



下载APP



大咖助场 | LMOS 说 C 语言（下）：用面向对象的思想开发 C 语言程序

2022-01-03 LMOS

《深入C语言和程序运行原理》

[课程介绍 >](#)**讲述：陈晨**

时长 11:43 大小 10.75M



你好，我是 LMOS。

在“LMOS 说 C 语言”的上篇里，我向你分享了 C 语言的起源，它与 UNIX 操作系统的联系，以及与 C 语言简单和困难相关的语言特性。今天我想和你聊聊，既然 C 语言是一把锋利但考验使用者技巧的瑞士军刀，我们可以拿它做什么，又怎么才能把它用好。

[领资料](#)

C 语言能干什么？



对于“C 语言能干什么”这个问题，我的回答是：C 语言能干一切其他语言能干的事。

C 语言自身的特性使得它能完全操作计算机所有的资源，因此它**生来就是开发操作系统等底层核心软件的**。不仅仅是开发操作系统，还有一些数据库和一些其他高级编程语言的编译器、解析器等。C 语言简单的语法，被 C++、Java、Go、JS 等语言效仿。其实从 C 语言的语言特性就可以看出来，它属于高级语言中的低级语言，又是低级语言中的高级语言，能适应一切底层开发。

然而，C 语言既然能做一切底层开发，就一定能做一切上层开发，只是对开发者的能力要求远高于 Java、Go 这些高级语言。其实，所有的高级语言都有共同的目标，就是降低开发者的学习使用成本和心智负担，从而降低软件的开发和维护成本。比如，Java 用虚拟机实现了一次编译处处运行，用垃圾内存回收机制解决了程序员使用内存的困难，不用时刻担心内存没有释放。这些归根结底是为了降低软件开发成本。

对于“C 语言能干一切其他语言能干的事”这句话，你可能还有这样的疑问：C 语言是一门面向过程的编程语言，而在工程应用中，我们多使用面向对象的编程方式。用 C 语言来做现代大型软件项目，是不是不太现实？

确实，由于 C 语言函数之间的强耦合和内存的低级控制特性，在它开发大型工程项目时，如果不设计好架构和相关的编码规则，将会给项目的开发、协同和后期维护带来很多困难。但是，C 语言是一门面向过程的编程语言，并不意味着我们不能用它来实现面向对象的编程方式。接下来，我就具体讲讲，怎么用面向对象的思想开发 C 语言程序。

面向过程和面向对象的两种思想

我先带你回顾下面向过程和面向对象这两种思想，以及一些容易混淆的相关概念。

对于计算过程的不同认识，产生了不同的计算模型。基于计算模型进行分类，我们可以将语言分为命令式、函数式、面向过程、面向对象四大类。如果从程序的本质上来看，可以进一步归纳为两种：命令式语言和说明式语言。

面向过程是命令式语言的主要实现手段，而面向对象是当前应用编程领域中最常用的语言类型。但是，无论从语言定义还是数据抽象发展来看，面向对象都是面向过程的衍生。

命令式这个词太过于学术化了，其实我们常见的编程语言，从汇编到 C 再到 C++、Java，都是命令式语言。命令式语言在很大程度上受到了“冯·诺依曼”计算体系的影响。

这个体系又以“存储”和“处理”为核心，其中存储被抽象为**内存**，处理被抽象为**运算指令和语句**。于是，命令式的核心就是**通过运算去改变内存（数据）**。

听到这里，你应该能把这些概念的关系理清楚了：面向过程 / 面向对象这些概念，和命令式并不在同一个维度上。前者是运算类型，表现为语言；后者着重表达的则是程序设计和开发的方法。

C 语言和 C++ 都是命令式语言，不过 C 语言是面向过程的语言，C++ 是面向对象的语言，那么面向过程和面向对象有什么区别呢？其实它们大同小异，只不过是“思考问题的方式”不同。为了方便你理解，这里我用“吃饭”来类比。

