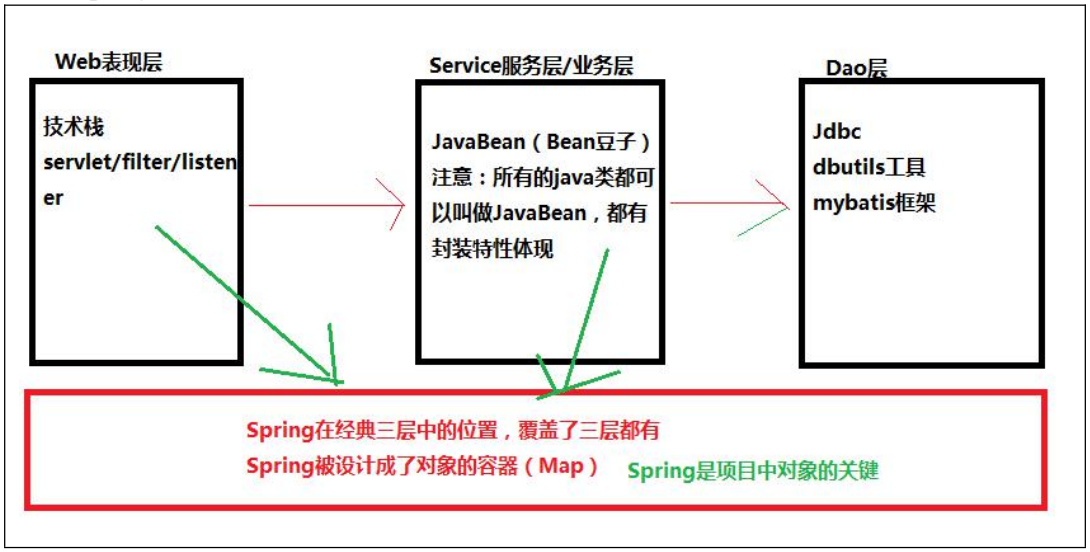
Spring

Spring 框架的江湖地位

- 职业生涯，离不开它
- Spring 框架基于 IOC 和 AOP 两大核心思想所做的技术实现，前无古人
- Spring 框架的体系非常庞大，Spring 全家桶

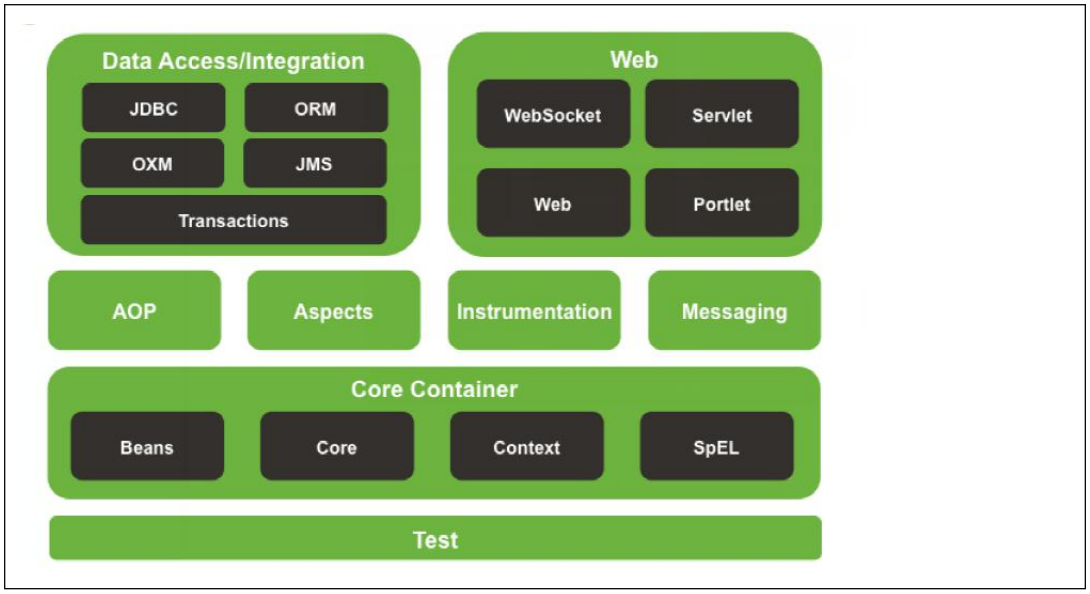
[了解]Spring 介绍

- 什么是 Spring
  - Spring 在经典三层中的位置，各个层都有体现



  - Spring 是一个一站式/full-stack（全栈）的框架
    不仅可以用 Spring 开发 web 层，也可以开发 service 层，还有 dao 层，归根揭底，它是一个对象的容器，它能够处理请求是因为这个容器中有能够处理请求的对象，能够操作数据库，是因为这个容器中有能够操作数据库的对象

    根本上，就是一个对象的容器（Map）
- Spring 的发展历程
  了解讲义
- Spring 的优势
  了解讲义
- Spring 的框架结构



注意：Spring 是一个模块化的框架，它的功能很强大，但是并不要求你完全都得用，图中每一个圆角矩形都代表一个 jar，使用哪些模块引入哪些模块即可

[理解]IOC 的概念和作用

说明：

Spring 所使用的两大思想 IOC 和 AOP，都不是 Spring 提出的，之前更倾向于理论化，Spring 框架把这两大思想在技术上做了非常好的实现。

- IOC 的概念

  IOC：Interverse of control （控制反转/反转控制）。原来我们在类中需要其他类时我们自己 new 对象，有了 Spring 之后，不需要我们自己 new，框架已经准备好了对象，等待我们使用，我们只需要问框架要对象即可

  我们丧失了一个权利（创建对象），得到了一个福利（不用管什么时候创建对象，以及对象什么时候销毁）

- IOC 的作用

  解决的是程序耦合问题

- 关于程序耦合(有一个认识即可)

  耦合：耦合度，为了完成一定的功能，一个类调用了另外一个类，所产生的影响，影响大小代表耦合度的高低。

  注意：耦合不能够绝对避免，耦合只能降低

  “高内聚、低耦合”，麻烦自己，方便别人，你修改时候，别人对你的调用就不用变动

[理解]自定义 IOC 实现程序解耦合

需求：开发 dao 层和 service 层模拟保存账户（Account），不具体操作数据库

- service 层调用 dao 层存在耦合，解耦合分析过程

```
public class AccountServiceImpl implements AccountService{

    /**
     * 调用dao层
     * new关键字让AccountServiceImpl和AccountDaoImpl直接产生了联系，发生了耦合
     * 考虑：使用工厂设计模式解决这个耦合问题
     * 工厂需要一些信息才能使用反射技术帮我们去实例化对象，就是全限定类名，比如com.itheima.spring.dao
       而且根据我们对框架的了解，全限定类名配置在xml配置文件中
        1、配置xml
        2、工厂读取xml处理
     */

    private AccountDao accountDao = (AccountDao) BeanFactory.getBean( id: "accountDao");
```

- 需要实例化的类的全限定类名配置在 xml 中，便于工厂使用反射技术实例化对象

```
<beans>
    <bean id="accountDao" class="com.spring.dao.AccountDaoImpl"></bean>
    <bean id="accountService" class="com.spring.service.AccountServiceImpl"></bean>
</beans>
```

