Системы проектирования программного продукта

Тема 6

Spring JBDC

Чистый JDBC

- Достаточно старая технология
- Неудобная (много шаблонного кода)
- Пользоваться ей напрямую в больших проектах – ад
- SQL Exception – checked и одно на все ошибки
- Нужно открывать/закрывать
- Дурацкий PreparedStatement
- Нужно обходить результирующий набор данных “руками”

Spring JDBC

- Для частичного решения этих проблем придумали паттерн Executor
- Spring JDBC - реализация этого паттерна от Spring-а

Plain JDBC vs. Spring JDBC

```
// Plain JDBC
final Connection connection = dataSource.getConnection();
try {
    final Statement statement = connection.createStatement();
    try {
        final ResultSet resultSet = statement.executeQuery("select count(*) from orders");
        try {
            resultSet.next();
            final int c = resultSet.getInt(1);
        } finally {
            resultSet.close();
        }
    } finally {
        statement.close();
    }
} catch (SQLException e) {
    throw new ServiceException(e);
} finally {
    connection.close();
}

//Spring
final int c = new JdbcTemplate(ds).queryForInt("select count(*) from orders");
```

Spring JDBC

- javax.sql.DataSource – управляет подключениями
- JdbcTemplate – центральный класс для выполнения запросов
- RowMapper – маппит строчку БД на объект
- JdbcDaoSupport – Немного упрощает конфигурирование

DataSource

DataSource

- Часть спецификации JDBC
- Подключаться к БД можно не только через DataSource
- Но Spring подключается через DataSource
- DataSource позволяет абстрагироваться от connection-ов и пулов
- Объект DataSource требуется создать самостоятельно или получить откуда-нибудь (JNDI, от Spring Boot)
- Автоматом через стартер

DataSource

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl("jdbc:mysql://localhost:3306/db");
    ds.setUsername("root");
    ds.setPassword("root");
    return ds;
}
```

Базовые классы

JdbcTemplate

JdbcTemplate – это главный класс в org.springframework.jdbc.core:

- Выполняет SQL-запросы
- Итерирует по результатам
- Ловит JDBC исключения

Для работы ему необходимы:

- DataSource
- RowMapper
- Собственно, SQL-запрос

JdbcTemplate

- Это база, но лучше NamedParameterJdbcTemplate
- threadsafe
- Может быть сконфигурирован однажды и использоваться несколькими DAO, а можно создавать несколько на один DataSource
- DataSource требуется для создания JdbcTemplate
- Можно передавать DataSource в DAO а потом в JdbcTemplate, но проще тестировать, когда передаётся сразу JdbcTemplate
- JdbcOperations – интерфейс для JdbcTemplate – и лучше использовать его

JdbcTemplate

```
@Repository
public class PersonDaoJdbc implements PersonDao {
    private final JdbcOperations jdbc;
    public PersonDaoJdbc (JdbcOperations jdbc) {
        this.jdbc = jdbc;
    }
    @Override
    public int count () {
        return jdbc.queryForObject("select count(*) from persons" , Integer.class );
    }
}
```

JdbcTemplate: insert, update, delete

```
@Override
public void insert (Person person) {
    jdbc.update("insert into persons (`id`, `name`) values (?, ?)",
    person.getId(), person.getName());
}
@Override
public void update (Person person) {
    jdbc .update("update persons set `name` = ? where `id` = ?",
    person.getName(), person.getId() );
}
@Override
public void deleteById (long id) {
    jdbc.update("delete from persons where `id` = ?", id);
}
```

JdbcTemplate: другие SQL запросы

```
@Override
public void createPersonsTable () {
    jdbc.execute("create table persons (`id` bigint primary key, " +
            "`name` varchar(255))");
}
```

RowMapper

- Интерфейс из org.springframework.jdbc.core
- Он описывает маппинг строчек ResultSet в конкретные объекты
- Используется в методе query() JdbcTemplate-а или в результате вызова хранимой процедуры
- Обычно сохраняется в поле DAO, если он stateless

RowMapper

