

文章编号:1001-9081(2006)09-2236-04

嵌入式系统开发中的设计模式

刘 刚¹, 邵志清¹, 肖立中¹, 温盛军², 梁宏昊¹

(1. 华东理工大学 信息科学与技术学院, 上海 200237;

2. 中原工学院 电子信息学院, 河南 郑州 450007)

(lg1070@126.com)

摘 要:嵌入式系统的广泛应用促使嵌入式系统软件开发者采用设计模式等先进技术来改善现有的软件开发方法。针对嵌入式系统软件的实际特性,提出在嵌入式软件开发中应用设计模式技术。首先对嵌入式系统软件开发现状及设计模式技术进行分析,然后详细讨论了设计模式技术在嵌入式系统软件开发中的应用过程。最后通过一个例子来阐述该方法的优越性。

关键词:嵌入式系统; 设计模式; 软件复用

中图分类号: TP311 **文献标识码:** A

Design pattern in embedded system development

LIU Gang¹, SHAO Zhi-qing¹, XIAO Li-zhong¹, WEN Sheng-jun², LIANG Hong-hao¹

(1. Department of Computer Science and Engineering,

East China University of Science and Technology, Shanghai 200237, China;

2. College of Electronic and Information, Zhongyuan Institute of Technology, Zhengzhou Hunan 450007, China)

Abstract: With the widespread use of embedded systems, some advanced technologies of software engineering are in need to instruct the development of embedded software, such as design pattern. The main problem that has to be addressed in this context is how to put the technique into the design of an embedded application, taking into account the true nature of embedded systems. In this paper, an overview of the current embedded software design and the technology of design pattern were given first, and then how to apply the design pattern in the design of the embedded software was discussed. At last, an example was given to illustrate the approach.

Key words: embedded software; design pattern; software reuse

0 引言

嵌入式系统无疑是当前最热门最有发展前途的 IT 应用领域之一。嵌入式系统用在一些专用设备上,通常这些设备的硬件资源(如处理器、存储器等)非常有限,并且对成本很敏感。由于嵌入式系统自身的特点,它的设计给计算机工作者带来了巨大的挑战:一方面,系统中用到的硬件环境可能有多种(包括同构的和异构的嵌入式系统等);另一方面,系统中用到的软件又必须要求考虑高效性、可靠性、移动性、实时性、互操作性、同步性等因素^[1]。这些情况导致嵌入式系统软件开发的难度很大。

当前提高软件产品质量、缩短开发周期、降低开发成本是软件开发人员的迫切要求,软件产品的模块化和可复用性是满足这一要求的唯一方法。而面向对象的方法和设计模式^[2]的思想是当前实现软件模块化、提高软件可复用性的最优方法。设计模式提供了一种封装设计知识的方法,这些设计知识为标准的分布式软件开发问题提供解决方案。举个例子,模式对于描述重复出现的“微型结构”,如反应堆(Reactor)和主动对象(Active Object)十分有用,这些微型结

构是对一些已被证明可用于构建分布式通信软件的通用对象结构的抽象。但是,模式是文档化的抽象并不直接产生可复用代码。因此,有必要进行构架的创建和使用。通过集成成组的抽象类或模板类,并定义它们的协作的标准途径,构架为应用提供了可复用的软件构件。构架实例化设计模式族,以帮助开发者避免对通用分布式软件组件的昂贵的重新发明。其成果是“半完成”的应用框架,它可以通过继承和实例化构架中的可复用“积木”组件来进行定制。因为构架与关键的分布式编程任务(比如服务初始化、错误处理、流控制、事件多路分离、并发控制)紧密地集成在一起,复用的范围可以显著地大于使用传统函数库,或是通常的 OO(面向对象)类库和 STL 那样的模板库。在目前的嵌入式实时系统中采用面向对象和设计模式的方法进行系统软件的设计还有很多困难,最主要的是底层实时操作系统(RTOS)没有提供有力支持,即使系统软件勉强采用了这些方法,代码的模块化、可移植性、可复用性也难有提高。因此,需要在 RTOS 和系统软件之间加入一层,以屏蔽底层各类 RTOS 的差异,为上层应用提供一个统一的接口,作为开发上层应用的平台。然后,完全采用这些方法和成熟有效的设计模式,同时利用软件体系结构

