【初学者慎入】Spring源码中的16种设计模式实现

Spring Framework 是一个庞大而复杂的框架，它涵盖了多个模块和组件，每个组件都有其独特的功能和作用。V 哥一直建议同学们在学习 Spring 时需要学习 Spring 的源码，说句实话，Spring 源码太 TM 优秀了，你不仅可以理解原理，还能在学习 Spring 源码的过程中，学到设计模式的应用，以下就是威哥在学习 Spring 源码时整理的16种设计模式应用的地方，分享给你。

Spring 源代码实现中使用了许多常见的设计模式，这些设计模式帮助 Spring 框架实现了灵活、可扩展和可维护的特性。以下是 Spring 源代码中常见的设计模式及其解释：

1、工厂模式（Factory Pattern）

在 Spring 中，BeanFactory 和 ApplicationContext 是工厂模式的一个典型应用。它们负责根据配置信息或注解来创建和管理 bean 实例，隐藏了具体对象的创建细节，提供了一种灵活的方式来管理对象的生命周期。

BeanFactory：

抽象工厂角色：BeanFactory 接口是抽象工厂角色，它定义了获取和管理 bean 对象的方法。

具体工厂角色：DefaultListableBeanFactory 类是 BeanFactory 接口的一个具体实现类，它负责具体的 bean 创建和管理工作。

角色分析：实现代码： 在 DefaultListableBeanFactory 类中，通过 ConcurrentHashMap 来存储 bean 的定义信息，即 bean 名称与对应的 BeanDefinition 对象之间的映射关系。当需要获取 bean 时，通过调用 getBean(String name) 方法，容器会根据名称从 beanDefinitionMap 中获取对应的 BeanDefinition对象，并根据 BeanDefinition 的配置信息创建或获取 bean 实例。

ApplicationContext：

抽象工厂角色：ApplicationContext 接口继承了 BeanFactory 接口，除了提供了获取和管理 bean 对象的方法外，还提供了更多的企业级特性，如国际化、事件发布、资源加载等。

具体工厂角色：AbstractApplicationContext 类是 ApplicationContext 接口的一个具体实现类，它继承了 DefaultListableBeanFactory，同时还实现了 ApplicationEventPublisher、MessageSource、ResourceLoader 等接口，提供了额外的功能。

角色分析：实现代码： AbstractApplicationContext 类在初始化过程中会加载所有的 bean 定义并创建 BeanFactory 实例，同时会对其他功能进行初始化，如事件监听器的注册、消息源的加载等。当需要获取 bean 时，可以通过调用 getBean(String name) 方法获取对应的 bean 实例。此外，AbstractApplicationContext 还提供了其他方法来支持事件发布、消息获取等功能。

2、单例模式（Singleton Pattern）

Spring 中的 BeanFactory 和 ApplicationContext 默认情况下都是单例的，即容器中的 bean 默认为单例对象。这样可以确保在整个应用程序中只有一个实例存在，节省了系统资源并提高了性能。BeanFactory 和 ApplicationContext 实现单例模式的机制略有不同。我们来看看它们的源码如何实现单例模式：

BeanFactory

在 BeanFactory 中，单例模式的实现主要依赖于缓存机制。BeanFactory 使用一个 ConcurrentHashMap 来缓存已经创建的单例对象。

```
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
```

当调用 getBean() 方法获取 bean 实例时，BeanFactory 会首先尝试从缓存中获取，如果缓存中存在对应的单例对象，则直接返回；如果缓存中不存在，则根据 bean 的定义信息创建新的实例对象，并放入缓存中。

```
public Object getBean(String name) {
    synchronized (this.singletonObjects) {
        Object singletonObject = this.singletonObjects.get(name);
        if (singletonObject == null) {
            singletonObject = createBean(name);
            this.singletonObjects.put(name, singletonObject);
        }
        return singletonObject;
    }
}
```

ApplicationContext

ApplicationContext 也使用了类似的缓存机制来实现单例模式，不过相比于 BeanFactory，ApplicationContext 会在容器启动时预先实例化所有的单例 bean，并将其放入缓存中，以便在应用程序运行期间能够快速获取到单例对象。

```
// AbstractApplicationContext.java

// 创建 singletonObjects 缓存
private final Map<String, Object> singletonObjects = new
ConcurrentHashMap<>(256);

// 预加载所有单例 bean
protected void refreshBeanFactory() throws BeansException {
    // ...
    // 预先实例化所有的 singleton bean
    preInstantiateSingletons();
}

// 预先实例化所有 singleton bean
protected void preInstantiateSingletons() throws BeansException {
    // 获取 bean 定义的名称列表
    String[] beanNames = getBeanDefinitionNames();
    // 遍历所有的 bean 定义
    for (String beanName : beanNames) {
        // 获取 bean 的定义信息
        RootBeanDefinition bd =
getMergedLocalBeanDefinition(beanName);
```

```
        // 判断是否是单例 bean
        if (bd.isSingleton()) {
            // 创建单例 bean 实例并放入缓存中
            Object bean = getBean(beanName);
            this.singletonObjects.put(beanName, bean);
        }
    }
}
```

BeanFactory 和 ApplicationContext 实现单例模式的机制都是基于缓存的方式，通过缓存已经创建的单例对象来确保每次调用 getBean() 方法获取单例对象时都返回同一个实例。这样可以节省系统资源，并且保证了单例对象的一致性。