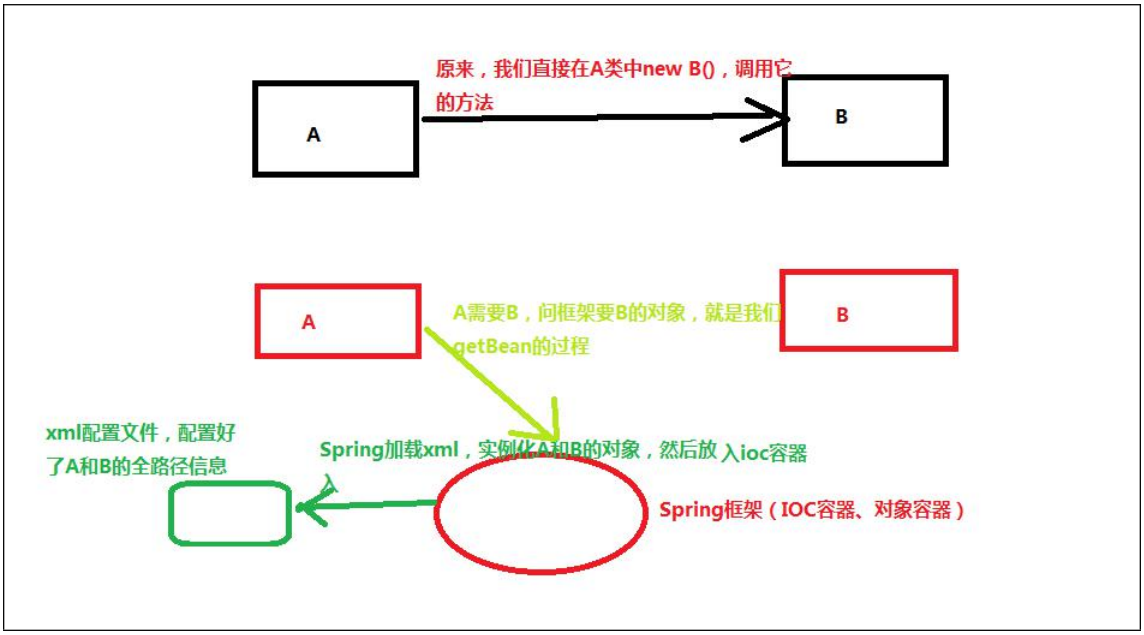
- 使用工厂模式解耦合，分析并完成工厂类的两大任务

```
package com.spring.factory;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;

import java.io.InputStream;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 产生对象的工厂
 * 任务一：读取解析xml，根据配置信息实例化对象，放入 map 集合（对象容器），等待使用
 * 任务二：提供一个从 map 中获取对象的方法（根据 id 获取对象）
 */
public class BeanFactory {

    // map 集合（对象容器，存储对象的）
    private static Map<String,Object> map = new HashMap<>();


    //任务一：读取解析xml，根据配置信息实例化对象，放入 map 集合（对象容器），等待使用
    static {
        // 读取解析 xml
        InputStream inputStream =
BeanFactory.class.getClassLoader().getResourceAsStream("beans.xml");
```

```
            SAXReader saxReader = new SAXReader();
            try {
                Document document = saxReader.read(inputStream);
                Element rootElement = document.getRootElement();
                List<Element> selectNodes = rootElement.selectNodes("//bean");
                if(selectNodes != null && selectNodes.size() > 0) {
                    for (int i = 0; i < selectNodes.size(); i++) {
                        Element element =    selectNodes.get(i);
                        String id = element.attributeValue("id");
                        String clazz = element.attributeValue("class");
                        Class aClass = Class.forName(clazz);
                        Object o = aClass.newInstance();
                        // 把反射生成的对象放入对象容器待用
                        map.put(id,o);
                    }
                }
            } catch (DocumentException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            }
        }



        //任务二：提供一个从 map 中获取对象的方法（根据 id 获取对象）
        public static Object getBean(String id) {
            return map.get(id);
        }

}
```

[掌握]使用 SpringIOC 实现程序解耦合

- 从 IOC 容器获取对象相比原有方式对比



- 引入 jar 坐标

```
<!-- 引入 spring 容器坐标-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

- service 层和 dao 层的业务代码完善
- 实例化 service 和 dao 代码，把它们的信息写在 spring 配置文件中

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

    <bean id="accountDao" class="com.spring.dao.AccountDaoImpl"/>
    <bean id="accountService" class="com.spring.service.AccountServiceImpl"/>
</beans>
```

- 启动容器，通过容器获取对象

```
import com.spring.dao.AccountDao;
import com.spring.service.AccountService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class MainTest {

    /**
     * 测试用例：使用 SpringIOC 获取对象
     */
    @Test
    public void testSpringIoc() {
        // 启动对象容器
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");
        // getBean
        AccountService accountService = (AccountService) applicationContext.getBean("accountService");
        AccountDao accountDao = (AccountDao) applicationContext.getBean("accountDao");
        //AccountService accountService1 = applicationContext.getBean(AccountServiceImpl.class);
        //AccountService accountService1 = applicationContext.getBean(AccountService.class);
        //applicationContext.getBean("accountService",AccountService.class);

        System.out.println(accountService);
        System.out.println(accountDao);
    }
}
```

[掌握]SpringIOC 的一些细节知识

ApplicationContext 和 BeanFactory

ApplicationContext 在启动 SpringIOC 容器的时候，会立即创建对象放入 map 中（**不存在影响性能的问题**）
BeanFactory 在启动 SpringIOC 容器的时候，不会立即创建对象，getBean 使用对象的时候，Spring 才去创建对象（类似于懒加载）

Bean 标签

- id：标识符，定位对象的。id 不能重复，没有 mybatis 中 namespace 之说了
  class ：所创建对象的全限定类名，类似于：com.spring.dao.AccountDaoImpl，不能

配置为接口，因为接口不能实例化

- scope ：对象的作用范围，单例和多例，单例：内存中这个类只有这一个对象，创建之后一直使用这一个；多例：当你 getBean 的时候，每 getBean 一次都给你创建一个新的对象

  singleton：单例/单个的意思，默认 bean 的 scope 都是单例

  prototype：多例的意思，每 getBean 一次就 new 一个新的对象返回

  request：在 web 应用的一个 request 请求内有效，下个请求内就重新 new 一个对象

  session：在 web 应用的一个 session 会话内有效，下个会话重新 new 一个对象
- 生命周期属性

  bean 即对象，对象就有生命周期，生命周期就有开始和结束，有时候希望在对象创建之后立马做点啥，或者销毁之前做事情

```
<bean id="accountDao" class="com.spring.dao.AccountDaoImpl" scope="sigle"
init-method="init1" destroy-method="detory1"></bean>
```

- 对象创建的三种方式
  - 通过配置 class 全限定类名框架底层使用反射技术来创建对象**(经常使用，推荐)**

```
<bean id="accountDao" class="com.spring.dao.AccountDaoImpl"></bean>
```

  - 静态方法**（自己 new 对象，然后加入到 Spring 的对象容器中管理）**

```
<!--bean产生方式：静态方法-->
<bean id="accountService" class="com.spring.factory.StaticFactory"
factory-method="createAccounts"></bean>
```

```
package com.spring.factory;

import com.spring.service.AccountService;
import com.spring.service.AccountServiceImpl;

public class StaticFactory {

    public static AccountService createAccountService() {
        return new AccountServiceImpl();
    }
}
```

  - 动态实例化方法（自己 new 对象，然后加入到 Spring 的对象容器中管理）（需要首先定义工厂的 bean，然后通过工厂 bean 对象调用里面的方法返回具体对象）

```
<!--bean产生方式：实例化方法 既然实例化方法，首先应该定义对象-->
<bean id="instanceFactory" class="com.spring.factory.InstanceFactory"></bean>
<bean id="accountService" factory-bean="instanceFactory" factory-method=
"createAccountService"></bean>
```

```
package com.spring.factory;
```

```java
import com.spring.service.AccountService;
import com.spring.service.AccountServiceImpl;

public class InstanceFactory {

    public AccountService createAccountService() {
        return new AccountServiceImpl();
    }
}
```

依赖注入（DI）

DI: dependence Inject （依赖注入），注入对象所依赖的对象

Set 注入(也叫设值注入，这是常用和推荐的用法)

```
<!--
    在bean的java类当中声明变量，然后添加变量的set方法（暴露给外界的传值接口）
    name：对应的是set方法setXXX中的XXX
    ref:当要传入的是一个bean对象的时候使用（已经定义的其他的bean的时候）
    value：当要传入的是普通数据（int、float、String）
-->
<!-- <property name="accountDao" ref="accountDao"/>

<property name="id" value="1"></property>
<property name="name" value="张三"></property>
<property name="money" value="10.123"></property>-->
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
">
```

