简介

spring是一个IoC和AOP框架

Spring的特性：

- 非侵入式：基于Spring开发的应用中的对象可以不依赖于Spring的API
- 依赖注入：DI（Dependency Injection）是反转控制（IoC：Inversion of Control）最经典的实现
- 面向切面编程：Aspect Oriented Programming（AOP）
- 容器：Spring提供一个容器，包含并管理应用对象的生命周期
- 组件化：Spring将众多简单的组件配置组合成一个复杂应用

Spring的核心就是提供了一个IoC容器，它可以管理所有轻量级的JavaBean组件，提供的底层服务包括组件的生命周期管理、配置和组装服务、AOP支持，以及建立在AOP基础上的声明式事务服务等。

容器

组件和容器

组件

具有一定功能的对象，比如servlet组件、DAO组件、service业务组件

容器

管理组件(创建、获取、保存、销毁)

比如tomcat是servlet容器

IoC和DI

IoC

- 控制：资源的控制权（资源的创建，获取，销毁等）
- 反转：对于资源的操作交给容器来做，而不是我们

DI

- 依赖：组件的依赖关系，比如NewsController依赖NewsServices（不是指jar包）
- 注入：通过setter方法、构造器等方式自动的注入（赋值）

具体来说，Spring IoC 容器管理的是应用程序中的 **Bean**，这些 Bean 实际上是通过类创建的对象实例。容器负责这些对象的生命周期管理、配置属性设置以及依赖注入等。

Spring IoC 容器管理的内容

1. **Bean 实例**：IoC 容器中存储的是由开发者定义的类所创建的对象（即Bean）。这些对象可以是服务层对象、数据访问对象（DAO）、控制器等任何类型的组件。
2. **依赖关系**：当一个 Bean 需要另一个 Bean 来完成其功能时，IoC 容器会自动将所需的依赖注入到该 Bean 中。这种依赖通常是其他对象实例。
3. **配置元数据**：这包括如何创建 Bean 的信息，如构造函数参数、属性值等。配置可以通过 XML 文件、注解或者 Java 配置类来提供。
4. **生命周期回调**：某些 Bean 可能需要在初始化后或销毁前执行特定的操作，IoC 容器支持注册相应的初始化和销毁方法。

注册

把组件放进容器中

```
//1.作用：跑起一个SPRING应用
//ApplicationContext：Spring应用上下文对象；ioc容器
ConfigurableApplicationContext ioc =
SpringApplication.run(Spring01IocApplication.class, args);
```

```
//3.给容器中注册一个自己的组件
//容器中的每一个组件都有自己的名字，方法名就是组件的名字，但是可以通过@Bean后的括号配置
@Bean("zhangsan")
public Person zhangsan(){
    Person person = new Person();
    person.setName("张三");
    person.setAge(18);
    person.setGender("男");
    return person;
}
```

获取组件

添加了以下组件：

```
@Bean("zhangsan")
public Person zhangsan(){
    Person person = new Person();
    person.setName("张三");
    person.setAge(18);
    person.setGender("男");
    return person;
}
@Bean("lisi")
public Person lisi(){
    Person person = new Person();
    person.setName("张三");
```

```
    person.setAge(18);
    person.setGender("男");
    return person;
}

@Bean
public Dog dog(){
    return new Dog();
}
```

获取的不同情况：

```
        //4.获取容器中的组件对象
        //组件的四大特性：名字、类型、作用域、对象
        //可以通过名字、类型获得组件对象

        //小结
        //从容器中获取组件
        //1）、如果获取组件而组件不存在,抛异常NoSuchBeanDefinitionException
        //2）、组件不唯一，
        //       用单个对象接收 NoUniqueBeanDefinitionException
        //       用集合接收，可以
        //3）、组件唯一，正确返回

        //4.1.按照组件的名字获取对象
        Person zhangsan = (Person) ioc.getBean("zhangsan");
        System.out.println("对象="+zhangsan);

        Dog dog = (Dog) ioc.getBean("dog");
        System.out.println("dog="+dog);

        //4.2.按照组件类型获取对象
//      Person bean=ioc.getBean(Person.class);
//      System.out.println("bean="+bean);

        //4.3.按照组件类型获取这种类型的所有对象
        Map<String, Person> map = ioc.getBeansOfType(Person.class);
        System.out.println("map="+map);

        //4.4按照类型加名字
        Person zhangsan1 = ioc.getBean("zhangsan", Person.class);
        System.out.println("zhangsan1="+zhangsan1);
```

组件的创建时机:

```
@Data
public class Dog {
        public Dog() {
            System.out.println("dog构造器");
        }
}


public static void main(String[] args) {
```

```
    ConfigurableApplicationContext ioc =
SpringApplication.run(Spring01IocApplication.class, args);
    System.out.println("=====ioc容器创建完成====");

    //2.获取组件
    Dog bean = ioc.getBean(Dog.class);
    System.out.println("bean="+bean);

    Dog bean1 = ioc.getBean(Dog.class);
    System.out.println("bean="+bean1);

    Dog bean2 = ioc.getBean(Dog.class);
    System.out.println("bean="+bean2);

}
```

输出：

```
dog构造器
2025-03-05T18:54:42.679+08:00  INFO 3788 --- [spring-01-ioc] [           main]
c.a.spring.ioc.Spring01IocApplication    : Started Spring01IocApplication in
0.577 seconds (process running for 0.902)
=====ioc容器创建完成====
bean=Dog()
bean=Dog()
bean=Dog()
```

