УДК 004.4'232 004.4'413

Оптимизация объектных моделей на примере библиотеки Colorer-take5.

И.В. Русских (irusskih@gmail.com), ННГУ, 2006г.

В статье проанализированы причины падения производительности в сложных программных системах объектно-ориентированного типа, рассмотрены общие принципы сериализации объектной модели. В контексте оптимизации библиотеки Colorer предложено решение «заморозки» модели, имеющее свои преимущества в сравнении с традиционными схемами сериализации.

The article analyses some causes of performance degradation in complex object oriented systems, general principles of object model serialization are reviewed. Model «freezing» solution is introduced within Colorer library description with its advantages against traditional serialization schemes outlined.

Введение.

На сегодняшний день объектно-ориентированное программирование является основной парадигмой разработки и поддержки программного обеспечения. Развившаяся из структурного программирования, объектная модель наиболее естественным для человека образом описывает структуру и поведение программных систем и позволяет оптимально и безболезненно взаимодействовать с аппаратной моделью вычислительных систем.

Одной из главных черт ООП является модульное построение, в котором независимые компоненты, выполняющие каждый свои задачи, затем компонуются в единую систему. При этом комплексные приложения обычно оперируют огромным числом динамических объектов, многие из которых создаются при активации приложения и существуют длительное время. Начальная загрузка приложения при этом может занимать длительное время в связи с необходимостью создания объектов динамически в памяти и их инициализации. На примере анализа объектной модели библиотеки Colorer можно рассмотреть возможные пути решения подобных проблем на разных этапах создания приложения.

1. Библиотека Colorer-take5, описание.

Colorer является специализированным синтаксическим анализатором, предназначенным для разбора текста и вносимых в него изменений в реальном времени, дальнейшей

подсветке синтаксиса и предоставлению среде редактирования информации о структуре редактируемого текста. Одной из особенностей, обеспечивающих гибкость и масштабируемость системы, является разработанная модель хранения синтаксических правил и описаний разбора в специальном формате HRC [1].

Формат основан на XML и позволяет быстро и эффективно описывать требуемые правила для целевых разбираемых языков программирования. HRC создает дополнительный уровень абстракции над стандартными средствами синтаксического разбора - регулярными выражениями и контекстами. В модели HRC введены понятия наследуемых контекстов (схем), типов языков программирования. Это позволяет легко описывать сложные, смешанные языки - такие как JSP, PHP, ASP, HTML. Структура этих и многих других языков может содержать в себе смесь более примитивных синтаксисов, и для эффективного их описания введены такие понятия как наследование контекстов, условное изменение их поведения (виртуализация) и др.

Система Colorer эффективно отличается от решений со схожими задачами ([2, 4]) эффективностью и простотой описания синтаксисов, реализуя в тоже время мощный инструментарий по поддержке работы программиста.

Удобство и гибкость разработанного языка HRC привели к накоплению и созданию описаний для огромного числа языков программирования, скриптов, языков разметки - на данный момент библиотека Colorer поддерживает более 170 раздельных языковых описаний. Многие описания развиваются вглубь, по пути усложнения структуры разбора и предоставления четкой модели текста. Так, изначально простые описания языка XML, описывающие лишь примитивные цветовые выделения тэгов и атрибутов, сейчас эволюционировали в сложные модели, анализирующие структуру документа в соответствии с его схемой (XSD), валидирующие и показывающие пользователю все ошибки не только в синтаксисе, но и в структуре. Многие из этих HRC описаний настолько сложны (в основном из за унаследованной сложности их XSD схем), что не создаются вручную, а генерируются автоматически из соответствующих моделей XML документов (XSD или DTD) [9].

2. Требования к производительности.

В то же время, такое развитие HRC привело и к отрицательным последствиям: для загрузки, компиляции и преобразования таких HRC моделей библиотеке Colorer требуется все больше и больше времени.

В абсолютном значении это время не так велико - на данный момент время загрузки всей базы в память средней машины (PIV, 3GHz) занимает около 10 секунд. При том, что библиотека подгружает модели HRC по мере надобности, загрузка каждого отдельного языка программирования занимает не больше 1 секунды. В то же время такие параметры системы, как общая нагрузка, активное использование виртуальной памяти (swapping),

файловой подсистемы, увеличивают среднее время загрузки приложения - для пользователя оно может возрасти до 4-5 секунд, что отрицательно сказывается на удобстве работы.

