第 3 章

Spring 快速上手

本章主要回顾 Spring 的基础知识 IOC 和 AOP、IOC 和 AOP 背后的实现原理和设计模式。这些设计模式包括单例模式、简单工厂模式、工厂方法模式和动态代理模式等。

3.1 Spring IOC 和 DI

3.1.1 Spring IOC 和 DI 概述

学习 Spring，经常会联系到 Spring 的 IOC（控制反转）和 DI（依赖注入）。在 Spring 环境下这两个概念是等同的，控制反转是通过依赖注入来实现的。

IOC 是指原先我们代码里面需要实现的对象创建、维护对象间的依赖关系，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一种描述来让容器知道需要创建的对象与对象的关系。依赖注入的目的是为了解耦，体现一种“组合”的理念。继承一个父类，子类将与父类耦合，组合关系使耦合度大大降低。

Spring IOC 容器负责创建 Bean，并通过容器将 Bean 注入到需要的 Bean 对象上。同时 Spring IOC 容器还负责维护 Bean 对象与 Bean 对象之间的关系。那么，Spring IOC 如何来体现对象与对象之间的关系呢？Spring 提供了 XML 配置和 Java 配置等方式，具体看下面的实例：

```
@Service
public class AyUserServiceImpl implements AyUserService{

    @Resource
    private AyUserDao ayUserDao;

    public List<AyUser> findAll() {
        return ayUserDao.findAll();
    }
}
```

Spring 提供的注解有很多，比如声明 Bean 的注解和注入 Bean 的注解，这些注解在工作中经常被使用，所以有必要在这里重新回顾一下。

声明 Bean 的注解：

- @Component　没有明确的角色。
- @Service　在服务层（业务逻辑层）被使用。
- @Repository　在数据访问层（dao层）被使用。
- @Controller　在表现层（控制层）被使用。

注入 Bean 的注解：

- @Autowired　Spring提供的注解。
- @Resource　JSR-250提供的注解。

注意，@Resource 这个注解属于 J2EE 的，默认安照名称进行装配，名称可以通过 name 属性进行指定。如果没有指定 name 属性，当注解写在字段上时，默认取字段名进行查找。如果注解写在 setter 方法上默认取属性名进行装配。当找不到与名称匹配的 bean 时才按照类型进行装配。但是需要注意的是，如果 name 属性一旦指定，就只会按照名称进行装配。具体代码如下：

```
@Resource(name = "ayUserDao")
private AyUserDao ayUserDao;
```

而@Autowired 这个注解是属于 Spring 的，默认按类型装配。默认情况下要求依赖对象必须存在，如果要允许 null 值，可以设置它的 required 属性为 false，如：@Autowired(required=false)，如果我们想使用名称装配，可以结合@Qualifier 注解进行使用。具体代码如下：

```
@Autowired
@Qualifier("ayUserDao")
private AyUserDao ayUserDao;
```

3.1.2 单例模式

Spring 依赖注入 Bean 实例默认都是单例的，所以有必要来回顾一下单例模式。

对于一个软件系统的某些类而言，无须创建多个实例，例如 Windows 任务管理器，如图 3-1 所示。

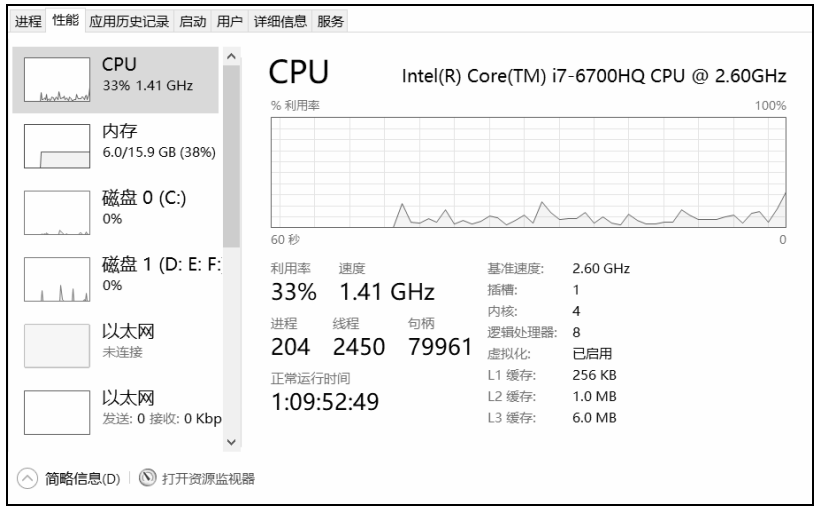

图 3-1 Window 任务管理器

我们没办法打开多个任务管理器，也就是说，在一个 Windows 系统中，任务管理器存在唯一性。为什么要这样设计呢？可以从以下两个方面来分析：其一，如果能弹出多个窗口，且这些窗口的内容完全一致，全部是重复对象，势必会浪费系统资源，任务管理器需要获取系统运行时的诸多信息，这些信息的获取需要消耗一定的系统资源，包括 CPU 资源及内存资源等，浪费是可耻的，而且根本没有必要显示多个内容完全相同的窗口；其二，如果弹出的多个窗口内容不一致，问题就更加严重了，这意味着在某一瞬间系统资源使用情况和进程、服务等信息存在多个状态，例如任务管理器窗口 A 显示“CPU 使用率”为 10%，窗口 B 显示“CPU 使用率”为 15%，到底哪个才是真实的呢？这纯属“调戏”用户，给用户带来误解，更不可取。由此可见，确保 Windows 任务管理器在系统中有且仅有一个非常重要。除了任务管理器外，数据库连接池、应用配置等都是使用单例的。

了解完单例模式的使用场景后，再来看看单例模式的定义。

- **单例模式**（Singleton Pattern）：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式是一种对象创建型模式。

先来看一个传统的创建类的代码：

```
/**
 * 描述：传统创建类实例
 * @author Ay
 * @create 2018/1/23.
 */
public class Case_1 {

    public static void main(String[] args) {
        Singleton singleton = new Singleton();
        Singleton singleton2 = new Singleton();
    }
}

/**
 * 描述：单例类
 */
class Singleton{

}
```

上述代码中，每次 new Singleton()，都会创建一个 Singleton 实例，显然不符合一个类只有一个实例的要求。所以需要对上述代码进行修改，具体修改点如下：

```
/**
 * 描述：单例模式实例
 * @author Ay
 * @create 2018/1/23.
 */
public class Case_1 {
    public static void main(String[] args) {
        //Singleton singleton = new Singleton();
        //单例
        Singleton singleton = Singleton.getInstance();
    }
}

/**
 * 描述：单例类（饿汉模式）
 */
class Singleton{
```

```
    //step 2. 自行对外提供实例
    private static final Singleton singleton = new Singleton();
    //step 1.构造函数私有化
    private Singleton(){}
    //step 3. 提供外界可以获得该实例的方法
    public static Singleton getInstance(){
        return singleton;
    }
}
```

单例模式的写法有很多种，上述代码是一个最简单的饿汉模式的实现方法，在类加载的时候就创建了单例类的对象。由上述代码可知，实现一个单例模式总共有三个步骤：

（1）构造函数私有化。
（2）自行对外提供实例。
（3）提供外界可以获得该实例的方法。

与饿汉模式相对应的还有懒汉模式，懒汉模式有延迟加载的意思，具体代码如下：

```
/**
 * 描述：懒汉模式（存在多线程并发的问题，不是正确的写法）
 * @author Ay
 * @create 2018/04/14
 */
class Singleton{
    private static Singleton singleton = null;
    private Singleton(){}
    public static Singleton getInstance(){
        //1.判断对象是否创建
        if(null == singleton){
            //2.创建对象
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

如果创建单例对象会消耗大量资源，那么延迟创建对象是一个不错的选择，但是懒汉模式有一个明显的问题，就是没有考虑线程安全问题，在多线程并发的情况下，会并发调用getInstance()方法，从而导致系统同时创建多个单例类实例，显然不符合要求。可以通过给getInstance()方法添加锁解决该问题，具体代码如下：

```
/**
 * 描述：懒汉模式（添加 synchronized 锁）
 * @author Ay
 * @create 2018/04/14
 */
class Singleton{
    private static Singleton singleton = null;
    private Singleton(){}
    public static synchronized Singleton getInstance(){
        //1.判断对象是否创建
        if(null == singleton){
            //2.创建对象
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

添加 synchronized 锁虽然可以保证线程安全，但是每次访问 getInstance()方法的时候，都会有加锁和解锁操作，同时 synchronized 锁是添加在方法上面，锁的范围过大，而单例类是全局唯一的，锁的操作会成为系统的瓶颈。因此，需要对代码再进行优化，由此引出了“双重校验锁”的方式，具体代码如下：

```
/**
 * 描述：双重校验锁（指令重排问题）
 * @author Ay
 * @create 2018/04/14
 */
class Singleton{
    private static Singleton singleton = null;
    private Singleton(){}
    public static  Singleton getInstance(){
        //第一次校验
        if(singleton == null){
            synchronized(Singleton.class){
                //第二次检验
                if(singleton == null){
                    //创建对象，非原子操作
                    singleton = new Singleton();
                }
            }
```

```
            }
            return singleton;
        }
    }
```

双重校验锁会出现指令重排的问题，所谓指令重排是指JVM为了优化指令，提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度。singleton = new Singleton()看似原子操作，其实不然，singleton = new Singleton()实际上可以抽象为下面几条JVM 指令：

```
//1：分配对象的内存空间
memory = allocate();
//2：初始化对象
ctorInstance(memory);
//3：设置 instance 指向刚分配的内存地址
singleton = memory;
```

上面操作 2 依赖于操作 1，但是操作 3 并不依赖于操作 2，所以 JVM 是可以针对它们进行指令的优化重排序的，经过重排序后如下：

```
//1：分配对象的内存空间
memory = allocate();
//3：instance 指向刚分配的内存地址，此时对象还未初始化
singleton = memory;
//2：初始化对象
ctorInstance(memory);
```

可以看到，指令重排之后，singleton 指向分配好的内存放在了前面，而这段内存的初始化被排在了后面。在线程 A 执行这段赋值语句，在初始化分配对象之前就已经将其赋值给 singleton 引用，恰好 B 线程进入方法判断 singleton 引用不为 null，然后就将其返回使用，导致程序出错。

为了解决指令重排的问题，可以使用 volatile 关键字修饰 singleton 字段。volatile 关键字的一个语义就是禁止指令的重排序优化，阻止JVM对其相关代码进行指令重排，这样就能够按照既定的顺序指令执行。修改后的代码如下：

```
/**
 * 描述：双重校验锁（volatile 解决指令重排问题）
 * @author Ay
 * @create 2018/04/14
 */
class Singleton{
```

```
    private static volatile  Singleton singleton = null;
    private Singleton(){}
    public static  Singleton getInstance(){
        if(singleton == null){
            synchronized(Singleton.class){
                if(singleton == null){
                    singleton = new Singleton();
                }
            }
        }
       return singleton;
    }
}
```

除了双重校验锁的写法外，比较推荐读者使用最后一种单例模式的写法：静态内部类的单例模式，具体代码如下：

```
/**
 * 描述：静态内部类单例模式（推荐的写法）
 * @author Ay
 * @create 2018/04/14
 */
class ___Singleton{

    //2：私有的静态内部类，类加载器负责加锁
    private static class SingletonHolder{
        private static ___Singleton singleton = new ___Singleton();
    }
    //1:私有化构造方法
    private ___Singleton(){}
    //3:自行对外提供实例
    public static ___Singleton getInstance(){
        return SingletonHolder.singleton;
    }
}
```

当第一次访问类中的静态字段时，会触发类加载，并且同一个类只加载一次。静态内部类也是如此，类加载过程由类加载器负责加锁，从而保证线程安全。这种写法相对于双重检验锁的写法，更加简洁明了，也更加不会出错。

3.1.3 Spring 单例模式源码解析

回顾完单例模式的内容，来看一下 Spring 的 BeanFactory 工厂如何实现单例模式。Spring 的依赖注入（包括 lazy-init 方式）都是发生在 AbstractBeanFactory 的 getBean 方法里。getBean 方法内部调用 doGetBean 方法，doGetBean 方法调用 getSingleton 方法进行 bean 的创建。非 lazy-init 方式，在容器初始化时进行调用，lazy-init 方式，在用户向容器第一次索要 bean 时进行调用。AbstractBeanFactory 类源码具体如下：

```
//省略代码
@Override
public Object getBean(String name) throws BeansException {
    return doGetBean(name, null, null, false);
}

protected <T> T doGetBean(final String name, @Nullable final Class<T>
            requiredType,@Nullable final Object[] args,
            boolean typeCheckOnly) throws BeansException {

    final String beanName = transformedBeanName(name);
    Object bean;

    // Eagerly check singleton cache for manually registered singletons.
        Object sharedInstance = getSingleton(beanName);
    //省略代码
}
@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
        Object singletonObject = this.singletonObjects.get(beanName);
        if (singletonObject == null && isSingletonCurrentlyInCreation
        (beanName)) {
            synchronized (this.singletonObjects) {
                singletonObject = this.earlySingletonObjects.get(beanName);
                if (singletonObject == null && allowEarlyReference) {
                    ObjectFactory<?> singletonFactory
                        = this.singletonFactories.get(beanName);
                    if (singletonFactory != null) {
                        singletonObject = singletonFactory.getObject();
                        this.earlySingletonObjects.put(beanName,
                                                        singletonObject);
```

```
                        this.singletonFactories.remove(beanName);
                    }
                }
            }
        }
        return singletonObject;
    }
```

