BREW 平台中接口的设计与实现

张 茜,张继荣
(西安邮电学院,陕西 西安 710061)

摘 要:为了说明 BREW 中如何实现用 C 语言来模拟 C++中的面向对象的特性,实现接口的声明与实现的分离、对多个接口的支持和接口的易扩展性。通过实例,阐述了 BREW 通过虚拟函数表将接口与实现分离,使用 ISHELL 接口对多个接口支持及扩展。该方法与普通的 C 语言实现的接口相比较,修改接口而不会影响到应用程序,而接口有更好的可扩展性,更容易管理。

关键词:BREW;接口;面向对象

中图分类号:TP391.9 **文献标识码**:A **文章编号**:1674-7720(2012)05-0011-03

Design and implementation of interface in BREW platform

Zhang Qian, Zhang Jirong
(Xi'an Institute of Posts and Telecommunications, Xi'an 710061, China)

Abstract: In order to illustrate how to use C language to simulate the C++, object-oriented features in BREW, achieve the separation ofInterface declaration and implementation support for multiple interfaces and interfaces easy scalability. In this paper, for example, BREW virtual function table describes the interface and implementation separation. Use ISHELL this interface to support multiple interfaces and expansion. The methods compare with common C language interface, modify the interface without affecting the application. The interface has better scalability and easier management.

Key words: BREW; interface; object-oriented

如今,手机已不是简单的语音通信工具,它已逐渐发展成为数据业务开发应用的平台。在这种情况下,高通公司推出了一个新的 BREW (Binary Runtime Environment for Wireless)平台。BREW 平台的出现使手机像电脑一样,可以应用更多的第三方软件,满足人们不同的需求,为用户提供更多的服务。

在手机软件开发中,许多问题都需要使用 C++中面向对象的方法来实现,以提高代码的可重用性、程序的模块化及健壮性;为了满足用户对新数据应用的需求,移动设备制造商也希望不重新开发专有的软件平台,就可以快速提供新业务,以降低移动设备的技术门槛和产品上市门槛。

高通公司推出的 BREW 平台,为无线设备设计专门提供了一个高效的应用程序执行环境及开发平台。在 BREW 平台,用 C 语言开发应用程序可以达到 C++的设计效果,而不需要开发专用的软件平台,又提高了程序开发的效率和新业务开发的速度。

本文通过一个例子,阐述 BREW 中如何实现接口的声明和实现的分离来达到面向对象的特性,提高代码的可重用性、健壮性以及其可扩展性和易于管理的特性。

1 BREW 接口的实现

1.1 BREW 平台简介

BREW 的全称是无线二进制运行环境。从基本的层面而言,BREW 平台就是手持设备上嵌入式芯片操作系统的接口或抽象层,可以将它看作是 PC 环境下 Microsoft Windows 的 Win32 API。BREW 平台是一组用于本地执行而编译并链接的二进制库,优化后能使应用程序利用无线服务和资源,控制流出或流入应用程序的事件流,能根据相应的事件启动、停止、中止或恢复应用程序。BREW 执行环境在运行时可以发现应用程序和任何相关的扩展[1]。

1.2 嵌入式系统结构

图 1 是分散式系统结构图。从图 1 中可以看出,SDK 需要使用运行平台的接口声明来开发应用程序,运行平

台负责根据用户的输入启动应用程序，而应用程序则通过运行平台的接口调用运行平台的函数库来实现功能[2]。

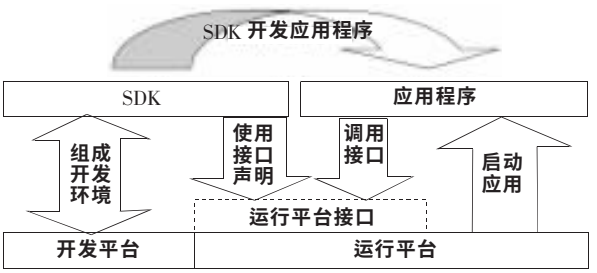

图 1 分散式系统结构图

假设程序运行到了需要调用平台函数的时候，由于当前的应用程序是开发者使用 SDK 开发的，就像平台不知道应用程序的地址一样，应用程序也不知道平台函数的地址，因此，所面临的问题就是怎样能够知道应用程序中所调用的平台函数的地址。虽然 SDK 中可以提供运行平台中每个函数的地址，但是因为平台会经常升级，导致每个函数的链接地址不固定，因此在平台升级时，SDK 和应用程序都需要同时升级。这样就不能实现“分散式”的升级了，这种程序的运行方式也就没有任何意义了。为解决分散式系统分散式升级的问题，BREW 提供了一个机制来解决应用程序调用函数的问题。

如果在开发过程中使用运行平台的接口声明，而在运行时应用程序使用真正的二进制接口，并在二进制层面调用接口函数[3]。则无论是 SDK 还是应用程序都与接口相关，解决这个问题的方式就是让接口和接口的实现之间分离。下面介绍 BREW 中的接口是如何实现的。

