4.9

1、指针变量相加减运算

指针变量可以和整形数据相加减，此时地址加减多少和指针类型有关；比如，如果在64位的操作系统上,p是指向整型数据的指针变量，那么p=p+2之后，p的存储的地址值变成0x2000 0008，也就说实际上地址值加了sizeof(int)\*2这么多；

指针之间可以相减，但是不可以相加，相减表示两个指针指向的内存之间相差多少个元素，比如，对于int类型的指针p和p1.p1-p的意义表示他们之间相隔多少个int类型的元素；

指针之间可以比较大小，表示在内存中他们指向位置的高低关系；

2、指针类型强制转换

Int a = 5; int \*p = &a; char \*cp = (char \*)&a; pm指向一个整型，cp指向整型数的第一个字节

经典例子：

1. **int** main(**void**)
2. {
3. **int** a[4] = {1, 2, 3, 4};
4. **int** \*ptr1=(**int** \*)(&a+1);
5. **int** \*ptr2=(**int** \*)((**int**)a+1);
6. **int** \*c = \*(a + 1);
7. printf("%x, %x,%x\n", ptr1[-1], \*ptr2,\*c);
8. **return** 0;
9. }
10. **int** n1 = 100;
11. **int**\* pn = &n1;
12. **float**\* p = NULL;
13. p = (**float**\*)pn;
14. cout << pn << endl;   0095FC9C
15. cout << \*pn << endl;   100
16. cout << \*p << endl;   1.4013e-043
17. 1）int指针和float指针的大小都是4字节
18. 2）pn指向n1，地址是0095FC9C
19. 3) \*pn是将0095FC9C地址中的内容按照int类型进行变量解释
20. 4）p = （float\*) pn;强制转换。即将0095FC9C地址中的内容按照float型进行变量解释。

经典题目

unsigned int nValue = 0x5D6C7B3E;

printf("%c", \*((unsigned char \*)(&nValue) + 2) + 2); n

printf("%X", \*((unsigned char \*)(&nValue) + 2) + 2); 0x6E

printf("%d", \*((unsigned char \*)(&nValue) + 2) + 2); 110

3、

1. **size\_t** fwrite(**const** **void**\* buffer, **size\_t** size, **size\_t** count, **FILE**\* stream);
2. -- buffer:指向数据块的指针
3. -- size:每个数据的大小，单位为Byte(例如：**sizeof**(**int**)就是4)
4. -- count:数据个数
5. -- stream:文件指针
6. **FILE** \*fp = fopen("F:\\Labwindows cvi\\test.txt","w");
7. fwrite(dataPtr,**sizeof**(**int**),DATA\_SIZE,fp); 写入DATA\_SIZE个4字节的int
8. fwrite(dataPtr,1,DATA\_SIZE\***sizeof**(unsigned **int**),fp);  和上面一样，写入DATA\_SIZE\***sizeof**(unsigned **int**)个字节

调用格式一fwrite(buf,sizeof(buf),1,fp); 成功返回1

调用格式二fwrite (buf,1,zeof(buf),fp); 成功返回实际写入的个数

写完数据后要调用fclose()关闭流，不关闭流的情况下，每次读或写数据后，文件指针都会指向下一个待写或者读数据位置的指针。

fread()函数：  
1. 作用：从一个文件流中读取数据  
2. 函数原型如下：

**[cpp]** [view plain](https://blog.csdn.net/yang2011079080010/article/details/52528261) [copy](https://blog.csdn.net/yang2011079080010/article/details/52528261)

1. **size\_t** fread(**void** \*buffer, **size\_t** size, **size\_t** count, **FILE** \*stream);
2. -- buffer:指向数据块的指针
3. -- size:每个数据的大小，单位为Byte(例如：**sizeof**(**int**)就是4)
4. -- count:数据个数
5. -- stream:文件指针

注意：返回值随着调用格式的不同而不同：  
(1) 调用格式：fread(buf,sizeof(buf),1,fp);  
读取成功时：当读取的数据量正好是sizeof(buf)个Byte时，返回值为1(即count)  
                       否则返回值为0(读取数据量小于sizeof(buf))  
(2)调用格式：fread(buf,1,sizeof(buf),fp);  
读取成功返回值为实际读回的数据个数(单位为Byte)

4、c++读写文件类

fstream file1("c:\\config.sys");

ifstream file2("c:\\pdos.def");//以输入方式打开文件

ofstream file3("c:\\x.123");//以输出方式打开文件

ofstream out("out.txt");

if(out.is\_open())

{

out<<"HELLO WORLD!"<<endl;

out.close();

}

`

结果在文件out.txt中写入了HELLO WORLD!

`

ifstream in("out.txt");

cha buffer[200];

if(in.is\_open())

{

while(!in.eof())

{

in.getline(buffer,100)

cout<<buffer<<endl;

out.close();

}

}

结果在屏幕输出 HELLO WORLD!

读数据案例：

//读取方式: 逐词读取, 词之间用空格区分

void ReadDataFromFileWBW()  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    ifstream fin("data.txt");    
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    string s;    
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    while( fin >> s )   
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {      
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        cout << "Read from file: " << s << endl;    
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

//读取方式: 逐行读取, 将行读入字符数组, 行之间用回车换行区分

void ReadDataFromFileLBLIntoString()  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    ifstream fin("data.txt");    
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    string s;    
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    while( getline(fin,s) )  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {      
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        cout << "Read from file: " << s << endl;   
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

第二种方法：

void ReadDataFromFileLBLIntoCharArray()  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    ifstream fin("data.txt");   
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    const int LINE\_LENGTH = 100;   
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    char str[LINE\_LENGTH];    
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    while( fin.getline(str,LINE\_LENGTH) )  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {      
https://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        cout << "Read from file: " << str << endl;  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
https://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

5、多线程基础

#include <pthread.h>

int pthread\_create(pthread\_t \*restrict tidp,

const pthread\_attr\_t \*restrict attr,

void \*(\*start\_rtn)(void \*),

void \*restrict arg);

成功返回0，否则返回错误码

参数说明：

* tidp：新创建的线程ID会被设置成tidp指向的内存单元。
* attr：用于定制各种不能的线程属性，默认为NULL
* start\_rtn：新创建的线程从start\_rtn函数的地址开始运行，该函数只有一个void类型的指针参数即arg，如果start\_rtn需要多个参数，可以将参数放入一个结构中，然后将结构的地址作为arg传入。

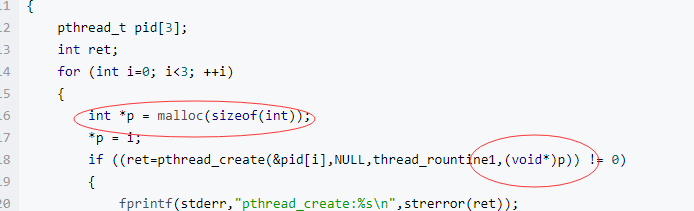
线程注意问题：



上面打印结果为：全部2-hello

原因：新创建的thread1、thread2还没取到i的值，主线程已经改变i的取值，所以，千万记得，**避免直接在传递的参数中传递发生改变的量，否则会导致结果不可测**

**解决办法**



案例：

//数据传递的结构体

struct Param {

int type;

char \*message;

};

void createThread() {

//定义传递参数的结构体，注意这里的new操作，这里非常重要！

Param \*param = new Param();

param->type = 1024;

param->message = "thread message";

//这里函数指针\*threadFunc采用&threadFunc

//参数void \*data使用param指针

pthread\_create(&pt, NULL, &threadFunc, param);

}

//必须是static的静态函数

static void \*threadFunc(void \*data) {

//参数需要做一次强转

Param \*param = (Param \*)data;

//日志打印，按照自己的需求处理。。

print("Thread", "message = %s type = %d",param->message, param->type);

pthread\_exit(0);

return NULL;

}

**C++线程使用的注意点**

1.复杂参数的传递需要使用结构体   
2. 结构体初始化一定要用Param \*param = new Param(); 用malloc(..);是不行的！   
3. 线程实际执行的函数必须定义为static void \*threadFunc(void \*data)，除了函数名称以外别的都必须一致   
4. 取参数的时候需要做一次强转：Param *param = (Param*)data;

6、关于send和recv函数

int send( SOCKET s, const char FAR \*buf, int len, int flags );

四个参数：套接字，存放应用程序要发送数据的缓冲区， 实际要发送的数据的字节数；一般为0

send函数只负责将数据提交给协议层。 当调用该函数时，send先比较待发送数据的长度len和套接字s的发送缓冲区的长度，如果len大于s的发送缓冲区的长度，该函数返回SOCKET\_ERROR； 如果len小于或者等于s的发送缓冲区的长度，那么send先检查协议是否正在发送s的发送缓冲中的数据； 如果是就等待协议把数据发送完，如果协议还没有开始发送s的发送缓冲中的数据或者s的发送缓冲中没有数据，那么send就比较s的发送缓冲区的剩余空间和len； 如果len大于剩余空间大小，send就一直等待协议把s的发送缓冲中的数据发送完，如果len小于剩余空间大小，send就仅仅把buf中的数据copy到剩余空间里（注意并不是send把s的发送缓冲中的数据传到连接的另一端的，而是协议传的，send仅仅是把buf中的数据copy到s的发送缓冲区的剩余空间里）。 如果send函数copy数据成功，就返回实际copy的字节数，如果send在copy数据时出现错误，那么send就返回SOCKET\_ERROR； 如果send在等待协议传送数据时网络断开的话，那么send函数也返回SOCKET\_ERROR。

int recv( SOCKET s, char FAR \*buf, int len, int flags);

recv先检查套接字s的接收缓冲区，如果s接收缓冲区中没有数据或者协议正在接收数据，那么recv就一直等待，直到协议把数据接收完毕。当协议把数据接收完毕，recv函数就把s的接收缓冲中的数据copy到buf中（注意协议接收到的数据可能大于buf的长度，所以在这种情况下要调用几次recv函数才能把s的接收缓冲中的数据copy完。recv函数仅仅是copy数据，真正的接收数据是协议来完成的），recv函数返回其实际copy的字节数。如果recv在copy时出错，那么它返回SOCKET\_ERROR；如果recv函数在等待协议接收数据时网络中断了，那么它返回0 。

  对方优雅的关闭socket并不影响本地recv的正常接收数据；如果协议缓冲区内没有数据，recv返回0，指示对方关闭；如果协议缓冲区有数据，则返回对应数据(可能需要多次recv)，在最后一次recv时，返回0，指示对方关闭。

在进行TCP协议传输的时候，要注意数据流传输的特点，recv和send不一定是一一对应的（一般情况下是一一对应），也就是说并不是send一次，就一定recv一次就接收完，有可能send一次，recv多次才接收完，也可能send多次，一次recv就接收完了。TCP协议会保证数据的有序完整的传输，但是如何去正确完整的处理每一条信息，是程序员的事情。

例如：服务器在循环recv，recv的缓冲区大小为100byte，客户端在循环send，每次send 6byte数据，则recv每次收到的数据可能为6byte，12byte，18byte，这是随机的，编程的时候注意正确的处理。

4.11

1、pair的应用

pair是将2个数据组合成一个数据，当需要这样的需求时就可以使用pair，如stl中的map就是将key和value放在一起来保存。另一个应用是，当一个函数需要返回2个数据的时候，可以选择pair。 pair的实现是一个结构体，主要的两个成员变量是first second 因为是使用struct不是class，所以可以直接使用pair的成员变量。使用方法如下：

初始化：

pair使用头文件iostream

pair<int, float> p1;

pair<int, float> p2(1, 1.2)

pair<int, float> p3(P1)

访问元素：

P1.firsr = 1; p1.sencod = 1.2

赋值operator =

Pair<int, float> p1， p2;

P1 = make\_pair(1, 1.5) // 常用

p2 = P1; // 赋值

2、map的使用

定义：

Typedef map<int, string> Mymap

Mymap m1;

添加数据：

M1[3] = “hello” or

M1.insert(pair<int string>(3, “hello”))

M1.insert(map<int, string> ::value\_type(3, “hello”))

遍历：

map<int, string>::iterator it;

for(it = M1.begin(); it != M1.end()++it;) {cout<< it->first<<endl;}

查找：

It = M1.find(112)

If (it != m1.end()) {find}

删除：

m1.erase(it)

排序：map种key 自动排序 不需要排序

判读是否为空m1.empty()

3、自定义排序

std::sort：对给定区间所有元素进行排序。

std::stable\_sort：对给定区间所有元素进行稳定排序，稳定排序算法能够维持相等元素的原有顺序。

 std::partial\_sort：对给定区间所有元素进行部分排序

用法：

**static** **bool** myfunction(**int** i, **int** j) { **return** (i < j); }

vector<**int**> myvector // 32 71 12 45 26 80 53 33

std::sort(myvector.begin(), myvector.begin() + 4); // 12 32 45 71 26 80 53 33

std::sort(myvector.begin(), myvector.end(), std::greater<**int**>()); // descending is to use std::greater()

sort(myvector.begin() + 4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

// 部分排序, 只获取前五个最小的，找出最大的前五个数，并且有序

vector<int> myvector; // 41 67 34 0 69 24 78 58 62 64

std::partial\_sort(myvector.begin(), myvector.begin() + 4, myvector.end());

// 0 24 34 41 69 67 78 58 62 64

这里和sort(myvector.begin(), myvector.begin() + 4) 不一样，这里是只排序前4个

// 自定义

std::partial\_sort(myvector.begin(), myvector.begin() + 5, myvector.end(),myfunction);

4、const\_cast 的用法 const\_cast只能转换指针或引用

主要是用来去掉const属性，当然也可以加上const属性。主要是用前者，后者很少用

**1. 常量指针被转化成非常量指针，转换后指针指向原来的变量(即转换后的指针地址不变)。**

1. //1. 指针指向类
2. **const** A \*pca1 = **new** A;
3. A \*pa2 = **const\_cast**<A\*>(pca1);  //常量对象转换为非常量对象
4. pa2->m\_iNum = 200;    //fine
5. //2. 指针指向基本类型
6. **const** **int** ica = 100;
7. **int** \* ia = **const\_cast**<**int** \*>(&ica);
8. \*ia = 200;
9. cout<< \*ia <<ica<<endl;   //200 100

