УДК 004.75

А. В. Васильев

Московский физико-технический институт (государственный университет)

Шаблон проектирования корпоративных Java-приложений, построенных на основе адаптивных моделей данных, обеспечивающий их масштабируемость

В некоторых областях, таких как телекоммуникация, медицина, образование, происходят достаточно частые изменения структур данных и требований к системам. Поэтому для уменьшения издержек на разработку таких систем компании применяют подходы, построенные на адаптивных моделях данных, — начальные затраты на разработку большие, однако в дальнейшем изменения обходятся достаточно дешево. На практике код, работающий с адаптивными моделями, оказывается невозможно покрыть автоматическими тестами, а система оказывается практически не масштабируемой, так как работает напрямую с реляционными структурами. В работе предложен подход к проектированию таких систем, обеспечивающий возможность тестирования и позволяющий заменить уровни хранения данных без каких-либо сложностей по мере роста количества информации.

Ключевые слова: Java, адаптивная модель данных, масштабируемые системы, автоматическое тестирование, шаблон проектирования корпоративных систем.

Изложение работы состоит из нескольких частей. В начале рассмотрены реляционные базы данных, рассказывается о применимости их для высоко нагруженных систем. Рассмотрены адаптивные модели данных, описаны их достоинства и недостатки. Дан пример применения современных In Memory Data Grid систем. Описан шаблон проектирования систем, решающий обозначенные проблемы. Во всех параграфах рассматривается простой пример, полностью описывающий основную идею исследования.

Предложенный подход решения использует возможности языка Java 5, очерчены основные моменты реализации, которые без ограничения общности могут быть применены для адаптивных систем любой сложности.

1. Введение

В настоящее время разработка заказных программных продуктов стала распространенной практикой, так как все больше компаний используют информационные системы для оптимизации бизнеса. В ходе разработки компаниям необходимо определиться с технологиями (если такое возможно), на основе которых будут реализованы подсистемы хранения и доступа к данным, автоматизации технологических процессов и их учета, аналитики, контроля бизнеса и так далее. Также возникают вопросы уровня контроля качества продукта, стоимости изменений, стоимости дальнейшей поддержки системы. Серьезные игроки помимо всего прочего задумываются о масштабируемости систем — будет ли возможна в принципе эксплуатация системы, если объем данных вырастет, скажем, на порядок?

Последние тенденции говорят о том, что взгляд на способ масштабирования меняется. Вертикальное масштабирование (то есть покупка более мощного аппаратного уровня) неоправданно дорого и имеет свои физические ограничения. Горизонтальное же масштабирование не подразумевает замену старого аппаратного окружения, а подразумевает покупку недостающих мощностей. Об этом факте свидетельствует появление ставших популярными в последнее время облачных технологий, систем «software on demand». Архитектурно, горизонтально масштабируемая система может представлять собой кластер, облако как построенное на собственных мощностях, так и на покупаемых у поставщиков таких услуг.

И, безусловно, горизонтально масштабируемые системы являются в первую очередь программной новацией.

Если взглянуть на историю, то в 90-х годах XX века появились работы (например, [1]), посвященные подходу к разработке систем на основе адаптивных моделей данных, которые обеспечивали очень высокую гибкость, но были по сути своей не масштабируемыми, в том числе из-за того, что были построены на основе реляционных базах данных. И теперь фактор «такого исторического наследия» мешает компаниям строить новые решения на имеющейся инфраструктуре.

2. Почему реляционный подход не работает

Как было упомянуто выше, классические реляционные базы данных по сути своей не являются масштабируемыми [2]. Безусловно, существуют подходы, которые помогают частично решить проблемы с производительностью, среди которых можно отметить: предварительный подсчет промежуточных результатов (является «заглушкой», которая решает проблемы локально), секционирование таблиц и индексов (требует грамотного администрирования и по сути все так же хранится в одном хранилище), покупку специализированных аппаратных решений (например, Oracle RAC). Однако на действительно больших объемах данных или при большом количестве транзакций реляционный подход перестает работать эффективно. Об этом факте свидетельствует появление InMemory баз данных, которые используются, например, такими сервисами, как Facebook, eBay.

Стоит отметить, что хотя описанные выше проблемы и имеют место, но с ними компании сталкиваются не сразу, поэтому реляционные базы данных все же присутствуют на рынке и справляются со своими задачами, хоть и в ограниченных рамках.

3. Общее устройство адаптивных моделей

В 90-х годах XX века была введена новая концепция адаптивных моделей данных (AOM, Adaptive Object Model), которая была весьма гибкой, хотя и была трудна для понимания неподготовленным специалистом. Адаптивная модель все считает сущностями (Entity), различающимися своим типом (Entity Type). Свойства сущностей в такой концепции принадлежат всему типу целиком.