可以发现，在ioc容器启动过程中就创建了组件对象

并且只创建了一次对象，这说明对象是单例的,每次获取直接从容器中拿。容器会提前创建组件

配置类

可以发现我们之前的添加组件的方式会使主程序类中的代码显得十分繁琐，这就可以用到配置类

我们可以给Bean（存放Person、Dog等类的包）创建一个同级目录config，并创建DogConfig、PersonConfig类（配置类），将以上添加组件的方法分类写进配置类。并添加`@Configuration`,这样也可以达到自定义组件的效果

```
@Configuration //告诉Spring容器，这是一个配置类
public class PersonConfig {
    //3.给容器中注册一个自己的组件
    //容器中的每一个组件都有自己的名字，方法名就是组件的名字
    @Bean("zhangsan")
    public Person zhangsan(){
        Person person = new Person();
        person.setName("张三");
        person.setAge(18);
        person.setGender("男");
        return person;
    }
    @Bean("lisi")
    public Person lisi(){
```

```
        Person person = new Person();
        person.setName("张三");
        person.setAge(18);
        person.setGender("男");
        return person;
    }

}
```

注意：配置类也是容器中的组件

MVC分层注解

- @Controller　控制层
- @Service　　服务层
- @Repository　持久层
- @Component　组件

分层注解只是方便人为识别，其实底层都是@Component普通组件

注意：分层注解起作用的前提是，这些组件必须在主程序所在的包及其子包结构下

批量扫描

```
@ComponentScan(basePackages="com")
```

组件批量扫描，将com包下用相关注解标记过的组件注册到容器中

但是规范编程一般就将主程序所在包当作包的根，其他包或java类都在此包下进行

导入第三方组件

第三方库想要导入容器中，无法快速标注分层注解

方法：

1. @Bean自己new，注册给容器（记得先import对应的库）

```
比如CoreConstants
import ch.qos.logback.core.CoreConstants;
....
public CoreConstants coreConstants(){
        return new CoreConstants();
}
```

2. 使用注解@Import(CoreConstants.class)

为了防止主程序上的注解太多导致乱，我们在config包下定义一个AppConfig配置类，将用法类似的注解(如ComponentScan)标在其上（不会影响注解的功能，因为在添加此配置类为组件之前，会扫描注解，将注解中标示要添加为组件的组件也加入容器）

```
@Import(CoreConstants.class)
@Configuration
@ComponentScan(basePackages="com")
public class AppConfig{
}
```

如果多个，放在大括号里，用逗号隔开

调整组件作用域

1. @Scope("prototype")　非单实例

   容器启动的期间不会创建非单实例组件的对象，什么时候获取什么时候创建

2. @Scope("singleton")　单实例

   容器启动的期间会创建非单实例组件的对象,容器启动完成前就创建了

3. @Scope("request")　同一个请求单实例

4. @Scope("session")　同一次绘画单实例

不写Scope注解的情况下，相当于singleton

3、4用到很少

单例情况下的懒加载

即让单例组件也变成：

容器启动的期间不会创建非单实例组件的对象，什么时候获取什么时候创建

利用工厂制造复杂Bean

应用场景：要制造的对象比较复杂，利用工厂方法进行创建

Bean:组件(实例)

在根目录（主程序所在包）下，创建一个factory包，编写一个类实现FactoryBean接口，并重写里面的方法

```
@Component
public class BYDFactory implements FactoryBean<Car> {
    //调用此方法给容器中制造对象
    @Override
    public Car getObject() throws Exception {
        Car car = new Car();
        return car;
    }
    //说明造的东西的类型
    @Override
    public Class<?> getObjectType() {
        return Car.class;
    }
```

```
    //true：单例； false：非单例
    @Override
    public boolean isSingleton() {
        return FactoryBean.super.isSingleton();
    }
}
```

```
public static void main(String[] args) {
    ConfigurableApplicationContext context =
SpringApplication.run(Spring01IocApplication.class, args);
    Car car = context.getBean(Car.class);
    Map<String,Car> cars = context.getBeansOfType(Car.class);
    System.out.println("cars: " + cars);
}
```

通过输出的结果`cars: {BYDFactory=com.atguigu.spring.ioc.bean.Car@41aaedaa}`可知，工厂创建的组件的名和工厂的名字一致，而类型则是FactoryBean<>中<>里面的泛型类型

@Conditional

此注解适用于方法和类

条件注册

满足条件的情况下，才加入此组件

举例：当前环境是windows，加入bill组件，mac则加入joseph组件

```
@Conditional(WindowsCondition.class)
@Bean
public Person joseph(){
    Person person = new Person();
    person.setName("乔布斯");
    person.setAge(18);
    person.setGender("男");
    return person;
}
@Conditional(MacCondition.class)
@Bean
public Person bill(){
    Person person = new Person();
    person.setName("比尔盖茨");
    person.setAge(18);
    person.setGender("男");
    return person;
}
```

```
@Conditional的参数类型是Class<? extends Condition>[]

public class WindowsCondition implements Condition {
    @Override
```

```
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        Environment environment = context.getEnvironment();
        String osName = environment.getProperty("OS");
        return osName.toLowerCase().contains("windows");

    }
}


public class MacCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        Environment environment = context.getEnvironment();
        String osName = environment.getProperty("OS");
        return osName.toLowerCase().contains("mac");
    }
}
```

```
public static void main(String[] args) {
        ConfigurableApplicationContext context =
SpringApplication.run(Spring01IocApplication.class, args);
        Map<String,Person> persons = context.getBeansOfType(Person.class);
        System.out.println("beans=" + persons);
}
```

最终显示只有比尔盖茨组件

@Conditional派生注解

- @ConditionOnMissingBean（）：没有括号内参数对应的组件时，添加
- @ConditionOnBean（）：有括号内参数对应的组件时，添加
- @ConditionOnResource（）：有资源，添加

```
@ConditionOnMissingBean（value={Person.class}）
@ConditionOnMissingBean（name="bill"）

@ConditionOnBean（name="bill"）

@ConditionOnResource（resources="classpath:haha.abc"）
//classpath表示未来打包进target/classes的包，main下的java和resources包都是
```

注入

@Autowired自动装配

原理：Spring调用 容器.getBean()