收稿日期:2006-03-30; 修订日期:2006-06-22 **基金项目:**国家自然科学基金资助项目(60373075; 60473055)

作者简介:刘刚(1979-),男,湖北咸宁人,博士研究生,主要研究方向:软件开发与验证、嵌入式软件; 邵志清(1966-),男,江苏常熟人,教授,博士生导师,主要研究方向:软件开发、验证方法; 肖立中(1981-),男,山东寿光人,博士研究生,主要研究方向:信息安全; 温盛军(1979-),男,福建三明人,讲师,主要研究方向:信息安全、人工智能; 梁宏昊(1983-),男,河北石家庄人,硕士研究生,主要研究方向:软件开发与验证、嵌入式软件。

思想来构造适用于嵌入式系统软件开发的构件库。

在本文中,针对目前嵌入式系统的软件开发中重复性工作过多的现状,我们研究如何将设计模式方法应用到嵌入式系统软件的开发之中。

1 概念描述

嵌入式系统软件由于其开发层次较低,是直接在硬件上开发时间紧要、安全紧要、高可靠性的系统,传统上是C语言和汇编语言的天下。传统的软件工程技术着重软件的可移植性、可复用性、可伸缩性、易维护性、低成本,借助于一个良好的平台,快速交付,支持业务过程的快速变革的高适应性系统。软件工程使平台以上的系统日臻完善,发展了面向对象技术、构件技术乃至直接使用软件服务。“相信”实现服务的构件、支持构件实现的平台(操作系统、网络软件及数据库软件)都是最优的或比较优秀的。显然,这与嵌入式系统要处处操心每个软件元件的性能、可靠性、安全性走的是两条路,嵌入式系统开发者不相信臃肿的通用平台能解决他们在有限资源下做出高性能系统所遇到的各种问题。所以,嵌入式系统开发者一般不关心软件工程技术的最新进展,也很少在他们的工作中采用软件工程新技术。开发工作停留在提供完善的模块和子系统层次上,强耦合的模块开发过程还是现今嵌入式系统开发的主流。

但从系统的观点来看,嵌入式系统也是系统,特别是硬件技术的快速发展,嵌入式系统软件也有快速适应硬件型号升级问题,也有业务快速变更要求可伸缩、可修改、可复用问题。1975年,美国国防部开发的Ada^[3]语言系统就是为了在主机上开发适应变化的系统。通过交叉编译生成机载、弹载的目标系统嵌入到应用环境中,一切系统维护、伸缩均在Ada源代码级软件上完成。在某种意义上,今日的面向对象技术得益于Ada的数据抽象和模块封装。由于Ada为保证军用软件而管理过严,这导致面向对象技术的蓬勃发展体现在C++上。于是嵌入式系统自然转向C++。但对象化对实时性、可靠性并没有直接的好处,所以在小型嵌入式应用中,面向对象依然不是主流。尽管如此,有了C++这个桥梁,为嵌入式系统和当前软件工程技术合流打下了良好的基础。

面向对象封装带来的松耦合,使它成为分布式可伸缩系统的首选技术。为了参与网络上提供的实时服务,为了支持应用系统的快速变更,快速提供嵌入式产品,大幅度降低成本,提供标准化、构件化产品,嵌入式系统的对象化、构件化势在必行。

