编码方法 目录

什么是软件编码？

- Programming ≠ Coding
- 一种观点
  - 软件编码是将软件设计模型机械地转换成源程序代码，这是一种低水平的、缺乏创造性的工作。
  - 软件程序员是所谓的“软件蓝领”。
- 问题
  - 你是否认同这种观点？
  - 如果不认同，你如何看待软件编码？

什么是软件编码？

- Professional Programmer = Software Engineer
- 正确观点
  - 软件编码是一个复杂而迭代的过程，包括程序设计和程序实现。
  - 软件编码要求
    - 正确地理解用户需求和软件设计思想
    - 正确地根据设计模型进行程序设计
    - 正确地而高效率地编写和测试源代码
  - 软件编码是设计的继续，会影响软件质量和可维护性。

软件编码包含的工作

- 程序设计
  - 理解软件的需求说明和设计模型
  - 补充遗漏的或剩余的详细设计
  - 设计程序代码的结构
- 设计审查
  - 检查设计结果
  - 记录发现的设计缺陷（类型、来源、严重性）
- 编写代码
  - 应用编码规范进行代码编写
  - 所编写代码应该是易验证的

软件编码的工作（续）

- 编译代码
  - 修改代码的语法错误
- 代码走查
  - 确认所写代码完成了所要求的工作
  - 记录发现的代码缺陷（类型、来源、严重性）

因影响编码质量的主要因素

- 良好的设计 （基础）

- 程序设计语言的选择
- 编码风格

  可靠性、可读性、可测试、可维护；

6.1 程序设计语言

一、程序设计语言的分类

1. 机器语言（Machine Language）

机器语言，是一种用二进制代码表示的低级语言，是计算机直接使用的指令代码。机器语言没有通用性、不能移植、因机器而异，因为处理机不同指令系统就不同。

用机器语言编写程序，都采用二进制代码形式，且所有的地址分配都以绝对地址的形式处理，存储空间的安排、寄存器、变址的使用也都由程序员自己计划。

6.1 程序设计语言

2、汇编语言（Assemble Language）

汇编语言，是一种使用助记符表示的低级语言。某一种汇编语言也是专门为某种特定的计算机系统而设计的。用汇编语言写成的程序，需经汇编程序翻译成机器语言程序才能执行。

汇编语言中的每条符号指令都与相应的机器指令有对应关系，同时又增加了一些诸如宏、符号地址等功能。虽然这种语言的命令比机器语言好记，但它并没有改变机器语言功能弱、指令少、繁琐、易出错、不能移植等的缺点。

内嵌汇编例子

例子（内存屏障）：

```
inline void CMBTreeMemoryBarrier()
{
    __asm__ __volatile__("" ::: "memory");
}
```

例子：（原子操作）

```
static __inline__ uint32_t atomic_compare_exchange(volatile uint32_t
* pv,
      const uint32_t nv, const uint32_t cv)
  {
    register unsigned int __res;
    __asm__ __volatile__ (
       LOCK_PREFIX "cmpxchgl %3,(%1)"
       : "=a" (__res), "=q" (pv) : "1" (pv), "q" (nv), "0" (cv));
    return __res;
  }
```

