Tesseract 笔记

目录

1 处理流程

1.1 程序主入口

api/tesseractmain.cpp

1.2 基本流程与主要方法

```
tesseract::TessBaseAPI api;

int rc = api.Init(datapath, lang, tesseract::OEM_DEFAULT,
                  &(argv[arg], argc - arg, NULL, NULL, false;)
```

Init 方法的原型如下:

```
int Init(const char* datapath, const char* language, OcrEngineMode mode,
         char **configs, int configs_size,
         const GenericVector<STRING> *vars_vec,
         const GenericVector<STRING> *vars_values,
         bool set_only_non_debug_params);
```

这些参数的意义分别是:

1. datapath: 数据目录，如果为空的话就用环境变量 TESSDATA_PREFIX
2. language: 语言参数，指定多语言时，第一个为主语言，其后的为附加语言
3. mode: 识别模式，有 TESSERACT、CUBE 以及混合的模式
4. configs:

在将 api 进行了初始化后，就创建 renderer:

```
if (renderer == NULL)
    renderer = new tesseract::TessTextRenderer();
```

renderer 的作用是根据指定的输出类型来产生不同类型的结果文件，默认情况下使用 txt，该类在 render.h 中被定义

然后就是对图片进行处理了:

```
if (pixs) {
    api.ProcessPage(pixs, 0, NULL, NULL, 0, renderer);
    pixDestroy(&pixs);
} else {
    FILE* fin = fopen(image, "rb");
    if (fin == NULL) {
        fprintf(stderr, "Cannot open input file: %s\n", image);
        exit(2);
    }
    fclose(fin);
    if (!api.ProcessPages(image, NULL, 0, renderer)) {
        fprintf(stderr, "Error during processing.\n");
        exit(1);
    }
}
```

识别的过程就是在这个调用过程中发生的

1.2.1 ProcessPages

在 ProcesPages 这个方法中，发生的事情是这样的。

1. 读取图片

   ```
   Pix *pix;
   pix = pixRead(filename);
   format = pixGetInputFOrmat(pix);
   ```

2. 初始化结果文件

   ```
   if (renderer && !renderer->BeginDocument(kUnknownTitle)) {
       success = false;
   }
   ```

   render.h 中的说明如下:

   ```
   Starts a new document with the given title.
   This clears the contents of the output data.
   ```

3. 判断图片是单页的图片还是多页的 tiff，并根据不同的格式进行处理

   ```
   // The file is a multi-pages tiff.
   if (npages > 0) {
       pixDestroy(&pix);
       for (; page < npages; ++page) {
           pix = pixReadTiff(filename, page);

           if (pix == NULL) break;

           // ...

           success &= ProcessPage(pix, page, filename, retry_config,
                                  timeout_millisec, renderer);
           pixDestroy(&pix);
   ```

```
        // ...

        if (tesseract_->tessedit_page_number >= 0 || npages == 1) {
            break;
        }
    }
}
else {
    // The file is not a tiff file.
    if (pix != NULL) {
        success &= ProcessPage(pix, 0, filename, retry_config,
                               timeout_millisec, renderer);
        pixDestroy(&pix);
    }
    else {
        // The file is not an image file, so try it as a list of filenames.
        // ...
    }
}
```

从这里看到，具体的处理是在 **ProcessPage** 这个方法中进行的。

4. 单页处理 TessBaseAPI::ProcessPage

函数原型为:

```
bool TessBaseAPI::ProcessPage(Pix* pix, int page_index, const char* filename,
                              const char* retry_config, int timeout_millisec,
                              TessResultRenderer* renderer);
```

函数说明为:

```
Recognizes a single page for ProcessPages, appending the text to text_out.
The pix is the image processed - filename and page_index are metadata
used by side-effect processes, such as reading a box file or formatting
as hOCR.
If non-zero timeout_millisec terminates processing after the timeout.
If non-NULL and non-empty, and some page fails for some reason,
the page is reprocessed with the retry_config config file. Useful
for interactively debugging a bad page.
The text is returned in renderer. Returns false on error.
```

从说明上来看，该函数做了以下事情:

1. 对单页的图片进行识别
2. 将识别结果追加到 text_out 中，这个 text_out 应该是一个 buffer 之类的东西

这个函数里核心的代码是这几行(在不设置超时， psm 不为 PSM_OSD_ONLY:0 PSM_AUTO_ONLY:2 的情况下):

```
if (timeout_millisec > 0) {
    // ...
}
else if (tesseract_->tessedit_pageseg_mode == PSM_OSD_ONLY ||
         tesseract_->tessedit_pageseg_mode == PSM_AUTO_ONLY) {
    // ...
}
else {
    failed = Recognize(NULL) < 0;
}
```

也就是说，真正的识别过程是在 Recognize 这个函数里进行的。

此外，发现这么几句:

```
if (tesseract_->tessedit_write_images) {
    Pix* page_pix = GetThresholdedImage();
    pixWrite("tessinput.tif", page_pix, IFF_TIFF_G4);
}
```

看样子应该存在 "tessedit_write_image" 这么一个选项，可以将二值化后的图片存下来。

1.2.2 Recognize

该函数位于 api/baseapi.cpp 中，函数原型为:

```
int TessBaseAPI::Recognize(ETEXT_DESC *monitor);
```

虽然有一个参数，但这个参数给的值都是 NULL，意义为何暂不追究。

1. FindLines

   在 Recognize 中首先就调用了 FindLines 这个方法，这个方法同样是 TessBaseAPI 这个类的方法

   ```
   int TessBaseAPI::FindLines();
   ```

   对该函数的说明为:

   ```
   Find lines from the image making th BLOCK_LIST.
   ```

   再做了初步的检查后，FindLines() 中调用了 PrepareForPageseg() 这个方法

   ```
   tesseract_->PrepareForPageseg();
   ```

   然后就是调用 SegmentPage 这个方法:

   ```
   if (tesseract_->SegmentPage(input_file_, block_list_, osd_tess, &osr) < 0)
       return -1;
   ```

   该函数(位于 ccmain/pagesegmain.cpp 中)的说明为:

   ```
   Segment the page according to the current value of tessedit_pageseg_mode.
   pix_binary_ is used as the source image and should not be NULL.
   On return the blocks list owns all the constructed page layout.
   ```

   这里说 pix_binary_ 被用作输入的图像数据，可见在此之前二值化已经完成，不过从目前跟进的流程来看，也不知道二值化是在哪里进行的。

   这个函数首先查找有没有所谓的 UNLV zone 文件，有的话就读取它:

   ```
   if (!PSM_COL_FIND_ENABLED(pageseg_mode) &&
       input_file != NULL && input_file->length() > 0) {
       STRING name = *input_file;
       const char* lastdot = strrchr(name.string(), '.');
       if (lastdot != NULL)
           name[lastdot - name.string()] = '\0';
       read_unlv_file(name, width, height, blocks);
   }
   ```

   UNLV zone 文件的作用暂且不考虑 ,这里调试跟进的条件是没有这个文件，所以这里直接跳过就好。

   在没有 UNLV zone 文件的条件下，处理的流程是:

   ```
   BLOCK_IT block_it(blocks);
   BLOCK* block = new BLOCK("", TRUE, 0, 0, 0, 0, width, height);
   block->set_right_to_left(right_to_left());
   block_it.add_to_end(block);
   ```

   对这段代码有一段简单说明:

   ```
   No UNLV file present. Work according to the PageSegMode.
   First make a single block covering the whole image.
   ```

   从这个说明和代码里来看，它是建立了一个 **block**, 细节暂且不管，可以肯定的是它保存了图片的宽度和高度信息。

   完了以后有一个判断:

```
if (PSM_OSD_ENABLED(pageseg_mode) || PSM_BLOCK_FIND_ENABLED(pageseg_mode) ||
    PSM_SPARSE(pageseg_mode)) {
    // ...
} else {
    // ...
}
```

第一个条件在 调试条件下不成立 ，所以直接进入了 else 分支中，该分支中的代码为:

```
deskew_ = FCOORD(1.0f, 0.0f);
reskew_ = FCOORD(1.0f, 0.0f);
if (pageseg_mode == PSM_CIRCLE_WORD) {
    Pix* pixcleaned = RemoveEnclosingCircle(pix_binary_);
    if (pixcleaned != NULL) {
        pixDestroy(&pix_binary_);
        pix_binary_ = pixcleaned;
    }
}
```

后面那个 if 语句也没有起作用。

之后调用了 AutoPageSeg 这个方法，在里面调用了 find_components 函数(textord/tordmain.cpp)

然后用了 TextordPage 这个函数(textord/textord.cpp)，该函数的作用是

```
make the textlines and words inside each block.
```

最后调用:

```
tesseract_->PrepareForTessOCR(block_list_, osd_tess, &osr);
```

1.3 图像二值化

下面的说法是错误的，二值化不是在 SetImage 前完成的，而是在之后，具体的地方在 api/baseapi.cpp 中的 Threshold 方法内，单从 Tesseract 本身来说，调用栈如下:

```
#0  tesseract::TessBaseAPI::Threshold (this=0x7fffffffcce0, pix=0x80c488) at baseapi.cpp:1959
#1  tesseract::TessBaseAPI::FindLines (this=0x7fffffffcce0) at baseapi.cpp:1996
#2  tesseract::TessBaseAPI::Recognize (this=0x7fffffffcce0, monitor=0x0) at baseapi.cpp:814
#3  tesseract::TessBaseAPI::ProcessPage (this=0x7fffffffcce0, pix=0x81e5b0, page_index=0,
    filename=0x7fffffffdd65 "/home/linusp/data/math_stem/math/077.jpg", retry_config=0x0, timeout_millisec=0,
    renderer=0x83f340) at baseapi.cpp:1158
#4  tesseract::TessBaseAPI::ProcessPages (this=0x7fffffffcce0,
    filename=0x7fffffffdd65 "/home/linusp/data/math_stem/math/077.jpg", retry_config=0x0, timeout_millisec=0,
    renderer=0x83f340) at baseapi.cpp:1041
#5  main (argc=5, argv=0x7fffffffd928) at ../api/tesseractmain.cpp:320
```

