

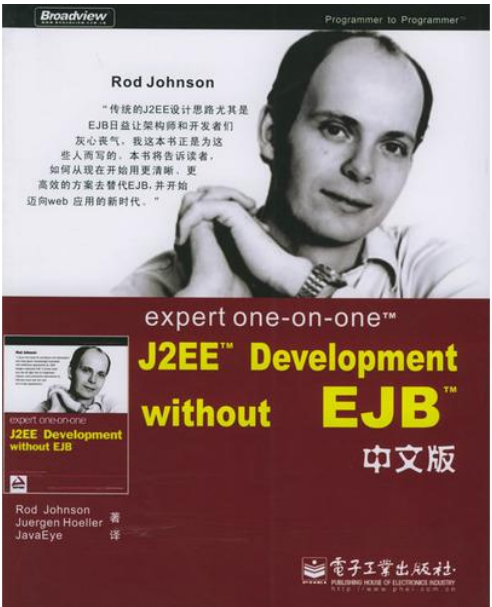


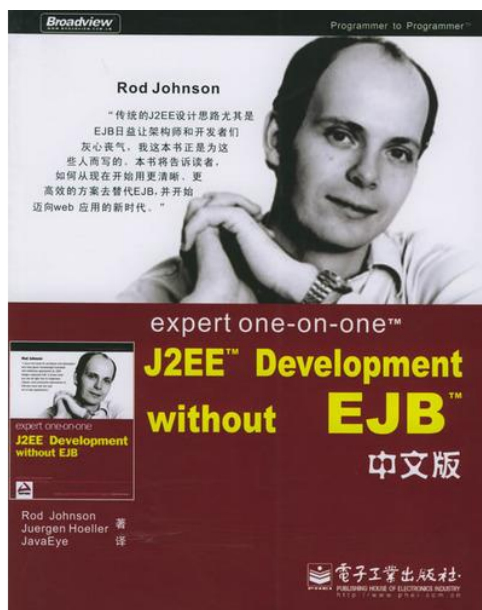
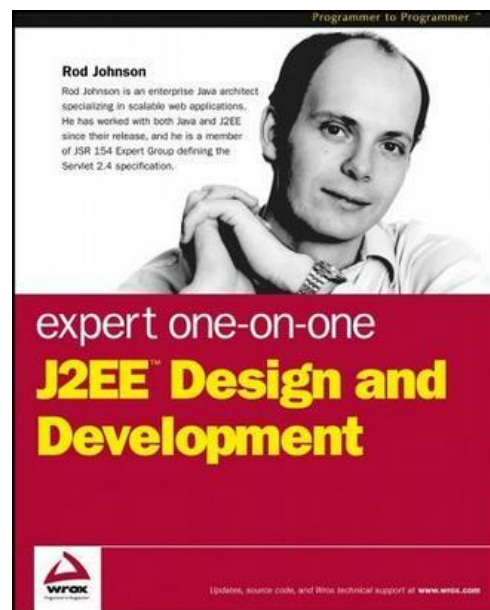
Spring 4.x

凯盛软件

推荐书籍

凯盛软件





- 官网
 - <https://spring.io/>
- 博客
 - <https://spring.io/blog>
- GitHub
 - <https://github.com/spring-projects>

<http://www.mkyong.com/tutorials/spring-tutorials/>

Spring Framework 是一个开源的Java / Java EE全功能栈 (full-stack) 的应用程序框架，以Apache许可证形式发布。

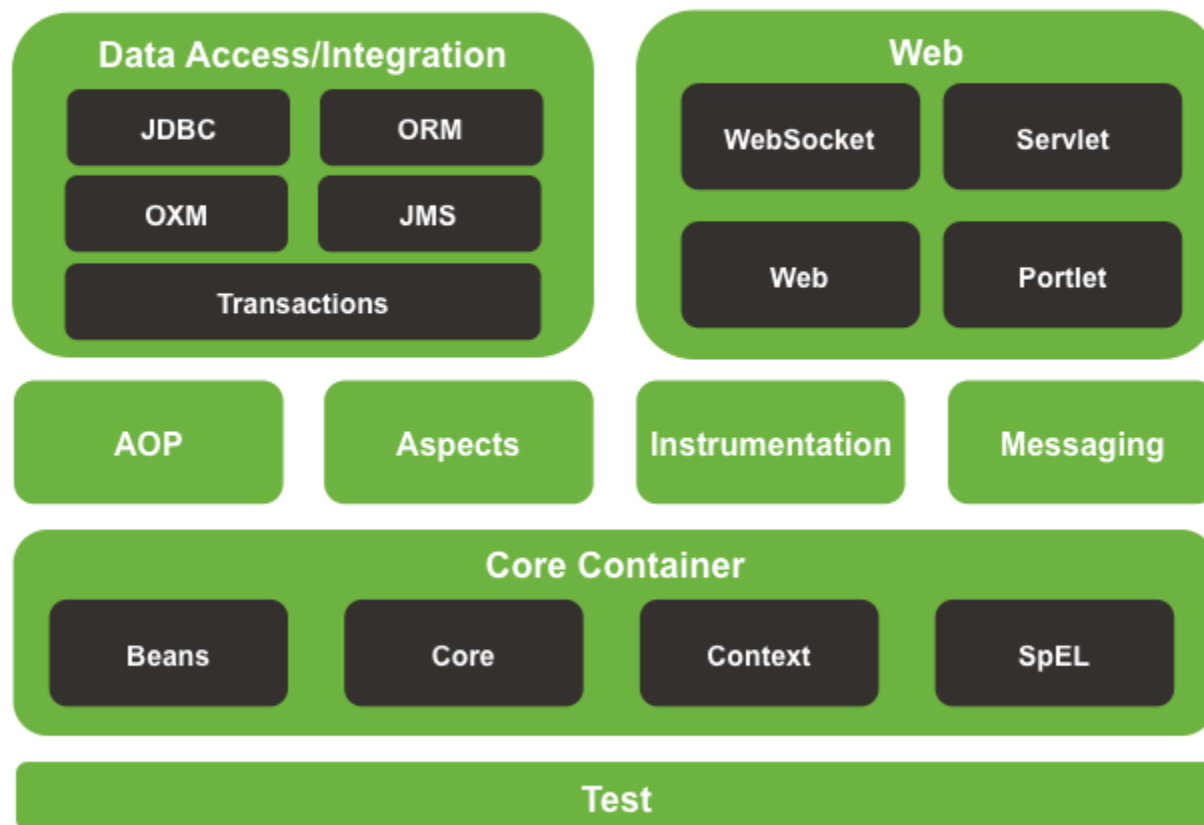
该框架基于 Expert One-on-One Java EE Design and Development (ISBN 0-7645-4385-7) 一书中的代码，最初由Rod Johnson和Juergen Hoeller等开发。

