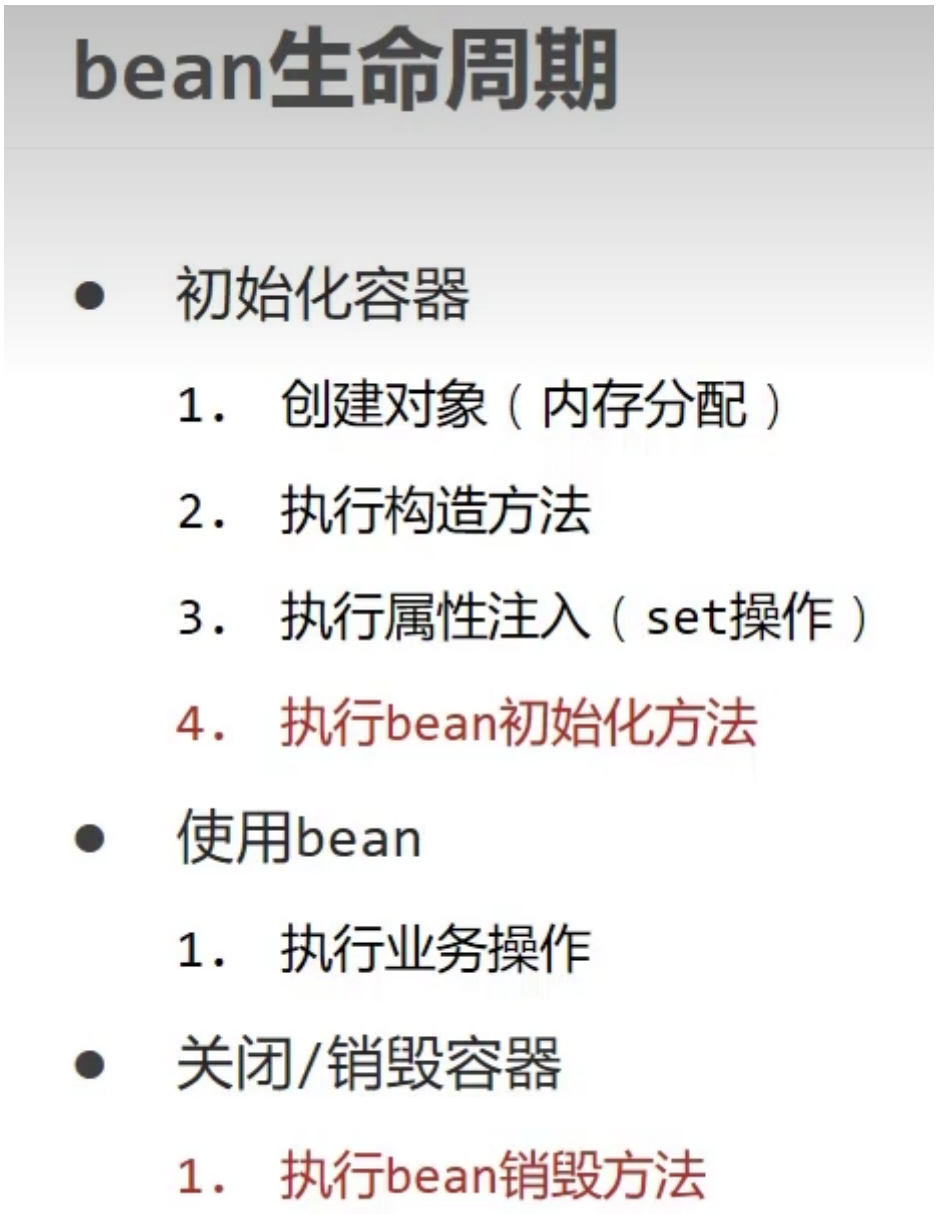
spring

实例化bean4种方式

1. 构造方法
2. 静态工厂
3. 实例工厂
4. FactoryBean



依赖注入方式

- 思考：向一个类中传递数据的方式有几种？
  - 普通方法（set方法）
  - 构造方法
- 思考：依赖注入描述了在容器中建立bean与bean之间依赖关系的过程，如果bean运行需要的是数字或字符串呢？
  - 引用类型
  - 简单类型（基本数据类型与String）
- 依赖注入方式
  - setter注入
    - 简单类型
    - 引用类型
  - 构造器注入
    - 简单类型
    - 引用类型

第三方bean管理

@bean

第三方bean依赖注入

- 引用类型：方法形参
- 简单类型：成员变量

AOP核心概念

- 连接点（JoinPoint）：程序执行过程中的任意位置，粒度为执行方法、抛出异常、设置变量等
  - 在SpringAOP中，理解为方法的执行
- 切入点（Pointcut）：匹配连接点的式子
  - 在SpringAOP中，一个切入点可以只描述一个具体方法，也可以匹配多个方法
    - 一个具体方法：com.itheima.dao包下的BookDao接口中的无形参无返回值的save方法
    - 匹配多个方法：所有的save方法，所有的get开头的方法，所有以Dao结尾的接口中的任意方法，所有带有一个参数的方法
- 通知（Advice）：在切入点处执行的操作，也就是共性功能
  - 在SpringAOP中，功能最终以方法的形式呈现
- 通知类：定义通知的类
- 切面（Aspect）：描述通知与切入点的对应关系