6.1 程序设计语言

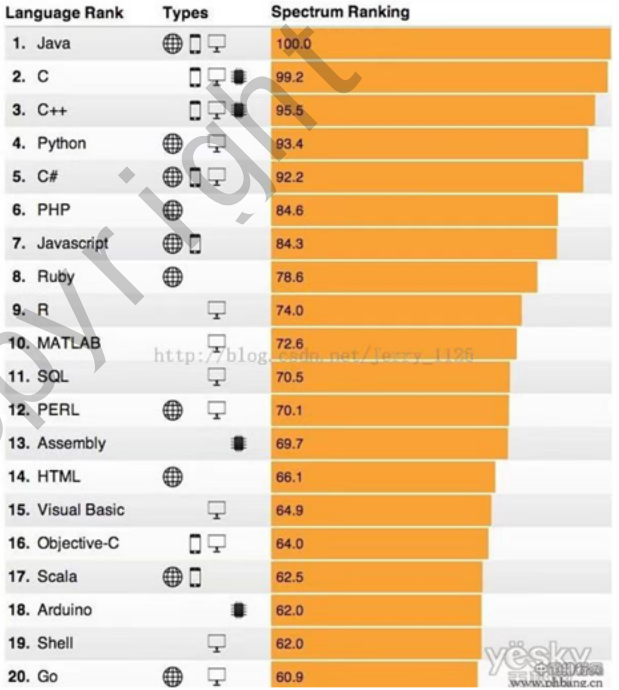
3、高级语言（High level Language）

高级语言是面向用户的、基本上独立于计算机种类和结构的语言。高级语言最大的优点是：形式上接近于算术语言和自然语言，概念上又接近于人们通常使用的概念。

高级语言的一个命令可以代替几条、几十条甚至几百条汇编语言的指令，因此，高级语言易学易用，通用性强且应用广泛。

常用高级编程语言

- C、C++、Go
- Java、Python、C#
- PHP、Ruby、R、
- Perl、Shell
- HTML、JavaScript



Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.2
3. C++		95.5
4. Python		93.4
5. C#		92.2
6. PHP		84.6
7. Javascript		84.3
8. Ruby		78.6
9. R		74.0
10. MATLAB		72.6
11. SQL		70.5
12. PERL		70.1
13. Assembly		69.7
14. HTML		66.1
15. Visual Basic		64.9
16. Objective-C		64.0
17. Scala		62.5
18. Arduino		62.0
19. Shell		62.0
20. Go		60.9

6.1 程序设计语言

4、第四代语言（Fourth Generation language，简称4GL）

第四代语言（4GL）的出现，将语言的抽象层次又提高到一个新的高度。第四代语言虽然也用不同的文法表示程序结构和数据结构，但第四代语言是在更高一级抽象的层次上表示这些结构。用第四代语言编码时只需说明“做什么”，而不需描述算法细节。（例如：SQL语言）