过程是对每个功能或者动作的精确实现。用“吃饭”来举例子：吃饭这个“功能”，包含怎么吃，吃多少。小猫能吃饭，人也能吃饭，但二者吃饭的“过程”肯定有区别。这个逻辑可以用下面的代码来描述：

[复制代码](#)

```
1 void cateat(cat* v) {
2     // 吃饭；
3     return;
4 }
5 void peopleeat(people* v) {
6     // 吃饭；
7     return;
8 }
9 int main(int argc, char const *argv[]) {
10     cateat(cat);
11     peopleeat(people);
12     return 0;
13 }
```

至于“面向对象”里的“对象”，可以这么理解：猫和人分别是两个对象，这两个对象都包含吃饭这个动作。对于人，会调用人的吃饭动作的函数；而对于猫，则会调用属于猫的吃饭动作。代码如下：

[复制代码](#)

```
1 class Cat { // 对象猫；
2     public:
3     void eat();
4 };
```

```
5 void Cat::eat(void) {
6     // 猫吃饭；
7     return;
8 }
9 class People { // 对象人；
10 public:
11     void eat();
12 };
13 void People::eat(void) {
14     // 人吃饭；
15     return;
16 }
17 int main(int argc, char const *argv[]) {
18     Cat c;
19     People p;
20     c.eat(); // 调用猫对象的吃饭动作；
21     p.eat(); // 调用人对象的吃饭动作；
22     return 0;
23 }
```

我们可以看到，面向过程和面向对象的思考方式截然不同。面向过程，是对每个不同动物的吃饭过程进行精确描述。而面向对象的思考方式却不同：认为猫和人是两个不同的对象，都有吃饭的动作，各自对吃饭这个动作进行封装和实现。最后，用对象自己调用自己的方法，完成相应的吃饭动作。

但仔细思考一下，猫和人其实都属于哺乳动物，哺乳动物间还是有一些共性的。那么，如何表示这种父子范畴关系呢？答案就是**在封装的基础上进行继承操作**。

这是因为，如果仅仅是把属性和方法封装成对象，这个意义还不是很大。封装是为了继承，继承是为了解耦和复用。当然，随着工程复杂度的发展，人们发现传统的单 / 多继承又会带来额外的复杂度，于是就又有了组合优于继承的思想，这里就不展开了。接下来让我们看看，怎么用 C 语言来实现封装和继承这两种面向对象编程的特性。

基于 C 语言的面向对象编程

首先，请一定要记住：面向对象是一种编程思想，并非特定语言（如 C++、Java）实现的功能。C++、Java 这些语言只是用语言的文法对这种思想进行规约，达到方便或者强制编程人员用面向对象的思想实现自己的代码逻辑的目的。

所以，我们不仅能用 C++、Java 这些“面向对象的编程语言”实现面向对象编程，用 C 或者汇编也可以实现，只是后者没有提供类似 C++、Java 中，可用于实现面向对象的语法

糖而已。下面，我们就一起用 C 来实现面向对象的编程方法。

封装

我们首先用 C 语言来实现封装。封装是面向对象中最基础的思想，即把一些属性和方法组织在一起，形成一个对象。


接下来，我会用我的课程 [《操作系统实战 45 讲》](#) 中的 Cosmos 的锁实现为实例，剖析用 C 语言来实现封装的方法。在操作系统中，用锁的模块很多，进程模块要用锁，内存模块也要用锁，它们对锁的要求也有不同。现在我们来封装最基本的锁，代码如下：

[复制代码](#)

```
1 typedef struct SPINLOCK {
2     volatile U32 Lock; // int 类型, 0 表示解锁, 1 表示加锁;
3 } SpinLock;
4 // 加锁;
5 void HalSpinLock(SpinLock *lock) {
6     __asm__ __volatile__(
7         "1: lock; xchg %0, %1\n"
8         "cml $0, %0\n"
9         "jnz 2f\n"
10        ".section .spinlock.text, \"\"ax\""
11        "2: cml $0, %1\n"
12        "jne 2b\n"
13        "jmp 1b\n"
14        ".previous\n"
15        :: "r"(1), "m"(*lock));
16    return;
17 }
18 // 解锁;
19 void HalUnSpinLock(SpinLock *lock) {
20     __asm__ __volatile__(
21         "movl $0, %0\n"
22         :: "m"(*lock));
23    return;
24 }
```