Spring Framework提供了一个简易的开发方式，这种开发方式，将避免那些可能致使底层代码变得繁杂混乱的大量的属性文件和帮助类。

https://zh.wikipedia.org/wiki/Spring_Framework



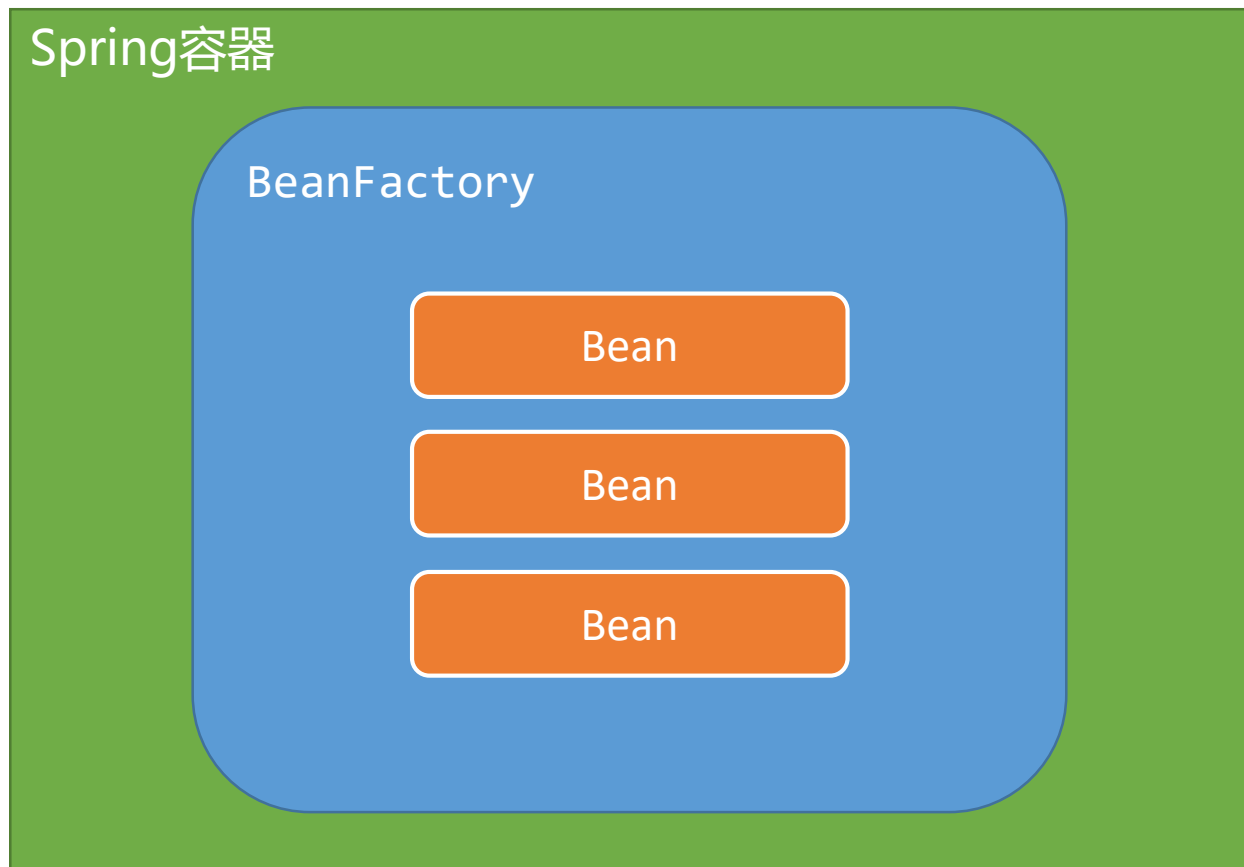
Spring Framework Runtime



- Bean管理
- IOC (DI)
- AOP

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-context</artifactId>  
  <version>4.3.5.RELEASE</version>  
</dependency>
```

4.3.12.RELEASE



applicationContext.xml