重新认识软件是有体系结构^[4]的。软件开发是一个软件过程,是近十年软件技术最重要的成就。过去的开发着眼于功能性能,无形中完成了一个软件过程,得到了一个有体系结构的产品。当时并不在意过程的好坏,体系结构是什么样的。通过验收能够满足使用的就是好产品。但情况并非如此顺利,使用中不断发现新的bug,用户一再要求改进。过程和体系结构在软件可适应性、易维护性上显现了极大的威力。不良的体系结构导致大量bug不断出现,甚至导致整个系统崩溃。不良的过程找不到可以分析的文档、数据,一个不大的改进每步都得从头做起,迟迟调不出来,大量窝工,成本飙升。于是,人们把软件开发从以功能性能为中心转而以软件体系结构为中心,精心改进软件过程,从而软件在改变之中依然

能保证符合需求的功能,并能保证高质量和低成本。

体系结构从总体上决定了软件可能达到的各项质量指标。软件过程通过精心安排软件制作的各项活动切实保证软件质量。该有的活动没有或不能及时到位都会造成低质量,高成本甚至项目失败。过程是产品质量最直接的保证。软件体系结构和软件过程相辅相成,一为软件内在质量保证,一为过程质量保证。所以,以体系结构为中心的最优过程开发已为软件世界广泛接受。

过程是以时间为进程的各项软件活动。开发活动完成都凝聚成有体系结构的软件,即它的后果(体系结构)是可分析的、可测试度量的。优秀的体系结构,正是软件设计的目标。于是人们研究分析体系结构并寻找实现设计的途径。

体系结构不外乎是实现各项功能、性能的元构件有机组合的集合。在面向对象的背景下,他们都是类对象,为了设计方便,把对象组成惯用的、不易出错的、可靠的、接近标准化的形式,则为模式(pattern)。最小的模式是几个类对象组成的构件,最大的模式是若干构件组成的子系统。

最早的设计模式是Liskov^[5]提出的七种基本模式,并提出了对象构成模式五条基本准则,为面向对象设计模式奠定了理论基础。随后的Gamma等人^[6]提出了常用的25种设计模式,为以模式设计软件体系结构提供了工程实践的基础。不过,Gamma等人讨论的模式作用域是局部的,我们把他们叫作机制式的设计模式(mechanistic design pattern),因为他们为对象的协作定义了各种机制。这种设计模式有较大的局限,只在单个的协作内。

当然,各种软件应用领域也都有本领域的模式,如果总结出来作为准规则,则本领域软件开发速度会成倍提高,而软件质量也较容易得到保证。所以对嵌入式领域来说,我们同样能构造出能适用于嵌入式领域的设计模式,来促进嵌入式系统软件的开发效率和质量。

2 应用思想

在开发复杂的嵌入式系统时,大部分困难是围绕在共享资源管理上,如何使管理既高效又稳定是必须解决的问题。本文所构造的模式重点放在高效的内存管理(把内存当作资源),以及一般软件资源(模型是对象)的稳定共享上,以确保整个系统的可调度能力。

在构造一个嵌入式系统软件时,有时候,需要设置一个对象集,在不同的时间,为了不同的目的执行。这时,我们可以构造如下模式:该模式在系统启动阶段创建一组对象池,以满足不同用户的需求。这个模式虽然不是用于解决动态内存分配问题,但可以应用于比静态内存分配模式更复杂的程序。

2.1 模式抽象

在许多嵌入式应用中,对象可能被大量的客户所请求。例如,在系统运行在一个复杂的不断变化的环境中时,有许多用户需要创建数据对象或消息对象。对这些对象的需求,即使我们可以限制对象需求的数目,预测一个最优的对象分配也是不可能的。在这种情况下,我们可以创建一个对象池,即创建对象,但不必初始化,一旦有需求请求,就可提供对象。客户程序可以请求对象,使用完后再放回对象池中。

2.2 模式结构

图1显示了此内存分配模式的结构图。参数化类

GenericPoolManager 实例化后创建所要去的特定的 PoolClass 类。被实例化类在图中是 ResourcePoolManager。一般情况下,在系统中有若干个这样的实例化池,但对于任何特定的 PoolClass 类型只做一个。每个池创建和管理一组对象,给他们分配空间,实现的方法是返回一个指针,引用或向客户返回一个指向已分配对象的句柄。然后再根据请求释放他们,即把释放后的对象放回到 freeObject 表中。在通常的情况下,系统在启动时创建一整套的池,且永不删除。这就避免了动态内存分配带来的问题,而且保留了许多好处。