В случае с библиотекой Colorer возникает еще один фактор, критичный ко времени начальной загрузки. Существуют интерфейсы библиотеки, нацеленные на использование ее в WEB серверах для расцветки в реальном времени исходных текстов в сети интернет. В таких средах нагрузка на сервер зависит от посещаемости ресурса, и в некоторых случая может быть очень велика. Существующее время инициализации библиотеки Colorer совершенно неприемлемо для подобных сред, в которых число запросов может колебаться от единиц до десятков и сотен в секунду. Для таких сред необходимо рассмотрение новых подходов, обеспечивающих достаточную производительность.

В Таблице 1 приведены среднее время инициализации библиотеки и время анализа для некоторых популярных языков программирования.

Таблица 1: Время инициализации по типам языков

Language	**Total**	**HRC**	**Parser**	**HRC%**	**Parser%**
HTML	0.959	0.921	0.038	96.0%	4.0%
XML	0.45	0.412	0.038	91.6%	8.4%
Perl	0.295	0.22	0.075	74.6%	25.4%
C++	0.445	0.418	0.027	93.9%	6.1%
Java	0.459	0.428	0.031	93.2%	6.8%

Для синтаксического анализа использовались примеры кода на соответствующих языках программирования средним объемом 2Kb. Время непосредственно анализа в данном случае несоизмеримо меньше времени инициализации библиотеки: каждый раз необходимо загружать в память, компилировать и подготавливать синтаксические описания для последующего анализа.

Конечно, существуют решения, позволяющие в конкретных окружениях избежать подобных проблем. Можно рассмотреть создание сервиса, имеющего единожды загруженную модель, и предоставляющего сервисы по мере надобности. В случае с WEB-средами существуют решения, позволяющие легко и быстро интегрировать подобные резидентные сервисы в систему web-сервера (такие как FastCGI для Apache).

Но все же, в общем случае подобные решения являются обходными маневрами, никак не раскрывающими проблему изначальной сложности программной среды. Так, подобные решения не применимы во встраиваемых и мобильных системах, практически всегда они требуют осторожного обращения в многопроцессорных, многопотоковых средах.

3. Сериализация, преимущества и недостатки.

Более прямой и многообещающий подход - рассмотрение и анализ исходной проблемы.

В случае с библиотекой Colorer описание всей модели, подлежащей загрузке, представлено в форме, удобной человеку. Это XML файлы, которые удобно редактировать вручную (или в XML редакторе), и которые затем на лету компилируются при загрузке во внутреннюю форму. Такие инструменты, как регулярные выражения, синтаксические структуры, так же выражены в удобной для человека форме и лишь при загрузке библиотеки компилируются.

Естественно, что алгоритмы анализа и компиляции, начиная с разбора XML синтаксиса и заканчивая выделением динамической памяти занимают процессорное время, порождают временные объекты в памяти и тем самым способствуют увеличению времени инициализации [8].

Рисунок 1 демонстрирует различные стадии инициализации библиотеки Colorer.