流程：

1. 按照类型，找到这个组件；

    1.0 如果仅有一个此类组件，直接注入

    1.1 如果找到多个，再按名字去找；变量名就是名字

        1.1.1 如果找到，注入

        1.1.2 找不到，报错

即先按照类型，再按照组件名

```
@ToString
@Data
@Controller
public class UserController {
    @Autowired
    UserService userService;

    @Autowired
    Person bill;

    //把所有这个类型的组件都拿来
    @Autowired
    List<Person> people;

    @Autowired
    Map<String, Person> map;

    //拿到ioc容器
    @Autowired
    ApplicationContext applicationContext;

}
```

```
public static void main(String[] args) {
    ConfigurableApplicationContext ioc =
SpringApplication.run(Spring01IocApplication.class, args);
    System.out.println("====================");

    UserController userController = ioc.getBean(UserController.class);
    System.out.println("UserController = " + userController);


}
```

输出：

```
//获取单个
UserController =
UserController(userService=com.atguigu.spring.ioc.service.UserService@5ee34b1b,
bill=Person(name=比尔盖茨, age=18, gender=男),

//获取全部同类对象
people=[Person(name=张三, age=18, gender=男), Person(name=null, age=0,
gender=null), Person(name=乔布斯, age=18, gender=男), Person(name=比尔盖茨, age=18,
gender=男)],

map={Zhangsan=Person(name=张三, age=18, gender=男), lisi=Person(name=null, age=0,
gender=null), joseph=Person(name=乔布斯, age=18, gender=男), bill=Person(name=比尔
盖茨, age=18, gender=男)},

//ioc容器

applicationContext=org.springframework.context.annotation.AnnotationConfigApplic
ationContext@6dee4f1b, started on Thu Mar 06 08:10:57 CST 2025)
```

@Qualifier和@Primary

@Qualifier（组件名）：

精确指定，如果容器中这样的组件有多个，则使用@Qualifier精确指定组件名

```
@ToString
@Data
@Service
public class UserService {
    @Qualifier("bill")
    @Autowired
    Person person;

}
```

@Primary:

指定主要，按类型获取单个，如果有多个组件，获取主要的

```
@Primary
//@Conditional(WindowsCondition.class)
@Bean
public Person bill(){
    Person person = new Person();
    person.setName("比尔盖茨");
    person.setAge(18);
    person.setGender("男");
    return person;
}
```

@Resource自动注入

```
@ToString
@Data
@Service
public class UserService {
/*    @Qualifier("bill")
    @Autowired
    Person person;*/
    @Resource
    UserDao userDao;

}
```

@Resource和@Autowired的区别?

- `@Autowired`：这是 Spring 框架提供的注解，属于 Spring 自己的依赖注入机制的一部分。
- `@Resource`：这是 Java 标准提供的注解，因此它不仅可以在 Spring 中使用，也可以在其他支持该标准的框架中使用。

@Resource具有更强的通用性

且@Autowired可以调为required=false

所以以后直接用@Autowired就行

构造器注入

```
@ToString
@Data
@Repository
public class UserDao {
    Dog dog;
    //因为UserDao标记为Repository,且UserDao类只有有参构造器
    //Spring在创建UserDao组件时调用有参构造器，就会传入一个Dog，实现注入
    public UserDao(Dog dog) {
        this.dog = dog;
    }
}
```

setter方法注入

```
@Autowired
public void setDog(Dog dog) {
    this.dog = dog;
}
//这里的dog也可以指定
@Autowired
public void setDog(@Qualifier("dog1")Dog dog) {
    this.dog = dog;
}
```

xxxAware感知接口

在Spring框架中，“感知接口”是指一系列特殊的接口，通过实现这些接口，Bean可以表明它们需要容器提供某些信息或资源。

```
@Service
public class hahaService implements EnvironmentAware {
    private Environment environment;


    @Override
    public void setEnvironment(Environment environment) {
        this.environment = environment;
    }

    public String getOsType(){
        return environment.getProperty("os.name");
    }
}
```

Spring发现实现了Environment接口，会自动将环境变量传入

@value 属性赋值

- @Value("字面值") :直接赋值
- @Value（“${key}”）:动态从配置文件取出某一项的值
- @Value（“${key:默认值}”）:动态从配置文件取出某一项的值,取不到则用默认值
- @Value("#{SpEL}") :Spring Expression Language;spring表达式语言。在其中的代码可以执行,甚至可以写url

比如:

```
@Value("#{T(java.util.UUID).randomUUID().toString()}")
private String id;

@Value("#{10*2}")
private Integer age;

@Value("#{'Hello World'.substring(0,5)}")
private String msg;
```

@ PropertySource

在使用value从配置文件中动态取出值时，如果要配置的属性很多，那么application.properties配置文件会显得冗长

所以在resources包下创建dog.properties等，将关于狗的配置写进

但是这样会发现报错，无法赋值

所以说明属性来源，把指定的文件**导入**容器中，供我们取值使用

```
@PropertySource("classpath:dog.properties")
@Data
@Component
public class Dog {
        public Dog() {
            System.out.println("dog构造器");
        }
}
```

补充:

classpath:表示从自己项目的类路径下找

classpath*: 表示从所有包的类路径下找

ResourceUtils获取资源

```
File file=ResourceUtils.getFile("classpath:abc.jpg");
int available=new FileInputStream(file).available();
sout(available);
//获取资源大小
```

