目　录

第一部分 抽象接口技术

第 1 章 面向对象编程 ……1
1.1 基础概念 ……1
1.1.1 对象 ……1
1.1.2 类 ……1
1.1.3 UML 类图 ……4
1.2 封装 ……5
1.2.1 “封装”示例 ……5
1.2.2 进一步分析——创建对象 ……8
1.2.3 进一步分析——销毁对象 ……15
1.3 继承 ……20
1.3.1 “继承”示例 ……20
1.3.2 进一步分析——初始化函数 ……23
1.3.3 进一步分析——解初始化函数 ……24
1.3.4 最少知识原则 ……24
1.4 多态 ……26
1.4.1 学生的“自我介绍” ……26
1.4.2 I/O 设备驱动 ……31
1.4.3 带检查功能的栈 ……33
第 2 章 抽象接口 ……52
2.1 基本概念 ……52
2.2 “接口”示例 ……52
2.2.1 在 C 语言中定义“接口” ……52
2.2.2 基于接口编写应用程序 ……53
2.2.3 实现类 ……54
2.2.4 主程序设计 ……56
2.3 更换底层硬件 ……57
第 3 章 依赖反转 ……59
3.1 问题引入 ……59
3.2 基本结构 ……63
3.3 依赖反转在 AWorks 中的应用 ……64
3.3.1 抽象接口层 ……64
3.3.2 应用程序的实现 ……70
3.3.3 抽象方法的实现 ……71

第二部分 组件开发规范

第 4 章 组件简介 ……74
4.1 组件的定义 ……74
4.2 组件化的优势 ……74
4.3 组件的设计原则 ……75
4.3.1 复用/发布等同原则（REP） ……75
4.3.2 共同闭包原则（CCP） ……75
4.3.3 共同复用原则（CRP） ……76
4.3.4 设计原则小结 ……76
4.4 特殊的组件 ……77
4.4.1 框架（Framework） ……77
4.4.2 平台（Platform） ……78
4.4.3 板级（Board） ……79
第 5 章 基本属性 ……84
5.1 名称（name） ……84
5.2 版本号（Version number） ……84
5.2.1 版本号的基本形式 ……86
5.2.2 先行版本号 ……86
5.2.3 编译信息 ……87
5.2.4 版本的优先层级判定 ……87
5.3 依赖（Dependency） ……89
5.3.1 组件依赖 ……89
5.3.2 框架依赖 ……93
5.3.3 平台依赖 ……93
5.3.4 板级依赖 ……94
5.4 组件仓库（Repository） ……94
5.5 简要描述（Description） ……94
5.6 License ……94
5.6.1 BSD ……95
5.6.2 MIT ……95
5.6.3 Apache Licence 2.0 ……95
5.6.4 GPL ……95
5.6.5 LGPL ……96
5.6.6 MPL ……96
5.7 贡献者（Contributors） ……96
5.8 开发语言（Programming Language） ……97
第 6 章 AXIO 工具 ……98
6.1 AXIO 简介 ……98
6.2 命令行工具 axio-cli ……98
6.2.1 简介 ……98
6.2.2 安装 ……99
6.3 Web 网站 ……100
第 7 章 组件的开发过程 ……101
7.1 创建组件目录 ……101
7.2 编写代码 ……101

7.3 添加组件描述文件 ……102
7.3.1 语法简介 ……103
7.3.2 基本属性表示 ……104
7.4 打包组件 ……105
7.5 发布组件 ……105

第三部分 MVVM 框架

第1章 面向对象编程

本章导读

提到面向对象编程，往往会涉及到三个重要的特性：封装、继承与多态。可能很多人对这几个词语并不陌生，甚至是耳熟能详。就像笔者随机的问了几个公司的实习生，“面向对象编程的三大特性是什么？”几乎都可以不假思索的回答：“封装、继承与多态”。但是，在实际编程中，应用这些特性真的能像回答问题这么熟练吗？

部分 C 程序员（特别是嵌入式 C 程序员）有一种误解：C 语言不是面向对象编程语言，C++、Java、Python 等更高级的语言才是，使用 C 语言是无法实现面向对象编程的。这种误解致使他们没有意识去接触一些优秀的面向对象编程方法，例如设计模式、设计原则、软件架构设计等等，进而很难开发出易维护、易部署、易重用、易管理的软件，很难面对项目需求的变更（扩展），很难开发和维护大型的复杂项目。

本章将站在实际应用的角度，以 C 语言为例，对这些特性进行详细的介绍。实际上，AWorks 的核心及基础组件均是使用 C 语言编写的（当然，这并不影响上层应用使用 C++等其它语言。其实，众所周知的 Linux，其内核主要也是使用 C 语言实现的），但其中应用了大量的面向对象编程思想。通过这一章的内容，读者也可以更进一步的理解 AWorks。

1.1 基础概念

1.1.1 对象

面向对象编程，“对象”是整个编程过程的关键。对象可以是人们要研究的任何事物，万事万物均可看作对象，从简单的整数到复杂的飞机、卫星等均可以看作对象。它不仅能表示具体的（看得见、摸得着）事物，还能表示抽象的规则（如信号量、客户端/服务器模型）、计划或事件。

关于“对象”，一种常见的解释是：“数据与函数的组合”。也就是说，每个对象都是由一组数据（用以描述对象的状态）和一组函数（对象支持的操作，用以描述对象的行为）组成的。由此可见，对象实现了数据和操作的结合，使数据和操作可以封装于“对象”这个统一体中。

在面向过程编程中，程序设计注重的是“过程”，先做什么，后做什么，在外界看来，整个程序由一系列散乱的数据和函数组合而成。而在面向对象编程中，程序设计注重的是“对象”，在外界看来，整个程序由一系列“对象”组合而成，数据和函数封装到了对象内部。

1.1.2 类

对象是有“类型”的。即：类。简言之，“类”是对一组对象共性的抽象，表示了**一类**对象，即一组具有相同特性（数据元素）和行为（功能）的对象。而对象，只是某个类中的一个具体化的个例，通常称之为类的实例。

实际上，C 语言中的基本类型也可以看作“类”，比如 int 类型，其表示了一组整数对象，它们具有一定的共性：数据上，各对象都有一个整数值，若 int 类型的数据为 32 位，则值的范围为：-2147483648 ~ 2147483647；操作上，各对象都支持加（+）、减（-）、乘（*）、除（/）、取模（%）、移位（>>、<<）、自加（++）、自减（--）等操作（显然，若是浮点类型，则仅支持加、减、乘、除，不支持取模、移位等运算）。这里的各个操作均使用了简化的符号表示（+、-、*、/等），这是因为整数相关的操作太过频繁了，编程语言进行了

表达方式的简化，本质上也可以将这些操作视为函数，例如，就整数加法而言，可以将“+”等效为一个特殊的函数，该函数的原型可能为：

```
int    add_int (int a, int b);
```

使用整数类型（int）可以快速创建一个对象（实例），例如：

```
int a = 5;
```

这里即创建了一个整数对象 a，其当前值为 5。

这里使用整数类型仅仅是对“类”作一个简单的说明，以便体会类的含义。在实际应用中，程序员往往需要根据实际情况自己去发现类、创建类。前面提到，类是对一组对象共性的抽象，显然，类就是抽象的结果。所谓抽象，就是“去异求同”，把各个对象中共同的、本质性的特征提取出来。比如香蕉、苹果、哈密瓜等，它们共同的特性就是水果。得出水果概念的过程，就是一个抽象的过程。在抽象时，同与不同，取决于从什么角度上来抽象。抽象的角度取决于分析问题的目的。例如，同样是学生：对于班主任来讲，因为其很可能要与每个学生的家长沟通，因此在班主任看来每个学生的共性可能是：姓名、学号、家长姓名、家长联系方式；而对于数学科任教师来讲，可能更加关心的是学生的成绩变化，则每个学生的共性可能是：姓名、学号、第一月末测试成绩、第二月末测试成绩、期中测试成绩……

对象通常是由数据（用以描述对象的状态）和函数（对象支持的操作，用以描述对象的行为）组成的，而类是对象共性的抽象，相应的，类也具有两部分内容：属性（数据的抽象）和方法（对象行为的抽象）。

除了封装属性和操作外，类还具有访问控制的能力，比如，某些属性和方法可以是私有的，不能被外界访问。通过访问控制，能够对内部数据提供不同级别的保护，以防止外界意外地改变或使用了私有部分。

1. 属性

类具有属性，它是对数据（对象的状态）的抽象。在 C 程序设计时，通常使用结构体类型来表示一个类，相关属性即包含在相应的结构体类型中。例如，学生具有如下属性：姓名、学号、性别、身高、体重等信息。则可以使用如下结构体类型表示“学生类”：

```
1    struct student {
2        char          name[10];                /* 姓名 （假定最长 10 字符） */
3        unsigned int  id;                      /* 学号                       */
4        char          sex;                     /* 性别:'M'，男；'F' ，女     */
5        float         height;                  /* 身高                       */
6        float         weight;                  /* 体重                       */
7    };
```

2. 方法

类具有方法，它是对象行为的抽象，用方法名和实现该操作的方法来描述。在 C 程序设计中，方法可以看作普通的函数，不过其通常有一个特点：函数的第一个参数，为“类”类型的指针，该指针指向了一个确定的对象，用以表明此次操作要作用于哪个对象，在方法实现时，即可通过该指针访问到对象中的各个属性。

例如，针对学生对象，为了对外展现学生自身的信息，假定每个学生都能够进行“自我介绍”，且自我介绍的格式是对外输出一个固定格式的字符串：

"Hi! My name is **xxx**, I'm a (boy/girl). My school number is **xxx**. My height is **xxx**cm and weight is **xxx**kg . "

其中的 xxx 在实际自我介绍时，应更换为该学生自身实际的信息。

基于此，可以为学生类定义并实现一个“自我介绍”的方法，方法的声明为：

```
void student_self_introduction (struct student *p_this);
```

其实现详见程序清单 1.1。

程序清单 1.1　自我介绍方法的实现

```
1    void student_self_introduction (struct student *p_this)
2    {
3    printf("Hi! My name is %s, I'm a %s. My school number is %d. My height is %fcm and weight is %fkg.\n",
4            p_this->name,
5            (p_this->sex == 'M') ? "boy" : "girl",
6            p_this->id,
7            p_this->height,
8            p_this->weight);
9    }
```

对于外界来讲，调用学生“自我介绍”的方法，就可以获知学生的全部信息。基于该类的定义，一个简易的应用程序（主程序）范例详见程序清单 1.2。

程序清单 1.2　学生类主程序范例

```
1    void main(void)
2    {
3            struct student zhangsan = {"zhangsan", 2010447222, 'M', 173, 60};
4            struct student lisi = {"lisi", 2010447223, 'M', 168, 65};;
5
6            student_self_introduction(&zhangsan);
7            student_self_introduction(&lisi);
8
9            // ...
10   }
```

在创建对象时，需要为各个成员赋初始值（比如姓名、学号等）。程序中采用结构体赋值的方法进行了赋值，这样赋值的前提是用户知道结构体类型的详细定义，也就是说，struct student 类型的定义，必须完全暴露给用户。这显然有些不妥，下面在介绍封装时，将对这个问题作进一步分析。

由程序清单 1.2 可知，类中的方法（student_self_introduction()）可以作用于任一学生类对象（即任意学生）。由此可见，“类”这个概念的提出，使人们在研究事物时，可以从研究单个对象上升到研究一组对象。极大的扩大了研究的“视野”，对于程序员来讲，编写的代码（例如，函数）将适用于一组对象，而非特定的某一个对象，提高了代码利用率。这有利于程序员组织更大的程序，面对更复杂的问题。

在实际应用中，不少程序员都喜欢编写出一堆非常类似的接口，它们仅通过某一个数字后缀（0、1、2……）来区分，例如，系统中使用到了 3 个栈，则程序员可能实现 3 个入栈函数，示意代码详见程序清单 1.3。

程序清单 1.3　系统中提供的 3 个入栈函数

```
1    int    push_stack0 (int data)                                    // 将数据压入栈 1 中
```

```
2    {
3        // 入栈代码
4    }
5    int   push_stack1 (int data)                          // 将数据压入栈 2 中
6    {
7        // 入栈代码
8    }
9    int   push_stack2 (int data)                          // 将数据压入栈 3 中
10   {
11       // 入栈代码
12   }
```

它们可能除了极小部分的处理不同之外，其它处理几乎完全相同（毕竟入栈逻辑是基本一致的）。这就是没有面向对象编程的思维，没有定义对象类型的概念，将操作直接针对每个具体对象（栈 1、栈 2、栈 3），而不是一组同类的对象（所有栈对象）。显然，由于 3 个栈的特性和行为都基本类似，因而可以定义一个“栈类型”，如此一来，入栈操作将属于栈类型中的一个方法，适用于所有栈对象。例如：

```
int   push_stack (stack *p_stack, int data);                // 将数据压入栈，  p_stack  指向具体的栈对象
```

此时，三个栈的入栈操作均可使用同一个方法：

```
push_stack(p_stack1, 1);
push_stack(p_stack2, 2);
push_stack(p_stack3, 3);
```

这里只是给出了一些示意性代码，使读者体会在面向对象编程中，使用“类”的设计来解决问题所带来的不同。

类是封装、继承和多态的基础，这三大特性都都需要通过“类”来体现。后文在介绍这 3 大特性时，还会进一步详细介绍类的定义与设计。

1.1.3 UML 类图

在面向对象的设计和开发过程中，通常使用 UML 工具来进行分析与设计。最基本的，就是使用 UML 类图来表示类以及描述类之间的关系。

在 UML 类图中，用一个**矩形框**表示一个类，矩形框内部被分隔为上、中、下三部分：上面部分为类的名字；中间部分为类的属性；下面部分为类的方法。对于属性和方法，还可以使用“+”、“-”修饰符来表示访问权限，“+”为公有属性、“-”为私有属性。

例如，前面简单的介绍了学生类，其类名为 student，属性包括姓名、学号、性别、身高、体重，方法有“自我介绍”方法。则其对应的类图详见图 1.1。

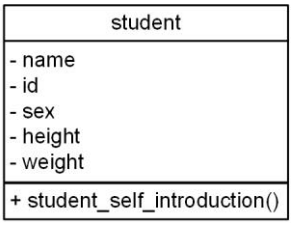


图 1.1　UML 类图

通常情况下，类中的所有属性均为私有属性，不建议用户直接访问，取而代之的是，对所有属性的访问都通过类提供的方法。基于此，假定了学生类中的所有属性均为私有属性，因而在所有属性前都增加了“-”修饰符。当然，在实际应用中，可以根据需要决定哪些是公有属性，哪些是私有属性。

UML 类图主要用于辅助分析和设计，在设计类时应聚焦在与当前问题有关的重要属性和行为，无关的属性和方法统统去掉，确保 UML 类图是简洁有效的。由于私有属性仅在内部使用，外界无需关心，因此，为了简化 UML，在 UML 类图中通常都不体现出私有属性和方法，除非某些特殊的私有属性和方法影响到问题的理解或者类的实现。基于此，可以简化图 1.1，省略私有属性，详见图 1.2。

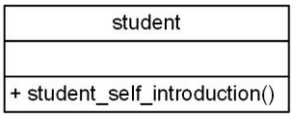


图 1.2　简化的 UML 类图

UML 类图以及其它 UML 元素都是辅助软件开发的工具，使用 UML 进行设计时，只要相关人员能够通过 UML 图看懂你的设计、不妨碍沟通就可以了，即使用草稿纸来作图也是可以的，所以，不用太过纠结那些细节，且一定要避免过度设计。

在后文的介绍中，会使用到一些简单的类图，以对当前介绍的问题进行辅助说明，这些类图都十分简洁易懂，即使读者没有 UML 类图方面的知识储备，也完全可以看懂。

1.2　封装

前面提到，类是对一组对象共性的抽象，封装了属性和方法。也就是说，“类”这个概念本身，就具有**封装**的意义，其封装了属性和方法。

所谓封装，从字面意义上理解，即把一组关联的数据和函数圈起来，使圈外面的代码只能看见部分函数，数据则完全不可见（一般来讲，不建议圈外看见任何数据，数据的访问都应通过类提供的方法，不应走任何捷径）。

1.2.1 “封装”示例

在 C 语言中，可以使用一个 C 文件（*.c 文件）和 H 文件完（*.h 文件）成“类”的定义，将所有需要封装的东西都存于 C 文件中，H 文件中只展现那些“对外可见、无需封装”的内容。

以栈的实现为例，将所有实现代码都存于 C 文件中，H 文件只包含与栈相关接口的声明，比如入栈和出栈等。头文件和源文件的示意内容分别详见程序清单 1.4 和程序清单 1.5。

程序清单 1.4　栈实现头文件内容范例（stack.h）

```
1   #ifndef __STACK_H
2   #define __STACK_H
3
4   struct stack;                                   /* 类型声明，无需关心类定义的具体细节 */
5
6   struct stack * stack_create (int size);         /* 创建栈，并指定栈空间的大小          */
7
8   int stack_push (struct stack *p_stack, int val);        /* 入栈                    */
9   int stack_pop (struct stack *p_stack, int *p_val);      /* 出栈                    */
10  int stack_delete (struct stack *p_stack);               /* 删除栈                  */
11
12  #endif
```

程序清单 1.5　栈实现源文件内容范例（stack.c）

```
1   #include "stack.h"
2   #include "stdlib.h"                             /* for malloc                      */
```

```
3
4   struct stack {
5       int                 top;                                    /* 栈顶                                 */
6       int                *p_buf;                                  /* 栈缓存                               */
7       unsigned int size;                                          /* 栈缓存的大小                         */
8   };
9
10  struct stack * stack_create (int size)
11  {
12      struct stack *p_stack = (struct stack *)malloc(sizeof(struct stack));
13      if (p_stack != NULL) {
14          p_stack->top      = 0;
15          p_stack->size     = size;
16          p_stack->p_buf = (int *)malloc(sizeof(int) * size);
17          if (p_stack->p_buf != NULL) {
18              return p_stack;
19          }
20          free(p_stack);                                          /* 分配栈内存失败                       */
21      }
22      return NULL;                                                /* 创建栈失败，返回 NULL                */
23  }
24
25  int stack_push (struct stack *p_stack, int val)
26  {
27      if (p_stack->top != p_stack->size) {
28          p_stack->p_buf[p_stack->top++] = val;
29          return 0;
30      }
31      return -1;
32  }
33
34  int stack_pop (struct stack *p_stack, int *p_val)
35  {
36      if (p_stack->top != 0) {
37          *p_val = p_stack->p_buf[--p_stack->top];
38          return 0;
39      }
40      return -1;
41  }
42
43  int stack_delete (struct stack *p_stack)
44  {
45      if (p_stack == NULL) {
46          return -1;
```

```
47          }
48          if (p_stack->p_buf != NULL) {
49                  free(p_stack->p_buf);
50          }
51          free(p_stack);
52          return 0;
53  }
```

显然，使用 stack.h 的程序是没有 struct stack 结构体成员的访问权限的，它们只能调用 stack.h 文件中声明的方法：创建（stack_create）、删除（stack_delete）、入栈（stack_push）、出栈（stack_pop）。对于外界用户来说，struct stack 这个结构体的内部细节，以及各个函数的具体实现方式都是不可见的。这正是完美的封装！

基于程序清单 1.4 中列举的各个方法，可以使用栈存储数据，示意代码详见程序清单 1.6。

程序清单 1.6　栈的使用范例（1）

```
1   #include "stack.h"
2   #include "stdio.h"
3
4   int main ()
5   {
6           int             val;
7           struct stack *p_stack = stack_create(20);
8
9           // 依次压入数据：2、4、5、8、10
10          stack_push(p_stack, 2);
11          stack_push(p_stack, 4);
12          stack_push(p_stack, 5);
13          stack_push(p_stack, 8);
14          stack_push(p_stack, 10);
15
16          // 依次弹出各个数据，并打印
17          stack_pop(p_stack, &val); printf("%d ", val);
18          stack_pop(p_stack, &val); printf("%d ", val);
19          stack_pop(p_stack, &val); printf("%d ", val);
20          stack_pop(p_stack, &val); printf("%d ", val);
21          stack_pop(p_stack, &val); printf("%d ", val);
22
23          stack_delete(p_stack);
24          return 0;
25  }
```

栈遵循“先进后出”的原则，若整个过程没有错误，则实际打印的出栈数据为：10、8、5、4、2。

由于所有细节都封装到了 C 文件内部，用户通过 stack.h 文件并不能看到 struct stack 结构体的具体定义，因此，用户也无法访问 stack 结构体中的成员。若用户尝试访问 struct stack 结构体中的成员，将会出错，错误代码示意详见程序清单 1.7。

程序清单 1.7　栈的错误使用范例

```
1    #include "stack.h"
2    #include "stdio.h"
3
4    int main ()
5    {
6        int          val;
7        struct stack *p_stack = stack_create(20);
8
9        // 依次压入数据：2、4、5、8、10
10       stack_push(p_stack, 2);
11       stack_push(p_stack, 4);
12       stack_push(p_stack, 5);
13       stack_push(p_stack, 8);
14       stack_push(p_stack, 10);
15
16       // 尝试查看栈顶的值
17       printf("Now stack top is %d ", p_stack->top);
18
19       stack_delete(p_stack);
20       return 0;
21   }
```

上述程序在编译阶段就会报错。错误发生在程序清单 1.7 的第 17 行，其访问了栈的内部数据：top，这是非法的。从编译器的角度看，在编译这段程序时，其还并不知道 struct stack 的具体定义，更不用说访问其中的成员了，因此，在编译这行代码时，通常会报出类型不完整（类型只有声明，没有定义，不能尝试访问其中的成员）的错误，例如：

```
error: dereferencing pointer to incomplete type 'struct stack'
```

上述“封装”示例是完美的。在语义上实现了封装的同时，编译器还会保证“封装”的正确运用，保证“封装”不会被用户意外的破坏。

由此可见，虽然 C 语言不是面向对象的编程语言，但是其也可以完美的实现封装。C 语言实现封装的一般做法为：在头文件中进行数据结构以及函数定义的前置声明（forward declare），然后在程序源文件中完成各个函数的具体实现以及数据结构的定义。此时，所有函数实现及定义细节均封装到了源文件中，对使用者来说是完全不可见的。

实际上，即使是面向对象编程语言（比如 C++），通常也达不到如此完美的封装程度。以 C++为例，使用 class 关键字定义的类，必须存放在 H 文件中，如此一来，类中的各个成员对外就是可见的。虽然可以通过某些语法限制外界对这些成员的访问（如 private 关键字），但无论如何，外界还是“看见了”类中的各个成员，知道了它们的存在，类的内部细节在一定程度上还是被暴露了，这样的封装显然是不完美的。

1.2.2 进一步分析——创建对象

1． 动态内存分配存在的问题

在程序清单 1.4 中，声明了创建对象的方法，基于此方法，可以创建多个栈对象，例如：

```
struct stack *p_stack1 = stack_create(20);
```

```
struct stack *p_stack2 = stack_create(30);
struct stack *p_stack3 = stack_create(50);
```

每个栈对象需要两部分内存：一是栈对象本身的内存（内存大小为 sizeof(struct stack)）；二是该栈对象用于存储数据的缓存（内存大小为 sizeof(int) * size，其中，size 由用户在创建栈时通过参数指定）。在栈对象的创建函数中，使用 malloc()分配了该对象所需的内存空间：

- 栈对象空间

```
struct stack *p_stack = (struct stack *)malloc(sizeof(struct stack));
```

- 缓存空间（存储数据的空间）

```
p_stack->p_buf = (int *)malloc(sizeof(int) * size);
```

虽然使用 malloc()分配内存空间非常方便，但这种做法也限制了对象内存的来源——必须使用动态内存。对于 PC 软件开发而言，“对象内存的来源”或许完全不需要考虑，因为 PC 机的内存通常都不是瓶颈，且 PC 系统通常都不是实时系统，对实时性要求不高。但在嵌入式系统中，内存往往是很大的瓶颈，致使很多应用场合可能并不太适合使用动态内存，主要有以下几个因素：

1) 内存小

运行嵌入式软件的硬件平台普遍内存小，小到甚至只有几 K RAM。在这种条件下使用动态内存是比较浪费的行为。主要有两方面原因：一是动态内存分配容易产生内存碎片；二是动态内存分配的软件算法本身，也会占用一定的内存空间。

2) 实时性要求高

部分嵌入式应用对实时性要求很高。但由于资源的限制，嵌入式系统中所使用的动态内存分配算法绝大部分都不是很完善（因其必须在算法空间复杂度和时间复杂度之间平衡），使得很难确保动态内存分配的实时性。

3) 内存泄漏

动态内存分配很容易出现内存泄漏。只要有一处内存的分配没有被有效释放，就会出现内存泄漏。

4) 软件编程复杂

虽然动态内存分配获取内存的方式非常简单，但在一个可靠的设计中，必须考虑内存分配失败的情况，并对其进行处理。如果系统存在大量的动态内存分配，则处处都需要考虑分配失败的情况。

由此可见，动态内存分配存在种种缺陷，可能在部分应用场合并不适宜。将对象内存的来源限制为动态内存分配的同时，也限制了该类的应用场合。致使部分应用场合因为内存来源的问题不得不放弃该类的使用。为了避免出现这种情况，不应该对内存来源加以限制。

2. 内存来源的探索

在 C 程序开发中，除了使用 malloc()得到一段内存空间外，还可以使用“直接定义变量”的形式分配一段内存。例如：要获取一个 int 类型数据的存储空间，可以有两种方式。使用 malloc()获取动态内存的方式如下：

```
int *p = (int *)malloc(sizeof(int));          // 指针 p 即指向了内存空间首地址
*p= 1;                                        // 设置这段内存空间中存储的整数值为 1
```

直接定义变量的形式如下：

```
int   a;                                      // 定义变量 a
```

```
a = 1;                                  // 赋值为 1
```

直接定义变量的形式，内存在编译阶段由编译器负责分配，无需用户作任何干预。根据变量定义位置的不同，实际内存的开辟位置也会存在一定的区别，主要有两类：

- 局部变量：内存开辟在栈中；
- 静态变量（static 修饰的变量）或全局变量：内存开辟在全局静态存储区。

这两种变量主要是声明周期的不同：局部变量在退出当前作用域后（比如函数返回），内存自动释放；静态变量或全局变量内存开辟在全局静态存储区，它们在程序的整个生命周期均有效。

由此可见，在 C 编程中，内存可以有 3 种来源，它们的优缺点对比详见表 1.1。

表 1.1　动态内存、静态内存和栈内存

内存类别	内存位置	生存周期	优点	缺点
动态内存	系统堆	直到调用 free()释放内存	灵活，可以随时按需分配和释放	内存分配可能失败，花费的时间可能不确定；需要处理内存分配失败的情况，增加程序的复杂性
静态内存	全局静态存储区（.data、.bss 存储段）	程序的整个运行周期	确定性好，只要程序能够编译、链接成功，内存一定能够分配成功	需要编程时确定内存的大小； 一直占用内存，无法释放
栈内存	系统栈（或任务栈）	函数调用周期	自动完成内存的分配和回收	内存太大会导致栈溢出

注：注意区分这里的栈内存和前面介绍的栈类，这里的栈内存仅仅是指在栈中的一段内存，而前面介绍的栈类，是实现了“先入后出”逻辑的一个“类”，核心提供了 PUSH 和 POP 方法。

由于静态内存和栈内存的定义形式基本相同（在编程时，使用定义变量的形式定义），同时，为了与动态内存相对应，往往将静态内存和栈内存都称之为静态内存。

各种不同来源的内存各有优劣。前面提到，stack_create()函数将内存的来源限制为动态内存不太合理。回顾 stack_create()函数的定义可以看到，该函数主要完成了两个工作：

- 相关内存的分配；
- 栈对象中各个成员的赋值（top、p_buf、size）。

既然内存来源有三种，那么为了避免内存来源受限，“相关内存的分配”这一步建议交由用户完成，以便用户根据实际需要自由选择内存的来源。基于此，可以将对象的创建拆分为两个独立的步骤：分配对象所需的内存；初始化对象。接下来将进一步探索这两个步骤如何实现。

3. 分配对象所需的内存

为避免内存来源受限，内存分配的工作建议交由用户完成，以便用户根据实际需要自由选择内存的来源。用户能够完成内存分配的前提是：用户知道应该分配的内存大小。前面提到，每个栈对象需要两部分内存：一是栈对象本身的内存（内存大小为：sizeof(struct stack)）；二是该栈对象用于存储数据的缓存（内存大小为 sizeof(int) * size，其中，size 由用户在创建栈时通过参数指定）。

- 栈对象本身的内存

栈对象本身的内存大小为 sizeof(struct stack)，若用户直接采用静态内存分配的方式（直接定义一个变量），则形式如下：

```
struct stack my_stack;
```

当然，用户也可以继续采用动态内存的分配方式，例如：

```
struct stack *p_stack = (struct stack *)malloc(sizeof(struct stack));
```

但是，若读者尝试将这两行代码直接放到主程序中，会发现根本编译不过！这是因为之前描述的“封装”特性，使外界看不到 struct stack 的具体定义，也就是说，对于外界而言，该类型仅仅只是声明了，并未定义，该类型对应变量的大小对外也是未知的。

在 C 语言中定义一个变量时，编译器将负责该变量所占用内存的分配。内存的位置与变量的作用域相关：局部变量（系统栈中）、全局变量和静态变量（全局静态存储区中）。内存的大小与类型相关。显然，要完成变量内存的分配，编译器必须知道变量所占用的存储空间大小。当一个变量的类型未定义时，无法完成该类型对应变量的定义，因此，如下语句在编译时会出错：

```
struct stack my_stack;
```

同理，sizeof 语句用于获得相应类型数据的大小，而未定义的类型显然是不知道其大小的，因此，动态内存分配中所使用的 sizeof(struct stack)语句也是错误的。

读者可能会有疑问，既然该类型未定义，为什么在主程序中定义该类型的指针变量却可以呢？比如程序清单 1.6 的第 7 行：

```
struct stack *p_stack = //...
```

虽然 struct stack 类型未定义，但在之前已经声明，因此，编译器知道其是一个“合法的结构体类型”。此外，这里定义的是一个指针变量，在特定系统中，指针变量所占用的内存大小是确定的，例如，在 32 位系统中，指针通常占用 4 个字节。换句话说，指针变量所占用的内存空间大小与其指向的数据类型无关，编译器无需知道其指向的数据类型，就可以完成指针变量内存的分配。因此，即使一个类型未定义，只要其声明了，就可以定义该类型的指针变量。只不过，需要注意的是，在完成该类型的定义之前，不得尝试访问该指针所指向的内容（前面封装的实现就充分利用了这个特性）。

由此可见，由于封装特性，将类型的定义存放在 C 文件时，外部无法基于类型的声明完成内存的分配。这里提出了 3 种解决方案。

（1） 将类的具体定义放到 H 文件中

为了使用户知道对象内存的大小，一种改动最小的办法是直接将类型的定义放在 H 文件中。更新后的 H 文件示意代码详见程序清单 1.8。

程序清单 1.8　栈实现头文件更新 1（stack.h）

```
1   #ifndef __STACK_H
2   #define __STACK_H
3
4   struct stack {                                  /* 类型定义                */
5       int             top;                        /* 栈顶                    */
6       int            *p_buf;                      /* 栈缓存                  */
7       unsigned int size;                          /* 栈缓存的大小            */
8   };
9
```

```
10     // ..... 其它函数声明
11
12   #endif
```

此时，对于外界来讲，类型已经定义，如下语句均可正常使用：

```
struct stack my_stack;                                                    // 静态内存分配
struct stack *p_stack = (struct stack *)malloc(sizeof(struct stack));     // 动态内存分配
```

但是，由于类型的定义存放到了 H 文件中，暴露了类中的成员，在一定程度上破坏了类的“封装”性。此时，外界可以直接访问类中的数据成员，如下语句将可以顺利编译通过：

```
printf("Now stack top is %d ", p_stack->top);
```

换句话说，类似程序清单 1.7 中第 17 行所示的“非法”访问，只能靠人为约定，这将极大的依赖于程序员的“素质”。“封装”无法再通过编译器保证，也使先前的“封装”变得不那么完美。

凡事都有取舍，牺牲一定的封装性，换来内存分配的灵活性，这也是在嵌入式系统中，基于 C 语言实现面向对象编程的一般做法（可能是因为将类的定义存放在 H 文件中更加符合程序员的编程风格）。读者在阅读嵌入式系统中的代码时，可能会发现很多类定义在 H 文件中，并没有封装在 C 文件中。很大程度上，就是这个原因导致的。

通过上面的描述可知，虽然类的定义存放在 H 文件中，但出于封装性考虑，外界任何时候都不应直接访问对象中的数据，作为一名合格的程序员，应该将其视为使用 C 语言实现面向对象编程的一条**准则**。这主要指导了两个方面：一是在设计类时，应考虑到用户可能访问的数据，并为这些数据提供相应的访问接口；二是在使用别人提供的类时，除非有特殊说明，否则都不应该尝试直接访问类中的数据。

