Module06-07
C++ ACE：杂项

C++ ACE – 杂项

- ACE 简介
- I/O 相关对象
- Reactor 框架
- Service Configuration 框架
- Task 框架
- Acceptor-Connector 框架
- Proactor 框架
- ➔ 杂项

杂项 - 概要

- 杂项：
  - 服务器端应用程序一般会需要：
    - 写日志， ACE 提供了强大而且方便使用的 ACE_Log_Msg 类
    - 读取配置文件，在 ACE 中可以通过 ACE_Configuration_Heap 来读取应用所需要的配置文件
    - 方便地访问和修改全局资源，最有效、安全的方式是将这些资源用一个 Singleton 实例来管理
  - 本次课程将通过示例来展示：
    - ACE_Log_Msg
    - ACE_Configuration_Heap
    - ACE_Singleton

    三个实用类的能力和用途。

杂项 - ACE_Log_Msg

- 杂项：
  - ACE_Log_Msg
  - ACE_Configuration_Heap
  - ACE_Singleton

ACE_Log_Msg - 写日志

- 写日志的宏
  - 写日志的操作是通过一系列的宏 (Macro) 来实现的，语法：

```
// 语法：severity为日志的严重级别
ACE_DEBUG((severity, formatting-args)); // 一般用于记录调试信息
ACE_ERROR((severity, formatting-args)); // 一般用于警告和错误信息
```

  - 示例：

```
const char* msg = "test message.";
ACE_DEBUG((LM_DEBUG, "(%t) Receive msg: %s\n", msg));

const char* testMst = "Message";
ACE_DEBUG((LM_INFO, "(%P) %D %s#%d\n", testMst, 120));
```

```
// 示例输出：
(3078994144) Receive msg: test message.
(3378) Fri May 28 2010 10:28:14.585661 Message#120
```

ACE_Log_Msg - 日志严重级别

- 日志严重级别列表

严重级别	说明
LM_TRACE	用于跟踪函数的调用
LM_DEBUG	调试信息
LM_INFO	类似 LM_DEBUG
LM_NOTICE	用于提示一些需要处理的状况
LM_WARNING	警告信息
LM_ERROR	错误信息
LM_CRITICAL	危险信息，如硬件设备错误
LM_ALERT	必须立即处理的事件，如损毁的系统数据库
LM_EMERGENCY	紧急信息，必须立即向所有用户广播

ACE_Log_Msg - 设置日志严重级别

- 使用 ACE_Log_Msg::priority_mask() 设置日志严重级别
  - 示例

```
void foo(void) {
    ACE_TRACE (ACE_TEXT ("foo")); // #1
    ACE_DEBUG ((LM_NOTICE, ACE_TEXT ("%IHowdy
Pardner\n"))); // #2
}

int main() {
    ACE_TRACE (ACE_TEXT ("main")); // #3
    ACE_LOG_MSG->priority_mask(LM_DEBUG | LM_NOTICE, // #0
            ACE_Log_Msg::PROCESS);
    ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IHi Mom\n"))); // #4
    foo();
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("%IGoodnight\n"))); // #5
}
```

上例中设置日志的级别为 :LM_DEBUG 和 LM_NOTICE，除此之外，其它所有级别的日志记录将不会输出。

ACE_Log_Msg - 关于 ACE_LOG_MSG

- 宏 ACE_LOG_MSG
  - ACE_LOG_MSG 是一个 ACE 预定义的宏：

```
#define ACE_LOG_MSG ACE_Log_Msg::instance()
```

  - ACE_Log_Msg 则是一个单例类

ACE_Log_Msg - 格式化字符

- 几个特别的格式化字符列表

字符	参数类型	输出
l		ACE_DEBUG 等宏出现在源文件中的行号
N		文件名
n		应用程序名（传给 ACE_Log_Msg::open() 的参数）
P（大写）		当前进程 id
p（小写）	ACE_TCHAR*	相当于 perror()， errno 的文字描述
S（大写）	int	信号值对应的名称
T		当前时间: hour:minite:second.usec
D		当前日期和时间
t		当前线程 id

