第 3 章 系统基础构件

3.1 日志模块

日志模块其实很重要，一是记录系统运行时的一举一动，排查问题、调试程序、性能分析、风险监控，很多时候都靠日志，二是对于特指的交易日志，可以用来记录系统崩溃瞬间用户的交易和账户信息，在系统重启时进行还原，确保业务正确且连续。而且日志输出调用几乎遍布整个系统，其性能好坏会影响系统整体和核心效率。

这里先讲一下普通磁盘文件日志记录。目前常用的几种 C++日志库有：boost-log、spdlog、Easylogging++等，使用：处理器 Intel Core i7-4790 3.60GHz - 4.00GHz，内存 RAM 16.0GB，硬盘 WDC WD10EZEX-08Y20A0 1.0TB，系统 Windows 10 64-bit 1709，简单测试性能如下：（条/秒）

boost-log (Boost v1.65.1)					
同步	逐条	218674	异步	逐条	825045
	逐条+追加	149670		逐条+追加	799083
	逐条+追加+换日	146165		逐条+追加+换日	784758
	逐条+换日	210724		逐条+换日	791305
	逐条+换日+追加	143926		逐条+换日+追加	780113
	缓存	416890		缓存(不太稳定)	791915
	缓存+追加	403661		缓存+追加	832063
	缓存+追加+换日	373078		缓存+追加+换日	823777
	缓存+换日	391625		缓存+换日	816682
	缓存+换日+追加	368291		缓存+换日+追加	794888

Easylogging++ (v9.95.3)(默认追加)					
同步	逐条	195870			
	缓存	10：566271	50：720240	100：710918	250：717860
异步	无法运行				

spdlog (v0.16.3)(有时新建，有时追加)			
同步	单线程	逐条	240698
		缓存	1244076
	多线程	逐条	216989
		缓存	1372567
	引用 log_info && 去掉 Format		
	单线程	逐条	249909
		缓存	1500000
	多线程	逐条	245553
		缓存	1500000
异步	单线程(4096)	逐条	182642 - 67505
		缓存	568000 - 311162
	单线程(8192)	逐条	204872 - 139790
		缓存	880327 - 332118
	多线程(8192)	逐条	882270 - 252547
		缓存	967300 - 340116
	缓存满了则阻塞，设 10240 字节及以上会崩溃		

不同日志库的架构和功能不同，进而导致性能也不尽相同，但架构越复杂功能越多肯定会导致性能下降。在功能相对单一模式相对固定的前提下，化简去繁，BasicX 库中的 SysLog 日志模块性能如下：（条/秒）

SysLog (BasicX v1.0.2)			
默认文件缓冲方式和大小 且 SetActiveSync(false) 且 动态链接			
Windows 10 x64 1709	逐条写入	单个线程	388255
		线程安全	377856
	系统缓存	单个线程	1233499
		线程安全	1186223
	本地缓存	单个线程	11139005
			静态链接只能略超 7000000
CentOS 7 x64 3.10	逐条写入	单个线程	736414
	系统缓存	单个线程	2729314
	本地缓存	单个线程	9590867
			静态链接性能差别不大

3.2 网络模块

文字。

文字。

3.3 插件模块

文字。

文字。

3.4 消息队列

3.4.1 链表有锁队列

链表有锁队列通过 Boost 中 Asio[01]的内置消息队列及 io_service 的 post 异步调用实现，比较简单，示例数据结构和成员方法如下：

```
struct DemoData {
    std::string m_data;
};

class DemoClass {
public:
    void CreateService();
    void AssignTask( std::string& data );
    void HandleTask();
public:
    std::shared_ptr<std::thread> m_thread;
    std::shared_ptr<boost::asio::io_service> m_service;
    std::mutex m_list_data_lock;
    std::list<DemoData> m_list_data;
```

```
};
```

