Gem #78: Куда делась моя память? (Часть 2)

Автор: Emmanuel Briot, AdaCore

Краткое содержание: Gem Ada #78 - Для контроля за использованием памяти, обнаружением утечек памяти и разрешения проблем общего характера, связанных с памятью, существует множество инструментов и библиотек. Эта серия из трёх публикаций содержит обзор этой тематики и описание работы с подобными инструментами.

Давайте начнём…

Если ваш стандарт кодирования не запрещает динамическое выделение, управление памятью является постоянной проблемой при разработке системы. Может потребоваться ограничить объем памяти, необходимый приложению, или могут возникнуть утечки памяти (блоки выделения, которые никогда не возвращаются в систему). Последнее является критической проблемой для долговременных приложений.

Часть II: Системная память (`System.Memory package`)

В предыдущем Gem мы обсуждали использование `GNAT.Debug_Pools` для обнаружения проблем с памятью. Другой подход, который несколько проще в использовании, поскольку он не требует изменений в приложении, заключается в переопределении пакета `System.Memory`. В GNAT и его run-time все фактические распределения памяти выполняются через этот пакет, который, среди прочего, низкоуровневая система вызывает `malloc ()` и `free ()`.

Если вы создадите собственную версию `System.Memory`, вы на самом деле коротко замыкаете все выделение и освобождение памяти и замените её на свою собственную версию System.Memory. Для этого скопируйте файл `s-memory.adb` в один из ваших исходных каталогов, измените его соответствующим образом и скомпилируйте своё приложение, передав «`-a`» переключатель в `gnatmake` (`gptbuild` в настоящее время не имеет эквивалентного переключателя, хотя используется файлы проекта должны работать, как ожидалось). Это гарантирует, что GNAT перекомпилит библиотеку с вашей модифицированной Системой.

Использование этого пакета не обеспечивает такую же безопасность, что и пулы отладки, поскольку он не проверяет правильность разметки, поэтому ваш код все равно может обращаться к недопустимой памяти. С другой стороны, использование `System.Memory` намного менее навязчиво в вашем коде. `System.Memory` лучше всего рассматривать как инструмент анализа производительности, а не инструмент отладки, хотя он позволит вам отслеживать ваш код для утечек памяти.

Библиотека `GNATCOLL` (последнее дополнение к технологии GNAT и часть последних выпусков клиентов и общественности) обеспечивает такую реализацию в виде пакета `GNATCOLL.Memory`. Этот пакет не является прямой заменой `System.Memory`, но для его использования требуется лишь минимальная работа, создав версию файла `s-memory.adb`, которая содержит следующее:

```
with GNATCOLL.Memory;
package body System.Memory is

   package M renames GNATCOLL.Memory;

   function Alloc (Size : size_t) return System.Address is
   begin
      return M.Alloc (M.size_t (Size));
   end Alloc;
```

```
   procedure Free (Ptr : System.Address) renames M.Free;

   function Realloc
      (Ptr  : System.Address;
       Size : size_t)
      return System.Address
   is
   begin
      return M.Realloc (Ptr, M.size_t (Size));
   end Realloc;

end System.Memory;
```

Затем вам необходимо изменить свой код, чтобы он правильно инициализировал `GNATCOLL.Memory`, который выполняется с помощью вызова, подобного следующему:

```
GNATCOLL.Memory.Configure (Activate_Monitor => True);
```

Мониторинг, предоставляемый этим пакетом `GNATCOLL`, по умолчанию не включён, чтобы ограничить накладные расходы на запущенной программе. В вашем приложении мониторинг можно активировать с помощью командной строки или с помощью определённой переменной среды. (Это все полностью под вашим контролем, поэтому, вам нужно будет сделать фактический вызов `Getenv`, а затем настроить.)

Затем вы можете использовать код в одном или нескольких местах, чтобы сбрасывать использование памяти в этот момент. Это делается путём вызова, например:

```
GNATCOLL.Memory.Dump (Size => 3, Report => Memory_Usage);
```

Такой вызов выведет на консоль три обратных пути для кода, выделившего наибольший объем памяти (среди выделенной в данный момент памяти). Существуют варианты, чтобы сбросить обратные пути, которые выполнили наибольшее количество выделений (в отличие от наибольшего размера выделения), или общий объем памяти, даже если эта память с тех пор была освобождена.

Такой дамп включает в себя `backtrace` с адресами, которые можно преобразовать в символьный `backtrace` с помощью внешнего инструмента `addr2line` следующим образом:

```
addr2line -e
```

Эта библиотека имеет очень лёгкие накладные расходы (в частности, когда она не активирована), чтобы вы могли распространять своё приложение с поддержкой `GNATCOLL.Memory`, а затем исследовать любые проблемы, возникающие в производственном коде. Фактически, наша собственная система GPS IDE теперь включает эту поддержку. Активация контролируется внешней переменной, а сброс состояния памяти может выполняться с помощью команды `Python` в любой момент времени без необходимости перекомпиляции GPS. (Обратите внимание, что `GNATCOLL` также предоставляет интерфейс для `Python`.)

Еще одна особенность `GNATCOLL.Memory` - возможность сброса всех счётчиков до нуля. Например, предположим, что мы хотим исследовать утечки памяти при открытии и закрытии редакторов в GPS. Если мы посмотрим на места, которые выделяют память, самые большие распределения, отображаемые на консоли, не относятся к самому редактору, а скорее к памяти, выделенной при запуске GPS (если вам интересно, это, как правило, память, связанная с кросс-справочная база данных). Поэтому мы бы сделали следующее: запустите GPS, сбросьте счётчики `GNATCOLL.Memory` до 0, откройте и закройте редактор, и используйте `Dump` памяти. В этот момент, если `Dump` печатает любую информацию на консоли, мы знаем, что это память, которая была

выделена после вызова `Reset`, и которая не была освобождена при закрытии редактора и, следовательно, скорее всего, была утечка памяти.

Соответствующее использование `Reset` и `Dump` позволяет контролировать использование памяти в определённых частях кода.

Отдельный инструмент, называемый `gnatmem`, также распространяется вместе с GNAT. Когда вы связываете приложение с ключом `-lgmem`, он будет прозрачно отображать все вызовы на стандартный `malloc` и `free` (из кода Ada), подобно `GNATCOLL.Memory`. На выходе программы создаётся файл на диске, который затем можно проанализировать с помощью `gnatmem`, чтобы выделить источники утечек памяти в вашем приложении. Этот инструмент не требует каких-либо изменений в вашем коде, но, с другой стороны, не предоставляет возможности отслеживать определённые разделы вашего кода, такие как `GNATCOLL.Memory`.

`Gnatmem` предоставляет несколько переключателей командной строки для управления отображением информации. Например, переключатель «`-m 0`» позволяет просматривать все места в коде, который когда-либо выделял память, даже если эта память была правильно освобождена после этого. Когда одно из таких мест делает миллионы распределений, иногда может быть более эффективным использование специального `Storage_Pool` и избежать системного вызова в `malloc` путём повторного использования памяти. Переключатель «`-s`» позволяет сортировать выходные данные различными способами.

В третьей части этой серии мы рассмотрим различные коммерчески доступные системные инструменты для мониторинга и анализа использования памяти.

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: 1/25/2010

Обсуждение…