Left or Right

GObject Tutorial

2009年8月26日 zooyoo 发表评论 阅读评论

GObject Tutorial
Ryan McDougall(2004)

目的

这篇文档可用于两个目的：一是作为一篇学习Glib的GObject类型系统的教程，二是用作一篇按步骤的使用GObject类型系统的入门文章。文章从如何用C语言来设计一个面相对想的类型系统开始，使用GObject作为假设的解决方案。这种介绍的方式可以更好的解释这个开发库为何采用这种形式来设计，以及使用它为什么需要这些步骤。入门文章被安排在教程之后，使用了一种按步骤的，实际的，没有过多解释的组织形式，这样对于某些实际的程序员会更有用些。

读者

这篇教程假想的读者是那些熟悉面向对象概念，但是刚开始接触GObject或者GTK+的开发人员。我会认为您已经了解一门面向对象的语言，和一些C语言的基本命令。

动机

使用一种根本不支持面向对象的语言来编写一个面向对象的系统，这让人听上去有些疯狂。然而我们的确有一些很好的理由来做这样的事情。 但我不会试着去证明作者决定的正确性，并且我认为读者自己就有一些使用GLib的好理由，这里我将指出这个系统的一些重要特性：

- C是一门可移植性很强的语言
- 一个完全动态的系统，新的类型可以在运行时被添加上

这样系统的可扩展性要远强于一门标准的语言，所以新的特性也可以被很快的加入进来。
对面向对象语言来说，面向对象的特性和能力是用语法来定义的。然而因为C并不支持面向对象，所以GObject系统必须要手动的将面向对象的能力移植过来。一般来说要实现这个目标需要一些乏味的工作或者偶尔使用某些神奇的手段。我需要做的是枚举出所有必要的步骤或”咒语”，来使得程序执行起来，也希望能说明这些步骤对您的程序意味着什么。

1. 创建一个非继承的对象

设计

在面向对象领域，对象包含两种类型成员：数据和方法，它们处于同一个对象引用之下。有一种办法可以使用C来实现对象，那就是C的结构体（struct），这样普通的公用成员可以是数据，方法则可以被实现为指向函数的指针。

然而这样的实现却存在着一些严重的缺陷：别扭的语法，类型安全问题，缺少封装。更实际的问题是 - 空间浪费严重。每个实例化后的对象需要一个4字节的指针来指向其每一个成员方法，而这些方法在类封装的范围内是完全相同的，所以这是完全冗余的。例如我们有一个类需要有4个成员方法，一个程序实例化了1000个这个类的对象，这样我们就浪费了接近16KB的空间。

很明显我们只保留一张包含这些指针的表，以供从这个类实例化出来的对象调用要好的多，这样会节省下不少内存资源。
这样的表被称作虚方法表（vtable），GObject系统为每个类在内存中保存了一份。当你想调用一个虚方法时，你必须先向系统请求查找这个对象所对应的虚方法表，这张表如上所述只包含了一个由函数指针组成的结构体。这样你就能复引用这个指针，通过它来调用方法了。

我们称这两种类型为“对象结构体”和“类结构体”，并且将这两种结构体的实例分别称为“对象实例”和“类实例“。这两种结构体合并在一起形成的一个概念上的单元，称作“类”，对这个“类”的实例称作“对象”。
为什么将这样的函数称作“虚函数”的原因是调用它需要在运行时查找合适的函数指针，这样就能允许继承自它的类覆盖这个方法（只需要简单的更改虚函数表中的函数指针指向相应函数入口即可）。这样子类在向上转型（upcast）为父类时就会正常工作，正如我们所知道的C++里的虚方法一样。

尽管这样做可以节省内存和实现虚方法，但从语法上来看，将成员方法与对象用“点操作符”关联起来的能力就不具备了。（译者：因为点操作符关联的将是struct里的方法，而不是vtable里的）。因此我们将使用如下的命名约定来声明类的成员方法：
NAMESPACE_TYPE_METHOD (OBJECT*, PARAMETERS)

非虚方法将被实现在一个普通的C函数里，虚方法也是实现在普通的C函数中，但不同的是这个函数将调用虚函数表中某个合适的方法。私有成员将被实现成只存活在源文件中，而不被导出声明在头文件中。

面向对象通常使用信息隐藏来作为封装的一部分，但在C中却没有简单的办法能隐藏私有成员。一种办法是将私有成员放到一个独立的结构体中，该结构体只定义在源文件中，再向你的公有对象结构体中添加一个指向这个私有类的指针。然而，在开放源代码的世界里，这种保护对于用户执意要做这件错误的事情是微不足道的。大部分的开发者也只是简单的写上几句注释，标明这些成员他们希望被保护为私有的，并且希望用户能尊重这种区别。
到现在为止我们有了两种不同的结构体，但我们没有一个简单的办法来通过一个实例化后的对象找到合适的虚方法表。如我们在上面暗指到的，这应该是系统的职责，系统只需要要求我们向它注册上新声明的类型，就应该能够处理这个问题。系统同时要求我们去向它注册（对象的和类的）结构体构造和析构函数（以及其他的重要信息），这样系统才能正确的实例化我们的对象。系统会通过枚举化所有的向它注册的类型来记录新的对象类型，并且要求**所有实例化对象的第一个成员是一个指向它自己类的虚函数表的指针，每个虚函数表的第一个成员是它在系统中保存的枚举类型的数字表示。**
类型系统要求所有类型的对象结构体和类结构体的第一个成员是一个特殊结构体。在对象结构体中，该特殊结构体是一个指向其类型的对象。因为C语言保证在结构体中声明的第一个成员也是在内存中保存的第一个数据，因此这个类对象可以很容易的通过将这个对象结构体转型而获得到。又因为类型系统要求我们将被继承的父结构体指针声明为子结构体的第一个成员，这样我们只需要在父类中声明一次这个特殊的结构体（译者：即那个指向其类型的对象），我们总是能够通过一次转型而找到虚函数表。
最后我们需要一些函数来定义如何管理对象的生命期：创建类对象的函数，创建实例对象的函数，销毁类对象的函数。但不需要销毁实例对象的函数，因为实例对象的内存管理是一个比较复杂的问题，我们将把这个工作留给更高层的代码来处理。

