Gem #100: Подсчёт ссылок в Ada - Часть 3 Слабые ссылки

Автор: Emmanuel Briot, AdaCore

Краткое содержание: В данной серии из трёх публикаций описывается возможная реализация автоматического управления памятью с помощью подсчёта ссылок. В части 1 объясняется, как применять контролируемый тип для реализации автоматического подсчёта ссылок, а также описываются некоторые аспекты правильной обработки подсчёта. В части 2 будут проанализированы некоторые проблемы, связанные с многозадачностью. В части 3 будет описана реализация подсчёта слабых ссылок.

Давайте начнём…

Как мы уже упоминали в первых двух частях этой серии Gem, GNATCOLL теперь включает пакет, который обеспечивает поддержку управления памятью с помощью подсчёта ссылок, включая использование эффективной синхронизированной встроенной функции надстроек и выборки в системах, где она доступна.

Есть одна вещь, которую типы с подсчётом ссылок не могут обрабатывать так же, как полномасштабный сборщик мусора: это косвенные ссылки которые образуют петли – циклы. Если `A` ссылается на `B`, который ссылается на `A`, ни один из них никогда не освободится. Сборщик мусора часто в состоянии обнаружить такие петли – циклы и освободить все объекты соответствующим образом, но такой случай не может быть обработан автоматически посредством подсчёта ссылок. Тем не менее, есть вариант подхода, который может обрабатывать такие случаи только с незначительными изменениями в коде.

Давайте рассмотрим пример: вы извлекаете значения из некоторого контейнера (например, базы данных) и хотите иметь локальный кэш, чтобы ускорить процесс. Код, вероятно, будет организован следующим образом:

- Получить значение подсчёта ссылок из контейнера. Его счётчик 1.
- Поместить его в кэш для дальнейшего использования. Счётчик теперь равен 2, так как сам кэш владеет ссылкой.
- Когда вы закончите использовать значение в своём алгоритме, вы отпустите ссылку, которая у вас была. Его счётчик снижается до 1 (кэш по-прежнему владеет ссылкой).

Из-за кэша значение никогда не освобождается из памяти. Это не хорошо, так как использование памяти будет только увеличиваться.

GNATCOLL предоставляет решение этой проблемы с помощью слабых ссылок. Это стандартный отраслевой термин для особого вида ссылок: у вас есть Тип, который указывает на тот же объект, что и истинный Тип с подсчётом ссылок, но этот тип не содержит ссылку. Таким образом, это не мешает счётчику достичь 0, а объекту освободиться.

Когда происходит освобождение, внутренние данные слабой ссылки сбрасываются. Таким образом, если вы извлекаете данные, хранящиеся в слабой ссылке, вы получаете `null`, а не ошибочный доступ к некоторой освобождённой памяти (что может рано или поздно привести к `Storage_Error`).

Если мы настроим кэш так, что он будет использовать слабые ссылки, код станет:

- Получить значение подсчёта ссылок из контейнера. Его счётчик 1.
- Поместить его в кэш, через слабую ссылку. Счётчик по-прежнему 1.

- Когда вы закончите использовать значение, счётчик опустится до 0, и память освободится.

- На данный момент кэш все ещё содержит слабую ссылку, но последняя использует немного памяти.

Используя немного более сложный код, можно, по сути, полностью удалить запись для кэша, когда значение будет освобождено, тем самым действительно освободив всю память для системы. Хотя GNATCOLL предоставляет возможность использования слабых ссылок, будущий пакет облегчит обработку таких кэшей.

Один из способов реализации слабых ссылок – добавление дополнительного указателя в Тип `Refcount`. GNATCOLL делает это необязательным: если вы хотите систематически иметь этот дополнительный указатель в структуре данных, Вы можете использовать слабые ссылки. В противном случае, вы все ещё имеете доступ к коду, который мы описали в первой части этой серии Gem.

Мы не будем вдаваться в детали реализации для слабой ссылки. Желающие могут посмотреть на код в `GNATCOLL.Refcount.Weakref`, который относительно небольшой.