```
<?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">  
  
</beans>
```

将Bean放入Spring容器

凯盛软件

```
<bean id="userDao" class="com.kaishengit.dao.UserDao"/>
```

- **id**: bean在容器中的名称，不能重复
- **class**: 放入Spring容器的类的完全限定名称

Spring是如何管理Bean

<http://blog.csdn.net/qiesheng/article/details/60869592>

从Spring容器中获取Bean

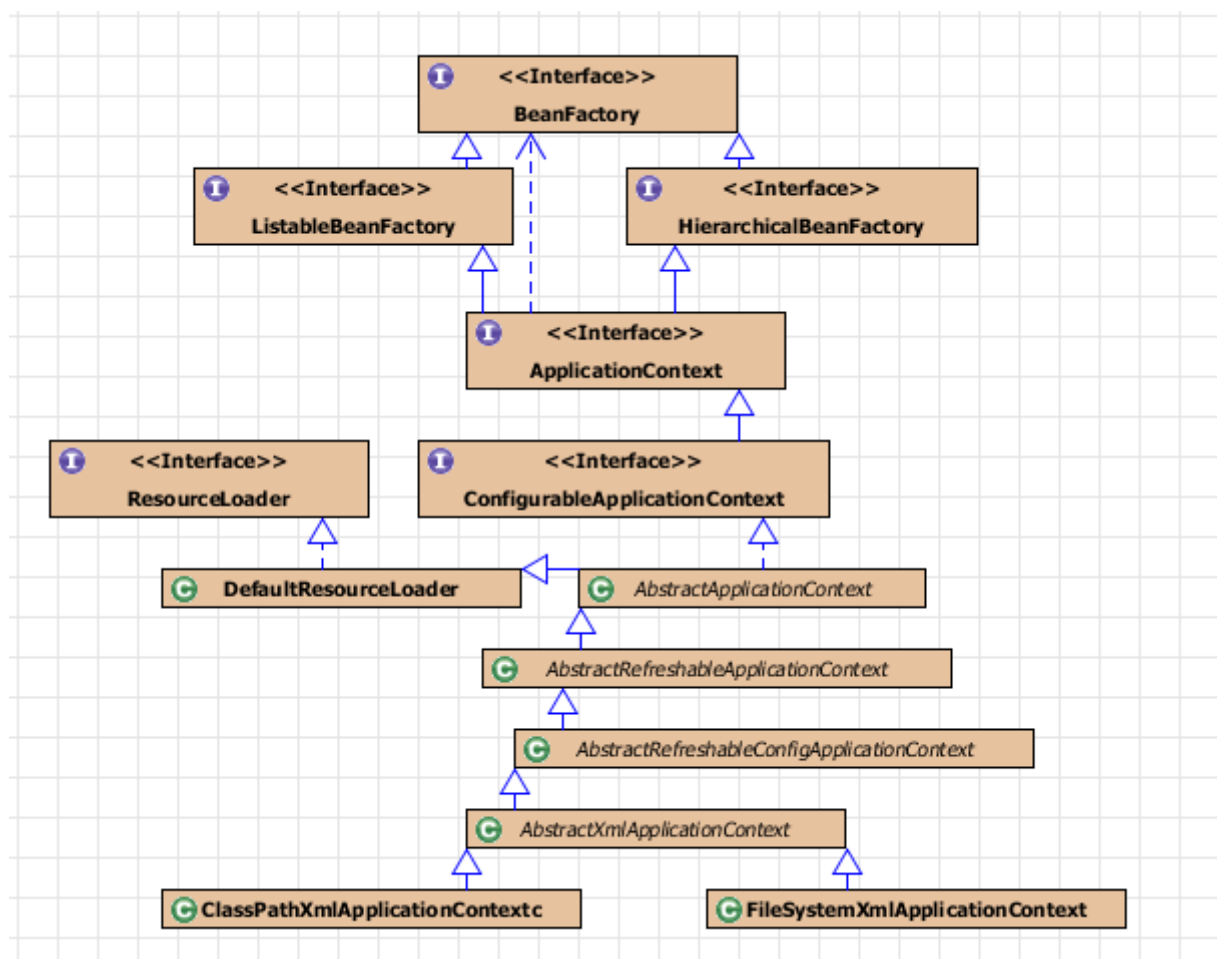
凯盛软件

// 获取Spring 容器

```
ApplicationContext applicationContext =  
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

// 从Spring 容器中获取Bean

```
UserDao userDao = (UserDao) applicationContext.getBean("userDao");
```



- 别名

```
<bean id="userDao" class="com.kaishengit.dao.UserDao"/>  
<alias name="userDao" alias="myUserDao"/>
```

- scope

- 放入spring容器管理的Bean默认为单例

```
<bean id="userDao" class="com.kaishengit.dao.UserDao" scope="singleton"/>
```

```
<bean id="userDao" class="com.kaishengit.dao.UserDao" scope="prototype"/>
```

- lazy init

- 放入Spring容器的Bean在容器启动时创建对象

```
<bean id="userDao" class="com.kaishengit.dao.UserDao" lazy-init="false"/>
```

- IOC 控制反转
- DI 依赖注入
- 作用：**降低类之间的耦合性**
- 两种形式
 - Set注入
 - 构造方法注入

```
public class UserService {  
  
    private UserDao userDao;  
  
    public void setUserDao(UserDao userDao) {  
        this.userDao = userDao;  
    }  
}  
  
<bean id="userDao" class="com.kaishengit.dao.UserDao"/>  
<bean id="userService" class="com.kaishengit.service.UserService">  
    <property name="userDao" ref="userDao"/>  
</bean>
```


更多Set注入方式

```
<bean id="userService" class="com.kaishengit.service.UserService">
    <property name="userDao">
        <ref bean="userDao"/>
    </property>
</bean>
```

```
<bean id="userService" class="com.kaishengit.service.UserService">
    <property name="userDao">
        <bean class="com.kaishengit.dao.UserDao"/>
    </property>
</bean>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userDao" class="com.kaishengit.dao.UserDao"/>
    <bean id="userService" class="com.kaishengit.service.UserService" p:userDao-ref="userDao"/>
```

```
public class UserService {  
  
    private UserDao userDao;  
    private Integer age;  
    private String name;  
    private List<String> list;  
    private Set<Double> set;  
    private Map<String,String> map;  
    private Properties properties;  
  
    //setter...
```

```
<property name="userDao" ref="userDao"/>
<property name="age" value="23"/>
<property name="name" value="Jack"/>
<property name="list">
    <list>
        <value>Rose</value>
        <value>Tom</value>
    </list>
</property>
```

```
<property name="set">
  <set>
    <value>34.45</value>
    <value>55.98</value>
  </set>
</property>
<property name="map">
  <map>
    <entry key="k1" value="v1"/>
    <entry key="k2" value="v2"/>
  </map>
</property>
```

```
<property name="properties">
  <props>
    <prop key="p1">v1</prop>
    <prop key="p2">v2</prop>
  </props>
</property>
```

- 放入Spring容器的Bean都应该有一个无参数的构造方法

```
public class UserService {
```

```
    private UserDao userDao;
```

```
    public UserService(UserDao userDao) {
```

```
        this.userDao = userDao;
```

```
    }
```

```
}
```

```
<bean id="userDao" class="com.kaishengit.dao.UserDao"/>
```

```
<bean id="userService" class="com.kaishengit.service.UserService">
```

```
    <constructor-arg name="userDao" ref="userDao"/>
```

```
</bean>                                name 构造方法参数的名称（推荐使用）
```

`</bean>` index 构造方法参数的索引，从0开始

`</bean>` type 构造方法参数的类型，使用时要求构造方法的参数类型唯一

应该选择哪种注入方式？

使用构造方法注入的理由:

- 构造方法注入使用强依赖规定,如果不给足够的参数,对象则无法创建。
- 由于Bean 的依赖都通过构造方法设置了,那么就不用写更多的 set 方法,有助于减少代码量。

使用 set 注入的理由:

- 如果Bean有很多的依赖,那么构造方法的参数列表会变的很长。
- 如果一个对象有多种构造方法,构造方法会造成代码量增加。
- 如果构造方法中有两个以上的参数类型相同,那么将很难确定参数的用途。

```
<bean id="userService" class="com.kaishengit.service.UserService" autowire="byName"/>
```

