MyBatis、Spring和SpringMVC

2018年8月18日 9:51

★ MyBatis

(1)MyBatis的主要作用

a、不屏蔽SQL，意味着可以更为精确地定位SQL 语句，可以对其进行优化和改造(SQL语句的优化)，这有利于互联网系统性能的提高，符合互联网需要性能优化的特点。

b、提供强大、灵活的映射机制，方便Java 开发者使用。提供动态SQL 的功能，允许我们根据不同条件组装SQL，这个功能远比其他工具或者Java 编码的可读性和可维护性高得多，满足各种应用系统的同时也满足了需求经常变化的互联网应用的要求。

c、在MyBatis 中，提供了使用Mapper 的接口编程，只要一个接口和一个XML 就能创建映射器，进一步简化我们的工作，使得很多框架API(Application Program Interface,应用程序接口) 在MyBatis 中消失，开发者能更集中于业务逻辑。

(2)MyBatis的核心组件——(SqlSessionFactoryBuilder(构造器)、SqlSessionFactory、Sqlsession和SQL Mapper)

SqlSessionFactoryBuilder：它会根据配置或者代码来生成SqlSessionFactory，采用的是分步构建的Builder 模式(设计模式部分)。

SqlSessionFactory(工厂接口)：依靠它来生成SqlSession，使用的是工厂模式。

SqlSession: 一个既可以发送SQL执行返回结果，也可以获取Mapper 的接口。在现有的技术中， 一般我们会让其在业务逻辑代码中"消失"，而使用的是MyBatis 提供的SQLMapper 接口编程技术，它能提高代码的可读性和可维护性。

SQL Mapper(映射器): 是MyBatis 中最重要、最复杂的组件，它由一个Java 接口和XML文件（或注解）构成，需要给出对应的SQL和映射规则。它负责发送SQL 去执行，将SQL 查询到的结果映射为一个POJO(实体类)，或者将POJO 的数据插入到数据库中。

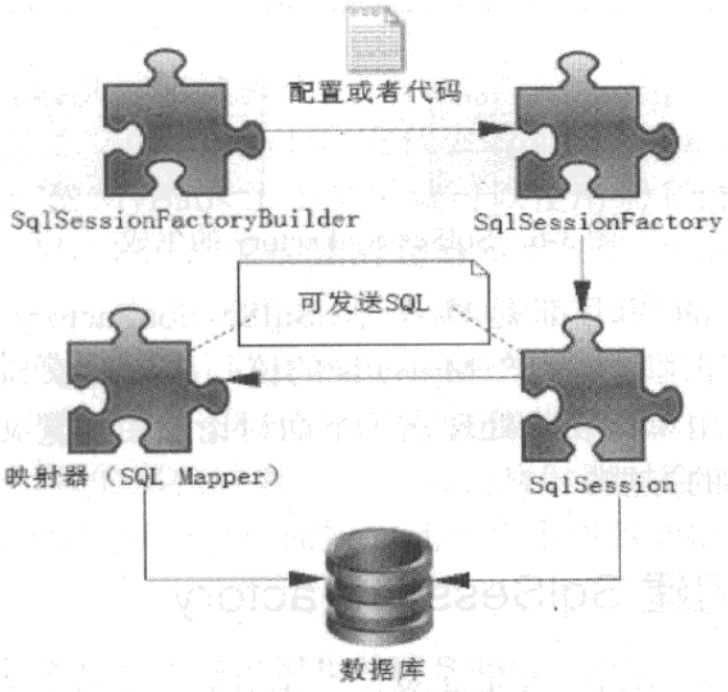

图 3-5 MyBatis 核心组件

★ 用XML的方式创建映射器：

```
<mapper namespace="com.learn.ssm.chapter3.mapper.RoleMapper">
    <select id="getRole" parameterType="long" resultType="role">
        select id, role_name as roleName, note from t_role where id = #{id}
    </select>
</mapper>
```

修改别名

有了这两个文件，就完成了一个映射器的定义。XML 文件还算比较简单，我们稍微讲解一下：

```
<mapper namespace="com.learn.ssm.chapter3.mapper.RoleMapper">
    <select id="getRole" parameterType="long" resultType="role">
        select id, role_name as roleName, note from t_role where id = #{id}
    </select>
</mapper>
```