由于这种方法是目前嵌入式系统中使用得最为广泛的一种方法，因此，后文重点使用这种方法讨论类。作为解决方案的补充，接下来还会继续介绍两种方案，与这种方案相比，它们具有一定的优点，但同时也存在一系列缺点。

（2） 在 H 文件中定义一个新的结构体类型

为了继续保持类的封装性，类的定义依然保留在 C 文件中。只不过与此同时，在 H 文件中定义一个新的结构体类型。在该结构体类型中，各个成员的顺序和类型与类定义完全一致，仅命名不同。例如，基于栈类型的定义，可以在 H 文件中定义一个与之基本相同的类型，定义如下：

```
struct stack_mem {
    int             dummy1;
    int            *dummy2;
    unsigned int    dummy3;
};
```

其中，各成员的顺序和类型均与 struct stack 的定义完全相同，以此保证两个类型数据所需要的内存空间完全一致。同时，为了屏蔽各个成员的具体含义，所有成员均以 dummy 开头进行命名。

此时，对于外界来讲，可以基于 struct stack_mem 类型完成内存的分配，例如：

```
struct stack_mem    my_stack;                                             // 静态内存分配
struct stack *p_stack = (struct stack *)malloc(sizeof(struct stack_mem));  // 动态内存分配
```

由此可见，使用这种解决方案时，类的实际定义依然没有暴露给外界，继续保持了良好

的封装。但是，这里定义了一个新的类型，给用户理解上造成了一定的困扰，此外，为了确保两个类型完全一致，就要求类的设计者在修改类的定义时，必须确保 struct stack_mem 类型也同步修改，这给类的维护工作带来了很大的挑战。不难想象，如果系统中有很多类，每个类的修改都必须同步修改对应的用于内存分配的结构体类型，这给维护工作带来的极大的挑战，稍有不慎，某一个类型没有同步修改，就可能造成严重的错误，且这种错误编译器不会给出任何提示，非常隐蔽，很难发现。

实际上，在 FreeRTOS 中，很多地方都采用了这种方法。以任务创建时需要使用到的任务控制块为例：为了用户可以使用静态的方式分配任务控制块内存，在 FreeRTOS.h 文件中，定义了 StaticTask_t（struct xSTATIC_TCB）类型，其中各个成员的类型和顺序均与 tasks.c 文件中定义的 tskTCB（struct tskTaskControlBlock）类型保持一致，确保用户可以通过 StaticTask_t 类型分配一个与 tskTCB 类型完全相同的内存空间。

（3） 使用宏的形式告知对象所需的内存大小

既然外界只需要知道对象内存的大小，那么就可以在开发过程中，使用 sizoeof()获得 struct stack 类型的大小，然后将其以宏的形式定义在 H 文件中。例如，在某一 32 位系统中，使用 sizeof()获知 struct stack 类型的长度为 12，则可以在 H 文件中定义一个宏，例如：

```
#define    STACK_MEM_SIZE    12
```

用户使用该宏可以轻松的完成内存分配，例如：

```
unsigned char    stack_mem[STACK_MEM_SIZE];
```

这种做法仅仅在头文件中新增了一个宏定义，类的定义依然保持的 C 文件中，“封装”完全没有被破坏，看起来也非常完美。但这种做法依然存在一些问题，因而一般很少采用，若读者需要在实际项目中尝试使用这种方法，应该十分谨慎的对待这些问题：

a) 对于同一个类型，不同系统中 sizeof()的结果可能不同

类型的长度与系统和编译器均相关。以 int 类型为例，在 32 位系统中，其往往为 32 位（4 字节），但在其它系统中，其字长可能不同，例如，在某些 16 位系统中，其位宽可能为 16 位（2 字节）。因此，同样是 sizeof(int)，结果可能为 4，也可能为 2。类似地，当使用 sizeof()获取类型的长度时，不同系统中获取的结果可能并不相同。这就导致 H 文件中的宏定义，每切换一个平台，都要重新测试验证一次，将其修改为正确的值。同时，由于实际类型的定义封装到了 C 文件中，因此，该修改过程只能有类的开发者完成，一般用户还无法完成。这就使得该类的跨平台特性很差。

b) 实际中，内存不仅有大小的要求，还有内存对齐的要求

前面只提到了内存大小的问题，没有提到内存对齐的问题。这是因为，在使用结构体类型定义变量时，编译器会保证该变量的内存按照结构体的要求进行对齐，而使用 unsigned char 类型定义数组时，例如：

```
unsigned char stack_mem[STACK_MEM_SIZE];
```

其对齐要求是按照 unsigned char 型数组的对齐要求进行的，两者的对齐要求不一定相同。对齐要求与具体的平台和编译器相关，例如，unsigned char 类型通常仅需按照单字节对齐，内存地址可以是 0、1、2、3……等任意地址，而 struct stack 结构体的各个成员均为 4 个字节（假定），该结构体类型的变量通常需要按照 4 字节对齐，内存首地址只能是以 0、4、8……结尾的地址。显然，若使用 unsigned char 类型定义的数组首地址不是 4 的整数倍，系统将可能出错。有关结构体对齐相关的内容，可以参阅《程序设计与数据结构》（周立功、周攀峰著）一书中对结构体的讲解（2.2 节）。

当使用动态内存分配时，系统通常会保证分配的内存按照系统的对齐系数（在 32 位系

统中，对齐系数即为 4 字节）对齐，实际上，当满足该条件时，均已满足结构体变量的对齐要求，因此，当使用动态内存分配时，往往不会遇到内存对齐的问题。

综合上述两个问题可见，通过一个宏告知用户需要分配的内存空间大小并不是十分合适，会遇到跨平台、内存对齐等多个注意事项，用户可能在不经意间出错。因此，在实际的嵌入式系统中，这种方法很少使用。

- 存储数据的缓存

存储数据的缓存大小为 sizeof(int) * size，其中的 size 本身就是由用户指定的，因此，这部分内存的大小用户很容易得知，进而完成内存的分配。

可以采用静态内存分配的方式（直接定义一个变量）完成内存的分配，例如：

```
int buf[20];
```

也可以采用动态内存分配的方式完成内存的分配，例如：

```
int *p_buf = (int *)malloc(sizeof(int) * 20);
```

4. 初始化对象

初始化对象相当于对象的一种操作（初始化），显然，该操作的具体细节用户不需要关心，为此，栈对象应该提供一个“初始化”方法，专用于对象的初始化。初始化时，需要指定栈对象的地址（以访问对象内存）、缓存地址（以访问缓存）及缓存大小，基于此，可以定义初始化函数的原型为：

```
int stack_init (struct stack *p_stack, int *p_buf, int size);
```

对于栈来讲，栈顶索引（top）的初始值恒为 0，因此，该值无需通过初始化函数的参数传递。int 类型的返回值常用于表示执行的结果（成功或失败）。该函数的实现示意代码详见程序清单 1.9。

程序清单 1.9　栈的初始化函数实现范例

```
1    int stack_init (struct stack *p_stack, int *p_buf, int size)
2    {
3        p_stack->top    = 0;
4        p_stack->size   = size;
5        p_stack->p_buf = p_buf;
6
7        return 0;
8    }
```

注：该初始化函数的实现仅作为原理性展示，没有做过多的错误处理或参数检查，显然，在实际应用中，p_stack 为 NULL 或 p_buf 为 NULL 等情况都是错误情况，应该作出处理。

至此，完成了将创建对象分离为“分配对象所需的内存”和“初始化对象”两个步骤。对象内存的来源完全交由的用户决定，用户根据需要获得内存后，再将相关内存的首地址传递给初始化函数。

前面介绍了内存的来源主要有三种：动态内存、静态内存和栈内存（详见表 1.1）。相应地，通常将来源于这三种内存的对象分别称之为：动态对象、静态对象和栈对象。根据三种内存的特性对比，可以将三种对象加以对比，详见表 1.2。

表 1.2　动态对象、静态对象和栈对象

对象类别	内存位置	生存周期	优点	缺点
动态对象	系统堆	直到调用 free()释放内存	灵活，可以随时按需创建和销毁对象	内存分配可能失败，花费的时间可能不确定；需要处理内存分配失败的情况，增加程序的复杂性
静态对象	全局静态存储区（.data、.bss 存储段）	程序的整个运行周期	确定性好，只要程序能够运行起来，对象一定能够创建成功	需要编程时确定对象的数量；一直占用内存，无法销毁；对象数量太多、太大时会影响系统启动时间
栈对象	系统栈（或任务栈）	函数调用周期	自动完成对象内存的分配和回收	对象太大会导致栈溢出

注：注意区分表中的栈对象和前面介绍的“栈对象”。表中的栈对象指的是内存开辟在栈中的所有对象；而前面介绍的“栈对象”，是“栈类”的一个具体实例，可以使用“栈类”提供的 PUSH 和 POP 方法。

根据这些特性，可以对这些对象的应用场合进行简单的说明和对比，详见表 1.3，以便用户选择合适的对象创建方式。

表 1.3　动态对象、静态对象和栈对象的应用场合

对象类别	应用场合
动态对象	不会频繁创建、销毁对象的应用[1]；内存占用太大的对象
静态对象	确定性要求较高，长生命周期的对象
栈对象	函数内部使用的临时对象；对象内存占用较小的对象

由于很多嵌入式应用对确定性要求较高，因此，在嵌入式系统设计中，往往都建议优先使用静态对象。如此一来，只要能够编译（包含链接）成功，应用程序往往就可以按照确定的流程正确执行，否则，若使用动态对象，则必须考虑对象创建失败的情况。

1.2.3 进一步分析——销毁对象

在程序清单 1.5 中，还实现了一个与 stack_create()对应的函数：stack_delete()。该函数用于销毁（删除）一个对象。设计该函数的初衷是，当一个栈对象不会再被使用时，可以通过该函数释放栈占用的资源，比如释放在 stack_create()函数中使用 malloc()分配的内存资源。

当将 stack_create()拆分为两步后，内存的分配将由用户决定，对应地，内存的释放也应由用户决定（只有用户知道内存的来源，进而也只有用户知道如何释放）。也就是说，在 stack_delete()函数中，无需再释放内存。回顾 stack_delete()函数的实现可以发现，该函数目前只做了内存释放相关的操作，当不需要释放内存时，该函数看起来已经没有存在的必要了。

实际上，stack_delete()和 stack_create()函数是对应的，当将 stack_create()拆分为“分配对象所需的内存”和“初始化对象”两个步骤后，stack_delete()也应该相应的拆分为两个步骤：“释放对象占用的内存”和“解初始化对象”（解初始化，部分读者可能第一次接触这个名称，不用太在意这个名称的命名，只要理解这里表达的是初始化对象的“逆过程”即可）。

1．释放对象占用的内存

前面已经提到，释放内存交由用户处理，释放方法与内存的来源相关。

- 动态内存的释放

若内存来自于动态内存分配，则应使用相应的释放内存函数（例如 free()）进行释放。在释放时，应确保分配的内存全部被有效释放。

例如，对于栈来讲，其占用了两部分内存：栈对象自身的内存和栈缓存。释放时应确保两部分内存均被释放，若某一部分内存被遗漏，忘记释放，将造成内存泄漏。随着程序的长期运行，内存不断泄漏，可能有系统崩溃的风险。

- 静态内存的释放

在使用 C 编程时，若使用静态内存（定义变量的形式），则内存的释放是系统自动完成的。若将对象定义为局部变量，内存开辟在系统栈中，则退出当前作用域后（函数返回）自动释放；若将对象定义为静态变量（static）或全局变量，则内存开辟在全局静态区，该区域的内存在应用程序的整个生命周期均有效，无法释放。

2. 解初始化对象

释放内存已交由用户处理，因此，对于类的设计来讲，重点是设计“解初始化对象”对应的函数，该函数与 stack_init()函数对应，通常命名为“*_deinit”，即：stack_deinit()。该函数通常用于释放在初始化对象时占用的其它资源。

对于纯软件对象（与硬件无关的软件），通常其只会占用内存资源，不会额外占用其它资源，对于这类对象，解初始化时可能无需做任何事情。例如，对于栈的实现来讲，在 stack_init()函数中仅对几个属性进行了赋值，没有额外占用其它任何资源，此时，stack_deinit()可能无需做任何事情，成为一个空函数，详见程序清单 1.10。

程序清单 1.10　栈的解初始化函数实现范例

```
1    int stack_deinit (struct stack *p_stack)
2    {
3        return 0;
4    }
```

在嵌入式系统中，经常会遇到一些与硬件相关的对象，其初始化时往往会占用一定的硬件资源： I/O 引脚、系统中断、系统总线……比如，某个 RTC（Real Time Clock）芯片具有闹钟功能，当闹钟事件发生时，其通过一个中断引脚通知主控 MCU。此时，若系统使用到 RTC 芯片的闹钟功能，则该 RTC 对象将占用一个中断引脚和一个中断资源，因此，在解初始化该 RTC 对象时，应该同时释放这些资源。

在实际应用中，读者也不必过于纠结解初始化函数究竟要做些什么事情，只需重点关注对象的初始化函数，查看其中是否分配、占用了某些资源。若有，则在解初始化函数中作相应的释放操作；若无，则解初始化函数留空。

将原 H 文件中的创建接口更新为初始化接口，删除接口更新为解初始化接口，更新后的 H 文件内容和 C 文件内容分别分别详见程序清单 1.11 和程序清单 1.12。

程序清单 1.11　栈实现头文件更新 2（stack.h）

```
1    #ifndef __STACK_H
2    #define __STACK_H
3
4    struct stack {                                  /* 类型定义                */
5        int             top;                        /* 栈顶                    */
6        int            *p_buf;                      /* 栈缓存                  */
7        unsigned int    size;                       /* 栈缓存的大小            */
```

```
8   };
9
10  int stack_init (struct stack *p_stack, int *p_buf, int size);      /* 初始化      */
11  int stack_push (struct stack *p_stack, int val);                   /* 入栈        */
12  int stack_pop (struct stack *p_stack, int *p_val);                 /* 出栈        */
13  int stack_deinit (struct stack *p_stack);                          /* 解初始化    */
14
15  #endif
```

程序清单 1.12　栈实现源文件更新（stack.c）

```
1   #include "stack.h"
2
3   int stack_init (struct stack *p_stack, int *p_buf, int size)
4   {
5       p_stack->top      = 0;
6       p_stack->size     = size;
7       p_stack->p_buf = p_buf;
8       return 0;
9   }
10
11  int stack_push (struct stack *p_stack, int val)
12  {
13      if (p_stack->top != p_stack->size) {
14          p_stack->p_buf[p_stack->top++] = val;
15          return 0;
16      }
17      return -1;
18  }
19
20  int stack_pop (struct stack *p_stack, int *p_val)
21  {
22      if (p_stack->top != 0) {
23          *p_val = p_stack->p_buf[--p_stack->top];
24          return 0;
25      }
26      return -1;
27  }
28
29  int stack_deinit (struct stack *p_stack)
30  {
31      return 0;
32  }
```

注：大量的实践应用表明，对于绝大部分纯软件（与硬件无关的软件），解初始化函数若保留，最终都会成为一个空函数。为了提高软件的简洁性，在部分类的设计中，可能删除了空的解初始化函数。但这里为了展示软件结构，依然保留的解初始化函数。

3. 销毁对象的顺序

在创建一个对象时，是先分配对象所需内存，再初始化对象。这个过程是自然形成的，并不需要作过多的说明，因为在初始化对象时，需要传递相应内存空间的首地址作为初始化函数的参数。这就保证了在初始化对象之前，必须完成相关内存的分配。而在销毁一个对象时，释放内存与调用解初始化函数并不能通过接口进行制约，因而必须进行特别说明。

值得注意的是，在销毁一个对象时，过程与创建对象恰恰相反，即应先解初始化对象，再释放对象占用的内存。因为在解初始化对象时，还会使用到对象中的数据，若先释放对象占用的内存，则对象在被解初始化之前，就被彻底销毁了，对象已经不存在了，显然无法再进行解初始化操作。

若内存来源于动态内存分配，则完整的应用程序范例详见程序清单 1.13。

程序清单 1.13　栈的使用范例（2）

```
#include "stack.h"
#include "stdio.h"
#include "stdlib.h"

int main ()
{
    int                 val;
    struct stack        *p_stack    =    (struct stack *)malloc(sizeof(struct stack));        // 动态内存分配
    int                 *p_buf      =    (int *)malloc(sizeof(int) * 20);

    // 初始化
    stack_init(p_stack, buf, 20);

    // 依次压入数据：2、4、5、8、10
    stack_push(p_stack, 2);
    stack_push(p_stack, 4);
    stack_push(p_stack, 5);
    stack_push(p_stack, 8);
    stack_push(p_stack, 10);

    // 依次弹出各个数据，并打印
    stack_pop(p_stack, &val); printf("%d ", val);
    stack_pop(p_stack, &val); printf("%d ", val);
    stack_pop(p_stack, &val); printf("%d ", val);
    stack_pop(p_stack, &val); printf("%d ", val);
    stack_pop(p_stack, &val); printf("%d ", val);

    // 解初始化
    stack_deinit(p_stack);
```

```
30
31          // 释放内存
32          free(p_stack);
33          free(p_buf);
34
35          return 0;
36      }
```

若内存来源于静态内存分配，则内存的分配和释放（如果可以释放）完全由系统自行完成，例如，内存以“局部变量”的形式分配，范例程序详见程序清单 1.14。

程序清单 1.14　栈的使用范例（3）

```
1       #include "stack.h"
2       #include "stdio.h"
3
4       int main ()
5       {
6           int                 val;
7           int                 buf[20];
8           struct stack        stack;
9           struct stack       *p_stack = &stack;
10          stack_init(p_stack, buf, 20);
11
12          // 依次压入数据：2、4、5、8、10
13          stack_push(p_stack, 2);
14          stack_push(p_stack, 4);
15          stack_push(p_stack, 5);
16          stack_push(p_stack, 8);
17          stack_push(p_stack, 10);
18
19          // 依次弹出各个数据，并打印
20          stack_pop(p_stack, &val); printf("%d ", val);
21          stack_pop(p_stack, &val); printf("%d ", val);
22          stack_pop(p_stack, &val); printf("%d ", val);
23          stack_pop(p_stack, &val); printf("%d ", val);
24          stack_pop(p_stack, &val); printf("%d ", val);
25
26          stack_deinit(p_stack);
27          return 0;
28      }
```

将其与程序清单 1.6 对比可以发现，在程序清单 1.6 中，使用一行代码（第 7 行）即可完成对象的创建，但在程序清单 1.14 中，需要使用多行代码（第 7 行 ~ 第 10 行）才能完成对象的创建，后续数据的出栈、入栈操作是完全相同的。此外，由于程序清单 1.14 中相关内存直接开辟在函数的运行栈中（局部变量），函数返回后会自动释放，也就无需再调用内存释放相关的函数。

从形式上看，虽然程序代码变得复杂了一些，但对象内存的来源更具有灵活性，使得栈的适用范围更加广泛。在部分系统中，在保证对象内存来源不受限制的同时，为了特殊情况下的便利性，往往还保留了基于动态内存分配创建对象的方法，在这种情况下，将同时提供 create 和 init 两套接口。

以 FreeRTOS 为例，其提供了两套创建任务的接口：xTaskCreate()和 xTaskCreateStatic()。其中，xTaskCreate()函数中采用动态内存分配的方法获得了任务相关内存：任务对象（任务控制块）和栈空间。而 xTaskCreateStatic()函数即用于以“静态”的方式创建任务，任务相关的内存需要用户通过函数的参数传递（实际上，该函数的作用就类似于 init 初始化函数，只不过其命名为了 Create）。

虽然初始化函数看起来和一般的方法函数没有区别，但是，初始化函数是在“创建对象”过程中被调用的。换句话说，初始化函数被调用时，对象还没有创建好，只有当初始化函数执行完毕后，对象才视为创建完毕。通常情况下，人们所提到的“类的方法”往往是作用于创建好的对象的。基于此，由于初始化函数的特殊作用，该函数通常不被视为是类提供的一般方法，不展示在类图中。同理，解初始化函数作为初始化函数的逆过程，表征了对象的销毁过程，一般也不视为类提供的一般方法，进而无需展示在类图中。简易的栈类图详见图 1.3。

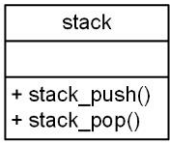


图 1.3　stack 类图

实际上，在绝大部分面向对象编程语言中，也有类似于初始化和解初始化的接口，以 C++为例，在定义类时，每个类都有两个特殊的函数：构造函数和析构函数。构造函数就相当于这里提到的初始化函数，其在创建一个对象时自动调用；析构函数就相当于这里提到的解初始化函数，其在销毁一个对象时自动调用。例如，以局部变量的形式定义一个对象，则在定义对象时，会自动调用构造函数；在退出当前作用域（函数返回）时，会自动调用析构函数。由此可见，面向对象编程语言为很多操作提供了语法特性上的原生支持，给实际编程带来了极大的便利。

1.3　继承

继承表示了一种类与类之间的特殊关系，即 **is-a** 关系（可理解为“是一个”、“是一种”），例如，姚明**是一个**篮球运动员，大象**是一种**哺乳动物，苹果**是一种**水果……

A **is-a** B，表明了 A 只是 B 的**一个**特例，并不是 B 的全部，就好比苹果（A）只是一种特殊的水果（B），还有很多其它水果（比如：梨子、香蕉、桃子等）。在 A is-a B 关系中，A（苹果）是**子类**，B（水果）是**父类**（又称基类、超类），表达了一般/特殊关系，A 是 B 的特殊化，而 B 是 A 的一般化。

子类是父类的一个特例，可以看作是在父类的基础上作了一些属性或方法的扩展，子类依然具有父类的属性和方法。基于此特点，可以使用继承关系，在一个已经存在的类的基础上，定义一个新类。新类将自动继承已存在类的属性和方法，并可根据需要，添加新的属性和方法。

继承使子类可以重用父类中已经实现的属性和方法，无需再重复设计和编程，以此实现代码最大限度的重用。代码复用是继承的主要作用，简言之，“继承”定义了一种方法，可以在某个作用域内（子类）对外部定义的某一组变量与函数进行覆盖（父类），使子类可以复用这些外部定义的变量与函数。

1.3.1 “继承”示例

在 C 语言编程中，往往在定义子类（子类结构体类型）时，通过将父类作为子类的第一个成员实现继承。之所以这样做，是因为在 C 语言结构体中，第一个成员（父类）的地

址和结构体自身（子类）的地址相同，当子类需要复用父类的方法时，子类的地址也可以作为父类的地址使用。

例如，在一个系统中，可能具有多个栈，为便于区分，每个栈可以具有不同的名称（系统栈、数据栈、符号栈……）。基于该需求，可以实现一个带名称的栈（为便于和普通栈区分，后文将其称之为“命名栈”），即在普通栈（详见程序清单 1.11）的基础上，增加一个“名称”属性，该属性使每个栈都具有一个可供识别的名称（类似于每个人的姓名），该栈类型的定义及接口声明详见程序清单 1.15，各接口的实现详见程序清单 1.16。

程序清单 1.15　命名栈头文件范例（stack_named.h）

```
1   #ifndef __STACK_NAMED_H
2   #define __STACK_NAMED_H
3
4   #include "stack.h"                                          /* 包含基类头文件   */
5
6   struct stack_named {
7       struct stack    super;                                  /* 基类（超类）     */
8       const char      *p_name;                                /* 栈名             */
9   };
10
11  /* 初始化 */
12  int stack_named_init (struct stack_named *p_stack, int *p_buf, int size, const char *p_name);
13
14  int stack_named_set (struct stack_named *p_stack, const char *p_name);      /* 设置名称         */
15  const char * stack_named_get (struct stack_named *p_stack);                 /* 获取名称         */
16
17  int stack_named_deinit (struct stack_named *p_stack);                       /* 解初始化         */
18  #endif
```

程序清单 1.16　命名栈源文件范例（stack_named.c）

```
1   #include "stack_named.h"
2
3   int stack_named_init (struct stack_named *p_stack, int *p_buf, int size, const char *p_name)
4   {
5       stack_init(&p_stack->super, p_buf, size);           /* 初始化基类       */
6       p_stack->p_name = p_name;                           /*  初始化子类成员  */
7       return 0;
8   }
9
10  int stack_named_set (struct stack_named *p_stack, const char *p_name)
11  {
12      p_stack->p_name = p_name;
13      return 0;
14  }
15
```

```
16  const char * stack_named_get (struct stack_named *p_stack)
17  {
18      return p_stack->p_name;
19  }
20
21  int stack_named_deinit (struct stack_named *p_stack)
22  {
23      return stack_deinit(&p_stack->super);                    /* 解初始化基类          */
24  }
```

通过程序清单 1.15 和程序清单 1.16 的代码可以发现，在实现“命名栈”时，除初始化函数和解初始化函数外，仅为新增的属性提供了设置和获取方法，栈的核心逻辑相关函数（入栈、出栈）无需再重复实现，入栈和出栈方法作为“命名栈”父类的方法，可以被复用。使用“命名栈”的应用程序范例详见程序清单 1.17。

程序清单 1.17　“命名栈”的使用范例（1）

```
1   #include "stack_named.h"
2   #include "stdio.h"
3
4   int main ()
5   {
6       int                 val;
7       int                 buf[20];
8       struct stack_named  stack_named;
9       struct stack_named  *p_stack_named = &stack_named;
10
11      stack_named_init(p_stack_named, buf, 20, "system");
12
13      printf("The stack name is %s!\n", stack_named_get(p_stack_named));
14
15      // 依次压入数据：2、4、5、8、10
16      stack_push((struct stack *)p_stack_named, 2);
17      stack_push((struct stack *)p_stack_named, 4);
18      stack_push((struct stack *)p_stack_named, 5);
19      stack_push((struct stack *)p_stack_named, 8);
20      stack_push((struct stack *)p_stack_named, 10);
21
22      // 依次弹出各个数据，并打印
23      stack_pop((struct stack *)p_stack_named, &val); printf("%d ", val);
24      stack_pop((struct stack *)p_stack_named, &val); printf("%d ", val);
25      stack_pop((struct stack *)p_stack_named, &val); printf("%d ", val);
26      stack_pop((struct stack *)p_stack_named, &val); printf("%d ", val);
27      stack_pop((struct stack *)p_stack_named, &val); printf("%d ", val);
28
29      stack_named_deinit (p_stack_named);
```

```
30          return 0;
31      }
```

程序中，因为父类（struct stack_named）和子类（struct stack）对应的类型并不相同，所以当父类方法（stack_push()、stack_pop()）作用于子类对象（stack_named）时，为了避免编译器输出“类型不匹配”的警告，必须对类型进行强制转换，比如入栈方法的使用：

```
stack_push((struct stack *)p_stack_named, 2);
```

否者，编译器可能会输出警告信息，例如：

```
warning: passing argument 1 of 'stack_push' from incompatible pointer type [-Wincompatible-pointer-types]
```

在 C 语言中，大量的使用类型强制转换存在一定的风险，例如，两个类之间没有继承关系，使用强制转换时将屏蔽编译器默认会输出的警告信息，导致这类错误在编译阶段无法发现。为了避免使用强制类型转换，可以多做一步操作，从子类中取出父类的地址进行传递，保证参数类型一致，例如：

```
stack_push(&p_stack_named->super, 2);
```

无论使用何种方法编程，看起来可能都没有那么优雅。这类问题的存在，主要是因为 C 语言并非真正的面向对象编程语言，使用 C 语言实现面向对象编程时，需要使用到一些看似“投机取巧”的手段。在真正的面向对象编程语言中，编译器可以识别继承关系，这种情况下，无需任何强制转换语句，父类的方法可以直接作用于子类。这是面向对象编程语言的优势，其在编程上为用户提供了相当程度的便利性。但无论怎样，核心思想还是基本一致的。使用 C 语言实现面向对象，由于部分工作需要手动完成，因而可以从一定程度上加深人们对面向对象编程的理解。今后在使用真正的面向对象编程语言时，也会更加得心应手。

在继承关系中，涉及到父类和子类两个类，在 UML 类图中，这两个类使用一个**空心三角箭头**连接，箭头指向的类是父类，通常情况下，父类在上，子类在下，示意图详见图 1.4。注意，子类中仅需表示出其扩展的属性和方法即可，无需重复展示父类中的属性和方法。

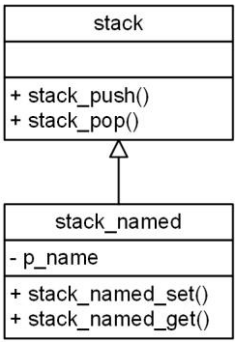


图 1.4　继承

实际上，继承还分为单重继承和多重继承。所谓单重继承，即子类只继承一个父类；所谓多重继承，即子类继承自多个父类，可以复用多个父类中的方法。为了避免二义性，通常都不建议使用多重继承。部分面向对象编程语言（例如 Java）也并不支持多重继承。因此，本书讨论的继承也仅限于单重继承。

1.3.2 进一步分析——初始化函数

回顾程序清单 1.16 中命名栈初始化函数的实现，详见程序清单 1.18。

程序清单 1.18　初始化函数实现范例

```
1   int stack_named_init (struct stack_named *p_stack, int *p_buf, int size, const char *p_name)
2   {
3       stack_init(&p_stack->super, p_buf, size);          /* 初始化基类        */
4       p_stack->p_name = p_name;                          /*   初始化子类成员  */
5       return 0;
6   }
```

其中先调用了父类的初始化函数（stack_init()），再初始化了命名栈特有的 p_name 属性。这里指出了一个隐含的规则：先初始化基类的成员，在初始化派生类特有的成员。该规

则与面向对象编程语言中构造函数的调用顺序是一致的：在建立一个对象时，首先调用基类的构造函数，然后再调用派生类的构造函数（使用面向对象编程语言时，在派生类的构造函数中仅需初始化派生类特有的成员，基类的构造函数是自动调用的，无需像命名栈初始化函数那样显式地调用）。

1.3.3 进一步分析——解初始化函数

由于解初始化函数（stack_named_deinit()）基本都是空的实现，因而看不出明显的顺序关系，但需要明确的是，解初始化的顺序与初始化的顺序是恰好相反的，应先对派生类中特有的数据“解初始化”，在对基类作解初始化操作。为了展示这个顺序，假设在解初始化函数中，命名栈需要将特有的 p_name 成员设置为 NULL，普通栈需要将 top 成员设置为 0，则解初始化函数的实现详见程序清单 1.19。

程序清单 1.19　解初始化函数实现范例

```
1    int stack_deinit (struct stack *p_stack)
2    {
3        p_stack->top = 0;
4        return 0;
5    }
6
7    int stack_named_deinit (struct stack_named *p_stack)
8    {
9        p_stack->p_name = NULL;
10       return stack_deinit(&p_stack->super);                    /* 解初始化基类          */
11   }
```

该规则与面向对象编程语言中“析构函数”的调用顺序是一致的：在销毁一个对象时，首先调用派生类的析构函数，然后再调用基类的析构函数。

1.3.4 最少知识原则

所谓 “最少知识原则”，通俗地讲就是，对一个类的使用者而言，不管你（类）的内部如何复杂都和我没关系，我只调用你提供的这些公有方法，其他的我一概不管。

显然，目前实现的“命名栈”并非如此，对于命名栈的使用者来讲，其必须知道命名栈与普通栈之间的继承关系，进而才可以正确的使用普通栈的入栈方法，操作命名栈，例如：

```
stack_push((struct stack *)p_stack_named, 2);
```

这对用户来说并不友好，因为其使用的是“命名栈”类（stack_named.h），却还要关心“普通栈”类（stack.h）。为了满足“最少知识原则”，命名栈也可以提供入栈和出栈方法，使用户仅需关心命名栈的公共接口，就可以完成命名栈的所有操作。当然，入栈和出栈方法可以直接复用普通栈的入栈和出栈方法，更新 stack_named.h 文件，增加两个接口，详见程序清单 1.20。

程序清单 1.20　命名栈头文件更新（stack_named.h）

```
1    #ifndef __STACK_NAMED_H
2    #define __STACK_NAMED_H
3
4    #include "stack.h"                                          /* 包含基类头文件   */
```

```
5
6   struct stack_named {
7       struct stack     super;                                  /* 基类（超类）     */
8       const char      *p_name;                                 /* 栈名             */
9   };
10
11  /* 初始化 */
12  int stack_named_init (struct stack_named *p_stack, int *p_buf, int size, const char *p_name);
13
14  int stack_named_set (struct stack_named *p_stack, const char *p_name);     /* 设置名称         */
15  const char * stack_named_get (struct stack_named *p_stack);                /* 获取名称         */
16
17  static inline int stack_named_push (struct stack_named *p_stack, int val)
18  {
19      return stack_push(&p_stack->super, val);
20  }
21
22  static inline int stack_named_pop (struct stack_named *p_stack, int *p_val)
23  {
24      return stack_pop(&p_stack->super, p_val);
25  }
26
27  int stack_named_deinit (struct stack_named *p_stack);                      /* 解初始化         */
28  #endif
```

头文件中增加了两个方法：stack_named_push()和 stack_named_pop()。程序中，由于这两个函数非常简单，只是调用了其父类中相应的方法，仅一行代码，因而使用了内联函数的形式，如此可以优化代码大小和执行速度。