构造函数注入

```
<constructor-arg name="id" value="4"/>
<constructor-arg name="name" value="王六"/>
<constructor-arg name="money" value="2.343"/>
<constructor-arg name="accountDao" ref="accountDao"/>
```

复杂对象注入

```
<property name="myStrs">
    <set>
        <value>mySet1</value>
        <value>mySet2</value>
        <value>mySet3</value>
    </set>
</property>

<property name="myList">
    <array>
        <value>myStr1</value>
        <value>myStr2</value>
        <value>myStr3</value>
    </array>
</property>

<property name="mySet">
    <list>
        <value>myList1</value>
        <value>myList2</value>
        <value>myList3</value>
    </list>
</property>

<property name="myMap">
    <props>
        <prop key="myProp1">propvalue1</prop>
        <prop key="myProp2">propvalue2</prop>
        <prop key="myProp3">propvalue3</prop>
    </props>

</property>

<property name="myProps">
    <map>
        <entry key="map1" value="mapvalue1"></entry>
        <entry key="map2" value="mapvalue2"></entry>
        <entry key="map3" value="mapvalue3"></entry>
    </map>
</property>
```

```
private String[] myStrs;
private List<String> myList;
private Set<String> mySet;
private Map<String,String> myMap;
private Properties myProps;
setter 方法…
```

[了解]扩展知识

Spring 中使用 BeanFactory 获取对象

```
@Test
public void testBeanFactory(){
    // 加载跟目录下的配置文件
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    Resource res = resolver.getResource("applicationContext.xml");
    // 创建 Bean 工厂类 注意 XmlBeanFactory 已经过期!
    BeanFactory factory = new XmlBeanFactory(res);
    AccountDao accountDAO = factory.getBean("accountDao", AccountDao.class);
    AccountService accountService = factory.getBean("accountService",
AccountService.class);
    System.out.println(accountDAO);
    System.out.println(accountService);
}
```

ccount 表的 CRUD 实现[掌握]

Dbutils 使用：
QueryRunner 对象的使用，提供了 query 和 update（新增、删除和修改）接口
BeanHandler 和 BeanListHandler 帮我们封装结果集数据

思路分析

- 从 Maven 坐标（jar）的角度分析
  - junit
  - mysql 驱动
  - dbutils 工具

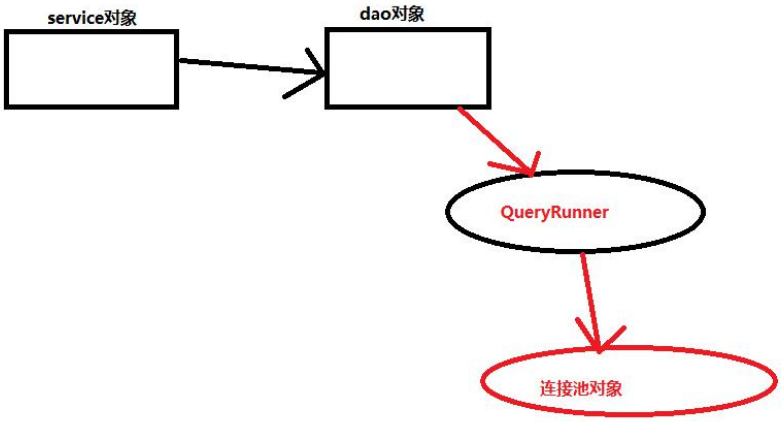
- c3p0/dbcp 数据库连接池

```
<!--dbcp连接池-->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.2.0</version>
</dependency>
```

Apache 提供的一个数据库连接池

- spring-context（容器功能，帮我们管理对象以及注入对象）

- 从开发步骤的角度分析
  - pom 文件引入坐标
  - pojo 类
  - dao 层开发
  - service 层开发
  - 使用 Spring-context 容器功能**管理涉及到的所有对象及其依赖注入关系**



每一个方块或者椭圆都是一个对象（需要定义 bean），然后维护他们的注入关系

xml 中配置如下

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


    <!--数据库连接池-->
    <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://localhost:3306/spring?characterEncoding=utf-8"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
```

```
    <!--queryrunner 对象-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"/>
    </bean>

    <!--dao 层对象-->
    <bean id="accountDao" class="com.spring.dao.AccountDaoImpl">
        <property name="queryRunner" ref="queryRunner"/>
    </bean>

    <!--service 层对象-->
    <bean id="accountService" class="com.spring.service.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"/>
    </bean>
</beans>
```

- DBCP 数据库连接池

```
/*c3p0连接池*/
ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();
comboPooledDataSource.setDriverClass("com.mysql.jdbc.Driver");
comboPooledDataSource.setJdbcUrl("jdbc:mysql://localhost:3306/spring?characterEncoding=ut
comboPooledDataSource.setUser("root");
comboPooledDataSource.setPassword("root");

// DBCP数据库连接池
BasicDataSource basicDataSource = new BasicDataSource();
basicDataSource.setDriverClassName("com.mysql.jdbc.Driver");
basicDataSource.setUrl("jdbc:mysql://localhost:3306/spring?characterEncoding=utf-8");
basicDataSource.setUsername("root");
basicDataSource.setPassword("root");
```

注意：所有的数据库连接池使用套路都一样，都是使用其中一个核心类，然后赋值必要的数据库连接四要素

[掌握]SpringIOC 的注解实现

原则：找 xml 中配置项和注解的对应
注意：使用注解不需要引入额外的 jar 包

- Bean 标签功能的注解实现方式

xml 形式	对应的注解形式
<bean>标签定义一个对象	**@Component("accountDao")** **bean 的 id 属性内容直接配置在注解后面**

	如果不配置，默认定义个这个 bean 的 id 为类的类名首字母小写 另外，针对分层代码开发 提供了@Componenet 的三种别名 @Controller、@Service、@Repository 分别用于控制层类、服务层类、dao 层类的 bean 定义 这四个注解的用法完全一样，只是为了更清晰的区分而已
<bean>标签的 scope 属性	@Scope("prototype")，默认单例
<bean>标签的 init-method 属性	@PostConstruct，注解加载方法上
<bean>标签的 destory-method 属性	@PreDestory，注解加载方法上

- 依赖注入的注解实现方式
  - @AutoWired 注解（Spring 框架提供的，推荐使用）

    按照类型注入

```
/*
* Autowired：按照类型自动注入
* 往往可以结合Qualifier注解进行精准注入（Qualifier可以指定id）
* */
@Autowired
@Qualifier("accountDao")
private AccountDao accountDao;
```

  - @Resource 注解（JDK 提供）

    按照 id 注入

```
@Resource(name="accountDao")
private AccountDao accountDao1;
```

    使用 Resource 注解的 name 属性来指定注入对象的 id
- 开启注解扫描开关

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

<context:component-scan bese-package="com.spring">
```

[掌握]Account 表 CRUD 的半 xml 半注解方式实现

企业开发使用最多的方式

- xml 配置文件依然存在，从配置文件启动容器
- 半 xml：指的是第三方 jar 中的对象的定义配置在 xml 中
- 半注解：自己开发的类定义 bean 使用注解方式

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

    <!--
        配置注解扫描
        base-package：配置所扫描包路径
    -->
    <context:component-scan base-package="com.spring"/>

    <!--数据库连接池-->
    <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property                                                    name="url"
value="jdbc:mysql://localhost:3306/spring?characterEncoding=utf-8"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>


    <!--queryrunner 对象-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"/>
    </bean>
</beans>
```

[掌握]Account 表 CRUD 的纯注解方式实现

纯注解方式（不需要引入额外的 jar 包）
其实是对半 xml 半注解模式的进一步改造

- **重点改造第三方 jar 中对象的定义（注解实现）**
- **容器初始化不是加载 xml 的方式了（纯注解不存在 xml 文件了）**

