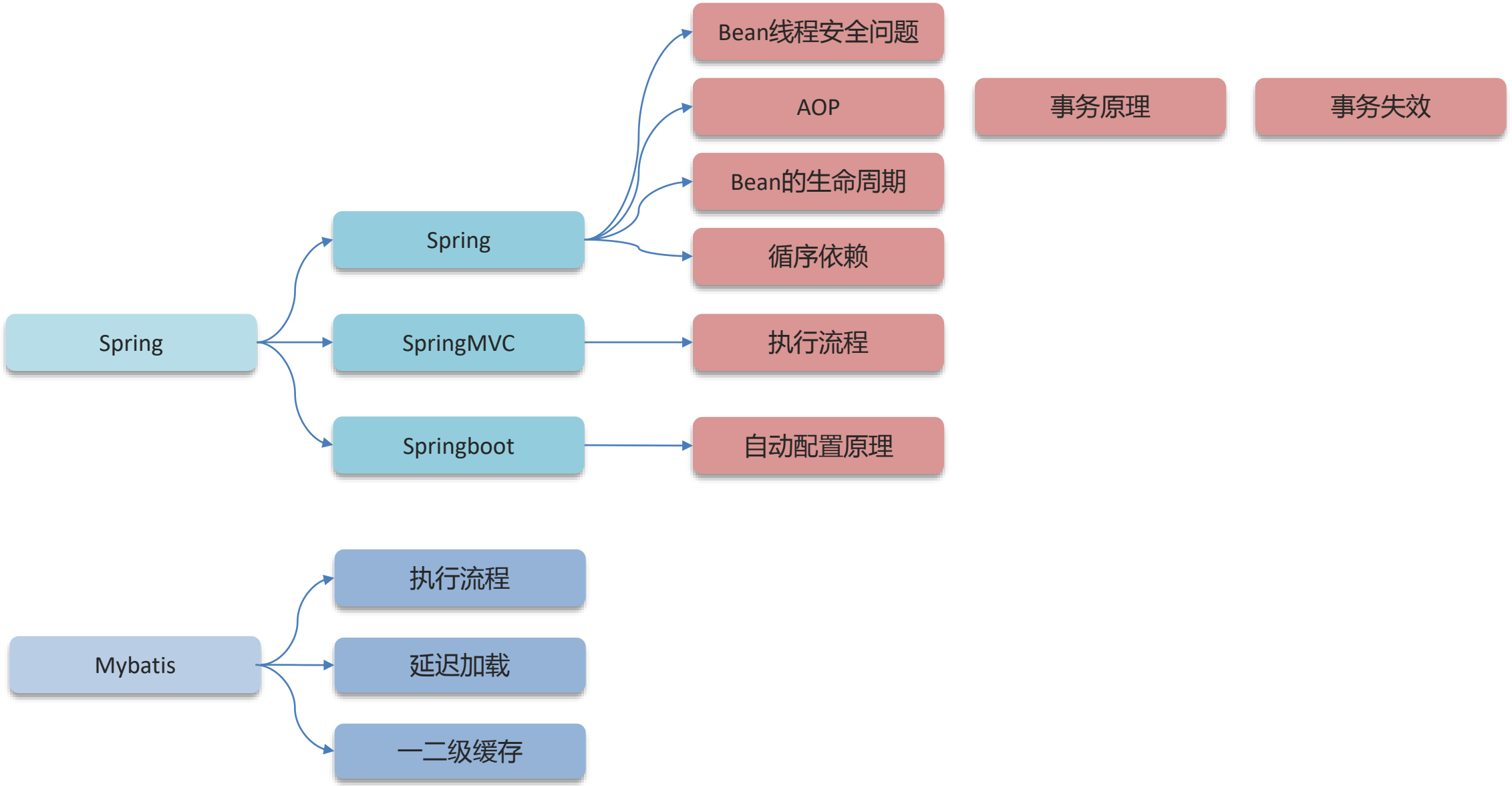
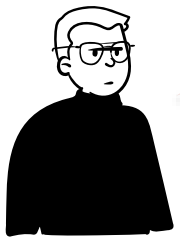
框架篇

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

多一句没有，少一句不行，用更短时间，教会更实用的技术！





Spring框架中的单例bean是线程安全的吗？

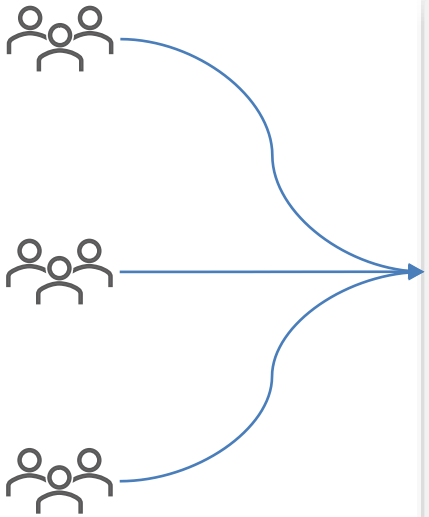
Spring框架中的bean是单例的吗？

```
@Service
@Scope("singleton")
public class UserServiceImpl implements UserService {


}
```

- singleton : bean在每个Spring IOC容器中只有一个实例。
- prototype：一个bean的定义可以有多个实例。

不是线程安全的

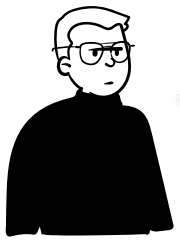
Spring框架中的单例bean是线程安全的吗？



```
@Controller
@RequestMapping("/user")
public class UserController {

    private int count;    成员方法需考虑线程安全

    @Autowired
    private UserService userService;

    @GetMapping("/getById/{id}")
    public User getById(@PathVariable("id") Integer id){
        count++;
        System.out.println(count);
        return userService.getById(id);
    }
}
```

服务器中的代码片段

Spring bean并没有可变的状态(比如Service类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。



Spring框架中的单例bean是线程安全的吗？

不是线程安全的

Spring框架中有一个@S

因为一般在spring的bea

量，是要考虑线程安全问

面试官：Spring框架中的单例bean是线程安全的吗？

候选人：

嗯！

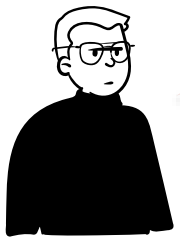
不是线程安全的，是这样的

当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求对应的业务逻辑（成员方法），如果该处理逻辑中有对该单列状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。

比如：我们通常在项目中使用的Spring bean都是不可可变的状态(比如Service类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。

如果你的bean有多种状态的话（比如 View Model对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用由“**singleton**”变更为“**prototype**”。



什么是AOP，你们项目中有没有使用到AOP

对AOP的理解

有没有真的用过aop

AOP称为面向切面编程，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。

常见的AOP使用场景：

- 记录操作日志
- 缓存处理
- Spring中内置的事务处理

什么是AOP,你们项目中有没有使用到AOP

记录操作日志思路

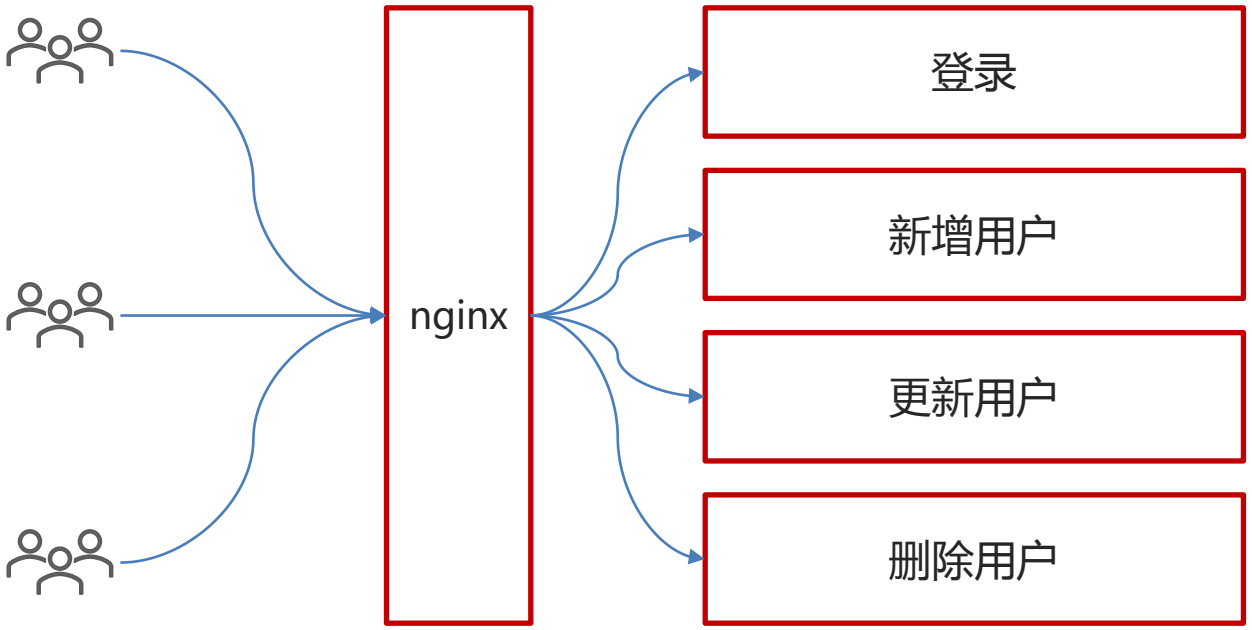
	用户名	请求方式	访问地址	模块名称	登录IP	操作时间	操作
	admin	GET	/monitor/operlog/export	操作日志	172.17.32.253	2023-02-06 15:29:19	查看
	admin	PUT	/clues/clue/gain	批量捞取	172.17.32.253	2023-02-06 14:55:50	查看
	admin	PUT	/clues/clue/false/9003	伪线索	172.17.32.253	2023-02-06 14:55:41	查看
	admin	PUT	/clues/clue/false/9002	伪线索	172.17.32.253	2023-02-06 14:55:32	查看
	admin	PUT	/clues/clue/false/9031	伪线索	172.17.32.253	2023-02-06 14:55:28	查看

