1. 库文件找不到

```

gcc Test.c -o Test `pkg-config --cflags --libs opencv`

将文件/etc/ld.so.conf中添加一行代码/usr/local/lib

当库文件找不到时, 使用 sudo ldconfig -v

```

2. 显示不出图片

```

安装libgtk2.0-dev， 然后重新编译

```

3. 配置ffmpeg

```

sudo CFLAGS="-fPIC" ./configure --enable-gpl --enable-nonfree --enable-pthreads --enable-libfaac --enable-libmp3lame --enable-libtheora --enable-libx264 --enable-libxvid --enable-x11grab --enable-libopencore-amrnb --enable-libopencore-amrwb --enable-libopencore-amrnb --enable-version3 --disable-optimizations --disable-asm

```

4. 查看opencv版本

```

pkg-config --modversion opencv

```

5. 完全删除OpenCV

```

sudo find  / "*opencv*" -exec rm -rf {}/;

```

6. 未解决

```

opencv rectangle ambiguous

```

######附 fPIC详解

>----

####ppc_85xx-gcc -shared -fPIC liberr.c -o liberr.so

>-fPIC 作用于编译阶段，告诉编译器产生与位置无关代码(Position-Independent Code)，

则产生的代码中，没有绝对地址，全部使用相对地址，故而代码可以被加载器加载到内存的任意位置，都可以正确的执行。

这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的。

```

gcc -shared -fPIC -o 1.so 1.c
```

这里有一个-fPIC参数

PIC就是position independent code

PIC使.so文件的代码段变为真正意义上的共享

```

>如果不加-fPIC,则加载.so文件的代码段时,代码段引用的数据对象需要重定位, 重定位会修改代码段的内容,这就造成每个使用这个.so文件代码段的进程在内核里都会生成这个.so文件代码段的copy.每个copy都不一样,取决于 这个.so文件代码段和数据段内存映射的位置.

>**不加fPIC编译出来的so,是要再加载时根据加载到的位置再次重定位的.(因为它里面的代码并不是位置无关代码)**
如果被多个应用程序共同使用,那么它们必须每个程序维护一份so的代码副本了.(因为so被每个程序加载的位置都不同,显然这些重定位后的代码也不同,当然不能共享)
我们总是用fPIC来生成so,也从来不用fPIC来生成a.
fPIC与动态链接可以说基本没有关系,libc.so一样可以不用fPIC编译,只是这样的so必须要在加载到用户程序的地址空间时重定向所有表目.
- - -
因此,不用fPIC编译so并不总是不好.
如果你满足以下4个需求/条件:
1.该库可能需要经常更新
2.该库需要非常高的效率(尤其是有很多全局量的使用时)
3.该库并不很大.

4.该库基本不需要被多个应用程序共享

- - -
>如果用没有加这个参数的编译后的共享库，也可以使用的话，可能是两个原因：
1：gcc默认开启-fPIC选项
2：loader使你的代码位置无关
从GCC来看，shared应该是包含fPIC选项的，但似乎不是所以系统都支持，所以最好显式加上fPIC选项。参见如下

```
`-shared'
    Produce a shared object which can then be linked with other
    objects to form an executable. Not all systems support this
    option. For predictable results, you must also specify the same
    set of options that were used to generate code (`-fpic', `-fPIC',
    or model suboptions) when you specify this option.(1)
```
>-fPIC 的使用，会生成 PIC 代码，.so 要求为 PIC，以达到动态链接的目的，否则，无法实现动态链接。
**non-PIC 与 PIC 代码的区别主要在于 access global data, jump label 的不同。**
比如一条 access global data 的指令，
non-PIC 的形势是：ld r3, var1
PIC 的形式则是：ld r3, var1-offset@GOT,意思是从 GOT 表的 index 为 var1-offset 的地方处
指示的地址处装载一个值,即var1-offset@GOT**处的4个 byte 其实就是 var1 的地址**。**这个地址只有在运行的时候才知道，是由 dynamic-loader(ld-linux.so) 填进去的。

>再比如 jump label 指令
non-PIC 的形势是：jump printf ，意思是调用 printf。
PIC 的形式则是：jump printf-offset@GOT,
**意思是跳到 GOT 表的 index 为 printf-offset 的地方处指示的地址去执行，**
**这个地址处的代码摆放在 .plt section，**
**每个外部函数对应一段这样的代码，其功能是呼叫dynamic-loader(ld-linux.so) 来查找函数的地址(本例中是 printf)，然后将其地址写到 GOT 表的 index 为 printf-offset 的地方，**
同时执行这个函数。这样，第2次呼叫 printf 的时候，就会直接跳到 printf 的地址，而不必再查找了。
**GOT 是 data section, 是一个 table, 除专用的几个 entry，每个 entry 的内容可以再执行的时候修改；**
**PLT 是 text section, 是一段一段的 code，执行中不需要修改。**
每个 target 实现 PIC 的机制不同，但大同小异。比如 MIPS 没有 .plt, 而是叫 .stub，功能和 .plt 一样。
可见，动态链接执行很复杂，比静态链接执行时间长;但是，极大的节省了 size，PIC 和动态链接技术是计算机发展史上非常重要的一个里程碑。

```

gcc manul上面有说
-fpic If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that -fpic does not work; in that case, recompile with -fPIC instead. (These maximums are 8k on the SPARC and 32k on the m68k and RS/6000. The 386 has no such limit.)

-fPIC If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k, PowerPC and SPARC. Position-independent code requires special support, and therefore works only on certain machines.
```

>关键在于GOT全局偏移量表里面的跳转项大小。
intel处理器应该是统一4字节，没有问题。
powerpc上由于汇编码或者机器码的特殊要求，所以跳转项分为短、长两种。

-fpic为了节约内存，在GOT里面预留了“短”长度。

而-fPIC则采用了更大的跳转项。
```