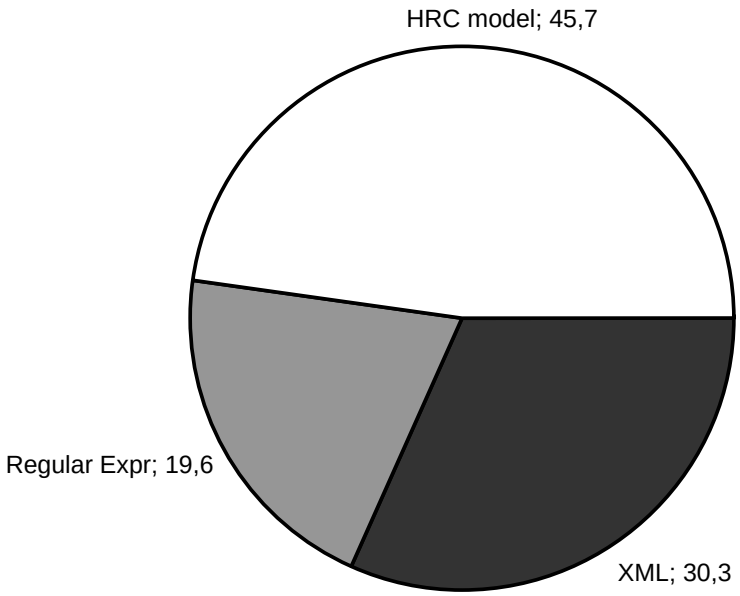

Рис. 1: Стадии инициализации библиотеки Colorer

Одним из возможных вариантов уменьшения времени загрузки является сериализация модели - частичная или полная. Сериализация предполагает исключение стадии преобразования данных из формы, доступной человеку в форму, доступную машине. Обычно для этого требуемые объекты в модели доопределяются методами, позволяющими объекту сохранить свой образ в постоянное хранилище (файл, поток) и извлечь его оттуда. Создав дополнительно инфраструктуру для сериализации объектов-контейнеров, в библиотеке Colorer потенциально можно получить бинарный образ HRC описаний, идентичный ис-

ходному, понятному человеку описанию. Таким образом, отбрасывается необходимость каждый раз передавать XML процессору на анализ огромное количество HRC файлов из базы описаний библиотеки.

Аналогично можно сериализовать объекты регулярных выражений так, чтобы они хранились в форме, более простой и быстрой с точки зрения алгоритма сопоставления регулярного выражения. Сохраненное состояние такого объекта можно будет сравнить с машинным кодом, скомпилированным из исходного, понятного для человека представления. Для изолированных объектов этот метод работает достаточно эффективно.

Задача сериализации усложняется при наличии в объектной модели сложных связей между объектами - что практически всегда встречается в реальных приложениях. Для сериализации таких групп объектов необходимо прилагать дополнительные усилия по идентификации объектов, сериализации идентификаторов и ссылок на объекты. Дополнительно к этому ООП добавляет сложности, связанные с полиморфизмом объектов из иерархий наследования. Для поддержки таких моделей необходимо вводить искусственную идентификацию типа сериализуемого объекта. После того, как все эти особенности будут учтены и поддержаны, задача сводится в общем случае к обходу и сериализации графа объектов [7].

Для всего этого, естественно, требуются дополнительные усилия по реализации. Объектная модель изначально, при первичном анализе будующей структуры приложения должна учитывать требования к сериализации и каждый тип объекта должен включать функциональность сериализации и десериализации. Конечно, это создает ощутимые трудности. В случае с уже существующим приложением (как в примере с библиотекой Colorer), его модель еще сложнее переделать под подобные требования.

Еще одна проблема связана уже с полученной производительностью. Конечно, в случае сложных алгоритмов компиляции и инициализации объектной модели, сериализация даст значительный выигрыш. Но все же, довольно весомым недостатком является тот факт, что при десериализации объектной модели все объекты будут воссоздаваться из сохраненного потока по отдельности. В случае крупных иерархий объектов, сложных связей между ними, это может занять значительное время, хотя и пропорциональное лишь количеству объектов в модели.

Возможное решение этой проблемы - разработка и дизайн системы изначально таким образом, чтобы она могла воспринимать сериализованную форму единым блоком. К сожалению это требует больших усилий при создании приложения и может приводить к утрате объектной парадигмы, смешению компонентов и как следствие к существенному увеличению усилий по поддержке и развитию приложения. Более того, такой подход практически не приемлем при рассмотрении уже спроектированных и работающих систем (как

в случае с библиотекой Colorer).

4. Альтернатива - заморозка объектной модели.

На примере библиотеки Colorer предлагается иной подход к решению подобных проблем. Это концепция «заморозки» модели, основная идея которой - автоматическое слежение за создаваемой объектной моделью и связями между объектами. С точки зрения языка C++, каждый объект в модели занимает некоторое пространство в динамической памяти. Будучи побайтово клонированным, изолированное поведение клона ничем не будет отличаться от поведения исходного объекта (с некоторыми допущениями). Склонировав уже загруженную объектную модель целиком, и сохранив ее в постоянное хранилище, можно восстановить этот клон единым блоком. Все связи между объектами, какими бы сложными они не были, восстановятся автоматически на основе информации о положении исходных динамических объектов.