从代码上来看，应该是使用的 OTSU 方法进行二值化

经验证，二值化最迟应该是在

```
api->SetImage(pix);
```

完成后就完成了，所以可以将二值化的操作定位到两个可能的步骤上:

1. pixRead(), 即读取图片时
2. SetImage(), 即将图片数据拷贝到 API 结构中时

```
void TessBaseAPI::SetImage(const unsigned char* imagedata,
                           int width, int height,
                           int bytes_per_pixel, int bytes_per_line) {
    if (InternalSetImage())
        thresholder_->SetImage(imagedata, width, height,
                               bytes_per_pixel, bytes_per_line);
}
```

thresholder_ 是 **ImageThresholder** 类型的对象指针，定义在

```
ccmain/thresholder.h
```

中，在相应的 cpp 文件中实现。

1.3.1 InternalSetImage

核心方法:

```
if (thresholder_ == NULL) {
    thresholder_ = new ImageThresholder;
}
```

构造了一个 ImageThresholder 对象

1.3.2 ImageThresholder::SetImage

ccmain/thresholder.cpp

函数说明:

```
Pix vs raw. which to use? Pix is the preferred input for efficiency,
since raw buffers are copied.
SetImage for Pix clones its input, so the source pix may be pixDestroyed immediately after,
but may not go away until after the Thresholder has finished with it.
```

意义不明

刚进入该函数时， pix 还是未二值化的图像，通过 pixDisplay 可以知道这一点

在该函数尾部调用了一个:

```
void ImageThresholder::Init();
```

但这个函数只是设置了一下图片的大小等相关信息

1.4 Shiro-Rekha (top-line) splitting

在 tesseract::Tesseract::PrepareForPageseg 中被调用

从网上查到的信息来看，似乎是用来切割孟加拉语、錫克教文等稀奇古怪的文字用的，实际上并没有进行调用，因为传入的参数:

```
max_pageseg_strategy
```

的值为 tesseract::ShiroRekhaSplitter::NO_SPLIT

上面说到的文字有一个特点是它们头上都有一横

信息来源:

- google code
- Shirorekha

2 函数解析

2.1 Tesseract::AutoPageSeg(ccmain)

函数说明:

```
Auto page segmentation. Divide the page image into blocks of uniform
text linespacing and images.

Resolution (in ppi) is derived from the input image.

The output goes in the blocks list with corresponding TO_BLOCKs in the
to_blocks list.
```

也就是说，区块的划分是在这里进行的

2.1.1 Tesseract::SetupPageSegAndDetectOrientation(ccmain)

```
Sets up auto page segmentation, determines the orientation, and corrects it.
Somewhat arbitrary chunk of functionality, factored out of AutoPageSeg to
facilitate testing.
```

按照函数说明，这里进行的是文本朝向的确定和更正

在这里又调用了另外一个函数:

```
LineFinder::FindAndRemoveLines(source_resolution_,
                               textord_tabfind_show_vlines,
                               pix_binary_,
                               &vertical_x, &vertical_y, music_mask_pix,
                               &v_lines, &h_lines);
```

然后调用了 FindImages 这个方法

```
Pix* ImageFinde::FindImages(Pix *pix);
```

然后就是调用了 find_components 这个方法:

```
void Textord::find_components(Pix *pix, BLOCK_LIST *blocks, TO_BLOCK_LIST *to_blocks);
```

然后调用了 SetupAndFilterNoise 方法:

```
void ColumnFinder::SetupAndFilterNoise(Pix *photo_mask_pix, TO_BLOCK *input_block);
```

以下是对这几个函数的阅读分析笔记

1. LineFinder::FindAndRemoveLines(textord/linefind.cpp)

   调用了方法 LineFinder::GetLineMasks(textord/linefind.cpp)

   ```
   Most of the heavy lifting of line finding. Gvein src_pix and its separte resolution, return image masks.
   ```

   将结果保存到 pix_music_mask 中后，调用 FindAndRemoveVLines ，不过在这里，由于调用完 GetLineMasks 后，pix_vline 和 pix_hline 都被清空，在 FindAndRemoveVLines 中未进行任何操作即退出。后面的 FindAndRemoveHLines 同样。

   然后呢，检查 pix_music_mask 中是否有内容，如果有，就用源图像减去 pix_music_mask:

   ```
   if (pix_music_mask != NULL && *pix_music_mask != NULL) {
       if (pixa_display != NULL)
           pixaAddPix(pixa_display, *pix_music_mask, L_CLONE);

       pixSubtract(pix, pix, *pix_music_mask);
   }
   ```

   结果如下图所示: remove_music.png

   然后就结束了。

   当然了，如果调用完 GetLineMasks 后 pix_vline、pix_hline 未被清空，是会进行其他的操作的，这里就不作深入了解了。

   1. DONE LineFinder::GetLineMasks

      以原始图像 test.png 为例:

test.png

二值化后的结果为:

test_bin.png

在这个函数里，依次进行了以下操作:

- pixCloseBrick(算子: 1,1)

  结果保存在 pix_closed 中，由于算子为(1,1),直接复制了一副数据并返回

  这个方法从注释上来看是用来填充图像中的空洞的

- pixOpenBrick(算子: 3,3)

  使用 pix_closed 作为输入，先进行了两次膨胀操作，然后进行了两次腐蚀操作(待确认)，将图像中的小块的像素全部擦除掉了 pixErode x 2, pixDilate x 2.

  结果保存为 pix_solid

- pixSubstract

  用 pix_closed 减去 pix_solid，结果保存在 pix_hollow 中

  pix_hollow.png

- pixOpenBrick

  分别用不同的算子(3,1和1,3)，用 pix_hollow 作为输入，得到两个结果: pix_vline 和 pix_hline

  pix_vline: pix_vline.png

  pix_hline: pix_hline.png

然后调用 pixZero 得到 v_empty 和 h_empty:

```
l_int32 v_empty = 0;
l_int32 h_empty = 0;
pixZero(*pix_vline, &v_empty);
pixZero(*pix_hline, &h_empty);
```

在 v_empty 和 h_empy 都是 0 的情况下，用 pix_vline 、 pix_hline 和 pix_closed 作为输入来调用 FilterMusic 方法，并将结果保存到 pix_music_mask 中

FilterMusci 除开会输出结果到 pix_music_mask 中外，还会修改 pix_vline、 pix_hline 和 v_emtpy、h_empty，在使用上述的 test_bin.png 的情况下， pix_vline、pix_hline 会被清空，而 v_empty 和 h_empty 会被置为 1(v_empty、h_empty本来就分别和 pix_vline、pix_hline相关联)

如果调用完 FilterMusic 后 pix_vline、pix_hline都被清空，则结束。

1. DONE pixZero

   判断图像中是否有前景像素点，当无前景像素点只有背景像素点(即为 zero)时返回 1 ，否则返回 0

2. DONE tesseract::FilterMusic(textord/linefind.cpp)

   先用 pix_vline 和 pix_hline 作为输入取交集得到了 intersection_pix intersection_pix.png

   然后用 pix_vline 作为输入，调用了 pixConnComp 这个 Leptonica 库里的方法，从名字以及注释来看是用来提取连通分量的。在这里其实就是得到了 pix_vline 中所有的直线的边界框。

   然后初始化输出 music_mask(Pix *) 为 NULL

   随后有一个循环，从得到的边界框中逐个访问，并进行处理

```
for (i = 0; i < nboxes; ++i) {
    Box *box = boxaGetBox(boxa, i, L_CLONE);
    l_int32 x, y, box_width, box_height;
    boxGetGeometry(box, &x, &y, &box_width, &box_height);

    int joins = NumTouchingIntersections(box, intersection_pix);

    if (joins >= 5 && (joins - 1) * max_stave_height >= 4 * box_height) {
        if (music_mask == NULL)
            music_mask = pixCreate(pixGetWidth(pix_vline), pixGetHeight(pix_vline), 1);

        pixSetInRect(music_mask, box);
    }

    boxDestroy(&box);
}
```

boxGetGeometry 这个方法就不必细说了，很简单地就能理解它的用途 —— 获取连通分量的边界框(bounding box)的尺寸和起始坐标。值得注意的是 NumTouchingIntersections 这个方法 和 pixSetInRect 这个方法

需要注意的是，这里以及 NumTouchingIntersections 方法中对 pixConnComp 的调用都是提取的 8-向连通分量

处理完这个循环后，得到的结果为: music_mask.png

得到 music_mask 后，调用 pixSeedfillBinary 对其进行形态学重建处理:

```
pixSeedfillBinary(music_mask, music_mask, pix_closed, 8);
```

重建后的结果为: music_mask_2.png

然后又对重建后的结果提取连通分量，并对每个分量逐个处理:

```
Boxa* boxa = pixConnComp(music_mask, NULL, 8);

// Iterate over the boxes to find music components.
// ^
// |
// +----上面这条注释中的 music components 是几个意思
int nboxes = boxaGetCount(boxa);

for (int i = 0; i < nboxes; ++i) {
    Box* box = boxaGetBox(boxa, i, L_CLONE);

    // 复制连通分量边界框所选中的 music_mask 中的区域
    Pix* rect_pix = pixClipRectangle(music_mask, box, NULL);

    // 计算该区域中的像素(前景?)点个数
    l_int32 music_pixels;
    pixCountPixels(rect_pix, &music_pixels, NULL);
    pixDestroy(&rect_pix);

    // 复制连通分量边界框所选中的源图像中的区域
    rect_pix = pixClipRectangle(pix_closed, box, NULL);

    // 计算该区域中的像素(前景?)点个数
    l_int32 all_pixels;
    pixCountPixels(rect_pix, &all_pixels, NULL);
    pixDestroy(&rect_pix);

    // kMinMusicPixelFraction = 0.75
    // 对同一区域，如果 music_mask 中的像素点个数少于源图像的某个比例
    // 则将 music_mask 中该区域的像素点清除
    if (music_pixels < kMinMusicPixelFraction * all_pixels) {
        // False positive. Delete from the music mask.
        pixClearInRect(music_mask, box);
    }

    boxDestroy(&box);
}
```

然后用 pixZero 对处理后的结果进行检查、判断:

```
l_int32 no_remaining_music;
pixZero(music_mask, &no_remaining_music);
if (no_remaining_music) {
    pixDestroy(&music_mask);
} else {
```

```
    // 用 pix_vline 减去 music_mask，保存在 pix_vline 中
    pixSubtract(pix_vline, pix_vline, music_mask);
    // 用 pix_hline 减去 music_mask，保存在 pix_hline 中
    pixSubtract(pix_hline, pix_hline, music_mask);

    pixZero(pix_vline, v_empty); // *v_empty: 0
    pixZero(pix_hline, h_empty); // *h_empty: 0
}
```

pixZero 判断的是图像中是否有像素点(前景?)