3、代理模式（Proxy Pattern）

Spring AOP 中的代理模式是实现切面功能的关键。Spring 使用代理对象来实现切面功能，它可以在方法调用前后执行额外的逻辑，如日志记录、性能监控等。

在 Spring 框架中，AOP（面向切面编程）的实现主要依赖于代理模式。Spring 使用动态代理来实现 AOP，主要有两种代理方式：基于 JDK 动态代理和基于 CGLIB 动态代理。

下面简要介绍 Spring 源码中如何使用代理模式实现 AOP：

基于 JDK 动态代理的 AOP：

当目标对象实现了接口时，Spring 使用 JDK 动态代理来创建 AOP 代理对象。在 JDK 动态代理中，代理对象实现了目标对象的所有接口，并将方法调用委托给 MethodInvocation 接口的实现类来处理。

Spring 源码中主要涉及到以下几个类和接口：

java.lang.reflect.Proxy：JDK 提供的动态代理类，用于创建代理对象。

org.springframework.aop.framework.JdkDynamicAopProxy：Spring 中用于基于 JDK 动态代理的 AOP 代理对象生成器。

org.springframework.aop.framework.ProxyFactory：Spring 中用于创建 AOP 代理对象的工厂类。

基于 CGLIB 动态代理的 AOP：

当目标对象没有实现接口时，Spring 使用 CGLIB 动态代理来创建 AOP 代理对象。在 CGLIB 动态代理中，代理对象继承了目标对象，并重写了目标对象的方法，以便在方法调用前后执行额外的逻辑。

Spring 源码中主要涉及到以下几个类和接口：

org.springframework.cglib.proxy.Enhancer：CGLIB 提供的动态代理类，用于创建代理对象。

org.springframework.aop.framework.CglibAopProxy：Spring 中用于基于 CGLIB 动态代理的 AOP 代理对象生成器。

无论是基于 JDK 动态代理还是基于 CGLIB 动态代理，Spring 都提供了统一的 AOP API，即 org.springframework.aop 包下的一系列接口和类，包括切面、通知、连接点等，以及各种 AOP 相关的配置方式，如 XML 配置、注解配置等。

Spring 框架通过代理模式实现了 AOP，使得开发者可以方便地在应用程序中添加和管理切面逻辑，实现了横切点的解耦和重用。

4、装饰器模式（Decorator Pattern）
==========

Spring 的 BeanPostProcessor 接口就是装饰器模式的一个应用。通过实现 BeanPostProcessor 接口，可以在 bean 实例化后、初始化前后动态地添加额外的处理逻辑。

在 Spring 源码中，BeanPostProcessor 接口的实现是基于装饰者模式的，它允许我们在 bean 实例化、依赖注入和初始化阶段对 bean 进行额外的处理

，而不需要修改原始的 bean 类。

下面是 Spring 源码中如何使用装饰者模式实现 BeanPostProcessor：

BeanPostProcessor 接口定义：

首先，我们来看一下 BeanPostProcessor 接口的定义：

```
package org.springframework.beans.factory.config;

public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException;

    Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException;
}
```

BeanPostProcessor 接口定义了两个方法：postProcessBeforeInitialization 和 postProcessAfterInitialization。它们允许我们在 bean 初始化前后对 bean 进行额外的处理。

AbstractApplicationContext 类：

在 Spring 源码中，AbstractApplicationContext 类实现了 BeanFactory 和 ApplicationContext 接口，并且提供了对 BeanPostProcessor 的支持。

```
package org.springframework.context.support;

public abstract class AbstractApplicationContext extends
DefaultResourceLoader implements ConfigurableApplicationContext,
DisposableBean {
    // ...
    protected void
finishBeanFactoryInitialization(ConfigurableListableBeanFactory
beanFactory) {
        // ...
        // 注册 BeanPostProcessor，对 bean 进行装饰
        for (BeanPostProcessor processor :
beanFactory.getBeanPostProcessors()) {
            beanFactory.addBeanPostProcessor(processor);
        }
```

```
        // ...
    }
    // ...
}

```

在 AbstractApplicationContext 类中，通过调用 finishBeanFactoryInitialization 方法，会注册所有的 BeanPostProcessor 实例到 BeanFactory 中，从而对 bean 进行装饰。

AbstractAutowireCapableBeanFactory 类：

最后，我们来看一下 AbstractAutowireCapableBeanFactory 类，它是 DefaultListableBeanFactory 类的父类，负责实现了 BeanFactory 的核心功能。

```
package org.springframework.beans.factory.support;

public abstract class AbstractAutowireCapableBeanFactory extends
AbstractBeanFactory implements AutowireCapableBeanFactory {
    // ...
    protected void invokeAwareMethods(final String beanName, final
Object bean) {
        // ...
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            if (processor instanceof Aware) {
                if (processor instanceof BeanNameAware) {
                    ((BeanNameAware) processor).setBeanName(beanName);
                }
                // ...
            }
        }
        // ...
    }
    // ...
}

```

在 AbstractAutowireCapableBeanFactory 类中，通过调用 invokeAwareMethods 方法，会遍历所有的 BeanPostProcessor 实例，如果发现实现了特定的 Aware 接口（如 BeanNameAware），则调用其对应的方

法来设置 bean 的相关属性。

Spring 源码中使用装饰者模式实现了 BeanPostProcessor 接口，允许我们在 bean 实例化、依赖注入和初始化阶段对 bean 进行额外的处理，从而实现了更灵活、可扩展的 bean 处理机制。