获取请求的用户名、请求方式、访问地址、模块名称、登录ip、操作时间，记录到数据库的日志表中

什么是AOP,你们项目中有没有使用到AOP

记录操作日志思路

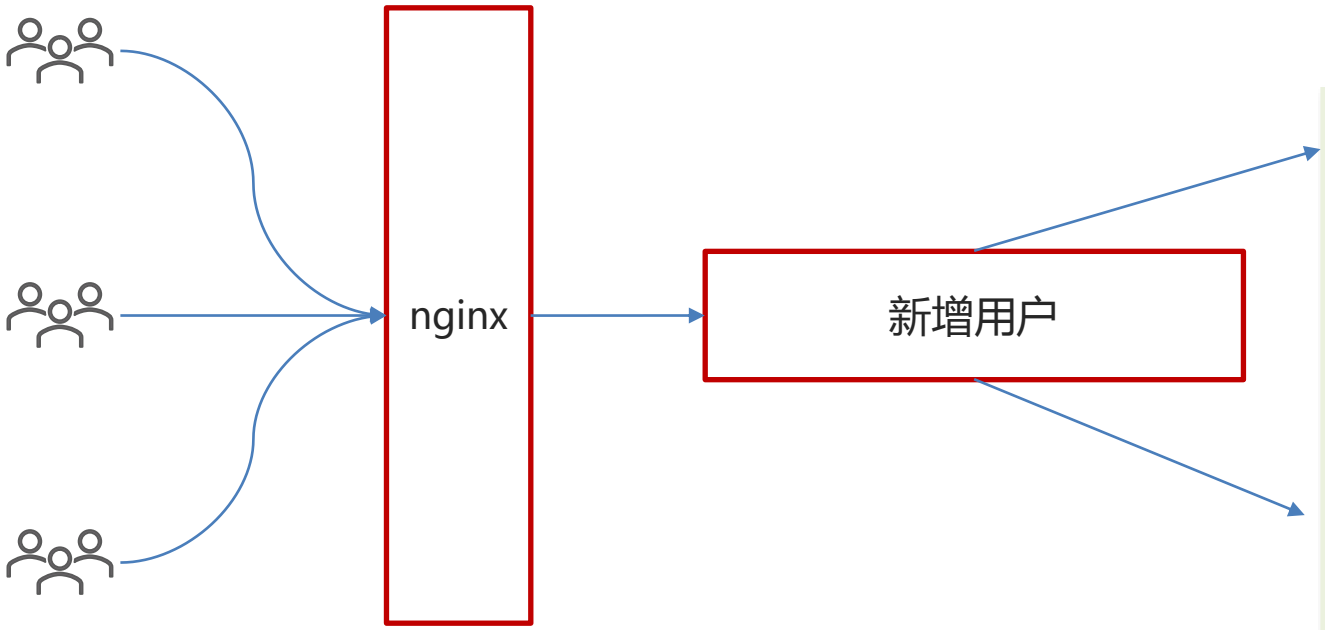
	用户名	请求方式	访问地址	模块名称	登录IP	操作时间	操作
	admin	GET	/monitor/operlog/export	操作日志	172.17.32.253	2023-02-06 15:29:19	查看
	admin	PUT	/clues/clue/gain	批量捞取	172.17.32.253	2023-02-06 14:55:50	查看
	admin	PUT	/clues/clue/false/9003	伪线索	172.17.32.253	2023-02-06 14:55:41	查看
	admin	PUT	/clues/clue/false/9002	伪线索	172.17.32.253	2023-02-06 14:55:32	查看
	admin	PUT	/clues/clue/false/9031	伪线索	172.17.32.253	2023-02-06 14:55:28	查看



什么是AOP,你们项目中有没有使用到AOP

记录操作日志思路

用户名	请求方式	访问地址	模块名称	登录IP	操作时间	操作
admin	GET	/monitor/operlog/export	操作日志	172.17.32.253	2023-02-06 15:29:19	查看
admin	PUT	/clues/clue/gain	批量捞取	172.17.32.253	2023-02-06 14:55:50	查看
admin	PUT	/clues/clue/false/9003	伪线索	172.17.32.253	2023-02-06 14:55:41	查看
admin	PUT	/clues/clue/false/9002	伪线索	172.17.32.253	2023-02-06 14:55:32	查看
admin	PUT	/clues/clue/false/9031	伪线索	172.17.32.253	2023-02-06 14:55:28	查看



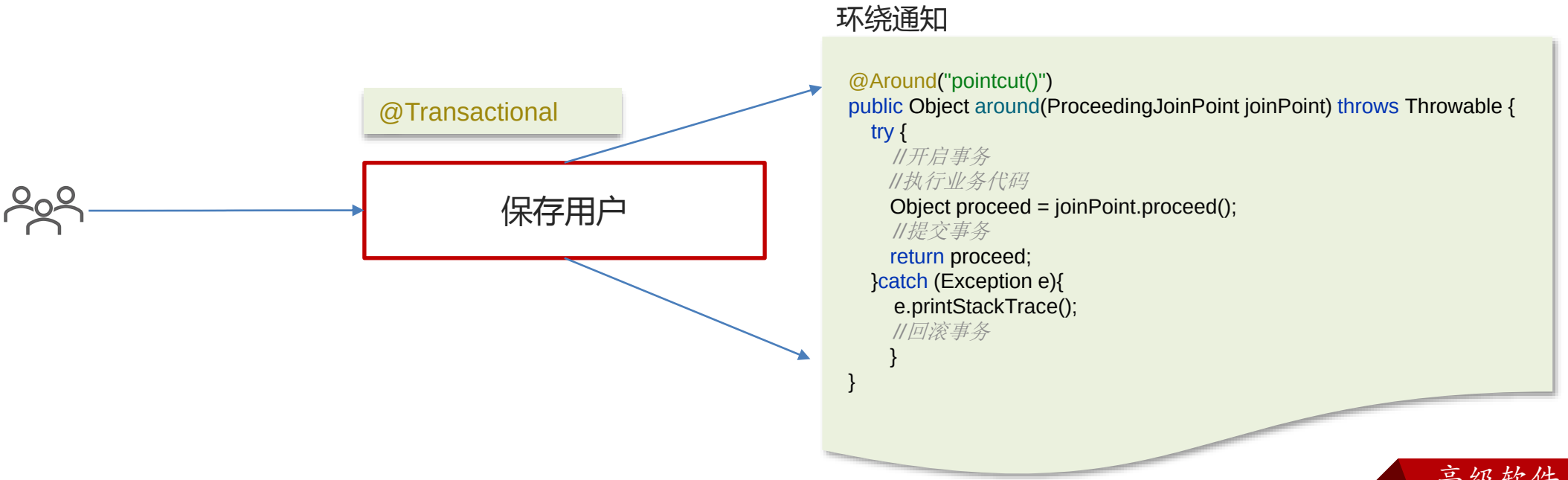
环绕通知

```
@Around("pointcut()")
public Object around(ProceedingJoinPoint joinPoint) {
        //获取用户名
        //获取请求方式
        //获取访问结果
        //获取模块名称
        //登录IP
        //操作时间

        //保存到数据库（操作日志）
        return null;
}
```

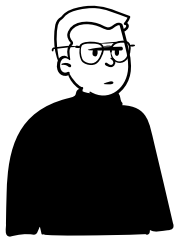
Spring中的事务是如何实现的

Spring支持编程式事务管理和声明式事务管理两种方式。

- 编程式事务控制：需使用TransactionTemplate来进行实现，对业务代码有侵入性，项目中很少使用
- 声明式事务管理：声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。



什么是AOP



面向切面编程，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取公共模块复用，降低耦合

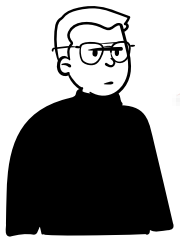
你们项目中有没有使用到AOP

记录操作日志，缓存，spring实现的事务

核心是：使用aop中的环绕通知+切点表达式（找到要记录日志的方法），通过环绕通知的参数获取请求方法的参数（类、方法、注解、请求方式等），获取到这些参数以后，保存到数据库

Spring中的事务是如何实现的

其本质是通过AOP功能，对方法前后进行拦截，在执行方法之前开启事务，在执行完目标方法之后根据执行情况提交或者回滚事务。



Spring中事务失效的场景有哪些

对spring框架的深入理解、复杂业务的编码经验

- 异常捕获处理
- 抛出检查异常
- 非public方法

Spring中事务失效的场景？

情况一：异常捕获处理

```
@Transactional
public void update(Integer from, Integer to, Double money) {
    try {
            //转账的用户不能为空
            Account fromAccount = accountDao.selectById(from);
           //判断用户的钱是否够转账
            if (fromAccount.getMoney() - money >= 0) {
           fromAccount.setMoney(fromAccount.getMoney() - money);
           accountDao.updateById(fromAccount);

           //异常
            int a = 1/0;

           //被转账的用户
            Account toAccount = accountDao.selectById(to);
           toAccount.setMoney(toAccount.getMoney() + money);
           accountDao.updateById(toAccount);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

原因

事务通知只有捉到了目标抛出的异常，才能进行后续的回滚处理，如果目标自己处理掉异常，事务通知无法知悉

解决

在catch块添加throw new RuntimeException(e)抛出

Spring中事务失效的场景？

情况二：抛出检查异常

```
@Transactional
public void update(Integer from, Integer to, Double money) throws FileNotFoundException {
      //转账的用户不能为空
      Account fromAccount = accountDao.selectById(from);
      //判断用户的钱是否够转账
      if (fromAccount.getMoney() - money >= 0) {
       fromAccount.setMoney(fromAccount.getMoney() - money);
       accountDao.updateById(fromAccount);
       //读取文件
      new FileInputStream("dddd");
       //被转账的用户
        Account toAccount = accountDao.selectById(to);
       toAccount.setMoney(toAccount.getMoney() + money);
       accountDao.updateById(toAccount);
   }
}
```

原因

Spring 默认只会回滚非检查异常

解决

配置rollbackFor属性

```
@Transactional(rollbackFor=Exception.class)
```

Spring中事务失效的场景？

情况三：非public方法导致的事务失效

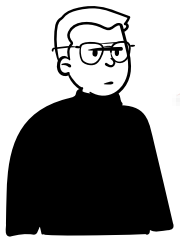
```
@Transactional(rollbackFor = Exception.class)
void update(Integer from, Integer to, Double money) throws FileNotFoundException {
        //转账的用户不能为空
        Account fromAccount = accountDao.selectById(from);
        //判断用户的钱是否够转账
        if (fromAccount.getMoney() - money >= 0) {
         fromAccount.setMoney(fromAccount.getMoney() - money);
         accountDao.updateById(fromAccount);

        //读取文件
        new FileInputStream("dddd");

