ГЛАВА 15

Планирование задач в Spring

Планирование задач является общей функциональной возможностью в корпоративных приложениях. Планирование задач в основном состоит из трех частей: задачи (представляющей собой порцию бизнес-логики, которая должна запускаться в определенное время или на регулярной основе), триггера (указывающего условие, при удовлетворении которого задача должна выполняться) и планировщика (который запускает задачу на основе информации, полученной от триггера).

В частности, в этой главе будут рассматриваться следующие темы.

- *Планирование задач в Spring*. Мы обсудим поддержку планирования задач в Spring, уделяя особое внимание абстракции `TaskScheduler`, которая появилась в версии Spring 3. Мы также опишем сценарии планирования, такие как планирование с фиксированным интервалом и с помощью выражения `cron`.
- *Асинхронное выполнение задач*. Мы покажем, как использовать новую аннотацию `@Async` в Spring 3 для выполнения задач асинхронным образом.

Создание проекта для примеров в STS

Начнем с создания проекта для примеров, рассматриваемых в этой главе. Создайте новый шаблонный проект Spring в STS, выбрав вариант Simple Spring JPA Utility Project (Простой служебный проект Spring JPA). Причина выбора этого шаблона проекта заключается в том, что мы намерены разработать пример задания, которое будет обновлять данные в СУРБД серверной части.

После создания в проект нужно добавить обязательные зависимости, описанные в табл. 15.1. Удостоверьтесь, что используется Spring 3.1 с JDK 6.

Таблица 15.1. Зависимости Maven для планирования задач

Идентификатор группы	Идентификатор артефакта	Версия	Описание
`org.springframework.data`	`spring-data-jpa`	1.0.1.RELEASE	Библиотека Spring Data JPA
`joda-time`	`joda-time`	2.0	API-интерфейс для работы с датой и временем под названием Joda-Time (`http://joda-time.sourceforge.net/`), который используется в Spring Data JPA. В этой главе мы будем применять его в объектах предметной области

Окончание табл. 15.1

Идентификатор группы	Идентификатор артефакта	Версия	Описание
joda-time	joda-time-hibernate	1.3	Библиотека Joda-Time для интеграции с Hibernate, обеспечивающая сохранение типов данных даты и времени
com.google.guava	guava	10.0.1	Полезные вспомогательные классы
org.slf4j	slf4j-log4j12	1.6.1	Библиотека ведения журналов (www.slf4j.org), которая будет использоваться в примерах этой главы. Данная библиотека соединяет ведение журнала SLF4J с библиотекой log4j

Реализация планирования задач в Spring

Корпоративные приложения часто нуждаются в планировании задач. Во многих приложениях разнообразные задачи (вроде отправки заказчикам уведомлений по электронной почте, запуска заданий в конце дня, выполнение обслуживания данных, обновление данных в пакетах и т.п.) должны планироваться к запуску на регулярной основе, либо через фиксированные интервалы (например, каждый час), либо по заданному расписанию (скажем, каждый вечер в 20:00 с понедельника по пятницу). Как упоминалось ранее, планирование задач состоит из трех частей: определение расписания (триггер), выполнение задачи (планировщик) и сама задача.

Для инициирования запуска задачи в Spring-приложении имеется множество разных способов. Один из них предусматривает запуск задания внешне из системы планирования, которая уже существует в среде развертывания приложений. Например, на многих предприятиях для планирования задач используются коммерческие системы, такие как Ctrl-M или CA Autosys. Если приложение выполняется на платформе Linux/Unix, можно пользоваться планировщиком crontab. Запуск заданий можно осуществлять отправкой запроса RESTful-WS к Spring-приложению, в результате чего контроллер Spring MVC инициирует выполнение задания.

Другой способ заключается в применении поддержки планирования задач, предлагаемой Spring. Для планирования задач в Spring доступны три варианта.

