Будущее WinRT или Going Native 2.0 перевод

[Windows](http://geektimes.ru/hub/windows/" \o "Вы не подписаны на этот хаб), [Microsoft](http://geektimes.ru/hub/microsoft/" \o "Вы не подписаны на этот хаб)

*Alexandre Mutel — создатель самой быстрой и самой полной .NET обертки для DirectX, единственной, поддерживающей Windows 8 Metro, работает R&D разработчиком игрового движка в SiliconStudio, участник французской демо-группы FRequency.*  
  
В последнее время мы слышим много шума о возвращении идеи «Going Native» после эры управляемых языков, таких как Java и .NET. В прошлом году, когда WinRT был только представлен, начали появляться недалекие комментарии, которые утверждали, что что .NET умер, а С++ возвращается во всей своей красе — истинный и единственно верный способ для разработки приложений, в то время, как JIT начинает все чаще появляться в мире скриптовых языков (JavaScript активнее всех использует преимущества JIT). Любой код так или иначе станет нативным перед выполнением — разница лишь в длине пути, по которому он пройдет, чтобы стать нативным, и насколько оптимизированным он будет. Значение слова «native» немного изменилось и стало неразрывно связано со словом «производительность». Даже будучи сильным пропагандистом управляемого языка [C#], его производительность на самом деле ниже хорошо написанного С++ приложения. Получается, мы должны просто принять этот факт и вернуться к C++, когда такие штуки как WinRT будут для нас основой межязыкового взаимодействия? По правде говоря, я бы хотел, чтобы .NET умер, и этот пост о том, почему и зачем.

Эра управляемых языков

Давайте пересмотрим недавнюю историю разработки на управляемых языках и отметим текущие проблемы. Помните слоган Java? «Write once runs everywhere». Это было представление новой парадигмы, когда полностью «безопасный» язык, основанный на виртуальной машине, связанный с богатым набором API, предоставит возможность с легкостью разрабатывать приложения под любую ОС и платформу. Это было начало эры управляемых языков. В то время как Java была достаточно успешно принята в разных отраслях разработки, она также была отвергнута многими разработчиками, которые были в курсе особенностей управления памяти, недостаточно оптимизированного JIT (хотя с тех пор все значительно улучшилось), огромного количества плохих архитектурных решений, вроде отсутствия поддержки структур, прямого доступа к памяти, а вызовы нативного кода через JNI были чрезвычайно неэффективными и трудоемкими (и даже [недавно](http://marxsoftware.blogspot.jp/2011/10/javaone-2011-from-java-se-2012-to-java.html) они рассматривали возможность убрать все нативные типы объектов и сделть все object — что за ужасная идея!).  
  
Также Java не смогла выполнить обещание, данное в самом слогане — на самом деле невозможно охватить единым API все возможности каждой платформы, что привело к таким вещам, как Swing — мягко говоря не самый оптимальный UI фреймворк. Также, изначально Java была разработана в расчете на единственный язык программирования, хотя многие увидели в JIT и байткоде возможность портировать скриптовые языки на Java JVM.  
  
В начале эры управляемых языков Microsoft пыталась войти на рынок Java с собственными расширениями для языка (об окончании этой истории знают все) и в итоге обзавелась собственной платформой для управляемых языков, которая в некоторых аспектах была лучше спроектирована и скомпонована: начиная с байткода, ключевого слова unsafe, вызова нативного кода, легковесного но очень эффективного JIT и NGEN, быстро развивающегося языка C#, C++/CLI и т. д. Изначально учитывая межязыковое взаимодействие и без бремени слогана Java (хотя Silverlight на MacOS или Moonlight были неплохими попытками).  
  