步骤 AOP入门案例（注解版）

⑤：绑定切入点与通知关系，并指定通知添加到原始连接点的具体执行**位置**

```
public class MyAdvice {
    @Pointcut("execution(void com.itheima.dao.BookDao.update())")
    private void pt(){}

    @Before("pt()")
    public void before(){
        System.out.println(System.currentTimeMillis());
    }
}
```

AOP工作流程

1. Spring容器启动
2. 读取所有切面配置中的切入点
3. 初始化bean，判定bean对应的类中的方法是否匹配到任意切入点
   - 匹配失败，创建对象
   - 匹配成功，创建原始对象（**目标对象**）的**代理**对象
4. 获取bean执行方法
   - 获取bean，调用方法并执行，完成操作
   - 获取的bean是代理对象时，根据代理对象的运行模式运行原始方法与增强的内容，完成操作

AOP切入点表达式

- 可以使用通配符描述切入点，快速描述
  - * ：单个独立的任意符号，可以独立出现，也可以作为前缀或者后缀的匹配符出现

```
execution (public * com.itheima.*.UserService.find* (*) )
```

匹配com.itheima包下的任意包中的UserService类或接口中所有find开头的带有一个参数的方法

  - .. ：多个连续的任意符号，可以独立出现，常用于简化包名与参数的书写

```
execution (public User com..UserService.findById (..) )
```

匹配com包下的任意包中的UserService类或接口中所有名称为findById的方法

  - + ：专用于匹配子类类型

```
execution(* *..*Service+.*(..))
```

- @Around注意事项
  1. 环绕通知必须依赖形参ProceedingJoinPoint才能实现对原始方法的调用，进而实现原始方法调用前后同时添加通知
  2. 通知中如果未使用ProceedingJoinPoint对原始方法进行调用将跳过原始方法的执行
  3. 对原始方法的调用可以不接收返回值，通知方法设置成void即可，如果接收返回值，必须设定为Object类型
  4. 原始方法的返回值如果是void类型，通知方法的返回值类型可以设置成void，也可以设置成Object
  5. 由于无法预知原始方法运行后是否会抛出异常，因此环绕通知方法必须抛出Throwable对象

```
@Around("pt()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice ...");
    Object ret = pjp.proceed();
    System.out.println("around after advice ...");
    return ret;
}
```

AOP通知获取数据

- 获取切入点方法的参数
  - JoinPoint：适用于前置、后置、返回后、抛出异常后通知
  - ProceedJointPoint：适用于环绕通知
- 获取切入点方法返回值
  - 返回后通知
  - 环绕通知
- 获取切入点方法运行异常信息
  - 抛出异常后通知
  - 环绕通知

①：在业务层接口上添加Spring事务管理

```
public interface AccountService {
    @Transactional
    public void transfer(String out,String in ,Double money);
}
```

注意事项

Spring注解式事务通常添加在业务层接口中而不会添加到业务层实现类中，降低耦合

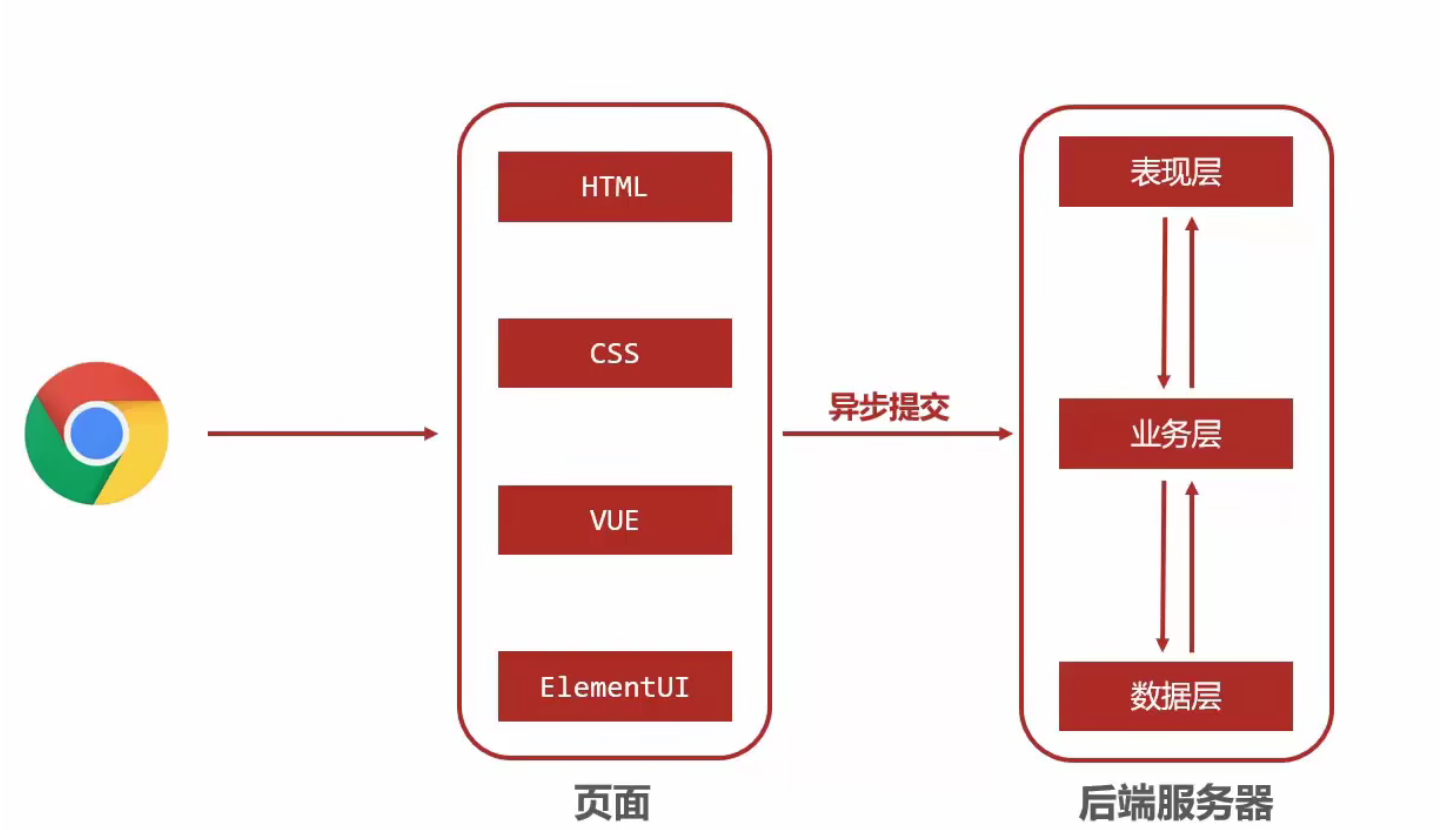
注解式事务可以添加到业务方法上表示当前方法开启事务，也可以添加到接口上表示当前接口所有方法开启事务

