ORB 算法流程（OpenCV 源码）

ORB 算法在 OpenCV 中已经得到实现，想着改进这个提取算法，于是先阅读了源码，以下是根据博客（ https://blog.csdn.net/haoliliang88/article/details/51841131 ）得到的算法流程。

1，首先是测试文件中的接口函数：（ orb_test.cpp ）

```
//定义ORB特征点对象并进行特征提取
//CV_WRAP explicit ORB(int nfeatures = 500, float scaleFactor = 1.2f, int nlevels = 8, int edgeThreshold = 31,
//int firstLevel = 0, int WTA_K=2, int scoreType=ORB::HARRIS_SCORE, int patchSize=31 );
ORB orb_obj( 3000, 1.2f, 8, 31, 0, 2, ORB::HARRIS_SCORE, 31 );
//steady_clock::time_point t1 = steady_clock::now();
orb_obj.detect(image1,keypoints1);
```

orb_obj 这个对象是根据利用 detect()函数进行关键点提取的，然后再去寻找了 detect()函数，而 ORB 类中并没有这个函数，只有一个 detecetImpl()的函数，于是便想着 detect()应该是父类的成员函数；

2，（ opencv-2.4.9\modules\features2d\include\opencv2\features2d\features2d.hpp ）

ORB 是 Feature2D 的子类：

```
/*!
 ORB implementation.
*/
class CV_EXPORTS_W ORB : public Feature2D
{
public:
```

Feature2D 是 FeatureDetector 的子类：

```
/*
 * Abstract base class for simultaneous 2D feature detection descriptor extraction.
 */
class CV_EXPORTS_W Feature2D : public FeatureDetector, public DescriptorExtractor
{
public:
    /*
```

detect()函数其实是 FeatureDetector 的成员函数，同时还有一个角 detectImpl()的虚函数：

```
/*
 * Abstract base class for 2D image feature detectors.
 */
class CV_EXPORTS_W FeatureDetector : public virtual Algorithm
{
public:
    virtual ~FeatureDetector();

    /*
     * Detect keypoints in an image.
     * image        The image.
     * keypoints    The detected keypoints.
     * mask         Mask specifying where to look for keypoints (optional). Must be a char
     *              matrix with non-zero values in the region of interest.
     */
    CV_WRAP void detect( const Mat& image, CV_OUT vector<KeyPoint>& keypoints, const Mat& mask=Mat() ) const;

    /*
     * Detect keypoints in an image set.
     * images       Image collection.
     * keypoints    Collection of keypoints detected in an input images. keypoints[i] is a set of keypoints detected in an images[i].
     * masks        Masks for image set. masks[i] is a mask for images[i].
     */
    void detect( const vector<Mat>& images, vector<vector<KeyPoint> >& keypoints, const vector<Mat>& masks=vector<Mat>() ) const;

    // Return true if detector object is empty
    CV_WRAP virtual bool empty() const;

    // Create feature detector by detector name.
    CV_WRAP static Ptr<FeatureDetector> create( const string& detectorType );

protected:
    virtual void detectImpl( const Mat& image, vector<KeyPoint>& keypoints, const Mat& mask=Mat() ) const = 0;

    /*
     * Remove keypoints that are not in the mask.
     * Helper function, useful when wrapping a library call for keypoint detection that
     * does not support a mask argument.
     */
    static void removeInvalidPoints( const Mat& mask, vector<KeyPoint>& keypoints );
};
```

3，（opencv-2.4.9\modules\features2d\src\detectors.cpp）

FeatureDetector:: detect()的函数实现：

```
void FeatureDetector::detect( const Mat& image, vector<KeyPoint>& keypoints, const Mat& mask ) const
{
    keypoints.clear();

    if( image.empty() )
        return;

    CV_Assert( mask.empty() || (mask.type() == CV_8UC1 && mask.size() == image.size()) );

    detectImpl( image, keypoints, mask );
}
```

即，FeatureDetector:: detect()为后继类留下了一个 detectImpl()的虚函数，不同的特征点根据自己的定义在这个函数中实现特征提取。

4，（opencv-2.4.9\modules\features2d\src\orb.cp）

在 orb.cpp 中找到了这个函数的实现，其调用了 ORB 重载的运算符"()"：

```
void ORB::detectImpl( const Mat& image, vector<KeyPoint>& keypoints, const Mat& mask) const
{
    (*this)(image, mask, keypoints, noArray(), false);
}
```

接下来便是 orb 自己的提取过程了：

```
void ORB::operator()( InputArray _image, InputArray _mask, vector<KeyPoint>& _keypoints,
                      OutputArray _descriptors, bool useProvidedKeypoints) const
```

5，流程图：

输入： InputArray _image,//输入的原图像
InputArray _mask, //掩膜图像
Vector<KeyPoint>& _keypoints,//输出关键点
OutputArray _descriptors, //输出描述子
Bool useProvidedKeypoints//是否用先前的关键点计算描述子

建立图像金字塔：

```
//建立图像金字塔容器
Vector<Mat> imagePyramid(levelsNum), maskPyramid(levelsNum);
//根据层数和尺度建立起金字塔
for (int level = 0; level < levelsNum; ++level)
{
  //部分代码
  Size sz(cvRound(image.cols*scale), cvRound(image.rows*scale));
  //先设置每一层的图像尺寸
  imagePyramid[level] = temp(Rect(border, border, sz.width, sz.height));
  //双线性插值对上一层图像进行缩放
  Resize(imagePyramid[level-1], imagePyramid[level], sz, 0, 0, INTER_LINEAR);
}
```

计算每一层图像上关键点：

```
computeKeyPoints(imagePyramid, maskPyramid, allKeypoints,
                 nfeatures, firstLevel, scaleFactor,
                 edgeThreshold, patchSize, scoreType)
{
  for (int level = 0; level < nlevels; ++level)
  {
    //提取FAST关键点
    FastFeatureDetector fd(20, true);
    fd.detect(imagePyramid[level], keypoints, maskPyramid[level]);
    //设置关键点所在层数和尺寸（层数越高尺寸越大）
    float sf = getScale(level, firstLevel, scaleFactor);
    keypoint->octave = level;
    keypoint->size = patchSize*sf;
    //计算关键点方向
    computeOrientation(imagePyramid[level], keypoints, halfPatchSize, umax);
  }
}
```

计算描述子并将关键点坐标投影到原图像上：

```
For (int level = 0; level < levelsNum; ++level)
{
  //在当前金字塔图像上计算描述子
  computeDescriptors(workingMat,Keypoints,desc,pattern,descriptorSize(),WTA_K);
  //将在金字塔上的图像坐标投影到原图像上
  float scale = getScale(level, firstLevel, scaleFactor);
  for (vector<KeyPoint>::iterator keypoint = keypoints.begin(),
                                  keypointEnd = keypoints.end();
       Keypoint != keypointEnd;
       ++keypoint)
  keypoint->pt *= scale;
}
```