

# Week 14

---

## day 4

---

### 设计模式

**设计模式：**

被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结

**注意：**

- 不要乱用
- 需结合实际情况（不要为了用设计模式而用）
- 设计模式不是万能的

### 优点

- 代码质量好，可靠性高
- 代码重用
- 代码更规范，更易理解

---

## 软件设计的原则

- S：单一职责原则

一个模块负责一个功能

一个类负责一个业务

一个API实现一个功能

将不同的功能分隔开

- O：开放封闭原则

对扩展开放：新功能、新需求可以去增加代码（增加一个类）

对修改封闭：不建议修改已有的代码（特别是底层的API）

- L：李氏替换原则

父类接受子类实例

任何一个基类可以出现的地方，子类一定可以出现

子类最好不要去重写父类方法，添加新方法即可，可以实现抽象方法

```
UserDao dao = new UserDaoImpl()
```

- I：接口隔离原则

一个接口里不要集成太多的功能

- D：依赖倒置原则

先去设计抽象的接口，再去实现子类，然后让接口接收对象实例

## 常见设计模式

### 单例模式 (重点)

概念：单一实例，一个类只对应一个实例

特点：

- 单例类只能有一个实例
- 构造方法私有（必须自己创建自己的唯一实例）
- 单例类必须给所有其他对象提供这一实例

实现：

- 线程不安全的懒加载(懒加载：即在将要使用时才创建对象)

多线程环境下可能会有多个实例出现

```
public class Singleton1 {  
    // 包含自己类型的成员变量  
    private static Singleton1 instance;  
    // 构造方法私有  
    private Singleton1() {}  
    // 提供一个方法给其他方法调用  
    public static Singleton1 getInstance() {  
        if (instance == null) {  
            instance = new Singleton1();  
        }  
        return instance;  
    }  
}
```

- 线程安全的懒加载（锁实现）

线程安全，但是加锁影响执行效率（线程阻塞需要挂起和恢复线程，都需要从用户态切到内核态）

```

public class Singleton2 {

    private static Singleton2 instance;

    private Singleton2() {}

    public synchronized static Singleton2 getInstance() {
        if (instance == null) {
            instance = new Singleton2();
        }
        return instance;
    }
}

```

- 立即加载（自动保证线程安全）

在类加载时完成实例化（就算有多个线程同时调用 `getSingleton`，也只会在类加载时执行一次实例化）

```

public class Singleton3 {

    private static Singleton3 instance = new Singleton3();

    // static {
    //     instance = new Singleton3();
    // }

    private Singleton3() {}

    public static Singleton3 getInstance() {
        return instance;
    }
}

```

- 线程安全的懒加载：静态内部类实现

```

/**
 * 通过静态内部类实现的线程安全的懒加载，就算有多个线程同时调用getInstance
 * 静态内部类的类加载只会执行一次
 * 也保证了该实例只会被创建一次
 */
public class Singleton {
    // 构造方法私有，单例类的实例只能由自身的方法向外提供
    private Singleton() {}

    // 向外提供该唯一实例，必须为static方法
    public static Singleton getInstance() {
        return Inner.getInstance();
    }

    private static class Inner {
        // 有该类型的静态变量，静态内部类的类加载在内部类的方法被调用时触发
    }
}

```

```
    static Singleton instance = new Singleton();
    // 返回该唯一实例
    static Singleton getInstance() {
        return instance;
    }
}
```

---

## 工厂模式

### 概念

用于生产对象，在创建对象时不会对客户暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象

#### 简单工厂：

传入一个类的名字，返回一个这种类的实例（不符合开闭原则）

#### 工厂方法模式：

创建一个工厂接口和创建多个工厂实现类

完全遵从开闭原则，不对任何已有的代码进行修改，而是直接新增一个类

---

## 代理模式（重点）

代理：为某个对象提供一个代理对象以控制对这个对象的访问

代理对象：中介

代理类负责为委托类预处理消息、过滤消息并转发消息，以及进行消息被委托类执行后的后续处理

需要执行到被代理类的方法

## 优点

可以在目标对象实现的基础上，增强额外的功能，扩展目标对象的功能

### 静态代理

基于组合或者继承实现代理（多一个中间类）（增强方法）

```
public class HouseProxy1 {
    private HouseOwner houseOwner = new HouseOwner();

    public boolean rentHouse(int money) {
        money -= 500;
        return houseOwner.rentHouse(500);
    }
}
```

```
public class HouseProxy2 extends HouseOwner {

    @Override
    public boolean rentHouse(int money) {
        money -= 500;
        return super.rentHouse(money);
    }
}
```

## 动态代理

直接给某个目标对象生成一个代理对象，而不需要代理类的存在  
(指定一组接口及委托类对象，就能动态获得代理对象)

## 动态代理的实现方式

- jdk提供了一个Proxy类，可以直接生成代理对象（前提是委托类要有一个接口）  
此时代理类与委托类是同一个接口的两个实现子类

```
Proxy.newProxyInstance(classloader, interfaces, invocationHandler)
```

要求代理对象与被代理对象的类加载器是一致的

被代理对象的接口

调用处理器

```
public class JdkProxyTest {

    @Test
    public void test1() {
        HouseOwner houseOwner = new HouseOwner();
        Rent rentProxy = (Rent)
            Proxy.newProxyInstance(houseOwner.getClass().getClassLoader(),
                houseOwner.getClass().getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method,
                        Object[] args) throws Throwable {
                        if ("rentHouse".equals(method.getName())) {
                            args[0] = (int)args[0] - 500;
                        }
                    }
                });
    }
}
```

```

        Object invoke = method.invoke(houseOwner, args);
        return invoke;
    }
});

System.out.println(rentProxy.rentHouse(1500));
System.out.println(rentProxy.rentHouse(2000));
}
}

```

- cglib: 通过继承实现，继承委托类  
此时代理类是委托类的子类

```

public class CglibTest {

    @Test
    public void test1() {
        HouseOwner houseOwner = new HouseOwner();
        HouseOwner houseProxy = (HouseOwner)
Enhancer.create(HouseOwner.class, new InvocationHandler() {
            @Override
            public Object invoke(Object o, Method method, Object[] objects)
throws Throwable {
                objects[0] = (int)objects[0] - 500;
                return method.invoke(houseOwner, objects);
            }
        });
        System.out.println(houseProxy.rentHouse(2000));
    }
}

```

## 建造者模式

更重视参数的设计

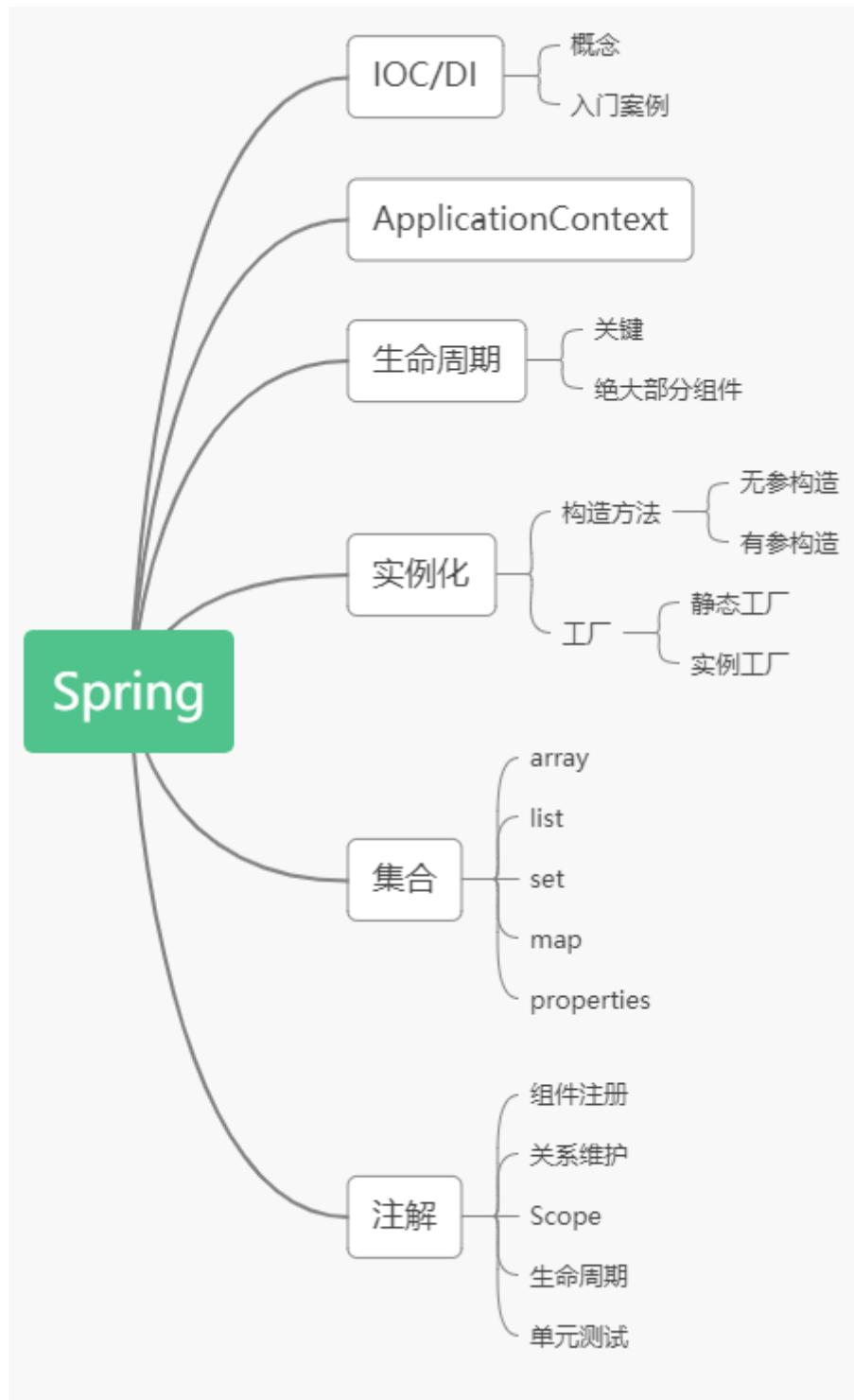
```

@Test
public void test1() {
    HumanBuilder humanBuilder = new HumanBuilder();
    // 链式调用 return this;
    Human human = humanBuilder
        .setBodyFat(true)
        .setHeadEq(200)
        .setHeadIq(150)
        .setHeadHair("scarce")
        .setLegLength(150)
        .build();
    System.out.println(human);
}

```

## day 5

# Spring



整合性轻量级框架

## 核心概念

IOC: inverse of control 控制反转

控制：实例的生成权

反转：由应用程序反转交给spring

原先是自己new出来的，现在交给spring创建

实例交给了spring管理，容器：spring容器，IOC容器

DI: dependency injection 依赖注入

spring容器与应用程序 (现在是spring容器富有了)

依赖: 应用程序依赖spring容器, 因为实例都归spring容器管

注入: spring容器注入应用程序, 注入了实例和常量等外部资源

AOP: 面向切面编程

## 入门案例1

- 创建相应的类
- 引入spring依赖
- 配置文件: 写bean标签
- 使用组件, 实例: 从applicationcontext中获取bean(getBean BeanFactory接口中定义的方法)

## 入门案例2

- 引入依赖
- 创建类
- 配置文件

```
<!--第一步, 注册组件-->
<bean id="userDao" class="com.cskaoyan.dao.UserDaoImpl"/>

<!--第二步, 维护组件之间关系--> ref指向被引用标签的id
<bean id="userService" class="com.cskaoyan.service.impl.UserServiceImpl">
    <!--property的name属性值对应的是set方法名(通常set方法和成员变量名一致)-->
    <!--<property name="userDao" ref="userDao"/>-->
    <!--在property下也可以写ref子标签, 使用ref标签的bean属性指定id-->
    <property name="userDao">
        // ref标签用于引用容器中的组件
        <ref bean="userDao"/>
    </property>
</bean>
```

- 单元测试

## 核心API

filesystem和classpath: 加载配置文件的位置不同

- classpath: 加载classpath目录下的配置文件  
(开发目录中的java和resources目录编译后都放到了classpath目录下)
- filesystem: 加载所给的文件路径下的配置文件

## bean实例化

- 构造方法

- 无参构造 (常用)

```
<!--无参构造方法，使用set方法赋值--> 所以要保证有setter
<bean class="com.cskaoyan.bean.NoParamConstructorBean">
    <!--字符串、基本类型、包装类使用value属性或者value子标签-->
    <property name="username" value="songge"/>
    <property name="password">
        <value>niupi</value>
    </property>
</bean>
```

- 有参构造

value或者ref

```
<!--有参构造方法-->
<bean class="com.cskaoyan.bean.HasParamConstructorBean">
    <constructor-arg name="username" value="ligenli"/>
    <constructor-arg name="password">
        <value>daqi</value>
    </constructor-arg>
</bean>
```

- 工厂

- 静态工厂：生产方法是静态的

```
<!--静态工厂-->
<bean id="userFromStaticFactory"
      class="com.cskaoyan.factory.StaticFactory" factory-method="create"/>
```

- 实例工厂

```
<!--实例工厂-->
<bean id="instanceFactory"
      class="com.cskaoyan.factory.InstanceFactory"/>

<!--factoryBean引用容器中的工厂组件id
     factoryMethod 工厂组件中的生产方法
     -->
<bean id="userFromInstanceFactory" factory-bean="instanceFactory"
      factory-method="create"/>
```

- 单元测试：通过application.getBean时必须使用id才能获取到工厂所创建的对象，用.class只能获取到工厂自身

- FactoryBean

- BeanFactory：applicationcontext的父接口，生产全部实例（里面规定了getBean方法）
    - XXXFactoryBean：生产出来的组件类型是XXX

implement FactoryBean 通过getObject的类型判断

```
<!--factoryBean-->
<bean id="userFactoryBean"
      class="com.cskaoyan.factory.UserFactoryBean">
    <property name="param1" value="songge"/>
    <property name="param2" value="niupi"/>
</bean>
```

## spring容器中bean的作用域

- singleton: 容器中这个组件是单例形式 (也是默认的范围)

针对的是组件，不是一个类

```
<bean id="user1" class="com.daxiao.User"></bean>
<bean id="user2" class="com.daxiao.User"></bean>
```

这里是两个组件，他们都属于同一个类，所以从容器可以取出他们两个，两个都是唯一的

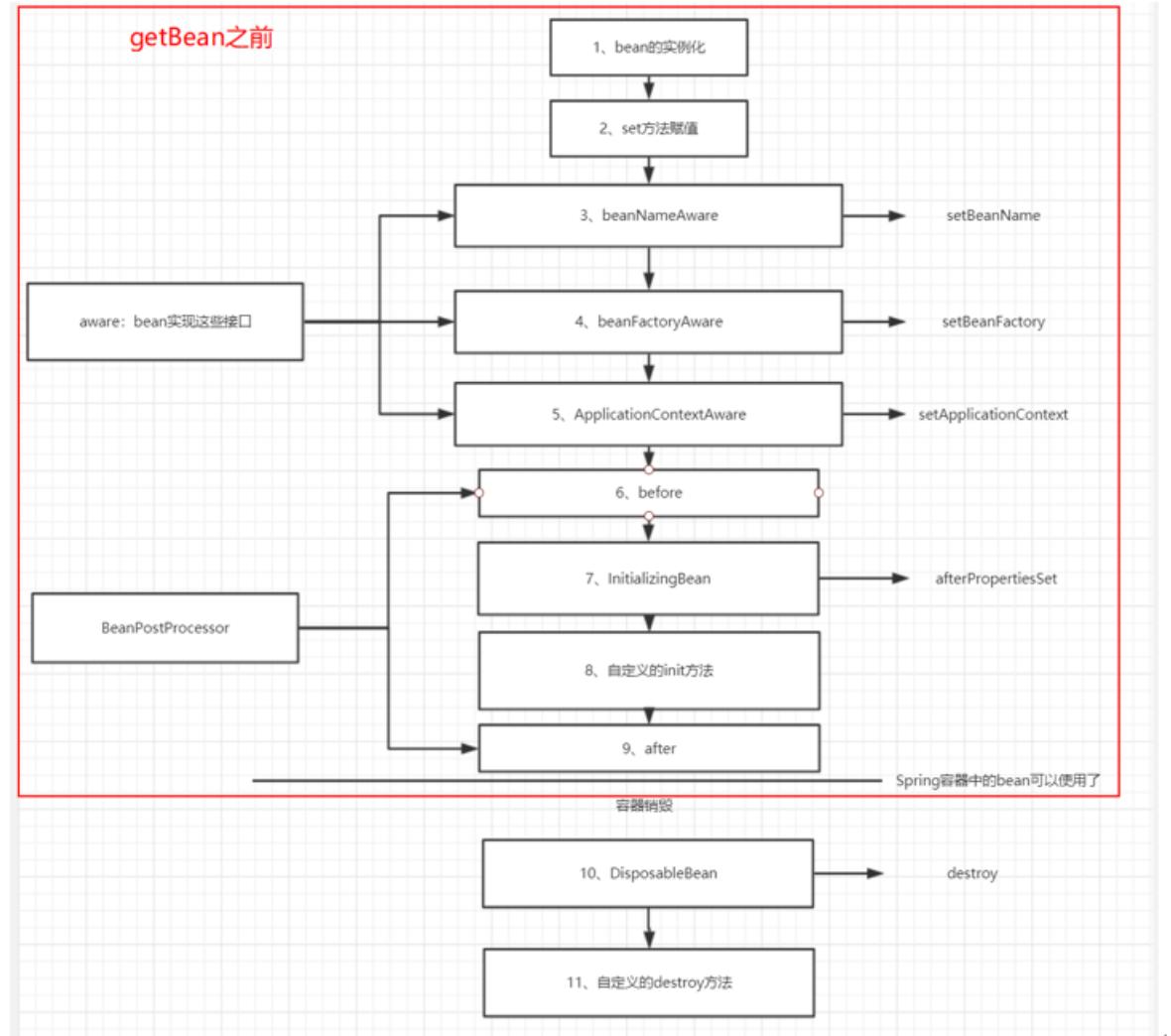
- prototype: 原型，每次getBean取出的都是一个新的对象

## 组件的含义：

- 如果是 singleton，那一个组件就对应唯一的一个对象
- 如果是 prototype，那一个组件就对应一堆内容相同但地址不同的对象

## bean的生命周期

第一步是调用无参构造方法



scope与生命周期相关的部分. 不同的scope对应的初始化时间也不同

- 单例:

生命周期是容器初始化的时候开始, `new ApplicationContext`

有容器销毁那部分的生命周期: 10 11

- prototype:

`getBean`的时候才开始初始化

没有10和11这个两个部分

## CollectionBean

组件中的成员变量类型是collection:

- 数组
- List,
- Set
- Map

map中的key: key/key-ref属性

map中的value: value-ref或者bean或者ref子标签

- Properties

- 在xml文件中，通过set方法，即property子标签下的array,list,set,map,props子标签来赋值
- 基本数据类型、字符串、包装类：用标签包裹值
- JavaBean(已经注册到容器的组件)：用  
或者是新创建的组件：

## 注解

先在xml文件中配置扫描包：让spring容器知道到哪个包目录下**扫描注解**

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

### • 组件注册

之前是通过xml文件中的bean标签来注册组件

现在是在类定义上写注解，注解中默认的值为value值，指的就是id值

什么都不写就是类名第一个字母小写

- @Component
- @Service：通常是加在service层的类上
- @Repository：加在dao层的类上
- @Controller：SpringMVC阶段注册组件的注解

实例工厂没有class

### • 注入

维护组件之间的关系

- 当成员变量是基本数据类型、字符串类型、包装类时

- 直接在成员变量上写注解：@Value("")
  - 引用额外的配置文件

先在properties文件中写键值对

然后再到xml文件中加载properties文件

@Value的时候，用引用变量的形式，不写死

```

service.username=Ligenli
service.age=22
resources
  application.xml
  param.properties

<!-- 加载classpath 下的额外的配置文件-->
<context:property-placeholder location="classpath:param.properties"/>

@Service
public class UserServiceImpl implements UserService{

    @Value("${service.username}")
    String username;
    @Value("${service.age}")
    Integer age;
}

}

```

- 也可以不到类的成员变量上面写注解，而是在xml文件中配置（如果有set方法）

也可以在spring配置文件中使用

```

<bean class="User">
    <property name="username" value="${service.username}" />
</bean>

```

- 当成员变量是组件类型（JavaBean类型时）

- @Autowired: 按照类型，要求这个类型的组件只有一个
- @Autowired + @Qualifier: 通过@Qualifier的value属性指定id
- @Resource: 默认按照类型，如果需要指定id，就使用name属性

```

@Resource(name = "userDaoImpl")
UserDao userDao;

@Autowired
@Qualifier("userDaoImpl")
UserDao userDao;

```

补充：@Autowired 如果类型没有匹配上，也可以按照 id 匹配

`@Autowired // 如果按照类型没有取到，会按照id去取  
UserDao userDaozzz;` 也会使用成员变量名作为组件id来注入

- scope/lifecycle

- scope

类定义上，默认不写仍然是singleton

`@Scope("prototype")` 也可以使用value属性指定scope，

注意：声明了prototype之后，在用注解方式注入实例的时候会注入多个不同的实例

- lifecycle

加载组件的方法中

`@PostConstruct`

`@PreDestroy`

- 单元测试
  - 导包：`spring-test`，注意要与`spring-context`包保持版本一致
  - 在测试类上增加注解
  - 然后可以直接让`spring`容器注入实例，而不用创建`context`对象

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:application.xml") 加载spring配置文件，注意不要漏掉了classpath
public class AnnotationTest {

    @Autowired
    UserService userService;
    @Autowired
    UserDao userDao;
    @Test
    public void mytest1(){
        //ApplicationContext applicationContext = new ClassPathXmlApplicationContext("application.xml");
        //applicationContext.getBean()
    }
}
```

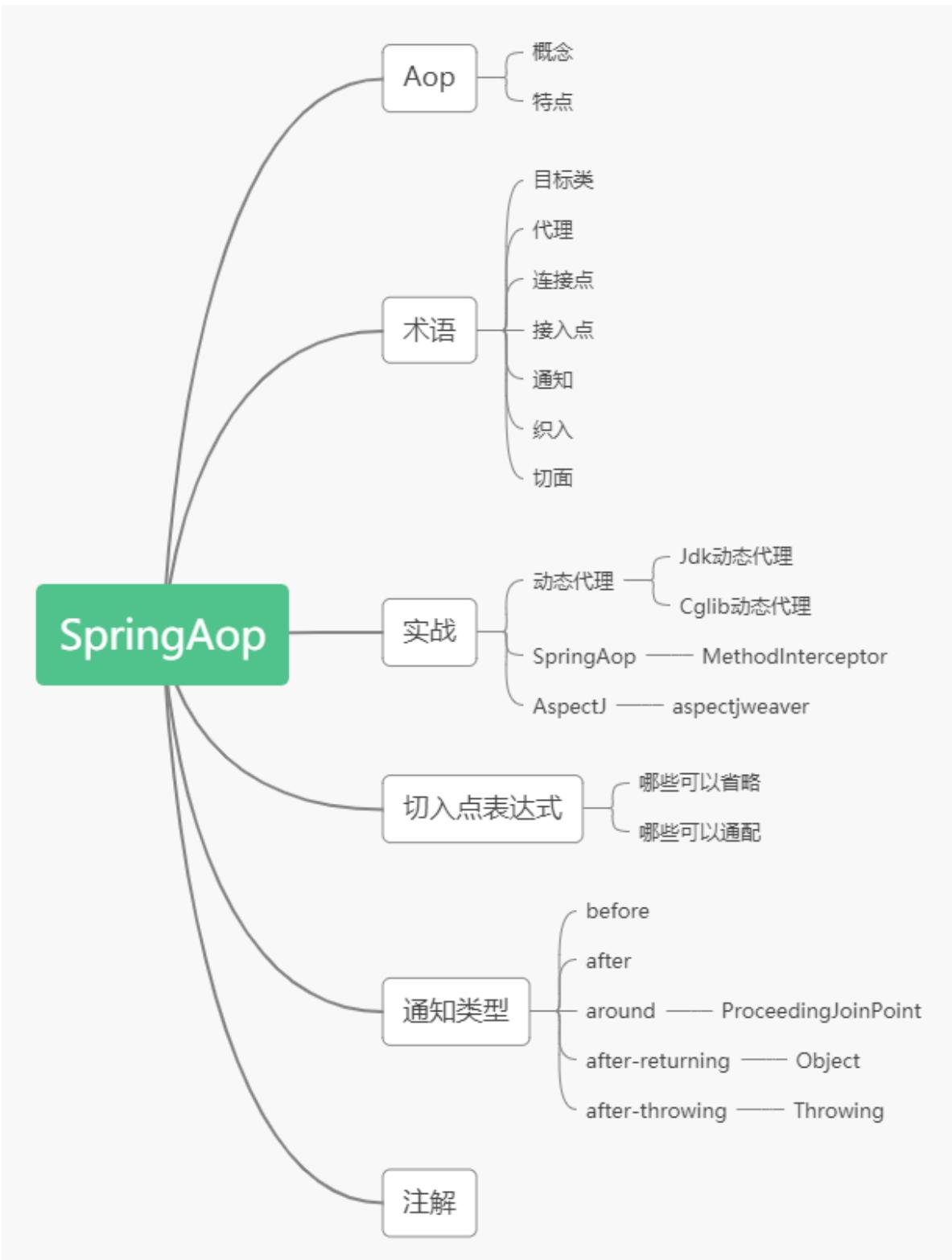
# Week 15

---

## day 1

---

### Spring AOP



## AOP

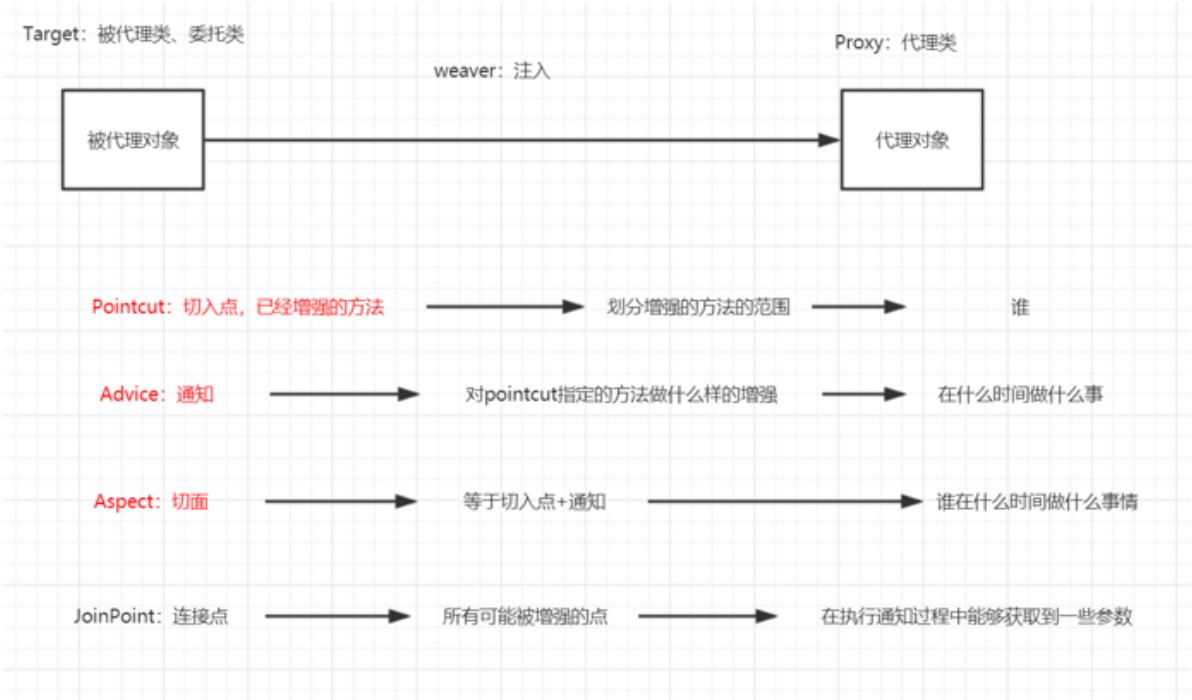
AOP: 基于动态代理实现的

对容器中的组件的方法进行增强

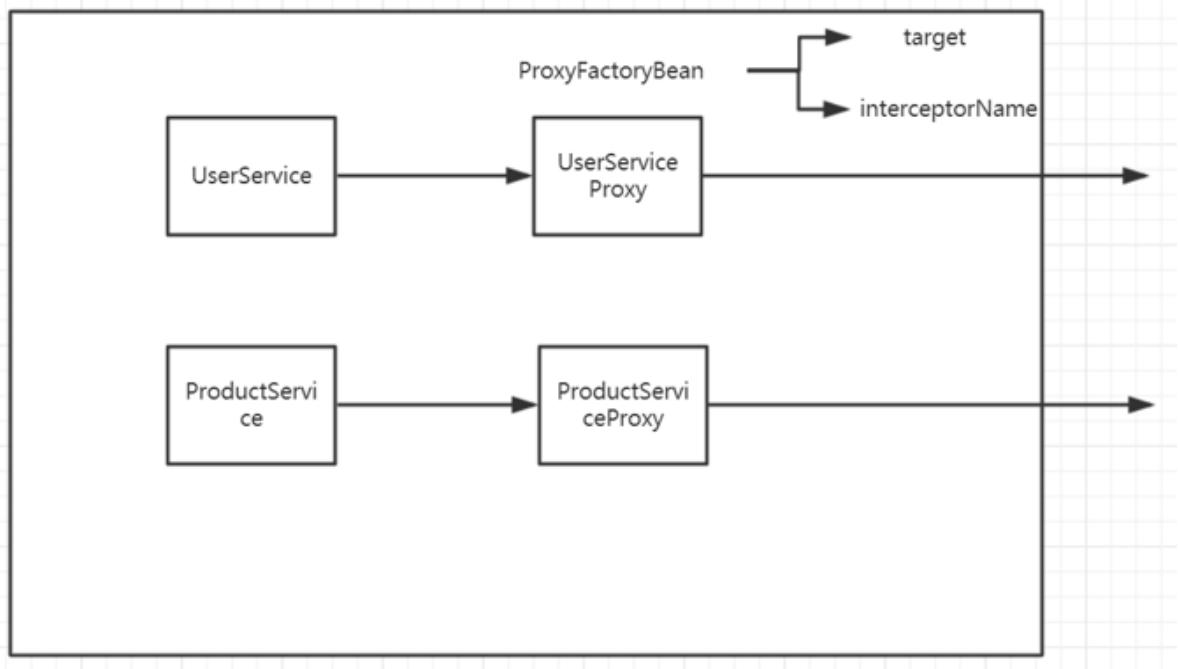
jdk和cglib两个都用

## AOP核心概念

- 1.Target 目标类 (需要被代理的类, 委托类)
- 2.JoinPoint 连接点 指被代理对象里那些可能会被增强的点 (方法) 如所有方法 (候选的可能被增强候选点)
- 3.PointCut 切入点 已经被增强的连接点。
- 4.Advice 通知(具体的增强的代码)。代理对象执行到Joinpoint所做的事情。
- 5.weaver 织入 (植入) 是指把advice应用到目标对象来创建新的代理对象的过程
- 6.Proxy 代理类 (动态代理生成的)
- 7.Aspect 切面 是切入点和通知的结合 (一个线是一条特殊的面->一个切入点和一个通知组成一个特殊的面)



## SpringAOP



代码实现：

使用ProxyFactoryBean：指定被代理对象和切面类 生成 代理对象

```
<bean id="userServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="userServiceImpl"/>
    <!--interceptor组件的id → set接收的是字符串，这里写的是value属性-->
    <property name="interceptorNames" value="customInterceptor"/>
</bean>
@Service
public class UserServiceImpl implements UserService{
    @Override
    public void sayHello() {
        System.out.println("hello songge");
    }
}
@Component
public class CustomInterceptor implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        System.out.println("正道的光");
        Object proceed = methodInvocation.proceed();
        System.out.println("照在大地上");
        return proceed;
    }
}
```

单元测试：

从容器中取出组件的三种方式

- `@Autowired`:按类型取

- `@Autowired`

`@Qualifier("id")`

先按类型再在类型下按id取

- `@Resource(name = "id")`

默认按类型，如果写了name属性就按id取

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:application.xml")
public class SpringAopTest {

    //@Autowired
    //@Qualifier("userServiceProxy")
    //UserService userService;

    //@Resource(name = "userServiceProxy")
    //UserService userService;

    @Autowired
    UserService userServiceProxy;

    @Test
    public void mytest1(){
        //userService.sayHello();
        userServiceProxy.sayHello();
    }
}

```

从容器中指定组件id取出组件

springAOP的缺点：

每一个组件都需要单独使用ProxyFactoryBean生成代理组件，并且每次取出都需要指定组件id(只指定类会报错，因为被代理的类会对应两个对象：一个是被代理对象，一个是代理对象)

## AspectJ

更侧重于划分所需增强的方法的范围

准备：

- 依赖：引入依赖 aspectjweaver (不要忘)

```

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
</dependency>

```

建议使用带org的

- 标签aop
  - 使用已有的
  - 去官网appendix找到
  - 创建模板
  - 自己改造

```

<?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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="*
                           http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">
    <context:component-scan base-package="com.cskaoyan"/>

```

- 切入点表达式：确定哪些方法要被增强

`execution(修饰符 返回值 包名.类名.方法名(形参))`

分析角度：能否省略、能否通配、特殊用法

- 修饰符

可以省略不写：匹配任意的修饰符

- 返回值

不能省略，可以用\*来通配

如果是基本类型、包装类、java.lang目录下的类可以直接写

javabean这些要写全类名如 `com.daxiao.bean.User`

- 包名.类名.方法名

除了头和尾不能省略（即包名的com和方法名不能省略（可以通配）），中间任意一部分都可以省略用 ..