事务相关配置

属性	作用	示例
readOnly	设置是否为只读事务	readOnly=true 只读事务
timeout	设置事务超时时间	timeout = -1（永不超时）
rollbackFor	设置事务回滚异常（class）	rollbackFor = {NullPointException.class}
rollbackForClassName	设置事务回滚异常（String）	同上格式为字符串
noRollbackFor	设置事务不回滚异常（class）	noRollbackFor = {NullPointException.class}
noRollbackForClassName	设置事务不回滚异常（String）	同上格式为字符串
propagation	设置事务传播行为	.....

事务传播行为

传播属性	事务管理员	事务协调员
REQUIRED（默认）	开启T	加入T
	无	新建T2
REQUIRES_NEW	开启T	新建T2
	无	新建T2
SUPPORTS	开启T	加入T
	无	无
NOT_SUPPORTED	开启T	无
	无	无
MANDATORY	开启T	加入T
	无	ERROR
NEVER	开启T	ERROR
	无	无
NESTED	设置savePoint，一旦事务回滚，事务将回滚到savePoint处，交由客户响应提交/回滚	




```
//日期参数
@RequestMapping("/dataParam")
@ResponseBody
public String dataParam(Date date,@DateTimeFormat(pattern="yyyy-MM-dd") Date date1){
    System.out.println("参数传递 date ==> "+date);
    System.out.println("参数传递 date1(yyyy-MM-dd) ==> "+date1);
    return "{'module':'data param'}";
}
```

string转date有默认格式，如果需要改格式使用DateTimeFormat

步骤 RESTful入门案例

②：设定请求参数（路径变量）

```
@RequestMapping(value = "/users/{id}" ,method = RequestMethod.DELETE)
@ResponseBody
public String delete(@PathVariable Integer id){
    System.out.println("user delete..." + id);
    return "{'module':'user delete'}";
}
```

rest是一种资源描述风格，restful是以rest风格访问资源

SSM整合

表现层数据封装

- 设置统一数据返回结果类

```
public class Result {
    private Object data;
    private Integer code;
    private String msg;
}
```

异常处理

异常处理器

- 异常处理器
  - 集中的、统一的处理项目中出现的异常

```
@RestControllerAdvice
public class ProjectExceptionAdvice {
    @ExceptionHandler(Exception.class)
    public Result doException(Exception ex){
        return new Result(666,null);
    }
}
```

项目异常处理方案

- 项目异常处理方案
  - 业务异常（BusinessException）
    - 发送对应消息传递给用户，提醒规范操作
  - 系统异常（SystemException）
    - 发送固定消息传递给用户，安抚用户
    - 发送特定消息给运维人员，提醒维护
    - 记录日志
  - 其他异常（Exception）
    - 发送固定消息传递给用户，安抚用户
    - 发送特定消息给编程人员，提醒维护（纳入预期范围内）
    - 记录日志

步骤 项目异常处理

①：自定义项目系统级异常

```
public class SystemException extends RuntimeException{
    private Integer code;
    public SystemException(Integer code,String message) {
        super(message);
        this.code = code;
    }

    public SystemException( Integer code,String message, Throwable cause) {
        super(message, cause);
        this.code = code;
    }

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }
}
```

步骤 项目异常处理

④：触发自定义异常

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    public Book getById(Integer id) {
        if( id < 0 ){
            throw new BusinessException(Code.PROJECT_BUSINESS_ERROR,"请勿进行非法操作！");
        }
        return bookDao.getById(id);
    }
}
```

3): 拦截并处理异常