Рассмотрим классический подход, когда под каждую новую сущность создается своя реляция. Допустим, имеется таблица Cars, в которой есть строка с именем MyCar, где колонке Color записано значение White. Как реляцию это можно записать следующим образом:

```
Cars: {Name = MyCar, Color = White}
```

Если человек (Me) владет этой машиной, то в таблице People могла бы быть следующая запись:

```
People: {Name = Me, Own = MyCar}
```

Теперь рассмотрим хранение описанных данных на примере упрощенной адаптивной модели. В терминах AOM эта же сама запись может быть записана следующим образом:

```
EntityType: {Name = Car}
Entity: {Name = MyCar, EntityType = Car}
Property: {Entity = MyCar, Property = Color, Value = White}

EntityType: {Name = Person}
Entity: {Name = Me, EntityType = Person}
Property: {Entity = Me, Property = Owns, Value = MyCar}
```

Таким образом, любую сущность (человек, дом, компьютер, IP-адрес) можно сохранить в одних и тех же таблицах, различая их по типам (EntityType). Подход позволяет с минимальными затратами изменять систему в зависимости от сложности адаптивной модели и предоставляемой ей функциями.

4. Практические применения и расширения адаптивных моделей

Когда адаптивные модели стали применять в реальных проектах, то применять их стали не «as is», а внося изменения в терминологию, вводя новые абстракции и специфику их обработки. Например, выделяют еще References, Attribute types и пр. Говорили даже о вырожденных случаях, когда обобщающая теория «ужимала» все до двух таблиц, — но такие подходы не прижились, так как были неоправданно сложными. На практике, наоборот, старались денормализовать модели для более эффективного их использования, и прижились адаптивные модели, где весь функционал системы умещался в десятке-другом таблиц. Такие решения были обоснованы в компаниях, где происходили частые изменения набора сущностей и их характеристик, например в телекоммуникационной области, медицине. Стоимость изменений была минимальна, что давало несомненные преимущества. В принципе такой подход применим в любой области, если известно, что будут происходить частые изменения набора сущностей и их параметров. В чем же проблема? Проблемы начинаются тогда, когда необходимо произвести с данными какие-либо бизнес-операции.

Также хотелось бы привести пример кода, основанного на сущностях AOM, который меняет цвет машины в примере, описанном выше по какому-либо условию (например, владелец запустил бизнес-операцию по смене цвета машины). Программирование с использованием AOM так и выглядит:

```
Entity car = EntityService
   .findEntityByEntityTypeAndEntityName("Car", "MyCar");
ParameterValue newColor = car.getParameter("Color");
newColor.setValue("Black");
car.setParameter(newColor);
```

В приведенном примере видно, что операция осуществляется в рамках бизнес-модели, которая описывается константами «Car», «Color». Весьма неудобный способ, по сравнению с классическим, с использованием конкретных сущностей, так как требует много дополнительного кода.

5. Подход к тестированию решений

Следующей проблемой, с которой столкнулись компании, работающие с AOM, было тестирование моделированных предметных областей. С приходом jUnit и методологий TDD [3] оказалось, что код, написанный в рамках Entity — EntityType — Property, очень тяжело тестировать. В частности, нельзя протестировать модульные решения, так как бизнес-логика модулей перемешана с объектами AOM. В случаях, когда тестирование все же можно было произвести, то оно оказывалось неоправданно дорогим (приходилось писать интеграционные тесты, требующие серьезного окружения). Не хватало уровней между AOM и логикой работы; напротив, вводить новый уровень и заниматься его поддержкой никто не хотел и не хочет.

6. Распределенные системы

Как уже было упомянуто, согласно тенденциям последних лет, способ хранения и обработки информации будет переходить от централизованных хранилищ к распределенным. Уже сейчас системы NoSql и NewSQL завоевывают популярность, так как они предоставляют отказоустойчивое хранение данных, позволяют проводить аналитику и много другое,

обеспечивая соответствие стандартам разработки корпоративных систем. Одним из лидеров является библиотека Hadoop.

Также в качестве примера хотелось бы привести пример кода, отвечающего за сохранение POJO объекта в хранилище для GemFire (In Memory Data Grid [4]):

```
@Entity public class Car implements Serializable {
   String name, color;
}
@Region public class CarRegion implements Region{ ... }
public static void main(String ... args)  {
   CarRegion carRegion = ... ;
   Car car = new Car("MyCar", "White");
   carRegion.put(car);     }
```

В этом примере под @Region подразумевается распределенное хранилище для типа Car, оно реализует интерфейс Map. Заметим, что тут предметная область смоделирована в терминах классов и их полей. Распределенные хранилища позволяют сохранять такие объекты.

7. Создание интеграционного решения для AOM и распределенных приложений

Выше были описаны некоторые проблемы, с которыми сталкиваются компании в процессе разработки. Так как же быть компаниям? Выбирать между гибкостью или производительностью? Предусматривать ли возможность делать тестирование? Что делать, если объемы данных вырастут и будет необходимо поддерживать существующие решения и одновременно разрабатывать новые? Суть этих вопросов в понимании данной статьи заключается в противопоставлении плюсов и минусов адаптивных моделей как в настоящем, так и в будущем.

