

Доступ к данным

Джонни Папа

Пять способов подстегнуть производительность SQL

Иногда, чтобы заставить приложение работать гораздо быстрее, достаточно подкрутить пару «гаек». Только надо знать — где! Рано или поздно сталкиваешься с ситуацией, когда какой-нибудь SQL-запрос в приложении действует вовсе не так, как было задумано: или не возвращает нужные данные, или выполняется слишком долго. Кому понравится, если ваше корпоративное приложение будет задерживать результаты запросов? Помните, как ваши родители и слушать не хотели объяснений, почему вы явились домой так поздно? Пользователей тоже не волнует, почему их запросы выполняются столько времени. («Прости, мама. В моем запросе было слишком много операторов LEFT JOIN.») Они хотят, чтобы приложения отвечали быстро, а аналитические данные в отчетах появлялись моментально. Меня самого раздражает, если какая-нибудь Интернет-страница грузится дольше десяти секунд. (Ну хорошо, дольше пяти.)

Помните, как ваши родители и слушать не хотели объяснений, почему вы явились домой так поздно? Пользователей тоже не волнует, почему их запросы выполняются столько времени. Они хотят, чтобы приложения отвечали быстро, а аналитические данные в отчетах появлялись моментально.

Для решения этих проблем важно разобраться в их причинах. С чего начать? Главная причина обычно связана со структурой базы данных и запросов. Сегодня я продемонстрирую четыре способа, позволяющих увеличить либо производительность, либо масштабируемость приложений на основе SQL Server. Я рассмотрю применение операторов LEFT JOIN и CROSS JOIN, а также получение значения IDENTITY. Но универсальных решений нет. Настройка базы данных и запросов требует времени, анализа и тестирования. Хотя все предлагаемые мной способы доказали свою эффективность, в вашем приложении какие-то из них окажутся лучше, чем другие.

Получение IDENTITY из запроса INSERT

Начну, пожалуй, с задачи, по которой мне задают массу вопросов: как получить значение IDENTITY после выполнения SQL-оператора INSERT? Зачастую проблема не в том, как написать соответствующий запрос, а в том, где и когда его выполнять. В SQL Server, чтобы получить значение IDENTITY, созданное в результате выполнения последнего SQL-оператора применительно к активному соединению с базой данных, используйте:

```
SELECT @@IDENTITY
```

Это совсем простой SQL-оператор, но важно иметь в виду вот что: если последним был не оператор INSERT или если он был выполнен на другом соединении, ожидаемого результата вы не получите. Этот код надо выполнять сразу после INSERT и на том же соединении, например так:

```
INSERT INTO Products (ProductName) VALUES ('Chalk')
```

```
SELECT @@IDENTITY
```

Выдав такой запрос по единственному соединению с базой данных Northwind, вы получите значение IDENTITY для нового товара — Chalk (мел). Например, в приложении на Visual Basic, использующем ADO, вы могли бы написать:

```
Set oRs = oCn.Execute("SET NOCOUNT ON; INSERT INTO Products (ProductName) _  
VALUES ('Chalk'); SELECT @@IDENTITY")
```

```
lProductID = oRs(0)
```

Этот код указывает SQL Server не возвращать число строк, выполнить оператор INSERT и вернуть значение IDENTITY, созданное для новой строки. Оператор SET NOCOUNT ON приводит к тому, что возвращаемый объект Recordset содержит одну строку и один столбец со значением IDENTITY. Без этого оператора вы получили бы сначала пустой объект Recordset (так как INSERT не возвращает никаких данных), а потом еще один объект Recordset, на этот раз со значением IDENTITY. Тут можно запутаться — особенно если не знаешь, что оператор INSERT тоже возвращает Recordset. Дело в том, что SQL Server следит за количеством строк и, если оно меняется из-за вставки строки, интерпретирует новую строку как объект Recordset. А реальные данные помещаются во второй Recordset. Хотя вы могли бы добраться до второго объекта Recordset ADO-методом NextRecordset, гораздо проще (и эффективнее) делать так, чтобы возвращался только один Recordset.

Данный способ решает задачу, но требует дополнительного кода в SQL-выражении. Другой способ добиться того же результата — ввести SET NOCOUNT ON перед INSERT, а в триггер FOR INSERT для таблицы поместить SELECT @@IDENTITY, как показано ниже. Тогда любой оператор INSERT, адресованный данной таблице, автоматически вернет значение IDENTITY.

```
CREATE TRIGGER trProducts_Insert ON Products FOR INSERT AS
```

```
SELECT @@IDENTITY
```

```
GO
```

Триггер срабатывает, только когда INSERT выполняется применительно к таблице Products, поэтому он всегда возвращает IDENTITY при успешном завершении запроса. Этот

способ позволяет унифицировать получение значений IDENTITY в рамках всего приложения.

Встраиваемые представления и временные таблицы

Иногда запросы должны объединять данные с другими данными, которые можно получить только через GROUP BY с последующей выдачей стандартного запроса. Например, если вам нужна информация о последних пяти заказах, вы сначала выясняете, что они собой представляют. Для этого выдается SQL-запрос, возвращающий идентификаторы заказов. Полученные данные помещаются во временную таблицу, после чего она объединяется с таблицей Products, и вы можете узнать, какие товары и в каком количестве проданы по этим заказам:

```
CREATE TABLE #Temp1 (OrderID INT NOT NULL, OrderDate DATETIME NOT NULL)
```

```
INSERT INTO #Temp1 (OrderID, OrderDate)
SELECT TOP 5 o.OrderID, o.OrderDate
FROM Orders o ORDER BY o.OrderDate DESC
```

```
SELECT p.ProductName, SUM(od.Quantity) AS ProductQuantity
FROM #Temp1 t
INNER JOIN [Order Details] od ON t.OrderID = od.OrderID
INNER JOIN Products p ON od.ProductID = p.ProductID
GROUP BY p.ProductName
ORDER BY p.ProductName
```

```
DROP TABLE #Temp1
```

Здесь создается временная таблица, которая заполняется данными, объединяется с другой таблицей и, наконец, удаляется. Этот запрос вызывает массу операций ввода-вывода, однако его можно переписать так, чтобы вместо временной таблицы использовалось встраиваемое представление (inline view) — запрос, выполняющий объединение в блоке FROM. При этом вы получаете тот же результат, но без интенсивного ввода-вывода:

```
SELECT p.ProductName,
SUM(od.Quantity) AS ProductQuantity
FROM (
SELECT TOP 5 o.OrderID, o.OrderDate
FROM Orders o
ORDER BY o.OrderDate DESC
) t
INNER JOIN [Order Details] od ON t.OrderID = od.OrderID
INNER JOIN Products p ON od.ProductID = p.ProductID
GROUP BY p.ProductName
ORDER BY p.ProductName
```

Этот запрос не только эффективнее предыдущего, но и короче. Кроме того, временные таблицы расходуют уйму ресурсов. Если данные нужны лишь для объединения, встраиваемые представления предпочтительнее.

Старайтесь не использовать LEFT JOIN и NULL

Конечно, бывают ситуации, когда без LEFT JOIN и значений NULL не обойтись. Но их применение далеко не всегда оправданно. Изменив структуру SQL-запросов, можно добиться того, что запрос, который выполнялся за несколько минут, бу-

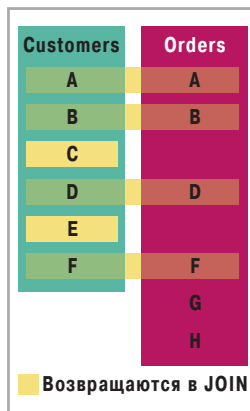


Рис. 1. Запрос

дет давать результат за несколько секунд. Иногда приходится изменять формат данных в запросе так, как это нужно вашему приложению. Кроме применения типа данных TABLE, снижающего потребление ресурсов, есть много других возможностей оптимизировать запрос. В частности, в SQL широко используется такой удобный оператор, как LEFT JOIN. Он позволяет извлекать все строки из первой таблицы и строки из второй таблицы, совпадающие и не совпадающие с первой. Так, если вам надо получить список всех клиентов и их заказов, то LEFT JOIN способен показать перечень клиентов, как сделавших заказы, так и не сделавших их.

Этим инструментом нередко злоупотребляют. LEFT JOIN приводит к серьезным издержкам, потому что сравнивает данные с NULL (несуществующими данными). В каких-то случаях без LEFT JOIN не обойтись, но цена его использования может быть очень высока. LEFT JOIN требует гораздо больше ресурсов, чем INNER JOIN, и, если вам удастся так переписать запрос, чтобы избавиться от всех LEFT JOIN, вы получите колоссальный выигрыш (рис. 1).

Один из способов ускорить выполнение запроса с LEFT JOIN — создать тип данных TABLE и вставить все строки из первой таблицы (находящейся по левую сторону LEFT JOIN), а затем добавить в TABLE значения из второй таблицы. Хотя этот процесс проходит в два этапа, он может сэкономить массу времени по сравнению со стандартным LEFT JOIN. Но лучше всего пробовать разные запросы и измерять время их выполнения — так вы найдете, какой запрос в ваших условиях обрабатывается быстрее остальных.

Проверяя быстродействие запросов, важно проводить измерения по несколько раз и в дальнейшем учитывать усредненные показатели. Запрос (или хранимая процедура) могут помещаться в соответствующий кэш SQL Server и при первом запуске выполняться дольше, чем при последующих. Кроме того, параллельно с вашим запросом могут обрабатываться другие запросы, выданные к тем же таблицам. Ваш запрос может быть приостановлен, пока другие запросы блокируют нужные таблицы.

Чтобы избавиться от задержек, вызываемых LEFT JOIN, надо соответствующим образом проектировать базу данных. Допустим, какие-то товары относятся к определенным категориям, а какие-то — нет. Если в таблице товаров хранится идентификатор категории, а конкретный товар ни под какую категорию не подпадает, то в соответствующее поле можно было бы записать значение NULL. Тогда, чтобы получить список всех товаров и их категорий, вам пришлось бы выполнить LEFT JOIN. Но можно создать категорию со значением «No Category» и тем самым указать отношение внешнего ключа (foreign key relationship), запрещающее значения NULL. Теперь вы сможете получать те же результаты через INNER JOIN. Хотя такой подход связан с добавлением лишних данных, он позволяет избежать вызовов ресурсоемких LEFT JOIN. Распространив эту концепцию на всю базу данных, вы существенно ускорите обработку запросов. Помните, даже несколько секунд много значат для ваших пользователей, а когда к базе данных одновременно выдается большое количество запросов, эти секунды складываются в минуты.

Декартовы произведения

Здесь я дам совет, идущий в разрез с общепризнанным мнением: в определенных ситуациях декартово произведение (Cartesian product) очень полезно. Почему-то декартовы произведения (CROSS JOIN) пользуются дурной славой, и разработчиков зачастую призывают вообще отказаться от них. Во многих случаях CROSS JOIN действительно приводят к неэффективному использованию ресурсов. Но их надо применять с умом — как и любой другой инструмент SQL. Например, декартово произведение очень удобно в запросе, который возвращает сводку по всем клиентам за каждый месяц, в том числе ничего не заказавшим в каком-либо месяце.

Рассмотрим SQL-код на **рис. 2**. Заметьте: если бы выполнялось стандартное внутреннее объединение (INNER JOIN) таблиц Customers и Orders с группированием по месяцам и суммированием продаж, то возвращались бы данные только по тем месяцам, в которые клиент делал заказы. То есть вы не получили бы нулевое значение для тех месяцев, когда клиент ничего не заказывал. Если вам нужно построить график, отражающий объем продаж по каждому клиенту, то для наглядности на этом графике должны быть нулевые значения по тем месяцам, когда продаж не было. SQL-код на **рис. 2** пропускает месяцы, в которые объем продаж равен нулю, так как в таблице Orders нет соответствующих строк (предполагается, что вы не храните того, чего не было).

Код на **рис. 3** длиннее, зато возвращает данные и по тем месяцам, когда никаких продаж не было. Сначала мы получаем список всех месяцев прошедшего года и помещаем их в первую таблицу с типом данных TABLE (@tblMonths). Затем получаем список всех компаний, что-либо закуповавших в течение этого периода, и помещаем его во вторую таблицу с тем же типом данных (@tblCustomers). В этих таблицах хранятся все данные, необходимые для создания набора результатов (resultset), за исключением реальных цифр, отражающих объемы продаж.

Все месяцы перечислены в первой таблице (12 строк), а все клиенты, что-либо закупившие в этот период, — во второй (в моем случае — 81 компания). Не все клиенты делали покупки в каждом из 12 месяцев, поэтому INNER JOIN или LEFT JOIN пропускал бы тех клиентов, которые в данном месяце ничего не покупали.

Декартово произведение возвращает всех клиентов и по всем месяцам. Фактически первая таблица умножается на вторую, а результатом является набор строк, число которых равно произведению количеств строк в первой и второй таблицах. Таким образом, я получаю таблицу @tblFinal с 972 строками. Далее я обновляю таблицу @tblFinal данными о ежемесячных объемах продаж по каждому клиенту в течение выбранного периода и произвожу выборку конечного набора строк.

CROSS JOIN следует использовать с осторожностью, так как его выполнение требует очень много ресурсов. Например, результат, достигаемый запросом CROSS JOIN, в котором применяются блоки WHERE, DISTINCT или GROUP BY, отфильтровывающие большинство строк, можно получить гораздо более эффективным внутренним объединением (INNER JOIN). Декартовы произведения очень полезны, когда вам нужны совокупные данные, например для построения графика. Но для других задач декартовы произведения луч-

Рис. 2. Выборка по всем клиентам и объемам продаж

```
set nocount on

DECLARE @dtStartDate DATETIME,
        @dtEndDate DATETIME,
        @dtDate DATETIME

SET @dtEndDate = '5/5/1997'

SET @dtEndDate = DATEADD(DD, -1, CAST(CAST((MONTH(@dtEndDate) + 1)
AS VARCHAR(2)) + '/01/' + CAST(YEAR(@dtEndDate) AS VARCHAR(4)) + '
23:59:59' AS DATETIME))
SET @dtStartDate = DATEADD(MM, -1 * 12, @dtEndDate)

SELECT CAST(YEAR(o.OrderDate) AS VARCHAR(4)) + '-' +
CASE
WHEN MONTH(o.OrderDate) < 10
THEN 000 + CAST(MONTH(o.OrderDate) AS VARCHAR(2))
ELSE CAST(MONTH(o.OrderDate) AS VARCHAR(2))
END AS sMonth,
c.CustomerID,
c.CompanyName,
c.ContactName,
SUM(od.Quantity * od.UnitPrice) AS mSales
FROM Customers c
INNER JOIN Orders o ON c.CustomerID = o.CustomerID
INNER JOIN [Order Details] od ON o.OrderID = od.OrderID
WHERE o.OrderDate BETWEEN @dtStartDate AND @dtEndDate
GROUP BY
CAST(YEAR(o.OrderDate) AS VARCHAR(4)) + '-' +
CASE
WHEN MONTH(o.OrderDate) < 10
THEN 000 + CAST(MONTH(o.OrderDate) AS VARCHAR(2))
ELSE CAST(MONTH(o.OrderDate) AS VARCHAR(2))
END,
c.CustomerID,
c.CompanyName,
c.ContactName
ORDER BY
c.CompanyName,
sMonth
```

ше не использовать — в большинстве случаев внутренние объединения гораздо эффективнее.

Всякая всячина

Вот еще несколько приемов, увеличивающих производительность SQL-запросов. Допустим, вы хотите сгруппировать всех продавцов по регионам и суммировать объемы их продаж, причем вам нужны только продавцы, помеченные в базе данных как активные. Вы могли бы группировать продавцов по регионам и использовать HAVING для исключения неактивных продавцов, но то же самое можно сделать с помощью WHERE. Блок WHERE уменьшает число строк, подлежащих группированию, и поэтому он эффективнее, чем HAVING. Фильтрация строк по какому-либо критерию в HAVING приводит к группированию и тех строк, которые были бы исключены в WHERE.

Другой прием, направленный на увеличение производительности, — применение ключевого слова DISTINCT вместо блока GROUP BY в тех случаях, когда нужно найти список различающихся строк (distinct list of data rows). Тогда запросы с ключевым словом DISTINCT будут эффективнее. GROUP BY следует использовать при вычислении функции суммирования вроде SUM, COUNT, MAX и т. д. Кроме того, ключевое слово DISTINCT нежелательно, когда точно известно, что данный запрос будет всегда возвращать уникальные строки. В этой ситуации применение DISTINCT просто излишне.

Рис. 3. Декартово произведение на практике

```

DECLARE @tblMonths TABLE (sMonth VARCHAR(7))
DECLARE @tblCustomers TABLE ( CustomerID CHAR(10),
    CompanyName VARCHAR(50),
    ContactName VARCHAR(50))
DECLARE @tblFinal TABLE ( sMonth VARCHAR(7),
    CustomerID CHAR(10),
    CompanyName VARCHAR(50),
    ContactName VARCHAR(50),
    mSales MONEY)

DECLARE @dtStartDate DATETIME,
    @dtEndDate DATETIME,
    @dtDate DATETIME,
    @i INTEGER

SET @dtEndDate = '5/5/1997'

SET @dtEndDate = DATEADD(DD, -1, CAST(CAST((MONTH(@dtEndDate) + 1) AS
    VARCHAR(2)) + '/01/' + CAST(YEAR(@dtEndDate) AS VARCHAR(4)) + '
    23:59:59' AS DATETIME))
SET @dtStartDate = DATEADD(MM, -1 * 12, @dtEndDate)

-- Заполнить первую таблицу списком всех месяцев
SET @i = 0
WHILE (@i < 12)
BEGIN
    SET @dtDate = DATEADD(mm, -1 * @i, @dtEndDate)
    INSERT INTO @tblMonths SELECT CAST(YEAR(@dtDate) AS VARCHAR(4)) + '-' +
        CASE
            WHEN MONTH(@dtDate) < 10
                THEN '0' + CAST(MONTH(@dtDate) AS VARCHAR(2))
            ELSE CAST(MONTH(@dtDate) AS VARCHAR(2))
        END AS sMonth

    SET @i = @i + 1
END

-- Включить всех клиентов, что-либо закуповавших в этот период, в таблицу 'y'
INSERT INTO @tblCustomers
    SELECT DISTINCT
        c.CustomerID,
        c.CompanyName,
        c.ContactName
    FROM Customers c
        INNER JOIN Orders o ON c.CustomerID = o.CustomerID

```

```

WHERE o.OrderDate BETWEEN @dtStartDate AND @dtEndDate

INSERT INTO @tblFinal
SELECT m.sMonth,
    c.CustomerID,
    c.CompanyName,
    c.ContactName,
    0
FROM @tblMonths m CROSS JOIN @tblCustomers c

UPDATE @tblFinal SET
    mSales = mydata.mSales
FROM @tblFinal f INNER JOIN
    (
        SELECT c.CustomerID,
            CAST(YEAR(o.OrderDate) AS VARCHAR(4)) + '-' +
            CASE WHEN MONTH(o.OrderDate) < 10
                THEN '0' + CAST(MONTH(o.OrderDate) AS VARCHAR(2))
            ELSE CAST(MONTH(o.OrderDate) AS VARCHAR(2))
            END AS sMonth,
            SUM(od.Quantity * od.UnitPrice) AS mSales
        FROM Customers c
            INNER JOIN Orders o ON c.CustomerID = o.CustomerID
            INNER JOIN [Order Details] od ON o.OrderID = od.OrderID
        WHERE o.OrderDate BETWEEN @dtStartDate AND @dtEndDate
        GROUP BY
            c.CustomerID,
            CAST(YEAR(o.OrderDate) AS VARCHAR(4)) + '-' +
            CASE WHEN MONTH(o.OrderDate) < 10
                THEN '0' + CAST(MONTH(o.OrderDate) AS VARCHAR(2))
            ELSE CAST(MONTH(o.OrderDate) AS VARCHAR(2))
            END
    ) mydata on f.CustomerID = mydata.CustomerID AND f.sMonth =
        mydata.sMonth

SELECT f.sMonth,
    f.CustomerID,
    f.CompanyName,
    f.ContactName,
    f.mSales
FROM @tblFinal f
ORDER BY
    f.CompanyName,
    f.sMonth

```

Как видите, для оптимизации запросов и реализации специфических бизнес-правил можно использовать самые разные способы. Весь фокус в том, чтобы проверять их в своих условиях и отбирать подходящие. Самое главное: тестировать, тестировать и еще раз тестировать. В рамках этой рубрики я намерен продолжить исследование концепций SQL Server, проектирования баз данных, способов эффективной индексации и парадигм защиты в SQL Server.

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

Джонни Папа (Johnny Papa) — вице-президент компании MJM Investigations по информационным технологиям (Поли, штат Северная Каролина). Автор книги «Professional ADO 2.5 RDS Programming with ASP 3.0» (Wrox, 2000). Часто выступает на конференциях. С ним можно связаться по адресу datapoints@lancelotweb.com.

Разработка расширяемых приложений Windows Forms

В среде .NET повторно используемые блоки кода — это классы, помещенные в сборки. Благодаря этому и важнейшим особенностям CLR (Common Language Runtime), в том числе наследованию классов, отражению (reflection) и интерфейсам, работа с подключаемыми модулями (plug-in) теперь резко упрощается. В этой колонке я покажу, как разработать MDI-приложение Windows Forms, расширяемое подключаемыми модулями («плагинами»). Такая структура обеспечит высокий уровень адаптируемости этого приложения.

Динамическое расширение реализуется за счет того, что подключаемый модуль распознается приложением при запуске, а впоследствии к нему можно обращаться через меню и панели инструментов. Подключаемый модуль реализует дополнительную для данного приложения функциональность. Если такой модуль установлен и зарегистрирован в системе, он загружается программой так же, как и ее «родные» компоненты. На уровне кода подключаемый модуль в .NET представляет собой скомпилированную сборку с одним или несколькими классами, вызываемых приложением. Пользователь или автор могут в любой момент расширить функциональность приложения, поддерживающего подключаемые модули. А при наличии открытой спецификации подключаемые модули для этого приложения могут быть разработаны третьей стороной.

Поддерживающего подключаемые модули. А при наличии открытой спецификации подключаемые модули для этого приложения могут быть разработаны третьей стороной. На рис. 1 показано, как такая модель работает в различных средах.

При запуске приложение ищет зарегистрированные подключаемые модули. Поиск ведется в конфигурационных файлах, специфичных для конкретной платформы (например, в 16-разрядной Windows просматриваются INI-файлы, а в 32-разрядной — файлы реестра). Эти файлы содержат всю информацию, нужную приложению для настройки обращения к компоненту (через функцию обратного вызова). Если вы используете функции в стиле C, достаточно имен этих функций и их DLL. Прототип функции обратного вызова (callback function) определяет контракт между приложением и внешним подключаемым модулем. При этом приложение полагается на то, что функция из зарегистрированной DLL будет

правильно обращаться со стеком, подготовленным перед вызовом этой функции.

В случае COM приложению нужен лишь программный идентификатор (progID) внешнего COM-объекта. Интерфейсы и IUnknown-метод QueryInterface позволяют тщательнее контролировать прототип вызываемого кода. Как эта модель работает в .NET, показано на рис. 2.

Настройка MDI-приложения

Начнем с создания приложения Windows Forms, которое послужит основой для нашей инфраструктуры подключаемых модулей. Оно будет обрабатывать XCH-документы — XML-файлы с исходными данными для построения диаграмм. XCH-документы создаются простым вводом данных в пустой DataGrid — один из элементов управления Windows Forms. Документы сохраняются в объектах DataSet и сериализуются в XML. При открытии XCH-документа данные копируются в элемент управления DataGrid, и из них «на лету» создается JPEG, отображаемый в элементе управления «графическое окно» (picture box control).

Создавать MDI-приложения в .NET гораздо легче. Во-первых, вы объявляете, что основная форма является MDI-контейнером. Для этого в конструкторе класса установите свойство IsMdiContainer в True — например, в Visual Basic это делается так:

```
Public Sub New()  
    MyBase.New()  
    InitializeComponent()
```

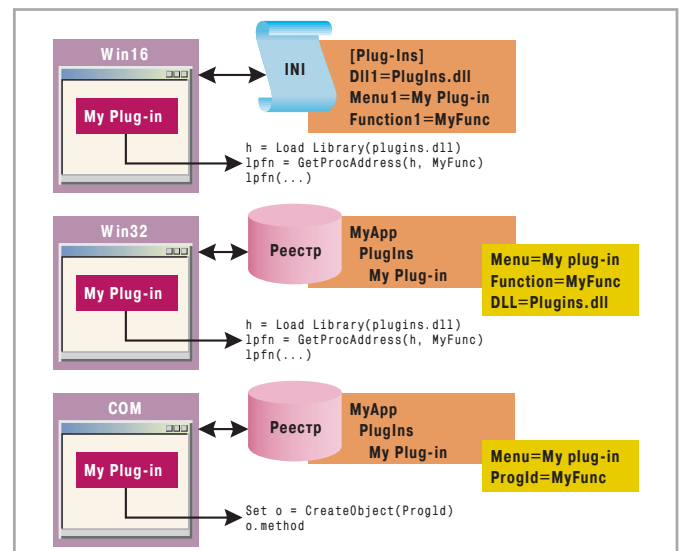


Рис. 1. Подключаемые модули в разных операционных средах

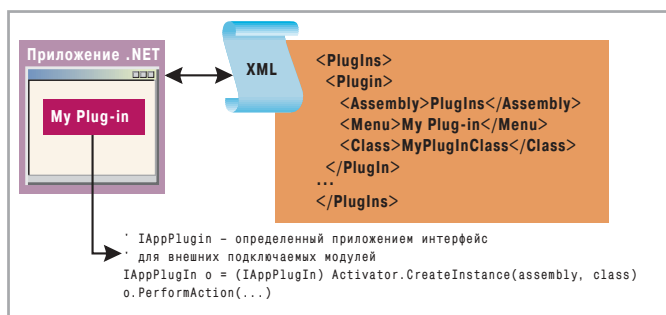


Рис. 2. Подключаемые модули в .NET

```

Me.IsMdiContainer = True
MainForm = Me
End Sub

```

Хотя Visual Basic этого не требует, вам может понадобиться определить внутренний член (в нашем примере — MainForm):

```
Private WithEvents MainForm As Form
```

который представляет саму форму и объявляется с ключевым словом WithEvents. В языке C# для получения событий формы необходимости в таком объявлении нет.

Во-вторых, настройте свойство MdiParent каждого дочернего окна на родительскую форму. Допустим, у вас есть класс дочерних документов XmlChartDocument. Тогда, чтобы созданное окно было MDI-потомком основной формы:

```

Dim doc As XmlChartDocument = New XmlChartDocument()
doc.MdiParent = Me
doc.Show()

```

Кроме того, класс Form реализует массу функций для поддержки MDI. Например, если вы хотите добавить стандартное меню Window для упорядочения окон, напишите:

```

Sub mnuWindowCascade_Click( _
    ByVal sender As Object, _
    ByVal e As EventArgs) _
    Handles mnuWindowCascade.Click
    Me.LayoutMdi(MdiLayout.Cascade)
End Sub

```

Теперь, чтобы упорядочить окна по горизонтали или вертикали, просто выберите подходящий элемент из MdiLayout перечисляемого типа и передайте его методу LayoutMdi класса Form.

Другая полезная функциональность — всплывающее меню, в котором перечисляются все открытые в настоящий момент дочерние документы (рис. 3). Заполнить меню списком MDI-документов нетрудно — достаточно добавить подмену (на иллюстрации это Windows) и установить его свойство MdiList в True. Это можно сделать на этапе проектирования, и тогда весь рутинный код, в том числе отвечающий за активизацию выбранного окна, добавляется автоматически.

