Table of Contents

gcc五分钟系列

简介

- 通过大量简单的例子介绍 `gcc` 的基本用法
- 每节内容简短，控制在5分钟以内
- 讲的是一些最基础的用法

关于版本

写本系列文章时，`gcc` 的版本是4.6.2。 本系列文章参照此版本进行介绍。

如果 `gcc` 的最新版本的参数或用法发生更改， 本系列文章可能不会及时更新。 所以请您在正式使用前 务必 确认您的 `gcc` 版本并参考 `gcc` 的说明文档。

在线阅读与电子书下载

gitbook 主页: https://www.gitbook.com/book/lexdene/gcc-five-minutes

第一节：前言

为啥要介绍**GCC**呢？

其实这个事源于光哥问我的一个问题: “C语言中宏定义影响的范围有多大？”

现在，IDE的易用与普及，使coder们越来越远离命令行的编译方式。IDE确实简单方便，这个毋庸置疑。

不过IDE造成的问题是，很多原理性的东西大家可能并不了解。我写本系列文章的目的，就是为了让大家能够更深入地了解一些原理性的东西。

本人水平有限，错误在所难免，希望大家看见以后能帮忙指出。

预备知识

本系列文章不介绍基础知识，所以在阅读本系列文章前需要有以下知识：

1. C/C++。本系列文章只介绍gcc与C/C++相关的用法。
2. shell（bash）。因为是在命令行下编译，所以你至少得会命令行啊。不用会的太多，基本的几个命令知道就行。

顺便一提

1. GNU。这个不需要我介绍吧？
2. 理查德·马修·斯托曼。GCC的最初作者，自由软件运动的精神领袖，GNU计划及自由软件基金会（FSF）的创立者。

好了，下面简单介绍一下我们的**GCC**吧。

GCC刚开始的名字叫GNU C语言编译器 `GNU C Compiler` ， 是大神 理查德·马修·斯托曼 在1985年写的。 那时候斯托曼只是为了写个好用的C语言编译器。

后来呢，GCC不止能编译C语言，开始能够编译各种语言， 包括： `C++` , `Fortran` , `Pascal` , `Objective-C` , `Java` , `Ada` 和 `Go` 等。 这个时候呢，它的名字也改了。改成叫GNU编译器套装 `GNU Compiler Collection` ，但是缩写没变，还是GCC。

不过，我们这个系列文章使用是它本来的概念：`GNU C Compiler`。因为我们止以C／C++为例介绍GCC的使用。

五分钟到了，下课。

第二节：编译我们的Hello World

干点啥事都得从Hello World开始，这是谁规定的？

代码：

```
#include <iostream>
using namespace std;
int main(){
    cout<<"Hello World"<<endl;
    return 0;
}
```

将上面代码保存为hello_world.cpp，然后打开终端，输入以下内容、回车：

```
g++ hello_world.cpp
./a.out
```

解释：

1. g++是gcc套装中专用于编译C++代码的编译器
2. g++后面参数，没有选项，只有源文件：
    i. g++会将它直接从源代码编译成可执行文件
    ii. 无论源文件名是什么，可执行文件的文件名肯定是a.out
3. ./a.out执行编译得到的可执行文件

命令行输出：

```
Hello World
```

今天就到这里

第三节：指定输出文件名

上一节，我们将hello_world.cpp编译成了a.out。

可是，a.out这个名字看起来这么讨厌呢..... 能不能整个别的名字呢？

答案是肯定的，方法是-o选项。

文档：（来自gcc man手册）

```
-o file
       Place output in file file.  This applies regardless to whatever
       sort of output is being produced, whether it be an executable file,
       an object file, an assembler file or preprocessed C code.

       If -o is not specified, the default is to put an executable file in
       a.out, the object file for source.suffix in source.o, its assembler
       file in source.s, a precompiled header file in source.suffix.gch,
       and all preprocessed C source on standard output.
```

