软件线 C++编程规范

drangon

2007-4-18 create

2011-11-14 update

1. 概述

本文档描述软件线团队在进行 C/C++编程开发时需要遵循的规范。

1.1. 目的

制定编程规范，目的是使代码易于理解，易于维护。团队保持相近的风格，也便于团队协作。遵循良好的编程规范，也有利于减少代码的 bug。

1.2. 实施

对于新开发的代码，需要按照本规范进行开发。对于已有的代码，可以沿用已有代码的风格，保持一致。或者在修改的函数或者类的同时，对相关代码进行整理，以符合规范。

除非团队统一规划确定整理风格，否则对已有代码中不需要修改的部分，不要进行代码风格整理。

团队定期进行代码 review 和评审，将近期的一些常见问题提出来进行改进。

2. 基础代码规范

2.1. 头文件和 CPP 文件

头文件例子：

```
/* Copyright (c) 2010~2011, 浙江大华技术股份有限公司, All rights reserved.
 * 2010-07
 *
 * DThread.h : 线程运行对象
 */

#ifndef INCLUDED_LIBDSL_DTHREAD_H
#define INCLUDED_LIBDSL_DTHREAD_H

#include <libdsl/DMutex.h>

#include <deque>
#include <map>
// using namespace std;   // ERROR : 头文件禁止使用using关键字

class DRefObj;      // 能够用前向声明，就不要直接 #include

BEGIN_NAMESPACE_DSL
////////////////////////////////////////////////

enum dsl_thread_state_e {
    DSL_THREAD_STATE_UNKNOWN = 0,  // 不应该出现的状态
    DSL_THREAD_STATE_IDLE,         // 线程未启动或者已经停止
    DSL_THREAD_STATE_RUNNING,      // 线程正在运行
    DSL_THREAD_STATE_STOPPING,     // 线程等待退出，等待回收资源（回收后返回IDLE状态）
};

class DThread
{
public:
    DThread() : m_state( DSL_THREAD_STATE_IDLE ) { }
    virtual ~DThread(){ Stop(); }

    // 启动线程
    virtual int Start();
    // 通知线程退出，
    virtual int SignalStop() { m_state = DSL_THREAD_STATE_STOPPING; return 0; }
    // 通知线程退出，并且等待线程退出完成才返回
    virtual int Stop();

    bool IsRunning() { return m_state == DSL_THREAD_STATE_RUNNING; }

    static void SleepMs( int ms );

protected:
    virtual int run();

protected:
    DRefObj * m_just_an_example;
    volatile dsl_thread_state_e m_state;
};

////////////////////////////////////////////////
END_NAMESPACE_DSL
```

```
#endif // INCLUDED_LIBDSL_DTHREAD_H
```

定义头文件时的一些注意事项：

- 文件开头写文件说明的注释，包括版权信息，文件主要内容等
- 使用 INCLUDED_<project>_<filename>_H 作为防止重复包含保护的名字，如果文件名本身带一定前缀，可省略 project 信息。
- 头文件中引用其他头文件时，要放在 namepsace 外部，项目头文件在前，系统头文件在后
- 头文件中禁止使用 using 关键字
- 头文件引用要以最少原则，避免不必要的依赖，不用到的头文件不要引用，能够前向声明就不要直接#include 引用
- 头文件中定义的内容也应该最少原则，只定义外部需要用到的内容，以减少潜在冲突。而对于只在内部使用的宏或者结构体等，应该定义在 CPP 文件中或者另一个专门的内部使用不对外发布的头文件中。
- 在最少原则的同时，头文件也要做到自完备。在 CPP 中使用时，不应该依赖于先引用了其他头文件。

