Знакомство со Spring Framework

1 Введение

В мире программирования на Java существует набор API для решения типичных задач (взаимодействие с хранилищами данных, в том числе БД, отображение данных, в том числе на web-страницах, и другие). Эти программные интерфейсы нельзя считать достаточно последовательно спроектированными друг относительно друга (это и понятно, создавались они независимо). Программы, которые обычно используют сразу несколько таких интерфейсов, наследуют эту непоследовательность. Spring призван предоставить программные решения, которые помогут создавать лучшую архитектуру для ваших программ, сделать их более логически прозрачными, удобочитаемыми и, как следствие, снизить подверженность ошибкам. Эти решения заключаются, в основном, в обертках для традиционных средств решения наиболее распространенных задач, последовательно реализующих наиболее удачные паттерны проектирования, и средствах управления зависимостями частей ваших проектов.

Подходя к проблемам, решаемым Spring более предметно, будем опираться на понятие «бизнес-приложения». Сразу стоит отметить, что слово «бизнес» здесь, подсказано термином «бизнес-логика» (см. ниже), которое не имеет никакого отношения к некоторому бизнесу в смысле коммерческой деятельности, а обозначает совокупность правил и зависимостей предметной области, которая отражается вашей программой. Итак под бизнес-приложением будет пониматься приложение, функциональность которого может быть разделена на три основных слоя:

- слой интерфейса пользователя (зачастую web-интерфейс)
- слой бизнес-логики
- слой данных (зачастую РБД)

Термин «слой» (tier или layer, зачастую также будет использоваться слово «уровень») указывает на то, что порядок их перечисления важен, то есть бизнес-логика вызывается посредством интерфейса пользователя, получает данные из слоя данных, обрабатывает их и, возможно, выдает результат через интерфейс. Можно возразить, что соответствующие уровни выделяются в любом приложении. В таком случае все упирается в принципы этого выделения, пока можно ограничиться пояснением, что каждый слой решает свои задачи с помощью специфических инструментов (в частности, определенных библиотек и внешних приложений, таких как сервер баз данных или шаблонный движок, осуществляющий рендеринг пользовательского интерфейса), таким образом создавая новые «слое-специфичные» зависимости в вашем проекте.

Одной из проблем существующих API является «интрузивность» (дословно — навязчивость). К сожалению, формальных определений этого понятия автору неизвестно, что приводит к необходимости давать пояснения исключительно на примерах или с помощью других «интуитивно понятных», то есть неопределенных понятий. В двух словах: посмотрев на прикладной код, использующий интрузивное API, можно точно сказать, что используется какое-то конкретное инородное для данного кода программное решение, но сложно сказать, какое же назначение имеет сам код. Иначе, логика вашего кода диктуется в большей степени используемым API, чем логикой решаемой задачи. Добавим сюда и такое объяснение: вы, наверное, уже успели заметить, что понятие «зависимости» играет в построении бизнес-приложений особую роль. Интрузивное API создает весьма жесткую зависимость в вашем проекте, если вы захотите его сменить, то вероятнее всего значительную часть кода придется просто выбросить.

Повторим теперь снова, что Spring это средство для управление зависимостями плюс создания последовательного дизайн приложения.

В следующем посте планируется больше конкретики, а именно, перечисление модулей Spring. Потом, я надеюсь, будут какие-то примерчики.

2 Состав Spring

Spring представляет собой набор «модулей», независимых друг от друга (программист может выбрать набор модулей необходимых ему и не использовать остальные). Основные модули, пришедшие еще из первой версии Spring, делятся на две части. Первая часть это модули, которые поддерживают особые техники или стили программирования: Inversion of Control и аспектно-ориентированное программирование. Вторая часть это модули содержащие классы-обертки для распространенных инструментов решения типичных задач (взаимодействие с БД, создание веб-интерфейса). В основном они полагаются на более низкоуровневые в некотором смысле, интрузивные API. При использовании оберток, предоставляемых Spring, достигается большая свобода от конкретных API, повышается логическая прозрачность кода. Перечислим модули, дав общую характеристику каждому.

