باسمه تعالی

برنامه سازی وب

دانشکده مهندسی کامپیوتر دانشگاه صنعتی شریف

زمستان ۱۴۰۳

استاد:

استاد پورسلطانی

تحقیقی درباره Transactionها در Spring boot

سید حسین سید مهدی جاسبی ۴۰۱۱۰۶۰۸۵

مجتبی فراتین ۴۰۱۱۰۶۳۰۶

احسان محترم ۴۰۱۱۰۶۴۵۸

فهرست عناوین

۱ بیان مسئله

در بحث توسعه نرم افزار، یکی از چالش‌های اصلی مدیریت تراکنش‌ها یا Transactions در ارتباط با منابع مختلف مانند پایگاه داده‌ها، صف‌ها و دیگر منابع تراکنشی است. اگر بخواهیم در یک مثال یکی از این چالش‌ها را توضیح دهیم می‌توان به ثبت‌نام یک دانشجو در سامانه آموزشی اشاره کرد. در فرآیند ثبت‌نام دانشجو ابتدا باید اطمینان حاصل شود که اگر ظرفیت درس تکمیل شد، تمامی عملیات‌های قبلی (مانند ثبت اطلاعات در پایگاه داده) برگشت داده شوند. (Rollback) همچنین، در صورت بروز خطا مانند مشکل در ارسال ایمیل یا ... باید پایگاه داده به حالت اولیه خود بازگردد. علاوه بر این، در محیط های چندکاربره مانند سیستم انتخاب واحد دانشگاه، چالش هم‌زمانی وجود دارد؛ یعنی اگر دو دانشجو هم‌زمان برای یک درس ثبت‌نام کنند، باید از بروز شرایط رقابتی و ثبت‌نام غیرمجاز جلوگیری شود. استفاده از یک مدل جامع برای مدیریت تراکنش‌ها می‌تواند مشکلاتی مانند پیچیدگی‌های زیاد و کدهای تکراری را برای برنامه‌نویس بسیار کاهش دهد و دیگر او را از فکر کردن به این موضوع آزاد کند تا بتواند به بخش‌های مهم دیگر مسئله و پروژه‌اش برسد. به همین دلیل، فریم‌ورک Spring یک مدل جامع را تدارک دیده است تا شرایط ساده‌تری برای برنامه‌نویسان فراهم آورد که از آن طریق تراکنش‌ها به صورت بهینه مدیریت شوند.

۲ مفاهیم کلی در تراکنش‌ها

تراکنش‌های پایگاه داده باید از اصول ACID پیروی کنند:

۱. Atomicity (اتمی بودن): تمام عملیات تراکنش یا باید به طور کامل انجام شود یا هیچ‌کدام انجام نشود.
۲. Consistency (یک‌پارچگی): داده‌ها قبل و بعد از تراکنش باید در یک حالت سازگار باشند.
۳. Isolation (ایزوله بودن): تراکنش‌ها نباید بر یکدیگر تأثیر منفی بگذارند.
۴. Durability (دوام): پس از تأیید نهایی، تغییرات تراکنش باید به صورت پایدار ذخیره شوند.[2]

۳ معرفی ابزار

همانطور که گفته شد Framework Spring این امکان را به توسعه‌دهندگان می‌دهد تا با استفاده ازSpring Boot ، کد یک بار نوشته شود و از استراتژی های مختلف مدیریت تراکنش در محیط‌های مختلف بهره‌برداری کرد. ابتدا دو تعریف زیر را برای بیان کارکرد فریم‌ورک Spring نیاز داریم:

تراکنش‌های جهانی: تراکنش‌های جهانی به شما اجازه می‌دهند با چندین منبع تراکنشی (مثل پایگاه داده‌ها و صف‌های پیام) کار کنید. این تراکنش‌ها از طریق JTA مدیریت می‌شوند که پیچیده است و نیاز به استفاده از JNDI دارد. این نوع تراکنش‌ها به طور معمول در سرورهای برنامه اجرا می‌شوند و برای استفاده از آن‌ها باید از CMT EJB استفاده کنید. این روش به سرور برنامه وابسته است و نوشتن کد اضافی برای کنترل تراکنش‌ها لازم است.

تراکنش‌های محلی: تراکنش‌های محلی تنها با یک منبع خاص، مثل یک پایگاه داده، کار می‌کنند. این تراکنش‌ها ساده‌تر هستند، ولی نمی‌توانند در تراکنش‌های جهانی مشارکت کنند. سرور برنامه در مدیریت این تراکنش‌ها دخالت ندارد و نمی‌تواند صحت تراکنش را در منابع مختلف تضمین کند. این نوع تراکنش‌ها برای برنامه‌های کوچک‌تر مناسب‌تر هستند.

