Аарон Сконнард

Введение в XPath 2.0

Более двух лет назад в одном из первых выпусков этой рубрики я писал об XPath версии 1.0 (см. http://msdn.microsoft.com/msdnmag/issues/0700/xml). В исходной спецификации утверждается: «XPath — это язык для адресации к частям XML документа». Выражения XPath дают разработчикам возможность с легкостью указывать узлы XML-документа, которые затем будут обрабатываться. Это позволяет вместо сложных алгоритмов обхода дерева XML-документа применять простые декларативные выражения. Так, следующее выражение идентифицирует все дочерние элементы элемента LineItem, атрибут Sku которого имеет значение 123, — потомки корневого элемента Invoice:

```
/Invoice//LineItem[@Sku='123']/*
```

Выражения XPath дают разработчикам возможность с легкостью указывать узлы XML-документа. Это позволяет вместо сложных алгоритмов обхода дерева документа использовать простые декларативные выражения.

Реализовать ту же логику с помощью традиционного XML API было бы крайне утомительно и сложно. Поэтому XPath обычно поддерживается как один из сервисных уровней в современных реализациях вроде DOM, XPathNavigator и т. д.:

```
// Код на C#, использующий DOM
XmlNodeList nodes =
   doc.SelectNodes("/Invoice//LineItem[@Sku='123']/*");
for (int i=0; i<nodes.Count; i++) {
    ... // обработка выбранных узлов
}
```

Кроме того, XPath интенсивно используется в XSLT 1.0 для указания входящих документов:

```
<xsl:apply-templates
   select="/Invoice//LineItem[@Sku='123']/*"/>
```

В общем, XPath предоставил XML-разработчикам огромные возможности и быстро стал неотъемлемой частью их инструментария. Для начала я познакомлю вас с XPath 1.0 (см. http://www.w3.org/TR/xpath).

Ограничения XPath 1.0

Хотя спецификация XPath 1.0 упростила решение многих типовых задач программирования, разработчики желают большего. Некоторые разделы в ней накладывали ограничения или создавали путаницу, и поэтому спецификация требовала «косметического ремонта». Разработчики подталкивали консорциум W3C к добавлению новых возможностей, связанных в основном с поддержкой других развивающихся W3C-спецификаций XML (таких как XML Schema, XML Query 1.0 и XSLT 2.0).

С момента выпуска XPath 1.0 XML Schema стала рекомендуемой спецификацией W3C и быстро вышла на позиции «официальной» системы типов для нескольких других разрабатываемых стандартов (вроде XQuery), связанных с Web-сервисами. Так как XML Schema становится неотъемлемой частью технологий XML, консорциум W3C стремится реализовать стандарт XPath со строгим контролем типов. Более того, при введении новейших разработок — XQuery 1.0 и XSLT 2.0 — обнаружилось много общих областей, в которых оба языка могли бы использовать одни и те же модель данных и синтаксис выражений. Таким «наименьшим общим знаменателем» стал XPath 2.0.

XPath 2.0

Все это явилось причиной разработки стандарта XPath 2.0 (http://www.w3.org/TR/XPath20), предназначенного для устранения проблем в стандарте XPath 1.0 и выполнения списка требований (подытоженных в документе XPath 2.0 Requirements на http://www.w3.org/TR/xpath20req):

- поддержка обратной совместимости;
- бóльшая простота в использовании;
- большее удобство в работе со строками и в сравнениях;
- поддержка семейства XML-стандартов (вроде XSLT 2.0 и XQuery 1.0);
- поддержка XML Schema (простых и составных типов).

Требование обратной совместимости не является абсолютным, так как оно менее важно, чем другие и все же W3C проделал хорошую работу по обеспечению обратной совместимости.

Стремление к простоте использования и упрощению работы в типичных ситуациях (например, при операциях со строками) вполне понятно. Однако последние два требования привели к более значимым изменениям. Я расскажу об основных аспектах нового стандарта, которые помогут вам подготовиться к работе с ним.

Если вы хотите поэкспериментировать с некоторыми примерами выражений, которые встретите в этой статье, скачайте XSLT-процессор SAXON 7.2 Майкла Кея (Michael Kay), содержащий довольно полную реализацию стандарта (http://saxon.sourceforge.net/saxon7.2/) за исключением того, что он не полностью поддерживает XML Schema. На момент написания статьи ни один из XML-процессоров (или API) Microsoft не поддерживает XPath 2.0, но ожидается, что такая поддержка появится вместе с реализацией XQuery 1.0 и XSLT 2.0. Точные даты выхода пока неизвестны, но имеет смысл ориенти-

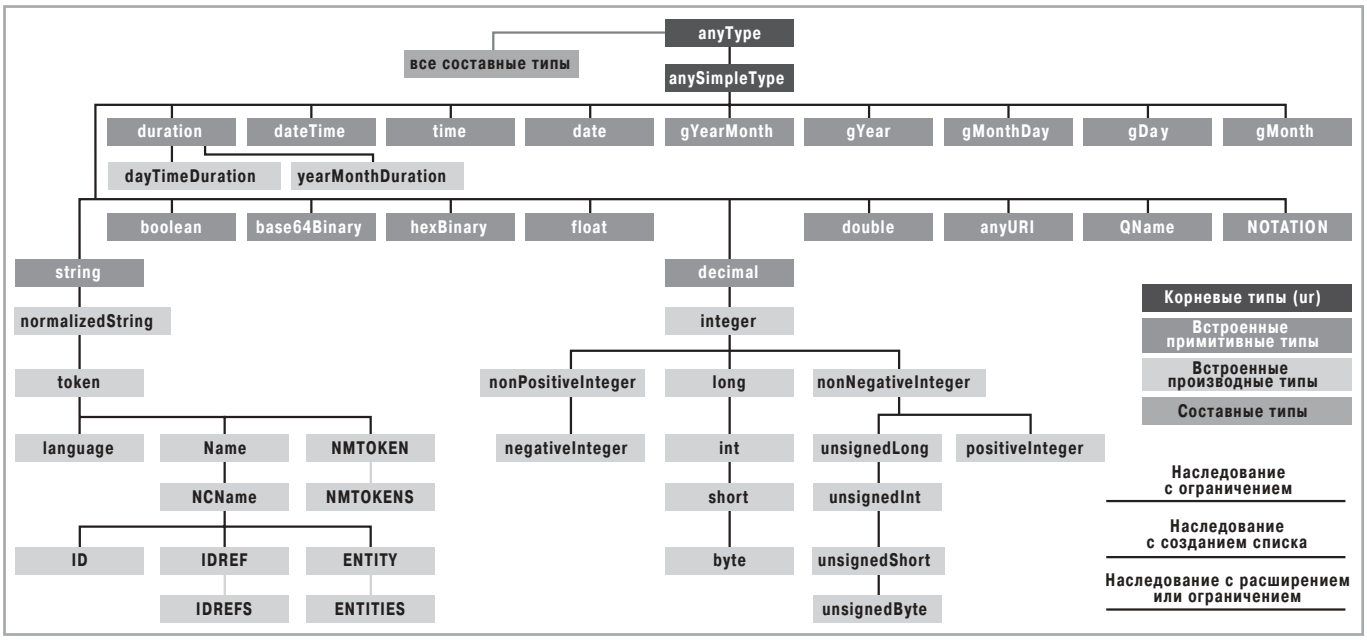


Рис. 1. Иерархия типов данных в XML Schema

роваться на выпуск спецификаций (которые, вероятно, появятся в будущем году).

Одна модель данных, управляющая всем

Самое большое изменение в XPath 2.0 спрятано глубоко внутри модели данных. Модель данных XPath 2.0 описывается в спецификации XQuery 1.0 and XPath 2.0 Data Model, используемой стандартами XPath 2.0, XSLT 2.0 и XQuery 1.0. Модель данных XPath 2.0 основана на Infoset с расширениями, необходимыми для поддержки XML Schema. В модели данных определяются семь типов узлов, составляющих структуру XML-документа: документ (корневой узел), элемент, текст, атрибут, пространство имен, инструкция обработки и узлы комментариев.

Типы узлов аналогичны типам узлов в модели данных XPath 1.0, а типы элементов и атрибутов расширены, так чтобы после проверки на допустимость в них добавлялась информация о типах XML Schema. Получаемая в результате «типизированная» модель данных называется PSVI (Post Schema-Validation Infoset).

Помимо основных типов узлов, в XPath 1.0 определяются четыре атомарных типа, используемых при обработке значений, которые содержатся в элементах или атрибутах: набор узлов, строковый, числовой и логический. Эта простая система типов позволяет интерпретировать текстовые значения как значения определенных типов. Рассмотрим варианты элемента price:

```
<price>10</price>
<price>10.0</price>
<price>10.00</price>
```

Если при сравнении элемента с определенным значением работать только с текстом, придется создавать громоздкую конструкцию, в которой проверяются всевозможные лексические форматы:

```
price = '10' || price = '10.0' || price = '10.00'
```

Если же взять на вооружение числовой тип, то можно просто задействовать следующее выражение, в котором текст автоматически преобразуется в числовое значение:

```
number(price) = 10
```

Это позволяет избежать обработки разных лексических форматов, которые могут встретиться в документе.

В XPath 2.0 система атомарных типов основывается на тех же принципах, но в нее включены все типы данных, определенные в спецификации XML Schema Datatypes (http://www.w3.org/TR/xmlschema-2/). На **рис. 1** показана иерархия типов данных XML Schema, в которой применяется наследование с ограничением (derivation by restriction). Это 19 простейших типов данных и многочисленные производные типы. Кроме того, в XPath 2.0 вводятся два новых подтипа длительности (duration): yearMonthDuration и dayTimeDuration, которые упрощают работу со значениями, задающими длительность. Все эти типы можно использовать в XPath 2.0.

PSVI не только поддерживает строго типизированные значения, но и позволяет в период выполнения получать доступ к информации о типе любого элемента или атрибута по аналогии с тем, как сейчас можно обращаться к именам элементов.

Система типов XPath 2.0 по-прежнему ограждает разработчика от большинства лексических деталей, позволяя работать главным образом с пространствами значений, но теперь появилось много новых пространств значений (value-spaces). Вы можете обращаться к типизированным значениям узлов посредством функции data и конструировать типизированные значения (или приводить тип значений) через различные функции-конструкторы, определенные в спецификации XQuery 1.0 and XPath 2.0 Functions and Operators. Так, в следующем выражении проверяется наличие элементов birthdate (типа xs:date, где xs связывается с пространством имен схемы) с датой, предшествующей 1 января 1972 г.:

```
data(birthdate) < xs:date('1972-01-01')
```

PSVI не только поддерживает строго типизированные значения, но и позволяет в период выполнения получать доступ к информации о типе любого элемента или атрибута по аналогии с тем, как сейчас можно обращаться к именам элементов. Элементы относятся к простым или составным типам, тогда как атрибуты — только к простым. Вот несколько новых операторов, используемых для проверки и приведения типов: instance of, cast as и treat as.

Оператор instance of позволяет проверить, относится ли операнд к определенному типу:

```
LineItem/node()[. instance of element of type xsd:double]

LineItem/*[. instance of xsd:double]
```

Это выражение идентифицирует все дочерние элементы LineItem, относящиеся к типу xsd:double. Оператор cast as позволяет явно приводить значения одного типа к другому, но не работает с пользовательскими типами. С другой стороны, оператор treat as поддерживает пользовательские типы и проверяет, является ли данное выражение экземпляром указанного типа данных. Если да, возвращается значение выражения, нет — генерируется ошибка. Treat as обычно применяется, чтобы гарантировать, что динамический тип значения является определенным подтипом статического типа значения. В спецификации XQuery 1.0 and XPath 2.0 Functions and Operators определяются многочисленные правила преобразования, используемые при приведении типов. По-видимому, это самая большая и сложная часть спецификации.

В XPath 2.0 предпринята попытка устранить путаницу за счет дифференциации выражений сравнения и предоставления возможности явного выбора между применением квантора существования и использованием квантора всеобщности.

Другое значительное изменение в модели данных связано с наборами узлов, точнее с их отсутствием. В XPath 1.0 одни выражения возвращали наборы узлов, другие — атомарные значения. В XPath 2.0 все значения являются последовательностями (sequences).

Последовательности

Название «последовательность» говорит само за себя — это и есть упорядоченная последовательность, содержащая ноль или более членов (items). Член последовательности может быть атомарным значением или относиться к одному из семи типов узлов. Узел, помещаемый в последовательность, сохраняет свою идентификацию и может входить в последовательность несколько раз.

Последовательности во многом отличаются от наборов узлов XPath 1.0 — не упорядоченных и содержащих только узлы без дубликатов. Хотя наборы узлов не упорядочены, они обрабатываются XSLT 1.0 в порядке документа (document order). Для поддержки обратной совместимости в XPath 2.0 выражения, задающие пути (path expressions), всегда генерируют последовательности в порядке документа и не включают в них дубликаты (для других типов выражений это не гарантируется). А в XSLT 2.0 последовательности просто обрабатываются в порядке последовательности (order of the sequence). Как вы вскоре увидите, последовательности обеспечивают гораздо бóльшую гибкость при обработке.

В XPath 2.0 каждое значение является последовательностью. Последовательности записываются в скобках, а их члены разделяются запятыми, например:

```
("Nathan", 1.32e0, true(), xs:date('2001-05-24'))
```

Эта последовательность состоит из четырех значений типов xs:string, xs:double, xs:boolean и xs:date. Следующая последовательность состоит из списка элементов Sku, далее идет список элементов Price, затем — список элементов Description:

```
(//Sku, //Price, //Description)
```

Пустая последовательность записывается как (). Даже атомарные значения рассматриваются как последовательности с длиной, равной 1; такие последовательности называются одинарными (singleton sequences) и могут быть записаны как («Nathan») или «Nathan». Эти две записи обрабатываются одинаково.

Другая важная характеристика последовательностей — это то, что они плоские, т. е. в них не может быть других последовательностей. Например, эти три последовательности идентичны:

```
(1, 2, 3, 4)
((1, 2), (3, 4))
(((1), (2, 3), (4)))
```

Так как последовательности являются основой всего языка, для работы с ними предусмотрены самые разнообразные операторы и функции.

Операторы множеств

Я уже познакомил вас с одним оператором для работы с последовательностями — с запятой. Запятая является оператором конкатенации, просто соединяющим две последовательности с сохранением их порядка. Для последовательностей есть и другие операторы: объединение (|), intersect и except.

Оператор | возвращает объединение двух последовательностей, из которого исключены дубликаты (как в XPath 1.0). Вот пример объединения (предполагается, что $node1, $node2, $node3 и $node4 ссылаются на разные узлы):

```
($node1, $node2, $node3) | ($node2, $node3, $node4) ->
($node1, $node2, $node3, $node4)
```

Оператор intersect возвращает пересечение двух последовательностей, из которого удалены дубликаты:

```
($node1, $node2, $node3) intersect ($node2, $node3, $node4)->
($node2, $node3)
```

И, наконец, except возвращает каждый узел первой последовательности, отсутствующий во второй последовательности, также с удалением дубликатов:

```
($node1, $node2, $node3) except ($node2, $node3, $node4) ->
($node1)
```

Помимо операторов множеств, имеется несколько других функций, работающих с последовательностями (см. также http://www.w3.org/TR/xquery-operators).

Обработка последовательностей

Как и следовало ожидать, в XPath 2.0 есть набор функций, реализующих основные операции со списками: insert, remove, item-at, index-of и subsequence. Первые две позволяют добавлять члены в последовательности и удалять их оттуда:

```
insert((1, 3, 4), 2, 2) -> (1, 2, 3, 4)
remove((1, 2, 3), 2)    -> (1, 3)
```

Функция item-at возвращает член с заданным индексом (что равносильно простому предикату позиции [index]), тогда как функция index-of возвращает индексы членов, имеющих заданные значения:

```
index-of((10, 20, 30), 20) -> 2
```

Subsequence возвращает непрерывное подмножество (подпоследовательность) по заданным начальному и конечному индексам.

Проверить длину последовательности можно несколькими способами, например посредством функций exists, empty и count. Exists возвращает true, если последовательность непустая, иначе возвращается false. Empty, наоборот, возвращает true, если последовательность пустая, в противном случае — false. Count, как и в XPath 1.0, возвращает количество членов последовательности.

```
empty(())         -> true
exists((1, 2, 3)) -> true
count((1, 2, 3)   -> 3
```

Кроме того, функции sum, avg, min и max позволяют выполнять над последовательностями математические вычисления:

```
sum(1, 2, 3) -> 6
avg(1, 2, 3) -> 2
min(1, 2, 3) -> 1
max(1, 2, 3) -> 3
```

Моя любимая функция — distinct-values — возвращает новую последовательность, содержащую только члены с уникальными значениями (все дубликаты удаляются, как в SQL-операторе SELECT DISTINCT). Предполагается, что последовательность состоит либо только из узлов, либо только из значений, иначе возникает ошибка. Так, следующее выражение идентифицирует все уникальные элементы Sku, которые содержатся в документе:

```
distinct-values(//Sku)
```

Эта функция удобнее, чем ее эквивалент из XPath 1.0:

```
//Sku[not(preceding::Sku = .)]
```

А функция distinct-nodes удаляет дублирующиеся узлы по идентификационным данным, а не по значениям.

Последовательности и предикаты

В XPath 1.0 можно, используя скобки, применять предикаты к определенным наборам узлов. Рассмотрим выражение:

```
//Price[1]
```

Казалось бы, оно возвращает первый элемент Price в порядке документа. Но на самом деле оно возвращает каждый элемент Price, являющийся первым дочерним элементом своего родителя (так получается из-за приоритета операторов). Это можно исправить, взяв в скобки набор узлов, который вы собираетесь фильтровать:

```
(//Price)[1]
```

Та же модель действует в отношении последовательностей. Предикаты позволяют отфильтровать любые ненужные члены последовательностей. При применении предиката к последовательности возвращается новая последовательность, которая содержит только члены, удовлетворяющие выражению предиката. Так, следующий предикат проверяет позицию члена (что равносильно использованию item-at):

```
(10, 20, 30)[2] -> (20)
```

А вот предикат чуть сложнее, который проверяет число дочерних элементов и значение атрибута id:

```
$seq[count(*) > 1 and @id < 1000]
```

В общем, все, что вы знаете о предикатах XPath 1.0, применимо и при их использовании с последовательностями XPath 2.0.

Приведение типов для последовательностей

Если вы используете последовательность в контексте, в котором нужен другой тип, выполняется попытка неявного приведения типа. Правила приведения типов для строк, чисел и логических значений в основном аналогичны правилам XPath 1.0. Например, если последовательность указывается там, где ожидается строковое значение, то берется значение первого члена последовательности, которое приводится к строковому типу. Строковые значения узлов вычисляются по таким правилам: для текста/атрибутов это просто их значение, а для элементов — результат конкатенации дочерних текстовых узлов. Если последовательность указывается там, где ожидается число, последовательность сначала приводится к строковому типу, затем этот строковый тип приводится к double. Единственным преобразованием, выполняемым немного иначе, является приведение к типу Boolean. Если последовательность пустая, в результате приведения типа получается false. Последовательность, содержащая одно значение типа Boolean, рассматривается как это значение, в остальных случаях непустая последовательность, содержащая минимум один узел, приводится к значению true.

Для выполнения итераций XPath 2.0 поддерживает выражение «for». Этот тип выражения удобен для генерации новых последовательностей из нескольких источников и аналогичен выражениям с явным применением кванторов.

В XPath 2.0, как и в XPath 1.0, есть функции явного приведения типов string, number и Boolean, а также (я уже говорил об этом) несколько дополнительных функций для приведения к типам данных XML Schema (xsd:date, xsd:unsignedInteger и т. д.).

Сравнения

Одна из наиболее неприятных областей XPath 1.0 — сравнение наборов узлов. Рассмотрим простое выражение:

```
Catalog/Prices/Price = 9.95
```

Оно принимает значение true, если существует хотя бы один элемент Price (являющийся подэлементом Catalog/Prices) с числовым значением, равным 9.95. Когда в логических выражениях такого вида применяются последовательности, результат принимает значение true, если сравнение истинно хотя бы для одного из членов последовательности. Это пример использования квантора существования (existential quantification). Интересно, что то же правило распространяется на любые другие операторы сравнения (<, <=, >, >= и т. д.) и даже на оператор «не равно» (!=):

```
Catalog/Prices/Price != 9.95
```

Результатом вычисления является true, если есть хотя бы один элемент Price (являющийся подэлементом Catalog/Prices) с числовым значением, не равным 9.95. Следовательно, вполне вероятно, что оба сравнения дадут true для одного и того же набора узлов. Нет нужды говорить, что все это приводит к большой путанице.

Логический оператор not позволяет проверить все члены последовательности. Это пример использования квантора всеобщности (universal quantification). Так, чтобы убедиться, что все элементы Price имеют значение 9.95, примените логический оператор not к последнему выражению:

```
not(Catalog/Prices/Price != 9.95)
```

А если хотите убедиться, что ни у одного элемента Price нет числового значения, равного 9.95, примените not к первому выражению:

```
not(Catalog/Prices/Price = 9.95)
```

В XPath 2.0 предпринята попытка устранить путаницу за счет дифференциации выражений сравнения и предоставления возможности явного выбора между применением квантора существования и квантора всеобщности. Чтобы обеспечить обратную совместимость для универсальных выражений сравнения на основе стандартных операторов =, !=, <, <=, > и >=, в XPath 2.0 сохранены правила по умолчанию стандарта XPath 1.0 (применение квантора существования, т. е. семантики «хотя бы для одного»). Кроме того, добавлены функции sequence-deep-equal и sequence-node-equal, предназначенные для сравнения последовательностей по значениям и по идентификациям узлов соответственно.

В XPath 2.0 используется отдельный механизм для сравнения простых значений. Атомарные значения членов сравниваются с помощью операторов eq, ne, lt, le, gt и ge (соответствующих =, !=, <, <=, > и >=). Также определены дополнительные операторы сравнения для одиночных узлов. Например, операторы is и isnot проверяют идентичность узлов:

```
Price[1] is Price[1]  -> true
Price[1] isnot Sku[1] -> true
```

Кроме того, можно проверять порядок узлов операторами << и >>. Первый возвращает true, если узел первого операнда находится в порядке документа перед узлом второго операнда, а второй делает прямо противоположное. Так как выражения сравнения значений и узлов обрабатывают одинарные последовательности, выбор между семантиками «для всех» (for all) или «хотя бы для одного» (for any) здесь просто не имеет смысла.

Явное применение кванторов

В XPath 2.0 появился тип выражений, позволяющий создавать выражения с явным применением кванторов. Можно указать, какие кванторы должны использоваться — существования (хотя бы для одного) или всеобщности (для всех), — применив ограничивающее выражение (satisfies) к узлам заданной последовательности:

```
some $item in //LineItem satisfies
   (($item/Price * $item/Quantity) > 100)

every $item in //LineItem satisfies
   (($item/Price * $item/Quantity) > 100)
```

Первое выражение возвращает true, если есть хотя бы один элемент LineItem с суммой (Price * Quantity), большей 100. Второе выражение возвращает true, только если сумма каждого LineItem больше 100.