```java
@RestControllerAdvice
public class ProjectExceptionAdvice {
    @ExceptionHandler(BusinessException.class)
    public Result doBusinessException(BusinessException ex){
        return new Result(ex.getCode(),null,ex.getMessage());
    }
    @ExceptionHandler(SystemException.class)
    public Result doSystemException(SystemException ex){
        // 记录日志（错误堆栈）
        // 发送邮件给开发人员
        // 发送短信给运维人员
        return new Result(ex.getCode(),null,ex.getMessage());
    }
    @ExceptionHandler(Exception.class)
    public Result doException(Exception ex){
        // 记录日志（错误堆栈）
        // 发送邮件给开发人员
        // 发送短信给运维人员
        return new Result(Code.SYSTEM_UNKNOW_ERROR,null,"系统繁忙，请联系管理员！");
    }
}
```

拦截器

- 拦截器（Interceptor）是一种动态拦截方法调用的机制
- 作用：
  - 在指定的方法调用前后执行预先设定后的的代码
  - 阻止原始方法的执行

拦截器与过滤器区别

- 归属不同：Filter属于Servlet技术，Interceptor属于SpringMVC技术
- 拦截内容不同：Filter对所有访问进行增强，Interceptor仅针对SpringMVC的访问进行增强

步骤 拦截器入门案例

①：声明拦截器的bean，并实现HandlerInterceptor接口（注意：扫描加载bean）

```
@Component
public class ProjectInterceptor implements HandlerInterceptor {
    public boolean preHandle(..) throws Exception {
        System.out.println("preHandle...");
        return true;
    }
    public void postHandle(..) throws Exception {
        System.out.println("postHandle...");
    }
    public void afterCompletion(..) throws Exception {
        System.out.println("afterCompletion...");
    }
}
```

②：定义配置类，继承WebMvcConfigurationSupport，实现addInterceptor方法（注意：扫描加载配置）

