Detailed explanation of design patterns in Spring

Guide frame Spring About 4339 words About 14 minutes

"What design patterns are used in JDK? What design patterns are used in Spring?" These two questions are common in interviews.

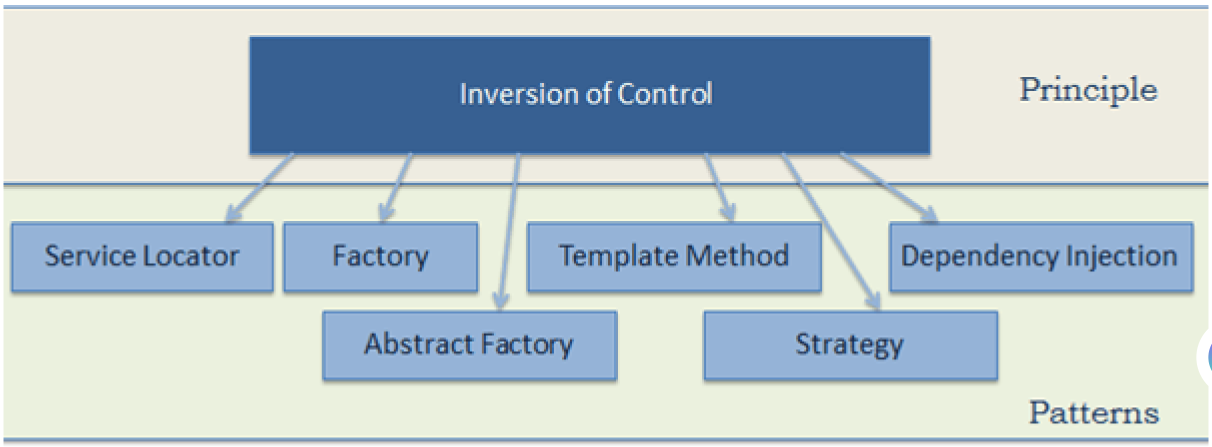
I searched online for explanations of Spring design patterns, but found that they were all the same, and most of them were outdated. So, I spent a few days summarizing them myself.

Due to my limited ability, please feel free to point out any errors in this article. In addition, due to the limited length of the article, I will only briefly touch upon the design patterns and some source code interpretation. The main purpose of this article is to review the design patterns in Spring.

Inversion of Control (IoC) and Dependency Injection (DI)

IoC (Inversion of Control) is a crucial concept in Spring. It's not a technology, but rather a decoupling design philosophy. Its primary purpose is to decouple dependent objects by leveraging a "third party" (Spring's IoC container)—the IoC container manages the objects, leaving you to use them—thus reducing the degree of coupling between code.

IoC is a principle, not a pattern. The following patterns (but not limited to) implement the IoC principle.



The Spring IoC container is like a factory. When we need to create an object, we only need to configure the configuration file/annotation, without having to worry about how the object is created. The IoC container is responsible for creating objects, wiring them together, configuring them, and handling the entire life cycle of these objects from creation to their complete destruction.

In a real-world project, if a Service class has hundreds or even thousands of underlying classes, and you need to instantiate the Service, you might have to figure out the constructors of all the underlying classes each time, which can be frustrating. With IoC, you only need to configure it and reference it where needed, which greatly increases project maintainability and reduces development effort.

> Regarding the understanding of Spring IOC, I recommend reading this answer on Zhihu: https://www.zhihu.com/question/23277575/answer/169698662 , which is very good.

How do you understand inversion of control? For example: "Object a depends on object b. When object a needs to use object b, it must create it itself. But when the system introduces an IOC container, there is no direct connection between object a and object b. At this time, when object a needs to use object b, we can specify the IOC container to create an object b and inject it into object a." The process of object a obtaining dependency on object b changes from active behavior to passive behavior, and control is reversed. This is the origin of the name inversion of control.

DI (Dependency Inject) is a design pattern for implementing inversion of control. Dependency injection is the process of passing instance variables into an object.