Так как ограничивающее выражение может быть любым, эта модель обеспечивает гораздо бóльшую гибкость, чем =,

!=, < и др. Более того, как показано в следующем примере, где проверяется декартово произведение последовательностей, ограничений на число задаваемых последовательностей нет:

```
some $x in (1, 2, 3), $y in (2, 3, 4)
   satisfies $x + $y = 4
```

Спецификация XPath 2.0 явно гибче и проще в использовании, чем XPath 1.0.

Другие усовершенствования в языке

В XPath 2.0 стало возможным проверять нескольких узлов за один прием:

```
LineItems/(Sku|Price)/text()
```

В XPath 1.0 с той же целью пришлось бы объединять два выражения:

```
LineItems/Sku/text() | LineItems/Price/text()
```

Кроме того, для выполнения итераций XPath 2.0 поддерживает выражение «for». Этот тип выражения удобен для генерации новых последовательностей из нескольких источников (например, объединений) и аналогичен выражениям с явным применением кванторов. Так, следующее выражение генерирует новую последовательность, содержащую сумму каждого элемента LineItem:

```
for $i in //LineItem return ($i/Price * $i/Quantity)
```

Это крайне полезно, поскольку можно передавать новые последовательности в другие функции для дальнейшей обработки. Например, это выражение вычисляет итоговую сумму по счету:

```
sum(for $i in //LineItem return ($i/Price * $i/Quantity))
```

Такое вообще невозможно в XPath 1.0 и очень утомительно при реализации в XSLT 1.0 (потребуются рекурсия и временное дерево результатов). Но вы можете использовать и более сложные преобразования. Вот как генерируется список продавцов и продаваемых ими товаров:

```
for $sp in distinct-values(//Salesperson)
return ($sp,
        for $item in //LineItem[Salesperson = $sp]
        return $item/Description)
```

Полученная последовательность содержит последовательность элементов Salesperson, за каждым из которых идет последовательность элементов LineItem с данными о предлагаемых товарах.

В XPath 2.0 также поддерживается встроенный условный оператор (if), синтаксически состоящий из традиционных ключевых слов if/then/else. Например, по значению исходной цены можно вычислить цену со скидкой:

```
if ($Price > 100)
   then ($Price * .90)
   else ($Price * .95)
```

Эти новые языковые конструкции применимы везде, где допустимы выражения XPath.

Улучшенная поддержка строк

В XPath 2.0 расширена поддержка обработки строк. Введено несколько новых функций: upper-case, lower-case, string-pad, matches, replace и tokenize. До появления upper-case и lower-case единственным способом преобразования строк в верхний или нижний регистр была неудобная и неэффективная функция translate, а до появления функции string-pad приходилось прибегать к рекурсии:

```
upper-case('Michael') -> MICHAEL
string-pad('-', 7)    -> '-------
```

Последние три функции служат для работы с регулярными выражениями (regular expressions). Функция matches позволяет проверить, соответствует ли данная входная строка данному регулярному выражению. Вот как, например, проверить, соответствует ли элемент SSNumber стандартному формату номера карточки социального обеспечения (в США):

```
matches(SSNumber, '\d{3}-\d{2}-\d{4}')
```

Функция replace позволяет заменять подстроки, соответствующие шаблону, а tokenize — разбивать строки на подстроки по заданному шаблону.

Кому мусор, а кому сокровище

XPath 2.0 занимает центральное место в семействе спецификаций XML. Он выступает в роли официального языка адресации, используемого другими спецификациями вроде XQuery 1.0 и XSLT 2.0. XPath 2.0 удобнее, так как обладает более обширной функциональностью и, насколько возможно, поддерживает обратную совместимость. Кроме того, в него введена поддержка XML Schema, что весьма ценно для разработчиков, применяющих строго типизированные данные.

Однако у стандарта XPath 2.0 есть и оппоненты. Многих XML-разработчиков (особенно тех, кто ушел в дебри XSLT) смущает возросшая сложность XPath 2.0. Их основная претензия в том, что использование XML Schema стало обязательным. Длительные дебаты по XPath 2.0 недавно выплеснулись в довольно жаркие дискуссии.

Разработчики, интенсивно работающие с базами данных или Web-сервисами, несомненно, сочтут, что преимущества нового стандарта стоят своей цены. Но я чувствую глубокую симпатию к тем XSLT-разработчикам, которым нужен новый усовершенствованный XPath без обязательной строгой типизации. Сложно сказать, чем это все обернется. Если у вас есть соображения на этот счет или вы хотите поделиться замечаниями о языке, направляйте их в общедоступный список рассылки W3C XPath/XQuery (http://www.public-qt-comments@w3.org).

Вопросы и комментарии (на английском языке) присылайте по адресу xml@microsoft.com.

Аарон Сконнард (Aaron Sconnard) — преподаватель и исследователь в компании DevelopMentor, где он разрабатывает учебные курсы по XML и Web-сервисам. Соавтор книг «Essential XML Quick Reference» (Addison-Wesley, 2001) и «Essential XML» (Addison-Wesley, 2000). С ним можно связаться по адресу http://staff.develop.com/aarons.

Кристоф Назар

Отладка

Два инструмента для выявления и устранения утечек GDI-ресурсов в Windows XP

Эта статья предполагает знание Win32 и C#

Уровень сложности: ① ② ❸

АННОТАЦИЯ В одной из предыдущих статей автор предложил простой метод обнаружения GDI-объектов (Graphical Device Interface), не освобождаемых надлежащим образом Win32-приложениями на платформах Windows 9*x*. Теперь он предлагает модифицированный вариант этого метода для новых версий Windows, которые требуют иного подхода к проблеме утечек GDI-ресурсов. В статье описаны два инструмента, позволяющие выявлять и устранять утечки GDI-ресурсов в приложениях, выполняемых в операционных системах Windows XP, Windows 2000 и Windows NT.

Статьи на смежную тематику:
http://msdn.microsoft.com/msdnmag/issues/02/08/EscapefromDLLHell/default.aspx и
http://msdn.microsoft.com/msdnmag/issues/02/06/debug/default.aspx.

Базовую информацию, необходимую для понимания этой статьи, см. в книгах Фен Хуаня (Feng Huan) «Windows Graphics Programming: Win32 GDI and DirectDraw» (Prentice Hall PTR, 2000), Джона Роббинса (John Robbins) «Debugging Applications» (Microsoft Press, 2002) и Джеффри Рихтера (Jeffrey Richter) «Microsoft Windows для профессионалов» 4-е изд. (Русская Редакция, М., 2000).

В Windows 95/98/Me описатель (handle) GDI-объекта — это 16-битное значение, которое позволяет любому приложению вызывать GDI-функции. В «MSDN Magazine» за март 2001 г. я показал, как на этих платформах создать GDIUsage — инструмент, отображающий информацию об использовании GDI-объектов всеми приложениями (см. статью «Resource Leaks: Detecting, Locating, and Repairing Your Leaky GDI Code» по ссылке http://msdn.microsoft.com/msdnmag/issues/01/03/leaks/leaks.asp). Нынешняя статья посвящена аналогичному инструменту для Windows XP. Метод, описанный здесь, также применим к Windows 2000 и Windows NT 4.0, но я буду говорить только о Windows XP.

Обсудив различия между платформами Windows 9*x* и Windows XP, я расскажу, как решить проблемы, возникающие при реализации инструмента, изображенного на **рис. 1**. Кроме того, я научу вас получать сведения о потребляемых процессом GDI-ресурсах с помощью механизма внедрения кода (code injection) и так модифицировать процесс или DLL, чтобы получать уведомления о создании GDI-объекта. Затем я покажу, как написать Win32-отладчик, управляющий процессом, как обеспечить взаимодействие между процессом и отладчиком и как реализовать диспетчер стека вызовов (call stack manager), предоставляющий дополнительные сведения о выделении GDI-ресурсов.

Windows 9*x* и Windows XP

В Windows XP с каждым процессом связан список GDI-объектов, управляемый в основном в режиме ядра драйвером устройства win32k.sys, который отвечает за реализацию подсис-

тем USER и GDI. Win32-приложения могут обращаться к этим системным сервисам через API, предоставляемый user32.dll и gdi32.dll. Так как список GDI-объектов ведется для каждого процесса по отдельности, передавать данный GDI-объект GDI-функциям может только то приложение, в котором он был создан.

Версия GDIUsage для Windows 9*x* использует API-функцию GetObjectType, которая возвращает по описателю тип GDI-объекта и проверяет, является ли переданное ей значение действительным GDI-описателем. Но в Windows XP в отличие от Windows 9*x* описатель GDI-объекта — 32-битное значение с возможным диапазоном от 0 до 0xFFFFFFFF, и для получения списка существующих GDI-объектов вы должны передать функции GetObjectType все значения из данного диапазона. С точки зрения производительности, это настоящая проблема. Время выполнения приведенного ниже кода исчисляется минутами, и, увы, очень много GDI-объектов, обнаруженных путем прогона тестового приложения, оказались недействительными (свыше 500!):

```
DWORD dwObjectType;
DWORD hGdi;
for (hGdi = 0; hGdi < 0xFFFFFFFF; hGdi++)
{
   dwObjectType = ::GetObjectType((HANDLE)hGdi);
   if (dwObjectType != 0)
   {
      TRACE("0x%08x -> %u\n", hGdi, dwObjectType);
   }
}
```

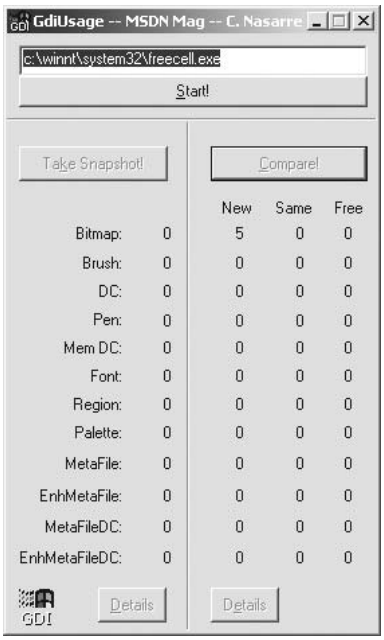


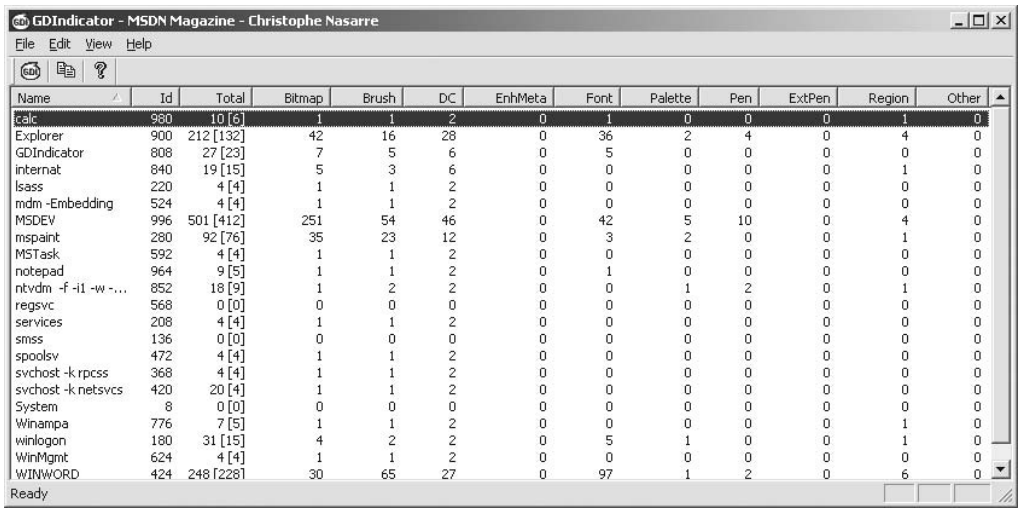
Рис. 1. Использование GDI-ресурсов в Windows 2000

Это значит, что для получения правильного списка объектов нужны дополнительные (и, вероятно, длительные) проверки с применением GetObject. Большое время выполнения цикла делает данный метод непригодным. В этой статье я рассмотрю два других метода. Первый использует таблицу GDI-описателей, а второй работает путем установки ловушек (hooks) на функции GDI API. Замечу, что первый метод официально не поддерживается и его работоспособность в следующих версиях ОС не гарантируется. Отсюда вытекает первая задача: как получить список действительных GDI-объектов более эффективным способом?

С помощью GDIUsage на платформе Windows 9*x* можно отображать GDI-объекты, используя функции, соответствующие типу каждого такого объекта, например, BitBlt для битовой карты или FillRect для кисти. Но поскольку эти API-функции не работают с GDI-объектами, созданными в другом процессе, исчезает ключевая характеристика моего инструмента — возможность «видеть» утечку ресурсов. С другой стороны, код отображения прекрасно работает в приложении, создавшем данный GDI-объект. Решение очевидно — код отображения должен выполняться в контексте другого процесса. Конкретную реализацию, использующую Windows-ловушки в качестве механизма межпроцессного взаимодействия, я опишу далее.

Рис. 2. Структура PEB, полученная с помощью WinDbg и kdex2x86

```
0:000> !kdex2x86.strct PEB
Loaded kdex2x86 extension DLL
struct   _PEB (sizeof=488)
+000 byte     InheritedAddressSpace
+001 byte     ReadImageFileExecOptions
+002 byte     BeingDebugged
+003 byte     SpareBool
+004 void     *Mutant
+008 void     *ImageBaseAddress
+00c struct   _PEB_LDR_DATA *Ldr
+010 struct   _RTL_USER_PROCESS_PARAMETERS *ProcessParameters
+014 void     *SubSystemData
+018 void     *ProcessHeap
+01c void     *FastPebLock
+020 void     *FastPebLockRoutine
+024 void     *FastPebUnlockRoutine
+028 uint32   EnvironmentUpdateCount
+02c void     *KernelCallbackTable
+030 uint32   SystemReserved[2]
+038 struct   _PEB_FREE_BLOCK *FreeList
+03c uint32   TlsExpansionCounter
+040 void     *TlsBitmap
+044 uint32   TlsBitmapBits[2]
+04c void     *ReadOnlySharedMemoryBase
+050 void     *ReadOnlySharedMemoryHeap
+054 void     **ReadOnlyStaticServerData
+058 void     *AnsiCodePageData
+05c void     *OemCodePageData
+060 void     *UnicodeCaseTableData
+064 uint32   NumberOfProcessors
+068 uint32   NtGlobalFlag
+070 union    _LARGE_INTEGER CriticalSectionTimeout
+070 uint32   LowPart
+074 int32    HighPart
+070 struct   __unnamed3 u
+070 uint32   LowPart
+074 int32    HighPart
+070 int64    QuadPart
+078 uint32   HeapSegmentReserve
+07c uint32   HeapSegmentCommit
+080 uint32   HeapDeCommitTotalFreeThreshold
+084 uint32   HeapDeCommitFreeBlockThreshold
+088 uint32   NumberOfHeaps
+08c uint32   MaximumNumberOfHeaps
+090 void     **ProcessHeaps
+094 void     *GdiSharedHandleTable
+098 void     *ProcessStarterHelper
+09c uint32   GdiDCAttributeList
+0a0 void     *LoaderLock
+0a4 uint32   OSMajorVersion
+0a8 uint32   OSMinorVersion
+0ac uint16   OSBuildNumber
+0ae uint16   OSCSDVersion
+0b0 uint32   OSPlatformId
+0b4 uint32   ImageSubsystem
+0b8 uint32   ImageSubsystemMajorVersion
+0bc uint32   ImageSubsystemMinorVersion
+0c0 uint32   ImageProcessAffinityMask
+0c4 uint32   GdiHandleBuffer[34]
+14c function *PostProcessInitRoutine
+150 void     *TlsExpansionBitmap
+154 uint32   TlsExpansionBitmapBits[32]
+1d4 uint32   SessionId
+1d8 void     *AppCompatInfo
+1dc struct   _UNICODE_STRING CSDVersion
+1dc uint16   Length
+1de uint16   MaximumLength
+1e0 uint16   *Buffer
```



Name	Id	Total	Bitmap	Brush	DC	EnhMeta	Font	Palette	Pen	ExtPen	Region	Other
calc	980	10 [6]	1	1	2	0	1	0	0	0	1	0
Explorer	900	212 [132]	42	16	28	0	36	2	4	0	4	0
GDIndicator	808	27 [23]	7	5	6	0	5	0	0	0	0	0
internat	840	19 [15]	5	3	6	0	0	0	0	0	1	0
lsass	220	4 [4]	1	1	2	0	0	0	0	0	0	0
mdm -Embedding	524	4 [4]	1	1	2	0	0	0	0	0	0	0
MSDEV	996	501 [412]	251	54	46	0	42	5	10	0	4	0
mspaint	280	92 [76]	35	23	12	0	3	2	0	0	1	0
MSTask	592	4 [4]	1	1	2	0	0	0	0	0	0	0
notepad	964	9 [5]	1	1	2	0	1	0	0	0	0	0
ntvdm -f -i1 -w -...	852	18 [9]	1	2	2	0	0	1	2	0	1	0
regsvc	568	0 [0]	0	0	0	0	0	0	0	0	0	0
services	208	4 [4]	1	1	2	0	0	0	0	0	0	0
smss	136	0 [0]	0	0	0	0	0	0	0	0	0	0
spoolsv	472	4 [4]	1	1	2	0	0	0	0	0	0	0
svchost -k rpcss	368	4 [4]	1	1	2	0	0	0	0	0	0	0
svchost -k netsvcs	420	20 [4]	1	1	2	0	0	0	0	0	0	0
System	8	0 [0]	0	0	0	0	0	0	0	0	0	0
Winampa	776	7 [5]	1	1	2	0	0	0	0	0	1	0
winlogon	180	31 [15]	4	2	2	0	5	1	0	0	1	0
WinMgmt	624	4 [4]	1	1	2	0	0	0	0	0	0	0
WINWORD	424	248 [228]	30	65	27	0	97	1	2	0	6	0

Рис. 3. Использование GDI-объектов выполняемыми процессами

Наконец, применив Win32-функции отладки в качестве связующего звена между кодом, выполняемым в разных процессах, мы напишем версию GDIUsage, работающую в Windows XP и на других 32-разрядных Windows-платформах.

Как GDI управляет описателями

В статье, опубликованной в номере за август 2002 г. (http://msdn.microsoft.com/msdnmag/issues/02/08/escapefromDLLHell), я получал описание структуры Process Environment Block (PEB) через WinDBG. Для нас особый интерес представляет поле GdiSharedHandleTable этой структуры (**рис. 2**). На самом деле это указатель на таблицу, где GDI хранит свои описатели, в том числе и созданные в других процессах. Фен Хуань (Feng Huan) в книге» «Windows Graphics Programming: Win32 GDI and Direct Draw» (Prentice Hall, 2002) описывает другой способ доступа к этой таблице, а также рассматривает структуру ее записей (всего их — 0x4000):

```
typedef struct
{
  DWORD pKernelInfo;
  // Это формат 2000/XP, в Windows NT у полей обратный порядок
  WORD  ProcessID;
  WORD  _nCount;
  WORD  nUpper;
  WORD  nType;
  DWORD pUserInfo;
} GDITableEntry;
```

Каждая запись содержит сведения о GDI-описателе, значение которого легко вычислить. Младшие 16 битов — это индекс записи таблицы, а старшие 16 битов хранятся в поле nUpper. Как следует из названия, поле ProcessID содержит идентификатор процесса, создавшего объект. Вооружившись этой информацией, можно написать цикл для перечисления объектов, используемых данным процессом, — именно это и делает GDIndicator (**рис. 3**).

Если вас интересуют разные способы получения списка выполняемых процессов, прочитайте мою статью в номере за июнь 2002 г. (http://msdn.microsoft.com/msdnmag/issues/02/06/debug). У каждого процесса есть идентификатор; сравнивая его с полем ProcessID, вы можете выбрать из общей таблицы GDI-объекты, принадлежащие этому процессу. Именно так и формируются значения счетчиков, отображаемые GDIndicator в колонках, соответствующих объектам разных типов.

В отличие от остальных колонок третья содержит два значения. Первое — результат вызова GetGuiResources (которая возвращает число описателей GDI-объектов, используемых процессом), а второе (в скобках) — сумма, полученная при просмотре общей таблицы GDI-описателей. Как видно на **рис. 3**, эти значения часто отличаются, причем GetGuiResources в таких случаях всегда возвращает большее значение. Никакого объяснения этому различию в документации нет, как нет и никакой очевидной его связи с предопределенными (stock) или неосвобожденными объектами. Возможно, GDI скрытно создает объекты, которые не хранятся в общей таблице и являются объектами, за которые вы не отвечаете.

Один из случаев такого скрытого выделения ресурсов имеет место при операциях со значками (icons). Когда вы создаете или загружаете значок, Windows для реализации эффекта прозрачности требуется два растровых изображения. Обычно одно из них — маска, другое — видимый рисунок. В отличие от битовых карт обработку значков поддерживает подсистема USER, а не GDI. Возможно, поэтому такие ресурсы и не учитываются в общей таблице GDI-описателей.

Отслеживание создания объектов с помощью API-ловушек

Как видите, узнать, какие GDI-объекты используются данным процессом, нелегко. А как узнать, кем создан объект: ко-

Табл. 1. Функции создания GDI-объектов

Тип объекта	API-функции
Битовая карта	LoadBitmapA, LoadBitmapW, CreateBitmap, CreateBitmapIndirect, CreateCompatibleBitmap
Кисть	CreateBrushIndirect, CreateSolidBrush, CreatePatternBrush, CreateDIBPatternBrush, CreateDIBPatternBrushPt, CreateHatchBrush
Контекст устройства	CreateCompatibleDC, CreateDCA, CreateDCW, CreateICA, CreateICW, GetDC, GetDCEx, GetWindowDC
Шрифт	CreateFontA, CreateFontW, CreateFontIndirectA, CreateFontIndirectW
Метафайл	CreateMetaFileA, CreateMetaFileW, CreateEnhMetaFileA, CreateEnhMetaFileW, GetEnhMetaFileA, GetEnhMetaFileW, GetMetaFileA, GetMetaFileW
Перо	CreatePen, CreatePenIndirect, ExtCreatePen
Регион	PathToRegion, CreateEllipticRgn, CreateEllipticRgnIndirect, CreatePolygonRgn, CreatePolyPolygonRgn, CreateRectRgn, CreateRectRgnIndirect, CreateRoundRectRgn, ExtCreateRegion
Палитра	CreateHalftonePalette, CreatePalette

дом приложения или самой подсистемой GDI? Если бы можно было получать уведомление о создании GDI-объекта, тогда можно было бы сохранить его описатель и таким образом вести список объектов, выделенных приложением. Увы, подобного механизма уведомлений в Win32 API нет.