其它格式化字符，类似 C 的 printf() 一族函数

ACE_Log_Msg - 日志输出重定向

- 设置日志输出的目的地
  - 默认情况下， ACE_Log_Msg 输出到标准错误 (stderr)，我们可以通过在 open() 函数和 set_flags() 函数来设置输出的目的地
  - set_flags() 的有效 flag

Flag	效果
STDERR	输出定向到标准输出
LOGGER	输出定向到本地日志守护进程
OSTREAM	输出定向到输出文件流
MSG_CALLBACK	将输出写到 callback 对象 (ACE_Log_Msg_Callback)
VERBOSE	详细输出模式，每次输出的内容包括应用程序的名称、时间戳、主机名、进程 id、严重级别
VERBOSE_LITE	较详细模式，每次输出的内容包括时间戳、严重级别
SILENT	不输出
SYSLOG	输出定向到系统事件日志

ACE_Log_Msg - 日志输出重定向示例

- 输出到文件流

```
int main(int argc, char* argv[]) {
    std::ofstream* output = new std::ofstream("server.log",
ios::app);
    ACE_LOG_MSG->open(argv[0],
        ACE_Log_Msg::OSTREAM | ACE_Log_Msg::VERBOSE_LITE);
    ACE_LOG_MSG->msg_ostream(output, 1);
    ACE_LOG_MSG->priority_mask(LM_DEBUG | LM_NOTICE,
                ACE_Log_Msg::PROCESS);

    ACE_DEBUG((LM_DEBUG, " (%P | %t) %d", 1));
}
```

杂项 - ACE_Configuration_Heap

- 杂项：
  - ACE_Log_Msg
  - ACE_Configuration_Heap
  - ACE_Singleton

ACE_Configuration_Heap - 配置文件

- 配置文件的格式
  - ACE_Configuration_Heap 支持传统的 ini 格式和 xml 格式的配置文件（这次仅讨论 ini 格式）
  - 一个简单的配置文件片段示例

```
# File: server.conf
[server]
    ip = 192.168.0.106
    port = 8868
    threads = 5

# connemt text.
[database]
    user = tiger
    passwd = scott
```

每个 [ ] 表示一个 Section
以 # 开头的行为注释

ACE_Configuration_Heap - 示例

- 读取配置的示例

```
#include <iostream>
#include <string>
#include <ace/Log_Msg.h>
#include <ace/Configuration_Import_Export.h>

int main() {
    using namespace std;

    ACE_Configuration_Heap config;
    int result = config.open();
    if (result != 0)
        ACE_ERROR_RETURN((LM_ERROR,"Open config error:
%d\n",result),-1);

    ACE_Ini_ImpExp importExport(config);
```

ACE_Configuration_Heap - 示例

- 读取配置的示例

```
    // 从配置文件server.conf中读取配置
    result = importExport.import_config("server.conf");
    if (result != 0)
        ACE_ERROR_RETURN((LM_ERROR,"(%t)Importing Error:
(%d)\n",result),-1);
    // 创建root key
    ACE_Configuration_Section_Key root = config.root_section();

    ACE_Configuration_Section_Key server;
    if (config.open_section(root, "server", 1, server) == 0) {
        ACE_TString ip, port, threads;
        if (config.get_string_value(server, "ip", ip) == 0)
            cout << ip << endl;
        if (config.get_string_value(server, "port", port) == 0)
            cout << port << endl;
        if (config.get_string_value(server, "threads", threads) ==
0)
            cout << threads << endl;
    }
```

ACE_Configuration_Heap - 示例

- 读取配置的示例

```
    ACE_Configuration_Section_Key database;
    if (config.open_section(root, "database", 1, database) == 0) {
        ACE_TString user, passwd;
        if (config.get_string_value(database, "user", user) == 0)
            cout << user << endl;
        if (config.get_string_value(database, "passwd", passwd) ==
0)
            cout << passwd << endl;
    }
}
```

杂项 - ACE_Singleton

- 杂项：
  - ACE_Log_Msg
  - ACE_Configuration_Heap
  - ACE_Singleton

ACE_Singleton - 简介

- 关于类 ACE_Singleton
  - 关于单例模式 (Singleton Pattern) 的详细介绍，请参考 GOF 的《 Design Patterns - Elements of Reusable Object-Oriented Software 》
  - 在多线程环境下，保证 Singleton 实例化的唯一性、也即线程安全的，且尽量减少同步（加锁、解锁）带来的开销，其内部采用的是 Double-Checked Locking 优化模式
  - ACE_Singleton 实现为一个类模板，通过模板参数的不同确定不同的同步机制
  - 可以直接将一个现存的 class 指定为一个单例类 (Singleton)

