人工智能 dlopen 在Linux C/C++用法详解

hi_clare · 2020年01月20日 · 390 次阅读

本文汇总了几篇讲解 dlopen 用法的文章，详细解析 dlopen 系列函数在 Linux C/C++ 中的使用。

目录

1.dlopen 系列 API

Linux 提供了一套 API 来动态装载库。这些 API 包括

```
#include <dlfcn.h>
void *(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
char *dlerror(void);
```

下面分别解释这几个函数。

dlopen

dlopen() 函数以指定模式打开指定的动态链接库文件，并返回动态链接库的句柄。参数 flag 有以下两种常用的值，并且必须指定其一。
RTLD_LAZY:在 dlopen 返回前，对于动态库中存在的未定义的变量 (如外部变量 extern，也可以是函数) 不执行解析，就是不解析这个变量的地址。
RTLD_NOW：与上面不同，他需要在 dlopen 返回前，解析出每个未定义变量的地址，如果解析不出来，在 dlopen 会返回 NULL.

dlsym

dlsym() 函数根据动态链接库操作句柄 (handle) 与符号 (symbol)，返回符号对应的地址。使用这个函数不但可以获取函数地址，也可以获取变量地址。参数的含义如下:
handle：由 dlopen 打开动态链接库后返回的指针；
symbol：要求获取的函数或全局变量的名称。

dlclose

dlclose() 函数用于关闭指定句柄的动态链接库，只有当此动态链接库的使用计数为 0 时,才会真正被系统卸载。

dlerror

当动态链接库操作函数,如 dlopen/dlsym/dlclose//执行失败时，dlerror() 函数可以返回最近的出错信息，返回值为 NULL 时表示操作函数执行成功。

C 语言用户需要包含头文件 dlfcn.h 才能使用上述 API。

2.dlopen 函数在 C 语言中的使用

下面通过一个简单的例子来示范 dlopen 函数在 C 语言中的使用。
首先是动态库的源代码

```
#include <stdio.h>
void hello(void)
{
    printf("hellon");
}
```

可执行文件的源代码，代码中会动态加载 so 库

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
    void *handle;
    void (*callfun) ();
    char *error;
    handle = dlopen("./hello.so", RTLD_LAZY);


if (!handle) {
        printf("%s n", dlerror());
        exit(1);
    }
    dlerror();
    callfun = dlsym(handle, "hello");
    if ((error = dlerror()) != NULL) {
```

朱老板
@hi_clare

```
        printf("%s n", error);
        exit(1);
    }
    callfun();
    dlclose(handle);
    return 0;
}
```

编译命令：

```
gcc -fPIC -shared -o hello.so hello.c
gcc -o main main.c -ldl
```

执行：
./main
就能看到结果

3.dlopen 与 constructor/destructor

dlopen 的 man page 里面有提到。在一些旧的代码中可能会用到两个特殊的函数：init 和_fini。_init 和_fini 函数用在装载和卸载某个模块时分别控制该模块的构造器和析构器 (或构造函数和析构函数)。他们的 C 语言原型如下：

```
void _init(void);
void _fini(void);
```

当一个库通过 dlopen() 动态打开或以共享库的形式打开时，如果_init 在该库中存在且被输出出来，则_init 函数会被调用。如果一个库通过 dlclose() 动态关闭或因为没有应用程序引用其符号而被卸载时，_fini 函数会在库卸载前被调用。当使用你自己的_init 和_fini 函数时，需要注意不要与系统启动文件一起链接。可以使用 GCC 选项 -nostartfiles 做到这一点。但是，使用上面的函数或 GCC 的-nostartfiles 选项并不是很好的习惯，因为这可能会产生一些意外的结果。相反，库应该使用**attribute** ((constructor)) 和**attribute** ((destructor)) 函数属性来输出它的构造函数和析构函数。如下所示：

```
void __attribute_ ((constructor)) x_init(void)
void attribute ((destructor)) x_fini(void)
```

构造函数会在 dlopen() 返回前或库被装载时调用。析构函数会在这样几种情况下被调用：dlclose() 返回前，或 main() 返回后，或装载库过程中 exit() 被调用时。下面用一个例子来说明。
动态库的代码如下：

```
#include <stdio.h>
#include <stdlib.h>

void attribute ((constructor)) Double_init() {
    printf("_init invoked!n");
}



void attribute ((destructor)) Double_fini() {
    printf("_fini invoked!n");
}



/* Customized routines here */
int Double(int arg) {
    return (arg + arg);
}
```

主程序的代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
    void *handle;
    int (*func) (int);
    const char *error;



/* shared object (1) */
    handle = dlopen("./share.so", RTLD_NOW);
    if (!handle) {
        fputs(dlerror(), stderr);
        exit(1);
    }
    func = dlsym(handle, "Double");
    if ((error = dlerror()) != NULL) {
```

```
        fputs(error, stderr);
        exit(1);
    }
    printf("invoking Double(2) => %dn", (*func) (2));
    dlclose(handle);
}
```

