Математические структуры и моделирование
2015. № 1(33). С. 93–100

УДК 004.623

СОКРАЩЕНИЕ ИСПОЛЬЗОВАНИЯ ПАМЯТИ В ПОЛЬЗОВАТЕЛЬСКОМ ИНТЕРФЕЙСЕ СЕРВИСА, УПРАВЛЯЮЩЕГО КОНФИГУРАЦИЕЙ БАНКОВСКОЙ ТОРГОВОЙ ПЛАТФОРМЫ

А.И. Журавлев
аспирант, e-mail: grusalex@gmail.com
Д.Н. Лавров
к.т.н., доцент, e-mail: lavrov@omsu.ru

Омский государственный университет им. Ф.М. Достоевского

Аннотация. В работе описываются методы сокращения потребления памяти Java-приложением — графическим интерфейсом, предназначенным для конфигурации банковской торговой платформы. Наиболее значительное изменение в программной реализации интерфейса — это применение паттерна отложенной загрузки данных в отношении репозиториев доменных объектов, что стало возможным благодаря выделению двух основных пользовательских сценариев использования приложения.

Ключевые слова: графический интерфейс пользователя, потребление памяти, отложенная загрузка данных, репозиторий, доменный объект.

Введение

Одной из основных характеристик для приложения, написанного на языке Java, является объем используемой памяти. Повышенное использование памяти приводит к замедлению работы программы и её возможным зависаниям при длительном функционировании, ухудшается масштабируемость приложения. Такие проблемы могут стать особенно неприятными на уровне непосредственно пользовательского интерфейса, так как в таком случае у пользователей наиболее отчётливо формируется устойчивый негативный опыт эксплуатации программного продукта.

Особенностью Java является наличие сборщика мусора. Он обеспечивает автоматическое управление памятью, освобождая разработчика от необходимости явно выделять и освобождать память, но не защищает от проблем, вызванных неоптимальным дизайном приложения или неграмотно написанным кодом. Эти проблемы могут проявляться не на ранних этапах разработки и использования программы, но с её развитием (например, ростом числа пользователей или объёма обрабатываемой информации), что приводит к необходимости выработки и реализации мер по сокращению использования памяти.

1. Описание оптимизируемого приложения

Подобного рода неприятности возникли применительно к конфигурационному сервису, поддерживающему функционирование платформы алгоритмической торговли на валютном рынке. Данная платформа разработана и используется в международном инвестиционном банке. Конфигурационный сервис является совокупностью графического интерфейса пользователя и сервера, взаимодействующего с графическим интерфейсом, базой данных и бизнес-компонентами платформы. Пользователям предоставляется возможность указывать и изменять параметры функционирования бизнес-компонент, а также отслеживать информацию о текущем состоянии платформы — например, курсы валют в клиентских сделках. Графический интерфейс пользователя и сервер разработаны на языке Java с использованием нескольких фреймворков, например Spring. Увеличение числа конфигурируемых бизнес-компонент негативно повлияло на потребление памяти пользовательским интерфейсом, так как вырос объем отображаемых конфигурационных данных (рис. 1).