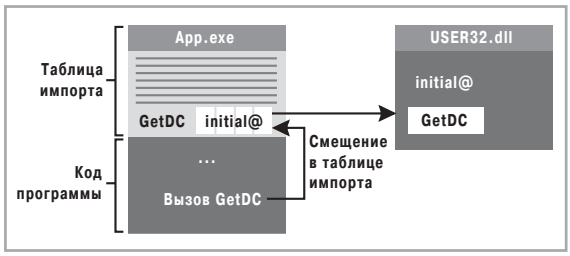


Рис. 4. Схема вызова функции GetDC

Чтобы определять момент создания нового объекта, придется перехватывать вызовы функций, перечисленных в **табл. 1**.

К счастью, Мэт Петрек (Matt Pietrek) в статье «Learn System-Level Win32 Coding Techniques by Writing an API Spy Program» в номере «MSJ» за декабрь 1994 г. рассказал, как реализовать в Win32 механизм слежения за вызовами API-функций. Для конкретного модуля (процесса или DLL) данный механизм позволяет подменить адрес вызываемой функции (или экспортируемой из DLL) адресом вашей функции. После этого при всяком вызове такой функции из наблюдаемого модуля вместо нее будет вызываться ваш обработчик.

С течением времени данный принцип перехвата вызовов API совершенствовался [см. рубрику «Bugslayer» Джона Роббинса (John Robbins) в номерах MSJ за февраль 1998 г. и июнь 1999 г. – http://www.microsoft.com/ msj/0298/bugslayer0298.htm и http://www.microsoft.com/msj/0699/Bugslayer/Bugslayer0699.htm], а возможные реализации для разных Windows-платформ можно найти в главе 22 книги Джеффри Рихтера «Microsoft Windows для профессионалов», изд. 4-е (Русская Редакция, М., 2000) и в книге Джона Роббинса «Debugging Applications» (Microsoft Press, 2000). В данном случае я использовал метод Джона Роббинса.

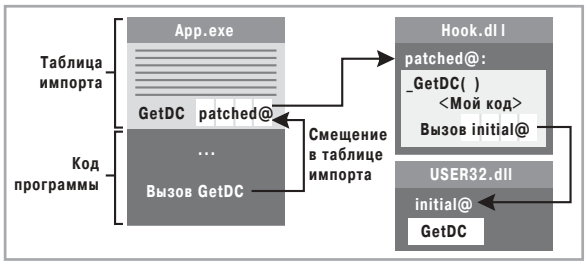


Рис. 5. Схема вызова перехваченной функции GetDC

Предложенная Джоном вспомогательная функция HookImportedFunctionsByName принимает список функций, вызовы которых должны быть перехвачены, загрузочный адрес системной DLL, которая их экспортирует, модуль, вызывающий эти функции, и список обработчиков, которым эти вызовы должны быть перенаправлены. Завершив свою работу, она возвращает список адресов перехваченных функций. Так, если App.exe вызывает GetDC из USSER32.DLL, карта памяти будет такой, как показано на **рис. 4**. Если же вызвать HookImportedFunctionsByName с параметрами, приведенными ниже, мы получим другую карту памяти (**рис. 5**):

- список перехватываемых системных функций (GetDC);
- адрес DLL (USER32.dll), экспортирующей функции;
- вызывающий модуль (App.exe вызывает GetDC напрямую);
- список функций-обработчиков (_GetDC из Hook.dll).

В данном примере список адресов перехваченных функций будет состоять из initial@.

Помимо функций, приведенных в **табл. 1**, тот же механизм позволяет перехватывать функции, освобождающие GDI-объекты (**табл. 2**). Получая уведомления как о создании, так и об удалении объектов, можно вести список текущих активных GDI-объектов.

Класс CGDIReflect предоставляет статические методы-интерфейсы (stub methods), вызываемые вместо системных функций. Он наследует от CAPIReflect, основная задача которого – перенаправлять с помощью макросов вызов функции из данного модуля статическому члену класса. Замещение выполняется вызовом DoReflect с передачей в качестве параметра описателя вызывающего модуля. Производный класс отвечает, как будет показано далее, за связывание каждой интересующей вас системной функции с соответствующей интерфейсной функцией (stub function).

Помимо механизма карты сообщений, определен и набор макросов, автоматизирующих объявление и описание интерфейсных функций. Параметр /P компилятора позволяет для каждого исходного файла получить файл с расширением .i, содержащий исходный код после раскрытия всех макросов. Эти файлы бывают очень большими, зато они позволяют точно определить, какой код будет выполняться на самом деле.

Перехват функций USER или GDI требует выполнения трех операций. Используя тот же пример, я покажу, как перехватить вызовы GetDC в user.dll.

Первый шаг – объявление в CGDIReflect статического метода с применением макроса DECLARE_REFLECT_API*xxx*, где *xxx* – число параметров функции (для GetDC, принимающей в качестве параметра HWND, это значение равно 1). Данное объявление помещается между BEGIN_REFLECT_LIST (он определяет скрытый вспомогательный метод TraceReflectCall, предоставляющий сервисы трассировки) и END_REFLECT_LIST (который не делает ровным счетом ничего):

```
BEGIN_REFLECT_LIST()
  DECLARE_REFLECT_API1(GetDC, HDC, HWND, hWnd)
END_REFLECT_LIST()
```

Однако макросу DECLARE_REFLECT_API нужно передать имя системной функции, тип ее возвращаемого значения и список параметров (с типами и именами). На основе этой информации макрос раскрывается в статический метод класса CGDIReflect (фактически это метод-интерфейс) с тем же прототипом, и он делает следующее. Во-первых, исходная системная функция вызывается через псевдоним (экземпляр которого создается позднее с помощью DEFINE_API_REFLECT). Затем описатель, созданный этим вызовом, и его тип сохраняются в структуре CHandleInfo (**рис. 6**) вместе с

Табл. 2. Функции для освобождения GDI-объектов

Тип объекта	API-функции для освобождения описателей
Контекст устройства	DeleteDC
Метафайл	DeleteMetaFile, CloseMetaFile
Расширенный метафайл	DeleteEnhMetaFile, CloseEnhMetaFile
Другие	DeleteObject

Рис. 6. Объявление класса CHandleInfo

```
class CHandleInfo
{
public:
   CHandleInfo()
   {
      m_hObject = NULL;  // описатель
      m_Type    = 0;     // тип (битовая карта, кисть...)
      m_pStack  = NULL;  // стек вызовов
      m_Depth   = 0;     // глубина стека вызовов
   }

   virtual ~CHandleInfo()
   {
      if (m_pStack != NULL)
         delete [] m_pStack;
   }

public:
   HANDLE   m_hObject;
   DWORD    m_Type;
   DWORD    m_Depth;
   DWORD*   m_pStack;
};
```

дополнительными данными, которые понадобятся впоследствии (см. обсуждение DoStackAddressDump в следующем разделе). Наконец, только что созданный описатель и ранее упомянутая структура связываются между собой и запоминаются в статическом члене-словаре (static map member) класса CGDIReflect.

На втором шаге каждый статический член объявляется псевдонимом адреса перехватываемой системной функции через макрос:

```
DEFINE_API_REFLECT(CGDIReflect, GetDC);
```

В моем примере он раскрывается в:

```
CGDIReflect::GetDCProc
CGDIReflect::__GetDC = 0;
```

На третьем и последнем шаге нужно создать все указанные члены, а потом использовать их при выполнении. Метод FillStubs вызывается в обоих случаях. Во-первых, он вызывается из функции Init с APIR_STATE_INIT в качестве параметра и без описателя модуля, вызовы которого будут перехватываться. Это происходит в тот момент, когда с помощью GetProcAddress вычисляется адрес системной функции [после чего он сохраняется в члене-псевдониме (__GetDC для GetDC)].

Далее FillStubs вызывается, когда возникает необходимость в перехвате вызовов новой DLL. DoReflect вызывает FillStubs, передавая в качестве параметра APIR_STATE_ENABLE. Таким образом, каждый системный вызов (GetDC), выполняемый модулем, загрузочный адрес которого передан как параметр, перенаправляется на статический метод-интерфейс (в моем примере — на _GetDC). Описанный способ перехвата вызовов похож на карту сообщений MFC:

```
BEGIN_IMPLEMENT_API_REFLECT()
  ...
  IMPLEMENT_API_REFLECT(hModule, "USER32.DLL", GetDC);
  ...
END_IMPLEMENT_API_REFLECT()
```

Чтобы упростить отладку макросов в период выполнения, в коде, получаемом после их раскрытия, размещен ряд вызовов AfxTrace. SetTraceLevel позволяет выбирать, какая операция подлежит трассировке (**табл. 3**).

Теперь у вас есть класс CGDIReflect, позволяющий перенаправить любые вызовы из данного модуля методу-интерфейсу, единственная задача которого — сохранять создаваемые GDI-объекты в словаре. Однако у этой реализации есть один минус: она не поддерживает многопоточность. Если исследуемое вами приложение — многопоточное и GDI-функции вызываются несколькими потоками, поведение может стать непредсказуемым из-за отсутствия синхронизации доступа к словарю описателей из разных потоков.

Мониторинг создания GDI-объектов с помощью стека вызовов

При каждом вызове любой из интересующих нас функций вызывается обработчик, который выполняет две операции. Во-первых, он включает только что созданный описатель и его тип в объект-оболочку CHandleInfo, объявленный в GDIReflect.h. Вторая операция интереснее. В объекте CHandleInfo сохраняются адреса всех функций из текущего стека вызовов, для чего применяется динамически создаваемый массив m_pStack типа DWORD (его размер хранится в m_Depth). Таким образом, помимо графического представления GDI-объекта, можно показывать и стек вызовов, приведших к его созданию (**рис. 7**).

Просмотреть стек вызовов позволяют imagehlp.dll и dbghelp.dll. Еще больше облегчает эту задачу разработанный Джоном Роббинсом класс CSymbolEngine, развитие которого описано в рубриках «Bugslayer» в «MSJ» за апрель 1998 г. (http://www.microsoft.com/msj/0498/bugslayer0498.htm) и февраль 1999 г. (http://www.microsoft.com/msj/0299/bugslayer/bugslayer0299.htm). Последняя версия, использующая DBGHELP, подробно рассмотрена в его книге «Debugging Applications», о которой я уже упоминал.

Класс CSymbolEngine — низкоуровневая оболочка многочисленных функций, экспортируемых DBGHELP. Класс CStackManager, реализованный в StackManager.cpp (см. каталог \Common на http://msdn.microsoft.com/msdnmag/code03.aspx), предоставляет средства более высокого уровня, из которых для нас представляют интерес лишь три метода. Первый — DoStackAddressDump — с помощью CSymbolEngine выделяет память для хранения содержимого текущего стека вызовов и заполняет ее адресами функций. Этот метод вызывается всеми нашими обработчиками и сохраняет ад-

Табл. 3. Флаги для SetTraceLevel

Флаг	Описание
API_TRACE_LEVEL_NONE	Трассировка отключена.
API_TRACE_LEVEL_GETPROCADDRESS	Трассировка при перехвате GDI-функции.
API_TRACE_LEVEL_GETMODULEHANDLE	Проверять, есть ли модуль, вызовы которого будут перехватываться.
API_TRACE_LEVEL_REFLECT_CALL	Уведомлять о перенаправлении вызова интерфейсной функции.
API_TRACE_LEVEL_DOREFLECT	Показывать, какой модуль обработан в DoReflect.
API_TRACE_LEVEL_INIT	Трассировка FillStub при получении адреса GDI-функции.
API_TRACE_LEVEL_HOOK_IMPORTED_FUNCTION	Регистрировать, какие GDI-функции перехватываются для данного модуля.

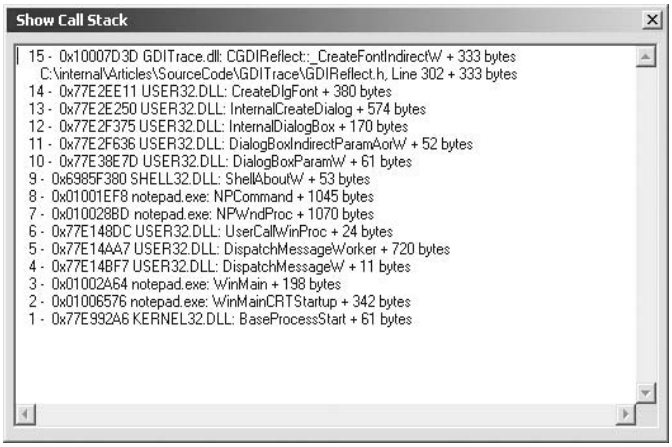


Рис. 7. Стек вызовов, приводящих к созданию GDI-объекта

рес каждой функции, выполнение которой привело к созданию данного объекта.

С шестнадцатеричными адресами легко работать компьютеру, но не человеку. Преобразовать массив адресов, сгенерированный DoStackAddressDump, в удобный формат (**рис. 7**) позволяет метод DumpStackAllocation. Он принимает на входе дамп стека и его размер и возвращает транслированный стек в CString. При вызове метода можно выбрать тип разделителя строк: либо \r\n для отображения CString в поле ввода, либо \n для вывода с помощью Trace или OutputDebugString. Никаких чудес в этом методе нет — для каждого адреса в массиве он вызывает ConvertStackAddressIntoFunctionName.

Чудеса в другом. Создавая дамп стека и сохраняя очередной адрес в выходном массиве, DoStackAddressDump также находит соответствующий этому адресу символ с помощью SymGetModuleInfo, SymGetSymFromAddr и SymGetLineFromAddr, определенных в CSymbolEngine (детали реализации см. в ConvertAddress в файле StackManager.cpp). Почему трансляция адреса выполняется именно на этом этапе? Ответ прост: только сейчас можно быть твердо уверенным, что соответствующая DLL-библиотека загружена в память, а когда впоследствии вызывается DumpStackAllocation, этой библиотеки может и не быть.

Если GDI-объекты создаются часто, в m_HandleMap будет сохранено много дампов стека. Однако объект CHandleInfo в этом словаре содержит массив адресов, а не массив транслированных строк. Весь трюк в использовании переменной-члена m_AddressToName. Она позволяет избежать хранения в памяти длинных строк для каждого адреса в дампе стека вместо коротких DWORD и тем самым резко уменьшить объем требуемой памяти. Еще одно преимущество — более быстрое создание дампа, так как m_AddressToName играет роль кэша, позволяющего уйти от повторных вычислений тех данных, которые были получены ранее.

Мы уже знаем, как получить список вызываемых GDI-функций, но нам по-прежнему нужно определить, из какого модуля они вызываются. Допустим, приложение, которое использует разделяемую MFC DLL, создает объект CPen для API-вызовов, работающих с перьями. Вызов CreatePen (возвращающий описатель пера) фактически выполняется в MFC DLL, а не в коде приложения. Перехватывая только API-вызовы из EXE-модуля, мы пропустим вызовы из используемой DLL.

Определение списка DLL с помощью API отладки

PSAPI-функция EnumProcessModules позволяет получить в Windows XP список всех DLL, загруженных процессом, очень легко; как это делается, рассказал Мэт Петрек (см. рубрику «Under the Hood» в «MSJ» за август 1996 г. по ссылке http://www.microsoft.com/msj/archive/S402.htm). Однако в случае библиотек, загружаемых динамически, наша задача усложняется. Чтобы определить, какие DLL загружает программа, нам придется перехватывать не только вызовы GDI-функций, но и вызовы из главной программы функции LoadLibrary (как ANSI-, так и Unicode-версии).

Остается решить два вопроса. Во-первых, как определить, какие DLL модифицировать в исследуемом процессе? Во-вторых, как обеспечить выполнение нашего кода внутри другого процесса? Если удастся найти такое решение, мы сможем получать графическое представление любого описателя GDI-объекта. В статье по отладке (опубликованной в августе 2002 г.) для определения списка DLL, загружаемых процессом статически или динамически, применялись отладочные Win32-функции. Основной недостаток этого метода — необходимость запуска исследуемого приложения в отладочном режиме.

Используя Win32-функции отладки, можно написать код, который запускает приложение как отлаживаемое, а затем получать уведомления о происходящих в нем событиях, например о загрузке новой DLL. А это как раз то, что нужно. Чтобы написать несложный отладчик, вы должны просто перегрузить виртуальные методы, вызываемые при возникновении отладочных событий. Класс CGDIDebugger наследует от класса CApplicationDebugger, о котором я уже рассказывал в статье за август 2002 г. В **табл. 4** перечислены методы, перегружаемые CGDIDebugger. Механизм взаимодействия отладчика и отлаживаемой программы мы рассмотрим позже.

Кроме этих методов, перегружен и OnOutputDebugStringDebugEvent — это позволяет помещать трассировочные сообщения, выводимые отлаживаемым процессом, в специальное окно списка, которое также поддерживает поиск строки и копирование выделенного текста в буфер обмена (**рис. 8**). При выполнении трассировочных вызовов с помощью TRACE или OutputDebugString результаты отображаются именно в этом окне. Это эффективное средство отладки, также позволяющее отличить вывод отладчика от вывода отлаживаемого приложения (по префиксу >).

Табл. 4. Методы, перегруженные CGDIDebugger

Перегруженный метод	Описание
PreLoadingProcess	Нужен для инициализации членов класса.
OnCreateProcessDebugEvent	Вызывает StartTraceGDI и внедряет код DLL в отлаживаемую программу.
OnExitProcessDebugEvent	Вызывает StopTraceGDI и выполняет очистку по завершении сеанса отладки.
OnLoadDLLDebugEvent	Вызывает PatchModule и добавляет DLL во внутренний список, уведомляя код в отлаживаемом процессе о необходимости перехвата GDI-вызовов из этой DLL.
OnUnloadDLLDebugEvent	Нужен для обновления внутреннего списка DLL.
OnThreadMessage	Принимает уведомление от кода, внедренного в отлаживаемый процесс.

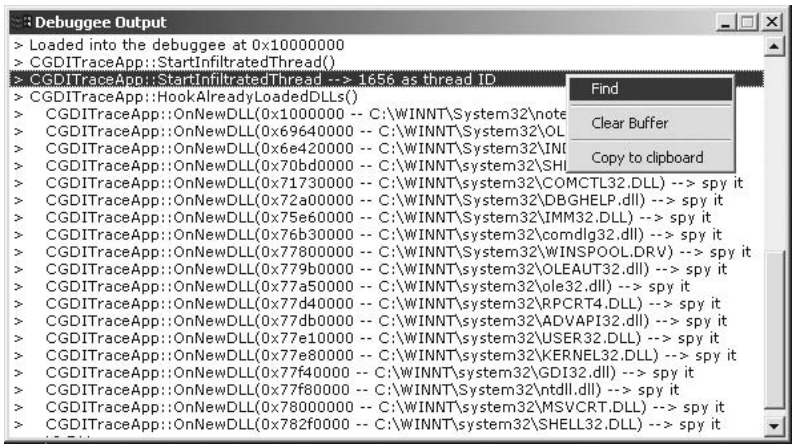


Рис. 8. Поиск строки

Внедрение кода в другой процесс

Остается проблема выполнения нашего кода в контексте другого приложения. К счастью, ее решение давным-давно описано Джеффри Рихтером в статье «Load Your 32-bit DLL into Another Process's Address Space Using INJLIBO» («MSJ» за май 1994 г.).

Поскольку обычно нас будут интересовать только приложения, использующие GDI API, можно предположить, что такое приложение создает хотя бы одно окно для отображения своих графических объектов. (Иначе зачем ему GDI?) Так что из всех решений наилучшим, похоже, является вариант с Windows-ловушками. В результате следующего вызова Windows будет обращаться к вашей функции обратного вызова GetMessageHookProc всякий раз, когда любой поток любого процесса будет выполнять GetMessage:

```
SetWindowsHookEx(WH_GETMESSAGE, GetMessageHookProc,
    hInstance, 0)
```

Это общесистемная ловушка (последний параметр равен 0), и код функции обратного вызова должен находиться в DLL, проецируемой на адресное пространство каждого процесса, поток которого вызывает GetMessage.

Так как мы контролируем и процедуру ловушки, и приложение-драйвер, то можем легко установить между ними канал связи посредством предопределенных сообщений. Поэтому из отладчика можно будет посылать запросы внедренному коду, выполняемому в контексте исследуемого приложения, — как раз то, что нужно для реализации диалогового окна отображения GDI-объекта! При перехвате процедурой ловушки первого сообщения она сначала перехватывает вызовы для уже загруженных DLL. Потом запускает новый поток для обработки запросов от отладчика. Таким образом, устанавливается канал связи между отладчиком и отлаживаемой программой (детали см. в StartInfiltratedThread в файле GDITrace.cpp).

Функции, которые обращаются к внедренной функции потока и вызываются отладчиком и процедурой ловушки в отлаживаемой программе, находятся в GDITrace.dll и реализованы в классе CGDITraceApp. Эта DLL статически связана с приложением-отладчиком GDIUsage, но загружается в процесс, вызвавший ловушку, динамически. Чтобы помочь вам разобраться в том, какая часть GDITrace используется в отладчике, а какая — в отлаживаемом процессе, функции, предназначенные для вызова отладчиком, объявлены в _GdiTrace.h и собраны вместе в GdiTrace.cpp. Но почему столь разные функции хранятся в одной DLL? Некоторые переменные должны быть доступны коду обоих типов, а реализовать совместное использование значений какой-либо переменной разными экземплярами одной DLL несложно (**рис. 9**).

Рис. 9. Использование общих переменных разными процессами

```
// Общие переменные
//
#pragma data_seg(".shared")
// Отладчик
   DWORD s_dwCallingThreadID = 0;

// Отлаживаемый процесс
   DWORD s_dwInfiltratedThreadID = 0;
   DWORD s_dwProcessID           = 0;

// Флаг инициализации перехвата API-вызовов
// устанавливается в TRUE при первом вызове
// внедренной функции-ловушки
   BOOL  s_bDebuggeeIsStarted = FALSE;
#pragma data_seg()

// Будьте внимательны - никаких лишних
// пробелов в следующей директиве быть не должно!
#pragma comment(linker,"-section:.shared,rws")
```

Здесь определяется раздел PE-файла с именем .shared и атрибутами read/write/shared (rws), содержащий пять разделяемых переменных с префиксом s_. Согласно данному объявлению Windows разместит эти переменные в блоке памяти, общем для всех процессов, в которые загружена данная DLL. Таким образом, эти переменные будут иметь одинаковые значения во всех процессах, в том числе в отладчике и отлаживаемой программе. Посмотрим, что происходит, когда отладчик запускает отлаживаемую программу, и как используются разделяемые переменные.

