Gem #157: Генерация кода и Gprbuild

Автор: Emmanuel Briot - AdaCore

Краткое содержание: Данная публикация посвящена тому, как настраивать `gprbuild` для вызова генераторов кода перед компиляцией самого кода.

Эта серия следует из Gem #152 и #155, которые мы рекомендуем сначала прочитать в качестве введения.

Давайте начнём…

`Gprbuild` был представлен AdaCore как замена `gnatmake`. Цель по-прежнему состоит в том, чтобы упростить создание целого приложения. Однако, в отличие от `gnatmake`, который изначально инициализирован таким образом, что ограничен языком Ada, `gprbuild` - это мультиязычный инструмент, и он может с радостью запускать компиляторы для различных языков - тех, которые вы используете в своем приложении, а затем связывать все полученные объектные файлы вместе, чтобы создать конечный исполняемый файл.

Работа `Gprbuild` описывается с помощью файла проекта (который использует расширение `.gpr`). В отличие от Unix '`make`' (традиционный инструмент, используемый для управления процессом сборки программы), файл проекта дает статическое описание проекта. Вам не нужно описывать точные команды, которые создаются для перекомпиляции файлов, а также не нужно описывать зависимости ваших источников или указывать, когда объектный файл должен быть воссоздан, потому что некоторые из источников исходных файлов, от которых он зависит, изменились.

Вместо этого сам `gprbuild` обладает знаниями о различных компилируемых языках. Например, он знает, что для Ada объектный файл (расширение `.o`) генерируется GNAT из набора исходных файлов с тем же базовым именем, но с другим расширением (обычно `.ads` или `.adb`). Он также знает, что исходный файл может зависеть от других исходных файлов, и когда любое из них изменяется, объектный файл также должен быть восстановлен.

Благодаря этим встроенным знаниям файл проекта для типичного чистого приложения Ada намного проще, чем эквивалентный `Makefile` (если, конечно, вы не вызываете `gnatmake` или `gprbuild` из этого `Makefile`). Вам нужно только указать его на исходные файлы, а остальное - автоматически. Аналогичная поддержка доступна для языков `C` и `Fortran`.

Однако в настоящее время многие приложения должны сначала генерировать часть своих источников из языков высокого уровня, таких как `UML` или `Simulink`. К счастью, `gprbuild` относительно легко распространить на другие языки, и этот Gem #157 описывает различные шаги, необходимые для этого.

Пользовательский генератор кода

Давайте сначала начнем с описания проблемы.

Предположим, у нас есть собственный генератор кода, который считывает информацию из одного или нескольких файлов XML, а затем генерирует один или несколько файлов Ada из них. Эти сгенерированные файлы Ada должны быть затем скомпилированы вместе с файлами Ada с ручной кодировкой, прежде чем можно будет связать конечный исполняемый файл.

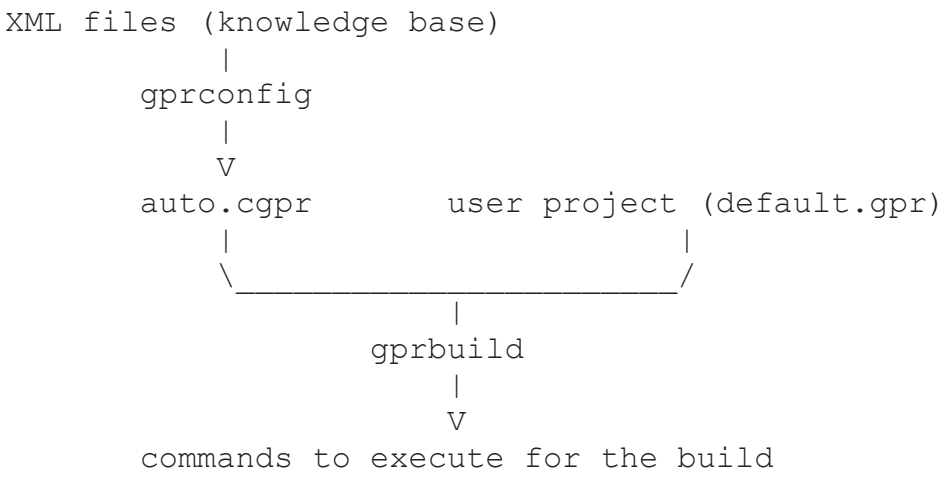
Конечно, для целей оптимизации мы хотели бы выполнить минимальную сумму перекомпиляции, когда XML-файл не был изменен (то есть не регенерировать файлы Ada), или когда никакой файл Ada не был изменен (хотя эта часть уже автоматически обрабатывается gprbuild).