```
@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        ...
    }
}
```

③：添加拦截器并设定拦截的访问路径，路径可以通过可变参数设置多个

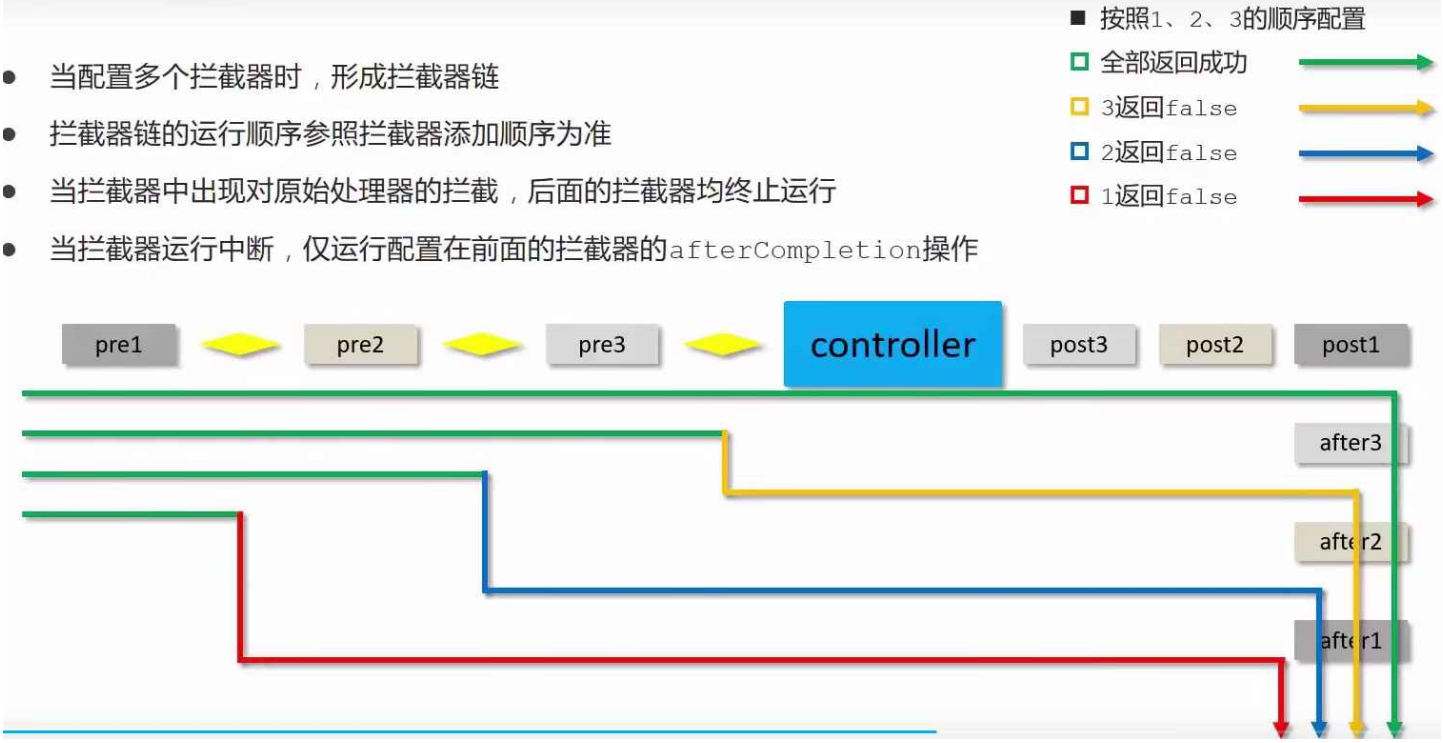
```
@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
    @Autowired
    private ProjectInterceptor projectInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(projectInterceptor).addPathPatterns("/books");
    }
}
```

2．拦截器执行顺序

- preHandle
  - return true
    - controller
    - postHandle
    - afterCompletion
  - return false
    - 结束

多拦截器执行顺序

- 当配置多个拦截器时，形成拦截器链
- 拦截器链的运行顺序参照拦截器添加顺序为准
- 当拦截器中出现对原始处理器的拦截，后面的拦截器均终止运行
- 当拦截器运行中断，仅运行配置在前面的拦截器的afterCompletion操作



聚合

- 聚合：将多个模块组织成一个整体，同时进行项目构建的过程称为聚合
- 聚合工程：通常是一个不具有业务功能的“空”工程（有且仅有一个pom文件）
- 作用：使用聚合工程可以将多个工程编组，通过对聚合工程进行构建，实现对所包含的模块进行同步构建
  - 当工程中某个模块发生更新（变更）时，必须保障工程中与已更新模块关联的模块同步更新，此时可以使用聚合工程来解决批量模块同步构建的问题

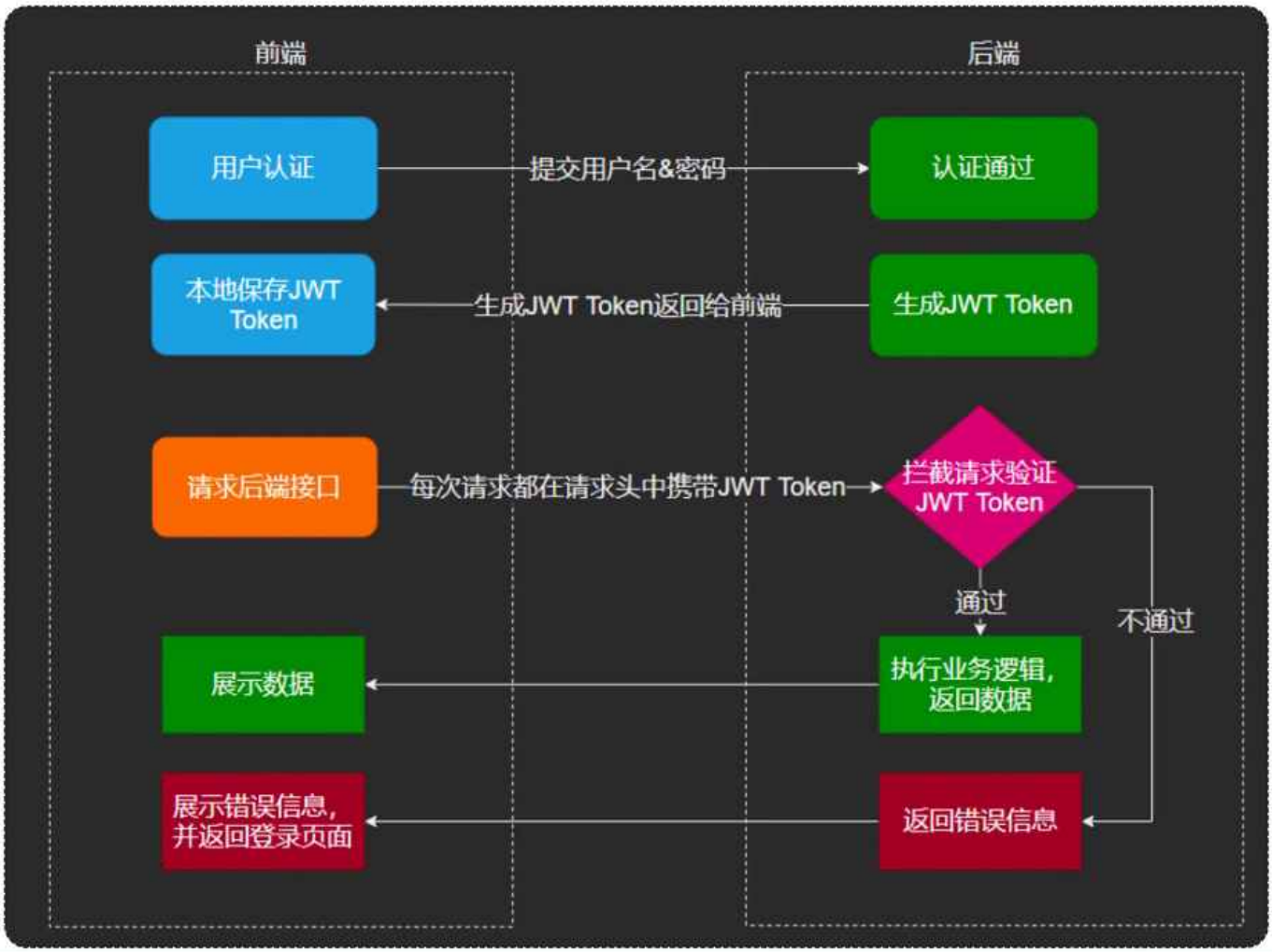
本地仓库

聚合与继承的区别

- 作用
  - 聚合用于快速构建项目
  - 继承用于快速配置
- 相同点：
  - 聚合与继承的pom.xml文件打包方式均为pom，可以将两种关系制作到同一个pom文件中
  - 聚合与继承均属于设计型模块，并无实际的模块内容
- 不同点：
  - 聚合是在当前模块中配置关系，聚合可以感知到参与聚合的模块有哪些
  - 继承是在子模块中配置关系，父模块无法感知哪些子模块继承了自己

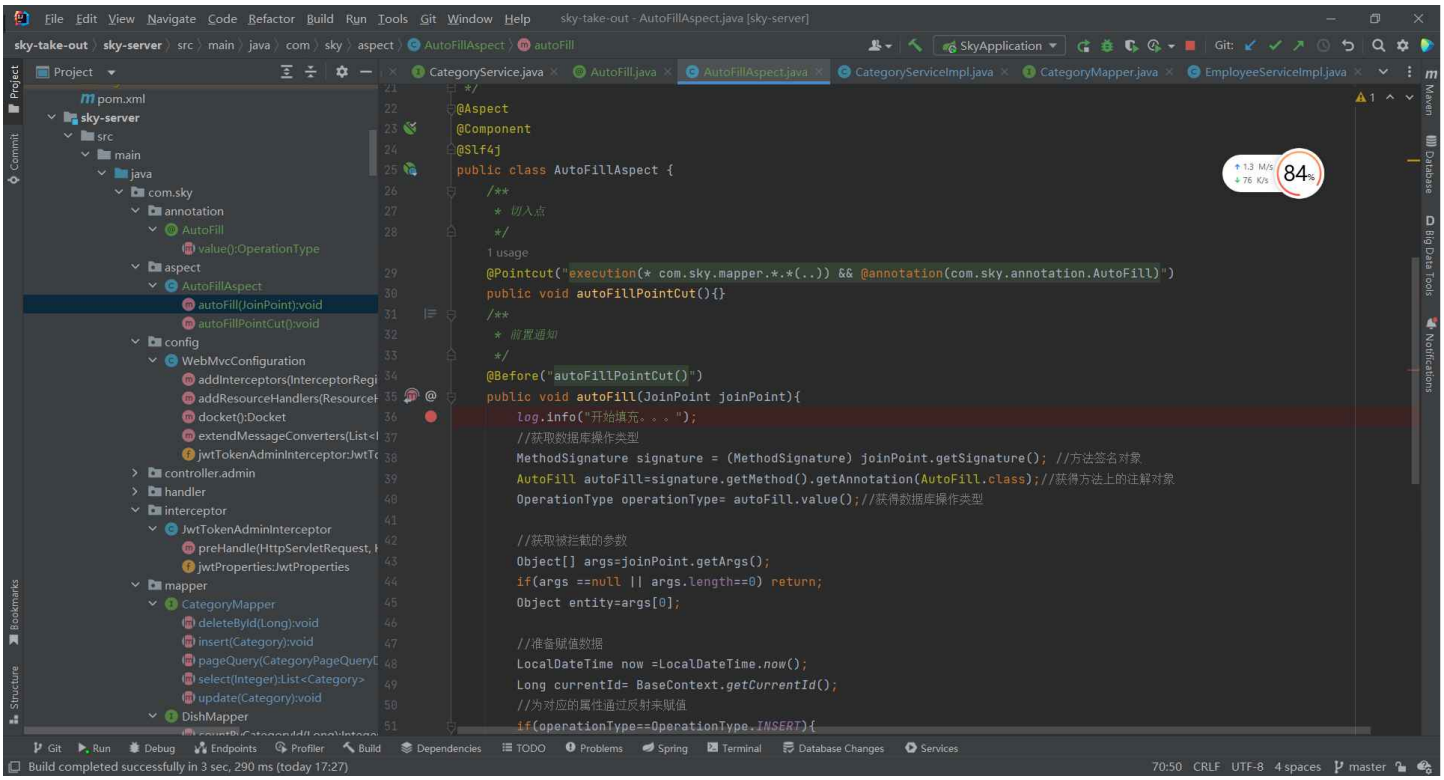
标准数据层CRUD功能

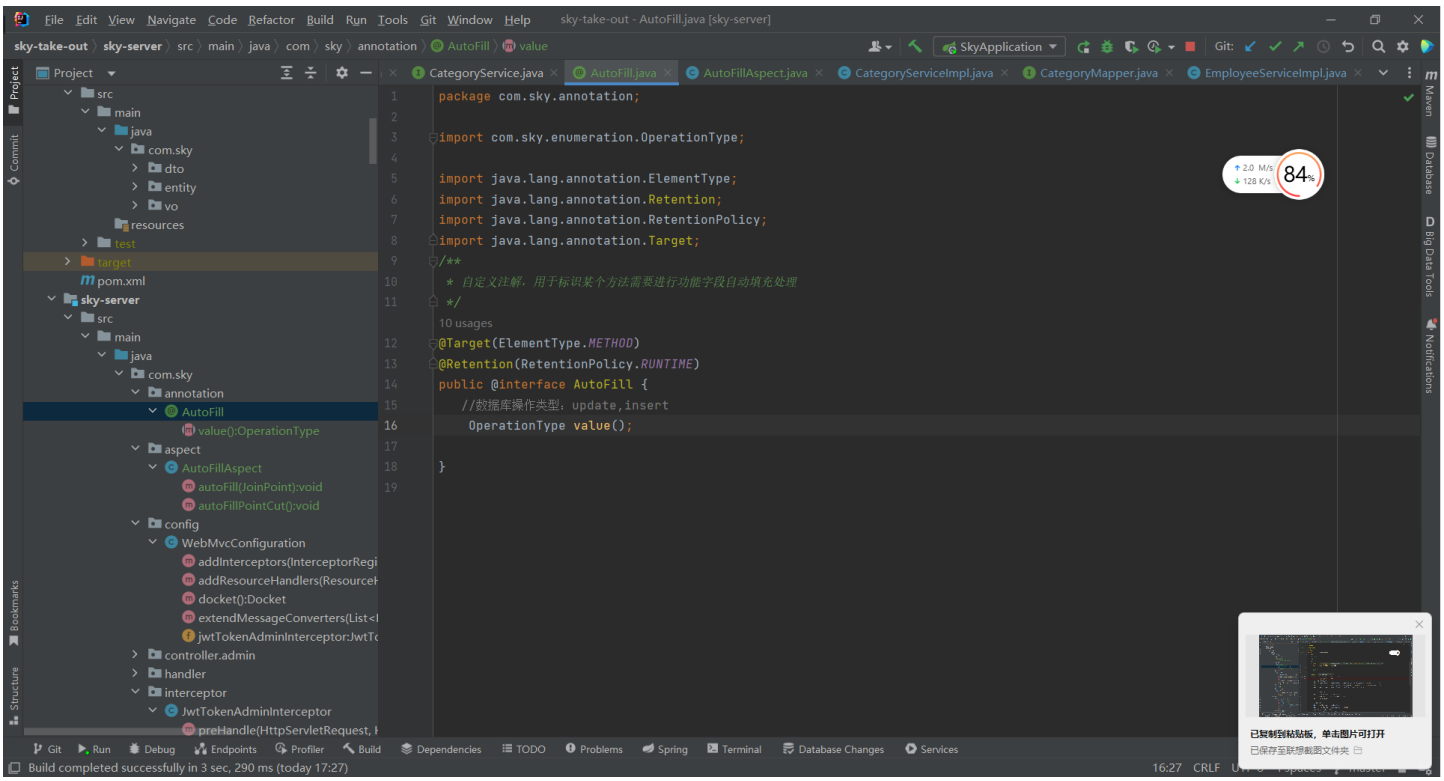
功能	自定义接口	MP接口
新增	`boolean save(T t)`	`int insert(T t)`
删除	`boolean delete(int id)`	`int deleteById(Serializable id)`
