

SpringBoot / Concept

Container & Framework

Servlet Container

A servlet container is a component of the JEE framework that provides a runtime environment for Java servlets.

(A Java servlet is a Java class that can be used to extend the functionality of a web server.)

Servlet containers are responsible for managing the lifecycle of servlets, providing them with access to resources such as the web server's request and response objects, and handling security and threading issues.

Related Classes:

- [jakarta.servlet.Servlet](#)
- [jakarta.servlet.Filter](#)
- [jakarta.servlet.ServletContextListener](#)
- [jakarta.servlet.ServletContext](#)
- [jakarta.servlet.http.HttpServletRequest](#)
- [jakarta.servlet.http.HttpServletResponse](#)
- [jakarta.servlet.http.Cookie](#)

Spring IOC Container

Inversion of Control (IoC) is also known as dependency injection (DI). It is a process whereby objects define their dependencies,

that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

The container then injects those dependencies when it creates the bean.

This process is fundamentally the inverse of the bean itself, controlling the instantiation or location of its dependencies by using direct construction of classes,

hence the name Inversion of Control (IoC), or a mechanism such as the Service Locator pattern.

This is in contrast to traditional software design, where each object is responsible for creating and managing its own dependencies.

Related Classes:

- [org.springframework.beans.factory.BeanFactory](#)
- [org.springframework.context.ApplicationContext](#)
- [org.springframework.context.support.AbstractApplicationContext](#)
- [org.springframework.beans.factory.config.BeanDefinition](#)
- [org.springframework.context.ApplicationEvent](#)
- [org.springframework.core.io.ClassPathResource](#)

Spring MVC

Definition

Spring MVC is a framework built on top of the Spring Framework, designed specifically for building web applications using the Model-View-Controller (MVC) design pattern.

Spring MVC leverages the Spring Container (typically an ApplicationContext) to manage its components.

This means that all the beans (controllers, services, repositories, etc.) used in a Spring MVC application are managed by the Spring Container.

(Including managing the lifecycle of Spring MVC beans, ensuring they are properly initialized, configured, and destroyed.)

Key components of Spring MVC include:

DispatcherServlet	The front controller that handles all incoming HTTP requests and dispatches them to appropriate handlers.
Controller	Classes annotated with @Controller that handle HTTP requests and return views or data.
Handler Mapping	Determines which controller should handle a given request.
View Resolver	Resolves view names to actual views (like JSPs, Thymeleaf templates, etc.).
ModelAndView	Holds both the model data and the view name.

Model-View-Controller (MVC) design pattern

The Model-View-Controller (MVC) design pattern is a widely-used architectural pattern that separates an application into three main interconnected components: Model, View, and Controller.

This separation helps to manage complexity, improve code maintainability, and facilitate parallel development.

Model:

- **Purpose:** Represents the data and the business logic of the application. It directly manages the data, logic, and rules of the application.
- **Responsibilities:**
 - Encapsulates the application's data.
 - Contains the logic to retrieve and manipulate data.
 - Notifies the View of any data changes (often through observer patterns or similar mechanisms).
- **Example:** In a library system, the Book class that contains attributes like title, author, and methods to update these attributes.

View:

- **Purpose:** Represents the presentation layer of the application. It displays the data from the Model to the user and sends user commands to the Controller.
- **Responsibilities:**
 - Renders the data from the Model to the user.
 - Receives user input and forwards it to the Controller.
- **Example:** In a web application, an HTML page or a JSP that displays a list of books.

Controller:

- **Purpose:** Acts as an intermediary between the Model and the View. It handles user input, processes it (often by interacting with the Model), and returns the appropriate View.
- **Responsibilities:**
 - Receives input from the View.
 - Calls the appropriate methods on the Model to handle user actions.
 - Chooses the View to display based on the results of the Model's actions.
- **Example:** A servlet or a Spring MVC controller method that handles a form submission for adding a new book.

JPQL

Java Persistence Query Language (JPQL) is a powerful query language used in Java for querying entities stored in a relational database.

It is part of the Java Persistence API (JPA), which is a specification for managing relational data in Java applications.

JPQL is designed to work with entity objects rather than directly with database tables, making it a more object-oriented approach to querying databases.

Repository syntax

JPA 返回 findAll 没有数据时, List 为空[], 不是 null;

And	findByNameAndPwd	where name=? and pwd=?
or	findByNameOrsex	where name=? or sex=?
Is ,Equals	findById,findByIdEquals	where id=?
Between	findByIdBetween	where id between ? and ?
LessThan	findByIdLessThan	where id <? //适用于时间 deleteByXXLessThan(LocalDateTime ldt)
LessThanEquals	findByIdLessThanEquals	where id <=?
GreaterThan	findByIdGreaterThan	where id >?
GreaterThanOrEqual	findByIdGreaterThanOrEqual	where id >=?
After	findByIdAfter	where id >?
Before	findByIdBefore	where id <?
IsNull	findByNameIsNull	where name is null
isNotNull,NotNull	findByNameNotNull	where name is not null
like	findByNameLike	where name like ?
NotLike	findByNameNotLike	where name not like ?
startingWith	findByNameStartingWith	where name like "%?"
EndingWith	findByNameEndingWith	where name like "?%"
containing	findByNameContaining	where name like "%?%"
orderBy	findByIdOrderByXDesc	where id=? order by x desc
Not	findByNameNot	where name <>?
In	findByIdIn(Collection<?> c)	where id in (?)
NotIn	findByIdNotIn(Collection<?> c)	where id not in (?)
True	findByAaaTrue	where aaa=true
False	findByAaaFalse	where aaa=false
Ignorecase	findByNameIgnoreCase	where UPPER(name)=UPPER(?)
distinct	findDistinctPeopleByLastnameOrFirstname	findPeopleDistinctByLastnameOrFirstname
limit	findTopByOrderBySeatNumberAsc	findFirstByOrderBySeatNumberAsc
queryFirst10ByLastname	findTop3ByLastname	findFirst10ByLastname

delete sql return value maping

```
void, int, Integer
```

Special parameter handling

`Page<User> findByLastname(String lastname, Pageable pageable);` not allow null to pageable or sort, you can choose `Pageable.unpaged()` or `Sort.unsorted()`.

`Slice<User> findByLastname(String lastname, Pageable pageable);`

`List<User> findByLastname(String lastname, Sort sort);`

`List<User> findByLastname(String lastname, Pageable pageable);`

Limiting Query Results

`User findFirstByOrderByLastnameAsc();`

`User findTopByOrderByAgeDesc();`

`Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);`

`Slice<User> findTop3ByLastname(String lastname, Pageable pageable);`

`List<User> findFirst10ByLastname(String lastname, Sort sort);`

`List<User> findTop10ByLastname(String lastname, Pageable pageable);`

Spring Nullable Check

`@Nullable`

`User findByEmailAddress(@Nullable EmailAddress emailAdress);`

Using SpEL Expressions

@Query("select t from #{entityName} t where t.attribute = ?1") It inserts the entityName of the domain type associated with the given repository.

```
List<T> findAllByAttribute(String attribute);
```

@Query("select u from User u where u.firstname = :#{customer.firstname}") access parameter object
List<User> findUsersByCustomersFirstname(@Param("customer") Customer customer);

@Query("select u from User u where u.firstname = ?1 and u.firstname=?#[0] and u.emailAddress = ?#[principal.emailAddress]") access SecurityContext info
List<User> findByFirstnameAndCurrentUserWithCustomQuery(String firstname);

@Query("select o from BusinessObject o where o.owner.emailAddress like ?#{hasRole('ROLE_ADMIN') ? '%' : principal.emailAddress}")
List<BusinessObject> findBusinessObjectsForCurrentUser();

@Query("select u from User u where u.lastname like %:[0]%' and u.lastname like %:lastname%")
List<User> findByLastnameWithSpelExpression(@Param("lastname") String lastname);

@Query("select u from User u where u.firstname like %?#[escape([0])%' escape ?#[escapeCharacter()]) In combination with the escape clause of the like expression available in JPQL and standard SQL this allows easy cleaning of bind parameters.
List<User> findContainingEscaped(String namePart);

WHERE Clause

= equal //e.firstName = 'Bob'
< less than // e.salary < 100000
> greater than // e.salary > :sal
<= less than or equal // e.salary <= 100000
>= greater than or equal // e.salary >= :sal

LIKE evaluates if the two string match, '%' and '_' are valid wildcards, and ESCAPE character is optional
// e.firstName LIKE 'A%' OR e.firstName NOT LIKE '%._%' ESCAPE '.';

BETWEEN evaluates if the value is between the two values // e.firstName BETWEEN 'A' AND 'C'

IS NULL compares the value to null, databases may not allow or have unexpected results when using = with null // e.endDate IS NULL

IN evaluates if the value is contained in the list // e.firstName IN ('Bob', 'Fred', 'Joe')

```
// e.firstName IN (:name1, :name2, :name3)  
// e.firstName IN (:name1)  
// e.firstName IN :names  
// e.firstName IN (SELECT e2.firstName from Employee e2 WHERE e2.lastName = 'Smith')
```

```
// e.firstName = (SELECT e2.firstName from Employee e2 WHERE e2.id = :id)  
// e.salary < (SELECT e2.salary from Employee e2 WHERE e2.id = :id)  
// e.firstName = ANY (SELECT e2.firstName from Employee e2 WHERE e.id <> e2.id)  
// e.salary <= ALL (SELECT e2.salary from Employee e2)
```

Select Queries

SELECT **NEW** com.acme.reports.EmpReport(e.firstName, e.lastName, e.salary) FROM Employee e The NEW operator can

be used with the fully qualified class name to return data objects from a JPQL query

```
SELECT e FROM Employee e JOIN e.address a WHERE a.city = :city
```

```
SELECT e FROM Employee e JOIN e.projects p JOIN e.projects p2 WHERE p.name = :p1 and p2.name = :p2
```

The JOIN clause allows any of the object's relationships to be joined into the query so they can be used in the WHERE clause.

JOIN does not mean the relationships will be fetched, unless the FETCH option is included.

JPA does not support the SQL UNION, INTERSECT and EXCEPT operations. Some JPA providers may support these.

SELECT e FROM Employee e JOIN FETCH e.address The FETCH option can be used on a JOIN to fetch the related objects in a single query.

```
SELECT e FROM Employee e LEFT JOIN e.address a ON a.city = :city
```

```
SELECT e FROM Employee e ORDER BY UPPER(e.lastName)
```

```
SELECT AVG(e.salary), e.address.city FROM Employee e GROUP BY e.address.city ORDER BY AVG(e.salary)
```

```
SELECT AVG(e.salary), e.address.city FROM Employee e GROUP BY e.address.city HAVING AVG(e.salary) > 100000
```

Parameters

```
SELECT e FROM Employee e WHERE e.firstName = :first and e.lastName = :last     customs parameter name
```

```
SELECT e FROM Employee e WHERE e.firstName = ?1 and e.lastName = ?2     sequential parameters
```

Return Type

```
Streamable<Person> findByFirstnameContaining(String firstname);
```

```
                  // Streamable<Person> result =
```

```
repository.findByFirstnameContaining("av").and(repository.findByLastnameContaining("ea"));
```

Streamable<Person> findByLastnameContaining(String lastname); You can express nullability constraints for repository methods by using Spring Framework's nullability annotations.

They provide a tooling-friendly approach and opt-in null checks during runtime

JPQL special operators

INDEX the index of the ordered List element, only supported with @OrderColumn // SELECT

```
ToDo FROM Employee e JOIN e.toDoList toDo WHERE INDEX(toDo) = 1
```

KEY the key of the Map element // SELECT p FROM

```
Employee e JOIN e.priorities p WHERE KEY(p) = 'high'
```

SIZE the size of the collection relationships, this evaluates to a sub-select // SELECT e

```
FROM Employee e WHERE SIZE(e.managedEmployees) < 2
```

IS EMPTY, IS NOT EMPTY evaluates to true if the collection relationship is empty, this evaluates to a sub-select // SELECT e FROM Employee e WHERE e.managedEmployees IS EMPTY

MEMBER OF, NOT MEMBER OF evaluates to true if the collection relationship contains the value, this evaluates to a sub-select // SELECT e FROM Employee e WHERE 'write code' MEMBER OF e.responsibilities

TYPE the inheritance discriminator value

```
// SELECT p FROM Project p WHERE TYPE(p) = LargeProject
```

TREAT treat (cast) the object as its subclass value (JPA 2.1 draft)

```
// SELECT e FROM Employee JOIN TREAT(e.projects as LargeProject) p WHERE p.budget > 1000000
```

FUNCTION call a database function (JPA 2.1 draft)

```
// SELECT p FROM Phone p WHERE FUNCTION('TO_NUMBER', p.areaCode) > 613
```

JPQL supported functions

```
- subtraction // e.salary - 1000
+ addition // e.salary + 1000
* multiplication // e.salary * 2
/ division // e.salary / 2
ABS absolute value // ABS(e.salary - e.manager.salary)
CASE defines a case statement // CASE e.status WHEN 0 THEN 'active' WHEN 1 THEN 'consultant' ELSE
'unknown' END
COALESCE evaluates to the first non null argument value // COALESCE(e.salary, 0)

CURRENT_DATE the current date on the database // CURRENT_DATE
CURRENT_TIME the current time on the database // CURRENT_TIME
CURRENT_TIMESTAMP the current date-time on the database // CURRENT_TIMESTAMP

LOCATE the index of the string within the string, optionally starting at a start index // LOCATE('-', e.lastName)
MOD computes the remainder of dividing the first integer by the second // MOD(e.hoursWorked / 8)
NULLIF returns null if the first argument equal to the second argument, otherwise returns the first argument //
NULLIF(e.salary, 0)
SQRT computes the square root of the number // SQRT(o.result)

LENGTH the character/byte length of the character or binary value // LENGTH(e.lastName)
LOWER convert the string value to lower case // LOWER(e.lastName)
CONCAT concatenates two or more string values // CONCAT(e.firstName, ' ', e.lastName)
SUBSTRING the substring from the string, starting at the index, optionally with the substring size //
SUBSTRING(e.lastName, 0, 2)
TRIM trims leading, trailing, or both spaces or optional trim character from the string
// TRIM(TRAILING FROM e.lastName), TRIM(e.lastName), TRIM(LEADING '-' FROM e.lastName)
UPPER convert the string value to upper case // UPPER(e.lastName)
```

SELECT COUNT(e) FROM Employee e Aggregation functions can include summary information on a set of objects.
These include MIN, MAX, AVG, SUM, COUNT.

Proxy Modes

Static Proxy Mode

A manually created proxy class that controls access to another object (the target object).

It typically implements the same interface as the target object and delegates method calls, optionally adding behavior before or after method invocation.

DataService

```
public interface DataService {
    void saveData(String data);
    String retrieveData();
}

DataServiceImpl
public class DataServiceImpl implements DataService {

    @Override
    public void saveData(String data) {
```

```

        System.out.println("Saving data: " + data);
    }

    @Override
    public String retrieveData() {
        String data = "Sample Data";
        System.out.println("Retrieving data: " + data);
        return data;
    }
}


```

DataServiceProxy

```

public class DataServiceProxy implements DataService {

    private DataService dataService;

    public DataServiceProxy(DataService dataService) {
        this.dataService = dataService;
    }

    @Override
    public void saveData(String data) {
        System.out.println("Before saving data");
        dataService.saveData(data);
        System.out.println("After saving data");
    }

    @Override
    public String retrieveData() {
        System.out.println("Before retrieving data");
        String data = dataService.retrieveData();
        System.out.println("After retrieving data");
        return data;
    }
}

```

StaticProxyExample

```

public class StaticProxyExample {

    public static void main(String[] args) {
        DataService realService = new DataServiceImpl();
        DataService proxy = new DataServiceProxy(realService);

        // Using the proxy
        proxy.saveData("Proxy Example Data");
        String retrievedData = proxy.retrieveData();
        System.out.println("Retrieved data: " + retrievedData);
    }
}

```

Dynamic Proxy Mode

JDK Dynamic Proxies

Characteristics:

Interface-Based	JDK dynamic proxies can only proxy interfaces. This means that the target class must implement at least one interface.
-----------------	---

Standard Java Library	JDK dynamic proxies are part of the standard Java library (java.lang.reflect.Proxy).
-----------------------	--

Performance	Typically, JDK dynamic proxies are slightly slower than CGLIB proxies due to their reflective nature.
-------------	---

Implementation:

JDK dynamic proxies use [reflection](#) to create proxy instances at runtime.

The proxy instance implements the specified interfaces and delegates method calls to an [InvocationHandler](#).

JDK dynamic proxies and CGLIB proxies **are both used at runtime** rather than during the class loading phase of the JVM's lifecycle.

These proxies are generated after the initial class loading phase of the JVM.

Once generated, they **are treated as normal classes** and loaded by the JVM's class loader alongside other classes in the application.

CGLIB Proxies

Characteristics:

Subclass-Based CGLIB proxies **create subclasses of the target class** and override its methods. This means that the target class does not need to implement any interfaces.

Spring uses CGLIB proxies when **the target object does not implement interfaces**.

External Library CGLIB proxies rely on the **CGLIB library**, which Spring includes as a dependency.

Performance CGLIB proxies are typically faster than JDK dynamic proxies because they directly generate bytecode for the proxy classes.

Implementation:

CGLIB proxies use **the bytecode generation library** to create a subclass of the target class at runtime.

The subclass overrides the methods to add the additional behavior.

JDK dynamic proxies and CGLIB proxies **are both used at runtime** rather than during the class loading phase of the JVM's lifecycle.

These proxies are generated after the initial class loading phase of the JVM.

Once generated, they **are treated as normal classes** and loaded by the JVM's class loader alongside other classes in the application.

SPEL

Spring Expression Language (SpEL) is a powerful expression language integrated within the Spring framework, designed to evaluate expressions dynamically at runtime.

It is similar to other expression languages such as OGNL (Object-Graph Navigation Language), MVEL (MVFLEX Expression Language), and JSP EL (JavaServer Pages Expression Language),

but it is specifically tailored to work with the features and capabilities of the Spring framework.

Related Classes:

`org.springframework.expression.EvaluationContext`

字面值

<code>#{}5</code>	整数
<code>#{}3.1415</code>	浮点数
<code>#{}9.87E4</code>	科学计数法
<code>#{'hello'}</code> 或 <code>#{"hello"}</code>	字符串 (使用单引号/双引号, 皆可)
<code>#{}true</code>	boolean 值
<code>{1,3,5,7}</code>	数组
<code>{{1,3,5,7},{0,2,4,6}}</code>	二维数组
<code>{"name": "name", password: "111"}</code>	对象

引用 context 变量

```
{#a}      使用 context 中的变量  
#{workersHolder.salaryByWorkers['John']} map 元素  
#{workersHolder.workers[0]}      list 元素  
#{workersHolder.workers.size()} list 元素
```

引用 Bean 并使用其属性与方法

```
#{a}      通过 ID 引用 bean (a 为 bean 的 id)  
#{a.b}    使用 bean 的属性  
#{a.c()}  使用 bean 的方法  
#{a.c().toUpperCase()} 可链式调用  
#{a.c()?.toUpperCase()} 通过 .? (类型安全的运算符) 避免空指针(NullPointerException)  [ a.c() 存在时才使用 toUpperCase() ]
```

使用类作用域

```
#T(java.lang.Math).PI      通过 T() 可以访问类作用域的方法和常量  
#T(java.lang.Math).random() 通过 T() 获取方法
```

运算符

#T(java.lang.Math).PI*circle.r^2}	计算符, 如 *、+、- ...
#{a.b==100} #{a.b eq 100}	比较运算符
#{a.b + 'accc'}	拼接字符串
#{scoreboard.score > 1000 ? "winner" : "loser"}	三元运算符
#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9._%+-]+\.\com'}	正则表达式
#{250 > 200 and 200 < 4000}	
#{400 > 300 or 150 < 100}	
#{250 > 200 && 200 < 4000}	
#{400 > 300 150 < 100}	
#{!true}	
#{not true}	
#{37 % 10}	

```
#{20 - 1}
#{'String1 ' + 'string2'}
#{2 > 1 ? 'a' : 'b'}
#{'100' matches '\d+'}
```

集合

`#{{jukebox.song[4].title}}` 通过`[]`获取数组元素

`members.[nationality == '中国']` 通过`[]`获取 map 元素

`members.! [key + '-' + value]` map 转 list

`#{'this is a test' [3]}` **[]也可用于字符串**

`#{{jukebox.songs.? [artist eq 'Aerosmith']}}` **.?[]**(查询运算符)对集合过滤【检查歌曲的 artist 属性是不是等于 Aerosmith，是的话放入新的集合。】

`.^[]`和`.[$[]]`，它们分别用来在集合中查询第一个匹配项和最后一个匹配项

投影运算符 (`.![]`)，它会从集合的每个成员中选择特定的属性放到另外一个集合中

Spring Circular Dependency

In Spring, circular dependencies occur **when two or more beans are interdependent on each other**, creating a cycle. For example, Bean A depends on Bean B, and Bean B depends on Bean A. This situation can lead to issues during the bean initialization process in the Spring container.

AbstractAutowireCapableBeanFactory

`createBeanInstance(String beanName, Class<?> beanClass)`

During the instantiation phase, Spring **creates an instance of the bean** using its constructor by reflection.

The fields `singletonObjects`, `singletonFactories`, `earlySingletonObjects` and `registeredSingletons` **is not directly involved in this phase**.

`addSingletonFactory(String beanName, ObjectProviderFactory<Object> factory) (Optional)`

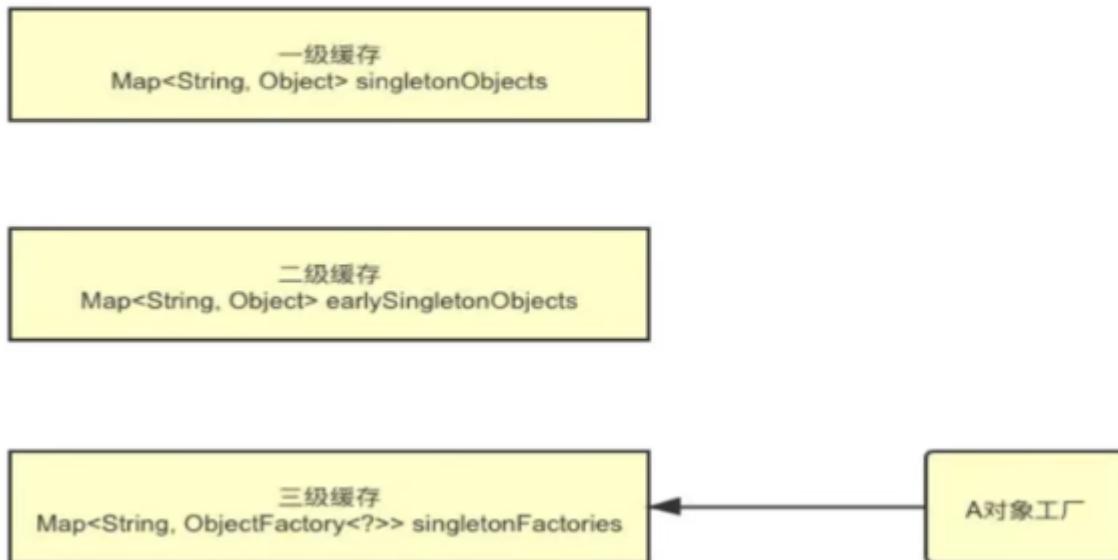
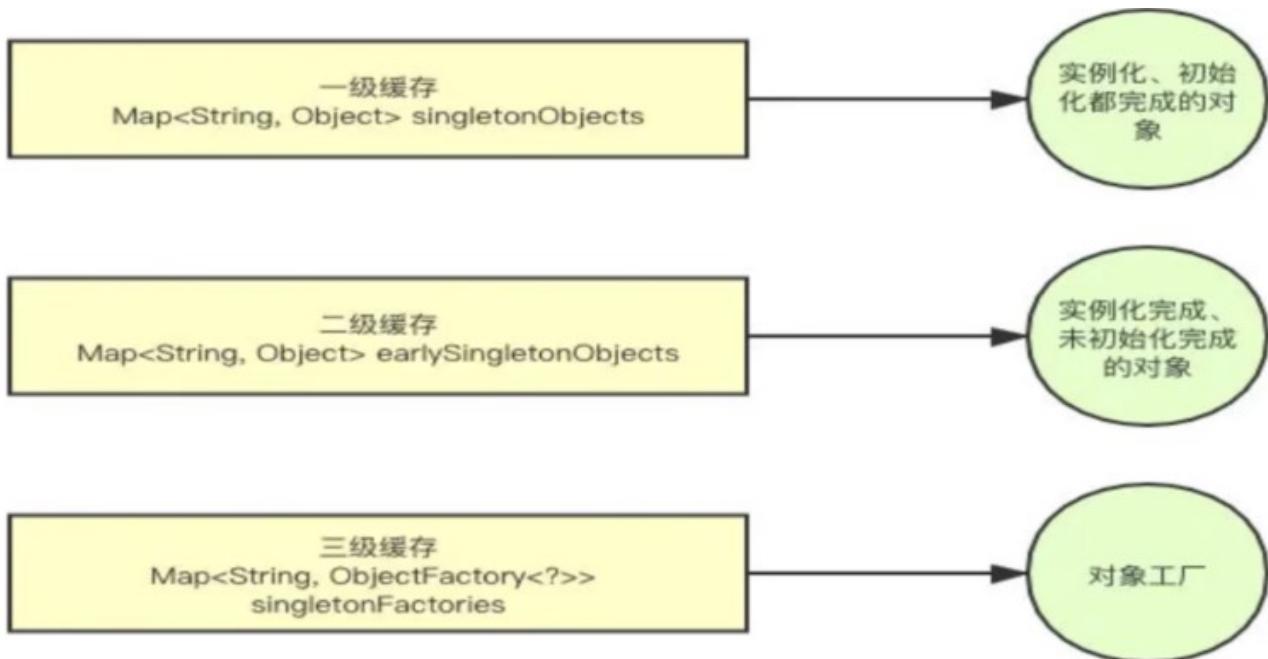
Registers **a factory method** into `singletonFactories` and add current bean into `registeredSingletons`.

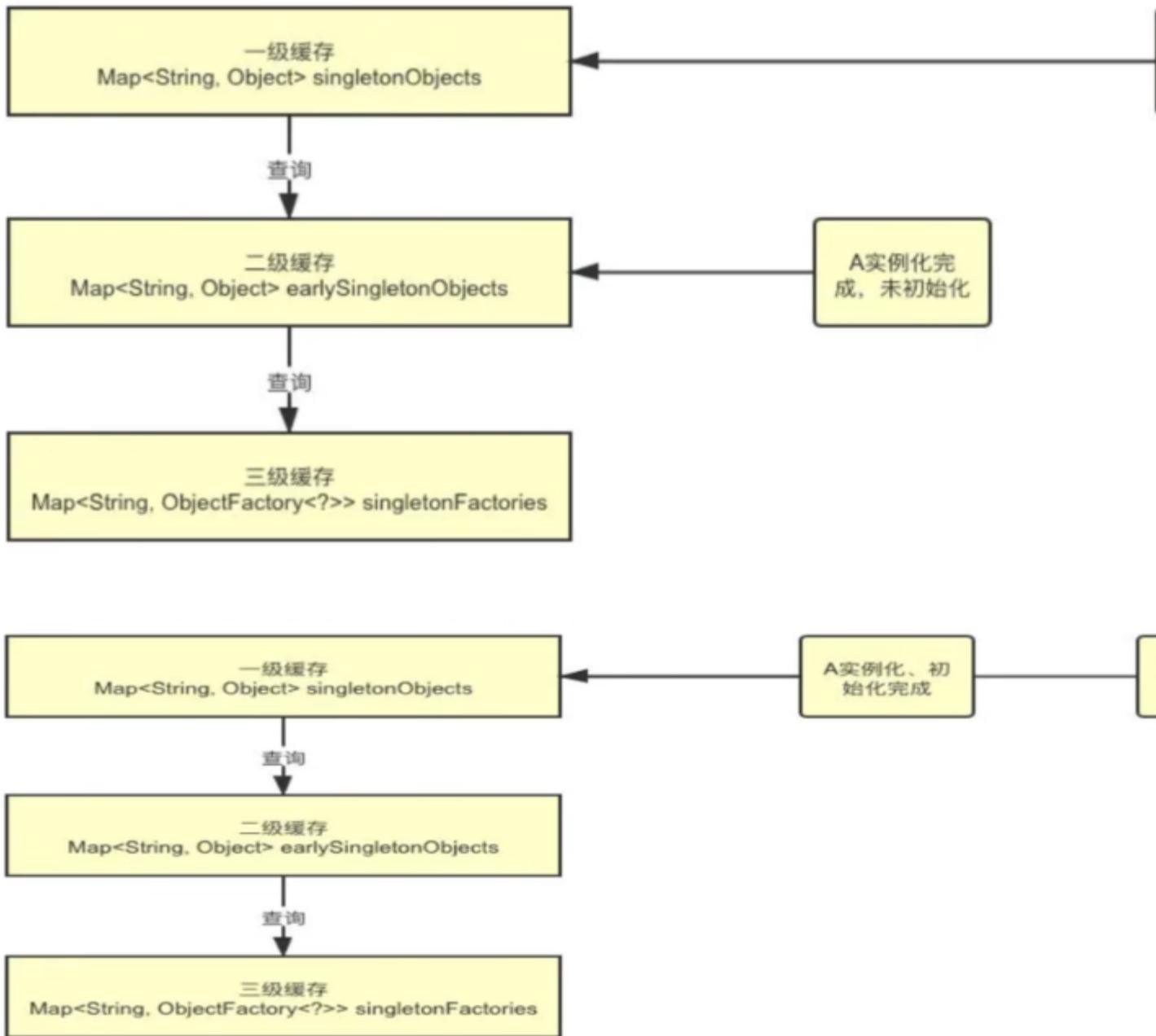
The factory method **attempts to get an early reference to the bean** for partial creation.

This early reference could be:

A partially configured instance stored in `earlySingletonObjects`.

Created in a specific way based on internal logic.





`populateBean(String beanName, Object bean)`

It's responsible for **injecting dependencies into the bean instance** based on the bean definition.

Handle circular dependency:

- 1) Circular Dependency Check

If the dependency is found in `singletonsCurrentlyInCreation`, it indicates a circular dependency.

- 2) Early Reference Usage (Optional)

If a circular dependency is detected, Spring might check the `earlySingletonObjects` map.

If the dependent bean (A) is present in `earlySingletonObjects` (because it was early initialized), Spring can use this reference for dependency injection even though A might not be fully configured yet.

This avoids creating a proxy object in this specific scenario.

- 3) Fallback to Proxy Creation

If the dependent bean isn't found in `earlySingletonObjects`, Spring falls back to creating a proxy object using the factory retrieved from `singletonFactories`.

This proxy allows for dependency injection to proceed even though A isn't fully formed yet.

initializeBean(String beanName, Object bean)

It's responsible for invoking any bean lifecycle methods like `@PostConstruct` and applying BeanPostProcessors (custom logic for bean initialization).

Once a bean is fully configured and initialized, `initializeBean` stores it in the `singletonObjects` map for future retrieval by dependency injection or other mechanisms.

Spring IOC

Spring Bean Creation Process

Spring Boot leverages a series of interfaces and classes to create, configure, and manage beans within your application.

Here's a breakdown of the entire process, incorporating the mentioned classes and their roles:

Bean Definition and Instantiation

You define beans either through annotations (`@Component`, `@Service`, etc.) or XML configuration files.

These definitions provide metadata about the bean, including its class, dependencies, scope (singleton, prototype), etc.

Class `AbstractApplicationContext` provides mechanisms for loading bean definitions, instantiating beans, and managing their dependencies.

Once the bean definition is retrieved from the `BeanFactory`, Spring executes all registered `BeanFactoryPostProcessor` implementations,

and then Spring Boot uses reflection to create an instance of the bean's class.

For regular beans, Spring uses the `BeanFactory` to create an instance of the bean.

If a bean definition corresponds to a `FactoryBean`, Spring first creates an instance of the `FactoryBean` using the `BeanFactory`.

After the `FactoryBean` instance is created, Spring calls the `getObject` method of the `FactoryBean` to create the actual bean instance that will be exposed to the application.

Constructor arguments are resolved by finding matching beans in the registry or using constructor injection.

Depending on the scope (singleton, prototype, etc.), Spring decides whether to create a new instance or return an existing one.

Property Population

Once the bean is instantiated, Spring populates the bean's properties.

This includes setting values and references to other beans.

Dependency injection occurs at this stage, where Spring injects dependencies as specified in the configuration metadata.

Aware Interface Callbacks

If the bean implements any of the following Aware interfaces, Spring calls the corresponding methods to pass relevant information to the bean:

- `BeanNameAware`
- `BeanFactoryAware`
- `ApplicationContextAware`

Initialization

BeanPostProcessor Pre-Initialization

Spring then applies any BeanPostProcessor implementations registered in the context.

The `postProcessBeforeInitialization` method of each `BeanPostProcessor` is called.

This allows for custom modification of new bean instances before any initialization callbacks.

Initialization

`@PostConstruct`

Spring checks for methods annotated with `@PostConstruct`.

These methods are executed **after bean properties have been set** and **before the afterPropertiesSet method of the InitializingBean interface..**

`InitializingBean`

If the bean implements the `InitializingBean` interface, Spring calls the `afterPropertiesSet` method.

`Custom init-method`

If a custom init-method is specified in the bean definition, this method is called after

`InitializingBean.afterPropertiesSet()`.

`@Configuration`

```
public class AppConfig {
```

```
    @Bean(initMethod = "customInit")
```

```
    public MyBean myBean() {
```

```
        return new MyBean();
```

```
    }
```

```
}
```

BeanPostProcessor Post-Initialization

Spring applies the `postProcessAfterInitialization` method of each `BeanPostProcessor`.

This step allows for further customization of new bean instances after initialization callbacks have been invoked.

Ready for Use

At this point, the bean is **fully initialized** and ready for use.

The container can now inject this bean into other beans as dependencies or make it available for application use.

Destruction (if applicable)

When the application context is closed, Spring performs cleanup of beans.

If the bean implements the `DisposableBean` interface, Spring calls the `destroy` method.

Alternatively, if a custom `destroy`-method is specified, it is invoked.

This step is crucial for releasing resources and performing any necessary cleanup.

Spring MVC Request Route

Request Handling Process in Spring Boot

Client Request

The client (e.g., a web browser) sends an HTTP request to the server.

Embedded Server

Spring Boot typically uses an embedded server like Tomcat, Jetty, or Undertow, which listens for incoming HTTP requests.

Filters

The request first passes through the `chain of servlet filters`. Filters are executed in the order they are configured.

Each filter can perform pre-processing tasks, such as logging, authentication, or modifying the request.

After the request processing by filters, the request is passed along the filter chain.

DispatcherServlet

The request is received by the `DispatcherServlet`, which is automatically configured by Spring Boot as the front controller.

Static Resource Handling

If the request is **for a static resource** (like HTML, CSS, JS, or images), Spring Boot's `built-in static resource handlers` serve the resource from specific locations on the classpath or filesystem.

Default locations include `/static`, `/public`, `/resources`, and `/META-INF/resources`.

The `ResourceHttpRequestHandler` serves static resources from the classpath and the filesystem.

Handler Mapping

If the request is not for a static resource, the DispatcherServlet consults HandlerMapping implementations to find the appropriate controller to handle the request.

Handler Interceptors (Pre-Processing)

Before the handler (controller) is invoked, the registered HandlerInterceptor instances are executed in the order they are configured.

The preHandle method of each interceptor is called. If any interceptor returns false, the request processing stops, and the response is returned.

Handler Adapter

The DispatcherServlet uses a HandlerAdapter to invoke the controller method.

The controller method mapped to the request URL is invoked.

Controllers are annotated with @RestController or @Controller and use request mapping annotations (@RequestMapping, @GetMapping, etc.).

For traditional MVC applications, the controller returns a ModelAndView object or simply a view name.

In RESTful services, the controller returns data directly, often as JSON or XML.

This adapter allows different types of handler methods to be executed uniformly.

View Resolver

For MVC applications, the DispatcherServlet consults a ViewResolver to determine which view to render.

The view resolver maps the logical view name to an actual view implementation.

View Rendering

The view is rendered with the model data, and the generated content is written to the response.

Response Sent

The DispatcherServlet sends the response back to the client.

Spring MVC interceptor chain

HandlerInterceptor Interface:

Interceptors in Spring MVC are implemented by the HandlerInterceptor interface.

Configuration

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoggingInterceptor()).addPathPatterns("/**");
    }
}
```

Spring Filter chain

Servlet Filters

Filters are Java objects that can be used to intercept HTTP requests and responses in a web application.

They are defined in the Java Servlet specification and are not specific to Spring, though they are often used within Spring applications.

Related Classes:

`jakarta.servlet.Filter`
`org.springframework.boot.web.servlet.FilterRegistrationBean`

Filter Chain

A filter chain is a series of filters that are applied to a request in a specific order.

The FilterChain interface is used to pass the request and response to the next filter in the chain.

Related Classes:

[org.springframework.web.servlet.HandlerInterceptor](#)
[org.springframework.web.servlet.config.annotation.WebMvcConfigurer](#)

MicroService

Structure

A microservice architecture (MSA) structures an application as a collection of **small, autonomous services** modeled around a business domain.

Here's an overview of the typical components and structure of a microservice architecture:

Comparison Table

Feature / Aspect	Dubbo	Spring Cloud	Spring Cloud Alibaba
Primary Focus	RPC framework for microservices	Comprehensive microservice toolkit	Alibaba Cloud middleware integration
Service Discovery	Zookeeper, Nacos	Eureka, Consul, Zookeeper	Nacos
Configuration Management	Nacos, Apollo	Spring Cloud Config, Consul	Nacos
Communication Style	RPC (binary)	HTTP (REST), messaging (Kafka, etc.)	RPC (Dubbo), HTTP (REST)
Load Balancing	Built-in (multiple strategies)	Ribbon (client-side)	Nacos
Circuit Breaker	Built-in	Hystrix, Resilience4j	Sentinel
API Gateway	No built-in solution	Spring Cloud Gateway, Zuul	Spring Cloud Gateway
Messaging	N/A	Kafka, RabbitMQ	RocketMQ
Distributed Tracing	SkyWalking	Spring Cloud Sleuth, Zipkin	Alibaba Cloud Tracing
Extensibility	Highly extensible	Highly extensible	Highly extensible with Alibaba products
Integration with Spring	Moderate	Full integration	Full integration
Use Case Suitability	Performance-critical RPC services	General microservice architectures	Microservices on Alibaba Cloud

RPC

Comparison of RPC and REST

Feature	RPC	REST
Communication Style	Function call	Resource-based
Protocol	Custom or specific protocols (e.g., gRPC)	HTTP/1.1, HTTP/2
Data Format	Often binary (e.g., Protocol Buffers)	JSON, XML
State Management	Stateless or stateful	Stateless
Use Case	Performance-critical, real-time	General-purpose web services
Scalability	Generally good, with some protocols offering load balancing and more	Very good, naturally scalable due to statelessness
Ease of Use	Requires more setup (IDLs, service definitions)	Simpler with HTTP methods and URLs
Error Handling	Custom error handling mechanisms	HTTP status codes

RPC (Remote Procedure Call)

RPC is a protocol **that one program can use to request a service** from a program located on another computer in a network.

It abstracts the complexity of the network communication, making remote interactions look like local method calls.

REST (Representational State Transfer)

REST is **an architectural style**, not a strict protocol. It focuses on **resources** and **how to manipulate them**.

Imagine resources like data entities (users, products) on a server.

Clients use standard HTTP methods (GET, POST, PUT, DELETE) to interact with these resources.

RESTful

This term refers to an API **that adheres to the principles of REST**. A RESTful API uses HTTP methods, standard status codes, and focuses on resources.

Versions

Please make a note that current spring framework version(5.3.25) has become vulnerable and hence we need to upgrade it to a safer version(6.0.4) in upcoming release.

This is a major version upgrade which requires Java17 as baseline.

We will upgrade this library and share the new ms-parent version in few days.

Please make a note that all the consumers who want to use new ms-parent has to upgrade to Java17, once this new version will be provided support there will be no any BD fix available on older ms-parent version.

References:

releases

<https://github.com/spring-projects/spring-framework/releases>

What's New in Spring Framework 6.x

<https://github.com/spring-projects/spring-framework/wiki/What%27s-New-in-Spring-Framework-6.x>

Compatibility

Spring Boot Version	Spring Framework Version	Compatible Spring Cloud Version
1.5.22.RELEASE	4.3.25.RELEASE	Edgware
2.0.9.RELEASE	5.0.13.RELEASE	Finchley.RELEASE, Finchley.SR1, Finchley.SR4
2.1.18.RELEASE	5.1.18.RELEASE	Greenwich.RELEASE, Greenwich.SR1, Greenwich.SR6
2.2.13.RELEASE	5.2.11.RELEASE	Hoxton.RELEASE, Hoxton.SR1, Hoxton.SR12
2.3.12.RELEASE	5.2.15.RELEASE	Hoxton.RELEASE, Hoxton.SR1, Hoxton.SR12
2.4.13	5.3.19	2020.0.0, 2020.0.6 (Ilford)
2.5.14	5.3.20	2020.0.0, 2020.0.6 (Ilford)
2.6.12	5.3.23	2021.0.0, 2021.0.9 (Jubilee)
2.7.18	5.3.27	2021.0.0, 2021.0.9 (Jubilee)
3.0.13	6.0.4	2022.0.0, 2022.0.5 (Oak)
3.1.12	6.0.12	2022.0.0, 2022.0.5 (Oak)
3.2.7	6.0.12	2022.0.0, 2022.0.5 (Oak)
3.3.1	6.1.3	2023.0.0, 2023.0.5 (Kilburn)
3.1.0 or later	6.1.0 or later	2024.0.0

Spring 6.0

<https://github.com/spring-projects/spring-framework/wiki/What%27s-New-in-Spring-Framework-6.x>

JDK 17+ and Jakarta EE 9+ Baseline

- Entire framework codebase based on [Java 17 source code level](#) now.
- Migration from [javax](#) to [jakarta](#) namespace for Servlet, JPA, etc.
- Runtime compatibility with [Jakarta EE 9](#) as well as [Jakarta EE 10 APIs](#).
- Compatible with latest web servers: [Tomcat 10.1](#), [Jetty 11](#), [Undertow 2.3](#).
- Early compatibility with [virtual threads](#) (in preview as of JDK 19).

General Core Revision

- Upgrade to ASM 9.4 and Kotlin 1.7.
- Complete CGLIB fork with support for capturing CGLIB-generated classes.
- Comprehensive foundation for [Ahead-Of-Time transformations](#).
- First-class support for [GraalVM](#) native images (see [related Spring Boot 3 blog post](#)).

Core Container

- Basic bean property determination without `java.beans.Introspector` by default.
- AOT processing support in `GenericApplicationContext` (`refreshForAotProcessing`).
- Bean definition transformation based on pre-resolved constructors and factory methods.
- Support for early proxy class determination for AOP proxies and configuration classes.
- `PathMatchingResourcePatternResolver` uses NIO and module path APIs for scanning, enabling support for classpath scanning within a GraalVM native image and within the Java module path, respectively.
- `DefaultFormattingConversionService` supports ISO-based default `java.time` type parsing.

Spring MVC

- [PathPatternParser](#) used by default (with the ability to opt into [PathMatcher](#)).
- Removal of outdated Tiles and FreeMarker JSP support.

General Web Revision

- [HTTP interface client](#) based on `@HttpExchange` service interfaces.
- Support for [RFC 7807 problem details](#).
- [Unified HTTP status code](#) handling.
- Support for Jackson 2.14.
- Alignment with Servlet 6.0 (while retaining runtime compatibility with Servlet 5.0).

Data Access and Transactions

- Support for predetermining JPA managed types (for inclusion in AOT processing).
- JPA support for [Hibernate ORM 6.1](#) (retaining compatibility with Hibernate ORM 5.6).
- Upgrade to [R2DBC 1.0](#) (including R2DBC transaction definitions).
- Aligned data access exception translation between JDBC, R2DBC, JPA and Hibernate.
- Removal of JCA CCI support.

Observability

Direct Observability instrumentation with [Micrometer Observation](#) in several parts of the Spring Framework. The `spring-web` module now requires `io.micrometer:micrometer-observation:1.10+` as a compile dependency.

- `RestTemplate` and `WebClient` are instrumented to produce HTTP client request observations.
- Spring MVC can be instrumented for HTTP server observations using the new `org.springframework.web.filter.ServerHttpObservationFilter`.
- Spring WebFlux can be instrumented for HTTP server observations using the new `org.springframework.web.filter.reactive.ServerHttpObservationFilter`.
- Integration with Micrometer [Context Propagation](#) for Flux and Mono return values from controller methods.

Testing

- Support for testing AOT-processed application contexts on the JVM or within a GraalVM native image.

- Integration with HtmlUnit 2.64+ request parameter handling.
- Servlet mocks (MockHttpServletRequest, MockHttpSession) are based on Servlet API 6.0 now.
- New MockHttpServletRequestBuilder.setRemoteAddress() method.
- The four abstract base test classes for JUnit 4 and TestNG no longer declare listeners via @TestExecutionListeners and instead now rely on registration of default listeners.

Spring 5.0

<https://github.com/spring-projects/spring-framework/wiki/What%27s-New-in-Spring-Framework-5.x>

JDK 8+ and Java EE 7+ Baseline

- Entire framework codebase based on **Java 8** source code level now.
 - Improved readability through inferred generics, lambdas, etc.
 - Conditional support for Java 8 features now in straight code.
- Full compatibility with JDK 9 for development and deployment.
 - On classpath as well as module path (with stable automatic module names).
 - Framework build and test suite passes on JDK 9 (runs on JDK 8 by default).
- Java EE 7 API level required in Spring's corresponding features now.
 - Servlet 3.1, Bean Validation 1.1, JPA 2.1, JMS 2.0
 - Recent servers: e.g. Tomcat 8.5+, Jetty 9.4+, WildFly 10+
- Compatibility with Java EE 8 API level at runtime.
 - Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0
 - Tested against Tomcat 9.0, Hibernate Validator 6.0, Apache Johnzon 1.1

Removed Packages, Classes and Methods

- Package beans.factory.access (BeanFactoryLocator mechanism).
- Package jdbc.support.nativejdbc (NativeJdbcExtractor mechanism).
- Package mock.staticmock removed from spring-aspects module.
 - No support for AnnotationDrivenStaticEntityMockingControl anymore.
- Packages web.view.tiles2 and orm.hibernate3/hibernate4 dropped.
 - Minimum requirement: Tiles 3 and Hibernate 5 now.
- Dropped support: Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava.
 - Recommendation: Stay on Spring Framework 4.3.x for those if needed.
- Many deprecated classes and methods removed across the codebase.
 - A few compromises made for commonly used methods in the ecosystem.

General Core Revision

- JDK 8+ enhancements:
 - Efficient method parameter access based on Java 8 reflection enhancements.
 - Selective declarations of Java 8 default methods in core Spring interfaces.
 - Consistent use of JDK 7 Charset and StandardCharsets enhancements.
- JDK 9 compatibility:
 - Avoiding JDK APIs which are deprecated in JDK 9 wherever possible.
 - Consistent instantiation via constructors (with revised exception handling).
 - Defensive use of reflection against core JDK classes.
- Non-null API declaration at the package level:
 - Nullable arguments, fields and return values explicitly annotated with @Nullable.
 - Primarily for use with IntelliJ IDEA and Kotlin, but also Eclipse and FindBugs.
 - Some Spring APIs are not tolerating null values anymore (e.g. in StringUtils).
- Resource abstraction provides isFile indicator for defensive getFile access.
 - Also features NIO-based readableChannel accessor in the Resource interface.

- File system access via NIO.2 streams (no FileInputStream/OutputStream used anymore).
- Spring Framework 5.0 comes with its own Commons Logging bridge out of the box:
 - spring-jcl instead of standard Commons Logging; still excludable/overridable.
 - Autodetecting Log4j 2.x, SLF4J, JUL (java.util.logging) without any extra bridges.
- spring-core comes with ASM 6.0 (next to CGLIB 3.2.5 and Objenesis 2.6).

Core Container

- Support for any `@Nullable` annotations as indicators for optional injection points.
- Functional style on GenericApplicationContext/AnnotationConfigApplicationContext
 - Supplier-based bean registration API with bean definition customizer callbacks.
- Consistent detection of transaction, caching, async annotations on interface methods.
 - In case of CGLIB proxies.
- XML configuration namespaces streamlined towards unversioned schemas.
 - Always resolved against latest xsd files; no support for deprecated features.
 - Version-specific declarations still supported but validated against latest schema.
- Support for candidate component index (as alternative to classpath scanning).

Spring Web MVC

- Full Servlet 3.1 signature support in Spring-provided Filter implementations.
- Support for Servlet 4.0 PushBuilder argument in Spring MVC controller methods.
- MaxUploadSizeExceededException for Servlet 3.0 multipart parsing on common servers.
- Unified support for common media types through MediaTypeFactory delegate.
 - Superseding use of the Java Activation Framework.
- Data binding with immutable objects (Kotlin / Lombok / `@ConstructorProperties`)
- Support for the JSON Binding API (with Eclipse Yasson or Apache Johnzon as an alternative to Jackson and GSON).
- Support for Jackson 2.9.
- Support for Protobuf 3.
- Support for Reactor 3.1 Flux and Mono as well as RxJava 1.3 and 2.1 as return values from Spring MVC controller methods targeting use of the new reactive WebClient (see below) or Spring Data Reactive repositories in Spring MVC controllers.
- New ParsingPathMatcher alternative to AntPathMatcher with more efficient parsing and [extended syntax](#).
- `@ExceptionHandler` methods allow RedirectAttributes arguments (and therefore flash attributes).
- Support for ResponseStatusException as a programmatic alternative to `@ResponseStatus`.
- Support script engines that do not implement Invocable via direct rendering of the script provided using ScriptEngine#eval(String, Bindings), and also i18n and nested templates in ScriptTemplateView via the new RenderingContext parameter.
- Spring's FreeMarker macros (`spring.ftl`) use HTML output formatting now (requiring FreeMarker 2.3.24+).

Spring WebFlux

- New `spring-webflux` module, an alternative to `spring-webmvc` built on a [reactive](#) foundation -- fully asynchronous and non-blocking, intended for use in an event-loop execution model vs traditional large thread pool with thread-per-request execution model.
- Reactive infrastructure in `spring-core` such as Encoder and Decoder for encoding and decoding streams of Objects; DataBuffer abstraction, e.g. for using Java ByteBuffer or Netty ByteBuf; ReactiveAdapterRegistry for transparent support of reactive libraries in controller method signatures.
- Reactive infrastructure in `spring-web` including `HttpMessageReader` and `HttpMessageWriter` that build on and delegate to Encoder and Decoder; server `HttpHandler` with adapters to (non-blocking) runtimes such as Servlet 3.1+ containers, Netty, and Undertow; `WebFilter`, `WebHandler` and other non-blocking contract alternatives to Servlet API equivalents.
- `@Controller` style, annotation-based, programming model, similar to Spring MVC, but supported in WebFlux, running on a reactive stack, e.g. capable of supporting reactive types as controller method arguments, never

blocking on I/O, respecting backpressure all the way to the HTTP socket, and running on extra, non-Servlet containers such as Netty and Undertow.

- New **functional programming model** ("WebFlux.fn") as an alternative to the @Controller, annotation-based, programming model -- minimal and transparent with an endpoint routing API, running on the same reactive stack and WebFlux infrastructure.
- New WebClient with a functional and reactive API for HTTP calls, comparable to the RestTemplate but through a fluent API and also excelling in non-blocking and streaming scenarios based on WebFlux infrastructure; in 5.0 the AsyncRestTemplate is deprecated in favor of the WebClient.

Kotlin support

- Null-safe API when using Kotlin 1.1.50 or higher.
- Support for Kotlin immutable classes with optional parameters and default values.
- Functional bean definition Kotlin DSL.
- Functional routing Kotlin DSL for WebFlux.
- Leveraging Kotlin reified type parameters to avoid specifying explicitly the Class to use for serialization/deserialization in various APIs like RestTemplate or WebFlux APIs.
- Kotlin null-safety support for @Autowired/@Inject and @RequestParam/@RequestHeader/etc annotations in order to determine if an injection point or handler method parameter is required or not.
- Kotlin script support in ScriptTemplateView for both Spring MVC and Spring WebFlux.
- Array-like setters added to Model, ModelMap and Environment.
- Support for Kotlin autowired constructor with optional parameters.
- Kotlin reflection is used to determine interface method parameters.

Testing Improvements

- Complete support for **JUnit 5's Jupiter** programming and extension models in the Spring TestContext Framework.
 - **SpringExtension**: an implementation of multiple extension APIs from JUnit Jupiter that provides full support for the existing feature set of the Spring TestContext Framework. This support is enabled via @ExtendWith(SpringExtension.class).
 - **@SpringJUnitConfig**: a composed annotation that combines @ExtendWith(SpringExtension.class) from JUnit Jupiter with @ContextConfiguration from the Spring TestContext Framework.
 - **@SpringJUnitWebConfig**: a composed annotation that combines @ExtendWith(SpringExtension.class) from JUnit Jupiter with @ContextConfiguration and @WebAppConfiguration from the Spring TestContext Framework.
 - **@EnabledIf**: signals that the annotated test class or test method is *enabled* if the supplied SpEL expression or property placeholder evaluates to true.
 - **@DisabledIf**: signals that the annotated test class or test method is *disabled* if the supplied SpEL expression or property placeholder evaluates to true.
- Support for **parallel test execution** in the Spring TestContext Framework.
- New **before** and **after** test execution callbacks in the Spring TestContext Framework with support for TestNG, JUnit 5, and JUnit 4 via the SpringRunner (but not via JUnit 4 rules).
 - New beforeTestExecution() and afterTestExecution() callbacks in the TestExecutionListener API and TestContextManager.
- MockHttpServletRequest now has getContentAsByteArray() and getContentAsString() methods for accessing the content (i.e., request body).
- The print() and log() methods in Spring MVC Test now print the request body if the character encoding has been set in the mock request.
- The redirectedUrl() and forwardedUrl() methods in Spring MVC Test now support URI templates with variable expansion.
- XMLUnit support upgraded to 2.3.

Spring Boot 2.0

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.0-Release-Notes>

Third-party Library Upgrades

Spring Boot 2.0 builds on and requires Spring Framework 5.

You might like to read about [the new features available in Spring Framework 5.0](#), and check out their [upgrade guide](#) before continuing.

We've upgraded to the latest stable releases of other third-party jars wherever possible. Some notable dependency upgrades in this release include:

- Tomcat 8.5
- Flyway 5
- Hibernate 5.2
- Thymeleaf 3

Reactive Spring

Many projects in the Spring portfolio are now offering first-class support for developing [reactive applications](#).

Reactive applications are fully asynchronous and non-blocking.

They're intended for use in an event-loop execution model (instead of the more traditional one thread-per-request execution model).

The ["Web on Reactive Stack"](#) section of the Spring Framework reference documentation provides an excellent primer to the subject.

Spring Boot 2.0 fully supports reactive applications via auto-configuration and starter-POMs.

The internals of Spring Boot itself have also been updated where necessary to offer reactive alternatives (the most noticeable being our embedded server support).

Spring WebFlux & WebFlux.fn

Spring WebFlux is a fully non-blocking reactive alternative to Spring MVC. Spring Boot provides auto-configuration for both annotation based Spring WebFlux applications, as well as WebFlux.fn which offers a more functional style API.

To get started, use the `spring-boot-starter-webflux` starter POM which will provide Spring WebFlux backed by an embedded Netty server. See the [Spring Boot reference documentation](#) for details.

Reactive Spring Data

Where the underlying technology enables it, Spring Data also provides support for reactive applications. Currently Cassandra, MongoDB, Couchbase and Redis all have reactive API support.

Spring Boot includes special starter-POMs for these technologies that provide everything you need to get started. For example, `spring-boot-starter-data-mongodb-reactive` includes dependencies to the reactive mongo driver and project reactor.

Reactive Spring Security

Spring Boot 2.0 can make use of Spring Security 5.0 to secure your reactive applications. Auto-configuration is provided for WebFlux applications whenever Spring Security is on the classpath.

Access rules for Spring Security with WebFlux can be configured via a `SecurityWebFilterChain`. If you've used Spring Security with Spring MVC before, this should feel quite familiar. See the [Spring Boot reference documentation](#) and [Spring Security documentation](#) for more details.

Embedded Netty Server

Since WebFlux [does not rely on Servlet APIs](#), we're now able to offer support for Netty as an embedded server for the first time.

The spring-boot-starter-webflux starter POM will pull-in Netty 4.1 and [Ractor Netty](#).

Note You can only use Netty as a reactive server. Blocking servlet API support is not provided.

HTTP/2 Support

[HTTP/2](#) support is provided for Tomcat, Undertow and Jetty.

Support depends on the chosen web server and the application environment (since the protocol is not supported out-of-the-box by JDK 8).

Configuration Property Binding

The mechanism used to bind Environment properties to [@ConfigurationProperties](#) has been completely overhauled in Spring Boot 2.0.

We've taken the opportunity to tighten the rules that govern relaxed binding and we've fixed many inconsistencies from Spring Boot 1.x.

The new Binder API can also be used outside of [@ConfigurationProperties](#) directly in your own code.

For example, the following will bind to a List of PersonName objects:

```
List<PersonName> people = Binder.get(environment)
    .bind("my.property", Bindable.listOf(PersonName.class))
    .orElseThrow(IllegalStateException::new);
```

The configuration source could be represented in YAML like this:

```
my:
  property:
    - first-name: Jane
      last-name: Doe
    - first-name: John
      last-name: Doe
```

For more information on the updated binding rules [see this wiki page](#).

Property Origins

YAML files and Properties files loaded by Spring Boot now include Origin information which can help you track where an item was loaded from.

Several Spring Boot features take advantage of this information and show it when appropriate.

For example, the BindException class thrown when binding fails is an OriginProvider.

This means origin information can be displayed nicely from a failure analyzer.

Another example is the env actuator endpoint which includes origin information when it's available.

The snippet below shows that the spring.security.user.name property came from line 1, column 27 of the application.properties file packaged in the jar:

```
{
  "name": "applicationConfig: [classpath:/application.properties]",
  "properties": {
    "spring.security.user.name": {
      "value": "user",
      "origin": "class path resource [application.properties]:1:27"
    }
  }
}
```

Converter Support

Binding makes use of a new ApplicationConversionService class which offers some additional converters which are especially useful for property binding.

Most noticeable are converters for Duration types and delimited strings.

The Duration converter allows durations to be specified in either ISO-8601 form, or as a simple string (for example 10m for 10 minutes with [support of other units](#)).

Existing properties have been changed to always use Duration. For instance, the session timeout can be configured to 180 seconds in application.properties as follows:

```
server.servlet.session.timeout=180s
```

The @DurationUnit annotation ensures back-compatibility by setting the unit that is used if not specified. For example, a property that expected seconds in Spring Boot 1.5 now has @DurationUnit(ChronoUnit.SECONDS) to ensure a simple value such as 10 actually uses 10s.

Delimited string conversion allows you to bind a simple String to a Collection or Array without necessarily splitting on commas.

For example, LDAP base-dn properties use @Delimiter(Delimiter.NONE) so that LDAP DNs (which typically include commas) are not misinterpreted.

Web Server

Apache Tomcat

- **Description:** Apache Tomcat is an open-source implementation of the Java Servlet, JavaServer Pages (JSP), and WebSocket technologies.
- **Purpose:** It serves as a servlet container, which means it can host Java-based web applications by processing servlets and JSPs.
- **Key Features:**
 - Lightweight and easy to use.
 - Supports the latest Java EE specifications.
 - Extensive configuration options.
 - Integration with various development tools.
- **Use Cases:** Ideal for small to medium-sized web applications, often used in development and testing environments.

Internal Structure of Tomcat

- 1) **Server**
 - **Description:** The top-level component that represents the entire Tomcat server. It manages and coordinates all the other components.
 - **Key Class:** org.apache.catalina.Server
 - **Configuration:** Defined in server.xml.
- 2) **Service**
 - **Description:** A logical grouping of one or more Connectors and a single Engine. Each service represents a functional unit of processing.
 - **Key Class:** org.apache.catalina.Service
 - **Configuration:** Defined within a <Service> element in server.xml.
- 3) **Connector**
 - **Description:** Manages network communications (HTTP, HTTPS, AJP, etc.) between clients and the server. It listens for incoming requests and passes them to the Engine for processing.
 - **Key Class:** org.apache.coyote.http11.Http11NioProtocol (for HTTP)
 - **Configuration:** Defined within a <Connector> element in server.xml.
- 4) **Engine**
 - **Description:** Processes requests received from the Connector and manages one or more Host components.
 - **Key Class:** org.apache.catalina.Engine
 - **Configuration:** Defined within an <Engine> element in server.xml.
- 5) **Host**
 - **Description:** Represents a virtual host (or domain) and manages multiple Context components. Each Host corresponds to a particular DNS name.
 - **Key Class:** org.apache.catalina.Host
 - **Configuration:** Defined within a <Host> element in server.xml.
- 6) **Context**
 - **Description:** Represents a single web application, corresponding to a specific Context path (URL path) within a Host. It is the main runtime environment for a web application.

If it's a static file, the Context directly serves the file using the filesystem, bypassing the servlet container.

If it's a dynamic resource, the Context creates a RequestDispatcher and forwards the request to the appropriate servlet.

- **Key Class:** org.apache.catalina.Context
- **Configuration:** Defined within a <Context> element in server.xml, context.xml, or within the web application's META-INF/context.xml.

7) **Wrapper**

- **Description:** Represents a single servlet instance within a Context. Each Wrapper manages the lifecycle of a single servlet.
- **Key Class:** org.apache.catalina.Wrapper
- **Configuration:** Typically configured in the web.xml file of the web application.

8) **Realm**

- **Description:** Manages security and authentication. A Realm provides the security service to the Engine, Host, or Context.
- **Key Class:** org.apache.catalina.Realm
- **Configuration:** Defined within a <Realm> element in server.xml.

9) **Loader**

- **Description:** Manages the Java class loader for a specific web application. Each Context has its own Loader to isolate the classes of different web applications.
- **Key Class:** org.apache.catalina.loader.WebappLoader
- **Configuration:** Defined within a <Loader> element in context.xml.

10) **Valve and Filter**

- **Valve:** Internal components that process requests and responses within a Pipeline. They are specific to Tomcat and are used for tasks like access logging and security checks.
- **Filter:** Standard Java EE components that intercept requests and responses, allowing for pre-processing or post-processing.
- **Key Classes:** org.apache.catalina.Valve, javax.servlet.Filter
- **Configuration:** Valves are configured in server.xml, while Filters are configured in the web.xml of a web application.

Workflow of a Request

- 1) **Receiving the Request:** A Connector receives an HTTP request from a client.
- 2) **Forwarding to Engine:** The Connector forwards the request to the Engine.
- 3) **Routing to Host:** The Engine routes the request to the appropriate Host based on the requested domain.
- 4) **Routing to Context:** The Host routes the request to the appropriate Context based on the URL path.
- 5) **Servlet Execution:** The Context forwards the request to the appropriate Wrapper, which invokes the target servlet.
- 6) **Response Handling:** The servlet processes the request, generates a response, which is sent back through the same path (Context → Host → Engine → Connector) to the client.

Configuration Files

- **server.xml:** The main configuration file for Tomcat server settings.
- **web.xml:** The deployment descriptor for web applications, defining servlets, filters, listeners, etc.
- **context.xml:** Configuration specific to individual web applications, defining parameters like data sources and environment entries.

Jetty

- **Description:** Jetty is an open-source, lightweight, and highly scalable Java-based web server and servlet container.
- **Purpose:** Designed for web application deployment, Jetty is known for its performance and scalability.
- **Key Features:**

- Lightweight and embeddable.
- High performance and scalability.
- Supports WebSockets, HTTP/2, and various other modern web protocols.
- Flexible and easy to integrate with other frameworks.
- **Use Cases:** Suitable for embedded applications, microservices, and highly scalable web applications.

JBoss (WildFly)

- **Description:** JBoss, now known as WildFly, is an open-source application server authored by Red Hat.
- **Purpose:** It is a Java EE (Enterprise Edition) application server, supporting a wide range of enterprise-level features.
- **Key Features:**
 - Full support for the Java EE specifications.
 - High performance and clustering capabilities.
 - Management and monitoring tools.
 - Extensive integration options with other enterprise tools and systems.
- **Use Cases:** Ideal for large-scale, enterprise-level applications requiring full Java EE support.

Undertow

- **Description:** Undertow is an open-source, lightweight, and highly flexible web server developed by Red Hat.
- **Purpose:** Designed for serving web applications, Undertow emphasizes low memory footprint and high performance.
- **Key Features:**
 - Lightweight and embeddable.
 - High performance with low memory usage.
 - Supports non-blocking (NIO) and blocking modes.
 - Provides support for HTTP/2 and WebSockets.
 - Flexible configuration options.
 - Easily embeddable in Java applications, such as those using WildFly.
- **Use Cases:** Suitable for microservices, lightweight web applications, and environments where high performance and low memory usage are critical.

Apache HTTP Server (Apache Web Server)

- **Description:** The Apache HTTP Server, commonly referred to as Apache, is a widely-used open-source web server.
- **Purpose:** It serves static and dynamic web content, supporting a variety of modules and configurations.
- **Key Features:**
 - Highly customizable with a modular architecture.
 - Supports a wide range of languages and technologies (e.g., PHP, Python, Perl).
 - Extensive security features and configurations.
 - Robust performance and scalability.
- **Use Cases:** Suitable for hosting static websites, dynamic web applications, and acting as a reverse proxy or load balancer.

Microsoft Internet Information Services (IIS)

- **Description:** IIS is a web server created by Microsoft, designed to run on Windows Server operating systems.
- **Purpose:** It is used to host websites and web applications, supporting a wide range of web technologies and protocols.
- **Key Features:**
 - Supports HTTP, HTTPS, FTP, FTPS, SMTP, and NNTP.
 - Integration with .NET framework and ASP.NET.
 - Robust security features.
 - Comprehensive management tools.

- **Use Cases:** Commonly used in enterprise environments for hosting .NET applications and services on Windows servers.

The screenshot shows the AWS Console Home page. At the top, there's a navigation bar with the AWS logo, a 'Services' dropdown, a search bar, and account information for 'Mumbai' and 'Deep Patel'. Below the navigation bar is the 'Console Home' header with a 'Info' link. To the right are buttons for 'Reset to default layout' and '+ Add widgets'. On the left, there's a sidebar titled 'Recently visited' with links to Lambda, IAM, VPC, CloudShell, AWS Resource Explorer, DynamoDB, EC2, IAM Identity Center, EFS, AWS Backup, and S3. Below this is a 'View all services' button. To the right, there's a section titled 'Applications (0)' with a 'Create application' button and a note that the current region is 'ap-south-1 (Current Region)'. A search bar for 'Find applications' is also present. The main content area shows a table with columns for Name, Description, Region, and Originating a... (partially visible). A message says 'No applications' and 'Get started by creating an application.' with a 'Create application' button. At the bottom, there are tabs for 'Welcome to AWS', 'AWS Health', and 'Cost and usage', along with links for 'CloudShell', 'Feedback', and copyright information: '© 2024, Amazon Web Services, Inc. or its affiliates.' and links for 'Privacy', 'Terms', and 'Cookie preferences'.

Services

Compute Services

Amazon ECS (Elastic Container Service)

Allows you to run containerized applications in a managed environment, providing a scalable and secure way to deploy microservices.

Amazon EKS (Elastic Kubernetes Service)

A managed Kubernetes service that provides a more customizable orchestration layer for containers. Ideal for teams already using Kubernetes.

AWS Lambda

Serverless compute service that allows you to run code without provisioning or managing servers. Suitable for smaller microservices or event-driven functions.

API Gateway

Amazon API Gateway

Enables you to create, publish, maintain, monitor, and secure APIs at any scale. It acts as a front door for your microservices, handling tasks such as traffic management, authorization, and monitoring.

Service Discovery and Configuration Management

AWS Cloud Map

Service discovery for microservices that automatically maps and registers instances of your microservices.

AWS Parameter Store or AWS Secrets Manager

Used for storing configuration parameters and secrets securely.

Storage

Amazon S3 (Simple Storage Service)

Object storage service for storing and retrieving any amount of data at any time.

Amazon RDS (Relational Database Service)

A managed relational database service that supports multiple database engines such as MySQL, PostgreSQL, and Oracle, suitable for microservices that require relational databases.

Amazon DynamoDB

A managed NoSQL database service designed for high-scale microservices that require low-latency access to data.

Messaging and Streaming:

Amazon SQS (Simple Queue Service)

A fully managed message queuing service that enables decoupling and scaling of microservices.

Amazon SNS (Simple Notification Service)

A fully managed pub/sub messaging service that makes it easy to decouple and scale microservices.

Amazon Kinesis

Used for real-time data streaming to handle data ingestion and processing for microservices that require real-time analytics.

Security

AWS IAM (Identity and Access Management)

Provides fine-grained access control to AWS resources for securing microservices.

AWS WAF (Web Application Firewall)

Protects microservices against common web exploits.

Monitoring and Logging

Amazon CloudWatch

Used to monitor the health and performance of microservices, collect logs, and set alarms.

AWS X-Ray

Helps debug and analyze microservices by tracing requests as they travel through different services in your architecture.

Networking

Amazon VPC (Virtual Private Cloud)

Allows you to launch AWS resources in a logically isolated virtual network, ensuring that your microservices are secure and can communicate privately.

AWS App Mesh

A service mesh that provides application-level networking to make it easy for your services to communicate with each other.

CI/CD and DevOps

AWS CodePipeline

A continuous integration and continuous delivery service for fast and reliable application and infrastructure updates.

AWS CodeBuild

Fully managed build service that compiles source code, runs tests, and produces software packages.

AWS CodeDeploy

Automates code deployments to any instance, including Amazon EC2 instances and Lambda functions.

SpringBoot / Usage

Prototype Bean Injection Problem

Classes

AppConfig

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
    public PrototypeBean prototypeBean() {  
        return new PrototypeBean();  
    }  
}
```

```

@Bean
public SingletonBean singletonBean() {
    return new SingletonBean();
}
}

SingletonBean
public class SingletonBean {

    // ..

    @Autowired
    private PrototypeBean prototypeBean;

    public SingletonBean() {
        logger.info("Singleton instance created");
    }

    public PrototypeBean getPrototypeBean() {
        logger.info(String.valueOf(LocalTime.now()));
        return prototypeBean;
    }
}

main
public static void main(String[] args) throws InterruptedException {
    AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);

    SingletonBean firstSingleton = context.getBean(SingletonBean.class);
    PrototypeBean firstPrototype = firstSingleton.getPrototypeBean();

    // get singleton bean instance one more time
    SingletonBean secondSingleton = context.getBean(SingletonBean.class);
    PrototypeBean secondPrototype = secondSingleton.getPrototypeBean();

    assertTrue(firstPrototype.equals(secondPrototype), "The same instance should be returned");
}

```

Both beans were initialized **only once**, at the startup of the application context.

Solution1

```

public class SingletonApplicationContextBean implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public PrototypeBean getPrototypeBean() {
        return applicationContext.getBean(PrototypeBean.class);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
        throws BeansException {
        this.applicationContext = applicationContext;
    }
}

```

Every time the `getPrototypeBean()` method is called, a new instance of `PrototypeBean` will be returned from the `ApplicationContext`.

However, this approach has serious disadvantages. It **contradicts the principle of inversion of control**, as we request the dependencies from the container directly.

Also, we fetch the prototype bean from the `ApplicationContext` within the `SingletonApplicationContextBean` class. This means coupling the code to the Spring Framework.

Solution2

```
@Component
public class SingletonLookupBean {

    @Lookup
    public PrototypeBean getPrototypeBean() {
        return null;
    }
}
```

Spring will override the getPrototypeBean() method annotated with @Lookup.

It then registers the bean into the application context. Whenever we request the getPrototypeBean() method, it returns a new PrototypeBean instance.

It will use CGLIB to generate the bytecode responsible for fetching the PrototypeBean from the application context.

Solution3

```
public class SingletonProviderBean {

    @Autowired
    private Provider<PrototypeBean> myPrototypeBeanProvider;

    public PrototypeBean getPrototypeInstance() {
        return myPrototypeBeanProvider.get();
    }
}
```

Custimize HttpServletRequestWrapper

Referenced classes

jakarta.servlet.http.HttpServletRequestWrapper

TestServletRequestWrapper

```
package com.simil.AAAconfig;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.commons.io.IOUtils;

import jakarta.servlet.ReadListener;
import jakarta.servlet.ServletInputStream;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletRequestWrapper;
import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Map;

public class TestServletRequestWrapper extends HttpServletRequestWrapper {

    private final ServletInputStream inputStream;

    private BufferedReader reader;

    private String requestBodyLine;

    /**
     * Constructs a request object wrapping the given request.
     *
     * @param request The request to wrap
     * @throws IllegalArgumentException if the request is null
     */
    public TestServletRequestWrapper(HttpServletRequest request) throws IOException {

```

```

super(request);
request.getParameterNames();
byte[] body = IOUtils.toByteArray(request.getInputStream());
this.inputStream = new RequestCachingInputStream(body);
}

@Override
public ServletInputStream getInputStream() throws IOException {
    if (this.inputStream != null) {
        return this.inputStream;
    }
    return super.getInputStream();
}

@Override
public BufferedReader getReader() throws IOException {
    if (this.reader == null) {
        this.reader = new BufferedReader(new InputStreamReader(getInputStream(),
getCharacterEncoding()));
    }
    return this.reader;
}

public Map<String, Object> getRequestBodyToMap() throws IOException {
    String jsonInfo = this.getRequestBodyToStr();
    ObjectMapper objectMapper = new ObjectMapper();
    return objectMapper.readValue(jsonInfo, Map.class);
}

public String getRequestBodyToStr() throws IOException {
    if (this.requestBodyLine == null) {
        BufferedReader reader = this.getReader();
        StringBuilder builder = new StringBuilder();
        String line = reader.readLine();
        while (line != null) {
            builder.append(line);
            line = reader.readLine();
        }
        requestBodyLine = builder.toString();
        reader.close();
    }
    return requestBodyLine;
}

private static class RequestCachingInputStream extends ServletInputStream {

    private final ByteArrayInputStream is;

    public RequestCachingInputStream(byte[] bytes) {
        this.is = new ByteArrayInputStream(bytes);
    }

    @Override
    public int read() throws IOException {
        return this.is.read();
    }

    @Override
    public boolean isFinished() {
        return this.is.available() == 0;
    }

    @Override
    public boolean isReady() {
        return true;
    }
}

```

```
    @Override
    public void setReadListener(ReadListener readlistener) {
    }
}
```

TestFilter

```
package com.simi.AAAconfig.filters;
import com.simi.AAAconfig.TestServletRequestWrapper;
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import lombok.extern.slf4j.Slf4j;

import java.io.IOException;

@WebFilter(filterName = "testFilter", urlPatterns = "/*")
@Slf4j
public class TestFilter implements Filter {
    @Override
    public void doFilter(HttpServletRequest servletRequest, HttpServletResponse servletResponse, FilterChain filterChain)
            throws IOException, ServletException {
        TestServletRequestWrapper testServletRequestWrapper=new
        TestServletRequestWrapper((HttpServletRequest)servletRequest);
        log.info("servlet request wrapped");
        filterChain.doFilter(testServletRequestWrapper, servletResponse);
    }
}
```

SpringBoot / Configuration

application.yml

application.yml 或 application.properties 或 application.yaml 项目默认配置文件 (在同一级目录下优先级为: properties > yml > yaml)

加载优先级: application-dev.yml > application.yml

YAML 语法: 大小写敏感,

数值前必须有空格

缩进不能用 Tab, 只能用空格

单引号字符串忽略转义'hello \n word'

logging:

 level:

 web: DEBUG SpringBoot 项目, 如果使用单独的 LogBack 配置文件, SpringBoot 的配置文件 application.properties 配置的 logging.level.root 将会覆盖的 Logback 配置文件中的 root 的配置:

```
server:
  connection-timeout: 5s # 5 seconds timeout
tomcat:
  keep-alive-timeout: 20s # 20 seconds keep-alive timeout
  max-keep-alive-requests: 100
```

server:

 port: 8099 启动端口

```
port: ${PORT}      从 vmoptions 中获取: -DPORT=80
maxHttpHeaderSize=102400000  header 参数大小限制
servlet:
  context-path: /springboot-demo
    Define the base URL path for the web application.
    It sets the root context for all the application's endpoints, effectively prefixing all the URLs.

  multipart:
    enabled: true
    max-file-size: 10MB
    max-request-size: 15MB

tomcat:
  uri-encoding: utf-8
  basedir: /usr/tmp/tomcat
  accesslog:
    enabled: false
    directory: logs    日志文件所在目录, 没有就创建, 可以是绝对路径或相对路径
    buffered: true     缓存输出 (周期性 flush)
    pattern: common   日志输出格式
    suffix: .log       日志后缀
    prefix: access_log 日志前缀
    rotate: true      日志旋转
```

```
spring:
config:
  activate:
    on-profile: local,dev
      结合三个短横线一起使用, 标识这一段配置仅在 local 或 dev 环境时才加载
application:
  name: sdk-trade
  程序名称
  main:
    web-application-type: none
    应用类型【REACTIVE, SERVLET, NONE】
    allow-circular-references: true
    允许循环依赖, 默认 false
  profiles:
    active: dev
    激活的其他的配置文件 application-dev.yml (application.yml 或 application.properties 一直生效)
mvc:
  static-path-pattern: /res/**
  定义根目录访问静态资源的 url 路径, 在 resource 目录下 (会导致 welcom 页失效, index 默认页面失效)
    hiddenmethod:
      filter:
        enable: true
        可以通过表单 psot 方式提交的值决定请求方式<input name="_method" value="put"/>
    log-request-details: true
    显示日志详细信息
resources:
  static-location: [classpath:/stac] 静态资源存放目录 (favicon.ico 项目图标)
  add-mappings: true            启用静态资源访问
  cache:
    period: 11000          缓存时间
```

```
servlet:
  multipart:
    max-file-size: 10MB          multipart/form-dta 请求 单个文件最大值
    max-request-size: 100MB       multipart/form-dta 请求 单次请求文件总最大值
datasource:
  username: root
  password: root
  driver-class-name: net.sf.log4jdbc.DriverSpy
  url: jdbc:log4jdbc:mysql://localhost:3306/springboot
  schema:
    - classpath:sql/function.sql      # 第一种初始化数据库方式，新版本已移除
    - classpath:sql/procedure.sql
  initialization-mode: ALWAYS
  separator:\\
sql:
  init:
    encoding: utf-8
    mode: always          # 设置模式，不要 never，不然不起作用
    platform: mysql
    username: root
    password: '你的密码'
    data-locations: classpath:data-all.sql      #这个数据的 sql 脚本的位置
    schema-locations: classpath:schema-all.sql   # 这个是创建表的 sql 脚本的位置
my-car:           自定义类配置（不用加引号，‘-’ 分割命名）
  userName: zhangsan     字符串
  boss: true            布尔值
  a: [xxx, sss]         数组
  arr:                 数组
    - XXX
    - SSS
strList: aaa, bbb, ccc
obj: { a:80, b:90}      对象
obj:
  a: 80
  b: 90
routes:          对象数组
  - id: template
  uri: http://127.0.0.1:8099
  predicates:
    - Path=/api-template/**
filters:
  - StripPrefix=1
  - id: generator
  uri: lb://boss-generator
  predicates:
    - Path=/api-generator/**
filters:
```

- StripPrefix=1

person:

name: \${my-car.obj.a} 引用上面定义的 my-car

--- 单个文件内多文件开始语法

server:

port: \${PORT} 从 vmoptions 中获取: -DPORT=80

... 单个文件内多文件结尾语法

spring.factories

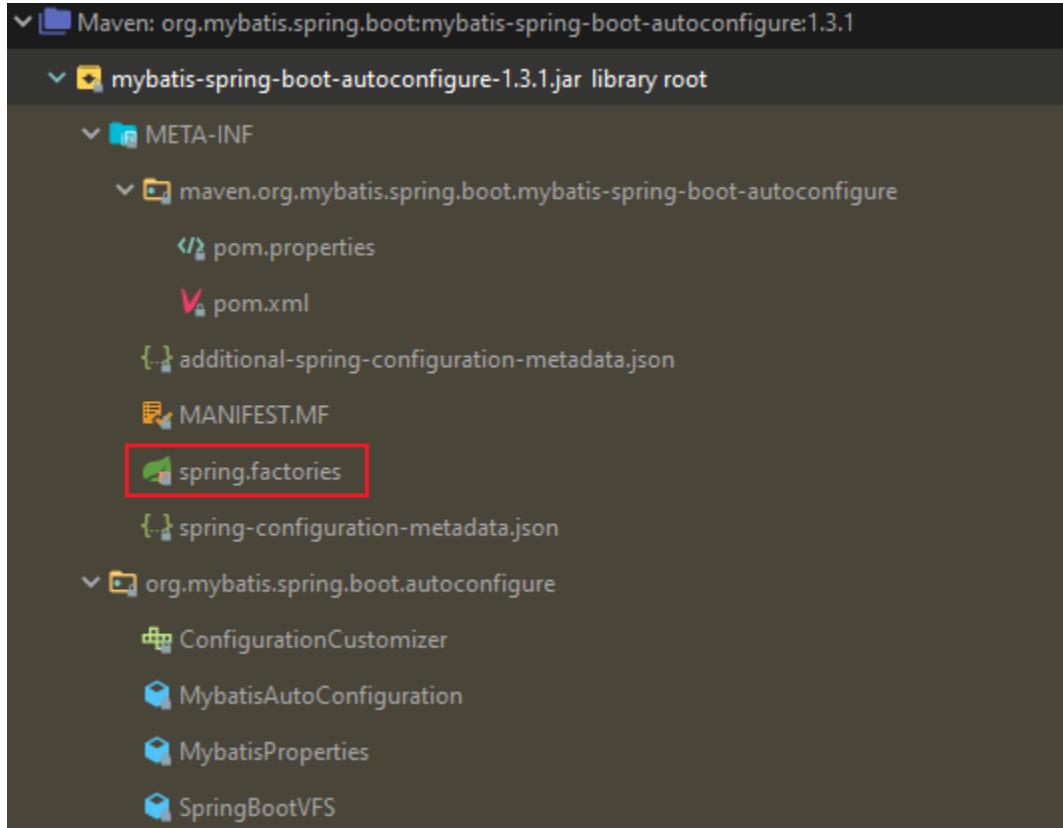
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.saidake.redis.config.MyAutoConfiguration

装载

自定义自动配置类的 bean

org.springframework.context.ApplicationContextInitializer=com.saidake.redis.config.MyApplicationContextInitializer

org.springframework.boot.SpringApplicationRunListener=com.saidake.redis.config.MySpringApplicationRunListener



pom.properties

META-INF/maven/\${groupId}/\${artifactId}/pom.xml

META-INF/maven/\${groupId}/\${artifactId}/pom.properties

The pom.xml and pom.properties files are packaged up in the JAR so that each artifact produced by Maven is self-describing and also allows you to utilize the metadata in your own application, should the need arise. One simple use might be to retrieve the version of your application.

Define Bean

Using Java Configuration

Person

```
public class Person {
```

```

private String firstName;
private String lastName;

public Person(String firstName, String secondName) {
    super();
    this.firstName = firstName;
    this.lastName = secondName;
}

@Override
public String toString() {
    return "Person [firstName=" + firstName + ", secondName=" + lastName + "]";
}
}

PersonConfig
@Configuration
public class PersonConfig {
    @Bean
    public Person personOne() {
        return new Person("Harold", "Finch");
    }

    @Bean
    public Person personTwo() {
        return new Person("John", "Reese");
    }
}

```

Using @Component Annotation

```

PersonOne
@Component
public class PersonOne extends Person {

    public PersonOne() {
        super("Harold", "Finch");
    }
}

PersonConfig
@Configuration
@ComponentScan("com.baeldung.multibeaninstantiation.solution2")
public class PersonConfig {
}

```

Using BeanFactoryPostProcessor

```

Human
public class Human implements InitializingBean {

    private Person personOne;

    private Person personTwo;

    @Override
    public void afterPropertiesSet() throws Exception {
        Assert.notNull(personOne, "Harold is alive!");
        Assert.notNull(personTwo, "John is alive!");
    }

    /* Setter injection */
    @Autowired
    public void setPersonOne(Person personOne) {
        this.personOne = personOne;
        this.personOne.setFirstName("Harold");
        this.personOne.setSecondName("Finch");
    }
}
```

```

    @Autowired
    public void setPersonTwo(Person personTwo) {
        this.personTwo = personTwo;
        this.personTwo.setFirstName("John");
        this.personTwo.setSecondName("Reese");
    }
}

Person
@Qualifier(value = "personOne, personTwo")
public class Person implements FactoryBean<Object> {
    private String firstName;
    private String secondName;

    public Person() {
        // initialization code (optional)
    }

    @Override
    public Class<Person> getObjectType() {
        return Person.class;
    }

    @Override
    public Object getObject() throws Exception {
        return new Person();
    }

    public boolean isSingleton() {
        return true;
    }

    // code for getters & setters
}

PersonFactoryPostProcessor
public class PersonFactoryPostProcessor implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
        Map<String, Object> map = beanFactory.getBeansWithAnnotation(Qualifier.class);
        for (Map.Entry<String, Object> entry : map.entrySet()) {
            createInstances(beanFactory, entry.getKey(), entry.getValue());
        }
    }

    private void createInstances(ConfigurableListableBeanFactory beanFactory, String beanName, Object bean) {
        Qualifier qualifier = bean.getClass().getAnnotation(Qualifier.class);
        for (String name : extractNames(qualifier)) {
            Object newBean = beanFactory.getBean(beanName);
            beanFactory.registerSingleton(name.trim(), newBean);
        }
    }

    private String[] extractNames(Qualifier qualifier) {
        return qualifier.value().split(",");
    }
}

PersonConfig
@Configuration
public class PersonConfig {
    @Bean
    public PersonFactoryPostProcessor PersonFactoryPostProcessor() {
        return new PersonFactoryPostProcessor();
    }

    @Bean
    public Person person() {
        return new Person();
    }
}

```

```
}

@Bean
public Human human() {
    return new Human();
}
}
```

Add Filter

@WebFilter

This registers the filter directly with the **Servlet container** (like Tomcat or Jetty) **via the Servlet 3.0 annotations**, making it a more container-centric way of adding filters.

TestFilter

```
package com.simi.AAAconfig.filters;

import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;
import jakarta.servlet.http.HttpServletRequest;
import org.apache.commons.lang3.StringUtils;
import org.springframework.util.CollectionUtils;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

@WebFilter(filterName = "testFilter", urlPatterns = "/*", initParams = @WebInitParam(name = "noFilterUrl",
value = "/test"))
public class TestFilter implements Filter {
    private List<String> noFilterUrls;

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // 从过滤器配置中获取 initParams 参数
        String noFilterUrl = filterConfig.getInitParameter("noFilterUrl");
        // 将排除的 URL 放入成员变量 noFilterUrls 中
        if (StringUtils.isNotBlank(noFilterUrl)) {
            noFilterUrls = new ArrayList<>(Arrays.asList(noFilterUrl.split(",")));
        }
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain)
        throws IOException, ServletException {
        // 若请求中包含 noFilterUrls 中的片段则直接跳过过滤器进入下一步请求中
        String url = ((HttpServletRequest)servletRequest).getRequestURI();
        Boolean flag = false;
        if (!CollectionUtils.isEmpty(noFilterUrls)) {
            for (String noFilterUrl : noFilterUrls) {
                if (url.contains(noFilterUrl)) {
                    flag = true;
                    break;
                }
            }
        }
        if (!flag) {
            // 过滤请求响应逻辑
        }
        filterChain.doFilter(servletRequest, servletResponse);
    }
}
```

```

@Override
public void destroy() {
}

}



## MainApplication


package com.sim;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletComponentScan;

@SpringBootApplication
@ServletComponentScan
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }
}

```

@Component

Using the @Component annotation to automatically register it.

MyFilter

```

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import java.io.IOException;

@Component
public class MyFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // Initialization code if necessary
    }

    @Override
    public void doFilter(javax.servlet.ServletRequest request,
                        javax.servlet.ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        // Log request details or manipulate request/response
        System.out.println("Request URL: " + httpRequest.getRequestURL());

        // Pass the request along the filter chain
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
        // Cleanup code if necessary
    }
}

```

FilterRegistrationBean

For more control over the filter's order or which URLs it applies to, use FilterRegistrationBean.

TestFilter

```
package com.simi.AAAconfig.filters;

import jakarta.servlet.*;
import lombok.extern.slf4j.Slf4j;

import java.io.IOException;

@Slf4j
public class TestFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {
        }
}
```

FilterRegistrationBean

```
package com.simi.AAAconfig;

import com.simi.AAAconfig.filters.TestFilter;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Configuration
public class FilterConfig {

    @Bean
    public FilterRegistrationBean<TestFilter> requestFilterRegistration() {
        FilterRegistrationBean<TestFilter> registration = new FilterRegistrationBean<>();
        registration.setFilter(new TestFilter());
        registration.setName("requestFilter");
        registration.setOrder(1);
        Map<String, String> paramMap = new HashMap<>();
        paramMap.put("noFilterUrl", "/test");
        registration.setInitParameters(paramMap);
        List<String> urlPatterns = new ArrayList<>();
        urlPatterns.add("/*");
        registration.setUrlPatterns(urlPatterns);
        return registration;
    }
}
```

Add Interceptor

WebMvcConfigurer

RequestInterceptor

```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component
public class RequestInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        // Logic to be executed before the request reaches the controller
        System.out.println("Pre-handle logic executed: " + request.getRequestURI());
    }
}
```

```

    // Returning true allows the request to proceed to the controller
    return true;
}

@Override
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
org.springframework.web.servlet.ModelAndView modelAndView) throws Exception {
    // Logic to be executed after the controller processes the request but before the view is rendered
    System.out.println("Post-handle logic executed");
}

@Override
public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
Exception ex) throws Exception {
    // Logic to be executed after the view has been rendered
    System.out.println("After-completion logic executed");
}
}

```

WebConfig

Now, you need to register this interceptor. You do this by creating a class that implements WebMvcConfigurer and overriding the addInterceptors method.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private RequestInterceptor requestInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // Register the interceptor and apply it to all paths
        registry.addInterceptor(requestInterceptor).addPathPatterns("/**");
    }
}

```

SpringBoot / org.springframework (spring)

aop

AnnotationAwareAspectJAutoProxyCreator

```

package org.springframework.aop.aspectj.annotation;
public class AnnotationAwareAspectJAutoProxyCreator extends AspectJAwareAdvisorAutoProxyCreator      实现 AOP 动态
代理

```

AOP 和@Async 注解虽然底层都是动态代理，但是具体实现的类是不一样的。

一般的 AOP 或者事务的动态代理是依靠 AnnotationAwareAspectJAutoProxyCreator 实现的，

@Async 是依靠 AsyncAnnotationBeanPostProcessor 实现的，并且都是在初始化完成之后起作用，

这也就是@Async 注解和 AOP 之间的主要区别，也就是处理的类不一样。

```

package org.springframework.aop.aspectj.autoproxy;
public class AspectJAwareAdvisorAutoProxyCreator extends AbstractAdvisorAutoProxyCreator

```

interceptor

ExposeInvocationInterceptor

```

package org.springframework.aop.interceptor;
public final class ExposeInvocationInterceptor implements MethodInterceptor, PriorityOrdered, Serializable

```

用于暴露拦截器链到 ThreadLocal 中，这样同一个线程下就可以来共享拦截器链了

```

public static final Advisor ADVISOR;
private static final ThreadLocal<MethodInvocation> invocation;
invoke
@Nullable
public Object invoke(MethodInvocation mi) throws Throwable {           // ReflectiveMethodInvocation 实例
    MethodInvocation oldInvocation = (MethodInvocation)invocation.get();
    invocation.set(mi);
}

Object var3;
try {
    var3 = mi.proceed();      // 现在需要调用 ReflectiveMethodInvocation 的 proceed()方法了
} finally {
    invocation.set(oldInvocation);
}

return var3;
}

```

framework

AbstractAutoProxyCreator

```

package org.springframework.aop.framework.autoproxy;
public abstract class AbstractAutoProxyCreator extends ProxyProcessorSupport
    implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware
Key Responsibilities


- Automatic Proxy Creation
            Automatically creates proxies for beans based on certain criteria, such as bean name patterns, annotations, or other custom logic.
- Advisor Application
            Applies advisors (which include pointcuts and advice) to the beans being proxied.
- Integration with Spring Container
            Integrates seamlessly with the Spring container, ensuring proxies are created as part of the bean lifecycle.


public Object postProcessAfterInitialization(@Nullable Object bean, String beanName)
protected abstract Object[] getAdvisesAndAdvisorsForBean(Class<?> beanClass, String beanName, @Nullable TargetSource
customTargetSource) throws BeansException;
Abstract method that must be implemented by subclasses to specify which advisors should be applied to a particular bean.
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey)
Internal method that determines whether a bean needs to be proxied and, if so, creates the proxy.

```

AbstractAdvisingBeanPostProcessor

```

package org.springframework.aop.framework;
public abstract class AbstractAdvisingBeanPostProcessor extends ProxyProcessorSupport implements BeanPostProcessor
public Object postProcessAfterInitialization(Object bean, String beanName)    bean 加载后执行，对方法入参的对象进行动态代理的。

```

当入参的对象的类加了@Async 注解，那么这个方法就会对这个对象进行动态代理，最后会返回入参对象的代理对象出去。

至于如何判断方法有没有加@Async 注解，是靠 isEligible(bean, beanName) 来判断的。

AdvisedSupport

```
public class ADVISEDSupport extends ProxyConfig implements ADVISED
```

Key Responsibilities

- Proxy Configuration
Maintains configuration details for AOP proxies, such as the target object, interfaces, and advice.
- Advice Management
Manages the list of interceptors or advisors to be applied to the target object.
- Proxy Creation
Provides methods for creating proxy instances with the configured advice and target object.

```
public void setTarget(Object target) Sets the target object to be proxied.
```

```
public void setTargetSource(@Nullable TargetSource targetSource) Sets the TargetSource which contains the target object.
```

```
public void addInterface(Class<?> intf) Adds an interface to be proxied.
```

```
public void setInterfaces(Class<?>... interfaces) Sets the interfaces to be proxied.
```

```
public void addAdvisor(Advisor advisor) Adds an Advisor to the list of advisors.
```

```
public void addAdvice(Advice advice) throws AopConfigException Adds an Advice instance directly.
```

```
public void removeAdvisor(int index) throws AopConfigException Removes an Advisor from the list.
```

```
public boolean removeAdvice(Advice advice) throws AopConfigException Removes an Advice from the list.
```

AopContext

```
package org.springframework.aop.framework;
```

```
public final class AopContext
```

```
static Object setCurrentProxy(@Nullable Object proxy)
```

```
public static Object currentProxy() throws IllegalStateException 获取当前代理对象
```

JdkDynamicAopProxy

```
package org.springframework.aop.framework;
```

```
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable
```

Key Responsibilities

- Proxy Creation
Creates proxy instances for target objects based on the interfaces they implement.
- Method Interception
Intercepts method calls on the proxy and applies the configured advice (interceptors or advisors).
- Advice Execution
Manages the execution of advice in the order specified.

```
private final ADVISEDSupport advised;
```

```
public JdkDynamicAopProxy(ADVISEDSupport config) throws AopConfigException  
ADVISEDSupport configuration.
```

Initializes the proxy with the given

invoke

```
@Nullable
```

```

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object oldProxy = null;
    boolean setProxyContext = false;
    TargetSource targetSource = this.advised.targetSource;
    Object target = null;

    Class var8;
    try {
        if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
            Boolean var18 = this.equals(args[0]);
            return var18;
        }

        if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
            Integer var17 = this.hashCode();
            return var17;
        }

        if (method.getDeclaringClass() != DecoratingProxy.class) {
            Object retVal;
            if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&
method.getDeclaringClass().isAssignableFrom(Advised.class)) {
                retVal = AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
                return retVal;
            }

            if (this.advised.exposeProxy) {
                oldProxy = AopContext.setCurrentProxy(proxy);
                setProxyContext = true;
            }

            target = targetSource.getTarget();
            Class<?> targetClass = target != null ? target.getClass() : null;
            //A. Obtain interceptor chain
            List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,
targetClass);
            if (chain.isEmpty()) {
                Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                //B. Invoke method
                retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
            } else {
                MethodInvocation invocation = new ReflectiveMethodInvocation(proxy, target, method, args,
targetClass, chain);
                retVal = invocation.proceed();
            }

            Class<?> returnType = method.getReturnType();
            if (retVal != null && retVal == target && returnType != Object.class &&
returnType.isInstance(proxy) && !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
                retVal = proxy;
            } else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
                throw new AopInvocationException("Null return value from advice does not match primitive
return type for: " + method);
            }
        }

        Object var12 = retVal;
        return var12;
    }

    var8 = AopProxyUtils.ultimateTargetClass(this.advised);
} finally {
    if (target != null && !targetSource.isStatic()) {
        targetSource.releaseTarget(target);
    }

    if (setProxyContext) {
        AopContext.setCurrentProxy(oldProxy);
    }
}

```

```

    }

    return var8;
}

```

ProxyFactoryBean

```

package org.springframework.aop.framework;
public class ProxyFactoryBean extends ProxyCreatorSupport
    implements FactoryBean<Object>, BeanClassLoaderAware, BeanFactoryAware

```

Key Responsibilities

- Creating Proxies
ProxyFactoryBean creates a proxy for a target bean, applying the specified advice (interceptors, advisors).
- Configuring AOP Proxies
It allows configuring aspects, target beans, and proxy interfaces.
- Managing Advice
It can manage a list of interceptors and advisors to be applied to the target bean.

```
private boolean singleton = true;           Determines if the proxy should be a singleton.
```

public Object getObject() throws BeansException	Returns the proxy object.
public Class<?> getObjectType()	Returns the type of the proxy.
public boolean isSingleton()	Indicates whether the proxy is a singleton.

Usage

```

@Configuration
public class AppConfig {

    @Bean
    public ProxyFactoryBean myProxy() {
        ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.setTarget(myTargetBean());
        proxyFactoryBean.setInterceptorNames("myInterceptor", "myAdvisor");
        proxyFactoryBean.setProxyInterfaces(new Class<?>[] { MyInterface.class });
        return proxyFactoryBean;
    }

    @Bean
    public MyTargetBean myTargetBean() {
        return new MyTargetBean();
    }

    @Bean
    public MyInterceptor myInterceptor() {
        return new MyInterceptor();
    }

    @Bean
    public MyAdvisor myAdvisor() {
        return new MyAdvisor();
    }
}

```

ReflectiveMethodInvocation

```

package org.springframework.aop.framework;

```

```

public class ReflectiveMethodInvocation implements ProxyMethodInvocation, Cloneable
proceed
@Nullable
public Object proceed() throws Throwable {
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
        //A. 当 currentInterceptorIndex 的值为拦截器链最后一个拦截器的下标时，才会满足，此时就会执行
        invokeJoinpoint()这行代码。
        使用反射执行目标对象的目标方法，此时拦截器链都执行完了，但是还没有执行完毕，而此时拦截器链中的 before 增强已
        经执行完毕了
        return this.invokeJoinpoint();
    } else {
        //A. 直接将 currentInterceptorIndex 作为拦截器链的下标，来从拦截器链
        interceptorsAndDynamicMethodMatchers 中获取拦截器
        Object interceptorOrInterceptionAdvice =
        this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
        if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
            InterceptorAndDynamicMethodMatcher dm =
            (InterceptorAndDynamicMethodMatcher)interceptorOrInterceptionAdvice;
            Class<?> targetClass = this.targetClass != null ? this.targetClass :
            this.method.getDeclaringClass();
            //A. 如果匹配失败，那么就跳过这个拦截器并调用下一个拦截器链中的下一个
            return dm.matcher().matches(this.method, targetClass, this.arguments) ?
            dm.interceptor().invoke(this) : this.proceed();
        } else {
            //A. 这里是核心方法，如果是普通拦截器，那么就直接执行
            aspectJ 的五种增强都是 MethodInterceptor 接口的实现类
            return ((MethodInterceptor)interceptorOrInterceptionAdvice).invoke(this);
        }
    }
}

invokeJoinpoint
@Nullable
protected Object invokeJoinpoint() throws Throwable {
    return AopUtils.invokeJoinpointUsingReflection(this.target, this.method, this.arguments);
}

invokeJoinpointUsingReflection
@Nullable
public static Object invokeJoinpointUsingReflection(@Nullable Object target, Method method, Object[] args)
throws Throwable {
    try {
        ReflectionUtils.makeAccessible(method);
        return method.invoke(target, args);
    } catch (InvocationTargetException var4) {
        throw var4.getTargetException();
    } catch (IllegalArgumentException var5) {
        throw new AopInvocationException("AOP configuration seems to be invalid: tried calling method [" +
method + "] on target [" + target + "]", var5);
    } catch (IllegalAccessException var6) {
        throw new AopInvocationException("Could not access method [" + method + "]", var6);
    }
}

```

beans

PropertyAccessor

```

package org.springframework.beans;
public interface PropertyAccessor 属性访问器
void setPropertyValue(java.lang.String s, @org.springframework.lang.Nullable java.lang.Object o) 设置属性

```

```
java.lang.Object getPropertyValue(java.lang.String s)    获取属性
```

BeanWrapper

```
package org.springframework.beans;  
public interface BeanWrapper      bean 封装  
    java.beans.PropertyDescriptor[] getPropertyDescriptors()  获取属性修饰器  
    java.lang.Object getWrappedInstance()  获取包裹的实例
```

BeanWrapperImpl

```
package org.springframework.beans;  
public class BeanWrapperImpl      bean 封装实现  
BeanWrapperImpl(java.lang.Object object)  直接传入对象
```

BeanUtils

```
package org.springframework.beans;  
public abstract class BeanUtils  
public static void copyProperties(Object source, Object target) throws BeansException 赋值包含 null  
    字段名不一致，属性无法拷贝  
    类型不一致，属性无法拷贝（基本类型 转 对应的包装类，这种可以转化，但是反过来如上个例子所说，如果包装类未赋值，  
    转换的时候会抛异常）  
    嵌套对象字段，将会与源对象使用同一对象，即使用浅拷贝  
public static void copyProperties(Object source, Object target, String... ignoreProperties) throws BeansException
```

factory

BeanFactory

```
package org.springframework.beans.factory;  
public interface BeanFactory
```

The BeanFactory interface in Spring is the root interface for accessing the Spring IoC container.

It provides basic methods for managing and accessing beans, making it a core component of the Spring framework's dependency injection mechanism.

1. Key Characteristics of BeanFactory

Core IoC Container

BeanFactory is the foundational interface for Spring's IoC container. It provides methods to manage and access beans.

Lazy Initialization

Beans are instantiated lazily in BeanFactory by default. This means a bean is created only when it is requested.

Lightweight

BeanFactory is lightweight and does not provide advanced features like event propagation, declarative mechanisms to create a bean, or the ability to resolve messages.

2. Key Components of BeanFactory

Bean Definition

A BeanDefinition is a data structure that contains information about a bean, including its class, scope, dependencies, and property values.

Bean Definition Registry

A BeanDefinitionRegistry is responsible for holding and managing bean definitions.

Bean Creation

The process of creating and configuring bean instances based on their definitions.

Related Classes:

org.springframework.beans.factory.BeanFactoryAware

org.springframework.beans.factory.BeanNameAware

org.springframework.beans.factory.DisposableBean

org.springframework.beans.factory.InitializingBean

org.springframework.core.Ordered

org.springframework.beans.factory.config.BeanPostProcessor

org.springframework.beans.factory.config.BeanFactoryPostProcessor

org.springframework.beans.factory.config.ConfigurableListableBeanFactory

org.springframework.integration.context.IntegrationObjectSupport

DefaultListableBeanFactory

XmlBeanFactory

AnnotationConfigApplicationContext

GenericApplicationContext

ClassPathXmlApplicationContext

FileSystemXmlApplicationContext

WebApplicationContext

<T> T getBean(String var1, Class<T> var2) 指定 bean 名称和类型获取 bean

<T> T getBean(Class<T> var1) 指定 bean 类型获取 bean

boolean containsBean(String var1) 是否包含 bean

boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

boolean isPrototype(String name) throws NoSuchBeanDefinitionException;

BeanFactoryAware

package org.springframework.beans.factory;

public interface BeanFactoryAware extends Aware 模板接口，用于获取 BeanFactory，但不知道具体的工厂实现

void setBeanFactory(BeanFactory var1) throws BeansException; Gets the current BeanFactory

BeanNameAware

package org.springframework.beans.factory;

public interface BeanNameAware extends Aware Gets the current bean name.

org.springframework.integration.context.IntegrationObjectSupport

void setBeanName(String name);

BeanClassLoaderAware

package org.springframework.beans.factory;

public interface BeanClassLoaderAware extends Aware 和 BeanFactoryAware 接口同理，获取 Bean 的类装载器

void setBeanClassLoader(ClassLoader classLoader);

BeanCurrentlyInCreationException

```
package org.springframework.beans.factory;
public class BeanCurrentlyInCreationException extends BeanCreationException @Async 注解遇上循环依赖的时候，Spring
    的确无法解决。
T getObject() throws Exception;
Class<?> getObjectType();
default boolean isSingleton() { return true; }
```

DisposableBean

```
package org.springframework.beans.factory;
public interface DisposableBean 允许在容器销毁该 bean 的时候获得一次回调
void destroy() throws Exception;
```

FactoryBean

```
package org.springframework.beans.factory;
public interface FactoryBean<T>
```

Key Responsibilities

- Customized Bean Creation
Defines a factory method for creating bean instances with potentially complex instantiation logic.
- Configuration
Allows configuration of factory behavior and initialization parameters.
- Lifecycle Management
Supports lifecycle methods for initialization and destruction of created objects.
- Type Safety
Ensures type-safe retrieval of created objects.

```
Class<?> getObjectType();
default boolean isSingleton()
```

HierarchicalBeanFactory

```
package org.springframework.beans.factory;
public interface HierarchicalBeanFactory 提供父容器的访问功能
```

```
BeanFactory getParentBeanFactory(); 返回本 Bean 工厂的父工厂
boolean containsLocalBean(String name); 本地工厂(容器)是否包含这个 Bean
```

InitializingBean

```
package org.springframework.beans.factory;
public interface InitializingBean 定义了每个 bean 的初始化前进行的动作
```

```
void afterPropertiesSet() throws Exception; Check whether BeanFactory has set all the bean properties, and satisfied
other dependencies.
@Override
public void afterPropertiesSet() throws Exception {
    Assert.notNull(personOne, "Harold is alive!");
    Assert.notNull(personTwo, "John is alive!");
}
```

ListableBeanFactory

```
package org.springframework.beans.factory;
public interface ListableBeanFactory 提供遍历搜索 bean 的功能，在容器内部
```

```
String[] getBeanDefinitionNames()          返回此 BeanFactory 中所包含的所有 Bean 定义的名称  
String[] getBeanNamesForType(@Nullable Class<?> var1)    返回此 BeanFactory 中所有指定类型的 Bean 的名字  
int getBeanDefinitionCount()           查看此 BeanFactory 中包含的 Bean 数量
```

ObjectProvider

```
package org.springframework.beans.factory;  
public interface ObjectProvider<T> extends ObjectFactory<T>, Iterable<T>
```

ObjectFactory

```
package org.springframework.beans.factory;  
public interface ObjectFactory<T>    普通的对象工厂接口
```

T getObject() throws BeansException

annotation

Autowired

```
package org.springframework.beans.factory.annotation;  
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD,  
ElementType.ANNOTATION_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Autowired
```

The @Autowired annotation in Spring is used to automatically inject dependencies into a Spring-managed bean. It is a core feature of Spring's Dependency Injection (DI) framework and helps in reducing boilerplate code by automatically wiring the dependencies based on type.

Key Features

- Type-Based Injection
By default, @Autowired wires dependencies by type. Spring looks for a bean of the matching type and injects it.
- Required Attribute
The required attribute (which is true by default) specifies whether the dependency is mandatory. If no matching bean is found, an exception is thrown if required is true.
- Field, Setter, and Constructor Injection
@Autowired can be used on fields, setter methods, and constructors.

Handling Multiple Beans

If there are multiple beans of the same type, you can use @Qualifier to specify which bean to inject.

@Autowired and @Resource

Injection Mechanism:

- @Autowired injects dependencies by type and allows for flexibility in injection (constructor, setter, or field).
- @Resource injects dependencies primarily by name, and optionally by type if no name is provided.

Configuration and Customization:

- @Autowired allows for the use of @Qualifier to specify which bean to inject when multiple candidates are available.
- @Resource uses the name of the bean for injection and requires explicit naming if needed.

Default Behavior:

- @Autowired defaults to injecting by type and may not work correctly if multiple beans of the same type exist

without additional configuration.

@Resource defaults to injecting by name, which can be more straightforward when dealing with specific bean names.

Optional Dependencies:

@Autowired allows **for optional dependencies** by setting required = false.

@Resource does not directly support optional dependencies; it expects the specified name or type to be present.

```
boolean required() default true;  
@Qualifier  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MyService {  
  
    @Autowired  
    @Qualifier("specificBean")  
    private MyRepository myRepository;  
  
    public void performService() {  
        myRepository.doSomething();  
    }  
}  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
required  
@Component  
public class MyService {  
  
    @Autowired(required = false)  
    private MyOptionalRepository myOptionalRepository;  
  
    public void performService() {  
        if (myOptionalRepository != null) {  
            myOptionalRepository.doSomething();  
        } else {  
            System.out.println("Optional dependency is not available");  
        }  
    }  
}
```

Lookup

```
package org.springframework.beans.factory.annotation;  
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Lookup
```

The @Lookup annotation is typically used on methods of a class that is managed by the Spring container.

When the annotated method is called, Spring dynamically overrides it to return a bean instance. This can be useful in scenarios where a singleton-scoped bean needs to depend on a prototype-scoped bean.

Key Responsibilities

- **Dependency Injection**

Facilitates injecting beans at runtime, ensuring **that the correct scope is respected**.

- **Method Overriding**

The container overrides the annotated method to provide the appropriate bean instance.

```
String value() default "";
Usage
import org.springframework.beans.factory.annotation.Lookup;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean {

    public void showPrototypeBean() {
        PrototypeBean prototypeBean = getPrototypeBean();
        prototypeBean.printMessage();
    }

    @Lookup
    protected PrototypeBean getPrototypeBean() {
        // Spring will override this method to return a new PrototypeBean instance
        return null;
    }
}
```

Qualifier

```
package org.springframework.beans.factory.annotation;
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE,
ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Qualifier
```

The `@Qualifier` annotation in Spring is used to resolve the ambiguity when multiple beans of the same type are present.

It works alongside `@Autowired` to specify which bean should be injected when there are multiple candidates.

Key Features

- Disambiguation
 - Specifies the exact bean to inject when multiple beans of the same type exist.
- Custom Naming
 - Works with custom bean names defined in the configuration.
- Flexible
 - Can be used on fields, setter methods, and constructor arguments.

```
String value() default "";
Custom Qualifier
@Qualifier("customQualifier")
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE})
public @interface CustomQualifier {
}

@Component
@CustomQualifier
public class MyServiceImpl1 implements MyService {
    // Implementation details
}

@Component
public class MyComponent {
```

```

private final MyService myService;

@Autowired
public MyComponent(@CustomQualifier MyService myService) {
    this.myService = myService;
}

public void performService() {
    myService.doSomething();
}
}

```

Value

```

package org.springframework.beans.factory.annotation;
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Value

```

The `@Value` annotation in Spring is used to inject values into fields, method parameters, and constructor arguments from property files, system properties, environment variables, and other sources.

It is a convenient way to externalize configuration and manage constants in a Spring application.

Key Features

- Property Injection
Injects values from property files, such as `application.properties` or `application.yml`.
- Environment Variables
Allows injection of environment variables.
- Default Values
Supports specifying default values in case the property is not found.

Read Configuration in Spring Boot

- `@Value`
- `@ConfigurationProperties`
- Environment interface
- `@PropertySource`
- Command Line
`java -jar myapp.jar --person.name=John --person.age=30`
- `@ConditionalOnClass`

`String value();`

Injecting a Property Value

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

```

```

@Component
public class MyComponent {

    @Value("${my.property}")
    private String myProperty;

    public void printProperty() {
        System.out.println("Property value: " + myProperty);
    }
}

```

Assume the following in `application.properties`:

`my.property=Hello, World!`

Injecting with a Default Value

```
@Value("${non.existent.property:Default Value}")
private String nonExistentProperty;
```

If non.existent.property is not found, Default Value will be used.

Injecting Environment Variables

```
@Value("${HOME}")
private String homeDirectory;
```

This injects the value of the HOME environment variable.

Using SpEL (Spring Expression Language)

You can use SpEL to perform more complex operations, such as concatenation or method calls.

```
@Value("#{systemProperties['user.name']}")
private String userName;
```

```
@Value("#{T(java.lang.Math).random() * 100}")
private double randomValue;
```

config

AutowireCapableBeanFactory

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

public interface **AutowireCapableBeanFactory** 让容器外的 Bean 被 spring 管理

```
<T> T createBean(Class<T> beanClass) throws BeansException; 创建一个给定 Class 的 bean 实例 (会执行此 Bean 所有的关于 Bean 生命周期的接口方法)
```

```
void autowireBean(Object existingBean) throws BeansException; 用于装配 Bean。
```

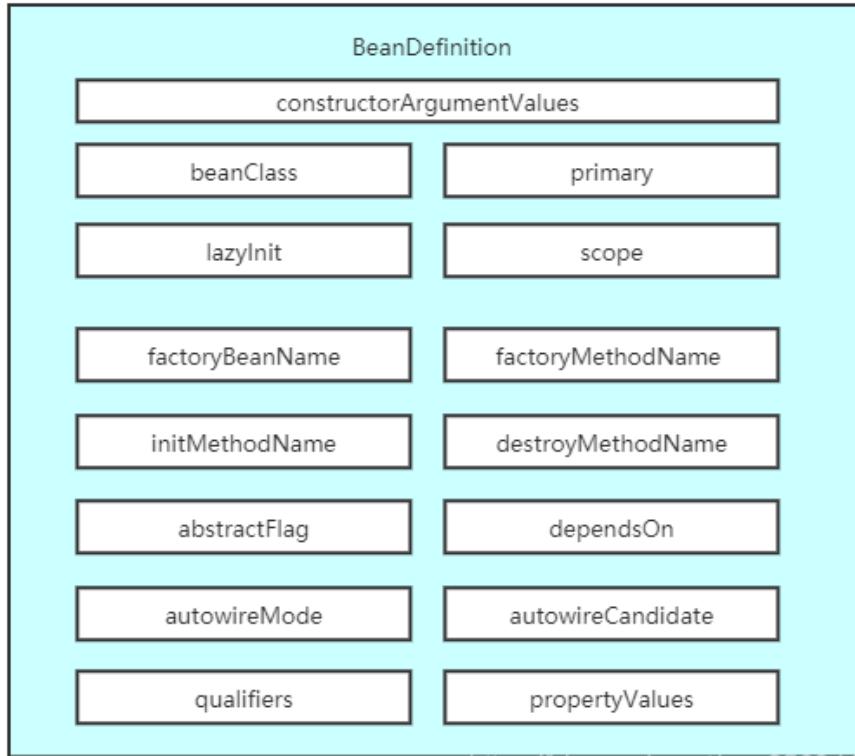
主要是通过反射获取到我们 new 出来的对象的属性及注解，若是注解时 Autowired、Value、Inject 时，进行 Bean 组装。

此方法执行完毕，我们 new 出来的方法就可以通过注解注入的 bean 进行操作了。

BeanDefinition

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

public interface **BeanDefinition** extends AttributeAccessor, BeanMetadataElement Spring 会扫描指定包下面的 Java 类，然后将其变成 beanDefinition 对象，然后 Spring 会根据 beanDefinition 来创建 bean



BeanPostProcessor

package org.springframework.beans.factory.config;
 public interface **BeanPostProcessor** 工厂钩子允许修改新的 bean 实例，实现了这个接口标记的将会用代理在 bean 实例化后做一些操作。

Object **postProcessBeforeInitialization**(Object bean, String beanName) throws BeansException;
 Object **postProcessAfterInitialization**(Object bean, String beanName) throws BeansException;
 作用

bean 实例化前做一些操作
 bean 实例化后做一些操作

BeanFactoryPostProcessor

package org.springframework.beans.factory.config;
 public interface **BeanFactoryPostProcessor**

Here, the custom BeanFactoryPostProcessor implementation gets invoked after the Spring container is initialized and before any Spring bean gets created.

This allows us to configure and manipulate the bean lifecycle.

The BeanFactoryPostProcessor scans all the classes annotated with @Qualifier.

Furthermore, it extracts names (bean ids) from that annotation, and manually creates instances of that class type with the specified names:

仅运行一次，定义了在 bean 工厂对象 创建后，bean 对象 创建前 执行的动作，操作创建后的工厂

void **postProcessBeanFactory**(ConfigurableListableBeanFactory var1) throws BeansException;

public void **postConstruct**();

ConfigurableBeanFactory

package org.springframework.beans.factory.config;
 public interface **ConfigurableBeanFactory** 配置 bean 工厂

```
void registerAlias(String var1, String var2);
void registerScope(String var1, Scope var2);
void addBeanPostProcessor(BeanPostProcessor beanPostProcessor);
```

ConfigurableListableBeanFactory

```
package org.springframework.beans.factory.config;
public interface ConfigurableListableBeanFactory extends ListableBeanFactory, AutowireCapableBeanFactory,
ConfigurableBeanFactory 提供 bean definition 的解析, 注册功能, 再对单例来个预加载 (解决循环依赖问题)
```

SingletonBeanRegistry

```
package org.springframework.beans.factory.config;
public interface SingletonBeanRegistry 运行期间注册单例 Bean
```

对于单实例 (singleton) 的 Bean 来说, BeanFactory 会缓存 Bean 实例, 所以第二次使用 getBean() 获取 Bean 时将直接从 IoC 容器的缓存中获取 Bean 实例。

Spring 在 DefaultSingletonBeanRegistry 类中提供了一个用于缓存单实例 Bean 的缓存器, 它是一个用 HashMap 实现的缓存器,

单实例的 Bean 以 beanName 为键保存在这个 HashMap 中。

boolean containsSingleton(String beanName)	检查此注册表是否包含具有给定名称的单例实例。
Object getSingleton(String beanName)	返回在给定名称下注册的 (原始) 单例对象。
int getSingletonCount()	返回在此注册表中注册的单例 bean 的数量。
Object getSingletonMutex()	返回此注册表使用的单例互斥锁 (对于外部协作者)。
String[] getSingletonNames()	返回在此注册表中注册的单例 bean 的名称。
void registerSingleton(String beanName, Object singletonObject)	在给定的 bean 名称下, 在 bean 注册表中将给定的现有对象注册为 singleton。

support

AbstractAutowireCapableBeanFactory

```
package org.springframework.beans.factory.support;
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactory
    implements AutowireCapableBeanFactory
```

createBeanInstance

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
    // Make sure bean class is actually resolved at this point.
    Class<?> beanClass = resolveBeanClass(mbd, beanName);

    if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed())
    {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());
    }

    Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
    if (instanceSupplier != null) {
        return obtainFromSupplier(instanceSupplier, beanName, mbd);
    }

    if (mbd.getFactoryMethodName() != null) {
```

```

        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }

    // Shortcut when re-creating the same bean...
    boolean resolved = false;
    boolean autowireNecessary = false;
    if (args == null) {
        synchronized (mbd.constructorArgumentLock) {
            if (mbd.resolvedConstructorOrFactoryMethod != null) {
                resolved = true;
                autowireNecessary = mbd.constructorArgumentsResolved;
            }
        }
    }
    if (resolved) {
        if (autowireNecessary) {
            return autowireConstructor(beanName, mbd, null, null);
        }
        else {
            return instantiateBean(beanName, mbd);
        }
    }

    // Candidate constructors for autowiring?
    Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
    if (ctors != null || mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR ||
        mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
        return autowireConstructor(beanName, mbd, ctors, args);
    }

    // Preferred constructors for default construction?
    ctors = mbd.getPreferredConstructors();
    if (ctors != null) {
        return autowireConstructor(beanName, mbd, ctors, null);
    }

    // No special handling: simply use no-arg constructor.
    return instantiateBean(beanName, mbd);
}

populateBean
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
    if (bw == null) {
        if (mbd.hasPropertyValues()) {
            throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
        }
        else {
            // Skip property population phase for null instance.
            return;
        }
    }

    if (bw.getWrappedClass().isRecord()) {
        if (mbd.hasPropertyValues()) {
            throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Cannot apply property values to a record");
        }
        else {
            // Skip property population phase for records since they are immutable.
            return;
        }
    }

    // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
    // state of the bean before properties are set. This can be used, for example,
    // to support styles of field injection.
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

```

```

        for (InstantiationAwareBeanPostProcessor bp : getBeanPostProcessorCache().instantiationAware) {
            if (!bp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                return;
            }
        }
    }

    PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

    int resolvedAutowireMode = mbd.getResolvedAutowireMode();
    if (resolvedAutowireMode == AUTOWIRE_BY_NAME || resolvedAutowireMode == AUTOWIRE_BY_TYPE) {
        MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
        // Add property values based on autowire by name if applicable.
        if (resolvedAutowireMode == AUTOWIRE_BY_NAME) {
            autowireByName(beanName, mbd, bw, newPvs);
        }
        // Add property values based on autowire by type if applicable.
        if (resolvedAutowireMode == AUTOWIRE_BY_TYPE) {
            autowireByType(beanName, mbd, bw, newPvs);
        }
        pvs = newPvs;
    }
    if (hasInstantiationAwareBeanPostProcessors()) {
        if (pvs == null) {
            pvs = mbd.getPropertyValues();
        }
        for (InstantiationAwareBeanPostProcessor bp : getBeanPostProcessorCache().instantiationAware) {
            PropertyValues pvsToUse = bp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);
            if (pvsToUse == null) {
                return;
            }
            pvs = pvsToUse;
        }
    }
}

boolean needsDepCheck = (mbd.getDependencyCheck() != AbstractBeanDefinition.DEPENDENCY_CHECK_NONE);
if (needsDepCheck) {
   PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
    //A dependency check involves verifying if all required dependencies are available before proceeding
    //with property population. This can indirectly help identify circular dependencies.
    checkDependencies(beanName, mbd, filteredPds, pvs);
}

if (pvs != null) {
    applyPropertyValues(beanName, mbd, bw, pvs);
}
}

initializeBean
protected Object initializeBean(String beanName, Object bean, @Nullable RootBeanDefinition mbd) {
    invokeAwareMethods(beanName, bean);

    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
            (mbd != null ? mbd.getResourceDescription() : null), beanName, ex.getMessage(), ex);
    }
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }
}

```

```
    return wrappedBean;
}
```

DefaultListableBeanFactory

```
package org.springframework.beans.factory.support;
public class DefaultListableBeanFactory      默认的 bean 工厂
    extends AbstractAutowireCapableBeanFactory
    implements ConfigurableListableBeanFactory, BeanDefinitionRegistry, Serializable

private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>(256); 真正的容器

public void preInstantiateSingletons() throws BeansException {
    if (logger.isTraceEnabled()) {
        logger.trace("Pre-instantiating singletons in " + this);
    }
    List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);      迭代副本以允许 init 方法注册新的 bean 定义
    for (String beanName : beanNames) {                                         触发所有非懒加载的单例 bean 的初始化
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            if (isFactoryBean(beanName)) {
                Object bean = getBean(FACTORY_BEAN_PREFIX + beanName);
                if (bean instanceof FactoryBean) {
                    final FactoryBean<?> factory = (FactoryBean<?>) bean;
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                        isEagerInit = AccessController.doPrivileged( (PrivilegedAction<Boolean>)
                            ((SmartFactoryBean<?>) factory)::isEagerInit,
                            getAccessControlContext());
                    }
                    else {
                        isEagerInit = (factory instanceof SmartFactoryBean &&
                            ((SmartFactoryBean<?>) factory).isEagerInit());
                    }
                    if (isEagerInit) {
                        getBean(beanName);
                    }
                }
            }
        }
    }
    else {
        getBean(beanName);
    }
}
}

for (String beanName : beanNames) {      触发所有适用 bean 的初始化后回调...
    Object singletonInstance = getSingleton(beanName);
    if (singletonInstance instanceof SmartInitializingSingleton) {
        final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;
```

```

        if (System.getSecurityManager() != null) {
            AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                smartSingleton.afterSingletonsInstantiated();
                return null;
            }, getAccessControlContext());
        }
        else {
            smartSingleton.afterSingletonsInstantiated();
        }
    }
}

```

DefaultSingletonBeanRegistry

```

package org.springframework.beans.factory.support;
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry

private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
private final Map<String, Object> earlySingletonObjects = new ConcurrentHashMap<>(16);

The earlySingletonObjects in Spring Framework is used to temporarily store partially initialized singleton bean instances
during their creation.

private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

private final Set<String> registeredSingletons = new LinkedHashSet<>(256);
private final Set<String> singletonsCurrentlyInCreation =
    Collections.newSetFromMap(new ConcurrentHashMap<>(16));

addSingletonFactory
protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}

```

BeanDefinitionRegistry

```

package org.springframework.beans.factory.support;
public interface BeanDefinitionRegistry          可以将 BeanDefinition 注册到 BeanFactory，提供了向容器中手动注册
BeanDefinition 对象的功能
void registerBeanDefinition(String var1, BeanDefinition var2) throws BeanDefinitionStoreException;    往 BeanFactory 容器中
添加 BeanDefinition，后续获取 Bean 也是从这个容器中获取

```

<p>public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor 的扩展，在注册 BeanDefinition 之前，操作 BenDefiniton 列表</p> <p>void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry var1) throws BeansException; 调用此方法，对注册器中的 BenDefiniton 对象进行增删改查的操作</p>	<p>对 BeanFactoryPostProcessor 在注册 BeanDefinition 之前，</p>
---	--

cache

Cache

```
package org.springframework.cache;  
public interface Cache    自 Spring 3.1 起, 提供了基于注解的 Cache 支持 (之前是用 aop 实现的), 且提供了 Cache 抽象。  
    提供基本的 cache 抽象, 方便切换各种底层 cache。  
    通过注解 Cache 可以实现逻辑代码透明缓存。  
    支持事故回滚时也自动回滚缓存。  
    支持复杂的缓存逻辑。
```

```
String getName();          获取缓存的名字  
Object getNativeCache();  获取底层使用的缓存, 如 Ehcache  
ValueWrapper get(Object key);      根据 key 得到一个 ValueWrapper, 然后调用其 get 方法获取值  
<T> T get(Object key, Class<T> type);  根据 key, 和 value 的类型直接获取 value  
void put(Object key, Object value);    往缓存放数据  
void evict(Object key);        从缓存中移除 key 对应的缓存  
void clear();                清空缓存  
interface ValueWrapper {  
    Object get();            得到真实的 value  
}  
}
```

CacheManager

```
package org.springframework.cache;  
public interface CacheManager  
    org.springframework.cache.support.SimpleCacheManager  
    org.springframework.cache.annotation.Cacheable
```

How CacheManager Works

- Cache Retrieval
 - When a cache operation (e.g., @Cacheable) is invoked, Spring will use the CacheManager **to retrieve or create the specified cache**.
- Cache Operations
 - The CacheManager provides methods to perform operations like **retrieving cache instances, checking cache existence, and managing cache lifecycle**.
- Custom CacheManager
 - You can define custom CacheManager beans **to customize the caching behavior** according to your application's needs.

Commonly Used CacheManager Implementations

Spring Boot supports several cache manager implementations out of the box, including:

- SimpleCacheManager
 - A simple implementation that manages a collection of caches in memory.
- ConcurrentMapCacheManager
 - Manages caches backed by ConcurrentHashMap instances.
- EhCacheCacheManager
 - Manages caches using EhCache.
- GuavaCacheManager
 - Manages caches using Google Guava.
- JcacheCacheManager
 - Manages caches using the JCache (JSR-107) specification.
- RedisCacheManager

- Manages caches using Redis.
- CaffeineCacheManager
 - Manages caches using Caffeine.