1. DONE NumTouchingIntersections(textord/linefind.cpp)

   函数注释为:

   ```
   Returns the number of components in the intersection_pix touched by line_box
   ```

   即对给定边界框，计算指定图像在该区域中的连通分量数量，其处理方法也很简单，就是将指定区域的数据拷贝出来，然后提取连通分量，计算数量:

   ```
   if (intersection_pix == NULL) return 0;
   Pix* rect_pix = pixClipRectangle(intersection_pix, line_box, NULL);
   Boxa* boxa = pixConnComp(rect_pix, NULL, 8);
   pixDestroy(&rect_pix);

   if (boxa == NULL) return false;
   int result = boxaGetCount(boxa);
   boxaDestroy(&boxa);
   return result;
   ```

   另外注意到前几个 box ，调用该函数的返回值都是 7 ，经调用 pixDisplay 发现这些 box 所包含的都是最长那几条线。

   猜想 **pixConnComp** 这个方法对结果按照边界框大小进行了排序

2. DONE pixSetInRect(leptonica: src/pix2.c)

   ```
   boxGetGeometry(box, &x, &y, &w, &h);
   pixRasterop(pix, x, y, w, h, PIX_SET, NULL, 0, 0);
   ```

   这里的 PIX_SET 是一个预设的常量，在我们这里应该是表示黑色的 0 这个值

   应该是将一个矩形区域里面的内容拷贝到了目标图像中

3. DONE pixSeedfillBinary(leptonica: src/seedfill.c)

   形态学重建，结果(保存到 music_mask 中)为 music_mask_2.png

2. ImageFind::FindImages()

   函数注释为:

   ```
   Finds image regions within the BINARY source pix (page image) and returns
   the image regions as a mask image.
   The returned pix may be NULL, meaning no images found.
   If not NULL, it must be PixDestroyed by the caller.
   ```

   过滤太小的图片:

   ```
   // Not worth looking at small imags
   if (pixGetWidth(pix) < kMinImageFindSize ||
       pixGetHeight(pix) < kMinImageFindSize) {
       return pixCreate(pixGetWidth(pix), pixGetHeight(pix), 1);
   }
   ```

   过滤条件是 100x100 ,小于这个尺寸的图片不进行图片查找

   然后调用了 pixReduceRankBinaryCascade 方法，该方法内调用了两个函数:

   - makeSubSampleTab2x

- pixReduceRankBinary2

pixReduceRankBinaryCascade 方法调用前有一条注释:

```
Reducee by factor 2
```

是不是说把图片按 1:2 的比例缩小了 ?

调用该方法后，得到的图片保存在 pixr 中，如下图所示: reduce_rank_binary_cascade.png

然后调用方法 pixGenHalftoneMask，该函数的说明为:

```
Halftone region extraction
```

该函数是 Leptonica 中的方法，从其方法来看，是返回了一个 "halftone mask" ，在使用 test_bin.png 这张图片时，该结果是一张空白的图像

传入 pixGenHalftoneMask 的有一个参数 ht_found, 如果得到的 halftone mask 是空白的，该变量的值会被置为 0 ，这种情况下函数 FindImages 将会直接返回。

大概还是找的图片不适合吧，里面虽然有图，但那个图是很规则的围棋棋盘，在 FindAndRemoveLines 里就被去掉了

1. pixReduceRankBinaryCascade(leptonica: src/binreduce.c)

   函数定义:

   ```
   PIX *pixReduceRankBinaryCascade(PIX *pixs,
                                   l_int32 level1,
                                   l_int32 level2,
                                   l_int32 level3,
                                   l_int32 level4);
   ```

   该函数将会最多进行四次按比例2倍缩小的操作: 从 level1 到 level4 ，如果是大于 0 的数，则用它作为 rank 阈值来对图像进行缩小操作，并将结果作为待操作的对象准备进行下一次的操作；一旦遇到不大于 0 的 level ，操作就停止，并把结果返回。

   在 ImageFind::FindImages 中对该函数的调用是:

   ```
   pixpixReduceRankBinaryCascade(pixs, 1, 0, 0, 0);
   ```

   故是将图片缩小了两倍。

   1. makeSubsampleTab2x

      该函数应该是一个辅助性功能函数，但是那个说明不太看得懂:

      ```
      This table permutes the bits in a byte, from
          0 4 1 5 2 6 7 3
      to
          0 1 2 3 4 5 6 7
      ```

      看说明是说重排一个字节里的位? 不过那个 from 是什么鬼，原始的位序不应该是下面这样的嘛?

      ```
      0 4 1 5 2 6 7 3
      ```

      或者上面这个序列是指得什么特殊情况?

      ```
      0   1   2       3
        4   5   6   7
      ```

      该函数没有输入，只有输出，使用 test_bin.png 时，得到一个空的字符串指针(l_uint8 *)。

   2. pixReduceRankBinary2

      该方法返回一个 Pix * 对象， 用 **pixDisplay** 查看发现该图片比原图要小

传入该函数的参数有三个:

```
PIX *pixReduceRankBinary2(PIX *pixs,
                          l_int32 level,
                          l_uint8 *intab);
```

下面是对该函数的注释:

```
Notes:
  (1) pixd is downscaled by 2x from pixs.
  (2) The rank threshold specifies the minimum number of ON
      pixels in each 2x2 region of pixs that are required to
      set the corresponding pixel ON in pixd.
  (3) Rank filtering is done to the UL corner of each 2x2 pixel block,
      using only logical operations.  Then these pixels are chosen
      in the 2x subsampling process, subsampled, as described
      above in pixReduceBinary2().
```

稍微翻译一下:

1. pixd 是从 pixs 按比例缩小两倍得到的结果
2. rank 阈值(参数中的 level) 设定了与 pixd 中某个 ON 像素对应的 pixs 中的 2x2 区域中最小 ON 像素个数

   这里的 ON pixels 不知道该如何翻译，由于是二值图像可以猜测是黑色还是还是白色，即可能是代表前景或者背景的像素。

2. pixGenHalftoneMask(leptonica: src/pageseg.c)

函数注释:

```
Halftone region extraction.
```

该函数内部调用了这么几个函数:

1. pixReduceRankBinaryCascade

   调用该方法时传入的 level 序列为 4,4,3,0 ，于是就将图片缩小了 8 倍

   缩小至 8 倍(源图片的 16 倍缩小)后，图像中已经全是白色(背景色)

   此过程中的结果如下图所示: reduce_seq.png

2. pixOpenBrick

   由于图像全为白色, OpenBrick 的操作已经没有什么意义

   对了，可以说 OpenBrick 是开操作吧，因为是先腐蚀然后再膨胀的。

3. pixExpandReplicate

   此处将上一步的结果作为输入，然后扩大了 8 倍:

   ```
   pixhs = pixExpandReplicate(pixt2, 8);
   ```

   当然了，结果还是空白的图一张，结果保存为 pixhs

4. pixCloseSafeBrick

   形态学闭操作? 从注释里来看，这个方法区别于 pixCloseBrick 的地方在于它为图像添加了虚拟的“边界”，以使对所有实际像素的操作都是一致的

   该函数的输入不是上一步的结果，而是传入 pixGenHalftoneMask 的图像 pixs (源图像2倍比例缩小)，结果如下: pix_close_safe_brick.png

   结果保存为 pixhm

5. pixSeedfillBinary

形态学重建，结果保存在 pixd 中

```
pixd = pixSeedfillBinary(NULL, pixhs, pixhm, 4);
```

此时的结果应该就是所谓的 halftone mask 了。得到的结果是一张全空白的图片，保存在 pixd 中

6. pixSubstract

当传入函数 pixGenHalftoneMask 的第二个参数不为 NULL 时，用传入图像 pixs 减去上一步得到的 halftone mask 也就是 pixd，并将结果保存在这个第二个参数中。

3. Textord::find_components(textord/tordmain.cpp)

函数说明为:

```
Find the C_OUTLINEs of the connected components in each block, put them in C_BLOBs, and filter them by size,
```

调用了方法下面这个方法来进行边缘检测

```
void extract_edges(Pix *pix, BLOCK *blocks);
```

方法是在一个循环中被调用的:

```
BLOCK_IT block_it(blocks);

for (block_it.mark_cycle_pt(); !block_it.cycled_list(); block_it.forward()) {
    BLOCK * block = block_it.data();

    if (block->poly_block() == NULL) {
        extract_edges(pix, block);
    }
}
```

这里的 blocks 是传入的参数， 来自传入方法 SetupPageSegAndDetectOrientation 的参数 blocks(BLOKCS)， 来自传入方法 AutoPageSeg 的参数 blocks(BLOCK_LIST), 来自传入方法 SegmentPage 的参数 blocks(BLOCK_LIST)， 为 TessBaseAPI 的数据成员 block_list_(BLOCK_LIST)

1. extract_edges(textord/edgblob.cpp)

函数说明:

```
Run the edge detector over the block and return a list of blobs.
```

函数定义为:

```
void extract_edges(Pix *pix, BLOCK *blocks);
```

在 find_components 中被调用时，blocks 中只有一个 block (以 data/math_stem/math/077_bin.jpg 作为输入)

```
box = {
    bot_left = {xcoord = 0, ycoord = 0},
    top_right = {xcoord = 1175, ycoord = 429},
    index = 0
}
```