        //被转账的用户
        Account toAccount = accountDao.selectById(to);
         toAccount.setMoney(toAccount.getMoney() + money);
         accountDao.updateById(toAccount);
    }
}
```

原因

Spring 为方法创建代理、添加事务通知、前提条件都是该方法是 public 的

解决

改为 public 方法

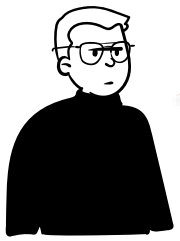


Spring中事务失效的场景有哪些

① 异常捕获处理，自己处理了异常，没有抛出，解决：手动抛出

② 抛出检查异常，配置rollbackFor属性为Exception

③ 非public方法导致的事务失效，改为public



Spring的bean的生命周期

Spring容器是如何管理和创建bean实例

方便调试和解决问题

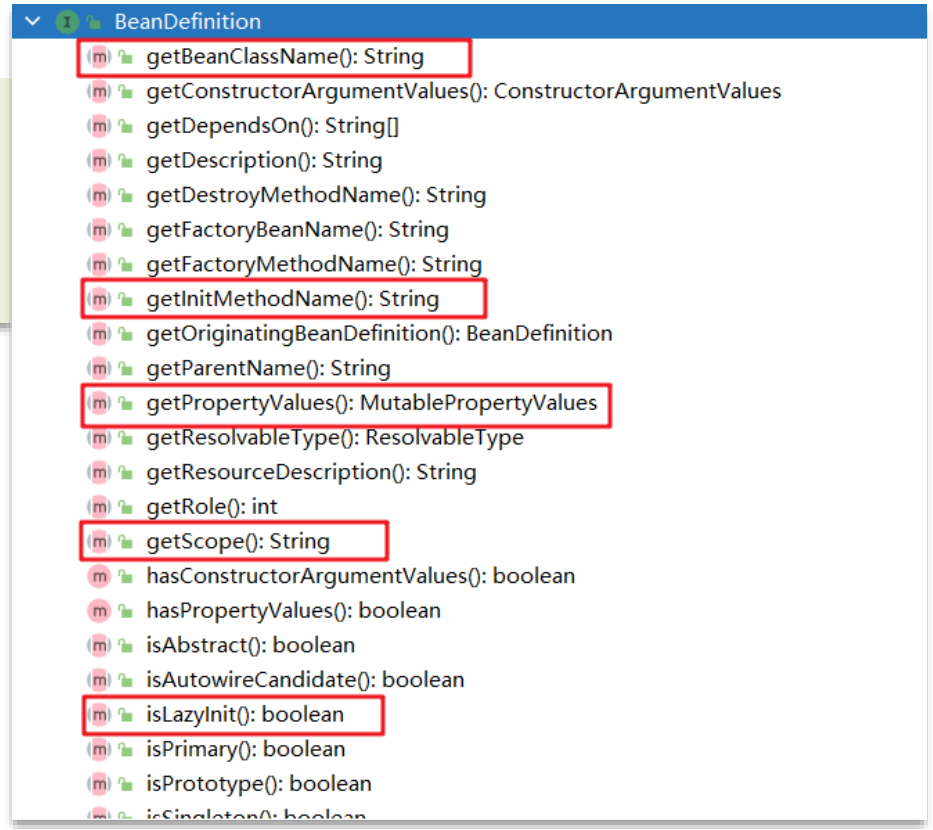
Bean的流程 → 代码验证

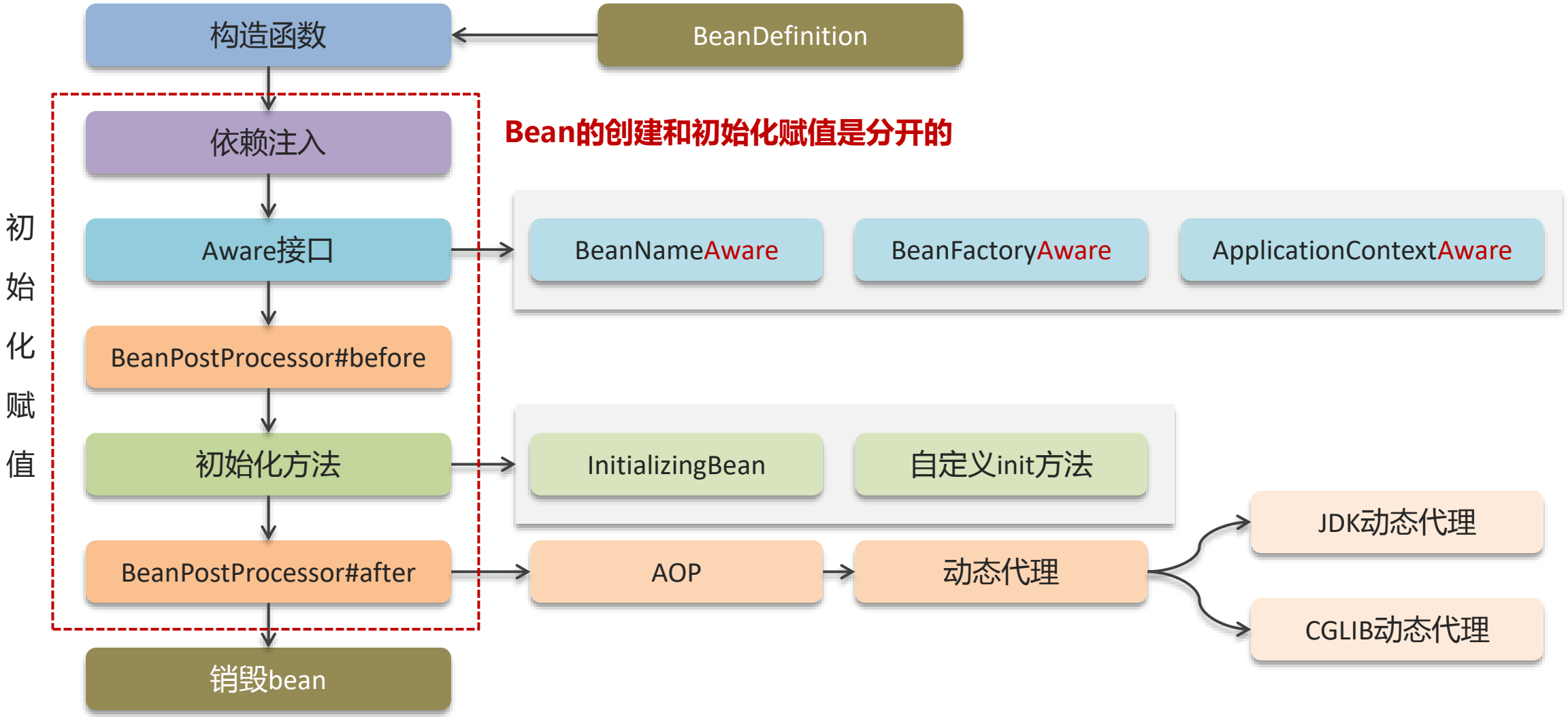
BeanDefinition

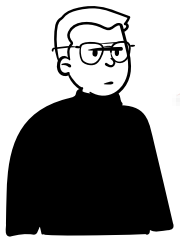
Spring容器在进行实例化时，会将xml配置的<bean>的信息封装成一个BeanDefinition对象，Spring根据BeanDefinition来创建Bean对象，里面有很多的属性用来描述Bean