```
Cache getCache(String var1);
Collection<String> getCacheNames();
```

Enable Caching

Enable caching in your Spring Boot application by adding the @EnableCaching annotation to one of your configuration classes.

```
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableCaching
public class CacheConfig { }
```

Add a Cache Manager Bean

```
import org.springframework.cache.CacheManager;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager("items", "users"); // Cache names
    }
}
```

Use Caching Annotations

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.stereotype.Service;

@Service
public class ItemService {

    @Cacheable(value = "items", key = "#id")
    public String getItemById(Long id) {
        // Simulate a time-consuming operation
        return "Item" + id;
    }

    @CachePut(value = "items", key = "#id")
    public String updateItem(Long id, String newItem) {
        return newItem; // Updates the cache and returns the new value
    }

    @CacheEvict(value = "items", key = "#id")
    public void deleteItem(Long id) {
        // Removes the item from the cache
    }
}
```

concurrent

ConcurrentMapCache

package org.springframework.cache.concurrent;

```
public class ConcurrentMapCache extends AbstractValueAdaptingCache  
public ConcurrentMapCache(String name)
```

interceptor

KeyGenerator

```
package org.springframework.cache.interceptor;  
@FunctionalInterface  
public interface KeyGenerator
```

The KeyGenerator is an interface used in the caching abstraction to generate cache keys for the cache operations.

When using caching annotations such as @Cacheable, @CachePut, and @CacheEvict,

the KeyGenerator helps in creating unique keys based on the method parameters, method name, or any custom logic you define.

Default Behavior

By default, Spring uses a simple key generation strategy:

- If no parameters are present, a simple key consisting of SimpleKey.EMPTY is used.
- If only one parameter is present, that parameter itself is used as the key.
- If multiple parameters are present, an instance of SimpleKey containing all parameters is used.

```
Object generate(Object target, Method method, Object... params);
```

Generate a key for the given method and its parameters.

support

SimpleCacheManager

```
package org.springframework.cache.support;  
public class SimpleCacheManager extends AbstractCacheManager
```

```
public void setCache(Collection<? extends Cache> cache)
```

Usage

```
@Bean  
Public CacheManager cacheManager(){  
    SimpleCacheManager cacheManager= new SimpleCacheManager();  
    cacheManager.setCache(Arrays.asList(new ConcurrentMapCache("<cach-name>"));  
    return cacheManager;  
}
```

annotation

Cacheable

```
package org.springframework.cache.annotation;  
public @interface Cacheable Triggers cache population.
```

Spring Cache supports a wide variety of caching options, including Redis, EhCache, and Caffeine. They can be used independently or in combination.

```
@AliasFor("cacheNames")
```

```
String[] value() default {};
```

Specifies the name of the cache(s) where the results should be stored and retrieved.

```
@AliasFor("value")
```

```
String[] cacheNames() default {};
```

```
String key() default "";
```

Defines the key under which the cached result should be stored and retrieved.

Uses Spring's SpEL (Spring Expression Language) to dynamically determine the key based on method parameters or

other variables.

Example using a method parameter: @Cacheable(value = "booksCache", key = "#isbn")

Multiple keys can be specified using a comma-separated list or an array.

```
String keyGenerator() default "";
String cacheManager() default "";
String cacheResolver() default "";
String condition() default "";
String unless() default "";
boolean sync() default false;
```

CacheEvict

```
package org.springframework.cache.annotation;
public @interface CacheEvict Triggers cache eviction.
@AliasFor("cacheNames")
String[] value() default {};
@AliasFor("value")
String[] cacheNames() default {};
String key() default "";
String keyGenerator() default "";
String cacheManager() default "";
String cacheResolver() default "";
String condition() default "";
boolean allEntries() default false;
boolean beforeInvocation() default false;
```

CachePut

```
package org.springframework.cache.annotation;
public @interface CachePut Updates the cache without interfering with the method execution.
@AliasFor("cacheNames")
String[] value() default {};
@AliasFor("value")
String[] cacheNames() default {};
String key() default "";
String keyGenerator() default "";
String cacheManager() default "";
String cacheResolver() default "";
String condition() default "";
String unless() default "";
```

Caching

```
package org.springframework.cache.annotation;
public @interface Caching
Cacheable[] cacheable() default {};
CachePut[] put() default {};
CacheEvict[] evict() default {};
```

CacheConfig

```
package org.springframework.cache.annotation;  
public @interface CacheConfig  
String[] cacheNames() default {};  
String keyGenerator() default "";  
String cacheManager() default "";  
String cacheResolver() default "";
```

EnableCaching

```
package org.springframework.cache.annotation;  
public @interface EnableCaching      Enable Spring Cache  
boolean proxyTargetClass() default false;  
AdviceMode mode() default AdviceMode.PROXY;  
int order() default 2147483647;
```

cglib

Enhancer

```
package org.springframework.cglib.proxy;  
public class Enhancer extends AbstractClassGenerator  
public void setSuperclass(Class superclass)  
public void setCallback(final Callback callback)
```

MethodInterceptor

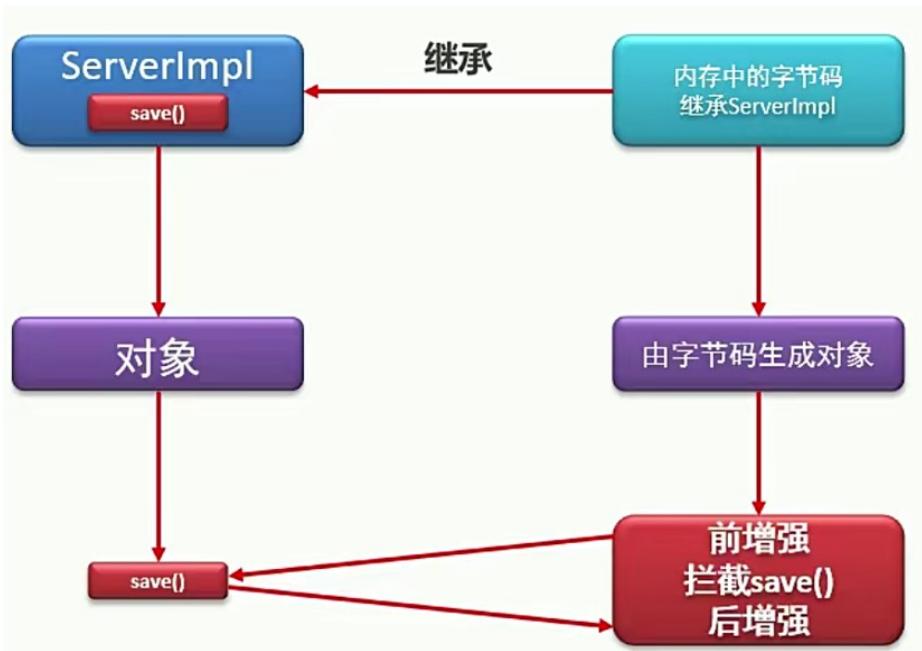
```
package org.springframework.cglib.proxy;  
public interface MethodInterceptor      CGLIB 动态代理，不需要实现接口，利用 asm 开源包，加载代理对象类的 class 文件，  
通过修改其字节码生成子类，覆盖其中的方法（针对类实现代理）
```

```
Object intercept(Object var1, Method var2, Object[] var3, MethodProxy var4)
```

原理

```
JdkDynamicAopProxy getProxy  
ProxyCreatorSupport createAopProxy  
ProxyFactory getProxy  
AbstractAutoProxyCreator createProxy  
AopUtils isAopProxy
```

文章：spring cglib 代理无法取到被代理类属性



CGLIB 的动态代理，是基于**现有类创建一个子类，并实例化子类对象**。

在调用动态代理对象方法时，都是先调用子类方法，子类方法中使用方法**增强 Advice** 或者**拦截器 MethodInterceptor** 处理子类方法调用后，选择性的决定是否执行父类方法。

那么假设在调用 `async1` 方法时，使用的是动态生成的子类的实例，那么 `this` 其实是**基于动态代理的子类实例对象**，`this` 调用是可以被 `Advice` 或者 `MethodInterceptor` 等处理逻辑拦截的，那么为何理论和实际不同呢？

这里大胆推测一下，其实 `async1` 方法中的 `this` 不是动态代理的子类对象，而是**原始的对象**，故 `this` 调用**无法通过动态代理**来增强。

使用

context

ApplicationListener

```

package org.springframework.context;
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener 程序监听器
    void onApplicationEvent(E event);

    static <T> ApplicationListener<PayloadApplicationEvent<T>> forPayload(Consumer<T> consumer) {
        return (event) -> {
            consumer.accept(event.getPayload());
        };
    }
}
  
```

ApplicationEvent

```

package org.springframework.context;
public abstract class ApplicationEvent extends EventObject 自定义应用事件
    org.springframework.context.ApplicationListener
    org.springframework.context.event.ApplicationEventMulticaster

    org.springframework.context.support.AbstractApplicationContext#publishEvent
  
```

```
org.springframework.context.event.AbstractApplicationEventMulticaster
```

ApplicationContext

```
package org.springframework.context;  
public interface ApplicationContext IOC 容器,  
    extends EnvironmentCapable, ListableBeanFactory, HierarchicalBeanFactory, MessageSource,  
    ApplicationEventPublisher, ResourcePatternResolver
```

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。

ApplicationContext 继承了 HierarchicalBeanFactory 和 ListableBeanFactory 接口，在此基础上，还通过多个其他的接口扩展了 BeanFactory 的功能：

ClassPathXmlApplicationContext 实现了 ApplicationContext

FileSystemXmlApplicationContext 可以加载文件系统中任意位置的配置文件，而 ClassPathXmlApplicationContext 只能加载类路径下的配置文件

```
org.springframework.context.ApplicationContextAware
```

```
org.springframework.web.context.ServletContextAware
```

```
org.springframework.context.support.AbstractApplicationContext
```

```
default void publishEvent(ApplicationEvent event) Publish an application event.
```

ApplicationContextInitializer

```
package org.springframework.context;  
public interface ApplicationContextInitializer<C extends ConfigurableApplicationContext>
```

```
void initialize(C applicationContext) Operations before initializing ApplicationContext
```

ApplicationContextAware

```
package org.springframework.context;  
public interface ApplicationContextAware extends Aware
```

Retrieve the instantiated bean from the existing spring context, After implementing the interface, this class can easily obtain all beans in the ApplicationContext

The situation of obtaining the ApplicationContext from the ApplicationContextAware only applies when the currently running code and the started Spring code are in the same spring context, otherwise the obtained ApplicationContext is empty.

It's recommended to use ApplicationContext instead of BeanFactory.

ApplicationContext supports almost all types of bean scopes, while BeanFactory only supports two scopes.

```
void setApplicationContext(ApplicationContext applicationContext)
```

ServletContextAware

```
package org.springframework.context;  
public interface ServletContextAware extends Aware In Spring, any class that implements the ServletContextAware interface can obtain ServletContext object.  
void setServletContext(ServletContext servletContext);
```

ApplicationEventPublisher

```
package org.springframework.context;  
public interface ApplicationEventPublisher
```

让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。

EmbeddedValueResolverAware

```
package org.springframework.context;  
public interface EmbeddedValueResolverAware extends Aware Obtains a single field in the properties file.  
void setEmbeddedValueResolver(StringValueResolver resolver);
```

ConfigurableApplicationContext

```
package org.springframework.context;  
public interface ConfigurableApplicationContext 扩展于 ApplicationContext，它新增加了两个主要的方法：refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。  
extends ApplicationContext, Lifecycle, Closeable
```

void refresh() 加载或刷新配置的持久化表示，可能是 XML 文件、属性文件或关系数据库模式。
由于这是一种启动方法，如果失败，它应该销毁已经创建的单例，以避免悬空资源。
换句话说，在调用该方法之后，应该实例化所有单例或根本不实例化单例。

void close(); 关闭应用上下文。
boolean isActive(); 确定此应用程序上下文是否处于活动状态，即它是否已至少刷新一次且尚未关闭。
void setParent(@Nullable ApplicationContext parent); 设置此应用程序上下文的父级（只有在创建此类的对象时，父级不可用时，才能在构造函数外部设置父级，例如在 WebApplicationContext 设置的情况下）

Lifecycle

```
package org.springframework.context;  
public interface Lifecycle 该接口是 Spring 2.0 加入的，该接口提供了 start() 和 stop() 两个方法，主要用于控制异步处理过程。  
在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现，  
ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。
```

MessageSource

```
package org.springframework.context;  
public interface MessageSource 为应用提供 i18n 国际化消息访问的功能；
```

SmartLifecycle

```
package org.springframework.context;  
public interface SmartLifecycle 所有 bean 都交给 Spring 容器来统一管理，其中包括每一个 bean 的加载和初始化。有时候我们
```

需要在 Spring 加载和初始化所有 bean 后，接着执行一些任务或者业务逻辑。

extends Lifecycle, Phased SmartLifecycle 是一个接口。当 Spring 容器加载所有 bean 并完成初始化之后，会接着回调实现该接口的类中对应的方法（start()方法）。

annotation

ConditionContext

```
package org.springframework.context.annotation;
public interface ConditionContext      上下文对象，用于获取环境，IOC 容器，ClassLoader 对象
```

Condition

```
package org.springframework.context.annotation;
public interface Condition    自定义 Conditional 条件，配合@Conditional 注解使用
```

```
boolean matches(ConditionContext var1, AnnotatedTypeMetadata var2)    true 则满足条件
```

ImportSelector

```
package org.springframework.context.annotation;
public interface ImportSelector    自定义 bean 导入器
```

```
public class MyImportSelector implements ImportSelector {
    public String[] selectImports(AnnotationMetadata icm) {
        return new String[]{"com.itheima.dao.impl.AccountDaoImpl"};
    }
}
```

```
@Configuration
@ComponentScan("com.itheima")
@Import(MyImportSelector.class)
public class SpringConfig {
}
```

ImportBeanDefinitionRegistrar

```
package org.springframework.context.annotation;
public interface ImportBeanDefinitionRegistrar    自定义 bean 定义注册器
```

```
void registerBeanDefinitions(AnnotationMetadata var1, BeanDefinitionRegistry var2)
```

```
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    public void registerBeanDefinitions(AnnotationMetadata icm, BeanDefinitionRegistry r) {
        ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(r, false);
        TypeFilter tf = new TypeFilter() {
            public boolean match(MetadataReader mr, MetadataReaderFactory mrf) throws
IOException {
                return true;
            }
        };
        scanner.addIncludeFilter(tf);
        //scanner.addExcludeFilter(tf);
        scanner.scan("com.itheima");
    }
}
```

```

public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.rootBeanDefinition(User.class).getBeanDefinition();
        registry.registerBeanDefinition("user", beanDefinition);
    }
}

```

ImportBeanDefinitionRegistrar

package org.springframework.context.annotation;
 public interface **ImportBeanDefinitionRegistrar** 支持我们自己写的代码封装成 BeanDefinition 对象; 实现此接口的类会回调 postProcessBeanDefinitionRegistry 方法, 注册到 spring 容器中。

把 bean 注入到 spring 容器不止有 @Service @Component 等注解方式; 还可以实现此接口。

AnnotationConfigApplicationContext

package org.springframework.context.annotation;
 public class **AnnotationConfigApplicationContext** 加载纯注解格式上下文 // new
 AnnotationConfigApplicationContext(SpringConfig.class);
 extends GenericApplicationContext implements AnnotationConfigRegistry

AspectJAutoProxyRegistrar

package org.springframework.context.annotation;
 class **AspectJAutoProxyRegistrar** implements ImportBeanDefinitionRegistrar

(annotation)

Bean

package org.springframework.context.annotation;
 @Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
 @Retention(RetentionPolicy.RUNTIME)
 @Documented
 public @interface **Bean**

The @Bean annotation in Spring is used to indicate that a method produces a bean to be managed by the Spring container.

It is typically used within `@Configuration` classes to define beans explicitly rather than using component scanning to discover and configure beans automatically.

Single DataSource bean

`SpringJdbcPersistenceServiceConfiguration` expects a single DataSource bean.

If one of the DataSource beans should be the default, mark it with `@Primary`, the other DataSource will not be used automatically unless explicitly referenced via `@Qualifier`.

Bean Scope

By default, beans created using @Bean are singleton scoped, meaning Spring manages only one instance of the bean per container.

You can specify other scopes such as prototype, request, session, etc., using the `@Scope` annotation in conjunction with `@Bean`.

the @Bean annotation uses the method name (`dataServiceTest`) as the bean name by default.

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;

```

`@Configuration`

```

public class AppConfig {
    @Bean
    @Scope("prototype")
    public DataService dataServiceTest() {
        return new DataServiceImpl();
    }
}

```

Dependency Injection

@Bean methods can accept parameters, allowing for dependency injection of other beans or properties.

```

@Bean
public DataSource dataSource(Environment env) {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setUrl(env.getProperty("jdbc.url"));
    dataSource.setUsername(env.getProperty("jdbc.username"));
    dataSource.setPassword(env.getProperty("jdbc.password"));
    return dataSource;
}

```

Conditional Bean Registration

You can conditionally register beans using @Bean methods along with @Conditional annotations.

```

@Bean
@Conditional(OnProductionCondition.class)
public DataService productionDataService() {
    return new ProductionDataService();
}

```

```

@AliasFor("name")
String[] value() default {};

```

ComponentScan

```

package org.springframework.context.annotation;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan

```

The @ComponentScan annotation in Spring is used to specify the base packages to scan for Spring-managed components such as @Component, @Service, @Repository, @Controller, and other user-defined annotations. It instructs Spring to detect and register beans (components) within the specified package(s) and its sub-packages.

Basic Usage

```

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.example.app")
public class AppConfig {
    // Configuration settings and other bean definitions
}

```

In this example, @ComponentScan is used in conjunction with @Configuration to enable component scanning in the com.example.app package and its sub-packages.

Spring will scan this package and register beans annotated with @Component, @Service, @Repository, @Controller, etc.

Multiple Base Packages

```
@ComponentScan(basePackages = {"com.example.service", "com.example.repository"})
```

Include and Exclude Filters

@ComponentScan allows you to specify include and exclude filters to customize component scanning behavior.

```
@ComponentScan(basePackages = "com.example", includeFilters = @Filter(type = ASSIGNABLE_TYPE,
```

```
classes = MyService.class),
excludeFilters = @Filter(type = ASSIGNABLE_TYPE, classes = MyController.class))
In this example, only beans of type MyService will be included, while beans of type MyController will be excluded from component scanning.
```

```
@AliasFor("basePackages")
String[] value() default {};
Filter[] includeFilters() default {};
Filter[] excludeFilters() default {};
```

Conditional

```
package org.springframework.context.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Conditional
```

The `@Conditional` annotation in Spring allows you to conditionally include or exclude a bean declaration **based on a specific condition or set of conditions**.

It provides a flexible mechanism to define beans dynamically, depending on runtime conditions, environment properties, or configuration settings.

Basic Usage

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Conditional;

@Configuration
public class AppConfig {

    @Bean
    @Conditional(MyCondition.class)
    public DataService dataService() {
        return new DataServiceImpl();
    }
}
```

In this example, `@Conditional` is used to specify that the `dataService()` bean should be created only if `MyCondition` evaluates to true.

If the condition is met, Spring will instantiate and register `DataService` as a bean; otherwise, it will be skipped.

Standard Conditions

Spring provides several standard conditions that can be used out-of-the-box:

```
@ConditionalOnBean
@ConditionalOnClass
@ConditionalOnProperty
@ConditionalOnMissingBean
@ConditionalOnMissingClass
```

These conditions allow you to control bean registration based on the presence of other beans, classes, properties, or their absence in the application context.

Custom Conditions

You can define custom conditions by implementing the `Condition` interface or extending from `SpringBootCondition` for Spring Boot applications.

```
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class MyCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
```

```

        // Condition logic based on context and metadata
        return true; // Return true to include the bean, false to exclude
    }
}

```

Implement the `matches` method to define custom logic based on environment properties, system state, or any other criteria.

Configuration

```

package org.springframework.context.annotation;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration

```

The `@Configuration` annotation in Spring is used to indicate that a class **declares one or more `@Bean` methods** and **may be processed** by the Spring container **to generate bean definitions and service requests** for those beans at runtime.

Declaring Configuration Classes

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public DataService dataService() {
        return new DataServiceImpl();
    }

    @Bean
    public UserService userService() {
        return new UserServiceImpl();
    }
}

```

In this example, `@Configuration` is used to mark `AppConfig` as a configuration class.

`AppConfig` contains `@Bean` methods (`dataService()` and `userService()`) that define Spring beans.

Bean Definition

Beans defined within a `@Configuration` class are managed by the Spring IoC container.

They are instantiated, configured, and managed as **singleton instances** within the Spring application context.

Dependency Injection

`@Configuration` classes support dependency injection between beans using `@Autowired`, `@Inject`, or `constructor injection`.

```

@Configuration
public class AppConfig {

    @Bean
    public UserService userService(DataService dataService) {
        return new UserServiceImpl(dataService);
    }
}

```

```
boolean proxyBeanMethods() default true;
```

DependsOn

```

package org.springframework.context.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)

```

@Documented

public @interface **DependsOn**

The `@DependsOn` annotation in Spring is used to declare bean dependencies explicitly.

It ensures that the beans specified in the `@DependsOn` annotation **are initialized before the annotated bean**.

This annotation helps manage complex bean dependencies and initialization order in the Spring application context.

Declaring Dependencies

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;

@Configuration
public class AppConfig {

    @Bean
    public DataService dataService() {
        return new DataServiceImpl();
    }

    @Bean
    @DependsOn("dataService")
    public UserService userService() {
        return new UserServiceImpl();
    }
}
```

In this example, `userService()` is annotated with `@DependsOn("dataService")`, indicating that the bean `userService` depends on `dataService`.

Spring ensures that `dataService` is fully initialized before `userService` during application context startup.

Multiple Dependencies

You can specify multiple dependencies using an array of bean names in `@DependsOn`.

```
@Bean
@DependsOn({"bean1", "bean2"})
public AnotherService anotherService() {
    return new AnotherServiceImpl();
}
```

EnableAspectJAutoProxy

package org.springframework.context.annotation;

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Import(AspectJAutoProxyRegistrar.class)

public @interface **EnableAspectJAutoProxy**

The `@EnableAspectJAutoProxy` annotation in Spring is used to enable support for **aspect-oriented programming** (AOP) using AspectJ annotations.

AOP allows you to modularize cross-cutting concerns, such as logging, security, and transaction management, which can be applied across different parts of your application.

boolean `proxyTargetClass()` default false;

Boolean flag (default `false`) to specify whether to proxy using CGLIB instead of JDK dynamic proxies.

This is useful when the target object does not implement interfaces.

boolean `exposeProxy()` default false;

Boolean flag (default `false`) **to expose the proxy to the current thread**.

Useful when invoking methods on the proxy itself (e.g., for accessing thread-bound objects like transaction contexts).

Enabling AspectJ Support

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
    // Other configuration beans and methods
}
```

Import

```
package org.springframework.context.annotation;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Import
```

The `@Import` annotation in Spring is used to **import one or more configuration classes** into a configuration class, typically annotated with `@Configuration`.

This allows you **to combine multiple configurations** into a single one, facilitating modular and organized configuration management within a Spring application.

The annotation class will be loaded into the specific container class and does not need to be declared as a bean.

【TYPE】

在被导入的类中可以继续使用`@Import` 导入其他资源

`@Import` 注解在同一个类上，仅允许添加一次，如果需要导入多个，使用数组的形式进行设定

导入内容：导入 Bean

 导入配置类

 导入 ImportSelector 实现类。一般用于加载配置文件中的类（AutoConfigurationImportSelector）

 导入 ImportBeanDefinitionRegistrar 实现类。

```
Class<?>[] value();
```

Importing Configuration Classes

You can import one or more configuration classes using `@Import`.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({AppConfig1.class, AppConfig2.class})
public class MainConfig {
    // Configuration beans and methods
}
```

Importing Configuration Components

Apart from importing configuration classes, you can also import **other Spring components** such as `@Bean` definitions or `@ComponentScan` configurations.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({MyBeanConfiguration.class, MyComponentScanConfiguration.class})
public class MainConfiguration {
    // Configuration beans and methods
}
```

Conditional Import

@Import can also be conditionally applied using the `@Conditional` annotation.

This allows you to import configurations based on certain conditions being met at runtime.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Conditional;

@Configuration
@Conditional(OnProductionCondition.class)
@Import(ProductionConfig.class)
public class AppConfig {
    // Configuration beans and methods
}
```

ImportResource

```
package org.springframework.context.annotation;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
public @interface ImportResource
```

The `@ImportResource` annotation in Spring is used to import XML configuration files into a Java-based Spring configuration class (`@Configuration`).

This allows you to combine XML-based and Java-based configurations within the same application context, leveraging the strengths of both approaches.

Importing XML Configuration

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

@Configuration
@ImportResource("classpath:applicationContext.xml")
public class AppConfig {
    // Java-based bean configurations and other methods
}
```

The XML configuration defined in `applicationContext.xml` will be merged with the Java-based bean definitions in `AppConfig`.

Multiple XML Files

You can import multiple XML configuration files using `@ImportResource`, providing an array of file paths.

```
@Configuration
@ImportResource({"classpath:applicationContext.xml", "classpath:spring-beans.xml"})
public class AppConfig {
    // Java-based bean configurations and other methods
}
```

Resource Locations

Resource locations specified in `@ImportResource` can be classpath-relative (classpath: prefix) or file system-relative (file: prefix).

```
@ImportResource({"classpath:config/applicationContext.xml", "file:/path/to/spring-beans.xml"})
```

```
@AliasFor("locations")
```

```
String[] value() default {};
```

Lazy

```
package org.springframework.context.annotation;
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.PARAMETER,
ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

public @interface Lazy

The `@Lazy` annotation in Spring is used to indicate **that a bean should be lazily initialized**.

Lazy initialization means that the bean will be created by the Spring container **only when it is first requested**, rather than eagerly at startup.

This can be beneficial for improving application startup time and reducing resource consumption, especially for beans that are not immediately needed.

Lazy Initialization

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;

@Configuration
public class AppConfig {

    @Bean
    @Lazy
    public DataService dataService() {
        return new DataServiceImpl();
    }

    @Bean
    public UserService userService() {
        return new UserServiceImpl();
    }
}
```

In this example, `dataService()` is annotated with `@Lazy`, indicating that the `DataService` bean should be lazily initialized.

By default, beans in Spring are eagerly initialized during application startup. Adding `@Lazy` ensures that `DataService` is initialized only when it is first requested by another bean or component.

Proxy-based Mechanism

Spring implements lazy initialization using a proxy-based mechanism.

When a bean is marked with `@Lazy`, Spring creates a proxy around it.

The actual bean instantiation is deferred until a method on the bean is called for the first time.

Injection Point

Lazy initialization affects the injection point of the bean. The proxy is injected initially, and the actual bean instance is created **upon the first method call**.

```
@Autowired
@Lazy
private DataService dataService;
```

Here, `dataService` is injected as a proxy. The actual `DataService` instance is created **when `dataService` methods are invoked for the first time**.

PropertySource

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(PropertySources.class)
public @interface PropertySource
```

The `@PropertySource` annotation in Spring is used to declare **a property source file** in a Spring application context.

It allows you to externalize configuration properties from your Spring application's core codebase into a properties file or other supported formats, enhancing flexibility and manageability.

Declaring Property Source

```

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {
    // Bean definitions and other configuration
}

```

In this example, `@PropertySource` is used to declare `application.properties` as the property source for the Spring application context.

The properties defined in `application.properties` can then be accessed and used in the application.

Multiple Property Sources

You can specify multiple property sources using an array of file paths.

```
@PropertySource({"classpath:config/app.properties", "file:/path/to/custom.properties"})
```

Property Resolution

Properties defined in the specified property source(s) can be accessed using Spring's Environment or `@Value` annotations.

```
import org.springframework.beans.factory.annotation.Value;
```

```

@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {

    @Value("${app.url}")
    private String appUrl;

    // Bean definitions and other configuration
}

```

In this example, `${app.url}` resolves to the value of `app.url` property defined in `application.properties`.

```

String name() default "";
boolean ignoreResourceNotFound() default false;

```

Profile

```

package org.springframework.context.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile

```

The `@Profile` annotation in Spring is used to indicate that a component, configuration class, or bean declaration should be active only when the specified profiles are active in the Spring environment.

This annotation helps in creating beans conditionally based on the environment or deployment context.

```

String[] value();
// value= "!prod"  Non-prod environment

```

Primary

```

package org.springframework.context.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Primary

```

The `@Primary` annotation in Spring is used to indicate a primary bean **when multiple beans of the same type are present** in the application context.

It resolves ambiguity when autowiring beans by specifying one of them as the primary candidate to be injected.

Primary Bean Declaration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
public class AppConfig {

    @Bean
    public DataService primaryDataService() {
        return new PrimaryDataService();
    }

    @Bean
    @Primary
    public DataService secondaryDataService() {
        return new SecondaryDataService();
    }
}
```

Resolution of Bean Conflicts

When multiple beans of the same type are available and **no specific bean is specified**, Spring will inject the primary bean.

```
@Autowired
private DataService dataService;
```

Scope

```
package org.springframework.context.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Scope
String value() default "";
ConfigurableBeanFactory.SCOPE_SINGLETON
```

This is the default scope for beans. It means **that only one instance of the bean will be created and shared throughout the entire Spring application context.**

ConfigurableBeanFactory.SCOPE_PROTOTYPE

A new instance of the bean will be created **each time it's requested.**

WebApplicationContext.SCOPE_REQUEST

A new instance of the bean will be created **for each HTTP request within a web application.**

WebApplicationContext.SCOPE_SESSION

A new instance of the bean will be created **for each HTTP session within a web application.**

```
String scopeName() default "";
ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;
```

ScopedProxyMode.TARGET_CLASS By default, Spring uses [CGLIB library](#) to directly subclass the objects.

ScopedProxyMode.INTERFACES To avoid CGLIB usage, we can configure the proxy mode with

ScopedProxyMode.INTERFACES, to use the [JDK dynamic proxy](#) instead.

support

AbstractApplicationContext

```
package org.springframework.context.support;
public abstract class AbstractApplicationContext
```

```

extends DefaultResourceLoader implements ConfigurableApplicationContext

refresh
public void refresh() throws BeansException, IllegalStateException {      // Implements initialization of the
    spring container
    synchronized(this.startupShutdownMonitor) {
        StartupStep contextRefresh = this.applicationStartup.start("spring.context.refresh");
        this.prepareRefresh();                                // Papare the spring container in
        advance, record startup events and tags.
        ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();    // Create a
        BeanFactory, destroy it if it already exists, and create it if it does not. (which implements loading of bean
        definitions)
        this.prepareBeanFactory(beanFactory);      // Config stand contexts features for the BeanFactory, such
        as class loaders, PostProcesser etc.

        try {
            this.postProcessBeanFactory(beanFactory);           //Allow some processing of the created
            BeanFactory in the spring context subclass.
            StartupStep beanPostProcess = this.applicationStartup.start("spring.context.beans.post-process");
            this.invokeBeanFactoryPostProcessors(beanFactory);  //Calls the BeanFactoryPostProcessor
            registered as a bean in the spring context, when the bean has not start instantiation.
            this.registerBeanPostProcessors(beanFactory);       //Register the BeanPostProcessor created to
            intercept beans.
            beanPostProcess.end();
            this.initMessageSource();
            this.initApplicationEventMulticaster();             // initialize context event broadcast.
            this.onRefresh();                                  // Template Method
            this.registerListeners();
            this.finishBeanFactoryInitialization(beanFactory); // Complete the initialization of the
            container (beanFactory.preInstantiateSingletons inside will complete the creation of singleton objects)
            this.finishRefresh();
        } catch (BeansException var10) {
            if (this.logger.isWarnEnabled()) {
                this.logger.warn("Exception encountered during context initialization - cancelling refresh
attempt: " + var10);
            }

            this.destroyBeans();          // destroys the created singleton to avoid resource overhang.
            this.cancelRefresh(var10);   // reset active
            throw var10;
        } finally {
            this.resetCommonCaches();   // Resets common introspection caching in the core of Spring, as we
            may no longer need metadata for singleton beans.
            contextRefresh.end();
        }
    }
}

```

```

publishEvent
protected void publishEvent(Object event, @Nullable ResolvableType eventType) {
    Assert.notNull(event, "Event must not be null");
    Object applicationEvent;
    if (event instanceof ApplicationEvent applEvent) {
        applicationEvent = applEvent;
    } else {
        applicationEvent = new PayloadApplicationEvent(this, event, eventType);
        if (eventType == null) {
            eventType = ((PayloadApplicationEvent)applicationEvent).getResolvableType();
        }
    }

    if (this.earlyApplicationEvents != null) {
        this.earlyApplicationEvents.add(applicationEvent);
    }
}

```

```

        } else {
            this.getApplicationEventMulticaster().multicastEvent((ApplicationEvent)applicationEvent, eventType);
        }

        if (this.parent != null) {
            ApplicationContext var5 = this.parent;
            if (var5 instanceof AbstractApplicationContext) {
                AbstractApplicationContext abstractApplicationContext = (AbstractApplicationContext)var5;
                abstractApplicationContext.publishEvent(event, eventType);
            } else {
                this.parent.publishEvent(event);
            }
        }
    }
}

```

initApplicationEventMulticaster

```

protected void initApplicationEventMulticaster() {
    ConfigurableListableBeanFactory beanFactory = this.getBeanFactory();
    if (beanFactory.containsLocalBean("applicationEventMulticaster")) {      //Determine whether the custom
ApplicationEventMulticaster exists.
        this.applicationEventMulticaster =
(ApplicationEventMulticaster)beanFactory.getBean("applicationEventMulticaster",
ApplicationEventMulticaster.class);
        if (this.logger.isTraceEnabled()) {
            this.logger.trace("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster +
"]);
        }
    } else {
        this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
        beanFactory.registerSingleton("applicationEventMulticaster", this.applicationEventMulticaster);
        if (this.logger.isTraceEnabled()) {
            this.logger.trace("No 'applicationEventMulticaster' bean, using [" +
this.applicationEventMulticaster.getClass().getSimpleName() + "]");
        }
    }
}

```

registerListeners

```

protected void registerListeners() {
    Iterator var1 = this.getApplicationListeners().iterator();

    while(var1.hasNext()) {
        ApplicationListener<?> listener = (ApplicationListener)var1.next();
        this.getApplicationEventMulticaster().addApplicationListener(listener);
    }

    String[] listenerBeanNames = this.getBeanNamesForType(ApplicationListener.class, true, false);
    String[] var7 = listenerBeanNames;
    int var3 = listenerBeanNames.length;

    for(int var4 = 0; var4 < var3; ++var4) {
        String listenerBeanName = var7[var4];
        this.getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
    }

    Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
    this.earlyApplicationEvents = null;
    if (!CollectionUtils.isEmpty(earlyEventsToProcess)) {
        Iterator var9 = earlyEventsToProcess.iterator();

        while(var9.hasNext()) {
            ApplicationEvent earlyEvent = (ApplicationEvent)var9.next();
            this.getApplicationEventMulticaster().multicastEvent(earlyEvent);
        }
    }
}

```

```

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
    if (beanFactory.containsBean("conversionService") && beanFactory.isTypeMatch("conversionService",
ConversionService.class)) {
        beanFactory.setConversionService((ConversionService)beanFactory.getBean("conversionService",
ConversionService.class));
    }
    if (!beanFactory.hasEmbeddedValueResolver()) {
        beanFactory.addEmbeddedValueResolver((strVal) -> {
            return this.getEnvironment().resolvePlaceholders(strVal);
        });
    }
    String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
    String[] var3 = weaverAwareNames;
    int var4 = weaverAwareNames.length;
    for(int var5 = 0; var5 < var4; ++var5) {
        String weaverAwareName = var3[var5];
        this.getBean(weaverAwareName);
    }
    beanFactory.setTempClassLoader((ClassLoader)null);
    beanFactory.freezeConfiguration();
    beanFactory.preInstantiateSingletons();
}

```

ClassPathXmlApplicationContext

```

package org.springframework.context.support;
public class ClassPathXmlApplicationContext extends AbstractXmlApplicationContext      加载 xml 格式上下文， 默认从类
路径加载配置文件 // new ClassPathXmlApplicationContext("applicationContext.xml");
public ClassPathXmlApplicationContext(String... configLocations)    加载多个配置文件

```

public Object getBean(String name) 获取 bean

FileSystemXmlApplicationContext

```

package org.springframework.context.support;
public class FileSystemXmlApplicationContext extends AbstractXmlApplicationContext

```

event

ContextRefreshedEvent

```

package org.springframework.context.event;
public class ContextRefreshedEvent extends ApplicationContextEvent 上下文更新事件，该事件会在 ApplicationContext 被初
始化或者更新时发布。也可以在调用 ConfigurableApplicationContext 接口中的 refresh()方法时被触发。

```

ContextStartedEvent

```

package org.springframework.context.event;
public class ContextStartedEvent extends ApplicationContextEvent 上下文开始事件，当容器调用
ConfigurableApplicationContext 的 Start()方法开始/重新开始容器时触发该事件。

```

ContextStoppedEvent

```
package org.springframework.context.event;
public class ContextStoppedEvent extends ApplicationContextEvent 上下文停止事件，当容器调用
ConfigurableApplicationContext 的 Stop()方法停止容器时触发该事件。
```

ContextClosedEvent

```
package org.springframework.context.event;
public class ContextClosedEvent extends ApplicationContextEvent 上下文关闭事件，当 ApplicationContext 被关闭时触发该事
件。容器被关闭时，其管理的所有单例 Bean 都被销毁。
```

ApplicationEventMulticaster

```
package org.springframework.context.event;
public interface ApplicationEventMulticaster 表示事件广播操作。
```

AbstractApplicationEventMulticaster

```
package org.springframework.context.event;
public abstract class AbstractApplicationEventMulticaster implements ApplicationEventMulticaster, BeanClassLoaderAware,
BeanFactoryAware
private final DefaultListenerRetriever defaultRetriever = new DefaultListenerRetriever();
final Map<ListenerCacheKey, CachedListenerRetriever> retrieverCache = new ConcurrentHashMap(64);
private ClassLoader beanClassLoader;
private ConfigurableBeanFactory beanFactory;
```

DefaultListenerRetriever

```
private class DefaultListenerRetriever {
    public final Set<ApplicationListener<?>> applicationListeners = new LinkedHashSet();
    public final Set<String> applicationListenerBeans = new LinkedHashSet();

    private DefaultListenerRetriever() {
    }

    public Collection<ApplicationListener<?>> getApplicationListeners() {
        List<ApplicationListener<?>> allListeners = new ArrayList(this.applicationListeners.size() +
this.applicationListenerBeans.size());
        allListeners.addAll(this.applicationListeners);
        if (!this.applicationListenerBeans.isEmpty()) {
            BeanFactory beanFactory = AbstractApplicationEventMulticaster.this.getBeanFactory();
            Iterator var3 = this.applicationListenerBeans.iterator();

            while(var3.hasNext()) {
                String listenerBeanName = (String)var3.next();

                try {
                    ApplicationListener<?> listener =
(ApplicationListener)beanFactory.getBean(listenerBeanName, ApplicationListener.class);
                    if (!allListeners.contains(listener)) {
                        allListeners.add(listener);
                    }
                } catch (NoSuchBeanDefinitionException var6) {
                }
            }
        }
        AnnotationAwareOrderComparator.sort(allListeners);
        return allListeners;
    }
}
```

SimpleApplicationEventMulticaster

```
package org.springframework.context.event;
public class SimpleApplicationEventMulticaster extends AbstractApplicationEventMulticaster 应用事件广播器接口
    (annotation)
```

EventListener

```
package org.springframework.context.event;
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Reflective
public @interface EventListener

@AliasFor("classes")
Class<?>[] value() default {};
```

core
(Reflection)

ParameterNameDiscoverer

```
package org.springframework.core;
public interface ParameterNameDiscoverer
    获取方法名称
```

MethodParameter

```
package org.springframework.core;
public class MethodParameter
```

(Container)

Ordered

```
package org.springframework.core;
public interface Ordered 来处理相同接口实现类的优先级问题。 // aop 实现类执行顺序
int HIGHEST_PRECEDENCE = Integer.MIN_VALUE; 最高优先级
int LOWEST_PRECEDENCE = Integer.MAX_VALUE; 最低优先级
```

PriorityOrdered

```
package org.springframework.core;
public interface PriorityOrdered extends Ordered PriorityOrdered 优先级大于 Ordered, 相同类型情况下比较 getOrder 的得到
    的优先级
```

LocalVariableTableParameterNameDiscoverer

```
package org.springframework.core;
public class LocalVariableTableParameterNameDiscoverer implements ParameterNameDiscoverer 获取参数名
    public String[] getParameterNames(Method method) 从方法中获取参数列表
```

MethodIntrospector

```
package org.springframework.core;
```

```

public final class MethodIntrospector 方法内省
public static <T> Map<Method, T> selectMethods(Class<?> targetType, final MethodIntrospector.MetadataLookup<T>
metadataLookup) 遍历传入的类的所有方法, 然后每次遍历一个方法都会调用传入的 MetadataLookup
    找出所有的类, 包括类和接口
    遍历类和接口
    调用回调函数
public interface MetadataLookup<T> {
    @Nullable
    T inspect(Method method);
}

```

convert

Converter (转换器处理)

```

package org.springframework.core.convert.converter;
public interface Converter<S, T> 参数类型转换, 通过注册自定义转换器, 将该功能加入到 springMvc 的转换服务
converterservice 中
    配置: applicationContext.xml 中将 bean 配置到 ConversionServiceFactoryBean 类的属
性中
    T convert(S var1); S 类型转换为 T 类型, 具体在实现类里指定

```

实现类

标量转换器

stringToBooleanConverter	String-Boolean
objectToStringconverter	Object -> String
stringToNumberConverterFactory	String -> Number (Inteder、 Long 等)
NumberToNumberConverterFactory	Number 子类型之间(Integer、 Long、 Double 等)
stringToCharacterConverter	string -> java.lang.Character
NunberToCharacterConverter	Number 子类型(Integer、 Long、 Double 等) -> java.lang.Character
CharacterToNumberFactory	java.lang.character -> Number 子类型(Integer、 Long、 Double 等)
stringToEnumconverterFactory	String -> enum 类型
EnumToStringconverter	enum 类型 -> string
stringToLocalConverter	string -> java.util.Local
PropertiesToStringConverter	java.util.Properties -> String
stringToPropsConverter	string -> java.util.Properties

集合、数组相关转换器

ArrayToCollectionconverter	数组 -> 集合(List、 set)
collectionToArrayConverter	集合(List、 set) -> 数组
ArrayToArrayconverter	数组间
collectionToCollectionConverter	集合间(List、 set)
MapToMapconverter	Map 间
ArrayToStringConverter	数组 -> string 类型
stringToArrayConverter	string -> 数组, trim 后使用","split
ArrayToobjectconverter	数组 -> object
objectToArrayConverter	object -> 单元素数组
collectionToStringConverter	集合(List、 set) -> string

stringTocollectionconverter	string -> 集合(List、set), trim 后使用 ",".split
collectionTo0bjectconverter	集合 -> object
objectTocollectionConverter	object -> 单元素集合

默认转换器

objectToobjectconverter	object 间
IdToEntityconverter	Id -> Entity
FallbackobjectTostringConverter	object-string

env

Environment

```
package org.springframework.core.env;
public interface Environment extends PropertyResolver
```

The Environment interface in Spring's core.env package is a central abstraction for working with profiles and properties in a Spring application. It is used to retrieve property values and check the active profiles within the application context.

Key Features

- **Property Resolution**
Allows resolving properties **from various sources** (e.g., properties files, system properties, environment variables).
- **Profile Management**
Provides methods **to check the active and default profiles**.
- **Property Sources**
Integrates with multiple property sources through the **PropertySources** abstraction.

```
boolean acceptsProfiles(Profiles profiles);
```

Basic Property Retrieval

```
import org.springframework.core.env.Environment;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component
public class MyBean {
```

```
    @Autowired
    private Environment env;

    public void printProperty() {
        String propertyValue = env.getProperty("my.property.key");
        System.out.println("Property value: " + propertyValue);
    }
}
```

Profile Management

```
import org.springframework.core.env.Environment;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component
public class ProfileChecker {
```

```
    @Autowired
    private Environment env;

    public void checkProfiles() {
        if (env.acceptsProfiles("development")) {
            System.out.println("Development profile is active");
        }
    }
}
```

```
}
```

PropertyResolver

```
package org.springframework.core.env;  
public interface PropertyResolver  
  
<T> T getProperty(String var1, Class<T> var2);
```

CommandLinePropertySource

```
package org.springframework.core.env;  
public abstract class CommandLinePropertySource<T> extends EnumerablePropertySource<T>  
public CommandLinePropertySource(String name, T source)  
public CommandLinePropertySource(T source)  
public final String getProperty(String name)  
public final boolean containsProperty(String name)  
  
protected abstract boolean containsOption(String name); Extend behavior for containsProperty method  
protected abstract List<String> getOptionValues(String name); Extend behavior for getProperty method  
protected abstract List<String> getNonOptionArgs(); Extend behavior for getProperty method
```

```
public void setNonOptionArgsPropertyName(String nonOptionArgsPropertyName) Extend state for  
EnumerablePropertySource class
```

PropertySource

```
package org.springframework.core.env;  
public abstract class PropertySource<T>  
public PropertySource(String name, T source)  
public PropertySource(String name)  
public abstract Object getProperty(String name);  
public boolean containsProperty(String name)  
  
public String getName()  
public T getSource()  
public static PropertySource<?> named(String name)  
static class ComparisonPropertySource extends StubPropertySource  
public static class StubPropertySource extends PropertySource<Object>
```

io

InputStreamSource

```
package org.springframework.core.io;  
public interface InputStreamSource 输入资源文件源  
InputStream getInputStream() throws IOException; 获取资源文件输入流
```

Resource

```
package org.springframework.core.io;  
public interface Resource extends InputStreamSource 资源文件  
    @Value(value="classpath:images/test.png") private Resource xxx; 直接注入资源  
String getFilename(); 获取文件名称
```

```
public class UrlResource extends AbstractFileResolvingResource 路径资源
public UrlResource(URL url) 构造一个 Url 资源
```

ResourceLoader

```
package org.springframework.core.io;
public interface ResourceLoader 资源加载器
Resource getResource(String location);
```

support

ResourcePatternResolver

```
package org.springframework.core.io.support;
public interface ResourcePatternResolver 资源文件处理
```

PathMatchingResourcePatternResolver

```
package org.springframework.core.io.support;
public class PathMatchingResourcePatternResolver 资源文件处理
    implements ResourcePatternResolver
public Resource getResource(String location) 获得资源文件 // resource.getResource("classpath:mapper/CityMapper.xml")
```

```
package org.springframework.core.io.support;
public abstract class PropertiesLoaderUtils 属性文件加载器
public static Properties loadProperties(Resource resource) throws IOException 从资源文件中加载属性文件
```

SpringFactoriesLoader

```
package org.springframework.core.io.support;
public final class SpringFactoriesLoader spring.factories 文件加载器
public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
private static final FailureHandler THROWING_FAILURE_HANDLER = SpringFactoriesLoader.FailureHandler.throwing();
private static final Log logger = LogFactory.getLog(SpringFactoriesLoader.class);
static final Map<ClassLoader, Map<String, SpringFactoriesLoader>> cache = new ConcurrentHashMap<>();
@Nullable
private final ClassLoader classLoader;
private final Map<String, List<String>> factories;

public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader)
```

forDefaultResourceLocation

```
public static SpringFactoriesLoader forDefaultResourceLocation(@Nullable ClassLoader classLoader) {
    return forResourceLocation("META-INF/spring.factories", classLoader); // get a instance of
SpringFactoriesLoader
}
```

forResourceLocation

```
public static SpringFactoriesLoader forResourceLocation(String resourceLocation, @Nullable ClassLoader
classLoader) {
    Assert.hasText(resourceLocation, "'resourceLocation' must not be empty");
    ClassLoader resourceClassLoader = classLoader != null ? classLoader :
SpringFactoriesLoader.class.getClassLoader();
    Map<String, SpringFactoriesLoader> loaders = (Map)cache.computeIfAbsent(resourceClassLoader, (key) -> {
        return new ConcurrentHashMap<>();
    });
    return (SpringFactoriesLoader)loaders.computeIfAbsent(resourceLocation, (key) -> {
```

```

        return new SpringFactoriesLoader(classLoader, loadFactoriesResource(resourceClassLoader,
resourceLocation));
    });
}

SpringFactoriesLoader
protected SpringFactoriesLoader(@Nullable ClassLoader classLoader, Map<String, List<String>> factories) {
    this.classLoader = classLoader;
    this.factories = factories;
}

loadFactoriesResource
protected static Map<String, List<String>> loadFactoriesResource(ClassLoader classLoader, String
resourceLocation) {
    Map<String, List<String>> result = new LinkedHashMap();

    try {
        Enumeration<URL> urls = classLoader.getResources(resourceLocation);

        while(urls.hasMoreElements()) {
            UrlResource resource = new UrlResource((URL)urls.nextElement());
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            properties.forEach((name, value) -> {
                List<String> implementations = (List)result.computeIfAbsent(((String)name).trim(), (key) -> {
                    return new ArrayList();
                });
                Stream var10000 =
Arrays.stream(StringUtils.commaDelimitedListToStringArray((String)value)).map(String::trim);
                Objects.requireNonNull(implementations);
                var10000.forEach(implementations::add);
            });
        }

        result.replaceAll(SpringFactoriesLoader::toDistinctUnmodifiableList);
        return Collections.unmodifiableMap(result);
    } catch (IOException var6) {
        throw new IllegalArgumentException("Unable to load factories from location [" + resourceLocation +
"]", var6);
    }
}

toDistinctUnmodifiableList
private static List<String> toDistinctUnmodifiableList(String factoryType, List<String> implementations) {
    return implementations.stream().distinct().toList();
}

load
public <T> List<T> load(Class<T> factoryType, @Nullable ArgumentResolver argumentResolver) {
    return this.load(factoryType, argumentResolver, (FailureHandler)null);
}

load
public <T> List<T> load(Class<T> factoryType, @Nullable ArgumentResolver argumentResolver, @Nullable
FailureHandler failureHandler) {
    Assert.notNull(factoryType, "'factoryType' must not be null");
    List<String> implementationNames = this.loadFactoryNames(factoryType);
    logger.trace(LogMessage.format("Loaded [%s] names: %s", factoryType.getName(), implementationNames));
    List<T> result = new ArrayList(implementationNames.size());
    FailureHandler failureHandlerToUse = failureHandler != null ? failureHandler : THROWING_FAILURE_HANDLER;
    Iterator var7 = implementationNames.iterator();
}

```

```

        while(var7.hasNext()) {
            String implementationName = (String)var7.next();
            T factory = this.instantiateFactory(implementationName, factoryType, argumentResolver,
failureHandlerToUse);
            if (factory != null) {
                result.add(factory);
            }
        }

        AnnotationAwareOrderComparator.sort(result);
        return result;
    }
}

```

loadFactoryNames

```

private List<String> loadFactoryNames(Class<?> factoryType) {
    return (List)this.factories.getOrDefault(factoryType.getName(), Collections.emptyList());
}

```

PropertiesLoaderUtils

```

package org.springframework.core.io.support;
public abstract class PropertiesLoaderUtils

```

loadProperties

```

public static Properties loadProperties(Resource resource) throws IOException {
    Properties props = new Properties();
    fillProperties(props, resource);
    return props;
}

```

fillProperties

```

public static void fillProperties(Properties props, Resource resource) throws IOException {
    InputStream is = resource.getInputStream();

    try {
        String filename = resource.getFilename();
        if (filename != null && filename.endsWith(".xml")) {
            props.loadFromXML(is);
        } else {
            props.load(is);
        }
    } catch (Throwable var6) {
        if (is != null) {
            try {
                is.close();
            } catch (Throwable var5) {
                var6.addSuppressed(var5);
            }
        }
        throw var6;
    }

    if (is != null) {
        is.close();
    }
}

```

task

TaskExecutor

```
package org.springframework.core.task;  
public interface TaskExecutor extends Executor 异步执行  
void execute(Runnable task);
```

SyncTaskExecutor

```
package org.springframework.core.task;  
public class SyncTaskExecutor implements TaskExecutor, Serializable 不异步执行调用。相反，每次调用都在调用线程中进行。它主要用于不需要多线程的情况下，比如在简单的测试用例中。
```

SimpleAsyncTaskExecutor

```
package org.springframework.core.task;  
public class SimpleAsyncTaskExecutor extends CustomizableThreadCreator implements AsyncListenableTaskExecutor, Serializable
```

TaskExecutor implementation that fires up a new Thread for each task, executing it asynchronously.

Supports limiting concurrent threads through `setConcurrencyLimit(int)`. By default, the number of concurrent task executions is unlimited.

This limit will block any calls that exceed it until a slot is released.

NOTE: This implementation does not reuse threads!

Consider a thread-pooling TaskExecutor implementation instead, in particular for executing a large number of short-lived tasks.

TaskDecorator

```
package org.springframework.core.task;  
public interface TaskDecorator 任务装饰器：在自定义的异步线程池 ThreadPoolTaskExecutor 中，初始化线程池时有 taskDecorator 这样一个任务装饰器，类似 aop，可对线程执行方法的始末进行增强。
```

RequestContextHolder 在父线程销毁的时候，会触发里面的 `resetRequestAttributes` 方法（即清除 `threadLocal` 里面的信息，即 `request` 中的信息会被清除），此时即使 `RequestContextHolder` 这个对象还是存在，子线程也无法继续使用它获取 `request` 中的数据了，父子线程结束时间不同会导致有时可行有时不可行

绕过销毁清除：，思路不变，还是继续使用 `threadLocal` 来传递我们需要使用到的变量，在父线程装饰前将所需变量取出来，然后在子线程中设置到 `threadLocal`，业务使用的时候从 `threadLocal` 中取即可。

三方：`TransmittableThreadLocal`

测试使用

```
public class ContextCopyingDecorator implements TaskDecorator {  
    @Override  
    public Runnable decorate(Runnable runnable) {  
        try {  
            HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.getRequestAttributes()).getRequest();  
            //获取父线程的 request 的 user-agent(示例)  
            String ua = request.getHeader("user-agent");  
            return () -> {  
                try {  
                    ThreadLocalData.setUa(ua); //将父线程的 ua 设置进子线程里  
                    runnable.run(); //子线程方法执行  
                } finally {  
                    ThreadLocalData.remove(); //清除线程 threadLocal 的值  
                }  
            };  
        }  
    }  
}
```

```

        } catch (IllegalStateException e) {
            return runnable;
        }
    }
}

public class ThreadLocalData {      公共数据
    public static final ThreadLocal<String> threadLocal = new ThreadLocal<>();
    public static String getUa(){
        return threadLocal.get();
    }
    public static void setUa(String ua){
        threadLocal.set(ua);
    }
    public static void remove(){
        threadLocal.remove();
    }
}

```

type

AnnotatedTypeMetadata

```

package org.springframework.core.type;
public interface AnnotatedTypeMetadata          注解元对象，用于获取注解定义的属性值
Map<String, Object> getAnnotationAttributes(String var1)  获取注解属性值

```

TypeFilter

```

package org.springframework.core.type.filter;
public interface TypeFilter  自定义类型过滤器
boolean match(MetadataReader var1, MetadataReaderFactory var2)  是否加载 bean

```

annotation

AnnotationAwareOrderComparator

```

package org.springframework.core.annotation;
public class AnnotationAwareOrderComparator
    extends OrderComparator
public static void sort(List<?> list)      排序类

```

AnnotationUtils

```

package org.springframework.core.annotation;
public abstract class AnnotationUtils      注解工具

```

```

public static <A extends Annotation> A findAnnotation(Method method, @Nullable Class<A> annotationType)      从方法
method 上找到指定类型的注解对象
// MessageLog annotation = AnnotationUtils.findAnnotation(targetMethod, MessageLog.class);

```

注解

```

package org.springframework.core.annotation;

```

@AliasFor 注解中指定的属性可以互相为别名进行使用 【METHOD】

@Order 控制配置类的加载顺序 (多个种类的配置出现后，优先加载系统级的，然后加载业务级的，避免细粒度的加载控制)

Ordered.LOWEST_PRECEDENCE

data
domain

Slice

```
package org.springframework.data.domain;  
public interface Slice<T> extends Streamable<T>    Slice 只关心是不是存在下一个分片/分页，不会去数据库 count 计算总条数、  
总页数（所以比较适合大数据量列表的的鼠标或手指滑屏操作，不关心总共有多少页，只关心有没有下一页）
```

```
List<T> getContent();    获取切片内容  
boolean hasContent();  是否有查询结果  
default Pageable getPageable() 当前切片的分页信息  
boolean isFirst();      是否是第一个切片  
boolean isLast();       是否是最后一个切片  
Pageable nextPageable(); 下一个切片的分页信息  
Pageable previousPageable(); 上一个切片的分页信息  
int getNumber();  
int getSize();  
int getNumberOfElements();  
Sort getSort();  
boolean hasNext();  
boolean hasPrevious();
```

Page

```
package org.springframework.data.domain;  
public interface Page<T> extends Slice<T>    分页结果  
int getTotalPages();    获取总页数  
long getTotalElements(); 获得总数据条数
```

Sort

```
package org.springframework.data.domain;  
public class Sort implements Streamable<Order>, Serializable  
public static Sort by(String... properties)          // Sort.by("firstname").ascending().and(Sort.by("lastname").descending());  
public static Sort by(List<Order> orders)  
public static Sort by(Order... orders)  
public static Sort by(Direction direction, String... properties)
```

Pageable

```
package org.springframework.data.domain;  
public interface Pageable
```

```
static Pageable unpaged()  
static Pageable ofSize(int pageSize)  
default boolean isPaged()  
default boolean isUnpaged()  
int getPageNumber();  
int getPageSize();  
long getOffset();  
Sort getSort();
```

```
default Sort getSortOr(Sort sort)
Pageable next();
Pageable previousOrFirst();
Pageable first();
Pageable withPage(int pageNumber);
boolean hasPrevious();
default Optional<Pageable> toOptional()
```

PageRequest

```
package org.springframework.data.domain;
public class PageRequest extends AbstractPageRequest

public static PageRequest of(int page, int size) // PageRequest.of(0,1) find only the first one
public static PageRequest of(int page, int size, Sort sort)
public static PageRequest of(int page, int size, Sort.Direction direction, String... properties)
```

AbstractPageRequest

```
package org.springframework.data.domain;
public abstract class AbstractPageRequest implements Pageable, Serializable

public abstract Pageable next();
public abstract Pageable previous();
public abstract Pageable first();
```

```
public int getPageSize()
public int getPageNumber()
public long getOffset()
public boolean hasPrevious()
public Pageable previousOrFirst()
```

mongodb

core

MongoTemplate

```
package org.springframework.data.mongodb.core;
public class MongoTemplate implements MongoOperations, ApplicationContextAware, IndexOperationsProvider mongo 模板操作实现类
```

MongoOperations

```
package org.springframework.data.mongodb.core;
public interface MongoOperations extends FluentMongoOperations
```

连接类:

```
com.mongodb.client.MongoClients
com.mongodb.client.MongoClient
```

核心类:

```
com.mongodb.client.MongoDatabase      见 springboot mongodb 源码详解
com.mongodb.client.MongoCollection
```

索引:

org.springframework.data.mongodb.core.index.@Indexed

文档实体类:

org.springframework.data.mongodb.core.mapping.@Document

org.springframework.data.annotation.@Id

查询条件:

org.springframework.data.mongodb.core.query.Query

org.springframework.data.mongodb.core.query.Criteria

org.springframework.data.domain.Sort

更新条件:

org.springframework.data.mongodb.core.query.Update

返回结果:

package com.mongodb.client.result.UpdateResult

package com.mongodb.client.result.DeleteResult

事务:

单节点 mongodb 不支持事务，需要搭建 MongoDB 复制集

@Configuration

public class TransactionConfig {

 @Bean

 MongoTransactionManager transactionManager(MongoDatabaseFactory dbFactory) {

 return new MongoTransactionManager(dbFactory);

 }

}

mongoOperations.aggregate(

 newAggregation(Order.class,

 match(where("id").is(order.getId())), match 匹配

 unwind("items"),

 project("id", "customerId", "items")

 .andExpression("\$items.price * \$items.quantity").as("lineTotal"),

 group("id")

 .sum("lineTotal").as("netAmount")

 .addToSet("items").as("items"),

 project("id", "items", "netAmount")

 .and("orderId").previousOperation()

 .andExpression("netAmount * [0]", taxRate).as("taxAmount")

 .andExpression("netAmount * (1 + [0])", taxRate).as("totalAmount")

),

 Invoice.class);

MongoCollection<Document> getCollection(String collectionName); 获取集合

<O> AggregationResults<O> aggregate(

 Aggregation aggregation, 聚合条件

 String collectionName, 集合名

 Class<O> outputType 输出类型

```

); 聚合查询

<T> T insert(T objectToSave, String collectionName); 插入数据
<T> T save(T objectToSave, String collectionName); 保存数据
DeleteResult remove(Object object, String collectionName);           删除数据
DeleteResult remove(Query query, Class<?> entityClass, String collectionName);   删除数据
DeleteResult remove(Query query, String collectionName);           删除数据
<T> List<T> findAllAndRemove(Query query, String collectionName);       删除多个数据
<T> List<T> findAllAndRemove(Query query, Class<T> entityClass);       删除多个数据
<T> List<T> findAllAndRemove(Query query, Class<T> entityClass, String collectionName);   删除多个数据
UpdateResult updateFirst(Query query, UpdateDefinition update, String collectionName);  更新第一条
UpdateResult updateMulti(Query query, UpdateDefinition update, String collectionName);  更新所有
DeleteResult remove(Object object, String collectionName);           删除一条数据
<T> List<T> findAll(Class<T> entityClass, String collectionName);      查询所有
<T> T findById(Object id, Class<T> entityClass, String collectionName);  根据 id 查询
<T> T findOne(Query query, Class<T> entityClass, String collectionName);  查找一个
<T> List<T> find(Query query, Class<T> entityClass, String collectionName);  查找一个
long count(Query query, String collectionName);  计算数量

```

aggregation

Aggregation

```

package org.springframework.data.mongodb.core.aggregation;
public class Aggregation 聚合查询

public static Aggregation newAggregation(List<? extends AggregationOperation> operations) 间接执行构造
public static Aggregation newAggregation(AggregationOperation... operations)          间接执行构造函数
public static <T> TypedAggregation<T> newAggregation(Class<T> type, List<? extends AggregationOperation> operations)
public static <T> TypedAggregation<T> newAggregation(Class<T> type, AggregationOperation... operations)

public static ProjectionOperation project(String... fields)
public static UnwindOperation unwind(String field)
public static ReplaceRootOperation replaceRoot(String fieldName)
public static LimitOperation limit(long maxElements)
public static AddFieldsOperation.AddFieldsOperationBuilder addFields()

```

index

注解

```

package org.springframework.data.mongodb.core.index;
@Indexed
int expireAfterSeconds() default -1;  创建一个 5 秒之后文档自动删除的索引 // @Indexed(expireAfterSeconds=5)

```

mapping

注解

```

package org.springframework.data.mongodb.core.mapping;
@Document 标注在 mongo 文档实体类上【TYPE】
String value() default "";    集合名称
String collection() default "";  集合名称, 等同 value
String language() default "";

```

query

Query

```
package org.springframework.data.mongodb.core.query;
public class Query      mongo 查询类
public static Query query(CriteriaDefinition criteriaDefinition)
public Query(CriteriaDefinition criteriaDefinition)
public Query with(Sort sort) 添加排序 // Query query = new
Query(Criteria.where("userName").is(userName)).with(Sort.by("age"));
public Query limit(int limit)
public Query skip(long skip)
```

Criteria

```
package org.springframework.data.mongodb.core.query;
public class Criteria implements CriteriaDefinition    查询条件
    筛选操作: Query query= new Query(Criteria.where("id").is(person.getId()));
public static Criteria where(String key)    指定一个搜索 key
public Criteria is(@Nullable Object value) 等于一个值
public Criteria in(Object... values)          $in 查询
public Criteria andOperator(Criteria... criteria)    合 并 两 个 查 询 条 件    // Criteria criteria = new
Criteria().andOperator(criteriaUserName, criteriaPassWord);
public Criteria orOperator(Criteria... criteria)    或两个查询条件
```

Update

```
package org.springframework.data.mongodb.core.query;
public class Update implements UpdateDefinition    更新操作
public Update()    空参创建更新条件
public Update set(String key, @Nullable Object value)    $set 语句
    result
```

UpdateResult

```
package com.mongodb.client.result;
public abstract class UpdateResult    更新结果
public abstract long getMatchedCount();  结果条数
```

DeleteResult

```
package com.mongodb.client.result;
public abstract class DeleteResult    删除结果
public abstract long getDeletedCount();  获取删除条数
```

util

```
package org.springframework.data.util;
public class Lazy<T> implements Supplier<T>    数据懒加载
```

annotation

注解

```
package org.springframework.data.annotation;
@Id  标注主键 【FIELD, METHOD, ANNOTATION_TYPE】
```

jpa

repository

JpaSpecificationExecutor

```
package org.springframework.data.jpa.repository;
public interface JpaSpecificationExecutor<T>
    you can't mix a @Query definition with a Specification
    主要提供了多条件查询的支持，并且可以在查询中添加分页和排序；
```

```
Optional<T> findOne(@Nullable Specification<T> spec);
List<T> findAll(@Nullable Specification<T> spec);
Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable);
List<T> findAll(@Nullable Specification<T> spec, Sort sort);
long count(@Nullable Specification<T> spec);
boolean exists(Specification<T> spec);
long delete(Specification<T> spec);
<S extends T, R> R findBy(Specification<T> spec, Function<FluentQuery.FetchableFluentQuery<S>, R> queryFunction);
```

JpaRepository

```
package org.springframework.data.jpa.repository;
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>, ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T>
    主要对继承父接口中方法的返回值进行了适配，因为在父类接口中通常都返回迭代器，需要我们自己进行强制类型转化。而在 JpaRepository 中，直接返回了 List。
    (QueryByExampleExecutor 接口的定义，支持 Example 动态查询,多条件查询)
```

```
void flush();
<S extends T> S saveAndFlush(S entity);
<S extends T> List<S> saveAllAndFlush(Iterable<S> entities);
default void deleteInBatch(Iterable<T> entities)
void deleteAllInBatch(Iterable<T> entities);
void deleteAllByIdInBatch(Iterable<ID> ids);
void deleteAllInBatch();
T getOne(ID id);
T getById(ID id);
T getReferenceById(ID id);
<S extends T> List<S> findAll(Example<S> example);
<S extends T> List<S> findAll(Example<S> example, Sort sort);
```

[annotation]

Query [CORE]

```
package org.springframework.data.jpa.repository;
public @interface Query
    org.springframework.data.repository.PagingAndSortingRepository
    org.springframework.data.repository.ListCrudRepository
    org.springframework.data.jpa.repository.JpaSpecificationExecutor
```

```
String value() default "";
boolean nativeQuery() default false;
```

```
Class<? extends QueryRewriter> queryRewriter() default QueryRewriter.IdentityQueryRewriter.class;           customs queries  
// queryRewriter = MyQueryRewriter.class
```

support

CrudMethodMetadataPostProcessor

```
package org.springframework.data.jpa.repository.support;  
class CrudMethodMetadataPostProcessor implements RepositoryProxyPostProcessor, BeanClassLoaderAware  
  
static class CrudMethodMetadataPopulatingMethodInterceptor implements MethodInterceptor  
    private static final ThreadLocal<MethodInvocation> currentInvocation = new NamedThreadLocal("Current AOP method invocation");  
    private final ConcurrentMap<Method, CrudMethodMetadata> metadataCache = new ConcurrentHashMap();  
    private final Set<Method> implementations = new HashSet();  
    invoke(CrudMethodMetadataPopulatingMethodInterceptor)  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
        Method method = invocation.getMethod();  
        if (!this.implementations.contains(method)) {  
            return invocation.proceed();  
        } else {  
            MethodInvocation oldInvocation = (MethodInvocation)currentInvocation.get();  
            currentInvocation.set(invocation);  
  
            Object methodMetadata;  
            try {  
                CrudMethodMetadata metadata =  
(CrudMethodMetadata)TransactionSynchronizationManager.getResource(method);  
                if (metadata == null) {  
                    methodMetadata = (CrudMethodMetadata)this.metadataCache.get(method);  
                    Object tmp;  
                    if (methodMetadata == null) {  
                        methodMetadata = new DefaultCrudMethodMetadata(method);  
                        tmp = (CrudMethodMetadata)this.metadataCache.putIfAbsent(method, methodMetadata);  
                        if (tmp != null) {  
                            methodMetadata = tmp;  
                        }  
                    }  
                }  
  
                TransactionSynchronizationManager.bindResource(method, methodMetadata);  
  
                try {  
                    tmp = invocation.proceed();  
                    return tmp;  
                } finally {  
                    TransactionSynchronizationManager.unbindResource(method);  
                }  
            }  
  
            methodMetadata = invocation.proceed();  
        } finally {  
            currentInvocation.set(oldInvocation);  
        }  
  
        return methodMetadata;  
    }  
}
```

repository

CrudRepository [CORE]

```
package org.springframework.data.repository;  
public interface CrudRepository<T, ID> extends Repository<T, ID>  
    org.springframework.data.repository.ListCrudRepository
```

主要是进行增删改查的方法

```
org.springframework.data.repository.PagingAndSortingRepository  
org.springframework.data.jpa.repository.JpaRepository  
org.springframework.data.jpa.repository.JpaSpecificationExecutor
```

ListCrudRepository

```
package org.springframework.data.repository;  
public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID>  
<S extends T> List<S> saveAll(Iterable<S> entities);  
List<T> findAll();  
List<T> findAllById(Iterable<ID> ids);
```

PagingAndSortingRepository

```
package org.springframework.data.repository;  
public interface PagingAndSortingRepository<T, ID> extends Repository<T, ID>    主要是进行排序或者分页  
Iterable<T> findAll(Sort sort);  
Page<T> findAll(Pageable pageable);
```

expression

EvaluationContext

```
package org.springframework.expression;  
public interface EvaluationContext
```

To provide a **flexible and extensible environment** that allows SpEL expressions to be evaluated in a given context.

This context includes access to variables, bean resolvers, type converters, and other utilities required for evaluating expressions.

```
void setVariable(String name, @Nullable Object value);
```

Usage

```
import org.springframework.expression.ExpressionParser;  
import org.springframework.expression.spel.standard.SpelExpressionParser;  
import org.springframework.expression.spel.support.StandardEvaluationContext;  
  
public class EvaluationContextExample {  
    public static void main(String[] args) {  
        // Root object for expression evaluation  
        Person person = new Person("Alice", 25);  
  
        // Create an expression parser  
        ExpressionParser parser = new SpelExpressionParser();  
  
        // Create an evaluation context with the root object  
        StandardEvaluationContext context = new StandardEvaluationContext(person);  
  
        // Set a variable in the context  
        context.setVariable("increment", 5);  
  
        // Evaluate an expression that uses the root object and the variable  
        Integer newAge = parser.parseExpression("age + #increment").getValue(context, Integer.class);  
  
        System.out.println("New Age: " + newAge); // Output: New Age: 30  
  
        // Evaluate an expression that accesses the root object's properties  
        String greeting = parser.parseExpression("'Hello, ' + name").getValue(context, String.class);  
  
        System.out.println(greeting); // Output: Hello, Alice  
    }  
  
    // The Person class used as the root object
```

```

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

```

ExpressionParser

```

package org.springframework.expression;
public interface ExpressionParser

```

Expression

```

package org.springframework.expression;
public interface Expression
<T> T getValue(EvaluationContext context, @Nullable Class<T> desiredResultType) throws EvaluationException;

```

common

TemplateAwareExpressionParser

```

package org.springframework.expression.common;
public abstract class TemplateAwareExpressionParser implements ExpressionParser 模板解析器
public Expression parseExpression(String expressionString) throws ParseException 解析表达式，此处没有@Value 的特有语法
#{}, 直接使用 spel 即可
#obj.dd 获取上下文中的操作对象

```

spel

support

SimpleEvaluationContext

```

package org.springframework.expression.spel.support;
public final class SimpleEvaluationContext implements EvaluationContext 为不需要 SpEL 语言语法的完整范围的表达式类别公开一个基本 SpEL 语言功能和配置选项的子集，并且应该进行有意义的限制。

```

示例包括但不限于数据绑定表达式和基于属性的过滤器。

StandardEvaluationContext

```

package org.springframework.expression.spel.support;
public class StandardEvaluationContext implements EvaluationContext 公开全套 SpEL 语言功能和配置选项。
public StandardEvaluationContext(@Nullable Object rootObject) 根对象，spel 调用方法时会调用 rootObject 的方法
public void registerFunction(String name, Method method) 注册一个方法，注册的方法必须是一个 static 类型的公有方法
    Method plusTen = MathUtils.class.getDeclaredMethod("plusTen", int.class);
    StandardEvaluationContext context = new StandardEvaluationContext();
    context.registerFunction("plusTen", plusTen);

```

standard

SpelExpressionParser 【入口】

```

package org.springframework.expression.spel.standard;
public class SpelExpressionParser extends TemplateAwareExpressionParser spel 解析器

```

org.springframework.expression.EvaluationContext	评估上下文，存储所有操作对象
org.springframework.expression.StandardEvaluationContext	标准评估上下文
org.springframework.expression.common.TemplateAwareExpressionParser	模板解析器

InternalSpelExpressionParser

```
package org.springframework.expression.spel.standard;
class InternalSpelExpressionParser extends TemplateAwareExpressionParser
```

format
annotation

DateTimeFormat

```
package org.springframework.format.annotation;
package org.springframework.format.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER, ElementType.ANNOTATION_TYPE})
public @interface DateTimeFormat
```

Specify the format of the front-end input time type. Otherwise, an exception is thrown and the format will not be converted.

Formats date/time values for http request parameters, path variables, form inputs, and model attributes in a web environment.

@DateTimeFormat cannot be used directly for formatting or parsing fields in an @RequestBody.

This annotation is applied to long, java.time, java.util.Date, or java.util.Calendar fields to format and parse date/time values automatically, based on a specific pattern or style.

ISO iso() default ISO.NONE;

Uses predefined ISO-8601 date/time formats (e.g., ISO.DATE, ISO.TIME, ISO.DATE_TIME).

String pattern() default "";

A custom date/time pattern (e.g., "yyyy-MM-dd", "dd/MM/yyyy", "yyyy-MM-dd'T'HH:mm:ss").

String style() default "SS";

Uses a style-based formatting for dates or times (e.g., SS, MM), where letters stand for the short and medium format, respectively.

@NumberFormat 针对前端参数 反序列化数字 (一般使用在类成员, 前端参数上, 支持 long 类型标注)

pattern = "#,##.##" 转换参数为 double

hateoas

MediaTypes

```
package org.springframework.hateoas;
public class MediaTypes
public static final String HAL_JSON_VALUE = "application/hal+json";
public static final MediaType HAL_JSON = MediaType.valueOf(HAL_JSON_VALUE);
public static final String ALPS_JSON_VALUE = "application/alps+json";
public static final MediaType ALPS_JSON = MediaType.parseMediaType(ALPS_JSON_VALUE);
```

RepresentationModel

```
package org.springframework.hateoas;
public class RepresentationModel<T extends RepresentationModel<? extends T>>
```

org.springframework.http.ResponseEntity

```
public T add(Link link)    Adds the given link to the resource.  
    RepresentationModel resource = new RepresentationModel();  
    resource.add(  
        LinkTo(  
            methodOn(MarkupController.class)  
            .renderMarkup(MediaType.TEXT_MARKDOWN, "")  
        )  
        .withRel("markup"));  
    resource.add(LinkTo(methodOn(GuidesController.class).listGuides()).withRel("guides"));  
    server
```

LinkBuilder

```
package org.springframework.hateoas.server;  
public interface LinkBuilder  
default Link withRel(String rel)    Creates the Link built by the current builder instance with the given link relation.  
    mvc
```

WebMvcLinkBuilder

```
package org.springframework.hateoas.server.mvc;  
public class WebMvcLinkBuilder extends TemplateVariableAwareLinkBuilderSupport<WebMvcLinkBuilder>  
public static WebMvcLinkBuilder linkTo(Object invocationValue)  
public static <T> T methodOn(Class<T> controller, Object... parameters)    Wrapper for  
DummyInvocationUtils.methodOn(Class, Object...) to be available in case you work with static imports of WebMvcLinkBuilder.
```

http

HttpRequest

```
package org.springframework.http;  
public interface HttpRequest extends HttpMessage    RestTemplate 发出的请求，spring 提供的一个 http 请求工具，表明都是  
用来进行 http 请求用的
```

```
default HttpMethod getMethod()  
String getMethodValue();  
URI getURI();
```

HttpEntity

```
package org.springframework.http;  
public class HttpEntity<T>    请求实体  
public HttpEntity(@Nullable T body, @Nullable MultiValueMap<String, String> headers)    设置请求体和请求头
```

HttpHeaders

```
package org.springframework.http;  
public class HttpHeaders implements MultiValueMap<String, String>, Serializable
```

HttpStatus

```
package org.springframework.http;  
public enum HttpStatus implements HttpStatusCode    http 状态码  
public static enum Series {    状态码段 (除了 200 段, 其他全是异常状态码), 解析规则是 StatusCode/100 取整。
```

```
        INFORMATIONAL(1),  
        SUCCESSFUL(2),  
        REDIRECTION(3),  
        CLIENT_ERROR(4),  
        SERVER_ERROR(5);  
    }  
}
```

HttpStatus

```
package org.springframework.http;  
public sealed interface HttpStatus extends Serializable permits DefaultHttpStatus, HttpStatus
```

MediaType

```
package org.springframework.http;
```

```
public class MediaType extends MineType implements Serializable 文件媒体类型
```

```
public static final MediaType TEXT_MARKDOWN;  
public static final String TEXT_MARKDOWN_VALUE = "text/markdown";
```

ResponseEntity

```
package org.springframework.http;  
public class ResponseEntity<T> extends HttpEntity<T>  
    org.springframework.hateoas.RepresentationModel
```

codc

ServerCodecConfigurer

```
package org.springframework.http.codec;  
public interface ServerCodecConfigurer extends CodecConfigurer
```

```
@Override  
ServerDefaultCodecs defaultCodecs();
```

```
@Override  
ServerCodecConfigurer clone();
```

```
static ServerCodecConfigurer create()
```

client

ClientHttpRequest

```
package org.springframework.http.client;  
public interface ClientHttpRequest extends HttpRequest, HttpOutputMessage 代表请求的客户端 (Netty、HttpComponents、  
OkHttp3, HttpURLConnection 对它都有实现)
```

ClientHttpResponse **execute()** throws IOException

ClientHttpRequestFactory

package org.springframework.http.client;

public interface **ClientHttpRequestFactory** RestTemplate 组件，函数式接口，用于根据 URI 和 HttpMethod 创建出一个 ClientHttpRequest 来发送请求（我们可以使用 Apache 的 HttpClient、OkHttp3、Netty4 的实现都可以，但这些都需要额外导包，默认情况下 Spring 使用的是 java.net.HttpURLConnection）

ClientHttpRequest **createRequest(URI uri, HttpMethod httpMethod)** throws IOException

ClientHttpRequestInterceptor

package org.springframework.http.client;

public interface **ClientHttpRequestInterceptor**

对 RestTemplate 的请求进行拦截的，在项目中直接使用 restTemplate.getForObject 的时候，会对这种请求进行拦截，经常被称为：RestTempalte 拦截器或者 Ribbon 拦截器；

HandlerInterceptor 是最常规的，其拦截的 http 请求是来自于客户端浏览器之类的，是最常见的 http 请求拦截器；

ClientHttpRequestInterceptor 是对 RestTemplate 的请求进行拦截的，在项目中直接使用 restTemplate.getForObject 的时候，会对这种请求进行拦截，经常被称为：RestTempalte 拦截器或者 Ribbon 拦截器；

RequestInterceptor 常被称为是 Feign 拦截器，由于 Feign 调用底层实际上还是 http 调用，因此也是一个 http 拦截器，在项目中使用 Feign 调用的时候，可以使用此拦截器；

SimpleClientHttpRequestFactory

package org.springframework.http.client;

public class **SimpleClientHttpRequestFactory** implements ClientHttpRequestFactory, AsyncClientHttpRequestFactory
Spring 内置默认的实现，使用的是 JDK 内置的 java.net.URLConnection 作为 client 客户端（JDK 的版本低于 1.8 的话，那么 Delete 请求是不支持 body 体的）

converter

HttpMessageConverter

package org.springframework.http.converter;

public interface **HttpMessageConverter<T>** 可以使用 xml,json 或其它形式进行“封送”和“解收”（marshall and unmarshall）
java 对象

它是通过从请求的 Accept 头中获取信息，从 HttpMessageConverter 的列表中，取得相应的 Converter 对象，将 java 对象进行“封送 marshall”成为 xml 或 json。（前提是使用了@ResponseBody 或@RestController）

它又可以通过从请求中获取 Content-Type 头信息，从从 HttpMessageConverter 的列表中，取得相应的 Converter 对象，将请求体进行“解收 unmarshall”，从而转化成为 java 对象（前提是使用了@RequestBody 对参数进行了注解）

ByteArrayHttpMessageConverter

package org.springframework.http.converter;

public class **ByteArrayHttpMessageConverter** extends AbstractHttpMessageConverter<byte[]> 转换字节数组数据

StringHttpMessageConverter

package org.springframework.http.converter;

public class **StringHttpMessageConverter** extends AbstractHttpMessageConverter<String> 转换字符串数据

ResourceHttpMessageConverter

package org.springframework.http.converter;

public class **ResourceHttpMessageConverter** extends AbstractHttpMessageConverter<Resource> 转换任何 octet stream 为
org.springframework.core.io.Resource 类型

FormHttpMessageConverter

```
package org.springframework.http.converter;  
public class FormHttpMessageConverter implements HttpMessageConverter<MultiValueMap<String, ?>> 将表单数据转换  
为 MultiValueMap<String, String>
```

xml

SourceHttpMessageConverter

```
package org.springframework.http.converter.xml;  
public class SourceHttpMessageConverter<T extends Source> extends AbstractHttpMessageConverter<T> 转换  
javax.xml.transform.Source
```

Jaxb2RootElementHttpMessageConverter

```
package org.springframework.http.converter.xml;  
public class Jaxb2RootElementHttpMessageConverter extends AbstractJaxb2HttpMessageConverter<Object> 将 Java 对象  
转换为 XML (仅当类路径上存在 JAXB2 时才添加)
```

json

Jackson2ObjectMapperBuilder

```
package org.springframework.http.converter.json;  
A builder used to create ObjectMapper instances with a fluent API.  
It customizes Jackson's default properties with the following ones:  
MapperFeature.DEFAULT_VIEW_INCLUSION is disabled  
DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES is disabled
```

org.springframework.boot.autoconfigure.jackson.Jackson2ObjectMapperBuilderCustomizer

```
public Jackson2ObjectMapperBuilder filters(FilterProvider filters)  
public Jackson2ObjectMapperBuilder featuresToEnable(Object... featuresToEnable)  
public Jackson2ObjectMapperBuilder serializerByType(Class<?> type, JsonSerializer<?> serializer)
```

MappingJackson2HttpMessageConverter

```
package org.springframework.http.converter.json;  
public class MappingJackson2HttpMessageConverter extends AbstractJackson2HttpMessageConverter 转换 JSON (仅当  
类路径中存在 Jackson 2 时添加), restTemplate 使用的转换器, springboot 中默认的 Json 消息转换器  
feed
```

AtomFeedHttpMessageConverter

```
package org.springframework.http.converter.feed;  
public class AtomFeedHttpMessageConverter extends AbstractWireFeedHttpMessageConverter<Feed> 转换 Atom 提要  
(仅当类路径中存在 Rome 时才添加)
```

RssChannelHttpMessageConverter

```
package org.springframework.http.converter.feed;  
public class RssChannelHttpMessageConverter extends AbstractWireFeedHttpMessageConverter<Channel> 转换 RSS 提要  
(仅当类路径中存在 Rome 时才添加)
```

jdbc

CannotGetJdbcConnectionException

```
package org.springframework.jdbc;
public class CannotGetJdbcConnectionException extends DataAccessResourceFailureException 无法获取 jdbc 连接异常
```

core

JdbcTemplate

```
package org.springframework.jdbc.core;
public class JdbcTemplate extends JdbcAccessor implements JdbcOperations  jdbc 模板对象，通过获取 dataSource 自动执行
sql
```

JDBC string URL

```
jdbc:oracle:thin:@<host>:<port>:<sid>
jdbc:oracle:thin:@//<host>:<port>/service
```

```
public int update(final String sql) 执行更新语句 // string sql = "insert into
```

```
student(name, age)values (?, ?);      jdbcTemplate.update (sql, " 李四 ", 25);
```

```
public <T> T queryForObject(String sql, Class<T> requiredType, @Nullable Object... args) 单字段查询可以使用专用的查询
方法，必须指定查询出的数据类型
```

query

```
@Nullable
public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws DataAccessException {
    Assert.notNull(sql, "SQL must not be null");
    Assert.notNull(rse, "ResultSetExtractor must not be null");
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Executing SQL query [" + sql + "]");
    }
}
```

```
class QueryStatementCallback implements StatementCallback<T>, SqlProvider {
    QueryStatementCallback() {
    }
```

```
@Nullable
public T doInStatement(Statement stmt) throws SQLException {
    ResultSet rs = null;
```

```
    Object var3;
    try {
        rs = stmt.executeQuery(sql);
        var3 = rse.extractData(rs);
    } finally {
        JdbcUtils.closeResultSet(rs);
    }
}
```

```
    return var3;
}
```

```
    public String getSql() {
        return sql;
    }
}
```

```
    return this.execute(new QueryStatementCallback(), true); //Executes command object.
}
```

queryForStream

```
public <T> Stream<T> queryForStream(final String sql, final RowMapper<T> rowMapper) throws
DataAccessException {
```

```
    class StreamStatementCallback implements StatementCallback<Stream<T>>, SqlProvider {
        StreamStatementCallback() {
        }
    }
```

```
    public Stream<T> doInStatement(Statement stmt) throws SQLException {
```

```

        ResultSet rs = stmt.executeQuery(sql);
        Connection con = stmt.getConnection();
        return (Stream)(new ResultSetSpliterator(rs, rowMapper)).stream().onClose(() -> {
            JdbcUtils.closeResultSet(rs);
            JdbcUtils.closeStatement(stmt);
            DataSourceUtils.releaseConnection(con, JdbcTemplate.this.getDataSource());
        });
    }

    public String getSql() {
        return sql;
    }
}

return (Stream)result((Stream)this.execute(new StreamStatementCallback(), false));
}
execute
@Nullable
private <T> T execute(StatementCallback<T> action, boolean closeResources) throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");
    Connection con = DataSourceUtils.getConnection(this.obtainDataSource());
    Statement stmt = null;

    Object var12;
    try {
        stmt = con.createStatement();
        this.applyStatementSettings(stmt);
        T result = action.doInStatement(stmt);
        this.handleWarnings(stmt);
        var12 = result;
    } catch (SQLException var10) {
        String sql = getSql(action);
        JdbcUtils.closeStatement(stmt);
        stmt = null;
        DataSourceUtils.releaseConnection(con, this.getDataSource());
        con = null;
        throw this.translateException("StatementCallback", sql, var10);
    } finally {
        if (closeResources) {
            JdbcUtils.closeStatement(stmt);
            DataSourceUtils.releaseConnection(con, this.getDataSource());
        }
    }
    return var12;
}
getSql
@Nullable
private static String getSql(Object obj) {
    String var10000;
    if (obj instanceof SqlProvider sqlProvider) {
        var10000 = sqlProvider.getSql();
    } else {
        var10000 = null;
    }
    return var10000;
}

```

注册 JdbcTemplate 模块对象 bean

```

@Bean ("jdbcTemplate")
public JdbcTemplate getJdbcTemplate(@Autowired DataSource dataSource){
    return new JdbcPemplate(datasource);
}

```

```

    }

    @Bean ( "jdbcTemplate2")
    public NamedParameterJdbcTemplate getJdbcTemplate2(@Autowired DataSource datasource){
        return new NamedParameterJdbcTemplate (dataSource);
    }

public List<Account> findAll(){
    string sql = "select *from account" ;
    return jdbcTemplate.query(sql,new BeanPropertyRowMapper<Account>(Account.class)          //使用 spring 自带的行映射解析器, 要求必须是标准封装
}
}

public List<Account> findAll(int pageNum, int preNum) {
    string sql = "select * from account limit ?,?"; 
    return jdbcTemplate.query(sql, new BeanPropertyRowMapper<Account>(Account.class)          //分页数据通过查询参数赋值 T
}

public Long getCount (){ 
    string sql = "select count(id)from account ";
    return jdbcTemplate.queryForObject (sql,Long.class);           //单字段查询可以使用专用的查询方法, 必须指定查询出的数据类型, 例如数据总量为 Long 类型
}
}

```

StatementCallback

```

package org.springframework.jdbc.core;
public interface StatementCallback<T>

```

BeanPropertyRowMapper

```

package org.springframework.jdbc.core;
public class BeanPropertyRowMapper<T>      行映射解析器

```

NamedParameterJdbcTemplate

```

package org.springframework.jdbc.core.namedparam;
public class NamedParameterJdbcTemplate      命名参数 sql 语句

```

使用

```

public void save (Account account) {
    string sql = "insert into account (name ,money)values ( :name , :money)";
    Map pm = new HashMap();
    pm.put ( "name" , account.getName() );
    pm.put ( "money" , account.getMoney() );
    jdbcTemplate.update(sql , pm);
}

```

datasource

DataSourceTransactionManager

```

package org.springframework.jdbc.datasource;
public class DataSourceTransactionManager      extends AbstractPlatformTransactionManager      implements
ResourceTransactionManager, InitializingBean
public void setDataSource(@Nullable DataSource dataSource)      为事务管理器 设置 与数据层相同的数据源、

```

DataSourceUtils

```
package org.springframework.jdbc.datasource;
public abstract class DataSourceUtils 数据源工具
public static Connection getConnection(DataSource dataSource) throws CannotGetJdbcConnectionException    获取连接
```

AbstractRoutingDataSource

```
package org.springframework.jdbc.datasource.lookup;
public abstract class AbstractRoutingDataSource extends AbstractDataSource implements InitializingBean
```

The `AbstractRoutingDataSource` class in Spring Framework is used to facilitate dynamic routing of database connections in applications.

It extends the `AbstractDataSource` class and acts as a routing mechanism that determines which data source to use at runtime based on a lookup key.

`@Nullable`

```
protected abstract Object determineCurrentLookupKey();
```

Determine the current lookup key. This will typically be implemented to check a thread-bound transaction context.

lang

[annotation]

```
package org.springframework.lang;
```

@NotNullApi Annotation at the package level that declares non-null as the default semantics for parameters and return values.

@NotNullFields Annotation at the package level that declares non-null as the default semantics for fields.

@NotNull Annotation to indicate that a specific parameter, return value, or field cannot be null not needed on parameters / return values and fields where `@NotNullApi` and `@NotNullFields` apply, respectively 【METHOD, PARAMETER, FIELD】

@Nullable Annotation to indicate that a specific parameter, return value, or field can be null. 【METHOD, PARAMETER, FIELD】

mail

javamail

JavaMailSender [CORE]

```
package org.springframework.mail.javamail;
public interface JavaMailSender extends MailSender    邮件发送器
    jakarta.mail.internet.MimeMessage
    org.springframework.mail.javamail. MimeMessageHelper
```

`MimeMessage createMimeMessage();` 创建 mime 格式消息

`MimeMessage createMimeMessage(InputStream contentStream)` throws `MailException`;

`void send(MimeMessage mimeMessage)` throws `MailException`; 发送 mime 格式消息

`void send(MimeMessage... mimeMessages)` throws `MailException`;

`void send(MimeMessagePreparator mimeMessagePreparator)` throws `MailException`;

`void send(MimeMessagePreparator... mimeMessagePreparators)` throws `MailException`;

JavaMailSenderImpl

```
package org.springframework.mail.javamail;
public class JavaMailSenderImpl implements JavaMailSender 邮件发送器实现
public void setUsername(@Nullable String username)      设置用户名/登录邮箱
public void setPassword(@Nullable String password)      设置密码/授权码
public void setHost(@Nullable String host)      设置发送邮件的主机地址 // test.nsroot.net
public void setPort(int port)      设置发送邮件的主机端口
public void setJavaMailProperties(Properties javaMailProperties)  设置邮件发送属性
public void setDefaultEncoding(@Nullable String defaultEncoding)  设置默认编码, 可选 //UTF-8
    mail.transport.protocol=smtp
    mail.smtp.auth=false
    mail.smtp.starttls.enable=false
    mail.debug=false
    mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory 可选
```

MimeMessageHelper

```
package org.springframework.mail.javamail;
public class MimeMessageHelper mime 格式消息 帮助器

public static final int MULTIPART_MODE_NO = 0;
public static final int MULTIPART_MODE_MIXED = 1;
public static final int MULTIPART_MODE RELATED = 2;
public static final int MULTIPART_MODE_MIXED RELATED = 3;

public MimeMessageHelper(MimeMessage mimeMessage, int multipartMode) throws MessagingException
public MimeMessageHelper(
    MimeMessage mimeMessage,
    int multipartMode,
    @Nullable String encoding      编码 // StandardCharsets.UTF_8.name()
) throws MessagingException

public void setFrom(
    String from,      发送邮箱 // noreply@163.com
    String personal   发送人名称 // XXX Sender Name
) throws MessagingException, UnsupportedEncodingException  设置 发送邮件地址
public void setTo(String[] to) throws MessagingException  设置 接收邮件地址
public void setCc(String[] cc) throws MessagingException  设置抄送人
public void setText(String text, boolean html) throws MessagingException  设置邮件内容, 可选为 html 格式
public void setSubject(String subject) throws MessagingException      设置邮件标题
public void addAttachment(
    String attachmentFilename,      附件名称
    InputStreamSource inputStreamSource,  输入文件流
    String contentType            附件类型
) throws MessagingException
public void addAttachment(
    String attachmentFilename,
    DataSource dataSource        通过数据资源载入文件
) throws MessagingException
```

AbstractEntityManagerFactoryBean [CORE]

```
package org.springframework.orm.jpa;
public abstract class AbstractEntityManagerFactoryBean
    implements FactoryBean<EntityManagerFactory>, BeanClassLoaderAware, BeanFactoryAware,
    BeanNameAware, InitializingBean, DisposableBean,
    EntityManagerFactoryInfo, PersistenceExceptionTranslator, Serializable
```

There are three ways to set up an **EntityManager** factory, which provides me with an EntityManager as a bridge between the OO model and the DB relational model

- LocalEntityManagerFactoryBean
- LocalContainerEntityManagerFactoryBean
- Find using JNDI

```
getObject
@NoArgsConstructor
public EntityManagerFactory getObject() {
    return this.entityManagerFactory;
}
```

```
setJpaVendorAdapter
public void setJpaVendorAdapter(@Nullable JpaVendorAdapter jpaVendorAdapter) {
    this.jpaVendorAdapter = jpaVendorAdapter;
}
```

USE

```
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean; //导入依赖的 package 包/类
@Bean
public EntityManagerFactory entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
    emf.setDataSource(dataSource());
    emf.setPackagesToScan("io.github.azanx.shopping_list.domain");
    emf.setPersistenceUnitName("spring-jpa-unit");
    emf.setJpaVendorAdapter(getHibernateAdapter());
    Properties jpaProperties = new Properties();
    jpaProperties.put("hibernate.dialect", env.getProperty("hibernate.dialect"));
    jpaProperties.put("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));
    jpaProperties.put("hibernate.show_sql", env.getProperty("hibernate.show_sql"));
    jpaProperties.put("hibernate.format_sql", env.getProperty("hibernate.format_sql"));
    emf.setJpaProperties(jpaProperties);
    emf.afterPropertiesSet();
    return emf.getObject();
}
```

LocalContainerEntityManagerFactoryBean

```
package org.springframework.orm.jpa;
public class LocalContainerEntityManagerFactoryBean extends AbstractEntityManagerFactoryBean implements
ResourceLoaderAware, LoadTimeWeaverAware
```

Applicable to projects that only use JPA for data access.

The FactoryBean works based on the JPA PersistenceProvider automatic detection configuration file, typically reading configuration information from "META-INF/persistence.xml".

The method is the simplest, but it cannot set the DataSource defined in Spring and does not support global

transactions managed by Spring.
This method is not recommended.
design for jpa database access method, This FactoryBean works
based on JPA PersistenceProvider to detect configuration files from the file "META-INF/persistence.xml"
This approach is easy, but not support spring global transaction manager and can't set Datasource

LocalEntityManagerFactoryBean

```
package org.springframework.orm.jpa;  
public class LocalEntityManagerFactoryBean extends AbstractEntityManagerFactoryBean  
    vendor
```

HibernateJpaVendorAdapter

```
package org.springframework.orm.jpa.vendor;  
public class HibernateJpaVendorAdapter extends AbstractJpaVendorAdapter  
    querydsl
```

QuerydslPredicateExecutor

```
package org.springframework.data.querydsl;  
public interface QuerydslPredicateExecutor<T>  
  
Optional<T> findOne(Predicate predicate);  
Iterable<T> findAll(Predicate predicate);  
Iterable<T> findAll(Predicate predicate, Sort sort);  
Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);  
Iterable<T> findAll(OrderSpecifier<?>... orders);  
Page<T> findAll(Predicate predicate, Pageable pageable);  
long count(Predicate predicate);  
boolean exists(Predicate predicate);  
<S extends T, R> R findBy(Predicate predicate, Function<FluentQuery.FetchableFluentQuery<S>, R> queryFunction);  
    scheduling
```

TaskScheduler

```
package org.springframework.scheduling;  
public interface TaskScheduler 任务调度  
    commonj
```

WorkManagerTaskExecutor

```
package org.springframework.scheduling.commonj;  
@Deprecated  
public class WorkManagerTaskExecutor extends JndiLocatorSupport implements AsyncListenableTaskExecutor,  
SchedulingTaskExecutor, WorkManager, InitializingBean  
这个实现使用CommonJ WorkManager作为它的支持服务提供者，它是在Spring应用程序上下文中设置基于CommonJ  
的WebLogic/WebSphere线程池集成的中心便利类。
```

concurrent

DefaultManagedTaskExecutor

```
package org.springframework.scheduling.concurrent;  
public class DefaultManagedTaskExecutor extends ConcurrentTaskExecutor implements InitializingBean 此实现在JSR-236兼容的运行时环境(如Java EE 7+应用服务器)中使用jndi获得的ManagedExecutorService,以取代CommonJ WorkManager.
```

ThreadPoolTaskExecutor

```
package org.springframework.scheduling.concurrent;
```

```
public class ThreadPoolTaskExecutor extends ExecutorConfigurationSupport implements AsyncListenableTaskExecutor, SchedulingTaskExecutor
```

ThreadPoolTaskExecutor is a Spring class used to manage a pool of threads for executing tasks asynchronously.

It's a powerful way to handle concurrent tasks in a Spring application.

See: [java.util.concurrent.ThreadPoolExecutor](#)

TaskRejectedException:

Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception

[Request processing failed: org.springframework.core.task.TaskRejectedException:

Executor [

java.util.concurrent.ThreadPoolExecutor@804443e[Running, pool size = 2, active threads = 2, queued tasks = 10, completed tasks = 0]

] did not accept task: java.util.concurrent.CompletableFuture\$AsyncSupply@7264ef72] with root cause

```
public void setCorePoolSize(int corePoolSize)
```

```
public void setMaxPoolSize(int maxPoolSize)
```

```
public void setThreadNamePrefix(@Nullable String threadNamePrefix)
```

```
public void setWaitForTasksToCompleteOnShutdown(boolean waitForJobsToCompleteOnShutdown)
```

Control how the executor behaves during shutdown.

By default, this is set to false, meaning the executor will not wait for tasks to complete before shutting down.

```
public void setAwaitTerminationSeconds(int awaitTerminationSeconds)
```

Specifies the amount of time, in seconds, that the executor will wait for tasks **to complete after the shutdown process** has been initiated.

This method works in conjunction with setWaitForTasksToCompleteOnShutdown to control the shutdown behavior of the thread pool.

```
public void setTaskDecorator(TaskDecorator taskDecorator)
```

Allows you to customize the tasks that are executed by the thread pool.

By using a TaskDecorator, you can modify or wrap tasks before they are executed by the threads in the pool.

This can be useful for adding additional behavior, such as logging, setting context information, or modifying the task execution environment.

The request context and related data (like security context, request attributes, etc.) **are not automatically propagated to asynchronous threads**.

Solutions and Techniques for Context Propagation

Manually Pass Context Information:

If you need specific information (like user credentials or request attributes) in the asynchronous thread, you can manually pass it as part of the task execution.

This means you would need to capture and pass the necessary data explicitly.

Use @Async with AsyncContext:

In Spring, if you use @Async to run methods asynchronously, you can manually set and propagate context information.

For instance, you can use a custom TaskDecorator to manage the context propagation.

Override

```
public void setRejectedExecutionHandler(@Nullable RejectedExecutionHandler rejectedExecutionHandler)
```

Set

a

custom rejection policy

```

createQueue
protected BlockingQueue<Runnable> createQueue(int queueCapacity) {
    if (queueCapacity > 0) {
        return new LinkedBlockingQueue<>(queueCapacity);
    }
    else {
        return new SynchronousQueue<>();
    }
}

```

Configuration

```

spring.task.execution.pool.core-size=5
spring.task.execution.pool.max-size=10
spring.task.execution.pool.queue-capacity=5
spring.task.execution.pool.keep-alive=60
spring.task.execution.thread-name-prefix=god-jiang-task-

```

Custom TaskDecorator

```

import org.springframework.scheduling.config.TaskDecorator;
import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;

public class SecurityContextTaskDecorator implements TaskDecorator {

    @Override
    public Runnable decorate(Runnable runnable) {
        SecurityContext context = SecurityContextHolder.getContext();
        return () -> {
            SecurityContextHolder.setContext(context);
            try {
                runnable.run();
            } finally {
                SecurityContextHolder.clearContext();
            }
        };
    }
}

```

Example with CompletableFuture:

```

import java.util.concurrent.CompletableFuture;

public class AsyncService {

    public CompletableFuture<Void> asyncMethod() {
        return CompletableFuture.runAsync(() -> {
            // Access and use context information here
        });
    }
}

```

ConcurrentTaskExecutor

```

package org.springframework.scheduling.concurrent;
public class ConcurrentTaskExecutor implements AsyncListenableTaskExecutor, SchedulingTaskExecutor

```

这个实现是 `java.util.concurrent.Executor` 的适配器实例。

还有一种替代方法，`ThreadPoolTaskExecutor`，它将 `Executor` 配置参数公开为 bean 属性。

很少需要直接使用 `ConcurrentTaskExecutor`，但是如果 `ThreadPoolTaskExecutor` 不够灵活，不能满足您的需求，那么 `ConcurrentTaskExecutor` 是另一种选择。

annotation

AsyncAnnotationBeanPostProcessor

```

package org.springframework.scheduling.annotation;

```

```
public class AsyncAnnotationBeanPostProcessor extends AbstractBeanFactoryAwareAdvisingPostProcessor 处理@Async  
注解
```

AsyncAnnotationBeanPostProcessor 这个类的对象是由@EnableAsync 注解放入到 Spring 容器的父类 AbstractAdvisingBeanPostProcessor 实现了 BeanPostProcessor 的 postProcessAfterInitialization 方法。当 Bean 的初始化阶段完成之后会回调 AsyncAnnotationBeanPostProcessor 的 postProcessAfterInitialization 方法。

AOP 和@Async 注解虽然底层都是动态代理，但是具体实现的类是不一样的。

一般的 AOP 或者事务的动态代理是依靠 AnnotationAwareAspectJAutoProxyCreator 实现的，@Async 是依靠 AsyncAnnotationBeanPostProcessor 实现的，并且都是在初始化完成之后起作用，这也就是@Async 注解和 AOP 之间的主要区别，也就是处理的类不一样。

(annotation)

Async

```
package org.springframework.scheduling.annotation;  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Reflective  
public @interface Async
```

String value() default "";

Specifies the name of the Executor bean to be used for executing the annotated method asynchronously. By default, if no value is specified, Spring uses a default executor.

By default, Spring uses a org.springframework.core.task.SimpleAsyncTaskExecutor to actually run these methods asynchronously.

Defining a Custom Executor:

```
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.scheduling.annotation.EnableAsync;  
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
  
@Configuration  
@EnableAsync  
public class AsyncConfig {  
  
    @Bean(name = "customExecutor")  
    public ThreadPoolTaskExecutor taskExecutor() {  
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
        executor.setCorePoolSize(5);  
        executor.setMaxPoolSize(10);  
        executor.setQueueCapacity(25);  
        executor.initialize();  
        return executor;  
    }  
}
```

EnableAsync

```
package org.springframework.scheduling.annotation;  
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Import({AsyncConfigurationSelector.class})
```

public @interface **EnableAsync**

By default, Spring uses a [org.springframework.core.task.SimpleAsyncTaskExecutor](#) to actually run these methods asynchronously.

But we can override the defaults at two levels: the application level or the individual method level.

```
Class<? extends Annotation> annotation() default Annotation.class;
```

By default, `@EnableAsync` detects Spring's `@Async` annotation and the EJB 3.1 `javax.ejb.Asynchronous`.

We can use this option to detect other, user-defined annotation types as well.

```
boolean proxyTargetClass() default false;
```

Indicates the type of proxy that should be used — CGLIB or JDK. This attribute has effect only if the mode is set to `AdviceMode.PROXY`.

```
AdviceMode mode() default AdviceMode.PROXY;
```

```
int order() default Integer.MAX_VALUE;
```

Sets the order in which `AsyncAnnotationBeanPostProcessor` should be applied. By default, it runs last so that it can take into account all existing proxies.

EnableScheduling

```
package org.springframework.scheduling.annotation;
```

```
@Target(ElementType.TYPE)
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Import(SchedulingConfiguration.class)
```

```
@Documented
```

```
public @interface EnableScheduling
```

You can add the `@EnableScheduling` annotation to any configuration class or component class, not just the startup (main) application class.

It is less common to add it to component class because component classes are typically focused on business logic or service functionality.

```
org.springframework.scheduling.annotation.Scheduled
```

Scheduled

```
package org.springframework.scheduling.annotation;
```

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
```

```
@Repeatable(Schedules.class)
```

```
@Reflective
```

```
public @interface Scheduled
```

Enable scheduling tasks within your application.

It allows you to define a method that will run at fixed intervals or according to a specified cron expression, without needing to manually manage the scheduling.

```
org.springframework.scheduling.annotation.EnableScheduling
```

```
String CRON_DISABLED = ScheduledTaskRegistrar.CRON_DISABLED;
```

```
String cron() default "";
```

A cron expression for more complex scheduling needs. The expression consists of six fields:

Seconds, Minutes, Hours, Day of month, Month, Day of week

```
String zone() default "";
long fixedRate() default -1;
```

Specifies the interval in milliseconds between method executions. The method is executed repeatedly at this interval.

Overlapping Executions:

If the execution time of the task exceeds the defined rate, the next execution will start even if the previous one is still running.

This can lead to multiple instances of the task executing simultaneously.

```
long fixedDelay() default -1;
```

Specifies the interval in milliseconds after the last method execution completes before the next execution starts.

Overlapping Executions:

The next execution will only start after the current execution has finished and the specified delay has elapsed.

This ensures that no two executions overlap.

security

access

AccessDeniedException (异常)

```
package org.springframework.security.access;
public class AccessDeniedException extends RuntimeException 访问异常
```

config

FilterOrderRegistration

```
package org.springframework.security.config.annotation.web.builders;
final class FilterOrderRegistration
FilterOrderRegistration();
```

HttpSecurity

```
package org.springframework.security.config.annotation.web.builders;
public final class HttpSecurity extends AbstractConfiguredSecurityBuilder<DefaultSecurityFilterChain, HttpSecurity>
    implements SecurityBuilder<DefaultSecurityFilterChain>, HttpSecurityBuilder<HttpSecurity> {

    public CsrfConfigurer<HttpSecurity> csrf() throws Exception          Add CsrfFilter
    public FormLoginConfigurer<HttpSecurity> formLogin() throws Exception     Add UsernamePasswordAuthenticationFilter
    public HttpBasicConfigurer<HttpSecurity> httpBasic() throws Exception      Add BasicAuthenticationFilter
    public AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry
        authorizeHttpRequests() throws Exception   Add AuthorizationFilter

    public HttpSecurity addFilterBefore(Filter filter, Class<? extends Filter> beforeFilter)
        //http.addFilterBefore(new TenantFilter(), AuthorizationFilter.class);
    public HttpSecurity addFilterAfter(Filter filter, Class<? extends Filter> afterFilter)
    public HttpSecurity addFilterAt(Filter filter, Class<? extends Filter> atFilter)
```

core

context

SecurityContextHolder [CORE]

org.springframework.security.core.context

public class SecurityContextHolder 单机系统：整个生命周期只有一个用户在使用。由于整个应用只需要保存一个 SecurityContext
(安全上下文即可)

多用户系统：整个生命周期可能同时有多个用户在使用。这时候应用需要保存多个
SecurityContext (安全上下文)

需要利用 ThreadLocal 进行保存，每个线程都可以利用 ThreadLocal 获取其自己的 SecurityContext，及安全上下文

Spring 登录控制流程：用户登录；根据用户 ID，获取当前用户所拥有的所有权限；把权限放到 session 中；显示用户所拥有的资源。

```
public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";
public static final String MODE_INHERITABLETHREADLOCAL = "MODE_INHERITABLETHREADLOCAL";
public static final String MODE_GLOBAL = "MODE_GLOBAL";
private static final String MODE_PRE_INITIALIZED = "MODE_PRE_INITIALIZED";
public static final String SYSTEM_PROPERTY = "spring.security.strategy";
private static String strategyName = System.getProperty("spring.security.strategy");
private static SecurityContextHolderStrategy strategy;
private static int initializeCount = 0;

public static void setContext(SecurityContext context)      设置当前上下文
public static SecurityContext getContext()     获取安全上下文
    // SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    // Authentication.getPrincipal()可以获取到代表当前用户的信息 (Context 不存在则创建一个空的返回)
```

SecurityContextHolderStrategy

```
package org.springframework.security.core.context;
public interface SecurityContextHolderStrategy   安全上下文存储策略
void clearContext();
SecurityContext getContext();
void setContext(SecurityContext context);
SecurityContext createEmptyContext();
```

getDeferredContext

```
default Supplier<SecurityContext> getDeferredContext() {
    return () -> {
        return this.getContext();
    };
}
```

setDeferredContext

```
default void setDeferredContext(Supplier<SecurityContext> deferredContext) {
    this.setContext((SecurityContext)deferredContext.get());
}
```

ThreadLocalSecurityContextHolderStrategy

```
package org.springframework.security.core.context;
final class ThreadLocalSecurityContextHolderStrategy implements SecurityContextHolderStrategy
private static final ThreadLocal<Supplier<SecurityContext>> contextHolder = new ThreadLocal();
```

setContext

```
public void setContext(SecurityContext context) {      // Directly set the value of context.
    Assert.notNull(context, "Only non-null SecurityContext instances are permitted");
    contextHolder.set(() -> {
        return context;
    });
}
```

```

getContext
public SecurityContext getContext() {
    return (SecurityContext)this.getDeferredContext().get();
}

setDeferredContext
public void setDeferredContext(Supplier<SecurityContext> deferredContext) { // Set a delay value.
    Assert.notNull(deferredContext, "Only non-null Supplier instances are permitted");
    Supplier<SecurityContext> notNullDeferredContext = () -> {
        SecurityContext result = (SecurityContext)deferredContext.get();
        Assert.notNull(result, "A Supplier<SecurityContext> returned null and is not allowed.");
        return result;
    };
    contextHolder.set(notNullDeferredContext);
}

getDeferredContext
public Supplier<SecurityContext> getDeferredContext() {
    Supplier<SecurityContext> result = (Supplier)contextHolder.get();
    if (result == null) {
        SecurityContext context = this.createEmptyContext();
        result = () -> {
            return context;
        };
        contextHolder.set(result);
    }
    return result;
}

```

```

createEmptyContext
public SecurityContext createEmptyContext() {
    return new SecurityContextImpl();
}

```

```

clearContext
public void clearContext() {
    contextHolder.remove();
}

```

GlobalSecurityContextHolderStrategy

```

package org.springframework.security.core.context;
final class GlobalSecurityContextHolderStrategy implements SecurityContextHolderStrategy
private static SecurityContext contextHolder;

```

```

setContext
public void setContext(SecurityContext context) { // Directly set the value of context
    Assert.notNull(context, "Only non-null SecurityContext instances are permitted");
    contextHolder = context;
}

```

```

getContext

```

```
public SecurityContext getContext() {
    if (contextHolder == null) {
        contextHolder = new SecurityContextImpl();
    }

    return contextHolder;
}
```

createEmptyContext

```
public SecurityContext createEmptyContext() {
    return new SecurityContextImpl();
}
```

clearContext

```
public void clearContext() {
    contextHolder = null;
}
```

ListeningSecurityContextHolderStrategy

```
package org.springframework.security.core.context;
```

```
public final class ListeningSecurityContextHolderStrategy implements SecurityContextHolderStrategy
```

setContext

```
public void setContext(SecurityContext context) {
    this.setDeferredContext(() -> {
        return context;
    });
}
```

getContext

```
public SecurityContext getContext() {
    return this.delegate.getContext();
}
```

setDeferredContext

```
public void setDeferredContext(Supplier<SecurityContext> deferredContext) {
    this.delegate.setDeferredContext(new PublishOnceSupplier(this.getDeferredContext(), deferredContext));
}
```

getDeferredContext

```
public Supplier<SecurityContext> getDeferredContext() {
    return this.delegate.getDeferredContext();
}
```

createEmptyContext

```
public SecurityContext createEmptyContext() {
    return this.delegate.createEmptyContext();
}
```

clearContext

```
public void clearContext() {
    Supplier<SecurityContext> deferred = this.delegate.getDeferredContext();
    this.delegate.clearContext();
    this.publish(new SecurityContextChangedEvent(deferred, SecurityContextChangedEvent.NO_CONTEXT));
}
```

```

PublishOnceSupplier
class PublishOnceSupplier implements Supplier<SecurityContext> {
    private final AtomicBoolean isPublished = new AtomicBoolean(false);
    private final Supplier<SecurityContext> old;
    private final Supplier<SecurityContext> updated;

    PublishOnceSupplier(Supplier<SecurityContext> old, Supplier<SecurityContext> updated) {
        if (old instanceof PublishOnceSupplier) {
            this.old = ((PublishOnceSupplier)old).updated; // Distinguish between the current supplier and
the regular supplier,                                         // as the true value of the current supplier
exists within the field.
        } else {
            this.old = old;
        }

        this.updated = updated;
    }

    public SecurityContext get() {
        SecurityContext updated = (SecurityContext)this.updated.get();
        if (this.isPublished.compareAndSet(false, true)) {           // Only detected during the first get.
            SecurityContext old = (SecurityContext)this.old.get();
            if (old != updated) {
                ListeningSecurityContextHolderStrategy.this.publish(new SecurityContextChangedEvent(old,
updated));
            }
        }
        return updated;
    }
}

publish
private void publish(SecurityContextChangedEvent event) {
    Iterator var2 = this.listeners.iterator();

    while(var2.hasNext()) {
        SecurityContextChangedListener listener = (SecurityContextChangedListener)var2.next();
        listener.securityContextChanged(event);
    }
}

```

authentication

UsernamePasswordAuthenticationToken

```

package org.springframework.security.authentication;
public class UsernamePasswordAuthenticationToken 检查输入的用户名和密码，并根据认证结果决定是否将这一结果传递给下
一个过滤器

```

InMemoryUserDetailsManager

```

package org.springframework.security.provisioning;
public class InMemoryUserDetailsManager implements UserDetailsService, UserDetailsPasswordService

```

web

FilterChainProxy

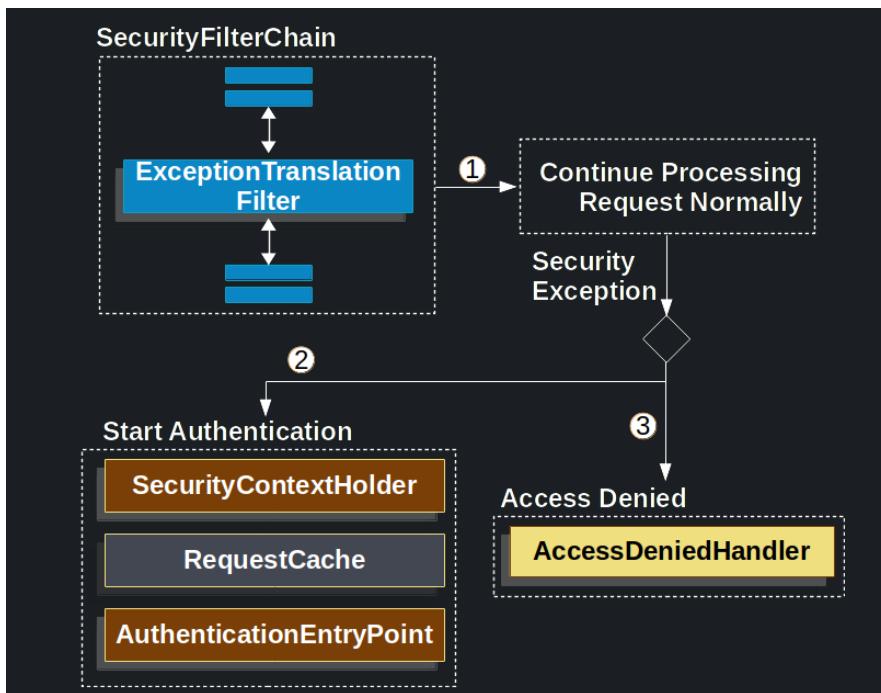
```
package org.springframework.security.web;  
public class FilterChainProxy extends GenericFilterBean  
  
private List<SecurityFilterChain> filterChains;  
public FilterChainProxy(List<SecurityFilterChain> filterChains)  
  
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,  
ServletException  
private List<Filter> getFilters(HttpServletRequest request)
```

SecurityFilterChain

```
package org.springframework.security.web;  
public interface SecurityFilterChain  
  
boolean matches(HttpServletRequest request);  
List<Filter> getFilters();
```

ExceptionTranslationFilter

```
package org.springframework.security.web.access;  
public class ExceptionTranslationFilter extends GenericFilterBean implements MessageSourceAware  
  
If the application does not throw an AccessDeniedException or an AuthenticationException, then  
ExceptionTranslationFilter does not do anything.
```



```
public ExceptionTranslationFilter(AuthenticationEntryPoint authenticationEntryPoint)
```

authentication

AuthenticationSuccessHandler

```
package org.springframework.security.web.authentication;
```

```
public interface AuthenticationSuccessHandler
```

```
void onAuthenticationSuccess(HttpServletRequest var1, HttpServletResponse var2, Authentication var3)
```

BasicAuthenticationConverter

```
package org.springframework.security.web.authentication.www;  
public class BasicAuthenticationConverter implements AuthenticationConverter  
public UsernamePasswordAuthenticationToken convert(HttpServletRequest request)
```

BasicAuthenticationFilter

```
package org.springframework.security.web.authentication.www;  
public class BasicAuthenticationFilter extends OncePerRequestFilter  
private BasicAuthenticationConverter authenticationConverter = new BasicAuthenticationConverter();  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws  
IOException, ServletException
```

util

AnyRequestMatcher (匹配)

```
package org.springframework.security.web.util.matcher;  
public final class AnyRequestMatcher implements RequestMatcher
```

AntPathRequestMatcher (匹配)

```
package org.springframework.security.web.util.matcher;  
public final class AntPathRequestMatcher implements RequestMatcher, RequestVariablesExtractor  
  
public AntPathRequestMatcher(String pattern, String httpMethod) // new AntPathRequestMatcher("/user/login",  
HttpMethod.POST.name())
```

MvcRequestMatcher (匹配)

```
package org.springframework.security.web.util.matcher;  
public class MvcRequestMatcher implements RequestMatcher, RequestVariablesExtractor
```

RequestMatcher (匹配)

```
package org.springframework.security.web.util.matcher;  
public interface RequestMatcher 请求匹配规则, 可配合过滤器, 过滤掉不需要 token 验证的 url  
boolean matches(HttpServletRequest var1)
```

RegexRequestMatcher (匹配)

```
package org.springframework.security.web.util.matcher;  
public final class RegexRequestMatcher implements RequestMatcher
```

oauth2 (spring-security-oauth2-jose)

NimbusJwtDecoder

```
package org.springframework.security.oauth2.jwt;  
public final class NimbusJwtDecoder implements JwtDecoder  
public static JwkSetUriJwtDecoderBuilder withJwkSetUri(String jwkSetUri)
```

JwtDecoder

```
package org.springframework.security.oauth2.jwt;  
@FunctionalInterface  
public interface JwtDecoder  
Jwt decode(String token) throws JwtException;
```

statemachine

State

```
public interface State<S, E>  
S getId();
```

The type of the state identifier. This is typically an enum or a String that uniquely represents the state within the state machine.

```
// OrderStates statId = state.getId();
```

StateMachineListenerAdapter

```
package org.springframework.statemachine.listener;  
public class StateMachineListenerAdapter<S, E> implements StateMachineListener<S, E>  
public void stateMachineStarted(StateMachine<S, E> stateMachine)
```

This method is called when the state machine is started.

```
public void stateMachineStopped(StateMachine<S, E> stateMachine)
```

This method is invoked when the state machine is stopped.

```
public void stateChanged(State<S, E> from, State<S, E> to)
```

This method is triggered whenever the state machine changes from one state to another.

```
public void transition(Transition<S, E> transition)
```

This method is called when a transition occurs between two states, triggered by an event.

stereotype

注解

```
package org.springframework.stereotype;  
@Component 注册一个 bean 到 IOC 容器中，注入类的构造可以通过参数接收其他 bean (@Controller, @Service, @Repository  
是@Component 衍生注解，功能同@Component) 【TYPE】  
@Controller  
@Service  
@Repository
```

test

web

MockMvc

```
package org.springframework.test.web.servlet;  
public final class MockMvc 模拟执行请求  
  
public ResultActions perform(RequestBuilder requestBuilder) 执行一个请求
```

```
package org.springframework.test.web.servlet;  
public interface ResultActions
```

```

ResultActions andExpect(ResultMatcher var1) 判断结果
    // andExpect( MockMvcResultMatchers.status().is(200) ) 状态码是否符合预期
    // andExpect( MockMvcResultMatchers.content().string("success")); 返回值是否符合预期
    // andExpect( MockMvcResultMatchers.model().attributeExists("user")) 验证存储模型数据
    // andExpect( MockMvcResultMatchers.view().name("user/view")) 验证 viewName
    // andExpect( MockMvcResultMatchers.forwardedUrl("/WEB-INF/jsp/user/view.jsp")) 验证视图渲染时 forward
到的 jsp
    // andExpect( MockMvcResultMatchers.status().isOk()) 验证状态码
    // andExpect(jsonPath("code").value("000")) 取出结果 json 中的一个键

```

```

ResultActions andDo(ResultHandler var1)
    .andDo( MockMvcResultHandlers.print() ) 添加一个结果处理器，输出 MvcResult 到控制台
MvcResult andReturn() 返回结果

```

MvcResult

```

package org.springframework.test.web.servlet;
public interface MvcResult

```

```
MockHttpServletResponse getResponse() 获取响应
```

MockMvcBuilders

```

package org.springframework.test.web.servlet.setup;
public final class MockMvcBuilders
private MockMvcBuilders() { }
public static DefaultMockMvcBuilder webAppContextSetup(WebApplicationContext context) 返回初始化的 MockMvc 对象（使用 AbstractMockMvcBuilder 的 build 方法）

```

MockMvcRequestBuilders

```

package org.springframework.test.web.servlet.request;
public abstract class MockMvcRequestBuilders 请求构建
public MockMvcRequestBuilders(){ }

public static MockHttpServletRequestBuilder post(URI uri) 构建 post 请求

```

MockHttpServletRequestBuilder

```

package org.springframework.test.web.servlet.request;
public class MockHttpServletRequestBuilder http 请求实例

public MockHttpServletRequestBuilder header(String name, Object... values) 设置一个请求头参数
public MockHttpServletRequestBuilder param(String name, String... values) 设置一个请求行参数

public MockHttpServletRequestBuilder accept(MediaType... mediaTypes) 设置返回值类型
public MockHttpServletRequestBuilder contentType(MediaType contentType) 请求数据格式
public MockHttpServletRequestBuilder content(String content) 请求体

```

context

注解

```

package org.springframework.test.context;
@TestPropertySource 覆盖系统环境变量和 Java 系统属性
locations = { "classpath:application-test.properties" } 引入外部配置文件

```

transaction

PlatformTransactionManager

```
package org.springframework.transaction;  
public interface PlatformTransactionManager
```

This is the central interface for transaction management in Spring. It defines methods for beginning, committing, and rolling back transactions.