IoC-контейнер — это особый модуль, являющийся ядром Spring (часто называется Core Container). Он составляет исключение сказанному выше о независимости: от него зависит большая часть остальных модулей. Техника Inversion of Control подразумевает следующее: если некоторый объект A в приложении зависит от другого объекта B (скажем, имеет поле типа B), то получением объекта B для A занимается IoC-контейнер, а не сам объект A; контейнер, исходя из своих настроек, решает, нужно ли создать новый

объект B, или получить его из пула, который также может поддерживаться контейнером, или еще что-то. Эта простая идея (в духе «разделяй и властвуй») дает немало преимуществ, как-то: снижается зацепление разных частей программного обеспечения друг между другом; пропадает необходимость многократного повторения кода поиска зависимостей в прикладном коде, последний становится более прозрачным для понимания; облегчает модульное тестирование.

АОП-модуль. Модуль для поддержки аспектно-ориентированного программирования. АОП стало ответом на несостоятельность объектного подхода (ставшего мейнстримом в программировании 90-х годов) в отношении так называемой сквозной функциональности (cross-cutting concerns), которая не поддавалась объектной декомпозиции и способствовала превращению кода, написанного на чистом ОО-языке, в «слоенный пирог». Типичными примерами сквозной функциональности являются логирование, управление транзакциями, управление авторизацией: обращение к этим средствам пронизывает прикладной код, хотя не имеет прямого отношения к решаемой им задаче. Spring предоставляет две альтернативы: свою реализацию ключевых сущностей АОП, как они определены AOP Alliance, объединением, разрабатывающим подходы к реализации АО-парадигмы в Java, или возможность относительно бесшовной интеграции вашего прикладного кода на Java со средствами наиболее популярного АОП-языка AspectJ.

Модуль доступа к данным. Java располагает стандартным программным пакетом для взаимодействия с реляционными базами данных JDBC. Тот, кто видел и писал код с использованием JDBC, может подтвердить, что такой код выглядит удручающе: JDBC довольно низкоуровневое средство. В связи с этим программисты снова и снова сталкивались с необходимостью

писать обертки для него, чтобы минимально смешивать свой код с кодом использования JDBC. Именно на такого сорта опыте основана часть этого модуля. Другая обеспечивает определенный уровень абстракции при использовании какого-либо из популярных решений объектно-реляционных отображений (ORM, object-relational mapping). Суть ORM заключается в том, что объектная модель данных, в русле которой чаще всего создается программное обеспечение, входит в противоречия с реляционной моделью данных, которая используется для хранения данных в РБД. Долгое время эта проблема решалась написанием вручную кода отображающего одну модель в другую. Следом стали появляться инструменты, автоматизирующие этот процесс. Spring подводит некоторую черту под опытом, накопленным различными производителями ORM-решений, позволяя абстрагировать ваш код от конкретного средства (заметим снова, что сам Spring не предоставляет такого решения, оставляя, как обычно, его выбор за вами, но доступ к такому средству через обертки Spring может иметь уже не раз упоминавшиеся преимущества).

Веб-модуль и Spring MVC. Рассчитано, как ясно из названия, на использование при построении веб-приложений. Model-View-Controller весьма популярная модель архитектуры веб-приложения (создавалась, правда, задолго до появления службы www), призванная, разумеется, четко разделять обязанности в нем. Ее не стоит путать с моделью бизнес-приложения, на которую мы решили ориентироваться. Если попытаться отобразить одно на другое, то получится так: View и Controller располагаются в презентационном слое (первое связано с конкретной технологией отображения информации, а второй инкапсулирует поведение вашего приложения с точки зрения пользователя: реакцию на нажатие кнопок, перехода по ссылкам), на долю Model приходится слой бизнес-логики. Для Java существует огромное количество веб-фреймверков, так или иначе реализующих MVC,

Spring MVC стал одним из них, а веб-модуль Spring поддерживает интеграцию вашего приложения с одним из таких фреймверков, подтягивает возможности IoC-контейнера к более специфическим особенностям web и, кроме того, предоставляет готовые решения для некоторых мелких, часто встречающихся здесь задач, таких как, к примеру, загрузка файлов по частям.

