



下载APP



## 21 | 知识串讲（上）：带你开发一个书店应用

2020-06-23 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：**Chrono**

时长 12:53 大小 11.81M



你好，我是 Chrono。

到今天为止，课程里的 C++ 知识就全部讲完了。前面我们总共学习了四大模块，我再带你做一个简略的回顾。

在“概论”单元，我带你从宏观的层面上重新认识了 C++，讲了它的四个生命周期和五个编程范式，分别介绍了在编码阶段、预处理阶段、编译阶段，C++ 能够做哪些事情，接着又重点说了在 C++ 里，运用哪些特性才能更好地实践面向对象编程。



在“语言特性”单元，我们一起研究了自动类型推导、常量、智能指针、异常、函数式编程这五个特性。这些特性是“现代” C++ 区别于“传统” C++ 的关键，掌握了它们，你就能够写出清晰、易读、安全的代码。

在“标准库”单元，我介绍了字符串、容器、算法和并发。它们是 C++ 标准库中最核心的部分，也是现代 C++ 和泛型编程的最佳应用和范例。学会了标准库，你才能说是真正理解了 C++。

在“技能进阶”单元，我为你挑选出了一些第三方工具，包括序列化、网络通信、脚本语言和性能分析，它们很好地补充完善了 C++ 语言和标准库，免去了我们“自己造轮子”的麻烦，让我们把精力集中在实现业务逻辑上。

除了上面的这“十八般武艺”，我还谈了能够帮你更好地运用 C++ 的设计模式和设计原则，介绍了几个比较重要、常用的模式，希望你在今后的实际开发工作中，能够有意识地写出灵活、可扩展的代码。

这么回顾下来，内容还真是不少啊。

为了让你更好地把这些知识融会贯通，接下来我会再用两节课的时间，从需求、设计，到开发编码、编译运行，再加上一些我自己的实用小技巧，详细讲解一个 C++ 程序的实际开发过程，把知识点都串联起来。

虽然说是“串讲”，但是你只要学过了前面的内容，就可以跟着我做出这个书店程序。不过，我担心有些知识点你可能忘记了，所以，涉及到具体的知识点时，我会给你标注出是在哪一节，你可以随时回去复习一下。