简单解释一下：

```
-o选项后面得加个文件名。它能够让gcc输入到指定的文件中，无论是可执行文件、目标文件、汇编文件还是预编译C代码。
```

所以呢，将第二节的命令改成：

```
g++ -o hello_world hello_world.cpp
./hello_world
```

命令行输出同样是：

```
Hello World
```

第四节：多文件工程

多文件

在写一些大型的C/C++程序的时候， 我们通常会把代码归门别类，放在多个文件中。 这样既方便查找，又方便阅读。

那么，gcc能否处理多个文件的情况呢？

答案是能，不过有一些需要注意的地方。

举个例子

比如我们现在有两个文件，sum.cpp和main.cpp。

其中sum.cpp中定义了一个sum函数， main.cpp使用这个函数计算两个整数的和。

sum.cpp代码如下：

```
int sum(int a, int b)
{
    return a + b;
}
```

main.cpp代码如下：

```
#include <iostream>

using namespace std;

int sum(int a, int b);

int main(){
    cout << sum(4, 7) << endl;
    return 0;
}
```

怎么编译呢？

错误方法：

```
g++ -o main main.cpp
```

如果您这样编译的话，会得到错误：

```
/tmp/xxxx.o: In function `main':
main.cpp:(.text+0xf): undefined reference to `sum(int, int)'
collect2: ld returned 1 exit status
```

很明显，从错误信息的第三行可以看出来（ld），是链接时错误。

造成这个错误的原因也很简单，main.cpp里面没有sum函数的定义，所以链接器（ld）会报怨，找不到sum函数的定义。

目标文件

C/C++是靠目标文件（扩展名为.o的文件）来实现多文件工程的。

首先、编译器将源文件编译成目标文件。 然后、链接器将多个目标文件链接成一个可执行文件。

那么，我们用gcc该如何做呢？

第一步，将源文件编译成目标文件

gcc使用-c参数来指定， 我要编译的是目标文件， 不是直接生成可执行文件。

这里用到了我们上一节课讲的-o参数， 它的含义，相信您一定记得。

```
g++ -c -o main.o main.cpp
g++ -c -o sum.o sum.cpp
```

第二步，将所有的目标文件链接起来，生成可执行文件

这没有新的参数， 和最初的编译方法一样。

```
g++ -o main main.o sum.o
```

执行一下试试？

```
./main
```

输出：

```
11
```

本节完

第五节：稍微多说点目标文件

上一节，我们讲到目标文件。那么目标文件究竟是什么？为什么通过目标文件能够实现多文件编程？

说一下我个人的理解：

编译

编译器将源文件编译成目标文件。它会进行基本的类型检查。

以这个代码为例说明：

```
class Foo{
public:
    int data()const{
        return this->_data;
    }
    void set_data(data){
        this->_data = data;
    }
private:
    int _data;
}

int bar(Foo* foo, int data){
    int origin_data = foo->data();
    foo->set_data(data);
    return origin_data;
}
```

这个代码编译是没有问题的。（当然链接的时候会报错，说没有main函数）。

但是如果只声明Foo而没有写它的定义会怎样呢？

```
class Foo;
int bar(Foo* foo, int data){
    int origin_data = foo->data();
    foo->set_data(data);
    return origin_data;
}
```

编译器会报错，说找不到 `类型Foo` 的定义。

为什么一定要有 `Foo` 的定义呢？因为编译器需要知道：

1. Foo有没有data的set_data这两个方法。
2. data和set_data这两个方法的参数类型和返回值类型是什么。

有了这两点，编译器才能对 `bar` 这个函数的函数体做类型检查。

但是如果 `bar` 这个函数没有函数体呢？或者说 `bar` 也只是一个声明，而不是定义，会怎么样呢？

```
class Foo;
int bar(Foo* foo, int data);
```