编译命令：

```
gcc -fPIC -shared -o share.so hello.c
gcc -o main main.c -ldl
```

执行：
./main
就能看到结果

4.dlopen 函数在 C++ 语言中的使用

在 C 语言中，加载一个库轻而易举（调用 dlopen、dlsym 和 dlclose 就够了），但对 C++ 来说，情况稍微复杂。动态加载一个 C++ 库的困难一部分是因为 C++ 的 name mangling，另一部分是因为 dlopen API 是用 C 语言实现的，因而没有提供一个合适的方式来装载 C++ 类。
在解释如何装载 C++ 库之前，最好再详细了解一下 name mangling。我推荐您了解一下它，即使您对它不感兴趣。因为这有助于您理解问题是如何产生的，如何才能解决它们。

4.1 Name Mangling

在每个 C++ 程序（或库、目标文件）中，所有非静态（non-static）函数在二进制文件中都是以 “符号（symbol）” 形式出现的。这些符号都是唯一的字符串，从而把各个函数在程序、库、目标文件中区分开来。在 C 中，符号名正是函数名：strcpy 函数的符号名就是 “strcpy”，等等。这可能是因为两个非静态函数的名字一定各不相同的缘故。而 C++ 允许重载（不同的函数有相同的名字但不同的参数），并且有很多 C 所没有的特性——比如类、成员函数、异常说明——几乎不可能直接用函数名作符 号名。为了解决这个问题，C++ 采用了所谓的 name mangling。它把函数名和一些信息（如参数数量和大小）杂糅在一起，改造成奇形怪状，只有编译器才懂的符号名。例如，被 mangle 后的 foo 可能 看起来像 foo@4%6^，或者，符号名里头甚至不包括 “foo”。其中一个问题是，C++ 标准（目前是 [ISO14882]）并没有定义名字必须如何被 mangle，所以每个编译器都按自己的方式来进行 name mangling。有些编译器甚至在不同版本间更换 mangling 算法（尤其是 g++ 2.x 和 3.x）。即使您搞清楚了您的编译器到底怎么进行 mangling 的，从而可以用 dlsym 调用函数了，但可能仅仅限于您手头的这个编译器而已，而无法在下一版编译器下工作。

4.2 类

使用 dlopen API 的另一个问题是，它只支持加载函数。但在 C++ 中，您可能要用到库中的一个类，而这需要创建该类的一个实例，这不容易做到。

4.3 解决方案

4.3.1 extern “C”

C++ 有个特定的关键字用来声明采用 C binding 的函数：extern “C” 。 用 extern “C” 声明的函数将使用函数名作符号名，就像 C 函数一样。因此，只有非成员函数才能被声明为 extern “C”，并且不能被重载。尽管限制多多，extern “C” 函数还是非常有用，因为它们可以象 C 函数一样被 dlopen 动态加载。冠以 extern “C” 限定符后，并不意味着函数中无法使用 C++ 代码了，相反，它仍然是一个完全的 C++ 函数，可以使用任何 C++ 特性和各种类型的参数。

4.3.2 加载函数

在 C++ 中，函数用 dlsym 加载，就像 C 中一样。不过，该函数要用 extern “C” 限定符声明以防止其符号名被 mangle。
示例 1.加载函数
代码:

```
#include <iostream>
#include <dlfcn.h>

int main() {
    using std::cout;
    using std::cerr;



cout << "C++ dlopen demonn";



// open the library
    cout << "Opening hello.so...n";
    void *handle = dlopen("./hello.so", RTLD_LAZY);



if (!handle) {
        cerr << "Cannot open library: " << dlerror() << 'n';
```

```
        return 1;
    }
    // load the symbol
    cout << "Loading symbol hello...n";
    typedef void (*hello_t) ();
    hello_t hello = (hello_t) dlsym(handle, "hello");
    if (!hello) {
        cerr << "Cannot load symbol 'hello': " << dlerror() << 'n';
        dlclose(handle);
        return 1;
    }
    // use it to do the calculation
    cout << "Calling hello...n";
    hello();


// close the library
    cout << "Closing library...n";
    dlclose(handle);
}


#include <iostream>


extern "C" void hello()
{
    std::cout << "hello" << 'n';
}
```

在 hello.cpp 中函数 hello 被定义为 extern “C”。它在 main.cpp 中被 dlsym 调用。函数必须以 extern “C” 限定，否则我们无从知晓其符号名。