修改	`boolean update(T t)`	`int updateById(T t)`
根据id查询	`T getById(int id)`	`T selectById(Serializable id)`
查询全部	`List<T> getAll()`	`List<T> selectList()`
分页查询	`PageInfo<T> getAll(int page, int size)`	`IPage<T> selectPage(IPage<T> page)`
按条件查询	`List<T> getAll(Condition condition)`	`IPage<T> selectPage(Wrapper<T> queryWrapper)`

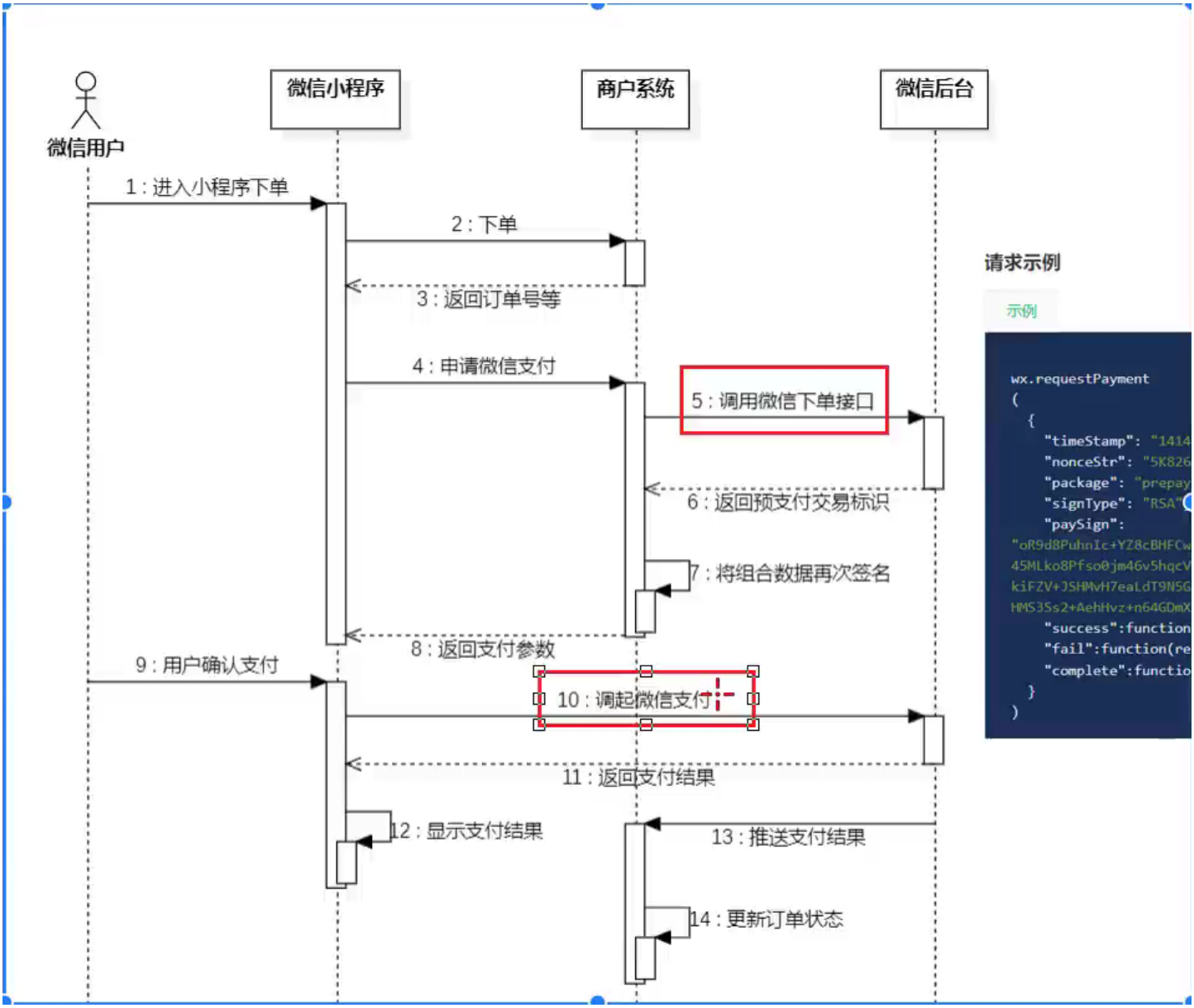


登录Token验证

自动填充，aop


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
sky-take-out - AutoFillAspect.java [sky-server]
sky-take-out › sky-server › src › main › java › com › sky › aspect › AutoFillAspect › autoFill
SkyApplication
Git:
Project
pom.xml
sky-server
src
main
java
com.sky
annotation
AutoFill
value():OperationType
aspect
AutoFillAspect
autoFill(JoinPoint):void
autoFillPointCut():void
config
WebMvcConfiguration
addInterceptors(InterceptorRegi
addResourceHandlers(Resourcef
docket():Docket
extendMessageConverters(List<
jwtTokenAdminInterceptor:JwtTo
controller.admin
handler
interceptor
JwtTokenAdminInterceptor
preHandle(HttpServletRequest,
jwtProperties:JwtProperties
mapper
CategoryMapper
deleteById(Long):void
insert(Category):void
pageQuery(CategoryPageQueryD
select(Integer):List<Category>
update(Category):void
DishMapper
CategoryService.java
AutoFill.java
AutoFillAspect.java
CategoryServiceImpl.java
CategoryMapper.java