然后调用了 block_edges 方法:

```
C_OUTLINE_LIST outlines;
C_OUTLINE_IT out_it = &outlines;
block_edges(pix, block, &out_it);
```

在调用完 block_edges 方法后，调用了 outline_to_blobs 方法:

```
outlines_to_blobs(block, bleft, tright, &outlines);
```

1. block_edges(textord/scanedg.cpp)

函数说明:

```
Extract edges from a PDBLK
```

函数定义:

```
void block_edges(Pix *t_pix, PDBLK *block, C_OUTLINE_IT *outline_it);
```

依赖类型、结构:

- BLOCK_LINE_IT/BLOCK_LINE
- CRACKEDGE

一个两层循环:

```
// tright 和 bleft 都是表示坐标的对象, tright 表示右上角的点， bleft 表示左下角的点
// 处理的对象是图片，并且以图像左下角作为坐标系原点，用 x 表示列，用 y 表示行，故外
// 层循环的含义是从顶行开始逐行往下处理
for (int y = tright.y() - 1; y > bleft.y() - 1; y--) {

    if (y > bleft.y() && y < tright.y()) {
        // 这里的 x 表示列，即从第一列起处理，对所有列进行同样的操作
        // 这里所做的操作是，将该行的所有像素拷贝到 bwline 这个数组中去
        // GET_DATA_BIT 是 leptonica 中的宏
        for (int x = 0; x < block_width; ++x) {
            bwline[x] = GET_DATA_BIT(line, x + bleft.x()) ^ 1;
        }

        // 复制完整行的数据后，调用 make_margins 方法
        make_margins(block, &line_it, bwline, margin, bleft.x(), tright.x(), y);
    }
    else {
        // 再进行到最后一次循环时进入该分支
        memset(bwline, margin, block_width * sizeof(bwline[0]));
    }

    // 每次循环都调用 line_edges 方法
    line_edges(bleft.x(), y, block_width, margin, bwline, ptrline, &free_cracks, outline_it);
}
```

其中 **GET_DATA_BIT** 是 leptonica 中定义的宏，其定义为:

```
#define GET_DATA_BIT(pdata, n) ((*((l_uint32 *)(pdata) + ((n) >> 5)) >> (31 - ((n) & 31))) & 1)
```

与之类似的还有一个 **SET_DATA_BIT**

```
#define SET_DATA_BIT(pdata, n) (*((l_uint32 *)(pdata) + ((n) >> 5)) |= (0x80000000 >> ((n) & 31)))
```

1. make_margins

函数说明为:

```
Get an image line and set to margin non-text pixels
```

使用 015.jpg ，结果该函数内什么都没干

该函数的作用为: 将 block 和 block 内部的 rect 之间用指定的像素填充，即其函数名的字面意思

```
make_margins(block, &line_it, bwline, margin, bleft.x(), tright.x(), y);
```

函数签名为:

```
void make_margins(          //get a line
    PDBLK *block,           //block in image
```

```
    BLOCK_LINE_IT *line_it,  //for old style
    uinT8 *pixels,           //pixels to strip
    uinT8 margin,            //white-out pixel
    inT16 left,              //block edges
    inT16 right,
    inT16 y                  //line coord
    );
```

各参数为:

- block:

  输入,只是用来判断该 block 的类型

- line_it:

  输入,用来获取该行的信息，如起点、宽度等

- bwline:

  输出,存储了像素值的数组

- margin:

  输入,边界像素(预设)

- bleft.x():

  输入

- tright.x():

  输入

- y:

  输入

总而言之，该方法只会对 bwline 进行修改，其他参数都只是作为输入。

2. line_edges

函数说明为:

```
Scan a line for edges and update the edges in progress.
When edges close into loops, send them for approximation.
```

重点在这个方法上。

调用该参数时(block_edges中)，形式为:

```
line_edges(bleft.x(), y, block_width, margin, bwline, ptrline, &free_cracks, outline_it);
```

传入的各个参数及作用为:

- bleft.x(): x

  输入, 该行的起始点，在这里都是0

- y: y

  输入, 该行的行号

- block_width: xext

  输入, 该行的宽度

- margin: uppercolour

  输入, 该行的边界像素值

- bwline: bwpos

输入, 该行每个像素点的像素值

- ptrline: prevline

  输入和输出， CRACKEDGE **类型，对第一行来说，内部全为 NULL，但对后面的行来说，就是 "上一行" 的什么东西了。

  虽然对第一行来说，作为输入时是全为 NULL ，但在处理过程中会对它进行修改

- free_cracks: free_cracks

  输入和输出, CRACKEDGE *类型

- outline_it: outline_it

  C_OUTLINE_IT *类型

代码里喜欢用 xext 来表示 x 方向上的长度，此外名为 "colour" 的都是表示颜色(或像素值)的变量。

该函数内部定义了一些局部变量，如下:

```
CrackPos pos = {free_cracks, x, y };
int xmax;                       // max x coord
int colour;                     // of current pixel
int prevcolour;                 // of previous pixel
CRACKEDGE *current;             // current h edge
CRACKEDGE *newcurrent;          // new h edge

xmax = x + xext;                // max allowable coord
prevcolour = uppercolour;       // forced plain margin
current = NULL;                 // nothing yet
```

代码中有一个宏名为 **FLIP_COLOUR** ，定义在同一个 cpp 文件里，其定义为:

```
#define FLIP_COLOUR(pix) (1-(pix))
```

从名字来看，应该是 "翻转" 的意思，因为这里的 uppercolour 在传入的时候值是 0 ，经过这个宏的操作后就会在 0 和 1 之间来回变化。

主体循环代码为分为两部分

```
for (; pos.x < xmax; pos.x++, prevline++) {
    colour = *bwpos++;
    if (*prevline != NULL) {
        // ...
    }

    else {
        // ...
    }
}
```

这里的 prevline 决定了该走哪个分支，所以它发生变化的地方是很关键的。从循环语句可以看到，prevline 这个指针也是在逐次往后移动的，所以对第一行来说，如果没有哪个操作能对 当前 **prevline** 指针指向地址的下一个地址 进行修改的话，循环将总是进入第二部分。而在循环第二部分涉及到的操作只有 v_edge 和 h_edge 两个方法，而这两个方法会对当前地址内部进行修改，不会修改 下一个地址 ，所以在处理第一行时，prevline 将会 只作为输出 。

实际上，这里在开始处理时， prevline 是上一行在经过该函数后得到的处理结果，然后又作为本次处理时的一个输入，因为是用完就丢弃，在作为输入用完后就地保存当前的处理结果，然后移动该指针。

而可以看到的是，在有对 prevline 进行赋值的地方，结果 大都是 函数 v_edge 的返回结果，此外在循环第一部分中还有一处将当前 prevline 指向的值修改为 NULL，那么修改 prevline 的条件可以总结如下:

- colour == prevcolour

  此时

  ```
  *prevline = NULL;
  ```

即如果当前像素点与上一个像素点的像素值相等时，就将 该点对应的输出设置为 **NULL** ，该操作是发生在循环第一部分中的，即要求上一行的对应的结果不为 NULL —— 如果同为 NULL 的话，直接保留即可，不需要修改。

- colour != precolour

  此时

  ```
  *prevline = v_edge(colour - prevcolour, *prevline, &pos);
  ```

  或者

  ```
  *prevline = v_edge(colour - prevcolour, current, &pos);
  ```

  即如果当前像素点与前一个像素点的像素值 不同 时，就调用 v_edge 方法，并将其返回结果填充到当前点对应的输出结果上。

要理解该函数，必须要弄明白以下各部分的含义:

- colour: 该行当前点的像素值(0或1)
- prevline:

  上一行中产生的 "边缘点" 数组，只在像素点发生变化的位置才有值，对于不变的区域，其值都是 NULL。

- uppercolour:

  对该传入参数，注释说明为:

  ```
  start of prev line
  ```

  也就是说，在传入的时候，其值是上一行的第一个点的像素值。由于在之前调用了 make_margins，就认为第一个点就是 margin 的值，而它的翻转之所以在 *prevline 不为 NULL 的时候，是因为 *prevline 不为 NULL 的时候，表示前后的点的像素值发生了改变。

- prevcolour: 该行前一个点的像素值(0或1)
- current:
- v_edge:
- h_edge:

  调用 h_edge 的方法为:

  ```
  current = h_edge(uppercolour - colour, *prevline, &pos);
  current = h_edge(uppercolour - colour, NULL, &pos);
  newcurrent  = h_edge(uppercolour - colour, *prevline, &pos);
  ```

  函数第一部分为:

  ```
  if (*pos->free_cracks != NULL) {
      newpt = *pos->free_cracks;
      *pos->free_cracks = newpt->next;
  }
  else {
      newpt = new CRACKEDGE;
  }
  ```

  这里的 pos->free_cracks 可见是一个单链表，而这里的意思就是，如果这个单链表不为空，就取一个节点，并将单链表往后移动；否则就创建一个新的节点。

  第二部分:

  ```
  newpt->pos.set_y(pos->y + 1);  // coords of pt
  newpt->stepy = 0;              // edge is horizontal
  ```

第一句不知道 为何要将当前位置的 **y** 加**1** ，而从第二句来看，可知 CRACKEDGE 中的 stepy 应为纵轴上的步长或类似的量。注意在调用 line_edges 时是从最上面的行逐行往下处理的，而且行号从上到下是逐渐减小的，那么这样的话就有两种可能:

1. pos.y 代表的是上面一行的行号
2. pos.y 代表的是本行的行号，+1 是为了从 1 开始计数

第三部分是一个分支语句:

```
if (sign > 0) {
newpt->pos.set_x(pos->x + 1);     // start location
newpt->stepx = -1;
newpt->stepdir = 0;
} else {
newpt->pos.set_x(pos->x);        // start location
newpt->stepx = 1;
newpt->stepdir = 2;
}
```

这里有两个量: stepx 和 stepdir。结合 v_edge 中的 stepdir 来看，其可能的值有 0, 1, 2, 3 四个，应该是代表上下左右四个方向，但具体是如何对应的，这个还不清楚。至于 stepx 是代表什么呢?在第四部分的代码中有用到该值，用来和待加入的边缘进行比较:

```
if (newpt->pox.x() + newpt->stepx == join->pos.x() && newpt->pos.y() == join->pos.y()) {
// ...
}
```

这里的一个比较条件是: 两者的 y 值相等 —— 这说明什么呢?