ACE_Singleton - Singleton 实现

- 简单的 Singleton 实现（非线程安全版本）

```
class Singleton {
public:
    static Singleton* instance() {
        if (0 == _instance)
            _instance = new Singleton;
        return _instance;
    }
    // 其它成员函数
private:
    static Singleton* _instance;

    // 将以下函数置为private
    Singleton() {}
    Singleton(const Singleton& s) {}
    Singleton& operator=(const Singleton& s) { return *this; }
};

Singleton* Singleton::_instance = 0; // 将指针 _instance 置0
```

ACE_Singleton - Singleton 实现

- 另一个简单的 Singleton 实现（非线程安全版本）

```
class Singleton2 {
public:
    static Singleton2& instance() {
        static Singleton2 obj;
        return obj;
    }
    // 其它成员函数
private:
    // 将以下函数置为private，且简单实现它们
    Singleton2() {
    }

    Singleton2(const Singleton2& s) {
    }

    Singleton2& operator=(const Singleton2& s) {
        return *this;
    }
};
```

ACE_Singleton - Singleton 实现

- Double-Checked Locking Singleton 实现（线程安全）

```
class Singleton {
public:
    static Singleton* instance() {
        if (0 == _instance) {
            boost::lock_guard<boost::mutex> lock(m);
            if (0 == _instance) _instance = new Singleton;
        }
        return _instance;
    }
    // 其它成员函数
private:
    static boost::mutex m;
    static Singleton* volatile _instance;
    // 将以下函数置为private
    Singleton() { }
    Singleton(const Singleton& s) { }
    Singleton& operator=(const Singleton& s) { return *this; }
};
Singleton* volatile Singleton::_instance = 0;
```

ACE_Singleton - Singleton 实现

- 简单的检验代码

```
int main(int argc, char* argv[]) {
    // 查看各个 instance 的地址，如果地址相同则表示为同一个对象
    Singleton* s1 = Singleton::instance();
    Singleton* s2 = Singleton::instance();
    cout << "address: " << s1 << " " << s2 << endl;

    Singleton2& s3 = Singleton2::instance();
    Singleton2& s4 = Singleton2::instance();
    cout << "address: " << &s3 << " " << &s4 << endl;
}
```

ACE_Singleton - 接口

- ACE_Singleton 的接口

```
template<class TYPE, class ACE_LOCK>
class ACE_Singleton: public ACE_Cleanup {
public:
    /// Global access point to the Singleton.
    static TYPE *instance(void);

    /// Cleanup method, used by <ace_cleanup_destroyer>
    /// to destroy the ACE_Singleton.
    virtual void cleanup(void *param = 0);

    /// Dump the state of the object.
    static void dump(void);
```

ACE_Singleton - 接口

- ACE_Singleton 的接口（续）

```
protected:
    /// Default constructor.
    ACE_Singleton(void);

    /// Contained instance.
    TYPE instance_;

#if !defined (ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES)
    /// Pointer to the Singleton (ACE_Cleanup) instance.
    static ACE_Singleton<TYPE, ACE_LOCK> *singleton_;
#endif /* ACE_LACKS_STATIC_DATA_MEMBER_TEMPLATES */

    /// Get pointer to the Singleton instance.
    static ACE_Singleton<TYPE, ACE_LOCK> *&instance_i(void);
};
```

ACE_Singleton - 使用

- ACE_Singleton 使用示例

```
#include <ace/Singleton.h>
class Participant;

class ChatRoom {
public:
    void join(Participant* user);
    void leave(Participant* user);
    void forwardMsg(const char* msg);
private:
    std::list<Participant*> users;
};
// 不加锁的方式
typedef ACE_Singleton<ChatRoom, ACE_Null_Mutex> Room;
// 加锁的方式
typedef ACE_Singleton<ChatRoom, ACE_Recursive_Thread_Mutex>
Room;
```

```
    Room::instance()->forwardMsg("hello");
```