Gem #114: Протоколирование с помощью функции GNATCOLL.Traces

Автор: Emmanuel Briot, AdaCore

Краткое содержание: Набор компонентов GNAT содержит пакет программ для протоколирования информации в различных текстовых файлах. Дополнительная информация может быть зарегистрирована для каждого протокола, что позволяет лучше понимать поведение приложения в условиях отсутствия программы отладки.

Давайте начнём…

При написании приложений мы часто добавляем операторы `Put_Line` для отладки начальной версии кода. Как только этот код работает, мы удаляем `Put_Line` и переходим к какой-то другой функции кода. Проблема заключается в том, что бывает повторное возникновение ошибки в этой части кода (и особенно, когда об этом сообщает пользователь, и Вы не можете выполнить отладку на его компьютере), Выходные данные трассировки по-прежнему будут полезны, но код который их формирует больше не присутствует в исполняемом файле.

Решение, конечно, существует как вывод трассы в какой-нибудь файл “`log`”, и оставить код формирующий трассу навсегда в исполняемом коде. Файл журнала трассы в конечном итоге станет очень большим, если у вас есть много трасс, и найти необходимую информацию там может быть не так просто. Таким образом, лучшее решение состоит в том, чтобы разделить трассировки на различные группы и иметь возможность отображать только некоторые группы, чтобы ограничить объем информации для просмотра.

Коллекция компонентов GNAT включает пакет `GNATCOLL.Traces`, которые обеспечивают поддержку этого решения. Этот пакет используется в различных продуктах AdaCore, в частности GPS.

Первое, что нужно сделать, это инициализировать модуль. Трассы настраиваются через внешний файл, который по умолчанию называется “`.gnatdebug`", и ищется в текущем каталоге.

```
   with GNATCOLL.Traces;   use GNATCOLL.Traces;
   with User;
   procedure Main is
   begin
      Parse_Config_File;   --  parses default ./.gnatdebug
      User.Proc;
   end Main;
```

Самый простой способ скомпилировать этот код - использовать файл проекта. Например:

```
with "gnatcoll";
project Default is
    for Main use ("main.adb");
end Default;

gprbuild -Pdefault.gpr
```

В остальной части кода мы можем создать поток. Все сообщения журнала отправляются в поток, и вы можете создавать столько Сообщений, сколько хотите. Сообщение журнала всегда будет иметь префикс имени потока, чтобы упорядочить трассировки в файле журнала. Вот как будет выглядеть код Ada:

```
package User is
   Stream1 : constant Trace_Handle := Create ("Stream1");

   procedure Proc is
```

```
   begin
      Trace (Stream1, "Some trace");
   end Proc;
end User;
```

Если вы скомпилируете этот код и запустите его, на самом деле ничего не будет записано. Это потому, что потоки трассировки по умолчанию отключены. Таким образом нам нужно создать файл конфигурации. Его Формат очень прост. В следующем примере демонстрируется ряд его возможностей:

```
+
> log
TRACE1=yes
TRACE2=no
TRACE3=yes > log2
DEBUG.COLORS=yes
```

Первая строка (”+") указывает на то, что мы хотели бы активировать все потоки по умолчанию. Таким образом, файл только с этой строкой уже покажет результат в нашем примере Ada.

Вторая строка (”>log") указывает, куда по умолчанию должен идти вывод потоков. Если этого не указано, то вывод идет в `stdout`. В противном случае можно указать имя файла. Расширенное использование позволяет определить другие типы перенаправления. Например, `GNATCOLL` поставляется по умолчанию с поддержкой перенаправления в `syslog` в Unix. Было бы относительно просто добавить поддержку пользовательских выходных данных, таких как сокет, база данных и т. д.

Следующие три строки настраивают определённые потоки и показывают, как активировать или отключить их. Пятая строка, в частности, перенаправляет один из потоков в другой файл журнала.

Последняя строка в примере активирует одну из дополнительных функций `GNATCOLL.Traces`. Активация "`DEBUG.COLORS`" будет выводить поток, используя различные цвета для различной информации, которая выводится на каждой строке.

Доступны и другие удобные возможности настройки. Например, если вы активируете "`DEBUG.ABSOLUTE_TIME`", каждая строка в файле журнала будет включать время сообщения. Более мощная функция которую вы можете активировать "`DEBUG.STACK_TRACE`", чтобы получить трассировку стека для каждого сообщения (обратите внимание, что трассировка должна быть преобразована через `addr2line`). Ещё один полезный функционал "`DEBUG.COUNT`”, который добавит количество сообщений, выводимых до сих пор в общем и для конкретного потока.

Вот как будет выглядеть файл журнала:

```
[STREAM1] 1/1 Some Trace (09:05:32.323)
[STREAM4] 1/2 Some Other Trace (09:05:33.125)
```

Как мы видели в примере, текст выводится с помощью функции трассировки. Есть две версии этого: та, которую мы видели, которая используется для вывода простого сообщения, и другая версия, которая принимает возникновение исключения в качестве параметра и выводит информацию об этом исключении. Существует также процедура `Assert`, которая выводит сообщение об ошибке, если условие ложно. Ошибка будет отображаться красным цветом, если цвета были активированы, что делает его легко найти в файле журнала.

Подготовка текста сообщения может быть дорогим (например, если текст должен содержать результат вызова функции, или требует много конкатенации строк). Для этого рекомендуется использовать следующий шаблон кодирования:

```
   if Active (Stream1) then
      Trace (Stream1, "Function result was " & Func(...));
   end if;
```

Файл журнала может быть затруднительно читать, когда он становится большим. `GNATCOLL.Traces` предоставляет возможность выделить трассы. Вот полный пример кода, вычисляющего числа Фибоначчи рекурсивно:

```
pragma Ada_05;
with Ada.Text_IO;        use Ada.Text_IO;
with GNATCOLL.Traces;    use GNATCOLL.Traces;

procedure Fibo is
   Me : constant Trace_Handle := Create ("FIBO");

   function Recurse (Num : Positive) return Positive is
      Result : Integer;
   begin
      if Num = 1 or else Num = 2 then
         return 1;
      end if;

      Increase_Indent (Me, "Computing" & Num'Img);
      Result := Recurse (Num - 1) + Recurse (Num - 2);
      Decrease_Indent (Me, "Done =>" & Result'Img);
      return Result;
   end Recurse;

begin
   Parse_Config_File;
   Put_Line (Recurse (5)'Img);
end Fibo;
```

Код должен быть скомпилирован с файлом проекта, как показано ранее. Текущий каталог должен содержать файл с именем “`.gnatdebug`"с одной строкой, содержащей "+". Выходные данные будут поступать в `stdout`:

```
[FIBO] 1/1 Computing 5
   [FIBO] 2/2 Computing 4
      [FIBO] 3/3 Computing 3
      [FIBO] 4/4 Done => 2
   [FIBO] 5/5 Done => 3
   [FIBO] 6/6 Computing 3
   [FIBO] 7/7 Done => 2
[FIBO] 8/8 Done => 5
 5
```

с формированием отступа, который увеличивается для каждого рекурсивного вызова.

`GNATCOLL.Traces` использует объектно-ориентированный код. Он предоставляет ряд специальных функций типа "hooks", через которые вы можете добавлять свои собственные дополнительные данные каждый раз, когда какая-то строка выводится в файл журнала. Для этого необходимо переопределить операции примитивов `Pre_Decorator` или `Post_Decorator`.

Связанный со статьёй текст программы

Attached Files отсутствуют

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

Об авторе

Сведений об авторе нет.

Last Updated: 10/13/2017
Posted on: 2/1/2012

Обсуждение…