5、观察者模式（Observer Pattern）
==========

Spring 的事件机制是观察者模式的一个应用。通过定义事件和监听器，可以实现对象之间的解耦，当事件发生时通知所有注册的监听器进行处理。

在 Spring 源码中，观察者模式被广泛应用于实现事件机制。Spring 的事件机制允许应用程序中的组件监听特定类型的事件，并在事件发生时执行相应的逻辑。下面是 Spring 源码中如何使用观察者模式实现事件机制的简要说明：

（1）事件类：

Spring 中的事件通常由具体的事件类表示，这些事件类通常继承自 ApplicationEvent 类。每个事件类通常包含与事件相关的信息，例如事件源、时间戳等。

在 Spring 中，事件类通常继承自 ApplicationEvent 类，例如 ContextRefreshedEvent、ContextStartedEvent 等。这些事件类包含了与事件相关的信息。

（2）事件监听器接口：

Spring 提供了一个 ApplicationListener 接口，用于监听特定类型的事件。监听器实现该接口，并通过泛型指定要监听的事件类型。

Spring 提供了 ApplicationListener 接口，用于监听特定类型的事件。事件监听器实现该接口，并实现 onApplicationEvent() 方法来处理事件。

```
public interface ApplicationListener<E extends ApplicationEvent>
```

```
extends EventListener {
    void onApplicationEvent(E event);
}
```

（3）事件发布器：

Spring 提供了一个 ApplicationEventPublisher 接口，用于发布事件。ApplicationContext 接口继承了 ApplicationEventPublisher，因此 ApplicationContext 是事件发布器的一个实现。

ApplicationContext 接口继承了 ApplicationEventPublisher 接口，因此 ApplicationContext 是事件发布器的一个实现。通过调用 publishEvent() 方法来发布事件。

```
public interface ApplicationEventPublisher {
    void publishEvent(ApplicationEvent event);
}
```

（4）事件监听器注册：

在 Spring 中，事件监听器可以通过两种方式进行注册：

通过在 Spring 配置文件中配置监听器 bean。

通过代码注册监听器。

事件监听器可以通过在 Spring 配置文件中配置监听器 bean，或者通过代码注册监听器来进行注册。

（5）事件发布：

当某个事件发生时，通过事件发布器发布该事件。事件发布器会将事件发送给所有注册的事件监听器。

```
context.publishEvent(new MyCustomEvent(this));
```

（6）事件处理：

事件监听器收到事件后，会执行相应的事件处理逻辑。每个事件监听器可以根据需要监听多个不同类型的事件。

小结：Spring 源码中使用观察者模式实现了事件机制，通过事件类、事件监听器接口、事件发布器和事件监听器注册等组件，使得应用程序中的组件可以监听特定类型的事件，并在事件发生时执行相应的逻辑。这种设计实现了组件之间的解耦和灵活的事件处理机制。

6、策略模式（Strategy Pattern）

Spring 中的各种策略接口和实现类，如 ResourceLoader、Environment、EnvironmentCapable 等，都是策略模式的应用。它们允许用户根据不同的需求选择不同的实现方式。

在 Spring 源码中，ResourceLoader、Environment 和 EnvironmentCapable 都是使用策略模式实现的。策略模式允许在运行时动态地选择算法或行为。下面分别介绍这三个接口在 Spring 源码中的实现方式：

ResourceLoader：

ResourceLoader 接口定义了资源加载器的行为，它允许应用程序在运行时动态地加载资源。在 Spring 源码中，ResourceLoader 接口的实现主要有 ClassPathResourceLoader、FileSystemResourceLoader 和 UrlResourceLoader。

ClassPathResourceLoader：用于从类路径加载资源。

FileSystemResourceLoader：用于从文件系统加载资源。

UrlResourceLoader：用于从 URL 地址加载资源。

这些实现类通过实现 ResourceLoader 接口，提供了不同的资源加载策略，从而使得应用程序能够根据需要选择不同的资源加载方式。

Environment 和 EnvironmentCapable：

Environment 接口用于表示应用程序的环境，它提供了一种统一的方式来获取环境相关的信息，如属性值、配置文件等。EnvironmentCapable 接口用于表示具有环境能力的组件，即可以获取到 Environment 对象的组件。

在 Spring 源码中，Environment 接口的实现主要有 StandardEnvironment、MutablePropertySources 和 PropertySource 等。而 EnvironmentCapable 接口的实现主要是 ApplicationContext 接口。

StandardEnvironment：提供了标准的环境实现，用于获取系统属性、环境变量、配置文件等。

MutablePropertySources：用于管理属性源，即配置文件、环境变量等来源。

PropertySource：表示属性源，可以是配置文件、环境变量等。

这些实现类通过实现 Environment 接口和 EnvironmentCapable 接口，提供了统一的获取环境信息的策略，使得应用程序能够灵活地获取环境相关的信息，并根据需要选择不同的环境配置。

小结：Spring 源码中使用策略模式实现了 ResourceLoader、Environment 和 EnvironmentCapable 接口，通过提供不同的实现类来提供不同的资源加载策略和环境获取策略，使得应用程序能够根据需要选择不同的实现方式，并实现了组件之间的解耦。