**2. 常量引用转为非常量引用。**

1. A a0;
2. **const** A &a1 = a0;
3. A a2 = **const\_cast**<A&>(a1);　//常量引用转为非常量引用

**3、常量对象(或基本类型)不可以被转换成非常量对象(或基本类型)。**

1. **//常量对象被转换成非常量对象时出错**
2. **const A ca;**
3. **A a = const\_cast<A>(ca);  //不允许**
4. **const int i = 100;**
5. **int j = const\_cast<int>(i);  //不允许**

5、智能指针

6、vector初始化

int myints[] = { 32,71,12,45,26,80,53,33 };

std::vector<int> myvector(myints, myints + 8);// 32 71 12 45 26 80 53 33

1. **const** **float**\* begin = output\_layer->cpu\_data();
2. **const** **float**\* end = begin + output\_layer->channels();
3. **return** std::vector<**float**>(begin, end);

4.12

1、某个目录下查找关键字

grep –n –H –R “hello” n显示行号， h显示文件名， r递归查找

2、修改mysql密码：mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('newpass');

4.17

$PATH 环境变量的作用

关于PATH的作用：  
PATH说简单点就是一个字符串变量，当输入命令的时候LINUX会去查找PATH里面记录的路径。比如在根目录/下可以输入命令ls,在/usr目录下也可以输入ls,但其实ls这个命令根本不在这个两个目录下，事实上当你输入命令的时候LINUX会去/bin,/usr/bin,/sbin等目录下面去找你此时输入的命令，而PATH的值恰恰就是/bin:/sbin:/usr/bin:……。其中的冒号使目录与目录之间隔开。

关于新增自定义路径：  
现在假设你新安装了一个命令在/usr/locar/new/bin下面，而你又想像ls一样在任何地方都使用这个命令，你就需要修改环境变量PATH了，准确的说就是给PATH增加一个值/usr/locar/new/bin。你只需要一行bash命令export PATH=$PATH:/usr/locar/new/bin。这条命令的意思太清楚不过了，使PATH自增:/usr/locar/new/bin,既PATH=PATH+":/usr/locar/new/bin";通常的做法是把这行bash命令写到/root/.bashrc的末尾，然后当你重新登陆LINUX的时候（应该是linux启动时就会执行这个文件），新的默认路径就添加进去了。当然这里你直接用source /root/.bashrc执行这个文件重新登陆了。你可以用echo $PATH命令查看PATH的值。

LD\_LIBRARY\_PATH的作用：

1、编译库相关知识

可以得出动态库的搜索路径搜索的先后顺序是：

1.编译目标代码时指定的动态库搜索路径；

2.环境变量LD\_LIBRARY\_PATH指定的动态库搜索路径；

3.配置文件/etc/ld.so.conf中指定的动态库搜索路径；

4.默认的动态库搜索路径/lib；

5.默认的动态库搜索路径/usr/lib。

4.18

1、cmake

一般来说，都是这样用

cmake ./

这样PROJECT\_BINARY\_DIR和PROJECT\_SOURCE\_DIR是等价的。也就是当前源码的目录。

如果执行cmake的时候，并不在源码的路径的话，比如

cmake ../src

这样的好处是cmake生成的文件和编译出来的东西，就不放在源码路径下了，保证了源码路径的干净整洁。

比如可以在src的同级目录下建立build目录。

然后在build目录下执行cmake ../src。

这样编译出来的东西和cmake生成的东西，都放到了build目录下了。并且

PROJECT\_BINARY\_DIR=全路径/build

PROJECT\_SOURCE\_DIR=全路径/src

Cmake list:

list(LENGTH <list><output variable>)  
  list(GET <list> <elementindex> [<element index> ...]  
       <output variable>)  
  list(APPEND <list><element> [<element> ...])  
  list(FIND <list> <value><output variable>)  
  list(INSERT <list><element\_index> <element> [<element> ...])  
  list(REMOVE\_ITEM <list> <value>[<value> ...])  
  list(REMOVE\_AT <list><index> [<index> ...])  
  list(REMOVE\_DUPLICATES <list>)  
  list(REVERSE <list>)  
  list(SORT <list>)

LENGTH返回列表的长度

GET返回列表中指定下标的元素

APPEND添加新元素到列表中

INSERT 将新元素插入到列表中指定的位置

REMOVE\_ITEM从列表中删除某个元素

REMOVE\_AT从列表中删除指定下标的元素

REMOVE\_DUPLICATES从列表中删除重复的元素

REVERSE 将列表的内容实地反转，改变的是列表本身，而不是其副本

SORT 将列表按字母顺序实地排序，改变的是列表本身，而不是其副本

2、安装库过程中遇到的坑

Sudo make && make install VS Sudo make && sudo make install

一些特殊符号：-- vs -

3、编译库和运行库

当安装一些开源软件时， 编译的时候需要指定头文件和动态库的位置

比如： -I/usr/local/include/mongocxx/v\_noabi\

-L/usr/local/lib

运行的时候不需要指定头文件，但是需要指定动态库，

动态库的位置可通过 ldd命令查看

我们也可以通过 export LD\_LIBRARY\_PATH=./lib/caffe 设置临时环境变量，也可以通过修改用户环境变量 gedit ~/.bashrc 或者修改全局变量 gedit /etc/profile ，在最后加上这一行

4、float类型转string

5、char \*str 转 string

如果字符串 包含\0， 则直接 string mystr = str;

如果不包含， 则 string mystr = string(str, len),一般用于读取二进制

4.23

1、shell相关操作

删除查询出来的文件：find $data/train –name \*.jpg |xargs rm

将查出来的数据拷贝到某个路径：

find $fata/tarin –name \*.jpg |xargs cp –n1 -i {} ./train

创建并循环数组:

$arr=(1 2 3 4 5 6)

for i in ${arr[@]}

cut和sed

find $fata/tarin –name \*.jpg |cut -d ‘/’-f2-4 | sed “s/$/ $i”

以“/”分割，获取2到4个字符串 sed 在最后一行加上“ $i”

2、

Sprintf 格式化输出、格式化某个字符串

//可以指定宽度，不足的左边补空格：

sprintf(s, "%8d%8d", 123, 4567); // " 123 4567"

sscanf 格式化输入

sscanf("123456 ", "%4s", buf);

printf("%s\n", buf); // "1234"

3、显示构造函数

class ClxComplex  
{  
public:  
    ClxComplex(double dReal = 0.0, double dImage = 0.0) { m\_dReal = dReal; dImage = dImage; }  
  
    double GetReal() const { return m\_dReal; }  
    double GetImage() const { return m\_dImage; }  
  
private:  
    double m\_dReal;  
    double m\_dImage;  
};

 我们知道，下面的3行代码是等价的：

ClxComplex lxTest = 2.0;  
ClxComplex lxTest = ClxComplex(2.0);  
ClxComplex lxTest = ClxComplex(2.0, 0.0);

  其实，对于前两行来说，编译器都是把它们转换成第3行的代码来实现的。因为我们写了构造函数，编译器就按照我们的构造函数来进行隐式转换，直接把一个double数值隐式转换成了一个ClxComplex的对象。可是，有些时候，我们不希望进行隐式转换，或者隐式转换会造成错误。比如下面的一个简化的字符串类：

class ClxString  
{  
public:  
    ClxString(int iLength);  
    ClxString(const char \*pString);  
    ~ClxString();  
  
private:  
    char \*m\_pString;  
};  
  
ClxString::ClxString(int iLength)  
{  
    if (iLength > 0)  
        m\_pString = new char[iLength];  
}  
  
ClxString::ClxString(const char \*pString)  
{  
    m\_pString = new char[strlen(pString)];  
    strcpy(m\_pString, pString);  
}  
  
ClxString::~ClxString()  
{  
    if (m\_pString != NULL)  
        delete m\_pString;  
}

这会给阅读代码造成不必要的歧义。

ClxString lxTest = 13;  // 等同于ClxString lxTest = ClxString(13);

ClxString lxTest = "A";  // 等同于ClxString lxTest = ClxString("A"); 正确

  那下面的代码就会初始化一个长度为65（字母A的ASCII码值，在C和C++中，字符是以ASCII值存储的）的字符串。错误的，这个时候需要用到显示

ClxString lxTest = 'A';  // 等同于ClxString lxTest = ClxString(65);

class ClxString  
{  
public:  
    explicit ClxString(int iLength);  
    ClxString(const char \*pString);  
    ~ClxString();  
  
private:  
    char \*m\_pString;  
};

在这种情况下，要想用字符串的长度来初始化一个ClxString对象，那就必须显示的调用构造函数：

ClxString lxTest = ClxString(13);

以下是错误的：

ClxString lxTest = 13;   
ClxString lxTest = 'A';

4.24.2018

1、cmake find\_packge 命令

用于寻找cmake 编译库的头文件和动态库

举个例子：以caffe为例

find\_packge(Caffe) // 表示默认路径

find\_packge(Caffe NO\_DEFAULT\_PATH PATHS “/home/hailong”)

include\_directories(${CAFFE\_INCLUDE\_DIRS})

cuda\_add\_executable(http\_park)

target\_link\_libraries(http\_park ${Caffe\_LIBRARIES}) //连接动态库

以bizip 为例：

cmake\_minimum\_required(VERSION 2.8)

project(helloworld)

add\_executable(helloworld hello.c)

find\_package(BZip2)

if (BZIP2\_FOUND)

include\_directories(${BZIP\_INCLUDE\_DIRS})

target\_link\_libraries (helloworld ${BZIP2\_LIBRARIES})

endif (BZIP2\_FOUND)

2、c++保护成员函数

派生类从基类公有继承时，派生类的成员函数可以直接访问基类的公有成员，但不能访问基类的私有成员。

       因此，为了便于派生类的访问，可以将基类的私有成员中需要提供给派生类访问的成员定义为保护成员。

       说白了就一句话，派生类可以访问protected权限的成员但是派生类的对象不能访问基类的成员。

3、成员函数后面加const 的作用

我们定义的[类的成员函数](https://www.baidu.com/s?wd=%E7%B1%BB%E7%9A%84%E6%88%90%E5%91%98%E5%87%BD%E6%95%B0&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4ujRYujwhujbkn1fdmhRL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWmzPjnLrH6Y)中，常常有一些成员函数不改变类的数据成员，也就是说，这些函数是"只读"函数，而有一些函数要修改类数据成员的值。如果把不改变数据成员的函数都加上const关键字进行标识，显然，可提高程序的可读性。其实，它还能提高程序的可靠性，已定义成const的成员函数，一旦企图修改数据成员的值，则编译器按错误处理。 const成员函数和const对象 实际上，const成员函数还有另外一项作用，即常量对象相关。对于内置的数据类型，我们可以定义它们的常量，用户自定义的类也一样，可以定义它们的常量对象。

非[静态成员](https://www.baidu.com/s?wd=%E9%9D%99%E6%80%81%E6%88%90%E5%91%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ryf3rAF9uj-bn1b1PAR30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHDsnWb4nWn1)函数后面加const（加到非成员函数或[静态成员](https://www.baidu.com/s?wd=%E9%9D%99%E6%80%81%E6%88%90%E5%91%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ryf3rAF9uj-bn1b1PAR30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHDsnWb4nWn1)后面会产生编译错误），表示成员函数隐含传入的this指针为 [const指针](https://www.baidu.com/s?wd=const%E6%8C%87%E9%92%88&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ryf3rAF9uj-bn1b1PAR30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHDsnWb4nWn1)，决定了在该成员函数中，任意修改它所在的类的成员的操作都是不允许的（因为隐含了对this指针的const引用）；唯一的例外是对于 mutable修饰的成员。加了const的成员函数可以被非const对象和const对象调用，但不加const的成员函数只能被非const对象调 用。

4、vector 定义个数

Vector<int> shape(4); // 容量为4

4.25.2018

1、shell 命令

压缩：tar –zcvf xxx.tar.gz xxx1 xxx2

解压：tar –zxvf xxxl.tar.gz

复制：scp ccc.tar.gz [oeasy@192.168.0.111:/home/oeasy](mailto:oeasy@192.168.0.111:/home/oeasy)

scp -r oeasy@192.168.0.223:/home/oeasy/PlateServer\_bin\_lhl/lib/darknet . //复制到当前

拷贝到192.168.0.111:/home/oeasy

2、cmake编译debug

1、通过命令行的方式

     cmake  -DCMAKE\_BUILD\_TYPE=Debug ..