- *Поддержка JDK-объекта* `Timer`. При планировании задач в Spring поддерживается объект `Timer` из JDK.
- *Интеграция с Quartz*. Платформа Spring интегрирована с Quartz Scheduler (`www.quartz-scheduler.org`) — популярной библиотекой планирования с открытым кодом.
- *Абстракция* `TaskScheduler`, *встроенная в Spring*. В версии Spring 3 появилась абстракция `TaskScheduler`, которая предлагает простой способ планирования задач и поддерживает наиболее типичные требования.

В этом разделе мы сосредоточим внимание на использовании абстракции `TaskScheduler` для планирования задач.

Введение в абстракцию `TaskScheduler`

В Spring-абстракции `TaskScheduler` имеются три главных участника.

- *Интерфейс `Trigger`*. Интерфейс `org.springframework.scheduling.Trigger` предоставляет поддержку для определения механизма запуска. В Spring доступны две реализации `Trigger`. Класс `CronTrigger` поддерживает запуск на основе выражения `cron`, а класс `PeriodicTrigger` — запуск на основе начальной задержки и затем фиксированного интервала.
- *Задача*. Задача — это порция бизнес-логики, запуск которой необходимо запланировать. В Spring задача может быть указана как метод внутри любого бина Spring.
- *Интерфейс `TaskScheduler`*. Интерфейс `org.springframework.scheduling.TaskScheduler` предоставляет поддержку для планирования задач. В Spring доступны три класса реализации интерфейса `TaskScheduler`. Класс `TimerManagerTaskScheduler` (из пакета `org.springframework.scheduling.commonj`) является оболочкой для интерфейса `commonj.timers.TimerManager` из CommonJ, который обычно используется в коммерческих серверах приложений JEE, таких как WebSphere, WebLogic и т.д. Классы `ConcurrentTaskScheduler` и `ThreadPoolTaskScheduler` (оба из пакета `org.springframework.scheduling.concurrent`) представляют собой оболочки для класса `java.util.concurrent.ScheduledThreadPoolExecutor`. Оба класса поддерживают запуск задач из разделяемого пула потоков.

На рис. 15.1 показаны отношения между интерфейсом `Trigger`, интерфейсом `TaskScheduler` и задачей (которая реализует интерфейс `java.lang.Runnable`).

Планировать задачи с применением абстракции `TaskScheduler` из Spring можно двумя способами. Один из них предусматривает использование пространства имен `task` в XML-конфигурации Spring, а другой — аннотаций. Давайте рассмотрим оба эти способа.