7、模板方法模式（Template Method Pattern）
==========

Spring 中的 JdbcTemplate 等是模板方法模式的应用。它们定义了执行数据库操作的标准流程，但将具体的操作委托给子类来实现。

在 Spring 源码中，JdbcTemplate 使用了模板方法模式（Template Method Pattern）来简化数据库访问操作，封装了常见的数据库操作流程，提供了模板方法供开发者使用，同时留出了部分方法供子类进行具体实现，以满足不同的数据库访问需求。下面简要介绍 Spring 源码中如何使用模板方法模式实现 JdbcTemplate：

（1）JdbcTemplate 类结构：

JdbcTemplate 类是 Spring JDBC 模块中的核心类，用于简化 JDBC 操作。它包含了一系列模板方法，如 query、update、batchUpdate 等，以及一些回调方法供开发者实现。

（2）模板方法模式在 JdbcTemplate 中的应用：

JdbcTemplate 类中的模板方法主要是 query、update、batchUpdate 等方法，它们定义了执行数据库操作的标准流程，但具体的操作实现由子类或者回调函数来完成。

（3）具体实现类：

在 JdbcTemplate 中，具体的数据库操作实现由 RowMapper、PreparedStatementSetter、ResultSetExtractor 等接口的实现类来完成。这些接口的实现类负责将 JDBC 操作所需的参数设置、结果集映射等具体逻辑进行实现。

（4）模板方法：

JdbcTemplate 类中的模板方法包含了一系列常见的数据库操作，如 query、update、batchUpdate 等。这些模板方法定义了数据库操作的流程，并提供了回调方法供开发者实现自定义的逻辑。

（5）示例代码：

下面是使用 JdbcTemplate 进行数据库查询的示例代码：

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
List<User> users = jdbcTemplate.query(
    "SELECT * FROM users",
    (resultSet, rowNum) -> {
        User user = new User();
        user.setId(resultSet.getLong("id"));
        user.setUsername(resultSet.getString("username"));
        user.setEmail(resultSet.getString("email"));
        return user;
    }
);
```

在这个示例中，query 方法是 JdbcTemplate 中的一个模板方法，它定义了执行查询操作的标准流程，但具体的结果集映射由回调函数 (resultSet, rowNum) -> {...} 完成，这就是模板方法模式的应用。

8、适配器模式（Adapter Pattern）

Spring MVC 中的 HandlerAdapter 就是适配器模式的应用。它允许不同类型的处理器（Controller）以统一的方式来处理请求，通过适配器将不同类型的处理器适配成统一的处理器接口。

在 Spring 源码中，HandlerAdapter 接口的实现是通过适配器模式实现的。HandlerAdapter 接口定义了处理器适配器的行为，它允许 Spring MVC 框架调用不同类型的处理器（Controller、HandlerMethod 等）并执行相应的处理逻辑。下面是 Spring 源码中如何使用适配器模式实现 HandlerAdapter 的简要说明：

HandlerAdapter 接口定义：

HandlerAdapter 接口定义了对处理器的调用方式，一般包括多个方法，如 supports() 方法用于判断适配器是否支持某种类型的处理器，handle() 方法用于执行处理器。

```
public interface HandlerAdapter {
    boolean supports(Object handler);

    ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception;
}

```

具体适配器类：

Spring 源码中提供了多个具体的适配器类，用于适配不同类型的处理器，如 SimpleControllerHandlerAdapter、HttpRequestHandlerAdapter、AnnotationMethodHandlerAdapter 等。

```
public class SimpleControllerHandlerAdapter implements HandlerAdapter
{
    public boolean supports(Object handler) {
        return (handler instanceof SimpleController);
    }

    public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        // 处理 SimpleController 类型的处理器
    }
}

```

具体的适配器类根据支持的处理器类型来判断是否支持某种类型的处理器，并实现相应的处理逻辑。

DispatcherServlet 中的调用：

在 DispatcherServlet 中，根据请求的处理器类型选择合适的 HandlerAdapter 来执行处理器。

```
protected ModelAndView haHandle(HttpServletRequest request,
```

```
HttpServletResponse response, Object handler) throws Exception {
    for (HandlerAdapter ha : this.handlerAdapters) {
        if (ha.supports(handler)) {
            return ha.handle(request, response, handler);
        }
    }
    return null;
}
```

DispatcherServlet 会遍历所有注册的 HandlerAdapter，找到支持当前处理器类型的适配器，并调用其 handle() 方法执行处理器。

通过适配器模式，Spring 实现了 HandlerAdapter 接口的多种具体适配器类，每个适配器类负责将不同类型的处理器适配到 HandlerAdapter 接口上，从而统一处理器的调用方式，提高了处理器的复用性和灵活性。

9、建造者模式（Builder Pattern）

在 Spring 中，通过建造者模式可以简化复杂对象的构建过程。比如，Spring 中的 RestTemplateBuilder 就是建造者模式的应用，用于构建 RestTemplate 实例。

在 Spring 源码中，RestTemplate 类的创建使用了建造者模式（Builder Pattern）。建造者模式允许逐步构建复杂对象，并且使得构建过程更加灵活，同时可以保证对象的一致性。下面是 Spring 源码中如何使用建造者模式实现 RestTemplate 的简要说明：

RestTemplateBuilder 类：

Spring 源码中提供了 RestTemplateBuilder 类，用于构建 RestTemplate 对象。RestTemplateBuilder 类实现了建造者模式，它提供了一系列方法用于设置 RestTemplate 的属性和特性，并最终构建出一个完整的 RestTemplate 对象。

示例代码：

下面是使用 RestTemplateBuilder 创建 RestTemplate 对象的示例代码：