还可以用url等参数获取资源，更多参数类型见源码

@Profile 多环境

```
@Configuration
public class dataSourceConfig {

    //1.定义环境标识
    //2.激活环境标识:
       //明确告诉spring当前环境
       //不说就是defalt环境

    //利用环境注解，只在某种环境下激活一个组件
    @Profile("dev")// @profile（“环境标识”）
    @Bean
    public MyDataSource dev() {
        MyDataSource dataSource = new MyDataSource();
```

```
        dataSource.setUrl("jdbc:mysql://localhost:3306/dev");
        dataSource.setUsername("dev_user");
        dataSource.setPassword("dev_pass");
        return dataSource;
    }
    @Profile("test")
    @Bean
    public MyDataSource test() {
        MyDataSource dataSource = new MyDataSource();
        dataSource.setUrl("jdbc:mysql://localhost:3306/test");
        dataSource.setUsername("test_user");
        dataSource.setPassword("test_pass");
        return dataSource;

    }
    @Profile("prod")
    @Bean
    public MyDataSource prod(){
        MyDataSource dataSource = new MyDataSource();
        dataSource.setUrl("jdbc:mysql://localhost:3306/dev");
        dataSource.setUsername("prod_user");
        dataSource.setPassword("prod_pass");
        return dataSource;
    }
}
```

```
@Component
public class DeliveryDao {
    /*
    * 问题一:数据源组件有三个
    * 告诉数据源，哪种数据源在哪种情况下生效。想到@Conditional
    * 但除此之外，还有@Profile
    * */
        @Autowired
        MyDataSource dataSource;
        void saveDelivery() {
            System.out.println("数据源："+dataSource);
        }
}
```

如何告诉spring当前环境呢?

在配置文件中写`spring.profiles.active=`后面加环境

此外，@Profile还可定义在类上,表示在指定环境下类才生效

生命周期

@Bean指定初始化、销毁方法

```
@Data
```

```
public class User {
    private String name;
    private String password;
    private Car car;
    @Autowired
    public void setCar(Car car) {
        System.out.println("自动注入");
        this.car = car;
    }
    public User(){
        System.out.println("User 构造器");
    }
    public void initUser(){
        System.out.println("@Bean 初始化：initUser");
    }
    public void destroyUser(){
        System.out.println("@Bean 销毁：destroyUser");
    }
}
```

```
@Configuration
public class UserConfig {
       //添加组件并
        @Bean(initMethod = "initUser",destroyMethod = "destroyUser")
        public User user() {
            return new User();
        }
}
```

```
public static void main(String[] args) {
    ConfigurableApplicationContext ioc =
SpringApplication.run(Spring01IocApplication.class, args);
    System.out.println("======ioc容器创建完成=========");
    User bean=ioc.getBean(User.class);
    System.out.println("运行："+bean);
}
```

输出：

```
User 构造器
自动注入
@Bean 初始化：initUser
...
======ioc容器创建完成=========
运行：User(name=null, password=null, car=com.atguigu.spring.ioc.bean.Car@af78c87)
@Bean 销毁：destroyUser
```

可以发现，组件的创建、属性值自动注入、组件初始化都在ioc创建完成前进行

组件的生命周期

构造器创建-->@Autowired属性注入-->@Bean指定的init方法进行初始化-->运行-->@Bean指定的destory方法进行销毁-->容器结束

InitializingBean、DisposableBean接口

```
@Data
public class User implements InitializingBean, DisposableBean {
    private String name;
    private String password;

    private Car car;

    @Autowired
    public void setCar(Car car) {
        System.out.println("自动注入");
        this.car = car;
    }

    public User() {
        System.out.println("User 构造器");
    }

    public void initUser() {
        System.out.println("@Bean 初始化：initUser");
    }

    public void destroyUser() {
        System.out.println("@Bean 销毁：destroyUser");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("destroy");
    }

    //见名知意，属性设置后（set赋值完成了）调用
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("afterPropertiesSet");
    }
}
```

```
User 构造器
自动注入
afterPropertiesSet
@Bean 初始化：initUser
...
======ioc容器创建完成=========
运行：User(name=null, password=null,
car=com.atguigu.spring.ioc.bean.Car@393881f0)
destroy
@Bean 销毁：destroyUser
```

所以 InitializingBean定义的afterPropertiesSet()在自动注入后、初始化组件前调用

DisposableBean定义的destory()在@Bean指定的destory方法前调用

@PreDestory、@PostConstruct

@PostConstruct：构造器执行后

@PreDestory：销毁前

后置处理器BeanPostProcessor接口

使用举例：在初始化后，给User类的组件的属性赋值

```
@Component
public class MyTestBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        if(bean instanceof User){
            bean.setUserName("");
        }
        return bean;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        return bean;
    }
}
```

其实@Autowired的实现也是靠后置处理器：

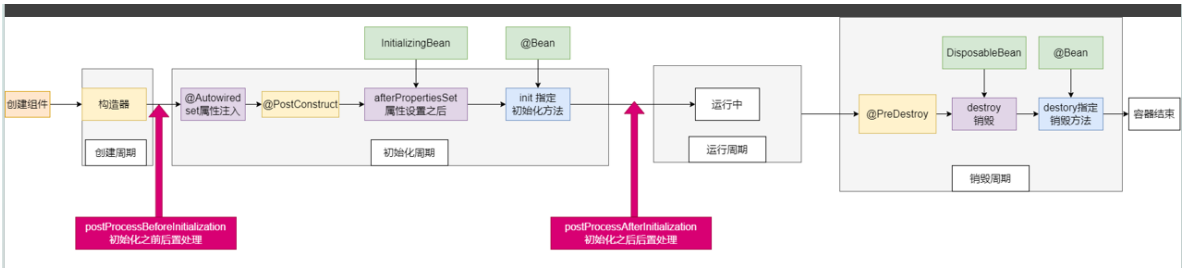
1.专门有一个处理@Autowired注解的AutowiredAnnotationBeanPostProcessor

2.每个Bean创建后，调用BeanPostProcessor的postProcessBeforeInitialization方法

3.postProcessBeforeInitialization会利用反射得到当前Bean的所有属性，利用反射，得到Bean属性上标注的所有注解，看有没有@Autowired注解

4.如果有，去容器中找到这个属性对应的组件（按类型、按名字找到）并装配（或报错）

小结



AOP

Aspect Oriented Programming(面向切面编程)

用日志引入

静态代理

静态代理是一种设计模式，它通过创建一个实现相同接口的代理类来控制对实际服务对象的访问。在静态代理中，“静态”意味着代理类是预先定义好的，并且在编译期就已经确定，而不是在运行时动态生成。

