

# Ch. 3 代码怎么写

2017.3

## § 3.0 如何学习一门编程语言

- Print “Hello World”
- 数据类型
- 基础逻辑 if elseif else while for switch
- 特性, OOP相关
  - 语言特性, C++都有哪些?
- 面向平台进行必要的开发
  - 方言
  - API

# 规则的重要性

\$ \_ \$\$ \_\_\_\_\_ \$ \_ \$\$\$ \_\_\_\_\_  
\_\_\_\_\_ \$ \_\_\_\_\_ \$\$\$  
\_\_\_\_\_ \$ \_\_\_\_\_  
\_\_\_\_\_ \_\_\_\_\_  
\_\_\_\_\_ \_\_\_\_\_  
\_\_\_\_\_ \$\$ \_\_\_\_\_ "ok" \_\_\_\_\_  
\_\_\_\_\_ \_\_\_\_\_



# 目录

- § 3.1 标识符相关
- § 3.2 注释相关
- § 3.3 逻辑相关
- § 3.4 特殊问题
- § 3.5 异常相关

# 标识符

## C++

- 标识符由字母、数字、下划线 “\_”组成（\$ 也可以）。
- 不能把C++关键字作为标识符。
- 标识符长度限制32字符。
- 标识符对大小写敏感。
- 首字符只能是字母或下划线，不能是数字。

## JAVA

- 标识符由字母、数字、下划线 “\_”、美元符号 “\$”组成，第一个字符不能是数字。
- 不能把java关键字和保留字作为标识符。
- 标识符没有长度限制。
- 标识符对大小写敏感。



# 标识符

请猜一下？

```
public void foo(){}

```

```
public void foo(){
```

```
public void fOO(){
```

```
class MyMain
{
public:
    void f00(){}
    void foo(){}
    void f00(){}
};
```

```
16  
17 public void foo() {}  
18 public void f00() {}  
19 public void fOO() {}
```

- x/X p/P u/U w/W z/Z v/V k/K(大小写相近的字母容易用错，不要同时出现，同时是指作用域范围内)
- o/O/ (i和p中间的字母) 数字0 l(L的小写)和 数字1 容易写错

# 标识符

```
/**
 * 你是否能正确应用?
 * 猜吧。
 */
private int v_1; // 数字 一
private int v_l; // 字母 L
```

```
12  /**
13     * 你是否能正确应用?
14     * 猜吧。
15     */
16     private int v_1; // 数字 一
17     private int v_l; // 字母 L
18
```

```
/**
 * 出错的风险高
 * @param x
 */
public void setX(int x)
{
    this.x = x;
}
private int x;
private int X;
```

# 标识符

- 命名要能反映出其作用
- 命名要有呼应
  - Init()/clear()
  - CreateInstance/DestroyInstance
  - MIN/MAX
- 建议采用驼峰/帕斯卡/匈牙利命名法



# Camel-Case

```
int sumScore;
```

```
long getSum();
```

```
int studentGrade;
```

```
static const MAX_VALUE = 12;
```

```
long GetSum();
```

# Hungarian Notation

global -> g\_

member -> m\_

static -> s\_

pointer -> p

char\*/wchar\_t\* -> psz

char[]/wchar\_t[] -> sz



# 都是什么类型？

- m\_pszName;
- HWND hWnd; CWnd\* pWnd;
- HDLG hDlg; CDialog\* pDlg;
- HDC hDC; CDC\* pDC;
- HGDIOBJ hGdiObj; CGdiObject\* pGdiObj;
- HPEN hPen; CPen\* pPen;

# 优劣比较

- 哪种更好？
- 各自的缺点是什么？



```
9      class A
10     {
11     public:
12         A(int myid) { myID = myid;}
13
14         void setName(int mi)
15         {
16             int myid = mi;
17             myid = myID;
18
19             m_nMyID = mi;
20             this->myID = myid;
21         }
22     private:
23         int myID;
24         int m_nMyID;
25     };
26
```

```
37     template <typename T>
38     class C
39     {
40     private:
41         std::map<std::vector<std::string>, std::vector<T> m_idkStupid;
42     };
```



# 规范太多怎么办？

- 尽可能和开发环境的API保持一致
- 一个项目按统一的方式
- 有必要的说明

# 变量和常量

- 变量尽量有意义，不要使用“鬼变量”

```
int x; int xxx; int xxxxx_2;
```

- 常量值尽可能描述清楚意义

```
if(1500 < x)    // 1500请加以注释，如果需要调试、改动或者经常使用，  
               // 那么建议定义常量
```