这个代码就不会有编译错误。

实际工程中这种做法很常用。比如把类型的声明和函数的声明写在 `a.h` 里面，把实现写在 `a.cpp` 里面。

如果 `b.cpp` 需要使用 `a.h` 中的类型和函数，它只需要 `#include "a.h"` 一下就好了。

由于 `a.h` 只包含声明，这个文件会比较短。这样会加快 `b.cpp` 的编译速度。

这样的技术叫做 `前向声明` 。

链接

前面的例子，`b.cpp` 会编译成目标文件 `b.o` ，但是由于 `b.cpp` 没有 `bar` 函数的定义，所以 `b.o` 里面会记录着：

> 我需要 `bar` 的定义。

那么同时，`a.cpp` 也会编译成目标文件 `a.o` ，它里面包含 `bar` 函数的定义，所以 `a.o` 里面会记录着：

> 我提供 `bar` 的定义。

最终在链接的时候，将 `a.o` 和 `b.o` 链接起来，它们就幸福地生活在了一起。

第六节：编译期符号检查

上一节，我们说了从源文件到目标文件的编译过程。 这一节，我想讨论一下编译期符号检查的问题。

比如，第四节的例子中， main.cpp文件中只有sum()函数的声明、而没有定义。

> 相信您能够分辨C/C++中声明和定义的区别。

可是将main.cpp编译成main.o的过程中，没有报任何错误。

这里，我们看下面这个例子：

我们只声明了一个结构体而没有定义这个结构体， 然后定义一个这个结构体的变量。 代码：

```
struct Poo;
int main()
{
    Poo a;
    return 0;
}
```

只编译而不链接：

```
g++ -c struct1.cpp
```

这个时候它会报错：

```
struct1.cpp: In function ‘int main()’:
struct1.cpp:4:6: error: aggregate ‘Poo a’ has incomplete type and cannot be defined
```

这说明，只有声明没有定义的结构体不能够定义变量。

把代码稍微改一下，定义这个结构体的一个指针：

```
struct Poo;
int main()
{
    Poo *a;
    return 0;
}
```

只编译不链接，没有任何问题。

这说明，虽然不能定义变量，但是可以定义指针。

再把代码稍微改一下，实例化一下：

```
struct Poo;
int main()
{
    Poo *a=new Poo;
    return 0;
}
```

这个时候也会报错：

```
struct1.cpp: In function ‘int main()’:
struct1.cpp:4:13: error: invalid use of incomplete type ‘struct Poo’
```

总结:

编译期会不会报错。（链接期会不会报错是另外一回事了，这里不讨论）

1. 编译期不会报错的几种情况：
    i. 调用一个只有声明没有定义的函数。
    ii. 定义一个只有声明没有定义的类型的指针。
    iii. 使用一个只有声明没有定义的变量。
2. 编译期会报错的几种情况：
    i. 定义一个只有声明没有定义的类型的变量。
    ii. 实例化一个只有声明没有定义的类型。

变量如何只声明不定义？ `int a;` 就已经是定义了。好吧，留个悬念，大家可以自己研究一下。

第七节：声明与定义的区别

解释了好多遍，还是有同学不了解在C/C++中声明和定义的区别。 这里简单介绍一下。

如何声明和定义

变量

声明：

```
extern int a;
```

定义： int a;