- autowire属性值：
 - no 默认值，不进行自动注入
 - byName 根据需要注入的属性名在容器内寻找名称相同的Bean，如果找到就注入，找不到就不注入
 - byType 根据需要注入的属性类型在容器找类型相同的Bean，如果找到就注入，找不到就不注入，如果找到多个类型相同的Bean，则抛出异常 `NoUniqueBeanDefinitionException`
 - constructor 类似byType注入，但是使用在构造方法之上
- 其中byName和byType方式都是基于set注入方式进行的

根据构造方法进行自动注入，先根据构造方法参数的名称从spring容器中查找名字相同的bean进行注入，如果没有找到再根据参数的类型从spring容器中查找类型相同的bean进行注入。

如果spring采用xml这种配置形式，一般不采用自动注入的方式，
如果spring采用注解的方式，会采用自动注入的方式。

代理模式

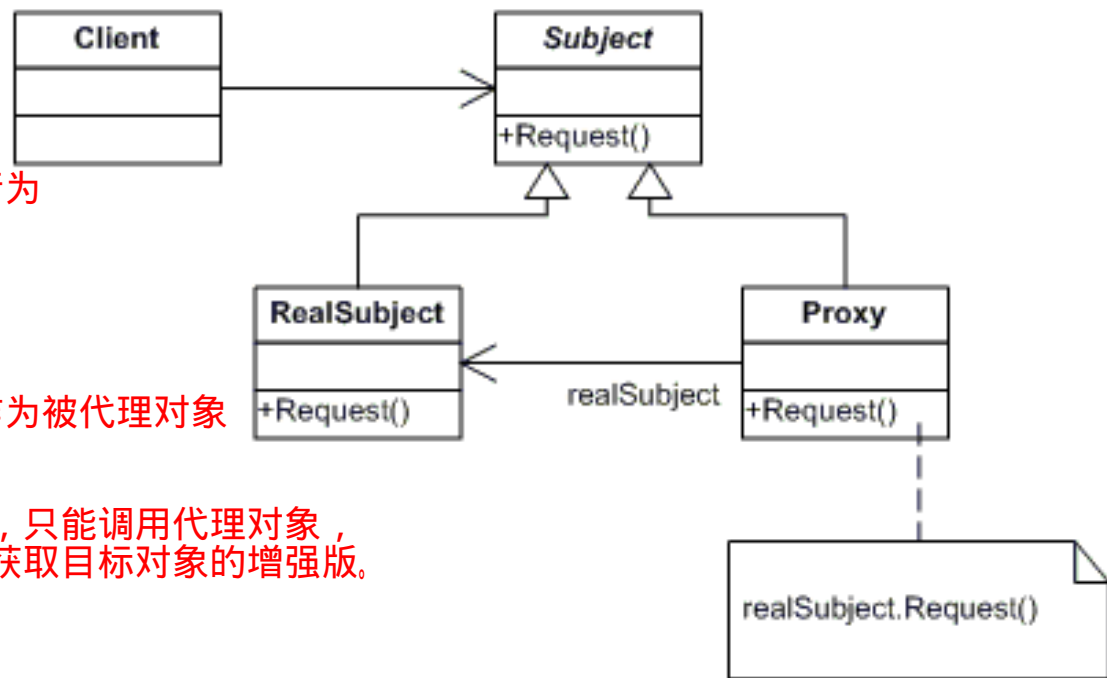
凯盛软件

接口：定义目标对象和代理对象的共同行为

目标对象：实现接口

代理对象：实现接口，并引用目标对象作为被代理对象

调用者/客户端：不能直接调用目标对象，只能调用代理对象，
或者调用代理对象可以获取目标对象的增强版。



- 程序自动产生代理对象 动态就是自动产生代理对象，使用动态代理模式的目的是增加通知。
- 两种方式 优先使用JDK接口实现
 - 使用JDK实现（要求目标对象必须有接口，根据接口动态产生实现类，并引用目标对象）
 - 使用CGLib实现（不要求目标对象有接口，动态产生目标对象的子类，调用目标对象的方法）
第三方工具类 根据目标对象动态产生一个子类代理对象，从而实现动态代理

jdk实现：

1. 实现java.lang.reflect.InvocationHandler接口，作为动态代理的模板
2. 调用java.lang.reflect.Proxy对象的newProxyInstance方法动态产生目标对象的代理类
3. 目标对象的接口指向动态产生的代理类对象

```
public class SubjectInvocationHandler implements InvocationHandler {
```

是产生代理类的一个模板

```
    private Object target;
```

```
    public SubjectInvocationHandler(Object target) {
```

```
        this.target = target;
```

```
    }
```

```
    @Override
```

method表示目标对象target其中的一个方法

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

```
        System.out.println("--Before--");
```

args表示方法的参数

```
        Object result = method.invoke(target, args); //代表目标对象target的方法的执行
```

```
        System.out.println("--After--");
```

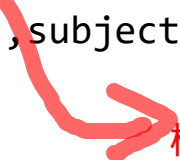
```
        return result;
```

```
    }
```

```
}
```

```
RealSubject realSubject = new RealSubject();
SubjectInvocationHandler subjectInvocationHandler = new SubjectInvocationHandler(realSubject);
接口 //产生的代理类需要用接口指向实现类
Subject subject = (Subject) Proxy.newProxyInstance(realSubject.getClass().getClassLoader(),
    realSubject.getClass().getInterfaces(), subjectInvocationHandler);

subject.sayHello();
```



根据模板产生的代理类

```
<dependency>
```

```
  <groupId>cglib</groupId>
```

```
  <artifactId>cglib</artifactId>
```

```
  <version>3.2.4</version>
```

```
</dependency>
```

1. 添加cglib的maven依赖

拦截器

```
public class SubjectMethodInterceptor implements MethodInterceptor {
```

```
  @Override
```

2. 实现MethodInterceptor接口

```
  public Object intercept(Object o, Method method, Object[] params, MethodProxy methodProxy)
```

```
    throws Throwable {
```

```
    System.out.println("--Before--");
```

```
    Object result = methodProxy.invokeSuper(o,params);
```

```
    System.out.println("--After--");
```

```
    return result;
```

```
  }
```

```
}
```

```
Enhancer enhancer = new Enhancer();    3.创建Enhancer类的对象，用来产生目标对象的子类
enhancer.setSuperclass(RealSubject.class);
enhancer.setCallback(new SubjectMethodInterceptor());

RealSubject realSubject = (RealSubject) enhancer.create();
realSubject.sayHello();
```