经过这样简单的包装之后，用户使用的所有方法都是作用于“命名栈”对象的，无需再使用类型强制转换等特殊的方法。更新后的“命名栈”使用范例详见程序清单 1.21。

程序清单 1.21　“命名栈”的使用范例（2）

```
1   #include "stack_named.h"
2   #include "stdio.h"
3
4   int main ()
5   {
6       int                 val;
7       int                 buf[20];
8       struct stack_named  stack_named;
9       struct stack_named *p_stack_named = &stack_named;
10
11      stack_named_init(p_stack_named, buf, 20, "system");
12
13      printf("The stack name is %s!\n", stack_named_get(p_stack_named));
```

```
14
15          // 依次压入数据：2、4、5、8、10
16          stack_named_push(p_stack_named, 2);
17          stack_named_push(p_stack_named, 4);
18          stack_named_push(p_stack_named, 5);
19          stack_named_push(p_stack_named, 8);
20          stack_named_push(p_stack_named, 10);
21
22          // 依次弹出各个数据，并打印
23          stack_named_pop(p_stack_named, &val); printf("%d ", val);
24          stack_named_pop(p_stack_named, &val); printf("%d ", val);
25          stack_named_pop(p_stack_named, &val); printf("%d ", val);
26          stack_named_pop(p_stack_named, &val); printf("%d ", val);
27          stack_named_pop(p_stack_named, &val); printf("%d ", val);
28
29          stack_named_deinit (p_stack_named);
30
31          return 0;
32      }
```

从用户编程角度看，包装后的“命名栈”对用户来讲更加友好（无需类型强制转换）。但在实际编程过程中，若所有继承关系都要这样封装一遍，可能显得有些繁琐、累赘。因此，一般来讲，对用户开放的类才需要这样做，如果某些类无需对用户开放，仅在内部使用，则可以酌情省略这些包装过程。

1.4 多态

所谓多态，其字面含义就是具有“多种形式”。从调用者的角度看对象，会发现它们非常相似，难以区分，但是这些被调用对象的内部处理实际上却各不相同。换句话说，各个对象虽然内部处理不同，但对于使用者（调用者）来讲，它们却是相同的。

1.4.1 学生的“自我介绍”

在前面对类的介绍（1.1.2）中，提到了学生类，其包含姓名、学号、性别、身高、体重等属性，并对外提供了一个“自我介绍”方法。按照栈的实现方法，可以编写对应的代码，范例程序详见程序清单 1.22 和程序清单 1.23。

程序清单 1.22　学生类定义（student.h）

```
1   #ifndef __STUDENT_H
2   #define __STUDENT_H
3
4   struct student {
5           char            name[10];           /* 姓名 （假定最长 10 字符）      */
6           unsigned int    id;                 /* 学号                           */
7           char            sex;                /* 性别:'M'，男；'F' ，女         */
8           float           height;             /* 身高                           */
9           float           weight;             /* 体重                           */
10  };
```

```
11
12  int student_init (struct student       *p_student,
13                    char                 *p_name,
14                    unsigned int          id,
15                    char                  sex,
16                    float                 height,
17                    float                 weight);
18
19  int student_self_introduction (struct student *p_student);
20
21  #endif
```

程序清单 1.23　学生类实现（student.c）

```
1   #include "student.h"
2   #include "stdlib.h"
3   #include "stdio.h"
4   #include "string.h"
5
6
7   int student_init (struct student *p_student,
8                     char                *p_name,
9                     unsigned int       id,
10                    char                 sex,
11                    float               height,
12                    float               weight)
13  {
14      memcpy(p_student->name, p_name, strlen(p_name));
15
16      p_student->id         = id;
17      p_student->sex        = sex;
18      p_student->height = height;
19      p_student->weight = weight;
20
21      return 0;
22  }
23
24  int   student_self_introduction (struct student *p_this)
25  {
26   printf("Hi! My name is %s, I'm a %s. My school number is %d. My height is %fcm and weight is %fkg.\n",
27              p_this->name,
28             (p_this->sex == 'M') ? "boy" : "girl",
29             p_this->id,
30             p_this->height,
31             p_this->weight);
```

```
32
33          return 0;
34      }
```

假设一个常见的场景：开学第一课老师往往不会上新课，很可能让班上所有同学依次作一个简单的自我介绍，以便同学之间相互认识。第一节课的内容就很简单了，调用所有同学的自我介绍方法即可，范例程序详见程序清单 1.24。

程序清单 1.24　“第一节课”实现范例（1）

```
1       void first_class (struct student *p_students, int num)
2       {
3               int i;
4               for (i = 0; i < num; i++) {
5                       student_self_introduction(&p_students[i]);
6               }
7       }
```

调用该函数前，需要将所有学生对象创建好，并存于一个数组中，假定一个班级有 50 个学生，则调用示意代码详见程序清单 1.25。

程序清单 1.25　“第一节课”调用范例

```
1       int main ()
2       {
3               struct student student[50];
4
5               /* 根据每个学生的信息，依次创建各个学生对象 */
6               student_init(&student[0], "zhangsan", 2010447222, 'M', 173, 60);
7               student_init(&student[1], "lisi",         2010447223, 'F', 168, 65);
8
9               // ...
10
11              /* 上第一节课 */
12              first_class(student, 50);
13      }
```

在上面的实现代码中，假定了学生的“自我介绍”格式是完全相同的，均是将个人信息陈述了一遍，显然，这样的自我介绍是“十分无趣”的，也体现不出每个学生的个性和差异。一些积极的学生，可能期望作出一个更加全面的自我介绍。例如，一个名叫张三的学生，其期望这样介绍自己：

“亲爱的老师，同学们！我叫张三，来自美丽的四川，今年 18 岁，是一个自信，开朗，友好，积极向上的人，我有着广泛的兴趣爱好，喜欢健身、打篮球、看书、下棋、听音乐，特别对主持和街舞有着浓厚的兴趣，希望能和同学们做好朋友，一起学习，一起玩耍……”

由此可见，每个学生可能有自己的一套自我介绍方法，并不期望千篇一律。若不基于多态的思想实现，最简单的方法可能是每个学生都提供一个自我介绍方法，例如：student_zhangsan_introduction()。显然，这种情况下，每个学生提供的方法都不相同（函数名不同），根本无法统一调用，此时，第一节课的上课内容将会大改，可能需要依次调用每

个不同学生提供的自我介绍方法，例如：

```
void first_class ()
{
    student_zhangsan_introduction(&zhangshan);          // 张三自我介绍
    student_lisi_introduction(&lisi);                   // 李四自我介绍
   // ….
}
```

简言之，无法使用同样的调用形式（函数）完成每个不同对象的“自我介绍”。对于调用者来讲，复杂度将急剧提升，其需要关注每个对象提供的特殊方法。使用多态的思想即可很好的解决这个问题，进而保证 fisrt_class()的内容不变，换言之，虽然每个对象方法的实现不同，但可以使用同样的形式调用它。

在 C 语言中，函数指针就是解决这个问题的“利器”，函数指针的原型决定了调用方法，例如，定义如下函数指针：

```
int (*student_self_introduction) (struct student *p_student);
```

无论该函数指针指向何处，都表示该函数指针指向的是 int 类型返回值，具有一个 p_student 参数的函数，其调用形式如下：

```
student_self_introduction(p_student);
```

函数指针的指向代表了函数的实现，显然，指向不同的函数就代表了不同的实现。基于此，为了使每个学生对象可以有自己独特的介绍方式，在学生类的定义中，可以不实现自我介绍方法，但可以通过函数指针约定自我介绍方法的调用形式。更新学生类的定义，详见程序清单 1.26。

程序清单 1.26　学生类定义更新（student.h）

```
1   #ifndef __STUDENT_H
2   #define __STUDENT_H
3
4   struct student {
5         int           (*student_self_introduction) (struct student *p_student);
5       char            name[10];                   /* 姓名 （假定最长 10 字符）      */
6       unsigned int    id;                         /* 学号                           */
7       char            sex;                        /* 性别:'M'，男；'F' ，女         */
8       float           height;                     /* 身高                           */
9       float           weight;                     /* 体重                           */
10  };
11
12  int student_init (struct student    *p_student,
13                    char              *p_name,
14                    unsigned int      id,
15                    char              sex,
16                    float             height,
17                    float             weight,
18                    int               (*student_self_introduction) (struct student *));
18
```

```
19  /* 学生类提供的自我介绍方法 */
20  static inline
21  int student_self_introduction (struct student *p_student)
22  {
23      return p_student->student_self_introduction(p_student);
24  }
25
26  #endif
```

此时，对于外界来讲，学生类“自我介绍方法”的调用形式并未发生任何改变，函数原型还是一样的（由于只有一行代码，因而以内联函数的形式存放到了头文件中）。基于此，“第一节课的内容”可以保持完全不变（详见程序清单 1.24）。

在这种方式下，每个对象在初始化时，需要指定自己特殊的自我介绍方法，例如张三对象的创建过程为：

```
1   int student_zhangsan_introduction(struct student *p_student)
2   {
3       const char *str = "亲爱的老师，同学们，我叫张三，来自美丽的四川，今年 18 岁，"
4                         "是一个自信，开朗，友好，积极向上的人，我有着广泛的兴趣爱"
5                         "好，喜欢健身、打篮球、看书、下棋、听音乐，特别对主持和街"
6                         "舞有着浓厚的兴趣，希望能和同学们做好朋友，一起学习，一起"
7                          "玩耍……";
8       printf("%s\n", str);
9       return 0;
10  }
11
12  int main ()
13  {
14      struct student student[50];
15
16      /* 根据每个学生的信息，依次创建各个学生对象 */
17      student_init(&student[0], "zhangsan", 2010447222, 'M', 173, 60, student_zhangsan_introduction);
18      // ...
19
20      /* 上第一节课 */
21      first_class(student, 50);
22  }
```

由此可见，多态的核心是：对于上层调用者，不同的对象可以使用完全相同的操作方法，但是每个对象可以有各自不同的实现方式。

多态是面向对象编程非常重要的特性，C 语言依赖**指针**实现多态，因此，指针在面向对象编程中，起到了非常重要的作用。指针一直以来被视为 C 语言的精华。熟练的使用指针，是一个 C 程序员的基本要求。

上面的例子只是引出了多态的概念，使读者对多态有一个初步的认识。虽然函数指针同属性一样，放在结构体类型的定义中，但函数指针不能视为普通的属性，其应视为一种方法（是被用来调用的），在面向对象编程中，通常将类中的函数指针称之为抽象方法（因其在

定义类时没有实现，所以是“抽象”的）。此外，往往还会将函数指针封装到一个特殊的类（抽象类）中。在后文更多的示例讲解中还会进一步展现。

1.4.2 I/O 设备驱动

在使用 C 编程时，都非常喜欢使用 printf()打印一些信息，例如 C 程序员接触的第一个程序（输出“Hello World”字符串），范例程序详见程序清单 1.27。

程序清单 1.27 “Hello World!”范例程序

```
1    #include "stdio.h"
2
3    int main ()
4    {
5        printf("Hello World!\n");
6        return 0;
7    }
```

在 PC 上运行这一段程序时，字符串信息可能输出到控制台中；而在一些嵌入式系统中，字符串信息可能会通过某个串口输出，进而在 PC 上通过一些串口调试相关的软件查看这些字符串信息。

printf()函数的解释是输出信息至 STDOUT（标准输出）。为了实现这个功能，STDOUT 可能会提供一个字符串输出方法，然后 printf()函数在内部调用了这个方法。显然，这里的 printf()函数就具有多态性，对于用户来讲，其调用形式是确定的，但内部具体输出信息到哪里，却会随着 STDOUT 的不同而不同。

接下来，描述 STDOUT 常见的一种实现。在很多操作系统中（比如，Linux），硬件设备（例如串口、温度传感器、ADC 数据采集等等）的操作方法都和文件操作方法类似（可以视为：一切皆文件），都可以通过 open()、close()、read()、write()、seek()等几个标准函数进行操作。为了达到这个目的，统一 I/O 设备的使用方法，系统强制要求每个 I/O 设备都要提供 open()、close()、read()、write()、seek()这几个标准函数的实现。也就是说，每个 I/O 设备的驱动程序，对这些标准函数的实现在函数调用上必须保持一致。

显然，在 C 语言中，不能同时出现多个同名函数，如果两个 I/O 设备都对外提供一个名为 open()的函数，势必为在链接时产生冲突。这本质上就是一个多态问题：即以同样的方法使用各个不同的 I/O 设备。为此，可以通过函数指针解决这个问题。

首先，定义 FILE 数据结构体，其中包含了相对应的 5 个函数指针，分别用于指向各个 I/O 设备针对这 5 个函数的实现。FILE 结构体类型定义详见程序清单 1.28。

程序清单 1.28 FILE 结构体类型定义（file.h）

```
1    struct FILE {
2        void (*open) (char *name, int mode);
3        void (*close) ();
4        int    (*read) ();
5        void (*write) ();
6        void (*seek) (long index, int mode);
7    };
```

注：这里仅作原理性展示，对这些函数指针所指向函数的参数、返回值并没有做出完整的定义。

基于此，对于控制台设备，其 I/O 驱动程序就会提供这 5 个函数的实际定义，并将 FILE 结构体的函数指针指向这些对应的函数实现，范例程序详见程序清单 1.29。

程序清单 1.29　控制台设备 I/O 驱动（console.c）

```
1    #include "file.h"
2
3    static void open (char *name, int mode)
4    {
5        // 实现代码
6    }
7
8    static void close ()
9    {
10       // 实现代码
11   }
12
13   static int read ()
14   {
15       // 实现代码
16   }
17
18   static void write ()
19   {
20       // 实现代码
21   }
22
23   static void seek (long index, int mode)
24   {
25       // 实现代码
26   }
27
28   struct FILE console = {open, close, read, write, seek};
```

程序仅作示意，并没有添加任何实际的实现代码。其中所有的函数实现都增加了 static 修饰符，保证这些函数的命名不会污染命名空间，进而避免与外部的一些函数产生命名冲突。对于该控制台设备，仅对外提供了一个可以使用的 FILE 对象：console。

现在，如果 STDOUT 的类型是 FILE *，且系统期望输出信息至控制台，则可以将 STDOUT 指向 console，即：

```
exttern struct FILE console;
struct FILE * STDOUT = &console;
```

基于此，在 printf()函数的实现中，可能最后通过调用 STDOUT->write() 实现字符串信息的输出。当然，通常情况下，printf()函数的实现较为复杂，因为其重点要实现字符串的格式化（例如，将其中的%d 使用整数替换等），因此，通常将往 STDOUT 输出字符信息的操作封装为一个函数，例如：

```
void putchar (char c)
{
    STDOUT->write(&c, 1);
}
```

在 printf()函数的实现中，调用该函数即可。无论实现形式如何，从本质上来看，最终都是通过 STDOUT 中的 write()函数实现了数据的输出。在不同系统中，当需要将 printf()输出的信息以不同的方式输出时，仅需修改 STDOUT 的指向即可。

上面展示了设备 I/O 的一般管理方法，其中的编程方法或技巧正是面向对象编程中多态的基础，也再一次展现了函数指针在多态中的重要地位。在 C 语言编程中，多态可以视为函数指针的一种典型应用。

1.4.3 带检查功能的栈

在 1.2 节中，实现了一个简单的栈（详见程序清单 1.11），其重点完成了栈的核心逻辑（入栈和出栈），假设现在增加了需求，需要实现一个带检查功能的栈，即：在数据入栈之前，必须进行特定的检查，“检查通过”后才能压人栈中。检查方式可能多种多样，假定目前有以下几种检查方式：

- 范围检查：必须在特定的范围之内，比如：1 ~ 9，才视为检查通过；
- 奇偶检查：必须是奇数（或者偶数），才视为检查通过；
- 变化检查：值必须增加（比上一次的值大），才视为检查通过。

1. 基于继承实现“带范围检查功能”的栈

先不考虑多种检查方式，仅考虑范围检查。参照“命名栈”的实现，使用继承方式，在普通栈的基础上，可以很快实现一个新类，范例程序详见程序清单 1.30 和程序清单 1.31。

程序清单 1.30　带范围检查的栈 H 文件（stack_with_range_check.h）

```
1   #ifndef __STACK_WITH_RANGE_CHECK_H
2   #define __STACK_WITH_RANGE_CHECK_H
3
4   #include "stack.h"                                          /* 包含基类头文件   */
5
6   struct stack_with_range_check {
7       struct stack        super;                              /* 基类（超类）     */
8       int                 min;                                /* 最小值           */
9       int                 max;                                /* 最大值           */
10  };
11
12  int stack_with_range_check_init (struct stack_with_range_check      *p_stack,
13                                   int                                *p_buf,
14                                   int                                 size,
15                                   int                                 min,
16                                   int                                 max);
17
18  int stack_with_range_check_push (struct stack_with_range_check *p_stack, int val);          /* 入栈    */
19
```

```
20  int stack_with_range_check_pop (struct stack_with_range_check *p_stack, int *p_val);       /* 出栈    */
21
22  #endif
```

程序清单 1.31　带范围检查的栈 C 文件（stack_with_range_check.c）

```
1   #include "stack_with_range_check.h"
2
3   int stack_with_range_check_init (struct stack_with_range_check       *p_stack,
4                                    int                                 *p_buf,
5                                    int                                  size,
6                                    int                                  min,
7                                    int                                  max)
8   {
9       /* 初始化基类 */
10      stack_init(&p_stack->super, p_buf, size);
11
12      /*   初始化子类成员 */
13      p_stack->min = min;
14      p_stack->max = max;
15      return 0;
16  }
17
18  int stack_with_range_check_push (struct stack_with_range_check *p_stack, int val)
19  {
20      if ((val >= p_stack->min) && (val <= p_stack->max)) {
21          return stack_push(&p_stack->super, val);
22      }
23      return -1;
24  }
25
26  int stack_with_range_check_pop (struct stack_with_range_check *p_stack, int *p_val)
27  {
28      return stack_pop(&p_stack->super, p_val);
29  }
```

注：程序中，假定范围值仅在初始化时指定，因而没有提供其它访问 min 和 max 的接口。此外，为了接口的简洁性，没有再展示解初始化函数的定义。虽然只在入栈时作检查，出栈和普通栈是完全相同的，但基于最小知识原则，也封装了一个 pop 接口，使该类的用户完全不需要关心普通栈。

该实现过程相对比较简单，一些特殊的属性（最大值，最小值）可以直接添加到新类的定义中，入栈和出栈方法都可以复用普通栈提供的相关方法。使用该栈的范例程序详见程序清单 1.32。

程序清单 1.32　带范围检查的栈使用范例（1）

```
1   #include "stack_with_range_check.h"
2   #include "stdio.h"
```

```
3
4   int main ()
5   {
6       int                     val;
7       int                     buf[20];
8       int                     i;
9       int                     test_data[5] = {2, 4, 5, 3, 10};
10
11      struct stack_with_range_check    stack;
12      struct stack_with_range_check *p_stack = &stack;
13
14      stack_with_range_check_init(p_stack, buf, 20, 1, 9);                    /* 允许范围，1 ~ 9 */
15
16      for (i = 0; i < 5; i++) {
17          if (stack_with_range_check_push(p_stack, test_data[i]) != 0) {
18              printf("The data %d push failed!\n", test_data[i]);
19          }
20      }
21
22      printf("The pop data: ");
23      while (1) {                                                             /* 弹出所有数据      */
24          if (stack_with_range_check_pop(p_stack, &val) == 0) {
25              printf("%d    ", val);
26          } else {
27              break;
28          }
29      }
30      return 0;
31  }
```

依照这个方法，可以继续实现其它检查方式的栈。由于核心是实现带检查功能的入栈函数，因而这里仅简单展示另外两种检查方式下入栈函数的实现，分别详见程序清单 1.33 和程序清单 1.34。

程序清单 1.33　奇偶检查入栈函数范例程序

```
1   int stack_with_oddeven_check_push (struct stack_with_oddeven_check *p_stack, int val)
2   {
3       if (((p_stack->iseven) && ((val % 2) == 0)) ||   ((!p_stack->iseven) && ((val %2) != 0))) {
4           return stack_push(&p_stack->super, val);   // 检查通过：偶校验且为偶数或奇校验且为奇数
5       }
6       return -1;
7   }
```

程序清单 1.34　变化检查入栈函数范例程序

```
1   int stack_with_change_check_push (struct stack_with_change_check *p_stack, int val)
```

```
2     {
3         if (((p_stack->pre_value < val) {
4             p_stack->pre_value = val;
5             return stack_push(&p_stack->super, val);          // 检查通过，本次入栈值大于上一次的值
6         }
7         return -1;
8     }
```

由此可见，这种实现方式存在一定的缺陷，因为不同检查方法对应的入栈函数均不相同，所以对于用户来讲，使用不同的检查功能，就必须调用不同的入栈函数。但是，观察上面实现的几个入栈函数，可以发现它们的入栈方法基本都是类似的，示意代码如下：

```
int stack_XXX_push (struct stack_XXX *p_stack, int val)
{
    if（检查通过）{
        return stack_push(&p_stack->super, val);
    }
    return -1;
}
```

显然，如果使用多态思想，将“检查”函数的调用形式标准化，那么就可以编写一个通用的、与具体检查方式无关的入栈函数。

2. 基于多态实现通用的“带检查功能的栈”

参考程序清单 1.26 中的做法，使用函数指针表示“检查功能”，函数指针可以指向不同的检查函数。基于此，可以定义一个包含函数指针的类：

```
struct stack_with_validate {
    struct stack    super;                                                  /* 基类（超类）      */
    int             (*validate) (struct stack_with_validate *p_this, int val);  /* 检查函数          */
};
```

注：和其它普通方法一样，类中抽象方法（函数指针）的第一个成员同样是指向该类对象的指针。

此时，数据入栈前的检查工作，完全交给 validate 指针所指向的函数实现。假定其指向的函数在检查数据时，返回 0 表示检查通过，其它值表示检查未通过。完整的带检查功能的栈实现范例详见程序清单 1.35 和程序清单 1.36。

程序清单 1.35　带检查功能的栈 H 文件（stack_with_validate.h）

```
1     #ifndef __STACK_WITH_VALIDATE_H
2     #define __STACK_WITH_VALIDATE_H
3
4     #include "stack.h"                                                    /* 包含基类头文件 */
5
6     struct stack_with_validate {
7         struct stack        super;                                      /* 基类（超类）     */
8         int                 (*validate) (struct stack_with_validate *p_this, int val);
9     };
10
11    int stack_with_validate_init (struct stack_with_validate      *p_stack,
```

```
12                                  int                             *p_buf,
13                                  int                              size,
14                                  int                            (*validate) (struct stack_with_validate *, int ));
15
16  int stack_with_validate_push (struct stack_with_validate *p_stack, int val);                    /* 入栈    */
17  int stack_with_validate_pop (struct stack_with_validate *p_stack, int *p_val);                   /* 出栈    */
18
19  #endif
```

程序清单 1.36　带检查功能的栈 C 文件（stack_with_validate.c）

```
1   #include "stack_with_validate.h"
2   #include "stdio.h"
3
4   int stack_with_validate_init (struct stack_with_validate    *p_stack,
5                                   int                         *p_buf,
6                                   int                          size,
7                                   int                        (*validate) (struct stack_with_validate *, int ))
8   {
9       /* 初始化基类 */
10      stack_init(&p_stack->super, p_buf, size);
11      p_stack->validate = validate;
12      return 0;
13  }
14
15  int stack_with_validate_push (struct stack_with_validate *p_stack, int val)
16  {
17      if ((p_stack->validate == NULL) || (p_stack->validate(p_stack, val) == 0)) {
18          return stack_push(&p_stack->super, val);
19      }
20      return -1;
21  }
22
23  int stack_with_validate_pop (struct stack_with_validate *p_stack, int *p_val)
24  {
25      return stack_pop(&p_stack->super, p_val);
26  }
```

其它具体的带某种检查功能的栈，重点是实现其中定义的 validate 方法。基于此，可以更新带范围检查的栈实现，范例程序详见程序清单 1.37 和程序清单 1.38。

程序清单 1.37　带范围检查的栈 H 文件更新（stack_with_range_check.h）

```
1   #ifndef __STACK_WITH_RANGE_CHECK_H
2   #define __STACK_WITH_RANGE_CHECK_H
3
4   #include "stack_with_validate.h"                                    /* 包含基类头文件    */
```

```
5
6   struct stack_with_range_check {
7           struct stack_with_validate      super;                  /* 基类（超类）     */
8           int                             min;                    /* 最小值           */
9           int                             max;                    /* 最大值           */
10  };
11
12  struct stack_with_validate * stack_with_range_check_init (struct stack_with_range_check *p_stack,
13                                                            int                           *p_buf,
14                                                            int                            size,
15                                                            int                            min,
16                                                            int                            max);
17  #endif
```

程序清单 1.38　带范围检查的栈 C 文件更新（stack_with_range_check.c）

```
1   #include "stack_with_range_check.h"
2
3   static int __validate (struct stack_with_validate *p_this, int val)
4   {
5           struct stack_with_range_check *p_stack = (struct stack_with_range_check *)p_this;
6
7           if ((val >= p_stack->min) && (val <= p_stack->max)) {
8                   return 0;                                                   /* 检查通过 */
9           }
10          return -1;
11  }
12
13  struct stack_with_validate * stack_with_range_check_init (struct stack_with_range_check    *p_stack,
14                                                            int                           *p_buf,
15                                                            int                            size,
16                                                            int                            min,
17                                                            int                            max)
18  {
19          /* 初始化基类 */
20          stack_with_validate_init(&p_stack->super, p_buf, size, __validate);
21
22          /*  初始化子类成员 */
23          p_stack->min = min;
24          p_stack->max = max;
25          return 0;
26  }
```

带范围检查的栈，主要目的就是实现“检查功能”对应的函数：__validate，并将其作为 validate 函数指针（抽象方法）的值。

这里看起来也是一种继承关系，的确如此。在面向对象编程中，包含抽象方法的类通常称之为抽象类，抽象类不能直接实例化（因为其还有方法没有实现），抽象类只能被继承，且由子类实现其中定义的抽象方法。

在 UML 类图中，抽象类的类名和其中的抽象方法均使用*斜体*表示，普通栈、带检查功能的栈和带范围检查的栈，它们之间的关系详见图 1.5。

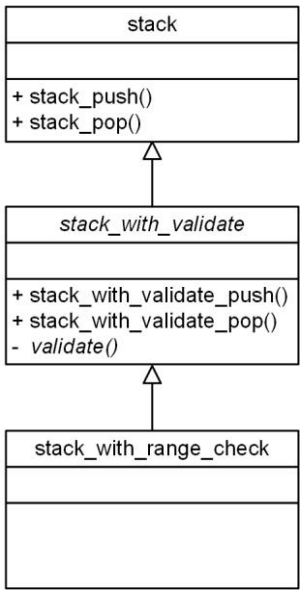


图 1.5　三个类之间的关系

带范围检查的栈，其主要作用是实现其父类中定义的抽象方法，进而创建一个真正的“带检查功能”的栈对象（此时的抽象方法已实现），该对象即可提交给外部使用。带范围检查的栈并没有其他特殊的方法，因而在其初始化完成后，通过初始化函数的返回值向外界提供了一个“带检查功能”的栈对象，后续用户即可使用 stack_with_validate.h 文件中的 push 和 pop 方法操作该对象。范例程序详见程序清单 1.39。

程序清单 1.39　带范围检查的栈使用范例（2）

```
1   #include "stack_with_range_check.h"
2   #include "stdio.h"
3
4   int main ()
5   {
6       int                 val;
7       int                 buf[20];
8       int                 i;
9       int                 test_data[5] = {2, 4, 5, 3, 10};
10
11      struct stack_with_range_check    stack;
12      struct stack_with_validate *p_stack = stack_with_range_check_init(&stack, buf, 20, 1, 9);
13
14      for (i = 0; i < 5; i++) {
15          if (stack_with_validate_push(p_stack, test_data[i]) != 0) {
16              printf("The data %d push failed!\n", test_data[i]);
17          }
18      }
19      printf("The pop data: ");
20      while (1) {                                                    /* 弹出所有数据 */
21          if (stack_with_validate_pop(p_stack, &val) == 0) {
22              printf("%d    ", val);
23          } else {
24              break;
25          }
```

```
26          }
27          return 0;
28      }
```

不难推论，无论何种检查方式，其主要目的都是创建“带检查功能”的栈对象（完成抽象方法的实现）。创建完毕后，对于用户来讲，操作方法都是完全相同的，与检查方式无关。为避免赘述，这里不再实现另外两种检查功能的栈，仅展示出他们的类图，详见图 1.6。实际上，基于程序清单 1.33 和程序清单 1.34 中实现的检查方法，以及程序清单 1.37 和程序清单 1.38 展示的“带范围检查的栈”，完成 stack_with_oddeven_check 和 stack_with_change_check 类的实现是非常容易的，读者可以自行尝试。

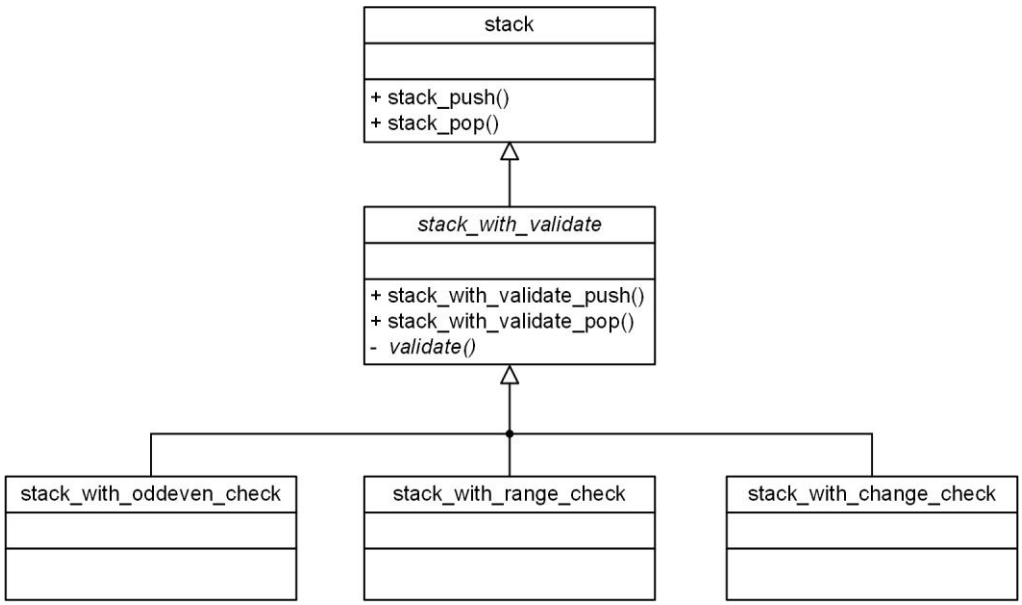


图 1.6　三种不同检查功能的栈

程序清单 1.39 所示的范例程序较为简单，通过栈的初始化函数（第 12 行）可以看出，当前“带检查功能的栈”使用的是“范围检查”。实际上，在一些大型项目中，初始化过程往往和实际的应用程序是分离的，也就是说，对于用户来讲，其仅会获取到一个 struct stack_with_validate *类型的指针，其指向某个“带检查功能的栈”，实际检查什么，用户可能并不需要关心，应用程序基于该类型指针编程，将使应用程序与具体检查功能无关，即使后续更换为其它检查方式，应用程序也不需要做任何改动，例如，一个简单的纯应用程序（仅应用代码，不包含初始化部分）范例详见程序清单 1.40。

程序清单 1.40　带检查功能的栈应用程序范例

```
1   #include "stack_with_validate.h"
2
3   int stack_validate_application (struct stack_with_validate *p_stack)
4   {
5           int i;
6           int val;
7           int test_data[5] = {2, 4, 5, 3, 10};
8
9           for (i = 0; i < 5; i++) {
```

```
10              if (stack_with_validate_push(p_stack, test_data[i]) != 0) {
11                  printf("The data %d push failed!\n", test_data[i]);
12              }
13          }
14
15          printf("The pop data: ");
16          while (1) {                                                    /* 弹出所有数据 */
17              if (stack_with_validate_pop(p_stack, &val) == 0) {
18                  printf("%d    ", val);
19              } else {
20                  break;
21              }
22          }
23          return 0;
24      }
```

该应用程序仅使用了 stack_with_validate.h 文件中的接口，与具体检查方式无关，无论何种检查方式，都可以使用这段程序对栈进行测试。

3. 分离检查功能（定义校验器类）

在前面的实现中，将检查功能视为栈的一种扩展（因而多处使用到了继承），检查逻辑直接在相应的扩展类中实现。这就致使检查功能与栈绑定在一起，检查功能的实现无法独立复用。如果要实现一个“**带检查功能的队列**”，同样是上述的 3 种检查逻辑，自然而言的期望能够复用检查逻辑相关的代码。显然，由于当前检查逻辑的实现与栈捆绑在一起，无法单独拿出来复用。

