

Summary of common Spring interview questions

 [Guide](#)  frame  Spring  About 13,470 words  About 45 minutes

This is a small advertisement that may be useful to you

- **Interview special edition** : Friends who are preparing for Java interviews can consider the interview special edition: "[Java Interview Guide](#)" (very high quality, specially designed for interviews, best used with JavaGuide).
- **Knowledge Planet** : Technical Column/One-on-one Questions/Resume Modification/Job Hunting Guide/Interview Check-in/Irregular Benefits, welcome to join [the JavaGuide official Knowledge Planet](#) .

This article mainly aims to deepen your understanding of Spring through some questions, so it will not involve too much code!

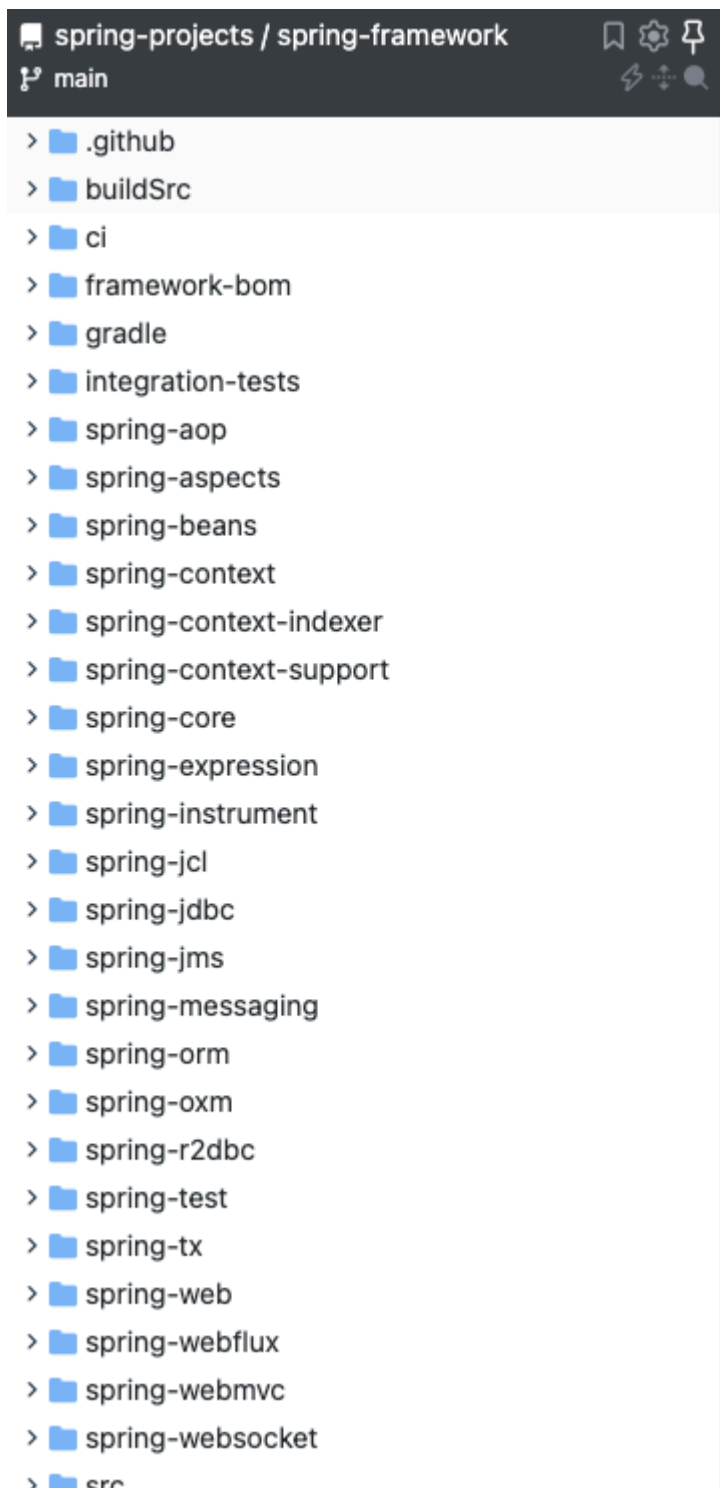
Many of the following questions were overlooked by me while using Spring, and I only addressed them by consulting numerous resources and books. There are also numerous articles online that summarize frequently asked Spring questions and interviews, but I feel like most of them are copied from one another, and many of the questions aren't very good, and some of the answers are flawed. So, I've spent a week of my spare time compiling these questions, hoping they'll be helpful.

Spring Basics

What is Spring Framework?

Spring is an open source lightweight Java development framework designed to improve developer efficiency and system maintainability.

When we say Spring framework, we generally mean Spring Framework, which is a collection of modules that can easily assist us in development. For example, Spring supports IoC (Inversion of Control) and AOP (Aspect-Oriented Programming), can easily access databases, can easily integrate third-party components (email, tasks, scheduling, caching, etc.), has good support for unit testing, and supports the development of RESTful Java applications.



The core idea of Spring is not to reinvent the wheel, to use it out of the box and improve development efficiency.

Spring translates to "spring," and its goal and mission is to bring spring to Java programmers! So touching!

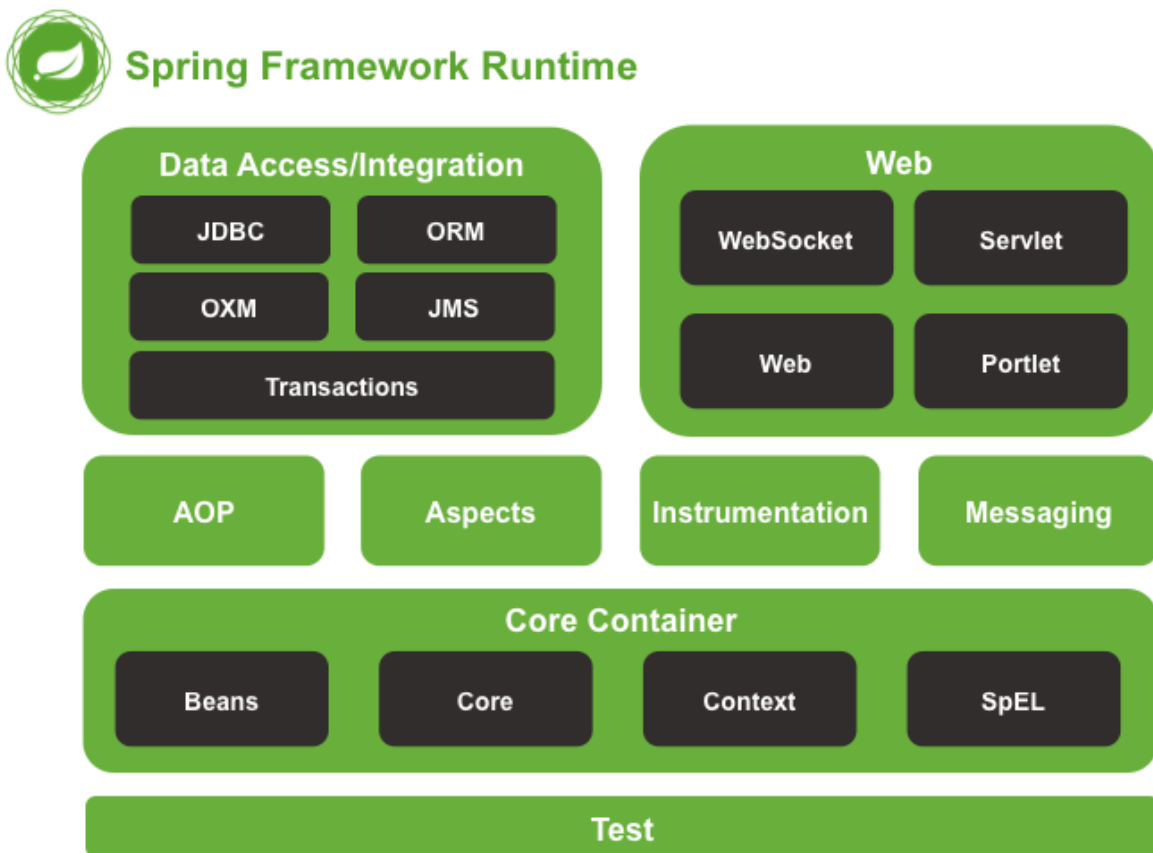
🤖 One more thing to mention: **The popularity of a language usually requires a killer application, and Spring is a killer application framework for the Java ecosystem.**

The core functions provided by Spring are mainly IoC and AOP. When learning Spring, you must understand the core ideas of IoC and AOP!

- Spring official website: <https://spring.io/>
- GitHub address: <https://github.com/spring-projects/spring-framework>

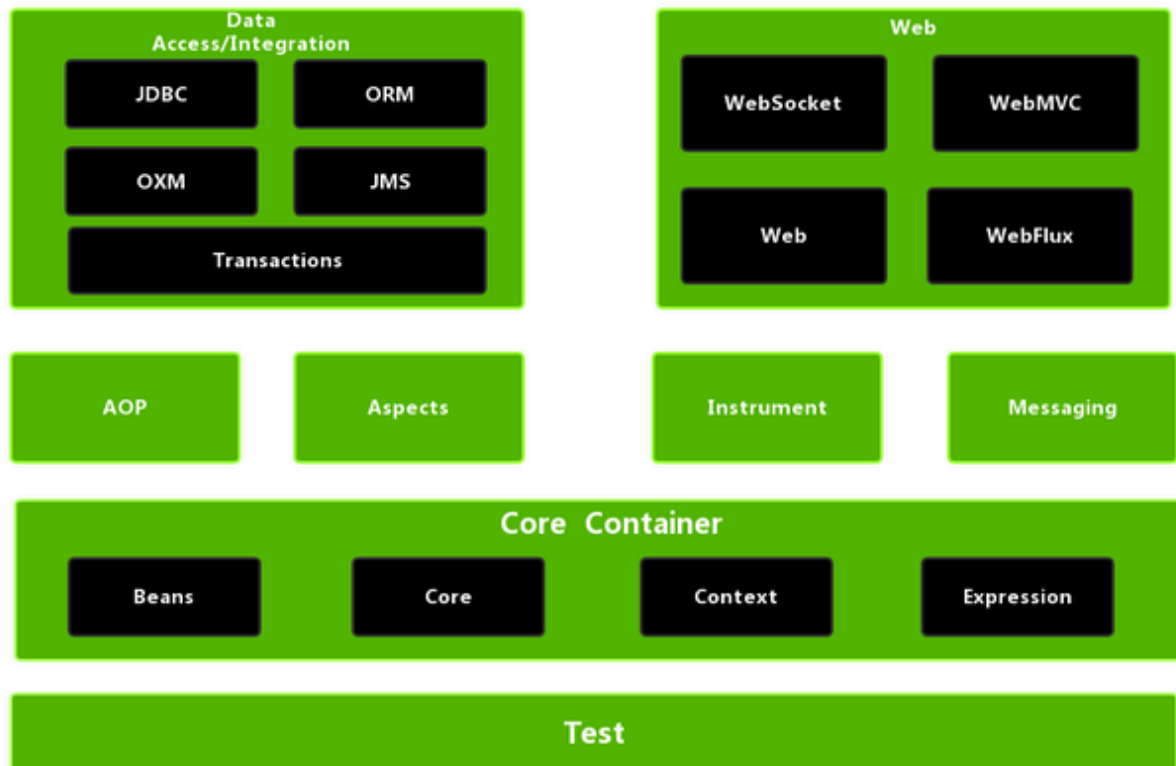
What modules does Spring include?