```
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl" lazy-init="true"/>
<bean id="userService" class="com.itheima.service.UserServiceImpl" scope="singleton">
    <property name="userDao" ref="userDao"></property>
</bean>
```

- beanClassName：bean 的类名
- initMethodName：初始化方法名称
- properryValues：bean 的属性值
- scope：作用域
- lazyInit：延迟初始化

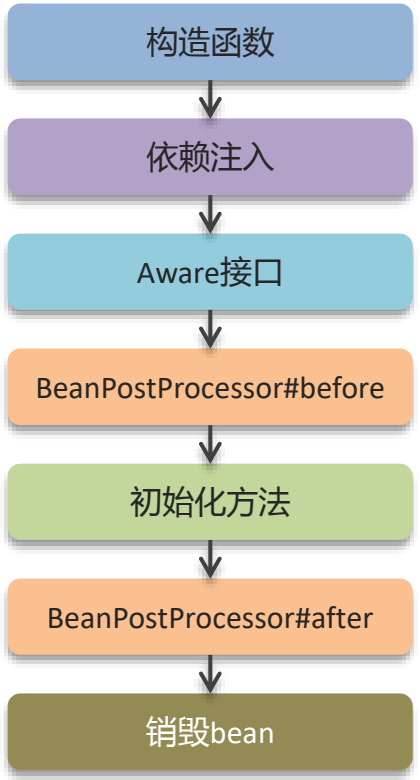


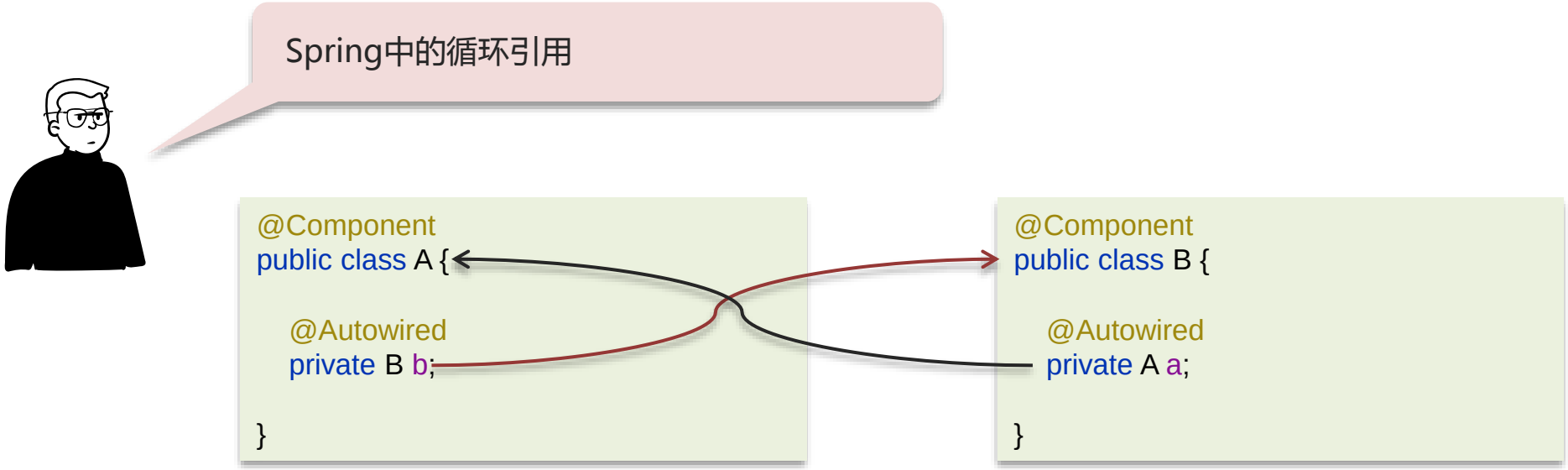

构造函数
BeanDefinition
依赖注入
Bean的创建和初始化赋值是分开的
初始化赋值
Aware接口
BeanNameAware
BeanFactoryAware
ApplicationContextAware
BeanPostProcessor#before
初始化方法
InitializingBean
自定义init方法
BeanPostProcessor#after
AOP
动态代理
JDK动态代理
CGLIB动态代理
销毁bean



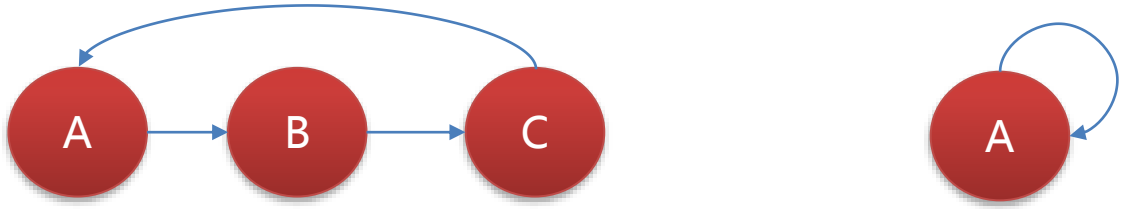
Spring的bean的生命周期

① 通过BeanDefinition获取bean的定义信息
② 调用构造函数实例化bean
③ bean的依赖注入
④ 处理Aware接口(BeanNameAware、BeanFactoryAware、ApplicationContextAware)
⑤ Bean的后置处理器BeanPostProcessor-前置
⑥ 初始化方法(InitializingBean、init-method)
⑦ Bean的后置处理器BeanPostProcessor-后置
⑧ 销毁bean

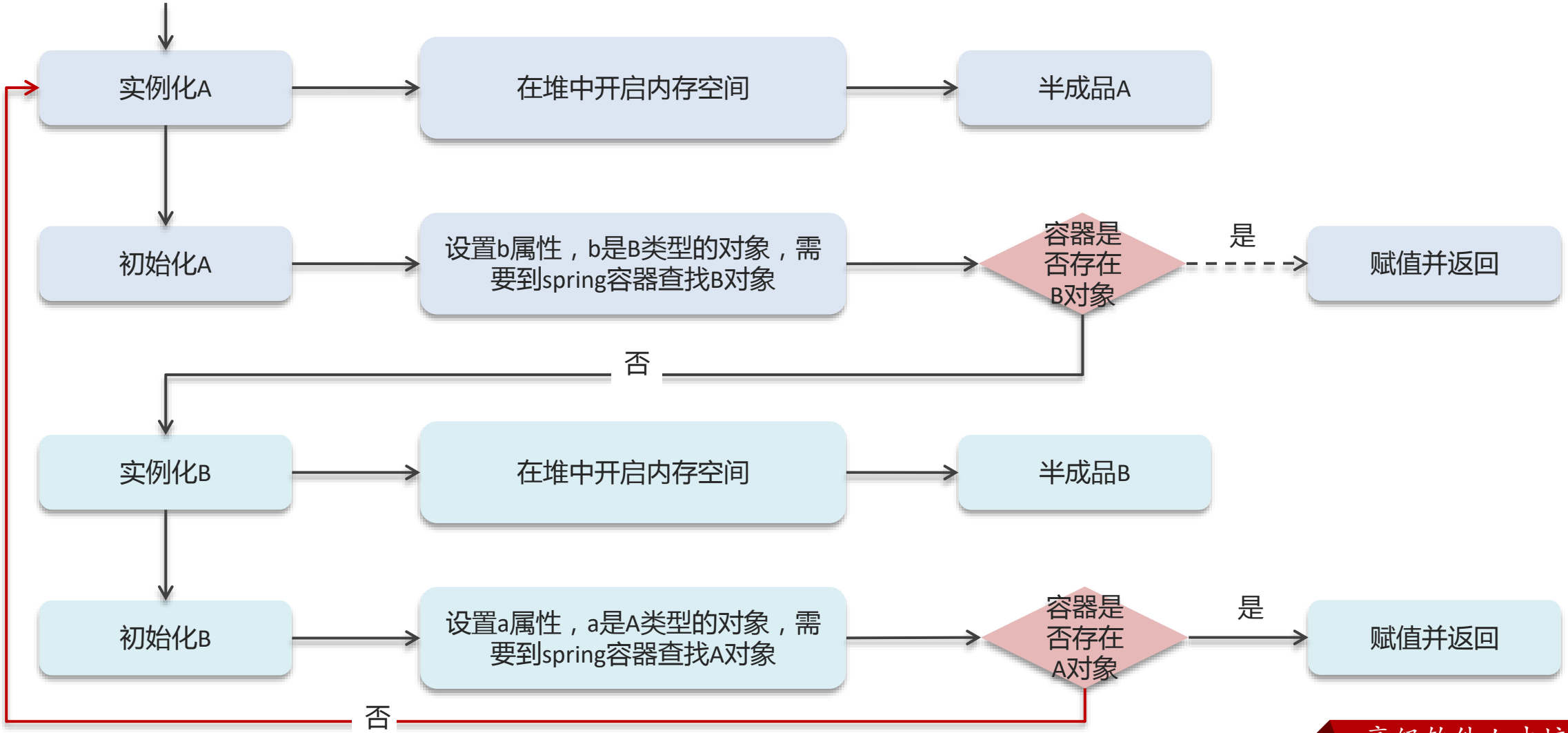




在创建A对象的同时需要使用的B对象，在创建B对象的同时需要使用到A对象



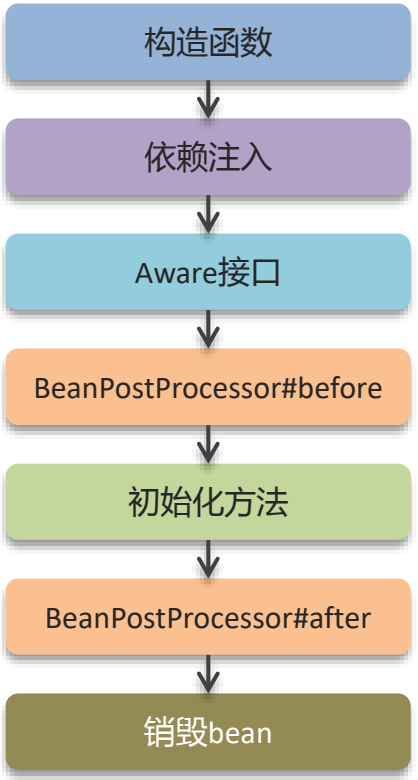
什么是Spring的循环依赖？



三级缓存解决循环依赖

Spring解决循环依赖是通过三级缓存，对应的三级缓存如下所示：

