha$$

这些计算都在构造函数中

实际上，OpenCV里的代码不是按照面积算的，而是按照边长来算的。也就是上面公式中的 $[s^2]$ 换成 s 。



3.6 特征点数量的分配计算

来自：<https://zhuanlan.zhihu.com/p/61738607>

金字塔层数越高，图像的面积越小，所能提取到的特征数量就越小。基于这个原理，我们可以按照面积将特征点均摊到金字塔每层的图像上。我们假设第0层图像的宽为 W ，长为 L ，缩放因子为 s （这里的 $0 < s < 1$ ）。那么整个金字塔总的面积为

$$\begin{aligned} S &= L \times W \times [s^2]^0 + L \times W \times [s^2]^1 + \cdots + L \times W \times [s^2]^{(n-1)} \\ &= \underbrace{L \times W}_C \times \frac{1 - [s^2]^n}{1 - [s^2]} = C \frac{1 - [s^2]^n}{1 - [s^2]} \end{aligned}$$

那么，单位面积的特征点数量为

$$Navg = \frac{N}{S} = \frac{N}{C \frac{1 - [s^2]^n}{1 - [s^2]}} = \frac{N \{1 - [s^2]\}}{C \{1 - [s^2]^n\}}$$

那么，第0层应分配的特征点数量为

$$N_0 = \frac{N \{1 - [s^2]\}}{\{1 - [s^2]^n\}}$$

接着那么，推出了第 α 层应分配的特征点数量为

$$N_\alpha = \frac{N \{1 - [s^2]\}}{\{1 - [s^2]^n\}} [s^2]^\alpha$$

实际上，OpenCV里的代码不是按照面积算的，而是按照边长来算的。也就是上面公式中的 $[s^2]$

换成 s 。

用 s 把分块提 $FAST$ 存入 ToDistributeKeys 的末经筛选，远大于图层分配个数的 kp 进行分发筛选，
留下 $nfeaturesPerLevel[\alpha]$ 个

3.7 使用四叉树对图层中的特征点进行平均和分发

DistributeOctTree 传入此函数的参数边界就是加入 $radius=3$ 的边界

- 如果图片的宽度比较宽，就先把分成左右 w/h 份。一般的 640×480 的图像开始的时候只有一个 node。
- 如果 node 里面的点数 > 1 ，把每个 node 分成四个 node，如果 node 里面的特征点为空，就不要了，删掉。
- 新分的 node 的点数 > 1 ，就再分裂成 4 个 node。如此，一直分裂。
- 终止条件为：node 的总数量 $> [\text{公式}]$ ，或者无法再进行分裂。
- 然后从每个 node 里面选择一个质量最好的 FAST 点。

提取特征点，
mono_tum.cc

SLAM.TrackMonocular (im, tframe);

System.cc

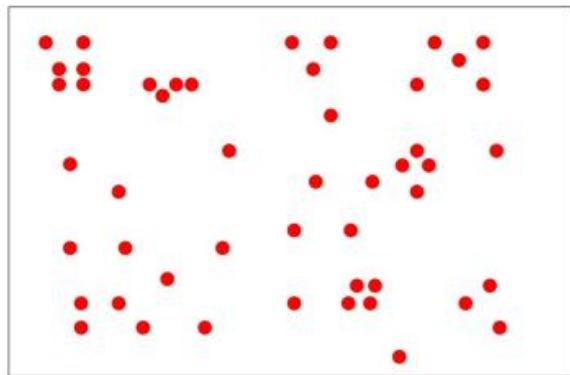
$T_{cw} = \text{mpTracker} \rightarrow \text{GrabImageMonocular}(im, timestamp)$;
① 转灰度

Frame.cc

单目，0 代表左 camera ② 构造 Frame
ExtractorORB (0, imGray)

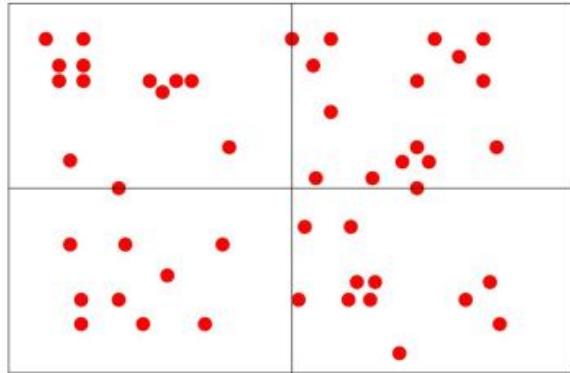
要求：保留N个点，N=25.