修改别名

有了这两个文件，就完成了一个映射器的定义。XML 文件还算比较简单，我们稍微讲解一下：

- <mapper>元素中的属性 namespace 所对应的是一个接口的全限定名，于是 MyBatis 上下文就可以通过它找到对应的接口。
- <select>元素表明这是一条查询语句，而属性 id 标识了这条 SQL，属性 parameterType="long" 说明传递给 SQL 的是一个 long 型的参数，而 resultType="role" 表示返回的是一个 role 类型的返回值。而 role 是之前配置文件 mybatis-config.xml 配置的别名，指代的是 com.learn.ssm.chapter3.pojo.Role。
- 这条 SQL 中的#{id}表示传递进去的参数。

另一个是驼峰映射，其对书写要求高，需要与数据库中的列名一一对应

注意，我们并没有配置 SQL 执行后和 role 的对应关系，它是如何映射的呢？其实这里采用的是一种被称为自动映射的功能，MyBatis 在默认情况下提供自动映射，只要 SQL 返回的列名能和 POJO 对应起来即可。这里 SQL 返回的列名 id 和 note 是可以和之前定义的 POJO 的属性对应起来的，而表里的列 role_name 通过 SQL 别名的改写，使其成为 roleName，也是和 POJO 对应起来的，所以此时 MyBatis 就可以把 SQL 查询的结果通过自动映射的功能映射成为一个 POJO。

注意：

(1) 由于SqlSessionFactory 是一个对数据库的连接池，所以它占据着数据库的连接资源。如果创建多个SqlSessionFactory，那么就存在多个数据库连接池，这样不利于对数据库资源的控制，也会导致数据库连接资源被消耗光，出现系统密机等情况，所以尽量避免发生这样的情况。因此在一般的应用中我们往往希望SqlSessionFactory 作为一个单例，让它在应用中被共享。

(2) 如果说SqlSessionFactory 相当于数据库连接池，那么SqlSession 就相当于一个数据库连接（ Connection 对象），你可以在一个事务里面执行多条SQL，然后通过它的commit、rollback 等方法， 提交或者回滚事务。所以它应该存活在一个业务请求中，处理完整个请求后，应该关闭这条连接，让它归还给SqlSessionFactory， 否则数据库资源就很快被耗费精光，系统就会瘫痪，所以用try... catch .. . finally ...语句来保证其正确关闭。

Spring

(1) Spring的主要作用

a、对于POJO(实体类)的潜力开发，提供轻量级编程，可以通过配置（ XML 、注解等）来扩展POJO的功能，通过依赖注入的理念去扩展功能，建议通过接口编程，强调OOD(Object-Oriented Design，面向对象设计) 的开发模式理念，降低系统耦合度，提高系统可读性和可扩展性。

b、提供切面编程，尤其是把企业的核心应用——数据库应用，通过切面消除了以前复杂的问try... catch ... finally代码结构，使得开发人员能够把精力更加集中于业务开发而不是技术本身，也避免了try...catch ... finally 语句的滥用。

c、为了整合各个框架和技术的应用， Spring 提供了模板类，通过模板可以整合各个框架和技术，比如支持Hibernate 开发的Hibernate Template 、支持MyBatis 开发的SqlSessionTemplate 、支持Redis开发的RedisTemplate 等，这样就把各种企业用到的技术框架整合到Spring 中，提供了统一的模板，从而使得各种技术用起来更简单。

(2) 核心理念——IoC(控制反转)和AoP(面向切面)

IoC(控制反转)：由spring来负责控制对象的生命周期和对象间的关系的，而不是由传统程序代码直接操控。IOC在spring里通过依赖注入实现的，主键之间的依赖关系由容器在运行期决定，由容器动态的将某种关系注入到主键中。

例如：之前我们先要在A中创建connection对象，再由connection对象来与数据库建立联系。有了spring后，我们只需告诉spring A中需要connection，至于connection怎么构造，何时构造，A不需要知道，在系统运行时，spring就会自动制造一个connection，然后注入到A中，由此完成对各个对象间的控制。