从上面代码中可以看到，Spring 依赖注入时，使用了双重校验锁的单例模式。首先从缓存 singletonObjects（实际上是一个 ConcurrentHashMap）中获取 bean 实例，如果为 null，对缓存 singletonObjects 加锁，然后再从缓存 earlySingletonObjects（实际上是一个 HashMap）中获取 bean 实例，如果继续为 null，就创建一个 bean。在这里 Spring 并没有使用私有构造方法来创建 bean，而是通过 singletonFactory.getObject() 返回具体 beanName 对应的 ObjectFactory 来创建 bean。一路跟踪下去，发现实际上是调用了 AbstractAutowireCapableBeanFactory 的 doCreateBean 方法，返回了 BeanWrapper 包装并创建的 bean 实例。AbstractAutowireCapableBeanFactory 的 doCreateBean 方法部分代码如下：

```
protected Object doCreateBean(final String beanName,
            final RootBeanDefinition mbd,final @Nullable Object[] args)
            throws BeanCreationException {
        // Instantiate the bean.
        BeanWrapper instanceWrapper = null;
        if (mbd.isSingleton()) {
            instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
        }
        if (instanceWrapper == null) {
            //创建 bean 实例，并返回 instanceWrapper
            instanceWrapper = createBeanInstance(beanName, mbd, args);
        }
        final Object bean = instanceWrapper.getWrappedInstance();
        Class<?> beanType = instanceWrapper.getWrappedClass();
        if (beanType != NullBean.class) {
            mbd.resolvedTargetType = beanType;
        }
        // Allow post-processors to modify the merged bean definition.
        synchronized (mbd.postProcessingLock) {
            if (!mbd.postProcessed) {
                try {
                    applyMergedBeanDefinitionPostProcessors(mbd,
                    beanType, beanName);
```

```
                }
                catch (Throwable ex) {
                throw new BeanCreationException
                    (mbd.getResourceDescription(),beanName,
                    "Post-processing of merged bean definition failed", ex);
                }
                mbd.postProcessed = true;
            }
        }

        // Eagerly cache singletons to be able to resolve circular references
        // even when triggered by lifecycle interfaces like BeanFactoryAware.
        boolean earlySingletonExposure = (mbd.isSingleton() &&
            this.allowCircularReferences &&
                isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            if (logger.isDebugEnabled()) {
                logger.debug("Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
            }
        //增加 beanName 和 ObjectFactory 的键值对应关系
        //获取 bean 的所有后处理器，并进行处理
        addSingletonFactory(beanName, () ->
                            getEarlyBeanReference(beanName, mbd, bean));
        }

//省略大量代码
}
```

- createBeanInstance(beanName, mbd, args)：创建bean实例并返回instanceWrapper。
- addSingletonFactory：增加beanName和ObjectFactory的键值对应关系。
- getEarlyBeanReference(beanName, mbd, bean)：获取bean的所有后处理器，并进行处理。如果是SmartInstantiationAwareBeanPostProcessor类型，就进行处理，如果没有相关处理内容，就返回默认的实现。

getEarlyBeanReference 的具体代码如下：

```
protected Object getEarlyBeanReference(String beanName,
                                       RootBeanDefinition mbd, Object bean) {
        Object exposedObject = bean;
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()){
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof SmartInstantiationAwareBeanPostProcessor){
                    SmartInstantiationAwareBeanPostProcessor ibp
                    = (SmartInstantiationAwareBeanPostProcessor) bp;
                    exposedObject
                    = ibp.getEarlyBeanReference(exposedObject, beanName);
                }
            }
        }
        return exposedObject;
    }
```

3.1.4　简单工厂模式详解

Spring 的 Bean 工厂大量使用工厂方法模式，而工厂方法模式是以简单工厂模式为基础的，所以有必要先来回顾下简单工厂模式的基础知识。

简单工厂模式（Simple Factory Pattern）用来定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。因为在简单工厂模式中用于创建实例的方法是静态（static）方法，因此简单工厂模式又被称为静态工厂方法（Static Factory Method）模式，它属于类创建型模式。

简单工厂模式的要点在于，当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。简单工厂模式结构比较简单，其核心是工厂类的设计，其结构如图 3-2 所示。

在简单工厂模式结构图中包含如下几个角色。

- Factory（**工厂角色**）：工厂角色即工厂类，它是简单工厂模式的核心，负责实现创建所有产品实例的内部逻辑；工厂类可以被外界直接调用，创建所需的产品对象；在工厂类中提供了静态的工厂方法 factoryMethod()，它的返回类型为抽象产品类型 Product。
- Product（**抽象产品角色**）：它是工厂类所创建的所有对象的父类，封装了各种产品对象的公有方法，它的引入将提高系统的灵活性，使得在工厂类中只需定义一个通用的工厂方法，因为所有创建的具体产品对象都是其子类对象。

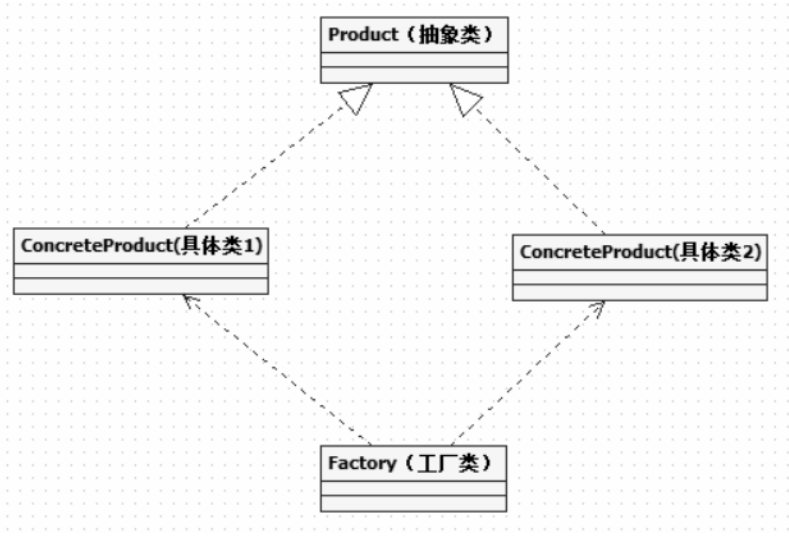

图 3-2　简单工厂模式结构图

- ConcreteProduct（**具体产品角色**）：它是简单工厂模式的创建目标，所有被创建的对象都充当这个角色的某个具体类的实例。每一个具体产品角色都继承了抽象产品角色，需要实现在抽象产品中声明的抽象方法。

来看一个简单工厂模式的例子：

```
/**
 * 描述：交通工具（简单工厂模式）
 * @author Ay
 * @create 2018/1/19.
 */
public class SimpleFactoryPattern {

    public static void main(String[] args) {
          //根据需要传入相关的交通工具名称，获取交通工具实例
        Vehicle vehicle = Factory.produce("car");
        vehicle.run();
    }
}
/**
 * 工厂类
 */
class Factory{
```

```
    //静态方法，生产交通工具
    public static Vehicle produce(String type){
        Vehicle vehicle = null;
        if(type.equals("car")){
            vehicle = new _Car();
            return vehicle;
        }
        if(type.equals("bus")){
            vehicle = new _Bus();
            return vehicle;
        }
        if(type.equals("bicycle")){
            vehicle = new _Bicycle();
            return vehicle;
        }
        return vehicle;
    }
}

/**
 * 交通工具（抽象类）
 */
interface Vehicle{

    void run();
}

/**
 * 汽车（具体类）
 */
class Car implements Vehicle{

    @Override
    public void run() {
        System.out.println("car run...");
    }
}

/**
 * 公交车（具体类）
```

```
 */
class Bus implements Vehicle{
    @Override
    public void run() {
        System.out.println("bus run...");
    }
}

/**
 * 自行车（具体类）
 */
class Bicycle implements Vehicle{
    @Override
    public void run() {
        System.out.println("bicycle run...");
    }
}
```

上述代码中，交通工具都被抽象为 Vehicle 类，而 Car、Bus、Bicycle 类是 Vehicle 类的实现类，并实现 Vehicle 的 run 方法，打印相关的信息。Factory 是工厂类，类中的静态方法 produce 根据传入的不同的交通工具的类型，生产相关的交通工具，返回抽象产品类 Vehicle。上述代码的类结构如图 3-3 所示。

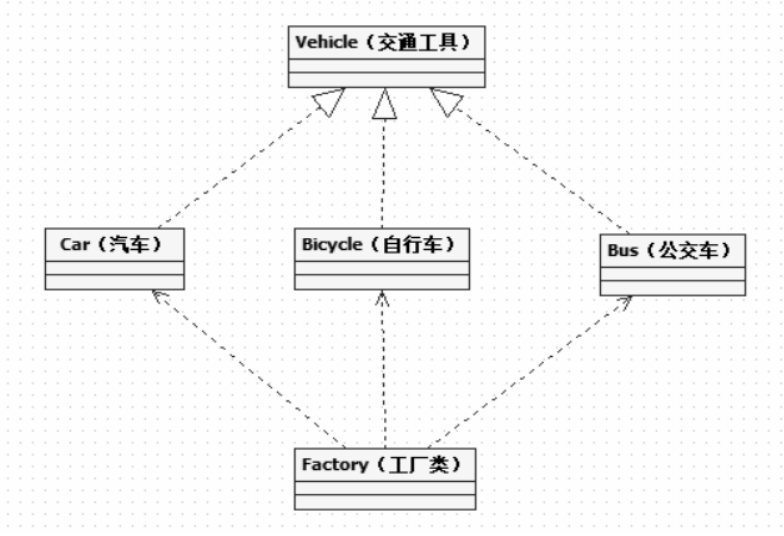

图 3-3　汽车与工厂类结构图

简单工厂模式的主要优点如下：

（1）工厂类包含必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的职责，而仅仅“消费”产品，简单工厂模式实现了对象创建和使用的分离。

（2）客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以在一定程度减少使用者的记忆量。

简单工厂模式的主要缺点如下：

（1）由于工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都会受到影响。

（2）使用简单工厂模式势必会增加系统中类的个数（引入了新的工厂类），增加了系统的复杂度和理解难度。

（3）系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。

（4）简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

3.1.5　工厂方法模式详解

在简单工厂模式中只提供一个工厂类，它需要知道每一个产品对象的创建细节，并决定何时实例化哪一个产品类。简单工厂模式最大的缺点是当有新产品要加入到系统中时，必须修改工厂类，需要在其中加入必要的业务逻辑，这违背了“开闭原则”。此外，在简单工厂模式中，所有的产品都由同一个工厂创建，工厂类职责较重，业务逻辑较为复杂，具体产品与工厂类之间的耦合度高，严重影响了系统的灵活性和扩展性，而工厂方法模式则可以很好地解决这一问题。

在工厂方法模式中，不再提供一个统一的工厂类来创建所有的产品对象，而是针对不同的产品提供不同的工厂，系统提供一个与产品等级结构对应的工厂等级结构。

工厂方法模式的定义：工厂方法模式（Factory Method Pattern）用来定义一个用于创建对象的接口，让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。工厂方法模式又简称为工厂模式（Factory Pattern），还可称作虚拟构造器模式（Virtual Constructor Pattern）或多态工厂模式（Polymorphic Factory Pattern）。工厂方法模式是一种类创建型模式。