2、set(CMAKE\_BUILD\_TYPE Debug CACHE STRING "set build type to debug")

     或者

     set(CMAKE\_BUILD\_TYPE "Debug")

3、du –h du –ha 查看当前磁盘占用情况

Cd – 返回上一级

5.4

1、程序的编译链接过程

源文件到可执行文件要经过以下过程：

源文件-》预处理-》编译-》汇编-》链接-》二进制可执行文件

预处理：

（1）将所有的#define删除，并展开所有的宏定义

（2）处理所有的条件预编译指令，如：#if  #ifdef #elif #else #endif

（3）处理#include预编译指令，将被包含的文件插进到该指令的位置，这个过程是递归的

（4）删除所有的注释//与/\* \*/

（5）添加行号与文件名标识，以便产生调试用的行号信息以及编译错误或警告时能够显示行号

（6）保留所有的#pragma编译器指令，因为编译器需要使用它们

编译：

词法分析-》语法分析-》语义分析-》优化

汇编：汇编代码-》机器指令

链接：多个目标文件、库-》可执行文件

2、函数模板和类模板

使用场景：

类模板：多个类有着共同操作，但是数据类型不同

函数模板：多个函数有着共同操作，但是数据类型不同

使用注意事项：

函数模板：声明与定义分离会报错

Stl中函数模板的声明和定义放在一起，如果使用传统发做法即.h文件里声明,在.cpp文件里定义，会出现连接错误，原因：函数模板只要在使用的时候才会实例化，编译的时候不会。

简单的解释：这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

具体解释：

假设test.h中声明了add函数，test.cpp定义实现了该函数，在main.cpp中你调用了add函数。

对普通函数来说，声明放在头文件中，定义放在源文件中，其它的地方要使用该函数时，仅需要包含头文件即可，因为编译器编译时是以一个源文件作为单元编译的，当它遇到不在本文件中定义的函数时，若能够找到其声明，则会将此符号放在本编译单元的外部符号表中，链接的时候自然就可以找到该符号的定义了。

而对模板函数来说，首先明确，模板函数是在编译器遇到使用模板的代码时才将模板函数实例化的。若将模板函数声明放在tem.h，模板定义放在tem.cpp,在main.cpp中包含头文件，调用add,按道理说应该实例化int add(int,int)函数，即生成add函数的相应代码，但是此时仅有声明，找不到定义，因此此时，它只会实例化函数的符号，并不会实例化函数的实现，即这个时候，在main.o编译单元内，它只是将add函数作为一个外部符号，这就是与普通函数的区别，对普通函数来说，此时的add函数已经由编译器生成相应的代码了，而对模板函数来说，此时并没有生成add函数对应的代码。此时编译main.cpp单元不会报错，但链接就会出现add函数未定义的错误。

要使模板声明与定义分开也不是没有办法：

法一：

Main函数同事包含.h 和.cpp

法二：

tem.h  
template<typename T> T add(T a,T b);

tem.cpp

using namespace std;  
//#include "tem.h"  
  
  
template <typename T> T add(T a, T b){return a+b;}  
template int add(int,int);//实例化定义，必须放在模板定义的后面

main.cpp

#include <iostream>  
#include "tem.h"  
using namespace std;  
int main()  
{  
cout << add(1,2);  
return 0;  
}

因此，我们可以通过显式的实例化定义，即通过加上语句temmplate int add(int,int),编译器看到此语句将会生成add方法的int版本，这样的话，再链接就不会报错了。此外，这样做通常也能够提高编译的效率。试想，如果在tem.h文件内定义模板，假如有三个源文件均包含了该头文件且均使用了模板（假定均调用了add模板的int版本），则在这三个源文件内必然都会生成add函数的实例。显然效率不高。而如果像上面那样使用该模板，则只会在tem.cpp文件中实例化。

类模板使用方法：

1. #include "stdafx.h"
2. #include <iostream>
4. **using** **namespace** std;
6. **template**<**class** dataType>
7. **class** Compare
8. {
9. **private**:
10. dataType x,y;
11. **public**:
12. Compare(dataType a,dataType b)
13. {
14. x=a;
15. y=b;
16. }
17. dataType getMax()
18. {
19. **return** (x>y)? x:y;
20. }
21. };
23. **int** main(**int** argc, **char**\* argv[])
24. {
25. //类模板的实例化（To 类）and类的实例化（To 对象）
26. Compare<**int**> cmp1(3,7);
27. cout<<cmp1.getMax()<<" is the Maximum of two Integer numbers"<<endl;
29. Compare<**float**> cmp2(12.3,23.4);
30. cout<<cmp2.getMax()<<" is the Maximun of two Float numbers"<<endl;
31. **return** 0;
32. }

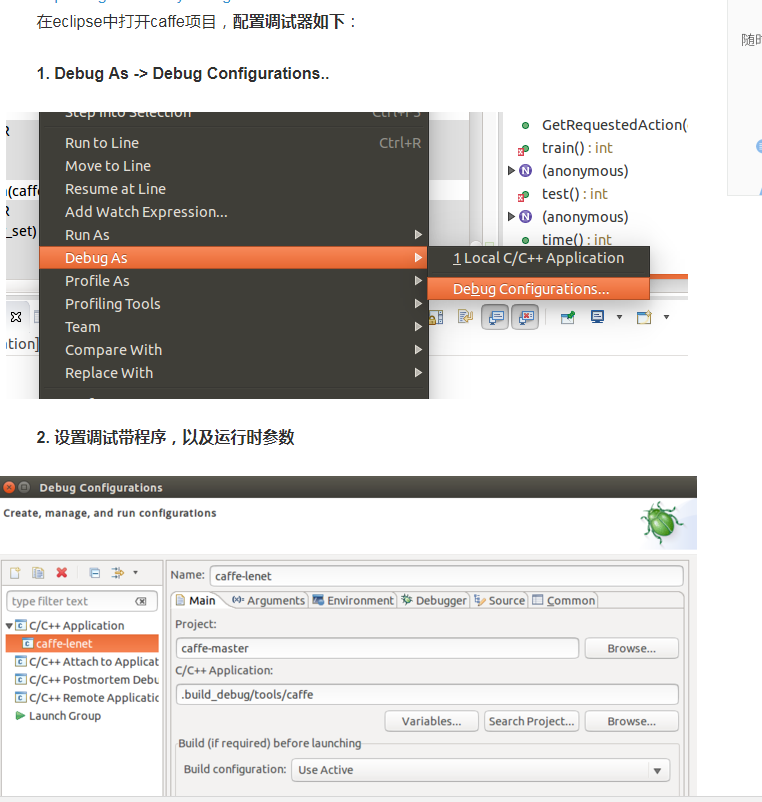
3、Linux下eclipce调试

cd /opt sudo mkdir jvm sudo mkdir ecplice

安装jdk 1.8版本：可使用sudo apt-get 查看版本：java –version

安装ecplice: wget   http://ftp.yz.yamagata-u.ac.jp/pub/eclipse/technology/epp/downloads/release/neon/3/eclipse-cpp-neon-3-linux-gtk-x86\_64.tar.gz

解压将其拷贝到/opt/ecplice/下 直接运行 ./eclipce &



注意事项：

* 1. 必须编译成debug类型 mkdir build cd build cmake .. DCMAKE\_BUILD\_TYPE=Debug make –j4
  2. 确保源文件存在编译环境下

查看opencv版本:

pkg-config opencv --modversion

查看库：pkg-config opencv –libs

5.14.2018

1、标准库在<algorithm>头中定义了两个模板函数std::min() 和 std::max()

用法：N = std::min<int>(labels\_.size(), N); // 获取两个int中较大的数

5.15

1、关于继承

派生类从基类公有继承时，派生类的成员函数可以直接访问基类的公有成员，但不能访问基类的私有成员。

       因此，为了便于派生类的访问，可以将基类的私有成员中需要提供给派生类访问的成员定义为保护成员。

2、构造函数能否继承

析构函数被继承过来了，但是构造函数不能够继承，我们在子类里面只是”调用“基类的构造函数。

注意：当我们需要调用父类的构造函数时，可以采用初始化列表的方法。

3、关于调用父类构造函数

子类在构造时，如果没有显式调用父类的构造函数，会先调用父类的默认构造函数（无参数的）

换句话说，在使用继承时，如果父类没有默认的构造函数，则子类的构造函数应当显式地调用父类的自定义构造函数。

例子如下：

1. #include <iostream>
2. **class** Base{
3. **public**:
4. Base(){
6. }
7. };
9. **class** Derive:**public** Base{
10. **public**:
11. Derive(**int** a,**int** b){  // 父类有默认构造函数
13. }
14. };
16. **int** main(){
18. Derive c(1,2);
20. **return** 0;
21. }

父类有默认构造函数，编译通过

1. #include <iostream>
2. **class** Base{
3. **public**:
4. Base(**int** a){
6. }
7. };
9. **class** Derive:**public** Base{
10. **public**:
11. Derive(**int** a,**int** b){   // 父类无默认构找函数，这里需要显示调用父类构造函数
13. }
14. };
16. **int** main(){
18. Derive c(1,2);
20. **return** 0;
21. }

父类没有默认的构造函数，编译失败

1. #include <iostream>
2. **class** Base{
3. **public**:
4. Base(**int** a){
5. **this**->a = a;
6. }
7. Base(){
9. }
10. **public**:
11. **int** a;
12. };
14. **class** Derive:**public** Base{
15. **public**:
16. Derive(**int** a){
18. }
19. };
21. **int** main(){
23. Derive c(1);
24. std::cout << c.a << std::endl;
25. **return** 0;
26. }

父类包含默认构造函数，编译通过

4、关于多态

条件：要有继承、要有virtual， 要有父类指针或者引用指向子类

例子：

1 #include "stdafx.h"

2 #include <iostream>

3 #include <stdlib.h>

4 using namespace std;

5

6 class Father

7 {

8 public:

9 void Face()

10 {

11 cout << "Father's face" << endl;

12 }

13

14 void Say()

15 {

16 cout << "Father say hello" << endl;

17 }

18 };

19

20

21 class Son:public Father

22 {

23 public:

24 void Say()

25 {

26 cout << "Son say hello" << endl;

27 }

28 };

29

30 void main()

31 {

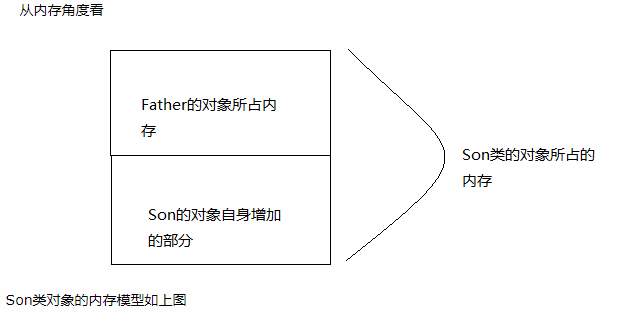
32 Son son;

33 Father \*pFather=&son; // 隐式类型转换

34 pFather->Say();

35 }

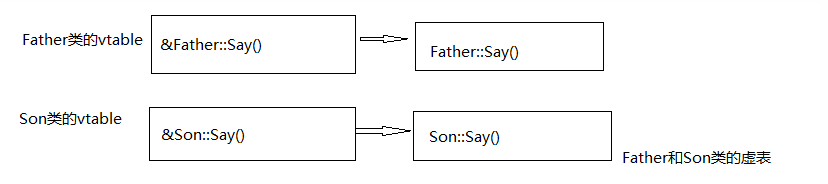
这里不能实现多态，没有virtual关键字告诉编译器动态绑定。



我们构造Son类的对象时，首先要调用Father类的构造函数去构造Father类的对象，然后才调用Son类的构造函数完成自身部分的构造，从而拼接出一个完整的Son类对象。当我们将Son类对象转换为Father类型时，该对象就被认为是原对象整个内存模型的上半部分，也就是上图中“Father的对象所占内存”，那么当我们利用类型转换后的对象指针去调用它的方法时，当然也就是调用它所在的内存中的方法，因此，输出“Father Say hello”，也就顺理成章了。

加上virtual后可实现多态，分析如下

编译器在编译的时候，发现Father类中有虚函数，此时编译器会为每个包含虚函数的类创建一个虚表(即 vtable)，该表是一个一维数组，在这个数组中存放每个虚函数的地址



如何定位呢？

编译器另外还为每个对象提供了一个虚表指针(即vptr)，这个指针指向了对象所属类的虚表，在程序运行时，根据对象的类型去初始化vptr，从而让vptr正确的指向了所属类的虚表，从而在调用虚函数的时候，能够找到正确的函数，对于第二段代码程序，由于pFather实际指向的对象类型是Son，因此vptr指向的Son类的vtable，当调用pFather->Son()时，根据虚表中的函数地址找到的就是Son类的Say()函数.

总结：

* 1. 每一个类都有虚表，父类有虚表，子类也有；
  2. 每个对象都会生成一个虚表指针，运行的时候动态决定的效果。

5、python os.walk用法

import os

for root, dirs, files in os.walk("."):

for name in files:

print(os.path.join(root, name))

for name in dirs:

print(os.path.join(root, name))

6、对某个目录下的图片重命名

7、总结split join strip

5.17

1、关于makefile

规则：

**target... : prerequisites ...**

**command**

**...**

换行要用Tab键。

第一个版本：

objects = main.o kbd.o command.o display.o \

             insert.osearch.o files.o utils.o

   edit : $(objects)

           cc -o edit $(objects)

   main.o : main.c defs.h

           cc -c main.c

   kbd.o : kbd.c defs.h command.h

           cc -c kbd.c

   command.o : command.c defs.h command.h

           cc -c command.c

   display.o : display.c defs.h buffer.h

           cc -c display.c

   insert.o : insert.c defs.h buffer.h

           cc -c insert.c

   search.o : search.c defs.h buffer.h

           cc -c search.c

   files.o : files.c defs.h buffer.h command.h

           cc -c files.c

   utils.o : utils.c defs.h

           cc -c utils.c

   clean :

           rm edit $(objects)

第二个版本：自动推导

objects = main.o kbd.o command.o display.o \

             insert.o search.o files.o utils.o

   edit : $(objects)

           cc -o edit $(objects)

   main.o : defs.h

   kbd.o : defs.h command.h

   command.o : defs.h command.h

   display.o : defs.h buffer.h

   insert.o : defs.h buffer.h

   search.o : defs.h buffer.h

   files.o : defs.h buffer.h command.h

   utils.o : defs.h

   .PHONY : clean

   clean :

           rm edit $(objects)

只要make看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果make找到一个whatever.o，那么whatever.c，就会是whatever.o的依赖文件。并且 cc -c whatever.c 也会被推导出来。

.PHONY意思表示clean是一个“伪目标”，。而在rm命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。、

案例解释：

 $(objects): %.o: %.c

           $(CC) -c $(CFLAGS) $< -o $@

这是一种静态模式：

<targets...>: <target-pattern>: <prereq-patterns ...>

　　　<commands>

targets定义了一系列的目标文件， target-parrtern是指明了targets的模式，也就是的目标集模式，prereq-parrterns是目标的依赖模式。

如果我们的<target-parrtern>定义成“%.o”，意思是我们的集合中都是以“.o”结尾的，而如果我们的<prereq-parrterns>定义成“%.c”，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。

而命令中的“$<”和“$@”则是自动化变量，“$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“$@”表示目标集。

假如上面： objects = foo.o bar.o，则可以写成如下模式。

foo.o : foo.c

           $(CC) -c $(CFLAGS) foo.c -o foo.o

   bar.o : bar.c

           $(CC) -c $(CFLAGS) bar.c -o bar.o

变量的高级用法：

foo := a.o b.o c.o  
bar := $(foo:.o=.c)  
这个示例中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中所有以“.o”字串“结尾”全部替换成“.c”，所以我们的“$(bar)”的值就是“a.c b.c c.c”。  
另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：  
foo := a.o b.o c.o  
bar := $(foo:%.o=%.c)

追加变量：

objects += another.o

$? 所有比目标新的依赖目标的集合

$^ 所有的依赖目标的集合

2、关于c++和c头文件

C语言不支持extern "C"语法，如果我们想写一个头文件，同时支持被C和C++引用，该怎么办？可以使用C++的宏 "\_\_cplusplus"来判断是不是C++编译器

#ifdef \_\_cplusplus

extern "C" {

#endif

// 正式定义。。。

#ifdef \_\_cplusplus

}

#endif

在c++的头文件中，为了支持c编译的库，经常这样做

#pragma once

extern "C" {

// 正式定义。。。 c头文件

}

Class {

….