检查功能与栈的绑定，主要在“带检查功能的栈”中体现，回顾该类的定义如下：

```
struct stack_with_validate {
    struct stack    super;                                          /* 基类（超类）      */
    int             (*validate) (struct stack_with_validate *p_this, int val);    /* 检查函数          */
};
```

其中，super 主要用于继承自普通栈，validate 表示一个抽象的数据检查方法，不同的检查方法，主要通过该指针所指向的函数体现。由于检查方法（validate）是该类的一个方法，显然，检查逻辑就与栈绑定了。

为了使检查逻辑不与栈绑定，需要将它们分离。为此，可以将检查逻辑放到独立的与栈无关的类中，例如，定义一个抽象的校验器类，来专门表示数据检查逻辑：

```
struct validator {
    int    (*validate) (struct validator *p_this, int val);         /* 检查函数          */
};
```

虽然该类中还是仅包含 validate 函数指针，但是需要注意该函数指针类型的变化，其第一个参数为指向校验器的指针，而在“带检查功能的栈”中，其第一个参数是指向“带检查功能的栈”的指针，这反映出该函数指针（抽象方法）所属类的不同。

通过该类的定义，明确的将检查逻辑封装到独立的校验器类中，与栈再无任何关系。不同的检查逻辑，可以在其子类中实现，在实现各个子类之前，参照图 1.6，可以很容易画出校验器类和各个子类之间的关系，详见图 1.7。

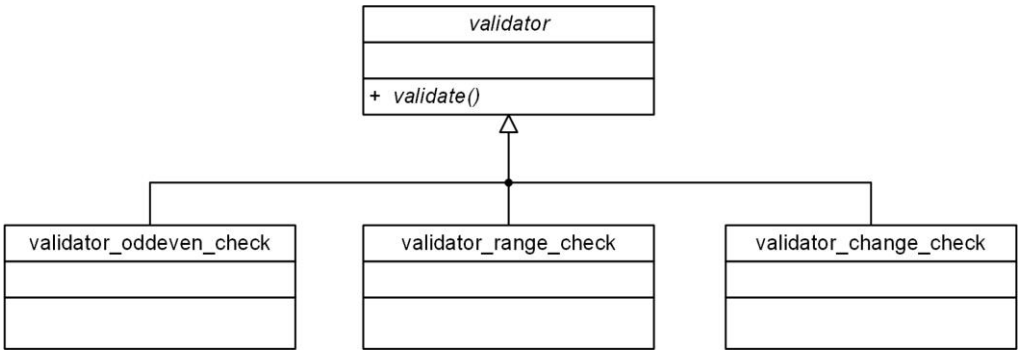


图 1.7　校验器类

由于校验器类仅包含一个函数指针，因此其只需要在头文件中定义出类即可，详见程序清单 1.41。

程序清单 1.41　校验器类定义（validator.h）

```
1   #ifndef __VALIDATOR_H
2   #define __VALIDATOR_H
3
4   struct validator {
5       int    (*validate) (struct validator *p_this, int val);
6   };
7
8   static inline
9   int validator_init(struct validator *p_validator,
10                      int                    (*validate) (struct validator *, int ))
11  {
12      p_validator->validate = validate;
13      return 0;
14  }
15
16  static inline
17  int validator_validate (struct validator *p_validator, int val)      /* 校验函数                          */
19  {
20      if (!p_validator->validate) {                                    /* 校验函数为空，视为无需校验        */
21          return 0;
22      }
23      return p_validator->validate(p_validator, val);
24  }
25
26  #endif
```

程序中，初始化函数负责为 validate 赋值，validator_validate 函数是校验器对外提供的校验函数，在其实现中仅调用了 validate 函数指针指向的函数。由于这些函数都比较简单，因而直接使用了内联函数的形式进行了定义。

接下来，仍然以范围校验为例，实现一个范围校验器，范例程序详见程序清单 1.42 和

程序清单 1.43。

程序清单 1.42　范围校验器 H 文件内容（validator_range_check.h）

```
#ifndef __VALIDATOR_RANGE_CHECK_H
#define __VALIDATOR_RANGE_CHECK_H

#include "validator.h"

struct validator_range_check {
        struct validator        super;
        int                     min;
        int                     max;
};

struct validator* validator_range_check_init (struct validator_range_check *p_validator, int min, int max);

#endif
```

程序清单 1.43　范围校验器 C 文件内容（validator_range_check.c）

```
#include "validator_range_check.h"

static int __validate (struct validator *p_this, int val)
{
    struct validator_range_check *p_stack = (struct validator_range_check *)p_this;
    if ((val >= p_stack->min) && (val <= p_stack->max)) {
        return 0;                                                    /* 检查通过 */
    }
    return -1;
}

struct validator* validator_range_check_init (struct validator_range_check *p_validator, int min, int max)
{
    validator_init(&p_validator->super, __validate);
    p_validator->min = min;
    p_validator->max = max;
    return &p_validator->super;
}
```

由于 validator_range_check 类仅用于实现 validator 抽象类中定义的抽象方法，其初始化函数可以直接对外返回一个标准的校验器（其中的抽象方法已实现）。按照同样的方法，可以实现 validator_oddeven_check 类和 validator_change_check 类。

将检查功能从“带检查功能的栈”中分离出来之后，“带检查功能的栈”中就无需再维护检查功能对应的抽象方法。其可以通过依赖的方式使用检查功能，即依赖一个校验器。在类图中，依赖关系可以使用一个虚线箭头表示，箭头指向被依赖的类，示意图详见图 1.8。

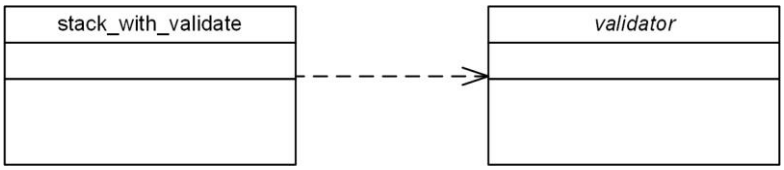


图 1.8　带检查功能的栈依赖于校验器

从某种意义上讲，它们之间的关系也可以视为关联或聚合关系。所谓关联，就是两个类之间有一定的联系，比如，公司与员工，父母与孩子，老师与学生等，显然，“带检查功能的栈”与校验器是存在联系的。聚合与关联类似，但更侧重整体与部分的关系，比如，汽车有引擎、轮子，而引擎和轮子是可以单独存在的。这里可能更接近聚合，因为从名字上看，“带检查功能的栈”就包含了“检查”语义。但无论怎样，都不影响 C 程序的设计。因为在 C 程序设计中，它们都通过一个指向另一个对象的指针来体现。“带检查功能的栈”类定义如下：

```
struct stack_with_validate {
    struct stack          super;                    /* 基类（超类）     */
    struct validator    *p_validator;               /* 依赖的校验器      */
};
```

由此可见，其核心变化仅仅是由一个 validate 函数指针变更为 p_validator 指针，虽然变化很小，却是两种截然不同的设计理念。之前的方式是定义了一个抽象方法，而现在的方式是依赖于一个校验器对象。

基于此，更新“带检查功能的栈”类的实现，详见程序清单 1.44 和程序清单 1.45。

程序清单 1.44　带检查功能的栈 H 文件更新（stack_with_validate.h）

```
1   #ifndef __STACK_WITH_VALIDATE_H
2   #define __STACK_WITH_VALIDATE_H
3
4   #include "stack.h"                                  /* 包含基类头文件 */
5   #include "validator.h"
6
7   struct stack_with_validate {
8       struct stack          super;                    /* 基类（超类）     */
9       struct validator *p_validator;
10  };
11
12  int stack_with_validate_init (struct stack_with_validate *p_stack,
13                                int                        *p_buf,
14                                int                         size,
15                                struct validator           *p_validator);
16
17  int stack_with_validate_push (struct stack_with_validate *p_stack, int val);
18  int stack_with_validate_pop (struct stack_with_validate *p_stack, int *p_val);
19
20  #endif
```

程序清单 1.45　带检查功能的栈 C 文件更新（stack_with_validate.c）

```
1   #include "stack_with_validate.h"
2   #include "stdio.h"
3
4   int stack_with_validate_init (struct stack_with_validate    *p_stack,
5                                 int                           *p_buf,
6                                 int                             size,
7                                 struct validator               *p_validator)
8   {
9       stack_init(&p_stack->super, p_buf, size);
10      p_stack->p_validator = p_validator;
11      return 0;
12  }
13
14  int stack_with_validate_push (struct stack_with_validate *p_stack, int val)
15  {
16      if ((p_stack->p_validator == NULL) || (validator_validate(p_stack->p_validator, val) == 0)) {
17          return stack_push(&p_stack->super, val);
18      }
19      return -1;
20  }
21
22  int stack_with_validate_pop (struct stack_with_validate *p_stack, int *p_val)
23  {
24      return stack_pop(&p_stack->super, p_val);
25  }
```

由于“带检查功能的栈”的应用接口（push 和 pop）并没有发生任何改变，因此，如程序清单 1.40 所示的应用程序可以被复用，主程序范例程序详见程序清单 1.46。

程序清单 1.46　主程序范例程序——测试更新后的带检查功能的栈

```
1   #include "stack_with_validate.h"
2   #include "validator_range_check.h"
3   #include "stdio.h"
4
5   int main ()
6   {
7       int buf[20];
8
9       struct stack_with_validate      stack;
10      struct validator_range_check validator_range_check;
11
12      /* 获取范围检查校验器 */
13      struct validator *p_validator = validator_range_check_init(&validator_range_check, 1, 9);
14
```

```
15          stack_with_validate_init(&stack, buf, 20, p_validator);
16
17          stack_validate_application(&stack);
18
19          return 0;
20      }
```

4. 定义抽象栈

定义校验器类后，整个系统实现了两种栈：普通栈（详见程序清单 1.11 和程序清单 1.12）和“带检查功能的栈”（程序清单 1.44 和程序清单 1.45）。我们知道，无论什么栈，对于用户来讲，无非是实现两个核心的逻辑：入栈和出栈。

两种栈都提供了入栈和出栈方法。普通栈提供的方法为：

```
int stack_push (struct stack *p_stack, int val);                    /* 入栈   */
int stack_pop (struct stack *p_stack, int *p_val);                  /* 出栈   */
```

“带检查功能的栈”提供的方法为：

```
int stack_with_validate_push (struct stack_with_validate *p_stack, int val);    /* 入栈   */
int stack_with_validate_pop (struct stack_with_validate *p_stack, int *p_val);   /* 出栈   */
```

对用户来说，其需要的是入栈和出栈操作，而两种类提供的方法名并不相同，因而在使用不同种类的栈时，调用的函数需要发生改变。

通过上面对多态的讲解可以发现，若基于多态的思想，将入栈和出栈定义为抽象方法（函数指针），则可以达到这样的效果：无论使用何种栈，都可以使用相同的方法来实现入栈和出栈。

基于此，定义抽象栈，示例代码详见程序清单 1.47。

程序清单 1.47　抽象栈类定义（stack.h）

```
1   #ifndef __STACK_H
2   #define __STACK_H
3
4   struct stack {
5           int (*push) (struct stack *p_stack, int val);
6           int (*pop) (struct stack *p_stack, int *p_val);
7   };
8
9   static inline
10  int stack_init (struct stack *p_stack,
11                          int             (*push) (struct stack *, int ),
12                          int             (*pop) (struct stack *, int *))
13  {
14          p_stack->push = push;
15          p_stack->pop    = pop;
16  }
17
18  static inline
19  int stack_push (struct stack *p_stack, int val)
```

```
{
        return p_stack->push(p_stack, val);
}

static inline
int stack_pop (struct stack *p_stack, int *p_val)
{
        return p_stack->pop(p_stack, p_val);
}

#endif
```

基于抽象栈的定义，可以使用抽象栈提供的接口实现一个通用的应用程序，该应用程序与任何底层细节无关，任何栈都可以使用该应用程序进行测试，范例程序详见程序清单 1.48。

程序清单 1.48　基于抽象栈实现的应用程序

```
#include "stack.h"
#include "stdio.h"

int stack_application (struct stack *p_stack)
{
        int i;
        int val;
        int test_data[5] = {2, 4, 5, 3, 10};

        for (i = 0; i < 5; i++) {
                if (stack_push(p_stack, test_data[i]) != 0) {
                        printf("The data %d push failed!\n", test_data[i]);
                }
        }

        printf("The pop data: ");
        while (1) {                                                    /* 弹出所有数据 */
                if (stack_pop(p_stack, &val) == 0) {
                        printf("%d    ", val);
                } else {
                        break;
                }
        }
        return 0;
}
```

可以看到，在具体实现栈之前，就可以开始编写应用程序了！接下来，先尝试实现普通栈，范例程序详见程序清单 1.49 和程序清单 1.50。

程序清单 1.49　普通栈 H 文件内容（stack_normal.h）

```
1   #ifndef __STACK_NORMAL_H
2   #define __STACK_NORMAL_H
3
4   #include "stack.h"
5
6   struct stack_normal {
7       struct stack        super;
8       int                 top;            /* 栈顶             */
9       int                 *p_buf;         /* 栈缓存           */
10      unsigned int        size;           /* 栈缓存的大小     */
11  };
12
13  struct stack * stack_normal_init (struct stack_normal *p_stack, int *p_buf, int size);
14
15  #endif
```

程序清单 1.50　普通栈 C 文件内容（stack_normalc）

```
1   #include "stack_normal.h"
2
3   static int __push (struct stack *p_this, int val)
4   {
5       struct stack_normal *p_stack = (struct stack_normal *)p_this;
6       if (p_stack->top != p_stack->size) {
7           p_stack->p_buf[p_stack->top++] = val;
8           return 0;
9       }
10      return -1;
11  }
12
13  static int __pop (struct stack *p_this, int *p_val)
14  {
15      struct stack_normal *p_stack = (struct stack_normal *)p_this;
16      if (p_stack->top != 0) {
17          *p_val = p_stack->p_buf[--p_stack->top];
18          return 0;
19      }
20      return -1;
21  }
22
23  struct stack * stack_normal_init (struct stack_normal *p_stack, int *p_buf, int size)
24  {
25      p_stack->top  = 0;
26      p_stack->size = size;
```

```
27          p_stack->p_buf = p_buf;
28          stack_init(&p_stack->super, __push, __pop);
29          return &p_stack->super;
30      }
```

基于普通类的实现，可以尝试使用如程序清单 1.48 所示的应用程序来测试普通栈类，主程序范例详见程序清单 1.51。

程序清单 1.51　主程序范例程序——测试普通栈

```
1       #include "stack_normal.h"
2       int main ()
3       {
4           int                         buf[20];
5           struct stack_normal stack;
6           struct stack *p_stack = stack_normal_init (&stack, buf, 20);
7           stack_application(p_stack);
8           return 0;
9       }
```

由于普通栈不会对压入数据进行检查，且仅压入了 5 个数据，缓存空间（大小 20）足够，因此，所有测试数据（2、4、5、3、10）都可以成功压入栈中，出栈数据与入栈顺序相反，即为：10、3、5、4、2。

“带检查功能的栈”是在普通栈的基础上，增加了检查功能，实现范例程序详见程序清单 1.52 和程序清单 1.53。

程序清单 1.52　带检查功能的栈 H 文件更新（stack_with_validate.h）

```
1       #ifndef __STACK_WITH_VALIDATE_H
2       #define __STACK_WITH_VALIDATE_H
3
4       #include "stack.h"                                          /* 包含基类头文件        */
5       #include "validator.h"
6
7       struct stack_with_validate {
8           struct stack            super;                      /* 基类（超类）          */
9           struct stack          *p_normal_stack;              /* 依赖于普通栈的实现    */
10          struct validator      *p_validator;
11      };
12
13      struct stack * stack_with_validate_init (struct stack_with_validate         *p_stack,
14                                               struct stack                       *p_normal_stack,
15                                               struct validator                   *p_validator);
16
17      #endif
```

程序清单 1.53　带检查功能的栈 C 文件更新（stack_with_validate.c）

```
1       #include "stack_with_validate.h"
```

```
2   #include "stdio.h"
3
4   static int __push (struct stack *p_this, int val)
5   {
6       struct stack_with_validate *p_stack = (struct stack_with_validate *)p_this;
7       if ((p_stack->p_validator == NULL) || (validator_validate(p_stack->p_validator, val) == 0)) {
8           return stack_push(p_stack->p_normal_stack, val);
9       }
10      return -1;
11  }
12
13  static int __pop (struct stack *p_this, int *p_val)
14  {
15      struct stack_with_validate *p_stack = (struct stack_with_validate *)p_this;
16      return stack_pop(p_stack->p_normal_stack, p_val);
17  }
18
19  struct stack * stack_with_validate_init (struct stack_with_validate *p_stack,
20                                           struct stack                *p_normal_stack,
21                                           struct validator            *p_validator)
22  {
23      stack_init(&p_stack->super, __push, __pop);
24      p_stack->p_validator    = p_validator;
25      p_stack->p_normal_stack = p_normal_stack;
26      return &p_stack->super;
27  }
```

基于“带检查功能的栈”的实现，同样可以使用如程序清单 1.48 所示的应用程序来进行测试，主程序范例详见程序清单 1.54。

程序清单 1.54　主程序范例程序——测试带检查功能的栈

```
1   #include "stack_normal.h"
2   #include "stack_with_validate.h"
3   #include "validator_range_check.h"
4
5   int main ()
6   {
7       int                             buf[20];
8       struct stack_normal             stack;
9       struct stack_with_validate      stack_with_validate;
10      struct validator_range_check    validator_range_check;
11
12      struct stack        *p_stack_normal = stack_normal_init (&stack, buf, 20);
13      struct validator    *p_validator    = validator_range_check_init(&validator_range_check, 1, 9);
14      struct stack        *p_stack        = stack_with_validate_init(&stack_with_validate,
```

```
15                                                          p_stack_normal,
16                                                          p_validator);
17
18          stack_application(p_stack);
19          return 0;
20      }
```

运行程序可以发现，数据 10 入栈会失败（其不再范围 1～9），因此，仅 2、4、5、3 可以顺利压入栈中，出栈数据与入栈顺序相反，即为：3、5、4、2。

由此可见，无论底层的各种栈如何实现，对于上层应用来讲，其可以使用同一套接口操作各种各样不同的栈。

这里仅作为一种**多态**的范例，在实际应用中，由于目前只存在两种栈，它们的含义区分也很明确，并不是一定需要使用多态的手段定义抽象栈。

前面介绍了多种多态示例，实际上，它们的核心解决方案都是相同的，即：定义抽象方法（函数指针），使上层应用可以使用同一套接口访问不同的对象。从类的角度看，每个类中操作的规约都是相同的，而这些类可以用不同的方式实现这些同名的操作，从而使得拥有相同接口的对象可以在运行时相互替换。

在实际应用中，**多态**是实现应用程序跨平台（AWorks 的特点之一）的关键。跨平台应用程序的基本目的是：同样的应用程序，可以在多个硬件平台上运行（比如：EasyARM-RT1052、IoT-A3352LI EPC-287C-L……），更换硬件时应用程序无需作任何改动。当应用程序需要使用某一硬件功能时（比如 I^2C 数据传输），其就定义相应的抽象方法，在不同的硬件平台中，都需要根据当前平台特性实现相应的抽象方法（“I^2C 数据传输功能”），但实现方法可以各不相同，毕竟不同芯片的 I^2C 操作方法通常都存在一些差异。如此一来，当应用程序需要通过 I^2C 传输数据时，仅需调用统一的接口即可。

多态非常强大，可以为应用程序定义统一的接口，而底层具有差异性的对象，就变成了应用程序的“插件”，可以任意插拔替换，均不会影响应用程序的编程。就好比上文提到的 I/O 设备驱动，系统定义了 I/O 设备驱动的接口，底层各式各样不同的 I/O 硬件设备，接口的统一使得它们都可以作为“插件”插入系统中使用，它们之间相互替换，对特定的应用程序不会造成任何影响（例如使用 printf()输出字符串信息）。

在嵌入式系统中，通常都会操作到一些具体的硬件，在传统的编程中，很容易编写出与具体硬件相关的应用程序。而基于多态的思想，可以很容易实现“**与硬件无关**”的应用程序。

基于多态的思想，还可以衍生出两个概念：抽象接口与依赖反转，它们的核心都是多态。在下一章还会对它们作进一步介绍，使读者可以更加深入的理解多态。

第2章 抽象接口

本章导读

基于多态可以实现“与硬件无关”的应用程序。在 C 编程中，多态的核心解决方法是充分利用“函数指针”，抽象接口就是只包含函数指针的类，它们非常抽象，不包含任何具体的实现，仅定义了函数的调用规则。上一章中定义的“抽象栈类”，本质上就属于抽象接口。本章将站在抽象接口的角度，进一步深入介绍，指导读者在实际的嵌入式编程中，利用抽象接口编写与“与硬件无关”的应用程序。

2.1 基本概念

在面向对象编程中，**抽象接口**又通常简称为**接口**（interface），接口本身就是抽象的。这里说所的“接口”，是一种在面向对象编程中的概念。并不是在 C 开发中，头文件中引出的 API。

“接口”是一种抽象的概念，其仅从宏观的角度描述了某一功能或动作，不涉及任何底层实现细节。例如，对于“数据存储”功能，可以定义相应的数据存储接口，但具体数据如何存储，接口并不关心，底层可以根据实际情况完成数据的存储，例如：使用 EEPROM、SPI FLASH、文件系统、数据库等等。

从形式上看，接口是一组抽象方法（只有声明，没有实现）的集合，接口只负责定义和维护这些抽象方法，并不负责实现方法。抽象方法的实现交由具体的实现类完成。在 UML 类图中，“接口”可以使用相应的接口类表示。为了更加明确的表示接口，需要在接口名上增加一个“<<interface>>”标记，同时，在接口的命名前加上字母“I”。接口与实现类之间使用“三角虚线箭头”连接，箭头指向接口。如图 2.1 所示的类图表示了“接口”和实现类之间的关系。图中，ITime 为接口，其定义了时间管理对应的两个抽象方法：设置时间（time_set()）与获取时间（time_get()）。 PCF85063 是实现类，其代表了一个型号为 PCF85063 的 RTC（Real-Time Clock，实时时钟）芯片实现了设置时间与获取时间功能。

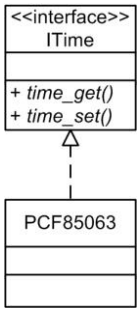


图 2.1 接口与实现类之间的关系

2.2 “接口”示例

2.2.1 在 C 语言中定义“接口”

在 C 语言中，可以使用函数指针表示抽象方法。定义函数指针时，其类型决定了实现函数的原型，即表示了方法的声明，且定义函数指针时并需要不关心其“具体实现”，即函数指针的指向。基于此，接口可以使用包含函数指针的结构体类型表示，例如，对于时间管理，可以定义相应的接口，接口中包含两个抽象方法：设置时间与获取时间，定义范例详见程序清单 2.1。

程序清单 2.1　itime 接口定义（itime.h）

```
1    struct itime {
2        int (*pfn_time_get) (struct itime *p_this, struct tm *p_tm);
3        int (*pfn_time_set) (struct itime *p_this, struct tm *p_tm);
4    };
```

其定义中包含的两个函数指针分别对应了两个抽象方法，函数指针的类型约定了对应实现函数的原型，它们有两个参数：p_this 和 p_tm。p_this 用于指向自身；p_tm 用于指向时间信息（struct tm 是标准 C 定义的时间结构体，包含年、月、日、时、分、秒等信息，详见程序清单 2.2）。int 类型的返回值可以用来返回函数执行的结果，例如，0 表示设置（或获取）时间成功，-1 表示设置（或获取）时间失败。

程序清单 2.2　标准时间类型定义（time.h）

```
1    struct tm {
2        int    tm_sec;                    // 秒，0 ~59
3        int    tm_min;                    // 分，0 ~ 59
4        int    tm_hour;                   // 小时，0 ~ 23
5        int    tm_mday;                   // 日期，1 ~ 31
6        int    tm_mon;                    // 月份，0 ~ 11
7        int    tm_year;                   // 年
8        int    tm_wday;                   // 星期
9        int    tm_yday;                   // 天数
10       int    tm_isdst;                  // 夏令时（一般不使用，值为 0 或-1）
11   };
```

也就是说，具体的时间获取函数和时间设置函数，它们也必须具有相同的类型，2 个参数，一个 int 类型返回值。

为了方便上层应用使用接口，可以提供相应的 API，详见程序清单 2.3。

程序清单 2.3　“接口”对外的 API（itime.h）

```
1    static inline
2    int itime_time_get    (struct itime *p_this, struct tm *p_tm)
3    {
4        return p_this->pfn_time_get(p_this, p_tm);
5    }
6
7    static inline
8    int itime_time_set    (struct itime *p_this, struct tm *p_tm)
9    {
10       return p_this->pfn_time_set(p_this, p_tm);
11   }
```

由于 API 的实现代码非常简单，因而直接以内联函数的形式存于.h 文件中。

2.2.2 基于接口编写应用程序

基于接口提供的 API，当上层应用需要设置或获取时间时，仅需调用相应的 API 即可。

例如，对于一个电子表应用，其需要提供设置时间和显示时间的功能，应用程序的示意代码详见程序清单 2.4。

程序清单 2.4　应用程序示意代码

```
1   void app_elec_watch (struct itime *p_time)
2   {
3       struct tm now_tm;
4
5       if  (用户修改了当前时间) {
6           struct tm set_tm = ...                                  // 用户设置的时间
7           itim_time_set (p_time, &set_tm);                        // 设置时间
8       }
9
10      itime_time_get (p_watch->p_rtc, &now_tm);                   // 获取当前时间
11
12      printf("Now time is : %04d-%02d-%02d %02d:%02d:%02d \r\n",
13                  now_tm.tm_year+1900,    now_tm.tm_mon + 1,      now_tm.tm_mday,
14                  now_tm.tm_hour,         now_tm.tm_min,          now_tm.tm_sec);
15
16      // ...其它处理，如在 LCD 上显示时间等
17  }
```

程序仅作示意之用，实际应用的功能可能会复杂很多，需要由多个函数协作完成。这里主要是为了展示应用程序的一个重要特点：应用程序完全基于接口提供的 API 实现，由于接口与底层硬件无关，因而应用程序也完全与底层硬件无关，换句话说，应用程序不会与具体的硬件绑定，可以跨平台复用。

这里还可以发现另外一个现象：接口还没实现（实现类还未定义），就可以开始编写应用程序了！由此可见，接口类的存在，完全隔离了应用程序和具体的实现类，使它们可以相互独立的开发。在一些大型项目中，就可以很容易实现分工合作，各司其职，应用程序和底层驱动的负责团队，可以同步展开工作。但这有一个前提，就是保证接口的稳定性，否则，一旦接口发生变化，对底层实现类和应用程序都会产生影响。

2.2.3 实现类

前面提到，定义接口的重点在于抽象方法的定义，并不需要关心具体的实现细节。接口中定义的抽象方法应由具体的实现类实现，在应用程序使用某一功能前，必须确保相应的抽象方法已被实现。对应于 C 程序开发，即抽象方法对应的函数指针必须赋值（具有实际的指向）后才能使用。

例如，某一型号为 PCF85063 的 RTC 芯片，可以提供时间的设置和获取功能，则其可以实现 struct itime 接口。据此，可以完成 PCF85063 实现类的定义，范例程序详见程序清单 2.5 和程序清单 2.6。

程序清单 2.5　pcf85063.h

```
1   #ifndef __PCF85063_H
2   #define __PCF85063_H
3
```

```
4   #include "itime.h"
5
6   struct pcf85063 {
7       struct itime    itime;
8   };
9
10  #endif          /* __PCF85063_H */
```

程序清单 2.6　pcf85063.c

```
1   #include "pcf85063.h"
2
3   static int __pcf85063_time_get (struct itime *p_this, struct tm *p_tm)
4   {
5       // 首先，从 PCF85063 中获取出年、月、日、时、分、秒等信息
6
7       if (获取成功) {
8           p_tm->tm_year    =    // 从 PCF85063 中获取出的 年 信息
9           p_tm->tm_mon     =    // 从 PCF85063 中获取出的 月 信息
10          p_tm->tm_mday    =    // 从 PCF85063 中获取出的 日 信息
11          p_tm->tm_hour    =    // 从 PCF85063 中获取出的 时 信息
12          p_tm->tm_min     =    // 从 PCF85063 中获取出的 分 信息
13          p_tm->tm_sec     =    // 从 PCF85063 中获取出的 秒 信息
14
15          return 0;                // 获取时间成功
16      }
17      return -1;              // 获取时间失败
18  }
19
20  static int __pcf85063_time_set (struct itime *p_this, struct tm *p_tm)
21  {
22      // 首先，将 p_tm 中的年、月、日、时、分、秒等信息设置到 PCF85063 中
23
24      if (设置成功) {
25          return 0;
26      }
27      return -1;
28  }
29
30  struct itime * pcf85063_init (struct pcf85063 *p_dev);
31  {
32      //... PCF85063 硬件初始化
33
34      p_dev->itime.pfn_time_set = __pcf85063_time_set;
35      p_dev->itime.pfn_time_get = __pcf85063_time_get;
```

```
36          return &p_dev->itime;
37  }
```

程序中，实现了接口类中定义的两个抽象方法，实现函数分别为：__pcf85063_time_set()和__pcf85063_time_get()。并对外提供了一个初始化函数：pcf85063_init()，在初始化函数中，完成了函数指针的赋值，并直接返回了一个 itime 接口类对象（已实现其中的抽象方法）。

2.2.4 主程序设计

为了启动“电子表应用程序”，应先完成 PCF85063 的初始化，以便获取到“接口实现”，进而传递给应用程序，使应用程序可以正常工作，主程序范例详见程序清单 2.7。

程序清单 2.7　主程序范例程序（使用 PCF85063）

```
1   #include "pcf85063.h"
2   struct pcf85063 pcf85063_dev;
3   int main()
4   {
5           struct itime *p_itime = pcf85063_init(&pcf85063_dev);
6           app_elec_watch (p_itime);                                   // 启动“电子表”应用程序
7           while (1) {
8           }
9   }
```

程序中，首先通过初始化函数获取到一个“接口实现”，然后将“接口实现”传递给应用程序，应用程序即可直接基于接口类提供的通用 API 编程。对应的类图详见图 2.2。

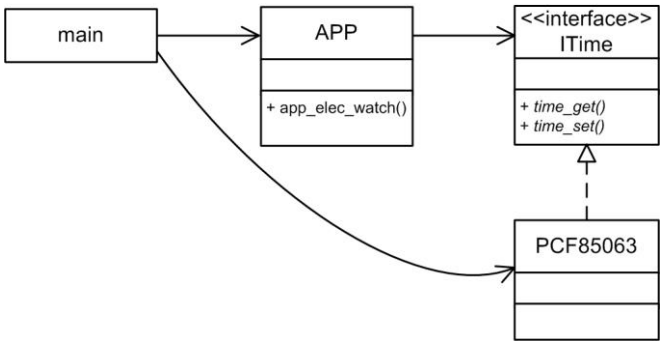


图 2.2　应用程序、接口与实现类之间的关系

图中，由于 APP 调用了接口类提供的 API，因此，其依赖于接口类。而 PCF85063 是接口类的一种具体实现。根据类图也可以明显的发现，APP 与底层硬件不存在任何关系。主程序（main()）是一个比较特殊的部分，在这里，并不将主程序视为应用程序的一部分，可以将其特殊看待，其一般与系统中的多个组件都存在一定的关联，负责对象的创建（如定义 PCF85063 对象）等操作，为应用程序的运行提供必要的运行环境。

上述程序是 C 语言示意代码，展示了接口实现的原理。使用 C 语言实现面向对象编程时，需要一些指针上的操作技巧。在一些支持“接口”概念的面向对象编程语言中（如 Java），其可以直接定义接口类，并使用特定语法标识某个具体类实现了相应的接口，如此一来，后续在使用接口类的各个 API 时，可以直接将具体类作为操作的对象，编译器根据语法，自动从具体类中获取抽象方法的实现，保证正确性。

2.3 更换底层硬件

前面基于PCF85063启动了应用程序。若需要更换底层硬件，将PCF85063更换为DS1302（DS1302也是一种RTC芯片，其也可以提供设置时间和获取时间功能，即实现ITime接口），则再增加一个DS1302实现类即可，范例程序详见程序清单2.8和程序清单2.9。

程序清单 2.8　ds1302.h

```
#ifndef __DS1302_H
#define __DS1302_H

#include "itime.h"

struct ds1302 {
    struct itime    itime;
};

#endif      /* __DS1302_H */
```