Spring boot می‌تواند به راحتی تراکنش‌ها را در هر نوع زیرساختی از جمله JDBC ، Hibernate و JTA مدیریت کند و به ما این امکان را می‌دهد که فقط با تغییر پیکربندی، نحوه مدیریت تراکنش‌ها را از تراکنش‌های محلی به تراکنش‌های جهانی یا برعکس تغییر دهیم، بدون اینکه نیاز به تغییر در کد برنامه‌نویسی باشد. در نهایت، Boot Spring به کمک Framework Spring، تراکنش‌ها را به سادگی از طریق الگوهای اعلامی و برنامه‌نویسی قابل مدیریت می‌کند و محیطی ساده و مقیاس‌پذیر برای برنامه‌های جاوا فراهم می‌کند.

Spring boot از دو روش برای مدیریت تراکنش‌ها پشتیبانی می‌کند:

۱. Declarative Transaction Management (مدیریت تراکنش اعلامی)
۲. Programmatic Transaction Management (مدیریت تراکنش برنامه‌نویسی)

رایج‌ترین روش مدیریت تراکنش در Spring استفاده از @Transactional است. این روش به شما امکان می‌دهد تا بدون نوشتن کد اضافی، مدیریت تراکنش را به Spring بسپارید.

```java
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
@Service
public class AccountService {
    @Transactional
    public void transferMoney(Long fromAccountId, Long toAccountId, Double amount) {
        // برداشت از حساب مبدا
        debit(fromAccountId, amount);
        // واریز به حساب مقصد
        credit(toAccountId, amount);
    }
    private void debit(Long accountId, Double amount) {
        // منطق برداشت پول از حساب
    }
    private void credit(Long accountId, Double amount) {
        // منطق واریز پول به حساب
    }
}
```

در این روش، اگر خطایی در هر یک از عملیات‌ها رخ دهد، کل تراکنش Rollback می‌شود.

در روش دوم از کلاس TransactionTemplate استفاده می‌شود که به ما کنترل بیشتری بر تراکنش می‌دهد.

```java
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;
@Service
public class AccountService {
    private final TransactionTemplate transactionTemplate;
    public AccountService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }
    public void transferMoney(Long fromAccountId, Long toAccountId, Double amount) {
        transactionTemplate.execute(status -> {
            debit(fromAccountId, amount);
            credit(toAccountId, amount);
            return null;
        });
    }
    private void debit(Long accountId, Double amount) {
        // منطق برداشت پول از حساب
    }
    private void credit(Long accountId, Double amount) {
        // منطق واریز پول به حساب
    }
}
```

Spring Boot به صورت خودکار TransactionManager را برای ما تنظیم می‌کند، اما در صورت نیاز می‌توانیم آن را به صورت دستی پیکربندی کنیم:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.orm.jpa.JpaTransactionManager;
import javax.persistence.EntityManagerFactory;
@Configuration
@EnableTransactionManagement
public class TransactionConfig {
    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}
```

۴ پیاده‌سازی مثال عملی

۴.۱ مثال عملی ساده

ابتدا به پیاده‌سازی یک مثال ساده می‌پردازیم. در Spring boot، برای پیکربندی تراکنش‌ها، ابتدا باید یک DataSource تعریف کنیم که به پایگاه داده متصل شود. این کار معمولاً در فایل‌های پیکربندی مانند application.properties یا application.yml انجام می‌شود، اما در مثال زیر[1]، از XML برای تعریف آن استفاده شده است. سپس، برای مدیریت تراکنشها، باید یک PlatformTransactionManager مانند DataSourceTransactionManager تعریف کنیم که به DataSource متصل شده و تراکنش‌ها را مدیریت می‌کند. این پیکربندی به ما این امکان را می‌دهد که تراکنش‌ها را به طور مؤثر و بدون نیاز به سرورهای پیچیده، مدیریت کنیم.

```java
@Autowired
private PlatformTransactionManager txManager;

public void someMethod() {
    // شروع یک تراکنش جدید
    TransactionStatus status = txManager.getTransaction(new DefaultTransactionDefinition());

    try {
        // کدهای مربوط به عملیات پایگاه داده
        // در اینجا فرض می‌کنیم که عملیاتی مانند درج داده‌ها انجام می‌دهیم

        // اگر هیچ مشکلی پیش نیاید، تراکنش را ثبا می‌کنیم
        txManager.commit(status);
    } catch (Exception e) {
        // اگر خطا پیش بیاید، تراکنش را برمیگردانیم
        txManager.rollback(status);
    }
}
```

با توجه به توضیحات قبلی، این مثال یک نوع از مدیریت تراکنش‌ها با استفاده از برنامه‌نویسی است.

۴.۲ مثال عملی پیچیده

در این مثال، دو حساب بانکی داریم و می‌خواهیم مقدار مشخصی پول از یک حساب به حساب دیگر انتقال دهیم. لازم است که بتوانیم در این برنامه حساب جدید بسازیم، به حساب مورد نظر واریز داشته باشیم و یا از آن برداشت کنیم و قابلیت انتقال وجه بین دو حساب را هم داشته باشیم.

توضیح کامل این برنامه و نتایج آن در فیلم الحاقی این تحقیق آمده است.

BankAccount:

```java
package com.example.demo.Model;
import jakarta.persistence.*;
import lombok.*;

16 usages
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class BankAccount {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    no usages
    private String owner;
    no usages
    private Double balance;

}
```

BankAccountRepository:

```java
package com.example.demo.Repository;
import com.example.demo.Model.BankAccount;
import org.springframework.data.jpa.repository.JpaRepository;

3 usages
public interface BankAccountRepository extends JpaRepository<BankAccount, Long> {
}
```

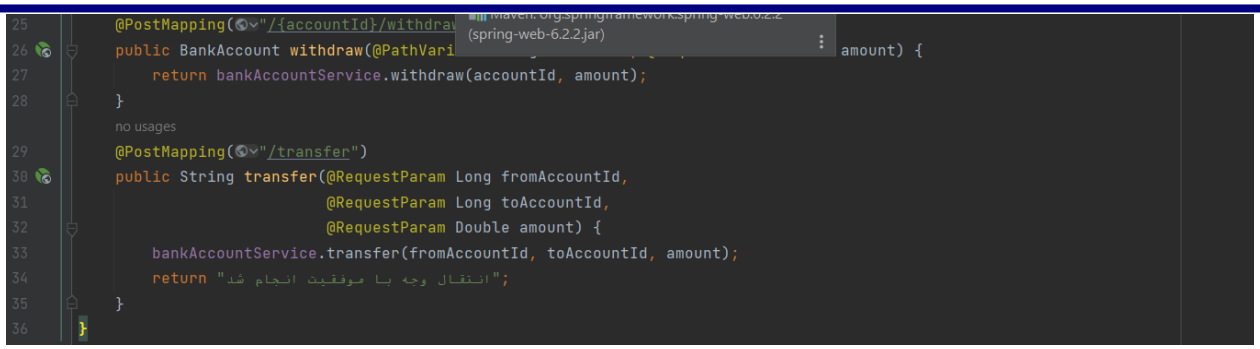
BankAccountService:

```java
package com.example.demo.Service;
import com.example.demo.Model.BankAccount;
import com.example.demo.Repository.BankAccountRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

3 usages
@Service
public class BankAccountService {
    5 usages
    private final BankAccountRepository bankAccountRepository;
    no usages
    public BankAccountService(BankAccountRepository bankAccountRepository) {
        this.bankAccountRepository = bankAccountRepository;
    }
    1 usage
    public BankAccount createAccount(String owner, Double initialBalance) {
        BankAccount account = BankAccount.builder()
                .owner(owner)
                .balance(initialBalance)
                .build();
        return bankAccountRepository.save(account);
    }
    3 usages
    public BankAccount getAccount(Long accountId) {
        return bankAccountRepository.findById(accountId)
                .orElseThrow(() -> new RuntimeException("حساب یافت نشد"));
    }
    2 usages
    @Transactional
```

```java
    @Transactional
    public BankAccount deposit(Long accountId, Double amount) {
        BankAccount account = getAccount(accountId);
        account.setBalance(account.getBalance() + amount);
        return bankAccountRepository.save(account);
    }
    2 usages
    @Transactional
    public BankAccount withdraw(Long accountId, Double amount) {
        BankAccount account = getAccount(accountId);
        if (account.getBalance() < amount) {
            throw new RuntimeException("موجودی کافی نیست");
        }
        account.setBalance(account.getBalance() - amount);
        return bankAccountRepository.save(account);
    }
    1 usage
    @Transactional
    public void transfer(Long fromAccountId, Long toAccountId, Double amount) {
        withdraw(fromAccountId, amount);
        deposit(toAccountId, amount);
    }
}
```

BankAccountController:

```java
package com.example.demo.Controller;
import com.example.demo.Model.BankAccount;
import org.springframework.web.bind.annotation.*;
import com.example.demo.Service.BankAccountService;

no usages
@RestController
@RequestMapping("/bank")
public class BankAccountController {
    6 usages
    private final BankAccountService bankAccountService;
    no usages
    public BankAccountController(BankAccountService bankAccountService) {
        this.bankAccountService = bankAccountService;
    }
    no usages
    @PostMapping("/create")
    public BankAccount createAccount(@RequestParam String owner, @RequestParam Double initialBalance) {
        return bankAccountService.createAccount(owner, initialBalance);
    }
    no usages
    @GetMapping("/{accountId}")
    public BankAccount getAccount(@PathVariable Long accountId) { return bankAccountService.getAccount(accountId); }
    no usages
    @PostMapping("/{accountId}/deposit")
    public BankAccount deposit(@PathVariable Long accountId, @RequestParam Double amount) {
        return bankAccountService.deposit(accountId, amount);
    }
    no usages
    @PostMapping("/{accountId}/withdraw")
```



۵ مراجع

[1] https://docs.spring.io/spring-framework/reference/data-access/transaction.html

[2] https://en.wikipedia.org/wiki/ACID