При старте отлаживаемой программы поток отладчика получает событие CREATE_PROCESS_DEBUG_EVENT, обрабатываемое в OnCreateProcessDebugEvent, которая в свою очередь вызывает StartTraceGDI. Данная функция вызывает SetSharedVariables, помещая идентификатор потока отладчика в s_dwCallingThreadID. Если идентификатор текущего процесса совпадает со значением s_dwProcessID, значит, процедура ловушки выполняется в контексте отлаживаемого приложения, и она начинает перехватывать GDI-вызовы из уже загруженных DLL. Далее процедура-ловушка запускает поток, который сохраняет свой идентификатор в s_dwInfiltratedThreadID. Наконец, когда выполнение процедуры-ловушки завершается успешно, переменная s_bDebuggeeIsStarted устанавливается в TRUE, что позволяет отладчику узнать, готов ли поток внедренного кода к обработке запросов.

Так как мы собираемся передавать между отлаживаемой программой и отладчиком список описателей GDI-объектов, нам нужен буфер больший, чем просто DWORD или BOOL. В дополнение к упомянутым пяти переменным используется проецируемый в память файл (memory mapped file) GDITrace_SharedBuffer, на проекцию которого указывает член m_lpvMem класса CGDITraceApp. Этот буфер инициализируется при инициализации DLL (детали см. в CGDITraceApp::InitInstance в файле GDITrace.cpp). Такая операция выполняется, только когда DLL загружаетсяв процесс GDIUsage как отладчик и в процесс отлаживаемого им приложения.

Разделяемая переменная s_dwProcessID позволяет различать эти два процесса. Пока отлаживаемый процесс не запущен, она равна 0, иначе она содержит идентификатор отлаживаемого процесса. Когда DLL загружена в какой-либо процесс, функция InitInstance перед созданием проецируемого в память файла проверяет, чему равна s_dwProcessID: 0 (GDIUsage) или значению GetCurrentProcessId (отлаживаемый процесс).

Взаимодействие между отладчиком и отлаживаемым процессом

Общая переменная s_dwInfiltratedThreadID используется отладчиком для асинхронной передачи запроса (простого Windows-сообщения TM_GET_LIST) через PostThreadMessage потоку, внедренному в отлаживаемый процесс. Другая общая переменная, s_dwCallingThreadID, нужна, когда отлаживаемый процесс должен уведомить отладчик о выполнении такого запроса. Скажем, когда пользователь щелкает кнопку «Take Snapshot!», GDIUsage необходимо получить из отлаживаемого процесса список созданных GDI-объектов.

Итак, GDIUsage посылает сообщение TM_GET_LIST потоку, внедренному в отлаживаемый процесс. Это приводит к вызову функции OnGetList с параметром UM_SNAPSHOT_READY, который выступает в роли ответного сообщения главному диалогу GDIUsage. Почему просто не взять то же сообщение TM_GET_LIST? Все дело в разделении кода. Для обновления соответствующей части UI-интерфейса GDIUsage список созданных GDI-объектов нужно получать при нажатии как кнопки «Take Snapshot!», так и кнопки «Compare» (хотя сам список требуется для других целей).

Подведем итоги. Сообщение TM_GET_LIST, направляемое потоку, запускает обработку GDI-объектов на стороне отлаживаемой программы. Кроме того, содержимое главного диалога GDIUsage можно обновлять двумя способами на основе двух пользовательских сообщений: UM_SNAPSHOT_READY или UM_COMPARE_READY.

Получив сообщение, внедренный поток пробуждается и вызывает для обработки запроса OnGetList. Этот метод класса CGDITraceApp просматривает список GDI-объектов, о создании которых внутри отлаживаемого процесса мы узнаем через модифицированные интерфейсные функции (patched stubs), и копирует данные каждого объекта (в формате GDI_LIST, приведенном ниже) в разделяемый буфер (проецируемый файл), на который указывает m_lpvMem:

```
typedef struct
{
   DWORD    dwType;
   HGDIOBJ  hObject;
} GDI_ITEM;

typedef struct
{
   DWORD    dwCount; // число заполненных элементов
                     // GDI_ITEM в Items
   GDI_ITEM Items[];
} GDI_LIST;
```

Чтобы сообщить отладчику о том, что список готов, то же самое сообщение TM_GET_LIST посылается обратно потоку с идентификатором s_dwCallingThreadID. Это сообщение принимается в контексте GDIUsage потоком, отвечающим за отладочные события, и перенаправляется в CGDIDebugger::OnThreadMessage. Данный метод уведомляет поток пользовательского интерфейса (UI), посылая главному диалогу соответствующее сообщение (в данном случае UM_SNAPSHOT_READY) с указателем на разделяемую память (на основе проецируемого файла), куда отлаживаемый процесс поместил информацию о GDI-объектах. Переданный таким образом отладчику список GDI-объектов используется при создании экземпляра CGdiResources его методом CreateFromList. Данный класс служит оболочкой списка GDI-объектов и предоставляет такие сервисы, как перечисление элементов списка и их отображение на экране.

Проблемы синхронизации

Прежде чем перейти к реализации отображения удаленного GDI-объекта, следует учесть вероятность взаимной блокировки (deadlock). Описанный выше метод сбора GDI-объектов является асинхронным, так как построен на обмене Windows-сообщениями между отладчиком и отлаживаемой программой. Если вам требуется строго синхронизированное взаимодействие, используйте Win32-объекты событий. Например, внедренный поток ожидает на некоем именованном событии, вызвав MsgWaitForMultipleObjectsEx, и пробуждается либо при поступлении в его очередь нового сообщения, либо при переходе события в незанятое состояние (signaled state) (детали см. в CGDITraceApp::InfiltratedThreadProc в файле GDITrace.cpp). В данной реализации событие просто сигнализирует потоку завершить свою работу и по сути не обеспечивает синхронизации.

UI-интерфейс GDIUsage позволяет получить список GDI-объектов, используемых другим процессом в некий момент, а затем сравнивать его с текущим состоянием этого процесса.

Другая ситуация, где может понадобиться действительно синхронизированное поведение, — перехват GDI-вызовов в момент загрузки DLL отлаживаемым приложением. Для перехвата GDI-вызовов отладчик должен как можно быстрее уведомлять об этом внедренный поток. Иначе отлаживаемый процесс может начать создавать GDI-объекты до того, как будут установлены перехватчики (interception stubs). Тогда было бы неплохо реализовать вот такой сценарий.

1. Поток отладчика узнает о загрузке DLL в отлаживаемый процесс, получив LOAD_DLL_DEBUG_EVENT, когда WaitForDebugEvent возвращает управление.
2. Переопределенный метод OnLoadDLLDebugEvent получает hModule, соответствующий DLL, сохраняет его в новой разделяемой переменной и требует от отлаживаемого процесса перехвата GDI-вызовов, переводя событие в незанятое состояние.
3. Во избежание рекурсивного вызова в случае загрузки отлаживаемым процессом еще одной DLL OnLoadDLLDebugEvent ожидает на другом событии, которое переводится в незанятое состояние, когда отлаживаемый процесс заканчивает перехват вызовов.
4. Поток, внедренный в отлаживаемый процесс, пробуждается, получая управление от MsgWaitForMultipleObjectsEx, так как одно из ожидаемых им событий переходит в незанятое состояние (освобождается).
5. Метод CGDITraceApp::OnNewDLL перенаправляет GDI-вызовы из DLL, загруженной по адресу в общей переменной, куда отладчик ранее записал hModule этой DLL.
6. Внедренный поток освобождает событие, на котором ждет отладчик.
7. Внедренный поток вызывает MsgWaitForMultipleObjectsEx в ожидании следующего запроса.
7. Выполнение потока отладчика возобновляется, так как освобождено событие, на котором он ждал.

Внимание! Два последних шага имеют одинаковые номера, так как выполняются в разных потоках. Предсказать, какой из этих потоков первым получит процессорное время, нельзя — соответствующее решение принимает планировщик ОС в зависимости от текущих условий.

Хотя данный сценарий выглядит превосходно, он приводит к взаимной блокировке потока отладчика и потока, внедренного в отлаживаемый процесс. Проблема возникает между шагами 3 и 4. Win32-функции отладки предполагают, что отладчик ждет событий от отлаживаемой программы с помощью WaitForDebugEvent. После возврата из этой функции все потоки в отлаживаемом процессе (даже не генерировавшие отладочное событие) приостанавливаются до вызова ContinueDebugEvent. Таким образом, поток отладчика остается на шаге 3, а шаг 4 никогда не будет выполнен отлаживаемой программой, поскольку в данном сценарии вызов ContinueDebugEvent предполагается по окончании синхронного взаимодействия. Так что запомните: какое-либо действие в отлаживаемой программе нельзя синхронизировать по отладочному событию, получаемому отладчиком.

В случае GDIUsage механизм на основе сообщений, по-видимому, обеспечивает достаточно быстрое уведомление. Проблема в другом. Внедренный поток запускается при получении отлаживаемым процессом первого сообщения, и к этому моменту все статически связанные DLL уже инициализированы. Если при этом одна из них создала GDI-объекты, GDIUsage этого не обнаружит. Значит, здесь нужен другой способ вызова кода, обеспечивающего перехват GDI-функций. GDIUsage позволяет обнаружить и определить место создания GDI-объектов, которые не освобождаются приложением в течение всего периода своего выполнения (кроме создаваемых в момент его запуска).

Отображение GDI-объектов

UI-интерфейс GDIUsage (**рис. 1**) позволяет получить список GDI-объектов, которые используются другим процессом в некий момент, а затем сравнивать его с текущим состоянием этого процесса. Список GDI-объектов хранится в экземпляре класса CGdiResources. Как уже говорилось при обсуждении TM_ GET_LIST, объект CGdiResources инициализируется вызовом CreateFromList после того, как список объектов

* Освобождать объекты, возвращаемые GetStockObject, не требуется, и их «утечка» невозможна. — *Прим. перев.*

передается из отлаживаемого процесса в отладчик через проецируемый в память файл.

В версии GDIUsage для Windows 9*x* класс CGdiResourcesDlg, отвечающий за отображение GDI-объектов, принимает указатель на объект CGdiResources как параметр. К сожалению, в версии программы для Windows XP класс CGdiResourcesDlg бесполезен в контексте процесса отладчика, так как при попытке использовать GDI-функции для отображения GDI-объекта, созданного в отлаживаемой программе, произойдет ошибка.

Решением данной проблемы является перемещение CGdiResourcesDlg и CGdiResources в GDITrace.dll, которая загружается в процесс отлаживаемой программы с помощью Windows-ловушки, как было показано ранее. Для передачи списка выделенных объектов обратно в отлаживаемый процесс применяется тот же механизм взаимодействия, но, прежде чем послать сообщение внедренному потоку, список нужно сохранить в контексте отладчика через проецируемый в память файл. Отладчик сериализует список CGdiResources (в зависимости от выбранной пользователем команды это либо текущее состояние, либо результат сравнения) в проекцию файла и посылает внедренному потоку сообщение TM_SHOW_LIST, параметром которого является описатель окна главного диалога GDIUsage. Эта операция выполняется CGDIDebugger::ShowList при сериализации и ShowRemoteList при асинхронной передаче сообщения.

Аналогично TM_GET_LIST сообщение TM_SHOW_LIST обрабатывается внедренным потоком, но направляется в CGDITraceApp::OnShowList. Этот метод инициализирует CGdiResources данными, сохраненными в проецируемом в память файле. Теперь диалог CGdiResourcesDlg может отобразить соответствующие GDI-объекты.

В отличие от сообщения TM_GET_LIST отладчику при посылке команды на отображение объектов не требуется в ответ кода возврата или иной информации. Однако в любом случае GDIUsage должен оставаться неактивным, пока пользователь не закроет диалог CGdiResourcesDlg. Вот почему описатель окна главного диалога GDIUsage передается в качестве родительского окна объекту CGdiResourcesDlg.

Реализация этого класса не изменилась по сравнению с версией для Windows 9*x*, если не считать двух усовершенствований. Во-первых, на основании сведений о формате GDI-описателя определяется, ссылается ли он на предопределенный (stock) объект, и, если да, добавляет к соответствующей строке звездочку (*). Но увидеть это в GDIUsage вам не удастся, так как API-функции, создающие предопределенные объекты, не перехватываются. Если хотите, сами напишите соответствующий код.*

Второе усовершенствование связано с необходимостью различать одинарный и двойной щелчок строки GDI-объекта в списке, чтобы отображать соответствующий стек вызовов (**рис. 7**). Чтобы избежать зависимости реализации CGdiResourcesDlg от реализации кода трассировки стека, был определен универсальный интерфейс обратного вызова INotificationCallBack. Класс CGDITraceApp является производным от него, реализует метод OnDoubleClick и регистрирует себя вызовом CGdiResourcesDlg::SetNotificationCallBack.

При двойном щелчке строки GDI-объекта CGdiResourcesDlg проверяет, зарегистрирован ли интерфейс обратного вызова. Если да, вызывается его метод DoubleClick, которому пе-

редается CGdiObj, связанный с выбранным объектом. CGDITraceApp извлекает из переданного CGdiObj стека вызовов и для его отображения создает экземпляр CCallStackDlg (детали см. в CGDITraceApp::OnDoubleClick в файле GDITrace.cpp).

Эта функциональность добавлена и в инструмент GDIndicator (**рис. 3**). В отличие от GDIUsage он использует для внедрения кода в другой процесс механизм на основе CreateRemoteThread, который я рассматривал в августовской статье. Интересно, что код сериализации и отображения совпадает с эквивалентным кодом GDIUsage — изменился лишь механизм удаленного взаимодействия.

Если использовать мои инструменты для просмотра объектов, выделенных программой NotePad, ее окно будет перерисовываться, и вы по-прежнему сможете выбирать команды из меню, но они не будут срабатывать, пока вы не закроете диалог.

А теперь — о двух подвохах. Поскольку диалог, отображающий GDI-объекты, выполняется в контексте другого процесса, происходят странные вещи. Во-первых, в Windows 2000 и Windows XP не так-то просто вывести на первый план окно из другого процесса. Чтобы это стало возможным, процесс — владелец текущего окна первого плана (foreground window) должен вызвать AllowSetForegroundWindow. Код диалогового окна выполняется как бесконечный цикл в контексте текущего потока другого процесса, пока пользователь не закроет диалог. В итоге, пока диалог не закроется, GDIndicator «висит». Во избежание такой неприятности его главное окно скрывается на все время работы диалога.

Со вторым подвохом справиться труднее. Окна, созданные в потоке, блокированном диалогом, получают не все направляемые им сообщения, так как фильтруются процедурой диалога. Например, если попробовать с помощью описанных в этой статье инструментов просмотреть список GDI-объектов, используемых программой NotePad, ее окно будет перерисовываться и вы по-прежнему сможете выбирать команды из меню, но они не будут срабатывать, пока вы не закроете диалог. В общем, применять описанные в статье инструменты непросто, но при выявлении утечек GDI-ресурсов они могут оказать неоценимую помощь.

Заключение

За время, разделяющее Windows 9*x* и Windows XP, в GDI многое изменилось. Хотя немалую часть моей программы составляет код, заимствованный из версии для Windows 9*x*, для получения списка GDI-объектов, выделенных другим процессом, применяются новые механизмы. Версия для Windows XP использует Win32-функции отладки, механизм внедрения DLL, Windows-ловушки, перехват API-функций и предоставляет дополнительные возможности по сравнению с версией для Windows 9*x*.

К поддержке отображения списка вызовов, приведших к созданию данного GDI-объекта, добавлен еще один штрих. По завершении удаленного процесса вызывается метод ExitInstance внедренной DLL. CGDITraceApp использует это уведомление, чтобы в последний раз получить список GDI-объектов, которые по-прежнему не освобождены. Аналогичный механизм для распознавания утечек памяти реализован в MFC на основе макроса DEBUG_NEW.

Описанные приемы можно применять для поиска утечек любых типов, если вы знаете список API-функций, используемых для создания интересующих вас ресурсов. Так, утечки объектов ядра позволяют выявить oh.exe и dh.exe из Platform SDK или ProcessExplorer (http://www.sysinternals.com). И хотя эти программы позволяют получить список объектов ядра, используемых любым процессом (а в ProcessExplorer есть и функция сравнения), дампы стека и завершающий дамп утечек, которые можно сгенерировать описанными здесь приемами, все же упрощают локализацию причин подобных проблем.

Кристоф Назар (Christophe Nasarre) — менеджер проектов в компании Business Objects (Франция). Автор ряда низкоуровневых утилит для Windows (еще со времен версии 3.0). С ним можно связаться по адресу cnasarre@montataire.net.

Основы и тонкости

Кен Спенсер

Автоматизированное создание Web-сервиса

Вопрос Стандарты, принятые в моей команде, требуют распределения бизнес-объектов между клиентским компьютером и сервером, поэтому мне нужен способ взаимодействия между объектами. Для этого подходит Microsoft .NET Remoting, но в силу многих причин мы применяем Web-сервис XML. Можно ли автоматизировать создание такого Web-сервиса?

Ответ Вы выбрали Web-сервисы, но тем, кого интересует дополнительная информация по выбору между Microsoft .NET Remoting и Web-сервисами, советую ознакомиться со статьей Тима Эвалда (Tim Ewald) в MSDN (http://msdn.microsoft.com/webservices/building/frameworkandstudio/default.aspx?pull=/library/en-us/dnbda/html/bdadotnetarch16.asp). А теперь отвечу на ваш вопрос.

Вы могли бы решить свою задачу, создав два Web-сервиса: первый создается с помощью отражения (reflection) и соответствует интерфейсу объекта, второй вызывает этот интерфейс. У меня на это ушло часа четыре, но в конце концов получилось.

За основу я взял код приложения, написанного для октябрьского номера за 2002 г. и работающего с пользовательскими атрибутами (http://msdn.microsoft.com/msdnmag/issues/02/10/AdvancedBasics/default.aspx). Рассмотрим этот код.

Его интерфейс — простая форма. ButtonClick для панели инструментов выглядит так:

```
Select Case ToolBar1.Buttons.IndexOf(e.Button)
   Case 0 : OpenFile()
   Case 1 : GenerateWebService()
   Case Else
End Select
```

Высокоуровневый метод GenerateWebService выполняет всю работу (**рис. 1**). Он вызывает функции, генерирующие код, и сохраняет их вывод в новых файлах. Остальная часть кода формы довольно проста и здесь не приводится (полный код можно скачать по ссылке http://msdn.microsoft.com/msdnmag/code03.aspx в разделе за январь).

Для загрузки выбранной сборки GenerateWebService вызывает метод LoadFrom, затем перебирает обнаруженные классы. Для каждого класса вызываются методы GenerateWSHeader и GenerateWSCode, которые и генерируют код, сохраняемый в ASMX- и VB-файлах. Кроме того, к текстовому полю добавляется имя класса. Функция WriteFile записывает как ASMX, так и VB-файлы.

В программах, создающих код, нет ничего особенного, но они требуют терпения. Дело в том, что нужно учитывать каждый символ и каждый пробел. Кроме того, нельзя забывать о концах строк, табуляторах и прочих средствах форматирования.

Теперь рассмотрим модуль GenRoutines — он содержит код, генерирующий функции (**рис. 2**). В таких программах нет ничего особенного, но они требуют терпения. Дело в том, что нужно учитывать каждый символ и каждый пробел. Кроме того, нельзя забывать о концах строк, табуляторах и прочих средствах форматирования, облегчающих восприятие кода. Необходимо также встроить обработку ошибок и прочие вспомогательные функции.

Взгляните на код, генерирующий Web-сервис. В первой строке модуля создается константа для URI Web-сервиса. Это позволяет легко подстраивать URI под стандарты, принятые в вашей организации.

Рис. 1. GenerateWebService

```
Sub GenerateWebService()
    Dim t As Type
    Dim othis As Object
    Dim oAssembly As [Assembly]
    Dim arrayOfTypes() As Type
    StatusBar1.Text = ""

    If sFileName = "" Then
        Exit Sub
    End If

    Try
        oAssembly = [Assembly].LoadFrom(sFileName)
        arrayOfTypes = oAssembly.GetExportedTypes()
        txtOutputWSHeader.Text = ""

        For Each t In arrayOfTypes
            If t Is Nothing Then
                MsgBox("Type was not found -- exiting")
                Exit Sub
            End If

            txtWSGenerated.Text &= t.Name & vbCrLf

           ' Записать ASMX-файл
            WriteFile(t.Name & ".asmx", GenerateWSHeader(t.Name))
           ' Записать файл кода
            WriteFile(t.Name & ".asmx.vb", GenerateWSCode(t))
            Next

    Catch exc As Exception
        StatusBar1.Text = "File " & sFileName & _
            "error occurred. " & exc.Message
    Finally
        oAssembly = Nothing
    End Try
End Sub
```

Функция GenerateWSHeader выводит заголовок Web-сервиса. Переданное ей имя класса становится именем Web-сервиса. Ее код несложен — это просто-напросто функция, обрабатывающая строки.

Теперь рассмотрим создание кода Web-сервиса (**рис. 2**). Именно тут выполняется настоящая работа. Сначала объявляются переменные, в том числе используемые в основном для обработки строк, но иногда хранящие информацию о методе. Например, для получения параметров метода или типа возвращаемого значения переменная oMI в цикле For Next ссылается на текущий метод. Точно так же в oCN хранится информация о конструкторе.