DataSourceTransactionManager	Default option, works with any JDBC database. Simple but less feature-rich.
HibernateTransactionManager	For Hibernate applications, leverages Hibernate's features for transaction management.
JpaTransactionManager	For JPA applications, works with JPA's transaction capabilities.
JdoTransactionManager	Manages transactions for JDO applications (less common).
JtaTransactionManager	Enables distributed transactions across multiple resources (complex setup).

```
TransactionStatus getTransaction(@Nullable TransactionDefinition var1);  
void commit(TransactionStatus var1) throws TransactionException;  
void rollback(TransactionStatus var1) throws TransactionException;
```

Usage

```
@Service  
public class MyService {  
  
    @Autowired  
    private MyDao dao;  
  
    @Autowired  
    private PlatformTransactionManager transactionManager;  
  
    public void updateUserDataManually(Long userId, String newEmail) throws Exception {  
        TransactionStatus status = null;  
        try {  
            // Begin the transaction manually  
            status = transactionManager.getTransaction(new DefaultTransactionDefinition());  
            dao.updateUserEmail(userId, newEmail);  
            // More logic here...  
  
            // Manually commit the transaction if successful  
            transactionManager.commit(status);  
        } catch (Exception ex) {  
            // Handle exception and potentially rollback  
            if (status != null) {  
                transactionManager.rollback(status);  
            }  
            throw ex;  
        } finally {  
            // No release method needed, Spring manages resources  
        }  
    }  
}
```

TransactionDefinition

```
package org.springframework.transaction;  
public interface TransactionDefinition
```

```
default int getPropagationBehavior()  
default int getIsolationLevel()  
default int getTimeout()
```

```
default boolean isReadOnly()  
default String getName()
```

TransactionStatus

```
package org.springframework.transaction;  
public interface TransactionStatus extends SavepointManager, Flushable  
boolean hasSavepoint()  
void flush()
```

```
boolean isNewTransaction()  
void setRollbackOnly()  
boolean isRollbackOnly()
```

Check whether the current transaction has been marked for rollback only. This means that the transaction **cannot be committed** and **will be rolled back** when it completes.

```
boolean isCompleted()
```

interceptor

TransactionAspectSupport

```
package org.springframework.transaction.interceptor;  
public abstract class TransactionAspectSupport implements BeanFactoryAware, InitializingBean
```

This class provides **the foundational support for transactional aspects**. It contains the core logic for managing transactions in Spring, including working with PlatformTransactionManager and TransactionInterceptor.

Key Responsibilities

- Transaction Management
Manages **the lifecycle of transactions**, including beginning, committing, and rolling back transactions.
- Transaction Context
Maintains **the context of ongoing transactions** and **ensures correct propagation behavior**.
- Transaction Configuration
Reads and applies transaction attributes from annotations or XML configurations.

```
protected TransactionInfo createTransactionIfNecessary(@Nullable PlatformTransactionManager tm,  
@Nullable TransactionAttribute txAttr, final String joinpointIdentification)
```

Begins a new transaction or participates in an existing one.

```
protected void commitTransactionAfterReturning(@Nullable TransactionInfo txInfo)
```

Commits the transaction after the method execution if no exceptions were thrown.

```
protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txInfo, Throwable ex)
```

Rolls back the transaction if an exception was thrown during method execution.

TransactionProxyFactoryBean

```
package org.springframework.transaction.interceptor;
```

```
public class TransactionProxyFactoryBean extends AbstractSingletonProxyFactoryBean implements BeanFactoryAware
```

Spring internally uses TransactionProxyFactoryBean **to create the transactional proxies**. This is a specialized factory bean that creates proxies and applies transactional behavior to the beans.

org.springframework.transaction.interceptor.TransactionInterceptor

org.springframework.transaction.interceptor.TransactionAspectSupport

org.springframework.transaction.support.TransactionSynchronizationManager

org.springframework.transaction.support.AbstractPlatformTransactionManager

org.springframework.transaction.PlatformTransactionManager

`org.springframework.transaction.support.TransactionTemplate`

```
public void setTransactionManager(PlatformTransactionManager transactionManager)
```

How it Works

Method/Class Annotation

You mark your methods or classes with the `@Transactional` annotation.

Public Methods

The `@Transactional` annotation should be applied to public methods.

This is because Spring AOP proxies **only intercept public method calls**.

Non-public methods (private, protected, or package-private) will not be proxied and thus will not have transactional behavior applied.

Final Methods:

Methods that are marked as final cannot be overridden, which means that Spring AOP proxies **cannot intercept calls to these methods**. Therefore, `@Transactional` will not work on final methods.

Final Classes:

If a class is marked as final, it cannot be subclassed. Spring AOP uses **subclassing** to create proxies, so `@Transactional` will not work on final classes.

Private Methods:

Private methods are not visible to subclasses and hence cannot be intercepted by Spring AOP proxies. Thus, `@Transactional` will not work on private methods.

Transaction Initiation

When a transactional method is invoked, Spring Boot retrieves **a transaction object** from the configured `PlatformTransactionManager`.

Database Operations

Your code **interacts with the database**, performing inserts, updates, or deletes.

Transaction Completion:

- Commit
If all database operations succeed, Spring Boot **calls the commit method** on the transaction object, making the changes permanent.
- Rollback
If any exception occurs during database operations, Spring Boot **calls the rollback method**, undoing any changes made within the transaction.

Transaction Propagation Behavior

In Spring's transaction management, propagation behavior defines how a transaction initiated in a method call **extends** or **influences transactions** in methods it calls.

It essentially determines whether **the called method inherits the transactional context of the calling method**.

Spring **provides several propagation behaviors** configurable through the `@Transactional` annotation's propagation attribute:

1. REQUIRED **Joins** or **starts** a new transaction.
2. REQUIRES_NEW Always **starts a new, independent transaction**.
3. SUPPORTS **Joins** if a transaction exists, otherwise **runs non-transactionally**.
4. NOT_SUPPORTED Runs **non-transactionally**, suspending any existing transaction.
5. MANDATORY **Joins** if a transaction exists, otherwise **throws an exception**.
6. NEVER Ensures no transaction is present, **throws an exception** if a transaction exists.
7. NESTED Creates a **nested transaction** within an existing one or **starts** a new transaction.

Handle Exception

1. RuntimeException (Unchecked Exception)

By default, Spring rolls back a transaction if a **RuntimeException or its subclass is thrown**.

```
@Transactional  
public void methodThatThrowsRuntimeException() {  
    // Some business logic  
    throw new RuntimeException("Something went wrong");  
    // Transaction will be rolled back  
}
```

2. Checked Exception (Exception)

By default, Spring does not roll back a transaction if a **checked exception** (i.e., a subclass of `Exception` but not `RuntimeException`) is thrown.

You **need to explicitly configure** Spring to roll back for checked exceptions.

```
@Transactional(rollbackFor = Exception.class)  
public void methodThatThrowsCheckedException() throws Exception {  
    // Some business logic  
    throw new Exception("Checked exception occurred");  
    // Transaction will be rolled back due to rollbackFor attribute  
}
```

3. Error

Errors (i.e., subclasses of `java.lang.Error`) are not exceptions, and they represent serious problems that a reasonable application should not try to catch.

By default, Spring **rolls back the transaction** when an Error is thrown.

```
@Transactional  
public void methodThatThrowsError() {  
    // Some business logic  
    throw new Error("Serious error occurred");  
    // Transaction will be rolled back  
}
```

4. Using try-catch Blocks

Using try-catch blocks within a transactional method can affect transaction management.

When you catch an exception, it prevents Spring from handling the exception and rolling back the transaction unless you **explicitly mark the transaction** for rollback.

```
@Transactional  
public void methodWithTryCatch() {  
    try {  
        // Some business logic  
        throw new RuntimeException("Something went wrong");  
    } catch (RuntimeException e) {  
        // Handle exception  
        System.out.println("Caught exception: " + e.getMessage());  
        // Manually mark transaction for rollback  
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();  
    }  
    // Transaction will be rolled back due to setRollbackOnly  
}
```

Self Invocation

When you **call a method on the same class**, the call **does not go through the proxy**, and thus the transaction management logic is not applied.

This is known as the "self-invocation" problem.

Spring uses CGLIB for proxying non-interface based beans, the self-invocation issue still occurs in this specific scenario.

```
@Service  
public class MyService {  
  
    @Transactional  
    public void outerMethod() {  
        // Business logic  
        innerMethod(); // This will not be transactional  
    }
```

```

    @Transactional
    public void innerMethod() {
        // More business logic
    }
}

```

Solutions

Refactor to Use Another Bean: Move the inner method to another bean and inject that bean.

```

@Service
public class MyService {

    @Autowired
    private MyOtherService myOtherService;

    @Transactional
    public void outerMethod() {
        // Business logic
        myOtherService.innerMethod(); // This will be transactional
    }
}

@Service
public class MyOtherService {

    @Transactional
    public void innerMethod() {
        // More business logic
    }
}

```

Use AOP Context Lookup: Programmatically obtain the proxy for the current object using AopContext.

```

@Service
public class MyService {

    @Transactional
    public void outerMethod() {
        // Business logic
        ((MyService) AopContext.currentProxy()).innerMethod(); // This will be transactional
    }

    @Transactional
    public void innerMethod() {
        // More business logic
    }
}

```

Enable AspectJ Mode

Use AspectJ mode instead of Spring AOP proxies.

This requires additional configuration and dependencies but allows transactions to work on self-invoked methods.

`@EnableAspectJAutoProxy`

Annotation to enable AspectJ proxy-based AOP.

`proxyTargetClass = true`

Forces the use of CGLIB proxies, enabling class-based proxies regardless of whether the class implements interfaces.

AppConfig

```

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class AppConfig {
    // Application configuration
}

```

```

MyService
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class MyService {

    @Transactional
    public void myTransactionalMethod() {
        // Business logic
        anotherTransactionalMethod(); // This will be transactional
    }

    @Transactional
    public void anotherTransactionalMethod() {
        // More business logic
    }
}

```

TransactionInterceptor

package org.springframework.transaction.interceptor;
 public class **TransactionInterceptor** extends TransactionAspectSupport implements MethodInterceptor, Serializable

This is an AOP interceptor that wraps method calls in a transaction.

It handles **starting a transaction before method execution** and **committing or rolling it back afterward**, depending on whether an exception is thrown.

The TransactionInterceptor works with the transaction manager (PlatformTransactionManager) to apply transaction logic around the method calls.

reactive

TransactionSynchronizationManager

package org.springframework.transaction.reactive;
 public class **TransactionSynchronizationManager** 事务同步管理器，spring 开启事务会在内部存储事务信息
 This class manages the synchronization of resources (like connections) with the current transaction.
 It is used internally by Spring to bind resources to the current transaction, ensuring that all operations performed within the transaction are properly synchronized.

```

public static String getCurrentTransactionName()          直接获取事务名称
public static Integer getCurrentTransactionIsolationLevel()  直接获取事务隔离级别
getResource
@Nullable
public static Object getResource(Object key) {
    // 从事务同步管理器中获取当前线程中绑定的连接
    // ConnectionHolder conHolder = (ConnectionHolder)TransactionSynchronizationManager.getResource(dataSource);
    Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);
    return doGetResource(actualKey);
}
bindResource
public static void bindResource(Object key, Object value) throws IllegalStateException {
    // 将获取到的连接绑定到当前线程
    Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);
    Assert.notNull(value, "Value must not be null");
    Map<Object, Object> map = (Map)resources.get();
}

```

```

if (map == null) {
    map = new HashMap();
    resources.set(map);
}

Object oldValue = ((Map)map).put(actualKey, value);
if (oldValue instanceof ResourceHolder resourceHolder) {
    if (resourceHolder.isVoid()) {
        oldValue = null;
    }
}

if (oldValue != null) {
    throw new IllegalStateException("Already value [" + oldValue + "] for key [" + actualKey + "] bound to thread");
}
}

```

support

TransactionTemplate

```

package org.springframework.transaction.support;
public class TransactionTemplate extends DefaultTransactionDefinition implements TransactionOperations, InitializingBean

```

This class provides a programmatic way to manage transactions. It simplifies the programmatic transaction management by **encapsulating transaction handling code in a template method**.

While @Transactional is typically used for declarative transaction management, TransactionTemplate can be used for more fine-grained, programmatic control.

```

public <T> T execute(TransactionCallback<T> action) throws TransactionException
private void rollbackOnException(TransactionStatus status, Throwable ex) throws TransactionException

```

AbstractPlatformTransactionManager

```

package org.springframework.transaction.support;
public abstract class AbstractPlatformTransactionManager implements PlatformTransactionManager, Serializable

```

This abstract class provides the base implementation for transaction management and is extended by the concrete transaction manager classes like DataSourceTransactionManager, JpaTransactionManager, etc.

It **provides common transaction handling logic** that can be reused by various transaction manager implementations.

```
public final void setTransactionSynchronizationName(String constantName)
```

annotation

注解

```
package org.springframework.transaction.annotation;
```

@Transactional 声明式事务，可以设置当前类/接口中所有方法或具体方法开启事务，但只能在 public 方法上 【TYPE, METHOD】

默认情况下，Spring 会对 **受检异常** 进行事务回滚，如果是 **非受检异常** 则不回滚（你写代码出现的空指针等异常，会被回滚，文件读写，网络出问题，spring 就没法回滚了）

在多线程中，@Transactional 注解不生效

多线程情况下，可以使用**数据库事务回滚**，而不是使用业务事务回滚 (java.sql.Connection::rollback)

```

readonly = false,
timeout = -1,
isolation = Isolation.DEFAULT,
rollbackFor = {ArithemticException.class, IOException.class},

```

```
noRollbackFor = {},  
propagation = Propagation.REQUIRES_NEW    事务模式, 默认 REQUIRED  
@EnableTransactionManagement    开启注解驱动, 等同 XML 格式中的注解驱动
```

ui freemarker

FreeMarkerTemplateUtils

```
package org.springframework.ui.freemarker;  
public abstract class FreeMarkerTemplateUtils  
public static String processTemplateIntoString(Template template, Object model) throws IOException, TemplateException
```

util

FixedBackOff

```
package org.springframework.util.backoff;  
public class FixedBackOff implements BackOff  
A simple BackOff implementation that provides a fixed interval between two attempts and a maximum number of retries.
```

CompositIterator

```
package org.springframework.util;  
public class CompositIterator<E> implements Iterator<E>  
private final Set<Iterator<E>> iterators = new LinkedHashSet();  
private boolean inUse = false;  
  
add  
public void add(Iterator<E> iterator) {  
    Assert.state(!this.inUse, "You can no longer add iterators to a composite iterator that's already in use");  
    if (this.iterators.contains(iterator)) {  
        throw new IllegalArgumentException("You cannot add the same iterator twice");  
    } else {  
        this.iterators.add(iterator);  
    }  
}
```

ClassUtils

```
package org.springframework.util;  
public abstract class ClassUtils    类工具  
  
public static boolean isPresent(String className, @Nullable ClassLoader classLoader)  
public static Class<?> forName(String name, @Nullable ClassLoader classLoader)    替代 Class.forName()  
public static ClassLoader getDefaultClassLoader()    获取默认类加载器
```

ObjectUtils

```
package org.springframework.util;  
public abstract class ObjectUtils    对象工具  
public static boolean isEmpty(@Nullable Object obj)    判断值是否为空, 即使已经分配内存, 但没有赋值, 依然是空 (null 判断值是否为空, 没有分配内存, 可能出现空指针异常)
```

BeanUtils

```
package org.springframework.util;
public abstract class BeanUtils    bean 工具

public static void copyProperties(Object source, Object target) throws BeansException    对象全拷贝
    source 必须含有 target 的全部属性, source 还可以有多余的属性。。
    target 所有属性都会被覆盖, 不管是否有值, 是否为 null

    source 和 target 必须都有 setter 和 getter 方法
    如果存在属性完全相同的内部类, 但是不是同一个内部类, 即分别属于各自的内部类, 则 spring 会认为属性不同, 不会
    copy;
```

CollectionUtils

```
package org.springframework.util;
public abstract class CollectionUtils    集合工具

public static List<?> arrayToList(@Nullable Object source)    普通数组转 list
public static boolean contains(@Nullable Enumeration<?> enumeration, Object element)    检查给定的枚举 是否包含指定元素
public static boolean contains(@Nullable Iterator<?> iterator, Object element)    检查给定的遍历器 是否包含指定元
素
public static boolean containsAny(Collection<?> source, Collection<?> candidates)    包含任意一个
```

PropertyPlaceholderHelper

```
package org.springframework.util;
public class PropertyPlaceholderHelper
public String replacePlaceholders(String value, final Properties properties)
```

Usage

```
import org.springframework.util.PropertyPlaceholderHelper;
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // Sample map with the key-value pair
        Map<String, String> map = new HashMap<>();
        map.put("orgId", "12345");

        // Input string containing the placeholder
        String input = "${orgId}";

        // Create a PropertyPlaceholderHelper instance
        PropertyPlaceholderHelper placeholderHelper = new PropertyPlaceholderHelper("${", "}");

        // Define a placeholder resolver using the map
        String resolvedString = placeholderHelper.replacePlaceholders(input, map::get);

        // Output the resolved value
        System.out.println("Resolved value: " + resolvedString);
    }
}
```

ReflectionUtils

```
package org.springframework.util;
public abstract class ReflectionUtils

public static void doWithMethods(Class<?> clazz, ReflectionUtils.MethodCallback mc, @Nullable ReflectionUtils.MethodFilter
```

mf)

StopWatch

```
package org.springframework.util;
public class StopWatch 时钟
public StopWatch()
public StopWatch(String id)
public void start() throws IllegalStateException 开始计时
public void start(String taskName) throws IllegalStateException 开始计时, 添加任务名
public void stop() throws IllegalStateException 停止计时
public boolean isRunning() Threre are currently tasks running.
public String getLastTaskName() throws IllegalStateException 获取最后任务名称, stop 之后才能获取
public boolean isRunning() 是否正在运行
public long getTotalTimeMillis() 获取总任务时间毫秒数
public double getTotalTimeSeconds() 获取任务总时间秒数
public int getTaskCount() 获取任务数
public StopWatch.TaskInfo[] getTaskInfo() 获取任务信息
```

MultiValueMap

```
package org.springframework.util;
public interface MultiValueMap<K, V> extends Map<K, List<V>> 多值 Map, Map 存储多个值的接口的扩展。
```

LinkedMultiValueMap

```
package org.springframework.util;
public class LinkedMultiValueMap<K, V> extends MultiValueMapAdapter<K, V> implements Serializable, Cloneable 这个
Map 的 value 有点特别, 他不能放任何值, 必须是一个 List。
```

ConcurrentReferenceHashMap

```
public class ConcurrentReferenceHashMap<K, V> extends AbstractMap<K, V> implements ConcurrentMap<K, V>
引用的使用意味着不能保证放置在 Map 中的项目随后可用。垃圾收集器可能会随时丢弃引用, 因此可能会出现未知线程正在静默
删除条目的情况。
```

Assert

```
package org.springframework.util;
public abstract class Assert 断言工具
public static void state(boolean expression, String message) IllegalStateException 检查
public static void notNull(@Nullable Object object, String message) IllegalArgumentException 检查
public static void assignable(Class<?> superType, Class<?> subType) 如果子类非继承父类, 抛出此异常
```

StringUtils

```
package org.springframework.util;
public abstract class StringUtils
```

commaDelimitedListToStringArray

```
public static String[] commaDelimitedListToStringArray(@Nullable String str) {
    return delimitedListToStringArray(str, ",");
}
```

```

delimitedListToStringArray
public static String[] delimitedListToStringArray(@Nullable String str, @Nullable String delimiter) {
    return delimitedListToStringArray(str, delimiter, (String)null);
}

delimitedListToStringArray
public static String[] delimitedListToStringArray(@Nullable String str, @Nullable String delimiter, @Nullable
String charsToDelete) {
    if (str == null) {
        return EMPTY_STRING_ARRAY;
    } else if (delimiter == null) {
        return new String[]{str};
    } else {
        List<String> result = new ArrayList();
        int pos;
        if (delimiter.isEmpty()) {
            for(pos = 0; pos < str.length(); ++pos) {
                result.add(deleteAny(str.substring(pos, pos + 1), charsToDelete));
            }
        } else {
            int delPos;
            for(pos = 0; (delPos = str.indexOf(delimiter, pos)) != -1; pos = delPos + delimiter.length()) {
                result.add(deleteAny(str.substring(pos, delPos), charsToDelete));
            }

            if (str.length() > 0 && pos <= str.length()) {
                result.add(deleteAny(str.substring(pos), charsToDelete));
            }
        }
    }

    return toStringArray((Collection)result);
}
}

```

validation

annotation

注解

```
package org.springframework.validation.annotation;
```

@Validated	开启校验, 如果数据异常则会统一抛出异常	【TYPE, METHOD, PARAMETER】
-------------------	----------------------	---------------------------

web

accept

ContentNegotiationManager

```
package org.springframework.web.accept;
```

```
public class ContentNegotiationManager implements ContentNegotiationStrategy, MediaTypeFileExtensionResolver
```

```
private final List<ContentNegotiationStrategy> strategies;
```

```
public ContentNegotiationManager(ContentNegotiationStrategy... strategies)
```

```

public List<MediaType> resolveMediaTypes(NativeWebRequest request) throws HttpMediaTypeNotAcceptableException
{
    Iterator var2 = this.strategies.iterator();

    List mediaTypes;
    do {
        if (!var2.hasNext()) {
            return MEDIA_TYPE_ALL_LIST;
        }

        ContentNegotiationStrategy strategy = (ContentNegotiationStrategy)var2.next();
        mediaTypes = strategy.resolveMediaTypes(request);
    }
}
```

```
    } while(mediaTypes.equals(MEDIA_TYPE_ALL_LIST));
    return mediaTypes;
}
```

ContentNegotiationStrategy

```
package org.springframework.web.accept;
public interface ContentNegotiationStrategy
List<MediaType> MEDIA_TYPE_ALL_LIST = Collections.singletonList(MediaType.ALL);
List<MediaType> resolveMediaTypes(NativeWebRequest webRequest) throws HttpMediaTypeNotAcceptableException;
bind
package org.springframework.web.bind;
```

WebDataBinder

```
public class WebDataBinder extends DataBinder
```

(exception)

MissingServletRequestParameterException

```
package org.springframework.web.bind;
public class MissingServletRequestParameterException extends ServletRequestBindingException
ServletRequestBindingException subclass that indicates a missing parameter.
```

annotation

RestController

```
package org.springframework.web.bind.annotation;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController
```

发送响应体，而不是跳转页面（等同@Controller + @ResponseBody，并且所有方法都是发送响应体，省略注释符，而不是跳转到某个页面） 【TYPE】

ResponseBody

```
package org.springframework.web.bind.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ResponseBody
```

发送响应体（所有方法都是发送响应体，无需在方法前添加 ResponseBody）【TYPE, METHOD】

该注解用于将 Controller 的方法返回的对象，通过适当的 [HttpMessageConverter](#) 转换为指定格式后，写入到 Response 对象的 body 数据区

返回的数据不是 html 标签的页面，而是其他某种格式的数据时（json, xml 等）使用

注解添加到 pojo 参数前方时，封装的异步提交数据按照 Pojo 的属性格式进行关系映射
注解添加到集合参数前方时，封装的异步提交数据按照集合的存储结构进行关系映射

Usage

```
@RequestMapping("/login")
@ResponseBody
public User login(User user){
    return user;
}
```

RequestMapping

```
package org.springframework.web.bind.annotation;
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Mapping
@Reflective(ControllerMappingReflectiveProcessor.class)
public @interface RequestMapping
```

Data binding

Spring's data binding mechanism will automatically populate the object properties based on the matching parameter names in the request.

```
@AliasFor("path")
```

```
String[] value() default {};
```

Defines the URL pattern to match.

You can specify one or more URL paths. value and path are used interchangeably.

```
@AliasFor("value")
```

```
String[] path() default {};
```

```
RequestMethod[] method() default {};
```

Specifies the HTTP method to match, such as GET, POST, PUT, DELETE.

If not specified, the method will handle all HTTP methods by default.

```
    @RequestMapping(value = "/submit", method = RequestMethod.POST)
```

```
String[] params() default {};
```

Narrows down the request mapping based on specific request parameters.

```
    @RequestMapping(value = "/filter", params = "type=special")
```

```
String[] headers() default {};
```

Matches requests based on HTTP headers.

```
    @RequestMapping(value = "/download", headers = "content-type=text/plain")
```

```
String[] consumes() default {};
```

Limits the requests based on the content type of the incoming request.

```
    @RequestMapping(value = "/upload", consumes = "multipart/form-data")
```

```
String[] produces() default {};
```

Restricts the request based on the media type (MIME type) that the method can produce (response content type).

```
    @RequestMapping(value = "/json", produces = "application/json")
```

Post Request

Jackson is used for request bodies (@RequestBody) and response serialization (@ResponseBody).

Spring does not use Jackson for @RequestParam, so you must manually handle the conversion.

@JsonCreator allows Jackson to map a single string to the correct enum.

@JsonValue ensures that when sending responses, only the code field is returned instead of the full enum name.

```
@RestController
@RequestMapping("/test")
public class MyController {
```

```
    @PostMapping
    public ResponseEntity<String> test(@RequestBody Color color) {
        return ResponseEntity.ok("Received: " + color);
    }
}
```

Get Request

POJO Object Parameters

```
@RequestMapping("/test")
public String testRequest(Person per) {
    // Person class contains fields 'id', 'name', 'age'.
}
http://localhost/testRequest/{id}?name=lala&age=20
```

When a request contains parameters that match the fields of a POJO (Plain Old Java Object), Spring automatically binds the request parameters to the corresponding POJO properties. This works for both query parameters and path variables.

POJO with Additional Parameters

```
@RequestMapping("/test")
public String testRequest(Person per, String name, Integer age) {
    // 'age' will be mapped both to the method parameter and the POJO field if they share the same name.
}
```

When you have additional parameters beyond the POJO (e.g., age), and they share the same name with POJO fields, Spring will map the non-POJO parameter to the POJO field as well.

POJO with Nested Object Parameters

```
@RequestMapping("/test")
public String testRequest(Person per) {
    // Person contains 'name' and 'address' object
}
http://localhost/testRequest?name=lala&address.province=beijing
```

When the POJO contains other objects as fields, the request parameter names must reflect the nested object structure.

POJO with Collection Fields (Complex Objects)

```
@RequestMapping("/test")
public String testRequest(Person per) {
    // Person contains List<Address> addressList
}
http://localhost/testRequest?addressList[0].province=bj&addressList[1].province=tj
http://localhost/testRequest?addressList=bj&addressList=tj
```

For collections of complex objects (e.g., List<Address>), parameters need to use an indexed structure or array name for binding.

POJO with Map Fields

```
@RequestMapping("/test")
public String testRequest(Person per) {
    // Person contains Map<String, Address> addressMap
}
http://localhost/testRequest?addressMap['home'].province=bj&addressMap['job'].province=tj
```

If the POJO contains a Map, the request parameter names must use the map keys as part of the parameter structure.

Other

Model model

在 Model 中添加属性，实现页面跳转到 jsp 时携带数据

```
<a> ${name} </a>
```

ModelAndView modelAndView

封装数据并封装视图，包含 Model 和 view 两个对象，传递数据并跳转页面，并自动通过设置的 viewName 跳转到对应页面

Model

RidirectAttributes
ServletResponse
SessionStatus
Errors/BindingResult
UriComponnetsBuilder
ServletUriComponentBuilder

Response

return "page.jsp" 跳转页面，返回值为 String，等同 return "/WEB-INF/page.jsp"
作为类注解时，跳转路径会加上前缀 // class = "/user" method
= "showPage" return "page.jsp" ==> user/page.jsp
使用 void 返回值时，默认使用访问路径作为页面地址的前缀后缀。 // method = "showPage" return
null ==> showpage
可以通过配置 InternalResourceViewReesolver 定义通用访问路径

return "forward:page.jsp" 转发，浏览器路径不变（默认）
return "redirect:page.jsp" 重定向，浏览器路径发生变化

return ResponseEntity ResponseBodyEmitter StreamingResponseBody HttpEntity HttpHeaders Callable DeferredResult
ListenableFuture CompletionStage WebAsyncTask

CrossOrigin

```
package org.springframework.web.bind.annotation;  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface CrossOrigin  
    自动添加跨域请求头，支持跨域访问
```

ControllerAdvice

```
package org.springframework.web.bind.annotation;  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Component  
public @interface ControllerAdvice
```

设置当前类为异常处理器类，它是一个 Controller 增强器，可对 controller 中被 @RequestMapping 注解的方法加一些逻辑处理
【TYPE】

有了服务降级就不会走入 ControllerAdvice 异常处理中了

ExceptionHandler

```
package org.springframework.web.bind.annotation;  
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Reflective(ExceptionHandlerReflectiveProcessor.class)  
public @interface ExceptionHandler
```

设置指定异常的处理方式【METHOD】

PathVariable

```
package org.springframework.web.bind.annotation;
```

```

@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface PathVariable

    获取路径匹配变量 【PARAMETER】

        @RequestMapping(value="module/download/apk/{ id:.+}",method=RequestMethod.GET)      // 可以使用
        Spring 正则表达式 SpEL
        @RequestMapping(value="viewFile/{module}/{ id:.+}",method=RequestMethod.GET)
        @PathVariable(value="id", required=false) Integer id, @PathVariable Map<String,String> pv
        value="id"          参数名
        required=false     是否必须参数

```

RequestParam

```

package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestParam

    获取请求行参数, application/json 参数, multipart/form-data 请求 (只支持 String)

        @RequestParam("age") String age      接收单个参数, 不指定名称默认就是参数名 age           @RequestParam Map<String,
        String> params      接受全部参数

        @RequestParam("strList") List<String> strList      接受数组参数, 且请求参数数量>1 个
        http://localhost/requestParam6?strList=Jockme&strList=Jocker
        springMvc 默认将 List 作为对象处理, 赋值前根据类型先创建对象, 然后将 strList 作为对象的属性进行处理。
        由于 List 是接口, 无法创建对象, 报无法找到构造方法异常, 修复类型为可创建对象的 ArrayList 类型后, 对象可以创建, 但
        没有 strList 属性, 因此数据为空。
        此时需要告知 springMVC 的处理器 strList 是一组数据, 而不是一个单一数据。
        通过@RequestParam 注解, 将数量大于 1 个 names 参数打包成参数数组后, springMvc 才能识别该数据格式, 并判定形参
        类型是否为数组或集合, 并按数组或集合对象的形式操作数据。

```

```

name = "userName",      请求行参数名
required = true,
defaultValue = "itheima"  默认值

```

RequestPart

```

package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestPart

    获取 multipart/form-data 请求 (必须指定键接收 String 或文件或 json。指定一个键接收 对应的文件或者多个文件, 文件值转
    json 或 map 时会出错)
        @RequestPart("alimgkey") MultipartFile alimg      @RequestPart("photokey") MultipartFile[] photos

```

RequestHeader

```

package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented

```

```
public @interface RequestHeader
    获取请求头参数
        @RequestHeader("User-Agent") String userAgent      @RequestHeader Map<String, String> header
```

RequestBody

```
package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestBody
    获取请求体内容 (application/octet-stream 直接用 String 接收)
        @RequestBody String content      JSONObject content
```

MatrixVariable

```
package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MatrixVariable
    获取矩阵变量, 需要手动开启此功能
        /name1;age=32;arr=byd,audi,yd  传递矩阵参数必须在 PathVaribale 匹配参数后      pathVar 指定匹配前方的
        PathVariable 匹配参数
        @MatrixVariable("age") String age          @ MatrixVariable(value="arr", pathVar="name")
        List<String> arr
```

CookieValue

```
package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CookieValue
    获取 Cookie 键对应的值
        @CookieValue("ga") String ga           Cookie cookie
```

RequestAttribute

```
package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestAttribute
    获取 servlet 的 request 参数 (获取转发参数)
        @RequestAttribute(value="age") String age      HttpServletRequest request
```

SessionAttribute

```
package org.springframework.web.bind.annotation;
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface SessionAttribute
    获取 Session 数据
        @SessionAttribute("name") String ga       Session cookie
```

ModelAttribute

```
package org.springframework.web.bind.annotation;  
@Target({ElementType.PARAMETER, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Reflective  
public @interface ModelAttribute
```

将方法的参数或方法的返回值绑定到指定的模型属性上，并返回给 Web 视图 【PARAMETER, METHOD】

运行流程：在标注的 controller，内部的每个方法执行前被执行

存值要求：在存入 map 时，map 的键值要和被@RequestMapping("/testModelAttribute")修饰的目标方法入参类型的第一个字符串的字符串一致

页面回显：页面获取到 model 数据，取出数据显示在页面

TestController

```
@Controller  
@RequestMapping("/springmvc")  
public class TestController {  
    @RequestMapping("/testModelAttribute")  
    public String testModelAttribute01(User user){  
        System.out.println("修改：" + user);  
        return "success";  
    }  
  
    @ModelAttribute  
    public void testModelAttribute(@RequestParam(value = "id", required = false)  
                                    Integer id, Map<String, Object> map){  
        if(id != null){  
            User user = new User(1, "Tom", "123456", "aa@aa.com", 12);  
            map.put("user", user);  
        }  
    }  
}
```

index.html

```
<!--  
模拟修改操作  
1. 原始数据：1, Tom, aa@aa.com, 12  
2. 密码不能被修改  
3. 表单回显，模拟操作直接填写对应的属性值  
-->  
<form action="/springmvc/testModelAttribute" method="post">  
    <input type="hidden" name="id" value="1">  
    username:<input type="text" name="username" value="Tom">  
    email:<input type="text" name="email" value="aa@aa.com">  

```

SessionAttributes

```
package org.springframework.web.bind.annotation;  
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Inherited  
@Documented  
public @interface SessionAttributes
```

声明放入 session 范围的变量名称，适用于 Model 类型数据传参

```

@Controller
@SessionAttributes(names={"name"})
public class ServletController {
    @RequestMapping("/setSessionData2")
    public String setSessionDate2(Model model) {
        model.addAttribute("name", "Jock2");
        return "page.jsp";
    }
}

```

InitBinder

package org.springframework.web.bind.annotation;

@Target({ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Reflective

public @interface **InitBinder**

Customize data binding for controller methods. It allows you to register custom editors or converters to preprocess request parameters before they are passed to controller methods.

@InitBinder applies only to parameters that match a specific type registered in WebDataBinder.

Usage

@RestController

@RequestMapping("/test")

public class MyController {

@GetMapping("/convert")

public ResponseEntity<String> test(@NotNull @RequestParam Color param1, @RequestParam String param2) {
 return ResponseEntity.ok("Converted: " + param1);
 }

@GetMapping("/other")

public ResponseEntity<String> anotherTest(@RequestParam String param3) {
 return ResponseEntity.ok("Received: " + param3);
 }

@InitBinder

public void initBinder(WebDataBinder binder) {
 binder.registerCustomEditor(Color.class, new PropertyEditorSupport() {
 @Override
 public void setAsText(String text) {
 setValue(Color.fromCode(text)); // Convert input string to Color enum
 }
 });
}

Only param1 will trigger @InitBinder.

The setValue method is defined in PropertyEditorSupport, which is a class in Java Beans that allows custom property editing.

By default, if setValue(null) is called, Spring will bind null to the parameter.

the @InitBinder method will be executed first to handle the binding and conversion of request parameters before the validation annotations, such as @NotNull, are checked.

Custom PropertyEditorSupport

```

public class BaseController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.registerCustomEditor(Date.class, new MyDateEditor());
        binder.registerCustomEditor(Double.class, new DoubleEditor());
        binder.registerCustomEditor(Integer.class, new IntegerEditor());
    }

    private class MyDateEditor extends PropertyEditorSupport {
        @Override
        public void setAsText(String text) throws IllegalArgumentException {
            SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            Date date = null;
            try {
                date = format.parse(text);
            } catch (ParseException e) {
                format = new SimpleDateFormat("yyyy-MM-dd");
                try {
                    date = format.parse(text);
                } catch (ParseException e1) {
                }
            }
            setValue(date);
        }
    }

    public class DoubleEditor extends PropertiesEditor {
        @Override
        public void setAsText(String text) throws IllegalArgumentException {
            if (text == null || text.equals("")) {
                text = "0";
            }
            setValue(Double.parseDouble(text));
        }

        @Override
        public String getAsText() {
            return getValue().toString();
        }
    }

    public class IntegerEditor extends PropertiesEditor {
        @Override
        public void setAsText(String text) throws IllegalArgumentException {
            if (text == null || text.equals("")) {
                text = "0";
            }
            setValue(Integer.parseInt(text));
        }

        @Override
        public String getAsText() {
            return getValue().toString();
        }
    }
}

```

support

WebDataBinderFactory

```

package org.springframework.web.bind.support;
public interface WebDataBinderFactory
    WebDataBinder createBinder(NativeWebRequest webRequest, @Nullable Object target, String objectName) throws Exception;

```

context

WebApplicationContext

org.springframework.web.context

public interface **WebApplicationContext** WebApplicationContext 是专门为 Web 应用准备的，它允许从相对于 Web 根目录的路径中装载配置文件完成初始化工作。

extends ApplicationContext

String **ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE** = WebApplicationContext.class.getName() + ".ROOT";
Context attribute to bind root WebApplicationContext to on successful startup.

String **SCOPE_REQUEST** = "request"; Scope identifier for request scope: "request".
Supported in addition to the standard scopes "singleton" and "prototype".

String **SCOPE_SESSION** = "session"; Scope identifier for session scope: "session".
Supported in addition to the standard scopes "singleton" and "prototype".

String **SCOPE_APPLICATION** = "application"; Scope identifier for the global web application scope: "application".
Supported in addition to the standard scopes "singleton" and "prototype".

String **SERVLET_CONTEXT_BEAN_NAME** = "servletContext"; Name of the ServletContext environment bean in the factory.

String **CONTEXT_PARAMETERS_BEAN_NAME** = "contextParameters"; Name of the ServletContext init-params environment bean in the factory.

Note: Possibly merged with ServletConfig parameters.
ServletConfig parameters override ServletContext parameters of the same name.

String **CONTEXT_ATTRIBUTES_BEAN_NAME** = "contextAttributes"; Name of the ServletContext attributes environment bean in the factory.

ServletContext **getServletContext()**; Return the standard Servlet API ServletContext for this application (WebApplicationContext)

ContextLoaderListener

org.springframework.web.context

```

public class ContextLoaderListener      上下文加载监听器，Spring 容器初始化
    extends ContextLoader implements ServletContextListener

public void contextInitialized(ServletContextEvent event) {      在启动项目时触发
    initWebApplicationContext(event.getServletContext());
}



## ContextLoader


org.springframework.web.context
public class ContextLoader

public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {      对 ApplicationContext 的
    初始化方法，也就是到这里正是进入了 springIOC 的初始化

    if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE) != null) {      判断
        当前 servletContext 容器中是否已经存在参数 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，避免重复创建 Spring 容器
        throw new IllegalStateException(
            "Cannot initialize context because there is already a root application context present - " +
            "check whether you have multiple ContextLoader* definitions in your web.xml!");
    }

    servletContext.log("Initializing Spring root WebApplicationContext");      打印初始化信息
    Log logger = LogFactory.getLog(ContextLoader.class);
    if (logger.isInfoEnabled()) {
        logger.info("Root WebApplicationContext: initialization started");
    }
    long startTime = System.currentTimeMillis();      获取开始时间

    try {
        if (this.context == null) {      当前上下文为空时，实例化一个根容器并存储在本地实例变量中，以确保它在 ServletContext 关闭时可用。
            this.context = createWebApplicationContext(servletContext);
        }
        if (this.context instanceof ConfigurableWebApplicationContext) {      如果 创建的是
            ConfigurableWebApplicationContext 类型的容器，进行容器配置。
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;
            if (!cwac.isActive()) {      上下文尚未刷新时，设置父容器、并刷新当前容器
                if (cwac.getParent() == null) {      没有父容器时，加载父容器
                    ApplicationContext parent = loadParentContext(servletContext);
                    cwac.setParent(parent);      设置父容器
                }
                configureAndRefreshWebApplicationContext(cwac, servletContext);      配置容器并刷新
            }
        }
        servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);
    }

    ClassLoader ccl = Thread.currentThread().getContextClassLoader();      获取当前上下文类加载器
}

```

```

        if (ccl == ContextLoader.class.getClassLoader()) { 如果当前类加载器 == 当前线程的类
加载器, 将容器保存到当前线程
            currentContext = this.context;
        } else if (ccl != null) {c 如果当前类加载器 != 当前线程的类加载
器, 保存到线程类加载器集合中 Map<ClassLoader, WebApplicationContext>
            }
            if (logger.isInfoEnabled()) {
                long elapsedTime = System.currentTimeMillis() - startTime;
                logger.info("Root WebApplicationContext initialized in " + elapsedTime + " ms");
            }
            return this.context;
        }
        catch (RuntimeException | Error ex) {
            logger.error("Context initialization failed", ex);
            servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, ex);
            throw ex;
        }
    }
}

```

```

protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac, ServletContext sc) {
    if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
        String idParam = sc.getInitParameter(CONTEXT_ID_PARAM);
        if (idParam != null) {
            wac.setId(idParam);
        }
        else {
            wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX +
ObjectUtils.getString(sc.getContextPath()));
        }
    }

    wac.setServletContext(sc);
    String configLocationParam = sc.getInitParameter(CONFIG_LOCATION_PARAM);
    if (configLocationParam != null) {
        wac.setConfigLocation(configLocationParam);
    }

    ConfigurableEnvironment env = wac.getEnvironment();
    if (env instanceof ConfigurableWebEnvironment) {
        ((ConfigurableWebEnvironment) env).initPropertySources(sc, null);
    }

    customizeContext(sc, wac);
    wac.refresh();
}

```

RequestContextHolder

```
package org.springframework.web.context.request;
public abstract class RequestContextHolder      便于随时都能取到当前请求的 request 对象
public static RequestAttributes getRequestAttributes()      获取 request 属性对象 // HttpServletRequest request
= (ServletRequestAttributes)RequestContextHolder.getRequestAttributes().getRequest(); 获取当前线程的 request 对象

public static RequestAttributes currentRequestAttributes() // RequestAttributes requestAttributes =
RequestContextHolder.currentRequestAttributes();
                                         // String str = (String)
requestAttributes.getAttribute("name",RequestAttributes.SCOPe_SESSION); 从 session 中获取值
                                         // HttpServletRequest request =
( (ServletRequestAttributes)requestAttributes ).getRequest();   获取 servlet request
                                         //          HttpServletResponse response      =
( (ServletRequestAttributes)requestAttributes ).getResponse();  获得 servlet response
```

ServletRequestAttributes

```
package org.springframework.web.context.request;
public class ServletRequestAttributes extends AbstractRequestAttributes 根据前端请求获取方法名、参数、路径等信息
public final HttpServletRequest getRequest()  获取请求对象
```

WebRequest

```
package org.springframework.web.context.request;
public interface WebRequest extends RequestAttributes  web 请求
```

```
Map<String, String[]> getParameterMap();  获取请求参数
```

NativeWebRequest

```
package org.springframework.web.context.request;
public interface NativeWebRequest extends WebRequest  请求
```

request

NativeWebRequest

```
package org.springframework.web.context.request;
public interface NativeWebRequest extends WebRequest
<T> T getNativeRequest(@Nullable Class<T> requiredType);
Return the underlying native request object, if available.
```

support

RequestHandledEvent

```
package org.springframework.web.context.support;
public class RequestHandledEvent extends ApplicationEvent 在 Web 应用中，当一个 http 请求 (request) 结束触发该事件。
```

WebApplicationObjectSupport

```
package org.springframework.web.context.support;
public abstract class WebApplicationObjectSupport extends ApplicationObjectSupport implements ServletContextAware
public final void setServletContext(ServletContext servletContext)
```

annotation

RequestScope

```
package org.springframework.web.context.annotation;  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Scope("request")  
public @interface RequestScope  
  
@AliasFor(  
    annotation = Scope.class  
)  
ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;
```

cors

UrlBasedCorsConfigurationSource

```
package org.springframework.web.cors;  
public class UrlBasedCorsConfigurationSource implements CorsConfigurationSource 跨域实现  
public void registerCorsConfiguration(String path, CorsConfiguration config) 添加跨域配置  
使用———————
```

```
@Component  
@Configuration  
public class GateWayCorsConfig  
{  
    @Bean  
    public CorsFilter corsFilter() {  
        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
        final CorsConfiguration corsConfiguration = new CorsConfiguration();  
        corsConfiguration.addAllowedHeader("*");  
        corsConfiguration.addAllowedOrigin("*");  
        corsConfiguration.addAllowedMethod("*");  
        source.registerCorsConfiguration("/**", corsConfiguration);  
        return new CorsFilter(source);  
    }  
}
```

CorsConfiguration

```
package org.springframework.web.cors;  
public class CorsConfiguration 跨域配置  
public void addAllowedHeader(String allowedHeader)  
public void addAllowedOrigin(String origin)  
public void addAllowedMethod(String method)
```

reactive (spring-webflux)

WebClient

```
package org.springframework.web.reactive.function.client;
```

```
public interface WebClient
static WebClient.Builder builder()
Builder filter(ExchangeFilterFunction filter);
Builder clientConnector(ClientHttpConnector connector);
```

WebFluxConfigurer

```
package org.springframework.web.reactive.config;
public interface WebFluxConfigurer

default void configureContentTypeResolver(RequestedContentTypeResolverBuilder builder)
default void addCorsMappings(CorsRegistry registry)
default void configurePathMatching(PathMatchConfigurer configurer)
default void addResourceHandlers(ResourceHandlerRegistry registry)
default void configureArgumentResolvers(ArgumentResolverConfigurer configurer)
default void configureHttpMessageCodecs(ServerCodecConfigurer configurer)
```

Configure custom HTTP message readers and writers or override built-in ones.

```
default void addFormatters(FormatterRegistry registry)
```

Add custom formatters or converters for data binding in request parameters and path variables.

```
default Validator getValidator()
default MessageCodesResolver getMessageCodesResolver()
default WebSocketService getWebSocketService()
default void configureViewResolvers(ViewResolverRegistry registry)
```

Custom Codecs

```
@Override
public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
    configurer.defaultCodecs().jackson2JsonEncoder(
        new Jackson2JsonEncoder(objectMapper)
    );
    configurer.defaultCodecs().jackson2JsonDecoder(
        new Jackson2JsonDecoder(objectMapper)
    );
}
```

Custom Formatter for LocalDate

A custom formatter that converts LocalDate to and from String in a specific format.

In this LocalDateFormatter, we define how to convert a String to a LocalDate (using parse) and how to format a LocalDate back to a String (using print).

```
import org.springframework.format.Formatter;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Locale;

public class LocalDateFormatter implements Formatter<LocalDate> {

    private static final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

    @Override
    public LocalDate parse(String text, Locale locale) {
        return LocalDate.parse(text, formatter);
    }
}
```

```
    @Override
    public String print(LocalDate date, Locale locale) {
        return date.format(formatter);
    }
}
```

Registering the Formatter in a WebFlux Configuration

You can register this formatter by overriding the `addFormatters` method in your custom `WebFluxConfigurer` implementation. Here, `LocalDateFormatter` is added to the `FormatterRegistry`, so whenever the application needs to bind a `LocalDate` from a string (e.g., from a query parameter or form field), **this formatter will be used automatically**.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.reactive.config.WebFluxConfigurer;

@Configuration
public class WebFluxConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // Registering the custom LocalDateFormatter
        registry.addFormatter(new LocalDateFormatter());
    }
}
```

server

WebFilter

```
package org.springframework.web.server;
```

```
public interface WebFilter
```

Intercepts requests

WebFilters are used to intercept web requests and perform chained processing.

Filters HTTP request-response exchanges

WebFilter beans are automatically used to filter each exchange in the application context.

Supports cross-cutting requirements

WebFilters can be used to implement application-agnostic requirements like security and timeouts.

```
Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain);
```

HttpWebHandlerAdapter

```
package org.springframework.web.server.adapter;
```

```
public class HttpWebHandlerAdapter extends WebHandlerDecorator implements HttpHandler
```

```
public Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response) 处理请求
```

servlet

AsyncHandlerInterceptor

```
package org.springframework.web.servlet;
```

```
public interface AsyncHandlerInterceptor extends HandlerInterceptor 自定义异步拦截器
```

```
default void afterConcurrentHandlingStarted(HttpServletRequest request, HttpServletResponse response, Object handler)
throws Exception
```

HandlerInterceptor

```
package org.springframework.web.servlet;
public interface HandlerInterceptor      Spring MVC Interceptor
    org.springframework.http.client.ClientHttpRequestInterceptor

default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception
    Executed before the actual handler (controller method) is invoked. It returns a boolean value indicating whether or not
    the execution chain should proceed.
default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable
ModelAndView modelAndView) throws Exception
    Executed after the handler method has been invoked, but before the view is rendered. Allows for modification of the
    ModelAndView object.
default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception
ex) throws Exception
    Executed after the complete request has finished processing, including view rendering. Useful for resource cleanup.
```

HandlerInterceptorAdapter

```
package org.springframework.web.servlet;

public abstract class HandlerInterceptorAdapter implements AsyncHandlerInterceptor      自定义拦截器 (Spring5.3 开始支持
直接实现 HandlerInterceptor 和/或 AsyncHandlerInterceptor。)
default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
Controller 方法处理之前, 返回值为 false 时, 被拦截的处理器将不执行
    request 请求对象
    response 响应对象
    handler 被调用的处理器对象, 本质上是一个方法对象, 对反射中的 Method 对象进行了再包装
default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable
ModelAndView modelAndView) Controller 方法处理完之后, DispatcherServlet 进行视图的渲染之前, 也就是说在这个方法中你可以
以对 ModelAndView 进行操作
    modelAndView 如果处理器执行完成具有返回结果, 可以读取到对应数据与页面信息, 并进行调整
default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception
ex)           DispatcherServlet 进行视图的渲染之后, 无论 Controller 方法有没有执行
    ex 如果处理器执行过程中出现异常对象, 可以针对异常情况进行单独处理
```

配置

application-context.xml >>

```
<mvc:interceptors>      配置顺序为先配置执行位置, 后配置执行类
    <mvc:interceptor>
        <mvc:mapping path="/showPage*" />
        <mvc:exclude-mapping path="/showPage*" />      用于剔除不符合要求的配置项, 加速配置过程, 支持通配符*
            * 表示任意名称, /*仅表示根路径下任意名称, 不再往下匹配目录
            ** 表示当前路径及其子路径, /**表示根路径及其子路径下任意名称
        <bean class="com.itheima.interceptor.MyInterceptor"/>      只能配置一个
    </mvc:interceptor>
</mvc:interceptors>
```

使用

```

public class MyInterceptor implements HandlerInterceptor {
    public boolean preHandle(.....) throws Exception {
        System.out.println("preHandle");
        return true;
    }
    public void postHandle(.....) throws Exception {
        System.out.println("postHandle");
    }
    public void afterCompletion(.....) throws Exception {
        System.out.println("afterCompletion");
    }
}

```

HandlerExecutionChain

package org.springframework.web.servlet;

public class **HandlerExecutionChain** 处理执行器 执行链，由处理执行器对象 和 HandlerInterceptor 组成。
 这个对象包含了一个 handler (一般都是 HandlerMethod，主要是包含了请求路径的处理 Method，以及对应的 Controller 的 Bean 实例，和一些其他相关信息)
 可以找到这个请求是对应哪个 Controller 的哪个方法，以及哪些拦截器

```

private final Object handler;      handler 实例
private final List<HandlerInterceptor> interceptorList; 拦截器，可以对一个请求进行前置，后置拦截
private int interceptorIndex; 拦截器的下标

public void addInterceptor(HandlerInterceptor interceptor) 添加拦截器
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws Exception 处理请
求的前置调用
void applyPostHandle(HttpServletRequest request, HttpServletResponse response, @Nullable ModelAndView mv) throws
Exception 后置请求的处理

```

HandlerExceptionResolver

package org.springframework.web.servlet;

public interface **HandlerExceptionResolver** 自定义异常处理器 (这种配置加载比较晚，必须在参数接收完后才执行)
 ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, @Nullable Object handler,
 Exception ex);

使用

```

@Component
public class ExceptionResolver implements HandlerExceptionResolver {
    public ModelAndView resolveException(HttpServletRequest request,
                                         HttpServletResponse response,
                                         Object handler,
                                         Exception ex) {
        System.out.println("异常处理器正在执行中");
        ModelAndView modelAndView = new ModelAndView();
        //定义异常现象出现后，反馈给用户查看的信息
        modelAndView.addObject("msg", "出错啦！");
        //定义异常现象出现后，反馈给用户查看的页面
        modelAndView.setViewName("error.jsp");
        return modelAndView;
    }
}

```

ModelAndView

```

package org.springframework.web.servlet;
public class ModelAndView
public void setViewName(@Nullable String viewName)      设置跳转页面，等同 return "page.jsp"    //
modelAndView.setViewName("forward:page.jsp")

```

DispatcherServlet

```

package org.springframework.web.servlet;
public class DispatcherServlet extends FrameworkServlet      Entrance for all requests.
    The core logic is all in the doDispatch method.
    invokes getHandler to obtain the Handler for processing the request based on the request, which determines
    the processing method of the Controller.
    Obtains the method information of the Controller corresponding to this request path through
    mappingRegistry. (RequestMappingInfo)
    See: invokes org.springframework.web.servlet.handler.AbstractHandlerMapping#getHandler
          invokes
          org.springframework.web.servlet.handler.AbstractHandlerMethodMapping#getHandlerInternal
          invokes
          org.springframework.web.servlet.handler.AbstractHandlerMethodMapping#lookupHandlerMet
          hod
    invokes getHandlerAdapter to obtain the HandlerAdapter based on the found handler
    The HandlerAdapter is an interface that defines how to adapt different types of handlers (e.g., controllers,
    annotated methods) to a common interface that can process the request.

```

```

protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException
doDispatch
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
}

```

```

HandlerExecutionChain mappedHandler = null;
boolean multipartRequestParsed = false;
WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

try {
    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            //A. checks if the request contains files, if so, it will return the
            MultipartHttpServletRequest instance.
            processedRequest = this.checkMultipart(request);

            multipartRequestParsed = processedRequest != request;

            //Retrieve the HandlerExecutionChain of the request from the HandlerMapping collection based
            on the request.
            //An execution chain containing instances of Handlers and multiple HandlerInterceptors, The
            Handlers determine the processing method of the Controller.
            mappedHandler = this.getHandler(processedRequest);

            if (mappedHandler == null) {
                this.noHandlerFound(processedRequest, response);
                return;
            }

            HandlerAdapter ha = this.getHandlerAdapter(mappedHandler.getHandler());
            String method = request.getMethod();
            boolean isGet = HttpMethod.GET.matches(method);
            if (isGet || HttpMethod.HEAD.matches(method)) {
                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                if ((new ServletWebRequest(request, response)).checkNotModified(lastModified) && isGet) {
                    return;
                }
            }
        }

        if (!mappedHandler.applyPreHandle(processedRequest, response)) {
            return;
        }

        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
        if (asyncManager.isConcurrentHandlingStarted()) {
            return;
        }

        this.applyDefaultViewName(processedRequest, mv);
        mappedHandler.applyPostHandle(processedRequest, response, mv);
    } catch (Exception var20) {
        dispatchException = var20;
    } catch (Throwable var21) {
        dispatchException = new ServletException("Handler dispatch failed: " + var21, var21);
    }

    this.processDispatchResult(processedRequest, response, mappedHandler, mv,
    (Exception)dispatchException);
    } catch (Exception var22) {
        this.triggerAfterCompletion(processedRequest, response, mappedHandler, var22);
    } catch (Throwable var23) {
        this.triggerAfterCompletion(processedRequest, response, mappedHandler, new
        ServletException("Handler processing failed: " + var23, var23));
    }
}

} finally {
    if (asyncManager.isConcurrentHandlingStarted()) {
        if (mappedHandler != null) {
            mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
    }
}

```

```

        }
    } else if (multipartRequestParsed) {
        this.cleanupMultipart(processedRequest);
    }

}

getHandler
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    if (this.handlerMappings != null) {
        Iterator var2 = this.handlerMappings.iterator();

        while(var2.hasNext()) {
            HandlerMapping mapping = (HandlerMapping)var2.next();
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }

    return null;
}

initLocaleResolver
private void initLocaleResolver(ApplicationContext context) { // initialize Dispatcher
    try {
        this.localeResolver = (LocaleResolver)context.getBean("localeResolver", LocaleResolver.class);
        if (this.logger.isTraceEnabled()) {
            this.logger.trace("Detected " + this.localeResolver);
        } else if (this.logger.isDebugEnabled()) {
            this.logger.debug("Detected " + this.localeResolver.getClass().getSimpleName());
        }
    } catch (NoSuchBeanDefinitionException var3) {
        this.localeResolver = (LocaleResolver)this.getDefaultStrategy(context, LocaleResolver.class);
        if (this.logger.isTraceEnabled()) {
            this.logger.trace("No LocaleResolver 'localeResolver': using default [" +
this.localeResolver.getClass().getSimpleName() + "]");
        }
    }
}

getDefaultStrategy
protected <T> T getDefaultStrategy(ApplicationContext context, Class<T> strategyInterface) {
    List<T> strategies = this.getDefaultStrategies(context, strategyInterface);
    if (strategies.size() != 1) {
        throw new BeanInitializationException("DispatcherServlet needs exactly 1 strategy for interface [" +
strategyInterface.getName() + "]");
    } else {
        return strategies.get(0);
    }
}
}

```

HandlerMapping

```

package org.springframework.web.servlet;
public interface HandlerMapping      handler 处理器映射
default boolean usesPathPatterns()
HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;

```

HandlerAdapter

```

package org.springframework.web.servlet;
public interface HandlerAdapter      handler 处理器的适配器，适配特定的 Handler 来处理相应的请求，实现对后续请求参数解析，

```

调用方法，解析返回值的等处理。

HTTP 在 SpringMVC 的整个请求流程：

- 根据请求路径获取对应的处理器（handler），里面有包含处理方法的信息等
- 根据不同的处理器，选用不同的处理适配器（HandlerAdapter）
- 用处理适配器对参数进行解析，填充
- 处理适配器调用处理方法
- 用处理适配器对返回值进行解析

```
boolean supports(Object var1);                                判断 HandlerAdapter 组件是否支持这个 handler 实例
ModelAndView handle(HttpServletRequest var1, HttpServletResponse var2, Object var3) throws Exception;      使用 handler
实例来处理具体的请求
long getLastModified(HttpServletRequest var1, Object var2);    获取资源的最后修改值
mvc
```

HttpRequestHandlerAdapter (dispatcher)

```
package org.springframework.web.servlet.mvc;
public class HttpRequestHandlerAdapter implements HandlerAdapter
{
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        ((HttpRequestHandler)handler).handleRequest(request, response);
        return null;
    }
}
```

SimpleControllerHandlerAdapter (dispatcher)

```
package org.springframework.web.servlet.mvc;
public class SimpleControllerHandlerAdapter implements HandlerAdapter
{
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        return ((Controller)handler).handleRequest(request, response);
    }
}
```

method

AbstractHandlerMethodAdapter (dispatcher)

```
package org.springframework.web.servlet.mvc.method;
public abstract class AbstractHandlerMethodAdapter          抽象 handler 处理器
    extends WebContentGenerator implements HandlerAdapter, Ordered

    public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception 调用 handlerInternal
    protected abstract ModelAndView handleInternal(HttpServletRequest request, HttpServletResponse response, HandlerMethod
handlerMethod) throws Exception; 需实现方法
```

RequestMappingInfoHandlerMapping

```
package org.springframework.web.servlet.mvc.method;
public abstract class RequestMappingInfoHandlerMapping extends AbstractHandlerMethodMapping<RequestMappingInfo>
```

annotation

RequestMappingHandlerAdapter (分发 - adapter)

```

package org.springframework.web.servlet.mvc.method.annotation;
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter implements BeanFactoryAware,
InitializingBean 支持方法上标注@RequestMapping 注解的 handler 适配器
private HandlerMethodArgumentResolverComposite argumentResolvers;
private HandlerMethodReturnValueHandlerComposite returnValueHandlers;

protected boolean supportsInternal(HandlerMethod handlerMethod) 是否支持当前 handlerMethod
protected ModelAndView handleInternal(HttpServletRequest request, HttpServletResponse response, HandlerMethod
handlerMethod) throws Exception 检测 request，并负责调用 HandlerMethod (处理器)，返回 ModelAndView
protected ModelAndView invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, HandlerMethod
handlerMethod) throws Exception 调用 handler 方法
private void initControllerAdviceCache() 初始化 ControllerAdvice 缓存

protected ModelAndView invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, HandlerMethod
handlerMethod) throws Exception
    将 request 和 response 封装一下
    获取 invocableMethod
    设置 argumentResolvers(参数解析器)
    设置 returnValueHandlers(返回值处理器)
    调用方法
    返回 ModelAndView
protected ServletInvocableHandlerMethod createInvocableHandlerMethod(HandlerMethod handlerMethod)

```

RequestMappingHandlerMapping (分发 - handler)

```

package org.springframework.web.servlet.mvc.method.annotation;
public class RequestMappingHandlerMapping extends RequestMappingInfoHandlerMapping implements
MatchableHandlerMapping, EmbeddedValueResolverAware
protected RequestMappingInfo getMappingForMethod(Method method, Class<?> handlerType) 获取请求信息
private RequestMappingInfo createRequestMappingInfo(AnnotatedElement element) 获取方法上面的
@RequestMapping 注解，没有的话就不创建了，只有打上了这个注解的才能被创建注册。
protected RequestMappingInfo createRequestMappingInfo(RequestMapping requestMapping, @Nullable
RequestCondition<?> customCondition) 创建好 Method 的 RequestMappingInfo

```

RequestBodyAdvice

```

package org.springframework.web.servlet.mvc.method.annotation;
public interface RequestBodyAdvice 可以在请求体数据被 HttpMessageConverter 转换前，后。执行一些逻辑代码。通常用来做
解密
boolean supports( 该方法用于判断当前请求，是否要执行 beforeBodyRead 方法
    MethodParameter methodParameter, handler 方法的参数对象
    Type targetType, handler 方法的参数类型
    Class<? extends HttpMessageConverter<?>> converterType 将会使用到的 Http 消息转换器类型
);
HttpInputMessage beforeBodyRead( 在 Http 消息转换器执转换，之前执行
    HttpInputMessage inputMessage, 客户端的请求数据
    MethodParameter parameter, handler 方法的参数对象
)

```

```

Type targetType,           handler 方法的参数类型
Class<? extends HttpMessageConverter<?>> converterType   将会使用到的 Http 消息转换器类类型
) throws IOException;

Object afterBodyRead(    在 Http 消息转换器执转换, 之后执行
    Object body,        转换后的对象
    HttpServletRequest inputMessage,  客户端的请求数据
    MethodParameter parameter,  handler 方法的参数对象
    Type targetType,      handler 方法的参数类型
    Class<? extends HttpMessageConverter<?>> converterType  使用的 Http 消息转换器类类型
);

Object handleEmptyBody(@Nullable Object body, HttpServletRequest inputMessage, MethodParameter parameter, Type
targetType, Class<? extends HttpMessageConverter<?>> converterType);  在 Http 消息转换器执转换, 之后执行, 处理空 body
的情况

```

ResponseBodyAdvice

```

package org.springframework.web.servlet.mvc.method.annotation;
public interface ResponseBodyAdvice<T>    其实是对加了@RestController(也就是@Controller+@ResponseBody)注解的处理
器将要返回的值进行包装处理。
boolean supports(
    MethodParameter returnType,
    Class<? extends HttpMessageConverter<?>> converterType
);
T beforeBodyWrite(
    @Nullable T body,
    MethodParameter returnType,
    MediaType selectedContentType,
    Class<? extends HttpMessageConverter<?>> selectedConverterType,
    ServerHttpRequest request,
    ServerHttpResponse response);

```

ExceptionHandlerExceptionResolver

```

package org.springframework.web.servlet.mvc.method.annotation;
public class ExceptionHandlerExceptionResolver extends AbstractHandlerMethodExceptionResolver implements
ApplicationContextAware, InitializingBean
    处理返回异常的主要实现地方, 他保存了@ControllerAdvice 的 Bean, 以及参数解析器, 返回值解析器, 同时生成
    ServletInvocableHandlerMethod。
    这样在解析异常的时候, 会调用我们@ControllerAdvice 或者 Controller 对应 Bean 的 ExceptionHandler 方法。
private void initExceptionHandlerAdviceCache()  初始化异常处理 handlerAdvice 缓存

```

ServletInvocableHandlerMethod (分发 - adapter)

```

package org.springframework.web.servlet.mvc.method.annotation;
public class ServletInvocableHandlerMethod extends InvocableHandlerMethod    对 InvocableHandlerMethod 做了增强,
实现在 InvocableHandlerMethod 的基础上, 不仅能对请求参数进行解析, 填充, 同时还支持对返回参数进行解析, 以及暴露出给外部
调用 invokeAndHandle 方法。
private HandlerMethodReturnValueHandlerComposite returnValueHandlers;  返回值处理器
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer, Object...
providedArgs) throws Exception  调用并且处理

```

第一步调用请求 (invokeForRequest)
第二步解析返回值 (handleReturnValue)

```
public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndView mavContainer, Object... providedArgs) throws Exception
```

获取解析好的参数
真正的调用 method 方法

PathVariableMapMethodArgumentResolver

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class PathVariableMapMethodArgumentResolver implements HandlerMethodArgumentResolver
```

针对路径是`test/{name}`的路径注解解析器

MatrixVariableMethodArgumentResolver

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class MatrixVariableMethodArgumentResolver extends AbstractNamedValueMethodArgumentResolver
```

MatrixVariableMapMethodArgumentResolver

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class MatrixVariableMapMethodArgumentResolver implements HandlerMethodArgumentResolver
```

ServletModelAttributeMethodProcessor

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class ServletModelAttributeMethodProcessor extends ModelAttributeMethodProcessor
```

RequestResponseBodyMethodProcessor

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class RequestResponseBodyMethodProcessor extends AbstractMessageConverterMethodProcessor
```

RequestPartMethodArgumentResolver

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class RequestPartMethodArgumentResolver extends AbstractMessageConverterMethodArgumentResolver
```

RequestAttributeMethodArgumentResolver

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class RequestAttributeMethodArgumentResolver extends AbstractNamedValueMethodArgumentResolver
```

ServletRequestMethodArgumentResolver

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class ServletRequestMethodArgumentResolver implements HandlerMethodArgumentResolver
```

ModelAndViewMethodReturnValueHandler

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class ModelAndViewMethodReturnValueHandler implements HandlerMethodReturnValueHandler
```

针对返回类型是 ModelAndView 返回值的处理器

DeferredResultMethodReturnValueHandler

```
package org.springframework.web.servlet.mvc.method.annotation;
```

```
public class DeferredResultMethodReturnValueHandler implements HandlerMethodReturnValueHandler
```

针对返回类型是 DeferredResult 返回值的处理器

RequestResponseBodyMethodProcessor

```
package org.springframework.web.servlet.mvc.method.annotation;  
public class RequestResponseBodyMethodProcessor extends AbstractMessageConverterMethodProcessor 针对返回类型  
是由`@ResponseBody`返回值的处理器
```

support

AbstractDispatcherServletInitializer

```
package org.springframework.web.servlet.support;  
public abstract class AbstractDispatcherServletInitializer 自定义实现类，基于 servlet3.0 规范，自定义 servlet 容器初始化配置类，加载 springMvc 核心配置类
```

```
public void onStartup(ServletContext servletContext) 实现父类的此方法，再自定义之后的操作  
protected FrameworkServlet createDispatcherServlet(WebApplicationContext servletAppContext) 创建 servlet 容器时，使用注解的方式加载 SpringMVC 配置类中的信息，并加载成 web 专用的 ApplicationContext 对象，  
protected abstract String[] getServletMappings(); 注解配置映射地址方式，服务于 springMvc 的核心控制器  
DispatcherServlet，一般配置为根路径 /
```

config

InterceptorRegistry

```
package org.springframework.web.servlet.config.annotation;  
public class InterceptorRegistry 拦截器注册类登记类
```

```
public InterceptorRegistration addInterceptor(HandlerInterceptor interceptor) 添加拦截器
```

InterceptorRegistration

```
package org.springframework.web.servlet.config.annotation;
```

```
public class InterceptorRegistration 拦截器注册类
```

```
public InterceptorRegistration addPathPatterns(String... patterns) 所有请求都被拦截，包括静态资源 //addPathPattern("/**")  
public InterceptorRegistration excludePathPatterns(String... patterns) 放行的请求
```

WebMvcConfigurer

```
package org.springframework.web.servlet.config.annotation;  
public interface WebMvcConfigurer 自定义 MVC 配置类
```

```
default void addInterceptors(InterceptorRegistry var1); 拦截器配置
```

```
registry.addInterceptor(new  
TestInterceptor()).addPathPatterns("/**").excludePathPatterns("/emp/toLogin", "/emp/login", "/js/**", "/css/**", "/images/**");
```

```
default void addResourceHandlers(ResourceHandlerRegistry registry) 静态资源处理，不会拦截普通请求路径（这里加  
Order，会导致所有普通请求不能访问）
```

```
registry.addResourceHandler("/sdklala/**").addResourceLocations("classpath:/META-INF/resources/"); 转发到 resources  
中（接收/sdklala/hello 同级所有资源）
```

```
registry.addResourceHandler("/sdklala/**").addResourceLocations("/img/"); 转发到 resources  
中
```

```
default void addViewControllers(ViewControllerRegistry registry) 视图跳转控制器（默认优先级 PathVariable >  
ViewControllerRegistry > ResourceHandlerRegistry）
```

```
registry.addViewController("/sdklala/swagger-ui").setViewName("forward:/swagger-ui/index.html"); //设置访问路
```

径为 "/" 跳转到指定页面

```
registry.addViewController("/sdklala/hello").setViewName("forward:/doc.html"); //路径转发到  
资源 (注意, 访问 hello 同级其他资源都会带有路径前缀/sdklala)  
registry.setOrder(Ordered.HIGHEST_PRECEDENCE); // 可以通过这个 设置优先级高于 PathVariable  
default void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer); 默认静态资源处理器  
default void configureViewResolvers(ViewResolverRegistry registry); 配置视图解析器  
default void configureContentNegotiation(ContentNegotiationConfigurer configurer); 配置内容裁决的一些选项  
default void addCorsMappings(CorsRegistry registry); 解决跨域问题
```

使用——

放行静态资源

```
public class SpringMvcConfiguration implements WebMvcConfigurer {  
    public void addResourceHandlers(ResourceHandlerRegistry registry) {  
        registry.addResourceHandler("/img/**").addResourceLocations("/img/");  
    }  
}
```

使用 Servlet 默认过滤规则

```
public class SpringMvcConfiguration implements WebMvcConfigurer {  
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {  
        configurer.enable();  
    }  
}
```

通过 InterceptorRegistry 的反射获取所有拦截器，并对所有拦截器 InterceptorRegistration 做操作 >>

```
/** * 通用拦截器排除设置, 所有拦截器都会自动加 springdoc-opapi 相关的资源排除信息, 不用在应用程序自身拦截器定义的地方去添加, 算是良心解耦实现。  
 */  
@SuppressWarnings("unchecked")  
@Override  
public void addInterceptors(InterceptorRegistry registry) {  
    try {  
        Field registrationsField = FieldUtils.getField(InterceptorRegistry.class, "registrations", true);  
        List<InterceptorRegistration> registrations = (List<InterceptorRegistration>) ReflectionUtils.getField(registrationsField, registry);  
        if (registrations != null) {  
            for (InterceptorRegistration interceptorRegistration : registrations) {  
                interceptorRegistration.excludePathPatterns("/springdoc**/**");  
            }  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

WebMvcConfigurationSupport

```
package org.springframework.web.servlet.config.annotation;  
public class WebMvcConfigurationSupport implements ApplicationContextAware, ServletContextAware 自定义 MVC 配置  
类, 功能更多  
protected void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) 添加参数处理器
```

注解

```
package org.springframework.web.servlet.config.annotation;  
@EnableWebMvc 开启 MVC 注解驱动, 等同<mvc:annotation-driven /> 不完全相同 (在 springboot 中会全面接管程序, 静态  
资源, 视图解析器会全部失效, ) 【TYPE】
```

filter

OrderedCharacterEncodingFilter

```
package org.springframework.boot.web.servlet.filter;  
public class OrderedCharacterEncodingFilter extends CharacterEncodingFilter implements OrderedFilter 优先级最高的 Filter  
handler
```

MatchableHandlerMapping

```
package org.springframework.web.servlet.handler;  
public interface MatchableHandlerMapping extends HandlerMapping
```

PathPatternMatchableHandlerMapping

```
package org.springframework.web.servlet.handler;  
class PathPatternMatchableHandlerMapping implements MatchableHandlerMapping
```

AbstractUrlHandlerMapping

```
package org.springframework.web.servlet.handler;  
public abstract class AbstractUrlHandlerMapping extends AbstractHandlerMapping implements  
MatchableHandlerMapping
```

SimpleUrlHandlerMapping

```
package org.springframework.web.servlet.handler;  
public class SimpleUrlHandlerMapping extends AbstractUrlHandlerMapping 通过 LinkedHashMap 的方式来存储 url 与  
handler 的映射关系  
private final Map<String, Object> urlMap = new LinkedHashMap();  
protected void registerHandlers(Map<String, Object> urlMap) throws BeansException  
public void initApplicationContext() throws BeansException  
public void setUrlMap(Map<String, ?> urlMap)  
public void setMappings(Properties mappings)
```

AbstractHandlerExceptionResolver

```
package org.springframework.web.servlet.handler;  
public abstract class AbstractHandlerExceptionResolver implements HandlerExceptionResolver, Ordered  
大部分的异常解  
析器的基类, 主要是针对不同的 handler 单独是否配置处理, 可以不用管
```

AbstractHandlerMethodExceptionResolver

```
package org.springframework.web.servlet.handler;  
public abstract class AbstractHandlerMethodExceptionResolver extends AbstractHandlerExceptionResolver 就顾名思义  
是对 HandlerMethod 的这种类型的 handler 处理。这个类也没干啥, 就是实现了一下 shouldApplyTo, 判断是否支持当前的 handler  
(dispatcher)
```

AbstractHandlerMapping

```
package org.springframework.web.servlet.handler;  
public abstract class AbstractHandlerMapping extends WebApplicationObjectSupport implements HandlerMapping,  
Ordered, BeanNameAware
```

public final HandlerExecutionChain **getHandler**(HttpServletRequest request) throws Exception 对于不同种类的实现方式获取出来的 handler 类型完全是不同的

```
    RequestMappingHandlerMapping = HandlerMethod
    ResourceHandlerFunction = ResourceHandlerFunction

public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    Object handler = this.getHandlerInternal(request);
    if (handler == null) {
        handler = this.getDefaultHandler();
    }

    if (handler == null) {
        return null;
    } else {
        if (handler instanceof String) {
            String handlerName = (String)handler;
            handler = this.obtainApplicationContext().getBean(handlerName);
        }

        if (!ServletRequestPathUtils.hasCachedPath(request)) {
            this.initLookupPath(request);
        }

        HandlerExecutionChain executionChain = this.getHandlerExecutionChain(handler, request);
        if (this.logger.isTraceEnabled()) {
            this.logger.trace("Mapped to " + handler);
        } else if (this.logger.isDebugEnabled() && !DispatcherType.ASYNC.equals(request.getDispatcherType())) {
            this.logger.debug("Mapped to " + executionChain.getHandler());
        }

        if (this.hasCorsConfigurationSource(handler) || CorsUtils.isPreFlightRequest(request)) {
            CorsConfiguration config = this.getCorsConfiguration(handler, request);
            if (this.getCorsConfigurationSource() != null) {
                CorsConfiguration globalConfig =
this.getCorsConfigurationSource().getCorsConfiguration(request);
                config = globalConfig != null ? globalConfig.combine(config) : config;
            }

            if (config != null) {
                config.validateAllowCredentials();
            }
        }

        executionChain = this.getCorsHandlerExecutionChain(request, executionChain, config);
    }

    return executionChain;
}
```

protected abstract Object **getHandlerInternal**(HttpServletRequest request) throws Exception

AbstractHandlerMethodMapping

```
package org.springframework.web.servlet.handler;

public abstract class AbstractHandlerMethodMapping<T> extends AbstractHandlerMapping implements InitializingBean
private HandlerMethodMappingNamingStrategy<T> namingStrategy;
private final AbstractHandlerMethodMapping<T>.MappingRegistry mappingRegistry = new
AbstractHandlerMethodMapping.MappingRegistry(); 路径映射【路径 - Controller 方法】 // /test/map -
TestController.test
```

protected HandlerMethod **getHandlerInternal**(HttpServletRequest request) throws Exception 实现方法，根据请求的路径去找

对应的 HandlerMethod 处理方法

找请求的路径，比如请求的路径/test/getInfo

根据请求的路径找 HandlerMethod

protected HandlerMethod **lookupHandlerMethod**(String lookupPath, HttpServletRequest request) throws Exception 根据路径找对应的 HandlerMethod

获取匹配上的路径的信息 (RequestMappingInfo)

将匹配上的路径信息和对应 HandleMethod 添加到 matches 里面

```
private void addMatchingMappings(Collection<T> mappings, List<AbstractHandlerMethodMapping<T>.Match> matches,  
HttpServletRequest request)    再次进行匹配筛选
```

匹配 POST 和 GET 方法是否一致

匹配要求的请求头是否一致

根据条件筛选完后，就会组装得到一个新的请求路径、请求头、请求方式等都完成符合这次请求的 RequestMappingInfo 信息

`protected void initHandlerMethods()` 把所有 Spring 的 Bean 都取出来，然后挨个去调用 `processCandidateBean`（一个`@Controller`注册到 `mappingRegistry`的过程）

1. processCandidateBean 里面判断是否有@Controller 或者@RequestMapping
 2. 将第二步检测通过的类的方法进行遍历，包括它的父类的，接口的方法
 3. 遍历方法的时候获取到@RequestMapping 上面的信息，生成 RequestMappingInfo
 4. 将类和方法的 RequestMappingInfo 合并成新的，返回给 mappingRegistry 去注册

protected void processCandidateBean(String beanName) 判断是否是匹配当去处理器的，也就是

`AbstractHandlerMethodMapping`的子类的`isHandler`是 true

protected abstract boolean **isHandler**(Class<?> beanType); 判断 bean1 是否是 handler, 这里就很明显了吧, 那些被标注了 @Controller, 或者 RequestMapping 注解的类就会通过, 进入 detectHandlerMethods, 然后被扫描, 分析。注册到 mappingRegistry

protected void detectHandlerMethods(Object handler)

1. 获取这个类的信息，这里就是一个 Controller 的 Class
 2. 获取用户自定义的 Class，有些是 cglib 的
 3. 获取有 RequestMapping 注解的方法
 4. 循环第三步的方法，注入到 mappingRegistry

protected abstract T **getMappingForMethod**(Method method, Class<?> handlerType) 这个函数最终由
RequestMappingHandlerMapping 实现了

```
static class MappingRegistration<T> {  
    public HandlerMethod getHandlerMethod()    直接返回 HandlerMethod  
}
```

SimpleServletHandlerAdapter

```
package org.springframework.web.servlet.handler;
```

```
public class SimpleServletHandlerAdapter implements HandlerAdapter
```

@Nullable

```
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception  
{
```

```
((Servlet)handler).service(request, response);
return null;
}
```

function

ResourceHandlerFunction

```
package org.springframework.web.servlet.function;
class ResourceHandlerFunction implements HandlerFunction<ServerResponse>
```

resource

ResourceHttpRequestHandler (dispatcher)

```
package org.springframework.web.servlet.resource;
public class ResourceHttpRequestHandler extends WebContentGenerator implements HttpServletRequestHandler,
EmbeddedValueResolverAware, InitializingBean, CorsConfigurationSource
```

Key Responsibilities

- Static Resource Handling
Maps incoming HTTP requests for static resources to their corresponding locations on the server.
- Caching
Supports caching mechanisms to improve performance by reducing the number of requests to the server for static resources.
- Path Resolution
Resolves resource paths and serves the corresponding resources from configurable locations (e.g., classpath, file system).

```
public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
Handles the incoming HTTP request and serves the static resource specified by the request path.
```

```
public void setLocations(List<Resource> locations)
```

Sets the locations from which resources should be served. Resources can be configured from the classpath or the file system.

method

HandlerMethod (dispatcher adapter)

```
package org.springframework.web.method;
public class HandlerMethod 这次请求对应的 Method(处理方法)
```

InvocableHandlerMethod (dispatcher adapter)

```
public class InvocableHandlerMethod extends HandlerMethod 对 HandlerMethod 做了增强，实现了对请求的参数进行解析，填充。
```

```
private static final Object[] EMPTY_ARGS = new Object[0];
private HandlerMethodArgumentResolverComposite resolvers = new HandlerMethodArgumentResolverComposite(); 参数解析器，对参数对应的参数解析器做了缓存
private ParameterNameDiscoverer parameterNameDiscoverer = new DefaultParameterNameDiscoverer();
private WebDataBinderFactory dataBinderFactory;
```

```
protected Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer, Object... providedArgs) throws Exception 把参数解析，比如我们使用@RequestBody 来接收一个参数，或者使用@RequestParam，又或者使用 Map
```

获取方法的参数列表

循环遍历参数

判断当前的解析器是否支持目前的参数

支持的话，就用解析器对它解析

protected Object doInvoke(Object... args) throws Exception 在 getMethodArgumentValues 拿到解析后的请求参数后，就会调用 doInvoke，这里就通过获取出 HandlerMethod 对应的处理方法和处理类，反射调用

获取方法

调用

support

HandlerMethodArgumentResolverComposite

package org.springframework.web.method.support;

public class HandlerMethodArgumentResolverComposite implements HandlerMethodArgumentResolver

把需要的解析器都加进来，组成一个 List，将自己变成一个组合而成的参数解析器，里面同样提供了判断是否支持该参数，以及解析参数等方法。

这样对于使用者来说，他只需要调用了参数解析器判断是否支持解析，然后再解析参数，

而对于实现参数解析器的人来说（比如我们有些情况下会需要自定义参数解析器），也只需要关注实现 HandlerMethodArgumentResolver 即可。

优化考虑点：

这里在找参数解析器的时候，找到后就会缓存起来。下次再请求同样的方法，然后同样的参数去找解析器的时候，就会快速返回了，这里的缓存思想也是我们可以借鉴的，很多地方为了性能考虑都是做了这样的缓存。

这个是一个比较好的组合的方式，通过实现 HandlerMethodArgumentResolver，然后组合其他同样实现了

HandlerMethodArgumentResolver 的真正参数解析器，最后提供一个增强后的组合参数解析器。

private final List<HandlerMethodArgumentResolver> argumentResolvers = new ArrayList(); 定义参数解析器列表

private final Map<MethodParameter, HandlerMethodArgumentResolver> argumentResolverCache = new

ConcurrentHashMap(256); 用于缓存参数解析器，Key 是 MethodParameter

private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter parameter) 找参数解析器，找到后同时会缓存起来

HandlerMethodReturnValueHandlerComposite

package org.springframework.web.method.support;

public class HandlerMethodReturnValueHandlerComposite implements HandlerMethodReturnValueHandler

private final List<HandlerMethodReturnValueHandler> returnValueHandlers = new ArrayList(); 返回值解析器

public boolean supportsReturnType(MethodParameter returnType) 判断返回值解析器是否支持当前的返回值

private HandlerMethodReturnValueHandler selectHandler(@Nullable Object value, MethodParameter returnType) 选择处理器，使用 selectHandler 来判断是否支持， handleReturnValue 来处理返回值。

public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType, ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception 对返回值进行解析

HandlerMethodReturnValueHandler

package org.springframework.web.method.support;

public interface HandlerMethodReturnValueHandler 返回值处理器，确定了 Controller 方法能返回的值的种类

boolean supportsReturnType(MethodParameter var1);

void handleReturnValue(@Nullable Object var1, MethodParameter var2, ModelAndViewContainer var3, NativeWebRequest var4) throws Exception

HandlerMethodArgumentResolver

```
package org.springframework.web.method.support;  
public interface HandlerMethodArgumentResolver
```

```
boolean supportsParameter(MethodParameter var1);
```

Whether the given method parameter is supported by this resolver

```
Object resolveArgument(MethodParameter var1, @Nullable ModelAndViewContainer var2, NativeWebRequest var3,  
@Nullable WebDataBinderFactory var4) throws Exception;
```

UserArgumentResolver

```
@Service  
public class UserArgumentResolver implements HandlerMethodArgumentResolver {  
    @Autowired  
    MiaoshaUserService userService;  
  
    public boolean supportsParameter(MethodParameter methodParameter) {          // 是否对这种参数类型  
        MethodParameter 做处理  
        Class<?> clazz = methodParameter.getParameterType();  
        return clazz==MiaoshaUser.class;      // 根据参数 对象类型判断  
        return (null != methodParameter.getParameterAnnotation(Login.class) && clazz ==  
MiaoshaUser.class);      ;      // 根据参数 上的注解判断  
    }  
    public Object resolveArgument(MethodParameter parameter, ModelAndView mavContainer,  
                                  NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws  
Exception {
```

```
        HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);  
        HttpServletResponse response = webRequest.getNativeResponse(HttpServletResponse.class);
```

```
        String paramToken = request.getParameter(MiaoshaUserService.COOKIE_NAME_TOKEN);  
        String cookieToken = getCookieValue(request, MiaoshaUserService.COOKIE_NAME_TOKEN);  
        if(StringUtils.isEmpty(cookieToken) && StringUtils.isEmpty(paramToken)) {  
            return null;  
        }  
        String token = StringUtils.isEmpty(paramToken)?cookieToken:paramToken;  
        return userService.getByToken(response, token);
```

```
    }  
    private String getCookieValue(HttpServletRequest request, String cookieName) {  
        Cookie[] cookies = request.getCookies();  
        for(Cookie cookie : cookies) {  
            if(cookie.getName().equals(cookieName)) {  
                return cookie.getValue();  
            }  
        }  
        return null;  
    }  
}
```

WebConfig

Add Mvc parameter resolver

```
@Configuration  
public class WebConfig extends WebMvcConfigurerAdapter {  
    @Autowired  
    UserArgumentResolver userArgumentResolver;  
    @Override  
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {  
        argumentResolvers.add(userArgumentResolver);  
    }  
}
```

UserController

```
@RequestMapping("/to_list")
public String list(Model model, MiaoShaUser user) {           // 参数处理器识别
    model.addAttribute("user", user);
    List<GoodsVo> goodsList = goodsService.listGoodsVo();
    model.addAttribute("goodsList", goodsList);
    return "goods_list";
}
```

ModelAndViewContainer

```
package org.springframework.web.method.support;
public class ModelAndViewContainer
```

annotation

RequestParamMethodArgumentResolver

```
package org.springframework.web.method.annotation;
public class RequestParamMethodArgumentResolver extends AbstractNamedValueMethodArgumentResolver implements
UriComponentsContributor     针对`@RequestParam`，参数是 String 或者啥类型的注解解析器
```

RequestParamMapMethodArgumentResolver

```
package org.springframework.web.method.annotation;
public class RequestParamMapMethodArgumentResolver implements HandlerMethodArgumentResolver     针对
`@RequestParam`，参数是 Map 类型的注解解析器
```

filter

CharacterEncodingFilter

```
package org.springframework.web.filter;
public class CharacterEncodingFilter      字符编码过滤器
public void setEncoding(@Nullable String encoding)  设置编码
```

DelegatingFilterProxy

```
package org.springframework.web.filter;
public class DelegatingFilterProxy extends GenericFilterBean
    Lazily get Filter that was registered as a Spring Bean.
    Delegate work to the Spring Bean.
```

```
private volatile Filter delegate;                      Delegate work to other filters.
public DelegatingFilterProxy(Filter delegate)
```

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) throws ServletException,
IOException
```

HiddenHttpMethodFilter

```
package org.springframework.web.filter;
```

public class **HiddenHttpMethodFilter**

This filter processes incoming HTTP POST requests **with a hidden _method parameter** in the request body.

It extracts the value of this parameter and **converts the request into the specified HTTP method** (e.g., PUT, DELETE).

This is particularly useful in web applications where you need to support these methods using traditional HTML forms.

OncePerRequestFilter

```
package org.springframework.web.filter;
```

```
public abstract class OncePerRequestFilter extends GenericFilterBean
```

OncePerRequestFilter is an abstract base class provided by Spring that makes sure the doFilterInternal() method **is called only once per request**.

It achieves this by internally checking whether the current request has already been filtered.

Classes extending the **OncePerRequestFilter** class:

```
org.springframework.web.filter.CorsFilter
```

```
org.springframework.web.filter.CharacterEncodingFilter
```

```
org.springframework.web.filter.HiddenHttpMethodFilter
```

```
org.springframework.web.filter.ForwardedHeaderFilter
```

```
org.springframework.web.filter.ShallowEtagHeaderFilter
```

```
org.springframework.web.filter.RequestContextFilter
```

```
protected abstract void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)  
throws ServletException, IOException;
```

multipart

MultipartFile

```
package org.springframework.web.multipart;
```

```
public interface MultipartFile          上传文件处理
```

```
extends InputStreamSource
```

```
String getName();           获取参数的名字 // photos
```

```
String getOriginalFilename();    获取文件的名字 //face53.png
```

```
String getContentType();      获取文件类型 //image/png
```

```
boolean isEmpty();          判断是否为空
```

```
long getSize();             获取文件的大小以字节为单位
```

```
byte[] getBytes();          把文件内容以字节数组的方式返回
```

```
InputStream getInputStream();  获取文件的字节流
```

```
void transferTo(File var1)     用来把 MultipartFil 类型变成 File // photo.transferTo( new File("D:\\Desktop\\new") );
```

MultipartResolver

```
package org.springframework.web.multipart;
```

```
依赖: commons-fileupload
```

```
public interface MultipartResolver      定义了文件上传过程中的相关操作，并对通用性操作进行了封装
```

```
MultipartResolver 接口底层实现类 commonsMultipartResovler
```

```
CommonsMultipartResovler 并未自主实现文件上传下载对应的功能，而是调用了 apache 的文件上传下载组件
```

```
配置: applicationContext.xml 中注册 bean 为 multipartResolver
```

```

<form action="/fileupload" method="post" enctype="multipart/form-data">
    上传LOGO :<input type="file" name="file"/><br/>
    <input type="submit" value="上传"/>
</form>

```

流数据



tomcat

- 创建 DiskFileItemFactory
- 创建 ServletFileUpload
- 解析请求 parseRequest(request);
- 遍历 FileItem 集合
- 获取文件类别 isFormField
- 写文件
- 删除临时文件

client

RestTemplate

```

package org.springframework.web.client;
public class RestTemplate extends InterceptorHttpAccessor implements RestOperations
    Timeout
        The RestTemplate class itself does not directly expose timeout parameters.
        Instead, timeouts are configured through the underlying HTTP client used by RestTemplate.

```

org.springframework.http.client.ClientHttpRequestFactory 请求工厂

restTemplate 请求，如果接口返回的不是 200 状态，则会抛出异常报错。

new RestTemplate()默认支持如下类型的 http 消息转换器

转换器名称	支持 MediaType	字符集
ByteArrayHttpMessageConverte000r null	application/octet-stream 和 */*	
StringHttpMessageConverter ISO-8859-1	text/plain	和 */*
ResourceHttpMessageConverter null	/*	
SourceHttpMessageConverter null	application/xml 和 text/xml 和 application/*+xml	
AllEncompassingFormHttpMessageConverter multipart/mixed UTF-8	application/x-www-form-urlencoded 和 multipart/form-data 和	

Jaxb2RootElementHttpMessageConverter application/xml 和 text/xml 和 application/*+xml
null

MappingJackson2HttpMessageConverter application/json 和 application/*+json
null

```
public RestTemplate(ClientHttpRequestFactory requestFactory)
public void setRequestFactory(ClientHttpRequestFactory requestFactory)    设置请求工厂
public <T> ResponseEntity<T> exchange(String url, HttpMethod method, @Nullable HttpEntity<?> requestEntity, Class<T>
responseType, Object... uriVariables)    远程请求
    url        请求路径
    method     请求的方法 (GET、POST、PUT 等)
    requestEntity  HttpEntity 对象，封装了请求头和请求体
    responseType   返回数据类型
    uriVariables  支持 PathVariable 类型的数据。
public <T> T getForObject(String url, Class<T> responseType, Object... uriVariables) throws RestClientException
    // String resop = restTemplate.getForObject("http://localhost:6666/gettest/?id=1", String.class);
```

ResponseErrorHandler

```
package org.springframework.web.client;
public interface ResponseErrorHandler    针对 RestTemplate 请求的 异常处理器
boolean hasError(ClientHttpResponse response) throws IOException;    判断 HttpResponse 是否是异常响应 (通过状态码)
void handleError(ClientHttpResponse response) throws IOException;    处理异常响应结果 (非 200 状态码)
```

DefaultResponseErrorHandler

```
package org.springframework.web.client;    可能抛出 UnknownHttpStatusCodeException    HttpClientErrorException
HttpServerErrorException
public class DefaultResponseErrorHandler implements ResponseErrorHandler    默认异常处理器
boolean hasError(ClientHttpResponse response) throws IOException;    判断 HttpResponse 是否是异常响应 (通过状态码)
void handleError(ClientHttpResponse response) throws IOException;    处理异常响应结果 (非 200 状态码)
```

UnknownHttpStatusCodeException

```
package org.springframework.web.client;
public class UnknownHttpStatusCodeException extends RestClientResponseException    未知状态码异常
```

HttpClientErrorException

```
package org.springframework.web.client;
public class HttpClientErrorException extends HttpStatusCodeException    请求状态码异常, 4xx 状态码
```

HttpServerErrorException

```
package org.springframework.web.client;
public class HttpServerErrorException extends HttpStatusCodeException    请求状态码异常, 5xx 状态码
```

HttpStatusCodeException

```
package org.springframework.web.client;
public abstract class HttpStatusCodeException extends RestClientResponseException    请求状态码异常, 请求不是 200, 则会抛
出异常报错
public HttpStatus getStatuscode()    获取响应状态码
```

RestClientResponseException

```
package org.springframework.web.client;
```

```
public class RestClientResponseException extends RestClientException 请求客户端异常
public String getResponseBodyAsString() 获取响应体
```

util

UriBuilder

```
package org.springframework.web.util;
public interface UriBuilder
URI build(Object... uriVariables);
UriBuilder host(@Nullable String host);
UriBuilder port(@Nullable String port);
UriBuilder path(String path);
```

UriComponents

```
package org.springframework.web.util;
public abstract class UriComponents implements Serializable 路径组件
public final UriComponents encode()
```

OpaqueUriComponents

```
package org.springframework.web.util;
final class OpaqueUriComponents extends UriComponents
```

HierarchicalUriComponents

```
package org.springframework.web.util;
final class HierarchicalUriComponents extends UriComponents
```

```
interface PathComponent extends Serializable
String getPath();
List<String> getPathSegments();
PathComponent encode(BiFunction<String, Type, String> encoder);
void verify();
PathComponent expand(UriComponents.UriTemplateVariables uriVariables, @Nullable UnaryOperator<String> encoder);
void copyToUriComponentsBuilder(UriComponentsBuilder builder);

static final class PathComponentComposite implements PathComponent
private final List<PathComponent> pathComponents;
public PathComponentComposite(List<PathComponent> pathComponents)
static final class PathSegmentComponent implements PathComponent
private final List<String> pathSegments;
public PathSegmentComponent(List<String> pathSegments)
static final class FullPathComponent implements PathComponent
private final String path;
public FullPathComponent(@Nullable String path)
```

UriComponentsBuilder

```
package org.springframework.web.util;
public class UriComponentsBuilder implements UriBuilder, Cloneable
```

```

public UriComponents build()

protected UriComponentsBuilder(UriComponentsBuilder other)
public static UriComponentsBuilder fromUriString(String uri)
    //      UriComponentsBuilder.fromUriString("http://mydomain/api/getToken")
    // "123").queryParam("appsecret", "secret123") .build().encode().toString();
public UriComponentsBuilder queryParam(String name, Object... values)

private interface PathComponentBuilder
    HierarchicalUriComponents.PathComponent build();
    PathComponentBuilder cloneBuilder();
private static class CompositePathComponentBuilder implements PathComponentBuilder
    private final Deque<PathComponentBuilder> builders = new ArrayDeque();
private static class PathSegmentPathComponentBuilder implements PathComponentBuilder
    private final List<String> pathSegments = new ArrayList();
private static class FullPathComponentBuilder implements PathComponentBuilder
    private final StringBuilder path = new StringBuilder();

```

SpringBoot / org.springframework.boot

org.springframework.boot.SpringApplication

org.springframework.boot.autoconfigure.SpringBootApplication

org.springframework.boot.context.event.SpringApplicationEvent
 org.springframework.boot.context.properties.ConfigurationProperties

org.springframework.boot.web.servlet.support.SpringBootServletInitializer

ApplicationArguments

```

package org.springframework.boot;
public interface ApplicationArguments
String[] getSourceArgs();
Set<String> getOptionNames();
boolean containsOption(String name);
List<String> getOptionValues(String name);
List<String> getNonOptionArgs();

```

ApplicationRunner

```

package org.springframework.boot;
public interface ApplicationRunner

```

ApplicationRunner is an interface in Spring Boot that functions similarly to CommandLineRunner, but provides a slightly more structured approach to accessing the application arguments.

Instead of passing a simple String array, ApplicationRunner uses an ApplicationArguments object, which provides convenience methods for working with command-line arguments.

Key Responsibilities

- Post-Initialization Logic

Allows execution of custom logic after the application context has been initialized.
- Access Command-Line Arguments

Provides a more structured way to access command-line arguments through the ApplicationArguments object.

- Integration with Spring Boot Lifecycle
Ensures that the run method is called **after all beans are fully initialized**.

`void run(ApplicationArguments args) throws Exception`

The primary method that must be implemented to provide the logic to be executed **after the application context is initialized**.

The method receives an ApplicationArguments object that encapsulates the command-line arguments.

BootstrapRegistry

```
package org.springframework.boot;
public interface BootstrapRegistry          启动注册器
```

BootstrapRegistryInitializer

```
package org.springframework.boot;
public interface BootstrapRegistryInitializer  启动注册初始化器 (函数式编程)
```

CommandLineRunner

```
package org.springframework.boot;
public interface CommandLineRunner
```

CommandLineRunner is an interface in Spring Boot that is used to execute code **after the Spring application context is fully initialized**.

It is often used to run some specific logic **once the application has started**, such as loading initial data, performing cleanup tasks, or running scheduled jobs.

Key Responsibilities

- Post-Initialization Logic
Allows execution of custom logic **after the application context has been initialized**.
- Access Command-Line Arguments
Provides access to command-line arguments passed to the application.
- Integration with Spring Boot Lifecycle
Seamlessly integrates with the Spring Boot lifecycle, ensuring that all beans are fully initialized before the run method is called.

`void run(String... args) throws Exception`

DefaultApplicationArguments

```
package org.springframework.boot;
public class DefaultApplicationArguments implements ApplicationArguments
```

private static class Source extends SimpleCommandLinePropertySource

DefaultBootstrapContext

```
package org.springframework.boot;
public class DefaultBootstrapContext implements ConfigurableBootstrapContext  默认启动容器
```

SpringApplication

```
package org.springframework.boot;
public class SpringApplication
```

Configuration Loading Order

1) Command Line Arguments

Properties specified as command line arguments when starting the application.

These have the highest precedence and can override properties defined elsewhere.

`java -jar myapp.jar --server.port=8090`

2) JNDI attributes from `java:comp/env`.

3) Java System Properties

Properties defined using the `-D` flag when starting the JVM. For example, `-Dserver.port=8081`.

```
java -Dserver.port=8091 -jar myapp.jar
```

4) OS Environment Variables

Environment variables set at the operating system level.

```
export SERVER_PORT=8092
```

5) A `RandomValuePropertySource` that only has properties in `random.*`.

6) Application Properties Files:

- `file:./config/` A `/config` subdir of the current directory.

- `file:./` The current directory

- `classpath:/config/` A classpath `/config` package

- `classpath:/` The classpath root

Profile-specific configuration files

Profile-specific configuration files such as `application-{profile}.properties` or `application-{profile}.yml`

which allow different configurations for different profiles

(e.g., `application-dev.properties` for the "dev" profile).

current directory

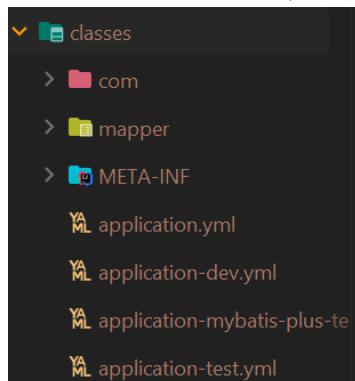
The "current directory" refers to the directory **from which the Java command** (e.g., `java -jar`) **is**

executed when you start your Spring Boot application.

Classpath

Classpath is a parameter used by the JVM to locate classes and resources.

It can include directories, JAR files, or ZIP files where the JVM will search for classes and resources.



7) `@PropertySource` Annotations

Properties defined using the `@PropertySource` annotation in Spring `@Configuration` classes.

```
@Configuration  
 @PropertySource("classpath:config.properties")  
 public class AppConfig {  
  
     @Value("${app.name}")  
     private String appName;  
  
     @Value("${app.version}")  
     private String appVersion;  
  
     // Getters to access the properties  
     public String getAppName() {  
         return appName;  
     }  
  
     public String getAppVersion() {  
         return appVersion;  
     }  
 }
```

- 8) Default properties (specified using `SpringApplication.setDefaultProperties`).

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(MyApplication.class);
        Map<String, Object> defaultProperties = new HashMap<>();
        defaultProperties.put("server.port", 8089);
        app.setDefaultProperties(defaultProperties);
        app.run(args);
    }
}
```

- 9) Custom Configurations

Any additional configuration sources that are registered programmatically by the developer using Spring's `PropertySources` API.

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(MyApplication.class);
        app.addListeners((ApplicationListener<ApplicationEnvironmentPreparedEvent>) event
        -> {
            MutablePropertySources propertySources =
            event.getEnvironment().getPropertySources();
            Map<String, Object> customProperties = new HashMap<>();
            customProperties.put("server.port", 8089);
            propertySources.addFirst(new MapPropertySource("customProperties",
            customProperties));
        });
        app.run(args);
    }
}
```

`public void setAdditionalProfiles(String... profiles)`

SpringApplication

```
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources)
{
    this.sources = new LinkedHashSet();
    this.bannerMode = Mode.CONSOLE;
    this.logStartupInfo = true;
    this.addCommandLineProperties = true;
    this.addConversionService = true;
    this.headless = true;
    this.registerShutdownHook = true;
    this.additionalProfiles = Collections.emptySet();
    this.isCustomEnvironment = false;
    this.lazyInitialization = false;
    this.applicationContextFactory = ApplicationContextFactory.DEFAULT;
    this.applicationStartup = ApplicationStartup.DEFAULT;
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));

    //A. deduce web application type from class path
    this.webApplicationType = WebApplicationType.deduceFromClasspath();

    //A. set bootstrap registry initializers
    this.bootstrapRegistryInitializers = new
    ArrayList(this.getSpringFactoriesInstances(BootstrapRegistryInitializer.class));
    //A. set initializers from spring.factories
    this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class));
    //A. set listeners from spring.factories
    this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));

    //A. deduce the main application class.
    this.mainApplicationClass = this.deduceMainApplicationClass();
}
```

```

getSpringFactoriesInstances
private <T> List<T> getSpringFactoriesInstances(Class<T> type) {
    return this.getSpringFactoriesInstances(type, (SpringFactoriesLoader.ArgumentResolver)null);
}

getSpringFactoriesInstances
private <T> List<T> getSpringFactoriesInstances(Class<T> type, SpringFactoriesLoader.ArgumentResolver
argumentResolver) {
    return SpringFactoriesLoader.forNameDefaultResourceLocation(this.getClassLoader()).load(type,
argumentResolver);
}

setInitializers
public void setInitializers(Collection<? extends ApplicationContextInitializer<?>> initializers) {
    this.initializers = new ArrayList(initializers);
}

setListeners
public void setListeners(Collection<? extends ApplicationListener<?>> listeners) {
    this.listeners = new ArrayList(listeners);
}

run
public ConfigurableApplicationContext run(String... args)
{
    long startTime = System.nanoTime();
    //A. Creates a new DefaultBootstrapContext which is used during the bootstrap phase of the
    application.
    DefaultBootstrapContext bootstrapContext = this.createBootstrapContext();
    ConfigurableApplicationContext context = null;
    //A. Configures the headless property, typically to avoid the need for a graphical environment.
    this.configureHeadlessProperty();
    //A. Retrieves the SpringApplicationRunListeners which will listen to the application startup events
    from spring.factories.
    SpringApplicationRunListeners listeners = this.getRunListeners(args);
    //A. Notifies the listeners that the application is starting.
    listeners.starting(bootstrapContext, this.mainApplicationClass);

    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
        //B. Prepares the environment for the application by setting up profiles, properties, and
        listeners.
        ConfigurableEnvironment environment = this.prepareEnvironment(listeners, bootstrapContext,
        applicationArguments);
        Banner printedBanner = this.printBanner(environment);
        //B. Creates the ConfigurableApplicationContext instance for the application (IOC Container).
        context = this.createApplicationContext();
        //B. Sets the application startup metrics.
        context.setApplicationStartup(this.applicationStartup);
        //B. Prepares the application context by setting up the environment, application arguments, and
        other necessary properties.
        (The IOC Container has been created, but the beans have not been loaded yet.)
        During this preparation phase, Spring Boot processes auto-configuration classes that are identified by
        the @EnableAutoConfiguration or @SpringBootApplication annotations.
        this.prepareContext(bootstrapContext, context, environment, listeners, applicationArguments,
        printedBanner);
        //B. Refreshes the application context, initializing all singleton beans and setting up the
        lifecycle.
        this.refreshContext(context);
    }
}

```

```

        this.afterRefresh(context, applicationArguments);
        Duration timeTakenToStartup = Duration.ofNanos(System.nanoTime() - startTime);
        if (this.logStartupInfo) {
            (new StartupInfoLogger(this.mainApplicationClass)).logStarted(this.getApplicationLog(),
timeTakenToStartup);
        }
        //B. Notifies the listeners that the application has started.
        listeners.started(context, timeTakenToStartup);
        //B. Calls all CommandLineRunner and ApplicationRunner beans.
        this.callRunners(context, applicationArguments);
    } catch (Throwable var12) {
        if (var12 instanceof AbandonedRunException) {
            throw var12;
        }
        //B. Handles the failure by notifying listeners and cleaning up resources.
        this.handleRunFailure(context, var12, listeners);
        throw new IllegalStateException(var12);
    }

    try {
        if (context.isRunning()) {
            Duration timeTakenToReady = Duration.ofNanos(System.nanoTime() - startTime);
            listeners.ready(context, timeTakenToReady);
        }
        return context;
    } catch (Throwable var11) {
        if (var11 instanceof AbandonedRunException) {
            throw var11;
        } else {
            this.handleRunFailure(context, var11, (SpringApplicationRunListeners)null);
            throw new IllegalStateException(var11);
        }
    }
}
}

refresh
protected void refresh(ConfigurableApplicationContext applicationContext) {
    applicationContext.refresh();
}
createApplicationContext
protected ConfigurableApplicationContext createApplicationContext() {
    return this.applicationContextFactory.create(this.webApplicationType);
}
prepareEnvironment
private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners,
    DefaultBootstrapContext bootstrapContext, ApplicationArguments applicationArguments) {
    // Create and configure the environment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    ConfigurationPropertySources.attach(environment);
    listeners.environmentPrepared(bootstrapContext, environment);
    DefaultPropertiesPropertySource.moveToEnd(environment);
    Assert.state(!environment.containsProperty("spring.main.environment-prefix"),
        "Environment prefix cannot be set via properties.");
    bindToSpringApplication(environment);
    if (!this.isCustomEnvironment) {
        EnvironmentConverter environmentConverter = new EnvironmentConverter(getClassLoader());
        environment = environmentConverter.convertEnvironmentIfNecessary(environment,
deduceEnvironmentClass());
    }
    ConfigurationPropertySources.attach(environment);
    return environment;
}

```

```
private Class<?> deduceMainApplicationClass() 推测 main 函数所在类（新建了一个运行异常对象，通过这个对象获取当前的调用函数堆栈数组 StackTrace，之后遍历这个堆栈数组元素的成员方法名，找到方法名为 main 的类，返回这个类）
private List<BootstrapRegistryInitializer> getBootstrapRegistryInitializersFromSpringFactories() 从 spring.factories 里实例化并 获取所有 启动初始化器
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, Object... args) 从 spring.factories 里获取指定类型的全类名，并通过反射创建实例。
private <T> List<T> createSpringFactoriesInstances(Class<T> type, Class<?>[] parameterTypes, ClassLoader classLoader, Object[] args, Set<String> names) 根据 names 中的 value 值使用反射创建相应的实例
```

```
private SpringApplicationRunListeners getRunListeners(String[] args) 从 spring.factories 里实例化 SpringApplicationRunListener 的实现类，并放入此容器
```

```
private DefaultBootstrapContext createBootstrapContext() 用 DefaultBootstrapContext 默认启动上下文 初始化所有启动上下文初始化器
```

SpringApplicationRunListeners

```
package org.springframework.boot;
class SpringApplicationRunListeners 启动监听器 列表 操作对象
SpringApplicationRunListeners(Log log, Collection<? extends SpringApplicationRunListener> listeners, ApplicationStartup applicationStartup)
void starting(ConfigurableBootstrapContext bootstrapContext, Class<?> mainApplicationClass)
```

SpringApplicationRunListener

```
package org.springframework.boot;
public interface SpringApplicationRunListener      SpringApplication 监听器
```

```
void starting(); 第一次启动 run 方法时立即调用，早期初始化。
```

```
void environmentPrepared(ConfigurableEnvironment environment) 在 environment 准备就绪后就会调用，在 ApplicationContext 创建前
void contextPrepared(ConfigurableApplicationContext context) 在 ApplicationContext 创建并且被准备后调用，ApplicationContextInitializer 执行后，但是在加载 sources 前
void contextLoaded(ConfigurableApplicationContext context) 在 ApplicationContext 装载后调用，但是在它 refresh 前
void started(ConfigurableApplicationContext context) 在 ApplicationContext 已经 refresh，并且应用已经启动
后调用，但是 CommandLineRunner 和 ApplicationRunner 没有被调用
void running(ConfigurableApplicationContext context) 在 run 方法执行完毕立即调用，并且 ApplicationContext 已经 refresh，CommandLineRunner 和 ApplicationRunner 已经被调用
void failed(ConfigurableApplicationContext context, Throwable exception) 当运行应用程序时发生错误时调用
```

SpringBootVersion

```
package org.springframework.boot;
public final class SpringBootVersion      springboot 版本信息类
```

```
public static String getVersion() 获取版本号
```

WebApplicationType

```
package org.springframework.boot;
```

```

public enum WebApplicationType {
    NONE,           The application should not run as a web application and should not start an embedded web server.
    SERVLET,        The application should run as a servlet-based web application and should start an embedded
    servlet web server.                                server.
    REACTIVE;      The application should run as a reactive web application and should start an embedded reactive web
    server.

```

```

private static final String[] SERVLET_INDICATOR_CLASSES = { "jakarta.servlet.Servlet",
    "org.springframework.web.context.ConfigurableWebApplicationContext" };
private static final String WEBMVC_INDICATOR_CLASS = "org.springframework.web.servlet.DispatcherServlet";
private static final String WEBFLUX_INDICATOR_CLASS = "org.springframework.web.reactive.DispatcherHandler";
private static final String JERSEY_INDICATOR_CLASS = "org.glassfish.jersey.servlet.ServletContainer";

```

```

deduceFromClasspath
static WebApplicationType deduceFromClasspath() {
    if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, null) && !ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS, null)
        && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, null)) {
        return WebApplicationType.REACTIVE;
    }
    for (String className : SERVLET_INDICATOR_CLASSES) {
        if (!ClassUtils.isPresent(className, null)) {
            return WebApplicationType.NONE;
        }
    }
    return WebApplicationType.SERVLET;
}

```

actuate

system

```

package org.springframework.boot.actuate.system;
public class DiskSpaceHealthIndicator extends AbstractHealthIndicator 磁盘健康检查

```

jdbc

```

package org.springframework.boot.actuate.jdbc;
public class DataSourceHealthIndicator extends AbstractHealthIndicator implements InitializingBean

```

admin

SpringApplicationAdminMXBean

```

package org.springframework.boot.admin;
public interface SpringApplicationAdminMXBean

```

SpringApplicationAdminMXBeanRegistrar

```

package org.springframework.boot.admin;
public class SpringApplicationAdminMXBeanRegistrar implements ApplicationContextAware, GenericApplicationListener,
    EnvironmentAware, InitializingBean, DisposableBean

```

Ansi8BitColor

```
package org.springframework.boot.ansi;
public final class Ansi8BitColor implements AnsiElement
```

AnsiBackground

```
package org.springframework.boot.ansi;
public enum AnsiBackground implements AnsiElement
```

AnsiColor

```
package org.springframework.boot.ansi;
public enum AnsiColor implements AnsiElement
```

AnsiColors

```
package org.springframework.boot.ansi;
public final class AnsiColors
```

AnsiElement

```
package org.springframework.boot.ansi;
public interface AnsiElement
```

AnsiOutput

```
package org.springframework.boot.ansi;
public abstract class AnsiOutput
```

AnsiPropertySource

```
package org.springframework.boot.ansi;
public class AnsiPropertySource extends PropertySource<AnsiElement>
```

AnsiStyle

```
package org.springframework.boot.ansi;
public enum AnsiStyle implements AnsiElement
```

autoconfigure

AutoConfigurationImportSelector

```
package org.springframework.boot.autoconfigure;
```

```
public class AutoConfigurationImportSelector
```

```
implements DeferredImportSelector, BeanClassLoaderAware, ResourceLoaderAware, BeanFactoryAware,
EnvironmentAware, Ordered
```

The `AutoConfigurationImportSelector` class in Spring Boot **dynamically selects and imports** necessary auto-configuration classes based on the application's environment, classpath, and specific conditions.

It's responsible for configuring and customizing the setup of your Spring Boot application **by automatically including relevant configurations** needed for various components like web servers, databases, and more.

selectImports

```
public String[] selectImports(AnnotationMetadata annotationMetadata)
```

This method is overridden to determine **which auto-configuration classes to import** based on the `AnnotationMetadata` provided.