## 项目设计

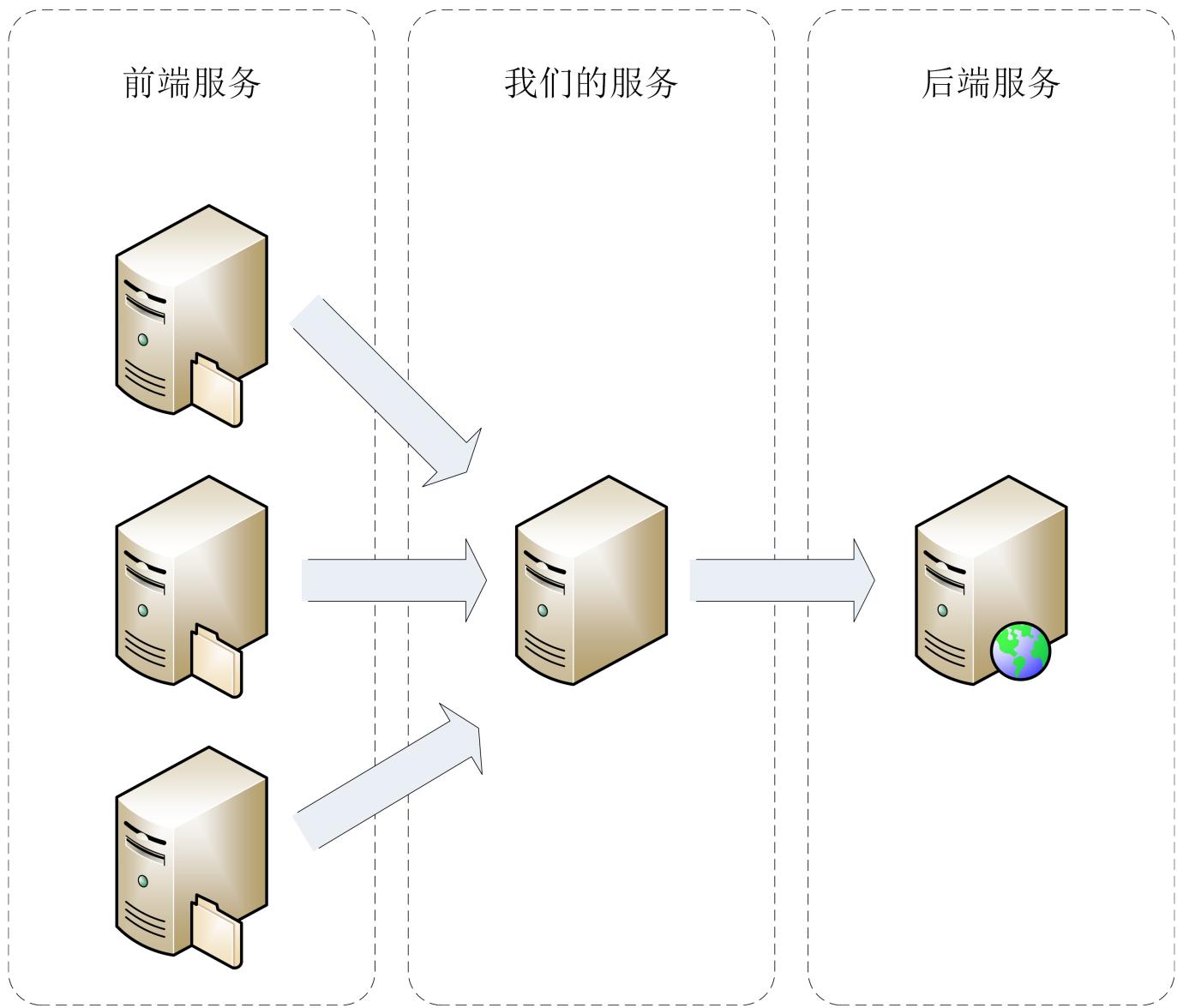
那么，该用个什么样的例子来串讲 C++ 的这些知识点呢？

说实话，找出一个合适的例子真的很难。因为大多数 C++ 实际项目都很大、很底层，还有各种依赖或者内部库，不好直接学习研究。

所以我再三考虑，决定借鉴一下 *C++ Primer* 里的书店例子，修改一下它的需求，然后完全重新开发，作为我们这个课程的综合示例。

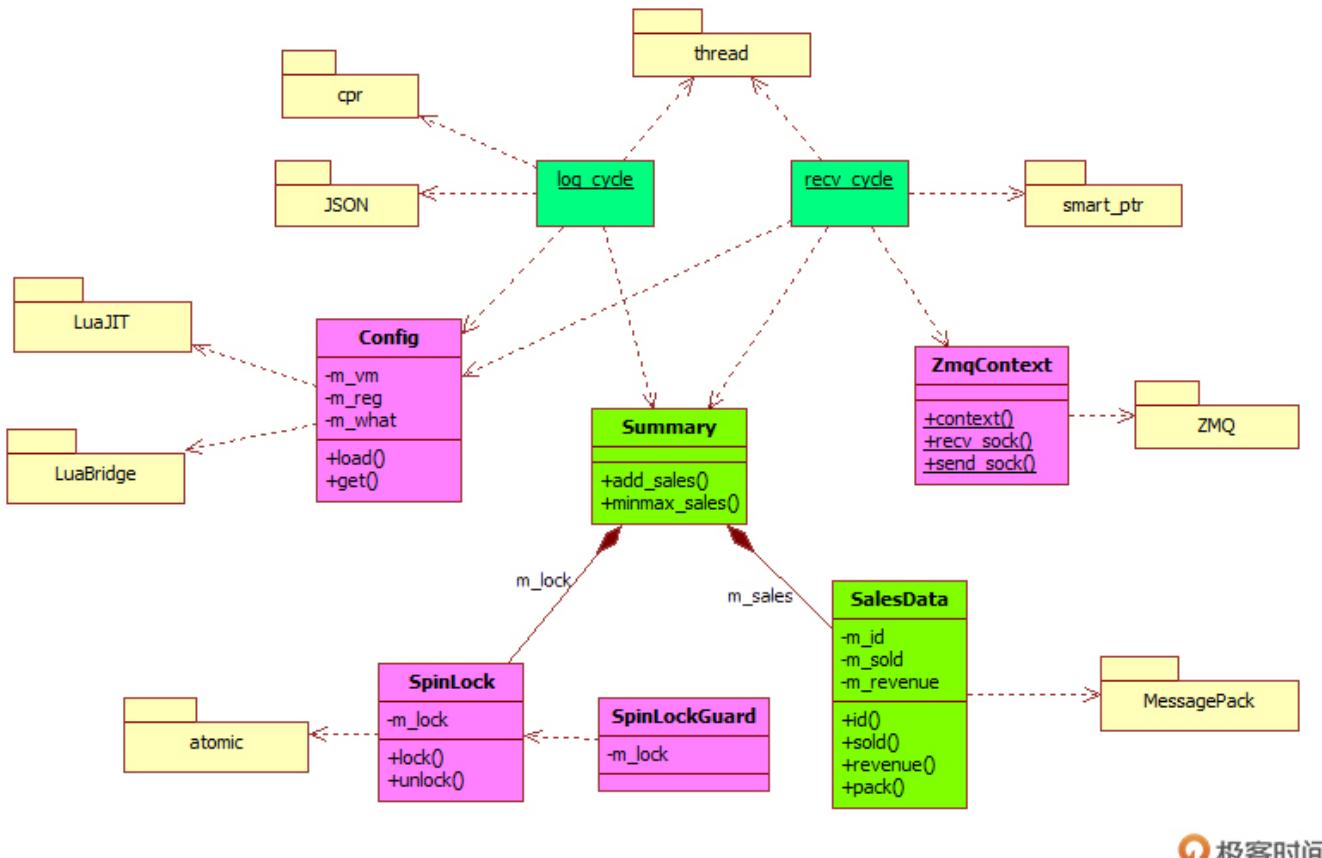
先介绍一下咱们这个书店程序。简单来说，就是销售记录管理，从多个渠道把书号、销售册数、销售额都汇总起来，做个统计分析，再把数据定期上报到后台。