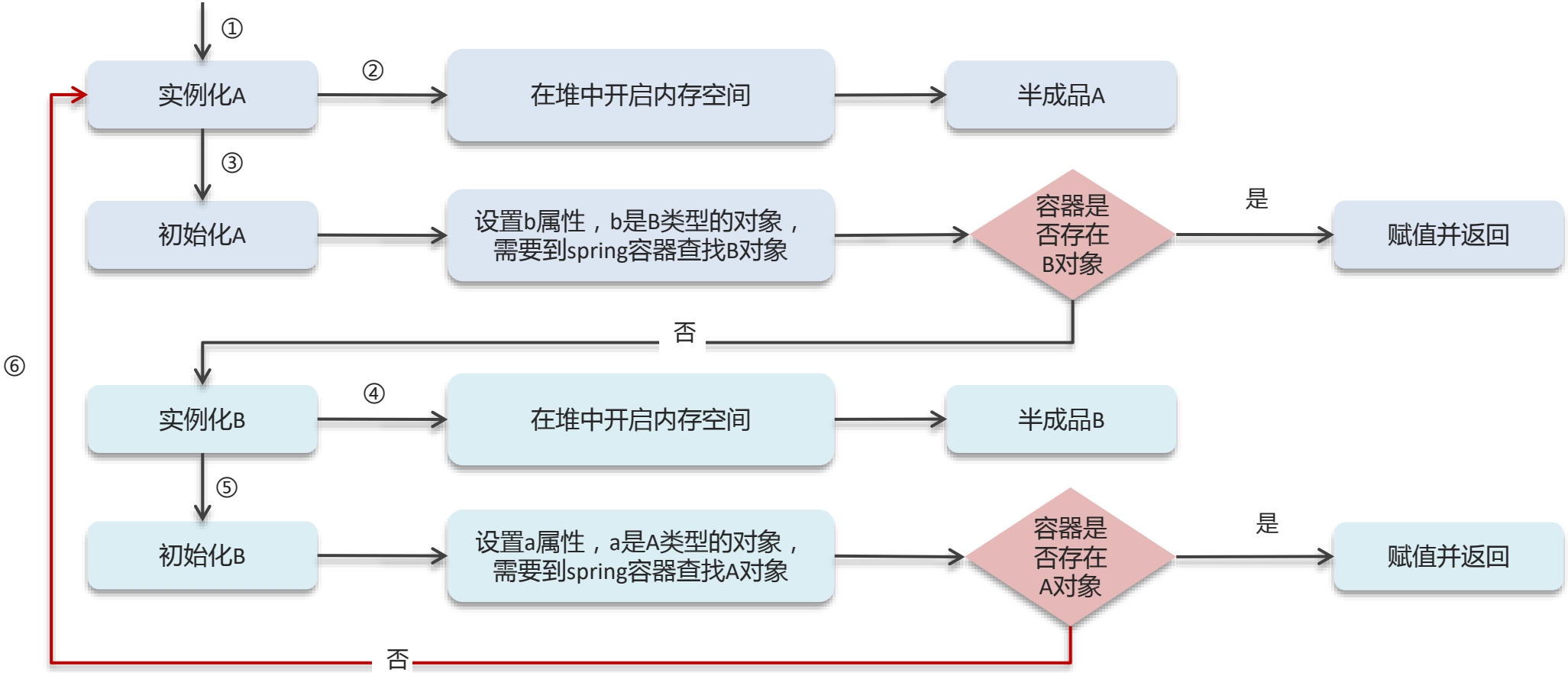
```
//单实例对象注册器
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {
    private static final int SUPPRESSED_EXCEPTIONS_LIMIT = 100;
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap(256);          一级缓存
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap(16);          三级缓存
    private final Map<String, Object> earlySingletonObjects = new ConcurrentHashMap(16);       二级缓存
}
```



缓存名称	源码名称	作用
一级缓存	singletonObjects	单例池，缓存已经经历了完整的生命周期，已经初始化完成的bean对象
二级缓存	earlySingletonObjects	缓存早期的bean对象（生命周期还没走完）
三级缓存	singletonFactories	缓存的是ObjectFactory，表示对象工厂，用来创建某个对象的

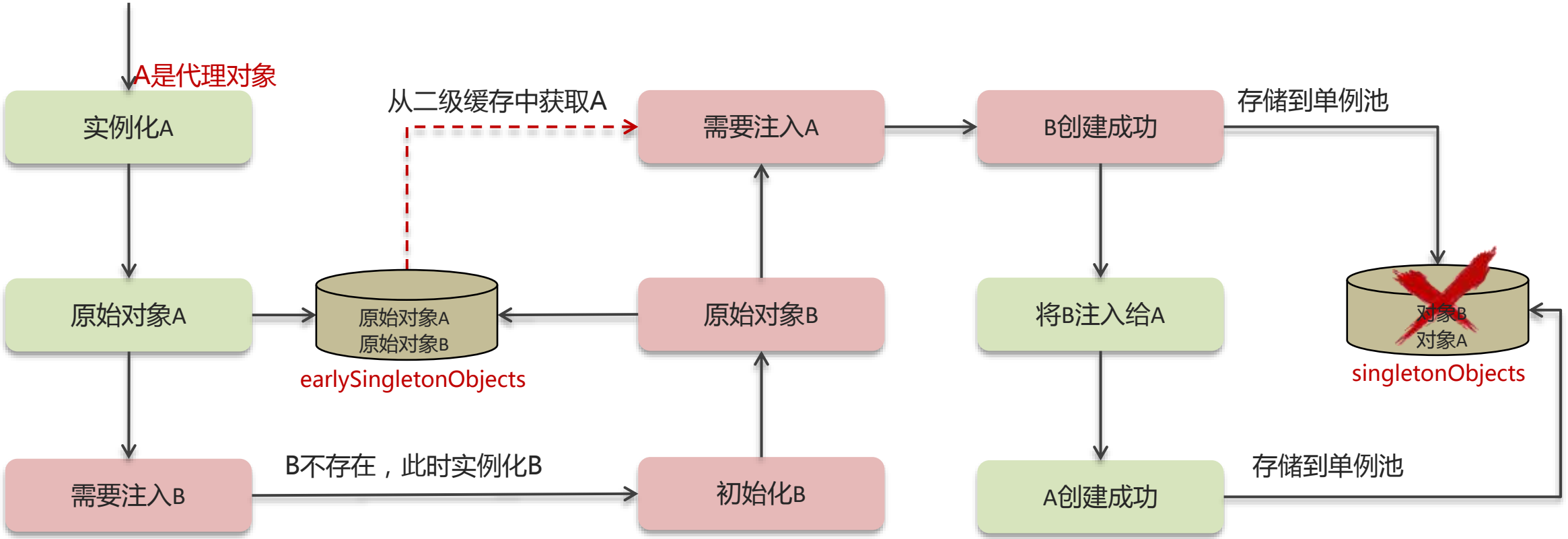
三级缓存解决循环依赖

一级缓存作用：限制bean在beanFactory中只存一份，即实现singleton scope，解决不了循环依赖

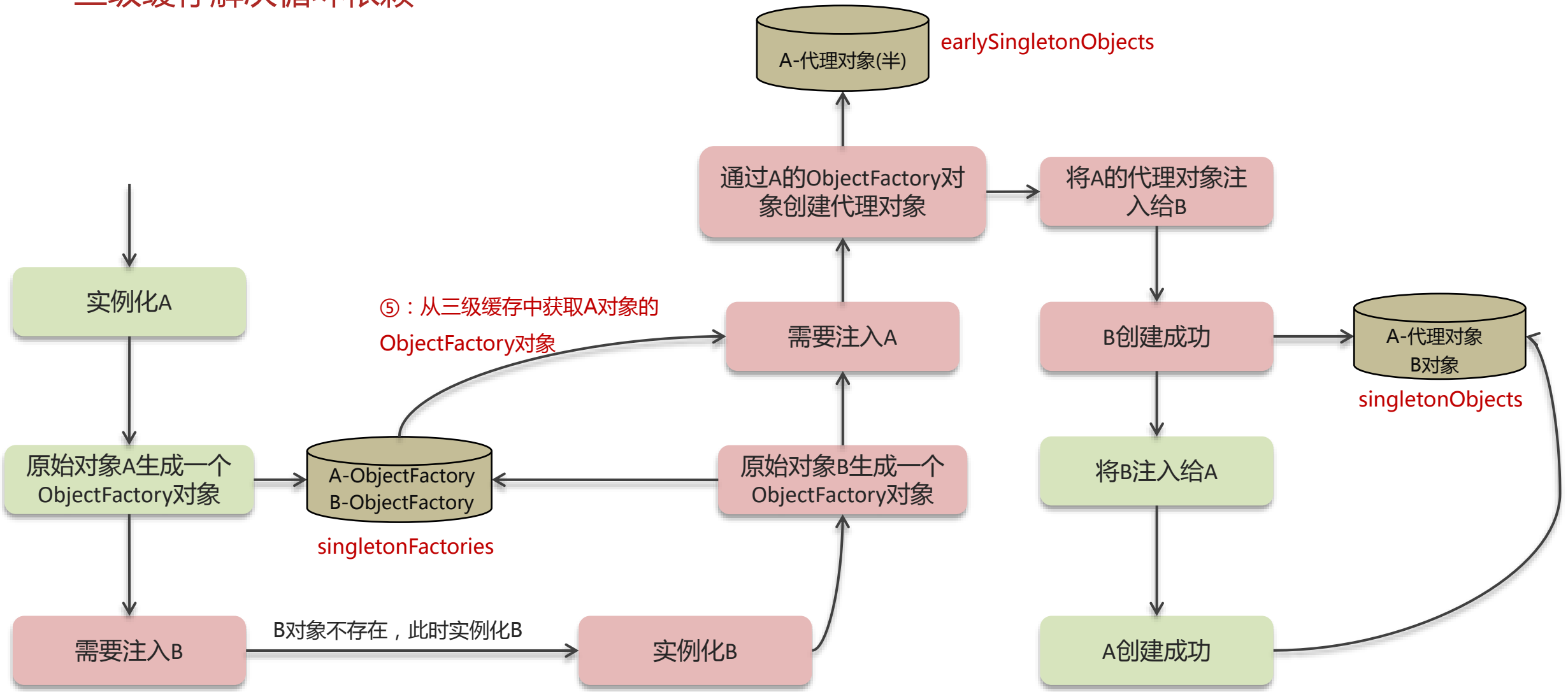


三级缓存解决循环依赖

如果要想打破循环依赖，就需要一个中间人的参与，这个中间人就是二级缓存。



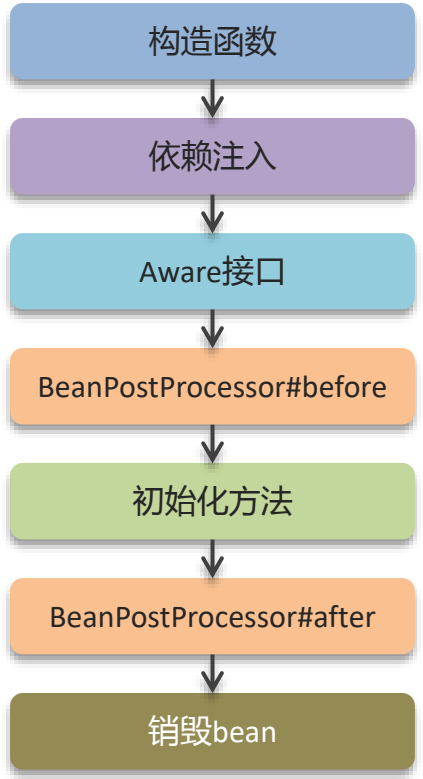
三级缓存解决循环依赖


A-代理对象(半)
earlySingletonObjects
通过A的ObjectFactory对象创建代理对象
将A的代理对象注入给B
实例化A
⑤：从三级缓存中获取A对象的ObjectFactory对象
需要注入A
B创建成功
A-代理对象
B对象
singletonObjects
原始对象A生成一个ObjectFactory对象
A-ObjectFactory
B-ObjectFactory
singletonFactories
原始对象B生成一个ObjectFactory对象
将B注入给A
需要注入B
B对象不存在，此时实例化B
实例化B
A创建成功

构造方法出现了循环依赖怎么解决？

