STEP 3

bmp图像学习

参考：

参考1

参考2

BMP文件总体上由4部分组成，分别是位图文件头、位图信息头、调色板和图像数据

- 位图文件头（bitmap-file header）
- 位图信息头（bitmap-information header）
- 彩色表/调色板（color table）**[只有8位（biBitCount=8）的图片有]**
- 位图数据（bitmap-data）

位图文件头

```
typedef  struct tagBITMAPFILEHEADER
{
    WORD            bfType;//位图文件类型，必须是0x4D42，即字符串“BM”，也就是说所有.bmp文件的头两个字节都是“BM”
    DWORD         bfSize;//位图文件大小，包括这14个字节
    WORD            bfReserved1;//保留字，设为0
    WORD            bfReserved2;//保留字，设为0
    DWORD         bfOffBits;//从文件头到实际的位图数据的偏移字节数，单位：字节
} BITMAPFILEHEADER;
```

位图信息头

```
typedef struct tagBITMAPINFOHEADER
{
    DWORD         biSize;//本结构所占用字节数，大小为40字节
    LONG             biWidth;//位图宽度，单位：字节
    LONG             biHeight;//位图高度，单位：字节
    WORD            biPlanes;//目标设备级别，必须为1
```

```
    WORD            biBitCount;//表示颜色时每个像素要用到的位数，常用的值为1(黑白二色图), 4(16色图), 8(256色), 24(真彩色图)
    DWORD        biCompression;// 位图是否压缩，其类型是 0(BI_RGB不压缩)， 1(BI_RLE8压缩类型)或2(BI_RLE4压缩类型)
    DWORD        biSizeImage;//实际的位图数据占用的字节数
    LONG            biXPelsPerMeter;//位图水平分辨率，每米像素数
    LONG            biYPelsPerMeter;//位图垂直分辨率，每米像素数
    DWORD        biClrUsed;//指定本图象实际用到的颜色数，如果该值为零，则用到的颜色数为2的biBitCount次幂个
    DWORD        biClrImportant;//指定本图象中重要的颜色数，如果该值为零，则认为所有的颜色都是重要的
} BITMAPINFOHEADER;
```

彩色表/调色板

```
typedef struct tagRGBQUAD
{
    BYTE    rgbBlue; //该颜色的蓝色分量(值范围为0-255)
    BYTE    rgbGreen; //该颜色的绿色分量(值范围为0-255)
    BYTE    rgbRed; //该颜色的红色分量(值范围为0-255)
    BYTE    rgbReserved; //保留值，设为0
} RGBQUAD;
```

位图数据

位图数据记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右，扫描行之间是从下到上。位图的一个像素值所占的字节数：

当biBitCount=1时，8个像素占1个字节；

当biBitCount=4时，2个像素占1个字节；

当biBitCount=8时，1个像素占1个字节；

当biBitCount=24时，1个像素占3个字节,按顺序分别为B,G,R；

Windows规定一个扫描行所占的字节数必须是

4的倍数（即以long为单位），不足的以0填充，

biSizeImage= ((((bi.biWidth * bi.biBitCount) + 31) & ~31) / 8) * bi.biHeight;

> 或者：
>
> lineBytes = (p1->width * p1->bitCount / 8 + 3) / 4 * 4;
>
> biSizeImage = lineBytes * p1->height;

省略文件头重复数据

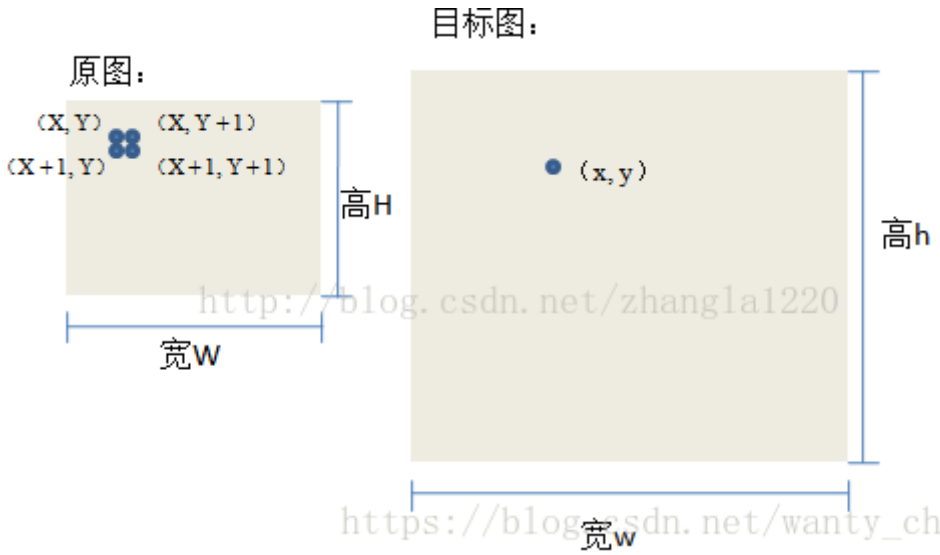
- 只保留图像文件的文件大小、图像宽度、图像高度和图像数据大小信息，甚至有时不需要保留文件大小这个数值，使用图像数据大小数值即可。

- ```
  unsigned char * pData;//读入图像数据的指针
  unsigned int width;//图像的宽
  unsigned int height;//图像的高
  unsigned short bitCount;//图像类型，每像素位数
  unsigned long len;//PixelLen
  RGBQUAD * pRGBQUAD;//颜色表指针
  ```

补零

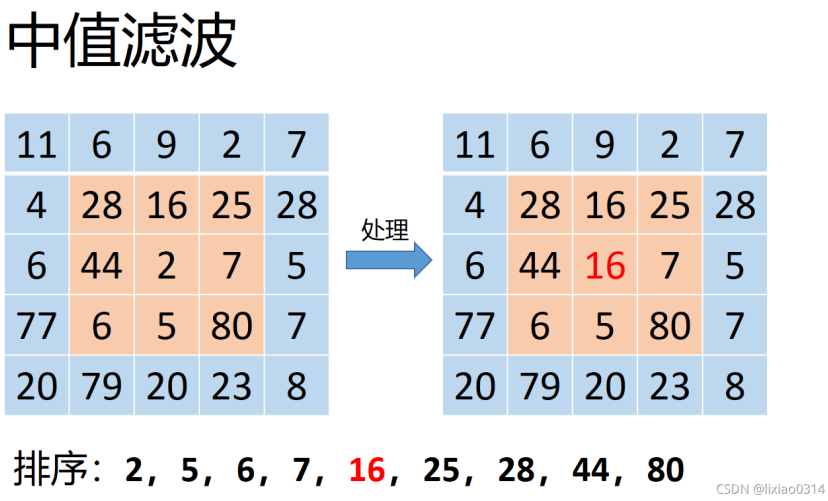
- Windows有“补零”的习惯！即要求位图的每一行像素所占字节数必须被4整除。若不能倍4整除，则在该位图每一行的十六进制码末尾“补”1至3个字节的“00”。
- “补零”只针对位图的宽进行检验