Step 1. 只有1个node



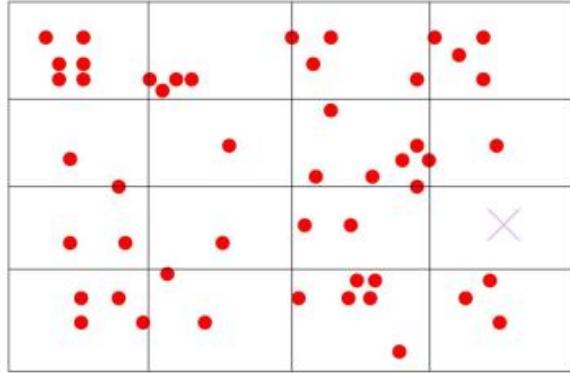
Step 2. 第1次分裂

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



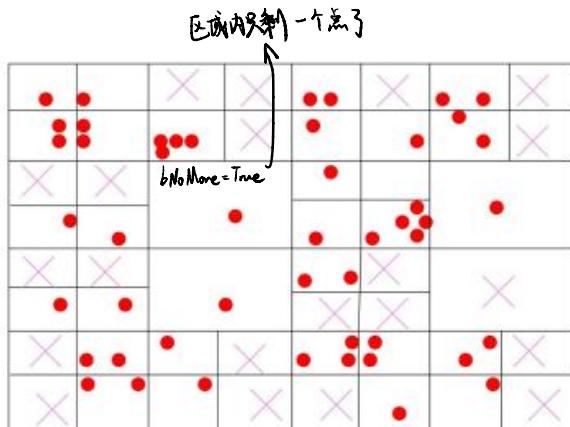
Step 3. 第2次分裂

- 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



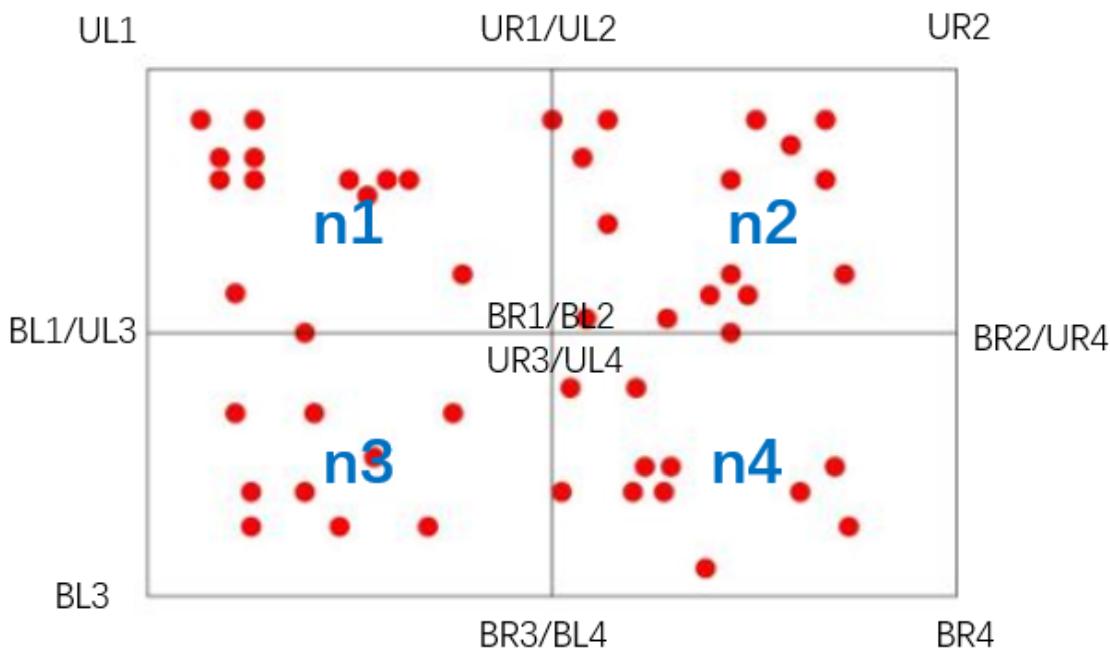
Step 5. 从每个node中，选择质量最好一个点



参考：<https://zhuanlan.zhihu.com/p/61738607>

ExtractorNode::DivideNode

U: up, B: bottom, L: left, R: right



节点分裂顺序：后加的先分裂

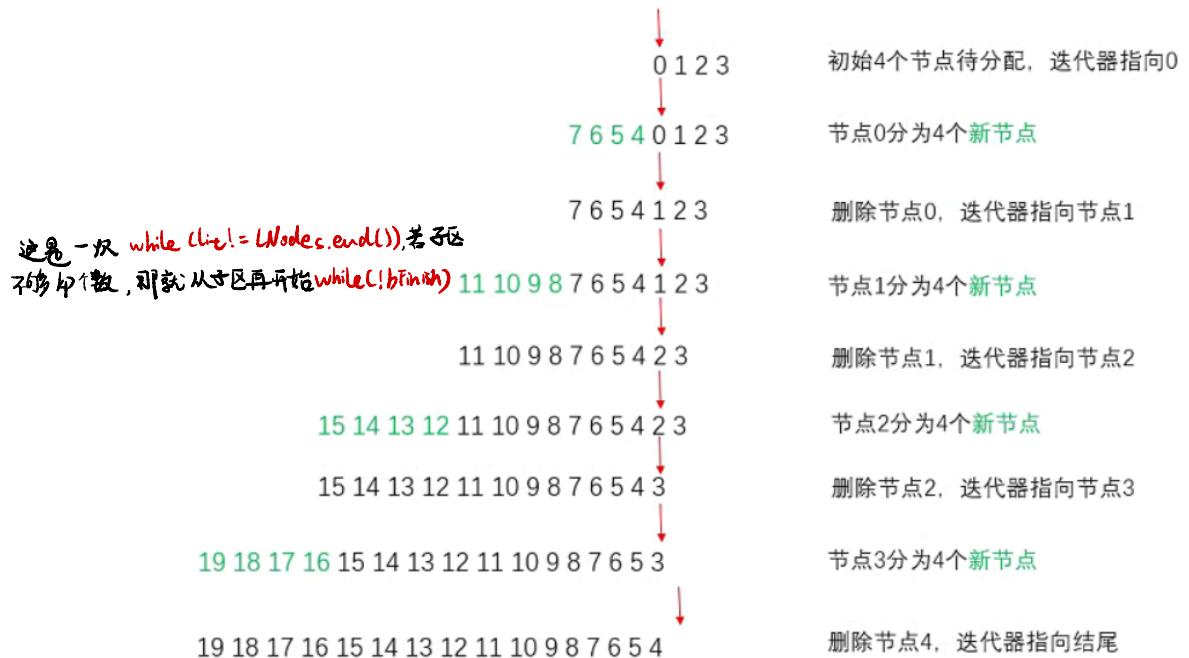
- ② 若 $n_{hi} = \text{round}(x/y) \neq 1$, 即长大于 1.5 倍宽, 那么 $hx = (\max X - \min X)/n_{hi}$, 把 X 进分块, 然后算各块边界坐标, 以链表的形式存在 $LNodes$ 里, 再把链表 $LNodes.back()$ 也就是当前刚用 push_back 进去的元素存入 $vInNodes[1]$, 等于说存到了两个结构中。

③ 再从 vToDistributeKeys 即 FAST 提的未排的点, 用 $vInNodes[kp.pt.x/HX] \rightarrow vKeys.push_back(kp)$, 用的是这个 通过点的 x 坐标 断在那个区域(一般就一个区) 后面用

④ 遍历 $LNodes$ 中所有提取器, 若 $lit \rightarrow \text{beNoMore}$ 为 true 只有一个 kp, 那就不再分了, $lit++$ 到下个; else $lit \rightarrow \text{DivideNode}$ 。