代码（头文件）

—

a. 使用struct关键字来创建实例对象和类对象，实现“C风格”的对象
我们向结构体名字前添加了一个下划线，然后又增加了一个前置的类型定义typedef，用来给我们的结构体一个合适的名字。这是因为C的语法不允许你声明SomeObject指针在SomeObject中（这对声明链表之类的数据结构很有用）。向上面的约定一节所描述的一样，我们还可以创建一个命名域，称其为“Some“。

```
01 /* 我们的“实例结构体”定义了所有的数据域，这使得对象将是唯一的 */
02 typedef struct _SomeObject SomeObject;
03 struct _SomeObject
04 {
05 GTypeInstance   gtype;
06
07 gint        m_a;
08 gchar*      m_b;
09 gfloat      m_c;
10 };
```

```
11
12 /* 我们的“类结构体”定义了所有的方法函数，这是被实例化出来的对象所共享的 */
13 typedef struct _SomeObjectClass SomeObjectClass;
14 struct _SomeObjectClass
15 {
16 GTypeClass  gtypeclass;
17
18 void        (*method1)  (SomeObject *self, gint);
19 void        (*method2)  (SomeObject *self, gchar*);
20 };
```

b. 声明一个函数，该函数可以在第一次被调用时向系统注册上对象的类型，在此后的调用时就会返回系统记录下的我们声明的那个类型所对应的唯一数字了。这个函数被成为”**get_type**”，返回值是”**GType**”类型，该类型实际上是一个系统用来区别已注册类型的整型数字。由于这个函数是**SomeObject**类型在设计和定义时专有的，我们替它在函数前加上“**some_object_**”。

```
1 /* 这个方法将返回我们新声明的对象类型所关联的GType类型 */
2 GType   some_object_get_type (void);
```

c. 声明管理我们对象生命期的函数：初始化时创建对象的函数，结束时销毁对象的函数。

```
1 /* 类/实例的初始化/销毁函数。它们的标记在gtype.h中定义。 */
2 void    some_object_class_init      (gpointer g_class, gpointer class_data);
3 void    some_object_class_final     (gpointer g_class, gpointer class_data);
4 void    some_object_instance_init   (GTypeInstance *instance, gpointer
  g_class);
```

d. 用**C**函数的通用约定来定义我们的类方法。

```
1 /* 所有这些函数都是SomeObject的方法. */
2 void    some_object_method1 (SomeObject *self, gint);   /* virtual */
3 void    some_object_method2 (SomeObject *self, gchar*); /* virtual */
4 void    some_object_method3 (SomeObject *self, gfloat); /* non-virtual */
```

e. 创建一些样板式代码（**boiler-plate code**），来符合规范，让生活更简单。

```
1 /* 好用的宏定义 */
2 #define SOME_OBJECT_TYPE        (some_object_get_type ())
3 #define SOME_OBJECT(obj)        (G_TYPE_CHECK_INSTANCE_CAST ((obj),
  SOME_OBJECT_TYPE, SomeObject))
4 #define SOME_OBJECT_CLASS(c)        (G_TYPE_CHECK_CLASS_CAST ((c),
  SOME_OBJECT_TYPE, SomeObjectClass))
5 #define SOME_IS_OBJECT(obj)     (G_TYPE_CHECK_TYPE ((obj), SOME_OBJECT_TYPE))
6 #define SOME_IS_OBJECT_CLASS(c)     (G_TYPE_CHECK_CLASS_TYPE ((c),
  SOME_OBJECT_TYPE))
7 #define SOME_OBJECT_GET_CLASS(obj)  (G_TYPE_INSTANCE_GET_CLASS ((obj),
  SOME_OBJECT_TYPE, SomeObjectClass))
```

Code（源程序）

—

现在我们可以继续实现我们刚刚声明过的源文件了。
由于虚函数现在只是一些函数指针，我们还要创建一些正常的、保存在内存中的、可以寻址到的C函数（声明为以”_impl”结尾的，并且不在头文件中导出的），在虚函数中将指向这些函数。
以”some_object_”开头的函数都是对应于SomeObject的定义的，这通常是因为我们会显式的将不同的指针转型到SomeObject，或者会使用类的其它特性。（译者：not very clear）
a. 实现虚方法。

```
/* 虚函数的实现 */
void    some_object_method1_impl (SomeObject *self, gint a)
{
self->m_a = a;
g_print ("Method1: %i\n", self->m_a);
}

void    some_object_method2_impl (SomeObject *self, gchar* b)
{
self->m_b = b;
g_print ("Method2: %s\n", self->m_b);
}
```

b. 实现所有公有方法。实现虚方法时，我们必须使用“GET_CLASS”宏来从类型系统中获取到类对象，用以调用虚函数表中的虚方法。非虚方法时，直接写实现代码即可。

```
/* 公有方法 */
void    some_object_method1 (SomeObject *self, gint a)
{
SOME_OBJECT_GET_CLASS (self)->method1 (self, a);
}

void    some_object_method2 (SomeObject *self, gchar* b)
{
SOME_OBJECT_GET_CLASS (self)->method2 (self, b);
}

void    some_object_method3 (SomeObject *self, gfloat c)
{
self->m_c = c;
g_print ("Method3: %f\n", self->m_c);
}
```