缩放：双线性插值原理



目标图(x, y) 映射到原图是(X + u, Y + v)（计算方法同最邻近插值）。设u与v分别为X + u，Y + v的小数部分。由于下标都是整数，因此原图其实并不存在该点。则取其附近四个领域点为(X, Y) (X, Y + 1) (X + 1, Y) (X + 1, Y + 1)，则目标图(x, y)处的值为 f(x , y) = f(X + u, Y + v) =f (X, Y) * (1 - u) * (1 - v) + f(X, Y + 1) * (1 - u) * v + f(X + 1, Y) * u * (1 - v) + f (X + 1, Y + 1) * u * v;

中值滤波原理



中值滤波是把数字图像或数字序列中一点的值用该点的一个邻域中各点值的中值代替，让周围的像素值接近真实值，从而消除孤立的噪声点。
  ```

代码

bmp.h:

```
#include <iostream>
#include <cstring>
#include<inttypes.h>
using namespace std;

typedef uint16_t WORD;
typedef uint32_t DWORD;
typedef int32_t LONG;
typedef uint8_t BYTE;

#pragma pack (1)
typedef  struct tagBITMAPFILEHEADER
{
    WORD            bfType;//位图文件类型，必须是0x4D42，即字符串“BM”，也就是说所有.bmp文件的头两个字节都是“BM”
    DWORD         bfSize;//位图文件大小，包括这14个字节
    WORD            bfReserved1;//保留字，设为0
    WORD            bfReserved2;//保留字，设为0
    DWORD         bfOffBits;//从文件头到实际的位图数据的偏移字节数，单位：字节
} BITMAPFILEHEADER;
#pragma pack (1)
typedef struct tagBITMAPINFOHEADER
{
    DWORD         biSize;//本结构所占用字节数，大小为40字节
    LONG             biWidth;//位图宽度，单位：字节
    LONG             biHeight;//位图高度，单位：字节
    WORD            biPlanes;//目标设备级别，必须为1
    WORD            biBitCount;//表示颜色时每个像素要用到的位数，常用的值为1(黑白二色图)，4(16色图)，8(256色)，24(真彩色图)
    DWORD         biCompression;// 位图是否压缩，其类型是 0(BI_RGB不压缩)， 1(BI_RLE8压缩类型)或2(BI_RLE4压缩类型)
    DWORD         biSizeImage;//实际的位图数据占用的字节数
    LONG             biXPelsPerMeter;//位图水平分辨率，每米像素数
    LONG             biYPelsPerMeter;//位图垂直分辨率，每米像素数
    DWORD         biClrUsed;//指定本图象实际用到的颜色数，如果该值为零，则用到的颜色数为2的biBitCount次幂个
    DWORD         biClrImportant;//指定本图象中重要的颜色数，如果该值为零，则认为所有的颜色都是重要的
} BITMAPINFOHEADER;
#pragma pack (1)
typedef struct tagRGBQUAD
{
    BYTE    rgbBlue; //该颜色的蓝色分量(值范围为0-255)
    BYTE    rgbGreen; //该颜色的绿色分量(值范围为0-255)
    BYTE    rgbRed; //该颜色的红色分量(值范围为0-255)
    BYTE    rgbReserved; //保留值，设为0
} RGBQUAD;

class PictureData
{
```

```
public:
    unsigned char * pData;//读入图像数据的指针
    unsigned int width;//图像的宽
    unsigned int height;//图像的高
    unsigned short bitCount;//图像类型，每像素位数
    unsigned long len;//PixelLen
    RGBQUAD * pRGBQUAD;//颜色表指针
};

#define iFilterW 3
#define iFilterH 3

void ErrorPrint(int ret);

bool ReadBmpFile(const char * filename, struct PictureData *p1) ;

bool WriteBmpFile(const char * filename, struct PictureData *p1);

bool CropBmpFile(const char * toReadfilename, const char * toCropfilename);

bool ZoomBmpFile(const char * toReadfilename, const char * toZoomfilename);

bool GrayBmpFile(const char * toReadfilename, const char * toGrayfilename);

unsigned char GetMedianNum(int * bArray, int iFilterLen);

bool MedianFilterBmpFile(const char * toMedianFilterfilename, const char *
MedianFilterfilename);
```

bmp.cpp:

```
#include"bmp.h"

void ErrorPrint(int ret)
{
    if (!ret)
    {
        cout << "failed!" << endl;

    }
    else
    {
        cout << "succeed!" << endl;
    }
}

bool ReadBmpFile(const char * filename, struct PictureData *p1)
{
    FILE * pf;
    pf = fopen(filename, "rb");
    if (NULL == pf)
    {
        cout << "ReadBmpFile - open file failed!" << endl;
        fclose(pf);
```

```
        return false;
    }

    BITMAPFILEHEADER bitMapFileHeader;
    BITMAPINFOHEADER bitMapInfoHeader;
    fread(&bitMapFileHeader, sizeof(BITMAPFILEHEADER), 1, pf);
    if (0x4D42 != bitMapFileHeader.bfType)
    {
        cout << "ReadBmpFile - it's not a bmp file!" << endl;
        return false;
    }
    cout << "ReadBmpFile - bitMapFileHeader:" << endl;
    cout << "bitMapFileHeader.bfType: " << bitMapFileHeader.bfType << endl;
    cout << "bitMapFileHeader.bfSize: " << bitMapFileHeader.bfSize << endl;
    cout << "bitMapFileHeader.bfOffBits: " << bitMapFileHeader.bfOffBits <<
endl;

    fread(&bitMapInfoHeader, sizeof(BITMAPINFOHEADER), 1, pf);
    //判断位深
    if(bitMapInfoHeader.biBitCount != 8 && bitMapInfoHeader.biBitCount != 24 &&
bitMapInfoHeader.biBitCount !=32)
    {
        cout << "ReadBmpFile - Format Error %d Bit! Only Supports 8-Bit, 24-Bit
and 32-Bit!" << endl;
        return false;
    }
    //判断尺寸
    if(bitMapInfoHeader.biWidth == 0 || bitMapInfoHeader.biHeight == 0)
    {
        cout << "Size Error! Width is zero or height is zero!" << endl;
        return false;
    }
    cout << "\nReadBmpFile - bitMapInfoHeader:" << endl;
    cout << "bitMapInfoHeader.biSize:" << bitMapInfoHeader.biSize << endl;
    cout << "bitMapInfoHeader.biWidth: " << bitMapInfoHeader.biWidth << endl;
    cout << "bitMapInfoHeader.biHeight: " << bitMapInfoHeader.biHeight << endl;
    cout << "bitMapInfoHeader.biCompression: " << bitMapInfoHeader.biCompression
<< endl;
    cout << "bitMapInfoHeader.biBitCount: " << bitMapInfoHeader.biBitCount <<
endl;
    cout << "bitMapInfoHeader.biSizeImage: " << bitMapInfoHeader.biSizeImage <<
endl;

    p1->width = bitMapInfoHeader.biWidth;
    p1->height = bitMapInfoHeader.biHeight;
    p1->bitCount = bitMapInfoHeader.biBitCount;

    if (8 == bitMapInfoHeader.biBitCount)
    {
        p1->pRGBQUAD = new RGBQUAD[256];
        fread(p1->pRGBQUAD, sizeof(RGBQUAD), 256, pf);
    }