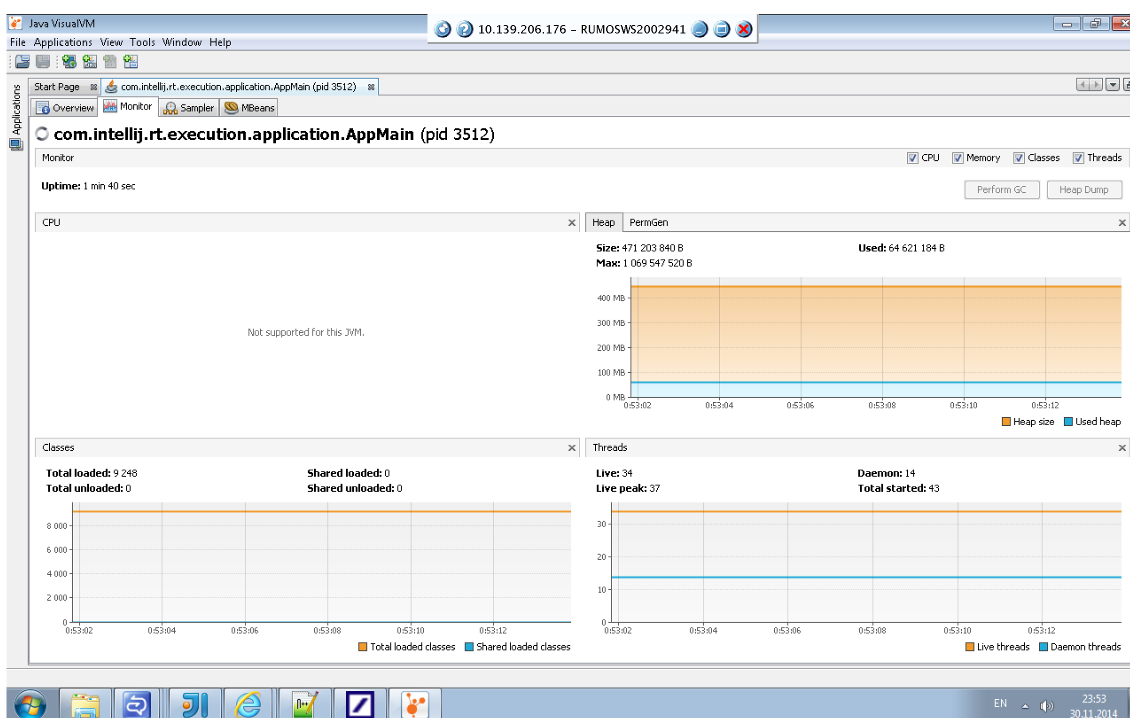

Рис. 1. Потребление памяти интерфейсом до оптимизации

Различные части конфигурации бизнес-компонент моделируются различными доменными объектами. Интерфейс пользователя состоит из вкладок (рис. 2), каждая из которых соответствует определённой части конфигурации и, следовательно, определённому доменному объекту.

Доменные объекты не имеют бизнес-логики, так как она инкапсулирована в бизнес-компонентах и отсутствует в сервере и пользовательском интерфейсе. Такой подход характерен для анемичной модели предметной области. В дан-

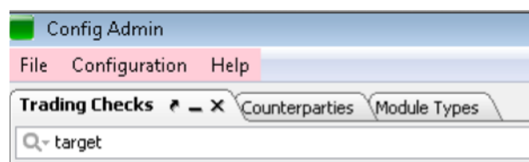

Рис. 2. Вкладки интерфейса для разных частей конфигурации

ном случае анемичная модель унифицировала доменные объекты, превращая их в разные наборы связанных параметров; каждый доменный объект является POJO (plain old Java-object, простой Java-объект в старом стиле), чьи различия проявляются лишь в типах данных, именах, бизнес-семантике параметров, но не в доменной логике, поскольку она полностью отсутствует и поддерживается другими частями системы. Доменные объекты сериализуются для передачи данных между сервером и пользовательским интерфейсом, что делает доменные объекты схожими с DTO (data transfer object, объект передачи данных) [1].

Разные доменные объекты хранятся в разных репозиториях, которые кэшируют данные и синхронизируют их с сервером. Синхронизация репозиториев происходит при изменениях, вносимых пользователями, и основана на использовании специальной характеристики репозитория — ревизии. Сервер публикует ревизию для каждого изменяемого репозитория, и клиентские репозитории могут запросить у сервера актуальные данные, чтобы синхронизировать своё состояние. Подобные механизмы упомянуты в паттерне Observer (Наблюдатель) [2].

Репозитории реализованы схоже с паттерном Repository (Репозиторий), описанным в [1].

```
public interface Repository<T extends DomainObject> {
    Map<Key, T> find();
    Map<Key, T> find(Predicate<? super T> predicate);
    T findByKey(Key key);
    Map<Key, T> findByKeys(Collection<Key> keys);
    boolean contains(Key key);
    void update(Collection<T> items);
    void removeAll(Collection<T> items, );
    void removeByKeys(Collection<Key> keys);
    void addListener(RepositoryListener<T> listener);
    void removeListener(RepositoryListener<T> listener);
    int size();
}
```

Как видно из приведённого интерфейса, репозиторий имеет операции, схожие с операциями из интерфейсов стандартных коллекций. Эти операции делают репозиторий высокоуровневой сущностью, скрывающей для классов-пользователей (в основном это модели данных визуальных компонент) репозитория низкоуровневые детали доступа к данным.

