gobject:GObject对象系统

疯狂代码 http://CrazyCoder.cn/ ĵ:http:/CrazyCoder.cn/DeveloperUtil/Article54026.html

前言

大多数现代计算机语言都带有自己类型和对象系统并附带算法结构正象GLib提供基本类型和算法结构(如链表、哈希表等)样GObject对象系统提供了种灵活、可扩展、并容易映射(到其它语言)面向对象C语言框架它实质可以概括为:

个通用类型系统用来注册任意、轻便、单根继承、并能推导出任意深度结构类型界面它照顾组合对象定制、化和内存管理类结构保持对象父子关系处理这些类型动态实现也就是说这些类型实现是在运行时重置和卸载；

个基本类型实现集如整型枚举型和结构型等；

个基本对象体系的上基本对象类型实现例子--GObject基本类型；

个信号系统允许用户非常灵活自定义虚或重载对象思路方法并且能充当非常有效力通知机制；

个可扩展参数/变量体系支持所有能被用作处理对象属性或其它参数化类型基本类型

类型(GType)和对象(GObject)

GLib中最有特色是它对象系统--GObject 它是以Gtype为基础而实现套单根继承C语言面向对象框架

GType 是GLib 运行时类型认证和管理系统GType API 是GObject基础系统所以理解GType是理解GObject关键Gtype提供了注册和管理所有基本数据类型、用户定义对象和界面类型技术实现(注意:在运用任GType和GObject的前必需运行g_type_init来化类型系统)

为实现类型定制和注册这目所有类型必需是静态或动态这 2者的静态类型永远不能在运行时加载或卸载而动态类型则可以静态类型由g_type_register_创建通过GTypeInfo结构来取得类型特殊信息动态类型则由g_type_register_dynamic创建用GTypePlugin结构来取代GTypeInfo并且还包括g_type_plugin_*系列API这些注册通常只运行次目是取得它们返回专有类类型标识

还可以用g_type_register_fundamental来注册基础类型它同时需要GTypeInfo和GTypeFundamentalInfo两个结构事实上大多数情况下这是不必要系统预先定义基础类型是优于用户自定义

(本文重点介绍创建和使用静态类型)

对象定义

在GObject系统中对象由 3个部分组成:

对象ID标识(唯无符号长整型所有此类对象共同标识)；

对象类结构(唯结构型由对象所有例子共同拥有)；

对象例子(多个结构型对象具体实现)

基于GObject对象到底是什么样呢？下面是基于GObject简单对象 -- Boy定义代码:

```
/* boy.h */
#ndef __BOY_H__
# __BOY_H__
# <glib-object.h>
# BOY_TYPE (boy_get_type)
# BOY(obj) (G_TYPE_CHECK_INSTANCE_CAST((obj),BOY_TYPE,Boy))
typedef struct _Boy Boy;
typedef struct _BoyClass BoyClass;
struct _Boy {
GObject parent;
//
g age;
gchar *name;
void (*cry)(void);
};
struct _BoyClass {
GObjectClass parent_;
//
void (*boy_born)(void);
};
GType       boy_get_type(void);
Boy*    boy_(void);
      boy_get_age(Boy *boy);
```

```
void       boy__age(Boy *boy,  age);
char* boy_get_name(Boy *boy);
void       boy__name(Boy *boy, char *name);
Boy*    boy__with_name(gchar *name);
Boy*    boy__with_age(g age);
Boy*    boy__with_name_and_age(gchar *name, g age);
void    boy_info(Boy *boy);
#end /* __BOY_H__*/
```

这是段典型C语言头文件定义包括编译预处理宏定义数据结构定义和声明；首先要看是两个数据结构对象Boy和BoyClass

结构类型_Boy是Boy对象例子就是说我们每创建个Boy对象也就同时创建了个Boy结构Boy对象中parent表示此对象父类GObject系统中所有对象共同根都是GObject类所以这是必须；其它成员可以是公共这里包括表示年龄age表示名字name和表示思路方法指针cry外部代码可以操作或引用它们

结构类型_BoyClass是Boy对象类结构它是所有Boy对象例子所共有BoyClass中parent_是GObjectClass同GObject是所有对象共有根样GObejctClass是所有对象类结构根在BoyClass中我们还定义了个指针boy_born也就是说这指针也是所有Boy对象例子共有所有Boy例子都可以它；同样如果需要话你也可以在类结构中定义其它数据成员