讨论： 自动代码生成技术

低代码平台

- 基于规则的自动生成（simulink ...）
- 基于AI的自动生成（Codex、GPT）

--- 从开源项目中学习并自动复制其代码，有侵权问题？

替代程序员？

总的来说，编码在软件工程中只占较小的一部分，分析和设计目前还不具备可替代性。即使自动代码生成技术比较成熟后，对整体开发效率的提升仍是有限的。

6.1 程序设计语言

二、程序设计语言的特点

1. 名字说明
2. 类型说明
3. 初始化
4. 程序对象的局部性
5. 程序模块
6. 循环控制结构
7. 分支控制给构
8. 异常处理
9. 独立编译

6.1 程序设计语言

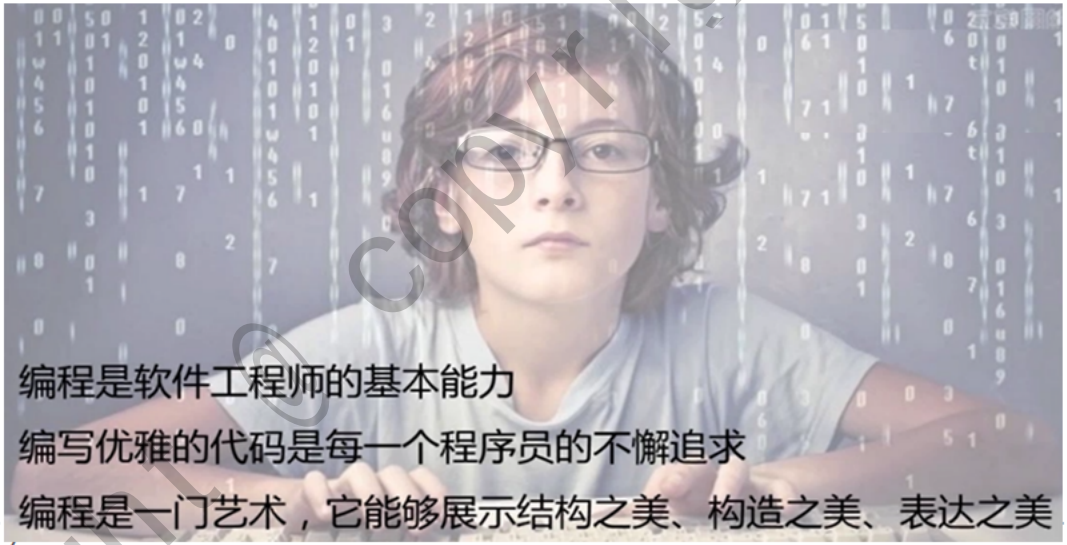
三、程序设计语言的选择

程序设计语言的选择常从以下几个方面考虑：

（1）项目的应用领域
（2）算法与计算的复杂性
（3）数据结构的复杂性
（4）效率
（5）可移植性
（6）程序设计人员的水平
（7）构造系统的模式

6.2 编码风格

- 高质量的代码不能只是运行正确的代码；
- 代码应该正确、易读、易于修改、易于扩展、易于维护；
- 提升自己的编程能力是每一个软件工程师的基本功。

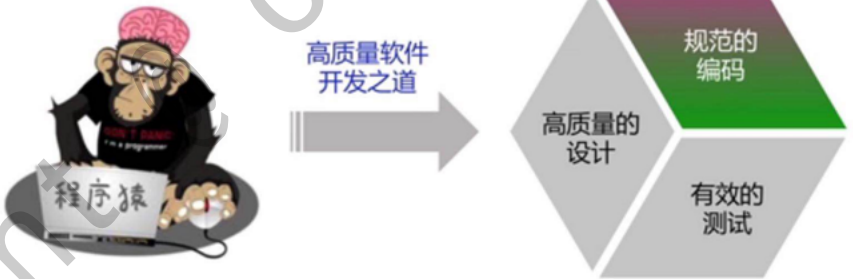


6.2 编码风格

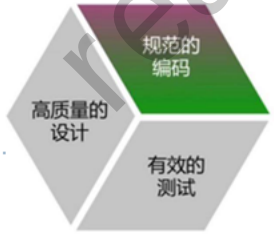
编码风格实际上是一种编码原则。

从20世纪70年代以来，编码的目标从强调效率转变到强调清晰。与此相应，编码风格也从追求“聪明”和“技巧”，变为提倡“简明”和“直接”。

人们逐渐认识到，良好的编码风格能在一定程度上弥补程序设计语言存在的缺点。反之，如果不注意编码风格，即使使用了结构化的现代语言，也很难写出高质量的程序。



软件编码规范



- 软件编码规范
  - 与特定语言相关的描写如何编写代码的规则集合
- 现实
  - 软件全生命周期的 70% 成本是维护
  - 软件在其生命周期中很少由原编写人员进行维护
- 目的
  - 提高编码质量，避免不必要的程序错误
  - 增强程序代码的可读性、可重用性和可移植性

编码规范的要求

- 基本要求
  - 程序结构清析且简单易懂，单个函数的行数一般不要超过 100 行（特殊情况例外）。
  - 算法设计应该简单且直接了当，代码要精简，避免出现垃圾程序。
  - 尽量使用标准库函数（类方法）和公共函数（类方法）。
  - 最好使用括号以避免二义性。
- 问题：以下示例有什么问题？如何修改？

```
if (cond1 && cond2 || cond3 && cond4 || cond5 && cond6)
    doSomethings();
```