```
RestTemplate restTemplate = new RestTemplateBuilder()
        .rootUri(”https://api.example.com”)
        .basicAuthentication(”username”, ”password”)
        .build();
```

在这个示例中，RestTemplateBuilder 提供了 rootUri()、basicAuthentication() 等方法用于设置 RestTemplate 的属性，最后调用 build() 方法构建出一个 RestTemplate 对象。

RestTemplateBuilder 类源码：

RestTemplateBuilder 类的源码如下所示：

```
public class RestTemplateBuilder {
    private String rootUri;
    private CredentialsProvider credentialsProvider;

    public RestTemplateBuilder rootUri(String rootUri) {
        this.rootUri = rootUri;
        return this;
    }

    public RestTemplateBuilder basicAuthentication(String username,
String password) {
        this.credentialsProvider = new BasicCredentialsProvider();
        this.credentialsProvider.setCredentials(AuthScope.ANY, new
UsernamePasswordCredentials(username, password));
        return this;
    }

    public RestTemplate build() {
        RestTemplate restTemplate = new RestTemplate();
        restTemplate.setUriTemplateHandler(new
DefaultUriBuilderFactory(this.rootUri));
        if (this.credentialsProvider != null) {
            restTemplate.setRequestFactory(new
HttpComponentsClientHttpRequestFactory(HttpClients.custom().setDefa
```

```
ultCredentialsProvider(this.credentialsProvider).build()));
        }
        return restTemplate;
    }
}
```

在 RestTemplateBuilder 类中，通过链式调用一系列方法来设置 RestTemplate 的属性，并在 build() 方法中构建出一个 RestTemplate 对象。在 build() 方法中，根据设置的属性创建 RestTemplate 对象，并设置相应的请求工厂、URI 模板处理器等属性。

通过使用建造者模式，Spring 实现了 RestTemplate 的创建过程，使得构建过程更加灵活，同时保证了 RestTemplate 对象的一致性和可复用性。

10、访问者模式（Visitor Pattern）

Spring 中的 BeanDefinitionVisitor 接口和其实现类是访问者模式的应用。它可以遍历访问容器中的所有 BeanDefinition 对象，并执行相应的操作。

Spring 源码中使用了访问者模式来实现 BeanDefinitionVisitor。BeanDefinitionVisitor 用于访问 BeanDefinition 树形结构中的每个节点，并执行特定的操作。下面是 Spring 源码中如何使用访问者模式实现 BeanDefinitionVisitor 的简要说明：

BeanDefinitionVisitor 接口：

Spring 提供了 BeanDefinitionVisitor 接口，用于定义访问者的行为。

```
public interface BeanDefinitionVisitor {
    void visitBeanDefinition(BeanDefinition beanDefinition);
}
```

BeanDefinitionVisitor 接口定义了 visitBeanDefinition() 方法，用于访问 BeanDefinition 树中的每个节点。

AbstractBeanDefinitionVisitor 抽象类：

Spring 提供了 AbstractBeanDefinitionVisitor 抽象类，实现了 BeanDefinitionVisitor 接口，并提供了默认的 visitBeanDefinition() 方法实现。在遍历 BeanDefinition 树时，AbstractBeanDefinitionVisitor 类会递归调用 visitBeanDefinition() 方法。

```
public abstract class AbstractBeanDefinitionVisitor implements
BeanDefinitionVisitor {
    public void visitBeanDefinition(BeanDefinition beanDefinition) {
        // 遍历 BeanDefinition 树，递归调用 visitBeanDefinition() 方法
        for (BeanDefinition bd : beanDefinition.getNestedBeanDefinitions())
{
            visitBeanDefinition(bd);
        }
    }
}
```

AbstractBeanDefinitionVisitor 类的主要作用是提供了默认的遍历逻辑，遍历 BeanDefinition 树并递归调用 visitBeanDefinition() 方法。

具体的访问者类：

在具体的应用场景中，开发者可以根据需要实现自定义的访问者类，并实现 visitBeanDefinition() 方法来执行特定的操作。例如，可以实现一个输出 BeanDefinition 信息的访问者类：

```
public class PrintingBeanDefinitionVisitor extends
AbstractBeanDefinitionVisitor {
    public void visitBeanDefinition(BeanDefinition beanDefinition) {
        System.out.println(”Bean name: ” +
beanDefinition.getBeanName());
        System.out.println(”Bean class: ” +
beanDefinition.getBeanClassName());
        // 输出其他 BeanDefinition 信息...
```

```
        super.visitBeanDefinition(beanDefinition);
    }
}
```

在这个示例中，PrintingBeanDefinitionVisitor 类继承了 AbstractBeanDefinitionVisitor 类，重写了 visitBeanDefinition() 方法，并在方法中输出了 BeanDefinition 的相关信息。

通过使用访问者模式，Spring 实现了 BeanDefinitionVisitor 接口的多种具体访问者类，每个访问者类负责执行不同的操作，使得开发者可以根据需要灵活地实现对 BeanDefinition 树的访问和操作。

11、责任链模式（Chain of Responsibility Pattern）

Spring Security 中的过滤器链就是责任链模式的一个应用。每个过滤器都可以选择是否处理当前的请求，如果处理，则将请求传递给下一个过滤器，如果不处理，则将请求返回给调用者。

在 Spring Security 源码中，过滤器链的实现主要使用了责任链模式。Spring Security 中的过滤器链负责处理 Web 请求，并按照一定的顺序依次调用不同的过滤器来完成安全认证、授权等操作。下面是 Spring Security 中如何使用责任链模式实现过滤器链的简要说明：

Filter 接口：

Spring Security 中的过滤器都实现了 Filter 接口，它定义了过滤器的基本行为。