}

5.21.2018

1、access

#define R\_OK 4 /\* Test for read permission. \*/

#define W\_OK 2 /\* Test for write permission. \*/

#define X\_OK 1 /\* Test for execute permission. \*/

#define F\_OK 0 /\* Test for existence. \*/

1. #include <stdio.h>
2. #include <unistd.h>
4. **int** main(**void**)
5. {
6. **if**(access("test.txt", R\_OK)==0)  printf("READ OK\n");
7. **if**(access("test.txt", W\_OK)==0)  printf("WRITE OK\n");
8. **if**(access("test.txt", X\_OK)==0)  printf("EXEC OK\n");
9. **if**(access("test.txt", F\_OK)==0)   printf("File exist\n");
10. }

5.23.2018

1、关于using namespace std;

第一，不要在头文件中使用，防止二义性，可读性也不好；

第二，自定义变量最好不要与现有的库名字相同，防止二义性；

2、静态成员函数不能调用类成员；

3、class定义，后面加分号；

4、当c++调用c时，需要用到extern “C”，c++编译器解析代码与c编译器不一样，c++编译器会带上类名，等其他信息，多以当c程序遇到c++编译器的时候需要告诉编译器按照c的方式来编译 通过宏\_\_cplusplus 类似判断；

事实上C++代码是无法直接调用C代码的，这是因为C++编译器在编译.cpp文件时生成的函数名与C编译器在编译.c文件时生成的函数名是不一样的。比如在重载中。

解决办法：

extern "C" {

int socket\_send(); // 明确的告诉C++编译器，这是一个用C语言编译的函数

}

这样虽然在C++编译下没有问题了，但是如果一个.c文件再去包含这个头文件时，又会发生问题，因为extern "C"不是C语言的关键字，这样.c文件又不能包含这个头文件了。如何能够既让.cpp文件能够包含这个头文件，又能让.c文件能够包含这个头文件呢，于是下面的写法就产生了：

#ifdef \_\_cplusplus

extern "C" {

#endif

.......

#ifdef \_\_cplusplus

}

#endif

以上c和c++都可以包含使用。

不要在extern "C" 中引用c++模板库，否则会报错。

5、使用g++编译，Util.cpp里面无extern “C”,没有问题，但是util.c无extern “C”会有问题，原因：

因为c++编译器编译c的时候需要告诉编译器按照c的方式来编译。

5.25.2018

1、关于卷积运算

**卷积核的厚度=被卷积的图像的通道数**   
**卷积核的个数=卷积操作后输出的通道数=特征图个数**

**卷积核的通道 = 输入图片的通道**

**图片卷积和的大小：N = (W-F+2P)/S + 1**

2、Alexnet参数

C1：96×11×11×3(卷积核个数/宽/高/厚度) 34848个   
C2：256×5×5×48（卷积核个数/宽/高/厚度） 307200个   
C3：384×3×3×256（卷积核个数/宽/高/厚度） 884736个   
C4：384×3×3×192（卷积核个数/宽/高/厚度） 663552个   
C5：256×3×3×192（卷积核个数/宽/高/厚度） 442368个   
R1：4096×6×6×256（卷积核个数/宽/高/厚度） 37748736个   
R2：4096×4096 16777216个   
R3：4096×1000 4096000个

C3\c4进行了通道合并串行

AlexNet 中60M参数

AlexNet只有8层，但是它需要学习的参数有60000000个，相比如他的层数，这是一个很可怕的数字了，我们来计算下这些参数都是怎么来的：

C1：96×11×11×3(卷积核个数/宽/高/厚度) 34848个

C2：256×5×5×48（卷积核个数/宽/高/厚度） 307200个

C3：384×3×3×256（卷积核个数/宽/高/厚度） 884736个

C4：384×3×3×192（卷积核个数/宽/高/厚度） 663552个

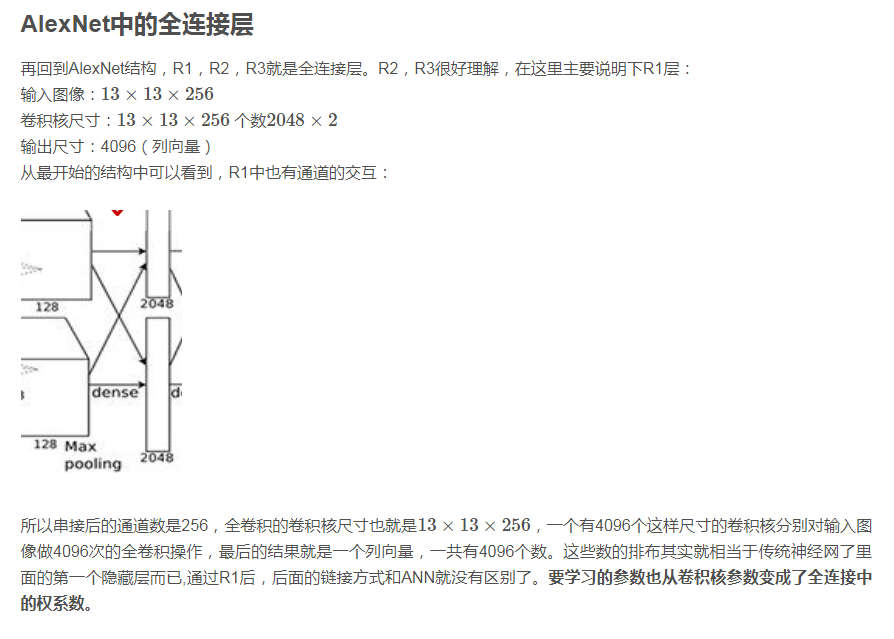
C5：256×3×3×192（卷积核个数/宽/高/厚度） 442368个

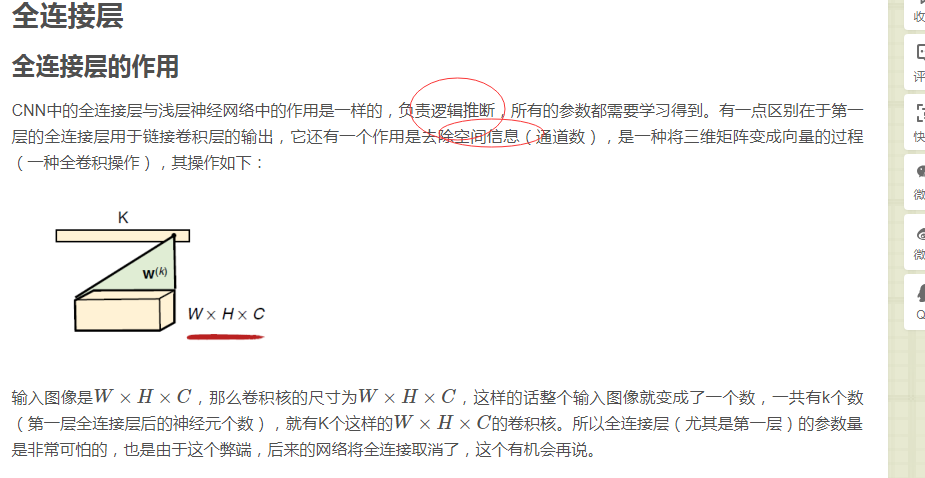
R1：4096×6×6×256（卷积核个数/宽/高/厚度） 37748736个

R2：4096×4096 16777216个

R3：4096×1000 4096000个

在R1中卷积核尺寸是6×6×256而不是13×13×256是因为经过了最大池化。可以看到，全连接层（尤其是第一层）参数数量占了绝大部分。

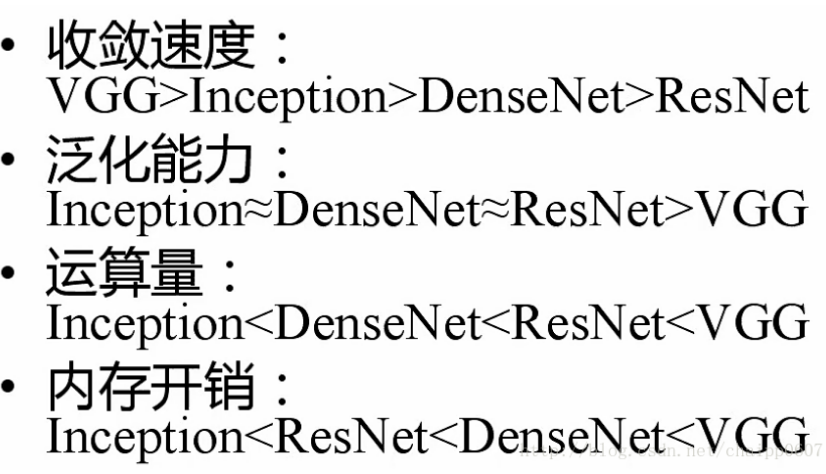




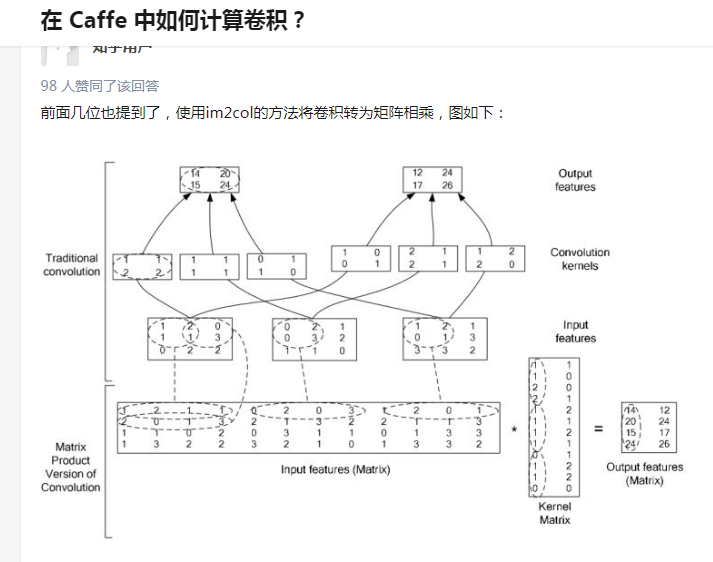
Googlenet中使用1\*1的卷积层降维：

由于稀疏结构的计算量大的问题，所以采用1x1的卷积来减少参数的计算，其中1x1 卷积解释为：

在3x3和5x5层前，各自增加一个1x1的卷积操作。1x1的卷积（或者网络层中的网络），提供了一个减少维度的方法。比如，我们假设你拥有一个输入层，体积是100x100x60（这并不定是图像的三个维度，只是网络中每一层的输入）。增加20个1x1的卷积滤波器，会让你把输入的体积减小到100x100x20。这意味着，3x3层和5x5层不需要处理输入层那么大的体积。这可以被认为是“池特征”（pooling of feature），因为我们正在减少体积的高度，这和使用常用的最大池化层（maxpooling layers）减少宽度和长度类似