1.3 软件开发和 C 语言

在 C 语言中，C 语言库的开发商开发了一个算法来实现字符串的搜索，为了实现这个功能，软件厂商生成了一个头文件 FastString.h[4]，内容如下：

```
typedef struct _IFastString
{
char *m_pString;
}IFastString;
void IFastString_CreateObject (IFastString *IFastString, char *pStr);
                                        //创建目标字符串对象
void IFastString_Release(IFastString *IFastString);
                                        //释放目标字符串对象
int IFastString_GetLength(IFastString *pIFastString);
                                        //获取目标字符串长度
int IFastString_Search (IFastString *IFastString, char *pSearchStr);
                                        //查找字符串，返回偏移量
```

在 BREW 接口中共有 4 个接口：CreateObject、Release、GetLength、和 Search。CreateObject 用来创建 IFastString 接口，Release 用来释放接口的资源，GetLength 用来获取字符串长度，Search 用来查找字符串。一般地，这个库的使用者会将.lib 库链接到自己的工程中，通过接口声明的头文件来使用库中的函数。这样，库函数将成为客户应用程序中的一部分。

假设 FastString 库占用了 1 MB 的空间，如果 4 个程序中都调用了这个接口，则 FastString 接口将会占用 4 MB 的空间，也就是说有 3 MB 的空间浪费掉了。图 2 是多个程序调用 FastString 库的示意图。另外，如果库厂商发现接口有缺陷，又没有办法替换已经存在的缺陷代码，一旦 FastString 接口链接到代码中，就不可能在用户设备上替换这部分代码。因此，库厂商不得不重新为每个应用程序的开发者广播发布新的库文件，并希望他们重新编译程序来使用新的代码。这在嵌入式系统中是不可能的。因为在这里 FastString 的角色就是运行平台，不可能每个应用程序都包含一个运行平台。解决这个问题的一种技术是使用动态链接库技术将 FastString 包含起来。

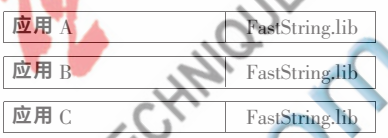

图 2 多个程序使用 FastString 库

1.4 动态链接库

动态链接库技术的典型应用是 Windows 操作系统中的动态链接库[4]。这种方法是将 FastString 源文件编译成特殊的二进制文件，并强迫 FastString 将所有的接口从二进制文件中引用出去，建立相应的引出表，以便于在运行时把每个接口的名字映射到对应的二进制接口地址上。同时还需要为使用者生成相应的引入库，使用者通过引入库可以获取每个接口的符号。当客户链接引入库时，这些符号信息会加入到当前的可执行文件中，运行时动态加载二进制库文件，并在执行时调用相应的程序。这样，即使多个程序调用 FastString 接口，FastString 代码也只需要一份，而且如果发现 FastString 代码有缺陷时，可以更新 FastString 二进制组件而不影响应用程序。图 3 是使用动态链接库时多个程序调用 FastString 库的示意图。

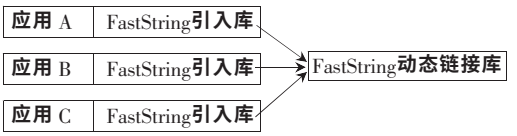

图 3 多个程序使用 FastString 库

1.5 虚拟函数表

但是在嵌入式系统中一般不支持动态链接库技术，因此在 BREW 中，采用虚拟函数表（VTBL）技术[4]实现接口和接口的分离。

下面是新版本的 FastString 的头文件：

```
typedef struct _IFastString IFastString;
typedef struct _IFastStringVtbl IFastStringVtbl;
typedef struct (*PFNCreateObject)
        (IFastString **ppIFastString, char *pStr);
```

```
struct _IFastString
{
struct IFastStringVtbl *pvt;
};
struct IFastStringVtbl
{
    void(*Release)(IFastString*pIFastString);
    int(*GetLength)(IFastString*pIFastString);
    int(*Search)(IFastString*pIFastString,char *pSearchStr);
};
 #define IFASTSTRING_Release (p)((IFastString*)p->pvt)->Release(p)          //释放目标字符串对象
 #define IFASTSTRING_GetLength(p)((IFastString*)p->pvt)->GetLength(p)       //获取目标字符串长度
 #define IFASTSTRING_Search (p)((IFastString*)p->pvt)->Search(p)            //查找字符串,返回偏移量
```

首先对 FastString 程序作一说明：在 FastString 头文件里定义了 IFastString 和 IFastStringVtbl 两个类型。IFastStringVtbl 类型是虚拟函数表类型，IFastString 中包含了指向虚拟函数表类型的指针。在接口定义时，使用((IFastString)p->pvt)调用虚拟函数表中的函数指针，这说明了如果要使用接口就必须先要提供 IFastString 的指针类型。可以看出，Release、GetLength 和 Find 已经实现了在 C 语言定义的接口和实现函数之间的分离。