```
public interface Filter {
    void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException;
}
```

Filter 接口中的 doFilter() 方法用于处理请求，并将请求传递给下一个过滤器或

目标资源。

FilterChain 接口：

Spring Security 中的过滤器链由 FilterChain 接口来管理，它定义了过滤器链的执行顺序。

```
public interface FilterChain {
    void doFilter(ServletRequest request, ServletResponse response)
throws IOException, ServletException;
}
```

FilterChain 接口中的 doFilter() 方法用于执行过滤器链中的下一个过滤器或目标资源。

责任链模式的实现：

在 Spring Security 中，过滤器链的实现使用了责任链模式，即将多个过滤器链接成一条链，每个过滤器依次处理请求，并将请求传递给下一个过滤器或目标资源。这样，每个过滤器只需要自己的业务逻辑，而不需要关心其他过滤器的存在。

FilterChainProxy 类：

在 Spring Security 中，FilterChainProxy 类是过滤器链的关键实现。它负责管理所有的过滤器链，并按照一定的顺序调用这些过滤器来处理请求。

```
public class FilterChainProxy extends GenericFilterBean {
    private List<SecurityFilterChain> filterChains;

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        // 遍历所有的过滤器链，按照顺序调用过滤器
        for (SecurityFilterChain chain : filterChains) {
```

```
            if (chain.matches(request)) {
                chain.doFilter(request, response);
                return;
            }
        }
        // 如果没有匹配的过滤器链，则调用默认的 FilterChain 继续处理请求
        chain.doFilter(request, response);
    }
}

```

在 FilterChainProxy 类中，通过遍历所有的过滤器链，按照顺序调用过滤器来处理请求。如果找到了匹配的过滤器链，则调用该过滤器链来处理请求；如果没有找到匹配的过滤器链，则调用默认的 FilterChain 继续处理请求。

Spring Security 源码中使用责任链模式实现了过滤器链，通过将多个过滤器链接成一条链，并按照一定的顺序调用这些过滤器来处理请求，实现了安全认证、授权等功能。责任链模式使得每个过滤器都可以独立地处理请求，并将请求传递给下一个过滤器或目标资源，从而提高了代码的灵活性和可维护性。

12、状态模式（State Pattern）

Spring 中的 Bean 生命周期管理就是状态模式的一个应用。Bean 可以处于不同的状态，如初始化中、可用、销毁中等，容器根据不同的状态执行相应的操作。

在 Spring 源码中，并没有显式地使用状态模式来实现 Bean 生命周期管理。但是，Spring 的 Bean 生命周期管理确实可以看作是一种状态机模式的实现。在 Spring 中，每个 Bean 都有其生命周期，包括初始化、使用和销毁等阶段，而 Spring 容器负责管理这些生命周期。虽然 Spring 没有定义一个专门的状态机类来管理 Bean 生命周期，但是它的生命周期管理机制确实可以视为一种状态机模式的实现。

以下是 Spring 如何管理 Bean 生命周期的简要说明：

Bean 生命周期阶段：

Spring 中的 Bean 生命周期包括多个阶段，主要包括实例化、属性赋值、初始化、使用和销毁等阶段。

Bean 生命周期回调方法：

在 Spring 中，Bean 的生命周期由一系列回调方法控制，这些回调方法可以由开发者自定义并在 Bean 的不同生命周期阶段被 Spring 容器调用。其中包括：

afterPropertiesSet() 方法：在所有属性设置完成之后，Spring 容器会调用该方法来执行一些初始化操作。

init-method 属性：可以在 XML 配置文件中使用 init-method 属性定义 Bean 的初始化方法。

destroy() 方法：在 Bean 被销毁之前，Spring 容器会调用该方法执行一些清理操作。

destroy-method 属性：可以在 XML 配置文件中使用 destroy-method 属性定义 Bean 的销毁方法。

Spring 容器的生命周期管理：

Spring 容器负责管理 Bean 的生命周期，包括实例化、初始化、使用和销毁等阶段。Spring 容器在初始化 Bean 时会调用相应的初始化方法，而在销毁 Bean 时会调用相应的销毁方法。

虽然 Spring 源码中并没有显式地使用状态模式来管理 Bean 生命周期，但是它的生命周期管理机制可以视为一种状态机模式的实现。在这个实现中，每个 Bean 都处于不同的生命周期阶段，并且在不同的阶段可能会有不同的行为，而 Spring 容器负责管理这些状态之间的转换和相应的行为。因此，尽管 Spring 源码中没有专门的状态模式的实现类，但是它的生命周期管理机制确实可以被视为一种状态机模式的实现。

13、迭代器模式（Iterator Pattern）
==========

Spring 中的 BeanDefinition 接口的实现类（如 RootBeanDefinition、GenericBeanDefinition 等）使用了迭代器模式，可以遍历访问 BeanDefinition 中的各个属性。

在 Spring 源码中，BeanDefinition 的集合通常是通过 List、Map 或其他集合类型来存储的，而对这些集合的遍历操作通常是通过迭代器模式来实现的。迭代器模式允许客户端代码以统一的方式访问集合中的元素，而无需了解底层集合的具体实现细节。下面是 Spring 源码中如何使用迭代器模式实现对 BeanDefinition 的遍历：

BeanDefinitionRegistry 接口：

Spring 中的 BeanDefinitionRegistry 接口定义了对 BeanDefinition 的注册和检索方法，如 registerBeanDefinition()、getBeanDefinition() 等。其中，BeanDefinitionRegistry 接口还提供了获取所有 BeanDefinition 的方法。