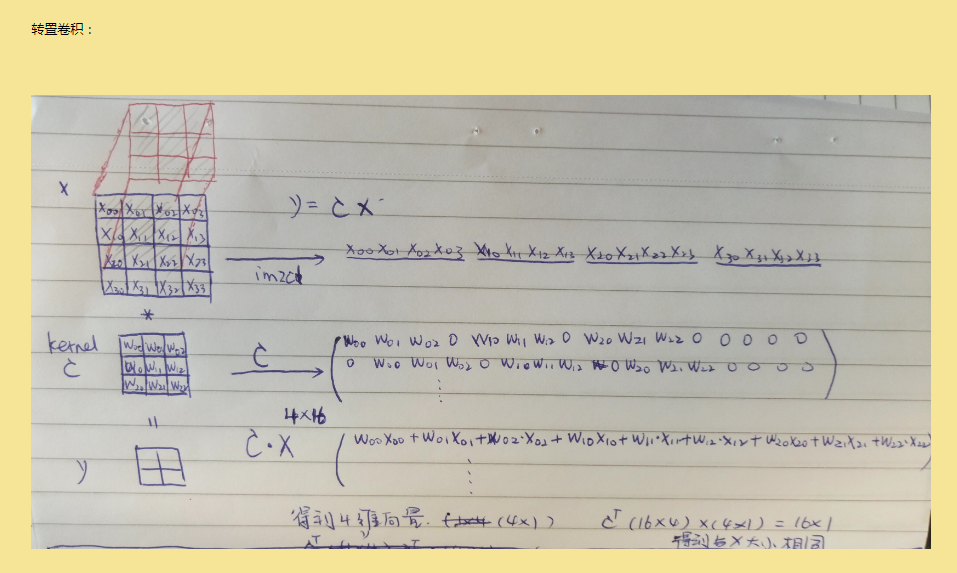


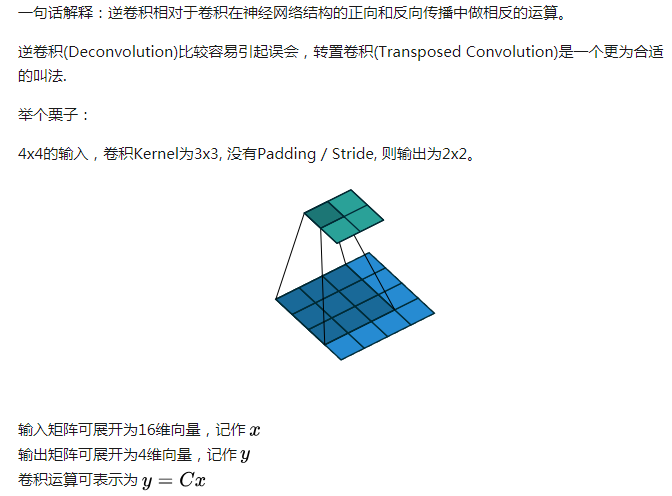
3、caffe里卷积的运算技巧

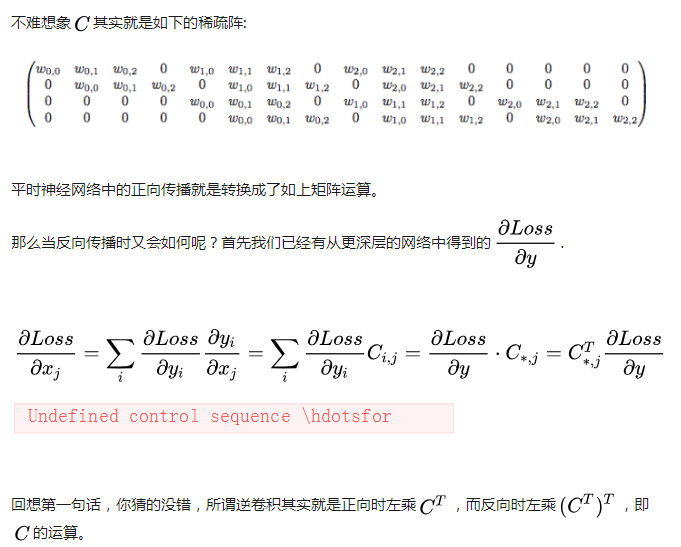


关于卷积的另外一种计算方法：

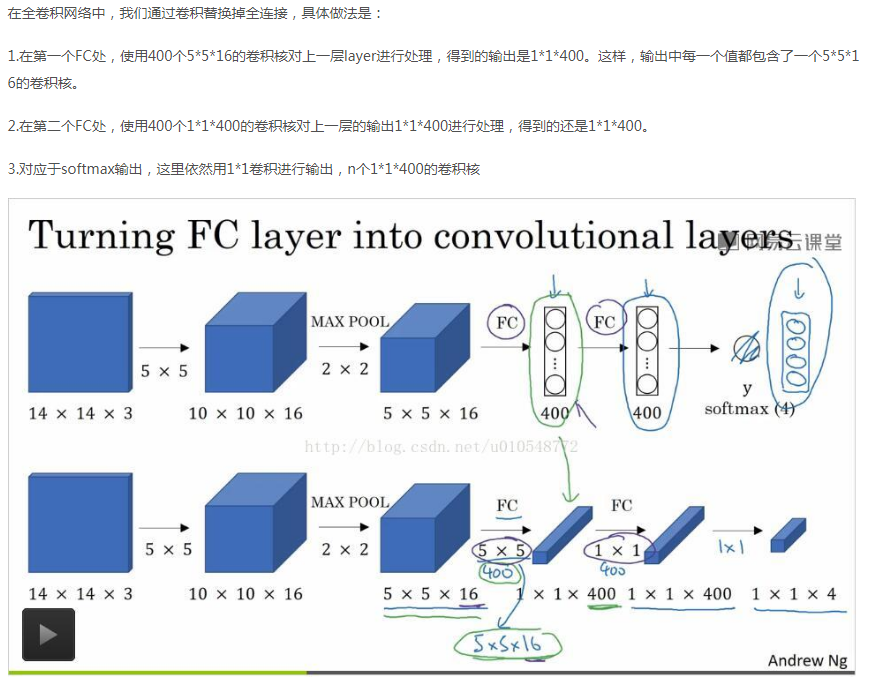
将输入图像转换成一维，将卷积核转换成多维







4、全连接转全卷积 ：分割检测中应用



5.28 检测算法

1、关于bounding-box regression

主要用于位置精修，找到一种预测坐标到groundtrueth的映射，使得最终结果尽量逼近真实的目标，近似的使用线性回归算法。主要包括平移和缩放两个步骤，

在rcnn中使用pool5层的特征向量来训练。具体过程如下：

<https://blog.csdn.net/u011534057/article/details/51235964>

2、nms原理 abcdef四个结果，首先按照概率来排序，比如，结果为fabcde, 拿出f分别和其他几个比较，当iou达到一定值时，就剔除掉，然后剩下的继续按照这个方法来剔除。

8.29.2018

精确率通常用于评价结果的质量，而召回率用来评价结果的完整性。

实际上，精确度（precision）是二元分类问题中一个常用的指标。

精确度就是标记为“正”，而确实是”正“的样本占所有标记为“正”的样本的比；和精确率一起出现的还有另一个指标召回率（recall）

关于模型评判标准：准确率、精确率、召回率、F1

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 实际值 | |
| 1 | 0 |
| 预测值 | 1 | TP | FP |
| 0 | FN | TN |

假如某个班级有男生80人,女生20人,共计100人.目标是找出所有女生.

现在某人挑选出50个人,其中20人是女生,另外还错误的把30个男生也当作女生挑选出来了

准确率：acc = (TN+TP)/TOTAL = (50+20) / 100

精确率：pri = TP/TP+FP = 20 / (20+30) # 也就是(20女生/(20女生+30误判为女生的男生)).

召回率：recall= TP / (TP + FN) # 也就是100%(20女生/(20女生+ 0 误判为男生的女生))

F1: F1 = 2\* pri\* recall /(pri + recall)

F1值就是精确值和召回率的调和均值

Fa = (squre(a) + 1)\* pri\* recall /(squre(a) \*pri + recall)

其中squre(a)为a的平方。

注意：准确率和召回率是互相影响的，理想情况下肯定是做到两者都高，但是一般情况下准确率高、召回率就低，召回率低、准确率高，当然如果两者都低，那是什么地方出问题了

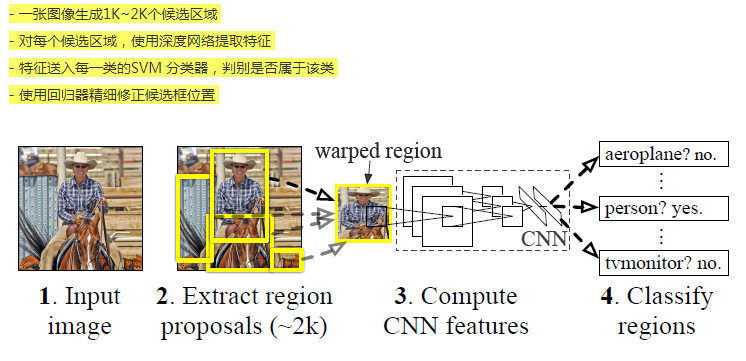
如果是做搜索，那就是保证召回的情况下提升准确率；如果做疾病监测、反垃圾，则是保准确率的条件下，提升召回。

在两者都要求高的情况下，可以用F1来衡量

5.29、rcnn系列检测算法

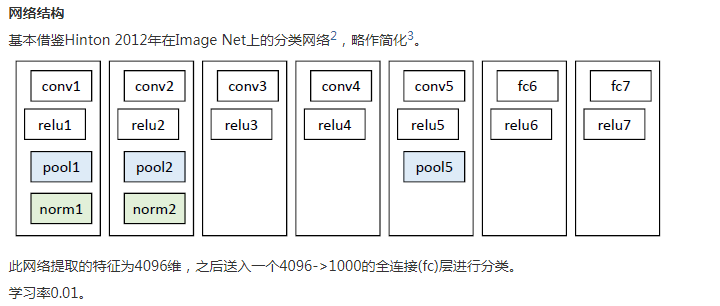
Rcnn: <https://www.cnblogs.com/zf-blog/p/6740736.html>

1、测试流程



位置精修前少了一步nms。

2、网络结构



3、训练过程

第一步：网络微调-》微调技巧

<https://www.cnblogs.com/louyihang-loves-baiyan/p/5038758.html>

使用上述网络，最后一层换成4096->21的全连接网络，学习率改成0.001，每一个batch包含32个正样本（属于20类）和96个背景。

正样本===》IOU > 0.5 + GT

负样本===》IOU < 0.5

第二步：训练20类的SVM分类器，SVM适用于小样本训练

使用fc7层输出的特征向量作为输入数据

正样本===》GT

负样本===》IOU < 0.3

第三步：bbox回归（位置精修）

输入为深度网络pool5层的4096维特征，输出为xy方向的缩放和平移。

采用线性回归，正样本===》pool5层的特征向量6\*6\*256维

Notes:

1、为什么要使用svm分类器？为什么微调的时候正样本iou>0.5？

因为网络分类器训练的时候采用的是正样本IOU大于0.5，分类不是很准确，那能不能IOU小一点，不能，因为微调数据量太小。

Sppnet:

解决的问题：

* 1. 多尺度输入
  2. 原图只需要卷积一次，降低运算量

存在的问题：未解决多阶段训练的繁琐过程

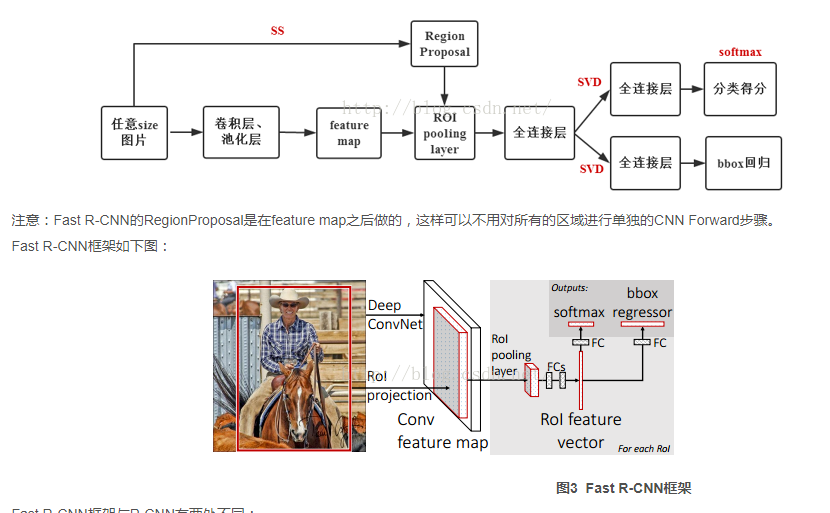
Fastrcnn：<https://blog.csdn.net/u011534057/article/details/51241831>

解决的问题：

* 1. Roipool层，固定输出，roi特征图映射
  2. 多任务学习，使用L1范式回归
  3. End-to-end训练方式
  4. Svd分解，加速运算

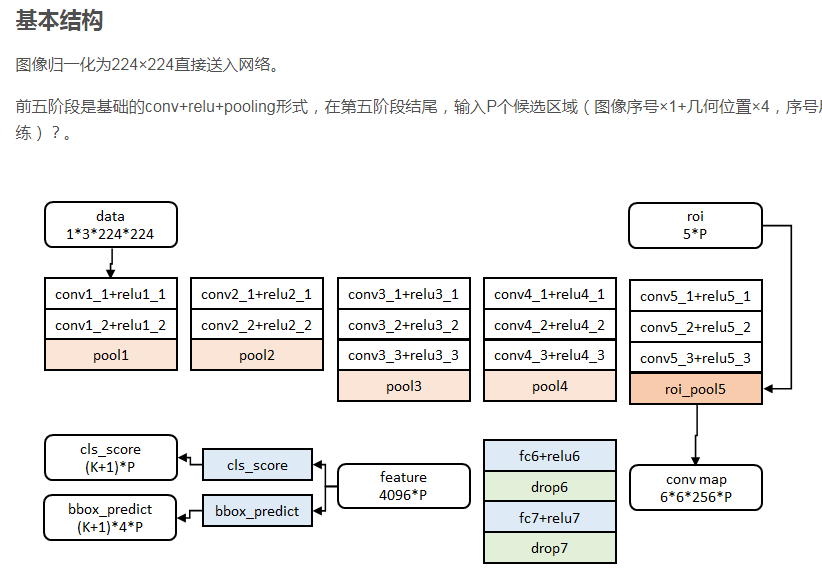
存在的问题：未解决多阶段训练的繁琐过程

1、测试流程



对于一张测试image，先运行selective search生成2000个region proposal，然后把image和它的region proposal输入Fast R-CNN网络；对每个测试region proposal，网络的前向传播会输出一个类别后验概率分布p和一系列预测的bounding box相对于该region proposal位置的偏移量坐标；然后根据概率p对每一类的bounding box进行非极大值抑制。

2、网络结构



* **Region of interest层(RoI pooling layer)**

         RoI层要做两件事：一是原image中的region proposal怎么在conv feature map上找到它的对应位置呢？二是要把这个region proposal对应的conv feature map中的特征用max-pooling转换成一个固定大小H×W的feature map。

         第一个问题：首先要明确，conv feature map的大小和原image是不一样的（比如VGG16，conv feature map可是14\*14\*512，原图227\*227\*3），所以肯定得有一个projection。由于原图生成conv feature map的过程是一系列卷积、下采样的操作，其中卷积主要是步长控制了输出大小。假设（x,y）是原图坐标，（xx,yy）是conv feature map坐标，S是CNN卷积操作中所有stride的乘积，给出映射公式如下：

（x,y） = （S\*xx,S\*yy）

反过来由原图求conv feature map坐标：xx = x/S+1，yy = y/S+1

         第二个问题：假设输入的feature RoI【也就是一个region proposal在conv feature map上对应的region】大小是h w，希望生成固定大小的H\*W的feature map。那么将feature RoI分成大小为（h/H）\*（w/W）的一共H\*W个子窗口，每个子窗口进行传统的max-pooling操作得到一个值【每个通道单独操作哒，所以输出通道数不变，和传统max-pooling一样一样的~】，所以就有H\*W个输出啦~