Связанный со статьёй текст программы

Attached Files C:\GNAT\2018\include\gnatcoll

```
------------------------------------------------------------------------------
--                              G N A T C O L L                             --
--                                                                          --
--                     Copyright (C) 2010-2017, AdaCore                     --
--                                                                          --
-- This library is free software;  you can redistribute it and/or modify it --
-- under terms of the  GNU General Public License  as published by the Free --
-- Software  Foundation;  either version 3,  or (at your  option) any later --
-- version. This library is distributed in the hope that it will be useful, --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE.                            --
--                                                                          --
--                                                                          --
--                                                                          --
--                                                                          --
--                                                                          --
-- You should have received a copy of the GNU General Public License and    --
-- a copy of the GCC Runtime Library Exception along with this program;     --
-- see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see    --
-- <http://www.gnu.org/licenses/>.                                          --
--                                                                          --
------------------------------------------------------------------------------

--  By definition, an object can never be freed while there are references
--  to it.
--  However, this simple scheme fails in some cases. For instance, imagine
--  you want to cache some refcounted type into a map. The map would then
--  own a reference to the object, which is thus never freed while the map
--  exists (presumably for the life of your application).
--  A solution to this problem is the notion of "weak reference": these act
--  as containers that point to the element, without owning a reference to
--  them. When the element is destroyed (because its refcount can now reach
--  0), the container is set to a special state that indicates the element
--  no longer exists.
--  With this scheme, the cache will still contain entries for the elements,
--  but those entries will return a Null_Ref when accessed.

package GNATCOLL.Refcount.Weakref is
```

```
39.    pragma Obsolescent (Weakref, "Use GNATCOLL.Refcount.Shared_Pointers");
40.
41.    type Weak_Refcounted
42.      is abstract new GNATCOLL.Refcount.Refcounted with private;
43.    --  A special refcounted type, which can manipulate weak references
44.
45.    overriding procedure Free (Self : in out Weak_Refcounted);
46.    --  If you need to override this procedure in your own code, you need to
47.    --  make sure you correctly call this inherited procedure.
48.
49.    type Proxy is new GNATCOLL.Refcount.Refcounted with record
50.       Proxied : Refcounted_Access;
51.    end record;
52.    package Proxy_Pointers is new Smart_Pointers (Proxy);
53.    --  An internal, implementation type.
54.    --
55.    --  A weak ref acts as a smart pointed with two level of indirection:
56.    --      type My_Type is new GNATCOLL.Refcount.Weakref.Refcounted with ...;
57.    --      package P is new Weakref_Pointers (My_Type);
58.    --      R  : P.Ref;
59.    --      WR : P.Weak_Ref;
60.    --  R now takes care of the reference counting for R.Data.
61.    --  R.Data is an access to My_Type, freed automatically.
62.    --
63.    --  WR now takes care of the reference counting for a Proxy, whose Proxied
64.    --  is set to R.Data. This does not hold a reference to R.Data. However,
65.    --  R.Data holds a reference to the proxy.
66.    --  As a result, the proxy is never freed while R.Data exists. But the
67.    --  latter can be destroyed even when the proxy exists.
68.
69.    generic
70.       type Encapsulated is abstract new Weak_Refcounted with private;
71.    package Weakref_Pointers is
72.       package Pointers is new Smart_Pointers (Encapsulated);
73.       subtype Encapsulated_Access is Pointers.Encapsulated_Access;
74.
75.       subtype Ref is Pointers.Ref;
76.       Null_Ref : constant Ref := Pointers.Null_Ref;
77.
78.       procedure Set (Self : in out Ref; Data : Encapsulated'Class)
79.         renames Pointers.Set;