```java
@Component
public class A {

    // B成员变量
    private B b;

    public A(B b){
        System.out.println("A的构造方法执行了...");
        this.b = b ;
    }
}
```

```java
@Component
public class B {

    // A成员变量
    private A a;

    public B(A a){
        System.out.println("B的构造方法执行了...");
        this.a = a ;
    }
}
```

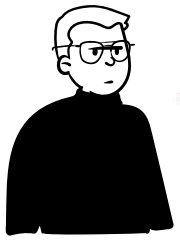


报错信息：

```
Is there an unresolvable circular reference?
```

解决：

```java
public A(@Lazy B b){
    System.out.println("A的构造方法执行了...");
    this.b = b ;
}
```



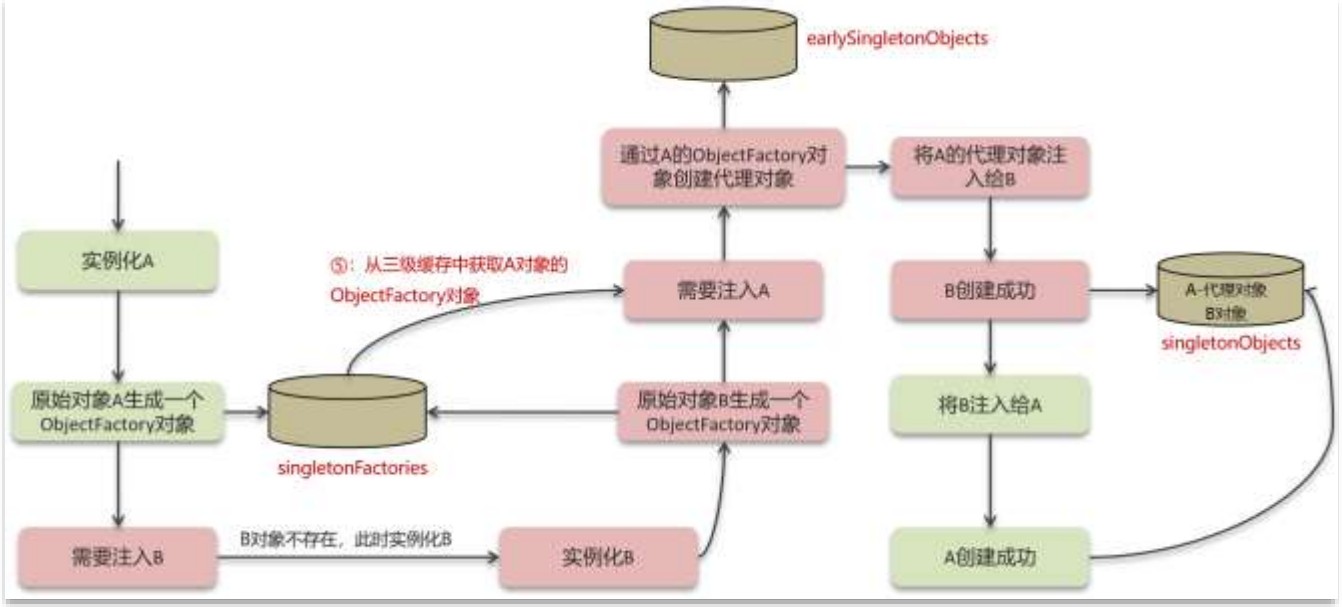
Spring中的循环引用

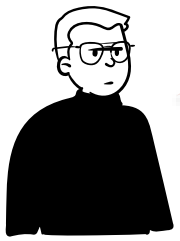
- 循环依赖：循环依赖其实就是循环引用,也就是两个或两个以上的bean互相持有对方,最终形成闭环。比如A依赖于B,B依赖于A
- 循环依赖在spring中是允许存在，spring框架依据三级缓存已经解决了大部分的循环依赖

① 一级缓存：单例池，缓存已经经历了完整的生命周期，已经初始化完成的bean对象

② 二级缓存：缓存早期的bean对象（生命周期还没走完）

③ 三级缓存：缓存的是ObjectFactory，表示对象工厂，用来创建某个对象的





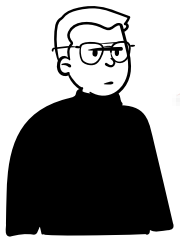
构造方法出现了循环依赖怎么解决？

A依赖于B，B依赖于A，注入的方式是构造函数

原因：由于bean的生命周期中构造函数是第一个执行的，spring框架并不能解决构造函数的的依赖注入

解决方案：使用@Lazy进行懒加载，什么时候需要对象再进行bean对象的创建

```
public A(@Lazy B b){
    System.out.println("A的构造方法执行了...");
    this.b = b ;
}
```

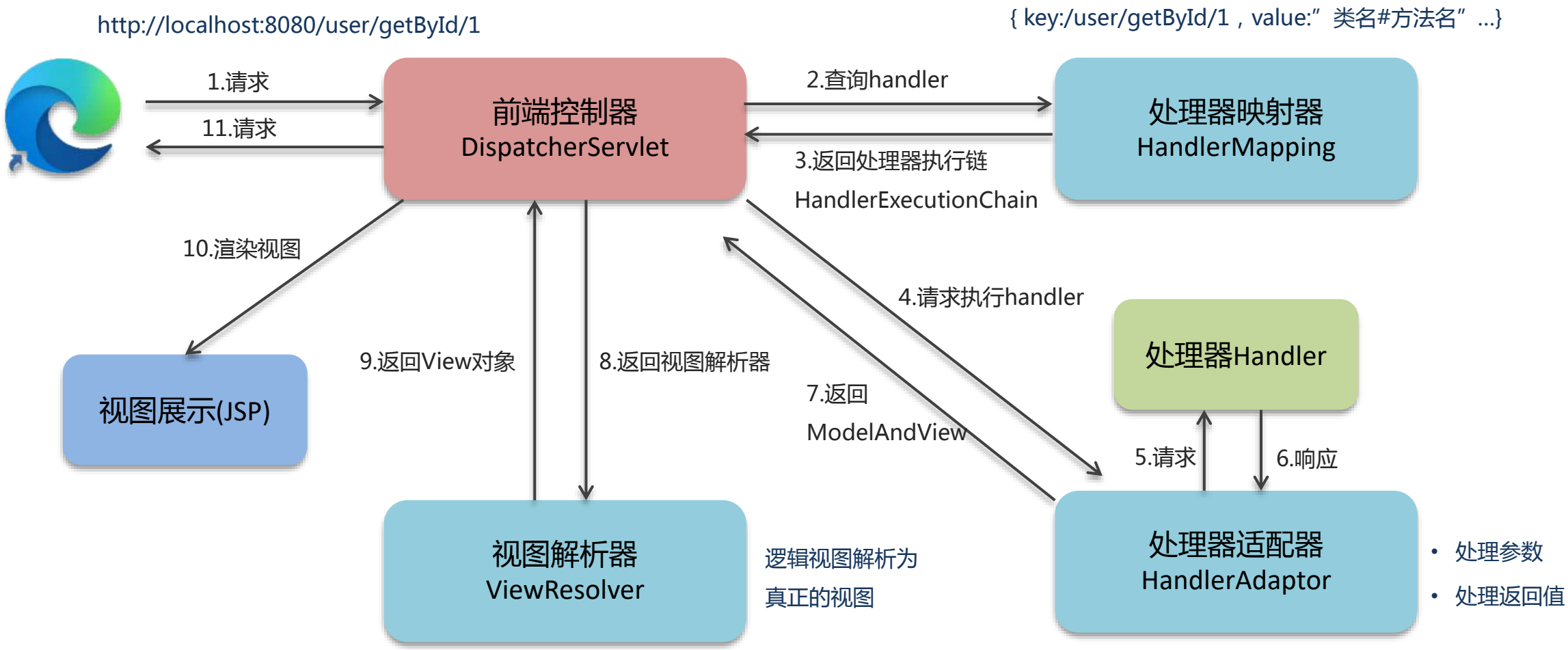


SpringMVC的执行流程知道嘛

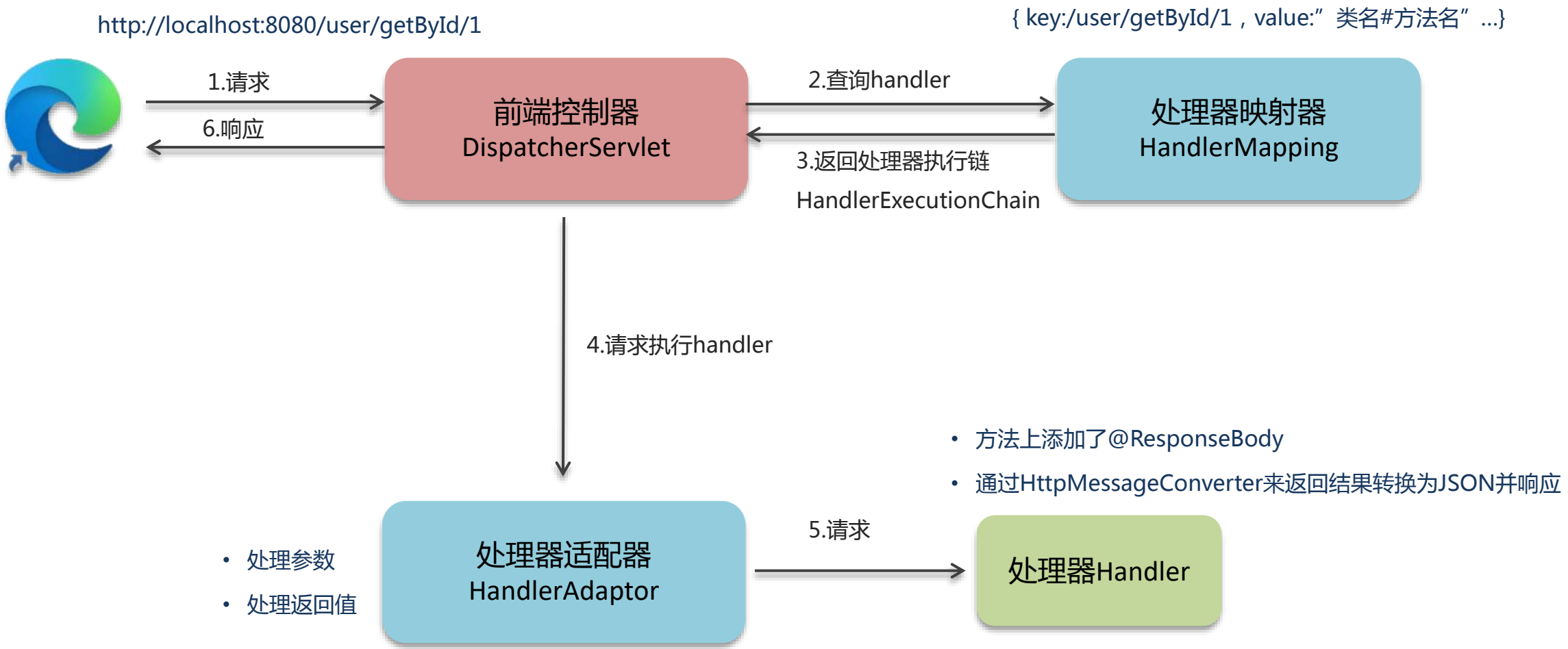
Springmvc的执行流程是这个框架最核心的内容

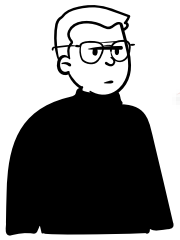
- 视图阶段（老旧JSP等）
- 前后端分离阶段（接口开发，异步）

视图阶段（JSP）


http://localhost:8080/user/getById/1
{ key:/user/getById/1，value:”类名#方法名”...}
1.请求
11.请求
前端控制器
DispatcherServlet
2.查询handler
3.返回处理器执行链
HandlerExecutionChain
处理器映射器
HandlerMapping
10.渲染视图
视图展示(JSP)
9.返回View对象
8.返回视图解析器
4.请求执行handler
7.返回
ModelAndView
处理器Handler
5.请求
6.响应
视图解析器
ViewResolver
逻辑视图解析为
真正的视图
处理器适配器
HandlerAdaptor
• 处理参数
• 处理返回值

前后端分离阶段（接口开发，异步请求）





SpringMVC的执行流程知道嘛

（版本1：视图版本，jsp）

① 用户发送出请求到前端控制器DispatcherServlet

② DispatcherServlet收到请求调用HandlerMapping（处理器映射器）

③ HandlerMapping找到具体的处理器，生成处理器对象及处理器拦截器(如果有)，再一起返回给DispatcherServlet。

④ DispatcherServlet调用HandlerAdapter（处理器适配器）

⑤ HandlerAdapter经过适配调用具体的处理器（Handler/Controller）

⑥ Controller执行完成返回ModelAndView对象

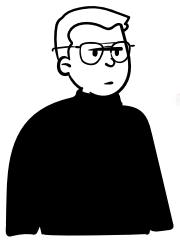
⑦ HandlerAdapter将Controller执行结果ModelAndView返回给DispatcherServlet

⑧ DispatcherServlet将ModelAndView传给ViewReslover（视图解析器）

⑨ ViewReslover解析后返回具体View（视图）

⑩ DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）

⑪ DispatcherServlet响应用户



SpringMVC的执行流程知道嘛

（版本2：前后端开发，接口开发）

① 用户发送出请求到前端控制器DispatcherServlet

② DispatcherServlet收到请求调用HandlerMapping（处理器映射器）

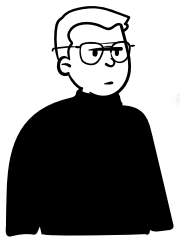
③ HandlerMapping找到具体的处理器，生成处理器对象及处理器拦截器(如果有)，再一起返回给DispatcherServlet。

④ DispatcherServlet调用HandlerAdapter（处理器适配器）

⑤ HandlerAdapter经过适配调用具体的处理器（Handler/Controller）

⑥ 方法上添加了@ResponseBody

⑦ 通过HttpMessageConverter来返回结果转换为JSON并响应



Springboot自动配置原理

Springboot中最高频的一道面试题，也是框架最核心的思想

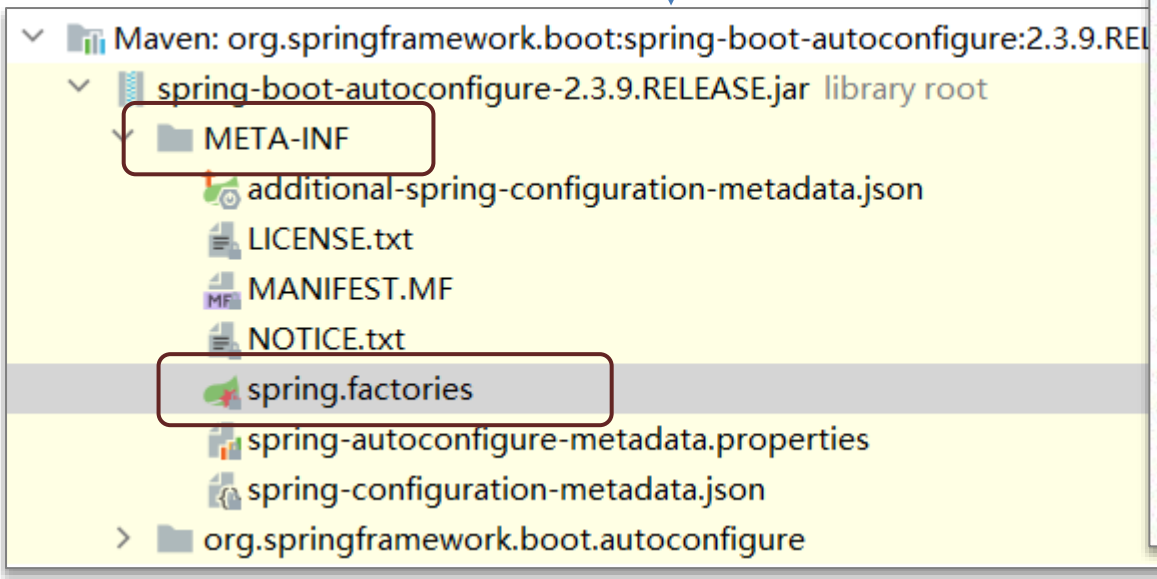
```java
@SpringBootApplication
public class UserApplication {