CPP 文件例子：

```
/* Copyright (c) 2010~2011, 浙江大华技术股份有限公司, All rights reserved.
 * 2010-07
 *
 * DThread.cpp : 线程运行对象
 */

#include "DThread.h"          // 类对应的头文件放第一个

#include <libdsl/DRefObj.h>

#include <string.h>

using std::map;     // 尽量只引入指定类型，而非整个namespace

BEGIN_NAMESPACE_DSL
////////////////////////////////////////////////

int DThread::Start()
{
    m_state = DSL_THREAD_STATE_RUNNING;
    if( DSL_THREAD_START( m_thread, s_ThreadProc, this ) ) {
        return 0;
    }
    else {
        m_state = DSL_THREAD_STATE_IDLE;
        return DSL_ERROR_FAILED;
    }
```

```
    return 0;
}

int DThread::Stop()
{
    if( m_state == DSL_THREAD_STATE_IDLE )
        return 0;
    SignalStop();
    DSL_THREAD_WAIT_CLOSE( m_thread );
    m_state = DSL_THREAD_STATE_IDLE;
    return 0;
}

void DThread::SleepMs( int ms )
{
    DSL_SLEEP_MS( ms );
    return;
}

////////////////////////////////////////////////
END_NAMESPACE_DSL
```

定义 CPP 文件时的一些注意事项：

- CPP 文件中第一个引用的头文件为对应的类的头文件，以验证类的头文件的自完备性，然后是其他项目头文件，最后是系统头文件。CPP 文件中引用头文件同样是最少原则，避免不必要的依赖。
- 类的成员函数的顺序，应该与类头文件中定义顺序保持一致

文件名命名时要注意大小写（windows 平台不区分大小写，但 linux 区分），保持统一风格，#include 语句中也要注意文件名的大小写。

2.2. 排版

代码编写时要注意排版，保持代码整齐，便于阅读。

- 缩进统一使用 TAB 键，不允许使用空格（复制代码要注意）
- 函数内大括号建议使用紧凑模式（左大括号不单独成行），也可以使用列对齐模式（即左大括号单独成行）。
- 代码中适当留空格，不要密密麻麻，例如运算符两端留空格，括号两内侧留空格等
- 代码中适当留空行（但一般不应出现两个连续空行），例如函数间留空行，函数内功能块间留空行等

- 每行只写一条语句。定义变量时也每行只定义一个变量。避免一个函数中定义太多变量，变量较多时要对变量意义用途加注释。
- 每行尽量不超过 80 字符（最大不超过 120 字符）
- 每函数尽量不要超过 200 行，函数内部的嵌套缩进尽量不要超过 4 层，通过将异常情况先处理，以及拆分子函数等方式来简化。

2.3. 命名和类型

函数、类、变量等命名：

- 一般是作用范围广的长一些，局部使用的短一些，尽量减少名字的作用范围，同时通过添加前缀、放入 namespace 等方式避免命名冲突，特别是比较短或者常见的名字。
- 命名应该有意义，除了循环变量外，不应使用 i、j、a、b 等无意义名字，缩写要符合习惯且保持统一
- 全局变量和函数使用“g_”前缀，静态变量和函数使用“s_”前缀（这两者应该尽量避免使用）。类成员变量使用“m_”前缀。
- 类的命名采用每个单词首字母大写的方式。
- 枚举类型的命名使用“_e”后缀
- 宏和枚举值命名使用大写加下划线的方式。程序中应避免直接使用数字常量，特别是会多次出现的时候，应该定义相应的宏或枚举，并且有相关注释。

2.4. 类的设计

定义类时的一些注意事项：

- 类和函数功能要单一，不要赋予太多职责
- 除了一些简易的数据结构类外，一般情况下，成员变量不能是 public，也不要添加太多成员变量的 get/set 函数，要从功能而非数据的角度思考设计
- 类的内部成员，按照 public 函数、protected 函数、private 函数、protected 成员变量、private 成员变量的顺序进行排列。每一类再按照功能逻辑进行分块，

块间使用空行隔开。

- 继承优先使用接口继承而不是功能继承，功能重用优先使用组合而非继承
- 父类的虚函数，子类重载时，必须加上 virtual（不加也是 virtual 从而易误解）