It uses conditions such as `@ConditionalOnClass`, `@ConditionalOnBean`, `@ConditionalOnProperty`, etc., to evaluate and select appropriate configurations.

```
{  
    if (!isEnabled(annotationMetadata)) {
```

```

        return NO_IMPORTS;
    }
    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}
getAutoConfigurationEntry
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata)
    Responsible for retrieving the AutoConfigurationEntry object that encapsulates the list of auto-configuration classes to
    be imported and their conditions.
{
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations);
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}

getCandidateConfigurations
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes
attributes)
    Returns a list of candidate configurations that match the specified criteria, typically based on annotations and
    conditions.
{
    List<String> configurations = ImportCandidates.load(AutoConfiguration.class, getBeanClassLoader())
        .getCandidates();
    Assert.notEmpty(configurations,
        "No auto configuration classes found in "
        + "META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports. If
you "
        + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}

```

condition

(annotation)

ConditionalOnBean

```

package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnBeanCondition.class})
public @interface ConditionalOnBean      存在这个 Bean 组件时触发    【TYPE, METHOD】
String[] name() default {};      存在 tom 组件时, 使方法或整个类生效 // "tom"

```

ConditionalOnMissingBean

```

package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})

```

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnBeanCondition.class})
public @interface ConditionalOnMissingBean      不存在这个 Bean 组件时触发
Class<?>[] value() default {};    存在 tom 组件时，使方法或整个类生效 // Test.class
```

ConditionalOnClass

```
package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnClassCondition.class})
public @interface ConditionalOnClass      All specific classes must exist
```

ConditionalOnMissingClass

```
package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnClassCondition.class})
public @interface ConditionalOnMissingClass    不存在这个 Class
```

ConditionalOnWebApplication

```
package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnWebApplicationCondition.class})
public @interface ConditionalOnWebApplication
```

ConditionalOnNotWebApplication

```
package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnWebApplicationCondition.class})
public @interface ConditionalOnNotWebApplication  当前应用不是 Web 应用时触发
```

ConditionalOnSingleCandidate

```
package org.springframework.boot.autoconfigure.condition;
```

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnBeanCondition.class})
public @interface ConditionalOnSingleCandidate 容器中指定的组件只有一个实例 或 多个实例中有@primary 标注的主实例
```

ConditionalOnProperty

```
package org.springframework.boot.autoconfigure.condition;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Documented
@Conditional({OnPropertyCondition.class})
public @interface ConditionalOnProperty 配置文件中配置了指定属性
    name = "enabled"          匹配的属性名 (直接配置成 aftersale.swagger.enabled 时, 不存在这个属性会报错)
    prefix = "aftersale.swagger" 前缀
    havingValue = "true"       值为 true 生效
    matchIfMissing=true       即使我们配置文件中不配置 aftersale.swagger.enabled=true, 也是默认生效的

String[] value() default {};
String prefix() default "";      前缀 // "aftersale.swagger"
String[] name() default {};     匹配的属性名 (直接配置成 aftersale.swagger.enabled 时, 不存在这个属性会报错) //
"enabled"
String havingValue() default ""; 值为 true 生效 // "true"
boolean matchIfMissing() default false; 即使我们配置文件中不配置 aftersale.swagger.enabled=true, 也是默认生效的
```

ConditionalOnResource

```
package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnResourceCondition.class})
public @interface ConditionalOnResource 项目类路径中存在某一资源时触发
```

ConditionalOnJava

```
package org.springframework.boot.autoconfigure.condition;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional({OnJavaCondition.class})
public @interface ConditionalOnJava 是指定的 java 版本号时触发
```

使用————

```
@ConditionalOnMissingBean
```

```
@Bean
public MyService myService() { // 没有 MyService 这个 bean 时会注册一个 bean
...
}
```

jackson

Jackson2ObjectMapperBuilderCustomizer

```
package org.springframework.boot.autoconfigure.jackson;
@FunctionalInterface
public interface Jackson2ObjectMapperBuilderCustomizer

    org.springframework.http.converter.json.Jackson2ObjectMapperBuilder

void customize(Jackson2ObjectMapperBuilder jacksonObjectMapperBuilder);
```

Usage

```
import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.JsonSerializer;
import com.fasterxml.jackson.databind.SerializerProvider;
import com.fasterxml.jackson.databind.ser.impl.SimpleFilterProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.autoconfigure.jackson.Jackson2ObjectMapperBuilderCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.io.IOException;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;

@Configuration
public class JacksonConfiguration {
    @Value("${spring.jackson.date-format}")
    private String dateFormat;

    @Bean
    public Jackson2ObjectMapperBuilderCustomizer jacksonCustomizer() {
        return builder -> {
            builder.filters(new SimpleFilterProvider().setFailOnUnknownId(false));
            builder.featuresToEnable(JsonParser.Feature.ALLOW_UNQUOTED_CONTROL_CHARS);
            builder.serializerByType(ZonedDateTime.class, new JsonSerializer<ZonedDateTime>() {
                @Override
                public void serialize(ZonedDateTime zonedDateTime, JsonGenerator generator,
                        SerializerProvider provider)
                        throws IOException {
                    generator.writeString(DateTimeFormatter.ofPattern(dateFormat).format(zonedDateTime));
                }
            });
        };
    }
}
```

jdbc

DataSourceAutoConfiguration

```
package org.springframework.boot.autoconfigure.jdbc;
```

```

@AutoConfiguration(
    before = {SqlInitializationAutoConfiguration.class}
)
@ConditionalOnClass({DataSource.class, EmbeddedDatabaseType.class})
@ConditionalOnMissingBean(
    type = {"io.r2dbc.spi.ConnectionFactory"}
)
@EnableConfigurationProperties({DataSourceProperties.class})
@Import({DataSourcePoolMetadataProvidersConfiguration.class})
public class DataSourceAutoConfiguration

@Configuration(
    proxyBeanMethods = false
)
@Conditional({PooledDataSourceCondition.class})
@ConditionalOnMissingBean({DataSource.class, XADatasource.class})
@Import({DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
    DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.OracleUcp.class, DataSourceConfiguration.Generic.class,
    DataSourceJmxConfiguration.class})
protected static class PooledDataSourceConfiguration {
    protected PooledDataSourceConfiguration() {
    }
}

```

DataSourceTransactionManagerAutoConfiguration

```

package org.springframework.boot.autoconfigure.jdbc;

@AutoConfiguration(
    before = {TransactionAutoConfiguration.class}
)
@ConditionalOnClass({JdbcTemplate.class, TransactionManager.class})
@AutoConfigureOrder(Integer.MAX_VALUE)
@EnableConfigurationProperties({DataSourceProperties.class})
public class DataSourceTransactionManagerAutoConfiguration

JdbcTransactionManagerConfiguration
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnSingleCandidate(DataSource.class)
static class JdbcTransactionManagerConfiguration {
    JdbcTransactionManagerConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean({TransactionManager.class})
    DataSourceTransactionManager transactionManager(Environment environment, DataSource dataSource,
        ObjectProvider<TransactionManagerCustomizers> transactionManagerCustomizers) {
        DataSourceTransactionManager transactionManager = this.createTransactionManager(environment,
            dataSource);
        transactionManagerCustomizers.ifAvailable((customizers) -> {
            customizers.customize(transactionManager);
        });
    }
}

```

```

        });
        return transactionManager;
    }

    private DataSourceTransactionManager createTransactionManager(Environment environment, DataSource dataSource) {
        return
        (DataSourceTransactionManager)((Boolean)environment.getProperty("spring.dao.exceptiontranslation.enabled",
        Boolean.class, Boolean.TRUE) ? new JdbcTransactionManager(dataSource) : new
        DataSourceTransactionManager(dataSource));
    }
}

```

JdbcTemplateAutoConfiguration

```

package org.springframework.boot.autoconfigure.jdbc;

@AutoConfiguration(
    after = {DataSourceAutoConfiguration.class}
)
@ConditionalOnClass({DataSource.class, JdbcTemplate.class})
@ConditionalOnSingleCandidate(DataSource.class)
@EnableConfigurationProperties({JdbcProperties.class})
@Import({DatabaseInitializationConfigurer.class, JdbcTemplateConfiguration.class,
NamedParameterJdbcTemplateConfiguration.class})
public class JdbcTemplateAutoConfiguration

```

JndiDataSourceAutoConfiguration

```

package org.springframework.boot.autoconfigure.jdbc;

@AutoConfiguration(
    before = {XADataSourceAutoConfiguration.class, DataSourceAutoConfiguration.class}
)
@ConditionalOnClass({DataSource.class, EmbeddedDatabaseType.class})
@ConditionalOnProperty(
    prefix = "spring.datasource",
    name = {"jndi-name"})
@EnableConfigurationProperties({DataSourceProperties.class})
public class JndiDataSourceAutoConfiguration

```

XADatasourceAutoConfiguration

```

package org.springframework.boot.autoconfigure.jdbc;

@AutoConfiguration(
    before = {DataSourceAutoConfiguration.class}
)
@EnableConfigurationProperties({DataSourceProperties.class})
@ConditionalOnClass({DataSource.class, TransactionManager.class, EmbeddedDatabaseType.class})
@ConditionalOnBean({XADataSourceWrapper.class})
@ConditionalOnMissingBean({DataSource.class})

```

```
public class XADatasourceAutoConfiguration implements BeanClassLoaderAware     Distributed transaction
autoconfiguration.
```

DataSourceProperties

```
package org.springframework.boot.autoconfigure.jdbc;
@ConfigurationProperties(
    prefix = "spring.datasource")
public class DataSourceProperties implements BeanClassLoaderAware, InitializingBean

private String url;                                set database connection url. (required)
private String username;                            set database connection username. (required)
private String password;                            set database connection password. (required)

private Class<? extends DataSource> type;        set connection pool type, default automatic acquisition. (optional)  //
com.zaxxer.hikari.HikariDataSource
private String driverClassName;                    set database driver class name, default automatic acquisition. (optional)  //
com.mysql.cj.jdbc.Driver

private ClassLoader classLoader;
private boolean generateUniqueName = true;         generate unique datasource name.
private String name;                             set datasource name, default automatic generation.

private String jndiName;
private EmbeddedDatabaseConnection embeddedDatabaseConnection;
private Xa xa = new Xa();
private String uniqueName;

initializeDataSourceBuilder
public DataSourceBuilder<?> initializeDataSourceBuilder() {
    return
    DataSourceBuilder.create(this.getClassLoader()).type(this.getType()).driverClassName(this.determineDriverClas
sName()).url(this.determineUrl()).username(this.determineUsername()).password(this.determinePassword());
}
```

determineUrl

```
public String determineUrl() {
    if (StringUtils.hasText(this.url)) {
        return this.url;
    } else {
        String databaseName = this.determineDatabaseName();
        String url = databaseName != null ? this.embeddedDatabaseConnection.getUrl(databaseName) : null;
        if (!StringUtils.hasText(url)) {
            throw new DataSourceBeanCreationException("Failed to determine suitable jdbc url", this,
this.embeddedDatabaseConnection);
        } else {
            return url;
        }
    }
}
```

determineDatabaseName

```

public String determineDatabaseName() {
    if (this.generateUniqueName) {
        if (this.uniqueName == null) {
            this.uniqueName = UUID.randomUUID().toString();
        }
    }

    return this.uniqueName;
} else if (StringUtils.hasLength(this.name)) {
    return this.name;
} else {
    return this.embeddedDatabaseConnection != EmbeddedDatabaseConnection.NONE ? "testdb" : null;
}
}

```

afterPropertiesSet

```

public void afterPropertiesSet() throws Exception {
    if (this.embeddedDatabaseConnection == null) {
        this.embeddedDatabaseConnection = EmbeddedDatabaseConnection.get(this.classLoader);
    }
}

```

DataSourceConfiguration

```

package org.springframework.boot.autoconfigure.jdbc;
abstract class DataSourceConfiguration

```

createDataSource

```

protected static <T> T createDataSource(DataSourceProperties properties, Class<? extends DataSource> type) {
    return properties.initializeDataSourceBuilder().type(type).build();
}

```

Hikari

```

static class Hikari {
    Hikari() {
    }

    @Bean
    @ConfigurationProperties(
        prefix = "spring.datasource.hikari"
    )
    HikariDataSource dataSource(DataSourceProperties properties) {
        HikariDataSource dataSource = (HikariDataSource)DataSourceConfiguration.createDataSource(properties,
HikariDataSource.class);
        if (StringUtils.hasText(properties.getName())) {
            dataSource.setPoolName(properties.getName());
        }

        return dataSource;
    }
}

@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnClass({org.apache.tomcat.jdbc.pool.DataSource.class})
@ConditionalOnMissingBean({DataSource.class})
@ConditionalOnProperty(
    name = {"spring.datasource.type"},
    havingValue = "org.apache.tomcat.jdbc.pool.DataSource",
    matchIfMissing = true
)

```

Tomcat

```
static class Tomcat {
    Tomcat() {
    }

    @Bean
    @ConfigurationProperties(
        prefix = "spring.datasource.tomcat"
    )
    org.apache.tomcat.jdbc.pool.DataSource dataSource(DataSourceProperties properties) {
        org.apache.tomcat.jdbc.pool.DataSource dataSource =
        (org.apache.tomcat.jdbc.pool.DataSource)DataSourceConfiguration.createDataSource(properties,
        org.apache.tomcat.jdbc.pool.DataSource.class);
        DatabaseDriver databaseDriver = DatabaseDriver.fromJdbcUrl(properties.determineUrl());
        String validationQuery = databaseDriver.getValidationQuery();
        if (validationQuery != null) {
            dataSource.setTestOnBorrow(true);
            dataSource.setValidationQuery(validationQuery);
        }
        return dataSource;
    }
}
```

JdbcTemplateConfiguration

```
package org.springframework.boot.autoconfigure.jdbc;
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnMissingBean({JdbcOperations.class})
class JdbcTemplateConfiguration
    jdbcTemplate
    @Bean
    @Primary
    JdbcTemplate jdbcTemplate(DataSource dataSource, JdbcProperties properties) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcProperties.Template template = properties.getTemplate();
        jdbcTemplate.setFetchSize(template.getFetchSize());
        jdbcTemplate.setMaxRows(template.getMaxRows());
        if (template.getQueryTimeout() != null) {
            jdbcTemplate.setQueryTimeout((int)template.getQueryTimeout().getSeconds());
        }
        return jdbcTemplate;
    }
}
```

metadata

DataSourcePoolMetadataProvidersConfiguration

```
package org.springframework.boot.autoconfigure.jdbc.metadata;
@Configuration(
    proxyBeanMethods = false
)
public class DataSourcePoolMetadataProvidersConfiguration
    (annotation)
```

AutoConfigureBefore

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE })
@Documented
public @interface AutoConfigureBefore
```

Hint that an auto-configuration should be applied **before** other specified auto-configuration classes.

```
Class<?>[] value() default {};
```

The auto-configure classes that should have not yet been applied.

```
String[] name() default {};
```

The names of the auto-configure classes that should have not yet been applied.

AutoConfigurationPackage

```
package org.springframework.boot.autoconfigure;
```

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({AutoConfigurationPackages.Registrar.class})
public @interface AutoConfigurationPackage 自动配置包, 指定了默认的包规则 @Import({Registrar.class})
```

AutoConfigureOrder

```
package org.springframework.boot.autoconfigure;
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface AutoConfigureOrder
int DEFAULT_ORDER = 0;
int value() default 0;
```

EnableAutoConfiguration

```
package org.springframework.boot.autoconfigure;
```

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

The `@EnableAutoConfiguration` annotation in Spring Boot automatically configures your application **based on the dependencies present on the classpath**.

It enables Spring Boot **to set up default configurations** for your application, such as setting up a web server if `spring-boot-starter-web` is included.

This simplifies the setup process by providing sensible defaults and reducing the need for manual configuration.

SpringBootApplication

```
package org.springframework.boot.autoconfigure;  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class), @Filter(type =  
FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })  
public @interface SpringBootApplication
```

The `@SpringBootApplication` annotation in Spring Boot is a convenience annotation **that combines three key annotations:**

`@SpringBootConfiguration`, `@EnableAutoConfiguration`, `@ComponentScan`
the `@SpringBootApplication` annotation does trigger the automatic configuration of all dependencies **before the execution of the run method** in `SpringApplication`.

```
@AliasFor(annotation = ComponentScan.class, attribute = "basePackages")  
String[] scanBasePackages() default {};
```

context
event

SpringApplicationEvent

```
package org.springframework.boot.context.event;  
public abstract class SpringApplicationEvent extends ApplicationEvent  
    org.springframework.boot.context.event.ApplicationContextInitializedEvent  
    org.springframework.boot.context.event.ApplicationEnvironmentPreparedEvent  
    org.springframework.boot.context.event.ApplicationFailedEvent  
    org.springframework.boot.context.event.ApplicationPreparedEvent  
    org.springframework.boot.context.event.ApplicationReadyEvent  
    org.springframework.boot.context.event.ApplicationStartedEvent  
    org.springframework.boot.context.event.ApplicationStartingEvent  
    org.springframework.boot.context.event.EventPublishingRunListener
```

ApplicationContextInitializedEvent

```
package org.springframework.boot.context.event;  
public class ApplicationContextInitializedEvent extends SpringApplicationEvent
```

`ApplicationContextInitializedEvent` is part of the `org.springframework.boot.context.event` package and is used within the Spring Boot application lifecycle.

It is particularly useful for scenarios where you need to **modify the context configuration**, register additional beans, or perform other setup tasks before the context is fully refreshed.

Key Responsibilities

Early Interception

Allows for interception and modification of the `ApplicationContext` **after it has been initialized but before it is refreshed**.

Context Configuration

Provides an opportunity **to add additional configuration, register beans, or perform other setup tasks**.

```
public ConfigurableApplicationContext getApplicationContext()
```

ApplicationEnvironmentPreparedEvent

```
package org.springframework.boot.context.event;
```

```
public class ApplicationEnvironmentPreparedEvent extends SpringApplicationEvent
```

Key Responsibilities

Environment Customization

Provides a hook to **customize the application's environment**, including modifying property sources, adding profiles, or configuring other environment-related settings.

Early Access

Occurs early **in the application startup process**, allowing modifications to be applied before the ApplicationContext is created.

ApplicationFailedEvent

```
package org.springframework.boot.context.event;
```

```
public class ApplicationFailedEvent extends SpringApplicationEvent
```

ApplicationFailedEvent is part of the Spring Boot application lifecycle and is published when an exception occurs that prevents the application from starting successfully.

It provides details about the exception that caused the failure.

Key Responsibilities

Error Handling

Provides a hook to **handle application startup failures gracefully**, such as logging the error details, sending notifications, or performing cleanup tasks.

Exception Details

Includes information about the exception that caused the application startup failure, helping developers diagnose and troubleshoot issues.

ApplicationPreparedEvent

```
package org.springframework.boot.context.event;
```

```
public class ApplicationPreparedEvent extends SpringApplicationEvent
```

ApplicationPreparedEvent is part of the Spring Boot application lifecycle and serves as a marker event between the application environment being prepared and the actual application context being refreshed. It allows for custom initialization tasks to be performed after the environment is configured but before the beans are loaded and initialized.

Key Responsibilities

Custom Initialization

Provides a hook to perform custom initialization tasks **after the application context is prepared but before it is fully refreshed**.

Access to Environment

Allows access to the ConfigurableEnvironment to **modify property sources or profiles** before the context refresh.

ApplicationReadyEvent

```
package org.springframework.boot.context.event;
```

```
public class ApplicationReadyEvent extends SpringApplicationEvent
```

ApplicationReadyEvent is part of the Spring Boot application lifecycle and is published when the application context is fully prepared and ready for use. It typically follows after ApplicationStartedEvent and signifies that the application is up

and running.

Key Responsibilities

Notification of Readiness

Signals that the application context has been fully refreshed and initialized, and the application is ready to handle business logic and respond to requests.

Post-Initialization Tasks

Provides a hook to execute tasks that should run after the application is fully started, such as triggering background jobs or performing additional configuration.

ApplicationStartedEvent

```
package org.springframework.boot.context.event;  
public class ApplicationStartedEvent extends SpringApplicationEvent
```

ApplicationStartedEvent is part of the Spring Boot application lifecycle and is published when the application context has started to load and is about to be refreshed.

It allows for early interception and customization before the full initialization of the application context.

Key Responsibilities

Early Initialization

Provides a hook to perform initialization tasks after the application context has started to load, but before it is fully refreshed.

Access to Context

Allows access to the application context in its early stages, enabling modifications or customizations before beans are fully initialized.

ApplicationStartingEvent

```
package org.springframework.boot.context.event;  
public class ApplicationStartingEvent extends SpringApplicationEvent
```

ApplicationStartingEvent is one of the earliest events in the Spring Boot application lifecycle.

It is triggered when the application context is just about to start loading and before any beans or configurations are initialized.

This event provides an opportunity to perform very early initialization tasks or to customize the application environment before it is fully configured.

Key Responsibilities

Early Initialization

Allows for performing tasks that need to be executed before the application context is fully initialized.

Access to Environment

Provides access to the application's environment to customize property sources, add profiles, or modify configurations before the application context is prepared.

EventPublishingRunListener

```
package org.springframework.boot.context.event;  
public class EventPublishingRunListener implements SpringApplicationRunListener, Ordered
```

The EventPublishingRunListener class is part of the core Spring Boot framework and is used internally to dispatch events at key points during the application's lifecycle. It extends the AbstractRunListener class and integrates with Spring's event publishing mechanism to notify registered listeners about significant milestones in the application's startup and shutdown processes.

Key Responsibilities

Event Publishing

Publishes events at critical points during the application lifecycle, such as when the application is starting, started, failed, or shutting down.

Integration with SpringApplication

Integrates with SpringApplication to provide hooks for event publishing at various stages of the application's execution.

properties

ConfigurationPropertiesBindingPostProcessor

```
package org.springframework.boot.context.properties;  
public class ConfigurationPropertiesBindingPostProcessor
```

implements BeanPostProcessor, PriorityOrdered, ApplicationContextAware, InitializingBean

ConfigurationPropertiesBindingPostProcessor is a key component in Spring Boot that handles the binding of external configuration properties to Java objects annotated with `@ConfigurationProperties`.

This processor is responsible for ensuring that the properties defined in configuration files (like application.properties or application.yml) are correctly mapped to the corresponding fields in your `@ConfigurationProperties` annotated classes.

Key Features

- Automatic Binding
Automatically binds properties from configuration files to the target bean properties.
- Validation
Supports validation of the configuration properties using JSR-303/JSR-380 annotations.
- Customization
Allows customization of the binding process through various strategies and converters.

How ConfigurationPropertiesBindingPostProcessor Is Registered

- Spring Boot Auto-Configuration

Spring Boot has a class called `ConfigurationPropertiesAutoConfiguration` that registers `ConfigurationPropertiesBindingPostProcessor`.

Custom Converter

You can customize the binding process by defining custom property editors or converters.

This allows you to transform property values before they are set on the target bean.

```
import org.springframework.boot.context.properties.ConfigurationPropertiesBinding;  
import org.springframework.core.convert.converter.Converter;  
import org.springframework.stereotype.Component;  
  
@Component  
@ConfigurationPropertiesBinding  
public class StringToCustomObjectConverter implements Converter<String, CustomObject> {  
  
    @Override  
    public CustomObject convert(String source) {  
        // Convert the source string to a CustomObject instance  
        return new CustomObject(source);  
    }  
}
```

(annotation)

ConfigurationProperties

```
package org.springframework.boot.context.properties;  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented
```

```
@Indexed
```

```
public @interface ConfigurationProperties
```

The `@ConfigurationProperties` annotation in Spring Boot is used to bind externalized properties from application configuration files (like `application.properties` or `application.yml`) to a Java object.

This feature helps to organize and manage configuration properties in a type-safe manner.

See: ConfigurationPropertiesBindingPostProcessor

Key Features

- Type-Safe Configuration
Binds properties from configuration files to Java objects.
- Supports Hierarchical Properties
Can map hierarchical properties to nested Java objects.
- Ease of Use
Simplifies the injection and management of configuration values.

```
@AliasFor("prefix")
```

```
String value() default "";
```

```
@AliasFor("value")
```

```
String prefix() default ""; 配置属性前缀 (zcloud.name 将加载到类属性 name 上, 逗号分隔可以直接用 List 接收) // "zcloud"
```

```
boolean ignoreInvalidFields() default false; 不忽略识别不了的字段
```

```
boolean ignoreUnknownFields() default true;
```

Configuration Properties Class

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
```

```
@Component
```

```
@ConfigurationProperties(prefix = "myapp")
```

```
public class MyAppProperties {
```

```
    private String name;
    private String description;
```

```
    // Getters and setters
```

```
    public String getName() {
```

```
        return name;
```

```
}
```

```
    public void setName(String name) {
```

```
        this.name = name;
```

```
}
```

```
    public String getDescription() {
```

```
        return description;
```

```
}
```

```
    public void setDescription(String description) {
```

```
        this.description = description;
```

```
}
```

```
}
```

Configuration File (`application.properties`)

```
myapp.name=My Application
```

```
myapp.description=This is a sample application
```

Using the Configuration Properties

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyAppService {

    private final MyAppProperties myAppProperties;

    @Autowired
    public MyAppService(MyAppProperties myAppProperties) {
        this.myAppProperties = myAppProperties;
    }

    public void printProperties() {
        System.out.println("App Name: " + myAppProperties.getName());
        System.out.println("App Description: " + myAppProperties.getDescription());
    }
}

```

Configuration Properties Class with Nested Properties

```

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

```

```

@Component
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {

    private String name;
    private String description;
    private final DatabaseProperties database = new DatabaseProperties();

    // Getters and setters

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public DatabaseProperties getDatabase() {
        return database;
    }

    public static class DatabaseProperties {

        private String url;
        private String username;
        private String password;

        // Getters and setters

        public String getUrl() {
            return url;
        }

        public void setUrl(String url) {
            this.url = url;
        }
    }
}

```

```

        public String getUsername() {
            return username;
        }

        public void setUsername(String username) {
            this.username = username;
        }

        public String getPassword() {
            return password;
        }

        public void setPassword(String password) {
            this.password = password;
        }
    }
}

```

Configuration File (application.yml)

```

myapp:
  name: My Application
  description: This is a sample application
  database:
    url: jdbc:mysql://localhost:3306/mydb
    username: myuser
    password: mypassword

```

EnableConfigurationProperties

```

package org.springframework.boot.context.properties;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import({EnableConfigurationPropertiesRegistrar.class})
public @interface EnableConfigurationProperties

```

The `@EnableConfigurationProperties` annotation in Spring Boot is used to enable support for `@ConfigurationProperties`-annotated beans.

It ensures that the `@ConfigurationProperties` beans are recognized and registered in the Spring context.

Key Features

- Automatic Configuration
Automatically registers `@ConfigurationProperties` beans without needing explicit bean definitions.
- Simplifies Configuration Management
Makes it easier to manage and inject configuration properties into Spring beans.
- Supports Multiple Properties Classes
Can enable multiple `@ConfigurationProperties` classes.

NestedConfigurationProperty

```

package org.springframework.boot.context.properties;

@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Nested
public @interface NestedConfigurationProperty

```

`@NestedConfigurationProperty` is an annotation in Spring Boot that is used to indicate that a field within a

@ConfigurationProperties class is a nested configuration property. This is useful when you have complex configuration properties that are better represented as separate objects.

Key Features

- Indicates Nesting
Clearly indicates that a field is a nested configuration property.
- Improves Readability
Helps in organizing and structuring complex configuration properties.
- Type-Safe Configuration
Maintains type-safety while handling nested configuration properties.

Nested Configuration Class

```
public class SecurityProperties {  
  
    private String username;  
    private String password;  
  
    // Getters and setters  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String username) {  
        this.username = username;  
    }  
  
    public String getPassword() {  
        return password;  
    }  
  
    public void setPassword(String password) {  
        this.password = password;  
    }  
}
```

Main Configuration Properties Class

```
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.boot.context.properties.NestedConfigurationProperty;  
import org.springframework.stereotype.Component;  
  
@Component  
@ConfigurationProperties(prefix = "myapp")  
public class MyAppProperties {  
  
    private String name;  
    private String description;  
  
    @NestedConfigurationProperty  
    private SecurityProperties security = new SecurityProperties();  
  
    // Getters and setters  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
    public void setDescription(String description) {
```

```

        this.description = description;
    }

    public SecurityProperties getSecurity() {
        return security;
    }

    public void setSecurity(SecurityProperties security) {
        this.security = security;
    }
}

Configuration File (application.properties)
myapp.name=My Application
myapp.description=This is a sample application
myapp.security.username=admin
myapp.security.password=secret

```

support

AbstractApplicationContext

```

package org.springframework.context.support;
public abstract class AbstractApplicationContext extends DefaultResourceLoader implements
ConfigurableApplicationContext
refresh
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        StartupStep contextRefresh = this.applicationStartup.start("spring.context.refresh");

        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            StartupStep beanPostProcess = this.applicationStartup.start("spring.context.beans.post-process");
            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);
            beanPostProcess.end();

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
        }
    }
}

```

```

        finishRefresh();
    }

    catch (BeansException ex) {
        if (logger.isWarnEnabled()) {
            logger.warn("Exception encountered during context initialization - " +
                        "cancelling refresh attempt: " + ex);
        }
    }

    // Destroy already created singletons to avoid dangling resources.
    destroyBeans();

    // Reset 'active' flag.
    cancelRefresh(ex);

    // Propagate exception to caller.
    throw ex;
}

finally {
    // Reset common introspection caches in Spring's core, since we
    // might not ever need metadata for singleton beans anymore...
    resetCommonCaches();
    contextRefresh.end();
}
}
}
}

```

annotation

ImportCandidates

```

package org.springframework.boot.context.annotation;
public final class ImportCandidates implements Iterable<String>

private static final String LOCATION = "META-INF/spring/%s.imports";

load
public static ImportCandidates load(Class<?> annotation, ClassLoader classLoader) {
    Assert.notNull(annotation, "'annotation' must not be null");
    ClassLoader classLoaderToUse = decideClassLoader(classLoader);
    String location = String.format(LOCATION, annotation.getName());
    Enumeration<URL> urls = findUrlsInClasspath(classLoaderToUse, location);
    List<String> importCandidates = new ArrayList<>();
    while (urls.hasMoreElements()) {
        URL url = urls.nextElement();
        importCandidates.addAll(readCandidateConfigurations(url));
    }
    return new ImportCandidates(importCandidates);
}

```

env

EnvironmentPostProcessor

```

package org.springframework.boot.env;
public interface EnvironmentPostProcessor 环境后置处理器，用户读取配置文件后 对环境变量进行操作
void postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application);

```

SimpleCommandLinePropertySource

```
package org.springframework.boot.env;
public class SimpleCommandLinePropertySource extends CommandLinePropertySource<CommandLineArgs>
protected boolean containsOption(String name)

public String[] getPropertyNames()
protected List<String> getOptionValues(String name)
protected List<String> getNonOptionArgs()
```

jdbc

DataSourceBuilder

```
package org.springframework.boot.jdbc;
public final class DataSourceBuilder<T extends DataSource>
```

```
DataSourceBuilder
```

```
private DataSourceBuilder(ClassLoader classLoader) {
    this.classLoader = classLoader;
    this.deriveFrom = null;
}
```

```
type
```

```
public <D extends DataSource> DataSourceBuilder<D> type(Class<D> type) {
    this.type = type;
    return this;
}
```

```
create
```

```
public static DataSourceBuilder<?> create(ClassLoader classLoader) {
    return new DataSourceBuilder(classLoader);
}
```

```
driverClassName
```

```
public DataSourceBuilder<T> driverClassName(String driverClassName) {
    this.set(DataSourceBuilder.DataSourceProperty.DRIVER_CLASS_NAME, driverClassName);
    return this;
}
```

```
url
```

```
public DataSourceBuilder<T> url(String url) {
    this.set(DataSourceBuilder.DataSourceProperty.URL, url);
    return this;
}
```

```
username
```

```
public DataSourceBuilder<T> username(String username) {
    this.set(DataSourceBuilder.DataSourceProperty.USERNAME, username);
    return this;
}
```

```
password
```

```
public DataSourceBuilder<T> password(String password) {
    this.set(DataSourceBuilder.DataSourceProperty.PASSWORD, password);
    return this;
}
```

orm

SpringPhysicalNamingStrategy

```
package org.springframework.boot.orm.jpa.hibernate;
public class SpringPhysicalNamingStrategy implements PhysicalNamingStrategy 在进行领域映射时,首字母小写, 大写字母变为下划线加小写
```

security
config

注解

```
package org.springframework.security.config.annotation.method.configuration;
@EnableGlobalMethodSecurity 开启项目中 @PreAuthorize、@PostAuthorize 以及 @Secured 注解的使用
    prePostEnabled=true
```

scheduling
annotation

AsyncConfigurer

```
package org.springframework.scheduling.annotation;
public interface AsyncConfigurer 异步任务线程池 (所有的异步任务共享)【生成 excel 文件】
    default Executor getAsyncExecutor 获取异步执行器
    default AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler 异步任务中的异常处理
TaskThreadPoolConfig >>
```

```
@Data
@ConfigurationProperties(prefix = "thread")
public class TaskThreadPoolConfig {
    private int corePoolSize;
    private int maxPoolSize;
    private int keepAliveSeconds;
    private int queueCapacity;
}
```

AsyncTaskExecutePool >>

```
@Slf4j
@Configuration
@EnableAsync
public class AsyncTaskExecutePool implements AsyncConfigurer {

    @Autowired
    private TaskThreadPoolConfig config; // 配置属性类, 见上面的代码

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor(){
            @Override
            public <T> Future<T> submit(Callable<T> task) {
                // 传入线程池之前先复制当前线程的 MDC
                return super.submit(ThreadMdcUtil.wrap(task, MDC.getCopyOfContextMap()));
            }
            @Override
            public void execute(Runnable task) {
                super.execute(ThreadMdcUtil.wrap(task, MDC.getCopyOfContextMap()));
            }
        };
    }
}
```

```

executor.setCorePoolSize(config.getCorePoolSize());
executor.setMaxPoolSize(config.getMaxPoolSize());
executor.setQueueCapacity(config.getQueueCapacity());
executor.setKeepAliveSeconds(config.getKeepAliveSeconds());
executor.setThreadNamePrefix("taskExecutor-");

// rejection-policy: 当 pool 已经达到 max size 的时候, 如何处理新任务
// CALLER_RUNS: 不在新线程中执行任务, 而是由调用者所在的线程来执行
executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
executor.initialize();
return executor;
}

@Override
public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {// 异步任务中异常处理
    return new AsyncUncaughtExceptionHandler() {
        @Override
        public void handleUncaughtException(Throwable arg0, Method arg1, Object... arg2) {
            log.error("======" + arg0.getMessage() + "=====",
                    arg0);
            log.error("exception method:" + arg1.getName());
        }
    };
}
}

```

test
mock

注解

package org.springframework.boot.test.mock.mockito;
@MockBean 模拟一个 service 实例, 测试的代码执行的时候不会真的去调用 service 的逻辑, 而是需要我们给一个模拟的动作。

context

注解

package org.springframework.boot.test.context;
@SpringBootTest 单元测试类, 使用 spring 测试驱动, 包含容器功能, 会准备好 web 环境
 classes= Application.class 默认启动整个程序, 可以只启动测试类
 webEnvironment=SpringBootTest.WebEnvironment.MOCK 测试环境【MOCK 是否启动 web 环境 RANDOM_PORT 使用随机端口作为 web 服务器端口 DEFINED_PORT 使用自定义的端口作为 web 服务器端口 NONE 不启动 web 环境】

@TestPropertySource 覆盖掉来自于系统环境变量、Java 系统属性、@PropertySource 的属性
 locations = {"classpath:application-test.properties"} 使用的属性文件

autoconfigure

注解

package org.springframework.boot.test.autoconfigure.web.servlet;
@AutoConfigureMockMvc 自动配置 MockMvc 的 bean 实例到类中

web

embedded

TomcatWebServer

```
package org.springframework.boot.web.embedded.tomcat;
public class TomcatWebServer implements WebServer
private void initialize() throws WebServerException    初始化服务器
private String getPortsDescription(boolean localPort)    获取端口描述
```

servlet

RegistrationBean

```
package org.springframework.boot.web.servlet;
public abstract class RegistrationBean implements ServletContextInitializer, Ordered
public void setOrder(int order)    设置 Filter 顺序
```

DynamicRegistrationBean

```
package org.springframework.boot.web.servlet;
public abstract class DynamicRegistrationBean<D extends Dynamic> extends RegistrationBean
public void setName(String name)          设置 Filter 名称
public Map<String, String> getInitParameters()  设置 Filter 参数
public void setInitParameters(Map<String, String> initParameters)
```

AbstractFilterRegistrationBean

```
package org.springframework.boot.web.servlet;
public abstract class AbstractFilterRegistrationBean<T extends Filter> extends DynamicRegistrationBean<Dynamic>
```

FilterRegistrationBean

```
package org.springframework.boot.web.servlet;
public class FilterRegistrationBean<T extends Filter> extends AbstractFilterRegistrationBean<T>
public void addUrlPatterns(String... urlPatterns)    指定过滤资源 (不支持/**语法, /*即为所有)
    registration.addUrlPatterns("/*");                过滤应用程序中所有资源, 当前应用程序根目录下的所有文件包括多级子目录下
    的所有文件, 注意这里*前有 "/"
    registration.addUrlPatterns(".html");            过滤指定的类型文件资源, 当前应用程序根目录下的所有 html 文件, 注意:
    *.html 前没有"/",否则错误
    registration.addUrlPatterns("/folder_name/*");    过滤指定的目录下的所有文件, 当前应用程序根目录下的 folder_name 子
    目录 (可以是多级子目录) 下所有文件
    registration.addUrlPatterns("/index.html");       过滤指定文件
```

Usage

```
FilterConf >>
@Configuration
public class FilterConf {

    @Autowired
    private CommonService commonService;

    @Bean
    public FilterRegistrationBean buildTokenFilter() {
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean();
        filterRegistrationBean.setOrder(3);
        filterRegistrationBean.setFilter(new TokenFilter(commonService));
        filterRegistrationBean.setName("TokenFilter");
        filterRegistrationBean.addUrlPatterns("/*");
        filterRegistrationBean.addInitParameter("passuri",
```

```

        ".*/login.*," +
        ".*/test.*," +
        ".*/download.*," +
        ".*/rpc.*," +
        ".*/logout.*," +
        ".*/validate.*"+
        ".*/swagger-resources.*,"+
        ".*/swagger-ui.html,"+
        ".*/webjars.*,"+
        ".*/v2.*");
    return filterRegistrationBean;
}
}
}

```

context

ServletWebServerApplicationContext

```

package org.springframework.boot.web.servlet.context;
public class ServletWebServerApplicationContext extends GenericWebApplicationContext implements
ConfigurableWebServerApplicationContext 当 springboot 使用内置 web 容器时的 IOC 容器类。会将内置 web 服务器生命周期
加入到 spring 容器的生命周期中
protected void prepareWebApplicationContext(ServletContext servletContext) 准备 web 应用容器
support

```

SpringBootServletInitializer

```

package org.springframework.boot.web.servlet.support;
public abstract class SpringBootServletInitializer implements WebApplicationInitializer

```

When deploying a Spring Boot application as a WAR file, you need to extend `SpringBootServletInitializer` and override its `configure` method.

This class binds Spring Boot's application context to the servlet container, allowing the application to be launched using the servlet container's `web.xml` configuration.

Key Responsibilities

- **WAR Deployment**

Enables the application to be deployed in a traditional servlet container by providing configuration methods.

- **Application Configuration**

Allows customization of the Spring Boot application configuration before it is launched by the servlet container.

```
protected SpringApplicationBuilder configure(SpringApplicationBuilder builder)
```

```
public void onStartup(ServletContext servletContext) throws ServletException
```

(annotation)

```
package org.springframework.boot.web.servlet;
```

```
@ServletComponentScan( basePackages = "com.dongfeng" ) 扫描手动注入的原生组件@WebServlet
```

SpringBoot / org.springframework.cloud

commons

InetUtils

```

package org.springframework.cloud.commons.util;
public class InetUtils implements Closeable

```

```
public InetUtils(final InetUtilsProperties properties)
```

```
public HostInfo findFirstNonLoopbackHostInfo()
public InetAddress findFirstNonLoopbackAddress()
boolean isPreferredAddress(InetAddress address)
public HostInfo convertAddress(final InetAddress address)
```

context
scope

GenericScope

```
package org.springframework.cloud.context.scope;
public class GenericScope implements Scope, BeanFactoryPostProcessor, BeanDefinitionRegistryPostProcessor,
DisposableBean      缓存 bean 容器
对每一个被管辖的 bean 都维护了一个读写锁 ReentrantReadWriteLock 用户控制并发读写
同时也维护了一个 GenericScope#BeanLifecycleWrapper 控制每个被管辖 Bean 的生命周期：缓存最新创建的 Bean 实例、创建过程 ObjectFactory、销毁回调函数。
```

```
public Object get(String name, ObjectFactory<?> objectFactory)  获取
```

```
private void setSerializationId(ConfigurableListableBeanFactory beanFactory)  设置序列化 id
```

config
(annotation)

```
package org.springframework.cloud.context.config.annotation;
```

```
@RefreshScope    刷新配置文件属性，和@ConfigurationProperties 配合使用 【TYPE, METHOD】
```

loadbalancer
core

RoundRobinLoadBalancer

```
package org.springframework.cloud.loadbalancer.core;
public class RoundRobinLoadBalancer implements ReactorServiceInstanceLoadBalancer    轮询，按公约后的权重设置轮询比率。(存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上)
```

openfeign
encoding

BaseRequestInterceptor

```
package org.springframework.cloud.openfeign.encoding;
```

```
public abstract class BaseRequestInterceptor implements RequestInterceptor    定义了 addHeader 方法，往 requestTemplate 添加非重名的 header
```

FeignAcceptGzipEncodingInterceptor

```
package org.springframework.cloud.openfeign.encoding;
```

```
public class FeignAcceptGzipEncodingInterceptor extends BaseRequestInterceptor    它的 apply 方法往 RequestTemplate 添加了名为 Accept-Encoding，值为 gzip,deflate 的 header
```

FeignContentGzipEncodingInterceptor

```
package org.springframework.cloud.openfeign.encoding;
```

```
public class FeignContentGzipEncodingInterceptor extends BaseRequestInterceptor    其 apply 方法先判断是否需要
```

compression, 即 mimeType 是否符合要求以及 content 大小是否超出阈值, 需要 compress 的话则添加名为 Content-Encoding, 值为 gzip,deflate 的 header

(annotations)

EnableFeignClients

```
package org.springframework.cloud.openfeign;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(FeignClientsRegistrar.class)
public @interface EnableFeignClients

String[] basePackages() default {};
Base packages to scan for annotated components.
```

SpringQueryMap

```
package org.springframework.cloud.openfeign;

@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.PARAMETER })
public @interface SpringQueryMap {

}
```

Define a POJO for Query Parameters

Create a POJO class that represents the query parameters you want to send with the HTTP request:

```
public class QueryParams {
    private String param1;
    private String param2;
    private Integer param3;

    // Getters and setters

    public String getParam1() {
        return param1;
    }

    public void setParam1(String param1) {
        this.param1 = param1;
    }

    public String getParam2() {
        return param2;
    }

    public void setParam2(String param2) {
        this.param2 = param2;
    }

    public Integer getParam3() {
        return param3;
    }

    public void setParam3(Integer param3) {
        this.param3 = param3;
    }
}
```

```
}
```

Define a Feign Client Interface

Create a Feign client interface and use the `@SpringQueryMap` annotation to map the POJO to query parameters:

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.cloud.openfeign.SpringQueryMap;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@FeignClient(name = "myFeignClient", url = "http://example.com")
public interface MyFeignClient {

    @GetMapping("/endpoint")
    String getWithQueryParams(@SpringQueryMap QueryParams queryParams);
}
```

client

ServiceInstance

```
package org.springframework.cloud.client;
public interface ServiceInstance 服务实例
```

```
String getServiceId();
String getHost();
int getPort();
boolean isSecure();
URI getUri();
Map<String, String> getMetadata();
```

loadbalancer

LoadBalancerClient

```
package org.springframework.cloud.client.loadbalancer;
```

```
public interface LoadBalancerClient extends ServiceInstanceChooser 负载均衡客户端
```

ServiceInstanceChooser

```
package org.springframework.cloud.client.loadbalancer;
```

```
public interface ServiceInstanceChooser 服务选择器
```

```
ServiceInstance choose(String serviceId) 选择一个服务
```

gateway

filter

GlobalFilter [CORE]

```
package org.springframework.cloud.gateway.filter;
```

```
public interface GlobalFilter
```

```
    @Component
    public class MyFilter implements GlobalFilter, Ordered {
        @Override
        public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
            return chain.filter(exchange); // Release chain
        }
        @Override
```

```
    public int getOrder() {
        return 0;
    }
}
```

```
Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);
```

ModifyResponseBodyGatewayFilterFactory

```
package org.springframework.cloud.gateway.filter.factory.rewrite;
public class ModifyResponseBodyGatewayFilterFactory extends
AbstractGatewayFilterFactory<ModifyResponseBodyGatewayFilterFactory.Config>
```

SpringBoot / org.apache

commons

logging

LogFactory

```
package org.apache.commons.logging;
public abstract class LogFactory 日志工厂
public static Log getLog(Class<?> clazz) 获取日志对象
```

LogAdapter

```
package org.apache.commons.logging;
final class LogAdapter 日志适配器
public static Log createLog(String name) 创建日志对象
private static enum LogApi { 日志 api 类型
    LOG4J,
    SLF4J_LAL,
    SLF4J,
    JUL;
}

private LogApi() {
}
}
```

catalina

connector

Request

```
package org.apache.catalina.connector;
public class Request implements HttpServletRequest
public Map<String, String[]> getParameterMap() Parse parameters and get names and values.
protected void parseParameters() Check the content-type header and call parseParts method.
private void parseParts(boolean explicit)
```

RequestFacade

```
package org.apache.catalina.connector;
public class RequestFacade implements HttpServletRequest
public Map<String, String[]> getParameterMap() Check access permissions and call the getParameterMap method of
the source request object.
```

core

ApplicationFilterChain

public final class **ApplicationFilterChain** implements FilterChain

public void **doFilter**(ServletRequest request, ServletResponse response) throws IOException, ServletException

private void **internalDoFilter**(ServletRequest request, ServletResponse response) throws IOException, ServletException

In debugging mode, set a breakpoint in **internalDoFilter()** within Tomcat's ApplicationFilterChain to inspect filters at runtime.

ContainerBase

package org.apache.catalina.core;

public abstract class **ContainerBase** extends LifecycleMBeanBase implements Container 基础容器

StandardService

package org.apache.catalina.core;

public class **StandardService** extends LifecycleMBeanBase implements Service 标准服务

protected void **startInternal()** throws LifecycleException 启动服务

standardService.start.name=Starting service [{0}]

StandardEngine

package org.apache.catalina.core;

public class **StandardEngine** extends ContainerBase implements Engine 标准引擎

standardEngine.start=Starting Servlet engine: [{0}]

protected synchronized void **startInternal()** throws LifecycleException 启动引擎

SpringBoot / Third Party Package

Core

Core Dependencies

Required Dependencies

spring-boot-starter-web / spring-boot-starter-webflux

```
<!-- Spring Boot Starter Web for Servlet-based web applications -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```
<!-- Spring Boot Starter WebFlux for reactive web applications -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Spring Boot Starter Web

Used for Servlet-based web applications.

Includes [spring-boot-starter-tomcat](#) for using Tomcat as the default server.

Can be replaced with [spring-boot-starter-jetty](#) or [spring-boot-starter-undertow](#) if desired.

Spring Boot Starter WebFlux

Used for reactive web applications.

Includes [spring-boot-starter-reactor-netty](#) for using Reactor Netty as the default server.

Can be replaced with [spring-boot-starter-tomcat](#), [spring-boot-starter-jetty](#), or [spring-boot-starter-undertow](#) if needed.

spring-boot-starter-test

```
<!-- Spring Boot Starter Test for testing dependencies -->
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```

<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusion>
<groupId>com.vaadin.external.google<
android-json
</dependency>

```

Supports common testing dependencies, including JUnit, Hamcrest, Mockito, and the spring-test module.

Relies on spring-boot-starter-actuator.

Dependency Management

Maven

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud-dependencies.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring-boot-dependencies.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>

    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>${spring-cloud-alibaba-dependencies.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

Gradle

```

plugins {
  id 'org.springframework.boot' version '3.1.0'
  This plugin simplifies building and running Spring Boot applications.
  It provides tasks to package your app as an executable JAR or WAR and manage your Spring Boot application's
  lifecycle.
  id 'io.spring.dependency-management' version '1.1.0'
  Manages dependency versions in a Maven-like way.
  It ensures consistent dependency versions across the project, especially when dealing with transitive dependencies in
  Spring Boot.
  id 'java'
  Gradle's standard Java plugin for compiling and packaging Java code. It includes tasks for building, running tests, and
  generating JAR files.
}

dependencyManagement {
  imports {
    mavenBom "org.springframework.boot:spring-boot-dependencies:3.0.6"
    mavenBom "org.springframework.cloud:spring-cloud-dependencies:2022.0.0"
  }
}

```

Startup Class for Web or WebFlux

@SpringBootApplication

```

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

```

Optional Dependencies

spring-boot-starter	这是 Spring Boot 的核心启动器，包含了自动配置、日志和 YAML
spring-boot-configuration-processor	给 自定义的配置类 生成元数据信息的，因为 spring 也不知道你有哪些配置类，所以搞了这个方便大家自定义 (optional 不传递，仅在编译期间使用，常用于自定义 starter，搭配@ConfigurationProperties)
spring-boot-autoconfigure	自动配置
spring-boot-starter-data-jdbc	连接数据库 JDBC
spring-boot-devtools	热加载 (秒级服务重载)
spring-boot-starter-jetty	引入了 Jetty HTTP 引擎 (用于替换 Tomcat，需要在 spring-boot-starter-web 中排除 spring-boot-starter-tomcat 依赖)
spring-boot-starter-tomcat	引入了 Spring Boot 默认的 HTTP 引擎 Tomcat。
spring-boot-starter-undertow	引入了 Undertow HTTP 引擎 (用于替换 Tomcat，需要在 spring-boot-starter-web 中排除 spring-boot-starter-tomcat 依赖)
spring-cloud-starter-bootstrap	能识别 yaml (yml 在父工程会出问题)

Parent Project Dependency Management

Version

依赖: spring-boot-starter-parent (版本同 springboot，父工程也是层层依赖的，层层继承)

兼容: springboot >> spring cloud alibaba (springboot2020 版之后不支持 ribbon)

添加兼容依赖: spring-cloud-starter-loadbalancer

spring.cloud.loadbalancer.ribbon.enable: false // 禁用 ribbon (高版本不需要)

兼容: spring cloud >> spring boot

2021.0.x aka Jubilee	2.6.x
2020.0.x aka Ilford	2.4.x, 2.5.x (Starting with 2020.0.3)
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x
spring 升级 6.0.6	

JDK 17+ 和 Jakarta EE 9+ 基线

整个框架代码库现在基于 Java 17 源代码级别。

从迁移 javax 到 jakartaServlet、JPA 等命名空间

```
javax.servlet = javax.servlet-api →  
    jakarta.servlet = jakarta.serlvet-api  
    jakarta.servlet = jakarta.persistence-api  
    jakarta.servlet = jakarta.transaction-api
```

与 Jakarta EE 9 和 Jakarta EE 10 API 的运行时兼容性。

兼容最新的网络服务器：Tomcat 10.1、Jetty 11、Undertow 2.3。

与虚拟线程的早期兼容性（从 JDK 19 开始预览）。

一般核心修订

升级到 ASM 9.4 和 Kotlin 1.7。

完整的 CGLIB 分支，支持捕获 CGLIB 生成的类。

提前转换的综合基础。

对 GraalVM 本机图像的一流支持（请参阅相关的 Spring Boot 3 博客文章）。

核心容器

java.beans.Introspector 默认情况下没有基本的 bean 属性确定。

GenericApplicationContext()中的 AOT 处理支持 refreshForAotProcessing。

基于预先解析的构造函数和工厂方法的 Bean 定义转换。

支持 AOP 代理和配置类的早期代理类确定。

PathMatchingResourcePatternResolver 使用 NIO 和模块路径 API 进行扫描，分别支持在 GraalVM 本机映像和 Java 模块路径中进行类路径扫描。

DefaultFormattingConversionService 支持基于 ISO 的默认 java.time 类型解析。

数据访问和交易

支持预先确定 JPA 托管类型（用于包含在 AOT 处理中）。

JPA 支持 Hibernate ORM 6.1（保持与 Hibernate ORM 5.6 的兼容性）。

升级到 R2DBC 1.0（包括 R2DBC 事务定义）。

删除 JCA CCI 支持。

application.yml >>

子项目，引入父工程定义的配置文件，会和当前子项目合并 // bootstrap.yml application-dev.yml

```
#=====  
===== Core  
===== Database  
===== Log
```

pow.xml >> 子项目

```
<parent>  
  <artifactId>sdk-service</artifactId>  
  <groupId>com.saidake</groupId>  
  <version>1.0.0-SNAPSHOT</version>  
</parent>  
<modelVersion>4.0.0</modelVersion>
```

```

<artifactId>sdk-mybatis-plus</artifactId>
<description>mybatis-plus </description>
<!--
=====
===== Customization -->
<!--
=====
===== Core -->
<!--
=====
===== Data Storage -->
<!--
=====
===== Security -->
<!--
=====
===== Function Extension -->
<!--
=====
===== Data Utils -->
<!--
=====
===== Log -->

```

pow.xml >> 父项目

```

<modelVersion>4.0.0</modelVersion>

<groupId>sdk.shop</groupId>      <!-- 删除 parent -->
<artifactId>sdk-boss</artifactId>
<version>1.0-SNAPSHOT</version>

<packaging>pom</packaging>

<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>

    <knife4j.version>2.0.5</knife4j.version>
    <swagger.version>2.9.2</swagger.version>

    <spring-cloud-alibaba-dependencies.version>2021.1</spring-cloud-alibaba-dependencies.version> <!-- spring cloud
alibaba 依赖管理 -->
    <spring-boot-dependencies.version>2.5.9</spring-boot-dependencies.version> <!-- spring boot
依赖管理 -->

```

```

<spring-cloud-dependencies.version>2020.0.4</spring-cloud-dependencies.version>           <!-- spring cloud 依
赖管理 -->
</properties>

<dependencies>
    <!--
=====
===== 继承依赖 -->
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>      # 核心继承依赖 (commons-lang3, commons-collections4, commons-
beanutils, lombok, spring-boot-starter-test, spring-cloud-starter-bootstrap)
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>

        <!--
=====
===== 自定义依赖-->
        <dependency>
            <groupId>com.zlt</groupId>
            <artifactId>zlt-common-spring-boot-starter</artifactId>
            <version>${project.version}</version>
        </dependency>
        <!--
=====
===== 自定义 module -->
        <dependency>
            <groupId>com.zlt</groupId>
            <artifactId>zlt-config</artifactId>
            <version>${project.version}</version>
        </dependency>
        <!--
=====
===== 父工程依赖管理-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>          <!-- Spring Cloud 依赖管理 -->
            <version>${spring-cloud-dependencies.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>

        <dependency>                                         <!-- Spring Boot 依赖管理 -->
            <groupId>org.springframework.boot</groupId>

```

```

<artifactId>spring-boot-dependencies</artifactId>
<version>${spring-boot-dependencies.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-dependencies</artifactId>    <!-- Spring Cloud alibaba 依赖管理 -->
    <version>${spring-cloud-alibaba-dependencies.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>      <!-- maven 打包插件 (子工程引入父工程时, 会引入这个插件导致打包) -->
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <!--可以把依赖的包都打包到生成的 Jar 包中, 默认 goal, mvn package 执行之后, 先将生成的 JAR 重命名为 XXX.origin, 再次使用 spring-boot:repackage 打包生成可执行的 JAR -->
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

<repositories>    # 配置仓库
    <repository>
```

```

<id>aliyun-repos</id>
<url>https://maven.aliyun.com/repository/public</url>
<snapshots>
  <enabled>false</enabled>
</snapshots>
</repository>
</repositories>

```

Custom Starter

Add Necessary Dependencies

Include Spring Boot dependencies required for your starter. In the pom.xml or build.gradle, add:

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>

```

Write Auto-Configuration Classes

Create an auto-configuration class annotated with @Configuration and @ConditionalOnMissingBean to provide beans if they are not already defined.

```

package com.example.mystarter.autoconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyStarterAutoConfiguration {

    @Bean
    public MyCustomService myCustomService() {
        return new MyCustomService();
    }
}

```

Provide Metadata for Auto-Configuration

Add an entry in META-INF/spring.factories (or META-

INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports for newer Spring Boot versions):

In Spring Boot 2.7, the use of the /META-INF/spring.factories file is discouraged, and support for /META-INF/spring.factories will be removed in Spring Boot 3.

[org.springframework.boot.autoconfigure.EnableAutoConfiguration.imports](#)

The EnableAutoConfiguration.imports file is scanned for classes to be used in the auto-configuration phase, which is later in the Spring Boot startup process. It doesn't control or list EnvironmentPostProcessor implementations.

[org.springframework.boot.autoconfigure.AutoConfiguration.imports](#)

[spring.factories](#)

```

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.example.mystarter.autoconfig.MyStarterAutoConfiguration

```

```
spring.factories
1 # Initializers
2 org.springframework.context.ApplicationContextInitializer=\
3 org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer, \
4 org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer
5
6 # Application Listeners
7 org.springframework.context.ApplicationListener=\
8 org.springframework.boot.autoconfigure.BackgroundPreinitializer

spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
com.example.mystarter.autoconfig.MyStarterAutoConfiguration
```

Define Configuration Properties (Optional)

Provide configuration properties for customization.

Use `@ConfigurationProperties` and annotate the class with `@EnableConfigurationProperties`.

```
package com.example.mystarter.config;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "mystarter")
public class MyStarterProperties {
    private String property1;
    private int property2;

    // Getters and Setters
}
```

Enable Configuration Properties

Link the properties to your auto-configuration class:

```
package com.example.mystarter.autoconfig;

import com.example.mystarter.config.MyStarterProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(MyStarterProperties.class)
public class MyStarterAutoConfiguration {

    @Bean
    public MyCustomService myCustomService(MyStarterProperties properties) {
        return new MyCustomService(properties.getProperty1(), properties.getProperty2());
    }
}
```

spring-boot-starter

Concepts

The `spring-boot-starter` dependency is a core starter module in Spring Boot that sets up a basic environment for developing Spring Boot applications.

Purpose

`spring-boot-starter` is designed to simplify Spring Boot application setup by providing essential dependencies and

configurations.

Included Functionality

Spring Core

Dependency: [spring-core](#), [spring-context](#), [spring-beans](#)

Provides fundamental Spring Framework components for dependency injection, bean management, and core container functionality.

Logging

Dependency: [logback-classic](#)

Description: Default logging framework used by Spring Boot, providing logging capabilities and integration with SLF4J.

Configuration

Dependency: [spring-boot-autoconfigure](#), [spring-boot-configuration-processor](#)

Description: Enables automatic configuration of Spring Boot applications based on classpath and property settings.

The [spring-boot-configuration-processor](#) helps in generating configuration metadata.

Classpath

Dependency: [spring-boot-starter](#), [spring-boot-starter-logging](#)

Description: Provides tools and utilities for managing classpath resources, including resource loading, application property management, and environment configuration.

spring-cloud-starter-bootstrap

在 SpringBoot 2.4.x 的版本之后，提供对于 bootstrap.properties/bootstrap.yaml 配置文件的支持，bootstrap 启动器

依赖:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>
```

Loading External Configuration Early

The bootstrap context is initialized before the main Spring application context.

This is crucial for loading external configurations that need to be available before the main context is created.

If you define Spring Cloud Config properties (like `spring.cloud.config.uri`) in `application.yml`,

the application might not be able to connect to the Config Server in time to fetch configuration properties before the main context is built.

For example, when using a configuration server (like Spring Cloud Config Server), the properties from the server need to be available early enough to configure the main application context properly.

```
# bootstrap.yml
spring:
  cloud:
    config:
      uri: http://config-server:8888
      name: myapp
      profile: dev
```

Support for Configuration Sources

It supports various external configuration sources like the Spring Cloud Config Server, Consul, or Zookeeper.

These sources often provide configuration properties that need to be applied early in the application lifecycle, which is what the bootstrap context facilitates.

spring-boot-devtools (development)

依赖: [spring-boot-maven-plugin](#) (也可以用 springloaded 的插件方式 `springboot:run`)

[pom.xml >>](#)

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <!-- devtools 依赖此配置 (否则, devtools 不生效) 。 -->
                <fork>true</fork>
            </configuration>
        </plugin>
    </plugins>
</build>

```

bootstrap.yml >> 有 bootstrap.yml 时要放入其中

```

spring:
  devtools:
    restart:
      enabled: false          #热部署生效
      #trigger-file=.reloadtrigger   #编辑触发重启
      #exclude: WEB-INF/**       #classpath 目录下的 WEB-INF 文件夹内容修改不重启
      #additional-paths: src/main/java #设置重启的目录
  freemarker:
    cache: false    # 页面不加载缓存, 修改即时生效
application.properties >>
spring.devtools.restart.trigger-file=.reloadtrigger
spring.devtools.restart.enabled=false

```

idea >> 需要保存时自动编译就配置

Settings

Search:

- Plugins
- > Version Control
- Build, Execution, Deployment
 - > Build Tools
 - Maven
 - Importing
 - Ignored Files
 - Runner
 - Running Tests
 - Repositories
 - Gradle
 - Gant
 - > Compiler

Build, Execution, Deployment > Compiler

Resource patterns:

`!?*.java;!?*.form;!?*.class;!?*.groovy;!?*.scala;!?*.flex`

Use ; to separate patterns and ! to negate a pattern. Accepted wildcards: ? — symbols; / — path separator; /**/ — any number of directories; <dir_name>:< with the specified name

Clear output directory on rebuild

Add runtime assertions for notnull-annotated methods and parameters Configure annotations

Automatically show first error in editor

Display notification on build completion

Build project automatically

(only works while not running)

Compile independent modules in parallel

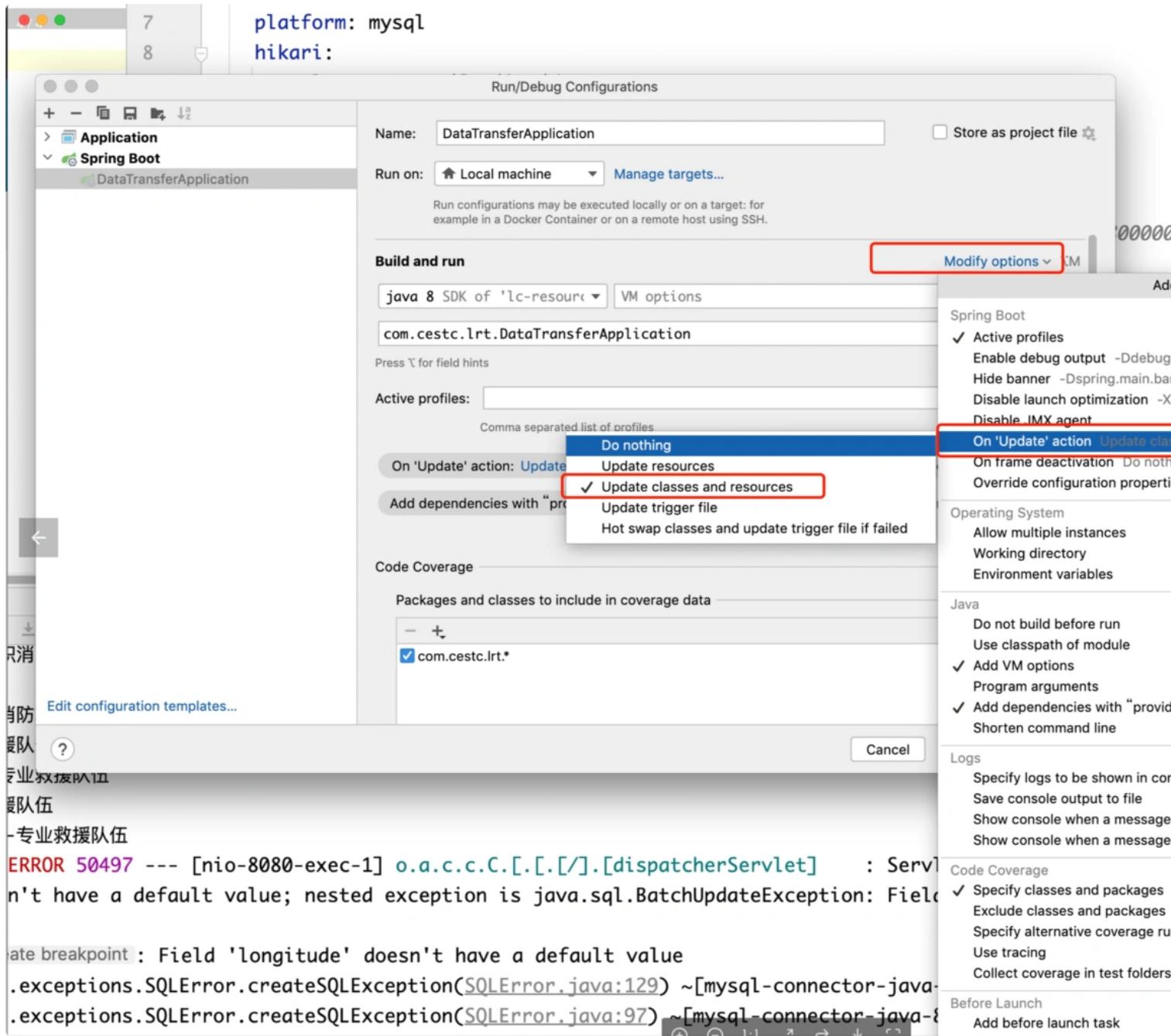
(may require larger heap size)

Rebuild module on dependency change

Shared build process heap size (Mbytes):

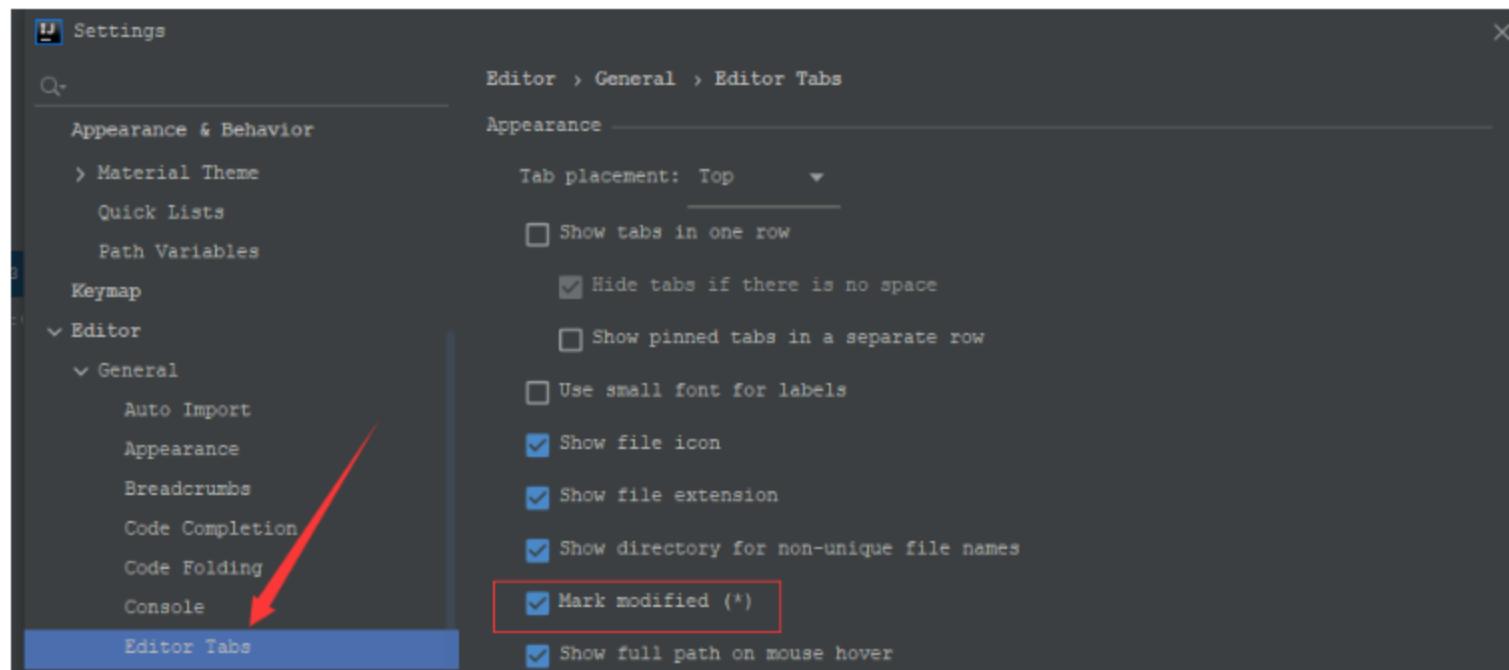
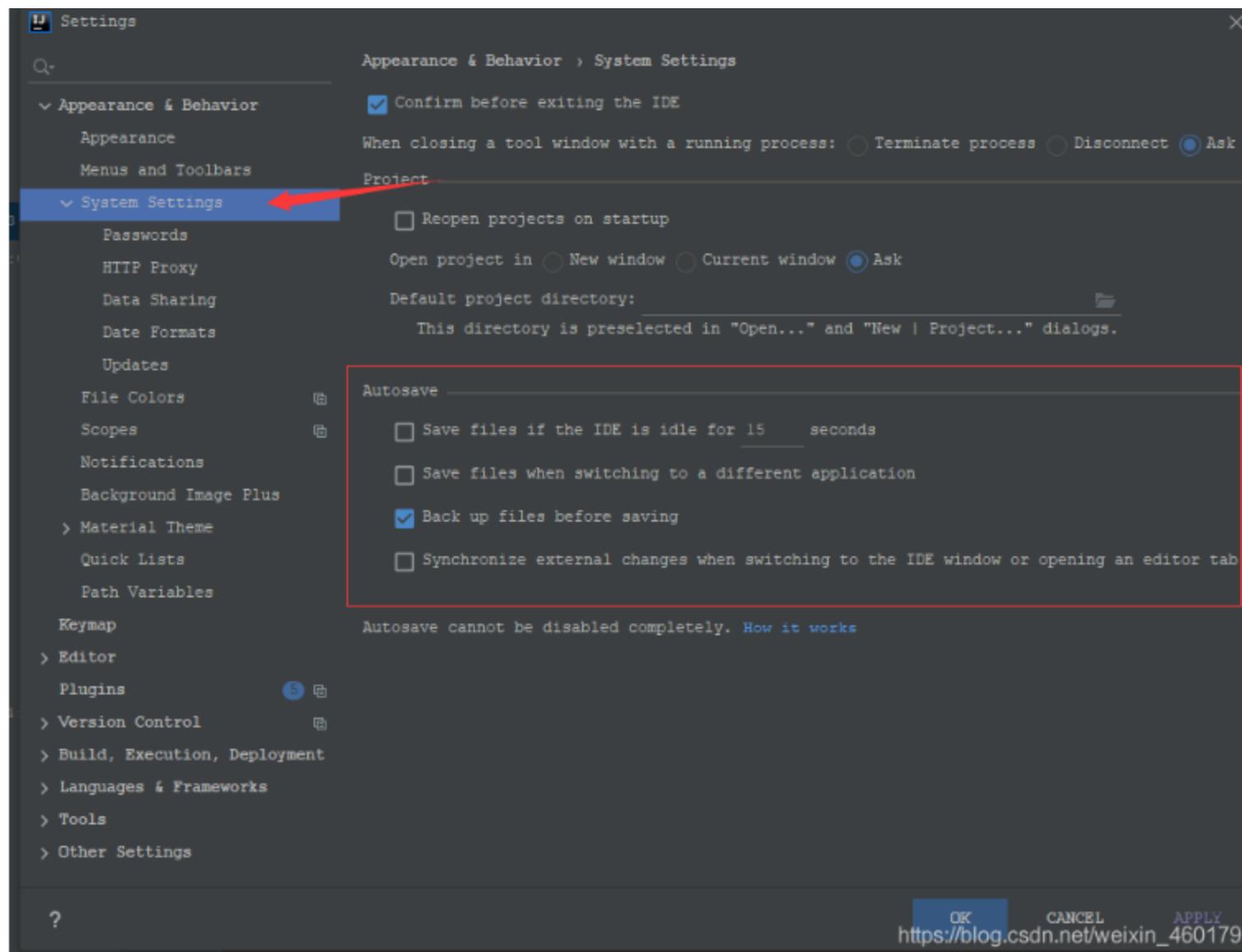
700





Ctrl + Shift + Alt + / compiler.automake.allow.when.app.running

file->setting->advanced setting->勾选 allow auto-make to start ...



spring-boot-starter-actuator (monitor)

Core

依赖:

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

management.endpoints.web.base-path=/manage 修改访问路径

Configuration

ActuatorMetricsConfig

@Configuration

```
public class ActuatorMetricsConfig {    解决 metrics 端点丢失
    @Bean
    InitializingBean forcePrometheusPostProcessor(BeanPostProcessor meterRegistryPostProcessor, PrometheusMeterRegistry
registry) {
        return () -> meterRegistryPostProcessor.postProcessAfterInitialization(registry, "");
    }
}
```

application.properties >

management.endpoints.web.base-path=/manage 访问路径调整

management.endpoints.web.path-mapping.health=healthcheck 健康检查路径调整

management.server.port=10111 访问端口调整

management.endpoint.health.show-details=always 显示健康检查详细信息

management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP 配置系统健康状态类型的严重程度

management.endpoint.shutdown.enabled=true 开发 shutdown

management.endpoints.web.exposure.exclude=beans 用來关闭某些 endpoints

management.endpoints.web.exposure.include=* 开放所有(不包含 shutdown)

management.endpoints.web.exposure.include=beans,mappings 只开放特定的端点

查看所有暴露端点: <http://localhost:8080/actuator>

/metrics

表 16-1 Actuator 端点说明

ID	描 述	是否默认启用
auditevents	公开当前应用程序的审查事件信息	是
beans	显示 Spring IoC 容器关于 Bean 的信息	是
conditions	显示自动配置类的评估和配置条件，并且显示他们匹配或者不匹配的原因	是
configprops	显示当前项目的属性配置信息（通过@ConfigurationProperties 配置）	是
env	显示当前 Spring 应用环境配置属性（ConfigurableEnvironment）	是
flyway	显示已经应用于 flyway 数据库迁移的信息	是
health	显示当前应用健康状态	是
httptrace	显示最新追踪信息（默认为最新 100 次 HTTP 请求）	是
info	显示当前应用信息	是
loggers	显示并修改应用程序中记录器的配置	是
liquibase	显示已经应用于 liquibase 数据库迁移的信息	是
metrics	显示当前配置的各项“度量”指标	是
mappings	显示由@RequestMapping（@GetMapping 和 @PostMapping 等）配置的映射路径信息	是
scheduledtasks	显示当前应用的任务计划	是
sessions	允许从 Spring 会话支持的会话存储库检索和删除用户会话，只是 Spring 会话对响应式 Web 应用还暂时不能支持	是
shutdown	允许当前应用被优雅地进行关闭（在默认的情况下不启用这个端点）	否
threaddump	显示线程泵	是
pow.xml >>		
management:		
endpoint:		
health:		
show-details: always	显示健康详细信息	
endpoints:		
web:		
exposure:		
include: *	将所有的监控 endpoint 暴露出来	

[metrics-core]

MetricRegistry

```
package com.codahale.metrics;
public class MetricRegistry implements MetricSet
```

ScheduledReporter

```
package com.codahale.metrics;
public abstract class ScheduledReporter implements Closeable, Reporter
```

[signalfx-codahale]

MetricMetadata

```
package com.signalfx.codahale.reporter;  
public interface MetricMetadata
```

spring-boot-starter-data-jdbc

依赖

```
spring-boot-starter-data-jdbc  
mysql-connector-java
```

使用

```
TestController >>  
@AutoWired  
JdbcTemplate jdbcTemplate  
Long count= queryForObject( "select * from boss", Long.class )
```

application.yml >

```
spring:  
  datasource:  
    name: druidDataSource  
    type: com.alibaba.druid.pool.DruidDataSource  
    driver-class-name: com.mysql.jdbc.Driver  
    url:  
      jdbc:mysql://123.3.3.3:3306/boss?useUnicode=true&characterEncoding=utf8&useSSL=false&autoReconnect=true  
    username:  
    password:  
  jdbc:  
    template:  
      query-timeout: 3    查询超时 (秒)
```

概念

JDBC (Java DataBase Connectivity java 数据库连接) jdbc 是一种用于执行 SQL 语句的 Java API,可以为多种关系型数据库提供统一访问,它是由一组用 Java 语言编写的类和接口组成的。

实现不同关系型数据库的连接!

数据库连接池 数据库连接池负责分配、管理和释放数据库连接,它允许应用程序重复使用一个现有的数据库连接而不是再重新建立一个。这项技术能明显提高对数据库操作的性能

如果想完成数据库连接池技术,就必须实现 javax.sql.DataSource 接口

DriverManager: 我们不需要通过 DriverManager 调用静态方法 registerDriver(),因为只要 Driver 类被使用,则会执行其静态代码块完成注册驱动

mysql5 之后可以省略注册驱动的步骤,在 jar 包中,存在一个 java.sql.Driver 配置文件,文件中指定了

```

com.mysql.jdbc.Driver
public class MyDataSource implements DataSource {

    //1. 准备容器，用于保存多个连接对象
    private static List<Connection> pool = Collections.synchronizedList(new ArrayList<>());

    //2. 定义静态代码块，通过工具类获取10个连接对象
    static {
        for(int i = 1; i <= 10; i++) {
            Connection con = JDBCUtils.getConnection();
            pool.add(con);
        }
    }

    //3重写getConnection()，用于获取一个连接对象
    @Override
    public Connection getConnection() throws SQLException {
        return null;
    }
}

```

归还数据库连接的方式

继承方式

定义一个类，继承 JDBC4 Connection【不能整体修改驱动包的功能】

```

/*
 * 自定义的连接对象
 * 1. 定义一个类，继承JDBC4Connection
 * 2. 定义Connection连接对象和容器对象的成员变量
 * 3. 通过有参构造方法为成员变量赋值
 * 4. 重写close方法，完成归还连接
 */
public class MyConnection1 extends JDBC4Connection{//1. 定义一个类，继承JDBC4Connection
    //2. 定义Connection连接对象和容器对象的成员变量
    private Connection con;
    private List<Connection> pool;

    //3. 通过有参构造方法为成员变量赋值
    public MyConnection1(String hostToConnectTo, int portToConnectTo, Properties info, String databaseToConnectTo, String url) {
        super(hostToConnectTo, portToConnectTo, info, databaseToConnectTo, url);  I
    }
}

```

装饰设计模式

实现 Connection 接口【大量方法需要重写】

```

/*
1. 定义一个类，实现Connection接口
2. 定义连接对象和连接池容器对象的成员变量
3. 通过有参构造方法为成员变量赋值
4. 重写close方法，完成归还连接
5. 剩余方法，还是调用原有的连接对象中的功能即可

💡
//1. 定义一个类，实现Connection接口
public class MyConnection2 implements Connection{

    //2. 定义连接对象和连接池容器对象的成员变量
    private Connection con;
    private List<Connection> pool;

    //3. 通过有参构造方法为成员变量赋值
    public MyConnection2(Connection con, List<Connection> pool) {
        this.con = con;
        this.pool = pool;
    }
}

```

适配器设计模式

提供一个适配器类，实现 Connection 接口，将所有方法进行实现（除了 close 方法），自定义连接类只需要继承这个中间类就可以了【还是需要在适配器内编写大量代码】

动态代理方式

在不改变目标对象方法的情况下对方法进行增强，通过 Proxy 来完成对 Connection 实现类对象的代理。代理过程中判断如果执行的是 close 方法，就将连接归还池中。如果是其他方法则调用连接对象原来的功能即可。

sprint-boot-starter-data-jpa

Configuration

自动根据数据库表生成实体类： File->Project Structure->Modules 添加 jpa

Views->Tool Windows -> persistence

setting->inspectiongs->JPA issues->query language check

controller

@GetMapping("/driver")

public BaseResponse listAuthenticationDriverInfo(@PageableDefault(value = 15, sort = { "id" }, direction = Sort.Direction.DESC) Pageable pageable, 封装 pageable 对象

表示默认

情况下我们按照 id 倒序排列，每一页的大小为 15

page，第几页，从 0 开始，默认为第 0 页

size，每一页的大小，默认为 20

sort，排序相关的信息，以 property,property(ASC|DESC) 的方式组织，例如 sort=firstname&sort=lastname,desc 表示在按 firstname 正序排列基础上按 lastname 倒序排列。

```

    @ApiParam(value = "状态") @RequestParam(required = false) AuthenticationCheckStatus
status,
    @ApiParam(value = "司机姓名") @RequestParam(required = false) String driverName {
    return new BaseResponse<>(authenticateService.listAuthenticationDriverInfo(pageable, status, driverName));
}

```

```
PageRequest pageable= new PageRequest(page, size);
Page<Users> p = this.usersDao.findAll(pageable);          获取分页数据
p.getTotalElements()  数据的总条数
p.getTotalPages()    数据的总页数
List<Users> list = p.getContent();      数据数组
dao>
public interface AuthenticationDriverDao extends PagingAndSortingRepository<AuthenticationDriver, Long> { 在
CrudRepository 基础上添加分页功能

    AuthenticationDriver findById(String id)    根据 id 返回
}
```

service

```
@Autowired
PortDao portDao;
public List<Shipping> myList(Long shipperId, ListShippingRequest req) {
    Sort sort = Sort.by(Sort.Order.desc("createAt"));          创建筛选
    List<Shipping> myList = Lists.newArrayList(shippingDao.findAll(sort)) .orElse("xx")   发现所有，前方为 null 时返回"xx"
}
```

application.yml

```
spring.jpa.hibernate.ddl-auto =update    数据库没有表则直接创建
spring.jpa.show-sql =true                打印 sql
spring.datasource.url=jdbc:mysql://localhost:3306/springboottest?useUnicode=true&characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.jdbc.batch_size=500
spring.jpa.properties.hibernate.order_inserts=true
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=trace
```

spring-boot-starter-data-mongodb

介绍

介绍

依赖:

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

核心类: org.springframework.data.mongodb.core.MongoOperations
org.springframework.data.mongodb.core.aggregation.Aggregation

org.mongodb.driver.cluster (optional)

```
failed:           Cluster description not yet available. Waiting for 36666 ms before timing out
success:         Adding discovered server maas-gt-p118-56662.nam.nsroot.net:37617 to client view of cluster
```

org.springframework.boot.actuate.data.mongo.MongoHealthIndicator

```
failed:           MongoDB health check failed
```

```

# 注意下的这种方式不能连接
# 因为此方法默认连接的权限机制是 mechanism=SCRAM-SHA-256
#spring:
# data:
#   mongodb:
#     host: 192.168.108.60
#     port: 27017
#     username: mason
#     password: 123456
#     database: appdb

spring:
  data:
    mongodb:
      # mongodb://用户名:密码@IP 地址:27017/数据库
      uri: mongodb://mason:123456@192.168.108.60:27017/appdb // This appdb must be the database where the current
logged in user was created.
      # database: appdb # 可以不用设置数据库

```

[mongodb-driver-core]
internal

SocketStream

```

package com.mongodb.internal.connection;
public class SocketStream implements Stream      socket 流
                                                 client
package com.mongodb.client.model;
public final class Indexes 索引工具
public static Bson ascending(String... fieldNames)    返回索引

```

[mongodb-driver-sync]
client

MongoClients

```

package com.mongodb.client;
public final class MongoClients 客户端连接工具
public static MongoClient create(String connectionString) 使用连接字符串连接到 mongodb // MongoClient
mongoClient = MongoClients.create("mongodb://smp:smp@127.0.0.1:27017");

```

MongoClient

```

package com.mongodb.client;
public interface MongoClient extends Closeable 客户端连接
MongoDatabase getDatabase(String var1);    获取数据库 // mongoClient.getDatabase("test_db")

```

MongoDatabase

```

package com.mongodb.client;
public interface MongoDatabase 数据库对象
MongoCollection<Document> getCollection(String var1);  获取集合

```

MongoCollection

```
package com.mongodb.client;

public interface MongoCollection<TDocument> 集合对象
InsertManyResult insertMany(List<? extends TDocument> var1);
InsertManyResult insertMany(List<? extends TDocument> var1, InsertManyOptions var2);
InsertManyResult insertMany(ClientSession var1, List<? extends TDocument> var2);
InsertManyResult insertMany(ClientSession var1, List<? extends TDocument> var2, InsertManyOptions var3);
DeleteResult deleteOne(Bson var1);
DeleteResult deleteOne(Bson var1, DeleteOptions var2);
DeleteResult deleteOne(ClientSession var1, Bson var2);
DeleteResult deleteOne(ClientSession var1, Bson var2, DeleteOptions var3);
DeleteResult deleteMany(Bson var1);
DeleteResult deleteMany(Bson var1, DeleteOptions var2);
DeleteResult deleteMany(ClientSession var1, Bson var2);
DeleteResult deleteMany(ClientSession var1, Bson var2, DeleteOptions var3);
TDocument findOneAndDelete(Bson var1);
TDocument findOneAndDelete(Bson var1, FindOneAndDeleteOptions var2);
TDocument findOneAndDelete(ClientSession var1, Bson var2);
TDocument findOneAndDelete(ClientSession var1, Bson var2, FindOneAndDeleteOptions var3);
void drop();    删除集合
FindIterable<TDocument> find();    找到所有数据
long countDocuments();    统计数量
String createIndex(Bson var1);    创建索引
void dropIndex(String var1);    删除索引
ListIndexesIterable<Document> listIndexes();    列出所有索引
```

FindIterable

```
package com.mongodb.client;
public interface FindIterable<TResult> extends Mongolterable<TResult>    结果遍历器
```

ListIndexesIterable

```
package com.mongodb.client;
public interface ListIndexesIterable<TResult> extends Mongolterable<TResult>    索引结果遍历器
```

Mongolterable

```
package com.mongodb.client;
public interface Mongolterable<TResult> extends Iterable<TResult>    mongo 返回结果遍历器
MongoCursor<TResult> iterator();    获取遍历器
```

MongoCursor

```
package com.mongodb.client;
public interface MongoCursor<TResult> extends Iterator<TResult>, Closeable    遍历器指针
boolean hasNext();
TResult next();
```

model

Filters

```
package com.mongodb.client.model;
```

```
public final class Filters 筛选语句
    筛选集合: collection.deleteOne(Filters.eq("code", "manager"));
public static <TItem> Bson eq(@Nullable TItem value)
public static <TItem> Bson eq(String fieldName, @Nullable TItem value)
```

[bson]

```
package org.bson;
public class Document implements Map<String, Object>, Serializable, Bson mongo 文档对象
public Document(String key, Object value) 简易文档对象
public Document(Map<String, ?> map) 复杂键值对文档对象
```

spring-boot-starter-mail (email)

```
spring-context-support
org.springframework.mail.javamail.JavaMailSender
```

application.yml >>

```
# email
spring:
  mail:
    host: smtp.163.com
    username: 13227379709@163.com
    password: abcd1234
    properties:
      mail:
        smtp:
          auth: true
        starttls:
          enable: true
          required: true
```

使用

MainController >>

```
@Autowired
private JavaMailSender sender; 发送消息

@GetMapping("/send")
public String sendMail(){
    SimpleMailMessage message = new SimpleMailMessage(); 邮件消息
    message.setFrom("13227379709@163.com"); 发出邮箱
    message.setTo("784516419@qq.com"); 目标邮箱
    message.setSubject("title"); 主题
    message.setText("content"); 内容
    sender.send(message);
    return "ok";
}
```

spring-boot-starter-hateoas

spring-boot-admin-starter-server (monitor)

管理和监控 spring-boot 程序

admin-server

@EnableAdminServer 在引导类上启用监控功能

admin-client

spring-boot-starter-security

Concept

Default login page: user = <generated password>

Core

Spring Security is a powerful and highly customizable [authentication and access control framework](#) for Java applications, particularly those built using the Spring Framework.

It provides a comprehensive solution for securing applications by handling common security requirements such as [authentication](#) (verifying user identity) and [authorization](#) (granting access based on user roles and permissions).

Components

Authentication

How It Works

Spring Security [checks the provided credentials](#) (like a username and password) against a user store (such as a database or an LDAP server).

If the credentials match, the user is considered authenticated.

Supported Methods

Form-Based Login

The user [submits a form with credentials](#), which are then verified.

HTTP Basic/Digest Authentication

The user's credentials [are sent with every request](#) and verified by the server.

OAuth2 and OpenID Connect

Supports authentication [through external providers](#) like Google, Facebook, or custom OAuth2 servers.

Custom Authentication Providers

Developers [can create custom authentication mechanisms](#) to suit specific needs.

Outcome

If authentication is successful, a [SecurityContext](#) is created and stored, typically in the session, to keep track of the user's authentication state.

Authorization

How It Works

Once a user is authenticated, Spring Security [checks the user's roles or authorities](#) against the required permissions for accessing certain resources (like web pages or API endpoints).

Types of Authorization

URL-Based Authorization

Controls access to web resources based on URL patterns.

For example, /admin/** can be restricted to users with the ADMIN role.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/admin/**").hasRole("ADMIN") // Only ADMIN can access /admin/**
```

```

        .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN") // USER and ADMIN can
access /user/*
            .anyRequest().authenticated() // All other requests require authentication
        )
        .formLogin(Customizer.withDefaults());
    return http.build();
}

```

Method-Level Authorization

Uses annotations like `@PreAuthorize` or `@Secured` to control access to individual methods in your code based on user roles or permissions.

```

@Service
public class AdminService {

    @PreAuthorize("hasRole('ADMIN')")
    public void performAdminTask() {
        // Only users with ADMIN role can execute this method
    }
}

```

Role-Based Access Control (RBAC)

Defines access rules based on roles assigned to users, such as `ROLE_USER`, `ROLE_ADMIN`, etc.

```

@Bean
public UserDetailsService userDetailsService() {
    UserDetails admin = User.withDefaultPasswordEncoder()
        .username("admin")
        .password("admin123")
        .roles("ADMIN") // Assign ADMIN role
        .build();

    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user")
        .password("user123")
        .roles("USER") // Assign USER role
        .build();

    return new InMemoryUserDetailsManager(admin, user);
}

```

Attribute-Based Access Control (ABAC)

Uses custom attributes and policies to define fine-grained access control.

```

@PreAuthorize("#user.department == 'IT' or hasRole('ADMIN')")
public void accessITResources(User user) {
    // Only users from IT department or with ADMIN role can access
}

```

Outcome

If the user has the necessary permissions, they are allowed to access the resource.

If not, access is denied, typically resulting in an `HTTP 403 Forbidden` response.

Password Management

Offers password encoding and storage mechanisms, including `bcrypt`, `pbkdf2`, and other secure hashing algorithms.

Supports password policy enforcement, such as complexity requirements and expiration rules.

Session Management

Manages user sessions, including session timeouts, maximum concurrent sessions, and session fixation protection.

Provides options for tracking session state and handling session-related events.

Protection Against Common Vulnerabilities

Protects against common security threats such as:

Cross-Site Request Forgery (CSRF)

Provides CSRF protection tokens to prevent unauthorized actions.

Cross-Origin Resource Sharing (CORS)

Configures CORS policies to control how web pages from different domains can interact with your application.

Clickjacking

Prevents clickjacking attacks by setting appropriate HTTP headers (e.g., X-Frame-Options).

Man-in-the-Middle (MITM) Attacks

Enforces HTTPS and secure transport protocols.

Customizable Security Filters

Uses a flexible chain of filters to handle security-related tasks, such as authentication, authorization, session management, and more.

Allows customization and extension of these filters to suit specific security needs.

Declarative Security Configuration

Supports declarative security configuration using Java annotations or XML, making it easy to define security rules without requiring a lot of code.

Offers both global and fine-grained security control over application components.

Auditing and Logging

Provides comprehensive logging and auditing capabilities for security events, such as authentication attempts, access denials, and session creation/destruction.

Useful for monitoring, troubleshooting, and compliance purposes.

Configuration

Add Dependencies

```
<dependencies>
    <!-- Spring Boot Starter Security -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
        <version>3.4.2</version>
    </dependency>
</dependencies>
```

Application Properties

Set some properties as needed.

```
spring.security.user.name=user
spring.security.user.password=password
```

SecurityFilterChain (Bean)

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            // Define URL-based authorization
            .authorizeHttpRequests(auth ->
                auth
                    .requestMatchers("/public/**").permitAll()
                    // Public endpoints
                    // .requestMatchers is used in spring Security 5.8+ (modern API)
                    // Spring Security automatically triggers authentication when authorization header is
                    // present even though you've added ".permitAll()"
                    .requestMatchers("/admin/**").hasRole("ADMIN")
            )
    }
}
```

```

        // Admin-only endpoints
        .anyRequest().authenticated()
            // All other endpoints require authentication
    )
// Enables form-based login (Default login method)
.formLogin(formLogin ->
    formLogin
        .loginPage("/login").permitAll()
            // Custom login page
        .defaultSuccessUrl("/home", true)
            // Redirect to home page on successful login
        .failureUrl("/login?error=true")
            // Redirect to login page on failure
)
// Enables logout functionality
.logout(logout ->
    logout
        .logoutUrl("/logout").permitAll()
            // Custom logout URL
        .logoutSuccessUrl("/login?logout=true")
            // Redirect to login page on logout
            // (If both logoutSuccessHandler and logoutSuccessUrl set are set,
            // logoutSuccessHandler will override logoutSuccessUrl.)
        .logoutSuccessHandler(new CustomLogoutSuccessHandler())
            // Custom logout handler
        .invalidateHttpSession(true)
            // Invalidate session on logout
        .deleteCookies("JSESSIONID")
            // Delete cookies on logout
)
// Enables HTTP Basic authentication
.httpBasic(Customizer.withDefaults())
// Enables OAuth2 login
.oauth2Login(Customizer.withDefaults())
    Configure OAuth2 provider details in application.yml:
    spring:
        security:
            oauth2:
                client:
                    registration:
                        google:
                            client-id: your-client-id
                            client-secret: your-client-secret
                            scope: email, profile
// Define OAuth2 configuration
.oauth2ResourceServer(
    oauth2 -> oauth2.jwt(jwt -> jwt.jwkSetUri("https://your-auth-server/.well-known/jwks.json"))
        // Configures the application as an OAuth 2.0 resource server and specifies that
        // JWT tokens will be used for authentication.
        // Ensure that the application fetches the signing key from the authorization
        // server.
)
// Enable CSRF protection
.csrf(csrf ->
    csrf
        .ignoringAntMatchers("/h2-console/**") // Disable CSRF for H2 console
)
.sessionManagement(session ->
    session
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
// Using a custom filter
.addFilterBefore(new CustomAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class)
    // You can explicitly configure the authentication filters to bypass requests to /api/simi/**.
// Enable frame options to allow H2 console to be embedded in an iframe
.exceptionHandling(ex ->
    ex
        .authenticationEntryPoint(new HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED)))
.headers(headers ->

```

```

        headers
            .frameOptions().sameOrigin() // Allow same-origin iframes
        );
    return http.build();
}
}

```

SecurityConfig (Configuration)

By default, Spring Boot will automatically enable Spring Security, and it will provide basic HTTP authentication out of the box. However, you may want to customize the security configuration.

Create a custom security configuration class by extending [WebSecurityConfigurerAdapter](#) (for Spring Security 5.6 and earlier) or using [SecurityFilterChain](#) (for Spring Security 5.7 and later).

Matching Rule

/api

Only the exact path /api

/api/v1/**

Any path that starts with /api/v1/.

```

/api/v1/
/api/v1/foo
/api/v1/foo/bar
/api/v1/foo/bar/baz

```

But does not match:

/api/v1 (exact match is not included)

/api/v2/foo (different version)

/api/foo (wrong path)

/api/*

Paths with exactly one level after /api/

```

/api/123
/api/sub

```

But does not match:

/api/ (trailing slash)

/api/foo/bar (nested path)

/api/v1/foo (subdirectories)

```

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // Define URL-based authorization
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .antMatchers("/public/**").permitAll()
                        // Allow access to public URLs
                        // antMatchers is used in older versions (deprecated in 5.8+)
                    .antMatchers("/admin/**").hasRole("ADMIN")
                        // Restrict access to admin URLs
                    .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
                        // USER and ADMIN can access /user/*
                    .anyRequest().authenticated()
                        // All other URLs require authentication
            )
    }
}

```

```

// Define form login configuration
.formLogin(formLogin ->
    formLogin
        .loginPage("/login").permitAll()
            // Custom login page
        .defaultSuccessUrl("/home", true)
            // Redirect to home page on successful login
        .failureUrl("/login?error=true")
            // Redirect to login page on failure
)
// Define OAuth2 configuration
.oauth2ResourceServer(oauth2ResourceServer ->
    oauth2ResourceServer.jwt()          // Configures the application as an OAuth 2.0 resource server
and specifies that JWT tokens will be used for authentication.
)
// Define logout configuration
.logout(logout ->
    logout
        .logoutUrl("/logout").permitAll()
            // Custom logout URL
        .logoutSuccessUrl("/login?logout=true")
            // Redirect to login page on logout
            (If both logoutSuccessHandler and logoutSuccessUrl setare set, logoutSuccessHandler
            will override logoutSuccessUrl.)
        .invalidateHttpSession(true)
            // Invalidate session on logout
        .deleteCookies("JSESSIONID")
            // Delete cookies on logout
)
// ignoringAntMatchers is Deprecated in Spring Security 5.8+
.logoutSuccessHandler(new CustomLogoutSuccessHandler())
// Enable CSRF protection
.csrf(csrf ->
    csrf
        .ignoringAntMatchers("/h2-console/**") // Disable CSRF for H2 console
)
// Enable frame options to allow H2 console to be embedded in an iframe
.headers(headers ->
    headers
        .frameOptions().sameOrigin() // Allow same-origin iframes
);
}

// Configure in-memory authentication for demonstration purposes
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user").password(passwordEncoder().encode("password")).roles("USER")
        .and()
        .withUser("admin").password(passwordEncoder().encode("admin")).roles("ADMIN");
}

// Define a password encoder bean
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(); // Use BCrypt as the password hashing algorithm
}
}

```

CustomAuthenticationProvider (Component)

Custom Authentication Providers allow you to define custom authentication mechanisms that are not supported by default.

This can be useful for integrating with external systems or implementing custom authentication logic.

Why Use Custom Authentication Providers?

External Systems: Integrate with third-party authentication systems.

Complex Authentication Logic: Apply complex authentication rules.

Legacy Systems: Support legacy authentication mechanisms.

```
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Component;

@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {

    @Autowired
    private final UserDetailsService userDetailsService;

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();

        // Custom authentication logic
        UserDetails user = userDetailsService.loadUserByUsername(username);
        if (user == null || !password.equals(user.getPassword())) {
            throw new BadCredentialsException("Invalid username or password");
        }
        return new UsernamePasswordAuthenticationToken(user, password, user.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

AuthenticationManager (Bean)

Register CustomAuthenticationProvider

```
@Bean
public AuthenticationManager authenticationManager(HttpSecurity http, CustomAuthenticationProvider provider)
throws Exception {
    return new ProviderManager(List.of(provider));
}
```

Custom Login Page (Optional)

If you want to use a custom login page, create an HTML page named `login.html` in the `src/main/resources/templates` directory:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Login</title>
</head>
<body>
    <h1>Login</h1>
    <form th:action="@{/login}" method="post">
        <div>
            <label>Username:</label>
            <input type="text" name="username" />
        </div>
```

```

<div>
    <label>Password:</label>
    <input type="password" name="password" />
</div>
<div>
    <button type="submit">Login</button>
</div>
</form>
</body>
</html>

```

LogoutSuccessHandler

If you have [custom logout logic](#) or need to perform additional actions when a user logs out, you can use Spring Security's `LogoutHandler` or `LogoutSuccessHandler`:

```

import org.springframework.security.core.Authentication;
import org.springframework.security.web.authentication.logout.LogoutSuccessHandler;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class CustomLogoutSuccessHandler implements LogoutSuccessHandler {

    @Override
    public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response,
                               Authentication authentication) throws IOException, ServletException {
        // Custom logic here
        response.sendRedirect("/login?logout");
    }
}

```

UserDetailsService (in memory - default)

To define a simple in-memory user for testing purposes, you can create a `UserDetailsService` bean.

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder() // Use a password encoder
            .username("user")
            .password("password")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}

```

UserDetailsService (db)

Instead of storing user credentials in memory, you can configure a JDBC or JPA-based `UserDetailsService` to fetch user details from a database.

This is more suitable for production environments.

Set Up the Database Schema

Create a table to store user credentials. For example:

```
-- Stores the username, password (hashed), and whether the user is enabled.
CREATE TABLE users (
    username VARCHAR(50) NOT NULL PRIMARY KEY,
    password VARCHAR(100) NOT NULL,
    enabled BOOLEAN NOT NULL
);

-- Stores the authorities or roles assigned to the user.
CREATE TABLE authorities (
    username VARCHAR(50) NOT NULL,
    authority VARCHAR(50) NOT NULL,
    FOREIGN KEY (username) REFERENCES users(username)
);
```

Configure DataSource and UserDetailsService

Configure a `JdbcUserDetailsService` to handle user authentication from the database.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.JdbcUserDetailsService;
import org.springframework.security.web.SecurityFilterChain;

import javax.sql.DataSource;

@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        // Configure JdbcUserDetailsService with custom queries if needed
        JdbcUserDetailsService userDetailsService = new JdbcUserDetailsService(dataSource);
        userDetailsService.setUsersByUsernameQuery(
            "select username, password, enabled from users where username = ?"
        );
        userDetailsService.setAuthoritiesByUsernameQuery(
            "select username, authority from authorities where username = ?"
        );
        return userDetailsService;
    }
}
```

Testing the Security Configuration

Open a web browser and navigate to `http://localhost:8080`. You will be redirected to the login page.

Enter the username and password defined in the `UserDetailsService` bean (`user` and `password` in this case).

After successful login, you will be able to access protected resources.

SecurityConfig

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    //The Username/Password Authentication only need SecurityFilterChain and UserDetailsService bean.
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests((authorize) -> authorize.anyRequest().authenticated())
            .authorizeHttpRequests((authorize) -> authorize.requestMatchers("/login").permitAll()
                .anyRequest().authenticated());
        .httpBasic(Customizer.withDefaults())
        .formLogin(Customizer.withDefaults());

        return http.build();
    }
}
```

```

}

@Bean
public UserDetailsService userDetailsService() {
    UserDetails userDetails = User.withDefaultPasswordEncoder()
        .username("user")
        .password("password")
        .roles("USER")
        .build();

    return new InMemoryUserDetailsManager(userDetails);
}

// AuthenticationManager bean to allow for custom authentication
@Bean
public AuthenticationManager authenticationManager(
    UserDetailsService userDetailsService,
    PasswordEncoder passwordEncoder) {
    DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
    authenticationProvider.setUserDetailsService(userDetailsService);
    authenticationProvider.setPasswordEncoder(passwordEncoder);

    return new ProviderManager(authenticationProvider);
}

@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
}

```

Service

```

@Service
public class MyServiceImpl implements MyService{      自定义安全配置 access 认证逻辑
    @Override
    public boolean hasPermission(HttpServletRequest request, Authentication authentication){
        String uri=request.getRequestURI();
        Object principal = authentication.getPrincipal();      获取用户对象
        if( principal instanceof UserDetails ){
            UserDetails userDetails = (UserDetails) principal;
            Collection<? extends GrantedAuthority> authorities = userDetails.getAuthorities();  新建 包含 GrantedAuthority 泛型
            的集合
            return authorities.contains(new SimpleGrantedAuthority(uri));  判断是否包含权限
        }
        return false;
    }
}

@Service
public class UserServiceImpl implements UserDetailsService{  自定义登录逻辑，实现后不会生成密码了
    @Autowired
    private PasswordEncoder pw;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException{
        if( !"admin".equals(username) ){  根据 username 查询数据库
            throw new UsernameNotFoundException("用户名或密码错误")
        }
        String password = pw.encode("123456")  根据查询的对象比较密码
    }
}
```

```

        return new User("admin", password, AuthorityUtils.commaSeparatedStringToAuthorityList("admin,normal,ROLE_abc" ) );
给用户生成两个权限，添加一个角色
    }
}

otherService >>
public interface MyService{
    boolean hasPermission(HttpServletRequest request, Authentication authentication);
}

acl
SecurityContext securityContext = SecurityContextHolder.getContext(); 上下文对象（可能为 null，需要判断）
Authentication authentication = securityContext.getAuthentication(); 从上下文中获取认证对象（可能为 null，需要判断）

PasswordEncoder
PasswordEncoder pw = new BCryptPasswordEncoder() 哈希加密
String encode = pw.encode("123456") 单向加密
boolean matches = pw.matches("123456", encode) 验证是否匹配

注解模式
@EnableGlobalMethodSecurity( securedEnabled =true ) 开启注解

controller>
@GetMapping("/driver")
@Secured("ROLE_abc") 登录后判断是否是角色，必须 ROLE_ 开头，区分大小写
@PreAuthorize( "hasRole('ROLE_USER') or hasRole('ROLE_ADMIN')" ) 只有 ROLE_USER 角色或 ROLE_ADMIN 角色的用户才能调用下方方法（ROLE_ 加不加都可以）
@PreAuthorize( "#id<10" ) 限制只能查询 Id 小于 10 的用户
@PreAuthorize( "principal.username.equals(#username)" ) 限制只能查询自己的信息 // public User find(String username) {
    @PreAuthorize( "#user.name.equals('abc')" ) 限制只能新增用户名称为 abc 的用户
    @PreAuthorize("true") 放行所有
    @PostAuthorize( "returnObject.id%2==0" ) 在方法调用完成后进行权限检查，它不能控制方法是否能被调用，只能在方法调用完成后检查权限决定是否要抛出
    @PreFilter( filterTarget="ids", value="filterObject%2==0" ) 对集合类型的参数或返回值进行过滤 //public void delete(List<Integer> ids, List<String> usernames) {
        @PostFilter( "filterObject.id%2==0" ) 对集合类型的参数或返回值进行过滤 // List<User> userList = new ArrayList<User>(); return userList;
    }

    public BaseResponse listAuthenticationDriverInfo(@PageableDefault Pageable pageable,
                                                    @ApiParam(value = "状态") @RequestParam(required = false) AuthenticationCheckStatus status,
                                                    @ApiParam(value = "司机姓名") @RequestParam(required = false) String driverName) {
        return new BaseResponse<>(authenticateService.listAuthenticationDriverInfo(pageable, status, driverName));
    }
}

spring-boot-starter-websocket
Core

```

How it Works Under the Hood

WebSocket Protocol Handling

The Spring WebSocket support manages the WebSocket protocol, including the initial handshake over HTTP, which upgrades the connection to the WebSocket protocol.

Integration with Web Servers

Spring Boot integrates with web servers like Tomcat, Jetty, or Undertow.

These servers provide the actual socket implementation.

When you use `spring-boot-starter-websocket`, Spring Boot configures the embedded web server to handle WebSocket connections.

Netty Integration

For high-performance scenarios, Spring can also use Netty, which is an asynchronous event-driven network application framework.

Netty provides a rich set of APIs to work with sockets.

Abstraction Layer

Spring WebSocket provides an abstraction layer that hides the complexity of direct socket programming.

The higher-level classes and interfaces in Spring WebSocket internally use socket classes to manage connections, read/write data, and handle events.

Example Flow

1. When a client initiates a WebSocket connection, the request is handled by the web server (e.g., Tomcat or Netty).
2. The server performs the WebSocket handshake and, upon success, upgrades the connection to a WebSocket connection.
3. The Spring WebSocket framework then manages the WebSocket session, routing messages to the appropriate handlers (e.g., `TextWebSocketHandler`).

Underlying Socket Classes

The following underlying socket classes are typically involved:

- Java NIO Classes
 - If you are using a web server like Tomcat or Jetty, they might use Java NIO (`java.nio.channels.SocketChannel`) to handle non-blocking IO operations.
- Netty Classes
 - If you are using Netty, it will use its own socket classes (e.g., `io.netty.channel.socket.SocketChannel`) for managing connections.

Cofiguration

Add Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>stomp-websocket</artifactId>
    <version>2.3.4</version>
</dependency>
```

Configure WebSocket

Create a configuration class to set up WebSocket endpoints and message brokers.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;
```

```

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws").setAllowedOrigins("*").withSockJS();
    }
}

```

Create a Message Handling Controller

Define a controller to handle incoming WebSocket messages and send responses.

```

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;
import org.springframework.web.util.HtmlUtils;

@Controller
public class WebSocketController {

    @MessageMapping("/hello")
    @SendTo("/topic/greetings")
    public Greeting greeting(HelloMessage message) throws Exception {
        Thread.sleep(1000); // simulated delay
        return new Greeting("Hello, " + HtmlUtils.htmlEscape(message.getName()) + "!");
    }
}

```

Define the HelloMessage and Greeting classes.

```

public class HelloMessage {
    private String name;

    public HelloMessage() {
    }

    public HelloMessage(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

public class Greeting {
    private String content;

    public Greeting() {
    }
}

```

```

public Greeting(String content) {
    this.content = content;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}
}

```

Create WebSocket Client

Create an HTML file with JavaScript to connect to the WebSocket server and send/receive messages.

```

<!DOCTYPE html>
<html>
<head>
<title>WebSocket Test</title>
<script src="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.5.1/sockjs.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js"></script>
<script type="text/javascript">
    var stompClient = null;

    function setConnected(connected) {
        document.getElementById('connect').disabled = connected;
        document.getElementById('disconnect').disabled = !connected;
        document.getElementById('conversationDiv').style.visibility = connected ? 'visible' : 'hidden';
        document.getElementById('response').innerHTML = '';
    }

    function connect() {
        var socket = new SockJS('/ws');
        stompClient = Stomp.over(socket);
        stompClient.connect({}, function (frame) {
            setConnected(true);
            console.log('Connected: ' + frame);
            stompClient.subscribe('/topic/greetings', function (greeting) {
                showGreeting(JSON.parse(greeting.body).content);
            });
        });
    }

    function disconnect() {
        if (stompClient !== null) {
            stompClient.disconnect();
        }
        setConnected(false);
        console.log('Disconnected');
    }

    function sendName() {
        var name = document.getElementById('name').value;
        stompClient.send('/app/hello', {}, JSON.stringify({'name': name}));
    }

    function showGreeting(message) {
        var response = document.getElementById('response');
        var p = document.createElement('p');
        p.appendChild(document.createTextNode(message));
        response.appendChild(p);
    }
</script>
</head>
<body>
<div>

```

```

<button id="connect" onclick="connect();">Connect</button>
<button id="disconnect" onclick="disconnect();" disabled="disabled">Disconnect</button>
</div>
<div id="conversationDiv">
  <input type="text" id="name" placeholder="Enter your name">
  <button id="send" onclick="sendName();">Send</button>
  <p id="response"></p>
</div>
</body>
</html>

```

spring-boot-starter-webflux

Core

Default Server Choices in Spring WebFlux

Netty

This is the most commonly used server with Spring WebFlux. It's a non-blocking, event-driven server that works well with the reactive paradigm.

When you use Spring WebFlux without specifying a server, it [defaults to Netty](#).

This means if you include the `spring-boot-starter-webflux` dependency and don't explicitly configure a server, Spring Boot will automatically configure Netty as the server.

Tomcat

Spring WebFlux [can also run on Tomcat](#), though Tomcat is traditionally used for blocking, servlet-based applications.

When used with WebFlux, Tomcat operates in a non-blocking mode.

Jetty

Like Netty, Jetty is another server option that supports non-blocking, reactive programming.

Changing the Server

If you want to use a different server, such as Tomcat or Jetty, you can do so by including the corresponding dependency in your project:

For Tomcat:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>

```

For Jetty:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>

```

Netty

Netty is a [high-performance, asynchronous, event-driven](#) network application framework that simplifies the development of network applications such as protocol servers and clients.

It provides a [comprehensive set of tools](#) for building scalable and efficient network applications.

Key Features

Asynchronous and Non-blocking

Netty uses [asynchronous I/O operations](#) to handle multiple connections efficiently.

This means it doesn't block threads [while waiting for I/O operations to complete](#), allowing the server to handle a large

number of concurrent connections with fewer resources.

Event-driven Architecture

Netty is built around an event-driven model where events like connection establishment, data reception, and disconnection are handled by event loops.

This architecture helps in writing efficient, non-blocking network code.

High Performance

Netty is optimized for performance and can handle a high volume of network traffic with low latency. It achieves this through efficient use of resources and high-throughput network I/O.

Flexible and Extensible

Netty provides a flexible API that allows developers to customize and extend its functionality.

It supports a wide range of protocols and can be easily adapted to various use cases.

Components

Event Loop

The core of Netty's event-driven model.

Event loops manage threads that handle I/O operations and dispatch events to the appropriate handlers.

Each event loop processes events for a set of channels (connections).

Netty provides two types of event loop groups:

Boss Event Loop Group

Handles incoming connection requests.

Worker Event Loop Group

Handles I/O operations for the accepted connections.

Channel

Represents a connection to a network entity. It is used to perform I/O operations such as reading from and writing to the network.

ChannelHandler

Handles inbound and outbound data. There are two main types:

Inbound Handlers

Process incoming data from the network.

Outbound Handlers

Handle outgoing data to the network.

ChannelPipeline

A sequence of ChannelHandler instances that process data as it flows through the pipeline.

Handlers are added to or removed from the pipeline as needed.

Handlers are added to the pipeline to handle specific tasks, such as encoding/decoding, business logic, and logging.

Bootstrap

Used to set up and configure a Netty server or client. It provides various options for configuring the channel, event loop groups, and other settings.

You start by configuring a ServerBootstrap for server applications or Bootstrap for client applications.

Configuration

Add Dependencies

Ensure that your pom.xml or build.gradle includes the necessary dependencies for Spring WebFlux.

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

Gradle:

```
implementation 'org.springframework.boot:spring-boot-starter-webflux'
```

Configure WebFlux Properties

You can configure various aspects of WebFlux, such as codecs, thread pools, etc., by extending WebFluxConfigurer or using properties in application.properties or application.yml.

```
spring:  
  webflux:  
    base-path: /api  
    session:  
      timeout: 30m
```

Create a Reactive Controller

Create a controller using the @RestController annotation, similar to a traditional Spring MVC controller, but return reactive types like Mono and Flux instead of String or List.

```
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
{@RestController  
public class ReactiveController {  
  
    @GetMapping("/mono")  
    public Mono<String> getMono() {  
        return Mono.just("Hello, Mono!");  
    }  
  
    @GetMapping("/flux")  
    public Flux<String> getFlux() {  
        return Flux.just("Hello", "from", "Flux");  
    }  
}}
```

Run the Application

Run your Spring Boot application as usual. The reactive endpoints will be available, and you can test them via HTTP clients or curl.

Optional Configuration

Add a WebFilter

Create a WebFilter

Here's an example of a simple WebFilter that logs incoming requests and responses:

In Spring Boot WebFlux, if you mark your filter class with @Component, it **will automatically be registered** in the application context, and no additional manual configuration is needed.

The above LoggingWebFilter class will be picked up by Spring Boot at runtime and applied to every incoming HTTP request.

```
import org.springframework.stereotype.Component;  
import org.springframework.web.server.WebFilter;  
import org.springframework.web.server.WebFilterChain;  
import reactor.core.publisher.Mono;  
import org.springframework.web.server.ServerWebExchange;  
  
{@Component  
public class LoggingWebFilter implements WebFilter {  
  
    @Override  
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {  
        // Log the request URI before processing  
        System.out.println("Incoming request: " + exchange.getRequest().getURI());  
  
        // Continue the filter chain (invoke the next filter or controller)  
        return chain.filter(exchange).doOnSuccess((done) -> {  
            // Log after the response is processed  
        });  
    }  
}}
```

```
        System.out.println("Outgoing response: " + exchange.getResponse().getStatusCode());
    });
}
}
```

Applying Filters Programmatically (if needed)

Although `@Component` is the most straightforward approach, you can also programmatically register filters using the `WebFilter` in your `WebFluxConfigurer`.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.config.WebFluxConfigurer;
import org.springframework.web.server.WebFilter;

@Configuration
public class WebFluxConfig implements WebFluxConfigurer {

    @Override
    public void addWebFilters(org.springframework.web.reactive.config.WebFluxConfigurer registry) {
        registry.addWebFilter(new LoggingWebFilter());
    }
}
```

Use Reactive Repositories

If you're working with data, you might also want to use reactive data repositories like `ReactiveMongoRepository` or `ReactiveCrudRepository`.

```
import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import reactor.core.publisher.Mono;

public interface ReactiveUserRepository extends ReactiveCrudRepository<User, String> {
    Mono<User> findByUsername(String username);
}
```

Handle Reactive Streams

Spring WebFlux natively supports reactive streams, so you can process, transform, and combine streams using operators like `map`, `flatMap`, `filter`, etc.

```
@GetMapping("/users")
public Flux<User> getAllUsers() {
    return userRepository.findAll()
        .filter(user -> user.isActive())
        .map(user -> transformUser(user));
}
```

Add a Test Case

You can write tests using `@WebFluxTest` for your controllers. Spring Boot provides `WebTestClient` for testing WebFlux applications in a reactive way.

```
@WebFluxTest(ReactiveController.class)
public class ReactiveControllerTest {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    public void testMonoEndpoint() {
        webTestClient.get().uri("/mono")
            .exchange()
            .expectStatus().isOk()
```

```
        .expectBody(String.class).isEqualTo("Hello, Mono!");  
    }  
}
```

[reactor-core]

MonoOperator

```
package reactor.core.publisher;  
public abstract class MonoOperator<I, O> extends Mono<O> implements Scannable  
    A decorating Mono Publisher that exposes Mono API over an arbitrary Publisher.  
    Useful to create operators which return a Mono.
```

Mono

```
package reactor.core.publisher;  
public abstract class Mono<T> implements CorePublisher<T>
```

Represents a single asynchronous result in reactive programming.

In the Reactive Streams specification, a Mono is a specialized Publisher that emits at most one item or completes empty (without emitting any items).

It can also signal an error if the computation fails.

Key Characteristics

Single Element

It either emits one element or completes without emitting any.

Automatic Subscription

When you return a Mono from a controller method, Spring WebFlux automatically subscribes to the Mono for you, handling the processing and sending the response back to the client.

Non-blocking

This approach remains non-blocking and efficient, allowing other requests to be processed while waiting for the Mono to complete.

Response Handling

The response is handled based on the result emitted by the Mono.

If it completes successfully, the value is sent back to the client. If an error occurs (e.g., if you return a Mono.error()), Spring WebFlux will handle the error response appropriately.

```
public static <T> Mono<T> just(T data)
```

Emit a single value.

```
    Mono<String> mono = Mono.just("Hello, World!");
```

```
public static <T> Mono<T> justOrEmpty(@Nullable T data)
```

```
public static <T> Mono<T> empty()
```

Emit no value, just complete.

```
    Mono<String> emptyMono = Mono.empty();
```

```
public static <T> Mono<T> error(Throwable error)
```

Emit an error.

```
    Mono<String> errorMono = Mono.error(new RuntimeException("Something went wrong"));
```

```
public static <T> Mono<T> fromCallable(Callable<? extends T> supplier)
```

Convert a Callable into a Mono.

```
Mono<String> callableMono = Mono.fromCallable(() -> "Result from a callable");
```

Mono can be used in an asynchronous context, such as handling HTTP requests or making database calls, while keeping the application reactive and non-blocking.

```
Mono<String> asyncMono = Mono.fromCallable(() -> {
    // Simulate a long-running task (like a database call)
    Thread.sleep(1000);
    return "Task result";
});
```

```
public static <T> Mono<T> fromFuture(CompletableFuture<? extends T> future)
```

Create a Mono, producing its value using the provided CompletableFuture.

```
public static <T> Mono<T> fromSupplier(Supplier<? extends T> supplier)
```

Create a Mono from a Supplier.

```
Mono<String> supplierMono = Mono.fromSupplier(() -> "Hello from Supplier");
```

```
public final Mono<T> switchIfEmpty(Mono<? extends T> alternate)
```

Fallback to an alternative Mono if this mono is completed without data

```
public final <R> Mono<R> map(Function<? super T, ? extends R> mapper)
```

Transform the item emitted by a Mono.

```
Mono<Integer> mono = Mono.just("5").map(Integer::parseInt);
```

```
public final <R> Mono<R> flatMap(Function<? super T, ? extends Mono<? extends R>> transformer)
```

Takes a function that maps an element from the original Mono to another Mono.

Applies this function to the element of the original Mono

Flatten the resulting Mono of Monos into a single Mono.

```
Mono<Integer> mono = Mono.just("5").flatMap(s -> Mono.just(Integer.parseInt(s)));
```

```
public final <R> Flux<R> flatMapMany(Function<? super T, ? extends Publisher<? extends R>> mapper)
```

Maps to a multi-value Publisher and flattens to a Flux<R>

```
public final Mono<T> filter(final Predicate<? super T> tester)
```

Conditionally process the emitted item.

```
Mono<Integer> filteredMono = Mono.just(10).filter(num -> num > 5);
```

```
public final Mono<T> contextWrite(Function<Context, Context> contextModifier)
```

The Mono.contextWrite method in Project Reactor is used to enrich the reactive context within the reactive stream, allowing you to propagate contextual data (such as metadata or thread-local information) without directly passing it through method parameters.

The context is available only within this stream.

This is useful in scenarios like logging, tracing, security, or any other cross-cutting concern where contextual data is required.

```
Mono<String> mono = Mono.just("Hello")
    .contextWrite(Context.of("key", "value"))           // Apply context with contextWrite
    .flatMap(data -> Mono.deferContextual(ctx -> {      // Retrieve and use context with
        deferContextual(
            String contextValue = ctx.get("key");
            return Mono.just(data + " with context value: " + contextValue);
        );
    })
    .mono.subscribe(System.out::println);          // Subscribe to trigger the execution
```

Steps

Define Context

You create and define some contextual data to be propagated in the reactive pipeline.

Retrieve Context

Use `Mono.deferContextual` to access the context data later in the reactive chain.

Apply Context

Use `contextWrite` to associate context with a reactive operation.

```

public final Mono<T> doOnNext(Consumer<? super T> onNext)
    Add side-effect behavior when a value is emitted.
    Mono<String> mono = Mono.just("Hello").doOnNext(val -> System.out.println("Value: " + val));
public final Mono<T> doOnError(Consumer<? super Throwable> onError)
    Add behavior triggered when the Mono completes with an error.
public final Mono<T> onErrorResume(Function<? super Throwable, ? extends Mono<? extends T>> fallback)
    Handle errors and provide a fallback value when an error occurs.
public final Mono<T> onErrorReturn(T fallback)
    Alternates the element that causes error to a fallback value given as a parameter.
public final Mono<T> onErrorMap(Function<? super Throwable, ? extends Throwable> mapper)
    Transforms the received exception to another exception.
public final <V> Mono<V> thenReturn(V value)
    Let this Mono complete successfully, then emit the provided value.
    On an error in the original Mono, the error signal is propagated instead.
public final Mono<Void> then()
    Chain and execute another Mono after the current one completes.
    Mono<Void> mono = Mono.just("Task done").then(Mono.empty());
public final <T2> Mono<Tuple2<T, T2>> zipWhen(Function<T, Mono<? extends T2>> rightGenerator)
    Wait for the result from this mono, use it to create a second mono via the provided rightGenerator function and
    combine both results into a Tuple2.
public final Disposable subscribe()

```

Basic Subscription

Calling subscribe() without parameters triggers the execution of the Mono immediately,
but it won't handle or respond to any emitted values, errors, or completion events.
Calling any of the subscribe methods causes the Mono to execute immediately, regardless of whether any
arguments are passed.

```

Mono<String> mono = Mono.just("Hello, World!");

// Triggers the Mono, but does nothing with the result
mono.subscribe();

```

Mono with Error Handling and Fallback

You can use Mono.onErrorResume() to handle errors and provide a fallback value when an error occurs.

```

Mono<String> monoWithError = Mono.error(new RuntimeException("Failure!"))
    .onErrorResume(error -> Mono.just("Fallback Value"));
monoWithError.subscribe(
    value -> System.out.println("Received: " + value), // Success (or fallback)
    error -> System.err.println("Error: " + error) // Will not be called, because of
onErrorResume
);

```

```
public final Disposable subscribe(Consumer<? super T> consumer)
```

Handling the Success Case

You can pass a single Consumer<T> to handle the emitted value when the Mono completes successfully.

This version allows you to handle the success case by providing a consumer that processes the emitted value.

```

Mono<String> mono = Mono.just("Hello, World!");

// Print the value emitted by the Mono
mono.subscribe(value -> System.out.println("Received: " + value));

```

```
public final Disposable subscribe(@Nullable Consumer<? super T> consumer, Consumer<? super Throwable>
errorConsumer)
```

Handling Success and Error Cases

You can pass two Consumers—one to handle the success case, and one to handle errors.

```

Mono<String> mono = Mono.just("Hello, World!");

// Handling both success and error cases
mono.subscribe(
    value -> System.out.println("Received: " + value),           // Success
    error -> System.err.println("Error: " + error)             // Error
);

```

```

public final Disposable subscribe(
    @Nullable Consumer<? super T> consumer,
    @Nullable Consumer<? super Throwable> errorConsumer,
    @Nullable Runnable completeConsumer
)

```

Handling Success, Error, and Completion

You can also handle the completion event separately by providing a third Runnable argument.

```

Mono<String> mono = Mono.just("Hello, World!");

// Handling success, error, and completion
mono.subscribe(
    value -> System.out.println("Received: " + value),           // Success
    error -> System.err.println("Error: " + error),             // Error
    () -> System.out.println("Completed!")                      // Completion
);

```

Flux

```

package reactor.core.publisher;
public abstract class Flux<T> implements CorePublisher<T>

```

Represents a reactive stream that can emit zero or more items (potentially unbounded) and can either complete successfully or with an error.

It is a more general reactive type than Mono<T>, which represents a stream with at most one item.

Key Characteristics

Multiple elements

Unlike Mono, a Flux can emit multiple elements, including zero, some, or an infinite number.

Asynchronous

It can handle data in a non-blocking, asynchronous fashion.

Reactive Streams compliant

Follows the Reactive Streams specification, which means it supports backpressure, a key concept in reactive programming to avoid overwhelming consumers with too many events.

Termination

A Flux can complete successfully after emitting zero or more items, or it can terminate with an error.

```

public static <T> Flux<T> just(T data)

```

Emit multiple elements.

```
    Flux<String> flux = Flux.just("Spring", "Reactor", "Flux");
```

```
public static <T> Flux<T> fromIterable(Iterable<? extends T> it)
```

Create a Flux from a Collection.

```
    List<String> data = Arrays.asList("Spring", "Reactor", "WebFlux");
    Flux<String> flux = Flux.fromIterable(data);
```

```
public static Flux<Integer> range(int start, int count)
```

Emit a range of numbers.

```
    Flux<Integer> flux = Flux.range(1, 5); // Emits 1, 2, 3, 4, 5
```

```
public static <T> Flux<T> empty()
```

Emit no items and complete immediately.