```
@Configuration //标识当前类是spring的一个配置类
@ConmponentScan(value={"com.spring"})
@PropertySource(value={"db.properties"})//@PropertySource注解用于加载属性配置文件
@Import(value={JdbcConfig.class})//@Import引入其他配置类
public class SpringConfig {

}
```

- @Configuration 注解，表名当前类是一个配置类
  @ComponentScan 注解，替代<context:component-scan>
  @ PropertySource,引入外部属性配置文件
  @Import 引入其他配置类
  @Value 对变量赋值，可以直接赋值，也可以使用${}读取资源配置文件中的信息
  @Bean 将方法返回对象加入 SpringIOC 容器

```
package com.spring.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.PropertySource;

@Configuration   // 标识当前类是 Spring 的一个配置类
@ComponentScan(value = {"com.spring"})
@PropertySource(value = {"db.properties"})   // @PropertySource 注解用于加载属性配置文件
@Import(value = {JdbcConfig.class})              // @Import 引入其他配置类
public class SpringConfig {
}
```

```
package com.spring.config;

import org.apache.commons.dbcp2.BasicDataSource;
import org.apache.commons.dbutils.QueryRunner;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
```

```java
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {

    //@Value 注解对变量赋值：1、可以直接赋值 2、可以使用${}获取属性文件配置项
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;



    /**
     *@Bean 注解加载方法上
     * 它可以把方法的返回值（返回值是一个对象），把这个对象加入到spring 的ioc 容器中
     *@return
     */
    @Bean("dataSource")
    public DataSource createDataSource() {
        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setDriverClassName(driver);
        basicDataSource.setUrl(url);
        basicDataSource.setUsername(username);
        basicDataSource.setPassword(password);
        return basicDataSource;
    }


    @Bean("queryRunner")
    public QueryRunner createQueryRunner(@Qualifier("dataSource") DataSource dataSource) {
        QueryRunner queryRunner = new QueryRunner(dataSource);
        return queryRunner;
    }
}
```

- **启动容器测试**

```
@Test
public void testQueryAccountById() throws Exception {
    // 启动 ioc 容器（全注解模式）
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(SpringConfig.class);
    AccountService accountService = (AccountService) applicationContext.getBean("accountService");
    Account account = accountService.queryAccountById(1);
    System.out.println(account);
}
```

注意：
使用 Spring 的时候，查找资源文件，咱们统一养成习惯加上 classpath：
classpath：和 classpath*：的区别，classpath 会去工程 classes 目录下查找（不包括 jar），classpath*会包括 jar 的所有的 classes 下去查找

[掌握]Spring 对 Junit 的支持

- 引入 spring-test 的 jar 包

```
<!--Spring 对 junit 测试的支持-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

- Spring 对 junit 支持所带来的好处
  - 1、配置之后不需要我们去启动容器
  - 2、测试哪个对象直接可以注入（使用注解）

  纯注解启动的形式

```
// 使用 Spring 自定义的 junit 运行器替换它默认的运行器
@RunWith(SpringJUnit4ClassRunner.class)
// 指定配置文件或者配置类
@ContextConfiguration(classes = {SpringConfig.class})
public class MainTest {

    @Autowired
```

```
    private AccountService accountService;

    @Test
    public void testQueryAccountById() throws Exception {
        Account account = accountService.queryAccountById(1);
        System.out.println(account);
    }
}
```

读取 xml 配置文件启动的形式

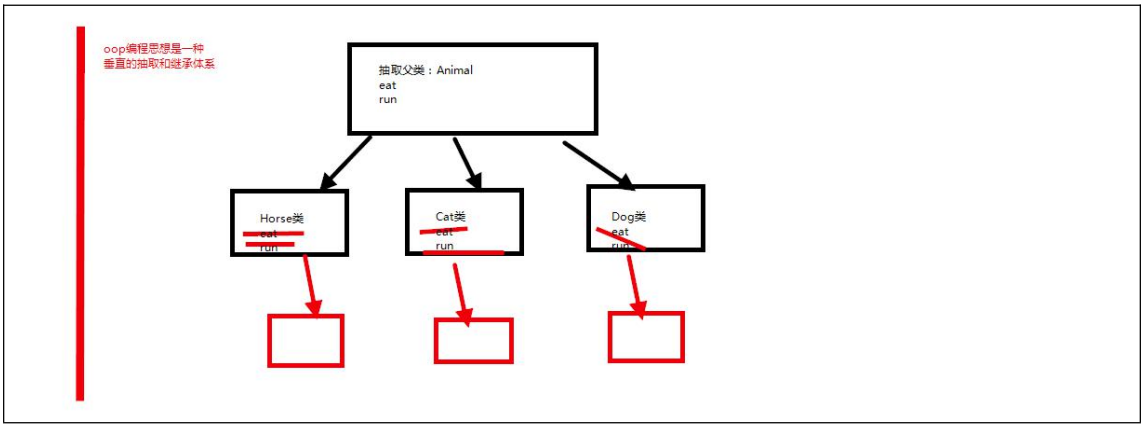
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:applicationContext.xml"})
public class MainTest {
```

最终，让我们非常方便的注入对象，测试对象

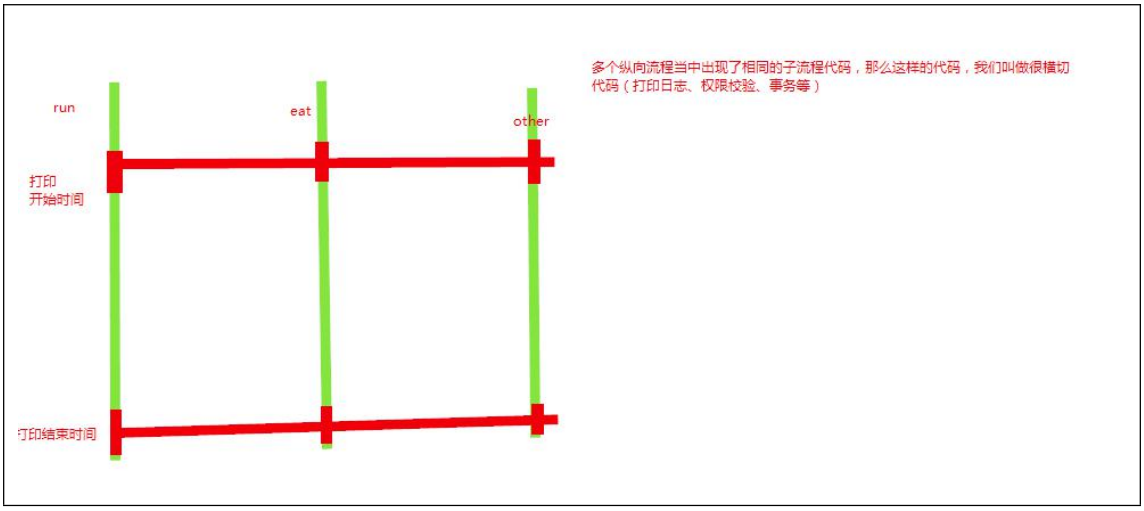
[理解]AOP 概述

什么是 AOP

AOP：Aspect Oriented Programming（面向切面编程/面向方面编程）
先从 OOP 说起（面向对象编程）
三大特征：封装、继承和多态
Horse 类、Cat 类、Dog 类，都有 eat()、run()方法，抽取父类（Animal 类），在父类中写上重复的代码，这样避免了代码重复问题



大多数情况下 OOP 编程都能够满足我们的需求，但是事情不是简单的，在一些场合使用 OOP 思想来解决代码重复问题，已经不行了



横切代码存在什么问题？

- 横切代码往往存在于多个方法中，是重复的
- 横切代码和业务代码混杂在一起，造成业务代码非常冗余，对于维护和编程都不是好现象

AOP 别出心裁的提出了横向抽取机制，把业务代码和横切代码分离，将业务代码和横切代码分离容易，但是最终终究要横切代码应用于业务代码，这是比较难的，也是 AOP 重点要解决的问题

即，不改变原有业务逻辑的基础上增强你的横切逻辑

AOP 的底层实现原理：动态代理技术；

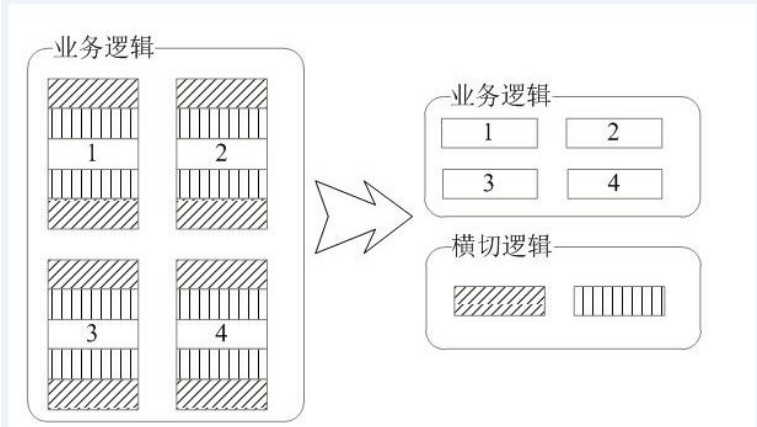


图 6-2　横向抽取

为什么叫 AOP（面向切面编程）

我们要做的事就是在不改变原有业务逻辑的基础上增强我们的横切逻辑，业务逻辑不能改，我们只能操作横切逻辑（切的意思所在，我们只能面向横切逻辑），AOP 要增强的横切逻辑往往影响的不是一个方法，往往很多方法，每一个方法如同一个散点，很多散点组成面（这是面向切面编程中“面”的意思所在），这就是为什么叫做面向切面编程

AOP 的实现方式

Spring 的 AOP 的实现方式就是动态代理技术

AOP 的作用及优势

- 作用：程序运行期间，在不改变原有业务逻辑的情况下进行方法增强
- 优势：减少重复代码，提高开发效率，业务逻辑和增强的横切逻辑分离便于维护

[理解]AOP 的具体应用

模拟转账

- aaa 账户向 bbb 账户转钱
- aaa 账户减钱，bbb 账户加钱（转账业务）
  - 查找到 aaa、bbb 账户（根据账户名查找账户）
  - 进行转账计算
  - 更新 aaa、bbb 账户信息到数据库（根据 id 更新账户）

案例：模拟转账（并且模拟转账异常）

- 汇款人账户减少一定的金额
- 收款人账户增加一定的金额
- 计算之后，更新数据库

问题：模拟转账异常（人为制造异常，在两次 update 之间造了异常）