编码规范的要求（续）

- 可读性要求：可读性第一，效率第二。
  - 源程序文件应有文件头说明，函数应有函数头说明。
  - 主要变量（结构、联合、类或对象）定义或引用时，注释要能够反映其含义。
  - 常量定义有相应说明。
  - 处理过程的每个阶段都有相关注释说明。
  - 在典型算法前都有注释。
  - 一目了然的语句不加注释。
  - 应保持注释与代码完全一致。
  - 利用缩进来显示程序的逻辑结构，缩进量统一为 4 个字节，不得使用 Tab 键的方式。
  - 对于嵌套的循环和分支程序，层次不要超过五层。

```
/*
  Push conditions if possible to all the materialized derived tables.
  Keep pushing as far down as possible making the call to this function
  recursively.

  @param thd      thread handler
  @returns false if success, true if error

  Since this is called at the end after applying local tranformations,
  call this function while traversing the query block hierarchy top-down.
*/
bool Query_block::push_conditions_to_derived_tables(THD *thd) {
  if (materialized_derived_table_count > 0)
    for (TABLE_LIST *tl = leaf_tables; tl; tl = tl->next_leaf) {
      if (tl->is_view_or_derived() && tl->uses_materialization() &&
          where_cond() && tl->can_push_condition_to_derived(thd)) {
        Item **where = where_cond_ref();
        Opt_trace_context *const trace = &thd->opt_trace;
        Condition_pushdown cp(*where, tl, thd, trace);
        // Make condition for the derived table
        if (cp.make_cond_for_derived()) return true;
        // The remaining condition that could not be pushed stays in this
        // WHERE clause.
        *where = cp.get_remainder_cond();
      }
    }
  /*
    Push conditions if possible to derived tables which were not merged. By
    running top-down, the resulting pushed down condition can be pushed down
    even more, in the case where a derived table contains an inner derived
    table.
   */
  for (Query_expression *unit = first_inner_query_expression(); unit;
       unit = unit->next_query_expression()) {
    for (Query_block *sl = unit->first_query_block(); sl;
         sl = sl->next_query_block()) {
      if (sl->push_conditions_to_derived_tables(thd)) return true;
    }
  }
  return false;
}
```

常用的编码规范

- 最常用的是google公司的编码规范：

https://github.com/google/styleguide

Our C++ Style Guide, Objective-C Style Guide, Java Style Guide, Python Style Guide, Shell Style Guide, HTML/CSS Style Guide, JavaScript Style Guide, AngularJS Style Guide, Common Lisp Style Guide, and Vimscript Style Guide are now available. We have also released cpplint, a tool to assist with style guide compliance, and google-c-style.el, an Emacs settings file for Google style.

If your project requires that you create a new XML document format, our XML Document Format Style Guide may be helpful. In addition to actual style rules, it also contains advice on designing your own vs. adapting an existing format, on XML instance document formatting, and on elements vs. attributes.

6.2 编码风格 –变量名的选择

- 采用有实际意义的变量名，望文知义。（尽量不要使用拼音）
- 长度原则：最小长度下的最大信息。
- 不用过于相似的变量名。不要用大小写来区分变量。
- 变量名中不要带有数字。
- 同一变量名不要具有多种含义。
- 显式声明变量。
- 对变量最好做出注释说明其含义（全程变量，成员变量）。

  例:成员变量以_结尾：void *cache_item_handle_;

- 命名规则尽量与所采用的操作系统或开发工具的风格保持一致。
- 程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。

一个例子 – 编码规范之命名

```
def fval(i):
    ret = 2
    n1 = 1
    n2 = 1
    i2 = i - 3
    while i2 >= 0:
        n1 = n2
        n2 = ret
        ret = n2 + n1
        i2 -= 1
    return 1 if i < 2 else ret
```

```
def fibonacci(position):
    if position < 2:
        return 1

    previous_but_one = 1
    previous = 1
    result = 2
    for n in range(2, position):
        previous_but_one = previous
        previous = result
        result = previous + previous_but_one
    return result
```

- 好的名字一目了然，不需要读者去猜测，甚至不需要注释。
- 例如，表示次数的量用***Times***，表示总量的用***Total***，表示平均值的用***Average***，表示和的量用***Sum***等。
- 函数名用动词或者动词+名次
- 变量名应用名语或者形容词+名语

匈牙利命名法

- 一个叫Charles Simonyi的程序员发明

[属性+]类型+对象

例如：

g_nLengh_of_array;

m_szUser_name;

6.2 编码风格-表达式的书写

简单直接地反映意图，不要卖弄技巧！

构造语句时应该遵循的原则是，每个语句都应该简单而直接，不能为了提高效率而使程序变得过分复杂；

6.2 编码风格 –表达式的书写

```
if ( !( char<0 || char > 9 ) )
```
改成
```
if ( char >= 0 && char <=  9 )
```
不要让读者绕弯子想。

- 尽量避免复杂的条件测试；
- 尽量减少对“非”条件的测试；
- 尽量少用中间变量
- 注意添加括号澄清计算意图，注意运算符的优先级
- 注意浮点运算的误差
- 注意整数运算的特点
- 调用函数的返回值，特别是异常值应该进行检查处理
- 如果语句的顺序很重要的话，应该加注释说明