3、训练过程

第一步：微调

网络除去末尾部分如下图，在ImageNet上训练1000类分类器。结果参数作为相应层的初始化参数，其余参数随机初始化。最后一层改成21。

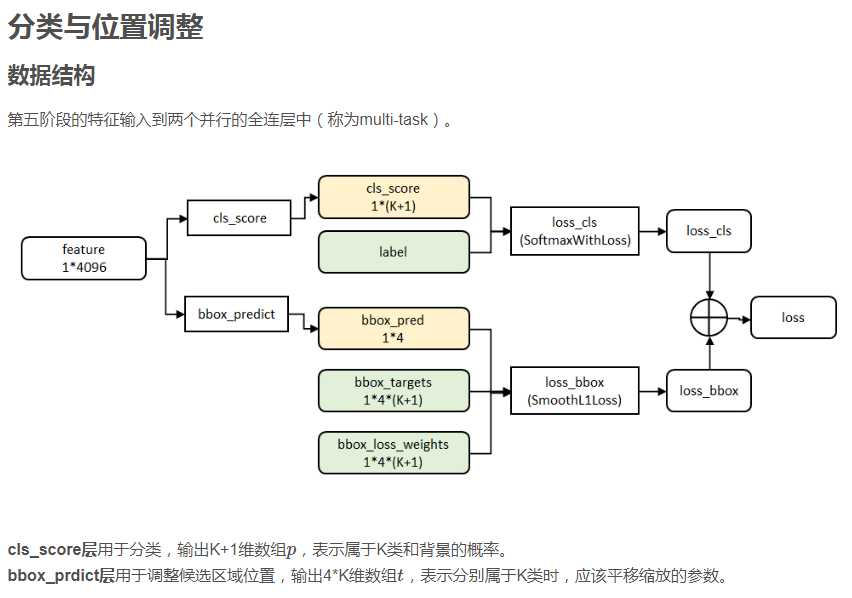
在调优训练时，每一个mini-batch中首先加入N张完整图片，而后加入从N张图片中选取的R个候选框。这R个候选框可以复用N张图片前5个阶段的网络特征。 实际选择N=2， R=128－> 每一个mini-batch中首先加入2张完整图片，而后加入从2张图片中选取的128个候选框。这128个候选框可以复用2张图片前5个阶段的网络特征。

正样本：占比25%， IOU>0.5

负样本：占比75%，0.1 < IOU < 0.5

随机梯度下降的mini-batch是层次化采样的：首先随机采样N=2张images，然后从每张图采样128/2=64个region proposals，因此就可以使得从同一张图片上采样的region proposals可以在前向和反向传播中共享计算。【你想，R-CNN和SPPnet的region proposal都不是来自同一张图的，那么每个region的conv feature map的计算就不能共享了！】。64个region proposals中有25%来自与ground-truth的IoU大于等于0.5的样本（R-CNN时正样本啦，label是1-K~），其余的是IoU在[0.1,0.5]之间的样本（label是0）。小于0.1的直觉上作为hard example mining了

第二步：回归

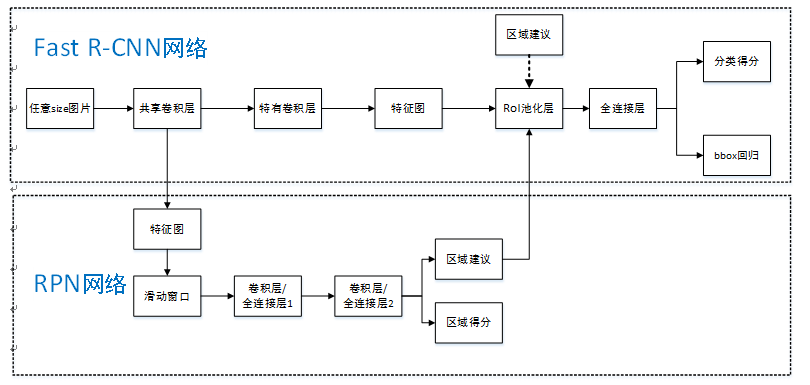


Fastercnn:

代码解析：<https://blog.csdn.net/g11d111/article/details/78823663>

原理阐述：<https://blog.csdn.net/williamyi96/article/details/77648047>

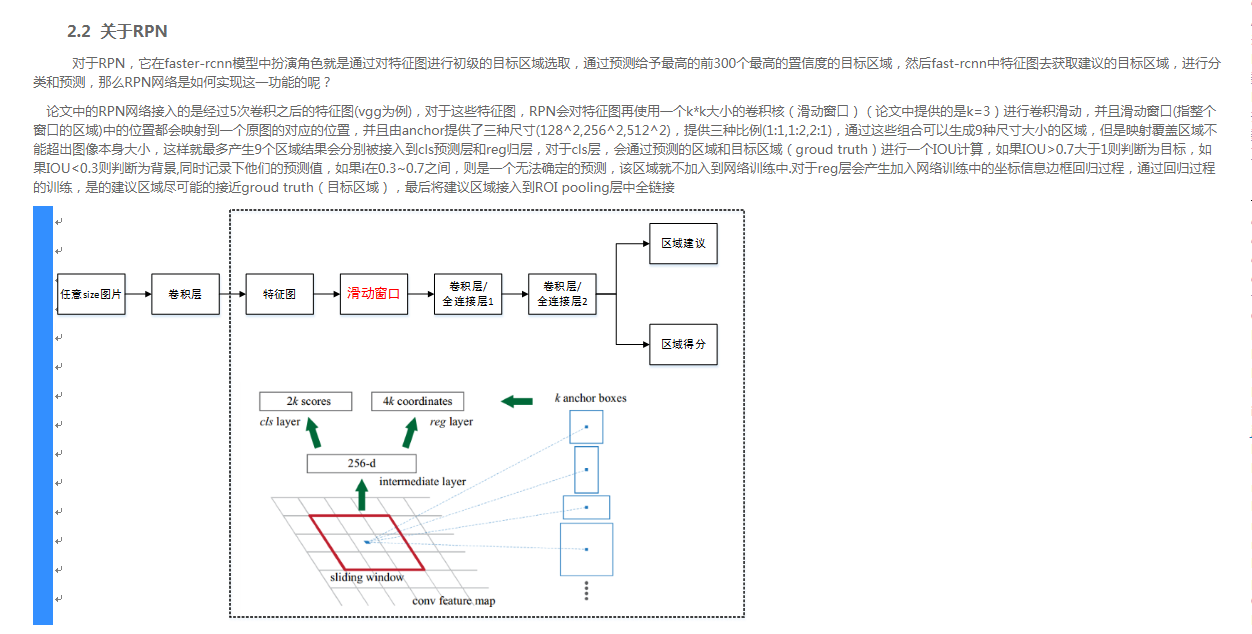
1、网络结构



2、训练过程

不管是RPN还是Fast R-CNN网络，其网络结构一部分来自于pre-trained model的卷积层（下文简称model），另一部分则是他们各自特有的结构（有卷积和FC，下文简称unique）。  
这里，我主要讲的就是训练过程。  
第一步：用model初始化RPN网络，然后训练RPN，在训练后，model以及RPN的unique会被更新。  
第二步：用model初始化Fast-rcnn网络，注意这个model和第一步一样。然后使用训练过的RPN来计算proposal，再将proposal给予Fast-rcnn网络。接着训练Fast-rcnn。训练完以后，model以及Fast-rcnn的unique都会被更新。  
说明：第一和第二步，用同样的model初始化RPN网络和Fast-rcnn网络，然后各自独立地进行训练，所以训练后，各自对model的更新一定是不一样的（论文中的different ways），因此就意味着model是不共享的（论文中的dont share convolution layers）。  
第三步：使用第二步训练完成的model来初始化RPN网络，第二次训练RPN网络。但是这次要把model锁定，训练过程中，model始终保持不变，而RPN的unique会被改变。  
说明：因为这一次的训练过程中，model始终保持和上一步Fast-rcnn中model一致，所以就称之为着共享。  
第四步：仍然保持第三步的model不变，初始化Fast-rcnn，第二次训练Fast-rcnn网络。其实就是对其unique进行finetune，训练完毕，得到一个文中所说的unified network。

3、RPN



<https://blog.csdn.net/sloanqin/article/details/51545125>

论文中的RPN网络接入的是经过5次卷积之后的特征图(vgg为例)，对于这些特征图，RPN会对特征图再使用一个k\*k大小的卷积核（滑动窗口）（论文中提供的是k=3）进行卷积滑动，并且滑动窗口(指整个窗口的区域)中的位置都会映射到一个原图的对应的位置，并且由anchor提供了三种尺寸(128^2,256^2,512^2)，提供三种比例(1:1,1:2,2:1)，通过这些组合可以生成9种尺寸大小的区域，但是映射覆盖区域不能超出图像本身大小，这样就最多产生9个区域结果会分别被接入到cls预测层和reg归层，对于cls层，会通过预测的区域和目标区域（groud truth）进行一个IOU计算，如果IOU>0.7大于1则判断为目标，如果IOU<0.3则判断为背景,同时记录下他们的预测值，如果i在0.3~0.7之间，则是一个无法确定的预测，该区域就不加入到网络训练中.对于reg层会产生加入网络训练中的坐标信息边框回归过程，通过回归过程的训练，是的建议区域尽可能的接近groud truth（目标区域），最后将建议区域接入到ROI pooling层中全链接.

5.31.2018

1、有关单例模式 –构造函数放在私有或者保护区域

**方法1：加锁的经典懒汉实现：**

class singleton

{

protected:

singleton()

{

pthread\_mutex\_init(&mutex);

}

private:

static singleton\* p;

public:

static pthread\_mutex\_t mutex;

static singleton\* initance();

};

pthread\_mutex\_t singleton::mutex;

singleton\* singleton::p = NULL;

singleton\* singleton::initance()

{

if (p == NULL)

{

pthread\_mutex\_lock(&mutex);

if (p == NULL)

p = new singleton();

pthread\_mutex\_unlock(&mutex);

}

return p;

}

#### 方法2：内部静态变量的懒汉实现

class singleton

{

protected:

singleton()

{

pthread\_mutex\_init(&mutex);

}

public:

static pthread\_mutex\_t mutex;

static singleton\* initance();

int a;

};

pthread\_mutex\_t singleton::mutex;

singleton\* singleton::initance()

{

pthread\_mutex\_lock(&mutex);

static singleton obj;

pthread\_mutex\_unlock(&mutex);

return &obj;

}

此方法也很容易实现，在instance函数里定义一个静态的实例，也可以保证拥有唯一实例，在返回时只需要返回其指针就可以了。推荐这种实现方法，真得非常简单。

#### 饿汉实现：本身就线程安全

class singleton

{

protected:

    singleton()

    {}

private:

    static singleton\* p;

public:

    static singleton\* initance();

};

singleton\* singleton::p = new singleton;

singleton\* singleton::initance()

{

    return p;

}

2、工厂模式

简单工厂：

class Product

10 {

11 public:

12 virtual void show() = 0;

13 };

14

15 class Product\_A : public Product

16 {

17 public:

18 void show()

19 {

20 cout << "Product\_A" << endl;

21 }

22 };

23

24 class Product\_B : public Product

25 {

26 public:

27 void show()

28 {

29 cout << "Product\_B" << endl;

30 }

31 };

32

33 class Factory

34 {

35 public:

36 Product\* Create(int i)

37 {

38 switch (i)

39 {

40 case 1:

41 return new Product\_A;

42 break;

43 case 2:

44 return new Product\_B;

45 break;

46 default:

47 break;

48 }

49 }

50 };

51

52 int main()

53 {

54 Factory \*factory = new Factory();

55 factory->Create(1)->show();

56 factory->Create(2)->show();

57 system("pause");

58 return 0;

59 }

工厂方法：

class Product

7 {

8 public:

9 virtual void show() = 0;

10 };

11

12 class Product\_A : public Product

13 {

14 public:

15 void show()

16 {

17 cout << "Product\_A" << endl;

18 }

19 };

20

21 class Product\_B : public Product

22 {

23 public:

24 void show()

25 {

26 cout << "Product\_B" << endl;

27 }

28 };

29

30 class Factory

31 {

32 public:

33 virtual Product\* create() = 0;

34 };

35

36 class Factory\_A : public Factory

37 {

38 public:

39 Product\* create()

40 {

41 return new Product\_A;

42 }

43 };

44

45 class Factory\_B : public Factory

46 {

47 public:

48 Product\* create()

49 {

50 return new Product\_B;

51 }

52 };

53

54 int main()

55 {

56 Factory\_A\* productA = new Factory\_A();

57 Factory\_B\* productB = new Factory\_B();

58

59 productA->create()->show();

60 productB->create()->show();

61 system("pause");

62 return 0;

63 }

和简单工厂区别在于，简单工厂违反了开放封闭原则，工厂方法解耦性更强。

抽象工厂：

//定义抽象类

5 class product1

6 {

7 public:

8 virtual void show() = 0;

9 };

10

11 //定义具体类

12 class product\_A1 :public product1

13 {

14 public:

15 void show(){ cout << "product A1" << endl; }

16 };

17

18 class product\_B1 :public product1

19 {

20 public:

21 void show(){ cout << "product B1" << endl; }

22 };

23

24 //定义抽象类

25 class product2

26 {

27 public:

28 virtual void show() = 0;

29 };

30

31 //定义具体类

32 class product\_A2 :public product2

33 {

34 public:

35 void show(){ cout << "product A2" << endl; }

36 };

37

38 class product\_B2 :public product2

39 {

40 public:

41 void show(){ cout << "product B2" << endl; }

42 };

43

44

45 class Factory

46 {

47 public:

48 virtual product1 \*creat1() = 0;

49 virtual product2 \*creat2() = 0;

50 };

51

52 class FactoryA

53 {

54 public:

55 product1 \*creat1(){ return new product\_A1(); }

56 product2 \*creat2(){ return new product\_A2(); }

57 };

58

59 class FactoryB

60 {

61 public:

62 product1 \*creat1(){ return new product\_B1(); }

63 product2 \*creat2(){ return new product\_B2(); }

64 };

65

66 int main()

67 {

68 FactoryA \*factoryA = new FactoryA();

69 factoryA->creat1()->show();

70 factoryA->creat2()->show();