c. 实现构造/析构方法。系统给我们的是泛型指针（我们也相信这个指针的确指向的是一个合适的对象），所以我们在使用它之前必须将其转型为合适的类型。

```
/* 该函数将在类对象创建时被调用 */
void    some_object_class_init      (gpointer g_class, gpointer class_data)
{
SomeObjectClass *this_class = SOME_OBJECT_CLASS (g_class);

/* fill in the class struct members (in this case just a vtable) */
this_class->method1 = &some_object_method1_impl;
this_class->method2 = &some_object_method2_impl;
}

/* 该函数在类对象不再被使用时调用 */
void    some_object_class_final     (gpointer g_class, gpointer class_data)
{
/* No class finalization needed since the class object holds no
pointers or references to any dynamic resources which would need
to be released when the class object is no longer in use. */
}

/* 该函数在实例对象被创建时调用。系统通过g_class实例的类来传递该实例的类。 */
void    some_object_instance_init   (GTypeInstance *instance, gpointer
g_class)
{
SomeObject *this_object = SOME_OBJECT (instance);
```

```
23
24 /* fill in the instance struct members */
25 this_object->m_a = 42;
26 this_object->m_b = 3.14;
27 this_object->m_c = NULL;
28 }
```

d. 实现能够返回给调用者SomeObject的GType的函数。该函数在第一次运行时，它通过向系统注册SomeObject来获取到GType。该GType将被保存在一个静态变量中，以后该函数再被调用时就无须注册可以直接返回该数值了。虽然使用一个独立的函数来注册该类型时可能的，但这样的实现可以保证类在使用前是被注册了的，该函数通常在第一个对象被实例化的时候调用。

```
01 /* 因为该类没有基类，所以基类构造/析构函数是空的 */
02 GType   some_object_get_type (void)
03 {
04 static GType type = 0;
05
06 if (type == 0)
07 {
08 /* 这是系统用来完整描述类型时如何被创建，构造和析构的结构体。 */
09 static const GTypeInfo type_info =
10 {
11 sizeof (SomeObjectClass),
12 NULL,                /* 基类构造函数 */
13 NULL,                /* 基类析构函数 */
14 some_object_class_init,     /* 类对象构造函数 */
15 some_object_class_final,    /* 类对象析构函数 */
16 NULL,                /* 类数据 */
17 sizeof (SomeObject),
18 0,               /* 预分配的字节数 */
19 some_object_instance_init   /* 实例对象构造函数 */
20             };
21
22 /* 因为我们的类没有父类，所以它将被认为是“基础类（fundamental）”，
23                     所以我们必须要告诉系统，我们的类既是一个复合结构的类（与浮点型，
整型，
24                     或者指针不同），并且时可以被实例化的（系统可以创建实例对象，例如
接口
25                     或者抽象类不能被实例化 */
26 static const GTypeFundamentalInfo fundamental_info =
27 {
28 G_TYPE_FLAG_CLASSED | G_TYPE_FLAG_INSTANTIATABLE
29 };
30
31 type = g_type_register_fundamental
32 (
33 g_type_fundamental_next (), /* 下一个可用的GType数 */
34 "SomeObjectType",        /* 类型的名称 */
35 &type_info,          /* 上面定义的type_info */
36 &fundamental_info,       /* 上面定义的fundamental_info */
37 0                /* 类型不是抽象的 */
38 );
39 }
40
41 return  type;
42 }
43
44 /* 让我们来编写一个测试用例吧！ */
45
```

```
46 int main()
47 {
48 SomeObject  *testobj = NULL;
49
50 /* 类型系统初始化 */
51 g_type_init ();
52
53 /* 让系统创建实例对象 */
54 testobj = SOME_OBJECT (g_type_create_instance (some_object_get_type()));
55
56 /* 调用我们定义了的方法 */
57 if (testobj)
58 {
59 g_print ("%d\n", testobj->m_a);
60 some_object_method1 (testobj, 32);
61
62 g_print ("%s\n", testobj->m_b);
63 some_object_method2 (testobj, "New string.");
64
65 g_print ("%f\n", testobj->m_c);
66 some_object_method3 (testobj, 6.9);
67 }
68
69 return  0;
70 }
```

最后需要考虑的

—

我们已经用C实现了第一个对象，但是做了很多工作，并且这并不是真正的面向对象，因为我们故意没有提及任何关于“继承”的方法。在下一节我们将看到如何让工作更加轻松，利用别人的代码－使SomeObject继承与内建的类GObject。
尽管在下文中我们将重用上面讨论的思想和模型，但是尝试去创建一个基础类型，使得它能像其它的GTK+代码一样的工作是非常困难和深入的。因此建议您总是继承GObject来创建新的类型，因为它帮您做了大量背后的工作，使得您的类型能工作的与GTK+要求的保持一致。

二、使用内建的宏定义来自动生成代码
设计

—

您可能已经注意到了，我们上面所做的大部分工作基本上都是机械性的、模板化的工作。大多数的函数都不是通用的，每创建一次类型我们就需要重写一遍。很显然这就是为什么我们发明了计算机的原因 － 让这些工作自动化，让我们的生活更简单！
好的，其实我们很幸运，因为C的预处理器将允许我们编写宏定义来定义新的类型，这样在编译时这些宏定义会自动展开成为合适的C代码。而且使用宏定义还能帮助我们减少人为错误。
然而自动化将使我们丢失部分灵活性。在上面描述的步骤中，我们能有许多可能的变化，但一个宏定义只能实现一种展开。如果这个宏提供了一种轻量级的展开，但我们想要的是一个完整的类型，这样我们仍然需要手写一大堆代码。如果宏提供了一个完整的展开，但我们需要的是一种轻量级的类型，我们将得到许多冗余的代码，花许多时间来填写这些我们用不上的桩代码，或者只是一些普通的错误代码。事实上C预处理器并没有设计成能够自动发现我们感兴趣的代码生成方式，它只包含有限的功能。