任意层级都可以用\*来通配，表示一个层级或者一个层级的一部分

- 形参

省略表示无参方法

可以使用\*来通配，一个\*表示任意的单个参数，多个参数之间用逗号分隔

可以使用 .. 来表示任意数量任意类型的参数

其中javabean要写全类名

- 代码

```

<!--$d是一个标识，expression写的是切入点表达式，通过切入点表达式指定增强范围-->
<!--修饰符：可以省略不代表任意修饰符-->
<!--返回值：不能省略，可以使用*通配，javabean要写全类名-->
<!--包名、类名、方法名：部分省略，头和尾不能省略，中间部分都可以使用..来省略-->
<aop:pointcut id="customPointcut1" expression="execution(public void com.cskaoyan.service.UserServiceImpl.say*)"/>
<!--中间的任意一部分都可以省略掉-->
<aop:pointcut id="customPointcut2" expression="execution(* com..say*)"/>
<aop:pointcut id="customPointcut3" expression="execution(.. com..service..say*)"/>
<!--可以使用*通配一个层级或一个层级的一部分-->
<aop:pointcut id="customPointcut4" expression="execution(public void *.cskaoyan.service.UserServiceImpl.say*)"/>
<!--返回值：省略不代表无参方法，使用*来通配某个任意类型的参数-->
<aop:pointcut id="customPointcut5" expression="execution(public void *.*.cskaoyan.service.UserServiceImpl.say*(..))"/>
<!--多个参数可以用*, 或者指定为相应类型-->
<aop:pointcut id="customPointcut6" expression="execution(public void *.*.cskaoyan.service.UserServiceImpl.say*(String,Integer))"/>
<!--..代表任意数量的任意类型的参数-->
<aop:pointcut id="customPointcut7" expression="execution(public void *.*.cskaoyan.service.UserServiceImpl.say*(..))"/>
<!--javabean要写全类名-->
<aop:pointcut id="customPointcut8" expression="execution(public void *.*.cskaoyan.service.UserServiceImpl.say*(com.cskaoyan.bean.User,String))"/>

<!--如果我们想要对service层的全部方法都做增强-->
<aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>

```

- AOP标签

[aop:config](#)

`<aop:pointcut id expression/>`

`<aop:advisor advice-ref="自定义通知组件的id" pointcut (-ref) />`

`<aop:aspect ref="切面类组件id">`

`<aop:pointcut`

`<aop:before method="上述的切面类组件中的方法名" pointcut(-ref)`

```
<aop:after  
<aop:around  
<aop:after-returning returning  
<aop:after-throwing throwing
```

## advisor

(通知器)

即自定义的通知类，需实现 MethodInterceptor 接口

- 自定义通知

```
@Component  
public class CustomInterceptor implements MethodInterceptor {  
    @Override  
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {  
        System.out.println("正道的光");  
        Object proceed = methodInvocation.proceed();  
        System.out.println("照在大地上");  
        return proceed;  
    }  
}
```

- advisor配置

```
@Component  
public class CustomInterceptor implements MethodInterceptor {  
<aop:config>  
    <!--id是一个标识，expression 写的是切入点表达式，通过切入点表达式指定增强逻辑-->  
    <aop:pointcut id="customPointcut" expression="execution(public void com.cskaoyan.service.UserService.say*())"/>  
    <!--advice-ref引用容器中的通知组件的id-->  
    <!--pointcut - advisor自己来写切入点表达式-->  
    <!--pointcut-ref → 引用配置过的aop: pointcut的对应id的切入点表达式-->  
    <aop:advisor advice-ref="customInterceptor" pointcut-ref="customPointcut"/>  
</aop:config>
```

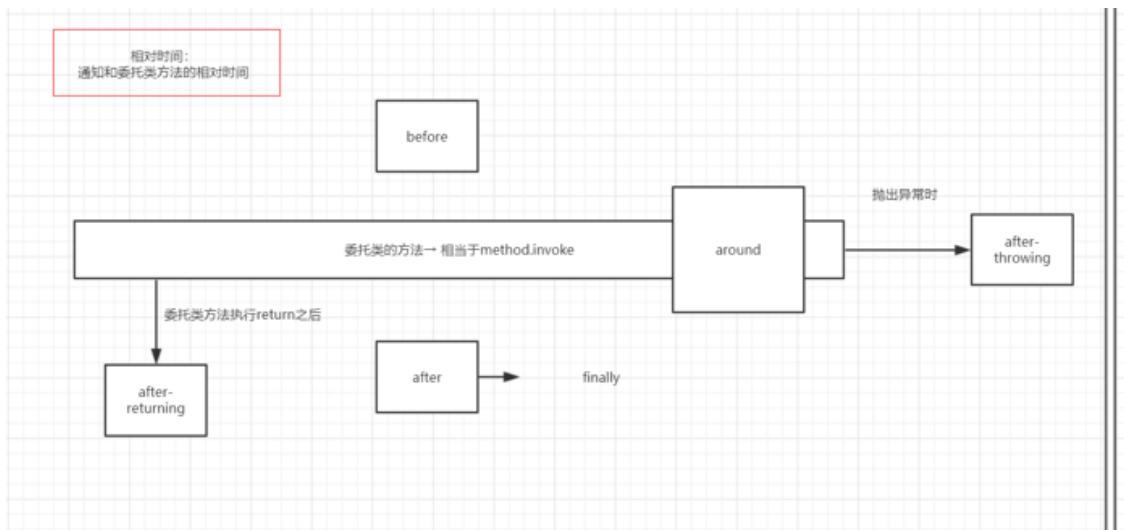
- 单元测试

```
@Autowired  
UserService userService;  
@Test  
public void mytest2() { userService.sayHello(); }  
@Test  
public void mytest3() { userService.sayGoodBye(); }
```

## aspect

用提供的已经定义好的通知：相对于委托类方法的执行时间不同

- 通知



- 定义切面类

- JointPoint (可设置为通知方法中的参数) : 可以通过它获取各种信息

```

public void mybefore(JoinPoint joinPoint){
    String name = joinPoint.getSignature().getName();
    System.out.println(joinPoint.getSignature().getDeclaringTypeName());
    System.out.println("method:" + name);      method相关

    Object[] args = joinPoint.getArgs();
    System.out.println(Arrays.asList(args));      形参

    Object target = joinPoint.getTarget();
    Object aThis = joinPoint.getThis();
    System.out.println(target.getClass());
    System.out.println(aThis.getClass());
    System.out.println("正道的光");
}

```

- 通知

- before

```

public void mybefore(){ 切面类中的方法名
    System.out.println("正道的光");
}

<aop:config>
    <!-- 这里的pointcut #aop: aspect 下的pointcut作用范围不同：全局变量和局部变量-->
    <!--<aop:pointcut id="" expression="" />-->
    <!--ref: 切面类组件id-->
    <aop:aspect ref="serviceAspect">
        <aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>
        <!--method写的是aop: aspect 对应的ref这个组件中的方法名-->
        <aop:before method="mybefore" pointcut-ref="servicePointcut"/>

```

- after: 类似于finally, 无论是否发生异常, 都可以执行到

```

    /**
     * 类似于finally，是一定可以执行到的
     */
    public void myafter(){
        System.out.println("照在大地上");
    }

<aop:config>
    <!--这里的pointcut和aop: aspect下的pointcut作用范围不同：全局变量和局部变量-->
    <!--<aop:pointcut id="" expression="" />-->
    <!--ref: 切面类组件id-->
    <aop:aspect ref="serviceAspect">
        <aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>
        <!--method写的是aop: aspect对应的ref这个组件中的方法名-->
        <aop:before method="mybefore" pointcut-ref="servicePointcut"/>
        <aop:after method="myafter" pointcut-ref="servicePointcut"/>

```

## ■ around

```

    /**
     * param ProceedingJoinPoint
     * return 返回值Object
     */
    public Object myaround(ProceedingJoinPoint joinPoint) throws Throwable {
        joinPoint.getArgs();
        System.out.println("around的前面");
        Object proceed = joinPoint.proceed(); //可以try-catch, 可以抛出异常
        //Object proceed = null;
        //try {
        //    proceed = joinPoint.proceed(); //执行的是委托类的方法
        //} catch (Throwable throwable) {
        //    throwable.printStackTrace();
        //}
        System.out.println("around的后面");
        return proceed;
    }

<aop:config>
    <!--这里的pointcut和aop: aspect下的pointcut作用范围不同：全局变量和局部变量-->
    <!--<aop:pointcut id="" expression="" />-->
    <!--ref: 切面类组件id-->
    <aop:aspect ref="serviceAspect">
        <aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>
        <!--method写的是aop: aspect对应的ref这个组件中的方法名-->
        <aop:before method="mybefore" pointcut-ref="servicePointcut"/>
        <aop:after method="myafter" pointcut-ref="servicePointcut"/>
        <aop:around method="myaround" pointcut-ref="servicePointcut"/>

```

## ■ after-returning: 可以拿到委托类对象的方法返回值

```

    /**
     * @param objectz 需要告诉aspectj, 以形参中的object来接收返回值
     */
    public void myafterReturning(Object objectz){
        System.out.println("委托类执行return结果: " + objectz);
    }

<aop:config>
    <!--这里的pointcut和aop: aspect下的pointcut作用范围不同：全局变量和局部变量-->
    <!--<aop:pointcut id="" expression="" />-->
    <!--ref: 切面类组件id-->
    <aop:aspect ref="serviceAspect">
        <aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>
        <!--method写的是aop: aspect对应的ref这个组件中的方法名-->
        <aop:before method="mybefore" pointcut-ref="servicePointcut"/>
        <aop:after method="myafter" pointcut-ref="servicePointcut"/>
        <aop:around method="myaround" pointcut-ref="servicePointcut"/>
        <!--returning属性的值写的是method属性对应方法的形参名-->
        <aop:after-returning method="myafterReturning" pointcut-ref="servicePointcut" returning="objectz"/>

```

## ■ after-throwing: 可以拿到异常对象

```

// public void myafterThrowing(Exception exception) {
public void myafterThrowing(Throwable exception) {
    System.out.println("抛出了异常: " + exception.getMessage());
}

<aop:config>
<!-- 这里的pointcut 和aop: aspect 下的pointcut作用范围不同：全局变量和局部变量-->
<!--<aop:pointcut id="" expression="/" />-->
<!--ref: 切面类组件id-->
<aop:aspect ref="serviceAspect">
    <aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>
    <!--method写的是aop: aspect 对应的ref这个组件中的方法名-->
    <aop:before method="mybefore" pointcut-ref="servicePointcut"/>
    <aop:after method="myafter" pointcut-ref="servicePointcut"/>
    <aop:around method="myaround" pointcut-ref="servicePointcut"/>
    <!--returning属性的值写的是iomethod属性对应方法的形参名-->
    <aop:after-returning method="myafterReturning" pointcut-ref="servicePointcut" returning="objectz"/>
    <aop:after-throwing method="myafterThrowing" pointcut-ref="servicePointcut" throwing="exception"/>
</aop:aspect>
</aop:config>

```

正道的光

around的前面

制造异常

抛出了异常: by zero

照在大地上 after通知，一定可以执行到

around后面部分

ater-returning部分

均没有执行到

### • 使用注解的形式来使用aspectj(重要)

```

<aop:config>
<!-- 这里的pointcut 和aop: aspect 下的pointcut作用范围不同：全局变量和局部变量-->
<!--<aop:pointcut id="" expression="/" />-->
<!--ref: 切面类组件id-->
<aop:aspect ref="serviceAspect"> 指定了组件为切面类
    <aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/> 定义了切入点表达式
    <!--method写的是aop: aspect 对应的ref这个组件中的方法名-->
    <aop:before method="mybefore" pointcut-ref="servicePointcut"/> 指定了切面组件中的方法对应的通知是什么,
    <aop:after method="myafter" pointcut-ref="servicePointcut"/> 指定了pointcut
    <aop:around method="myaround" pointcut-ref="servicePointcut"/>
    <!--returning属性的值写的是iomethod属性对应方法的形参名-->
    <aop:after-returning method="myafterReturning" pointcut-ref="servicePointcut" returning="objectz"/>
    <aop:after-throwing method="myafterThrowing" pointcut-ref="servicePointcut" throwing="exception"/> 指定了形参中接收返回值和异常的参数名
</aop:aspect>
</aop:config>

```

- 打开切面相关的注解开关

```
<aop:aspectj-autoproxy/>
```

- 切面类组件

  - 定义为切面类

```

/**
 * 类定义上需要增加@Aspect 和@Component注解
 */
@Aspect // 在容器中注册一个组件
public class ServiceAspect {

```

  - 定义切入点表达式

以方法的形式存在，方法名就是表达式的id

@Pointcut注解中的value就是具体的表达式

方法体和形参中不需要写内容

```

<aop:pointcut id="servicePointcut" expression="execution(* com..service..*(..))"/>
@Pointcut("execution(* com..service..*(..))") 切入点表达式
private void servicePointcut(){}

```

■ 通知方法

```

@Pointcut("execution(* com..service..*(..))")
private void servicePointcut(){}
    // 可以引用切入点表达式
//@Before("servicePointcut()")//pointcut-ref="servicePointcut"
@Before("execution(* com..service..*(..))") //pointcut="切入点表达式" 也可以直接写
public void mybefore() { System.out.println("正道的光"); }

/**
 * 类似于finally，是一定可以执行到的
 */
@After("servicePointcut()")
public void myafter() { System.out.println("照在大地上"); }

/**
 * param 参数ProceedingJoinPoint
 * return 返回值Object
 */
@Around("servicePointcut()")
public Object myaround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("around的前面");
    Object proceed = joinPoint.proceed();

    System.out.println("around的后面");
    return proceed;
}

/**
 * @param object 需要告诉AspectJ，以形参中的object来接收返回值
 */
@AfterReturning(value = "servicePointcut()", returning = "object2") 需要指定返回值和异常
public void myafterReturning(Object object2) { System.out.println("委托类执行return结果：" + object2); }

// public void myafterThrowing(Exception exception){
@AfterThrowing(value = "servicePointcut()", throwing = "exception")
public void myafterThrowing(Throwable exception) { System.out.println("抛出了异常：" + exception.getMessage()); }

```

- 使用自定义的注解指定需增强的方法

Pointcut中：之前用切入点表达式(`execution(需作用于的方法)`)，现在用自定义注解  
`@annotation(注解全类名)`

- 自定义注解

```

@Retention(RetentionPolicy.RUNTIME)//注解在什么时候生效
@Target(ElementType.METHOD)//注解写在方法上
public @interface CountTime {
}

```

- Pointcut中使用自定义注解

```

@Aspect
@Component
public class CountTimeAspect {
    // @annotation(自定义注解的全类名)          自定义注解全类名
    @Pointcut("@annotation(com.cskaoyan.anno.CountTime)")
    private void countTimePointcut(){}

    @Around("countTimePointcut()")
    public Object countTimeAround(ProceedingJoinPoint joinPoint){
        long start = System.currentTimeMillis();
        Object proceed = null;
        try {
            proceed = joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        long end = System.currentTimeMillis();
        System.out.println(joinPoint.getSignature().getName() + "执行的时间为:" + (end - start));
        return proceed;
    }
}

@Service
public class UserServiceImpl implements UserService{
    @Override
    @CountTime
    public void sayHello() { System.out.println("hello songge"); }

    @Override
    public void sayHello(String content) { System.out.println("hello " + content); }

    @Override
    @CountTime
    public int add(int a, int b) {
        System.out.println("执行add方法");
        return a + b;
    }

    @Override
    public void createException() {
        System.out.println("制造异常");
        int i = 1/0;
    }
}

```

- 小结：

- 具体的业务用具体的通知，不必全部都使用
- 顺序：只需关注每个通知与委托类方法之间的顺序
- 方法：切入点是用来指定需增强的方法的，必须是容器中组件的方法
- 需要对外提供方法的类才注册到容器中（javabean一般不需要）

## day 2

### JdbcTemplate

着重看组件注册的过程

#### 依赖

mysql-connector-java

spring-jdbc(tx)

druid

## SE代码下的JdbcTemplate

```
@Test  
public void mytest1(){  
    DruidDataSource dataSource = new DruidDataSource();  
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");  
    dataSource.setUrl("jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf8");  
    dataSource.setUsername("root");  
    dataSource.setPassword("123456");  
  
    JdbcTemplate jdbcTemplate = new JdbcTemplate();  
    jdbcTemplate.setDataSource(dataSource);  
  
    String sql = "select name from j23_account_t where id = ?";  
    String s = jdbcTemplate.queryForObject(sql, String.class, ...args: 2);  
    System.out.println("name为" + s);  
}
```

看到new要想要组件注册  
看到set方法要想到property的name属性

## Spring整合

spring-context

spring-test

将datasource和JdbcTemplate注册到容器中

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
    <property name="url" value="jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf-8"/>  
    <property name="username" value="root"/>  
    <property name="password" value="123456"/>  
</bean>  
  
<bean class="org.springframework.jdbc.core.JdbcTemplate">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

## 补充一个autowired注解的使用

加在组件的方法上：在组件的实例化中会调用到该方法

且在形参中可以取出容器中的其他组件

```
// autowired特殊用法  
@Autowired  
private void init(DataSource dataSource) {  
    setDataSource(dataSource);  
}
```

## JdbcDaoSupport的拓展

```

public final void setDataSource(DataSource dataSource) {      datasource发生了变化
    if (this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource()) {
        this.jdbcTemplate = this.createJdbcTemplate(dataSource);
        this.initTemplateConfig();
    }
}

@Repository
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

    // @Autowired
    // DataSource dataSource;

    /**
     * @param dataSource
     * autowired方法的形参是从容器中取出的 → 默认是按照类型的
     * → 如果这个类型的组件在容器中不止一个咋办 → @Qualifier来指定id
     */
    @Autowired
    public void aaaa(@Qualifier("dataSource") DataSource dataSource){
        setDataSource(dataSource);
        System.out.println("bbbbbbbb");
    }
    // @Autowired
    // public void aaaa(DataSource dataSource){
    //     setDataSource(dataSource);
    //     System.out.println("bbbbbbbb");
    // }
    public AccountDaoImpl() { System.out.println("accountDao init"); }
    @Override
    public Integer selectMoneyById(Integer id) {
        String sql = "select money from j23_account_t where id = ?";
        Integer integer = getJdbcTemplate().queryForObject(sql, Integer.class, id);
        return integer;
    }
}

```

## Spring-tx

### 事务回顾

事务特性: ACID

- 原子性
- 一致性
- 隔离性
- 持久性

事务并发执行引起的问题

- 脏读:

一个事务向数据库写数据, 但该事务还没提交或终止, 另一个事务B就看到了事务A写入数据库的数据

- 不可重复读:

一个事务有对同一个数据项的多次读取, 但是在某前后两次读取之间, 另一个事务更新该数据项, 并且提交了。在后一次读取时, 感知到了提交的更新 (隔离性: 感知不到其他事务的执行)

- 幻读:

一个事务需要进行前后两次统计, 在这两次统计期间, 另一个事务插入了新的符合统计条件的记录, 并且提交了, 导致前后两次统计的数据不一致 (平白无故多了一些记录, 像产生幻觉一样)

处理问题：数据库隔离级别

隔离级别	脏读	不可重复读	幻读
读未提交	有	有	有
读已提交	无	有	有
可重复读	无	无	有
串行化	无	无	无

注：mysql中默认的隔离级别为 可重复读，且不会导致幻读

## Spring事务核心接口

**PlatformTransactionManager**: spring事务管理的核心

我们这里使用的是它的实现类DataSourceTransactionManager

```
// 通过事务详情获得事务状态，获取事务状态后，Spring根据传播行为来决定如何开启事务
TransactionStatus getTransaction(TransactionDefinition definition)
// 根据状态提交
void commit(TransactionStatus status)
// 根据状态回滚
void rollback(TransactionStatus status)
```

## TransactionStatus

用于获取事务的状态：

- isNewTransaction():boolean 是否是新的事务
- hasSavepoint():boolean 是否有保存点
- setRollbackOnly():void 设置回滚
- isRollbackOnly():boolean 是否回滚
- flush():void 刷新
- isCompleted():boolean 是否完成

## TransactionDefinition

用于定义事务的名称、隔离级别、传播行为、超时时间长短、只读属性

传播行为：

如何共享事务：包含事务方法之间的相互调用

如：ServiceA -> methodA

ServiceB -> methodB

- required: (最常用)

如果没有事务就新建一个事务，如果有事务就加入进来，看为同一个事务

外层事务与内层事务要么一起提交，要么一起回滚

- requires\_new:

如果没有事务就新建一个事务，如果有就当做一个新的事务（内层事务比较自私）

外层事务发生异常，外层回滚，内层不回滚

内层事务发生异常，内外层都回滚

- nested:

嵌套事务（集体与个人的关系）

外层事务发生异常，内外层都回滚

内层事务发生异常，只有内层回滚

## 事务案例

不管什么方式，都要先有一个平台事务管理器

PlatformTransactionManager → datasource

```
<!--PlatformTransactionManager-->          spring事务的核心：平台事务管理器
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>          依赖于一个数据库连接池
```

准备工作：搭建一个转账业务

- 方法一：手动添加事务：TransactionTemplate(依赖于transactionmanager)

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

<bean class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

```

@Service
public class AccountServiceImpl implements AccountService{

    @Autowired
    TransactionTemplate transactionTemplate;
    @Autowired
    AccountDao accountDao;
    @Override
    public void transfer(Integer fromId, Integer destId, Integer money) {
        Integer fromMoney = accountDao.selectMoneyById(fromId) - money;//转出之后剩余的money
        Integer destMoney = accountDao.selectMoneyById(destId) + money;//接收之后剩余的money

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {
                accountDao.updateMoney(fromId, fromMoney);
                int i = 1/0;
                accountDao.updateMoney(destId, destMoney); 需要增加事务的代码放入到DoInTransaction方法中
            }
        });
    }
}

```

- 方法二：通过TransactionProxyFactoryBean

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:application.xml")
public class TransferTest {

    @Autowired
    @Qualifier("accountServiceProxy")
    AccountService accountService;

    @Test
    public void mytest1() { accountService.transfer( fromid: 1, destId: 2, money: 100); }

<bean id="accountServiceProxy" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <!-- 委托类组件id-->
    <property name="target" ref="accountServiceImpl"/>
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
            <!--key 对应的是方法名， value 对应的是definition
            属性级别: ISOLATION_XXX
            传播行为: PROPAGATION_XXX
            超时: timeout_ 数字 单位是秒
            只读: readOnly
            -XXXException: rollBackFor
            +Exception: noRollBackFor
            -->
            <prop key="transfer">ISOLATION_DEFAULT,PROPAGATION_REQUIRED,readonly</prop>
        </props>
    </property>
</bean>

```

- 方法三：advisor

pointcut: 指定方法增加事务

```

<!-- 使用advisor注意使用织入包-->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
</dependency>

<?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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx" 引入tx约束
       xsi:schemaLocation="http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/tx https://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">

<!--PlatformTransactionManager-->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
<aop:config>
    <aop:pointcut id="txPointcut" expression="execution(* com..service..*(..))"/>
    <aop:advisor advice-ref="transactionInterceptor" pointcut-ref="txPointcut"/>
</aop:config>
<!--spring-tx提供了个通知 → 引入tx标签-->
<tx:advice id="transactionInterceptor" transaction-manager="transactionManager"> 平台事务管理器
    <tx:attributes>
        <!--name属性方法名 definition-->
        <tx:method name="transfer" propagation="REQUIRED" isolation="DEFAULT"/> Definition
    </tx:attributes>
</tx:advice>

```

- 方法四：声明式事务注解（最常用）

注解在哪，哪里就开启事务

- 导入spring-jdbc的依赖
- 在 application.xml 中注册datasource组件和transactionmanager组件

```

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${db.driverClassName}"/>
    <property name="url" value="${db.url}"/>
    <property name="username" value="${db.username}"/>
    <property name="password" value="${db.password}"/>
</bean>

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

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

```

- beans标签属性中加上tx的两行

```

xmlns:tx="http://www.springframework.org/schema/tx"
http://www.springframework.org/schema/tx
https://www.springframework.org/schema/tx/spring-tx.xsd

```

#### 4. 打开事务驱动，使得 @Transactional 注解生效

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

#### 5. 在指定的类或者方法上使用 @Transactional 注解

```
package com.cskaoyan.set1;
import ...
@Service
@Transactional //写在类上意味着当前类全部方法都增加事务
public class AccountServiceImpl implements AccountService {
    @Autowired
    AccountDao accountDao;

    @Override
    @Transactional //指定当前方法增加事务
    public void transfer(Integer fromId, Integer destId, Integer money) {
        Integer fromMoney = accountDao.selectMoneyById(fromId) - money; //转出之后剩余的money
        Integer destMoney = accountDao.selectMoneyById(destId) + money; //接收之后剩余的money
    }
}

String value() default "";
String transactionManager() default "";
Propagation propagation() default org.springframework.transaction.annotation.Propagation.REQUIRED;
Isolation isolation() default org.springframework.transaction.annotation.Isolation.DEFAULT;
int timeout() default -1;
boolean readOnly() default false;
Class<? extends Throwable>[] rollbackFor() default {};
String[] rollbackForClassName() default {};
Class<? extends Throwable>[] noRollbackFor() default {};
String[] noRollbackForClassName() default {};

Set<?> @Transactional(isolation = Isolation.REPEATABLE_READ, propagation = Propagation.REQUIRED, timeout = 5, readOnly = true)
public void transfer(Integer fromId, Integer destId, Integer money) {
    Integer fromMoney = accountDao.selectMoneyById(fromId) - money; //转出之后剩余的money
    Integer destMoney = accountDao.selectMoneyById(destId) + money; //接收之后剩余的money
    accountDao.updateMoney(fromId, fromMoney);
    int i = 1/0;
    accountDao.updateMoney(destId, destMoney);
}
```

## Javaconfig

用java代码来做spring配置(重要)

配置类：用来取代 application.xml

```
@Configuration
@Configuration
public class ApplicationConfiguration { }
```

### 组件注册

bean标签变为配置类中的@Bean注解

```

    /**
     * xml → java代码： bean标签 想到new property子标签想到set方法
     * 方法体里要完成实例的配置
     * 组件id：
     *   1、使用@Bean注解中的value属性指定组件；
     *   2、@Bean注解没有使用value属性，组件id为方法名
     */
    //@Bean("druidDatasource")
    @Bean
    public DataSource dataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf-8");
        dataSource.setUsername("root");
        dataSource.setPassword("123456");
        return dataSource;
    }

    /**
     * 在形参中传入的参数，是从容器中取出的：
     *          1、默认是按照类型取出
     *          2、如果容器中该类型组件不止一个，通过@Qualifier指定组件id
     * @return
     */
    @Bean
    public DataSourceTransactionManager transactionManager(@Qualifier("dataSource") DataSource dataSource){
        DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();
        return transactionManager;
    }

```

@Bean注解的value属性  
 返回值：bean标签的class或其接口  
 方法名：默认组件id  
 形参：从容器中取组件，可以使用@Qualifier指定id  
 方法体：bean标签class属性对应的实例去完成实例化  
 class → new方法  
 property标签 → set方法

## 其他的转化

- 扫描包设置

```

<context:component-scan base-package="com.cskaoyan"/>
@Configuration
@ComponentScan("com.cskaoyan")

```

- 引入properties配置文件

```

<context:property-placeholder location="classpath:db.properties"/>
@Configuration
@ComponentScan("com.cskaoyan")
@PropertySource("classpath:db.properties")

```

- aop:aspectj-autoproxy

```

<aop:aspectj-autoproxy/>
@Configuration
@ComponentScan("com.cskaoyan")
@PropertySource("classpath:db.properties")
@EnableAspectJAutoProxy

```

- 事务驱动