- 用英文(或英文缩写)定义，不易翻译的，用拼音，并加以注释

```
int hp;          // hp就是HP  
byte char2byte;  // 字符转字节后的结果  
string value4use; // value for use  
string qigong    // 气功/奇功
```



# 变量和常量

常量的定义方法

关键字 `const static` / `final`

宏 （不利于调试）



可以写一个类维护所有的常量，从配置文件中读取

# 变量和函数（方法）

```
8 class Sample
9 {
10     /**
11      * 定义时需要计算
12      */
13     void fun()
14     {
15         int cube = 20 * 30 * 40;    // 立方体的体积
16         int cb = 24000;            // 20 * 30 * 40
17
18         // do something
19     }
20
21     /**
22      * 频繁的计算
23      */
24     void fun_do()
25     {
26         for (int i = 0; i < 10000; i++)
27         {
28             int cube = 20 * 30 * 40;
29
30             // do something
31         }
32     }
33
34     /**
35      * 常量用起来方便
36      */
37     static const int MAX_CUBE = 20 * 30 * 40;
38 };
```

```
public class PrioritySample {
    /**
     * 定义时需要计算
     */
    public void fun()
    {
        int cube = 20 * 30 * 40;    // 立方体的体积
        int cb = 24000;            // 20 * 30 * 40
    }

    // do something

    /**
     * 频繁的计算
     */
    public void fun_do()
    {
        for (int i = 0; i < 10000; i++)
        {
            int cube = 20 * 30 * 40;
        }

        // do something
    }

    /**
     * 常量用起来方便
     */
    final private int MAX_CUBE = 20 * 30 * 40;
}
```



# 函数

- 默认参数问题
- 重载/重写问题 (overload/override)
- 避免大文件，大类，大函数体

```
44     bool full = false;
45     void setFullScreen(bool v = true)
46     {
47         full = v;
48     }
49     void test()
50     {
51         setFullScreen();
52     }
```

# 函数

- 重载方法过多不是好的设计
  - getConnection (String url)
  - getConnection (String ip,String path)
  - getConnection (String ip,String port,String path)
  - getConnection (String ip,String port,String path,String user,String passwd)
  - getConnection(ConnectParam param):把参数封装在类中

# 函数

- 方法被频繁重写不是好设计
  - 如果父类方法频繁被子类重写，考虑把该方法变成抽象方法
  - 父类方法应只负责通用的、较少变化的逻辑



```
2 /**
3  * 接口方式
4  * @author devuser
5  *
6  */
7 public interface ConstInterface
8 {
9     public int PORT = 28993;
10    public String IP = "172.12.10.12";
11    public boolean test = true;
12 }
13
```

```
1 /**
2  * 常量类
3  * @author devuser
4  *
5  */
6 public class ConstClass
7 {
8     private ConstClass() {}
9
10    public int PORT = 28993;
11    public String IP = "172.12.10.12";
12    public boolean test = true;
13 }
```

# 目录

- § 3.1 标识符相关
- § 3.2 注释相关
- § 3.3 逻辑相关
- § 3.4 特殊问题
- § 3.5 异常相关

# 注释

- 使代码让人容易读懂
  - 自己可能一段时间后会忘
  - 别人看你的代码的时候，不至于给你打电话

```
/**
```

```
 * function 注释
```

```
 */
```

```
// hp就是HP    ! 无意义的注释
```



# 注释

```
////////////////////////////////////  
////////////////////////////////////  
// //////////众里寻她千百度的注释//////////  
////////////////////////////////////  
////////////////////////////////////
```



```
/******  
/* 不好维护的注释*****  
/* 真不好维护*****  
/* 新版本多加了几个字怎么办 **就是在这个地*/  
/* 方要怎么办呢? *****  
/******
```

# 注释

- 单行注释（短注释）

//.....