    //数据每行字节数为4的倍数
```

```
    unsigned int lineBytes = (bitMapInfoHeader.biWidth *
bitMapInfoHeader.biBitCount / 8 + 3) / 4 * 4;//步幅是从一行像素的开头到下一行的开头所需
的字节数。
    p1->len=bitMapInfoHeader.biHeight * lineBytes;//总共需要的字节数
    p1->pData = new unsigned char[p1->len];
    fread(p1->pData, sizeof(unsigned char), p1->len, pf);

    fclose(pf);
    return true;
}

bool WriteBmpFile(const char * filename, struct PictureData *p1)
{
    FILE * pf;
    pf = fopen(filename, "wb");
    if (NULL == pf)
    {
        cout << "WriteBmpFile - open file failed!" << endl;
        fclose(pf);
        return false;
    }

    int colorTablesize = 0;
    if (p1->bitCount == 8)
    {
        colorTablesize = 1024;
    }

    //待存储图像数据每行字节数为4的倍数
    int lineBytes = (p1->width * p1->bitCount / 8 + 3) / 4 * 4;

    //申请位图文件头结构变量，填写文件头信息
    BITMAPFILEHEADER bitMapFileHeader;
    bitMapFileHeader.bfType = 0x4D42;//bmp类型
    bitMapFileHeader.bfSize = sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER) + colorTablesize + lineBytes * p1->height;
    bitMapFileHeader.bfReserved1 = 0;
    bitMapFileHeader.bfReserved2 = 0;
    bitMapFileHeader.bfOffBits = 54 + colorTablesize;

    //申请位图信息头结构变量，填写信息头信息
    BITMAPINFOHEADER bitMapInfoHeader;
    bitMapInfoHeader.biBitCount = p1->bitCount;
    bitMapInfoHeader.biClrImportant = 0;
    bitMapInfoHeader.biClrUsed = 0;
    bitMapInfoHeader.biCompression = 0;
    bitMapInfoHeader.biHeight = p1->height;
    bitMapInfoHeader.biPlanes = 1;
    bitMapInfoHeader.biSize = 40;
    bitMapInfoHeader.biSizeImage = lineBytes * p1->height;
    bitMapInfoHeader.biWidth = p1->width;
    bitMapInfoHeader.biXPelsPerMeter = 0;
    bitMapInfoHeader.biYPelsPerMeter = 0;

    //写文件头进文件
```

```
    fwrite(&bitMapFileHeader, sizeof(BITMAPFILEHEADER), 1, pf);
    //写位图信息头进内存
    fwrite(&bitMapInfoHeader, sizeof(BITMAPINFOHEADER), 1, pf);

    //如果灰度图像，有颜色表，写入文件
    if (p1->bitCount == 8)
    {
        fwrite(p1->pRGBQUAD, sizeof(RGBQUAD), 256, pf);
    }
    cout << "WriteBmpFile - bitMapFileHeader: " << endl;
    cout << "bitMapFileHeader.bfType: " << bitMapFileHeader.bfType << endl;
    cout << "bitMapFileHeader.bfSize: " << bitMapFileHeader.bfSize << endl;
    cout << "bitMapFileHeader.bfOffBits: " << bitMapFileHeader.bfOffBits <<
endl;
    cout << "\nWriteBmpFile - bitMapInfoHeader:" << endl;
    cout << "bitMapInfoHeader.biSize:" << bitMapInfoHeader.biSize << endl;
    cout << "bitMapInfoHeader.biWidth: " << bitMapInfoHeader.biWidth << endl;
    cout << "bitMapInfoHeader.biHeight: " << bitMapInfoHeader.biHeight << endl;
    cout << "bitMapInfoHeader.biCompression: " << bitMapInfoHeader.biCompression
<< endl;
    cout << "bitMapInfoHeader.biBitCount: " << bitMapInfoHeader.biBitCount <<
endl;
    cout << "bitMapInfoHeader.biSizeImage: " << bitMapInfoHeader.biSizeImage <<
endl;


    fwrite(p1->pData, sizeof(unsigned char), p1->height * lineBytes, pf);
    fclose(pf);
    return true;
}

bool CropBmpFile(const char * toReadfilename, const char * toCropfilename)
{
    struct PictureData *pCrop = new struct PictureData;
    pCrop->pRGBQUAD = NULL;
    pCrop->pData = NULL;
    bool ret = ReadBmpFile(toReadfilename, pCrop);
    cout << "CropBmpFile - Read bmp file ";
    ErrorPrint(ret);
    cout << endl;
    if(!ret)
    {
        return false;
    }

    unsigned int OffsetWidth, OffsetHeight, CropWidth, CropHeight;
    cout << "CropBmpFile - Enter OffsetWidth, OffsetHeight, CropWidth,
CropHeight: " << endl;
    cin >> OffsetWidth >> OffsetHeight >> CropWidth >> CropHeight;

    if (OffsetWidth + CropWidth > pCrop->width || OffsetHeight + CropHeight >
pCrop->height || OffsetWidth < 0 || OffsetHeight < 0 || CropWidth <= 0 ||
CropHeight <= 0)
    {
        cout << "CropBmpFile - Crop Size Error!" << endl;
```

```
        return false;
    }

    unsigned int depthBytes = pCrop->bitCount / 8;//一字节八位， 每像素位数/8=每像素字节
    unsigned int lineBytes = ((pCrop->width * depthBytes + 3) / 4) * 4;//步幅是从一行像素的开头到下一行的开头所需的字节数。

    unsigned int dstLineBytes = ((CropWidth * depthBytes + 3) / 4) * 4;//步幅是从一行像素的开头到下一行的开头所需的字节数。
    unsigned long dstPixelLen = dstLineBytes * CropHeight;//总共需要的字节数
    unsigned char *CropData = new unsigned char[dstPixelLen];

    for (unsigned int i = 0; i < CropHeight; i++) {
        memcpy(CropData + i * dstLineBytes, pCrop->pData + (i + pCrop->height - CropHeight - OffsetHeight) * lineBytes + OffsetWidth * depthBytes, dstLineBytes);
    }

    pCrop->pData = CropData;
    pCrop->len = dstPixelLen;
    pCrop->height = CropHeight;
    pCrop->width = CropWidth;

    ret = WriteBmpFile(toCropfilename, pCrop);
    cout << "CropBmpFile - Write bmp file " ;
    ErrorPrint(ret);
    if(!ret)
    {
        return false;
    }

    delete[] pCrop->pRGBQUAD;
    pCrop->pRGBQUAD = NULL;
    delete[] pCrop->pData;
    pCrop->pData = NULL;
    delete pCrop;

    return true;
}

bool ZoomBmpFile(const char * toReadfilename, const char * toZoomfilename)
{
    //读取
    struct PictureData *pZoom = new struct PictureData;
    pZoom->pRGBQUAD = NULL;
    pZoom->pData = NULL;
    bool ret = ReadBmpFile(toReadfilename, pZoom);
    cout << "ZoomBmpFile - Read bmp file ";
    ErrorPrint(ret);
    cout << endl;
    if(!ret)
    {
        return false;
    }
```