Обе платформы использовали похожий монолитный стек: метаданные, байткод, JIT и сборщик мусора — все это тесно связано. Соответственно, с производительностью были похожие проблемы: JIT подразумевает задержку при запуске, а выполнение кода не такое быстрое, как должно бы быть. Основные причины:

* JIT производит недостаточные оптимизации в сравнении с C++ -O2, потому что он должен генерировать код очень быстро (также, в отличае от Java HotSpot JVM, .NET JIT не может на лету подменять существующий код на более оптимизированный).
* Типы .NET, такие как Array, всегда делают проверку границ при доступе (не считая простых циклов, где JIT может убирать проверки, если условие завершения цикла меньше или равно длине массива).
* Сборщик мусора останавливает все потоки во время сборки (хотя новый сборщик мусора в .NET 4.5 в этом отношении несколько [улучшили](http://blogs.msdn.com/b/dotnet/archive/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps.aspx)), что может приводить к непредсказуемым падениям производительности.

Но даже с такой недостаточной производительностью, управляемая экосистема с универсальным фреймворком — это король продуктивности и межязыкового взаимодействия, с достойной общей производительностью для всех поддерживаемых языков. Апогеем эры управляемых языков был вероятно запуск WindowsPhone и VisualStudio 2010 (которая использовала WPF для рендеринга интерфейса, хотя сам WPF работал поверх приличного количества нативного кода). Управляемые языки были единственным дозволенным способом разрабатывать приложения в то время. Это было не лучшее, что могло случиться, принимая во внимание длинный список неразрешенных проблем с производительностью .NET, достаточно длинный, чтобы стимулировать «нативных разработчиков» нанести ответный удар, и они имели на это полное право.  
  
Так вышло, что это означает в каком-то смысле отказ от .NET. Я не так много знаю о внутренней кухне Microsoft, но судя по частым сообщениям, там наблюдается сильное противостояние между подразделениями. Хорошо это или плохо, но для .NET в последние годы создается впечатление, что у Microsoft заканчивается запал (например, нет практически никаких значительных улучшений в JIT/NGEN, множество неразрешенных запросов на улучшение производительности, включая такие вещи, как SIMD, которых разработчики ждут уже очень долго). И мне кажется, что все эти изменения возможны только в том случае, если .NET будет глобальной стратегией и при сильной поддержке и участии всех подразделений.  
  
В то же время, Google начал продвигать свою технологию NativeClient, позволяющую запускать нативный код в песочнице прямо из браузера. В прошлом году, следуя тренду «Going Native», Microsoft объявила, что даже HTML5, разработанный для следующего IE, будет нативным! Sic.  
  
В "[Reader Q&A: When will better JITs save managed code?](http://herbsutter.com/2012/04/02/reader-qa-when-will-better-jits-save-managed-code/)" Herb Sutter, один из евангелистов «Going Native», приводит некоторые интересные доводы о том, что философия «Going Native» думает о JIT ("[Can JITs be faster?](http://tirania.org/blog/archive/2012/Apr-04.html)" ответный пост Miguel de Icaza) с множеством неточных фактов, но давайте просто рассмотрим ключевой: даже если JIT в будущем станет лучше, управляемые языки уже сделали выбор между производительностью и безопасностью в пользу безопасности. Тем самым для них уже заказан путь в высшую лигу.  
  
И в этот момент появляется WinRT, который немного сглаживает острые углы. Используя часть философии .NET (метаданные и некоторые общие типы, такие как строки и массивы) и старую добрую модель COM (как общий знаменатель для нативного межязыкового взаимодействия), WinRT пытается решить проблемы взаимодействия языков за пределами мира CLR (что означает отсутствие потерь производительности у C++) и предоставить более современное API для ОС. Является ли это ответом на главный вопрос жизни, вселенной и всего такого? Не особо. Для WinRT выбрали курс на явное сближение технологий, который потенциально может привести к великолепным вещам, но пока что нет уверенности в правильном выборе пути. Но что может быть этим «правильным путем»?

Going Native 2.0 — производительность для всех

Проверки на безопасность могут иметь негативное влияние на производительность, но управляемый код не обречен всю жизнь запускаться только поверх медленного JIT (например, Mono умеет запускать C# код, нативно скомпилированный через LLVM на iOS/Linux) и было бы довольно легко расширить байткод небезопасными инструкциями чтобы предоставить контролируемое улучшение производительности (вроде отмены проверки границ массива).  
  