```
    Flux<String> flux = Flux.empty();
```

```

public static <T> Flux<T> error(Throwable error)
    Emit an error.
    Flux<String> flux = Flux.error(new RuntimeException("Error occurred"));

public final <V> Flux<V> map(Function<? super T, ? extends V> mapper)
    Transform the emitted elements.
    Flux<Integer> flux = Flux.just(1, 2, 3).map(i -> i * 2); // Emits 2, 4, 6
public final <R> Flux<R> flatMap(Function<? super T, ? extends Publisher<? extends R>> mapper)
    Asynchronously transform elements into Flux (or Mono).
    Flux<String> flux = Flux.just("A", "B", "C")
        .flatMap(letter -> Flux.just(letter.toLowerCase(), letter.toUpperCase()));
public final <R> Flux<R> flatMapIterable(Function<? super T, ? extends Iterable<? extends R>> mapper)
    If an exception is thrown within the flatMapIterable, the Flux will complete with the error.

public final Flux<T> filter(Predicate<? super T> p)
    Filter elements based on a condition.
    Flux<Integer> filteredFlux = Flux.range(1, 10).filter(i -> i % 2 == 0); // Emits even numbers
public final Flux<T> take(long n)
    Limit the number of emitted items.
    Flux<Integer> limitedFlux = Flux.range(1, 10).take(3); // Emits 1, 2, 3 and completes
public static <T1, T2, O> Mono<O> zip(Mono<? extends T1> p1, Mono<? extends T2> p2, BiFunction<? super T1, ? super T2, ? extends O> combinator)
    Combine two or more Flux sources element by element.
    Flux<String> letters = Flux.just("A", "B", "C");
    Flux<Integer> numbers = Flux.just(1, 2, 3);
    Flux<String> zipped = Flux.zip(letters, numbers, (letter, number) -> letter + number);
        // Emits A1, B2, C3
public static <I> Flux<I> merge(Publisher<? extends I>... sources)
    Merge multiple Flux streams into one.
    Flux<String> mergedFlux = Flux.merge(Flux.just("A", "B"), Flux.just("C", "D"));
        // Emits A, B, C, D

public static Flux<Long> interval(Duration period)
    Emit items at regular time intervals (useful for streaming or polling).
    Flux<Long> intervalFlux = Flux.interval(Duration.ofSeconds(1)); // Emits a value every second
    Flux<Long> flux = Flux.interval(Duration.ofSeconds(1)).take(5); // Emits 5 values every second
    flux.subscribe(System.out::println);
public final Flux<T> delayElements(Duration delay)
    Introduces a delay between the emission of each item in the stream
    Flux.range(1, 10).delayElements(Duration.ofSeconds(1));

public final Mono<T> doOnNext(Consumer<? super T> onNext)
    Perform side-effects on each emitted item.
    Flux<String> flux = Flux.just("A", "B", "C")
        .doOnNext(val -> System.out.println("Value: " + val)); // Prints each value
public final Flux<T> onErrorResume(Function<? super Throwable, ? extends Publisher<? extends T>> fallback)
    Subscribe to a returned fallback publisher when any error occurs, using a function to choose the fallback depending on the error.
public final void subscribe(Subscriber<? super T> actual)

```

[reactor-extra]

CacheMono

package reactor.cache;

public class CacheMono

```
public static <KEY, VALUE> MonoCacheBuilderCacheMiss<KEY, VALUE> lookup( Function<KEY, Mono<Signal<? extends VALUE>>> reader, KEY key)
```

Restore a Mono<VALUE> from the reader Function (see below) given a provided key.

If no value is in the cache, it will be calculated from the original source which is set up in the next step.

public interface MonoCacheBuilderCacheMiss<KEY, VALUE>

```
default MonoCacheBuilderCacheWriter<KEY, VALUE> onCacheMissResume(Mono<VALUE> other)
```

Setup original source to fallback to in case of cache miss.

public interface MonoCacheBuilderCacheWriter<KEY, VALUE>

```
Mono<VALUE> andWriteWith(BiFunction<KEY, Signal<? extends VALUE>, Mono<Void> > writer);
```

Set up the cache writer BiFunction to use to store the source data into the cache in case of cache miss.

[reactor-netty]

TcpClient

```
package reactor.netty.tcp;
```

public abstract class TcpClient

```
public static TcpClient create()
```

```
public final <T> TcpClient option(ChannelOption<T> key, @Nullable T value)
```

```
public final TcpClient doOnConnected(Consumer<? super Connection> doOnConnected)
```

Connection

```
package reactor.netty;
```

```
@FunctionalInterface
```

public interface Connection extends DisposableChannel

```
default Connection addHandlerLast(ChannelHandler handler)
```

[netty-handler]

ReadTimeoutHandler

```
package io.netty.handler.timeout;
```

public class ReadTimeoutHandler extends IdleStateHandler

WriteTimeoutHandler

```
package io.netty.handler.timeout;
```

public class WriteTimeoutHandler extends ChannelOutboundHandlerAdapter

[spring-webflux]

ExchangeFilterFunction

```
package org.springframework.web.reactive.function.client;
```

```
@FunctionalInterface
```

public interface ExchangeFilterFunction

Represents a function that filters an exchange function.

The filter is executed when a Subscriber subscribes to the Publisher returned by the WebClient.

```
Mono<ClientResponse> filter(ClientRequest request, ExchangeFunction next);
```

```
default ExchangeFilterFunction andThen(ExchangeFilterFunction afterFilter)
```

```
default ExchangeFunction apply(ExchangeFunction exchange)
```

```
static ExchangeFilterFunction ofRequestProcessor(Function<ClientRequest, Mono<ClientRequest>> processor)
static ExchangeFilterFunction ofResponseProcessor(Function<ClientResponse, Mono<ClientResponse>> processor)
```

WebFluxConfigurer

```
package org.springframework.web.reactive.config;
```

```
public interface WebFluxConfigurer
```

Defines **callback methods** to customize the configuration for WebFlux applications enabled via `@EnableWebFlux`.

```
default void configureContentTypeResolver(RequestedContentTypeResolverBuilder builder)
default void addCorsMappings(CorsRegistry registry)
default void configurePathMatching(PathMatchConfigurer configurer)
default void addResourceHandlers(ResourceHandlerRegistry registry)
default void configureArgumentResolvers(ArgumentResolverConfigurer configurer)
default void configureHttpMessageCodecs(ServerCodecConfigurer configurer)
default void addFormatters(FormatterRegistry registry)
```

Add **custom Converters and Formatters** for performing type conversion and formatting of **annotated controller method arguments**.

```
@Nullable
```

```
default Validator getValidator()
```

```
@Nullable
```

```
default MessageCodesResolver getMessageCodesResolver()
```

```
@Nullable
```

```
default WebSocketService getWebSocketService()
```

```
default void configureViewResolvers(ViewResolverRegistry registry)
```

spring-statemachine-core

Configuration

Add Dependencies

```
<dependency>
  <groupId>org.springframework.statemachine</groupId>
  <artifactId>spring-statemachine-core</artifactId>
  <version>3.2.0</version>
</dependency>
```

Define States and Events

Define the states and events for your state machine.

For example, you might have an order processing workflow with states like ORDER_PLACED, ORDER_CONFIRMED, and ORDER_SHIPPED, and events like CONFIRM_ORDER and SHIP_ORDER.

```
public enum OrderStates {
    ORDER_PLACED,
    ORDER_CONFIRMED,
    ORDER_SHIPPED
}

public enum OrderEvents {
    CONFIRM_ORDER,
    SHIP_ORDER
}
```

Configure the State Machine

Create a configuration class to set up the state machine. Use annotations like @EnableStateMachine to enable state machine support in your Spring Boot application.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.statemachine.config.EnableStateMachine;
import org.springframework.statemachine.config.EnumStateMachineConfigurerAdapter;
import org.springframework.statemachine.config.builders.StateMachineConfigurationConfigurer;
import org.springframework.statemachine.config.builders.StateMachineStateConfigurer;
import org.springframework.statemachine.config.builders.StateMachineTransitionConfigurer;

@Configuration
@EnableStateMachine
public class StateMachineConfig extends EnumStateMachineConfigurerAdapter<OrderStates, OrderEvents> {

    @Override
    public void configure(StateMachineConfigurationConfigurer<OrderStates, OrderEvents> config) throws
Exception {
        config.withConfiguration()
            .autoStartup(true);
    }

    @Override
    public void configure(StateMachineStateConfigurer<OrderStates, OrderEvents> states) throws Exception {
        states.withStates()
            .initial(OrderStates.ORDER_PLACED)
            .state(OrderStates.ORDER_CONFIRMED)
            .state(OrderStates.ORDER_SHIPPED);
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<OrderStates, OrderEvents> transitions) throws
Exception {
        transitions
            .withExternal()
            .source(OrderStates.ORDER_PLACED).target(OrderStates.ORDER_CONFIRMED).event(OrderEvents.CONFI
RM_ORDER)
            .and()
            .withExternal()
            .source(OrderStates.ORDER_CONFIRMED).target(OrderStates.ORDER_SHIPPED).event(OrderEvents.SHIP
_ORDER);
    }
}
```

State Machine Listeners

Implement StateMachineListener to log information when the state machine transitions between states or handles events.

```
import org.springframework.statemachine.listener.StateMachineListenerAdapter;
import org.springframework.statemachine.state.State;
import org.springframework.statemachine.transition.Transition;
import org.springframework.stereotype.Component;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Component
public class StateMachineListener extends StateMachineListenerAdapter<OrderStates, OrderEvents> {

    private static final Logger logger = LoggerFactory.getLogger(StateMachineLogger.class);
    private StateMachine<OrderStates, OrderEvents> stateMachine;
    private
    public setStateMachine(StateMachine stateMachine){
        this.stateMachine = stateMachine.
    }

    @Autowired
    private OrderRepository orderRepository;
```

```

@Override
public void stateChanged(State<OrderStates, OrderEvents> from, State<OrderStates, OrderEvents> to) {
    StateMachine<OrderStates, OrderEvents> stateMachine = (StateMachine<OrderStates, OrderEvents>)
from.getStateMachine();
    Order order = (Order) stateMachine.getExtendedState().getVariables().get("order");
    if (order != null) {
        order.setStatus(to.getId());
        orderRepository.save(order);
    }
    logger.info("State changed from " + from.getId() + " to " + to.getId());
}

@Override
public void transition(Transition<OrderStates, OrderEvents> transition) {
    StateMachine<OrderStates, OrderEvents> stateMachine = (StateMachine<OrderStates, OrderEvents>)
transition.getStateMachine();
    Order order = (Order) stateMachine.getExtendedState().getVariables().get("order");
    if (order != null) {
        System.out.println("Transitioning order: " + order.getId() + " from " +
                           transition.getSource().getId() + " to " + transition.getTarget().getId());
    }
    logger.info("Transition: " + transition.getSource().getId() + " -> " +
transition.getTarget().getId());
}

@Override
public void stateMachineStarted(StateMachine<OrderStates, OrderEvents> stateMachine) {
    logger.info("State machine started");
}

@Override
public void stateMachineStopped(StateMachine<OrderStates, OrderEvents> stateMachine) {
    logger.info("State machine stopped");
}
}

```

Order Entity

```

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    private OrderStates status;

    // Other fields like order details, customer info, etc.

}

```

OrderService

Here's the service class that will use the state machine to handle state transitions and update the Order status in the database:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.statemachine.StateMachine;
import org.springframework.statemachine.config.StateMachineFactory;

```

```

import org.springframework.statemachine.support.DefaultStateMachineContext;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private StateMachineFactory<OrderStates, OrderEvents> stateMachineFactory;

    @Autowired
    private StateMachineListener stateMachineListener;

    @Transactional
    public void confirmOrder(Long orderId) {
        Order order = orderRepository.findById(orderId).orElseThrow(() -> new RuntimeException("Order not found"));
        StateMachine<OrderStates, OrderEvents> stateMachine = build(order);

        // Send the CONFIRM_ORDER event to transition to ORDER_CONFIRMED state
        stateMachine.sendEvent(OrderEvents.CONFIRM_ORDER);

        // Update the order status in the database
        order.setStatus(stateMachine.getState().getId());
        orderRepository.save(order);
    }

    @Transactional
    public void shipOrder(Long orderId) {
        Order order = orderRepository.findById(orderId).orElseThrow(() -> new RuntimeException("Order not found"));
        StateMachine<OrderStates, OrderEvents> stateMachine = build(order);

        // Send the SHIP_ORDER event to transition to ORDER_SHIPPED state
        stateMachine.sendEvent(OrderEvents.SHIP_ORDER);

        // Update the order status in the database
        order.setStatus(stateMachine.getState().getId());
        orderRepository.save(order);
    }

    private StateMachine<OrderStates, OrderEvents> build(Order order) {
        StateMachine<OrderStates, OrderEvents> stateMachine = stateMachineFactory.getStateMachine();
        // Create a new listener instance for each state machine

        // StateMachineListenerAdapter<OrderStates, OrderEvents> listener = new
        StateMachineListenerAdapter<>() {
            // Implement event handling logic specific to this state machine instance
            // };

        // Register the listener with the state machine
        // stateMachine.addStateListener(listener);

        stateMachine.addStateListener(stateMachineListener);

        // Start the state machine and set it to the current state of the order
        stateMachine.stop();
        stateMachine.getStateMachineAccessor().doWithAllRegions(
            accessor -> accessor.resetStateMachine(
                new DefaultStateMachineContext<>(order.getStatus(), null, null, null)
            )
        );
        stateMachine.start();
    }
}

```

```

        stateMachine.getExtendedState().getVariables().put("order", order);
        return stateMachine;
    }
}

```

Persistence

Activemq

Concept

Add Dependencies

```

<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-broker</artifactId>
    <version>5.17.4</version> <!-- Use the latest version -->
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>

```

Configure ActiveMQ

```

spring:
  activemq:
    broker-url: tcp://localhost:61616
    user: admin
    password: admin
    packages:
      trust-all: true
      This option is used to specify whether all Java packages should be trusted for deserialization.
      Setting this to true can be convenient but poses security risks, as it can allow deserialization of potentially
      malicious classes. It is safer to explicitly trust only specific packages.

    in-memory: false
      Setting this to true will start an embedded in-memory broker. This is useful for testing purposes when you don't
      want to depend on an external broker.

  pool:
    enabled: true
      Enables connection pooling, which can improve performance by reusing connections rather than creating new
      ones for each operation.

    max-connections: 10
      Specifies the maximum number of connections that can be allocated by the connection pool.

    idle-timeout: 30000
      The maximum amount of time (in milliseconds) that a pooled connection can remain idle before being removed
      from the pool.

    expiration-check-interval: 1000
      The interval (in milliseconds) at which the pool checks for expired connections.

```

Create a Message Producer

JmsTemplate-based Producer

The JmsTemplate class provided by Spring simplifies the process of sending messages to a JMS destination. It abstracts many

of the complexities involved in JMS communication.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class Producer {

    @Autowired
    private JmsTemplate jmsTemplate;

    public void sendMessage(String destination, String message) {
        jmsTemplate.convertAndSend(destination, message);
    }
}
```

JmsTemplate Customization

You can customize the JmsTemplate to specify delivery modes, priorities, and time-to-live for messages.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

import javax.jms.Message;

@Component
public class CustomJmsTemplateProducer {

    @Autowired
    private JmsTemplate jmsTemplate;

    public void sendMessage(String destination, String message) {
        jmsTemplate.execute(session -> {
            Message jmsMessage = session.createTextMessage(message);
            // Set additional properties or headers
            jmsTemplate.convertAndSend(destination, jmsMessage);
            return null;
        });
    }
}
```

Async Producer with JmsTemplate

You can also send messages asynchronously using Spring's JmsTemplate. This can be useful for improving performance by offloading message sending to a separate thread.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

@Component
public class AsyncJmsTemplateProducer {

    @Autowired
    private JmsTemplate jmsTemplate;

    @Async
    public void sendAsyncMessage(String destination, String message) {
        jmsTemplate.send(destination, new MessageCreator() {
            @Override
```

```

        public Message createMessage(Session session) throws JMSException {
            return session.createTextMessage(message);
        }
    });
}

@Async
public void sendAsyncMessage2(String destination, String message) {
    Destination destination = new ActiveMQQueue(queue);
    jmsTemplate.send(destination, session -> {
        ObjectMessage objectMessage = session.createObjectMessage(JsonUtils.format(object));
        objectMessage.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY, time);
        return objectMessage;
    });
}
}

```

Producer with JMS API

If you need more control over the JMS connection and session, you can [use the JMS API directly to create producers](#).

```

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class JmsApiProducer {

    @Autowired
    private ConnectionFactory connectionFactory;

    public void sendMessage(String destination, String message) throws JMSException {
        try (Connection connection = connectionFactory.createConnection()) {
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            Topic topic = session.createTopic(destination);
            MessageProducer producer = session.createProducer(topic);
            TextMessage textMessage = session.createTextMessage(message);
            producer.send(textMessage);
        }
    }
}

```

Create a Message Consumer

JmsListener Annotation

The `@JmsListener` annotation is the simplest and most commonly used method for creating message consumers in Spring Boot. It allows you to annotate methods to handle incoming messages.

```

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class Consumer {

    @JmsListener(destination = "test-queue")
    public void receiveMessage(String message) {
        System.out.println("Received message: " + message);
    }
}

```

```
    }
}
```

MessageListener Interface

Implementing the `MessageListener` interface provides more control over the message consumption process, allowing you to handle messages asynchronously.

```
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
import org.springframework.stereotype.Component;

@Component
public class MessageListenerConsumer implements MessageListener {

    @Override
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            try {
                String text = ((TextMessage) message).getText();
                System.out.println("Received message: " + text);
            } catch (JMSEException e) {
                e.printStackTrace();
            }
        }
    }
}
```

JmsTemplate

Using `JmsTemplate`, you can synchronously receive messages. This approach is less common for continuous message consumption but useful for on-demand message polling.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
public class JmsTemplateConsumer {

    @Autowired
    private JmsTemplate jmsTemplate;

    @PostConstruct
    public void receiveMessage() {
        String message = (String) jmsTemplate.receiveAndConvert("test-queue");
        System.out.println("Received message: " + message);
    }
}
```

DefaultMessageListenerContainer

The `DefaultMessageListenerContainer` is a more advanced and flexible option for consuming messages, offering features like dynamic scaling and transaction management.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.listener.DefaultMessageListenerContainer;

@Configuration
public class JmsConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
        DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
        factory.setConcurrency("1-10");
    }
}
```

```
        factory.setSessionAcknowledgeMode(Session.CLIENT_ACKNOWLEDGE);
        return factory;
    }
}
```

Other

依赖: `spring-boot-starter-activemq` (`spring-boot-starter`, `spring-jms`, `activemq-broker<activemq-client, openwire-legacy, guava, jackson-databind>`, `jms-api`)

`activemq-pool` 消息队列连接池

生产者/消费者配置项

`application.yml >>`

`spring:`

`activemq:`

`broker-url: tcp://192.168.22.129:15672`

`in-memory: true`

`user: active`

`password: active`

`close-timeout: 15s` # 在考虑结束之前等待的时间【删除】

`in-memory: true` # 默认代理 URL 是否应该在内存中。如果指定了显式代理，则忽略此值。

`non-blocking-redelivery: false` # 是否在回滚回滚消息之前停止消息传递。这意味着当启用此命令时，消息顺序不会被保留。【删除】

`send-timeout: 0` # 等待消息发送响应的时间。设置为 0 等待永远。【删除】

`queue-name: active.queue` # 【删除】

`topic-name: active.topic.name.model` # 【删除】

`pool:`

`enabled: true`

`max-connections: 80` #连接池最大连接数

`idle-timeout: 30000` #空闲的连接过期时间，默认为 30 秒

`jms:`

`pub-sub-domain: false`

`listener:`

`concurrency: 20`

使用

`activeMqUtil >>`

`@Component`

`public class ActiveMqUtil {`

`@Autowired`

`private JmsTemplate jmsTemplate;`

`protected final Log logger = LoggerFactory.getLog(ActiveMqUtil.class);`

```

public void sendMessage(String queue, Object object) {
    logger.info(">>>>发送 queue:" + queue + "消息" + JsonUtils.format(object));
    Destination destination = new ActiveMQQueue(queue);
    this.jmsTemplate.convertAndSend(destination, JsonUtils.format(object));
}

public void sendMessage(String queue, Object object, Long time) {
    logger.info(">>>>发送 queue:" + queue + "消息" + JsonUtils.format(object) + ",延时" + (time / 1000L / 60L) + "分钟");
    Destination destination = new ActiveMQQueue(queue);
    jmsTemplate.send(destination, session -> {
        ObjectMessage objectMessage = session.createObjectMessage(JsonUtils.format(object));
        objectMessage.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY, time);
        return objectMessage;
    });
}

public void sendMessage(String queue, Object object, String time) {
    Date date = DateUtils.stringToDate(time, DateUtils.TIME_PATTERN);
    Long nowTime = System.currentTimeMillis();
    Long dateTime = date.getTime();
    if (nowTime >= dateTime) {
        sendMessage(queue, object);
        return;
    }
    logger.info(">>>>发送 queue:" + queue + "消息" + JsonUtils.format(object) + ",定时发送 : " + time + "corn : " + time);
    Destination destination = new ActiveMQQueue(queue);
    jmsTemplate.send(destination, session -> {
        ObjectMessage objectMessage = session.createObjectMessage(JsonUtils.format(object));
        objectMessage.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY, (dateTime -
System.currentTimeMillis()));
        return objectMessage;
    });
}

public void sendMessageTime(String queue, Object object, String time) {
    Date date = DateUtils.stringToDate(time, DateUtils.DATE_TIME_PATTERN);
    Long nowTime = System.currentTimeMillis();
    Long dateTime = date.getTime();
    if (nowTime >= dateTime) {
        sendMessage(queue, object);
        return;
    }
    logger.info(">>>>发送 queue:" + queue + "消息" + JsonUtils.format(object) + ",定时发送 : " + time + "corn : " + time);
    Destination destination = new ActiveMQQueue(queue);
    jmsTemplate.send(destination, session -> {
        ObjectMessage objectMessage = session.createObjectMessage(JsonUtils.format(object));
        objectMessage.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY, (dateTime -

```

```

        System.currentTimeMillis());
        return objectMessage;
    });
}

public void sendMessage(String queue, String time) {
    logger.info(">>>>发送 queue:" + queue + ",定时发送 : " + time + "corn : " + time);
    Destination destination = new ActiveMQQueue(queue);
    jmsTemplate.send(destination, session -> {
        ObjectMessage objectMessage = session.createObjectMessage();
        objectMessage.setStringProperty(ScheduledMessage.AMQ_SCHEDULED_CRON, time);
        return objectMessage;
    });
}
}

```

ProducerController >>

```

@RestController
public class ProducerController {
    @Autowired
    private ActiveMqUtil activeMqUtil;

    @PostMapping("/queue/test")
    public void sendQueue(@RequestBody String str) {
        activeMqUtil.sendMessage(MqConstant.INSERT_PLATFORM_MESSAGE, MessageConvert.convertPlatform(null,
        voucherBo.getMerchantId(), 6, redisMerchantVo.getName(), voucherBo.getId()));
    }
}

```

SystemMqListener >>

```

@Component
public class SystemMqListener {
    @Autowired
    private BusinessService businessService;
    private static final Logger LOG = LoggerFactory.getLogger(SystemMqListener.class);

    @JmsListener(destination = MqConstant.INIT_BUSINESS_ES) // queue 模式的消费者 containerFactory="topicListener"指定
    为 topic 模式消费者
    public void initEsBusinessMessage() {
        LOG.info("SYSTEM MQ :" + MqConstant.INIT_BUSINESS_ES);
        Result result = businessService.initEsBusiness();
        LOG.info("SYSTEM MQ :" + MqConstant.INIT_BUSINESS_ES + "result:" + JsonUtils.format(result));
    }
}

```

Caffeine

[caffeine]

Caffeine

```
package com.github.benmanes.caffeine.cache;
```

```
public final class Caffeine<K, V>
```

```
public static Caffeine<Object, Object> newBuilder()
```

Constructs a new Caffeine instance with default settings, including strong keys, strong values, and no automatic eviction of any kind.

```
public Caffeine<K, V> maximumSize(@NonNegative long maximumSize)
```

Specifies the maximum number of entries the cache may contain.

Note that the cache may evict an entry before this limit is exceeded or temporarily exceed the threshold while evicting.

As the cache size grows close to the maximum, the cache evicts entries that are less likely to be used again.

For example, the cache may evict an entry because it hasn't been used recently or very often.

```
public Caffeine<K, V> expireAfterWrite(@NonNegative long duration, @NotNull TimeUnit unit)
```

Specifies that each entry should be automatically removed from the cache once a fixed duration has elapsed after the entry's creation, or the most recent replacement of its value.

```
public <K1 extends K, V1 extends V> Caffeine<K1, V1> removalListener(@NotNull RemovalListener<? super K1, ? super V1> removalListener)
```

Specifies a listener instance that caches should notify each time an entry is removed for any reason.

Each cache created by this builder will invoke this listener as part of the routine maintenance described in the class documentation above.

Warning: after invoking this method, do not continue to use this cache builder reference; instead use the reference this method returns.

At runtime, these point to the same instance, but only the returned reference has the correct generic type information so as to ensure type safety.

```
public Caffeine<K, V> recordStats()
```

Enables the accumulation of CacheStats during the operation of the cache.

Without this Cache. stats will return zero for all statistics.

Note that recording statistics requires bookkeeping to be performed with each operation, and thus imposes a performance penalty on cache operation.

```
public <K1 extends K, V1 extends V> AsyncLoadingCache<K1, V1> buildAsync( @NotNull AsyncCacheLoader<? super K1, V1> loader)
```

Builds a cache, which either returns a CompletableFuture already loaded or currently computing the value for a given key,

or atomically computes the value asynchronously through a supplied mapping function or the supplied AsyncCacheLoader.

This method does not alter the state of this Caffeine instance, so it can be invoked again to create multiple independent caches.

Databricks

Configuration

Databricks CLI

[Install the Databricks CLI](#)

```
brew tap databricks/databricks-cli
```

```
brew install databricks-cli  
databricks --version  
Verify Installation  
databricks configure  
Run the following command to authenticate  
Enter your Databricks host URL and access token when prompted.  
https://myworkspace.cloud.databricks.com/
```

Druid

依赖: druid-spring-boot-starter = com.alibaba
常用版本: 1.1.10 1.2.8

application.yml > druid-spring-boot-starter 配置完 application.yml 就可以直接访问 <http://localhost:8099/druid>

```
#===== druid 配置  
spring:  
datasource:  
druid: # 连接池配置(通常来说, 只需要修改 initialSize、minIdle、maxActive  
    initial-size: 10 # 线程池初始大小  
    max-active: 500 # 线程池中最大连接数  
    min-idle: 10 # 线程池最小空闲数, Druid 会定期扫描连接数情况, 如果扫描的值大于该值就关闭多余的连接数, 小于就创建符合要求的连接数  
        # 这个参数的主要用处是突然有大量的请求的时候, 就会创建新的连接数, 这是个比较耗时的操作  
    max-wait: 60000 # 配置获取连接等待超时的时间  
    pool-prepared-statements: true # 打开 PSCache, 并且指定每个连接上 PSCache 的大小  
    max-pool-prepared-statement-per-connection-size: 20  
    validation-query: SELECT 'x'  
    test-on-borrow: false  
    test-on-return: false  
    test-while-idle: true  
    time-between-eviction-runs-millis: 60000 # 配置间隔多久才进行一次检测, 检测需要关闭的空闲连接, 单位是毫秒  
    min-evictable-idle-time-millis: 300000 # 配置一个连接在池中最小生存的时间, 单位是毫秒  
    filters: stat,wall,slf4j # 开启 SQL 监控, SQL 防火墙, log4j/slf4j (Session 登陆就有)  
    #aop-patterns: com.saidake.controller.* # 监控这个包里的所有内容  
    web-stat-filter: # web 应用监控, URL 监控 (统计访问 servlet 次数)  
        enabled: true  
        url-pattern: /*  
        exclusions: "*.js , *.gif ,*.jpg ,*.png ,*.css ,*.ico , /druid/*"  
        session-stat-max-count: 1000  
        profile-enable: true  
        session-stat-enable: false  
    stat-view-servlet: # 监控页功能设置  
        enabled: true  
        url-pattern: /druid/* # 根据配置中的 url-pattern 来访问内置监控页面, 如果是上面的配置, 内置监控页面的首  
页是/druid/index.html  
        reset-enable: true # 允许清空统计数据  
        login-username: admin  
        login-password: admin123  
        #allow: 127.0.0.1 # IP 白名单, 没有配置或者为空, 则允许所有访问  
        #deny: 192.168.31.253 # IP 黑名单, 若白名单也存在, 则优先使用  
filter:  
stat:  
    slow-sql-millis: 2000 # 超过 1000ms 为慢查询  
    log-slow-sql: true # 日志打印慢查询
```

```

wall:          # SQL 防火墙设置
  enabled: true
  config:
    drop-table-allow: false  # 不允许删除表

Config >
@Configuration
public class DruidConfig {
  @Bean
  @ConfigurationProperties(prefix = "spring.datasource")
  public DruidDataSource druidDataSource(){      开启 Druid
    return new DruidDataSource();
  }

  @Bean
  public ServletRegistrationBean statViewServlet() {  开启监控页
    ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new StatViewServlet(), "/druid/*");
    return servletRegistrationBean;
  }

  @Bean
  public FilterRegistrationBean webStatFilter(){      开启 Web 应用, URL 监控
    WebStatFilter webStatFilter=new WebStatFilter();
    FilterRegistrationBean<WebStatFilter> filterFilterRegistrationBean=new FilterRegistrationBean<>(webStatFilter);
    return filterFilterRegistrationBean;
  }
}

c3p0

```

Elasticsearch

Core

Make update and query operations mutually exclusive

Using Redisson for distributed locking

```

import org.redisson.Redisson;
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class LockService {

  private final RedissonClient redissonClient;

  @Autowired
  public LockService(RedissonClient redissonClient) {
    this.redissonClient = redissonClient;
  }

  public void executeWithLock(String lockKey, Runnable task) {
    RLock lock = redissonClient.getLock(lockKey);
    try {
      // Try to acquire the lock with a timeout
      if (lock.tryLock(10, 60, TimeUnit.SECONDS)) {
        try {
          task.run();
        } finally {

```

```

        lock.unlock();
    }
} else {
    // Handle the case where the lock could not be acquired
    throw new RuntimeException("Could not acquire lock");
}
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new RuntimeException("Interrupted while acquiring lock", e);
}
}
}

```

Use the Lock in Your Elasticsearch Operations

You can now use the LockService to ensure that updates and reads to Elasticsearch are synchronized:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ElasticsearchService {

    private final LockService lockService;
    private final ElasticsearchRestTemplate elasticsearchRestTemplate;

    @Autowired
    public ElasticsearchService(LockService lockService, ElasticsearchRestTemplate elasticsearchRestTemplate)
    {
        this.lockService = lockService;
        this.elasticsearchRestTemplate = elasticsearchRestTemplate;
    }

    public void updateDocument(String documentId, Object updatedData) {
        lockService.executeWithLock("updateLock:" + documentId,
            () -> {
                // Perform Elasticsearch update operation
                // For example: elasticsearchRestTemplate.save(updatedData);
            }
        );
    }

    public Object readDocument(String documentId) {
        // Read operations can be performed without the lock or with a different lock strategy if needed
        return elasticsearchRestTemplate.get(documentId, YourDocumentClass.class);
    }
}

```

Configuration

pom.xml

```

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-elasticsearch</artifactId>
    </dependency>
    <!-- Add Elasticsearch client library if needed -->
    <dependency>
        <groupId>org.elasticsearch.client</groupId>
        <artifactId>elasticsearch-rest-high-level-client</artifactId>
        <version>7.17.1</version> <!-- Adjust version according to your Elasticsearch version -->
    </dependency>
</dependencies>

```

application.yml

```
spring:
```

```
elasticsearch:  
  uris: http://localhost:9200  
  username: elastic  
  password: changeme
```

Define an Entity

Define a model class to represent the documents stored in Elasticsearch. Use annotations to specify how this class should be indexed.

```
import org.springframework.data.annotation.Id;  
import org.springframework.data.elasticsearch.annotations.Document;  
  
{@Document(indexName = "products")  
public class Product {  
  
    @Id  
    private String id;  
    private String name;  
    private String description;  
    private double price;  
  
    // Getters and setters  
}}
```

Create a Repository Interface

Create a repository interface that extends [ElasticsearchRepository](#) to handle CRUD operations for your entity.

```
import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;  
  
public interface ProductRepository extends ElasticsearchRepository<Product, String> {  
    // Custom query methods can be added here  
}
```

Use the Repository in Your Service or Controller

Inject the repository into your service or controller to perform operations on Elasticsearch.

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;
```

```
import java.util.Optional;  
  
@Service  
public class ProductService {  
  
    @Autowired  
    private ProductRepository productRepository;  
  
    public Product saveProduct(Product product) {  
        return productRepository.save(product);  
    }  
  
    public Optional<Product> findProductById(String id) {  
        return productRepository.findById(id);  
    }  
  
    public Iterable<Product> findAllProducts() {  
        return productRepository.findAll();  
    }  
  
    public void deleteProduct(String id) {  
        productRepository.deleteById(id);  
    }  
}
```

Example Controller

You can create a REST controller to expose Elasticsearch operations via HTTP endpoints.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.Optional;

@RestController
@RequestMapping("/products")
public class ProductController {

    @Autowired
    private ProductService productService;

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
        return productService.saveProduct(product);
    }

    @GetMapping("/{id}")
    public Optional<Product> getProduct(@PathVariable String id) {
        return productService.findProductById(id);
    }

    @GetMapping
    public Iterable<Product> getAllProducts() {
        return productService.findAllProducts();
    }

    @DeleteMapping("/{id}")
    public void deleteProduct(@PathVariable String id) {
        productService.deleteProduct(id);
    }
}

```

Optional

Others

ElasticSearchConfig >> 连接配置

```

@Configuration
@ConfigurationProperties(prefix = "elasticsearch")
@Component
@Qualifier
@Setter
public class ElasticSearchConfig {
    private String host;
    private int port;
    private String host2;
    private int port2;
    private String host3;
    private int port3;
    @Bean
    public RestHighLevelClient client(){
        return new RestHighLevelClient(RestClient.builder(
            new HttpHost(
                host,
                port,
                "http"
            ),
            new HttpHost(
                host2,
                port2,
                "http"
            ),
            new HttpHost(
                host3,
                port3,
                "http"
            )
        ).build());
    }
}

```

```

        "http"
    ),
})
}
ESTest >> 添加索引
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes = ConsumerApplication.class)
public class ESTest {
    @Autowired
    private RestHighLevelClient client;

    @Resource
    private PersonMapper personMapper;
    //=====
    //===== 数
    //数据库数据写入
    @Test
    public void mybatisTest() throws IOException {
        List<Person> personList=personMapper.selectByExample(new PersonExample());
        System.out.println(personList);
        BulkRequest bulkRequest=new BulkRequest();
        for (Person person:personList){
            IndexRequest indexRequest=new IndexRequest("person");
            System.out.println(person.getAlias());
            Map map=new HashMap();
            map.put("name",person.getPointBoth());
            map.put("age",person.getIdCard());
            map.put("address",person.getAlias());
            indexRequest.id(person.getId()+"").source(map);
            bulkRequest.add(indexRequest);
        }
        //执行批量操作
        BulkResponse bulkItemResponses=client.bulk(bulkRequest, RequestOptions.DEFAULT);
        RestStatus status=bulkItemResponses.status();
        System.out.println(status);
    }
    //=====
    //===== 文
    //档操作
    @Test
    public void testBulk() throws IOException {
        BulkRequest bulkRequest=new BulkRequest();
        //删除1号记录
        DeleteRequest deleteRequest=new DeleteRequest("person","1");
        bulkRequest.add(deleteRequest);
        //添加6号记录
        Map map=new HashMap();
        map.put("name","六号");
        IndexRequest indexRequest=new IndexRequest("person").id("6").source(map);
        //source(JSON.toJSONString(person), XContentType.JSON)
        bulkRequest.add(indexRequest);
        //修改3号记录
        Map map2=new HashMap();
        map2.put("name", "三号");
        UpdateRequest updateRequest=new UpdateRequest("person", "3").doc(map2);
        bulkRequest.add(updateRequest);
        //执行批量操作
        BulkResponse bulkItemResponses=client.bulk(bulkRequest, RequestOptions.DEFAULT);
        RestStatus status=bulkItemResponses.status();
        System.out.println(status);
    }
    //=====
    //===== 添

```

加索引

```
@Test
public void testGetIndex() throws IOException {
    //添加索引
    IndicesClient indicesClient=client.indices(); //获取操作索引对象
    CreateIndexRequest createIndexRequest=new CreateIndexRequest("saidake");
    String mapping="{\n" +
        "    \"properties\":{\"\n" +
        "        \"name\":{\n" +
        "            \"type\":\"keyword\"\n" +
        "        },\n" +
        "        \"age\":{\n" +
        "            \"type\":\"integer\"\n" +
        "        },\n" +
        "        \"address\":{\n" +
        "            \"type\":\"text\", \n" +
        "            \"analyzer\":\"ik_max_word\"\n" +
        "        }\n" +
        "    }\n" +
        "}";
    createIndexRequest.mapping(mapping,XContentType.JSON);
    CreateIndexResponse
createIndexResponse=indicesClient.create(createIndexRequest,RequestOptions.DEFAULT);
System.out.println(createIndexResponse.isAcknowledged());
    //获取索引
    GetIndexRequest getIndexRequest=new GetIndexRequest("saidake");
    GetIndexResponse getIndexResponse=indicesClient.get(getIndexRequest,RequestOptions.DEFAULT);
    boolean exists=indicesClient.exists(getIndexRequest,RequestOptions.DEFAULT); //判断索引是否存在
    Map<String , MappingMetadata> mappingMetadataMap=getIndexResponse.getMappings();
    for (String key:mappingMetadataMap.keySet()){
        System.out.println(key+": "+mappingMetadataMap.get(key));
    }
}
//===== 查
```

查询索引

```
@Test
public void testSearchIndex() throws IOException {
    //构建查询请求对象
    SearchRequest searchRequest = new SearchRequest( "person");
    //创建查询条件构建器
    SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
    //创建查询条件
    QueryBuilder query = QueryBuilders.matchQuery("name", "李四");//查询所有文档, 可修改查询方式

    //为查询条件构建器 指定查询条件
    sourceBuilder.query(query);
    sourceBuilder.from(0);
    sourceBuilder.size(100);

    //添加桶聚合查询
    AggregationBuilder aggregationBuilder= AggregationBuilders.terms("myage").field("age").size(100);
    sourceBuilder.aggregation(aggregationBuilder);

    //添加高亮查询
    HighlightBuilder highlightBuilder=new HighlightBuilder();
    highlightBuilder.field("address");
    highlightBuilder.preTags("<font color='red'>");
    highlightBuilder.postTags("</font>");
    sourceBuilder.highlighter(highlightBuilder);

    //添加查询条件构建器
```

```

searchRequest.source(sourceBuilder);

// 获取查询结果
SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);
// 获取命中对象
SearchHits searchHits = searchResponse.getHits();
//获取总记录数
long value = searchHits.getTotalHits().value;
System.out.println("总记录数:"+value);
SearchHit[] hits=searchHits.getHits();
for (SearchHit hit:hits){
    String sourceAsString = hit.getSourceAsString(); //获取 j 串数据
    System.out.println(sourceAsString);
    //获取高亮结果
    Map<String, HighlightField> highlightFieldMap=hit.getHighlightFields();
    HighlightField highlightField=highlightFieldMap.get("address");
    Text[] fragments=highlightField.fragments();
    System.out.println(fragments[0].toString());
}
//获取聚合结果
Aggregations aggregations=searchResponse.getAggregations();
Map<String, Aggregation> aggregationMap=aggregations.asMap();
Terms myage= (Terms)aggregationMap.get("myage");
List<? extends Terms.Bucket> buckets= myage.getBuckets();

for (Terms.Bucket bucket:buckets){
    System.out.println(bucket.getKey());
}
}
}
}

```

Hazelcast

Core

DataStructure

IMap

A distributed, partitioned, and thread-safe implementation of java.util.Map. Supports features like eviction, expiration, and querying.

IQueue

A distributed, blocking queue similar to java.util.concurrent.BlockingQueue. It supports operations like offer, poll, and peek.

ISet

A distributed implementation of java.util.Set. It ensures unique elements across the cluster.

IList

A distributed list similar to java.util.List. Allows duplicate elements and preserves insertion order.

MultiMap

A specialized map where a key can be associated with multiple values. Similar to java.util.Map<K, Collection<V>>.

IReplicatedMap

A map where entries are replicated to all nodes, offering low-latency reads and eventual consistency.

ICache

An implementation of the javax.cache.Cache interface. It's a distributed cache supporting JCache (JSR 107) standard features.

IAtomicLong

A distributed implementation of java.util.concurrent.atomic.AtomicLong, which provides a way to perform atomic updates on a long value across the cluster.

IAtomicReference

A distributed implementation of java.util.concurrent.atomic.AtomicReference. It allows atomic updates to a reference value across the cluster.

ICountDownLatch

A distributed implementation of java.util.concurrent.CountDownLatch, used to synchronize tasks across the cluster.

ISemaphore

A distributed implementation of java.util.concurrent.Semaphore, used to control access to a set of shared resources across the cluster.

ILock

A distributed implementation of java.util.concurrent.locks.Lock. It provides distributed locking capabilities.

ReliableTopic

A distributed publish/subscribe messaging system that supports reliable message delivery with a configurable backing data structure.

Ringbuffer

A distributed ring buffer data structure, typically used for buffering messages or data streams.

FlakeldGenerator

A distributed ID generator that produces unique IDs across the cluster without a single point of contention.

PNCounter

A distributed, conflict-free replicated data type (CRDT) that supports increment and decrement operations, providing strong eventual consistency.

CardinalityEstimator

A distributed data structure that provides a probabilistic estimate of the number of distinct elements.

HyperLogLog

A probabilistic data structure used for estimating the cardinality of a dataset (i.e., the number of unique elements).

ExecutorService

A distributed implementation of java.util.concurrent.ExecutorService for running distributed tasks asynchronously.

ScheduledExecutorService

A distributed implementation of java.util.concurrent.ScheduledExecutorService for scheduling and executing tasks in a distributed manner.

Distributed Lock

Using ILock Interface

Hazelcast provides an ILock interface, which is a distributed implementation of java.util.concurrent.locks.Lock.

In this example, hazelcastInstance.getLock("my-distributed-lock") provides a distributed lock with the name "my-distributed-lock".

The lock.lock() method acquires the lock, and lock.unlock() releases it.

```
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.ILock;

public class DistributedLockExample {

    public static void main(String[] args) {
        // Create or connect to a Hazelcast instance
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();

        // Get a distributed lock
        ILock lock = hazelcastInstance.getLock("my-distributed-lock");

        // Acquire the lock
        lock.lock();
    }
}
```

```

        try {
            // Critical section
            System.out.println("Lock acquired. Performing critical operation..."); 
            // Perform operations that require synchronization
        } finally {
            // Release the lock
            lock.unlock();
            System.out.println("Lock released.");
        }

        // Shutdown Hazelcast instance
        hazelcastInstance.shutdown();
    }
}

```

Using FencedLock (CP Subsystem)

Starting from Hazelcast 4.0, you can also use the FencedLock provided by the CP ([Consistency and Partition tolerance](#)) subsystem for strong consistency guarantees.

The FencedLock provides linearizability and is suitable for scenarios where strong consistency is required.

```

import com.hazelcast.cp.CPSubsystem;
import com.hazelcast.cp.lock.FencedLock;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class FencedLockExample {

    public static void main(String[] args) {
        // Create or connect to a Hazelcast instance
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();

        // Get the CP subsystem
        CPSubsystem cpSubsystem = hazelcastInstance.getCPSubsystem();

        // Get a fenced lock
        FencedLock lock = cpSubsystem.getLock("my-fenced-lock");

        // Acquire the lock
        lock.lock();
        try {
            // Critical section
            System.out.println("Fenced lock acquired. Performing critical operation..."); 
            // Perform operations that require synchronization
        } finally {
            // Release the lock
            lock.unlock();
            System.out.println("Fenced lock released.");
        }

        // Shutdown Hazelcast instance
        hazelcastInstance.shutdown();
    }
}

```

Configuration

pom.xml

```

<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast-spring</artifactId>
    <version>5.1</version>
</dependency>
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast</artifactId>
    <version>5.1</version>

```

```

</dependency>
Beans

import com.hazelcast.config.Config;
import com.hazelcast.config.JoinConfig;
import com.hazelcast.config.NetworkConfig;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.Hazelcast;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HazelcastConfiguration {

    @Bean
    public HazelcastCacheManager cacheManager() {
        return new HazelcastCacheManager(hazelcastInstance());
    }

    @Bean
    public HazelcastInstance hazelcastInstance() {
        Config config = new Config();
        NetworkConfig network = config.getNetworkConfig();
        JoinConfig join = network.getJoin();
        join.getMulticastConfig().setEnabled(false);
        join.getTcpIpConfig().addMember("127.0.0.1").setEnabled(true);
        return Hazelcast.newHazelcastInstance(config);
    }
}

```

Using Hazelcast in Your Application

```

import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.map.IMap;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class HazelcastController {

    @Autowired
    private HazelcastInstance hazelcastInstance;

    @GetMapping("/map")
    public String useMap() {
        IMap<String, String> map = hazelcastInstance.getMap("my-distributed-map");
        map.put("key", "value");
        return "Stored value: " + map.get("key");
    }
}

```

Log Analysis

Connection

com.client.impl.connection.ClientConnectionManager

success: Authenticated with server [sd-gw3u-l'cgj'.nam.g\$rhgghth.net]:8667:72d648b4-2ed5-4a9a-b6d2-53e4297c5e8f,
server version: 5.3.6, local address: /10.254.31.238:48909

failed: Unable to get live cluster connection, retry in 71268 ms, attempt: 9, cluster connect timeout: INFINITE, max
backoff: 668% ms

[hazelcast]

HazelcastInstance

```
package com.hazelcast.core;  
CPS subsystem getCPSubsystem();
```

CPSubsystem

```
package com.hazelcast.cp;  
public interface CPSubsystem
```

CP Data Structures:

ILock:

A distributed lock that guarantees only one thread or client can acquire it at any time across the cluster.

ISemaphore:

A distributed semaphore for controlling access to shared resources.

ICountDownLatch:

A distributed countdown latch for synchronizing tasks across the cluster.

IAtomicLong and IAtomicReference:

Distributed atomic operations on long values and references.

Leader Election:

The subsystem can elect a leader among nodes for specific tasks, ensuring coordination in a distributed environment.

Raft group

The Raft group is an implementation of the Raft consensus algorithm. Raft is used to ensure that all members (nodes) in the group agree on the order and outcome of operations (like locking or unlocking).

This ensures consistency across the distributed system.

FencedLock

The FencedLock itself is built upon several underlying data structures and mechanisms within Hazelcast, specifically tailored to provide strong consistency guarantees via the Raft consensus algorithm.

The FencedLock provided by Hazelcast's CPSubsystem does not directly support an expiration time or a lease mechanism that automatically releases the lock after a certain period.

How FencedLock Works:

Lock Acquisition:

When a client attempts to acquire a FencedLock, a request is sent to the leader of the Raft group managing that lock.

The leader proposes this operation to the other members of the group.

If the majority of the group members agree, the lock is granted, and a new fence token is issued.

Lock Release:

Releasing the lock also follows a similar consensus process, ensuring that the lock state is updated consistently across the cluster.

Fault Tolerance:

Since the lock is managed by a Raft group, even if some nodes in the group fail, as long as a majority of the nodes are still operational, the lock can still be acquired and released.

```
FencedLock getLock(@Nonnull String var1);
```

hibernate

Core

Hibernate is a powerful, high-performance **object-relational mapping** (ORM) framework for Java.

It is designed to simplify the development of database-centric applications by abstracting away the complexities of direct database interactions and providing a consistent API for accessing relational databases.

Hibernate is a popular object-relational mapping (ORM) framework for Java applications.

It helps developers map Java classes to database tables and Java data types to SQL data types, providing a framework to manage persistent data.

Hibernate simplifies database interactions by allowing developers **to interact with the database using Java objects** instead of SQL statements.

Configuration

```
spring:  
  jpa:  
    hibernate:  
      naming:  
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

application.properties

```
# Database Configuration  
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase  
spring.datasource.username=root  
spring.datasource.password=password  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
  
# Hibernate Configuration  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

```
spring.jpa.properties.hibernate.query.jpaql_strict_compliance=false
```

If the stricter JPQL grammar checks are causing issues during startup, you **can adjust the Hibernate configuration** to control this behavior.

You might want to disable certain validations or adjust them according to your needs.

```
spring.jpa.properties.hibernate.query.fail_on_pagination_over_collection_fetch=false
```

Ensures queries with pagination and collection fetches are valid.

[hibernate-core]

SessionFactory

```
package org.hibernate;  
public interface SessionFactory extends EntityManagerFactory, Referenceable, Serializable, Closeable 通过它获取会话和执行  
数据库操作的工厂类
```

Hibernate Session 对象不是线程安全的，所以我们不应该在多线程环境中使用它。我们可以在单线程环境中使用它，因为它比打开新会话要快。

Hibernate 中的 StatelessSession 不实现第一级缓存，也不与任何二级缓存交互。由于它是无状态的，因此它不实现事务性后写或自动脏检查或对关联实体进行级联操作。

Session `openSession()` throws HibernateException; 总是打开一个新会话。完成所有数据库操作后，我们应该关闭此会话对象。

应该在多线程环境中为每个请求打开一个新会话。对于 Web 应用程序框架，我们可以根据需要为每个请求或每个会话选择打开一个新会话。

Session `getCurrentSession()` throws HibernateException; 返回绑定到上下文的会话，此会话对象属于 hibernate 上下文，因此我们不需要关闭它。会话工厂关闭后，此会话对象将关闭

StatelessSession `openStatelessSession()`; 从 hibernate 获取无状态会话对象，无状态会话也会忽略集合。通过无状态会话执行的操作绕过 Hibernate 的事件模型和拦截器。它更像是普通的 JDBC 连接，并没有提供使用 hibernate 框架带来的任何好处。

StatelessSessionBuilder `withStatelessOptions()`;

annotations
(annotation)

GenericGenerator

```
package org.hibernate.annotations;  
  
@Target({ElementType.PACKAGE, ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
@Repeatable(GenericGenerators.class)  
public @interface GenericGenerator    自定义主键生成策略，搭配 javax.persistence.GeneratedValue  
String name();      generator name // "idGenerator"  
String strategy();   generator strategy // "com.lala.UsersSeqGenerator"  
Parameter[] parameters() default {};
```

@TypeDef 自定义数据类型【TYPE, PACKAGE】

name="json" 类型名

typeClass = JsonStringType.class 具体类型

@CreationTimestamp 在插入时，对日期类型属性 创建当前时间默认值【METHOD, FIELD】

@UpdateTimestamp 在更新时，对日期类型属性 创建当前默认值【METHOD, FIELD】

@Formula 计算字段

value = "timestamp(shipping_timeline->>'\$.deliveryAckTime')" 当前字段映射到 json 字段中

@TYPE 指定存储的数据类型（常用于存储 json 字段）

type = "json" 指定自定义数据类型

boot

PhysicalNamingStrategyStandardImpl

```
package org.hibernate.boot.model.naming;  
public class PhysicalNamingStrategyStandardImpl implements PhysicalNamingStrategy, Serializable 表名不做修改，直接  
映射
```

collection

PersistentBag

```
package org.hibernate.collection.spi;
```

```
public class PersistentBag<E> extends AbstractPersistentCollection<E> implements List<E>
public void clear()
```

engine

SharedSessionContractImplementor

```
package org.hibernate.engine.spi;
public interface SharedSessionContractImplementor extends SharedSessionContract, JdbcSessionOwner, Options,
LobCreationContext, WrapperOptions, QueryProducerImplementor, CoercionContext
```

event

DefaultMergeEventListener

```
package org.hibernate.event.internal;
public class DefaultMergeEventListener extends AbstractSaveEventListener<MergeContext> implements
MergeEventListener
onMerge
public void onMerge(MergeEvent event) throws HibernateException {
    EventSource session = event.getSession();
    EntityCopyObserver entityCopyObserver = this.createEntityCopyObserver(session);
    MergeContext mergeContext = new MergeContext(session, entityCopyObserver);

    try {
        this.onMerge(event, mergeContext);
        entityCopyObserver topLevelMergeComplete(session);
    } finally {
        entityCopyObserver.clear();
        mergeContext.clear();
    }
}

onMerge
public void onMerge(MergeEvent event, MergeContext copiedAlready) throws HibernateException {
    EventSource source = event.getSession();
    Object original = event.getOriginal();
    if (original != null) {
        LazyInitializer lazyInitializer = HibernateProxy.extractLazyInitializer(original);
        Object entity;
        if (lazyInitializer != null) {
            if (lazyInitializer.isUninitialized()) {
                LOG.trace("Ignoring uninitialized proxy");
                event.setResult(source.load(lazyInitializer.getEntityName(),
lazyInitializer.getInternalIdentifier()));
                return;
            }

            entity = lazyInitializer.getImplementation();
        } else if (managedTypeHelper.isPersistentAttributeInterceptable(original)) {
            PersistentAttributeInterceptor interceptor =
managedTypeHelper.asPersistentAttributeInterceptable(original).$_hibernate_getInterceptor();
            if (interceptor instanceof EnhancementAsProxyLazinessInterceptor) {
                EnhancementAsProxyLazinessInterceptor proxyInterceptor =
(EnhancementAsProxyLazinessInterceptor)interceptor;
                LOG.trace("Ignoring uninitialized enhanced-proxy");
                event.setResult(source.load(proxyInterceptor.getEntityName(),
proxyInterceptor.getIdentifer()));
                return;
            }

            entity = original;
        } else {
            entity = original;
        }
    }

    if (copiedAlready.containsKey(entity) && copiedAlready.isOperatedOn(entity)) {
```

```

LOG.trace("Already in merge process");
event.setResult(entity);
} else {
    if (copiedAlready.containsKey(entity)) {
        LOG.trace("Already in copyCache; setting in merge process");
        copiedAlready.setOperatedOn(entity, true);
    }

    event.setEntity(entity);
    EntityState entityState = null;
    PersistenceContext persistenceContext = source.getPersistenceContextInternal();
    EntityEntry entry = persistenceContext.getEntry(entity);
    if (entry == null) {
        EntityPersister persister = source.getEntityPersister(event.getEntityName(), entity);
        Object id = persister.getIdentifier(entity, source);
        if (id != null) {
            EntityKey key = source.generateEntityKey(id, persister);
            Object managedEntity = persistenceContext.getEntity(key);
            entry = persistenceContext.getEntry(managedEntity);
            if (entry != null) {
                entityState = EntityState.DETACHED;
            }
        }
    }

    if (entityState == null) {
        entityState = EntityState.getEntityState(entity, event.getEntityName(), entry, source,
false);
    }
}

switch (entityState) {
    JPA 在执行更新操作时，需要在更新之前查询数据库，以获取实体对象的当前持久化状态和版本号等信息，  

    同时检查是否有其他事务对该实体对象进行了修改。如果 JPA 检测到实体对象在更新前被其他事务修改过，那么会  

    抛出 OptimisticLockException 异常，  

    提示应用程序立即停止当前操作，以避免数据冲突和数据不一致等问题。
}

```

在 JPA 中，执行更新前首先查询数据库的逻辑，是为了确保数据的一致性和事务的可重复读。如果没有这个查询逻辑，

JPA 在执行更新操作时将无法检测到同时对同一个实体对象进行的并发修改，可能会导致数据的不一致。

JPA 中的更新操作是基于实体对象的状态变更识别的，需要将实体对象的状态从 Detached 状态转换为 Managed 状态，然后将其保存到数据库中。

在保存之前，JPA 需要重新查询数据库获取实体对象的持久化状态，

并将实体对象与数据库的记录进行比较，生成对应的 SQL 语句，以确保数据的一致性和完整性。

因此，JPA 在执行更新操作时需要重新查询数据库，是为了保证数据的一致性和事务的可重复读。

不过，如果数据量较大或并发请求过多，这种查询逻辑会极大地影响系统的性能，因此建议根据具体业务场景采取适当的优化措施。

```

case DETACHED:
    this.entityIsDetached(event, copiedAlready);
    break;
case TRANSIENT:
    this.entityIsTransient(event, copiedAlready);
    break;
case PERSISTENT:
    this.entityIsPersistent(event, copiedAlready);
    break;
default:
    throw new ObjectDeletedException("deleted instance passed to merge", (Object)null,
EventUtil.getLoggableName(event.getEntityName(), entity));
}

```

```

        }
    }
}

entityIsPersistent
protected void entityIsPersistent(MergeEvent event, MergeContext copyCache) {
    LOG.trace("Ignoring persistent instance");
    Object entity = event.getEntity();
    EventSource source = event.getSession();
    EntityPersister persister = source.getEntityPersister(event.getEntityName(), entity);
    copyCache.put(entity, entity, true);
    this.cascadeOnMerge(source, persister, entity, copyCache);
    this.copyValues(persister, entity, entity, source, copyCache);
    event setResult(entity);
}

copyValues
protected void copyValues(EntityPersister persister, Object entity, Object target, SessionImplementor source,
MergeContext copyCache) {
    if (entity == target) {
        TypeHelper.replace(persister, entity, source, entity, copyCache);
    } else {
        Object[] copiedValues = TypeHelper.replace(persister.getValues(entity), persister.getValues(target),
persister.getPropertyTypes(), source, target, copyCache);
        persister.setValues(target, copiedValues);
    }
}

```

id

IdentifierGenerator

```

package org.hibernate.id;
public interface IdentifierGenerator      id 生成器

```

IdentifierGeneratorFactory

```

package org.hibernate.id.factory;
public interface IdentifierGeneratorFactory      id 生成器工厂
                                                 enhanced

```

SequenceStyleGenerator

```

package org.hibernate.id.enhanced;
public class SequenceStyleGenerator implements PersistentIdentifierGenerator, BulkInsertionCapableIdentifierGenerator
                                                 internal

```

SessionFactoryImpl

```

package org.hibernate.internal;
public class SessionFactoryImpl implements SessionFactoryImplementor
public StatelessSession openStatelessSession(Connection connection) 无状态 jdbc 连接
                                                 type

```

CollectionType

```

package org.hibernate.type;
public abstract class CollectionType extends AbstractType implements AssociationType
replace

```

```

public Object replace(Object original, Object target, SharedSessionContractImplementor session, Object owner,
Map<Object, Object> copyCache) throws HibernateException {
    if (original == null) {
        return null;
    } else if (!Hibernate.isInitialized(original)) {
        if (((PersistentCollection)original).hasQueuedOperations()) {
            if (original == target) {
                AbstractPersistentCollection<?> pc = (AbstractPersistentCollection)original;
                pc.replaceQueuedOperationValues(this.getPersister(session), copyCache);
            } else {
                LOG.ignoreQueuedOperationsOnMerge(MessageHelper.collectionInfoString(this.getRole()),
                ((PersistentCollection)original).getKey());
            }
        }
    }

    return target;
} else {
    Object result = target != null && target != original && target !=
LazyPropertyInitializer.UNFETCHED_PROPERTY && (!(target instanceof PersistentCollection)
|| !((PersistentCollection)target).isWrapper(original)) ? target : this.instantiateResult(original);
    result = this.replaceElements(original, result, owner, copyCache, session);
    if (original == target) {
        boolean wasClean = target instanceof PersistentCollection
&& !((PersistentCollection)target).isDirty();
        this.replaceElements(result, target, owner, copyCache, session);
        if (wasClean) {
            ((PersistentCollection)target).clearDirty();
        }
    }

    result = target;
}

return result;
}
}

replaceElements
public Object replaceElements(Object original, Object target, Object owner, Map copyCache,
SharedSessionContractImplementor session) {
    Collection result = (Collection)target;
    result.clear();
    Type elemType = this.getElementType(session.getFactory());
    Iterator var8 = ((Collection)original).iterator();

    while(var8.hasNext()) {
        Object o = var8.next();
        result.add(elemType.replace(o, (Object)null, session, owner, copyCache));
    }

    if (original instanceof PersistentCollection && result instanceof PersistentCollection) {
        PersistentCollection<?> originalPersistentCollection = (PersistentCollection)original;
        PersistentCollection<?> resultPersistentCollection = (PersistentCollection)result;
        this.preserveSnapshot(originalPersistentCollection, resultPersistentCollection, elemType, owner,
copyCache, session);
        if (!originalPersistentCollection.isDirty()) {
            resultPersistentCollection.clearDirty();
        }
    }

    return result;
}
}

```

TypeHelper

```

package org.hibernate.type;
@Internal
public class TypeHelper

```

```

replace
public static Object[] replace(Object[] original, Object[] target, Type[] types,
SharedSessionContractImplementor session, Object owner, Map<Object, Object> copyCache) {
    Object[] copied = new Object[original.length];

    for(int i = 0; i < types.length; ++i) {
        if (original[i] != LazyPropertyInitializer.UNFETCHED_PROPERTY && original[i] !=
PropertyAccessStrategyBackRefImpl.UNKNOWN) {
            if (target[i] == LazyPropertyInitializer.UNFETCHED_PROPERTY) {
                copied[i] = types[i].replace(original[i], (Object)null, session, owner, copyCache);
            } else {
                copied[i] = types[i].replace(original[i], target[i], session, owner, copyCache);
            }
        } else {
            copied[i] = target[i];
        }
    }

    return copied;
}

```

query

SemanticException

```

package org.hibernate.query;
public class SemanticException extends QueryException 语义解析错误

```

QueryProducerImplementor

```

package org.hibernate.query.spi;
public interface QueryProducerImplementor extends QueryProducer
QueryImplementor getNamedQuery(String var1);

```

[hibernate-validator]

validator

(annotation)

```
package org.hibernate.validator.constraints;
```

@Length 字符串的长度必须在指定的范围内

```

message="error"    错误提示消息
min=0
max=11

```

@Range 字符串的长度必须在指定的范围内

```

message="error"    值必须在范围内
min=3333L
max=444L

```

hikariCP

Concept

<https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing>

Configuration

application.yml >>

```

spring:
datasource:
hikari:

```

```

connection-timeout: 10000 # default 30 seconds
validation-timeout: 3000 # The maximum time for connecting the tested activity, default 5 seconds.
idle-timeout: 60000 # Maximum allowed idle time in the Connection pool, default 10 minutes.
max-lifetime: 60000 # The lifespan of a connection (in milliseconds) is released if it is not used after a timeout,
default 30 minutes.
maximum-pool-size: 10 # The maximum number of connections allowed in the Connection pool, including idle and
active connections, default 10
minimum-idle: 5 # The minimum number of free connections allowed in the Connection pool, default 10
pool-name: HikariPool-1 # Set connection pool name, defulat automatic generation.
auto-commit: true # Transaction auto commit, default true.
connection-test-query: select 1 from dual # Test coneciton query. Using the JDBC4
<code>Connection.isValid()</code> method
# to test connection validity can be more efficient on some databases and
is recommended.
login-timeout: 5
read-only: false

```

p6spy

```

<dependency>
  <groupId>p6spy</groupId>
  <artifactId>p6spy</artifactId>
  <version>3.8.7</version>
</dependency>

```

[HikariCP]

HikariDataSource

```

package com.zaxxer.hikari;
public class HikariDataSource extends HikariConfig implements DataSource, Closeable

```

HikariConfig

```

package com.zaxxer.hikari;
public class HikariConfig implements HikariConfigMXBean
private static final Logger LOGGER = LoggerFactory.getLogger(HikariConfig.class);
private static final char[] ID_CHARACTERS =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
private static final long CONNECTION_TIMEOUT;
private static final long VALIDATION_TIMEOUT;
private static final long SOFT_TIMEOUT_FLOOR;
private static final long IDLE_TIMEOUT;
private static final long MAX_LIFETIME;
private static final long DEFAULT_KEEPALIVE_TIME = 0L;
private static final int DEFAULT_POOL_SIZE = 10;
private static boolean unitTest;
private volatile String catalog;
private volatile long connectionTimeout;

```

```
private volatile long validationTimeout;
private volatile long idleTimeout;
private volatile long leakDetectionThreshold;
private volatile long maxLifetime;
private volatile int maxPoolSize;
private volatile int minIdle;
private volatile String username;
private volatile String password;
private long initializationFailTimeout;
private String connectionInitSql;
private String connectionTestQuery;
private String dataSourceClassName;
private String dataSourceJndiName;
private String driverClassName;
private String exceptionOverrideClassName;
private String jdbcUrl;
private String poolName;
private String schema;
private String transactionIsolationName;
private boolean isAutoCommit;
private boolean isReadOnly;
private boolean isIsolateInternalQueries;
private boolean isRegisterMbeans;
private boolean isAllowPoolSuspension;
private DataSource dataSource;
private Properties dataSourceProperties;
private ThreadFactory threadFactory;
private ScheduledExecutorService scheduledExecutor;
private MetricsTrackerFactory metricsTrackerFactory;
private Object metricRegistry;
private Object healthCheckRegistry;
private Properties healthCheckProperties;
private long keepaliveTime;
private volatile boolean sealed;
```

HikariConfigMXBean

```
package com.zaxxer.hikari;
public interface HikariConfigMXBean
long getConnectionTimeout();
void setConnectionTimeout(long var1);
long getValidationTimeout();
void setValidationTimeout(long var1);
long getIdleTimeout();
void setIdleTimeout(long var1);
long getLeakDetectionThreshold();
```

```
void setLeakDetectionThreshold(long var1);
long getMaxLifetime();
void setMaxLifetime(long var1);
int getMinimumIdle();
void setMinimumIdle(int var1);
int getMaximumPoolSize();
void setMaximumPoolSize(int var1);
void setPassword(String var1);
void setUsername(String var1);
String getPoolName();
String getCatalog();
void setCatalog(String var1);
```

javaeocontrol

Kafka

Configuration

Add Dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
</dependencies>
```

Configure Kafka Properties

kafka:

bootstrap-servers: 52.82.98.209:10903,52.82.98.209:10904

producer: # producer configuration

bootstrap-servers: localhost:9092 # applies only to the Kafka producer

retries: 0

Defines the maximum number of retry attempts in case of a failure.

For example, setting retries to 5 means the producer will retry sending a message up to 5 times if it encounters a transient error.

acks: 1

acks (short for acknowledgments) is a producer configuration that determines how many acknowledgments the producer needs from the Kafka brokers before considering a write operation successful.

acks=0:

The producer does not wait for any acknowledgment from the broker. The message is sent and forgotten immediately.

Pros: Lowest latency.

Cons: Highest risk of message loss if the broker fails after receiving the message but before persisting it.

acks=1:

The producer waits for an acknowledgment from the leader broker only. The leader broker acknowledges receipt of the message after writing it to its local log.

Pros: Balanced latency and reliability.

Cons: Potential data loss if the leader broker fails after acknowledging but before replicating the message to followers.

acks=all (or acks=-1):

The producer waits for acknowledgments from all in-sync replicas (ISRs) of the partition before considering

the message sent.

This ensures that the message is replicated to all in-sync brokers.

Pros: Highest durability and reliability. Ensures message replication to all in-sync replicas.

Cons: Increased latency due to waiting for all replicas.

compression-type: snappy

Configures how the producer compresses messages before sending them to Kafka brokers.

Compression helps reduce the amount of data sent over the network and stored on disk, which can improve performance and reduce costs.

none (Default)

Description: No compression is applied.

Use Case: Suitable for scenarios where you want to minimize CPU usage and don't need to optimize for network or storage.

gzip

Description: Uses the GZIP algorithm for compression. GZIP provides high compression ratios but may have higher CPU overhead compared to other algorithms.

Compression Ratio: High

Use Case: Ideal for scenarios where data size reduction is critical and CPU resources are available to handle the overhead.

snappy

Description: Uses the Snappy algorithm, designed to provide a balance between compression ratio and speed. It is faster and has lower CPU usage compared to GZIP but may provide a slightly lower compression ratio.

Compression Ratio: Moderate

Use Case: Well-suited for cases where you need a good balance between compression efficiency and speed. Commonly used when performance is a priority.

lz4

Description: Uses the LZ4 algorithm, which is optimized for speed. It offers very fast compression and decompression speeds with a compression ratio similar to Snappy.

Compression Ratio: Moderate

Use Case: Ideal for high-throughput scenarios where fast compression and decompression are crucial.

zstd

Description: Uses the Zstandard (Zstd) algorithm, which is a newer compression algorithm designed for high compression ratios and fast speeds. It allows fine-tuning of compression levels and offers better performance compared to GZIP and Snappy.

Compression Ratio: High (configurable)

Use Case: Suitable for scenarios where you need both high compression ratios and fast speeds. It provides more flexibility by allowing you to configure different compression levels.

buffer-memory: 33554432 # Producer buffer size

The producer maintains an internal buffer to hold records that are waiting to be sent.

The size of this buffer is controlled by buffer-memory.

This buffer allows the producer to collect and batch records efficiently before sending them.

batch-size: 16384 # Batch size

When the producer sends records to the Kafka broker,

it tries to group records together into batches to optimize the number of requests.

The size of these batches is controlled by batch-size. If you have a high message throughput, a larger batch size can lead to better performance.

linger-ms: 1

Sets the amount of time to wait before sending a batch of messages to increase throughput.
key-serializer: org.apache.kafka.common.serialization.StringSerializer
value-serializer: org.apache.kafka.common.serialization.StringSerializer # Serializer

properties:

partitioner.class: com.example.MyPartitioner
enable.idempotence: true

In the context of Kafka, idempotence guarantees that a record will be delivered exactly once and in the original order it was sent, even if retries are needed.

Enables the producer to handle transient errors and retries without duplicating messages.

delivery.timeout.ms: 120000

This parameter defines the maximum time in milliseconds that the producer will attempt to send a message. If the message cannot be delivered within this time, the producer will fail the send operation and trigger a TimeoutException.

max.in.flight.requests.per.connection: 5

Controls the maximum number of unacknowledged requests (or messages) that the producer can send to a broker on a single connection before receiving acknowledgments for previous requests.

Parallelism:

It allows the producer to send multiple messages concurrently to the broker, improving throughput by not waiting for an acknowledgment (ACK) for each message before sending the next one.

Ordering Guarantees:

Setting this value too high can impact the ordering of messages. If retries are enabled (via retries parameter), and multiple requests are in flight, it's possible that a request might fail and be retried, potentially arriving after subsequent requests that were sent later but did not fail.

This can lead to out-of-order messages.

Throughput vs. Ordering:

The default value is usually 5, which strikes a balance between maximizing throughput and maintaining message order.

If strict ordering is critical, you might set this value to 1, ensuring that only one message is in flight at a time, but this could reduce throughput.

consumer: # consumer configuration
bootstrap-servers: localhost:9092

group-id: javagroup # Default consumer group ID
enable-auto-commit: true # Enable automatic offset commit(false: Manual Offset Commit)
auto-commit-interval-ms: 100 # Offset commit delay (ms)
auto-offset-reset: latest

earliest:
When there is no committed offset for a consumer group, or if the committed offset is invalid (e.g., deleted due to log retention),
the consumer will start reading from the earliest available message in the partition.

latest: (Default)
When there is no committed offset for a consumer group, the consumer will start reading from the latest message in the partition.
This means it will skip all past messages and only start consuming new messages that arrive after the consumer has started.

none:

If there are no committed offsets for a consumer group, Kafka will throw an exception to the consumer, and no messages will be consumed.

This is useful when you want to ensure that the consumer does not consume messages unless there is an existing committed offset.

fetch:

min-bytes: 50000

Minimum amount of data fetched in bytes

Ensures that the consumer fetches at least this amount of data from the broker.

max-wait-ms: 500

Specifies the maximum amount of time the broker will wait to fulfill a fetch request if the fetch.min.bytes threshold isn't met.

session:

timeout-ms: 10000

Determines the amount of time a consumer has to send a heartbeat to the Kafka broker before it is considered failed.

max-poll-records: 100 # Maximum number of records returned in a single poll

Limits the number of records returned in a single call to poll().

max-poll-interval-ms: 300000

Specifies the maximum time a consumer can take to process messages before the broker considers it as failed.

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.apache.kafka.common.serialization.StringDeserializer # Deserializer

properties:

partition.assignment.strategy: org.apache.kafka.clients.consumer.RoundRobinAssignor

org.apache.kafka.clients.consumer.RoundRobinAssignor

partition.assignment.strategy: org.apache.kafka.clients.consumer.RangeAssignor (Default Option).

org.apache.kafka.clients.consumer.StickyAssignor

Configure Producer

Programmatic Config

```
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.support.serializer.JsonSerializer;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaProducerConfig {

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, MyPartitioner.class.getName()); // Custom Partitioner
        configProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");
```

```

        configProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}

```

Custom Partitioner

A Custom Partitioner in Kafka is used on the producer side. It determines the partition to which a particular message will be sent.

```

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;

import java.util.Map;

public class MyPartitioner implements Partitioner {

    @Override
    public void configure(Map<String, ?> configs) {
        // Any configuration needed for the partitioner
    }

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster
cluster) {
        // Custom partitioning logic
        // Example: send all messages with a specific key to a particular partition
        int partition = 0;
        if (key != null) {
            partition = Math.abs(key.hashCode()) % cluster.partitionCountForTopic(topic));
        }
        return partition;
    }

    @Override
    public void close() {
        // Any cleanup needed for the partitioner
    }
}

```

Configure Consumer

References:

- [org.springframework.kafka.core.ConsumerFactory](#)
- [org.springframework.kafka.core.DefaultKafkaConsumerFactory](#)
- [org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory](#)

```

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

import java.util.HashMap;
import java.util.Map;

@EnableKafka
@Configuration

```

```

public class KafkaConsumerConfig {

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        configProps.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
        configProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        configProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(configProps);
    }

    // Multi-Threaded Consumption || Single-Threaded Consumption
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
        ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3); // Number of threads
        The factory will create a container with 3 threads for processing messages. Each thread will handle messages
        from one or more partitions.
        With multiple threads, the Kafka consumer can process messages from different partitions in parallel,
        improving throughput and resource utilization.
        factory.setAckMode(ContainerProperties.AckMode.MANUAL); // Manual acknowledgment, all messages will
        need to be manually acknowledged.

        Ack Modes Overview:
        MANUAL: Requires explicit acknowledgment. This mode gives you control over exactly when to commit the
        offsets.
        RECORD: Acknowledges each record individually.
        BATCH: Acknowledges offsets for a batch of records.
        TIME: Acknowledges offsets periodically based on time intervals.
        return factory;
    }
}

```

Create a Message Producer

Basic Producer

Sends messages to a Kafka topic with standard configurations.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class BasicMessageProducer {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    private static final String TOPIC = "my_topic";

    public void sendMessage(String message) {
        kafkaTemplate.send(TOPIC, message);
    }

    // Method to send a message with a key
    // If a key is provided, Kafka will consistently send messages with the same key to the same partition.
    public void sendMessageWithKey(String key, String message) {
        kafkaTemplate.send(TOPIC, key, message);
    }
}

```

```

// Method to send a message with headers
public void sendMessageWithHeaders(String message, Map<String, Object> headers) {
    Message<String> messageWithHeaders = MessageBuilder.withPayload(message)
        .setHeader(KafkaHeaders.TOPIC, TOPIC)
        .copyHeaders(headers)
        .build();

    kafkaTemplate.send(messageWithHeaders);
}

```

Transactional Producer

Sends messages in a transactional manner, ensuring atomicity.

The executeInTransaction method is used to perform Kafka operations within a transactional context.

This ensures that either **all operations within the transaction are successfully committed**, or **none are committed** if any operation fails.

Transaction Process and Rollback

Transaction Begin:

The producer begins a transaction by marking a series of messages to be part of a transaction.

Message Sending:

The producer sends messages to Kafka within this transaction. These messages are marked with a transaction ID.

Commit or Rollback:

At the end of the transaction, the producer can either commit or rollback the transaction:

Commit:

If the producer commits the transaction, all the messages sent within this transaction are marked as committed, and they become visible to consumers.

Rollback:

If the producer rolls back the transaction, the messages are not deleted immediately. Instead, they are marked as aborted and are not visible to consumers.

This ensures that the aborted messages do not accumulate indefinitely and waste disk space.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.stereotype.Service;

import java.util.Properties;

@Service
public class TransactionalMessageProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    @Autowired
    public TransactionalMessageProducer() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "my-transactional-id");

        ProducerFactory<String, String> producerFactory = new DefaultKafkaProducerFactory<>(props);
    }
}

```

```

        this.kafkaTemplate = new KafkaTemplate<>(producerFactory);
        kafkaTemplate.setTransactionId("my-transactional-id");
    }

    public void sendMessageInTransaction(String message) {
        kafkaTemplate.executeInTransaction(kafkaTemplate -> {
            kafkaTemplate.send("my_topic", message);
            return true;
        });
    }
}

```

Asynchronous Producer

Sends messages asynchronously and provides callbacks to handle success or failure.

The asynchronous producer sends messages to Kafka **without waiting for an immediate acknowledgment** from the broker.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.kafka.support.ListenableFuture;
import org.springframework.kafka.support.ListenableFutureCallback;
import org.springframework.stereotype.Service;

import java.util.Properties;

@Service
public class AsynchronousMessageProducer {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    private static final String TOPIC = "my_topic";

    public void sendMessageAsync(String message) {
        ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(TOPIC, message);

        future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
            @Override
            public void onSuccess(SendResult<String, String> result) {
                System.out.println("Sent message=[ " + message +
                    " ] with offset=[ " + result.getRecordMetadata().offset() + " ]");
            }
        });

        @Override
        public void onFailure(Throwable ex) {
            System.out.println("Unable to send message=[ "
                + message + " ] due to : " + ex.getMessage());
        }
    });
}

```

Synchronous Producer

Sends messages synchronously, waiting for the producer to confirm the message has been sent.

The synchronous producer waits for **an acknowledgment from the Kafka broker** before continuing.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;

import java.util.Properties;

```

```
@Service
public class SynchronousMessageProducer {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    private static final String TOPIC = "my_topic";

    public void sendMessageSync(String message) {
        try {
            SendResult<String, String> result = kafkaTemplate.send(TOPIC, message).get();
            System.out.println("Sent message=[" + message + "] with offset=[" +
result.getRecordMetadata().offset() + "]");
        } catch (Exception e) {
            System.out.println("Unable to send message=[" + message + "] due to : " + e.getMessage());
        }
    }
}
```

Create a Message Consumer

Basic Consumer

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;
```

```
@Service
public class MessageConsumer {

    @KafkaListener(topics = "my_topic", groupId = "my-group")
    public void consume(String message) {
        System.out.println("Received Message: " + message);
        // Automatic commit will be handled by Kafka
    }
    @KafkaListener(topics = {"topic1", "topic2"}, groupId = "my-group")
    public void consume(String message) {
        System.out.println("Received Message: " + message);
        // Handle the message from either topic1 or topic2
    }

    @KafkaListener(topics = "my_topic", groupId = "my-group")
    public void consume(ConsumerRecord<String, String> record,
                       @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
        System.out.println("Received Message: " + record.value() + " from partition: " + partition);
    }

    // The @Header annotation is used to retrieve the headers of the message along with the message value.
    @KafkaListener(topics = "my_topic", groupId = "my-group")
    public void consume(String message, @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String key) {
        System.out.println("Received Message: " + message);
        System.out.println("Received Key: " + key);
    }

    // The @Header annotation is used to retrieve the headers of the message along with the message value.
    @KafkaListener(topics = "my_topic", groupId = "my-group")
    public void consume(String message, @Header(KafkaHeaders.RECEIVED_HEADERS) Map<String, Object> headers) {
        System.out.println("Received Message: " + message);
        System.out.println("Received Headers: " + headers);
    }

    @KafkaListener(topics = "my_topic", groupId = "my-group")
    public void manualOffset(@Payload String message,
                           @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
                           @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
                           Consumer consumer,
                           Acknowledgment ack) {
        try {
            consumer.commitSync();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        consumer.commitAsync();
    } catch (Exception e) {
        System.out.println("commit failed");
    } finally {
        try {
            consumer.commitSync();
        } finally {
            consumer.close();
        }
    }
}

}

JSON Message Consumer

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.converter.StringJsonMessageConverter;
import org.springframework.kafka.annotation.KafkaListenerConfigurer;
import org.springframework.kafka.config.KafkaListenerEndpointRegistrar;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.stereotype.Service;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.beans.factory.annotation.Autowired;

@Service
public class JsonMessageConsumer {

    @KafkaListener(topics = "json_topic", groupId = "json-group")
    public void consume(Message message) {
        System.out.println("Received JSON Message: " + message.getContent());
    }
}

@Configuration
public class KafkaConfig implements KafkaListenerConfigurer {

    @Autowired
    private ConsumerFactory<String, String> consumerFactory;

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
        ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        factory.setMessageConverter(new StringJsonMessageConverter());
        return factory;
    }

    @Override
    public void configureKafkaListeners(KafkaListenerEndpointRegistrar registrar) {
        registrar.setContainerFactory(kafkaListenerContainerFactory());
    }
}
}

```

When messages are in JSON format and need to be converted to Java objects.

```

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class JsonKafkaConsumer {

    private final ObjectMapper objectMapper = new ObjectMapper();

```

```

@KafkaListener(topics = "my_json_topic", groupId = "json-group")
public void consume(byte[] messageBytes) {
    try {
        MyCustomObject message = objectMapper.readValue(messageBytes, MyCustomObject.class);
        System.out.println("Received Message: " + message);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static class MyCustomObject {
    private String field1;
    private int field2;

    // Getters and Setters

    @Override
    public String toString() {
        return "MyCustomObject{" +
            "field1='" + field1 + '\'' +
            ", field2=" + field2 +
            '}';
    }
}
}

```

Manual Acknowledgment

```

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.messaging.handler.annotation.Header;
import org.springframework.stereotype.Service;

@Service
public class MessageConsumer {

    // In this setup, Acknowledgment is used to manually commit the offsets after processing the message.
    @KafkaListener(topics = "my_topic", groupId = "my-group")
    public void consume(ConsumerRecord<String, String> record, Acknowledgment acknowledgment) {
        System.out.println("Received Message: " + record.value());
        System.out.println("Message Offset: " + record.offset());

        // Manually acknowledge the message / Manually commit the offset
        If you have configured automatic offset commits by setting enable.auto.commit to true,
            then the acknowledgment.acknowledge() method is not needed. Kafka handles offset commits automatically
            at regular intervals
        Even if you have enable.auto.commit set to true, you can still manually acknowledge offsets to commit them in
        advance,
        Kafka considers the message associated with the committed offset as already consumed.
        but the standard behavior of auto-commit will override manual acknowledgment in terms of committing
        offsets.
        acknowledgment.acknowledge();
    }
}

```

Error Handling

```

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.listener.ErrorHandler;
import org.springframework.kafka.listener.ListenerExecutionFailedException;
import org.springframework.kafka.listener.MessageListenerContainer;
import org.springframework.kafka.listener.adapter.ReplyHeadersConfigurer;

```

```

import org.springframework.stereotype.Service;

@Service
public class ErrorHandlingConsumer {

    @KafkaListener(topics = "error_topic", groupId = "error-group", errorHandler = "customErrorHandler")
    public void consume(String message) {
        System.out.println("Received Message: " + message);
        // Simulate an error
        if (message.contains("error")) {
            throw new RuntimeException("Error occurred while processing message");
        }
    }

    @Bean
    public ErrorHandler customErrorHandler() {
        return new ErrorHandler() {
            @Override
            public void handle(Exception thrownException, ConsumerRecord<?, ?> data) {
                System.out.println("Error occurred: " + thrownException.getMessage() + " for message: " +
data.value());
            }

            @Override
            public void handle(Exception thrownException, ConsumerRecord<?, ?> data, MessageListenerContainer
container) {
                // Custom error handling logic
                System.out.println("Error occurred: " + thrownException.getMessage() + " for message: " +
data.value());
            }

            @Override
            public boolean isAckAfterHandle() {
                return false;
            }
        };
    }
}

```

Batch Message

```

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class BatchMessageConsumer {

    @KafkaListener(topics = "batch_topic", groupId = "batch-group", containerFactory = "batchFactory")
    public void consume(List<String> messages) {
        System.out.println("Received Batch Messages: " + messages);
    }
}

@Configuration
@EnableKafka
public class KafkaBatchConfig {

    @Autowired
    private ConsumerFactory<String, String> consumerFactory;

    @Bean

```

```

public ConcurrentKafkaListenerContainerFactory<String, String> batchFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory);
    factory.setBatchListener(true);
    There is no direct configuration parameter in the application.yml file that maps exactly to
    factory.setBatchListener(true) for enabling batch processing in Spring Kafka.
    factory.getContainerProperties().setPollTimeout(3000); // Optional: Set the poll timeout
    return factory;
}
}

```

Dynamic Topic Subscription

To dynamically subscribe to topics based on runtime conditions.

```

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class DynamicTopicKafkaConsumer {

    @KafkaListener(topics = "#{dynamicTopicName}", groupId = "my-group")
    public void consume(String message) {
        System.out.println("Received Message: " + message);
    }
}

```

[spring-kafka]

ReactiveKafkaConsumerTemplate

```

package org.springframework.kafka.core.reactive;
public class ReactiveKafkaConsumerTemplate<K, V>

```

Provides support for reactive programming with Kafka in Spring applications.

It allows you to consume messages from Kafka topics in a non-blocking, asynchronous manner using Project Reactor.

```

ReactiveKafkaConsumerTemplate kafkaTemplate = new
ReactiveKafkaProducerTemplate<>(SenderOptions.create(props));
reactor.kafka.sender.SenderOptions

```

```
public Flux<ConsumerRecord<K, V>> receiveAutoAck()
```

Automatically acknowledges a batch of received messages. This method is used before the application processes the message.

```

public ReactiveKafkaProducerTemplate(SenderOptions<K, V> senderOptions)
public ReactiveKafkaProducerTemplate(SenderOptions<K, V> senderOptions, RecordMessageConverter messageConverter)

```

ConcurrentKafkaListenerContainerFactory

```

package org.springframework.kafka.config;
public class ConcurrentKafkaListenerContainerFactory<K, V> extends
AbstractKafkaListenerContainerFactory<ConcurrentMessageListenerContainer<K, V>, K, V>
org.springframework.kafka.listener.DefaultErrorHandler

```

```
public void setConcurrency(Integer concurrency)
```

This method sets the number of threads used for processing Kafka messages concurrently.

Each thread corresponds to a partition of the Kafka topic.

@Override

```
public ContainerProperties getContainerProperties()
```

Obtain the properties template for this factory - set properties as needed and they will be copied to a final properties instance for the endpoint.

```
factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
public void setContainerCustomizer(ContainerCustomizer<K, V, C> containerCustomizer)
```

Customize the KafkaListenerContainer for each listener created by the factory.

On the other hand, directly setting properties on the factory applies the configuration globally to all listeners created by the factory.

This is useful when you want to fine-tune the container's properties or add specific behaviors for individual listeners.

You can use this method to apply configurations such as setting a different 'ConsumerRebalanceListener', adjusting the 'ContainerProperties', or setting log levels for each listener container.

```
factory.setContainerCustomizer(container -> {
    if (container.getContainerProperties().getTopics()[0].equals("criticalTopic")) {
        container.getContainerProperties().setPollTimeout(1000); // Critical topic, shorter poll
    timeout
    } else {
        container.getContainerProperties().setPollTimeout(5000); // Less critical topics, longer
    poll timeout
    }
});
```

```
public void setCommonErrorHandler(CommonErrorHandler commonErrorHandler)
```

DefaultErrorHandler

```
package org.springframework.kafka.listener;
public class DefaultErrorHandler extends FailedBatchProcessor implements CommonErrorHandler
```

DefaultErrorHandler (formerly known as 'SeekToCurrentErrorHandler') is a class in Spring Kafka that provides error-handling capabilities for Kafka message listener containers.

It is used to handle exceptions that occur during the processing of Kafka messages, offering features such as retry mechanisms, message skipping, and dead-letter queue forwarding.

When to Use 'DefaultErrorHandler':

When you need to handle transient issues (e.g., network timeouts) and want to retry message consumption.

If you want to forward problematic messages to a dead-letter topic for future analysis.

When you want to control what happens after retries are exhausted (e.g., log the error, skip the message, etc.).

```
public DefaultErrorHandler(BackOff backOff)
public DefaultErrorHandler(ConsumerRecordRecoverer recoverer)
```

Usage

```
import org.springframework.kafka.listener.DefaultErrorHandler;
import org.springframework.kafka.listener.KafkaMessageListenerContainer;
import org.apache.kafka.common.TopicPartition;
import org.springframework.util.backoff.FixedBackOff;

@Bean
public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory = new
    ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    // Configuring retry with a max of 3 retries and 1 second backoff time
}
```

```

DefaultErrorHandler errorHandler = new DefaultErrorHandler(
    (consumerRecord, exception) -> {
        // Custom logic after retries are exhausted
        System.out.println("Retries exhausted for record: " + consumerRecord.value());
    },
    new FixedBackOff(1000L, 3L) // 1 second backoff, 3 retries
);
factory.setCommonErrorHandler(errorHandler);
return factory;
}

```

[reactor-kafka]

SenderOptions

```

package reactor.kafka.sender;
public interface SenderOptions<K, V>
{
    org.springframework.kafka.core.reactive.ReactiveKafkaConsumerTemplate
    static <K, V> SenderOptions<K, V> create()
    static <K, V> SenderOptions<K, V> create(@NonNull Map<String, Object> configProperties)
    static <K, V> SenderOptions<K, V> create(@NonNull Properties configProperties)
}

```

Mybatis

Core

Concept

MyBatis is an open-source persistence framework that simplifies the implementation of SQL in Java applications. It automates the mapping between **SQL databases** and **objects** in Java, providing a flexible and less invasive way to interact with databases compared to traditional JDBC and ORM frameworks like Hibernate.

Key Features of MyBatis

- SQL-Centric
Allows developers to write SQL queries directly.
- Flexible Mapping
Supports XML and annotation-based mapping between Java objects and database tables.
- Dynamic SQL
Provides capabilities to dynamically generate SQL statements.
- Caching
Supports first-level (session) and second-level (mapper) caching.
- Lazy Loading
Supports lazy loading of associations and collections.

Workflow

Spring Boot Initialization

The `SpringApplication.prepareContext` method processes auto-configuration classes that are identified by the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations.

Spring Boot's auto-configuration mechanism includes MyBatis auto-configuration (`MyBatisAutoConfiguration` class).

The auto-configuration class is loaded based on the presence of MyBatis classes in the classpath.

MyBatis-specific properties are read from `application.properties` or `application.yml`.

Properties such as `mybatis.config-location`, `mybatis.mapper-locations`, `mybatis.type-aliases-package`, etc., are processed.

SqlSessionFactory Creation and Initialization

The `SqlSessionFactoryBean` is a FactoryBean that utilizes `SqlSessionFactoryBuilder` to create `SqlSessionFactory`.

The `DataSource` is injected into the `SqlSessionFactoryBean`.

1) Configuration Loading

The `SqlSessionFactoryBean` is a Spring factory bean that creates and configures a MyBatis `SqlSessionFactory`.

During initialization (afterPropertiesSet() method), it reads MyBatis configuration files, including `mybatis-config.xml`, if specified.

2) Scan Mapper XML Files

`SqlSessionFactoryBean` scans the specified `mapperLocations` (typically in `application.properties` or `application.yml`) for XML files that define SQL mappings (`<select>`, `<insert>`, `<update>`, `<delete>` statements).

3) MappedStatement Creation

For each SQL mapping defined in the XML files, `SqlSessionFactoryBean` parses these mappings.

It creates corresponding `MappedStatement` objects that encapsulate the SQL statement, parameter mappings, result mappings, and other metadata.

4) Registration

The created `MappedStatement` objects are registered in the MyBatis `Configuration` instance associated with the `SqlSessionFactory`.

They are stored in a map within the Configuration, where the key typically represents the statement ID (namespace.statementId).

5) Initialization

`SqlSessionFactoryBean` initializes the `SqlSessionFactory`, which is a central part of MyBatis.

Configuration properties are set, such as type aliases, type handlers, and mapper locations.

Mapper Scanning

MyBatis-Spring uses `MapperScannerConfigurer` to scan specified base packages for interfaces annotated with `@Mapper`.

When Spring needs to inject `BlogMapper` into other Spring-managed beans, it actually injects the `MapperFactoryBean` instance.

The `MapperFactoryBean` creates a proxy around the `BlogMapper` interface.

This proxy intercepts method calls, manages transactions (if configured), and delegates the method execution to the actual `SqlSession`.

SqlSessionTemplate Creation and Injection

`SqlSessionTemplate` is created and acts as a thread-safe wrapper for `SqlSession`.

It's injected into mapper implementations, ensuring consistent transaction management.

Mapper interfaces are injected with `SqlSessionTemplate` instances.

Mapper methods are dynamically implemented using JDK dynamic proxies or CGLIB proxies.

Spring Boot Context Refresh

After the initial configuration and bean creation, the Spring Boot application context is refreshed.

Beans are fully initialized and ready for use.

Method Invocation

- 1) Method Invocation
The proxy created by `MapperFactoryBean` intercepts the method call to `BlogMapper.selectBlog(1)`.
- 2) MappedStatement Retrieval
MyBatis uses reflection or configuration to locate the `MappedStatement` associated with `BlogMapper.selectBlog(int id)`.
- 3) Parameter Binding
The parameter (`id` in this case) is bound to the SQL statement defined in `MappedStatement`.
- 4) Executor Invocation
The `SqlSession` uses its configured Executor (e.g., `SimpleExecutor`, `ReuseExecutor`, etc.) **to execute the SQL statement** defined by `MappedStatement`.
The Executor prepares the SQL statement, sets parameters, executes the statement against the database, and retrieves results.
- 5) Result Handling
The Executor processes the result set obtained from the database **based on the `ResultMap` configured in `MappedStatement`**.
It maps the result set rows to Java objects (`Blog` objects in this case) using the defined mappings.
- 6) Return to Application
The proxy returns the mapped Java objects or values (`Blog` object in this case) to the application code.

MyBatis Cache

MyBatis cache **stores the results of SQL queries** in memory.

1. First Level Cache

Enabled by default.

- Stores objects retrieved from the database during a session.
- the first level cache is stored within the `SqlSession` object.
- Avoids redundant queries for the same data within the session.

2. Second Level Cache

Configured in `mybatis-config.xml`.

- Requires cache implementations (ehcache, Redis, etc.).
- Entities and queries need explicit caching configuration.
- Improves performance by sharing cached data across sessions.

Example Usage

```
<!-- Example configuration in mybatis-config.xml for second level cache -->
<configuration>
    <settings>
        <setting name="cacheEnabled" value="true"/>
    </settings>
    <typeAliases>
        <!-- Type aliases configuration -->
    </typeAliases>
    <mappers>
        <!-- Mapper interfaces configuration -->
    </mappers>
    <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
</configuration>
```

Lazy loading

Lazy loading in MyBatis is a feature that allows the framework to defer loading of non-immediate associations or collections until they are explicitly accessed.

This approach helps optimize performance by reducing the initial amount of data fetched from the database, especially when dealing with complex object graphs.

1. Association Lazy Loading

Lazy loading can be applied to associations between objects. For example, if an object has a reference to another object (one-to-one or many-to-one relationships), MyBatis can defer loading of the associated object until it is accessed.

2. Collection Lazy Loading

Similarly, for collections (one-to-many or many-to-many relationships), MyBatis can postpone fetching the collection elements from the database until the collection is accessed.

3. Configuration:

Lazy loading is configured in MyBatis at two levels:

Global Level: Using configuration properties in [mybatis-config.xml](#) or programmatically in SqlSessionFactoryBean.

Statement Level: By specifying `fetchType="lazy"` in XML mappings or using `@Lazy` annotation for Java-based mappings.

Configuration

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=root
spring.datasource.password=password
mybatis.mapper-locations=classpath:mapper/*.xml
mybatis.configuration.lazy-loading-enabled=true
mybatis.configuration.cache-enabled=true
```

MyBatisConfig

```
@Configuration
@MapperScan("com.example.mapper")
public class MyBatisConfig {
    @Bean
    public DataSource dataSource() {
        return DataSourceBuilder.create()
            .url("jdbc:mysql://localhost:3306/mydatabase")
            .username("root")
            .password("password")
            .build();
    }

    @Bean
    public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
        SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
        factoryBean.setDataSource(dataSource);
        factoryBean.setMapperLocations(new
PathMatchingResourcePatternResolver().getResources("classpath:mapper/*.xml"));
        return factoryBean.getObject();
    }
}
```

BlogMapper.java

```
package com.example.mapper;

import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

@Mapper
public interface BlogMapper {
    @Select("SELECT * FROM Blog WHERE id = #{id}")
    Blog selectBlog(int id);
}
```

BlogMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
```

```

<mapper namespace="com.example.mapper.BlogMapper">
    <select id="selectBlog" parameterType="int" resultType="com.example.model.Blog">
        SELECT * FROM Blog WHERE id = #{id}
    </select>
</mapper>

```

BlogService

```

@Service
public class BlogService {
    private final BlogMapper blogMapper;

    @Autowired
    public BlogService(BlogMapper blogMapper) {
        this.blogMapper = blogMapper;
    }

    public Blog getBlogById(int id) {
        return blogMapper.selectBlog(id);
    }
}

```

[mybatis]

```

package org.apache.ibatis.session;
public interface SqlSessionFactory
SqlSession openSession();          获取 SqlSession 构建者对象，并开启手动提交事务
SqlSession openSession(boolean var1);  获取 SqlSession 构建者对象，如果参数为 true，则开启自动提交事务
SqlSession openSession(ExecutorType var1);  获取 SqlSession 构建者对象，并选择执行器类型

```

```

package org.apache.ibatis.jdbc;
public abstract class AbstractSQL<T>
public T SELECT(String... columns)      根据字段拼接查询语句
public T FROM(String... tables)         根据表名拼接语句
public T WHERE(String... conditions)   根据条件拼接语句
public T INSERT_INTO(String tableName)  根据表名拼接新增语句
public T INTO_VALUES(String... values) 指定插入的系列值
public T VALUES(String columns, String values) 根据字段和值拼接插入数据语句
public T UPDATE(String table)          根据表名拼接修改语句
public T DELETE_FROM(String table)    根据表名拼接删除语句

```

```

package org.apache.ibatis.jdbc;
public class SQL extends AbstractSQL<SQL>  sql 构建工具

```

```

package org.apache.ibatis.session;
public interface SqlSession           构建者对象接口。用于执行 SQL、管理事务、接口代理。
Connection getConnection();          获得连接对象
<T> T getMapper(Class<T> var1);    获得指定接口的代理实现类对象
<E> List<E> selectList(String var1, Object var2); 执行查询语句，返回 List 集合
<T> T selectOne(String var1, Object var2); 执行查询语句，返回一个结果对象
int insert(String var1, Object var2); 执行新增语句，返回影响行数
int update(String var1, Object var2); 执行修改语句，返回影响行数
int delete(String var1, Object var2); 执行删除语句，返回影响行数提交事务
void rollback();                     回滚事务
void commit();                      提交事务

```

```

void close();                                释放资源

package org.apache.ibatis.io;
public class Resources    资源工具
public static InputStream getResourceAsStream(String resource)    通过类加载器返回指定资源的字节输入流 // 
Resources.getResourceAsStream("MyBatisConfig.xml");

package org.apache.ibatis.session;
public class SqlSessionFactoryBuilder    获取 SqlSessionFactory 工厂对象的功能类
public SqlSessionFactory build(InputStream inputStream)    通过指定资源字节输入流获取 SqlSession 工厂对象

package org.apache.ibatis.executor;
public interface Executor                    负责动态 SQL 的生成和查询缓存的维护
package org.apache.ibatis.executor;
public class SimpleExecutor extends BaseExecutor    每执行一次 update 或 select，就开启一个 Statement 对象，用完立刻关闭
Statement 对象
package org.apache.ibatis.executor;
public class BatchExecutor extends BaseExecutor    update 批处理执行器，使用 Statement 的 addBatch 批处理执行（缓存了
多个 Statement，等待 executeBatch 逐一执行）
package org.apache.ibatis.executor;
public class ReuseExecutor extends BaseExecutor    执行 update 或 select，重复使用 Statement 对象。
                                                    以 sql 作为 key 查找 Statement 对象，存在就使用，不存在就
                                                    创建
                                                    执行完后，不关闭 Statement 对象，而是放置于 Map<String,
Statement> 内，供下一次使用。

```

[mybatis-spring]

```

package org.mybatis.spring.annotation;
@MapperScan(value="com.dongfeng.mybatisplus")    不需要对每个 Mapper 都添加@Mapper 注解，可用于
springboot 引导类

```

test 使用

```

InputStream is = Resources.getResourceAsStream("mybatis-config.xml");    // 加载核心资源配置文件
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);    // 获取 SqlSession 工厂对象
SqlSession sqlSession = sqlSessionFactory.openSession(true);    // 通过 SqlSession 工厂对象获取
SqlSession 对象
List<Student> list = sqlSession.selectList("StudentMapper.selectAll");    // 执行映射配置文件中的 sql 语句，并接收
结果
StudentDao studentDao = sqlSession.getMapper(StudentDao.class)
sqlSession.close()
is.close()

```

mybatis 使用

dao 映射配置

StudentMapper >>

```

public interface StudentMapper {           创建一个 dao
    int[] getAllStudentIds();
}

resource/mapper/StudentMapper.xml >>      创建一个 dao 映射
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.morris.dao.StudentMapper">          映射到 mapper 方法上
    <resultMap id="orderInfo" type="com.morris.po.OrderInfo">
        <id property="id" column="order_id"/>
        <result property="orderNumber" column="order_number"/>
        <association property="orderAddress" javaType="com.morris.po.OrderAddress">
            <result property="region" column="region"/>
            <result property="district" column="district"/>
            <result property="city" column="city"/>
            <result property="street" column="street"/>
        </association>
    </resultMap>

    <select id="getAllStudentIds" resultType="int">
        select id
        from student;
    </select>

</mapper>

//验证
public static void main(String [] args) {
    SqlSession session = getSqlSession();
    StudentMapper sm = session.getMapper(StudentMapper.class);
    int[] ids = sm.getAllStudentIds();
    for (int i : ids) {
        System.out.println(i);
    }
}

</mapper>

<select id="queryList" resultType="map">
    select table_name tableName, engine engine, table_comment tableComment, create_time createTime from //必须重
命名驼峰，才能映射到 entity 对象中（下划线映射到 map 的结果： TABLE_COMMENT=用户共享人表,
TABLE_NAME=t_user_share)
    information_schema.tables
    where table_schema = (select database())
    <if test="p.tableName != null and p.tableName.trim() != "">
        and table_name like concat('%', #{p.tableName}, '%')      // 对应 mapper 中的参数，默认直接取参数名
@Param("p") Map<String, Object> map (多个参数时，要使用@Param 指定参数名)
    </if>

```

```

    order by create_time desc
</select>

<select id="selectCondition" resultType="student" parameterType="student">
    SELECT * FROM student
    <where>
        <if test="id != null and id !='">           条件判断
            id = #{id}
        </if>
        <if test="name != null" >           查询条件拼接
            AND name = #{name}
        </if>
        <if test="age != null">
            AND age = #{age}
        </if>
    </where>
</select>

<![CDATA[ and DATE_FORMAT(o.create_time,'%Y-%m-%d')>=#{startDate} ]]>           <!-- 被<![CDATA[]]>这个标记所包含
的内容将表示为纯文本 -->

<select id="se1ectBylds" resultType="student" parameterType="list">
    SELECT * FROM student WHERE id IN
    <where>
        <if test="passFields!=null and passFields.size()>0" >
            <foreach collection="passFields" open="(" close=")" item="item" separator=",">           <!-- 循环遍历 testList。适用于多个参数或者的关系 -->
                #{item}
            </foreach>
        </if>
    </where>
</select>

<update id="update" parameterType="student" resultType="student">
    id          Dao 层接口的方法名
    parameterType 映射 Dao 层接口方法的参数类型。
    resultType   映射 Dao 层接口方法的返回值类型。
    update account set money = money + #{money} where name = #{name }
        #{} 是预编译处理, ${}是字符串替换 (当做占位符来用)
        #{} MyBatis 在处理#{}时, 会将 sql 中的#{}替换为?号, 调用 PreparedStatement 的 set 方法来赋值
        在使用排序时应该用$: ORDER BY ${id}    如果使用#${id}, 则会被解析成 错误形式 ORDER BY "id"
</update>
```

```

<!------- 表关联 -->

<resultMap id="testOne" type="Student">
    <id column="id" property="id" />
    <result column="origin_fk" property="origin_fk"/>
    <association property="personObj" javaType="Person" >
        association 配置被包含对象的映射关系
        property 被包含实体对象 属性名
        javaType 被包含实体对象的 数据类型
        <id column="pid" property="pid" /> 配置主键映射关系标签
        <result column="point_fk" property="point_fk"/> 配置非主键映射关系标签
        <result column="alias" property="alias" />
        <result column="gender" property="gender" />
    </association>
    <collection property="personList" ofType="Person" >
        collection 配置被包含的集合对象映射关系
        property 被包含集合对象的变量名
        ofType 集合中保存对象的实际数据类型
        <id column="pid" property="pid" />
        <result column="point_fk" property="point_fk"/>
        <result column="alias" property="alias" />
        <result column="gender" property="gender" />
    </collection>
</resultMap>

<select id="findAll" resultMap="testOne">
    SELECT * FROM student s, person p where s.origin_fk=p.point_fk      一次性查出所有关联结果，再逐一进行字段映射
</select>

```

```

<!------- 复用 sql 语句 -->

<sql id="sdk-select">SELECT * FROM student </sql>
<select id="selectAll" resultType="com.saidake.mybatis.entity.Student" resultMap="Student" >    <!-- resultMap 使用的是上方定义的 resultMap 内定义的别名 -->
    <include refid="sdk-select"/>
</select>

```

定义一个 entity

UserEntity >> 定义一个 entity (仅仅是一个 dto)
@Data
public class UserEntity {
 public int id;
 public int age;
}

定义一个 mapper

StudentMapper >> 定义一个 mapper

```

@Component
public interface StudentMapper {
    @Select("SELECT * FROM student")                                列表操作
    @Results(id="xxr", value={                                     // 当数据库字段名与实体类对应的属性名不一致时，可以使用@Results 映射
        来将其对应起来，必须配合@Select 使用。 （普通外键列不需要关联，会自动拉取外键值，外键表实体包含指向表的 Person 对象时才使
        用）
            id          // 封装可重用 id
            value       // 映射值
        @Result(column = "id", property = "id"),      // 定义了 Result 数组
        @Result(column = "age", property = "age"),
        @Result(
            property = "personObj",           // 当前表实体 接收字段
            javaType = String.class,         // 查询结果类型
            column = "point_fk",             // 当前表 作为查询参数的字段
            one = @One(select = "com.saidake.mybatisplus.mapper.PersonMapper.selectByPointFk") // 一对一查询固定属
        性， select 调用接口方法
            many=@Many(select="com.saidake.mybatisplus.mapper.PersonMapper.selectAllByPointFk") //一对多查询固定属
        性 (多对多借助中间表)
        )
    })
    List<Student> findAll();      // 可以不用注解，直接 mapper.xml 映射

    @Select("SELECT * FROM student")
    @Results("xxr")                           //引用其他映射关系
    List<Student> findAll();                列表操作

    @Insert("INSERT INTO student VALUES (#{id},#{name},#{age})")      新增操作，将内容映射到对象字段上
    (通过#{xx}获取传递给函数的参数)
    void insert(Student student);

    @Delete("DELETE FROM student WHERE id = #{id}")                    删除操作
    void delete(Integer id);

    @Update("UPDATE student SET name = #{name},age = #{age} WHERE id = #{id}")     修改操作
    void updateById(Student student);

    @Select("SELECT * FROM student WHERE id = #{id}")                  查询操作
    Student detail(Integer id);

    @SelectProvider (type = ReturnSql.class , method = "getSelectAll")      通过外部 sql 构建类获取 sql 语句
    public abstract List<Student> selectAll();

    @InsertProvider (type = ReturnSql.class , method = "getInsert")        插入类型 sql 语句
    public abstract void insert ();

    @UpdateProvider (type = ReturnSql.class , method = "getUpdate")        插入类型 sql 语句

```

```

public abstract void update();

@DeleteProvider (type = ReturnSql.class , method = "getDelete")           插入类型 sql 语句
public abstract void delete();
}

外部 sql 构建类
ReturnSql >>
public class ReturnSql {          外部 sql 构建类
    public String getSelectAll(){
        return new SQL(){{
            SELECT("*");
            FROM("student");
        }}
    }
}

```

配置项

application.yml >>

```

mybatis:
  config-location: classpath:mybatis-config.xml      # 配置文件 (resource 内)
  mapper-locations: classpath:mapper/*.xml          #数据库操作 (resource 内)
  type-aliases-package: com.sdk.mybatisplus.entity   #类型别名
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
  # configuration:                                #和 config-location 配置冲突, 不能同时使用 (一般去掉)
  # map-underscore-to-camel-case: true             #能够请求下划线数据库字段, 如 user_id

mybatis-config.xml >>      mybatis 详细配置
<?com.myutil.entity.xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  [删除] <properties resource="jdbc.properties"/>      <!--引入数据库连接的配置文件, 以下每个标签都必须按照顺序-->
  <settings>
    [删除] <setting name="logImpl" value="log4j"/>      <!--配置 LOG4J -->
    <setting name="logImpl" value="STDOUT_LOGGING" />
    <setting name="callSettersOnNulls" value="true"/>     <!-- 解决查询返回结果含 null 没有对应字段值问题 -->
  </settings>

```

```

<typeAliases>                  <!-- 为全类名起别名-->
  <typeAlias alias="Integer" type="java.lang.Integer" />
    <!-- type 指定全类名 -->
    <!-- alias 别名 -->
  <typeAlias alias="Long" type="java.lang.Long" />
  <typeAlias alias="HashMap" type="java.util.HashMap" />
  <typeAlias alias="LinkedHashMap" type="java.util.LinkedHashMap" />
  <typeAlias alias="ArrayList" type="java.util.ArrayList" />
  <typeAlias alias="LinkedList" type="java.util.LinkedList" />
  <typeAlias alias="JSONObject" type="com.alibaba.fastjson.JSONObject" />
  <typeAlias alias="JSONArray" type="com.alibaba.fastjson.JSONArray" />

```

[删除] <package name="com.saidake.mybatisplus.entity" /> <!-- 为包下所有类起别名，就是类名 -->
</typeAliases>

[删除] <environments default="mysql"> <!-- environments 配置数据库环境，环境可以有多个。default 属性指定使用的是哪个-->
<environment id="mysql"> <!-- environment 配置数据库环境 id 属性唯一标识-->
<transactionManager type="JDBC"></transactionManager> <!-- transactionManager 事务管理。 type 属性，采用 JDBC 默认的事务-->
<dataSource type="POOLED"> <!-- dataSource 数据源信息 type 属性 连接池-->
<property name="driver" value="\${driver}" /> <!-- property 获取数据库连接的配置信息 -->
<property name="url" value="\${url}" />
<property name="username" value="\${username}" />
<property name="password" value="\${password}" />
</dataSource>
</environment>
</environments>

[删除] <mappers> <!-- 映射配置 -->
<mapper resource="com/itheima/one_to_one/OneToOneMapper.xml"/> <!-- mappers 引入映射配置文件 -->
<mapper resource="com/itheima/one_to_many/OneToManyMapper.xml"/>
<mapper resource="com/itheima/many_to_many/ManyToManyMapper.xml"/> <!-- 将包内的映射器接口实现全部注册
<package name="com.saidake.mybatisplus" />
为映射器，使用注解 -->
</mappers>
</configuration>

mybatis-plus

依赖: mybatis-plus-boot-starter = com.baomidou (generator 时使用)
<!-- https://mvnrepository.com/artifact/com.baomidou/mybatis-plus-boot-starter -->
<dependency>
<groupId>com.baomidou</groupId>
<artifactId>mybatis-plus-boot-starter</artifactId>
<version>3.5.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.17</version>
</dependency>

application.yml >> boss-config

```
#----- mybatis-plus 公共配置
mybatis-plus:
  global-config:
    db-config:
      logic-delete-value: 1      #逻辑删除配置
```

```
logic-not-delete-value: 0
configuration:          # 原生配置
map-underscore-to-camel-case: true
cache-enabled: false
log-impl: org.apache.ibatis.logging.stdout.StdoutImpl # 这个配置会将执行的 sql 打印出来，在开发或测试的时候可以用
call-setters-on-nulls: false # 如果查询结果中包含空值的列，则 MyBatis 在映射的时候，不会映射这个字段
```

application-dev.yml >> boss-main

```
mybatis-plus:
  mapper-locations: classpath:/mapper/*Mapper.xml
  typeAliasesPackage: com.saidake.main.model    #实体扫描，多个 package 用逗号或者分号分隔
  global-config:
    db-config:
      id-type: auto
```

配置

MpMetaObjectHandler >> 自动插入配置

```
package com.saidake.AAConfig;
```

```
import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;
import org.apache.ibatis.reflection.MetaObject;
import org.springframework.stereotype.Component;

import java.util.Date;

/**
 * 自动补充插入或更新时的值
 *
 * @author luoYong
 * @date 2022年3月28日12:35:45
 */
@Component
public class MpMetaObjectHandler implements MetaObjectHandler {

    /**
     * 插入时的填充策略
     *
     * @param metaObject
     */
    @Override
    public void insertFill(MetaObject metaObject) {
        this.setFieldValByName("createTime", new Date(), metaObject);
        this.setFieldValByName("updateTime", new Date(), metaObject);
    }

    /**
     * 更新时的填充策略
     *
```

```
* @param metaObject
*/
@Override
public void updateFill(MetaObject metaObject) {
    this.setFieldValByName("updateTime", new Date(), metaObject);
}

}
```

MybatisPlusConfig >> 分页配置扫描

```
package com.saidake.AAConfig;

import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
import com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

//Spring boot 方式
@Configuration
//此处需要配置扫描，否则运行可能运行失败
@MapperScan("com.saidake.mybatisplus")
public class MybatisPlusConfig {

    //可在 controller 同级目录下新建一个 config 包,
    //再创建一个 MybatisPlusConfig 类,
    //将最新版@bean 和以下的内容直接复制到类中
    // 最新版
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.H2));
        return interceptor;
    }

}
```

mysql-connector-java

版本：同 mysql 版本

application.yml > 当前模块

```
spring:
  datasource:
    url:
      jdbc:mysql://${sdk.datasource.ip}:${sdk.datasource.port}/${sdk.datasource.database}?useUnicode=true&characterEncoding=utf8&useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=GMT%2B8
      username: ${sdk.datasource.username}
      password: ${sdk.datasource.password}
      driver-class-name: com.mysql.cj.jdbc.Driver
```

application-dev.yml >> config 模块

```

sdk:
datasource:
  ip: localhost
  username: root
  password: root
  port: 3306
  database: medicine

jdbc:mysql://localhost:3306/medicine?useUnicode=true&characterEncoding=utf8&useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=GMT%2B8
  useUnicode=true
  characterEncoding=utf8
  useSSL=false
  allowPublicKeyRetrieval=true
  serverTimezone=GMT%2B8
  useServerPrepStmts=true      开启预编译
  cachePrepStmts=true         预编译缓存，一次编译，多次运行（避免不同的 PreparedStatement 对象来执行相同的 SQL 语句时，出现编译两次的现象）

```

mybatis reverse code generator

依赖: mybatis-generator-maven-plugin mybatis-generator-core (覆盖 mapper 后将整个项目 install 一下)

配置项

pom.xml >>

```

<plugin>
  <groupId>org.mybatis.generator</groupId>
  <artifactId>mybatis-generator-maven-plugin</artifactId>
  <version>1.4.0</version>

  <configuration>
    <configurationFile> ${basedir}/src/main/resources/generator-config.xml</configurationFile>      <!-- 定义配置文件 -->
    <verbose>true</verbose>          <!-- 输出详细信息 -->
    <overwrite>true</overwrite>        <!-- 覆盖生成文件 -->
  </configuration>

  <executions>
    <execution>
      <id>Generate MyBatis Artifacts</id>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>

  <dependencies>
    <dependency>
      <groupId>org.mybatis.generator</groupId>

```

```

<artifactId>mybatis-generator-core</artifactId>
<version>1.4.0</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.17</version>
</dependency>
</dependencies>
</plugin>

generator-config.xml >> (生成后需要添加@MapperScan)
<?xml version="1.0" encoding="UTF-8"?>      <!-- 标签顺序不能修改 -->
<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>

    <context id="mysqlGenerator" targetRuntime="MyBatis3">
        [删除]      <properties resource="mybatis/jdbc.properties"/>          <!--引入自定义配置文件 -->
        <plugin type="org.mybatis.generator.plugins.UnmergeableXmlMappersPlugin" />  <!--覆盖生成的 XML 文件-->

        <commentGenerator>          <!--配置去掉所有生成的注释-->
            <property name="suppressAllComments" value="true"/>
        </commentGenerator>

        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
            connectionURL="jdbc:mysql://127.0.0.1:3306/kong"
            userId="root"
            password="root">          <!--设置数据库连接驱动 url 报错时将 "&" 改写为
        & -->
            <property name="useInformationSchema" value="true"/>
            <property name="nullCatalogMeansCurrent" value="true"/>  <!--只生成自己指定的用户表 -->
        </jdbcConnection>

        [删除]      <javaTypeResolver>          <!--类型解析器 -->
            <property name="forceBigDecimals" value="false" />          <!-- 默认 false, 把 jdbc decimal 和 numeric 类型解析为
        integer, 为 true 时把 jdbc decimal 和 numeric 类型解析为 java.math.BigDecimal-->
        </javaTypeResolver>

            <javaModelGenerator targetPackage="com.dongfeng.mybatis.entity" targetProject="src/main/java"/>
        <!-- 生成实体的包名和位置-->
            <sqlMapGenerator targetPackage="mapper" targetProject="src/main/resources"/>

```

```

<!-- 生成 mapper.xml 配置文件位置 -->
<javaClientGenerator type="XMLMAPPER" targetPackage="com.dongfeng.mybatisplus"
targetProject="src/main/java"/>      <!-- targetPackage 和 targetProject: 生成的 interface 文件的包和位置 -->

<table tableName="express_code" domainObjectName="ExpressCode">
    <generatedKey column="id" sqlStatement="Mysql" identity="true"/>
[删除]        <property name="useActualColumnNames" value="true" />           <!-- 不使用驼峰命名，使用真实列名 -->
</table>
<table tableName="express" domainObjectName="Express">
    <generatedKey column="id" sqlStatement="Mysql" identity="true"/>
</table>

<table tableName="%">           <!-- 指定要生成的表，用%通配符匹配全部的表-->
    <generatedKey column="id" sqlStatement="MySql" identity="true"/>
</table>
</context>

</generatorConfiguration>
ERROR: 数据库后有两点：table 标签添加 schema 属性

```

生成类解析

UserEntityMapper >>

```

public interface UserEntityMapper {
    long countByExample(UserEntityExample example);
    int deleteByExample(UserEntityExample example);
    int deleteByPrimaryKey(Integer id);
    int insert(UserEntity record);           // 所有的字段都会添加一遍，即使有的字段没有值
    int insertSelective(UserEntity record);    // 只给有值的字段赋值
    List<UserEntity> selectByExample(UserEntityExample example);
    UserEntity selectByPrimaryKey(Integer id);
    int updateByExampleSelective(@Param("record") UserEntity record, @Param("example") UserEntityExample example);
    int updateByExample(@Param("record") UserEntity record, @Param("example") UserEntityExample example);
    int updateByPrimaryKeySelective(UserEntity record); // 根据 id 更新，只给有值的字段赋值
    int updateByPrimaryKey(UserEntity record);
}

```

service 使用

MybatisTest >>

```

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class MybatisTest {

    @Resource
    private TUserMapper tUserMapper;

    @Test
    public void test() {
        TUserExample userExample = new TUserExample();

```

```

userExample.setOrderByClause("age asc");//升序
userExample.setDistinct(false);//不去重

// criteria 查询
TUserExample.Criteria criteria1 = userExample.createCriteria();
TUserExample.Criteria criteria2 = userExample.createCriteria();
criteria1.andEmailEqualTo("testlogin@qq.com"); //criteria 自动添加到 sql 语句中
criteria2.andAddressLike("%lala%");
userExample.or(criteria2);
// 直接查询
userExample.or().andEmailEqualTo("testlogin@qq.com");

List<TUser> userList = tUserMapper.selectByExample(userExample);
System.out.println(userList);
}
}

```

mybatis-plus reverse code generator

依赖: mybatis-plus-boot-starter = com.baomidou mybatis-plus-generator = com.baomidou velocity-engine-core = org.apache.velocity
 常用版本: 3.4.3.4 3.4.1 2.2
 hikari

ojdbc6

[ojdbc6]

OracleDataSource

```

package oracle.jdbc.pool;
public class OracleDataSource implements DataSource, Serializable

```

pagehelper

依赖: pagehelper = com.github.pagehelper
 自动配置依赖: pagehelper-spring-boot-starter = com.github.pagehelper

版本搭配: 5.2.0 + 1.3.0

源码解析

```

package com.github.pagehelper;
public class PageHelper 分页器

package com.github.pagehelper;
public class PageInfo<T> 封装分页相关参数
public long getTotal() 获取总条数
public int getPages() 获取总页数
public int getPageNum() 获取当前页
public int getPageSize() 获取每页显示条数
public int getPrePage() 获取上一页
public int getNextPage() 获取下一页
public boolean isFirstPage() 获取是否是第一页

```

```

public boolean isIsLastPage()    获取是否是最后一页

package com.github.pagehelper;
public class Page<E>          分页结果处理

package com.github.pagehelper.page;
public abstract class PageMethod
public static <E> Page<E> startPage(int pageNum, int pageSize) 对下方查询进行分页

```

service 使用

```

public List<Student> list() {
    PageHelper.startPage(1,3)
    List<Student> list= studentMapper.findAll();      // list 就是分页后的结果数组了
    PageInfo info = new PageInfo(list);              // 对返回数组进行封装
    return info;
}

```

Redis

Reference:

<https://www.baeldung.com/spring-data-redis-tutorial>

Dependencies:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>3.2.0</version>
</dependency>
or
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>3.2.0</version>
</dependency>

<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>5.1.2</version>
    <type>jar</type>
</dependency>

```

Core

Redis Transaction

```

import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.SessionCallback;
import org.springframework.stereotype.Service;

@Service

```

```

public class RedisService {

    private final RedisTemplate<String, Object> redisTemplate;

    public RedisService(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    public void performTransactionalOperations() {
        redisTemplate.execute((SessionCallback<Void>) redisConnection -> {
            redisConnection.multi();
            redisConnection.set("key1".getBytes(), "value1".getBytes());
            redisConnection.set("key2".getBytes(), "value2".getBytes());
            redisConnection.exec();
            return null;
        });
    }
}

```

Distributed Lock

Add Distributed Lock

```

package com.jgrccb.shop.common.aspect;

import com.jgrccb.shop.common.exception.RRException;
import com.jgrccb.shop.common.annotation.AddLock;
import com.jgrccb.shop.common.lock.RedisLockUtil;
import com.jgrccb.shop.common.lock.SpelUtil;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.AnnotationUtils;
import org.springframework.stereotype.Component;

import java.lang.reflect.Method;

@Aspect
@Component
@Configuration
@Slf4j
public class AddLockAspect {
    private static final Logger LOG = LoggerFactory.getLogger(AddLockAspect.class);

    @Autowired
    private RedisLockUtil redisLockUtil;

    @Pointcut("@annotation(com.jgrccb.shop.common.annotation.AddLock)")
    public void addLockAnnotationPointcut() {

    }

    @Around(value = "addLockAnnotationPointcut()")
    public Object addKeyMethod(ProceedingJoinPoint joinPoint) throws Throwable {
        String message = "";
        Object proceed = null;
        Signature signature = joinPoint.getSignature();
        MethodSignature methodSignature = (MethodSignature) signature;
        Method targetMethod = methodSignature.getMethod();
    }
}

```

```

Object target = joinPoint.getTarget();
Object[] arguments = joinPoint.getArgs();
AddLock annotation = AnnotationUtils.findAnnotation(targetMethod, AddLock.class);
String spel = null;
if (annotation != null) {
    spel = annotation.key();
}
String redisKey = "lock:" + SpelUtil.parse(target, spel, targetMethod, arguments);
boolean isLock = false;
long excuteTime = 0;
try {
    // Add a key if the key doesn't exists and add expiration timeout.
    if (annotation != null) {isLock = redisLockUtil.addRedisLock(redisKey, annotation.timeout());}
    if (isLock) {
        excuteTime = System.currentTimeMillis();
        proceed = joinPoint.proceed();
    }
} catch (Exception exception) {
    //如果我自己加锁成功，出了异常则将锁释放掉
    if (isLock) {
        redisLockUtil.deleteRedisLock(redisKey);
    }
    message = exception.getMessage();
    LOG.error("redisCache error", exception);
    LOG.info("redisCache:" + message);
    throw new RRException(message);
} finally {
    //加锁失败抛出异常
    if (!isLock) {
        LOG.error("重复提交," + redisKey, 1032);
    } else {
        //如果 service 执行时间大于 timeout 的话，可能 redis 已经自动释放锁了，所以这边不在主动释放锁
        if ((System.currentTimeMillis() - excuteTime) < annotation.timeout() * 1000) {
            redisLockUtil.deleteRedisLock(redisKey);
        }
        if (proceed == null) {
            LOG.error(message);
        }
    }
}
return proceed;
}

}

@Configuration
public class RedisLockUtil {

    @Autowired
    private RedisUtils redisUtils;

    public boolean addRedisLock(String key, Long timeout) {
        return redisUtils.setLock(key, timeout);
    }

    public void deleteRedisLock(String key) {
        redisUtils.del(key);
    }
}

public boolean setLock(String key, Long expire) {
    Boolean b = (Boolean) redisTemplate.execute(new RedisCallback<Boolean>() {
        @Override
        public Boolean doInRedis(RedisConnection connection) throws DataAccessException {
            RedisSerializer keySerializer = redisTemplate.getKeySerializer();

```

```

        Object obj = connection.execute("set", keySerializer.serialize(key),
            SafeEncoder.encode("NX"),
            SafeEncoder.encode("EX"),
            Protocol.toByteArray(expire));
        return obj != null;
    }
});
return b;
}

```

Release Distributed Lock

```

public static final String UNLOCK LUA = "if redis.call('get',KEYS[1]) == ARGV[1] then return
redis.call('del',KEYS[1]) else return 0 end";

public boolean releaseLock(String key, String requestId) {
    // 释放锁的时候，有可能因为持锁之后方法执行时间大于锁的有效期，此时有可能已经被另外一个线程持有锁，所以不能直接删除
    try {
        List<String> keys = new ArrayList<>();
        keys.add(key);
        List<String> args = new ArrayList<>();
        args.add(requestId);

        // 使用 lua 脚本删除 redis 中匹配 value 的 key，可以避免由于方法执行时间过长而 redis 锁自动过期失效的时候误删其他线程的锁
        // spring 自带的执行脚本方法中，集群模式直接抛出不支持执行脚本的异常，所以只能拿到原 redis 的 connection 来执行脚本
        RedisCallback<Long> callback = (connection) -> {
            Object nativeConnection = connection.getNativeConnection();
            // 集群模式和单机模式虽然执行脚本的方法一样，但是没有共同的接口，所以只能分开执行
            // 集群模式
            if (nativeConnection instanceof JedisCluster) {
                return (Long) ((JedisCluster) nativeConnection).eval(UNLOCK LUA, keys, args);
            }

            // 单机模式
            else if (nativeConnection instanceof Jedis) {
                return (Long) ((Jedis) nativeConnection).eval(UNLOCK LUA, keys, args);
            }
            return 0L;
        };

        Object result = redisTemplate.execute(callback);

        return result != null && Long.parseLong(result.toString()) > 0;
    } catch (Exception e) {
        log.error("release lock occurred an exception", e);
    }
    return false;
}

```

Deduct Stock

```

public static final String STOCK LUA = "local value = redis.call('hget', KEYS[1], ARGV[1]) if not value
then return -1 end value = tonumber(value) local decrby = tonumber(ARGV[2]) if value >= decrby then return
redis.call('hincrby', KEYS[1], ARGV[1], -decrby) end return -1";

public Long deductStock(String name, String key, int num) {
    // 脚本里的 KEYS 参数
    List<String> keys = new ArrayList<>();
    keys.add(name);
    // 脚本里的 ARGV 参数

```

```

List<String> args = new ArrayList<>();
args.add(key);
args.add(Integer.toString(num));

Object result = redisTemplate.execute(new RedisCallback<Long>() {
    @Override
    public Long doInRedis(RedisConnection connection) throws DataAccessException {
        Object nativeConnection = connection.getNativeConnection();
        // 集群模式和单机模式虽然执行脚本的方法一样，但是没有共同的接口，所以只能分开执行
        // 集群模式
        if (nativeConnection instanceof JedisCluster) {
            return (Long) ((JedisCluster) nativeConnection).eval(STOCK_LUA, keys, args);
        }

        // 单机模式
        else if (nativeConnection instanceof Jedis) {
            return (Long) ((Jedis) nativeConnection).eval(STOCK_LUA, keys, args);
        }
        return -1L;
    }
});
return Long.parseLong(result.toString());
}

```

Http Session

Configure Spring Session to use Redis by creating a configuration class:

The `@EnableRedisHttpSession` annotation ensures that HTTP sessions are stored in Redis.