```
/**
 * 转账思路分析：
 * 1、根据账户名查询出汇款人账户和收款人账户
 * 2、计算转账金额：汇款人—money，收款人+money
 * 3、把上述计算后的数据更新到数据库
 * @param fromName  汇款人
 * @param toName    收款人
 * @param money     转账金额
 * @throws Exception
 */
@Override
public void transfer(String fromName, String toName, Float money) throws Exception {
    Account fromAccount = accountDao.queryAccountByName(fromName);
    Account toAccount = accountDao.queryAccountByName(toName);
```

```
        // 计算转账
        fromAccount.setMoney(fromAccount.getMoney() – money);
        toAccount.setMoney(toAccount.getMoney() + money);

        // 更新到数据库
        accountDao.updateAccountById(fromAccount);
        // 人为制造的异常
        int c = 1/0;
        accountDao.updateAccountById(toAccount);
}
```

问题分析：

- 两次 update 走了两个事务，归根结底是两次 update 使用了两个不同的连接
- 当前事务在 dao 层，因为你调用了两次 dao，两次又走了不同的事务

解决方案：

- 两次 update 使用同一个连接（两次 update 有一个关系：就是在一个线程中，这样把连接绑定到当前线程）
- 事务控制在 service 层

创建 ConnectionUtil 类

```
package com.spring.utils;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

public class ConnectionUtil {

    // 存放和当前线程绑定的连接
    private ThreadLocal<Connection> threadLocal = new ThreadLocal<>();

    private DataSource dataSource;

    /**
     * 返回当前线程绑定的连接对象
     * @return
     */
    public Connection getCurrentThreadConn() throws SQLException {
        Connection connection = threadLocal.get();
```

```
            if(connection == null) {
                connection = dataSource.getConnection();
                threadLocal.set(connection);
            }
            return connection;
        }


        /**
         * 解除线程和连接的绑定关系
         */
        public void remove() {
            threadLocal.remove();
        }
}
```

```
package com.spring.dao;

import com.spring.pojo.Account;
import com.spring.utils.ConnectionUtil;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Repository;

@Repository("accountDao")
public class AccountDaoImpl implements AccountDao {

    @Autowired
    @Qualifier("queryRunner")
    private QueryRunner queryRunner;

    private ConnectionUtil connectionUtil;

    @Override
    public Account queryAccountByName(String name) throws Exception {
        String sql = "select id,name,money from account where name=?";
        return queryRunner.query(connectionUtil.getCurrentThreadConn(),sql,new
BeanHandler<Account>(Account.class),name);
    }

    @Override
    public int updateAccountById(Account account) throws Exception {
```

```
            String sql = "update account set name=?,money=? where id=?";
            return
queryRunner.update(connectionUtil.getCurrentThreadConn(),sql,account.getName(),account.ge
tMoney(),account.getId());
        }
}
```

- 事务应该加载 service 业务层上，而且应该由我们自己去控制（关闭事务自动提交）

```
package com.spring.utils;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

/**
 * 事务管理器：开启事务、提交事务、回滚事务等
 */
@Component("transactionManager")
public class TransactionManager {

    @Autowired
    @Qualifier("connectionUtil")
    private ConnectionUtil connectionUtil;


    /**
     * 开启事务（就是关闭连接的事务自动提交功能）
     */
    public void beginTransaction() throws Exception {
        connectionUtil.getCurrentThreadConn().setAutoCommit(false);
    }


    /**
     * 提交事务
     * @throws Exception
     */
    public void commit() throws Exception{
        connectionUtil.getCurrentThreadConn().commit();
    }


    /**
```

```
     * 回滚事务
     *@throws Exception
     */
    public void rollback() throws Exception{
        connectionUtil.getCurrentThreadConn().rollback();
    }


    /**
     * 释放资源
     *@throws Exception
     */
    public void release() throws Exception{
        // 关闭连接
        connectionUtil.getCurrentThreadConn().close();
        // 解除绑定关系
        connectionUtil.remove();
    }
}
```

```
@Override
public void transfer(String fromName, String toName, Float money) throws Exception {
    try{
        // 开启事务（把事务的自动提交关闭即可）
        transactionManager.beginTransaction();

        // 业务逻辑代码

        Account fromAccount = accountDao.queryAccountByName(fromName);
        Account toAccount = accountDao.queryAccountByName(toName);

        // 计算转账
        fromAccount.setMoney(fromAccount.getMoney() - money);
        toAccount.setMoney(toAccount.getMoney() + money);

        // 更新到数据库
        accountDao.updateAccountById(fromAccount);
        // 人为制造的异常
        int c = 1/0;
        accountDao.updateAccountById(toAccount);

        // 提交事务
        transactionManager.commit();
    }catch (Exception e) {
```

```
            e.printStackTrace();
            // 回滚事务
            transactionManager.rollback();
        }finally {
            // 释放资源
            transactionManager.release();
        }

}
```

我们不可能在每一个方法中添加 try…catch…finally 控制，因此引入动态代理技术来在不改变原有业务逻辑代码的基础上做逻辑增强

```
package com.spring.factory;

import com.spring.utils.TransactionManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 代理对象工厂
 */
@Component("proxyFactory")
public class ProxyFactory {