Factory Design Pattern

Spring uses the factory pattern to create bean objects through `BeanFactory` or . `ApplicationContext`

Comparison between the two:

- `BeanFactory` : Delayed injection (injection only when a bean is used) `ApplicationContext` takes up less memory and the program starts faster than .
- `ApplicationContext` : When the container starts, all beans are created at once, regardless of whether you need them or not. `BeanFactory` It only provides the most basic dependency injection support. `ApplicationContext` It has been extended

`BeanFactory` to provide additional features in addition to `BeanFactory` some existing features, so `ApplicationContext` it is generally used by developers.

`ApplicationContext` Three implementation classes:

1. `ClassPathXmlApplication` : Treat context files as classpath resources.
2. `FileSystemXmlApplication` : Load context definition information from an XML file in the file system.
3. `XmlWebApplicationContext` : Load context definition information from an XML file in the Web system.

Example:

```java
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationContext;

public class App {
  public static void main(String[] args) {
    ApplicationContext context = new FileSystemXmlApplicationContext(
        "C:/work/IOC
Containers/springframework.applicationcontext/src/main/resources/bear
factory-config.xml");

    HelloApplicationContext obj = (HelloApplicationContext)
context.getBean("helloApplicationContext");
    obj.getMsg();
  }
}
```

Singleton Design Pattern

In our system, there are some objects that we only need one of, for example: thread pools, caches, dialog boxes, registry keys, log objects, objects that act as drivers for printers, graphics cards, etc. In fact, there can only be one instance of such objects. Creating multiple instances may cause problems such as abnormal program behavior, excessive resource usage, or inconsistent results.

Benefits of using the singleton pattern :

- For frequently used objects, the time spent on object creation can be omitted, which is a considerable system overhead for those heavyweight objects.
- Since the number of new operations is reduced, the frequency of system memory usage will also be reduced, which will reduce GC pressure and shorten GC pause time.

The default scope of a bean in Spring is singleton. In addition to the singleton scope, Spring also has the following scopes for beans:

- **prototype** : A new bean instance is created each time it is retrieved. That is, `getBean()` two consecutive Bean instances are obtained.
- **request** (only available for Web applications): Each HTTP request will generate a new bean (request bean), which is only valid within the current HTTP request.
- **session** (only available for Web applications): Each HTTP request from a new session will generate a new bean (session bean), which is only valid within the current HTTP session.
- **application/global-session** (only available for Web applications): Each Web application creates a bean (application bean) when it starts. This bean is valid only during the startup time of the current application.
- **websocket** (only available for Web applications): A new bean is created for each WebSocket session.

Spring `ConcurrentHashMap` implements the singleton pattern in a special way by implementing a singleton registry.

The core code for Spring to implement singleton is as follows:

```java
//      ConcurrentHashMap
private final Map<String, Object> singletonObjects = new
ConcurrentHashMap<String, Object>(64);

public Object getSingleton(String beanName, ObjectFactory<?>
singletonFactory) {
        Assert.notNull(beanName, "'beanName' must not be null");
        synchronized (this.singletonObjects) {
            //
            Object singletonObject =
this.singletonObjects.get(beanName);
            if (singletonObject == null) {
                //...
                try {
                    singletonObject = singletonFactory.getObject();
                }
                //...
                //
                addSingleton(beanName, singletonObject);
            }
            return (singletonObject != NULL_OBJECT ?
singletonObject : null);
        }
    }
    //
    protected void addSingleton(String beanName, Object
singletonObject) {
            synchronized (this.singletonObjects) {
                this.singletonObjects.put(beanName,
(singletonObject != null ? singletonObject : NULL_OBJECT));

            }
        }
}
```

Does Singleton Bean have thread safety issues?

Most of the time we don't use multithreading in our projects, so few people pay attention to this issue. Singleton beans have threading issues, mainly because when multiple threads operate on the same object, there is resource competition.

There are two common solutions:

1. Try to avoid defining mutable member variables in Bean.
2. Define a member variable in the class `ThreadLocal` and save the required mutable member variables in `ThreadLocal` it (a recommended method).