You don't need to manually handle session storage in Redis; Spring Session handles it automatically.

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;

@Configuration
@EnableRedisHttpSession
public class SessionConfig {

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        // Configure and return a RedisConnectionFactory bean if needed
        return new LettuceConnectionFactory();
    }
}

```

Configuration

Add Dependencies

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<!-- Optional for HttpSession Storage --&gt;
&lt;dependency&gt;
    &lt;groupId&gt;org.springframework.session&lt;/groupId&gt;
    &lt;artifactId&gt;spring-session-data-redis&lt;/artifactId&gt;
&lt;/dependency&gt;
</pre>

```

application.yml

```

spring
  redis:
    host: 127.0.0.1

```

```

port: 6379
database: 8
password:
timeout: 10000      # 连接超时时间毫秒值
jedis:
  pool:
    max-active: 150    # 连接池最大连接数 (使用负值表示没有限制)
    max-wait: -1       # 连接池最大阻塞等待时间 (使用负值表示没有限制)
    max-idle: 10        # 连接池中的最大空闲连接
    min-idle: 5         # 连接池中的最小空闲连接

session
  store-type: redis    # Cache Session

```

Beans

First, using the Jedis client, we're defining a connectionFactory.

Then we defined a RedisTemplate using the jedisConnectionFactory. This can be used for querying data with a custom repository.

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.listener.RedisMessageListenerContainer;

@Configuration
public class RedisConfig {
    @Bean
    JedisConnectionFactory jedisConnectionFactory() {
        JedisConnectionFactory jedisConFactory = new JedisConnectionFactory();
        jedisConFactory.setHostName("localhost");
        jedisConFactory.setPort(6379);
        return jedisConFactory;
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(jedisConnectionFactory());
        return template;
    }

    // The second way
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
        return template;
    }

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {
        RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10)) // Set the TTL for cache entries
            .disableCachingNullValues();

        return RedisCacheManagerBuilderCustomizer.customize(redisConnectionFactory)
            .withCacheConfiguration("default", config)
            .build();
    }
}

```

```

@Bean
public KeyGenerator keyGenerator() {
    return (target, method, params) -> {
        StringBuilder sb = new StringBuilder();
        sb.append(target.getClass().getSimpleName());
        sb.append(".").append(method.getName()).append("(");
        for (Object param : params) {
            sb.append(param.toString());
        }
        sb.append(")");
        return sb.toString();
    };
}

@Bean
public RedisMessageListenerContainer redisContainer(RedisConnectionFactory connectionFactory) {
    RedisMessageListenerContainer container = new RedisMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    return container;
}
}

```

Redis Entity

```

@RedisHash("Student")
public class Student implements Serializable {

    public enum Gender {
        MALE, FEMALE
    }

    private String id;
    private String name;
    private Gender gender;
    private int grade;
    // ...
}

```

Redis JPA Repository

```

@Repository
public interface StudentRepository extends CrudRepository<Student, String> {}

```

Optional

RedisKeyspaceEventListener

```

import org.springframework.data.redis.connection.Message;
import org.springframework.data.redis.connection.MessageListener;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.listener.KeyspaceEventMessageListener;
import org.springframework.data.redis.listener.RedisMessageListenerContainer;
import org.springframework.stereotype.Component;

@Component
public class RedisKeyspaceEventListener extends KeyspaceEventMessageListener {

    public RedisKeyspaceEventListener(RedisMessageListenerContainer listenerContainer) {
        super(listenerContainer);
    }

    @Override
    protected void doHandleMessage(Message message) {
        String event = new String(message.getBody());
        String channel = new String(message.getChannel());
        System.out.println("Received event: " + event + " on channel: " + channel);

        // Add your custom handling logic here
    }
}

```

Enable Session Cache

If you are using Spring Session with Redis, calling the session.invalidate() method **will automatically clear the user session data from Redis**.

Automatic Removal:

Spring Session with Redis **automatically removes the session data from Redis** when session.invalidate() is called.

This ensures that all data associated with the invalidated session is purged from the Redis store, making it inaccessible for future requests.

Automatic Invalidation:

When the session.invalidate() method is called, the server **will invalidate the session** and typically also handle the expiration of the JSESSIONID cookie automatically. Here's how it works:

Client-Side:

The JSESSIONID cookie is a session cookie that **helps the server identify the session** for subsequent requests.

Server-Side:

When the server invalidates the session, it sends a new Set-Cookie header to the client to expire the JSESSIONID cookie.

Spring Session (and most servlet containers) will handle **sending the appropriate Set-Cookie header with the JSESSIONID cookie set to expire**.

This ensures that the client's browser will no longer send the old JSESSIONID cookie in subsequent requests.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;

@SpringBootApplication
@EnableRedisHttpSession // This annotation enables Redis-based session management
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

Use Redis to store and retrieve session data
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpSession;

@RestController
@RequestMapping("/session")
public class SessionController {

    @GetMapping("/set")
    public String setSession(HttpSession session) {
        session.setAttribute("username", "john_doe");
        return "Session data set";
    }

    @GetMapping("/get")
    public String getSession(HttpSession session) {
        return "Username: " + session.getAttribute("username");
    }
}
```

[spring-data-redis]

cache

RedisCacheConfiguration

```
package org.springframework.redis.cache;
```

```

public class RedisCacheConfiguration          redis 配置
public static RedisCacheConfiguration defaultCacheConfig()    缓存配置
public RedisCacheConfiguration serializeKeysWith(SerializationPair<String> keySerializationPair)   定义 key 的序列化协议,
同时的 hash key 也被定义.
public RedisCacheConfiguration serializeValuesWith(SerializationPair<?> valueSerializationPair)    定义 value 的序列化协
议, 同时的 hash value 也被定义.
public RedisCacheConfiguration disableCachingNullValues()    禁止缓存 Null 对象. 这个需求而定

```

RedisCacheManager

```

package org.springframework.data.redis.cache;
public class RedisCacheManager extends AbstractTransactionSupportingCacheManager
public static RedisCacheManagerBuilder builder(RedisConnectionFactory connectionFactory)

public static class RedisCacheManagerBuilder
    public RedisCacheManagerBuilder cacheDefaults(RedisCacheConfiguration defaultCacheConfiguration)
    public RedisCacheManager build()

```

connection

JedisClientConfiguration

```

package org.springframework.data.redis.connection.jedis;
public interface JedisClientConfiguration
static JedisClientConfigurationBuilder builder()
interface JedisClientConfigurationBuilder
    JedisPoolingClientConfigurationBuilder usePooling()
    JedisClientConfiguration build()
interface JedisPoolingClientConfigurationBuilder
    JedisPoolingClientConfigurationBuilder poolConfig(GenericObjectPoolConfig poolConfig)
    JedisClientConfigurationBuilder and()
    JedisClientConfigurationBuilder readTimeout(Duration readTimeout)    读取超时时间

```

RedisConnection

```

package org.springframework.data.redis.connection;
public interface RedisConnection extends RedisCommandsProvider, DefaultedRedisConnection, AutoCloseable

```

Object execute(String command, byte[]... args);

The execute method allows you to send custom commands to Redis that may not be directly supported by higher-level abstractions in the Spring Data Redis library.

This method is useful for advanced use cases where you need to interact with Redis at a low level.

Parameters

- command: A String representing the Redis command to be executed (e.g., "SET", "GET", "HSET").
- args: A variable-length argument (byte[]...) representing the arguments for the command.

RedisPassword

```

package org.springframework.data.redis.connection;
public class RedisPassword          redis 密码处理
public static RedisPassword of(@Nullable String passwordAsString)

```

core

RedisTemplate

```
package org.springframework.data.redis.core;
public class RedisTemplate<K, V>
    extends RedisAccessor implements RedisOperations<K, V>, BeanClassLoaderAware
        org.springframework.data.redis.connection.RedisConnection execute method

public RedisSerializer<?> getKeySerializer()
    Returns the key serializer used by this template.

public ValueOperations<K, V> opsForValue()
public <HK, HV> HashOperations<K, HK, HV> opsForHash()
public ListOperations<K, V> opsForList()
public Boolean delete(K key)

public <T> T execute(RedisCallback<T> action)
    Execute custom operations using the native Redis connection. It provides a way to directly interact with Redis without
    being restricted to the predefined methods in RedisTemplate.

Example:
public boolean setLock(String key, long expire) {
    Boolean b = (Boolean) redisTemplate.execute(new RedisCallback<Boolean>() {
        @Override
        public Boolean doInRedis(RedisConnection connection) throws DataAccessException {
            RedisSerializer keySerializer = redisTemplate.getKeySerializer();
            Object obj = connection.execute("set", keySerializer.serialize(key),
                SafeEncoder.encode("NX"),
                SafeEncoder.encode("EX"),
                Protocol.toByteArray(expire));
            return obj != null;
        }
    });
    return b;
}
```

RedisCallback

```
package org.springframework.data.redis.core;
public interface RedisCallback<T>

T doInRedis(RedisConnection connection) throws DataAccessException;
```

ValueOperations

```
package org.springframework.data.redis.core;
public interface ValueOperations<K, V>          操作字符串
void set(K key, V value);                      插入
void set(K key, V value, long timeout, TimeUnit unit);  插入
V get(Object key);
V getAndSet(K key, V value);
Long increment(K key, long delta);           自增自减 // redisTemplate.opsForValue().increment(key, -delta); 负数为减
Double increment(K key, double delta);       自增自减
```

HashOperations

```
package org.springframework.data.redis.core;
public interface HashOperations<H, HK, HV>          操作哈希值
void put(H key, HK hashKey, HV value);           插入
void putAll(H key, Map<? extends HK, ? extends HV> m);    插入多个
HV get(H key, Object hashKey);        查询
Map<HK, HV> entries(H key);      一次查出所有
Long delete(H key, Object... hashKeys);   删除
```

ListOperations

```
package org.springframework.data.redis.core;
public interface ListOperations<K, V>          操作 list 值
Long rightPush(K key, V value);       插入
Long rightPushAll(K key, V... values);  插入多个
Long size(K key);                  查询大小
List<V> range(K key, long start, long end);  查询多个, 0 开始 到 size
Long remove(K key, long count, Object value);  删除单个 , 删除等于 value 的 count 个值
                                                listener
```

KeyspaceEventMessageListener

```
package org.springframework.data.redis.listener;
public abstract class KeyspaceEventMessageListener implements MessageListener, InitializingBean, DisposableBean
The KeyspaceEventMessageListener class in Redis is typically used in conjunction with Spring Data Redis to listen for
keyspace events.
These events notify your application when certain actions occur on Redis keys, such as key expiration, deletion, or
modification.
protected abstract void doHandleMessage(Message message);
```

RedisSerializer

```
package org.springframework.data.redis.serializer;
public interface RedisSerializer<T>
RedisSerializer is an interface that defines methods for serializing and deserializing objects to and from Redis.
Serialization is essential for converting objects into a format that can be stored in Redis,
and deserialization is needed to convert the stored data back into objects.
```

```
byte[] serialize(@Nullable T t) throws SerializationException;
T deserialize(@Nullable byte[] bytes) throws SerializationException;
```

[jedis]

JedisPoolConfig

```
package redis.clients.jedis;
public class JedisPoolConfig extends GenericObjectPoolConfig    连接池配置
```

[commons-pool]

GenericObjectPoolConfig

```
package org.apache.commons.pool2.impl;
public class GenericObjectPoolConfig extends BaseObjectPoolConfig
public void setMaxTotal(int maxTotal)
```

BaseObjectPoolConfig

```
package org.apache.commons.pool2.impl;
public abstract class BaseObjectPoolConfig extends BaseObject implements Cloneable
public void setTestOnBorrow(boolean testOnBorrow)    当应用向连接池申请连接时，连接池会判断这条连接是否是可用的
public void setTestOnReturn(boolean testOnReturn)    默认 false
public void setTestWhileIdle(boolean testWhileIdle)   应用向连接池申请连接，并且 testOnBorrow 为 false 时，连接池将会判断
连接是否处于空闲状态，如果是，则验证这条连接是否可用。
```

使用

```
Jedis jedis=new Jedis("127.0.0.1",6379);      直接连接 redis
jedis.set("name","itheima");
jedis.close();
```

```
public static Jedis getJedis(){
    JedisPoolConfig jpc = new JedisPoolConfig();    可以封装在静态代码块内
    jpc.setMaxTotal(50);
    jpc.setMaxIdle(10);
    JedisPool jp =new JedisPool(jpc, "127.0.0.1",6379 );
    return jp.getResource();
}
```

SdkRedisConfig >>

```
package com.zhongping.AAAconfig;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.connection.RedisPassword;
import org.springframework.data.redis.connection.RedisStandaloneConfiguration;
import org.springframework.data.redis.connection.jedis.JedisClientConfiguration;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.StringRedisTemplate;
import redis.clients.jedis.JedisPoolConfig;

import java.time.Duration;

@Configuration
@EnableCaching
public class BossRedisConfig {

    @Value("${spring.redis.host}")
    private String host;

    @Value("${spring.redis.port}")
    private int port;

    @Value("${spring.redis.timeout}")
    private int timeout;

    @Value("${spring.redis.password}")
    private String password;

    @Value("${spring.redis.database}")
    private int database;
```

```

@Value("${spring.redis.jedis.pool.max-active}")
private int maxActive;

@SuppressWarnings("unchecked")
@Bean(name = "bossRedisTemplate")
@Primary
public StringRedisTemplate otosaasRedisTemplate() {
    //配置 redisTemplate
    StringRedisTemplate redisTemplate = new StringRedisTemplate();
    redisTemplate.setConnectionFactory(otosaasRedisConnectionFactory(host, database));
    // setSerializer(redisTemplate);
    return redisTemplate;
}

public RedisConnectionFactory otosaasRedisConnectionFactory(String host, int database) {
    return createRedisConnectionFactory(database, host, port, password, timeout);
}

public JedisConnectionFactory createRedisConnectionFactory(int database, String host, int port, String
password, int timeout) {
    JedisPoolConfig poolConfig = new JedisPoolConfig();
    poolConfig.setMaxTotal(maxActive);
    poolConfig.setTestOnBorrow(true);
    poolConfig.setTestOnReturn(false);
    poolConfig.setTestWhileIdle(true);
    JedisClientConfiguration clientConfig = JedisClientConfiguration.builder()
        .usePooling().poolConfig(poolConfig).and().readTimeout(Duration.ofMillis(timeout)).build();

    // 单点 redis
    RedisStandaloneConfiguration redisConfig = new RedisStandaloneConfiguration();
    // 哨兵 redis
    // RedisSentinelConfiguration redisConfig = new RedisSentinelConfiguration();
    // 集群 redis
    // RedisClusterConfiguration redisConfig = new RedisClusterConfiguration();
    redisConfig.setHostName(host);
    redisConfig.setPassword(RedisPassword.of(password));
    redisConfig.setPort(port);
    redisConfig.setDatabase(database);

    return new JedisConnectionFactory(redisConfig,clientConfig);
}

}

@.Autowired
private RedisTemplate redistemplate

redistemplate.opsForValue().set(RedisKey.TOKEN_KEY + email, token, 30, TimeUnit.DAYS);

```

Redisson

Core

Redisson is a Java-based Redis client that provides an advanced and feature-rich framework for interacting with Redis. It simplifies the usage of Redis in Java applications by offering a variety of distributed data structures, services, and utilities, all designed to be highly reliable and performant.

Components

Distributed Data Structures: Redisson provides implementations for common Java data structures that are backed by Redis.

These include:

- RMap (distributed map)
- RSet (distributed set)
- RQueue (distributed queue)
- RList (distributed list)
- RScoredSortedSet (distributed sorted set)
- RBucket (distributed object holder)
- RHyperLogLog (distributed hyperloglog)
- RBitSet (distributed bitset)

Distributed Services: Redisson offers various distributed services, including:

Locks:

`RLock`, `RReadWriteLock`, `RSemaphore`, and `RCountDownLatch` for distributed locking mechanisms.

AtomicLong:

`RAtomicLong` for atomic long operations.

AtomicDouble:

`RAtomicDouble` for atomic double operations.

ExecutorService:

`RScheduledExecutorService` for distributed task scheduling and execution.

Reactive and Asynchronous API:

Redisson provides a **reactive API** based on the Reactor framework and an asynchronous API based on Java's `CompletableFuture`.

Tomcat, Spring, and Hibernate Integration:

Redisson integrates with popular Java frameworks and containers to provide session management and caching.

Support for Clustered, Sentinel, and Single Node Redis:

Redisson supports various Redis configurations, including single instance, clustered, and Sentinel setups.

Reliability and Failover:

Redisson includes features for automatic failover and reconnection, ensuring high availability and resilience.

Using Redisson for Distributed Locks

You can also use Redisson for distributed locking. Here's an example of how to use a distributed lock with Redisson in a Spring service:

```
import org.redisson.api.RLock;
import org.redisson.api.RedissonClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.concurrent.TimeUnit;

@Service
public class LockService {

    @Autowired
    private RedissonClient redissonClient;

    public void doSomethingWithLock() {
        RLock lock = redissonClient.getLock("myLock");
        try {
            // Acquire the lock with a timeout
            if (lock.tryLock(10, 30, TimeUnit.SECONDS)) {
                try {
                    // Critical section: perform your business logic here
                    System.out.println("Lock acquired, doing something...");
                } finally {
                    lock.unlock();
                }
            }
        } catch (Exception e) {
            // Handle exception
        }
    }
}
```

```
        }
    } else {
        System.out.println("Could not acquire lock");
    }
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}
```

Using RDelayedQueue

```
import org.redisson.api.RDelayedQueue;
import org.redisson.api.RQueue;
import org.redisson.api.RedissonClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import java.util.concurrent.TimeUnit;

@Service
public class DelayedQueueService {

    @Autowired
    private RedissonClient redissonClient;

    private RQueue<String> queue;
    private RDelayedQueue<String> delayedQueue;

    @PostConstruct
    public void init() {
        // Initialize the RQueue and RDelayedQueue
        queue = redissonClient.getQueue("myQueue");
        delayedQueue = redissonClient.getDelayedQueue(queue);
    }

    public void addMessage(String message, long delay, TimeUnit timeUnit) {
        // Add a message to the delayed queue
        delayedQueue.offer(message, delay, timeUnit);
    }

    public void processMessages() {
        // Process messages from the queue
        while (true) {
            String message = queue.poll();
            if (message != null) {
                System.out.println("Processing message: " + message);
            } else {
                break;
            }
        }
    }

    public void shutdown() {
        // Remove all delayed messages and delete the delayed queue
        delayedQueue.destroy();
    }
}
```

Testing the DelayedQueueService

You can now test the `DelayedQueueService` by adding messages and processing them after the delay:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.concurrent.TimeUnit;
```

```

@SpringBootApplication
public class RedissonApplication implements CommandLineRunner {

    @Autowired
    private DelayedQueueService delayedQueueService;

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

    @Override
    public void run(String... args) throws Exception {
        // Add messages to the delayed queue
        delayedQueueService.addMessage("Message 1", 5, TimeUnit.SECONDS);
        delayedQueueService.addMessage("Message 2", 10, TimeUnit.SECONDS);

        // Wait for messages to be processed
        Thread.sleep(15000);

        // Process messages
        delayedQueueService.processMessages();

        // Shutdown the service
        delayedQueueService.shutdown();
    }
}

```

Configuration

pom.xml

```

<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson-spring-boot-starter</artifactId>
    <version>3.17.4</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>

```

application.properties

```

spring.redis.host=localhost
spring.redis.port=6379

# Optional: Configure Redisson specific settings
redisson.file=classpath:redisson.yaml

```

Alternatively, you can configure Redisson using a configuration file (redisson.yaml or redisson.json).

redisson.yaml

```

singleServerConfig:
    address: "redis://127.0.0.1:6379"
    connectionPoolSize: 64
    connectionMinimumIdleSize: 24
threads: 16
nettyThreads: 32

```

Beans

```

import org.redisson.Redisson;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;
import org.springframework.cache.CacheManager;

```

```

import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.io.IOException;

@Configuration
@EnableCaching
public class RedissonConfig {

    @Bean(destroyMethod = "shutdown")
    public RedissonClient redissonClient() throws IOException {
        Config config = Config.fromYAML(RedissonConfig.class.getClassLoader().getResource("redisson.yaml"));
        return Redisson.create(config);
    }

    // Use the @Cacheable annotation to use caching
    @Bean
    public RedissonSpringCacheManager cacheManager(RedissonClient redissonClient) {
        return new RedissonSpringCacheManager(redissonClient);
    }
}

```

[redisson]

Redisson

```

package org.redisson;

public class Redisson implements RedissonClient

    public RLock getLock(String name) {
        return new RedissonLock(this.commandExecutor, name);
    }
    public RLock getSpinLock(String name, LockOptions.BackOff backOff) {
        return new RedissonSpinLock(this.commandExecutor, name, backOff);
    }
    public RLock getMultiLock(RLock... locks) {
        return new RedissonMultiLock(locks);
    }
    public RReadWriteLock getReadWriteLock(String name) {
        return new RedissonReadWriteLock(this.commandExecutor, name);
    }

    public RLock getFairLock(String name) {
        return new RedissonFairLock(this.commandExecutor, name);
    }
}

```

RedissonLock

```

package org.redisson;

public class RedissonLock extends RedissonBaseLock
protected long internalLockLeaseTime;
protected final LockPubSub pubSub;
final CommandAsyncExecutor commandExecutor;

public void lock()
public void unlock()

```

lock

```
private void lock(long leaseTime, TimeUnit unit, boolean interruptibly) throws InterruptedException
```

Core Mechanism

Key Generation:

A unique key is generated based on the lock name. This key will be used in Redis for the lock operation.

Redis Command:

The SETNX command is used to acquire the lock. This command sets the value of a key only if it does not already exist.

If the key doesn't exist, the lock is acquired successfully.

If the key already exists, it means another process holds the lock, and the current thread will wait or return false depending on the lock mode (blocking or non-blocking).

Lease Time:

The SETNX command also sets an expiration time for the key. This expiration time is called the lease time.

If the lock holder doesn't renew the lease before it expires, the lock is automatically released.

Watchdog:

To prevent accidental lock loss due to network or application failures, Redisson introduces a watchdog.

The watchdog periodically renews the lock's expiration time before it expires.

Unlock:

When the lock is no longer needed, the unlock method is called.

This method uses the DEL command to delete the lock key from Redis.

```
{
    long threadId = Thread.currentThread().getId();
    Long ttl = this.tryAcquire(-1L, leaseTime, unit, threadId);
    if (ttl != null) {
        CompletableFuture<RedissonLockEntry> future = this.subscribe(threadId);
        this.pubSub.timeout(future);
        RedissonLockEntry entry;
        if (interruptibly) {
            entry = (RedissonLockEntry)this.commandExecutor.getInterrupted(future);
        } else {
            entry = (RedissonLockEntry)this.commandExecutor.get(future);
        }
    }

    // The method enters a loop to repeatedly attempt to acquire the lock.
    try {
        while(true) {
            ttl = this.tryAcquire(-1L, leaseTime, unit, threadId);
            if (ttl == null) {
                return;
            }

            if (ttl >= 0L) {
                try {
                    entry.getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
                } catch (InterruptedException var14) {
                    if (interruptibly) {
                        throw var14;
                    }

                    entry.getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
                }
            } else if (interruptibly) {
                entry.getLatch().acquire();
            } else {
                entry.getLatch().acquireUninterruptibly();
            }
        }
    } finally {
        this.unsubscribe(entry, threadId);
    }
}
```

```

        }
    }
}

tryAcquireAsync
private <T> RFuture<Long> tryAcquireAsync(long waitTime, long leaseTime, TimeUnit unit, long threadId)
{
    RFuture ttlRemainingFuture;
    if (leaseTime > 0L) {
        ttlRemainingFuture = this.tryLockInnerAsync(waitTime, leaseTime, unit, threadId,
RedisCommands.EVAL_LONG);
    } else {
        ttlRemainingFuture = this.tryLockInnerAsync(waitTime, this.internalLockLeaseTime,
TimeUnit.MILLISECONDS, threadId, RedisCommands.EVAL_LONG);
    }

    CompletionStage<Long> f = ttlRemainingFuture.thenApply((ttlRemaining) -> {
        if (ttlRemaining == null) {
            if (leaseTime > 0L) {
                this.internalLockLeaseTime = unit.toMillis(leaseTime);
            } else {
                this.scheduleExpirationRenewal(threadId);
            }
        }
    });

    return ttlRemaining;
});
return new CompletableFutureWrapper(f);
}

```

PublishSubscribe

```

package org.redisson.pubsub;
abstract class PublishSubscribe<E extends PubSubEntry<E>>

subscribe
public CompletableFuture<E> subscribe(String entryName, String channelName) {
    AsyncSemaphore semaphore = this.service.getSemaphore(new ChannelName(channelName));
    CompletableFuture<E> newPromise = new CompletableFuture();
    semaphore.acquire(() -> {
        if (newPromise.isDone()) {
            semaphore.release();
        } else {
            E entry = (PubSubEntry)this.entries.get(entryName);
            if (entry != null) {
                entry.acquire();
                semaphore.release();
                entry.getPromise().whenComplete((r, e) -> {
                    if (e != null) {
                        newPromise.completeExceptionally(e);
                    } else {
                        newPromise.complete(r);
                    }
                });
            } else {
                E value = this.createEntry(newPromise);
                value.acquire();
                E oldValue = (PubSubEntry)this.entries.putIfAbsent(entryName, value);
                if (oldValue != null) {
                    oldValue.acquire();
                    semaphore.release();
                    oldValue.getPromise().whenComplete((r, e) -> {
                        if (e != null) {

```

```

                newPromise.completeExceptionally(e);
            } else {
                newPromise.complete(r);
            }
        });
    } else {
        RedisPubSubListener<Object> listener = this.createListener(channelName, value);
        CompletableFuture<PubSubConnectionEntry> s =
this.service.subscribeNoTimeout(LongCodec.INSTANCE, channelName, semaphore, new
RedisPubSubListener[]{listener});
        newPromise.whenComplete((r, e) -> {
            if (e != null) {
                s.completeExceptionally(e);
            }
        });
        s.whenComplete((r, e) -> {
            if (e != null) {
                value.getPromise().completeExceptionally(e);
            } else {
                value.getPromise().complete(value);
            }
        });
    }
}
});
```

api

RDelayedQueue

```
package org.redisson.api;
public interface RDelayedQueue<V> extends RQueue<V>, RDestroyable
void offer(V var1, long var2, TimeUnit var4);
```

The RDelayedQueue in Redisson is a specialized queue that allows you **to delay the processing of elements** until a specified time.

This is useful for scenarios where you need to schedule tasks or process items **after a delay**.

```
void offer(V var1, long var2, TimeUnit var4);
```

Add an element to the queue with a specified delay.

This means the element will not be immediately available for consumption; instead, it will only be accessible after the given delay has passed.

```
RFuture<Void> offerAsync(V var1, long var2, TimeUnit var4);
```

Override

```
List<V> poll(int var1);
RFuture<V> pollAsync();
```

RedissonClient

```
package org.redisson.api;
public interface RedissonClient
RLock getLock(String var1);
RLock getSpinLock(String var1);
RLock getSpinLock(String var1, LockOptions.BackOff var2);
RLock getMultiLock(RLock... var1);
```

```
RLock getFairLock(String var1);
RReadWriteLock getReadWriteLock(String var1);
```

Rabbitmq Configuration

pom.xml

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

application.yml

```
[publisher-confirms, auto-ack]
spring:
  rabbitmq:
    host: localhost          # RabbitMQ server hostname or IP address.
    port: 5672                # RabbitMQ server port.
    username: guest           # Username for RabbitMQ server authentication.
    password: guest           # Password for RabbitMQ server authentication.
    virtual-host: /            # Virtual host to connect to in RabbitMQ.
    addresses:                # Comma-separated list of RabbitMQ servers for failover.
    requested-heartbeat: 60    # Heartbeat timeout in seconds.
    connection-timeout: 30000  # Connection timeout in milliseconds.
    publisher-confirms: true   # Enable publisher confirms for reliable message delivery.
    publisher-returns: true    # Enable publisher returns for undelivered messages.
    template:
      mandatory: true         # Set mandatory flag on messages to trigger returns.
      retry:
        enabled: true          # Enable retry functionality for failed messages.
        max-attempts: 3         # Maximum number of retry attempts.
        initial-interval: 1000ms # Initial interval between retries.
        multiplier: 2.0          # Multiplier for increasing the retry interval.
        max-interval: 10000ms    # Maximum interval between retries.
    listener:
      simple:
        auto-ack: false          #
                                    # true: the message is automatically acknowledged as soon as it is delivered to the consumer, regardless
                                    # of whether the consumer has successfully processed it.
                                    # false: the consumer does not automatically acknowledge messages when they are received. Instead,
                                    # the consumer must manually acknowledge the message after processing it.
        auto-startup: true         # Automatically start the container on application startup.
        concurrency: 3             # Number of concurrent consumers.
        max-concurrency: 10         # Maximum number of concurrent consumers.
        acknowledge-mode: auto     # Acknowledgment mode (none, manual, auto).
        prefetch: 1                 # Number of messages to prefetch per consumer.
      direct:
        auto-startup: true         # Automatically start the container on application startup.
        acknowledge-mode: manual   # Acknowledgment mode (none, manual, auto).
    cache:
      channel:
        size: 25                  # Size of the channel cache.
        checkout-timeout: 1000       # Timeout for checking out a channel from the cache.
      connection:
        mode: channel              # Connection factory caching mode (channel, connection).
        size: 10                   # Size of the connection cache.
    listener:
      direct:
        auto-startup: true         # Automatically start the listener container on startup.
        acknowledge-mode: manual   # Acknowledgment mode (none, manual, auto).
        concurrency: 3             # Number of concurrent consumers.
        max-concurrency: 10         # Maximum number of concurrent consumers.
```

```

prefetch: 1           # Number of messages to prefetch per consumer.

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest

spring:
  rabbitmq:
    listener:
      simple:
        auto-ack: false   # (false: acknowledge the message manually)

```

Beans

```

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Exchange;
import org.springframework.amqp.core.ExchangeBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.QueueBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {

    public static final String QUEUE_NAME = "myQueue";
    public static final String EXCHANGE_NAME = "myExchange";
    public static final String ROUTING_KEY = "myRoutingKey";

    // if a producer sends two messages to a RabbitMQ queue, a single consumer will consume those messages in
    // the order they were sent
    @Bean
    public Queue myQueue() {
        return QueueBuilder.durable(QUEUE_NAME).build();
    }

    @Bean
    public Exchange myExchange() {
        return ExchangeBuilder.topicExchange(EXCHANGE_NAME).build();
    }

    @Bean
    public Binding binding(Queue myQueue, Exchange myExchange) {
        return BindingBuilder.bind(myQueue).to(myExchange).with(ROUTING_KEY).noargs();
    }
}

```

Create a Message Producer

Basic Producer

```

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class RabbitMQProducer {

    private static final String EXCHANGE_NAME = "my_exchange";
    private static final String ROUTING_KEY = "my_routing_key";

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void sendMessage(String message) {
        rabbitTemplate.convertAndSend(EXCHANGE_NAME, ROUTING_KEY, message);
        // The routing key is used for the binding component to determine how messages are routed from

```

```

exchanges to queues.

}

public void sendMessageWithExpiration(String message, long expirationMillis) {
    Message rabbitMessage = MessageBuilder
        .withBody(message.getBytes())
        .setExpiration(String.valueOf(expirationMillis))
        .build();

    rabbitTemplate.send(EXCHANGE_NAME, ROUTING_KEY, rabbitMessage);
}

public void sendMessageWithProperties(String message) {
    MessageProperties messageProperties = new MessageProperties();
    messageProperties.setHeader("myHeader", "myValue");
    Message amqpMessage = new Message(message.getBytes(), messageProperties);

    rabbitTemplate.send(EXCHANGE_NAME, ROUTING_KEY, amqpMessage);
}

}

```

Create a Functional Message Producer

Transactional Producer

Sends messages within a transaction to ensure reliable delivery.

```

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ProducerWithTransactions {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    private static final String EXCHANGE_NAME = "my_exchange";
    private static final String ROUTING_KEY = "my_routing_key";

    public void sendMessageWithTransactions(String message) {
        rabbitTemplate.execute(channel -> {
            channel.txSelect(); // Begin transaction
            rabbitTemplate.convertAndSend(EXCHANGE_NAME, ROUTING_KEY, message);
            channel.txCommit(); // Commit transaction
            return null;
        });
    }
}

```

Asynchronous Producer

Ensures that RabbitMQ has received the message using publisher confirms.

the producer with publisher confirms **does not wait for the acknowledgments of the consumer**. Instead, it **waits for acknowledgments from RabbitMQ itself**.

```

import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageProperties;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ProducerWithConfirms {

    @Autowired
    private RabbitTemplate rabbitTemplate;

```

```
private static final String EXCHANGE_NAME = "my_exchange";
private static final String ROUTING_KEY = "my_routing_key";
```

To use the `setConfirmCallback` function in Spring AMQP with RabbitMQ, you must enable the `publisher-confirms` option. This option ensures that RabbitMQ sends an acknowledgment (ack) to the producer when it has successfully received and persisted a message.

```
public void sendMessageWithConfirms(String message) {
    rabbitTemplate.setMandatory(true); // Enable publisher confirms
    rabbitTemplate.convertAndSend(EXCHANGE_NAME, ROUTING_KEY, message, new CorrelationData(message));

    rabbitTemplate.setConfirmCallback((correlationData, ack, cause) -> {
        if (ack) {
            System.out.println(" [x] Message acknowledged by RabbitMQ");
        } else {
            System.out.println(" [x] Message not acknowledged by RabbitMQ: " + cause);
        }
    });
    System.out.println(" [x] Sent with confirms '" + message + "'");
}
}
```

Create a Message Consumer

Basic Consumer

Simple scenarios where you need to process messages as plain text or byte arrays.

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class BasicRabbitMQConsumer {

    @RabbitListener(queues = "my_queue")
    public void consume(String message) {
        System.out.println("Received Message: " + message);
    }

    @RabbitListener(queues = "my_queue")
    public void consume(Message message) {
        String body = new String(message.getBody());
        String headerValue = (String) message.getMessageProperties().getHeaders().get("my-header");
        System.out.println("Received Message: " + body + " with Header: " + headerValue);
    }
}
```

JSON Message Consumer

When messages are in JSON format and need to be converted to Java objects.

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class JsonRabbitMQConsumer {

    private final ObjectMapper objectMapper = new ObjectMapper();

    @RabbitListener(queues = "my_json_queue")
    public void consume(byte[] messageBytes) {
        try {
            MyCustomObject message = objectMapper.readValue(messageBytes, MyCustomObject.class);
            System.out.println("Received Message: " + message);
        } catch (Exception e) {

```

```

        e.printStackTrace();
    }
}
}

```

Create a Functional Message Consumer

Manual Acknowledgment

To control the acknowledgment mode of messages (auto, manual, etc.).

```

import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Service;

@Service
public class AckModeRabbitMQConsumer {

    private final RabbitTemplate rabbitTemplate;

    public AckModeRabbitMQConsumer(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    @RabbitListener(queues = "my_queue", ackMode = "MANUAL")
    public void consume(Message message, Channel channel) throws IOException {
        try {
            // Process message
            System.out.println("Received Message: " + new String(message.getBody()));
            // Manually acknowledge
            channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);
        } catch (Exception e) {
            // Handle exception and optionally reject message
            channel.basicNack(message.getMessageProperties().getDeliveryTag(), false, true);
        }
    }
}

```

Error Handling

To handle exceptions and implement retry logic when processing messages.

```

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class ErrorHandlingRabbitMQConsumer {

    @RabbitListener(queues = "my_queue")
    public void consume(String message) {
        try {
            // Process message
            System.out.println("Received Message: " + message);
        } catch (Exception e) {
            // Log and handle exception
            System.err.println("Error processing message: " + e.getMessage());
            // Optionally requeue or move message to a dead letter queue
        }
    }
}

```

Batch Message Consumer

When you need to process messages in batches for efficiency.

```

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.stereotype.Service;

```

```

@Service
public class BatchRabbitMQConsumer {

    @RabbitListener(queues = "my_queue", ackMode = "AUTO")
    public void consume(List<String> messages) {
        for (String message : messages) {
            System.out.println("Received Message: " + message);
        }
    }
}

```

Dynamic Queue Binding

To dynamically bind queues at runtime based on specific conditions or configurations.

```

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DynamicQueueRabbitMQConsumer {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = "#{dynamicQueue.name}")
    public void consume(String message) {
        System.out.println("Received Message: " + message);
    }
}

```

[amqp-client]

```

package com.rabbitmq.client;
public class ConnectionFactory          连接工厂 // ConnectionFactory factory = new ConnectionFactory();
    implements Cloneable
public Connection newConnection() throws IOException, TimeoutException      创建一个新的连接
public interface Connection           连接对象
    extends ShutdownNotifier, Closeable
Channel createChannel()             创建一个 Channel
public interface Channel            频道对象
    extends ShutdownNotifier, AutoCloseable
DeclareOk exchangeDeclare(String exchange, BuiltinExchangeType type) throws IOException;
    exchange  交换机名称
    type      交换机类型
DeclareOk queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object>
arguments) throws IOException;     声明队列 (队列名不存在则会创建该队列, 如果已存在则不会创建)
    queue      队列名称
    durable    是否持久化, 当 mq 重启, 依然存在
    exclusive  是否独占, 只能有一个消费 8005 监听这队列, 当 Connection 关闭时, 是否删除队列
    autoDelete 是否自动删除, 当没有 consumer 时, 自动删除掉
    arguments   参数
BindOk queueBind(String queue, String exchange, String routingKey) throws IOException;      绑定交换机 (需要绑定队列路由
键时, 对同一个队列多次调用即可)
    queue      队列名称
    exchange   交换机名称
    routingKey 设置队列的路由键 (如果交换机类型为 fanout, routingKey 设置为 "")

```

```

void basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body) throws IOException;
    exchange 交换机名称 (使用默认交换机设置为 "")
    routingKey 寻找符合的队列路由键 (direct: routingKey 设置为队列名 fanout: routingKey 设置为"")
    props 配置信息
    body 发送二进制消息数据
void basicAck(long deliveryTag, boolean multiple)      手动签收
    deliveryTag 消息的传递标记
    multiple 是否多个消息
void basicNack(long deliveryTag, boolean multiple, boolean requeue)  拒绝签收 (一次可以拒收多个消息)
    deliveryTag 消息的传递标记
    multiple 是否多个消息
    requeue 是否重回队列, 并且 broker 重新发送消息给消费者
void basicReject(long deliveryTag, boolean requeue)    拒绝签收 (一次只能拒收一个消息)
    deliveryTag 消息的传递标记
    requeue 是否重回队列, 并且 broker 重新发送消息给消费者

```

[spring-rabbit]

```

package org.springframework.amqp.rabbit.listener.api;
public interface ChannelAwareMessageListener      队列监听器
    extends MessageListener
void onMessage(Message message, Channel channel)      接受消息事件

package org.springframework.amqp.rabbit.core;
public class RabbitTemplate          rabbit 访问模板
    extends RabbitAccessor
    implements BeanFactoryAware, RabbitOperations, MessageListener, ListenerContainerAware, Listener, Lifecycle,
    BeanNameAware
public interface ConfirmCallback      内部接口, 确认模式
    void confirm(CorrelationData correlationData, boolean ack, String cause)
        correlationData 相关配置信息
        ack 交换机是否成功收到了消息
        cause 失败原因
public interface ReturnCallback      内部接口, 回退模式
    void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey)
        message 消息对象
        replyCode 错误码
        replyText 错误信息
        exchange 交换机
        routingKey 路由键
public void convertAndSend(String exchange, String routingKey, Object message)      发送消息
public void convertAndSend(String exchange, String routingKey, Object message, MessagePostProcessor
messagePostProcessor)    添加消息处理器发送消息
public void setConfirmCallback(RabbitTemplate.ConfirmCallback confirmCallback)    确认模式的回调, 交换机收到后执行//
setConfirmCallback( new RabbitTemplate.ConfirmCallback(){ public void confirm(...){} } )
public void setReturnCallback(RabbitTemplate.ReturnCallback returnCallback)      回退模式的回调, 交换机到队列失败后执
行 (rabbitmq 也提供了事务机制, 但是性能较差)
public void setMandatory(boolean mandatory)      回退模式触发时, 将消息返回给生产者

```

[spring-amqp]

Message

```
package org.springframework.amqp.core;  
public class Message implements Serializable  
public MessageProperties getMessageProperties()
```

MessageProperties

```
package org.springframework.amqp.core;  
public class MessageProperties implements Serializable  
public long getDeliveryTag()  
public void setExpiration(String expiration)
```

MessagePostProcessor

```
package org.springframework.amqp.core;  
public interface MessagePostProcessor  
Message postProcessMessage(Message message) throws AmqpException
```

MessageBuilder

```
package org.springframework.amqp.core;  
public final class MessageBuilder extends MessageBuilderSupport<Message>
```

MessageBuilderSupport

```
package org.springframework.amqp.core;  
public abstract class MessageBuilderSupport<T>  
public MessageBuilderSupport<T> setExpiration(String expiration)
```

Set the expiration property in the message's headers when publishing it.

This expiration time indicates how long the message should remain in the queue before it is considered expired and can be discarded or moved to a dead-letter queue (DLQ).

Spring Data JPA

Core

Workflow

Spring Boot Initialization

When a Spring Boot application starts, it performs classpath scanning and loads configuration properties.

The @SpringBootApplication annotation triggers component scanning and auto-configuration.

Auto-configures JPA repositories based on classpath settings, other beans, and various property settings by JpaRepositoriesAutoConfiguration.

EntityManagerFactory Creation

The LocalContainerEntityManagerFactoryBean configures and creates the EntityManagerFactory, and initializes the EntityManagerFactory.

The JPA provider, such as Hibernate, is responsible for creating and managing proxy instances, not the EntityManager itself.

```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory(EntityManagerFactoryBuilder  
builder, DataSource dataSource) {  
    return builder  
        .dataSource(dataSource)  
        .packages("com.example.entity") // Your entity package  
        .persistenceUnit("default")
```

```

        .build();
    }
    public void afterPropertiesSet() {
        // Create EntityManagerFactory
        this.entityManagerFactory = createNativeEntityManagerFactory();
    }
}

```

EntityManager Creation

`EntityManagerFactory` Creates `EntityManager` instances, which manage the lifecycle of entities and handle operations such as persist, merge, remove, and find.

Classes annotated with `@Entity` are recognized as JPA entities.

These classes are managed by the JPA provider (like Hibernate) and are registered with the `EntityManagerFactory`.

If lazy loading is enabled, the `EntityManager` ensures that proxies are created for lazy-loaded associations.

```

@Entity
public class Blog {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    private String content;

    // Getters and setters
}

```

Repository Creation

`JpaRepository` Provides CRUD operations on entities.

```

org.springframework.data.repository.Repository
org.springframework.data.repository.CrudRepository
org.springframework.data.repository.PagingAndSortingRepository
org.springframework.data.jpa.repository.JpaRepository
org.springframework.data.repository.query.QueryByExampleExecutor
public interface BlogRepository extends JpaRepository<Blog, Long> {
    Blog findByTitle(String title);
}

```

Application Logic Execution

A service class that uses `BlogRepository` to interact with the database.

SQL Query Execution

Spring Data JPA translates the repository method call into a JPA query.

`SimpleJpaRepository` Constructs the query from the method name.

```

public T getOne(ID id) {
    return em.getReference(getDomainClass(), id);
}

```

`EntityManager` Creates a JPA query.

```

public <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery) {
    return new QueryImpl<T>((QueryImpl<T>) criteriaQuery, this);
}

```

Transaction Management

`TransactionManager` manages transactions, ensuring data consistency.

```

@Bean
public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
    return new JpaTransactionManager(entityManagerFactory);
}

```

Annotation `@Transactional` Declares transaction boundaries.

```

@Service
@Transactional
public class BlogService {

```

```

    @Autowired
    private BlogRepository blogRepository;

    public Blog getBlogByTitle(String title) {
        return blogRepository.findByTitle(title);
    }
}

```

Proxy Creation in JPA

Lazy Loading

- In JPA, entities can have associations that are lazily loaded.
This means that the related entities are not immediately loaded from the database but are instead represented by proxy instances.
- For example, when you access a **OneToMany** or **ManyToOne** relationship annotated with **FetchType.LAZY**, a proxy instance is returned instead of the actual related entity or collection.
This proxy will load the actual data only when it is accessed.

Role of Hibernate (Default JPA Provider)

Hibernate

- Hibernate, as the default JPA provider in Spring Boot, is responsible for the actual creation and management of proxy instances.
- **HibernateProxy**
This is the interface for proxies created by Hibernate. The actual proxy classes implement this interface.
- **LazyInitializer**
This component in Hibernate is responsible for initializing the proxy when its methods are accessed.

Use Id Generator for Oracle

Create a Sequence in Oracle

You need to create a sequence in your Oracle database that will generate unique IDs.

```

CREATE SEQUENCE my_sequence
START WITH 1
INCREMENT BY 1
NOCACHE
NOCYCLE;

```

Define an Entity and Use the Sequence

Next, define your JPA entity and configure it to use the Oracle sequence for generating IDs.

You can use the **@SequenceGenerator** and **@GeneratedValue** annotations to specify the sequence.

```

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;

@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "my_sequence_gen")
    @SequenceGenerator(name = "my_sequence_gen", sequenceName = "my_sequence", allocationSize = 1)
    private Long id;

    private String name;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {

```

```

        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

Custom ID Generator

Define the Custom ID Generator

First, create a class that implements the IdentifierGenerator interface provided by Hibernate. This class will contain the logic for generating unique IDs.

```

import org.hibernate.engine.spi.SharedSessionContractImplementor;
import org.hibernate.id.IdentifierGenerator;
import org.springframework.stereotype.Component;

import java.io.Serializable;
import java.util.UUID;

@Component
public class CustomIdGenerator implements IdentifierGenerator {

    @Override
    public Serializable generate(SharedSessionContractImplementor session, Object object) {
        // Custom logic to generate unique ID
        return UUID.randomUUID().toString(); // Example using UUID
    }
}

```

Annotate the Entity with the Custom Generator

```

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name = "my_entity")
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY, generator = "custom-id-generator")
    @GenericGenerator(name = "custom-id-generator", strategy = "com.yourpackage.CustomIdGenerator")
    private String id;

    private String name;

    // Getters and setters
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {

```

```
        this.name = name;
    }
}
```

Configuration

Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc6</artifactId>          oracle 需要
    <version>11.2.0.4</version>
</dependency>
```

application-dev.yml >>

```
spring:
  jpa:
    database: oracle      specify database type
    show-sql: true         show hibernate sql query
    // Hibernate: select * from (select t1_0.STU_ID c0,t1_0.CLASS c1,t1_0.PER_ID c2,t1_0.ROLE c3 from TEST_STUDENT
t1_0 where t1_0.STU_ID=?) where rownum<=?
  datasource:
    driver-class-name: oracle.jdbc.driver.OracleDriver
    url: jdbc:oracle:thin:@localhost:1521:orcl
    username: sdk
    password: sdk
```

Usage

UserEntity >> 表实体

```
@Data
@Entity
@Table(name = "w_user_t")
@TypeDef(name = "json", typeClass = JsonStringType.class)

@MappedSuperclass
@EntityListeners(ShippingChangeListener.class) 对实体属性变化的跟踪，它提供了保存前，保存后，更新前，更新后，删除前，删除后等状态，就像是拦截器一样，你可以在拦截方法里重写你的个性化逻辑。
@GenericGenerator(name = "idGenerator", strategy = "uuid")

@NamedNativeQueries(value={ 使用 NamedNativeQueries +SqlResultSetMapping 来封装返回值
    @NamedNativeQuery(
        name = "studentInfoByld",
        query = " select * from student_info where stu_id = ? ",
        resultClass = Student.class
    ),
    @NamedNativeQuery(
        name= "studentJoinCourse",
```

```

        query = " select * from student_info,course",
        resultSetMapping = " courseInfo"
    )
}

@SqlResultSetMapping(name="courseInfo",entities=@EntityResult(entityClass=Course.class))
@SqlResultSetMapping(name="studentInfo",
    entities={
        @EntityResult(entityClass=Student.class,fields={          字段映射
            @FieldResult(name="stuid",column="student_id"),
            @FieldResult(name="sex",column="stu_sex"),
            @FieldResult(name="stuName",column="stu_name"),
            @FieldResult(name="stuAge",column="stu_age")
        }
    }

@NamedNativeQuery(
    name="CarDto",
    query="select name, color from car",
    resultSetMapping="CarDto"
)
@SqlResultSetMapping(
    name="CarDto",
    classes=@ConstructorResult(
        targetClass=CarDto.class,
        columns={
            @ColumnResult(name="name"),      如果名称为 CASE_ID 是可以自动映射到驼峰上的
            @ColumnResult(name="color")
        }
    )
)
public class UserEntity {
    @Id
    @GeneratedValue(generator = "idGenerator")
    @Column(length = 50)
    private String id;

    @CreationTimestamp
    @Column(updatable = false, nullable = false)
    @JsonProperty(value = "other_name", access = JsonProperty.Access.READ_ONLY)
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss", timezone = "GMT+8")
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @DiffIgnore
    private Timestamp createdAt;           服务端取出值时 用的字段名称

    @UpdateTimestamp
    @Column(nullable = false)
    @JsonProperty(access = JsonProperty.Access.READ_ONLY)
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss")
    @DiffIgnore
}

```

```

private Timestamp updateAt;

@Formula(value = "timestamp(shipping_timeline->'$.deliveryAckTime')")
@DiffIgnore
@Transient
private String deliveryAckTime; 非数据库的字段要加@Transient 注解

@JsonFormat(shape = JsonFormat.Shape.STRING, timezone = "GMT+8", pattern = "yyyy-MM-dd HH:mm:ss")
private Timestamp cancelTime;

@Enumerated(EnumType.STRING)
private AuthenticationCheckStatus status = AuthenticationCheckStatus.submitted; 枚举 AuthenticationCheckStatus

@Type(type = "json")
@Column(columnDefinition = "json")
private DriverSignInModel driverSignInInfoAtReturnPort = new DriverSignInModel();

@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "card_id", referencedColumnName="role_id")
private AuthenticationIDCard authenticationIDCard;

@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "pick_up_port_id", referencedColumnName="role_id")
private Port pickUpPort = new Port();

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.EAGER) 新建的 UserEntity 可
以连带创建 UserRole，但是必须指定主键等必须的字段
@JoinColumn(name = "role_id", referencedColumnName="role_id", insertable=false, updatable=false)
@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
private Set<UserRole> roles = Sets.newHashSet();

@Convert(converter = JPACryptoConverter.class) 必须实现 AttributeConverter
private String password;

public static Specification<RoutePrice> find(Long targetPortId, Long originPortId, String route) { 构建一个条件查询
    return (root, criteriaQuery, criteriaBuilder) -> {

        List<Predicate> predicateList = new ArrayList<>();

        Predicate teacherPre2 = criteriaBuilder.equal(root.get("routePriceStatus"), RoutePriceStatus.ENABLED);
        predicateList.add(teacherPre2);

        if (!StringUtils.isEmpty(targetPortId)) {
            Predicate teacherPre = criteriaBuilder.equal(root.join("targetPort").get("id"), targetPortId);
            predicateList.add(teacherPre);
        }
    }
}

```

```

        }

        if (!StringUtils.isEmpty(originPortId)) {
            Predicate teacherPre = criteriaBuilder.equal(root.join("originPort").get("id"), originPortId);
            predicateList.add(teacherPre);
        }

        if (!StringUtils.isEmpty(route)) {
            Predicate teacherPre = criteriaBuilder.equal(root.get("route"), route);
            predicateList.add(teacherPre);
        }

        return criteriaQuery.where(predicateList.toArray(new Predicate[predicateList.size()])).getRestriction();
    };
}

}

```

entity 使用

UserService >> 使用

```

@Repository
@Transactional
public class Dao<T, ID extends Serializable> implements GenericDao<T, ID> {

    @PersistenceContext
    private EntityManager entityManager; //实体管理对象

    @Override
    public <S extends T> S persist(S entity) {

        Objects.requireNonNull(entity, "Cannot persist a null entity");

        entityManager.persist(entity);

        return entity;
    }

    @Transactional(readOnly=true)
    public List<CarDto> fetchCars() {
        Query query = entityManager.createNamedQuery("CarDto");
        List<CarDto> result = query.getResultList();

        return result;
    }

    protected EntityManager getEntityManager() {
        return entityManager;
    }
}
```

```
public class StudentDao {  
    public Student queryStudentById(EntityManager entitymanager,Integer id){  
        return entitymanager.createNamedQuery("studentInfoById", Student.class).setParameter(1, 1).getSingleResult();  
    }  
}
```

entityListener 使用

@Slf4j

@Component

@Transactional

```
public class ShippingChangeListener {      Jpa Listener 不被 Spring 管理，并且在 ApplicationContext 准备前就创建了  
    @Autowired  
    private ShippingLogDao shippingLogDao; // 在 listener 里调用 repository 会导致死循环，repository.findAll() 会刷新数据，  
导致 PostUpdate 再次触发
```

```
private Javers javers;
```

```
public ShippingChangeListener() {
```

```
    javers = JaversBuilder.javers().registerJaversRepository(new InMemoryRepository(CommitIdGenerator.RANDOM))  
判断两个类改动字段有哪些
```

```
    .registerCustomComparator(new CustomDoubleIgnorePrecisionComparator(), double.class).build();  
}
```

```
@PostPersist      // 新增时触发，更新不触发
```

```
@Transactional(Transaction.TxType.REQUIRES_NEW)
```

```
public void postPersist(Shipping shipping) {
```

```
/*
```

```
private Long userId;
```

```
private String userName;
```

```
private String type;
```

```
private Long shippingId;
```

```
@Type(type = "json")
```

```
@Column(columnDefinition = "json")
```

```
private Shipping shipping;
```

```
private String diff;
```

```
*/
```

```
ShippingLog shippingLog = new ShippingLog();
```

```
User curUser = UserUtil.getCurrentUser();
```

```
shippingLog.setUserId(curUser.getId());
```

```
shippingLog.setUserName(curUser.getUsername());
```

```
shippingLog.setType("CREATE");
```

```
shippingLog.setShippingId(shipping.getId());
```

```
shippingLog.setShipping(shipping);
```

```
shippingLogDao.save(shippingLog);
```

```
}
```

```

@PostUpdate // 更新时触发，新增不触发
@Transactional(Transaction.TxType.REQUIRES_NEW)
public void postUpdate(Shipping shipping) {
    ShippingLog shippingLog = new ShippingLog();
    User curUser = UserUtil.getCurrentUser();
    shippingLog.setUserId(curUser.getId());
    shippingLog.setUserName(curUser.getUsername());
    shippingLog.setType("UPDATE");
    shippingLog.setShippingId(shipping.getId());
    shippingLog.setShipping(shipping);
    ShippingLog oldShipping = shippingLogDao.getFirstByShippingIdOrderByCreateAtDesc(shippingLog.getShippingId());
    if (oldShipping != null) {
        Diff diff = javers.compare(oldShipping.getShipping(), shipping);
        shippingLog.setDiff(diff.toString());
    }
    shippingLogDao.save(shippingLog);
}

@PostRemove
@Transactional(Transaction.TxType.REQUIRES_NEW)
public void postRemove(Shipping shipping) {
    ShippingLog shippingLog = new ShippingLog();
    User curUser = UserUtil.getCurrentUser();
    shippingLog.setUserId(curUser.getId());
    shippingLog.setUserName(curUser.getUsername());
    shippingLog.setType("DELETE");
    shippingLog.setShippingId(shipping.getId());
    shippingLog.setShipping(shipping);
    shippingLogDao.save(shippingLog);
}

public static class CustomDoubleIgnorePrecisionComparator implements CustomPropertyComparator<Double,
ValueChange> {
    private static final double DELTA = 0.00001;
    @Override
    public ValueChange compare(Double aDouble, Double t1, GlobalId globalId, Property property) {
        double diff = Math.abs(aDouble - t1);
        if (diff <= DELTA) {
            return null;
        }
        return new ValueChange(globalId, property.getName(), aDouble, t1);
    }
}
}

```

repository 使用

```

@Data
@JsonInclude(JsonInclude.Include.NON_NULL)
public interface ShipperDriverTeamContractDao extends PagingAndSortingRepository<ShipperDriverTeamContract, Long> {
    @Query(value = "select name,age from student where studentid=:studentid ",nativeQuery = true) 原生查询
    Map<String, Object> findById(@param("studentid") String studentid);

    @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1", countQuery = "SELECT count(*) FROM USERS WHERE
LASTNAME = ?1", nativeQuery = true)
    Page<User> findByLastname(String lastname, Pageable pageable); 原生分页查询

    Page<RoutePrice> findAll(Specification<RoutePrice> specification, Pageable var2); 根据规格和分页器查询列表（使用
hql 语句， PageRequest 类构建 Pageable）

    Page<User> getUsersPage(PageParam pageParam, String nickName) {
        Specification<User> specification = new Specification<User>() { 根据断言构造一个规格
            @Override
            public Predicate toPredicate(Root<User> root, CriteriaQuery<?> query, CriteriaBuilder cb) { //返回断言
                List<Predicate> predicates = new ArrayList<>();
                if(StringUtils.isNotBlank(nickName)){
                    Predicate likeNickName = cb.like(root.get("nickName").as(String.class),nickName+"%");
                    predicates.add(likeNickName);
                }
                return cb.and(predicates.toArray(new Predicate[0]));
            }
        };
        Pageable pageable = new PageRequest(pageParam.getPage()-1,pageParam.getLimit()); //分页信息
        return this.userRepository.findAll(specification,pageable);
    }

    Page<User> queryFirst10ByLastname(String lastname, Pageable pageable); PagingAndSortingRepository 自带的分页功
能

    Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
    // userRepository.findAll( Sort.by("createTime"));
    userRepository.findAll( Sort.by("author").ascending.and(Sort.by("createTime").descending)); 先按 author 升序排序再按照
createTime 降序排序
    // Pageable pageable = PageRequest.of( 0, 10, Sort.by("createTime")) 分页和排序一起

    @Transactional
    @Modifying(clearAutomatically = true) // 在执行此查询后不会清除持久性上下文，要手动添加 (Modifying queries can
only use void or int/Integer as return type)
    @Query(value= "delete from OverseaServiceEntity t where t.booking = ?1 ") 删除
    void deleteAllByBooking(Booking booking);

    List<ShipperDriverTeamContract> findAllByShipper(User shipper);
    List<Shipping> findByDriverIdAndStatusAndDeliveryAckTimeBetween(Long driverId, ShippingStatus status, String

```

```

beginTime,      String endTime);

@Query("select s from Shipping s where s.driverId=?1 and (s.status=?2 or s.status=?3 or s.status=?4 or s.status=?5) and
s.deliveryAckTime >?6 and s.deliveryAckTime < ?7" )    使用计算字段 deliveryAckTime
List<Shipping> findByDriverIdAndStatusAndDeliveryAckTimeBetween(Long driverId,
                                                               ShippingStatus status2,ShippingStatus status3,ShippingStatus
status4,ShippingStatus status5,
                                                               String beginTime,
                                                               String endTime);

@Query(value = "select distinct t.targetPort from RoutePrice t where t.originPort.id = :originPortId and
t.routePriceStatus=:status")  根据对象的字段筛选
List<Port> findAllByOriginPortIdAndRoutePriceStatus(Long originPortId,RoutePriceStatus status);

}

```

PersonRepositoryImpl >> 自定义 repository

```

@Repository
public class PersonRepositoryImpl implements PersonDao{

    @PersistenceContext
    private EntityManager em;

    @Override
    public void test() {
        //只是用来测试
        Person person = em.find(Person.class, 1);
        System.out.println(person);
    }
}

```

service 使用

```

@Service
@Transactional
public class IncomeService{

    @PersistenceContext
    EntityManager entityManager;

    public Page<IncomeDaily> findIncomeDailysByPage(PageParam pageParam, String cpld, String appId, Date start, Date
end, String sp) {
        StringBuilder countSelectSql = new StringBuilder();
        countSelectSql.append("select count(*) from IncomeDaily po where 1=1 ");

        StringBuilder selectSql = new StringBuilder();
        selectSql.append("from IncomeDaily po where 1=1 ");

```

```

Map<String, Object> params = new HashMap<>();
StringBuilder whereSql = new StringBuilder();
if(StringUtils.isNotBlank(cpld)){
    whereSql.append(" and cpld=:cpld ");
    params.put("cpld",cpld);
}
if (end != null)
{
    whereSql.append(" and po.bizDate <= :endTime");
    params.put("endTime", end);
}

String countSql = new StringBuilder().append(countSelectSql).append(whereSql).toString();
Query countQuery = this.entityManager.createQuery(countSql, Long.class);
this.setParameters(countQuery, params);
Long count = (Long) countQuery.getSingleResult();

String querySql = new StringBuilder().append(selectSql).append(whereSql).toString();
Query query = this.entityManager.createQuery(querySql, IncomeDaily.class);
this.setParameters(query,params);
if(pageParam != null){ //分页
    query.setFirstResult(pageParam.getStart());
    query.setMaxResults(pageParam.getLength());
}

List<IncomeDaily> incomeDailyList = query.getResultList();
if(pageParam != null) { //分页
    Pageable pageable = new PageRequest(pageParam.getPage(), pageParam.getLength());
    Page<IncomeDaily> incomeDailyPage = new PageImpl<IncomeDaily>(incomeDailyList, pageable, count);
    return incomeDailyPage;
} else{ //不分页
    return new PageImpl<IncomeDaily>(incomeDailyList);
}
}

/***
 * 给 hql 参数设置值
 */
private void setParameters(Query query, Map<String, Object> params){
    for(Map.Entry<String, Object> entry:params.entrySet()){
        query.setParameter(entry.getKey(), entry.getValue());
    }
}
}

controller >>
Sort sort = Sort.by(Sort.Order.desc("createAt"));
Pageable pageable = PageRequest.of(page, size, sort);

```

Specification<RoutePrice> specification = RoutePrice.find(originPortId, targetPortId, route); 返回一个 Specification

Seata

Core

Seata (Simple Extensible Autonomous Transaction Architecture) is an open-source distributed transaction solution that provides high performance and easy-to-use distributed transaction services:

Seata implements distributed transactions internally using a two-phase commit (2PC) protocol.

CAP and BASE

The CAP and BASE theories are two fundamental concepts in the realm of distributed systems, especially when it comes to understanding the trade-offs between consistency, availability, and partition tolerance.

CAP Theorem

The CAP theorem, also known as Brewer's theorem, states that in a distributed data store, it is impossible to simultaneously guarantee all three of the following:

Consistency (C)

Every read receives the most recent write or an error. In other words, all nodes in a distributed system have the same data at the same time.

Availability (A)

Every request (read or write) receives a response, regardless of whether it was successful or failed. The system is always operational and responds to requests.

Partition Tolerance (P)

The system continues to operate despite network partitions. A network partition is a communication breakdown between nodes, causing them to be split into isolated groups.

Trade-offs in CAP

According to the CAP theorem, in the presence of a network partition, a distributed system has to choose between:

Consistency and Partition Tolerance (CP)

The system will maintain consistency across all nodes but may become unavailable during a partition.

Availability and Partition Tolerance (AP)

The system will remain available to respond to requests but may not guarantee consistency during a partition.

BASE Theory

BASE is an alternative approach to the traditional ACID properties of databases (Atomicity, Consistency, Isolation, Durability).

BASE stands for:

Basically Available (BA)

The system guarantees availability, meaning that the system is always available to handle requests, even if some parts of it are degraded.

Soft State (S)

The system's state may change over time, even without input. This implies that the system does not have to be consistent all the time.

Eventual Consistency (E)

The system will eventually become consistent once it stops receiving new updates. This means that, given enough time, all replicas will converge to the same state.

Distributed Algorithm

Two-Phase Commit (2PC)

Two-Phase Commit (2PC) is a distributed algorithm that ensures all participants in a distributed transaction either commit or roll back their changes, achieving atomicity. It consists of two phases:

Prepare Phase

The coordinator node sends a prepare request to all participant nodes.

Each participant node executes the transaction up to the point where it is ready to commit and then votes to either

commit or abort.

Commit Phase

If all participants vote to commit, the coordinator sends a commit request to all participants.

If any participant votes to abort, the coordinator sends an abort request to all participants.

Each participant then commits or rolls back the transaction based on the coordinator's decision.

Three-Phase Commit (3PC)

Three-Phase Commit (3PC) is an extension of 2PC that adds an additional phase to reduce the chances of blocking in case of a coordinator failure. It includes:

Prepare Phase

Similar to 2PC, the coordinator sends a prepare request to all participants.

Pre-Commit Phase

If all participants vote to commit, the coordinator sends a pre-commit request to all participants.

Participants acknowledge the pre-commit request, ensuring they are ready to commit.

Commit Phase

The coordinator sends a final commit request to all participants.

Participants commit the transaction and release any held resources.

Components

Transaction Manager (TM)

Initiates and manages global transactions on the client side.

Resource Manager (RM)

Manages resources involved in a transaction, registers branch transactions, and drives commit or rollback operations.

Transaction Coordinator (TC)

Manages global transactions, maintains transaction status, and coordinates commits and rollbacks.

Process Flow

Transaction Initiation

- The TM initiates a global transaction and assigns a global transaction ID.
- The TM registers the transaction with the TC.

Branch Transaction Execution

- The TM sends a branch transaction request to each participating RM.
- Each RM enlists in the global transaction and executes the branch transaction.

Branch Transaction

A local transaction executed within a resource manager as part of a global transaction.

Each branch transaction has a unique branch ID assigned by the TM.

Branch transactions are executed in a consistent order to ensure atomicity.

- The RM records the outcome of the branch transaction (commit or rollback).

Commit/Rollback Decision

- The TM reports the status of each branch transaction to the TC.
- The TC analyzes the transaction status and determines whether to commit or rollback the transaction.

Commit/Rollback Execution

- The TC instructs the TM to commit or rollback the transaction.
- The TM sends the corresponding request to each RM.
- Each RM executes the corresponding action on its branch transaction.

Additional Considerations

Timeout:

The TM sets a timeout for the transaction to prevent indefinite waiting.

Retry:

If a branch transaction fails, the TM can retry the transaction.

Compensation:

In case of a rollback, Seata can use compensation mechanisms to undo the effects of the failed transactions.

Transaction Mode:

Seata supports different transaction modes (AT, TCC, Saga) to adapt to various use cases.

Configuration

Add Dependencies

```
<dependency>
    <groupId>io.seata</groupId>
    <artifactId>seata-spring-boot-starter</artifactId>
    <version>1.7.0</version>
</dependency>
```

Configure Seata Properties

```
seata:
    application-id: your_application_id
        The unique identifier for your application within the Seata cluster.
    server-addr: 127.0.0.1:8091
        The address of the Seata server (e.g., 127.0.0.1:8091).
    service-group-name: default
        The name of the service group to which your application belongs.
    register-server-mode: standalone
        The mode in which the Seata client registers with the server (e.g., standalone, file, db).

# Registry configuration (for distributed environments)
registry-type: nacos
    The type of registry used for service discovery (e.g., nacos, zookeeper, etcd).
registry-address: nacos://127.0.0.1:8848
    The address of the registry server (e.g., nacos://127.0.0.1:8848).

# File configuration (for standalone mode)
file-config-dir: /path/to/your/config/dir

# Database configuration (for database storage mode)
store-mode: db
    The storage mode for transaction data (e.g., db).
db-datasource-url: jdbc:mysql://localhost:3306/seata?useUnicode=true&characterEncoding=UTF-
8&serverTimezone=UTC
db-datasource-username: seata
db-datasource-password: your_password

# Log configuration
log-level: info
log-file-path: /path/to/your/log/file.log

# Other properties
client-id: your_client_id
    The client ID for authentication with the Seata server.
client-secret: your_client_secret
    The client secret for authentication with the Seata server.
retry-times: 3
    The number of times to retry a failed transaction.
retry-interval: 5000
```

The interval between retries.

`timeout: 30000`

The timeout for the transaction.

Transaction configuration

`transaction-role: TM`

The role of the application in the transaction (e.g., TM, RM).

`transaction-id-generator-class: io.seata.core.transaction.id.DefaultTransactionIdGenerator`

`transaction-group-name: my-transaction-group`

The name of the transaction group.

`transaction-mode: TCC`

The transaction mode (e.g., AT, TCC, Saga).

AT (Atomic Transaction) Mode [Default]

Core principle: Based on two-phase commit (2PC) and XA protocol.

Process:

- The transaction manager (TM) initiates a global transaction and assigns a global transaction ID.
- The TM sends branch transactions to participating resource managers (RMs).
- The RMs execute the branch transactions and record the outcome.
- The TM collects the status of all branch transactions and decides whether to commit or rollback the global transaction.
- If all branch transactions are successful, the TM sends a commit request to all RMs.
- If any branch transaction fails, the TM sends a rollback request to all RMs.

Advantages: Simple to implement and widely supported.

Disadvantages: May suffer from performance degradation in high-concurrency scenarios due to the 2PC protocol.

TCC (Try-Confirm-Cancel) Mode

Core principle: A more flexible approach that allows for compensation operations.

Process:

- The TM initiates a global transaction and assigns a global transaction ID.
- The TM calls the "try" method on each RM to reserve resources.
- If all "try" methods are successful, the TM calls the "confirm" method on each RM to commit the transaction.
- If any "try" method fails, the TM calls the "cancel" method on each RM to roll back the transaction.

Advantages: More flexible than AT mode, can handle complex scenarios.

Disadvantages: Requires developers to implement "try", "confirm", and "cancel" methods for each resource.

Saga Mode

Core principle: A distributed transaction pattern that **uses compensation actions** to ensure consistency.

Process:

- The TM initiates a global transaction and defines a sequence of local transactions.
- The TM executes the local transactions one by one.
- If any local transaction fails, the TM executes the corresponding compensation action to roll back the previous successful transactions.

Advantages: Highly flexible and can handle complex scenarios.

Disadvantages: Requires careful design and implementation of compensation actions.

Choosing the Right Mode:

The choice of transaction mode depends on your specific application requirements. Consider the following factors:

Complexity of your application: For simple applications, AT mode may be sufficient. For more complex scenarios, TCC or Saga mode may be more suitable.

Performance requirements: AT mode may suffer from performance degradation in high-concurrency scenarios.

TCC and Saga mode can offer better performance in such cases.

Development effort: TCC and Saga mode require more development effort to implement compensation actions.

Enable Seata Transaction Manager

Add the `@EnableGlobalTransaction` annotation to your main Spring Boot application class.

Use Seata Transaction Annotations

Annotate your service methods with `@GlobalTransactional` to enable distributed transaction support.

```
@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @GlobalTransactional
    public void createOrder(Order order) {
        // Business logic to create the order
        orderRepository.save(order);
        // ... other operations
    }
}
```

Zookeeper [2]

Core

Distributed Lock

Create a service that uses Curator's InterProcessMutex for distributed locking:

```
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.locks.InterProcessMutex;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.concurrent.TimeUnit;

@Service
public class DistributedLockService {

    private final CuratorFramework client;

    @Autowired
    public DistributedLockService(CuratorFramework client) {
        this.client = client;
    }

    public void performTaskWithLock() {
        InterProcessMutex lock = new InterProcessMutex(client, "/my/lock/path");
        boolean acquired = false;
        try {
            // Acquire the lock
            acquired = lock.acquire(5, TimeUnit.SECONDS);
            if (acquired) {
                // Perform the task
                System.out.println("Lock acquired, performing task...");
                // Simulate task
                Thread.sleep(2000);
            } else {
                System.out.println("Could not acquire lock");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (acquired) {
```

```
        try {
            // Release the lock
            lock.release();
            System.out.println("Lock released");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Configuration

Add Dependencies

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>5.2.0</version>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>5.2.0</version>
</dependency>
```

Configure ZooKeeper Properties

zookeeper.connect-string=localhost:2181

`zookeeper.session-timeout=30000`

Create ZooKeeper Configuration Class

Create a configuration class to set up the CuratorFramework client:

```
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.retry.ExponentialBackoffRetry;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ZookeeperConfig {

    @Value("${zookeeper.connect-string}")
    private String connectString;

    @Value("${zookeeper.session-timeout}")
    private int sessionTimeout;

    @Bean(initMethod = "start", destroyMethod = "close")
    public CuratorFramework curatorFramework() {
        return CuratorFrameworkFactory.newClient(connectString, sessionTimeout, sessionTimeout, new
ExponentialBackoffRetry(1000, 3));
    }
}
```

Create a Service to Interact with ZooKeeper

Create a service class that uses the CuratorFramework client to interact with ZooKeeper:

```

import org.apache.curator.framework.CuratorFramework;
import org.apache.zookeeper.CreateMode;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ZookeeperService {

    @Autowired
    private CuratorFramework curatorFramework;

    public void createNode(String path, byte[] data) throws Exception {
        curatorFramework.create()
            .creatingParentsIfNeeded()
            .withMode(CreateMode.PERSISTENT)
            .forPath(path, data);
    }

    public byte[] getData(String path) throws Exception {
        return curatorFramework.getData().forPath(path);
    }

    public void updateData(String path, byte[] data) throws Exception {
        curatorFramework.setData().forPath(path, data);
    }

    public void deleteNode(String path) throws Exception {
        curatorFramework.delete().forPath(path);
    }
}

```

Use the Service in a Controller or Another Service

Create a controller to expose ZooKeeper operations via HTTP endpoints:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/zookeeper")
public class ZookeeperController {

    @Autowired
    private ZookeeperService zookeeperService;

    @PostMapping("/create")
    public String createNode(@RequestParam String path, @RequestParam String data) {
        try {
            zookeeperService.createNode(path, data.getBytes());
            return "Node created successfully";
        } catch (Exception e) {
            e.printStackTrace();
            return "Error creating node: " + e.getMessage();
        }
    }

    @GetMapping("/get")
    public String getData(@RequestParam String path) {
        try {
            byte[] data = zookeeperService.getData(path);
            return new String(data);
        } catch (Exception e) {
            e.printStackTrace();
            return "Error getting data: " + e.getMessage();
        }
    }
}

```

```

@PutMapping("/update")
public String updateData(@RequestParam String path, @RequestParam String data) {
    try {
        zookeeperService.updateData(path, data.getBytes());
        return "Data updated successfully";
    } catch (Exception e) {
        e.printStackTrace();
        return "Error updating data: " + e.getMessage();
    }
}

@DeleteMapping("/delete")
public String deleteNode(@RequestParam String path) {
    try {
        zookeeperService.deleteNode(path);
        return "Node deleted successfully";
    } catch (Exception e) {
        e.printStackTrace();
        return "Error deleting node: " + e.getMessage();
    }
}
}

```

[curator-client]

```

package org.apache.curator;
public interface RetryPolicy

```

```

package org.apache.curator.retry;
public class ExponentialBackoffRetry 重试策略
    extends SleepingRetry
public ExponentialBackoffRetry(int baseSleepTimeMs, int maxRetries)

```

[curator-framework]

```

package org.apache.curator.framework;
public class CuratorFrameworkFactory 连接工厂
public static CuratorFramework newClient(String connectString, int sessionTimeoutMs, int connectionTimeoutMs, RetryPolicy
retryPolicy)

```

connectstring	连接字符串。zk server 地址和端口 "192.168.149.135:2181,192.168.149.136:2181"
sessionTimeoutMs	会话超时时间单位 ms
connectionTimeoutMs	连接超时时间单位 ms
retryPolicy	重试策略

```

package org.apache.curator.framework;
public interface CuratorFramework 节点操作
    extends Closeable
CreateBuilder create() 创建节点操作
GetDataBuilder getData() 查询节点操作
GetChildrenBuilder getChildren() 查询子节点操作
SetDataBuilder setData(); 修改节点操作

```

```

package org.apache.curator.framework.listen;
public class ListenerContainer<T> 监听容器
    implements Listenable<T>
public void addListener(T listener) 添加一个监听器 (可以使用匿名类 NodeCacheListener)

```

```

package org.apache.curator.framework.api;

```

```

public interface CreateBuilder          创建节点操作
    extends BackgroundPathAndBytesable<String>, CreateModable<ACLBackgroundPathAndBytesable<String>>,
           ACLCreateModeBackgroundPathAndBytesable<String>, Compressible<CreateBackgroundModeACLable>
protected ACLCreateModePathAndBytesable<String> creatingParentsIfNeeded();   创建父节点选择，多级创建时需要

package org.apache.curator.framework.api;
public interface CreateModable<T>      创建模式选择（创建选项）
T withMode(CreateMode var1);           添加模式，默认持久模式，意外断开节点不会删除临时节点

package org.apache.curator.framework.api;
public interface BackgroundPathAndBytesable<T>
    extends Backgroundable<ErrorListenerPathAndBytesable<T>>, PathAndBytesable<T>

package org.apache.curator.framework.api;
public interface PathAndBytesable<T>      路径选择（选项）
T forPath(String var1) throws Exception   创建无数据的节点
T forPath(String var1, byte[] var2) throws Exception  创建带数据的节点（没有指定数据，则将当前客户端的 ip 作为数据存储）

package org.apache.curator.framework.api;
public interface Pathable<T>            路径选择（选项）
T forPath(String var1) throws Exception;

package org.apache.curator.framework.api;
public interface Storable<T>           状态查询（选项）
T storingStatIn(Stat var1)             将状态存到自定义的状态对象中 // 默认 0,0,0,0,0,0,0,0,0 ==>
25,25,1647508131679,1647508131679,0,1,0,0,12,1,32
public interface Versionable<T>        版本查询（选项）
T withVersion(int var1)               版本选择

```

[curator-recipes]
locks

InterProcessMutex

```

package org.apache.curator.framework.recipes.locks;
public class InterProcessMutex implements InterProcessLock, Revocable<InterProcessMutex>

```

Internal Workflow

Acquire Lock:

When a client wants to acquire the lock, it creates **an ephemeral sequential node** in a specified lock path, e.g., /locks/mylock/lock-.

Zookeeper **appends a monotonically increasing number** to this path, e.g., /locks/mylock/lock-0000000000, /locks/mylock/lock-0000000001, etc.

The client then retrieves the list of children in the lock path and determines its position based on the sequence number.

Determine Lock Ownership:

If the client's node **has the smallest sequence number**, it has acquired the lock.

If not, the client **watches the node with the next smallest sequence number** (i.e., the node immediately before its node).

For example, if the client's node is /locks/mylock/lock-0000000003, it sets a watch on /locks/mylock/lock-0000000002.

This ensures that when the preceding node is deleted (indicating that the previous lock holder has released the lock), the client will be notified and can check if it can now acquire the lock.

Release Lock:

When the client releases the lock, it deletes its ephemeral sequential node.

This deletion triggers the watch on the next node, allowing the next client in line to acquire the lock.

```
private final LockInternals internals;
private final String basePath;
private final ConcurrentMap<Thread, LockData> threadData;
private static final String LOCK_NAME = "lock-";
```

cache

NodeCache

```
package org.apache.curator.framework.recipes.cache;
public class NodeCache          监听一个节点
    implements Closeable
public NodeCache(CuratorFramework client, String path)
public ListenerContainer<NodeCacheListener> getListenable()      获取监听容器
public void start(boolean buildInitial)      开始监听
    buildInitial      开启监听时, 是否加载缓冲数据
public ChildData getCurrentData()      获取当前数据
```

PathChildrenCache

```
package org.apache.curator.framework.recipes.cache;
public class PathChildrenCache      监控一个节点的所有子节点.
public PathChildrenCache(CuratorFramework client, String path, boolean cacheData)
```

TreeCache

```
package org.apache.curator.framework.recipes.cache;
public class TreeCache          可以监控整个树上的所有节点, 类似于 PathChildrenCache 和 NodeCache 的组合
```

PathChildrenCacheEvent

```
package org.apache.curator.framework.recipes.cache;
public class PathChildrenCacheEvent      所有子节点事件
public PathChildrenCacheEvent.Type getType()  获取发生的事件类型
public ChildData getData()      获取改变后的数据
```

ChildData

```
package org.apache.curator.framework.recipes.cache;
public class ChildData      改变后的数据
    implements Comparable<ChildData>
public byte[] getData()      获取当前节点数据
```

NodeCacheListener

```
package org.apache.curator.framework.recipes.cache;
public interface NodeCacheListener      监听一个节点
void nodeChanged() throws Exception;      节点改变事件
```

PathChildrenCacheListener

```
package org.apache.curator.framework.recipes.cache;
public interface PathChildrenCacheListener 监控一个节点的所有子节点.
void childEvent(CuratorFramework var1, PathChildrenCacheEvent var2) throws Exception 节点改变事件
```

TreeCacheListener

```
package org.apache.curator.framework.recipes.cache;
public interface TreeCacheListener 可以监控整个树上的所有节点, 类似于 PathChildrenCache 和 NodeCache 的组合
void childEvent(CuratorFramework var1, TreeCacheEvent var2) throws Exception;
[zk]
package org.apache.zookeeper;
public interface Watcher 原生监听事件, 只能触发一次
void process(WatchedEvent var1) 触发回调
public class ZooKeeper 连接客户端
public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher) throws IOException 建立连接 //
ZooKeeper("192.168.100.100:2181",30*1000, new Watcher() )
public Stat exists(String path, Watcher watcher) throws KeeperException, InterruptedException 触发时机: 节点被创建,
节点数据被修改, 节点被删除
public void getData(String path, Watcher watcher, DataCallback cb, Object ctx) 触发时机: 节点被创建,
节点数据被修改, 节点被删除
public List<String> getChildren(String path, Watcher watcher) throws KeeperException, InterruptedException 触发时机:
节点被删除, 子节点被创建, 子节点被删除
```

```
package org.apache.zookeeper.data;
public class Stat 保存各种节点信息, 对应 Storable 的 storingStatIn 方法进行查询。
    implements Record
public Stat(long czxid, long mzxid, long ctime, long mtime, int version, int cversion, int aversion, long ephemeralOwner, int
dataLength, int numChildren, long pzxid) 状态对象构造函数
public int getVersion() 对应 zookeeper 客户端的 dataVersion 数据版本
```

原生方式使用

```
final CountDownLatch countDownLatch=new CountDownLatch(1);
final ZooKeeper zooKeeper = new ZooKeeper("192.168.254.135:2181,192.168.254.136:2181,192.168.254.137:2181", 4000, new
Watcher() {
    @Override
    public void process(WatchedEvent event) {
        System.out.println("默认事件: " + event.getType());
        if(Event.KeeperState.SyncConnected==event.getState()){
            //如果收到了服务端的响应事件, 连接成功
            countDownLatch.countDown();
        }
    }
});
countDownLatch.await(); //使用 CountDownLatch 等待两个守护线程执行完毕、建立连接后再执行之后的操作 (避免连接还未建立,
就执行下方代码)

zooKeeper.create("/zk-wuzz","1".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
//通过 exists 绑定事件
Stat stat=zooKeeper.exists("/zk-wuzz", new Watcher() {
```

```

@Override
public void process(WatchedEvent event) {
    System.out.println(event.getType()+"->"+event.getPath());
    try {
        //再一次去绑定事件 ,但是这个走的是默认事件
        zooKeeper.exists(event.getPath(),true);
    } catch (KeeperException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
//通过修改的事务类型操作来触发监听事件
stat=zooKeeper.setData("/zk-wuzz","2".getBytes(),stat.getVersion());
Thread.sleep(1000);
zooKeeper.delete("/zk-wuzz",stat.getVersion());
System.in.read();

```

建立连接使用

CuratorTest >>

```

@SpringBootTest
public class CuratorTest {
    private CuratorFramework client;

    @Before
    public void testConnect() throws Exception {
        RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000, 10);
        //    CuratorFramework client= CuratorFrameworkFactory.newClient("192.168.22.129:2181",60*1000,15*1000,retryPolicy);
        client = CuratorFrameworkFactory.builder().connectString("192.168.22.129:2181")
            .sessionTimeoutMs(68 * 1000)
            .connectionTimeoutMs(15 * 1000)
            .retryPolicy(retryPolicy).namespace("sdk").build();
        client.start();
    }

    @Test
    public void testData() throws Exception {
    //=====
    ===== 节点创建
        client.create().creatingParentsIfNeeded().forPath("/test/lala1");
    //=====
    ===== 节点查询
        byte[] data = client.getData().forPath("/test");
        System.out.println(new String(data));
    }
}

```

```
//  
===== 子节点查询  
List<String> data = client.getChildren().forPath("/test");  
System.out.println(data);  
//  
===== 状态查询  
Stat status = new Stat();  
System.out.println(status);  
client.getData().storingStatIn(status).forPath("/test/lala1");  
System.out.println(status);  
int version = status.getVersion();  
//  
===== 修改节点，添加乐观锁版本操作  
client.setData().withVersion(version).forPath("/test/lala1", "itcast".getBytes());  
//  
===== 监听一个节点  
NodeCache nodeCache = new NodeCache(client, "/test");  
nodeCache.getListenable().addListener(new NodeCacheListener() {  
    @Override  
    public void nodeChanged() throws Exception {  
        System.out.println("[/test] node changed");  
        //获取修改节点后的数据  
        byte[] data = nodeCache.getCurrentData().getData();  
        System.out.println(new String(data));  
    }  
});  
nodeCache.start(true);  
//  
===== 监听所有子节点  
PathChildrenCache pathChildrenCache = new PathChildrenCache(client, "/test", true);  
pathChildrenCache.getListenable().addListener(new PathChildrenCacheListener() {  
    @Override  
    public void childEvent(CuratorFramework curatorFramework, PathChildrenCacheEvent pathChildrenCacheEvent)  
throws Exception {  
        System.out.println("[/test] node changed");  
        System.out.println(pathChildrenCacheEvent);  
        PathChildrenCacheEvent.Type type = pathChildrenCacheEvent.getType();  
        if (type.equals(PathChildrenCacheEvent.Type.CHILD_UPDATED)) {  
            byte[] data = pathChildrenCacheEvent.getData().getData();  
            System.out.println(new String(data));  
        }  
    }  
}
```

```

    });
    pathChildrenCache.start();
}

=====
===== 监听节点和其所有子节点
TreeCache treeCache = new TreeCache(client, "/test");
treeCache.getListenable().addListener(new TreeCacheListener() {
    @Override
    public void childEvent(CuratorFramework curatorFramework, TreeCacheEvent treeCacheEvent) throws Exception {
        System.out.println("[/test] node changed");
    }
});
treeCache.start();
}

=====
===== 线程循环不管断
    while (true) {
    }
}
}

@After
public void testConnectClose() throws Exception {
    if (client != null) {
        client.close();
    }
}

}

```

分布式锁使用

Ticket12306 >>

```

public class Ticket12306 implements Runnable {
    private InterProcessMutex lock;
    private int tickets=10;

    public Ticket12306() {
        RetryPolicy retryPolicy = new ExponentialBackoffRetry(3000, 10);
        CuratorFramework client = CuratorFrameworkFactory.builder().connectString("192.168.22.129:2181")
            .sessionTimeoutMs(68 * 1000)
            .connectionTimeoutMs(15 * 1000)
            .retryPolicy(retryPolicy).namespace("sdk").build();
        client.start();
        try {
            System.out.println("/test: "+new String(client.getData().forPath("/locks")));
        } catch (Exception e) {
            e.printStackTrace();
        }
        lock=new InterProcessMutex(client,"/locks");
    }
}

```

```

}

@Override
public void run() {
    while (true){
        try{
            boolean flag=lock.acquire(60, TimeUnit.SECONDS);
            if (tickets>0){
                System.out.println(Thread.currentThread()+" tickets:"+tickets);
                Thread.sleep(100);
                tickets--;
            }else if(tickets==0){
                System.out.println(Thread.currentThread()+" tickets soled out");
                break;
            }
        }catch (Exception ex){
            ex.printStackTrace();
       }finally {
            try {
                lock.release();
                System.out.println(Thread.currentThread()+" lock release");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

LockTest >>

```

public class LockTest {
    public static void main(String[] args){
        Ticket12306 ticket12306=new Ticket12306();
        Thread t1=new Thread(ticket12306,"one");
        Thread t2=new Thread(ticket12306,"two");
        t1.start();
        t2.start();
    }
}

```

Data Utils

lombok

依赖: **lombok** = org.projectlombok

兼容: tomcat7-maven-plugin 版本冲突

打开 lombok 最终生成代码的文件: 在文件夹 target/./classname.class 中找到生成的类, 并在 IntelliJ IDEA 中打开它。

Configuration

Add Dependencies

Include the Lombok dependency in your pom.xml file:

<dependency>

```
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<version>1.18.26</version> <!-- Use the latest version -->
<scope>provided</scope>
</dependency>
```

For Gradle

```
dependencies {
    implementation 'org.projectlombok:lombok:1.18.26' // Use the latest version
    annotationProcessor 'org.projectlombok:lombok:1.18.26'
}
```

[lombok]

Getter

```
package lombok;
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Getter
```

The `@Getter` annotation is applied to a field or a class, and it automatically generates a getter method for each field in the class.

The generated method will have the typical naming convention, like `getFieldName()`. If the field is a boolean, it may generate a `isFieldName()` method.

The getter and setter methods generated by Lombok using the `@Getter` and `@Setter` annotations are public by default.

```
lombok.AccessLevel value() default lombok.AccessLevel.PUBLIC;
```

If you need the generated getter or setter to have a different access modifier (e.g., private, protected, or package-private),

you can specify the access level using the `AccessLevel` parameter.

Setter

```
package lombok;
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Setter
```

The `@Setter` annotation works similarly to `@Getter`, but it generates a setter method for the field.

The generated method will have the typical naming convention like `setFieldName(value)`.

```
lombok.AccessLevel value() default lombok.AccessLevel.PUBLIC;
```

SneakyThrows

```
package lombok;
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.SOURCE)
public @interface SneakyThrows
```

Convert other exceptions to `RuntimeException`

```
Class<? extends Throwable>[] value() default java.lang.Throwable.class;
```

AllArgsConstructor

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface AllArgsConstructor
```

Full parameter constructor, containing all parameters except for final type,

but it does not include a call to a superclass's constructor with arguments.

The `@AllArgsConstructor` annotation in Lombok generates a constructor with parameters for all fields in the class, including transient fields.

```
String staticName() default "";
AllArgsConstructor.AnyAnnotation[] onConstructor() default {};
AccessLevel access() default AccessLevel.PUBLIC;
```

experimental

Accessors

```
package lombok.experimental;
@Target({ElementType.TYPE, ElementType.FIELD})
@Retention(RetentionPolicy.SOURCE)
public @interface Accessors
```

Customize the behavior of `getter` and `setter` methods in a class.

It allows you to configure how getters and setters are generated, providing flexibility in naming conventions and method chaining.

Fluent Accessors

You can make getters and setters `return this` (the current object) instead of void, enabling method chaining.

Prefix Handling

You can define a prefix that will be stripped from the field names when generating the getter and setter methods.

No Argument Required

You can specify that certain fields should not have setters generated.

```
boolean chain() default false;
```

If set to true, setters return this, allowing method chaining.

```
boolean fluent() default false;
```

If set to true, the generated getter and setter methods will not use the conventional get and set prefixes. Instead, they will use the field name directly.

For example, `getName()` becomes `name()`, and `setName()` becomes `name(value)`

```
String[] prefix() default {};
```

Defines a prefix that will be stripped from field names when generating getter and setter methods.

For example, if fields are prefixed with `m_`, using `prefix = "m_"` will generate getter and setter methods without the prefix.

SuperBuilder

```
package lombok.experimental;
@Target(TYPE)
@Retention(SOURCE)
public @interface SuperBuilder
    Add @SuperBuilder to both the base class and the subclass.
    No need to manually write the constructor
```

```
String builderMethodName() default "builder";
String buildMethodName() default "build";
boolean toBuilder() default false;
```

```
String setterPrefix() default "";
```

Other

@Data 添加 get、set 方法、equals、canEquals、hashCode、toString 方法 【TYPE】
@Getter、@Setter、@ToString、@EqualsAndHashCode、@RequiredArgsConstructor

@Getter 添加 get 方法 【TYPE】
lazy=true 将在第一次调用这个 getter 时计算一个值，并从那时起用一个类中字段 缓存它。如果计算数值需要大量的 CPU，或者数值需要大量的内存，这就很有用

@Setter 添加 set 方法 【TYPE】

@ToString 添加 toString 方法，当是枚举时，不会返回 name()，而是所有 field 值的打印 【TYPE】
callSuper=true 使用父类继承属性

@EqualsAndHashCode(callSuper=true) 生成 equals(Object other) 和 hashCode()方法，避免
callSuper=true 用自己的属性和从父类继承的属性 来生成 hashCode

@NoArgsConstructor 无参构造器 【TYPE】
access=AccessLevel.PRIVATE 限制为 private

@RequiredArgsConstructor 为 final 或者@NonNull 字段添加构造函数，在 bean 类中可以省略@Autowired，自动在构造中赋值
staticName="of" 除了之前添加的构造函数，再添加一个同样参数的静态工厂函数 of
access=AccessLevel.PRIVATE 构造函数访问权限

@SneakyThrows 将其它异常转换为 RuntimeException 【CONSTRUCTOR, METHOD】

@UtilityClass 工具类标记。lombok 会自动生成一个抛出异常的私有构造函数，将你添加的任何显式构造函数标记为错误，并将该类标记为 final。

```
package lombok.experimental;
```

@Accessors(chain = true) 访问控制
chain=true setter 返回 this
fluent=true 更简洁的 getter 函数 getToken() => authToken()，适用于属性设置器的非 final 字段，也允许链式调用

@NonNull 字段不为空，添加了@RequiredArgsConstructor 注释后会添加非空检测，如果字段是非 final 的，并且我们为它们添加了@Setter，也会发生这种情况。【FIELD, METHOD, PARAMETER, LOCAL_VARIABLE, TYPE_USE】

```
package lombok.extern.slf4j;
```

@Slfj 添加打印函数，log.info('xxxx') 【TYPE】

@Builder 添加数据构建器 //
PageResult.<TUserEntity>builder().data(list).code(0).count(pageEntity.getTotal()).build();

@Cleanup 确保在代码执行后，在退出当前作用域之前自动清除给定资源 // @Cleanup InputStream is = this.getClass().getResourceAsStream("res.txt"); 【LOCAL_VARIABLE】
value="dispose" 自定义释放资源方法名

@Synchronized 比原生 synchronized 更安全的锁（原生 其他客户端代码最终也可以在我们的实例上同步，这可能会导致意外的死锁）

@Delegate 有效成分继承 //
types= {HasContactInformation.class} 使用模型类 实现 适配器接口内容 // @Delegate(types = {HasContactInformation.class}) private final ContactInformationSupport contactInformation = new ContactInformationSupport();

依赖: fastjson = com.alibaba

常用版本: 1.2.75

源码解析 fastjson1.2.58

```
package com.alibaba.fastjson;
public abstract class JSON implements JSONStreamAware, JSONAware
public static String toJSONString(Object object, SerializerFeature... features)
public static String toJSONString(Object object, boolean prettyFormat)          对象转 j 串 (true 设置为
带格式)

package com.alibaba.fastjson.serializer;
public enum SerializerFeature {    序列化特征
    QuoteFieldNames,           // 输出 key 时是否使用双引号,默认为 true
    UseSingleQuotes,           // 使用单引号而不是双引号,默认为 false
    WriteMapNullValue,         // 是否输出值为 null 的字段, 默认为 false
    WriteEnumUsingToString,
    WriteEnumUsingName,
    UseISO8601DateFormat,     // Date 使用 ISO8601 格式输出, 默认为 false
    WriteNullListAsEmpty,       // List 字段如果为 null,输出为[],而非 null
    WriteNullStringAsEmpty,     // 字符类型字段如果为 null,输出为" ",而非 null
    WriteNullNumberAsZero,      // 数值字段如果为 null,输出为 0,而非 null
    WriteNullBooleanAsFalse,    // Boolean 字段如果为 null,输出为 false,而非 null
    SkipTransientField,        // 默认 true, 类中的 Get 方法对应的 Field 是 transient, 序列化时将不会被忽
略。
    SortField,                 // 按字段名称排序后输出。默认为 false
/** @deprecated */
@Deprecated
    WriteTabAsSpecial,         // 把\t 做转义输出, 默认为 false
    PrettyFormat,               // 结果是否格式化,默认为 false
    WriteClassName,              // 序列化时写入类型信息, 默认为 false。反序列化是需用到
    DisableCircularReferenceDetect,
    WriteSlashAsSpecial,        // 对斜杠'/' 进行转义
    BrowserCompatible,           // 将中文都会序列化为\uXXXX 格式, 字节数会多一些, 但是能兼容 IE 6, 默认
认为 false
    WriteDateUseDateFormat,      // 全局修改日期格式,默认为 false。 JSON.DEFAULT_DATE_FORMAT =
"yyyy-MM-dd" ; JSON.toJSONString(obj, SerializerFeature.WriteDateUseDateFormat);
    NotWriteRootClassName,
/** @deprecated */
    DisableCheckSpecialChar,
    BeanToArray,                // 将对象转为 array 输出
    WriteNonStringKeyAsString,
    NotWriteDefaultValue,
    BrowserSecure,
    IgnoreNonFieldGetter,
    WriteNonStringValueAsString,
    IgnoreErrorGetter,
    WriteBigDecimalAsPlain,
```

2.0.1

2.0.0

1.2.80

1.2.79

1.2.78

1.2.77

1.2.76

1.2.75

1.2.74

1.2.73

1.2.72

1.2.71

1.2.70

1.2.69_sec12

1.2.69_sec11

1.2.69

1.2.68.sec10

1.2.68

1.2.67.sec10

1.2.67

1.2.66

1.2.62

1.2.61.sec10

1.2.61

1.2.60.sec10

1.2.60.sec09

1.2.60

1.2.59

1.2.58.sec10

1.2.58.sec09

1.2.58.sec06

1.2.58

1.2.57.sec10

```
MapSortField;
```

```
}
```

JSON

```
JSON.parse( "[1,2,3]" )          j串转换成 fast 数组或对象 【JSONObject 或 JSONArray 】      //  
JSONObject obj=(JSONObject) JSON.parse(str);  
JSON.parseObject( {"a":1, 'b':2} )      j串转换成 fast 对象 (内部自动转换数组对象为 JSONObject 或 JSONArray)  
【JSONObject】  
JSON.parseObject("{'a':1, 'b':2}", HashMap.class );    j串 转换成 指定类型对象           // HashMap<String, Object> target  
=JSON.parseObject(obj,HashMap.class);  
JSON.parseArray ( "[1,2,3]" )        j串转换成 fast 数组 (内部自动转换数组对象为 JSONObject 或 JSONArray)  
【JSONArray】  
JSON.parseArray("[1,2,3]", ArrayList.class );   j 串 转换成 指定类型数组           // ArrayList <String > target  
=JSON.parseObject(obj,HashMap.class);  
JSON.toJson( Obiect obj )          对象转 fast 数组或对象           【JSONObject 或 JSONArray 】  
BeanUtils.copyProperties( JSON.toJavaObject( (JSONObject)obj, ContainerDetailModel.class), containerDetailEntity);  
jsonObject.putAll(jsonObject); 合并两个 jsonObject
```

```
SerializeConfig serializeConfig=new SerializeConfig();  
aserializeConfig.propertyNamingStrategy= PropertyNamingStrategy.SnakeCase; 转下划线  
String json = JSON.toJSONString(object, serializeConfig);
```

JSONArray

```
JSONArray jsonArray=new JSONArray(); 相当于 List<Object> 【类型可不同】  
getString( 3 );       获取索引 3 的值，自动转换为 String (严格转换)  
getJSONObject( 3 );  获取索引 3 的值，并自动转换为 JSONObject  
getJSONArray( 3 );   自动转换值为 JSONArray  
getObject( 3, HashMap.class);  获取指定类型的值
```

JSONObject

```
JSONObject jsonObject=new JSONObject();  相当于 Map<String, Object>  
element("data", data)    添加或替换一对键值 (val 为 null 时会移除 key, 返回替换前的 JSONObject)  
getString( "key" );     自动转换值为 String (严格转换)  
getJSONObject( "key" ); 自动转换值为 JSONObject  
getJSONArray( "key" ); 自动转换值为 JSONArray  
getObject( "key", HashMap.class);  获取指定类型的值
```

jackson

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->  
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-databind</artifactId>  
  <version>2.15.3</version>  
</dependency>
```

[jackson-databind]

ObjectMapper

```
package com.fasterxml.jackson.databind;  
public class ObjectMapper extends ObjectCodec implements Versioned, Serializable
```

```
// objectMapper.readValue(s, Map.class)
```

枚举直接反序列化时，**jackson 不支持**，可以使用@JsonCreator 为枚举添加支持

内部字段：

枚举	序列化 name() 获取枚举名称，	反序列化 ColorEnum.fromValue()
java.util.Date	序列化 getTime() 获取时间戳，	反序列化 public Date(long date)
java.sql.Timestamp	序列化 getTime() 获取时间戳，	反序列化 public Timestamp(long time)
java.sql.Date	序列化 getTime() 获取时间戳，	反序列化 public Date(long date)
java.sql.Time	序列化 toString() 获取字符串， s)	反序列化 public static Time valueOf(String
java.time.Instant	序列化内置 InstantSerializer	序列化内置 InstantDeserializer
// 1679404870.066608900		
java.time.LocalDateTime	序列化内置 LocalDateTimeSerializer	序列化内置 LocalDateTimeDeserializer
// [2023,3,21,21,9,25,700857200]		
java.time.LocalDate	序列化内置 LocalDateSerializer	序列化内置 LocalDateDeserializer
// [2023,3,21]		
java.time.LocalTime	序列化内置 LocalTimeSerializer	序列化内置 LocalTimeDeserializer
// [21,20,18,880852300]		

public ObjectMapper [configure](#)(Feature f, boolean state)

设置解析策略，如果 json 中有新增的字段并

且是实体类中不存在的，不报错

SerializationFeature.FAIL_ON_EMPTY_BEANS	如果是空对象的时候是否报错，默认 false 不报错
SerializationFeature.WRITE_DATES_AS_TIMESTAMPS	修改序列化后日期格式 (# setDateFormat)
SerializationFeature.INDENT_OUTPUT	格式化输出 j 串
SerializationFeature.WRITE_CHAR_ARRAYS_AS_JSON_ARRAYS	char[] 数组序列化为 String 类型，默认 false 关闭
DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES	反序列化时，遇到未知字段是否报错， 默认 false 关闭
DeserializationFeature.ACCEPT_EMPTY_STRING_AS_NULL_OBJECT	当 JSON 字段为 ""(EMPTY_STRING) 时，解析为普通的 POJO 对象抛出异常。开启后，该 POJO 的属性值为 null， 默认 false 关闭
MapperFeature.PROPAGATE_TRANSIENT_MARKER	忽略 transient 修饰的属性， 默认 false 关闭
JsonGenerator.Feature.WRITE_NUMBERS_AS_STRINGS	序列化 Number 类型及子类为 {"amount1": 1.1}。开启后，
序列化为 String 类型，即 {"amount1": "1.1"}， 默认 false 关闭	
JsonGenerator.Feature.WRITE_BIGDECIMAL_AS_PLAIN	使用 BigDecimal.toString() 序列化。开启后，使用
BigDecimal.toPlainString 序列化，不输出科学计数法的值， 默认 false 关闭	
SerializationFeature.WRITE_ENUMS_USING_TO_STRING	枚举类型序列化方式， 默认情况下使用 Enum.name()。开
启后，使用 Enum.toString()。注：需重写 Enum 的 toString 方法， 默认 false 关闭	

public ObjectMapper [setSerializationInclusion](#)(JsonInclude.Include incl) 序列化时，排除一些指定类型的字段

JsonInclude.Include.NON_DEFAULT 属性为默认值不序列化

JsonInclude.Include.ALWAYS 所有属性

JsonInclude.Include.NON_EMPTY 属性为空 ("") 或者为 NULL 都不序列化

JsonInclude.Include.NON_NULL 属性为 NULL 不序列化

public ObjectMapper [enable](#)(MapperFeature... f) 开启指定配置 //

enable(SerializationFeature.INDENT_OUTPUT);

public ObjectMapper [setPropertyNamingStrategy](#)(PropertyNamingStrategy s) 设置序列化时，字段转换策略

PropertyNamingStrategy.SNAKE_CASE 下划线

PropertyNamingStrategy

```

public ObjectMapper setVisibility(PropertyAccessor forMethod, Visibility visibility)
public ObjectMapper setDateFormat(DateFormat dateFormat)          设置序列化日期格式
public ObjectMapper enableDefaultTyping(ObjectMapper.DefaultTyping dti)
public ObjectMapper registerModule(Module module)      处理不同的时区偏移格式，需要禁用
SerializationFeature.WRITE_DATES_AS_TIMESTAMPS

    JavaTimeModule javaTimeModule = new JavaTimeModule();
    javaTimeModule.addSerializer(LocalDateTime.class, new
        LocalDateTimeSerializer(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));
    javaTimeModule.addDeserializer(LocalDateTime.class, new
        LocalDateTimeDeserializer(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));
public void writeValue(JsonGenerator g, Object value)           写到文件中      // writeValue(new
File("D:/test.txt"), xxuser);
    throws IOException, JsonGenerationException, JsonMappingException
public String writeValueAsString(Object value) throws JsonProcessingException
public byte[] writeValueAsBytes(Object value) throws JsonProcessingException      json 转 j 串， (json 为对象或集合)
                                                                json 转 byte 数组 (json 为对象或集合)

public JsonNode readTree(String content)                      j 串转 json
public <T extends JsonNode> T valueToTree(Object fromValue) throws IllegalArgumentException 对象转 JsonNode
public <T> T readValue(String content, Class<T> valueType)           j 串转 json, json 为对象或集合
    throws IOException, JsonParseException, JsonMappingException

public <T> T readValue(byte[] src, Class<T> valueType)           byte 数组转为对象 // readValue(byteArr, XwjUser.class)
    throws IOException, JsonParseException, JsonMappingException
public <T> T convertValue(Object fromValue, Class<T> toValueType) 对象 转换成指定类型
    fromValue 为字符串时，必须添加转换 J 串 的 构造函数：
        public Song(String json) throws JsonProcessingException
            Song song = new ObjectMapper().readValue(json, Song.class);
            this.mid = song.getMid();
            this.musicName = song.getMusicName();
            this.musicSinger = song.getMusicSinger();
            this.musicLyric = song.getMusicLyric();
            this.musicType = song.getMusicType();
        }
annotation

```

JsonIgnoreProperties

```

package com.fasterxml.jackson.annotation;
@Target({ElementType.ANNOTATION_TYPE, ElementType.TYPE,
    ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@JacksonAnnotation
public @interface JsonIgnoreProperties
    ignore specific properties during JSON serialization and deserialization. It can be applied at the class level or object level.
    public boolean ignoreUnknown() default false;

```

```

public boolean allowGetters() default false;
public boolean allowSetters() default false;

Usage
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.databind.ObjectMapper;

// Ignore unknown properties
@JsonIgnoreProperties(ignoreUnknown = true)
class User {
    private String name;
    private int age;

    public User() {} // Default constructor for deserialization

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}

public class Main {
    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        // JSON contains an extra field "address", which is ignored
        String json = "{\"name\":\"Alice\", \"age\":25, \"address\":\"NY\"}";

        User user = mapper.readValue(json, User.class);
        System.out.println("Name: " + user.getName() + ", Age: " + user.getAge());
    }
}

```

JsonSerialize

```

package com.fasterxml.jackson.databind.annotation;

@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.TYPE,
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@com.fasterxml.jackson.annotation.JacksonAnnotation
public @interface JsonSerialize

```

Customize the serialization process of a Java object into JSON.

It allows you to specify a custom serializer for a field, method, or entire class.

```
public Class<? extends JsonSerializer> using() default JsonSerializer.None.class;
```

Specify a custom serializer for a field, method, or entire class.

```
using = LocalDateTimeSerializer.class
```

Usage1

```

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.databind.annotation.JsonSerialize;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import java.io.IOException;

// Custom Serializer for the whole class
class CustomExampleSerializer extends JsonSerializer<Example> {
    @Override
    public void serialize(Example value, JsonGenerator gen, SerializerProvider serializers) throws
IOException {
        gen.writeStartObject();

```

```

        gen.writeStringField("customName", "Serialized: " + value.getName());
        gen.writeEndObject();
    }
}

// Custom Deserializer for the whole class
class CustomExampleDeserializer extends JsonDeserializer<Example> {
    @Override
    public Example deserialize(JsonParser p, DeserializationContext ctxt) throws IOException {
        JsonNode node = p.getCodec().readTree(p);
        String name = node.get("customName").asText().replace("Serialized: ", "");
        return new Example(name);
    }
}

```

```

// Annotate the class with @JsonSerialize and @JsonDeserialize
@JsonSerialize(using = CustomExampleSerializer.class)
@JsonDeserialize(using = CustomExampleDeserializer.class)
class Example {
    private String name;

    public Example() {} // Default constructor for deserialization

    public Example(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

Usage2

```

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.databind.JsonSerializer;
import com.fasterxml.jackson.databind.SerializerProvider;
import com.fasterxml.jackson.databind.annotation.JsonSerialize;
import java.io.IOException;

// Custom serializer
class CustomDateSerializer extends JsonSerializer<String> {
    @Override
    public void serialize(String value, JsonGenerator gen, SerializerProvider serializers) throws IOException {
        gen.writeString("Serialized: " + value);
    }
}

```

```

// Model class using @JsonSerialize
class Example {
    @JsonSerialize(using = CustomDateSerializer.class)
    private String name;

    public Example(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

JsonDeserialize

```

package com.fasterxml.jackson.databind.annotation;

@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.TYPE,
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)

```

```

@com.fasterxml.jackson.annotation.JacksonAnnotation
public @interface JsonDeserialize
    Customize the deserialization process of a Java object into JSON.

public Class<? extends JsonDeserializer> using() default JsonDeserializer.None.class;
    Specify a custom deserializer for a field, method, or class in a Java object.
        using = LocalDateTimeDeserializer.class

Usage
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonDeserializer;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import java.io.IOException;

// Custom deserializer
class CustomDateDeserializer extends JsonDeserializer<String> {
    @Override
    public String deserialize(JsonParser p, DeserializationContext ctxt) throws IOException {
        return "Deserialized: " + p.getText();
    }
}

// Model class using @JsonDeserialize
class Example {
    @JsonDeserialize(using = CustomDateDeserializer.class)
    private String name;

    public String getName() {
        return name;
    }
}

```

[jackson-annotations]

Customize the mapping between JSON property names and Java object fields or methods.
It is useful when the JSON property names differ from the Java field names or when you need to control the serialization/deserialization behavior.

JsonProperty

```

package com.fasterxml.jackson.annotation;
@Target({ElementType.ANNOTATION_TYPE, ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@JacksonAnnotation
public @interface JsonProperty

```

Customize the mapping between **JSON property names** and **Java object fields or methods**.

It is useful when the JSON property names differ from the Java field names or when you need to control the serialization/deserialization behavior.

```
String value() default USE_DEFAULT_NAME;
```

```
Access access() default Access.AUTO;
```

```
public enum Access {
```

`AUTO`,

`READ_ONLY`,

The property will only be serialized (written to JSON). It will not be deserialized (i.e., ignored when reading from JSON).

`WRITE_ONLY`,

The property will only be deserialized (read from JSON). It will not be included in the serialized output (i.e., ignored when writing JSON).

READ_WRITE Default option. The property will be used both for reading from JSON (deserialization) and writing to JSON (serialization).

```
;  
}
```

Specify Field Visibility

You can annotate fields directly, making the mapping behavior more explicit if you don't want to rely on getters or setters.

```
public class Car {  
    @JsonProperty("car_make")  
    private String make;  
  
    @JsonProperty("car_model")  
    private String model;  
  
    // Getters and setters  
}
```

JsonPropertyOrder

```
package com.fasterxml.jackson.annotation;  
@Target({ElementType.ANNOTATION_TYPE, ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR,  
ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
@JacksonAnnotation  
public @interface JsonPropertyOrder
```

Define the order in which properties of a class are serialized to JSON.

It controls the sequence in which the fields are written in the JSON output, which is particularly useful when the order matters for readability or compatibility with specific APIs.

Usage

In this example, the Person class has three fields (name, age, and email), and @JsonPropertyOrder ensures that when the object is serialized, the fields appear in the specified order.

```
import com.fasterxml.jackson.annotation.JsonProperty;  
import com.fasterxml.jackson.annotation.JsonPropertyOrder;  
  
@JsonPropertyOrder({"name", "age", "email"})  
public class Person {  
    @JsonProperty("name")  
    private String name;  
  
    @JsonProperty("age")  
    private int age;  
  
    @JsonProperty("email")  
    private String email;  
  
    // Getters and Setters  
}
```

JsonAlias

```
package com.fasterxml.jackson.annotation;  
@Target({ElementType.ANNOTATION_TYPE, // for combo-annotations  
ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER// for properties (field, setter, ctor param)  
})  
@Retention(RetentionPolicy.RUNTIME)
```

```
@JacksonAnnotation  
public @interface JsonAlias
```

Define alternative names (aliases) for a property when deserializing JSON.

This is useful when your application needs to handle multiple JSON formats, or when the property name has changed over time but you still want to support legacy field names in the JSON input.

Usage

In this example, the username field can be deserialized from JSON fields named "username", "user_name", or "userId".

```
import com.fasterxml.jackson.annotation.JsonAlias;  
import com.fasterxml.jackson.annotation.JsonProperty;  
  
public class User {  
    @JsonProperty("username")  
    @JsonAlias({"user_name", "userId"})  
    private String username;  
  
    // Getters and Setters  
}
```

JsonFormat

```
package com.fasterxml.jackson.annotation;  
@Target({ElementType.ANNOTATION_TYPE, ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,  
ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@JacksonAnnotation  
public @interface JsonFormat
```

Control how data is serialized (written to JSON) and deserialized (read from JSON), particularly for date and time formats.

This annotation is commonly applied to fields or methods to specify custom formatting rules, such as the pattern for formatting `LocalDateTime`, `Date`, or other date-time objects, or to control how enums are serialized.

Difference between `@DateTimeFormat` and `@JsonFormat`

`@DateTimeFormat`

Specify the format of the front-end input time type. Otherwise, an exception is thrown and the format will not be converted.

Formats date/time values for request parameters, form inputs, and model attributes in a web environment.

`@DateTimeFormat` cannot be used directly for formatting or parsing fields in an `@RequestBody`.

`@JsonFormat`

Specify the time type parameter format passed in by the front end, or specify the time type format of the back end's response to the front end;

Formats date/time values for JSON serialization and deserialization when returning responses or receiving JSON payloads in REST services.

```
public String pattern() default "";  
public Shape shape() default Shape.ANY;  
public String locale() default DEFAULT_LOCALE;
```

Specify a locale (language and region) for formatting dates, times, or numbers during serialization and deserialization.

`locale = "fr_FR"`

Specifies the locale as French (France). In this case, the days and months will be formatted in French.

So, for a date like 2024-09-20, it might output as "vendredi, 20 septembre 2024" (Friday, 20 September 2024).

```

public String timezone() default DEFAULT_TIMEZONE;

public enum Shape
{
    ANY,
    NATURAL,
    SCALAR,
    ARRAY,
    OBJECT,
    NUMBER,
    NUMBER_FLOAT,
    NUMBER_INT,
    STRING,
    BOOLEAN,
    BINARY
;
}

```

Date and Time Formatting

`@JsonFormat` is often used to define the format for date and time fields.

You can specify the desired format using the `pattern` attribute, which follows the `java.time` or `SimpleDateFormat` conventions. Here, the `eventDate` field will be serialized and deserialized in the "yyyy-MM-dd HH:mm:ss" format.

```

import com.fasterxml.jackson.annotation.JsonFormat;
import java.time.LocalDateTime;

public class Event {
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss", timezone = "UTC")
    private LocalDateTime eventDate;

    // Getters and setters
}

```

Enum Formatting

You can also use `@JsonFormat` to control how enums are serialized.

By default, Jackson serializes enums by their name, but `@JsonFormat` can specify whether to serialize them by index or as a `string`.

Here, the `status` field will be serialized as a number (index of the enum).

```

public enum Status {
    ACTIVE, INACTIVE
}

public class User {
    @JsonFormat(shape = JsonFormat.Shape.NUMBER)
    private Status status;

    // Getters and setters
}

```

Using the Enum in Your Model

You can then use the enum in your model class like this:

```
import com.fasterxml.jackson.annotation.JsonFormat;
```

```

public class Person {
    private String name;

    @JsonFormat(shape = JsonFormat.Shape.STRING)
    private Gender gender;

    // Getters and Setters
}

```

Using `@JsonFormat` to Specify a Pattern

If you want to control the output format of the enum values:

```
import com.fasterxml.jackson.annotation.JsonFormat;
```

```

@JsonFormat(shape = JsonFormat.Shape.STRING)
public enum Gender {
    MALE, FEMALE, OTHER;
}

```

Customizing Enum Serialization

```
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonValue;
```

```

public enum Gender {
    MALE("M"),
    FEMALE("F"),
    OTHER("O");

    private final String code;

    Gender(String code) {
        this.code = code;
    }

    @JsonValue
    public String getCode() {
        return code;
    }

    @JsonCreator
    public static Gender fromCode(String code) {
        for (Gender gender : Gender.values()) {
            if (gender.getCode().equals(code)) {
                return gender;
            }
        }
        throw new IllegalArgumentException("Invalid code: " + code);
    }
}

```

JsonAutoDetect

```
package com.fasterxml.jackson.annotation;

@Target({ElementType.ANNOTATION_TYPE, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@JacksonAnnotation
public @interface JsonAutoDetect
```

Controls the visibility of properties during the process of serialization and deserialization.

It allows you to specify which fields or methods (getters, setters, etc.) should be visible for Jackson to include in JSON operations.

This is useful when you want more control over how your object's fields and methods are exposed to JSON.

Visibility `fieldVisibility()` default `Visibility.DEFAULT`;

Controls the visibility of class fields.

```

Visibility getterVisibility() default Visibility.DEFAULT;
    Controls the visibility of getter methods.
Visibility isGetterVisibility() default Visibility.DEFAULT;
    Controls the visibility of is getter methods (for boolean fields).
Visibility setterVisibility() default Visibility.DEFAULT;
    Controls the visibility of setter methods.
Visibility creatorVisibility() default Visibility.DEFAULT;
    Controls the visibility of constructors or factory methods.

```

public enum Visibility {	
ANY,	All fields/methods are visible.
NON_PRIVATE,	Only non-private fields/methods are visible.
PROTECTED_AND_PUBLIC,	Protected and public fields/methods are visible.
PUBLIC_ONLY,	Only public fields/methods are visible.
NONE,	No fields/methods are visible.
DEFAULT;	Uses the default visibility rules.

Fields

 ANY (all fields are visible, regardless of access modifiers like private, protected, or public)

Getters (standard)

 PUBLIC_ONLY (only public getter methods are visible)

Is-getters (isX() methods for boolean properties)

 PUBLIC_ONLY

Setters

 PUBLIC_ONLY

Creators (like constructors)

 PUBLIC_ONLY

Usage

By default, Jackson will only serialize public fields or fields with getters.

If you want Jackson to serialize private fields without getters, you can use `@JsonAutoDetect`.