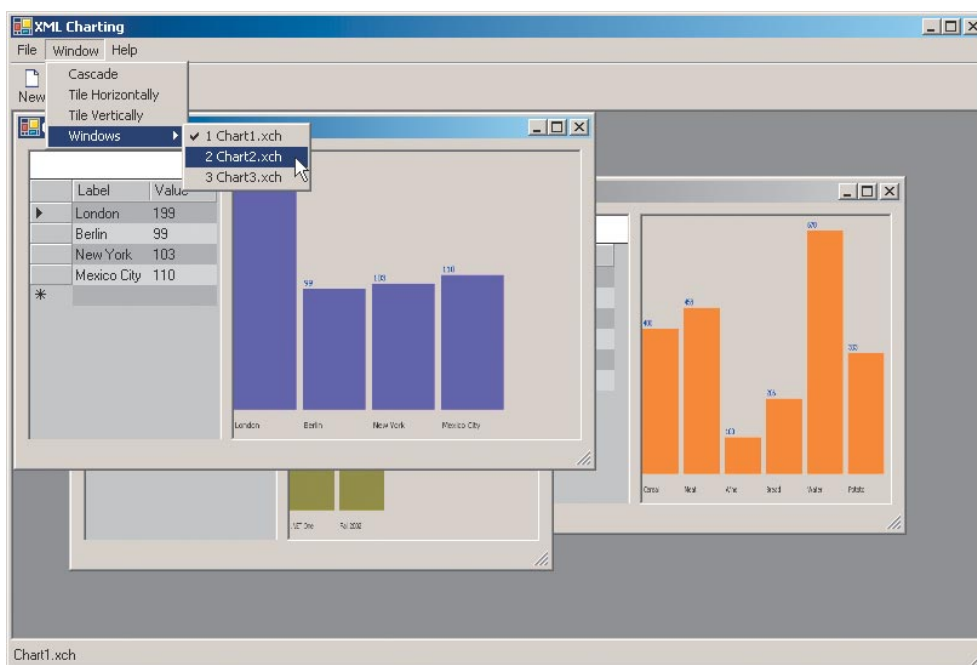


Рис. 3. Все дочерние окна

Поддержка подключаемых модулей

В хорошо спроектированное расширяемое приложение недостающую функциональность можно добавлять в любой момент без перекомпиляции. Создавая соответствующую инфраструктуру, соблюдайте четыре правила:

- определите точки вставки подключаемых компонентов;
- определите программный интерфейс, через который приложение будет взаимодействовать с подключаемыми модулями;
- интегрируйте компоненты подключаемых модулей с пользовательским интерфейсом приложения (меню, панели управления и т. д.);
- настройте приложение и компоненты подключаемых модулей на взаимодействие друг с другом.

Программный интерфейс приложения — набор функций, которые разбиваются на несколько категорий. Каждая категория отражает контекст, в котором должны работать функции этой категории. Стандартные приложения предоставляют статический набор функциональности. Расширяемые приложения, напротив, позволяют динамически добавлять новые функции. Функциональность, реализуемая подключаемым модулем, должна укладываться в одну из категорий, определенных вами в данном приложении.

Одна из категорий функциональности — все операции, которые можно выполнять независимо от того, открыты ли дочерние документы. В эту категорию попадают функции, связанные с элементами меню New и Open. Поэтому подключаемый модуль может содержать собственные функции для создания или открытия документов и другие функции, независимые от документов.

Другая функциональная группа обеспечивает работу с активным документом. К ней относятся такие операции, как Save, Save As и Print. В зависимости от потребностей можно связать тот же модуль с контекстным меню документа. Например, если вы предполагаете добавить команду Send through e-mail, то самое подходящее для нее место — меню File, так как данная функция работает с текущим документом.

Команда Change Chart Color, напротив, специфична для окна документа и будет уместнее в контекстном меню. Другие внешние модули могут быть добавлены к любым базовым стандартным меню вроде Window или Help. Наконец, ничто не мешает предусмотреть возможность создания совершенно новых всплывающих меню. Функциональность, которую буду реализовать я, показана в **табл. 1**.

Точка вставки определяет, как именно приложение должно модифицировать свое меню (и, возможно, панели инструментов) в присутствии зарегистрированного подключаемого модуля. Указанные точки вставки приведены просто для примера, это отнюдь не догма.

А как регистрировать подключаемые модули? Любое расширяемое приложение должно указывать, где следует регистрировать его подключаемые модули, — точно так же, как это делает Windows Explorer для расширений оболочки и Visual Studio для надстроек. Win32- и COM-приложения использовали с этой целью реестр. Но приложение Windows Forms в .NET требует регистрировать подключаемые модули в своем файле .CONFIG или в каком-нибудь XML-файле.

В файле .CONFIG можно создавать новые конфигурационные разделы с нестандартными элементами. По умолчанию поддерживаются только простые параметры в виде пар «имя-значение» и единственный тэг с произвольным числом атрибутов. Для хранения информации со сложной структурой, необходимой подключаемым модулям, этого недостаточно. Здесь есть два выхода. Вы можете отказаться от использования файла .CONFIG и заменить его XML-файлом, структурированным в соответствии с данными. Или встроить XML-данные о конфигурации в стандартный файл .CONFIG, создав для их чтения специальный обработчик конфигурационного раздела (configuration section handler). Такой обработчик — всего лишь .NET-класс, реализующий интерфейс IConfigurationSectionHandler. Очевидно, что атрибуты и дочерние элементы нестандартного раздела зависят от примененного вами обработчика.

```
<configuration>
  <configSections>
    <section name="PlugIns"
      type="YourHandlerClass, assembly" />
  </configSections>
  <PlugIns>
    ...
  </PlugIns>
</configuration>
```

Подробнее об обработчиках конфигурационных разделов см. в документации MSDN. А я использую отдельный XML-файл, помещаемый в тот же каталог, что и исполняемый файл.

Регистрация подключаемых модулей

Как определить схему конфигурационного XML-файла? Дело ваше, но я предпочитаю табличный формат, который позволяет легко загружать данные в набор объектов Data-

Table и управлять ими с использованием простого и привычного синтаксиса. Пример репозитория зарегистрированных подключаемых модулей приведен на **рис. 4**.

А какая информация нужна для регистрации подключаемого модуля? Она зависит от специфики вашего приложения, но некоторые данные все же обязательны. Как минимум, надо указать текст для элемента меню и битовую карту (растровое изображение) для панели инструментов, если подключаемый модуль должен быть доступен и с такой панели. Кроме того, вы должны задать имя класса, реализующего подключаемый модуль, и, конечно, имя его сборки. Наверное, вы захотите определить и точку вставки в меню. В моем примере точки вставки всех модулей фиксированы (это не относится к модулю, который создает всплывающее меню).

На **рис. 5** приведен код, отвечающий за слияние стандартного меню с элементами меню подключаемого модуля в период выполнения. Конфигурационный XML-файл считывается в объект DataSet, при этом каждому типу подключаемого модуля соответствует свой объект DataTable. Каждая строка каждой таблицы связана с подключаемым модулем определенного типа. В столбцах хранится информация, необходимая для работы с подключаемым модулем. Обратите внимание на класс PluggableMenuItem, экземпляр которого создается в коде на **рис. 5**.

```
Public Class PluggableMenuItem
  Inherits MenuItem
  Public AssemblyName, ClassName As String
  Public PlugInType As BWSLib.PlugInType
  Public Sub New(text As String, onClick As EventHandler)
    MyBase.New(text, onClick)
  ...
  End Sub
End Class
```

Класс PluggableMenuItem наследует всю функциональность класса MenuItem, но добавляет к ней специальный конструктор и несколько новых открытых членов. В элементах данных (data members) хранятся тип подключаемого модуля, имена сборки и класса. Класс загружается при выполнении программы, после чего вызывается один из его методов через механизм отражения.

Табл. 1. Некоторые типы подключаемых модулей

Тип	Точка вставки	Описание
FileMenu	За элементом New в меню File	Добавляет в меню File функциональность того же логического уровня, что и New или Open. Контекстная информация не нужна.
DocumentFileMenu	За элементом Save As в меню File	В меню File добавляется новая функция, работающая с активным дочерним окном. В отсутствие дочернего документа блокируется. Контекстная информация описывает окно документа или некоторые члены его класса.
DocumentContextMenu	Добавляется в контекстное меню	Аналог DocumentFileMenu, но новая функция появляется в контекстном меню документа.
WindowMenu	Добавляется в меню Window	Добавляет функциональность в меню Window. Контекстная информация не нужна.
HelpMenu	Добавляется в меню Help	Добавляет функциональность в меню Help. Контекстная информация не нужна.
NewPopupMenu	Вставляется в указанную позицию строки меню	Добавляет нестандартное всплывающее меню. Контекстная информация не нужна.

Рис. 4. Репозиторий подключаемых модулей в XML

```
<?xml version="1.0" encoding="Windows-1252" ?>
<!-- Плагин, зарегистрированный для XML CHART -->
<AppPlugIns>
  <FileMenu>
    <Text>New from Query...</Text>
    <Assembly>ExpowarePlugIns</Assembly>
    <Class>Expoware.NewFromQuery</Class>
  </FileMenu>
  <FileMenu>
    <Text>-</Text>
  </FileMenu>
  <DocumentFileMenu>
    <Text>Send through email</Text>
    <Assembly>ExpowarePlugIns</Assembly>
    <Class>Expoware.SendThroughEmail</Class>
  </DocumentFileMenu>
  <DocumentContextMenu>
    <CheckItem>True</CheckItem>
    <Text>View as a Pie Chart</Text>
    <Assembly>AnotherCompanyPlugIns</Assembly>
    <Class>AnotherCompany.PieChart</Class>
  </DocumentContextMenu>
  <WindowMenu>
    <Text>-</Text>
  </WindowMenu>
  ...
</AppPlugIns>
```

Так как событие `OnClick` поддерживается классом `MenuItem`, в двух элементах данных содержится информация, необходимая для того, чтобы при щелчке элемента меню подключаемого модуля обработчик мог легко получить имя сборки и класса:

```
Private Sub StdOnClickHandler(sender As Object, e As EventArgs)
    Dim mnuItem As PluggableMenuItem
    mnuItem = CType(sender, PluggableMenuItem)
    MessageBox.Show(mnuItem.ClassName & _
        ", " & mnuItem.AssemblyName)
End Sub
```

При запуске приложение просматривает конфигурационный XML-файл, обнаруживает зарегистрированные подключаемые модули и обновляет меню, добавляя к нему новые элементы. Кроме того, слияние меню можно реализовать с помощью специальных методов и свойств класса `Form`. В частности, методы `MergeMenu` и `MergeType` в сочетании со свойством `MergeOrder` позволяют объединить два меню.

Базовый класс для подключаемых модулей

Подключаемый модуль — это экземпляр класса .NET, динамически загружаемый из указанной сборки. Такой класс должен соответствовать определенной структуре, обеспечивающей работу функций обратного вызова. Обычно для этого реализуется определенный интерфейс в классе подключаемого модуля. В данном случае я определяю интерфейс для каждого поддерживаемого типа подключаемого модуля. Впрочем, это не всегда обязательно: у подключаемых модулей всех типов может быть один интерфейс. Но, по моему, вы ждете от меня описания наиболее универсального подхода. Исходя из интерфейса, вы проектируете базовый класс для подключаемого модуля:

```
Interface IAppPlugIn
    Sub Invoke(...)
End Sub
End Interface
Public Class AppPlugIn
```

```
    Implements IAppPlugIn
    Public Overridable Sub PerformAction(...)
        Implements IAppPlugIn.Invoke
    ...
End Sub
End Class
```

Интерфейс и базовые классы должны находиться в отдельной сборке, с которой будут связываться как подключаемые модули, так и главное приложение. Эта сборка создает механизм взаимодействия, благодаря которому приложение становится расширяемым. Интерфейсы, которые я намерен задействовать в своем приложении-примере, показаны на **рис. 6**. Если методы интерфейса принимают параметры базовых типов .NET, тогда все отлично. В ином случае вам придется определить собственные типы в той же интерфейсной сборке. (Вы могли бы использовать еще одну отдельную сборку, на которую ссылается интерфейсная, но оставлять собственные типы неопределенными нельзя.)

Рис. 5. Слияние меню

```
Private Sub SetupMenu()
    Dim mnuPopup, mnuItem As MenuItem

    ' Обращаемся к конфигурационному файлу меню
    Dim ds As DataSet = New DataSet()
    If Not File.Exists(CONFIGFILE) Then
        Return
    End If
    ds.ReadXml(CONFIGFILE)

    ' Динамически добавляем элементы в существующие
    ' всплывающие меню
    AddFileMenuPlugIns(ds)
    AddDocumentFileMenuPlugIns(ds)
    AddWindowMenuPlugIns(ds)
    AddHelpMenuPlugIns(ds)

    ' Добавляем новые всплывающие меню
    AddNewPopupMenuPlugIns(ds)
End Sub

' Все процедуры AddXXXPlugIns имеют сходную структуру

Private Sub AddFileMenuPlugIns(ByVal ds As DataSet)
    ModifyAppMenu(ds, "MenuFile", mnuFileNew.Index + 1, mnuPopupFile)
End Sub

Private Sub ModifyAppMenu(ByVal pluginType As PlugInType, _
    ByVal tableName As String, _
    ByVal menuIndex As Integer, _
    ByVal mnuInsertPoint As MenuItem)
    Dim mnuPopup, mnuItem As PluggableMenuItem
    Dim config As DataTable
    Dim menuExt As DataRow

    config = ds.Tables(tableName)
    If config Is Nothing Then
        Return
    End If

    Dim index As Integer = menuIndex
    For Each menuExt In config.Rows
        mnuItem = New PluggableMenuItem(menuExt("Text"), _
            AddressOf StdOnClickHandler)
        mnuItem.AssemblyName = menuExt("Assembly")
        mnuItem.ClassName = menuExt("Class")
        mnuItem.PlugInType = pluginType

        mnuInsertPoint.MenuItems.Add(index, mnuItem)
        index += 1
    Next
End Sub
```

Рис. 6. Интерфейсы

```
' Интерфейс для компонентов, подключаемых к FileMenu
Public Interface IXmlChartFileMenuPlugIn
    Sub PerformAction(ByVal f As BWSLib.XmlChartMainForm)
End Interface

' Интерфейс для компонентов, подключаемых к DocumentFileMenu
Public Interface IXmlChartDocumentFileMenuPlugIn
    Sub PerformAction(ByVal docWindow As XmlChartDocument)
End Interface

' Интерфейс для компонентов, подключаемых к DocumentContextMenu
Public Interface IXmlChartDocumentContextMenuPlugIn
    Sub PerformAction(ByVal docWindow As XmlChartDocument)
End Interface

' Интерфейс для компонентов, подключаемых к WindowMenu
Public Interface IXmlChartWindowMenuPlugIn
    Sub PerformAction(ByVal f As BWSLib.XmlChartMainForm)
End Interface

' Интерфейс для компонентов, подключаемых к HelpMenu
Public Interface IXmlChartHelpMenuPlugIn
    Sub PerformAction(ByVal f As BWSLib.XmlChartMainForm)
End Interface

' Интерфейс для компонентов, подключаемых к NewPopuMenu
Public Interface IXmlChartNewPopuMenuPlugIn
    Sub PerformAction(ByVal f As BWSLib.XmlChartMainForm)
End Interface
```

Теперь рассмотрим несколько практических вариантов. Допустим, вы пишете подключаемый модуль, принимающий ссылку на активное окно документа, — например, уже упоминавшийся класс `XmlChartDocument`. Так как этот класс является пользовательским типом и на него ссылаются интерфейсы подключаемого модуля, его следует определить в отдельной интерфейсной сборке, а не в главном приложении. Не очень изящно, но работать будет. А как быть, если понадобится написать подключаемый модуль другого типа, которому тоже нужна ссылка на основную форму приложения? Значит, и этот класс придется поместить в сборку с интерфейсами. В итоге вы получите приложение Windows Forms, состоящее из крошечного EXE-файла и огромной сборки. Если вас это устраивает — вперед.

Однако гораздо лучше и элегантнее использовать промежуточные (возможно, абстрактные) классы. В интерфейсной сборке определяется надкласс для типов, которые вы действительно собираетесь применять. В этом надклассе содержатся те же определения, что и в истинном классе, но они или пусты, или реализованы по минимуму. Таким образом, вместо `System.Windows.Forms.Form` предком главной формы является, скажем, класс `XmlChartMainForm`, в свою очередь наследующий от `Form`. Кроме того, в классе `XmlChartMainForm` определяются все нужные вам дополнительные методы и свойства. На **рис. 7** показана инфраструктура, благодаря которой третьи стороны могут создавать подключаемые модули, а приложения — обращаться к этим модулям (вызов одного из подключаемых модулей демонстрирует **рис. 8**).

Механизм взаимодействия включает три уровня: сборку приложения, интерфейсную

сборку и подключаемый модуль, который тоже является сборкой. Интерфейсная сборка должна считаться неизменяемой, так как на нее ссылаются из двух других уровней. Любое изменение в ней может нарушить работу подключаемых модулей.

Вместо того чтобы определять абстрактный класс и в сборке приложения создавать производные от него классы, я реализовал класс документа (`XmlChartDocument`) в интерфейсной сборке. Но для класса основной формы я предпочел первый подход. Класс `MainApp`, который реализует форму — основное окно для пользовательского интерфейса приложения, наследует от класса `XmlChartMainForm`. Последний просто добавляет новый открытый метод (`OpenDocumentFromFile`) к стандартному интерфейсу класса `Form`. Я определил два собственных типа разными способами только ради того, чтобы продемонстрировать оба подхода. С точки зрения проектирования, подход на основе промежуточных абстрактных классов эффективнее и изящнее.

В интерфейсной сборке надо также определить интерфейс для каждого поддерживаемого подключаемого модуля и базовый класс, реализующий его. Вот как создать, например, экземпляр класса, реализующего модуль, подключаемый к `FileMenu`:

```
Dim a As String = mnuItem.AssemblyName
Dim c As String = mnuItem.ClassName
Dim o As IXmlChartFileMenuPlugIn
o = CType(Activator.CreateInstance(a, c).Unwrap(), _
    IXmlChartFileMenuPlugIn)
o.PerformAction(Me)
```

О глобальном объекте `Activator` — чуть позже.

Обычно класс, реализующий интерфейсы, используется как базовый при создании компонента подключаемого модуля. Обратите внимание: в нашем случае интерфейсы и реализующие их классы взаимозаменяемы. Конечно, интерфейс — не класс, но для компонентов подключаемых модулей интерфейс является уникальным и неизменяемым механизмом взаимодействия. Хотя класс подключаемого модуля предоставляет больше методов и свойств, чем определено в интерфейсе, они никогда не будут вызываться все. Вот почему вы можете смело приводить экземпляр объекта, полученный от объекта `Activator`, к типу интерфейса.

Создание и вызов подключаемых модулей

Следующий фрагмент кода иллюстрирует, как создать компонент, который встраивается в меню `File` и автоматически становится доступным при наличии хотя бы одного открытого документа:

```
using System;
using BWSLib;
using System.Windows.Forms;
```

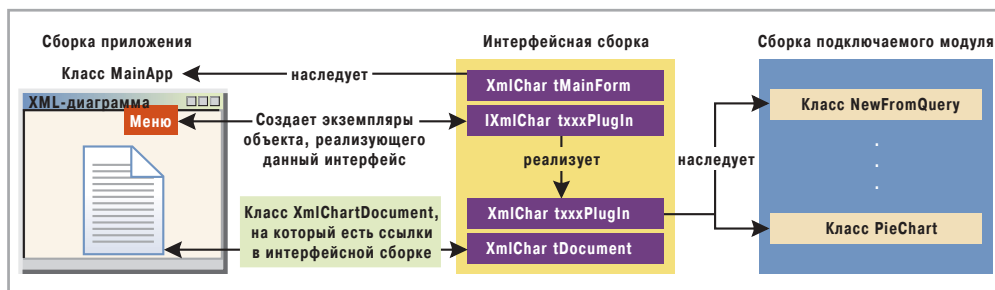


Рис. 7. Инфраструктура для подключаемых модулей

```

namespace Expoware
{
    public class SendThroughEmail : XmlChartDocumentFileMenuPlugIn
    {
        public override void PerformAction(XmlChartDocument docWindow)
        {
            MessageBox.Show("Sending \"" + docWindow.FileName + "\" _
            & " through email...", "Send through email");
        }
    }
}

```

Чтобы создать подключаемые модули других типов, просто измените базовый класс. Заметьте, что приложение само блокирует и разблокирует доступ к подключаемым модулям в зависимости от текущего контекста.

Структура подключаемого модуля очень проста. В принципе, от вас требуется лишь переопределить метод PerformAction для конкретного базового класса. Если функциональность подключаемого модуля требует обращений к другим классам, создания форм или доступа к базам данных, включите все необходимые ссылки и код в класс подключаемого модуля. Затем скомпилируйте подключаемый модуль в DLL-сборку и поместите туда, где ее сможет найти главное приложение. Сборки обычно устанавливаются как закрытые или как глобальные. Закрытые сборки предназначены только для конкретного приложения и поэтому помещаются в корневой каталог этого приложения. Глобальные, или общие, сборки могут использоваться несколькими приложениями, и с их установкой возникают некоторые трудности. Они должны храниться в GAC (Global Assembly Cache), и, кроме того, у них должно быть строгое имя (strong name). Подключаемые модули, как правило, специфичны для конкретного приложения, так что вы скорее всего будете устанавливать их как закрытые сборки.

Вы уже видели фрагмент XML-файла, используемого для хранения информации о зарегистрированных подключаемых модулях (рис. 4). В этом файле вы должны указать текст для элемента меню, имя сборки и имя истинного класса. Если вы хотите добавить в меню разделитель, используйте дефис (-) в текстовом элементе.

Итак, как же загружаются и вызываются подключаемые модули? Подключаемый модуль — это класс, но приложение ничего о нем не знает, пока не просмотрит конфигурационный XML-файл. После этого оно загружает сборку, содержащую класс, и создает экземпляр этого класса. Учтите, что имя класса подключаемого модуля нельзя использовать как ссылочный тип, так как по контракту оно им не является.

Чтобы обойти проблему позднего связывания, .NET предоставляет глобальный объект Activator, позволяющий создавать экземпляры классов в тех случаях, когда вам известно лишь имя сборки, имя класса и один из реализованных интерфейсов. Объект Activator может создать локальный или удаленный экземпляр типа, используя конструктор класса, очень похожий на конструктор по умолчанию. Объект Activator, часто применяемый для создания удаленных объектов, не снижает и не повышает производительность программы в сравнении с оператором New. Однако именно он предоставляет уникальную возможность создавать сильно типизированные объекты на основе слабо типизированной информации.

Рис. 8. Вызов класса подключаемого модуля

```

Private Sub StdOnClickHandler(ByVal sender As Object, _
    ByVal e As EventArgs)

    Dim mnuItem As PluggableMenuItem
    mnuItem = CType(sender, PluggableMenuItem)
    Dim a As String = mnuItem.AssemblyName
    Dim c As String = mnuItem.ClassName

    ' Получить класс требуемого подключаемого модуля
    ' и вызвать метод PerformAction
    Select Case mnuItem.PlugInType
        Case PlugInType.FileMenu
            Dim o As IXmlChartFileMenuPlugIn
            o = CType(Activator.CreateInstance(a, c).Unwrap(), _
                IXmlChartFileMenuPlugIn)
            o.PerformAction(Me)
            ...
            o.PerformAction(Me.ActiveMdiChild)
            ' Если элемент меню можно помечать галочкой,
            ' поставить галочку!
            If mnuItem.CheckItem Then
                mnuItem.Checked = Not mnuItem.Checked
            End If
        Case PlugInType.HelpMenu
            Dim o As IXmlChartHelpMenuPlugIn
            o = CType(Activator.CreateInstance(a, c).Unwrap(), _
                IXmlChartHelpMenuPlugIn)
            o.PerformAction(Me)
            ...
    End Select
End Sub

```

Код, в котором создается и вызывается экземпляр класса подключаемого модуля, был показан на рис. 8. Процедура StdOnClickHandler связывается со всеми элементами меню, которые формируются подключаемыми модулями. Сведения о сборке и классе она получает от класса PluggableMenuItem. Остальное возлагается на Activator. Обратите внимание на применение метода Unwrap. У метода CreateInstance объекта Activator есть несколько перегруженных версий, которые возвращают либо тип Object, либо тип ObjectHandle. В частности, все версии, принимающие имя сборки, всегда возвращают тип ObjectHandle. Это контейнер, который служит оболочкой для созданного экземпляра объекта. Его задача — обеспечить маршалинг экземпляров объекта по значению через AppDomains в целевые домены, не копируя сборку, в которой определен данный тип. Так вот, Unwrap открывает доступ к самому объекту.

Примеры подключаемых модулей

Несколько таких модулей включены в исходный код, который можно скачать по адресу <http://download.microsoft.com/download/msdnmagazine/code/July02/WXP/EN-US/CuttingEdge0207.exe>. Подключаемый модуль NewFromQuery вызывается из меню File и позволяет создать документ на основе результатов SQL-запроса. Он отображает форму, выдающую SQL-запрос к базе данных Northwind, и помещает результат в таблицу. Щелкнув кнопку Save, Вы можете сохранить нижележащий DataSet в XCH-файл.

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

Дино Эспозито (Dino Esposito) — консультант и преподаватель из Рима. Автор книги «Building Web Solutions with ASP.NET and ADO.NET» (Microsoft Press, 2002). Теперь в основном занимается преподаванием технологий ASP.NET и ADO.NET в компании Wintellect (<http://www.wintellect.com>). С Дино можно связаться по адресу dinoe@wintellect.com.

Rotor

Shared Source CLI предоставляет исходный код для FreeBSD-реализации .NET

Эта статья предполагает знание CLR и C++

Уровень сложности: ① ② ③

АННОТАЦИЯ Текущий дистрибутив Shared Source CLI включает две реализации: для Win32 и FreeBSD. Внимательно изучая Shared Source CLI (а это более 9000 файлов, в том числе около 1300 открытых классов), можно многое узнать о внутреннем устройстве CLR. Но исследовать такой объем исходного кода — задача неподъемная. В статье обсуждается лишь часть того, о чем можно узнать из исходного кода CLR, — например, как реализуется JIT-компиляция. Автор помогает понять, как контролировать выполнение кода и что происходит при отладке и загрузке классов. Не менее полезно и изучение операций, необходимых для настройки исполняющей среды CLR (CLR runtime).

Статьи на смежную тематику:

<http://msdn.microsoft.com/msdnmag/issues/1100/gci/gci.asp>,

<http://msdn.microsoft.com/msdnmag/issues/02/02/PE/PE.asp> и

<http://www.microsoft.com/msj/0199/bugslayer/bugslayer0199.htm>.

Базовая информация, необходимая для понимания этой статьи:

<http://msdn.microsoft.com/net/sscli>, <http://msdn.microsoft.com/net/ecma>,

http://developer.intel.com/design/intarch/intel386/docs_386.htm и

<http://discuss.develop.com>.

Будучи болезненно любопытным, я люблю подсматривать, как работают технологии, когда они думают, что их никто не видит. Как часто говорит один мой приятель, эффективность программиста прямо пропорциональна тому, насколько хорошо он понимает нижележащие уровни технологий. ATL-программисту не уйти далеко без понимания CLSID и QueryInterface, а MFC-программист должен сначала помучиться еще и с WNDPROC. Я уж не говорю о том, каких усилий стоит эффективная работа с драйверами устройств.

При программировании в этих средах вы должны помнить, что заклинания, которым вы так долго учились, могут дать совсем не тот результат. Это все из-за того, что Windows (как и любые другие операционные системы) просто загружает EXE-модуль и очертя голову бросается его исполнять; ей глубоко безразлично, какой код вы запускаете и вообще правилен ли он. Она целиком полагается на то, что ваш компилятор уже проделал всю трудную работу и сгенерировал код, который худо-бедно, но работать будет.