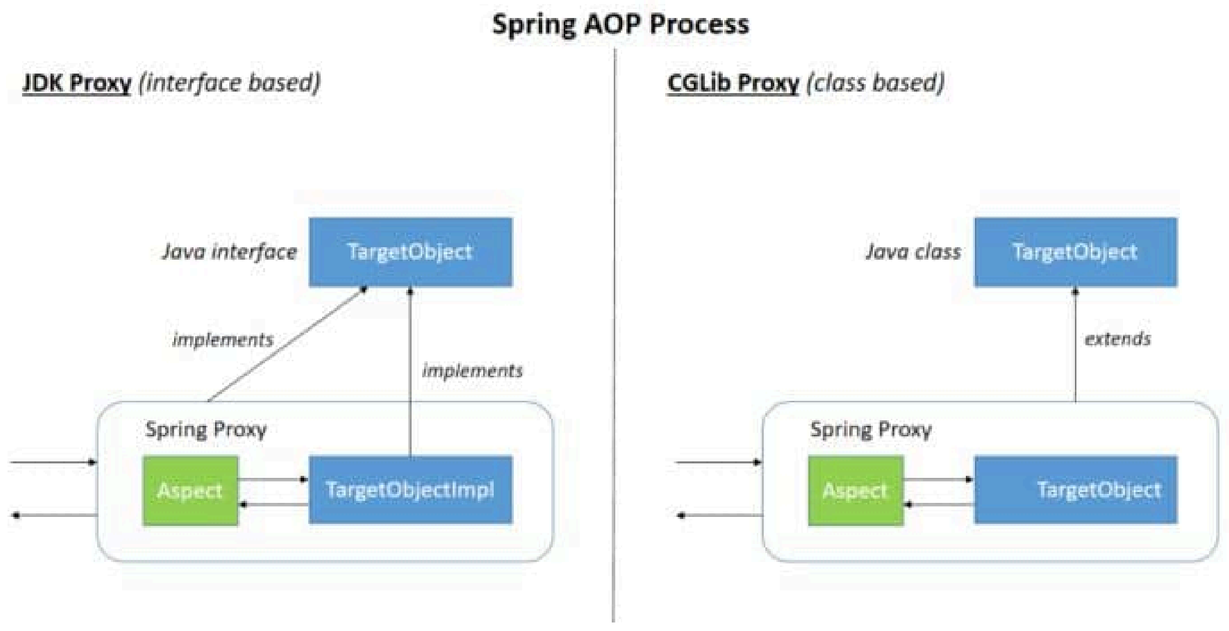
However, most beans are actually stateless (no instance variables) (such as Dao, Service). In this case, beans are thread-safe.

Proxy Design Pattern

Application of proxy mode in AOP

AOP (Aspect-Oriented Programming) can encapsulate logic or responsibilities are not related to the business but are commonly called by business modules (such as transaction processing, log management, permission control, etc.), thereby reducing duplicate code in the system, reducing the coupling between modules, and facilitating future scalability and maintainability.

Spring AOP is based on dynamic proxy. If the object to be proxied implements an interface, Spring AOP will use **JDK Proxy** to create a proxy object. For objects that do not implement the interface, JDK Proxy cannot be used for proxying. At this time, Spring AOP will use **Cglib** to generate a subclass of the proxied object as a proxy, as shown in the following figure:



Of course, you can also use AspectJ. Spring AOP has integrated AspectJ, which is probably the most complete AOP framework in the Java ecosystem.

By using AOP, we can abstract common functionality and use it directly where needed, significantly simplifying the code. This also makes it easier to add new features, improving system scalability. AOP is used in scenarios such as logging and transaction management.

What is the difference between Spring AOP and AspectJ AOP?

Spring AOP is a runtime enhancement, while AspectJ is a compile-time enhancement. Spring AOP is based on proxying, while AspectJ is based on bytecode manipulation.

Spring AOP has integrated AspectJ, which is arguably the most complete AOP framew in the Java ecosystem. AspectJ is more powerful than Spring AOP, but Spring AOP is relatively simpler.

If we have relatively few aspects, then the performance difference between the two is not significant. However, when there are too many aspects, it is better to choose AspectJ, which is much faster than Spring AOP.