2. Отложенная загрузка репозиториев доменных объектов

Изначально при старте интерфейса происходила загрузка всех репозиториев, что способствовало более быстрому открытию соответствующих вкладок пользователем, так как при каждом последующем открытии вкладки не требовалось загружать данные повторно. Однако при поиске причин повышенного потребления памяти стали очевидными некоторые особенности работы разных пользователей. Все множество пользователей можно разделить на 2 группы — трейдеры и техподдержка. Большинство трейдеров постоянно использовало лишь вкладки, отображающие данные о текущем бизнес-состоянии платформы, изредка открывая конфигурационные вкладки для изменения и сверки параметров. Напротив, пользователи из техподдержки более активно использовали конфигурационную функциональность интерфейса, выполняя операции по созданию, удалению, копированию, загрузке и обновлению различных объектов конфигурации по заданиям трейдеров, причём один такой пользователь редко использовал несколько вкладок одновременно.

С учётом этих особенностей, было предложено и реализовано наиболее значительное изменение в работе репозиториев — lazy load (отложенная, «ленивая» загрузка).

Основные ее принципы описаны в паттерне Lazy Load [1], где утверждается, что объект поддерживает отложенную загрузку, если он не содержит все требуемые данные, но способен получить их при необходимости.

Согласно этому утверждению было решено изменить функционирование репозиториев так, чтобы изначально все они были пусты и загружали данные лишь при открытии пользователем соответствующей части интерфейса. При этом возник вопрос — надо ли очищать репозитории после того, как пользователь закончил работу с его данными (другими словами, закрыл вкладку с конфигурацией). Было принято решение осуществлять подобную очистку, потому что в течение непродолжительного времени пользователь мог бы открыть последовательно все вкладки, что при отсутствии очистки оставило бы только одну возможность сократить расход памяти — перезаписи интерфейса. Такая ситуация крайне нежелательна, так как пользователь теряет контроль над платформой во время перезапуска. Разумеется, при каждом открытии вкладки необходимо время на загрузку данных, но практика показала, что это время невелико и приемлемо для пользователей.

Реализация репозиториев была изменена следующим образом. Во-первых, были введены 3 состояния — EMPTY, LOADING, FULL. В состоянии EMPTY репозиторий игнорирует изменение ревизии, что снижает издержки на синхронизацию репозиториев как на сервере, так и в пользовательских интерфейсах. Состояние репозитория периодически проверяется отдельным потоком, который при наступлении состояния LOADING запрашивает данные с сервера, переводя репозиторий в состояние FULL. Во-вторых, был изменен интерфейс Repository — у большинства операций появились аналоги, в параметрах которых указывается коллбэк, вызываемый после того, как данные загружены. Преимущественно в коллбэках выполняется инициализация и обновление графических

компонент, например, таблиц.

Метод find до оптимизации:

```
public Map<Key, T> find() {
    lockCache();
    try {
        return copyCache();
    } finally {
        unlockCache();
    }
}
```

Метод find после оптимизации:

```
public void find(final Callback<Map<Key, T>> callback) {
    final Map<Key, T> copiedCache;
    lockCache();
    try {
        if (!isFull()) {
            setState(LOADING);
            callbackQueue.add(new Runnable() {
                    callback.call(copyCache());
                });
            return;
        }
        copiedCache = copyCache();
    } finally {
        unlockCache();
    }
    callback.call(copiedCache);
}
```

3. Меры оптимизации использования памяти, применённые помимо использования отложенной загрузки репозиториев

Дальнейшие оптимизационные шаги проводились в отношении доменных объектов.