Проанализировав имеющиеся факты, существующие решения и заметные тенденции, было предложено решение, которое позволяет решить описанные проблемы, вести эффективную разработку в настоящем и иметь возможность легко «перебраться» на другие базы данных в дальнейшем.

Идея заключается в том, чтобы иметь бизнес-модель в виде, позволяющем:

1. проводить тестирование описанным выше способом,
2. иметь единожды заданные значения констант соответствия сущностей в AOM к сущностям в модели,
3. иметь возможность легко заменить AOM-модель на любое другое хранилище, если этого потребует конкретная предметная область.

Решение было сделано в виде набора связанных между собой интерфейсов, аннотированных константами. Сначала приводится пример такого отображения, после чего рассматриваются подробности и аспекты его работы.

Итак, беря за основу бизнес-модель, описанную выше, мы имеем следующее Java-представление:

```
@EntityType("Car") interface Car {
   @Parameter("Color") String getColor();
   @Name String getName();
}
@EntityType("Person") interface Person {
   @Name String getName();
```

```
    @Parameter("Owns") Car getCar();
}
```

Подразумевается, что аннотации, приведенные выше, уже созданы.

В описании введена специальная аннотация @Name — так как обычно имя не выделяют как обычный параметр, а с целью удобства восприятия человеком хранят его вместе с сущностями (Entity) в таблицах. Описанная выше модель обслуживается специальным сервисом, который позволяет по идентификатору (первичному ключу) находить соответствующий объект. Объект создается динамически, на основе интерфейса. При обращении к методам интерфейса вызовы транслируются на соответствующие обращения к объектам адаптивной модели.

Пример реализации такого типизированного сервиса:

```
interface BusinessService {
    <T> T findById(Class<T> clazz, String id);
    void store(String id, Serializable entity);
}
```

Таким образом, обращение к нему выглядит следующим образом:

```
BusinessService businessService = ... ;
Person me = businessService.findById(Person.class, "Me");
String personName = me.getName();
Car myCar = me.getCar();
String color = myCar.getColor();
```

Перейдем к рассмотрению реализации метода findById на поверхностном уровне, так как это не является целью статьи и хотелось бы только в общем виде обозначить основные моменты реализации.

1. Создание динамических экземпляров классов осуществляется посредством стандартных методов, предоставляемых Java:

   ```
   java.lang.reflect.InvocationHandler handler
      = new AomInvocationHandler(...);
   Car car = (Car) java.lang.reflect.Proxy.newProxyInstance(
      getClassLoader(), new Class[]{Car.class}, handler);
   ```

   Теперь внутри handler-a можно перехватить любое обращение к интерфейсу и возвратить значение, соответствующее прописанной аннотации.

2. Получение аннотации делается внутри handler-a, там же можно получить идентификатор параметра. Пример получения значения Color из аннотации для метода getColor может быть сделано примерно следующим образом:

   ```
   java.lang.reflect.Method method = ... ;
   Parameter param = method.getAnnotation(Parameter.class);
   String paramId = param.value();
   ```

3. Перейдём к вопросу о тестировании подобного рода систем. Допустим, что у нас есть метод, который возвращает цвет машины человека: `String getCarColor(Person person)`. Для его тестирования теперь достаточно просто создать окружение и не нужно иметь реального хранилища данных (использованы элементы API концепции TDD, например Mockito):

```
Car carMock = mock(Car.class);
when(carMock.getColor()).thenReturn("Black");
Person personMock = mock(Person.class);
when(personMock.getCar()).thenReturn(carMock);
assertEquals("Black", tested.getCarColor(personMock));
```

Такого рода тесты невозможны при отсутствии описанного уровня на основе AOM.

8. Заключение

Описанное решение применяется в проекте, над которым работает автор статьи, и практические тесты показывают незначительные (примерно 2–3% времени) дополнительные нагрузки, что свидетельствует о применимости описанного подхода в очерченных рамках. Нагрузки обусловлены созданием дополнительных классов, что является достаточно простой операций.

Новым в работе является идея введения дополнительного уровня между AOM и кодом, выполняющим бизнес-логику приложения. Описаны проблемы, которые он призван решить и описаны преимущества его введения: возможность масштабирования системы при изменении AOM на другую и обеспечение корректного модульного тестирования бизнес-логики.

Литература

1. *Joseph W. Yoder, Reza Razavi.* Metadata and Adaptive Object-Models // ECOOP'2000 Workshop Reader. Lecture Notes in Computer Science. — 2000. — V. 1964.
2. *Marc Seeger.* Key-Value Stores: a practical overview. — Computer Science and Media Ultra-Large-Sites SS09 Stuttgart, Germany. — 21 September 2009.
3. *Kent Beck.* Test-Driven Development by Example. — Addison Wesley, 2002. — 240 p.
4. Jack Belzer. Encyclopedia of Computer Science and Technology. — V. 14. Very Large Data Base Systems to Zero-Memory and Markov Information Source / Marcel Dekker Inc. 1980.

Поступила в редакцию 04.12.2012