Spring 4.x version :



Spring 5.x version :

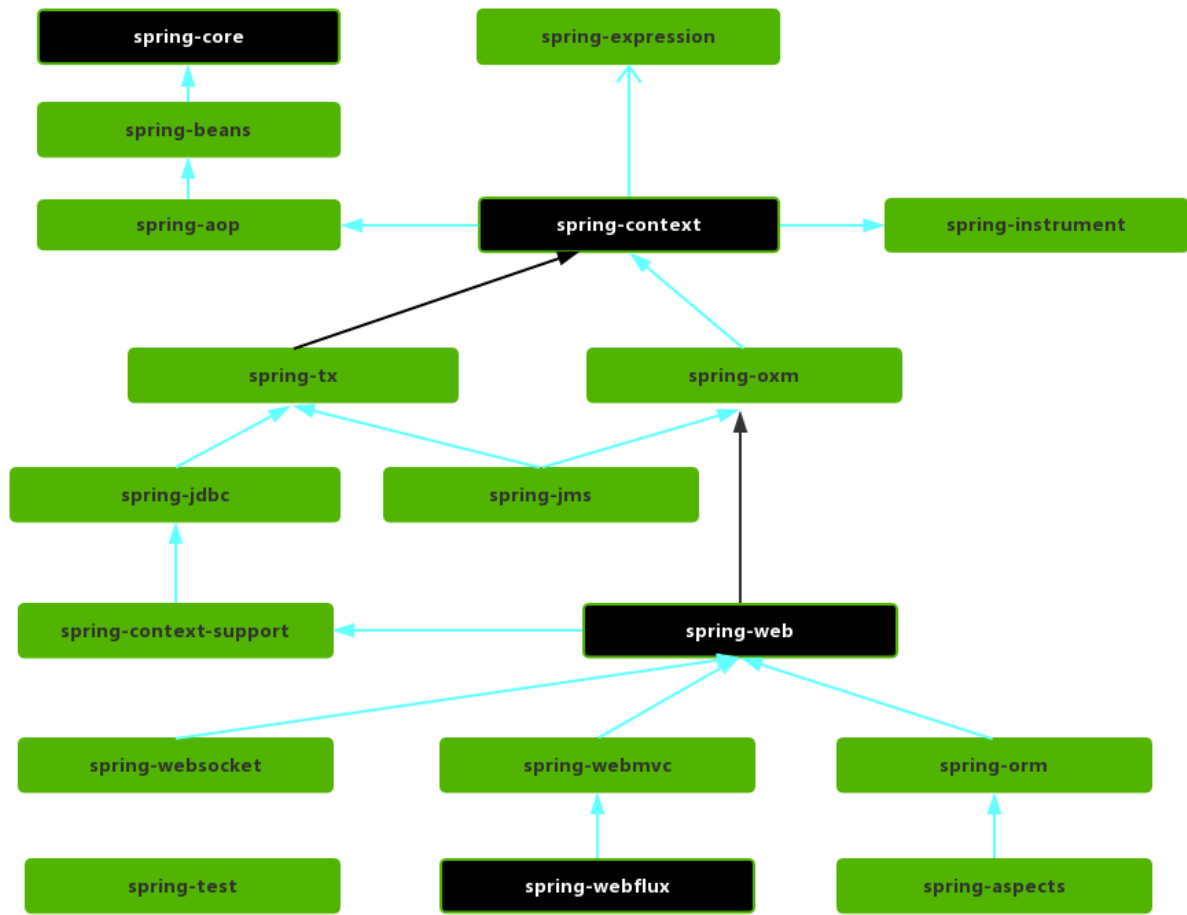




The Portlet component of the Web module in Spring 5.x has been deprecated, and the WebFlux component for asynchronous responsive processing has been added.

The dependencies of Spring modules are as follows:





Core Container

The core module of the Spring framework, or the foundational module, primarily provides support for IoC dependency injection. All other Spring features rely on this module, as can be seen in the dependency diagram of the various Spring modules above.

- **spring-core** : The basic core tool class of the Spring framework.
- **spring-beans** : Provides support for bean creation, configuration, and management.
- **spring-context** : Provides support for internationalization, event propagation, resource loading and other functions.
- **spring-expression** : Provides support for Spring Expression Language (SpEL). It depends only on the core module and does not depend on other modules. It can be used alone.

AOP



- **spring-aspects** : This module provides support for integration with AspectJ.

- **spring-aop** : provides aspect-oriented programming implementation.
- **spring-instrument** : Provides the ability to add an agent to the JVM. Specifically, it provides a weaving agent for Tomcat, which delivers class files to Tomcat as if they were loaded by a class loader. Don't worry if you don't understand it; this module has very limited use cases.

Data Access/Integration

- **spring-jdbc** : Provides an abstract JDBC framework for database access. Different databases have their own APIs for operating them, and Java programs only need to interact with the JDBC API, thus shielding them from the database's influence.
- **spring-tx** : Provides support for transactions.
- **spring-orm** : Provides support for ORM frameworks such as Hibernate, JPA, and iBatis.
- **spring-oxm** : Provides an abstraction layer to support OXM (Object-to-XML-Mapping), such as JAXB, Castor, XMLBeans, JiBX and XStream.
- **spring-jms** : Message service. Since Spring Framework 4.1, it also provides inheritance of the spring-messaging module.

Spring Web

- **spring-web** : Provides some basic support for the implementation of Web functions.
- **spring-webmvc** : Provides implementation of Spring MVC.
- **spring-websocket** : Provides support for WebSocket, which enables two-way communication between the client and the server.
- **spring-webflux** : Provides support for WebFlux, a new reactive framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the Servlet API and is fully asynchronous.

Messaging

spring-messaging is a new module added since Spring 4.0. Its main responsibility is to integrate some basic message transmission applications into the Spring framework.

Spring Test

The Spring team advocates test-driven development (TDD). With the help of Inversion of Control (IoC), unit testing and integration testing become easier.



Spring's test module has good support for commonly used test frameworks such as JUnit (unit testing framework), TestNG (similar to JUnit), Mockito (mainly used for mock objects), PowerMock (solves problems with Mockito such as the inability to simulate final, static, and private methods).

What is the relationship between Spring, Spring MVC, and Spring Boot?

Many people can't tell the difference between Spring, Spring MVC, and Spring Boot! Here's a brief introduction to them. It's actually very simple and there's nothing too advanced about them.

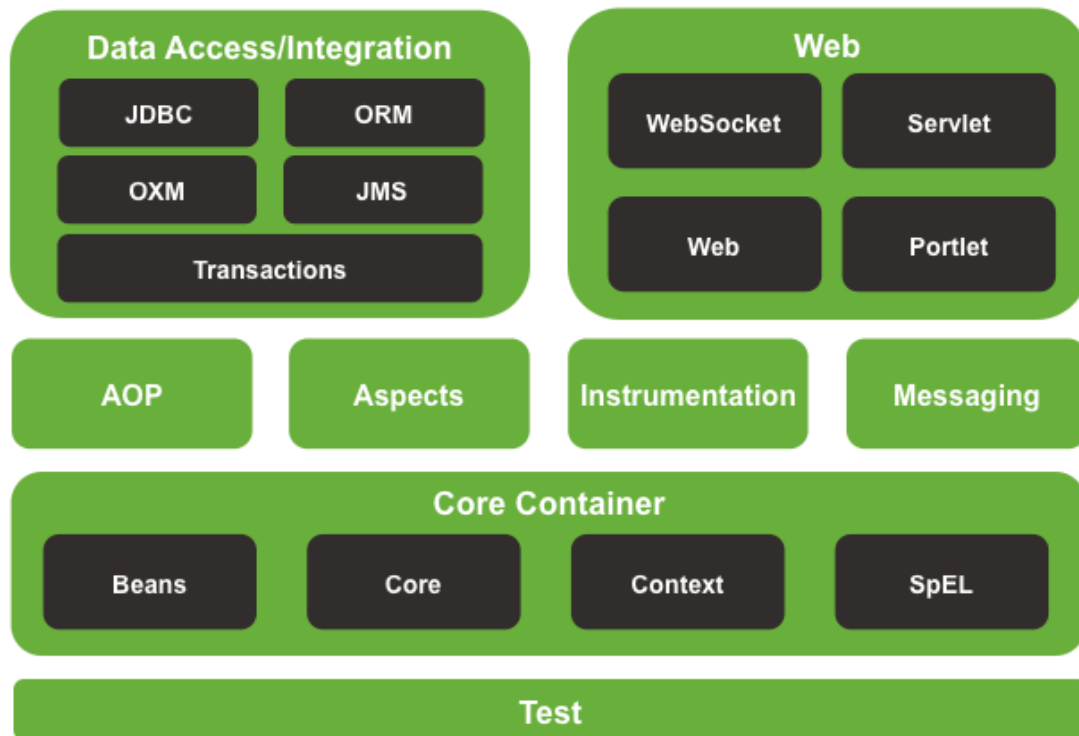
Spring includes multiple functional modules (mentioned above), the most important of which is the Spring-Core module (which mainly provides support for IoC dependency injection functions). The functional implementation of other modules in Spring (such as Spring MVC) basically depends on this module.

The following figure corresponds to Spring 4.x. In the latest 5.x version, the Portlet component of the Web module has been deprecated, and the WebFlux component for asynchronous responsive processing has been added.

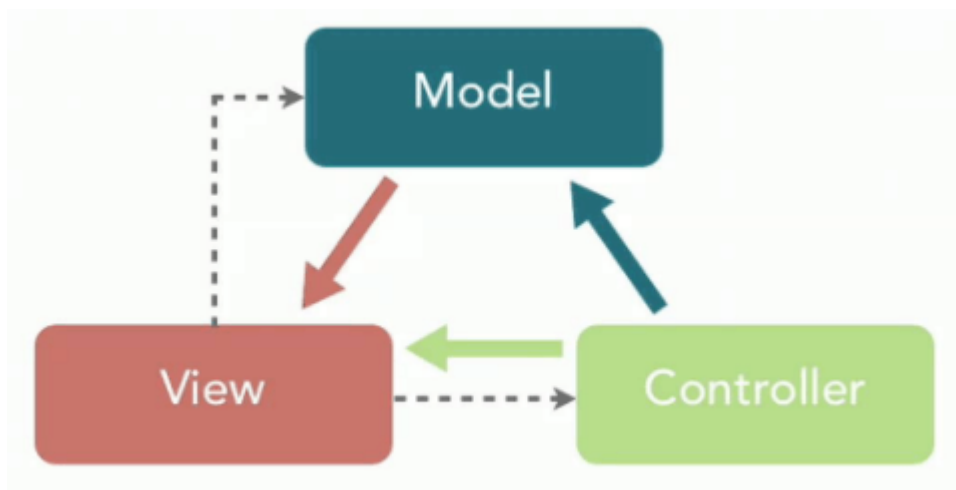




Spring Framework Runtime



Spring MVC is a crucial Spring module that enables you to quickly build web applications using the MVC architecture. MVC stands for Model, View, and Controller, and its core concept is to organize code by separating business logic, data, and display.



Developing with Spring was cumbersome due to various configuration issues. For example, enabling certain Spring features required explicit configuration using XML or Java. 1 Spring Boot was born!

Spring aims to simplify J2EE enterprise application development. Spring Boot aims to simplify Spring development (less configuration files, works out of the box!).

Spring Boot only simplifies the configuration. If you need to build a web application with MVC architecture, you still need to use Spring MVC as the MVC framework. It's just that Spring Boot simplifies many configurations of Spring MVC for you, making it truly out-of-the-box!