代码

—

创建一个新类型的代码非常简单：G_DEFINE_TYPE_EXTENDED (TypeName, function_prefix, PARENT_TYPE, GTypeFlags, CODE)。
第一个参数是类型的名称。第二个是函数名称的前缀，这样能够与我们的命名规则保持一致。第三个是我们希望继承自的基类的GType。第四个是添加到GTypeInfo结构体里的GTypeFlag。第五个是在类型被注册后应该立刻被执行的代码。
看看下面的代码将被展开成为什么样将会对我们有更多的启发。

```
1 G_DEFINE_TYPE_EXTENDED (SomeObject, some_object, 0, some_function())
```

实际展开后的代码将随着系统版本不同而不同。你应该总是检查一下展开后的结果而不是凭主观臆断。
展开后的代码（清理了空格）：

```
01 static void some_object_init (SomeObject *self);
02 static void some_object_class_init (SomeObjectClass *klass);
03 static gpointer some_object_parent_class = ((void *)0);
04
05 static void some_object_class_intern_init (gpointer klass)
06 {
07 some_object_parent_class = g_type_class_peek_parent (klass);
08 some_object_class_init ((SomeObjectClass*) klass);
09 }
10
11 GType some_object_get_type (void)
12 {
13 static GType g_define_type_id = 0;
14 if ((g_define_type_id == 0))
15 {
16 static const GTypeInfo g_define_type_info =
17 {
18 sizeof (SomeObjectClass),
19 (GBaseInitFunc) ((void *)0),
20 (GBaseFinalizeFunc) ((void *)0),
21 (GClassInitFunc) some_object_class_intern_init,
22 (GClassFinalizeFunc) ((void *)0),
23 ((void *)0),
24 sizeof (SomeObject),
25 0,
26 (GInstanceInitFunc) some_object_init,
27 };
28
29 g_define_type_id = g_type_register_static
30 (
31 G_TYPE_OBJECT,
32 "SomeObject",
33 &g_define_type_info,
34 (GTypeFlags) 0
35 );
36
37 { some_function(); }
38 }
39
40 return g_define_type_id;
41 }
```

该宏定义了一个静态变量“
_parent_class”，它是一个指针，指向我们打算创建对象的基类。这在你想去找到虚方法继承自哪里时派上用场，并且这个基类不是由GObject继承下来的基类（译者：not very clear），主要用于链式触发析构函数，这些函数也几乎总是虚的。我们接下来的代码将不再使用这个结构，因为有其它的函数能够不使用静态变量来做到这一点。
你应该注意到了，这个宏没有生成基类的构造析构以及类对象析构函数，如果你需要这些函数，就要自己动手了。

3. 创建一个继承自GObject的对象
设计
—

尽管我们现在能够生成一个基本的对象，但事实上我们故意略过了类型系统的上下文：作为一个复杂库套件的基础 — 那就是图形库GTK+。GTK+的设计要求所有的类应该继承自一个根类。这样就至少能允许一些公共的基础功能能够被共享：如支持信号（让消息可以很容易的从一个对象传递到另一个），通过引用计数来管理对象生命期，支持属性（针对对象的数据域生成简单的setting和getting函数），支持构造和析构函数（用来设置信号、引用计数器、属性）。当我们让对象继承自GObject时，我们获得了上述的一切，并且当与其它基于GObject的库交互时会更加容易。然而，在这章我们不讨论信号、引用计数和属性，或者任何其它专门的特性，这里我们将集中描述继承是在类型系统中如何工作的。
我们都知道，如果LuxuryCar继承自Car，那么LuxuryCar就是Car加上一些新的特性。那我们要如何让系统去实现这样的功能呢？我们可以使用C语言里结构体的一个特性来实现：结构体定义里的第一个成员一定是在内存的最前面。如果我们要求所有的对象将它们的基类声明为它们自己结构体的第一个成员的话，那么我们就能迅速的将指向某个对象的指针转型为指向它基类的指针！尽管这个技巧很好用，并且语法上非常干净，但这种转型的方式只适用于指针 — 你不能这样转型一个普通的结构体。
这种转型技巧是类型不安全的。虽然把一个对象转型为它的基类对象是完全合法的，但实际上非常的不明智（译者：not very clear）。这取决于程序员来保障他的转型是安全的。

创建类型的实例
—
了解了这个技术后，究竟类型系统是如何实例化对象的呢？第一次我们使用g_type_create_instance让系统创建一个实例对象时，它必须要先创建一个类对象供实例来使用。如果该类结构体继承自其它类，系统则需要先创建和初始化这些基类。系统依靠我们指定的结构体（*_get_type函数中的GTypeInfo结构体），来完成这个工作，这个结构体描述了每个对象的实例大小，类大小，构造函数和析构函数。
— 要用g_type_create_instance来实例化一个对象
如果它没有相关联的类对象
创建它并且将其加入到类的层次中
创建实例对象并且返回指向它的指针