函数

声明：

```
void fun();
```

定义：

```
void fun(){
}
```

结构体／类

声明：

```
class Bar;
```

定义：

```
class Bar{
public:
    Bar();
    int foo();
};
```

这里虽然 `Bar::Bar` 和 `Bar::foo` 都只有声明，没有定义。但是 `class Bar` 已经定义完了。

声明和定义的本质区别是什么？

声明没有为符号分配存储空间、定义会为符号分配存储空间。

变量

```
extern int a;
```

这句话是告诉编译器，变量a在别的地方有了，所以不需要在这为它分配存储空间了。

```
int a;
```

这句话是告诉编译器，我需要创建一个变量，请为它分配存储空间。

函数

```
void fun();
```

函数体包含若干语句，编译后会产生若干指令。声明是告诉编译器，这个函数的指令保存在别的地方了。

```
void fun(){
}
```

这里会为这个函数分配存储空间、保存函数的指令。

结构体／类

```
class Bar;
```

同样不分配存储空间。

```
class Bar{};
```

一个结构体／类在定义的时候会产生它的各种指针，其中最重要的是它的函数地址表。

结构体内部的成员变量会在它初始化的时候才会分配内存。

为什么会有编译期错误

为什么一个没有定义的结构体/类，在定义它的变量的时候会报告编译期错误？

因为有的结构体/类不能定义它的变量。（比如它的构造函数是private的。在 `设计模式` 中，C++常用这种技术来实现单例模式。）

编译器只有在看到它的定义，才能知道它是否可以定义变量。

结尾

感觉解释的不是很清楚，不过我能力也就如此了。

本节完。

第八节：前言

谁规定前言必须在最前面了？ 前言说点啥呢？想到啥就说啥吧。

看书比例不超过20%

感觉对于一个工程师来说，重要的是实践。 我个人在学习C++的过程中，看书的时间绝对不超过20%，剩下的时间全部用来实践了。 我现在（2011年8月）积累下来的C++代码应该快要到2W行了（POJ上150道AC，有1W行。DGP项目有1W行）。

写本系列，很多东西都是我曾经犯下的错误。有句话说，实践越多，犯错误的机会越多。犯错误越多，学习的越扎实。

书上讲了很多东西，放下书就都忘光光了。可是犯一次错误会让自己记住好久。

文档是万能的

gcc的man手册那么长（感觉比长城都长。） 不过less命令有增量搜索功能（我严重怀疑man命令是调用less来查看man手册内容的，证据是界面和很多命令都是相同的）。 需要某个选项的文档，直接在man里面搜索。

另外，man手册是英文的，所以我觉得，想做大牛，英语必须要好。 不过多好才算好呢？我有一个同学，英语六级，给她find命令的man文档，她完全看不懂。 我英语四级还不到，可是我能看懂。 所以，英语多好才算好呢？ 我真的不知道怎么回答。

博客是认识大牛最直接的方法

我真正提高，学到了大量在书本和实践中学不到的东西， 是在今年（2011年）年初，从一位学长的博客中。

后来，又陆续通过rss订阅了许多大牛的博客， 学到了非常非常多的从未接触过的东西，开阔眼界。

朋友的鼓励和支持是我坚持下去的唯一动力

其实我感觉，我C++和gcc的水平也大概只有2W行相当。 而且对于软件工程、设计模式完全不懂。

不过我希望能够将我知道的东西发布出来，以期得到大牛们的指导。 由于工作原因，我可能无法每天坚持更新， 不过，朋友的鼓励和支持是我坚持下去的唯一动力。 只要有人看，我就愿意继续写下去。

先说这么多，以后如果想到别的东西我会再发一篇《前言续》。

第九节：编译过程

接下来几节，我会讲一些编译期相关的优化选项及预编译期相关的优化选项。 至于汇编期和链接期.....我就完全不懂了

我不懂汇编语言。 链接期也涉及了很多我完全不懂的问题。

那么，编译期是什么意思？ 预编译期又是什么意思？

这个涉及到编译的整个过程。 我记得前面某节好像提到过这个话题，这里详细的讲一下。

从源代码（**xxx.cpp**）生成可执行文件（**a.out**）一共分为四个阶段：

1. 预编译阶段

此时编译器会处理源代码中所有的预编译指令。 预编译指定非常有特点，全部以“**#**”开头

想想，以“**#**”开头的命令有哪些

不同的命令有不同的处理方法，#include命令的处理方法就是赤裸裸的复制粘贴。 将#include后面的文件的内容赤裸裸地复制粘贴到#include命令所在的位置。