Сокращение копирования объектов также позволило сократить потребление памяти. Изменения доменных объектов, осуществляемые и сохраняемые другими пользователями, отображались в интерфейсе определённого пользователя с помощью листенеров RepositoryListener, связанных с моделями данных различных визуальных компонентов — например, множество валют отображалось в двух различно выглядящих таблицах с разными наборами атрибутов-столбцов. Каждый листенер всегда копировал доменные объекты (модифицированные

другими пользователями) перед передачей их модели визуального компонента. Очевидно, такое защитное копирование (defensive copy) избыточно, и копии объектов должны создаваться лишь при модификации их атрибутов, что и было реализовано при оптимизации.

В Java для каждого примитивного типа данных (int, double, long, boolean, byte) имеется класс-оболочка (Integer, Double, Long, Boolean, Byte). Объекты таких классов занимают больше памяти, чем примитивные значения, и создаются, например, в подобных фрагментах кода:

```
int x = 10;
Integer y = x;
```

Во многих случаях объекты-оболочки избыточны, и код доменных объектов был проверен, чтобы устранить проявления такой избыточности — например, с помощью следующего рефакторинга:

```
public Integer getPipSize();
public void setPipSize(Integer pipSize);
public int getPipSize();
public void setPipSize(int pipSize);
```

Кроме того, некоторые доменные объекты имели свойства, поддерживаемые Java-коллекциями, элементами которых являются объекты-оболочки (например, Map<Integer, Integer>). В таких случаях могут быть использованы коллекции, реализованные на основе примитивных типов данных. Например, такие коллекции предоставляет HPPC (High Performance Primitive Collections for Java).

Помимо этого было установлено, что во многих доменных объектах методы hashCode и equals были реализованы с помощью классов HashCodeBuilder и EqualsBuilder из Apache Commons. Другими словами, лишние объекты HashCodeBuilder и EqualsBuilder создавались при каждом вызове hashCode и equals, что часто происходит при использовании доменных объектов в коллекциях — например, когда вызываются операции contains или remove. Сложно найти достоинства такой реализации hashCode и equals, кроме как повышение выразительности кода. hashCode и equals в случае использования HashCodeBuilder и EqualsBuilder преобразовывались следующим образом:

```
public int hashCode() {
  return new HashCodeBuilder().append(x).append(y).toHashCode();
}

public int hashCode() {
  return 31 * x + y;
}
```

Сравнительный анализ доменных объектов позволил определить 2 наиболее «тяжеловесных», значительно превосходящих по своему размеру остальные.