Этот пример будет применяться аналогичным образом при использовании таких генераторов кода, как, например, Lex или Yacc.

Описание генератора кода для gprbuild

Конечно, поскольку это специальный генератор кода, `gprbuild` ничего не знает об этом по умолчанию, и нам нужно все настроить. Это можно сделать непосредственно в файле проекта, но `gprbuild` обеспечивает гибкость сверх этого.

Сам `Gprbuild` не имеет жестко закодированных знаний о языках компиляции. Вместо этого он считывает всю необходимую информацию из файла конфигурации (обычно с расширением `.cgpr`). Файл конфигурации не пишется пользователями, но генерируется из набора файлов XML («базы знаний») с помощью второго инструмента, называемого `gprconfig`. Поведение высокого уровня выглядит следующим образом:



Что нам нужно сделать, это создать новый файл XML для базы знаний.

Давайте оставим язык XML для файлов чистого XML на случай, если приложение содержит файлы, не связанные с генерацией кода. Вместо этого мы «изобрели» новый язык, условно названный «**`xml_for_ada`**». `Gprbuild` должен автоматически найти источники для этого языка (они имеют стандартное расширение `.xml`).

Файл XML будет выглядеть следующим образом:

```
XML_For_Ada
codegen
xml_for_ada
1.0
```

```
package Naming is
   -- How to recognize XML files
   for Body_Suffix ("xml_for_ada") use ".xml";
end Naming;
package Compiler is
   -- describes our code generation, from XML to Ada
```

```
      for Driver ("xml_for_ada") use "codegen";
      for Object_File_Suffix ("xml_for_ada") use ".ads";
      for Object_File_Switches ("xml_for_ada") use ("-o", "");
      for Required_Switches ("xml_for_ada") use ("-g");
        -- always use this switch
      for Dependency_Switches ("xml_for_ada") use ("-M");
        -- -M file.d (indicates the dependency file)
   end Compiler;
```

Первая часть («compiler_description») описывает, как найти исполняемый файл для генератора кода. Его имя - «codegen», и его нужно найти только в том случае, если в проекте пользователя указано, что он использует язык «xml_for_ada». На данный момент номер версии жестко запрограммирован, но можно было бы запросить у самого codegen текущий номер версии, который можно было бы использовать позже, чтобы изменить поддержку gprbuild для него в зависимости от версии.

Предполагая, что генератор кода найден, вторая часть файла XML («configuration») указывает, какой код необходимо добавить в файл конфигурации gprbuild. Вот где происходит волшебство gprbuild.

Во-первых, мы даем gprbuild знать, как распознать наши XML-файлы (расширение «.xml»).Во-вторых, мы говорим, что для обработки этих файлов необходимо вызвать исполняемый файл с именем «codegen», который будет генерировать файлы Ada с расширением «.ads».Для правильной обработки зависимостей (что минимизирует перекомпиляцию), gprbuild необходимо знать имя сгенерированного файла. Для нас это файл с расширением «.ads» (хотя мы могли бы, конечно, создать дополнительные расширения для файлов). Когда для одного файла Ada требуется несколько файлов XML, генератор кода должен сгенерировать файл зависимостей (в основном аналогичный извлечению из Makefile), который указывает список этих файлов. В нашем случае мы решили, что это имя файла зависимости будет передано генератору кода через ключ -M.

Наконец, мы можем решить, что некоторые ключи обязательны для генератора кода, и мы покажем пример с ключом -g.