Перечислены основные модули Spring, однако далеко не все. Я упомяну более мелкие или мало используемые (отчасти по причине того, что появились они недавно) средства совсем бегло. Но для начала требуется сказать следующее. Одну из главных ролей в мире создания бизнес-приложений на Java играет, как нетрудно догадаться, компания Sun, которая выпускает многочисленные спецификации, детально описывающие способы решения широкого круга задач. Эти спецификации объединены под общим названием Java Enterprise Edition (Java EE). Spring изначально создавался в противовес наиболее тяжеловесным частям Java EE.

Однако с существенной частью Java EE Spring успешно взаимодействует (не следует путать взаимодействие с системами, реализующими спецификации Sun, и саму реализацию: Spring предоставляет первое); часть оставшихся модулей поддерживают взаимодействие с такими спецификациями: Java Management Extensions (JMX, стандарт на средства мониторинга и конфигурирования приложений), Java EE Connector API (JCA, стандарт на взаимодействие с legacy-системами, в том числе не-Java системами, и вообще на интеграцию создававшихся независимо информационных систем).

И остается группа модулей, направленных на поддержку создания распределенных систем. Модуль удаленного взаимодействия (Remoting): сюда входит абстрагирование от конкретных технологий, упомянем две наибо-

лее типичные: RMI и JAX-RPC (соответственно, бинарный и текстовый XML протоколы удаленных взаимодействий). Модуль для асинхронного взаимодействия (взаимодействия на основе передачи сообщений), предназначенный для связи с системами, поддерживающими еще одну спецификацию Java EE — Java Message Service (JMS). И, наконец, отдельный фреймверк, не входящий в состав Spring Framework, но доступный для скачивания на сайте Spring: Spring Web Services, который как следует из названия, предоставляет ряд средств, облегчающий создание веб-служб (распределенных систем, базирующихся на нескольких спецификациях W3C, описывающих формат сообщений, язык описания интерфейсов системы и другое, для которых характерно плотное использование XML).

В заключение стоит упомянуть еще один фреймверк с которым у Spring давно сложились хорошие отношения, и сейчас он почти часть Spring, это Acegi Security System: система для предоставляющая решения авторизации и аутентификации в приложениях (в основном, веб).

3 Пример

Наша цель состоит в демонстрации возможностей IoC-контейнера Spring, являющегося, как было отмечено во втором посте, ядром всего фреймверка и служащего для управления зависимостями в приложении. При построении программной системы в этом примере мы будем ориентироваться на модель бизнес-приложения, описанную в первом посте. В качестве примера мы рассмотрим информационную систему, предоставляющую сведения о теннисных турнирах: она должна содержать информацию о проводимых матчах, участвующих игроках, составлять расписание турниров. Естественно, мы не будем приводить весь исходный код, необходимый для функционирования такой системы, а ограничимся лишь эскизом, позволя-

ющим обозначить проблемы при построении корпоративных приложений и познакомить с механизмами их решения, предлагаемыми Spring.

Краткая характеристика приложения. На уровне интерфейса используется библиотека визуальных компонент Swing (входит в состав стандартной библиотеки Java): это решение по-существу означает, что мы не будем останавливаться на особенностях организации (более традиционного) веб-интерфейса, и вообще: то, что касается конструирования графического интерфейса, будет по большей части опущено, акцент сделан на его связи со слоем бизнес-логики. Слой доступа к данным будет реализован с помощью распространенного паттерна Data Access Object (DAO), который инкапсулирует запросы к некоторой базе данных; в частности, с его помощью достигается важная цель: концентрация использования SQL-запросов в одном месте программы (наборе DAO-класов).