Решение по концепции напоминает функцию заморозки (hibernation), позволяющую быстро сохранить и восстановить после выключения состояние персонального компьютера. Отличие в том, что представленное решение работает на уровне модели программной системы, а не на уровне аппаратной модели или операционной системы. Собственно реализация этого решения совершенно не связана с целевой областью приложения и в библиотеке Colorer представлена в виде изолированного компонента. Заморозка работает на низком уровне (уровне связей указателей), и не интересуется структурой модели и логическими связями между объектами.

Реализация основывается на подмене системных методов выделения и освобождения динамической памяти (это возможно в модели языка C++). Для создания замороженного образа модели, приложение начинает загрузку в обычном режиме. Одновременно с этим переопределенные методы работы с памятью контролируют местоположение все запрошенных в модели объектов. После завершения инициализации модели, готовое к работе приложение активирует процедуру сериализации. Объекты побайтово копируются и упаковываются в единый блок, связи между объектами автоматически анализируются и строится таблица относительных ссылок между объектами. Такое построение возможно благодаря тому, что модуль заморозки владеет информацией об адресах все объектов в памяти и способен эвристически отличить данные от указателей на другие объекты.

Схожие идеи в мобильных и realtime системах иногда именуются «Memory Blasting», или «ROMizing». В библиотеке же Colorer замороженная модель фактически является кэшем скомпилированных символьных описаний HRC и позволяет существенно сократить время загрузки библиотеки. Диаграммы 2 и 3 показывают время инициализации библиотеки в различных вариантах конфигурации базы описаний HRC (в секундах).

Из диаграмм видно, что восстановление полной базы из замороженного состояния за-

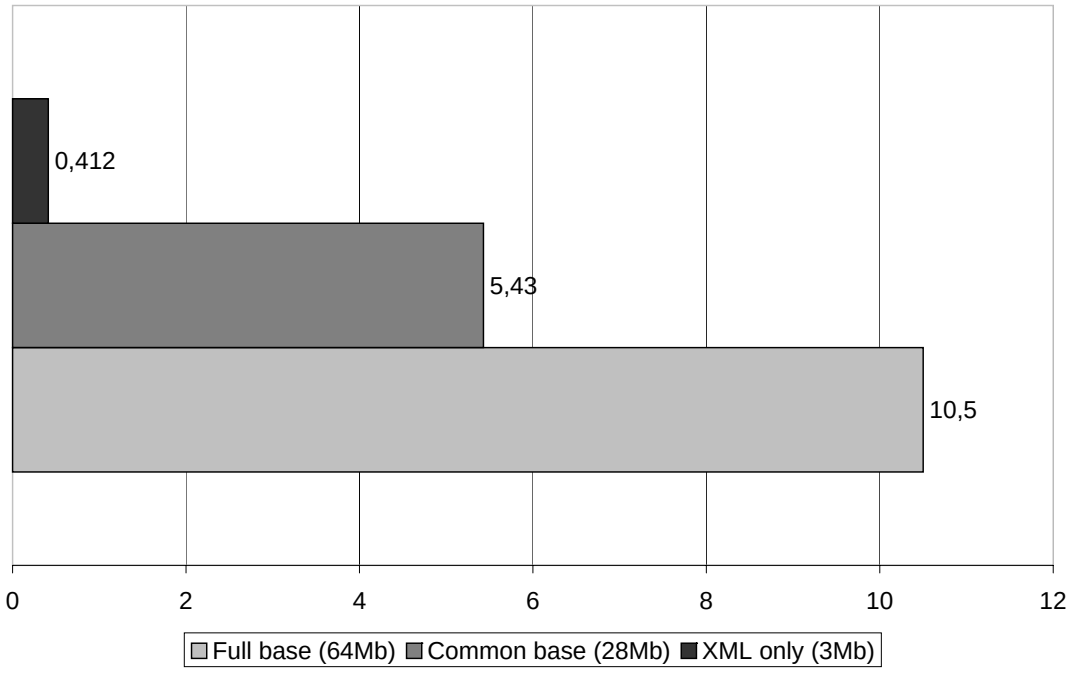

Рис. 2: Dynamic compilation

нимает времени меньше, чем даже частичная динамическая инициализация. Более того, время восстановление базы из ROMized образа пропорционально лишь размеру этого образа и фактически равно времени его чтения с устройства хранения.