EmployeeServiceImpl.java
*/
@Aspect
@Component
@Slf4j
public class AutoFillAspect {
/**
* 切入点
*/
1 usage
@Pointcut("execution(* com.sky.mapper.*.*(..)) && @annotation(com.sky.annotation.AutoFill)")
public void autoFillPointCut(){}
/**
* 前置通知
*/
@Before("autoFillPointCut()")
public void autoFill(JoinPoint joinPoint){
log.info("开始填充。。。");
//获取数据库操作类型
MethodSignature signature = (MethodSignature) joinPoint.getSignature(); //方法签名对象
AutoFill autoFill=signature.getMethod().getAnnotation(AutoFill.class);//获得方法上的注解对象
OperationType operationType= autoFill.value();//获得数据库操作类型
//获取被拦截的参数
Object[] args=joinPoint.getArgs();
if(args ==null || args.length==0) return;
Object entity=args[0];
//准备赋值数据
LocalDateTime now =LocalDateTime.now();
Long currentId= BaseContext.getCurrentId();
//为对应的属性通过反射来赋值
if(operationType==OperationType.INSERT){
84%
Git Run Debug Endpoints Profiler Build Dependencies TODO Problems Spring Terminal Database Changes Services
Build completed successfully in 3 sec, 290 ms (today 17:27)
70:50 CRLF UTF-8 4 spaces master


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
sky-take-out - AutoFill.java [sky-server]
sky-take-out › sky-server › src › main › java › com › sky › annotation › AutoFill › value
SkyApplication
Git:
Project
src
main
java
com.sky
dto
entity
vo
resources