```
@Override
public Person getById(long id) {
    return jdbc.queryForObject("select * from persons where id = ?",
            new Object[]{id}, new PersonMapper()
    );
}

@Override
public List<Person> getAll() {
    return jdbc.query("select * from persons", new PersonMapper());
}
```

RowMapper

```
private static class PersonMapper implements RowMapper<Person>{
    @Override
    public Person mapRow (ResultSet resultSet, int i) throws SQLException {
        long id = resultSet.getLong("id");
        String name = resultSet.getString("name");
        return new Person(id, name);
    }
}
```

NamedParameterJdbcTemplate

- Дает возможность задавать параметры не по индексам, а по именам
- Дает доступ к обычному JdbcTemplate через метод
- NamedParameterOperations – интерфейс, используйте его (можно мокать)
- Использовать только NamedParameterJdbcTemplate, а не обычный JdbcTemplate!

NamedParameterJdbcTemplate

```
@Repository
public class PersonDaoJdbc implements PersonDao {
    private final NamedParameterJdbcOperations jdbc;

    public PersonDaoJdbc(NamedParameterJdbcOperations jdbc) {
        this.jdbc = jdbc;
    }

    @Override
    public Person getById(long id) {
        final Map<String, Object> params = new HashMap<>(1);
        params.put("id", id);
        return jdbc.queryForObject("select * from persons where id = :id",
                params, new PersonMapper()
        );
    }
}
```

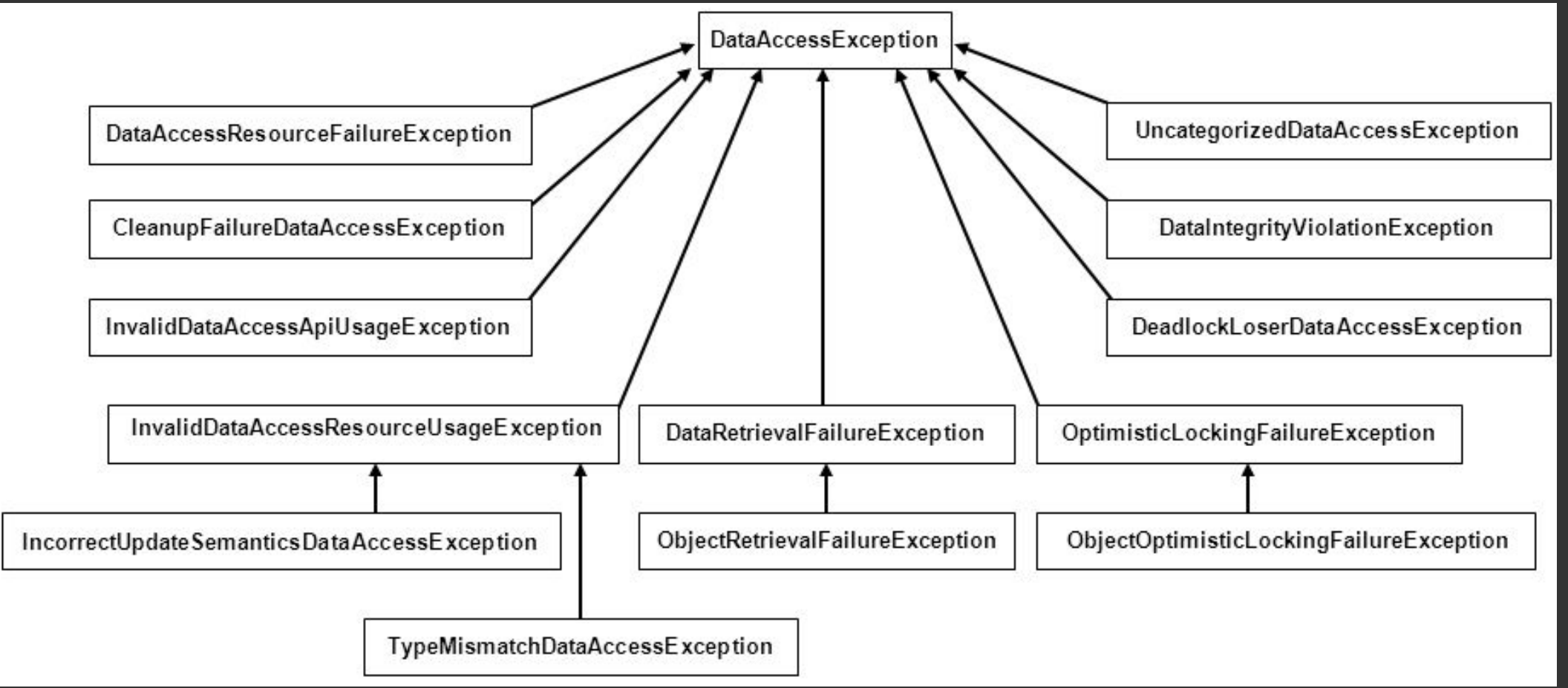
Обработка исключений

Обработка исключений

- Spring преобразует исключения, зависящие от платформы, такие как SQLException в иерархию собственных исключений, DataAccessException – корень этой иерархии;

- Это runtime-исключения (!)

Обработка исключений



Работа с ключом, генерируемым БД

Работа с ключом, генерируемым БД

- Чаще всего мы все же не вставляем id из приложения
- Обычно значение поля-идентификатора генерирует БД. Например с помощью автоинкрементного поля (типа BIGSERIAL)
- А знать id только, что вставленной записи иногда очень нужно
- В Spring-JDBC за получение сгенерированного базой id отвечает KeyHolder
- А точнее его реализация: GeneratedKeyHolder

Работа с ключом, генерируемым БД