静态代理的基本概念

- **目标对象（Real Subject/target）**：指的是真正执行业务逻辑的对象。
- **代理对象（Proxy）**：持有一个对目标对象的引用，并在调用方法之前或之后添加额外的行为（如日志记录、权限验证等），然后再调用目标对象的方法。

实现步骤

1. **定义接口**：首先需要定义一个接口，该接口声明了所有目标对象和代理对象都将实现的方法。
2. **实现目标对象**：创建一个类实现上述接口，这个类就是我们想要代理的目标对象。
3. **创建代理对象**：创建另一个类同样实现该接口，在这个类中持有目标对象的一个实例，并重写接口中的方法，在这些方法中可以添加额外的功能，然后调用目标对象对应的方法。

```
public interface MathCalculate {
    //定义 四则运算
    int add(int a, int b);
    int sub(int a, int b);
    int mul(int a, int b);
    int div(int a, int b);
}
```

```
public class MathCalculateImpl implements MathCalculate {
    @Override
    public int add(int a, int b) {
        int result = a + b;
        System.out.println("结果:"+result);
        return result;
    }
    @Override
    public int sub(int a, int b) {
        int result = a - b;
        return result;
    }
    @Override
    public int mul(int a, int b) {
        int result = a * b;
        return result;
    }
    @Override
    public int div(int a, int b) {
        int result = a / b;
        return result;
```

```
    }
}
```

```
public class CalculateStaticProxy implements MathCalculate {

    private MathCalculate target;
    public CalculateStaticProxy(MathCalculate target) {
        this.target = target;
    }

    @Override
    public int add(int a, int b) {
        System.out.println("【日志】 add 开始 参数："+a+","+b);
        int result = target.add(a, b);
        System.out.println("【日志】 add 返回 结果："+result);
        return result;
    }

    @Override
    public int sub(int a, int b) {
        int result = target.sub(a, b);
        return result;
    }

    @Override
    public int mul(int a, int b) {
        int result = target.mul(a, b);
        return result;
    }

    @Override
    public int div(int a, int b) {
        int result = target.div(a, b);
        return result;
    }
}
```

```
public class MathTest {
    @Test
    void test() {
        MathCalculate mathCalculate = new MathCalculateImpl();
        CalculateStaticProxy calculateStaticProxy = new
CalculateStaticProxy(mathCalculate);
        int result=calculateStaticProxy.add(1,2);
        System.out.println(result);
    }
}
```

输出：

```
【日志】 add 开始 参数：1,2
结果:3
【日志】 add 返回 结果：3
3
```

- **优点**：
  - 简单直观，易于理解和实现。
  - 可以在不修改目标对象的前提下增加额外功能。
- **缺点**：
  - 每个业务类都需要一个代理类，增加了类的数量，可能导致维护成本上升。
  - 如果接口增加新的方法，则需要同时更新目标类和代理类。
- **适用场景**：
  - 当需要在不改变原始代码的情况下给某些方法添加额外功能时。
  - 如日志记录、事务管理、权限验证等横切关注点的处理。

动态代理

简单实现

运行期间才决定好代理关系

定义：目标对象在执行期间被动态拦截，插入指定逻辑 (拦截器的思想)

```
public interface MathCalculate {
    //定义 四则运算
    int add(int a, int b);
    int sub(int a, int b);
    int mul(int a, int b);
    int div(int a, int b);
}
```

```
public class MathCalculateImpl implements MathCalculate {
    @Override
    public int add(int a, int b) {
        int result = a + b;
        System.out.println("结果:"+result);
        return result;
    }
    @Override
    public int sub(int a, int b) {
        int result = a - b;
        return result;
    }
    @Override
    public int mul(int a, int b) {
        int result = a * b;
        return result;
    }
    @Override
    public int div(int a, int b) {
        int result = a / b;
        return result;
    }
}
```

```
public class MathTest {
    @Test
    void test(){
        MathCalculate target = new MathCalculateImpl();
        InvocationHandler ih=new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                System.out.println("执行target方法前");
                Object result=method.invoke(target,args);
                System.out.println("执行target方法，得到result后");
                return 0;
            }
        };
        MathCalculate proxy =(MathCalculate) Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                ih
        );
        int result1=proxy.add(1,2);
        System.out.println(result1);
    }
```

使用了匿名内部类的方式创建了一个InvocationHandler接口的实现类的对象ih

用Proxy.newProxyInstance()方法生成代理类的对象（代理类是JVM在运行时动态生成的），强转成MathCalculate类型

用proxy对象调用add方法，实际上将调用转发给了ih，并调用invoke方法（这些我并没有写出来，应该是JVM运行时执行的）

invoke方法使我们可以实现日志的加入

ai说：

> 当你调用`Proxy.newProxyInstance()`时，JVM会在运行时为代理对象动态生成一个具体的类（通常是在内存中），这个类实现了你提供的所有接口。每个方法都包含将调用转发给`InvocationHandler`的逻辑

动态代理的核心概念

1. **Proxy类**：提供用于创建动态代理类和实例对象的方法。
2. **InvocationHandler接口**：所有动态代理类都需要实现的一个接口，定义了如何处理代理对象的方法调用。

工作流程概述

- **步骤1：定义接口和实现类**：首先需要有一个或多个接口以及这些接口的具体实现类。
- **步骤2：创建InvocationHandler**：实现`InvocationHandler`接口，定义如何处理方法调用。
- **步骤3：使用Proxy.newProxyInstance()创建代理对象**：通过提供类加载器、接口数组及自定义的`InvocationHandler`来创建代理对象。
- **步骤4：调用代理对象的方法**：当你调用代理对象上的方法时，实际上是在调用由JVM生成的代理类中对应的方法，这些方法会自动将调用转发给关联的`InvocationHandler`的`invoke`方法。