- 多行注释（块注释）

/\*

\*/



# 注释

- 源码文件头注释

```
/**
```

```
*Class: classname
```

```
*Package: packagename
```

```
*
```

```
*ver      date      author
```

```
*1.0    2015-03-09  authorname
```

```
*Copyright(c) 2015,LEDO All Rights Reserved
```

```
*/
```



# 注释

- 方法的注释

```
/**
 *
 * @param url      获取method的url地址
 * @param keepCookie 是否保存cookie
 * @return String 方法名
 * @throws Exception
 */
private String getMethod(String url, boolean keepCookie) throws Exception
```



如果代码有改动，一定要记得修改相关的注释！“过期”的注释比没有注释危害更大！


# 目录

- § 3.1 标识符相关
- § 3.2 注释相关
- § 3.3 逻辑相关
- § 3.4 特殊问题
- § 3.5 异常相关




# 简单语句

```
18 int value()  
19 {  
20     if(a)  
21         return 2;  
22  
23     return 3;  
24 }  
25  
26 int val()  
27 {  
28     return a ? 2 : 3;  
29 }
```



```
void fun()  
{  
    if(NULL == frd)  
        return;  
  
    frd->doSomething();  
    frd->close();  
}  
  
void f()  
{  
    if(NULL != frd)  
    {  
        frd->doSomething();  
        frd->close();  
    }  
}
```



# 写简化的逻辑

- 尽可能方便地加入新的判断
- 尽可能方便地加入新的逻辑
- 结构化的逻辑最容易理解、最容易维护

# 条件

```
/**
 * 特别难看难维护的逻辑
 * 超过3层“{}”
 * @return
 */
public boolean badEnough ()
{
    if(a)
    {
        if(b)
        {
            if(c)
                return true;
            else
                return false;
        }
        else
            return false;
    }
    else
        return false;
}
```

```
public boolean ok ()
{
    if (a && b && c)
        return true;
    else
        return false;
}

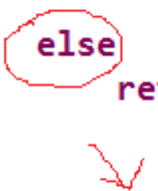
public boolean sook()
{
    return a && b && c;
}
```



# 条件

```
public boolean ok ()
{
    if (a && b && c)
        return true;
    else
        return false;
}

public boolean sook()
{
    return a && b && c;
}
```



```
/**
 * 分条式代码，代码多，但简洁容易维护
 * @return
 */
public boolean recommend ()
{
    if(! a)
        return false;

    if(! b)
        return false;

    if(! c)
        return false;

    return true;
}
```

# 条件

```
/**
 * 不要在条件语句里直接调用方法
 * 因为你都不知道下面三个方法哪些被执行过
 * @return
 */
public boolean init(){
    return initConf() && initData() && initConnections();
}
```

# 循环

```
/**
 * 超过 3 层的“{}”
 * 逻辑难懂，不利于维护
 */
public void fool()
{
    if (a)
    {
        for (int i = 0; i < CONST; i++)
        {
            if (b)
            {
                if (c)
                {
                    doSomething();
                } else
                {
                    doSomething();
                }
            }
        }
    }
}
```

```
/**
 * 按照原则整理的易维护版本
 */
public void waste()
{
    if (!a)
        return;

    for (int i = 0; i < CONST; i++)
    {
        if (!b)
            continue;

        if (!c)
            continue;

        doSomething();
    }
}
```



# 循环

```
/**
 * 循环次数减少，效率改进（注：要按实际逻辑整理化简）
 */
public void perform()
{
    if (!a)
        return;

    if (! b)
        return;

    if (! c)
        return;

    for (int i = 0; i < CONST; i++)
    {
        doSomething();
    }
}
```

# 循环

```
/**
 * 可运行程序
 */
public void justok()
{
    while( val )
    {
        doSomething();
    }
}
```

```
/**
 * 推荐做法，做循环检测
 */
public void best()
{
    int count = 0;
    while( val )
    {
        doSomething();

        if(count ++ > MAX_LOOP )
        {
            error("too many loops");
            break;
        }
    }
}
```

# Switch

```
/*  
 * 写switch的时候一定不要忘了break和default  
 */  
switch (key) {  
  
    case value1:  
        break;  
  
    case value2:  
        break;  
  
    case value3:  
        break;  
  
    default:  
        break;  
}
```



# 目录

- § 3.1 标识符相关
- § 3.2 注释相关
- § 3.3 逻辑相关
- § 3.4 特殊问题
- § 3.5 异常相关

# 运算符优先级

- 1、先算哪部分？
- 2、会不会受到写法的影响？

0x07	+	0x01	<<	3
0x07	+	0x01<< 3		
0x07+0x01		<< 3		

即便能记得住，也还是用“（）”处理一下吧



# 数值运算

- 基本数的值范围
- 涉及到精度问题，请减少float、double等类型的使用
- 位操作
- 容易产生的溢出



# JAVA 的一些数据类型

- Byte [-128, 127] 1个字节  $[-2^7, 2^7-1]$
- short [-32768, 32767] 2个字节  $[-2^{15}, 2^{15}-1]$
- int [-2147483648, 2147483647] 4个字节  $[-2^{31}, 2^{31}-1]$
- long [-9223372036854774808, 9223372036854774807] 8个字节  $[-2^{63}, 2^{63}-1]$