```

<tx:annotation-driven transaction-manager="transactionManager"/>
@Configuration
@ComponentScan("com.cskaoyan")
@PropertySource("classpath:db.properties")
@EnableAspectJAutoProxy
@EnableTransactionManagement

```

- 单元测试

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ApplicationConfiguration.class, XXX.class})
public class AnnotationTest {
}

```

加载配置类

[spring xmind](#)

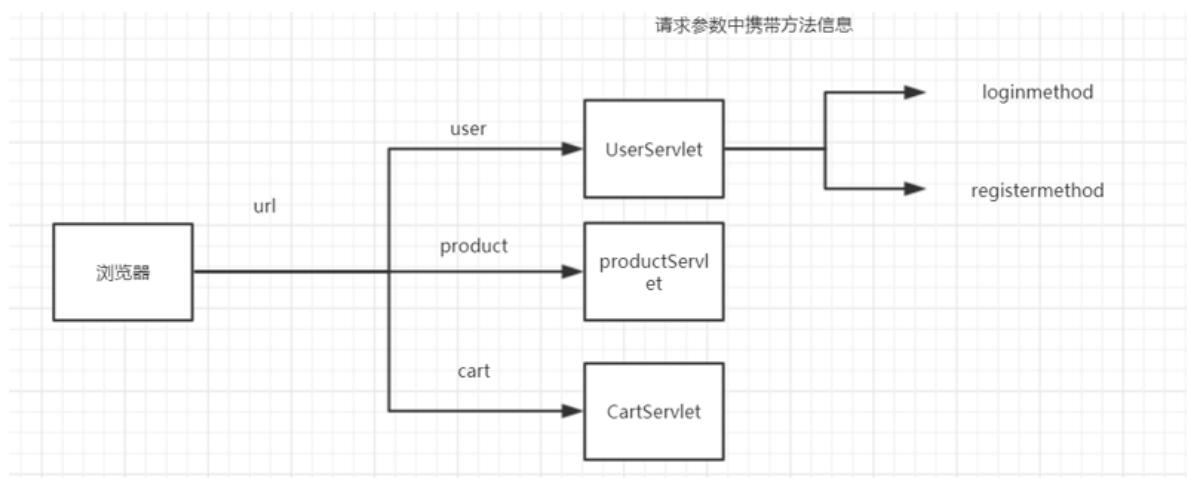
## day 4

### Spring-MVC

在spring框架的基础上使用mvc分层架构来做web应用

原来的MVC架构：

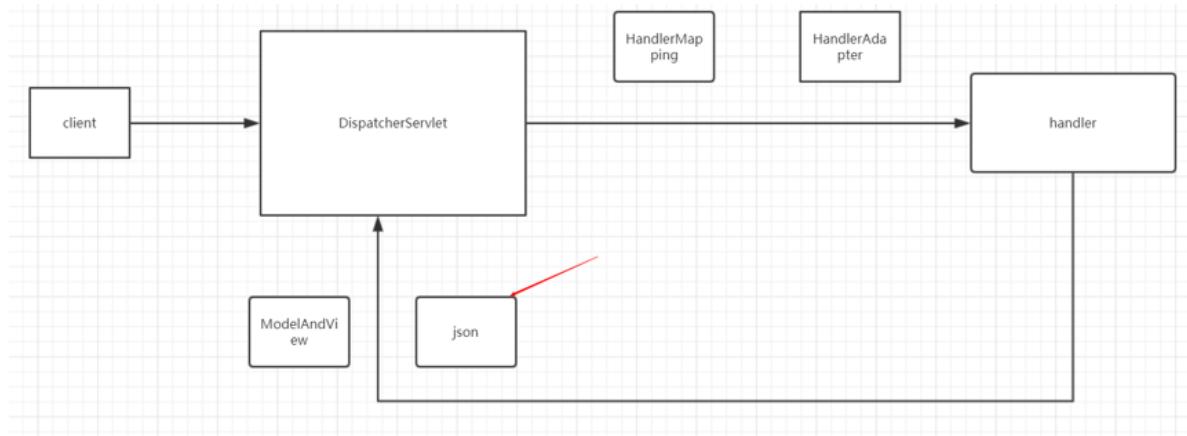
根据url分发到不同的servlet，再分发到具体的方法进行处理



SpringMVC架构：

只有一个全局的DispatcherServlet (在首次访问时加载, 初始化)，再经过映射器 (将url映射到组件的id上) 与适配器(请求头与请求参数)转发到具体的**handler方法**中进行处理

最后handler方法返回 ModelAndView 或者是 json



url -> handler

# SpringMVC入门案例

## 入门案例一

加载spring配置文件，来维护一个spring环境

DispatcherServlet, HandlerMapping, HandlerAdapter

- 导包：5spring + 2mvc + 1logging  
spring-webmvc + servlet-api(javax provided)
- web.xml

配置DispatcherServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app.
    version="4.0">
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:application.xml</param-value>加载Spring配置文件
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>配置映射范围
    </servlet-mapping>
</web-app>
```

- 注册映射器与适配器到容器中

```
<!-- 组件id和url建立映射关系-->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<!-- 简单控制器的适配器-->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
```

- handler注册（这样就跟原来EE的一样）

```
<context:component-scan base-package="com.cskaoyan"/>
@Component("/hello")组件id作为url
public class HelloHandler implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) {
        ModelAndView modelAndView = new ModelAndView();view
        modelAndView.setViewName("/WEB-INF/jsp/hello.jsp");
        modelAndView.addObject("content", "stringValue");model
        return modelAndView;
    }
}
```

- 运行



```
localhost:8080/hello
hello springmvc
@Component("/hello")
public class HelloHandler implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("/WEB-INF/jsp/hello.jsp");
        modelAndView.addObject("content", "stringValue");
        return modelAndView;
    }
}
```

- 分析：

一个url映射到一个handler组件上

handler以组件的形式存在

## 入门案例二

handler以方法的形式存在：即容器中Controller组件中的方法，url现在直接映射到方法上（主要形式）

- spring配置文件 `application.xml`

写 `<mvc:annotation-driven/>` 就相当于在容器中注册了映射器和适配器

controller组件要在扫描包范围内

```
<!--RequestMappingHandlerMapping
    RequestMappingHandlerAdapter
    @RequestMapping 全新注解 handler方法 → 携带url信息
    handler方法要在Controller组件中
-->
<mvc:annotation-driven/>
```

←

- 在Controller中使用 `@RequestMapping` (组件上和方法上都可以用)

```
@Controller
public class UserController {

    @RequestMapping("/login")
    public ModelAndView hello(){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("/WEB-INF/jsp/hello.jsp");
        modelAndView.addObject( attributeName: "content", attributeValue: "springmvc");
        return modelAndView;
    }

    @RequestMapping("register")
    public ModelAndView hello2(){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("/WEB-INF/jsp/hello.jsp");
        modelAndView.addObject( attributeName: "content", attributeValue: "songge");
        return modelAndView;
    }
}
```

不同的url映射到不同的handler方法中

## handler方法的使用

`@Requestmapping`

### url路径映射

将请求url与handler方法建立映射关系：

在方法上的 `@RequestMapping` 注解中使用 `value` 属性

- 将多个请求url映射到同一个handler方法上

因为value是一个字符串数组所以可以设为多个值

```
String[] value() default {};
```

`@RequestMapping(value = {"hello", "hello2", "hell5"})` value属性接收到是数组，可以写多个url

- url可以使用\*来通配，通配一个层级，或者一个层级的一部分

goodbyesongge goodbyelegenli goodbyexxx → goodbye\*

goodbye/songge goodbye/ligenli goodbye/xxx → goodbye/\*

```
@RequestMapping({"goodbye"})
public ModelAndView goodbye1(){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("/WEB-INF/jsp/goodbye.jsp");
    modelAndView.addObject(attributeName: "content", attributeValue: "司徒雷登");
    return modelAndView;
}

@RequestMapping({"goodbye/*"})
public ModelAndView goodbye2(){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("/WEB-INF/jsp/goodbye.jsp");
    modelAndView.addObject(attributeName: "content", attributeValue: "I am");
    return modelAndView;
}
```

## 窄化请求映射

某个Controller专门管理某类请求，如：

user/login handler方法上 → “login”

user/register handler方法上 → “register”

user/logout handler方法上 → “logout”

所以将共同的url前缀 /user 放在Controller组件的 `@RequestMapping` 中

注意：不要将url共同前缀写在@Controller上，那是在写组件的id

```
@Controller("user") 这样写实际上是组件id，并没有窄化请求url,
//@RequestMapping("user") 使用@RequestMapping注解才能窄化
public class ShortenController {
```

## 请求方法限定

通过@RequestMapping中的method属性指定允许的方法，同样可以写多个，满足一个就行

多个请求方法之间的关系是OR

```
@RequestMapping(value = "method/multi", method = {RequestMethod.GET, RequestMethod.POST})  
public ModelAndView multiMethod(){  
    ModelAndView modelAndView = new ModelAndView();  
    modelAndView.setViewName("/WEB-INF/jsp/method.jsp");  
    modelAndView.addObject(attributeName: "method", attributeValue: "GET或POST");  
    return modelAndView;  
}
```

也可以通过注解的形式来做：

```
//@RequestMapping(value = "method/get", method = RequestMethod.GET)  
@GetMapping("method/get")  
  
//@RequestMapping(value = "method/post", method = RequestMethod.POST)  
@PostMapping("method/post")
```

## 请求参数限定

通过params属性限

如果有多个参数，那就是每个参数都要有

也可以用等式关系对请求参数做简单的限定（之后会用validator来做校验）

多个参数之间的关系是and，就是都要有

```
@RequestMapping(value = "login", params = {"username", "password"})  
public ModelAndView login(){  
    ModelAndView modelAndView = new ModelAndView();  
    modelAndView.setViewName("/WEB-INF/jsp/paramlimit.jsp");  
    return modelAndView;  
}  
  
@RequestMapping(value = "login", params = {"username!=songge", "password"})  
public ModelAndView login(){  
    ModelAndView modelAndView = new ModelAndView();  
    modelAndView.setViewName("/WEB-INF/jsp/paramlimit.jsp");  
    对请求参数可以做简单的限定
```

## 请求头限定

通用的限定：通过headers属性

```
@RequestMapping(value = "hello", headers = {"abc", "def"}) 多个请求头之间是and的关系
```

特定的请求头

- produces (属性)
- -> Accept (请求头)

服务器要生产的类型就是浏览器希望接受的类型

```
@RequestMapping(value = "accept", produces = "application/abc")  
public ModelAndView accept(){  
    ModelAndView modelAndView = new ModelAndView();  
    限定了Accept为application/abc
```

- consumers (属性) -> Content-Type (请求头)

服务器要消费的类型就是浏览器发过来的类型

```
@RequestMapping(value = "content/type", consumes = "application/def")限定了Content-Type  
public ModelAndView contentType(){
```

注意：

- 前面都要+application
- 浏览器发送的Accept中有 \*/\* 即通配接收任意内容，所以不会被拦截

## handler方法的返回值

### 处理视图的情况

- void

形参中为request和response，退化为了EE中的servlet

```
@RequestMapping("void")  
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException  
    request.setAttribute("content", "void");  
    request.getRequestDispatcher("/WEB-INF/jsp/void.jsp").forward(request, response);  
}  
  
webapp  
  WEB-INF  
    jsp  
      void.jsp
```

通过request和response处理视图和模型数据  
返回值为void的视图: void

此时的request域就成为了model (model底层就是request域)

- ModelAndView

通过 setViewName 设置视图的路径

通过 addObject 往model中添加<String, Object>

- String：视图名/转发或重定向

- 视图名 (viewName)

■ 物理视图名：即字符串直接作为视图名，model的数据写入Model对象中

```
@RequestMapping("string")  
public String string(Model model){  Model写在形参中  
    model.addAttribute("content", "string");  
    return "/WEB-INF/jsp/string.jsp"; // 字符串返回值作为viewName  
}  
  
webapp  
  WEB-INF  
    jsp  
      string.jsp
```

viewName  
返回值为字符串: string

■ 逻辑视图名

viewName = prefix + 返回值 + suffix

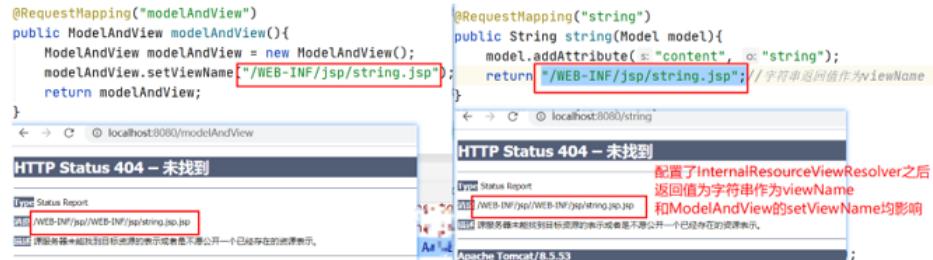
需在容器中注册一个视图解析器的组件，并设置prefix和suffix组件

就是进行一个直接的字符串拼接

```

<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>

```



- 转发和重定向到另一个handler方法，而非转到jsp的视图处理

```

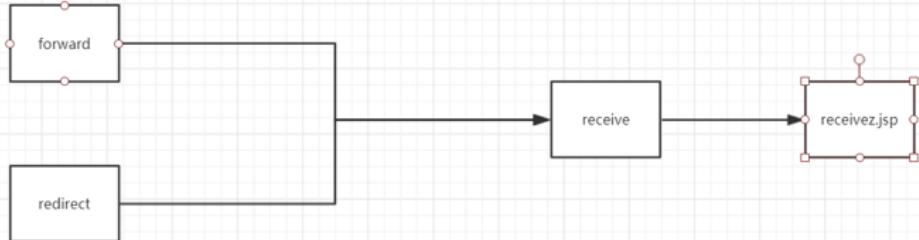
@Controller
public class ForwardRedirectController {

    @RequestMapping("forward")
    public String forward(){
        return "forward:/receive";
    }

    @RequestMapping("redirect")
    public String redirect(){
        return "redirect:/receive";
    }

    @RequestMapping("receive")
    public String receive(){
        return "receivez"; // 这里是作为逻辑视图名使用
    }
}

```



localhost:8080/forward

接收转发和重定向

localhost:8080/receive

接收转发和重定向 做了重定向url会发生变化

视图和转发重定向最前面记得加 / (最好所有的地方都加)

没有加/默认就是相对路径，相对于当前handler方法映射的url按照某个规律来相对

即：去掉请求url的最后一级，加上对应的值

abc/def/hij “WEB-INF/jsp/hello.jsp” → abc/def/ WEB-INF/jsp/hello.jsp

abc “WEB-INF/jsp/hello.jsp” → WEB-INF/jsp/hello.jsp

abc/def/hij “forward:hello” → abc/def/hello

- 直接返回json的情况 (最常用)

handler方法的返回值直接写成javabean、数据、list等类型就可以了

1. 导入依赖: Jackson-databind

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.11.1</version>
</dependency>
```

2. handler方法上增加@ResponseBody

BaseResponseVo的设计思想: 代码重用, 方法重载

```
@Data
public class BaseResponseVo {
    Object data;
    String msg;
    int errno;

    public static BaseResponseVo ok(Object data) {
        BaseResponseVo baseResponseVo = BaseResponseVo.ok();
        baseResponseVo.setData(data);
        return baseResponseVo;
    }

    public static BaseResponseVo ok() {
        BaseResponseVo baseResponseVo = new BaseResponseVo();
        baseResponseVo.setMsg("查询成功");
        baseResponseVo.setErrno(0);
        return baseResponseVo;
    }

    public static BaseResponseVo fail() {
        BaseResponseVo baseResponseVo = new BaseResponseVo();
        baseResponseVo.setMsg("查询失败");
        baseResponseVo.setErrno(500);
        return baseResponseVo;
    }
}
```

---

## handler方法的形参

### 请求参数的封装

form表单中构造请求

```
<h1>通过request获得请求参数</h1>
<form action="/request/parameter" method="get">
    用户: <input type="text" name="username"><br>
    密码: <input type="text" name="password"><br>
    <input type="submit"> form表单中input标签的name属性值,
</form>          就是请求参数名
```

▼ Query String Parameters view source view URL encoded

username: songge  
password: niupi

和input标签的name属性值是一致的

### 1. 通过request取出参数 (不建议)

```
{"data": "songge : niupi", "msg": "查询成功", "errno": 0}
```

▼ Query String Parameters view source view URL encoded  
username: songge  
password: niupi

可以通过request但是不建议

```
@RequestMapping("request/parameter") //request/parameter?username=songge&password=niupi
@ResponseBody
public BaseResponseVo requestParameteter(HttpServletRequest request){
    String username = request.getParameter("username");
    String password = request.getParameter("password");

    return BaseResponseVo.ok(username + " : " + password);
}
```

### 2. handler方法中直接用形参接收

可以直接接受字符串，包装类，基本数据类型（不推荐使用，因为输入参数为空时null无法放入基本数据类型中存储，而包装类可以）

请求参数名（即input标签中的name属性）与handler方法中的形参名对应

字符串，包装类，基本数据类型

**请求参数名和handler方法的形参对应**

```

<h1>形参直接接收请求参数</h1>
<form action="/direct/parameter" method="get">
    用户: <input type="text" name="username"><br>
    密码: <input type="text" name="password"><br>
    年龄: <input type="text" name="age"><br>
    婚否: <input type="text" name="married"><br>
    <input type="submit">
</form>
Query String Parameters view source view URL encoded
username: songge
password: niupi
age: 25
married: true

```

```

/**
 * 直接在handler方法上接收请求参数 → 请求参数名和handler方法的形参名一致
 */
@RequestMapping("direct/parameter")//direct/parameter?username=songge&password=niupi
@ResponseBody
public BaseResponseVo directParameteter(String username, String password, Integer age, boolean married){

    System.out.println(age);
    System.out.println(married = true );
    return BaseResponseVo.ok(username + " : " + password);
}

```

▶ username = "songge"  
▶ password = "niupi"  
▶ age = [Integer@4991] 25  
▶ married = true

```

/**
 * 直接在handler方法上接收请求参数 → 请求参数名和handler方法的形参名一致
 */
@RequestMapping("direct/parameter")//direct/parameter?username=songge&password=niupi
@ResponseBody
public BaseResponseVo directParameteter(String username, String password, Integer age, boolean married){

    System.out.println(age);
    System.out.println(married);
    return BaseResponseVo.ok(username + " : " + password);
}

```

更建议使用包装类来接收  
防止没有传参的情况

**日期格式的接收:** 利用 `@DateTimeFormat`, 在其中指定格式(javabean成员变量上也可以用)

```

@RequestMapping("birthday")      接收Date需要指定格式
@ResponseBody
public BaseResponseVo birthday(@DateTimeFormat(pattern = "yyyy-MM-dd") Date birthday){
    return BaseResponseVo.ok(birthday);
}

```

## 文件的接收

- 导入依赖 commons-fileupload/io
- 注册组件: `CommonsMultipartResolver`(其中组件id必须为multipartResolver)

```

<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
commons-fileupload:commons-fileupload:1.4
commons-io:commons-io:2.2

public static final String MULTIPART_RESOLVER_BEAN_NAME = "multipartResolver";
<!--id是一个固定写法, 不能够写其他值--&gt; id为指定值
&lt;bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"&gt;
    &lt;property name="maxUploadSize" value="512000"/&gt;
&lt;/bean&gt;
</pre>

```

表单:

```

<h1>接收文件格式数据 → 文件上传</h1>
<form action="/file/upload" enctype="multipart/form-data" method="post">
    文件: <input type="file" name="myfile"><br>
    <input type="submit">
</form>

```

请求参数名, 在handler方法中就要有对应的MultipartFile来接收

handler方法：

```
<h1>接收文件格式数据 → 文件上传</h1>
<form action="/file/upload" enctype="multipart/form-data" method="post">
    文件: <input type="file" name="myfile"><br>
    <input type="submit">
</form>
@RequestMapping("file/upload")
@RequestBody
public BaseResponseVo fileUpload(MultipartFile myfile){

    String originalFilename = myfile.getOriginalFilename(); 上传的文件名是啥，获得的就是啥
    File file = new File(parent: "D:\\spring", originalFilename); //用来接收上传的文件
    try {
        myfile.transferTo(file);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return BaseResponseVo.ok();
}
```

## day 5

### handler方法中的形参

可以上传多个文件：

- jsp中：在  标签中添加multiple属性
- handler方法中：接收的参数类型为MultipartFile[] (数组类型)

解决类型转化问题：String -> xxx

使用converter

#### 1. 定义converter组件

定义一个类实现Converter<S, T> 接口，实现其中的convert方法，将其注册为组件

#### 2. 注册conversionService (FormattingConversionServiceFactoryBean)

并设置其中的Converters属性 (set)，引用已经注册的converter组件

```
<bean id="formattingConversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <ref bean="string2DateConverter"/>
        </set>
    </property>
</bean>
```

引用已经注册的converter组件

#### 3. 通知springMVC

```
<mvc:annotation-driven conversion-service="formattingConversionService"/>
```

#### 4. 使用，直接用就行，遇到相应的类型转换会自动调用该转换器

## 接收参数为数组时

- jsp中：多个请求参数名一致时
- handler方法形参中：用数组接收

## 接收参数为JavaBean时

- 请求参数中的参数名要与javabean中的成员变量名对应（实际上是set方法）
- 嵌套javabean：使用`name='userDetail.phone'`来给类型为javabean的成员变量的成员变量复制  
如`name='userDetail.phone'`（前端一般不会这么好）
- 成员变量为list的时候：与嵌套javabean一致，只不过要加下标（很少使用）  
`orderList[i]`就是`order`，实际上是一样的

The diagram illustrates the mapping between JSP form fields, Java Bean properties, and a list of objects.

**JSP Form Fields:**

```
<%--orderList封装--%>
订单1: 名字<input type="text" name="orderList[0].name"><br>
订单1: 价格<input type="text" name="orderList[0].money"><br>
订单2: 名字<input type="text" name="orderList[1].name"><br>
订单2: 价格<input type="text" name="orderList[1].money"><br>
订单3: 名字<input type="text" name="orderList[2].name"><br>
订单3: 价格<input type="text" name="orderList[2].money"><br>
订单4: 名字<input type="text" name="orderList[3].name"><br>
订单4: 价格<input type="text" name="orderList[3].money"><br>
<input type="submit">
```

**Java Bean Structure:**

```
@Data
public class User {
    String username; @Data
    Integer age; public class Order {
        Boolean married; String name;
        Date birthday; Integer money;
        String[] hobbies; }

    UserDetail userDetail;
    List<Order> orderList;
}
```

**Java Bean State:**

```
userDetail.email: songge@cskaoyan.com
orderList[0].name: cd
orderList[0].money: 15
orderList[1].name: basketball
orderList[1].money: 20
orderList[2].name: car
orderList[2].money: 100
orderList[3].name: tesla
orderList[3].money: 2500
```

**Annotations:**

始终请求参数名等于javabean的成员变量名，在list这个成员变量对应的请求参数名后面增加了下标来做分组

## 解决中文乱码问题（主要针对post请求，get请求中请求参数都拼接在url后）

在 web.xml 中设置filter

```
<!--中文乱码问题-->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <!--对应类中的set方法-->
    <!--对应setForceEncoding方法-->
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
    <!--对应setEncoding方法-->
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
```

```

</init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

## 接收json数据

请求报文： POST请求， Content-Type: application/json

- 以JavaBean接收，参数前加 @RequestBody 注解  
涉及到类型转化都会尝试去找相应的converter

```

@RequestMapping("json/receive")
@ResponseBody
public BaseRespVo jsonReceive(@RequestBody User user){
    return BaseRespVo.ok(user);
}

{
    "username": "songge",
    "password": "niupi",
    "age": "5", — 5
    "hobbies": ["sing", "dance", "rap", "basketball"]
}

```

判断是否为数组 (List) 数据, 看是否有中括号[]  
判断为数值还是字符串, 看是否有双引号

利用postman发送请求  
1、方法为POST  
2、Content-Type: application/json  
3、body中选择raw中JSON  
并且封装json数据

```

@Data
public class User {
    String username;
    String password;
    Integer age;
    String[] hobbies;
}

```

- 以Map<String, Object>为形参的形式接收 (不推荐, 因为需要通过参数名get方法才能取出数据)

## 其他形参

- request和response, 返回值为void, 与原来的EE一样
- model, 返回字符串作为viewName
- Cookie和Session
  - Cookie: 需要通过 request.getCookies() 获取
  - Session:
    - 通过request获取
    - 或直接在形参中写 HttpSession 获取

## Restful风格请求

rest: 表示性状态传递, 代码风格, 即响应结果用json

@ResponseBody 可以写在类上, 表示类中所有的handler方法均以json数据响应

其中 @RestController 等价于 @ResponseBody + @Controller

- @PathVariable(url)(常用)

如 user/query/{id}

从请求url的一部分中提取出请求参数

如 `username/article/details/articleId`

可以在RequestMapping中使用{}标明请求参数的位置

在形参中用注解@PathVariable("paramName")进行接收

```
@RequestMapping("{usernamea}/article/details/{articleIdb}")
public BaseRespVo articleDetails(@PathVariable("usernamea") String username,
                                  @PathVariable("articleIdb") String articleId){
    return BaseRespVo.ok(username + ":" + articleId);
}
▶ p username = "songge"
▶ p articleId = "11110000"
```

{}里的值和@PathVariable注解的value属性值对应

- @RequestParam (请求参数) (没什么用)

```
@RequestMapping("admin/login")//?usernamea=xxx&passwordb=xxx
public BaseRespVo login(@RequestParam("usernamea") String username,
                       @RequestParam("passwordb") String password){
    return BaseRespVo.ok();
}
```

请求参数名和@RequestParam注解中的value属性值一致

- @RequestHeader (请求头)

指定请求头中的key可以拿到对应的值，还可以用数组接收

```
Gecko1.0 Chrome/84.0.4147.125 Safari/537.36
@RequestMapping("get/header")
public BaseRespVo getHeader(@RequestHeader("Accept") String accept,
                           @RequestHeader("Host") String host){

    return BaseRespVo.ok();
}
▶ accept = "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9"
▶ host = "localhost:8080"
```

指定请求头中的key可以拿到对应的值

也可以以数组形式接收

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

```
@RequestMapping("get/header")//也可以以数组的形式接收，通过逗号，分割开的
public BaseRespVo getHeader(@RequestHeader("Accept") String[] accept,
                           @RequestHeader("Host") String host){
```

- @CookieValue

可以直接取出某个cookie的key对应的value

```
@RequestMapping("cookie/value")
public BaseRespVo cookieValue(@CookieValue("songge") String value,
                            @CookieValue("ligenli") String value2){
```

- @SessionAttribute

可以直接从session域中按照key取出对应的值

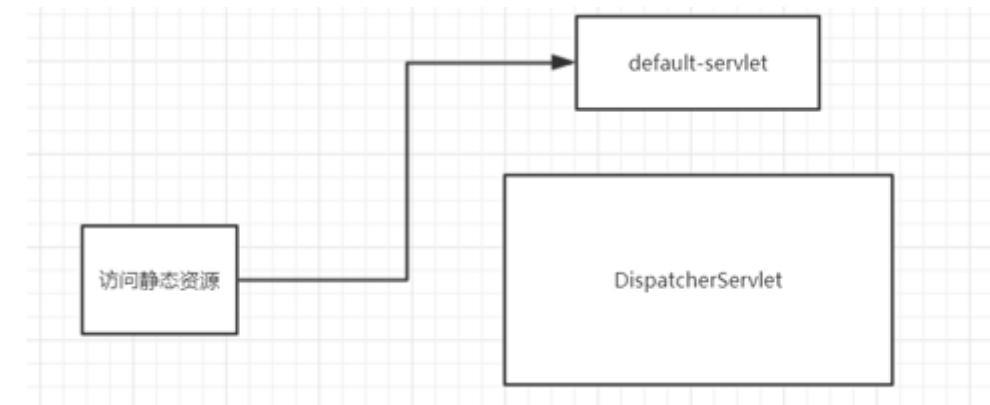
```
@RequestMapping("session/attribute")
public BaseRespVo getSessionAttribute(@SessionAttribute("username") String username){
    return BaseRespVo.ok(username);
}
```

在session中按照这个key取出对应的值

## 静态资源访问

### 1. web.xml中使用默认的servlet

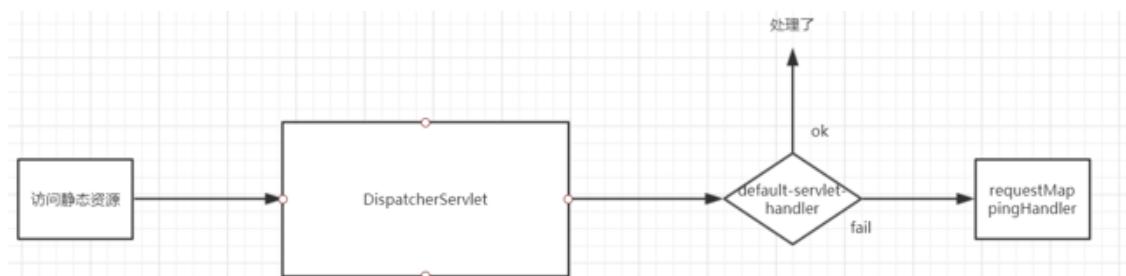
让访问静态资源的请求直接转到默认的servlet中处理



每一个类型的文件都对应一个mapping，映射到默认servlet中（只能到web根目录下）

```
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.jpg</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.png</url-pattern>
</servlet-mapping>
```

### 2. 默认的servlet的handler



请求在经过DispatcherServlet后，首先经过默认的handler判断一下，如果是静态资源就直接处理，否则就放行，然后转到映射器与适配器

在application.xml中添加

```
<mvc:default-servlet-handler/>
```

### 3. 静态资源映射 (推荐)

请求url与文件直接建立映射关系

在application.xml中直接设置

- classpath路径
- web根目录路径
- 文件路径 (最实用, 因为图片这些静态资源一般不放在根目录下)

(有一个专门处理的handler把文件以流的形式写回)

其中的 `**` 表示匹配任意层级任意字符

拼接出来的url = location后面的字符串 + `**`

```
<!--mapping: 请求url-->
<!--location: 文件路径
    1、classpath路径
    2、web根路径
    3、文件路径
-->                                         location最后有个/
<!--mapping 中的值 + 相对于location 的值-->
<mvc:resources mapping="/pic1/**" location="classpath:/img/" /><!--1、classpath路径-->
<mvc:resources mapping="/pic2/**" location="/WEB-INF/img/" /><!--2、web根路径-->
<mvc:resources mapping="/pic3/**" location="file:D:/spring/" /><!--3、文件路径-->
```

## springMVC的异常处理

- HandlerExceptionResolver

返回值是 ModelAndView, 只需注册到容器即可, 所有的异常都会进来

可以通过 instanceof 判断异常类型, 然后进行个性化处理

通过还可以通过自定义的Exception携带一些信息

- ExceptionHandler (异常类型 → 具体的方法)

可以返回 ModelAndView, 也可以返回 json (主流)

按照异常的类型, 映射到相应的handler方法上进行处理

@ControllerAdvice + @ExceptionHandler (value属性中指明要处理的异常类型) +  
@ResponseBody