Частичная заморозка модели, которая так же возможна, уменьшает время инициализации до значения 70 мсек. Такой режим может использоваться для работы библиотеки в режиме веб-сервиса, когда заранее замораживается ограниченный набор наиболее часто запрашиваемых типов данных, а синтаксисы, не попадающие в этот набор, загружаются обычным способом.

Реализация заморозки не требует абсолютно никаких изменений в готовом приложении. Платой за это является довольно жесткий набор ограничений, и условий применимости данного решения.

В общем случае замораживаемая модель должна быть «закрытой». В объектах модели не должно присутствовать активных ссылок на какие-либо внешние ресурсы, не поддающиеся сериализации. К таким ресурсам относятся, например, открытые дескрипторы файловых объектов, объектов операционной системы.

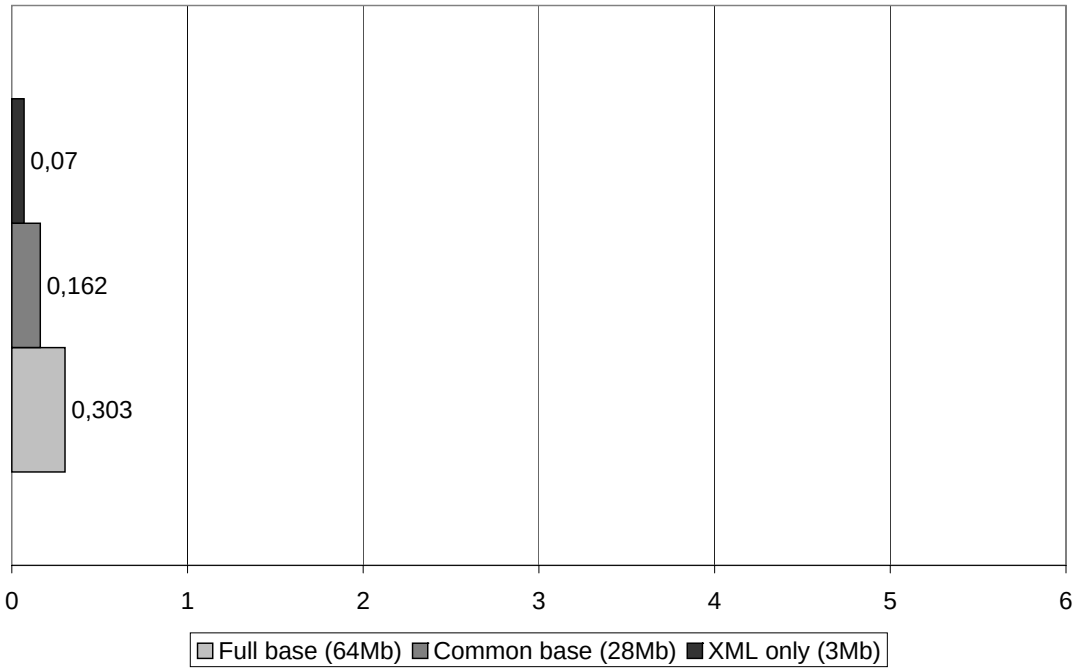

Рис. 3: ROMization (диаграмма масштабирована)

В случае с языком C++ его спецификация не разрешает побайтового копирования сложных объектов. Такое копирование может выполняться только встроенным конструктором клонирования, поэтому может существовать риск возникновения проблем при использовании некоторых реализаций языка C++. Реализация заморозки была успешно протестирована на компиляторах Microsoft и GCC (платформы Windows и Linux).