工厂方法模式提供一个抽象工厂接口来声明抽象工厂方法，而由其子类来具体实现工厂方法，创建具体的产品对象。工厂方法模式结构如图 3-4 所示。

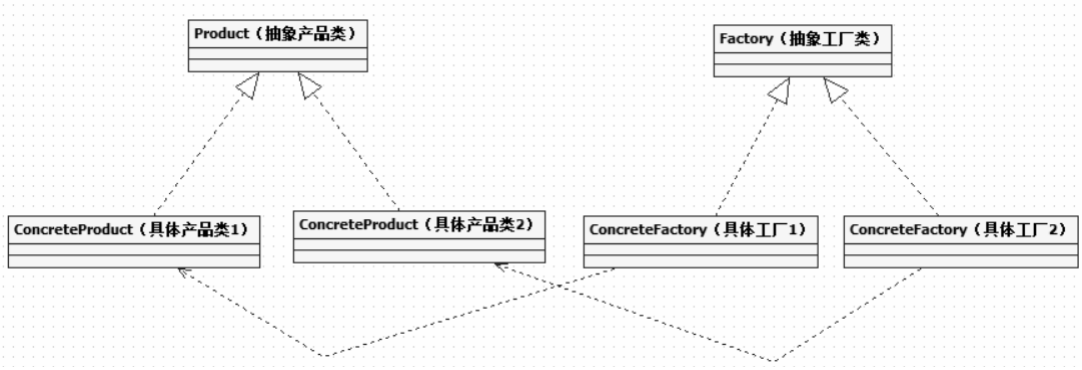

图 3-4　工厂方法模式类结构图

在工厂方法模式结构图中包含如下几个角色。

- Product（抽象产品类）：它是定义产品的接口，是工厂方法模式所创建对象的超类型，也就是产品对象的公共父类。
- ConcreteProduct（具体产品类）：它实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建，具体工厂和具体产品之间一一对应。
- Factory（抽象工厂类）：在抽象工厂类中，声明了工厂方法（Factory Method），用于返回一个产品。抽象工厂是工厂方法模式的核心，所有创建对象的工厂类都必须实现该接口。
- ConcreteFactory（具体工厂类）：它是抽象工厂类的子类，实现了抽象工厂中定义的工厂方法，并可由客户端调用，返回一个具体产品类的实例。

下面来看一个汽车与工厂实例，具体代码如下：

```
/**
 * 描述：汽车与工厂（工厂方法模式）
 * @author Ay
 * @create 2018/1/19.
 */
public class FactoryMethodPattern {
    public static void main(String[] args) throws Exception {
        //生产汽车
        Factory carFactory = new CarFactory();
        Vehicle car = carFactory.produce();
        car.run();
        //生产公交车
        Factory busFactory = new BusFactory();
        Vehicle bus = busFactory.produce();
        bus.run();
```

```
        //生产自行车
        BicycleFactory bicycleFactory = new BicycleFactory();
        Vehicle bicycle = bicycleFactory.produce();
        bicycle.run();
    }
}
/**
 * 抽象工厂类
 */
interface Factory {
    //生产
    Vehicle produce();
}
/**
 * 汽车工厂
 */
class CarFactory implements Factory {
    @Override
    public Vehicle produce() {
        return new Car();
    }
}

/**
 * 公交车工厂
 */
class BusFactory implements Factory {
    @Override
    public Vehicle produce() {
        return new Bus();
    }
}
/**
 * 自行车工厂
 */
class BicycleFactory implements Factory{
    @Override
    public Vehicle produce() {
        return new Bicycle();
    }
```

```
}
/**
*交通工具
*/
interface Vehicle {
    void run();
}

/**
 * 汽车
 */
class Car implements Vehicle {
    @Override
    public void run() {
        System.out.println("car run...");
    }
}

/**
 * 公交车
 */
class Bus implements Vehicle {
    @Override
    public void run() {
        System.out.println("bus run...");
    }
}

/**
 * 自行车
 */
class Bicycle implements Vehicle {
    @Override
    public void run() {
        System.out.println("bicycle run...");
    }
}
```

上述代码中，Vehicle 类是抽象产品类，而 Car、Bus、Bicycle 类是具体产品类，并且实现 Vehicle 类的 run 方法。每一种具体产品类都有一一对应的工厂类 CarFactory、BusFactory、

BicycleFactory 等，所有的工厂都有共同的抽象父类 Factory。汽车与工厂具体的类结构如图 3-5 所示。

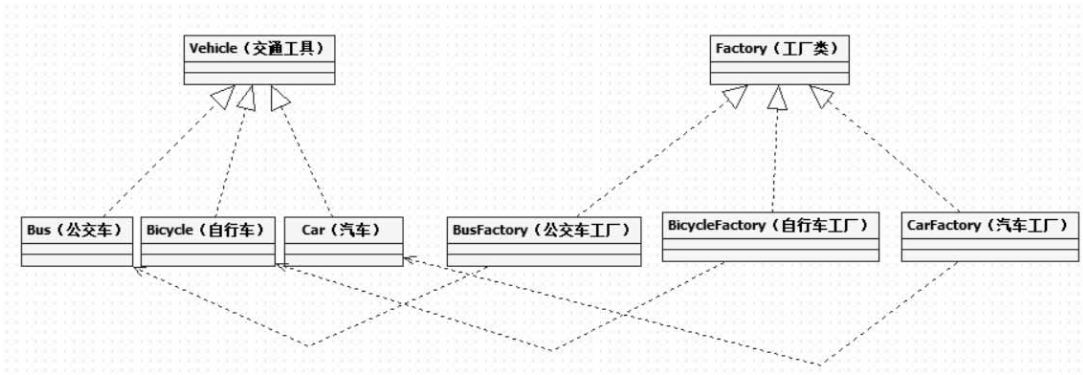

图 3-5　汽车与工厂类结构图

与简单工厂模式相比，工厂方法模式最重要的区别是引入了抽象工厂角色，抽象工厂可以是接口，也可以是抽象类或者具体类。在抽象工厂中声明了工厂方法但并未实现工厂方法，具体产品对象的创建由其子类负责，客户端针对抽象工厂编程，可在运行时再指定具体工厂类，具体工厂类实现了工厂方法，不同的具体工厂可以创建不同的具体产品。

3.1.6　Spring Bean 工厂类详解

Spring 的 bean 工厂类都是存放在 spring-beans-5.0.3.RELEASE.jar 包中的，可以使用 Intellij IDEA 查看 Spring 的类结构图，具体方法是：【打开 DefaultListableBeanFactory 类】→【右键】→【Diagrams】→【Show Diagram】→【Java Class Diagrams】，便可打开如图 3-6 所示的 Spring Bean 工厂类结构图。

- BeanFactory：定义获取bean及bean的各种属性。
- SimgletonBeanRegistry：定义对单例的注册及获取。
- DefaultSimgletonBeanRegistry：对接口SimgletonBeanRegistry函数的实现。
- FactoryBeanRegistrySupport：在DefaultSimgletonBeanRegistry基础上增加了对BeanRegistry的特殊处理功能。
- HierachicalBeanFactory：继承BeanFactory，在BeanFactory定义的功能的基础上增加了对parentFactory的支持。
- ListableBeanFactory：根据条件获取bean的配置清单。
- ConfigurableBeanFactory：提供配置Factory的各种方法。
- AbstractBeanFactory：综合FactoryBeanRegistrySupport和ConfigurableBeanFactory的功能。

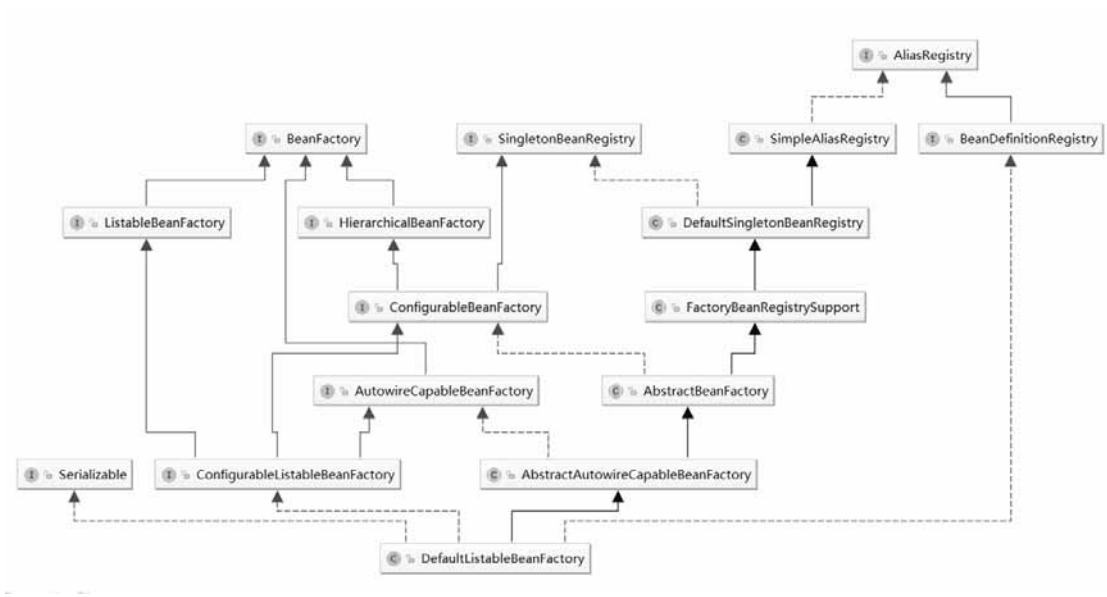

图 3-6　Spring Bean 工厂类结构图

- AutowireCapableBeanFactory：提供创建bean、自动注入、初始化以及应用bean的后处理器。
- ConfigurableListableBeanFactory：BeanFactory配置清单，指定忽略类型及接口等。
- AbstractAutowireCapableBeanFactory：综合AbstractBeanFactory并对接口AutowireCapableBeanFactory进行实现。
- DefaultListableBeanFactory：整个Bean加载的核心部分，是Spring注册和加载Bean的默认实现。

Spring 源码中有非常多的地方用到了工厂模式，几乎是无处不见，但是笔者决定以读者最为常用的 Bean 来讲解，用 Spring 很多程度上是依赖它的对象管理，也就是 IOC 容器对于 Bean 的管理，Spring 的 IOC 容器如何创建和管理 Bean 其实是比较复杂的，它并不在我们本书的讨论范围中，我们关心的是 Spring 如何利用工厂模式来实现更加优良的松耦合设计。

接下来看一下 Spring 中非常重要的一个类 AbstractFactoryBean 是如何利用工厂模式的。

```
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>,
BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {
        /**
        * Expose the singleton instance or create a new prototype instance.
        * @see #createInstance()
        * @see #getEarlySingletonInterfaces()
        */
```

```
        @Override
        public final T getObject() throws Exception {
            if (isSingleton()) {
                return (this.initialized ?
                    this.singletonInstance : getEarlySingletonInstance());
            }
            else {
                return createInstance();
            }
        }

        protected abstract T createInstance() throws Exception;
}
```

- AbstractFactoryBean：实现FactoryBean类，主要是实现getObject方法，返回Bean实例。
- getObject()：如果是单例且已经创建，返回单例模式，未创建调用getEarlySingletonInstance方法，不是单例模式，调用createInstance方法。
- createInstance()：由子类负责创建具体对象。

3.2　Spring AOP

3.2.1　Spring AOP 概述

AOP 为 Aspect Oriented Programming 的缩写，意为面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP 是 OOP 的延续，是软件开发中的一个热点，也是 Spring 框架中的一个重要内容，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性和开发效率。

AOP 主要的功能有日志记录、性能统计、安全控制、事务处理和异常处理等。

3.2.2　Spring AOP 核心概念

首先，来简单理解一下什么是切面，具体请看图 3-1 所示。

Spring AOP 切面简单理解就像一把刀，在代码执行过程中，可以随意地插入或拔出。在插入的位置或拔出的位置可以“任意妄为”地做自己喜欢的事情，比如记录日志、控制事务、安全验证服务，等等。

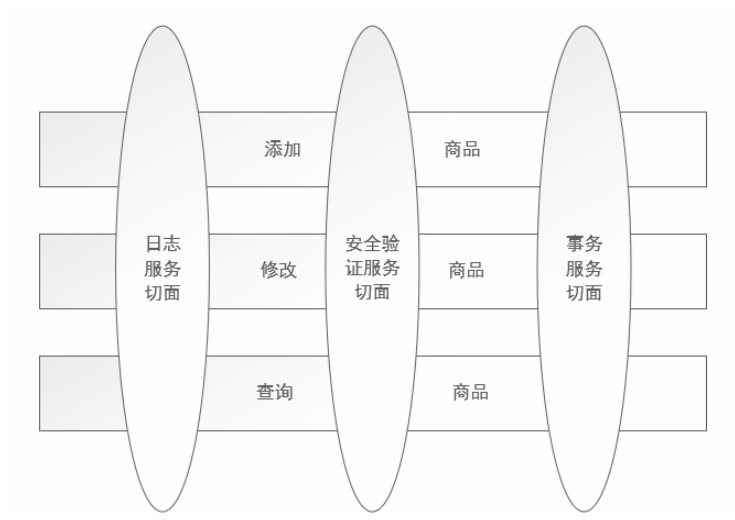

图 3-7　Spring AOP 切面

Spring AOP 核心概念如表 3-1 所示。

表 3-1　Spring AOP 核心概念

名　　称	说　　明
横切关注点	对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点
切面（aspect）	类是对物体特征的抽象，切面就是对横切关注点的抽象
连接点（joinpoint）	被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在 Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或构造器
切入点（pointcut）	对连接点进行拦截的定义
通知（advice）	所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知 5 类
目标对象	代理的目标对象
织入（weave）	将切面应用到目标对象并导致代理对象创建的过程
引入（introduction）	在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段