JVM可能会生成类似如下的代理类（伪代码）：

```
public final class $Proxy0 extends Proxy implements Service {
```

```
    private static Method m1;

    public $Proxy0(InvocationHandler h) {
        super(h);
    }

    @Override
    public void performTask() {
        try {
            // 将方法调用转发给 InvocationHandler 的 invoke 方法
            super.h.invoke(this, m1, null);
        } catch (Error | RuntimeException e) {
            throw e;
        } catch (Throwable e) {
            throw new UndeclaredThrowableException(e);
        }
    }

    static {
        try {
            m1 = Service.class.getMethod("performTask");
        } catch (NoSuchMethodException e) {
            throw new NoSuchMethodError(e.getMessage());
        }
    }
}
```

类加载器的作用

`newProxyInstance`方法的第一个参数是一个类加载器，用于加载动态生成的代理类。类加载器负责将生成的代理类加载到VM中，使其可以被实例化和使用。

改写

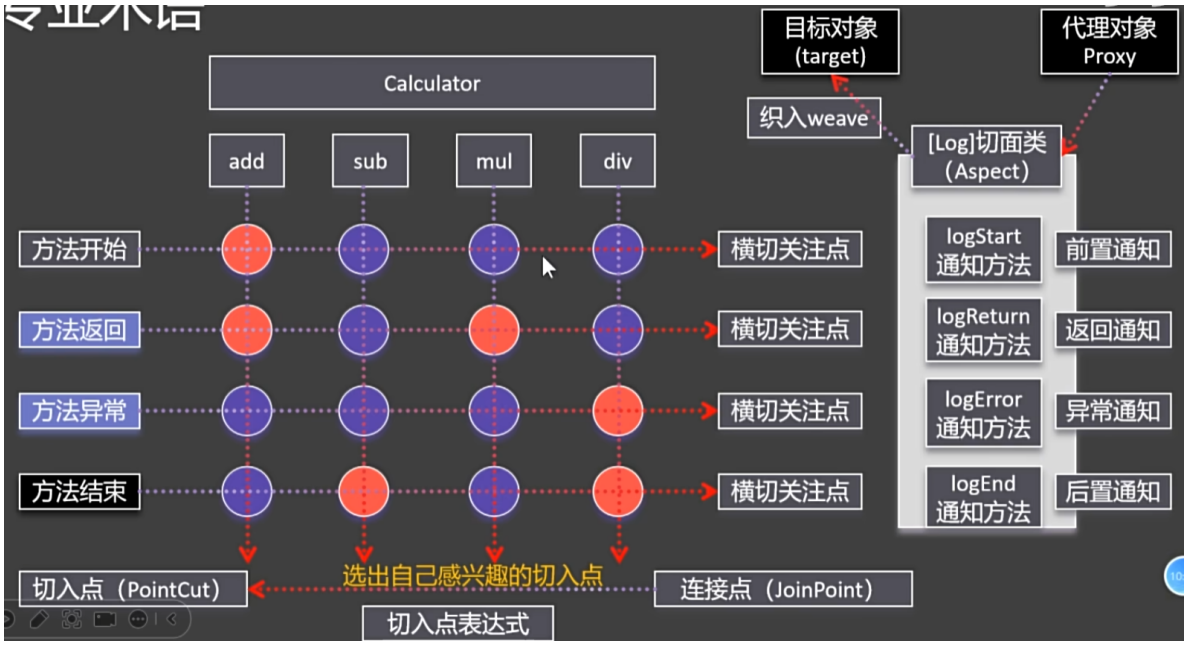
对上面的简单实现进行改写

```
//获得代理对象的静态方法
public class DynamicProxy {

    public static Object getProxyInstance(Object target) {

        return Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                (proxy, method, args) -> {
                    String name = method.getName();
                    System.out.println("【日志】：【"+name+"】开始：参数:"+
Arrays.toString(args));
                    Object result = method.invoke(target, args);
                    System.out.println("【日志】：【"+name+"】结束");
                    return result;
                }
                );
    }
}
```

```
public class MathTest {
    @Test
    public void test() {
        MathCalculate proxy = (MathCalculate) DynamicProxy.getProxyInstance(new
MathCalculateImpl());
        proxy.add(1,2);
    }
```

无论哪个类型的对象需要动态代理，都可以调用上面的静态方法获得代理对象

AOP专业术语



切面实现示例

编写切面和通知方法实现AOP

- 导入AOP依赖

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

- 编写切面Aspect、编写通知方法、指定切入点表达式

```
@Aspect//告诉spring这个组件是个切面
@Component
public class LogAspect {

    /*
    * 告诉spring，以下通知何时何地运行
    *何时？
    * @Before 方法执行前运行
```

```
    * @AfterReturning 方法执行正常返回结果运行
    * @AfterThrowing  方法抛出异常运行
    * @After 方法执行之后运行
    *何地？
    *    切入点表达式
    *        execution(方法的全签名)
    *                全签名：
    *                全写法:public int
com.atguigu.spring.aop.calculate.MathCalculate.add(int a,int b) throws
ArithmeticException
    *                省略写法:int add(int a,int b)
    *                允许通配符
    *
    * */

    @Before("execution(int *(int,int))")
    public void logStart(){
        System.out.println("【切面-日志】开始...");
    }

    @After("execution(int *(int,int))")
    public void logEnd(){
        System.out.println("【切面-日志】结束...");
    }

    @AfterReturning("execution(int *(int,int))")
    public void logReturn(){
        System.out.println("【切面-日志】返回...");
    }

    @AfterThrowing("execution(int *(int,int))")
    public void logException(){
        System.out.println("【切面-日志】异常...");
    }


}
```

- 测试AOP动态织入

```
@SpringBootTest
public class AopTest {

    @Autowired
    MathCalculate mathCalculate;
    @Test
    void test01(){
        //System.out.println(mathCalculate);
        mathCalculate.add(1,2);
    }
}
```