另外，h_edge 这个函数，从其名称及注释上来看应该是用于 生成一个新的水平边缘点然后加入到给定的边缘中去

- join_edges

与 prevline 相对的，还有一个名为 current 的 CRACKEDGE 类型的指针，用来存储函数 h_edges 的返回结果，但该函数中是如何将该结果保存的呢?该变量是函数内部局部变量，也没有返回出去，唯一可能发生保存操作的地方只有在调用 join_edges 的地方:

```
join_edges(current, *prevline, free_cracks, outline_it);
```

确实如此， join_edges 的作用是 将 **current** 和 ***prevline** 连接起来 。:

```
edge2->prev->next = edge1->next;
edge1->next->prev = edge2->prev;
edge1->next = edge2;
edge2->prev = edge1;
```

此外值得注意是 join_edges 内部的一个方法调用 `complete_edge` :

```
if (edge1->next = edge2) {
    complete_edge(edge1, outline_it); // already closed
    edge1->prev->next = *free_cracks; // attach freelist to end
    *free_cracks = edge1;             // and free list
}
```

查看 complete_edge 函数的实现(textord/edgloop.cpp)可以发现它生成了一个 bouding_box 然后加入到了 outline_it 中去。

循环第一部分:

```
uppercolour = FLIP_COLOUR(uppercolour);
if (colour == prevcolour) {
    if (colour == uppercolour) {
        // finish a line
        join_edges(current, *prevline, free_cracks, outline_it);
        current = NULL;        // no edge now
```

```
    } else {
        // new horiz edge
        current = h_edge(uppercolour - colour, *prevline, &pos);
    }
    *prevline = NULL;        // no change this time
} else {
    if (colour == uppercolour)
        *prevline = v_edge(colour - prevcolour, *prevline, &pos);
    // 8 vs 4 connection
    else if (colour == WHITE_PIX) {
        join_edges(current, *prevline, free_cracks, outline_it);
        current = h_edge(uppercolour - colour, NULL, &pos);
        *prevline = v_edge(colour - prevcolour, current, &pos);
    } else {
        newcurrent = h_edge(uppercolour - colour, *prevline, &pos);
        *prevline = v_edge(colour - prevcolour, current, &pos);
        current = newcurrent;  // right going h edge
    }
    prevcolour = colour;     // remember new colour
}
```

循环第二部分:

```
if (colour != prevcolour) {
    *prevline = current = v_edge(colour - prevcolour, current, &pos);
    prevcolour = colour;
}
if (colour != uppercolour)
    current = h_edge(uppercolour - colour, current, &pos);
else
    current = NULL;          // no edge now
```

在循环结束后还有一个结束步骤:

```
if (current != NULL) {
    // out of block
    if (*prevline != NULL) {     // got one to join to?
        join_edges(current, *prevline, free_cracks, outline_it);
        *prevline = NULL;          // tidy now
    } else {
        // fake vertical
        *prevline = v_edge(FLIP_COLOUR(prevcolour)-prevcolour, current, &pos);
    }
} else if (*prevline != NULL) {
    //continue fake
    *prevline = v_edge(FLIP_COLOUR(prevcolour)-prevcolour, *prevline, &pos);
}
```

- h_edge
- v_edge
- join_edge

2. outlines_to_blobs (textord/edgblob.cpp)

函数说明:

```
Gather gogether outlines into blobs using the usual bucket sort
```

桶排序?

函数总共就三行:

```
OL_BUCKETS buckets(bleft, tright);
fill_buckets(outlines, &buckets);
empty_buckets(block, &buckets);
```

看上去像是先用 outlines 里的东西把 "桶" 装满了，然后再倒出到 block 里

4. SetupAndFilterNoise(textord/colfind.cpp)

函数说明:

```
Performs initial processing on the blobs in the input_block:
Setup the part_grid, stroke_width_, nontext_map.
Obvious noise blobs are filtered out and used to mark the nontext_map_.
Initial stroke-width analysis is used to get local text alignment
direction, so the textline projection_ map can be setup.
```

```
On return, IsVerticallyAlignedText may be called (now optionally) to
determine the gross textline alignment of the page.
```

该函数内部调用了一个较重要的方法为:

```
FindTextlineDirectionANdFixBrokenCJK
```

5. ColumnFinder::IsVerticallyAlignedText(textord/colfind.cpp)

   调用另外一个函数然后返回

   1. StrokeWidth::TestVerticalTextDirection(textord/colfind.cpp)

      首先调用 CollectHorizVertBlobs 从输入的 blocks (TO_BLOCK) 中的 blobs 和 large_blobs 中收集 blobs，也就是说，underlines、noise_blobs 和 small_blobs 被忽略(参考数据结构: TO_BLOCK)

      1. tesseract::CollectHorizVertBlobs(textord/colfind.cpp)

         循环从作为输入的 input_blobs (BLOBNBOX_LIST *) 中读取 blob，并取 blob 的边界框 (TBOX)，对于宽高比或者高宽比超过 2 的抛弃不用，只对 1-2 之间的进行处理。

         怎么处理的呢?调用 UniquelyVertical 或者 UniquelyHorizontal ，如果哪一者返回为真就把它添加到对应的集合里去 (BLOBNBOX_C_IT)，如果两者都返回 false 则把它添加到 "不水平也不垂直" 的集合里去。

         不过这里的判断—— UniquelyHorizontal 和 UniquelyVertical 都只是用对象内部的成员做逻辑操作之后返回，说明这些早就在之前就已经完成了，这里只是 取出 信息而已。

         经查询，对内部的 horz_possible_ 和 vert_possible_ (即上述函数用来判断垂直还是水平用到的内部成员) 的更改是通过方法 set_horz_possible 和 set_vert_possible 来进行设置的，而调用这两个方法的地方只在 textord/strokewidth.cpp 中进行

         ```
         ./textord/strokewidth.cpp:1123:    blob->set_horz_possible(true);
         ./textord/strokewidth.cpp:1129:      blob->set_horz_possible(false);
         ./textord/strokewidth.cpp:1134:    blob->set_horz_possible(false);
         ./textord/strokewidth.cpp:1213:      blob->set_horz_possible(true);
         ./textord/strokewidth.cpp:1216:      blob->set_horz_possible(false);
         ```

         经查，上述几处调用只发生在两个地方:

         - StrokeWidth::SetNeihbourFlows
         - StoresWidth::SmoothNeighbourTypes

         可以对这两个地方进行阅读以了解如何进行判断的。

2.1.2 ColumnFinder::FindBlocks

在 FindBlocks 开头就调用了三个方法:

- FindImagePartitions
- TransferImagePartsToImageMask
- FindImagePartitions

刚进入函数时，调用了这么一条语句:

```
pixOr(photo_mask_pix, photo_mask_pix, nontext_map_);
```

这里的 photo_mask_pix 都是空白的一张图像，而 nontext_map_ 中有非文字类成分的轮廓在里面。

但没有见到对 nontext_map_ 的局部变量定义，也不是传入的参数，说明它是 ColumnFinder 这个类的成员。从 ColumnFinder 类的定义来看，这个方法比较可疑:

```
// Performs initial processing on the blobs in the input_block:
// Setup the part_grid, stroke_width_, nontext_map_.
// Obvious noise blobs are filtered out and used to mark the nontext_map_.
// Initial stroke-width analysis is used to get local text alignment
// direction, so the textline projection_ map can be setup.
// On return, IsVerticallyAlignedText may be called (now optionally) to
```

```
// determine the gross textline alignment of the page.
void SetupAndFilterNoise(Pix* photo_mask_pix, TO_BLOCK* input_block);
```

注释说明，该方法设置了三个东西:

- part_grid
- stroke_width
- nontext_map_

果然，该方法在 SetupPageSegAndDetectOrientation 中被调用了

1. ImageFind::FindImagePartitions

   在开头调用 ConnCompAndRectangularize 方法，在进入该方法前，传入该函数的地一个参数所表示的图像已经检测出了图像，并把图像以外的内容全部擦除了。

   1. ImageFind::ConnCompAndRectangularize

      在该方法中调用了 Leptonica 的方法 pixConnComp 来获得各个连通分量及其边界框

      然后循环处理这些连通分量:

      ```
      int npixes = pixaGetCount(*pixa);
      for (int i = 0; i < npixes; ++i) {
          // ...
          Pix *img_pix = pixaGetPix(*pixa, i, L_CLONE);
          pixDisplayWrite(img_pix, textord_tabfind_show_images);
          if (pixNearlyRectangular(img_pix, kMinRectangularFraction,
                                   kMaxRectangularFraction,
                                   kMaxRectangularGradient,
                                   &x_start, &y_start, &x_end, &y_end)) {
              // ....
          }

          pixDestroy(&img_pix);
      }
      ```

      1. pixConnComp

         调用 pixConnComPixa

      2. pixConnCompPixa

         Leptonica 的方法，函数注释:

         ```
         /*!
         *  pixConnCompPixa()
         *
         *      Input:  pixs (1 bpp)
         *              &pixa (<return> pixa of each c.c.)
         *              connectivity (4 or 8)
         *      Return: boxa, or null on error
         *
         *  Notes:
         *      (1) This finds bounding boxes of 4- or 8-connected components
         *          in a binary image, and saves images of each c.c
         *          in a pixa array.
         *      (2) It sets up 2 temporary pix, and for each c.c. that is
         *          located in raster order, it erases the c.c. from one pix,
         *          then uses the b.b. to extract the c.c. from the two pix using
         *          an XOR, and finally erases the c.c. from the second pix.
         *      (3) A clone of the returned boxa (where all boxes in the array
         *          are clones) is inserted into the pixa.
         *      (4) If the input is valid, this always returns a boxa and a pixa.
         *          If pixs is empty, the boxa and pixa will be empty.
         */
         ```