```
public interface BeanDefinitionRegistry {
    void registerBeanDefinition(String beanName, BeanDefinition beanDefinition);
    BeanDefinition getBeanDefinition(String beanName);
    boolean containsBeanDefinition(String beanName);
    int getBeanDefinitionCount();
    String[] getBeanDefinitionNames();
}
```

BeanDefinitionIterator 实现：

Spring 源码中通常会有一个 BeanDefinitionIterator 类来实现对 BeanDefinition 的迭代操作。这个类通常是一个迭代器模式的实现，它封装了对 BeanDefinition 集合的遍历逻辑，并提供了统一的访问接口。

```
public class BeanDefinitionIterator implements Iterator<BeanDefinition> {
    private final BeanDefinitionRegistry registry;
    private final String[] beanDefinitionNames;
    private int index = 0;

    public BeanDefinitionIterator(BeanDefinitionRegistry registry) {
        this.registry = registry;
        this.beanDefinitionNames = registry.getBeanDefinitionNames();
```

```
    }

    public boolean hasNext() {
        return index < beanDefinitionNames.length;
    }

    public BeanDefinition next() {
        String beanName = beanDefinitionNames[index];
        BeanDefinition beanDefinition =
registry.getBeanDefinition(beanName);
        index++;
        return beanDefinition;
    }

    public void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```

在 BeanDefinitionIterator 类中，通过实现 Iterator 接口，封装了对 BeanDefinition 集合的遍历逻辑，提供了 hasNext()、next() 等方法用于访问 BeanDefinition。

使用示例：

下面是使用 BeanDefinitionIterator 类遍历 BeanDefinition 集合的示例代码：

```
BeanDefinitionRegistry registry = ...; // 获取 BeanDefinitionRegistry 对象
BeanDefinitionIterator iterator = new BeanDefinitionIterator(registry);
while (iterator.hasNext()) {
    BeanDefinition beanDefinition = iterator.next();
    // 处理 BeanDefinition
}
```

在这个示例中，通过 BeanDefinitionIterator 类遍历了 BeanDefinition 集合，并执行了相应的操作。

通过使用迭代器模式，Spring 实现了对 BeanDefinition 集合的统一遍历操作

，使得客户端代码可以以统一的方式访问 BeanDefinition，而无需了解底层集合的具体实现细节。这种方式提高了代码的灵活性和可维护性。

14、命令模式（Command Pattern）

Spring MVC 中的 HandlerMapping 就是命令模式的一个应用。它将请求映射到相应的处理器（Controller），并且支持不同类型的映射策略。

在 Spring 源码中，HandlerMapping 的实现通常使用了命令模式（Command Pattern）。HandlerMapping 的作用是根据请求的 URL 映射到相应的处理器（Handler），而命令模式可以将请求的处理过程封装为一个命令对象，从而使得请求的处理过程与调用方解耦。下面是 Spring 源码中如何使用命令模式实现 HandlerMapping 的简要说明：

HandlerMapping 接口：

Spring 提供了 HandlerMapping 接口，用于定义请求的映射规则，并将请求映射到相应的处理器。

```
public interface HandlerMapping {
    HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception;
}
```

HandlerMapping 接口定义了一个 getHandler() 方法，用于根据请求获取相应的处理器。

HandlerExecutionChain 类：

Spring 中的 HandlerExecutionChain 类用于封装请求的处理器及其拦截器链。

```
```

```
public class HandlerExecutionChain {
    private Object handler;
    private List<HandlerInterceptor> interceptors;

    // Getter 和 Setter 方法...
}
```

HandlerExecutionChain 类包含了一个 handler 属性，用于存储请求的处理器，以及一个 interceptors 属性，用于存储拦截器链。

具体命令类：

Spring 中通常会有多个具体的命令类来实现不同的 HandlerMapping 策略。每个具体的命令类负责根据不同的映射规则将请求映射到相应的处理器。

```
public class SimpleUrlHandlerMapping implements HandlerMapping {
    private Map<String, Object> urlMap;

    public HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
        String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);
        Object handler = urlMap.get(lookupPath);
        if (handler == null) {
            return null;
        }
        HandlerExecutionChain chain = new HandlerExecutionChain(handler);
        chain.addInterceptors(getInterceptors(lookupPath, handler));
        return chain;
    }

    // 其他方法...
}
```

在这个示例中，SimpleUrlHandlerMapping 类实现了 HandlerMapping 接口，并根据 URL 映射规则将请求映射到相应的处理器。

调用者：

在 Spring 中，DispatcherServlet 充当了调用者的角色，它会根据请求选择合适的 HandlerMapping，并调用其 getHandler() 方法来获取处理器。

```
public class DispatcherServlet extends HttpServlet {
    private List<HandlerMapping> handlerMappings;

    protected void doDispatch(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        HandlerExecutionChain handler = null;
        for (HandlerMapping hm : this.handlerMappings) {
            handler = hm.getHandler(request);
            if (handler != null) {
                break;
            }
        }
        if (handler == null) {
            // 未找到合适的处理器
            return;
        }
        // 执行拦截器链和处理器
        handler.applyPreHandle(request, response);
        Object handlerObj = handler.getHandler();
        // 调用处理器...
        handler.applyPostHandle(request, response, mv);
        handler.triggerAfterCompletion(request, response, ex);
    }
}
```