```

import com.fasterxml.jackson.annotation.JsonAutoDetect;

@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.ANY)
public class Person {
    private String name;
    private int age;

    // No getters or setters
}

```

Other

@JsonInclude	返回指定格式的 json 数据, 如果属性返回值为空, 则不返回任何内容
value = <code>JsonInclude.Include.NON_NULL</code>	前端获取的数据为 <code>other_name</code> , 后端获取为当前的成员字段名 <code>createAt</code>
<code>JsonInclude.Include.NON_DEFAULT</code>	属性为默认值不序列化
<code>JsonInclude.Include.ALWAYS</code>	所有属性
<code>JsonInclude.Include.NON_EMPTY</code>	属性为空 ("") 或者为 NULL 都不序列化
<code>JsonInclude.Include.NON_NULL</code>	属性为 NULL 不序列化

@JsonIgnoreProperties 选择性忽略类中的属性，通常作用于类上。

```
value={"name", "sex"}
```

@JsonIgnore 序列化时，忽略一个字段

@JsonBackReference 解决对象中存在双向引用导致的无限递归 (infinite recursion) 问题。

@JsonManagedReference

@JsonCreator 只能用在静态方法或者构造方法，指定 对象反序列化时的构造函数 或者 工厂方法，必须返回一个当前类对象（默认反序列化使用 无参构造函数 和 set 方法）

```
@JsonCreator  
public static ConfigModel create(@JsonProperty("id") long id, @JsonProperty("name") String name, String transientField ) {  
    // 必须指定@JsonProperty，那些字段需要映射  
    ConfigModel model = new ConfigModel();  
    model.setId(id);  
    model.setName(name);  
    System.out.println("factory");  
    return model;  
}
```

@JsonValue 只指定类中的一个方法为序列化方法，指定序列化后的字符串，一个类只能用一个，可以标注在 set 方法上

@JsonView 指定 Controller 响应那些数据

UserController >>

```
@RestController
```

```
class UserController {
```

```
    @GetMapping("/user/internal")  
    @JsonView(User.Views.Internal.class)  
    User getPublicUser() {  
        return new User("internal", "external", "john");  
    }
```

```
    @GetMapping("/user/public")  
    @JsonView(User.Views.Public.class)  
    User getPrivateUser() {
```

```
        return new User("internal", "external", "john");  
    }
```

```
}
```

User >>

```
class User {
```

```
    User(String internalId, String externalId, String name) {  
        this.internalId = internalId;  
        this.externalId = externalId;  
        this.name = name;  
    }
```

```
    @JsonView(User.Views.Internal.class)  
    String internalId;  
    @JsonView(User.Views.Public.class)  
    String externalId;  
    @JsonView(User.Views.Public.class)
```

```
String name;
static class Views {
    static class Public {
    }
    static class Internal extends Public {
    }
}
}
```

[jackson-datatype-jsr310]

InstantSerializer

```
package com.fasterxml.jackson.databind.jsr310.ser;
public class InstantSerializer extends InstantSerializerBase<Instant>    为 Instant 提供的序列化器
```

LocalDateTimeSerializer

```
package com.fasterxml.jackson.databind.jsr310.ser;
public class LocalDateTimeSerializer extends JSR310FormattedSerializerBase<LocalDateTime>
    LocalDateTimeSerializer is a predefined serializer in Jackson used to serialize LocalDateTime objects to JSON.
    By default, LocalDateTime does not get serialized in a human-readable format (e.g., 2023-09-20T14:30:00).
    However, with LocalDateTimeSerializer, you can customize how LocalDateTime is serialized.
```

The default formatter used by the LocalDateTimeSerializer in Jackson is `DateTimeFormatter.ISO_LOCAL_DATE_TIME`. This formatter outputs LocalDateTime objects in a format that follows the ISO-8601 standard, which is:

`yyyy-MM-dd'T'HH:mm:ss.SSS`

```
public LocalDateTimeSerializer(DateTimeFormatter f)
```

Specify a Custom Date Format

If you want to use a custom format, you can pass a DateTimeFormatter to the LocalDateTimeSerializer.

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.annotation.JsonSerialize;
import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;
import com.fasterxml.jackson.datatype.jsr310.ser.LocalDateTimeSerializer;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Event {

    @JsonSerialize(using = LocalDateTimeSerializer.class)
    private LocalDateTime eventTime;

    public Event(LocalDateTime eventTime) {
        this.eventTime = eventTime;
    }

    public LocalDateTime getEventTime() {
        return eventTime;
    }

    public void setEventTime(LocalDateTime eventTime) {
        this.eventTime = eventTime;
    }

    public static void main(String[] args) throws Exception {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    }
}
```

```

LocalDateTimeSerializer serializer = new LocalDateTimeSerializer(formatter);

ObjectMapper mapper = new ObjectMapper();
JavaTimeModule module = new JavaTimeModule();
module.addSerializer(LocalDateTime.class, serializer);
mapper.registerModule(module);

Event event = new Event(LocalDateTime.now());
String json = mapper.writeValueAsString(event);
System.out.println(json);
}
}

```

LocalDateSerializer

package com.fasterxml.jackson.databind.jsr310.ser;
public class **LocalDateSerializer** extends JSR310FormattedSerializerBase<LocalDate> 为 LocalDate 提供的序列化器

LocalTimeSerializer

package com.fasterxml.jackson.databind.jsr310.ser;
public class **LocalTimeSerializer** extends JSR310FormattedSerializerBase<LocalTime> 为 LocalTime 提供的序列化器
gson

依赖:

```

<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.9</version>
</dependency>

```

[gson]

package com.google.gson;
public final class **Gson** 序列化工具对象
public String **toJson**(Object src) json 转 J 串
public <T> T **fromJson**(String json, Class<T> classOfT) throws JsonSyntaxException J 串 转 json

注解:

@SerializedName 序列化工具
value = "test_date"
alternate = {"testTT"} 候选项

```

GsonBuilder gsonBuilder=new GsonBuilder();
gsonBuilder.setFieldNamingPolicy(FieldNamingPolicy.LOWER_CASE_WITH_UNDERSCORES);
Gson gson=gsonBuilder.create();
String json = gson.toJson(object);

```

javers

源码解析 javers-core3.9.6

@DiffIgnore 忽略比较此字段 【METHOD, FIELD, TYPE】

dozer

可以将一个对象递归拷贝到另外一个对象。既支持简单的对象映射，也支持复杂的对象映射

```
<!-- https://mvnrepository.com/artifact/net.sf.dozer/dozer -->
<dependency>
    <groupId>net.sf.dozer</groupId>
    <artifactId>dozer</artifactId>
    <version>5.5.1</version>
</dependency>
```

Configuration

application.yml

```
dozer:
  mapping-files:
    - classpath:dozer/dozer-config.xml
```

dozer-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://dozer.sourceforge.net
           http://dozer.sourceforge.net/schema/beanmapping.xsd">

<configuration>
    <date-format>MM/dd/yyyy HH:mm</date-format>                                // global date format

    <custom-converters> <!-- these are always bi-directional -->
        <converter type="org.dozer.converters.TestCustomConverter" >
            <class-a>org.dozer.vo.CustomDoubleObject</class-a>
            <class-b>java.lang.Double</class-b>
        </converter>
        <converter type="org.dozer.converters.TestCustomHashMapConverter" > // global
    converter(CustomConverter instances need to be injected into the DozerBeanMapper)
        <class-a>
            org.dozer.vo.TestCustomConverterHashMapObject
        </class-a>
        <class-b>
            org.dozer.vo.TestCustomConverterHashMapPrimeObject
        </class-b>
    </converter>
    </custom-converters>
</configuration>

<mapping map-id="cct" wildcard="false">                                         // default wildcard matching
true(mapping all fields, On the other hand, only specified fields will be mapped.)
                                                // and defines custom map id used by
dozerBeanMapper#map(Object, Object, String)
    <class-a>org.dozer.vo.TestObjectFoo</class-a>
    <class-b>org.dozer.vo.TestObjectFooPrime</class-b>
    <field>
        <a get-method="getOneFoo">oneFoo</a>
        <b>oneFooPrime</b>
    </field>
</mapping>

<mapping>
    <class-a>org.dozer.vo.SimpleObj</class-a>
    <class-b>org.dozer.vo.SimpleObjPrime2</class-b>
    <field custom-converter="org.dozer.converters.StringAppendCustomConverter">      // custom converter
based on dozerBean#setCustomConverterWithId
        <a>field1</a>
        <b>field1Prime</b>
    </field>
</mapping>
<mapping>
    <class-a>org.dozer.vo.SimpleObj</class-a>
```

```

<class-b>org.dozer.vo.SimpleObjPrime2</class-b>
<field custom-converter-id="CustomConverterWithId">          //  custom converter using converter id
  <a>field1</a>
  <b>field1Prime</b>
</field>
</mapping>

<mapping>
  <class-a>org.dozer.vo.TestObject</class-a>
  <class-b>org.dozer.vo.TestObjectPrime</class-b>
  <field>
    <a date-format="MM/dd/yyyy HH:mm:ss:SS">dateString</a> // date type mapping
    <b>dateObject</b>
  </field>
</mapping>

<mapping date-format="MM-dd-yyyy HH:mm:ss">
  <class-a>org.dozer.vo.TestObject</class-a>
  <class-b>org.dozer.vo.TestObjectPrime</class-b>
  <field>
    <a>dateString</a>
    <b>dateObject</b>
  </field>
</mapping>

<mapping>
  <class-a>org.dozer.vo.Individuals</class-a>
  <class-b>org.dozer.vo.FlatIndividual</class-b>           // index mapping
  <field>
    <a>usernames[0]</a>
    <b>username1</b>
  </field>
  <field>
    <a>aliases.otherAliases[0]</a>
    <b>primaryAlias</b>
  </field>
</mapping>

</mappings>

```

TestCustomConverter

```

public class TestCustomConverter implements CustomConverter {

  public Object convert(Object destination, Object source,
    Class destClass, Class sourceClass) {
    if (source == null) {
      return null;
    }
    CustomDoubleObject dest = null;
    if (source instanceof Double) {
      // check to see if the object already exists
      if (destination == null) {
        dest = new CustomDoubleObject();
      } else {
        dest = (CustomDoubleObject) destination;
      }
      dest.setTheDouble(((Double) source).doubleValue());
      return dest;
    } else if (source instanceof CustomDoubleObject) {
      double sourceObj =
        ((CustomDoubleObject) source).getTheDouble();
      return new Double(sourceObj);
    } else {
      throw new MappingException("Converter TestCustomConverter "
        + "used incorrectly. Arguments passed in were:"
        + destination + " and " + source);
    }
  }
}

```

```
}
```

3.12.0

3.11

3.10

3.9

3.8.1

3.8

3.7

3.6

3.5

3.4

3.3.2

3.3.1

3.3

3.2.1

3.2

3.1

3.0.1

3.0

[dozer]

DozerBeanMapper

```
package org.dozer;
public class DozerBeanMapper implements Mapper 对象拷贝工具
public void setMappingFiles(List<String> mappingFileUrls) 设置 mapping 文件,
路径都在 resources 资源文件下 // Arrays.asList("dozer-mapping/formrequest_tintity-mapping.xml")
public void map(Object source, Object destination, String mapId) throws MappingException
public <T> T map(Object source, Class<T> destinationClass, String mapId) throws MappingException
public <T> T map(Object source, Class<T> destinationClass) throws MappingException 构造新的
destinationClass 实例对象, 通过 source 对象中的字段内容, 映射到 destinationClass 实例对象中, 并返回新的
destinationClass 实例对象。 // dozerBeanMapper.map(obj, Target.class)
```

```
public void setCustomConverters(List<CustomConverter> customConverters) 设置自定义转
换器
public void setCustomConvertersWithId(Map<String, CustomConverter> customConvertersWithId)
```

DozerConverter

```
package org.dozer;
public abstract class DozerConverter<A, B> implements ConfigurableCustomConverter
public abstract B convertTo(A var1, B var2);
public abstract A convertFrom(B var1, A var2);
```

CustomConverter

```
package org.dozer;
public interface CustomConverter
Object convert(Object var1, Object var2, Class<?> var3, Class<?> var4);
```

commons-lang3

依赖: commons-lang3 = org.apache.commons

```
<dependency>
<groupId>org.apache.commons</groupId>
<artifactId>commons-lang3</artifactId>
<version>3.12.0</version>
</dependency>
```

[commons-lang3]

StringUtils

```
package org.apache.commons.lang3;
public class StringUtils 字符串工具
public static <T> String join(T... elements) 将元素拼接成字符串
public static String join(Iterable<?> iterable, String separator) 根据分隔符拼接
public static boolean isNotBlank(CharSequence cs) 返回 false: null, 长度为空, 不是都为空格
public static <T extends CharSequence> T defaultIfBlank(T str, T defaultStr) 默认字符串
public static String[] splitByWholeSeparator(String str, String separator) 分割字符串, 包含 null 多情况处理
public static String repeat(char ch, int repeat) 重复字符, repeat 为负数时, 返回空字符串
```

StringEscapeUtils

```
public class StringEscapeUtils 转义工具  
public static final String unescapeJava(String input) 去掉转义字符
```

Range

```
package org.apache.commons.lang3;  
public final class Range<T> implements Serializable 范围工具
```

ArrayUtils

```
package org.apache.commons.lang3;  
public class ArrayUtils 数组工具  
public static <T> T[] addAll(T[] array1, T... array2) 合并两个原始数组  
public static Object[] nullToEmpty(Object[] array) 针对原始数组的取默认值
```

SerializationUtils

```
package org.apache.commons.lang3;  
public class SerializationUtils  
public static <T extends Serializable> T clone(final T object)  
Deep clone an Object using serialization.
```

commons-collections4

依赖: commons-collections4 = org.apache.commons

```
<dependency>  
    <groupId>org.apache.commons</groupId>  
    <artifactId>commons-collections4</artifactId>  
    <version>4.4</version>  
</dependency>
```

源码解析

```
package org.apache.commons.collections4;  
public class ListUtils list 工具  
  
public static <T> List<T> defaultIfNull(List<T> list, List<T> defaultList)  
集合判断:  
例 1: 判断集合是否为空:  
CollectionUtils.isEmpty(null): true  
CollectionUtils.isEmpty(new ArrayList()): true  
CollectionUtils.isEmpty({a,b}): false  
  
例 2: 判断集合是否不为空:  
CollectionUtils.isNotEmpty(null): false  
CollectionUtils.isNotEmpty(new ArrayList()): false  
CollectionUtils.isNotEmpty({a,b}): true
```

例 3 2 个集合间的操作：

集合 a: {1,2,3,3,4,5}

集合 b: {3,4,4,5,6,7}

CollectionUtils.union(a, b)(并集): {1,2,3,3,4,4,5,6,7}

CollectionUtils.intersection(a, b)(交集): {3,4,5}

CollectionUtils.disjunction(a, b)(交集的补集): {1,2,3,4,6,7}

CollectionUtils.disjunction(b, a)(交集的补集): {1,2,3,4,6,7}

CollectionUtils.subtract(a, b)(A 与 B 的差): {1,2,3}

CollectionUtils.subtract(b, a)(B 与 A 的差): {4,6,7}

commons-beanutils

依赖: commons-beanutils = org.apache.commons

AAA--Spring.docx#BeanUtils

BeanUtils

```
package org.apache.commons.beanutils;
public class BeanUtils
public static void copyProperties(Object dest, Object orig) throws IllegalAccessException, InvocationTargetException    会进
行类型转换
1. 字段名不一致，属性无法拷贝
2. 类型不一致，将会进行默认类型转换（如 Integer 和 String 互转），转换不了的需要自定义转换器——（包装类转基本类型，如
果包装类没有初始化，则会抛异常，因为不能给基本类型赋值 null，因此最好不要在类变量中使用基本类型）
3. 嵌套对象字段，将会与源对象使用同一对象，即使用浅拷贝
```

PropertyUtils

```
package org.apache.commons.beanutils;
public class PropertyUtils
public static void copyProperties(Object dest, Object orig) throws IllegalAccessException, InvocationTargetException,
NoSuchMethodException
PropertyUtils.copyProperties 应用的范围稍为窄一点，它只对名字和类型都一样的属性进行 copy，如果名字一样但类型不一样，它
会报错
```

commons-io

依赖: commons-io = commons-io

commons-io

FileUtils

```
package org.apache.commons.io;
public class FileUtils   文件工具
public static File getTempDirectory()                                Get System Temporary Folder. //
C:\Users\saideake\AppData\Local\Temp
public static void copyFile(File srcFile, File destFile) throws IOException      Copy files, If the write file does not exist,
create it even if the folder does not exist.
copied within the same folder. // ajava.txt => ajava-backup.txt          If only the file name is provided. It will be
public static void copyFileToDirectory(File srcFile, File destDir) throws IOException     Copy files to directory.
// ajava.txt => C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
```

```
public static Collection<File> listFiles(File directory, IOFileFilter fileFilter, IOFileFilter dirFilter)      List sub files.
    // FileUtils.listFiles(new File("D:/eclipse-workspace/ProgramExercise"), EmptyFileFilter.NOT_EMPTY, null)      Filter
    empty files and do not read subfolders.
    // FileUtils.listFiles(new File("D:/eclipse-workspace/ProgramExercise"), EmptyFileFilter.NOT_EMPTY,
    DirectoryFileFilter.INSTANCE)
    // FileUtils.listFiles(new File("D:/eclipse-workspace/ProgramExercise"), new SuffixFileFilter("java"),
    DirectoryFileFilter.INSTANCE)
    // FileUtils.listFiles(
        new File("D:/eclipse-workspace/ProgramExercise"),
        FileFilterUtils.or(new SuffixFileFilter("java"), new SuffixFileFilter("class")),
        DirectoryFileFilter.INSTANCE)
public static void writeStringToFile(File file, String data, Charset charset) throws IOException

public static void writeStringToFile(File file, String data, Charset charset, boolean append) throws IOException
```

IOUtils

```
package org.apache.commons.io;
public class IOUtils
public static int copy(InputStream inputStream, OutputStream outputStream) throws IOException
    output
```

UnsynchronizedByteArrayOutputStream

```
package org.apache.commons.io.output;
public final class UnsynchronizedByteArrayOutputStream extends AbstractByteArrayOutputStream   字节输出流
    filefilter
```

DirectoryFileFilter

```
package org.apache.commons.io.filefilter;
public class DirectoryFileFilter extends AbstractFileFilter implements Serializable
public static final IOFileFilter DIRECTORY = new DirectoryFileFilter();
public static final IOFileFilter INSTANCE;           Read subfolders
```

EmptyFileFilter

```
package org.apache.commons.io.filefilter;
public class EmptyFileFilter extends AbstractFileFilter implements Serializable
```

SuffixFileFilter

```
package org.apache.commons.io.filefilter;
public class SuffixFileFilter extends AbstractFileFilter implements Serializable
public SuffixFileFilter(String suffix)     Construct a SuffixFileFilter.
```

commons-configuration2

依赖: org.apache.commons = commons-configuration2

常用版本: 2.2

commons-configuration

依赖: commons-configuration = commons-configuration

常用版本: 1.10

commons-compress

概念: Apache 开源组织提供的用于压缩解压的工具包

```
package org.apache.commons.compress.utils;
public final class ByteUtils   字节工具
```

```
package org.apache.commons.compress.archivers.zip;
public abstract class ZipUtil 压缩包工具
public static long dosToJavaTime(long dosTime)          Dos 时间转换为 Java 时间
static boolean canHandleEntryData(ZipArchiveEntry entry)  此库是否能够读取或写入给定条目
                                                        commons-codec
```

依赖:

```
<!-- https://mvnrepository.com/artifact/commons-codec/commons-codec -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.15</version>
</dependency>
```

```
jdk.1.8 删除了 sun.misc.BASE64Encoder  sun.misc.BASE64Decoder;
new BASE64Encoder().encode(encrypted); => Base64.encodeBase64String(encrypted);
```

源码解析

Base64

```
package org.apache.commons.codec.binary;
public class Base64 extends BaseNCodec
public static byte[] decodeBase64(String base64String)  解码 base64 字符串
                                                        hutool
```

依赖:

```
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>5.7.22</version>
</dependency>
```

源码解析 hutool-all-5.7.20

```
package cn.hutool.json;
public class JSONUtil      json 工具
public static String toJsonStr(Object obj)    对象变 j 串
```

```
package cn.hutool.json;
public class JSONConfig implements Serializable
```

guava

```
<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>32.1.1-jre</version>
</dependency>
```

包含了若干被 Google 的 Java 项目广泛依赖的核心库： 集合、缓存、原生类型支持、并发库、通用注解，字符串处理，I/O

[guava]

```
package com.google.common.collect;
public final class Lists  列表工具
public static <E> ArrayList<E> newArrayList(E... elements)      创建一个 ArrayList
public static <T> List<List<T>> partition(List<T> list, int size)

package com.google.common.collect;
public final class Collections2 集合工具

package com.google.common.collect;
public abstract class ImmutableList<K, V> implements Map<K, V>, Serializable 不可修改的 map
public static <K, V> ImmutableList<K, V> of( K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5) 初始化一个 map    //
Map<String, Integer> left = ImmutableList.of("a", 1, "b", 2, "c", 3);

package com.google.common.base;
public final class Strings 字符串工具
public static boolean isNullOrEmpty(@Nullable String string) 不为 null 或空字符串 返回真
public static String nullToEmpty(@Nullable String string) 为 null 时返回 空字符串

package com.google.common.base;
public final class Enums 枚举工具
public static <T extends Enum<T>> Optional<T> getIfPresent(Class<T> enumClass, String value) 获取如果存在 // Enums.getIfPresent(AccountTypeEnum.class).orNull()
public abstract <V> Optional<V> transform(Function<? super T, V> function) 对 Optional 进行转换，为空什么都不做 // opt.transform(AccountType::getKk).orNull()

package com.google.common.base;
public final class Preconditions 前提条件工具
public static void checkArgument(boolean expression, @Nullable Object errorMessage) 为假抛出异常
IllegalArgumentException // Preconditions.checkArgument(true, "此运单不可取消!");
public static <T> T checkNotNull(T reference, @Nullable Object errorMessage) 为 null 抛出异常 NullPointerException // Preconditions.checkNotNull(shipping, "运单不存在!");

package com.google.common.base;
public final class Predicates 断言工具
public static <T> Predicate<T> not(Predicate<T> predicate) 不是判断
public static <T> Predicate<T> or(Predicate... components) 或者判断
public static Predicate<Object> instanceOf(Class<?> clazz) 是某个类的实例
public static <T> Predicate<T> and(Predicate<? super T> first, Predicate<? super T> second) 结合连个条件返回新条件

package com.google.common.hash;
public final class BloomFilter<T> implements Predicate<T>, Serializable 布隆过滤器
public static <T> BloomFilter<T> create(   Funnel<? super T> funnel, int expectedInsertions, double fpp ) 预期插入
数，预期 false 概率
```

```
package com.google.common.io;
public final class ByteStreams    字节流

package com.google.common.io;
public final class Files    文件工具
    static byte[] readFile(InputStream in, long expectedSize) throws IOException    读取文件成字节流
```

joda-time

依赖:

```
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.10.6</version>
</dependency>
```

transmittable-thread-local

依赖: `transmittable-thread-local` = com.alibaba

常用版本: 2.12.1

`transmittable-thread-local`: 当线程池有线程复用时, 确保子线程的 `InheritableThreadLocal` 变量永远跟该线程创建时的父线程的 `InheritableThreadLocal` 一致, 弥补 `InheritableThreadLocal` 的不足 (通常应用于方法级监控的中间件中)

[transmittable-thread-local]

```
package com.alibaba.ttl;
public class TransmittableThreadLocal<T> extends InheritableThreadLocal<T> implements TtlCopier<T>

public final T get()
public final void set(T value)
public final void remove()

private void addThisToHolder()
```

Usage

```
package com.simi.sandbox.transmittablethreadlocal;
import com.alibaba.ttl.TransmittableThreadLocal;
import com.alibaba.ttl.TtlRunnable;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadLocalExample {

    private static ThreadLocal<String> inheritableThreadLocal = new InheritableThreadLocal<>();
    private static ThreadLocal<String> transmittableThreadLocal = new TransmittableThreadLocal<>();

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(1);

        // Set initial values
        inheritableThreadLocal.set("Initial Inheritable");
        transmittableThreadLocal.set("Initial Transmittable");

        System.out.println("Main Thread:");
        System.out.println("[Main] inheritableThreadLocal: " + inheritableThreadLocal.get());
        System.out.println("[Main] transmittableThreadLocal: " + transmittableThreadLocal.get());

        // Submit the first task
        executorService.submit(TtlRunnable.get(() -> {
```

```

        System.out.println("=====");
        System.out.println("Task 1 - Thread: " + Thread.currentThread());
        System.out.println("[Task 1] inheritableThreadLocal: " + inheritableThreadLocal.get());
        System.out.println("[Task 1] transmittableThreadLocal: " + transmittableThreadLocal.get());
        // Modify the values in the thread
        inheritableThreadLocal.set("Modified Inheritable in Task 1");
        transmittableThreadLocal.set("Modified Transmittable in Task 1");
    });

    // Update values in the main thread
    inheritableThreadLocal.set("Updated Inheritable in Main");
    transmittableThreadLocal.set("Updated Transmittable in Main");

    // Submit the second task
    executorService.submit(TtlRunnable.get() -> {
        System.out.println("=====");
        System.out.println("Task 2 - Thread: " + Thread.currentThread());
        System.out.println("[Task 2] inheritableThreadLocal: " + inheritableThreadLocal.get());
        System.out.println("[Task 2] transmittableThreadLocal: " + transmittableThreadLocal.get());
    });

    executorService.shutdown();
}
}

```

Main Thread:
[Main] inheritableThreadLocal: Initial Inheritable
[Main] transmittableThreadLocal: Initial Transmittable
=====
Task 1 - Thread: Thread[pool-1-thread-1,5,main]
[Task 1] inheritableThreadLocal: Initial Inheritable
[Task 1] transmittableThreadLocal: Initial Transmittable
=====
Task 2 - Thread: Thread[pool-1-thread-1,5,main]
[Task 2] inheritableThreadLocal: Modified Inheritable in Task 1
[Task 2] transmittableThreadLocal: Updated Transmittable in Main

SXSSFWorkbook hibernate-validator

hibernate-validator = org.hibernate.validator: 可以使用 hibernate 提供的验证注解

依赖:

常用版本: 6.2.0.Final, 6.1.7.Final, 6.1.6.Final, 6.1.5.Final

protoBuf

依赖:

```

<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.12.0</version>
</dependency>

```

使用

freemarker

基于模板和要改变的数据，并用来生成输出文本（HTML 网页、电子邮件、配置文件、源代码等）的通用工具

依赖:

```
<dependency>
</dependency>
```

[freemarker]

Template

```
package freemarker.template;
public class Template extends Configurable 模板
public void process(Object dataModel, Writer out) throws TemplateException, IOException 处理数据
```

Configuration [CORE]

```
package freemarker.template;
public class Configuration extends Configurable implements Cloneable, ParserConfiguration
    org.springframework.ui.freemarker.FreeMarkerTemplateUtils
```

```
public void setClassForTemplateLoading(Class resourceLoaderClass, String basePackageName)
```

```
public void setDefaultEncoding(String encoding)
```

```
public void setTemplateExceptionHandler(TemplateExceptionHandler templateExceptionHandler)
```

```
    configuration = new Configuration(Configuration.VERSION_2_3_25);
    configuration.setClassForTemplateLoading(new TranscriptServiceUtil().getClass(), basePackageName: "/email");
    configuration.setDefaultEncoding("UTF-8");
    configuration.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);
```

```
public Template getTemplate(String name) throws TemplateNotFoundException, MalformedTemplateNameException,
ParseException, IOException
```

```
    Template template = config.getTemplate(sendEmailRequest.getEmailTemplateType().getTemplateName());
```

jsch

依赖:

```
<!-- https://mvnrepository.com/artifact/com.jcraft/jsch -->
<dependency>
    <groupId>com.jcraft</groupId>
    <artifactId>jsch</artifactId>
    <version>0.1.55</version>
</dependency>
```

jasypt

依赖:

```
<dependency>
    <groupId>org.jasypt</groupId>
    <artifactId>jasypt</artifactId>
    <version>1.9.3</version>
</dependency>
```

jasypt

BasicTextEncryptor

```
package org.jasypt.util.text;
public final class BasicTextEncryptor implements TextEncryptor 基础文本加密器
public void setPassword(String password) 设置密码
```

EncryptableProperties

```
package org.jasypt.properties;  
public final class EncryptableProperties extends Properties 加密属性  
    rsql-parser  
        [rsql-parser]
```

RSQParser

```
package cz.jirutka.rsql.parser;  
@Immutable  
public final class RSQParser  
public RSQParser(Set<ComparisonOperator> operators)
```

RSQLOperators

```
package cz.jirutka.rsql.parser.ast;  
public abstract class RSQLOperators  
public static Set<ComparisonOperator> defaultOperators()
```

Extensions

AWS

介绍：AWS SDK for Java 提供适用于 Amazon Web Services 的 Java API。利用此开发工具包，开发者可以轻松构建使用 Amazon S3、Amazon EC2、DynamoDB 等。

依赖：

```
<dependency>  
    <groupId>com.amazonaws</groupId>  
    <artifactId>aws-java-sdk-s3</artifactId>  
    <version>1.11.327</version>  
</dependency>  
<dependency>  
    <groupId>com.amazonaws</groupId>  
    <artifactId>aws-java-sdk-core</artifactId>  
    <version>1.11.327</version>  
</dependency>
```

aspectj

Core

Key Concepts in Spring AOP

Aspect

A module that encapsulates a concern that cuts across multiple objects. An aspect can be implemented using a regular class annotated with `@Aspect`.

Join Point

A point during the execution of a program, such as the execution of a method or the handling of an exception. In

Spring AOP, a join point always represents a method execution.

Pointcut

A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut. Spring AOP uses AspectJ pointcut expression language by default.

```
@Pointcut("execution(* com.example.service.*.*(..))")  
public void serviceMethods() {  
    // Pointcut for all methods in service package  
}
```

Advice

Action taken by an aspect at a particular join point. Different types of advice include:

- Before Advice Executed before a join point.
- After Advice Executed after a join point completes.
- After Returning Advice Executed after a join point completes normally.
- After Throwing Advice Executed if a method exits by throwing an exception.
- Around Advice Executed before and after a join point.

```
@Before("execution(* com.example.service.*.*(..))")  
public void logBefore(JoinPoint joinPoint) {  
    System.out.println("Before method: " + joinPoint.getSignature().getName());  
}  
@After("execution(* com.example.service.*.*(..))")  
public void logAfter(JoinPoint joinPoint) {  
    System.out.println("After method: " + joinPoint.getSignature().getName());  
  
}  
@AfterReturning(pointcut = "execution(* com.example.service.*.*(..))", returning = "result")  
public void logAfterReturning(JoinPoint joinPoint, Object result)  
{  
    System.out.println("After method  
returning: " + joinPoint.getSignature().getName() + " - Result: " + result);  
}  
@AfterThrowing(pointcut = "execution(* com.example.service.*.*(..))", throwing = "exception")  
public void logAfterThrowing(JoinPoint joinPoint, Throwable exception)  
{  
    System.out.println("After method throwing exception: " + joinPoint.getSignature().getName()  
+ " - Exception: " + exception.getMessage());  
}  
@Around("execution(* com.example.service.*.*(..))")  
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {  
    System.out.println("Before method: " + joinPoint.getSignature().getName());  
  
    Object result = joinPoint.proceed();  
    System.out.println("After method: " + joinPoint.getSignature().getName()  
+ " - Result: " + result);  
    return result;  
}
```

Target Object

The object being advised by one or more aspects. Also referred to as the proxied object in Spring AOP.

Proxy

An object created by the AOP framework to implement the aspect contracts (advices).

Spring AOP uses JDK dynamic proxies or CGLIB proxies to create proxy objects.

Weaving

The process of linking aspects with other application types or objects to create an advised object.

This can be done at compile time, load time, or at runtime. Spring AOP performs weaving at runtime.

Introduction

An introduction allows you to add new methods or fields to existing classes. In Spring AOP, this is done using the @DeclareParents annotation.

```
@DeclareParents(value = "com.example.service.*+", defaultImpl = DefaultUsageTracking.class)  
public static UsageTracking mixin;
```

Self-invocation

Reference:

[TransactionProxyFactoryBean](#)

Methods like @Transactional and @Async lose their functionality **during self-invocation** (when a method calls another method in the same class).

The issue arises because self-invocation **bypasses the proxy object**. For example, a call to this.addOne() in the intercepted method won't trigger AOP logic.

Solution:

Use AopContext.currentProxy() to access the proxy object and invoke the method:

```
((Real) AopContext.currentProxy()).addOne();
```

Key Classes and Interfaces

ProxyFactoryBean

This is a FactoryBean implementation that creates **a JDK dynamic proxy or a CGLIB proxy** for the target bean.

AdvisedSupport

This class is **the core representation of a proxy configuration** in Spring AOP. It holds the target source and a list of advisors (advice and pointcuts).

JdkDynamicAopProxy

This class implements the AopProxy interface and is responsible for creating JDK dynamic proxies.

If you are using the default option @EnableAspectJAutoProxy(proxyTargetClass = false), Spring AOP will use JDK dynamic proxies, which only work with interfaces.

If your class does not implement an interface, the aspect will not be applied.

ProxyConfig

Base class for AOP proxy configuration management.

How It Works

Proxy Creation

When a bean requires proxying (e.g., annotated with @Transactional),

Spring checks whether the target bean class implements any interfaces.

If it does, it uses **JdkDynamicAopProxy** to create a JDK dynamic proxy.

Otherwise, it will use **CGLIB** (Code Generation Library) to create a proxy for the class.

Example of creating a JDK dynamic proxy:

```
ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
proxyFactoryBean.setTarget(targetBean);
proxyFactoryBean.addAdvice(new TransactionInterceptor(transactionManager,
transactionAttributeSource));
MyInterface proxy = (MyInterface) proxyFactoryBean.getObject();
```

Interception

The proxy intercepts method calls and **delegates to the advised target**, applying any advice (such as transactions or security checks) configured.

Invocation Handling

JdkDynamicAopProxy uses an **InvocationHandler** to manage method invocations. This handler applies the advice around the method execution.

Dynamic data source switching

Define Custom Annotation

Create a custom annotation to indicate which data source should be used for a specific method or class.

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
```

```

public @interface DataSource {
    String value();
}

Configure Multiple Data Sources
In your Spring configuration, define multiple data sources and a dynamic data source routing class.

@Configuration
public class DataSourceConfig {

    @Bean(name = "dataSource1")
    @Primary
    public DataSource dataSource1() {
        return DataSourceBuilder.create().url("jdbc:url1").username("user1").password("password1").build();
    }

    @Bean(name = "dataSource2")
    public DataSource dataSource2() {
        return DataSourceBuilder.create().url("jdbc:url2").username("user2").password("password2").build();
    }

    @Bean(name = "dynamicDataSource")
    public DataSource dynamicDataSource() {
        Map<Object, Object> targetDataSources = new HashMap<>();
        targetDataSources.put("dataSource1", dataSource1());
        targetDataSources.put("dataSource2", dataSource2());

        DynamicRoutingDataSource dynamicDataSource = new DynamicRoutingDataSource();
        dynamicDataSource.setDefaultTargetDataSource(dataSource1());
        dynamicDataSource.setTargetDataSources(targetDataSources);
        return dynamicDataSource;
    }
}

```

Implement the Dynamic DataSource Routing Class

Create a class that extends AbstractRoutingDataSource to handle the dynamic switching.

`org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource`

```

public class DynamicRoutingDataSource extends AbstractRoutingDataSource {

    private static final ThreadLocal<String> contextHolder = new ThreadLocal<>();

    public static void setDataSourceKey(String key) {
        contextHolder.set(key);
    }

    public static String getDataSourceKey() {
        return contextHolder.get();
    }

    public static void clearDataSourceKey() {
        contextHolder.remove();
    }

    @Override
    protected Object determineCurrentLookupKey() {
        return getDataSourceKey();
    }
}

```

Implement the AOP Aspect for DataSource Switching

Create an AOP aspect that intercepts methods annotated with @DataSource and switches the data source accordingly.

```

@Aspect
@Component
public class DataSourceAspect {

    @Before("@within(dataSource) || @annotation(dataSource)")

```

```

public void switchDataSource(JoinPoint point, DataSource dataSource) {
    String dataSourceKey = dataSource.value();
    if (DynamicRoutingDataSource.getDataSourceKey() == null ||
        !DynamicRoutingDataSource.getDataSourceKey().equals(dataSourceKey)) {
        DynamicRoutingDataSource.setDataSourceKey(dataSourceKey);
    }
}

@After("@within(dataSource) || @annotation(dataSource)")
public void clearDataSource(JoinPoint point, DataSource dataSource) {
    DynamicRoutingDataSource.clearDataSourceKey();
}
}

```

Usage Example

Now, you can use the `@DataSource` annotation to specify which data source to use for specific methods or classes.

```

@Service
public class UserService {

    @DataSource("dataSource1")
    public void methodUsingDataSource1() {
        // Implementation for dataSource1
    }

    @DataSource("dataSource2")
    public void methodUsingDataSource2() {
        // Implementation for dataSource2
    }
}

```

Expressions

Pointcut Expressions

`execution(public User com.itheima.service.UserService.findById(int))`

Matches the `findById` method in the `UserService` class by full class name.

`execution(public User com..UserService.findById(int))`

The `..` matches multiple consecutive symbols.

`execution(public * com.itheima.*.UserService.find*())`

The `*` matches a single arbitrary symbol; matches any method in `UserService` starting with `find` and taking one parameter of any type.

`execution(public * com.itheima.*.UserService+.*())`

The `+` matches subclasses or implementing types of `UserService`.

`execution(* com.itheima.*()) && execution(* com.itheima2.*())`

The `&&` operator matches if both expressions are true.

`execution(* com.itheima.*()) || execution(* com.itheima2.*())`

The `||` operator matches if either expression is true.

`!execution(* com.itheima.*())`

The `!` operator negates the match.

`execution(* com.itheima.*()) && args(a,b)`

Matches methods with two arguments `a` and `b`. `args` locks the variable names, and `execution` locks the variable types (e.g., `public void before(int a, int b)`).

`execution(public !static * *(..))`

Matches non-static methods; `..` matches any number of arguments, and `*` matches a single argument.

Keywords and Matching Patterns

`execution`

Matches methods based on method declarations and `fully qualified class names`.

args

Matches methods based on method parameters (e.g., @Pointcut("args(com.ms.aop.args.demo1.UserModel,...)") matches methods with parameters of specific types).

within

Matches methods in target classes based on package or fully qualified class names. For example:

within(com.test.*)

Matches all methods in the com.test package.

within(com.test..*)

Matches all methods in com.test and its sub-packages.

within(com.test.UserService)

Matches all methods in the UserService class.

this

Matches **based on the type of the AOP proxy object** (dynamic proxy), handling methods declared in the proxy class but not those inherited from parent classes.

target

Matches **based on the type of the proxied target object** (can include inherited methods).

@args

Matches methods with parameters annotated with a specific annotation.

@within

Matches classes annotated with a specific annotation (static).

@target

Matches runtime target objects annotated with a specific annotation.

@annotation

Matches methods annotated with a specific annotation.

execution (public User com.itheima.service.UserService.findById(int)) 切入点表达式, 全类名

execution (public User com..UserService.findById(int))	.. 匹配多个连续符号
execution (public * com.itheima.*.UserService.find* (*))	* 匹配单个任意符号
execution (public * com.itheima.*.UserService+. * (*))	+ 匹配子类型
execution (* com.itheima.* (*)) && execution (* com.itheima2.* (*))	&& 同时成立
execution (* com.itheima.* (*)) execution (* com.itheima2.* (*))	任意一个成立, 这里的星号表达式表示方法
!execution (* com.itheima.* (*))	! 不成立匹配
execution (* com.itheima.* (*)) && args(a,b)	锁定通知变量名, execution 可以锁定变量类型
// public void before(int a, int b)	
execution(public !static * *(..))	匹配非 static 方法, .. 表示任意个数参数 * 表示任意一个参数

关键字: 描述表达式的匹配模式 (参看关键字列表)

execution 根据**方法声明和全类名**匹配 目标方法 (比 within 范围大)

args 根据**方法参数** 匹配 目标方法 // @Pointcut("args(com.ms.aop.args.demo1.UserModel,...)") 匹配任意多个参数

within 根据**包名或全类名**匹配 目标类, 拦截**包中所有类的所用方法** // within(com.test.*) 匹配 test 包 // within(com.test..*) 匹配 test 包,

以及 test 子包

// within(com.test.UserService) 匹配

UserService 类下的所有方法

this 根据目标对象 aop 生成的代理对象的类型匹配 目标类，父类或父接口 Father 未重写的方法不会被处理 //
this(com.xyz.service.Father) GrandFather say() --> Father say() --> Son father.say() 匹配 son.other()

不匹配 (此时 this 不匹配, target 匹配)

target 根据被代理的目标对象的类型匹配 目标类，父类或父接口 Father 未重写的方法也会被处理 //
target(com.xyz.service.Father) GrandFather say() --> Father say() --> Son say() father.say() 匹配 son.say()
匹配 (此时 this, target 都匹配) 0

@args 根据注解全类名 和 方法参数 匹配 目标方法 // @args(com.test.MyAnnotation, ..) 匹配 标注了当前注解 MyAnnotation，并且参数类型相同的 方法 (.. 表示匹配任何参数)

@within 根据注解全类名 匹配 目标类，静态匹配，适合拦截静态方法 // @within(com.test.MyAnnotation)
匹配 标注了当前注解 MyAnnotation 的 类

@target 根据注解全类名 匹配 目标类，运行时匹配，适合拦截非静态方法 // @target(com.test.MyAnnotation)
匹配 标注了当前注解 MyAnnotation 的 类

@annotation 根据注解全类名 匹配 目标方法 // @annotation(com.test.MyAnnotation)
bean
reference pointcut

访问修饰符：方法的访问控制权限修饰符

类名：方法所在的类（此处可以配置接口名称）

异常：方法定义中指定抛出的异常

含有@Transactional 和@Async 的方法 testAsync，在一个类中被 aop 拦截的方法 testAop 中被调用时，发生 this 自调用，
@Transactional 和@Async 会失效

可以看到切面的内容是定义在 invoke 方法中，而我们实际通过反射调用的 add 方法是原本 Real 实例中的 add 方法，因此 Real 实例中的 this 就是 Real 实例本身，而不是代理类，所以最后调用的 this.addOne()方法，不会做异步处理。

Aop 避免自调用：((Real) AopContext.currentProxy()).addOne(); 在 apo 拦截的方法 add 内部添加这个调用

Configuration

Add Dependencies

Maven Configuration

```
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.22.1</version>
    <scope>runtime</scope>
</dependency>
```

Or

```

<dependencies>
    <!-- Spring Boot Starter AOP for AspectJ support -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
</dependencies>

```

Gradle Configuration

```

dependencies {
    // Spring Boot Starter AOP for AspectJ support
    implementation 'org.springframework.boot:spring-boot-starter-aop'
}

```

Enable AspectJ Auto-Proxying

In Spring Boot, `@EnableAspectJAutoProxy` is automatically enabled when you include the `spring-boot-starter-aop` dependency. However, if you want to explicitly enable AspectJ auto-proxying in a configuration class,

```

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
public class AopConfig {
}

```

Create an Aspect Class

An Aspect is a class that contains methods for cross-cutting concerns. These methods are typically annotated with AspectJ annotations like `@Before`, `@After`, `@Around`, etc.

```

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component // Make sure the aspect is registered as a Spring Bean
public class LoggingAspect {

    // Define a pointcut expression for the methods to be advised
    @Before("execution(* com.example..*(..))")
    public void logBefore() {
        System.out.println("Before method execution...");
    }
}

```

Define Target Classes and Methods

Now, create a service or any other class that will be targeted by the aspect.

```

import org.springframework.stereotype.Service;

@Service
public class MyService {

    public void performTask() {
        System.out.println("Performing task...");
    }

    public void anotherTask() {
        System.out.println("Another task...");
    }
}

```

[aspectjweaver]

```
package org.aspectj.lang
public interface JoinPoint
Object[] getArgs()          获取代理函数参数

package org.aspectj.lang
public interface ProceedingJoinPoint extends JoinPoint    执行的连接点
Object proceed()          实现对原始方法调用, 无法预知原始方法中是否出现异常 // pjp.proceed(pjp.getArgs()) 可以不穿
参数, 也可以获取函数参数
Object[] getArgs();        获取目标方法参数值
Signature getSignature();  获取目标方法
Object getTarget();        获取 IOC 容器内目标对象

package org.aspectj.lang
public interface Signature    方法
```

```
package org.aspectj.lang.reflect;
public interface MethodSignature extends CodeSignature    方法名和形参列表共同组成方法签名。
Class getReturnType();    获取返回值
Method getMethod();      获取方法
```

```
package org.springframework.context.annotation
@EnableAspectJAutoProxy    设置当前类开启 AOP 注解驱动的支持, 加载 AOP 注解 (在 SpringConfig 类上使用) 【TYPE】
使用
```

```
com.saidake.aop.MyAdvice >>      xml 式配置使用
public class MyAdvice {
    public void testfunc(ProceedingJoinPoint pjp){
        Object ret=pjp.proceed();
        System.out.println("test func Myadvice");
    }
}
```

```
com.saidake.aop.AopAdvice >>    声明式配置使用
@Aspect                  设置当前类为切面类 (使用@Order 注解通过变更 bean 的加载顺序改变通知的加载顺序)
@Component
@Configuration
public class AOPAdvice {
    @Pointcut("execution ( *.* ( .. ) ) ")    使用当前方法名作为切入点引用名称
    public void pt() {}                      切面类中定义的切入点只能在当前类中使用, 如果想引用其他类中定义的切入点使用
    "类名.方法名()"引用
                                    切入点最终体现为一个方法, 无参无返回值, 无实际方法体内容, 但不能是抽象方法
    (引用切入点时必须使用方法调用名称, 方法后面的()不能省略)
```

```
@Before("pt()")
前置通知, 原始方法执行前执行, 如果通知中抛出异常, 阻止原始方法运行 (同
一个通知类中, 相同通知类型以方法名排序为准。不同通知类中, 以类名排序为准)
```

```

public void before() {
    System.out.println("前置 before. ..");
}

@After("pt()") 后置通知，原始方法执行后执行，无论原始方法中是否出现异常，都将执行通知
public void after() {
    System.out.println("后置 after. ..");
}

@AfterReturning(value="pt()",returning="ret") 返回后通知，原始方法正常执行完毕并返回结果后执行，如果原始方法中抛出异常，无法执行
returning 设定使用通知方法参数 接收返回值（目标方法必须返回指定类型的值或没有返回值）
public void afterReturing(Object ret) {
    System.out.println("返回后 afterReturing. ..");
}

@AfterThrowing(value="pt()",throwing="t") 异常后通知，原始方法抛出异常后执行，如果原始方法没有抛出异常，无法执行
throwing 设定使用通知方法参数接收原始方法中抛出的异常对象名
public void afterThrowing(Throwable t) {
    System.out.println("抛出异常后 afterThrowing. . .");
}

@Around("pt()") 环绕通知，在原始方法执行前后添加逻辑，还可以阻止原始方法的执行
public Object around(ProceedingJoinPoint pjp, SysLog sysLog) throws Throwable {
    // 参数二可以用来获取注解对象，也可以省略
    System.out.println("环绕前 around before. . .");
    Object ret = pjp.proceed(); // try catch 时，抛出的各种异常同样可以被 ControllerAdvice 拦截
    System.out.println("环绕后 around after. . .");
    return ret; // 返回原始返回值
}

```

ehcache

依赖:

```

<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.10.0</version>
</dependency>

```

配置:

EhCacheConfig >>

```

@Bean
public Cache<Long, String> testCache(){
    CacheConfiguration<Long, String> cacheConfiguration= CacheConfigurationBuilder.newCacheConfigurationBuilder(
        Long.class, String.class, ResourcePoolsBuilder.heap(100)
    ).withExpiry(ExpiryPolicyBuilder.timeToIdleExpiration(Duration.ofMinutes(50L))).build();
}

```

```

CacheManager cacheManager =
CacheManagerBuilder.newCacheManagerBuilder().withCache("testCache",cacheConfiguration)
    .build();
cacheManager.init();
return cacheManager.getCache("testCache",Long.class, String.class);
}

```

源码解析 ehcache-3.7.0

```

package org.ehcache.config.builders;
public class CacheConfigurationBuilder<K, V> implements FluentCacheConfigurationBuilder<K, V,
CacheConfigurationBuilder<K, V>> 缓存构建器
public static <K, V> CacheConfigurationBuilder<K, V> newCacheConfigurationBuilder(Class<K> keyType, Class<V>
valueType, ResourcePools resourcePools) 创建新的 builder
public CacheConfigurationBuilder<K, V> withExpiry(ExpiryPolicy<? super K, ? super V> expiry) 设置过期时间
public CacheConfiguration<K, V> build() 返回缓存配置

```

```

package org.ehcache.config.builders;
public class ResourcePoolsBuilder implements Builder<ResourcePools> 资源池构建器
public static ResourcePoolsBuilder heap(long entries) 设置资源池堆大小

```

```

package org.ehcache.config.builders;
public final class ExpiryPolicyBuilder<K, V> implements Builder<ExpiryPolicy<K, V>> 过期策略构建起
public static ExpiryPolicy<Object, Object> timeToLiveExpiration(Duration timeToLive) 时间转换成过期时间

```

```

package org.ehcache.config.builders;
public class CacheManagerBuilder<T extends CacheManager> implements Builder<T> 缓存管理构建器
public static CacheManagerBuilder<CacheManager> newCacheManagerBuilder() 创建新的缓存管理构建器
public <K, V> CacheManagerBuilder<T> withCache(String alias, CacheConfiguration<K, V> configuration) 添加缓存信息
public T build() 构建管理器

```

```

package org.ehcache;
public interface CacheManager extends Closeable 缓存管理器
void init() throws StateTransitionException 初始化
<K, V> Cache<K, V> getCache(String alias, Class<K> keyType, Class<V> valueType) 获取最终构建的缓存区域

```

SAP CAP

The SAP Cloud Application Programming Model (CAP) is a framework of **languages**, **libraries**, and **tools** for building enterprise-grade services and applications.

The CAP framework features a mix of proven and broadly adopted open-source and SAP technologies, as highlighted in the figure below.

On top of open source technologies, CAP mainly adds:

- **Core Data Services (CDS)** as our universal modeling language for both domain models and service definitions.
- **Service SDKs and runtimes** for Node.js and Java, offering libraries to implement and consume services as well as generic provider implementations serving many requests automatically.

Reference

<https://cap.cloud.sap/docs/java/getting-started>

Changes in JUnit 5 Compared to JUnit 4

Annotation

```
@Before      --> @BeforeEach
@BeforeClass --> @BeforeAll
@Rule        --> @ExtendWith
```

Package

```
package org.junit; --> package org.junit.jupiter.api;
```

Environment Variables

The environment variables configured for starting SpringApplication typically **won't take effect** in JUnit tests or when running mvn test.

Pass environment variables to junit test:

- @TestPropertySource(properties = "key=value") for Spring tests
- System.setProperty("key", "value") before the test runs
- Configuring the surefire plugin in pom.xml to pass environment variables

Configuration

Add Dependencies

Spring Boot comes with testing libraries, including JUnit 5 and Spring Boot's test framework, automatically included via the spring-boot-starter-test dependency.

Add the following dependency to your pom.xml (if you're using Maven):

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

For Gradle

```
// JUnit 5 and Mockito dependencies for testing
testImplementation 'org.springframework.boot:spring-boot-starter-test' // Includes JUnit 5 by default
testImplementation 'org.mockito:mockito-core:5.5.0' // Add Mockito if needed
testImplementation 'org.mockito:mockito-jupiter:5.5.0' // For Mockito integration with JUnit 5
```

Simple Unit Test

JUnit 5 (also known as Jupiter) is the default testing framework in Spring Boot. A basic unit test involves writing test cases for individual methods or components without starting the full Spring context.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class MyServiceTest {

    @Test
    void testAdd() {
        MyService myService = new MyService();
        int result = myService.add(2, 3);
        assertEquals(5, result); // JUnit assertion
    }
}
```

Spring Boot Test

In many cases, you will want to test Spring-managed components like controllers, services, and repositories. Spring Boot provides the `@SpringBootTest` annotation to load the entire application context for integration tests.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import static org.junit.jupiter.api.Assertions.assertEquals;

@SpringBootTest(classes = SimiWebFluxApp.class)
class MyServiceIntegrationTest {

    @Autowired
    private MyService myService;

    @Test
    void testAdd() {
        int result = myService.add(2, 3);
        assertEquals(5, result); // JUnit assertion
    }
}
```

Mockito Test

Using `SpringBootTest`

If your service class has dependencies (like repositories or other services), you can `mock` them using Mockito to isolate the unit being tested.

The `@Mock` annotation is used to `create mock objects`, and the `@InjectMocks` annotation is used to `inject these mocks into the object` being tested.

```
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.boot.test.context.SpringBootTest;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

@SpringBootTest
class MyServiceTest {

    @Mock
    private MyRepository myRepository;

    @InjectMocks
    private MyService myService;

    @Test
    void testFindById() {
        MyEntity entity = new MyEntity(1L, "Test Name");
        when(myRepository.findById(1L)).thenReturn(Optional.of(entity));

        String result = myService.findNameById(1L);
        assertEquals("Test Name", result);
    }
}
```

Using `ExtendWith`

Mockito can be integrated with JUnit 5 via the `MockitoExtension`, which `automates mock initialization`, removing the need for `MockitoAnnotations.openMocks(this)`.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
```

```

import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.when;
import static org.junit.jupiter.api.Assertions.assertEquals;

@ExtendWith(MockitoExtension.class) // Enable MockitoExtension
class MyServiceTest {

    @Mock
    private MyRepository myRepository; // This mock will be injected into myService instance.

    @InjectMocks
    private MyService myService;

    @Test
    void testFindById() {
        // Define behavior of the mock
        when(myRepository.findById(1L)).thenReturn(Optional.of(new MyEntity(1L, "Test")));

        // Run the test
        String result = myService.findNameById(1L);
        assertEquals("Test", result);
    }
}

```

WebMvc Test

To test only the web layer (controllers) without loading the entire application context, use the `@WebMvcTest` annotation.

```

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(MyController.class)
class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetEndpoint() throws Exception {
        mockMvc.perform(get("/api/my-endpoint"))
            .andExpect(status().isOk());
    }
}

```

[mockito-core]

Mockito

```

@CheckReturnValue
public class Mockito extends ArgumentMatchers
public static Stubber doReturn(Object toBeReturned)

```

When to Use `doReturn`

Use `doReturn` when stubbing a spy to prevent real method execution.

Use `doReturn` when dealing with final/private methods (in combination with PowerMockito).

Avoid `doReturn` if `when(...).thenReturn(...)` works fine, as it is more readable.

```

@Test
void testDoReturnWithMock() {
    List<String> mockList = mock(List.class);
}

```

```

    // Stubbing with doReturn
    doReturn("Mocked Value").when(mockList).get(0);

    // Output: Mocked Value
    System.out.println(mockList.get(0));
}

Using doReturn for Any Argument
doReturn("mocked result").when(mockService).fetchData(any());

```

public static Stubber `doReturn`(Object toBeReturned, Object... toBeReturnedNext)

This method is particularly useful for scenarios where you want a method to return different values on consecutive calls.

```

@Test
void testDoReturnWithMultipleValues() {
    // Create a mock List
    List<String> mockList = mock(List.class);

    // Stub the get method to return different values on subsequent calls
    doReturn("First Value").doReturn("Second Value").when(mockList).get(0);

    // First call returns "First Value"
    System.out.println(mockList.get(0)); // Output: First Value

    // Subsequent call still returns "First Value" as we didn't change the index
    System.out.println(mockList.get(0)); // Output: First Value

    // Now let's stub another call for a different index
    doReturn("New Value").when(mockList).get(1);

    // Call for index 1
    System.out.println(mockList.get(1)); // Output: New Value
}

```

public static <T> T `verify`(T mock, VerificationMode mode)

`mock` → The mocked or spied object being verified.

`mode` → **The verification mode**, which defines how many times the method should have been called.

Verify that a specific method was called on a mocked or spied object **a certain number of times or with specific verification conditions**.

```

import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.List;

class VerifyTest {
    @Test
    void testVerifyWithMode() {
        List<String> mockList = mock(List.class);

        mockList.add("A");
        mockList.add("B");
        mockList.add("B");

        verify(mockList, times(1)).add("A"); // ✓ Called once
        verify(mockList, times(2)).add("B"); // ✓ Called twice
        verify(mockList, never()).add("C"); // ✓ Never called
    }
}

```

OngoingStubbing

```

package org.mockito.stubbing;
@NotExtensible
public interface OngoingStubbing<T>

```

```

OngoingStubbing<T> thenReturn(T var1);
OngoingStubbing<T> thenReturn(T var1, T... var2);
OngoingStubbing<T> thenThrow(Throwable... var1);
OngoingStubbing<T> thenThrow(Class<? extends Throwable> var1);
OngoingStubbing<T> thenThrow(Class<? extends Throwable> var1, Class<? extends Throwable>... var2);
OngoingStubbing<T> thenCallRealMethod();
OngoingStubbing<T> thenAnswer(Answer<?> var1);

```

Define custom behavior for mocked methods using a lambda expression or an implementation of the Answer interface.

This is useful when you need dynamic responses based on input arguments or when you want to return a new instance on each invocation.

```

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

public class MockResultSetTest {
    @Test
    void testExecuteQueryReturnsNewResultSet() throws SQLException {
        // Mock Statement
        Statement statement = mock(Statement.class);

        // Configure statement.executeQuery to return a new ResultSet each time
        when(statement.executeQuery(anyString())).thenAnswer(invocation -> {
            // Create a new ResultSet mock for each call
            ResultSet resultSet = mock(ResultSet.class);
            doReturn(true, false).when(resultSet).next(); // First call returns true, second call
        returns false
            return resultSet;
        });

        // First call
        ResultSet resultSet1 = statement.executeQuery("SELECT * FROM table");
        assertTrue(resultSet1.next());
        assertFalse(resultSet1.next());

        // Second call should return a fresh instance
        ResultSet resultSet2 = statement.executeQuery("SELECT * FROM table");
        assertTrue(resultSet2.next());
        assertFalse(resultSet2.next());

        // Ensure the two result sets are different instances
        assertNotSame(resultSet1, resultSet2);
    }
}

OngoingStubbing<T> then(Answer<?> var1);
<M> M getMock();

```

(annotation)

Spy

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
@Documented
public @interface Spy

```

In Mockito, `@Spy` is used to create a partial mock of a real object.

Difference between `@Spy` and `@Mock`.

- `@Mock` creates a **pure mock**, meaning all method calls return default values (e.g., null for objects, 0 for numbers) unless explicitly stubbed. The actual method implementations are **not** invoked.
- `@Spy` creates a **partial mock**, meaning the actual methods are called unless explicitly stubbed. It allows spying on real objects while enabling selective method stubbing.

Mock

```
package org.mockito;  
@Target({FIELD, PARAMETER})  
@Retention(RUNTIME)  
@Documented  
public @interface Mock
```

Create mock instances of a class or an interface.

These mocks simulate the behavior of the actual class/interface and can be programmed to return specific values or throw exceptions when methods are called.

InjectMocks

```
package org.mockito;  
@Documented  
@Target(FIELD)  
@Retention(RUNTIME)  
public @interface InjectMocks
```

When you annotate a field with `@InjectMocks`, Mockito automatically injects the mocks (created with `@Mock`) into the fields and constructor parameters of the class under test.

If you're not using `@ExtendWith(MockitoExtension.class)`, you must call `MockitoAnnotations.openMocks(this);` in a `@BeforeEach` method to initialize the mocks.

```
MockitoAnnotations.openMocks(this);
```

[junit-jupiter-api]

ExtendWith

```
package org.junit.jupiter.api.extension;  
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@Repeatable(Extensions.class)  
@API(  
    status = Status.STABLE,  
    since = "5.0"  
)
```

```
public @interface ExtendWith
```

Register custom extensions, which provide additional behavior to test classes, similar to JUnit 4's `@Rule` and `@RunWith`.

```
@ExtendWith(MockitoExtension.class)
```

Enables Mockito integration.

```
@ExtendWith(SpringExtension.class)
```

Used in Spring Boot tests.

Using a Custom Extension

```

import org.junit.jupiter.api.extension.*;

@ExtendWith(MyExtension.class)
public class MyTest {

    @Test
    void test() {
        System.out.println("Executing test...");
    }
}

class MyExtension implements BeforeEachCallback, AfterEachCallback {
    @Override
    public void beforeEach(ExtensionContext context) {
        System.out.println("Before each test");
    }

    @Override
    public void afterEach(ExtensionContext context) {
        System.out.println("After each test");
    }
}

```

Output:

```

Before each test
Executing test...
After each test

```

@Rule in Junit4

In JUnit 4, @Rule marks a public, non-static instance field that holds a test rule.

This rule is applied to each test method within the class.

@Rule is just a marker, while the actual logic is inside the `TestRule` implementation.

A JUnit 4 rule is an object that implements `TestRule`, and its logic is executed before and after each test method.

When you annotate a field with `@Rule`, JUnit automatically manages the lifecycle of that rule, assigning it a value and executing its logic `before and after each test method`.

```

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class MyTest {

    @Rule
    public ExpectedException exceptionRule = ExpectedException.none();

    @Test
    public void testException() {
        exceptionRule.expect(IllegalArgumentException.class);
        throw new IllegalArgumentException("Error");
    }
}

```

Other

介绍

JUnit5 = JUnit Platform + Junit Jupiter + Junit Vintage

JUnit Platform: Junit Platform 是在 JVM 上启动测试框架的基础，不仅支持 Junit 自制的测试引擎，其他测试引擎也可以接入

JUnit Jupiter: JUnit5 新的测试引擎，用于在 Junit Platform 上运行

JUnit Vintage: 兼容 JUnit4.x, Junit3.x 测试引擎，为了照顾老的项目 【SpringBoot2.4 以上版本移除了默认对 Vintage 的依

赖，需要兼容junit4 需要自动引入】

注意：junit 单元测试不支持多线程，主线程结束子线程也会被终结（用 main 函数就行）

junit-jupiter-api

JUnit 4 对应使用的是：@Before 和 @BeforeClass

JUnit 5 对应使用的是：@BeforeEach 和 *@BeforeAll

Assertions

```
package org.junit.jupiter.api;
```

@BeforeEach 每个之前

@BeforeAll 所有之前

```
package org.junit;
```

从 4.11 版本开始,JUnit 将默认使用一个确定的,但不可预测的顺序(MethodSorters.DEFAULT)。

要改变测试执行的顺序只需要在测试类(class)上使用 @FixMethodOrder 注解,并指定一个可用的 MethodSorter 即可:

@FixMethodOrder(MethodSorters.JVM) :保留测试方法的执行顺序为 JVM 返回的顺序。每次测试的执行顺序有可能会所不同。

@FixMethodOrder(MethodSorters.NAME_ASCENDING) :根据测试方法的方法名排序,按照词典排序规则(ASC,从小到大,递增)。

使用

XXXTest >>

```
@SpringBootTest
```

```
@AutoConfigureMockMvc
```

@RunWith(SpringRunner.class) junit4 使用, 可选

@ExtendWith(SpringExtension.class) junit5 使用, 可选

```
    @ExtendWith(MockitoExtension.class)
```

@TestInstance 测试实例的生命周期, 加上后@BeforeAll 就可以加在非静态方法上了

@DisplayName("xxx") 测试名称, 可选

@Transactional 测试完毕自动回滚, 可选 (javax.transaction.Transactional)

@WebAppConfiguration 可以在单元测试的时候, 不用启动 Servlet 容器, 就可以获取一个 Web 应用上下文, 可选

```
class XXXTest{
```

```
    @Autowired
```

```
    private WebApplicationContext context;
```

```
    private MockMvc mockMvc;
```

```
    private ObjectMapper mapper;
```

```
    @Before
```

```
    public void setup() {
```

```
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context).apply(springSecurity()).build();
```

```
        this.mapper = new ObjectMapper();
```

```
        this.mapper.setPropertyNamingStrategy(PropertyNamingStrategy.SNAKE_CASE); // 转驼峰
```

```
}
```

```
    @Test
```

```
    public void orderSubmitByCyToCy() {
```

```
        RequestBuilder request = null;
```

```
        try {
```

```

request = MockMvcRequestBuilders.post("/api/shipping/shipper/booking_order_submit") //构建一个请求
    .content(this.mapper.writeValueAsString(submitBookingOrderRequest))
    .contentType(MediaType.APPLICATION_JSON_UTF8);
} catch (JsonProcessingException e) {
    e.printStackTrace();
}
try {
    ResultActions resultActions = mockMvc.perform(request); //执行一个请求
    resultActions.andReturn().getResponse().setCharacterEncoding("UTF-8");
    resultActions.andExpect(status().isOk()) // 期望值
        .andExpect(jsonPath("code").value("000"))
        .andDo(MockMvcResultHandlers.print())
        .andReturn();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

@Test 单元测试方法 (Junit4: org.junit.Test Junit5: org.junit.jupiter.api.Test)

@Transactional 测试完毕自动回滚 (javax.transaction.Transactional)

@RepeatedTest 方法可以重复执行

@DisplayName("xxx") 测试名称

@Tag 测试类别

@BeforeEach 每个测试方法前运行 (可以获取 webApplicationContext)

@AfterEach 每个测试方法后运行

@BeforeAll 所有测试方法前运行 (静态 static)

@AfterAll 所有测试方法后运行 (静态 static)

@Disabled 测试方法不用执行

@Timeout(value=500,unit=TimeUnit.MILLISECONDS) 超时

@ExtendWith

@RepeatedTest(5) 多次执行

void testfunc(){

assumeTrue(xxtrue, ["xxxmsg"]) 假设失败跳过

Assertions.assertEquals(5, xxxval, ["xxxmsg"]) 相等断言 (前面的断言执行后, 后面的代码不会再执行)

assertNotEquals(5, xxxval, ["xxxmsg"]) 不相等断言

assertSame(xxobj, xxobj, ["xxxmsg"]) 两个对象指向相同对象断言

assertNotSame(xxobj, xxobj, ["xxxmsg"]) 两个对象指向不同对象断言

assertArrayEquals(xxarr, xxarr, ["xxxmsg"]) 两个对象或原始数组是否相等

assertTrue(xxtrue, ["xxxmsg"]) true 断言

assertFalse false 断言

assertNull 为 null 断言

assertNotNull 不为 null 断言

assertTimeout(Duration.ofMillis(1000), ()->Thread.sleep(500)) 超时断言

assertThrow(ArithmeticException.class, ()->{ xxxexecode }, ["xxxmsg"]) 断定一定出现异常

```

    assertAll("test", ()->assertTrue(true),()>assertEquals(1,2) )      组名，多组断言需要全部成功

    fail("xxx")      直接使单元测试失败
}

```

@ParameterizedTest 参数化测试，每个参数都执行一次测试 (junit-jupiter-params)

```

@ValueSource(strings={"xx", "xx"})          为参数化测试指定入参来源，支持基础类型
@NullSource                                null 入参
@EnumSource                                 枚举入参
@CsvFileSource                            csv 文件入参
@MethodSource("stringProvider")           读取指定方法返回值作为参数化入参（方法返回一个流） static Stream<String>
stringProvider(){ return Stream.of("xx", "xx") }

void testparam(String argument){

}

static Stream<Arguments> stringIntAndListProvider() {      提供参数
    return Stream.of(
        arguments("apple", 1, Arrays.asList("a", "b")),
        arguments("lemon", 2, Arrays.asList("x", "y"))
    );
}

```

Stack<Object> stack;

@Nested 嵌套测试（外层的 Test 不能 驱动 内层的 BeforeEach 之类的方法提前运行）

```

class XXXTestInside{
    @Test
    void testinside(){  内层测试会触发外层的 BeforeEach 之类的方法运行
    }
}

```

netty

Netty: 基于 JAVA NIO 类库的异步通信框架，是典型的 Reactor 模型结构。

基于 JAVA NIO 提供的 API 实现。它提供了对 TCP、UDP 和文件传输的支持。

作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞 的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果。

Reactor 线程模型

常用的 Reactor 线程模型有三种，Reactor 单线程模型，Reactor 多线程模型，主从 Reactor 多线程模型。

1) Reactor 单线程模型：所有的 IO 操作都在同一个 NIO 线程上面完成

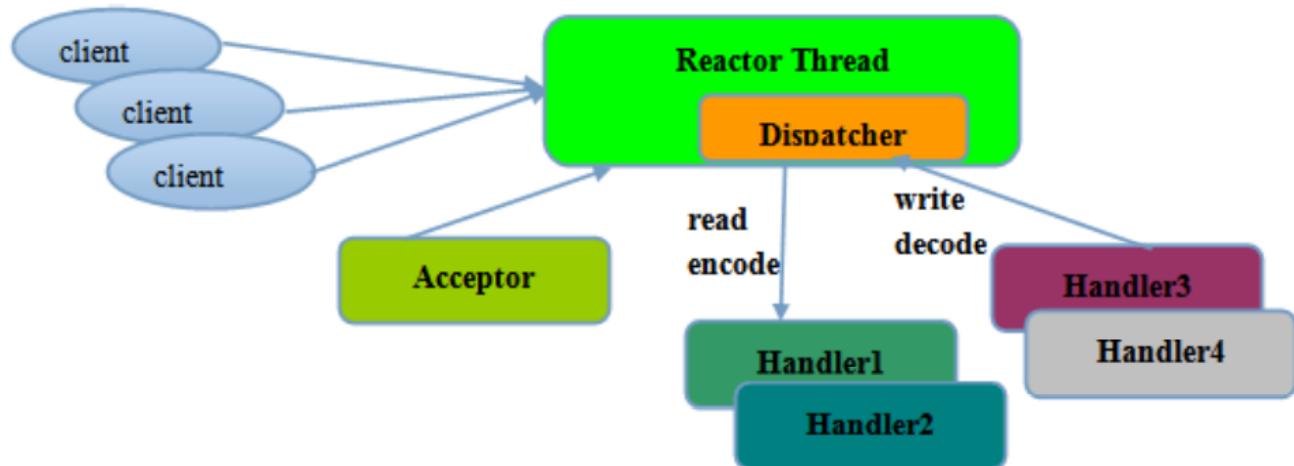
由于所有的 IO 操作都不会导致阻塞，理论上一个线程可以独立处理所有 IO 相关的操作

例如，通过 Acceptor 接收客户端的 TCP 连接请求消息，链路建立成功之后，通过 Dispatch 将对应的 ByteBuffer 派发到指定的 Handler 上进行消息解码。用户 Handler 可以通过 NIO 线程将消息发送给客户端。

NIO 线程的职责如下：

- 作为 NIO 服务端，接收客户端的 TCP 连接；
- 作为 NIO 客户端，向服务端发起 TCP 连接；
- 读取通信对端的请求或者应答消息；

d) 向通信对端发送消息请求或者应答消息。

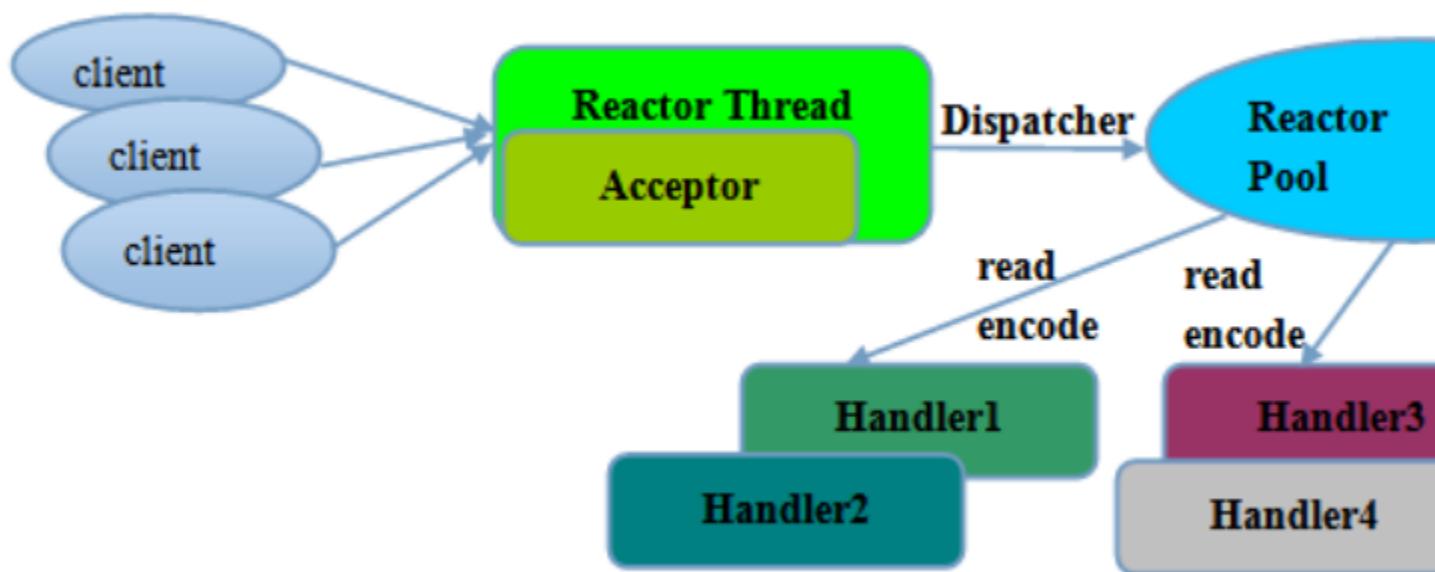


2) Reactor 多线程模型

Reactor 多线程模型与单线程模型最大的区别就是有一组 NIO 线程处理 IO 操作。

有专门一个 NIO 线程 Acceptor 线程用于监听服务端，接收客户端的 TCP 连接请求；

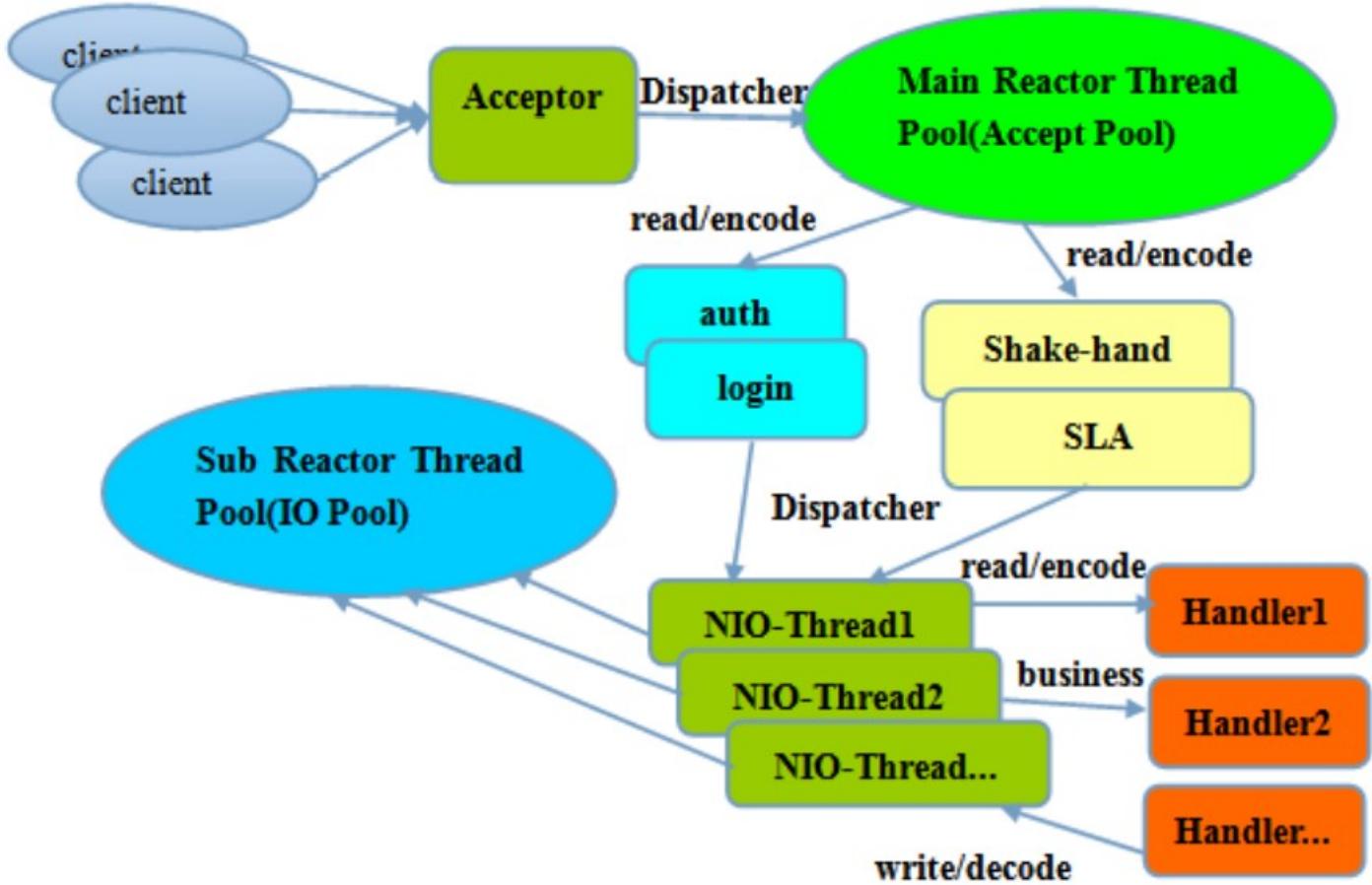
网络 IO 操作由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线程池实现，它包含一个任务队列和 N 个可用的线程，由这些 NIO 线程负责消息的读取、解码、编码和发送；



3) 主从 Reactor 多线程模型

主从 Reactor 多线程模型 服务端用于接收客户端连接的不再是个 1 个单独的 NIO 线程，而是一个独立的 NIO 线程池。

Acceptor 接收到客户端 TCP 连接请求处理完成后（可能包含接入认证等），将新创建的 SocketChannel 注册到 IO 线程池（sub reactor 线程池）的某个 IO 线程上，由它负责 SocketChannel 的读写和编解码工作。



Acceptor 线程池仅仅只用于客户端的登陆、握手和安全 认证，一旦链路建立成功，就将链路注册到后端 subReactor 线程池的 IO 线程上，由 IO 线程负责后续的 IO 操作。

Netty 中的 Boss 线程组充当 mainReactor，处理建立连接的请求。

NioWorker 线程组充当 subReactor，处理 IO 事件（默认 NioWorker 的个数是 Runtime.getRuntime().availableProcessors()）

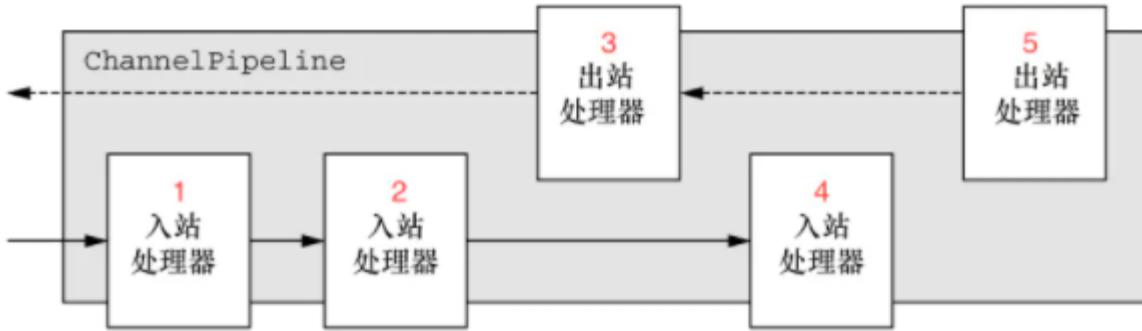
在处理新来的请求 时，NioWorker 读完已收到的数据到 ChannelBuffer 中，之后触发 ChannelPipeline 中的 ChannelHandler 流。

源码解析

```

package io.netty.channel;
public interface EventLoopGroup extends EventExecutorGroup    通过 nio 方式来接收连接和处理连接
public interface ChannelFuture extends Future<Void>      相当于 JDK 的 java.util.concurrent.Future 用于异步操作通知回调
ChannelFuture sync() throws InterruptedException;
Channel channel();
public interface Channel extends AttributeMap, ChannelOutboundInvoker, Comparable<Channel>
ChannelFuture closeFuture();    监听服务器关闭监听
public interface ChannelPipeline extends ChannelInboundInvoker, ChannelOutboundInvoker, Iterable<Entry<String,
ChannelHandler>>    将多个 ChannelHandler 组合在一起，形成一个链条，这个链条会拦截 Channel 上的事件，然后在链条中传播。

```



```
ChannelPipeline addLast(ChannelHandler... var1);
public interface EventLoopGroup extends EventExecutorGroup
```

```
package io.netty.channel.nio;
public class NioEventLoopGroup extends MultithreadEventLoopGroup    声明线程组
```

```
package io.netty.util.concurrent;
public interface EventExecutorGroup extends ScheduledExecutorService, Iterable<EventExecutor>
Future<?> shutdownGracefully();    优雅退出，释放线程池资源
```

```
package io.netty.bootstrap;
public class ServerBootstrap extends AbstractBootstrap<ServerBootstrap, ServerChannel>    负责初始化 netty 服务器，并且
开始监听端口的 socket 请求。
public ServerBootstrap group(EventLoopGroup group)    将线程组传递到 ServerBootstrap 中
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup)
public B channel(Class<? extends C> channelClass)          传递第一个 Channel 类类型 (NioServerSocketChannel)
相当于 NIO 中的 ServerSocketChannel 类
public ServerBootstrap childHandler(ChannelHandler childHandler)    绑定 I/O 事件处理类
public abstract class AbstractBootstrap<B extends AbstractBootstrap<B, C>, C extends Channel> implements Cloneable
public ChannelFuture bind(int inetPort)    服务器绑定端口监听
```

```
package io.netty.handler.codec;
public class LineBasedFrameDecoder extends ByteToMessageDecoder
public LineBasedFrameDecoder(int maxLength)      region 解决粘包/拆包问题
                                                 reactor
```

依赖: reactor-core = io.projectreactor

源码解析 reactor-core-3.4.14

```
package reactor.core.publisher;
public abstract class Mono<T> implements CorePublisher<T>

public static <T> Mono<T> just(T data)    恶汉式，比较霸道，只创建一次          // Mono<Date> m1 =
Mono.just(new Date());
public static <T> Mono<T> defer(Supplier<? extends Mono<? extends T>> supplier)    懒汉型，比较懒，每次调用才创建  // 
Mono<Date> m2 = Mono.defer(() -> Mono.just(new Date()));

public final void subscribe(Subscriber<? super T> actual)    传入内部值执行函数
```

搭配 actuator

springdoc.show-actuator=true

swagger2 配置项

swagger2 依赖:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-beanValidators</artifactId>      <!-- 验证插件 JSR303 规范与数据校验 jdk 中的 JSR303 规范
(javax.validation 包) -->
    <version>2.9.2</version>
</dependency>
```

版本搭配: knif4j-spring-boot-starter 2.0.2 = springfox-swagger2 2.6.1

knif4j-spring-boot-starter 2.0.5 = springfox-swagger2 2.9.2

ui 插件: swagger-ui-layer

<http://localhost:8099/docs.html>

knife4j-spring-ui = com.github.xiaoymin

<http://localhost:8099/doc.html>

knife4j-spring-boot-starter (knif4j 增强版)

springfox-swagger-ui = io.springfox

<http://localhost:8099/swagger-ui.html>

功能接口: /api-docs /swagger-resources

ERROR: 父版本覆盖了, 依赖中的 swagger 版本

SwaggerConfig (swagger2) >>

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket createRestApi() {    // 第一个分组 bean
        return new Docket(DocumentationType.SWAGGER_2) // 文档类型
            .groupName("aaaaa")          // 分组名
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.testjpa.controller")) // 为当前包生成接口文档
            .paths(PathSelectors.any())
    }
}
```

```

        .build();
    }

    @Bean
    public Docket createRestApi2() { // 第二个分组 bean
        return new Docket(DocumentationType.SWAGGER_2) // 文档类型
            .groupName("bbbbbb") // 分组名
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.testjpa.controller")) // 为当前包生成接口文档
            .paths(PathSelectors.any())
            .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("Spring Boot 中使用 Swagger2 构建 RESTful APIs") // 文档标题
            .description("更多请关注 http://www.baidu.com")
            .version("1.0")

            .termsOfServiceUrl("http://www.baidu.com")
            .build();
    }
}

```

swagger3 配置项 (推荐使用 springdoc-openapi-ui)

swagger3 兼容：SpringBoot 2.2+

Springfox 路径匹配基于 AntPathMatcher，而 SpringBoot2.6.X 使用的是 PathPatternMatcher

swagger3 依赖：

```

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>

```

ERROR：不能和 springdoc-openapi-ui 一起使用

SwaggerConfig (swagger3) >>

```

@Configuration
@EnableOpenApi
@Profile({"prod", "test", "bluetooth-app"})
public class SwaggerConfig implements WebMvcConfigurer {

    @Bean
    public Docket docket(){
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .enable(true)
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.saidake.generator.controller"))
            .paths(PathSelectors.any())
            .build();
    }
}

```

```

private ApiInfo apiInfo(){
    return new ApiInfoBuilder()
        .title("XX 项目接口文档")
        .description("XX 项目描述")
        .contact(new Contact("作者", "作者 URL", "作者 Email"))
        .version("1.0")
        .build();
}

@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/test/**").addResourceLocations("classpath:/META-INF/resources/");
}

@Override
public void addViewControllers(ViewControllerRegistry registry){
    registry.addViewController("/test/swagger-resources/configuration/ui").setViewName("forward:/swagger-
resources/configuration/ui");           //设置访问路径为 "/" 跳转到指定页面
    registry.addViewController("/test/swagger-resources").setViewName("forward:/swagger-resources");   //设置访问路径为
"/" 跳转到指定页面
    registry.addViewController("/test/v2/api-docs").setViewName("forward:/v2/api-docs");                  //设置访问路径为
"/" 跳转到指定页面
    registry.addViewController("/test/swagger").setViewName("forward:/test/doc.html");
}
}

application.yml >>
spring:
  mvc: # 这个 mvc 的配置是 springboot2.6.1 不支持 swagger3 的折衷配置, 后面考虑升级 Springboot 版本或降级版本
    pathmatch:
      matching-strategy: ant_path_matcher

```

[swagger-annotations]

```

package io.swagger.annotations;

@Api          类生成注释
  value="账号"
  tags="接口标题",
  description = "描述"      【接口标题  描述 >】
  authorizations = {@Authorization(value="access_token")})  添加在 Docket 内 securitySchemes 配置的请求头

@ApiOperation  方法生成注释
  tags="列表",
  value="接口说明"
  notes = "展开补充",
  response = ListShippingModel.class           //  /v1/api/file/getFileListByPage 接口说明      展开补充 (根据"
  列表" 新分类的接口标题)

```

@ApiParam 函数参数说明【PARAMETER, METHOD, FIELD】

`example = "2021-09-24 00:00:00"` 示例值

`value = "柜号"` 参数说明

`required=true` 参数是否必须传

@ApiImplicitParams 用在方法上包含一组参数说明

@ApiImplicitParam: 用来注解来给方法入参增加说明。

`name="username"` 参数名

`value="用户名"` 参数的汉字说明、解释

`required=true` 参数是否必须传

`paramType="body"` 参数放在哪个地方

· `header` --> 请求参数的获取: `@RequestHeader`

· `query` --> 请求参数的获取: `@RequestParam`

· `path` (用于 restful 接口) --> 请求参数的获取: `@PathVariable`

· `body` (不常用)

· `form` (不常用)

`dataType`: 参数类型, 默认 `String`, 其它值 `dataType="Integer"`

`defaultValue`: 参数的默认值

@ApiResponse: 用于表示一组响应

@ApiResponse: 用在`@ApiResponses` 中, 一般用于表达一个错误的响应信息

`code="500"` 响应码

`message="【自定义 code 响应码】请求参数错误"` 错误信息

`response`: 抛出异常的类

@ApiIgnore 忽略此拦截

@ApiModelProperty(subTypes = {Port.class}) 表名

@ApiModelProperty(value = "是否整柜进口, 分为整柜出口和整柜进口, 默认 false 为整柜出口", example = "false") 表字段

[springfox-swagger-common]

package springfox.documentation.swagger.web;

public interface **SwaggerResourcesProvider** extends Supplier<List<SwaggerResource>> {} 模板接口

[springfox-core]

package springfox.documentation.builders;

public class **ApiInfoBuilder** api 信息构建器

public ApiInfoBuilder **title**(String title) 标题

public ApiInfoBuilder **description**(String description) 描述

public ApiInfoBuilder **contact**(Contact contact) 联系人

public ApiInfoBuilder **version**(String version) 版本

public ApiInfo **build**() 创建一个 new ApiInfo

package springfox.documentation.builders;

public class **RequestHandlerSelectors** 请求选择器

public static Predicate<RequestHandler> **basePackage**(String basePackage)

package springfox.documentation.builders;

public class **PathSelectors** 路径选择器

public static Predicate<String> **any()** 所有通过

[springfox-spi]

```
package springfox.documentation.spi;
public class DocumentationType extends SimplePluginMetadata 文档类型
public static final DocumentationType SWAGGER_12 = new DocumentationType("swagger", "1.2");
public static final DocumentationType SWAGGER_2 = new DocumentationType("swagger", "2.0");
public static final DocumentationType OAS_30 = new DocumentationType("openApi", "3.0");
```

[springfox-spring-web]

```
package springfox.documentation.spring.web.plugins;
public class Docket implements DocumentationPlugin 文档摘要
public Docket(DocumentationType documentationType)
public Docket apiInfo(ApiInfo apiInfo) api 信息
public Docket enable(boolean externallyConfiguredFlag) 是否启用
public ApiSelectorBuilder select() 返回 api 选择器
public ApiSelectorBuilder apis(Predicate<RequestHandler> selector) 选择的 controller api
package springfox.documentation.spring.web.plugins;
public class ApiSelectorBuilder api 选择器
public Docket build() 构建一个摘要
```

[springfox-swagger-core]

```
package springfox.documentation.service;
public class ApiKey extends SecurityScheme api 权限验证
public ApiKey(String name, String keyname, String passAs) name 为参数名 keyname 是页面传值显示的 keyname, name 在 swagger 鉴权中使用
```

网关集成标准配置项

SwaggerHandler >>

```
@RestController
@RequestMapping("/swagger-resources" )
public class SwaggerHandler {
    private final SwaggerResourcesProvider swaggerResources;

    @Autowired(required = false)
    private SecurityConfiguration securityConfiguration;

    @Autowired(required = false)
    private UiConfiguration uiConfiguration;

    @Autowired
    public SwaggerHandler(SwaggerResourcesProvider swaggerResources) {
        this.swaggerResources = swaggerResources;
    }

    // Swagger 安全配置, 支持 oauth 和 apiKey 设置
    @GetMapping("/configuration/security" )
    public Mono<ResponseEntity<SecurityConfiguration>> securityConfiguration() {
        return Mono.just(new ResponseEntity<>(
            Optional.ofNullable(securityConfiguration).orElse(SecurityConfigurationBuilder.builder().build()), HttpStatus.OK));
    }
}
```

```

// Swagger UI 配置
@GetMapping("/configuration/ui" )
public Mono<ResponseEntity<UiConfiguration>> uiConfiguration() {
    return Mono.just(new ResponseEntity<>(
        Optional.ofNullable(uiConfiguration).orElse(UiConfigurationBuilder.builder().build()), HttpStatus.OK));
}

// Swagger 资源配置，微服务中这各个服务的 api-docs 信息
@GetMapping
public Mono<ResponseEntity> swaggerResources() {
    return Mono.just((new ResponseEntity<>(swaggerResources.get(), HttpStatus.OK)));
}
}

```

SwaggerResourceConfig >>

```

@Component
@Primary
@EnableConfigurationProperties(SwaggerAggProperties.class)
public class SwaggerResourceConfig implements SwaggerResourcesProvider { // 此类定义了 swagger 网关层的开放接口，在访问 swagger-ui 中会拉去此接口的数据。
    private final RouteLocator routeLocator;
    private final GatewayProperties gatewayProperties;

    @Resource
    private SwaggerAggProperties swaggerAggProperties;

    public SwaggerResourceConfig(RouteLocator routeLocator, GatewayProperties gatewayProperties) {
        this.routeLocator = routeLocator;
        this.gatewayProperties = gatewayProperties;
    }

    @Override
    public List<SwaggerResource> get() {
        List<SwaggerResource> resources = new ArrayList<>();
        Set<String> routes = new HashSet<>();

        routeLocator.getRoutes().subscribe(route -> routes.add(route.getId())); // 取出 Spring Cloud Gateway 中的 route

        gatewayProperties.getRoutes().stream().filter( // 结合 application.yml 中的路由配置，只获取有效的
route 节点
            routeDefinition -> (
                routes.contains(routeDefinition.getId()) && swaggerAggProperties.isShow(routeDefinition.getId())
            )
        ).forEach(routeDefinition -> routeDefinition.getPredicates().stream()
            .filter(predicateDefinition -> ("Path").equalsIgnoreCase(predicateDefinition.getName()))
            .forEach(predicateDefinition -> resources.add(
                swaggerResource(
                    routeDefinition.getId(),

```

```

        predicateDefinition.getArgs().get(NameUtils.GENERATED_NAME_PREFIX + "0").replace("/**",
swaggerAggProperties.getApiDocsPath())
    )
)
)
);
return resources;
}

private SwaggerResource swaggerResource(String name, String location) {
    SwaggerResource swaggerResource = new SwaggerResource();
    swaggerResource.setName(name);
    swaggerResource.setLocation(location);
    swaggerResource.setSwaggerVersion(swaggerAggProperties.getSwaggerVersion());
    return swaggerResource;
}
}
}

```

knif4j 标准配置项

SwaggerConfig (knife-spring-boot-starter) >>

```

@Configuration
@EnableSwagger2
@Import(BeanValidatorPluginsConfiguration.class)
public class SwaggerConfig {

    @Bean
    public Docket createWMS() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .groupName("AQPG")
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.dhc.traffi.controller")) // 扫描的路径包
            .paths(PathSelectors.any()) // 处理 指定的所有路径
            .build()
            .securitySchemes(securitySchemes()) // 配置一些接口 登录后才能访问及操作
            .securityContexts(securityContexts());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("API 接口") // title 设置文档的标题
            .description ("Rest 接口测试") // description 设置文档的描述
            .contact(new Contact("", "", "")) // contact 设置文档的联系方式
            .termsOfServiceUrl("") // termsOfServiceUrl 设置文档的 License 信息
            .version("1.0") // version 设置文档的版本信息
            .build();
    }
}

```

knife4j 微服务整合

依赖: `knife4j-spring-boot-starter` = com.github.xiaoymin (前端 UI 模块, 只需要在网关服务处引入, 为每个服务都生成 API 文档)

```
<dependency>
    <groupId>com.github.xiaoymin</groupId>
    <artifactId>knife4j-spring-boot-starter</artifactId>
</dependency>
```

`knife4j-micro-spring-boot-starter` = com.github.xiaoymin (后端的 Jar 包, 每个服务都需要引入)

knife4j 微服 前端 UI 网关配置项 (gateway 引入)

`application.yml` >> 网关配置

aftersale:

swagger-agg:

generate-routes: system 根据是否包含 gateway 配置的的 routeid 生成文档

`SwaggerResourceConfig` >> 在网关上添加 Swagger 资源配置, 用于聚合其他微服务中 Swagger 的 api-docs 访问路径;

```
@Component
@EnableConfigurationProperties(SwaggerAggProperties.class)
@Primary
public class SwaggerResourceConfig implements SwaggerResourcesProvider {
    private final RouteLocator routeLocator;
    private final GatewayProperties gatewayProperties;

    @Resource
    private SwaggerAggProperties swaggerAggProperties;

    public SwaggerResourceConfig(RouteLocator routeLocator, GatewayProperties gatewayProperties) {
        this.routeLocator = routeLocator;
        this.gatewayProperties = gatewayProperties;
    }

    @Override
    public List<SwaggerResource> get() {
        List<SwaggerResource> resources = new ArrayList<>();
        Set<String> routes = new HashSet<>();

        routeLocator.getRoutes().subscribe(route -> routes.add(route.getId())); //取出 Spring Cloud Gateway 中的 route
        gatewayProperties.getRoutes().stream().filter //结合 application.yml 中的路由配置, 只获取有效的
route 节点
            routeDefinition -> (
                routes.contains(routeDefinition.getId()) && swaggerAggProperties.isShow(routeDefinition.getId())
            )
        ).forEach(routeDefinition -> routeDefinition.getPredicates().stream()
            .filter(predicateDefinition -> ("Path").equalsIgnoreCase(predicateDefinition.getName()))
        ).forEach(predicateDefinition -> resources.add(
            swaggerResource(
                routeDefinition.getId(),
                predicateDefinition.getArgs().get(NameUtils.GENERATED_NAME_PREFIX + "0").replace("/**",
                    swaggerAggProperties.getApiDocsPath())
            )
        ))
    }
}
```

```

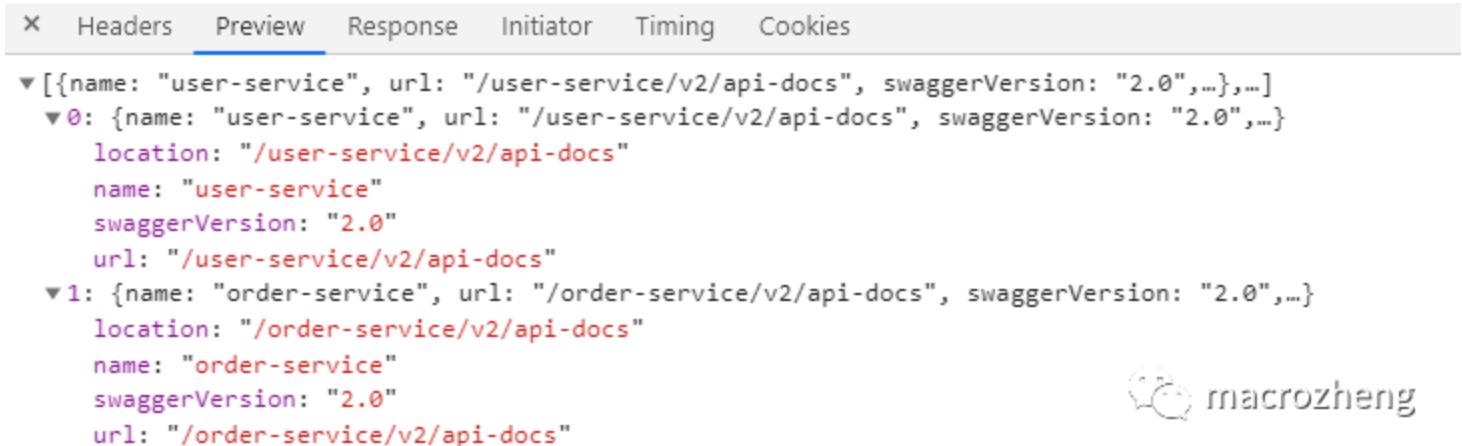
        )
    )
)
);
return resources;
}

private SwaggerResource swaggerResource(String name, String location) {
    SwaggerResource swaggerResource = new SwaggerResource();
    swaggerResource.setName(name);
    swaggerResource.setLocation(location);
    swaggerResource.setSwaggerVersion(swaggerAggProperties.getSwaggerVersion());
    return swaggerResource;
}
}

```

SwaggerHandler >> 自定义 Swagger 各个配置的节点，简单来说就是自定义 Swagger 内部的各个获取数据的接口

<http://localhost:9201/user-service/v2/api-docs> ==> <http://localhost:9201/swagger-resources>



```

x Headers Preview Response Initiator Timing Cookies
▼ [ {name: "user-service", url: "/user-service/v2/api-docs", swaggerVersion: "2.0", ...}, ... ]
  ▼ 0: {name: "user-service", url: "/user-service/v2/api-docs", swaggerVersion: "2.0", ...}
    location: "/user-service/v2/api-docs"
    name: "user-service"
    swaggerVersion: "2.0"
    url: "/user-service/v2/api-docs"
  ▼ 1: {name: "order-service", url: "/order-service/v2/api-docs", swaggerVersion: "2.0", ...}
    location: "/order-service/v2/api-docs"
    name: "order-service"
    swaggerVersion: "2.0"
    url: "/order-service/v2/api-docs"

```



```

@RestController
@RequestMapping("/swagger-resources")
public class SwaggerHandler {
    private final SwaggerResourcesProvider swaggerResources;

    @Autowired(required = false)
    private SecurityConfiguration securityConfiguration;

    @Autowired(required = false)
    private UiConfiguration uiConfiguration;

    @Autowired
    public SwaggerHandler(SwaggerResourcesProvider swaggerResources) {
        this.swaggerResources = swaggerResources;
    }

    // Swagger 安全配置，支持 oauth 和 apiKey 设置
    @GetMapping("/configuration/security")
    public Mono< ResponseEntity< SecurityConfiguration >> securityConfiguration() {

```

```

        return Mono.just(new ResponseEntity<>(
            Optional.ofNullable(securityConfiguration).orElse(SecurityConfigurationBuilder.builder().build()), HttpStatus.OK));
    }

    // Swagger UI 配置
    @GetMapping("/configuration/ui" )
    public Mono<ResponseEntity<UiConfiguration>> uiConfiguration() {
        return Mono.just(new ResponseEntity<>(
            Optional.ofNullable(uiConfiguration).orElse(UiConfigurationBuilder.builder().build()), HttpStatus.OK));
    }

    // Swagger 资源配置，微服务中这各个服务的 api-docs 信息
    @GetMapping
    public Mono< ResponseEntity> swaggerResources() {
        return Mono.just((new ResponseEntity<>(swaggerResources.get(), HttpStatus.OK)));
    }
}

```

knife4j 微服 swagger 后端服务配置项 (service 引入)

application.yml >> service 配置

```

aftersale:
  swagger:
    enabled: true
    title: 认证中心
    description: 认证中心接口文档
    version: 1.0
    base-package: com.central.oauth.controller

```

SwaggerAutoConfiguration >>

```

package com.saidake.common.swagger;

import com.github.xiaoymin.knife4j.spring.annotations.EnableKnife4j;
import com.google.common.base.Predicate;
import com.google.common.base.Predicates;
import com.google.common.collect.Lists;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import springfox.bean.validators.configuration.BeanValidatorPluginsConfiguration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.ParameterBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.schema.ModelRef;

```

```
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.ApiKey;
import springfox.documentation.service.AuthorizationScope;
import springfox.documentation.service.Contact;
import springfox.documentation.service.Parameter;
import springfox.documentation.service.SecurityReference;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spi.service.contexts.SecurityContext;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;
import java.util.Set;
import java.util.stream.Collectors;

@Configuration
@EnableSwagger2
@EnableKnife4j
@Import(BeanValidatorPluginsConfiguration.class)
public class SwaggerAutoConfiguration implements BeanFactoryAware {
    private static final String AUTH_KEY = "Authorization";

    private BeanFactory beanFactory;

    @Bean
    @ConditionalOnMissingBean
    public SwaggerProperties swaggerProperties() {
        return new SwaggerProperties();
    }

    @Bean
    @ConditionalOnMissingBean
    @ConditionalOnProperty(name = "zlt.swagger.enabled", matchIfMissing = true)
    public List<Docket> createRestApi(SwaggerProperties swaggerProperties) {
        ConfigurableBeanFactory configurableBeanFactory = (ConfigurableBeanFactory) beanFactory;
        List<Docket> docketList = new LinkedList<>();

        // 没有分组
        if (swaggerProperties.getDocket().size() == 0) {
            final Docket docket = createDocket(swaggerProperties);
            configurableBeanFactory.registerSingleton("defaultDocket", docket);
            docketList.add(docket);
            return docketList;
        }
    }
}
```

```

// 分组创建
for (String groupName : swaggerProperties.getDocket().keySet()) {
    SwaggerProperties.DocketInfo docketInfo = swaggerProperties.getDocket().get(groupName);

    ApiInfo apiInfo = new ApiInfoBuilder()
        .title(docketInfo.getTitle().isEmpty() ? swaggerProperties.getTitle() : docketInfo.getTitle())
        .description(docketInfo.getDescription().isEmpty() ? swaggerProperties.getDescription() :
docketInfo.getDescription())
        .version(docketInfo.getVersion().isEmpty() ? swaggerProperties.getVersion() : docketInfo.getVersion())
        .license(docketInfo.getLicense().isEmpty() ? swaggerProperties.getLicense() : docketInfo.getLicense())
        .licenseUrl(docketInfo.getLicenseUrl().isEmpty() ? swaggerProperties.getLicenseUrl() : docketInfo.getLicenseUrl())
        .contact(
            new Contact(
                docketInfo.getContact().getName().isEmpty() ? swaggerProperties.getContact().getName() :
docketInfo.getContact().getName(),
                docketInfo.getContact().getUrl().isEmpty() ? swaggerProperties.getContact().getUrl() :
docketInfo.getContact().getUrl(),
                docketInfo.getContact().getEmail().isEmpty() ? swaggerProperties.getContact().getEmail() :
docketInfo.getContact().getEmail()
            )
        )
        .termsOfServiceUrl(docketInfo.getTermsOfServiceUrl().isEmpty() ? swaggerProperties.getTermsOfServiceUrl() :
docketInfo.getTermsOfServiceUrl())
    .build();

    // base-path 处理
    // 当没有配置任何 path 的时候，解析/**
    if (docketInfo.getBasePath().isEmpty()) {
        docketInfo.getBasePath().add("/");
    }
    List<Predicate<String>> basePath = new ArrayList<>(docketInfo.getBasePath().size());
    for (String path : docketInfo.getBasePath()) {
        basePath.add(PathSelectors.ant(path));
    }

    // exclude-path 处理
    List<Predicate<String>> excludePath = new ArrayList<>(docketInfo.getExcludePath().size());
    for (String path : docketInfo.getExcludePath()) {
        excludePath.add(PathSelectors.ant(path));
    }

    Docket docket = new Docket(DocumentationType.SWAGGER_2)
        .host(swaggerProperties.getHost())
        .apiInfo(apiInfo)
        .globalOperationParameters(assemblyGlobalOperationParameters(swaggerProperties.getGlobalOperationParam
eters()),
            docketInfo.getGlobalOperationParameters()))
        .groupName(groupName)

```

```
.select()
.apis(RequestHandlerSelectors.basePackage(docketInfo.getBasePackage()))
.paths(
    Predicates.and(
        Predicates.not(Predicates.or(excludePath)),
        Predicates.or(basePath)
    )
)
.build()
.securitySchemes(securitySchemes())
.securityContexts(securityContexts());

configurableBeanFactory.registerSingleton(groupName, docket);
docketList.add(docket);
}

return docketList;
}

/***
 * 创建 Docket 对象
 *
 * @param swaggerProperties swagger 配置
 * @return Docket
 */
private Docket createDocket(final SwaggerProperties swaggerProperties) {
    ApiInfo apiInfo = new ApiInfoBuilder()
        .title(swaggerProperties.getTitle())
        .description(swaggerProperties.getDescription())
        .version(swaggerProperties.getVersion())
        .license(swaggerProperties.getLicense())
        .licenseUrl(swaggerProperties.getLicenseUrl())
        .contact(new Contact(swaggerProperties.getContact().getName(),
            swaggerProperties.getContact().getUrl(),
            swaggerProperties.getContact().getEmail()))
        .termsOfServiceUrl(swaggerProperties.getTermsOfServiceUrl())
        .build();

    // base-path 处理
    // 当没有配置任何 path 的时候，解析 /**
    if (swaggerProperties.getBasePath().isEmpty()) {
        swaggerProperties.getBasePath().add("**");
    }

    List<Predicate<String>> basePath = new ArrayList<>();
    for (String path : swaggerProperties.getBasePath()) {
        basePath.add(PathSelectors.ant(path));
    }

    // exclude-path 处理
```

```

List<Predicate<String>> excludePath = new ArrayList<>();
for (String path : swaggerProperties.getExcludePath()) {
    excludePath.add(PathSelectors.ant(path));
}

return new Docket(DocumentationType.SWAGGER_2)
    .host(swaggerProperties.getHost())
    .apiInfo(apiInfo)
    .globalOperationParameters(buildGlobalOperationParametersFromSwaggerProperties(
        swaggerProperties.getGlobalOperationParameters()))
    .select()
    .apis(RequestHandlerSelectors.basePackage(swaggerProperties.getBasePackage()))
    .paths(
        Predicates.and(
            Predicates.not(Predicates.or(excludePath)),
            Predicates.or(basePath)
        )
    )
    .build()
    .securitySchemes(securitySchemes())
    .securityContexts(securityContexts());
}

@Override
public void setBeanFactory(BeanFactory beanFactory) {
    this.beanFactory = beanFactory;
}

private List<SecurityContext> securityContexts() {
    List<SecurityContext> contexts = new ArrayList<>(1);
    SecurityContext securityContext = SecurityContext.builder()
        .securityReferences(defaultAuth())
        //.forPaths(PathSelectors.regex("^^(?!auth).*$"))
        .build();
    contexts.add(securityContext);
    return contexts;
}

private List<SecurityReference> defaultAuth() {
    AuthorizationScope authorizationScope = new AuthorizationScope("global", "accessEverything");
    AuthorizationScope[] authorizationScopes = new AuthorizationScope[1];
    authorizationScopes[0] = authorizationScope;
    List<SecurityReference> references = new ArrayList<>(1);
    references.add(new SecurityReference(AUTH_KEY, authorizationScopes));
    return references;
}

private List<ApiKey> securitySchemes() {

```

```
List<ApiKey> apiKeyList = new ArrayList<>(1);
ApiKey apiKey = new ApiKey(AUTH_KEY, AUTH_KEY, "header");
apiKeyList.add(apiKey);
return apiKeyList;
}

private List<Parameter> buildGlobalOperationParametersFromSwaggerProperties(
    List<SwaggerProperties.GlobalOperationParameter> globalOperationParameters) {
    List<Parameter> parameters = Lists.newArrayList();

    if (Objects.isNull(globalOperationParameters)) {
        return parameters;
    }
    for (SwaggerProperties.GlobalOperationParameter globalOperationParameter : globalOperationParameters) {
        parameters.add(new ParameterBuilder()
            .name(globalOperationParameter.getName())
            .description(globalOperationParameter.getDescription())
            .modelRef(new ModelRef(globalOperationParameter.getModelRef()))
            .parameterType(globalOperationParameter.getParameterType())
            .required(Boolean.parseBoolean(globalOperationParameter.getRequired()))
            .build());
    }
    return parameters;
}

/**
 * 局部参数按照 name 覆盖局部参数
 *
 * @param globalOperationParameters
 * @param docketOperationParameters
 * @return
 */
private List<Parameter> assemblyGlobalOperationParameters(
    List<SwaggerProperties.GlobalOperationParameter> globalOperationParameters,
    List<SwaggerProperties.GlobalOperationParameter> docketOperationParameters) {
    if (Objects.isNull(docketOperationParameters) || docketOperationParameters.isEmpty()) {
        return buildGlobalOperationParametersFromSwaggerProperties(globalOperationParameters);
    }

    Set<String> docketNames = docketOperationParameters.stream()
        .map(SwaggerProperties.GlobalOperationParameter::getName)
        .collect(Collectors.toSet());

    List<SwaggerProperties.GlobalOperationParameter> resultOperationParameters = Lists.newArrayList();
    if (Objects.nonNull(globalOperationParameters)) {
        for (SwaggerProperties.GlobalOperationParameter parameter : globalOperationParameters) {
            if (!docketNames.contains(parameter.getName())) {
                resultOperationParameters.add(parameter);
            }
        }
    }
    resultOperationParameters.addAll(docketOperationParameters);
    return resultOperationParameters;
}
```

```

        if (!docketNames.contains(parameter.getName())) {
            resultOperationParameters.add(parameter);
        }
    }

    resultOperationParameters.addAll(docketOperationParameters);
    return buildGlobalOperationParametersFromSwaggerProperties(resultOperationParameters);
}
}

```

SwaggerProperties >>

```

package com.saidake.common.swagger;

import lombok.Data;
import lombok.Getter;
import lombok.Setter;
import org.springframework.boot.context.properties.ConfigurationProperties;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

@Data
@ConfigurationProperties("aftersale.swagger")
public class SwaggerProperties {

    /**是否开启 swagger*/
    private Boolean enabled;
    /**标题*/
    private String title = "";
    /**描述*/
    private String description = "";
    /**版本*/
    private String version = "";
    /**许可证*/
    private String license = "";
    /**许可证 URL*/
    private String licenseUrl = "";
    /**服务条款 URL*/
    private String termsOfServiceUrl = "";
    /**联系人*/
    private Contact contact = new Contact();
    /**swagger 会解析的包路径*/
    private String basePackage = "";

    /**swagger 会解析的 url 规则*/
    private List<String> basePath = new ArrayList<>();
}

```

```
/**在 basePath 基础上需要排除的 url 规则*/
private List<String> excludePath = new ArrayList<>();

/**分组文档*/
private Map<String, DocketInfo> docket = new LinkedHashMap<>();

/**host 信息*/
private String host = "";

/**全局参数配置*/
private List<GlobalOperationParameter> globalOperationParameters;
```

```
@Setter
@Getter
public static class GlobalOperationParameter{
    /**参数名*/
    private String name;

    /**描述信息*/
    private String description;

    /**指定参数类型*/
    private String modelRef;

    /**参数放在哪个地方:header,query,path,body,form*/
    private String parameterType;

    /**参数是否必须传*/
    private String required;
}
```

```
@Data
public static class DocketInfo {
    /**标题*/
    private String title = "";
    /**描述*/
    private String description = "";
    /**版本*/
    private String version = "";
    /**许可证*/
    private String license = "";
    /**许可证 URL*/
    private String licenseUrl = "";
    /**服务条款 URL*/
    private String serviceUrl = "";
```

```

private String termsOfServiceUrl = "";

private Contact contact = new Contact();

/**swagger 会解析的包路径*/
private String basePackage = "";

/**swagger 会解析的 url 规则*/
private List<String> basePath = new ArrayList<>();
/**在 basePath 基础上需要排除的 url 规则*/
private List<String> excludePath = new ArrayList<>();

private List<GlobalOperationParameter> globalOperationParameters;
}

@Data
public static class Contact {
    /**联系人*/
    private String name = "";
    /**联系人 url*/
    private String url = "";
    /**联系人 email*/
    private String email = "";
}

}

```

sprongdoc-openapi-ui

介绍

官网: <https://springdoc.org/v2/#getting-started>

依赖:

```

<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.9</version>
</dependency>

```

依赖: spring-boot v3

```

<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.4</version>
</dependency>

```

[springdoc-openapi-ui]

@Tag 接口类【@Api】

name = 'xxclass' 名称, 必选字段

@Operation 接口方法【@ApiOperation】

summary = "foo" 名称, 必选字段

```

description = "bar"    描述
hidden = true      隐藏, 相当于@Apilgnore
responses={ @ApiResponse(...) }
security=@SecurityRequirement(...)

@Parameter 【@ApiParam @ApilmplicitParam】
hidden = true      隐藏, 相当于@Apilgnore

@Parameters 【@ApilmplicitParams】

@ApiResponse 【@ApiResponse】
responseCode="404"
description="foo"
content=@Content(...)