```
//输入缩放因子
double factorX;
double factorY;
cout << "ZoomBmpFile - Enter zoom factorX, factorY: " <<endl;
cin >> factorX >> factorY;

//判断缩放因子
if (factorX <= 0 || factorY <= 0)
{
    cout << "ZoomBmpFile - Factor Error! Cannot be negative!" << endl;
    return false;
}

double factorReciprocalX = 1 / factorX;
double factorReciprocalY = 1 / factorY;
unsigned int dstWidth = (unsigned int)(pZoom->width * factorX);
unsigned int dstHeight = (unsigned int)(pZoom->height * factorY);

unsigned int depthBytes = pZoom->bitCount / 8;
unsigned int dstLineBytes = ((dstWidth * depthBytes + 3) / 4) * 4;
unsigned char *dstBytes = new unsigned char[dstLineBytes * dstHeight];
unsigned int lineBytes = ((pZoom->width * depthBytes + 3) / 4) * 4;

for (unsigned int h = 0; h < dstHeight; h++)
{
    for (unsigned int w = 0; w < dstWidth; w++)
    {
        //原图像的真实位置
        double srcRealX = (w + 0.5) * factorReciprocalX - 0.5;
        double srcRealY = (h + 0.5) * factorReciprocalY - 0.5;
        /*double srcRealX = w * factorReciprocalX;
        double srcRealY = h * factorReciprocalY;*/

        //原图像对应的像素点位置
        int srcX = (int)srcRealX;
        int srcY = (int)srcRealY;

        //原图像位置偏移量
        double offsetX = srcRealX - srcX;
        double offsetY = srcRealY - srcY;

        //原图像位置的临近4个点
        int leftUp = srcY * lineBytes + srcX * depthBytes;
        int rightUp = srcY * lineBytes + (srcX + 1) * depthBytes;
        int leftDown = (srcY + 1) * lineBytes + srcX * depthBytes;
        int rightDown = (srcY + 1) * lineBytes + (srcX + 1) * depthBytes;

        if (srcY + 1 == dstHeight - 1)
        {
            leftDown = leftUp;
            rightDown = rightUp;
        }
        if (srcX + 1 == dstWidth - 1)
        {
```

```
                rightUp = leftUp;
                rightDown = leftDown;
            }

            //目的图像像素位置索引
            int index = h * dstLineBytes + w * depthBytes;
            for (unsigned int i = 0; i < depthBytes; i++)
            {
                double part1 = pZoom->pData[leftUp + i] * (1 - offsetX) * (1 -
offsetY);
                double part2 = pZoom->pData[rightUp + i] * offsetX * (1 -
offsetY);
                double part3 = pZoom->pData[leftDown + i] * offsetY * (1 -
offsetX);
                double part4 = pZoom->pData[rightDown + i] * offsetY * offsetX;

                dstBytes[index + i] = (unsigned char)(part1 + part2 + part3 +
part4);
            }
        }
    }

    pZoom->pData = dstBytes;
    pZoom->len = dstLineBytes * dstHeight;
    pZoom->width = dstWidth;
    pZoom->height = dstHeight;

    ret = WriteBmpFile(toZoomfilename, pZoom);
    cout << "ZoomBmpFile - Write bmp file " ;
    ErrorPrint(ret);
    if(!ret)
    {
        return false;
    }

    delete[] pZoom->pRGBQUAD;
    pZoom->pRGBQUAD = NULL;
    delete[] pZoom->pData;
    pZoom->pData = NULL;
    delete pZoom;
    return true;
}

bool GrayBmpFile(const char * toReadfilename, const char * toGrayfilename)
{
    struct PictureData *pGray = new struct PictureData;
    pGray->pRGBQUAD = NULL;
    pGray->pData = NULL;
    bool ret = ReadBmpFile(toReadfilename, pGray);
    cout << "GrayBmpFile - Read bmp file ";
    ErrorPrint(ret);
    cout << endl;
    if(!ret)
    {
        return false;
```

```
    }

    if(24 != pGray->bitCount)
    {
        cout << "GrayBmpFile - The file is't a RGB bmp file!" << endl;
        return false;
    }

    unsigned char *pGrayData;
    //因为24为真彩色位图数据的一个像素用3各字节表示，灰度图像为1个字节
    pGray->bitCount = 8;
    //一个扫描行所占的字节数必须是4的倍数(即以long为单位),不足的以0填充
    //所以如果当前pGray->width如果不是4的倍数时，要在后面补0直到为4的倍数
    int lineBytes = (pGray->width * pGray->bitCount / 8 + 3) / 4 * 4;
    int oldLineBytes = (pGray->width * 24 / 8 + 3) / 4 * 4;
    pGrayData = new unsigned char[lineBytes * pGray->height];

    //定义灰度图像的颜色表
    pGray->pRGBQUAD = new RGBQUAD[256];
    for (int i = 0; i < 256; i++)
    {
        (*(pGray->pRGBQUAD + i)).rgbBlue = i;
        (*(pGray->pRGBQUAD + i)).rgbGreen = i;
        (*(pGray->pRGBQUAD + i)).rgbRed = i;
        (*(pGray->pRGBQUAD + i)).rgbReserved = 0;
    }
    //将RGB转换为灰度值
    int red, green, blue;
    BYTE gray;
    //char gray_1;

    for (int i = 0; i < pGray->height; i++)
    {
        //位图数据(pGray->pData)中存储的实际像素数为pGray->width个,而一个扫描行要
lineByte个字节,
        //多余出来的是上面补的0,所以要转换的要是实际的像素数,
        //因为转换前后pGray->width是相同的，而lineByte是不同的，也就是后面补的0不同
        for (int j = 0; j < pGray->width; j++)
        {
            red = *(pGray->pData + i*oldLineBytes + 3 * j + 2);
            green = *(pGray->pData + i*oldLineBytes + 3 * j + 1);
            blue = *(pGray->pData + i*oldLineBytes + 3 * j );
            //实际应用时，希望避免低速的浮点运算，所以需要整数算法。
            gray = (BYTE)((77 * red + 151 * green + 28 * blue) >> 8);
            //gray_1 = red*0.299 + green*0.587 + blue*0.114;
            *(pGrayData + i*lineBytes + j) = gray;
        }
    }
    pGray->pData = pGrayData;

    ret = WriteBmpFile(toGrayfilename, pGray);
    cout << "GrayBmpFile - Write bmp file " ;
    ErrorPrint(ret);
    if(!ret)
    {
```