#define命令分为带参宏和不带参宏。 #define命令的处理方法，学名叫宏展开。 其实不带参宏的处理方法就是赤裸裸的字符串替换。

此阶段完成后，会产生一篇完全不包含预编译指令的代码。

使用gcc的-E选项可以查看预编译结果：

```
g++ -E xxx.cpp
```

但是这个命令不会把处理结果保存在文件中，而是放在标准输出中。

例子：见本文最后例一。

2. 汇编阶段

此时编译器会将预处理过的代码进行汇编。

使用gcc的-S选项可以查看汇编结果：

```
g++ -S xxx.cpp
```

之后会在当前目录下产生一个xxx.s的文件，里面保存的是汇编代码。

汇编我懂的不多，就不帖了。

3. 编译阶段

此时编译器会将汇编代码编译成目标文件。

使用gcc的-c选项可以生成目标文件：

```
g++ -c xxx.s
```

也可以从源代码直接生成目标文件：

```
g++ -c xxx.cpp
```

gcc会通过扩展名自动判断处理的是汇编代码还是C++代码。

到此，（狭义上的）编译器已经完成它的全部工作。

4. 链接阶段

此时已经没有编译器的事情了。链接工作交由链接器来处理。 链接器会将多个目标文件链接成可执行文件。

我们可以通过gcc来进行链接，但是实际上，gcc还是调用ld命令来完成链接工作的。

```
g++ xx1.o xx2.o
```

本节完

接下来几节， 计划先讲编译阶段的pipe选项、O选项和W选项。 因为编译阶段事情比较少。

然后讲预编译阶段的I选项和D选项。

例子

例一

main.cpp

```
#include "sum.h"
#define SUCCESS 0
int main(){
    int c=sum(1,2);
    return SUCCESS;
}
```

sum.h

```
#ifndef SUM_H
#define SUM_H

int sum(int a,int b);

#endif // SUM_H
```

预编译

```
g++ -E main.cpp
```

输出

```
# 1 "main.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.cpp"
# 1 "sum.h" 1

int sum(int a,int b);
# 2 "main.cpp" 2

int main(){
    int c=sum(1,2);
    return 0;
}
```

第十节：编译期优化选项（一）——pipe

中间文件

上一节讲到，从源代码生成最终的可执行文件需要四个步骤，并且还会产生中间文件。

可是我们在对一个源文件编译的时候，直接执行g++ xxx.cpp能够得到可执行文件a.out。 并没有中间文件啊！中间文件在哪里？

答案是，在/tmp目录下。想看吗？跟着我做。

1. 在终端中执行g++ xxx.cpp
2. 在另外一个终端中执行 `ls /tmp/cc* 2>/dev/null`

看见什么了？ 什么也没有啊！ 说明你太慢了。

你需要在第一个命令完成前，执行第二个命令，否则什么也看不见。 你大概只有不到0.1秒的时间。

你需要在0.1秒的时间内完成

- 切换终端
- 输入命令
- 回车

这些事情，你做得到吗？

所以， 还是写一个脚本来看吧。

```
#!/bin/bash
g++ main.cpp &
sleep 0.05
ls --color=auto /tmp/cc*
```

在我的电脑上，时间是0.05的时候可以看到如下结果：

```
/tmp/cc9CD8ah.o   /tmp/ccj9uXNd.s
```

可以看到，有.s汇编文件，.o目录文件。

所以，实际上gcc将中间文件放在了/tmp目录下，并且在编译完成后会将其删除。

编译优化

不得不说， C/C++的编译速度真是慢的出奇。

如果你曾经有编译一百万行以上C++代码的经历的话， 你会知道， 电脑开机一周只是为了编译，其实不算是过分的事情。

但是如果你认为，gcc已经放弃治疗了， 那你就太误解gcc了。

C/C++中有很多技术用来提高编译速度， 比如`前向声明`。

而且gcc也提供了一些选项来加快编译速度。

回到本文一开始的话题来。 如果编译过程中产生很多临时文件的话， 那么磁盘IO将会成为编译速度的重要瓶颈。

我们需要将上一步的结果交给下一步处理，有没有什么比较快的方法？

如果您了解linux的话，会立即想到一个牛X闪闪的东西：管道。

将上一步编译的结果通过管道传递给下一步，这不需要IO操作，全部在内存中完成， 效率上会有非常大的提高。

gcc提供了这个功能，方法是使用-pipe选项。

```
g++ -pipe main.cpp
```

下面是gcc的man手册中关于-pipe选项的解释：

```
-pipe
    Use pipes rather than temporary files for communication between the
    various stages of compilation.  This fails to work on some systems
    where the assembler is unable to read from a pipe; but the GNU
    assembler has no trouble.
```