.NET кажется куда проще: пишешь исходные тексты, они включаются в сборки в виде IL-кода и метаданных, а об основном заботится сама исполняющая среда, пока вы попиваете pina coladas*, пользуясь выдавшимся свободным временем. Эта кажущаяся простота — на самом деле хорошо поставленный фокус, который посрамил бы самого Волшебника Изумрудного города. Иллюзия поддерживается общезыковой исполняющей средой (common language runtime, CLR),

* Коктейль из рома с ананасовым соком. — Прим. перев.

которая за кулисами просто разрывается: перемалывает код и выполняет его, жонглирует подсистемами управления памятью и защитой — причем делает все это одновременно.

Именно потому, что CLR выполняет столько дополнительной работы, она превращается в среду управляемого кода. Вместо того чтобы загрузить код в процесс и забыть о нем, CLR обращается с кодом как с чем-то текучим и динамичным. Код загружается в промежуточном формате, например:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    Idstr    "Hello, World!"
    call    void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

Ассемблерный код, показанный на **рис. 1**, не генерируется до тех пор, пока он действительно не понадобится. Дело в том, что у исполняющей среды куда более интимные отношения с вашим кодом, чем те, к которым вы, вероятно, привыкли. Понимание того, как переплетается ваш код с кодом самой CLR, в конечном счете поможет вам писать более эффективные программы.

CLI

Набор спецификаций, известных под общим названием Common Language Infrastructure (CLI) (<http://msdn.microsoft.com/net/ecma>), — это Библия управляемого кода. Как и многие священные писания, CLI имеет огромный объем и определяет буквально все — от набора инструкций промежуточного языка (intermediate language, IL) до формата сборок и метаданных .NET. Если вы работаете с .NET, вам просто необходимо ознакомиться с тем, что изложено в CLI, так как сама Micro-

soft создает минимум еще одну реализацию (.NET Compact Framework для Windows CE), да и другие группы работают над собственными реализациями.

У CLI много общего с другими священными писаниями: она тоже немного скучна и трудна для понимания. Спецификации — вещь важная, но, на мой взгляд, от них не получишь того же удовольствия, как

от пошагового исследования настоящего кода в отладчике. Я, конечно, не могу запустить руки в исходный код CLR, зато у меня есть нечто, почти столь же полезное: реализация Shared Source CLI, которая в самой Microsoft, как вы, возможно, слышали, проходит под кодовым названием «Rotor».

Мешок сладостей

Впервые услышав о Shared Source CLI, я подумал, что это, наверное, какая-нибудь урезанная реализация, выпущенная Microsoft только для того, чтобы отделаться от назойливой критики, но наконец-то раздобыв ее, я был просто поражен. Shared Source CLI вовсе не слеплена на скорую руку. Кодовая база Shared Source CLI является подмножеством исходного кода CLR и в наиболее важных частях практически

Рис. 1. Ассемблерный x86-код для Hello.Main

```
//Пролог функции
push    ebp
mov     ebp,esp
push    esi
xor     esi,esi
push    esi
push    ecx
push    edx
//Обнулить фрейм стека
mov     ecx,1
addr_a:
push    0
loop    addr_a
//Загрузить "Hello, World" и вызвать Console.WriteLine
mov     eax, <элемент из таблицы строковых констант>
mov     eax,dword ptr [eax]
mov     ecx,eax
push    ecx
mov     eax, &Console.WriteLine
call    dword ptr [eax]
add     esp,4
//Проверка: вызов check_stack в fjitdef.h
push    esp
push    ebp
mov     eax,14h
push    eax
mov     eax, &check_stack
call    eax
//Эпилог функции
add     esp,0Ch
mov     esi,dword ptr [ebp-4]
mov     esp,ebp
pop     ebp
ret
```

идентична ему. JIT-компилятор и механизм сбора мусора проще, чем в коммерческом продукте, но все равно Shared Source CLI — вещь серьезная. Ядро исполняющей подсистемы (execution engine) поддерживает все известные особенности CLR, в том числе механизм Remoting, модель контекстов, защиту доступа к коду и т. д. Кодовая база состоит из более чем 9000 исходных файлов. Так что Shared Source CLI — крупнейшая кодовая база из числа когда-либо открывавшихся Microsoft для всеобщего пользования. Для сравнения: в MFC всего 13 Мб кода, размазанного по 300 (или около того) файлам, а в ATL (моем прежнем фаворите) и того меньше — около 1 Мб.

Большая часть этой функциональности определяется спецификациями CLI, и я не слишком удивился ей. Но я отпал, увидев, насколько богата окружающая ее инфраструктура. Кроме ядра исполняющей подсистемы, там есть различающий версии загрузчик сборок (version-aware assembly loader) (в просторечии — «Fusion»), применяемый в самой CLR, кэш глобальных сборок (global assembly cache), физически отделенный от CLR Global Assembly Cache (GAC), и полный комплект собственных файлов .config, включая security.config и machine.config.

Единственное, чего недостает в ядре исполняющей подсистемы, — специфических для Windows технологий вроде COM Interop и поддержки сборок смешанного режима, которые содержат машинный x86-код, известный под названием «It just works» (IJW) («он просто работает»). Они полезны в Windows, но вряд ли кому нужны вне этой среды — их присутствие только усложнило бы и без того непростой код.

Заполучить исходный код полноценного ядра исполняющей подсистемы — это здорово, но с одним ядром было бы трудно сделать что-нибудь интересное. Помимо самого яд-

ра, в Shared Source CLI содержится исходный код практически всех инструментов .NET, поставляемых с CLR (некоторых из них нет даже в самой CLR!), в том числе полнофункциональных компиляторов C# и JScript, а также переносимых (портируемых) версий всех известных вам средств .NET — ILASM, ILDASM, cordbg.exe, sn.exe, gacutil.exe, al.exe, caspol.exe, peverify.exe и др. Shared Source CLI также включает исходный код множества инструментов отладки и средств диагностики.

Даже если вы никогда не запустите ни одного исполняемого файла в Shared Source CLI, их исходный код существенно упростит понимание многих базовых процессов в CLR, например подпись, верификацию и загрузку сборок. Он же станет и отправной точкой для создания массы новых инструментов вроде отладчиков и средств профилирования (profilers). Надеюсь, что очень скоро они появятся.

Библиотеки кода и Shared Source CLI

Теоретически ядра исполняющей подсистемы и сопутствующих инструментов вполне достаточно, но на практике Shared Source CLI была бы почти бесполезна без библиотек классов. В спецификациях CLI учтен тот факт, что разные платформы предъявляют разные требования к ресурсам, и поэтому в них описываются два стандартных профиля реализации. Разработчики, желающие создавать реализации с абсолютным минимумом функциональности, могут придерживаться Kernel Profile. Этот профиль минималистичен до крайности — в нем нет даже поддержки математических операций с плавающей точкой. Kernel Profile настолько скромен, что все известные мне реализации CLI опираются на более функциональный профиль — Compact Profile.

Compact Profile предоставляет дополнительную функциональность, например XML, поддержку сетей и пространств имен отражения (reflection namespaces); от этой платформы уже есть какой-то толк. Два стандартных профиля и их взаимосвязи (описанные в спецификациях CLI, Partition IV) показаны на рис. 2.

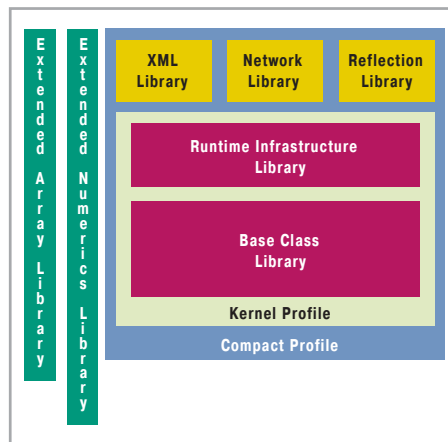


Рис. 2. Профили CLI

Любая функциональность, выходящая за рамки Compact Profile, — забота того, кто создает свою реализацию. Естественно, CLR намного функциональнее Compact Profile и предоставляет куда больше классов, чем того требует CLI; кроме того, CLR предусматривает поддержку для Windows Forms, ADO.NET и ASP.NET.

Реализация Shared Source CLI так далеко не заходит. Она довольно близка к Compact Profile, предоставляя чуть больше функциональности. Shared Source CLI поставляется с исходным кодом пяти сборок: Mscorlib.dll, System.dll, System.xml.dll, System.runtime.remoting.dll и System.runtime.serialization.formatters.soap.dll.

В этих библиотеках нет всех классов, которые содержатся в соответствующих CLR-версиях, тем не менее в них есть все самое необходимое, а это примерно 1300 открытых классов. Различия между Shared Source CLI и коммерческой CLR сведены в табл. 1.

Поскольку Shared Source CLI — своего рода демонстрация CLR, а не коммерческий продукт, вы не найдете здесь ни Windows Forms, ни ADO.NET, ни Web-сервисы. Я пытался загрузить некоторые из этих библиотек в среде Shared Source CLI, но все они полагаются на классы, которых нет в текущей версии дистрибутива Shared Source CLI. По-моему, как раз это поправимо — я с оптимизмом смотрю на то, что ADO.NET и Windows Forms, вероятно, удастся заставить работать в текущей реализации Shared Source CLI, если ее дополнить нужными классами.

Shared Source CLI, Windows и FreeBSD

Спецификация CLI вовсе не обязывает управляемый код работать только в Windows. Чтобы доказать это, Microsoft предусмотрела возможность компиляции и запуска Shared Source CLI на платформах FreeBSD Unix и Windows XP. Поскольку Shared Source CLI производна от CLR, а та опирается на Win32, нужен какой-то уровень для абстрагирования основной кодовой базы от специфики конкретной ОС. Но, вместо того чтобы изобретать еще один уровень абстракции, в коде Shared Source CLI делается ставка на некое подмножество

Win32. Соответствующий набор функций, называемый Platform Abstraction Layer (PAL), образует фундамент практически для всего, что есть в Shared Source CLI.

PAL реализован в виде пары библиотек (rotor_pal.dll и rotor_palrt.dll), которые совместно экспортируют около 500 функций. Эти библиотеки — единственные во всем дистрибутиве Shared Source CLI, в которых учитывается специфика нижележащей ОС. Все

Табл. 1. Библиотеки классов Shared Source CLI

Библиотека	Что отсутствует	Открытые классы	Отсутствующие открытые классы
System.dll	Большая часть Microsoft.Win32 Большая часть System.ComponentModel Большая часть System.Diagnostics Классы System.IO System.Timers	237	274
System.runtime.remoting.dll	HttpRemotingHandler	25	3
System.xml.dll	Открытые классы не изменены	212	0
Mscorlib.dll	Microsoft.Win32.Registry Многое из System.Runtime.InteropServices System.Security.Cryptography	867	188
System.runtime.Serialization.formatters.Soap.dll	Ничего	2	0

остальное (в том числе компиляторы, инструментальные средства и исполняющая среда) размещается поверх этих двух DLL (рис. 3).

PAL полностью определен в заголовочном файле `rotor_pal.h` и документирован в `pal_guide.html`, который находится в каталоге `docs\techinfo`. Текущий дистрибутив Shared Source CLI включает две реализации: для Win32 и FreeBSD; они помещаются в каталог `pal`. Как вы понимаете, Win32 PAL куда проще FreeBSD PAL — файлы его исходного кода занимают около 1 Мб и по большей части представляют собой тонкие оболочки вокруг Win32 и исполняющей среды.

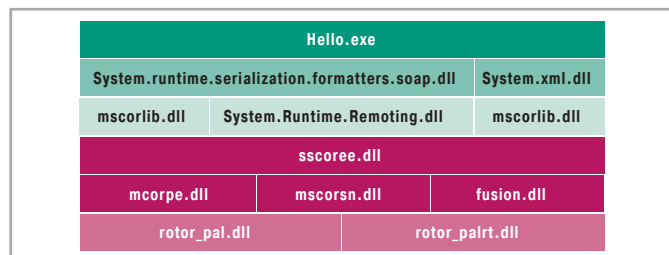


Рис. 3. Зависимости в Shared Source CLI

FreeBSD PAL значительно сложнее, так как он увязывает Win32-функции с FreeBSD. Некоторые из этих функций (вроде `lstrcpu`) портируются сравнительно легко, но остальные (например, `CreateFileMapping`) заставляют PAL проделывать основательную работу. В результате файлы исходного кода FreeBSD PAL занимают 2,5 Мб и весьма интересны для чтения.

Однако перенос Shared Source CLI на новую платформу — это не просто перенос PAL. Почему? Из-за JIT-компиляции. JIT-компиляторы по самой своей природе тесно связаны с конкретным семейством процессоров и спецификой их поддержки в данной ОС. JIT-компилятор в Shared Source CLI обходит эту проблему, используя весьма оригинальную систему макросов, которые при компиляции на процессорных платформах Intel генерируют машинный x86-код, а в любых других случаях раскрываются в набор функций ANSI C. На момент написания статьи Shared Source CLI еще не работала на процессорных архитектурах, отличных от Intel, например на PowerPC, но думаю, что такое положение вещей долго не продлится.

Архитектура исполняющей среды

У исполняющей среды Shared Source CLI три основные задачи: загрузка/выполнение кода, управление памятью и управление защитой. Вроде бы ничего необычного, но на практике каждая из задач крайне трудна. И они еще больше усложняются тем, что соответствующие подсистемы активно взаимодействуют между собой и поэтому сильно влияют друг на друга. Информация, добываемая от JIT-компилятора, помогает сборщику мусора в его охоте на неиспользуемые объекты. В свою очередь объем свободной памяти может влиять на то, как и когда компилируется код, а защита и подавно должна быть всепроникающей. Но многие, даже более сложные, механизмы CLR в Shared Source CLI упрощены, что облегчает изучение их взаимосвязей.

Объем кода все равно огромен: реализация базовой исполняющей среды по своим масштабам сравнима со всей инфраструктурой MFC 4.2. К счастью, самая интересная часть ко-

да сосредоточена вокруг нескольких центральных классов. Код исполняющей среды находится в каталоге `sscli\clr\src\vm`, а код JIT-компилятора (по размеру и сложности примерно соответствующий ATL) живет отдельно — в каталоге `sscli\clr\src\fjit`. Сердцевина исполняющей среды состоит из следующих трех классов.

- **ClassLoader** (`clsload.h`) управляет загрузкой IL-кода и метаданных и распоряжается тем, когда и как запускать JIT-компилятор. Основная часть работы ClassLoader связана с управлением экземплярами `EEClass` (загляните в `class.h` и `class.cpp`).
- **Fjit** (`fjit.h`) — класс, который считывает IL-код и генерирует машинный код для процессоров типа x86. Код, отвечающий за саму JIT-компиляцию, вынесен в отдельный каталог `sscli\clr\src\fjit`. (Подозреваю, что имя Fjit образовалось от «Fast JIT», так как именно это название мелькало в ранних бета-версиях CLR.)
- **GCHeap** (`gc.h`) заведует управлением памятью и сборкой мусора. GCHeap управляет выделением и очисткой памяти, а также контролирует Finalization.

Высокоуровневые взаимосвязи этих классов показаны на рис. 4. Рассказать обо всех трех классах в одной статье невозможно, поэтому основное внимание я уделю трансформации метаданных в выполняемый код — об остальном поговорим в другой раз.

Приступая к работе с Shared Source CLI

Сначала вы должны скачать кодовую базу Shared Source CLI с официальной страницы Shared Source CLI (<http://msdn.microsoft.com/net/sscli>). Дистрибутив представляет собой сжатый TAR-файл размером 10 Мб; для его разархивирования используйте WinZip. После этого настройте среду сборки Shared Source CLI. Просто открыть проект Visual Studio и нажать клавиши `Ctrl+Shift+B` не получится.

Первый шаг в Windows или FreeBSD — настройка переменных окружения, используемых в процессе сборки. Для этого достаточно запустить файл `env.bat` из корневого каталога установленной кодовой базы. (Unix-программисты могут использовать в зависимости от оболочки либо `env.sh`, либо `env.csh`.) `Env.bat` выполняет в основном две задачи: настраивает переменные окружения, контролирующие процесс сборки, и задает пути для большего удобства работы с такими инструментами, как `csc.exe` или `sn.exe`.

При сборке кодовой базы Shared Source CLI можно указать, нужны ли проверки в исполняющей среде, и выбрать степень оптимизации кода через необязательный параметр `env.bat`.

`Env.bat` позволяет задать тип сборки с помощью еще одного необязательного параметра, указав `fast`, `fastchecked` или

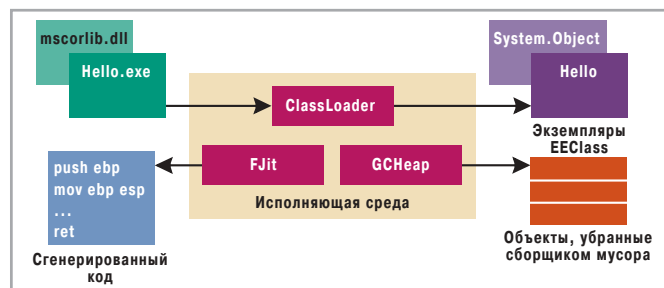


Рис. 4. Центральные классы исполняющей среды

checked. Этот параметр модифицирует определенные символы и флаги, передаваемые компилятору. При отладочной сборке (checked build) в генерируемый код включаются проверки на внутреннюю целостность и отменяются все виды оптимизации. А при отладочной сборке с оптимизацией (fastchecked build) (вариант по умолчанию) в генерируемый код включаются проверки на внутреннюю целостность, но разрешается оптимизация. Получаемый в результате такой сборки код работает заметно быстрее, но может оказаться непредсказуемым в отладчике. Наконец, нормальная сборка без включения отладочного кода (fast build), как и следовало ожидать, дает двоичные файлы, работающие предельно быстро. В любом случае env.bat выводит информацию о текущей сборке, например:

```
Setting environment for using Microsoft Visual Studio .NET tools.
(If you also have Visual C++ 6.0 installed and wish to use its tool
from the command line, run vcvars32.bat for Visual C++ 6.0.)
CLR environment (C:\sscli\clr\bin\rotorenv.bat)
Building for Operating System - NT32
    Processor Family - x86
    Processor - i386
    Build Environment = C:\sscli\rotorenv
    Build Type - checked
```

Настроив среду сборки, вы можете начать саму сборку, запустив файл buildall.cmd из корневого каталога установленной кодовой базы или сценарий buildall оболочки Unix. Buildall.cmd — довольно простой командный файл, так как основная работа перекладывается на запутанную систему make-файлов.

Весь процесс идет под управлением программы build.exe. Она знакома разработчикам драйверов устройств, использующих DDK, но большинство других программистов о ней ничего не знают (так же, как и я). Если вы хотите внести какие-нибудь изменения в процесс сборки Shared Source CLI, советую прочесть рубрику «Bugslayer» за январь 1999 г., которую ведет Джон Роббинс (John Robbins). (Я, кстати, упомянул ее в списке статей на смежную тематику.) В ней подробно описывается, как пользоваться программой build.exe. Более детальные сведения включены в документацию (каталог docs\buildtools).

Запустив сборку, пойдите выпейте чашечку кофе или прогуляйтесь. Например, на моей машине, в которой установлен процессор Pentium с тактовой частотой 1,1 ГГц и 512 Мб оперативной памяти, полная сборка занимает 20–30 минут. По окончании сборки в каталоге \build появится новое дерево каталогов, в котором содержится примерно 30 DLL и 23 EXE, не считая вспомогательных файлов. В этой структуре каталогов размещаются компоненты, знакомые тем, кто работает с CLR, в том числе мини-GAC, каталог config с файлом machine.config и, конечно, ядро исполняющей подсистемы, реализованное в Shared Source CLI.

«Hello, World!»

Лучший способ как следует разобраться во взаимодействии различных элементов исполняющей среды — проследить за компиляцией какой-нибудь простой программы и ее выполнением в Shared Source CLI. Помните: эта среда взаимодействует с кодом почти так же, как коммерческая, поэтому многое, чему вы научитесь, лазая по пещерам Shared Source CLI, имеет прямое отношение и к самой CLR.

Более сложная программа лишь помешает увидеть, как вступают в действие основные компоненты исполняющей среды Shared Source CLI. Так что я возьму самую простую программу, которая только известна человечеству:

```
class Hello{
    static void Main(){
        System.Console.WriteLine("Hello, World!");
    }
}
```

Начнем с компиляции и воспользуемся новеньким, как с иголки, компилятором C# из дистрибутива Shared Source CLI. Особой необходимости именно в этом компиляторе нет — код, скомпилированный в других средах, работает в Shared Source CLI без всякой модификации (и наоборот), если только он не обращается к библиотекам, которые не поставляются с Shared Source CLI. После запуска dumpbin /imports применительно к полученному двоичному файлу вы увидите, что в нем всего одна зависимость — от функции _CorExeMain в mscoree.dll:

```
Dump of file hello.exe
File Type: EXECUTABLE IMAGE
Section contains the following imports:
    mscoree.dll

    402000 Import Address Table
    402374 Import Name Table
    0 time date stamp
    0 Index of first forwarder reference

    0 _CorExeMain
```

Файл явно не должен работать в Shared Source CLI — никакой mscoree.dll и в помине нет. Ядро исполняющей подсистемы Shared Source CLI находится в sscoree.dll. И все же в этом скомпилированном EXE-файле есть зависимость от коммерческой CLR. В чем дело?

Всякий раз, когда вы компилируете исполняемый файл под .NET, компилятор создает в EXE-файле крохотную интерфейсную заглушку (stub). В Windows эта заглушка позволяет управляемым исполняемым файлам обманывать загрузчик, маскируясь под обычные Windows-программы. Как только Windows запускает управляемое приложение, заглушка вызывает _CorExeMain (вот откуда эта зависимость), которая на самом деле является тонкой оболочкой _CorExeMain2 (она-то и делает реальную работу). Среда, поддерживающие управляемый код (как, например, Shared Source CLI), игнорируют эту заглушку — в принципе они могут с ее помощью проверять, действительно ли программа содержит управляемый код, но на практике используют собственные загрузчики, знающие, как загружать и запускать такой код.

Поскольку у Shared Source CLI имеется свой загрузчик, она экспортирует не _CorExeMain, а _CorExeMain2 (рис. 5).

Запуск кода в среде Shared Source CLI требует проецирования в память соответствующей сборки, загрузки sscoree.dll и вызова _CorExeMain2. Shared Source CLI поставляется с кро-

Лучший способ как следует разобраться во взаимодействии различных элементов исполняющей среды — проследить за компиляцией простой программы и ее выполнением Shared Source CLI.

Рис. 5. _CorExeMain2

```

__int32 STDMETHODCALLTYPE _CorExeMain2( // код завершения EXE
    PBYTE pUnmappedPE,                // -> на проецируемый
                                        // в память код
    DWORD cUnmappedPE,                // размер этого кода
    LPWSTR pImageNameIn,              // -> на имя EXE
    LPWSTR pLoadersFileName,          // -> на имя загрузчика
    LPWSTR pCmdLine)                  // -> на командную строку
{
    BOOL bRetVal = 0;
    PEFFile *pFile = NULL;
    HRESULT hr = E_FAIL;
    if (!StrongNameSignatureVerification(pImageNameIn,
        SN_INFLAG_INSTALL|SN_INFLAG_ALL_ACCESS|SN_INFLAG_RUNTIME, NULL) &&
        StrongNameErrorInfo() != (DWORD) CORSEC_E_MISSING_STRONGNAME) {
        LOG((LF_ALL, LL_INFO10, "Program exiting due to strong name
            verification failure\n"));
        return -1;
    }
    HRESULT result = CoInitializeEE(COINITEE_DEFAULT);
    INSTALL_COMPLUS_EXCEPTION_HANDLER();
    hr = PEFFile::Create(pUnmappedPE, cUnmappedPE,
        pImageNameIn,
        pLoadersFileName,
        NULL,
        &pFile,
        FALSE);
    if (SUCCEEDED(hr)) {
        hr = SystemDomain::ExecuteMainMethod(pFile, pImageNameIn);
        bRetVal = SUCCEEDED(hr);
    }
exit:
    EEShutdown(FALSE);
    SafeExitProcess(GetLatchedExitCode());
    return (GetLatchedExitCode());
}

```

сечной программой (clix.exe), которая как раз это и делает. Она очень проста — в ней всего 400 строк на C++. Вы должны передать ей имя своего управляемого исполняемого файла:

```
clix hello.exe <аргументы>
```

Управление выполнением

Shared Source CLI открывает широкий простор для экспериментов и отладки, даже если вы еще не написали ни строчки кода. Например, класс EEConfig (он находится в eeconfig.h)

Табл. 2. Конфигурационные параметры Shared Source CLI

Параметр	Значение	Комментарий
BreakOnEELoad	0 1	Прервать при запуске
BreakOnEEShutdown	0 1	Прервать при завершении работы
BreakOnClassBuild	Имя класса	Прервать при загрузке класса с указанным именем
JitBreak	Класс::метод	Прервать перед компиляцией метода
JitHalt	Класс::метод	Прервать при выполнении метода
JitTrace	0 1	Регистрировать порядок JIT-компиляции
LogEnable	0 1	Мастер-ключ записи в журнал
LogLevel	1...10	Уровень детализации регистрируемой информации
LogToFile	0 1	Записывать регистрируемую информацию в файл
LogFile	Имя файла	Направлять регистрируемую информацию в файл с указанным именем
LogToDebugger	0 1	Направлять регистрируемую информацию отладчику
LogToConsole	0 1	Направлять регистрируемую информацию на консоль
JitOptimizeType	{ 0...3 }	Виды оптимизации {все, по размеру, по скорости, на усмотрение компилятора}

Табл. 3. Переменные окружения

Конфигурационный параметр	Описание
LogToDebugger	Вывод в OutputDebugString
LogToFile	Вывод в файл, указанный в LogFile
LogToConsole	Вывод на консоль

вызывается из сотен мест кодовой базы; он возвращает конфигурационные параметры, влияющие на выполнение соответствующих блоков кода Shared Source CLI. При выполнении «Hello, World» исполняющая среда проверяет значения не менее чем 105 различных параметров, контролирующих все аспекты работы кода. Некоторые из наиболее интересных параметров приведены в табл. 2. На данный момент подавляющее большинство параметров нигде не документировано, но я обнаружил, что исходный код класса EEConfig дает неплохое представление о многих из них. В крайнем случае ищите в кодовой базе ссылки на нужный вам параметр — таким способом обычно удастся понять его предназначение.

EEConfig использует вспомогательный класс REGUTIL, который ищет конфигурационные параметры сначала в переменных окружения, потом в файле Rotor.ini (в каталоге %USERPROFILE) и, наконец, в файле Rotor.ini, который находится в каталоге конфигурации данной сборки (например, в build\v1.x86checked.rotor\rotor\rotor.ini).

Формат Rotor.ini соответствует формату стандартных INI-файлов Windows, а ваши параметры должны находиться в разделе [Rotor]. Если вы настраиваете параметры через переменные окружения, не забудьте, что имена параметров должны начинаться с «COMPlus_»; таким образом, «COMPlus_foo=bar» в этой среде эквивалентно «foo=bar» в INI-файле.

В целом, наиболее полезны параметры, управляющие выводом информации. Исполняющая среда интенсивно использует весьма гибкий код в utilcode\log.cpp, отвечающий за запись в журнал; при этом вы можете выбирать степень детализации этой информации и место ее вывода. Главный ключ, разрешающий или запрещающий запись в журнал, — параметр LogEnable. Если он равен 1, запись разрешена, но никакой информации вы не получите, пока не укажете, куда ее записывать. Для этого надо установить одну или несколько переменных окружения

(табл. 3) в 1.

Степень детализации определяется значением параметра LogLevel (1–10). Эти значения связаны с директивами #define в log.h. Запуская программу «Hello, World!», я опробовал все значения, чтобы понять, как они влияют на вывод, и получил результаты, показанные в табл. 4.