         从注释来看，其功能是寻找 "4向或八向连通分量" 的边界框，并将每一个连通分量都保存到一个 pix 中去。

         pixa 中的 pix 保存了这些 pix，然后 boxa 保存每个 pix 在原图中的位置信息

2. TODO ImageFind::TransferImagePartsToImageMask(textord/colfind.cpp)
3. TODO ImageFind::FindImagePartitions(textord/colfind.cpp)

2.2 TODO Textord::TextordPage

2.3 资源文件的读取和解析

tesseract::Tesseract::init_tesseract_internal

-> tesseract::Tesseract::init_tesseract_lang_data

3 数据结构

3.1 DONE Pix

为 Leptonica 中定义的结构，别名为 PIX

```
typedef struct Pix {
    l_uint32 w;
    l_uint32 h;
    l_uint32 d;
    l_uint32 spp;
    l_uint32 wpl;
    l_uint32 refcount;
    l_int32 xres;
    l_int32 yres;
    l_int32 informat;
    l_int32 special;
    char *text;
    struct PixColormap *colormap;
    l_uint32 *data;
}PIX;
```

3.2 TODO Boxa

3.3 TODO Pixa

Leptonica 中的结构，定义为:

```
typedef struct Pixa {
    l_int32 n;
    l_int32 ncalloc;
    l_uint32 refcount;
    Pix **pix;
    Boxa *boxa;
}
```

Pix array，不过 Boxa 的结构还不太清楚

3.4 DONE STRING

定义于 ccutil/strngs.h 中

Tesseract 内部定义的字符串类型，实现了很多字符串的操作方法，其数据全部放在数据成员 **data_** 中，该成员的类型为 STRING_HEADER

STRING_HEADER 是一个结构体:

```
typedef struct STRING_HEADER {
    int capacity_;
    mutable int used_;
} STRING_HEADER;
```

没想明白字符串是怎么存进去的……

但在 STRING 类内部有两个获取字符串内容的方法是:

```
inline char* GetCStr() {
    return ((char *)data_) + sizeof(STRING_HEADER);
}

inline const char* GetCStr() {
    return ((const char *)data_) + sizeof(STRING_HEADER);
}
```

强制类型转换，这也可以?

该类内部还有一个叫做 AlloData 的方法:

```
char* STRING::AllocData(int used, int capacity) {
    data_ = (STRING_HEADER *)alloc_string(capacity + sizeof(STRING_HEADER));

    // header is the metadata for this memory block
    STRING_HEADER* header = GetHeader();
    header->capacity_ = capacity;
    header->used_ = used;
    return GetCStr();
}
```

该方法在一个构造方法中被调用:

```
STRING::STRING(const char* cstr) {
    if (cstr == NULL) {
        // Empty STRINGs contain just the "\0".
        memcpy(AllocData(1, kMinCapacity), "", 1);
    } else {
        int len = strlen(cstr) + 1;
        char* this_cstr = AllocData(len, len);
        memcpy(this_cstr, cstr, len);
    }
    assert(InvariantOk());
}
```

从这里来看，好像确实是这样的

3.5 BITS16

3.6 DONE DIR128

DIR 是 direction 的缩写，定义于 ccstruct/mod128.h 中:

```
class DLLSYM DIR128 {
private:
    inT8 dir;                   // a direction
public:
    // constructors, other functions
}
```

3.7 DONE CRACKEDGE

定义于 ccstruct/crakedge.h 中:

```
class CRACKEDGE {
public:
    CRACKEDGE() {}

    ICOORD pos;                 /*position of crack */
    inT8 stepx;                 //edge step
    inT8 stepy;
    inT8 stepdir;               //chaincode
    CRACKEDGE *prev;            /*previous point */
    CRACKEDGE *next;            /*next point */
};
```

3.8 DONE CrackPos

定义于 textord/scanedg.h 中

```
typedef struct CrackPos {
    CRACKEDGE **free_cracks;
    int x;
    int y;
}CrackPos;
```

3.9 TODO OL_BUCKETS

定义于 textord/edgblob.h 中:

```
class OL_BUCKETS
{
public:
    OL_BUCKETS(               //constructor
        ICOORD bleft,         //corners
        ICOORD tright);

    ~OL_BUCKETS () {             //cleanup
        delete[]buckets;
    }
    C_OUTLINE_LIST *operator () (//array access
        inT16 x,                 //image coords
        inT16 y);
    //first non-empty bucket
    C_OUTLINE_LIST *start_scan() {
        for (index = 0; buckets[index].empty () && index < bxdim * bydim - 1;
             index++);
        return &buckets[index];
    }
    //next non-empty bucket
    C_OUTLINE_LIST *scan_next() {
        for (; buckets[index].empty () && index < bxdim * bydim - 1; index++);
        return &buckets[index];
    }
    inT32 count_children(                    //recursive sum
        C_OUTLINE *outline,  //parent outline
        inT32 max_count);    // max output
    inT32 outline_complexity(                // new version of count_children
        C_OUTLINE *outline,  // parent outline
        inT32 max_count,     // max output
        inT16 depth);        // level of recursion
    void extract_children(                    //single level get
        C_OUTLINE *outline,  //parent outline
        C_OUTLINE_IT *it);   //destination iterator

private:
    C_OUTLINE_LIST * buckets;    //array of buckets
    inT16 bxdim;                 //size of array
    inT16 bydim;
    ICOORD bl;                   //corners
    ICOORD tr;
    inT32 index;                 //for extraction scan
};
```

3.10 DONE ELIST_LINK

定义于 ccutil/elst.h 中

```
class DLLSYM ELIST_LINK
{
    friend class ELIST_ITERATOR;
    friend class ELIST;

    ELIST_LINK *next;

public:
    ELIST_LINK() {
        next = NULL;
    }
    //constructor

    ELIST_LINK(                       //copy constructor
        const ELIST_LINK &) {  //dont copy link
        next = NULL;
    }

    void operator= (             //dont copy links
        const ELIST_LINK &) {
        next = NULL;
    }
};
```

该类是一个较重要的类， Tesseract 中的许多类都继承了 ELIST_LINK .

从类的定义可知， ELIST_LINK 将赋值运算符与拷贝构造函数都显示地关闭(使其无效化)了，那么对 ELIST_LINK 的操作是如何执行的呢? 其节点数据又是存储在何处的呢 ?

3.11 DONE ELIST

定义于 ccutil/elst.h 中，是一个单链表类:

```
class ELIST {
  private:
    ELIST_LINK *last;

    ELIST_LINK * First(void);
  public:
    ELIST(void);
    void internal_clear(void (*)(ELIST_LINK *));
    bool empty(void) const;
    bool singleton(void) const;
    void shallow_copy(ELIST *);
    void internal_deep_copy(ELIST_LINK *(*)(ELIST_LINK *), const ELIST *);
    void assign_to_sublist(ELIST_ITERATOR *, ELIST_ITERATOR *);
    inT32 length(void) const;
    void sort(int (*)(const void *, const void *));
    ELIST_LINK * add_sorted_and_find(int (*)(const void *, const void *), bool, ELIST_LINK *);
    bool add_sorted(int (*)(const void *, const void *), bool, ELIST_LINK *);
}
```

通过 ELIST_ITERATOR 类来进行访问，大致是这样的， ELIST_LINK 是单链表节点，而 ELIST 是包含表头的链表，还封装了一些针对整个链表的操作(复制、属性等)，而 ELIST_ITERATOR 则用来遍历链表或增、删节点。

不过 ELIST_LINK 中没有实现存储有用数据的成员，不知道其数据将会保存在哪里，或者只在继承的子类里根据自己的情况来实现数据存储?

另外有几个找不到实现的方法:

- zapper
- ELIST::internal_deep_copy

上面的猜测是正确的，以 BLOCK_LIST 为例，涉及以下类:

- PDBLK: 包含具体的数据
- BLOCK: 多继承，继承了 ELIST_LINK 和 PDBLK，就成了一个包含具体数据的单链表节点
- BLOCK_LIST: 继承自 ELIST
- BLOCK_IT: 继承自 ELIST_ITERATOR

这样，在具体应用的时候，就是先有一个 BLOCK_IT ，然后通过向其中添加 BLOCK 类对象来扩充 BLOCK_LIST

3.12 DONE ICOORDELT_IT

- State "DONE" from "TODO" [2015-05-11 一 11:57]

继承自 ELIST_LINK, ccstruct/points.h

```
class DLLSYM ICOORDELT:public ELIST_LINK, public ICOORD
    //embedded coord list
{
public:
    ///empty constructor
    ICOORDELT() {
    }
    ///constructor from ICOORD
    ICOORDELT (ICOORD icoord):ICOORD (icoord) {
    }
    ///constructor
    ///@param xin x value
    ///@param yin y value
    ICOORDELT(inT16 xin,
              inT16 yin) {
        xcoord = xin;
        ycoord = yin;
    }

    static ICOORDELT* deep_copy(const ICOORDELT* src) {
        ICOORDELT* elt = new ICOORDELT;
        *elt = *src;
        return elt;
    }

};
```

表示坐标的单链表，就是这样。

3.13 DONE PDBLK

ccstruct/pdblock.h

对该类的注释为:

```
Page block
```

```
class PDBLK
{
    friend class BLOCK_RECT_IT;    //< block iterator

public:
    ///empty constructor
    PDBLK() {
        hand_poly = NULL;
        index_ = 0;
    }
    ///simple constructor
    PDBLK(inT16 xmin,  //< bottom left
          inT16 ymin,
          inT16 xmax,  //< top right
          inT16 ymax);

    ///destructor
    ~PDBLK () {
        if (hand_poly) delete hand_poly;
    }

    // 其他方法

protected:
    POLY_BLOCK *hand_poly;       //< wierd as well
    ICOORDELT_LIST leftside;     //< left side vertices
    ICOORDELT_LIST rightside;    //< right side vertices
    TBOX box;                    //< bounding box
    int index_;                  //< Serial number of this block.
};
```