*C++ Primer* 里的书店程序是本地运行的，为了演示课程里讲到的 C++ 特性，我把它改成了网络版。不过，拓扑结构并不复杂，我画了张图，你可以看一下。



项目的前期需求就算是定下来了，接着就要开始做设计了，这就要用到设计模式和设计原则的知识了（[第 19 讲](#)、[第 20 讲](#)）。

不过这个系统还是比较简单的，不需要用什么复杂的分析手段，就能够得出设计，主要应用的是单一职责原则、接口隔离原则和包装外观模式。这里我也画了一个 UML 图，可以帮助你理解程序的架构。



极客时间

下面我就对照这个 UML 类图，结合开发思路和源码，仔细说一下具体的 C++ 开发，完整的源码都放在了 [GitHub](#) 上，课下可以仔细地看一下。

## 核心头文件

首先要说的是我写 C++ 项目的一个习惯，定义核心头文件：**cpplang.hpp**。它集中了 C++ 标准头和语言相关的定义，被用于其他所有的源文件。

注意，在写它的時候，最好要有文件头注释（[第 2 讲](#)），而且要有“Include guard”（[第 3 讲](#)），就像下面这样：

```

1 // Copyright (c) 2020 by Chrono
2
3 #ifndef _CPP_LANG_HPP          // Include guard
4 #define _CPP_LANG_HPP          // Include guard
5
6 #include <cassert>            // C++标准头文件
7 ...
8
9 #endif // _CPP_LANG_HPP

```

复制代码

在核心头文件里，我们还可以利用预处理编程，使用宏定义、条件编译来屏蔽操作系统、语言版本的差异，增强程序的兼容性。

比如，这里我就检查了 C++ 的版本号，然后定义了简化版的“deprecated”和“static\_assert”：

 复制代码

```
1 // must be C++11 or later
2 #if __cplusplus < 201103
3 #   error "C++ is too old"
4 #endif // __cplusplus < 201103
5
6 // [[deprecated]]
7 #if __cplusplus >= 201402
8 #   define CPP_DEPRECATED [[deprecated]]
9 #else
10 #   define CPP_DEPRECATED [[gnu::deprecated]]
11 #endif // __cplusplus >= 201402
12
13 // static_assert
14 #if __cpp_static_assert >= 201411
15 #   define STATIC_ASSERT(x) static_assert(x)
16 #else
17 #   define STATIC_ASSERT(x) static_assert(x, #x)
18 #endif
```