Следующее ограничение - привязка сериализованной модели не только к аппаратной платформе, но и к версии компилятора и даже к версии самого приложения. Минимальное изменение в объектной модели и перекомпиляция приложения делает замороженную модель некорректной. Для корректного обнаружения несоответствия между объектной моделью приложения и существующим ROMized образом можно использовать версионность или присвоение временных меток.

При интеграции данного решения в конечные среды могут возникнуть особенности, связанные с необходимостью обработки таблиц виртуальных функций объектов. Во многих операционных средах исполняемый образ приложения может занимать различное место в адресном пространстве. Часто это зависит от способа загрузки приложения, иных

факторов. Это в свою очередь сказывается на адресах виртуальных функций, которые в реализациях C++ хранятся в контексте объекта. Для корректного функционирования загруженной модели необходимо корректировать эти таблицы с адресами, а это требует наличия у операционной системы интерфейса по определению базового адреса приложения. Это так же накладывает ограничения на использование решения в системах со смешанными моделями, состоящими из объектов, принадлежащих физически различным компонентам.

Заключение.

Несмотря на описанные ограничения, предлагаемое решение является действенной альтернативой иным способам сериализации объектной модели в первую очередь из-за простоты применения и внедрения: решение не требует никаких структурных и архитектурных изменений объектной модели приложения.

В отличие от традиционных способов сериализации, заморозка никак не вредит целостности и объектной структурированности модели. Она не требует отказываться от абстракции и объектного представления модели. Решение оптимально подходит для систем с проблемами производительности, которые невозможно разрешить иным способом.

Многие из ограничений заморозки модели исчезают при рассмотрении встраиваемых или мобильных систем, что связано с их спецификой (единое адресное пространство, централизованная инициализация). Помимо улучшения производительности заморозка модели улучшает показатели работы с памятью. Вся модель представляется единым блоком, а это избавляет систему от ресурсоемкой задачи динамического выделения памяти [8].

Возможность реализации подобной системы на других языках программирования зависит от динамических свойств этих языков. Так, система тривиальна и стабильна в языке C: исчезают особенности, связанные с поддержкой виртуальных функций. В иных объектных системах (например, в системах со сборкой мусора) реализация описанного способа невозможна без дополнительной поддержки со стороны среды выполнения (компилятора, интерпретатора).

Предоставление средой информации о внутренней структуре объекта (ссылки, типы полей) потенциально позволило бы убрать эвристическую часть из работы описанного алгоритма, связанную с поиском ссылок и адресов методов в таблицах виртуальных функций. Подобные reflection-механизмы, дающие доступ к структуре самого языка, к сожалению, не стандартизированы языком C++ и недоступны напрямую.

Несмотря на все это, при использовании в библиотеке Colorer, описанная система приносит ощутимую выгоду, причем не только в специализированных случаях (Web-сервисы и т.д.), но и в каждодневной работе у рядовых пользователей.

Список литературы

[1] Igor Russkih. Colorer-take5 Library, HRC Language Reference. `http://colorer.sf.net/`

[2] Andrew J. Ko and Brad A. Myers. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. CHI 2006 Proceedings, 387-396.

[3] Borenstein, N. S. The Evaluation of Text Editors: A Critical Review of the Roberts and Moran Methodology Based on New Experiments, ACM CHI 1985 Proceedings, p. 99-105.

[4] Boshernitsan, M., Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools, University of California, Berkeley, Technical Report CSD-01-1149, 2001.

[5] Teitelbaum, T. and Reps, T., The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, CACM, 24, 9, (1981), 563-573.

[6] Doug Lea. Some storage allocation techniques for container classes. C++ Report, 1989.

[7] Jiri Soukup. Taming C++: Pattern Classes and Persistence for Large Projects. Addison-Wesley, 1994.

[8] Русских И.В. Оптимизация системы динамического распределения памяти в приложениях на примере библиотеки Colorer-take5. // Вестник Нижегородского университета, сер. Математическое моделирование и оптимальное управление, Н.Новгород, вып. 1(27), 2004. с.234-242.

[9] Русских И.В. Основные принципы разработки библиотеки Colorer. // Методы и средства обработки сложной графической информации. Тезисы докладов. Н.Новгород, 2003 г. с. 80-82.

[10] Rational Quantify. `http://www.rational.com`