编译命令：

```
g++ -fPIC -shared -o hello.so hello.cpp
g++ -o main main.cpp -ldl
```

运行
./main
可以看到结果。

4.3.3 加载类

加载类有点困难，因为我们需要类的一个实例，而不仅仅是一个函数指针。我们无法通过 new 来创建类的实例，因为类不是在可执行文件中定义的，况且（有时候）我们连它的名字都不知道。解决方案是：利用多态性！ 我们在可执行文件中定义一个带虚成员函数的接口基类，而在模块中定义派生实现类。通常来说，接口类是抽象的（如果一个类含有虚函数，那它就是抽象的）。因为动态加载类往往用于实现插件，这意味着必须提供一个清晰定义的接口——我们将定义一个接口类和派生实现类。接下来，在模块中，我们会定义两个附加的 helper 函数，就是众所周知的 “类工厂函数（class factory functions）（译者注：或称对象工厂函数）” 。其中一个函数创建一个类实例，并返回其指针; 另一个函数则用以销毁该指针。这两个函数都以 extern “C” 来限定修饰。为了使用模块中的类，我们用 dlsym 像示例 1 中加载 hello 函数那样加载这两个函数，然后我们就可以随心所欲地创建和销毁实例了。

我们用一个一般性的多边形类作为接口，而继承它的三角形类（译者注：正三角形类）作为实现。
主文件代码

```
//----------
//main.cpp:
//----------
#include "polygon.h"
#include <iostream>
#include <dlfcn.h>

int main() {
    using std::cout;
    using std::cerr;


// load the triangle library
    void *triangle = dlopen("./triangle.so", RTLD_LAZY);
    if (!triangle) {
        cerr << "Cannot load library: " << dlerror() << 'n';
        return 1;
    }
    // reset errors
    dlerror();
```

```
// load the symbols
    create_t *create_triangle = (create_t *) dlsym(triangle, "create");
    const char *dlsym_error = dlerror();
    if (dlsym_error) {
        cerr << "Cannot load symbol create: " << dlsym_error << 'n';
        return 1;
    }


destroy_t *destroy_triangle = (destroy_t *) dlsym(triangle, "destroy");
    dlsym_error = dlerror();
    if (dlsym_error) {
        cerr << "Cannot load symbol destroy: " << dlsym_error << 'n';
        return 1;
    }
    // create an instance of the class
    polygon *poly = create_triangle();


// use the class
    poly->set_side_length(7);
    cout << "The area is: " << poly->area() << 'n';


// destroy the class
    destroy_triangle(poly);


// unload the triangle library
    dlclose(triangle);
}
```

polygon 接口类头文件

```
//----------
//polygon.hpp:
//----------
#ifndef POLYGON_HPP
#define POLYGON_HPP

class polygon {
  protected:
    double side_length_;


public:
    polygon():side_length_(0) {}
    virtual ~ polygon() {}


void set_side_length(double side_length) {
        side_length_ = side_length;
    }


virtual double area() const = 0;
};


// the types of the class factories
typedef polygon *create_t();
typedef void destroy_t(polygon *);


#endif
```

triangle 实现类库文件

```
//----------
//triangle.cpp:
//----------
#include "polygon.h"
#include <cmath>

class triangle:public polygon {
  public:
    virtual double area() const {
        return side_length_ * side_length_ * sqrt(3) / 2;
    }
};
```

```
// the class factories
extern "C" polygon * create() {
    return new triangle;
}


extern "C" void destroy(polygon * p) {
    delete p;
}
```

编译命令：

```
g++ -fPIC -shared -o triangle.so polygon.h triangle.cpp
g++ -o main2 main1.cpp polygon.h -ldl
```

运行
./main2
可以看到结果

加载类时有一些值得注意的地方：
◆ 你必须在模块或者说共享库中同时提供一个创造函数和一个销毁函数，且不能在执行文件内部使用 delete 来销毁实例，只能把实例指针传递给模 块的销毁函数处理。这是因为 C++ 里头，new 操作符可以被重载；这容易导致 new-delete 的不匹配调用，造成莫名其妙的内存泄漏和段错误。这在用不同的标准库链接模块和可执行文件时也一样。
◆ 接口类的析构函数在任何情况下都必须是虚函数（virtual）。因为即使出错的可能极小，近乎杞人忧天了，但仍旧不值得去冒险，反正额外的开销微不足道。如果基类不需要析构函数，定义一个空的（但必须虚的）析构函数吧，否则你迟早要遇到问题，我向您保证。你可以在 comp.lang.c++ FAQ(http://www.parashift.com/c++-faq-lite/) 的第 20 节了解到更多关于该问题的信息。

参考文章：

http://blog.chinaunix.net/uid-10540984-id-3508235.html
http://www.cnblogs.com/leaven/archive/2011/01/28/1947180.html
http://www.faqs.org/docs/Linux-mini/C++-dlopen.html
http://www.linuxjournal.com/article.php?sid=3687
http://linux.die.net/man/3/dlopen
http://beyond99.blog.51cto.com/1469451/737840
http://blog.linux.org.tw/~jserv/archives/001561.html