```
@ControllerAdvice
```

```
public class ExceptionControllerAdvice {
```

```
    @ExceptionHandler({CustomException.class})
```

```
    public String customException(CustomException e){
```

```
        String username = e.getUsername();
```

```
        String message = e.getMessage();
```

```
        return "/WEB-INF/jsp/custom.jsp";
```

```
}
```

视图名

```
    @ExceptionHandler({CustomException2.class})
```

```
    @ResponseBody
```

```
    public BaseRespVo customException2(CustomException2 e){ json
```

```
        return BaseRespVo.fail("songgeniupile");
```

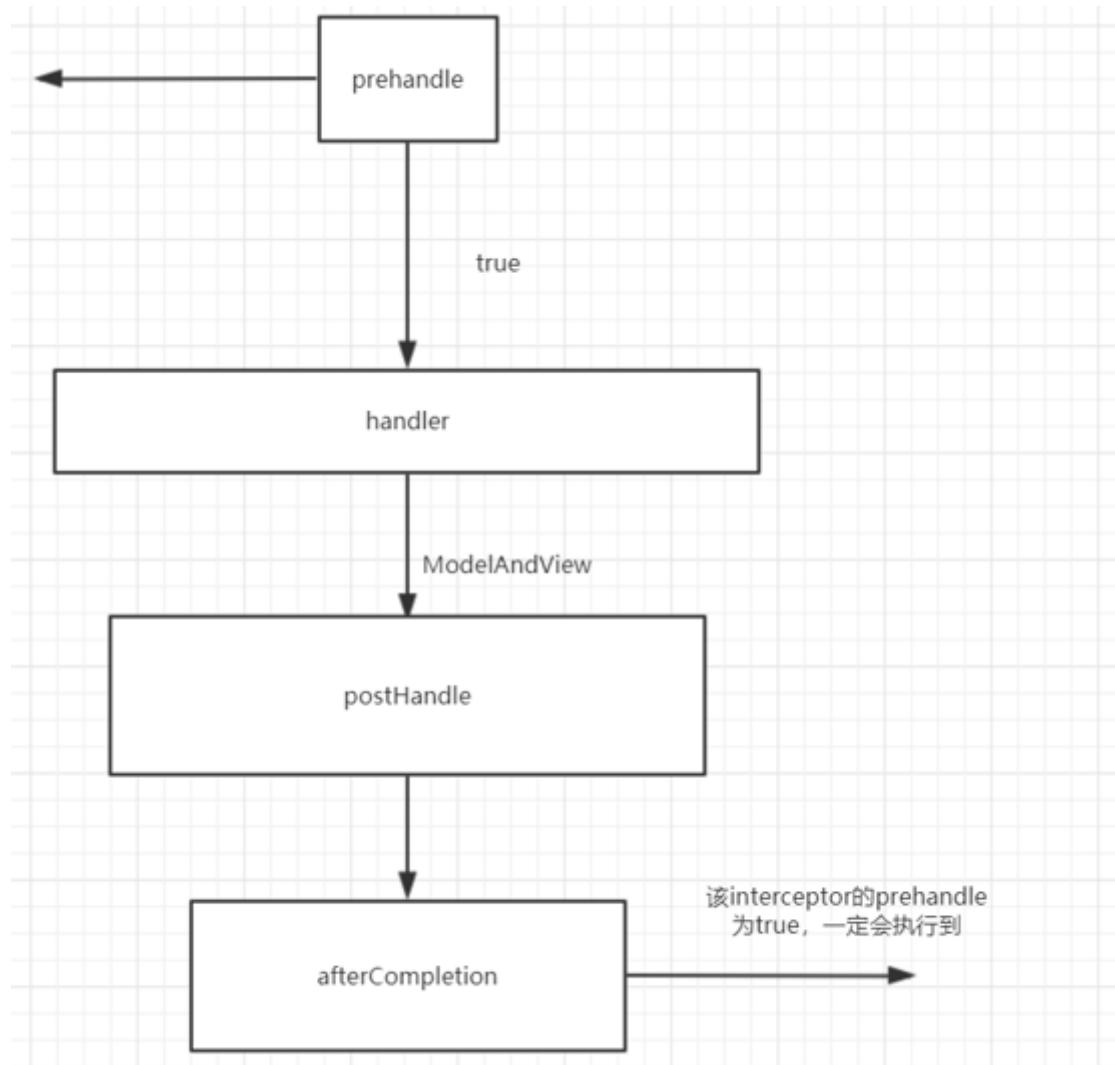
```
}
```

tip: @RestControllerAdvice = @ControllerAdvice + @ResponseBody

## day 6

### HandlerInterceptor

直接访问jsp不进入prehandle: 有默认的\*.jsp的servlet



#### 入门案例1

- 注册HandlerInteceptor组件 (实现接口, 重写方法)

```
@Component  
public class CustomInterceptor implements HandlerInterceptor {  
    @Override
```

- 在 application.xml 中的mvc interceptors中 引用该interceptor组件

```

<mvc:interceptors>
    <!-- 全局-->
    <ref bean="customInterceptor"/>
</mvc:interceptors>

```

- preHandle返回false时，执行不到其作用范围内的handler，及其对应的postHandler和afterCompletion
- postHandle可以做后处理：可以处理视图或者json
- afterCompletion：只要其对应的preHandle为true，就一定能执行到

```

@Override
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {
    //modelAndView.setViewName("/WEB-INF/jsp/post.jsp");
    //如果响应的是json数据，也可以做后处理
    //response.getWriter().append("");
    System.out.println("postHandle");
}

```

如果handler方法上有@ResponseBody  
ModelAndView为null

## interceptor作用范围

- 全局范围(dispatcherServlet的全局)：直接在interceptors标签下写ref（引用已存在的组件）或者bean(新注册)标签

```

<mvc:interceptors>
    <!-- 全局-->
    <ref bean="customInterceptor"/>
    <bean class="com.cskaoyan.interceptor.CustomInterceptor"/>
</mvc:interceptors>

```

- 局部范围：在interceptor子标签中mapping path 中指定需要拦截的url路径

```

<mvc:interceptors>
    <!-- 全局-->
    <!--<ref bean="customInterceptor"/>
    <bean class="com.cskaoyan.interceptor.CustomInterce
    <mvc:interceptor>
        <!--path中填的是url-->
        <!--/hello 只针对hello请求-->
        <!--/hello* 针对helloxxx请求-->
        <!--/hello/* 针对hello/xxx一级任意目录的请求-->
        <!--/hello/** 针对/hello/多级任意目录的请求-->
        <mvc:mapping path="/hello/**"/>
        <ref bean="customInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>

```

## 多个interceptor之间的执行顺序

```

<mvc:interceptors>
    <ref bean="customInterceptor1"/>
    <ref bean="customInterceptor2"/>
    <ref bean="customInterceptor3"/>
</mvc:interceptors>

```

书写顺序，执行顺序

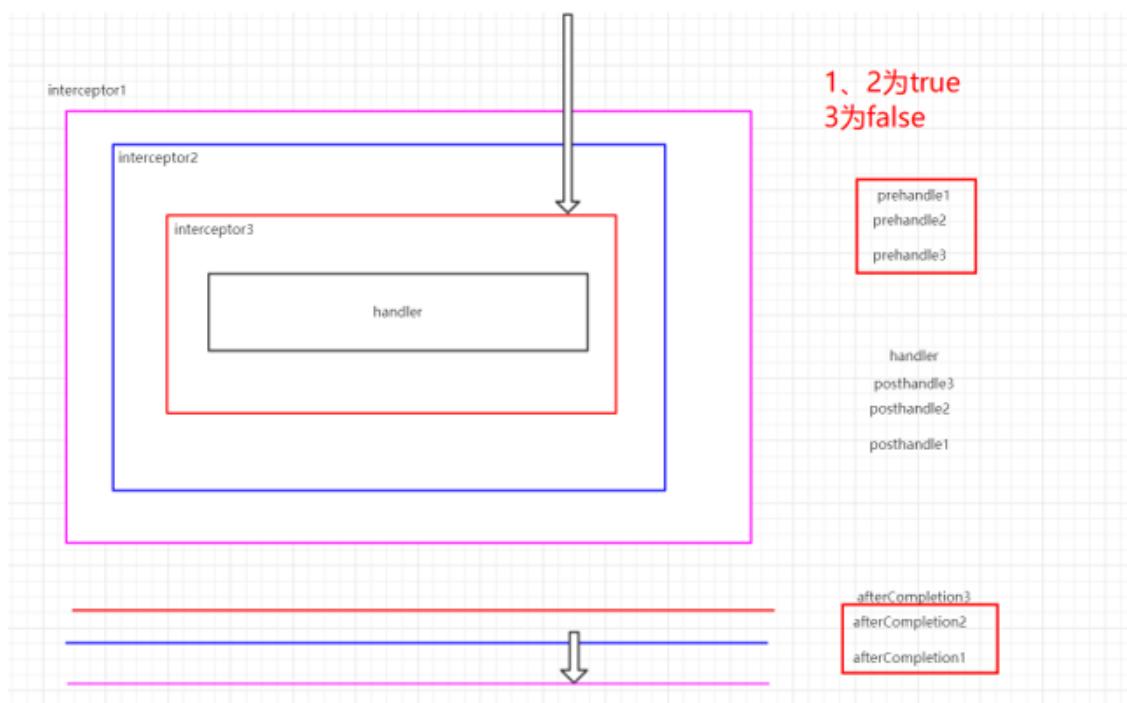
- 若均为true (套娃模型)

123(preHandle) 321(postHandle) 321(afterCompletion)

- 若出现false

preHandle为true，则向下执行，并且可以执行到对应的afterCompletion

若preHandle为false，则中断流程



# Week 16

## day 1

### Locale

根据Locale中的信息，进行区域个性化的处理，如zh\_CN, en\_US

- 可以直接从handler方法的形参中取出（此时拿到的是默认locale）
- 管理Locale
  - 在application.xml中注册CookieLocaleResolver组件
  - 设置其id, cookieName (从cookie中的哪个属性拿到值)  
和defaultLocale的值

```

<!-- 配置10： 小熊修改为共担组-->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="cookieName" value="languagezzz"/>
    <property name="defaultLocale" value="en_US"/>
</bean>
<!--<bean class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>-->

```

Name	Value	Do...	Path	Ex...	Size	Htt...	Sec...	Sa...	Pri...
JSESSIONID	213794EBC447EC385C39EB7D...	loc...	/	Ses...	42	✓			Me...
languagezzz	zh_CN	loc...	/	Ses...	16				Me...
gentle	daqi	loc...	/	Ses...	11	✓			Me...
niupi	shuai	loc...	/	Ses...	12				Me...
songge	niupi	loc...	/	Ses...	11				Me...

## MessageSource

可以通过不同的locale从一个Buddle的配置文件中取出不同的值

- 注册组件

```

<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <!-- 加我的配置文件的名字，注意加上classpath-->
    <property name="basename" value="classpath:message"/>
    <!--properties配置文件的简称-->
    <property name="defaultEncoding" value="utf-8"/> 和你在file encodings里properties配置文件编码相关
</bean>

```

- 使用messageSource (要指定组件id值，因为容器中不止一个MessageSource的组件)

```

@.Autowired
@Qualifier("messageSource") 指定id取出
MessageSource messageSource;

@RequestMapping("hello/{key}")
public BaseRespVo hello(@PathVariable("key") String key){ 第三个参数Locale信息
    //第一个参数 → String → 就是配置文件中的key
    String message = messageSource.getMessage(key, objects: null, Locale.getDefault());
    return BaseRespVo.ok(message);
}

```

- locale与messageSource结合

使得同一个key在不同的locale下，value值不同

创建几个properties文件，使之成为一个bundle

厉害niupi：

en\_US: awesome → message\_en\_US.properties

zh\_CN: 牛皮 → message\_zh\_CN.properties

th\_Th: เน่ → message\_th\_TH.properties

ja\_JP: すごい → message\_ja\_JP.properties

- 组件注册

```

<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <!-- 加载的配置文件的名字, 注意加上classpath-->
    <property name="basename" value="classpath:message"/>
    <!--properties配置文件的编码-->
    <property name="defaultEncoding" value="utf-8"/>
</bean>

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="defaultLocale" value="zh_CN"/>
    <property name="cookieName" value="language"/>
</bean>

```

- 使用 (之前第三个参数使用默认的locale)

```

/**
 * 国际化
 *      1、key从messageSource中取值
 *      2、locale利用LocaleResolver通过cookie中的name取出不同的值
 */
@RequestMapping("i18n/{key}")
public BaseRespVo i18n(@PathVariable("key") String key, Locale locale){
    //第一个参数 → String → 就是配置文件中的key
    String message = messageSource.getMessage(key, objects: null, locale);
    return BaseRespVo.ok(message);
}

```

- 第二个参数Object[]: 在配置文件中充当一个占位符的作用



```

/**
 * 第二个参数object[]
 */
@RequestMapping("niupi/{param}")
public BaseRespVo niupi(@PathVariable("param") String param, Locale locale){
    //第一个参数 → String → 就是配置文件中的key
    String message = messageSource.getMessage("niupi", new Object[]{param}, locale);
    return BaseRespVo.ok(message);
}

```

## Validator

### 请求参数校验 (重要)

前端校验: 只限制了页面中提交的请求, 还可以通过其他手段构造请求

后端校验: 校验逻辑直接和javabean中的成员变量绑定 (要在形参的JavaBean前面加上@Validated)

- 导包

```

<dependency>
    <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator</artifactId>
            <version>6.1.0.Final</version>
</dependency> spring版本号+1

```

- 配置 (注册组件并通知mvc)

```

<mvc:annotation-driven validator="localValidatorFactoryBean">

<bean id="localValidatorFactoryBean" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>
</bean>

```

- 代码 (直接在成员变量上写注解)

The screenshot shows a Java code editor with the following code:

```

@Data
public class User {
    @NotNull
    @Size(min = 6)
    String username;
    @Size(min = 6, max = 10)
    String password;
}

@RequestMapping("login")
//public BaseRespVo login(String username, String password){
public BaseRespVo login(@ValidUser user){
    return BaseRespVo.o...
}

```

A red box highlights the `@NotNull` and `@Size` annotations on the `username` field. Another red box highlights the `@ValidUser` annotation on the `user` parameter. A callout bubble points to the `Validated` option in an auto-complete dropdown, which is part of the `javax.validation` package.

通过和javabean的成员变量绑定，完成对应请求参数的校验

- 常见注解

常见的注解 (Bean Validation 中内置的 constraint)

`@Null` 被注释的元素必须为 null

`@NotNull` 被注释的元素必须不为 null

`@Size(max=, min=)` 被注释的元素的大小必须在指定的范围内

`@AssertTrue` 被注释的元素必须为 true

`@AssertFalse` 被注释的元素必须为 false

`@Min(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

`@Max(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

`@DecimalMin(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

`@DecimalMax(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

`@Digits(integer, fraction)` 被注释的元素必须是一个数字，其值必须在可接受的范围内

`@Past` 被注释的元素必须是一个过去的日期 Date

`@Future` 被注释的元素必须是一个将来的日期

`@Pattern(regex=, flag=)` 被注释的元素必须符合指定的正则表达式

#### Hibernate Validator 附加的 constraint

`@NotBlank(message =)` 验证字符串非null，且长度必须大于0

`@Email` 被注释的元素必须是电子邮箱地址

@Length(min=,max=) 被注释的字符串的大小必须在指定的范围内

@NotEmpty 被注释的字符串的必须非空

@Range(min=,max=,message=) 被注释的元素必须在合适的范围内

- message
  - 自定义错误消息 (message在注解定义中有默认值, 在注解使用时可以覆盖掉)

```
@Data
public class User {
    @NotNull
    @Size(min = 6,message = "username length min 6")
    String username;
    @Size(min = 6,max = 10,message = "password length between 6 and 10")
    String password;
}

/**
 *
 * @param bindingResult 参数校验的结果 → 哪一个参数发生错误了, 发生的错误是什么
 */
@RequestMapping("login")
//public BaseRespVo login(String username, String password){
public BaseRespVo login(@Valid User user, BindingResult bindingResult){
    if (bindingResult.hasFieldErrors()){
        FieldError fieldError = bindingResult.getFieldError();
        String field = fieldError.getField();
        Object rejectedValue = fieldError.getRejectedValue();
        String defaultMessage = fieldError.getDefaultMessage();
        String message = "参数" + field + "携带了" + rejectedValue + "没有完成校验: " + defaultMessage;
        return BaseRespVo.fail(message);
    }
    return BaseRespVo.ok();
}
```

自定义错误消息

- 和messageSource结合

```
<mvc:annotation-driven validator="localValidatorFactoryBean"/>

<bean id="localValidatorFactoryBean" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>
    <property name="validationMessageSource" ref="messageSource"/>
</bean>

<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:message"/>
    <property name="defaultEncoding" value="utf-8"/>
</bean>

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="cookieName" value="language"/>
    <property name="defaultLocale" value="zh_CN"/>
</bean>
```

```
@Data
public class User {
    @NotNull
    // @Size(min = 6,message = "username length min 6")
    @Size(min = 6,message = "{error.username}")
    String username;
    // @Size(min = 6,max = 10,message = "password length between 6 and 10")
    @Size(min = 6,max = 10,message = "{error.password}")
    String password;
}
```

这里前面不用加\$

大括号里写配置文件中的key

- 如果不做国际化, localeResolver可以不注册

```

<mvc:annotation-driven validator="localValidatorFactoryBean"/> 如果不做国际化

<bean id="localValidatorFactoryBean" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>
    <property name="validationMessageSource" ref="messageSource"/>
</bean>

<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:message"/> 配置文件只创建message.properties
    <property name="defaultEncoding" value="utf-8"/>
</bean>

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="cookieName" value="language"/>
    <property name="defaultLocale" value="zh_CN"/>
</bean>

```

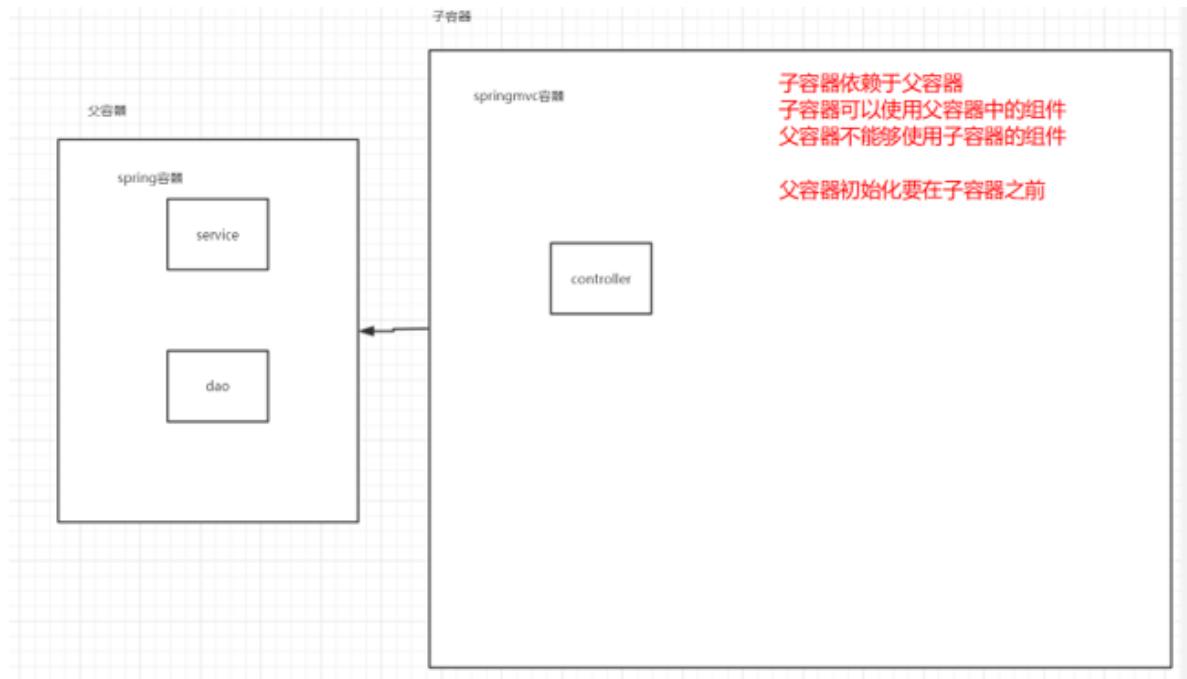
这个组件可以不注册

## Spring和SpringMVC的整合

### Spring容器与SpringMVC容器的分离

- 父容器：Spring，通过listener（context（应用）加载的时候启动）来加载Spring对应的配置文件：applicaiton.xml
- 子容器：SpringMVC，通过DispatcherServlet加载 application-mvc.xml

子容器依赖于父容器，子容器可以使用父容器中的组件，父容器初始化要在子容器之间



- ContextLoaderListener：在应用挂载时加载Spring容器

```

<!--contextLoaderListener 加载spring配置文件初始化spring容器-->
<!--并且是在DispatcherServlet 初始化之前-->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:application.xml</param-value>
</context-param>

```

Spring配置文件加载

- DispatcherServlet：在第一次被访问时加载SpringMVC容器

```

<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:application-mvc.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

```

- context:component-scan:把controller从spring容器中移除出去 (不然就冗余了)

```

<!--Spring 配置文件 -->
<context:component-scan base-package="com.cskaoyan">
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

注解的全类名  
按照注解的类型排除 移除的是controller注解标注的组件

## SpringMVC JavaConfig

即在配置类中注册组件

spring配置类 替代 application.xml

springMVC配置类 替代 applicaiton-mvc.xml

AACDSI 替代 web.xml 去加载spring和springMVC的配置类

**AACDSI(不用注册为配置类或者组件)**

重写父类中的方法

ApplicationInitializer要继承AACDSI这个抽象类 (替代web.xml)

```

/**
 * 替换web.xml
 * AACDSI -> 重写父类中的方法
 */
public class ApplicationInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfiguration.class}; Spring配置类加载
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{MvcConfiguration.class}; SpringMVC配置类的加载
    }

    /**
     * DispatcherServlet的servlet-mapping 中的 url-pattern
     */
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}

```

要写@Configuration和@ComponentScan注解

```
<!--Spring配置文件 → Spring 容器-->
<context:component-scan base-package="com.cskaoyan">
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

@Configuration
@ComponentScan(value = "com.cskaoyan",
    excludeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = Controller.class))
public class SpringConfiguration {
}

@Configuration
@ComponentScan(value = "com.cskaoyan",
    excludeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = {Controller.class, EnableWebMvc.class}))
public class SpringConfiguration {
}

```

配置类中的注解对应标签

MVC的配置类也在spring配置类的扫描包范围内，  
需要排除出去

需要将Controller和EnableWebMvc注解下的类排除出去

保证spring容器中只有spring相关的组件 (service和repository)

这样就将spring容器和springMVC容器分家了

## SpringMVC配置类

要有@EnableWebMvc和@ComponentScan这两个注解

并实现WebMvcConfigurer这个接口(所有之前需要通知mvc的注解，都是这个接口里面的方法 (默认方法，需要自己去重写) )

@EnableWebMvc and <mvc:annotation-driven /> have the same purpose

```
/**
 * 实现这个WebMvcConfigurer接口 → springmvc的组件配置
 */
@EnableWebMvc
@ComponentScan("com.cskaoyan.controller")
public class MvcConfiguration implements WebMvcConfigurer {
}
```

## 其他组件注册

- CharacterEncodingfilter

原来是在web.xml中，现在是在AACDSI

```

/**
 * @WebXml
 * AACDSI + 将所有注解
 */
public class ApplicationInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfiguration.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() { return new Class[]{MvcConfiguration.class}; }

    /**
     * DispatcherServlet的servlet-mapping中url-pattern
     */
    @Override
    protected String[] getServletMappings() { return new String[]{"/"}; }

    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter filter = new CharacterEncodingFilter();
        filter.setEncoding("utf-8");
        filter.setForceEncoding(true);
        return new Filter[]{filter};
    }
}

```

CharacterEncodingFilter  
重写父类中的方法

- 静态资源映射

之前: <mvc: resources mapping location/>

```

//静态资源映射
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    //mapping → addResourceHandler
    //location → addResourceLocations
    registry.addResourceHandler("pathPatterns: "/pic1/**").addResourceLocations("file:D:/spring/");//文件路径
    registry.addResourceHandler("pathPatterns: "/pic2/**").addResourceLocations("classpath:/"); //classpath:java/resources
    registry.addResourceHandler("pathPatterns: "/pic3/**").addResourceLocations("//"); //webroot → webapp
}

```

- interceptor

之前:

mvc:interceptors

<bean或ref标签  
mvc:interceptor  
<mvc:mapping path/>  
<ref或bean标签

```

//interceptor的配置
@Override
public void addInterceptors(InterceptorRegistry registry) {
    //interceptor配置: addInterceptor
    //作用范围: addPathPatterns
    registry.addInterceptor(new CustomInterceptor()).addPathPatterns("/hello/**");
}

```

- multipartResolver (导包 + 注册组件 (bean注解))

```

commons-fileupload (io) ←

<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>

@Bean
public CommonsMultipartResolver multipartResolver(){ 组件id是固定值
    return new CommonsMultipartResolver();
}

```

- localeResolver

```

@Bean("localeResolver")
public CookieLocaleResolver localeResolver(){ 组件id也是固定值
    CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
    cookieLocaleResolver.setCookieName("language");
    cookieLocaleResolver.setDefaultLocale(Locale.US);
    return cookieLocaleResolver;
}

```

- viewResolver

```

@Bean
public InternalResourceViewResolver internalResourceViewResolver(){
    InternalResourceViewResolver internalResourceViewResolver = new InternalResourceViewResolver();
    internalResourceViewResolver.setPrefix("/WEB-INF/jsp/");
    internalResourceViewResolver.setSuffix(".jsp");
    return internalResourceViewResolver;
}

```

- validator

```

<mvc:annotation-driven validator="localValidatorFactoryBean"/> 1. 注册validator组件
<bean id="localValidatorFactoryBean" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"> 2. 通知SpringMVC使用validator
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>
    <property name="validationMessageSource" ref="messageSource"/>
</bean>

//@Bean
public Validator localValidatorFactoryBean(){
    LocalValidatorFactoryBean localValidatorFactoryBean = new LocalValidatorFactoryBean();
    localValidatorFactoryBean.setProviderClass(HibernateValidator.class);
    return localValidatorFactoryBean;
}
//@Autowired
//Validator validator; 也可以使用@Bean和@Autowired搭配的方式
@Override
public Validator getValidator() {
    //return validator;
    Validator validator = localValidatorFactoryBean();
    return validator;
}

```

- converter

对于conversionService: 取出、给它增加上converter (PostConstruct方法中) 、放回容器

@Primary表示容器中如果有两个以上的同类组件，以这个为准

```

//converter
@Autowired
ConfigurableConversionService conversionService; 1、取出来

@PostConstruct
public void addConverters(){
    conversionService.addConverter(new String2DateConverter());
}

@Bean
@Primary
public ConfigurableConversionService conversionService(){
    return conversionService;
} 3、放回去

```

- handlerExceptionResolver

```

@Component      只需要注册到容器中
public class CustomHandlerExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
                                         HttpServletResponse httpServletResponse, Object o, Exception e) {
        return null;
    }
}

```

- messageSource: bean标签变为bean注解即可

## Spring阶段的组件

- jdbcTemplate

导包:

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>

```

```

    org.springframework:spring-jdbc:5.2.8.RELEASE
        org.springframework:spring-beans:5.2.8.RELEASE (omitted for duplicate)
        org.springframework:spring-core:5.2.8.RELEASE (omitted for duplicate)
    ▶ org.springframework:spring-tx:5.2.8.RELEASE

```

```

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.22</version>
</dependency>

```

```

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>

```

在spring配置类中注册

```
@Bean  
public DataSource druidDatasource(){  
    DruidDataSource dataSource = new DruidDataSource();  
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");  
    dataSource.setUrl("jdbc:mysql://localhost:3306/j23_db");  
    dataSource.setUsername("root");  
    dataSource.setPassword("123456");  
    return dataSource;  
}  
  
@Bean  
public JdbcTemplate jdbcTemplate(DataSource dataSource){  
    return new JdbcTemplate(dataSource);  
}
```

- transactionmanager

```
@Configuration  
@ComponentScan(value = "com.cskaoyan",  
    excludeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION,classes = {Controller.class, EnableAspectJAutoProxy.class}),  
    @EnableTransactionManagement)  
public class SpringConfiguration {  
  
    @Bean  
    public DataSource druidDatasource(){  
        DruidDataSource dataSource = new DruidDataSource();  
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");  
        dataSource.setUrl("jdbc:mysql://localhost:3306/j23_db");  
        dataSource.setUsername("root");  
        dataSource.setPassword("123456");  
        return dataSource;  
    }  
  
    @Bean  
    public JdbcTemplate jdbcTemplate(DataSource dataSource) 来源于容器 容器中这个类型的组件只有一个  
    {  
        return new JdbcTemplate(dataSource);  
    }  
  
    @Bean  
    public DataSourceTransactionManager transactionManager(DataSource dataSource){  
        return new DataSourceTransactionManager(dataSource);  
    }  
}
```

- aspectj

第一步：导包

```
<dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjweaver</artifactId>  
    <version>1.9.5</version>  
</dependency>  
  
@Configuration  
@ComponentScan(value = "com.cskaoyan",  
    excludeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION,classes = {Controller.class, EnableAspectJAutoProxy.class}),  
    @EnableTransactionManagement  
    @EnableAspectJAutoProxy)  
public class SpringConfiguration {
```

# Mybatis

## 介绍

是一个持久层的ORM框架

对象 关系 映射

一个user对象就对应关系表中的一条记录

user对象 -> insert -> 数据库中的记录 (操作的单位都是对象)

数据库中的记录 -> select -> user对象

需要把sql语句进行集中管理

之前：sql语句与java代码耦合在一起，解耦后sql语句与java代码分开

集中管理的地方：映射文件，不同分类的sql语句放到不同的映射文件中

重要特性：

动态sql：根据条件的不同，执行不同的sql语句

高级映射：

- 输入映射：为sql语句提供参数
- 输出映射：对结果集中的字段的封装

---

## 入门案例一

sql放入到映射文件中集中管理了：需要确定执行的是哪个映射文件中的哪一个sql语句

通过映射文件id+ sql语句的id找到唯一的sql语句

这就意味着映射文件id不能重复，同一个映射文件中sql语句id不能重复

一个映射文件就是一个namespace

- 导包

mybatis

mysql-connector-java (获取connection, mybatis有自己的datasource)

junit