```
        return false;
    }

    delete[] pGray->pRGBQUAD;
    pGray->pRGBQUAD = NULL;
    delete[] pGray->pData;
    pGray->pData = NULL;
    delete pGray;

    return true;
}

unsigned char GetMedianNum(int * bArray, int iFilterLen)
{
    int i, j;// 循环变量
    unsigned char bTemp;

    // 用冒泡法对数组进行排序
    for (j = 0; j < iFilterLen - 1; j++)
    {
        for (i = 0; i < iFilterLen - j - 1; i++)
        {
            if (bArray[i] > bArray[i + 1])
            {
                // 互换
                bTemp = bArray[i];
                bArray[i] = bArray[i + 1];
                bArray[i + 1] = bTemp;
            }
        }
    }

    // 计算中值
    if ((iFilterLen & 1) > 0)//&按位与
    {
        // 数组有奇数个元素，返回中间一个元素
        bTemp = bArray[(iFilterLen + 1) / 2];
    }
    else
    {
        // 数组有偶数个元素，返回中间两个元素平均值
        bTemp = (bArray[iFilterLen / 2] + bArray[iFilterLen / 2 + 1]) / 2;
    }

    return bTemp;
}

bool MedianFilterBmpFile(const char * toMedianFilterfilename, const char *
MedianFilterfilename)
{

    struct PictureData *pFilter = new struct PictureData;
    pFilter->pRGBQUAD = NULL;
    pFilter->pData = NULL;
    bool ret = ReadBmpFile(toMedianFilterfilename, pFilter);
```

```
    cout << "MedianFilterBmpFile - Read bmp file ";
    ErrorPrint(ret);
    cout << endl;
    if(!ret)
    {
        return false;
    }

    if(24 != pGray->bitCount)
    {
        cout << "MedianFilterBmpFile - The file is't a RGB bmp file!" << endl;
        return false;
    }

    unsigned char *FilterData;
    unsigned char *oldData;
    oldData = pFilter->pData;
    int  aValue[iFilterH*iFilterW];          // 指向滤波器数组的指针

    int lineBytes = (pFilter->width * pFilter->bitCount / 8 + 3) / 4 * 4;
    FilterData = new unsigned char[lineBytes * pFilter->height];
    int iFilterHM= (iFilterH - 1)/ 2;
    int iFilterWM = (iFilterW - 1) / 2;

 //中值滤波
    for (int i = iFilterHM; i < pFilter->height - iFilterHM; i++)
        for (int j = iFilterWM; j < pFilter->width - iFilterWM; j++)
            for (int k = 0; k < 3; k++)
            {

                for (int m  = 0; m< iFilterH; m++)
                    for (int n = 0; n < iFilterW; n++)
                        {
                            aValue[m * iFilterW + n] = *(oldData + lineBytes*
(i+m-1) + (j + n-1) * 3 + k);
                        }
                *(FilterData + pFilter->width * 3 * i + j * 3 + k) =
GetMedianNum(aValue, iFilterH * iFilterW);

            }
 //边缘
    for (int i = 0; i < pFilter->height; i++)
        for (int j = 0; j < pFilter->width; j++)
                for (int k = 0; k < 3; k++)
                {


                    if ((i<iFilterHM) && (j<iFilterWM))
                                *(FilterData + lineBytes * i + j * 3 + k) = *
(FilterData + lineBytes* (i + iFilterHM) + (j + iFilterWM) * 3 + k);
                    else if ((i<iFilterHM) && (j>=iFilterWM)&&(j<(pFilter->width
- iFilterWM)))
                        *(FilterData + lineBytes * i + j * 3 + k) = *(FilterData
+ lineBytes* (i + iFilterHM) + j  * 3 + k);
```

```
                    else if ((i<iFilterHM) && ( j >= (pFilter->width -
iFilterWM)))
                        *(FilterData + lineBytes * i + j * 3 + k) = *(FilterData
+ lineBytes* (i + iFilterHM) + (j - iFilterWM) * 3 + k);
                    else if ((i>=iFilterHM) && i <(pFilter->height - iFilterHM)
&& (j< iFilterWM))
                        *(FilterData + lineBytes * i + j * 3 + k) = *(FilterData
+ lineBytes* i + (j + iFilterWM) * 3 + k);
                    else if ((i>=iFilterHM) && i < (pFilter->height - iFilterHM)
&& (j >= (pFilter->width - iFilterWM)))
                        *(FilterData + lineBytes * i + j * 3 + k) = *(FilterData
+ lineBytes* i + (j - iFilterWM) * 3 + k);
                    else if ((i >= (pFilter->height - iFilterHM)) && (j<
iFilterWM))
                        *(FilterData + lineBytes * i + j * 3 + k) = *(FilterData
+ lineBytes* (i - iFilterHM) + (j + iFilterWM) * 3 + k);
                    else if ((i >= (pFilter->height - iFilterHM)) && (j>=
iFilterWM)&&(j<(pFilter->width - iFilterWM)))
                                *(FilterData + lineBytes * i + j * 3 + k) = *
(FilterData + lineBytes* (i - iFilterHM) + j * 3 + k);
                    else if ((i >= (pFilter->height - iFilterHM)) && ( j >=
(pFilter->width - iFilterWM)))
                        *(FilterData + lineBytes * i + j * 3 + k) = *(FilterData
+ lineBytes* (i - iFilterHM) + (j - iFilterWM) * 3 + k);


                }
    pFilter->pData = FilterData;


    ret = WriteBmpFile(MedianFilterfilename, pFilter);
    cout << "MedianFilterBmpFile - Write bmp file " ;
    ErrorPrint(ret);
    if(!ret)
    {
        return false;
    }

    delete[] pFilter->pRGBQUAD;
    pFilter->pRGBQUAD = NULL;
    delete[] pFilter->pData;
    pFilter->pData = NULL;
    delete pFilter;

    return true;
}
```

main.cpp:

```
#include"bmp.h"

int main()
{
    //判断
```

```
    bool ret;
    //进行操作的图像
    const char * toReadfilename = "bmppicture/photo1.bmp";
    const char * Cropfilename = "bmppicture/crop.bmp";
    const char * Zoomfilename = "bmppicture/zoom.bmp";
    const char * Grayfilename = "bmppicture/gray8bit.bmp";
    const char * toMedianFilterfilename =
"bmppicture/SaltandPepperNoiseBmp.bmp";
    const char * MedianFilterfilename = "bmppicture/medianfilter.bmp";

    int choice;
    cout << "Enter your choice: (1 : Crop, 2 : Zoom, 3 : Gray , 4 : Median
Filter)" << endl;
    cin >> choice;
    switch(choice)
    {
    case 1:
        ret = CropBmpFile(toReadfilename, Cropfilename);
        cout << "Crop ";
        ErrorPrint(ret);
        break;
    case 2:
        ret = ZoomBmpFile(toReadfilename, Zoomfilename);
        cout << "Zoom ";
        ErrorPrint(ret);
        break;
    case 3:
        ret = GrayBmpFile(toReadfilename, Grayfilename);
        cout << "Gray ";
        ErrorPrint(ret);
        break;
    case 4:
        ret = MedianFilterBmpFile(toMedianFilterfilename,MedianFilterfilename);
        cout << "Median Filter ";
        ErrorPrint(ret);
        break;
    default:
        cout << "Did anything." << endl;
        break;
    }

    return 0;
}
```

Makefile

参考：Makefile Tutorial By Example

参考2

如何编写一个Makefile文件

```
# 自定义依赖关系，源文件（后缀为.cpp)经过编译汇编生成目标文件（后缀为.o)
# 目标文件执行生成可执行文件（类似与bmpout）
#-c 只允许执行到汇编步骤，不允许链接。
```

```
# 写gcc命令时候，前面要tab按键一下
# 不写-o参数，生成默认的可执行文件名为a.out，这里我们修改为bmpout

# OBJS   代替  依赖文件
# CC     代替  g++
# CFLAGS 代替  编译命令
# $^ 代替 上面的指令
# RM 代替 rm -f
# $@ 代替 目标文件

OBJS=main.o bmp.o
CC=g++
CFLAGS=-c

bmpout:$(OBJS)
    $(CC) $(OBJS) -o $@
main.o : main.cpp
    $(CC) $^ $(CFLAGS) -o $@
bmp.o : bmp.h bmp.cpp
    $(CC) bmp.cpp $(CFLAGS) -o $@
clean:
    $(RM) *.o bmpout -r
```

一些知识点：

- **Makefile** 必须使用制表符而不是空格进行缩进，否则`make`将失败。

辅助视频（讲的很清晰）

- 目标文件不存在or依赖文件更新的时候进行操作，存在显示`is up to date.`
- ```
  targets: prerequisites
      command
      command
      command
  ```
- ```
  some_file:
      touch some_file

  clean:
      rm -f some_file
  ```

  `clean`：
  - 它不是第一个目标 (默认目标)，也不是先决条件。 这意味着除非显式调用`make lean`，否则它永远不会运行。
  - 它不是一个文件名。 如果你碰巧有一个名为 “clean” 的文件，则此目标将无法运行。
- 变量只能是字符串。 你通常会希望使用`:=`(允许你追加到一个变量)，但`=`也可以。`?=` 仅设置尚未设置的变量
- 使用`${}`或`$()`引用变量
- `$@`是包含目标名称的 自动变量 。当一个规则有多个目标时，将为每个目标运行命令。
- `*`会在你的文件系统中搜索匹配的文件名。 建议始终将其包装在`wildcard`函数中。
- “*” 可以在目标，先决条件或`wildcard`函数中使用，但不能在变量定义中直接使用。

- 当 `*` 不匹配任何文件时，它将维持原样 (除非在`wildcard` 函数中运行)
- `$?`比目标新的所有先决条件（依赖）。
- `$^`所有先决条件（依赖）。
- 隐式规则使用的重要变量包括：
  - `CC`: 编译C程序的程序；默认`cc`
  - `CXX`: 编译C++程序的程序; 默认 “g++”
  - `CFLAGS`: 要提供给C编译器的额外标志
  - `CXXFLAGS`: 给C++编译器的额外标志
  - `CPPFLAGS`: 要提供给C预处理器的额外标志
  - `LDFLAGS`: 当编译器应该调用链接器时，会给他们额外的标志

```
CC = gcc # 隐式规则标志
CFLAGS = -g # 隐式规则的标志。打开调试信息

# 隐式规则 #1：blah是通过C链接器隐式规则构建的
# 隐式规则 #2: blah.o是通过C编译隐式规则构建的，因为blah.c存在
blah: blah.o

blah.c:
    echo "int main() { return 0; }" > blah.c

clean:
    rm -f blah*
```

- **静态模式规则**

```
objects = foo.o bar.o all.o
all: $(objects)

# 这些文件通过上面隐式规则进行编译
foo.o: foo.c
bar.o: bar.c
all.o: all.c

all.c:
    echo "int main() { return 0; }" > all.c

%.c:
    touch $@

clean:
    rm -f *.c *.o all
```

使用静态模式规则的更高效的方式:

```
objects = foo.o bar.o all.o
all: $(objects)

# 这些文件通过隐式规则进行编译
# 语法 - targets ...: target-pattern: prereq-patterns ...
# 在第一个目标foo.o的情况下，目标模式与foo.o匹配，并将“词干”设置为“foo”。
```