Рис. 2. Модуль GenRoutines

```
Imports System
Imports System.Reflection
Module GenRoutines

Private Const localURI As String = "http://mycouri.org/"

    Public Function GenerateWSHeader(ByVal Name As String) _
        As String
        Dim localWSHeader As String
        If Name = "" Then
            Return ""
        End If

        localWSHeader = "<%@ WebService Language=""vb"" "
        localWSHeader &= " Codebehind=""" & Name & _
            ".asmx.vb"" Class=""" & Name & """ %>"

        Return localWSHeader
    End Function

Public Function GenerateWSCode(ByVal t As Type) As String
    Dim oMI As MethodInfo
    Dim oPR As PropertyInfo
    Dim oCN As ConstructorInfo
    Dim bAtLeastOne As Boolean
    Dim localOutput As String
    Dim localTempString As String
    Dim oParam As ParameterInfo
    Dim localMethodCall As String
    Dim localObjectName As String
    Dim localAssemblyName As String
    Dim i As Integer

    localOutput = "Imports System.Web.Services" & vbCrLf
    localOutput &= "<WebService(Namespace:=""" & _
        localURI & """)> _" & vbCrLf

    localOutput &= "Public Class " & t.Name & vbCrLf
    localOutput &= vbTab & _
        "Inherits System.Web.Services.WebService" _
        & vbCrLf & vbCrLf

    localObjectName = "o" & t.Name

    localAssemblyName = t.AssemblyQualifiedName.ToString

    i = InStr(localAssemblyName, ",")

    localAssemblyName = Left(localAssemblyName, i - 1)

    localOutput &= vbTab & "Private " & _
        localObjectName & " as New " & _
        localAssemblyName & "()" & vbCrLf

    localOutput &= "'Generate Constructor " & vbCrLf
    localOutput &= "Public Sub New()" & vbCrLf
    localOutput &= "End Sub" & vbCrLf
    localOutput &= "End Sub" & vbCrLf
    localOutput &= vbCrLf

    ' Методы
    For Each oMI In t.GetMethods()
        If CheckName(oMI.Name) Then
            bAtLeastOne = False
            localOutput &= "<WebMethod()> _" & vbCrLf
            localOutput &= "Public Function " & _
                oMI.Name & "("
            localMethodCall = "Dim localReturn as " & _
                oMI.ReturnType.ToString & vbCrLf
            localMethodCall &= vbTab & "try" & vbCrLf
            localMethodCall &= vbTab & vbTab & _
                "localReturn = " & _
                localObjectName & "." & oMI.Name
            localMethodCall &= "("
            localTempString = ""

            For Each oParam In oMI.GetParameters
                If bAtLeastOne Then _
                    localTempString &= " _ " & _
                        vbCrLf & vbTab & ","
                    localTempString &= _
                        oParam.Name & " " & " as "
                    localTempString &= _
                        oParam.ParameterType.ToString
                    If bAtLeastOne Then _
                        localMethodCall &= ","
                    localMethodCall &= oParam.Name

                     If localTempString <> "" Then
                       bAtLeastOne = True
                       localOutput &= localTempString
                     End If
                Next

                localOutput &= ") as " & _
                         oMI.ReturnType.ToString & vbCrLf

                    localMethodCall &= ")" & vbCrLf

                    localMethodCall &= vbTab & _
                       "Catch Exc as Exception"  & vbCrLf
                    localMethodCall &= vbTab & vbTab & _
                        "Throw New Exception(""Error in "", _
                        Exc.InnerException)" & vbCrLf
                    localMethodCall &= vbTab & "End Try" & _
                        vbCrLf & vbCrLf

                    localMethodCall &= vbTab & "Return " & _
                        "localReturn" & vbCrLf

                    localOutput &= localMethodCall & vbCrLf

                    localOutput &= "End Function" & vbCrLf & _
                        vbCrLf
                End If
            Next

            localOutput &= vbCrLf & vbCrLf
            localOutput &= "End Class" & vbCrLf

            Return localOutput
    End Function

    Function CheckName(ByVal MethodName As String) As Boolean
        Select Case MethodName
            Case "GetHashCode" : Return False
            Case "ToString" : Return False
            Case "Equals" : Return False
            Case "GetType" : Return False
            Case Else
                Return True
        End Select
    End Function
End Module
```

Первые несколько строк начинают выводить код. Как видите, вы просто формируете операторы исходного кода и добавляете получаемые строки к переменной localOutput:

```
localOutput = "Imports System.Web.Services" & vbCrLf
localOutput &= "<WebService(Namespace:=""" & _
    localURI & """)> _" & vbCrLf
```

Для создания кода, генерирующего переменные и ссылки, можно установить собственные стандарты кодирования. Так, следующая строка создает впоследствии используемую в Web-сервисе переменную, имя которой соответствует ссылке на объект:

```
localObjectName = "o" & t.Name
```

Далее задается имя сборки и извлекается подстрока слева до первой точки:

```
localAssemblyName = t.AssemblyQualifiedName.ToString
i = InStr(localAssemblyName, ",")
localAssemblyName = Left(localAssemblyName, i - 1)
```

Это необходимо из-за того, что метод AssemblyQualifiedName возвращает в разделенной точками строке больше информации, чем мне нужно. Для формирования корректного имени просто извлекается часть строки слева до первой точки:

```
localOutput &= vbTab & "Private " & _
    localObjectName & " as New " & _
    localAssemblyName & "()" & vbCrLf
```

Мой код создает только конструктор по умолчанию без параметров. Если вам нужно добавить к нему дополнительный код, сделайте это сами.

Первый цикл For Each на каждой итерации создает параметры для каждого метода. Кроме того, код помечает имя метода атрибутом WebMethod. Чтобы выяснить, выводить ли этот метод, вызывается функция CheckName. Например, выводить ToString (метод класса по умолчанию) обычно не требуется. Взгляните, как переменная oMI используется для получения имени метода, его параметров и возвращаемого типа. Например, для вывода имени метода я делаю так:

```
localOutput &= "<WebMethod()> _" & vbCrLf
localOutput &= "Public Function " & _
    oMI.Name & "("
```

Код, генерируемый этим фрагментом, содержит переменную localReturn, в которой Web-методы хранят значения, которые возвращаются вызываемыми ими методами.

Внутренний цикл For Each (снова **рис. 2**) обрабатывает параметры. Он довольно хитрый, так как ему нужно учитывать все параметры, их типы и корректный синтаксис при наличии нескольких параметров.

Наконец, следующий код определяет тип возвращаемого методом параметра и добавляет его к генерируемой строке:

```
localOutput &= ") as " & _
  oMI.ReturnType.ToString  & vbCrLf
```

Далее создается обработчик ошибок Web-метода:

```
localMethodCall &= vbTab & "Catch Exc as Exception" & vbCrLf
localMethodCall &= vbTab & vbTab & "Throw New Exception( _
    ""Error in "", Exc.InnerException)" & vbCrLf
localMethodCall &= vbTab & "End Try" & vbCrLf & vbCrLf
```

Последние строки кода во внешнем цикле For Each завершают Web-метод, добавляя код, передающий возвращаемое значение, и оператор End Class.

При генерации кода Web-сервиса сначала объявляются переменные, в том числе используемые в основном для обработки строк, но иногда хранящие информацию о методе.

В текущей версии этой утилиты нужно изменить ссылку в ASMX-файле так, чтобы она указывала на пространство имен вашего проекта. Например, созданный утилитой атрибут Class надо изменить, добавив пространство имен для приложения, в котором он используется:

```
<%@ WebService Language="vb"  Codebehind="Order.asmx.vb"
    Class="WSTesterForGeneratedWS.Order" %>
```

Утилита готова, но давайте поразмышляем о процессе в целом. После того как код стал выглядеть прилично, я поместил его в проект Web-сервиса и проверил на ошибки. Исправив их, я вновь сгенерировал Web-сервис. Затем стал тестировать, исправлять ошибки и снова тестировать. Когда Web-сервис больше не давал ошибок в Visual Studio .NET, я запустил его в тестовом режиме и проверил каждый метод. Так я убедился, что все работает.

Есть еще один способ генерации кода — применение класса System.CodeDom из .NET Framework. Этот альтернативный подход я продемонстрирую в следующий раз. Кроме того, я включил простой пример использования CodeDom (CodeDomSample.vb) в файлы, которые можно скачать для этой статьи. Уже закончив статью, я выловил и устранил в коде несколько ошибок.

Вопросы и комментарии (на английском языке) присылайте по адресу basics@microsoft.com.

Кен Спенсер (Ken Spencer) работает в компании 32X Tech (http://www.32X.com), которая занимается разработкой ПО и обучением, а также предоставляет консалтинговые услуги по технологиям Microsoft.

Джеффри Рихтер

Безопасная синхронизация потоков

Синхронизацию потоков чаще всего используют для поддержки взаимоисключающего доступа нескольких потоков к разделяемому ресурсу. В Win32 API самый быстрый и эффективный способ синхронизации для взаимоисключающего доступа нескольких потоков, выполняемых в одном процессе, — применение структуры CRITICAL_SECTION и связанных с ней функций. В Microsoft .NET Framework такой структуры нет, но поддерживается аналогичный механизм для взаимоисключающего доступа нескольких потоков, работающих в одном процессе; данный механизм основан на структурах SyncBlock и классе System.Threading. Monitor.

.NET Framework поддерживает механизм, упрощающий синхронизацию потоков.

Сегодня речь пойдет о том, как в .NET Framework задействовать этот распространенный вид синхронизации потоков. Я объясню, почему структура SyncBlock и объект Monitor устроены именно так, а не иначе, расскажу, как они работают и почему их нельзя использовать привычным образом, без особых мер предосторожности.

Великая Идея

В .NET Framework применяется объектно-ориентированный подход. Это означает, что разработчики создают объекты и манипулируют ими, вызывая члены их типа. Время от времени к одному объекту обращается несколько потоков; чтобы предотвратить его повреждение, нужна синхронизация потоков. Проектируя .NET Framework, парни из Microsoft решили создать новый механизм, который упростил бы синхронизацию.

Основная идея такова: во-первых, с каждым объектом, память под который выделена из кучи, сопоставлена структура данных (очень похожая на CRITICAL_SECTION из Win32), пригодная для синхронизации потоков. Во-вторых, в библиотеке классов .NET Framework есть методы, которые, получив ссылку на объект, используют сопоставленную с ним структуру для синхронизации потоков.

В Win32 неуправляемый C++-класс, поддерживающий такой механизм, выглядел бы по аналогии с тем, что показано на **рис. 1**. В сущности, .NET Framework наделяет каждый объект полем, похожим на CRITICAL_SECTION, и берет на себя его инициализацию и удаление. Разработчику остается лишь написать код, который входит в это поле и покидает его по мере необходимости, а затем добавить во все методы, требующие синхронизации потоков.

Реализация Великой Идеи

Теперь ясно, что сопоставлять поле CRITICAL_SECTION с каждым объектом в куче весьма расточительно, особенно если синхронизация доступа к большинству объектов никогда не понадобится. Поэтому разработчики .NET Framework ввели более эффективную поддержку описанной выше функциональности.

При инициализации общеязыковая исполняющая среда (CLR) создает кэш структур SyncBlock. Структура SyncBlock — это блок памяти, который по мере необходимости можно связывать с любыми объектами. У SyncBlock те же поля, что и у Win32-структуры CRITICAL_SECTION.

С каждым новым объектом, созданным в куче, связываются два дополнительных поля. Первое — MethodTablePointer — содержит адрес таблицы методов типа. По сути, этот указатель позволяет получить сведения о типе любого из объектов

Рис. 1. Структура CRITICAL_SECTION для каждого объекта

```
class SomeType {
private:
   // Закрытое поле CRITICAL_SECTION,
   // связанное с каждым объектом
   CRITICAL_SECTION m_csObject;

public:
   SomeType() {
      // Конструктор объекта инициализирует его поле
      // CRITICAL_SECTION
      InitializeCriticalSection(&m_csObject);
   }

   ~SomeType() {
      // Деструктор объекта удаляет его поле CRITICAL_SECTION
      DeleteCriticalSection(&m_csObject);
   }

   void SomeMethod() {
      // Методы используют принадлежащее объекту
      // поле CRITICAL_SECTION для синхронизации
      // доступа нескольких потоков к этому объекту
      EnterCriticalSection(&m_csObject);

      // Здесь размещается код, требующий синхронизации...
      LeaveCriticalSection(&m_csObject);
   }

   void AnotherMethod() {
      // Методы используют принадлежащее объекту
      // поле CRITICAL_SECTION для синхронизации
      // доступа нескольких потоков к этому объекту
      EnterCriticalSection(&m_csObject);

      // Здесь размещается код, требующий синхронизации...
      LeaveCriticalSection(&m_csObject);
   }
};
```

в куче. Кстати, когда вы вызываете метод GetType из класса System.Object, он определяет тип объекта, следуя по указателю из поля MethodTablePointer. Второе дополнительное поле, SyncBlockIndex, содержит индекс (в виде 32-разрядного целого значения со знаком) структуры SyncBlock в кэше этих структур.

При создании объекта его поле SyncBlockIndex инициализируется отрицательным значением, указывающим, что объект не ссылается ни на одну из структур SyncBlock. Если в дальнейшем будет вызван метод, синхронизирующий доступ к этому объекту, CLR найдет в кэше свободную структуру SyncBlock и поместит указатель на нее в поле SyncBlockIndex. Другими словами, структуры SyncBlock «на лету» связываются с объектами, нуждающимися в синхронизирующих полях. Если потоков, требующих синхронизации доступа к этому объекту, больше нет, значение поля SyncBlockIndex сбрасывается (путем записи в него отрицательного числа), а структура SyncBlock освобождается, после чего она может быть связана с другим объектом.

Принцип работы этого механизма иллюстрирует схема на **рис. 2**. В секции «Структуры данных CLR» на этой схеме видно, что с каждым из известных системе типов связана своя структура данных; кроме того, имеется набор структур SyncBlock. В секции «Управляемая куча» изображены три объекта: A, B и C. У каждого из них поле MethodTablePointer ссылается на таблицу методов соответствующего типа. Пользуясь таблицей методов, можно узнать тип любого объекта. Поэтому нетрудно определить, что объекты A и B — экземпляры типа SomeType, а объект C — экземпляр типа AnotherType.

Заметьте, что поле SyncBlockIndex объекта A установлено в 0. Это говорит о том, что структура SyncBlock #0 в настоящее время используется объектом A. В аналогичном поле объекта B содержится –1, следовательно, с этим объектом не связано ни одной структуры SyncBlock. У объекта C в поле SyncBlockIndex записано значение 2, значит, этот объект использует SyncBlock #2. В этом примере структура SyncBlock #1 не задействована и в дальнейшем может быть связана с каким-нибудь объектом.

Как видите, с каждым объектом в куче логически сопоставлена структура SyncBlock, позволяющая быстро получить монопольный доступ к этому объекту. Однако физически структура SyncBlock связывается с объектом, только когда она реально требуется, а как только эта структура становится ненужной, связь тут же разрывается. Это позволяет эффективнее использовать память. Кстати, при необходимости в кэше создаются дополнительные структуры SyncBlock, так что можно не беспокоиться, что они вдруг закончатся из-за одновременной синхронизации множества объектов.

Операции со структурой SyncBlock через Monitor

Разобравшись в инфраструктуре SyncBlock, рассмотрим блокировку объектов. Методы класса System.Threading.Monitor (все они статические) позволяют блокировать или разблокировать объект. Чтобы блокировать объект, вызовите следующий метод:

```
public static void Enter(object obj);
```

Метод Enter сначала проверяет, не отрицательное ли значение содержится в поле SyncBlockIndex указанного объекта, и, если это так, находит свободную структуру SyncBlock и записывает в SyncBlockIndex ее индекс. Связав SyncBlock с

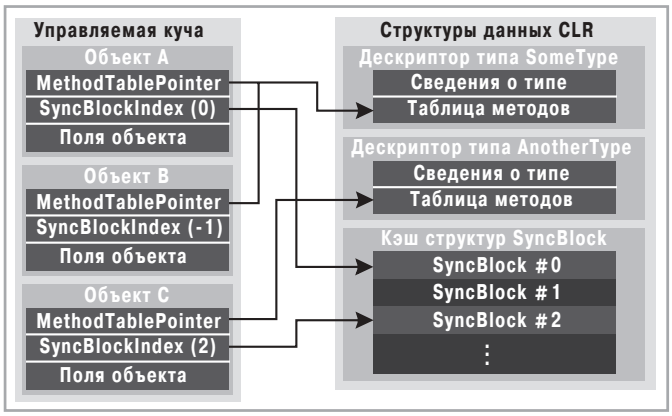


Рис. 2. Синхронизация потоков в .NET

объектом, метод проверяет, не принадлежит ли только что связанная структура другому потоку. Если в данный момент эта структура свободна, вызывающий поток становится владельцем связанного с ней объекта. Если же при вызове метода Enter оказалось, что структура SyncBlock принадлежит другому потоку, вызывающий поток приостанавливается, пока текущий владелец SyncBlock не откажется от нее.

Если вы предпочитаете подстраховаться, вызывайте вместо Enter одну из версий метода TryEnter:

```
public static Boolean TryEnter(object obj);

public static Boolean TryEnter(object obj,
    int millisecondsTimeout);

public static Boolean TryEnter(object obj, TimeSpan timeout);
```

Первая версия просто проверяет, может ли вызывающий поток захватить структуру SyncBlock, и, если да, возвращает true. Остальные версии позволяют задавать время ожидания вызывающим потоком освобождения структуры SyncBlock. Все эти методы возвращают false, если вызывающий поток не может стать владельцем структуры SyncBlock.

Завладев структурой SyncBlock, код может безопасно обращаться к полям связанного с ней объекта. Закончив работу с объектом, поток должен освободить SyncBlock вызовом метода Exit:

```
public static void Exit(object obj);
```

Если связанная с заданным объектом структура SyncBlock не принадлежит вызывающему потоку, метод Exit генерирует исключение SynchronizationLockException. Заметьте, что поток может рекурсивно захватывать SyncBlock; эта структура считается свободной, если каждому успешному вызову Enter (либо TryEnter) соответствует вызов Exit.

Синхронизация по методике Microsoft

Взгляните на **рис. 3**: показанный там код демонстрирует, как блокировать или разблокировать объект методами Enter и Exit объекта Monitor. Обратите внимание, что реализация этого свойства требует вызовов Enter и Exit, а также наличия временной переменной (dt). Это очень важно в ситуации, когда поток вызывает метод PerformTransaction в момент обращения другого потока к тому же свойству.

C#-оператор lock упрощает код

Поскольку такие ситуации, в которых приходится сначала вызвать Enter, потом обратиться к защищенному ресурсу и,

наконец, вызвать Exit, встречаются очень часто, C# поддерживает особый синтаксис для упрощения соответствующего кода. Хотя два фрагмента C#-кода, показанных на **рис. 4**, функционально одинаковы, второй из них гораздо проще благодаря C#-оператору lock. Взгляните для примера на новую, улучшенную версию свойства LastTransaction (**рис. 5**): теперь временная переменная не нужна.

Рис. 3. Применение методов Enter и Exit

```
class Transaction {

   // Закрытое поле, хранящее время
   // выполнения последней транзакции
   private DateTime timeOfLastTransaction;

   public void PerformTransaction() {
      // Блокировать этот объект
      Monitor.Enter(this);

      // Выполнить транзакцию...

      // Записать время последней транзакции
      timeOfLastTransaction = DateTime.Now;

      // Разблокировать этот объект
      Monitor.Exit(this);
   }

   // Открытое свойство только для чтения, возвращающее
   // время последней транзакции
   public DateTime LastTransaction {
      get {
         // Блокировать этот объект
         Monitor.Enter(this);

         // Сохранить время последней транзакции
         // во временной переменной
         DateTime dt = timeOfLastTransaction;

         // Разблокировать этот объект
         Monitor.Exit(this);

         // Вернуть значение из временной переменной
         return(dt);
      }
   }
}
```

Рис. 4. Обычный и упрощенный способы блокировки/разблокировки

```
// Обычная функция
public void SomeMethod() {
   // Блокировать этот объект
   Object oTemp = this;
   Monitor.Enter(oTemp);

   try {
          // Обратиться к объекту
          ...
          // Разблокировать объект
   }

   finally {
     Monitor.Exit(oTemp);
   }

   // Вернуть управление
}

// Упрощенная функция
public void SomeMethod() {
   // Блокировать этот объект
   lock (this) {

      // Обратиться к объекту
      ...
      // Разблокировать объект
   }

   // Вернуть управление
}
```

Оператор lock не только упрощает код, но и гарантирует вызов метода Monitor.Exit, освобождающий структуру SyncBlock, — даже при возникновении исключения в блоке try. С синхронизирующими механизмами всегда следует применять обработку исключений, чтобы гарантировать корректное освобождение блокировок. При использовании оператора lock компилятор создаст соответствующий код автоматически. Кстати, в Visual Basic .NET есть оператор SyncLock, который делает то же, что и оператор lock в C#.

Синхронизация статических членов по методике Microsoft

Класс Transaction демонстрирует, как синхронизировать доступ к полям объекта. А как быть, если в типе определено несколько статических полей и статических методов, обращающихся к этим полям? В этом случае экземпляр типа не находится в куче, а значит, нет ни связанной структуры SyncBlock, ни ссылки на объект, которую можно было бы передать методам Enter и Exit класса Monitor.

Но оказывается, блок памяти, содержащий дескриптор типа, сам является объектом из кучи. Хотя это не отражено на **рис. 2**, в действительности блоки памяти с дескрипторами типов SomeType и AnotherType — тоже объекты и, следовательно, обладают полями MethodTablePointer и SyncBlockIndex. Это означает, что с типом можно связать структуру SyncBlock, а ссылку на объект типа — передать методам Enter и Exit класса Monitor. В версии класса Transaction, показанной на **рис. 6**, все его члены сделаны статическими, а метод PerformTransaction и свойство LastTransaction изменены так, чтобы продемонстрировать рекомендуемый Microsoft способ синхронизации доступа к статическим членам.

В коде его метода и свойства нет ключевого слова this, так как оно неприменимо в ссылках на статические члены. Вме-

Рис. 5. Упрощенный класс Transaction

```
class Transaction {

   // Закрытое поле, хранящее время
   // выполнения последней транзакции
   private DateTime timeOfLastTransaction;

   public void PerformTransaction() {
      lock (this) {
         // Выполнить транзакцию...

         // Записать время последней транзакции
         timeOfLastTransaction = DateTime.Now;
      }
   }

   // Открытое свойство только для чтения, возвращающее
   // время последней транзакции
   public DateTime LastTransaction {
      get {
         lock (this) {
            // Вернуть время последней транзакции
            return timeOfLastTransaction;
         }
      }
   }
}
```

сто этого я передаю оператору lock ссылку на объект дескриптора типа, полученную C#-оператором typeof (он возвращает ссылку на дескриптор указанного типа). В Visual Basic .NET аналогичная функциональность поддерживается оператором GetType.

Почему Великая Идея оказалась не столь великой

Как видите, идея насчет того, чтобы логически сопоставить синхронизирующие структуры со всеми объектами в куче, теоретически выглядит неплохо, но на практике это просто ужас! Позвольте объяснить почему. Помните фрагмент неуправляемого кода на C++, показанный в начале статьи? Доведись вам писать самостоятельно такой код, разве вы сделаете поле CRITICAL_SECTION открытым? Конечно, нет — это было бы просто смешно! Сделав его открытым, вы дадите возможность прикладному коду манипулировать структурой CRITICAL_SECTION, и тогда любой вредоносный код сможет элементарно заблокировать любой поток, использующий экземпляры соответствующего типа.

Догадались, в чем дело? SyncBlock — это не что иное, как открытая синхронизирующая структура, которая есть у любого объекта в куче! Любая программа в любой момент может передать методам Enter и Exit класса Monitor ссылку на какой угодно кусок кода. Более того, этим методам можно передать ссылку на дескриптор любого типа.

Код на **рис. 7** демонстрирует, сколь ужасна может быть подобная ситуация. Здесь Main создает объект App и входит в его блокировку. В один прекрасный момент начинается сбор мусора (в моем примере он выполняется принудительно). Вызванный при этом метод Finalize объекта App пытается блокировать объект. Но специализированный CLR-поток Finalize не в состоянии получить блокировку, поскольку она принадлежит первичному потоку приложения. В итоге специализированный CLR-поток останавливается, и больше ни один принадлежащий процессу (а в нем может быть несколько AppDomain'ов) объект не удастся подготовить к уничтожению, равно как не получится и освободить память в куче, занятую объектами, которые требуют подготовки к уничтожению!