```

<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.5</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

- mybatis配置文件

xml文件中必须配置的项: datasource(获取connection)和mappers (sql语句)

- 映射文件 如userMapper.xml #{}: 在select标签中代表?

```

<?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">
<!--namespace就是该映射文件的唯一id, 映射文件的namespace(命名空间)不能重复--&gt;
&lt;mapper namespace="userNamespace"&gt;          约束
&lt;/mapper&gt;                                     映射文件的id → 唯一性 → 不同的映射文件的namespace不能相同
</pre>

```

- 通过映射文件的namespace和sql语句的id找到所需的sql语句

通过sqlSession对象

SqlSessionFactoryBuilder (通过输入流获取mybatis.xml信息) -> SqlSessionFactory ->  
SqlSession (必须保证每个线程用的sqlSession是不同的)

```

<!--namespace就是该映射文件的唯一id，映射文件的namespace(命名空间)不能重复-->
<mapper namespace="userNamespace">
    <select id="selectUsernameById" resultType="string">
        select username from j23_user_t where id = #{id}
    </select>
</mapper>

@Test
public void mytest() throws IOException {
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //加载classpath下的mybatis的配置文件
    InputStream inputStream = Resources.getResourceAsStream("mybatis.xml");
    SqlSessionFactory factory = builder.build(inputStream);

    SqlSession sqlSession = factory.openSession();
    String username = sqlSession.selectOne("userNamespace.selectUsernameById", o: 1);
    System.out.println("username = " + username);
}

```

String, Object  
查询结果的记录数只有一条，使用selectOne  
为sql语句提供参数

- SqlSession的几个其他方法

## 查询 select

mapper中的select标签

`resultType`: javabean 对象关系映射 在select标签中不能省略

查询结果为javabean时：

对象	关系表										
<pre> @Data public class User {     Integer id;     String username;     String password;     Integer age;     String gender; } </pre>	<table border="1" style="width: 100px; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">id</th> <th style="text-align: center;">username</th> <th style="text-align: center;">password</th> <th style="text-align: center;">age</th> <th style="text-align: center;">gender</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">songge</td> <td style="text-align: center;">niupi</td> <td style="text-align: center;">20</td> <td style="text-align: center;">male</td> </tr> </tbody> </table>	id	username	password	age	gender	1	songge	niupi	20	male
id	username	password	age	gender							
1	songge	niupi	20	male							

```

public void mytest2() throws IOException {
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    InputStream resourceAsStream = Resources.getResourceAsStream("mybatis.xml");
    SqlSessionFactory sqlSessionFactory = builder.build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();

    //返回值类型和resultType对应
    User user = sqlSession.selectOne("userNamespace.selectUserById", o: 1);
    System.out.println("user = " + user);
}

<!--查询单条user这个javabean-->
<!--resultType是查询必须的 → 不能省略-->
<!--resultType javabean类型 → javabean的全类名-->
<select id="selectUserById" resultType="com.cskaoyan.bean.User">
    select id,username,password,age,gender from j23_user_t where id = #{id}
</select>    数据库表的列名和javabean的成员变量名 目前是一致的，直接封装

```

注：lombok: scope是provided 只在编译的时候使用 用来加入一些java代码

查询结果为List时：select标签中的resultType仍然为javabean

(mybatis底层默认查询返回结果都是list, selectOne时通过get(0)得到单个结果)

```
@Test  
public void mytest3(){  
    List<User> objects = sqlSession.selectList( s: "userNamespace.selectUsers");  
}  
  
<!-- 查询单条user这个javabean--&gt;<br/><!-- resultType是查询必须的 → 不能省略--&gt;<br/><!-- resultType javabean类型 → javabean的全类名--&gt;<br/><select id="selectUserById" resultType="com.cskaoyan.bean.User">  
    select id,username,password,age,gender from j23_user_t where id = #{id}  
</select>  
  
<!-- 查询List-->  
<!-- 如果是查询多条数据, resultType 依然写的是单条记录对应的类型-->  
<select id="selectUsers" resultType="com.cskaoyan.bean.User">  
    select id,username,password,age,gender from j23_user_t  
</select>
```

相对于查询单条记录，方法不同  
查询结果的记录数  
→ 单条 0或1  
→ 多条 ≥0

注意：

- junit中的before和after方法  
在每个测试方法执行前后都会执行
- beforeclass afterclass方法  
在所有的测试方法执行前和执行后执行
- 改造单元测试：  
sqlsession不是线程安全的 所以需要每次使用都从factory中获取，  
保证每个线程中所用的sqlSession都是不同的  
所以before中放得到sqlSession的动作  
after方法中放入commit和close方法 (操作之前先判断是否为null)  
又因为只需要获得一次Factory，所以获得factory的代码可以放在beforeClass中

## 新增 insert

要写table(具体的字段名) 因为表的字段可能发生变化

#{} 里面填javabean成员变量的名字 实际上是get方法

需要手动commit才能生效

```
<!--新增insert-->  
<!--传入的第二个参数object为user 这样的一个javabean对象 → #{} → javabean的成员变量名  
(get方法) -->  
<insert id="insertUser" >  
    insert into j23_user_t (username,password,age,gender)  
    values  
    (#{username},#{password},#{age},#{gender})  
</insert>
```

## 修改 update

```
<!--修改update-->
<update id="updateUser">
    update j23_user_t set gender = #{gender} where id = #{id}
</update>
```

## 删除 delete

```
<!--删除delete-->
<delete id="deleteUser">
    delete from j23_user_t where id = #{id}
</delete>
```

## 小结:

- 通过sqlsession执行不同的方法做增删改查：需两个参数  
String: 命名空间 + sql ID 来获得唯一的sql的语句  
Object: 为sql提供参数： xml文件中用#{ }来表示  
基本数据类型、包装类、字符串： #{ } 里面的字符随便写  
javabean: #{ } 中写的必须是javabean的成员变量名 才能调用其get方法
- 不同的操作对应mapper文件中不同的标签 insert delete select update
- select标签 中的 resultType属性不能省略， 对应的单条记录的类型
- 增删改的返回类型均为int 表示操作影响的数据库的记录数 不能够写resultType

## 配置项

标签之间的顺序是有要求的

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration> <!--配置 -->
    <properties/> <!--属性-->
    <settings/> <!--设置-->
    <typeAliases/> <!--类型命名-->
    <typeHandlers/> <!--类型处理器-->
    <objectFactory/> <!--对象工厂-->
    <plugins/> <!--插件-->
    <environments> <!--配置环境 -->
        <environment> <!--环境变量 -->
            <transactionManager/> <!--事务管理器-->
            <dataSource/> <!--数据源-->
        </environment>
    </environments>
    <databaseIdProvider/> <!--数据库厂商标识-->
    <mappers/> <!--映射器-->
</configuration>

```

mybatis.xml文件中的

- properties 后面不会使用到

单项配置：

```

<properties>
    <property name="db.driver" value="com.mysql.jdbc.Driver"/>
    <property name="db.username" value="root"/>
</properties>
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver" value="${db.driver}"/>
            <property name="url" value="jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf-8"/>
            <property name="username" value="${db.username}"/>
            <property name="password" value="123456"/>
        </dataSource>
    </environment>
</environments>

```

引入properties文件：

给properties中的属性加前缀. 一种规范，如db.username, 直接用username可能会引用到系统变量

```

db.url=jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf-8
db.password=123456

<configuration>
    <properties resource="db.properties">
        <property name="db.driver" value="com.mysql.jdbc.Driver"/>
        <property name="db.username" value="root"/>
    </properties>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${db.driver}"/>
                <property name="url" value="${db.url}"/>
                <property name="username" value="${db.username}"/>
                <property name="password" value="${db.password}"/>
            </dataSource>
        </environment>
    </environments>

```

- settings (后面讲)

缓存和懒加载

### 设置 (settings)

[后面讲](#)

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项设置的含义、默认值等。

设置名	描述	有效值	默认值
cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true   false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置 <code>fetchType</code> 属性来覆盖该项的开关状态。	true   false	false

- typeAliases

给resultType中javabean的全类名取一个别名 (在resultType中就可以直接只写alias)

对单条配置取别名

```

<typeAliases>
    <typeAlias type="com.cskaoyan.bean.User" alias="usera"/>      mybatis.xml
</typeAliases>

<!-- 配置别名后，仍然可以使用全类名 -->
<select id="selectUserById" resultType="usera">
    select id,username,password,age,gender from j23_user_t where id = #{id}
</select>                                mapper

```

批量取别名：直接指定一个包，该包下的所有javabean的别名都变成其简单类名的全小写形式

```

<typeAliases>
    <!--<typeAlias type="com.cskaoyan.bean.User" alias="usera"/>-->
        类名的纯小写的形式
    <!-- 该包目录下的bean的别名是什么-->
    <package name="com.cskaoyan.bean"/>
</typeAliases>

```

//alias → userdetail
//alias → user

```

public class UserDetail { @Data
}
public class User {
}

```

## mybatis自身提供的别名

关注基本类型、包装类、字符串

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

- typeHandlers

类型处理器，类似于converter

javatype: javabean中成员变量的类型

jdbctype: 关系表中列的类型

javatype ↔ jdbctype

- mappers:

- resource属性:

加载classpath下的映射文件

- url属性:

加载文件路径下的映射文件

url= file:///

```
<mapper url="file:///D:/WorkSpace/j23_workspace/codes/day11-mybatis1/demo2-configuration/src/main/resources/mapper/UserMapper.xml"/>
```

- o class属性：

配置的是接口

获取sql语句的方式从 命名空间 + sql语句id 变为了接口名 + 方法名

要保证如下对应关系：

1. 接口的java文件要与mapper.xml文件编译后在同一级目录下，且文件名相同
2. mapper.xml中的命名空间即它的id，为接口的全类名（接口中的方法只能传入一个Object类型的参数）
3. mapper.xml中的sql语句的id 为接口中的方法名

- o package属性

### 对应关系：

- o 接口和映射文件要在同一级目录，并且同名



- o mapper.xml中的命名空间为接口的全类名
- o 接口中的方法名，对应映射文件中sql语句标签的id

所以：同一个接口中方法名字不能重复（不支持方法重载）（以为同一个映射文件中，标签id不能重复）

```

<mappers>
    <!-- 加载在<namespace>下的映射文件-->
    <!--<mapper resources="mapper/UserMapper.xml">-->
    <!-- 加载<mapper>的映射文件-->
    <mapper url="file:///D:/WorkSpace/j23_workspace/codes/day11-mybatis1/demo2-configuration/src/main/resources/mapper/UserMapper.xml"/>
</mappers>

<mappers>
    <!-- 配置的是接口
        → 接口对应映射文件 → 接口的全类名: namespace
        → 接口中的方法对应 sql语句的id
        → 通过调用接口中的方法 → 唯一与之对应的sql语句
    对应关系:
        1、接口和映射文件要在同一级目录，并且同名
        2、命名空间 为接口的全类名
        3、接口中的方法名 对应映射文件中的sql语句标签的id
    注意事项: 同一个映射文件里 标签id 不能重复
    -->
    <mapper class="com.cskaoyan.mapper.UserMapper"/>
</mappers>

public interface UserMapper {
    String selectUsernameById(Integer id);
    //接口中不允许重载
    //String selectUsernameById(String username, String password);
}

<mapper namespace="com.cskaoyan.mapper.UserMapper">
    <select id="selectUsernameById" resultType="string">
        </select>
</mapper>

```

对应关系!

注意事项!

多个文件的配置（要对应多个接口）

```

<package name="com.cskaoyan.mapper"/> 扫描包下的全部接口均按上述的要求配置
</mappers>

```

注：

sql还是在映射文件，但是取出sql语句的方式变成了接口 + 方法

mybatis通过动态代理（字节码生成 想想Invoke方法就行），为接口提供了其对象

使用：

```

@Test
public void mytest1(){
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    String username = mapper.selectUsernameById(2); //namespace id 参数 结果
    System.out.println("username = " + username);
}

```

## log4j 日志框架

mybatis支持

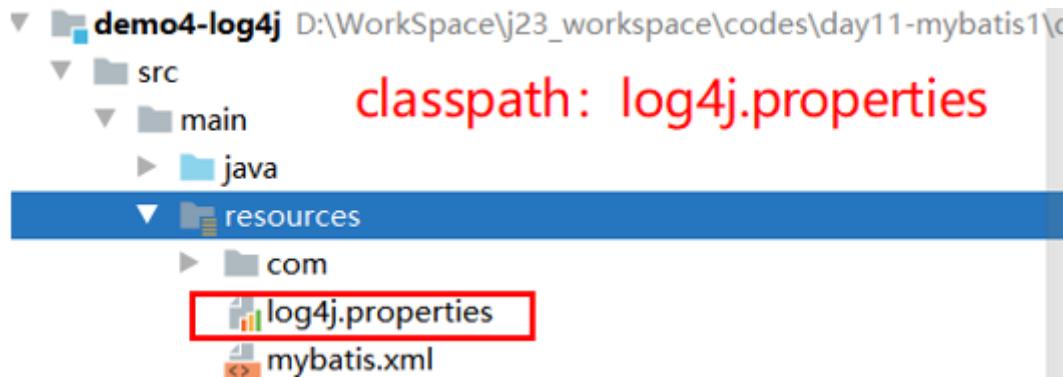
- 引入log4j的依赖

```

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>

```

- 引入配置文件: log4j.properties



可以用来看预编译的sql语句 与 输入的参数

配置文件:

appender: 添加器, 将日志记录添加到文件中

```

#Appender
#org.apache.log4j.ConsoleAppender (控制台)
#org.apache.log4j.FileAppender (文件)
#org.apache.log4j.DailyRollingFileAppender (每天产生一个日志文件)
#org.apache.log4j.RollingFileAppender (文件大小到达指定尺寸的时候产生一个新的文件)
#org.apache.log4j.WriterAppender (将日志信息以流格式发送到任意指定的地方)
### direct log messages to stdout ####
log4j.appender.stdout=org.apache.log4j.ConsoleAppender   类型是console, id是stdout
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file hibernate.log ####
log4j.appender.file=org.apache.log4j.FileAppender        类型是File, id是file
log4j.appender.file.File=d://21mybatis.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

```

layout:

输出内容的格式, 就用pattern, 有时间信息, 而且文件内容较小 (输出日志会消耗IO资源)

## 自定义的layout

ConversionPattern

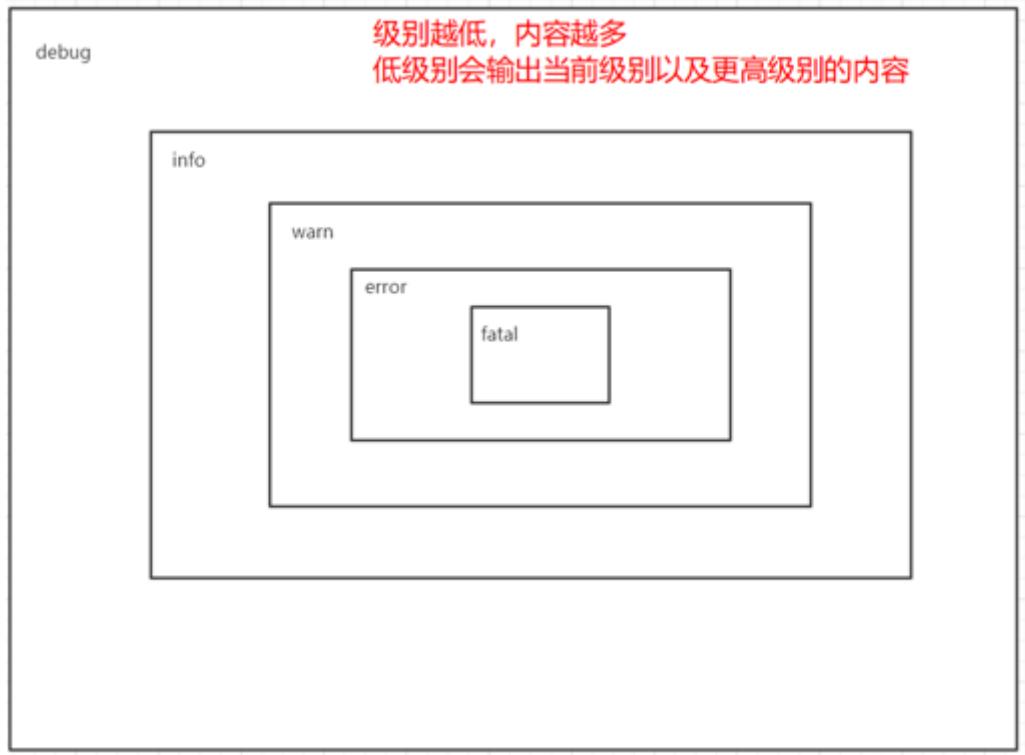
```
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

- %d  
表示时间 {ABSOLUTE} 时间格式 {yyyy-MM-dd} 也可以
- %p  
表示日志级别, %5p 日志级别占用5个字符
- %c  
类别信息 (所涉及的class或者方法) {数字}表示类别信息输出多少级, 从后往前
- %L  
行号: 源码中的第多少行输出的
- %m  
message消息
- %n  
换行

## 日志级别

debug info warn error fatal

- 级别越高, 输出内容越少
- 级别越低, 输出内容越多
- 低级别会输出当前级别以及更高级别的内容 (套娃模型)



自行输出日志: (用传入class的方式获取logger对象, 一般作为全局变量)

```

public class LogTest {
    Logger logger = Logger.getLogger(LogTest.class);
    //Logger logger = Logger.getLogger(this.getClass());

    @Test
    public void mytest(){
        logger.debug("debug");
        logger.info("info");
        logger.warn("warn");
        logger.error("error");
        logger.fatal("fatal");
    }
}

```

这些信息的输出要看日志级别，当前级别和更高级别的内容会输出

只传入message

## day 3

### 高级映射

#### 输入映射

为映射文件中的sql语句提供参数：

- 接口中的形参要怎么写
- 映射文件中的#{ }里写什么 (#{}在预编译语句中就是? 切记切记)

#### 单个参数

- 基本类型 包装类 字符串

#{} 里面随便写

预编译后就是个 ?

- javabean 或者 map

**成员变量名** (get方法)

Map的key (get方法中使用的key)

注意：单个参数为Javabean加注解的话就是用： #{user.username}了

#### 多个参数

依然关注#{ }中写什么

- param1, param2, .. paramn
- arg0, arg1.. argn-1

- 基本类型 包装类 字符串

param或arg

- javabean或map

param或arg 通过 . 点出具体的成员变量名或者map中的key

### 使用注解@Param

使用接口中方法形参前面@Param注解中的value值

写了什么就用什么，点. 的使用规则也是一样的

---

## 输出映射

最常用的一个场景

就是对sql语句查询的结果集的封装

结果集：对应mapper接口方法的返回值

返回值为：基本类型、包装类和字符串（对应数据库表中的一列）

特征：结果集中的字段只有一个

### 单条记录

返回的记录数为：0/1（select后只跟一个列且where后使用了唯一性索引）

```
//通常是查询满足某个条件的 某一列的值或记录数
String selectUsernameById(@Param("id") Integer id);
<select id="selectUsernameById" resultType="java.lang.String">
    select username from j23_user_t where id = #{id}
</select>          通常只写一个列或别名
@Test
public void mytest1(){
    String username = mapper.selectUsernameById(2);
    logger.info("username = " + username);
}
Preparing: select username from j23_user_t where id = ?
Parameters: 2(Integer)
Total: 1
```

### 多条记录

- 返回的记录数 $\geq 0$
- 返回值为数组或者list都可以（可以互相替换）
- resultType还是单条记录对应的java类型

## 单条和多条的比较

- 接口中方法的返回值是不同的
- 查询结果允许的记录数不同

## Javabean

底层：

- 先通过class对象 newInstance
- 然后调用结果集字段名对应的set方法赋值
- 返回javabean

表的列名有时候和javabean的成员变量名不一致

- 使用as

## 单条记录

```
//javabean
User selectUserById(@Param("id") Integer id);

<select id="selectUserById" resultType="com.cskaoyan.bean.User">
    select id,username,password,age,gender,update_date as updateDate from j23_user_t where id = #{id}
</select>
@Test
public void mytest3(){
    User user = mapper.selectUserById(2);
    logger.info("user = " + user);          查询结果的列名与javabean的成员变量名一致
}
Preparing: select id,username,password,age,gender,update_date as updateDate from j23_user_t where id = ?
Parameters: 2(Integer)
Total: 1
```

## 多条记录

使用javabean的数组或者list

```

//javabean
User selectUserById(@Param("id") Integer id);
User[] selectUserArray();
List<User> selectUserList(); 可以以数组或list的形式接收

<!--user数组-->
<select id="selectUserArray" resultType="com.cskaoyan.bean.User">
    select id,username,password,age,gender,update_date as updateDate from j23_user_t
</select> 映射文件中的写法一样
<!--userList-->
<select id="selectUserList" resultType="com.cskaoyan.bean.User">
    select id,username,password,age,gender,update_date as updateDate from j23_user_t
</select>

@Test
public void mytest4(){
    User[] users1 = mapper.selectUserArray();
    for (User user : users1) {
        logger.info("users1:" + user);
    }
    List<User> users2 = mapper.selectUserList();
    for (User user : users2) {
        logger.info("users2:" + user);
    }
}

selectUserArray:137 - ==> Preparing: select id,username,password,age,gender,update_date as updateDate from j23_user_t
selectUserArray:137 - ==> Parameters:
selectUserArray:137 - <== Total: 2
selectUserList:137 - ==> Preparing: select id,username,password,age,gender,update_date as updateDate from j23_user_t
selectUserList:137 - ==> Parameters:
selectUserList:137 - <== Total: 2

```

## 单条和多条的比较

- 接口中方法的返回值不同
- 查询结果允许的记录数不同

**resultMap** 可以替换 **resultType** (select标签中一定要有这两个属性之一)

- **resultType**: 结果集的字段名与javabean成员变量名一致
- **resultMap**: 结果集的字段名与javabean成员变量名对应
- **column**(结果集中的字段名)
- **property** (javabean的成员变量名) (javabean的set方法)
- **type**: 原来的**resultType**

补充

```

<!-- resultMap-->
<!-- resultMap标签的type属性写的就是原先resultType里写的内容 → javabean的全类名、typeAlias-->
<resultMap id="userMap" type="com.cskaoyan.bean.User"> property来源于父标签的成员变量名
    <!--column始终是查询结果的列名-->
    <!--property始终是父标签类型javabean的成员变量名-->
    <result column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="password" property="password"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="update_date" property="updateDate"/>
</resultMap>
<!--javabean-->
<select id="selectUserByIdMap" resultMap="userMap">
    select id,username,password,age,gender,update_date from j23_user_t where id = #{id}
</select>

<!--user数组-->
<select id="selectUserArrayMap" resultMap="userMap">
    select id,username,password,age,gender,update_date from j23_user_t
</select>
<!--userList-->          result标签中的column
<select id="selectUserListMap" resultMap="userMap">
    select id,username,password,age,gender,update_date from j23_user_t
</select>

```

## day 4

### 动态SQL标签

动态sql：根据输入参数的不同去动态地改变所执行的sql语句

#### where标签

作用：

- 将where语句后的内容，放入到where标签中
- 连接词的去除 如 where and -> where
- where的去除 如 .. where -> .. (后面没有跟任何条件的时候)

#### if标签

根据test中的判断条件来决定是否拼接sql语句 (场景：条件查询)

查询：根据 username 和 password 条件查询

```
List<User> selectUserByUsernameAndPassword(@Param("username") String username,
                                            @Param("password") String password);

<!-- 利用 if 标签来做 条件查询--&gt;
<!-- if 标签中的 test 属性写的是条件 → 输入映射可以写的值 → @Param 注解的 value 属性值--&gt;
&lt;select id="selectUserByUsernameAndPassword" resultMap="userMap"&gt;
    select id,username,password,age,gender,update_date from j23_user_t
    &lt;where&gt;
        &lt;if test="username != null"&gt;
            username = #{username}
        &lt;/if&gt;
        &lt;if test="password != null"&gt;
            and password = #{password}
        &lt;/if&gt;
    &lt;/where&gt;
&lt;/select&gt;</pre>

test属性中写条件 → 决定是否拼接if标签中的sql语句  
test属性中可以使用的值 是输入映射可以使用的值 → @Param中的value属性值


```

## test中的语法

- 变量部分：输入映射中@Param注解中的value

可以直接使用，不用加#{..}

(无注解时是param1..)

在#{ } 中可以写什么 在这里就可以写什么

- 常量

- 数值：直接写

- 字符串：用单引号 'daxiao'

- 比较符：

等于： ==

不等于： !=

大于： gt

小于： lt

大于等于： gt .. or == .. 如： (age gt 20 or age == 20)

小于等于： lt .. or == ..

- 连接词：多个条件之间的关系

and

or

- null值： null

- 字符串长度： 'daixao'.length

## choose-when-otherwise (if-else)

choose标签下

有when和otherwise两个子标签

when标签中可以使用test属性，代表if

otherwise标签代表的是else

```
<select id="selectUserByAgeChooseWhen" resultMap="userMap">
    select id,username,password,age,gender,update_date from j23_user_t
    <where>
        <choose>
            <when test="age gt 20 or age == 20">
                age &gt;= #{age}
            </when>
            <otherwise>
                age &lt; #{age}
            </otherwise>
        </choose>
    </where>
</select>
```

注意：在test=""中输入映射可以直接用，而不用加#{}, gt lt直接写要加上or来表示>=<=

```
@Test
public void mytest4(){
    List<User> users1 = mapper.selectUserByAgeChooseWhen(20);
    System.out.println(users1);
    Total: 2

    List<User> users2 = mapper.selectUserByAgeChooseWhen(21);
    System.out.println(users2);
    Total: 0

    List<User> users3 = mapper.selectUserByAgeChooseWhen(19);
    System.out.println(users3);
    Total: 0
}
```

注意：

• 在test=""中输入映射可以直接用，而不用加#{}, gt lt直接写要加上or来表示>=<=

• 在xml文件中 &gt; 就是一个转义字符 就是等于在这个位置放了一个>号，所以可以直接写 &gt;= #{age}

```
<resultMap id="selectUserMap" type="user">
    <result column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="password" property="password"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="update_time" property="updateTime"/>
</resultMap>

<select id="selectUserByAgeChoosewhenOtherwise" resultMap="selectUserMap">
    select <include refid="userColumnsForSelect"/> from user_t
    <where>
        <choose>
            <when test="age gt 20 or age == 20">
                age &gt;= #{age}
            </when>
            <otherwise>
                age &lt; #{age}
            </otherwise>
        </choose>
    </where>
</select>
```

**trim标签**: 在选择性更新的时候，解决末尾逗号的问题

- prefix: 最前面拼接一个内容
- suffix: 最后面拼接一个内容
- prefixOverrides: 如果最前面出现了该值，就去掉
- suffixOverrides: 如果最后面出现了该值，就去掉

**set标签**

相当于 trim + prefix(set) + suffixOverrides(,) 一个小封装

主要针对update

### sql-include

抽出重复代码 引用

- sql标签：定义一个sql语句片段
- include标签：引用sql标签中的内容

```
<select id="selectUserById" resultMap="userMap">
    select id,username,password,age,gender,update_date from j23_user_t
    <where>
        id = #{id}
    </where>
</select>
<sql id="baseSelectUser">
    select id,username,password,age,gender,update_date from j23_user_t
</sql>
<select id="selectUserById" resultMap="userMap">
    <include refid="baseSelectUser"/>
    <where>
        id = #{id}
    </where>
</select>
```

改造前

改造后

使用场景：

- select中的列名
- where中的条件

**selectKey标签** (放在insert标签内部) (有点类似触发器)

在insert语句之前或者之后执行

- order为AFTER: 在insert语句之后执行

原因：想要获得插入记录的id值（由数据库自动生成）针对Insert语句

方法：对输入映射中传入的参数的成员变量进行赋值，调用了其set方法，

通过用于查询的javabean对象的成员变量的值传给外界

- keyColumn: 与selectKey标签下的select语句的结果集的字段名对应
- keyProperty: 与传入的javabean的成员变量名对应
- resultType: select语句的结果集中的返回类型

```

//selectKey
int insertUser(@Param("user") User user);
<!--selectKey → 查询结果封装给输入映射传入的值-->
<!--
    order: selectKey中的语句执行顺序 → 相对于insert语句的
    keyColumn: selectKey查询语句中查询结果的列名
    keyProperty: 查询结果要封装给谁 → 通常写的是和输入映射的值相关 → @Param注解的value属性值相关
    resultType: keyProperty的类型
-->
<insert id="insertUser">
    <selectKey order="AFTER" keyColumn="increatedId" keyProperty="user.id" resultType="int">
        select LAST_INSERT_ID() as increasedId
    </selectKey>
    insert into j23_user_t
    (username,password,age,gender,update_date)
    values ({@user.username},#{@user.password},#{@user.age},#{@user.gender},#{@user.updateDate})
</insert>

```

@Test  
public void mytest7(){  
 User user = new User();  
 user.setUsername("zhengshuangshuang");  
 user.setPassword("bushengqi");  
 user.setAge(28);  
 user.setGender("man");  
 user.setUpdateDate(new Date());  
 int insert = mapper.insertUser(user); 注意这个insert方法的返回值并不是自增的id  
自增的id在user的id里  
 logger.info("insert = " + insert);  
 logger.info("user = " + user);  
}

Preparing: insert into j23\_user\_t (username,password,age,gender,update\_date) values (?, ?, ?, ?, ?)  
Parameters: zhengshuangshuang(String), bushengqi(String), 28(Integer), man(String), 2020-08-27 16:21:28.719(Timestamp)  
Updates: 1

Preparing: select LAST\_INSERT\_ID() as increasedId selectKey中的语句相对于insert的order是AFTER  
Parameters:  
Total: 1

- order为BEFORE

场景：插入一个user数据，username是随机生成的：uuid在insert执行之前生成

同样还是将select语句的结果赋值给javabean的成员变量

```

select UUID() as username;

insert into j23_user_t (username, password, age, gender, UPDATE_date)
values (?, ?, ?, ?, ?)