Spring AOP 通知（advice）分成 5 类，具体如表 3-2 所示。

表 3-2　Advice 通知类型

名　　称	说　　明
前置通知（Before advice）	在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常
正返回通知（After returning advice）	在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行

（续表）

名　　称	说　　明
异常返回通知（After throwing advice）	在连接点抛出异常后执行
后通知（After（finally）advice）	在连接点执行完成后执行，不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容
环绕通知（Around advice）	环绕通知围绕在连接点前后，比如一个方法调用的前后。这是最强大的通知类型，能在方法调用前后自定义一些操作。环绕通知还需要负责决定是继续处理 join point（调用 ProceedingJoinPoint 的 proceed 方法）还是中断执行

3.2.3　JDK 动态代理实现日志框架

Spring AOP 内部是使用动态代理模式来实现的，这一节通过动态代理模式来实现最简单的日志框架，帮助读者快速理解 Spring AOP 的内部实现原理。

首先，在 springmvc-mybatis-book 项目包 com.ay.test 下创建业务接口类 BusinessClassService，具体代码如下：

```
package com.ay.test;
/**
 * 描述：业务类接口
 * @author Ay
 * @create 2018/04/22
 **/
public interface BusinessClassService {

    void doSomeThing();
}
```

BusinessClassService 业务接口类可以理解为日常开发业务创建的接口类，接口中有一个简单的方法 doSomeThing。然后，开发业务类的实现类 BusinessClassServiceImpl，具体代码如下：

```
package com.ay.test;

/**
 * 描述：业务实现类
 * @author Ay
 * @create 2018/04/22
 **/
public class BusinessClassServiceImpl implements BusinessClassService{

    /**
```

```
     * 处理业务
     */
    public void doSomeThing(){
        System.out.println("do something ......");
    }
}
```

实现类 BusinessClassServiceImpl 实现了 BusinessClassService 接口，并实现了 doSomeThing 方法，在方法中打印“do something ......”。

接着，开发日志接口类 MyLogger，具体代码如下：

```
package com.ay.test;
import java.lang.reflect.Method;
/**
 * 描述：日志类接口
 * @author Ay
 * @create 2018/04/22
 **/
public interface MyLogger {
    /**
     * 纪录进入方法时间
     */
    void saveIntoMethodTime(Method method);
    /**
     * 纪录退出方法时间
     */
    void saveOutMethodTime(Method method);
}
```

- saveIntoMethodTime：记录进入方法的时间。
- saveOutMethodTime：记录退出方法的时间。

接口类 MyLogger 开发完成之后，用 MyLoggerImpl 类实现它，具体代码如下：

```
package com.ay.test;
import java.lang.reflect.Method;

/**
 * 描述：日志实现类
 * @author Ay
 * @create 2018/04/22
 **/
```

```
    public class MyLoggerImpl implements MyLogger {

        public void saveIntoMethodTime(Method method) {
            System.out.println("进入" + method.getName()  +
                               "方法时间为: " + System.currentTimeMillis());
        }

        public void saveOutMethodTime(Method method) {
            System.out.println("退出" + method.getName() + "方法时间为: " +
                               System.currentTimeMillis());
        }
    }
```

MyLoggerImpl 类实现接口 MyLogger，并实现 saveIntoMethodTime 和 saveOutMethodTime 方法，在方法内部打印进入/退出方法的时间。

最后，实现最重要的类 MyLoggerHandler，具体代码如下：

```
    package com.ay.test;
    import javax.annotation.Resource;
    import java.lang.reflect.InvocationHandler;
    import java.lang.reflect.Method;

    /**
     * 描述：日志类 Handler
     * @author Ay
     * @create 2018/04/22
     **/
    public class MyLoggerHandler implements InvocationHandler {

        //原始对象
        private Object objOriginal;
        //这里很关键
        private MyLogger myLogger = new MyLoggerImpl();

        public MyLoggerHandler(Object obj){
            super();
            this.objOriginal = obj;
        }

        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
```

```
        Object result = null;
        //日志类的方法：保存进入方法的时间
        myLogger.saveIntoMethodTime(method);
        //调用代理类方法
        result = method.invoke(this.objOriginal ,args);
        //日志类方法：保存退出方法的时间
        myLogger.saveOutMethodTime(method);
        return result;
    }
}
```

- InvocationHandler：该接口中仅定义了一个方法：public Object invoke(Object obj, Method method, Object[] args)，在使用时，第一个参数obj一般是指代理类，method是被代理的方法，args为该方法的参数数组。这个抽象方法在代理类中动态实现。

所有的代码开发完成之后，开发测试类 MyLoggerTest 进行测试，具体代码如下：

```
package com.ay.test;
import java.lang.reflect.Proxy;
/**
 * 描述：测试类
 * @author Ay
 * @create 2018/04/22
 **/
public class MyLoggerTest {

    public static void main(String[] args) {
        //实例化真实项目中业务类
        BusinessClassService businessClassService =
                                new BusinessClassServiceImpl();
        //日志类的 handler
        MyLoggerHandler myLoggerHandler =
                                new MyLoggerHandler(businessClassService);
        //获得代理类对象
        BusinessClassService businessClass = (BusinessClassService)
        Proxy.newProxyInstance(businessClassService.getClass().
        getClassLoader(),businessClassService.getClass().getInterfaces(),
        myLoggerHandler);
        //执行代理类方法
        businessClass.doSomeThing();
    }
}
```

- Proxy.newProxyInstance：该类即为动态代理类，static Object newProxyInstance (ClassLoader loader, Class[] interfaces, InvocationHandler h)，返回代理类的一个实例，返回后的代理类可以当作被代理类使用。在Proxy.newProxyInstance方法中，共有以下三个参数:
  - ClassLoader loader：targetObject.getClass().getClassLoader()目标对象通过getClass方法获取类的所有信息后，调用getClassLoader()方法来获取类加载器。获取类加载器后，可以通过这个类型的加载器，在程序运行时，将生成的代理类加载到JVM即Java虚拟机中，以便运行时需要。
  - Class[] interfaces：targetObject.getClass().getInterfaces()获取被代理类的所有接口信息，以便于生成的代理类可以具有代理类接口中的所有方法。
  - InvocationHandler h：使用动态代理是为了更好地扩展，比如在方法之前做什么操作，之后做什么操作，这个时候这些公共的操作可以统一交给代理类去做。此时需要调用实现了InvocationHandler 类的一个回调方法。

运行测试类的 main 方法，便可以在 Intellij IDEA 控制台查看打印信息，具体信息如下：

```
进入 doSomeThing 方法时间为：1524385006965
do something ......
退出 doSomeThing 方法时间为：1524385006966
```

以上就是利用动态代理模式实现简单的日志框架，具体的结构如图 3-8 所示。

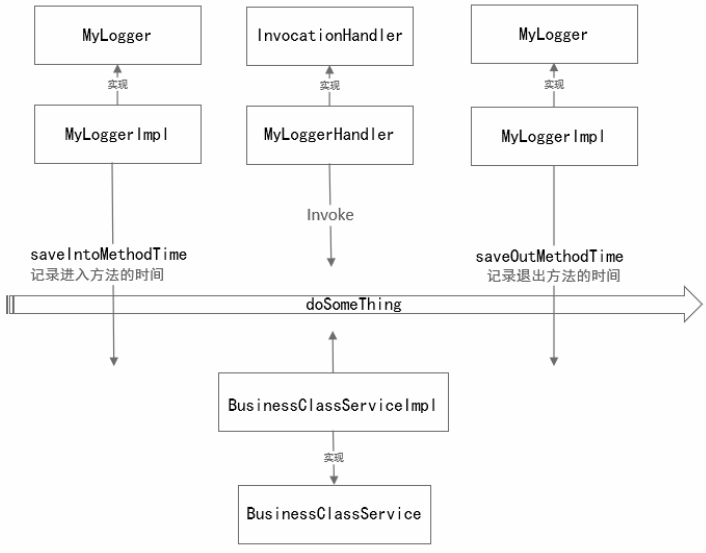

图 3-8　Spring AOP 实现日志框架结构图

这里总结一下 JDK 动态代理的一般实现步骤：

（1）创建一个实现 InvocationHandler 接口的类 MyLoggerHandler，它必须实现 invoke 方法。
（2）创建被代理的类 BusinessClassService 以及接口 BusinessClassServiceImpl。
（3）调用 Proxy 的静态方法 newProxyInstance，创建一个代理类。
（4）通过代理类调用方法。

3.2.4 Spring AOP 实现日志框架

使用 Spring AOP 的注解方式实现日志框架是非常简单的。首先，在配置文件 spring-mvc.xml 中添加配置，具体代码如下：

```
<aop:aspectj-autoproxy proxy-target-class="true">
```

- </aop:aspectj-autoproxy>：声明自动为Spring容器中那些配置@aspectJ切面的bean创建代理，织入切面。<aop:aspectj-autoproxy />有一个proxy-target-class属性，默认为false，表示使用JDK动态代理织入增强，当配置poxy-target-class为true时，表示使用CGLib动态代理技术织入增强。不过即使设置proxy-target-class为false，如果目标类没有声明接口，则Spring将自动使用CGLib动态代理。

配置添加完成之后，要定义一个切面 LogInterceptor，具体代码如下：

```
package com.ay.proxy;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

/**
 * 描述：日志拦截类（切面）
 * @author Ay
 * @create 2018/04/22
 **/
@Aspect
@Component
public class LogInterceptor {

    @Before(value = "execution(* com.ay.controller.*.*(..))")
    public void before(){
        System.out.println("进入方法时间为:" + System.currentTimeMillis());
    }
```

```
    @After(value = "execution(* com.ay.controller.*.*(..))")
    public void after(){
        System.out.println("退出方法时间为:" + System.currentTimeMillis());
    }
}
```

- @Aspect：标识LogInterceptor类为一个切面，供容器读取。
- @Before：在所拦截方法执行之前执行before方法。
- @After：在所拦截方法执行之后执行after方法。
- @Around：可以同时在所拦截方法的前后执行一段逻辑。
- execution切入点指示符：com.ay.controller.*.*(..)表示在controller包中定义的任意方法的执行。execution切入点指示符执行表达式的格式如下：

```
execution（modifiers-pattern? ret-type-pattern declaring-type-pattern?
name-pattern（param-pattern）throws-pattern?）
```

翻译为：

```
execution(方法修饰符　方法返回值　方法所属类 匹配方法名（方法中的形参表）方法申明抛出的异常)
```

其中黑色字体部分不能省略，各部分都支持通配符“*”来匹配全部。比较特殊的为形参表部分，其支持以下两种通配符：

```
"*" ：代表一个任意类型的参数。
"..":代表零个或多个任意类型的参。
```

例如：

```
()：匹配一个无参方法
(..)：匹配一个可接受任意数量参数和类型的方法
(*)：匹配一个接受一个任意类型参数的方法
(*, Integer)：匹配一个接受两个参数的方法，第一个可以为任意类型，第二个必须为 Integer。
```

下面举一些 execution 的使用实例，具体内容见表 3-3。

表 3-3　切入点表达式实例

切入点表达式	说　　明
execution(public * * (..))	匹配所有目标类的 public 方法，第一个*为返回类型，第二个*为方法名
execution(* save* (..))	匹配所有目标类以 save 开头的方法，第一个*代表返回类型

（续表）

切入点表达式	说　明
execution(**product(*,String))	匹配目标类所有以 product 结尾的方法，并且其方法的参数表第一个参数可为任意类型，第二个参数必须为 String
execution(* aop_part.Demo1.service.*(..))	匹配 service 接口及其实现子类中的所有方法
execution(* aop_part.*(..))	匹配 aop_part 包下的所有类的所有方法，但不包括子包
execution(* aop_part..*(..))	匹配 aop_part 包下的所有类的所有方法，包括子包。（当".."出现在类名中时，后面必须跟“*”,表示包、子孙包下的所有类）
execution(* aop_part..*.*service.find*(..))	匹配 aop_part 包及其子包下的所有后缀名为 service 的类中，所有方法名必须以 find 为前缀的方法
execution(*foo(String,int))	匹配所有方法名为 foo，且有两个参数，其中，第一个的类型为 String，第二个的类型为 int
execution(* foo(String,..))	匹配所有方法名为 foo，且至少含有一个参数，并且第一个参数为 String 的方法（后面可以有任意个类型不限的形参）