71

72 FactoryB \*factoryB = new FactoryB();

73 factoryB->creat1()->show();

74 factoryB->creat2()->show();

75

76 return 0;

77 }

当你要强调一系列相关的产品对象的设计以便进行联合使用时。当你提供一个产品类库，而只想显示它们的接口而不是实现时使用。

6.19.2018

1、c++ int 和string的相互转换

#include <sstream>

#include <iostream>

string& int2str(const int &int\_temp)

{

std::stringstream stream;

std::string str\_temp;

stream << int\_temp;

stream >> str\_temp;

return str\_temp;

}

int str2int(string &str)

{

std::stringstream stream(str);

int int\_temp;

stream >> int\_temp;

return int\_temp;

}

在c语言里：

1. void str2int(int &int\_temp,const string &string\_temp)
2. {
3. int\_temp=atoi(string\_temp.c\_str());
4. }

char buffer[20]="10379cend$3";  
char \*stop;  
long int x = strtol(buffer, &stop, 2); // 字符串转长整型

注意和atoi的区别：

Atoi：如果我们输入的是前面是有效的数值字符，那么返回前面的数值，后面非法的不返还

Strtol：根据base来判断是否合法，只返回合法的数值， stop指向剩余非法的字符串。

1. void int2str(const int &int\_temp, string &string\_temp)
2. {
3. char s[12];             //设定12位对于存储32位int值足够
4. itoa(int\_temp,s,10);            //itoa函数亦可以实现，但是属于C中函数，在C++中推荐用流的方法
5. string\_temp=s;
6. }

itoa并不是一个标准的C函数，它是Windows特有的，如果要写跨平台的程序，请用sprintf 如：

char str[255];  
sprintf(str, "%x", 100); //将100转为16进制表示的字符串；

2、写文件

bool writeTxt(string txtpath, string line)

{

std::ofstream in;

in.open(txtpath.c\_str(), ios::app); //ios::app 如果有文件，则追加，若没有则创建

in << line << "\n";

in.close();//关闭文件

return true;

}

3、关于unsigned int int float double之间的转换

低级别的可以默认转化为高级别的，但是高级别的转低级别的必须强制转换

4、关于c++类里面的多线程、

在C++的类中，普通成员函数不能作为pthread\_create的线程函数，如果要作为pthread\_create中的线程函数，必须是static ! 在C语言中，我们使用pthread\_create创建线程，线程函数是一个全局函数，所以在C++中，创建线程时，也应该使用一个全局函数。static定义的类的成员函数就是一个全局函数。

类成员函数当作回调函数使用方法：

int pthread\_create( pthread\_t \*restrict tidp , const pthread\_attr\_t \*restrict attr , void\* (\*start\_rtn)(void\*) , void \*restrict arg );

　　第一个参数为指向线程标识符的指针。

　　第二个参数用来设置线程属性。

　　第三个参数是线程运行函数的起始地址，即回调函数。

最后一个参数是运行函数的参数。

实现该功能的好处：静态成员函数是不能使用非静态成员变量的（静态成员无this，它属于类的，只有实例能访问费静态成员函数），使用该功能后就可以使用类的非静态成员。

普通成员函数作为pthread\_create的线程函数有以下两方法：

**方法一：回调函数为普通函数，但在函数体内执行成员函数**

class MyClass

{

pthread\_t TID;

public:

void func()

{

//子线程执行代码

}

bool startThread()

{//启动子线程

int ret = pthread\_create( &TID , NULL , callback , this );

if( ret != 0 )

return false;

else

return true;

}

};

static void\* callback( void\* arg )

{//回调函数

((MyClass\*)arg)->func();调用成员函数

return NULL;

}

int main()

{

MyClass a;

a.startThread();

}

**方法二：回调函数为类内静态成员函数，在其内部调用成员函数**

**在方法一上稍作更改，把回调函数搬到类MyClass里，这样就保持了封装性。代码如下：**

class MyClass

{

static MyClass\* CurMy;//存储回调函数调用的对象

static void\* callback(void\*);//回调函数

pthread\_t TID;

void func()

{

//子线程执行代码

}

void setCurMy()

{//设置当前对象为回调函数调用的对象

CurMy = this;

}

public:

bool startThread()

{//启动子线程

setCurMy();

int ret = pthread\_create( &TID , NULL , MyClass::callback , NULL );

if( ret != 0 )

return false;

else

return true;

}

};

MyClass\* MyClass::CurMy = NULL;

void\* MyClass::callback(void\*)

{

CurMy->func();

return NULL;

}

int main()

{

MyClass a;

a.startThread();

}

6.25

1、使用变量要先初始化；

案例 float score；score += 0.7;

Std::cout << score<<std::endl; // score值不确定性

7.2

1、合并文件

cat 2007\_train.txt 2007\_val.txt > train.txt

2、unrar解压文件

unrar e test.rar 解压文件到当前目录 不保持原有状态

unrar x test.rar /path/to/extract 解压到指定目录，保持原有状态

tar –czvf 1.tar.gz xxx.txt xxx.txt

tar –xzvf 1.tar.gz //解压到当前目录

7.11 Python学习笔记

1、#同时打开两个文件

# with open('ha.log','r',encoding='utf-8') as f1 , open('ha1.log','a',encoding='utf-8') as f2:

# for line in f1:#循环f1 逐行获取f1的内容

# f2.write(line)#这样就将ha.log文件的内容 复制到f2中

2、进入正题：如何使用一个for循环同时循环多个变量呢？使用tuple。如下，同时循环i和j变量。

for (i, j) in [(1, 2), (2, 3), (4, 5)]:

print(i, j)

输出结果如下：

(1, 2)

(2, 3)

(4, 5)

使用zip

x = [1, 2, 3]

y = [-1, -2, -3]

for (i, j) in zip(x, y):

print(i + j)

输出结果：

0

0

0

2017.7.16

1、Python io模块

判断文件或目录是否存在：

os.path.exist(dir#file)

os.path.isdir(dir)

拷贝文件：

shutil.copyfile(oldfile, newfile)

shutil.copy(dir, new)

移动文件：

shutil.move(oldfile, newfile创建文件：

列出所有文件和目录列表：

Dirlist = Os.listdir(dir)

删除文件：os.remove(file) os.rmdir(dir)

拼接目录：os.path.join(xx, xx, xx)

抽样文件：testlist = random.sample(srclist, 100)

遍历列表：train = [pic for pic in trainlist if pic not in testlist]

重命名：os.rename(old, new)

写文件：

With open(file, ‘w’) as outfile:

Outfile.write(“xxx” + ‘\n’)

With open(file, ‘r’) as infile:

For line in infile:

#Todo line 是一行的数据

With open(file1, ‘r’) as f1, open(file2, ‘r’) as f2:

#todo

读写图片：cv2.imread(path) cv2.imwrite(todir, img)

去除字符串左右字符：

Str.strip(‘\n’)

Str.strip(‘\n\’’).split(‘ ’)

获取文件大小：

Os.path.getsize(file)

2、panda 数据处理

读数据：person\_info = pd.read\_csv('test.csv')

打印信息：

print (person\_info.shape) 矩阵行列

print (person\_info.columns)

print (person\_info.index)

print (person\_info.head())

print (person\_info.head()[“age”])

遍历元素：

For I in range(index):

Print(person\_info.loc[I, “age”])

新增列：

For I in range(index):

person\_info[I, “name”] = “xxx”

获取指定列：

person\_info[0:100, [“name”, “age”]]

转换数据类型：

person\_info =pd.DataFrame(data= person\_info, dtype=np.int8)

数据保存：

data.to\_csv("data.txt", sep=" ")

#不保存行索引和列索引

test\_data.to\_csv("valid.txt", sep=' ', index=False, header=False)

7.18、

1、单行列表的使用

X= [1, 2, 3, 4]

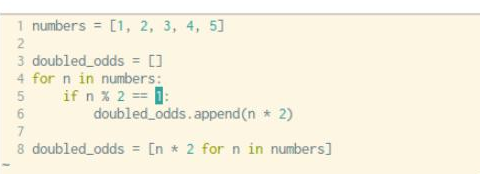
Out=[]

For item in x:

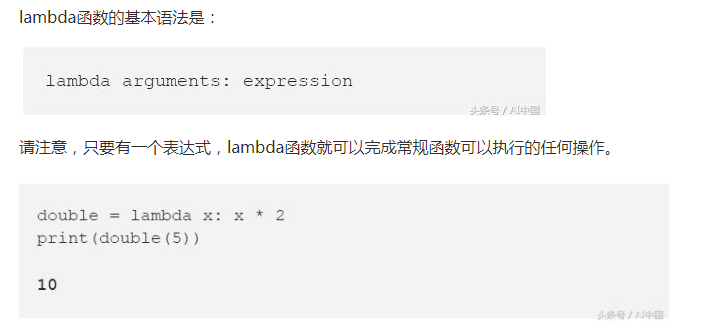
Out.append(item\*\*2)

转换成一行列表：

Out = [n\*\*2 for n in x]



2、lambda的使用



3、map和filter的使用

map()函数接收两个参数，一个是函数，一个是序列，map将传入的函数依次作用到序列的每个元素，并把结果作为新的list返回

L1 = map(lambda x:x\*\*2, [1, 2,3])

For I in l1:

Print(i) # 1 4 9

和map()类似，filter()也接收一个函数和一个序列。和map()不同的时，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

L1 = filter(lambda x:x>2, [1, 2,3])

For I in l1:

Print(i) # 3

L1 = map(lambda x,y:x+y, [1, 2,3], [1, 2, 3])

For I in l1:

Print(i) # 2 4 6

l3 = map(int, '123')

for I in l3:

print(i) # 1 2 3

reduce把一个函数作用在一个序列[x1, x2, x3…]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是

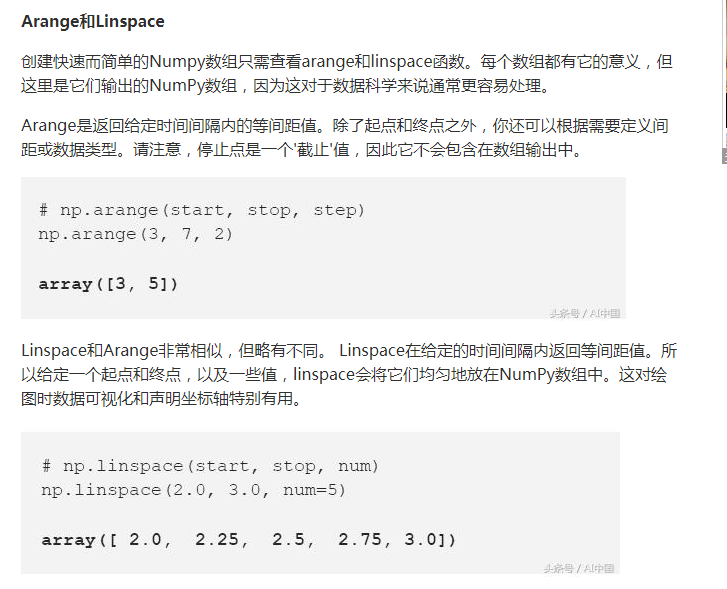
>>> def add(x, y):

... return x + y

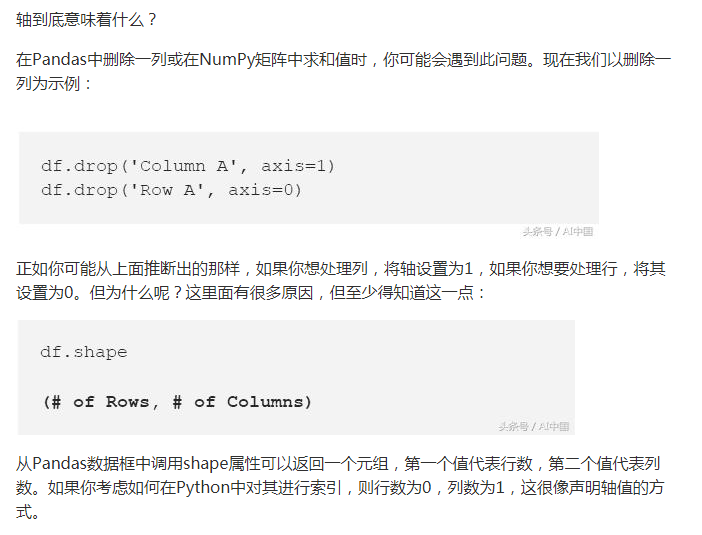
>>> reduce(add, [1, 3, 5, 7, 9])

25

4、arange() 和linepace（）



6、pandas



Python相关基础知识：

1、**深拷贝和浅拷贝的区别是什么**

深拷贝是将对象本身复制给另一个对象。这意味着如果对对象的副本进行更改时不会影响原对象。在 Python 中，我们使用 deepcopy（）函数进行深拷贝

Import copy

B = copy.deepcopy(a) # 深拷贝

C = copy.copy(a) # 浅拷贝

2、**列表和元祖有什么不同**

列表可变，元祖不可变

Mylist = [1, 2, 3]

Mylist[0] = 3; # true

Mytuple = (1, 2, 3)

Mytuple[0] = 2 # error

3、**Python 中的三元表达式**

与 C++不同, 在 Python 中我们不需要使用 ? 符号，而是使用如下语法：

如果 [expression] 为真, 则 [on true] 部分被执行。如果表示为假则 [on false] 部分被执行

[on true] if [expression]else [on false]

a, b = 1, 2

min = a if a > 0 else b

等于如下：

If a > 0:

min = a

else:

min = b

print(‘a’) if a<b else print(‘c’)

4、**什么是 Flask**

Flask 是一个使用 Python 编写的轻量级 Web 应用框架，使用 BSD 授权。其 WSGI 工具箱采用 Werkzeug，模板引擎则使用 Jinja2。除了 Werkzeug 和 Jinja2 以外几乎不依赖任何外部库。因为 Flask 被称为轻量级框架。

5、**如何在 Python 中管理内存**

opencv 用一个私有堆内存空间来放置所有对象和数据结构，我们无法访问它。由解释器来管理它。不过使用一些核心 API，我们可以访问一些 Python 内存管理工具控制内存分配。