首先使用独立的线程调用 CreateService()方法启动 Asio 服务：

```
void DemoClass::CreateService() {
    m_service = std::make_shared<boost::asio::io_service>();
    boost::asio::io_service::work work( *m_service );
    m_thread = std::make_shared<std::thread>( std::bind( static_cast<std::
               size_t( boost::asio::io_service::* )( )>( &boost::asio::
               io_service::run ), m_service ) );
    m_thread->join();
}
```

输入消息数据，数据队列为空说明消息处理方法 HandleTask()自循环已结束，需要调用 m_service->post()再次唤醒进行消息处理自循环：

```
void DemoClass::AssignTask( std::string& data ) {
    m_list_data_lock.lock();
    bool write_in_progress = !m_list_data.empty();
    DemoData demo_data;
    demo_data.m_data = data;
    m_list_data.push_back( demo_data );
    m_list_data_lock.unlock();
    if( !write_in_progress ) {
        m_service->post( std::bind( &DemoClass::HandleTask, this ) );
    }
}
```

输出消息数据，数据队列不空就调用 m_service->post()继续处理消息数据：

```
void DemoClass::HandleTask() {
    DemoData* demo_data = &m_list_data.front(); // 肯定存在
    // 输出消息数据
    m_list_data_lock.lock();
    m_list_data.pop_front();
    bool write_on_progress = !m_list_data.empty();
    m_list_data_lock.unlock();
    if( write_on_progress ) {
        m_service->post( std::bind( &DemoClass::HandleTask, this ) );
```

```
        }
    }
```

在 Asio 文档 chat_client.cpp[02]示例中，do_write()和 handle_write()方法并没有给 std::deque<chat_message>类型的 write_msgs_缓存结构加锁，本示例中加锁主要是出于以下考虑：

首先，存在极小的可能，HandleTask()正在输出 m_list_data 最后一个数据，AssignTask()进入判断 m_list_data 不为空，不需要调用 m_service->post()，但此时就被操作系统挂起了，HandleTask()输出完 pop_front()数据后判断 m_list_data 为空，也不调用 m_service->post()，AssignTask()恢复执行并 push_back()数据后，没调用 m_service->post()，因此整个消息处理循环不再继续，最后被 push_back()的数据留在 m_list_data 中，即使再有新的数据到来，AssignTask()进入判断 m_list_data 不为空，也不再调用 m_service->post()。缓存列表的 empty()判断与 push_back()或 pop_front()操作需要通过锁实现原子性来确保输出循环。

其次，因为 C++标准库在不同的编译器中实现方式不一样，互斥机制在不同的操作系统下实现也不一样，除非特意重写并声明是线程安全的，一般使用的标准库并不是线程安全。《Effective STL》中也要求多线程同时读写访问时调用容器的成员函数期间都要锁定该容器。比如上述代码中的 std::list 容器，在未开放源码，不知道具体实现的情况下，无法确保多线程同时调用 empty()、push_back()、pop_front()等成员函数时，临界状态内部头尾指针移动是否会产生冲突进而导致数据异常或程序崩溃。

当然这里 AssignTask()的实现并不是很高效，TaskItem 的创建、赋值、存入容器，需要三次内存拷贝，改为在 TaskItem 构造时就直接传入赋值可以减少一次内存拷贝，如果使用 std::move 语义则有可能将内存拷贝降低到零次。

3.4.2 双子微锁队列[03]

双子微锁队列。

3.4.3 环形无锁队列[04]

环形无锁队列。

注释说明

<01>. https://www.boost.org/doc/libs/1_73_0/doc/html/boost_asio.html
<02>. https://www.boost.org/doc/libs/1_73_0/doc/html/boost_asio/example/

cpp03/chat/chat_client.cpp

〈03〉. 本小节所述方法均为自主原创，如需转载请务必联系作者并注明出处。

〈04〉. 本小节所述方法均为自主原创，如需转载请务必联系作者并注明出处。

参考文献

[01].