切面类 LogInterceptor 开发完成之后，重新启动 springmvc-mybatis-book 项目，项目成功启动后，在浏览器输入网址：http://localhost:8080/user/findAll，便可以在 Intellij IDEA 开发工具的控制台看到如下的打印信息：

```
进入方法时间为:1524411433320
id: 1
name: 阿毅
id: 2
name: 阿兰
退出方法时间为:1524411434036
```

3.2.5 静态代理与动态代理模式

Spring AOP 使用的是动态代理模式，所以有必要简单学习一下动态代理模式。

代理模式定义如下：

代理模式给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。

代理模式是一种对象结构型模式。在代理模式中引入了一个新的代理对象，代理对象在客户端对象和目标对象之间起到中介的作用，它去掉客户不能看到的内容和服务或者增添客户需要的额外的新服务。

代理模式的结构比较简单，其核心是代理类，为了让客户端能够一致性地对待真实对象和代理对象，在代理模式中引入了抽象层，代理模式结构如图 3-9 所示。

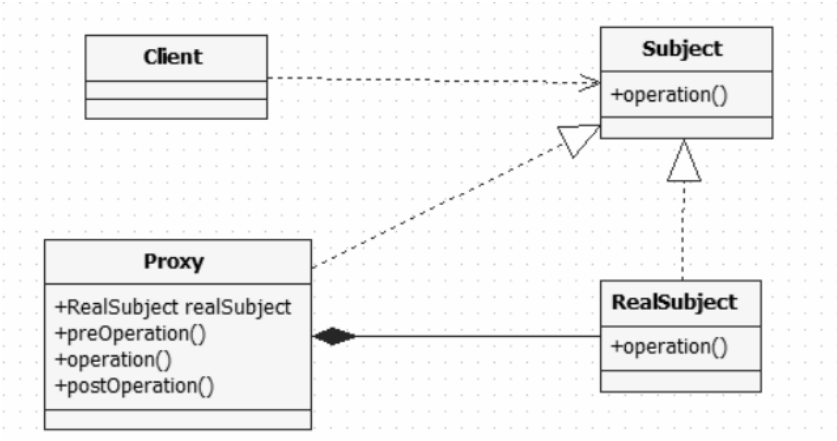

图 3-9　代理模式类结构图

由图 3-9 可知，代理模式包含如下三个角色：

- Subject（**抽象主题角色**）：它声明了真实主题和代理主题的共同接口，这样一来在任何使用真实主题的地方都可以使用代理主题，客户端通常需要针对抽象主题角色进行编程。
- Proxy（**代理主题角色**）：它包含了对真实主题的引用，从而可以在任何时候操作真实主题对象；在代理主题角色中提供一个与真实主题角色相同的接口，以便在任何时候都可以替代真实主题；代理主题角色还可以控制对真实主题的使用，负责在需要的时候创建和删除真实主题对象，并对真实主题对象的使用加以约束。通常，在代理主题角色中，客户端在调用所引用的真实主题操作之前或之后还需要执行其他操作，而不仅仅是单纯调用真实主题对象中的操作。
- RealSubject（**真实主题角色**）：它定义了代理角色所代表的真实对象，在真实主题角色中实现了真实的业务操作，客户端可以通过代理主题角色间接调用真实主题角色中定义的操作。

静态代理模式具体的代码如下：

```
package com.ay.test;
/**
 * 描述：客户端类
 * @author Ay
 * @create 2018/04/22
 **/
public class ProxyPattern{
    public static void main(String[] args) {
```

```
        //为每个 RealSubject 创建代理类 Proxy
        Proxy proxy = new Proxy(new RealSubject());
        proxy.operation();
    }
}
/**
 * 描述：抽象主题类
 * @author Ay
 * @create 2018/04/22
 **/
abstract class Subject {
    abstract void operation();
}

/**
 * 描述：具体主题类
 * @author Ay
 * @create 2018/04/22
 **/
class RealSubject extends Subject{

    void operation() {
        System.out.println("operation ......");
    }
}
/**
 * 描述：代理类
 * @author Ay
 * @create 2018/04/22
 **/
class Proxy extends Subject{

    private Subject subject;

    public Proxy(Subject subject){
        this.subject = subject;
    }

    void operation() {
        //前置处理
```

```
            this.preOperation();
            //具体操作
            subject.operation();
            //后置处理
            this.postOperation();
        }

        void preOperation(){
            System.out.println("pre operation......");
        }

        void postOperation(){
            System.out.println("post operation......");
        }
    }
```

上面介绍的代理模式也被称为“静态代理模式”，这是因为在编译阶段就要为每个 RealSubject 类创建一个 Proxy 类，当需要代理的类很多时，就会出现大量的 Proxy 类，所以可以使用 JDK 动态代理解决这个问题。关于 JDK 动态代理实例，读者可以参考 3.2.3 节，JDK 动态代理的实现原理是动态创建代理类并通过指定类加载器加载，然后在创建代理对象时将 InvokerHandler 对象作为构造参数传入。当调用代理对象时，会调用 InvokerHandler.invoke()方法，并最终调用真正业务对象的相应方法。

第 4 章

MyBatis 映射器与动态 SQL

本章主要介绍 MyBatis 常用的映射器元素、动态 SQL 元素、MyBatis 注解配置和关联映射。

4.1 MyBatis 映射器

4.1.1 映射器的主要元素

Mybatis 提供了强大的映射器，并且提供了丰富的映射器元素，具体如表 4-1 所示。

表 4-1 映射器元素

元素名称	描　　述
select	映射查询语句
insert	映射插入语句
update	映射更新语句
delete	映射删除语句
sql	可以被其他语句引用的可重用语句块
resultMap	用来描述如何从数据库结果集中来加载对象

（续表）

元素名称	描　　述
cache	给定命名空间的缓存配置
cache-ref	其他命名空间缓存配置的引用

接下来详细讨论映射器中主要元素的用法。

4.1.2　select 元素

select 元素是 Mybatis 中最常用的元素之一，select 元素可以从数据库读取数据，组装数据给业务人员。比如可以在配置文件 AyUserMapper.xml 中使用 select 元素，根据用户 Id 查询 ay_user 表（2.2.4 节已创建）中的具体用户，具体代码如下：

```
<select id="findById" parameterType="String"
                                        resultType="com.ay.model.AyUser">
       SELECT * FROM ay_user
       WHERE id = #{id}
</select>
```

这个语句被称为 findById，接受一个 String 类型的参数，并返回一个 User 类型的对象。参数#{id}是告诉 MyBatis 创建一个预处理语句参数。通过 JDBC，这样的一个参数在 SQL 中会由一个“？”来标识，并被传递到一个新的预处理语句中。上面的 SQL 语句执行时会生成如下 JDBC 代码：

```
String findById =  "SELECT * FROM ay_user WHERE id = ? "
PreparedStatement ps = conn.prepareStatement(findById);
ps.setString(1,id);
```

接口 AyUserDao 中定义的方法如下：

```
AyUser findById(String id);
```

select 元素提供了很多配置属性，具体如表 4-2 所示。

表 4-2　select 元素配置

属性名称	描　　述
id	它和 mapper 的命名空间组合起来是唯一的，id 的值和 DAO 接口的方法名一致。如果不唯一或者不一致，MyBatis 将抛出异常
parameterType	将会传入语句参数类的全名码或者别名，这个属性是可选的，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数，默认值为 unset。可以选择 JavaBean、Map 等复杂的参数类型传递给 SQL

（续表）

属性名称	描　　述
parameterMap	即将废弃的元素，不再讨论
resultType	从语句中返回期望类型的类的完全限定名或别名。注意如果是集合的情形，那应该是集合可以包含的类型，而不能是集合本身。返回时可以使用 resultType 或者 resultMap，但不能同时使用。结果集将通过 JavaBean 的规范映射或定义为 int、double、float 等参数
resultMap	它是映射集的引用，将执行强大的映射功能，可以使用 resultType 或者 resultMap 其中的一个，resultMap 可以给予我们自定义映射规则的机会
flushCache	它的作用是调用 SQL 后，是否要求 MyBatis 清空之前查询的本地缓存和二级缓存，取值为 false/true，默认为 false
useCache	启动二级缓存的开关，取值 true/false，默认值为 true
timeout	设置超时参数，等超时的时候抛出异常，单位为秒
fetchSize	获取记录的总条数设定
statementType	告诉 MyBatis 使用哪个 JDBC 的 Statement 工作，取值为 STATEMENT、 PREPARED 或者 CALLABLE。默认为 PREPARED
resultSetType	它的值包括 FORWARD_ONLY（游标允许向前访问）|SCROLL_SENSITIVE（双向滚动，并及时跟踪数据库更新，以便更改 resultSet 中的数据）|SCROLL_INSENSITIVE（双向滚动，但不及时跟踪数据库更新，数据库里的数据修改，并不在 resultSet 中反应过来）
databaseId	如果设置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句，如果带或者不带的语句都有，则不带的会被忽略
resultOrdered	这个设置仅针对嵌套结果 select 语句：如果设置为 true，就说假设包含了嵌套结果集或者分组了，这样的话，当返回一个主结果行的时候，就不会发生对前面结果集引用的情况。这就使得获取嵌套的结果集时不至于导致内存不够用。默认为 false
resultSets	适应于多个结果集的情况，它将列出执行 SQL 后每个结果集的名称，每个名称之间用逗号分隔。使用比较少

下面再来看几个 select 元素的例子：

```
//实例 1：通过名称查询用户
<select id="findByName"  parameterType="String" resultType=
                                                    "com.ay.model.AyUser">
    SELECT * FROM ay_user
    WHERE name = #{name}
</select>
//实例 2：通过名称查询用户个数
<select id="countByName"  parameterType="String" resultType="int">
    SELECT count(*) FROM ay_user
```

```
    WHERE name = #{name}
</select>
```

对应的 AyUserDao 接口如下：

```
List<AyUser> findByName(String name);
int countByName(String name);
```

4.1.3　insert 元素

insert 元素用来映射 DML 语句，MyBatis 会在执行插入之后返回一个整数，来表示进行操作后插入的记录数，insert 元素的属性和 select 元素属性差不多，特有的属性如表 4-3 所示。

表 4-3　insert 元素配置

属性名称	描　　述
useGeneratedKeys	令 MyBatis 使用 JDBC 的 useGeneratedKeys 方法来获取由数据库内部生成的主键，例如 MySQL 和 SQL Server 自动递增字段，Oracle 的序列等，使用它时必须给 keyProperty 或者 keyColumn 赋值
keyProperty	表示以哪个列作为属性的主键，不能和 keyColumn 同时使用
keyColumn	指明第几列是主键，不能和 keyProperty 同时使用，只接受整形参数

下面来看几个例子：

```
//实例 1：插入用户数据
<insert id="insert" parameterType="com.ay.model.AyUser">
INSERT INTO ay_user(id, name, password) VALUE (#{id}, #{name}, #{password});
</insert>
//实例 2：插入用户数据，主键自增
<insert id="insert" useGeneratedKeys="true"
keyProperty="id" parameterType="com.ay.model.AyUser">
    INSERT INTO ay_user(name, password) VALUE (#{name}, #{password});
</insert>
```

对应的 AyUserDao 接口如下：

```
int insert(AyUser ayUser);
```

4.1.4　selectKey 元素

可以使用 keyProperty 属性指定哪个是主键字段，同时使用 useGeneratedKeys 属性告诉 MyBatis 这个主键是否使用数据库内置策略生成。实际工作中往往并非想象中的那么简单，比如希望通过原有主键 Id + 1 的方式生成主键 Id，具体代码如下：

```
    <insert id="insert" useGeneratedKeys="true"
    keyProperty="id" parameterType="com.ay.model.AyUser">
        <selectKey keyProperty="id" resultType="int" order="BEFORE">
            SELECT MAX(id) + 1 AS id FROM ay_user
        </selectKey>
            INSERT INTO ay_user(id, name, password) VALUE (#{id}, #{name},
#{password});
    </insert>
```

selectKey 元素描述如下：

```
    <selectKey
      keyProperty="id"
      resultType="int"
      order="BEFORE"
      statementType="PREPARED">
```

selectKey 元素属性配置具体如表 4-4 所示。

表 4-4　selectKey 元素配置

属性名称	描　　述
keyProperty	selectKey 语句结果应该被设置的目标属性，一般会设置到 id 属性，如果希望得到多个生成的列，可以用逗号分隔属性名称列表
keyColumn	匹配属性的返回结果集中的列名称。如果希望得到多个生成的列，可以用逗号分隔属性名称列表
resultType	结果的类型
order	可以设置为 BEFORE 或者 AFTER。设置为 BEFORE，那么它会首先选择主键，设置 keyProperty，然后执行插入语句。如果设置为 AFTER，那么先执行插入语句，然后是 selectKey 元素
statementType	与 select 元素属性相同，具体看 4.1.2 节的内容