PDBLK 构造的时候设置了 leftside 和 rightside

```
ICOORDELT_IT left_it = &leftside;
ICOORDELT_IT right_it = &rightside;

hand_poly = NULL;
left_it.set_to_list (&leftside);
right_it.set_to_list (&rightside);
//make default box
left_it.add_to_end (new ICOORDELT (xmin, ymin));
left_it.add_to_end (new ICOORDELT (xmin, ymax));
right_it.add_to_end (new ICOORDELT (xmax, ymin));
right_it.add_to_end (new ICOORDELT (xmax, ymax));
```

可见 leftside 代表的是图像最左列， rightside 代表图像最右列。

3.14 TODO BLOCK_RECT_IT

定义于 ccstruct/pdblock.h 中

```
class DLLSYM BLOCK_RECT_IT       //rectangle iterator
{
public:
    ///@param blkptr block to iterate
    BLOCK_RECT_IT(PDBLK *blkptr);

    ///start (new) block
    void set_to_block (
        PDBLK * blkptr);           //block to iterate

    ///start iteration
    void start_block();

    ///next rectangle
    void forward();

    ///test end
    BOOL8 cycled_rects() {
        return left_it.cycled_list () && right_it.cycled_list ();
    }

    ///current rectangle
    void bounding_box(ICOORD &bleft,
                      ICOORD &tright) {
```

```
        //bottom left
        bleft = ICOORD (left_it.data ()->x (), ymin);
        //top right
        tright = ICOORD (right_it.data ()->x (), ymax);
    }

private:
    inT16 ymin;                 //< bottom of rectangle
    inT16 ymax;                 //< top of rectangle
    PDBLK *block;               //< block to iterate
    ICOORDELT_IT left_it;       //< boundary iterators
    ICOORDELT_IT right_it;
};
```

从函数 forward 的实现来看

3.15 TODO BLOCK_LINE_IT

并非继承自 ELIST_ITERATOR，类型定义于 ccstruct/pdblock.h 中:

```
///rectangle iterator
class DLLSYM BLOCK_LINE_IT
{
public:
    ///constructor
    ///@param blkptr from block
    BLOCK_LINE_IT (PDBLK * blkptr)
        :rect_it (blkptr) {
        block = blkptr;            //remember block
    }

    ///start (new) block
    ///@param blkptr block to start
    void set_to_block (PDBLK * blkptr) {
        block = blkptr;            //remember block
        //set iterator
        rect_it.set_to_block (blkptr);
    }

    ///get a line
    ///@param y line to get
    ///@param xext output extent
    inT16 get_line(inT16 y,
                   inT16 &xext);

private:
    PDBLK * block;              //< block to iterate
    BLOCK_RECT_IT rect_it;      //< rectangle iterator
};
```

这里依赖 BLOCK_RECT_IT

3.16 DONE BLOCK

类定义于 ccstruct/ocrblock.h 中:

```
class BLOCK:public ELIST_LINK, public PDBLK
{
    friend class BLOCK_RECT_IT;    //block iterator

public:
    BLOCK()
        : re_rotation_(1.0f, 0.0f),
          classify_rotation_(1.0f, 0.0f),
          skew_(1.0f, 0.0f) {
        right_to_left_ = false;
        hand_poly = NULL;
    }
    BLOCK(const char *name,  //< filename
          BOOL8 prop,        //< proportional
          inT16 kern,        //< kerning
          inT16 space,       //< spacing
          inT16 xmin,        //< bottom left
          inT16 ymin,
          inT16 xmax,        //< top right
          inT16 ymax);

    ~BLOCK () {
    }

    // 其他方法

private:
```

```
    BOOL8 proportional;          //< proportional
    bool right_to_left_;         //< major script is right to left.
    inT8 kerning;                //< inter blob gap
    inT16 spacing;               //< inter word gap
    inT16 pitch;                 //< pitch of non-props
    inT16 font_class;            //< correct font class
    inT32 xheight;               //< height of chars
    float cell_over_xheight_;    //< Ratio of cell height to xheight.
    STRING filename;             //< name of block
    ROW_LIST rows;               //< rows in block
    PARA_LIST paras_;            //< paragraphs of block
    C_BLOB_LIST c_blobs;         //< before textord
    C_BLOB_LIST rej_blobs;       //< duff stuff
    FCOORD re_rotation_;         //< How to transform coords back to image.
    FCOORD classify_rotation_;   //< Apply this before classifying.
    FCOORD skew_;                //< Direction of true horizontal.
    ICOORD median_size_;         //< Median size of blobs.
};
```

该类继承自 PDBLK , 从其结构上来看，应该是用于表示处理图片时找到的 "块"

3.17 DONE BLOCK_LIST

通过宏定义，继承自 ELIST

```
extern void BLOCK_zapper(ELIST_LINK* link);

class BLOCK_LIST : public ELIST {

    public: BLOCK_LIST() :ELIST() {}

    BLOCK_LIST( const BLOCK_LIST&) {
        DONT_CONSTRUCT_LIST_BY_COPY.error("BLOCK_LIST", ABORT, __null);
    }
    void clear() {
        ELIST::internal_clear(&BLOCK_zapper);
    }

    ~BLOCK_LIST() {
        clear();
    }

    void deep_copy(const BLOCK_LIST* src_list, BLOCK* (*copier)(const BLOCK*));

    void operator=( const BLOCK_LIST&) {
        DONT_ASSIGN_LISTS.error( "BLOCK_LIST", ABORT, __null );
    }
};

class BLOCK_IT : public ELIST_ITERATOR {
    public: BLOCK_IT():ELIST_ITERATOR(){}

    BLOCK_IT( BLOCK_LIST* list):ELIST_ITERATOR(list){}

    BLOCK* data() {
        return (BLOCK*) ELIST_ITERATOR::data();
    }

    BLOCK* data_relative( inT8 offset) {
        return (BLOCK*) ELIST_ITERATOR::data_relative( offset );
    }

    BLOCK* forward() {
        return (BLOCK*) ELIST_ITERATOR::forward();
    }

    BLOCK* extract() {
        return (BLOCK*) ELIST_ITERATOR::extract();
    }

    BLOCK* move_to_first() {
        return (BLOCK*) ELIST_ITERATOR::move_to_first();
    }

    BLOCK* move_to_last() {
        return (BLOCK*) ELIST_ITERATOR::move_to_last();
    }
};
```

该宏名为 ELISTIZE 和 ELISTIZEH，其中 ELISTIZEH 用于在头文件中生成要生成的类的定义，ELISTIZE 用于在 C/CPP 文件中生成对应的类的实现。

ELISTIZEH 宏由三部分组成:

- ELISTIZEH_A

  定义一个删除函数:

```
#define ELISTIZEH_A(CLASSNAME)                                  \
                                                                \
    extern DLLSYM void CLASSNAME##_zapper(ELIST_LINK* link);
```

- ELISTIZEH_B

  定义 LIST 类:

```
#define ELISTIZEH_B(CLASSNAME)                                                  \
                                                                                \
/************************************************************************** \
 *                          CLASS - CLASSNAME##_LIST                        \
 *                                                                          \
 *                          List class for class CLASSNAME                  \
 *                                                                          \
 **************************************************************************/ \
                                                                            \
    class DLLSYM                CLASSNAME##_LIST : public ELIST             \
    {                                                                       \
public:                                                                     \
CLASSNAME##_LIST():ELIST() {}                                               \
/* constructor */                                                           \
                                                                            \
CLASSNAME##_LIST(           /* dont construct */                            \
    const CLASSNAME##_LIST&)                            /*by initial assign*/ \
{ DONT_CONSTRUCT_LIST_BY_COPY.error( QUOTE_IT( CLASSNAME##_LIST ),          \
                                     ABORT, NULL ); }                       \
                                                                            \
void                        clear()                     /* delete elements */ \
{ ELIST::internal_clear( &CLASSNAME##_zapper ); }                           \
                                                                            \
~CLASSNAME##_LIST() /* destructor */                                        \
{ clear(); }                                                                \
                                                                            \
/* Become a deep copy of src_list*/                                         \
void deep_copy(const CLASSNAME##_LIST* src_list,                            \
               CLASSNAME* (*copier)(const CLASSNAME*));                     \
                                                                            \
void                        operator=(                  /* prevent assign */ \
    const CLASSNAME##_LIST&)                                                \
    { DONT_ASSIGN_LISTS.error( QUOTE_IT( CLASSNAME##_LIST ),                \
                               ABORT, NULL ); }
```

- ELISTIZEH_C

  定义一个 IT 类:

```
#define ELISTIZEH_C( CLASSNAME )                                                \
    };                                                                          \
                                                                                \
                                                                                \
                                                                                \
/************************************************************************** \
 *                          CLASS - CLASSNAME##_IT                          \
 *                                                                          \
 *                          Iterator class for class CLASSNAME##_LIST       \
 *                                                                          \
 *  Note: We don't need to coerce pointers to member functions input                \
 *  parameters as these are automatically converted to the type of the base         \
 *  type. ("A ptr to a class may be converted to a pointer to a public base         \
 *  class of that class")                                                                   \
 **************************************************************************/        \
                                                                                                \
      class DLLSYM                CLASSNAME##_IT : public ELIST_ITERATOR                  \
      {                                                                                             \
      public:                                                                                         \
          CLASSNAME##_IT():ELIST_ITERATOR(){}                   \
                                                                                                \
          CLASSNAME##_IT(                                                   \
              CLASSNAME##_LIST*           list):ELIST_ITERATOR(list){}                      \
                                                                                                \
          CLASSNAME*          data()                                                    \
          { return (CLASSNAME*) ELIST_ITERATOR::data(); }                           \
                                                                                                \
          CLASSNAME*          data_relative(                                                \
              inT8                    offset)                                               \
          { return (CLASSNAME*) ELIST_ITERATOR::data_relative( offset ); }        \
                                                                                                \
          CLASSNAME*          forward()                                                 \
```

```
        { return (CLASSNAME*) ELIST_ITERATOR::forward(); }                          \
                                                                                    \
        CLASSNAME*          extract()                                                \
        { return (CLASSNAME*) ELIST_ITERATOR::extract(); }                          \
                                                                                    \
        CLASSNAME*          move_to_first()                                          \
        { return (CLASSNAME*) ELIST_ITERATOR::move_to_first(); }                    \
                                                                                    \
        CLASSNAME*          move_to_last()                                           \
        { return (CLASSNAME*) ELIST_ITERATOR::move_to_last(); }                     \
    };
```