test
target
pom.xml
sky-server
annotation
AutoFill
value():OperationType
aspect
AutoFillAspect
autoFill(JoinPoint):void
autoFillPointCut():void
config
WebMvcConfiguration
controller.admin
handler
interceptor
JwtTokenAdminInterceptor
CategoryService.java
AutoFill.java
AutoFillAspect.java
CategoryServiceImpl.java
CategoryMapper.java
EmployeeServiceImpl.java
package com.sky.annotation;
import com.sky.enumeration.OperationType;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
* 自定义注解，用于标识某个方法需要进行功能字段自动填充处理
*/
10 usages
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AutoFill {
//数据库操作类型，update,insert
OperationType value();
}
84%
已复制到粘贴板，单击图片可打开
已保存至联想截图文件夹
Git Run Debug Endpoints Profiler Build Dependencies TODO Problems Spring Terminal Database Changes Services
Build completed successfully in 3 sec, 290 ms (today 17:27)
16:27 CRLF


微信用户
微信小程序
商户系统
微信后台
1 : 进入小程序下单
2 : 下单
3 : 返回订单号等
4 : 申请微信支付
5 : 调用微信下单接口
6 : 返回预支付交易标识
7 : 将组合数据再次签名
8 : 返回支付参数
9 : 用户确认支付
10 : 调起微信支付
11 : 返回支付结果
12 : 显示支付结果
13 : 推送支付结果
14 : 更新订单状态
请求示例
示例
wx.requestPayment
(
{
"timeStamp": "1414
"nonceStr": "5K826
"package": "prepay
"signType": "RSA"
"paySign":
"oR9d8PuhnIc+YZ8cBHFCw
45MLko8Pfso0jm46v5hqcV
kiFZV+JSHMvH7eaLdT9N5G
HMS3Ss2+AehHvz+n64GDmX
"success":function
"fail":function(re
"complete":functio
}
)