4.1.5　update 元素

update 元素用来映射 DML 语句，主要用来更新数据库中的数据，MyBatis 会在执行更新操作之后返回一个整数，来表示进行操作后更新的记录数，update 元素属性和 select 元素属性差不多，它们特有的属性如表 4-5 所示。

表 4-5　update 元素配置

属性名称	描　　述
useGeneratedKeys	令 MyBatis 使用 JDBC 的 useGeneratedKeys 方法来获取由数据库内部生成的主键，例如 MySQL 和 SQL Server 自动递增字段，Oracle 的序列等，使用它时必须给 keyProperty 或者 keyColumn 赋值
keyProperty	表示以哪个列作为属性的主键，不能和 keyColumn 同时使用
keyColumn	指明第几列是主键，不能和 keyProperty 同时使用，只接受整形参数

下面来看一个具体的实例：

```
<update id="update" parameterType="com.ay.model.AyUser">
    UPDATE ay_user SET
    name = #{name},
    password = #{password}
    WHERE id = #{id}
</update>
```

对应的 AyUserDao 接口如下：

```
int update(AyUser ayUser);
```

4.1.6　delete 元素

delete 元素用来映射 DML 语句，主要用来删除数据库中的数据，MyBatis 会在执行删除操作之后返回一个整数，来表示进行删除后更新的记录数，delete 元素属性和 select 元素属性差不多，它们特有的属性如表 4-6 所示。

表 4-6　delete 元素配置

属性名称	描　　述
useGeneratedKeys	令 MyBatis 使用 JDBC 的 useGeneratedKeys 方法来获取由数据库内部生成的主键，例如 MySQL 和 SQL Server 自动递增字段，Oracle 的序列等，使用它时必须给 keyProperty 或者 keyColumn 赋值
keyProperty	表示以哪个列作为属性的主键，不能和 keyColumn 同时使用
keyColumn	指明第几列是主键，不能和 keyProperty 同时使用，只接受整形参数

下面来看几个具体的实例：

```
//实例 1：根据 id 删除记录
<delete id="delete" parameterType="int">
    DELETE FROM ay_user
```

```
    WHERE id = #{id}
</delete>
//实例 2：根据 name 删除记录
<delete id="deleteByName" parameterType="String">
      DELETE FROM ay_user
      WHERE name = #{name}
</delete>
```

4.1.7 sql 元素

sql 元素可以被用来定义可重用的 SQL 代码段，可以包含在其他语句中。它可以被静态地（在加载参数时）参数化，比如有一条 SQL 语句需要查询十几个字段映射到 JavaBean 中去，而其他的 SQL 语句也有类似的需求，重复写这些字段显然不合理，那么就可以用 sql 元素对这些字段进行“封装”，以达到重复使用。

下面来看几个具体的实例：

```
<sql id="userField">
   a.id as "id",
   a.name as "name",
   a.password as "password"
</sql>
<!-- 获取所有用户 -->
<select id="findAll" resultType="com.ay.model.AyUser">
   Select
       //使用 refid 进行引用
   <include refid="userField"/>
   from ay_user a
</select>
```

可以很方便地使用 include 元素的 refid 属性进行引用，还可以使用定制参数来使用 sql 元素，具体代码如下：

```
<sql id="userField">
          //注意：这里使用$符合而不是#符号，否则程序出现异常
      ${prefix}.id as "id",
      ${prefix}.name as "name",
      ${prefix}.password as "password"
</sql>
<!-- 获取所有用户 -->
<select id="findAll" resultType="com.ay.model.AyUser">
   select
```

```
    <include refid="userField">
        <property name="prefix" value="a"/>
    </include>
    from ay_user a
</select>
```

4.1.8　#与$区别

4.1.7 节中的 SQL 语句，使用${prefix}而不使用#{ prefix }，它们之间的区别是：

（1）#{}将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号，具体示例如下：

```
order by #{id}
//如果 id 传入 11，则 sql 解析成：
order by “11”
```

（2）${}将传入的数据直接显示生成在 sql 中，具体示例如下：

```
order by #{id}
//如果 id 传入 11，则 sql 解析成：
order by 11
```

（3）#方式能够很大程度防止 sql 注入，$方式无法防止 sql 注入。

综上所述，一般建议采用#，而不是$。

4.1.9　resultMap 结果映射集

resultMap 结果映射集是 Mybatis 中重要的元素，它的作用是告诉 MyBatis 从结果集中取出的数据转换为开发者所需要的对象。resultMap 元素还包含其他的元素，具体如下：

```
<resultMap>
<constructor>           /*用来将查询结果作为参数注入到实例的构造方法中*/
      <idArg/>          /*标记结果作为 ID*/
      <arg/>            /*标记结果作为普通参数*/
</constructor>
<id/>                   /*一个 ID 结果，标记结果作为 ID*/
<result/>               /*一个普通结果，JavaBean 的普通属性或字段*/
<association>           /*关联其他的对象*/
</association>
<collection>            /*关联其他的对象集合*/
</collection>
```

```
<discriminator>          /*鉴别器，根据结果值进行判断，决定如何映射*/
      <case></case>      /*结果值的一种情况，将对应一种映射规则*/
</discriminator>
</resultMap>
```

先来看一个具体的示例，代码如下：

```
<sql id="userField">
   ${prefix}.id as "id",
   ${prefix}.name as "name",
   ${prefix}.password as "password"
</sql>
<resultMap id="userMap" type="com.ay.model.AyUser">
   <id property="id" column="id"/>
   <result property="name" column="name"/>
   <result property="password" column="password"/>
</resultMap>

<select id="findAll" resultMap="userMap">
   select
   <include refid="userField">
      <property name="prefix" value="a"/>
   </include>
   from ay_user a
</select>
```

使用 POJO 存储结果集是最常用的方式，也是推荐的方式。resultMap 元素的属性 id 代表这个 resultMap 的标识，type 代表需要映射的 POJO。<id/>元素表示对象的主键，property 代表 POJO 的属性名称，column 代表数据库 SQL 的列名，这样 POJO 和数据库 SQL 的结果就一一对应起来。

result 和 id 两个元素共有的属性如表 4-7 所示。

表 4-7　result 和 id 元素属性配置

属性名称	描　　述
property	令 MyBatis 使用 JDBC 的 useGeneratedKeys 方法来获取由数据库内部生成的主键，例如 MySQL 和 SQL Server 自动递增字段，Oracle 的序列等，使用它时必须给 keyProperty 或者 keyColumn 赋值
column	对应 SQL 的列
javaType	配置 Java 的类型，可以是特定的类完全限定名或者 MyBatis 上下文的别名

（续表）

属性名称	描　　述
jdbcType	配置数据库类型
typeHandler	类型处理器，允许我们用特定的处理器来覆盖 MyBatis 默认的处理器。这要制定 jdbcType 和 javaType 相互转化的规则

其他的元素，比如 collection、association、discriminator 等元素，将会在后续的章节进行描述。

4.2　动态 SQL

4.2.1　动态 SQL 概述

在项目开发过程中，经常需要根据不同的条件拼接 SQL 语句，而 MyBatis 提供了对 SQL 语句动态的组装能力。MyBatis 采用功能强大的基于 OGNL 的表达式来完成动态 SQL。

常用的动态 SQL 元素如表 4-8 所示。

表 4-8　动态 SQL 元素

属性名称	描　　述
if	单条件分支判断语句
choose、when 和 otherwise	多条件分支判断语句，相当于 Java 中的 case when 语句
trim, where, set	用于处理 SQL 拼装问题，辅助元素
foreach	循环语句

4.2.2　if 元素

if 元素主要用来做判断语句，比如要按照名称 name 查询相关的用户，但是 name 参数是可填可不填的条件，如果用户没有填写 name 参数，就不要使用它作为查询条件。具体看下面的示例：

```
<sql id="userField">
    a.id as "id",
    a.name as "name",
    a.password as "password"
</sql>
<resultMap id="userMap" type="com.ay.model.AyUser">
    <id property="id" column="id"/>
```

```
    <result property="name" column="name"/>
    <result property="password" column="password"/>
</resultMap>
//通过用户名 name 和密码 password 查询用户
<select id="findByNameAndPassword" parameterType="String"
                                        resultMap="userMap">
    SELECT
    <include refid="userField"></include>
    from ay_user a
    WHERE 1 = 1
    <if test="name != null and name != ''">
        and name = #{name}
    </if>
    <if test="password != null and password != ''">
        and password = #{password}
    </if>
</select>
```

对应的 AyUserDAO 接口代码如下：

```
List<AyUser> findByNameAndPassword(@Param("name") String name,
@Param("password")String password);
```

if 标签常常与 test 属性联合使用且是必选属性。上述代码中，通过判断 name 或者 password 参数是否为空，如果不为空，拼凑 SQL 语句进行查询。如果为空，则忽略。

4.2.3 choose、when、otherwise 元素

与 if 元素的二重选择相比，choose、when、otherwise 元素提供三重选择，有点类似 switch..case..default 语句，具体示例代码如下所示：

```
<sql id="userField">
    a.id as "id",
    a.name as "name",
    a.password as "password"
</sql>
<resultMap id="userMap" type="com.ay.model.AyUser">
    <id property="id" column="id"/>
    <result property="name" column="name"/>
    <result property="password" column="password"/>
</resultMap>
//通过名称 name 和密码 password 查询用户
```

```
<select id="findByNameAndPassword" parameterType="String"
                                        resultMap="userMap">
     SELECT
     <include refid="userField"></include>
     from ay_user a
     WHERE 1 = 1
     <choose>
        <when test="name != null and name != ''">
           and name = #{name}
        </when>
        <when test="password != null and password != ''">
           and password = #{password}
        </when>
        <otherwise>
             ORDER BY id DESC
        </otherwise>
     </choose>
</select>
```

<choose>标签里可以包含多个<when>标签，<otherwise>标签是可选的并不是必填选项，比如下面的代码：

```
<select id="findByNameAndPassword" parameterType="String"
                                        resultMap="userMap">
     SELECT
     <include refid="userField"></include>
     from ay_user a
     WHERE 1 = 1
     <choose>
        <when test="name != null and name != ''">
           and name = #{name}
        </when>
        <when test="password != null and password != ''">
           and password = #{password}
        </when>
     </choose>
</select>
```

4.2.4 trim、where、set 元素

trim 是更灵活地用来去处多余关键字的标签，它可以实现 where 和 set 的效果。具体内容看下面的示例：

```
<select id="findByNameAndPassword" parameterType="String"
        resultMap="userMap">
    SELECT
      <include refid="userField"></include>
      from ay_user a
      <trim prefix="WHERE"  prefixOverrides="AND">
        <if test="name != null and name != ''">
            and name = #{name}
        </if>
        <if test="password != null and password != ''">
           and password = #{password}
        </if>
    </trim>
</select>
```

假如 name 和 password 字段都不为空，上面的代码相当于如下的 SQL 语句：

```
SELECT
a.id as "id",
a.name as "name",
a.password as "password"
from ay_user a
WHERE name = #{name} and password = #{password}
```

再来看另外一个示例：

```
<update id="update" parameterType="com.ay.model.AyUser">
    UPDATE ay_user
    <trim prefix="SET" suffixOverrides=",">
        <if test="name != null and name != ''">
           name = #{name},
        </if>
        <if test="password != null and password != ''">
           password = #{password},
        </if>
    </trim>
```

```
    WHERE id = #{id}
</update>
```

假如 name 和 password 字段都不为空，上面的代码相当于如下的 SQL 语句：

```
UPDATE ay_user
SET
name = #{name},
password = #{password}
WHERE id= #{id}
```

trim 元素属性配置如表 4-9 所示。

表 4-9　trim 属性元素配置

属性名称	描　　述
prefix	表示在 trim 标签包裹的部分前面添加内容。注意：是在没有 prefixOverrides，suffixOverrides 属性的情况下
prefixOverrides	有 prefix 属性的情况下， prefixOverrides 属性表示去掉 SQL 语句前缀的内容
suffix	表示在 trim 标签包裹的部分后面添加内容。注意：是在没有 prefixOverrides，suffixOverrides 属性的情况下
suffixOverrides	有 prefix 属性的情况下，suffixOverrides 属性表示去掉 SQL 语句后缀的内容

编写 SQL 语句的时候，通常喜欢写这样的 SQL 语句：

```
<select id="findByName"  parameterType="String" resultType=
                                                    "com.ay.model.AyUser">
    SELECT * FROM ay_user WHERE 1 = 1
    <if test="name != null and name != ''">
        and name = #{name}
    </if>
</select>
```