而 ELISTIZE 的定义则为:

```
#define ELISTIZE(CLASSNAME)                                                 \
DLLSYM void CLASSNAME##_zapper(ELIST_LINK* link) {                          \
    delete reinterpret_cast<CLASSNAME*>(link);                                  \
}                                                                               \
\
/* Become a deep copy of src_list*/                                             \
void CLASSNAME##_LIST::deep_copy(const CLASSNAME##_LIST* src_list,              \
                                 CLASSNAME* (*copier)(const CLASSNAME*)) {                  \
    \
    CLASSNAME##_IT from_it(const_cast<CLASSNAME##_LIST*>(src_list));            \
    CLASSNAME##_IT to_it(this);                                                 \
    \
    for (from_it.mark_cycle_pt(); !from_it.cycled_list(); from_it.forward())    \
        to_it.add_after_then_move((*copier)(from_it.data()));                     \
}
#endif
```

ccstruct/ocrblock.h 中调用 ELISTIZEH(BLOCK) 就产生了 BLOCK_LIST 和 BLOCK_IT 两个类，与此同时， BLOCK_IT 类内部需要 BLOCK 类，此时 BLOCK 类需要自行实现

3.18 DONE ICOORD

定义于 ccstruct/points.h 中

```
class ICOORD {
public:
    ICOORD() {
        xcoord = ycoord = 0;
    }
    // other functions

private:
    inT16 xcoord;
    inT16 ycoord;
};
```

同时在 points.h 中还有名为 ICOORDELT 的类继承了 ELIST_LINK 和 ICOORD

与 ICOORD 类似的还有 FCOORD ，大致相同，不同之处在于其 x, y 的类型是 float.

3.19 DONE TBOX

定义于 ccstruct/rect.h 中

```
class DLLSYM TBOX {
public:
    TBOX ():       // empty constructor making a null box
    bot_left (MAX_INT16, MAX_INT16), top_right (-MAX_INT16, -MAX_INT16) {
    }
    TBOX(          // constructor
        const ICOORD pt1,   // one corner
        const ICOORD pt2);  // the other corner

    // other functions
private:
    ICOORD bot_left;
    ICOORD top_right;
};
```

表示一个 bounding box (边界框)

3.20 NEXT EdgeOffset

一个结构体，定义于 ccstruct/countln.h 中，定义为:

```
struct EdgeOffset {
    inT8 offset_numerator;
    uinT8 pixel_diff;
    uinT8 direction;
};
```

对这个结构有大片的注释:

> Simple struct to hold the 3 values needed to compute a more precise edge position and direction.

对这三个字段的说明为:

- offset_numerator:

  > The offset_numerator is the difference between the grey threshold and the mean pixel value.

- pixel_diff

  > pixel diff is the difference between the pixels in the edge.

  边缘由两个点组成，一般来说，这两个点是连续的，而且这两者的差(梯度)最大，这个 pixel_diff 存储的就是这两个点的灰度差。

- direction

  > The direction stores the quantized feature direction for the given step computed from the edge gradient.(Using binary_angle_plus_pi)

  binary_angle_plus_pi 是类 FCOORD 中的一个静态成员。

举例，某一行有五个像素，其灰度值依次为:

```
50, 60, 100, 110, 120
```

记为 p1, p2, p3, p4, p5，假设二值化阈值为 70,那么

- pixel_diff: 100 - 60 = 40
- offset_numerator: (70 - (100 + 60) / 2) - (100 - 60) = -50 (?)
- direction: ?

3.21 TODO C_OUTLINE

- State "TODO" from "DONE" [2015-05-11 一 14:45]
- State "DONE" from "TODO" [2015-05-11 一 10:42]

定义于 ccstruct/countln.h 中，继承自 ELIST_LINK

```
class DLLSYM C_OUTLINE:public ELIST_LINK {
public:
    C_OUTLINE() {  //empty constructor
        steps = NULL;
        offsets = NULL;
    }
    // other constructor

    // other functions

private:
    // other functions

    TBOX box;                    // bounding box
    ICOORD start;                // start coord
    inT16 stepcount;             // no of steps
    BITS16 flags;                // flags about outline
    uinT8 *steps;                // step array
    EdgeOffset* offsets;         // Higher precision edge.
    C_OUTLINE_LIST children;     // child elements
    static ICOORD step_coords[4];
};
```

依赖以下类:

- [x] ICOORD
- [x] TBOX
- [x]

  EdgeOffset (struct)

  ```
  struct EdgeOffset{
      inT8 offset_numerator;
      uinT8 pixel_diff;
      uinT8 direction;
  };
  ```

- [x]

  BITS16

  对基础数据类型的封装

- [x] CRACKEDGE
- [x] DIR128

3.22 TODO C_BLOB

定义在 ccstruct/stepblob.h 中，继承自 ELIST_LINK

```
class C_BLOB:public ELIST_LINK
{
public:
    C_BLOB() {
    }
    explicit C_BLOB(C_OUTLINE_LIST *outline_list);

    // Simpler constructor to build a blob from a single outline that has
    // already been fully initialized.
    explicit C_BLOB(C_OUTLINE* outline);

    // Builds a set of one or more blobs from a list of outlines.
    // Input: one outline on outline_list contains all the others, but the
    // nesting and order are undefined.
    // If good_blob is true, the blob is added to good_blobs_it, unless
    // an illegal (generation-skipping) parent-child relationship is found.
    // If so, the parent blob goes to bad_blobs_it, and the immediate children
    // are promoted to the top level, recursively being sent to good_blobs_it.
    // If good_blob is false, all created blobs will go to the bad_blobs_it.
    // Output: outline_list is empty. One or more blobs are added to
    // good_blobs_it and/or bad_blobs_it.
    static void ConstructBlobsFromOutlines(bool good_blob,
                                           C_OUTLINE_LIST* outline_list,
                                           C_BLOB_IT* good_blobs_it,
                                           C_BLOB_IT* bad_blobs_it);

    // Sets the COUT_INVERSE flag appropriately on the outlines and their
    // children recursively, reversing the outlines if needed so that
    // everything has an anticlockwise top-level.
    void CheckInverseFlagAndDirection();

    TBOX bounding_box();  //compute bounding box

    // other functions

private:
    C_OUTLINE_LIST outlines;     //master elements
};
```

此外，该类内部有几个名为 **render** 的方法，看起注释似乎是说能将其数据转换为 **Pix** 类型，对调试什么的应该有帮助

依赖以下类:

- C_OUTLINE_LIST
- C_OUTLINE

看 C_OUTLINE 这个类就好了

3.23 TODO BLOBNBOX

定义于 ccstruct/blobbox.h 中，继承自 ELIST_LINK

```
class BLOBNBOX: public ELIST_LINK {
public:
     BLOBNBOX() {
          ConstructionInit();
     }
     // other functions

private:
    C_BLOB *cblob_ptr;
    TBOX box;
    TBOX read_box;
    BLOBNBOX *base_char_blob_;
    BLOBNBOX *neighbours_[BND_COUNT];
    // other members
};
```

依赖的其他类有:

- C_BLOB
- TBOX

BND_COUNT 是在 blobbox.h 中定义的枚举常量:

```
// enum for elements of arrays that refer to neighbours.
// NOTE: keep in this order, so ^2 can be used to flip direction.
enum BlobNeighbourDir {
    BND_LEFT,
    BND_BELOW,
    BND_RIGHT,
    BND_ABOVE,
    BND_COUNT
};
```

其值为 4 ，这里就是定义了一个长度为 4 的数组，用来保存四向相邻的 BLOB

3.24 TODO TO_BLOCK

单纯继承自 ELIST_LINK，定义于 ccstruct/blobbox.h 中:

```
class TO_BLOCK:public ELIST_LINK
{
public:
    TO_BLOCK() : pitch_decision(PITCH_DUNNO) {
        clear();
    }                            //empty
    TO_BLOCK(                    //constructor
        BLOCK *src_block);  //real block
    ~TO_BLOCK();

    void clear();  // clear all scalar members.

    TO_ROW_LIST *get_rows() {  //access function
        return &row_list;
    }


    void print_rows() {  //debug info
        TO_ROW_IT row_it = &row_list;
        TO_ROW *row;

        for (row_it.mark_cycle_pt(); !row_it.cycled_list();
             row_it.forward()) {
            row = row_it.data();
            tprintf("Row range (%g,%g), para_c=%g, blobcount=" INT32FORMAT
                    "\n", row->min_y(), row->max_y(), row->parallel_c(),
                    row->blob_list()->length());
        }
    }

    // other functions

    BLOBNBOX_LIST blobs;         //medium size
    BLOBNBOX_LIST underlines;    //underline blobs
    BLOBNBOX_LIST noise_blobs;   //very small
    BLOBNBOX_LIST small_blobs;   //fairly small
    BLOBNBOX_LIST large_blobs;   //big blobs
    BLOCK *block;                //real block
    PITCH_TYPE pitch_decision;   //how strong is decision
    float line_spacing;          //estimate
    // other members

private:
```

```
    TO_ROW_LIST row_list;        //temporary rows
};
```

可以看到，该类依赖好几个类，分别是：

- BLOBNBOX_LIST(查看 BLOBNBOX 即可)
- BLOCK
- TO_ROW_LIST(查看 TO_ROW 即可)

BLOCK 应该是最外层的一个边界框，而 blobs 啊、 underlines啊、noise_blobs、small_blobs 这些应该是内部的子框。大概是这个样子吧。

3.25 TODO BLOBNBOX_C_IT