Рис. 6. Новая версия класса Transaction

```
class Transaction {

   // Закрытое поле, хранящее время
   // выполнения последней транзакции
   private static DateTime timeOfLastTransaction;

   public static void PerformTransaction() {
      lock (typeof(Transaction)) {
         // Выполнить транзакцию...

         // Записать время последней транзакции
         timeOfLastTransaction = DateTime.Now;
      }
   }

   // Открытое свойство только для чтения, возвращающее
   // время последней транзакции
   public static DateTime LastTransaction {
      get {
         lock (typeof(Transaction)) {
            // Вернуть время последней транзакции
            return timeOfLastTransaction;
         }
      }
   }
}
```

К счастью, эту проблему можно решить, но для этого придется игнорировать инфраструктуру и рекомендации Microsoft. Определите в типе закрытое поле System.Object, создайте объект и передавайте оператору lock (в C#) или SyncLock (в Visual Basic .NET) ссылку на закрытый объект. На **рис. 8** показано, как переписать класс Transaction, чтобы синхронизирующий

Рис. 7. Потоки, столкнувшиеся лоб в лоб

```
using System;
using System.Threading;

class App {
   static void Main() {
      // Создать экземпляр объекта App
      App a = new App();

      // Этот вредоносный код входит в блокировку
      // на объекте и не покидает ее
      Monitor.Enter(a);

      // Для демонстрации сделаем объект недостижимым
      // и заставим CLR выполнить сбор мусора
      a = null;
      GC.Collect();

      // Заставим CLR ждать завершения всех методов Finalize.
      // Вот и все - потоки заблокировали друг друга!
      GC.WaitForPendingFinalizers();

      // Сюда мы никогда не попадем!
      Console.WriteLine("Leaving Main");
   }

   // Это метод Finalize в типе App
   ~App() {
      // Ради демонстрации заставим специализированный CLR-поток
      // блокировать объект. Поскольку блокировкой владеет поток
      // Main, он и CLR-поток взаимно блокируются!
      lock (this) {
         // Притворяемся, будто мы здесь что-то делаем...
      }
   }
}
```

Рис. 8. Класс Transaction с закрытым полем Object

```
class Transaction {

   // Закрытое поле Object, используемое
   // исключительно для синхронизации
   private Object objLock = new Object();

   // Закрытое поле, хранящее время
   // выполнения последней транзакции
   private DateTime timeOfLastTransaction;

   public void PerformTransaction() {
      lock (objLock) {
         // Выполнить транзакцию...

         // Записать время выполнения последней транзакции
         timeOfLastTransaction = DateTime.Now;
      }
   }

   // Закрытое свойство только для чтения, возвращающее
   // время последней транзакции
   public DateTime LastTransaction {
      get {
         lock (objLock) {
            // Вернуть время последней транзакции
            return timeOfLastTransaction;
         }
      }
   }
}
```

объект был доступен лишь своему классу, а **рис. 9** иллюстрирует, как сделать то же самое для класса Transaction, у которого все члены статические.

Как-то странно создавать объект System.Object только для синхронизации с использованием класса Monitor. Чувствую, что Microsoft неверно спроектировала класс Monitor. По идее, для каждого из типов, требующих синхронизации, следует создавать свой экземпляр Monitor. Статические методы должны быть методами экземпляра, которым не нужен параметр System.Object. Это решило бы все проблемы и существенно упростило бы модель программирования для разработчиков.

Кстати, если вы создаете сложный тип с множеством полей, то, вероятно, вашим методам и свойствам понадобится единовременно блокировать лишь часть полей объекта. Вы всегда можете блокировать некое поле, передав его объект оператору lock или методу Monitor.Enter. Конечно, это допустимо, только если поля закрытые (советую всегда делать их таковыми). Чтобы заблокировать сразу несколько полей, можно по очереди передать их оператору lock или методу Enter, как обычно. Альтернативный способ — создать объект System.Object, предназначенный только для блокировки специфического набора полей. Чем меньше наборы полей, которыми вы оперируете при блокировке, тем выше производительность и масштабируемость вашего кода.

Неупакованные экземпляры типов значений

В завершение хочу обратить ваше внимание на ошибку в синхронизации, «выслеживание» которой отняло у меня несколько часов, когда я столкнулся с ней впервые. Вот фрагмент кода, демонстрирующий эту ошибку:

```
class AnotherType {

   // Неупакованный (unboxed) тип значения (Boolean)
   private Boolean flag = false;

   public Boolean Flag {
      set {
         Monitor.Enter(flag); // упаковывает флаг
                              // и блокирует объект
         flag = value;        // реальное значение не защищено
         Monitor.Exit(flag);  // упаковывает флаг и пытается
                              // разблокировать объект
      }
   }
}
```

Вероятно, вы удивитесь, узнав, что в этом коде вообще нет синхронизации! Причина в том, что флаг — это неупакованный тип значения (unboxed value type), а не ссылочный тип (reference type). У экземпляров неупакованных типов значений нет дополнительных полей MethodTablePointer и SyncBlockIndex. Это означает, что с экземпляром неупакованного типа значения нельзя связать структуру SyncBlock.

Методы Enter и Exit класса Monitor требуют ссылки на объект, память под который выделена из кучи. Обнаружив код, который пытается передать неупакованный тип значения методу, требующему ссылки на объект, компиляторы C#, Visual Basic .NET и многие другие автоматически генерируют код, упаковывающий экземпляр типа значения. В упакованном экземпляре есть поля MethodTablePointer и SyncBlockIndex, поэтому его можно использовать для синхрони-

Рис. 9. Класс Transaction со статическими членами

```
class Transaction {

   // Закрытое статическое поле Object, используемое
   // исключительно для синхронизации
   private static Object objLock = new Object();

   // Закрытое поле, хранящее время
   // выполнения последней транзакции
   private static DateTime timeOfLastTransaction;

   public static void PerformTransaction() {
      lock (objLock) {
         // Выполнить транзакцию...

         // Записать время выполнения последней транзакции
         timeOfLastTransaction = DateTime.Now;
      }
   }

   // Закрытое свойство только для чтения, возвращающее
   // время последней транзакции
   public static DateTime LastTransaction {
      get {
         lock (objLock) {
            // Вернуть время последней транзакции
            return timeOfLastTransaction;
         }
      }
   }
}
```

зации потоков. Однако при каждом вызове синхронизирующего метода создается новый упакованный экземпляр типа значения, поэтому блокировка снимается вовсе не с того объекта, который был блокирован.

Так, если в предыдущем фрагменте кода вызвать метод-аксессор set свойства Flag, он в свою очередь вызовет метод Enter класса Monitor. Этот метод требует ссылочного типа, поэтому флаг упаковывается, и методу Enter передается указатель на упакованную версию флага. Теперь структура SyncBlock, связанная с упакованным объектом, принадлежит вызывающему потоку. Если другой поток попытается обратиться сейчас к этому свойству, флаг будет упакован еще раз, в результате получится новый объект со своей структурой SyncBlock. Кроме того, при вызове метода Exit передаваемый ему тип значения снова упаковывается.

Как я сказал, мне пришлось потратить несколько часов на поиск причины этой ошибки. Если вы хотите синхронизировать доступ к неупакованному экземпляру типа значения, создайте объект System.Object и используйте для синхронизации именно его (**рис. 10**).

Рис. 10. Теперь синхронизация есть

```
class AnotherType {

   // Неупакованный тип значения (Boolean)
   private Boolean flag = false;

   // Закрытое поле Object, используемое для
   // синхронизации доступа к флагу
   private Object flagLock = new Object();

   public Boolean Flag {
      set {
         Monitor.Enter(flagLock);
         flag = value;
         Monitor.Exit(flagLock);
      }
   }
}
```

Между прочим, если не вызывать методы Enter и Exit объекта Monitor напрямую, а использовать оператор lock, компилятор C# защитит вас от случайных попыток блокировать тип значения. При передаче оператору lock неупакованного экземпляра типа значения компилятор C# генерирует ошибку. Например, если передать оператору lock значение типа Boolean (bool в C#), компилятор выдаст сообщение «error CS0185: 'bool' is not a reference type as required by the lock statement». А компилятор Visual Basic .NET в аналогичной ситуации выдаст сообщение «error BC30582: 'SyncLock' operand cannot be of type 'Boolean' because 'Boolean' is not a reference type».

Вопросы и комментарии (на английском языке) присылайте по адресу dot-net@microsoft.com.

Джеффри Рихтер (Jeffrey Richter) – один из соучредителей компании Wintellect (http://www.wintellect.com), которая специализируется на обучении и консалтинге в области технологий .NET и Windows. Автор ряда книг по программированию, в том числе «Applied Microsoft .NET Framework Programming» (Microsoft Press, 2002)*.

* Эта книга переведена и выпущена издательством «Русская Редакция» под названием «Программирование на платформе Microsoft .NET Framework». – *Прим. перев.*

Basic и .NET

Тэд Пэттисон

Реализация обратных вызовов с помощью группового делегата

Эта статья продолжает декабрьскую за прошлый год, в которой я познакомил вас с базовыми концепциями и приемами программирования, связанными с делегатами (http://msdn.microsoft.com/msdnmag/issues/02/12/basicinstincts/default.aspx). Я исхожу из того, что вы прочитали предыдущую статью и уже знаете основы программирования делегатов. В частности, вы должны уметь определять тип-делегат (delegate type), создавать объект-делегат (delegate object), связанный с методом-обработчиком (handler method), и запускать этот метод вызовом метода Invoke объекта-делегата.

Архитектура на основе делегатов обеспечивает полиморфизм так же, как и архитектура на основе интерфейсов. Кроме того, делегаты предоставляют встроенную поддержку для операций с несколькими методами-обработчиками за счет группового связывания.

Сегодня я создам прикладную архитектуру на основе делегатов, использующую уведомления об обратных вызовах (callback notifications), и объясню, как делегаты поддерживают связывание источника уведомлений с несколькими методами-обработчиками через механизм группового связывания (multicasting).

Реализация обратных вызовов с помощью делегатов

Вообразите, что вы проектируете приложение с неким классом BankAccount, содержащим метод Withdraw. Допустим, вы хотите, чтобы другая часть этого приложения как-то реагировала на любую попытку снятия через объект BankAccount суммы, превышающей 5000 долларов. Отправной точкой можно считать такую схему:

```
Class BankAccount
  Sub Withdraw(ByVal Amount As Decimal)
    If (Amount > 5000) Then
      '*** Послать уведомление заинтересованным сторонам
    End If
    '*** Провести снятие денег со счета
  End Sub
End Class
```

В этом примере объект BankAccount будет действовать как источник уведомлений. Следовательно, он должен предоставлять какой-то способ, который позволил бы слушателю сообщать о своей заинтересованности в получении уведомлений. Иначе говоря, объект BankAccount должен поддерживать регистрацию слушателем своего метода-обработчика для обратных вызовов. Начнем с определения нового типа-делегата LargeWithdrawHandler:

```
Delegate Sub LargeWithdrawHandler(ByVal Amount As Decimal)
```

Теперь модифицируем класс BankAccount, чтобы он выступал в роли источника уведомлений. Для этого в него следует добавить два члена: закрытое поле (private field) с именем handler, тип которого совпадает с типом-делегатом LargeWithdrawHandler, и метод RegisterHandler, позволяющий другому коду регистрировать объект-делегат для приема уведомлений об обратных вызовах:

```
Class BankAccount
  Private handler As LargeWithdrawHandler
  Sub RegisterHandler(ByVal handler As LargeWithdrawHandler)
    Me.handler = handler
  End Sub
  '*** Остальные члены класса опущены
End Class
```

Как видите, метод RegisterHandler принимает единственный параметр типа LargeWithdrawHandler. Реализация RegisterHandler присваивает его значение полю handler, благодаря чему объект BankAccount увидит объект-делегат и выполнит его метод-обработчик.

После вызова метода RegisterHandler с указанием объекта-делегата любой метод в классе BankAccount сможет запустить метод-обработчик вызовом Invoke применительно к зарегистрированному объекту-делегату. Вот как вызвать Invoke из метода Withdraw класса BankAccount:

```
Sub Withdraw(ByVal Amount As Decimal)
  '*** Посылать уведомления, если это требуется
  If (Amount > 5000) AndAlso (Not handler Is Nothing) Then
    handler.Invoke(Amount)
  End If
  '*** Провести снятие денег со счета
End Sub
```

Метод Withdraw проверяет, содержит ли поле handler допустимую ссылку или оно равно Nothing. Это поле равно Nothing, пока не вызван RegisterHandler, и ваш код не должен пытаться запускать метод Invoke в отсутствие реальной ссылки.

Класс BankAccount написан так, что посылает уведомление всякий раз, когда снимаемая со счета сумма превышает 5000 долларов. Теперь создадим метод-обработчик, способный реагировать на эти уведомления. Взгляните на следующий класс:

```
Class AccountHandlers
  Shared Sub GetApproval(ByVal Amount As Decimal)
```

```
    '*** Блокировать операцию снятия денег со счета
    '*** до тех пор, пока ее не одобрит менеджер
  End Sub
  Shared Sub LogWithdrawToDB(ByVal Amount As Decimal)
    '*** Записать информацию о снятой сумме в базу данных
  End Sub
  Shared Sub LogWithdrawToFile(ByVal Amount As Decimal)
    '*** Записать информацию о снятой сумме в файл журнала
  End Sub
End Class
```

Должные сигнатуры методов GetApproval, LogWithdrawToDB и LogWithdrawToFile позволяют использовать их в качестве методов-обработчиков уведомлений, посылаемых объектом BankAccount.

А сейчас напишем простое приложение, которое увязывает все это воедино. Во-первых, оно должно создавать объект BankAccount. Потом объект-делегат, связанный с целевым методом-обработчиком, например с GetApproval. И, наконец, вызывать метод RegisterHandler, передавая ему ссылку на объект-делегат.

```
Module MyApp

  Sub Main()
    '*** Создать объект BankAccount
    Dim acc1 As New BankAccount()
    '*** Создать объект-делегат
    '*** и зарегистрировать метод обратного вызова
    acc1.RegisterHandler(AddressOf AccountHandlers.GetApproval)
    '*** Сделать что-нибудь, что приведет
    '*** к инициации обратного вызова
    acc1.Withdraw(5001)
  End Sub
End Module
```

Здесь оператор New при создании объекта-делегата не применяется. Вместо этого используется более удобный сокращенный синтаксис. Поскольку метод RegisterHandler ожидает параметр с типом LargeWithdrawHandler, вы можете просто указать оператор AddressOf с именем метода, как и было показано в этом фрагменте кода.

На данном этапе я получаю приложение, уведомляющее об обратном вызове с помощью делегата. Когда метод Main вызывает метод Withdraw объекта BankAccount и передает параметр со значением 5001, реализация Withdraw обращается к методу GetApproval через ссылку на объект-делегат, хранящуюся в поле handler.

Заметьте, что это приложение — хороший пример архитектуры с нежесткими связями (loosely coupled design). Класс BankAccount не знает и знать не хочет, какой тип метода-обработчика вы используете. GetApproval легко заменить методом-обработчиком LogWithdrawToDB или LogWithdrawToFile. Столь же нетрудно создать другой метод в другом классе и задействовать его в качестве метода обратного вызова. Отсюда можно заключить, что архитектура на основе делегатов обеспечивает полиморфизм не хуже архитектуры на основе интерфейсов.

Однако один важный вопрос еще не решен. Текущая реализация BankAccount поддерживает обратные вызовы только одного метода-обработчика. К счастью, делегаты предоставляют встроенную поддержку для операций с несколькими методами-обработчиками за счет группового связывания.

Групповое связывание

Как я уже говорил, каждый тип-делегат поддерживает манипуляции с несколькими методами-обработчиками. Такую поддержку типы-делегаты получают за счет наследования от класса MulticastDelegate, определенного в пространстве имен System. Преимущество группового связывания в том, что оно позволяет объединить несколько методов-обработчиков в список и сопоставить его с единственным объектом-делегатом. Когда вы вызовете Invoke применительно к этому объекту-делегату, код класса MulticastDelegate поочередно выполнит каждый метод-обработчик в списке.

Чтобы вы поняли, как все это работает, допустим, что вы хотели бы связать два совершенно разных метода-обработчика с одним объектом-делегатом. Это можно сделать вызовом общего (shared) метода Combine класса System.Delegate. Вы передаете ему два объекта-делегата и на выходе получаете новый объект — групповой делегат (групповой объект-делегат). Вот пример:

```
'*** Создать два индивидуальных делегата
Dim handler1, handler2 As LargeWithdrawHandler
handler1 = AddressOf AccountHandlers.GetApproval
handler2 = AddressOf AccountHandlers.LogWithdrawToDB

'*** Объединить их в групповой делегат
Dim result As [Delegate] = [Delegate].Combine(handler1, handler2)

'*** Преобразовать ссылку в тип LargeWithdrawHandler
Dim handlers As LargeWithdrawHandler
handlers = CType(result, LargeWithdrawHandler)

'*** Выполнить методы-обработчики в списке
handlers.Invoke(5001)
```

Обратите внимание на квадратные скобки вокруг имени класса — [Delegate]. Они нужны из-за того, что Delegate — еще и ключевое слово. Квадратные скобки являются своего рода escape-символами, которые сообщают компилятору Visual Basic .NET, что вы используете Delegate как имя класса.

Вызов Combine возвращает ссылку на только что созданный объект — групповой делегат. Заметьте, что тип, возвращаемый методом Combine, относится к универсальному Delegate, который преобразуется в более специфичный тип-делегат (здесь — в LargeWithdrawHandler). Это нужно для вызова метода Invoke.

Теперь обсудим, как реализованы групповые делегаты. Такой делегат — не более чем связанный список (linked list) объектов-делегатов. Закрытая реализация каждого объекта-делегата содержит поле, предназначенное для хранения ссылки на предыдущий объект-делегат в списке. Возможно, вам показалось, что было бы гораздо понятнее, если бы делегат хранил ссылку на следующий, а не на предыдущий делегат в списке. Все дело в порядке, в котором выполняются методы-обработчики. Но подробнее об этом я расскажу чуть позже.

Делегат в начале списка содержит ссылку на предыдущий делегат, а делегат в конце списка — значение Nothing. Поэтому позиция объекта-делегата в списке очень важна.

Когда вы вызываете Combine, он связывает два или более объекта-делегата и возвращает ссылку на объект-делегат, находящийся в начале списка. Обратите внимание на то, что в последнем примере вызывалась перегруженная реализация Combine, принимающая два параметра. Эта реализация

Combine создает групповой список и помещает в его начало объект-делегат, переданный во втором параметре. Таким образом, в данном случае делегат в начале списка связывается с методом LogWithdrawToDB. Закрытое поле этого объекта-делегата содержит ссылку на предыдущий объект-делегат, связанный с методом GetApproval.

Есть и другая перегруженная версия Combine, которая принимает массив объектов-делегатов. Тогда первый делегат из массива помещается в позицию 0 — в начало списка. Вы можете вызвать любую перегруженную версию Combine.

Обращайте внимание на то, куда помещается каждый объект-делегат в списке, так как это позволяет контролировать, какой метод-обработчик будет выполняться первым.

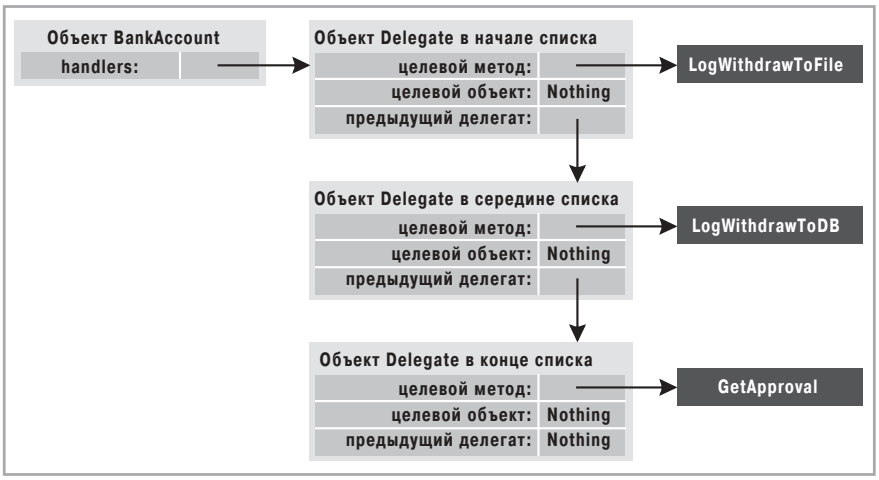


Рис. 2. Групповые делегаты, выполняющие обработчики

Когда вы вызываете Invoke объекта-делегата, находящегося в начале списка, класс MulticastDelegate перечисляет список и выполняет метод Invoke каждого делегата в цепочке. Методы-обработчики выполняются последовательно и синхронно.

Объект-делегат в начале группового списка не запускает свой метод-обработчик до тех пор, пока не будет вызван метод Invoke предыдущего делегата. Управление от делегата в начале списка передается делегату в конце списка, и первым выполняется метод-обработчик делегата, размещенного в конце списка. То есть обработчики всегда выполняются с конца к началу. Следовательно, в моем примере с двумя делегатами, объединенными в группу, метод GetApproval срабатывает до метода LogWithdrawToDB.

Обратные вызовы с помощью группового делегата

Теперь вернемся к классу BankAccount и добавим в него поддержку группового связывания. Я изменю реализацию класса так, чтобы объект BankAccount мог выполнять обратные вызовы списка методов-обработчиков. Определение класса дано на **рис. 1**.

Во-первых, обратите внимание на то, что поле handler переименовано в handlers; это подчеркивает, что данное поле может хранить ссылку на групповой объект-делегат. Однако само поле по-прежнему базируется на типе-делегате LargeWithdrawHandler.

Я также изменил реализацию метода RegisterHandler. Теперь она вызывает метод Combine для добавления нового объекта-делегата в существующий список. Заметьте, что эта реализация RegisterHandler передает новый делегат во втором параметре при вызове Combine. То есть новый делегат попадет в начало списка и будет выполнен последним. Однако RegisterHandler можно переписать так, чтобы новый делегат помещался в конец списка и поэтому выполнялся первым:

```
Sub RegisterHandler(ByVal handler As LargeWithdrawHandler)
  '*** Добавить новый обработчик к концу группового списка
  Dim NewList As [Delegate] = _
    [Delegate].Combine(handler, handlers)
  handlers = CType(NewList, LargeWithdrawHandler)
End Sub
```

Вероятно, вы заметили и тот факт, что в методе Withdraw мне понадобилось лишь переименовать поле handler в handlers. Вызов Invoke осуществляется точно так же, как и раньше. Это одно из наиболее важных преимуществ групповых делегатов. При их применении можно не беспокоиться, сколько именно обработчиков связывается с объектом-делегатом. Источник уведомлений просто вызывает Invoke, и все методы-обработчики выполняются автоматически.