当系统创建一个新的类对象时，它先会分配足够的内存来放置这个最终的类对象（译者：“最终的”意指这个新的类对象，相对于其继承的基类们）。然后从最顶端的基类开始到最末端的子类对象，内存级别的用基类的成员域覆写掉这个最终类对象的成员域。这就是子类如何继承自基类的。当把基类的数据复制完后，系统将会在当前状态的类对象中执行基类的“基类初始化“函数。这个覆写和执行“基类初始化”的工作将循环多次，直到这个继承链上的每个基类都被处理过后才结束。接下来系统将在这个最终的类对象上执行最终子类的“基类初始化”和“类初始化”函数。函数“类初始化”有一个参数，该参数可以被认为是类对象构造函数的参数，即上文所提到的“类数据”。
细心的读者可能会问，为什么我们已经有了一个完整的基类对象的拷贝还需要它的基类初始化函数？因为当完整拷贝无法为每个类重新创建出某些数据时，我们就需要基类初始化函数。例如，一个类成员可能指向另外一个对象，并且我们想要每个类对象的成员都指向它自己的对象（内存的拷贝只是“浅拷贝”，我们也许需要一次“深拷贝”）。有经验的GObject程序员告诉我基类初始化函数其实在实际中很少用到。
当系统创建一个新的实例对象时，它会先分配足够的内存来将这个实例对象放进去。在从最顶端的基类开始调用这个基类的“实例初始化”函数在当前的状态下，直到最终的子类。最后，系统在最终类对象上调用最终子类的“实例初始化”函数。
我来总结一下上面所描述到的算法：
— 实例化一个类对象
为最终对象分配内存
从基类到子类开始循环
复制对象内容以覆盖掉最终对象的内容
在最终对象上运行对象自己的基类初始化函数
在最终对象上运行最终对象的基类初始化函数
在最终对象上运行最终对象的类初始化函数（附带上类数据）

— 实例化一个实例对象
为最终对象分配内存
从基类刀子类开始循环
在最终对象上运行实例初始化函数
在最终对象上运行最终对象的实例初始化函数
此时初始化了的类对象和实例对象都已经被创建，系统将实例对象的类指针指向到类对象，这样实例对象就能找到类对象所包含的虚函数表。这就是系统如何实例化已注册类型的过程；其实GObject实现了自己的构造和析构语义正如我们上面所描述的那样！

创建GObject实例

—

前面我们使用g_type_create_instance来创建一个实例对象。然而事实上GObject给我们提供了一个新的API来创建gobject，在上面我们讨论的所有问题之上。GObject实现了三个新方法来被这个API调用，用来创建和销毁新的GObject对象：构造函数(constructor)，部署函数(dispose)以及析构函数(finalize)。

因为C语言缺少很多真正面向对象的语言所具备的多态特性，特别是认出多个构造函数的能力，所以GObject的构造函数需要一些更复杂的实现：

我们怎样才能灵活的传递不同种类的初始化信息到我们的对象中，使得构造函数更加的容易实现？我们也许会考虑限制我们自己只使用拷贝构造函数，用我们需要的数据来填充一个静态”初始化对象“，然后将这个”初始化对象“传递到这个拷贝构造函数中，来完成这个任务 — 简单但是不是非常灵活。

事实上GObject的作者们提供了一种更加通用的解决方案，同时还提供了很好使的getting和setting方法来操作对象的成员数据，这种机制被称作”属性“。在系统中我们的属性用字符串来命名，使用界限和类型检查来保护。属性还可以被声明为仅构造时可写，就像C++中的const变量一样。

属性使用了一种多态的类型（GValue），这种类型允许程序员在不了解其类型的前提下安全的复制一个值。GValue通过记录下值所持有的GType来工作，并且使用类型系统来确认它总是具有一个虚函数，该函数可以处理将其复制到另一个GValue和转换为另一种GType的能力。我们将在下一章讨论GValues和属性。

要为一个GObject创建一个新的属性，我们要定义它的类型、名字，以及默认值，然后创建一个封装这些信息的“属性规格”对象。在GObject的类初始化函数中，我们可以通过g_object_class_install_property来将属性规格绑定到GObject的类对象上。

任何子对象要添加一个新的属性必须覆盖它从GObject继承下来的set_property和get_property虚方法。这些方法是什么将在下一节中介绍。

使用属性我们可以向构造函数传递一组属性规格，加上我们希望的初始值，然后简单的调用GObject的set_property，这样就能获得属性带给我们的神奇功效。然而，下面将看到，构造函数是不会被我们直接调用的。

另一个GObject的构造函数的特性不是那么明显，每个构造函数需要接受一个GType作为其参数之一，并且当它变为其基类时需要将这个GType传递给它基类的构造函数。这是因为GObject的构造函数使用子类的GType来调用g_type_create_instance，这样GObject的构造函数必须要知道它的最终子类对象的GType。

如果我们自己定义构造函数，我们必须覆盖继承自基类的构造函数。自定义的构造函数必须得沿着“继承链”向上，在做任何其他的工作前，先调用完基类的构造函数。然而，因为我们使用了属性，实际上我们从来不用覆盖掉默认的构造函数。

我必须要为上面的离题而道歉，但是这是我们理解系统是如何工作的所必须要克服的困难。如上面所描述的，我们现在能理解GObject的构造函数，g_object_new。这个函数接受一个用于描述继承类的GType类型，一系列属性名（就是C的字符串）和GValue对作为参数。

这一系列属性被转换为键值对列表，以及相关的属性规格，这些属性规格将被在类初始化函数里被安装到系统中。定义在类对象中的构造函数将在被调用时传入GType和构造属性。从最底端的子类构造函数到最顶端的基类构造函数，这条链会一直触发直到GObject的构造函数被执行 — 这实际上才是第一个真正执行的初始化程序。GObject的构造函数现调用g_type_create_instance，并传下我们通过g_object_new一路带上的GType，这样我们上面所描述的细节将会发生，最终创建出实例。然后它获得最终对象的类，对从构造函数传入的所有构造属性调用set_property方法。这就是为什么我们加入一个新属性时必须要覆盖get_/set_property方法的原因。当这一串构造函数返回后，包含在其中的代码将从基类执行到子类。