其余定义包括 3种种是取得Boy对象类型IDboy_get_type这是必须有；另种是创建Boy对象例子boy_和boy__with_*这是非常清晰明了创建对象方式当然你也可以用g_object_来创建对象；第 3种是设定或取得Boy对象属性成员值boy_get_*和boy__*正常情况下这 3种都是个对象所必需另外个boy_info用来显示此对象当前状态

宏在GObject系统中用得相当广泛也相当重要这里我们定义了两个非常关键宏BOY_TYPE宏封装了boy_get_type可以直接取得并替代Boy对象ID标识；BOY(obj)宏是G_TYPE_CHECK_INSTANCE_CAST宏再次封装目是将个Gobject对象强制转换为Boy对象这在对象继承中十分关键也经常用到

对象实现

下面代码实现了上面Boy对象定义:

```
/* boy.c */
# "boy.h"
```

```
enum { BOY_BORN, LAST_SIGNAL };
 g boy_signals[LAST_SIGNAL] = { 0 };
 void boy_cry (void);
 void boy_born(void);
 void boy_init(Boy *boy);
 void boy__init(BoyClass *boy);
GType boy_get_type(void)
{
       GType boy_type = 0;
      (!boy_type)
      {
            const GTypeInfo boy_info = {
                (BoyClass),
                NULL,NULL,
                (GClassInitFunc)boy__init,
                NULL,NULL,
                (Boy),
                0,
                (GInstanceInitFunc)boy_init
           };
           boy_type = g_type_register_(G_TYPE_OBJECT,"Boy",&boy_info,0);
      }
       boy_type;
}
 void boy_init(Boy *boy)
{
      boy->age = 0;
      boy->name = "none";
      boy->cry = boy_cry;
}
 void boy__init(BoyClass *boy)
{
      boy->boy_born = boy_born;
      boy_signals[BOY_BORN] = g_signal_("boy_born",
                        BOY_TYPE,
                        G_SIGNAL_RUN_FIRST,
```

```
                        G_STRUCT_OFFSET(BoyClass,boy_born),
                        NULL,NULL,
                        g_cclosure_marshal_VOID__VOID,
                        G_TYPE_NONE, 0, NULL);
}
Boy *boy_(void)
{
      Boy *boy;
      boy = g_object_(BOY_TYPE, NULL);
      g_signal_emit(boy,boy_signals[BOY_BORN],0);
       boy;
}
 boy_get_age(Boy *boy)
{
       boy->age;
}
void boy__age(Boy *boy,  age)
{
      boy->age = age;
}
char *boy_get_name(Boy *boy)
{
       boy->name;
}
void boy__name(Boy *boy, char *name)
{
      boy->name = name;
}
Boy*    boy__with_name(gchar *name)
{
      Boy* boy;
      boy = boy_;
      boy__name(boy, name);
       boy;
}
Boy*    boy__with_age(g age)
```

```
{
	Boy* boy;
	boy = boy_;
	boy__age(boy, age);
	 boy;
}
Boy *boy__with_name_and_age(gchar *name, g age)
{
	Boy *boy;
	boy = boy_;
	boy__name(boy,name);
	boy__age(boy,age);
	 boy;
}
 void boy_cry (void)
{
	g_pr("The Boy is crying ......n");
}
 void boy_born(void)
{
	g_pr("Message : A boy was born .n");
}
void    boy_info(Boy *boy)
{
	g_pr("The Boy name is %sn", boy->name);
	g_pr("The Boy age is %dn", boy->age);
}
```

在这段代码中出现了实现Boy对象关键这是在Boy对象定义中未出现也是没必要出现就是两个化boy_init和boy__init它们分别用来化例子结构和类结构它们并不被在代码中明显关键是将其用宏转换为地址指针然后赋值到GTypeInfo结构中然后由GType系统自行处理同时将它们定义为静态也是非常必要

GTypeInfo结构中定义了对象类型信息包括以下内容:

包括类结构长度(必需即我们定义BoyClass结构长度)；

基础化(base initialization function可选)；

基础结束化(base finalization function可选)；

(以上两个可以对对象使用内存来做分配和释放操作使用时要用GBaseInitFunc和GBaseFinalizeFunc来转换为指针本例中均未用到故设为NULL)

类化(即我们这里boy__init用GInit宏来转换可选仅用于类和例子类型)；

类结束(可选)；

例子化(可选即我们这里boy_init)；

最后个成员是GType变量表(可选)

定义好GTypeInfo结构后就可以用g_type_register_来注册对象类型了

g_type_register_用来注册对象类型它第个参数是表示此对象父类对象类型我们这里是G_TYPE_OBJECT这个宏用来表示GObject父类；第 2个参数表示此对象名称这里为"Boy"；第 3个参数是此对象GTypeInfo结构型指针这里赋值为&boyinfo；第 4个参数是对象注册成功后返回此对象整型ID标识

g_object_用来创建个基于G_OBJECT对象它可以有多个参数第个参数是上面说到已注册对象标识ID；第2个参数表示后面参数数量如果为0则没有第 3个参数；第 3个参数开始类型都是GParameter类型它也是个结构型定义为:

```
struct GParameter{
        const gchar* name;
        GValue value;
};
```

有关GValue它是变量类型统定义它是基础变量容器结构用于封装变量值和变量类型可以GOBJECT文档GVALUE部分

信号定义和应用

在GObject系统中信号是种定制对象行为手段同时也是种多种用途通知机制初学者可能是在GTK+中首先接

触到信号这概念事实上在普通界面编程中也可以正常应用这可能是很多初学者未曾想到

个对象可以没有信号也可以有多个信号当有或多个信号时信号名称定义是必不可少此时C语言枚举类型功能就凸显出来了用LAST_SIGNAL来表示最后个信号(不用实现信号)是种非常良好编程风格这里为Boy对象定义了个信号BOY_BORN在对象创建时发出表示Boy对象诞生

同时还需要定义静态整型指针来保存信号标识以便于下步处理信号时使用

对象类结构是所有对象例子所共有我们将信号也定义在对象类结构中如此信号同样也是所有对象例子所共有任意个对象例子都可以处理信号因此我们有必要在在类化中创建信号(这也可能是GObject设计者初衷)g_signal_用来创建个新信号它详细使用思路方法可以在GObjectAPI文档中找到信号创建成功后返回个信号标识ID如此就可以用发射信号g_signal_emit向指定义对象例子发射信号从而执行相应功能

本例中每创建个新Boy对象就会发射次BOY_BORN信号也就会执行次我们定义boy_born也就输出行"Message : A boy was born ."信息

对象属性和思路方法

对象例子所有属性和思路方法般都定义在对象例子结构中属性定义为变量或变量指针而思路方法则定义为指针如此我们定要定义为类型当为指针赋值时才能有效

对象继承

以下为继承自Boy对象Man对象实现Man对象在Boy对象基础上又增加了个属性job和个思路方法bye

```
#ndef __MAN_H__
# __MAN_H__
# "boy.h"
# MAN_TYPE    (man_get_type)
# MAN(obj) (G_TYPE_CHECK_INSTANCE_CAST((obj),MAN_TYPE,Man))
typedef struct _Man Man;
typedef struct _ManClass ManClass;
struct _Man {
    Boy parent;
    char *job;
    void (*bye)(void);
```

```
};
struct _ManClass {
      BoyClass parent_;
};
GType man_get_type(void);
Man*    man_(void);
gchar* man_get_gob(Man *man);
void    man__job(Man *man, gchar *job);
Man*    man__with_name_age_and_job(gchar *name, g age, gchar *job);
void man_info(Man *man);
#end //__MAN_H__
/* man.c */
# "man.h"
 void man_bye(void);
 void man_init(Man *man);
 void man__init(Man *man);
GType man_get_type(void)
{
       GType man_type = 0;
      (!man_type)
      {
            const GTypeInfo man_info = {
                 (ManClass),
                 NULL, NULL,
                 (GClassInitFunc)man__init,
                 NULL, NULL,
                 (Man),
                 0,
                 (GInstanceInitFunc)man_init
            };
            man_type = g_type_register_(BOY_TYPE, "Man", &man_info, 0);
      }
       man_type;
}
 void man_init(Man *man)
{
```

```
	man->job = "none";
	man->bye = man_bye;
}
 void man__init(Man *man)
{
}
Man*    man_(void)
{
	Man *man;
	man = g_object_(MAN_TYPE, 0);
	 man;
}
gchar* man_get_gob(Man *man)
{
	 man->job;
}
void    man__job(Man *man, gchar *job)
{
	man->job = job;
}
Man*    man__with_name_age_and_job(gchar *name, g age, gchar *job)
{
	Man *man;
	man = man_;
	boy__name(BOY(man), name);
	boy__age(BOY(man), age);
	man__job(man, job);
	 man;
}
 void man_bye(void)
{
	g_pr("Goodbye everyone !n");
}
void man_info(Man *man)
{
	g_pr("the man name is %sn", BOY(man)->name);
```

```
    g_pr("the man age is %dn", BOY(man)->age);
    g_pr("the man job is %sn", man->job);
}
```

