Gem #110: Программные расширения Plug-ins и библиотеки общего пользования Shared Libraries в Ada - Часть 2

Автор: Pascal Obry, EDF R&D

Краткое содержание: В Gem #110 мы продолжим обсуждение поддержки библиотек общего пользования в GNAT, и поясним, как создавать программные расширения в Ada.

Давайте начнём…

В первой части этой серии из двух частей мы увидели, что автономные разделяемые библиотеки обладают необходимыми свойствами для использования в качестве динамических подключаемых модулей. В этой жемчужине мы исследуем, как динамически загружать и выгружать код в приложении Ada. По сути, динамически загружаемый код компилируется в общую библиотеку, и при изменении этой общей библиотеки она автоматически перезагружается.

Для этого нам понадобятся три модуля:

Registry (Реестр) -- модуль обрабатывает загрузку, выгрузку и регистрацию подключаемых модулей

Computer (Вычислитель) - плагин, выполняющий простое вычисление с использованием двух целых чисел и возвращающий результат

Main -- основное приложение, использующее подключаемый модуль компьютера

Registry

Каждый подключаемый модуль наследует от общего типа Any. Тип ссылки (associated access type) Ref используется для записи всех загруженных плагинов в хэш-карте:

```
package Plugins is
   type Any is abstract tagged null record;
   type Ref is access all Any'Class;
   --  A reference to any loaded plug-in
end Plugins;
```

Наш модуль вычислений (`Computer`) описывается:

```
package Plugins.Computer is

   Service_Name : constant String := "COMPUTER";
   --  Name of the service provided by this plug-in
   type Handle is abstract new Any with null record;
   type Ref is access all Handle'Class;

   function Call
     (H : not null access Handle; A, B : Integer) return Integer is abstract;
end Plugins.Computer;
```

Обратите внимание, что это только абстрактное представление, общее для всех модулей. В этом представлении описываются все процедуры, поддерживаемые подключаемым модулем. Конкретная реализация плагина вычислителя будет дана в модуле Computer.

Спецификация пакета Registry:

```
with Plugins;
package Registry is
```

```
   procedure Discover_Plugins;

   procedure Register
     (Service_Name : String; Handle : not null access Plugins.Any'Class);

   procedure Unregister (Service_Name : String);

   function Get (Service_Name : String) return access Plugins.Any'Class;

end Registry;
```

Процедуры `Register`, `Unregister` и `Get` тривиальны. Ссылка на службу подключаемых модулей записывается в хэш-карту методом `Register` (процедура), удаляется при отмене регистрации методом `Unregister` и извлекается методом `Get`. Эта часть не нуждается в дальнейшем обсуждении.

Процедура `Discover_Plugins` самая сложная. Вот как это работает. Эта процедура проверяет каталог подключаемых модулей на наличие общих библиотек с префиксом "`libplugin_`". Если такое имя найдено, оно переименовывается удалением подстроки "`plugin`" и загружается динамическим компоновщиком (см. `Shared_Lib.Load` процедуру в исходный файл кода registry в архивном zip-файле Gem). При загрузке общей библиотеки код подготавливается и используется для регистрации самого себя (т. е. для регистрации имени службы, связанного со ссылкой на объект) в реестре.

На данный момент сервис доступен и может быть использован для основного приложения.

Обратите внимание, что карта со всеми загруженными общими библиотеками сохраняется. Если обнаруживается, что общая библиотека уже загружена, она сначала выгружается. Это вызывает код завершения, который используется подключаемым модулем для отмены регистрации.

```
   procedure Discover_Plugins is

      function Plugin_Name (Name : String) return String is
         K : Integer := Strings.Fixed.Index (Name, "plugin_");
      begin
         return Name (Name'First .. K - 1) & Name (K + 7 .. Name'Last);
      end Plugin_Name;

      use Directories;
      use type Calendar.Time;

      S          : Search_Type;
      D          : Directory_Entry_Type;
      Only_Files : constant Filter_Type :=
                     (Ordinary_File => True, others => False);
      Any_Plugin : constant String :=
                     "libplugin_*." & Shared_Lib.File_Extension;
   begin
      Start_Search (S, "plugins/", Any_Plugin, Only_Files);
      while More_Entries (S) loop
         Get_Next_Entry (S, D);

         declare
            P     : Shared_Lib.Handle;
            Name  : constant String := Simple_Name (D);
            Fname : constant String := Full_Name (D);
            Pname : constant String := Plugin_Name (Fname);
         begin
```

```
            --  Proceed if plug-in file is older than 5 seconds (we do not want
to try
            --  loading a plug-in not yet fully compiled/linked).
            if Modification_Time (D) < Calendar.Clock - 5.0 then
               Text_IO.Put_Line ("Plug-in " & Name);
               if Loaded_Plugins.Contains (Pname) then
                  Text_IO.Put_Line ("... already loaded, unload now");
                  P := Loaded_Plugins.Element (Pname);
                  Shared_Lib.Unload (P);
               end if;
               --  Rename plug-in (first removing any existing plug-in)
               if Exists (Pname) then
                  Delete_File (Pname);
               end if;
               Rename (Fname, Pname);
               --  Load it
               P := Shared_Lib.Load (Pname);
               Loaded_Plugins.Include (Pname, P);
            end if;
         end;
      end loop;
   end Discover_Plugins;
```