本节完

第十一节：编译期优化选项（二）——O（上）

一篇很简单的代码

```
int createNum();
void putNum(int a);

int sum(int a,int b)
{
    return a+b;
}

int main()
{
    int x=createNum();
    int y=createNum();
    int z=sum(x,y);
    putNum(z);
    return 0;
}
```

我们来查看一下它的汇编代码：

```
g++ -s main.cpp
```

得到一个main.s。打开这个文件，截取其中main函数的一小段，加上一些注释，如下：

```
call    _Z9createNumv   ;调用createNum()函数
movl    %eax, -4(%rbp)  ;将返回值压栈
call    _Z9createNumv   ;再调用createNum()函数
movl    %eax, -8(%rbp)  ;将返回值压栈
movl    -8(%rbp), %edx  ;将栈顶数据放在寄存器edx中
movl    -4(%rbp), %eax  ;同上，放在寄存器eax中
movl    %edx, %esi      ;将寄存器edx中的数据作为sum()函数的第一个参数
movl    %eax, %edi      ;将寄存器eax中的数据作为sum()函数的第二个参数
call    _Z3sumii        ;调用sum()函数
movl    %eax, -12(%rbp) ;将返回值压栈
movl    -12(%rbp), %eax ;将栈顶数据放在寄存器eax中
movl    %eax, %edi      ;将寄存器eax中的数据作为putNum()函数的第一个参数
call    _Z6putNumi      ;调用putNum()函数
```