If语句

- 布尔变量的正确使用。

```
if (flag)      // 表示flag为真
if (!flag)     // 表示flag为假
```

- 整型变量与零值比较，应当将整型变量用“==”或“！=”直接与0比较。
- 应当将指针变量用“==”或“！=”与NULL比较。
- 不可将浮点变量用“==”或“！=”与任何数字比较。应该设法转化成“>=”或“<=”形式。

If语句

```
if(condition)
        return x;
return y;
```

```
if (condition)
{
        return x;
}
else
{
        return y;
}
```

循环语句

- 在多重循环中，注意循环层次，以减少CPU跨切循环层的次数。

循环语句

- 如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层。例（b)的效率比(a)的高。

```
for (row=0; row<100; row++)
{
   for ( col=0; col<5; col++ )
   {
      sum = sum + a[row][col];
   }
}
```

示例(a)

```
for (col=0; col<5; col++ )
{
   for (row=0; row<100; row++)
   {
      sum = sum + a[row][col];
   }
}
```

示例(b)

循环语句

- 不可在for 循环体内修改循环变量，防止for 循环失去控制 。
- 建议for语句的循环控制变量的取值采用“半开半闭区间”写法。

switch语句

- 每个case语句的结尾不要忘了加break，否则将导致多个分支重叠（除非有意使多个分支重叠）。
- 不要忘记最后那个default分支。即使程序真的不需要default处理，也应该保留语句 default : break;

```
switch (num) {
    case 1:
        console.log('1');
        break;
    case 2:
        console.log('2');
        break;
    case 3:
        console.log('3');
        break;
    default:
        console.log('没有匹配上');
        break;
}
```

尽量避免多个case共享处理的情景，后续修改时容易引入问题。

例如：1和3共享处理，如果将来在1和3中间加入了2，并做了break，则1的处理丢失了。

中间加入case，容易引起问题

```
case 1:
case 3:
    console.log("1&&2");
    break;
```

GOTO语句的使用

- goto语句可以灵活跳转，但是经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句。
- 在处理例外时，用goto语句从多重循环体中一下子跳到外面，用不着写很多次的break语句。

其他（1）

- 当心那些视觉上不易分辨的操作符发生书写错误。
  （“==”与“=”，
  “||”、“&&”、“<=”、“>=”易发生“丢1”失误。
  编译器却不一定能自动指出这类错误。）

  例如：if（a==1）误写为if（a=1）。 建议写为if（1==a）

- 变量（指针、数组）被创建之后应当及时初始化，以防止把未被初始化的变量当成右值使用。
- 当心变量的初值、缺省值错误，或者精度不够。
- 当心数据类型转换发生错误。尽量使用显式的数据类型转换 。

其他（2）

- 当心变量发生上溢或下溢，数组的下标越界。
- 当心忘记编写错误处理程序。
- 尽量使用标准库函数，不要“发明”已经存在的库函数。
- 尽量不要使用与具体硬件或软件环境关系密切的变量。

6.2 编码风格 -输入输出

- 对所有的输入数据都要进行检验，识别错误的输入，以保证每个数据的有效性；
- 检查输入项的各种重要组合的合法性，必要时报告输入状态信息；
- 使得输入的步骤和操作尽可能简单，并保持简单的输入格式；
- 输入数据时，应允许使用自由格式输入；
- 应允许缺省值；

```
int CbBlockCacheReader::parse_record_buffer(。。。
                                ObSSTableReader* sstable_reader)
{
  。。。
  if (NULL == sstable_reader)
  {
    ret = CB_ERROR;
    CBSYS_LOG(WARN, "sstable reader is NULL");
  }
```