Как видите, более высокие значения приводят к генерации колоссального объема данных. Гигантский размер дампов — не самое интересное. Куда важнее содержащаяся в них информация. Она помогает разобраться в том, что происходит в незнакомом исходном коде при его пошаговом выполнении в отладчике. При отладке кода я предпочитаю устанавливать

Табл. 4. Значения LogLevel

Значение	Символьное имя	Объем информации для «Hello World!»
10	LL EVERYTHING	31355 (2997 Kб)
9	LL_INFO1000000	10149 (925 Kб)
8	LL_INFO100000	10149 (925 Kб)
7	LL_INFO10000	4772 (417 Kб)
6	LL_INFO1000	2954 (282 Kб)
5	LL_INFO100	1273 (96 Kб)
4	LL_INFO10	564 (42 Kб)
3	LL_WARNING	0
2	LL_ERROR	0
1	LL_FATALERROR	0

переменную LogToDebugger — тогда получаешь представление о том, что правильного сделал твой код, пока не попал в точку прерывания.

Отладка с применением Visual Studio .NET

Есть много отладчиков, с помощью которых можно пошагово пройти исходный код Shared Source CLI. В Windows — это WinDBG, ntsd и Visual Studio .NET. Выбор зависит от того, что вы цените больше: мощь или удобство в использовании. Самые мощные отладчики — WinDBG и ntsd. Shared Source CLI поставляется с расширением WinDBG — SOS, или «Son of Strike» (еще одно внутрифирменное кодовое название), который позволяет исследовать весьма мудреные вещи вроде того, какие виды объектов Shared Source CLI поддерживает в стеке вызова. Если вы уже знакомы с WinDBG, то, наверное, предпочтете его — документация на SOS находится в каталоге docs\techinfo.

Может, я и болезненно любопытен, но все же не мазохист, и обычно верен отладчику Visual Studio .NET. Он дает 98% того, что мне нужно, на порядок проще в использовании, чем WinDBG или ntsd, и определенно соответствует тем целям, которые я ставил перед собой в этой статье.

В Shared Source CLI нет SLN-проектов для cli.exe, поэтому вы не можете просто загрузить ее в Visual Studio .NET и нажать клавишу F5. Впрочем, можно сделать нечто очень близкое к тому. При первой попытке отладки с помощью Visual Studio .NET я создал проект C# и настроил параметр Start Application в свойствах проекта на cli.exe. Это дало результат, далекий от задуманного. Во-первых, приходится отключать управляемую отладку, потому что соответствующий отладчик так и норовит зависнуть в среде Shared Source CLI (и его трудно винить за это). Во-вторых, пошаговое выполнение кода идет слишком медленно: на моей машине каждый оператор отнимал около пяти секунд. Это быстро заставило меня искать решение получше. И еще один немаловажный довод: при запуске Visual Studio с рабочего стола (т. е. из пользовательского интерфейса Windows) настраивать переменные окружения, о которых я уже рассказывал, весьма неудобно.

Выискивая более эффективные способы применения Visual Studio .NET, я ненароком наткнулся на необязательный аргумент /debugexe у программы devenv.exe. Как выяснилось, за ним скрывается самый легкий путь. Набрав в командной строке из каталога hello:

```
devenv /debugexe %sscli%\build\<buildversion>\cli.exe hello.exe
```

вы создаете отладочный проект Visual Studio .NET, построенный вокруг cli.exe. Нажимая клавишу F11 (Step Into), вы попадаете прямо в функцию main программы cli. Теперь пошагово выполняйте код и очень скоро вы окажетесь перед воротами в исполняющую среду (рис. 6).

Увы, поставить точку прерывания прямо в выражение Console.WriteLine в Hello.cs нельзя. Не забудьте, что мы имеем дело с неуправляемым отладчиком из Visual Studio, а он бесечно игнорирует любые точки прерывания, которые вы ставите в коде на C# или Visual Basic .NET. Пока еще нет отладчика, который позволял бы одновременно вести отладку и просматривать переменные в управляемом коде на уровне виртуальной машины.

Итак, каким же образом сборка Hello.exe превращается в код? Функция _CorExeMain2 начинает с подготовки ядра исполняющей подсистемы и передачи ему hello.exe. Перед запуском вашего кода _CorExeMain2 вызывает StrongNameSignatureVerification для проверки цифровой подписи в соответствующей сборке. Если запускаемый образ успешно проходит проверку, в дело вступает цепочка кода инициализации (через CoInitializeEE), которая в конечном счете приводит в функцию EEStartup (ceemain.cpp). В ней можно блуждать целыми днями: EEStartup запускает практически все вспомогательные сервисы, необходимые для выполнения вашего кода, в том числе сборщик мусора, JIT-компилятор и диспетчер пула потоков (thread pool manager).

После инициализации всех сервисов в ядре исполняющей подсистемы CoInitializeEE возвращает управление _CorExeMain2. К этому моменту считайте, что ключ зажигания повернут и что исполняющая среда готова сорваться с места. _CorExeMain2 нажимает на педаль газа вызовом SystemDomain::ExecuteMainMethod.

Загрузка классов

Каждая управляемая программа начинается с вызова основной точки входа, которой в Hello.exe, конечно, является Main. На этом этапе класс Hello существует в виде кучки IL-кода и записей в таблицах метаданных в Hello.exe. Поскольку исполняющая среда вызывает Main (или то, что ее заменяет), ExecuteMainMethod должен считать эти таблицы и сформировать представление класса в памяти. Эта задача настолько трудна, что в Shared Source CLI ею занимается отдельный класс — ClassLoader.

ClassLoader продлевает весьма внушительную работу. Ему приходится определять число полей в загружаемом классе и их типы, чтобы исполняющая среда могла потом создавать экземпляры этого класса. Ему приходится распутывать взаимосвязи между загружаемым классом и его предками, так как у базовых классов свои члены и поля, часть которых может быть переопределена в загружаемом классе. Все это занимает примерно 15 000 строк в файлах clsload.cpp, class.cpp и

Гигантский размер дампов — не самое интересное. Куда важнее содержащаяся в них информация. Она действительно помогает разобраться в том, что происходит в незнакомом исходном коде при его пошаговом выполнении в отладнике.

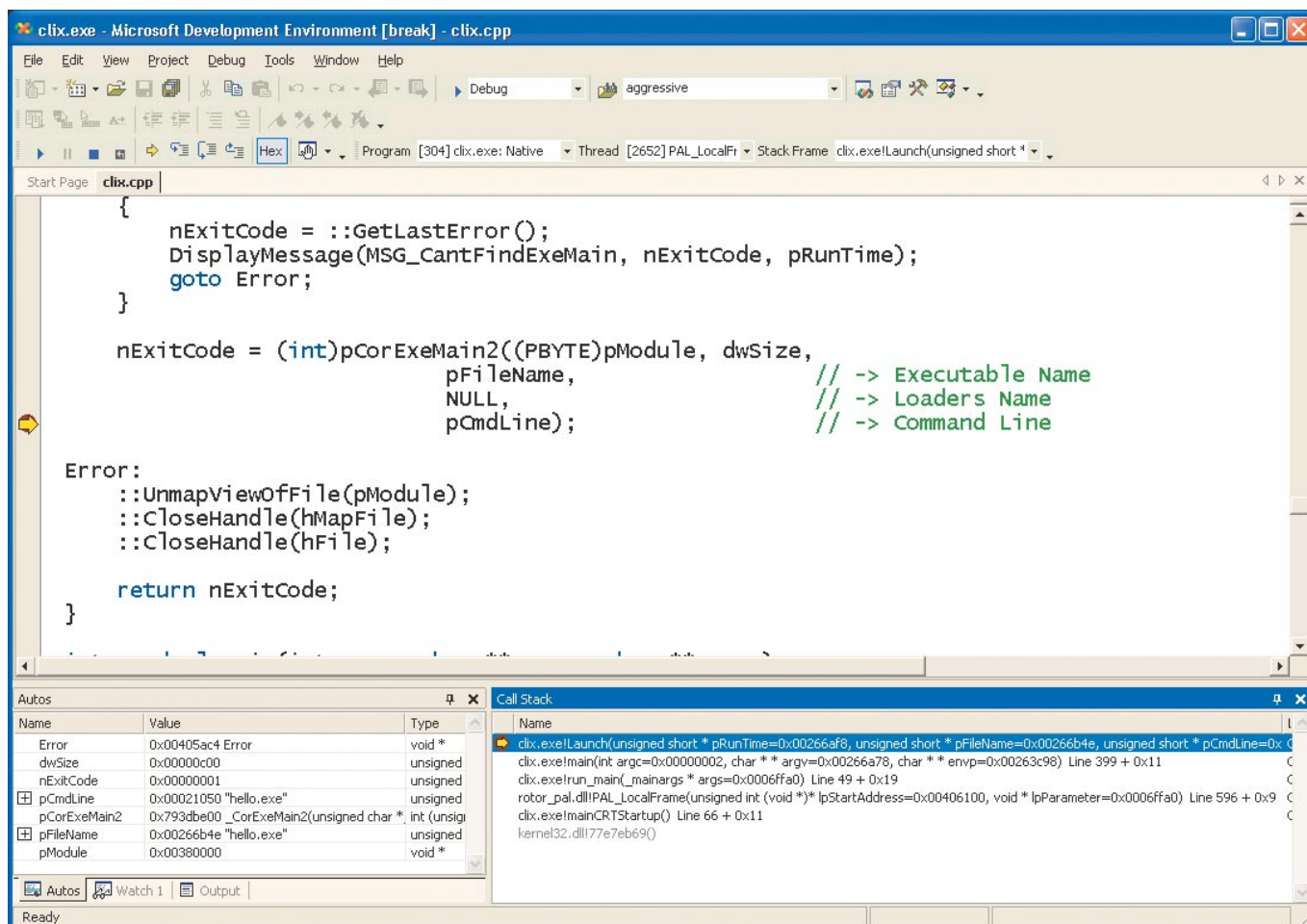


Рис. 6. Вход в Shared Source CLI через devenv/debugexe

method.cpp. Центральный метод, координирующий выполнение всей работы, — `ClassLoader::LoadTypeHandleFromToken`:

```
HRESULT ClassLoader::LoadTypeHandleFromToken(
    [in] Module *pModule,
    [in] mdTypeDef cl,
    [out] EEClass** ppClass,
    [inout] OBJECTREF *pThrowable)
```

`LoadTypeHandleFromToken` принимает пару {модуль, маркер доступа}, которая идентифицирует нужный класс. Затем он перебирает метаданные и на их основе создает экземпляр класса `EEClass`, который возвращается вызвавшему коду. Каждый класс, используемый ядром исполняющей подсистемы Shared Source CLI, в конечном счете загружается через `LoadTypeHandleFromToken`, так что это весьма популярный

кусочек кода. Поставив в нем точку прерывания, вы сможете наблюдать за загрузкой всех классов. Это любопытно... в первый раз. Успокоившись и решив заняться настоящим делом, вы, наверное, захотите наблюдать за загрузкой только своего класса. Тут вы облегчите себе жизнь, воспользовавшись конфигурационными параметрами, о которых я уже упоминал. Настройте `BreakOnClassBuild` на имя интересующего вас класса, и `ClassLoader` откроет диалоговое окно ASSERT перед самым началом работы `LoadTypeHandleFromToken`. Учтите, что это нужно делать до запуска отладчика Visual Studio и что указывать надо полное имя: например, `System.Object`, а не `Object`. Это позволит вам пошагово пройти процесс загрузки класса и увидеть, как зарождается объект `EEClass`.

Еще один полезный параметр — `ShouldDumpOnClassLoad`. Если вы настроите его на имя нужного класса, в файл журнала будут записываться детальные сведения о созданном объекте `EEClass`. Такой способ создания дампа класса Hello дает вывод, показанный на рис. 7.

В отличие от объектов COM-класса объекты `EEClass` занимают центральное место в работе исполняющей среды. Каждый класс, загружаемый CLR, представляется в период выполнения экземпляром класса `EEClass`. Если вы когда-нибудь пользовались пространством имен `System.Reflection`, то уже имели дело с `EEClass`, потому что все полезные свойства и методы `System.Type` и его дружественные члены вроде `MethodInfo` и `FieldInfo` реализуются на основе информации, извлекаемой из соответствующего `EEClass`. На рис. 8 показан `EEClass` для класса Hello.

Рис. 7. Дамп для класса Hello

```
Method table summary for 'Hello':
Number of static fields: 0
Number of instance fields: 0
Number of static obj ref fields: 0
Number of declared fields: 0
Number of declared methods: 2
Vtable (with interface dups) for 'Hello':
Total duplicate slots = 201
slot 0: System.Object[mscorlib.dll]::ToString 0x852073 (slot = 0)
slot 1: System.Object[mscorlib.dll]::Equals 0x852223 (slot = 1)
slot 2: System.Object[mscorlib.dll]::GetHashCode 0x852253 (slot = 2)
slot 3: System.Object[mscorlib.dll]::Finalize 0x852113 (slot = 3)
<- vtable ends here
slot 4: Hello[hell.exe]::Main 0x8350AB (slot = 4)
slot 5: Hello[hell.exe]::ctor 0x8350D3 (slot = 5)
```

Выполнение управляемого кода

Загрузив класс Hello, исполняющая среда готова к выполнению основного метода. Остается лишь одна проблема: Hello.Main все еще находится в виде IL-кода.

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    Idstr "Hello, World!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

Чтобы выполнить эту функцию, исполняющая среда сначала должна запустить JIT-компилятор, который генерирует машинный x86-код (вы уже видели его на **рис. 1**).

Shared Source CLI, как и CLR, не компилирует методы до их первого вызова. Для уменьшения издержек, связанных с JIT-компиляцией, Shared Source CLI никогда не запускает JIT-компилятор явным образом. Вместо этого она настраивает одну хитроумную структуру данных для автоматического запуска этого компилятора. В каждом объекте EEClass (в члене m_pMethodTable) хранится таблица объектов MethodDesc (**рис. 8**). Управляемое выполнение начинается с вызова этой функции. В случае Hello ClassLoader вызывает функцию Call применительно к MethodDesc для Hello::Main.

На **рис. 9** таблица методов представлена более детально. Каждый элемент в таблице хранит член m_CodeOrIL, который изначально устанавливается на IL-процедуру в несущей сборке. Каждый элемент также сопоставлен с крошечной интерфейсной заглушкой из настоящего исполняемого кода — в пяти байтах, предшествующих адресу MethodDesc. MethodDesc::Call использует эту крохотную прокладку для обращения к заглушке и выполнения содержащегося в ней кода. Вероятно, вы обратили внимание на адреса, проставленные рядом с каждым методом в недавно показанном дампе, — это адреса заглушки, сопоставленной с методами.

Как видно на **рис. 9**, изначально заглушка содержит машинную инструкцию CALL (для процессоров типа x86), передающую управление глобальной подпрограмме, которую я называю универсальным шлюзом JIT (Universal JIT Thunk). По адресам, перечисленным в дампе, вы найдете значение 0xE8 — это операционный код (opcode) относительного вызова функции (relative function call) на процессорах типа x86.

Универсальный шлюз JIT — небольшая ассемблерная подпрограмма, передающая управление JIT-компилятору. Последний ищет IL-код через m_CodeOrIL и генерирует машинный код. После этого m_CodeOrIL и заглушка обновляются так, чтобы они указывали на только что сгенерированный код: при следующем вызове Call управление тут же будет передано машинному коду (**рис. 10**).

Этот ловкий прием обеспечивает каскадную JIT-компиляцию в сложных программах. Всякий раз,

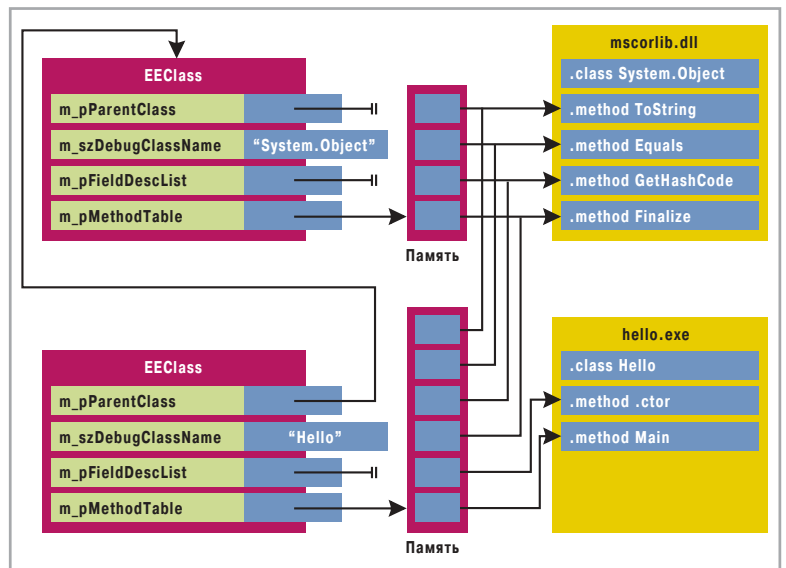


Рис. 8. Объекты класса в памяти

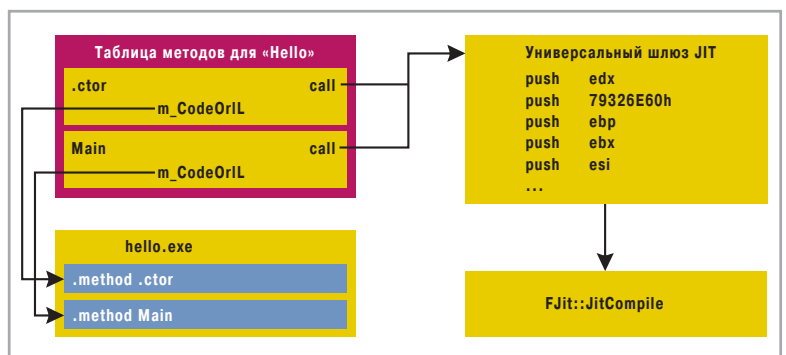


Рис. 9. Перед JIT-компиляцией

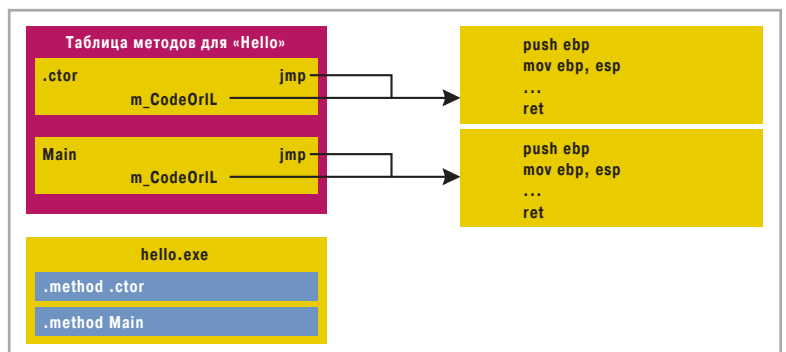


Рис. 10. После JIT-компиляции

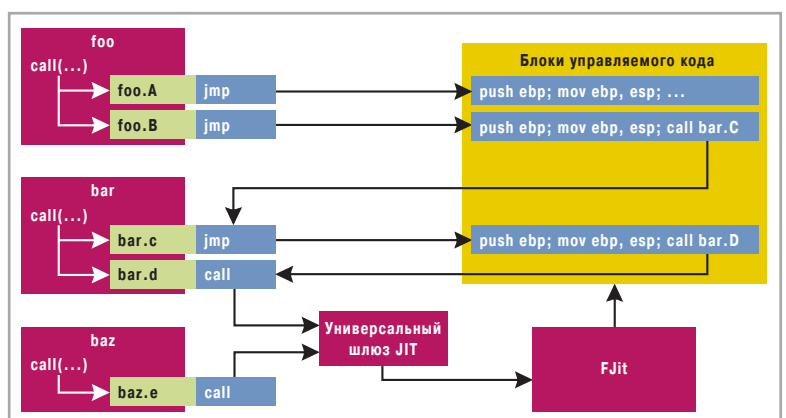


Рис. 11. Пример JIT-компиляции

Рис. 12. Стек вызова

```

...
sscoree.dll!MethodDesc::DoPreStub
sscoree.dll!PreStubWorker
0x0008fcd5                <- JIT
sscoree.dll!_CallDescrWorkerInternal@16
...
sscoree.dll!MethodDesc::DoPreStub
sscoree.dll!PreStubWorker
0x0008fcd5                <- JIT
0x02d42e05
sscoree.dll!_CallDescrWorkerInternal@16
...
sscoree.dll!MethodDesc::Call

```

когда JIT-компилятор генерирует порцию машинного кода, откуда вызываются другие методы, он выдает инструкции CALL для вызова соответствующей заглушки, и весь процесс повторяется. Например, на **рис. 11** методы A и B класса Foo уже скомпилированы.

При компиляции Foo.B вызов Bar.C был разрешен в заглушку, хранимую объектом EEClass для Bar. То же самое относится к вызову Bar.D из Bar.C. Метод Bar.D еще не скомпилирован, поэтому, когда Bar.C впервые вызывает Bar.D, управление передается JIT-компилятору. В результате вы получаете нечто, похожее на стек вызова, показанный на **рис. 12**.

В этом примере универсальный шлюз JIT находится по адресу 0x008fcd5. Первое обращение к MethodDesc::Call привело к вызову метода, пропущенного через JIT-компилятор, а из этого метода вызван второй метод, который тоже надо было транслировать JIT-компилятором. По мере того как приложение обрастает скомпилированным кодом, оно все реже докучает JIT-компилятору.

Когда работа JIT-компилятора завершается, он возвращает управление только что модифицированной интерфейсной заглушке, откуда управление передается машинному коду. И в конце концов выполняется код, показанный на **рис. 1**.

Заключение

Конфигурационные параметры JitHalt и JitBreak упрощают наблюдение за компилируемыми методами. Оба параметра служат одной цели: помогают попасть в интересующий вас метод. JitBreak позволяет подключиться к FJit::CompileMethod как раз в том месте, где начинается самое веселье, и на-

блюдовать за компиляцией функции. JitHalt сообщает JIT-компилятору вставить в сгенерированный блок кода DebugBreak (INT 3) как первую инструкцию, что заставляет отладчик останавливаться при каждом вызове данного метода. Обе переменные должны быть инициализированы строкой в формате «класс::метод». Например:

```
JitBreak=Hello::Main
```

будет приводить при компиляции Hello::Main к открытию диалогового окна наподобие показанного на **рис. 13**. В этих переменных можно пользоваться символами подстановки, поэтому оператор:

```
JitBreak=Hello::*
```

вызывает прерывание каждый раз, когда компилируется какой-нибудь метод, реализованный в классе Hello. Вы можете легко следить за тем, какие функции транслируются JIT-компилятором, присвоив JitTrace значение, равное 1. Более того, параметр JitOptimizeType позволяет управлять тем, какой код будет генерировать JIT-компилятор.

JitOptimizeType принимает значения от 0 до 4, которые соответствуют значениям переменной перечислимого типа, определенной в eeconfig.h:

```

enum { OPT_BLENDED,
OPT_SIZE,
OPT_SPEED,
OPT_RANDOM,
OPT_DEFAULT = OPT_BLENDED };

```

Поиграв какое-то время с разными параметрами, вы поймете, что готовы (и хотите) исследовать запутанные ла-

биринты управляемого кода.

Концепция управляемого кода — это, пожалуй, самое важное из того, что приходило к нам из Редмонда*. Она подразумевает куда более тесные связи между платформой и вашим кодом. Разобравшись в этих связях, вы сможете писать более эффективные программы. Shared Source CLI — потрясающий инструмент для этой цели, а его внутреннее устройство во многом аналогично CLR. Приемы, которые я показал в статье, помогут вам начать свой трудный путь по этой кодовой базе и получить представление о том, как стыкуются друг с другом ее отдельные части. Работа с Shared Source CLI позволит вам углубить свои знания CLR и научиться тому, чему без исходного кода научиться нельзя. Так раскройте же занавес, смотрите и получайте удовольствие!

Джейсон Уиттингтон (Jason Whittington) — независимый консультант по программному обеспечению (живет в Аризоне). Большую часть времени работает на DevelopMentor, преподавая технологии .NET. В свободное от преподавания время трудится над книгой «CLR Internals» в соавторстве с Питером Дрэйтоном (Peter Drayton).

* В Редмонде находится штаб-квартира корпорации Microsoft. — Прим. перев.

C#

XML-комментарии позволяют создавать документацию прямо из исходных файлов в Visual Studio .NET

Эта статья предполагает знание XML, XSLT и C#.

Уровень сложности: ① ② ③

АННОТАЦИЯ C# позволяет разработчикам встраивать XML-комментарии в исходный код — полезная возможность, особенно когда над проектом работают несколько программистов. Синтаксический анализатор (parser) C# может раскрывать такие XML-тэги, предоставляя дополнительную информацию, и экспортировать их во внешний документ для дальнейшей обработки. В этой статье рассматривается применение XML-комментариев и объясняются соответствующие тэги. Автор показывает, как настроить проект на экспорт XML-комментариев в подходящую для других разработчиков документацию. Также демонстрируется, как использовать комментарии для генерации справочных файлов.

Статьи на смежную тематику:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs700/html/expvsnetlab2.asp>.

Базовую информацию, необходимую для понимания этой статьи, см. на Web-страницах по HTML Help (<http://msdn.microsoft.com/library/en-us/htmlhelp/html/vsconHH1Start.asp>) и Visual Studio (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vsstartpage.asp>).

В любом проекте всегда найдется кто-то недовольный документацией. Руководителям группы хочется больше комментариев в тексте программ, техническим писателям — детальных сведений об исходном коде, отделу контроля качества — функциональных спецификаций и т. д. Но даже если бы эти документы были написаны, все равно пришлось бы воевать за соответствие их версий.

Разве не здорово, если бы вся эта информация хранилась в одном месте? Лучше всего — в самом исходном коде; это облегчило бы изменение документации одновременно с исходным кодом. Но разбираться в чужом коде — задача неблагодарная даже для умеющих программировать. Что уж говорить о тех, кто не слишком подкован в программировании. Кроме того, на вставку документации в код нужно время.

В статье объясняется, как XML-комментарии помогают решить многие из этих проблем. Комментарии к коду, руководства пользователя, планы тестирования и множество другой документации можно генерировать из одного источника через XML-тэги. Я расскажу, как вставлять XML-комментарии и разрешать их экспорт в другой файл. Далее мы обсудим все доступные тэги и формат XML-файла, в который они помещаются, и на практике создадим справочные файлы на основе XML-комментариев и XSLT.

XML-комментарии

Каждый XML-комментарий начинается с трех символов «`///`». Первые два обозначают комментарий и указывают компилятору игнорировать дальнейший текст, а третий сообща-

ет синтаксическому анализатору, что это XML-комментарий, который следует обрабатывать по-особому.

Когда разработчик набирает три символа «`///`» подряд, интегрированная среда разработки (IDE) Visual Studio .NET проверяет, не предшествуют ли они распознаваемому типу или определению элемента типа. Если да, IDE автоматически вставляет некоторые тэги. После этого разработчик может добавить другие нужные ему тэги и значения. Например, ниже приведены XML-тэги, генерируемые, когда три символа «`///`» предшествуют функции-члену, принимающей некий параметр.

```
/// <summary>
///
/// </summary>
/// <param name="strFilePath"></param>
public void LoadXMLFromFile(string strFilePath)
```

Это далеко не все тэги, известные Visual Studio .NET IDE. Хотя текущая реализация IntelliSense для XML-комментариев не поддерживает все перечисленные в спецификации C# тэги, недостающие всегда можно вставить вручную.

После того как комментарии вставлены, неплохо бы что-нибудь с ними сделать. Под «что-нибудь» обычно подразумевается экспорт во внешний файл всего текста, расположенного за тремя символами «`///`».

Есть несколько predefined XML-тэгов, известных синтаксическому анализатору. Обнаруживая эти тэги (при условии, что разработчик правильно ими воспользовался), он форматирует их текст и выводит в текстовый файл. Если встречается тэг, отличный от predefined, он и его текст без изменений включаются во внешний файл. Благодаря этому вы можете создавать собственные тэги или даже использовать известные тэги из других источников (например, HTML).