Теперь дадим более подробное описание оставшегося слоя: слоя бизнес-логики. Начнем с описания интерфейса (в смысле интерфейса в ООП, а не графического интерфейса), обеспечивающего функционирование теннисного турнира. Мы поместим в него всего один метод, отвечающий за старт матча с заданным идентификатором. Проблемы, которые могут появиться: некорректный идентификатор матча, матч уже закончился, матч не может быть начат из-за того, что не завершились предыдущие, матч не может состояться по каким-либо другим причинам (погода, травма или отказ игрока) — отражены в возможных исключениях. Если ничего из перечисленного не произошло, возвращается объект, представляющий матч.

```
package edu.sfedu.mmcs.it.spring;

public interface TournamentMatchManager {
```

```
  Match startMatch(long matchId) throws
    UnknownMatchException, MatchIsFinishedException,
    PreviousMatchesNotFinishedException,
    MatchCannotBePlayedException;
}
```

Класс, реализующий этот интерфейс:

```
package edu.sfedu.mmcs.it.spring;

public class DefaultTournamentMatchManager implements
        TournamentMatchManager {
  private MatchDao matchDao;

  public void setMatchDao(MatchDao matchDao) {
    this.matchDao = matchDao;
  }

  protected void verifyMatchExists(long matchId) throws
          UnknownMatchException {
    if (!this.matchDao.doesMatchExist(matchId)) {
      throw new UnknownMatchException();
    }
  }

  protected void verifyMatchIsNotFinished(long matchId) throws
          MatchIsFinishedException {
    if (this.matchDao.isMatchFinished(matchId)) {
      throw new MatchIsFinishedException();
    }
  }

  /* другие методы пропущены для краткости */
```

```
  public Match startMatch(long matchId) throws
          UnknownMatchException, MatchIsFinishedException,
          PreviousMatchesNotFinishedException,
          MatchCannotBePlayedException {
    verifyMatchExists(matchId);
    verifyMatchIsNotFinished(matchId);
    Players players = null;
    if (doesMatchDependOnPreviousMatches(matchId)) {
      players =
          findWinnersFromPreviousMatchesElseHandle(matchId);
    } else {
      players = findPlayersForMatch(matchId);
    }
    return new Match(players.getPlayer1(),
                                players.getPlayer2());
  }
}
```

Данная реализация нуждается в минимальных пояснениях: интерфейсный метод startMatch(), являющийся частью нашей бизнес-логики, состоит из последовательной проверки нескольких условиий («бизнес-правил»), основанной на содержимом базы данных; все обращения к базе данных происходят посредством DAO-объекта. MatchDao является представителем слоя доступа к данным, от которого зависит бизнес-логика:

```
package edu.sfedu.mmcs.it.spring;
public interface MatchDao {
  boolean doesMatchExist(long matchId);
  boolean isMatchFinished(long matchId);
  boolean isMatchDependantOnPreviousMatches(long matchId);
  boolean arePreviousMatchesFinished(long matchId);
```

```
  Player findWinnerFromFirstPreviousMatch(long matchId);
  //...
}
```

Классы, реализующие MatchDao, вероятно, должны иметь доступ к базе данных, содержащей необходимую информацию. Создание соединений с базой данных для отправки запросов отведено на долю классов, реализующих стандартный интерфейс javax.sql.DataSource (классы эти мы не пишем сами, а также получаем в готовом виде; на этой детали не стоит заострять внимание).