После модификации класса BankAccount можно переписать приложение, чтобы оно регистрировало три разных метода-обработчика, способных реагировать на уведомления о снятии крупных сумм со счета:

```
Sub Main()
  '*** Создать объект BankAccount
  Dim acc1 As New BankAccount()
  '*** Зарегистрировать методы-обработчики
  acc1.RegisterHandler(AddressOf _
     AccountHandlers.GetApproval)
  acc1.RegisterHandler(AddressOf _
     AccountHandlers.LogWithdrawToDB)
  acc1.RegisterHandler(AddressOf _
     AccountHandlers.LogWithdrawToFile)
  '*** Сделать что-нибудь, что приведет
  '*** к инициации обратного вызова
  acc1.Withdraw(5001)
End Sub
```

Отсюда должно быть ясно, что групповое связывание позволяет размещать методы-обработчики так, чтобы они выполнялись в предсказуемом порядке. Высокоуровневая схема этой архитектуры показана на **рис. 2**. Метод GetApproval выполняется первым, так как он помещен в конец списка. Сле-

Рис. 1. Класс BankAccount

```
Class BankAccount
   Private handlers As LargeWithdrawHandler
   Sub RegisterHandler(ByVal handler As LargeWithdrawHandler)
    '*** Добавить новый обработчик в начало группового списка
    Dim NewList As [Delegate] = _
       [Delegate].Combine(handlers, handler)
    handlers = CType(NewList, LargeWithdrawHandler)
   End Sub
   Sub Withdraw(ByVal Amount As Decimal)
    '*** Посылать уведомления, если это требуется
    If (Amount > 5000) AndAlso (Not handlers Is Nothing) Then
      handlers.Invoke(Amount)
    End If
    '*** Провести снятие денег со счета
   End Sub
End Class
```

дующим срабатывает метод LogWithdrawFromDB. Метод LogWithdrawToFile включен в начало списка и выполняется последним.

На **рис. 3** показано законченное приложение, построенное на архитектуре, о которой я только что рассказал. Прежде чем двигаться дальше, вы должны понять, что создается здесь с помощью делегатов: архитектура с нежесткими связями для реализации обратных вызовов с поддержкой группового связывания. Это приложение нетрудно адаптировать под поддержку еще большего числа методов-обработчиков и их регистрацию через делегаты.

Вызов метода GetInvocationList

Во многих случаях источнику уведомлений достаточно вызывать Invoke для выполнения всех целевых методов-обработчиков, сопоставленных с групповым объектом-делегатом. Однако бывают ситуации, когда может понадобиться более тонкое управление. Например, чтобы определить, сколько целевых методов-обработчиков добавлено в список делегатов. Или чтобы написать код, способный корректно справляться с

Рис. 3. Реализация архитектуры на основе делегатов для уведомления об обратных вызовах

```
Imports System

Delegate Sub LargeWithdrawHandler(ByVal Amount As Decimal)

Class BankAccount
  Private handlers As LargeWithdrawHandler
  Sub RegisterHandler(ByVal handler As LargeWithdrawHandler)
    Dim NewList As [Delegate] = _
      [Delegate].Combine(handlers, handler)
    handlers = CType(NewList, LargeWithdrawHandler)
  End Sub
  Sub Withdraw(ByVal Amount As Decimal)
    '*** Посылать уведомления, если это требуется
    If (Amount > 5000) AndAlso (Not handlers Is Nothing) Then
      handlers.Invoke(Amount)
    End If
    '*** Провести снятие денег со счета
  End Sub
End Class

Class AccountHandlers
  Shared Sub GetApproval(ByVal Amount As Decimal)
    '*** Блокировать операцию снятия денег со счета
    '*** до тех пор, пока ее не одобрит менеджер
  End Sub
  Shared Sub LogWithdrawToDB(DBByVal Amount As Decimal)
    '*** Записать информацию о снятой сумме в базу данных
  End Sub
  Shared Sub LogWithdrawToFile(ByVal Amount As Decimal)
    '*** Записать информацию о снятой сумме в файл журнала
  End Sub
End Class

Module MyApp
  Sub Main()
    '*** Создать объект BankAccount
    Dim acc1 As New BankAccount()
    '*** Зарегистрировать методы обратного вызова
    acc1.RegisterHandler(AddressOf _
      AccountHandlers.GetApproval)
    acc1.RegisterHandler(AddressOf _
      AccountHandlers.LogWithdrawToDB)
    acc1.RegisterHandler(AddressOf _
      AccountHandlers.LogWithdrawToFile)
    '*** Сделать что-нибудь, что приведет
    '*** к инициации обратного вызова
    acc1.Withdraw(5001)
  End Sub
End Module
```

исключениями, которые генерируются методами-обработчиками, включенными в список.

Класс Delegate предоставляет открытый метод экземпляра (instance method) с именем GetInvocationList. Когда вы вызываете этот метод применительно к групповому делегату, он возвращает массив ссылок на индивидуальные объекты-делегаты. Этот массив позволяет определять, сколько методов-обработчиков связано с групповым делегатом на данный момент:

```
'*** Определить число целевых методов-обработчиков
Dim HandlerCount As Integer = handlers.GetInvocationList().Length
```

Кроме того, вызов GetInvocationList в какой-то мере упрощает перебор объектов-делегатов и явный запуск их методов-обработчиков (по одному единовременно). Поскольку GetInvocationList возвращает массив ссылок на объекты-делегаты, вы можете перебирать делегаты в цикле For Each:

```
Sub Withdraw(ByVal Amount As Decimal)
  '*** Посылать уведомления, если это требуется
  If Not (Amount > 5000) AndAlso (Not handlers Is Nothing) Then
    Dim handler As LargeWithdrawHandler
    For Each handler In handlers.GetInvocationList()
      handler.Invoke(Amount)
    Next
  End If
  '*** Провести снятие денег со счета
End Sub
```

Код, который вы только что видели, на самом деле всего лишь вызывает метод Invoke объектов-делегатов, ссылки на которые хранятся в поле handlers. И тут возникает вопрос, зачем вообще вызывать GetInvocationList для перечисления объектов-делегатов в списков. Одна из причин — бóльшая гибкость в управлении, когда метод-обработчик генерирует исключение.

Делегаты — основной механизм для асинхронного выполнения метода во вторичном потоке.

Рассмотрим пример. Представьте себе групповой делегат, связанный с 10 методами-обработчиками. Что будет, если вы вызовете Invoke, а седьмой обработчик сгенерирует исключение? Первые шесть обработчиков выполнены успешно. Исключение, сгенерированное седьмым обработчиком, приводит к неожиданному завершению метода Invoke. Следовательно, восьмой, девятый и десятый обработчики вообще не будут выполнены.

Проблема в том, что при таком подходе нельзя выяснить, какие обработчики выполнены успешно, какие — нет и кто из них вызвал исключение. Но вы сможете узнать все это, явно вызывая Invoke для каждого объекта-делегата в блоке Try:

```
Dim handler As LargeWithdrawHandler
For Each handler In handlers.GetInvocationList()
  Try
    handler.Invoke(Amount)
  Catch ex As Exception
    '*** Обработать исключение и продолжить
  End Try
Next
```

Вторая причина, по которой вам может понадобиться вызов GetInvocationList и доступ к индивидуальным объектам-делегатам, заключается в том, что это позволяет получать возвращаемые значения и выходные параметры более чем из одного метода-обработчика. Если вы вызываете Invoke применительно к групповому объекту-делегату и используете выходной параметр или возвращаемое значение, то получите параметр или значение, которое возвращается методом-обработчиком, выполняемым последним, т. е. первым делегатом в списке. Однако, написав дополнительный код для перебора делегатов в списке и явно вызывая Invoke для каждого делегата, вы сможете получить выходной параметр и возвращаемое значение любого метода-обработчика.

Заметьте, что GetInvocationList возвращает массив, представляющий моментальный снимок списка делегатов. Иначе говоря, GetInvocationList предоставляет список делегатов, существовавший на момент вызова этого метода. Если вы добавите новый объект к групповому делегату, массив, полученный от GetInvocationList ранее, окажется некорректным. В этом случае вы должны вновь вызвать GetInvocationList, чтобы получить обновленный список делегатов.

Заключение

В этой и предыдущей статьях мы обсудили принципы программирования с использованием делегатов. Теперь вы знаете, что делегат — это программируемый механизм связывания для реализации обратного вызова между источником уведомлений и одним или более методами-обработчиками. Преимущество делегатов в том, что они сочетают в себе контроль типов и полиморфизм интерфейсов с эффективностью и гибкостью указателей на функции.

Без понимания фундаментальных принципов программирования с использованием делегатов нельзя стать продвинутым программистом на Visual Basic .NET. Причин тому, на мой взгляд, две. Во-первых, вся обработка событий в .NET Framework строится исключительно на делегатах. Если вы действительно хотите выжать максимум возможного из таких событийно-управляемых прикладных инфраструктур, как Windows Forms или ASP.NET, тогда вам следует спуститься на более низкий уровень и оперировать делегатами (когда это нужно). А во-вторых, делегаты являются основным механизмом для асинхронного выполнения метода в другом потоке.

Вопросы и комментарии (на английском языке) присылайте по адресу instinct@microsoft.com.

Тэд Пэттисон (Ted Pattison) – преподаватель и исследователь в DevelopMentor (http://www.develop.com), где он является одним из руководителей курса по Visual Basic. Автор книги «Programming Distributed Applications with COM and Microsoft Visual Basic 6.0» (Microsoft Press, 2000).

Под капотом

Мэт Петрек

Просмотр метаданных в .NET EXE-файлах с помощью MetaViewer

Microsoft .NET — сравнительно новая платформа (она была в виде первой бета-версии, когда я писал эту статью), и многим еще только предстоит исследовать все ее углы и закоулки. Лично мне наиболее интересны метаданные .NET: это отправная точка в изучении других областей, связанных с .NET. Метаданные представляют собой информацию, которая используется общеязыковой исполняющей средой (common language runtime, CLR) и описывает все, что относится к классам, функциям, свойствам, ресурсам и прочим элементам в исполняемом файле. Сегодня я расскажу о своей программе MetaViewer, которая отображает метаданные, содержащиеся в исполняемом .NET-файле.

MetaViewer вовсе не является самой авторитетной или совершенной программой для изучения метаданных — если вам нужно нечто в таком духе, возьмите ILDASM из .NET SDK. Нет, MetaViewer просто дала мне возможность написать нетривиальный .NET-код, который делает кое-что полезное. Кроме того, MetaViewer — сравнительно несложная программа, и ее можно легко расширить так, чтобы она сообщала детали, интересные именно вам.

Почему я решил заново изобрести колесо и написать собственное средство просмотра метаданных, когда есть готовые утилиты от Microsoft? Во-первых, мне хотелось посмотреть, что дают Windows Forms, а во-вторых (и это главное), меня не удовлетворяет текущая версия средств просмотра метаданных. ILDASM — отличный инструмент, если речь идет о полноте предоставляемой информации, но его исходный код закрыт. К тому же ILDASM отображает информацию в таком формате, который пришелся бы по душе только пионерам, стоявшим у истоков компьютерной индустрии.

Помимо ILDASM, Microsoft предлагает еще один браузер метаданных — ClsView; он находится в каталоге \Framework-SDK\Samples\ClsView. Хотя утилита ClsView написана очень аккуратно, беда в том, что это приложение ASP.NET. Если на вашем компьютере нет службы Internet Information Services (IIS), ClsView совершенно бесполезна. Можете обозвать меня брюзгой, но я пока не готов к конвертации всех своих программ в Web-приложения.

Обзор метаданных

Если вы занимались программированием под COM, то, видимо, знакомы с IDL и библиотеками типов. Эти взаимосвязанные технологии позволяют описывать интерфейс и его методы. Для каждого метода хранится информация о всех его параметрах, в том числе об их типах. Все эти данные нужны COM Automation, а также, например, при маршалинге параметров между COM-компонентами, выполняемыми в разных процессах.

Метаданные .NET можно считать IDL и библиотеками типов на стероидах. Метаданные намного полнее и точнее, чем IDL. Еще важнее, что они не являются чем-то необязательным. .NET CLR абсолютна зависима от метаданных, потому что на их основе она узнает, какие сборки используются вашим кодом, что представляют собой ваши методы, как ваши классы структурируются в памяти, какие ресурсы доступны в исполняемом файле и многое другое. Прежде чем описывать исходный код MetaViewer, я решил дать обзор метаданных — такими, какими они видятся через .NET-классы отражения (reflection classes).

Иерархия метаданных начинается с класса System.Reflection.Assembly. Объект Assembly соответствует одной или нескольким DLL, из которых состоит .NET-сборка (assembly). Класс Assembly содержит массу сведений, включая список модулей (DLL), образующих сборку, описание находящихся в ней ресурсов (битовые карты и т. д.), а также версию сборки.

CLR абсолютна зависима от метаданных, потому что на их основе она узнает, какие сборки используются вашим кодом, что представляют собой ваши методы и многое другое.

Под классом Assembly располагается класс System.Reflection.Module. Он представляет единственную DLL. В настоящее время большинство сборок состоит из одного модуля, но полагаться на такую корреляцию между Assembly и Module не стоит. Класс Module также является контейнером для типов в модуле. На данный момент можно считать, что тип соответствует .NET-классу, определенному на любом языке.

Класс System.Type представляет .NET-тип: определение класса, определение интерфейса или класс значения (value class) (обычно структуру).

У каждого экземпляра класса Type имеется набор (collection) членов, описываемых экземпляром класса System.Reflection.MemberInfo; при этом каждый член может быть одним из перечисленных:

```
Method      (System.Reflection.MethodInfo)
Constructor (System.Reflection.ConstructorInfo)
Property    (System.Reflection.PropertyInfo)
Field       (System.Reflection.FieldInfo)
Event       (System.Reflection.EventInfo)
```

Получить доступ к информации обо всех членах Type можно через класс MemberInfo. Однако каждый экземпляр

MemberInfo на самом деле является базовым классом для одного или нескольких производных типов, перечисленных выше. Следовательно, если данный экземпляр MemberInfo представляет поле, вы можете привести его к FieldInfo и тем самым получить доступ к информации, специфичной для полей.

Что касается классов MethodInfo и ConstructorInfo, то они создают в иерархии метаданных еще один уровень. У методов и конструкторов могут быть параметры, и эти параметры представляет класс System.Reflection.ParameterInfo. Для параметра можно получить его тип (как System.Type) и (в большинстве случаев) имя.

Для полноты картины вернемся к классу System.Reflection.Module. Объект Module содержит не только все экземпляры System.Type, но и все методы и поля, объявленные на уровне модуля, т. е. вне определения какого-либо класса. Обычно это имеет место в C++ с Managed Extensions, когда функции и переменные объявляются на уровне файла или на глобальном уровне.

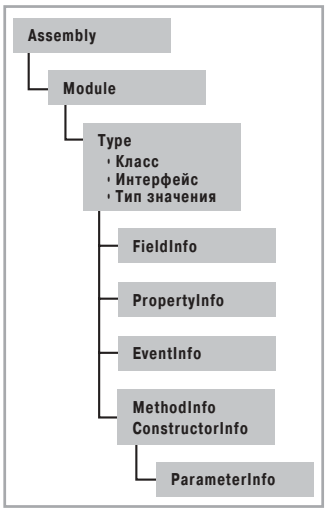


Рис. 1. Иерархия метаданных

Иерархия метаданных, о которой я только что рассказал, показана на **рис. 1**. Если вас интересует более обстоятельный обзор метаданных .NET, прочтите мою статью «Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework» в номере за октябрь 2000 г. (http://msdn.microsoft.com/msdnmag/issues/1000/metadata/metadata.asp).

Windows Forms

Первый опыт работы с Windows Forms я получил, используя инструмент WinDes, поставляемый с .NET SDK. Выбрав создание новой формы на основе Win32 в C#, я добавил на нее элемент управления «древовидный список» (treeview control), текстовое поле (textbox), поле ввода (edit control) и несколько кнопок. После сохранения этой формы я с удивлением заметил, что WinDes создал единственный CS-файл — без дополнительного файла, описывающего разметку формы.

Изучив код, сгенерированный для формы, я обнаружил, что он динамически создает все элементы управления в процессе инициализации формы, причем каждый из них представляется как экземпляр .NET-объекта, а форма содержит переменные-члены, ссылающиеся на эти элементы. Так, в следующем фрагменте кода создаются текстовое поле и иерархический список, а затем инициализируются соответствующие переменные-члены формы:

```
this.text_details = new System.WinForms.TextBox();
this.treeView1 = new System.WinForms.TreeView();
```

А как насчет свойств элементов управления, значения которых я задавал в WinDes? Каждое свойство настраивается единственным выражением на C# сразу после создания элемента управления. Например, эта строка указывает, что текстовое поле text_details предназначено только для чтения:

```
text_details.ReadOnly = true;
```

В некоторых аспектах программирование на основе Windows Forms принципиально отличается от традиционной разработки пользовательских интерфейсов (UI) в Win32 с применением файлов ресурсов (RC) или даже в версиях Visual Basic, появившихся до эпохи .NET. В RC-файлах и в прежних версиях Visual Basic содержимое и свойства форм хранились как структурированные данные, отделенные от кода обработки событий. Когда форма (aka диалог) создается по старой модели, некий системный код считывает информацию о разметке и конструирует форму за вас. В противоположность этому Windows Forms не отделяет разметку форм от кода, который создает и инициализирует их. Это упрощает настройку содержимого и свойств формы в процессе ее создания.

Приятная особенность программирования с использованием Windows Forms в том, что, если вы привыкли к созданию UI на Visual Basic, вы на 95% готовы к работе с Windows Forms. Вместо того чтобы работать с голым User32, вы имеете дело с .NET-классами, инкапсулирующими окна, формы и элементы управления. При этом, как правило, не приходится возиться с оконными сообщениями и заниматься прочими низкоуровневыми махинациями.

Единственное заметное на мой взгляд отличие при программировании с применением Windows Forms — в поддержке обработчиков событий. Такие обработчики добавляются динамически. Чтобы создать обработчик события, вы создаете экземпляр его класса. Конструктор класса принимает аргумент, указывающий функцию, с помощью которой вы хотите обрабатывать данное событие. Затем экземпляр обработчика связывается с конкретным элементом управления. В C# это делается с помощью оператора +=. Вот пример, где метод treeView1_AfterSelect определяется как обработчик события TreeView.AfterSelect:

```
treeView1.AfterSelect +=
  new System.WinForms.TreeViewEventHandler(
  treeView1_AfterSelect);
```

Пока все, о чем я говорил, довольно просто. Но что происходит с битовыми картами, значками и прочим? Я набрел на ответ, добавив ImageList к форме, чтобы помечать узлы TreeView броскими картинками, подсказывающими смысл каждого узла. Эта операция заставила WinDes сгенерировать второй файл — на этот раз с расширением .resX. Кстати, добавление двоичных «ресурсов» привело к созданию в коде, инициализирующем мою форму, класса System.Resources.ResourceManager, который обеспечивает доступ к скомпилированным ресурсам.

Заглянув в resX-файл, я обнаружил, что на самом деле это XML-файл. Чтобы преобразовать его в нечто, чем можно пользоваться, запустите программу ResGen — она транслирует XML-файл в двоичный с расширением .resource. После этого вы можете встроить resource-файл в исполняемый или оставить его как отдельный файл в сборке. Если вы работаете на C#, то при параметре /res resource-файл помещается в исполняемый, а при параметре /linkres остается отдельно от него.

Программа MetaViewer

Проектируя MetaViewer, я решил, что она будет:

- принимать в командной строке имя исполняемого .NET-файла и отображать типы и их члены;
- сообщать все детали по каждому типу и члену в отдельной секции окна;

- искать тип по любой части его имени;
- выводить информацию уровня сборки (например, сведения об импортируемых сборках) в отдельном окне.

Главная форма MetaViewer показана на **рис. 2**. На этой иллюстрации MetaViewer отображает собственные метаданные. Исходный код для этой формы находится в файле MetaViewer.cs (**рис. 3**). (Для удобства чтения я не привожу код, сгенерированный WinDes.) Изучив его, вы заметите, что он представляет собой смесь кода Windows Forms и кода для доступа к метаданным через классы отражения.

Основная часть формы MetaViewer содержит TreeView со всеми типами. Каждый узел TreeView верхнего уровня можно раскрыть и просмотреть члены

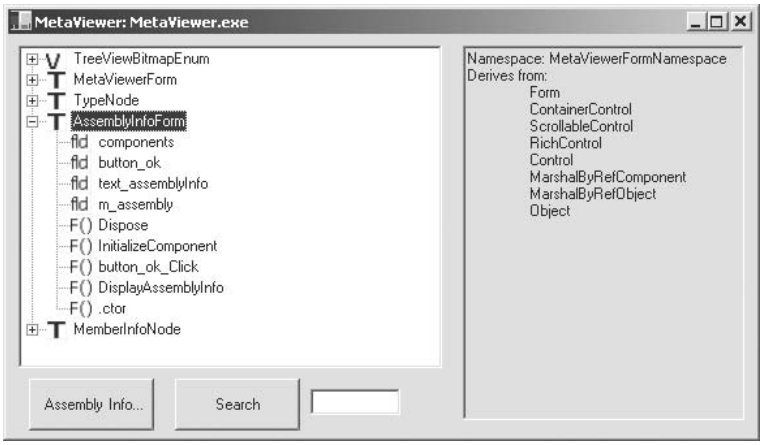


Рис. 2. Форма MetaViewer

Рис. 3. MetaViewer.cs

```
//=======================================
// Мэт Петрек
// MSDN Magazine, март 2001
// Файл: MetaViewer.cs
//=======================================
namespace MetaViewerFormNamespace
{
    using System;
    using System.Drawing;
    using System.ComponentModel;
    using System.WinForms;
    using System.Reflection;
    using System.Text.RegularExpressions;

    // Перечислимое, связывающее узлы TreeView
    // с соответствующими значками
    enum TreeViewBitmapEnum
    {
        Unknown = 0,
        Type = 1,
        Interface = 2,
        ValueClass = 3,
        Method = 4,
        Field = 5,
        Property = 6,
        Event = 7
    }

    public class MetaViewerForm : System.WinForms.Form
    {
        private string m_strFileName;
        private Assembly m_assembly = null;
        private System.ComponentModel.Container components;
        public System.WinForms.ImageList imageList1;
        private System.WinForms.TextBox text_details;
        private System.WinForms.TreeView treeView1;
        private System.WinForms.TextBox text_search;
        private System.WinForms.Button button_search;
        private System.WinForms.Button button_assembly;

        public MetaViewerForm( string strFileName )
        {
            m_strFileName = strFileName;

            // Требуется для поддержки дизайнера Windows Forms
            InitializeComponent();

            DisplayMetaData( );
        }

        public override void Dispose()
        {
            base.Dispose();
            components.Dispose();
        }

        public static void Main(string[] args)
        {
            if (args.Length < 1 )
            {
                MessageBox.Show("Syntax: MetaViewer <filename>");
                return;
            }

            try
            {
                Application.Run(new MetaViewerForm(args[0]));
            }

            catch( Exception e )
            {
                e.ToString(); // убиваем предупреждение
                              // от компилятора насчет
                              // "неиспользуемого e" :-)
            }
        }

        private void InitializeComponent()
        {
            System.Resources.ResourceManager resources =  new
                System.Resources.ResourceManager(
                typeof(MetaViewerForm));
            this.components = new System.ComponentModel.Container();
            this.imageList1 = new System.WinForms.ImageList();
            this.text_details = new System.WinForms.TextBox();
            this.text_search = new System.WinForms.TextBox();
            this.treeView1 = new System.WinForms.TreeView();
            this.button_search = new System.WinForms.Button();
            this.button_assembly = new System.WinForms.Button();

            imageList1.ImageSize = new System.Drawing.Size(16, 16);
            imageList1.ImageStream = (System.WinForms.
                ImageListStreamer) resources.GetObject(
                "imageList1.ImageStream");
            imageList1.ColorDepth = System.WinForms.
                ColorDepth.Depth8Bit;
            imageList1.TransparentColor = System.
                Drawing.Color.Transparent;

            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.Text = "MetaViewer";
            this.ClientSize = new System.Drawing.Size(728, 412);

            text_details.Location = new System.Drawing.Point(504, 8);
            text_details.Text = "";
            text_details.Multiline = true;
            text_details.TabIndex = 0;
            text_details.ReadOnly = true;
            text_details.TabStop = false;
            text_details.Size = new System.Drawing.Size(216, 392);

            treeView1.Location = new System.Drawing.Point(8, 8);
            treeView1.Text = "treeView1";
            treeView1.Size = new System.Drawing.Size(480, 352);
```

см. след. стр.