```
@Override
public long insert (Person person) {
    MapSqlParameterSource params = new MapSqlParameterSource();
    params.addValue("name", person.getName());
    KeyHolder kh = new GeneratedKeyHolder();
    jdbc.update("insert into persons (`name`) values (‘:name’)", params, kh);
    return kh.getKey().longValue();
}
```

Стартер

Spring Boot Starter Jdbc

- Поднимает DataSource (настраивается в application.yml)
- Создаёт JdbcOperations
- И NamedParameterJdbcOperations
- Подключает транзакционность
- Автоматически выполняет файлы schema.sql, data.sql

DataSource

```
spring:
  datasource:
    url: jdbc:mysql://localhost/test
    username: dbuser
    password: dbpassword
    driver-class-name: com.mysql.jdbc.Driver
    initialization-mode: always
    schema: schema.sql
    data: data.sql
```

Тестирование

JDBCTest

- Поднимает часть контекста, ответственную за БД. В т.ч. выполняются schema.sql и data.sql

- По умолчанию создает вначале теста и откатывает в конце транзакцию

- Нужен @RunWith(SpringRunner.class) при использовании JUnit4

- Нужен @ExtendWith(SpringExtension.class) при использовании JUnit5

- У последних версий Spring Boot @ExtendWith(SpringExtension.class) есть внутри @JdbcTest, так что его писать не обязательно

JDBCTest

```
@DisplayName("Dao для работы с персонами ")
@JdbcTest
@Import(App.PersonDaoJdbc.class)
public class PersonDaoJdbcTest {
    @Autowired
    private App.PersonDaoJdbc dao;
    @DisplayName("возвращать заданного пёрсона по его Id ")
    void shouldReturnExpectedPersonById () {
        // Write test here
    }
}
```

JDBCTest

- @BeforeTransaction/@AfterTransaction – отмечают void методы теста, которые выполнятся до/после тестового метода если он завернут в транзакцию
- @Commit/@Rollback(value = false) – позволяют принудительно закоммитить тестовую транзакцию