```

```

<insert id="insertUserOrderBefore">
    <selectKey order="BEFORE" keyColumn="username" keyProperty="user.username" resultType="string">
        select uuid() as username
    </selectKey>
    insert into j23_user_t
    (username,password,age,gender,update_date)
    values ({@user.username},#{@user.password},#{@user.age},#{@user.gender},#{@user.updateDate})
</insert>

```

Preparing: select uuid() as username 给到输入映射传入的user  
Parameters:  
Total: 1

Preparing: insert into j23\_user\_t (username,password,age,gender,update\_date) values (?, ?, ?, ?, ?)  
Parameters: c661b125-e83f-11ea-820b-80e04c364a23(String), bushengqi(String), 28(Integer), man(String), 2020-08-27 16:31:56.227(Timestamp)  
Updates: 1

## insert标签中的属性：useGeneratedKeys + keyProperty

也是获取自增的id（也是小封装）

keyProperty中的值和输入映射里可以使用的内容相关，即@Param中的值

```

// 使用insert标签的属性 获得自增的主键
int insertUserGenerateKey(@Param("user") User user);

<!-- 自增主键-->
<insert id="insertUserGenerateKey" useGeneratedKeys="true" keyProperty="user.id">
    insert into j23_user_t
    (username,password,age,gender,update_date)
    values (#{user.username},#{user.password},#{user.age},#{user.gender},#{user.updateDate})
</insert>

insert into j23_user_t (username,password,age,gender,update_date) values (?, ?, ?, ?, ?)
xiaoLiang(String), bushengqi(String), 20(Integer), man(String), 2020-08-27 16:39:42.808(Timestamp)
1

```

## foreach遍历

重要

输入映射：提供list

和foreach代码作类比

```

for (Integer id : ids) {
    sb.append(id);
    sb.append(",");
}
<foreach collection="array" item="id">
|
</foreach>

```

- collection的写法：
  - 没写@param前 和形参的类型相关：array 或者list
  - 写了之后 按@Param中的value值
- item：每一项 -> #{item}
- separator： foreach标签中的内容之间加一个东西

每两个 foreach 标签里面内容之间的分隔符 ↵

- open：最左边加一个东西
- close：最右边加一个东西
- 插入一个user的list时，想要获得每个user插入后自动生成的id

在userGeneratedKey对应的keyProperty里面写： users.id 把自增的Id赋值到每个users的每个user中

使用场景：

- in

```

<!--foreach-->
<sql id="base_select">
    select id,username,password,age,gender,update_date from j23_user_t
</sql>
<!-- collection: 填的值和输入映射相关
    数组: array
    list: list
    @Param: value 属性值
    separator: foreach标签里的内容每两个之间的分隔符
    open 和 close: 整个foreach执行完成最前和最后的补充内容
-->
<select id="selectUserByIdInArray" resultMap="userMap">
    <include refid="base_select"/>
    <where>
        id in
            <foreach collection="array" item="id" separator="," open="(" close=")">
                #{id}
            </foreach> #{}里的值和item相关
    </where>
</select>

```

Preparing: select id,username,password,age,gender,update\_date from j23\_user\_t WHERE id in ( ?, ?, ?, ?, ?, ? )  
Parameters: 1(Integer), 2(Integer), 3(Integer), 4(Integer), 5(Integer)  
Total: 5

- insert多条记录

```

//foreach → 批量插入多条记录
int insertUsers(@Param("users") List<User> users);
<insert id="insertUsers">
    insert into j23_user_t (username,password,age,gender,UPDATE_date)
    values
        <foreach collection="users" item="user" separator=",">
            (#{user.username},#{user.password},#{user.age},#{user.gender},#{user.updateDate})
        </foreach>
</insert>
@Test
public void mytest12(){
    User user1 = new User(id: null, username: "user1", password: "password1", age: 21, gender: "male", new Date());
    User user2 = new User(id: null, username: "user2", password: "password2", age: 22, gender: "male", new Date());
    User user3 = new User(id: null, username: "user3", password: "password3", age: 23, gender: "male", new Date());
    User user4 = new User(id: null, username: "user4", password: "password4", age: 24, gender: "male", new Date());
    ArrayList<User> users = new ArrayList<>();
    users.add(user1);
    users.add(user2);
    users.add(user3);
    users.add(user4);

    int i = mapper.insertUsers(users);
    logger.info("insert = " + i);
}
insert into j23_user_t (username,password,age,gender,UPDATE_date) values (?,?,?,?) , (?,?,?,?) , (?,?,?,?) , (?,?,?,?)
user1(String), password1(String), 21(Integer), male(String), 2020-08-27 17:39:13.6(Timestamp), user2(String), password2(String),
4

```

### 获得自增的主键

```

//foreach → 批量插入多条记录
int insertUsers(@Param("users") List<User> users);
<insert id="insertUsers" useGeneratedKeys="true" keyProperty="users.id">
    insert into j23_user_t (username,password,age,gender,UPDATE_date)
    values
        <foreach collection="users" item="user" separator=",">
            (#{user.username},#{user.password},#{user.age},#{user.gender},#{user.updateDate})
        </foreach>
</insert>

```

# 多表查询

## 一对一

如: User ↔ UserDetail

### 数据库表关系

在一张表中维护另一张表的唯一标识的字段 (一般是主键)

如: 在userDetail表中维护一个userId

### Javabean关系

在一个Javabean持有另一个javabean的成员变量 (has-a)

```
public class User {  
  
    UserDetail userDetail;  
}
```

## 查询

- 分次查询:

两次单表查询, 先通过username找出user表中对应的信息

再通过user的id来查找出对应的userDetail的信息

```
// 根据username查询user信息 → user 需要绑定对应的UserDetail信息  
User selectUserByName(@Param("username") String username);  
<resultMap id="userMap" type="com.cskaoyan.bean.User">  
    <result property="id" column="id"/>  
    <result property="username" column="username"/>  
    <result property="password" column="password"/>  
    <result property="age" column="age"/>  
    <result property="gender" column="gender"/>  
    <result property="updateDate" column="update_date"/>  
<!--  
    column: 查询结果的列名 → 并且为第二次查询提供参数  
    property: 始终对应父标签type的javabean的成员变量名  
    select: 第二次查询的 |命名空间+id|  
-->  
    <association property="userDetail" column="id"  
        select='com.cskaoyan.mapper.UserMapper.selectUserDetailById' />  
</resultMap>  
<select id="selectUserByName" resultMap="userMap">  
    select id,username,password,age,gender,update_date from j23_user_t where username = #{username}  
</select>  
<!-- 第二次查询#{ }里的值如何写 → 随便写-->  
<select id="selectUserDetailById" resultType="com.cskaoyan.bean.UserDetail">  
    select id,phone,email from j23_user_detail_t where user_id = #{userId}  
</select>
```

association标签: 右边是一

- column属性：查询结果的列名 并且还可以为第二次查询提供参数
- property属性：父标签javabean的成员变量名
- select属性：第二次查询所使用的select语句：命名空间（接口的全类名） + sql的id

- **连接查询**

通常使用左连接，保证左一的信息都在

一次把所有需要封装的数据都查询出来，再进行封装

封装userDetail时

- association标签
  - property属性：父标签javabean的属性
  - javatType属性：association所需封装的类型
  - result子标签：一样有column 和 property属性

---

## 一对多

一个user -> 多个order

### 数据库表关系

在order表中维护user\_id (唯一标识：主键字段)

### javabean关系

在一个类中创建另一个类的对应类型的list

```
public class User {  
    List<Order> orders;  
}
```

在一的类中创建另一个多的类对应类型的List

查询：

- 分次查询：

```

User selectUserByName(@Param("username") String username);

<mapper namespace="com.cskaoyan.mapper.UserMapper">
    <resultMap id="userMap" type="user">
        <result property="id" column="id"/>
        <result property="username" column="username"/>
        <result property="password" column="password"/>
        <result property="age" column="age"/>
        <result property="gender" column="gender"/>
        <result property="updateDate" column="update_date"/>
        <!--一对多 使用collection标签-->
        <collection property="orders" column="id"
            select="com.cskaoyan.mapper.UserMapper.selectOrdersByUid"/>
    </resultMap>
    <select id="selectUserByName" resultMap="userMap">
        select id,username,password,age,gender,update_date from j23_user_t where username = #{username}
    </select>

    <resultMap id="orderMap" type="order">
        <result property="id" column="id"/>
        <result property="ordername" column="ordername"/>
        <result property="money" column="money"/>
    </resultMap>
    <select id="selectOrdersByUid" resultMap="orderMap">
        select id,ordername,money from j23_order_t where user_id = #{userid}
    </select>
</mapper>

```

和一对一分次查询比较：  
区别：  
标签不同：association和collection  
select属性值对应的查询结果记录数  
→ 第二次查询的记录数  
    一对一：记录数为0或1  
    一对多：记录数≥0  
联系：  
    property, column, select属性

注意：分次查询时，如果第一条语句没查出任何记录，第二条语句就不会执行

- collection标签

column

property

select

均与association一样

但是连接查询时：内部的javabean的类型要写在ofType中

接口返回值为User和List的区别

- 返回记录数不同而已

连接查询

- collection

property

ofType

result: property column

```

<!-- 左连接查询-->
<resultMap id="userMapLeft" type="user">
    <result property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="password" column="password"/>
    <result property="age" column="age"/>
    <result property="gender" column="gender"/>
    <result property="updateDate" column="update_date"/>
    <collection property="orders" ofType="com.cskaoyan.bean.Order">
        <result property="id" column="oid"/>
        <result property="ordername" column="ordername"/>
        <result property="money" column="money"/>
    </collection>
</resultMap>
<select id="selectUserByNameLeft" resultMap="userMapLeft">
    select
        u.id,
        u.username,
        u.password,
        u.age,
        u.gender,
        u.update_date,
        o.id as oid,
        o.ordername,
        o.money
    from j23_user_t u
    LEFT JOIN j23_order_t o on u.id = o.user_id
    where u.username = #{username}
</select>

```

一对多：分次和连接查询的联系和区别  
 联系：都是collection标签，都使用property属性  
 区别：  
 分次：column、select  
 连接查询：ofType，使用result子标签

连接查询：一对一连接和一对多连接  
 联系：property属性都有类型的表达，使用result子标签进行映射  
 区别：  
 标签不同 → 一对一association；一对多collection  
 类型表达不同 → 一对一 javaType；一对多ofType

## 多对多

学生与课程

本质：双向的一对多

### 数据库表关系

维护关系：通过一个中间表

### javabean关系

互为一对多

```

public class Course {
    List<Student> students;
}

public class Student {
    List<Course> courses;
}

```

互为一对多

查询：

分次查询

```

<!-- 分次查询-->
<sql id="selective_student_name">
    <where>
        <if test="studentName != null">
            student_name = #{studentName}
        </if>
    </where>
</sql>
<resultMap id="studentMap" type="com.cskaoyan.bean.Student">
    <result property="id" column="id"/>
    <result property="studentName" column="student_name"/>
    <collection property="courses" column="id"
                select="com.cskaoyan.mapper.StudentMapper.selectCourseByStudentId"/>
</resultMap>
<select id="selectStudentByName" resultMap="studentMap">
    select id,student_name from j23_student_t
    <include refid="selective_student_name"/> 做了个条件查询
</select>
<select id="selectCourseByStudentId" resultType="com.cskaoyan.bean.Course">
    select
        c.id,
        c.course_name as courseName
    from j23_course_t c
    LEFT JOIN j23_relationship_t r ON c.id = r.course_id
    where r.student_id = #{stuid}
</select>

```

## 连接查询

使用了两次连接

```

<!-- 连接查询-->
<resultMap id="studentMapLeft" type="com.cskaoyan.bean.Student">
    <result property="id" column="id"/>
    <result property="studentName" column="student_name"/>
    <collection property="courses" ofType="com.cskaoyan.bean.Course">
        <result property="id" column="cid"/>
        <result property="courseName" column="course_name"/>
    </collection>
</resultMap>
<select id="selectStudentByNameLeft" resultMap="studentMapLeft">
    SELECT
        s.id,
        s.student_name,
        c.id as cid,
        c.course_name
    from j23_student_t s
    LEFT JOIN j23_relationship_t r on s.id = r.student_id
    LEFT JOIN j23_course_t c on c.id = r.course_id
    <include refid="selective_student_name"/> 两次连接
</select>

```

# settings

## 缓存cache

mybatis缓存有两级

### 一级缓存

一级缓存是默认开启的

同一个sqlsession (可以是同一个sqlSession中获得的不同的mapper)

缓存的表现：使用相同的查询时 没有进行预编译

在执行sqlSession的commit时 或者DML时失效

```
@Test
public void mytest4(){
    SqlSession sqlSession = factory.openSession();

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    String username1 = userMapper.selectUsernameById(1); //执行查询并生成缓存
    String username2 = userMapper.selectUsernameById(1); //使用缓存
    sqlSession.commit(); //清理缓存
    String username3 = userMapper.selectUsernameById(1); //执行查询并生成缓存
    String username4 = userMapper.selectUsernameById(1); //使用缓存
}
```

### 二级缓存

命名空间级别 -> 接口的全类名 -> 同一个mapper接口都可以

开启步骤：

- mybatis.xml中的settings中的cacheEnabled设置为true (也可以不设置)
- 查询结果要是可序列化的 (javabean 实现Serializable接口)
- 映射文件中要开启缓存

```
<settings>
    <setting name="cacheEnabled" value="true"/> 1
</settings>

@Data
public class User implements Serializable { 2
<mapper namespace="com.cskaoyan.mapper.UserMapper">
    <cache/> 3
```

```

/**
 * 使用了不同sqlSession获得的不同mapper
 */
@Test
public void mytest3(){
    SqlSession sqlSession1 = factory.openSession();
    SqlSession sqlSession2 = factory.openSession();
    SqlSession sqlSession3 = factory.openSession();
    SqlSession sqlSession4 = factory.openSession();

    UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
    UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
    UserMapper userMapper3 = sqlSession3.getMapper(UserMapper.class);
    UserMapper userMapper4 = sqlSession4.getMapper(UserMapper.class);
    String username1 = userMapper1.selectUsernameById(1);
    sqlSession1.commit(); // sqlSession1.close(); 要提交或关闭，才会放入到二级缓存
    String username2 = userMapper2.selectUsernameById(1);
    String username3 = userMapper3.selectUsernameById(1);
    String username4 = userMapper4.selectUsernameById(1);
    String username5 = userMapper4.selectUsernameById(1);

}

selectUsernameById:137 - ==> Preparing: select username from j23_user_t where id = ?
selectUsernameById:137 - ==> Parameters: 1(Integer)
selectUsernameById:137 - <== 命名空间 Total: 1
60 - Cache Hit Ratio [com.cskaoyan.mapper.UserMapper]: 0.5
60 - Cache Hit Ratio [com.cskaoyan.mapper.UserMapper]: 0.6666666666666666
60 - Cache Hit Ratio [com.cskaoyan.mapper.UserMapper]: 0.75
60 - Cache Hit Ratio [com.cskaoyan.mapper.UserMapper]: 0.8

```

## 懒加载

需要使用的时候才进行查询（即使用到相关的get方法的时候）

默认不开启

开启懒加载后默认为懒加载

```

<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>

@Test
public void mytest2(){
    User user = userMapper.selectUserByName(username: "songge");
    System.out.println(user.getPassword()); 注意：直接打印user或使用debug的时候会使用到
                                                .toString方法，如果你对toString方法重写了，就会使用到get方法 → 导致执行到第二次查询
}

```

注意：

打印user对象或者debug时，都会用到其toString()方法，

其中会用到getUserDetail方法时

eager: 在设置了全局都是懒加载的情况下，想让它立即加载

association标签中的fetchType属性中写eager

```
<association property="userDetail" column="id" fetchType="eager" association或collection中使用
    select="com.cskaoyan.mapper.UserMapper.selectUserDetailByUid"/>fetchType属性
```

## day 6

### typehandler

解决的问题：输入输出映射过程中参数类型无法直接转化（需要是一个对象的成员变量才能用到typehandler）

```
class User{
    Integer id;
    UserDetail userDetail; //UserDetail
    Integer[] orderIds;
}
```

场景：

有些需要直接在表中存数据（代表对象的json字符串）而非id

不经常变 或者不作为查询条件

	Java中	数据库表中
user_detail	UserDetail对象	varchar
order_ids	Integer[]对象	varchar

#### 需要转换的情况

- 输入映射：向预编译的sql语句提供参数  
javaType.UserDetail -> jdbcType.varchar
- 输出映射：对查询的结果集的封装  
jdbcType.varchar -> javaType.UserDetail

#### 使用步骤

- 在mybatis.xml中配置typeHandlers（单个类或者包目录）

```

<typeHandlers>
    <!--handler要写typehandler的全类名-->
    <!--<typeHandler handler="com.cskaoyan.typehandler.UserDetailTypehandler"/>-->
    <package name="com.cskaoyan.typehandler"/>
</typeHandlers>

```

配置单个或包目录

- 定义typeHandlers

- 继承BaseTypeHandler 泛型中写javaType
- 在注解 @MappedTypes 中写javaType
- 在注解 @MappedJdbcTypes 中写 jdbcType

```

@MappedTypes(UserDetail.class) //javabean中成员变量类型 JavaType
@MappedJdbcTypes(JdbcType.VARCHAR) //表中某列的类型 JdbcType
//BaseTypehandler 中的泛型写MappedTypes里的类型 → javabean中的成员变量的类型
public class UserDetailTypehandler extends BaseTypeHandler<UserDetail> {

```

- 处理输入映射时的转换的方法

```

@sneakyThrows
@Override
public void setNonNullParameter(PreparedStatement preparedStatement, int index, UserDetail userDetail, JdbcType jdbcType) {
    String value = objectMapper.writeValueAsString(userDetail);
    preparedStatement.setString(index, value); 转换为jdbcType能够接收的类型
}
为预编译的sql语句提供参数

```

预编译的第一个参数

输入映射传入的JavaType

- 处理输出映射时的转换的方法

```

//get方法 → 结果集的封装
@sneakyThrows
@Override
public UserDetail getNullableResult(ResultSet resultSet, String columnLabel) throws SQLException {
    System.out.println("string = " + columnLabel);
    String result = resultSet.getString(columnLabel);
    return parseResult(result);
}

@sneakyThrows
@Override
public UserDetail getNullableResult(ResultSet resultSet, int index) throws SQLException {
    String result = resultSet.getString(index);
    return parseResult(result);
}

@sneakyThrows
@Override
public UserDetail getNullableResult(CallableStatement callableStatement, int index) throws SQLException {
    String result = callableStatement.getString(index);
    return parseResult(result);
}

private UserDetail parseResult(String result) throws JsonProcessingException {
    return objectMapper.readValue(result, UserDetail.class);
}

```

先获取结果集 的某列

将jdbcType对应的类型的結果集  
转换为JavaType类型的数据

jackson::objectmapper, 转换器

readValue: json字符串 -> javaType

writeValueAsString: javaType -> json字符串



# SSM

在Spring的环境下使用mybatis，  
即将mybatis对应的组件（mapper接口的实例），交给spring容器管理  
业务逻辑的处理放在service层

## 使用步骤

- 导入依赖

```
<!--mybatis对spring的整合-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.5</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
<!--下面是新的-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.5</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId> 用于事务
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.22</version>
</dependency>
```

- 在spring容器中注册组件

DruidDataSource

SqlSessionFactoryBean(依赖于datasource)

MapperScannerConfigurer(依赖于sqlSessionFactoryBean)

```

<bean id="datasource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf-8"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</bean>
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="datasource"/>
</bean>
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
    <property name="basePackage" value="com.cskaoyan.mapper"/>
</bean>

```

就是生产sqlSessionFactory的

- 使用mybatis

```

@Service
public class UserServiceImpl implements UserService{
    @Autowired
    UserMapper userMapper;
    @Override
    public User queryUserById(Integer id) {
        return userMapper.selectUserById(id);
    }
}

```

容器中的组件 默认都是单例

从容器中可以直接取出组件

- 其他配置：在sqlSessionFactory这个bean标签下

可以设置原来在mybatis.xml中的一些属性

```

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="datasource"/>
    <property name="typeAliasesPackage" value="com.cskaoyan.bean"/>
    <property name="typeHandlers" value="com.cskaoyan.typehandler"/>
    <property name="configLocation" value="classpath:mybatis.xml"/>
</bean>

```

- JavaConfig

```

//datasource
@Bean
public DruidDataSource dataSource(){
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/j23_db?useUnicode=true&characterEncoding=utf-8");
    dataSource.setUsername("root");
    dataSource.setPassword("123456");
    return dataSource;
}

//sqlSessionFactory
@Bean
public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
    SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(dataSource);
    //sqlSessionFactoryBean.setTypeHandlersPackage();
    //sqlSessionFactoryBean.setTypeAliasesPackage();
    return sqlSessionFactoryBean;
}

//MapperScannerConfigurer
@Bean
public MapperScannerConfigurer mapperScannerConfigurer(){
    MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer();
    mapperScannerConfigurer.setSqlSessionFactoryBeanName("sqlSessionFactory");
    mapperScannerConfigurer.setBasePackage("com.cskaoyan.mapper");
    return mapperScannerConfigurer;
}

```

没有通过@Bean注解的value属性指定组件id  
默认采用方法名作为组件id

- mybatis中使用事务

Spring容器中注册DataSourceTransactionManager

配置：

application.xml中

```

xml → <tx:annotation-driven transaction-manager=>←
                                         开启事务的注解驱动
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="datasource"/>
</bean>
<tx:annotation-driven transaction-manager="transactionManager"/>

```

javaConfig中

```

javaConfig → @EnableTransactionManagement
@EnableTransactionManagement
public class SpringConfiguration {
    @Bean
    public DataSourceTransactionManager transactionManager(DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }
}

```

# Week 17

## day 1

# SpringBoot

## 简介

- 约定大于配置 (贯穿整个springboot)

springboot 和组件之间的约定

如果没有指定具体的组件，springboot帮你注册一个默认的(有的框架可能需要配置属性，如数据库连接池)

如果指定了具体的组件，以指定的组件为准

以此集成第三方框架

- 内置了javaee容器(tomcat，所以不用自己添加tomcat了)，

把tomcat的配置也放到了jar包中 (使用Jar包启动) 在spring-boot-web里面

*Spring Boot is just a couple of AutoConfigurations classes (== normal Spring @Configurations), that create @Beans for you if certain @Conditions are met.*

注：

- 长期支持的jddk版本：8 11
- sb1 <= jdk1.7
- sb2 >= jdk1.8

## 搭建一个springboot应用

默认是spring应用

### 网页搭建

<https://start.spring.io/>

Project  
● Maven Project    项目构建方式  
○ Gradle Project    Language  
● Java    ○ Kotlin    ○ Groovy

Spring Boot  
○ 2.4.0 (SNAPSHOT)    ○ 2.4.0 (M2)    ○ 2.3.4 (SNAPSHOT)    ● 2.3.3  
○ 2.2.10 (SNAPSHOT)    ○ 2.2.9    ○ 2.117 (SNAPSHOT)    ○ 2.116

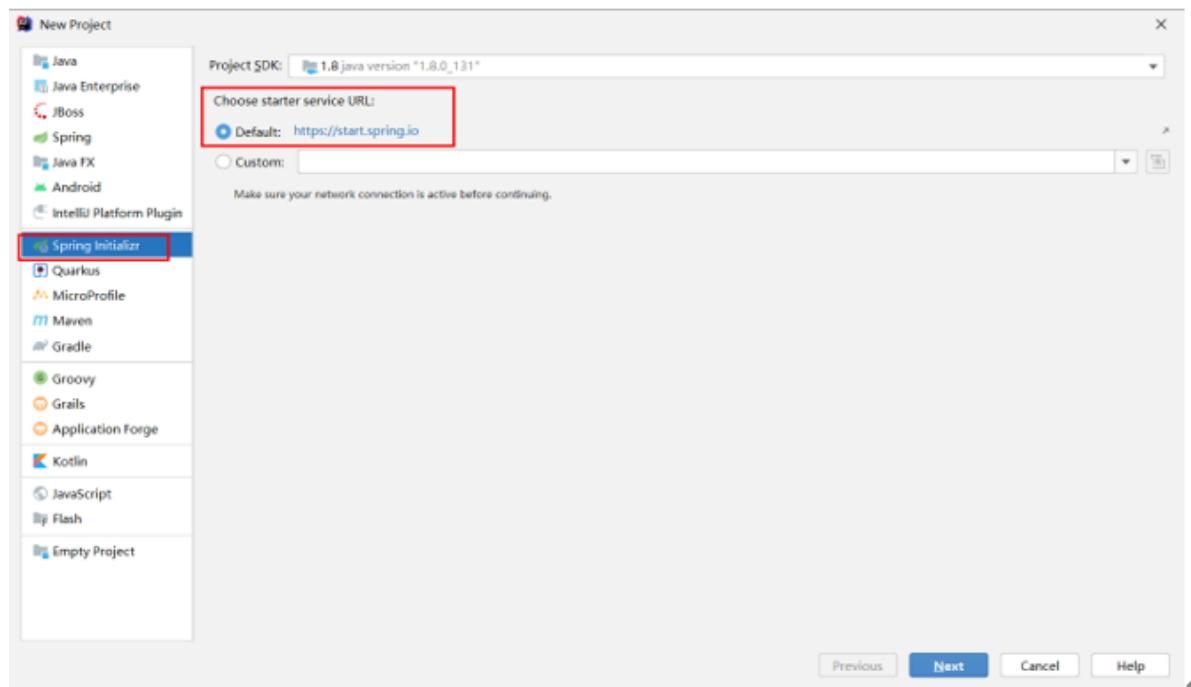
Dependencies  
ADD DEPENDENCIES... CTRL + B  
可以选择对springboot支持的框架  
No dependency selected

Project Metadata  
Group: com.example  
Artifact: demo  
Name: demo  
Description: Demo project for Spring Boot  
Package name: com.example.demo

Packaging: ● Jar    ○ War  
默认以jar包形式打包，内置tomcat容器  
Java: ○ 14    ● 11    ○ 8

注意：.gitignore文件要放在git仓库的根目录，才能生效

## idea搭建



## 启动类

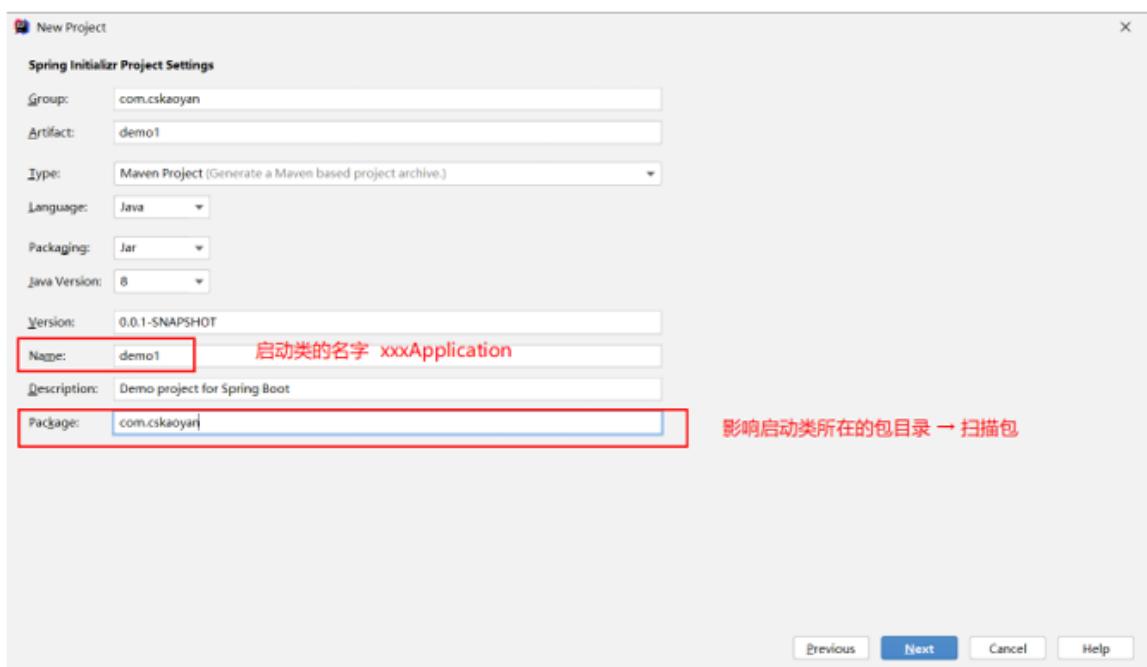
包含一个main方法

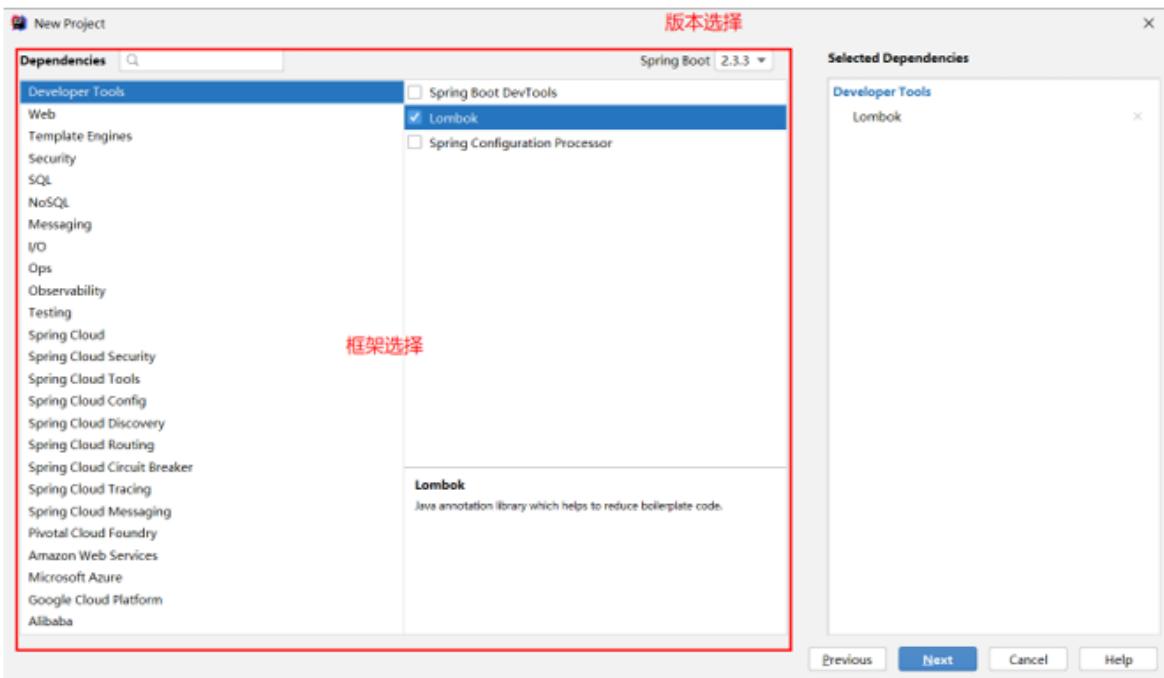
springboot应用的扫描包范围（即@ComponentScan）就是启动类所在的包目录

在扫描包范围内的有注解的组件都会成为spring容器中的组件

启动main方法就可以启动springboot应用（ee项目就是se项目的一个枝叶）

(之前是启动一个tomcat)





## 引入springboot对web的支持

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

导包即可

导入依赖时会注册一些默认的组件

避免要我们自己去写一些bean注解/标签

## pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.3.RELEASE</version>      springboot应用的版本
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

parent:父工程

dependecies:父工程的父工程 里面有依赖的默认版本号，所以有的依赖可以不设置版本号

(mysql驱动默认是8.0 要自己改成5的版本)

## spring-boot-starter-xxx

就是xxx框架对springboot的支持的依赖

- 包含了这个框架对spring支持的依赖
- springboot对约定大于配置的内容 (默认注册一些组件以供使用)  
所以通常只需要引入spring-boot-starter-xxx, 就可以使用对应的框架

---

## SpringBoot应用

### 启动类

可以用mvn package

打包成jar包 生成的jar包在target目录下

java -jar xxx.jar(jar包名字) 就可以启动一个springboot应用

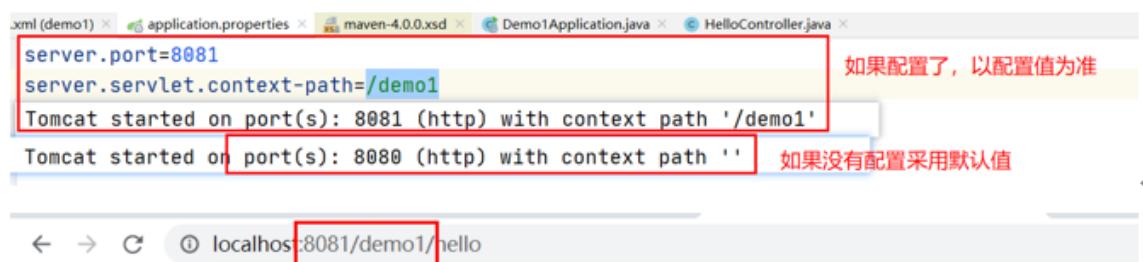
### springboot配置文件

springboot配置文件的名字

applicaiton.properties(yml) 或者 application-xxx.properties(yml)

### tomcat相关配置

- 启动端口号 server.port=
- 应用名路径 server.servlet.context.path=



The screenshot shows an IDE interface with several tabs open: XML (demo1), application.properties, maven-4.0.0.xsd, Demo1Application.java, and HelloController.java. The application.properties tab contains the following configuration:

```
server.port=8081
server.servlet.context-path=/demo1
```

Two annotations are present: "如果配置了, 以配置值为准" (If configured, use the configured value) is above the first line, and "如果没有配置采用默认值" (If no configuration is provided, use the default value) is above the second line.