Спецификация `Shared_Lib` поставляется с двумя телами, один для Windows и один для GNU/Linux. Соответствующее тело выбирается автоматически.

Обратите внимание, что переименование плагина требуется в Windows, так как невозможно записать в общую библиотеку, которая используется.

Computer

Вот спецификация модуля Computer, который является реализацией абстрактного типа, описанного выше:

```
with Plugins.Computer;
package Computer is
   type Handle is new Plugins.Computer.Handle with null record;
   overriding function Call
     (H : not null access Handle; A, B : Integer) return Integer;
end Computer;
```

Тело пакета `Computer` простое. Тип `Life_Controller` используется для управления жизнью объекта `Computer`. Процедура `Initialize` (вызывается при загрузке подключаемого модуля) регистрирует имя службы, а процедура `Finalize` (вызывается при выгрузке модуля) отменяет регистрацию имени службы.

```
with Ada.Finalization;
with Registry;
package body Computer is

   use Ada;

   type Life_Controller is new Finalization.Limited_Controlled with null
record;
   overriding procedure Initialize (LC : in out Life_Controller);
   overriding procedure Finalize (LC : in out Life_Controller);

   H : aliased Handle;

   overriding function Call
```

```
      (H : not null access Handle; A, B : Integer) return Integer is
   begin
      return A + B;
   end Call;

   overriding procedure Finalize (LC : in out Life_Controller) is
   begin
      Registry.Unregister (Plugins.Computer.Service_Name);
   end Finalize;

   overriding procedure Initialize (LC : in out Life_Controller) is
   begin
      Registry.Register (Plugins.Computer.Service_Name, H'Access);
   end Initialize;
   LC : Life_Controller;

end Computer;
```

Main

`Main` – самая лёгкая часть. Мы просто зацикливаемся, и раз в секунду, когда обнаруживаем подключаемые модули plug-ins, мы вызываем службу `Computer`, если она найдена.

```
with Ada.Text_IO;
with Plugins.Computer;
with Registry;
procedure Run is
   use Ada;
   use type Plugins.Ref;

   H      : Plugins.Ref;
   Result : Integer;

begin
   loop
      Text_IO.Put_Line ("loop...");
      Registry.Discover_Plugins;
      H := Plugins.Ref (Registry.Get (Plugins.Computer.Service_Name));
      if H /= null then
         Result := Plugins.Computer.Ref (H).Call (5, 7);
         Text_IO.Put_Line ("Result : " & Integer'Image (Result));
      end if;
      delay 1.0;
   end loop;
end Run;
```

Чтобы создать и выполнить программу, распакуйте zip-файл, который присоединён к этой статье Gem, и выполните следующие команды:

```
$ gnat make -p -Pmain

$ gnat make -p -Pcomputer

$ ./run
```

Теперь, пока приложение работает, отредактируйте файл `computer.adb` заменив операцию сложение на умножение в функции `Call`. Затем перекомпилируйте подключаемый модуль plug-in `computer`:

```
$ gnat make -p -Pcomputer
```

Через некоторое время вы увидите, что новый код плагина был загружен автоматически.

Обратите внимание, что расширенный пример, иллюстрирующий динамическую загрузку и выгрузку plug-ins, включён как часть примеров GNAT.

Связанный со статьёй текст программы

plug-in.zip

http://www.adacore.com/uploads_gems/plug-in.zip

Об авторе

Сведений нет.

Last Updated: 11/24/2017
Posted on: 9/21/2011

Обсуждение…