 ⑤ DivideNode , 俗 $n_1 \dots n_4$ 定坐标, 把 $vkeys$ 即未分割前的 kpoints 分配到 四叉树的 $n_1 \dots n_4$ 内。
 要分的 ExtractorNode, 这是一个线性数据结构, 有该 Node 的 UL, UR, BL, BR
 ⑥ 子区域若 $kp > 0$, 会 $LNodes.push_front(kp)$, 放在链表开头, 并把 $\text{make_pair}(n, vkeys.size(), \&LNodes.front())$ 存入 $vSizeAndPointerToNode$, 并把 ExtractorNode 定义的 iterator 用 $LNodes.front().lit = LNodes.begin()$, 会把 $n_1 \dots n_4$ 都处理一遍
 最后把母结点 $lit = LNodes.erase(lit)$ 移掉。

⑦ 会判断 $LNodes.size()$ 是否 $>= N$ 即要求图层的 kp 个数或 $= \text{preVsize}$ 即没变后节点个数不变, 也就是分不开了, 那么 $bFinish = \text{true}$ 。

⑧ 当分割子区域快到 kp 个数时, 会用 Δ 这里的 $kp.size()$ 排序, 先对点多的分四叉, 即 else if((int) $LNodes.size() + \text{toExpand} * 3) > N$)
 while ($lit != LNodes.end()$) 这两个循环的子区, 因为 $vSizeAndPointerToNode$ 每次子分区会 clear(), 这里就是足够分了, 先从 kp 多的分, 分够就 break。
 累加, 待分的区域减一次分区 while $lit != LNodes.end()$ 清空一次。



分区取FAST → 大区分子区，分成4个数一致 → 取各子区响应值最高