上面的代码中，我们定义了一个 SpinLock 数据结构，并且围绕这个结构写好了两个函数，一个加锁，一个解锁。其中，代码是用嵌入汇编实现的，这里我们不用管它们的实现，只需明白它们能加锁和解锁就行了。

不过，仅仅是这样还体现不出封装的意义，我们继续修改代码：

 复制代码

```
1 typedef struct MLOCK {
2     SpinLock Locks; // 锁;
3     UInt Count; // 计数器;
4     void (*MLocked)(MLock* Lock); // 加锁函数指针;
5     void (*MUnLock)(MLock* Lock); // 解锁函数指针;
6 } MLock;
7 // 初始化;
8 void MLockInit(MLock* init) {
9     SpinLockInit(&init->Locks);
10    init->Count = 0;
11    init->MLocked = KrlMmLocked;
12    init->MUnLock = KrlMmUnLock;
13    return;
14 }
15 // 加锁;
16 void KrlMmLocked(MLock* lock) {
17     HalSpinLock(&lock->Locks); // 调用基类加锁函数;
18     lock->Count++;
19     return;
20 }
21 // 解锁;
22 void KrlMmUnLock(MLock* lock) {
23     HalUnSpinLock(&lock->Locks); // 调用基类解锁函数;
24     lock->Count--;
25     return;
26 }
27 MLock Lock; // 定义一个 Lock 对象;
28 MLockInit(&Lock); // 初始化对象;
29 Lock.MLocked(&Lock); // 调用对象方法;
30 Lock.MUnLock(&Lock);
```

上述代码中，MLock 结构中的 SpinLock 相当于基类，并且扩展封装了一个计数器和两个成员方法，形成了新的 MLock 锁。

MLock 锁封装了底层锁的实现机制，使用者不用考虑底层实现，在任何需要使用 MLock 的地方，只要定义一个 MLock 类型的对象，并对其初始化，需要的时候调用其中对应的方法就行了。你看，是不是有点 C++ 的味道了？只是 C++ 用语法糖包装了这些实现细节，而 C 语言的语法上没有 new，没有 class，也没有构造函数。但是 C 语言有 struct，有函数指针，可以自己写初始化函数。

继承

在面向对象的编程思想中，把属性和方法封装成一个个对象是为了继承。若非如此，就失去了封装对象的意义。上面的 MLock 只是封装加上简单的继承，下面我们来看看复杂点的继承。

在操作系统中，内存管理需要很多数据结构（如果你想深入了解这些，可以看看我在极客时间的课程 [《操作系统实战 45 讲》](#)），内存管理的各种数据结构都需要锁来避免程序并发运行带来的破坏性结果。下面，我用其中几个结构作为实例，示范一下“继承”这个概念。

先来看这段代码：

[复制代码](#)

```
1 // 物理地址块头链；
2 typedef struct PABHLIST {
3     MLock Lock; // 锁对象；
4     U32 Status;
5     UInt Order;
6     UInt InOrderPmsadNR;
7     UInt FreePmsadNR;
8     UInt PmsadNR;
9     UInt AllocCount;
10    UInt FreeCount;
11    List FreeLists;
12    List AllocLists;
13    List OveLists;
14 } PABHList;
15 // 内存拆分合并结构；
16 typedef struct MSPLITMER {
17     MLock Lock; // 锁对象；
18     U32 Status;
19     UInt MaxSMNR;
20     UInt PhySMNR;
21     UInt PreSMNR;
22     UInt SPlitNR;
23     UInt MerNR;
24     PABHList PAddrBlockArr[MSPLMER_ARR_LMAX];
25     PABHList OnePAddrBlock;
26 } MSplitMer;
27 // 内存节点；
28 typedef struct MNode {
29     List Lists;
30     MLock Lock; // 锁对象；
31     UInt Status;
32     UInt Flags;
33     UInt NodeID;
34     UInt CPUID;
```



```
35 Addr NodeMemAddrStart;
36 Addr NodeMemAddrEnd;
37 PHYMSpaceArea* PMSAreaPtr;
38 U64 PMSAreaNR;
39 U64 NodeMemSize;
40 Addr NodeMemResvAddrStart;
41 Addr NodeMemResvAddrEnd;
42 U64 NodeMemResvSize;
43 MArea MAreaArr[MEMAREA_MAX];
44 PMSADDireArr PMSADDir;
45 } MNode;
```

