Gem #103: Архетипы кода программирования в реальном времени - Часть 5. Связь между задачами-пример производитель-потребитель

Автор: Marco Panunzio, University of Padua

Краткое содержание: В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определённым образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

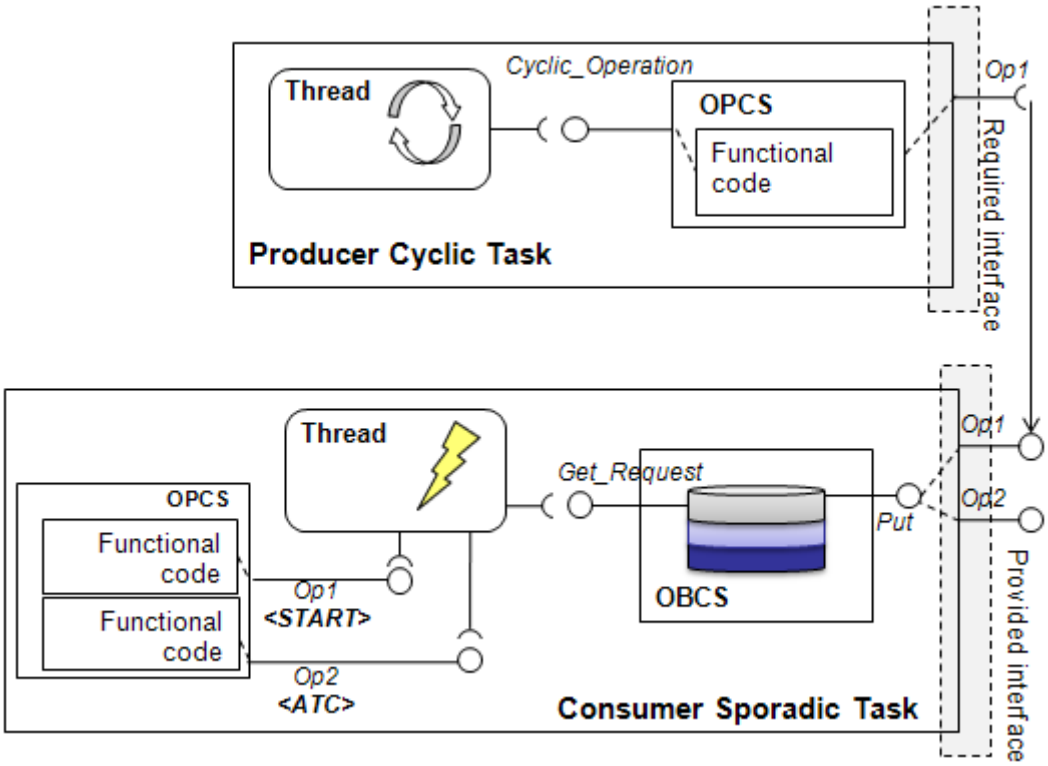
Введение.

В предыдущем Ada Gem мы завершили создание полной спорадической задачи. В этом Ada Gem, который заканчивает нашу мини-серию, мы хотим завершить этот пример, добавив реализацию связи между различными задачами. В частности, мы исследуем, как мы можем правильно управлять вызовами операций вне задачи. Эти вызовы выполняются с OPCS, и их необходимо правильно маршрутизировать в конечную точку связи.

Давайте начнём…

Связь между задачами-пример производитель-потребитель

Предположим, мы хотим реализовать простую модель сотрудничества между производителем и потребителем, изображённую на рисунке ниже.



Продюсер (Producer) представляет собой циклическую задачу, которая после некоторой обработки создаёт некоторые данные, которые отправляются в спорадическую задачу Потребителя (Consumer). Данные передаются как параметр операции `Op1`.

Это означает, что мы должны оборудовать циклическую задачу Продюсера (Producer) средствами общения с спорадической задачей Потребителя (Consumer). Однако созданная нами структура задач инкапсулирует функциональный код внутри структуры, называемой OPCS. Поэтому внутри функционального кода Продюсера (Producer) мы не можем напрямую вызвать `Op1`

Потребителя (Consumer) (то есть функциональный / последовательный код), но мы должны вызвать соответствующий предоставленный интерфейс для всей задачи Потребителя (Consumer).