Когда экспорт настроен правильно, синтаксический анализатор C# извлекает XML-комментарии, форматирует их и помещает в выбранный XML-файл. Задать местонахождение и имя файла XML-документа, а также разрешить экспорт можно на страницах свойств проекта.

Для этого выполните четыре операции:

- откройте свойства проекта (для этого в окне Solution Explorer можно щелкнуть имя проекта правой кнопкой мыши и выбрать команду Properties);
- в появившемся диалоговом окне щелкните папку Configuration Properties;
- щелкните Build;
- в правой секции окна вы увидите поле свойства XML Documentation File. Укажите в нем путь (относительно каталога проекта) и имя нужного файла.

В данном случае в качестве имени документа я ввел GiveHelpDoc.xml (рис. 1). Если вы не укажете здесь путь и имя файла (вариант по умолчанию), XML-комментарии не будут извлекаться в отдельный файл.

Распознаваемые тэги

Я разделяю тэги XML-комментариев на две категории. В первую входят тэги, которые я называю первичными (primary

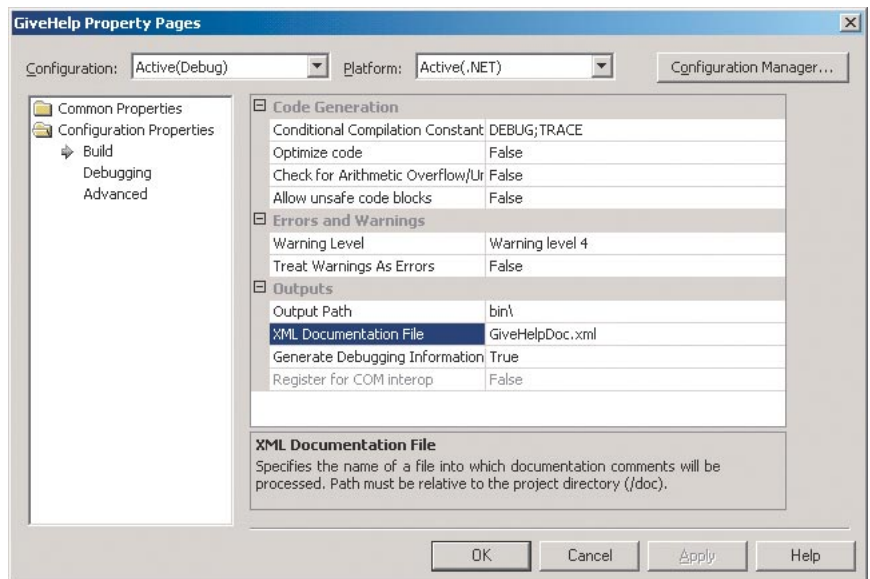


Рис. 1. Настройка обработки XML-комментариев

tags); с них всегда начинается какая-либо группа тэгов XML-комментариев. Первичные тэги никогда не вкладываются в другие тэги. Ко второй категории я отношу тэги, используемые совместно с первичными в качестве модификаторов текста. Я называю их вспомогательными (support tags). Чтобы различать вспомогательные и HTML-тэги, первые набираются в нижнем регистре, вторые — в верхнем. Примеры первичных и вспомогательных тэгов, которые будут рассмотрены в статье, показаны на рис. 2.

Существует 9 первичных тэгов: `<remarks>`, `<summary>`, `<example>`, `<exception>`, `<param>`, `<permission>`, `<returns>`, `<seealso>` и `<include>`. В этом контексте тэг `<remarks>` используется для описания какого-либо типа, например класса:

```
/// <remarks>
/// Класс, содержащий функции для преобразований в справочных файлах
/// </remarks>
```

В документации по языку C# рекомендуется применять тэг `<remarks>` для описания типа, а тэг `<summary>` — для описания элемента типа. Как ни странно, если перед типом ввести комбинацию «`///`», Visual Studio .NET IDE вставит тэг `<summary>`. Следовательно, тэг `<remarks>` надо вводить вручную.

Из всех тэгов в файлах исходного кода на C# чаще всего встречается `<summary>`. Он описывает элементы (члены) типа, включая методы, свойства и поля:

```
/// <summary>
/// Этот атрибут, основанный на XmlDocument, содержит
/// XML-документацию по проекту
/// </summary>
```

С помощью тэга `<summary>` можно описывать и сами типы, но это не рекомендуется. В документации по XML-комментариям с этой целью предлагается работать с тэгом `<remarks>`.

Тэг `<example>` выделяет пример использования элемента. Примером может быть любой текст, в том числе фрагмент кода:

```
/// <example>
/// <code>
/// // Создать класс для преобразований
/// GiveHelpTransforms ght = new GiveHelpTransforms();
```

```

/// // Загрузить XML в свойство SourceXML
/// ght.LoadXMLFromFile(
///     "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveHelpDoc.xml");
/// </code>
/// </example>

```

Код обычно помечается тэгом `<code>` (о нем — в разделе по вспомогательным тэгам).

Тэг `<exception>` документирует исключения, которые может генерировать элемент. Для описания нескольких исключений можно использовать несколько таких тэгов. В отличие от большинства других тэгов, `<exception>` имеет один атрибут — `cref`. Его значение — имя возбуждаемого исключения. При этом вы должны указывать существующий класс, так как синтаксический анализатор C# проверяет его и извлекает из него контекстную информацию, помещаемую в XML-документацию. Я поясню этот механизм чуть позже.

В коде, приведенном в моей статье, никакие исключения не генерируются, но я все же покажу применение атрибута `cref` в тэге `<exception>`:

```

/// <exception cref="SampleException">
///     Обычно здесь описывается исключение
/// </exception>

```

Тэг `<param>` описывает параметры метода или свойства. Он автоматически вставляется IDE, если перед методом указыва-

ются три символа `«///»`. Тэг `<param>` имеет один атрибут — `name` — совпадающий с именем параметра:

```

/// <param name="strFilePath">Путь к файлу, содержащему
/// исходный XML.</param>

```

Разрешение на доступ к элементам указывается в тэге `<permission>`. Соответствующий ему текст содержит описание разрешения. К значению этого тэга не предъявляется никаких требований. Это может быть информация о виде разрешения на доступ: является ли элемент открытым (`public`), закрытым (`private`), защищенным (`protected`) и др. Но значением может быть и область видимости (`scope`) с информацией о том, статический ли это метод. В общем, можете включать туда все, что сочтете нужным.

У тэга `<permission>` имеется единственный атрибут — `cref`. В документации атрибут `cref` в этом контексте описывается как «ссылка на элемент или поле, доступное из текущей среды компиляции». Его обычно устанавливают как `System.Security.PermissionSet`:

```

/// <permission cref="System.Security.PermissionSet">Открытый
/// доступ</permission>

```

Тэг `<returns>` похож на тэг `<param>`, но описывает возвращаемое значение метода или свойства:

```

/// <returns>HTML-код, содержащий список типов из XML-
/// документации.</returns>

```

Рис. 2. Фрагмент файла GiveHelpTransforms.cs

```

namespace GiveHelp
{
    /// <remarks>
    /// Class that contains functions to do transformations to help
    /// files. The source XML is loaded into the <see cref="SourceXML"/>
    /// property (e.g. <c><I>obj</I>.SourceXML = "<I>XML goes
    /// here</I>"</c>). One of the associated member functions
    /// (<see cref="GiveTypeListHTMLHelp"/>,
    /// <see cref="GiveMemberListHTMLHelp"/>,
    /// <see cref="GiveMemberHTMLHelp"/>)
    /// is called to initiate and then return the transformation.
    /// <para>
    /// <list type="table">
    /// <listheader>
    /// <term>Help Page</term>
    /// <description>Function to call</description>
    /// </listheader>
    /// <item><term>List of Types</term>
    /// <description>GiveTypeListHTMLHelp</description></item>
    /// <item><term>List of members</term>
    /// <description>GiveMemberListHTMLHelp</description></item>
    /// <item><term>Help for a single member</term>
    /// <description>GiveMemberHTMLHelp</description></item>
    /// </list>
    /// </para>
    /// </remarks>
    /// <permission cref="">public</permission>
    /// <example><code>
    /// // Создать класс, осуществляющий преобразования
    /// GiveHelpTransforms ght = new GiveHelpTransforms();
    /// // Загрузить XML в свойство SourceXML
    /// ght.LoadXMLFromFile(
    ///     "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveHelpDoc.xml");
    /// // Выполнить преобразование и вывести строку
    /// Response.Write( ght.
    /// GiveMemberHTMLHelp(Request.QueryString.Get("Type"),
    /// Request.QueryString.Get("Member")) );
    /// </code></example>
    public class GiveHelpTransforms
    {
        ...
        /// <summary>
        /// Calling this function will take the XML in <see
        /// cref="SourceXML"/>
        /// and translate it to a list of Members in the specified type.
        /// </summary>
        /// <param name="strType">The fully qualified name of the type
        /// that the member is in.</param>
        /// <returns>The HTML that lists the types that are in the XML
        /// documentation.</returns>
        /// <seealso cref="GiveTypeListHTMLHelp"/>
        /// <seealso cref="GiveMemberHTMLHelp"/>
        /// <example><code>
        /// // Создать класс, осуществляющий преобразования
        /// GiveHelpTransforms ght = new GiveHelpTransforms();
        /// // Загрузить XML в свойство SourceXML
        /// ght.LoadXMLFromFile(
        ///     "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveHelpDoc.xml");
        /// // Выполнить преобразование и вывести строку
        /// Response.Write(
        /// ght.GiveMemberListHTMLHelp(Request.
        /// QueryString.Get("Type"));
        /// </code></example>
        /// <permission cref="">public</permission>
        public string GiveMemberListHTMLHelp(string strType)
        {
            // Загрузить соответствующий XSLT
            XsltTransform xsltTransformFile = new XsltTransform();
            xsltTransformFile.Load(
                "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveTypeMemberListHelp.xsl");

            // Создать репозиторий для хранения выходных результатов
            System.Text.StringBuilder sbTransformed =
                new System.Text.StringBuilder();
            System.IO.TextWriter tw =
                (System.IO.TextWriter)new
                System.IO.StringWriter(sbTransformed);

            // Параметр strType передается таблице стилей
            XsltArgumentList arglist = new XsltArgumentList();
            arglist.AddParam("WhichType", "", strType);

            xsltTransformFile.Transform(m_xdSourceXML, arglist, tw);

            return tw.ToString();
        }
        ...
    }
}

```

В тэге `<seealso>` указываются «прочие» ссылки по этому тематическому разделу (topic). Этот тэг обычно не содержит текстовое значение — только атрибут `href` со ссылкой на нужный символ (тип, элемент, поле и т. д.).

```
/// <seealso href="GiveMemberListHTMLHelp"/>
```

Компилятор XML определяет содержимое по указанному символу и соответственно использует его при компиляции XML-документации. Я вернусь к этой теме в разделе, посвященном файлу XML-документации.

Когда компилятор извлекает XML-комментарии из кода, любой файл, указанный в тэге `<include>`, раскрывается, и содержащиеся в нем комментарии интерпретируются так, будто они находятся в самом исходном коде. Благодаря этому

Когда компилятор извлекает XML-комментарии из кода, любой файл, указанный в тэге `<include>`, раскрывается, и содержащиеся в нем комментарии интерпретируются так, будто они находятся в самом исходном коде.

комментарии можно хранить во внешнем файле. Правда, тогда их не будет «под рукой», и лично я стараюсь не пользоваться тэгом `<include>`. Но если у вас очень длинные комментарии, такой вариант, вероятно, вас устроит.

В тэге `<include>` должно присутствовать несколько атрибутов, характеризующих внешний файл. Атрибут `file` задает имя файла, причем путь может быть относительным или пол-

ным. Включаемый файл представляет собой XML-документ с XML-комментариями. Атрибут `path` — это выражение XPath, указывающее на родительский элемент XML-комментария во внешнем документе:

```
/// <include file='MyXMLCommentFile.xml'
/// path='doc/members/member[@name="T:MyExampleClass"]/*' />
public class MyExampleClass
{
    /// <include file='MyXMLCommentFile.xml'
    /// path='doc/members/member[@name="M:MyExampleMethod"]/*' />
    public string MyExampleMethod(string strReturnThis)
    {
        return strReturnThis;
    }
}
```

На рис. 3 приведен текст файла `MyXMLCommentFile.xml`. Заметьте, что в этом примере я указал родительский элемент, содержащий XML-комментарии, с помощью XPath. Реальный формат файла выбирается самим разработчиком. Я предпочел формат, аналогичный формату файлов XML-документов, генерируемых компилятором (об этом — позже).

Вспомогательные тэги

Существует 11 вспомогательных тэгов: `<c>`, `<code>`, `<list>`, `<listheader>`, `<item>`, `<term>`, `<description>`, `<para>`, `<param-ref>`, `<see>` и `<value>`.

Тэг `<c>` определяет строку текста как код и часто встречается в текстах описаний:

```
/// Исходный XML загружается в свойство <see href="SourceXML"/>
/// (например, <c><I>obj</I>.SourceXML = "<I>Сюда помещается
/// XML</I>"</c>).
```

Тэг `<code>` определяет раздел текста как блок кода. Он часто используется внутри блока тэга `<example>` (вы уже видели это), указывая, что формат текста в комментариях должен быть сохранен:

```
/// <code>
/// // Создать класс, осуществляющий преобразования
/// GiveHelpTransforms ght = new GiveHelpTransforms();
/// // Загрузить XML в свойство SourceXML
/// ght.LoadXMLFromFile(
///     "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveHelpDoc.xml");
/// </code>
```

Тэг `<list>` применяется внутри других тэгов комментариев и определяет специализированный список (specialized list). Тип списка задается атрибутом, значением которого может быть `number` (нумерованный), `bullet` (маркированный) или `table` (таблица). Внутри тэга `<list>` размещаются тэги, обозначающие компоненты списка, например:

```
/// <list type="table">
/// <listheader>
/// <term>Help Page</term>
/// <description>Function to call</description>
/// </listheader>
/// <item><term>List of types</term>
/// <description>GiveTypeListHTMLHelp</description></item>
/// <item><term>List of members</term>
/// <description>GiveMemberListHTMLHelp</description></item>
/// <item><term>Help for a single member</term>
/// <description>GiveMemberHTMLHelp</description></item>
/// </list>
```

Тэг `<listheader>` содержит информацию о заголовке списка (см. предыдущий пример). Он обычно используется для списков табличного типа.

Тэг `<item>`, как вы, наверное, уже догадались, обозначает элемент в списке. Он может включать в себя тэги `<term>` и `<description>`. Последние всегда являются дочерними по отношению к тэгу `<listheader>` или `<item>`, используются парно, и их назначение очевидно.

Тэг `<para>` идентифицирует новый абзац и очень похож на тэг `<P>` в HTML. Понятно, что он предназначен для разбиения длинных комментариев на абзацы:

```
/// <summary>Эта аннотация.
/// <para>Это новый абзац.</para>
/// </summary>
```

Рис. 3. `MyXMLCommentFile.xml`

```
<doc>
  <members>
    <member name="T:MyExampleClass">
      <remarks>
        This is my example class that does <b>nothing</b>.
      </remarks>
    </member>
    <member name="M:MyExampleMethod">
      <summary>
        This method just returns a string.
      </summary>
      <param name="strReturnThis">
        This is a string parameter that will just be returned.
      </param>
      <returns>
        Just returns the input parameter.
      </returns>
    </member>
  </members>
</doc>
```

Рис. 4. Фрагмент файла GiveHelpDoc.xml

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>GiveHelp</name>
  </assembly>
  <members>
    <member name="T:GiveHelp.GiveHelpTransforms">
      <remarks>
        Class that contains functions to do transformations to help
        files. The source XML is loaded into the <see
        cref="P:GiveHelp.GiveHelpTransforms.SourceXML"/> property
        (e.g. <c><I>obj</I>.SourceXML = "<I>XML goes here</I>"
        </c>). One of the associated member functions (<see
        cref="M:GiveHelp.GiveHelpTransforms.
        GiveMemberListHTMLHelp(System.String)"/>, <see
        cref="M:GiveHelp.GiveHelpTransforms.GiveMemberHTMLHelp
        (System.String)"/>) is called to initiate and then return
        the transformation.
      </remarks>
      <para>
        <list type="table">
          <listheader>
            <term>Help Page</term>
            <description>Function to call</description>
          </listheader>
          <item><term>List of types</term>
            <description>GiveTypeListHTMLHelp</description></item>
          <item><term>List of members</term>
            <description>GiveMemberListHTMLHelp</description>
          </item>
          <item><term>Help for a single member</term>
            <description>GiveMemberHTMLHelp</description></item>
        </list>
      </para>
    </member>
    <permission cref="!:>public</permission>
    <example><code>
      // Создать класс, осуществляющий преобразования
      GiveHelpTransforms ght = new GiveHelpTransforms();
      // Загрузить XML в свойство SourceXML
      ght.LoadXMLFromFile(
        "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveHelpDoc.xml");

      // Выполнить преобразование и вывести строку
      Response.Write(
        ght.GiveMemberHTMLHelp(Request.QueryString.Get("Type"),
        Request.QueryString.Get("Member")) );
    </code></example>
  </member>
  ...
  <member name="P:GiveHelp.GiveHelpTransforms.SourceXML">
    <value>
      The SourceXML property contains the XML that will be used in
      the transformations by the member functions for this class.
    </value>
    <permission cref="!:>public</permission>
  </member>
</members>
</doc>
```

Если в тексте комментария надо выделить некий параметр, используется тэг <paramref>. Он помещается там, где в тексте должно выводиться имя параметра. Основное назначение этого тэга — особым образом выделять имя параметра. У него один атрибут — name, который представляет собой имя параметра, выводимого в тексте в особом формате:

```
/// Загружает XML-документацию в файл, заданный
/// <paramref>strFilePath</paramref>.
```

Тэг <see> употребляется в тексте других тэгов комментариев для обозначения гиперссылки (ссылки). Он вставляется как часть текста и обычно включает один атрибут, cref:

```
/// Чтобы выполнить преобразование и вернуть результат,
/// вызывается одна из функций-членов
/// (<see cref="GiveTypeListHTMLHelp"/>,
```

```
/// <see cref="GiveMemberListHTMLHelp"/>,
/// <see cref="GiveMemberHTMLHelp"/>).
```

Атрибут cref — ссылка на существующий символ. Более подробную информацию см. в описании тэга <description>.

Тэг <value> определяет значение какого-либо свойства. Он аналогичен тэгу <remarks>, описывающему классы, и тэгу <summary>, описывающему остальные элементы.

```
/// <value>
/// Свойство SourceXML содержит XML, используемый
/// функциями-членами этого класса при преобразованиях.
/// </value>
```

Файл XML-документа

Я уже пояснил, как настроить проект для экспорта XML-комментариев. На рис. 4 приведен фрагмент файла XML-документа, созданного после компиляции моего проекта-примера. Если вы сравните его с исходными XML-комментариями (рис. 2), то заметите, что некоторые из экспортированных комментариев были отформатированы.

Тэги, приведенные в табл. 1, требуют при обработке особого форматирования. Почти все эти тэги форматируются раскрытием атрибута cref. То есть синтаксический анализатор C# изменяет исходное значение атрибута cref: во-первых, определяет классификацию значения (например, является ли оно типом или свойством), во-вторых, раскрывает значение в полное имя. Атрибут cref со значением «MyMember» (т. е. cref=«MyMember») после раскрытия синтаксическим анализатором может выглядеть так:

```
M:MyProject.MyType.MyMember
```

M: является классификатором. Компилятор, помещая в XML-документацию ссылки на элементы, классифицирует их одним из пяти способов. Допустимые классификаторы показаны в табл. 2.

Другие части раскрытого имени из предыдущего примера тоже значимы. «MyNamespace.MyType.MyMember» — это полное имя (fully qualified name), определенное синтаксическим анализатором C#. Имя можно разбивать на части разделителем (точкой). Каждая часть имеет четкий смысл. MyName-

Табл. 1. Результаты обработки тэгов

Тэг	Результат
<exception>	Раскрывается атрибут cref
<permission>	Раскрывается атрибут cref
<seealso>	Раскрывается атрибут cref
<include>	Тэги ссылки на включаемые файлы копируются в файл документации
<see>	Раскрывается атрибут cref

Табл. 2. Классификаторы

Значение	Применяется для:	Префикс
Тип	Классов, делегатов	T:
Поле	Переменных-членов	F:
Метод	Процедур и функций	M:
Свойство	Свойств	P:
Событие	Событий	E:
Неизвестное	Ссылки, которую не удалось раскрыть	!:

Табл. 3. Примеры cref

Значение атрибута cref	Раскрытое имя
cref="MyClass"	"T:MyProject.MyClass"
cref="MyField"	"F:MyProject.MyClass.MyField"
cref="MyMethod"	"M:MyProject.MyClass.MyMethod"
cref="MyProperty"	"E:MyProject.MyClass.MyProperty"
cref="MyEvent"	"P:MyProject.MyClass.MyEvent"
cref="MyUnknown"	"!:MyUnknown"

спаре представляет пространство имен, MyType — тип, а MyMember является именем элемента (члена). Если раскрыть значение cref не удастся, компилятор ставит перед исходным значением cref префикс «!»: (табл. 3).

Такие классификаторы применяются не только при раскрытии атрибутов cref. Как вы вскоре убедитесь, в XML-документации этот формат используется и для имен элементов.

Вот как выглядит базовая высокоуровневая структура XML-документа:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Name</name>
  </assembly>
  <members>
    <member name="name">
      Сюда помещаются XML-комментарии
    </member>
    ...
  </members>
</doc>
```

Первая строка — стандартный заголовок, за которым идет корневой элемент <doc>. Следующий тэг — <assembly>. Под ним находятся тэги <name>, идентифицирующие имена сборок, созданных из кода проекта. Таким образом, обычно встречается только один элемент — <name>. Значение, подставляемое в него компилятором C#, соответствует имени проекта, и его нельзя изменить с помощью XML в исходном коде.

Далее размещается блок <members>, содержащий записи для всех элементов, с которыми сопоставлены XML-комментарии. Каждый элемент идентифицируется собственным тэгом <member>.

У тэга <member> единственный атрибут — name. Синтаксический анализатор C# генерирует значение этого атрибута в зависимости от имени элемента, с которым связа-

ны тэги комментария. Схему именования я уже пояснил. Основной XML, находящийся внутри тэга <member>, состоит из XML-комментариев.

Генерация справочных файлов

Трудно перечислить все, что можно сделать с документом, в котором содержатся XML-комментарии, но чаще всего на его основе генерируются (с помощью XSLT) страницы справки в формате HTML.

Я создал проект GiveHelp, содержащий простой класс GiveHelpTransforms, несколько ASP-страниц и XSLT для преобразований (исходный код к этой статье можно скачать по адресу <http://download.microsoft.com/download/msdnmagazine/code/June02/WXP/EN-US/xmlc.exe>). Это Web-приложение получает XML-документацию из локального XML-файла и создает справочные файлы. Оно спроектировано так, чтобы XML-документацию можно было получать из любого приложения.

ASP-страницы в этом проекте предназначены для быстрого отображения результатов преобразований в окне браузера. Кроме того, они служат интерфейсом к классу GiveHelpTransforms, позволяя инициировать преобразования XML в HTML.

Класс GiveHelpTransform нужен по двум причинам. Во-первых, XSLT из этого проекта дает возможность создавать инфраструктуру для преобразований XML в HTML в других программах. А во-вторых, мне понадобился класс, в котором я мог бы генерировать XML-документацию для демонстрации этого проекта. Как это ни удивительно, но в моих примерах приложение использует собственную XML-документацию для генерации справочных файлов, поясняющих разработчику, как использовать XML-документацию.

Assembly: GiveHelp

Types

GiveHelpTransforms

Class that contains functions to do transformations to help files. The source XML is loaded into the [SourceXML](#) property (e.g. `obj.SourceXML = "XML goes here"`). One of the associated member functions ([GiveTypeListHTMLHelp](#), [GiveMemberListHTMLHelp\(System.String\)](#), [GiveMemberHTMLHelp\(System.String\)](#)) is called to initiate and then return the transformation.

Help Page	Function to call
List of Types	GiveTypeListHTMLHelp
List of members	GiveMemberListHTMLHelp
Help for a single member	GiveMemberHTMLHelp

GiveTypeHelp

The codebehind source for *GiveTypeHelp.aspx*
This source page gives type level help. Given an input file, it uses XSLT to create the help file.

GiveTypeMemberHelp

The codebehind source for *GiveTypeMemberHelp.aspx*

This source page gives help for a member that belongs to a type in the project. Given an input file, it uses XSLT to create the help file.

GiveTypeMemberListHelp

The codebehind source for *GiveTypeMemberListHelp.aspx*
This source page gives a list of members that belong to a particular type. Given an input file, it uses XSLT to create the help file.

Рис. 5. Список типов

Преобразования

В этом разделе я исхожу из того, что вы уже знаете XSLT. Поэтому рассматриваю только XSLT, используемый для преобразования XML-документа в HTML-страницу.

Я разбил свою справочную систему на 3 уровня. Верхний — это список типов. На этом уровне на HTML-странице (рис. 5) представлены все типы и их описания из XML-документации. Каждое имя типа является ссылкой на страницу, где перечислены все элементы этого типа. Таким образом, средний уровень — это список элементов (с описаниями). Имя элемента представляет собой ссылку на следующий уровень, где даются подробные сведения о выбранном элементе, в том числе описание, разрешения, параметры, исключения, пример и ссылки на дополнительную информацию.

Нужный нам XSLT находится в файле GiveTypeHelp.xsl. Он преобразует XML-комментарии в HTML, который можно просматривать в браузере. Для вывода списка типов в HTML список определений создается парой тэгов <DL>, причем имя типа помещается внутрь тэга <DT>, а его описание — внутрь тэга <DD>. В результате имена типов выводятся по левому полю страницы, а их описания — с отступом.

Классы извлекаются из документа. Помните, что все имена типов содержат префикс «T:». Для их выборки применяется выражение на XSLT:

```
<xsl:apply-templates select="
  ./members/member[starts-with(@name, 'T:')]"/>
```

Эта строка находится в начале блока, который начинает обработку элемента <doc>. Далее присутствует блок, ограниченный выражениями:

```
<xsl:template match="member[starts-with(@name, '.')] ">
...
</xsl:template>
```

Этот блок вызывается тэгом <xsl:apply-templates> из предыдущего фрагмента. Оба выражения указывают XSLT-процессору обрабатывать лишь элементы, являющиеся типами.

Для форматирования и размещения каждого типа на HTML-странице его имя извлекается из паче, при этом должно существовать его описание. На рис. 6 показано, как это делается с помощью тэгов <DT> и <DD>.

Между тэгами <DT> создается ссылка на ASP-страницу, которая генерирует HTML для отображения списка элементов. Между тэгами <DD> обрабатываются тэги <remarks> и <summary>. Что здесь происходит, я объясню в следующем разделе. Результаты показаны на рис. 5.

Список элементов

При выборе типа из списка браузер переадресуется на страницу GiveTypeMemberListHelp.aspx. На ней содержится список всех элементов этого типа. Фрагмент исходного кода XSLT показан на рис. 7.

