Русскоязычный перевод документации Eiffel по технологии SCOOP

Аннотация. В настоящее время многоядерные процессоры получили широкое распространение в ПК. Однако сдерживающим фактором для увеличения производительности является наличие ПО, оптимизированного под параллельные вычисления. Основной проблемой при программировании параллельных систем является наличие разделяемых общих ресурсов. Существующие решения для параллельного программирования на языках C++, Java, C# и др. вводят большое количество ключевых слов, опираются на устаревшие низкоуровневые концепции, не сочетающиеся с объектно-ориентированным подходом. Все это требует существенного изменения мышления программиста при переходе к параллельному программированию, а также большого количества изменений при переписывании существующих последовательных программных систем. В рамках языка Eiffel разработана принципиально новая модель параллельного программирования SCOOP, которая добавляет в язык всего одно ключевое слово. Эта модель, опираясь на высокоуровневые понятия процессора, выделенного типа и выделенного вызова, хорошо сочетается с остальными элементами языка, что позволяет экономиться время при проектировании, отладке и тестировании параллельных систем.

Параллелизм в Eiffel с использованием SCOOP

Concurrent Eiffel with SCOOP

Перевод: Семченков Сергей
Редактор перевода: Когтенков Александр

Содержание

1. Обзор

SCOOP (Simple Concurrent Object-Oriented Programming) – это простое параллельное объектно-ориентированное программирование. SCOOP позволяет разработчикам создавать объектно-ориентированные системы, поддерживающие параллельную работу нескольких активных вычислительных устройств. Стоит отметить, что программирование в SCOOP производится на высоком уровне абстракции, скрывающем детали реализации работы с ними. Основная идея состоит в упрощении разработки параллельного ПО. Базовые принципы SCOOP были сформулированы в 1993 году. С тех пор исследования и разработки уточнили эту модель и привели ее в то состояние, которое реализовано в EiffelStudio в настоящее время.

Примечание. Если Вы начинаете работать со SCOOP, прочтите раздел документации Практические вопросы SCOOP, а также скомпилируйте и поработайте с соответствующими примерами.

2. Параллелизм

Параллелизм в вычислении – это ситуация, в которой компьютерная система выполняет несколько параллельных вычислений для достижения конечной цели. Параллельное выполнение может быть организовано с помощью разнообразных вычислительных устройств: отдельные сетевые компьютерные системы, различные процессоры на одном ЦПУ, различные ядра одного чипа, различные потоки внутри процесса, различные процессы одного ЦПУ и т.д.

Параллельные системы не вызывали бы так много проблем, если бы части системы на различных процессорах, процессах, или потоках были бы полностью независимыми, то есть не разделяли бы общих ресурсов. Однако это редкий случай. В параллельных системах одновременно исполняемые программные элементы могут разделять и разделяют ресурсы, взаимодействуя между собой. При этом возникают различные дефекты в форме состояний гонки, нарушений атомарности, взаимной блокировки (deadlock). Эти проблемы сводятся к двум правилам относительно доступа к разделяемым ресурсам.

1. **Избегайте взаимных блокировок.** Убедитесь в том, что никакие два потока не будут ждать бесконечно вследствие того, что каждый нуждается в ресурсе, контролируемом другим потоком.
2. **Обеспечьте равнодоступность.** Убедитесь в том, что каждый поток управления получит возможность исполнения.

Управление параллелизмом – богатая тема для исследований в вычислительной технике. Было разработано множество схем для управления параллельным вычислением. SCOOP – это модель такого рода, причем она серьезно отличается от других исследований.

Во-первых, целью SCOOP является перевод понятия параллелизма на более высокий уровень по сравнению с инструментами и средствами, доступными в настоящее время. Это означает, что если Вам необходимо написать систему с несколькими потоками обработки, то это можно сделать и без SCOOP, используя средства, уже используемые в многопоточном программировании, такие как семафоры и мьютексы. Или Вы можете написать её, используя только механизмы SCOOP. Система, предназначенная для выполнения на нескольких процессорах или нескольких ядрах процессора, может быть написана с использованием тех же самых механизмов SCOOP, которые используются в многопоточной системе.

Во-вторых, модель SCOOP реализована на Eiffel, поэтому зависит от механизма проектирования по контракту с несколько измененной семантикой контрактов и с добавлением ключевого слова **separate** в язык. Семантика предусловий параллельной версии отличается от последовательной версии. Необходимо понять и другие концепции и правила, однако параллельные программы на Eiffel с использованием SCOOP похожи на последовательные программы на этом языке.