The Tomcat startup logs show two entries:

```
Tomcat started on port(s): 8081 (http) with context path '/demo1'
Tomcat started on port(s): 8080 (http) with context path ''
```

The browser address bar at the bottom shows: localhost:8081/demo1/hello, with the path part highlighted by a red box.

### 另外一种格式的配置文件

也是key=value形式

yml (推荐使用)

- 遇到点. 要替换为冒号: + 换行缩进 (表示从属关系) (用空格来缩进, 不要使用tab制表符)
- 缩进几个空格都可以, 保持同一级对齐即可
- 等于号= 要替换为冒号: + 一个空格 (表示键值对关系)

server:  
port: 8081  
servlet:  
context-path: /demo1

m.xml (demo1) × application.properties × maven-4.0.0.xsd

server.port=8081  
server.servlet.context-path=/demo1

使用配置文件中key对应的值

```
@RestController  
public class HelloController {  
  
    @Value("${user.username}")  
    String username; //application.yml 中的 user.username 对应的值  
  
    @RequestMapping("hello")  
    public String hello(){  
        return "hello sb web " + username;  
    }  
}
```

pom.xml (demo1) × application.properties × application.yml × maven-4.0.0.xsd

1 server:  
2 port: 8081  
3 servlet:  
4 context-path: /demo1  
5 user:  
6 username: songge

**@ConfigurationProperties(prefix = "xxx")** (其他的框架中给组件中的成员变量赋值也是通过这个注解)

出现原因：像上面那样一个一个 @value 引用配置文件中的key太繁琐了

直接用这个注解就行，可以打通配置文件与组件之间的关系

实际上用的是组件的set方法（所以组件要有set方法）

-小写字母 <=> 大写字母

```

@Component
@ConfigurationProperties(prefix = "file") //配置文件中的key 等于 prefix加上成员变量名
@Data
public class FileComponent {

    //@Value("${file.basePath}")
    String basePath;
    String pngPath;
    String jpgPath;
    String xmlPath;
    Integer maxSize;
}

component = {FileComponent@5371} "FileComponent(basePath=d://spring, pngPath=d://spring/png, jpgPath=d://spring/jpg, xmlPath=d://spring/xml, maxSize=512)

file:
basePath: d://spring
pngPath: d://spring/png
jpgPath: d://spring/jpg
xmlPath: d://spring/xml
maxSize: 512

```

遇到大写字母可以修改为-和小写字母

自定义配置项的提示：（让配置文件也提示自己写的javabean）

需要导入依赖，然后再重新启动springboot

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>

```

重新启动你的 springboot

### 其他类型的成员变量的赋值

String Integer boolean List Map JavaDetail

从properties文件中取值

```

file2.base-path=d://spring
file2.max-size=512
file2.open=true

file2.list1=listdata1,listdata2,listdata3
file2.list2[0]=listdata1
file2.list2[1]=listdata2
file2.list2[2]=listdata3

file2.map1.key1=value1
file2.map1.key2=value2
file2.map1.key3=value3

file2.map2[key1]=value1
file2.map2[key2]=value2
file2.map2[key3]=value3

file2.file-detail.param1=songge
file2.file-detail.param2=ligenli

```

```

@Component
@ConfigurationProperties(prefix = "file2")
@Data
public class FileComponent {
    String basePath;
    Integer maxSize;
    boolean open;

    List list1;
    List list2;

    Map map1;
    Map map2;

    FileDetail fileDetail;
}

```

The screenshot shows the state of the `FileComponent` object. The `basePath` field is set to `d://spring`, `maxSize` to `512`, and `open` to `true`. The `list1` field contains three elements: `listdata1`, `listdata2`, and `listdata3`. The `list2` field contains three elements: `listdata1`, `listdata2`, and `listdata3`. The `map1` field contains three entries: `key1` with value `value1`, `key2` with value `value2`, and `key3` with value `value3`. The `map2` field contains three entries: `key1` with value `value1`, `key2` with value `value2`, and `key3` with value `value3`. The `fileDetail` field is an instance of `FileDetail` with parameters `param1` set to `songge` and `param2` set to `ligenli`.

从yml文件文件中取值

```

@Component
@ConfigurationProperties(prefix = "file2")
@Data
public class FileComponent {
    String basePath;
    Integer maxSize;
    boolean open;

    List list1;
    List list2;

    Map map1;
    Map map2;

    FileDetail fileDetail;
    FileDetail fileDetail2;
}

```

```

file2:
  base-path: d://spring
  max-size: 512
  open: true

  list1: listdata1,listdata2,listdata3
  list2:
    - listdata1
    - listdata2
    - listdata3

  map1:
    key1: value1
    key2: value2
  map2: {key1: value1,key2: value2,key3: value3}
  file-detail1:
    param1: songge
  file-detail2:
    param1: songge
    param2: ligenli

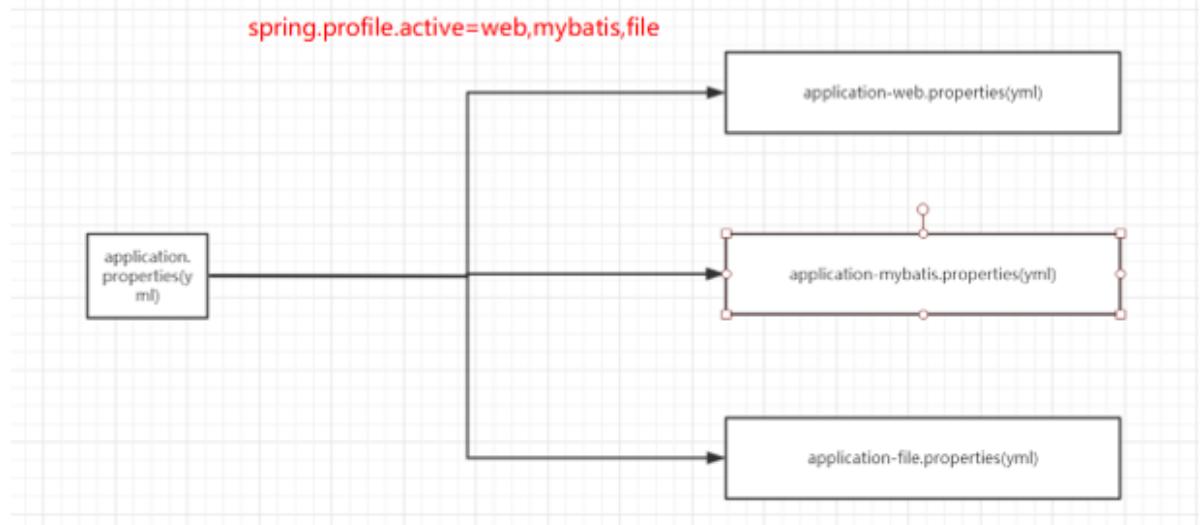
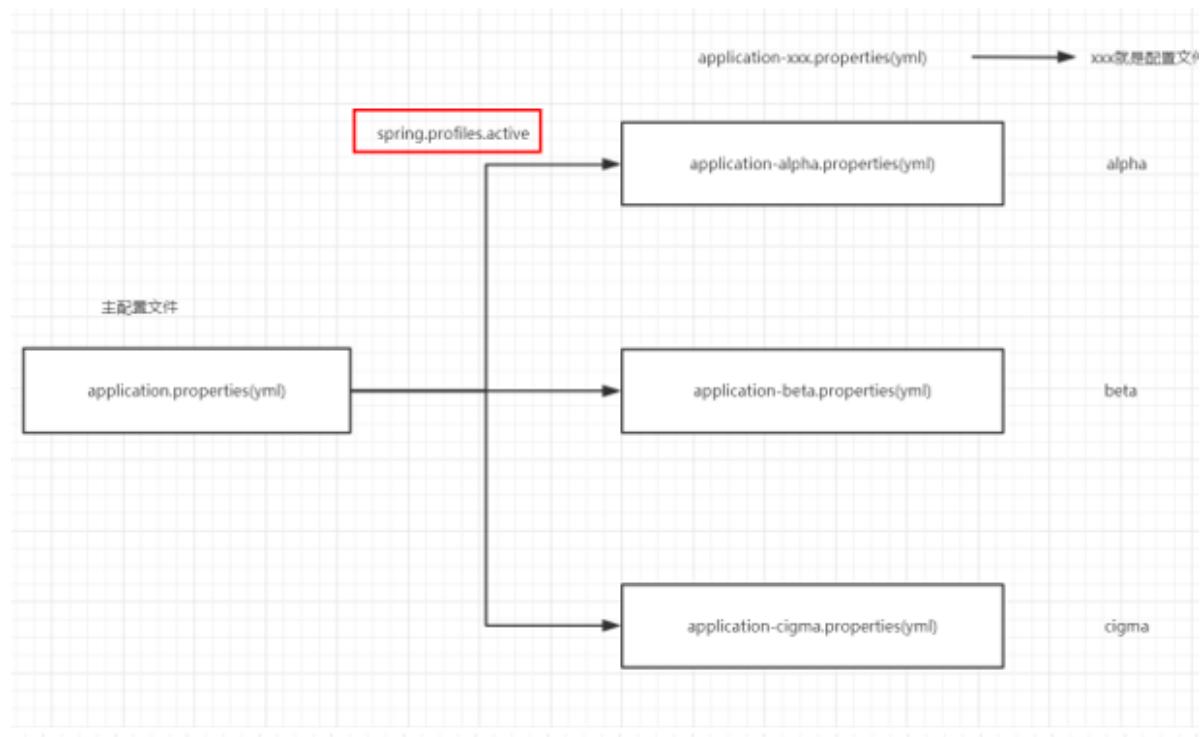
```

The screenshot shows the state of the `FileComponent` object. The `basePath` field is set to `d://spring`, `maxSize` to `512`, and `open` to `true`. The `list1` field contains three elements: `listdata1`, `listdata2`, and `listdata3`. The `list2` field contains three elements: `listdata1`, `listdata2`, and `listdata3`. The `map1` field contains two entries: `key1` with value `value1` and `key2` with value `value2`. The `map2` field contains three entries: `key1` with value `value1`, `key2` with value `value2`, and `key3` with value `value3`. The `fileDetail1` field is an instance of `FileDetail` with parameter `param1` set to `songge`. The `fileDetail2` field is an instance of `FileDetail` with parameters `param1` set to `songge` and `param2` set to `ligenli`.

注：javabean与Map有点像

## 多配置文件（分流，解耦）

properties



yml文件的另一种用法

一个yml文件，可以表达多个配置文件，共用的属性可以放在主配置文件中

```
#上面作为主配置文件
spring:
  profiles:
    active: cigma
    --- 来分隔开多个配置文件
    spring:
      profiles: alpha 指定配置文件的名字
    server:
      port: 8081
    ---
    spring:
      profiles: beta
    server:
      port: 8082
    ---
    spring:
      profiles: cigma
    server:
      port: 8083
```

### 特定值的配置 (没什么用)

random: 在容器初始化的时候完成赋值  
\${}

### 配置文件中引用本文件中已存在的key (常用)

```
#上面作为主配置文件
spring:
  profiles:
    active: cigma
  file3:
    max-size: ${random.int}
    base-path: e:/spring/xxx 引用了配置文件中的key
    png-path: ${file3.base-path}/png
    jpg-path: ${file3.base-path}/jpg
    xml-path: ${file3.base-path}/xml
```

## 引用外部配置文件 (不常用)

在启动类上加注解，引用外部的springboot配置文件 @PropertySource

```
@SpringBootApplication  
@PropertySource(value = "classpath:file3.properties")  
public class Demo3Application {  
    public static void main(String[] args) { SpringApplication.run(Demo3Application.class, args); }  
}
```

放在启动类加载额外的配置文件

## 引用外部的spring配置文件 (引入老项目的配置时用)

springboot建议使用javaconfig注册组件

```
@SpringBootApplication  
@PropertySource(value = "classpath:file3.properties")  
@ImportResource(locations = "classpath:beans.xml")  
public class Demo3Application {  
    public static void main(String[] args) { SpringApplication.run(Demo3Application.class, args); }  
}
```

引入额外的spring配置文件  
但是建议使用javaconfig注册组件

## 配置文件启动的优先级

主要给运维使用

---

## 约定大于配置原理

过程：

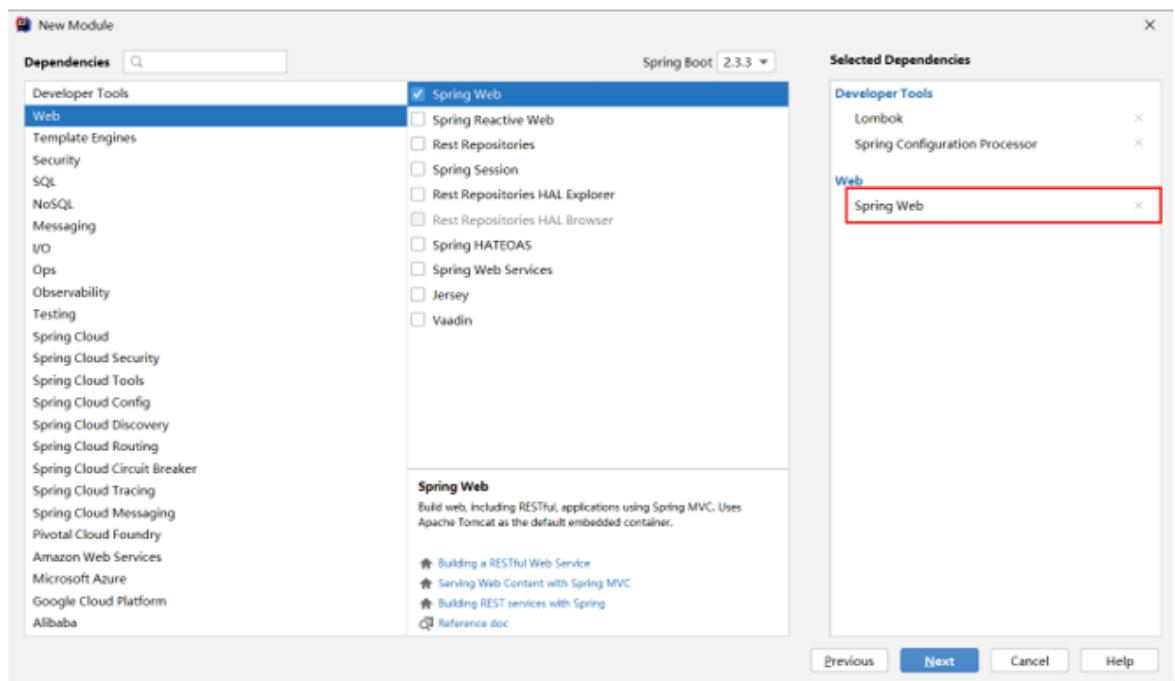
- @SpringBootApplication -> @EnableAutoConfiguration (允许自动配置)  
-> selector -> META-INF/spring.factories(jar)  
(到各个依赖的这个目录下找到factories文件，里面都是该依赖的自动配置类  
即每个框架都提供了注册默认组件的配置类)
- 然后形成一个list：自动配置类的list，  
让自动配置类帮我们完成组件注册的工作

配置类做的事情：

- @ConditionalOn (如果组件已注册，该段代码生效)  
@ConditionalOnMissing (如果组件还没注册，该段代码生效)  
即：如果该组件已注册，就按已经注册的来，否则就按默认的来 (约定大于配置)

## springboot对springMVC的支持

引入依赖 spring-boot-starter-web



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

## 静态资源映射

默认的mapping: /\*\*

```
"name": "spring.resources.static-locations",      对应的就是mvc: resources里的location属性
"type": "java.lang.String[]",
"description": "Locations of static resources. Defaults to classpath:[\\META-INF\\resources\\/, \\resources\\/], [\\static\\/, \\public\\/]"
"sourceType": "org.springframework.boot.autoconfigure.web.ResourceProperties",
"defaultValue": [
    "classpath:\\META-INF\\resources\\/",
    "classpath:\\resources\\/",
    "classpath:\\static\\/",
    "classpath:\\public\\/"
]

spring:
  mvc:          可以自己在yml文件中写自定义的配置 <mvc:resources
    static-path-pattern: /pic/**           mapping
  resources:
    static-locations: file:d:/spring/, classpath:/static/   location
```

## mvc相关的配置

一个是在配置文件中输入spring.mvc然后去找配置

二可以使用javaconfig的方式进行配置

写springMVC的配置类时：

- @EnableWebMvc：全面接管springmvc的配置，其他的配置全部不生效
- @Configuraition：和在配置文件中的配置是互补的配置（增量）

### Converter配置

只需要注册到容器中即可 @Component

```
@Component
public class String2DateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String s) {
        return null;
    }
}
```

## springboot对Mybatis的支持

步骤：

- mybatis-spring-boot-starter
- mysql-connector-java（指定版本5...自己用的就是5..的版本）
- 在启动类上面 + @MapperScan注解（要不然找不到Mapper文件）
- 在yml文件中配置数据库连接池相关信息

```
spring:
  mvc:
    static-path-pattern: /pic/**
  resources:
    static-locations: file:/spring/,classpath:/static/
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/j23_db
    username: root
    password: 123456

@SpringBootApplication
@MapperScan("com.cskaoyan.mapper")
public class Demo4Application {

    public static void main(String[] args) { SpringApplication.run(Demo4Application.class, args); }

}
```

注意：不要随便引用一个还没使用的框架的依赖，因为有的组件需要配置一些属性才能初始化

默认数据源

com.zaxxer.hikari.HikariDataSource

修改数据源

```
spring:  
  mvc:  
    static-path-pattern: /pic/**  
  resources:  
    static-locations: file:d:/spring/,classpath:/static/  
  datasource:  
    driver-class-name: com.mysql.jdbc.Driver  
    url: jdbc:mysql://localhost:3306/j23_db  
    username: root  
    password: 123456  
  type: com.alibaba.druid.pool.DruidDataSource
```

mybatis自身的一些配置

```
mybatis:  
  type-handlers-package:  
  type-aliases-package:
```

彩蛋: banner

## day 2

### 访问

官网:

<http://182.92.235.201/>

两部分:

- 后台管理
- 小程序

抓包:

url ↔ handler方法中的@RequestMapping

形参的接收：

请求参数如何封装（是否封装为json）

json建议使用javabean

可以进行参数校验

json {} 表示对象 [] 表示List

方法的返回值：对应response中的数据

## 文件上传

到一个地方

图片回显

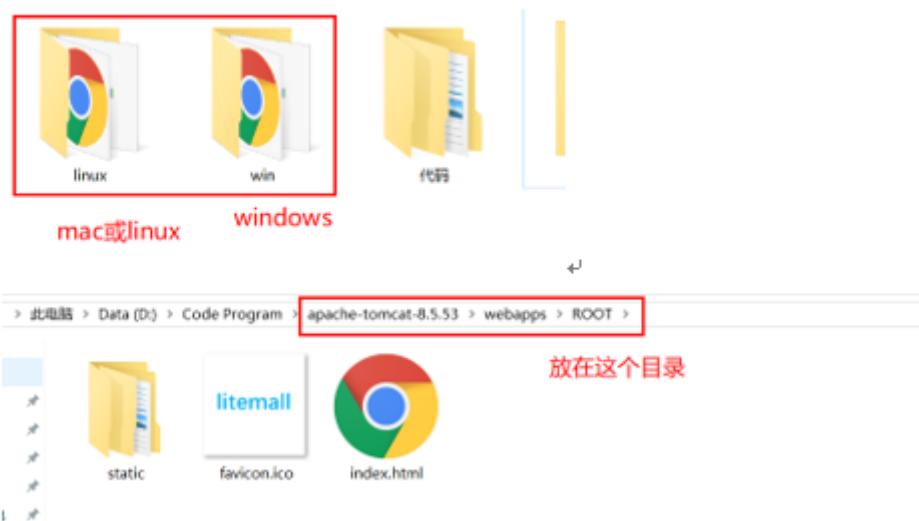
静态资源映射

## 前端应用的启动

前端文件放在root下面的webapp

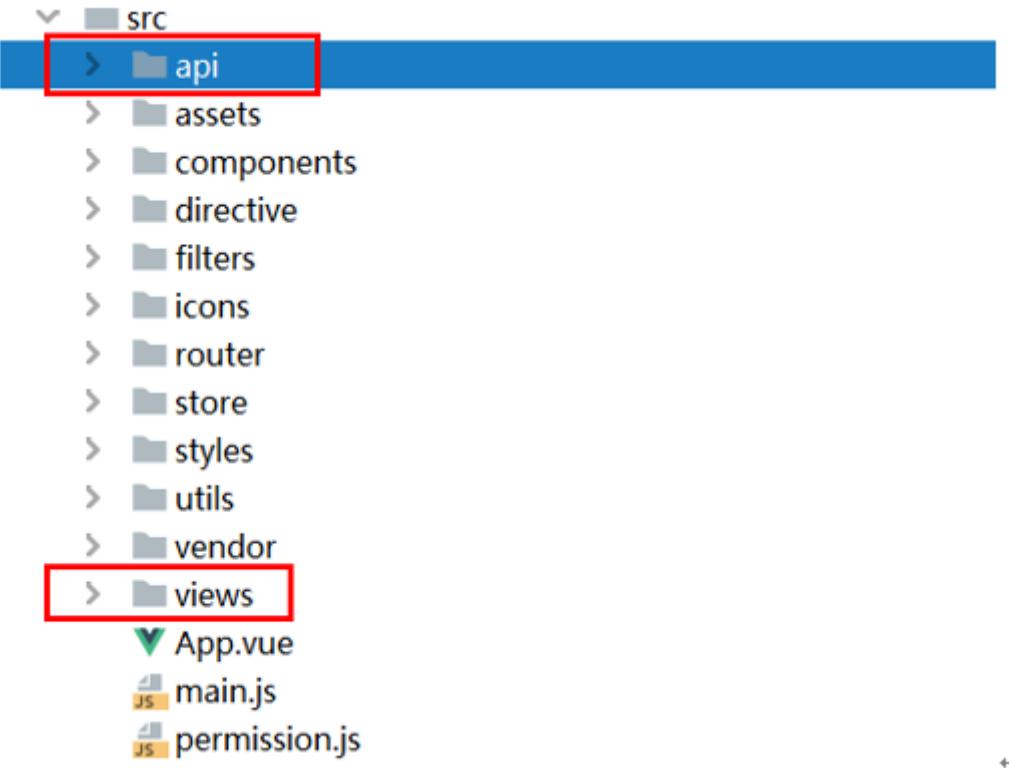
启动tomcat

本地前端应用的启动 ↗



npm运行（可以修改前端代码）

npm run dev



## 把小组的git仓库先建好

- 创建仓库
- 小组成员 -> contributor
- 在仓库的根目录下创建一个.gitignore
- 不要使用idea里的工具：用bash命令行  
看git status 提交的是什么  
然后再执行git add  
git add User\*  
git add \*.java

.idea文件夹写相对路径：.idea/

target 考虑project是否在仓库的根目录

### 分支的使用

master

#### master dev(用这个)

项目搭建好后，在master上创建一个dev分支

在master分支上执行 git checkout dev 切换到dev分支

开发阶段都在dev分支上开发

## master dev 个人分支

master上开启dev分支

再从dev上开启个人分支 如: dev\_ysp

每个人在个人分支上开发，每开发一定的功能，再将个人分支上开发的内容提交到dev上  
(把个人分支合并到dev分支中)

## 注意事项

- 保证代码没有编译错误，能够正常运行再提交代码
- 及时提交：差不多半天提交一次
- 冲突处理很正常

## 搭建一个后端应用的架子

- 引入Mybatis和springMVC框架
- 注册corsFilter解决跨域问题（前后端分离项目，端口号发生变化）

```
@Configuration
public class CorsConfig {
    private CorsConfiguration buildConfig() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.addAllowedOrigin("*"); // 1 设置访问源地址
        corsConfiguration.addAllowedHeader("*"); // 2 设置访问源请求头
        corsConfiguration.addAllowedMethod("*"); // 3 设置访问源请求方法
        return corsConfiguration;
    }

    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration( path: "/**", buildConfig()); // 4 对接口配置跨域设置
        return new CorsFilter(source);
    }
}
```

## handler方法

- 请求参数是什么
- 响应结果是什么

debug思路

shiro

可以去掉这个值: 5000

## 数据库

The screenshot shows the MySQL Workbench interface. On the left, the database tree for 'test23' is visible, containing various tables like 'cotacymall\_ad', 'cotacymall\_admin', etc. In the center, a table named 'cotacymall\_comment' is selected. To its right, the DDL code for this table is displayed:

```
CREATE TABLE `cotacymall_comment` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL DEFAULT '0' COMMENT '如果是评论人;如果是type=0, 则是商品评论;如果是type=1, 则是专题评论',
  `type` tinyint(2) NOT NULL DEFAULT '0' COMMENT '评论类型: 如果type=0, 则是商品评论;如果是type=1, 则是专题评论',
  `content` varchar(1000) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '评论内容',
  `user_id` int(11) NOT NULL DEFAULT '0' COMMENT '评论者的用户ID',
  `head_id` int(11) NOT NULL DEFAULT '0' COMMENT '评论的头ID',
  `pic_url` varchar(1000) DEFAULT NULL COMMENT '图片地址列表, 使用JSON数据格式',
  `star` smallint(6) DEFAULT '1' COMMENT '评分, 1-5',
  `add_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP COMMENT '更新时间',
  `delete_flag` tinyint(1) DEFAULT '0' COMMENT '逻辑删除',
  PRIMARY KEY (`id`) USING BTREE,
  KEY `id_value` (`value_id`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=5012 DEFAULT CHARSET=utf8 ROW_FORMAT=DYNAMIC COMMENT='评论表';
```

Below the table structure, there is a dropdown menu set to 'utf8mb4\_general\_ci'.

可以看ddl(创建表的语句)

## mybatis逆向工程

根据数据库的表创建对应的javabean mapper 映射文件

不要在已有的项目或module中创建

### 依赖

```
<dependencies>
  <dependency>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-core</artifactId>
    <version>1.4.0</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
  </dependency>
</dependencies>
```

## main方法

main方法下的file路径：工作目录 + 相对路径

```
public class Generator {  
    public void generator() throws Exception{  
        List<String> warnings = new ArrayList<~>();  
        boolean overwrite = true; //指向逆向工程配置文件  
        File configFile = new File( pathname: "src/main/resources/generatorConfig.xml");  
        System.out.println(configFile.getAbsolutePath());  
        ConfigurationParser cp = new ConfigurationParser(warnings);  
        Configuration config = cp.parseConfiguration(configFile);  
        DefaultShellCallback callback = new DefaultShellCallback(overwrite);  
        MyBatisGenerator myBatisGenerator =  
            new MyBatisGenerator(config, callback, warnings);  
  
        myBatisGenerator.generate( callback: null);  
    }  
  
    public static void main(String[] args) throws Exception {  
        try {  
            Generator generatorSqlmap = new Generator();  
            generatorSqlmap.generator();  
        }  
        catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

加载配置文件

关注配置文件

生成路径

```

<javaModelGenerator targetPackage="com.cskaoyan.bean"
                     targetProject=".\\src\\main\\java">
    <!-- enableSubPackages: 是否允许子包, 是否让schema作为包的后缀
       即targetPackage.schemaName.tableName -->
    <property name="enableSubPackages" value="true" />
    <!-- 从数据库返回的值是否清理前后的空格 -->
    <property name="trimStrings" value="true" />
</javaModelGenerator>

<sqlMapGenerator targetPackage="com.cskaoyan.mapper"
                  targetProject=".\\src\\main\\resources">
    <!-- enableSubPackages: 是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="true" />
</sqlMapGenerator>

<javaClientGenerator type="XMLMAPPER"
                      targetPackage="com.cskaoyan.mapper"
                      targetProject=".\\src\\main\\java">
    <!-- enableSubPackages: 是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="true" />
</javaClientGenerator><!--> mac或linux/

```

## 生成的方法

单表操作足够了

byExample

createCriteria

criteria

and方法：字符串拼接

每次都新建一个example (非线程安全)

```

@Test
public void mytest1(){
    AdMapper mapper = sqlSession.getMapper(AdMapper.class);
    AdExample adExample = new AdExample(); //每次构造条件时构造一个新的example
    AdExample.Criteria criteria = adExample.createCriteria();
    //criteria.andIdIn(Arrays.asList(1,2,3));
    //criteria.andNameLike("% %");
    //criteria.andPositionEqualTo((byte) 1);
    //criteria.setEnabledEqual(true);
    criteria.andIdIn(Arrays.asList(1, 2, 3)).andNameLike(value: "% %").andPositionEqualTo((byte) 1).setEnabledEqual(true)
    List<Ad> ads = mapper.selectByExample(adExample);
    for (Ad ad : ads) {
        System.out.println(ad);
    }
}