- 面向切面编程，目的是降低类直接的耦合程度
- 基于动态代理实现

目标对象

代理对象

通知

切入点

- AOP的通知类型

- 前置通知
- 后置通知
- 异常通知
- 最终通知
- 环绕通知

步骤：

1. 添加maven依赖

2. 配置文件中添加schema

3. 创建一个通知类，用于存放各种通知（方法）

4. 在xml中进行配置：将通知类放入spring容器，掌握各种通知配置的对应节点名称

添加Maven依赖并修改配置文件

凯盛软件

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>4.3.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>4.3.5.RELEASE</version>
</dependency>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">
```

```
public class AopAspect {  
  
    public void beforeAdvice() {  
        System.out.println("前置通知");  
    }  
    public void afterAdvice() {  
        System.out.println("后置通知");  
    }  
    public void exceptionAdvice() {  
        System.out.println("异常通知");  
    }  
    public void finallyAdvice() {  
        System.out.println("最终通知");  
    }  
}
```

```
<bean id="aopAspect" class="com.kaishengit.aop.AopAspect"/>
```

```
<aop:config>
```

```
  <aop:aspect ref="aopAspect">
```

```
    <aop:pointcut id="pointcut" expression="execution(* com.kaishengit.service.*.*(..))"/>
```

```
    <aop:before method="beforeAdvice" pointcut-ref="pointcut"/>
```

```
    <aop:after-returning method="afterAdvice" pointcut-ref="pointcut"/>
```

```
    <aop:after-throwing method="exceptionAdvice" pointcut-ref="pointcut"/>
```

```
    最终通知 <aop:after method="finallyAdvice" pointcut-ref="pointcut"/>
```

```
  </aop:aspect>
```

```
</aop:config>
```

表示返回类型，
*表示不限制

本包及其子包

所有方法

参数列表

```
public void aroundAdvice(ProceedingJoinPoint joinPoint) {  
    try {  
        System.out.println("--before--");  
        Object result = joinPoint.proceed(); // 目标对象方法的执行  
        System.out.println("--after--");  
    } catch (Throwable throwable) {  
        throwable.printStackTrace();  
        System.out.println("--exception--");  
    } finally {  
        System.out.println("--finally--");  
    }  
}
```

```
<aop:config>
```

```
    <aop:aspect ref="aopAspect">
```

```
        <aop:pointcut id="pointcut" expression="execution(* com.kaishengit.service..*.*(..))"/>
```

```
        <aop:around method="aroundAdvice" pointcut-ref="pointcut"/>
```

```
    </aop:aspect>
```

```
</aop:config>
```

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

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

- Spring内置注解
 - @Component
 - @Service
 - @Repository

自动扫描：在xml添加context的schema

bean管理：@Component
@Service
@Repository
@Named 需添加maven依赖
@Scope 设定bean是否是单例
@Lazy 懒加载

- JSR330
 - @Named

IOC/DI: @Autowired
@Inject
@Resource

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

```
@Repository
public class UserDao {

    public void save() {
        System.out.println("UserDao save...");
    }
}
```

```
import javax.inject.Named;
```

```
@Named
public class UserDao {

    public void save() {
        System.out.println("UserDao save...");
    }
}
```

```
@Repository
@Scope("prototype")
@Lazy
public class UserDao {

    public void save() {
        System.out.println("UserDao save...");
    }
}
```


- Spring内置注解
 - @Autowired
- JSR330注解
 - @Inject
- JSR250注解
 - @Resource

```
@Service
public class UserService {

    private UserDao userDao;

    @Autowired
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

```
@Service
public class UserService {

    @Autowired
    private UserDao userDao;
```

```
<aop:aspectj-autoproxy/> //开启基于注解的aop
```

```
@Component
```

```
@Aspect
```

```
public class AopAspect {
```

```
    @Pointcut("execution(* com.kaishengit.service..*.*(..))")
```

```
    public void pointcut(){}
```

```
    @Before("pointcut()")
```

```
    public void beforeAdvice() {
```

```
        System.out.println("前置通知");
```

```
    }
```

```
    @AfterReturning("pointcut()")
```

```
    public void afterAdvice() {
```

```
        System.out.println("后置通知");
```

```
    }
```

1. 在xml中开启基于注解的AOP支持

2. 将通知类放入spring容器，并设置为

AOP通知类 @Component @Aspect

3. 其他注解

```
@AfterThrowing("pointcut()")
public void exceptionAdvice() {
    System.out.println("异常通知");
}


@After("pointcut()")
public void finallyAdvice() {
    System.out.println("最终通知");
}

@Around("pointcut()")
public void aroundAdvice(ProceedingJoinPoint joinPoint) {
    try {
        System.out.println("--before--");
        Object result = joinPoint.proceed(); // 目标对象方法的执行
        System.out.println("--after--");
    } catch (Throwable throwable) {
        throwable.printStackTrace();
        System.out.println("--exception--");
    } finally {
        System.out.println("--finally--");
    }
}
```

基于DI的Spring单元测试

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>4.3.5.RELEASE</version>
</dependency>
```

1. 添加spring-test的maven依赖

2.  `@RunWith(SpringJUnit4ClassRunner.class)`
`@ContextConfiguration(locations = "classpath:applicationContext.xml")` //告诉spring从classpath中查找文件
`public class SpringTestCase {`

 `@Autowired`
 `private UserService userService;`

 `@Test`
 `public void load() {`
 `System.out.println(userService);`
 `}`
`}`

替代applicationContext.xml

```
package com.kaishengit;
```

```
import org.springframework.context.annotation.ComponentScan;
```

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

```
import org.springframework.context.annotation.EnableAspectJAutoProxy;
```

```
@Configuration
```

```
@ComponentScan
```

```
@EnableAspectJAutoProxy
```

```
public class Application {
```

```
}
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = Application.class)
public class SpringTestCase {
```

```
ApplicationContext applicationContext = new AnnotationConfigApplicationContext(Application.class);
applicationContext.getBean("userDao");
```

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.3.5.RELEASE</version>

</dependency>

1. 添加maven依赖

2. 配置数据库连接池

添加dbcp2依赖，mysql驱动依赖

3. 配置jdbcTemplate

注入数据库连接池

update () 执行增删改三种语句

queryForObject () 返回单条查询结果

query () 返回多条查询结果

RowMapper接口 (两种)

4. @PropertySource


```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:///mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="rootroot"/>
</bean>
```

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

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">  
  <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>
```

config.properties

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql:///mydb

jdbc.username=root

jdbc.password=root

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

```
@Repository  
public class UserDao {  
  
    @Autowired  
    private JdbcTemplate jdbcTemplate;
```