WHERE 1 = 1 这样的条件显然很奇怪，所以可以使用 WHERE 标签优化上面的 SQL 语句，具体代码如下：

```
<select id="findByName"  parameterType="String" resultType=
                                                    "com.ay.model.AyUser">
    SELECT * FROM ay_user
    <where>
        <if test="name != null and name != ''">
            and name = #{name}
        </if>
```

```
    </where>
</select>
```

这样当 where 元素里面的条件成立的时候，才会加入 where 这个 SQL 关键字到组装的 SQL 里面，否则就不加入。

set 元素在执行 SQL 更新中会使用到，先来看一个传统代码写法，具体如下：

```
<update id="update" parameterType="com.ay.model.AyUser">
    UPDATE ay_user SET
    name = #{name},
    password = #{password}
    WHERE id = #{id}
</update>
```

上面代码中没有使用 set 元素，可以对代码进行优化，具体优化后的代码如下：

```
<update id="update" parameterType="com.ay.model.AyUser">
    UPDATE ay_user
    <set>
        <if test="name != null and name != ''">
            name = #{name},
        </if>
        <if test="password != null and password != ''">
            password = #{password},
        </if>
    </set>
    WHERE id = #{id}
</update>
```

set 元素遇到逗号，它会把对应的逗号去掉，比如上面代码中的 password = #{password}，不需要自己写判断语句去除逗号。

4.2.5 foreach 元素

foreach 元素是一个循环语句，作用是遍历集合，支持数组、List、Set 等。具体看下面的示例：

```
//根据 Id 集合查询用户列表
<select id="findByIds" resultType="com.ay.model.AyUser">
    SELECT * FROM ay_user
    WHERE id in
    <foreach item="item" index="index" collection="list"
            open="(" separator="," close=")">
```

```
        #{item}
    </foreach>
</select>
```

foreach 属性元素具体的配置如表 4-10 所示。

表 4-10　foreach 属性元素配置

属性名称	描　　述
item	循环中当前的元素
index	配置的是当前元素在集合的位置下标
collection	接口传递进来的参数名称，可以是一个数组、List、Set 等集合
open	配置的是以什么符号将集合中的元素包装起来，如“(”
separator	各个元素的分隔符
close	配置的是以什么符号将集合中的元素包装起来，如“)”

4.2.6　bind 元素

bind 元素可以从 OGNL 表达式中创建一个变量并将其绑定到上下文。在进行模糊查询的时候，比如通过用户名称 name 查询用户，就常会用到 bind 元素，具体可以看下面的实例：

```
<select id="findByNameAndPassword"
parameterType="String" resultType="com.ay.model.AyUser">
    <bind name="name_pattern" value="'%' + name + '%'"/>
    <bind name="password_pattern" value="'%' + password + '%'"/>
    SELECT * FROM ay_user
    <where>
        <if test="name != null and name != ''">
            and name LIKE #{name_pattern}
        </if>
        <if test="password != null and password != ''">
            and password LIKE #{password_pattern}
       </if>
    </where>
</select>
```

上述的 select 元素中，定义了多个 bind 元素，bind 元素的属性 value 的值："'%' + password + '%'" 会赋值给 name_pattern，然后就可以在 SQL 中直接使用 name_pattern 变量。

4.3 MyBatis 注解配置

4.3.1 MyBatis 常用注解

在前面的章节中，MySQL 的映射器、动态 SQL 语句等知识都是使用基于 XML 的配置方式，其实除了使用 XML 的配置方式，还可以使用基于注解的配置方式。MyBatis 提供了很多好用的注解供使用，具体见表 4-11。

表 4-11 mybatis 常用注解

属性名称	描　　述
@Select	映射查询 SQL 语句
@SelectProvider	Select 语句的动态 SQL 映射。允许指定一个类和一个方法在执行时返回运行的查询语句。有两个属性：type 和 method，type 属性是类的完全限定名，method 是该类中的那个方法名
@Delete	映射删除 SQL 语句
@DeleteProvider	Delete 语句的动态 SQL 映射。允许指定一个类和一个方法在执行时返回运行的删除语句。有两个属性：type 和 method，type 属性是类的完全限定名，method 是该类中的那个方法名
@Insert	映射插入 SQL 语句
@InsertProvider	Delete 语句的动态 SQL 映射。允许指定一个类和一个方法在执行时返回运行的插入语句。有两个属性：type 和 method，type 属性是类的完全限定名，method 是该类中的那个方法名
@Update	映射更新 SQL 语句
@UpdateProvider	Delete 语句的动态 SQL 映射。允许指定一个类和一个方法在执行时返回运行的更新语句。有两个属性：type 和 method，type 属性是类的完全限定名，method 是该类中的那个方法名
@Result	列和属性之间的单独结果映射。属性包括：id、column、property、javaType、jdbcType、type、Handler、one、many。其中 id 属性是一个布尔值，表示是否被用于主键映射
@Results	多个结果映射（Result）列表
@Options	提供配置选项的附加值，通常在映射语句上作为附加功能配置出现
@One	复杂类型的单独属性值映射

（续表）

属性名称	描　　述
@Many	复杂类型的集合属性映射
@Param	当映射器的方法需要多个参数时，注解可以被应用于映射器方法参数来给每个参数取个名字。否则，多参数默认将会以它们的顺序位置和 SQL 语句中的表达式进行映射

4.3.2　@Select 注解

@Select 注解与 XML 配置里的 select 标签相对应，@Results 注解与 resultMap 标签相对应，当在 AyUserDao 接口中使用注解的配置方式时，就不需要在 XML 里面配置，具体实例如下所示：

```
@Repository
public interface AyUserDao {
    //实例 1：查询所有的用户列表
@Select("SELECT * FROM ay_user")
    List<AyUser> findAll();
        //实例 2：查询所有的用户列表
        @Select("SELECT * FROM ay_user")
    @Results({
           @Result(id = true,column = "id",property = "id"),
           @Result(column = "name",property = "name"),
           @Result(column = "password",property = "password")
    })
    List<AyUser> findAll();
    //实例 3：通过 id 查询用户
    @Select("SELECT * FROM ay_user WHERE id = #{id}")
    AyUser findById(String id);
    //实例 4：通过用户名获取用户
    @Select("SELECT * FROM ay_user WHERE name = #{name}")
    List<AyUser> findByName(String name);

}
```

4.3.3　@Insert、@Update、@Delete 注解

@Insert、@Update、@Delete 注解与 XML 配置里的 insert、update、delete 标签相对应，具体实例如下：

```
@Repository
public interface AyUserDao {
    //实例 1：插入用户数据
    @Insert("INSERT INTO ay_user(name,password) VALUES(#{name}, #{password})")
    @Options(useGeneratedKeys = true, keyProperty = "id")
    int insert(AyUser ayUser);
    //实例 2：更新用户数据
    @Update("UPDATE  ay_user SET name = #{name}, password = #{password} WHERE
    id = #{id}")
    int update(AyUser ayUser);
    //实例 3：根据用户 id 删除用户
    @Delete("DELETE FROM ay_user WHERE id = #{id}")
    int delete(int id);
    //实例 4：根据用户名删除用户
    @Delete("DELETE FROM ay_user WHERE name = #{name}")
    int deleteByName(String name);

}
```

@Options 注解提供配置选项的附加值，通常在映射语句上作为附加功能配置出现。

4.3.4 @Param 注解

当映射器的方法需要多个参数时，@Param 注解可以被应用于映射器方法参数来给每个参数取个名字。否则，多参数默认将会以它们的顺序位置和 SQL 语句中的表达式进行映射。具体实例如下：

```
@Select("SELECT * FROM ay_user WHERE name = #{name} and password =
                                                        #{password}")
List<AyUser> findByNameAndPassword(@Param("name") String name,
@Param("password")String password);
```

除了使用@Param 注解映射参数之外，还有其他的方式来映射参数，只是不是很推荐，这里简单的总结一下其他的方法，读者可以做简单的对比。

1. Map的映射方式

AyUserDao 接口的方法定义如下：

```
List<AyUser> findByNameAndPassword(Map<String, String > map);
```

对应的 XML 配置如下：

```
<select id="findByNameAndPassword" parameterType="java.util.Map"
                                        resultMap="userMap">
    SELECT * from ay_user a
    <where>
        <if test="name != null and name != ''">
            and name = #{name}
        </if>
        <if test="password != null and password != ''">
            and password = #{password}
        </if>
    </where>
</select>
```

服务层 AyUserServiceImpl 调用方式如下：

```
public List<AyUser> findByNameAndPassword(Map<String,String> map) {
        Map<String,String> map = new HashMap<String, String>();
        map.put("name","al");
        map.put("password","123");
        return ayUserDao.findByNameAndPassword(map);
}
```

2. 顺序映射方式

AyUserDao 接口的方法定义如下：

```
List<AyUser> findByNameAndPassword(String name,String password);
```

对应的 XML 配置如下：

```
<select id="findByNameAndPassword" parameterType="String"
                                        resultMap="userMap">
    SELECT * from ay_user a
    WHERE 1 = 1 AND name = #{param1} AND password = #{param2}
</select>
```

顺序映射方式是通过参数的顺序进行映射的，name 参数对应#{ param1}，#{password}参数对应 param2，当然这是新版的 Mybatis 提供的，旧版本的 MyBatis 是用#{0}、#{1}进行映射的。

4.4 MyBatis 关联映射

4.4.1 关联映射概述

之前的章节中，我们已介绍了使用 Mybatis 对数据库单表进行映射和执行增删改查操作，但是在现实的项目中进行数据库建模时，需要遵循数据库设计范式的要求，对现实中的业务模型进行拆分，封装到不同的数据表中，表与表之间存在着一对多或多对多的对应关系。进而，对数据库的增删改查操作的主体，也就从单表变成了多表。那么 Mybatis 中是如何实现这种多表关系的映射呢？这是接下来要学习的重点。

关联映射大致可以分为：一对一、一对多、多对多，下面将一一展开描述。

4.4.2 一对一

一对一关联关系在现实生活中很多，比如：一个人只能有一个身份证或者一个母亲。首先，在数据库 springmvc-mybatis-book 中创建数据库表 ay_user、ay_user_address，具体代码如下：

```
-- ----------------------------
-- Table structure for ay_user
-- ----------------------------
DROP TABLE IF EXISTS 'ay_user';
CREATE TABLE 'ay_user' (
  'id' bigint(32) NOT NULL AUTO_INCREMENT,
  'name' varchar(10) DEFAULT NULL,
  'password' varchar(64) DEFAULT NULL,
  'age' int(10) DEFAULT NULL,
  'address_id' bigint(32) DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'FK_address_id' ('address_id'),
  CONSTRAINT 'FK_address_id' FOREIGN KEY ('address_id')
REFERENCES 'ay_user_address' ('id')
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

-- ----------------------------------
-- Table structure for ay_user_address
-- ----------------------------------
DROP TABLE IF EXISTS 'ay_user_address';
CREATE TABLE 'ay_user_address' (
```

```
  'id' bigint(32) NOT NULL,
  'name' varchar(255) DEFAULT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

表 ay_user、ay_user_address 对应的 Model 代码如下：

```
/**
 * 用户实体
 * @author Ay
 * @date  2018/04/02
 */
public class AyUser implements Serializable{

    private Integer id;
    private String name;
    private String password;
    private Integer age;
    //用户和地址一一对应，即一个用户只有一个老家地址
    private AyUserAddress ayUserAddress;

        //省略 set、get 方法
}

/**
 * 描述：用户地址实体
 * @author Ay
 * @create 2018/05/01
 **/
public class AyUserAddress  implements Serializable {

    private Integer id;
    private String name;

    //省略 set、get 方法
}
```

用户和老家地址是一对一关系，即一个用户只能有一个老家地址。在 AyUser 类中定义一个 ayUserAddress 属性，用来映射一对一的关联关系，表示一个人的老家地址。

AyUser 类和 AyUserAddress 类创建完成之后，创建对应的 DAO 接口 AyUserDao 和 AyUserAddressDao，具体代码如下：

```
@Repository
public interface AyUserDao {
```

```
    //根据 id 查询用户
    AyUser findById(String id);
}

@Repository
public interface AyUserAddressDao {
    //根据 id 查询用户地址
    AyUserAddress findById(Integer id);
}
```

DAO 接口 AyUserDao 和 AyUserAddressDao 创建完成之后，继续创建对应的 XML 配置文件 AyUserMapper.xml 和 AyUserAddressMapper.xml，AyUserAddressMapper.xml 具体代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ay.dao.AyUserAddressDao">
    <select id="findById"
 parameterType="Integer" resultType="com.ay.model.AyUserAddress">
       SELECT * FROM ay_user_address WHERE id = #{id}
    </select>
</mapper>
```

AyUserMapper.xml 配置文件具体代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ay.dao.AyUserDao">