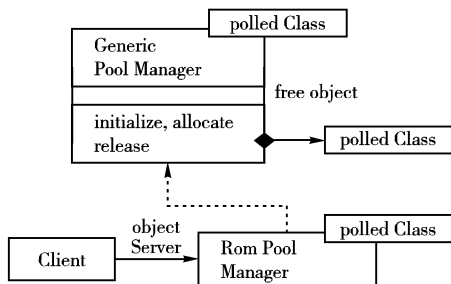


图1 内存分配模式的结构

2.3 模式中的各个角色

1) 客户 (client) 是系统中任意一个对象,它需要使用 ResourceClass 类的一个或多个对象。为了获取一个对象,它们调用 Resource Pool ::allocate(); 通过调用 Resource Pool ::release() 向对象池归还对象。

2) 类属池管理器 (Generic Pool Manager) 是参数化 (模板) 类,它使用形式参数 PoolClass 和 BufferSize 来指明实例化的对象的类和需要创建的数目。

3) 内存资源池管理器 (Resource Pool Manager) 是实例化的 Resource Pool Manager,在其中定义特定的类 resource Class。指定的对象数作为实际参数传入。在嵌入式系统有多个这样的实例,但每个 Resource 类只有一个。

2.4 模式分析

由于内存是在嵌入式系统启动时分配且永不删除的,不会有运行时作动态内存分配的时间的不确定性的问题和内存碎片化问题。这样,此模式特别适用的嵌入式系统如下:不同的客户需求需要一组公共对象,但这些对象不能在设计阶段确定如何分配,因为该模式允许在系统运行期间,根据需要分配对象。

这种内存资源分配模式是嵌入式系统管理内存分配的多种方案之一。这种模式可以与抽象工厂模式 (abstract factory pattern)^[6] 混合使用,以满足在不同环境下的资源分配。

3 模式实现和例子

3.1 模式实现

这个通用的模式是很容易实现的。为了使该模式在 C++ 中更容易实现,通常我们会重写 new 和 delete 操作符,以便使用管理各种 PoolClass 类型的内存资源管理器。下面我们给出了这个模式 C++ 实现的源代码。

```

#include <list>
using namespace std;
Class PoolEmpty {
};
template < class Resource, int nElements >
class GenericPool {

```

```

list < Resource * > freeList;
public:
GenericPool(void) {
    for (int j=0; j < nElements; j++) {
        freeList.push_back( new Resource);
    };
};
Resource * allocate( void) {
    Resource * R;
    if ( freeList.size() > 0) {
        R = freeList.begin();
        freeList.pop_front();
        return R
    } else {
        throw new PoolEmpty;
    };
};
void release( Resource * R) {
    freeList.push_back( R);
};
};
class BusMessage {
    string s;
};
int main( void)
{
    GenericPool < BusMessage, 1000 > busMessagePool;
    return 0;
}

```

3.2 实现例子

模式的构造和实现需要通过例子来证明其在开发中的有效性。下面我们通过一个嵌入式系统的开发来证明。

一个嵌入式温度测量系统由于要求每秒钟测量周围环境的实际温度 3 次,然后将此数值显示到显示器上。这个系统最大的特点是数据产生的比较频繁,如果采用传统的静态内存分配模式,将会很快造成资源不足的后果,所以我们采用上述的内存管理模式,即构造一个对象池。图 2 显示此模式的对象模型,该模型派生于图 1 的模式。