执行insert update delete操作

```
public void save(User user) {  
    String sql = "insert into t_user(username,password) values(?,?)";  
    jdbcTemplate.update(sql,user.getUsername(),user.getPassword());  
}  
  
public void update(User user){  
    String sql = "update t_user set username=?,password=? where id = ?";  
    jdbcTemplate.update(sql,user.getUsername(),user.getPassword(),user.getId());  
}  
  
public void delete(Integer id) {  
    String sql = "delete from t_user where id = ?";  
    jdbcTemplate.update(sql,id);  
}
```

```
public User findById(Integer id) {  
    String sql = "select * from t_user where id = ?";  
    return jdbcTemplate.queryForObject(sql, new RowMapper<User>() {  
        @Override  
        public User mapRow(ResultSet resultSet, int i) throws SQLException {  
            User user = new User();  
            user.setId(resultSet.getInt("id"));  
            user.setUsername(resultSet.getString("username"));  
            user.setPassword(resultSet.getString("password"));  
            return user;  
        }  
    }, id);  
}
```

```
public List<User> findAll() {  
    String sql = "select * from t_user";  
    return jdbcTemplate.query(sql, new RowMapper<User>() {  
        @Override  
        public User mapRow(ResultSet resultSet, int i) throws SQLException {  
            User user = new User();  
            user.setId(resultSet.getInt("id"));  
            user.setUsername(resultSet.getString("username"));  
            user.setPassword(resultSet.getString("password"));  
            return user;  
        }  
    });  
}
```

```
public List<User> findAll() {  
    String sql = "select * from t_user";  
    return jdbcTemplate.query(sql, new BeanPropertyRowMapper<User>(User.class));  
}
```

```
public User findByUserName(String userName) {  
    String sql = "select * from t_user where username = ?";  
    return jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<User>(User.class), userName);  
}
```

```
public Long count() {  
    String sql = "select count(*) from t_user";  
    return jdbcTemplate.queryForObject(sql, new SingleColumnRowMapper<Long>());  
}
```

Long.class

- DEFAULT 使用数据库默认隔离级别
- READ_UNCOMMITTED 允许读取尚未提交的数据。可能导致脏读、幻读或不可重复读。
- READ_COMMITTED 允许从已经提交的并发事务读取。可以防止脏读，但依然会出现幻读和不可重复读。
- REPEATABLE_READ 对相同字段的多次读取结果是相同的，除非数据被当前事务改变。可以防止脏读和不可重复读，但幻读依然出现。
- SERIALIZABLE 完全符合ACID的隔离级别，确保不会发生脏读，幻读和不可重复读。