源文件 FastString.c 如下：

```
#include"FastString.h"
 #include<string.h>
  typedef struct _CFastString
  {
IFastStringVtbl *pvt;              //指向虚拟函数表的指针
char *m_pString;                   //指向字符串的指针
int m_Len;                         //存储字符串的长度
}CFastString;
//函数声明
static void IFASTSTRING_Release(IFastString*pIFastString);
static void IFASTSTRING_GetLength(IFastString*pIFastString);
static void IFASTSTRING_Search(IFastString*pIFastString,char*pSearchStr);
 IFastStringVtbl gvtFastString = {IFASTSTRING_Release,
 IFASTSTRING_GetLength,
 IFASTSTRING_Search
};
void IFastString_CreateObject(...)
{...}
...
```

CFastString 结构体中有 IFastStringVtbl 类型的指针，而且这个指针在结构体的最顶部。另外还发现在 IFastString 结构体的最顶部也包含了 IFastStringVtbl 的指针，即 CFastString 是 IFastString 的超集。从而可以看出，CreateObject 函数中返回的 IFastString 指针其实是指向 CFastString 的指针。在 FastString.C 源文件中还定义了一个 IFastStringVtbl 的变量 gvtFastString，并为这个变量初始化成各个对应的函数，这个变量就是虚拟函数表。虚拟函数表的示意图如图 4 所示。

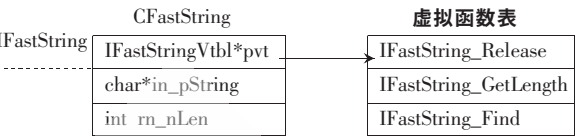

图 4 虚拟函数表

可以发现，除了 CreateObject 成员之外，其余的三个成员函数（IFastString_Release、IFastString_ GetLength、IFastString_Find）都添加到了虚拟函数表中，而且这个虚拟函数表还可以随着需求的增加而进行无限扩大，这样只用了一个函数 CreateObject 就实现了无限多个接口与实现之间的分离。

由于用户需要使用 CreateObject 来获取 IFastString 指针，因此将其与实现分离的方法是：因为应用程序的启动过程，对于一个程序，无论是由 main 函数或者其他函数作为启动函数，都允许启动时传递参数，因此把这个 CreateObject 函数作为参数传递给应用程序就可以了。至此应用程序、接口和实现之间已经分离了。

1.6 支持多个接口和接口的扩展性

实现了接口和实现之间的分离，但一个平台不会只有一个接口，还包含了其他用途的接口。但又不能把所有接口的 CreateObject 作为参数传递给应用程序启动函数，所以要对现有接口进行扩展来实现只要传递一个参数就能创建多个接口的功能。就像设计模式里面的工厂模式，创建一个专门的类用来创建别的类。因此，本设计采用增加一个称为 Shell 的接口来管理其他接口。

在 Shell 接口中，定义了 CreateInstance 接口函数，其代码如下：

```
static void IShell_CreateInstance (IShell *pIShell, int nClassID, void **ppObj, unsigned int nUserData)
{
...
switch(nClassID)
{
case CLASSID_FASTATRING:
    IFastString_CreateObject ((IFastString**)ppObj,(char*)nUserData);
    break;
case ...
    ...
...
    }
}
```

其作用是通过参数 nClassID 来创建指定的接口实例。IShell_CreateObject 函数用来创建 Shell 接口本身，然后再通过 Shell 接口来创建其他接口。也就是说，在应用程序启动时，先创建 Shell 接口，然后通过 IShell_CreateInstance 来创建其他接口。这样，不但实现了接口的管理工作，而且方便了接口的扩展。

本文通过一个例子介绍了 BREW 中接口的实现，说明使用虚拟函数表能够实现接口的声明与实现的分离，从而当平台升级或修改接口时不会影响应用程序，提高了 CPU 利用率，并使程序更加健壮。一个应用程序会使用很多接口，而 Shell 接口可以用来创建其他接口，使 BREW 接口有了很大的扩展性。使用 C 语言开发的 BREW 平台具有面向对象的封装性、继承性和多态性。

参考文献

[1] 陈秀寓. 基于 brew 平台的多态机制实现 [J]. 软件工程师，2010(2)：104-106.

[2] 赵建祥，高礼中.基于 BREW 的手机软件模块设计[J].仪仪器仪表用户，2009(5)：47-49.

[3] 费宁.BREW 实现机制深入分析[J].江苏通信技术，2006(4)：15-17.

[4] 焦玉海.深入 BREW 开发[EB/OL](2005-10-04)[2011-10-01]http://down.51cto.com/data/250317.

(收稿日期：2011-09-07)

作者简介：

张茜，女，1987 年生，硕士，主要研究方向：基于 BREW 平台的研究。