AoP(面向切面)：是面向对象开发的一种 补充，它允许开发人员在不改变原来模型的基础上动态的修改模型以满足新的要求(开发人员可以在不改变业务逻辑的基础上可以动态的增加日志、安全或者异常处理的功能)。例如在项目中；我们的增删改查操作都需要权限验证，我们不希望权限验证的代码残留在增删改查的方法里面。我们可以通过AOP在程序运行的时候，动态的将我们的权限代码植入到我们增删改查的方法的前面，以完成权限的验证。

(3) AOP的实现方式：动态代理；动态代理分为jdk和cglib(面试常问)

jdk动态代理和cglib代理的区别

a、JDK动态代理是面向接口，在创建代理实现类时比CGLib要快，创建代理速度快。

b、CGLib动态代理是通过字节码底层继承要代理类来实现（如果被代理类被final关键字所修饰，那么会创建失败），在创建代理这一块没有JDK动态代理快，但是运行速度比JDK动态代理要快。

有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。由此可以看出，jdk动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。

动态代理的例子：

代码清单 2-7：定义接口

```
public interface HelloWorld {
    public void sayHelloWorld();
}
```

然后提供实现类 HelloWordImpl 来实现接口，如代码清单 2-8 所示。

代码清单 2-8：实现接口

```
public class HelloWorldImpl implements HelloWorld {
    @Override
    public void sayHelloWorld() {
        System.out.println("Hello World");
    }
}
```

这是最简单的 Java 接口和实现类的关系，此时可以开始动态代理了。按照我们之前的分析，先要建立起代理对象和真实服务对象的关系，然后实现代理逻辑，所以一共分为两个步骤。

在 JDK 动态代理中，要实现代理逻辑类必须去实现 java.lang.reflect.InvocationHandler 接口，它里面定义了一个 invoke 方法，并提供接口数组用于下挂代理对象，如代码清单 2-9 所示。

代码清单 2-9：动态代理绑定和代理逻辑实现

```
public class JdkProxyExample implements InvocationHandler {

    //真实对象
    private Object target = null;

    /**
     * 建立代理对象和真实对象的代理关系，并返回代理对象
     * @param target 真实对象
```

```
     * @return  代理对象
     */
    public Object bind(Object target) {
        this.target = target;
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), this);
    }

    /**
     * 代理方法逻辑
     * @param proxy 代理对象
     * @param method 当前调度方法
     * @param args 当前方法参数
     * @return 代理结果返回
     * @throws Throwable 异常
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("进入代理逻辑方法");
        System.out.println("在调度真实对象之前的服务");
        Object obj = method.invoke(target, args);//相当于调用 sayHelloWorld
方法
        System.out.println("在调度真实对象之后的服务");
        return obj;
    }
}
```

第 1 步，建立代理对象和真实对象的关系。这里是使用了 bind 方法去完成的，方法里面首先用类的属性 target 保存了真实对象，然后通过如下代码建立并生成代理对象。

```
Proxy.newProxyInstance(target.getClass()
.getClassLoader(), target.getClass().getInterfaces(), this);
```

其中 newProxyInstance 方法包含 3 个参数。

- 第 1 个是类加载器，我们采用了 target 本身的类加载器。
- 第 2 个是把生成的动态代理对象下挂在哪些接口下，这个写法就是放在 target 实现的接口下。HelloWorldImpl 对象的接口显然就是 HelloWorld，代理对象可以这样声明：`HelloWorld proxy = xxxx;`。
- 第 3 个是定义实现方法逻辑的代理类，this 表示当前对象，它必须实现 InvocationHandler 接口的 invoke 方法，它就是代理逻辑方法的现实方法。

第 2 步，实现代理逻辑方法。invoke 方法可以实现代理逻辑，invoke 方法的 3 个参数的含义如下所示。

- proxy，代理对象，就是 bind 方法生成的对象。
- method，当前调度的方法。
- args，调度方法的参数。

当我们使用了代理对象调度方法后，它就会进入到 invoke 方法里面。

代码清单 2-10：测试 JDK 动态代理

```
public void testJdkProxy() {
    JdkProxyExample jdk = new JdkProxyExample();
    //绑定关系，因为挂在接口 HelloWorld 下，所以声明代理对象 HelloWorld proxy
    HelloWorld proxy = (HelloWorld)jdk.bind(new HelloWorldImpl());
    //注意，此时 HelloWorld 对象已经是一个代理对象，它会进入代理的逻辑方法 invoke 里
proxy.sayHelloWorld();
}
```