初步优化

大家觉得，是不是很麻烦？

每次调用一个函数之后，先压栈，然后又转到寄存器中，这很浪费时间。

gcc会这么笨吗？当然不会。 gcc的-O选项（注意，是大写。回想一下，小写-o选项是干什么的？前面讲过）就是用来处理编译期优化的。

我们重新产生一下汇编代码，但是使用-O选项。

```
g++ -O -s main.cpp -o main.O1.s
```

现在打开main.O1.s文件， 看，里面函数的返回值没有经过入栈和出栈的过程， 直接传入下一个函数的参数。这样减少了六条汇编代码。

进一步优化

可是细想想，sum()函数有点多余。 它实际上只是做了一个加法， 但是我们仍然需要调用这个函数来完成它的功能。

gcc会这么笨吗？ 当然不会。

-O选项还可以加数字，表示优化的级别。

没有数字默认是1，最大可以加到3。 优化级别越高，产生的代码的执行效率就越高。

我们用级别2试一下：

```
g++ -O2 -s main.cpp -o main.O2.s
```

现在打开main.O2.s， 大家可以看到，调用sum()函数的代码都不见了， 取而代之的是一条加法指令来完成两个整数的相加。

-O3的效果我就不试了。而且，如果不加-O选项，优化级别就是0。

编译效率

既然-O后面的的数字越大，产生的代码越优化，那么为什么不直接用-O3？

原因是，优化的级别越高， 虽然最后生成的代码的执行效率就会越高， 但是编译的过程花费的时间就会越长。

在执行效率和编译时间之间， 需要做出一个权衡。

gcc没有擅自做这个决定， 而是把决定的权力留给了用户。

在linux的世界里，有这样一个观点： 不限制用户的发挥， 但是让用户为自己的行为负责。

所以linux的很多软件都有很多选项。

这样有好处也有坏处， 好处是，用户拥有更多的权力， 坏处就是，彩笔们一看到这么多选项，瞬间就傻了。

本节完

第十三节：编译期优化选项（四）——W

优秀的程序员不应该忽略任何的warning

先看第一段代码：

```
int fun(){
}
int main(){
    fun();
}
```

很简单，对吧？

有错误吗？事实上是没有的。

编译一下：`g++ return-type.cpp`。 也没有任何问题。

可是事实上，fun函数没有return语句。

这很有可能是程序员的疏忽造成的。 而且，很多严重的问题可能就是由一些很幼小的疏忽造成的。

我们希望，gcc在遇见这类问题的时候，能够给我们一个提示。

-W选项

还好，gcc提供了一个-W选项。 我们使用这样的命令来编译：

```
g++ -Wreturn-type return-type.cpp
```

它仍然能够正常编译，生成可执行文件， 但是，它会输出一句warning：

```
return-type.cpp: In function ‘int fun()’:
return-type.cpp:3:1: warning: no return statement in function returning non-void
```

不错吧？

解释一下，-W是打开警告输出，后面接的是警告的种类。 gcc将警告分为好多种（将近一百种）。 return-type只是检查返回值类型。

再来一个

```
//uninitialized.cpp
int fun(){
    int a;
    return a;
}
int main(){
    fun();
}
```

按照正常方式编译：g++ uninitialized.cpp。没有任何问题。

我们打开uninitialized种类的警告，这样编译：

```
g++ -Wuninitialized uninitialized.cpp
```

它输出的warning是这样的：

```
uninitialized.cpp: In function ‘int fun()’:
uninitialized.cpp:4:12: warning: ‘a’ is used uninitialized in this function
```

开启全部警告

但是，种类那么多，一个一个加会不会很麻烦？

哈哈！gcc的-W选项有个种类叫all。 猜是什么意思？

打开所有种类的警告。

很方便吧？

强制不许忽略警告

上面只是在可能出现问题的地方输出了警告， 但是警告并不影响正常编译。

有没有办法要求必须处理掉所有的警告呢？

有。

gcc有个选项，叫-Werror。 它会把所有的警告当成错误进行处理。

本节完。

第十四节：预编译期选项（一）——|

include命令

相信大家对C/C++中的 `#include` 指令有所了解， 相信您能够区别引号和尖括号的作用。

简单来说:

- 引号的作用：先在当前目录下搜索文件，然后在系统目录下搜索文件
- 尖括号的作用：仅在系统目录下搜索文件。

但是，如果大家做过Qt的开发，就会发现：

- 包含一个Qt库的头文件时，需要用尖括号即可。
- Qt库的头文件并不在系统目录下。

例如，我用的是ubuntu linux maverick，使用apt-get安装qt4。

以 `<QObject>` 这个头文件为例，它的存放路径是 `/usr/include/qt4/QtCore/QObject` 。

那么Qt是如何做到的呢？

-I选项

我们可以打开qmake生成的Makefile，看到Qt在编译参数的变量中加上了这样一句：

```
-I/usr/include/qt4/QtCore
```

这个-I是gcc的一个选项， 它的作用是把一个路径加入到系统路径， 这样当使用include指令时， 就可以用尖括号来指定文件。

那么怎么指定多个路径呢？多写几遍就可以了！

```
-I/usr/share/qt4/mkspecs/linux-g++ -I../../../dgp/othello/client -I/usr/include/qt4/Qt
Core -I/usr/include/qt4/QtNetwork -I/usr/include/qt4/QtGui -I/usr/include/qt4 -I../../
../dgp/Dlut-Game-Platform/api/include -I../../../dgp/roommodel/include/client -I../../
../dgp/roommodel/include/common -I. -I. -I../../../dgp/othello/client -I.
```