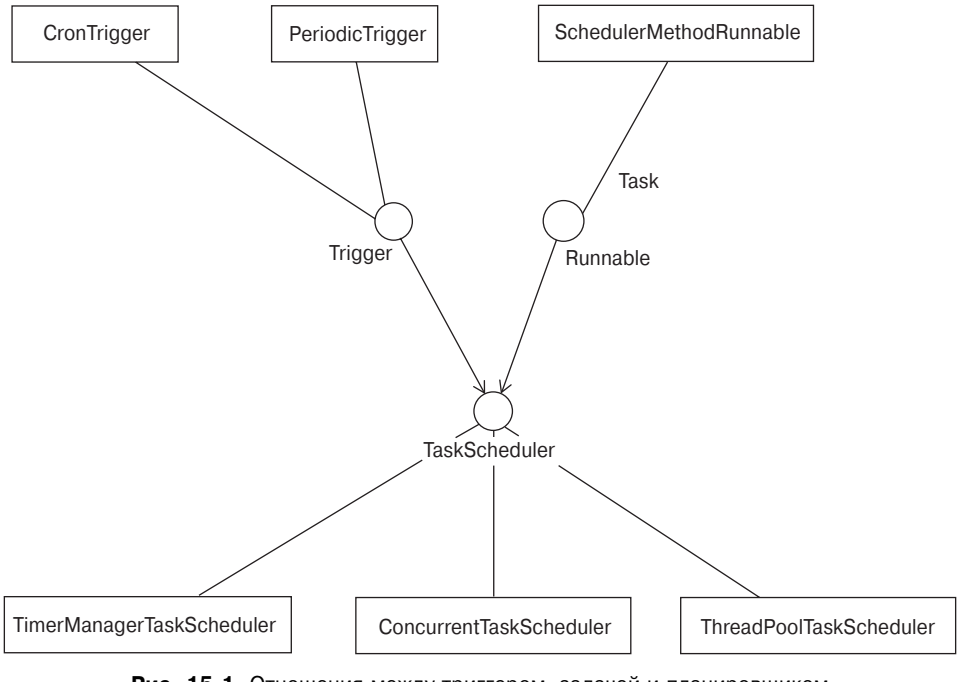

Рис. 15.1. Отношения между триггером, задачей и планировщиком

Пример задачи

Для демонстрации планирования задач в Spring реализуем сначала простое задание, а именно — приложение, обслуживающее базу данных с информацией об автомобилях. В листинге 15.1 показан класс `Car`, который реализован как сущностный класс JPA.

Листинг 15.1. Класс `Car`

```
package com.apress.prospring3.ch15.domain;

import static javax.persistence.GenerationType.IDENTITY;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;

@Entity
@Table(name="car")
public class Car {

    private Long id;

    private String licensePlate;

    private String manufacturer;

    private DateTime manufactureDate;

    private int age;

    private int version;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return id;
    }

    @Column(name="LICENSE_PLATE")
    public String getLicensePlate() {
        return licensePlate;
    }

    @Column(name="MANUFACTURER")
    public String getManufacturer() {
        return manufacturer;
    }

    @Column(name="MANUFACTURE_DATE")
    @Type(type="org.joda.time.contrib.hibernate.PersistentDateTime")
    public DateTime getManufactureDate() {
        return manufactureDate;
    }

    @Column(name="AGE")
    public int getAge() {
        return age;
    }
```

```
    @Version
    public int getVersion() {
        return version;
    }

    // Методы установки не показаны.

    public String toString() {
        return "License: " + licensePlate + " - Manufacturer: " + manufacturer
            + " - Manufacture Date: " + manufactureDate + " - Age: " + age;
    }
}
```

В листингах 15.2 и 15.3 приведен код сценариев создания таблицы (`schema.sql`) и заполнения ее тестовыми данными (`test-data.sql`) для сущностного класса `Car`.

Листинг 15.2. Сценарий для создания таблицы

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CAR (
     ID INT NOT NULL AUTO_INCREMENT
    , LICENSE_PLATE VARCHAR(20) NOT NULL
    , MANUFACTURER VARCHAR(20) NOT NULL
    , MANUFACTURE_DATE DATE NOT NULL
    , AGE INT NOT NULL DEFAULT 0
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CAR_1 (LICENSE_PLATE)
    , PRIMARY KEY (ID)
);
```

Листинг 15.3. Сценарий для заполнения таблицы тестовыми данными

```
insert into car (license_plate, manufacturer, manufacture_date)
  values ('LICENSE-1001', 'Ford', '1980-07-30');
insert into car (license_plate, manufacturer, manufacture_date)
  values ('LICENSE-1002', 'Toyota', '1992-12-30');
insert into car (license_plate, manufacturer, manufacture_date)
  values ('LICENSE-1003', 'BMW', '2003-1-6');
```

А теперь определим уровень обслуживания для сущности `Car`. Мы будем использовать проект Spring Data JPA и его поддержку абстракции репозитория. В листинге 15.4 представлен интерфейс `CarRepository`.

Листинг 15.4. Интерфейс `CarRepository`

```
package com.apress.prospring3.ch15.repository;

import org.springframework.data.repository.CrudRepository;

import com.apress.prospring3.ch15.domain.Car;

public interface CarRepository extends CrudRepository<Car, Long> {
}
```

Здесь нет ничего особенного; мы просто реализовали интерфейс `CrudRepository <Car,Long>`. В листингах 15.5 и 15.6 показан интерфейс `CarService` его класс реализации `CarServiceImpl`.

Листинг 15.5. Интерфейс `CarService`