Посмотрим, как может быть реализован класс, реализующий MatchDao. В этом месте применена маленькая хитрость: вначале декларировалось использование Spring в части только IoC-контейнера, но в данном случае просто грех не воспользоваться классом, принадлежащем к модулю доступа к данным Spring Framework (см. второй пост серии) — JdbcTemplate; он предоставляет исключительно удобную обертку над стандартным низкоуровневым API для взаимодействия с РСУБД в Java — JDBC. Объект этого класса, как и следовало ожидать, зависит от объекта, реализующего интерфейс DataSource, и позволяет выполнять разнообразные запросы к соответствующей полученному DataSource базе данных.

```
package edu.sfedu.mmcs.it.spring;

import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcMatchDao implements MatchDao {
  private JdbcTemplate jdbcTemplate;
```

```
  public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }

  public boolean doesMatchExist(long matchId) {
    return 1 == jdbcTemplate.queryForInt(
       "SELECT COUNT(0) FROM T_MATCHES WHERE MATCH_ID = ?",
       new Object[] { new Long(matchId) }
    );
  }
  /* другие методы пропущены для краткости */
}
```

Остается последний штрих: класс, отвечающий за графический интерфейс пользователя и точка входа в программу, где будут создаваться основные классы программы (сделаем её статическим методом в этом же классе). Для инициализции DataSource нужны реквизиты доступа к базе данных: аккаунт (имя пользователя БД и пароль), адрес БД и имя так называемого класса-драйвера JDBC. Единственное, что важно тут понимать, так это то, что в больших приложениях такую информацию обычно никогда не хранят в исходном коде. Наиболее очевидная альтернатива: задавать их как параметры командной строки при запуске приложения. Для получения этих параметров в программе используется статический метод класса System — getProperty().

```
package edu.sfedu.mmcs.it.spring;

import  javax.sql.DataSource;
import  org.apache.commons.dbcp.BasicDataSource;
// BasicDataSource implements DataSource
import  javax.swing.JFrame;
```

```
// JFrame базовый класс для графического окна приложения

/// при создании объекта этого класса на экране
/// появится основное окно нашего приложения
public class TennisApplication extends JFrame {
  private TournamentMatchManager tournamentMatchManager;

  // поля, обозначающие элементы управления
  // на форме приложения: кнопки, поля ввода –
  // пропущены для краткости

  // обратите внимание на использование интерфейсов,
  // а не конкретных классов там, где это возможно
  public SwingApplication(TournamentMatchManager
                          tournamentMatchManager) {
    this.tournamentMatchManager = tournamentMatchManager;
    /* некоторый код инициализации пропущен для краткости */
  }

  // точка входа в программу
  public static void main(String[] args) throws Exception {
  // Создание и настройка DataSource
  // с помощью параметров командной строки
    DataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName(
          System.getProperty("jdbc.driverClassName"));
    dataSource.setUrl(System.getProperty("jdbc.url"));
    dataSource.setUsername(
                  System.getProperty("jdbc.username"));
    dataSource.setPassword(
                  System.getProperty("jdbc.password"));

    JdbcMatchDao matchDao = new JdbcMatchDao();
```

```
    matchDao.setDataSource(dataSource);
    TournamentMatchManager tournamentMatchManager =
       new DefaultTournamentMatchManager();
    tournamentMatchManager.setMatchDao(matchDao);
    new SwingApplication(tournamentMatchManager);
  }
}
```

Запуск этой программы должен выглядеть примерно так:

```
~$ java edu.sfedu.mmcs.it.spring.TennisApplication
-Djdbc.driverClassName=org.hsqldb.jdbcDriver
-Djdbc.url=jdbc:hsqldb:hsql://localhost/ -Djdbc.username=sa
-Djdbc.password=pass
```

Фрагмент метода main() после пустой строки это самая важная часть примера: все, что делалось до этой поры, было предназначено, чтобы увидеть его. Это код, «склеивающий» разные части нашего приложения. По мере разрастания приложения, его доля будет становиться все больше и одновременно конфигурировать приложение будет все сложней. Причем в случае, если нужно иметь несколько конфигураций, то менять их исправляя такого рода код чрезвычайно неудобное и подверженное ошибкам дело. Кроме того, видно, что часть конфигурации приложения проводится через параметры командной строки, часть в исходном коде — эта непоследовательность тоже затрудняет управление приложением.