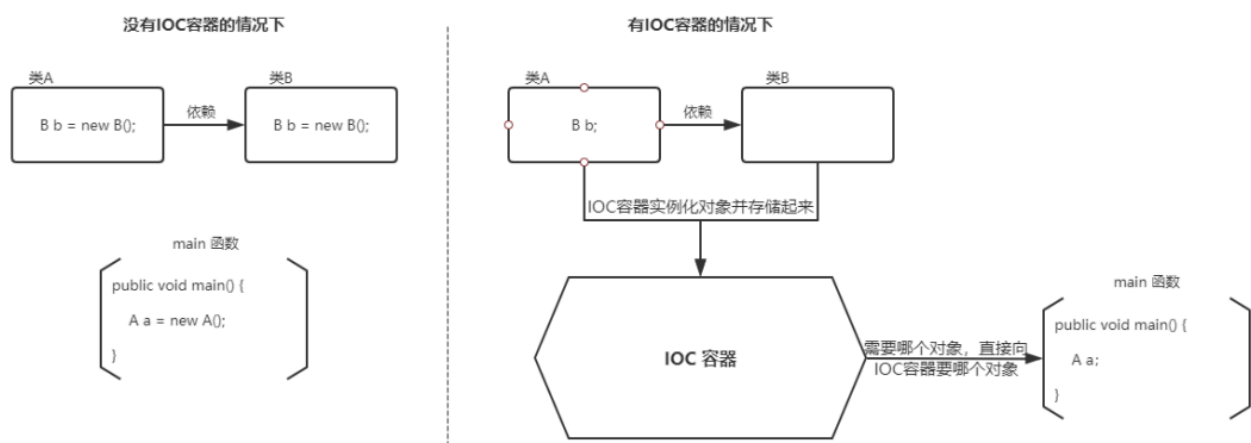
Spring IoC

Talk about your understanding of Spring IoC

IoC (Inversion of Control) is a design concept, not a specific technical implementation. The idea behind IoC is to transfer control over object creation, which would normally be manually created in a program, to the Spring framework. However, IoC isn't unique to Spring and is also used in other languages.

Why is it called Inversion of Control?

- **Control** : refers to the power to create (instantiate, manage) objects
- **Inversion** : Control is handed over to the external environment (Spring framework, IoC container)



The interdependencies between objects are managed by the IoC container, which then injects the objects. This greatly simplifies application development and frees applications from complex dependencies. The IoC container acts like a factory; when we need to create an object, we simply configure the configuration file/annotation without having to worry about how the object is created.

In a real-world project, a Service class might depend on many other classes. If you need to instantiate this service, you might have to figure out the constructors of all its underlying classes each time, which can be frustrating. With IoC, you only need to configure it and reference it where needed, greatly improving project maintainability and reducing development effort.

In Spring, the IoC container is the carrier used by Spring to implement IoC. The IoC container is actually a Map (key, value) that stores various objects.

In the Spring era, we generally configured Beans through XML files. Later, developers felt that XML files were not a good way to configure, so SpringBoot annotation configuration gradually became popular.

Related reading:

- [IoC source code reading](#)
- [IoC & AOP Explained \(Quickly Understand\)](#)

What are Spring Beans?

Simply put, Bean refers to objects managed by the IoC container.

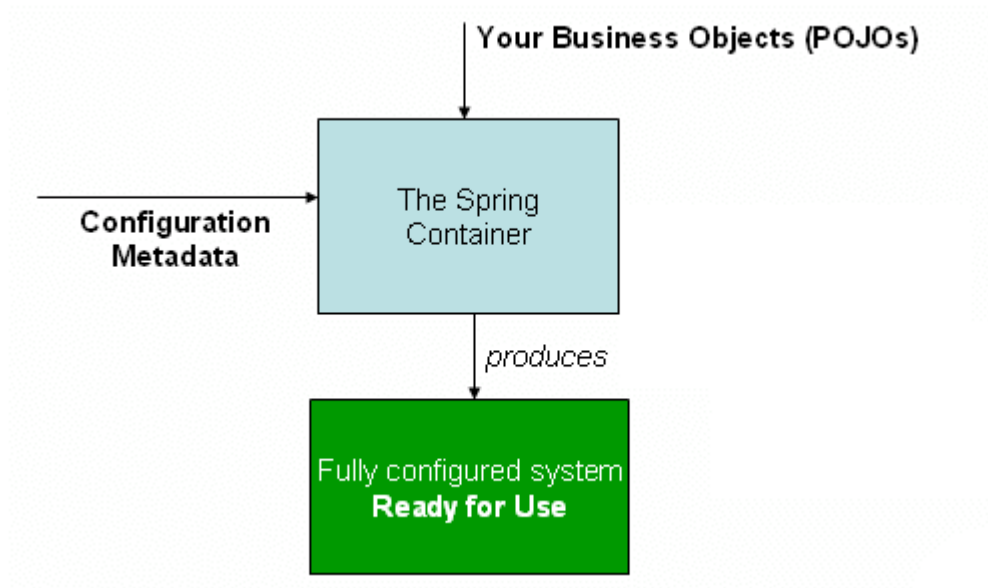
We need to tell the IoC container which objects to help us manage, which is defined through configuration metadata. Configuration metadata can be XML files, annotations or Java configuration classes.

```
1      <!-- Constructor-arg with 'value' attribute -->
2      <bean id="..." class="...">
3          <constructor-arg value="..." />
4      </bean>
```

xml

The following diagram simply shows how the IoC container uses configuration metadata to manage objects.





`org.springframework.beans` These `org.springframework.context` two packages are the basis of IoC implementation. If you want to study the source code related to IoC, you can check it out.

What are the annotations for declaring a class as a Bean?

- `@Component` : A general annotation that can mark any class as Spring a component. If you don't know which layer a bean belongs to, you can use `@Component` annotations.
- `@Repository` : Corresponding to the persistence layer, namely the Dao layer, it is mainly used for database-related operations.
- `@Service` : Corresponding to the service layer, it mainly involves some complex logic and requires the use of the Dao layer.
- `@Controller` : Corresponding to the Spring MVC control layer, it is mainly used to accept user requests and call Service the layer to return data to the front-end page.

What is the difference between `@Component` and `@Bean`?

- `@Component` Annotations act on classes, while `@Bean` annotations act on methods.
- `@Component` Typically, a bean is automatically detected and automatically wired into the Spring container through classpath scanning (we can use `@ComponentScan` annotations to define the path to scan, identify the classes to be wired, and then automatically wire them into the Spring bean container).
`@Bean` Annotations are typically used to define the bean generated in a method with the annotation, `@Bean` telling Spring that this is an instance of a certain class to return it to me when I need it.



- `@Bean` Annotations `@Component` are more customizable than annotations, and in many places we can only `@Bean` register beans through annotations. For example, when we reference classes in third-party libraries and need to assemble them into Spring the container, we can only `@Bean` do so through annotations.

`@Bean` Annotation usage examples:

```
1  @Configuration                                     java
2  public class AppConfig {
3      @Bean
4      public TransferService transferService() {
5          return new TransferServiceImpl();
6      }
7
8  }
```

The above code is equivalent to the following xml configuration

```
1  <beans>                                             xml
2      <bean id="transferService"
3      class="com.acme.TransferServiceImpl"/>
4  </beans>
```

The following example is `@Component` not possible with .

```
1  @Bean                                             java
2  public OneService getService(status) {
3      case (status) {
4          when 1:
5              return new serviceImpl1();
6          when 2:
7              return new serviceImpl2();
8          when 3:
9              return new serviceImpl3();
10     }
11 }
```

What are the annotations for injecting beans?



Spring's built-in `@Autowired` and JDK's built-in `@Resource` and `@Inject` can be used to inject beans.

Annotation	Package	Source
@Autowired	org.springframework.bean.factory	Spring 2.5+
@Resource	javax.annotation	Java JSR-250
@Inject	javax.inject	Java JSR-330

@Autowired And @Resource the ones used more.

What is the difference between @Autowired and @Resource?

Autowired It is a built-in annotation of Spring. The default injection method is `byType` (matching by type), which means that the interface type will be matched and injected into the Bean (implementation class of the interface) first.

What's the problem with this? When an interface has multiple implementation classes, `byType` this method cannot correctly inject objects, because at this time Spring will find multiple options that meet the conditions at the same time, and by default it doesn't know which one to choose.

In this case, the injection method will become `byName` (matching by name), which is usually the class name (with the first letter lowercase). For example, in the code below `smsService`, this is the name I am talking about, which should be easier to understand.

```

1 // smsService
2 @Autowired
3 private SmsService smsService;
```

java

For example, `SmsService` the interface has two implementation classes: `SmsServiceImpl1` and `SmsServiceImpl2`, and both are managed by the Spring container.

```

1 //      byName      byType      bean
2 @Autowired
3 private SmsService smsService;
4 //      SmsServiceImpl1      bean
5 @Autowired
6 private SmsService smsServiceImpl1;
```

java


```

8      //          SmsServiceImpl          bean
9      // smsServiceImpl
10     @Autowired
11     @Qualifier(value = "smsServiceImpl")
        private SmsService smsService;

```

We still recommend `@Qualifier` specifying the name explicitly through annotations rather than relying on the variable name.

`@Resource` It is an annotation provided by JDK. The default injection method is `byName` . If the corresponding bean cannot be matched by name, the injection method will become `byType` .

`@Resource` There are two important properties that are commonly used in daily development: `name` (name) and `type` (type).

```

1      public @interface Resource {
2          String name() default "";
3          Class<?> type() default Object.class;
4      }

```

java

If you specify only `name` the attribute, the injection method is `byName` . If you specify only `type` the attribute, the injection method is `byType` . If you specify both the `name` and `type` attributes (not recommended), the injection method is `byType + byName` .

```

1      //          byName    byType          bean
2      @Resource
3      private SmsService smsService;
4      //          SmsServiceImpl          bean
5      @Resource
6      private SmsService smsServiceImpl;
7      //          SmsServiceImpl          bean
8      @Resource(name = "smsServiceImpl")
9      private SmsService smsService;

```

java

To summarize briefly:

- `@Autowired` It is an annotation provided by Spring and `@Resource` an annotation provided by JDK.
- `Autowired` The default injection method is `byType` (matching by type) and `@Resource` the default injection method is `byName` (matching by name).



- When an interface has multiple implementation classes, `@Autowired` both `@Resource` must be named to correctly match the corresponding bean. `Autowired` You can explicitly specify the name through `@Qualifier` annotations or `@Resource` through `name` attributes.
- `@Autowired` Supports use on constructors, methods, fields, and parameters. `@Resource` Mainly used for injection on fields and methods, not supported on constructors or parameters.

What are the ways to inject beans?

Common ways of dependency injection (DI):

1. Constructor injection: Inject dependencies through the class constructor.
2. Setter injection: Inject dependencies through the Setter method of a class.
3. Field injection: Use annotations (such as `@Autowired` or `@Resource`) directly on the fields of a class to inject dependencies.

Constructor injection example:

```

1  @Service
2  public class UserService {
3
4      private final UserRepository userRepository;
5
6      public UserService(UserRepository userRepository) {
7          this.userRepository = userRepository;
8      }
9
10     //...
11 }

```

java

Setter injection example:

```

1  @Service
2  public class UserService {
3
4      private UserRepository userRepository;
5
6      // Spring 4.3
7      @Autowired
8      public void setUserRepository(UserRepository userRepository) {

```

java

```
9         this.userRepository = userRepository;  
10    }  
11  
12    //...  
13 }
```

Field injection example:

```
1  @Service  
2  public class UserService {  
3  
4      @Autowired  
5      private UserRepository userRepository;  
6  
7      //...  
8  }
```

java


Constructor injection or Setter injection?

Spring officially has an answer to this question: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html#beans-setter-injection> .

Here I mainly extract, summarize and improve Spring's official suggestions.

Spring officially recommends constructor injection , which has the following advantages:

1. **Dependency integrity**: Ensures that all required dependencies are injected when the object is created, avoiding the risk of null pointer exceptions.
2. **Immutability**: helps create immutable objects and improves thread safety.
3. **Initialization guarantee**: Components are fully initialized before use, reducing potential errors.
4. **Testing convenience**: In unit tests, you can pass in mock dependencies directly through the constructor without having to rely on Spring container injection.

Constructor injection is suitable for handling **required dependencies** , while **Setter injection** is more suitable for **optional dependencies** that can have default values or be set dynamically during the object lifecycle. Although `@Autowired` Setter methods can  be used to handle required dependencies, constructor injection is still a better choice.

In some cases (such as third-party classes that do not provide Setter methods), constructor injection may be **the only option**.

What are the scopes of beans?

The scopes of beans in Spring are usually as follows:

- **singleton** : There is only one bean instance in the IoC container. Beans in Spring are singletons by default, which is an application of the singleton design pattern.
- **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** (available only for Web applications): Each Web application creates a bean (application bean) when it starts. The 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.

How to configure the scope of the bean?