2.5. 函数设计

定义函数时的一些注意事项：

- 函数的输入参数中，简单类型直接使用，复杂类型使用 const T & ，或 const T * （这时可以为 NULL）。输出参数使用 T * （不能使用 T & ），如果需要内部申请外面释放（这种情况应尽量避免，并且需要有良好注释说明），则使用 T ** 。避免将参数同时做输入输出用。
- 函数的返回值如果是指针，指向动态生成内容，且需要外部释放的话，必须有注释说明。一般情况下都应该提供成对的申请和释放函数。
- 函数一般返回 0 表示成功，负数表示失败。返回指针的话，非 NULL 表示成功，NULL 表示失败，其他返回值必须明确说明。一般返回 bool 类型的函数，都是以 IsXXXX()、HasXXXX()方式命名。

2.6. 注释

注释方面：

- 代码中适当注释，要提供有效信息。例如代码块中添加“// <1>”“// <2>”来说明主要流程和步骤。
- 对于不完善的代码，添加“// FIXME”表示未实现或有 bug 必须尽快修改，添加“// TODO”的注释表示需要优化或者补充完备（注意都是大写）。另外 // 和 /* 后面要留一个空格。
- 注释应该解释“为什么要这样做”，而不仅仅“做了什么”
- 对于 for、if、while 等代码块，如果代码行数较多，那么在代码块结束的右大括号处，要加上相关注释

2.7. 其他

其他方面：

- 不要使用 C++ exception
- 不要使用 C++ RTTI
- 在栈上（局部变量）不能定义超过 8K 的数组或者对象实例，应改成从堆中申请（通过 new 或者 malloc）
- 优先级不清晰的地方，要添加括号
- 宏的定义中要适当添加括号：如 #define A(x,y) ((x) - (y) + 3)
- 所有变量使用前都应该初始化
- 检查并修正所有的编译 warning，使用 cppcheck、pclint 等工具辅助检查
- 使用 ASSERT 来进行断言，及早暴露问题，同时也相当于注释。ASSERT 中的代码不能有副作用，只能调用 const 性质函数，ASSERT 中的代码有可能不被执行。

3. 代码编写的注意事项

3.1. 内存管理

注意内存分配和释放，防止内存泄露。

- new 和 delete、new []和 delete[]、malloc 和 free 要对应
- 释放后要将指针置空，如：delete m_data, m_data = NULL;
- 对于每一个 new 等申请的内存，要明确何时释放，要检查各可能的执行路径是否忘记 delete 释放。
- new 和 delete 必须在同一个层次中，如果某个类的函数返回一个申请的内存，那么应该由这个类的另一个函数进行释放。不能出现一个模块申请的内存在另一模块释放，或者一个 DLL 申请的内存在另一个 DLL 释放等情况。

3.2. 缓冲区越界

代码中要注意检查输入输出缓冲区大小，防止读写越界。

- 代码中对数组进行读写时，需要判断数组下标是否在数组大小范围内，防止读写越界。函数间传递数组时，一般都应该带上数组大小。
- 不要使用 strcpy()、gets()等不安全的函数。（另外注意 strncpy()的语义，不一定“\0”结束，必定写满 n 字节。注意 snprintf()的语义，必定以“\0”结束，不同情况下的 size 值。注意 fgets()的语义，必定以“\0”结束，有可能带“\n”。）