DispatcherServlet 类在执行请求时会遍历所有的 HandlerMapping，并调用其 getHandler() 方法获取处理器。

通过使用命令模式，Spring 实现了 HandlerMapping 的抽象，将请求的处理过程封装为命令对象，使得请求的处理过程与调用方解耦。这种方式提高了代码的灵活性和可维护性。

15、解释器模式（Interpreter Pattern）

Spring EL（Expression Language）就是解释器模式的一个应用。Spring EL 可以解析和执行字符串表达式，支持动态求值和类型转换等功能。

在 Spring 源码中，Spring EL（表达式语言）的实现并没有直接使用解释器模式。但是，Spring EL 的实现可以被视为一种解释器模式的应用。Spring EL 是一个功能强大的表达式语言，它允许在运行时对对象进行查询和操作，并且可以嵌入到 Spring 框架的各个地方，如 XML 配置文件、注解、SpEL 注解等。下面简要介绍 Spring EL 的实现方式：

Spring EL 表达式解析：

Spring EL 表达式通常由一个表达式解析器（Expression Parser）来解析和执行。表达式解析器负责将表达式字符串解析成相应的表达式对象，并执行该表达式以获取结果。

SpEL（Spring Expression Language）：

在 Spring 中，Spring EL 的主要实现是 SpEL（Spring Expression Language）。SpEL 提供了一个 ExpressionParser 接口，用于解析表达式，并提供了一个 EvaluationContext 接口，用于执行表达式。

ExpressionParser 接口：

ExpressionParser 接口定义了将表达式字符串解析为表达式对象的方法。

```
public interface ExpressionParser {
    Expression parseExpression(String expressionString);
}
```

EvaluationContext 接口：

EvaluationContext 接口定义了执行表达式的方法，并提供了对变量、函数等上下文信息的访问。

```
public interface EvaluationContext {
    Object lookupVariable(String name);
    // 其他方法...
}
```

SpelExpressionParser 类：

Spring 中的 SpelExpressionParser 类实现了 ExpressionParser 接口，用于解析 SpEL 表达式。SpelExpressionParser 类将表达式字符串解析为 SpelExpression 对象，并执行该表达式以获取结果。

```
public class SpelExpressionParser implements ExpressionParser {
    public Expression parseExpression(String expressionString) {
        return new SpelExpression(expressionString);
    }
}
```

SpelExpression 类：

SpelExpression 类表示一个 SpEL 表达式，它封装了表达式字符串，并提供了执行表达式的方法。

```
public class SpelExpression implements Expression {
    private final String expressionString;

    public SpelExpression(String expressionString) {
        this.expressionString = expressionString;
    }

    public Object getValue(EvaluationContext context) throws
EvaluationException {
        // 执行表达式，并返回结果
    }
```

```
    // 其他方法...
}
```

通过使用 SpEL，Spring 实现了一个功能强大的表达式语言，它允许在运行时对对象进行查询和操作，并且可以嵌入到 Spring 框架的各个地方。虽然 Spring EL 的实现并没有直接使用解释器模式，但是它的执行过程可以被视为一种解释器模式的应用，即将表达式字符串解析为表达式对象，并执行该表达式以获取结果。

16、备忘录模式（Memento Pattern）

Spring 中的状态保存和恢复机制（如事务管理中的保存点）就是备忘录模式的一个应用。它可以保存对象的内部状态，并在需要时将对象恢复到之前的状态。

在 Spring 源码中，并没有显式地使用备忘录模式来实现状态保存和恢复机制。但是，Spring 中的一些功能确实可以被视为备忘录模式的一种应用，尤其是在 AOP（面向切面编程）和事务管理等方面。

在 AOP 中，Spring 使用切面（Aspect）来捕获方法调用，并在方法调用前后执行一些附加操作。这些附加操作通常包括保存方法调用前的状态，执行方法调用，然后根据需要恢复到之前的状态。尽管 Spring 源码中并没有专门的备忘录模式的实现类，但是 AOP 中的这种状态保存和恢复机制可以被视为一种备忘录模式的应用。

在事务管理中，Spring 使用事务切面来管理事务的开启、提交、回滚等操作。在进行事务管理时，Spring 会保存当前的数据库连接状态、事务状态等信息，并在事务执行完毕后根据需要进行恢复。虽然 Spring 源码中也没有专门的备忘录模式的实现类，但是事务管理中的这种状态保存和恢复机制也可以被视为备忘录模式的一种应用。

虽然 Spring 源码中并没有直接使用备忘录模式来实现状态保存和恢复机制，但是在 AOP 和事务管理等功能中，Spring 使用了类似备忘录模式的机制来保存和恢复对象的状态，从而提供了更加灵活和可靠的功能。

——好了，以上是威哥给大家整理了16种常见的设计模式在 Spring 源码中的运用，学习 Spring 源码成为了 Java 程序员的标配，你还知道Spring 中哪些

源码中运用了设计模式，欢迎留言与威哥交流。

原文链接: https://juejin.cn/post/7380994802744819749