Template Method

The Template Method pattern is a behavioral design pattern that defines the skeleton of an algorithm in an operation and defers some steps to subclasses. The Template Method allows subclasses to redefine the implementation of certain specific steps of an algorithm without changing its structure.

```java
public abstract class Template {
    //
    public final void TemplateMethod(){
        PrimitiveOperation1();
        PrimitiveOperation2();
        PrimitiveOperation3();
    }

    protected void  PrimitiveOperation1(){
        //
    }

    //
    protected abstract void PrimitiveOperation2();
    protected abstract void PrimitiveOperation3();

}
public class TemplateImpl extends Template {

    @Override
    public void PrimitiveOperation2() {
        //
    }

    @Override
    public void PrimitiveOperation3() {
        //
    }
}
```

In Spring `JdbcTemplate` , `HibernateTemplate` classes that end with "Template" and other database operations use the Template pattern. Generally, we use inheritance to implement the Template pattern, but Spring does not use this approach. Instead, it uses the Callback pattern in conjunction with the Template Method pattern, achieving code reuse while increasing flexibility.

Observer Pattern

The Observer pattern is an object-behavioral pattern. It represents a dependency relationship between objects. When one object changes, all dependent objects react accordingly. The Spring event-driven model is a classic application of the Observer pattern. The Spring event-driven model is very useful and can decouple code in many scenarios. For example, if we need to update the product index every time we add a product, the Observer pattern can be used to solve this problem.

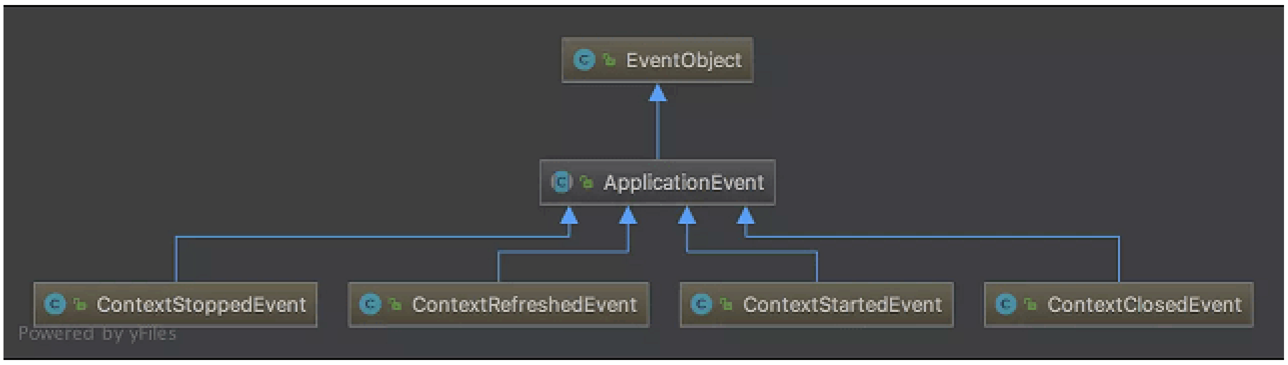
Three roles in Spring's event-driven model

Event Role

`ApplicationEvent` (`org.springframework.context` Under the package) plays the role of event, this is an abstract class that inherits `java.util.EventObject` and implements `java.io.Serializable` the interface.

The following events exist by default in Spring, they are all `ApplicationContextEvent` implementations of (inherited from `ApplicationContextEvent`):

- `ContextStartedEvent` : `ApplicationContext` Events triggered after startup;
- `ContextStoppedEvent` : `ApplicationContext` Event triggered after stopping;
- `ContextRefreshedEvent` : `ApplicationContext` Event triggered after initialization or refresh is completed;
- `ContextClosedEvent` : `ApplicationContext` Event triggered after closing.



Event Listener Role

`ApplicationListener` It plays the role of event listener. It is an interface that defines only one `onApplicationEvent()` method for handling `ApplicationEvent` . `ApplicationListener` The source code of the interface class is as follows. It can be seen that the interface definition shows that the events in the interface only need to be implemented `ApplicationEvent` . Therefore, in Spring, we only need to implement `ApplicationListener` the interface `onApplicationEvent()` method to complete the listening event.

```java
package org.springframework.context;
import java.util.EventListener;
@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent>
extends EventListener {
    void onApplicationEvent(E var1);
}
```

Event Publisher Role

`ApplicationEventPublisher` Acts as an event publisher and is also an interface.

```java
@FunctionalInterface
public interface ApplicationEventPublisher {
    default void publishEvent(ApplicationEvent event) {
        this.publishEvent((Object)event);
    }

    void publishEvent(Object var1);
}
```

`ApplicationEventPublisher` `publishEvent()` This method of the interface `AbstractApplicationContext` is implemented in the class. Reading the implementation of this method, you will find that the event is actually `ApplicationEventMulticaster` broadcasted through. There are too many details to analyze here. I may write a separate article to mention it later.