При запросе страницы GiveTypeMemberListHelp.aspx ей передается параметр, значением которого является полное имя типа (например, /GiveTypeMemberListHelp.aspx?Type=MyNamespase.MyType). Страница извлекает это имя и для преобразования XML в HTML передает его в XSLT-процессор через параметр WhichType.

```
<xsl:param name="WhichType" />
```

Этот параметр используется XSLT-процессором для выборки только тех тэгов <member>, которые описывают элементы данного типа. Вот как это делается в XSLT:

Рис. 6. Извлечение имени типа и описания

```
<DT>
<A>
  <xsl:attribute name="href">
    GiveTypeMemberListHelp.aspx?Type=<xsl:value-of
      select="substring-after(@name, '.')" />
  </xsl:attribute>
  <B>
    <xsl:value-of select="substring-after(@name, '.')" />
  </B>
</A>
</DT>
<DD>
  <xsl:apply-templates select="remarks | summary" />
</DD>
```

Рис. 7. GiveTypeMemberListHelp.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html" />
  <xsl:param name="WhichType" />
  <xsl:template match="doc">
    <xsl:apply-templates select="./assembly/name" />
    <H2>
      Members
    </H2>
    <xsl:apply-templates select="./members/member[starts-
      with(@name,concat('F:', $WhichType))]" mode="fields" />
    <xsl:apply-templates select="./members/member[starts-
      with(@name,concat('P:', $WhichType))]" mode="properties" />
    <xsl:apply-templates select="./members/member[starts-
      with(@name,concat('M:', $WhichType))]" mode="methods" />
    <xsl:apply-templates select="./members/member[starts-
      with(@name,concat('E:', $WhichType))]" mode="events" />
  </xsl:template>
  <xsl:template match="name">
    <TABLE width="100%" border="1" cellspacing="0" cellpadding="10"
      bgcolor="lightskyblue">
      <TR>
        <TD align="center">
          <H1>
            <xsl:text>Assembly: </xsl:text>
            <xsl:value-of select="." />
            <BR />
            <xsl:text>Type: </xsl:text>
            <xsl:value-of select="substring-after($WhichType, '.')" />
          </H1>
        </TD>
      </TR>
    </TABLE>
  </xsl:template>
  <xsl:template match="member" mode="methods">
    <xsl:if test="position() = 1">
      <P></P>
      <H3>Methods</H3>
    </xsl:if>
    <DL>
      <DT>
        <A>
          <xsl:attribute name="href">
            GiveTypeMemberListHelp.aspx?
              Member=<xsl:value-of select="substring-after(@name, '.')" />
          </xsl:attribute>
          <B>
            <xsl:value-of select="substring-
              after(@name,concat($WhichType, '.'))" />
          </B>
        </A>
      </DT>
      <DD>
        <xsl:apply-templates select="remarks | summary" />
      </DD>
    </DL>
    <P></P>
  </xsl:template>
  ...
```

```

<xsl:apply-templates select="./members/member[starts-
with(@name,concat('F:',$WhichType))]" mode="fields" />
<xsl:apply-templates select="./members/member[starts-
with(@name,concat('P:',$WhichType))]" mode="properties" />
<xsl:apply-templates select="./members/member[starts-
with(@name,concat('M:',$WhichType))]" mode="methods" />
<xsl:apply-templates select="./members/member[starts-
with(@name,concat('E:',$WhichType))]" mode="events" />

```

В частности, этот код вызывает шаблоны для полей, свойств, методов и событий. Используя XPath, я применяю эти шаблоны к элементам <member>, чьи атрибуты name удовлетворяют образцу. Таким образом (pattern) служит формат, который вы уже видели:

```
*:Namespace.Type.Member
```

Так как шаблоны для каждого элемента очень похожи, я рассмотрю только шаблон для методов из GiveTypeMemberListHelp.xsl (рис. 7). Первый тэг, <xsl:if>, проверяет, является ли данный элемент первым. Если да, записывается заголовок. Блок кода между тэгами <DT> создает ссылку на страницу с более подробной информацией об этом элементе. Блок кода между тэгами <DD> извлекает информацию из блока

<summary> или <remarks> и обрабатывает ее. В следующем разделе я поясню, как обрабатываются тэги <summary> и <remarks>.

Элемент типа

Файл GiveTypeMemberHelp.xsl содержит XSLT, рассматриваемый в этом разделе. Как и уровень списка элементов, этот уровень принимает один параметр. Кроме типа передается полное имя:

```

<xsl:param name="WhichMember" />
Тип извлекается из имени элемента:
<xsl:param name="WhichType" select="concat(concat( substring-
before($WhichMember,'.'), '.'), substring-before(substring-
after($WhichMember,'.'), '.'))" />
Сложновато, но работает.
Обработка XSLT начинается со строк:
<xsl:template match="doc">
  <xsl:apply-templates select="./assembly/name" />
  <xsl:apply-templates select="./members/
member[contains(@name,$WhichMember)]" />
</xsl:template>

```

Рис. 8. Фрагмент файла GiveTypeMemberHelp.xsl

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="html" />
  <xsl:param name="WhichMember" />
  <xsl:param name="WhichType" select="concat(concat( substring-
before($WhichMember,'.'), '.'), substring-before(substring-
after($WhichMember,'.'), '.'))" />
  <xsl:template match="doc">
    <xsl:apply-templates select="./assembly/name" />
    <xsl:apply-templates select="./members/
member[contains(@name,$WhichMember)]" />
  </xsl:template>
  <xsl:template match="name">
    ...
  </xsl:template>
  <xsl:template match="member[contains(@name,$WhichMember)]">
    <H2>
      <xsl:choose>
        <xsl:when test="starts-with(@name,'M:')">
          Method
        </xsl:when>
        <xsl:when test="starts-with(@name,'F:')">
          Field
        </xsl:when>
        <xsl:when test="starts-with(@name,'P:')">
          Property
        </xsl:when>
        <xsl:when test="starts-with(@name,'E:')">
          Event
        </xsl:when>
      </xsl:choose>
    </H2>
    <TABLE width="100%" border="1" cellspacing="0" cellpadding="10"
bgcolor="silver">
      <TR>
        <TD>
          <B>
            <xsl:value-of select="substring-
after($WhichMember,concat($WhichType,'.'))" />
          </B>
        </TD>
      </TR>
    </TABLE>
    <P></P>
    <B>Description:</B>
    <DL>
      <xsl:apply-templates select="remarks" />
    </DL>
  </xsl:template>

```

```

    <xsl:apply-templates select="summary" />
  </DL>
  <DL>
    <xsl:apply-templates select="value" />
  </DL>
  <B>Permission:</B>
  <DL>
    <xsl:apply-templates select="permission" />
  </DL>
  <xsl:if test="not(starts-with(@name,'F:'))">
    <B>Parameters:</B>
    <DL>
      <xsl:apply-templates select="param" />
    </DL>
  </xsl:if>
  <xsl:if test="not(starts-with(@name,'E:')) and not(starts-
with(@name,'P:'))">
    <B>Returns:</B>
    <DL>
      <xsl:apply-templates select="returns" />
    </DL>
  </xsl:if>
  <B>Exceptions:</B>
  <DL>
    <xsl:apply-templates select="exception" />
  </DL>
  <B>Example:</B>
  <DL>
    <xsl:apply-templates select="example" />
  </DL>
  <B>See Also:</B>
  <DL>
    <xsl:apply-templates select="seealso" />
  </DL>
  </xsl:template>
  <xsl:template match="summary | remarks | value">
    <DT></DT>
    <DD>
      <xsl:apply-templates />
    </DD>
  </xsl:template>
  ...
  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates />
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
...

```

Табл. 4. HTML-эквиваленты XML-комментариев	
Тэг XML-комментария в .NET	HTML-эквивалент
<c>	<CODE>
<para>	<P>
<paramref>	<I>
<see>	<A>
<list type="table">	<TABLE>
<list type="bullet">	
<list type="number">	
<listheader> (в таблицах)	<THEAD>
<listheader> (в списках)	
<item> (в заголовках таблиц)	<TR>[<TH>]
<item> (в таблицах)	<TR>[<TD>]
<item> (в списках)	
<term> (в заголовках таблиц)	<TH>
<term> (в таблицах)	<TD>
<term> (в списках)	К значению тэга добавляется «-»
<description> (в заголовках таблиц)	<TH>
<description> (в таблицах)	<TD>
<description> (в списках)	Нет
<code>	<PRE>
*	Прочие тэги просто копируются

В обработке элементов <name> нет ничего замысловатого. Второй элемент <xsl:apply-templates> начинает обработку члена, переданного в таблицу стилей через переменную \$WhichMember. В следующей строке (она взята с **рис. 8**) обрабатываются XML-комментарии, относящиеся к выбранному члену:

```
<xsl:template match="member[contains(@name,$WhichMember)]">
```

Почти весь этот код должен быть знаком тем, кто уже работает с XSLT.

Вместе со вставкой различных заголовков обрабатываются и первичные тэги, относящиеся к данному члену. Это делается с помощью элементов вроде <xsl:apply-templates select="summary" /> и <xsl:apply-templates select="param" />, которые присутствуют в GiveTypeMemberHelp.xsl.

Обработка тэгов

После начального форматирования, например вставки заголовков и задания типа выравнивания, таблица стилей указывает XSLT-процессору отформатировать первичные и вспомогательные тэги XML-комментариев. Этот процесс и его результаты одинаковы для всех таблиц стилей из моего проекта.

В разных таблицах стилей обрабатываются разные первичные тэги. В списках типов и элементов (членов) обрабатываются только тэги <summary> и <remarks>. На уровне сведений об элементе обрабатываются все первичные тэги. Эта обработка начинается с выражений выборки, имеющих вид <xsl:apply-templates select="primary tag"/>, где "primary tag" заменяется на имя первичного тэга.

Выполнив эти выражения, процессор обрабатывает блок, который выглядит примерно так:

Рис. 9. Фрагмент HTML-кода, полученного из GiveHelpDoc.xml

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- saved from url=(0122)http://localhost/GiveHelp/
GiveTypeMemberHelp.aspx?Member=GiveHelp.GiveHelpTransforms.
GiveMemberListHTMLHelp(System.String) -->
<HTML>
  <HEAD>
    <META http-equiv="Content-Type" content="text/html; charset=utf-8">
    <META content="MSHTML 6.00.2462.0" name="GENERATOR">
    <META content="C#" name="CODE_LANGUAGE">
    <META content="JavaScript (ECMAScript)"
      name="vs_defaultClientScript">
    <META content="http://schemas.microsoft.com/intellisense/ie5"
      name="vs_targetSchema">
  </HEAD>
  <BODY>
    <TABLE cellSpacing="0" cellPadding="10" width="100%"
      bgColor="lightskyblue" border="1">
      <TBODY>
        <TR>
          <TD align="middle">
            <H1>
              Assembly: GiveHelp
            <BR>
              Type: GiveHelpTransforms
            </H1>
          </TD>
        </TR>
      </TBODY>
    </TABLE>
    <H2>
      Method
    </H2>
    <TABLE cellSpacing="0" cellPadding="10" width="100%"
      bgColor="silver" border="1">
      <TBODY>
        <TR>
          <TD>
            <B>GiveMemberListHTMLHelp(System.String)</B>
          </TD>
        </TR>
      </TBODY>
    </TABLE>
    <P>
    </P>
    <B>Description:</B>
    <DL>
    </DL>
    <DL>
      <DT>
        <DD>
          Calling this function will take the XML in
          <A href="http://localhost/GiveHelp/
          GiveTypeMemberHelp.aspx?Type=GiveHelp.
          GiveHelpTransforms&Member=GiveHelp.
          GiveHelpTransforms.SourceXML"> SourceXML</A>
          and translate it to a list of
          Members in the specified type.
        </DD>
      </DT>
    </DL>
    <DL>
    </DL>
    <B>Permission:</B>
    <DL>
      <DT>
        <DD>
          <DL>
            <DT><A href="http://localhost/GiveHelp/
              GiveTypeHelp.aspx?Type="></A>
            <DD>
              public
            </DD>
          </DL>
        </DD>
      </DT>
    </DL>
```

см. след. стр.

Рис. 9. Фрагмент HTML-кода, полученного из GiveHelpDoc.xml (продолжение)

```

        </DL>
    </DD>
</DT>
</DL>
<B>Parameters:</B>
<DL>
    <DT>
        <DD>
            <DL>
                <DT><I>strType</I>
                <DD>
                    The fully qualified name of the type that
                    the member is in.
                </DD>
            </DL>
        </DD>
    </DL>
</B>Returns:</B>
<DL>
    <DT>
        <DD>
            The HTML that lists the types that are in the
            XML documentation.
        </DD>
    </DT>
</DL>
<B>Exceptions:</B>
<DL>
</DL>
<B>Example:</B>
<DL>
    <DT>
        <DD>
            <PRE> // создать класс, выполняющий преобразования

            GiveHelpTransforms ght = new GiveHelpTransforms();

```

```

// Загрузить XML в свойство SourceXML
ght.LoadXMLFromFile(
    "E:\\Inetpub\\wwwroot\\GiveHelp\\GiveHelpDoc.xml");
// Выполнить преобразование и записать строку
Response.Write(
    ght.GiveMemberListHTMLHelp(
        Request.QueryString.Get("Type")));
</PRE>
</DD>
</DT>
</DL>
<B>See Also:</B>
<DL>
    <DT>
        <DD>
            <A href="http://localhost/GiveHelp/
                GiveTypeMemberHelp.aspx?Type=GiveHelp.
                GiveHelpTransforms&amp;Member=
                GiveHelp.GiveHelpTransforms.
                GiveTypeListHTMLHelp">
                GiveHelpTransforms.GiveTypeListHTMLHelp</A> <DT></DT>
        <DD>
            <A href="http://localhost/GiveHelp/
                GiveTypeMemberHelp.aspx?Type=GiveHelp.
                GiveHelpTransforms&amp;Member=GiveHelp.
                GiveHelpTransforms.GiveMemberHTMLHelp(
                    System.String)">
                GiveHelpTransforms.GiveMemberHTMLHelp(
                    System.String)
            </A>
        </DD>
    </DT>
</DL>
</BODY>
</HTML>

```

```
<xsl:template match="primary tag">
```

```
...
```

```
<xsl:apply-templates />
```

```
...
```

```
</xsl:template>
```

В нем создаются HTML-теги, охватывающие дочерние узлы первичного тэга. У них всех есть одна общая особенность — вызов `<xsl:apply-templates />`. С этого вызова начинается обработка вспомогательных тэгов, являющихся дочерними узлами первичных тэгов.

Вспомогательные тэги обычно заменяются HTML-эквивалентами. Вот как выглядит, например, шаблон тэга `<c>`:

```

<xsl:template match="c">
    <CODE>
        <xsl:apply-templates />
    </CODE>
</xsl:template>

```

Таким образом, тэг `<c>` заменяется HTML-тэгом `<CODE>`. Соответствие между тэгами XML-комментариев и HTML-тэгами показано в табл. 4.

На рис. 9 показан HTML-код, который получен из XML-кода (рис. 4) с применением XSLT-кода, приведенного на рис. 8.

Заключение

Хотя инструменты, предназначенные для выборки комментариев из исходного кода, существовали и раньше, они никогда не находили широкого применения, потому что были слишком сложны и слабо интегрировались с основными системами разработки. Благодаря технологии XML и интеграции с языком и редактором XML-комментарии в C# лишены этих недостатков. Еще большие преимущества XML-комментариев в том, что их можно преобразовывать в другой формат, например HTML.

Дж. Эндрю Шейфер (J. Andrew Schafer) — консультант по разработке решений в нью-йоркской компании Avanade Inc. (<http://www.avanade.com>). С ним можно связаться по адресу aschafer@bigfoot.com.

ОСНОВЫ И ТОНКОСТИ

Кен Спенсер

Просмотр значений DataSet в отладочном окне

Вопрос Читая вашу колонку за апрель, я понял, что мне нужно просматривать значения DataSet в отладочном окне. Проход по всему дереву для поиска значения конкретной ячейки отнимает слишком много времени. Нет ли способа попроще?

Ответ Для демонстрации одного из подходов к решению этой проблемы я взял программу, созданную мной для апрельской колонки, слегка ее доработал и добавил кое-какой отладочный код (см. обработчик события Form_Load на **рис. 1**).

Допустим, вы создаете DataSet по одной или нескольким таблицам, как в коде на **рис. 1**. Затем, тестируя приложение, вы видите, что в DataGridView никаких значений нет. Что делать?

Есть несколько вариантов. Самый простой из известных мне — исследовать DataSet и его значения, используя либо Command Window, либо Debug.Write. Можно также раскрыть DataSet в окне Autos или в других отладочных окнах для доступа к отдельному значению. Сначала рассмотрим вариант с отладочным окном.

Я поставил точку прерывания (breakpoint) в обработчик Form_Load (**рис. 1**) в строку, где DataSet получает набор значений. Потом запустил программу и прогнал ее до этой строки, а затем выполнил и саму строку. Теперь я мог обращаться к объекту DataSet. Тут-то и начинается самое интересное. DataSet — объект очень сложный, с обширной иерархией свойств и наборов (collections). Чтобы найти то его место, где находятся требуемые значения, придется потратить много времени на просмотр иерархии, но эта задача решаема. Результаты, которые я получил в окне Autos, показаны на **рис. 2**.

Следующая строка кода дает представление о глубине иерархии, через которую мне пришлось пройти, чтобы найти

данные каждого из столбцов. Первый ноль в иерархии указывает на таблицу с нулевым номером, а последний — на первое поле первой записи (вы видите его почти в самом верху окна на **рис. 2**).

```
ds.Tables.Tables.List_1.Items(0).list_1.Items(0).ItemArray(0)
```

Преимущество такого режима просмотра в том, что вам доступны все данные и можно раскрывать и свертывать то одну, то другую запись. А минус — можно забыть эту иерархию, и тогда в следующий раз вам не удастся быстро повторить все действия в том же порядке.

Чтобы познакомиться с другим подходом, взгляните на Command Window. Для использования Command Window я вставил точку прерывания в строку, где создается DataSet (или же в него добавляется новая таблица). Например, в коде на **рис. 1** точка прерывания находится в строке:

```
ds = RunSQLWithDataSet(sSQL, "employees")
```

Код RunSQLWithDataSet показан на **рис. 3**. Когда программа после запуска останавливается на этой строке, для ее выполнения нажмите клавишу F10, а затем переключитесь в Command Window. Предположим, вас интересует, сколько записей в таблице. Тогда в Command Window можно ввести команду:

```
>? ds.tables(0).rows.count
```

Команда сообщит, что число записей равно 9, а значит, какие-то данные в DataSet есть. Теперь посмотрим на конкретные ячейки (поля), о чем спрашивал читатель. Один из способов сделать это — снова воспользоваться Command Window и просто ввести в него команду:

```
>? ds.tables(0).rows.item(2).item(0)
```

Как видите, она выводит данные первого поля третьей записи. Взгляните на следующую последовательность команд, введенную мной для просмотра полей первой записи*:

```
? ds.tables(0).rows.item(0).item(0)
1 {Integer}
[Integer]: 1 {Integer}
```

* Эта последовательность команд (в том виде, в каком она показана здесь) выводит первое поле первой записи, а также третье и четвертое поля третьей записи. Взгляните также на **рис. 2** или **рис. 5** и вы убедитесь, что из первой записи действительно извлекается лишь первое поле. — Прим. перев.

Простейший вариант — исследовать DataSet и его значения, используя либо Command Window, либо Debug.Write. Можно также раскрыть объект DataSet в одном из отладочных окон для доступа к отдельному значению.

Рис. 1. Код обработчика Form_Load

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim ds As DataSet
    Dim rw As DataRow
    Dim cl As DataColumn
    Dim sSQL As String
    sSQL = "select * from employees"
    ds = RunSQLWithDataSet(sSQL, "Employees")
    DataGridView1.DataSource = ds.Tables("Employees")
    lblTraceListener.Text = Debug.Listeners.Item(0).Name
    Debug.WriteLine("-----")
    Debug.WriteLine("Employee DS Data")
    For Each rw In ds.Tables("Employees").Rows
        Debug.Write(rw.Item("EmployeeID") & vbTab)
        Debug.Write(rw.Item("LastName") & vbTab)
        Debug.Write(rw.Item("FirstName") & vbTab)
        Debug.Write(rw.Item("Title") & vbTab)
        Debug.WriteLine(rw.Item("BirthDate"))
    Next
    Debug.WriteLine("-----")
```

Name	Value	Type
Item	<cannot view indexed property>	Object
Item	<cannot view indexed property>	Object
ItemArray	{Length=18}	Object()
(0)	1 {Integer}	Integer
(1)	"Davolio"	String
(2)	"Nancy"	String
(3)	"Sales Representative"	String
(4)	"Ms."	String
(5)	#12/8/1948#	Date
(6)	#5/1/1992#	Date
(7)	"507 - 20th Ave. E. □ □ Apt. 2A"	String
(8)	"Seattle"	String
(9)	"WA"	String
(10)	"98122"	String
(11)	"USA"	String
(12)	"(206) 555-9857"	String
(13)	"5467"	String
(14)	{Length=21626}	Byte()
(15)	"Education includes a BA in psychology from Colorado State University in 19"	String
(16)	2 {Integer}	Integer
(17)	"http://accweb/emmployees/davolio.bmp"	String
HasErrors	False	Boolean
(1)	{System.Data.DataRow}	System.Data.DataRow
(2)	{System.Data.DataRow}	System.Data.DataRow
(3)	{System.Data.DataRow}	System.Data.DataRow
(4)	{System.Data.DataRow}	System.Data.DataRow
(5)	{System.Data.DataRow}	System.Data.DataRow
(6)	{System.Data.DataRow}	System.Data.DataRow
(7)	{System.Data.DataRow}	System.Data.DataRow
(8)	{System.Data.DataRow}	System.Data.DataRow
(9)	Nothing	Object
(10)	Nothing	Object
(11)	Nothing	Object
(12)	Nothing	Object

Рис. 2. Иерархия данных

```
>> ds.tables(0).rows.item(2).item(2)
"Janet" {String}
String: "Janet"
>> ds.tables(0).rows.item(2).item(3)
"Sales Representative" {String}
String: "Sales Representative"
>
```

В данных примерах я указывал порядковые номера столбцов и строк, но при желании можно использовать и имена полей:

```
>> ds.tables(0).rows.item(2).item("Title")
```

Наверное, вы сейчас думаете, как же запомнить всю иерархию объектов, используемую в командах. Но есть способ облегчить себе жизнь. Когда при отладке приходится часто об-

ращаться к свойствам DataSet, команды можно сохранить. Я, например, взял некоторые из ранее показанных команд и вставил их в исходный код программы как комментарии. Теперь, если мне нужна какая-нибудь команда, я просто копирую ее в Command Window:

```
' Команды отладки DataSet:
'? ds.tables(0).rows.count
'? ds.tables(0).rows.item(0).item(0)
```

Будут ли эти команды полезны во многих приложениях? Видимо, да. Чтобы упростить их использование, я скопировал закомментированные команды в ToolBox и переименовал созданный элемент в «DS Debug Cmds — Command Window». Теперь я могу перетащить команды в любую программу и вставить их обратно в Command Window. Кстати, когда элементам ToolBox присваиваешь описательные имена, их потом гораздо легче находить.

Рассмотрим еще один способ решения проблемы. Можно создать процедуру, отображающую значения нескольких столбцов (см. отладочный раздел в коде на рис. 1). В ней начало и конец вывода очерчиваются пунктирными линиями. В Output Window выводится пять значений. Вы просто запускаете программу, содержащую этот код, и смотрите нужные значения в Output Window. Это лучше, чем снова и снова выполнять команды в Command Window; кроме того, данный код остается в программе в течение всего процесса отладки.

А как насчет того, чтобы обобщить такой подход? Например, неплохо бы иметь отдельную функцию, принимающую DataSet и показывающую все значения определенной таблицы в Output Window. Я написал функцию (рис. 4), которая как раз это и делает. Вы можете поместить мою функцию в один из модулей проекта и для просмотра полей таблицы вызывать ее так:

Рис. 3. RunSQLWithDataSet

```
Public Function RunSQLWithDataSet(ByVal sSQL As String, _
Optional ByVal sTable As String = "Table1") As DataSet
Dim oDataSet As New DataSet()
Dim oDataAdapter As SqlClient.SqlDataAdapter

Try
' Создаем DataAdapter
oDataAdapter = New _
SqlClient.SqlDataAdapter( _
sSQL, ConnectionString)

' Заполняем DataSet из DataAdapter
oDataAdapter.Fill(oDataSet, sTable)

' Возвращаем значение функции
Return oDataSet
Catch e As Exception
Throw New Exception( _
"An exception occurred in RunSQLWithDataSet", e)
Finally
End Try
End Function
```

Рис. 4. ShowDSInOutWindow

```

Function ShowDSInOutWindow(ByVal ds As DataSet, _
    Optional ByVal sTable As String = "")

' Универсальная функция для отладки DataSet
Dim cldebug As DataColumn
Dim tbdebug As DataTable
Dim rw As DataRow

If sTable = "" Then
    tbdebug = ds.Tables(0)
Else
    tbdebug = ds.Tables(sTable)
End If

Debug.WriteLine("-----")
Debug.WriteLine(vbTab & vbTab & "DS Table: " & _
    tbdebug.ToString())
Dim bFirst As Boolean = True

For Each rw In tbdebug.Rows
    If bFirst Then
        For Each cldebug In tbdebug.Columns
            If cldebug.DataType.Name <> "Byte[]" Then
                Debug.Write( _
                    cldebug.ColumnName & vbTab)
            End If
        Next cldebug
        Debug.WriteLine("")
        bFirst = False
    End If

    For Each cldebug In tbdebug.Columns
        If cldebug.DataType.Name <> "Byte[]" Then
            Debug.Write(rw.Item(cldebug) & vbTab)
        End If
    Next cldebug
    Debug.WriteLine("")
Next
Debug.WriteLine("-----")
End Function

```

```

' Вывод таблицы в Output Window
ShowDSInOutWindow(ds, "Employees")

```

Образец вывода функции для первых трех записей показан на рис. 5.

Теперь можно легко просмотреть данные в каждом столбце. И даже преобразовать их в таблицу Microsoft Word или скопировать в электронную таблицу вроде Microsoft Excel для дополнительного анализа. Допустим, вы выполняете какие-то вычисления и вам нужно их проверить. Тогда воспользуйтесь функцией ShowDSInOutWindow, чтобы получить данные, загрузите их в Excel и проанализируйте все вычисления.

На рис. 4 видно, что для обработки таблицы DataSet применяются вложенные циклы For Each. В первом внутреннем цикле For Each выводятся имена столбцов, а во втором внутреннем цикле For Each обрабатываются поля каждой строки. Внешний цикл For Each перебирает все строки.

Кроме того, обратите внимание на код, который выводит имя таблицы:

```
Debug.WriteLine(vbTab & vbTab & "DS Table: " & tbdebug.ToString())
```

Если вы присваиваете таблицам описательные имена, то с помощью этой процедуры вам будет несложно отлаживать свои таблицы.

Рис. 5. Образец вывода

```

-----
DS Table: Employees
ran at 3/30/2002 5:42:02 PM
EmployeeID    LastName    FirstName    Title    TitleOfCourtesy
BirthDate    HireDate    Address    City    Region    PostalCode
Country    HomePhone    Extension    Notes    ReportsTo    PhotoPath
1    Davolio    Nancy    Sales Representative    Ms.    12/8/1948    5/1/
1992    507 - 20th Ave. E.
Apt. 2A    Seattle    WA    98122    USA    (206) 555-9857    5467
Education includes a BA in psychology from Colorado State University in
1970. She also completed "The Art of the Cold Call." Nancy is a member
of Toastmasters International.    2    http://accweb/employees/
davolio.bmp
2    Fuller    Andrew    Vice President, Sales    Dr.    2/19/1952    8/
14/1992    908 W. Capital Way    Tacoma    WA    98401    USA    (206)
555-9482    3457    Andrew received his BTS commercial in 1974 and a
Ph.D. in international marketing from the University of Dallas in 1981.
He is fluent in French and Italian and reads German. He joined the
company as a sales representative, was promoted to sales manager in
January 1992 and to vice president of sales in March 1993. Andrew is a
member of the Sales Management Roundtable, the Seattle Chamber of
Commerce, and the Pacific Rim Importers Association.    http://
accweb/employees/fuller.bmp
3    Leverling    Janet    Sales Representative    Ms.    8/30/1963    4/
1/1992    722 Moss Bay Blvd.    Kirkland    WA    98033    USA    (206)
555-3412    3355    Janet has a BS degree in chemistry from Boston
College (1984). She has also completed a certificate program in food
retailing management. Janet was hired as a sales associate in 1991 and
promoted to sales representative in February 1992.    2    ttp://accweb/
employees/leverling.bmp
-----

```

Заключение

Ответ на вопрос месяца осветил несколько моментов. Во-первых, отладочные окна в Visual Studio .NET позволяют исследовать данные какого-либо объекта в период выполнения программы. Во-вторых, окно Command Window можно использовать для более глубокого изучения данных в период выполнения. Для экономии времени при повторных проверках команды лучше сохранять. Наконец, можно создавать подпрограммы, автоматизирующие выполнение этих задач, и тем самым формировать библиотеки инструментов, которые упрощают отладку программ.