В-третьих, SCOOP использует распространенный механизм передачи аргумента для определения необходимости предоставления монопольного доступа.

Мы изучим более подробно основные принципы работы и механизмы совместного использования этих возможностей.

3. Объектно-ориентированные вычисления

Известная в Eiffel конструкция объектно-ориентированных вычислений

```
x.f (a)
```

остается допустимой и в SCOOP. Однако её смысл в случае SCOOP немного отличается. В последовательном Eiffel её можно интерпретировать как вызов компонента f применительно к объекту (цели вызова), прикрепленному к сущности x, с возможным аргументом a. С другой стороны, x.f (a) можно трактовать как применение компонента к объекту, ассоциированному с x. Все это верно для последовательного Eiffel, однако для SCOOP необходимо различать вызов компонента и применение компонента. Различия станут понятны после обсуждения понятий процессоров и выделенных вызовов.

4. Процессоры

В контексте SCOOP процессор – абстрактное понятие.

Определение: Процессор. Процессор – автономный поток управления, способный последовательно выполнять инструкции на одном или более объектах.

Примечание. Дополнительно о процессорах:

1. Процессор – абстрактное понятие, не подразумевающее конкретную реализацию параллельности.
2. В любой конкретный момент времени каждый объект времени выполнения обрабатывается одним процессором. Каждый процессор может обрабатывать любое количество объектов.

В последовательном Eiffel мы понимаем, что существует какой-то процессор, который исполняет нашу систему, но не придаем этому большого значения. Мы рассматриваем его как аппаратное обеспечение, на котором может запускаться ПО.

Термин процессор (или по-другому обработчик) является важным в SCOOP и рассматривается несколько по-другому, чем в обычном Eiffel, то есть не как аппаратный процессор. В параллельной системе может быть любое количество процессоров. Здесь термин используется в более абстрактном значении. В SCOOP мы рассматриваем процессор как любой автномный поток управления, способный применять компоненты к объектам. Если бы Вы писали ПО для аппаратной реализации с несколькими процессорами, то реальные процессоры соответствовали бы процессорам SCOOP. Однако если бы Вы писали программную систему с использованием нескольких потоков

обработки, то эти потоки соответствовали бы процессорам SCOOP.

Примечание. В первоначальной реализации SCOOP в EiffelStudio версии 6.8 поддержка нескольких потоков обработки является единственным механизмом. Каждый процессор SCOOP реализован как отдельный поток выполнения в едином процессе.

Наличие нескольких процессоров в SCOOP проявляется тогда, когда вызовы компонентов применяются к объекту на процессоре, отличном от того, с которого был произведен вызов. Безусловно, это важное различие между вызовом компонента и применением компонента. Можно рассматривать вызов компонента как регистрацию или постановку в очередь запросов на применение компонентов.

5. Выделенные типы и выделенные вызовы

В SCOOP вводятся понятия выделенности.

Определение: Выделенный тип. Выделенный тип – это тип, объявленный с использованием ключевого слова **separate**.

Определение: Выделенный вызов. Выделенный вызов – это вызов, цель которого принадлежит выделенному типу.

Определяющим фактором для использования нескольких процессоров является использование выделенных типов и выделенных вызовов. В работающей системе каждый объект обрабатывается процессором, однако в случае отсутствия выделенных типов и выделенных вызовов в системе будет использован только один процессор во время выполнения, то есть все вызовы будут невыделенными, а соответственно нет параллельного выполнения, основанного на SCOOP.

Пусть сущность использует ключевое слово **separate** в своем объявлении:

```
my_x: separate X
```

Ключевое слово **separate** служит признаком того, что применение компонента к объекту, прикрепленного к ссылке my_x, может производиться на другом процессоре по отношению к тому, с какого был произведен вызов. Такие вызовы, как my_x.f, будут рассматриваться как выделенные вызовы, а my_x принадлежит выделенному типу **separate** X.

Вызов компонента применительно к my_x, например

```
my_x.f
```

будет рассматриваться в общем случае как выделенный вызов потому, что это вызов компонента применительно к объекту выделенного типа, и следовательно может быть применен на другом процессоре. В дальнейшем Вы узнаете, что выделенные вызовы являются допустимыми только в определенных контекстах.