```
package com.apress.prospring3.ch15.service;

import java.util.List;
import com.apress.prospring3.ch15.domain.Car;

public interface CarService {
    public List<Car> findAll();
    public Car save(Car car);
    public void updateCarAgeJob();
}
```

Листинг 15.6. Класс `CarServiceImpl`

```
package com.apress.prospring3.ch15.service.jpa;

// Операторы импорта опущены.
@Service("carService")
@Repository
@Transactional
public class CarServiceImpl implements CarService {
    final Logger logger = LoggerFactory.getLogger(CarServiceImpl.class);

    @Autowired
    CarRepository carRepository;

    @Transactional(readOnly=true)
    public List<Car> findAll() {
        return Lists.newArrayList(carRepository.findAll());
    }

    public Car save(Car car) {
        return carRepository.save(car);
    }

    public void updateCarAgeJob() {
        // Обновить возраст автомобиля.
        List<Car> cars = findAll();
        DateTime currentDate = DateTime.now();
        int age;
        logger.info("");
        logger.info("Car age update job started");
        for (Car car: cars) {
            age = new Period(car.getManufactureDate(), currentDate,
                    PeriodType.years()).getYears();
            car.setAge(age);
            save(car);
            logger.info("Car age update--- " + car);
        }
        logger.info("Car age update job completed successfully");
        logger.info("");
    }
}
```

В коде определены два метода; один из них извлекает информацию обо всех автомобилях, а другой сохраняет обновленный объект `Car`. Третий метод, `updateCarAgeJob()`, представляет собой задание, которое должно запускаться на регулярной основе и обновлять возраст автомобиля, имея дату его изготовления и текущую дату.

В листинге 15.7 представлена конфигурация Spring для поддержки рассматриваемого примера приложения с автомобилями (`car-job-app-context.xml`).

Листинг 15.7. Файл `car-job-app-context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:repository="http://www.springframework.org/schema/data/repository"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd
      http://www.springframework.org/schema/data/repository
      http://www.springframework.org/schema/data/repository/
spring-repository-1.0.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:schema.sql"/>
      <jdbc:script location="classpath:test-data.sql"/>
   </jdbc:embedded-database>
   <bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan"
         value="com.apress.prospring3.ch15.domain"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>
```

```
    <context:annotation-config/>
    <jpa:repositories base-package="com.apress.prospring3.ch15.repository"
        entity-manager-factory-ref="emf"
        transaction-manager-ref="transactionManager"/>
    <context:component-scan
        base-package="com.apress.prospring3.ch15.service.jpa" />
</beans>
```

Эта конфигурация должна выглядеть знакомой. А теперь займемся планированием задания по обновлению возраста автомобилей в Spring.

Планирование задач с использованием пространства имен `task`

Подобно другим пространствам имен в Spring, пространство имен `task` предоставляет упрощенную конфигурацию для планируемых задач с применением абстракции `TaskScheduler`, доступной в Spring.

Пространство имен `task` для планирования задач используется очень просто. В листинге 15.8 приведено содержимое конфигурационного файла `task-namespace-app-context.xml`.

Листинг 15.8. Конфигурация Spring, использующая пространство имен `task`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
    xmlns:task="http://www.springframework.org/schema/task"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-3.1.xsd">
    <import resource="car-job-app-context.xml"/>
    <task:scheduler id="myScheduler" pool-size="10"/>
    <task:scheduled-tasks scheduler="myScheduler">
        <task:scheduled ref="carService" method="updateCarAgeJob"
            fixed-delay="10000"/>
    </task:scheduled-tasks>
</beans>
```

Как показано в листинге 15.8, был импортирован контекст для приложения работы с автомобилями. Натолкнувшись на дескриптор `<task:scheduler>`, Spring создаст экземпляр класса `ThreadPoolTaskScheduler`, при этом атрибут `pool-size` задает размер пула потоков, который планировщик может использовать. Внутри дескриптора `<task:scheduled-tasks>` модет быть запланирована одна или большее число задач. В дескрипторе `<task:scheduled>` задача может ссылаться на бин Spring (в данном случае это бин `carService`) и специфический метод этого бина (в рассматриваемом примере это метод `updateCarAgeJob()`). Атрибут `fixed-delay` указывает Spring на необходимость создания `PeriodicTrigger` как реализации `Trigger` для `TaskScheduler`.

В листинге 15.9 приведен код программы для тестирования планирования задач.

Листинг 15.9. Тестирование планирования задач в Spring