XML method:

```
1 <bean id="..." class="..." scope="singleton"></bean> xml
```

Annotation method:

```
1 @Bean java
2 @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
3 public Person personPrototype() {
4     return new Person();
5 }
```

Are beans thread-safe?



Whether a bean in the Spring framework is thread-safe depends on its scope and state.

Here we will use the two most commonly used scopes, prototype and singleton, as examples. Almost all scenarios use the default singleton scope, so we will focus on the singleton scope.

In prototype scope, a new bean instance is created for each access, eliminating resource contention and thread safety issues. In singleton scope, there is only one bean instance in the IoC container, which may lead to resource contention (depending on whether the bean is stateful). If the bean is stateful, thread safety issues may arise (stateful beans are objects with mutable member variables).

Stateful Bean Example:

```
1 // List java
2 @Component
3 public class ShoppingCart {
4     private List<String> items = new ArrayList<>();
5
6     public void addItem(String item) {
7         items.add(item);
8     }
9
10    public List<String> getItems() {
11        return items;
12    }
13 }
```

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

Stateless Bean Example:

```
1 // java
2 @Component
3 public class UserService {
4
5     public User findUserById(Long id) {
6         //...
7     }
8     //...
9 }
```

There are three common solutions to the thread safety problem of stateful singleton beans:

1. **Avoid mutable member variables** : Try to design beans to be stateless.
2. **Usage ThreadLocal** : Store mutable member variables in `ThreadLocal` to ensure thread independence.
3. **Use synchronization mechanism** : Use `synchronized` or `ReentrantLock` for synchronization control to ensure thread safety.

Here we take as `ThreadLocal` an example to demonstrate `ThreadLocal` the scenario of saving user login information:

```
1 public class UserThreadLocal {
2
3     private UserThreadLocal() {}
4
5     private static final ThreadLocal<SysUser> LOCAL =
6 ThreadLocal.withInitial(() -> null);
7
8     public static void put(SysUser sysUser) {
9         LOCAL.set(sysUser);
10    }
11
12    public static SysUser get() {
13        return LOCAL.get();
14    }
15
16    public static void remove() {
17        LOCAL.remove();
18    }
19 }
```

java

Do you know the life cycle of Bean?

1. **Create an instance of the Bean** : The Bean container first finds the Bean definition in the configuration file, and then uses the Java reflection API to create an instance of the Bean.
2. **Bean property assignment/filling** : Set relevant properties and dependencies for the bean, such as `@Autowired` objects injected by annotations, `@Value` injected values, setter method or constructor injection dependencies and values, and `@Resource` various injected resources.
3. **Bean initialization** :
 - If the Bean implements `BeanNameAware` the interface, call `setBeanName()` the method and pass in the name of the Bean.



- If the Bean implements `BeanClassLoaderAware` the interface, call `setBeanClassLoader()` the method and pass in `ClassLoader` the instance of the object.
- If the Bean implements `BeanFactoryAware` the interface, call `setBeanFactory()` the method and pass in `BeanFactory` the instance of the object.
- Similar to the above, if other `*.Aware` interfaces are implemented, the corresponding methods are called.
- If there is an object related to the Spring container that loaded this Bean `BeanPostProcessor`, execute `postProcessBeforeInitialization()` the method
- If the Bean implements `InitializingBean` the interface, execute `afterPropertiesSet()` the method.
- If the bean definition in the configuration file contains `init-method` the attribute, the specified method is executed.
- If there is a object associated with the Spring container that loaded this Bean `BeanPostProcessor`, execute `postProcessAfterInitialization()` the method.

4. **Destroy Bean** : Destruction does not mean to destroy the Bean immediately, but to record the Bean's destruction method first. When the Bean or container needs to be destroyed in the future, these methods will be called to release the resources held by the Bean.

- If the Bean implements `DisposableBean` the interface, execute `destroy()` the method.
- If the bean definition in the configuration file contains `destroy-method` the attribute, execute the specified bean destruction method. Alternatively, you can directly use `@PreDestroy` annotations to mark the method to be executed before the bean is destroyed.

`AbstractAutowireCapableBeanFactory` In the method, `doCreateBean()` we can see that these four stages are executed in sequence:

```

1  protected Object doCreateBean(final String beanName, final      java
2  RootBeanDefinition mbd, final @Nullable Object[] args)
3      throws BeanCreationException {
4
5      // 1.      Bean
6      BeanWrapper instanceWrapper = null;
7      if (instanceWrapper == null) {
8          instanceWrapper = createBeanInstance(beanName, mbd, args);
9      }
10
11     Object exposedObject = bean;
12     try {

```

```

13         // 2. Bean      /
14         populateBean(beanName, mbd, instanceWrapper);
15         // 3. Bean
16         exposedObject = initializeBean(beanName, exposedObject,
17 mbd);
18     }
19
20     // 4.      Bean-
21     try {
22         registerDisposableBeanIfNecessary(beanName, bean, mbd);
23     }
24
25     return exposedObject;
26 }

```

Aware The interface allows Bean to get Spring container resources.

The interfaces provided in Spring Aware are:

1. BeanNameAware : Inject the current bean corresponding to beanName;
2. BeanClassLoaderAware : Inject the ClassLoader that loads the current bean;
3. BeanFactoryAware : Injects BeanFactory a reference to the current container.

BeanPostProcessor Interface is a powerful extension point provided by Spring for modifying Bean.

```

1 public interface BeanPostProcessor {
2
3     //
4     default Object postProcessBeforeInitialization(Object bean,
5 String beanName) throws BeansException {
6         return bean;
7     }
8
9     //
10    default Object postProcessAfterInitialization(Object bean,
11 String beanName) throws BeansException {
12        return bean;
13    }
14 }

```

- `postProcessBeforeInitialization` : After Bean instantiation and property injection are completed, `InitializingBean#afterPropertiesSet` the method and the custom `init-method` method are executed;
- `postProcessAfterInitialization` : Similar to the above, but executed after `InitializingBean#afterPropertiesSet` the method and the custom method. `init-method`

`InitializingBean` and `init-method` are extension points provided by Spring for Bean initialization.

```

1 public interface InitializingBean {
2     //
3     void afterPropertiesSet() throws Exception;
4 }

```

Specify `init-method` the method to specify the initialization method:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6       http://www.springframework.org/schema/beans/spring-beans.xsd">
7
8       <bean id="demo" class="com.chaycao.Demo" init-method="init()"/>
9
10    </beans>

```

How to remember?

1. Overall, it can be simply divided into four steps: instantiation -> property assignment -> initialization -> destruction.
2. The initialization step involves many steps, including Aware dependency injection of the interface, `BeanPostProcessor` processing before and after initialization, and initialization operations of `InitializingBean` and `.init-method`
3. The destruction step will register the relevant destruction callback interface, and finally destroy it through `DisposableBean` and `.destroy-method`

Finally, let me share a clear diagram (source: [How to Remember the Life Cycle of Spring Beans](#)).

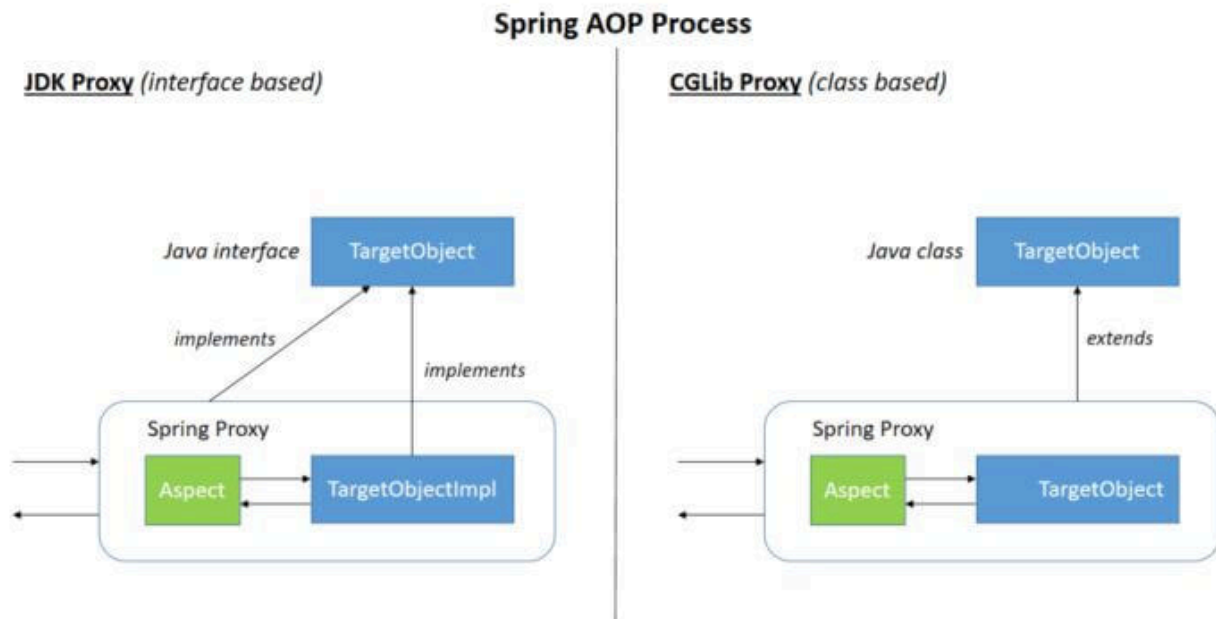




Talk about your understanding of AOP

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 should be considered the most complete AOP framework in the Java ecosystem.

Some professional terms involved in AOP aspect programming:

the term	meaning
Target	Notified party
Proxy	The proxy object created after applying the advice to the target object
JoinPoint	All methods defined in the class of the target object are connection points
Pointcut	Join points intercepted/enhanced by aspects (entry points are always join points, but join points are not necessarily entry points)
Advice	Enhanced logic/code, that is, what to do after intercepting the connection point of the target object
Aspect	Pointcut + Advice
Weaving	Apply the notification to the target object, thereby generating the process action of the proxy object



What is the difference between Spring AOP and AspectJ AOP?

characteristic	Spring AOP	AspectJ
Enhancement method	Runtime enhancement (based on dynamic proxy)	Compile-time enhancement, class loading enhancement (direct manipulation of bytecode)
Entry point support	Method level (within the scope of Spring Bean, final and static methods are not supported)	Method level, field, constructor, static method, etc.
performance	The runtime relies on proxies, which has a certain overhead and has low performance when there are many facets.	Runtime without agent overhead, higher performance
Complexity	Simple, easy to use, suitable for most scenarios	Powerful, but relatively complex
Usage scenarios	Relatively simple AOP requirements in Spring applications	High-performance and high-complexity AOP requirements

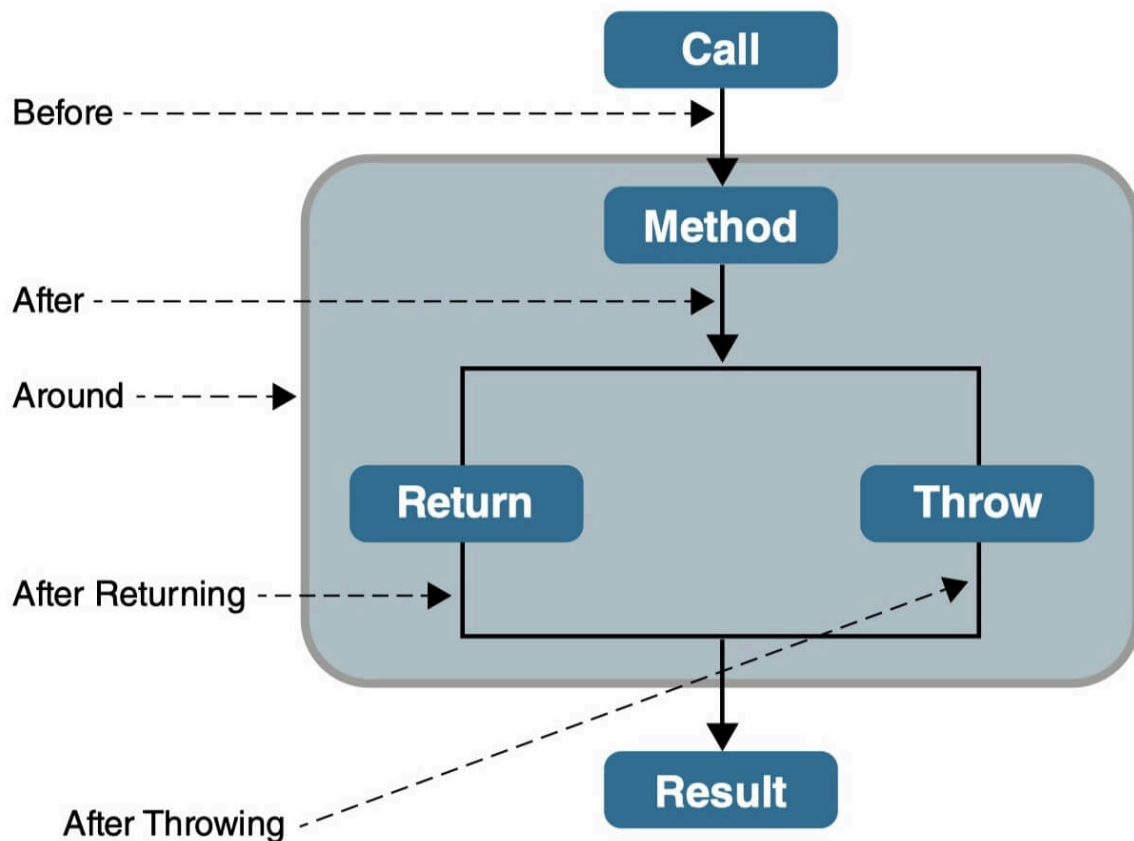
How to choose?

- **Functionality considerations** : AspectJ supports more complex AOP scenarios, while Spring AOP is simpler and easier to use. If you need to enhance final methods, static methods, field access, constructor calls, or apply enhancement logic to non-Spring managed objects, AspectJ is the only choice.
- **Performance considerations** : When the number of aspects is small, the performance difference between the two is not significant, but when there are more aspects, AspectJ performs better.

In a nutshell : Use Spring AOP first for simple scenarios; choose AspectJ for complex scenarios or high performance requirements.



What are the common types of AOP advice?



- **Before** (pre-notification): triggered before the target object's method is called
- **After** (post notification): triggered after the target object's method is called
- **AfterReturning** (return notification): triggered after the method call of the target object is completed and the result value is returned
- **AfterThrowing** (Exception Notification): Triggered after a method on the target object throws or triggers an exception. AfterReturning and AfterThrowing are mutually exclusive. If the method call succeeds without exception, a value is returned; if the method throws an exception, no value is returned.
- **Around** advice: Programmatically control the method call of the target object. Around advice has the widest range of operation among all advice types. Because it can directly obtain the target object and the method to be executed, around advice can arbitrarily do things before or after the target object's method call, or even not call the target object's method.



How to control the execution order of multiple aspects?

1. @Order Annotations are usually used to directly define the order of aspects