## 自旋锁

有了核心头文件之后，我们的 C++ 程序就有了一个很好的起点，就可以考虑引入多线程，提高吞吐量，减少阻塞。

在多线程里保护数据一般要用到互斥量（Mutex），但它的代价太高，所以我设计了一个自旋锁，它使用了原子变量，所以成本低，效率高（[第 14 讲](#)）。

自旋锁被封装为一个 SpinLock 类，所以就要遵循一些 C++ 里常用的面向对象的设计准则（[第 5 讲](#)、[第 19 讲](#)），比如用 final 禁止继承、用 default/delete 显式标记构造 / 析构函数、成员变量初始化、类型别名，等等，你可以看看代码：

 复制代码

```
1 class SpinLock final // 自旋锁类
```

```
2 {
3     public:
4         using this_type    = SpinLock;           // 类型别名
5         using atomic_type = std::atomic_flag;
6     public:
7         SpinLock() = default;                 // 默认构造函数
8         ~SpinLock() = default;
9
10        SpinLock(const this_type&) = delete; // 禁止拷贝
11        SpinLock& operator=(const this_type&) = delete;
12     private:
13         atomic_type m_lock {false};          // 成员变量初始化
14
```

在编写成员函数的时候，为了尽量高效，需要给函数都加上 noexcept 修饰，表示绝不会抛出异常（[第 9 讲](#)）：

 复制代码

```
1 public:
2     void lock() noexcept           // 自旋锁定，绝不抛出异常
3     {
4         for(;;) {                // 无限循环
5             if (m_lock.test_and_set()) { // 原子变量的TAS操作
6                 return;            // TAS成功则锁定
7             }
8             std::this_thread::yield(); // TAS失败则让出线程
9         }
10    }
11
12    void unlock() noexcept       // 解除自旋锁定，绝不抛出异常
13    {
14        m_lock.clear();
15    }
```

为了保证异常安全，在任何时候都不会死锁，还需要利用 RAII 技术再编写一个 LockGuard 类。它在构造时锁定，在析构时解锁，这两个函数也应该用 noexcept 来优化：

 复制代码

```
1 class SpinLockGuard final           // 自旋锁RAII类，自动解锁
2 {
3     public:
4         using this_type    = SpinLockGuard; // 类型别名
5         using spin_lock_type = SpinLock;
6     public:
7         SpinLockGuard(const this_type&) = delete; // 禁止拷贝
```

```
8     SpinLockGuard& operator=(const this_type&) = delete;
9 public:
10    SpinLockGuard(spin_lock_type& lock) noexcept
11        : m_lock(lock)
12    {
13        m_lock.lock();
14    }
15
16    ~SpinLockGuard() noexcept
17    {
18        m_lock.unlock();
19    }
20 private:
21    spin_lock_type& m_lock;
22 ..
```

这样自旋锁就完成了，有了它就可以在多线程应用里保护共享的数据，避免数据竞争。

## 网络通信

自旋锁比较简单，但多线程只是书店程序的基本特性，它的核心关键词是“网络”，所以下面就来看看服务里的“重头”部分：网络通信。

正如我之前说的，在现代 C++ 里，应当避免直接使用原生 Socket 来编写网络通信程序（[第 16 讲](#)）。这里我选择 ZMQ 作为底层通信库，它不仅方便易用，而且能够保证消息不丢失、完整可靠地送达目的地。

程序里使用 ZmqContext 类来封装底层接口（包装外观），它是一个模板类，整数模板参数用来指定线程数，在编译阶段就固定了 ZMQ 的多线程处理能力。

对于 ZMQ 必需的运行环境变量（单件），我使用了一个小技巧：**以静态成员函数来代替静态成员变量**。这样就绕过了 C++ 的语言限制，不必在实现文件 “\*.cpp” 里再写一遍变量定义，全部的代码都可以集中在 hpp 头文件里：

 复制代码

```
1 template<int thread_num = 1>           // 使用整数模板参数来指定线程数
2 class ZmqContext final
3 {
4 public:
5     static                         // 静态成员函数代替静态成员变量
6     zmq_context_type& context()
```

```
7     {
8         static zmq_context_type ctx(thread_num);
9         return ctx;
10    }
11
```

然后，我们要实现两个静态工厂函数，创建收发数据的 Socket 对象。

这里要注意，如果你看 `zmq.hpp` 的源码，就会发现，它的内部实际上是使用了异常来处理错误的。所以，这里我们不能在函数后面加上 `noexcept` 修饰，同时也就意味着，在使用 ZMQ 的时候，必须要考虑异常处理。