    @Autowired
    @Qualifier("transactionManager")
    private TransactionManager transactionManager;

    public Object getProxy(Object object) {
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
object.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                Object result = null;

                try{
                    // 开启事务（把事务的自动提交关闭即可）
                    transactionManager.beginTransaction();
```

```
                    // 业务逻辑代码
                    result = method.invoke(object,args);
                    // 提交事务
                    transactionManager.commit();
                }catch (Exception e) {
                    e.printStackTrace();
                    // 回滚事务
                    transactionManager.rollback();
                }finally {
                    // 释放资源
                    transactionManager.release();
                }
                return result;
            }
        });
    }
}
```

[掌握]Spring 中的 AOP 使用

AOP 相关术语

- Joinpoint（连接点） （程序中所有方法的方法前、方法后、异常时等时机都是连接点）

  横切程序执行的特定位置，比如类开始初始化前，类初始化之后，类中某个方法调用前、调用后，方法抛出异常后（时机）等，这些代码中的特定点就称为“连接点”。

  Spring 仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时以及方法调用前后这些程序执行点织入增强。

  我们知道黑客攻击系统需要找到突破口，没有突破口就无法进行攻击，从这一角度上来说，AOP 是一个黑客（因为它要向目前类中嵌入额外的代码逻辑），连接点就是 AOP 向目标类打入楔子的**候选点**。
- Pointcut（切入点）（定位你感兴趣的方法）

  一个类中可以有很多个方法，每个方法又有多个 Joinpoint，在这么多个方法中，如何定位到自己感兴趣的方法呢？靠的是切点

  注意：切点只定位到某个方法上，所以如果希望定位到具体连接点上，还需要提供方位信息

  比如：如果把一个方法理解成数据表中的一条记录的话，那么切入点就好比你 select 语句的 where 条件，就可以定位到你感兴趣的方法
- Advice（通知/增强）

  增强的**第一层意思就是你的横切逻辑代码**（增强逻辑代码）

  在 Spring 中，增强除了用于描述横切逻辑外，包含一层意思就是横切逻辑执行的

方位信息。

刚刚说了切点只能定位到方法，在进一步使用方位信息就可以定位到我们感兴趣的连接点了（方法调用前、方法调用后还是方法抛出异常时等）。

- Target（目标对象）即委托对象

  增强逻辑的织入目标类。比如未添加任何事务控制的 AccountServiceImplNoTcf 类

- Weaving（织入） 形象的描述了增强横切逻辑的整个过程

  织入是将增强逻辑/横切逻辑添加到目标类具体连接点上的过程，AOP 像一台织布机，将目标类、增强或者引介通过 AOP（其实就是动态代理技术）这台织布机天衣无缝地编织到一起。

  Spring 采用动态代理织入。

- Proxy（代理）代理对象

  一个类被 AOP 织入增强后，就产出了一个结果类，它是融合了原类和增强逻辑的代理类。

- Aspect（切面）

  切面由切点和增强组成。

  切面=切点+增强

  =切点+方位信息+横切逻辑

  =连接点+横切逻辑

  最终切面完成：把横切逻辑织入到哪些方法的方法前/后等

  本质：把横切逻辑增强到连接点（切点和方位信息都是为了确定连接点）上

Spring 关于 JDK/CGLIB 动态代理的选择

Spring 发现涉及到接口那就使用 JDK 动态代理，如果不涉及接口就使用 CGLIB 动态代理

AOP:日志、性能监控、事务、权限控制

基于 XML 的 AOP 配置

- 需求：在 Service 层代码的不同方法的不同连接点 JoinPoint 织入日志

  把 Account 表的 service 层进行 crud 模拟（dao 层就不需要了）

- 引入 POM 坐标

```
<!--使用 Spring 的 AOP 功能-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<!--SpringAop 功能依赖的 jar-->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.9</version>
```

```
</dependency>
```

- applicationContext.xml 配置
- 切入点表达式说明（切入点表达式语法使用的是 AspectJ 框架的表达式语法）

```
execution(表达式)
表达式语法：execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

- XML 配置及四种常用 Advice 通知类型（即四种方位）
  - aop:before ：配置前置通知/增强，指定增强的方法在切入点方法之前执行
  - aop:after-returning：配置后置通知，切入点方法正常执行之后，相当于图中位置
    此处等同于 try{ …}catch(){}finally{}，红色点标注的时机
  - aop:after-throwing：配置异常通知，切入点方法执行产生异常后执行，它和后置通知只能执行一个，等同于 catch 中的时机
  - aop:after：配置最终通知，无论切入点方法执行时是否有异常，它都会在其后面执行，**等同于 finally 中的执行时机**
- 环绕通知：它是 spring 为我们提供的一种可以在代码中手动控制增强方法何时执行的方式，灵活度比较高，**设置可以控制原业务逻辑是否执行,其他四种通知类型的原有业务逻辑，肯定执行！**

注意：通常情况下，环绕通知都是独立使用的，不要和上面的四种通知类型混合使用

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

    <context:component-scan base-package="com.spring"/>



    <!--配置 AOP 内容都放置在 aop:config 标签下-->
    <!--横切逻辑 bean-->
    <bean id="logUtil" class="com.spring.utils.LogUtil"/>
    <aop:config>
```

```
        <!--aop 主要就是配置切面，所以 aop:aspect
            切面= 切点+方位信息+横切逻辑
            ref：指向横切逻辑
            <aop:before>等标签已经表明方位信息
            切点：使用 pointcut 进行定义
        -->
        <aop:aspect id="logAspect" ref="logUtil">
            <!--aop:pointcut 单独配置切入点表达式，供引用-->
            <aop:pointcut id="pt1" expression="execution(* com.spring.service.*.*(..))"/>
            <!--<aop:before method="printBeforeMethod" pointcut-ref="pt1"/>
            <aop:after-returning method="printAfterReturn" pointcut-ref="pt1"/>
            <aop:after-throwing method="printAfterThrowing" pointcut-ref="pt1"/>
            <aop:after method="printAfterMethod" pointcut-ref="pt1"/>-->

            <aop:around method="printRound" pointcut-ref="pt1"/>
        </aop:aspect>
    </aop:config>
</beans>
```

基于注解的 AOP 配置

- 半 xml 半注解形式
  - 在 xml 配置文件中开启 AOP 注解开关

```
<!--aop注解驱动的开关-->
<aop:aspectj-autoproxy/>
```

  - 在横切逻辑类上配置

```
package com.spring.utils;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LogUtil {

    @Pointcut("execution(* com.spring.service.*.*(..))")
    public void pt1() {
```

```
    }

/*   *//**
     * 方法执行之前打印
     *//*
    @Before("pt1()")
    public void printBeforeMethod() {
        System.out.println("方法执行之前打印");
    }


    *//**
     * 方法结束时打印
     *//*
    @After("pt1()")
    public void printAfterMethod() {
        System.out.println("方法执行之后（结束）打印");
    }


    *//**
     * 方法异常时打印
     *//*
    @AfterThrowing("pt1()")
    public void printAfterThrowing() {
        System.out.println("方法异常时打印");
    }


    *//**
     * 方法正常执行（没有抛出异常）时打印
     *//*
    @AfterReturning("pt1()")
    public void printAfterReturn() {
        System.out.println("方法正常执行（返回）打印");
    }*/


    /**
     * 环绕通知，灵活度比较高，可以手动控制在业务逻辑之前、之后、异常时等连接点执行逻辑
     * 也可以控制原有业务逻辑是否执行，通过直接在参数中声明形参 ProceedingJoinPoint 即可，类似于动态代理中的 method.invoke()
     * @param proceedingJoinPoint
```

```
     * @return
     */
    @Around("pt1()")
    public Object printRound(ProceedingJoinPoint proceedingJoinPoint) {