```
1 //
2 @Order(3)
3 @Component
4 @Aspect
5 public class LoggingAspect implements Ordered {
```

java

2. Implement Ordered the interface rewriting getOrder method.

```
1 @Component
2 @Aspect
3 public class LoggingAspect implements Ordered {
4
5     // ....
6
7     @Override
8     public int getOrder() {
9         //
10        return 1;
11    }
12 }
```

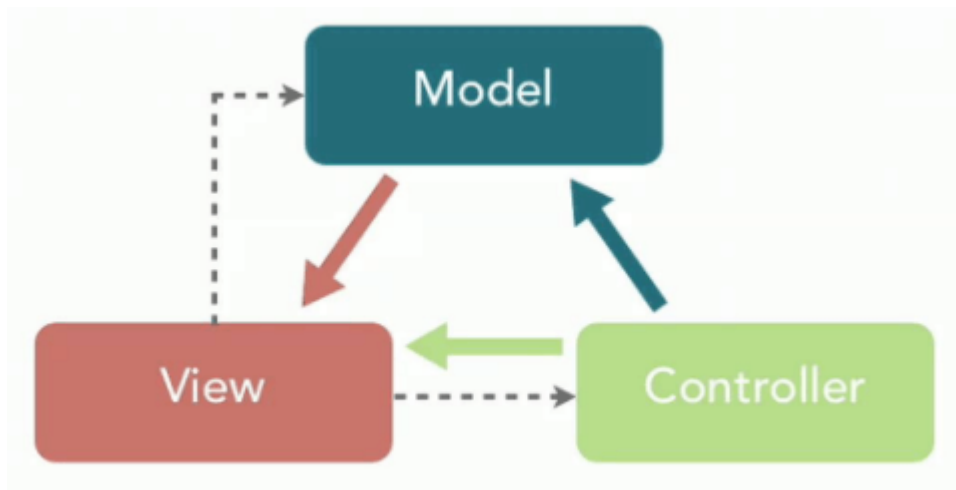
java

Spring MVC

Tell us about your understanding of Spring MVC?

MVC is the abbreviation of Model, View, Controller. Its core idea is to organize code by separating business logic, data, and display.





Many people online say that MVC is not a design pattern, but just a software design specification. I personally prefer to think that MVC is just one of many design patterns.

The java-design-patterns project has an introduction to MVC.



iluwatar / java-design-patterns

Watch 3.9k Unstar 69.4k Fork 21.6k

Code Issues 228 Pull requests 38 Actions Wiki Security Insights

master java-design-patterns / model-view-controller /

Go to file Add file ...

JackieNim fix: Fixed pages showing up in wrong language (#1752) ✓ f597fc1 on 20 May History

..

etc #1113 Add uml-reverse-mapper plugin 2 years ago

src Use lombok, reformat, and optimize the code (#1560) 5 months ago

README.md fix: Fixed pages showing up in wrong language (#1752) 1.67 KB 3 months ago

pom.xml Set version for the next development iteration 2.43 KB 4 months ago

README.md

layout	title	folder	permalink	categories	language	tags
pattern	Model-View-Controller	model-view-controller	/patterns/model-view-controller/	Architectural	en	Decoupling

Intent

Separate the user interface into three interconnected components: the model, the view and the controller. Let the model manage the data, the view display the data and the controller mediate updating the data and redrawing the display.

Class diagram