```
# 然后用该词干替换prereq模式中的“%”
$(objects): %.o: %.c

all.c:
    echo "int main() { return 0; }" > all.c

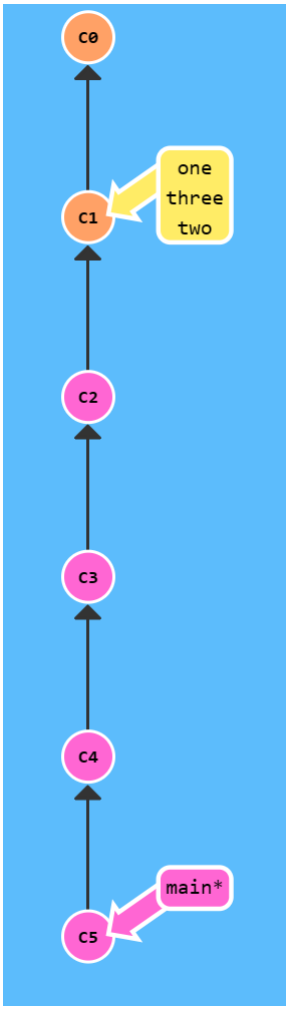
%.c:
    touch $@

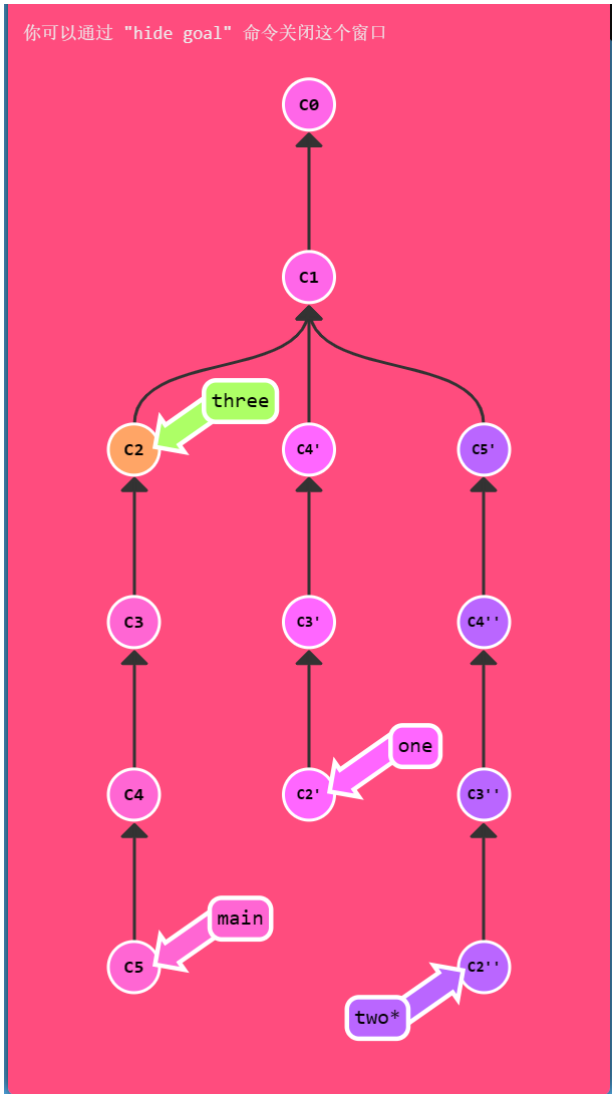
clean:
    rm -f *.c *.o all
```

- 静态模式规则和过滤器

  `filter` 过滤器函数可以在静态模式规则中使用，以匹配正确的文件。

```
obj_files = foo.result bar.o lose.o
src_files = foo.raw bar.c lose.c

all: $(obj_files)

$(filter %.o,$(obj_files)): %.o: %.c
    echo "target: $@ prereq: $<"
$(filter %.result,$(obj_files)): %.result: %.raw
    echo "target: $@ prereq: $<"

%.c %.raw:
    touch $@

clean:
    rm -f $(src_files)
```

- 在命令之前添加 `@` 以阻止其打印
  还可以运行带有`-s`的make命令，等同于在每行前面添加一个`@`
- 在运行时添加 `-k`，即使面对错误也要继续运行。 如果你想一次查看Make的所有错误，这将非常有用。
- 在命令前添加 `-` 以抑制错误
- 添加`-i`以使每个命令都会发生这种情况。

git

> [Learn Git Branching]
>
> 网站里面不太方便看，记录一下

- Git 仓库中的提交记录保存的是目录下所有文件的快照，像是把整个目录复制，然后再粘贴一样。
- Git 希望提交记录尽可能地轻量，因此在你每次进行提交时，它并不会盲目地复制整个目录。条件允许的情况下，它会将当前版本与仓库中的上一个版本进行对比，并把所有的差异打包到一起作为一个提交记录。
- Git 保存了提交的历史记录。这也是为什么大多数提交记录的上面都有父节点的原因 —— 我们会在图示中用箭头来表示这种关系。对于项目组的成员来说，维护提交历史对大家都有好处。
- `git commit`创建一个新的提交记录。

- Git 的分支也非常轻量。它们只是简单地指向某个提交纪录 。建议：

  ```
  早建分支！多用分支！
  ```

  这是因为即使创建再多的分支也不会造成储存或内存上的开销，并且按逻辑分解工作到不同的分支要比维护那些特别臃肿的分支简单多了。

  先记住使用分支其实就相当于在说：“我想基于这个提交以及它所有的父提交进行新的工作。”

- `git checkout <name>`切换分支

分支与合并

- `git merge <name>`

  新建一个分支，在其上开发某个新功能，开发完成后再合并回主线。

  在 Git 中合并两个分支时会产生一个特殊的提交记录，它有两个父节点。

- `git rebase <name>`

  Rebase 实际上就是取出一系列的提交记录，“复制”它们，然后在另外一个地方逐个的放下去。

  Rebase 的优势就是可以创造更线性的提交历史。

在提交树上移动

- HEAD是一个对当前所在分支的符号引用 —— 也就是指向你正在其基础上进行工作的提交记录。

  HEAD 总是指向当前分支上最近一次提交记录。大多数修改提交树的 Git 命令都是从改变 HEAD 的指向开始的。

  HEAD 通常情况下是指向分支名的（如 bugFix）。在提交时，改变了 bugFix 的状态，这一变化通过 HEAD 变得可见。

  如果想看 HEAD 指向，可以通过 `cat .git/HEAD` 查看， 如果 HEAD 指向的是一个引用，还可以用 `git symbolic-ref HEAD` 查看它的指向。

- **相对引用**
  - 通过指定提交记录哈希值的方式在 Git 中移动不太方便。在实际应用时，并没有像本程序中这么漂亮的可视化提交树供你参考，所以你就不得不用 `git log` 来查查看提交记录的哈希值。
  - 并且哈希值在真实的 Git 世界中也会更长（译者注：基于 SHA-1，共 40 位）。例如前一关的介绍中的提交记录的哈希值可能是 `fed2da64c0efc5293610bdd892f82a58e8cbc5d8`。
  - 比较令人欣慰的是，Git 对哈希的处理很智能。你只需要提供能够唯一标识提交记录的前几个字符即可。因此我可以仅输入 `fed2` 而不是上面的一长串字符。
  - 使用相对引用的话，你就可以从一个易于记忆的地方（比如 `bugFix` 分支或 `HEAD`）开始计算。
    - 使用 `^` 向上移动 1 个提交记录
    - 使用 `~<num>` 向上移动多个提交记录，如 `~3`
  - 强制修改分支位置：直接使用 `-f` 选项让分支指向另一个提交。
    - `git branch -f main HEAD~3` 将 main 分支强制指向 HEAD 的第 3 级父提交。

撤销变更

- 在 Git 里撤销变更的方法很多。和提交一样，撤销变更由底层部分（暂存区的独立文件或者片段）和上层部分（变更到底是通过哪种方式被撤销的）组成。主要关注后者。

  主要有两种方法用来撤销变更：`git reset` 和 `git revert`。

- `git reset` 通过把分支记录回退几个提交记录来实现撤销改动。（在reset后， 所做的变更还在，但是处于未加入暂存区状态。）

  `git reset HEAD~1`(head指向要撤销的)

- 在本地分支中使用 `git reset` 很方便，但是这种“改写历史”的方法对大家一起使用的远程分支是无效的。

  为了撤销更改并分享给别人，我们需要使用 `git revert`。在要撤销的提交记录后面多了一个新提交。这是因为新提交记录 `C2'` 引入了**更改** —— 这些更改刚好是用来撤销 `C2` 这个提交的。也就是说 `C2'` 的状态与 `C1` 是相同的。revert 之后就可以把更改推送到远程仓库与别人分享了。

  `git revert HEAD`(head指向要撤销的).

整理提交记录

- `git cherry-pick <提交号>...` 将一些提交复制到当前所在的位置（`HEAD`）。
- 当知道所需要的提交记录（**并且**还知道这些提交记录的哈希值）时, 用 cherry-pick 最简单。
- 当不清楚想要的提交记录的哈希值（想从一系列的提交记录中找到想要的记录）时，可以利用交互式的 rebase 。
- 交互式 rebase 指的是使用带参数 `--interactive` 的 rebase 命令, 简写为 `-i`。

  在命令后增加了这个选项, Git 会打开一个 UI 界面并列出将要被复制到目标分支的备选提交记录，它还会显示每个提交记录的哈希值和提交说明，提交说明有助于理解这个提交进行了哪些更改。

  在实际使用时，所谓的 UI 窗口一般会在文本编辑器 —— 如 Vim —— 中打开一个文件。

  当 rebase UI界面打开时, 能做3件事:

  - 调整提交记录的顺序（通过鼠标拖放来完成）
  - 删除你不想要的提交（通过切换 `pick` 的状态来完成，关闭就意味着不想要这个提交记录）
  - 合并提交。

  Git 会严格按照在对话框中指定的方式进行复制。

  > head->C5,需要对C2~C5进行操作
  >
  > ...C2 <- C3 <- C4 <- C5
  >
  > `git rebase HEAD~4`

提交的技巧1

- 想对某个以前的提交记录进行一些小小的调整，尽管那个提交记录并不是最新的了。
- 通过下面的方法：
  - 先用 `git rebase -i` 将提交重新排序，然后把想要修改的提交记录挪到最前
  - 然后用 `git commit --amend` 来进行一些小修改
  - 接着再用 `git rebase -i` 来将他们调回原来的顺序
  - 最后我们把 main 移到修改的最前端（用你自己喜欢的方法），就大功告成啦!

提交的技巧2

- cherry-pick 可以将提交树上任何地方的提交记录取过来追加到 HEAD 上（只要不是 HEAD 上游的提交就没问题）。