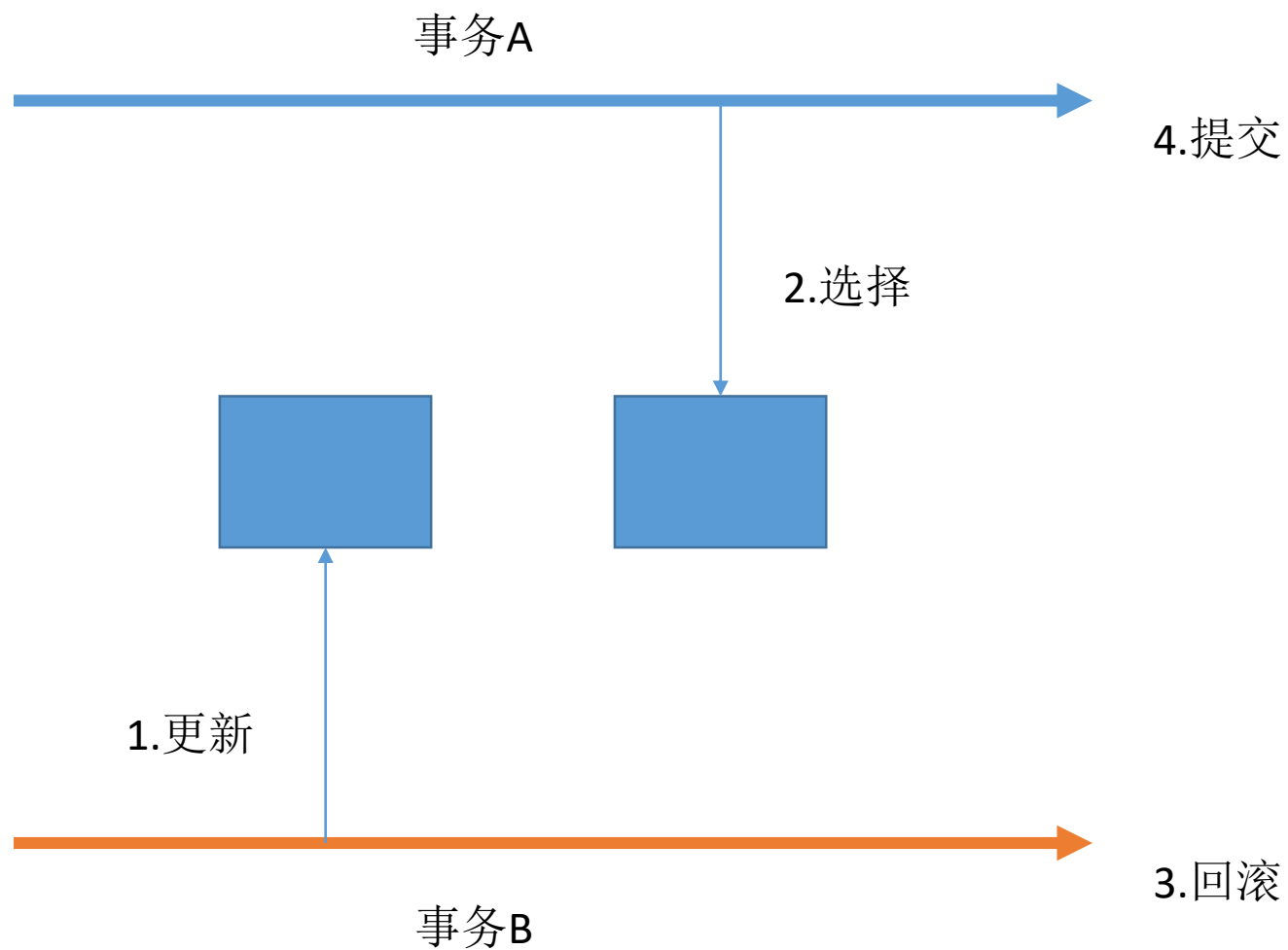
事务：为了保证数据的一致性

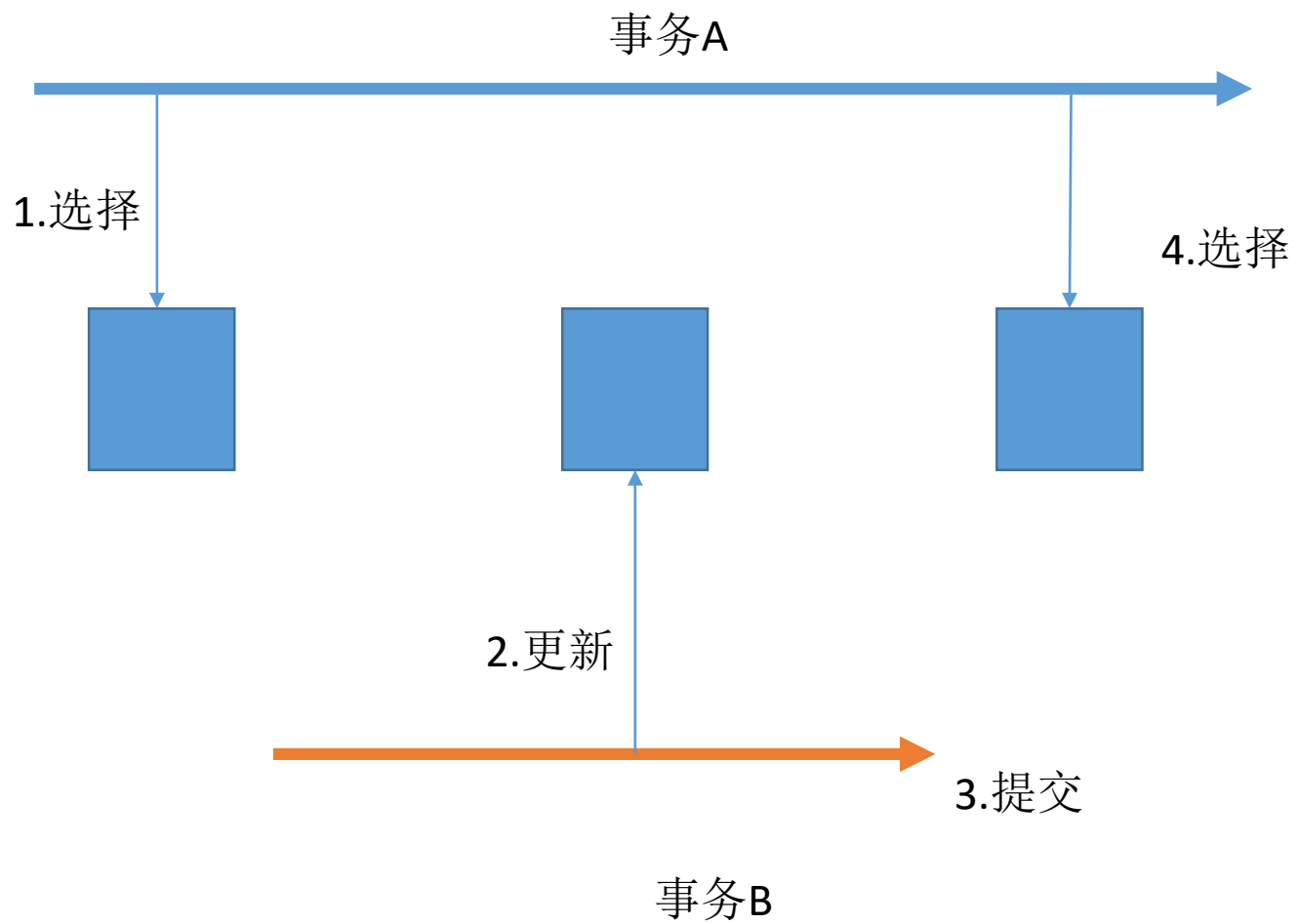
分类： 编程式事务：将控制事务的代码放到业务中
声明式事务：将事务从业务代码中提取出来，