Но самой явной проблемой сейчас является отсутствие сильной инфрастуктуры для межязыковых компиляторов. Начиная с компилятора, используемого в IE 10 JavaScript JIT, .NET JIT и NGЕN компиляторы, Visual C++ компилятор (и многие другие) — все они используют разный код для практически одной и той же трудоемкой и сложной задачи — генерации эффективного машинного кода. Имея в распоряжении единый компилятор — это очень важный шаг для предоставления высокопроизводительного кода, доступного для всех языков.  
  
Felix9 на Channel9 [обнаружил](http://channel9.msdn.com/Forums/Coffeehouse/MS-working-on-a-same-compiler-for-C-AND-C--Not-in-incubation-but-for-production-), что Microsoft в действительности может работать над этой проблемой. Это определенно хорошие новости, но проблема «производительности для всех» лишь небольшая часть от общей картины. На самом деле «верный путь», упоминаемый ранее, — это более широкая интегрированная архитектура, не только улучшенный [LLVM](http://llvm.org/) стек, но поддержанная многолетним опытом Microsoft в разных областях (компилятор C++, JIT, сборщик мусора, метаданные и т. д.) система, которая будет предоставлять **полностью расширяемую и модульную «CLR»**, состоящую из:

* **Промежуточный язык среднего уровня**. Поддерживающий отражение, очень похожий на LLVM IR или байткод .NET, определяющий общие типы данных (примитивы, строки, массивы и т. д.). Должен быть доступен API похожий на System.Reflection.Emit. Векторизованные типы (SIMD) должны быть такими же базовыми, как int и double. IL код не должен ограничиваться только CPU, но также должен позволять делать вычисления на GPU (как это делают AMP расширения для C++). Должна быть возможность представить байткод HLSL с помощью этого IL, используя преимущества единой инфрастуктуры компиляторов (см. далее). IL без типов также должен быть доступен, чтобы было проще переносить на него динамические языки программирования.
* **Динамически связываемые библиотеки и исполняемые файлы, такие как сборки .NET, предоставляющие метаданные, IL код**, поддерживающие отражение. Во время разработки код должен связываться со сборками (IL кодом), а не с устаревшими заголовочными файлами C/C++).
* **Компилятор из IL в машинный код**, который может быть интегрирован в JIT, настольное приложение или в облачный компилятор, или комбинация всего этого. Этот компилятор должен предоставлять векторизацию настолько, насколько это поддерживает целевая платформа. IL код должен компилироваться в машинный код во время установки или развертывания, используя информацию об архитектуре системы (во время разработки это можно сделать сразу после компиляции в IL). Этапы компиляции должны быть доступны через API и должны предоставлять точки расширения везде, где это возможно (предоставляя доступ к IL, оптимизации IL, или встраиванию собственных трансформаций из IL в машинный код). Настройки оптимизации должны варьироваться от быстрой компиляции (вроде JIT) до агрессивной оптимизации (заранее скомпилированные приложения или горячая замена кода в JIT на более производительный). Профиль приложения может также использоваться для автоматической подстройки локализованных оптимизаций. Этот компилятор должен поддерживать продвинутые cценарии использования JIT, такие как динамический анализ кода, On Stack Replacement (OSR, позволяющий замену кода для сложных вычислений на более оптимальный код прямо во время выполнения), в отличае от текущего компилятора .NET JIT, который компилирует метод во время его первого запуска. Оптимизации подобного рода очень важны в динамических сценариях, когда вывод типов (type inference) происходит уже после компиляции (как в случае Javascript).
* **Расширяемый компонент для выделения памяти, позволяющий параллельные выделения памяти. Сборщик мусора будет одной из возможных реализаций**. Большая часть приложений будут использовать именно его для большинства объектов, в то время как наиболее критичные для производительности объекты будут использовать другие стратегии выделения памяти (вроде подсчета ссылок, используемого в COM/WinRT). Не дожно быть никаких ограничений на использование нескольких стратегий выделения памяти в одном приложении (имено это и просходит в .NET, когда приложению приходится прибегать к использованию вызовов нативных функций для создания объектов за пределами CLR).