程序清单 2.9　ds1302.c

```
#include "ds1302.h"

static int __ds1302_time_get (struct itime *p_this, struct tm *p_tm)
{
    // 首先，从 DS1302 中获取出年、月、日、时、分、秒等信息

    if (获取成功) {
        p_tm->tm_year   =       // 从 DS1302 中获取出的 年 信息
        p_tm->tm_mon    =       // 从 DS1302 中获取出的 月 信息
        p_tm->tm_mday   =       // 从 DS1302 中获取出的 日 信息
        p_tm->tm_hour   =       // 从 DS1302 中获取出的 时 信息
        p_tm->tm_min    =       // 从 DS1302 中获取出的 分 信息
        p_tm->tm_sec    =       // 从 DS1302 中获取出的 秒 信息
        return 0;               // 获取时间成功
    }
    return -1;                  // 获取时间失败
}

static int __ds1302_time_set (struct itime *p_this, struct tm *p_tm)
{
    // 首先，将 p_tm 中的年、月、日、时、分、秒等信息设置到 DS1302 中

    if (设置成功) {
        return 0;
    }
    return -1;
```

```
27  }
28
29  struct itime * ds1302_init (struct ds1302 *p_dev);
30  {
31          //...     DS1302 硬件初始化
32
33          p_dev->itime.pfn_time_set = __ds1302_time_set;
34          p_dev->itime.pfn_time_get = __ds1302_time_get;
35          return &p_dev->itime;
36  }
```

基于 DS1302 实现类，若主程序需要使用 DS1302 启动应用程序，则仅需将程序清单 2.7 中的 PCF85063 关键字修改为 DS1302，范例程序详见程序清单 2.10。

程序清单 2.10　主程序范例程序（使用 DS1302）

```
1   #include "ds1302.h"
2   struct ds1302 ds1302_dev;
3   int main()
4   {
5           struct itime *p_itime = ds1302_init(&ds1302_dev);
6           app_elec_watch (p_itime);                                   // 启动“电子表”应用程序
7           while (1) {
8           }
9   }
```

由此可见，PCF85063 和 DS1302 均可实现 ITime 定义的抽象方法，都可以作为 ITime 的实现类。由于应用程序（app_elec_watch）直接基于接口类提供的通用 API 实现，因此，无论底层使用何种硬件（PCF85063 或 DS1302），都不会影响到应用程序。在实际应用中，除 PCF85063 和 DS1302 外，还有很多其它型号的 RTC 芯片，比如 RX8025T、PCF2127、PCF85263 等等，它们都可以作为 ITime 的实现类，对应的类图详见图 2.3。

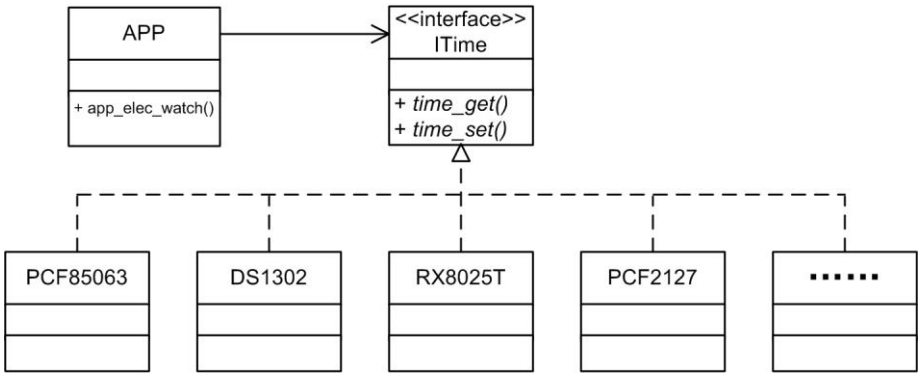


图 2.3　ITime 接口类与多种实现类之间的关系

由于所有具体的实现类都要基于接口类中定义的抽象方法来实现（功能、函数原型等由抽象方法规定），因此，通常情况下，接口类会被很多具体的实现类所依赖，若修改了接口类，则改动会影响所有的实现类，影响范围很大。基于此，接口类应该是非常稳定的，不应该轻易变化。实际上，这对接口类的定义提出了很高的要求，接口类不能随便定义，应考虑到可能的变化，合理、正确的定义各个抽象方法。

第3章 依赖反转

本章导读

依赖反转，顾名思义，就是将依赖颠倒过来，在传统的嵌入式编程中，很容易编写出与具体硬件绑定的应用程序，基于抽象接口，可以反转这个现象，使应用程序不再与具体硬件相关（抽象接口可以看作依赖反转的一种典型应用）。本章将从嵌入式系统中遇到的问题入手，逐步介绍依赖反转的实现方法，最后还特别介绍了依赖反转在 AWorks 中的应用，使读者可以对 AWorks 作进一步了解。

3.1 问题引入

在嵌入式系统开发过程中，相信不少程序员或多或少都接触过“main()大循环函数”：整个应用程序控制在一个 main()函数内，其它功能通过各种各样的函数实现，在 main()主循环中依次调用各个函数，形如程序清单 3.1。

程序清单 3.1 mian()函数典型形式

```
1   void main( )
2   {
3       while (1) {
4           func_H1();
5           func_H2();
6           func_H3();
7           // ...
8       }
9   }
```

当系统比较复杂时，main()函数直接调用的往往是一些高层函数，这些高层函数还会调用其它一些中层函数，例如，func_H1()的一种实现形式详见程序清单 3.2。

程序清单 3.2 一些高层函数的实现形式

```
1   void func_H1( )
2   {
3       func_M1();
4       func_M2();
5   }
```

当然，这些中层函数可能还会调用一系列低层函数，例如，func_M1 的一种实现形式详见程序清单 3.3。

程序清单 3.3 一些中层函数的实现形式

```
1   void func_M1( )
2   {
3       func_L1();
4       func_L2();
5   }
```

如此一来，经过层层调用，整个调用关系最终将成为一个以 main()函数为根的“树结构”，详见图 3.1。

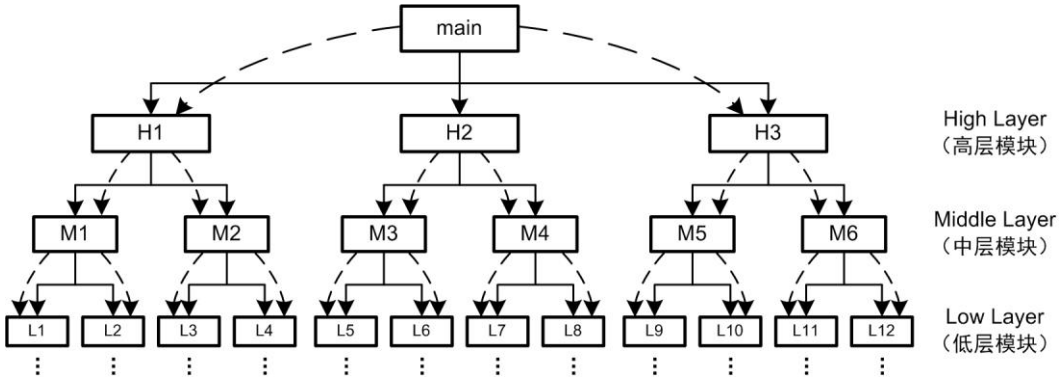


图 3.1　main()依次调用各模块函数形成的调用关系和依赖关系

图中，虚线箭头表示函数调用关系（控制流，可以理解为程序的运行走向），实线箭头表示模块依赖关系（从程序设计角度，模块代码之间的依赖关系），由此可见，模块与模块之间的依赖关系，与调用关系保持完全一致。例如，func_H1()（H1 模块）调用了 func_M1（M1 模块），则 H1 模块将依赖于 M1 模块，依赖体现在：H1 模块必须知道 M1 模块提供了哪些接口以及接口的原型，以便调用。在 C 程序中，H1 模块需要使用“#include”语句包含 M1 模块提供的头文件，例如，#include "m1.h"。换句话说，H1 模块必须知道 M1 模块的存在，以及存在的具体形式。同理，func_M1 调用了 func_L1（假定其为 L1 模块中的一个函数），则模块 M1 依赖于模块 L1。

这就是面向过程编程的典型结构，在这个结构中，由于是高层模块直接调用低层模块，因此，最终会形成一种高层模块依赖于低层模块的结构。

为便于理解，不妨以一个具体的应用进行说明。例如，要做一个电子表，先不考虑时间显示部分，仅考虑如何获取和设置时间。显然，为了获取到精确的时间，需要由一个 RTC（Real-Time Clock）芯片来提供时间信息，包括年、月、日、时、分、秒等信息。市面上，RTC 芯片有很多，常见的有：PCF85063、RX8025T、DS1302 等。

若选择使用 PCF85063 芯片，则在使用前，需要编写芯片的驱动，并提供相应的获取和设置时间 API，驱动示意代码详见程序清单 3.4 和程序清单 3.5。

程序清单 3.4　PCF85063 模块 H 文件（pcf85063.h）

```
1    int pcf85063_time_get(struct tm *p_tm);          // 获取时间
2    int pcf85063_time_set(struct tm *p_tm);          // 设置时间
```

程序清单 3.5　PCF85063 模块 C 文件（pcf85063.c）

```
1    int pcf85063_time_get (struct tm *p_tm)
2    {
3        // 首先，从 PCF85063 中获取出年、月、日、时、分、秒等信息
4        if (获取成功) {
5            p_tm->tm_year    =    // 从 DS1302 中获取出的 年 信息
6            p_tm->tm_mon     =    // 从 DS1302 中获取出的 月 信息
7            p_tm->tm_mday    =    // 从 DS1302 中获取出的 日 信息
8            p_tm->tm_hour    =    // 从 DS1302 中获取出的 时 信息
9            p_tm->tm_min     =    // 从 DS1302 中获取出的 分 信息
```

```
10              p_tm->tm_sec      =     // 从 DS1302 中获取出的 秒 信息
11              return 0;               // 获取时间成功
12          }
13          return -1;                  // 获取时间失败
14      }
15
16      int pcf85063_time_set (struct tm *p_tm)
17      {
18          // 首先，将 p_tm 中的年、月、日、时、分、秒等信息设置到 PCF85063 中
19
20          if (设置成功) {
21              return 0;
22          }
23          return -1;
24      }
```

此时，应用程序模块（假定名为“APP”）可以直接调用该驱动提供的 API 进行时间的设置和获取，应用程序模块的范例程序详见程序清单 3.6 和程序清单 3.7。

程序清单 3.6　应用程序模块 H 文件（app.h）

```
1   void app_elec_watch();                                          // 运行应用程序，具体参数暂不理会
```

程序清单 3.7　应用程序模块 C 文件（app.c）

```
1   #include "pcf85063.h"
2
3   void app_elec_watch ()
4   {
5       struct tm now_tm;
6
7       if   (用户修改了当前时间) {
8           struct tm set_tm = ...                                  // 用户设置的时间
9           pcf85063_time_set (&set_tm);                            // 设置时间
10      }
11      pcf85063_time_get (&now_tm);                                // 获取当前时间
12      printf("Now time is : %04d-%02d-%02d %02d:%02d:%02d \r\n",
13                          now_tm.tm_year+1900,    now_tm.tm_mon + 1,      now_tm.tm_mday,
14                  now_tm.tm_hour,         now_tm.tm_min,          now_tm.tm_sec);
15      // ...其它处理，如在 LCD 上显示时间等
16  }
```

在主程序（main()函数）中，可以直接调用 app_elec_watch()启动整个应用程序，范例程序详见程序清单 3.8。

程序清单 3.8　主程序范例程序（main.c）

```
1   #include "app.h"
2   void main( )
```

```
3   {
4       while(1) {
5           app_elec_watch();                          // 运行时间 获取&显示 应用
6           // 其它操作，如设置闹钟等
7       }
8   }
```

这里仅为展示面向过程编程中的典型结构，并没有详细实现各个模块中接口。

通过上面的实现过程可以发现，主程序模块（main）调用了应用程序模块（app）中的函数，因而主程序模块依赖于 APP 模块，这种依赖是容易理解的，毕竟主程序需要通过调用 app_time_run()实现整个应用程序，程序清单 3.8 的第一行也明确包含了 app.h 头文件（#include "app.h"），这是一种源码级别的依赖关系。

同理，APP 模块调用了 PCF85063 模块中的 pcf85063_time_get()函数，因而 APP 模块依赖于 PCF85063 模块，这种依赖也是容易理解的，因为 APP 模块需要通过 PCF85063 获取时间信息。显然，PCF85063 是低层模块（具体的硬件），据此可以更充分的理解高层模块依赖于低层模块。

基于此，可以画出主程序、APP 以及 PCF85063 之间的依赖关系，详见图 3.2。

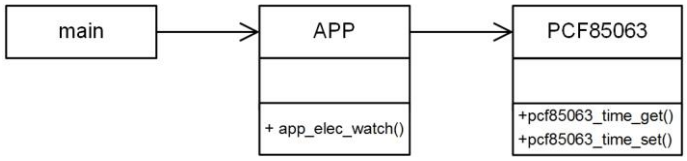


图 3.2　传统设计中的依赖关系图

在面向过程的编程中，这种情形随处可见，高层模块依赖了大量的低层模块。这种结构有一个明显的缺点：修改低层模块时，将影响高层模块，并且，在实际应用中，越低层的模块，实际上越有可能被修改。例如，虽然项目初期决定使用 PCF85063，但后续因为各种原因（价格、供货周期等），需要更换为其它芯片（如 DS1302），那么，应用程序就必须修改为使用 DS1302 模块，其获取时间的接口可能变化为 ds1302_time_get()，如此一来，应用程序模块中获取时间的语句就必须作对应修改，详见程序清单 3.9。

程序清单 3.9　应用程序模块 C 文件（app.c）—— 基于 DS1302 获取时间

```
1   #include "ds1302.h"
2
3   void app_elec_watch ()
4   {
5       struct tm now_tm;
6
7       if  (用户修改了当前时间) {
8           struct tm set_tm = ...                          // 用户设置的时间
9           ds1302_time_set (&set_tm);                      // 设置时间
10      }
11      ds1302_time_get (&now_tm);                          // 获取当前时间
12      printf("Now time is : %04d-%02d-%02d %02d:%02d:%02d \r\n",
13                  now_tm.tm_year+1900,    now_tm.tm_mon + 1,      now_tm.tm_mday,
```

```
14                      now_tm.tm_hour,          now_tm.tm_min,          now_tm.tm_sec);
15          // ...其它处理，如在 LCD 上显示时间等
16      }
```

由此可见，其依赖的模块将由 PCF85063 变更为 DS1302。当然，这种情况下，可能修改 3 行（第 1 行、第 9 行和第 11 行）代码就可以完成整个应用程序的修改，但这仅仅是因为应用程序过于简单而已，实际项目中，调用关系通常会非常复杂，修改的地方可能会非常多，“牵一发而动全身”，可能引起应用程序大量改动，甚至重构。

解决这个问题的方法就是就是“依赖反转”，顾名思义，依赖反转就是将依赖关系颠倒，转变为低层模块依赖于高层模块。由于高层模块不再依赖于低层模块，因此，低层模块的任何变化（如更换芯片等）都不会影响到高层模块。

3.2 基本结构

实际上，在第 2 章中介绍抽象接口时，就提到了将硬件从 PCF85063 更换为 DS1302 时，应用程序无需作任何变化。由此可见，抽象接口可以达到依赖反转的目的：低层模块的变化不会影响高层模块（应用程序）。既然能达到该目的，本质上也就实现了依赖反转，由此可见，抽象接口是实现依赖反转的关键。接下来，对此作一个系统性的介绍。

实现依赖反转的核心是在低层模块智商，增加一个接口层，示意图详见图 3.3。

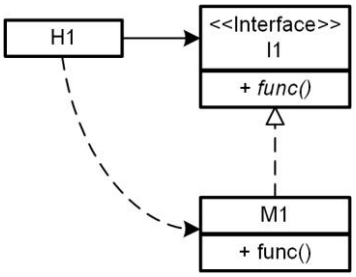


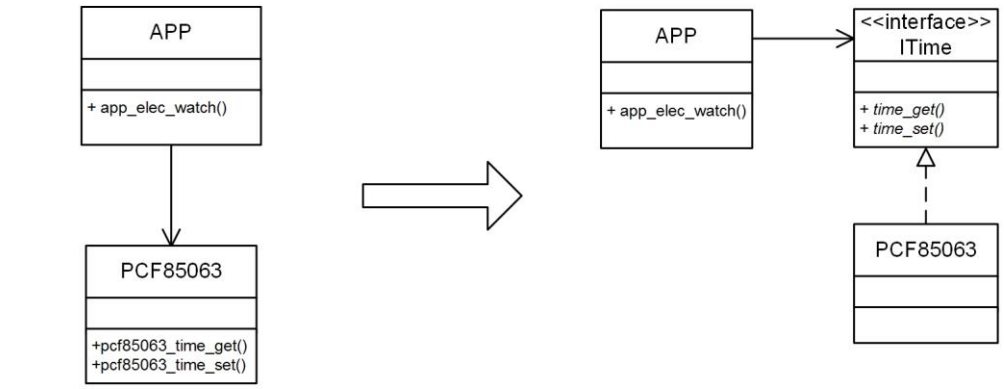
图 3.3　依赖反转示意图

图中新增了一个名为 I1 的抽象接口，其中定义了抽象方法：func()，低层模块作为抽象接口的实现类，实现了其中的 func()方法。此时，对于高层模块 H1，其仅依赖于接口 I1（实线箭头的指向），不再依赖于 M1，换句话说，在 C 程序设计时，H1 模块函数不再需要包含 M1 模块的头文件（如#include "m1.h"）。但从程序运行角度（虚线，控制流），其最终实际上调用的函数依然是 M1 模块中的函数，也就是说，程序运行过程没有发生任何改变（程序的功能不会改变），仅从程序设计角度，依赖关系发生了变化。总之，高层模块 H1 不再依赖于下层模块 M1。

对于下层模块 M1，其同样依赖于 I1（实际上，这里更确切的可以理解为一种继承或实现关系，但本质上，也是一种特殊的依赖关系），此时，可以发现，没有任何模块依赖于下层模块 M1，也就意味着，它的改变将不会影响系统的任何其他部分。

基于这种思想，可以将图 3.1 中所有的依赖关系进行反转。事实上，通过这种机制，无论我们面对怎样的源代码级别的依赖关系，都可以将其反转，软件设计者可以完全控制系统中所有源代码级别的依赖关系，而不再受到系统控制流的限制。不管哪个模块调用或者被调用，都可以随意更改源代码的依赖关系。

在第 2 章中，已经展示了应用程序（APP）、接口和实现类之间的关系，详见图 2.2。而程序清单 3.7 中的应用程序，其直接调用了 PCF85063 提供了接口，对应关系详见图 3.2。由此可见，使用抽象接口实现依赖反转后的关系变化详见图 3.4。



（a）高层模块直接依赖于低层模块　　（b）增加接口层，实现依赖反转

图 3.4　依赖反转变化示意图

图中主要展示了高层模块（APP）和低层模块（PCF85063）之间的关系变化，没有展示出特殊的主程序（main）模块。通过箭头的指向可以发现，依赖关系发生了“反转”。

3.3　依赖反转在 AWorks 中的应用

AWorks 充分利用了“依赖反转机制”，解除了上层模块对低层模块（特别是底层硬件）的依赖，进而实现了“一次编程、终身使用、跨平台”。本节继续以 RTC 为例进行说明，使读者了解 AWorks 对这类问题的典型处理方法。

3.3.1 抽象接口层

在 AWorks 中，定义了抽象的“RTC 服务”（RTC 能为系统提供实时时间功能，可以视为其向系统提供了时间服务，因而这里形象的称之为“RTC 服务”），其存在的核心意义同样是完成抽象方法的定义，其完整定义详见程序清单 3.10。

程序清单 3.10　RTC 抽象服务的定义

```
1  struct awbl_rtc_service {
2          struct awbl_rtc_service          *p_next;          // 指向下一个 RTC 服务
3          const struct awbl_rtc_servinfo   *p_servinfo;      // RTC 服务的相关信息
4          const struct awbl_rtc_servopts   *p_servopts;      // RTC 服务的抽象方法
5          void                             *p_cookie;        // RTC 服务自定义的数据
6  };
```

其中包含了四个成员：p_next、p_servinfo、p_servopts 和 p_cookie。下面，依次对各个成员作详细介绍。

（1）p_next

p_next 为指向下一个 RTC 服务的指针，用以将多个 RTC 服务组织成一条单向链表进行管理，详见图 3.5。

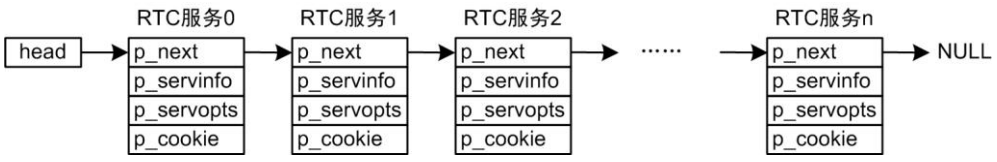


图 3.5　RTC 服务链式管理

可能读者会有疑问，什么时候会有多个 RTC 服务呢？当系统中存在多个 RTC 模块时（如 MCU 内部 RTC、PCF85063、RX8025T、DS1302 等），虽然一般情况下不会同时使用到多个 RTC 模块，但 AWorks 作为一个通用的平台，必须考虑到更全面的应用情况，比如，现在的 MCU 一般都会在内部集成 RTC 外设，CPU 可以直接访问，效率很高，但精度相对较低。外部 RTC 芯片通常采用某种总线（如 I^2C 总线）进行通信，相比于 CPU 直接访问，效率差了很多，但外部 RTC 芯片有功耗低、时间精度高等优点，因此，在部分应用中，可能短时间内，采用内部 RTC，但为避免内部 RTC 时间误差的积累，每隔一定时间从外部 RTC 芯片中读取出精确时间，对内部 RTC 进行校时。

（2）p_servinfo

p_servinfo 表示 RTC 服务的相关信息，其仅包含一个 ID 号，便于用户可以直接通过 ID 访问指定的 RTC 服务，其定义详见程序清单 3.11。

程序清单 3.11　struct awbl_rtc_servinfo 类型的定义

```
1    struct awbl_rtc_servinfo {
2        int                 rtc_id;                                    // RTC 编号
3    };
```

这里增加了一个 ID 号的概念，其有什么意义呢？在第 2 章中，讲解了基于抽象 RTC 类，获取时间和设置时间的 API 原型为（定义详见程序清单 2.3）：

```
1    int itime_time_get    (struct itime *p_this, struct tm *p_tm);
2    int itime_time_set    (struct itime *p_this, struct tm *p_tm);
```

下面仅以获取时间为例进行说明。在获取时间时，需要一个指向“接口实现”的指针（struct itime *p_this），可以根据实际情况，使用指向不同的“接口实现”（如 PCF85063、DS1302）指针。在实际大型项目中，这种方式会使指针越来越多，且到处传递，极有可能造成指针的滥用，使得程序结构变得混乱不堪。而实际上，仅站在用户的角度，该指针无非是指定使用的实现类。因此，AWorks 为避免用户直接使用各类指针，将指针使用一个 ID 号替代，用户仅需使用 ID 来指定其要使用的实现类。例如，假定系统有 4 个 itime 接口的实现类：内部 RTC、PCF85063、RX8025T、DS1302，则它们的 ID 可能分别为 0、1、2、3。对于用户来说，不再需要关心任何指针，使用简单的数字即可。系统内部会自动根据数字，找到相应服务的指针，示意代码详见程序清单 3.12。

程序清单 3.12　根据 ID 寻找 RTC 服务的范例程序

```
1    static struct awbl_rtc_service * __rtc_id_to_serv (int id)
2    {
3        struct awbl_rtc_service *p_serv_cur = __gp_rtc_serv_head;          // 从链表头开始寻找
4
5        while ((p_serv_cur != NULL)) {
6            if (id == p_serv_cur->p_servinfo->rtc_id) {                    // 比较 ID 是否相同
7                return p_serv_cur;
8            }
9              p_serv_cur = p_serv_cur->p_next;
10       }
11       return NULL;
12   }
```

（2）p_servopts

p_servopts 为 RTC 服务定义的抽象方法（抽象接口层的核心，抽象方法的定义是实现依赖反转的关键），其定义详见程序清单 3.13。

程序清单 3.13　struct awbl_rtc_servopts 类型的定义

```
1    struct awbl_rtc_servopts {
2        aw_err_t (*time_get)(void *p_cookie, aw_tm_t *p_tm);          // 读取 RTC 时间
3        aw_err_t (*time_set)(void *p_cookie, aw_tm_t *p_tm);          // 设置 RTC 时间
4    };
```

注：aw_tm_t 是 AWorks 中定义的时间类型，其与标准 C 的时间类型 struct tm 是完全相同的。aw_err_t 为 AWorks 定义的错误号类型，本质上为一个有符号整数类型，一般地，返回 AW_OK（0）表示执行成功，负值表示执行失败。

其中，定义了两个函数指针（抽象方法）：time_get 和 time_set。回顾第 2 章中定义的 itime 接口，其包含的两个抽象方法如下：

```
1    struct itime {
2        int (*pfn_time_get) (struct itime *p_this, struct tm *p_tm);
3        int (*pfn_time_set) (struct itime *p_this, struct tm *p_tm);
4    };
```

将其与程序清单 3.13 中定义抽象方法的形式相比可以发现，这里的抽象方法不是直接存于抽象类中，在抽象类中，仅有一个 p_servopts 指针，其指向了包含各个抽象方法的结构体，此外，抽象方法的第一个参数不再是指向抽象类自身的指针，而是一个 void *指针。

暂时不考虑 AWorks 中的定义方式，如果按照上一节中的做法，抽象类可能定义为（为便于描述，后文将这种定义方式称之为“旧的定义方式”，以区别与 AWorks 中的定义方式）：

```
1    struct awbl_rtc_service {
2        aw_err_t (*time_get)( struct awbl_rtc_service *p_this, aw_tm_t *p_tm);      // 读取 RTC 时间
3        aw_err_t (*time_set)( struct awbl_rtc_service *p_this, aw_tm_t *p_tm);      // 设置 RTC 时间
4
5        struct awbl_rtc_service          *p_next;          // 指向下一个 RTC 服务
6        const struct awbl_rtc_servinfo   *p_servinfo;      // RTC 服务的相关信息
7        // 其它成员
8    };
```

这种方式比较简洁易懂，抽象方法直接包含在抽象类中，且抽象方法的第一个参数为指向抽象类自身的指针。下面一一分析为什么 AWorks 要做这样的改变。

➢ 为什么抽象方法要单独放到一个结构体中？

一般情况下，抽象方法都不止一个，且对于一个具体类，各个抽象方法的实现是确定的，例如，PCF85063 实现获取和设置两个抽象方法，其具体实现在模块内部是确定的，示意代码详见程序清单 3.14。

程序清单 3.14　PCF85063 获取和设置函数的实现

```
1    static int __pcf85063_time_get (struct awbl_rtc_service *p_this, struct tm *p_tm)
2    {
3        // 首先，从 PCF85063 中获取出年、月、日、时、分、秒等信息
```

```
4
5           if (获取成功) {
6                   p_tm->tm_year     =     // 从 PCF85063 中获取出的 年 信息
7                   p_tm->tm_mon      =     // 从 PCF85063 中获取出的 月 信息
8                   p_tm->tm_mday     =     // 从 PCF85063 中获取出的 日 信息
9                   p_tm->tm_hour     =     // 从 PCF85063 中获取出的 时 信息
10                  p_tm->tm_min      =     // 从 PCF85063 中获取出的 分 信息
11                  p_tm->tm_sec      =     // 从 PCF85063 中获取出的 秒 信息
12
13                  return 0;                   // 获取时间成功
14          }
15          return -1;                  // 获取时间失败
16  }
17
18  static int __pcf85063_time_set (struct awbl_rtc_service *p_this, struct tm *p_tm)
19  {
20          // 首先，将 p_tm 中的年、月、日、时、分、秒等信息设置到 PCF85063 中
21
22          if (设置成功) {
23                  return 0;
24          }
25          return -1;
26  }
```

如果按照旧的定义方式，则在 PCF85063 的初始化中，需要依次为各个成员赋值，如：

```
p_dev->isa.time_get = __pcf85063_time_get;
p_dev->isa.time_set = __pcf85063_time_set;
```

这就意味着，设备中需要耗费 2 个指针大小的内存（在 32 位系统中，即为 8 个字节）。如果按照 AWorks 中的定义方式，则首先将两个抽象方法的实现放在一个结构体常量中，即：

```
const struct awbl_rtc_servopts __g_pcf85063_ops = {
        __pcf85063_time_get,
        __pcf85063_time_set
};
```

PCF85063 的初始化可能变化为：

```
p_dev->isa.p_servopts = &__g_pcf85063_ops;
```

这种情况下，在 PCF85063 设备中，仅需耗费一个指针，占用 4 个字节的内存，对于抽象方法特别多的抽象类，内存耗费的对比将更为明显。由此可见，通过将所有抽象函数打包到一个结构体中，可以减小内存的占用。当然，由于新增了结构体常量的定义，ROM 空间将会随之增加，但在实际应用项目中，ROM 往往比 RAM 便宜得多。

至此，读者可能还是会有疑问，虽然降低了 RAM 的占用，但增加了 ROM 的占用，且总的存储空间（RAM + ROM）耗费还会增加一个指针（p_servopts）的占用。实际上，恰好相反，绝大部分情况下，使用 AWorks 中的定义方式会极大的节省存储空间。这是因为具体类对应的对象往往不止一个，例如，系统中存在多个 PCF85063 对象（虽然实际应用不太可能同时使用多个 PCF85063，但这里仅以这种情况作为示例进行说明，讲述其中的道理，读

者可以类比于其它实际情况，例如，使用多个温度传感器采集多个点的温度）：

```
1    struct pcf85063 pcf85063_dev0;                    // 定义一个 PCF85063 对象
2    struct pcf85063 pcf85063_dev1;                    // 定义一个 PCF85063 对象
3    struct pcf85063 pcf85063_dev2;                    // 定义一个 PCF85063 对象
4    struct pcf85063 pcf85063_dev3;                    // 定义一个 PCF85063 对象
5    // .....
```

此时，情况就大不相同了，因为多个 PCF85063 均是 PCF85063 类的实例，每个实例均需要占用内存。

如果按照旧的定义方式，每个实例中直接存储所有抽象方法的实现，则与装载实现函数相关的存储空间耗费为（假定有 10 个实例）：

$$10_{\text{实例个数}} \times 2_{\text{抽象方法个数}} = 20$$

如果按照 AWorks 中的定义方式，每个实例只耗费一个指针内存（p_servopts），再加上结构体常量占用的 ROM 空间（大小与抽象方法个数相同的），则与装载实现函数相关的存储空间耗费为（假定有 10 个实例）：

$$10_{\text{实例个数}} + 2_{\text{抽象方法个数}} = 12$$

由此可见，在旧的定义方法中，实例个数与抽象方法个数之间是乘积关系（且全是占用的 RAM 空间），而在 AWorks 的定义方法中，实例个数与抽象方法个数之间是求和关系（且部分内容占用的是 ROM 空间）。随着抽象类中抽象方法个数的增加与实例个数的增加，两者之间的差异将越来越大。

此外，从运行效率的角度看，AWorks 中的定义方式依然更具有优势，因为在旧的定义方式中，需要依次为每个抽象方法赋值；而在 AWorks 中，无论抽象方法的个数是多少，都仅需赋值一次（将结构体首地址赋值给 p_servopts）。

基于此，AWorks 统一改变了定义方式，将所有抽象方法打包到一个结构体中，实际上，这种方式并非 AWorks 独创，其在行业内被大量使用，装载抽象方法的结构体被称之为“虚函数表”。

- 为什么将抽象方法的第一个参数修改为了 void *指针？

在第 2 章的“实现类”中，基于继承思想，通常将待实现的接口作为第一个成员，例如：

```
struct pcf85063 {
    struct itime    itime;
    // ... 其它成员，如 PCF85063 使用的引脚信息、I²C 总线信息等
};
```

以便在接口实现中，将 struct itime *类型的指针强制转换为 struct pcf85063*类型的指针使用，以便进一步访问类中的各个私有成员。例如：

```
static int __pcf85063_time_get (struct itime *p_this, struct tm *p_tm)
{
    struct pcf85063*p_dev = (struct pcf85063*) p_this;
    // 通过 p_dev 访问 PCF85063 设备自身相关的私有成员


    // 其它操作
}
```

试想一下，如果一个设备实现了两种接口，如果继续基于继承的思想，则继承自两个抽象类（例如，部分 RTC 芯片具有闹钟功能，在提供实时时钟功能的同时，还可以提供闹钟功能），此时，按照设计规则，两个抽象类都必须作为第一个成员，显然，这是无法做到的。这是 C 语言的局限性所造成的（C 语言不是一种面向对象的编程语言，因此，在使用面向对象编程时，需要利用一些技巧）。

另一方面，通过前面的实现可以发现，抽象方法是由具体类实现的，因此，抽象方法的第一个参数实则是在具体类的实现函数使用的，抽象接口层并不会使用该参数，也就无需关心该参数的具体值。

基于此，为了得到更大的灵活性，AWorks 在定义抽象方法时，将抽象方法的第一个参数的类型设置为了 void *类型（抽象接口层不会使用该参数，不用关心该参数的具体类型），其值由抽象方法的实现者（即具体类）决定，抽象接口层仅在调用抽象方法时，将值传递给抽象方法的具体实现。