6. Доступ к разделяемым ресурсам

Как было сказано ранее, основной проблемой параллельных систем является правильное использование ресурсов, которые могут разделяться между одновременно работающими процессорами.

Обычные решения проблемы включают использование "критических секций" кода. Это секции кода, в которых осуществляется доступ к разделяемым ресурсам. Только одному процессору разрешено выполнение критической секции в каждый момент времени. Таким образом, если один процессор захочет выполнить критическую секцию, которую уже исполняет другой процессор, первый процессор должен будет ждать выполнения. Схемы синхронизации процессов гарантируют "взаимоисключающую блокировку" доступа к критической секции.

Вместо того, чтобы использовать критические секции, SCOOP полагается на механизм передачи аргумента для гарантирования контролируемого доступа. Как результат, на выделенные вызовы накладывается ограничение.

Правило: Выделенный аргумент. Выделенный вызов a_x.f (a) является допустимым только если a_x является аргументом вызывающей подпрограммы.

Для того, чтобы выделенный вызов был допустимым в соответствии с этим правилом, целью вызова должен быть формальный аргумент подпрограммы, в которой происходит вызов. Следующий код содержит недопустимый и допустимый вызовы.

```
my_separate_attribute: separate SOME_TYPE
        ...
calling_routine
        -- Одна подпрограмма
    do
        my_separate_attribute.some_feature -- Недопустимый вызов:
        -- вызов компонента на выделенном атрибуте
        enclosing_routine (my_separate_attribute) -- Передача выделенного
        -- атрибута в качестве аргумента
    end

enclosing_routine (a_arg: separate SOME_TYPE)
        -- Другая подпрограмма
    do
        a_arg.some_feature -- Допустимый вызов: вызов компонента на
        -- выделенном аргументе
    end
```

В приведенном коде my_separate_attribute является атрибутом класса и принадлежит выделенному типу. В первой строчке подпрограммы calling_routine используется непосредственный вызов подпрограммы для применения some_feature к my_separate_attribute. Этот вызов является недопустимым. Во второй строчке происходит вызов компонента enclosing_routine и передача my_separate_attribute в качестве аргумента. Компонент enclosing_routine принимает аргумент типа separate SOME_TYPE. Внутри компонента enclosing_routine вызов some_feature применительно к a_arg является допустимым.

Дополнительная информация: компонент launch_producer в примере Producer-consumer введён, чтобы не нарушать правило выделенного аргумента.

В компоненте calling_routine, приведенном ранее, вызов к enclosing_routine имеет выделенный аргумент.

```
enclosing_routine (my_separate_attribute) -- Передача выделенного
-- атрибута в качестве аргумента
```

Так как аргумент my_separate_argument принадлежит выделенному типу, то он обрабатывается на другом процессоре относительно того, с какого был произведен вызов enclosing_routine. В результате выполнение enclosing_routine будет отложено до момента времени, когда процессор, обрабатывающий my_separate_argument, будет доступен для монопольного доступа. Этот тип задержки описывается правилом ожидания.

Правило: Ожидание. Вызов подпрограммы с выделенными аргументами будет выполнен, когда все соответствующие процессоры будут доступны для монопольного доступа и обработки в течение всего времени выполнения подпрограммы.

Дополнительная информация.

1. Изучите компонент {PHILOSOPHER}.eat в примере Dining philosophers. Этот компонент содержит два выделенных аргумента и будет ожидать выполнения до тех пока, когда процессоры, соответствующие обоим аргументам, будут доступны.
2. Как реализовано правило ожидания

6.1 Контролируемые выражения

Допустимые цели для выделенных вызовов, таких как a_arg в enclosing_routine, называются контролируемыми.

Определение: Контролируемое/неконтролируемое выражение. Выражение является **контролируемым**, если оно прикреплено, и выполняется одно из двух условий:

1. Оно не принадлежит выделенному типу.
2. Оно принадлежит выделенному типу и обрабатывается тем же процессором, что и выделенные аргументы в содержащей их подпрограмме.

Если ни одно из этих условий не выполняется, то выражение является **неконтролируемым**.

Определение контролируемого выражения означает, что такое выражение контролируемо по отношению к обрабатываемому процессором контексту, в котором используется выражение, то есть все объекты, необходимые выражению, находятся под контролем (доступны для монопольного доступа) текущего процессора и не могут быть изменены другими процессорами.