80.       procedure Set (Self : in out Ref; Data : access Encapsulated'Class)
81.         renames Pointers.Set;
82.       function Get (P : Ref) return Encapsulated_Access
83.          renames Pointers.Get;
84.       function "=" (P1, P2 : Ref) return Boolean
85.         renames Pointers."=";
86.       function "=" (P1, P2 : Pointers.Encapsulated_Access) return Boolean
87.         renames Pointers."=";
88.       --  The manipulation of the smart pointers
89.
90.       subtype Weak_Ref is Proxy_Pointers.Ref;
91.       Null_Weak_Ref : constant Weak_Ref := Weak_Ref (Proxy_Pointers.Null_Ref);
92.       function "=" (P1, P2 : Weak_Ref) return Boolean
93.         renames Proxy_Pointers."=";
94.
95.       function Get_Weak_Ref (Self : Ref'Class) return Weak_Ref;
96.       --  Return a weak reference to Self.
97.       --  It does not hold a reference to Self, which means that Self could be
98.       --  destroyed while the weak reference exists. However, this will not
99.       --  result
100.      --  in a Storage_Error when you access the reference.
101.
102.      function Was_Freed (Self : Weak_Ref'Class) return Boolean;
103.      --  True if the weakly referenced element was freed (thus Get would
104.      --  return Null_Ref). It is more efficient to use this function than
```

```
      --  compare the result of Get with Null_Ref, since the latter will need
      --  to play with refcounting.

      function Get (Self : Weak_Ref'Class) return Ref;
      procedure Get (Self : Weak_Ref'Class; R : out Ref'Class);
      --  Return the weakly referenced object. This will return Null_Ref
      --  if the object has already been destroyed.
      --  The procedure version can be used if you have subclassed Ref.
      --  The code should look like:
      --
      --      --  Create the smart pointer
      --      Tmp : Refcounted_Access := new My_Refcounted_Type;
      --      R   : Ref := Allocate (Tmp);  --  Hold a ref to Tmp
      --
      --      WRef := Get_Weak_Ref (R);  -- Does not hold a ref to Tmp
      --
      --      R := Null_Ref;   --  Releases ref to Tmp, and free Tmp
      --      we now have Get (WRef) = null
      --
      --  In the case of a multitasking application, you must write your code
      --  so that the referenced type is not freed while you are using it. For
      --  instance:
      --      declare
      --         R : constant Ref := Get (WRef);  --  hold a ref to Tmp
      --      begin
      --         if R /= Null_Ref then
      --             ... manipulate R
      --             Tmp cannot be freed while in the declare block, since we
      --             own a reference to it
      --         end if;
      --      end;
   end Weakref_Pointers;

private

   type Weak_Refcounted
     is abstract new GNATCOLL.Refcount.Refcounted
   with record
      Proxy : Proxy_Pointers.Ref;  --  Hold a reference to a proxy
   end record;

end GNATCOLL.Refcount.Weakref;
```

```
------------------------------------------------------------------------------
--                              G N A T C O L L                             --
--                                                                          --
--                     Copyright (C) 2010-2017, AdaCore                     --
--                                                                          --
-- This library is free software;  you can redistribute it and/or modify it --
-- under terms of the  GNU General Public License  as published by the Free --
-- Software  Foundation;  either version 3,  or (at your  option) any later --
-- version. This library is distributed in the hope that it will be useful, --
-- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE.                            --
--                                                                          --
--                                                                          --
--                                                                          --
--                                                                          --
--                                                                          --
-- You should have received a copy of the GNU General Public License and    --
-- a copy of the GCC Runtime Library Exception along with this program;     --
-- see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see    --
-- <http://www.gnu.org/licenses/>.                                          --
--                                                                          --
------------------------------------------------------------------------------