        Object result = null;
        try {
            System.out.println("环绕通知的之前执行");
            result = proceedingJoinPoint.proceed();
            System.out.println("环绕通知的正常执行时执行");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
            System.out.println("环绕通知的异常执行时执行");
        }
        System.out.println("环绕通知方法结束时执行");
        return result;
    }
}
```

- 全注解模式下注解配置开关

```
@Configuration //标识当前类是 spring 的一个配置类
@ConmponentScan(value={"com.spring"})
@EnableAspectJAutoProxy
public class SpringConfig {
}
```

学习 spring 中的 AOP 时要明确的事

- AOP 的使用场景
  AOP 的应用场景往往是受限的，它一般只适合于那些具有横切逻辑的应用场合：如性能监测、访问控制、事务管理以及日志记录等。不过，这丝毫不影响 AOP 作为一种新的软件开发思想在软件开发领域所占有的地位。
- 开发阶段（我们完成）
  - 编写核心业务代码
    大部分程序员来做，要求熟悉业务需求。
  - 抽取公用代码制作成通知，进行 AOP 配置
    一般由专门的 AOP 编程人员来做
- 运行阶段（Spring 框架完成）
  Spring 框架监控切入点方法的执行。一旦监控到切入点方法被运行，Spring 框架使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻辑运行。

[应用]JdbcTemplate 数据库操作工具使用

Spring 框架提供的一个数据库操作工具，使用方法和 Dbutils 非常相似

Dbutils	JdbcTemplate
核心对象 QueryRunner	核心对象 JdbcTemplate
update 接口（增、删、改）	update 接口（增、删、改），操作过程和方式和 dbutils 一样
query 接口查询	queryForObject（单个值/单个对象）
	query 接口（返回集合）

Dbutils 返回单个对象使用 BeanHandler 封装数据，返回 list 集合使用 BeanListHandler 封装数据。BeanHandler 和 BeanListHandler 都是 ResultsetHandler 接口的实现类

在 JdbcTemplate 中，封装数据也提供了接口 RowMapper，但是没有给你实现类，只能我们自己去实现这个接口，告诉工具结果集封装规则

JdbcTemplate 入门使用

- 需求：使用 JdbcTemplate 工具完成 Account 表的 Crud
- 操作步骤
  - 引入 jar 包坐标

```
<!--jdbcTemplate 使用的jar-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<!--事务相关的jar，往往和jdbc 一起引入-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

- CRUD 程序

```
import com.spring.pojo.Account;
import org.junit.Before;
import org.junit.Test;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
```

```
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

public class JdbcTemplateTest {

    private JdbcTemplate jdbcTemplate;

    @Before
    public void before() {
        // 数据库连接池（使用 Spring 内置的数据库连接池）
        DriverManagerDataSource driverManagerDataSource = new
DriverManagerDataSource();
        driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver");

driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/spring?characterEncodin
g=utf-8");
        driverManagerDataSource.setUsername("root");
        driverManagerDataSource.setPassword("root");


        // 实例化 JdbcTemplate 工具的核心对象
        jdbcTemplate = new JdbcTemplate(driverManagerDataSource);
        // jdbcTemplate.setDataSource(driverManagerDataSource);
    }



    /**
     * jdbcTemplate 工具之 insert 操作
     */
    @Test
    public void testSaveAccount() {
        String sql = "insert into account(name,money) values(?,?)";
        int   count = jdbcTemplate.update(sql,"柳岩",100f);
    }

    /**
     * jdbcTemplate 工具之 update 操作
     */
    @Test
    public void testUpdateAccountById() {
```

```
        String sql = "update account set name=?,money=? where id=?";
        int count = jdbcTemplate.update(sql,"唐嫣",1000f,4);
        System.out.println(count);
    }


    /**
     * jdbcTemplate 工具之 delete 操作
     */
    @Test
    public void testDeleteAccountById() {
        String sql = "delete from account where id=?";
        int count = jdbcTemplate.update(sql,4);
        System.out.println(count);
    }


    /**
     * jdbcTemplate 工具之查询单个值（总记录数）
     */
    @Test
    public void testQueryAccountCount() {
        String sql = "select count(id) from account";
        /*
           返回单个值使用 queryForObject 接口
           param1：sql 语句
           param2：返回值的数据类型
        */
        long count = jdbcTemplate.queryForObject(sql,long.class);
        System.out.println("account 表总记录数： " + count);
    }


    /**
     * jdbcTemplate 工具之查询单个值（字符串类型）（使用 queryForObject 接口）
     */
    @Test
    public void testQueryAccountNameById() {
        String sql = "select name from account where id=?";
        String name = jdbcTemplate.queryForObject(sql,String.class,1);
        System.out.println(name);
    }

    /**
```

```
     * jdbcTemplate 工具之查询单个对象（使用 queryForObject 接口）
     */
    @Test
    public void testQueryAccountById() {
        String sql = "select id,name,money from account where id=?";
        Account account = jdbcTemplate.queryForObject(sql, new AccountRowMapper(),
1);

        System.out.println(account);
    }


    /**
     * jdbcTemplate 工具之查询返回对象集合（使用 query 接口）
     */
    @Test
    public void testQueryAccountList() {
        String sql = "select id,name,money from account";
        List<Account> list = jdbcTemplate.query(sql, new AccountRowMapper());
        if(list != null && list.size() > 0) {
            for (int i = 0; i < list.size(); i++) {
                Account account =   list.get(i);
                System.out.println(account);
            }
        }
    }

    /**
     * AccountRowMapper 内部类是对 RowMapper 接口的实现
     *  在查询单个对象或者对象结合的时候封装数据使用
     */
    public class AccountRowMapper implements RowMapper<Account>{
        @Override
        public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
            Account account = new Account();
            account.setId(rs.getInt("id"));
            account.setName(rs.getString("name"));
            account.setMoney(rs.getFloat("money"));
            return account;
        }
    }
}
```

JdbcTemplate 开发 Dao 层的两种方式

- 需求：开发 service 层和 dao 层，service 层调用 dao 层，在 dao 层中使用 JdbcTemplate 实现 Account 表的 Crud（半 xml 半注解形式实现）

方式一：定义 JdbcTemplate 为普通 Bean

普通玩法：和 Dbutils 一样，在 Spring 配置文件中定义 JdbcTmeplate 的 bean 对象

```
<!--JdbcTemplate 工具-->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

dao 层使用 JdbcTmeplate 对象，直接注入使用

```
package com.spring.dao;

import com.spring.pojo.Account;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void saveAccount(Account account) throws Exception {
        String sql = "insert into account(name,money) values (?,?)";
        jdbcTemplate.update(sql,account.getName(),account.getMoney());
    }

    @Override
    public void updateAccountById(Account account) throws Exception {
        String sql = "update account set name=?,money=? where id=?";
        jdbcTemplate.update(sql,account.getName(),account.getMoney(),account.getId());
    }
```

```
    @Override
    public void deleteAccountById(Integer id) throws Exception {
        String sql = "delete from account where id=?";
        jdbcTemplate.update(sql,id);
    }

    @Override
    public Account queryAccountById(Integer id) throws Exception {
        String sql = "select id,name,money from account where id=?";
        return jdbcTemplate.queryForObject(sql,new AccountRowMapper(),id);
    }

    @Override
    public List<Account> queryAccountList() throws Exception {
        String sql = "select id,name,money from account";
        return jdbcTemplate.query(sql,new AccountRowMapper());
    }


    public class AccountRowMapper implements RowMapper<Account>{
        @Override
        public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
            Account account = new Account();
            account.setId(rs.getInt("id"));
            account.setName(rs.getString("name"));
            account.setMoney(rs.getFloat("money"));
            return account;
        }
    }
}
```

方式二：继承 JdbcDaoSupport 类

玩法二和玩法一的区别在于：dao 层实现类中 jdbcTemplate 对象的获取方式

Dao 层实现类直接继承 JdbcDaoSupport 父类，从父类中获取 JdbcTemplate 对象

注意：**推荐使用方式一**，继承 JdbcDaoSupport 类的方式只支持 xml 配置，不支持注解方式

必须在 xml 中定义 dao 实现类对象（不能直接在 dao 实现类上添加@Repository 注解）