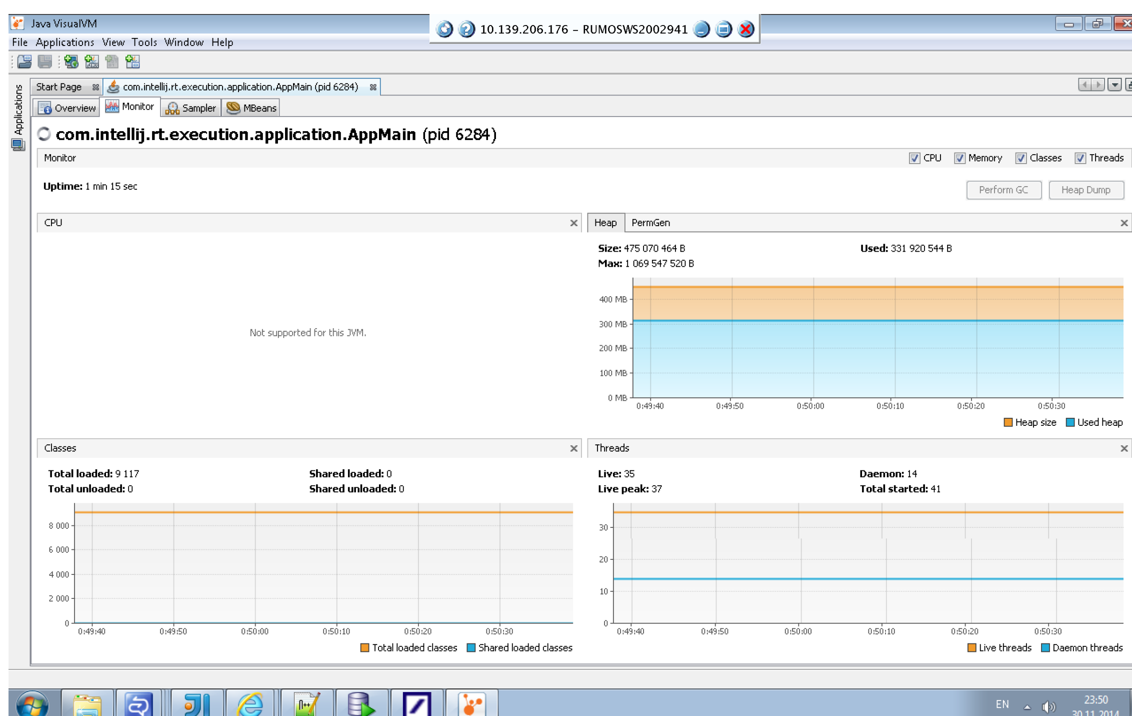

Рис. 3. Потребление памяти интерфейсом после оптимизации

Они имеют уникальные цифровые идентификаторы и строковые дескрипторы, генерируются специальными скриптами и загружаются из сгенерированных файлов в интерфейс. Упомянутые идентификаторы использовались в свойствах других разновидностей конфигурации для ссылки на генерируемые объекты, причём пользовательский интерфейс для визуализации этих ссылок использовал строковые дескрипторы, получение которых требовало загрузки «тяжеловесных» объектов целиком.

Интерфейс представляет экземпляры «тяжеловесных» объектов пользователю как обычный текст при необходимости их частичного редактирования, которое происходит сравнительно редко.

В отношении выявленных «тяжеловесных» объектов были применены 2 изменения:

1. Разделение «тяжеловесного» объекта на 2 — «лёгкий» объект, состоящий из идентификатора и дескриптора, и «тяжёлый» объект, состоящий из идентификатора и сгенерированных данных. Выделение «лёгкого» объекта позволило облегчить ссылки на «тяжеловесные» объекты при визуализации других частей конфигурации. Связь обоих объектов осуществляется с помощью идентификатора.

2. К сгенерированным данным «тяжёлого» объекта было применено сжатие, превратившее их в BLOB (binary large object, большой бинарный объект) [1]. Интерфейс проводит обратное преобразование в текстовый формат лишь при необходимости редактирования; сервер проводит данное преобразование перед отправкой «тяжеловесных» конфигураций бизнес-компонентам.

4. Заключение

К сожалению, измерение того, насколько сократилось потребление памяти благодаря каждому оптимизационному изменению не проводилось, и была получена лишь интегральная оценка во время тестирования перед единовременным релизом всех описанных изменений. Эта оценка иллюстрируется, например, следующим скриншотом из jPofiler (рис. 3).

В целом уже в первую неделю использования оптимизированной версии интерфейса от пользователей перестали поступать жалобы и нарекания, относящиеся к большому потреблению памяти приложением. Если до оптимизации расход памяти измерялся сотнями мегабайт, то оптимизированный интерфейс потребляет несколько десятков мегабайт у большинства пользователей. Подводя итог, можно увидеть, что некоторые изменения (применение паттернов Lazy Load и BLOB) относятся к улучшению технического дизайна приложения с учётом особенностей его использования, а некоторые — к улучшению качества кода (например, устранение избыточного использования объектов-обёрток).

Литература

1. Фаулер М., Райс Д., Фоммел М., Хайет Э., Ми Р., Стаффорд Р. Шаблоны корпоративных приложений. М. : Вильямс, 2010. 544 с.
2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб. : Питер, 2001. 368 с.

REDUCING OF MEMORY USAGE FOR THE USER INTERFACE MANAGING THE BANK TRADING PLATFORM CONFIGURATION

A.I. Zuravlev
Post-graduate student, e-mail: grusalex@gmail.com
D.N. Lavrov
Ph.D. (Eng.), Associate Professor, e-mail: lavrov@omsu.ru

Omsk State University n.a. F.M. Dostoevskiy

Abstract. The article describes methods for reducing of memory consumption in a Java-application which is a graphical user interface intended to configure a bank trading platform. The most significant change in the implementation of the interface is use of the Lazy Load pattern for repositories of domain objects that became possible due to identification of two basic use cases of the application.

Keywords: graphical user interface, memory consumption, lazy load of data, repository, domain object.