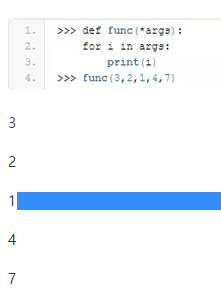
6、**什么是猴子补丁？**

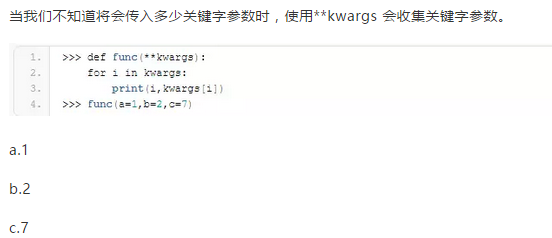
运行时修改类和模板

7、**解释一下 \*args 和 \*\*kwargs**

\*args 和 \*\*kwargs 主要用于函数定义。 你可以将不定数量的参数传递给一个函数。不定的意思是：预先并不知道, 函数使用者会传递多少个参数给你, 所以在这个场景下使用这两个关键字。其实并不是必须写成\*args 和\*\*kwargs。 \*(星号)才是必须的. 你也可以写成\*ar 和\*\*k. 而写成\*args 和\*\*kwargs只是一个通俗的命名约定

如果我们不知道将多少个参数传递给函数，比如当我们想传递一个列表或一个元组值时，就可以使用\*args。





8、**编程实现计算文件中的大写字母数**

With open(“1.txx”) as file:

Count = 0

For item in file.read():

If item.isupper():

Count +=1

Print(count)

9、**如何随机打乱列表中元素，要求不引用额外的内存空间**

Fro random import shuffle

Mylist=[1, 2,3]

Shuffle(mylist)

Print(mylist)

10、删除前置后置空格

‘ sssss’.lstrip()

‘sssss ’.rstrip()

‘ sss ’.trip()

35个python技巧

<http://www.iteye.com/news/28925-Python-Language-Tricks>

11、time模块

Time.time() #返回时间戳

Last\_time = time.time()

间隔：d = time.time() – last\_time

返回当前cpu时间：

Time.clock()

返回当前时间信息:

Time.localtime(xxx)

格式化时间：

#Tme.strftime(“%Y-%m-%d %H:%M:%S”, time.localtime())

time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())

输出：

2018-08-09 09:47:27

日期转时间戳：#先转时间组，再转时间戳

Date = ‘2017-08-08 12:00:00’

Time\_array = time.strptime(Date, “%Y-%m-%d %H:%M:%S”)

Timestamp = time.mktime(time\_array)

重新格式化日期：

Date = ‘2017-08-08 12:00:00’

Time\_array = time.strptime(Date, “%Y-%m-%d %H:%M:%S”)

New\_date = time.strftime(%Y%m%d %H%M%S,time\_array)

Datetime模块：

year=datetime.datetime.now().year

month=datetime.datetime.now().month

if not os.path.exists(logroot):

os.mkdir(logroot)

logfilepath=logroot+str(year)+'\_'+str(month)+'.log'

logging.basicConfig(filename=logfilepath,filemode='a',level=logging.DEBUG)

now=datetime.datetime.now()

logging.info(str(now)+' '+'nginx waiting...')

12、遍历列表

List = [1, 2, 3, 4]

法一：

For I in range(len(list))

Print(list[i])

法二： 最常用

For item in list:

Print(list.index[item] , item)

法三：

For I, val in enumerate(list):

Print(I, val)

7.27.2018

1、关于构造函数

一、默认构造函数的形式包括以下两种：

Obj() { xxx} 和Obj(int i=0, int j=2) {xxx}

调用方法如下：

Obj obj1; 和 obj1 = new Obj;

二、当类中没有写任何构造函数时，编译器提供默认构造函数；写了构造函数时，编译器不提供默认构造函数，需要自己写，防止Obj obj1;这样的错误；

三、类定义中，如果未提供自己的拷贝构造函数，则C++提供一个默认拷贝构造函数，就像没有提供构造函数时，C++提供默认构造函数一样。

2、explicit 的用法

class A

{

public:

explicit A(int a)

{

cout<<"创建类成功了!"<<endl;

}

};

int main()

{

A a=10; // 这里编译错误， 不能使用隐式转换，这里相当于直接调用A(10)

return 0;

}

注意：explicit

第一：此关键字表示只能显示调用构造函数，只能对用户自己定义的对象起作用，不对默认构造函数起作用，此关键字只能够修饰构造函数。而且构造函数的参数只能有一个。

第二：何时用？我们不希望自动类型转换的时候用，其实标准库好多构造函数都是explicit的

比如说vector <int> ivec(10); //这种定义看起来一目了然，可读性好；

第三：何事不用

当我们需要隐式转换的时候

比如说String类的一个构造函数，String(const char\*)；

定义成这样的好处,在需要隐式转化的时候编译器会自动地帮我们转换，标准库里面的String就是一个好的证明。

具体来说：

我们可以这样String str="helloworld";//直接调用构造函数

7.30

1、将用户oeasy添加到组docker

sudo usermod -aG docker oeasy

开机启动：$ sudo systemctl enable docker

//删除指定容器

$ docker rm container\_ID/name

//删除所有已经停止的容器

$ docker rm $(docker ps -a -q)

//删除指定镜像

$ docker rmi image\_name

8.28

1, 如果服务端的Socket比客户端的Socket先关闭，会导致客户端出现TIME\_WAIT状态，占用系统资源。

所以，必须等客户端先关闭Socket后，服务器端再关闭Socket才能避免TIME\_WAIT状态的出现。

2, 在linux下写socket的程序的时候，如果尝试send到一个disconnected socket上，就会让底层抛出一个SIGPIPE信号，可使用sigaction() 或者signal()注册信号处理。

3、判断客户端连接断开的方法：

法一：当recv()返回值小于等于0时，socket连接断开。但是还需要判断 errno是否等于 EINTR，如果errno == EINTR 则说明recv函数是由于程序接收到信号后返回的，socket连接还是正常的，不应close掉socket连接。

法二：心跳包

法三：若使用了select等系统函数，若远端断开，则select返回1，recv返回0则断开。其他注意事项同法一

法四：

int keepAlive = 1; // 开启keepalive属性

int keepIdle = 60; // 如该连接在60秒内没有任何数据往来,则进行探测

int keepInterval = 5; // 探测时发包的时间间隔为5 秒

int keepCount = 3; // 探测尝试的次数.如果第1次探测包就收到响应了,则后2次的不再发.

setsockopt(rs,SOL\_SOCKET,SO\_KEEPALIVE,(void\*)&keepAlive,sizeof(keepAlive));

setsockopt(rs, SOL\_TCP, TCP\_KEEPIDLE, (void\*)&keepIdle, sizeof(keepIdle));

setsockopt(rs, SOL\_TCP, TCP\_KEEPINTVL, (void \*)&keepInterval, sizeof(keepInterval));

setsockopt(rs, SOL\_TCP, TCP\_KEEPCNT, (void \*)&keepCount, sizeof(keepCount));

设置后，若断开，则在使用该socket读写时立即失败，并返回ETIMEDOUT错误

法五：struct tcp\_info info;

int len=sizeof(info);

getsockopt(sock, IPPROTO\_TCP, TCP\_INFO, &info, (socklen\_t \*)&len);

if((info.tcpi\_state==TCP\_ESTABLISHED)) 则说明未断开 else 断开

4、关于Python多线程

start(): 线程准备就绪，等待CPU调度

is/setDaemon(bool): 获取/设置是后台线程（默认前台线程（False））。（在start之前设置）

第一：当设置了后台运行时，主线程挂了，子线程无论成功与否都会退出；不设置后台运行时你，主线程结束后，子线程会继续运行。

第二： join()阻塞当前上下文环境的线程，直到调用此方法的线程终止或到达指定的timeout，即使设置了setDeamon（True）主线程依然要等待子线程结。

5、关于Lock和Rlock类

Lock（指令锁）是可用的最低级的同步指令。Lock处于锁定状态时，不被特定的线程拥有。Lock包含两种状态——锁定和非锁定，以及两个基本的方法。

可以认为Lock有一个锁定池，当线程请求锁定时，将线程至于池中，直到获得锁定后出池。池中的线程处于状态图中的同步阻塞状态。

RLock（可重入锁）是一个可以被同一个线程请求多次的同步指令。RLock使用了“拥有的线程”和“递归等级”的概念，处于锁定状态时，RLock被某个线程拥有。拥有RLock的线程可以再次调用acquire()，释放锁时需要调用release()相同次数。

可以认为RLock包含一个锁定池和一个初始值为0的计数器，每次成功调用 acquire()/release()，计数器将+1/-1，为0时锁处于未锁定状态

基本用法：

gl\_num = 0

lock = threading.RLock()

# 调用acquire([timeout])时，线程将一直阻塞，

# 直到获得锁定或者直到timeout秒后（timeout参数可选）。

# 返回是否获得锁。

def Func():

lock.acquire()

global gl\_num

gl\_num += 1

time.sleep(1)

print gl\_num

lock.release()

for i in range(10):

t = threading.Thread(target=Func)

t.start()

两者区别：

import threading

lock = threading.Lock() #Lock对象

lock.acquire()

lock.acquire() #产生了死锁。

lock.release()

lock.release()

print lock.acquire()

import threading

rLock = threading.RLock() #RLock对象

rLock.acquire()

rLock.acquire() #在同一线程内，程序不会堵塞。

rLock.release()

rLock.release()

6、关于python参数是值传递还是传引用传递

python不允许程序员选择采用传值还是传引用。Python参数传递采用的肯定是“传对象引用”的方式。这种方式相当于传值和传引用的一种综合。如果函数收到的是一个可变对象（比如字典或者列表）的引用，就能修改对象的原始值－－相当于通过“传引用”来传递对象。如果函数收到的是一个不可变对象（比如数字、字符或者元组）的引用，就不能直接修改原始对象－－相当于通过“传值'来传递对象。

7、多线程时只能使用一个核

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

8.30.2018

Pytorch 常用：

1、range()函数和arrange()函数

range(0,5) #生成一个range object,而不是[0,1,2,3,4]

c = [i for i in range(0,5)] #从0 开始到4，不包括5，默认的间隔为1

range(0, 1, 0.1) #range中的setp 不能使float err 报错

arange([start,] stop[, step,], dtype=None)根据start与stop指定的范围以及step设定的步长，生成一个 ndarray

np.arange(3) =》 array([0, 1, 2])

arange(0, 1, 0.1) true

array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])

2、torch.index\_select()的示例

x = torch.randn(3, 4) #生成二行三列的tensor

indices = torch.LongTensor([0, 2])

y = torch.index\_select(x, 0, indices) # 获取x中第0,2行

z = torch.index\_select(x, 1, indices) # 获取x中第0,2列

3、模块导入

from importlib import import\_module

module = import\_module('model.' + args.model.lower())

self.model = module.make\_model(args).to(self.device)

9.03

1、关于有序无序字典

在python中，dict是哈希存储的，所以默认无序的，但是模块collections中的子模块OrderedDict有序。

一、举例

def test():

d = {}

d['a'] = 'A'

d['b'] = 'B'

d['c'] = 'C'

for k, v in d.items():

print(k, v)

无序

d1 = OrderedDict()

d1['a'] = 'A'

d1['b'] = 'B'

d1['c'] = 'c'

d1['1'] = '1'

d1['2'] = '2'

for k, v in d1.items():

print(k, v)

结果有序

二、OrderedDict对象的字典对象，如果其顺序不同那么Python也会把他们当做是两个不同的对象

三、

dd = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}

#按key排序

kd = collections.OrderedDict(sorted(dd.items(), key=lambda t: t[0]))

print kd

#按照value排序

vd = collections.OrderedDict(sorted(dd.items(),key=lambda t:t[1]))

print vd

#输出

OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

9.10

1、format用法

print '{0},{1}'.format('chuhao',20)

print '{},{}'.format('chuhao',20)

print '{1},{0},{1}'.format('chuhao',20)

print '{name},{age}'.format(age=18,name='chuhao')

通过映射 list

a\_list = ['chuhao',20,'china']

print 'my name is {0[0]},from {0[2]},age is {0[1]}'.format(a\_list)

#my name is chuhao,from china,age is 20

#通过映射 dict

b\_dict = {'name':'chuhao','age':20,'province':'shanxi'}

print 'my name is {name}, age is {age},from {province}'.format(\*\*b\_dict)

#my name is chuhao, age is 20,from shanxi

9.11

x = np.array([[1,2,3],[4,5,6]])

y = np.expand\_dims(x,axis=0) # 表示在在第0维上增加维度，用法类似于torch里面的unsqueeze()

print("y.shape: ",y.shape) # (1, 2, 3)

1.14.2019:字符编码笔记：ASCII，Unicode 和 UTF-8

计算机内部，所有信息最终都是一个二进制值。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从00000000到11111111

ASCII: ASCII 码一共规定了128个字符的编码，比如空格SPACE是32（二进制00100000），大写的字母A是65（二进制01000001）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的一位统一规定为0

Unicode: 如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是 Unicode，就像它的名字都表示的，这是一种所有符号的编码.

Unicode 当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字严。具体的符号对应表，可以查询unicode.org，或者专门的汉字对应表.

UTF-8: 互联网的普及，强烈要求出现一种统一的编码方式。UTF-8 就是在互联网上使用最广的一种 Unicode 的实现方式. UTF-8 是 Unicode 的实现方式之一.

UTF-8 的编码规则很简单，只有二条:

1）对于单字节的符号，字节的第一位设为0，后面7位为这个符号的 Unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

2）对于n字节的符号（n > 1），第一个字节的前n位都设为1，第n + 1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的 Unicode 码。

--------------------------------------------------------------------------------------------------------------

Unicode符号范围 | UTF-8编码方式

(十六进制) | （二进制）

----------------------+---------------------------------------------

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

--------------------------------------------------------------------------------------------------------------

1.21

1、创建链接符 ln -s test1.txt test.txt 命令： ln –s 源文件（已存在的） 目标文件