首先通过 bind 方法绑定了代理关系，然后在代理对象调度 sayHelloWorld 方法时进入了代理的逻辑，测试结果如下：

```
进入代理逻辑方法
在调度真实对象之前的服务
Hello World
在调度真实对象之后的服务
```

此时，在调度打印 Hello World 之前和之后都可以加入相关的逻辑，甚至可以不调度 Hello World 的打印。

★ spring MVC

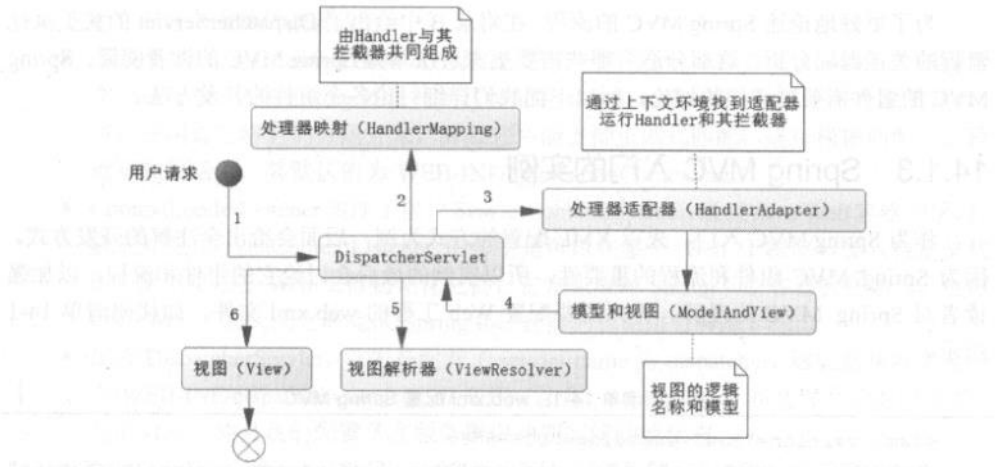

图 14-4　Spring MVC 的组件和流程图

图 14-4 中的阿拉伯数字给出了 Spring MVC 的服务流程及其各个组件运行的顺序，这是 Spring MVC 的核心。

首先，Spring MVC 框架是围绕着 DispatcherServlet 而工作的，所以这个类是其最为重要的类。从它的名字来看，它是一个 Servlet，那么根据 Java EE 基础的学习，我们知道它可以拦截 HTTP 发送过来的请求，在 Servlet 初始化（调用 init 方法）时，Spring MVC 会根据配置，获取配置信息，从而得到统一资源标识符（URI，Uniform Resource Identifier）和处理器（Handler）之间的映射关系（HandlerMapping），为了更加灵活和增强功能，Spring MVC 还会给处理器加入拦截器，所以还可以在处理器执行前后加入自己的代码，这样就构成了一个处理器的执行链（HandlerExecutionChain），并且根据上下文初始化视图解析器等内容，当处理器返回的时候就可以通过视图解析器定位视图，然后将数据模型渲染到视图中，用来响应用户的请求了。

当一个请求到来时，DispatcherServlet 首先通过请求和事先解析好的 HandlerMapping 配置，找到对应的处理器（Handler），这样就准备开始运行处理器和拦截器组成的执行链，而运行处理器需要有一个对应的环境，这样它就有了一个处理器的适配器（HandlerAdapter），通过这个适配器就能运行对应的处理器及其拦截器，这里的处理器包含了控制器的内容和其他增强的功能，在处理器返回模型和视图给 DispacherServlet 后，DispacherServlet 就会把对应的视图信息传递给视图解析器（ViewResolver）。注意，这一步取决于是否使用逻辑视图，如果是逻辑视图，那么视图解析器就会解析它，然后把模型渲染到视图中去，最后响应用户的请求；如果不是逻辑视图，则不会进行处理，而是直接通过视图渲染数据模型。

附件:

1 重叠构造器模式

我们先来看看程序员一向习惯使用的重叠构造器模式，在这种模式下，你提供第一个只有必要参数的构造器，第二个构造器有一个可选参数，第三个有两个可选参数，依此类推，最后一个构造器包含所有的可选参数。下面看看其编程实现：

```
/**
 * 使用重叠构造器模式
 */
public class Person {
    //必要参数
    private final int id;
    private final String name;
    //可选参数
    private final int age;
    private final String sex;
    private final String phone;
    private final String address;
    private final String desc;
public Person(int id, String name) {
        this(id, name, 0);
    }

public Person(int id, String name, int age) {
        this(id, name, age, "");
    }

public Person(int id, String name, int age, String sex) {
        this(id, name, age, sex, "");
    }

public Person(int id, String name, int age, String sex, String phone) {
        this(id, name, age, sex, phone, "");
    }

public Person(int id, String name, int age, String sex, String phone, String address) {
        this(id, name, age, sex, phone, address, "");
    }

public Person(int id, String name, int age, String sex, String phone, String address, String desc) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.sex = sex;
        this.phone = phone;
        this.address = address;
        this.desc = desc;
    }
}
```

重叠构造器可行，但是当有许多参数的时候，创建使用代码会很难写，并且较难以阅读。

2 JavaBeans模式

遇到许多构造器参数的时候，还有第二种代替办法，即JavaBeans模式。在这种模式下，调用一个无参构造器来创建对象，然后调用setter办法来设置每个必要的参数，以及每个相关的可选参数：

```
/**
 * 使用JavaBeans模式
 */
public class Person {
    //必要参数
    private int id;
    private String name;
    //可选参数
    private int age;
    private String sex;
    private String phone;
    private String address;
    private String desc;
public void setId(int id) {
        this.id = id;
    }

public void setName(String name) {
        this.name = name;
    }
```

```
public void setAge(int age) {
        this.age = age;
    }

public void setSex(String sex) {
        this.sex = sex;
    }

public void setPhone(String phone) {
        this.phone = phone;
    }

public void setAddress(String address) {
        this.address = address;
    }

public void setDesc(String desc) {
        this.desc = desc;
    }
}
```

JavaBeans模式自身有着很重要的缺点。因为构造过程被分到了几个调用中，在构造过程中JavaBean可能处于不一致的状态。类无法仅仅通过检验构造器参数的有效性来保证一致性。

3 Builder模式（推荐）

既能保证像重叠构造器模式那样的安全性，也能保证像JavaBeans模式那么好的可读性。这就是Builder模式的一种形式，不直接生成想要的对象，而是让客户端利用所有必要的参数调用构造器（或者静态工厂），得到一个builder对象。然后客户端在builder对象上调用类似于setter的方法，来设置每个相关的可选参数。最后，客户端调用无参的builder方法来生成不可变的对象。这个builder是它构建类的静态成员类。

```
/**
 * 使用Builder模式
 */
public class Person {
    //必要参数
    private final int id;
    private final String name;
    //可选参数
    private final int age;
    private final String sex;
    private final String phone;
    private final String address;
    private final String desc;
private Person(Builder builder) {
        this.id = builder.id;
        this.name = builder.name;
        this.age = builder.age;
        this.sex = builder.sex;
        this.phone = builder.phone;
        this.address = builder.address;
        this.desc = builder.desc;
    }

public static class Builder {
        //必要参数
        private final int id;
        private final String name;
        //可选参数
        private int age;
        private String sex;
        private String phone;
        private String address;
        private String desc;

public Builder(int id, String name) {
            this.id = id;
            this.name = name;
        }

public Builder age(int val) {
            this.age = val;
            return this;
        }

public Builder sex(String val) {
            this.sex = val;
            return this;
        }
```

```
        public Builder phone(String val) {
            this.phone = val;
            return this;
        }

        public Builder address(String val) {
            this.address = val;
            return this;
        }

        public Builder desc(String val) {
            this.desc = val;
            return this;
        }

        public Person build() {
            return new Person(this);
        }
    }
}
```

注意：Person是不可变的，所有的默认参数值都单独放在一个地方。builder的setter方法返回builder本身。以便可以把连接起来。

```
/**
 * 测试使用
 */
public class Test {

    public static void main(String[] args) {
        Person person = new Person.Builder(1, "张三")
                .age(18).sex("男").desc("测试使用builder模式").build();
        System.out.println(person.toString());
    }
}
```