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

```java
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
```

- @SpringBootConfiguration：该注解与 @Configuration 注解作用相同，用来声明当前也是一个配置类。
- @ComponentScan：组件扫描，默认扫描当前引导类所在包及其子包。
- @EnableAutoConfiguration：SpringBoot实现自动化配置的核心注解。

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
```

```java
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
```



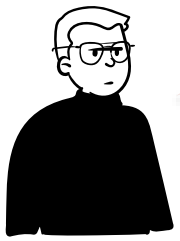

```java
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnClass({RedisOperations.class})
@EnableConfigurationProperties({RedisProperties.class})
@Import({LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class})
public class RedisAutoConfiguration {
    public RedisAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean(
        name = {"redisTemplate"}
    )
    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory redisCo
        RedisTemplate<Object, Object> template = new RedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
```

是一个配置类

判断是否有对应字节码

判断环境中没有对应的bean

Springboot自动配置原理



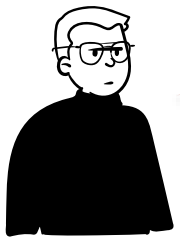
1，在Spring Boot项目中的引导类上有一个注解@SpringBootApplication，这个注解是对三个注解进行了封装，分别是：

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

2，其中@EnableAutoConfiguration是实现自动化配置的核心注解。 该注解通过@Import注解导入对应的配置选择器。

内部就是读取了该项目和该项目引用的Jar包的的classpath路径下META-INF/spring.factories文件中的所配置的类的全类名。 在这些配置类中所定义的Bean会根据条件注解所指定的条件来决定是否需要将其导入到Spring容器中。

3, 条件判断会有像@ConditionalOnClass这样的注解，判断是否有对应的class文件，如果有则加载该类，把这个配置类的所有的Bean放入spring容器中使用。



Spring框架常见注解（Spring、Springboot、Springmvc）

- Spring 的常见注解有哪些？
- SpringMVC常见的注解有哪些？
- Springboot常见注解有哪些？

Spring 的常见注解有哪些？

注解	说明
@Component、@Controller、@Service、@Repository	使用在类上用于实例化Bean
@Autowired	使用在字段上用于根据类型依赖注入
@Qualifier	结合@Autowired一起使用用于根据名称进行依赖注入
@Scope	标注Bean的作用范围
@Configuration	指定当前类是一个 Spring 配置类，当创建容器时会从该类上加载注解
@ComponentScan	用于指定 Spring 在初始化容器时要扫描的包
@Bean	使用在方法上，标注将该方法的返回值存储到Spring容器中
@Import	使用@Import导入的类会被Spring加载到IOC容器中
@Aspect、@Before、@After、@Around、@Pointcut	用于切面编程（AOP）

SpringMVC常见的注解有哪些？

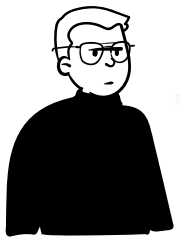
注解	说明
@RequestMapping	用于映射请求路径，可以定义在类上和方法上。用于类上，则表示类中的所有的方法都是以该地址作为父路径
@RequestBody	注解实现接收http请求的json数据，将json转换为java对象
@RequestParam	指定请求参数的名称
@PathViriable	从请求路径下中获取请求参数(/user/{id})，传递给方法的形式参数
@ResponseBody	注解实现将controller方法返回对象转化为json对象响应给客户端
@RequestHeader	获取指定的请求头数据
@RestController	@Controller + @ResponseBody

Springboot常见注解有哪些？

注解
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan

注解	说明
@Component、@Controller、@Service、@Repository	使用在类上用于实例化Bean
@Autowired	使用在字段上用于根据类型依赖注入
@Qualifier	结合@Autowired一起使用用于根据名称进行依赖注入
@Scope	标注Bean的作用范围
@Configuration	
@ComponentScan	
@Bean	
@Import	
@Aspect、@Before、@After、@Around、@Pointcut	

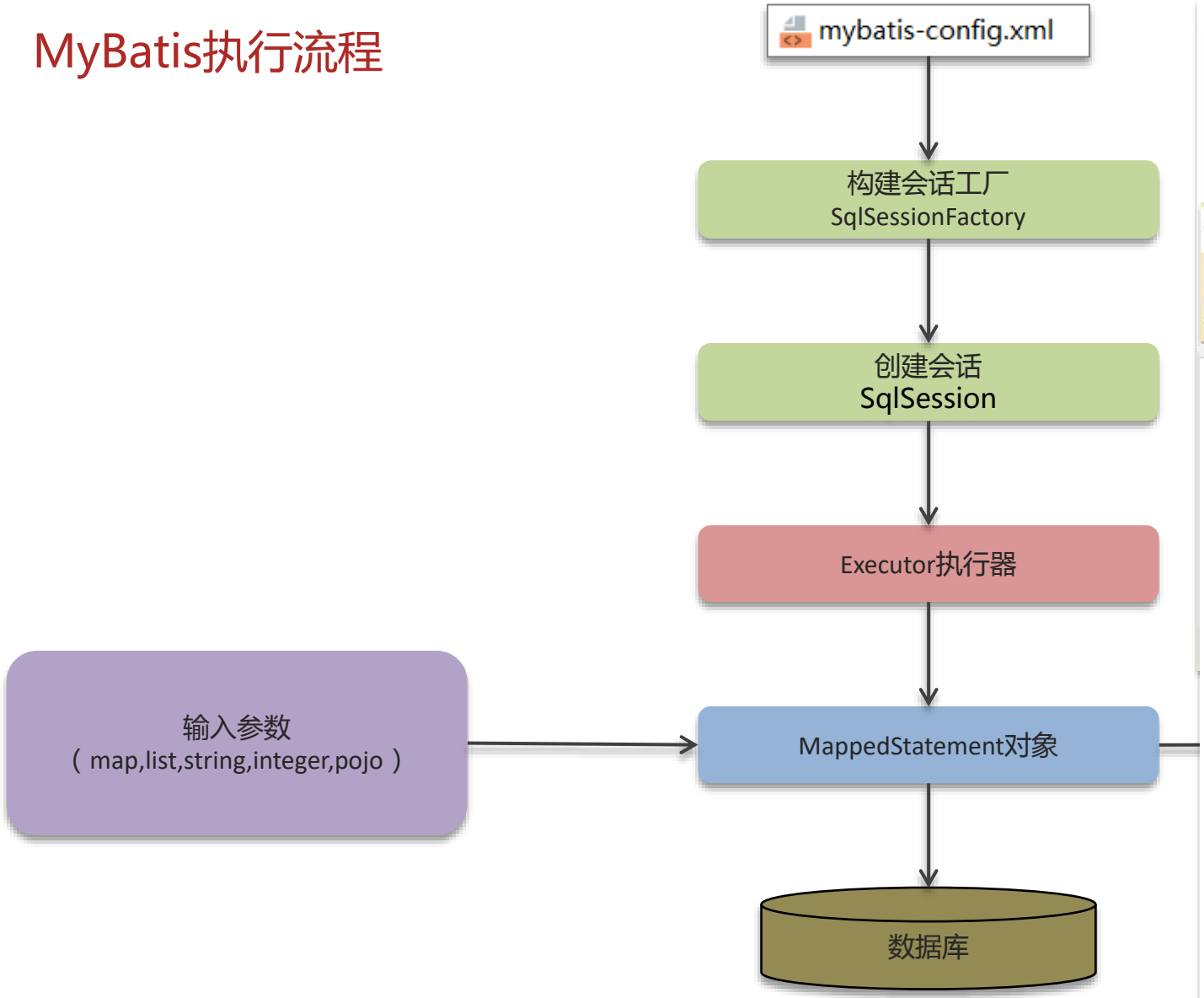
注解	
@RequestMapping	用于映射请求路径，可以定义在类上和方法
@RequestBody	注解实现接收http请求的json数据，将json转
@RequestParam	指定请求参数的名称
@PathViriable	从请求路径下中获取请求参数(/user/{id})，
@ResponseBody	注解实现将controller方法返回对象转化为js
@RequestHeader	获取指定的请求头数据
@RestController	@Controller + @ResponseBody



MyBatis执行流程

- 理解了各个组件的关系
- Sql的执行过程（参数映射、sql解析、执行和结果处理）

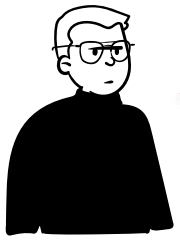
MyBatis执行流程