当基类构造函数返回后，就轮到子类来执行它自己的初始化代码的。这样执行代码的顺序就成为：

a. 从GObject到ChildObject运行实例初始化函数

b. 从GObject到ChildObject运行构造函数

任何剩余的没有传递到构造函数的属性将使用set_property方法在最后一次设置。

读者也许会猜想在什么情况下需要覆盖默认构造函数，将自己的代码放到他们自己的构造函数里？因为我们所有的属性都可以使用虚方法set_property来设置，所以基本上没有覆盖GObject的默认构造函数的必要。

我仍尝试使用伪码总结一下GObject的构造函数过程：

— 基于提供的属性键值对的列表创建合适的GObject对象：

在键值对列表中查找对应的规格

调用最终对象的构造函数并传入规格列表和类型

递归的向下调用GObject的构造函数

调用g_type_create_instance，并传入类型

调用虚方法set_property，传入规格列表

调用set_property，传入剩下的属性

GObject将属性区分为两类，构造和“常规”。

销毁GObject实例

—

当该做的工作完成后，我们可以看看不需要这个对象时会发生些什么。然而面向对象中析构的概念在GObject实现时被分解为了两步：处理和销毁。
“处理”方法在对象知道自己将要被销毁时调用。在该方法中，指向一些资源的引用应该被释放，这样可以避免造成循环引用或者资源稀缺。“处理”方法可以被调用任意次，因此该方法应该能够安全的处理多次调用。一般常见的做法是使用一个静态变量来保护”处理“方法。在“处理”方法调用后，对象本身应该依然能够使用，除非产生了不可恢复的错误（如段错误），所以，“处理”方法不允许释放或者改动某些对象成员。可恢复的错误，例如返回错误码或者空指针则不应该有影响。
“销毁”方法会在对象自己被从内存中清理掉之前释放剩余的资源引用，因此它只能被调用一次。这种分成两个步骤的过程降低了引用计数策略中循环引用发生的可能。
如果我们自定义“处理”和“销毁”方法，就必须要覆盖掉默认的从基类继承下来的相同方法。这两个方法从子类开始调用，沿着继承链向上知道最顶端的基类。
与构造函数不同的是，只要新的对象分配了资源，我们就需要自己实现“处理”和“销毁”方法，来覆盖掉继承自基类的相同方法。
知道某些销毁代码放置到哪里比较合适其实不是一件容易的事。然而，当与引用计数的库（如GTK+）打交道时，我们应该在“处理”方法中解除对其它资源对象的引用，而在“销毁”方法中释放掉所有的内存或者关闭所有的文件描述字。
上面我们讨论过g_object_new，但是我们什么时候来销毁这些对象呢？其实上面也有提示过，GObject使用了引用计数的技术，也就是说它为有多少个其它对象或函数现在正在“使用”或者引用这个对象保存了一个整型的数据。当你在使用GObject时，如果你向保护你的对象不在使用时被销毁掉，你必须及早调用g_object_ref，将对象作为参数传递给它。这样就为引用计数器增加了1。如果你没有做这件事就意味着你允许对象被自动销毁，这也许会导致你的程序崩溃。
同样的，当对象完成了它的任务后，你必须要调用g_object_unref。这样会使引用计数器减少1，并且系统会检查它是否为0.当计数器为0时，对象将被先调用“处理”方法，最终被“销毁”掉。如果你没有解除到该对象的引用，则会导致内存泄漏，因为计数器永远不会回到0。
现在我们已经准备好了来写一些代码了！但是不要让上面冗长和复杂的描述吓唬到您。如果你没有完全理解上面所提到的，别紧张 — GObject的确是很复杂的！继续读下去，你会看到许多细节，试试一些例子程序，或者去睡觉吧，明天再来接着读。
下面的程序与第一个例子很相似，事实上我去掉了更多的不合逻辑的、冗余的代码。

代码（头文件）

—

1. 我们仍然按照上面的方式继续，但是这次将把基类对象放到结构体的第一个成员位置上。事实上就是GObject。

```
01 /* 我们的“实例结构体”定义了所有的数据域，这使得对象将是唯一的 */
02 typedef struct _SomeObject SomeObject;
03 struct _SomeObject
04 {
05         GObject         parent_obj;
06
07         /* 下面应该是一些数据 */
08 };
09
10 /* 我们的“类结构体”定义了所有的方法函数，这是被实例化出来的对象所共享的 */
11 typedef struct _SomeObjectClass SomeObjectClass;
12 struct _SomeObjectClass
13 {
14         GTypeClass      parent_class;
15
16         /* 下面应该是一些方法 */
17 };
```

2. 头文件剩下的部分与第一个例子基本相同。

代码（源文件）

—

我们需要增加一些对被覆盖的GObject方法的声明

```
/* 这些是GObject的构造和析构方法，它们的声明在gobject.h中 */
void    some_object_constructor (GType this_type, guint n_properties,
GObjectConstructParam *properties)
{
        /* 如果有子类要继承我们的对象，那么this_type将不是SOME_OBJECT_TYPE，
           g_type_peek_parent再是SOME_OBJECT_TYPE的话，将会造成无穷循环 */

        GObjectClass *parent_class = g_type_class_peek (g_type_peek_parent
(SOME_OBJECT_TYPE()));

        some_object_parent_class-> constructor (self_type, n_properties,
properties);

        /* 这里很少需要再做其它工作 */
}

void    some_object_dispose (GObject *self)
{
        GObjectClass *parent_class = g_type_class_peek (g_type_peek_parent
(SOME_OBJECT_TYPE()));
        static gboolean first_run = TRUE;

        if (first_run)
        {
                first_run = FALSE;

                /* 对我们持有引用的所有GObject调用g_object_unref，但是不要破坏这个
对象 */

                parent_class-> dispose (self);
        }
}

void    some_object_finalize (GObject *self)
{
        GObjectClass *parent_class = g_type_class_peek (g_type_peek_parent
(SOME_OBJECT_TYPE()));

        /* 释放内存和关闭文件 */

        parent_class-> finalize (self);
}
```