关键在于定义对象时将父对象例子定义为Boy父类设定为BoyClass在注册此对象时将其父对象类型设为BOY_TYPE在设定对象属性时如用到父对象属性要强制转换下如取得对象name属性就必须用BOY(obj)->nameMan本身没有name属性而其父对象Boy有所以用BOY宏将其强制为Boy类型对象

测试我们定义对象

```
# <glib.h>
# "boy.h"
# "man.h"
（argc, char *argv)
{
    Boy *tom, *peter;
    Man *green, *brown;
    g_type_init;//注意化类型系统必需
    tom = boy__with_name("Tom");
    tom->cry;
    boy_info(tom);
    peter = boy__with_name_and_age("Peter", 10);
    peter->cry;
    boy_info(peter);
    green = man_;
    boy__name(BOY(green), "Green");
//设定Man对象name属性用到其父对象Boy思路方法
    boy__age(BOY(green), 28);
    man__job(green, "Doctor");
    green->bye;
    man_info(green);
    brown = man__with_name_age_and_job("Brown", 30, "Teacher");
    brown->bye;
    man_info(brown);
}
```

Makefile文件如下:

```
CC = gcc
all:
	$(CC) -c boy.c `pkg-config --cflags glib-2.0 gobject-2.0`
	$(CC) -c man.c `pkg-config --cflags glib-2.0 gobject-2.0`
	$(CC) -c .c `pkg-config --cflags glib-2.0 gobject-2.0`
	$(CC) -o simple boy.o man.o .o `pkg-config --libs glib-2.0 gobject-2.0`
```

执行make命令编译编译结束后执行./simple运行此测试输出结果如下:

```
Message : A boy was born .
The Boy is crying ......
The Boy name is Tom
The Boy age is 0
Message : A boy was born .
The Boy is crying ......
The Boy name is Peter
The Boy age is 10
Goodbye everyone !
the man name is Green
the man age is 28
the man job is Doctor
Goodbye everyone !
the man name is Brown
the man age is 30
the man job is Teacher
```

Makefile中用到`pkg-config -cflags -libs gobject-2.0`在GLIB中将线程(gthread)插件(gmoudle)和对象系统(gobject)这 3个子系统区别对待编译时要注意加入相应参数

本文只是概要介绍了如何定义和实现GObject对象GObject系统中还有很多相关内容如:枚举和标识类型(Enumeration and flags types)；Gboxed是Gtype系统中注册种封装为不透明C语言结构类型机制；许多对象用到参数对象都是C结构类型使用者不必了解其结构内部定义即不透明GBoxed即是实现这功能机制；标准参数和变量类型定义(Standard Parameter and Value Types)等它们都以C语言来开发是深入了解和掌握GObject关键

透过以上代码实现我们还可以看出以GLIB为基础GTK+/GNOME开发环境所具有独特编程风格和独到开发

思想这点在长期编程实战中会体验得更深刻

有了GObject系统这基础GTK+通过它将X窗口环境中Control控件(Widget)巧妙封装起来这使开发LINUX平台上GUI应用更方便更快捷

以上代码在Redhat 8.0 Linux平台GLIB2.2.1环境下编译通过

2009-1-15 22:29:58
疯狂代码 http://CrazyCoder.cn/