# C++ 的一些数据类型

VC 编译器 32位系统

- bool 1字节
- char 1字节 [-128, 127] [0, 255](unsigned)
- wchar\_t 2字节
- short 2字节 [-32768, 32767] [0, 65535](unsigned)
- int 4字节 [-2147483648, 2147483647] [0, 4294967295](unsigned)
- long 4字节
- float 4字节
- double 8字节
- long long 8字节(\_\_int64)



# 容易产生的溢出

- 数值运算
- 移位操作（<<、>>、>>>）
- 类型转换会导致数值截断（大类型转小类型）



# 一些计算示例

```
public void somefun()
{
    short s = 32767;
    s ++;
    // do something

    s << 18;
    s >>> 8;
}
```

```
short getTripleValue(short s)
{
    return s * 3;
}
```

```
short long2short(long l)
{
    return (short) l;
}
```

# 越界

- 数组
- 数组越界问题
- 迭代器越界

# 递归

- 递归的终结条件
- 递归是否浪费栈空间（问题规模预期）
- 合理递归



# 迭代器问题 java

```
/**
 * 删除list中的偶数操作
 */
for(Integer i:list)
{
    if(i%2 != 0)
        continue;

    list.remove(i);
}
```

# 迭代器问题 C++

```
156  /**
157   * 删除map中key为偶数的示例
158   * 会导致 迭代器的失效
159   */
160  void erase_even()
161  {
162      std::map<int, std::string> is_map;
163
164      std::map<int, std::string>::iterator it = is_map.begin();
165      for(; it != is_map.end(); it++)
166      {
167          if(0 == (it->first % 2))
168              is_map.erase(it);
169      }
170  }
```

# 迭代器问题 C++

```
147 void erase_even()  
148 {  
149     std::map<int, std::string> is_map;  
150  
151     std::map<int, std::string>::iterator it = is_map.begin();  
152     for(; it != is_map.end(); )  
153     {  
154         if(0 == (it->first % 2))  
155             is_map.erase(it ++);  
156         // it = is_map.erase(it); //VC 下OK  
157         // gcc 下erase是返回类型是 void  
158         // 不同的STL实现会有差别  
159         else  
160             it ++;  
161     }  
162 }
```



# 字符串格式化

- `format`
- `sprintf`、`snprintf`等函数
- 最关键的`%s,%d,%u`等等，不好进行覆盖性测试，请尽量避免使用，用`stringstream`、字符串拼接等方式替代
- 实现`toString()`

# 关于NULL/null

- 对空指针、对象、引用对象都要进行有效的判断和处理
- 不要等着系统的异常处理来处理这些可预见性的问题



# 关于调试

- 提交的代码中，请让调试的信息失效
- 请不要使用printf/print/println/cout<<这类输出到标准输出设备的方式输出调试信息
- 宏覆盖调试
- 便于调试的方法



# 目录

- § 3.1 标识符相关
- § 3.2 注释相关
- § 3.3 逻辑相关
- § 3.4 特殊问题
- § 3.5 异常相关

# 异常处理

- 尽量采用三段式结构：
  - **try**{  
    //做你要做的事情  
} **catch**(你能处理的Exception e){  
    //处理你能解决的问题，不能解决的，向上抛出  
} **finally** {  
    //不管问题有没有发生, 都要处理的工作  
}

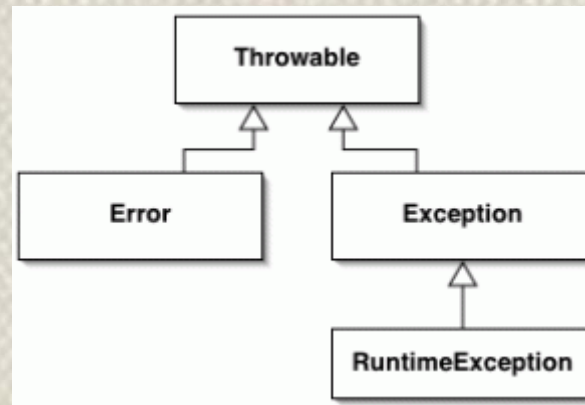
- 请不要忽视处理抛出的异常

```
5-  /**
6    * 合理使用 异常处理，对异常一定要进行处理，而不是置之不理，
7    * 打出日志也可以，出现问题可以定位
8    */
9-  void blind()
10   {
11     try
12     {
13         // dosomething
14     }
15     catch(Exception e)
16     {
17         // 什么也不做，不推荐的做法！
18     }
19 }
```