抽象接口层为了保存该参数的值，以便在调用抽象方法时传递，在抽象的 RTC 服务中，增加了一个 void*类型的 p_cookie 成员，其具体赋值由实现类完成（在实现抽象方法的同时，完成 p_cookie 的赋值，后文介绍 p_cookie 成员时，再作进一步介绍），抽象接口层仅负责简单的传递，例如，若需获取时间，则调用形式如下：

```
p_serv->p_servopts->time_get(p_serv->p_cookie, p_tm);
```

同理，为了避免其它模块直接访问抽象类中的函数指针成员，可以将函数指针的访问放在模块内部，使外部模块可以直接调用相应的 API 获取时间或设置时间，AWorks 中定义了获取时间接口和设置时间接口，详见表 3.1。

表 3.1　RTC 通用接口函数（aw_rtc.h）

函数原型	功能简介
aw_err_t aw_rtc_time_get (int rtc_id, aw_tm_t *p_tm);	获取时间
aw_err_t aw_rtc_time_set (int rtc_id, aw_tm_t *p_tm);	设置时间

将 aw_rtc_time_get()与第 2 章中介绍的 itime_time_get()对比可以发现，主要变化仅仅是第一个参数由 struct itime *类型的 p_this 修改为了 int 类型的 rtc_id。类似的，这两个接口的实现同样不涉及硬件细节，仅需通过调用抽象方法实现即可，示意代码详见程序清单 3.15。

程序清单 3.15　获取和设置时间接口的实现

```
1   aw_err_t aw_rtc_time_get (int rtc_id, aw_tm_t *p_tm)
2   {
3       struct awbl_rtc_service *p_serv = __rtc_id_to_serv(rtc_id);
4       if (p_serv != NULL) {
5           return p_serv->p_servopts->time_get(p_serv->p_cookie, p_tm);
6       }
7       return -AW_ENXIO;                          // AWorks 定义的错误号，表示设备不存在
8   }
9
10  aw_err_t aw_rtc_time_set(int rtc_id, aw_tm_t *p_tm)
11  {
12      struct awbl_rtc_service *p_serv = __rtc_id_to_serv(rtc_id);
```

```
13          if (p_serv != NULL) {
14              return p_serv->p_servopts->time_set(p_serv->p_cookie, p_tm);
15          }
16          return -AW_ENXIO;                                   // AWorks 定义的错误号，表示设备不存在
17      }
```

可以发现，在接口的实现中，增加了一个将 rtc_id 转换为指向 RTC 服务指针的过程。

（4）p_cookie

如同上面提到的问题，为了提高程序设计的灵活性，使继承时不被成员在结构体中的位置所限制。抽象方法中的第一个参数都设置为了 void *类型的参数，此处的 p_cookie 即为调用抽象方法时，传入第一个参数的值。p_cookie 成员的具体赋值由实现类完成，抽象接口层无需关心 p_cookie 的具体值及其含义。

例如，在 PCF85063 的初始化函数中，完成抽象方法的赋值时，同样需要完成 p_cookie 的赋值，一般地，为了在具体函数实现中访问设备自身的成员，通常将其设置为指向设备自身的指针，即：

```
p_dev->isa.p_servopts = &__g_pcf85063_ops;
p_dev->isa.p_cookie   = p_dev;
```

如此一来，在抽象方法的实现中，同样可以将 p_cookie 转换为指向设备自身的指针使用，达到和 p_this 指针一样的效果，示意程序详见程序清单 3.16。

程序清单 3.16　PCF85063 获取函数的实现——使用 p_cookie 指针

```
1     static aw_err_t __pcf85063_time_get (void *p_cookie, aw_tm_t *p_tm)
2     {
3           struct pcf85063 *p_dev = (struct pcf85063 *)p_cookie;
4
5           // 通过 p_dev 访问 PCF85063 设备自身相关的私有成员
6
7           // 其它操作
8     }
```

使用这种方法时，无论 isa 成员在具体类中的位置如何（无论是否为第一个成员），上述关系均成立，均可在实现函数中，将 p_cookie 转换为指向自身的指针使用，达到 p_this 目的。由此可见，通过这种设计方式，消除了抽象类在具体类中的位置要求，极大的提高了程序设计的灵活性。

3.3.2 应用程序的实现

对于 AWorks 来讲，上层模块即为具体的应用程序，如电子表应用（elec_watch），其与程序清单 2.4 所示的应用程序设计基本一致。不同的是，由于 AWorks 将通过指针访问抽象接口层优化为了通过 ID 访问抽象接口层，因此，应用程序的启动函数无需再传入指针参数，仅需传入一个 rtc_id 即可，范例程序详见程序清单 3.17。

程序清单 3.17　应用程序实现范例

```
1     void app_elec_watch (int rtc_id)
2     {
3           struct tm now_tm;
```

```
4
5           if   (用户修改了当前时间) {
6                   struct tm set_tm = ...                                  // 用户设置的时间
7                   aw_rtc_time_set(rtc_id, &set_tm);                       // 设置时间
8           }
9
10          aw_rtc_time_get(rtc_id, &now_tm);                               // 获取当前时间
11
12          printf("Now time is : %04d-%02d-%02d %02d:%02d:%02d \r\n",
13                          now_tm.tm_year+1900,    now_tm.tm_mon + 1,      now_tm.tm_mday,
14                          now_tm.tm_hour,         now_tm.tm_min,          now_tm.tm_sec);
15
16          // ...其它处理，如在 LCD 上显示时间等
17  }
```

为了启动该应用程序，直接在主程序中调用 app_elec_watch()，并传入一个合适的 rtc_id 即可。范例详见程序清单 3.18。

程序清单 3.18　主程序范例程序（使用 ID 为 1 的 RTC）

```
1   #include "pcf85063.h"
2   struct pcf85063 pcf85063_dev;
3   int aw_main()
4   {
6           app_elec_watch (0);                                             // 启动“电子表”应用程序
7           while (1) {
8           }
9   }
```

注：在 AWorks 中，应用程序入口为 aw_main()。

程序中假定使用 ID 为 0 的 RTC，该应用程序要正确运行，比如需要 ID 为 0 的 RTC 存在。下节会继续介绍如何实现抽象方法，并为具体的 RTC 器件指定其对应的 ID。

与程序清单 2.7 所示的主程序对比可以发现，虽然接口提供的 API 仅仅是指针类型的 p_this 变为了整数类型的 rtc_id，但是这种变化使得应用程序的设计变得更为便捷。在使用 p_this 指针时，在启动应用程序前，必须得到一个有效的 p_this，详见程序清单 2.7 的第 5 行；而使用整数类型的 rtc_id 后，可以直接使用整形值（0、1、2……），使应用程序的设计不受到指针的任何约束，即不用考虑指针的指向（有效的指针如何获取）。当然，在实际系统中，必须保证 rtc_id 对应的“RTC 服务”是有效的。

3.3.3 抽象方法的实现

在定义抽象的 RTC 服务时，抽象方法采用了“虚函数表”的形式，指向设备自身的指针 p_this 采用了 p_cookie 的形式，因此，抽象的 RTC 服务不一定作为具体类的第一个成员。同样以 PCF85063 为例，可以定义相应的实现类，详见程序清单 3.19。

程序清单 3.19　定义具体的 PCF85063 类

```
1   struct pcf85063 {
2           // ... 其它成员
```

```
3           struct awbl_rtc_service        rtc_serv ;
4           // ... 其它成员
5       };
```

PCF85063 要实现的主要功能即为获取时间与设置时间，实现范例详见程序清单 3.20。

程序清单 3.20　实现 RTC 服务中的抽象方法

```
1       static int __pcf85063_time_get (void *p_cookie, aw_tm_t *p_tm)
2       {
3           struct pcf85063 *p_dev = (struct pcf85063 *)p_cookie;
4
5           // 首先，从 PCF85063 中获取出年、月、日、时、分、秒等信息
6
7           if (获取成功) {
8               p_tm->tm_year     =     // 从 PCF85063 中获取出的 年 信息
9               p_tm->tm_mon      =     // 从 PCF85063 中获取出的 月 信息
10              p_tm->tm_mday     =     // 从 PCF85063 中获取出的 日 信息
11              p_tm->tm_hour     =     // 从 PCF85063 中获取出的 时 信息
12              p_tm->tm_min      =     // 从 PCF85063 中获取出的 分 信息
13              p_tm->tm_sec      =     // 从 PCF85063 中获取出的 秒 信息
14
15              return 0;               // 获取时间成功
16          }
17          return -1;              // 获取时间失败
18      }
19
20      static int __pcf85063_time_set (void *p_cookie, aw_tm_t *p_tm)
21      {
22          struct pcf85063 *p_dev = (struct pcf85063 *)p_cookie;
23
24          // 首先，将 p_tm 中的年、月、日、时、分、秒等信息设置到 PCF85063 中
25
26          if (设置成功) {
27              return 0;
28          }
29          return -1;
30      }
31
32      const struct awbl_rtc_servopts __g_pcf85063_ops = {
33          __pcf85063_time_get,
34          __pcf85063_time_set
35      };
```

同理，在使用 PCF85063 之前，必须完成对象的初始化，核心是完成各指针的赋值，示意代码详见程序清单 3.21。

程序清单 3.21　PCF85063 初始化函数实现

```
1    int pcf85063_init (struct pcf85063 *p_dev)
2    {
3        p_serv = &p_dev ->rtc_serv;
4
5        p_serv->p_next          = NULL;
6        p_serv->p_servopts      = &__g_pcf85063_servopts;
7        p_serv->p_cookie        = (void *)p_dev;                    // p_cookie 为设备自身
8
9        // 其它成员赋值，如 ID 信息（一般通过配置信息完成）
10       p_serv->p_servinfo = ...;
11       return 0;
12   }
```

其中，RTC 器件对应的 ID 信息通常是可以配置的，因此，其 ID 信息往往定义在外部的配置文件中，例如：

```
const struct awbl_rtc_servinfo pcf85063_servinfo = {
    0
};
```

该配置信息可以通过特定的方式传入底层驱动中。这里仅简单地展示了抽象方法的实现过程，使读者理解 AWorks 中“依赖反转”的实现方法。实际上，在 AWorks 中，外围设备及驱动统一由 AWBus-lite 进行管理，包括设备的初始化（简言之，用户无需在主程序中手动调用 PCF85063 的初始化函数）以及将“设备实现的服务”提交给上层（如 PCF85063 实现的 RTC 服务，需要加入到 RTC 服务链表中）。这些具体的实现细节就涉及到完整的驱动开发流程，需要对 AWBus-Lite 有一定的了解。更多信息可以参考《面向 AWorks 框架和接口的 C 编程》（上）的第 12、13 章（京东链接：https://item.m.jd.com/product/12486808.html），或者《AWBus 设备驱动开发指南》。

第4章 组件简介

本章导读

上一部分（第 1 章 ~ 第 3 章）重点介绍了类的设计。在大型系统的开发中，如果以类为单位进行管理，将变得十分繁琐（因为类太多了）。因此，在实际的项目开发中，往往以组件的形式进行管理，一个组件可以包含多个类，本章将介绍组件相关的一些基本概念。

4.1 组件的定义

组件（Component）是软件的部署单元，是整个软件系统在部署过程中可以独立完成部署的最小实体。组件能够以某种形式（比如，“.zip”文件、“.DLL”文件、“.jar”文件）独立发布。

4.2 组件化的优势

当软件系统被组件化后，相当于把整个软件系统拆分为了一个一个可独立部署的组件。如此一来，软件系统的开发与维护将从一个大的系统中解脱出来，聚焦于一个一个独立的组件以及组件之间的关系。

总结起来，组件化主要有以下几个优点：

(1) 易开发：不同的组件可以交由不同的团队同时开发；

(2) 易维护：维护工作由维护整个系统分解到维护各个独立的组件；

(3) 易部署：每个组件都可以独立部署；

(4) 易重用：每个组件实现了某种单一的功能，高内聚，低耦合，更易重用；

(5) 方便管理：各个组件可以使用工具进行管理，为组件的维护和重用提供极大的便利；

(6) 可以标准化开发过程：以组件为核心，可以制定标准化的开发过程，类似于本文档。

相信绝大部分开发者都有过这样的经历：当花费一整天的时间，好不容易搞定了一段代码，第二天上班时却发现这段代码莫名其妙地又不能工作了。出现这种情况的原因，很有可能是因为其他人修改了部分你依赖的代码。

在传统的非组件化开发模式中，多个程序员往往会同时维护一个源码目录，修改、完善里面的代码。每个人都在修改自己负责部分的代码，当修改好后，其他人也修改了代码，此时，又不得不继续修改自己的代码，以适应其他人所提交的修改，如此恶性循环，使得要发布一个稳定的项目软件变得十分困难，往往需要所有开发者一起，花费大量的时间，处理好各式各样的兼容性问题，最终才能得到一个稳定的版本。

当将研发项目划分为一些独立的组件后，这些组件可以交由单人或一个团队来负责开发，各个组件都有属于自己的源码目录，源码的修改和维护互不影响。当一个组件完成某个版本后，可以为之分配一个版本号，放入一个贡献目录，并通知到其他需要使用该组件的开发者。这样一来，每个人都可以依赖于这些组件公开的发布版本来进行开发，而组件开发者则可以继续去修改自己的私有版本。

在共享目录中，存放了一个组件的所有版本，每当一个组件发布新版本后，其他依赖这个组件的团队都可以自主决定是否立即采用新版本，若不采用，该团队可以继续使用旧版本，直到他们准备好采用新版本为止。

如此一来，任何一个组件的变更都不会立即影响到其它团队，在开发过程中，就不会出

现因其他人修改了代码而导致不能正常工作的情形了，组件的开发进程完全由组件开发者决定，不会因其依赖组件的更新而受到影响。何时选用新版本将由组件开发者自主决定，更新工作（更新至使用指定版本）自己就可以完成，不再需要所有开发者集中在一起，花费大量的时间，共同处理更新、集成问题。

4.3 组件的设计原则

在组件化时，需要保持良好的组件粒度，若组件粒度过细（一味的追求“职责单一”），极端地，可能把每对.c 和.h 都视为一个组件，这将使得发布的组件非常之多，管理和维护起来也并非易事。反之，若组件粒度过粗，将很多相干的、不相干的代码都整合到一个组件中，则组件化程度不够，与没有组件化时的差异可能不大，进而达不到组件化的真正目的。

基于此，为了保持良好的组件粒度以及可维护性，可复用性等，在构建一个组件时，可以参照如下三个基本原则，综合考虑，权衡利弊，以构建良好的组件。

- 复用/发布等同原则：REP，Release Reuse Equivalency Principle；
- 共同闭包原则：CCP，Common closure Principle；
- 共同复用原则：CRP，Common Reuse Principle。

下面分别对各个原则作详细介绍。为便于描述，将组件视为由一个类（或模块）或多个类（或模块）组成，虽然在使用非面向对象编程语言（比如 C）编程时，没有类（Class）的语法，但在基于“面向对象编程”的思想进行编程时，所有的一切（代码或模块）都可以视为类和对象。

4.3.1 复用/发布等同原则（REP）

REP：软件复用的最小粒度应等同于其发布的最小粒度。

该原则旨在确保组件的可复用性，其表达的意思是：组件复用是基本要求，组件中的内容，应该为复用性而组合在一起。组件中的各个部分，必须是彼此紧密联系的，也就是说，一个组件不能由一组毫无关联的类组成，它们之间应该有一个共同的主题或者大方向。

在追求软件复用的过程中，需要有标准的发布流程，例如：版本号（语义化版本），发布时间，变更内容等，只有那些通过版本追踪系统发布的组件才能被高效地复用。

组件中所包含的类必须是可以同时发布的，这意味着它们共享相同的版本号与版本跟踪，并且包含在相同的发行文档中，这些内容都应该同时对该组件的作者和用户有意义。其深层含义既是对用户的承诺，也是对作者的约束。组件是否向后兼容？是否包含破坏性的变更？升级的注意事项？

4.3.2 共同闭包原则（CCP）

CCP：应该将那些会同时修改，并且为相同目的而修改的类放到同一个组件中，而将不会同时修改，并且不会为了相同目的而修改的类放到不同的组件中。

该原则旨在确保组件的可维护性，其表达的意思是：尽量把变更原因、变更频率相同的类和模块放到同一个组件当中。这样做的好处是，当相关功能更新时，可以把源代码的变更局限在某一个组件当中，而不需要横跨多个组件，从而减少部署、验证和发布的次数。

从另一个角度看，也可以将其视为针对组件设计的“单一职责原则”，即一个组件不应该同时存在多个变更的原因，一个组件中的所有类和模块，变更的原因应该是相同的。

对大部分应用程序来说，可维护性的重要性要远远高于可复用性，如果某程序中的代码必须要进行某些变更。那么这些变更最好都体现在同一个组件中，而不是分布于很多个组件中，因为如果这些变更都集中在同一个组件中，在变更结束后，就只需要重新部署该组件，

其它组件不需要被重新验证、重新部署。

总而言之，CCP 的主要作用就是约束设计者，要将所有可能会被一起修改的类或模块集中在一起。这是局部化影响的优势，封装可变因素，将同一时间变更的点聚合到一个组件中，以此形成一个“闭包”。

4.3.3 共同复用原则（CRP）

CRP：不要强迫一个组件的用户依赖他们不需要的东西。

该原则旨在避免不必要的发布，将用户需要同时复用到的类和模块放在同一个组件中，它们会被用户同时使用到，功能应该是密切关联的。

CRP 不仅仅用于指导应该将哪些内容放到同一个组件中，更重要的是，指导应该将哪些类或模块分开。

当一个组件引用了另一个组件时，就等于增加了一条依赖关系。由于这种依赖关系的存在，每当被依赖的组件发生变更时，依赖它们的组件一般也需要作出相应的变更。即使它们不需要进行代码级的变更，一般也免不了需要被重新编译、验证和部署，哪怕它们根本不关心被依赖组件中的变更。

因此，当依赖一个组件时，最好是依赖其中所有的类或模块，而不是仅依赖部分的类和模块。换句话说，组件中的类和模块，最好是不能拆分的，即不应该出现用户只依赖部分类的情况，否则，后续就会花费大量的时间与精力来做不必要的组件部署。

基于 CRP，组件和组件之间的依赖应该达成一种默契——如果不需要完全使用某个组件中所有的模块，那么就不要依赖它。

从另一个角度看，CRP 也可以视为针对组件设计的“接口隔离原则”，即一个组件不应该依赖带有冗余类和模块（不会使用到的类和模块）的组件。简而言之，就是不要依赖不需要用到的东西，否则，这些即使不会用到的东西，更新或变更也会产生很大的影响。

4.3.4 设计原则小结

前面简要描述了组件的三个设计原则，它们之间彼此存在一定的竞争关系，REP 和 CCP 原则是黏合性原则，主要指导将哪些代码集合到一个组件中，它们会让组件变得更大，而 CRP 原则是排除性原则，重点是指导排除不需要依赖的东西，它会让组件变得更小。在设计过程中，应该综合考虑，平衡各方利弊，不能一味的追求满足某些原则。例如，只关注 REP 和 CRP，忽略 CCP，则会出现这种现象：即使是简单的变更，也会同时影响到许多组件。相反，若只关注 CCP 和 REP，忽略 CRP，则会导致过多冗余的发布。

实际上，随着项目的发展，对原则的关注点可能发生变化，例如，在项目初期，还没有可复用性的明显需求，此时，可维护性可能比可重用性更加重要，对 REP 的关注就会较少，而随着时间的推移，可复用性可能变得越来越重要，此时，就会更加注重 REP。因此，用户不必过渡设计，始终想着如何适应所有原则，设计出最优秀的组件，组件设计和代码设计是一样的，存在一个递进的过程，随着开发的进行，可以在开发过程中不断完善。

本节内容较为抽象，当组件开发经验较少时，较难深刻理解这些原则，因而实际上也很难使用好各个原则。开发者可以通过本节内容对组件设计原则留下一定的印象，在实际开发过程中遇到问题时，比如为某个设计决策犹豫不决时，再来看看这些原则，可能会带给你一些启发，帮助你更快、更好、更果断的做出决策。

4.4 特殊的组件

4.4.1 框架（Framework）

框架是一个较为抽象的概念，通常情况下，可以将一些复杂的、大型的、可供用户使用（基于其进行产品的设计与开发）的软件称之为“框架”。

一种通俗的定义是：框架是一种经过校验、具有一定功能的半成品软件。经过校验是指框架本身经过测试，且框架自身所具有的功能已经实现；具有一定功能是指框架可以完成特定的功能；半成品软件是指框架自身是一个软件，但其往往不是最终的产品，最终产品需要用户基于框架编写其它程序才能完成。

框架主要有以下 3 个特点：

(1) 具有一定的规模

框架通常是具有一定规模的软件，正因为如此，框架通常包含了多个子组件（内置组件），以将框架划分为多个“子问题”进行处理。

(2) 回调机制

这里所指的回调机制是一种广义的回调机制，即提供了一种方法，在合适的时机调用用户提供的函数。用户仅需关心这些会被框架调用的函数设计，框架内部的实现机制，用户往往无需关心。

(3) 控制了流程

框架同时制定了一系列规约。软件设计不能“随心所欲”，必须在框架的约束下进行。这些规约在一定程度上控制了软件的设计和运行流程。

操作系统可以视为一种典型的框架。首先，操作系统一般都不简单，包含了诸多代码，具有一定的规模；其次，操作系统提供了诸多服务，实现了很多功能，也具有回调机制。一种典型的回调机制就是用户程序入口函数（例如，AWorks 中的 aw_main()），用户程序从 aw_main()开始执行，至于 aw_main()之前及之后（假如 aw_main()函数返回了）做了什么处理，用户无需关心，完全由操作系统（框架）负责。这在一定程度上也控制了整个程序的流程，约定了用户程序从何处开始编写，从何处开始执行。

常见的框架有：AWorks、AMetal、AWTK 等。框架可以视为一种特殊的组件，其与普通组件相比，主要特殊在：

(1) 规模的不同

前文已经提到，框架往往具有一定的规模，为了便于管理，框架内部往往还会包含子组件。而普通组件就是单一的组件，功能、职责和接口等都相对单一。

(2) 影响范围的不同

由于很多软件都会基于框架开发，因此，框架变化时造成的影响往往是全局的，影响众多组件；而普通组件的变化，造成的影响往往是局部的，只会影响组件自身或者依赖于它的几个组件。例如，AWorks 完成了一些通用宏的定义，比如标准错误码，一些常用的标准错误码详见表 4.1。

表 4.1 常见错误号的含义

错误号	含义
AW_EPERM	操作不允许
AW_ENOENT	文件或目录不存在

续上表

错误号	含义
AW_ESRCH	进程不存在
AW_EINTR	调用被中断
AW_EIO	I/O 错误
AW_ENXIO	设备或地址不存在
AW_E2BIG	参数列表太长
AW_ENOEXEC	可执行文件格式错误
AW_EBADF	文件描述符损坏
AW_ECHILD	没有子进程
AW_EAGAIN	资源暂不可用，需重试
AW_ENOMEM	空间（内存）不足
AW_EFAULT	地址错误
AW_ENOTEMPTY	目录非空
AW_EBUSY	设备或资源忙
AW_EEXIST	文件已经存在
AW_EXDEV	跨设备连接
AW_ENOSPC	设备剩余空间不足
AW_EPIPE	损坏的管道

这就意味着，运行在 AWorks 中的任一函数，只要其返回值为标准错误码类型：aw_err_t。则具有如下含义：

- 若返回值为 AW_OK，则表示操作成功；
- 若返回值为负数，则表示操作失败，失败的原因可根据的错误号（表 4.1 中仅列出了部分错误号）判断；
- 若返回值为正数，通常也表示操作成功，且返回的正值还包含了一些有意义的信息，例如，在使用 aw_nvram_get()接口获取存储在非易失存储器中的数据时，其返回值表示成功读取到的字节数。

显然，若标准错误码的定义发生改变，则将影响 AWorks 中所有返回值类型为 aw_err_t 的函数，因此，框架中定义的东西，一般不会轻易改变，框架的更新将比普通组件的更新更为谨慎。

4.4.2 平台（Platform）

平台表示了软件运行的一种环境，主要分为两类：通用桌面平台和嵌入式平台。

1. 通用桌面平台

这类平台具有通用性，比如 win32 平台，基于这些平台开发的软件可以在这些通用平台上直接运行，而与具体硬件无关。

2. 嵌入式平台

相比于桌面平台，嵌入式平台更加复杂，芯片种类繁多，通用性差，不同厂商、不同型号的芯片差异较大。嵌入式平台可以理解为具体芯片家族的软件支持包，即基于该软件支持包，用户可以更加便捷的使用芯片。比如 RT105x 系列 MCU，其结构框图详见图 4.1。

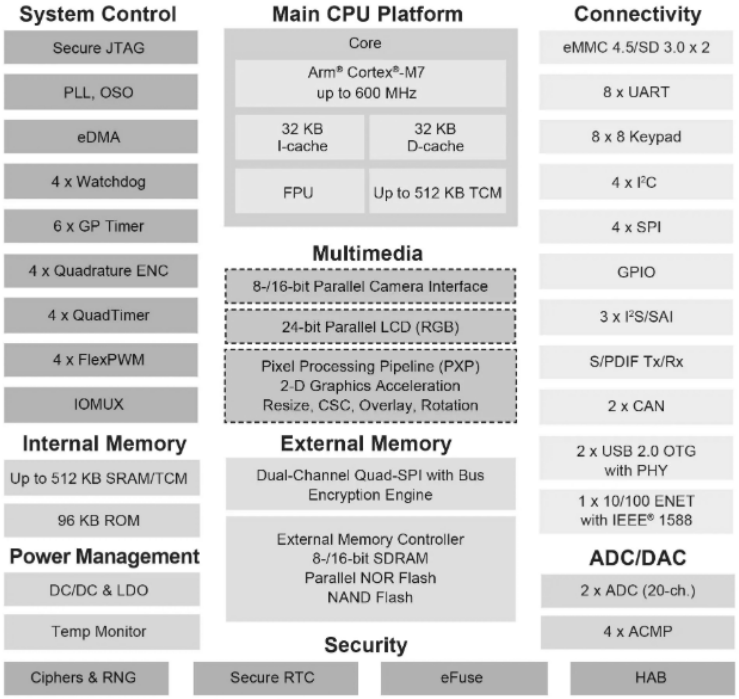


图 4.1　RT105x 结构框图

由图可见，在 RT105x 芯片家族中，MCU 包含了 CPU 及一系列片上外设：CPU 包含 ARM Cortex-M7 内核以及 I-cache、D-cache、FPU、TCM 等，片上外设包含 SD、UART、I²C、SPI、GPIO、I²S 等。

平台中包含了芯片家族相关的代码，比如 CPU 启动文件、链接脚本以及片上外设的驱动（串口驱动、ADC 驱动……）等。

4.4.3 板级（Board）

这类组件主要用于描述一个具体的硬件板，与实际的硬件板设计相关，比如，EasyARM-RT1052，实物图详见图 4.2。

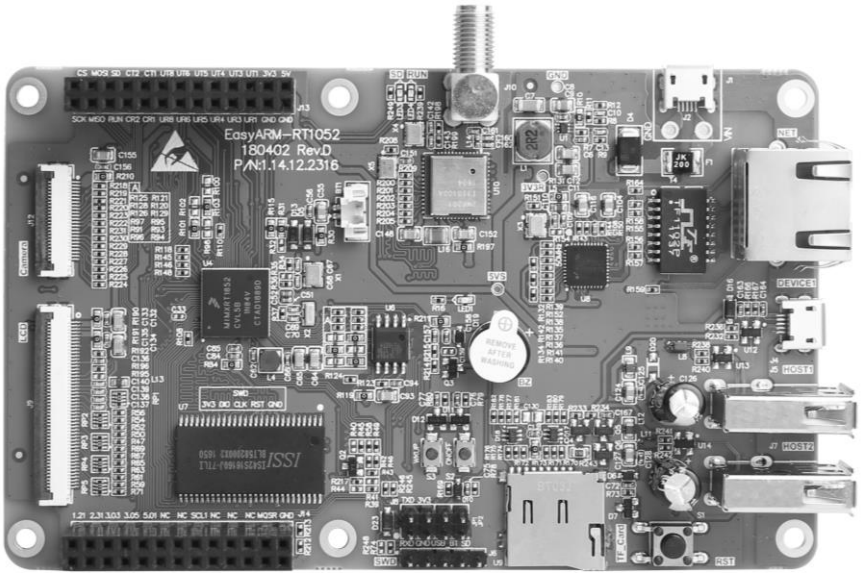


图 4.2　EasyARM-RT1052 实物图

板级组件主要描述了硬件板上的相关资源及配置。例如，对于 EasyARM-RT1052，其具有如下板级资源：

(1) CPU：RT1052，32KB 的 ICache 和 32KB 的 DCache；

(2) 32MB 的 SDRAM，最高频率可达 166M；

(3) 4 个兼容 I^2C 总线标准、在 FM+模式下数据传输速率可高达 1Mbit/s 的 I^2C 接口；

(4) 8 个全双工、标准 NRZ 格式、可编程 7/8-bit 数据位、高达 5.0Mbit/s 的 UART；

(5) 4 个接收/发送 FIFO 达 16 字的 SPI 接口；

(6) 支持 scatter/gather 的 Enhanced DMA；

(7) 高达 1MS/s、具有 10 个单端外部模拟输入、12 位分辨率、10/11 位精度的 ADC；

(8) 1 个 10/100Mbit/s 的以太网接口，支持 IEEE1588 协议；

(9) 1 个 ZW6201 的 WIFI 模块，符合 IEEE802.11b/g/n 规范；

(10) 2 个 CAN 总线接口；

(11) 1 个 8 位 CSI 接口；

(12) 2 个 USB 2.0 接口；

(13) 1 个 SD 卡接口；

(14) 1 个 LCD 接口；

(15) 1 个无源蜂鸣器。

注意，仅嵌入式平台才有对应的板级，通用桌面平台没有板级的概念。这是因为通用桌面平台是通用的，与具体硬件（芯片、硬件板等）无关，换句话说，win32 应用程序，不会因为 CPU 或主板的变化而变化；而嵌入式平台是一个芯片家族的软件支持包，芯片家族往往包含一系列芯片，而实际应用使用的是一个特定型号的芯片，板级组件确定的描述了使用芯片家族中的哪个芯片。显然，芯片本身并不能直接单独使用，还必须基于芯片制作相关的硬件板（核心板、评估板等）后才能使用，这也是嵌入式开发的一般模型。

1. 板级与嵌入式平台的关系

从面向对象编程的角度看，板级可以看作嵌入式平台（后文将其简称为“平台”，若无特殊说明，“平台”均指嵌入式平台，而非通用桌面平台）的具体实例。即：平台可以看作类，是抽象的，板级是一个确定的对象，是具体的。平台类抽象的表示了一组板级对象，它们具有共同特征：以平台表示的芯片家族中的某一具体芯片为核心，添加一些外围器件形成的产品硬件原型。例如，ZLG 以 RT105x 芯片家族为核心，推出了两种型号的硬件板：EasyARM-RT1052、M105x-EV-Board，它们与平台之间的关系示意图详见图 4.3。

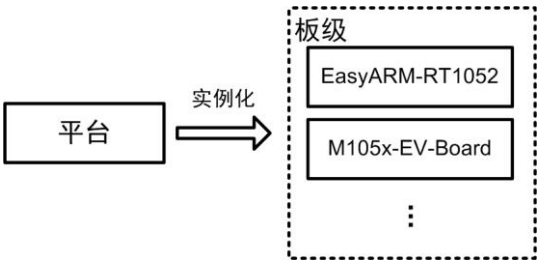


图 4.3 平台与板级之间的关系

实际上，不同应用目的可以有不同的硬件板，使得同一平台对应的硬件板十分丰富，以 RT105x 芯片家族为例，除 ZLG 推出的硬件板外，其它以 RT105x 为核心做各式各样应用的

厂商，每种产品对应的硬件板，都可以看作平台的一个实例。

嵌入式平台的概念相对较大，读者可能不易理解。可以从一个具体的点出发（例如，平台提供的 UART 驱动），来理解平台与板级之间的关系。例如，平台提供了 UART 驱动，UART 驱动是一个类，其往往适用于该芯片家族中的多个串口：UART0、UART1、UART2、UART3……也就是说，多个串口可以复用一份驱动。但具体使用了多少个串口，则由板级确定。示意图详见图 4.4。