```

classDiagram
    class GiantController {
        <<Java Class>>
        com.iluwatar
        +GiantController(GiantModel, GiantView)
        +getHealth() Health
        +setHealth(Health) void
        +getFatigue() Fatigue
        +setFatigue(Fatigue) void
        +getNourishment() Nourishment
        +setNourishment(Nourishment) void
        +updateView() void
    }
    class GiantView {
        <<Java Class>>
        com.iluwatar
        +GiantView()
        +displayGiant(GiantModel) void
    }
    class GiantModel {
        <<Java Class>>
        com.iluwatar
        +health Health
        +fatigue Fatigue
    }
    GiantController --> GiantView : -view 0..1
    GiantController --> GiantModel : -giant 0..1
  
```

https://blog.csdn.net/qq_34337272

To truly understand Spring MVC, let's first look at Model 1 and Model 2, the eras without Spring MVC.

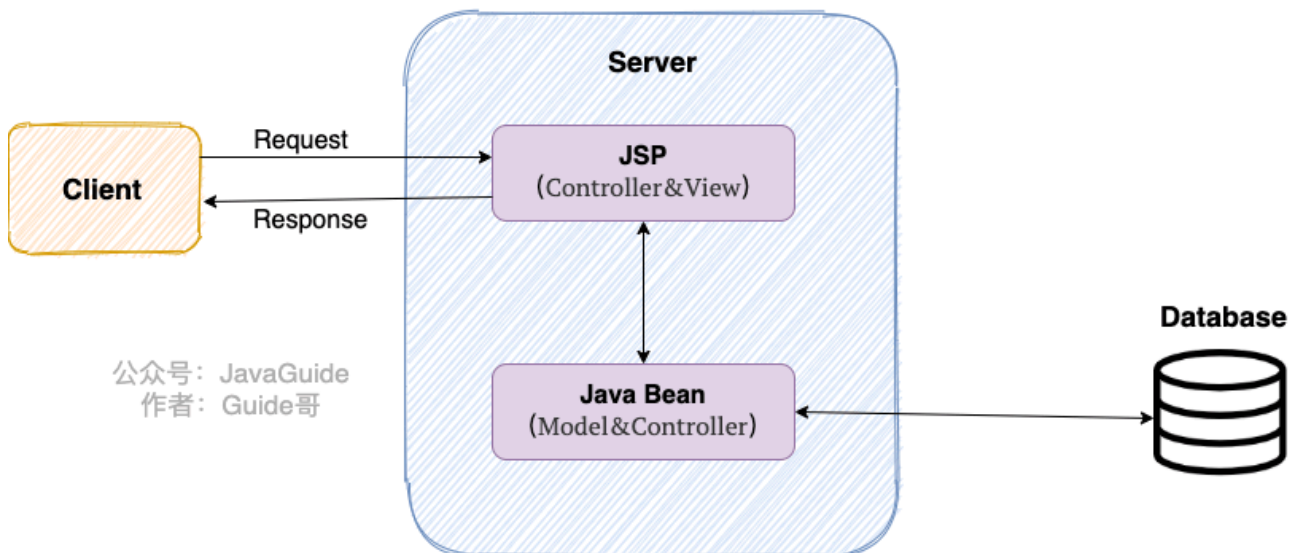
Model 1 Era

Many people who learned Java backend relatively late may not have been exposed to Java Web application development in the Model 1 era. In Model 1, the entire Web application was composed almost entirely of JSP pages, with only a small number of JavaBeans used to handle database connections, access, and other operations.



In this model, JSP serves as both the control layer (Controller) and the presentation layer (View). Obviously, this model presents numerous problems. For example, the intermingling of control logic and presentation logic results in extremely low code reuse. Furthermore, the interdependence between the front-end and back-end makes testing and maintenance difficult, and significantly reduces development efficiency.

MVC - Model 1



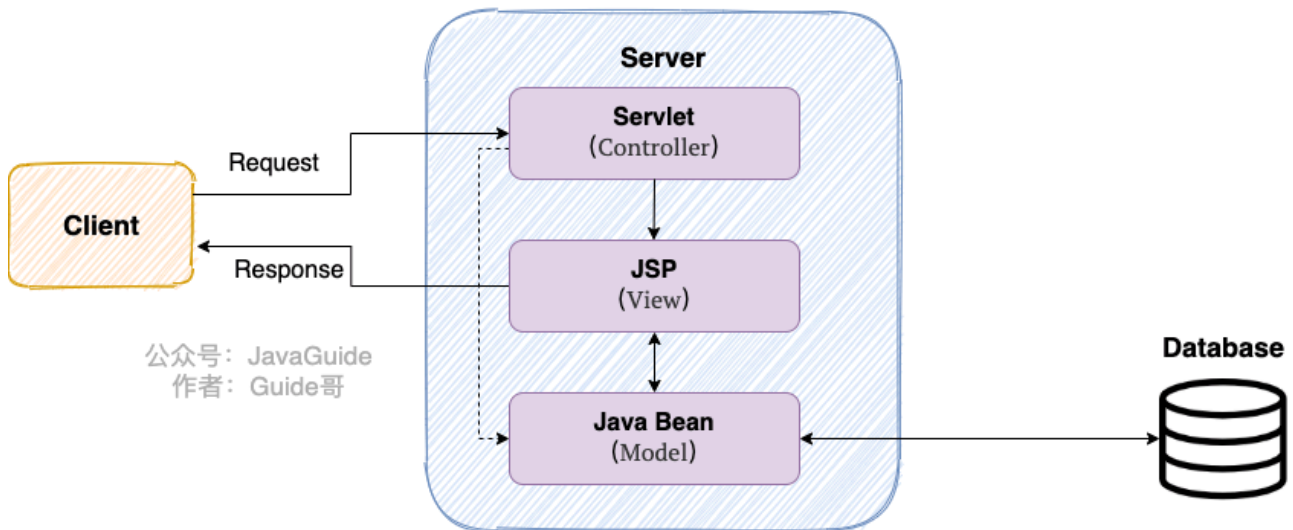
Model 2 Era

Friends who have learned Servlet and made related demos should be familiar with the development model of "Java Bean (Model) + JSP (View) + Servlet (Controller)", which is the early JavaWeb MVC development model.

- **Model:** The data involved in the system, that is, dao and beans.
- **View:** Displays the data in the model, just for display.
- **Controller:** accepts user requests, sends the requests to the Model, and finally returns data to the JSP and displays it to the user



MVC - Model 2



There are still many problems in the Model2 mode. The degree of abstraction and encapsulation of Model2 is far from enough. When using Model2 for development, it is inevitable to reinvent the wheel, which greatly reduces the maintainability and reusability of the program.

As a result, many Java Web development-related MVC frameworks emerged, such as Struts2, but Struts2 is relatively cumbersome.

Spring MVC Era

With the popularity of the Spring lightweight development framework, the Spring MVC framework has emerged in the Spring ecosystem. Spring MVC is currently the most advanced MVC framework. Compared to Struts 2, Spring MVC is simpler and more convenient to use, more efficient in development, and faster in execution.

MVC is a design pattern, and Spring MVC is an excellent MVC framework. Spring MVC helps streamline web-tier development and integrates naturally with the Spring framework. With Spring MVC, backend projects are typically divided into a Service layer (for business processing), a Dao layer (for database operations), an Entity layer (for entity classes), and a Controller layer (for returning data to the frontend).

What are the core components of Spring MVC?

Remember the following components, and you will remember how SpringMVC work 

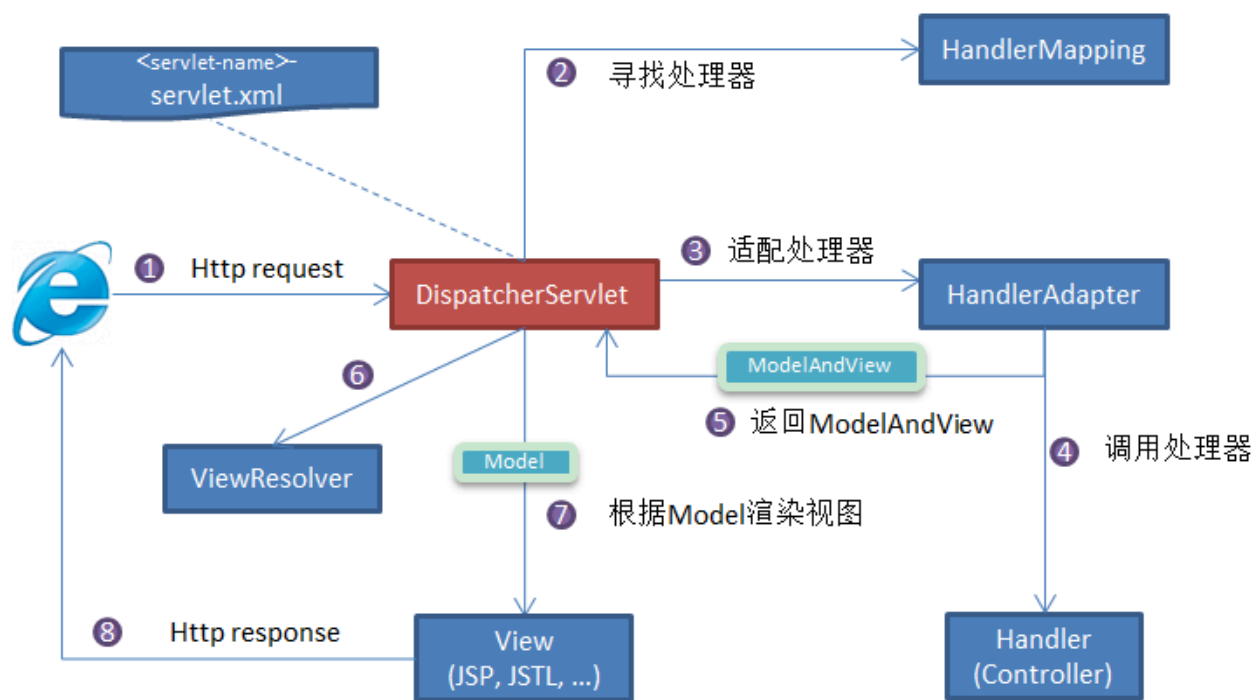
- **DispatcherServlet : The core central processing unit** , responsible for receiving requests, distributing them, and giving responses to clients.

- **HandlerMapping** : **The processor mapper** matches and searches for the handlers based on the URL Handler , and encapsulates the interceptors involved in the request Handler together.
- **HandlerAdapter** **Processor adapter** , based on HandlerMapping the found Handler , adapts and executes the corresponding Handler ;
- **Handler** : **Request handler** , the handler that handles the actual request.
- **ViewResolver** : **View resolver** , according to Handler the logical view/view returned, parses and renders the real view and passes it to DispatcherServlet the response client

Do you understand how SpringMVC works?

The Spring MVC principle is shown in the following figure:

I didn't draw the diagram of how SpringMVC works myself, so I found a very clear and intuitive one on the Internet to save time. The original source is unknown.



Process description (important):

1. The client (browser) sends a request, DispatcherServlet intercepts the request.
2. DispatcherServlet Called based on the request information HandlerMapping . HandlerMapping Matches the URL to find the controller that can handle it Handler (that is, what we usually call Controller a controller), and encapsulates the interceptors involved in the request Handler .



3. `DispatcherServlet` Call `HandlerAdapter` the adapter to execute `Handler` .
4. `Handler` After completing the processing of the user request, a `ModelAndView` object will be returned to the user `DispatcherServlet` . `ModelAndView` As the name suggests, it contains information about the data model and the corresponding view.
`Model` The returned data object `View` is a logical one `View` .
5. `ViewResolver` Will `View` find the actual one based on the logic `View` .
6. `DispatcherServlet` Pass the returned `Model` to `View` (view rendering).
7. Return `View` to the requester (browser)

The above process is how traditional development models (such as JSP and Thymeleaf) work. However, the current mainstream development approach is to separate the front-end and back-end. In this context, `View` the concept of Spring MVC has undergone some changes. Since `View` the front-end framework (such as Vue and React) is usually responsible for handling the page, the back-end is no longer responsible for rendering the page, but only for providing data. Therefore:

- When the front-end and back-end are separated, the back-end usually no longer returns a specific view, but returns **pure data** (usually in JSON format), which is rendered and displayed by the front-end.
- `View` The `__register__` part is often not required in scenarios where the front-end and back-end are separated. The Spring MVC controller method only needs to return data, `ModelAndView` and no longer returns. Instead, it returns the data directly, and Spring will automatically convert it to JSON format. Accordingly, `ViewResolver` it will no longer be used.

How to do it?

- Use `@RestController` annotations instead of traditional `@Controller` annotations so that all methods will return data in JSON format by default instead of trying to parse views.
- If you are using it `@Controller` , you can combine it with `@ResponseBody` annotations to return JSON.

How to unify exception handling?

It is recommended to use annotations to unify exception handling, specifically the two annotations `@ControllerAdvice` + will be used. `@ExceptionHandler`



```

1  @ControllerAdvice
2  @ResponseBody
3  public class GlobalExceptionHandler {
4
5      @ExceptionHandler(BaseException.class)
6      public ResponseEntity<?> handleAppException(BaseException ex,
7  HttpServletRequest request) {
8          //.....
9      }
10
11     @ExceptionHandler(value = ResourceNotFoundException.class)
12     public ResponseEntity<ErrorResponse>
13     handleResourceNotFoundException(ResourceNotFoundException ex,
14     HttpServletRequest request) {
15         //.....
16     }
17 }

```

Controller In this exception handling mode, the exception handling logic (AOP) will be woven into all or specified . Controller When the method in throws an exception, @ExceptionHandler it will be handled by the method modified by the annotation.

ExceptionHandlerMethodResolver The method in getMappedMethod the annotation determines which @ExceptionHandler method will handle the exception.

```

1  @Nullable
2  private Method getMappedMethod(Class<? extends Throwable>
3  exceptionType) {
4      List<Class<? extends Throwable>> matches = new ArrayList<>();
5      // mappedMethods
6
7      for (Class<? extends Throwable> mappedException :
8  this.mappedMethods.keySet()) {
9          if (mappedException.isAssignableFrom(exceptionType)) {
10             matches.add(mappedException);
11         }
12     }
13     //
14     if (!matches.isEmpty()) {
15         //
16         matches.sort(new ExceptionDepthComparator(exceptionType));
17         //

```

```
18         return this.mappedMethods.get(matches.get(0));
19     }
20     else {
21         return null;
22     }
23 }
```

From the source code, we can see that: **getMappedMethod()** first, all the method information that can match the exception handling method will be found, then they will be sorted from small to large, and finally the smallest matching method (that is, the one with the highest matching degree) will be taken.

What design patterns are used in Spring Framework?

For a detailed introduction to the following design patterns, please refer to my article [Detailed Explanation of Design Patterns in Spring](#).

- **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, and 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 .
- ...



Spring's Circular Dependencies

Do you know about Spring circular dependencies? How to solve them?

Circular dependency refers to a circular reference between bean objects, where two or more beans hold references to each other, for example, `CircularDependencyA` → `CircularDependencyB` → `CircularDependencyA`.

```
1  @Component                                     java
2  public class CircularDependencyA {
3      @Autowired
4      private CircularDependencyB circB;
5  }
6
7  @Component
8  public class CircularDependencyB {
9      @Autowired
10     private CircularDependencyA circA;
11 }
```

The self-dependence of a single object can also lead to circular dependencies, but the probability of this is extremely low and is considered a coding error.

```
1  @Component                                     java
2  public class CircularDependencyA {
3      @Autowired
4      private CircularDependencyA circA;
5  }
```

The Spring Framework solves this problem by using a three-level cache to ensure that beans are created correctly even in the presence of circular dependencies.

The three-level cache in Spring is actually three Maps, as follows:



```


1      //
2      /** Cache of singleton objects: bean name to bean instance. */
3      private final Map<String, Object> singletonObjects = new
4      ConcurrentHashMap<>(256);
5
6      //
7      /** Cache of early singleton objects: bean name to bean instance.
8      */
9      private final Map<String, Object> earlySingletonObjects = new
10     HashMap<>(16);
11
12     //
13     /** Cache of singleton factories: bean name to ObjectFactory. */
14     private final Map<String, ObjectFactory<?>> singletonFactories =
15     new HashMap<>(16);

```

In simple terms, Spring's three-level cache includes:

1. **Level 1 cache (singletonObjects)** : This cache stores final beans (instantiated, property-populated, and initialized). This singleton pool is used for Spring's singleton properties. Generally, beans are retrieved from this pool, but not all beans are stored in the singleton pool; for example, prototype beans are not.
2. **Second-level cache (earlySingletonObjects)** : stores transition beans (semi-finished products, properties not yet filled), that is, objects generated in the third-level cache. When used in conjunction with the third-level cache, it can prevent the generation of new proxy objects `ObjectFactory` for each call in the case of AOP `. ObjectFactory#getObject()`
3. **Level 3 cache (singletonFactories)** : stores `ObjectFactory` methods `ObjectFactory` (the method `getObject()` that is ultimately called `getEarlyBeanReference()`) that can generate the original bean object or proxy object (if the bean is proxied by an AOP aspect). Level 3 cache only works for singleton beans.

Next, let's talk about the process of Spring creating Beans:

1. First get it from **the first-level cache singletonObjects** , and return it if it exists;
2. If it does not exist or the object is being created, it is retrieved from **the secondary cache earlySingletonObjects** ;
3. If it has not been obtained yet, go to **the third-level cache singletonFactories** to obtain it. By executing `ObjectFacotry` , `getObject()` you can obtain the object.  successful acquisition, remove it from the third-level cache and add the object to the second-level cache.

Stored in the third-level cache are `ObjectFactory` :

```

1  public interface ObjectFactory<T> {
2      T getObject() throws BeansException;
3  }

```

java

When Spring creates a Bean, if circular dependencies are allowed, Spring will expose the Bean object that has just been instantiated but whose properties have not yet been initialized. Here, `addSingletonFactory` a object is added to the third-level cache through the method `ObjectFactory` :

```

1  // AbstractAutowireCapableBeanFactory # doCreateBean #
2  public abstract class AbstractAutowireCapableBeanFactory ... {
3      protected Object doCreateBean(...) {
4          //...
5
6          //          ()->getEarlyBeanReference
7  ObjectFactory      getObject()
8      addSingletonFactory(beanName, () ->
9  getEarlyBeanReference(beanName, mbd, bean));
      }
  }

```

java

As mentioned above when talking about the process of Spring creating Beans, if neither the first-level cache nor the second-level cache can get the object, the third-level cache will be used to get the object through `ObjectFactory` the `getObject` method.