# 异常处理原则

- 具体明确（不要总是抛出捕获Exception e）
- 尽早抛出
- 延迟捕获



```
try {
    String temp = new String();
    url = new URL("http://localhost:8080/myServlet/welcome");
    httpURLConn = (HttpURLConnection) url.openConnection();
    httpURLConn.setDoOutput(true);
    httpURLConn.setRequestMethod("GET");
    httpURLConn.setIfModifiedSince(999999999);
    httpURLConn.setRequestProperty("Referer", "http://localhost:80");
    httpURLConn.setRequestProperty("User-Agent", "test");
    httpURLConn.connect();
    InputStream in = httpURLConn.getInputStream();
    BufferedReader bd = new BufferedReader(new InputStreamReader(in));
    while ((temp = bd.readLine()) != null) {
        System.out.println(temp);
    }
} catch (MalformedURLException e) {
    e.printStackTrace();
} catch (ProtocolException e) {
    e.printStackTrace();
    throw e;
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} finally {
    if (httpURLConn != null) {
        httpURLConn.disconnect();
    }
}
```

```
35- /**
36-  * 捕获过早，处理不好
37-  */
38- public void bad()
39- {
40-     RetValue ret = null;
41-     try{
42-         ret = getRetValue();
43-     }catch(Exception e)
44-     {
45-         e.printStackTrace();
46-     }
47-
48-     ret.doSomething();
49-     ret.close();
50- }
```

```
52- /**
53-  * 延迟捕获
54-  */
55- public void ok()
56- {
57-     RetValue ret = null;
58-     try
59-     {
60-         ret = getRetValue();
61-         ret.doSomething();
62-     }catch(Exception e)
63-     {
64-         e.printStackTrace();
65-     }
66-     finally
67-     {
68-         ret.close();
69-     }
70- }
```



- 别把异常当正常！ 对可预料到的问题（错误），请不要用“抛出异常然后捕获”的方式进行处理（不要把抛出的异常对象，当作返回值使用）

## C++ 示例

```
65 /**
66  * 示例：查找最小值，list 是空的怎么处理
67  * 返回错误码的方式
68  */
69 int getMinValue(std::list<int> & list)
70 {
71     int i = 0;
72
73     // list是空的，怎么处理
74     if(list.empty())
75     {
76         return -1; // 这不是个好主意
77         // return NULL; // 一样的，同上，NULL不就是0嘛
78     }
79
80     // i = somevalue;
81     return i;
82 }
```

```
84 /**
85  * 示例：查找最小值，list 是空的怎么处理
86  * 抛出异常的方式
87  */
88 int getMinVal(std::list<int> & list)
89 {
90     int i = 0;
91     int e = -1; // 这是个异常
92
93     if(list.empty())
94     {
95         throw e; // 这样处理，有效
96     }
97
98     // i = somevalue;
99     return i;
100 }
```

## JAVA 示例

```
/**
 * 示例：获取最小值，如果list是null,怎么处理
 * 返回错误码 的方式
 * @param list
 * @return
 */
public int getMinValue(java.util.List<Integer> list)
{
    int i = 0;

    // list是空的，怎么处理
    if(null == list)
    {
        return -1;    // 这不是个好主意
    }

    // i = somevalue;
    return i;
}
```

```
/**
 * 示例：获取最小值，调用该函数时捕获异常
 * 抛出异常的方式
 * @param list
 * @return
 */
public int getMin(java.util.List<Integer> list)
{
    // list是空的，怎么处理
    if(null == list)
    {
        // java 里，这种方式不是很推荐
        throw new IllegalArgumentException("list is null");
    }

    int i = 0;
    // i = somevalue;
    return i;
}
```



## JAVA 示例

```
/**
 * 示例：获取最小值
 * 对象方式，装箱类型
 * @param list
 * @return
 */
public Integer getMinInteger(java.util.List<Integer> list)
{
    Integer i = null;

    // list是空的，怎么处理
    if(null == list)
    {
        return null;
    }

    // i = somevalue
    return i;
}
```

Q & A