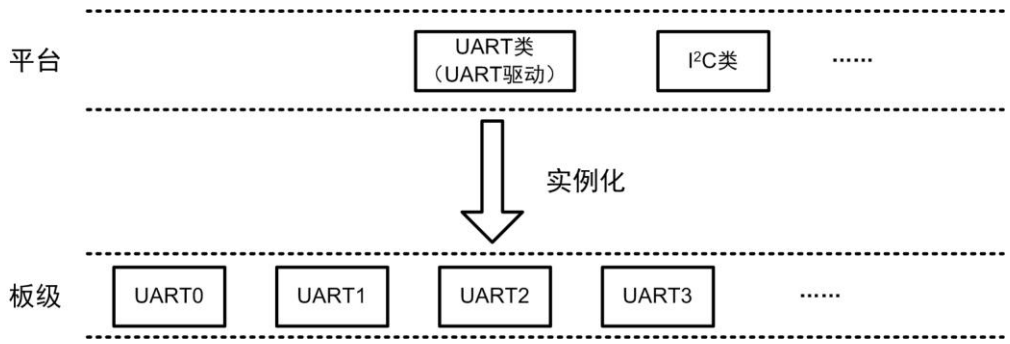


图 4.4　从 UART 驱动的角度理解平台与板级之间的关系

由此可见，板级上的各个串口，实际上就是平台提供的 UART 驱动的具体实例，板级的定义相当于完成了串口驱动类的实例化。从更加宏观的角度看，板级就是平台实例化的结果，是平台的一个具体实例。这也就意味着，嵌入式平台（抽象的）不可脱离于板级（具体的）而单独使用，在嵌入式系统中，用户都是基于板级作应用程序的开发。

在实例化过程中，还完成了相关资源的裁剪和配置。

- 裁剪

嵌入式平台通常提供了芯片家族中所有外设的驱动，但具体硬件板并不一定全部使用，例如，虽然 RT105x 支持 LCD，但部分应用可能不需要使用 LCD，此时，应用对应的硬件板就没有 LCD 接口。

- 配置

例如，平台提供了 UART 驱动，但 UART 具体使用的 I/O 口在不同硬件板中可能不同，I/O 口的确定通常在硬件设计时完成，因此，这些 I/O 口的描述也将由板级组件完成。

2. 板级在软件体系结构中可以发挥重要的作用

在“接口”的介绍中，提到了接口设计的重要性。接口可以使应用程序与底层硬件（芯片型号、硬件驱动等）完全隔离，实现应用程序的跨平台复用。也就是说，应用程序仅依赖于抽象的“接口”，不依赖于具体的硬件。应用程序所依赖的接口需要通过某种方式（如参数）进行传递，详见程序清单 2.7 所实现的应用程序。

由于应用程序接口参数的存在（如 app_elec_watch()函数的 p_time 参数），使得在启动应用程序时（通常在 main()函数中），需要先从具体的硬件中获取到相应的接口实现（通常可以通过初始化函数完成），再将其传递给应用程序，详见程序清单 2.10 中所示的主程序实现。对应的关系图详见图 2.2。由此可见，虽然应用程序与具体硬件无关，但实际用于启动应用程序的主程序，依然与具体硬件绑定，任何底层硬件的修改都将导致主程序的修改。

实际上，对于绝大部分软件，做到这种程度已经不错了，毕竟，修改任何底层硬件，只需要修改主程序，按照组件化思想编写的应用程序并不需要做任何改动。

但是，考虑到在嵌入式系统中，底层硬件非常多。各种通信总线：SPI、I²C、USB 等，各种外围器件：PCF85063、EEPROM、SPI Nor FLASH、NAND FLASH 等，如果每个硬件

设备都需要用户在主程序中完成初始化，并获取相应的“接口实现”，则主程序将会随着硬件设备的增加渐渐的显得臃肿不堪，难以维护。

由于板级描述了系统的硬件资源及布局，因此，在上电时系统可以根据板级描述自动完成系统硬件的初始化，并将所有硬件提供的“服务”保存下来，为了便于寻址，为每个“服务”分配一个唯一 ID，使得后续可以使用 ID 索引到相应的“服务”。

例如，硬件板上有 3 个 RTC：芯片内部 RTC、PCF85063 和 DS1302，它们均可提供 RTC 服务（实现 RTC 接口中定义的抽象方法：获取时间与设置时间），则可以将他们提供的 RTC 服务（抽象接口实现）保存在系统中，并为各个服务分配一个唯一 ID，比如：0、1、2。应用程序即可直接基于 ID 编程，系统中间层根据 ID 找到相应的服务（接口实现），然后通过这些接口完成相应的操作（获取时间、设置时间等）。这个过程对应的示意图详见图 4.5。

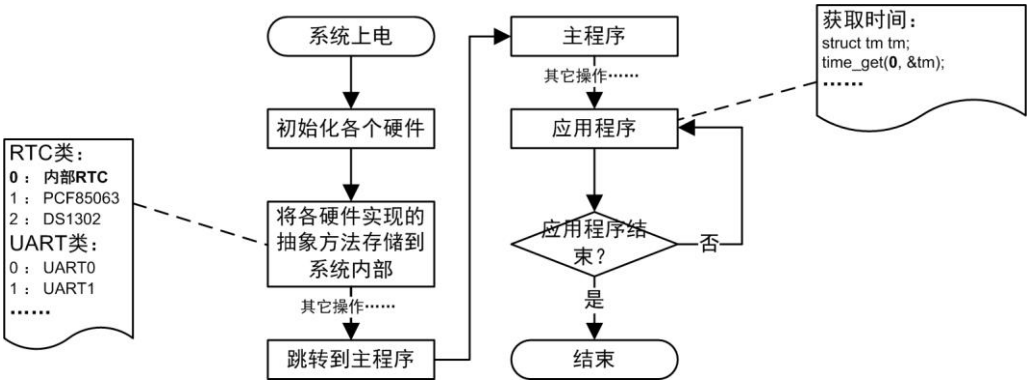


图 4.5　系统上电自动完成硬件初始化

系统中具体如何保存硬件设备提供的服务，以及 ID 和接口实现如何对应，不同的系统可能有不同的实现方法，但核心思想是一样的。读者仅需体会到，板级的定义，完成了硬件资源的描述，系统可据此自动完成相关硬件的初始化，无需用户在主程序中手动处理。

这样的处理还有一个额外的好处：可以将抽象方法的定义封装到系统内部，避免应用程序“看”到过多的细节。以获取时间为例，在添加系统中间层之前，获取时间的接口原型为：

```
int itime_time_get    (struct itime *p_this, struct tm *p_tm);
```

此时，struct itime 结构体类型是应用可见的，应用程序可以看到其中的成员、函数指针定义等（虽然应用程序并不会直接访问这些成员）。

在添加系统中间层之后，应用程序仅需指定其所使用的服务 ID，系统中间层再根据 ID 找到相应的服务，因此，可以为上层应用提供一个新的接口，其原型如下：

```
int time_get (int id, struct tm *p_tm);
```

该函数的实现过程为：首先，通过 ID 找到相应的 RTC 服务，即 struct itime *类型的指针，然后再调用 itime_time_get()获取实际的时间，最后存储至 p_tm 并返回给上层应用。示意性伪代码详见程序清单 4.1。

程序清单 4.1　获取时间伪代码

```
1     int time_get (int id, struct tm *p_tm)
2     {
3           struct itime *p_time = get_rtc_serv_by_id(id);
4           if (p_time == NULL) {          // 找不到这个 ID 对应的服务
5                 return -1;
```

```
6           }
7           return itime_time_get(p_time, p_tm);
8       }
```

由此可见，对于上层应用来讲，只会“看”到 time_get()接口，不会“看”到 struct itime 结构体。struct itime 这个抽象类的定义，完全封装到了系统内部，对应用程序实现了隐藏。

如此一来，用户主程序不再需要管理硬件的初始化以及获取“接口实现”，并将接口实现传递给应用程序。在程序清单 2.7 和程序清单 2.10 所示的主程序中，由于使用得器件不同，器件所对应的初始化函数不同，因此，主程序中初始化部分的代码是不同的。而使用新的软件结构后，主程序只需要指定应用程序使用的 ID 即可，范例程序详见程序清单 4.2。

程序清单 4.2　主程序范例程序

```
1       int main()
2       {
3           app_elec_watch (0);                                    // 启动“电子表”应用程序
4           while (1) {
5           }
6       }
```

由此可见，主程序中不再存在与具体器件相关的代码，换句话说，主程序不再直接依赖于底层硬件。此时，主程序、应用程序、接口类、实现类之间的关系详见图 4.6。

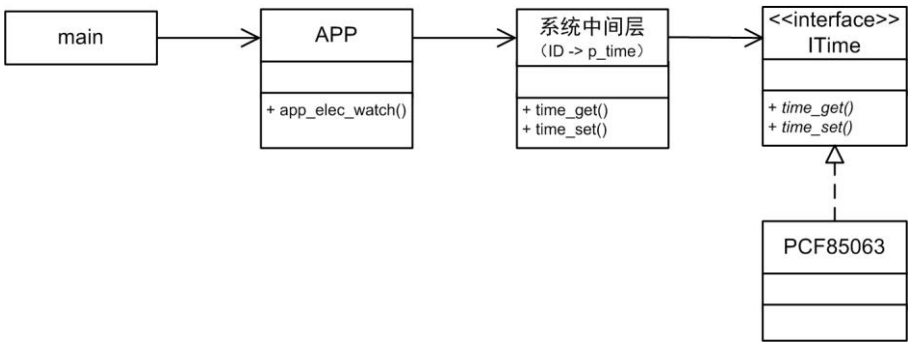


图 4.6　应用程序、接口与实现类之间的关系

图中，可以把系统中间层看作一个黑盒子，其实现了 ID 到 p_time 的转换，即通过 ID 获取到相应的 RTC 服务，然后根据应用程序的需要完成时间的获取或设置（为应用程序提供了上层接口：time_get()和 time_set()），中间层分离了应用程序与“接口”，使应用程序所需要了解的内容更少，“看不到”（本身也无需关心）底层硬件和接口设计的更多细节。

将其与图 2.2 比较可以发现，不再存在主程序到 PCF85063 的依赖线。这种情况下，当器件更换时，例如，从 PCF85063 更换至 DS1302，只要 ID 号不变，主程序就无需作任何改动。若 ID 号发生了变化，例如，并没有更换 PCF85063，而是新增了一个 DS1302，且主程序期望应用程序使用 DS1302，则相应的 ID 号就必须作对应的修改。

综上所述，板级组件描述了板上的硬件资源，明确使用到的芯片外设以及相关外设的配置。板级是平台的具体实例，需要使用到平台中定义的类，因此，通常情况下，板级组件会依赖于对应的嵌入式平台。例如，为了使用硬件板上的 UART 接口，就必须使用到平台提供的 UART 驱动。此外，板级组件可以辅助系统结构的优化，简化主程序的设计，使与硬件相关的初始化在进入用户主程序之前自动完成。

第5章 基本属性

本章导读

本章详细介绍了组件的基本属性，这些属性都是组件的重要特征，后续组件的管理，很大程度上都依赖于组件的这些属性。

5.1 名称（name）

名称是组件的标识名。名称必须是全局唯一的，即在整个组件系统中，组件的名称应互不相同。

组件命名的基本要求：

1. 名称由字母、数字、下划线（“_”）和减号（“-”）组成，不可使用其它特殊字符（例如：.%^&#$@）；
2. 首字符只能为字母；
3. 字母全部小写，分隔符使用下划线（“_”）或减号（“-”）；
4. 前缀通常由组件所处框架决定，例如，AWorks 框架下的所有组件，命名必须以“aw_”开头。

例如，几个常见组件的典型命名详见表 5.1。

表 5.1 组件命名举例

组件	名称
AWorks SHELL 组件	“aw_shell”
AWorks USB 组件	“aw_usb”
AWorks 基础工具组件	“aw_base”
AWorks LoRa 组件	“aw_lora”

一般情况下，组件名称反映了该组件中程序代码的命名空间，程序中需要对外的公共部分（公共接口、全局变量等）均以组件名作为命名空间。公共接口应该以组件名称作为前缀，紧接下划线，再加上具体的功能名。例如，shell 组件中有一个初始化接口，其命名可能为：“aw_shell_init()”。

部分特殊组件（主要是一些“大杂烩”组件）可以例外。例如，基础工具组件中包含了链表工具，则链表接口的命名前缀可以是“aw_list”，而并不一定为“aw_base_list”。此外，一些组件可能名称较长，当将其名称作为程序的命名空间时，可能被缩写，例如，modbus 组件中，modbus 可能被缩写为“mb”。但无论怎样，都应确保命名空间的唯一性，避免组件之间的程序命名产生冲突。

5.2 版本号（Version number）

版本号是组件版本的标识号，版本号可以帮助用户判断组件发布更新的进程。在传统的软件管理中，版本号没有明确的含义，发布者在发布新版本的软件时，往往根据“变化大小”，随意增加版本号，而“变化大小”是一个非常模糊的概念，对其的理解因人而异。这就使得版本号逐渐发展为几个数字和字符的随意组合，没有包含更多的“语义”，用户只能根据版本号的大小判定软件的新旧，但新老版本之间的差异有多大，能否兼容等，并不能直接从版

本号中看出。随着版本数目的增加，各版本之间的关系无法从版本号中获得，版本的管理将变得十分混乱。

例如，应用程序使用到了某个组件，假定原应用程序使用的组件版本为 1.0.0，现在该组件的版本更新至了 1.8.2，可能涉及到大量的功能增加、性能优化等，应用程序发现版本号变化比较大，考虑将应用程序更新至最新版本。从兼容性考虑，可能有以下两种情况：

1. 兼容。应用程序无需作任何修改，直接更新组件，重新构建应用程序即可，例如，若组件以静态库的形式提供，则只需要更新库文件，并重新编译生成应用程序即可，这种情况下，工作量很小；
2. 不兼容。应用程序代码需要修改，例如组件中的某些接口修改了（原型、参数等），则应用程序需要修改代码，将使用旧接口的代码修改为使用新接口。由于涉及代码的修改，应用程序修改后需要做全面的测试，避免因改动产生问题。如此一来，工作量可能会很大。

若没有为版本号约定语义，则无法直接从版本号中获取到兼容性信息，此时，用户很难直接评估更新应用程序所需的代价。并且，实际应用程序往往不只使用到了一个组件，组件可能会很多，此时，将更难评估。这将使得用户很难抉择，该不该更新，能不能更新，通常情况下，最终都会导致“不敢更新”。

但是，在部分情况下又不得不更新。例如，一个组件往往会被其他组件所依赖，系统中的多个组件可能依赖了同一个组件的不同版本。例如，一个系统中，使用到了三个组件：A、B、C，它们都依赖了组件 D，但依赖的是不同版本，分别依赖了版本 1、版本 2 和版本 3，示意图详见图 5.1。

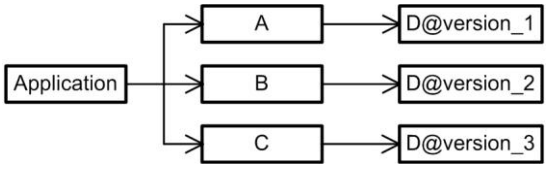


图 5.1　系统中多个组件依赖了同一组件的不同版本

通常情况下，同一组件的不同版本并不能加入到同一个工程中进行编译、链接。在实际应用中，系统最终只能使用一个明确的版本。

若不知道组件 D 各个版本之间的兼容性，则用户在确定一个使用的版本后，需要将其它未依赖该版本的组件更新至依赖同一版本。例如，用户最终决定使用版本 3，则必须更新组件 A 和组件 B，使它们均依赖于版本 3，在这个过程中，必然涉及到大量的测试以及组件代码的修改。当软件系统十分庞大时，类似情况会越来越多，需要花费大量的时间处理软件版本的抉择与更新，如此发展下去，软件的更新将会逐渐变成一种“噩梦”。

若用户可以知道各个版本之间的兼容性，则可以很快的作出最优选择。例如，三个版本都是向下兼容的：即版本 3 兼容版本 2，版本 2 兼容版本 1，则选择使用最新版本即可，组件本身不需要做任何改动。若仅部分版本兼容，则根据实际情况，更新部分组件即可。例如，版本 3 兼容版本 2，但版本 2 不兼容版本 1，则组件 B 和组件 C 不需要作任何改动，仅需将组件 A 更新至使用版本 3 即可。

综上所述，为了规范版本号的使用，避免出现版本号混乱不堪的局面，按照“语义化版本控制规范”，定义了版本号的格式及其更新方式，为版本号赋予了既定的“语义”，使版本号及其更新方式包含了更多的信息：相邻版本间的底层代码和修改内容，版本优先层级、版本兼容性等，后文会逐一介绍各项规则。

5.2.1 版本号的基本形式

版本号的基本形式（标准版本号）为：major.minor.patch（记为 X.Y.Z），即由主版本号、次版本号和修订号三部分组成，且各部分之间使用句点（“.”）分割，若版本号为 1.2.4，则表明主版本号为 1，次版本号为 2，修订号为 4。

版本号的设定与更新必须遵守如下规范：

(1) 在版本号中，主版本号、次版本号和修订号必须存在，缺一不可；

(2) 主版本号、次版本号和修订号均为非负整数（≥0）；

(3) 数字使用十进制表示，且禁止在数字前补 0，比如，1.01.02 是非法版本号；

(4) 版本号更新时，各部分必须按照数值顺序递增，合法的版本发布顺序：1.9.1 -> 1.10.0 -> 1.11.0 -> 2.0.0；

(5) 任何版本号的软件在发布之后，都禁止改变该版本软件的内容，任何修改都必须以新版本再行发布；

(6) 主版本号为零（0.y.z）的软件处于开发初始阶段，一切都可能随时被改变，这样的公共 API 不应该被视为稳定版；

(7) 1.0.0 视为第一个正式版，用于界定公共 API 的形成。在这一版本之后，所有的版本号更新都应基于公共 API 及其修改内容，接下来的三条规则分别描述了各部分的更新规则；

(8) 主版本号（X.y.z | x > 0）在有任何不兼容的修改被加入公共 API 时递增，即做了不兼容的 API 修改，例如，接口原型变化（参数增减）、接口删除或接口语义发生重大变化等，每当主版本号递增时，次版本号和修订号必须归零；

(9) 次版本号（x.Y.z | x > 0）在有向下兼容的新功能出现时递增。任何公共 API 的功能被标记为弃用时必须递增，在内部程序有大量改进时也可以递增，每当次版本号递增时，修订号必须归零；

(10) 修订号 Z（x.y.Z | x > 0）在只做了向下兼容的问题修正时才递增，这里的修正指的是针对不正确结果而进行的内部修改，可以理解为 BUG 修复，注意，只进行了 BUG 修复时才更新修订号，若同时有功能性新增或不兼容的 API 变化，则应按照规则更新次版本号或主版本号，修订号随之归零。

通过以上各个规则的描述可知，主版本号增加后，向下是不再兼容的，使用老版本的应用程序升级至使用新版本时，需要重新适配、测试。而次版本号和修订号的增加可以做到向下兼容。若版本号严格按照上述规则进行更新，则通过版本号就可以很容易的判定各版本软件之间的兼容性。

5.2.2 先行版本号

当有重大功能更新或较大改动时，若不能完全保证这个版本的稳定性，则可以在发布正式版之前，发布一系列“先行版本”（pre-release version）。在使用一段时间并测试稳定后，再发布稳定的正式版本。先行版本号添加到“主版本号.次版本号.修订号”后面，使用连接号（“-”）与前面的版本号分割。

先行版本号本身由一个或多个标识符组成，标识符仅可由 ASCII 码中的英文字母（a-z 或 A-Z）、数字（0-9）和连接号（-）组成，不能使用其它字符（比如下划线），当存在多个标识符时，使用句点（“.”）分割各标识符，比如：alpha、alpha.1、alpha.2、beta.0、beta.1、rc.1、rc-123 等。一些带先行版本号的完整版本号范例有：1.0.0-alpha、1.0.0-alpha.1、1.0.0-0.3.7、1.0.0-beta.0、1.0.0-rc.0、1.0.0-rc.1、1.0.0-rc-123。

1. 基本规则

先行版本号的设定与更新必须遵守如下规范：

(1) 在版本号中，先行版本号是可选的，可以不存在先行版本号；

(2) 若存在先行版本号，则不能为空，即不能在连接号（“-”）后留空，比如：1.0.0-；

(3) 数字型的标识符使用十进制表示，且禁止在数字前补 0，比如：1.0.0-0.01.07 是非法版本号；

(4) 当一个标识符中同时存在数字和非数字时，如 1.0.1-alpha.3da，标识符“3da”视为非数字标识符，非数字标识符可以使用字符 0 作为前缀，比如：1.0.1-alpha.0911da3 是合法版本号；

(5) 从时间上看，先行版本发布于对应稳定版本之前，例如，一个典型的版本发布顺序：1.0.0-alpha -> 1.0.0-beta -> 1.0.0-rc -> 1.0.0；

(6) 被标上先行版本号则表示这个版本并非稳定版本，而且可能无法达到兼容的需求，即 1.1.0-alpha 不能视为稳定的 1.1.0 版本，其不一定能够兼容 1.0.0 版本。

2. 先行版本号的一般形式

先行版本号并没有像“版本号基本形式”那样，定义为 X.Y.Z 的固定格式，其本身的内容形式没有强制规定，但一般来讲，使用得最多的有 3 类：

- alpha

内部版本，具有多个内部版本时，可以使用 alpha.0、alpha.1、alpha.2、alpha.3……后面的数字越大，表明版本越新。

- beta

公测版本，具有多个公测版本时，可以使用 beta.0、beta.1、beta.2、beta.3……

- rc

即 Release candiate，正式版本的候选版本，最接近正式版本，类似地，可以使用 rc.0、rc.1 区分多个 RC 版本。

5.2.3 编译信息

编译信息（Build metadata）可以是一些附加的标注信息，如时间、SHA 值等，与版本号之间使用加号（“+”）连接（若存在先行版本号，则编译信息应位于先行版本号之后）。

编译信息同样由一个或多个标识符组成，且标识符仅可由 ASCII 码中的英文字母（a-z 或 A-Z）、数字（0-9）和连接号（-）组成，不能使用其它字符（比如下划线），当存在多个标识符时，使用句点（“.”）分割各标识符，几个带编译信息的版本号范例为：1.0.0-alpha+001、1.0.0+20130313144700、1.0.0-beta+exp.sha.5114f85。

编译信息的添加应遵循如下规则：

(1) 编译信息是可选的，版本号可以不附带编译信息；

(2) 编译信息若存在，则不能为空，即不能在加号（“+”）后留空，错误范例：1.0.0-alpha+；

(3) 编译信息中的数字型标志符可以在前方补 0，例如，1.0.0-alpha+009 是合法版本号；

(4) 编译信息不参与版本比较，若两个版本号仅编译信息不同，则两个版本视为相同等级，不存在孰新孰旧的概念。

5.2.4 版本的优先层级判定

所谓“优先层级”，指的是不同版本在排序时如何比较判断，以确定版本发布的先后顺

序，优先层级越高，版本越新。参与比较的主要有 4 部分内容：主版本号、次版本号、修订号以及先行版本号。注意，编译信息不参与版本号的比较。

比较方法：从左到右，依次比较每个标识符号，直到发现不同的部分，不同部分的差异即可决定优先层级，发现第一处不同之后，无需再继续比较余下的部分。

1. 主版本号

当主版本号不同时，以主版本号的数值进行比较，数值越大，优先层级越高，例如：1.5.0 < 2.0.0。

2. 次版本号

当主版本号相同，而次版本号不同时，以次版本号的数值进行比较，数值越大，优先层级越高，例如：1.4.0 < 1.5.0。

3. 修订号

当主版本号、次版本号相同，而修订号不同时，以修订号的数值进行比较，数值越大，优先层级越高，例如：2.1.0 < 2.1.1。

4. 先行版本号

当主版本号、次版本号及修订号都相同时，具有先行版本号的优先层级低于对应的稳定版本，例如：1.0.0-alpha < 1.0.0。通过“先行版本号”的字面意思也可以理解，所谓“先行”，即在发布对应正式版之前，先发行的一些过渡版本。

若两个先行版本号对应了同一稳定版本，即具有相同主版本号、次版本号及修订号的两个先行版本号，则其优先层级通过从左至右的每个被句点分隔的标识符号来比较，直到找到一个差异值后决定，差异部分的比较规则如下：

(1) 只有数字的标志符号以数值高低比较，例如：1.0.0-beta.2 < 1.0.0-beta.11（2 < 11）；

(2) 有字母或连接号时则逐字以 ASCII 的排序来比较，例如：1.1.0-beta <1.1.0-rc（b < r）；

(3) 数字标识符比非数字标识符优先层级低，例如：1.0.0-rc.12< 1.0.0-rc.ba（12 < ba）；

(4) 标识符同时存在数字和非数字时，整个标识符视为非数字标识符，例如，1.1.0-alpha.3174632 < 1.1.0-alpha.0911da3 （0911da3 存在非数字，整体视为非数字标识符，而 3174632 为纯数字，数字标志符号比非数字标识符号优先层级低）；

(5) 若开头的标识符号完全相同，仅以句点分割的栏位数目不同，则栏位比较多的先行版本号优先层级更高，例如：1.0.0-alpha < 1.0.0-alpha.1（栏位数目 1 < 栏位数目 2）。

(6) 无论怎样，所有先行版本号的优先层级都低于对应的稳定版本，一个完整的比较范例：1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0- rc.1 < 1.0.0。

例如，表 5.2 中展示了 AWorks 中某一组件的版本发布记录，当前最新版本为 2.1.0。

表 5.2 版本发布举例

版本号	发布时间
2.1.0	2 天前
2.0.1	10 天前
2.0.0	17 天前
2.0.0-rc.0	19 天前
2.0.0-beta	24 天前

续上表

版本号	发布时间
2.0.0-alpha.2	1 个月前
2.0.0-alpha.1	2 个月前
2.0.0-alpha.0	2 个月前
1.3.0	2 个月前
1.3.0-rc	2 个月前
1.3.0-bata	2 个月前
1.3.0-alpha	2 个月前
1.2.0	5 个月前
1.1.1	5 个月前
1.1.0	5 个月前
1.1.0-rc	5 个月前
1.1.0-beta.1	5 个月前
1.1.0-beta	5 个月前
1.0.0	7 个月前

由表可见，在发布 2.0.0 正式版之前，发布了很多先行版本，可以看出 2.0.0 版本存在着重大的更新，相对来讲，发布得非常谨慎。

5.3 依赖（Dependency）

一个组件要正常工作，可能需要依赖一些其它的组件。例如，AWorks 的基础工具组件提供了链表服务，若某一组件需要使用链表，则可以直接依赖 AWorks 的基础工具组件，进而使用其中的链表服务。

除依赖其它组件之外，组件可能对其运行的环境也有所要求，例如，在日常应用中，很多软件都有对操作系统的依赖，部分软件可能只能运行在 Windows 上，而一些软件又只能运行在 Linux 上。这种关系本质上也是依赖，一种对环境的依赖。根据嵌入式软件的实际情况，定义了 3 种类型的环境依赖（特殊的组件依赖）：

- 框架（Framework）：组件仅可运行在特定的框架中，比如 AWorks；
- 平台（Platform）：组件仅可运行在特定的芯片平台中，比如 RT105x；
- 板级（Board）：组件仅可运行在特定的硬件板上，比如 EasyARM-RT1052。

下面，分别对组件依赖以及三种类型的环境依赖作进一步详细介绍。

5.3.1 组件依赖

1. 表示方法

组件依赖用于描述组件之间的依赖关系，依赖组件时，必须指定依赖的版本（依赖的版本可以是一个确定的版本，也可以是一个范围，表示该范围内的版本均可，后文会详细介绍版本范围的表示方法），依赖关系的示意图详见图 5.2。

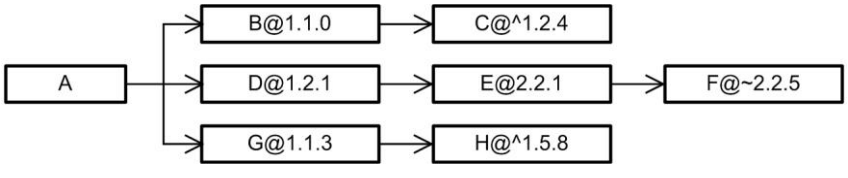


图 5.2 组件依赖示意图

依赖关系使用有向箭头表示，箭头指向所依赖的组件。组件名与版本号之间使用“@”符号连接，表示组件对应的版本。图中表示组件 A 依赖了 3 个组件：B、D、G，依赖的版本分别为 1.1.0、1.2.1、1.1.3。

在图 5.2 中，组件 A 依赖的组件均指定了确切的依赖版本。在实际应用中，除了依赖确定的版本外，还可以指定依赖版本的一个范围，例如，在组件 B 开发完成时，其依赖的组件 C 版本为 1.2.4（即基于该版本测试验证的），根据“语义化版本”的约定，只要主版本号不增加，则后续版本 1.x.y（x >= 2, y >= 4）均是向下兼容的，换句话说，组件 B 可以直接使用组件 C 的这些版本而无需作任何更新。这种情况下，可以指定其依赖的版本为：^1.2.4，即版本号前增加一个向上的小三角符号（“^”），其表示兼容这个版本的都可以，表示的语义即为：≥1.1.0 且 < 2.0.0。

另一种常见的指定兼容版本的方法是版本号前加一个波浪符（“~”），此时，表示修订号可以变化，但次版本号不能变化，例如，在图 5.2 中，组件 E 依赖于组件 F，组件 F 的版本表示为~2.2.5，则表示组件 E 可以使用组件 F 的版本范围为：≥2.2.5 且 < 2.3.0。

“^”和“~”这两个符号是使用得最为广泛的两个表示范围的符号，除此之外，也可以直接使用大于、小于等符号表示一个明确的范围，常用符号详见表 5.3。

表 5.3　常用范围符号

符号	具体含义	举例
<	小于某一指定版本	<1.2.7
<=	小于或等于某一指定版本	<=1.2.7
>	大于某一指定版本	>1.2.7
>=	大于等于某一指定版本	>=1.2.7
==	指定一个确定的版本，此符号可省略，也可写作“=	=1.2.7 ==1.2.7 1.2.7
!=	不为某一版本	!=1.2.7
*	所有版本	任意版本
^	确保版本兼容性时，次版本号、修订号均可变化。一个例外情况是，非正式版（0.y.z）是开发过程中的版本，视为不稳定的，版本随时可能改变，此时，若次版本号不为 0，则允许修订号变化；否则，修订号也不允许变化。 具体规则：在满足>=x.y.z 条件时，从左至右，第一个不为 0 的版本号保持不变，后续版本号可变。	^1.2.3 表示：>=1.2.3, <2.0.0 ^0.2.3 表示：>=0.2.3, <0.3.0 ^0.0.3 表示：>=0.0.3, <0.0.4
~	确保版本兼容性时，仅修订号可变化，即大致匹配某个版本。具体规则：在满足>=x.y.z 条件时，主版本号和次版本号保持不变，修订号可以变化。	~1.2.3 表示：>=1.2.3, <1.3.0

表 5.3 中的各个符号都可以表示一个版本范围，当存在多个版本范围时，可以使用逗号（“,”）连接多个范围，表示需要同时满足所有的范围。例如：“~1.2.7,!=1.2.8”表示 1.2.7 至 1.3.0（不含）的版本均可，但除 1.2.8 这个版本之外。

(1) 依赖版本中先行版本号的指定

在指定依赖版本时，若未指定先行版本号，则在选择合适的版本时，先行版本号不参与比较，例如，指定依赖的版本为“>=1.2.7”，则实际版本“1.2.7-alpha”也是满足该要求的，

因为在判断“1.2.7-alpha”这个版本时，会忽略先行版本号，视为版本“1.2.7”，而“1.2.7”是满足“>=1.2.7”这一条件的。

若需要先行版本号也参与比较，则必须在指定依赖版本时，同时指定先行版本号，例如，指定依赖的版本为“>=1.2.7-beta”，则版本“1.2.7-alpha”不属于该范围，而版本“1.2.7-rc”属于该范围。

若在指定依赖版本时，不期望指定一个具体的先行版本号，则可以将依赖版本的先行版本号留空，例如：“>=1.2.7-”，此时，“1.2.7-alpha”、“1.2.7-beta”均不再属于该范围，因为先行版本的优先层级均低于对应的稳定版本。

(2) 依赖版本中编译信息的指定

在介绍优先层级的判定时提到，编译信息并不参与优先级的比较，若依赖版本需要指定编译信息，则只能使用等于或不等于两种条件，例如：“==1.2.7+build1”，“!=1.2.7+build1”。其它判断条件均属于非法的，例如：“>=1.2.7+build1”。

2. 基本原则

为了确保系统的良性发展，组件依赖必须遵守一定的规约，下面重点介绍两个十分重要的原则：无依赖环原则和稳定依赖原则。

1) 无依赖环原则

无依赖环原则，即不应出现循环依赖，例如，A 依赖 B、B 依赖 C、C 又依赖 A，示意图详见图 5.3。

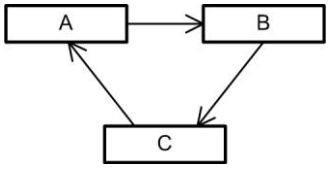


图 5.3 循环依赖

循环依赖会给实际项目带来诸多麻烦，一个依赖环中的组件相互依赖，致使任何组件依赖其中的一个组件，都将间接依赖环中的所有组件，极大的增加了组件之间的耦合度。循环依赖会使组件的独立维护工作变得十分困难，测试和发布的难度都将持续增加。