```

1  class A {
2      //      B
3      private B b;
4  }
5  class B {
6      //      A
7      private A a;
8  }

```

java

Taking the above circular dependency code as an example, the entire process of resolving circular dependencies is as follows:

- After Spring creates A, it finds that A depends on B, so it creates B. B depends on A, so it creates A again.



- When B creates A, A has a circular dependency. Since A has not yet been initialized, there is no A in **the first and second level caches** .
- Then call `getObject()` the method in the third-level cache to obtain **the object exposed in the early stage** of A , that is, call the method added above to generate an **early exposure object** `getEarlyBeanReference()` of A ;
- Then `ObjectFactory` remove this from the third-level cache and put the previously exposed object into the second-level cache. Then B injects this previously exposed object into the dependency to support circular dependencies.

Is using only two levels of cache sufficient? Without AOP, it's indeed possible to resolve circular dependencies using only the first and second levels of cache. However, when AOP is involved, the third level of cache becomes crucial because it ensures that even if multiple requests for earlier references are made during the bean creation process, the same proxy object is always returned, thus avoiding the problem of multiple proxy objects for the same bean.


Finally, let's summarize how Spring solves the three-level cache :

In the third-level cache, you only need to remember how Spring supports circular dependencies. That is, if a circular dependency occurs, go to **the third-level cache singletonFactories** and get the object stored in the third-level cache `ObjectFactory` and call its `getObject()` method to obtain the early exposure object of the circular dependency object (although it has not been initialized yet, you can get the storage address of the object in the heap), and put this early exposure object into the second-level cache, so that it will not be initialized repeatedly when there is a circular dependency!

However, this mechanism has some drawbacks, such as increased memory overhead (requiring maintenance of a three-level cache, meaning three maps) and reduced performance (requiring multiple checks and conversions). Furthermore, circular dependencies are not supported in some cases, such as non-singleton beans and `@Async` annotated beans.

Can @Lazy resolve circular dependencies?

`@Lazy` Used to identify whether a class needs lazy loading/delayed loading. It can be used on classes, methods, constructors, method parameters, and member variables.

Spring Boot 2.2 adds **a global lazy loading property** . When enabled, global bean.  set to lazy loading and created when needed.

The configuration file configures global lazy loading:

```

1  #    false                                properties
2  spring.main.lazy-initialization=true

```

The encoding method sets global lazy loading:

```

1  SpringApplication springApplication=new      java
2  SpringApplication(Start.class);
3  springApplication.setLazyInitialization(false);
   springApplication.run(args);

```


If not necessary, try not to use global lazy loading. Global lazy loading will make the bean loading slower when it is first used, and it will delay the discovery of application problems (problems will only appear when the bean is initialized).

If a bean is not marked as lazy, it is created and initialized during Spring IoC container startup. If a bean is marked as lazy, it is not instantiated immediately when the Spring IoC container starts, but is created when it is first requested. This can help reduce initialization time during application startup and can also be used to resolve circular dependencies.

How is the circular dependency problem `@Lazy` solved? Here's an example: suppose there are two beans, A and B, with a circular dependency. `@Lazy` After adding an annotation to A's constructor (delaying the instantiation of Bean B), the loading process is as follows:

- First, Spring will create A's bean, and when creating it, it needs to inject B's properties;
- Since A is annotated `@Lazy`, Spring will create a proxy object of B and inject this proxy object into the B property in A;
- Then start to execute the instantiation and initialization of B. When injecting the A attribute in B, A has been created at this time and can be injected into it.

From the above loading process, we can see that `@Lazy` the key to solving circular dependencies lies in the use of proxy objects.

- **@Lazy In the absence of :** When the Spring container is initialized , A it will try to create it immediately B , and during the creation B process it will try to create it again A , which will eventually lead to a circular dependency (that is, infinite recursion and eventually throw an exception).
- **@Lazy When using :** Spring does not create a HttpClient immediately B . Instead, it injects a B proxy object for HttpClient. Because HttpClient B is not yet fully initialized, A HttpClient can complete initialization successfully. The proxy object triggers th  A actual initialization of HttpClient when the HttpClient instance actually calls the HttpClient method . B B

@Lazy This can, to a certain extent, break the circular dependency chain, allowing the Spring container to successfully complete bean creation and injection. However, @Lazy this isn't a fundamental solution, especially in scenarios like constructor injection and complex multi-level dependencies. Therefore, best practice remains to avoid circular dependencies in design.

Does SpringBoot allow circular dependencies to occur?

Prior to Spring Boot 2.6.x, circular dependencies were allowed by default. This meant that even if your code contained circular dependencies, it generally wouldn't generate an error. However, Spring Boot 2.6.x and later officially no longer recommends writing code with circular dependencies. Developers are advised to reduce unnecessary dependencies when writing code. This is actually the most important thing we should do. Circular dependencies are a design flaw in themselves, and we shouldn't rely too heavily on Spring at the expense of coding standards and quality. Perhaps a future Spring Boot version will completely prohibit circular dependencies.

After SpringBoot 2.6.x, if you don't want to refactor the circular dependency code, you can also use the following methods:

- Set in the global configuration file to allow circular dependencies: `spring.main.allow-circular-references=true` . This is the simplest and crudest way, not recommended.
- Adding annotations to beans that cause circular dependencies @Lazy is a recommended approach. @Lazy This annotation is used to indicate whether a class requires lazy loading or delayed loading. It can be applied to classes, methods, constructors, method parameters, and member variables.
- ...

Spring Transactions

For a detailed introduction to Spring transactions, see my article [Spring Transaction Explanation](#).

How many ways does Spring manage transactions?

- **Programmatic transactions** : hard-coded in the code (recommended in distributed systems): transactions are managed manually. If `TransactionTemplate` the `TransactionManager` transaction scope is too large, the transaction will not be



committed and will cause timeouts. Therefore, the granularity of transactions is smaller than that of locks.

- **Declarative transactions** : configured in XML configuration files or directly based on annotations (recommended for single applications or simple business systems): actually implemented through AOP (`@Transactional` the full annotation method based on is most commonly used)

What are the transaction propagation behaviors in Spring transactions?

The purpose of transaction propagation is to solve the transaction problem of mutual calls between business layer methods .

When a transactional method is called by another transactional method, you must specify how the transaction should propagate. For example, the method may continue to run in the existing transaction, or it may start a new transaction and run in its own transaction.

Possible values for correct transaction propagation behavior are as follows:

1. **TransactionDefinition.PROPGATION_REQUIRED**

`@Transactional` The most commonly used transaction propagation behavior is the default behavior for annotations we often use . If a transaction currently exists, it will be joined; if there is no transaction currently, a new transaction will be created.

2. **TransactionDefinition.PROPGATION_REQUIRES_NEW**

Create a new transaction. If a transaction currently exists, suspend the current transaction. That is, regardless of whether the outer method starts a transaction or not,

`Propagation.REQUIRES_NEW` the modified inner method will start its own transaction, and the opened transactions are independent of each other and do not interfere with each other.

3. **TransactionDefinition.PROPGATION_NESTED**

If a transaction currently exists, create a transaction to run as a nested transaction of the current transaction; if there is no transaction currently, the value is equivalent to `TransactionDefinition.PROPGATION_REQUIRED` .

4. **TransactionDefinition.PROPGATION_MANDATORY**



If a transaction currently exists, join the transaction; if no transaction currently exists, throw an exception. (mandatory: mandatory)

This is rarely used.

If the following three transaction propagation behaviors are incorrectly configured, the transaction will not be rolled back:

- **TransactionDefinition.PROPGATION_SUPPORTS** : If a transaction currently exists, join the transaction; if there is no transaction currently, continue running in a non-transactional manner.
- **TransactionDefinition.PROPGATION_NOT_SUPPORTED** : Run in non-transactional mode. If a transaction currently exists, suspend the current transaction.
- **TransactionDefinition.PROPGATION_NEVER** : Run in non-transactional mode and throw an exception if a transaction currently exists.

What are the isolation levels in Spring transactions?

Like transaction propagation behavior, Spring also defines an enumeration class for ease of use: Isolation

```
1  public enum Isolation {
2
3      DEFAULT(TransactionDefinition.ISOLATION_DEFAULT),
4
5      READ_UNCOMMITTED(TransactionDefinition.ISOLATION_READ_UNCOMMITTED),
6      READ_COMMITTED(TransactionDefinition.ISOLATION_READ_COMMITTED),
7
8      REPEATABLE_READ(TransactionDefinition.ISOLATION_REPEATABLE_READ),
9      SERIALIZABLE(TransactionDefinition.ISOLATION_SERIALIZABLE);
10
11     private final int value;
12
13     Isolation(int value) {
14         this.value = value;
15     }
16
17     public int value() {
18         return this.value;
19     }
20 }
```



Below I will introduce each transaction isolation level in turn:

- **TransactionDefinition.ISOLATION_DEFAULT** : Use the default isolation level of the backend database, the default `REPEATABLE_READ` isolation level used by MySQL and the default isolation level used by Oracle `READ_COMMITTED` .
- **TransactionDefinition.ISOLATION_READ_UNCOMMITTED** : The lowest isolation level. This isolation level is rarely used because it allows reading of uncommitted data changes, **which may result in dirty reads, phantom reads, or non-repeatable reads.**
- **TransactionDefinition.ISOLATION_READ_COMMITTED** : Allows reading of data that has been committed by concurrent transactions, **which can prevent dirty reads, but phantom reads or non-repeatable reads may still occur**
- **TransactionDefinition.ISOLATION_REPEATABLE_READ** : The results of multiple reads of the same field are consistent, unless the data is modified by the transaction itself. **Dirty reads and non-repeatable reads can be prevented, but phantom reads may still occur.**
- **TransactionDefinition.ISOLATION_SERIALIZABLE** The highest isolation level, fully compliant with ACID isolation. All transactions are executed sequentially, preventing interference between transactions. **This level prevents dirty reads, non-repeatable reads, and phantom reads .** However, this level can severely impact application performance. It is not typically used.

Do you understand the `@Transactional(rollbackFor = Exception.class)` annotation?

Exception There are two types of exceptions: runtime exceptions `RuntimeException` and non-runtime exceptions. Transaction management is crucial for enterprise applications, as it ensures data consistency even in the event of an exception.

When `@Transactional` the annotation is applied to a class, all public methods of the class will have this type of transaction attribute. At the same time, we can also use the annotation at the method level to override the class level definition.

`@Transactional` The default rollback policy is to roll back the transaction only when encountering `RuntimeException` (Runtime Exception) or (Checked Exception) `Error` , but not `Checked Exception` (Checked Exception). This is because Spring considers `RuntimeException` `Error` and `Error` to be unexpected errors, while checked exceptions are expected errors that can be handled by business logic.



Defines zero (0) or more exception **classes**, which must be subclasses of **Throwable**, indicating which exception types must cause a transaction rollback.

By default, a transaction will be rolled back on **RuntimeException** and **Error** but not on checked exceptions (business exceptions). See `org.springframework.transaction.interceptor.DefaultTransactionAttribute.rollbackOn(Throwable)` for a detailed explanation.

This is the preferred way to construct a rollback rule (in contrast to `rollbackForClassName`), matching the exception type, its subclasses, and its nested classes. See the **class-level javadocs** for further details on rollback rule semantics and warnings regarding possible unintentional matches.

请参阅: `rollbackForClassName`,
`org.springframework.transaction.interceptor.`
`RollbackRuleAttribute.RollbackRuleAttribute(Class)`,
`org.springframework.transaction.interceptor.`
`DefaultTransactionAttribute.rollbackOn(Throwable)`