 复制代码

```
1 public:
2     static
3     zmq_socket_type recv_sock(int hwm = 1000) // 创建接收Socket
4     {
5         zmq_socket_type sock(context(), ZMQ_PULL); // 可能抛出异常
6
7         sock.setsockopt(ZMQ_RCVHWM, hwm);
8
9         return sock;
10    }
11
12     static
13     zmq_socket_type send_sock(int hwm = 1000) // 创建发送Socket
14     {
15         zmq_socket_type sock(context(), ZMQ_PUSH); // 可能抛出异常
16
17         sock.setsockopt(ZMQ SNDHWM, hwm);
18
19         return sock;
20     }
```

现在，有了 `ZmqContext` 类，书店程序的网络基础也就搭建出来了，后面就可以用它来收发数据了。

## 配置文件解析

接下来，我要说的是解析配置文件的类 `Config`。

大多数程序都会用到配置文件来保存运行时的各种参数，常见的格式有 INI、XML、JSON，等等。但我通常会选择把 Lua 嵌入 C++，用 Lua 脚本写配置文件（[🔗 第 17 讲](#)）。

这么做的好处在哪里呢？

Lua 是一个完备的编程语言，所以写起来就非常自由灵活，比如添加任意的注释，数字可以写成 “ $m \times n$ ” 的运算形式。而 INI、XML 这些配置格式只是纯粹的数据，很难做到这样，很多时候需要在程序里做一些转换工作。

另外，在 Lua 脚本里，我们还能基于 Lua 环境写一些函数，校验数据的有效性，或者采集系统信息，实现动态配置。

总而言之，就是把 Lua 当作一个“可编程的配置语言”，让配置“活起来”。

给你看一下配置文件的代码吧，里面包含了几个简单的值，配置了服务器的地址、时间间隔、缓冲区大小等信息：

 复制代码

```
1 config = {  
2  
3     -- should be same as client  
4     -- you could change it to ipc  
5     zmq_ipc_addr = "tcp://127.0.0.1:5555",  
6  
7     -- see http_study's lua code  
8     http_addr = "http://localhost/cpp_study?token=cpp@2020",  
9  
10    time_interval = 5,    -- seconds  
11  
12    max_buf_size = 4 * 1024,  
13 }
```

Config 类使用 shared\_ptr 来管理 Lua 虚拟机（[🔗 第 17 讲](#)），因为封装在类里，所以，你要注意类型别名和成员变量初始化的用法（[🔗 第 5 讲](#)）：

 复制代码

```
1 class Config final // 封装读取Lua配置文件
```

```
2 {
3 public:
4     using vm_type      = std::shared_ptr<lua_State>;    // 类型别名
5     using value_type   = luabridge::LuaRef;
6 public:
7     Config() noexcept           // 构造函数
8     {
9         assert(m_vm);
10        luaL_openlibs(m_vm.get()); // 打开Lua基本库
11    }
12     ~Config() = default;       // 默认析构函数
13 private:
14     vm_type      m_vm          // 类型别名定义Lua虚拟机
15     {luaL_newstate(), lua_close}; // 成员变量初始化
16 }
```

加载 Lua 脚本的时候还要注意一点，外部的脚本有可能会写错，导致 Lua 解析失败。但因为这个问题极少出现，而且一出现就很严重，没有配置就无法走后续的流程，所以非常适合用异常来处理（[第 9 讲](#)）。

`load()` 函数不会改变虚拟机成员变量，所以应该用 `const` 修饰，是一个常函数：

 复制代码

```
1 public:
2     void load(string_view_type filename) const // 解析配置文件
3     {
4         auto status = luaL_dofile(m_vm.get(), filename.c_str());
5
6         if (status != 0) {                      // 出错就抛出异常
7             throw std::runtime_error("failed to parse config");
8         }
9     }
```

为了访问 Lua 配置文件里的值，我决定采用 “key1.key2” 这样简单的两级形式，有点像 INI 的小节，这也正好对应 Lua 里的表结构。

想要解析出字符串里的前后两个 key，可以使用正则表达式（[第 11 讲](#)），然后再去查询 Lua 表。

因为构造正则表达式的成本很高，所以我把正则对象都定义为成员变量，而不是函数里的局部变量。

正则的匹配结果 (`m_what`) 是“临时”的，不会影响常量性，所以要给它加上 `mutable` 修饰。

 复制代码

```
1 private:
2     const regex_type m_reg {R"^(?(\w+)\.(\w+)$)"};
3     mutable match_type m_what; // 注意是mutable
```