上面的三个数据结构，都需要用锁来保护其自身数据的完整体，避免并发访问带来的各种问题。要访问，先加锁，一旦加锁别人就无法访问了，这样就能保证数据是安全访问的，不会读取到状态不一致的数据。

你可能想问：难道我们要每种数据结构都写一套加锁、解锁的代码吗？当然不是。我们只需要在其他结构里包括这个 MLock 就行了，相当于继承 MLock 类。这样，我们就可以访问结构时先调用 MLock 中的加锁操作。例如：

[复制代码](#)

```
1 MNode node;
2 MnodeInit(&node);
3
4 node.Lock.MLocked(&node.Lock);
5 node.PMSAreaNR++;
6 node.Lock.MUnLock(&node.Lock);
```

从这段代码中我们看到，操作 MNode 中的数据，首先会调用 MNode 下的 Lock 中的加锁操作，然而这个 Lock 对象是继承于 MLock 类的。

由此我们可以发现，任何数据结构只要包含（继承）MLock 类，就可以具有锁的功能了，而不用知道锁是如何实现的。并且，我们如果需要移植代码到不同的机器上，只用改动 MLock 中 SpinLock 的实现，就好了。这种高内聚，低耦合的状态，正是衡量软件工程设计是否优良的重要指标。

我们在工作中，不仅仅是要追求代码运行正常与否，更要在这个基础之上追求代码的可读性、可维护性、软件架构的优雅性。正如我们看到的这样，面向过程和面向对象是两种不

同的编程设计思想。我们可以取其优势以用之，把这两种思想融会贯通，这样就能用面向过程的编程语言实现面向对象的编程方法。这就像剑法大成的独孤求败一样，已经不在乎用什么剑了。在他眼里，草木竹石均可为剑，以至于能达到更高的境界，“无剑胜有剑”。

重点回顾

今天的分享就到这里了，最后我来给你总结一下。

1. 首先，我们了解了 C 语言能干什么：从操作系统到编译器，从数据库到应用软件，C 语言都可以非常高效地实现它们。
2. 然后，我们对比了面向过程和面向对象这两种不同编程思想的“思考方式”。
3. 既然面向对象是一种编程思想，那么用 C 语言这种面向过程的编程语言也可以实现。最后，我们通过大量的实例，实现了面向对象的封装和继承特性。

写在最后

C 语言是一把利剑，用好了威力无穷。如果想最大程度地发挥它的威力，我们还需要把它跟工程实践相结合，可以尝试用 C 语言开发一个工程，比如操作系统、数据库等。

今天我举的实例仅仅是一个数据结构，如果稍微扩展一下思维，你就会发现：一个 C 语言模块文件就是一个对象类，其中的数据结构和函数就是这个对象类的成员数据和成员方法。在 [《操作系统实战 45 讲》](#) 这门课中，我带同学们实现了一个基于 x86 平台的 64 位多进程的操作系统——Cosmos。Cosmos 中的 CPU 类、MMU 类、List 类、RBTree 类、Atomic 类、Queue 类等，都是用 C 语言基于面向对象的思想实现的。总之，如果你想看到更多 C 语言在操作系统中应用的案例，我在这门课里等你。

感谢你看到这里，如果今天的内容让你有所收获，欢迎把它分享给你的朋友。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 2

 提建议

© 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 大咖助场 | LMOS 说 C 语言（上）：为什么说 C 语言是一把瑞士军刀？

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



涨价倒计时 🔔

今日订阅 **¥89**，1月12日涨价至**¥199**

精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。