Идея очень близка к стеку CLR, однако она не принуждает приложения запускаться поверх JIT компилятора (да, в .NET есть NGEN, но он был разработан для ускорения загрузки, не для ускорения общей работы, кроме того это черный ящик и он работает только со сборками, установленными в GAC) и позволяет смешанные стратегии выделения памяти: с использованием сборщика мусора и без него.  
  
В подобной системе межязыковое взаимодействие будет более простым, не жертвуя производительностью ради простоты и наоборот. В идеале, сама ОС должна быть построена на базе подобной архитектуры. Возможно именно эта идея была (есть?) в основе таких проектов, как [Redhawk](http://www.zdnet.com/blog/microsoft/microsoft-codename-redhawk-lives-in-windows-8/9233) (это что касается компилятора) или [Midori](http://www.zdnet.com/blog/microsoft/goodbye-xp-hello-midori/1466) (что касается ОС). В подобной интегрированной системе, возможно, только драйверы будут требовать прямого доступа к железу.  
  
Felix9 также [раскопал](http://channel9.msdn.com/Forums/Coffeehouse/MS-patent-Optimizer-as-an-AppStore-Service-Cloud-JIT-/c539c486cf914d3ab718a0a700fb473e), что промежуточный байткод, более низкоуровневый, чем MSIL (байткод .NET), называемый MDIL, уже может использоваться и он может быть именно тем промежуточным байткодом, описываемым ранее. Хотя, если посмотреть на соответствующий патент "[INTERMEDIATE LANGUAGE SUPPORT FOR CHANGE RESILIENCE](http://www.freepatentsonline.com/y2011/0258616.html)", то в спецификации можно найти x86 инструкции, которые не совсем подпадают под определение архитектурно независимого байткода. Возможно они оставят MSIL без изменений и задействуют MDIL на более низком уровне. Скоро узнаем.  
  
Итак, какие проблемы с этой точки зрения решает WinRT? Метаданные, немного API, поддерживающего песочницы и межязыковое взаимодействие в зачаточной стадии (хотя есть общие типы данных и метаданные). Как видим, не густо, эдакий COM++. Также очевидно, что **WinRT не предоставляет продвинутые оптимизации, когда мы используем его API**. Например, нам не позволено иметь структуру со встраиваемыми методами. Каждый вызов метода в WinRT — это виртуальный вызов, который обязательной пойдет через таблицу виртуальных методов (а в некоторых случаях требуется несколько виртуальных вызовов, когда например используется статический метод). Простейшие чтение-запись свойства потребуют виртуального вызова. Это явно неэффективно. Судя по всему WinRT нацелен только на более высокоуровневые API, не позволяя сценарии, в которых мы бы хотели использовать высокопроизводительный код везде, где это возможно, минуя слой виртуальных вызовов и невстраиваемого кода. В итоге мы имеем расширенную модель COM — это не совсем то, что можно было бы назвать «Building the Future».

Продуктивность и производительность для C# 5.0

Язык вроде C# был бы идеальным кандидатом для такой модульной CLR системы, и мог бы с легкостью переноситься на уже существующий промежуточный байткод. Но чтобы эффективно использовать подобную систему, C# должен быть улучшен в нескольких аспектах:

* **Больше небезопасных конструкций**, когда мы могли бы выключить «управляемое» поведение вроде проверки границ массивов (вроде «супер небезопасного режима», когда мы могли бы использовать инструкции кеширования в CPU для доступа к элементам массива, подобного рода «продвинутые» штуки сейчас невозможно делать с управляемыми массивами без использования недокументированных трюков).
* **Конфигурируемый оператор new**, который бы поддерживал разные схемы распределения памяти.
* **Векторизованные типы** (вроде float4 в HLSL) должны быть добавлены к базовым типам. Об этом уже давно просили (с ужасными патчами в XNA WP для «решения» этой проблемы).
* **Легковесное взаимодействие с нативным кодом**: в текущем состоянии переход от управляемого к неуправляемуму коду довольно затратный даже без передачи каких бы то ни было параметров. Переход к неуправляемому коду должен быть возможен без x86/x64 инструкций пролога/эпилога, которые сейчас генерируются в .NET JIT.

Кроме производительности есть и другие не менее важные области:

* **Обобщения (generics) везде** — в конструкторах и неявных преобразованиях типов, с более продвинутыми конструкциями (контрактами для операторов и т. д.), ближе к гибкости шаблонов С++, но более безопасные и менее загроможденные.
* **Наследование и финализаторы в структурах** (чтобы позволить выполнение легковесного кода при завершении метода без использования громоздких паттернов вроде try/finally и using).
* **Больше метапрограммирования**. Методы расширения для статических типов, примеси (добавление содержимого класса внутрь другого класса, удобно для вещей вроде математических функций), модификация классов/типов/методов во время компиляции (например, методы, которые бы вызывались во время компиляции для добавления других методов или свойств к классу, чем-то похоже на [eigenclass в Ruby](http://stackoverflow.com/questions/1630815/why-isnt-the-eigenclass-equivalent-to-self-class-when-it-looks-so-similar), вместо использования T4 шаблонов для генерации кода).
* **Встроенный литерал или тип, которым бы можно было выразить ссылку на объект языка** (класс, свойство, метод), используя простую конструкцию вроде: symbol LinkToMyMethod = @ MyClass.MyMethod;вместо использования Linq выражений. Это сделало бы более надежным код для таких вещей как INotifyPropertyChanged или упростило бы все системы, основанные на свойствах, вроде WPF (который в текущем состоянии содержит много дублирующего кода).

Основная идея в том, что **нужно меньше добавить в C#, чем убрать из C++, чтобы полностью задействовать возможности подобной интегрированной системы, увеличить продуктивность разработчика и без сопутствующих потерь производительности**. Кто-то может поспорить, что С++ уже предлагает все это и даже более, но именно поэтому С++ такой загроможденный (с точки зрения синтаксиса) и опасный для большинства разработчиков. Он позволяет небезопасный код абсолютно везде, в то время как в каждом приложении есть вполне определенные места, где он действительно нужен (что приводит к проблемам с памятью, которые проще исправить, если бы эти места были явно обозначены в коде, как это сделано с ключевым словом asm). Намного проще и безопаснее отслеживать подобные области в коде, чем иметь их повсеместно.

Что же далее?

Мы надеемся, что Microsoft выбрала путь от общего к частному и начала с выпуска WinRT, предоставляющего универсальное API для всех языков и простое межязыковое взаимодействие. И что затем они представят все эти более продвинутые возможности в следующих версиях их ОС. Но это идеальная ситуация и будет интересно наблюдать, сможет ли Microsoft справиться с этим. Даже если недавно объявили, что .NET приложения в WP8 будут иметь преимущества компиляции в облаке, до сих пор мы мало что знаем об этом: это просто адаптированный NGEN (который, напомню, не ориентирован на производительность и генерирует код очень похожий на тот, что генерирует JIT) или еще не представленный компилятор RedHawk?  
  
У Microsoft наверняка что-то есть в загашнике, учитывая многие годы разработки компиляторов C++, JIT, сборщика мусора и всех связанных R&D проектов, которые у них есть…  
  
Подводя итоги — .NET должен умереть и уступить свое место более интегрированной, ориентированной на производительность, общей среде, где бы управляемое (безопасность и продуктивность) и неуправляемое (производительность) были бы тесно связаны. И это должно быть структурной частью следующего витка развития WinRT.