在 C++ 正则库的帮助下，处理字符串就太轻松了，拿到两个 key，再调用 `LuaBridge` 就可以获得 `Lua` 脚本里的配置项。

不过，为了进一步简化客户代码，我把 `get()` 函数改成了模板函数，显式转换成 `int`、`string` 等 C++ 标准类型，可读性、可维护性会更好。

 复制代码

```
1 public:
2     template<typename T> // 转换配置值的类型
3     T get(string_view_type key) const // const常函数
4     {
5         if (!std::regex_match(key, m_what, m_reg)) { // 正则匹配
6             throw std::runtime_error("config key error"); // 格式错误抛异常
7         }
8
9         auto w1 = m_what[1].str(); // 取出两个key
10        auto w2 = m_what[2].str();
11
12        auto v = getGlobal( // 获取Lua表
13            m_vm.get(), w1.c_str());
14
15        return LuaRef_cast<T>(v[w2]); // 取表里的值，再做类型转换
16    }
```

到这里呢，`Config` 类也就完成了，可以轻松解析 `Lua` 格式的配置文件。

## 小结

今天，我用一个书店程序作为例子，把前面的知识点都串联起来，应用到了这个“半真实”的项目里，完成了 UML 类图里的外围部分。你也可以把刚才说的核心头文件、自旋锁、`Lua` 配置文件这些用法放到自己的实际项目里去试试。

简单小结一下今天的内容：

1. 在项目起始阶段，应该认真做需求分析，然后应用设计模式和设计原则，得出灵活、可扩展的面向对象系统；
2. C++ 项目里最好要有一个核心头文件（cpplang.hpp），集中定义所有标准头和语言特性，规范源文件里的 C++ 使用方式；
3. 使用原子变量（atomic）可以实现自旋锁，比互斥量的成本要低，更高效；
4. 使用 ZMQ 可以简化网络通信，但要注意它使用了异常来处理错误；
5. 使用 Lua 脚本作为配置文件的好处很多，是“可编程的配置文件”；
6. 在编写代码时要理解、用好 C++ 特性，恰当地使用 final、default、const 等关键字，让代码更安全、更可读，有利于将来的维护。

今天，我们分析了需求，设计出了架构，开发了一些工具类，但还没有涉及业务逻辑代码，下节课，我会带你看看容器、算法、线程，还有 lambda 表达式的实践应用，看看它们是怎么服务于具体业务的。

## 课下作业

最后是课下作业时间，给你留一个思考题：你能说出，程序里是怎么应用设计模式和设计原则的吗？

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

# 课外小贴士

1. 绘图工具能够很好地辅助软件设计，这里使用的是Visio和StarUML。
2. 在编写类的时候，反复用“= delete”禁止拷贝非常麻烦，在Boost程序库里，有一个很小的工具类noncopyable，它实现了等价的功能，直接继承就行，能够节约大量代码。
3. 正文里实现的自旋锁还不能算是严格的自旋锁，因为它TAS失败后立刻调用了yield()让出了CPU，真正的自旋锁应当再用一个循环，不停地try\_lock“自旋”。



618 课程特惠

# 618 好课 5 折起

优惠口令立减 ¥15

618gogogo



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 设计模式（下）：C++是怎么应用设计模式的？

下一篇 轻松话题（一）| 4本值得一读再读的经典好书

## 精选留言 (2)

写留言



reverse

2020-06-23

关于UML图 想学的同学可以看一下《UML大象》

展开 ▼

作者回复: uml其实并不难，也没有必要完全学通学精，本质上就是个画图。

先学会最基本的类图、时序图，其他的可以在实践中慢慢学。



1

1



泡泡龙

2020-06-23

老师能否说说需求到UML图的过程，是怎么把需求提到UML的。还有就是给一张UML图，应该怎么看？

作者回复：

1.这个就是基本的需求分析了，简单来说，就是提取出需求里的名词和动词，转化成相应的类，比如销售记录、配置文件、锁、XX主循环。

2.UML图有很多种，这里用到的是类图，表示的是类之间的关系，需要理解UML的基本语言要素，比如聚合、联系等，然后可以随便选一个类作为起点，像爬虫一样，沿着关系链接去看它相关的类，逐步去理解这些类是如何配合工作的。