3.3. 输入验证

凡是对于不可信任的输入数据都需要进行验证。对于可信任的数据，但可能出现损坏的，也要进行校验和容错处理。

验证输入数据时，定义合法数据的格式而不是非法数据的格式，因为定义非法数据容易遗漏。验证时，只有合法数据才处理，其他数据都拒绝，而不是试图排除非法数据。

3.4. 大小端的处理

目前我们的 CPU 基本都是小端为主，但在嵌入式设备上，会碰到大端的情况，需要进行处理。

凡是涉及与其他机器程序交换的数据，如网络二进制协议，二进制文件数据等，需要考虑大小端问题。内部使用的数据，以及文本格式的数据不受影响。

因此制定二进制格式的通信协议或文件格式时，要明确规定大小端。代码中，需要进行指定大小端格式到本机大小端格式的转换。

使用某些标准网络协议时，也要注意其所规定的大小端格式，程序中要与本地大小端格式相互转换。

由于现在小端比较普遍，可以将协议定义成小端，然后将转换函数定义成空，以后碰到大端机器时，再补充相应转换函数。

3.5. 兼容 64 位平台

由于 windows 和 linux 在 64 位中对 long 的大小定义不一致。因此要考虑相关差异。不能假设 long 就是 4 字节，也不能假设 long 能容纳一个指针。

使用整数相关类型时，如果需要限定长度，那么需要使用 int32_t / uint64_t 等固定长度类型（#include <dsl/dslbase.h>）。

需要将指针赋值给整数时，需要使用 uintptr_t 类型而不是 long。

3.6. 平台可移植性处理

平台差异包括 windows 和 linux 平台，或者 QT、MFC 等库的差异。

平台的差异应该封装在底层库 DssBaseLib 和上层界面库和界面程序中，中间的业务逻辑模块编码中不应该直接使用平台相关的内容（特别是 VC 开发中，业务逻辑代码不要使用 MFC 库函数如 CString 等。）

3.7. 中文和多语言支持

代码中的中文注释，统一使用 GBK 编码，linux 下编辑时要注意 locale 设置。

程序调试日志中禁止使用中文。

数据库内部存储的内容，统一使用 UTF-8 格式。Web 统一使用 UTF-8 格式。

界面显示的内容，要注意多语言支持，一般是代码中写 ASCII（英文），然后通过语言文件转换成相应的语言。具体参见多语言支持的库。

除注释外，代码中不应该出现其他中文字符，实在有需要的，应该通过读取文件实现。

语言文件及其他涉及中文和多语言支持的，应该尽量使用 UTF-8 格式，内存中业务处理过程中的中文，也应该尽量使用 UTF-8 格式。

网络协议等的中文字符具体参照相应协议格式，自定义协议时优先使用 UTF-8 格式。

4. 其他编程规范

4.1. SVN 代码管理

SVN 中存放源代码，不存放中间过程文件（如 xxx.obj 等），依赖的外部库和最终输出文件由模块和项目具体来规定。

文件改名字使用 svn move，不要先删除再重新添加。文件复制使用 svn copy，不要自行复制后再 svn add。

提交的代码必须是可编译通过，功能初步自测（以及通过 gtest 测试项）。

提交时必须写日志，说明修改的内容，提交的粒度要适中，要相对完整。

修改代码要相互沟通，避免在不知情时同时修改相同的文件的相同位置，造成大量代码冲突 merge 会很麻烦。

4.2. 文档编写

如果有相关格式要求或者文档模板的，按相关要求和模板编写。否则遵循下列要求。

模块的业务功能设计、技术方案设计、重大 bug 分析、技术预研、以及需要开会讨论的内容等，都应该编写相关文档。一般是编写初稿、提出问题、会议讨论、总结完善整理文档的流程。

简短的文档使用 txt 格式。相对正式或者较长的文档使用 Word 格式。

文档要简洁，紧凑，要提供有用信息，保持更新。

文档之间内容不要互相拷贝，造成冗余（这样容易出现更新后不一致的情况），而是应该直接列出相关参考文档。

较长文档要注意条理性，规划好各级标题，能够通过“文档结构图”了解文档内容的整体情况。

建议将本文档样式设为缺省（word 2003 中，执行菜单“工具”→“模板和加载项”，选择左下角“管理器”。选择左边的标题、正文等样式，复制到右边的 Normal.dot 中即可），然后可以通过 ctrl+alt+1、2、3 来快速设定各级标题。