7. Синхронные и асинхронные вызовы подпрограмм

Если мы думаем о исполнении программ в последовательном Eiffel, то как правило не различаем вызов компонента и применение компонента. Например, при выполнении следующей последовательности вызовов компонент:

```
x.f
y.g
```

следует ожидать завершения применения x.f перед началом выполнения y.g.

В параллельном Eiffel с использованием технологии SCOOP эти понятия различны. Так происходит потому, что конкретный вызов компонента, например, x.f, может быть сделан на одном процессоре, а последовательное применение компонента (применение f к x) выполняться на другом процессоре.

Определение: Синхронный вызов компонента. Синхронный вызов компонента – это вызов компонента, при котором выполнение команд вызывающего клиента не возобновляется до завершения применения компонента.

Определение: Асинхронный вызов компонента. Асинхронный вызов компонента – это такой вызов, при котором происходит "регистрация" вызова клиента на применение соответствующего компонента, исполняемого процессором поставщика.

После асинхронного вызова компонента выполнение команд на клиенте продолжается незамедлительно, возможно параллельно применению компонента на каком-либо ином процессоре. Мы пересмотрим эту ситуацию после изучения того, что делает вызов синхронным или асинхронным.

7.1 Что делает вызов синхронным или асинхронным

Каждый вызов компонента является синхронным или асинхронным. Существует правило, которое позволяет определить для каждого конкретного вызова компонента, является ли он синхронным или асинхронным.

Вызов компонента является **синхронным** в следующих случаях.

S1. Он не является выделенным вызовом.

S2. Он является выделенным вызовом:

S2.1. запросом;

S2.2. командой, которая имеет как минимум один фактический аргумент ссылочного типа и он или

S2.2.1 является выделенным аргументом содержащей его подпрограммы или

S2.2.2 Current.

Вызов компонента является **асинхронным** в следующих случаях:

A1. Он является выделенным вызовом к команде, не имеющей аргументов, или аргументы которой не соответствуют пункту S2.2.

Рассмотрим более подробно случаи, соответствующие синхронному вызову.

Случай S1 является типичным в последовательном Eiffel, где все вызовы не являются выделенными, а значит синхронные. В параллельном Eiffel с использованием SCOOP будет встречаться множество невыделенных вызовов, являющихся синхронными.

В случае S2.1 говорится о том, что если выделенный вызов является запросом, то он должен быть синхронным. Несмотря на то, что применение компонента будет происходить скорее всего на другом процессоре, инструкции, следующие за запросом, будут зависеть от результата этого запроса, то есть необходимо ждать завершения применения компонента. Эта ситуация называется ожиданием по необходимости.

Случай S2.2 описывает ситуацию, при которой вызов имеет хотя бы один фактический аргумент, который является ссылкой на текущий объект (Current) или выделенным формальным аргументом подпрограммы, содержащей вызов. В этом случае клиент вызывает подпрограмму и передает аргументы, являющиеся контролируемыми в контексте вызывающей подпрограммы. Фактическими аргументами являются объекты, к которым процессор клиента имеет монопольный доступ в содержащей его подпрограмме. Для того, чтобы процессор поставщика смог применить компонент (предположительно обращаясь к объектам аргументов), клиент должен передать монопольный доступ к этим объектам поставщику. Это производится с использованием механизма “передача доступа”. Поскольку клиент передал монопольный доступ процессору поставщика, он не может продолжить выполнение инструкций до тех пор, пока не завершится применение вызванного компонента процессором поставщика, а процессор поставщика не передаст монопольный доступ обратно клиенту. Следовательно, этот тип вызовов должен быть синхронным.

Теперь рассмотрим единственный случай A1, соответствующий асинхронному вызову.

Выделенные вызовы к командам являются асинхронными (за исключением случая S2.2). Это означает, что когда клиент выполняет асинхронный вызов компонента, он записывает (регистрирует) необходимость применения соответствующего компонента. Вместо ожидания завершения применения компонента клиентская подпрограмма продолжает выполнять инструкции, следующие за асинхронным вызовом.

В этом случае происходит параллельное выполнение. Процессор клиента может продолжить выполнение в то время, когда процессор, контролирующий цель асинхронного вызова, применяет компонент.