```
<!--继承 JdbcDaoSupport 的方式-->
    <bean id="accountDao" class="com.spring.dao.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"/>
```

```
    </bean>
```

dao 实现类中继承 JdbcDaoSupport 父类

```
/**
 * Dao层实现类直接继承JdbcDaoSupport父类，从父类中获取JdbcTemplate对象
 */
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao
```

Spring 内置数据源

```
<!--使用Spring内置的数据库连接池，该连接池在Spring-jdbc这个jar中-->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

[理解掌握]Spring 的声明式事务控制

- 关于编程式事务和声明式事务
  编程式事务：在业务代码中添加事务控制代码，这样的事务控制机制，叫做编程式事务
  声明式事务：通过 xml 配置或者注解方式就达到事务控制的目的，叫做声明式事务

关于事务

- 事务基本特性（ACID，是针对单个事务的一个完美状态）
  - 原子性：一个事务内的操作要成功都成功，要失败都失败。比如转账案例，转账和到账要么同时成功要么同时失败。
  - 一致性：一致性和原子性描述的是同一件事，只不过角度不一样。原子性是从一个事务内的操作的角度来说的，要成都成要失败都失败。一致性是从数据的角度来说的，比如转账，转账的初始数据状态（1000,1000），现在转 100 块，转账完成后，对外来说，数据状态要么（900,1100），要么是（1000,1000），不能够出现（900，1000）等中间状态
  - 隔离性：比如事务 1 给员工涨工资 2000 元，但是事务 1 尚未提交事务，事务 2 查询工资，发现工资涨了 2000 块。这就是脏读（读到了未提交的数据）
  - 持久性：事务一旦提交即生效，即使数据库服务器宕机，那么重启之后数据也应该是事务提交之后的状态，不会是之前的状态
- 事务并发问题
  - 脏读
    - 场景

财务人员发起事务 1，给员工 A 涨了 1w 块钱，但是尚未提交事务。此时员工 A 发起了事务 2 查询工资，发现涨了 1w 块钱。

- 幻读（针对 insert 和 delete 操作）
  - 场景
    - 事务 1 查询所有工资为 1w 的员工的总数，查出来 10 个，此时事务尚未关闭。
    - 事务 2，由财务人员发起，新来两个员工，工资也是 1w，向数据表中插入了两条数据，并且提交事务
    - 事务 1 再次查询工资为 1w 的员工个数，发现有 12 个。见鬼了
- 不可重复读（针对 update 操作）
  - 场景
    - 员工 A 发起事务 1，查询工资，工资为 1w 块钱，此时事务尚未关闭
    - 财务人员发起了事务 2 给员工 A 涨了 2000 块钱，并且提交事务
    - 员工 A 通过事务 1 再次查询工资，发现工资为 1.2w，原来的 1w 那个数据读不到了，叫做不可重复读。

- 事务隔离级别（解决是事务并发问题的）
  - 读未提交（极端）：Read Uncommited，读到了未提交的数据，什么并发问题也解决不了。**不要采取这种方案**
  - 读已提交：Read Committed：解决脏读问题，解决不了幻读和不可重复读的问题（因为幻读和不可重复读问题的造成，本身就是已提交事务造成的）
  - 可重复读：Read Repeatable，解决脏读和不可重复读的问题。解决不了幻读问题，因为可重复读针对的是 update 操作，在可重复读机制下，会对要 update 的语句进行加锁，加锁状态下其他事务无法修改。但是其他事务可以对表新增和删除
  - 串行化（极端）：Serializable，事务一个个来，是最安全的隔离机制，但是效率较低，比如 ATM 机。
  - 默认，数据库的默认，mysql 默认的隔离机制是可重复读，Oracle 的默认隔离机制是读已提交
- 事务传播行为

  事务往往在 service 层进行控制，如果出现 service 层方法 A 调用了另外一个 service 层方法 B，A 和 B 方法本身都已经被添加了事务控制，那么 A 调用 B 的时候，就需要进行事务的一些协商，这就叫做事务的传播行为。

  A 调用 B，我们站在 B 的角度来观察来定义事务的传播行为

```
REQUIRED:如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值）
SUPPORTS:支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）
MANDATORY：使用当前的事务，如果当前没有事务，就抛出异常
REQUERS_NEW:新建事务，如果当前在事务中，把当前事务挂起。
NOT_SUPPORTED:以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
NEVER:以非事务方式运行，如果当前存在事务，抛出异常
NESTED:如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行 REQUIRED 类似的操作。
```

Spring 声明式事务配置

pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.9</version>
</dependency>
```

基于 XML

Spring 包容性很强，对 dao 层很多技术都会提供支持

jdbc：connection.commit()

mybatis：sqlSession.commit()

hibernate：session.commit();

Spring 针对上述众多的事务调用方式，为了一统江湖，定义顶层接口（开启事务、提交事务、回滚事务等）

针对不同的技术体系，封装不同的事务管理实现类

事务管理器类，就是关于事务的横切逻辑，我们使用 jdbc/jdbcTemplate/mybatis 使用的是 Spring 提供的 DataSourceTransactionMannager 事务管理类，这个类就如同自定义 AOP 中开发的 TransactionMannager 类

事务控制本质：将横切逻辑（DataSourceTransactionMannager，此处的横切逻辑就是事务的控制逻辑）增强到感兴趣的方法上，进行事务控制

基于注解

- 半注解半 xml 形式

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

```
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
">

    <context:component-scan base-package="com.spring"/>

    <context:property-placeholder location="classpath:db.properties"/>


    <!--使用 Spring 内置的数据库连接池，该连接池在 Spring-jdbc 这个 jar 中-->
    <bean                                                    id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!--定义 jdbcTemplate 对象-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>


    <!--Spring 的声明式事务是基于 AOP 技术实现
        横切逻辑单独定义为一个 bean
        事务管理器底层还需要数据库连接池的支持
    -->
    <bean                                                    id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>


    <tx:advice id="txAdvice" transaction-manager="transactionManager">

        <!--tx:attributes 允许我们对事务细节进行配置-->
        <tx:attributes>
            <!--CRUD 中只有查询是只读事务，其他都是非只读事务
                所以在这里可以首先配置所有方法都是非只读事务
                然后单独配置查询方法为只读事务，覆盖上面的配置
                这样，简化配置量
                read-only：是否只读
```

```
                    propagation：事务传播行为
                    isolation：事务隔离级别，默认 defalut
                    timeout：超时时间，-1 没有限制，单位秒
                    rollback-for：出现什么异常的时候回滚事务，往往不需要配置，有异常就回滚
                    no-rollback-for：出现什么异常的时候不回滚事务，往往不需要配置，有异常就回滚
                -->
                <tx:method name="*" read-only="false" propagation="REQUIRED" isolation="DEFAULT" timeout="-1" />
                <!--查询操作为只读事务即可，事务传播行为为SUPPORTS，有事务就用，没有就不用-->
                <tx:method name="query*" read-only="true" propagation="SUPPORTS"/>
            </tx:attributes>
        </tx:advice>


        <aop:config>
            <!--Spring 声明式事务使用 aop:advisor 替代 aop 中的 aop:aspect
                至于方位信息不需要指定了（对于事务来说）
            -->
            <aop:advisor advice-ref="txAdvice" pointcut="execution(* com.spring.service.*.*(..))"/>
        </aop:config>
</beans>
```

- 全注解形式开启开关

```
@Configuration //标识当前类是 spring 的一个配置类
@PropertySource({"classpath:db.properties"})
@EnableTransactionManagement
public class SpringConfig {
}
```