Рис. 3. MetaViewer.cs *(продолжение)*

```
    treeView1.TabIndex = 0;
    treeView1.AfterSelect += new
        System.WinForms.TreeViewEventHandler(
        treeView1_AfterSelect);
    treeView1.ImageList = imageList1;
    treeView1.HideSelection = false;

    text_search.Location = new System.
        Drawing.Point(232, 376);
    text_search.TabIndex = 3;
    text_search.Size = new System.Drawing.Size(224, 20);

    button_search.Location = new System.
        Drawing.Point(128, 368);
    button_search.Size = new System.Drawing.Size(96, 40);
    button_search.TabIndex = 2;
    button_search.Text = "Search";
    button_search.Click += new
        System.EventHandler(button_search_Click);

    button_assembly.Location = new System.
        Drawing.Point(16, 368);
    button_assembly.Size = new System.Drawing.Size(96, 40);
    button_assembly.TabIndex = 1;
    button_assembly.Text = "Assembly Info...";
    button_assembly.Click +=
        new System.EventHandler(button_assembly_Click);

    this.Controls.Add(text_details);
    this.Controls.Add(treeView1);
    this.Controls.Add(text_search);
    this.Controls.Add(button_search);
    this.Controls.Add(button_assembly);
}

//=====================================================
// Поиск Type, содержащего строку в текстовом
// элементе управления
//=====================================================
protected void button_search_Click(object sender,
                                   System.EventArgs e)
{
    int currentIndex = treeView1.SelectedNode.Index;
    int maxIndex = treeView1.GetNodeCount(false);

    // Параметр "i" означает, что поиск следует
    // выполнять, не обращая внимания на регистр букв
    Regex regex = new Regex( text_search.Text, "i" );
    TreeNode node = treeView1.Nodes[currentIndex];

    if ( regex.IsMatch(node.Text) )
        currentIndex++;

    int i;
    for ( i = currentIndex; i < maxIndex; i++ )
    {
        node = treeView1.Nodes[i];

        // if ( node.Text == text_search.Text )
        if ( regex.IsMatch(node.Text) )
        {
            treeView1.SelectedNode = node;
            ActiveControl = treeView1;
            break;
        }
    }

    if ( i == maxIndex )
        MessageBox.Show( "No matches found" );
}

//=====================================================
// Если нажата кнопка "Assembly Info...", показать окно
//=====================================================
protected void button_assembly_Click( object sender,
                                      System.EventArgs e)
{
    AssemblyInfoForm assemblyInfoForm =
    new AssemblyInfoForm( m_assembly );
    assemblyInfoForm.Show();
}

//=====================================================
// Если выбран узел в TreeView, обновить секцию детальных
// сведений
//=====================================================
protected void treeView1_AfterSelect( object sender,
    System.WinForms.TreeViewEventArgs e)
{
    text_details.Clear();
    if ( e.node is TypeNode )
    {
        ShowTypeDetails( ((TypeNode)e.node).m_type );
    }
    else if ( e.node is MemberInfoNode )
    {
        ShowMemberInfoDetails(((
            MemberInfoNode)e.node).m_memberInfo);
    }
}

//=====================================================
// Процедура верхнего уровня, вызываемая для отображения
// метаданных из сборки
//=====================================================
private void DisplayMetaData()
{
    try
    {
        m_assembly = Assembly.Load( m_strFileName );
    }
    catch( Exception e )
    {
        MessageBox.Show( e.Message );
        throw e; // Application.Exit здесь,
                 // похоже, не поддерживается...
    }

    // Перебираем все модули в сборке
    foreach ( Module module in m_assembly.GetModules() )
    {
        // Добавляем типы из модуля в TreeView
        PopulateTypes( module.GetTypes() );

        // Добавляем методы и свойства уровня модуля
        PopulateMembers( module.GetMethods(),
            treeView1.Nodes );
        PopulateMembers( module.GetFields(),
            treeView1.Nodes );
    }
    this.Text = "MetaViewer: " + m_strFileName;
}

//=====================================================
// Заполняет основной TreeView типами из модуля
//=====================================================
protected void PopulateTypes( Type[] arTypes )
{
    foreach ( Type t in arTypes )
    {
        TypeNode typeNode = new TypeNode( t );
        TreeViewBitmapEnum treeViewBitmapEnum;
        if ( t.IsClass )
            treeViewBitmapEnum = TreeViewBitmapEnum.Type;
        else if ( t.IsInterface )
            treeViewBitmapEnum =
                TreeViewBitmapEnum.Interface;
        else if ( t.IsValueType )
            treeViewBitmapEnum =
                TreeViewBitmapEnum.ValueClass;
        // Пока обрабатываем управляемые и неуправляемые
        // классы значений одинаково
        else if ( t.IsUnmanagedValueType )
            treeViewBitmapEnum =
                TreeViewBitmapEnum.ValueClass;
        else
            treeViewBitmapEnum = TreeViewBitmapEnum.Unknown;
```

см. стр. 85

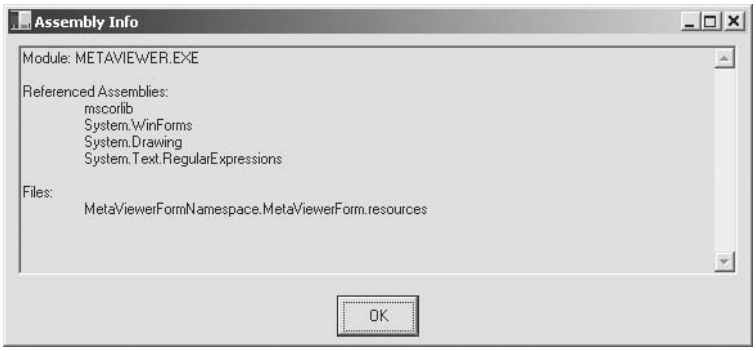


Рис. 4. Assembly Info

данного типа. В правой секции окна показываются детальные сведения о выбранном элементе. Если выбран какой-нибудь тип, в ней сообщается о пространстве имен, которому он принадлежит, и полная иерархия наследования. Если же вы раскрываете узел типа и указываете один из его членов, выводится соответствующая информация о данном члене. Для метода в секции детальных сведений показывается, есть ли у него атрибуты virtual, static, public, private или PInvoke (Platform Invoke), и перечисляются параметры метода и тип возвращаемого значения. В случае поля вы увидите его тип и атрибут (static, public или private). Как извлекаются все эти сведения, вам станет понятно из исходного кода метода ShowMemberInfoDetails на **рис. 3**.

Внизу главной формы MetaViewer расположены две кнопки и поле ввода. Чтобы найти какой-либо тип, просто наберите любую часть его имени (регистр букв не имеет значения) в поле ввода, а затем нажмите кнопку Search. Если программа найдет совпадение, соответствующий тип будет выделен подсветкой. Поиск начинается с текущего выбранного узла, поэтому вы можете повторно нажимать кнопку Search для поиска других типов с похожими именами. Я готов первым согласиться, что мой UI для

Рис. 5. AssemblyInfoForm.cs

```
//=======================================
// Мэт Петрек
// MSDN Magazine, март 2001
// Файл: AssemblyInfoForm.cs
//=======================================
namespace MetaViewerFormNamespace
{
    using System;
    using System.WinForms;
    using System.Reflection;

    public class AssemblyInfoForm : System.WinForms.Form
    {
        private System.ComponentModel.Container components;
        private System.WinForms.Button button_ok;
        private System.WinForms.TextBox text_assemblyInfo;
        private Assembly m_assembly;

        public AssemblyInfoForm(Assembly assembly)
        {
            m_assembly = assembly;
            // Требуется для поддержки дизайнера Windows Forms
            InitializeComponent();
            DisplayAssemblyInfo();
        }

        public override void Dispose() {
            base.Dispose();
            components.Dispose();
        }

        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.button_ok = new System.WinForms.Button();
            this.text_assemblyInfo = new System.WinForms.TextBox();

            button_ok.Location = new System.Drawing.Point(256, 320);
            button_ok.Size = new System.Drawing.Size(64, 32);
            button_ok.TabIndex = 0;
            button_ok.Text = "OK";
            button_ok.Click += new System.EventHandler(
                button_ok_Click);

            text_assemblyInfo.Location = new System.
                Drawing.Point(8, 8);
            text_assemblyInfo.ReadOnly = true;
            text_assemblyInfo.Multiline = true;
            text_assemblyInfo.TabIndex = 0;
            text_assemblyInfo.TabStop = false;
            text_assemblyInfo.Size = new System.
                Drawing.Size(560, 296);
            text_assemblyInfo.ScrollBars = ScrollBars.Both;

            this.Text = "Assembly Info";
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(576, 356);

            this.Controls.Add(button_ok);
            this.Controls.Add(text_assemblyInfo);
        }

        protected void button_ok_Click(object sender,
                                       System.EventArgs e)
        {
            this.Close();
        }

        //=====================================================
        protected void DisplayAssemblyInfo()
        {
            // Выводим список модулей в сборке
            foreach ( Module module in m_assembly.GetModules() )
            {
                text_assemblyInfo.AppendText(
                    String.Format("Module: {0}\r\n", module.Name) );
            }

            //=================================================
            // Выводим список сборок,
            // на которые ссылается данная сборка
            text_assemblyInfo.AppendText( "\r\n" );
            text_assemblyInfo.AppendText(
                "Referenced Assemblies:\r\n" );

            foreach( AssemblyName assemblyName in
                m_assembly.GetReferencedAssemblies() )
            {
                text_assemblyInfo.AppendText(
                    String.Format("\t{0}\r\n", assemblyName.Name) );
            }

            //=================================================
            // Выводим список файлов в сборке
            text_assemblyInfo.AppendText( "\r\n" );
            text_assemblyInfo.AppendText( "Files:\r\n" );

            foreach ( String strResourceName in
                m_assembly.GetManifestResourceNames() )
            {
                text_assemblyInfo.AppendText(
                    String.Format("\t{0}\r\n", strResourceName) );
            }
        }
    }
}
```

поиска не столь интуитивно понятен, как в коммерческих программах, но результат все же неплох, учитывая, что этот UI реализован лишь парой строк кода.

Наконец, кнопка Assembly Info открывает отдельную форму, показанную на **рис. 4**. Исходный код для этой формы находится в файле AssemblyInfo.cs (**рис. 5**). Хотя метаданные содержат тонны информации о сборке, я извлекаю лишь самую важную: имя модуля, список импортируемых сборок и список файлов (как правило, ресурсов), принадлежащих данной сборке.

Вернемся к главной форме. Одно из моих приятных впечатлений от использования Windows Forms и .NET — легкость связывания метаданных с их представлением в Windows Forms. Я просто позволил UI-классам хранить мои данные. Ключ к этой простоте кроется в наследовании.

Используя элемент управления TreeView из Windows Forms, вы добавляете элементы включением экземпляров TreeNode. Объявив собственные классы производными от TreeNode, я смог хранить информацию, извлеченную из метаданных, в том же классе, где и данные TreeNode. Пример тому вы найдете в классе MemberInfoNode, объявленном в MemberNode.cs (**рис. 6**). Этот класс наследует от TreeNode, в который добавляется поле для хранения экземпляра MemberInfo.

Кроме MemberInfoNode, я создал класс TypeNode, тоже производный от TreeNode. TypeNode представляет тип метаданных и включает поле для хранения соответствующего экземпляра System.Type. Чтобы добавить узел члена в TreeView, я передаю либо TypeNode, либо MemberInfoNode, а не просто TreeNode.

Когда происходит событие TreeView (например, указан какой-нибудь узел), его обработчик получает ссылку на выбранный TreeNode. Мой код берет TreeNode и приводит его обратно к MemberInfoNode или TypeNode, чтобы извлечь нужную информацию. Откуда я узнаю, в какой производный класс следует выполнить преобразование? Здесь весьма кстати C#-оператор is:

```
if ( e.node is TypeNode )
{
    ShowTypeDetails( ((TypeNode)e.node).m_type );
}
else if ( e.node is MemberInfoNode )
{
    ShowMemberInfoDetails(
        ((MemberInfoNode)e.node).m_memberInfo );
}
```

Решение проблем

Одна из проблем, с которыми я столкнулся в MetaViewer, состояла в том, что я мог видеть только открытые члены типов в сборке. Вернувшись к своей первоначальной программе ReflectMeta (она была описана в моей статье, опубликованной в октябре 2000 г.), я обнаружил, что и с ней точно та же проблема. Попытавшись найти ее причину, ничего очевидного я не заметил. И уже хотел было написать сообщение о выявленной ошибке, но в последний момент передумал: проблема настолько бросалась в глаза, что вряд ли могла быть вызвана ошибкой в .NET Framework.

Снова вернувшись к описанию метода GetMember в System.Type, я увидел, что это перегруженный метод. Моя программа вызывала простой метод GetMember без всяких параметров. Внимательное чтение документации подсказало мне, что он возвращает лишь открытые члены типа. Чтобы получить полный список членов, следовало вызывать другой метод GetMember — тот, который принимает параметр BindingFlags.

Урок из этой истории таков: библиотека .NET-классов огромна, и в ней активно используется перегрузка методов. Когда не удается добиться нужного, попробуйте вернуться назад и найти подходящие перегруженные методы. Кстати, MFC-программистам будет гораздо легче освоить .NET Framework, чем их коллегам, использовавшим Visual Basic.

С другой проблемой я столкнулся при реализации поиска типа по имени. В C++ моим другом была функция strstr, а в .NET мне не удалось найти ничего похожего. Я потратил немало времени, проверяя и перепроверяя методы класса String, но все было тщетно, пока я, наконец, не набрел на класс RegEx (Regular Expression) в пространстве имен System.Text.RegularExpressions.

Через пару минут мой код уже перебирал каждый из узлов Type и использовал метод RegEx.IsMatch, проверяя, подходит ли имя типа к искомой строке. К сожалению, этот метод чувствителен к регистру букв. Вспомнив предыдущий урок с перегруженными методами, я вернулся назад и обнаружил, что у класса RegEx есть второй конструктор, который позволяет указывать параметры поиска. Финальную реализацию кода поиска в методе button_search_Click можно увидеть на **рис. 3**.

Третья проблема возникла с неуправляемыми типами значений (unmanaged value types). Обычно это структуры классического C++, используемые для взаимодействия (interop), и память под них не выделяется из управляемой кучи (managed heap). Содержимое .NET-файла corhdr.h свидетельствует о том, что применение неуправляемых типов значений не поощряется. Однако я нашел много мест, где они все равно используются. Мне не хватило терпения создавать отдельную битовую карту TreeView еще и под неуправляемые типы значений, и я взял для них ту же битовую карту, что и для управляемых.

Разрабатывая MetaViewer, я здорово развлекся и попутно освоил многие возможности C# и .NET Framework. Пока что я не намерен отказываться от C++, но на меня произвело сильное впечатление, каких результатов можно добиться, написав сравнительно немного кода. Надеюсь, моя работа послу-

Рис. 6. MemberNode.cs

```
//=======================================
// Мэт Петрек
// MSDN Magazine, март 2001
// Файл: MemberNode.cs
//=======================================
namespace MetaViewerFormNamespace
{
    using System.WinForms;
    using System.Reflection;

    public class MemberInfoNode : TreeNode
    {
        // System.Reflection.MemberInfo сопоставлен с этим узлом
        public MemberInfo m_memberInfo;

        public MemberInfoNode(MemberInfo memberInfo)
        {
            m_memberInfo = memberInfo;
            Text = m_memberInfo.Name;
        }
    }
}
```

Рис. 3. MetaViewer.cs *(окончание)*

```
            typeNode.ImageIndex = typeNode.SelectedImageIndex =
                (int)treeViewBitmapEnum;
            treeView1.Nodes.Add( typeNode );
            MemberInfo[] arMemberInfo = t.GetMembers(
                BindingFlags.LookupAll |
                BindingFlags.DeclaredOnly
                );
            PopulateMembers( arMemberInfo, typeNode.Nodes );
        }
    }

    //=====================================================
    // Добавляет члены типа в соответствующий узел TreeView
    //=====================================================
    protected void PopulateMembers( MemberInfo[] arMemberInfo,
        TreeNodeCollection treeNodeCollection)
    {
        foreach( MemberInfo member in arMemberInfo )
        {
            MemberInfoNode memberInfoNode =
                new MemberInfoNode(member);
            TreeViewBitmapEnum treeViewBitmapEnum2;

            if ( member is MethodInfo ||
                 member is ConstructorInfo )
                treeViewBitmapEnum2 = TreeViewBitmapEnum.Method;
            else if ( member is FieldInfo )
                treeViewBitmapEnum2 = TreeViewBitmapEnum.Field;
            else if ( member is PropertyInfo )
                treeViewBitmapEnum2 =
                    TreeViewBitmapEnum.Property;
            else if ( member is EventInfo )
                treeViewBitmapEnum2 = TreeViewBitmapEnum.Event;
            else
                treeViewBitmapEnum2 =
                    TreeViewBitmapEnum.Unknown;

            memberInfoNode.ImageIndex =
                memberInfoNode.SelectedImageIndex
                = (int)treeViewBitmapEnum2;
            treeNodeCollection.Add( memberInfoNode );
        }
    }

    //=====================================================
    // Записывает в секцию детальных сведений информацию
    // о выбранном Type
    //=====================================================
    protected void ShowTypeDetails( Type t )
    {
        text_details.AppendText(
            String.Format("Namespace: {0}\r\n", t.Namespace) );

        // Показывает иерархию наследования до System.Object
        text_details.AppendText( "Derives from:\r\n" );
        while ( true )
        {
            Type baseType = t.BaseType;
            if ( baseType == null )
                break;
            text_details.AppendText( "\t" +
                baseType.Name + "\r\n" );
            t = t.BaseType;
        }
    }

    //=====================================================
    // Записывает в секцию детальных сведений информацию
    // о выбранном члене
    //=====================================================
    protected void ShowMemberInfoDetails(MemberInfo memberInfo)
    {
        if ( memberInfo is MethodBase )
        {
            MethodBase methodBase = (MethodBase)memberInfo;

        if ( methodBase.IsVirtual )
            text_details.AppendText( "Virtual\r\n" );
        if ( methodBase.IsStatic )
            text_details.AppendText( "Static\r\n" );
        if ( methodBase.IsPublic )
            text_details.AppendText( "Public\r\n" );
        if ( methodBase.IsPrivate )
            text_details.AppendText( "Private\r\n" );
        if ( (methodBase.Attributes &
            (MethodAttributes)MethodAttributes.PinvokeImpl)
            != 0 )
        {
            text_details.AppendText( "PInvoke\r\n" );
        }

        if ( memberInfo is MethodInfo )
        {
            text_details.AppendText(
                String.Format("returns {0}\r\n",
                ((MethodInfo)memberInfo).ReturnType.Name) );
        }

        ParameterInfo [] arParamInfo =
            methodBase.GetParameters();

        if ( arParamInfo.Length > 0 )
            text_details.AppendText( "Parameters:\r\n" );
            foreach ( ParameterInfo paramInfo in arParamInfo )
            {
                text_details.AppendText( String.Format(
                    "\t{0} {1}\r\n",
                    paramInfo.ParameterType.Name,
                    paramInfo.Name));
            }
        }
        else if ( memberInfo is FieldInfo )
        {
            FieldInfo fieldInfo = (FieldInfo)memberInfo;
            if ( fieldInfo.IsStatic )
                text_details.AppendText( "Static\r\n" );
            if ( fieldInfo.IsPublic )
                text_details.AppendText( "Public\r\n" );
            if ( fieldInfo.IsPrivate )
                text_details.AppendText( "Private\r\n" );

            text_details.AppendText(
                memberInfo.GetType().Name );
        }
        else if ( memberInfo is PropertyInfo )
        {
            PropertyInfo propertyInfo =
                (PropertyInfo)memberInfo;

            if ( propertyInfo.CanRead )
                text_details.AppendText( "Readable\r\n" );
            if ( propertyInfo.CanWrite )
                text_details.AppendText( "Writeable\r\n" );

            text_details.AppendText(
                memberInfo.GetType().Name );
        }
    }
}       // Конец класса MetaViewerForm
}       // Конец пространства имен
```

жит вам неплохой отправной точкой в исследовании метаданных и создании собственного средства для их просмотра.

Исходный код для этой статьи можно скачать по ссылке http://msdn.microsoft.com/msdnmag/code01.aspx в разделе за март (2001 г.).

Вопросы и комментарии (на английском языке) присылайте по адресу hood@microsoft.com.

Мэт Петрек (Matt Pietrek) – соучредитель Интернет-компании Mindreef LLC. До этого был ведущим архитектором линейки продуктов Compuware/NuMega Bounds Checker. На его сайте (http://www.wheaty.net) есть страница с часто задаваемыми вопросами (FAQ) и информацией о предыдущих статьях и колонках.