```
Class<? extends Throwable>[] rollbackFor() default {};
```

If you want to modify the default rollback strategy, you can use `@Transactional` the annotation's `rollbackFor` and `noRollbackFor` attributes to specify which exceptions need to be rolled back and which exceptions do not need to be rolled back. For example, if you want all exceptions to roll back the transaction, you can use the following annotation:

```
1 @Transactional(rollbackFor = Exception.class)
2 public void someMethod() {
3     // some business logic
4 }
```

If you want to prevent certain exceptions from rolling back the transaction, you can use the following annotations:

```
1 @Transactional(noRollbackFor = CustomException.class)
2 public void someMethod() {
3     // some business logic
4 }
```

Spring Data JPA



The most important thing about JPA is practical application. Here we only summarize a small part of the knowledge points.

How to de-persist a field in the database using JPA?

If we have the following class:

```
1  @Entity(name="USER")
2  public class User {
3
4      @Id
5      @GeneratedValue(strategy = GenerationType.AUTO)
6      @Column(name = "ID")
7      private Long id;
8
9      @Column(name="USER_NAME")
10     private String userName;
11
12     @Column(name="PASSWORD")
13     private String password;
14
15     private String secret;
16
17 }
```

java

What if we want to prevent `secret` this field from being persisted, that is, not stored in the database? We can use the following methods:

```
1  static String transient1; // not persistent because of static
2  final String transient2 = "Satish"; // not persistent because of
3  final
4  transient String transient3; // not persistent because of transient
5  @Transient
   String transient4; // not persistent because of @Transient
```

java

Generally, the latter two methods are used more often. I personally use annotations more often.



What does JPA's auditing feature do? What is its use?

The audit function mainly helps us record the specific behavior of database operations, such as who created a record, when it was created, who last modified it, and when it was last modified.

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  @MappedSuperclass
5  @EntityListeners(value = AuditingEntityListener.class)
6  public abstract class AbstractAuditBase {
7
8      @CreatedDate
9      @Column(updatable = false)
10     @JsonIgnore
11     private Instant createdAt;
12
13     @LastModifiedDate
14     @JsonIgnore
15     private Instant updatedAt;
16
17     @CreatedBy
18     @Column(updatable = false)
19     @JsonIgnore
20     private String createdBy;
21
22     @LastModifiedBy
23     @JsonIgnore
24     private String updatedBy;
25 }
```

java

- `@CreatedDate` : Indicates that this field is a creation time field. When this entity is inserted, the value will be set.
- `@CreatedBy` : Indicates that this field is the creator. When this entity is inserted, the value will be set

`@LastModifiedDate` , `@LastModifiedBy` similarly.



What are the annotations for the association relationships between entities?

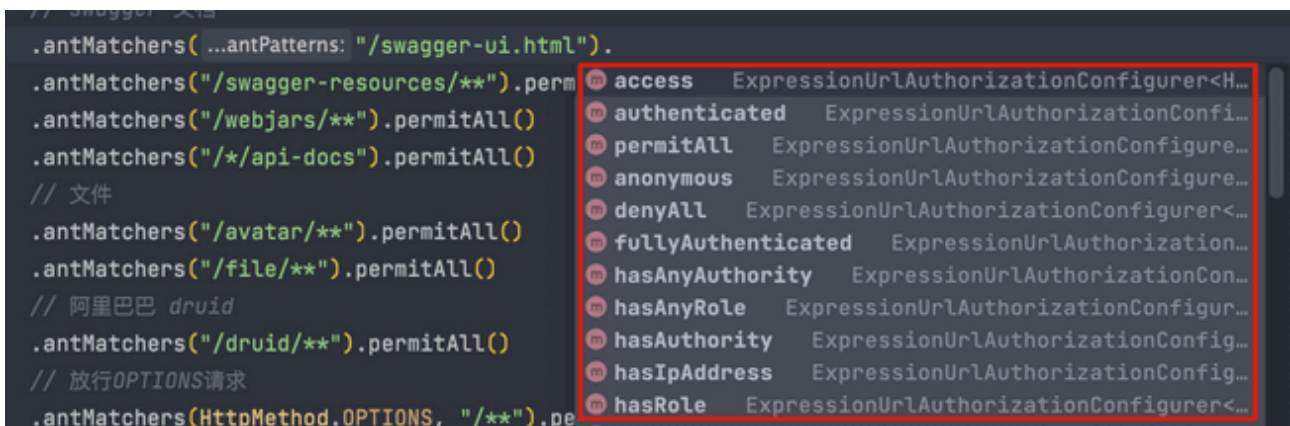
- @OneToOne : One-on-one.
- @ManyToMany : Many-to-many.
- @OneToMany : One to many.
- @ManyToOne : Many to one.

Many-to-many relationships can also be expressed using @ManyToOne and @OneToMany

Spring Security

The important thing about Spring Security is actual combat. Here we only summarize a small part of the knowledge points.

What are the ways to control access permissions requested?



- `permitAll()` : Unconditionally allow any form of access, regardless of whether you are logged in or not.
- `anonymous()` : Allow anonymous access, that is, you can access it without logging in.
- `denyAll()` : Unconditionally reject any form of access.
- `authenticated()` : Only authenticated users are allowed to access.
- `fullyAuthenticated()` : Only users who are logged in or logged in through remote are allowed to access.
- `hasRole(String)` : Only allow specified roles to access.

- `hasAnyRole(String)` : Specify one or more roles. Users who meet one of the roles can access the function.
- `hasAuthority(String)` : Only users with specified permissions are allowed to access
- `hasAnyAuthority(String)` : Specify one or more permissions. Users who meet any of them can access the service.
- `hasIpAddress(String)` : Only allow users with specified IP addresses to access.

Is there a difference between `hasRole` and `hasAuthority`?

You can take a look at this article by Songge: [Is there a difference between `hasRole` and `hasAuthority` in Spring Security?](#) , which introduces it in more detail.

How to encrypt passwords?

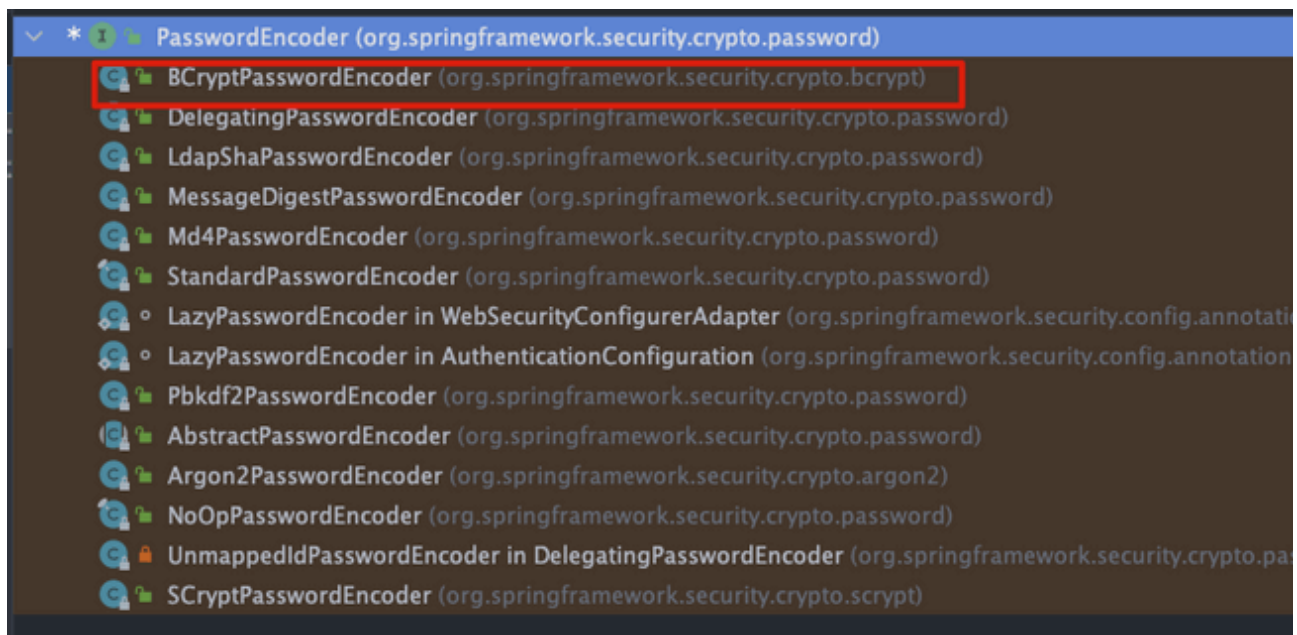
If we need to save sensitive data such as passwords in the database, we need to encrypt it before saving it.

Spring Security provides a variety of encryption algorithm implementations that are out of the box and very convenient. The interfaces of these encryption algorithm implementation classes are as follows `PasswordEncoder` . If you want to implement an encryption algorithm yourself, you also need to implement `PasswordEncoder` the interface.

`PasswordEncoder` The interface has a total of 3 methods that must be implemented.

```
1  public interface PasswordEncoder {
2      //
3      String encode(CharSequence var1);
4      //
5      boolean matches(CharSequence var1, String var2);
6      //                                     false
7      default boolean upgradeEncoding(String encodedPassword) {
8          return false;
9      }
10 }
```

java



The official recommendation is to use an encryption algorithm implementation class based on the bcrypt strong hash function.

How to gracefully change the encryption algorithm used by the system?

If we suddenly find during the development process that the existing encryption algorithm cannot meet our needs and need to be replaced with another encryption algorithm, what should we do at this time?

The recommended approach is `DelegatingPasswordEncoder` to be compatible with multiple different password encryption schemes to adapt to different business needs.

As the name suggests, `DelegatingPasswordEncoder` it is actually a proxy class, not a new encryption algorithm. What it does is to proxy the encryption algorithm implementation class mentioned above. After Spring Security 5.0, the default is to `DelegatingPasswordEncoder` encrypt passwords based on .

refer to

- Spring Technology Insider
- "Learn Spring in Depth from Scratch": <https://juejin.cn/book/685791186301639064>
- <http://www.cnblogs.com/wmyskxz/p/8820371.html>
- <https://www.journaldev.com/2696/spring-interview-questions-and-answers>
- <https://www.edureka.co/blog/interview-questions/spring-interview-questions/>

- <https://www.cnblogs.com/clwydjgs/p/9317849.html>
- <https://howtodoinjava.com/interview-questions/top-spring-interview-questions-with-answers/>
- <http://www.tomaszezula.com/2014/02/09/spring-series-part-5-component-vs-bean/>
- <https://stackoverflow.com/questions/34172888/difference-between-bean-and-autowired>

JavaGuide官方公众号 (微信搜索JavaGuide)



- 1、公众号后台回复“PDF”获取原创PDF面试手册
- 2、公众号后台回复“学习路线”获取Java学习路线最新版
- 3、公众号后台回复“开源”获取优质Java开源项目合集
- 4、公众号后台回复“八股文”获取Java面试真题+面经

Recently Updated 2025/1/16 11:54

Contributors: SnailClimb , Enda Lin , yellowgg , wangxiaowu , Kou Shuang , Snailclimb , shuang.kou , ipofss , Weiyang Zhu , guide , Sumeet Mahajan , cnsl , Tan Jiuding , TommyMerlin , anaer , huhongtao , kaka2634 , liubobo , fjut_shark , Guide , Mr.Hope , DaZuiZui , jun , 843294669 , aucub , imlee2021 , 1020325258 , qksuki

Copyright © 2025 Guide