```xml
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <!--数据库连接信息-->
            <property name="driver" value="com.mysql.jdb
<select id="selectById" resultType="user">
    select * from tb_user where id = #{id};
</select>
```

```
this = {SimpleExecutor@1961}
ms = {MappedStatement@1957}
  resource = "com/itheima/mapper/UserMapper.xml"
  configuration = {Configuration@1967}
  id = "com.itheima.mapper.UserMapper.selectById"
  fetchSize = null
  timeout = null
  statementType = {StatementType@1970} "PREPARED"
  resultSetType = {ResultSetType@1971} "DEFAULT"
  sqlSource = {RawSqlSource@1972}
    sqlSource = {StaticSqlSource@2253}
      sql = "select * from tb_user where id = ?;"
      parameterMappings = {ArrayList@2255}  size = 1
      configuration = {Configuration@1967}
  cache = null
  parameterMap = {ParameterMap@1973}
  resultMaps = {Collections$UnmodifiableRandomAccessList@1974}  size
  flushCacheRequired = false
  useCache = true
```

MyBatis执行流程



① 读取MyBatis配置文件：mybatis-config.xml加载运行环境和映射文件

② 构造会话工厂SqlSessionFactory

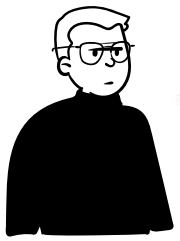
③ 会话工厂创建SqlSession对象（包含了执行SQL语句的所有方法）

④ 操作数据库的接口，Executor执行器，同时负责查询缓存的维护

⑤ Executor接口的执行方法中有一个MappedStatement类型的参数，封装了映射信息

⑥ 输入参数映射

⑦ 输出结果映射



Mybatis是否支持延迟加载？

Mybatis支持延迟记载，但默认没有开启

什么叫做延迟加载？

用户表 1 n 订单表

```
private Integer id;
private String username;
private String password;
private String gender;
private String addr;
private LocalDateTime birthday;
private List<Order> orderList;
```

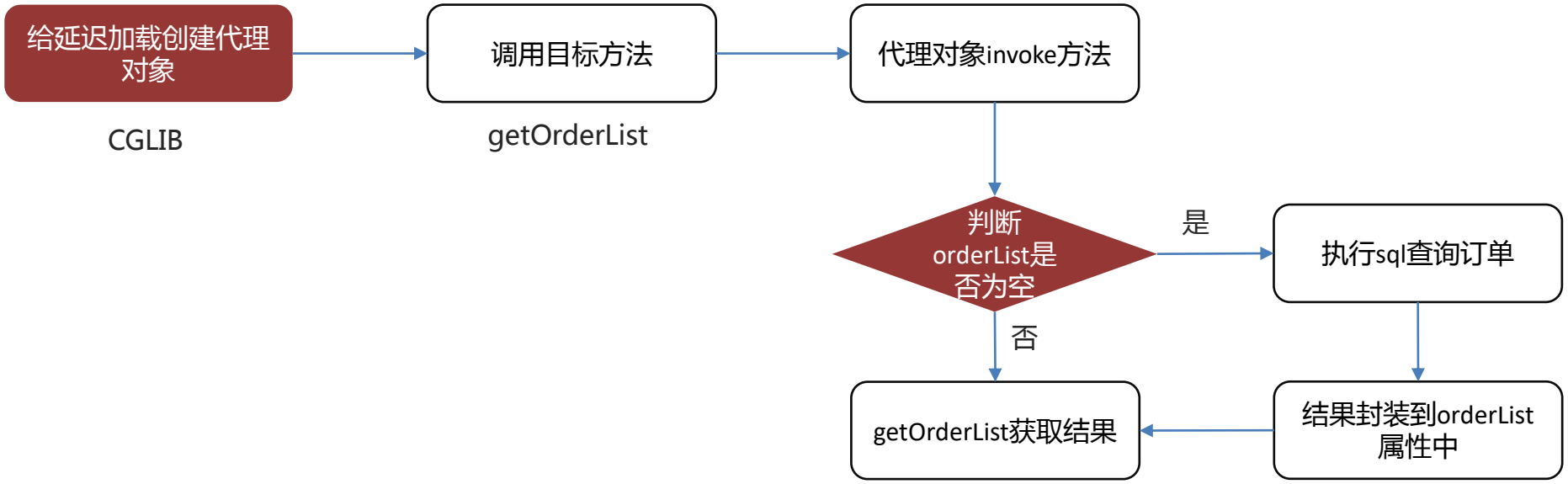
```
private Integer id;
private String name;
private Integer userId;
private Integer status;
private User user;
```

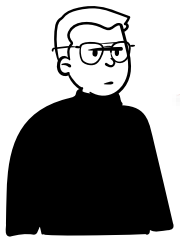
查询用户的时候，把用户所属的订单数据也查询出来，这个是立即加载

查询用户的时候，暂时不查询订单数据，当需要订单的时候，再查询订单，这个就是延迟加载

延迟加载的原理

1. 使用CGLIB创建目标对象的代理对象
2. 当调用目标方法user.getOrderList()时，进入拦截器invoke方法，发现user.getOrderList()是null值，执行sql查询order列表
3. 把order查询上来，然后调用user.setOrderList(List<Order> orderList) ，接着完成user.getOrderList()方法的调用



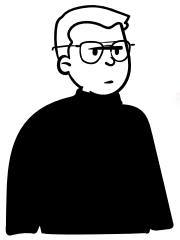


Mybatis是否支持延迟加载？

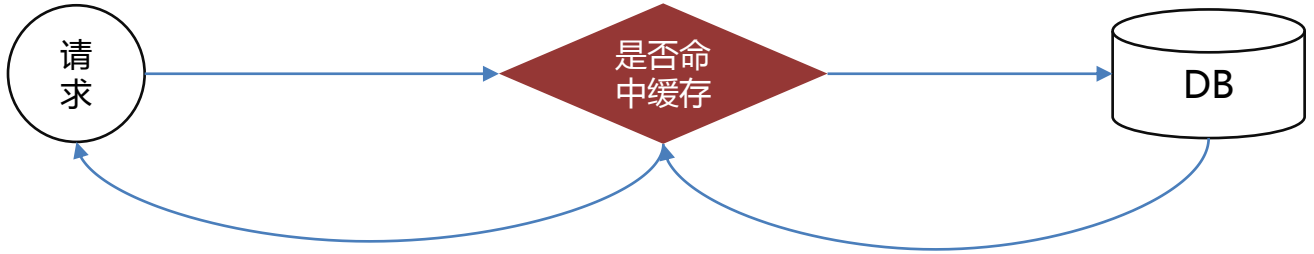
- 延迟加载的意思是：就是在需要用到数据时才进行加载，不需要用到数据时就不加载数据。
- Mybatis支持一对一关联对象和一对多关联集合对象的延迟加载
- 在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false，默认是关闭的

延迟加载的底层原理知道吗？

1. 使用CGLIB创建目标对象的代理对象
2. 当调用目标方法时，进入拦截器invoke方法，发现目标方法是null值，执行sql查询
3. 获取数据以后，调用set方法设置属性值，再继续查询目标方法，就有值了



Mybatis的一级、二级缓存用过吗？



- 本地缓存，基于PerpetualCache，本质是一个HashMap
- 一级缓存：作用域是session级别
- 二级缓存：作用域是namespace和mapper的作用域，不依赖于session

一级缓存

一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当Session进行flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存

```
//2. 获取SqlSession对象，用它来执行sql
SqlSession sqlSession = sqlSessionFactory.openSession();
//3. 执行sql
//3.1 获取UserMapper接口的代理对象
UserMapper userMapper1 = sqlSession.getMapper(UserMapper.class);
UserMapper userMapper2 = sqlSession.getMapper(UserMapper.class);

User user = userMapper1.selectById(6);
System.out.println(user);

System.out.println("----------------------");
User user1 = userMapper2.selectById(6);
System.out.println(user1);
```

只会执行一次sql查询

二级缓存

二级缓存是基于namespace和mapper的作用域起作用的，不是依赖于SQL session，默认也是采用 PerpetualCache，HashMap 存储

```
//2. 获取SqlSession对象，用它来执行sql
SqlSession sqlSession1 = sqlSessionFactory.openSession();

//3. 执行sql
//3.1 获取UserMapper接口的代理对象
UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);

User user1 = userMapper1.selectById(6);
System.out.println(user1);
sqlSession1.close();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

System.out.println("----------------------");
UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
User user2 = userMapper2.selectById(6);
System.out.println(user2);

//4.关闭资源
sqlSession2.close();
```

查询两次SQL

二级缓存默认是关闭的

开启方式，两步：

1，全局配置文件

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

2，映射文件

使用<cache/>标签让当前mapper生效二级缓存

二级缓存

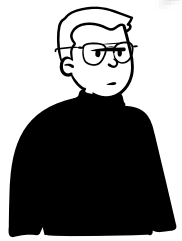
注意事项：

1，对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了新增、修改、删除操作后，默认该作用域下所有 select 中的缓存将被 clear

2，二级缓存需要缓存的数据实现Serializable接口

3，只有会话提交或者关闭以后，一级缓存中的数据才会转移到二级缓存中

Mybatis的一级、二级缓存用过吗？



- 一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当Session进行flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存
- 二级缓存是基于namespace和mapper的作用域起作用的，不是依赖于SQL session，默认也是采用PerpetualCache，HashMap 存储。需要单独开启，一个是核心配置，一个是mapper映射文件

Mybatis的二级缓存什么时候会清理缓存中的数据

当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了新增、修改、删除操作后，默认该作用域下所有 select 中的缓存将被 clear。


黑马程序员
www.itheima.com
传智教育旗下高端IT教育品牌