Что предлагает вместо этого Spring? Выносить конфигурацию (управление зависимостями) в отдельный текстовый (XML) файл. В программе больше нет склеивающего кода: контейнер Spring считывает конфигурационный файл и создает в соответствии с ним необходимые объекты. Предположим, имя конфигурационного файла передается в командной строке, точка входа вынесена в отдельный класс (метода main() больше нет в

TennisApplication):

```
package edu.sfedu.mmcs.it.spring;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

/// Загрузчик Spring
public class SpringBootstrap {
  public static void main(String[] args) throws Exception {
    /* В качестве аргумента командной строки ожидается
       имя файла конфигурации Spring */
    if (args.length == 0) {
       throw new IllegalArgumentException(
          "Передайте имя конфигурационного" +
          " файла Spring в качестве аргумента" +
          " командной строки!");
    }

    /* Создание контейнера */
    BeanFactory factory =
        new XmlBeanFactory(new FileSystemResource(args[0]));
    /* Теперь объекты, описанные в конфиг. файле, созданы,
        на экране появилась форма нашего приложения */
  }
}
```

Что представляет из себя конфигурационный файл? Это XML, в котором перечислены так называемые «бины». Под бином в мире Java понимаются разные вещи, Spring-бины нестрого можно определить как компоненты приложения, нуждающиеся в управлении своими зависимостями.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean   id="tennisApplication"
          class="edu.sfedu.mmcs.it.spring.TennisApplication">
    <constructor-arg ref="tournamentMatchManager"/>
  </bean>

  <bean id="tournamentMatchManager"
     class=
     "edu.sfedu.mmcs.it.spring.DefaultTournamentMatchManager">
    <property name="myMatchDao" ref="matchDao"/>
  </bean>

  <bean   id="matchDao"
          class="edu.sfedu.mmcs.it.spring.JdbcMatchDao">
    <property name="myDataSource" ref="dataSource"/>
  </bean>

  <bean   id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource">
    <property  name="driverClassName"
               value="org.hsqldb.jdbcDriver"/>
    <property  name="url"
               value="jdbc:hsqldb:hsql://localhost"/>
    <property name="username" value="sa"/>
    <property name="password" value="pass"/>
  </bean>
</beans>
```

Элементы property указывают, что для инъекции зависимости необходимо использовать заботливо написанный заранее «сетер» (метод set*() в двух из трех наши классов), иначе можно использовать constructor-arg для указания того, что зависимость передается при вызове конструктора объекта.

Какие преимущества дает такой подход к организации приложения? Представим, что в приложении одну компоненту нужно заменить другой. Примерами такой ситуации могут быть тестирование компоненты, когда все или часть взаимодействующих с ней классов являются не полноценными классами, а «заглушками», выдающими заранее известную информацию (мы можем тестировать графический интерфейс, не поднимая базу данных: для этого достаточно передать объект-заглушку, реализующий TournamentMatchManager, и выдающий всегда один фиксированный ответ); это могут быть другие реализации DataSource (скажем, с поддержкой пула соединений с базой данных: такие решения зачастую более эффективными при больших нагрузках), другие реализации DAO (использующие для доступа к БД не JDBC, а некоторые более удобные автоматизированные средства, позволяющие вообще не писать SQL). Для каждого такого варианта пришлось бы изменять программный код, перекомпилировать его. Практика показывает, что такие языки общего назначения как Java не так удобны для создания склеивающего кода, который позволял бы манипулировать конфигурацией приложения. Раньше для таких целей часто использовались скриптовые языки (которые тоже подчас называли «склеивающими»), сейчас широко используются IoC-контейнеры, примером которых и является ядро Spring Frmework.