GObjectConstructParam是一个有两个成员的结构体，一个是GParamSpec类型，就是对参数的一组描述，另外一个是GValue类型，就是一组对应的值。

```
/* 这是GObject的Get和Set方法，它们的声明在gobject.h中 */
void    some_object_get_property (GObject *object, guint property_id, GValue
*value, GParamSpec *pspec)
{
}

void    some_object_set_property (GObject *object, guint property_id, const
GValue *value, GParamSpec *pspec)
{
}

/* 这里是我们覆盖函数的地方，因为我们没有定义属性或者任何域，下面都是不需要的 */
void    some_object_class_init          (gpointer g_class, gpointer
```

```
    class_data)
12  {
13          GObjectClass    *this_class     = G_OBJECT_CLASS (g_class);
14
15          this_class-> constructor        = &some_object_constructor;
16          this_class-> dispose            = &some_object_dispose;
17          this_class-> finalize           = &some_object_finalize;
18
19          this_class-> set_property       = &some_object_set_property;
20          this_class-> get_property       = &some_object_get_property;
21  }
```

要想讨论关于创建和销毁GObject，我们就必须要了解属性和其它特性。然而，我把操作属性的示例放到下一节来叙述。以避免过于复杂而使得你灰心。在你对这些概念有些实作经验后，它们将开始显现出来存在的意义。正如上面所言，我们将自己限制在创建一个基础的GObject类，在下一节我们将真正的编写一些函数。 重要的是我们获得了让下面的学习更轻松的工具。

4. 属性
上面已经提到属性是个很奇妙的东西，以及它是如何使用的，但是在深入介绍属性之前，我们又得先离题一会。

GValues
—
C是一门强类型语言，也就是说变量的声明的类型必须和它被使用的方式保持一致，否则编译器就会报错。这是一件好事，它使得程序编写起来更迅速，帮助我们发现可能会导致系统崩溃或者不安全的问题。但这又是件坏事，因为程序员实际上活在一个很难什么事都能严格的世界上，而且我们也希望声明的类型能够具备多态的能力 — 也就是说类型能够根据上下文来改变它们自己的行为。上面所讨论过的继承，通过C语言的转型我们可以获得一些多态的能力。然而，当使用无类型指针作为参数传递给函数时，可能会产生问题。幸运的是，类型系统给了我们另外一个C语言没有的工具：GType。
让我们更清楚的说明一下问题。我需要一种数据类型，可以实现一个可以容纳多类型元素的链表，我想为这个链表编写一些接口，可以不依赖于任何特定的类型，并且不需要我为每种数据类型声明一个冗余的函数。这种接口必然能涵盖多种类型，所以我们称它为GValue（Generic Value，泛型）。我们该如何实现这样一个东西呢？
我们创建了封装这种类型的结构体，它具有两个成员域：所有可表现的基础类型的联合（union），和表示保存在这个union中的值的GType。这样我们就可以将值的类型隐藏在GValue中，并且通过检查对GValue的操作来保证类型是安全的。这样还减少了多余的以类型为基础的操作接口（如get_int，set_float，…），统一换成了g_value_*的形式。
细心的读者会发现每个GValue都占据了至少最大的基础类型的内存数量（通常是8字节），加上GType自己的大小。GValues在空间上不是最优的，包含了不小的浪费，因此它不应该被用到太大的数量级。它最常用在定义一些泛型的API。

```
/* 让我们使用GValue来复制整型数据！  */
#define g_value_new(type) g_value_init (g_new (GValue, 1), type)

GValue *a = g_value_new (G_TYPE_UCHAR);
GValue *b = g_value_new (G_TYPE_INT);
int c = 0;

g_value_set_uchar (a, ‘a’);
g_value_copy (a, b);

c = g_value_get (b);
g_print (“w00t: %d\n”, c);

g_free (a);
g_free (b);
```

设计

—

我们已经在上面接触过属性了，所以我们也有了对它们的初步判断，但我们将继续来了解一下设计它们的最初动机。
要编写一个泛型的属性设置机制，我们需要一个将其参数化的方法，以及与实例结构体中的成员变量名查重的机制。从外部上看，我们希望使用C字符串来区分属性和公有API，但是内部上来说，这样做会严重的影响效率。因此我们枚举化了属性，使用一个索引来标示代码中的属性。
上面提过属性规格，在Glib中被称作GParamSpec，它保存了对象的gype，对象的属性名称，对象枚举ID，系统需要这样一个能把所有东西都粘在一起的大胶水。
当我们需要设置或者获取一个属性的值时，调用g_object_set/get_property，需要指定属性的名字，并且带上GValue用来保存我们要设置的值。g_object_set_property函数将在GParamSpec中查找我们要设置的属性名称，查找我们对象的类，并且调用对象的set_property方法。这意味着如果我们要增加一个新的属性，我们必须覆盖默认的set/get_property方法。而且基类包含的属性将被它自己的set/get_property方法所正常处理，因为GParamSpec就是从基类传递下来的。最后，我们必须通过事先通过对象的class_init方法来加入一个GParamSpec参数！
假设我们已经有了如上一节所描述的那样一个可用的框架，那么现在让我们来为SomeObject加入处理属性的代码！