Пример кода для очень простого генератора кода показан ниже:

```
with Ada.Text_IO; use Ada.Text_IO;
with GNAT.Command_Line; use GNAT.Command_Line;
procedure Codegen is
   F : File_Type;
begin
   loop
      case Getopt ("g M: o:") is
         when 'g' => -- some random switch passed to the compiler
            null;
         when 'M' => -- We need to generate the dependency file.
            Create (F, Out_File, Parameter);
            Put_Line (F, "b.ads: ../src/b.xml");
            Close (F);
         when 'o' => -- Name of the object file
            -- We need to parse the XML file and generate code.
            -- Let's simulate it.
            Create (F, Out_File, Parameter);
            Put_Line (F, "package B is");
            Put_Line (F, "   procedure Foo is null;");
```

```
            Put_Line (F, "end B;");
            Close (F);
         when others =>
            exit;
      end case;
   end loop;
end Codegen;
```

Пользовательский проект

Теперь сложная часть завершена, и мы можем перейти к написанию проекта, использующего этот генератор кода. Поскольку мы предоставили эту информацию в общем виде для `gprbuild`, мы можем создать несколько таких проектов, не дублируя работу, описанную выше.

Настройка следующая:

```
   gprconfig_db/
       Contains the XML file we created above for gprbuild
    src/
```

Этот каталог должен содержать файлы `.xml`, используемые для генерации кода, а также файлы Ada с ручным кодированием. Давайте предположим, что он содержит `b.xml` и `a.adb`

```
   generated/
       This directory will contain the Ada files generated from the XML files.
   obj/
```

Этот каталог будет содержать объектные файлы, полученные в результате компиляции файлов Ada.

`Gprbuild` должен знать все свои источники при запуске, поэтому на практике нам потребуется выполнить два запуска `gprbuild`: один для генерации Ada из XML, а второй для компиляции всех файлов Ada и связывания исполняемого файла. Это означает, что набор исходных файлов отличается на двух этапах: на первом этапе источники - это файлы XML, а «объектные файлы» - это файлы Ada; на втором шаге источники - это файлы Ada, а объектные файлы - это обычные объектные файлы «`.o`».

Мы могли бы реализовать эту настройку с двумя разными файлами проекта, по одному на каждый шаг. Тем не менее, мой собственный предпочтительный подход заключается в использовании одного файла проекта с переменной сценария, которая указывает текущий шаг.

Вот файл проекта:

```
project Default is
   type Compilation_Step is ("Step_1", "Step_2");
   Step : Compilation_Step := External ("STEP", "Step_1");
   case Step is
      when "Step_1" =>
         for Languages use ("xml_for_ada");
         for Source_Dirs use ("src");
         for Object_Dir use "generated";
      when "Step_2" =>
         for Languages use ("Ada");
         for Main use ("a.adb");
         for Source_Dirs use ("src", "generated");
         for Object_Dir use "obj";
   end case;
```

```
end Default;
```

После всех этих настроек сама компиляция выполняется с помощью этих двух простых команд:

```
gprbuild --db gprconfig_db -Pdefault -XSTEP=Step_1
gprbuild -Pdefault -XSTEP=Step_2
```

Когда они запускаются в первый раз, вывод:

```
> gprbuild --db gprconfig_db -Pdefault -XSTEP=Step_1
codegen -g -Mb.d b.xml -o b.ads
> gprbuild -Pdefault -XSTEP=Step_2
gcc -c a.adb
gcc -c b.ads
gprbind a.bexch
gnatbind a.ali
gcc -c b__a.adb
gcc a.o -o a
```

Во второй раз (если со времени первого запуска файл не был изменен), мы получаем ожидаемое:

```
> gprbuild -Pdefault -XSTEP=Step_2
gprbuild: "a" up to date
```

Если мы сейчас изменим b.xml, тогда b.ads будет сгенерирован заново, а затем перекомпилирован, как на первом шаге.

Еще одна хитрость: мы могли бы фактически описать третий шаг (`Step_3`), который будет значением по умолчанию для внешней переменной `STEP`. На этом третьем этапе проект будет содержать языки для `xml_for_ada` и Ada. Назначение этого искусственного третьего шага состояло бы в том, чтобы загрузить этот проект непосредственно в GPS, что позволяет легко редактировать как XML, так и файлы Ada.

Связанный со статьёй текст программы

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Автор: Emmanuel Briot - AdaCore

Эммануэль Брио работает в AdaCore с 1998 года. Он принимал участие в различных проектах, в частности, ориентированных на графические пользовательские интерфейсы, включая `GtkAda`, `GPS`, `XML/Ada`, `GnatTracker` и нашу внутреннюю CRM. Он получил степень инженера в Национальной школе телекоммуникаций (Брест, Франция) (Ecole Nationale des Telecommunications).

Last Updated: 11/24/2017
Posted on: 3/3/2014
Обсуждение…