- ```
  #c0 <- c1 <-c2 <- C3
  git checkout main
  git cherry-pick C2
  git commit --amend
  git cherry-pick C3
  ```

Git Tags

- 分支很容易被人为移动，并且当有新的提交时，它也会移动。分支很容易被改变，大部分分支还只是临时的，并且还一直在变。
- git的tag可以（在某种程度上 —— 因为标签可以被删除后重新在另外一个位置创建同名的标签）永久地将某个特定的提交命名为里程碑，然后就可以像分支一样引用了。
- 它们并**不会随着新的提交而移动**。你也不能切换到某个标签上面进行修改提交，它就像是提交树上的一个锚点，标识了某个特定的位置。
- `git tag <标签名> <提交记录name>`，如果不指定提交记录，Git 会用 `HEAD` 所指向的位置。

Git Describe

- 由于标签在代码库中起着"锚点"的作用，Git 还为此专门设计了一个命令用来**描述**最近的锚点（也就是标签），它就是 `git describe`
- Git Describe 能帮你在提交历史中移动了多次以后找到方向；在用 `git bisect`（一个查找产生 Bug 的提交记录的指令）找到某个提交记录时， 可能会用到这个命令。
- `git describe` 的语法是：

  ```
  git describe <ref>
  ```

  `<ref>` 可以是任何能被 Git 识别成提交记录的引用，如果你没有指定的话，Git 会使用你目前所在的位置（`HEAD`）。

  它输出的结果是这样的：

  ```
  <tag>_<numCommits>_g<hash>
  ```

  `tag` 表示的是离 `ref` 最近的标签， `numCommits` 是表示这个 `ref` 与 `tag` 相差有多少个提交记录， `hash` 表示的是你所给定的 `ref` 所表示的提交记录哈希值的前几位。

  当 `ref` 提交记录上有某个标签时，则只输出标签名称

多分支 rebase

- ```
  git rebase master bugFix
  git rebase bugFix side
  git rebase side another
  git branch -f master another
  ```

- `git rebase <目标位置分支名> <要移动的分支名>`
- 会把该分支的父提交一起复制移动合并。

选择父提交记录

- 操作符 `^` 与 `~` 符一样，后面也可以跟一个数字。

- 但是该操作符后面的数字与 `~` 后面的不同，并不是用来指定向上返回几代，而是**指定合并提交记录的某个父提交**。还记得前面提到过的一个合并提交有两个父提交吧，所以遇到这样的节点时该选择哪条路径就不是很清晰了。
- Git 默认选择合并提交的“第一个”父提交，在操作符 `^` 后跟一个数字可以改变这一默认行为。
- ```
  git checkout HEAD~ #移动HEAD到其的上一个
  git checkout HEAD^2
  git checkout HEAD~2
  ```

  相当于`git checkout HEAD~^2~2`

纠缠不清的分支

操作前：



操作：

```
git checkout one
git cherry-pick C4 C3 C2
git checkout two
git cherry-pick C5 C4 C3 C2
git branch -f three C2
```

操作后：
```



Git Clone

- 远程仓库并不复杂, 在如今的云计算盛行的世界很容易把远程仓库想象成一个富有魔力的东西, 但实际上它们只是你的仓库在另个一台计算机上的拷贝。你可以通过因特网与这台计算机通信 —— 也就是增加或是获取提交记录
- 话虽如此, 远程仓库却有一系列强大的特性:
    - 首先也是最重要的的点, 远程仓库是一个强大的备份。本地仓库也有恢复文件到指定版本的能力, 但所有的信息都是保存在本地的。有了远程仓库以后，即使丢失了本地所有数据, 你仍可以通过远程仓库拿回你丢失的数据。
    - 还有就是, 远程让代码社交化了! 既然你的项目被托管到别的地方了, 你的朋友可以更容易地为你的项目做贡献(或者拉取最新的变更)
- 现在用网站来对远程仓库进行可视化操作变得越发流行了(像 GitHub), 但远程仓库**永远**是这些工具的顶梁柱, 因此理解其概念非常的重要!

远程上传代码至github

参考:

从 0 开始学习 Git + GitHub

git使用步骤——断开与原来远程仓库的连接，连接新仓库并上传

将文件移至新位置 - GitHub 文档

问题解决

```
ssh-keygen -t rsa -C #生成ssh key，把 id_rsa.pub 的内容添加到 GitHub 上
ssh -T git@github.com #测试连接是否成功
#进入项目目录
git init
git branch -m mywork
git add .
git commit -m "提交注释"
git remote add origin git@github.com:YQQ23-3-9/Learning-tasks-22.git
git pull --rebase origin mywork
git push -u origin mywork

#期间使用git status检查状态
#git remote -v查看远程连接
```