一种简单的解决方案是应用“依赖反转”策略，将其中一条依赖线反向，进而打破循环依赖。例如，对于图 5.3，可以将组件 B 依赖组件 C 反转为组件 C 依赖组件 B（在组件 B 中定义接口层即可实现，详见图 3.4），示意图详见图 5.4。

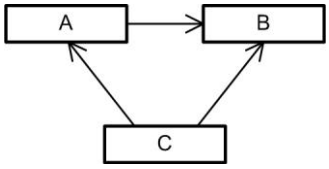


图 5.4 消除循环依赖

这种情况下，当外部组件依赖组件 B 时，将不会额外引入对组件 C 和组件 A 的依赖。在实际应用中，组件数目可能往往不止 3 个，依赖关系较为复杂，但无论怎样，都不建议出现依赖环，一个无依赖环的组件依赖示意图详见图 5.5。

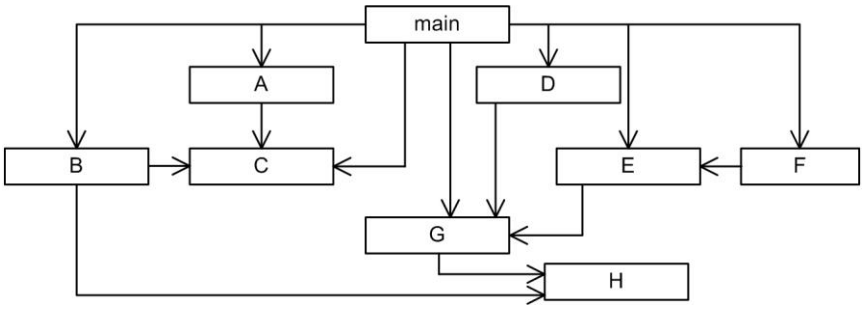


图 5.5　典型应用依赖关系图——无依赖环

图中展示了 9 个组件之间的依赖关系，依赖关系较为复杂，但不存在任何依赖环，是一种十分建议的设计。若设计没有考虑周全，使组件 H 依赖了组件 F，则将出现依赖环，示意图详见图 5.6，图中虚线框类的四个组件形成了依赖环。

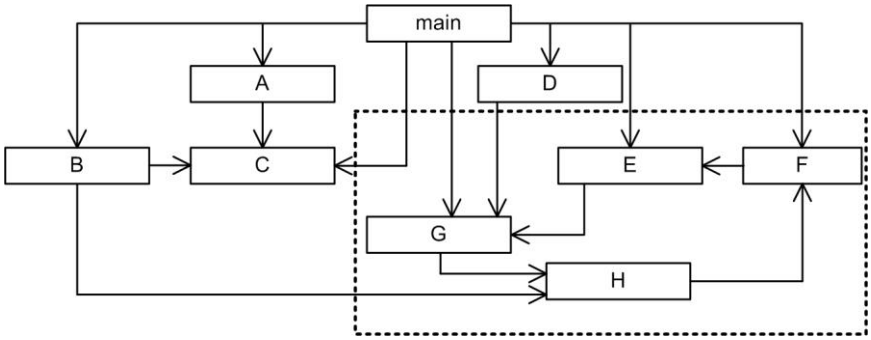


图 5.6　典型应用依赖关系图——存在依赖环

2)　稳定依赖原则

稳定依赖原则，即依赖关系应该指向更稳定的方向。越稳定的组件，被改动的可能性越小，例如，接口类组件等。如此一来，依赖它的组件就不至于因为依赖组件的修改而被迫修改。稳定依赖示意图详见图 5.7。

图 5.7　稳定依赖

随着依赖关系向更加稳定的方向发展，当系统变更时，箭头起点的组件由于不稳定性将更加可能发生变化，但是，这种变化对其所依赖的稳定组件不会产生任何影响，以此可以降低每次变更所付出的代价。

若依赖关系颠倒，稳定的组件依赖了不稳定的组件，那么，当不稳定的组件被修改时，稳定组件被迫也必须修改，最终使得稳定的组件也变得不稳定了。示意图详见图 5.8。

图 5.8　违反稳定依赖原则

随着依赖关系向更加易变的方向发展，当系统变更时，箭头末端的组件由于不稳定性将更加可能发生变化，此时，它的修改将影响其它所有直接或间接依赖它的组件。这将出现“牵一发而动全身”的典型现象，每次变更都需要付出很高的代价。

这个关系可能比较好理解，但在实际应用中，如何判定组件的稳定性呢？一般来讲，越抽象的东西越稳定，越具体的东西越不稳定。在 AWorks 中，一些接口是基于“功能”抽象

的，例如，LED 接口、ADC 接口等，它们与具体硬件无关，可以视为是非常稳定的，变更的可能性很小。而一些具体的事物，比如，具体的硬件、芯片型号等，都是可能随着时间的推移随时变更的，一种常见的情况是，产品升级时，需要更换性价比更高、性能更强、资源更多的芯片。

若遵守稳定依赖原则，由于依赖箭头指向更加稳定的方向，因此，最稳定的组件是处于箭头末端的组件，它有一个明显的特点：不再依赖其它任何组件，示意图详见图 5.9。

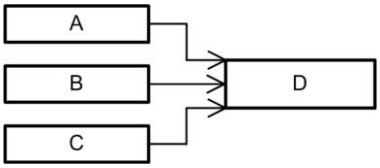


图 5.9　最稳定的组件

图中，组件 D 仅被其它组件依赖，但不依赖任何其它组件。同理，最不稳定的组件，处于箭头的起点，它的特点是：不被其它任何组件所依赖。

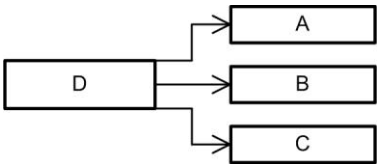


图 5.10　最不稳定的组件

图中，组件 D 仅依赖其它组件，但不被任何其它组件依赖。此时，组件 D 的修改将不影响任何其它组件，虽然其不稳定，但其修改时需要付出的代价却是很小的。由此可见，组件的稳定性与其依赖和被依赖的关系有关。若遵守稳定依赖原则，则越稳定的组件，被其它组件依赖的数目就越多，而依赖其它组件的数目越少。

5.3.2 框架依赖

在 4.4.1 节中，简要介绍了框架的概念，框架中通常包含了一些通用的定义。若某一组件使用到了框架提供的定义或服务（例如，AWorks 框架定义的标准错误号），则组件将依赖于该框架。框架依赖意味着该组件只能在其依赖的框架中运行。

部分组件可能不与任何框架相关，例如，使用标准 C 开发的“Hello World！”测试组件，示意代码详见程序清单 5.1。

程序清单 5.1　使用标准 C 开发的与框架无关的代码

```
1    int test_hello_world (void)
2    {
3        printf ("Hello World!\r\n");
4        return 0;
5    }
```

5.3.3 平台依赖

部分组件可能与具体运行的芯片平台相关，例如，某一组件是基于 RT105x 芯片开发的，且只能在该芯片上运行，则表明该组件依赖于 RT105x 平台。

对于绝大部分组件来讲，都应做到“跨平台复用”，因此，直接依赖于特定平台的组件

会很少，这部分组件往往与芯片底层有着密切关联。若对运行的平台没有要求，则无需指定平台依赖。

5.3.4 板级依赖

部分组件可能与具体的 Board 相关，例如，运行在特定 Board 上的 Demo 程序等，这部分组件只能在特定的 Board 上运行。若对运行的 Board 没有要求，则无需指定板级依赖。

5.4 组件仓库（Repository）

组件仓库指定了组件代码的存放位置。通常情况下，公司内部开发的组件都应使用 git 进行管理，并且存于 gitlab 服务器上。开发一个组件前，可先申请一个相应的仓库，以方便代码的管理和维护。例如，aw_usb 组件对应的 gitlab 地址为：git@gitlab:aworks2/aw_usb.git

5.5 简要描述（Description）

简要描述是对组件的概括性描述，以简要的描述组件的功能、使用方法、注意事项等，帮组用户更加高效的选择和使用组件。

5.6 License

License 是版权许可证，表明了软件版权，规定了使用、复制、发布软件的条款和条件。常用的开源许可证有 GPL、BSD、MIT、MPL、Apache 和 LGPL，它们之间的区别与联系详见图 5.11。

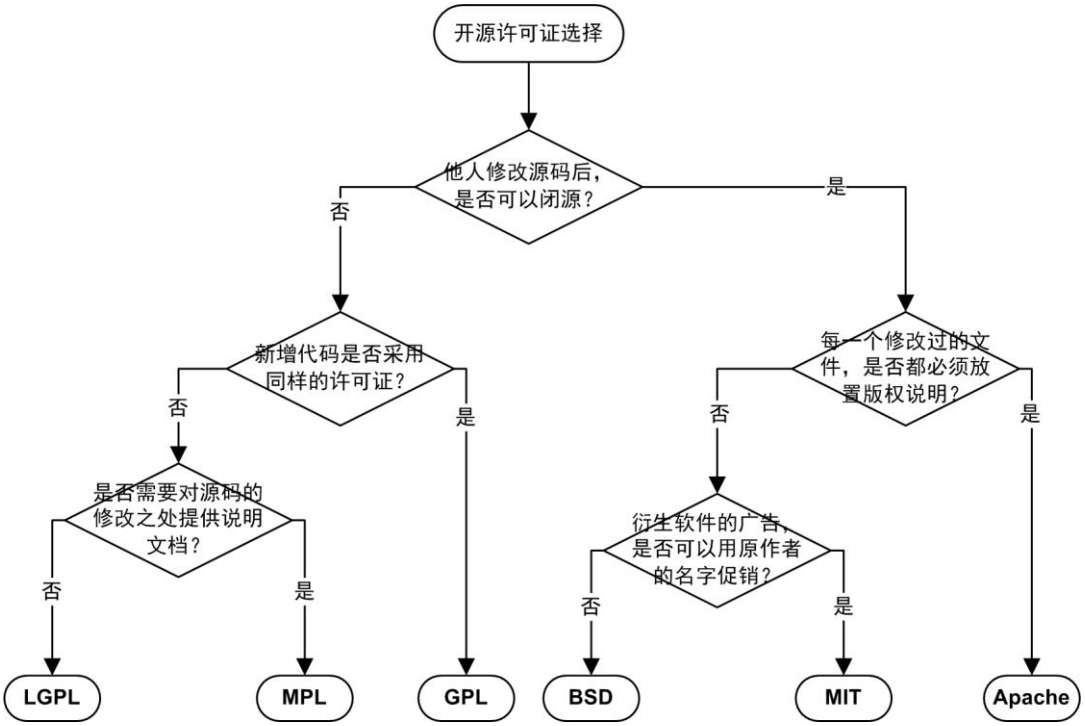


图 5.11　几种常见的开源许可证对比

下面分别对各个许可证作简要介绍，使读者对 License 有一定的了解，在确定一个软件选用何种 License 时，可以参考这些 License 的定义，也可以在这些 License 的基础上，根据实际情况，定义自己特有的 License。

5.6.1 BSD

BSD（Berkly Software Distribution）开源协议是一个给予使用者很大自由的协议。基本上使用者可以“为所欲为”，可以自由的使用、修改源代码，也可以将修改后的代码作为开源或者专有软件再发布。但“为所欲为”的前提是当你发布使用了 BSD 协议的代码，或者以 BSD 协议代码为基础做二次开发自己的产品时，需要满足三个条件：

(1) 如果再发布的产品中包含源代码，则在源代码中必须带有原来代码中的 BSD 协议；

(2) 如果再发布的只是二进制类库/软件，则需要在类库/软件的文档和版权声明中包含原来代码中的 BSD 协议；

(3) 不可以用开源代码的作者/机构名字和原来产品的名字做市场推广。

BSD 鼓励代码共享，但需要尊重代码作者的著作权。BSD 由于允许使用者修改和重新发布代码，也允许使用者在 BSD 代码上开发商业软件发布和销售，是一种对商业集成很友好的协议。很多公司和企业在选用开源产品的时候都首选 BSD 协议，因为可以完全控制这些第三方的代码，在必要的时候可以修改或者二次开发。

5.6.2 MIT

MIT（名称来源于麻省理工学院，即：Massachusetts Institute of Technology）和 BSD 一样，是一种非常宽范的许可协议。作者只想保留版权，而无任何其它限制，新软件以二进制和源代码的形式发布均可，甚至还可以用原作者的名义进行推广，前提条件仅仅是在软件发布时，需要包含原许可协议的声明。

5.6.3 Apache Licence 2.0

Apache Licence 是著名的非盈利开源组织 Apache 采用的协议。该协议和 BSD 类似，同样鼓励代码共享和尊重原作者的著作权，同样允许代码修改、再发布（作为开源或商业软件）。需要满足的条件也和 BSD 类似：

(1) 需要给代码的用户一份 Apache Licence；

(2) 如果你修改了代码，需要在被修改的文件中说明；

(3) 在衍生的代码中（修改和由源代码衍生的代码中）需要带有原来代码中的协议、商标、专利声明和其它原来作者规定需要包含的说明；

(4) 如果再发布的产品中包含一个 Notice 文件，则在 Notice 文件中需要带有 Apache Licence。你可以在 Notice 中增加自己的许可，但不可以表现为对 Apache Licence 构成更改。

Apache Licence 也是对商业应用友好的许可。使用者也可以在需要的时候修改代码来满足需要并作为开源或商业产品发布/销售。

5.6.4 GPL

GPL（General Public License）和 BSD、Apache Licence 等鼓励代码重用的许可证很不一样。GPL 的出发点是代码的开源使用和引用/修改/衍生代码的开源使用，不允许修改后和衍生的代码做为闭源的商业软件发布和销售。

GPL 协议的主要内容是只要在一个软件中使用（“使用”指类库引用，修改后的代码或者衍生代码）GPL 协议的产品，则该软件产品必须也采用 GPL 协议，既也必须是开源和免费的。这就是所谓的“传染性”。

人们熟知的 Linux 就是采用了 GPL。GPL 的“传染性”使得人们可以免费的使用各种

linux，包括商业公司的 linux 和 linux 上各种各样的由个人、组织以及商业软件公司开发的免费软件了。

由于 GPL 严格要求使用了 GPL 类库的软件产品必须使用 GPL 协议，对于使用 GPL 协议的开源代码，商业软件或者对代码有保密要求的部门就不适合采用 GPL 软件作为类库和二次开发的基础。

其它细节如再发布的时候需要伴随 GPL 协议等和 BSD/Apache 等类似。

5.6.5 LGPL

LGPL（Lesser General Public License）是为类库使用设计的开源协议。和 GPL 要求任何使用/修改/衍生至 GPL 类库的的软件都必须采用 GPL 协议不同。LGPL 允许商业软件通过类库引用的（link）方式使用 LGPL 类库而不需要开源商业软件的代码。这使得采用 LGPL 协议的开源代码可以被商业软件作为类库引用并发布和销售。

但是，如果对 LGPL 协议的代码进行修改，则所有修改的代码，涉及修改部分的额外代码都必须开源，且采用 LGPL 协议。LGPL 协议的开源代码很适合作为第三方类库被商业软件引用，但不适合希望以 LGPL 协议代码为基础，通过修改和衍生的方式做二次开发的商业软件采用。

GPL/LGPL 都保障原作者的知识产权，避免有人利用开源代码复制并开发类似的产品。

5.6.6 MPL

MPL（Mozilla Public License）是 1998 年初 Netscape 的 Mozilla 小组为其开源软件项目设计的软件许可证。MPL 许可证出现的最重要原因就是，Netscape 公司认为 GPL 许可证没有很好地平衡开发者对源代码的需求和他们利用源代码获得的利益。同著名的 GPL 许可证和 BSD 许可证相比，MPL 在许多权利与义务的约定方面与它们相同，但 MPL 还有以下几个显著的不同之处：

(1) MPL 虽然要求对于经 MPL 许可证发布的源代码的修改也要以 MPL 许可证的方式再许可出来，以保证其他人可以在 MPL 的条款下共享源代码。但是，在 MPL 许可证中对“发布”的定义是“以源代码方式发布的文件”，这就意味着 MPL 允许一个企业在自己已有的源代码库上加一个接口，除了接口程序的源代码以 MPL 许可证的形式对外许可外，源代码库中的源代码就可以不用 MPL 许可证的方式强制对外许可。这为借鉴别人的源代码用做自己商业软件开发的行为留了一个豁口；

(2) 允许被许可人将经过 MPL 许可证获得的源代码同自己其它类型的代码混合得到自己的软件程序；

(3) 对软件专利的态度，MPL 许可证不像 GPL 许可证那样明确表示反对软件专利，但是却明确要求源代码的提供者不能提供已经受专利保护的源代码（除非他本人是专利权人，并书面向公众免费许可这些源代码），也不能在将这些源代码以开放源代码许可证形式许可后再去申请与这些源代码有关的专利；

(4) 对源代码修改时，必须对源代码修改的时间和修改的方式有描述，并存于一个专门的文件中，使所有再发布者都可以得到这些信息。

5.7 贡献者（Contributors）

贡献者通常是直接参与开发的人员，可以是具体的开发者，也可以是团队、公司等。若使用 gitlab 管理组件源码，则相关信息可以直接从服务器上提取。

5.8 开发语言（Programming Language）

开发组件所使用的编程语言，例如：C、C++、JS、python2/3 等。

第6章 AXIO 工具

本章导读

AXIO 工具是 AWorks 生态圈中的一个工具，主要用于组件的管理、自动构建等。本章将对该工具作简要介绍。在后续章节的介绍中，均会涉及到 AXIO 工具的使用，在具体使用到的地方再对特定的使用方法作详细说明。

6.1 AXIO 简介

组件化理解起来相对容易，在一些小型项目中，人们可以简单地将一组.c 和.h 视为一个可重用的组件，人人都可以说自己的设计是组件化的。此时，完全依靠人力手动管理各个组件，还勉强行得通。

随着项目复杂度、项目数量的不断增加，继续依靠人力手动管理各个组件将使管理工作变得十分困难，组件的维护及更新、组件的复用都将很难有序处理。试想一下，成百上千个组件，各个组件有一系列不同的版本，组件之间还存在一定的依赖关系……完全依靠人力实现这些事务有序的管理，几乎不可能。此外，受情绪、状态、经验、技术水平、理解力等因素的影响，人是不稳定的，是极易犯错的，完全依靠人力手动管理的系统是不可靠的。

与“人力手动管理”对应的是：“工具自动管理”，这种情况下，人们要做的事情，仅仅是使用工具。由此可见，组件化要实际落地，除了需要制定一些规范（第 1 章和第 2 章均有提及）之外，还需要工具的支撑。就像在微软的组件对象模型中，通过一系列的规范和工具，才能保证组件二进制兼容、跨语言和跨进程调用，确保组件易获取、可重用、易重用、易维护和可替换。

在这样的大背景下，诞生了一个辅助 AWorks 生态圈软件设计的工具：AXIO，其具有自动构建、组件管理等功能。名称“AXIO”可以分为两部分理解：“AX”和“IO”。“AX”中的“X”可以看作数学领域中的未知数 X，代表任何东西，“AX”即代表了 AWorks 生态圈中一系列以字母 A 作为前缀的生态圈软件，比如：AWorks、AMetal、AWPI、AWTK 等，意味着该工具可以实现对它们的管理，同时，AXIO 本身以字母 A 开头，表明其也是 AWorks 生态圈中的一员。“IO”即 I/O，寓为“开放中创新”（InnovationintheOpen），“输入/输出”（Input/Output） 也是另一层意思，与 AXIO 的目标一致，输入一些东西（比如源码等），输出一些东西（比如 SDK、库文件等）。国外类似的工具命名为 platformio，所以 axio/AXIO 的命名既有内在的含义，也符合惯例。

AXIO 主要分为两部分：命令行工具 axio-cli；Web 网站。

6.2 命令行工具 axio-cli

6.2.1 简介

axio-cli 用于构建软件开发活动中的相关制品，如库文件、可执行文件、PDF 文档、API 手册、SDK 包等任何制品。一些典型的构建过程有：编译源码，生成相应的库文件发布；编译源码，生成相应的可执行程序（如.exe 文件）发布；打包相关代码、库文件、文档、工具，生成 SDK 发布。

axio-cli 控制了构建流程，即控制了组件的开发过程，包括源码管理、编译、打包等过程。在组件开发过程中，经常会使用到 axio-cli，以更加便捷的开发出符合规范的组件。

为了提高组件的稳定性，axio 还集成了 CI（Continuous Integration，持续集成）、单元

测试等辅助功能。

为了便于更好的对软件进行管理，axio-cli 将传统软件拆分为六个大类别：platform、board、 framework、component、driver、appliation。其中，platform、board、framework 的概念在 4.4 节中有详细介绍；component 为普通的一般组件；driver 为驱动类组件，例如，外围器件 PCF85063 芯片的驱动程序即为驱动类组件；application 为应用类组件，这里的应用指通用型应用，比如网速测试应用、Wi-Fi 模块自动配网应用等。axio 控制着每一类软件制品的生命周期。

axio-cli 将管理软件的基本单位称之为“包”（package，一个包含具体内容的文件夹、压缩包等），包中可以是组件源码，也可以是库文件、辅助工具（例如，GCC 编译器等）等。在 axio-cli 看来，所有软件制品的最终存在形式都是一个“包”。

组件的基本概念是：可以独立部署的最小单元。而“包”就是部署的形式，“包”可以非常方便的进行部署（上传网站，下载到本地等等）。

6.2.2 安装

要在本地使用命令行工具，必须先安装好 axio-cli。安装主要分为以下 3 步：

1. 安装 python

axio-cli 是使用 python2 编写的工具，因此，在使用 axio 前，应确保安装了 python2（推荐版本：python2.7.13）。python 的安装较为简单，可以访问 python 官网（https://www.python.org/）下载相应的安装包直接安装即可。

2. 获取 axio 安装包

axio 安装包可从 axio 的 web 网站上下载，也可联系我司获取。安装包的具体存在形式是一个压缩包，比如：axio-1.0.41a9.tar.gz（本质上也是一个组件，压缩包的名称为组件名+版本号，版本号遵循语义化版本的约定，实际版本以获取的安装包为准）。

注：若用户获取到的压缩包是.zip 格式，则应解压后使用，解压后会存在一个以.tar.gz 格式的压缩包。

3. 安装 axio

在 CMD 命令行窗口中输入如下命令：

```
> pip install axio-1.0.41a9.tar.gz
```

安装自动进行，安装成功完成后，会输出如下信息：

```
Successfully installed axio-1.0.41a9
```

安装完成后，在 CMD 命令窗口输入 axio，若打印出 axio 的相关信息，详见图 6.1，则表明 axio 安装成功。

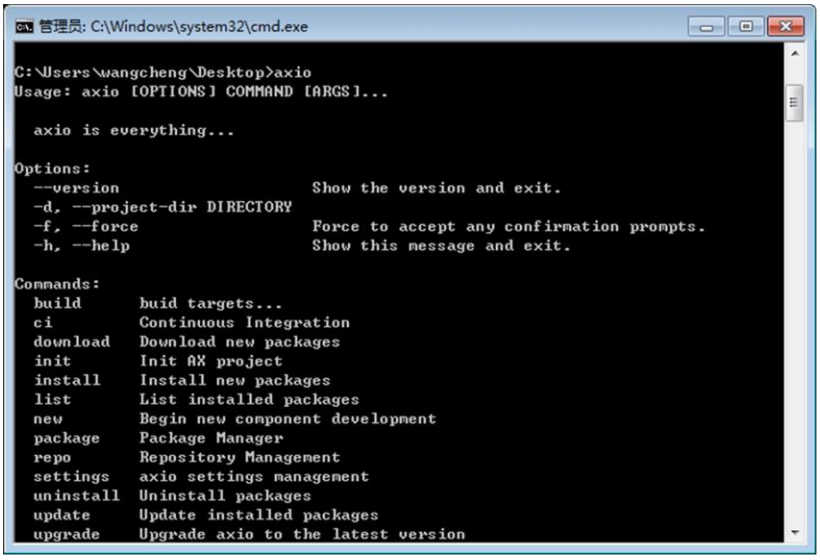


图 6.1　axio 帮助信息

由此可见，axio 命令行工具的基本用法为：

```
axio [选项] 命令 [参数]
```

其中，选项和参数是可选的。具体选项和命令在后文开发过程中，实际使用到某一功能时再详细介绍，以便读者可以结合示例更好的理解。

6.3　Web 网站

Web 网站是一个组件托管网站，集中管理 axio-cli 构建出的所有软件制品（如库文件、可执行文件、PDF 文档、API 手册、SDK 包等），并提供分类展示、过滤搜索、下载、问题反馈等功能，页面示意图详见图 6.2。

图 6.2　AXIO 首页

Web 网站同时具备权限管理功能，可以使不同的人员具有不同的权限，访问不同的软件制品。例如，开发人员可以访问源码类组件，而市场、销售人员仅需查看最终面向用户的软件，如 SDK、PDF 文档等。

第7章 组件的开发过程

本章导读

本章详细介绍了组件的开发过程，从目录的创建开始，步步深入，作了非常详细的介绍。包括代码编写、组件描述文件的编写等等。

7.1 创建组件目录

每个组件都应该存放于一个独立的文件夹内，文件夹的命名最好能够体现组件的功能及用途。通常情况下，可以直接将组件名作为文件夹的命名。为便于理解和学习基于 AXIO 实现组件化管理的具体流程及方法，这里以开发一个输出“hello”信息的应用程序作为范例。

首先，新建一个文件夹，并将文件夹命名为“hello”，以此作为组件的根目录，示意图详见图 7.1。

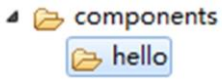


图 7.1 创建 hello 组件根目录

这里创建了 hello 组件的根目录，其位于一个名为 components 的文件夹内，这是一般建议的做法：将组件统一存放到一个确定的位置，方便本地组件的管理。当然，这并不是必须的，组件文件夹可以存放在任何位置。

该根目录下的所有文件即构成了一个包，后续即可使用 axio 对这个包进行管理。

7.2 编写代码

创建好组件根目录后，即可进行实际的开发工作，编写相关的代码。通常情况下，代码主要分为 2 个部分，源文件（如.c 文件）和头文件（如.h 文件）。

编写代码时，应严格遵循代码规范。假定 hello 组件非常简单，仅用于输出“hello”字符串，头文件和源文件的示意代码详见程序清单 7.1 和程序清单 7.2。

程序清单 7.1 hello.h 文件内容

```
1   #ifndef __HELLO_H
2   #define __HELLO_H
3
4   #ifdef __cplusplus
5   extern "C" {
6   #endif   /* __cplusplus   */
7
8   /**
9    *\brief 应用函数声明
10   */
11  void hello_info_output(void);
12
13  #ifdef __cplusplus
14  }
```

```
15  #endif   /* __cplusplus   */
16
17  #endif   /* __HELLO_H */
```

程序清单 7.2　hello.c 文件内容

```
1   #include <stdio.h>
2
3   void hello_info_output (void)
4   {
5       printf("hello\n");
6   }
```

实际上，整个 hello 组件对外仅提供了一个 API：hello_info_output()，用于输出一段字符串信息。这里也展示了一种惯例：组件内的 API 命名通常以组件名作为命名空间的一部分，即 hello 组件中的 API 以“hello_”作为前缀。

axio 对组件中源码的存放位置并没有强制规定，可以有极大的灵活性。以 hello 组件为例，其头文件和源文件的存放形式非常灵活，常见的几种存放形式详见图 7.2。

图 7.2　组件源码的存放位置

这种灵活性使得 axio 可以非常便捷的管理已经开发好的软件包（程序代码在使用 AXIO 工具前已经开发好），比如一些第三方软件（emWin、LwIP、UFFS 等）。即在使用 AXIO 管理这些软件包时，软件包的组织结构可以保持原样，无需作任何调整。

为便于叙述，后文以图 7.2 中的方式 3 为例进行介绍，在这种方式中，头文件和源文件分别存于 include 文件夹和 source 文件夹中。

7.3　添加组件描述文件

在第 5 章中详细介绍了组件的基本属性，在 axio 管理系统中，这些属性统一在组件描述文件中描述。组件描述文件的名称固定为：component.json，存放路径为组件根目录，示意图详见图 7.3。

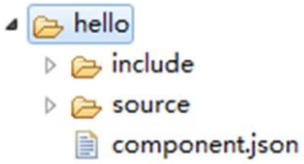


图 7.3　组件描述文件的存放位置

组件描述文件实质上是一个文本文件，在 Windows 环境中，可以直接新建一个记事本，然后将其名称（含文件类型）修改为 component.json；在 Linux 环境中，可以使用 touch 命令创建：

```
$ touch component.json
```

组件描述文件是可读性很高的 json 格式文件，一个极简的组件描述文件内容范例详见程序清单 7.3。

程序清单 7.3　组件描述文本内容范例

```
{
   "name": "hello",
   "version": "1.0.0",
}
```

其中描述了名称和版本号两个属性，名称为“hello”，版本为 1.0.0。

7.3.1 语法简介

组件描述文件本质上是一个 JSON 格式的文件。JSON（JavaScript Object Notation） 是 JavaScript 对象表示法，是一种轻量级的文本数据交换格式，类似于 XML。JSON 具有自我描述性，易于理解，其独立于编程语言。

简言之，JSON 就是用文本字符串表示对象，语法格式非常简洁，易于理解和解析。例如，使用一个文本表示程序员小明，内容范例详见程序清单 7.4。

程序清单 7.4　JSON 文本范例

```
{
      "name": "小明",
      "age": 24,
      "height": 1.65,
      "skills": ["C", "Java", "Python","C++"]
}
```

JSON 语法规则相对简单，主要有以下 4 点：

1) 数据在“名称/值”对中；

2) 数据由逗号分隔；

3) 花括号保存对象；

4) 方括号保存数组。

一个 JSON 文件通常直接描述了一个对象，例如，组件描述文件描述了组件对象，因此，JSON 文件中的内容，通常被一个花括号括起来（文件以“{”开头，以“}”结尾），表明所有内容表示了一个主体对象。

对象的数据使用“名称/值”对表示，具体表示方法为：

```
字段名称（必须使用双引号包围）：值
```

即以字段名称开始，后紧接冒号，然后是值。需要注意的是，在 JSON 中，字符串必须使用双引号包围，由于名称字段恒为字符串，因此，名称字段必须使用双引号包围。例如："name": "小明"。

JSON 中的值有 6 种类型，详见表 7.1。

表 7.1　JSON 中的值类型

值类型	描述	示例
数字	整数或浮点数，直接使用，无需双引号包围	"age": 24
字符串	必须使用双引号包围	"name": "小明"
逻辑值（布尔）	布尔类型的值，true 或者 false	"is_absent": true
数组	一组相同类型的值，使用方括号包围	"skills": ["C", "Java", "Python"]

对象	值的类型也可以是一个对象，对象使用花括号包围	"build_info" : {
		"change_log":"CHANGELOG.rst",
		"readme":"README.rst"
		}

null	表示空值	"phonenumber": null

特别地，数组中的值类型，可以是普通的数字、字符串、布尔值，也可以是数组和对象，数组中包含一组对象的示意文本如下：

```
{
    "employees": [
        { "firstName":"John" , "lastName":"Doe" },
        { "firstName":"Anna" , "lastName":"Smith" },
        { "firstName":"Peter" , "lastName":"Jones" }
    ]
}
```

更多 JSON 语法可以查阅 JSON 相关的文档，对于组件描述文件来讲，只要掌握“名称/值”对以及值的各种类型即可。

7.3.2 基本属性表示

组件描述文件是 JSON 格式的文本，该文本中的各个“名称/值”对即表示了组件的各个属性。每个“名称/值”对表示了一个属性。

通常情况下，组件描述文件中，仅需描述与组件构建相关的属性（如组件名称、版本号、依赖关系等）。其它属性与组件构建无关，在组件目录中的相关文件中体现即可，比如组件的 License 等。表 7.2 中列出了组件描述文件中常见的几种属性。

表 7.2　组件描述文件中的常见属性

属性	字段名	值类型	示例	默认值（字段省略时使用）
名称	"name"	字符串	"name": "hello"	目录名，component.json 文件所在目录的名字
版本号	"version"	字符串	"version": "1.0.0"	CHANGELOG 文件中的版本号，若无，则为 0.0.0
描述	"description"	字符串	"description": "AWorks Bus Lite"	null
依赖	"dependencies"	字符串数组	"dependencies": ["foo"]	空数组：[]
平台	"platforms"	字符串数组	"platforms":["windows_x86@^1"]	空数组：[]
框架	"frameworks"	字符串数组	"frameworks" : ["aworks"]	空数组：[]
板级	"boards"	字符串数组	"boards" : ["m105x-ev-board"]	空数组：[]

注意，在指定依赖的组件时（包括平台、框架、板级），可以在组件名后使用@符号连接依赖的版本。例如：

```
{
   "name": "zlib_adapter",
   "version": "1.0.0",
   "dependencies":["foo@^1"],
   "frameworks" : ["aworks@^1"]
}
```

当未指定依赖的版本时，默认将使用最新的版本。

7.4 打包组件

7.5 发布组件