6.3 程序内部文档

- 1. 描述性注释嵌在程序之中。

➢边写代码边注释，变量定义和分支语句一般要有注释。

➢修改代码同时修改相应的注释。
不再有用的注释要删除。

➢注释应该与程序一致。但不应注释含义已十分清楚的代码。
注释应是对代码的“提示”，而不是重复程序语句。

➢注意格式，注释的位置应与被描述的代码相邻。
一般为代码的上方且为左对齐。

➢当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

2. 序言性注释

- 通常置于每个程序模块的开头部分，它应当给出程序的整体说明，对于理解程序本身具有引导作用。
- 序言性注释包括：
  - 程序标题；
  - 有关本模块功能和目的的说明；
  - 主要算法；
  - 接口说明：包括调用形式，参数描述，子程序清单；
  - 有关数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
  - 模块位置：在哪一个源文件中，或隶属于哪一个软件包；
  - 开发简历：模块设计者，复审者，复审日期，修改日期及有关说明等。

3. 功能性注释

- 功能性注释嵌在源程序体中，用以描述其后的语句或程序段是在做什么工作，或是执行了下面的语句会怎么样，而不要解释下面怎么做。
- 例如

```
/* ADD AMOUNT TO TOTAL */
TOTAL = AMOUNT + TOTAL
```

上面注释不清楚，如果注明把月销售额计入年度总额，便使读者理解了下面语句的意图：

```
/* ADD MONTHLY-SALES TO ANNUAL-TOTAL */
TOTAL = AMOUNT + TOTAL
```

- 要点
  - 描述一段程序，而不是每一个语句；
  - 用缩进和空行，使程序与注释容易区别；
  - 注释要正确。

```
////////////////////////////////////////////////////////////////////////////////
// FUNCTION: ServiceMain
// OVERVIEW: 服务主函数
// PURPOSE :  在服务控制要求的处理函数登录之后、
//              调用服务执行函数，进行执行处理。
// RETURNS :  无
////////////////////////////////////////////////////////////////////////////////
void  CServiceModule::ServiceMain
(
      DWORD           dwArgc,     // 参数的个数
      LPTSTR*         lpszArgv    // 指向参数表的指针
)
{

  TRACE(_T("ServiceModule: ServiceMain\n"));

  //在服务控制要求的处理函数登录
  m_status.dwCurrentState           = SERVICE_START_PENDING;
  m_hServiceStatus      = RegisterServiceCtrlHandler(m_szServiceName, _Handler);

  if (m_hServiceStatus == NULL)
  {
    TRACE(_T("ServiceModule: Handler not installed\n"));
    return;
  }

  SetServiceStatus(SERVICE_START_PENDING);

  m_status.dwWin32ExitCode          = S_OK;
  m_status.dwCheckPoint             = 0;
  m_status.dwWaitHint  = 0;

  // 执行函数返回时、服务停止。
  Run();

  // 服务终了
 LogEvent(MSG_FREE_FORMAT, EVENTLOG_INFORMATION_TYPE, IDS_SERVICE_END);
  SetServiceStatus(SERVICE_STOPPED);
}
```

练习1-if语句

- 请写出 BOOL flag 与“零值”比较的 if 语句：
- 请写出 float x 与“零值”比较的 if 语句：
- 请写出 char *p 与“零值”比较的 if 语句：

标准答案 1

标准答案：
if (flag)
if (!flag)
	如下写法均属不良风格，不得分。
	`if (flag == TRUE)`
	`if (flag == 1 )`
	`if (flag == FALSE)`
	`if (flag == 0)`

标准答案 2

标准答案示例：
 	
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON)
 	
不可将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”此类形式。
	如下是错误的写法。
	`if (x == 0.0)`
	`if (x != 0.0)`

标准答案 3

标准答案：
if (NULL == p)
if (p != NULL)
	如下写法均属不良风格。
	if (p == 0)
	if (p != 0)
	if (p)
	if (!)
	if(*p == NULL)

请简述以下两个**for**循环的优缺点

```
// 第一个
for (i=0; i<N; i++)
{
    if (condition){
        DoSomething();
    }
    else {
        DoOtherthing();
    }
}
```

优点：
缺点：

```
// 第二个
if (condition)
{
    for (i=0; i<N; i++)
        DoSomething();
}
else
{
    for (i=0; i<N; i++)
        DoOtherthing();
}
```

优点：
缺点：

答案

优点：程序简洁 缺点：多执行了N-1次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。	优点：循环的效率高 缺点：程序不简洁