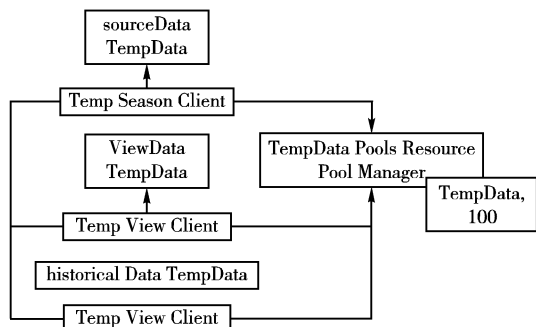


图2 对象模型结构

由图 2 可以看到,该对象模型中一共有四个对象。首先是 TempDataPool 对象的创建,然后依次创建 100 个 TempData 对象来存储测量温度值,这些 TempData 对象由 TempDataPool 来管理。该模型的其他三个对象是 Client 类的实例化对象: TempSensor 对象是一个温度计,每秒钟测量三次温度,同时分配一个 TempData 对象来存储信息。这个对象首先向 GUI 视图对象 TempView 报告温度,TempView 是一个 GUI 对象,它用来在显示器上向用户显示温度,一旦它已显示了值,会释放 TempData 对象,并把它放回资源池中,紧接着 TempSensor 对

象向 TempHistory 对象报告温度,TempHistory 对象管理最后 10 秒钟的温度历史数值,对于最初的 20 个数据,它不会删除向它传送数据的 TempData 对象,但当它收到新的数据值时,它会释放存储老的数据值的 TempData 对象。

为了描述模式应用的过程,我们给出了该模式运行时的顺序图,如图 3 所示。

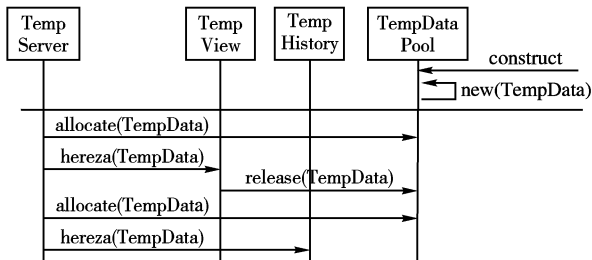


图3 模式顺序

从图3可以看出,Temp Data Pool 被构造时,会创建 100 个 TempData 对象实例。然后会根据这个系统功能的要求赋予这些对象不同的作用,但一旦功能完成,这些 TempData 对象会被释放,这样就保证了在一个嵌入式系统中有限的内存资源被合理使用。

4 结语

嵌入式系统软件面临着规模大、复杂性高而开发周期相对较短,必须具备可定制和演化的能力等挑战,因此一个借鉴

重用性高、扩展性和维护性好的设计模式思想来进行嵌入式系统软件开发显得非常必要。广义上讲,一个设计模式可以是一个算法、一种数据结构,但是在实践中,特别是在面向对象设计方法中,模式一般由一组协作的类构成,可以解决特定的问题。在嵌入式系统软件开发过程中,归纳并应用设计模式,可以较好的实现软件复用,提高软件生产率^[7]。同时便于将嵌入式系统理论研究结果进行形式化、半形式化的规范描述,在嵌入式系统理论和嵌入式系统软件实现之间建立起一座桥梁。

参考文献:

- [1] 沈连丰, 宋铁成, 叶芝慧, 等. 嵌入式系统及其开发应用[M]. 北京: 电子工业出版社, 2005. 11 - 12.
- [2] 孙鸣. 使用设计模式改善程序结构[R]. IBM developer Works, 2001.
- [3] 王振宇, 梁先忠. Ada 软件开发技术[M]. 北京: 国防工业出版社, 2001. 2 - 3.
- [4] BUSCHMANN F, MEUNIER R, ROHNERT H, et al. A System of Patterns: Pattern-Oriented Software Architecture[M]. New York, NY: John Wiley and Sons, 1996. 15 - 16.
- [5] LISKOV B, GUTTAG J. Abstraction and specification in program development[M]. McGraw-Hill, 1986. 5 - 9.
- [6] GAMMA E, HELM R, JOHNSON R, et al. Design Patterns: Elements of Reusable Object-Oriented Software[M]. MA: Addison-Wesley, 1995. 1 - 2.
- [7] JACOBSON I. Software Reuse - architecture and Organization for Business Success[M]. New York: ACM Press, 1997. 33 - 34.