    <resultMap id="userMap" type="com.ay.model.AyUser">
       <id property="id" column="id"/>
       <result property="name" column="name"/>
       <result property="password" column="password"/>
       <association property="ayUserAddress" column="address_id"
             select="com.ay.dao.AyUserAddressDao.findById"
 javaType="com.ay.model.AyUserAddress">
       </association>
    </resultMap>

    <select id="findById"  parameterType="String" resultMap="userMap">
      SELECT * FROM ay_user
      WHERE id = #{id}
```

```
    </select>

</mapper>
```

AyUserAddressMapper.xml 配置文件中通过<select>标签定义了一个通过 id 查询地址的简单查询。AyUserMapper.xml 配置文件中同样也配置了通过 id 查询用户的简单查询，查询返回的结果 resultMap。resultMap 标签通过<association>元素映射一对一的关联关系，select 属性表示会使用 column 属性的 address_id 的值作为参数执行 AyUserAddressDao 中定义的 findById 方法查询对应的地址数据，查询出的地址数据被封装到 property 属性的 ayUserAddress 对象当中。这样一对一的关联映射就完成了。

4.4.3　一对多

一对多关联关系在现实生活中也有很多，比如，一个学校可以包含多个学生，一个学生只属于一个学校，学校和学生就是一对多关系，而学生和学校就是多对一的关系。首先，在数据库 springmvc-mybatis-book 中创建数据库表 ay_student、ay_school，具体代码如下：

```
-- ----------------------------
-- Table structure for ay_student
-- ----------------------------
DROP TABLE IF EXISTS 'ay_student';
CREATE TABLE 'ay_student' (
  'id' bigint(32) NOT NULL,
  'name' varchar(255) DEFAULT NULL,
  'age' int(2) DEFAULT NULL,
  'school_id' bigint(32) DEFAULT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Table structure for ay_school
-- ----------------------------
DROP TABLE IF EXISTS 'ay_school';
CREATE TABLE 'ay_school' (
  'id' bigint(32) NOT NULL,
  'name' varchar(255) DEFAULT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据库表 ay_student 和 ay_school 创建完成之后，接着创建对应的实体类 AyStudent 和 AySchool，具体代码如下：

```
/**
 * 描述：学生实体
 * @author Ay
 * @create 2018/05/01
 **/
public class AyStudent implements Serializable {

    private Integer id;
    private String name;
    private Integer age;
    //一个学生只能在一个学校
    private AySchool aySchool;
    //省略 set、get 方法
}
```

一个学生只能在一个学校里，所以在 AyStudent 实体里维护 aySchool 属性，AySchool 实体类的代码如下：

```
/**
 * 描述：学校实体
 * @author Ay
 * @create 2018/05/01
 **/
public class AySchool implements Serializable {

    private Integer id;
    private String name;
    //一个学校有多个学生
    private List<AyStudent> students;

//省略 set、get 方法
}
```

一个学校里面有很多的学生，所以在 AySchool 实体类中维护 students 的集合列表，该集合列表是一个 List 对象。实体类 AyStudent 和 AySchool 创建完成之后，继续创建 AyStudentDao 和 AySchoolDao 接口，具体代码如下：

```
/**
 * 描述：学生 DAO 接口
 * @author Ay
 * @create 2018/05/01
 **/
public interface AyStudentDao {
    List<AyStudent> findBySchoolId(Integer id);
}

/**
 * 描述：学校 DAO 接口
 * @author Ay
 * @create 2018/05/01
 **/
public interface AySchoolDao {

    AySchool findById(Integer id);
}
```

AyStudentDao 和 AySchoolDao 接口创建完成之后，创建对应的 XML 配置文件，AyStudentMapper.xml 具体代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ay.dao.AyStudentDao">
    <resultMap id="studentMap" type="com.ay.model.AyStudent">
        <id property="id" column="id"/>
        <result property="name" column="name"/>
        <result property="age" column="age"/>
        <association property="aySchool" javaType="com.ay.model.AySchool">
            <id property="id" column="id"/>
            <result property="name" column="name"/>
        </association>
    </resultMap>
     <! -- 根据 id 查询学生，关联 ay_school 表 -->
    <select id="findById" parameterType="Integer" resultMap="studentMap">
        SELECT * FROM ay_student s , ay_school c
        WHERE s.school_id = c.id
        AND s.id = #{id}
    </select>
```

```
    <! -- 根据学校 id 查询学生 -->
<select id="findBySchoolId"
parameterType="Integer" resultType="com.ay.model.AyStudent">
       SELECT * FROM ay_student WHERE school_id = #{school_id}
   </select>
</mapper>
```

在 AyStudentMapper.xml 配置文件中，使用<select>标签，通过 id 查询学生实体，同时关联了 ay_school 表，查询结果返回到 studentMap 中。在 studentMap 中，使用< association >元素映射多对一的关联关系。因为<select id="findById" ..>的 SQL 语句是一条多表连接，在查询学生的同时，会把对应的学校查询出来。

AySchoolMapper.xml 具体代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ay.dao.AySchoolDao">

   <resultMap id="schoolMap" type="com.ay.model.AySchool">
      <id property="id" column="id" />
      <result property="name" column="name"/>
      <collection property="students" javaType="ArrayList" column="id"
                ofType="com.ay.model.AyStudent"
                fetchType="lazy"
                 select="com.ay.dao.AyStudentDao.findBySchoolId">
         <id property="id" column="id"/>
         <result property="name" column="name"/>
         <result property="age" column="age"/>
      </collection>
   </resultMap>

   <!-- 根据 id 查询学校 -->
   <select id="findById" parameterType="Integer" resultMap="schoolMap">
      SELECT * FROM ay_school WHERE id = #{id}
   </select>
</mapper>
```

在 AySchoolMapper.xml 配置文件中，使用<select>标签，通过 id 查询学校实体，返回结果到 schoolMap 中。在 resultMap 中，使用<collection>元素映射一对多的关联关系，select 属性表示会使用 column 属性的 id 值作为参数执行 AyStudentDao 接口中的 findBySchoolId 查询该学校下所有的学生数据，查询出的数据将被封装到 property 表示的 students 对象中。

4.4.4　多对多

多对多关联关系在现实生活中也有很多，比如，用户和角色的关系，一个用户可以有多个角色，每个角色也可以有多个用户；学生和老师的关系，一个学生可以跟多个老师，每个老师也可以教多个学生等。首先，在数据库 springmvc-mybatis-book 中创建数据库表 ay_user、ay_role、ay_user_role_rel，具体代码如下：

```
-- ----------------------------
-- Table structure for ay_user
-- ----------------------------
DROP TABLE IF EXISTS 'ay_user';
CREATE TABLE 'ay_user' (
  'id' bigint(32) NOT NULL AUTO_INCREMENT,
  'name' varchar(10) DEFAULT NULL,
  'password' varchar(64) DEFAULT NULL,
  'age' int(10) DEFAULT NULL,
  'address_id' bigint(32) DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'FK_address_id' ('address_id'),
  CONSTRAINT 'FK_address_id' FOREIGN KEY ('address_id')
REFERENCES 'ay_user_address' ('id')
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Table structure for ay_role
-- ----------------------------
DROP TABLE IF EXISTS 'ay_role';
CREATE TABLE 'ay_role' (
  'id' bigint(32) NOT NULL,
  'name' varchar(255) DEFAULT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Table structure for ay_user_role_rel
-- ----------------------------
DROP TABLE IF EXISTS 'ay_user_role_rel';
CREATE TABLE 'ay_user_role_rel' (
  'id' bigint(32) NOT NULL,
```

```
  'user_id' bigint(32) DEFAULT NULL,
  'role_id' bigint(32) DEFAULT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据库表 ay_user、ay_role、ay_user_role_rel 创建完成之后，创建对应的实体对象 AyUser、AyRole、AyUserRoleRel，具体代码如下所示：

```
/**
 * 用户实体
 * @author Ay
 * @date  2018/04/02
 */
public class AyUser implements Serializable{

    private Integer id;
    private String name;
    private String password;
    private Integer age;
    //用户与角色是多对多关系，一个用户有多个角色
    private List<AyRole> ayRoleList;
}

/**
 * 描述：角色实体
 *
 * @author Ay
 * @create 2018/05/01
 **/
public class AyRole implements Serializable {
    private Integer id;
    private String name;
    //角色与用户是多对多关系，一个角色对应多个用户
    private List<AyUser> ayUserList;
}

/**
 * 描述：用户角色关联实体
 * @author Ay
 * @create 2018/05/01
 **/
```

```
public class AyUserRoleRel implements Serializable {

    private Integer id;
    //用户 id
    private Integer userId;
    //角色 id
    private Integer roleId;
}
```

用户和角色是多对多关联关系，在 AyUser 实体对象中定义 List<AyRole> ayRoleList 角色属性，用来维护用户和角色的关系。同理，在 AyRole 实体对象中定义 List<AyUser> ayUserList 用户属性，用来维护角色与用户的关系。实体对象 AyUser、AyRole、AyUserRoleRel 创建完成之后，创建对应的 DAO 接口对象，具体代码如下：

```
@Repository
public interface AyUserDao {
     AyUser findById(String id);
}

@Repository
public interface AyRoleDao {
     AyRole findById(String id);
}
```

AyUserDao 和 AyRoleDao 接口创建完成之后，创建对应的 XML 配置文件 AyUserMapper.xml 和 AyRoleMapper.xml，AyUserMapper.xml 配置文件的具体代码如下所示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ay.dao.AyUserDao">

    <resultMap id="userMap" type="com.ay.model.AyUser">
        <id property="id" column="id"/>
        <result property="name" column="name"/>
        <result property="password" column="password"/>
        <collection property="ayRoleList" javaType="ArrayList" column="id"
                  ofType="com.ay.model.AyRole"
                  fetchType="lazy"
                  select="com.ay.dao.AyRoleDao.findByUserId">
            <id property="id" column="id"/>
```

```
            <result property="name" column="name"/>
        </collection>
    </resultMap>

    <select id="findById"  parameterType="String" resultMap="userMap">
      SELECT * FROM ay_user
      WHERE id = #{id}
    </select>

    <select id="findByRoleId" parameterType="Integer"
                              resultType="com.ay.model.AyUser">
        SELECT * FROM ay_user WHERE id in(
          select user_id from ay_user_role_rel where role_id = #{roleId}
        )
    </select>
</mapper>
```

AyRoleMapper.xml 配置文件的具体代码如下所示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.ay.dao.AyRoleDao">

    <resultMap id="roleMap" type="com.ay.model.AyRole">
        <id property="id" column="id"/>
        <result property="name" column="name"/>
        <collection property="ayUserList" javaType="ArrayList" column="id"
                  ofType="com.ay.model.AyUser"
                  fetchType="lazy"
                  select="com.ay.dao.AyUserDao.findByRoleId">
            <id property="id" column="id"/>
            <result property="name" column="name"/>
            <result property="password" column="password"/>
        </collection>
    </resultMap>
    <select id="findById" parameterType="String" resultMap="roleMap">
        SELECT * FROM ay_role WHERE id = #{id}
    </select>
    <select id="findByUserId" parameterType="Integer"
                                  resultType="com.ay.model.AyRole">
```

```
        SELECT * FROM ay_role WHERE id in(
          select role_id from ay_user_role_rel where user_id = #{userId}
        )
    </select>
</mapper>
```

在 AyUserMapper.xml 配置文件中，通过<select>标签查询用户信息，将返回的结果存放到 userMap 中。userMap 中定义<collection>元素映射一对多的关联关系，select 属性表示会使用 column 属性的 id 值作为参数执行 AyRoleDao 接口中的 findByUserId 方法，查询出的数据将被封装到 property 表示的 ayRoleList 对象当中。

在 AyRoleMapper.xml 配置文件中，通过<select>标签查询角色信息，将返回的结果存放到 roleMap 中。roleMap 中定义<collection>元素映射一对多的关联关系，select 属性表示会使用 column 属性的 id 值作为参数执行 AyUserDao 接口中的 findByRoleId 方法，查询出的数据将被封装到 property 表示的 ayUserList 对象当中，fetchType="lazy"表示使用懒加载的方式加载数据。

开启懒加载模式，需要在 mybatis-config.xml 配置文件中添加如下代码：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
       PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 全局配置参数，需要时再设置 -->
    <settings>
        //省略代码
        <!-- 通过日志记录显示 mybatis 的执行过程 -->
        <setting name="logImpl" value="log4j" />
        <!-- lazyLoadingEnabled 设置为懒加载 -->
        <setting name="lazyLoadingEnabled" value="true"/>
        <!-- aggressiveLazyLoading 主动加载为 false -->
        <setting name="aggressiveLazyLoading" value="false"/>
    </settings>
</configuration>
```