6.4 程序效率

程序的效率是指程序的执行速度及程序所需占用的内存的存储空间。

程序编码是最后提高运行速度和节省存储的机会，因此在此阶段不能不考虑程序的效率。

6.4 程序效率

尽管效率是值得追求的目标，但不应为了非必需的效率提高而牺牲代码的清晰性、可读性和正确性。应记住下面三条准则。

（1）效率是一种性能需求，目标值应当在需求分析阶段给出。软件效率应以需求为准，不应以人力所及为准。

（2）好的设计可以提高效率。

（3）代码效率与代码的简单性相关。

6.4 程序效率 – 注意点

- 不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率。
- 以提高程序的全局效率为主，提高局部效率为辅。
- 先优化数据结构和算法，再优化执行代码。
- 有时候时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷。例如多花费一些内存来提高性能。

6.4 程序效率 – 如何提高代码效率1

一、代码效率

（1）应先简化算术和逻辑的表达式。

（2）仔细研究嵌套的循环，以确定是否有语句可以从内层往外移。

（3）尽量避免使用多维数组。

（4）尽量避免使用指针和复杂的列表。

（5）使用执行时间短的算术运算。

（6）即使语言允许，一般也不要采用混合数据类型。

（7）尽量使用整数表达式和布尔表达式。

6.4 程序效率- 如何提高代码效率2

二、存储器效率

采用结构化程序设计，将程序功能合理分块，使每个模块或一组密切相关模块的程序体积大小与每页的容量相匹配，可减少页面调度、减少内外存交换，提高存储器效率。

6.4 程序效率 – 如何提高代码效率3

三、输入/输出的效率

（1）所有输入/输出都应该有缓冲，以减少过多的通信次数。

（2）对辅存（如磁盘），应选用最简单的访问方法。

（3）辅存的输入/输出，应该以块为单位进行。

（4）终端和打印机的输入/输出，应当考虑设备的特性，以提高输入/输出的质量和速度。

6.4 程序效率 – 如何提高代码效率4

四、数据库访问效率

（1）采用性能高的数据库。

（2）建立合理的索引，避免扫描多余数据，避免表扫描！（例如NOT IN都是最低效的）。

（3）应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

（4）下面的查询也将导致全表扫描：

select id from t where name like ‘%abc%’ 。

参考：

https://www.cnblogs.com/ShaYeBlog/archive/2013/07/31/3227244.html。

6.5 设计方法论

一 自顶向下的程序开发方法

二 自低向上的程序开发方法

6.5 设计方法论

程序设计自动化：

一：使用某种方式精确地定义用户的需求，由专门的程序将用户需求的定义转变成程序代码。

二：模块化积累方式。

三：扩展的自动化程序设计范型。

6.5 设计方法论

一、代码文档化：指编码时适当选择标识符的名字、适当安排注释和注重程序的整个组织形式。

二、数据说明：程序或模块在其可执行部分的前面都集中了一些说明语句 ，出于阅读理解和维护的要求，最好使其规范化，使说明的先后次序固定 。

三、语句构造 ：每条语句都应当简单而直接，同时也不应为了追求运行效率而使代码复杂化，这样会减低程序的可读性。

四、输入/输出：遵循源程序的输入输出风格。

程序设计工具：

一：编译程序

二：代码管理系统

CVS、git、VSS。。。

小结

- ✓ 编程风格
- ✓ 语言选择
- ✓ 编程方法
- ✓ 编程工具

```
void   fval(i)
{
      int ret = 2;
      n1 = 1;
      n2 = 1;
      i2 = i-3;
      while ( i2 >= 0)
      {
          n1 = n2;
          n2 = ret;
          ret = n1 + n2;
          i2 -= 1;
       }
       if (i<2)
         return 1;
       else
         return ret;
}
```

```
int fibonacci(int position)
{
     int result = 0;
     int prev_prev = 1;
     int prev = 1;
     result = prev_prev + prev;

     if(position <= 0){
         result = -1;
     }else if(position < 2){
         result = 1;
     }else{
         int count = position - 3;
         while( count >= 0 ) {
             prev_prev = prev; prev = result;
             result = prev_prev + prev;
             count--;
          }
     }
     return result;
 }
```

END