Эту функциональность легко расширить. Например, вам нужно исследовать значения столбцов и в готовом приложении. Здесь возможно несколько вариантов. Скажем, вывести DataSet как XML. Или использовать вместо класса Debug класс Trace: вывод Trace можно перехватывать при выполнении приложения. Несложно определить и ключ, который будет включать или отключать трассировку в файл или другое хранилище данных. Разумеется, трассировочные данные можно выводить и в журнал событий.

Исходный код для этой статьи см. по адресу <http://download.microsoft.com/download/msdnmagazine/code/July02/WXP/EN-US/Advanced-Basics0207.exe>

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 программного обеспечения и обучением, а также предоставляет консалтинговые услуги по технологиям Microsoft.

Сериализация в период выполнения, часть 2*

В предыдущей статье из этой рубрики я объяснил, как при сериализации пользоваться двумя форматирующими объектами (formatters), включенными в Framework Class Library (FCL), и как определять сериализуемые типы. В этой колонке я продолжу описание сериализации, но сосредоточусь на более сложных приемах, которые позволяют при определении типа точно указывать, как именно он будет сериализован и десериализован себя.

Попутно мы спроектируем тип, требующий полного контроля за своей сериализацией и десериализацией. System.Collections.Hashtable — как раз такой тип. При его сериализации

В случае Hashtable вы должны сначала десериализовать все его объекты, а потом вручную добавить их в Hashtable, используя новое значение хэш-кода для каждого объекта.

в поток должны быть записаны объект Hashtable и все объекты, на которые тот ссылается. А при десериализации надо сконструировать новый объект Hashtable и все управляемые им объекты; кроме того, следует корректно настроить все ссылки на объекты. Проблема в том, что для вновь десериализованных объектов не гарантируется иден-

тичность хэш-кодов. Поэтому в случае Hashtable вы должны сначала десериализовать все его объекты, а потом вручную добавить их в Hashtable, используя новое значение хэш-кода для каждого объекта.

Чтобы тип полностью контролировал свою сериализацию и десериализацию, он должен реализовать интерфейс System.Runtime.Serialization.ISerializable, определяемый как:

```
public interface ISerializable {
    void GetObjectData(SerializationInfo, StreamingContext context);}
```

В нем всего один метод, но в любом типе, в котором используется этот интерфейс, реализуется два метода: GetObjectData и специальный конструктор.

Форматирующий объект, сериализуя связанную цепочку объектов (object graph), проверяет тип каждого объекта. Если тип реализует интерфейс ISerializable, тогда форматирующий объект создает новый объект System.Runtime.Serialization.SerializationInfo — он содержит истинный набор значений, которые должны быть сериализованы для объекта.

Создавая SerializationInfo, форматирующий объект передает два параметра: Type и System.Runtime.Serialization.IFor-

matterConverter. Первый идентифицирует объект, подлежащий сериализации. Для однозначной идентификации типа нужны строковое имя типа и сведения о его сборке (в том числе имя сборки, версия, культура и открытый ключ). При создании объект SerializationInfo получает полное имя типа (внутренним обращением к Type-свойству FullName) и сохраняет эту строку в закрытом поле. Вы можете узнать полное имя типа через свойство FullTypeName объекта SerializationInfo. Аналогичным образом конструктор получает информацию о сборке для данного типа (последовательно обращаясь к Type-свойству Module, Module-свойству Assembly и Assembly-свойству FullName) и тоже сохраняет эту строку в закрытом поле. Вы можете выяснить идентификацию сборки через свойство AssemblyName объекта SerializationInfo.

Хотя вы можете сами настраивать свойства FullTypeName и AssemblyName объекта SerializationInfo, я не советую этого делать. Если вы хотите изменить сериализуемый тип, лучше вызвать метод SetType объекта SerializationInfo, передав ссылку на нужный объект Type. Вызов SetType гарантирует правильность задания полного имени типа и его сборки. Потом я покажу, как вызывать SetType.

Как только объект SerializationInfo сконструирован и инициализирован, форматирующий объект вызывает метод GetObjectData типа, передавая ссылку на объект SerializationInfo. Этот метод отвечает за определение того, какая информация нужна для сериализации объекта, и за ее добавление в объект SerializationInfo. GetObjectData указывает, какую информацию следует сериализовать, вызывая один из перегруженных методов AddValue, предоставляемых типом SerializationInfo. AddValue вызывается по разу для каждой порции данных, которую вы хотите добавить.

Код на **рис. 1** демонстрирует, как тип Hashtable реализует интерфейсы ISerializable и IDeserialization для контроля за сериализацией и десериализацией своих объектов. Каждый метод AddValue принимает имя String и какие-либо данные. Обычно эти данные относятся к простому типу вроде Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal или Date/Time. Кроме того, вы можете вызывать AddValue, передавая ему ссылку на такой Object, как String. После того как GetObjectData добавляет всю информацию, необходимую для сериализации, он возвращает управление форматирующему объекту.

Для добавления информации о сериализации своего типа всегда используйте перегруженные методы AddValue. Если тип какого-то поля реализует интерфейс ISerializable, никогда не вызывайте GetObjectData применительно к этому полю. Вместо этого добавьте поле вызовом AddValue; форматирую-

* Продолжение. Часть 1 читайте в MSDN Magazine/Русская Редакция. 2002. Спецвыпуск №2.

щий объект увидит, что тип данного поля реализует `ISerializable`, и сам вызовет `GetObjectData`. А если это сделаете вы, форматирующий объект может не узнать о том, что при десериализации потока он должен сначала создать новый объект.

Далее форматирующий объект берет все значения, добавленные в `SerializationInfo`, и сериализует их в байтовый поток. Вы заметите, что методу `GetObjectData` передается еще один параметр — ссылка на объект `System.Runtime.Serialization.StreamingContext`. Методы `GetObjectData` большинства типов напрочь игнорируют этот параметр, но об этом — позже.

Теперь, когда вы знаете, как задать всю информацию, используемую при сериализации, перейдем к десериализации. Извлекая какой-либо объект из байтового потока, форматирующий объект выделяет память под новый объект (статическим методом `GetUninitializedObject`, принадлежащим типу `System.Runtime.Serialization.FormatterServices`). Изначально все поля этого объекта содержат 0 или `null`. Форматирующий объект проверяет, реализует ли данный тип интерфейс `ISerializable`. Если да, он пытается вызвать специальный конструктор, чьи параметры идентичны параметрам метода `GetObjectData`.

Если ваш класс «опечатан» (указано ключевое слово `sealed`), настоятельно советуем объявить этот конструктор закрытым. Это предотвратит случайный вызов конструктора каким-нибудь кодом. А если ваш класс открыт, тогда объявите конструктор защищенным, чтобы его могли вызывать только производные классы. Кстати, в исполняющей среде есть один внутренний механизм, который позволяет вызывать специальный конструктор независимо от того, как он объявлен.

Специальный конструктор принимает ссылку на объект `SerializationInfo`, содержащий все значения, добавленные в него при сериализации. Он может вызвать любой из методов `GetXxx` (`GetBoolean`, `GetChar`, `GetByte`, `GetSByte`, `GetInt16`, `GetUInt16`, `GetInt32`, `GetUInt32`, `GetInt64`, `GetUInt64`, `GetSingle`, `GetDouble`, `GetDecimal`, `GetDateTime`, `GetString` или `GetValue`), передавая строковое имя значения. Потом значения, возвращаемые этими методами, используются для инициализации полей нового объекта. Когда конструктор возвращает управление, создание объекта должно быть закончено.

Десериализуя поля объекта, вы должны вызывать тот метод `Get`, который подходит типу значения, переданного методу `AddValue` при сериализации этого объекта. Иными словами, если метод `GetObjectData` вызывал `AddValue` с передачей ему значения типа `Int32`, то при десериализации объекта следует вызвать метод `GetInt32`. Если тип значения в потоке не совпадает с тем, что вы пытаетесь получить, форматирующий объект пробует привести значение в потоке к требуемому типу через `IFormatterConvert`.

Как я уже говорил, при создании объекту `SerializationInfo` передается объект, тип которого реализует интерфейс `IFormatterConverter`. Поскольку форматирующий объект отвечает за создание объекта `SerializationInfo`, он выбирает такой `IFormatterConverter`, какой ему нужен. Типы `BinaryFormatter` и `SoapFormatter`, предоставляемые Microsoft, всегда создают экземпляр типа `System.Runtime.Serialization.FormatterConverter`. Форматирующие объекты от Microsoft не предусматривают иных способов выбора `IFormatterConverter`.

Типу `FormatterConverter` известно, как преобразовывать значения между базовыми типами, например `Int32` в `Int64`. Для таких преобразований `FormatterConverter` приводит сериализованный (или исходный) тип к интерфейсу `IConvertible`,

Рис. 1. Сериализация и десериализация

```
[Serializable]
public class Hashtable : ISerializable, IDeserializationCallback {
    // Здесь должны быть внутренние поля Hashtable (не показаны)

    SerializationInfo _savedSI; // только при десериализации

    // Метод, управляющий сериализацией Hashtable.
    // Советую использовать здесь Explicit Interface Method Impl.
    void ISerializable.GetObjectData(
        SerializationInfo info, StreamingContext context) {

        // Вызовы info.AddValue для того, чтобы вручную добавить
        // большинство полей объекта Hashtable в поток при
        // сериализации (не показаны)

        // Создаем массив Object для объектов
        // key/value объекта Hashtable
        Object[] keys = new Object[countOfEntries];
        Object[] values = new Object[countOfEntries];

        // Заполняем массивы ссылками на объекты
        // key/value объекта Hashtable (не показано)

        // Добавляем объекты key/value в поток
        info.AddValue("Keys", keys);
        info.AddValue("Values", values);
    }

    // Специальный конструктор (нужен интерфейсу ISerializable)
    // для управления десериализацией Hashtable
    protected Hashtable(
        SerializationInfo info, StreamingContext context) {

        // Сохраняем поток. Нам нужно отложить заполнение объекта
        // Hashtable до тех пор, пока не завершится десериализация
        // объектов key/value, так как значение хэш-кода для объектов
        // key скорее всего повлияет на их поля.
        _savedSI = info;
    }

    // Метод, вызываемый после десериализации всех объектов key/value.
    // Советую использовать здесь Explicit Interface Method Impl.
    void IDeserializationCallback.OnDeserialization(Object sender) {

        // Все объекты в потоке десериализованы.
        // Теперь можно заполнить объект Hashtable.

        // Вызываем _savedSI.GetValue, чтобы вручную инициализировать
        // большинство полей объекта Hashtable из потока (не показано)

        // Получаем объекты key/value из потока
        Object[] keys = _savedSI.GetValue("Keys");
        Object[] values = _savedSI.GetValue("Values");

        // Добавляем в Hashtable каждую пару key/value
        for (Int32 x = 0; x < keys.Length; x++) {
            Add(keys[x], values[x]);
        }

        // Записываем в это поле null, чтобы сборщик мусора
        // мог утилизировать объект SerializationInfo
        _savedSI = null;
    }
}
```

а затем вызывает подходящий метод этого интерфейса. Чтобы спокойно десериализовать объекты какого-либо сериализуемого типа в объекты некоего другого типа, возможно, вам стоит подумать о соответствующей реализации в интерфейсе `IConvertible`. Объект `FormatterConverter` используется только при десериализации объектов и при вызове метода `Get`, тип которого не совпадает с типом значения в потоке.

Вместо вызова разных методов `Get` специальный конструктор мог бы вызывать `GetEnumerator`, который возвращает объект `System.Runtime.Serialization.SerializationInfoEnumera-`

tor — он перебирает все значения, содержащиеся в объекте `SerializationInfo`. Каждое перечисленное им значение является объектом `System.Runtime.Serialization.SerializationEntry`.

Конечно, вы вправе определить собственный тип как производный от типа, реализующего `ISerializable`-метод `GetObjectData` и специальный конструктор. Если ваш тип также реализует `ISerializable`, тогда ваши варианты `GetObjectData` и специального конструктора должны вызывать те же функции в базовом классе, чтобы сериализация и десериализация объекта проходила корректно (не забывайте об этом, иначе вам не удастся должным образом сериализовать и десериализовать свои объекты). Позже я поясню, как правильно определить тип `ISerializable`, чей базовый класс не реализует этот интерфейс.

Если в вашем производном типе нет дополнительных полей и поэтому нет нужды в особой сериализации/десериализации, вам вообще не требуется реализовать `ISerializable`. Как и все члены интерфейса, `GetObjectData` является виртуальным методом и вызывается для должной сериализации объекта. Кроме того, формирующий объект рассматривает специальный конструктор как «виртуализированный» («virtualized»). То есть при десериализации формирующий объект будет проверять тип, экземпляр которого он пытается создать. Если в этом типе нет специального конструктора, формирующий объект будет сканировать все базовые классы, пока не найдет один из них с таким конструктором.

В типе могут быть поля, ссылающиеся на другие объекты. Вызов специального конструктора гарантирует правильность

Рис. 2. ISerializable

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;
using System.Reflection;

///////////////////////////////////////////////////

[Serializable]
class Person {
    private String title;

    public Person(String title) {
        this.title = title;
    }

    public override String ToString() {
        return String.Format("{0}", title);
    }
}

///////////////////////////////////////////////////

[Serializable]
class Employee : Person {
    private String title;

    public Employee(String title) : base("Person") {
        this.title = title;
    }

    public override String ToString() {
        return String.Format("{0} -> {1}", title, base.ToString());
    }
}

///////////////////////////////////////////////////

[Serializable]
class Manager : Employee, ISerializable {
    private String title;

    public Manager() : base("Employee") {
        this.title = "Manager";
    }

    void ISerializable.GetObjectData(
        SerializationInfo info, StreamingContext context) {

        // Сериализуем нужные значения для этого класса
        info.AddValue("title", title);

        // Получаем набор сериализуемых элементов
        // для нашего класса и для базовых классов
        Type thisType = this.GetType();
        MemberInfo[] mi =
            FormatterServices.GetSerializableMembers(thisType, context);

        // Сериализуем поля базового класса в объект info
        for (Int32 i = 0 ; i < mi.Length; i++) {

            // Не сериализовать поля для этого класса
            if (mi[i].DeclaringType == thisType) continue;
            info.AddValue(mi[i].Name, ((FieldInfo) mi[i]).GetValue(this));
        }

        protected Manager(SerializationInfo info, StreamingContext context) :
            base(null) {

            // Получаем набор сериализуемых элементов
            // для нашего класса и для базовых классов
            Type thisType = this.GetType();
            MemberInfo[] mi = FormatterServices.GetSerializableMembers(
                thisType, context);

            // Десериализуем поля базового класса из объекта info
            for (Int32 i = 0 ; i < mi.Length; i++) {
                // Не десериализовать поля для этого класса
                if (mi[i].DeclaringType == thisType) continue;

                // Чтобы упростить кодирование, считаем
                // данный член объектом FieldInfo
                FieldInfo fi = (FieldInfo) mi[i];

                // Инициализируем поле десериализованным значением
                fi.SetValue(this, info.GetValue(fi.Name, fi.FieldType));
            }

            // Десериализуем значения, сериализованные для этого класса
            title = info.GetString("title");
        }

        public override String ToString() {
            return String.Format("{0} -> {1}", title, base.ToString());
        }
    }

    ///////////////////////////////////////////////////

    class App {
        public static void Main() {
            FileStream stream = new FileStream(@"Data.dat", FileMode.Create);
            IFormatter formatter = new SoapFormatter();
            Manager m = new Manager();
            Console.WriteLine(m.ToString());
            formatter.Serialize(stream, m);
            stream.Close();

            stream = new FileStream(@"Data.dat", FileMode.Open);
            m = (Manager) formatter.Deserialize(stream);
            stream.Close();
            Console.WriteLine(m.ToString());
        }
    }
}
```

Табл. 1. Свойства StreamingContext

Свойство	Тип	Описание
State	Streaming-ContextStates	Набор битовых флагов, указывающих источник или приемник сериализуемых/десериализуемых объектов
Context	Object	Ссылка на объект, который содержит любую пользовательскую информацию о контексте

инициализации таких полей. Иначе говоря, их значения будут ссылками на созданные объекты. Так как поля этих объектов, возможно, еще не инициализированы, в специальном конструкторе не должно быть кода, который обращается к элементам подобного объекта.

Если ваш тип должен обращаться к элементам типа, на который он ссылается (например, вызывать его методы), тогда рекомендуется реализовать и метод OnDeserialization интерфейса IDeserializationCallback. Когда вызывается этот метод, все объекты уже инициализировали свои поля. Но выяснить, в каком порядке они вызывали свой метод OnDeserialization, нельзя. А значит, хотя поля инициализированы, вы все равно не узнаете, закончена ли десериализация объекта, на который ссылается ваш тип, если этот объект реализует интерфейс IDeserializationCallback.

Код для сериализации и десериализации Hashtable на **рис. 1** иллюстрирует эти механизмы.

Табл. 2. Значения битовых флагов

Флаг	Значение	Описание
CrossProcess	0x0001	Источник или приемник — другой процесс на той же машине
CrossMachines	0x0002	Источник или приемник — на другой машине
File	0x0004	Источник или приемник — файл; данные могут быть десериализованы другим процессом
Persistence	0x0008	Источник или приемник — некое хранилище вроде базы данных, файла и т. д.; данные могут быть десериализованы другим процессом
Remoting	0x0010	Источник или приемник — механизм Remoting в неизвестном месте, которое может находиться на той же или на другой машине
Other	0x0020	Источник или приемник не известен
Clone	0x0040	Цепочка объектов клонируется; десериализация данных будет выполняться в том же процессе, поэтому сериализация описателей и других неуправляемых ресурсов безопасна
CrossAppDomain	0x0080	Источник или приемник — другой AppDomain
All	0x00FF	Источник или приемник может находиться в любом контексте, перечисленном выше (контекст по умолчанию)

ISerializable

Интерфейс ISerializable открывает широчайшие возможности, позволяя типу полностью контролировать сериализацию и десериализацию его экземпляров. Однако за эти возможности приходится расплачиваться: теперь тип отвечает за сериализацию и всех полей своего базового типа. Сериализовать поля базового типа несложно, если этот тип тоже реализует интерфейс ISerializable (вы просто вызываете метод GetObjectData базового типа). Но когда-то вам может понадобиться тип, который полностью контролирует свою сериализацию и чей базовый тип не реализует интерфейс ISerializable. Тогда ваш класс должен сам сериализовать поля базового типа. На **рис. 2** показано, как реализовать метод GetObjectData интерфейса ISerializable и конструктор для сериализации полей базового типа.

В исходном коде на **рис. 2** базовый класс Person помечен лишь пользовательским (custom) атрибутом SerializableAttribute. От Person наследует Employee, который тоже помечен SerializableAttribute. От Employee наследует Manager, помеченный тем же атрибутом и реализующий интерфейс ISerializable. Чтобы ситуация стала еще интереснее, во всех трех классах определено закрытое поле title типа String. Вызывая метод AddValue объекта SerializationInfo, вы не можете добавить несколько значений под одинаковым именем. Мой код выходит из этой ситуации, идентифицируя каждое поле с добавлением имени его класса. Например, когда метод GetObjectData объекта Manager вызывает AddValue для сериализации поля title объекта Employee, имя значения записывается как «Employee+title».

Кроме того, каждый класс переопределяет метод ToString, заставляя его отображать иерархию полей title. Метод Main обращается к ToString перед сериализацией и после десериализации: это позволяет сравнить исходный вывод программы с конечным и убедиться в правильности сериализации и десериализации полей title.

Потоковые контексты

Цепочку объектов можно сериализовать, адресуя байтовый поток самым разнообразным получателям: тому же процессу, другому процессу на той же машине, другому процессу на другой машине и т. д. В некоторых (весьма редких) ситуациях объект должен знать, куда именно он сериализуется, чтобы соответственно изменить свой статус. Так, объект — оболочка объекта ядра «семафор», зная, что он будет десериализован в том же процессе, мог бы разрешать сериализацию своего описателя объекта ядра. Определив, что десериализация предполагается в другом процессе, но на той же машине, он мог бы сериализовать строковое имя семафора, а при десериализации в любом процессе на другой машине — просто генерировать исключение.

Многие из уже упомянутых методов, в том числе GetObjectData и SetObjectData, принимают в качестве одного из параметров потоковый контекст. Соответствующая структу-

Интерфейс ISerializable открывает широчайшие возможности, позволяя типу полностью контролировать сериализацию и десериализацию его экземпляров.

ра StreamingContext очень проста и предоставляет два открытых свойства только для чтения (табл. 1).

Метод, принимающий структуру StreamingContext, может проверить битовые флаги в свойстве State и определить источник или приемник сериализуемых/десериализуемых объектов. Возможные значения битовых флагов перечислены в табл. 2.

Теперь, когда вы знаете, как получить эту информацию, я расскажу, как задать ее. Интерфейс IFormatter (реализуемый типами BinaryFormatter и SoapFormatter) определяет StreamingContext-свойство Context, доступное для чтения и записи. При создании форматирующий объект инициализирует собственное свойство Context так, чтобы установить StreamingContextStates в All, а ссылку на дополнительный объект состояния (additional state object) — в null.

Создав форматирующий объект, вы можете сформировать структуру StreamingContext, используя любой из битовых флагов StreamingContextState, и (необязательно) передать

Метод, принимающий структуру StreamingContext, может проверить битовые флаги в свойстве State и определить источник или приемник сериализуемых/десериализуемых объектов.

ссылку на объект, содержащий любую нужную вам дополнительную информацию о контексте. После этого, прежде чем вызывать метод Serialize или Deserialize форматирующего объекта, остается лишь настроить его свойство Context на новый объект StreamingContext. Как сообщить форматирующему объекту, что цепочка объектов сериализуется и десериализуется с

единственной целью клонировать все объекты в этой цепочке, я уже показывал в исходном коде метода DeepClone (см. мою статью в предыдущей колонке).

Сериализация и десериализация с преобразованием типов

Инфраструктура сериализации в .NET Framework открывает широчайшие возможности, и в этом разделе мы обсудим, как спроектировать тип, способный сериализовать или десериализовать себя в другой тип или объект. (Примеров тому — уйма.)

Некоторые типы (вроде System.DBNull и System.Reflection.Missing) допускают создание только одного экземпляра в каждом AppDomain. Такие типы часто называются единичными (singletons). Если у вас есть массив с 10 элементами и каждый элемент ссылается на объект DBNull, сериализация и десериализация элементов этого массива не должна приводить к появлению 10 новых объектов DBNull в AppDomain. После десериализации массива каждый элемент должен ссылаться на уже существующий в данном AppDomain объект DBNull.

Для других типов (например, System.Type, System.Reflection.Assembly и прочих типов отражения наподобие MemberInfo) можно создавать по одному экземпляру на тип, на сборку, на член и т. д.

В случае удаленно управляемых объектов CLR сериализует информацию о серверном объекте, который при десериализации на клиенте заставляет CLR создавать прокси-объект. Тип этого прокси-объекта отличается от типа серверного объ-

екта, но транспарентен клиентскому коду. Когда клиент вызывает методы экземпляра применительно к прокси-объекту, внутренний код последнего перенаправляет вызовы на сервер, который и обрабатывает их.

Рис. 3 иллюстрирует, как правильно сериализовать и десериализовать единичный тип. Тип, допускающий создание только одного своего экземпляра в AppDomain, представлен классом Singleton. Код на рис. 4 предназначен для проверки кода сериализации и десериализации в Singleton.

Когда тип Singleton загружается в AppDomain, CLR вызывает статический конструктор, который создает объект Singleton и сохраняет ссылку на него в статическом поле theOneObject. Класс Singleton не предоставляет никаких открытых конструкторов, благодаря чему никто не сможет создать еще один экземпляр этого класса.

В SingletonSerializationTest создается массив с двумя элементами, каждый из которых ссылается на объект Singleton. Элементы инициализируются вызовом статического метода GetSingleton объекта Singleton. Этот метод возвращает ссылку на один объект Singleton. Первый вызов метода WriteLine объекта Console приводит к появлению на экране строки «True», подтверждающей, что оба элемента массива ссылаются на один и тот же объект.

Далее SingletonSerializationTest вызывает метод Serialize форматирующего объекта для сериализации массива и его

Рис. 3. Сериализация и десериализация Singleton

```
// В AppDomain должен быть только один экземпляр этого типа
[Serializable]
public sealed class Singleton : ISerializable {
    // Это единственный экземпляр данного типа
    private static readonly Singleton theOneObject = new Singleton();

    // Это поля экземпляра
    public String someString;
    public Int32 someNumber;

    // Закрытый конструктор, позволяющий данному типу создать Singleton
    private Singleton() {
        // Здесь делаем все, что нужно для инициализации Singleton
        someString = "This is a string field";
        someNumber = 123;
    }

    // Метод, возвращающий ссылку на Singleton
    public static Singleton GetSingleton() {
        return theOneObject;
    }

    // Метод, вызываемый при сериализации Singleton.
    // Советую использовать здесь Explicit Interface Method Impl.
    void ISerializable.GetObjectData(
        SerializationInfo info, StreamingContext context) {

        info.SetType(typeof(SingletonSerializationHelper));
        // Добавлять другие значения не требуется
    }

    // ПРИМЕЧАНИЕ: специальный конструктор здесь не нужен,
    // потому что он вообще не вызывается
}

[Serializable]
internal sealed class SingletonSerializationHelper : IObjectReference {
    // У этого объекта нет полей (хотя они могли бы быть)

    // Метод, вызываемый после десериализации этого объекта
    public Object GetRealObject(StreamingContext context) {
        return Singleton.GetSingleton();
    }
}
```

Рис. 4. Проверка кода Singleton

```

static void SingletonSerializationTest(Stream stream) {
    // Создаем массив с двумя элементами,
    // которые ссылаются на один объект Singleton
    Singleton[] a1 = {
        Singleton.GetSingleton(),
        Singleton.GetSingleton()
    };

    Console.WriteLine(
        "Do both array elements refer to the same object? " +
        (a1[0] == a1[1]));    // выводит на экран "True"

    // Сериализуем элементы массива
    formatter.Serialize(stream, a1);

    // Десериализуем элементы массива
    stream.Position = 0;
    Singleton[] a2 = (Singleton[]) formatter.Deserialize(stream);

    Console.WriteLine(
        "Do both array elements refer to the same object? " +
        (a2[0] == a2[1]));    // выводит на экран "True"

    Console.WriteLine(
        "Do all array elements refer to the same object? " +
        (a1[0] == a2[0]));    // выводит на экран "True"
}

```

элементов. Сериализуя первый Singleton, формирующий объект обнаруживает, что тип Singleton реализует интерфейс ISerializable, и вызывает метод GetObjectData. Тот обращается к SetType, передавая тип SingletonSerializationHelper, который указывает формирующему объекту сериализовать объект Singleton как объект SingletonSerializationHelper. Здесь AddValue не вызывается, и информация о дополнительных полях в байтовый поток не записывается.

Так как формирующий объект автоматически распознает, что оба элемента массива ссылаются на один объект, он и сериализует только один объект.

После сериализации массива SingletonSerializationTest вызывает метод Deserialize формирующего объекта. Обработывая поток, формирующий объект пытается десериализовать объект SingletonSerializationHelper, потому что ранее его «обманом» заставили сериализовать именно этот объект. (В сущности, это и является причиной отсутствия в классе Singleton специального конструктора, который обычно требуется для реализации интерфейса ISerializable.) Создав объект SingletonSerializationHelper, формирующий объект видит, что этот

тип реализует интерфейс System.Runtime.Serialization.IObjectReference, определенный в FCL так:

```

public interface IObjectReference {
    Object GetRealObject(StreamingContext context);
}

```

Если тип реализует этот интерфейс, формирующий объект вызывает метод GetRealObject. Данный метод возвращает ссылку как раз на тот объект, который вам и нужен после десериализации. В моем примере метод GetRealObject в типе SingletonSerializationHelper возвращает ссылку на объект Singleton, уже существующий в AppDomain. Поэтому, когда метод Deserialize формирующего объекта возвращает управление, в массиве a2 содержатся два элемента, каждый из которых ссылается на объект Singleton в данном AppDomain. А объект SingletonSerializationHelper, использовавшийся при десериализации во вспомогательных целях, тут же становится недоступным; при сборе мусора он будет уничтожен.

Второй вызов WriteLine тоже выводит «True», подтверждая, что оба элемента массива a2 ссылаются на один и тот же объект. Третий (и последний) вызов WriteLine вновь выводит «True» — это свидетельствует о том, что все элементы в обоих массивах ссылаются на один объект.

Заключение

На этот раз я показал вам, как определять типы, способные полностью контролировать сериализацию и десериализацию своих экземпляров. В следующей статье (последней из трех), которая будет опубликована в этой рубрике, мы продолжим обсуждение сериализации и сосредоточимся на том, как переопределить в программе метод сериализации того или иного типа. В частности я расскажу о приемах, которые позволяют сериализовать типы, не рассчитанные на сериализацию, а также поясню, как десериализовать объект в тип другой версии.

Джеффри Рихтер (Jeffrey Richter) — автор книг «Applied Microsoft .NET Framework Programming» (Microsoft Press, 2002)* и «Programming Applications for Microsoft Windows» (Microsoft Press, 1999)**. Он является одним из основателей компании Wintellect (<http://www.Wintellect.com>) и специализируется в области программирования для .NET и Win32.

* Перевод этой книги вскоре будет издан «Русской Редакцией» под названием «Программирование на платформе Microsoft .NET Framework». — *Прим. перев.*

** Эта книга выпущена издательствами «Русская Редакция» и «Питер» в 2000 г. под названием «Windows для профессионалов» (4-е издание) — *Прим. перев.*

Нахождение основного окна и получение имени EXE-модуля

Вопрос Как найти основное окно процесса? Я пишу утилиту наподобие Spy и хочу получить основное окно (HWND) процесса, чтобы послать ему какое-нибудь сообщение вроде WM_ACTIVATEAPP.

Джон Ризгинс (John Riggins)

Ответ А о каком основном окне вы говорите? У процесса может быть не одно такое окно. Пример — Microsoft Outlook. Запустив эту программу, вы увидите окно, где перечисляются Inbox и другие папки. Если вы откроете почтовое сообщение, появится еще одно окно, в котором будет показано само сообщение. Оба окна основные — в том смысле, что у них нет ни предка, ни владельца. Выхватив из ножен свой верный Spy, вы заметите, что даже класс у них один и тот же: `rctrl_renwnd32`. В общем, по флагам стилей или по имени класса узнать, какое из основных окон «настоящее», нельзя.

А найти основные окна нетрудно. Вызывая `EnumWindows` и `GetWindowThreadProcessId`, ищите все окна верхнего уровня с идентификатором интересующего вас процесса. Я написал небольшой класс `CMainWindowIterator`, который делает как раз то, что вам нужно (**рис. 1**).