```
package com.apress.prospring3.ch15.schedule;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ScheduleTaskSample {

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:task-namespace-app-context.xml");
        ctx.refresh();

        while (true) {
        }
    }
}
```

Класс `ScheduleTaskSample` довольно прост; он лишь загружает `ApplicationContext` и организует бесконечный цикл. Если приложение развернуто в среде сервера приложений, планировщик продолжит выполняться.

Запуск этой программы даст следующий вывод, получаемый каждые десять секунд:

```
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] - <>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update job started>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update--- License: LICENSE-1001 — Manufacturer: Ford -
Manufacture Date: 1980-07-30T00:00:00.000+08:00 - Age: 31>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update--- License: LICENSE-1002 — Manufacturer: Toyota -
Manufacture Date: 1992-12-30T00:00:00.000+08:00 - Age: 18>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update--- License: LICENSE-1003 — Manufacturer: BMW -
Manufacture Date: 2003-01-06T00:00:00.000+08:00 - Age: 8>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] -
<Car age update job completed successfully>
INFO [com.apress.prospring3.ch15.service.jpa.CarServiceImpl] - <>
```

В этом выводе видно, что атрибуты `age` автомобилей были успешно обновлены.

Кроме фиксированного интервала доступен более гибкий механизм планирования, предусматривающий использование выражения `cron`. Измените следующую строку в листинге 15.8:

```
<task:scheduled ref="carService" method="updateCarAgeJob"
fixed-delay="10000"/>
```

такой строкой:

```
<task:scheduled ref="carService" method="updateCarAgeJob" cron="0 * * * * *"/>
```

После такого изменения снова запустите тестовую программу с классом `ScheduleTaskSample` и вы увидите, что задание будет выполняться каждую минуту. На странице руководства по `CronTrigger` из библиотеки Quartz Scheduler (`www.quartz-scheduler.org/documentation/quartz-2.1.x/tutorials/crontrigger`) приведено детальное описание структуры и показаны примеры выражений `cron`.

Планирование задач с использованием аннотаций

Еще один способ для планирования задач с применением абстракции `TaskScheduler` из Spring предполагает использование аннотаций. Для этой цели в Spring предусмотрена аннотация `@Scheduled`.

Чтобы включить поддержку аннотаций для планирования задач, необходимо предоставить дескриптор `<task:annotation-driven>` в XML-конфигурации Spring. Содержимое конфигурационного файла `task-annotation-app-context.xml` показано в листинге 15.10.

Листинг 15.10. Конфигурация Spring для планирования на основе аннотаций

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
    xmlns:task="http://www.springframework.org/schema/task"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-3.1.xsd">
    <import resource="car-job-app-context.xml"/>

    <task:scheduler id="myScheduler" pool-size="10"/>
    <task:annotation-driven scheduler="myScheduler"/>
</beans>
```

В листинге 15.10 с помощью дескриптора `<task:annotation-driven>` включается поддержка планирования на основе аннотаций, с указанием в атрибуте `scheduler` ссылки на бин `myScheduler`.

Чтобы запланировать выполнение конкретного метода в бине Spring, просто аннотируйте метод посредством `@Scheduled` и передайте требования к планированию. В листинге 15.11 приведен фрагмент кода из переделанного класса `CarServiceImpl`.