输出:

```
【切面-日志】开始...
结果:3
【切面-日志】返回...
【切面-日志】结束...
```

切入点表达式的其他写法

- args：参数是指定类型或其子类型的任何连接点

```
@Before("args(int,int)")
```

- @annotation :判定方法上有没有标注解

```
@Before("@annotation(注解的全路径)")
```

细节

AOP底层原理

1.Sping会为每个被切面切入的组件创建代理对象（Sping CGLIB创建的，没有接口也可以）

2.代理对象中保存了切面类里所有通知方法构成的增强器链

3.目标方法执行时，会执行增强器链中需要执行的通知方法

JoinPoint连接点信息

JoinPoint：包装了当前目标方法的所有信息

示例：

```
    @Before("execution(int *(int,int))")
    public void logStart(JoinPoint joinPoint) {
        //拿到方法全签名
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        //方法名
        String name=signature.getName();
        //目标方法传来的参数值
        Object[] args = joinPoint.getArgs();

        System.out.println("【切面-日志】 【"+name+"】 开始:参数列表:【"+
Arrays.toString(args)+"】");
    }

    @After("execution(int *(int,int))")
    public void logEnd(JoinPoint joinPoint) {
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        String name = signature.getName();
        System.out.println("【切面-日志】【"+name+"】后置...");
    }

    @AfterReturning(value="execution(int *(int,int))",
            returning="returnValue")//returning="returnValue"获取目标方法返回值
```

```
    public void logReturn(JoinPoint joinPoint,Object returnValue){
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        String name = signature.getName();

        System.out.println("【切面-日志】【"+name+"】返回:"+returnValue);
    }

    @AfterThrowing(value = "execution(int *(int,int))",
    throwing="e")//获取目标方法抛出的异常
    public void logException(JoinPoint joinPoint,Throwable e){
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        String name = signature.getName();
        System.out.println("【切面-日志】【"+name+"】异常...错误信息:
【"+e.getMessage()+"】");
    }


}
```

@Pointcut抽取切入点表达式

```
@Pointcut("execution(int *(int,int))")
public void Pointcut(){};


@Before("Pointcut()")
public void logStart(JoinPoint joinPoint) {
    //拿到方法全签名
    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    //方法名
    String name=signature.getName();
    //目标方法传来的参数值
    Object[] args = joinPoint.getArgs();

    System.out.println("【切面-日志】 【"+name+"】 开始:参数列表:【"+
Arrays.toString(args)+"】");
}
```

多切面执行顺序

单切面执行顺序：前置-->目标方法-->返回/异常-->后置

我们再添加一个切面：

```
@Component
@Aspect
public class AuthAspect {
    @Pointcut("execution(int *(int,int))")
    public void Pointcut(){};

    @After("Pointcut()")
    public void after(){
```

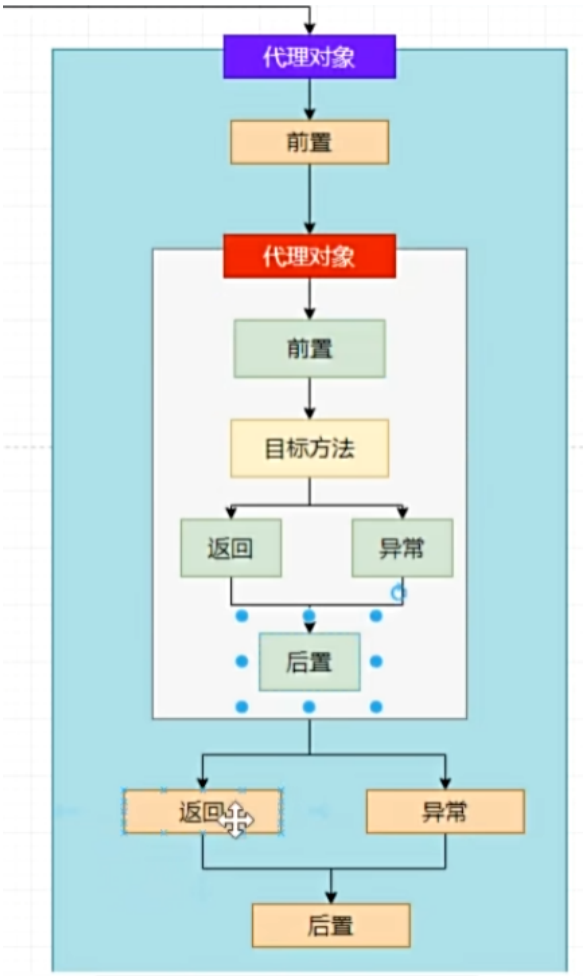
```
        System.out.println("【切面-权限】后置");
    }
    @Before("Pointcut()")
    public void before(){
        System.out.println("【切面-权限】前置");
    }
    @AfterReturning("Pointcut()")
    public void afterReturning(){
        System.out.println("【切面-权限】返回");
    }
    @AfterThrowing("Pointcut()")
    public void afterThrowing(){
        System.out.println("【切面-权限】异常");
    }
}
```

多切面执行的顺序是怎样的呢?

```
【切面-权限】前置
【切面-日志】 【add】 开始:参数列表:【[1, 2]】
结果:3
【切面-日志】【add】返回:3
【切面-日志】【add】后置...
【切面-权限】返回
【切面-权限】后置
```

在这个例子中，其实日志切面是权限切面的目标方法

可以用一个图来表示:



可以用@order（数字）决定切面优先级顺序，数字越小优先级越高。对应的切面也就在上图的越外层

三级缓存

假设我们有两个单例Bean A和B，它们之间存在循环依赖：

- 当创建Bean A时，发现它依赖于Bean B，因此尝试去创建Bean B。
- 在创建Bean B的过程中，发现它依赖于Bean A。此时，如果Bean A已经存在于`singletonFactories`（三级缓存）中，则可以从这里获取Bean A的一个早期引用，将其放入`earlySingletonObjects`（二级缓存），并返回给Bean B使用。
- 接着继续完成Bean B的初始化，之后再回到Bean A的创建流程，完成Bean A的初始化。
- 最终，将完全初始化好的Bean A和Bean B放入`singletonObjects`（一级缓存）中。

