DynamicModules: Руководство Программиста

DocumentId:GradSoft-PR-r-15.09.2001-v1.0.0

Ruslan Shevchenko

May 8, 2003

Contents

1 Введение

DynamicModules - это кроссплатформенная компонета на C++, предназначенная для динамической загрузки разделяемых библиотек (`lib<smth>.so` для UNIX, `<smth>.dll` для Windows NT)

Пакет **DynamicModules** разработан и поддерживается компанией GradSoft и поставляется в исходных кодах. Домашняя страница разработчика http://www.gradsoft.kiev.ua/

Этот документ является неформальным описанием пакета. Полная специфиткация находится в описании API

www.gradsoft.kiev.ua/common/Products/Toolbox/DynamicModule/API/

2 Общие сведения

Пакет DynamicModules позволяет строить программу в виде набора, состоящего из главного модуля и (нескольких) динамически загружаемых библиотек.

Библиотека реализует некоторый, предоставленный пользователю, интерфейс и обеспечивает доступ к реализации через статический объект, произведенный от класса `DynamicModule` или `DynamicModuleTag`.

Головная программа получает доступ к такому объекту ("корневому объекту сервиса") с помощью методов класса `DynamicModules` (методы load, unload, getModule), и создает себе классы для конкретной работы через него.

Разработчик головного модуля компилирует программу с библиотекой `libDynamicModules.{a,so}` для UNIX либо `DynamicModules.lib` для Windows NT.

3 Простой пример

3.1 Структура примера

Простой пример использования средств DynamicModules состоит из следующих компонент:

1. общего заголовочного файла HelloInterface.h, который должен включать заголовочный файл DynamicModules.h
2. загружаемого модуля HelloModule.cpp
3. основной программы Main.cpp

3.2 Общий заголовочный файл

В общем заголовочном файле описаны структуры данных, которыми пользуются и клиент, и библиотека:

1. класс HelloInterface - интерфейс того средства, которое, собственно, будет трудиться.
2. класс HelloModule - интерфейс "корневого объекта" сервиса – т.е. средства, при помощи которого клиент будет получать доступ к реализации HelloInterface

(в сумме это все, что должно быть известно клиенту):

```
#include <GradSoft/DynamicModules.h>

/**
 * интерфейс, который требует какой-то нетривиальной реализации:
 **/
```

```
 class HelloInterface
 {
 public:
   virtual ~HelloInterface() {}
   virtual void hello() = 0;
 };

/**
 * 'корневой объект' сервиса:
 **/
 class HelloModule: public GradSoft::DynamicModule
 {
 public:
  /**
   * возвращает ссылку на экземпляр HelloInterface:
   **/
   virtual HelloInterface* create(const char* name) = 0;
 };
```

3.3 Загружаемый модуль

Файл HelloModule.cpp содержит реализацию HelloModule и HelloInterface:

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

// реализуем основную функциональность:

class Hello: public HelloInterface
{
public:

  Hello(const char* x) {}
  virtual ~Hello() {}
  virtual void hello()
  {
    std::cerr << "Hi! I'm Hello" << std::endl;
  }
};

// и соответствующий 'корневой объект' (или Factory):

class Hello1Module: public HelloModule
{
public:
```

```
////
// Мы должны перегрузить следующие методы класса DynamicModule:
////

  // имя сервиса
  const char* name() const { return "Hello"; }

  // информация о версии
  int versionMajor() const { return 1; }
  int versionMinor() const { return 0; }
  int versionSubMinor() const { return 0; }

  // автор
  const char* author() const { return "Grad-Soft LTD"; }

////
// И реализовать специфический метод HelloModule:
////

  HelloInterface* create(const char* args)
  {
    return new Hello(args);
  }

};

// теперь создаем модуль:
Hello1Module tagHelloModule;

// и экспортируем объект:
EXPORT_OBJECT(tagHelloModule)
```

3.4 Головной модуль

```
#include <HelloInterface.h>
#include <memory>
#include <iostream>

using namespace GradSoft;

int main()
{
 try {
```

```
  // Загружаем модуль Hello из текущей директории:
  DynamicModule& dmHello = DynamicModules::load("tagHelloModule","./Hello");

  // Для того, чтобы вызвать create, преобразуем к типу HelloModule:
  HelloModule& helloModule = dynamic_cast<HelloModule&>(dmHello);

  // Используем:
  {
   std::auto_ptr<HelloInterface> h1 ( helloModule.create("xxx") );
   h1->hello();
  }

  // Выгружаем модуль:
  DynamicModules::unload("tagHelloModule");

 }catch(const DynamicModules::Error& ex){

  // Сообщаем о случившемся:
  std::cerr << "Error during loading DynamicModule" << std::endl;
  std::cerr << ex.what() << std::endl;
  return 1;
 }
 return 0;
}
```

3.5 Собираем все вместе

Для того, чтобы собрать описанную программу, необходимо:

1. Скомпилировать разделяемую библиотеку `Hello.{so,dll}` при помощи команды типа:
   (a) Для UNIX с компилятором GCC:
   ```
   g++ -shared -o Hello.so [...] HelloModule.cpp
   ```
   (b) Для Windows NT:
   ```
   cl Hello1Module.cpp /MD /GR /GX /D "WIN32" [...] /link -DLL /out:Hello.dll
   ```
   (Внимание: флаги /MD /GR /D "WIN32" должны использоваться обязательно)
2. Скомпилировать Main.cpp с библиотекой `libDynamicModules.{a,so}` для UNIX либо `DynamicModules.lib` для Windows NT:
   (a) Для Linux с компилятором GCC:
   ```
   g++ Main.cpp -ldl [...] libDynamicModules.a
   ```
   (b) Для Windows NT:

```
cl Main.cpp /MD /GR /GX /D "WIN32" [...] /link DynamicModules.lib
```

Запустив полученный таким образом выполняемый файл, получаем следующий результат:

```
Hi! I'm Hello
```

4 API DynamicModule

4.1 Класс DynamicModule

DynamicModule - это абстрактный C++ класс, конкретный экземпляр которого должен определяться в разделяемой бибилиотеке.

Программист должен перегузить в нем следующие методы:

- `const char* name() const` - возвращает имя модуля.
- `const char* author() const` - возвращает имя автора модуля.
- `int versionMajor() const` - возвращает основной номер версии.
- `int versionMinor() const` - возвращает второй номер версии.
- `int versionSubMinor() const` - возвращает третий номер версии.

Несколько слов о нумерации версий: мы в своих продуктах используем трехзначные номера версий, где знаки имеют следующий смысл:

1 - основной номер версии. Он меняется, если мы изменяем API программиста несовместимым образом.

2 - второй номер версии. Он меняется, если мы изменяем бинарное представление или протокол передачи данных (в сетевом случае)

3 - третий номер версии. Изменение третьего номера версии должно сохранять бинарную совместимость.

Заметим, что в случае разделяемых библиотек, на бинарную совместимость влияют размеры классов, так что если Вы добавили в какой-то интерфейсный класс внутреннюю переменную, то вам надо менять *второй номер версии* . Поэтому мы рекомендуем включать в совместно-используемые интерфейсные залоловки только абстрактные классы либо использовать `Pimpl` идиому.

4.2 Тег библиотеки

Один из C++ модулей библиотеки должен содержать глобальную переменную того же типа, что и модуль, с уникальным, среди всех загружаемых главной программой модулей, именем, которое мы рекомендуем строить как `tag<ИмяМодуля>` или `<имяМодуля>Tag`

Как Вы видели в примерах, мы дополнительно пишем

```
EXPORT_OBJECT(tag<ИмяМодуля>)
```

Это синтаксический сахар для сокрытия особенностей Windows API (где Вы должны использовать специальное ключевое слово для экспортирования функциональности библиотеки)

Одна особенность: это определение должно находиться *вне* любого пространства имен C++.

Т. е. ваш файл с определением модуля должен выглядеть следующим образом:

```
 ... includes ..

namespace MyCompany {

....

class MyModule: public DynamicModule
{
 ...
};

} // end of namespace (MyCompany)

MyCompany::MyModule tagMyModule;
```

Кстати, Вам имеет смысл добавлять имена пространств имен (извините за тавтологию!) к именам модулей.

Т. е. может быть лучше поступить так:

```
MyCompany::MyModule tagMyCompany_MyModule;
```

4.3 Класс DynamicModules

Класс DynamicModules - это синглетон, который обеспечивает управление библиотеками.

Основные методы, это:

```
DynamicModules::load(const char* moduleName, const char* fname);
```

- загружает модуль `<fname>.so` (для UNIX), либо `<fname>.dll` (для Windows NT), и запоминает его под именем `moduleName` (внимание: при работе под UNIX `moduleName` должно совпадать с именем той глобальной переменной, которая была экспортирована при помощи макроса `EXPORT_OBJECT`).

```
DynamicModules::unload(const char* moduleName);
```

- уменьшает внутренний счетчик ссылок и выгружает модуль, если этот счетчик ссылок оказывается равным нулю.

После выгрузки модуля, ссылка на этот модуль, возвращенная `load`, становится недействительной, любое ее использование приводит к неопределенному поведению программы.

```
DynamicModules::getModule(const char* moduleName);
```

- то же самое, что и `load` за тем исключением, что когда модуль уже загружен, счетчик ссылок не изменяется.

5 Обработка ошибок

Методы DynamicModules могут генерировать исключения типа `DynamicModules::Error`.

Как обычно, метод what() возвращает строку - описание ошибки. Конкретный перечень возможных сообщений зависит от операционной системы.

6 Заключительные замечания

Эта библиотека предоставляет возможность использовать модель слабо связных компонент на C++. Она более удобна, чем COM или XPCOM для реализации функциональности plugin вследствие своей простоты, с другой стороны она не ограничивает программиста в использовании либо реализации других компонентых моделей - вы можете совмещать, а не выбирать.

Кстати, одна из областей ее применения - реализация выбора между двумя технологиями [в стандартной архитектуре сервисов `GradSoft` вы можете выбирать низлежащий COM или CORBA сервис, просто определив в системе одну из двух библиотек-прослоек].

7 Перечень изменений

24-01-2002
1. пример использования:
   (a) выделен в отдельную секцию;
   (b) упрощен;
   (c) сделан компилируемым;
2. исправлены отдельные ошибки в описании API.

24-09-2001 коррекция опечаток.

21-09-2001 первая полная версия.

15-09-2001 создан.