Листинг 15.11. Переделанный класс `CarServiceImpl`

```
package com.apress.prospring3.ch15.service.jpa;

import org.springframework.scheduling.annotation.Scheduled;

// Остальной код не показан.

public class CarServiceImpl implements CarService {

    @Scheduled(fixedDelay=10000)
    //@Scheduled(fixedRate=10000)
    //@Scheduled(cron="0 * * * * *")
    public void updateCarAgeJob() {
        // Остальной код не показан.
    }
}
```

Тестовая программа представлена в листинге 15.12.

Листинг 15.12. Тестирование планирования на основе аннотаций

```
package com.apress.prospring3.ch15.schedule;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ScheduleTaskAnnotationSample {

    public static void main(String[] args) {

        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:task-annotation-app-context.xml");
        ctx.refresh();

        while (true) {
        }
    }
}
```

Выполнение этой программы даст тот же самый вывод, что и пример использования пространства имен `task`. Меняя атрибут в аннотации `@Scheduled`, можно опробовать разные механизмы запуска (`fixedDelay`, `fixedRate`, `cron`). Протестируйте это самостоятельно.

Асинхронное выполнение задач в Spring

В версии Spring 3.0 также поддерживается использование аннотаций для выполнения задачи асинхронным образом. Необходимо просто аннотировать метод с помощью `@Async`.

Чтобы увидеть это в действии, рассмотрим небольшой пример. В листингах 15.13 и Listing 15.14 показан интерфейс `AsyncService` и класс реализации `AsyncServiceImpl`.

Листинг 15.13. Интерфейс `AsyncService`

```
package com.apress.prospring3.ch15.service;
import java.util.concurrent.Future;
public interface AsyncService {
    public void asyncTask();
    public Future<String> asyncWithReturn(String name);
}
```

Листинг 15.14. Класс `AsyncServiceImpl`

```
package com.apress.prospring3.ch15.service.async;

import java.util.concurrent.Future;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Service;

import com.apress.prospring3.ch15.service.AsyncService;

@Service("asyncService")
public class AsyncServiceImpl implements AsyncService {

    final Logger logger = LoggerFactory.getLogger(AsyncServiceImpl.class);
```

```
    @Async
    public void asyncTask() {
        logger.info("Start execution of async. task");
        try {
            Thread.sleep(10000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        logger.info("Complete execution of async. task");
    }
    @Async
    public Future<String> asyncWithReturn(String name) {
        logger.info("Start execution of async. task with return");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        logger.info("Complete execution of async. task with return");
        return new AsyncResult<String>("Hello: " + name);
    }
}
```

В интерфейсе `AsyncService` определены два метода. Метод `asyncTask()` — это простая задача, которая записывает информацию в журнал.

Метод `asyncWithReturn()` принимает аргумент типа `String` и возвращает экземпляр интерфейса `java.util.concurrent.Future<V>`. Обратите внимание на код, выделенный полужирным; после завершения `asyncWithReturn()` результат хранится в экземпляре класса `org.springframework.scheduling.annotation.AsyncResult<V>`, который реализует интерфейс `Future<V>` и может использоваться для извлечения результата выполнения в более позднее время.

В листинге 15.15 показано содержимое файла конфигурации Spring (`async-app-context.xml`).

Листинг 15.15. Конфигурация Spring для асинхронной задачи

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-3.1.xsd">
    <context:annotation-config/>
    <context:component-scan
        base-package="com.apress.prospring3.ch15.service.async"/>
    <task:annotation-driven />
</beans>
```

Дескриптор `<task:annotation-driven />` в листинге 15.15 необходим для поддержки аннотации `@Async`. Тестовая программа представлена в листинге 15.16.

Листинг 15.16. Тестирование асинхронной задачи

```
package com.apress.prospring3.ch15.schedule;

import java.util.concurrent.Future;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring3.ch15.service.AsyncService;

public class AsyncTaskSample {

   public static void main(String[] args) {

      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:async-app-context.xml");
      ctx.refresh();

      AsyncService asyncService = ctx.getBean(
         "asyncService", AsyncService.class);

      for (int i = 0; i < 5; i++)
         asyncService.asyncTask();