可以用绝对路径、相对路径。 甚至可以把当前路径加入到系统路径中， 这样就可以用尖括号来指定当前目录下的一个文件了。

扩展阅读：Qt的pro文件中的 qt += network 的作用

假如我要使用Qt中的 `QTcpServer` 这个类， 我需要写上

```
#include <QtNetwork/QTcpServer>
```

但是如果我在.pro文件中加上

```
qt += network
```

之后，我的代码中只需要写

```
#include <QTcpServer>
```

就可以了。

原理是因为在加上

```
qt += network
```

之后，Makefile中就会多出

```
-I/usr/include/qt4/QtNetwork
```

这样一句。

本节完。

第十五节：预编译期选项（二）：D

debug版和release版

一般我们在开发项目时，都会做两个版本，一个debug版，一个release版。

通常我们在debug版本中会加入调试输出，方便查找BUG。

而在release版本中，删除这些输出，以提高执行速度并减少可执行文件的大小。

实现的方式是通过宏。

以一个简单的函数为例，这个函数仅仅输出参数的值。但在debug版本中，输出一些多余的信息，比如函数名。

例1

```
#define __DEBUG__

void output(int a){
#ifdef __DEBUG__
    cerr<<"this is debug infor : "<<__FUNCTION__<<' '<<a<<endl;
#endif
    cout<<a<<endl;
}
```

如此，将#define **DEBUG**注释掉就是release版本。

但是在一个大工程中，我们将所有的#define **DEBUG**注释掉是一件非常痛苦的事情。

例2

gcc提供了一种简单的方法，可以在编译时添加一些宏定义。使用-D选项。

代码如下：

```
void output(int a){
#ifdef __DEBUG__
    cerr<<"this is debug infor : "<<__FUNCTION__<<' '<<a<<endl;
#endif
    cout<<a<<endl;
}
```

我们用 `g++ main.cpp -D__DEBUG__` 编译出来的就是debug版本，用 `g++ main.cpp` 编译出来的就是release版本。

扩展：

linux内核的部分编译配置选项就是通过-D来设置的，如此，我们不需要修改源代码， 只需要修改参数就可以编译出不同配置的linux内核。

本节完。

第十七节：库的使用（一）：使用

c与zip

半年以前的事情了。有一次，我在一个C项目中需要读写zip压缩包文件。很幸运地，我发现了libzip库。[http://packages.debian.org/squeeze/libzip-dev] 然后，一切问题就都解决了。咋解决的？听我慢慢道来。

库

编程中，有一件很神奇的事情，叫做 复用 。通常，我们将一个功能封装在一个函数中，目的是为了以后遇见相同功能的时候，不需要把这个函数再写一遍。可是，函数级别的复用有时候会显得力不从心。因此，出现了 库 技术。库的优点有很多，例如：

- 它是编译好的二进制文件，体积小，不需要用户自己编译。
- 发布简单。使用方便。

通常，库根据链接方式的不同，分为两种：静态链接和动态链接。这里我们先介绍动态链接的方法。

使用libzip

我写了一个简单的程序，zipdemo，它的目的是输出一个zip包里的文件列表。使用libzip，一共分为三步。

1.安装libzip。

```
sudo apt-get install libzip-dev
```

这里需要注意一下，软件包libzip包含的是运行时库，而我们需要的是开发库，也就是libzip-dev。

2.编写源代码，使用libzip中的函数。

2.1.首先要包含头文件。

```
#include <zip.h>
```

2.2.使用libzip中的函数。

函数的列表及每个函数的详细介绍可以在libzip的man手册中找到。

3.编译链接。

3.1.由于zip.h头文件是放在/usr/include下的，因此不需要显式地指定include目录。编译的步骤没有任何区别。

```
gcc -c -o zipdemo.o zipdemo.c
```

3.2.链接时，需要指定链接的动态库的名字。这里我们使用-l参数。

```
gcc -lzip -o zipdemo zipdemo.o
```

zipdemo源代码及Makefile可以参见本项目git仓库。

本节完

> 附. 其实 *debian* 的软件仓库里面基本上已经涵盖了我们需要的各类编程库，尤其以C库居多。每次当我需要某功能的时候，我都会先在 *debian* 的软件仓库中搜索一下，有没有已经提供的库。搜索地址：[http://www.debian.org/distrib/packages]