代码（头文件）

—

1. 除了我们增加了两个属性外，其余同上面的一样

```
01 /* 我们的“实例结构体”定义了所有的数据域，这使得对象将是唯一的 */
02 typedef struct _SomeObject SomeObject;
03 struct _SomeObject
04 {
05         GObject         parent_obj;
06
07         /* 新增加的属性 */
08         int             a;
09         float           b;
10
11         /* 下面应该是一些数据 */
12 };
```

代码（源文件）

—

1. 创建一个枚举类型用来内部记录属性。

```
1 enum
2 {
3         OBJECT_PROPERTY_A = 1 << 1;
4         OBJECT_PROPERTY_B = 1 << 2;
5 };
```

2. 实现新增的处理属性的函数。

```
01 void    some_object_get_property (GObject *object, guint property_id, GValue
   *value, GParamSpec *pspec)
02 {
03         SomeObject *self = SOME_OBJECT (object);
04
05         switch (property_id)
06         {
07                 case OBJECT_PROPERTY_A:
08                         g_value_set_int (value, self-> a);
09                         break;
10
11                 case OBJECT_PROPERTY_B:
```

```
                        g_value_set_float (value, self-> b);
                        break;

                default: /* 没有属性用到这个ID！！ */
        }
}

void    some_object_set_property (GObject *object, guint property_id, const
GValue *value, GParamSpec *pspec)
{
        SomeObject *self = SOME_OBJECT (object);

        switch (property_id)
        {
                case OBJECT_PROPERTY_A:
                        self-> a = g_value_get_int (value);
                        break;

                case OBJECT_PROPERTY_B:
                        self-> b = g_value_get_float (value);
                        break;

                default: /* 没有属性用到这个ID！！ */
        }
}
```

3. 覆盖继承自基类的set/get_property方法，并且传入GParamSpecs。

```
/* 这里是我们覆盖函数的地方 */
void    some_object_class_init          (gpointer g_class, gpointer
class_data)
{
        GObjectClass    *this_class     = G_OBJECT_CLASS (g_class);
        GParamSpec      *spec;

        this_class-> constructor        = &some_object_constructor;
        this_class-> dispose            = &some_object_dispose;
        this_class-> finalize           = &some_object_finalize;

        this_class-> set_property       = &some_object_set_property;
        this_class-> get_property       = &some_object_get_property;

        spec = g_param_spec_int
        (
                "property-a",                           /* 属性名称 */
                "a",                                    /* 属性昵称 */
                "Mysterty value 1",                     /* 属性描述 */
                5,                                      /* 属性最大值 */
                10,                                     /* 属性最小值 */
                5,                                      /* 属性默认值 */
                G_PARAM_READABLE |G_PARAM_WRITABLE      /* GParamSpecFlags */
        );
        g_object_class_install_property (this_class, OBJECT_PROPERTY_A, spec);

        spec = g_param_spec_float
        (
                "property-b",                           /* 属性名称 */
                "b",                                    /* 属性昵称 */
                "Mysterty value 2                       /* 属性描述 */
```

```
                0.0,                                    /* 属性最大值 */
                1.0,                                    /* 属性最小值 */
                0.5,                                    /* 属性默认值 */
                G_PARAM_READABLE |G_PARAM_WRITABLE      /* GParamSpecFlags */
        );
        g_object_class_install_property (this_class, OBJECT_PROPERTY_B, spec);
}
```

分类: C/C++, Linux 标签:
评论 (1) Trackbacks (0) 发表评论 Trackback

1. 
   dyei
   2009年10月21日15:30 | #1
   回复 | 引用

   非常完美，不过gobject硬是将编译期的一些工作放到动态运行期实现，这个就导致它的一点性能问题。

1. 本文目前尚无任何 trackbacks 和 pingbacks.

昵称
电子邮箱 (我们会为您保密)
网址

订阅评论
提交评论 (Ctrl+Enter)

address alignment问题 使用 Strace 和 GDB 调试工具的乐趣
订阅

Recent Posts

- t43安装ubuntu 10.04 LTS黑屏解决办法
- install eclipse + CDT + ADT on ubuntu 10.04
- git服务器搭建zz
- 设置往vim中粘贴数据时不自动缩进
- imagecache module of drupal not work with nginx
- google ghs 替代方法
- 《Android系统开发》笔记4-底层库和程序
- 用ubuntu 10.04安装盘修复Grub2
- 《Android系统开发》笔记3-Android内核与驱动
- android的logcat详细用法zt

Tag Cloud

入门 分区 安装 插件 教程 添加新标签 版本控制 粘贴 编程思想，Thinking in c++ 视频 activity Android binder bluetooth bluez C/C++ eclipse extern git grub H.261 H.263 H.264 intent Java Linux livemedia MAP_SHARED mmap Mpeg mplayer

NAL nalu native nginx opencore packetvideo rtp svn tftp ubuntu ubunu v4l2 vim Web

Categories

- Android
- bluetooth
- C/C++
- Debug
- Java
- Linux
- Media & Stream
- Uncategorized
- Web
- 乒乒乓乓
- 天黑闭眼
- 读书笔记

Blogroll

- Development Blog
- Documentation
- Plugins
- Suggest Ideas
- Support Forum
- Themes
- WordPress Planet

Archives

- 2010年八月
- 2010年七月
- 2010年六月
- 2010年五月
- 2010年四月
- 2010年三月
- 2010年一月
- 2009年十一月
- 2009年十月
- 2009年九月
- 2009年八月
- 2009年七月
- 2009年六月
- 2009年五月
- 2009年四月
- 2009年三月
- 2009年二月
- 2009年一月
- 2006年十一月
- 2006年十月
- 2006年九月
- 2006年七月
- 2006年六月
- 2006年五月
- 2006年四月
- 2006年三月
- 2006年二月
- 2005年十二月

- 2005年十月

Meta

- 登录