Summary of Spring's event flow

1. Define an event: implement an inheritance `ApplicationEvent` and write the corresponding constructor;
2. Define an event listener: implement `ApplicationListener` the interface and override `onApplicationEvent()` the method;
3. Use event publisher to publish messages: You can publish messages through the method `ApplicationEventPublisher` of . `publishEvent()`

Example:

```java
//            ,     ApplicationEvent
public class DemoEvent extends ApplicationEvent{
    private static final long serialVersionUID = 1L;

    private String message;

    public DemoEvent(Object source,String message){
        super(source);
        this.message = message;
    }

    public String getMessage() {
         return message;
          }


//                 ,    ApplicationListener
onApplicationEvent()
@Component
public class DemoListener implements ApplicationListener<DemoEvent>
{

    //    onApplicationEvent
    @Override
    public void onApplicationEvent(DemoEvent event) {
        String msg = event.getMessage();
        System.out.println("              "+msg);
    }

}
//                 ApplicationEventPublisher    publishEvent()

@Component
public class DemoPublisher {

    @Autowired
    ApplicationContext applicationContext;

    public void publish(String message){
        //
        applicationContext.publishEvent(new DemoEvent(this, message));
    }
}
```

`DemoPublisher` When the method of is called `publish()` , for example `demoPublisher.publish("    ")` , the console will print: .

Adapter Pattern

Adapter Pattern converts one interface into another interface that the client expects. The adapter pattern enables classes with incompatible interfaces to work together.

Adapter pattern in Spring AOP

We know that the implementation of Spring AOP is based on the proxy mode, but the enhancement or advice of Spring AOP uses the adapter mode, and the related interface is `AdvisorAdapter` .

Common types of Advice include: `BeforeAdvice` (before the target method is called, pre-advice), `AfterAdvice` (after the target method is called, post-advice), `AfterReturningAdvice` (after the target method is executed, before return), etc. Each type of Advice has a corresponding interceptor: `MethodBeforeAdviceInterceptor` , `AfterReturningAdviceInterceptor` , `ThrowsAdviceInterceptor` etc.

Spring's predefined notifications must be adapted to `MethodInterceptor` objects of the interface (method interceptor) type through the corresponding adapter (for example, `MethodBeforeAdviceAdapter` by calling `getInterceptor` the method, `MethodBeforeAdvice` adapting to `MethodBeforeAdviceInterceptor`).

Adapter Pattern in Spring MVC

In Spring MVC, the request is invoked and parsed `DispatcherServlet` based on the request information . Once the corresponding controller is found , the adapter begins processing. The adapter implementation class is used to adapt the target class, acting as the target
class. `HandlerMapping` `Handler` `Handler` `Controller` `HandlerAdapter` `HandlerAdapter` `Controller`

Why use the Adapter pattern in Spring MVC?

`Controller` There are many types of Spring MVC , and different types `Controller` handle requests in different ways. If you don't use the adapter pattern, `DispatcherServlet` you can directly obtain the corresponding type `Controller` and make your own judgment, as shown in the following code:

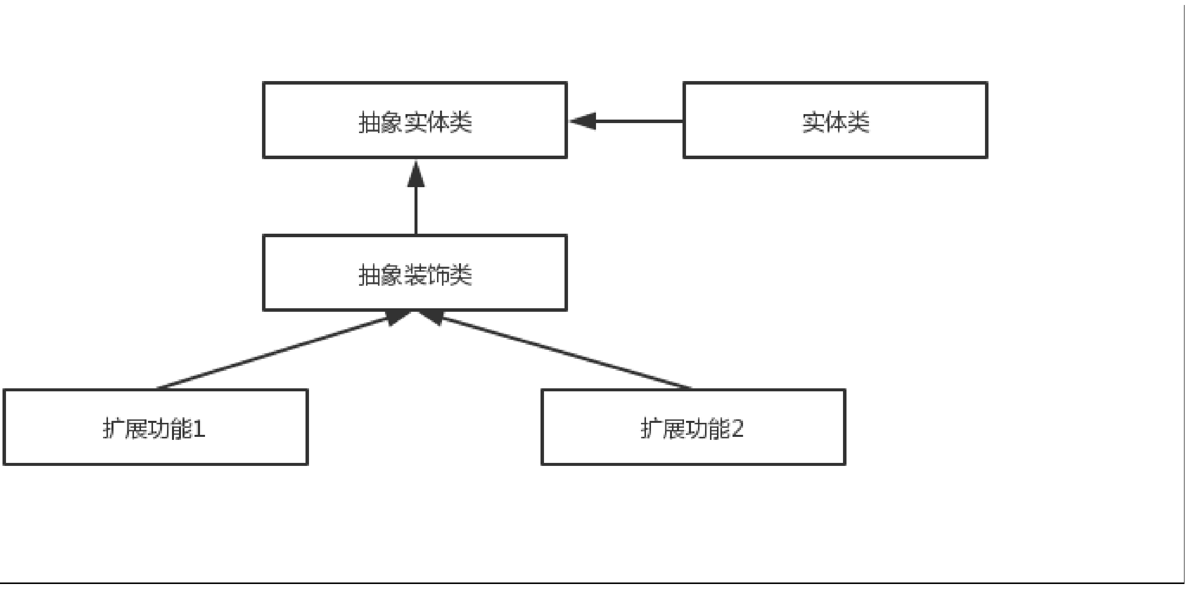
```java
if(mappedHandler.getHandler() instanceof MultiActionController){
   ((MultiActionController)mappedHandler.getHandler()).xxx
}else if(mappedHandler.getHandler() instanceof XXX){
    ...
}else if(...){
   ...
}
```

If we add another `Controller` type, we have to add another line of judgment statement to the above code. This form makes the program difficult to maintain and violates the open-closed principle in the design pattern - open for extension and closed for modification.

Decorator pattern

The Decorator pattern allows you to dynamically add additional attributes or behaviors to an object. Compared to inheritance, the Decorator pattern is more flexible. Simply pu when we need to modify existing functionality but don't want to modify the original code directly, we can design a Decorator to wrap around the original code. The Decorator pattern is used extensively in the JDK, such as `InputStream` in families . Subclasses

`InputStream` within a class, such as `FileInputStream` (reading files) and (adding cache to significantly speed up file reading) , extend functionality without modifying the code. `BufferedInputStream` `InputStream`



When configuring a DataSource in Spring, the DataSource may refer to different databases and data sources. Can we dynamically switch between different data sources based on customer needs while minimizing code changes to the original class? This is where the decorator pattern comes in (I don't quite understand the specific principles behind this). The wrapper pattern used in Spring contains "<pre>" `Wrapper` or " <pre>" in the class name `Decorator` . These classes essentially dynamically add additional responsibilities to an object.

Summarize

What design patterns are used in Spring Framework?

- **Factory Design Pattern** : Spring uses factory pattern to `BeanFactory` create `ApplicationContext` bean objects.
- **Proxy design pattern** : implementation of Spring AOP functionality.
- **Singleton design pattern** : Beans in Spring are singletons by default.
- **Template method pattern** : In Spring `jdbcTemplate` , `hibernateTemplate` classes that end with Template and other classes that operate on the database use the template pattern.
- **Wrapper Design Pattern** : Our project needs to connect to multiple databases, different customers may access different databases as needed during each visit. This pattern allows us to dynamically switch between different data sources based on customer needs.

- **Observer pattern:** Spring event-driven model is a classic application of the observer pattern.
- **Adapter pattern** : Spring AOP enhancement or advice uses the adapter pattern, and spring MVC also uses the adapter pattern adaptation `Controller` .
- ...

refer to

- Spring Technology Insider
- https://blog.eduonix.com/java-programming-2/learn-design-patterns-used-spring-framework/
- https://www.tutorialsteacher.com/ioc/inversion-of-control
- https://design-patterns.readthedocs.io/zh_CN/latest/behavioral_patterns/observer.html
- https://juejin.im/post/5a8eb261f265da4e9e307230
- https://juejin.im/post/5ba28986f265da0abc2b6084

Recently Updated2025/7/27 10:31

Contributors: SnailClimb , Kou Shuang , wjch , ipofss , guide , Jarvan-Song , wangtong , kuluqi163 , Guide , Erzbir , Mr.Hope , smy1999