Давайте посмотрим, как мы можем достичь этой цели при создании пакета `Producer` (аналогично пакету `Consumer`, который мы представили в предыдущем Ada Gem серии).

```
package Producer is
   type Producer_FC is new Controlled with private;
   type Producer_FC_Ref is access all Producer_FC'Class;
   type Producer_FC_Static_Ref is access all Producer_FC;
   -- [code omitted]
   overriding
   procedure Initialize(This : in out Producer_FC);
   procedure Op0 (This : in out Producer_FC);
   procedure Set_x(This : in out Producer_FC;
                   v : in Consumer.Consumer_FC_Ref);
private
   type Producer_FC is new Controlled with record
      x : Consumer.Consumer_FC_Ref;
   end record;
end Producer;
```

Мы добавляем элемент `x` в запись `Producer_FC`, который представляет ссылку на Потребителя, который предоставляет `Op1`, который потребляет данные, производимые Продюсером. Читатель должен отметить, что статическим типом этой ссылки является OPCS Потребителя (который назывался `Consumer_FC`).

```
package body Producer is
   -- [procedure Initialize omitted]
   procedure Op0(This : in out Producer_FC) is
   begin
      This.x.Op1([T1_VALUE]);
   end Op1;

   procedure Set_x(This : in out Producer_FC;
                   v : in Consumer.Consumer_FC_Ref)
   is
   begin
      This.x := v;
   end Set_x;
end Producer;
```

В теле процедуры `Op0` (где задан последовательный код, выполняемый заданием Продюсера), мы вставляем вызов в `Op1`, который выполняется с использованием ссылки `x`.

```
-- Package spec
type s1_T is new Consumer.Consumer_FC with record
   Op1_Ref : access procedure (a : in Types.T1; b : in Types.T2);
   Op2_Ref : access procedure (a : in Types.T1);
end record;
overriding
procedure Op1(This : in out s1_T; a : in Types.T1; b : in Types.T2);
overriding
procedure Op2(This : in out s1_T; a : in Types.T1);
-- Package body
procedure Op1(This : in out s1_T; a : in Types.T1; b : in Types.T2) is
begin
   This.Op1_Ref.all(a,b);
end Op1;
```

Выше, в другом пакете, мы создаём новый тип `s1_T`. Этот тип расширяет `Consumer_FC` и добавляет указатель на процедуру с сигнатурой `Op1`, операцию, которую мы вызываем на стороне Продюсера, и `Op2`. Мы также переопределяем `Op1` для `s1_T`, так что вызов `Op1` переиздаётся только что определённому указателю.

Аналогично, предположим, что мы создаём новый тип `s0_T`, который расширяет `Producer_FC`.

В нескольких оставшихся фрагментах кода, приведённых ниже, мы приводим пример с созданием циклической задачи для Продюсера и спорадической задачей для Потребителя.

```
s0_Instance : aliased s0_T;
package My_Cyclic_Producer_Task is new
     Op0_Cyclic_Producer.My_Sporadic_Factory(
           Thread_Priority => 1,
           Period => 2000,
           OPCS_Instance =>
               Producer.Producer_FC(s0_Instance)'access);
s1_Instance : aliased s1_T;
package My_Sporadic_Consumer_Task is new
     Op1_Op2_Sporadic_Consumer.My_Sporadic_Factory(
           Thread_Priority => 2,
           Ceiling => 2,
           MIAT => 500,
           OPCS_Instance =>
               Consumer.Consumer_FC(s1_Instance)'access);
```

Обратите внимание, что мы передаём указатель на `s0_Instance` (соответственно `s1_Instance`) в качестве OPCS во время создания циклической (соответственно спорадической) задачи производителя (соответственно потребителя).

```
s1_Instance.Op1_Ref := My_Sporadic_Consumer_Task.Op1'access;
s1_Instance.Op2_Ref := My_Sporadic_Consumer_Task.Op2'access;
```

Наконец, с назначением выше, мы можем наложить, что всякий раз, когда `Op1` вызывается на `s1_Instance`, вызов направляется на операцию `Op1` на предоставленном интерфейсе пользовательской спорадической задачи, а оттуда следует правильная цепочка делегирования (перенаправление для OBCS и повторного подтверждения запроса в очереди OBCS).

```
s0_Instance.Set_x(Consumer.Consumer_FC_Ref(s1_Instance'access));
```

Наконец, мы устанавливаем связь между Продюсером и Потребителем с вышеуказанным призывом. Фактически, это гарантирует, что вызов `Op1` внутри OPCS Продюсера (`Producer_FC`) является вызовом предоставленного интерфейса потребительской спорадической задачи.

Вывод

В этом мини серии Ada Gems мы описали набор архетипов кода, совместимых с Ravenscar, для реализации повторяющихся шаблонов в системах реального времени. Мы представили две основные модели для реализации циклических и спорадических задач, прокомментировали их недостатки и показали, как улучшить их, чтобы реализовать спорадические операции с параметрами и пример комплексной политики очередей. Наконец, мы показали, как выполнять межзадачную связь.

Архетипы кода, которые мы описали, использовались для генерации кода на трассе HRT-UML / RCM проекта ASSERT, финансируемого ЕС.

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

Сведения об авторе отсутствуют

Last Updated: 11/24/2017
Posted on: 4/11/2011

Обсуждение…