关键点总结

- **一级缓存**：存储完全初始化好的Bean实例。
- **二级缓存**：存储早期暴露的Bean实例，即尚未完成全部初始化步骤的Bean实例。
- **三级缓存**：存储Bean工厂对象，用于生成早期的Bean实例。

详见DefaultSingletonBeanFactory类的getSingleton(String,boolean)方法

环绕通知

@Around :可以控制目标方法是否执行，修改目标方法参数、执行结果等

环绕通知固定写法如下：

```
public Object aroundAdvice(ProceedingJointPoint pjp) throws Throwable{
}
```

举例：

```
@Aspect
@Component
public class AroundAspect {
    @Pointcut("execution(int *(int,int))")
    public void Pointcut(){};
    @Around("Pointcut()")
    public Object aroundAdvice(ProceedingJoinPoint jpj) throws Throwable {
        //获取目标方法的参数
        Object[] args = jpj.getArgs();
        //前置
        System.out.println("环绕-前置通知 参数："+ Arrays.toString(args));
        Object result = null;
        //调用目标方法并返回
        try {
            result= jpj.proceed(args);
            System.out.println("环绕-返回通知 返回值："+result);
        } catch (Throwable e) {
            System.out.println("环绕-异常通知"+e.getMessage());
            throw e;//一定要抛出，不然外一层切面感知不到异常
        }finally {
            System.out.println("环绕-后置通知");
```

```
        }
        return result;
    }
}
```

应用场景

日志记录

- **描述**：在系统中的不同层次和位置添加日志记录，以便追踪应用程序的行为。
- **示例**：在方法调用前后记录输入参数、输出结果以及执行时间。

事务管理

- **描述**：确保一组操作要么全部成功，要么全部失败，以保持数据的一致性和完整性。
- **示例**：在服务层的方法上应用事务控制，自动开始、提交或回滚数据库事务。

缓存优化

- **描述**：减少对昂贵资源（如数据库查询）的访问次数，通过缓存结果来加速响应速度。
- **示例**：为频繁调用但计算成本高的方法添加缓存逻辑，首次调用时存储结果，后续调用直接返回缓存值。

异常处理

- **描述**：统一处理异常，避免在每个可能抛出异常的地方编写相同的错误处理代码。
- **示例**：定义全局异常处理器，在捕获到特定类型的异常时执行预定义的操作，如记录错误日志或向客户端发送错误信息。

权限检查等。。。

事务

准备数据源、JdbcTemplate

- 创建模块时勾选Spring Data JDBC和MySQL Driver
- 在application.properties中写入

```
spring.datasource.url=jdbc:mysql://localhost:3306/spring_tx
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=abc123
```

就可以对数据库进行操作了

- 还提供了JdbcTemplate方便对数据库进行操作

@Transactional声名事务

```
//开启基于注释的自动化事务管理
@EnableTransactionManagement
```

```
//给方法加上事务
@Transactional
```

事务管理器参数

作用：控制事务的获取、提交、回滚

底层默认使用JdbcTransactionManager

原理：

事务管理器配合事务拦截器

1.事务管理器TransactionManager :控制提交和回滚

2.事务拦截器TransacntionInterceptor：控制何时提交和回滚

2.1completeTransactionAfterThrowing（txInfo，ex）用此方法回滚

2.2commitTransactionAfterReturning（txInfo）用此方法提交

timeout参数

设定超时时间，事务超过此事件限制则回滚。

int类型, 以s为单位

细节:超时时间是指从方法开始到最后一次数据库操作结束的时间（如果在最后一次dao操作后，使用Thread.sleep(time)使超过时间，不算超时）

rollbackFor参数

类型：Class<? extends Throwable>[]

作用：不是所有异常都一定引起事务回滚，所以我们通过rollbackFor**额外**指明哪些异常需要回滚

异常：

运行时异常(unchecked exception【非受检异常】)

编译时异常(checked exception【受检异常】)

回滚的默认机制:

运行时异常：回滚

编译时异常：不回滚

通常指定为:

```
@Transaction(robackFor={Exception.class})
```

同时还有noRollbackFor参数（不说了）

隔离级别参数

数据库给写的操作加了锁，所以不用担心会发生问题。但是如果一读一写，那么就需要控制读，避免发生问题。

- **READ_UNCOMMITTED**

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
public BigDecimal getBookPriceById(Integer id) {
    String sql = "select * from book where id = ?";
    return jdbcTemplate.queryForObject(sql, BigDecimal.class, id);

}
```

给test加上断点，在命令行中开启事务并修改数据

可以读到尚未提交的数据

此外，还有READ_COMMITTED , REPEATABLE_READ , SERIALIZABLE等隔离等级，回看mysql笔记以复习。

mysql默认REPEATABLE_READ

propagation传播行为

传播行为	产生效果
REQUIRED	支持当前事务，如果不存在则创建一个新的事务
SUPPORTS	支持当前事务，如果不存在则非事务性执行
MANDATORY	支持当前事务，如果不存在则抛出异常
REQUIRES_NEW	创建一个新事务，并在存在当前事务时挂起当前事务
NESTED	如果当前存在事务，则在嵌套事务中执行，否则像 REQUIRED 一样运行
NOT_SUPPORTED	非事务执行，如果存在当前事务则暂停
NEVER	非事务性地执行，如果存在事务则抛出异常

比如在checkout方法（已开启事务）中给扣除金额的方法设置REQUIRED_NEW，给减少库存的方法设置REQUIRED，那么当check事务回滚时，金额不回滚，库存回滚。

注意一定要考虑出现异常的情况，比如上面例子中如果REQUIRE_NEW的方法抛出异常，虽然不在同一个事务，但抛出的异常可能会导致checkout整个大事务回滚。