```

```
24. pragma Ada_05;
25. with GNATCOLL.Atomic;   use GNATCOLL.Atomic;
26.
27. package body GNATCOLL.Refcount.Weakref is
28.    use Proxy_Pointers;
29.
30.    ----------
31.    -- Free --
32.    ----------
33.
34.    overriding procedure Free (Self : in out Weak_Refcounted) is
35.    begin
36.       if Self.Proxy /= Proxy_Pointers.Null_Ref then
37.          Proxy (Get (Self.Proxy).all).Proxied := null;
38.          Self.Proxy := Proxy_Pointers.Null_Ref;
39.       end if;
40.       Free (Refcounted (Self));   --  ??? static call to a "null" procedure
41.    end Free;
42.
43.    ----------------------
44.    -- Weakref_Pointers --
45.    ----------------------
46.
47.    package body Weakref_Pointers is
48.       use Pointers;
49.
50.       ------------------
51.       -- Get_Weak_Ref --
52.       ------------------
53.
54.       function Get_Weak_Ref (Self : Ref'Class) return Weak_Ref is
55.          Data : constant Encapsulated_Access := Self.Get;
56.          P    : Proxy_Pointers.Ref;
57.          D    : Proxy_Pointers.Encapsulated_Access;
58.       begin
59.          if Data = null then
60.             return Null_Weak_Ref;
61.          end if;
62.
63.          P := GNATCOLL.Refcount.Weakref.Weak_Refcounted'Class (Data.all).Proxy;
64.          if P = Proxy_Pointers.Null_Ref then
65.             D := new Proxy'(GNATCOLL.Refcount.Refcounted
66.                             with Proxied => Refcounted_Access (Data));
67.             Set (P, D);  --  now owns a reference to D
68.             Weak_Refcounted'Class (Data.all).Proxy := P;
69.          end if;
70.
71.          return Weak_Ref (P);
72.       end Get_Weak_Ref;
73.
74.       ---------------
75.       -- Was_Freed --
76.       ---------------
77.
78.       function Was_Freed (Self : Weak_Ref'Class) return Boolean is
79.          P : constant access Proxy :=
80.                Proxy_Pointers.Get (Proxy_Pointers.Ref (Self));
81.       begin
82.          return P = null or else P.Proxied = null;
83.       end Was_Freed;
84.
85.       ---------
86.       -- Get --
87.       ---------
88.
89.       procedure Get (Self : Weak_Ref'Class; R : out Ref'Class) is
```

```
90.          P : constant access Proxy :=
91.                Proxy_Pointers.Get (Proxy_Pointers.Ref (Self));
92.       begin
93.          if Was_Freed (Self) then
94.             R.Set (null);
95.          else
96.             --  A subtetly here: it is possible that the element is actually
97.             --  being freed, and Free() is calling Get on one of the weakref.
98.             --  In such a case, we do not want to resuscitate the element
99.
100.            if P.Proxied.Refcount = 0 then
101.               R.Set (null);
102.            else
103.               --  Adds a reference to P.Proxied
104.               R.Set (Encapsulated_Access (P.Proxied));
105.            end if;
106.         end if;
107.      end Get;
108.
109.      ---------
110.      -- Get --
111.      ---------
112.
113.      function Get (Self : Weak_Ref'Class) return Ref is
114.         Result : Ref;
115.      begin
116.         Get (Self, Result);
117.         return Result;
118.      end Get;
119.
120.   end Weakref_Pointers;
121.
122. end GNATCOLL.Refcount.Weakref;
```

```
1. ------------------------------------------------------------------------------
2. --                             G N A T C O L L                              --
3. --                                                                          --
4. --                     Copyright (C) 2010-2017, AdaCore                     --
5. --                                                                          --
6. -- This library is free software;  you can redistribute it and/or modify it --
7. -- under terms of the  GNU General Public License  as published by the Free --
8. -- Software  Foundation;  either version 3,  or (at your  option) any later --
9. -- version. This library is distributed in the hope that it will be useful, --
10. -- but WITHOUT ANY WARRANTY;  without even the implied warranty of MERCHAN- --
11. -- TABILITY or FITNESS FOR A PARTICULAR PURPOSE.                           --
12. --                                                                         --
13. --                                                                         --
14. --                                                                         --
15. --                                                                         --
16. --                                                                         --
17. -- You should have received a copy of the GNU General Public License and   --
18. -- a copy of the GCC Runtime Library Exception along with this program;    --
19. -- see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see   --