(上接第 2235 页)

最后将二者拼接形成 T_{red} : $|T_{red}^1| = 15$, $|T_{red}^2| = 6$, $|T_{red}| = \max(|T_{red}^1|, |T_{red}^2|) = 15$, 这样我们仅用了 15 个测试用例就覆盖了全部的 I/O 关系, 相对于组合测试, 节约了 98% 的费用, 同聂长海算法的比较, 节约了 75% 的费用。

显然, 令 $m = \max(|R(y_1)|, \dots, |R(y_m)|)$, 无论是 T_{min} 还是 T_{red} , 都要满足 $\max(|R(y_1)|, \dots, |R(y_m)|) \leq T_{min}$ 或者 $\max(|R(y_1)|, \dots, |R(y_m)|) \leq T_{red}$, 所以找到的最优测试用例集越是与 $\max(|R(y_1)|, \dots, |R(y_m)|)$ 接近, 算法效果就应该越好。

虽然算法 1 和算法 3 也能得到最优解, 但由于其复杂性, 实际价值不高。现在来分析一下算法 5 的时间复杂度, 主要取决于第③步对每个 R^i 的处理。假设 $NY_k^i = \max(NY_j^i)$, 对每个 R^i 来说, 算法的时间复杂度大概为: $NY_k^i \cdot \sum_{j \neq ik}^{1 \leq j \leq l} (NY_j^i)$ 。假设 $NY_k = \max(NY_k^i)$, 则算法 5 的时间复杂度为: $\sum_{j \neq ik}^{1 \leq j \leq l} \left[(NY_k^i) \cdot \sum_{j \neq ik} (NY_j^i) \right]$, 因为 $NY_j^i \leq NY_k^i \leq NY_k$ 。因此最坏情况下, 我们算法的复杂度为 $(m-1)(NY_k)^2$ 。

4 结语

测试用例集的约简技术和优化, 可以大大地缩减测试计划, 降低测试成本。Schroeder 证明了基于 I/O 关系的测试用例集约简技术能够保证不降低检错能力, 但是他提出的 3 种

算法要么因为复杂度太高, 要么结果距最优解太大, 都难以用于实际中。聂长海将参数组合覆盖技术引入基于 I/O 关系的测试用例集约简, 但是他在约简的同时又增加了冗余信息致使结果难以达到最优解。

算法 5 充分利用了已知的 I/O 关系, 通过对 I/O 关系自身的特点(包含和关联)进行分析来对 I/O 关系约简和分组, 然后把每个相关组视为独立的 I/O 关系分别进行处理; 对每个输出所涉及的输入变量进行组合覆盖, 进而利用关联性把这些组合覆盖的测试数据进行水平拼接。最后再把各个相关组的结果进行水平拼接。实践证明, 这样产生的结果不仅最接近最优解, 并且时间复杂度也指数级下降, 得到了较大优化。

参考文献:

- [1] COHEN DM, DALAL SR, PARELIUS J, et al. The combinatorial design approach to automatic test generation[J]. IEEE Software, 1996, 13(5): 83 - 87.
- [2] COHEN DM, DALAL SR, FREDMAN ML, et al. The AETG system: An approach to testing based on combinatorial design[J]. IEEE Transactions on Software Engineering, 1997, 23(7): 437 - 444.
- [3] TAI KC, LEI Y. A test generation strategy for pairwise testing[J]. IEEE Trans. on Software Engineering, 2002, 28(1): 109 - 111.
- [4] SCHOEDER PJ. Black-box test reduction using input-output analysis[D]. Department of Computer Science Illinois Institute of Technology, Chicago, IL, USA, 2001.
- [5] 聂长海. 测试用例集约简技术研究[D]. 南京: 东南大学计算机科学与工程系, 2003.