8. Проектирование по контракту и SCOOP

Основой метода Eiffel является проектирование по контракту. Предусловия, постусловия и инварианты классов используются в Eiffel для расширения программных интерфейсов до спецификации ПО. В параллельном Eiffel с использованием технологии SCOOP они остаются такими же, как и в последовательном Eiffel. Однако в связи с параллельным выполнением при использовании SCOOP семантика времени выполнения элементов, реализующих проектирование по контракту, будет другой.

8.1 Предусловия

Роль предусловий в SCOOP отличается от последовательного Eiffel. В последовательном Eiffel мы рассматриваем предусловие подпрограммы как множество обязанностей потенциально возможных вызывающих подпрограмм. То есть, это множество условий, которые должны быть истинны, чтобы можно ожидать корректного выполнения подпрограммы. Мы рассматриваем предусловия в последовательном Eiffel как **условия корректности**. Типичным примером может быть подпрограмма вычисления квадратного корня, которая возвращает квадратный корень переданного значения аргумента. Предусловие для этой подпрограммы будет заключаться в том, что аргумент должен быть неотрицательный. Вызывающая подпрограмма ответственна за выполнение этого свойства аргумента при каждом вызове компонента.

В параллельном Eiffel допускаются те же самые условия корректности, однако в этих случаях роль клиента немного изменяется. В случае предусловия, зависящего от неконтролируемого объекта, даже если клиент проверяет условие перед вызовом, нет гарантии того, что действие, выполненное другим параллельно работающим процессором, не нарушит предусловие в промежутке между его проверкой и применением компонента. Таким образом, клиент не может гарантировать выполнимость предусловия. Такой тип предусловий называется **неконтролируемым предусловием**.

Определение: Контролируемое/неконтролируемое утверждение (предусловие или постусловие). Выражение предусловия или постусловия для компонента f является **контролируемым**, если после замены фактических аргументов формальными, выражение включает только вызовы к сущностям, которые являются контролируемыми в контексте подпрограммы, вызывающей f.
В противном случае выражение утверждения (предусловия или постусловия) является **неконтролируемым**.

Определение контролируемости или неконтролируемости конкретного выражения зависит от контекста вызываемой подпрограммы. Это означает, что конкретное выражение для компонента f может быть как контролируемым, так и неконтролируемым в зависимости от вызывающей подпрограммы.

Неконтролируемые выражения предусловий требуют адаптации семантики предусловий.

Примечание. Нарушение **контролируемого предусловия** вызывает исключение в вызывающей подпрограмме как только нарушение было обнаружено. Нарушение **неконтролируемого предусловия** не вызывает исключение в вызывающей подпрограмме. Вместо этого, применение компонента будет ожидать момента времени, когда предусловие будет выполнено.

Таким образом, ответственность клиента ограничена только контролируемыми предусловиями. Неконтролируемые предусловия являются условиями ожидания.

Дополнительная информация: компонент {PRODUCER}.store в примере Producer-consumer. Когда происходит вызов {PRODUCER}.produce, компонент {PRODUCER}.store задаёт условие ожидания.

8.2 Постусловия

Как и в случае предусловий, эффект параллельного выполнения изменяет взгляд и на постусловия.

Если подпрограмма выполнена корректно, то постусловие будет выполнено в момент ее завершения. Это утверждение верно вне зависимости от того, есть ли параллельное выполнение или нет. Однако когда постусловие включает выделенные вызовы или сущности, то клиенты должны осторожно относиться к зависимости от состояний, гарантируемых постусловиями.

8.3 Инварианты класса

Правило выделенного аргумента говорит о том, что выделенные вызовы допустимы только для целей, являющихся формальными аргументами включающих их подпрограмм. Поскольку инварианты класса не являются подпрограммами, а следовательно, не имеют аргументов, выделенные вызовы недопустимы в инвариантах класса.

Примечание. Технически можно написать в инварианте класса инлайн агент, передающий аргументы выделенных типов, а затем выполнить выделенный вызов внутри инлайн агента. Однако в общем случае можно считать, что инварианты класса не содержат выделенных вызовов.

Семантика инвариантов класса точно такая же, как и в последовательном Eiffel, так как инварианты не включают выделенные вызовы. Переводя в термины SCOOP, можно сказать, что инварианты класса гарантируют допустимость выполнения каждого конкретного объекта на процессоре, контролирующем этот объект.