```
DWORD pid = // идентификатор какого-то процесса
CMainWindowIterator itw(pid);
for (HWND hwnd = itw.First(); hwnd; hwnd=itw.Next()) {
    // Что-то делаем
}
```

Проще некуда! `CMainWindowIterator` наследуется от более универсального класса `CWindowIterator`, который служит оболочкой `::EnumWindows` в C++, скрывая неуклюжий обратный вызов. В `CWindowIterator` есть виртуальная функция `OnWindow`; ее можно переопределять в производных классах для перечисления окон по заданным вами критериям. `OnWindow` в `CMainWindowIterator` сообщает только о тех окнах, которые принадлежат указанному процессу.

```
// (m_pid настраивается в ctor)
BOOL CMainWindowIterator::OnWindow(HWND hwnd)
{
    if ((GetWindowLong(hwnd, GWL_STYLE) & WS_VISIBLE)) {
        DWORD pidwin;
        GetWindowThreadProcessId(hwnd, &pidwin);
        if (pidwin==m_pid)
            return TRUE;
    }
    return FALSE;
}
```

Иначе говоря, окно является основным в каком-либо процессе, если у него нет предка (само собой, раз `CWindow-`

`Iterator` использует `::EnumWindows`), если оно видимо и если у него идентификатор нужного процесса. Проверка на `WS_VISIBLE` важна потому, что некоторые приложения создают кучу невидимых окон верхнего уровня. (Те, кого интересует пример программы, работающей с `CMainWindowIterator`, пусть читают дальше.) Если вы пользуетесь .NET, то можете получить описатель основного окна, принадлежащий объекту `Process`, из свойства `Process.MainWindowHandle`. Например, на C#:

```
Process p = // получаем объект процесса;
IntPtr hwnd = p.MainWindowHandle;
```

Очевидно, что где-то глубоко внутри Windows все же знает, какое окно действительно основное. Увы, я так и не нашел, как сделать то же самое на C/C++.

Вопрос Как найти имя файла программы (EXE), сопоставленного с другим процессом? Я перепробовал все комбинации `GetModuleFileName`, `GetModuleInstance` и `GetModuleHandle`, но, похоже, ни одна из них не срабатывает.

Фрэнк Уэстон (Frank Weston)

Ответ Проблема с этими функциями в том, что они годятся только для модулей, загруженных в текущий выполняемый процесс, а не в какой-то другой. Для вашей задачи нужно нечто иное, и здесь появляется развилка. Если вы пишете для Windows NT 4.0, Windows 2000 или Windows XP, можете использовать PSAPI, сравнительно новую DLL с функциями, которые сообщают информацию о процессах и модулях. Если же вы пишете для Windows 95, Windows 98 или Windows Me, у вас нет другого выбора, кроме ToolHelp. Раз уж я весь из себя такой современный, покажу, как это делается через PSAPI. Насчет ToolHelp читайте Q175030 «HOWTO: Enumerate Applications in Win32» в MSDN.

Одна из функций в PSAPI — `GetModuleFileNameEx`. Она принимает описатели процесса и модуля и возвращает имя модуля. А как узнать, какой из модулей в процессе является тем EXE, который и породил этот процесс? Есть еще одна PSAPI-функция, `EnumProcessModules`, которая заполняет переданный ей массив описателями всех модулей в процессе.

У процесса может быть более одного основного окна. Если вы запустите Outlook, то увидите окно, где перечисляются Inbox и другие папки. После того как вы откроете почтовое сообщение, появится еще одно окно. И оба окна основные — в том смысле, что у них нет ни предка, ни владельца.

EnumProc.h

```

////////////////////////////////////
// MSDN Magazine - июль 2002
// Если этот код работает, его написал Пол Дилация,
// а если нет, то я не знаю, кто его писал.
// Компилируется в Visual Studio 6.0 и в Visual Studio .NET.
// Работает в Windows XP и, наверное, в Windows 2000.
// ПРИМЕЧАНИЕ: для экономии места приведен не весь исходный код;
// полную версию см. в Интернете (адрес в конце статьи).

// Перебирает окна верхнего уровня. Инкапсулирует ::EnumWindows.
class CWindowIterator {
protected:
    HWND* m_hwnds;      // массив с hwnd для данного PID
    DWORD m_nAlloc;     // размер массива
    DWORD m_count;      // число найденных HWND
    DWORD m_current;    // текущий HWND
    static BOOL CALLBACK EnumProc(HWND hwnd, LPARAM lp);

    // виртуальный перечислитель
    virtual BOOL OnEnumProc(HWND hwnd);
    // переопределяется, чтобы отфильтровывать определенные виды окон
    virtual BOOL OnWindow(HWND hwnd) {return TRUE;}

public:
    CWindowIterator(DWORD nAlloc=1024);
    ~CWindowIterator();

    DWORD GetCount() { return m_count; }
    HWND First();
    HWND Next() {
        return m_hwnds && m_current < m_count ? m_hwnds[m_current++] : NULL;
    }
};

// Перебираем окна верхнего уровня в процессе
class CMainWindowIterator : public CWindowIterator {
protected:
    DWORD m_pid;        // идентификатор процесса
    virtual BOOL OnWindow(HWND hwnd);
public:
    CMainWindowIterator(DWORD pid, DWORD nAlloc=1024);
    ~CMainWindowIterator();
};

// Итератор (перечислитель) системных процессов.
// Всегда пропускает первый процесс (IDLE) с PID=0.
class CProcessIterator {
protected:
    DWORD* m_pids;      // массив идентификаторов процессов
    DWORD m_count;      // размер массива
    DWORD m_current;    // следующий элемент массива
public:
    CProcessIterator();
    ~CProcessIterator();
    DWORD GetCount() { return m_count; }
    DWORD First();
    DWORD Next() {
        return m_pids && m_current < m_count ? m_pids[m_current++] : 0;
    }
};

// Перечисляет модули в процессе.
// Первым всегда является главный EXE.
class CProcessModuleIterator {
protected:
    HANDLE m_hProcess;  // описатель процесса
    HMODULE* m_hModules; // массив описателей модулей
    DWORD m_count;      // размер массива
    DWORD m_current;    // описатель следующего модуля
public:
    CProcessModuleIterator(DWORD pid);
    ~CProcessModuleIterator();
    HANDLE GetProcessHandle() { return m_hProcess; }
    DWORD GetCount() { return m_count; }
    HMODULE First();
    HMODULE Next() {

```

```

        return m_hProcess && m_current < m_count ? m_hModules[m_current++]
        : 0;
    }
};

```

EnumProc.cpp

```

////////////////////////////////////
// MSDN Magazine - июль 2002
// ПРИМЕЧАНИЕ: для экономии места приведен не весь исходный код;
// полную версию см. в Интернете (адрес в конце статьи).
//
#include "EnumProc.h"

CWindowIterator::CWindowIterator(DWORD nAlloc)
{
    assert(nAlloc>0);
    m_current = m_count = 0;
    m_hwnds = new HWND [nAlloc];
    m_nAlloc = nAlloc;
}

CWindowIterator::~CWindowIterator()
{
    delete [] m_hwnds;
}

HWND CWindowIterator::First()
{
    ::EnumWindows(EnumProc, (LPARAM)this);
    m_current = 0;
    return Next();
}

// Статическая: передает результат виртуальной
BOOL CALLBACK CWindowIterator::EnumProc(HWND hwnd, LPARAM lp)
{
    return ((CWindowIterator*)lp)->OnEnumProc(hwnd);
}

// Виртуальная: добавляет HWND в массив,
// если OnWindow это одобряет
BOOL CWindowIterator::OnEnumProc(HWND hwnd)
{
    if (OnWindow(hwnd)) {
        if (m_count < m_nAlloc)
            m_hwnds[m_count++] = hwnd;
    }
    return TRUE; // продолжаем перебор
}

CMainWindowIterator::CMainWindowIterator(DWORD pid, DWORD nAlloc)
: CWindowIterator(nAlloc)
{
    m_pid = pid;
}

CMainWindowIterator::~CMainWindowIterator()
{
}

// Переопределение виртуальной: является ли это окно
// основным в процессе?
BOOL CMainWindowIterator::OnWindow(HWND hwnd)
{
    if (GetWindowLong(hwnd, GWL_STYLE) & WS_VISIBLE) {
        DWORD pidwin;
        GetWindowThreadProcessId(hwnd, &pidwin);
        if (pidwin==m_pid)
            return TRUE;
    }
    return FALSE;
}

CProcessIterator::CProcessIterator()

```

см. след. стр.

Рис. 1. CMainWindowIterator (продолжение)

```

{
    m_pids = NULL;
}

CProcessIterator::~CProcessIterator()
{
    delete [] m_pids;
}

// Получаем первый процесс: вызывая EnumProcesses,
// находим все процессы, а возвращаем только первый
DWORD CProcessIterator::First()
{
    m_current = (DWORD)-1;
    m_count = 0;
    DWORD nalloc = 1024;
    do {
        delete [] m_pids;
        m_pids = new DWORD [nalloc];
        if (EnumProcesses(m_pids, nalloc*sizeof(DWORD), &m_count)) {
            m_count /= sizeof(DWORD);
            m_current = 1; // внимание: пропускаем процесс IDLE с pid=0
        }
    } while (nalloc <= m_count);

    return Next();
}

CProcessModuleIterator::CProcessModuleIterator(DWORD pid)
{
    m_hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
        PROCESS_VM_READ,
        FALSE, pid);
}

CProcessModuleIterator::~CProcessModuleIterator()
{
    CloseHandle(m_hProcess);
    delete [] m_hModules;
}

// Получаем первый модуль: вызывая EnumProcessModules,
// находим все модули, а возвращаем только первый
HMODULE CProcessModuleIterator::First()
{
    m_count = 0;
    m_current = (DWORD)-1;
    m_hModules = NULL;
    if (m_hProcess) {
        DWORD nalloc = 1024;
        do {
            delete [] m_hModules;
            m_hModules = new HMODULE [nalloc];
            if (EnumProcessModules(m_hProcess, m_hModules,
                nalloc*sizeof(DWORD), &m_count)) {
                m_count /= sizeof(HMODULE);
                m_current = 0;
            }
        } while (nalloc <= m_count);
    }
    return Next();
}

```

Первый элемент этого массива всегда относится к главному модулю, поэтому вы пишете:

```

DWORD count;
HMODULE hm[1];
EnumProcessModules(hProcess, hm, 1, &count);

```

и получаете первый HMODULE, а потом вызываете GetModuleFileNameEx. Чтобы продемонстрировать все это на практике, я написал небольшую программу, lp, которая перечисляет процессы с именами их модулей и основными окнами (**рис. 2**). Образец ее вывода представлен на **рис. 3**. (Заметьте: и у Windows Explorer, и у Outlook имеется по два «основных» окна.) Программа lp использует третью PSAPI-функцию, EnumProcesses, чтобы перечислить все выполняемые процессы, но, следуя той же логике, что и в ответе на предыдущий вопрос (в случае с CWindowIterator и CMainWindowIterator), я инкапсулирую весь кошмар функций EnumProcesses и EnumProcessModules в два класса-итератора, CProcessIterator и CProcessModuleIterator (**табл. 1** и **рис. 1**). Благодаря им исходный код lp упрощается. Перечисляйте процессы так:

```

CProcessIterator itp;
for (DWORD pid=itp.First(); pid;
    pid=itp.Next()) {
    // Обрабатываем каждый процесс
}

```

А вот как получить имя EXE, создавшего процесс:

```

CProcessModuleIterator itm(pid);
HMODULE hModule = itm.First(); // .EXE
TCHAR modname[_MAX_PATH];
GetModuleBaseName(itm.GetProcessHandle(),
    hModule, modname, _MAX_PATH);

```

Так как lp сообщает лишь имя самого файла модуля (без пути к нему), я использую вместо GetModuleFileNameEx функцию GetModuleBaseName. А поскольку для перечисле-

Табл. 1. Удобные итераторы процессов, модулей и окон

Класс-итератор	Описание
CProcessIterator	Перебирает все процессы в системе. Вызывает EnumProcesses из PSAPI.
CProcessModuleIterator	Перебирает все модули в процессе. Первый модуль — это EXE, породивший данный процесс. Вызывает EnumProcessModules из PSAPI.
CWindowIterator	Перебирает все окна верхнего уровня. Это C++-оболочка для ::EnumWindows. Вы можете переопределить OnWindow, чтобы указать собственные критерии отбора окон.
CProcessWindowIterator	Специализированная версия CWindowIterator. Перебирает все основные окна процесса.

ния модулей CProcessModuleIterator сам открывает процесс, мне не приходится вызывать OpenProcess. Обращаясь к CProcessModuleIterator::GetProcessHandle, я получаю описатель уже открытого процесса. Далее lp применяет CMainWindowIterator к каждому процессу, чтобы получить HWND всех основных окон. Большую часть исходного кода lp занимает логика, которая интерпретирует ключи, указанные в командной строке. Например, -s заставляет перечислять имена классов окон, а -t — названия (заголовки) окон.

Смеха ради я переписал lp на C# (**рис. 4**). Исходный код заметно укоротился. И не удивительно: в .NET ведь все гораздо проще. Никакие PSAPI или доморощенные классы-итераторы больше не нужны; в .NET уже есть классы для перечисления процессов и модулей. Так, Process.GetProcesses возвращает массив объектов Process — все процессы в системе. А от каждого объекта Process можно получить его главный модуль и описатели окон.

Рис. 2. Файл lp.cpp

```

////////////////////
// MSDN Magazine - июль 2002
// Если этот код работает, его написал Пол Дилация,
// а если нет, то я не знаю, кто его писал.
// Программа lp перечисляет процессы из командной строки.
// Иллюстрирует, как использовать мои классы-итераторы
// из EnumProc.h и EnumProc.cpp.
// ПРИМЕЧАНИЕ: для экономии места приведен не весь исходный код;
// полную версию см. в Интернете (адрес в конце статьи).
//
#include "EnumProc.h"

#define tpf _tprintf // скрывает уродливые детали

// глобальные ключи
BOOL bClassName=FALSE; // показывает имя класса окна
BOOL bTitle=FALSE;     // показывает название окна
BOOL bBare=FALSE;      // без заголовка

// проверка ключа: / или -
inline BOOL isswitch(TCHAR c) { return c=='/' || c=='-'; }

////////////////////
// Главная точка входа
//
int main(int argc, TCHAR* argv[], TCHAR* envp[])
{
    // Разбираем командную строку.
    // Ключи могут быть указаны в любом порядке.
    for (int i=1; i<argc; i++) {
        if (isswitch(argv[i][0])) {
            for (UINT j=1; j<tcslen(argv[i]); j++) {
                switch(tolower(argv[i][j])) {
                    case 'c': bClassName=TRUE; break;
                    case 't': bTitle=TRUE; break;
                    case 'b': bBare=TRUE; break;
                    case '?': default: return help();
                }
            }
        } else { return help(); }
    }

    // Перебираем все процессы
    int count=0;
    BOOL bFirstModule = TRUE;
    CProcessIterator itp;
    for (DWORD pid=itp.First(); pid; pid=itp.Next()) {
        // Примечание: первый модуль в CProcessModuleIterator -
        // EXE этого процесса
        TCHAR modname[_MAX_PATH];
        CProcessModuleIterator itm(pid);
        HMODULE hModule = itm.First(); // .EXE
        if (hModule) {
            GetModuleBaseName(itm.GetProcessHandle(),
                               hModule, modname, _MAX_PATH);
            BOOL bFirstWindow = TRUE;
            CMainWindowIterator itw(pid);
            for (HWND hwnd = itw.First(); hwnd; hwnd=itw.Next()) {
                if (bFirstModule) {
                    if (!bBare) {
                        tpf(_T("PID %-13s%-8s\n"), _T("Module"), _T("HWND"),
                            bClassName ? _T("class name") : _T(""),
                            bClassName && bTitle ? _T("/") : _T(""),
                            bTitle ? _T("title") : _T(""));
                        tpf(_T("-----\n"));
                    }
                    bFirstModule = FALSE;
                }
                if (bFirstWindow) {
                    tpf(_T("%04x %-13s"), pid, modname);
                    bFirstWindow = FALSE;
                } else {
                    tpf(_T("%18s"), _T(" "));
                }
                char classname[256], title[256];
                GetClassName(hwnd, classname, sizeof(classname));
                GetWindowText(hwnd, title, sizeof(title));
                tpf(_T("%08x %-8s\n"), hwnd,
                    bClassName ? classname : _T(""),
                    bTitle ? title : _T(""));
            }
            bFirstWindow || count++;
        }
    }

    if (!bBare) {
        tpf(_T("-----\n"));
        tpf(_T("%d processes found.\n"), count);
    }

    return 0;
}

void help() { /* print help message on STDOUT */ }

```

```

Process[] procs = Process.GetProcesses();
int len = procs.GetLength(0);
for (int i=0; i<len; i++) {
    Process p = procs[i];
    if (p.Id!=0) {
        int hwnd = p.MainWindowHandle.ToInt32();
        if (hwnd!=0) { // если основное окно есть:
            ProcessModule pm = p.MainModule;
            String modname = pm.ModuleName;
            ...
        }
    }
}

```

p.MainModule — это главный EXE-модуль (чтобы получить все такие модули, используйте p.Modules, набор ProcessModuleCollection), а p.MainWindowHandle — описатель основного окна (об этом странном, но полезном свойстве см. мои комментарии в конце ответа на предыдущий вопрос). Хотя программа lp куда проще на C#, вся эта лег-

кость в программировании обходится дорого. Когда я запускаю C++-версию lp, результат выводится моментально. А при переходе на .NET ожидание длится целую секунду — даже при 768 Мб оперативной памяти. C# и .NET просто не могут дать того быстродействия, которое присуще C/C++ и API-функциям. Удивляться тут нечему, за удобство всегда

```

C:\>lp /ct
PID  Module      HWND      class name/title
-----
05a8  Explorer.EXE 00030048  Shell_TrayWnd
06b4  4NT.EXE      0001007e  Progman Program Manager
02a0  EPSILON.EXE  000100a2  ConsoleWindowClass 4NT - lp /ct
0bhc  IEFRAME      000302b8  EpsBlankClass qa.txt - Epsilon
0a3c  PHOTO-PAINT  000d020c  IEFrFrame Yahoo! - Microsoft Internet Explorer
0a90  OUTLOOK.EXE  000e036c  PhotoPaint Corel PHOTO-PAINT 9 - lp.bmp
0a90  OUTLOOK.EXE  000201f6  rctrl_renwnd32 Inbox - Microsoft Outlook
0a90  OUTLOOK.EXE  000d0194  rctrl_renwnd32 Untitled - Message <Plain Text>

6 processes found.
C:\>

```

Рис. 3. Образец вывода программы lp

Рис. 4. Файл Ip.cs

```

////////////////////////////////////
// MSDN Magazine - июль 2002
// Если этот код работает, его написал Пол Дилация,
// а если нет, то я не знаю, кто его писал.
// Программа Ip перечисляет процессы из командной строки.
// ПРИМЕЧАНИЕ: для экономии места приведен не весь исходный код;
// полную версию см. в Интернете (адрес в конце статьи).
//

using System;
using System.Text;
using System.Diagnostics;
using System.Runtime.InteropServices;
using Win32API; // оболочка для "родного" API

class MyApp {
    // глобальные ключи командной строки
    static bool bBare = false; // не показывать заголовок
    static bool bClassName = false; // показать имя класса окна
    static bool bTitle = false; // показать название окна

    [STAThread]
    // главная точка входа
    static int Main(string[] args) {
        // Разбираем командную строку.
        // Ключи могут быть указаны в любом порядке.
        for (int i=0, len=args.GetLength(0); i<len; i++) {
            if (args[i].StartsWith("/") || args[i].StartsWith("-") ) {
                for (int j=1; j<args[i].Length; j++) {
                    switch (args[i][j]) {
                        case 'b': bBare = true; break;
                        case 'c': bClassName = true; break;
                        case 't': bTitle = true; break;
                        case '?': default: return help();
                    }
                }
            }
        }

        // Перебираем все процессы.
        // Process.GetProcesses возвращает массив.
        Int32 count=0;

```

```

        Process[] procs = Process.GetProcesses();
        bool bFirstModule=true;
        for (int i=0, len=procs.GetLength(0); i<len; i++) {
            Process p = procs[i];
            if (p.Id!=0) {
                int hwnd = p.MainWindowHandle.ToInt32();
                if (hwnd!=0) { // если основное окно есть:
                    ProcessModule pm = p.MainModule;
                    String modname = pm.ModuleName;
                    if (bFirstModule) {
                        if (!bBare) {
                            Console.WriteLine("PID Module Hwnd
                                {0}{1}{2}",
                                    bClassName ? "class name" : "",
                                    bClassName && bTitle ? "/" : "",
                                    bTitle ? "title" : "");
                            Console.WriteLine("----");
                        }
                        bFirstModule=false;
                    }
                    ~StringBuilder cname = new StringBuilder(256);
                    Win32.GetClassName(hwnd, cname, cname.Capacity);
                    StringBuilder title = new StringBuilder(256);
                    Win32.GetWindowText(hwnd, title, title.Capacity);

                    Console.WriteLine("{0:x4} {1}{2:x8} {3} {4}",
                        p.Id, modname.PadRight(13),
                        hwnd,
                        bClassName ? cname.ToString() : "",
                        bTitle ? title.ToString() : "");
                    count++;
                }
            }
        }
        if (!bBare) {
            Console.WriteLine("----");
            Console.WriteLine("{0} processes found.", count);
        }
        return count;
    }
    static void help() { /* Console.WriteLine(help message) */ }
}

```

приходится платить. Конечно, ни один программист в здравом уме не станет использовать .NET для написания консольного приложения — разве что в учебных целях. Так применять .NET — это вроде как слетать за хлебом на космическом челноке. Вы когда-нибудь обращали внимание на слово «net» (сеть) в .NET? Оно указывает, что .NET нацелена на Web-приложения, где пара миллиардов лишних процессорных тактов ничего не значит (какие могут быть счета между друзьями?).

Дополнение

В своей предыдущей колонке я отвечал на вопрос Дестена Желонга (Destin Szelong) о скрытии функций (function hiding). Эта проблема возникает, когда у вас есть базовый класс с перегруженными виртуальными функциями и вы переопределяете одну из них в производном классе. В этом случае остальные функции автоматически не наследуются, потому что компилятор, найдя имя данной функции в ближайшей области видимости, дальнейший поиск прекращает. В той колонке я показал, как обойти эту проблему, определив функцию производного класса, которая явно вызывает ту же функцию из базового класса.

```

class D : public B {
public:
    virtual void test(int x) {
        B::test(x); // явное обращение к базовому классу
    }
};

```

Несколько читателей, в том числе такой гуру, как Скотт Мэйерз (Scott Meyers), подсказали мне, что есть и другой выход.

```

class D : public B {
public:
    using B::test;
    // замещение другой test
};

```

Это переводит B::test в область видимости D. Преимущество нового варианта в том, что вам не понадобится переписывать D, если кому-то захочется расширить B, добавив в него другие перегруженные функции test. А насколько меньше текста надо будет набирать! Когда вы используете using, все перегруженные функции B::test попадают в область видимости D и поэтому наследуются. Преимущество это или недостаток — зависит от того, что именно вы пытаетесь сделать. Если вы хотите оставить некоторые функции test скрытыми от пользователей класса D, можно применить первый способ (с явным вызовом функции из базового класса) либо объявить эти функции закрытыми (private) или защищенными (protected). Тот, кто читал книгу Скотта «More Effective C++» (Addison-Wesley, 1995), знает, что с помощью using можно обойти закрытость даже базовых классов (страницы 143–144).

```

class B { // какой-то класс
public:
    func1();
    func2(double d);
    virtual test();
    virtual test(int x);
};

// D наследует от B с использованием private
class D : private B {
public:
    // Делаем все функции B::test открытыми!
    using B::test;
};

```

Такой трюк уместен, когда В нужно тщательно скрыть, но открыть доступ к одной или двум функциям. И это лишний раз доказывает, что в С++ любой кувырок можно выполнить минимум двумя способами.

Удачи в программировании!

Исходный код можно скачать по адресу <http://download.microsoft.com/download/msdnmagazine/code/July02/WXP/EN-US/CQA0207.exe>.

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

Пол Дилация (Paul DiLascia) — внештатный писатель, консультант, а также независимый Web-дизайнер и разработчик пользовательских интерфейсов. Автор книги «Windows++: Writing Reusable Windows Code in C++» (Addison-Wesley, 1992). С ним можно связаться по адресу askpd@pobox.com или через <http://www.dilascia.com>.