20. -- <http://www.gnu.org/licenses/>.                                         --
21. --                                                                         --
22. ------------------------------------------------------------------------------
23.
24. --  This package provides a number of low-level primitives to execute
25. --  task-safe operations.
26. --  When possible, these operations are executed via one of the intrinsic
27. --  atomic operations of the compiler (generally implemented with special
28. --  support from the CPU).
29.
30. with System.Atomic_Counters;
31.
32. package GNATCOLL.Atomic is
```

```
33.
34.    subtype Atomic_Counter is System.Atomic_Counters.Atomic_Unsigned;
35.
36.    Minus_One : constant Atomic_Counter :=
37.       System.Atomic_Counters."-" (0, 1);
38.
39.    function Is_Lock_Free return Boolean;
40.    --  Whether the implementation uses the processor's atomic operations
41.    --  or falls back on using locks
42.
43.    function Sync_Add_And_Fetch
44.      (Ptr   : access Atomic_Counter;
45.       Value : Atomic_Counter) return Atomic_Counter
46.      with Inline_Always;
47.    --  Increment Ptr by Value. This is task safe (either using a lock or
48.    --  intrinsic atomic operations). Returns the new value (as set, it
49.    --  might already have been changed by another task by the time this
50.    --  function returns.
51.
52.    function Sync_Sub_And_Fetch
53.      (Ptr   : access Atomic_Counter;
54.       Value : Atomic_Counter) return Atomic_Counter
55.      with Inline_Always;
56.    --  Decrement Ptr by Value.
57.
58.    procedure Sync_Add_And_Fetch
59.      (Ptr : access Atomic_Counter; Value : Atomic_Counter)
60.      with Inline_Always;
61.    procedure Sync_Sub_And_Fetch
62.      (Ptr : access Atomic_Counter; Value : Atomic_Counter)
63.      with Inline_Always;
64.    --  Same as above, but ignore the return value.
65.
66.    procedure Increment
67.       (Value : aliased in out Atomic_Counter) with Inline_Always;
68.    procedure Decrement
69.       (Value : aliased in out Atomic_Counter) with Inline_Always;
70.    function Decrement
71.       (Value : aliased in out Atomic_Counter) return Boolean
72.       with Inline_Always;
73.    --  Similar to the Sync_Add_And_Fetch and Sync_Sub_And_And, but
74.    --  always increment or decrement by one.
75.    --  On some systems (x86) this uses faster assembly instructions.
76.    --  Decrement returns True if the value reaches 0.
77.
78.    function "+"
79.       (Left, Right : Atomic_Counter) return Atomic_Counter is abstract;
80.    function "-"
81.       (Left, Right : Atomic_Counter) return Atomic_Counter is abstract;
82.    --  Prevent standard operations on these counters
83.
84.    function Unsafe_Decrement (Value : in out Atomic_Counter) return Boolean
85.       with Inline_Always;
86.    procedure Unsafe_Increment (Value : in out Atomic_Counter)
87.       with Inline_Always;
88.    --  These are unsafe operations. If you have two threads, and they all try
89.    --  to do "Unsafe_Add (A, 2)" at the same time, when A was initially 0,
90.    --  you could end up with the following values in A:
91.    --      2 (both threads have read 0, then added 2)
92.    --      4 (thread 1 has read and incremented, then thread 2)
93.    --  If you use the other operations above, you always end up with 4.
94.
95.    function "="
96.       (Left, Right : Atomic_Counter) return Boolean
97.       renames System.Atomic_Counters."=";
98.    --  Make the operator visible
```

```
 99.
100.    generic
101.       type Element_Type (<>) is limited private;
102.       type Element_Access is access Element_Type;
103.    function Sync_Bool_Compare_And_Swap
104.       (Ptr    : access Element_Access;
105.        Oldval : Element_Access;
106.        Newval : Element_Access) return Boolean;
107.    --  If Ptr is equal to Oldval, set it to Newval and return True.
108.    --  Otherwise, return False and do not modify the current value.
109.    --  This operation is task safe and atomic.
110.
111.    function Sync_Bool_Compare_And_Swap_Counter
112.       (Ptr    : access Atomic_Counter;
113.        Oldval : Atomic_Counter;
114.        Newval : Atomic_Counter) return Boolean;
115.    function Sync_Val_Compare_And_Swap_Counter
116.       (Ptr    : access Atomic_Counter;
117.        Oldval : Atomic_Counter;
118.        Newval : Atomic_Counter) return Atomic_Counter;
119.    --  A version that works with Atomic_Counter.
120.    --  Ptr.all is set to Newval if and only if it is currently set to Oldval.
121.    --  Returns True if the value was changed.
122.    --  The second version returns the initial value of Ptr.all
123.
124. end GNATCOLL.Atomic;
```

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

Об авторе

Сведений нет.

Last Updated: 10/13/2017
Posted on: 2/28/2011

Обсуждение…