```

如果是Mysql关键字要加` `

byPrimaryKey

```

<delete id="deleteByPrimaryKey" parameterType="java.lang.Integer">
    <!--
        WARNING - @mbg.generated
        This element is automatically generated by MyBatis Generator, do not modify.
        This element was generated on Tue Sep 01 11:48:01 CST 2020.
    -->
    delete from cskaoyanmall_ad
    where id = #{id,jdbcType=INTEGER}
</delete>

```

按照主键

selective

根据变量是否为null，选择性的插入和更新

注意事项：

- 生成映射文件时，先删除已有的映射文件（因为是增量更新）
- 不要在已有的项目中使用逆向工程
- 在逆向工程配置文件中配置包名

```

<javaModelGenerator targetPackage="com.cskaoyan.bean"
                     targetProject=".\\src\\main\\java">
    <!-- enableSubPackages: 是否允许子包, 是否让schema作为包的后缀
        即targetPackage.schemaName.tableName -->
    <property name="enableSubPackages" value="true" />
    <!-- 从数据库返回的值是否清理前后的空格 -->
    <property name="trimStrings" value="true" />
</javaModelGenerator>
<sqlMapGenerator targetPackage="com.cskaoyan.mapper"
                  targetProject=".\\src\\main\\resources">
    <!-- enableSubPackages: 是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="true" />
</sqlMapGenerator>

```

包名和你项目的包名一致

```

<javaClientGenerator type="XMLMAPPER"
                      targetPackage="com.cskaoyan.mapper"
                      targetProject=".\\src\\main\\java">
    <!-- enableSubPackages: 是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="true" />
</javaClientGenerator><!-->

```

## 分页插件

pageHelper

引入依赖：

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.2.5</version>
</dependency>
```

**配置：**

```
    } pagehelper:  
        helper-dialect: mysql
```

```
if (page != null && limit != null) {
    PageHelper.startPage(page, limit);
}

GoodsExample example = new GoodsExample();
if (name != null){
    example.createCriteria().andNameLike("%" + name + "%");
}
List<Goods> goods = mapper.selectByExample(example);
PageInfo<Goods> pageInfo = new PageInfo<Goods>(goods); //在pageInfo中放入查询结果
long total = pageInfo.getTotal();

return goods;
```

`pageInfo`: 可以符合条件的总的记录数

**指定包 配置日志级别**

```
logging:  
  level:  
    com.cskaoyan.mapper: debug
```

## debug思路

对每一行代码都有一个预期值

debug运行到改行：将实际值与预期值比较

- 符合预期：向下看
  - 不符合预期：向上看

P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	
商场管理																					
行政区域		品牌制造商				商品类目				订单管理				通用问题				关键词			
admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin	admin		
region	brand	brand	brand	brand	category	category	category	category	order	order	order	order	order	issue	issue	issue	issue	keyword	keyword		
list	list	create	update	delete	list	create	update	delete	list	detail	refund	ship	list	create	issue	update	delete	create	update		
	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

找到特殊字符 **ctrl + shit + f**



条件查询：

```
@Override
public OrderListVo orderList(OrderListBo orderListBo) {
    Integer page = orderListBo.getPage();
    Integer limit = orderListBo.getLimit();
    OrderListVo orderListVo = new OrderListVo();
    if (page != null && limit != null) {
        PageHelper.startPage(page, limit);
    }
    OrderExample orderExample = new OrderExample();
    OrderExample.Criteria criteria = orderExample.createCriteria();
    orderExample.setOrderByClause(orderListBo.getSort() + " " +
        orderListBo.getOrder());
    if (orderListBo.getUserId() != null) {
        criteria.andUserIdEqualTo(orderListBo.getUserId());
    }
    if (orderListBo.getOrderStatusArray() != null) {
        criteria.andOrderStatusEqualTo((short)
    (int)orderListBo.getOrderStatusArray());
    }
    if (!StringUtils.isEmpty(orderListBo.getOrdersn())) {
        criteria.andOrdersnLike("%" + orderListBo.getOrdersn() + "%");
    }

    List<Order> orders = orderMapper.selectByExample(orderExample);
    long total = new PageInfo<>(orders). getTotal();
    orderListVo.setTotal(total);
    orderListVo.setItems(orders);
    return orderListVo;
}
```

# day 4

## #{}和\${}

- #{} 为预编译语句提供参数（会把整个输入的东西当成一个参数）
- \${} 直接进行字符串的拼接 会有sql注入的风险

字符串拼接

```
<select id="selectNameByIdJing" resultType="java.lang.String">
    select username from j23_user_t where id = #{id}
</select>
```

```
Preparing: select username from j23_user_t where id = ?
Parameters: 20(Integer)
Total: 1 #{}  
+J
```

```
<select id="selectNameByIdDollar" resultType="java.lang.String">
    select username from j23_user_t where id = ${id}
</select>
```

```
Preparing: select username from j23_user_t where id = 20
Parameters: ${}
Total: 1
```

sql注入问题：当sql语句形成方式为字符串拼接时 当查询条件中跟了一个or true的条件时 表中的所有记录都会受到sql语句的影响

#{} 输入 20 or 1 = 1 是在数据库中就是 where id = "20 or 1 = 1" （即把输入的参数当成一个字符串整体来看待）

```
<select id="selectNameByIdDollar" resultType="java.lang.String">
    select username from j23_user_t where id = ${id}
</select>

@Test
public void mytest4(){
    List<String> strings = userMapper.selectNameByIdDollar(20 + " or 1=1");
    System.out.println(strings);
}
```

```
Preparing: select username from j23_user_t where id = 20 or 1=1
Parameters:
Total: 20
```

orderByClause中就用到了\${}, 做的是直接的字符串拼接

```
</if>
<if test="orderByClause != null">
    order by ${orderByClause}
</if>
```

## 静态资源映射

文件上传到哪里

```
@RequestMapping("/storage/create")
public BaseRespVo storageCreate(MultipartFile file) throws IOException {
    Storage storage = new Storage();

    String filename = file.getOriginalFilename();
    String contentType = file.getContentType();
    int size = file.getBytes().length;
    String s = UUID.randomUUID().toString();
    String key = s + "-" + filename;
    File address = new File("d:/Spring/storage/", key);
    file.transferTo(address);
    String prefix = "http://localhost:8083/wx/storage/fetch/";
    String url = prefix + key;
    Date date = new Date();

    storage.setKey(key);
    storage.setName(filename);
    storage.setType(contentType);
    storage.setSize(size);
    storage.setUrl(url);
    storage.setAddTime(date);
    storage.setUpdateTime(date);

    Storage returnStorage = mallservice.storageCreate(storage);

    return BaseRespVo.ok(returnStorage);
}
```

```
spring:
  mvc:
    static-path-pattern: /wx/storage/fetch/**
  resources:
    static-locations: file:d:/Spring/storage/
```

## typehandler

处理输入输出映射

json中是数组，java中最好是List

```
gallery=[]
roles_id=[1,2,3]
```

## 小程序

(代替浏览器给后端服务器发送请求)

团购和支付不需要做

## 日志对象

springboot中有自带的logger对象

```
Logger logger = LoggerFactory.getLogger(this.getClass());
```

## jsonFormat

规定Date在json字符串中的日期时间形式 并加上时区信息

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", timezone = "GMT + 8")
private Date addTime;
```

---

## 权限管理

概念：

- 认证 en
- 授权 or
- 认证是授权的前提，即先认证再授权
- 由Subject执行认证和授权

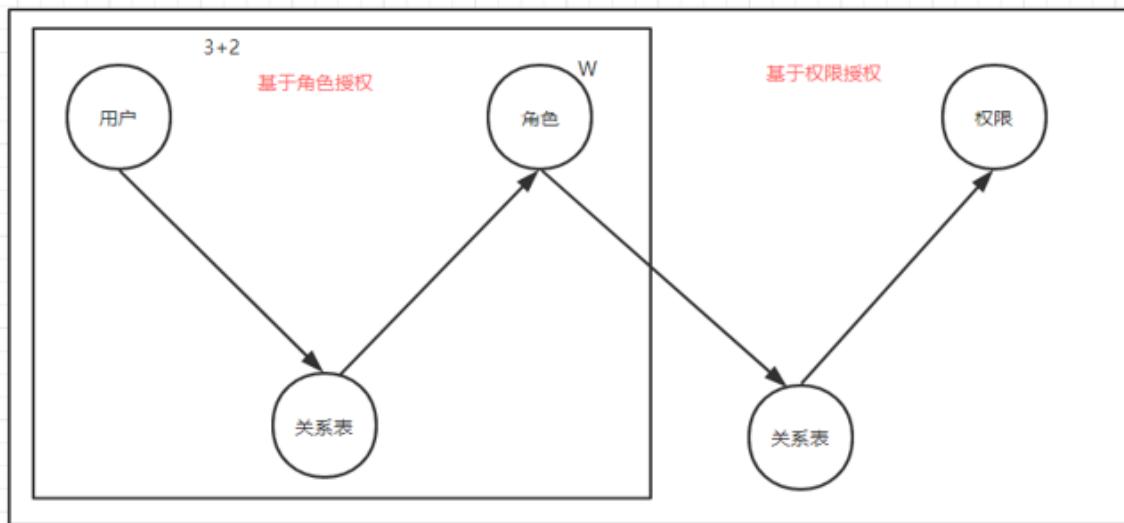
### 权限管理模型

- 基本模型：用户（个体）和权限（名词） 多对多
- 通用模型：用户和角色多对多  
    角色和权限多对多

### 建立在通用模型下的权限管理

- 基于角色授权（条件变更时修改代码太多）
- 基于权限授权（条件变更时添加代码即可）

案例：食堂模型



- 用户与角色建立联系(数据库admin表中的role\_ids)
- 角色与权限建立联系

## shiro

### 模型

- SecurityManager: 安全管理器 (核心组件)
- Authen: 认证
- Author: 授权
- **Realm**: 要在realm中获得用户的:
  - 认证信息 (比对密码)
  - 权限信息 (联表查询获取perm的list进行授权)

### 入门案例

- 引入依赖

```

<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-core</artifactId>
    <version>1.4.1</version>
</dependency>
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>

```

- 认证

```

@Test
public void test1(){
    //首先获得SecurityManager
    IniSecurityManagerFactory securityManagerFactory = new IniSecurityManagerFactory( iniResourcePath: "classpath:first.ini");
    SecurityManager securityManager = securityManagerFactory.getInstance();
    //获得subject
    SecurityUtils.setSecurityManager(securityManager);
    Subject subject = SecurityUtils.getSubject();
    //subject执行认证(Login)
    //提供执行认证的用户名和密码信息，放入到token中
    UsernamePasswordToken authenticationToken = new UsernamePasswordToken( username: "zhaoge", password: "niubi");
    subject.login(authenticationToken); //通过认证器去执行认证Authenticator

    boolean authenticated = subject.isAuthenticated(); //判断是否认证通过
    System.out.println(authenticated);
}

```

核心步骤: `subject.login(token)`

具体过程:

- ModularRealmAuthenticator
- authenticate
- 调用realm (此时还是我们配置的realm)
- 根据realm的size去判断执行single还是multi
- realm中的 `doGetAuthenticationInfo` 返回值为info
- 获得的authenticationInfo信息与通过token的username去获得存在于ini文件中的信息进行比对

```

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {
    UsernamePasswordToken upToken = (UsernamePasswordToken)token;
    SimpleAccount account = this.getUser(upToken.getUsername());
    if (account != null) {
        if (account.isLocked()) {
            throw new LockedAccountException("Account [" + account + "] is locked.");
        }

        if (account.isCredentialsExpired()) {
            String msg = "The credentials for account [" + account + "] are expired";
            throw new ExpiredCredentialsException(msg);
        }
    }

    return account;
}

```

是我们后续需要自己实现的一个方法

- 授权

ini文件中的配置 (之后数据是在数据库中)

注意：权限是个动名词:n:v (即对某个资源可以进行什么操作)

```
[users]
songge=zhenhuai,role1          密码后增加角色，可增加多个角色
zhaoge=niubi,role2             用户和角色 多对多
heidashuai=fangjia,role3

[roles]
role1=user:query,user:update   角色和权限 多对多
role2=user:delete,user:insert
role3=user:query,user:update,user:delete,user:insert
```

### 对于角色的判断

```
private void authorBaseRole(Subject subject) {
    boolean role1 = subject.hasRole("role1");      对单个角色判断
    System.out.println("是否拥有role1的角色: " + role1);

    ArrayList<String> roleList = new ArrayList<>();
    roleList.add("role1");
    roleList.add("role2");
    roleList.add("role3");
    boolean[] hasRoles = subject.hasRoles(roleList); 分别对每一个角色进行判断
    System.out.println("对role123分别拥有的角色: " + Arrays.toString(hasRoles));

    boolean hasAllRoles = subject.hasAllRoles(roleList); //判断list里的权限是否全部拥有
    System.out.println("是否拥有全部角色: " + hasAllRoles); 判断是否全为true
}
```

### 对于权限的判断

```
private void authorBasePermmision(Subject subject) {
    String insertPermission = "user:insert";
    String deletePermission = "user:delete";
    String updatePermission = "user:update";
    String queryPermission = "user:query";           逐个权限进行判断
    boolean[] permitteds = subject.isPermitted(insertPermission, deletePermission, updatePermission, queryPermission);
    System.out.println("增删改查多个的权限: " + Arrays.toString(permitteds));

    boolean permitted1 = subject.isPermitted(queryPermission); 对单个权限进行判断
    System.out.println("单个权限: " + permitted1);

    boolean permittedAll = subject.isPermittedAll(insertPermission, deletePermission, updatePermission, queryPermission); 权限是否全为true
    System.out.println("是否拥有全部权限: " + permittedAll);
}
```

## Realm

realm:

在realm中返回认证和授权信息

认证器: modularRealmAuthenticator

**doGetAuthenticationInfo**

**doGetAuthorizationInfo**

[users]

[roles]

Subject

SecurityManager

Authenticator

Authorizer

permission

## 自定义的realm

获得当前用户的认证信息和授权信息

导包:

Shiro-core

commons-logging

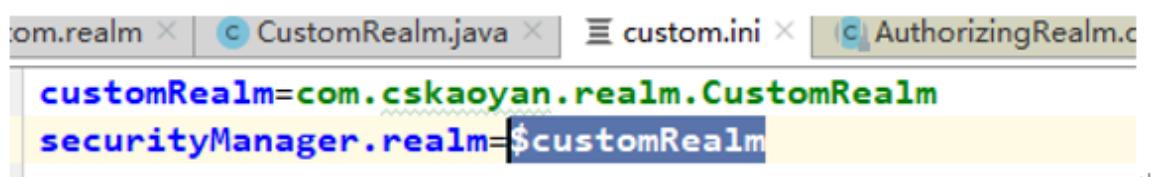
junit

自定义的realm: 继承AuthorizingRealm

```
public class CustomRealm extends AuthorizingRealm {  
    @Override  
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  
        // 处理授权信息  
        return null;  
    }  
  
    @Override  
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {  
        // 处理认证信息  
        return null;  
    }  
}
```

配置文件:

设置SecurityManager的realm为自定义的realm



## 在sb中整合shiro框架

- 引入依赖: shiro-spring

```

<!--shiro对spring的支持-->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.4.1</version>
</dependency>

```

shiro-web和shiro-core

- 注册组件：（在自定义的配置类中 @Configuration）

- ShiroFilterFactoryBean

filter来过滤url请求

loginUrl：如果没有认证通过，则重定向到loginUrl(登录页面)

```

@Bean
public ShiroFilterFactoryBean shiroFilterFactoryBean(DefaultWebSecurityManager securityManagerz){
    ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();
    //认证失败后重定向的url
    shiroFilterFactoryBean.setLoginUrl("/unAuthc");
    shiroFilterFactoryBean.setSecurityManager(securityManagerz);
    //最重要的事情
    //对请求过滤 filter
    //key 请求url value对应的是过滤器
    LinkedHashMap<String, String> filterMap = new LinkedHashMap<>();
    //login(username,password) success
    //失败则重新登录

    filterMap.put("/auth/login", "anon");匿名拦截器
    filterMap.put("/unAuthc", "anon");
    filterMap.put("/index", "anon");
    //当你分配了perm1的权限时才能访问need/perm这请求
    //filterMap.put("/need/perm", "perms[perm1]");
    //优先的方式是声明式
    //filterMap.put("/auth/logout", "logout");
    filterMap.put("/**", "authc"); 认证拦截器
    shiroFilterFactoryBean.setFilterChainDefinitionMap(filterMap);

    return shiroFilterFactoryBean;
}

```

- SecurityManager

```

@Bean
public DefaultWebSecurityManager securityManagerz(CustomRealm customRealm,DefaultWebSessionManager webSessionManager){
    DefaultWebSecurityManager defaultWebSecurityManager = new DefaultWebSecurityManager();
    defaultWebSecurityManager.setRealm(customRealm);
    defaultWebSecurityManager.setSessionManager(webSessionManager());
    defaultWebSecurityManager.setSessionManager(webSessionManager);
    return defaultWebSecurityManager;
}

```

- SessionManager (token)

```

public class CustomSessionManager extends DefaultWebSessionManager { 继承sessionManager
    @Override
    protected Serializable getSessionId(ServletRequest srequest, ServletResponse response) { 重写其获得sessionId方法
        HttpServletRequest request = (HttpServletRequest) srequest;
        String sessionId = request.getHeader("X-cskaoyan-mall-Admin-Token");
        if (sessionId != null && !"".equals(sessionId))
        {
            return sessionId;
        }
        return super.getSessionId(request, response);
    }
}

```

- AuthorizationAttributeSourceAdvisor (通知器)

需要引入aspectjweaver依赖

```

/*
 * 声明式授权 注解需要的组件
 */
@Bean
public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(DefaultWebSecurityManager securityManagerz){
    AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = new AuthorizationAttributeSourceAdvisor();
    authorizationAttributeSourceAdvisor.setSecurityManager(securityManagerz);
    return authorizationAttributeSourceAdvisor;
}

```

- 执行认证

```

/*
 * 认证
 */
@RequestMapping("auth/login")
@ResponseBody
public String login(String username, String password){
    Subject subject = SecurityUtils.getSubject();
    try {
        subject.login(new UsernamePasswordToken(username, password));
    } catch (AuthenticationException e) {
        return "forward:/index";
        //e.printStackTrace();
    }
    Serializable id = subject.getSession().getId();
    return (String) id;
}

```

- 登出

```

@RequestMapping("auth/logout")
public String logout(){
    Subject subject = SecurityUtils.getSubject();
    subject.logout();
    return "/index";
}

```

- 获得当前用户信息

```

@RequestMapping("info")
public BaseRespVo info(String token){
    Subject subject = SecurityUtils.getSubject();
    String principal = (String) subject.getPrincipal();
}

```

- 过滤器

过滤器简称	对应的 java 类
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter
user	org.apache.shiro.web.filter.authc.UserFilter
logout	org.apache.shiro.web.filter.authc.LogoutFilter

- anon:

例如 /admins/\*\* = anon 没有参数表示可以匿名使用

- authc:

例如: /admins/user/\*\* = authc 表示需要认证 (登录) 才能使用

FormAuthenticationFilter是表单认证

- perms:

例如: /admins/user/\*\* = perms[user:add]

参数可以写多个，多个参数之间必须加上引号，并且参数之间用逗号分隔

/admins/user/\*\* = perms[user:add, user:modify]

当有多个参数是必须每个参数都通过才能通过（默认）

- o user:

例如：/admins/user/\*\* = user

没有参数表示必须存在用户，用户身份认证通过或记住认证通过

当执行登入操作时不做检查

# Week 18

## day1

### 多个realm处理不同账号体系的认证



- 在容器中注册多个realm

```
@Component ←  
public class WxRealm extends AuthorizingRealm {  
  
    @Component ←  
    public class AdminRealm extends AuthorizingRealm {
```

- 自定义token信息（自己定义一个类继承UsernamePasswordToken类 并添加type属性）

```

    @RequestMapping("admin/login")
    public String adminLogin(String username, String password){
        MallToken adminToken = new MallToken(username, password, type: "admin");
        Subject subject = SecurityUtils.getSubject();
        subject.login(adminToken);
        return "success";
    }
    @RequestMapping("wx/login")
    public String wxLogin(String username, String password){
        MallToken wxToken = new MallToken(username, password, type: "wx");
        Subject subject = SecurityUtils.getSubject();
        subject.login(wxToken);
        return "success";
    }
}

```

不同的认证携带不同的type信息

```

@Data
public class MallToken extends UsernamePasswordToken {
    String type; 增加了type信息
    public MallToken(String username, String password, String type) {
        super(username, password);
        this.type = type;
    }
}

```

- 注册自定义认证器

自定义认证器: `ModularRealmAuthenticator`

并且重写父类的 `doAuthenticate` 方法, 修改其中的 `realms` 数据

通过 `token` 携带的 `type` 信息进行筛选, 分派到具体的 `realm` 上面

```

public class CustomAuthenticator extends ModularRealmAuthenticator {
    @Override
    protected AuthenticationInfo doAuthenticate(AuthenticationToken authenticationToken) throws AuthenticationException {
        this.assertRealmsConfigured();
        Collection<Realm> originRealms = this.getRealms();
        //对realms来进行分发
        MallToken token = (MallToken) authenticationToken;
        String type = token.getType();
        ArrayList<Realm> realms = new ArrayList<>();
        for (Realm originRealm : originRealms) {
            //AdminRealm adminrealm admin/wx
            if (originRealm.getName().toLowerCase().contains(type)){
                realms.add(originRealm);
            }
        }
        return realms.size() == 1 ? this.doSingleRealmAuthentication((Realm)realms.iterator().next(), authenticationToken) :
            this.doMultiRealmAuthentication(realms, authenticationToken);
    }
}

```

通过token中的type信息进行筛选  
筛选出符合当前认证体系下的realm

注册该组件, 并配置其成员变量 `realms`

```

@Bean
public CustomAuthenticator authenticator(AdminRealm adminRealm, WxRealm wxRealm){
    CustomAuthenticator customAuthenticator = new CustomAuthenticator();
    ArrayList<Realm> realms = new ArrayList<>();
    realms.add(adminRealm);
    realms.add(wxRealm);
    customAuthenticator.setRealms(realms);
    return customAuthenticator;
}

```

配置认证器的realms信息

# Druid的拦截统计功能

直接引入druid-spring-boot-starter依赖 配置完数据源即可使用

- 引入依赖

```
<!--springboot 对druid的支持-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.1</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
    <scope>runtime</scope>
</dependency>
```

- 配置数据源

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/j18_db?useUnicode=true&characterEncoding=utf-8
    username: root
    password: 123456
```

- 开始使用

访问localhost:8080/druid 此处填对应你的应用的ip+端口号+context-path+druid访问

The screenshot shows the Druid Monitor interface with the URL `localhost:8080/druid/index.html`. The main content area displays the 'Stat Index' page, which includes a table with various system statistics. The table rows include:

Stat Index 查看JSON API	
版本	1.1.10
驱动	com.alibaba.druid.mock.MockDriver com.mysql.jdbc.Driver com.mysql.jdbc.PureMySQLDriver com.alibaba.druid.proxy.DruidDriver
自动归功	true
连接次数	0
java版本	1.8.0_131
Jvm 容积	Java HotSpot(TM) 64-Bit Server VM
classpath 路径	<ul style="list-style-type: none"><li>!Coding Env:JavaPath1.8.0_131!/jniWithCharsets.jar</li><li>!Coding Env:JavaPath1.8.0_131!/jniWithDeploy.jar</li><li>!Coding Env:JavaPath1.8.0_131!/jniWithAccessOnBridge-64.jar</li><li>!Coding Env:JavaPath1.8.0_131!/jniWithAccessOnData.jar</li><li>!Coding Env:JavaPath1.8.0_131!/jniWithAccessOnString.jar</li><li>!Coding Env:JavaPath1.8.0_131!/jniWithAccessOnAccess.jar</li></ul>

# 阿里云组件

- oss (文件)

依赖:

```
<!--oss-->
<dependency>
    <groupId>com.aliyun.oss</groupId>
    <artifactId>aliyun-sdk-oss</artifactId>
    <version>2.5.0</version>
</dependency>
```

bucket: 文件夹

endPoint: 节点

accessKeyId:

accessKeySecret:

```
@Test
public void mytest() throws JsonProcessingException {
    String accessKeyId = "LTAI4Fr5gfYhcVjLMqeRGB";
    String accessKeySecret = "IrkcHu6dZyrjPZRushg076P5392";
    String endPoint = "oss-cn-beijing.aliyuncs.com";
    String bucket = "cskaoyan";

    //准备个文件
    File file = new File( parent: "C:\\\\Users\\\\Administrator\\\\Desktop", child: "logo.png");
    //FileInputStream fileInputStream = new FileInputStream(file);
    String fileName = UUID.randomUUID().toString().replaceAll( regex: "-", replacement: "") + ".png";
    System.out.println(fileName); //上传完成 要通过bucket+endpoint+文件名进行访问

    OSSClient ossClient = new OSSClient(endPoint, accessKeyId, accessKeySecret);

    /*MultipartFile myfile;
    InputStream inputStream = myfile.getInputStream();*/

    PutObjectResult putObjectResult = ossClient.putObject(bucket, fileName, file);

    ObjectMapper objectMapper = new ObjectMapper();
    String s = objectMapper.writeValueAsString(putObjectResult);
    System.out.println(s);
}
```

id、密码和服务器

sms (短信)

```
<!--SMS-->
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-core</artifactId>
    <version>4.0.3</version>
</dependency>
```

```

    @Test
    public void mytest() throws JsonProcessingException {
        String accessKeyId = "LTAI4Fr5gfYhcVjLMqeRGbuT";
        String accessKeySecret = "IrkcHu6dZyrjPZRushg076P5392HJ1";
        String signName = "stone4j"; 签名
        String templateCode = "SMS_173765187"; 模板code
        String phoneNumber = "13260604399";

        DefaultProfile profile = DefaultProfile.getProfile( regionId: "cn-hangzhou", accessKeyId, accessKeySecret);
        IAcsClient client = new DefaultAcsClient(profile);

        CommonRequest request = new CommonRequest();
        request.setMethod(MethodType.POST);
        request.setDomain("dysmsapi.aliyuncs.com");
        request.setVersion("2017-05-25");
        request.setAction("SendSms");
        request.putQueryParameter( name: "RegionId", value: "cn-hangzhou");
        request.putQueryParameter( name: "PhoneNumbers", phoneNumber); 电话
        request.putQueryParameter( name: "SignName", signName);
        request.putQueryParameter( name: "TemplateCode", templateCode);
        request.putQueryParameter( name: "TemplateParam", value: "{\"code\": \"65536\"}");
        try {
            CommonResponse response = client.getCommonResponse(request);
            System.out.println(response.getData());
        } catch (ServerException e) {
            e.printStackTrace();
        } catch (ClientException e) {
            e.printStackTrace();
        }
    }

```

## 整合思路

通过组件注册的思路进行整合

将配置项放入配置文件中，通过 `@ConfigurationProperties` 注解打通联系

- 组件注册

```

    aliyun:
      accessKeyId: LTAI4Fr5gfYhcVjLMqeRGbuT
      accessKeySecret: IrkcHu6dZyrjPZRushg076P5392HJ1
    oss:
      bucket: cskaoyan
      endPoint: oss-cn-beijing.aliyuncs.com
    sms:
      signName: stone4j
      templateCode: SMS_173765187
  
```

The diagram illustrates the configuration of the `AliyunComponent`. It shows three nested configurations: `aliyun`, `oss`, and `sms`. The `aliyun` section contains `accessKeyId` and `accessKeySecret`. The `oss` section contains `bucket` and `endPoint`. The `sms` section contains `signName` and `templateCode`. Red arrows point from the configuration keys to their corresponding fields in the `AliyunComponent` class. The `AliyunComponent` class itself has annotations: `@Component`, `@ConfigurationProperties(prefix = "aliyun")`, and `@Data`. It contains fields for `accessKeyId` and `accessKeySecret`, along with `Oss` and `Sms` objects.

```

@Component
@ConfigurationProperties(prefix = "aliyun")
@Data
public class AliyunComponent {
    String accessKeyId;
    String accessKeySecret;
    Oss oss;
    Sms sms;
}

@Data
public class Oss {
    String bucket;
    String endPoint;
}

@Data
public class Sms {
    String signName;
    String templateCode;
}
  
```

- 提供服务

```

public OSSClient getOssClient(){
    OSSClient ossClient = new OSSClient(oss.getEndPoint(), accessKeyId, accessKeySecret);
    return ossClient;
}

public PutObjectResult fileUpload(String fileName, File file){
    OSSClient ossClient = getOssClient();
    PutObjectResult putObjectResult = ossClient.putObject(oss.getBucket(), fileName, file);
    return putObjectResult;
}

public PutObjectResult fileUpload(String fileName, InputStream inputStream){
    OSSClient ossClient = getOssClient();
    PutObjectResult putObjectResult = ossClient.putObject(oss.getBucket(), fileName, inputStream);
    return putObjectResult;
}

public void sendMsg(String phoneNumber, String code){
    String signName = sms.getSignName();
    String templateCode = sms.getTemplateCode();

    DefaultProfile profile = DefaultProfile.getProfile(regionId: "cn-hangzhou", accessKeyId, accessKeySecret);
    IAcsClient client = new DefaultAcsClient(profile);

    CommonRequest request = new CommonRequest();
    request.setMethod(MethodType.POST);
    request.setDomain("dysmsapi.aliyuncs.com");
    request.setVersion("2017-05-25");
    request.setAction("SendSms");
    request.putQueryParameter(name: "RegionId", value: "cn-hangzhou");

    request.putQueryParameter(name: "PhoneNumbers", phoneNumber);

    request.putQueryParameter(name: "SignName", signName);
    request.putQueryParameter(name: "TemplateCode", templateCode);
    request.putQueryParameter(name: "TemplateParam", value: "{\"code\":\""+code+"\"}");
}

    e.printStackTrace();
}
System.out.println(response.getData());
}

```

## 加密算法

**对称性加密：**

加密和解密都使用同一个秘钥。8 钥

**非对称加密：**

公私钥是成对存在的

加密解密使用的是不同的秘钥

公（公开）私（私有）钥

- 公钥加密 私钥解密 web多个client server
- 私钥加密 公钥解密
- 私钥加密（私钥签名 公钥验证） 公钥解密

公私

私公

私私公公

## hash算法

不是一种加密算法，它的过程是单向的 不可逆的

特定长度：不管原始值有多大，生成结果都是该特定长度