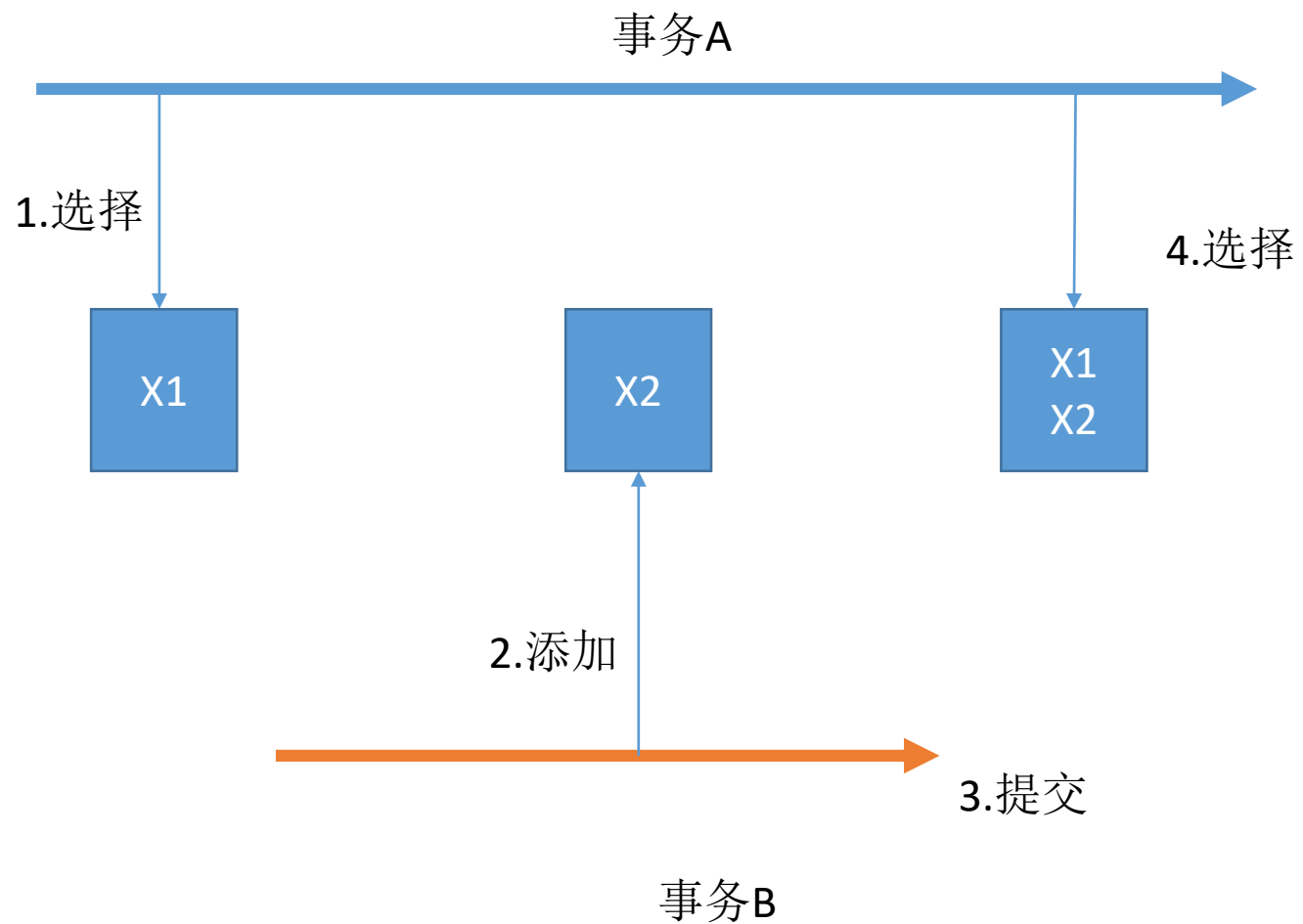
Spring中事务的实现本质是AOP

使用步骤： 添加maven依赖
配置文件中配置事务管理器
开启基于注解的事务配置
使用@Transactional实现事务，应在需要的方法上添加事务，不需要的不用添加，
例如查询、单条的增删改，一般在service层添加事务
(添加在类上，类中所有方法执行都会在事务中，添加方法上，该方法执行会在事务中)
默认当方法发生运行时异常会自动回滚事务，可以通过rollbackFor改变这种属性

- 脏读：一个事务读取到另一个事务没有提交到的数据。
- 不可重复读：在同一事务中，多次读取同一数据返回的结果不同。
- 幻读：一个事务读取到另一个事务已经提交的事务。







事务隔离级别	脏读	不可重复读	幻读
READ_UNCOMMITTED	允许	允许	允许
READ_COMMITTED	禁止	允许	允许
REPEATABLE_READ	禁止	禁止	允许
SERIALIZABLE	禁止	禁止	禁止

- REQUIRED 业务方法需要在事务中运行。如果方法运行时，已经处在一个事务中，那么加入到这个事务中，否则自己创建一个事务。（大部分情况下使用）
- NOT-SUPPORTED 声明方法需要事务。如果方法没有关联到一个事务，容器会为它开启一个事务，如果方法在一个事务中被调用，该事务将会被挂起，在方法调用结束后，原先的事务会恢复执行。
- REQUIREDNEW 业务方法必须在自己的事务中运行。一个新的事务将被启动，而且如果有一个事务正在运行，则将这个事务挂起，方法运行结束后，新事务执行结束，原来的事务恢复运行。

- **MANDATORY** 该方法必须运行在一个现有的事务中，自身不能创建事务，如果方法在没有事务的环境下被调用，则会抛出异常。
- **SUPPORTS** 如果该方法在一个事务环境中运行，那么就在这个事务中运行，如果在事务范围外调用，那么就在一个没有事务的环境下运行。
- **NEVER** 表示该方法不能在有事务的环境下运行，如果在有事务运行的环境下调用，则会抛出异常
- **NESTED** 如果一个活动的事务存在，则运行在一个嵌套的事务中，如果没有活动事务，则按照REQUIRED事务方式执行。该事务可以独立的进行提交或回滚，如果回滚不会对外围事务造成影响

<!-- JDBC 事务管理器 -->

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

<!-- 基于注解的事务 -->

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```



```
@Service
@Transactional
public class UserService {

    @Autowired
    private UserDao userDao;

    public void save(User user) {
        userDao.save(user);
    }

    @Transactional(readonly = true)
    public User findById(Integer id) {
        return userDao.findById(id);
    }
}
```

- 隔离级别的设置，默认为READ_COMMITTED

`@Transactional(isolation = Isolation.SERIALIZABLE)`

- 传播属性的设置，默认为REQUIRED

`@Transactional(propagation = Propagation.REQUIRED)`

- 回滚设置，默认发生运行时异常事务才会回滚，非运行时异常事务不回滚

`@Transactional(rollbackFor = Exception.class)`

<!-- 基于XML的事务-->

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

```
  <tx:attributes>
```

```
    <tx:method name="save*" propagation="REQUIRED"/>
```

```
    <tx:method name="del*" />
```

```
    <tx:method name="edit*" />
```

```
    <tx:method name="find*" read-only="true"/>
```

```
  </tx:attributes>
```

```
</tx:advice>
```

```
<aop:config>
```

```
  <aop:pointcut expression="execution(* com.kaishengit.service..*.*(..))" id="myPointcut"/>
```

```
  <aop:advisor advice-ref="txAdvice"/>
```

```
</aop:config>
```

```
<dependency>
```

```
  <groupId>org.mybatis</groupId>
```

```
  <artifactId>mybatis</artifactId>
```

```
  <version>3.4.0</version>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.mybatis</groupId>
```

```
  <artifactId>mybatis-spring</artifactId>
```

```
  <version>1.3.1</version>
```

```
</dependency>
```

```
<!--MyBatis SQLSessionFactory-->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <!--数据源-->
    <property name="dataSource" ref="dataSource"/>
    <!--别名类所在包-->
    <property name="typeAliasesPackage" value="com.kaishengit.pojo"/>
    <!--Mapper文件所在位置-->
    <property name="mapperLocations" value="classpath:mapper/*.xml"/>
    <!--其他配置-->
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration">
            <property name="mapUnderscoreToCamelCase" value="true"/>
        </bean>
    </property>
</bean>
```

```
<!--Mapper 自动扫描-->
```

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
```

```
    <property name="basePackage" value="com.kaishengit.mapper"/>
```

```
</bean>
```

过滤Spring的日志输出

凯盛软件

```
log4j.appender.sout=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.sout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.sout.layout.ConversionPattern=[%p](%d)-%m-%l-%n
```

```
log4j.logger.org.springframework=INFO
```

```
log4j.rootLogger=debug,sout
```