      Future<String> result1 = asyncService.asyncWithReturn("Clarence");
      Future<String> result2 = asyncService.asyncWithReturn("John");
      Future<String> result3 = asyncService.asyncWithReturn("Robert");

      try {
         Thread.sleep(6000);

         System.out.println("Result1: " + result1.get());
         System.out.println("Result2: " + result2.get());
         System.out.println("Result3: " + result3.get());
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

В листинге 15.16 мы вызываем метод `asyncTask()` пять раз и метод `asyncWithReturn()` три раза с разными аргументами, а затем извлекаем результат после паузы в шесть секунд.

Запуск этой программы дает следующий вывод:

```
2011-11-23 17:46:16,276 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task with return>
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task with return>
```

```
2011-11-23 17:46:16,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Start execution of async. task with return>
2011-11-23 17:46:21,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task with return>
2011-11-23 17:46:21,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task with return>
2011-11-23 17:46:21,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task with return>
Result1: Hello: Clarence
Result2: Hello: John
Result3: Hello: Robert
2011-11-23 17:46:26,277 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
2011-11-23 17:46:26,278 INFO [com.apress.prospring3.ch15.service.async.
AsyncServiceImpl] — <Complete execution of async. task>
```

В этом выводе видно, что все вызовы начались в одно и то же время. Три вызова со значениями возврата завершились первыми и отобразились в консольном выводе. Наконец, были завершены пять вызовов метода `asyncTask()`.

Планирование задач в примере приложения

В приложении SpringBlog мы будем хранить записи хронологии для записей блога и связанных с ними комментариев в целях аудита. Тем не менее, для экономии пространства в базе данных мы также решили хранить только записи хронологии для 30 дней. Чтобы удовлетворить этому требованию, мы воспользуемся поддержкой абстракции `TaskScheduler` в Spring 3 и реализуем запланированное задание для удаления записей аудита, которые старее 30 дней. Это задание будет запускаться ежесуточно в полночь.

Для планирования задач мы будем применять стиль аннотаций. Первым делом мы определим интерфейс для задания по обслуживанию, который показан в листинге 15.17.

Листинг 15.17. Интерфейс `HousekeepingService`

```
package com.apress.prospring3.springblog.service;

public interface HousekeepingService {

    /**
     * Запланированное задание для удаления записей аудита.
     */
    public void auditPurgeJob();

}
```

В интерфейсе `HousekeepingService` определен метод `auditPurgeJob()` для задачи обслуживания записей аудита. В листинге 15.18 представлен класс реализации с примененной аннотацией `@Scheduled`.

Листинг 15.18. Класс `HousekeepingServiceImpl`

```
package com.apress.prospring3.springblog.service.jpa;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.apress.prospring3.springblog.service.HousekeepingService;

@Service("housekeepingService")
@Repository
@Transactional
public class HousekeepingServiceImpl implements HousekeepingService {

    @Value("${audit.record.history.days}")
    private int auditHistoryDays;

    @Scheduled(cron="0 0 0 * * ?")
    public void auditPurgeJob() {
        // Логика удаления записей аудита.
    }
}
```

Как показано в листинге 15.18, к методу `auditPurgeJob()` была применена аннотация `@Scheduled` с выражением `cron`. Это выражение `cron` означает, что задание должно запускаться ежесуточно в полночь. Для простоты сопровождения количество дней, в течение которых будут храниться записи аудита, вынесено во внешний файл свойств. Чтобы включить планирование задач в стиле аннотаций, мы определим дескриптор `<task:annotation-driven>` в корневом `WebApplicationContext`.

За дополнительными деталями обращайтесь в главу 21.

Резюме

В этой главе была рассмотрена поддержка планирования задач в Spring. Мы сосредоточили внимание на встроенной в Spring абстракции `TaskScheduler` и продемонстрировали ее использование для удовлетворения потребностей планирования задач на примере с обновлением данных. Мы также показали, как в Spring 3 поддерживается аннотация для асинхронного выполнения задач.