@Content
mediaType="application/json",
schema=@Schema(implementation=Store.class)
array = @ArraySchema(schema = @Schema(implementation = User.class)) 数组类型的 schema

@SecurityRequirement
name="need auth"

```

@Schema 模型 【@ApiModel @ApiModelProperty】

title="modelName" 类名上为必选项

name="test_str" 名称
accessModel = READ_ONLY 相当于 hidden=true, 隐藏字段
maxLength=5 最大程度
maximum="10000" 最大值
minimum="1" 最小值
type="string" 重定义类型, 便于显示 example, 针对 Date 等格式的数据

String ref() default ""; // "#/components/schemas/user"

Class<?> implementation() default Void.class;
Class<?> not() default Void.class;
Class<?>[] oneOf() default {};
Class<?>[] anyOf() default {};
Class<?>[] allOf() default {};
Class<?>[] subTypes() default {};

配置

application.yml >>

```

springdoc:
show-actuator=true      显示 actuator 监控请求信息
swagger-ui:
path: /swagger           # 自定义的文档界面访问路径。默认访问路径是/swagger-ui.html
doc-expansion: none      # 字符串类型, 一共三个值来控制操作和标记的默认展开设置。它可以是"list" (仅展开标记)、
"full" (展开标记和操作) 或"none" (不展开任何内容)。
displayRequestDuration: true # 布尔值。控制"试用"请求的请求持续时间 (毫秒) 的显示。
showExtensions: true      # 布尔值。控制供应商扩展 (x-) 字段和操作、参数和架构值的显示。

```

```

showCommonExtensions: true    # 布尔值。控制参数的扩展名 (pattern、maxLength、minLength、maximum、
minminimum) 字段和值的显示。
disable-swagger-default-url: true    # 布尔值。禁用 swagger 用户界面默认 petstore url。(从 v1.4.1 开始提供)。
api-docs:
  enabled: true      # enabled the /v3/api-docs endpoint
  path: /swagger-ui/api-docs          # 自定义的文档 api 元数据访问路径。默认访问路径是/v3/api-docs
  resolve-schema-properties: true     # 布尔值。在@Schema (名称 name、标题 title 和说明 description, 三个属性) 上启
用属性解析程序。
  writer-with-default-pretty-printer: true   # 布尔值。实现 OpenAPI 规范的打印。
# ===== swagger 配置 =====#
swagger:
  application-name: ${spring.application.name}
  application-version: 1.0
  application-description: springdoc openapi 整合 Demo
  try-host: http://localhost:${server.port}

```

SpringdocOpenapiConfig >>

```

import io.swagger.v3.oas.annotations.enums.ParameterIn;
import io.swagger.v3.oas.models.Components;
import io.swagger.v3.oas.models.ExternalDocumentation;
import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.headers.Header;
import io.swagger.v3.oas.models.info.Info;
import io.swagger.v3.oas.models.info.License;
import io.swagger.v3.oas.models.media.StringSchema;
import io.swagger.v3.oas.models.parameters.HeaderParameter;
import io.swagger.v3.oas.models.parameters.Parameter;
import io.swagger.v3.oas.models.security.SecurityScheme;
import io.swagger.v3.oas.models.servers.Server;
import org.apache.commons.lang3.reflect.FieldUtils;
import org.springdoc.core.SwaggerUiConfigProperties;
import org.springdoc.core.customizers.OpenApiCustomiser;
import org.springframework.boot.SpringBootVersion;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.util.ReflectionUtils;
import org.springframework.web.servlet.config.annotation.InterceptorRegistration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Configuration
public class SpringdocOpenapiConfig implements WebMvcConfigurer {
  @Bean
  public GroupedOpenApi actuatorApi() { //添加 actuator 请求

```

```
        return GroupedOpenApi.builder().group("Actuator")
            .pathsToMatch("/actuator/**")
            .build();
    }

@Bean
public GroupedOpenApi oracleApi() {      //添加普通请求
    return GroupedOpenApi.builder()
        .group("AAA")
        .pathsToMatch("/**")
        .pathsToExclude("!/actuator/**")
        .build();
}

@Bean
public GroupedOpenApi actuatorApi() {      //添加 actuator 请求
    return GroupedOpenApi.builder().group("Actuator")
        .pathsToMatch("/actuator/**")
        .build();
}

@Bean
public OpenAPI springDocOpenAPI() {
    Components components = new Components();      //配置认证、请求头参数
    components
        .addSecuritySchemes("sdk-key", new
SecurityScheme().type(SecurityScheme.Type.HTTP).scheme("bearer").bearerFormat("JWT")) // spring security 认证选项
    WebSecurityConfigurerAdapter
        .addSecuritySchemes("sdk-scheme", new SecurityScheme().type(SecurityScheme.Type.HTTP).scheme("basic"));
        .addParameters("myHeader111", new Parameter().in("header").schema(new StringSchema()).name("myHeader1"));
        .addParameters("myHeader111", new Parameter().in(ParameterIn.HEADER.toString()).name("Custom-Header-
Version"))
        .description("jwt token").schema(new StringSchema()).example("v1").required(false));
        .addParameters("myHeader1", new Parameter().in("header").schema(new StringSchema()).name("myHeader1"));
        .addHeaders("myHeader2", new Header().description("myHeader2 header").schema(new StringSchema()));
        .addHeaders("myHeader2", new Header().description("myHeader2 header").schema(new
StringSchema()).extensions(myHeader2extensions));
        .addParameters("myGlobalHeader", new HeaderParameter().required(true).name("My-Global-
Header").description("My Global Header").schema(new StringSchema()).required(false));
    return new OpenAPI()
        .components(components)
        .servers(Collections.singletonList(new Server(){setUrl("");})) // 接口调试路径
        .info(new Info()
            .title("Api Doc")
            .description("description")
            .version("Spring Boot1 Version: " + SpringBootVersion.getVersion())
            .license(new License().name("Apache 2.0").url("https://www.apache.org/licenses/LICENSE-2.0.html")));
}
```

```

)
    .externalDocs(new ExternalDocumentation()
        .description("SpringDoc Full Documentation")
        .url("https://springdoc.org/"))
);
}

/**
 * 添加全局的请求头参数
 */
@Bean
public OpenApiCustomiser customerGlobalHeaderOpenApiCustomiser() {
    return openApi -> openApi.getPaths().values().stream().flatMap(pathItem ->
pathItem.readOperations().stream())
    .forEach(operation -> {
        Parameter customHeaderVersion = new
Parameter().in(ParameterIn.HEADER.toString()).name("Custom-Header-Version")
            .description("jwt token").schema(new StringSchema()).example("v1").required(false);
        operation.addParametersItem(customHeaderVersion);
    });
}
}

```

1.7

1.7-beta1

1.6.4

1.6.3

1.6.2

1.6.1

1.6

1.6-beta2

1.6-beta1

1.5

velocity

基于 java 的模板引擎，提供 vm 文件支持

依赖：velocity = org.apache.velocity

常用版本：1.7

thymeleaf

```

application.yml >
spring:
  thymeleaf:
    prefix: classpath:/templates/
    cache: false
config >
import org.springframework.ui.Model;
@GetMapping("/index")
public String showPage(Model model) {
    model.addAttribute("message", "HelloWorld");
    return "index";
}

```

变量取值\${ a } 选择变量 *{ a } 消息 #{ a } 链接 @{ a } 片段表达式 ~{ a }

webflux

依赖：spring-webflux

webflux：SpringWebflux 是 Spring5 中添加的新模块，用于 web 开发，功能和 SpringMVC 相似，Webflux 使用的是响应式编程方式

Webflux 是一种异步非阻塞的框架（非阻塞和阻塞、异步和同步）

异步和同步是针对调用者的，调用者发送请求。如果调用者等待对方回应之后才去做其他事情，就是同步；如果发送请求之后不等待对方的回应，而是接着做其他的事情，就是异步。

阻塞和非阻塞是针对被调用者的，被调用者在接收到请求之后，在完成请求任务之后才给出反馈的，就是阻塞；如果收到

请求之后先给出反馈，然后再去完成请求任务的，就是非阻塞
函数式编程：webflux 使用函数式编程方式实现路由请求
非阻塞式：在有限资源下，提高系统吞吐量和伸缩性，以 Reactor 为基础实现响应式编程
SpringWebflux 和 SpringMVC 的比较
SpringMVC 方式实现： 同步阻塞，基于 SpringMVC+Servlet+Tomcat
SpringWebflux 方式实现： 异步非阻塞，基于 SpringWebflux+Reactor+Netty

2.3.0

2.2.0

2.1.2

2.1.1

2.1.0

2.0.2

2.0.1

2.0.0

1.9.2

1.9.1

1.9.0

1.8.2

1.8.1

1.8.0

1.7.2

1.7.1

1.7.0

1.6.2

1.6.1

1.6.0

1.5.2

1.5.1

1.5.0

1.4.2

1.4.1

logging

```
logging:  
#level 日志等级 指定命名空间的日志输出  
level:  
  com.fishpro.log: debug  
#file 指定输出文件的存储路径  
file: logs/app.log  
#pattern 指定输出场景的日志输出格式  
pattern:  
  console: "%d %-5level %logger : %msg%n"  
  file: "%d %-5level [%thread] %logger : %msg%n"
```

com.fasterxml.jackson

```
@JsonProperty(value = "fullName",access = JsonProperty.Access.READ_ONLY)
```

```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern= "yyyy-MM-dd HH:mm:ss")
```

javers-core

```
java 对象变对比工具
```

ehcache

```
jvm 虚拟机缓存
```

dockerfile-maven-plugin

```
基于一个 Spring Boot 的项目生成对应的 docker 镜像
```

```
编译：mvn clean package dockerfile:build
```

pom.xml >>

```
<build>  
  <plugin>  
    <groupId>com.spotify</groupId>  
    <artifactId>dockerfile-maven-plugin</artifactId>  
    <configuration>  
      <contextDirectory>${project.basedir}</contextDirectory>      上下文路径配置，此处设置为项目根路径  
    </configuration>
```

```
      <repository>${project.artifactId}</repository>  
      <buildArgs>  
        <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>  
      </buildArgs>
```

```
    </configuration>  
  </plugin>  
</build>
```

xxl-job

```
分布式 定时任务
```

依赖: xxl-job-core

常用版本: 2.3.0

application.properties >>

```
xxl.job.admin.addresses=http://127.0.0.1:8083/xxl-job-admin # 调度中心部署跟地址
xxl.job.accessToken= # 执行器通讯 TOKEN
xxl.job.executor.appname=xxl-job-executor-sample # 执行器AppName
xxl.job.executor.address= # 执行器注册
xxl.job.executor.ip= # 执行器IP
xxl.job.executor.port=9999 ##### 执行器端口号
xxl.job.executor.logpath=/tmp/xxl-job/ ##### 执行器运行日志文件存储磁盘路径
xxl.job.executor.logretentiondays=-1 ##### 执行器日志文件保存天数 (-1 表示永不过期)
```

源码分析 xxl-job-core3.3.0

package com.xxl.job.core.biz.model;
public class ReturnT<T> implements Serializable 返回值

hazelcast-spring

介绍: 分布式缓存

```
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-all</artifactId>
  <version>4.2.5</version>
</dependency>
```

配置

hazelcase.yaml >> hazelcast 启动的时候会 load 这个文件, 如果这个文件不存在, 或者没有 hazelcast 相关的配置文件和@Bean, 那么将不能启动 hazelcast。相关的配置不能放置在 application.yml 中

hazelcast:

```
cluster-name: hazelcast-cluster #必须指定
```

```
network:
```

```
join:
```

```
multicast:
```

```
enabled: true
```

CommandController >>

```
package com.dhb.hazelcast.demo.controller;

import com.dhb.hazelcast.demo.bean.CommandResponse;
import com.hazelcast.core.HazelcastInstance;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.ConcurrentMap;

@RestController
public class CommandController {

    @Autowired
    private HazelcastInstance hazelcastInstance;
```

```

private ConcurrentMap<String, String> retrieveMap() {
    return hazelcastInstance.getMap("map");
}

@PostMapping("/put")
public CommandResponse put(@RequestParam(value = "key") String key, @RequestParam(value = "value") String value) {
    retrieveMap().put(key, value);
    return new CommandResponse(value);
}

@GetMapping("/get")
public CommandResponse get(@RequestParam(value = "key") String key) {
    String value = retrieveMap().get(key);
    return new CommandResponse(value);
}

@GetMapping("/getSize")
public CommandResponse getSize() {
    int value = retrieveMap().size();
    return new CommandResponse(value + "");
}

}

```

libphonenumber (电话号码)

依赖:

```

<dependency>
    <groupId>com.googlecode.libphonenumber</groupId>
    <artifactId>libphonenumber</artifactId>
    <version>8.0.0</version>
</dependency>

```

源码解析 libphonenumber8.0.0

```

package com.google.i18n.phonenumbers;
public class PhoneNumberUtil      号码工具

public static synchronized PhoneNumberUtil getInstance()    获取实例
public PhoneNumber parse(String numberToParse, String defaultRegion) throws NumberParseException  根据国家号解析手
机号码，长度或格式不合法会抛出异常 // phoneNumberUtil.parse("+8612374166484", "US");

```

quartz

配置

```

## quartz 定时任务,采用数据库方式
quartz:
  job-store-type: jdbc
  initialize-schema: embedded
  auto-startup: true          #定时任务启动开关, true-开 false-关
  startup-delay: 1s            #延迟 1 秒启动定时任务
  overwrite-existing-jobs: true #启动时更新已存在的 Job
  properties:
    org:
      quartz:
        scheduler:
          instanceName: MyScheduler

```

```
instanceId: AUTO
jobStore:
    class: org.springframework.scheduling.quartz.LocalDataSourceJobStore
    driverDelegateClass: org.quartz.impl.jdbcjobstore.StdJDBCDelegate
    tablePrefix: QRTZ_
    isClustered: true
    misfireThreshold: 12000
    clusterCheckinInterval: 15000
threadPool:
    class: org.quartz.simpl.SimpleThreadPool
    threadCount: 10
    threadPriority: 5
    threadsInheritContextClassLoaderOfInitializingThread: true
```

schedlock-spring

源码分析 schedlock-spring4.21.0

```
@SchedulerLock    spring 任务锁
    name="XXScheduler_SysLogPurgeJob"    任务名称
    lockAtLeastFor="PT5M"        至少锁 5 分钟
```

BouncyCastle

源码分析 bcprov-jdk15on-1.58

```
package org.bouncycastle.jce.provider;
public final class BouncyCastleProvider extends Provider implements ConfigurableProvider    提供者

package org.bouncycastle.x509;
public class X509V3CertificateGenerator  证书生成器
    issuer  证书颁发者
    subject  证书使用者
public void setIssuerDN(X500Principal var1)    设置证书颁发者 标识名
public void setNotBefore(Date var1)    设置证书有效期
public void setNotAfter(Date var1)    设置证书有效期
public void setPublicKey(PublicKey publicKey) throws IllegalArgumentException    设置公钥
public void setSerialNumber(BigInteger bi)          设置证书序列号
public void setSignatureAlgorithm(String algorithm)    设置签名算法
    algorithm = "SHA1withRSA"
public void setSubjectDN(X500Principal var1)    设置证书使用者标识名
public X509Certificate generate(PrivateKey var1) throws CertificateEncodingException, IllegalStateException,
NoSuchAlgorithmException, SignatureException, InvalidKeyException  使用一个私钥生成一个证书，主要是为了进行签名操作

package org.bouncycastle.asn1.x509;
public class AlgorithmIdentifier extends ASN1Object  用于加密操作的算法
public static AlgorithmIdentifier getInstance(Object var0)  获取实例
public ASN1ObjectIdentifier getAlgorithm()          获得算法

package org.bouncycastle.asn1.pkcs;
public interface PKCSObjectIdentifiers  pkcs 辨识器
ASN1ObjectIdentifier rsaEncryption = pkcs_1.branch("1");
```

```
package org.bouncycastle.operator;
public class DefaultSignatureAlgorithmIdentifierFinder implements SignatureAlgorithmIdentifierFinder

package org.bouncycastle.operator;
public class DefaultDigestAlgorithmIdentifierFinder implements DigestAlgorithmIdentifierFinder

package org.bouncycastle.operator.bc;
public class BcRSAContentSignerBuilder extends BcContentSignerBuilder    BcRSA 内容签名人构建器
public BcRSAContentSignerBuilder(AlgorithmIdentifier sigAlgId, AlgorithmIdentifier digAlgId)

package org.bouncycastle.operator.bc;
public abstract class BcContentSignerBuilder    Bc 内容签名人构建器
public ContentSigner build(AsymmetricKeyParameter privateKey) throws OperatorCreationException 构建一个内容签名人

package org.bouncycastle.asn1.x500;
public class X500Name extends ASN1Object implements ASN1Choice    证书名称

package org.bouncycastle.crypto.params;
public class AsymmetricKeyParameter implements CipherParameters 非对称密钥参数

package org.bouncycastle.crypto.util;
public class PublicKeyFactory 公钥工厂
public static AsymmetricKeyParameter createKey(byte[] var0) throws IOException           创建一个非对称密钥参数
public static AsymmetricKeyParameter createKey(SubjectPublicKeyInfo var0) throws IOException   创建一个非对称密钥参数

package org.bouncycastle.crypto.util;
public class SubjectPublicKeyInfoFactory 使用者公钥信息工厂
public static SubjectPublicKeyInfo createSubjectPublicKeyInfo(AsymmetricKeyParameter var0) throws IOException   创建使用者公钥信息

package org.bouncycastle.asn1.x509;
public class SubjectPublicKeyInfo extends ASN1Object 获得使用者公钥信息

package org.bouncycastle.cert;
public class X509v3CertificateBuilder    证书构建器
public X509v3CertificateBuilder(X500Name issuer, BigInteger serial, Date notBefore, Date notAfter, X500Name subject, SubjectPublicKeyInfo publicKeyInfo)
public X509v3CertificateBuilder(X500Name issuer, BigInteger serial, Date notBefore, Date notAfter, Locale dateLocale, X500Name subject, SubjectPublicKeyInfo publicKeyInfo)
public X509v3CertificateBuilder(X500Name issuer, BigInteger serial, Time notBefore, Time notAfter, X500Name subject, SubjectPublicKeyInfo publicKeyInfo)

package org.bouncycastle.cert;
public class X509CertificateHolder implements Encodable, Serializable 证书保存器
public Certificate toASN1Structure()    返回基础 ASN1 本持有人证书结构。
```

```
package org.bouncycastle.pkcs;
public class PKCS10CertificationRequestBuilder PKCS10 证书请求构建器
public PKCS10CertificationRequestBuilder(X500Name subject, SubjectPublicKeyInfo publicKeyInfo)
public PKCS10CertificationRequest build(ContentSigner signer) 构建一个 PKCS10 证书请求
```

```
package org.bouncycastle.pkcs;
public class PKCS10CertificationRequest PKCS10 证书请求
public SubjectPublicKeyInfo getSubjectPublicKeyInfo() 获取使用人公钥信息
public X500Name getSubject() 获取使用人
```

```
package org.bouncycastle.crypto.params;
public class RSAKeyParameters extends AsymmetricKeyParameter RSA 密钥参数
public BigInteger getModulus() 获取模数
public BigInteger getExponent() 获取指数
```

使用

```
X509V3CertificateGenerator certGen = new X509V3CertificateGenerator();
certGen.setIssuerDN(new X500Principal(DN_CA));
certGen.setNotAfter(new Date(System.currentTimeMillis() + 100 * 24 * 60 * 60 * 1000));
certGen.setNotBefore(new Date());
certGen.setPublicKey(getRootPublicKey());
certGen.setSerialNumber(BigInteger.TEN);
certGen.setSignatureAlgorithm(SIG_ALG);
certGen.setSubjectDN(new X500Principal(DN_CA));
X509Certificate certificate = certGen.generate(getRootPrivateKey());
PKCS12BagAttributeCarrier bagAttr = (PKCS12BagAttributeCarrier)certificate;
bagAttr.setBagAttribute(
    PKCSObjectIdentifiers.pkcs_9_at_friendlyName,
    new DERBMPString("Digicert Coperation Certificate"));
writeFile("H:/certtest/ca.cer", certificate.getEncoded());
```

```
final AlgorithmIdentifier sigAlgId = new DefaultSignatureAlgorithmIdentifierFinder().find(SIG_ALG);
final AlgorithmIdentifier digAlgId = new DefaultDigestAlgorithmIdentifierFinder().find(sigAlgId);
```

```
PublicKey publicKey = getRootPublicKey();
PrivateKey privateKey = getRootPrivateKey();
```

```
X500Name issuer = new X500Name(DN_CA);
BigInteger serial = BigInteger.TEN;
Date notBefore = new Date();
Date notAfter = new Date(System.currentTimeMillis() + 100 * 24 * 60 * 60 * 1000);
X500Name subject = new X500Name(DN_CA);
```

```
AlgorithmIdentifier algId = AlgorithmIdentifier.getInstance(PKCSObjectIdentifiers.rsaEncryption.toString());
System.out.println(algId.getAlgorithm());
```

```

AsymmetricKeyParameter publicKeyParameter = PublicKeyFactory.createKey(publicKey.getEncoded());
SubjectPublicKeyInfo publicKeyInfo = SubjectPublicKeyInfoFactory.createSubjectPublicKeyInfo(publicKeyParameter);

//SubjectPublicKeyInfo publicKeyInfo = new SubjectPublicKeyInfo(algId, publicKey.getEncoded()); //此种方式不行，生成
证书不完整
X509v3CertificateBuilder x509v3CertificateBuilder = new X509v3CertificateBuilder(issuer, serial, notBefore, notAfter,
subject, publicKeyInfo);

BcRSAContentSignerBuilder contentSignerBuilder = new BcRSAContentSignerBuilder(sigAlgId, digAlgId);
AsymmetricKeyParameter privateKeyParameter = PrivateKeyFactory.createKey(privateKey.getEncoded());
ContentSigner contentSigner = contentSignerBuilder.build(privateKeyParameter);

X509CertificateHolder certificateHolder = x509v3CertificateBuilder.build(contentSigner);
Certificate certificate = certificateHolder.toASN1Structure();
writeFile("H:/certtest/ca.cer", certificate.getEncoded());

```

httpClient5

ConnectionKeepAliveStrategy

```

package org.apache.hc.client5.http;
public interface ConnectionKeepAliveStrategy 长连接策略

```

WeChat

Core

WeChat Pay Integration

Integrating WeChat Pay into your Spring Boot application involves [setting up a payment gateway](#) that allows users to make payments using WeChat.

This typically involves:

Obtaining WeChat Pay credentials

You'll need to [register as a merchant](#) on the WeChat Pay platform and [obtain your app ID, app secret](#), and other necessary credentials.

Configuring Spring Boot

Set up Spring Boot to handle the communication with WeChat Pay's API.

Implementing payment flow

Create endpoints in your Spring Boot application [to handle payment requests, generate payment codes, and verify payment results](#).

Configuration

Add Dependencies

WeChat Pay SDK:

Include the necessary dependencies for the WeChat Pay SDK in your project's build configuration (Maven or Gradle).

This SDK provides the classes and methods for interacting with WeChat Pay's API.

HTTP client:

If not included in the WeChat Pay SDK, add a dependency for an HTTP client like Apache HttpClient or OkHttp.

Configure WeChat Pay Properties

Create a configuration class to hold your WeChat Pay credentials (app ID, app secret, etc.).

Use Spring's property source abstraction to load these properties from a properties file or environment variables.

```

@Configuration
@PropertySource("classpath:wechatpay.properties")
public class WeChatPayConfig {

    @Value("${wechat.pay.appId}")
    private String appId;

    @Value("${wechat.pay.mchId}")
    private String mchId;

    @Value("${wechat.pay.key}")
    private String key;

    @Bean
    public WXPayConfig wxPayConfig() {
        WXPayConfig config = new WXPayConfig();
        config.setAppId(appId);
        config.setMchId(mchId);
        config.setKey(key);
        // Set other configurations as needed (e.g., signType, certPath)
        return config;
    }

    @Bean
    public WXPay wxPay() {
        WXPayConfig config = wxPayConfig();
        return new WXPay(config);
    }
}

```

Create a WeChat Pay Service

Create a service class to encapsulate the logic for interacting with WeChat Pay's API.

Use the WeChat Pay SDK to create instances of the required classes (e.g., WXPay, WXPayRequest).

Implement methods for:

- Generating payment codes (e.g., QR codes)
- Querying order status
- Handling refunds

Create REST Controllers

Expose REST endpoints in your Spring Boot application to handle payment requests.

These endpoints will receive payment requests from your frontend application and call the corresponding methods in your WeChat Pay service.

```

public class PaymentRequest {
    private String body;
    private String outTradeNo;
    private Integer totalFee;
    // ... other fields
}

public class PaymentResponse {
    private String prepayId;
    private Long timestamp;
    private String sign;
    // ... other fields
}

```

```

@RestController
@RequestMapping("/payment")
public class PaymentController {

```

```

@Autowired
private WXPay wxPay;

@PostMapping("/create")
public ResponseEntity<PaymentResponse> createPayment(@RequestBody PaymentRequest request) {
    try {
        // Generate payment code using WXPay
        WXPayUnifiedOrderRequest unifiedOrderRequest = new WXPayUnifiedOrderRequest();
        unifiedOrderRequest.setAppId(wxPay.getConfig().getAppId());
        unifiedOrderRequest.setMchId(wxPay.getConfig().getMchId());
        unifiedOrderRequest.setNonceStr(UUID.randomUUID().toString());
        // Set other request parameters as needed
        unifiedOrderRequest.setBody(request.getBody());
        unifiedOrderRequest.setOutTradeNo(request.getOutTradeNo());
        unifiedOrderRequest.setTotalFee(request.getTotalFee());
        unifiedOrderRequest.setTradeType("JSAPI"); // For web payments
        // ... set other parameters

        WXPayUnifiedOrderResult unifiedOrderResult = wxPay.unifiedorder(unifiedOrderRequest);

        if (unifiedOrderResult.getReturnCode().equals("SUCCESS") &&
unifiedOrderResult.getResultCode().equals("SUCCESS")) {
            PaymentResponse response = new PaymentResponse();
            response.setPrepayId(unifiedOrderResult.getPrepayId());
            response.setTimestamp(System.currentTimeMillis());
            response.setSign(wxPay.sign(response, wxPay.getConfig().getKey()));
            return ResponseEntity.ok(response);
        } else {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(new
PaymentResponse(unifiedOrderResult.getReturnMsg()));
        }
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(new PaymentResponse("Payment
failed: " + e.getMessage()));
    }
}
}

```

selenium-java

Core

Selenium Grid Hub

The Selenium Grid is a separate tool used for running automated tests across different browsers, operating systems, and machines.

It allows for parallel execution of tests, which is useful for scaling up testing environments.

Selenium Grid is a feature of the Selenium framework that allows for the distribution of test execution across multiple environments.

Components

Selenium Hub

The central point that receives test requests and distributes them to the appropriate nodes.

The Hub acts as a central point that manages the browser nodes and routes the test commands to the appropriate node.

Browser Nodes

Machines or environments where tests are executed. Each node can be configured with different browsers and operating systems.

When you start a browser node (e.g., Chrome node), it automatically includes the appropriate driver (ChromeDriver) for that browser.

Therefore, you do not need to start ChromeDriver separately.

URL

For the Hub Status

```
http://localhost:4444/status
```

For the Session Creation (using POST, not GET)

To create a new session, you should use the following endpoint:

```
POST http://localhost:4444/wd/hub/session
```

Using Docker

Prerequisites

Docker should be installed on your Mac. You can download and install Docker Desktop from Docker's official website.

Start the Selenium Hub Container

Open a terminal and run the following command to start the Selenium Hub:

```
docker run -d -p 4444:4444 --name selenium-hub selenium/hub
```

This command starts the Selenium Hub on port 4444.

Start Browser Nodes

If you haven't started any browser nodes yet, ensure that you do so.

The Hub alone does not run tests; you need at least one browser node to handle the requests.

For Chrome, run:

```
docker run -d --link selenium-hub:hub selenium/node-chrome
```

For Firefox, run:

```
docker run -d --link selenium-hub:hub selenium/node-firefox
```

These commands link the browser nodes to the Selenium Hub.

Set Up a Node

You'll need to register a node with the Selenium Hub that will use the local ChromeDriver.

Here's how to run a node that connects to your existing ChromeDriver:

```
podman run -d --name chrome-node \
--network host \
-e HUB_HOST=localhost \
-e HUB_PORT=4444 \
-v /path/to/chrome-for-testing:/opt/chrome \
-v /path/to/chromedriver:/usr/bin/chromedriver \
docker.io/selenium/node-chrome
```

Use the selenium/standalone-chrome Image

Alternatively, you can use the selenium/standalone-chrome image,

which includes both the Chrome browser and the Selenium server in a single container.

This is particularly useful if you don't need to separate the Hub and Node.

Use the Actual Path

```
sudo podman run -d --name chrome-node \
--network host --platform linux/amd64 \
-e HUB_HOST=localhost \
-e HUB_PORT=4444 \
-v "/Applications/chrome-mac-arm64/Google Chrome for Testing.app/Contents/MacOS/Google Chrome for \
Testing:/opt/chrome" \
-v "/Applications/chromedriver-mac-arm64/chromedriver:/usr/bin/chromedriver" \
docker.io/selenium/node-chrome
```

Access the Selenium Grid Console

Open a browser and navigate to <http://localhost:4444> to view the Selenium Grid console and check the status of the hub and nodes.

Configuration

Add Dependencies

```
<dependencies>
  <!-- Selenium dependency -->
  <dependency>
```

```

<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>4.15.0</version>
</dependency>

<!-- WebDriverManager dependency -->
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>5.5.3</version>
</dependency>

<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-chrome-driver</artifactId>
    <version>4.25.0</version> <!-- or the latest version -->
</dependency>
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-firefox-driver</artifactId>
    <version>4.25.0</version> <!-- or the latest version -->
</dependency>

</dependencies>

```

Configure WebDriverManager

You can create a configuration class in Spring Boot to set up the web driver using WebDriverManager.

```

import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class WebDriverConfig {
    @Bean
    public WebDriver chromeDriver() {
        // Set up ChromeDriver using WebDriverManager
        WebDriverManager.chromedriver().setup();
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--no-sandbox");
        options.addArguments("--disable-dev-shm-usage");
        //options.addArguments("--headless"); // Optional: run in headless mode if needed
        options.addArguments("--remote-debugging-port=9222");
        options.addArguments("--disable-web-security"); // Disable web security to avoid CORS issues
        options.addArguments("--remote-allow-origins=*");

        // Set timeouts
        options.setImplicitWaitTimeout(Duration.ofSeconds(10)); // Adjust as necessary
        options setPageLoadTimeout(Duration.ofSeconds(30)); // Adjust as necessary

        return new ChromeDriver(options);
    }
}

```

Using the WebDriver in Your Tests

You can now inject the WebDriver bean into your test classes or services:

```

package com.example.service;

import org.openqa.selenium.WebDriver;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

```

```

@Service
public class BrowserService {

    @Autowired
    private WebDriver chromeDriver;

    public void performTest() {
        chromeDriver.get("https://example.com");
        String title = chromeDriver.getTitle();
        System.out.println("Page title is: " + title);

        // Perform further browser interactions here...

        // Close the browser
        chromeDriver.quit();
    }
}

```

[selenium-remote-driver]

RemoteWebDriver

```

package org.openqa.selenium.remote;
@Augmentable
public class RemoteWebDriver implements WebDriver, JavascriptExecutor, HasInputDevices, HasCapabilities,
HasVirtualAuthenticator, Interactive, PrintsPage, TakesScreenshot
public RemoteWebDriver(Capabilities capabilities)
public RemoteWebDriver(URL remoteAddress, Capabilities capabilities)

```

jnativehook

NativeKeyEvent

```

package com.github.kwhat.jnativehook.keyboard;
public class NativeKeyEvent extends NativeInputEvent
public static String getKeyText(int keyCode)

```

File Processing

pdfbox

[pdfbox]

pdmodel

PDDocument

```

package org.apache.pdfbox.pdmodel;
public class PDDocument implements Closeable
public void save(File file) throws IOException
public void close() throws IOException
public void addPage(PDPage page)

```

PDPage

```

package org.apache.pdfbox.pdmodel;
public class PDPage implements COSObjectable, PDContentStream
public PDRectangle getMediaBox()

```

PDPageContentStream [CORE]

```

package org.apache.pdfbox.pdmodel;
public final class PDPageContentStream implements Closeable
public PDPageContentStream(PDDocument document, PDPage sourcePage) throws IOException
PDDocument pdfDocument = new PDDocument();

```

```

PDPage page = new PDPage();
String str=Files.readString(Path.of("<read-file-path>"), StandardCharsets.UTF_8);
byte[] decode = Base64.decode(str);
File outputFile=new File("<output-file-path>");

pdfDocument.addPage(page);
PDRectangle mediaBox = page.getMediaBox();
PDPageContentStream contentStream = new PDPageContentStream(pdfDocument, page);
PDIImageXObject fromByteArray = PDIImageXObject.createFromByteArray(pdfDocument, decode, null);
contentStream.drawImage(fromByteArray, 0, 0);

contentStream.close();
pdfDocument.save(outputFile);
pdfDocument.close();

public void drawImage(PDIImageXObject image, float x, float y) throws IOException    Draw Image
public void close() throws IOException

```

common

PDRectangle

```

public class PDRectangle implements COSObjectable
public PDRectangle()
public PDRectangle(float width, float height)
public PDRectangle(float x, float y, float width, float height)

```

itextpdf

依赖:

```

<dependency>
  <groupId>com.itextpdf </groupId>
  <artifactId>kernel</artifactId>
  <version>7.2.1</version>
</dependency>
<dependency>
  <groupId>com.itextpdf </groupId>
  <artifactId>io</artifactId>
  <version>7.2.1</version>
</dependency>

<dependency>
  <groupId>com.itextpdf </groupId>
  <artifactId>layout</artifactId>
  <version>7.2.1</version>
</dependency>

```

源码解析

```

package com.itextpdf.kernel.pdf
public class PdfReader implements Closable    Pdf 阅读器

```

```
package com.itextpdf.kernel.pdf  
public class PdfDocument implements IEventDispatcher, Closable    Pdf 文件对象
```

apache poi

依赖:

```
<dependency>  
    <groupId>org.apache.poi</groupId>  
    <artifactId>poi-ooxml</artifactId>  
    <version>5.2.2</version>  
</dependency>
```

Core

Reading Large Files

Apache POI **does not directly support** reading an XLSX file from within a ZIP file without extracting it first. Reading the XLSX file directly without compression is generally the best approach for optimizing read performance. When you read a compressed file (like a ZIP containing an XLSX file), additional overhead is introduced due to the decompression step, which can slow down the reading process.

Anchor

The Anchor in the context of adding a VML shape (such as a checkbox) to an Excel sheet using Apache POI defines the position and size of the shape relative to the cells in the worksheet.

The anchor is specified using a string value with eight comma-separated integers.

startColumn: The column number where the shape starts (0-based index).

startOffsetX: The offset within the starting column, in points.

startRow: The row number where the shape starts (0-based index).

startOffsetY: The offset within the starting row, in points.

endColumn: The column number where the shape ends (0-based index).

endOffsetX: The offset within the ending column, in points.

endRow: The row number where the shape ends (0-based index).

endOffsetY: The offset within the ending row, in points.

Checkbox

```
public static void addCheckboxToCell(XSSFSheet sheet) throws NoSuchFieldException, IllegalAccessException  
{  
    XSSFVMLDrawing drawing = null;  
    if (sheet.getCTWorksheet().getLegacyDrawing() != null) {  
        String legacyDrawingId = sheet.getCTWorksheet().getLegacyDrawing().getId();  
        drawing = (XSSFVMLDrawing)sheet.getRelationById(legacyDrawingId);  
    } else {  
        int drawingNumber = sheet.getPackagePart().getPackage()  
            .getPartsByContentType(XSSFRelation.VML_DRAWINGS.getContentType()).size() + 1;  
        POIXMLDocumentPart.RelationPart rp =  
            sheet.createRelationship(XSSFRelation.VML_DRAWINGS, XSSFFactory.getInstance(),  
drawingNumber, false);  
        drawing = rp.getDocumentPart();  
        String rId = rp.getRelationship().getId();  
        sheet.getCTWorksheet().addNewLegacyDrawing().setId(rId);  
    }  
  
    CTGroup grp = (CTGroup)CTGroup.Factory.newInstance();  
    CTShape shape = grp.addNewShape();  
    shape.setId("_x0000_t201");
```

```

shape.setType("#201" );
shape.setStyle("position:absolute; visibility:hidden");
shape.setFillcolor("#fffffe1");
shape.setInsetmode(STInsetMode.AUTO);
shape.addNewFill().setColor("#fffffe1");
CTShadow shadow = shape.addNewShadow();
shadow.setOn(STTrueFalse.T);
shadow.setColor("black");
shadow.setObscured(STTrueFalse.T);
shape.addNewPath().setConnecttype(STConnectType.NONE);
shape.addNewTextbox().setStyle("mso-direction-alt:auto");
CTClientData cldata = shape.addNewClientData();
cldata.setObjectType(STObjectType.NOTE);
cldata.addNewMoveWithCells();
cldata.addNewSizeWithCells();
cldata.addNewAnchor().setStringValue("1, 15, 0, 2, 3, 15, 3, 16");
cldata.addNewAutoFill().setStringValue("False");
cldata.addNewRow().setBigIntegerValue(BigInteger.valueOf(0L));
cldata.addNewColumn().setBigIntegerValue(BigInteger.valueOf(0L));
XmlCursor grpCur = grp.newCursor();

Field rootField = XSSFVMLDrawing.class.getDeclaredField("root");
rootField.setAccessible(true);
XmlDocument root = ( XmlDocument )rootField.get(drawing);
XmlCursor xml = root.getXml().newCursor();
grpCur.copyXmlContents(xml);

// com.microsoft.schemas.vml.CTShape shape = CTShape.Factory.newInstance();
// shape.setId("_x0000_s_x0000_t201");
// shape.setType("#_x0000_t201");
// shape.setFillcolor("#fffffe1");
// CTClientData cldata = shape.addNewClientData();
// cldata.addNewAnchor().setStringValue("3, 0, 3, 0, 4, 0, 4, 0");
// cldata.setObjectType(com.microsoft.schemas.office.excel.STObjectType.CHECKBOX);
// cldata.addTextVAlign("Center");
// cldata.addChecked(java.math.BigInteger.valueOf(1));
//
// XmlCursor xmlCursor = shape.newCursor();
// Field rootField = XSSFVMLDrawing.class.getDeclaredField("root");
// rootField.setAccessible(true);
// XmlDocument root = ( XmlDocument )rootField.get(drawing);
// XmlCursor sourceCursor = root.addNewXml().newCursor();
// xmlCursor.copyXmlContents(sourceCursor);
// xmlCursor.close();
}

```

Configuration

```

<!-- Java API To Access Microsoft Format Files-->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>5.3.0</version>
</dependency>
<!-- Java API To Access Microsoft Format Files -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml-schemas</artifactId>
    <version>4.1.2</version>
</dependency>

```

[xml-beans]

DocumentFactory

```
package org.apache.xmlbeans.impl.schema;
public class DocumentFactory<T> extends AbstractDocumentFactory<T>
DocumentFactory
public DocumentFactory(SchemaTypeSystem typeSystem, String typeHandle) {
    super(typeSystem, typeHandle);
}

newInstance
public T newInstance() {
    return this.getTypeLoader().newInstance(this.getType(), (XmlOptions)null);
}

newInstance
public T newInstance(XmlOptions options) {
    return this.getTypeLoader().newInstance(this.getType(), options);
}
```

AbstractDocumentFactory

```
package org.apache.xmlbeans.impl.schema;
public class AbstractDocumentFactory<T> extends ElementFactory<T>
AbstractDocumentFactory
public AbstractDocumentFactory(SchemaTypeSystem typeSystem, String typeHandle) {
    super(typeSystem, typeHandle);
}
```

ElementFactory

```
package org.apache.xmlbeans.impl.schema;
public class ElementFactory<T>
private final SchemaType type;
private final SchemaTypeSystem typeSystem;
ElementFactory
public ElementFactory(SchemaTypeSystem typeSystem, String typeHandle) {
    this.typeSystem = typeSystem;
    this.type = (SchemaType)typeSystem.resolveHandle(typeHandle);
}
getTypeLoader
public SchemaTypeSystem getTypeLoader() {
    return this.typeSystem;
}
```

SchemaTypeSystemImpl

```
package org.apache.xmlbeans.impl.schema;
public class SchemaTypeSystemImpl extends SchemaTypeLoaderBase implements SchemaTypeSystem
SchemaTypeSystemImpl
public SchemaTypeSystemImpl(Class<?> indexclass) {
    String fullname = indexclass.getName();
    this._name = fullname.substring(0, fullname.lastIndexOf(46));
    XBeanDebug.LOG.atTrace().log("Loading type system {}", this._name);
    this._classloader = indexclass.getClassLoader();
    this._linker = SchemaTypeLoaderImpl.build((SchemaTypeLoader[])null, (ResourceLoader)null,
this._classloader, this.getMetadataPath());
    this._resourceLoader = new ClassLoaderResourceLoader(this._classloader);

    try {
        this.initFromHeader();
    } catch (Error | RuntimeException var4) {
```

```
XBeanDebug.LOG.atDebug().withThrowable(var4).log(var4.getMessage());
throw var4;
}

XBeanDebug.LOG.atTrace().log("Finished loading type system {}", this._name);
}
```

[poi]
poifs
filesystem

FileMagic

```
package org.apache.poi.poifs.filesystem;

public enum FileMagic {

    prepareToCheckMagic
    public static InputStream prepareToCheckMagic(InputStream stream) {
        return (InputStream)(stream.markSupported() ? stream : new BufferedInputStream(stream));
    }

    valueOf
    public static FileMagic valueOf(InputStream inp) throws IOException {
        if (!inp.markSupported()) {
            throw new IOException("getFileMagic() only operates on streams which support mark(int)");
        } else {
            byte[] data = IOUtils.peekFirstNBytes(inp, 44);
            return valueOf(data);
        }
    }
}
```

util

IOUtils

```
package org.apache.poi.util;

public final class IOUtils {

    peekFirstNBytes
    public static byte[] peekFirstNBytes(InputStream stream, int limit) throws IOException, EmptyFileException {
        checkByteSizeLimit(limit);
        stream.mark(limit);
        UnsynchronizedByteArrayOutputStream bos = new UnsynchronizedByteArrayOutputStream(limit);
        copy(new BoundedInputStream(stream, (long)limit), (OutputStream)bos);
        int readBytes = bos.size();
        if (readBytes == 0) {
            throw new EmptyFileException();
        } else {
            if (readBytes < limit) {
                bos.write(new byte[limit - readBytes]);
            }
        }
        byte[] peekedBytes = bos.toByteArray();
        if (stream instanceof PushbackInputStream) {
            PushbackInputStream pin = (PushbackInputStream)stream;
            pin.unread(peekedBytes, 0, readBytes);
        } else {
            stream.reset();
        }
        return peekedBytes;
    }
}
```

```
package org.apache.poi.ss.usermodel;
```

Date1904Support

```
public interface Date1904Support
```

```
boolean isDate1904();
```

[poi-ooxml-lite]

WorkbookDocument

```
package org.openxmlformats.schemas.spreadsheetml.x2006.main;
```

```
public interface WorkbookDocument extends XmlObject
```

```
    org.apache.xmlbeans.impl.schema.DocumentFactory
```

```
    org.apache.xmlbeans.impl.schema.AbstractDocumentFactory
```

```
    org.apache.xmlbeans.impl.schema.ElementFactory
```

```
    org.apache.poi.schemas.ooxml.system.ooxml.TypeSystemHolder
```

```
    org.apache.xmlbeans.impl.schema.SchemaTypeSystemImpl
```

```
DocumentFactory<WorkbookDocument> Factory = new DocumentFactory(TypeSystemHolder.typeSystem,
```

```
"workbookec17doctype");
```

```
SchemaType type = Factory.getType();
```

```
CTWorkbook getWorkbook();
```

```
void setWorkbook(CTWorkbook var1);
```

```
CTWorkbook addNewWorkbook();
```

TypeSystemHolder

```
package org.apache.poi.schemas.ooxml.system.ooxml;
```

```
public final class TypeSystemHolder extends SchemaTypeSystemImpl
```

```
public static final TypeSystemHolder typeSystem = new TypeSystemHolder();
```

```
private TypeSystemHolder() {  
    super(TypeSystemHolder.class);  
}
```

[poi-ooxml]

hssf

HSSFWorkbook

```
package org.apache.poi.hssf.usermodel;
```

```
public final class HSSFWorkbook extends POIDocument implements Workbook
```

Excel2003 以前（包括 2003）的版本，扩展名是.xls 表示一个 excel 文件工作薄，操作

opc

ZipPackage

```
package org.apache.poi.openxml4j.opc;
```

```
public final class ZipPackage extends OPCPackage
```

```
ZipPackage(InputStream in, PackageAccess access) throws IOException
```

ZipHelper

```
package org.apache.poi.openxml4j.opc.internal;
```

```
public final class ZipHelper  
public static ZipArchiveThresholdInputStream openZipStream(InputStream stream) throws IOException
```

openxmlformats

CTWorksheet

```
package org.openxmlformats.schemas.spreadsheetml.x2006.main;  
public interface CTWorksheet extends XmlObject 老版本 sheet  
CTLegacyDrawing getLegacyDrawing(); 获取适配绘图器
```

CTShapeStyle

```
package org.openxmlformats.schemas.drawingml.x2006.main;  
public interface CTShapeStyle extends XmlObject 形状
```

```
package org.openxmlformats.schemas.spreadsheetml.x2006.main;  
public interface CTLegacyDrawing extends XmlObject 适配绘图器
```

STShapeType

```
package org.openxmlformats.schemas.drawingml.x2006.main;  
public interface STShapeType extends XmlToken  
public static final class Enum extends StringEnumAbstractBase {  
    static final int INT_ACTION_BUTTON_HELP = 163;  
}
```

STPresetLineDashVal

```
package org.openxmlformats.schemas.drawingml.x2006.main;  
public interface STPresetLineDashVal extends XmlToken 线类型
```

openxml4j

opc

ZipPackage

```
package org.apache.poi.openxml4j.opc;  
public final class ZipPackage extends OPCPackage  
private final ZipEntrySource zipArchive;  
  
ZipPackage  
ZipPackage(InputStream in, PackageAccess access) throws IOException {  
    super(access);  
    ZipArchiveThresholdInputStream zis = ZipHelper.openZipStream(in);  
  
    try {  
        this.zipArchive = new ZipInputStreamZipEntrySource(zis);  
    } catch (RuntimeException | IOException var5) {  
        IOUtils.closeQuietly(zis);  
        throw var5;  
    }  
}  
  
getPartsImpl  
protected PackagePartCollection getPartsImpl() throws InvalidFormatException {  
    PackagePartCollection newPartList = new PackagePartCollection();  
    if (this.zipArchive == null) {  
        return newPartList;  
    } else {  
        ZipArchiveEntry contentTypeEntry = this.zipArchive.getEntry("[Content_Types].xml");  
        if (contentTypeEntry == null) {
```

```

        boolean hasMimetype = this.zipArchive.getEntry("mimetype") != null;
        boolean hasSettingsXML = this.zipArchive.getEntry("settings.xml") != null;
        if (hasMimetype && hasSettingsXML) {
            throw new ODFNotOfficeXmlFileException("The supplied data appears to be in ODF (Open Document) Format. Formats like these (eg ODS, ODP) are not supported, try Apache ODFToolkit");
        } else if (!this.zipArchive.getEntries().hasMoreElements()) {
            throw new NotOfficeXmlFileException("No valid entries or contents found, this is not a valid OOXML (Office Open XML) file");
        } else {
            throw new InvalidFormatException("Package should contain a content type part [M1.13]");
        }
    } else if (this.contentTypeManager != null) {
        throw new InvalidFormatException("ContentTypeManager can only be created once. This must be a cyclic relation?");
    } else {
        try {
            this.contentTypeManager = new ZipContentTypeManager(this.zipArchive.getInputStream(contentTypeEntry), this);
        } catch (IOException var6) {
            throw new InvalidFormatException(var6.getMessage(), var6);
        }

        List<EntryTriple> entries =
(List)Collections.list(this.zipArchive.getEntries()).stream().filter((zipArchiveEntry) -> {
            return !ignoreEntry(zipArchiveEntry);
        }).map((zae) -> {
            return new EntryTriple(zae, this.contentTypeManager);
        }).filter((mm) -> {
            return mm.partName != null;
        }).sorted().collect(Collectors.toList());
        Iterator var4 = entries.iterator();

        while(var4.hasNext()) {
            EntryTriple et = (EntryTriple)var4.next();
            et.register(newPartList);
        }

        return newPartList;
    }
}
}

addRelationship
public PackageRelationship addRelationship(URI targetUri, TargetMode targetMode, String relationshipType, String id) {
    if (id == null || id.length() == 0) {
        if (this.nextRelationshipId == -1) {
            this.nextRelationshipId = this.size() + 1;
        }

        do {
            id = "rId" + this.nextRelationshipId++;
        } while(this.relationshipsByID.get(id) != null);
    }

    PackageRelationship rel = new PackageRelationship(this.container, this.sourcePart, targetUri, targetMode, relationshipType, id);
    this.addRelationship(rel);
    if (targetMode == TargetMode.INTERNAL) {
        this.internalRelationshipsByTargetName.put(targetUri.toASCIIString(), rel);
    }
}

return rel;
}

```

OPCPackage

package org.apache.poi.openxml4j.opc;

```

public abstract class OPCPackage implements RelationshipSource, Closeable
OPCPackage
OPCPackage(PackageAccess access) {
    if (this.getClass() != ZipPackage.class) {
        throw new IllegalArgumentException("PackageBase may not be subclassed");
    } else {
        this.packageAccess = access;
        ContentType contentType = newCorePropertiesPart();
        this.partUnmarshallers.put(contentType, new PackagePropertiesUnmarshaller());
        this.partMarshallers.put(contentType, new ZipPackagePropertiesMarshaller());
    }
}
open
public static OPCPackage open(InputStream in) throws InvalidFormatException, IOException {
    OPCPackage pack = new ZipPackage(in, PackageAccess.READ_WRITE);
    try {
        if (pack.partList == null) {
            pack.getParts();
        }
        return pack;
    } catch (RuntimeException | InvalidFormatException var3) {
        IOUtils.closeQuietly(pack);
        throw var3;
    }
}

```

RelationshipSource

```

package org.apache.poi.openxml4j.opc;
public interface RelationshipSource
    PackageRelationship addRelationship(PackagePartName var1, TargetMode var2, String var3);
    PackageRelationship addRelationship(PackagePartName var1, TargetMode var2, String var3, String var4);

```

PackageRelationshipCollection

```

package org.apache.poi.openxml4j.opc;
public final class PackageRelationshipCollection implements Iterable<PackageRelationship>
private static final Logger LOG = LogManager.getLogger(PackageRelationshipCollection.class);
private final TreeMap<String, PackageRelationship> relationshipsByID; //when the PackageRelationship is updated,
update its associated map storage data.
private final TreeMap<String, PackageRelationship> relationshipsByType;
private HashMap<String, PackageRelationship> internalRelationshipsByTargetName;
private PackagePart relationshipPart;
private PackagePart sourcePart;
private PackagePartName partName;
private OPCPackage container;
private int nextRelationshipId;

```

PackageRelationshipCollection

```

PackageRelationshipCollection()
{
    this.relationshipsByID = new TreeMap();
    this.relationshipsByType = new TreeMap();
    this.internalRelationshipsByTargetName = new HashMap();
    this.nextRelationshipId = -1;
}
PackageRelationshipCollection
public PackageRelationshipCollection(PackageRelationshipCollection coll, String filter) {

```

```

this();
Iterator var3 = coll.relationshipsByID.values().iterator();

while(true) {
    PackageRelationship rel;
    do {
        if (!var3.hasNext()) {
            return;
        }

        rel = (PackageRelationship)var3.next();
    } while(filter != null && !rel.getRelationshipType().equals(filter));

    this.addRelationship(rel);
}
}

addRelationship
public PackageRelationship addRelationship(URI targetUri, TargetMode targetMode, String relationshipType,
String id) {
    if (id == null || id.length() == 0) {
        if (this.nextRelationshipId == -1) {
            this.nextRelationshipId = this.size() + 1;
        }

        do {
            id = "rId" + this.nextRelationshipId++;
        } while(this.relationshipsByID.get(id) != null);
    }

    PackageRelationship rel = new PackageRelationship(this.container, this.sourcePart, targetUri, targetMode,
relationshipType, id);
    this.addRelationship(rel);
    if (targetMode == TargetMode.INTERNAL) {
        this.internalRelationshipsByTargetName.put(targetUri.toASCIIString(), rel);
    }
}

return rel;
}

addRelationship
public void addRelationship(PackageRelationship relPart) {
    if (relPart != null && relPart.getId() != null && !relPart.getId().isEmpty()) {
        this.relationshipsByID.put(relPart.getId(), relPart);
        this.relationshipsByType.put(relPart.getRelationshipType(), relPart);
    } else {
        throw new IllegalArgumentException("invalid relationship part/id: " + (relPart == null ? "<null>" :
relPart.getId()) + " for relationship: " + relPart);
    }
}

removeRelationship
public void removeRelationship(String id) {
    PackageRelationship rel = (PackageRelationship)this.relationshipsByID.get(id);
    if (rel != null) {
        this.relationshipsByID.remove(rel.getId());
        this.relationshipsByType.values().remove(rel);
        this.internalRelationshipsByTargetName.values().remove(rel);
    }
}

iterator
public Iterator<PackageRelationship> iterator() {
    return this.relationshipsByID.values().iterator();
}

spliterator
public Spliterator<PackageRelationship> spliterator() {
    return this.relationshipsByID.values().spliterator();
}

```

```

findExistingInternalRelation
public PackageRelationship findExistingInternalRelation(PackagePart packagePart) {
    return
(PackageRelationship)this.internalRelationshipsByTargetName.get(packagePart.getPartName().getName());
}
clear
public void clear() {
    this.relationshipsByID.clear();
    this.relationshipsByType.clear();
    this.internalRelationshipsByTargetName.clear();
}

```

ZipHelper

```

package org.apache.poi.openxml4j.opc.internal;
public final class ZipHelper
openZipStream
public static ZipArchiveThresholdInputStream openZipStream(InputStream stream) throws IOException {
    InputStream checkedStream = FileMagic.prepareToCheckMagic(stream);
    verifyZipHeader(checkedStream);
    return new ZipArchiveThresholdInputStream(new ZipArchiveInputStream(checkedStream));
}
verifyZipHeader
private static void verifyZipHeader(InputStream stream) throws NotOfficeXmlFileException, IOException {
    InputStream is = FileMagic.prepareToCheckMagic(stream);
    FileMagic fm = FileMagic.valueOf(is);
    switch (fm) {
        case OLE2:
            throw new OLE2NotOfficeXmlFileException("The supplied data appears to be in the OLE2 Format. You
are calling the part of POI that deals with OOXML (Office Open XML) Documents. You need to call a different
part of POI to process this data (eg HSSF instead of XSSF)");
        case XML:
            throw new NotOfficeXmlFileException("The supplied data appears to be a raw XML file. Formats such
as Office 2003 XML are not supported");
        default:
    }
}

```

microsoft

STObjectType

```

package com.microsoft.schemas.office.xlsx;
public interface STObjectType extends XmlString    开发者类型

```

CTShape

```

package com.microsoft.schemas.vml;
public interface CTShape extends XmlObject
CTClientData addNewClientData();    添加新的 client 数据

```

CTClientData

```

package com.microsoft.schemas.office.xlsx;
public interface CTClientData extends XmlObject
void setObjectType(com.microsoft.schemas.office.xlsx.STObjectType.Enum var1);    设置对象类型

```

ss

Workbook

```

package org.apache.poi.ss.usermodel;
public interface Workbook extends Closeable, Iterable<Sheet>          工作薄方法
Sheet createSheet(String var1);           创建一个 sheet (之后再 get)
Sheet getSheetAt(int var1);             根据索引获取 sheet
Sheet getSheet(String var1);           根据 sheet 名获取 sheet
CellStyle createCellStyle();            创建一个风格
Font createFont();                   创建一个字体

```

CellStyle

```

package org.apache.poi.ss.usermodel;
public interface CellStyle
void setFillForegroundColor(short var1);    设置颜色 // IndexedColors.WHITE.getIndex()
void setFillPattern(FillPatternType var1);   设置填充 // CellStyle.SOLID_FOREGROUND
HSSFCellStyle.SOLID_FOREGROUND
void setBorderBottom(BorderStyle var1);     设置边框 // HSSFCellStyle.BORDER_THIN
void setBorderTop(BorderStyle var1);        设置边框
void setAlignment(HorizontalAlignment var1);  设置靠边 // HSSFCellStyle.ALIGN_CENTER
void setFont(Font var1);                  设置字体

```

Font

```

package org.apache.poi.ss.usermodel;
public interface Font          字体
void setFontHeightInPoints(short var1); // (short) 12
void setBold(boolean var1);       加粗

```

CellBase

```

package org.apache.poi.ss.usermodel;
public abstract class CellBase implements Cell  格子基础操作
public void setCellValue(double value)         设置格子的值
public final void setCellType(CellType cellType)  设置格子格式
public final void setCellFormula(String formula) // cell.setCellFormula("SUM(C2:C3)");

```

CellType

```

package org.apache.poi.ss.usermodel;
public enum CellType  格子格式
    _NONE(-1),
    NUMERIC(0),    数字, 日期
    STRING(1),     字符串
    FORMULA(2),    公式
    BLANK(3),
    BOOLEAN(4),    布尔
    ERROR(5);
public class ShapeTypes 形状类型  DrawingML 中所有已知类型的自动形状
    public static final int ACTION_BUTTON_HELP = 163;

```

PackageHelper

```
package org.apache.poi.ooxml.util;
public final class PackageHelper Provides handy methods to work with OOXML packages
    org.apache.poi.openxml4j.opc.OPCPackage
    org.apache.poi.openxml4j.opc.ZipPackage
    org.apache.poi.openxml4j.opc.internal.ZipHelper
    org.apache.poi.poifs.filesystem.FileMagic
    org.apache.poi.util.IOUtils

open
public static OPCPackage open(InputStream stream, boolean closeStream) throws IOException {
    OPCPackage var2;
    try {
        var2 = OPCPackage.open(stream);
    } catch (InvalidFormatException var6) {
        throw new POIXMLError(var6);
    } finally {
        if (closeStream) {
            stream.close();
        }
    }
}
return var2;
}
```

xssf

SXSSFWorkbook

```
package org.apache.poi.xssf.streaming;
public class SXSSFWorkbook implements Workbook 表示一个 excel 文件工作薄, 可以缓存
public SXSSFWorkbook(int rowAccessWindowSize) 创建工作薄, 指定缓存大小 rowAccessWindowSize
public void setCompressTempFiles(boolean compress) 压缩临时文件
```

XSSFWorkbook

```
package org.apache.poi.xssf.usermodel;
public class XSSFWorkbook extends POIXMLDocument implements Workbook, Date1904Suppor 表示一个 excel 文件工作
薄, 操作 Excel2007 的版本, 扩展名是.xlsx
public XSSFWorkbook(InputStream is) throws IOException 从输入流 inputStream 中读取 excel (try catch)
public List<XSSFPictureData> getAllPictures() 获取所有图片
public XSSFCreationHelper getCreationHelper() 创建工具
XSSFWorkbook
private XSSFWorkbook(InputStream is, boolean closeStream) throws IOException {
    this(PackageHelper.open(is, closeStream));
}
```

XSSFWorkbook

```
public XSSFWorkbook(OPCPackage pkg) throws IOException {
    super(pkg);
    this._udfFinder = new IndexedUDFFinder(new UDFFinder[]{AggregatingUDFFinder.DEFAULT});
    this._missingCellPolicy = MissingCellPolicy.RETURN_NULL_AND_BLANK;
    this.cellFormulaValidation = true;
    this.xssfFactory = XSSFFactory.getInstance();
    this.beforeDocumentRead();
    this.load(this.xssfFactory);
    this.setBookViewsIfMissing();
}
```

```

beforeDocumentRead
protected void beforeDocumentRead() {
    if (this.getCorePart().getContentType().equals(XSSFRrelation.XLSB_BINARY_WORKBOOK.getContentType())) {
        throw new XLSBUnsupportedException();
    } else {
        this.pivotTables = new ArrayList();
        this.pivotCaches = new ArrayList();
    }
}

setBookViewsIfMissing
private void setBookViewsIfMissing() {
    if (!this.workbook.isSetBookViews()) {
        CTBookViews bvs = this.workbook.addNewBookViews();
        CTBookView bv = bvs.addNewWorkbookView();
        bv.setActiveTab(0L);
    }
}

load
protected final void load(POIXMLFactory factory) throws IOException {
    Map<PackagePart, POIXMLDocumentPart> context = new HashMap();

    try {
        this.read(factory, context);
    } catch (OpenXML4JException var4) {
        throw new POIXMLException(var4);
    }

    this.onDocumentRead();
    context.clear();
}
}

read
protected void read(POIXMLFactory factory, Map<PackagePart, POIXMLDocumentPart> context) throws
OpenXML4JException {
    PackagePart pp = this.getPackagePart();
    if (pp.getContentType().equals(XWPFRrelation.GLOSSARY_DOCUMENT.getContentType())) {
        LOG.atWarn().log("POI does not currently support template.main+xml (glossary) parts. Skipping this
part for now.");
    } else {
        POIXMLDocumentPart otherChild = (POIXMLDocumentPart)context.put(pp, this);
        if (otherChild != null && otherChild != this) {
            throw new POIXMLException("Unique PackagePart-POIXMLDocumentPart relation broken!");
        } else if (pp.hasRelationships()) {
            PackageRelationshipCollection rels = this.packagePart.getRelationships();
            List<POIXMLDocumentPart> readLater = new ArrayList();
            Iterator var7 = rels.iterator();

            while(var7.hasNext()) {
                PackageRelationship rel = (PackageRelationship)var7.next();
                if (rel.getTargetMode() == TargetMode.INTERNAL) {
                    URI uri = rel.getTargetURI();
                    PackagePartName relName;
                    if (uri.getRawFragment() != null) {
                        relName = PackagingURIHelper.createPartName(uri.getPath());
                    } else {
                        relName = PackagingURIHelper.createPartName(uri);
                    }

                    PackagePart p = this.packagePart.getPackage().getPart(relName);
                    if (p == null) {
                        LOG.atError().log("Skipped invalid entry {}", rel.getTargetURI());
                    } else {
                }
            }
        }
    }
}

```

```
        POIXMLDocumentPart childPart = (POIXMLDocumentPart)context.get(p);
        if (childPart == null) {
            childPart = factory.createDocumentPart(this, p);
            if (this instanceof XDDFChart && childPart instanceof XSSFWorbook) {
                ((XDDFChart)this).setWorkbook((XSSFWorbook)childPart);
            }
            childPart.parent = this;
            context.put(p, childPart);
            readLater.add(childPart);
        }

        this.addRelation(rel, childPart);
    }
}

var7 = readLater.iterator();

while(var7.hasNext()) {
    POIXMLDocumentPart childPart = (POIXMLDocumentPart)var7.next();
    childPart.read(factory, context);
}

}
```

onWorkbookCreate

```
private void onWorkbookCreate() {
    this.workbook = (CTWorkbook)CTWorkbook.Factory.newInstance();
    CTWorkbookPr workbookPr = this.workbook.addNewWorkbookPr();
    workbookPr.setDate1904(false);
    this.setBookViewsIfMissing();
    this.workbook.addNewSheets();
    POIXMLProperties.ExtendedProperties expProps = this.getProperties().getExtendedProperties();
    expProps.getUnderlyingProperties().setApplication("Apache POI");
    this.sharedStringSource = (SharedStringsTable)this.createRelationship(XSSFRrelation.SHARED_STRINGS,
this.xssfFactory);
    this.stylesSource = (StylesTable)this.createRelationship(XSSFRrelation.STYLES, this.xssfFactory);
    this.stylesSource.setWorkbook(this);
    this.namedRanges = new ArrayList();
    this.namedRangesByName = new ArrayListValuedHashMap();
    this.sheets = new ArrayList();
    this.pivotTables = new ArrayList();
    this.externalLinks = new ArrayList();
}
```

XSSFSheet

```
package org.apache.poi.xssf.usermodel;
public class XSSFSheet extends POIXMLDocumentPart implements Sheet, OoxmlSheetExtensions 表示一个 sheet
public XSSFRow createRow(int rounum)    创建行
public int getLastRowNum()            获取最后一行自己的索引, 索引从 0 开始
public XSSFRow getRow(int rounum)      获取行, 默认第一行为标题行, index =0, 1, 2..
public void setColumnWidth(int columnIndex, int width)    设置列宽
public XSSFDrawing createDrawingPatriarch()    创建一个绘图
public CTWorksheet getCTWorksheet()    获取老版本 sheet
```

XSSFRow

```
package org.apache.poi.xssf.usermodel;
```

```

public class XSSFRow implements Row, Comparable<XSSFRow> 表示一行
public XSSFCell createCell(int columnIndex) 创建一个格子
public XSSFCell getCell(int cellnum) 获取一个格子
public XSSFCell getCell(int cellnum, MissingCellPolicy policy) 获取一个格子, 指定缺失策略
    policy: Row.MissingCellPolicy.CREATE_NULL_AS_BLANK 当取到 null, 返回空字符串 blank:
              Row.MissingCellPolicy.RETURN_NULL_AND_BLANK 当取到 null 返回 null, 取到 blank 返回 blank:
              Row.MissingCellPolicy.RETURN_BLANK_AS_NULL 当取到空字符串 blank, 返回 null:
public class CellIterator implements Iterator<Cell>
    final int maxColumn = SXSSFRow.this.getLastCellNum();
    int pos;

```

XSSFCreationHelper

```

package org.apache.poi.xssf.usermodel;
public class XSSFCreationHelper implements CreationHelper 创建工具
public XSSFClientAnchor createClientAnchor() 创建一个客户端锚点
public XSSFRichTextString createRichTextString(String text) 创建一个富文本

```

XSSFRichTextString

```

package org.apache.poi.xssf.usermodel;
public class XSSFRichTextString implements RichTextString 富文本字符串

```

XSSFComment

```

public class XSSFComment implements Comment 注释
public void setString(RichTextString string) 设置内容
public void setAuthor(String author) 设置作者

```

XSSFVMLDrawing

```

package org.apache.poi.xssf.usermodel;
public final class XSSFVMLDrawing extends POIXMLDocumentPart

```

newDrawing

```

private void newDrawing() {
    this.root = (XmlDocument) XmlDocument.Factory.newInstance();
    XmlCursor xml = this.root.addNewXml().newCursor();
    Throwable var2 = null;

    try {
        ShapelayoutDocument layDoc = (ShapelayoutDocument)ShapelayoutDocument.Factory.newInstance();
        CTShapeLayout layout = layDoc.addNewShapelayout();
        layout.setExt(STExt.EDIT);
        CTIdMap idmap = layout.addNewIdmap();
        idmap.setExt(STExt.EDIT);
        idmap.setData("1");
        xml.toEndToken();
        XmlCursor layCur = layDoc.newCursor();
        Throwable var7 = null;

        try {
            layCur.copyXmlContents(xml);
        } catch (Throwable var53) {
            var7 = var53;
            throw var53;
        }
    }
}

```

```

} finally {
    if (layCur != null) {
        if (var7 != null) {
            try {
                layCur.close();
            } catch (Throwable var51) {
                var7.addSuppressed(var51);
            }
        } else {
            layCur.close();
        }
    }
}

CTGroup grp = (CTGroup)CTGroup.Factory.newInstance();
CTShapetype shapetype = grp.addNewShapetype();
this._shapeTypeId = "_x0000_t202";
shapetype.setId(this._shapeTypeId);
shapetype.setCoordsize("21600,21600");
shapetype.setSpt(202.0F);
shapetype.setPath2("m,l,21600r21600,121600,xe");
shapetype.addNewStroke().setJoinstyle(STStrokeJoinStyle.MITER);
CTPath path = shapetype.addNewPath();
path.setGradientshapeok(STTrueFalse.T);
path.setConnecttype(STConnectType.RECT);
xml.toEndToken();
XmlCursor grpCur = grp.newCursor();
Throwable var10 = null;

try {
    grpCur.copyXmlContents(xml);
} catch (Throwable var52) {
    var10 = var52;
    throw var52;
} finally {
    if (grpCur != null) {
        if (var10 != null) {
            try {
                grpCur.close();
            } catch (Throwable var50) {
                var10.addSuppressed(var50);
            }
        } else {
            grpCur.close();
        }
    }
}

} catch (Throwable var56) {
    var2 = var56;
    throw var56;
} finally {
    if (xml != null) {
        if (var2 != null) {
            try {
                xml.close();
            } catch (Throwable var49) {
                var2.addSuppressed(var49);
            }
        } else {
            xml.close();
        }
    }
}
}
}

```

XSSFDrawing

```
package org.apache.poi.xssf.usermodel;
public final class XSSFDrawing extends POIXMLDocumentPart implements Drawing<XSSFSimpleShape>    SpreadsheetML 绘图
public XSSFClientAnchor createAnchor(int dx1, int dy1, int dx2, int dy2, int col1, int row1, int col2, int row2) 创建锚点
public XSSFFTextBox createTextbox(XSSFClientAnchor anchor) 创建一个文本输入框
public XSSFSimpleShape createSimpleShape(XSSFClientAnchor anchor)    创建一个简单的形状
public XSSFConnector createConnector(XSSFClientAnchor anchor)        创建一个形状
public XSSFSimpleShapeGroup createGroup(XSSFClientAnchor anchor)    创建一个形状
public XSSFComment createCellComment(ClientAnchor anchor)    创建一个注释

package org.apache.poi.xssf.usermodel;
public final class XSSFConnector extends XSSFSimpleShape    形状
protected XSSFConnector(XSSFDrawing drawing, CTConnector ctShape)    创建一个形状
```

XSSFClientAnchor

```
package org.apache.poi.xssf.usermodel;
public class XSSFClientAnchor extends XSSFAutoShape implements ClientAnchor    客户端锚点
public XSSFClientAnchor(int dx1, int dy1, int dx2, int dy2, int col1, int row1, int col2, int row2) col1 和 row1 代表左上坐标。 col2 和 row2 代表右下坐标。
```

XSSFSimpleShape

```
package org.apache.poi.xssf.usermodel;
public class XSSFSimpleShape extends XSSFSimpleShape implements Iterable<XSSFFTextParagraph>, SimpleShape, TextContainer
简单形状
```

XSSFCell

```
package org.apache.poi.xssf.usermodel;
public final class XSSFCell extends CellBase          表示一个格子
public String getStringCellValue()                  获取格子的 String 值
public double getNumericCellValue()                获取格子的数字值
public void setCellComment(Comment comment)    设置格子注释
```

Usage

```
XSSFWorkbook workbook = new XSSSSFWorkbook( getClass().getResourceAsStream( "/V1.2_price.xlsx" ) );
```

```
CellStyle headerStyle = workbook.createCellStyle();
Font headerFont = workbook.createFont();
headerStyle.setFont(headerFont);
XSSFSheet xssfSheet = xssfWorkbook.getSheetAt(0); 读取子表
XSSFRow row = xssfSheet.getRow(0);                  默认第一行为标题行, index = 0
XSSFCell cell = row.createCell(i);
```

```
sheet.shiftRows(6, 12, -5);
从第 6 (在 excel 中指第 7 行) 行到第 12 (在 excel 中指第 13 行) 行全部向上移 5 行
```

```

//将第二行复制到第三行
XmlObject copy = twoRow.getCTRow().copy();
table.addRow(new XWPFTableRow((CTRow) copy, table));
twoRow.getCell(0).setText("第二行");
//获取复制的第三行
XWPFTableRow threeRow = table.getRow(2);
threeRow.getCell(0).setText("第三行");

```

最近用 POI 实现根据模板导出 excel，需要从中间行插入查询到的数据

但是用 creatRow 生成的数据会覆盖后面的模板内容

查了 API 没有找到插入行的方法

不过找到 shiftRows 方法将最后的空行移到需要插入行的位置，再用 createRow 生成

```

sheet.shiftRows(insertRowNum, sheet.getLastRowNum(), 1,true,false);
sheet.createRow(insertRowNum);

```

// 获取 cell 值

```

private Object getStringCellValue(XSSFCell cell) {
    String cell1Str1;
    if (cell.getCellType() == CellType.NUMERIC) {
        String format = cell.getCellStyle().getDataFormatString();
        if ("yyyy/mm/dd HH:mm".equals(format) || "m/d/yy h:mm".equals(format)      常用格式: "m/d/yy"
            || "yyyy-mm-dd".equals(format)
            || "yyyy-mm-dd HH:mm:ss".equals(format)) {
            return new SimpleDateFormat("yyyy-MM-dd HH:mm").format(cell.getDateCellValue());
        }
        return cell.getNumericCellValue();
    } else {
        cell1Str1 = cell.getStringCellValue();
    }
    return cell1Str1;
}

```

```

String path = "d:/测试/测试.xlsx";
createNewFile(path);
FileOutputStream fos = new FileOutputStream(path);
xssfWorkbook.write(fos);

```

获取 VMLDrawing 对象

```

private static XSSFVMLDrawing getVMLDrawing(XSSFSheet sheet) throws Exception {
    XSSFVMLDrawing drawing = null;
    if (sheet.getCTWorksheet().getLegacyDrawing() != null) {
        String legacyDrawingId = sheet.getCTWorksheet().getLegacyDrawing().getId();
        drawing = (XSSFVMLDrawing)sheet.getRelationById(legacyDrawingId);
    } else {
        int drawingNumber = sheet.getPackagePart().getPackage()
            .getPartsByContentType(XSSFRelation.VML_DRAWINGS.getContentType()).size() + 1;
        POIXMLDocumentPart.RelationPart rp =

```

```

        sheet.createRelationship(XSSFRelation.VML_DRAWINGS, XSSFFactory.getInstance(), drawingNumber, false);
        drawing = rp.getDocumentPart();
        String rId = rp.getRelationship().getId();
        sheet.getCTWorksheet().addNewLegacyDrawing().setId(rId);
    }
    return drawing;
}

```

Add Checkbox

<https://stackoverflow.com/questions/58733418/creating-a-checkbox-in-xlsx-using-apache-poi-java>

apache-fop

依赖:

```

<dependency>
    <groupId>org.apache.xmlgraphics</groupId>
    <artifactId>fop</artifactId>
    <version>2.6</version>
</dependency>

```

fop -xml param_test.xml -xsl param_test.xsl -pdf param_test.pdf -param position 1

Functions

函数 substring(\$data, \$counter, 1) concat(a, "​") length()

Structure Tags

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format" exclude-result-prefixes="fo">

```

<xsl:template match="/"> 匹配所有数据对象，匹配需转换的 xml 文件中的根结点

match=" ProductPdfInfo " 匹配指定数据对象】

<xsl:param name="data" select="0" /> 定义值变量

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"> 根节点

xml:lang="en" 语言

xmlns:fo="http://www.w3.org/1999/XSL/Format" 指定 fo 格式化】

<fo:layout-master-set> 页面模板列表

<fo:simple-page-master master-name="A4" page-height="29.7cm" page-width="21cm" margin-top="2cm" 定义一个页面

margin-bottom="2cm" margin-left="2cm" margin-right="2cm">

<fo:region-body/> body 容器

</fo:simple-page-master>

</fo:layout-master-set>

```

<fo:page-sequence master-reference="A4">          引用一个页面
    <fo:flow flow-name="xsl-region-body">      流动容器, xsl-region-body 对应的页面 simple-page-master
        内部的 body 容器
        <fo:static-content flow-name="xsl-region-body">
            <fo:inline-container>
                inline-progression-dimension="15.0%" 宽度占比
                content-height="scale-down-to-fit"
                vertical-align="center"
                <fo:inline>Test Block1</fo:inline>
            </fo:inline-container>
            <fo:block-container>
                block-progression-dimension="left" "auto" "12mm" "100%"
                block-progression-dimension.minimum="left" "auto" "12mm" "100%"
                <fo:block>Test Block2</fo:block>
                keep-together.within-page="always"
                <fo:block font-size="10pt">
                    <fo:table table-layout="fixed" width="100%" border-collapse="separate">
                        <fo:table-column column-width="4cm"/>
                        <fo:table-column column-width="4cm"/>
                        <fo:table-column
                            column-width="proportional-column-width(1)"
                            number-columns-repeated="6"      Specifies the repetition of a table-
                            column specification n times.
                        />

                        <fo:table-body>
                            <fo:table-row>
                                <fo:table-cell>
                                    <fo:block-container></fo:block-container>
                                </fo:table-cell>
                                <fo:table-cell>
                                    <fo:block-container></fo:block-container>
                                </fo:table-cell>
                            </fo:table-row>
                        </fo:table-body>
                    </fo:table>
                </fo:block>
            </fo:block-container>
        </fo:flow>
    </fo:page-sequence>
</fo:root>
</xsl:template>

```

</xsl:stylesheet>

Element Tags

<fo:external-graphic> 图片元素
 xf:alt-text="Citi Logo" 提示文字

<fo:list-item> 包含列表中的每个项目
<fo:list-item-label> 包含用于 list-item 的标签 - 典型地，包含一个数字或者字符的 <fo:block>
<fo:list-item-body> 包含 list-item 的内容/主体 - 典型地，一个或多个 <fo:block> 对象

<xsl:if test="(pdfGenerationRequest/accountReconPdfData)" 存在时才渲染内部内容
<xsl:for-each select="pdfGenerationRequest/myList"> 遍历
 <xsl:with-param name="achPage" select="achPage"> 引用元素对象中的字段值
 select=""<string-content>" pass string parameters
 <xsl:if test="position() !=last()" 索引不是最后一个时
 <fo:block page-break-before="always"></fo:block> 分页
 <xsl:if test="position() >=2" 索引大于等于 2 时
<xsl:choose> Choose one element
 <xsl:choose>
 <xsl:when test="expression">
 </xsl:when>
 <xsl:otherwise>
 </xsl:otherwise>
 </xsl:choose>

<xsl:value-of select="pdfGenerationRequest/form/accountText"> 引用 transform 中 src 的数据
<xsl:value-of select="contactNumber[phoneType='MOBILE'][position()='1']" > 选出 phoneType='MOBILE'的数据，并且是第一个
<xsl:value-of select="pdfGenerationRequest/form/ownerships[formater:checkOwnerType(ownerType/text(),
'ACCOUNT_SIGNER') =Yes'] [signerType='RESOLUTION_AUTHORITY'] > 调用 formater 的方法返回值

<xsl:apply-templates select="ProductPdflInfo" /> 转换节点内元素，如果文件内定义了匹配 ProductPdflInfo 的模板，则使用模板转换
<xsl:call-template name="account-reconciliation"> 引入其他 template (默认继承之前的页面参数)
 <xsl:with-param name="achPage" select="achPage"> 参数作为页面对象传入到 template 中，之后可以通过 select 直接选出，而参数是需要添加符号"\$xxx"
<fo:instream-foreign-object> 调整用于内联图形或 "generic" 类对象，该对象的内容尺寸大小是通过调用对象的 size 属性来定义的
 <fo:instream-foreign-object
 xf:alt-text="No checkbox unchecked">
 <svg:svg width="5" height="5">
 <svg:rect width="5" height="5" style="stroke:black; fill:none;" />
 </svg:svg>
 </fo:instream-foreign-object>

Display Attributes

background-color="#f00"
border="1px solid green" 边框
border-right="1px solid green" 边框

Position Attributes

```

position="absolute"
left="7mm"
top="7mm"

margin="0 1 2 3"
margin-bottom="1.5cm"    下边距
text-align="center"        中间对齐
space-before="5mm"
space-after="5mm"         space before 和 space after 是 block 与 block 之间起分割作用的空白。

```

Usage

```

//A. Public data.
File baseDir = new File(".");
File outDir = new File(baseDir, "out");
outDir.mkdirs();

// Setup input and output files(you can convert data object to xmlfile).
File xmlfile = new File(baseDir, "xml/xml/projectteam.xml");
File xsldfile = new File(baseDir, "xml/xslt/projectteam2fo.xsl");
File pdffile = new File(outDir, "ResultXML2PDF.pdf");

// Configure fopFactory as desired.
final FopFactory fopFactory = FopFactory.newInstance(new File(".").toURI());
FOUserAgent foUserAgent = fopFactory.newFOUserAgent();
foUserAgent.setTitle("test");

// Setup output
OutputStream out = new java.io.FileOutputStream(pdffile);
out = new java.io.BufferedOutputStream(out);

try {
    // Construct fop with desired output format.
    Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF, foUserAgent, out);

    // Setup XSLT file
    TransformerFactory factory = TransformerFactory.newInstance();
    Transformer transformer = factory.newTransformer(new StreamSource(xsldfile));

    // Set the value of a <param> in the stylesheet (optional).
    transformer.setParameter("versionParam", "2.0");

    // Setup input for XSLT transformation (xmlfile or data object)
    Source src = new StreamSource(xmlfile);

    // Resulting SAX events (the generated FO) must be piped through to FOP
    Result res = new SAXResult(fop.getDefaultHandler());

    // Start XSLT transformation and FOP processing
    transformer.transform(src, res);
} finally {
    out.close();
}

```

opencsv

依赖:

```

<!-- https://mvnrepository.com/artifact/com.opencsv/opencsv -->
<dependency>
    <groupId>com.opencsv</groupId>

```

```
<artifactId>opencsv</artifactId>
<version>5.6</version>
</dependency>
```

源码解析 opencsv5.6

```
package com.opencsv;
public class CSVReader implements Closeable, Iterable<String[]>          csv 文件阅读器
public CSVReader(Reader reader)
public List<String[]> readAll() throws IOException, CsvException
public String[] readNext() throws IOException, CsvValidationException      读取所有数据到 List 中，内部也是使用迭代器
                                                               读取下一行的数据数组
```

easyexcel

依赖:

```
<!-- https://mvnrepository.com/artifact/com.alibaba/easyexcel -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>easyexcel</artifactId>
    <version>3.1.1</version>
</dependency>
```

[easyexcel-core]

ReadListener

```
package com.alibaba.excel.read.listener;
public interface ReadListener<T> extends Listener
void invoke(T var1, AnalysisContext var2);                                Read every piece of data.
void doAfterAllAnalysed(AnalysisContext var1);                          All data parsing is complete.
```

AnalysisEventListener

```
package com.alibaba.excel.event;
public abstract class AnalysisEventListener<T> implements ReadListener<T>
public void invokeHeadMap(Map<Integer, String> headMap, AnalysisContext context)  Read table head.
```

annotation
(annotation)

ExcelProperty

```
package com.alibaba.excel.annotation;
public @interface ExcelProperty      Mark the XML read column for the Java class.
String[] value() default {""];   Column Title
int index() default -1;           Column index
```

freemarker

概念

模板引擎

[freemarker]

Configuration

```
package freemarker.template;
```

```

public class Configuration extends Configurable implements Cloneable, ParserConfiguration 核心配置类
public static final Version VERSION_2_3_0; 提供版本选择
public static final Version VERSION_2_3_19;
public static final Version VERSION_2_3_20;
public static final Version VERSION_2_3_21;
public static final Version VERSION_2_3_22;
public static final Version VERSION_2_3_23;
public static final Version VERSION_2_3_24;
public static final Version VERSION_2_3_25;
public static final Version VERSION_2_3_26;
public static final Version VERSION_2_3_27;
public static final Version VERSION_2_3_28;
public static final Version VERSION_2_3_29;
public static final Version VERSION_2_3_30;
public static final Version VERSION_2_3_31;
public static final Version DEFAULT_INCOMPATIBLE_IMPROVEMENTS;

public Configuration(Version incompatibleImprovements) 根据版本构造
public void setClassForTemplateLoading(
    Class resourceLoaderClass, 资源加载类 // getClass(),
    String basePackagePath resource 相对路径 // "/email-template/"
)
public void setDefaultEncoding(String encoding) 设置默认编码
public void setTemplateExceptionHandler(TemplateExceptionHandler templateExceptionHandler) 设置模板异常处理方式

```

TemplateExceptionHandler

```

package freemarker.template;
public interface TemplateExceptionHandler 异常处理方式
TemplateExceptionHandler IGNORE_HANDLER
TemplateExceptionHandler RETHROW_HANDLER 重新抛出
TemplateExceptionHandler DEBUG_HANDLER
TemplateExceptionHandler HTML_DEBUG_HANDLER

```

xalan

[xalan]

ThreadControllerWrapper

```

package org.apache.xml.utils;
public class ThreadControllerWrapper
private static ThreadController m_tpool = new ThreadController();
public static Thread runThread(Runnable runnable, int priority) {
    return m_tpool.run(runnable, priority);
}

```

```

public static class ThreadController
    public ThreadController() {
    }

    public Thread run(Runnable task, int priority) {
        Thread t = new Thread(task);
        t.start();
    }
}

```

```

        return t;
    }

    public void waitThread(Thread worker, Runnable task) throws InterruptedException {
        worker.join();
    }
}

```

snakeyaml

[snakeyaml]

Yaml

```

package org.yaml.snakeyaml;
public class Yaml

```

For custom list type, you can define a class that extends the ArrayList class to load a single parameter.

```

public Yaml()
public Yaml(BaseConstructor constructor)
    // Yaml yaml = new Yaml(new Constructor(Student.class));
    // Yaml smpYml = new Yaml(new CustomClassLoaderConstructor(SmpYmlProperties.class,
    SmplInit.class.getClassLoader()));
public <T> T load(String yaml)
    // Map<String, Object> map = (Map)( (Map)config ).get("spring");
    // Map<String, Object> load = smpYml.load(configPath);
    // Object ret = yaml.load("name: jerry");
    // DataSource ret = yaml.loadAs(this.getClass().getClassLoader().getResourceAsStream("source.yml"),
    DataSource.class);
public <T> T load(InputStream io)
public <T> T load(Reader io)

```

CustomClassLoaderConstructor

```

package org.yaml.snakeyaml.constructor;
public class CustomClassLoaderConstructor extends Constructor    specify ClassLoader for the Constructor

```

dom4j

```

<dependency>
    <groupId>org.dom4j</groupId>
    <artifactId>dom4j</artifactId>
</dependency>
<!-- https://mvnrepository.com/artifact/jaxen/jaxen -->
<dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>2.0.0</version>
</dependency>

```

Usage

Read

```

import java.net.URL;

```

```
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.SAXReader;

public class Foo {

    public Document parse(URL url) throws DocumentException {
        SAXReader reader = new SAXReader();
        Document document = reader.read(url);
        return document;
    }
}
```

Write

```
import org.dom4j.Document;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.XMLWriter;

public class Foo {

    public void write(Document document) throws IOException {

        // lets write to a file
        try (FileWriter fileWriter = new FileWriter("output.xml")) {
            XMLWriter writer = new XMLWriter(fileWriter);
            writer.write( document );
            writer.close();
        }

        // Pretty print the document to System.out
        OutputFormat format = OutputFormat.createPrettyPrint();
        writer = new XMLWriter(System.out, format);
        writer.write( document );

        // Compact format to System.out
        format = OutputFormat.createCompactFormat();
        writer = new XMLWriter(System.out, format);
        writer.write(document);
        writer.close();
    }
}

// 创建一个 Document 实例
Document doc = DocumentHelper.createDocument();

// 添加根节点
Element root = doc.addElement("root");

// 在根节点下添加第一个子节点
Element oneChildElement= root.addElement("person").addAttribute("attr", "root noe");
oneChildElement.addElement("people")
    .addAttribute("attr", "child one")
```

```

    .addText("person one child one");
oneChildElement.addElement("people")
    .addAttribute("attr", "child two")
    .addText("person one child two");

// 在根节点下添加第一个子节点
Element twoChildElement= root.addElement("person").addAttribute("attr", "root two");
twoChildElement.addElement("people")
    .addAttribute("attr", "child one")
    .addText("person two child one");
twoChildElement.addElement("people")
    .addAttribute("attr", "child two")
    .addText("person two child two");

// xml 格式化样式
// OutputFormat format = OutputFormat.createPrettyPrint(); // 默认样式

// 自定义 xml 样式
OutputFormat format = new OutputFormat();
format.setIndentSize(2); // 行缩进
format.setNewlines(true); // 一个结点为一行
format.setTrimText(true); // 去重空格
format.setPadText(true);
format.setNewLineAfterDeclaration(false); // 放置 xml 文件中第二行为空白行

// 输出 xml 文件
XMLWriter writer = new XMLWriter(new FileOutputStream(file), format);
writer.write(doc);
System.out.println("dom4j CreateDom4j success!");

```

Note

最近在研究 XBRL GL 的有关内容，在项目中要求吧 XBRL GL 导入到 11179 注册库中，根据 11179 建立数据库，然后从 XBRL GL 分类标准中导入数据到数据库。在导入过程中需要用到 dom4j 来读取 XBRL GL 文件，用 selectnodes 来选取制定的元素，发现总是空值，查看 XPATH 也没有错，着实困扰了很长时间。后来发现，原来是 xmlns 在作怪，把 xml 文件开头的 xmlns 属性去掉，一切 OK!

另外，如果要选取当前节点的子节点，要用“./节点的相对路径”才能完成此任务。

可能是因为 XML 文件带有命名空间.比如

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

因为 **dom4j** 不能识别带命名空间的节点，所以在读取带命名空间的 XML 时，要在每个节点前加上命名空间，方法如下例子：

```

Document doct=reader.read(xmlFile);
    HashMap nsMap=new HashMap();
    nsMap.put("ns","http://java.sun.com/xml/ns/jbi");      //加入命名空间
    //获取子节点
    XPath xsub=doct.createXPath("//ns:title");

```

```
xsub.setNamespaceURIs(nsMap);
Element element = (Element) xsub.selectSingleNode(doc);
```

参考

dom4j 解析带命名空间的 xml 文件的节点的例子，只要给 XPath 设置默认的命名空间就行了

```
SAXReader saxReader = new SAXReader();
HashMap nsMap = new HashMap();
nsMap.put("pom", "http://maven.apache.org/POM/4.0.0");
Document readPomDocument = saxReader.read(writePomPath);
Document appendPomDocument = saxReader.read(appendXmlPath);
//A. foreach replace in append.xml
List<Node> replaceNodeList = appendPomDocument.selectNodes("/root/replace");
for (Node replaceNode : replaceNodeList) {
    if(replaceNode.getNodeType() != Node.ELEMENT_NODE) continue;
    Element replaceElement = (Element) replaceNode;
    String replaceNodeXpath = replaceElement.getAttribute("xpath");
    XPath replaceNodeXpathSecond = readPomDocument.createXPath("/pom:project/pom:dependencies");
    replaceNodeXpathSecond.setNamespaceURIs(nsMap);
    List<Node> pomCheckParentNodeList = replaceNodeXpathSecond.selectNodes(readPomDocument);
    //B. foreach ele in append.xml
```

```
private static final class NameSpaceCleaner extends VisitorSupport {
    public void visit(Document document) {
        ((DefaultElement) document.getRootElement())
            .setNamespace(Namespace.NO_NAMESPACE);
        document.getRootElement().additionalNamespaces().clear();
    }
    public void visit(Namespace namespace) {
        namespace.detach();
    }
    public void visit(Attribute node) {
        if (node.toString().contains("xmlns")
            || node.toString().contains("xsi:")) {
            node.detach();
        }
    }
    public void visit(Element node) {
        if (node instanceof DefaultElement) {
            ((DefaultElement) node).setNamespace(Namespace.NO_NAMESPACE);
        }
    }
}
```

修改和删除 xmlns 属性是无效的

Clone And Synchronize

```
DefaultElement clone = (DefaultElement)firstElement.clone();
clone.setNamespace(readPomDocument.getRootElement().getNamespace());
[dom4j]
```

DocumentHelper

```
package org.dom4j;
public final class DocumentHelper
public static Document createDocument()
```

Document

```
package org.dom4j;
public interface Document extends Branch
Document addComment(String var1);
Element getRootElement();
void setRootElement(Element var1);
```

Element

```
package org.dom4j;
public interface Element extends Branch
Element addAttribute(String var1, String var2);
Element addAttribute(QName var1, String var2);
Iterator<Element> elementIterator();
List<Element> elements();
    List elements = e.getParent().elements();
    elements.add(elements.indexOf(e), lr);
Element addText(String var1);
String attributeValue(String var1);
boolean remove(Namespace var1);
```

Node

```
package org.dom4j;
public interface Node extends Cloneable
List<Node> selectNodes(String var1);          // List<Node> list = document.selectNodes("//foo/bar");
Node selectSingleNode(String var1);           // document.selectSingleNode("//foo/bar/author");
String valueOf(String var1);                  // String name = node.valueOf("@name");
```

SAXReader

```
package org.dom4j.io;
public class SAXReader
public Document read(File file) throws DocumentException
public Document read(String systemId) throws DocumentException
```

Introduction

SLF4J serves as a simple facade or abstraction for various logging frameworks.

It allows applications to bind with different logging frameworks at deployment time, without requiring changes to the application code.

[slf4j-api]

Logger

```
public interface Logger  
void info(String var1, Object var2);
```

MDC

package org.slf4j;
public class MDC
(Mapped Diagnostic Context) in SLF4J (Simple Logging Facade for Java) allows you to store contextual information specific to a thread.

This information can be added to log messages, enhancing them with details like session IDs or user IDs for better debugging and tracing capabilities without altering the log message format itself.

MDC is thread-local, ensuring each thread has its own isolated context map.

LoggerFactory

```
package org.slf4j;  
public final class LoggerFactory 日志工厂  
public static Logger getLogger(String name) 获取日志对象  
public static Logger getLogger(Class<?> clazz) 获取日志对象
```

logback

Core

Introduction

Logback is a logging framework that is both fast and flexible, designed as a successor to the popular Log4j framework. It implements the [SLF4J API](#) and serves as the default logging implementation in many Spring Boot applications.

Dependency

```
<dependency>  
  <groupId>ch.qos.logback</groupId>  
  <artifactId>logback-classic</artifactId> <!-- Include Logback Classic for logging -->  
</dependency>
```

Configuration

依赖: spring-boot-starter-logging (spring-boot-starter 内已经包含)

```
<dependency>  
  <groupId>ch.qos.logback</groupId>  
  <artifactId>logback-classic</artifactId>  
  <version>1.2.11</version>  
</dependency>
```

兼容: 与 Alibaba Nacos 中集成的 Logback 框架冲突, 不能自定义 logback-sdks.xml 的名称, 只能使用 logback-spring.xml

vm option >>

-Dspring.output.ansi.enabled=ALWAYS 彩色输出

application.yml >>

logging:

```
config: classpath:logback-dev.xml #指定 logback 日志文件 (默认 logback-spring.xml)
```

```
level:  
root: info  
org.mybatis: debug  
java.sql: debug  
org.springframework.web: debug  
org.hibernate.SQL=DEBUG  
org.hibernate.type.descriptor.sql.BasicBinder: TRACE  
org.springframework.jdbc.core.JdbcTemplate: DEBUG  
org.springframework.jdbc.core.StatementCreateUtils: TRACE
```

logback-smp.xml >>

```
<?xml version="1.0" encoding="UTF-8"?>  
<configuration  
    scan="true"          配置文档如果发生改变，将会被重新加载，默认值为 true  
    scanPeriod="10 seconds" 监测配置文档修改的时间间隔，默认单位是毫秒（当 scan 为 true 时，此属性生效。默认的时间  
    间隔为 1 分钟）  
    debug="false"        打印 logback 内部日志。默认值为 false。  
    >  
  
<springProperty name="LOG_PATH" scope="context" source="sdklog.logpath" default="../logs"/>      <!--  
application.properties 内定义的值会被插入到 logger 上下文中，也可以使用 -->  
    <springProperty name="APP_NAME" scope="context" source="spring.application.name"/>  
    <springProperty name="SERVER_IP" scope="context" source="spring.cloud.client.ip-address" defaultValue="0.0.0.0"/>  
    <springProperty name="SERVER_PORT" scope="context" source="server.port" defaultValue="0000"/>  
  
[删除]  <property name="CONSOLE_LOG_PATTERN" value="${CONSOLE_LOG_PATTERN:-%clr(%d{yyyy-MM-dd  
HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(--  
-){faint} %clr([%15.15t])}{faint} %clr(%  
40.40logger{39}){cyan} %clr():{faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>      <!-- 参考彩色日志格式  
配置 -->  
  
    <property name="CONSOLE_LOG_PATTERN" value="%clr(%date{yyyy-MM-dd  
HH:mm:ss.SSS}){blue} %clr(-){faint}%clr(%5level) %clr(${PID:- }){magenta} %clr(--  
-){faint} %clr([%thread])}{yellow} %clr(%logger){cyan}%clr():{faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>  
<!-- 自定义彩色日志格式配置，自定义属性，可以使"${}"来使用变量， faint blue yellow magenta orange cyan -->  
[删除]<property name="CONSOLE_LOG_PATTERN"  
value="[${APP_NAME}{yellow}:%clr(${SERVER_IP}){cyan}:%clr(${SERVER_PORT}){cyan}] - %clr(%date{yyyy-MM-dd  
HH:mm:ss.SSS}){blue} %clr(-){faint}%clr(%5level) %clr(${PID:- }){magenta} %clr(--  
-){faint} %clr([%thread])}{yellow} %clr(%logger){cyan} %clr():{faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>  
<!-- 自定义彩色日志格式配置， 详细版 -->  
[删除]<property name="CONSOLE_LOG_PATTERN"  
value="[${APP_NAME}{yellow}:%clr(${SERVER_IP}){cyan}:%clr(${SERVER_PORT}){cyan}] - %clr(%date{yyyy-MM-dd  
HH:mm:ss.SSS}){blue} %clr(-){faint}%clr(%5level) %clr(${PID:- }){magenta} %clr(--){faint} %clr([%-  
15.15thread])}{yellow} %clr(%-40.40logger){cyan} %clr():{faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>  
<!-- 自定义彩色日志格式配置， 截断长度 -->  
  
<property name="FILE_LOG_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS} - ${APP_NAME} - %-5level ---
```

```

[%thread] %logger{50} - %msg%n "/>      <!-- 自定义文件日志格式配置 -->

<contextName>logback</contextName>

<conversionRule conversionWord="clr" converterClass="org.springframework.boot.logging.logback.ColorConverter" />
<!--日志格式和颜色渲染，彩色日志依赖的渲染类 -->
<conversionRule conversionWord="wex"
converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />
<conversionRule conversionWord="wEx"
converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter" />

<!-- %date 时间格式 %d -->
<!-- %level 日志级别 %p -->
<!-- %thread 日志所在线程名-->
<!-- %logger 包名，会缩短简写-->
<!-- %msg 应用程序提供的信息-->
<!-- %n 平台的换行符\n 或\r\n-->
<!-- -%20.30logger 短于 20 个字符在左侧填充空格。超过 30 个字符从开头截断。负号表示向左靠齐-->
<!-- %logger{26} 超过 26 的包名会被缩写 -->

<!-- ${testKey:-%t } 输出 testKey 所对应的 value, 默认为-%t-->

<!-- gray 灰色 white 白色 yellow 黄色 green 绿色 red 红色 black 黑色 magenta 洋红 boldMagenta 加粗洋红 cyan 青色 -->

<appender name="CONSOLE_LOG" class="ch.qos.logback.core.ConsoleAppender">          <!--输出到控制台，此日志 appender 是为开发使用，只配置最底级别，控制台输出的日志级别是大于或等于此级别的日志信息-->
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>debug</level>
    </filter>
    <encoder>
        <Pattern>${CONSOLE_LOG_PATTERN}</Pattern>
        <charset>UTF-8</charset>          <!-- 设置字符集 -->
    </encoder>
</appender>

<appender name="FILE_LOG" class="ch.qos.logback.core.rolling.RollingFileAppender">      <!--输出到文档，每天产生一个文件， level 为 DEBUG 日志-->
    <!--<file>${LOG_PATH}/sdk-${APP_NAME}.log</file>          <!--&lt;!&ndash; 输出日志路径，配置滚动日志文件时，当前的日志输出到此文件，明天之后的日志输出到滚动日志 pattern 文件&ndash;&gt;-->
    <encoder>          <!--日志文档输出格式-->
        <pattern>${FILE_LOG_PATTERN}</pattern>
        <charset>UTF-8</charset>          <!-- 设置字符集 -->
    </encoder>

```

```

</encoder>
<rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">           <!-- 基于时间和大
小的滚动策略-->
    <fileNamePattern>${LOG_PATH}/sdklog-${APP_NAME}-%d{yyyy-MM-dd}.%i.log</fileNamePattern>           <!--
日志按照天滚动 -->
        <!-- clp.%i.log.zip      后缀以".zip"或".gz"结尾，则开启日志文件压缩 -->
        <!-- %d 决定以什么时间维度轮转(但实际轮转时机取决于日志事件的到达时间)，比如%d{yyyy/MM}:每个月开始的时候轮
转,%d 默认为 yyyy-MM-dd: 按天轮转 -->
        <!-- %i 为文件按照 maxFileSize 大小规定轮转后的序号 -->
    <maxFileSize>100MB</maxFileSize>           <!--单个日志文件最大大小，当文件达到该大小则触发截
断 (以及压缩) -->
    <maxHistory>60</maxHistory>           <!--日志文件保留最大时间滚动周期，比如当
fileNamePattern 中%d 以为 dd 结尾时，则保留 60 天-->
</rollingPolicy>
<!--<filter class="ch.qos.logback.classic.filter.LevelFilter">           &lt;!&ndash; 此日志文档只记录 debug 级别的
&ndash;&gt;-->
    <!--<level>DEBUG</level>-->
    <!--<onMatch>ACCEPT</onMatch>-->
    <!--<onMismatch>DENY</onMismatch>-->
<!--</filter>-->
<filter class="ch.qos.logback.classic.filter.ThresholdFilter">           <!-- 过滤掉低于 INFO 级别的日志，记录等于或高于 INFO
的日志-->
    <level>INFO</level>
</filter>
    <!-- 日志级别从低到高分为 TRACE < DEBUG < INFO < WARN < ERROR < FATAL，如果设置为 WARN，则低于
WARN 的信息都不会输出 -->
</appender>

```

[修改] <**springProfile** name="dev"> <!--指定开发环境生效的 appender，打印控制台，springProfile 可省 -->

```

<root level="info">
    <appender-ref ref="CONSOLE_LOG" />
    <appender-ref ref="FILE_LOG" />
</root>
</springProfile>

```

[删除] <logger name="org.hibernate" level="WARN"/> <!-- 用来设置某一个包或者具体的某一个类的日志打印级别（非
必须） -->

```

<logger name="org.springframework" level="WARN"/>
<logger name="com.opensymphony" level="WARN"/>
<logger name="org.apache" level="WARN"/>
<logger name="com.zhongping.mybatis.mapper.support" level="debug" additivity="false">
    <appender-ref ref="LOGS" />
    <appender-ref ref="STDOUT" />
</logger>
<logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="WARN"/>
<logger name="org.hibernate.type.descriptor.sql.BasicExtractor" level="DEBUG"/>
<logger name="org.hibernate.SQL" level="INFO"/>

```

```

<logger name="org.hibernate.engine.QueryParameters" level="INFO"/>
<logger name="org.hibernate.engine.query.HQLQueryPlan" level="INFO"/>

<logger name="org.eclipse.jetty" level="WARN"/>
<logger name="org.jboss.netty" level="INFO"/>
    <!-- name 用来指定受此 logger 约束的某一个包或者具体的某一个类 -->
    <!--level 用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF，还有一个特殊值 INHERITED 或者同义词 NULL，代表强制执行上级的级别。
        如果未设置此属性，那么当前 logger 将会继承上级的级别。 -->
    <!-- addtivity 是否向上级 logger 传递打印信息。默认是 true -->
</configuration>

```

[logback-core]

LayoutBase

```

package ch.qos.logback.core;
public abstract class LayoutBase<E> extends ContextAwareBase implements Layout<E>

```

log4j
Core

Log4j is another widely-used logging framework in Java.

Versions up to Log4j 1.x use the log4j artifact, and it can be integrated with SLF4J using the slf4j-log4j12 binding.
Log4j 2.x has a different architecture and can be integrated directly with SLF4J.

Dependency

Log4j 1.x

```

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.8.0-beta4</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.8.0-beta4</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>

```

Log4j 2.x

```

<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId> <!-- Include SLF4J bridge for Log4j2 -->
    <version>2.x.x</version> <!-- Replace with the version of Log4j 2.x -->
</dependency>

```

log4j.properties >> 自动加载根目录此资源文件

```
#日志级别: DEBUG < INFO < WARN < ERROR < FATAL
log4j.rootLogger=info,stdout,file
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}]-5p %c(line:%L) %x-%m%
```

```
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=../logs/isk.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}]-5p %c(line:%L) %x-%m%
```

```
private static final logger LOGGER = LoggerFactory.getLogger(MyClass.class);
```

PatternLayout

%d{pattern}

Date and time of the log event. You can specify a format, such as %d{yyyy-MM-dd HH:mm:ss.SSS}.

%d{yyyy-MM-dd HH:mm:ss.SSS} Logs the date and time in the specified format.

%p / %level

The log level (e.g., DEBUG, INFO, WARN, ERROR).

%-5level Logs the level of the message, left-aligned with a width of 5 characters.

%c{length} / %logger{36}

The name of the logger (e.g., the class name), where length specifies how many characters of the logger name to include.

%logger{36} Logs the logger name, truncated to the last 36 characters.

%m / %msg

The **actual log message**.

%n

A platform-independent **line separator**.

%X{key}

Retrieve a value from the MDC (Mapped Diagnostic Context) by key, which can be useful for including trace IDs and span IDs in your logs

%X{traceId:-} Retrieves the trace ID from the MDC; if not available, it defaults to a hyphen.

%X{spanId:-} Retrieves the span ID from the MDC with the same fallback.

Configuration

Add Dependencies

Using Log4j in a Spring Boot project is a straightforward process. Below are the steps to set up and use Log4j (specifically Log4j 2) for logging in a Spring Boot application:

For Maven:

```
<dependencies>
    <!-- Spring Boot Starter Web (or any other starter) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Log4j 2 dependencies -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-log4j2</artifactId>
    </dependency>
```

```

<!-- Optional: If you want to add Log4j 2 XML configuration support -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
</dependency>
</dependencies>

```

For Gradle:

```

// Log4j 2 dependencies
implementation 'org.apache.logging.log4j:log4j-slf4j-impl' // Use Log4j2 SLF4J binding
implementation 'org.apache.logging.log4j:log4j-core' // Core Log4j2 dependency
// Test dependencies
testImplementation 'org.slf4j:slf4j-api' // SLF4J API for testing
testImplementation 'org.apache.logging.log4j:log4j-slf4j-impl' // SLF4J binding for Log4j

```

```

dependencies {
    // Spring Boot Starter Web (or any other starter)
    implementation 'org.springframework.boot:spring-boot-starter-web'

    // Log4j 2 dependencies
    implementation 'org.springframework.boot:spring-boot-starter-log4j2'

    // Optional: If you want to add Log4j 2 XML configuration support
    implementation 'org.apache.logging.log4j:log4j-core'
}

```

Create Log4j2 Configuration File

Create a configuration file for Log4j2. You can use either XML, JSON, or YAML format. The configuration file should be placed in the `src/main/resources` directory.

`log4j2.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!--
        Configuration element: The root element for the Log4j2 configuration.
        The "status" attribute controls internal Log4j2 logging.
        Values: "OFF", "ERROR", "WARN", "INFO", "DEBUG", "TRACE".
    -->

    <Properties>
        <!--
            Properties element: Used to define key-value pairs that can be referenced in the configuration.
            You can reference properties using ${propertyName}.
        -->
        <Property name="logPath">logs</Property>
    </Properties>

    <Appenders>
        <!--
            Appenders element: Defines where the log messages will be sent.
            Common types: Console, File, RollingFile, etc.
        -->
        <Console name="ConsoleAppender" target="SYSTEM_OUT">
            <!--
                Console Appender: Writes logs to the console.
                "target" specifies the output stream: SYSTEM_OUT (default) or SYSTEM_ERR.
            -->
            <PatternLayout>
                <!-- PatternLayout: Defines the format of the log message. -->

```

```

        <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%t] %-5p %c{1} - %m%n</Pattern>
    </PatternLayout>
</Console>

<File name="FileAppender" fileName="${logPath}/app.log">
    <!--
        File Appender: Writes logs to a specified file.
        "fileName" specifies the location of the log file.
    -->
    <PatternLayout>
        <!-- Reuse the same pattern layout as the console appender. -->
        <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%t] %-5p %c{1} - %m%n</Pattern>
    </PatternLayout>
</File>

<RollingFile name="RollingFileAppender" fileName="${logPath}/rolling_app.log"
              filePattern="${logPath}/rolling_app-%d{yyyy-MM-dd}.log.gz">
    <!--
        RollingFile Appender: Creates a new log file when certain conditions are met.
        "filePattern" defines the naming pattern for rolled-over log files.
        Log files can also be compressed, like ".gz" here.
        - name: A unique identifier for the appender.
        - fileName: The main log file where log events are initially written.
        - filePattern: The naming pattern for the rolled files, utilizing placeholders for date and
index.
    -->
    <PatternLayout>
        <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%t] %-5p %c{1} - %m%n</Pattern>
    </PatternLayout>
    <Policies>
        <!--
            Policies element: Defines the conditions that trigger the log rollover.
            Common policies include TimeBasedTriggeringPolicy and SizeBasedTriggeringPolicy.
        -->
        <TimeBasedTriggeringPolicy interval="1" modulate="true"/>
        <SizeBasedTriggeringPolicy size="10MB" />
        <!--
            TimeBasedTriggeringPolicy: Rolls over logs based on time intervals.
            - interval: Number of time units (days) for rolling.
            - modulate: Ensures rollover aligns with the start of the time period (e.g., midnight).
            SizeBasedTriggeringPolicy: Rolls over logs when they exceed the specified size (e.g.,
10MB).
            - size: The maximum size of the log file before it rolls over.
        -->
    </Policies>
</RollingFile>
<RollingRandomAccessFile name="RollingRandomAccessFileAppender"
                         fileName="logs/app.log"
                         filePattern="logs/app-%d{yyyy-MM-dd}-%i.log.gz">
    <!--
        RollingRandomAccessFile: An appender that writes log events to a file with rolling
capabilities.
    -->
    <PatternLayout>
        <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%t] %-5p %c{1} - %m%n</Pattern>
    </PatternLayout>
    <Policies>
        <TimeBasedTriggeringPolicy interval="1" modulate="true"/>
        <SizeBasedTriggeringPolicy size="10MB"/>
    </Policies>

    <DefaultRolloverStrategy max="5" fileIndex="min"/>
    <!--
        DefaultRolloverStrategy: Manages the rollover behavior of log files.
        - max: Maximum number of backup log files to retain.
        - fileIndex: Determines the naming scheme for rolled-over files; "min" indicates incremental
naming.
        - stopCustomActionsOnError: whether to stop executing asynchronous actions if an error occurs
    -->

```

```

-->
<Delete basePath="logs" maxAge="30"/>
<!--
    Delete: A policy for automatically removing old log files.
    - basePath: The directory where the log files are located.
    - maxAge: Specifies the age (in days) of log files to be deleted.
-->
<IfFileName>
<!--
    IfFileName: A condition that evaluates whether a file name matches certain criteria.
-->
<FileNamePattern>app-.*\.log\.gz</FileNamePattern>
<!--
    FileNamePattern: Specifies the regex pattern for matching file names for conditional
operations.
-->
</IfFileName>

<IfLastModified>
<!--
    IfLastModified: A condition that checks if the file was last modified within a
certain timeframe.
-->
<MaxAge>30</MaxAge>
<!--
    MaxAge: Specifies the maximum age (in days) of a file to consider it for operations.
-->
</IfLastModified>
</Delete>
</DefaultRolloverStrategy>
</RollingRandomAccessFile>
</Appenders>

<Loggers>
<!--
    Loggers element: Defines different loggers for various packages or classes.
    Can specify different log levels and appenders.
-->

<Root level="info">
<!--
    Root logger: Captures all log events that don't match a specific logger.
    The "level" attribute defines the minimum log level (e.g., info, debug, error).
-->
<AppenderRef ref="ConsoleAppender" />
<AppenderRef ref="FileAppender" />
</Root>

<Logger name="com.example" level="debug" additivity="false">
<!--
    Custom logger: Logs specific to the "com.example" package.
    - name: The logger's name, typically matching a specific package or class.
    - level: The minimum log level for this logger.
    - additivity: determines whether log events are also passed to parent loggers.
-->
<AppenderRef ref="RollingFileAppender" />
</Logger>
<AsyncLogger name="com.example" level="info">
<!--
    AsyncLogger: An asynchronous logger that improves performance by using a separate thread for
logging.
-->
</AsyncLogger>
</Loggers>
</Configuration>
Example: log4j2.xml
<?xml version="1.0" encoding="UTF-8"?>
```

```

<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1} - %m%n"/>
        </Console>
        <File name="FileLogger" fileName="logs/app.log">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1} - %m%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
            <AppenderRef ref="FileLogger"/>
        </Root>
    </Loggers>
</Configuration>

```

Customize Log Levels

logging:

level:

root: INFO

com.example: DEBUG

log4jdbc

Introduction

log4jdbc is an extension library that enables logging of SQL statements executed by JDBC drivers.

It works with [SLF4J](#) and various logging frameworks like [Log4j](#) and [Logback](#).

Dependency

```

<dependency>
    <groupId>com.googlecode.log4jdbc</groupId>
    <artifactId>log4jdbc</artifactId>
    <version>1.2</version>
    <scope>runtime</scope>
</dependency>

```

Configuration

application.yml >>

```

spring:
  datasource:
    url:
      jdbc:mysql://${boss.datasource.ip}:${boss.datasource.port}/${boss.datasource.database}?useUnicode=true&characterEncoding=utf8&useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=GMT%2B8
      username: ${boss.datasource.username}
      password: ${boss.datasource.password}
    #   driver-class-name: com.mysql.cj.jdbc.Driver
    #   driver-class-name: net.sf.log4jdbc.sql.jdbcapi.DriverSpy

```

log4jdbc.log4j2.properties >>

```

# If you use SLF4J. First, you need to tell log4jdbc-log4j2 that you want to use the SLF4J logger
# mapper 根路径
log4jdbc.debug.stack.prefix=com.project.system.api.mapper
log4jdbc.spylogdelegator.name=net.sf.log4jdbc.log.slf4j.Slf4jSpyLogDelegator
log4jdbc.auto.load.popular.drivers=false
log4jdbc.drivers=com.mysql.cj.jdbc.Driver
log4jdbc.dump.booleanastruefalse=true
log4jdbc.dump.sql.addsemicolon=true

```

```

logback-boss.xml >>
<!-- 监控 sql 日志输出-->
<!-- 如想看到表格数据, 将 OFF 改为 INFO -->
<logger name="jdbc.resultsettable" level="INFO" additivity="false">
    <appender-ref ref="CONSOLE_LOG"/>
</logger>

<!-- 包含 SQL 语句实际的执行时间 及 sql 语句 (与 jdbc.sqlonly 功能重复) -->
<logger name="jdbc.sqltiming" level="OFF" additivity="false">
    <appender-ref ref="CONSOLE_LOG"/>
</logger>

<!-- 仅仅记录 SQL 语句, 会将占位符替换为实际的参数-->
<logger name="jdbc.sqlonly" level="INFO" additivity="false">
    <appender-ref ref="CONSOLE_LOG"/>
</logger>

<!-- 包含 ResultSet 的信息, 输出篇幅较长 -->
<logger name="jdbc.resultset" level="OFF" additivity="false">
    <appender-ref ref="CONSOLE_LOG"/>
</logger>

<!-- 输出了 Connection 的 open、close 等信息 -->
<logger name="jdbc.connection" level="OFF" additivity="false">
    <appender-ref ref="CONSOLE_LOG"/>
</logger>

<!-- 除了 ResultSet 之外的所有 JDBC 调用信息, 篇幅较长 -->
<logger name="jdbc.audit" level="OFF" additivity="false">
    <appender-ref ref="CONSOLE_LOG"/>
</logger>

```

依赖: slf4j-api, slf4j-log4j12

```

<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.14.1</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.14.1</version>
</dependency>

```

jul-to-slf4j

Introduction

jul-to-slf4j is a bridge that redirects logging messages from Java Util Logging (JUL), also known as `java.util.logging`, to SLF4J. JUL is the default logging framework included in the Java Development Kit (JDK).

Dependency

```

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>

```

```
<version>x.x.x</version> <!-- Replace with the version you need -->
</dependency>
```

jcl-over-slf4j

jcl-over-slf4j is a bridge that redirects logging messages from [Jakarta Commons Logging](#) (JCL) to [SLF4J](#). JCL is a popular logging facade used in many libraries and applications.

Dependency

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>x.x.x</version> <!-- Replace with the version you need -->
</dependency>
```

Microservice

Dubbo RPC

CORE

Apache Dubbo is a high-performance, Java-based open-source RPC (Remote Procedure Call) framework.

It's often used for service registration and discovery.

Dubbo and Feign

Dubbo

Protocol:

Dubbo is a [high-performance RPC](#) (Remote Procedure Call) [framework](#) that [uses binary protocols](#) for communication, making it more efficient than [text-based protocols like HTTP](#).

Dubbo supports multiple protocols such as [Dubbo](#), [RMI](#), [Hessian](#), [HTTP](#), [WebService](#), [Thrift](#), and [Redis](#).

Despite this wide range of protocol support, the Dubbo official documentation [recommends using the Dubbo protocol](#).

Service Discovery:

Dubbo Integrates [with various registries](#) such as Zookeeper, Nacos, Etcd, and Consul.

By default, Dubbo uses [Zookeeper](#) as its service registry.

Load Balancing:

It provides [built-in load balancing strategies](#) such as random, round-robin, and least-active.

Fault Tolerance:

Provides comprehensive [built-in fault tolerance mechanisms](#) with advanced features like retries, failover, and automatic service downgrades.

Web Container

Dubbo [does not require a web container to function](#), as it is a general RPC (Remote Procedure Call) framework designed to be lightweight and flexible.

It operates independently of web containers like Tomcat or Jetty, and services can be exposed and consumed without the need for a traditional web server.

Optional:

A web container can be used if your application needs to serve HTTP-based Dubbo services or integrate with web applications.

Feign:

Protocol:

Feign is a [declarative HTTP client](#) for Java, which simplifies HTTP-based service calls. It primarily uses [HTTP/HTTPS](#) protocols for communication.

Service Discovery:

Feign can integrate with Spring Cloud's service discovery components, such as Eureka.

Load Balancing:

Feign relies on Ribbon or Spring Cloud LoadBalancer for load balancing.

Fault Tolerance:

Relies on integration with external libraries like Hystrix or Resilience4j to achieve fault tolerance, focusing more on simplicity and declarative configuration.

Dubbo Performance

Reduced Header Overhead:

By avoiding traditional HTTP headers, Dubbo reduces the size of each request and response, leading to lower bandwidth usage and faster transmission.

Efficient Serialization:

Dubbo supports various serialization methods (e.g., Hessian, Protobuf, JSON) that are more efficient than the default serialization mechanisms used in HTTP-based communication.

These serialization methods are optimized for performance and reduce the size of the data being transmitted.

Persistent Connections:

Dubbo can use persistent connections (e.g., TCP connections) which avoid the overhead of establishing new connections for each request.

This is similar to HTTP keep-alive but with even lower overhead due to the custom protocol.

Multiplexing:

Dubbo's protocol supports request multiplexing, allowing multiple requests and responses to be sent over the same connection simultaneously, improving throughput and reducing latency.

Core Components

Provider

Service implementation that registers itself with a service registry.

Consumer

Service client that discovers and invokes services from the registry.

Registry

Centralized repository for service metadata and endpoint addresses.

Protocol

Defines the communication rules between providers and consumers.

Cluster

Manages multiple service instances for load balancing and failover.

Monitor

Tracks and reports service metrics for performance monitoring and diagnostics.

Load Balancing in Dubbo

Dubbo provides several strategies for load balancing to distribute requests among different service instances. The available strategies include:

Round Robin:

Description: Distributes requests sequentially across all available service instances.

Use Case: Uniform distribution of load when all service instances are equally capable of handling requests.

Random Load Balancing:

Description: Randomly selects a service instance from the available list.

Use Case: Simple use cases where load distribution does not need to be based on the state of the instances.

Least Active:

Description: Selects the service instance with the fewest number of active requests.

Use Case: Helps in distributing load more evenly by favoring instances with less current load.

Consistent Hashing:

Description: Uses a consistent hashing algorithm to ensure that requests with the same parameters are consistently routed to the same service instance.

Use Case: Useful when session persistence or consistent routing is needed.

Session Sticky (Sticky Session):

Description: Routes requests from a specific client to the same service instance, if possible.

Use Case: Ideal for stateful applications where the same instance should handle a session's requests.

Workflow

Startup

When the Spring Boot application starts, it initializes the application context and scans for configuration classes and components.

Dubbo Configuration Class

Spring Boot loads the Dubbo configuration with `DubboAutoConfiguration`, `DubboConfigConfiguration`, `DubboServiceAnnotationAutoConfiguration`, `DubboReferenceAnnotationAutoConfiguration`.

Auto-Configuration:

`DubboAutoConfiguration`

This class is part of the `dubbo-spring-boot-autoconfigure` module and is responsible for setting up Dubbo in the Spring Boot application. It includes:

`@Configuration`: Marks the class as a configuration class.

`@EnableConfigurationProperties`: Enables Dubbo-specific properties.

`@ConditionalOnClass`: Ensures that Dubbo is on the classpath.

Configuration Properties

`DubboConfigConfiguration`

Handles the creation and configuration of `Dubbo configuration beans` like `ApplicationConfig`, `RegistryConfig`, `ProtocolConfig`, etc.

Annotation Processing

`DubboServiceAnnotationAutoConfiguration`

Processes the `@DubboService` annotations to expose Dubbo services.

`@DubboService`: This annotation is used on service implementation classes to expose them as Dubbo services.

Service Exposure

`ServiceBean`: A Spring bean for a Dubbo service. It handles the lifecycle of the service.

`ServiceBean.onApplicationEvent()`: This method is called when the Spring context is refreshed, triggering the export of the Dubbo service.

Reference Injection

`DubboReferenceAnnotationAutoConfiguration`

Processes the `@Reference` annotations to inject Dubbo service references.

`@DubboReference`: This annotation is used on fields or setter methods to inject a Dubbo service reference.

`ReferenceBean`

A Spring bean for a Dubbo reference. It handles the lifecycle of the reference.

`ReferenceBean.afterPropertiesSet()`

This method is called to initialize the reference after the properties are set, creating the proxy for the remote service.

Configuration

pom.xml

```
<dependency>
<groupId>org.apache.dubbo</groupId>
```

```

<artifactId>dubbo-spring-boot-starter</artifactId>
<version>3.0.0</version>
</dependency>

application.yml
spring:
  application:
    name: dubbo-spring-boot-demo

dubbo:
  application:
    name: dubbo-spring-boot-demo
  registry:
    address: zookeeper://localhost:2181      # Address of the service registry (e.g., ZooKeeper).
  protocol:
    name: dubbo
    port: 20880
  scan:
    base-packages: com.example.dubbo      # Packages to scan for Dubbo services.

```

Define Dubbo Services

Define your Dubbo service interfaces and implementations.

Service Interface

```

package com.example.dubbo;

public interface HelloService {
    String sayHello(String name);
}

```

Service Implementation

Ensure your service **provider** is registered with Dubbo. This is done by annotating the implementation class with **@DubboService**, which registers the service in the service registry.

```

package com.example.dubbo;
import org.apache.dubbo.config.annotation.DubboService;

@DubboService
public class HelloServiceImpl implements HelloService {

    @Override
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}

```

To **consume** Dubbo services, you need to create a client application and use the **@DubboReference** annotation.

After your Spring Boot application starts. It will register itself with the registry and expose the defined services.

```

package com.example.dubbo;

import org.apache.dubbo.config.annotation.DubboReference;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @DubboReference
    private HelloService helloService;

    @GetMapping("/hello")
    public String hello(@RequestParam String name) {
        return helloService.sayHello(name);
    }
}

```

```
}
```

Feign RPC

A declarative web service client that [simplifies HTTP API calls](#). It integrates with [Ribbon](#) for client-side load balancing and with [Hystrix](#) for fault tolerance.

Core

Configuration

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Enable Feign Clients

Next, you need to enable Feign clients in your Spring Boot application by using the `@EnableFeignClients` annotation in your main application class:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableFeignClients
public class FeignExampleApplication {
    public static void main(String[] args) {
        SpringApplication.run(FeignExampleApplication.class, args);
    }
}
```

Create a Feign Client Interface

Define a Feign client interface with the RESTful web service endpoints you want to call. Use the `@FeignClient` annotation to specify the name of the service and the base URL:

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@FeignClient(name = "example-service", url = "http://example.com/api")
public interface ExampleServiceClient {

    @GetMapping("/items/{id}")
    Item getItemById(@PathVariable("id") Long id);

    @GetMapping("/items")
    List<Item> getAllItems();
}
```

Use the Feign Client

You can now inject and use the Feign client in your Spring components (e.g., services or controllers):

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ItemService {
```

```

@.Autowired
private ExampleServiceClient exampleServiceClient;

public Item getItemById(Long id) {
    return exampleServiceClient.getItemById(id);
}

public List<Item> getAllItems() {
    return exampleServiceClient.getAllItems();
}
}

```

Configuration (Optional)

You can customize the Feign client configuration by creating a configuration class:

```

import feign.Logger;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FeignConfig {

    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}

```

And then use the configuration in your Feign client:

```

@FeignClient(name = "example-service", url = "http://example.com/api", configuration = FeignConfig.class)
public interface ExampleServiceClient {
    // ...
}

```

Example Controller

Here is how you can use the Feign client in a controller to expose endpoints in your own application:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ItemController {

    @Autowired
    private ItemService itemService;

    @GetMapping("/items/{id}")
    public Item getItemById(@PathVariable("id") Long id) {
        return itemService.getItemById(id);
    }

    @GetMapping("/items")
    public List<Item> getAllItems() {
        return itemService.getAllItems();
    }
}

```

[feign-core]

RequestInterceptor

```
package feign;
public interface RequestInterceptor
    org.springframework.http.client.ClientHttpRequestInterceptor
```

Zero or more RequestInterceptors may be configured for purposes such as adding headers to all requests.

No guarantees are given with regards to the order that interceptors are applied.

Once interceptors are applied, Target.apply(RequestTemplate) is called to create the immutable http request sent via Client.execute(Request, Request.Options).

```
void apply(RequestTemplate var1)
```

Called for every request. Add data using methods on the supplied RequestTemplate.

RequestTemplate

```
package feign;
public final class RequestTemplate implements Serializable
    public RequestTemplate header(String name, String... values)
```

封装了执行请求需要的相关信息，比如请求方式、路径等。

在模板内添加一个请求头

ReflectiveFeign

```
package feign;
public class ReflectiveFeign extends Feign
    private static class BuildEncodedTemplateFromArgs extends ReflectiveFeign.BuildTemplateByResolvingArgs
        public RequestTemplate create(Object[] argv)
            该方法会根据方法元数据创建 RequestTemplate (在创建模板时，会
            直接调用 BuildEncodedTemplateFromArgs 的 resolve 方法)
        protected RequestTemplate resolve(Object[] argv, RequestTemplate mutable, Map<String, Object> variables)
            resolve
            中会使用编码器，将实体类转为请求体，最后调用 RequestTemplate 的解析方法处理参数
```

最后，

这种请求方式的参数就被转化为了 byte 数组封装在 RequestTemplate 中了

```
private static class BuildTemplateByResolvingArgs implements Factory
    private RequestTemplate addQueryMapQueryParameters(Map<String, Object> queryMap, RequestTemplate mutable)
        循环键值对，并将其转化为 QueryTemplate，可以看到直接使用值的 ToString 方法进行值的获取
```

最后，

值不为 null 的键值对，会被添加到 QueryTemplate 中，在执行请求时会将这些查询参数解析出来，这些参数会拼接在 URL 后面。

template

UriTemplate

```
package feign.template;
public class UriTemplate extends Template
```

QueryTemplate

```
package feign.template;
public final class QueryTemplate
```

HeaderTemplate

```
package feign.template;
```

```
public final class HeaderTemplate extends Template
```

BodyTemplate

```
package feign.template;  
public final class BodyTemplate extends Template
```

auth

BasicAuthRequestInterceptor

```
package feign.auth;  
public class BasicAuthRequestInterceptor implements RequestInterceptor
```

往 RequestTemplate 添加名为 Authorization 的 header

Hystrix

Core

Hystrix is an open-source [latency and fault tolerance library](#) developed by Netflix.

It's designed to isolate access points to remote systems, services, and third-party libraries, stopping cascading failures and enabling resilience in complex distributed systems.

By using Hystrix, developers can manage failures gracefully and ensure that their applications remain responsive even when external systems fail.

WorkFlow

Spring Boot Initialization

When a Spring Boot application starts, it performs classpath scanning and loads configuration properties.

The `@SpringBootApplication` annotation triggers [component scanning](#) and [auto-configuration](#).

If you've annotated your application with `@EnableHystrix`, Spring Boot will [load the Hystrix configuration with HystrixAutoConfiguration class](#).

The `HystrixCommandAspect` aspect that intercepts methods annotated with `@HystrixCommand` and manages Hystrix command execution.

When the annotated method is called, Hystrix [creates a command instance using HystrixCommand](#).

```
@Aspect  
public class HystrixCommandAspect {  
    @Around("@annotation(com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand)")  
    public Object methodsAnnotatedWithHystrixCommand(ProceedingJoinPoint joinPoint) throws Throwable {  
        // logic to execute Hystrix command  
    }  
}
```

Command Execution

When the annotated method is called, Hystrix [creates a command instance using HystrixCommand](#).

Metrics

Hystrix [collects metrics and exposes them](#) through various dashboards for monitoring the status of circuits, thread pools, and command executions.

Actuator

Add Actuator Dependency

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

Enable Hystrix Metrics Endpoint

Actuator exposes several useful endpoints. If Hystrix is on your classpath, Actuator will expose a [/actuator/hystrix.stream](#) endpoint automatically.

Recommendations for 2000 QPS System

`circuitBreaker.requestVolumeThreshold = 200`

Given 2000 QPS, setting this to 200 means the circuit breaker will evaluate the error rate after every 100 milliseconds, providing a good balance between sensitivity and stability.

`circuitBreaker.errorThresholdPercentage = 50`

Setting this to 50 means the circuit breaker will trip if 50% of the requests fail, allowing it to react to significant issues while avoiding false positives.

`circuitBreaker.sleepWindowInMilliseconds = 10000 (10 seconds)`

This gives the system sufficient time to recover before testing again, preventing frequent state transitions and potential instability.

`execution.isolation.thread.timeoutInMilliseconds = 2000 (2 seconds)`

Ensures that long-running requests do not hold up threads, which is critical in high-QPS systems.

`metrics.rollingPercentile.windowInMilliseconds = 60000 (1 minute)`

This helps smooth out short-term fluctuations and provides a more stable view of system performance.

`execution.isolation.strategy = THREAD`

For high QPS systems, thread isolation can be more suitable to handle concurrency effectively. However, this depends on your system's architecture and resource constraints.

Configuration

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

application.yml

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 3000
```

Beans

Configure a `RestTemplate` bean if you are using it to call external services:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Enable Hystrix

Enable Hystrix in your Spring Boot application by adding the `@EnableHystrix` annotation to your main application class:

```
import org.springframework.boot.SpringApplication;
```

```

import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootApplication
@EnableHystrix
public class HystrixApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixApplication.class, args);
    }
}

```

Define a Service with Hystrix Command

Create a service class and use the `@HystrixCommand` annotation to specify the fallback method to be called in case of failure:

```

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @HystrixCommand(fallbackMethod = "fallbackMethod")
    public String callExternalService() {
        return restTemplate.getForObject("http://external-service/api", String.class);
    }

    public String fallbackMethod() {
        return "Fallback response";
    }
}

```

Monitoring and Dashboard (Optional)

Hystrix also provides a dashboard to monitor metrics. To enable the dashboard, add the following dependencies:

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>

```

Enable the dashboard in your main application class:

You can then access the dashboard at <http://localhost:8080/hystrix>.

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

@SpringBootApplication
@EnableHystrixDashboard
public class HystrixApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixApplication.class, args);
    }
}

```

[hystrix-core]

HystrixCircuitBreaker

```

package com.netflix.hystrix;
public interface HystrixCircuitBreaker

```

```

boolean allowRequest();
    The allowRequest() method determines if a request should be allowed to proceed or be short-circuited.
    If the circuit is forced open, no requests are allowed.
    If the circuit is forced closed, all requests are allowed.
    If the circuit is open, the allowSingleTest() method is called to potentially allow a single test request.
    If the circuit is closed, all requests are allowed.

boolean isOpen();

void markSuccess();
    The markSuccess() method is called when a request succeeds. If the circuit is open, it resets the circuit to closed and
    resets the metrics.

```

[hystrix-javanica]

HystrixCommand

```

package com.netflix.hystrix.contrib.javanica.annotation;
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface HystrixCommand

```

By default, when you annotate a method with @HystrixCommand, it executes synchronously. The method call will block until the result is available or an error occurs.

To execute the method asynchronously, you can configure the @HystrixCommand to return a Future, Observable, or Single.

In these cases, the method call will return immediately with a non-blocking result, and you can handle the response asynchronously.

```

String threadPoolKey() default "";
    Specifies the thread pool to use for the command.
HystrixProperty[] threadPoolProperties() default {};
    Customizes thread pool properties.

```

```

String groupKey() default "";
    A key to group related commands.
String commandKey() default "";
    A unique identifier for the command.
String fallbackMethod() default "";
    Specifies the name of the fallback method.
HystrixProperty[] commandProperties() default {};
    Customizes command properties like execution timeout, circuit breaker settings, etc.

```

```

Class<? extends Throwable>[] ignoreExceptions() default {};
    Specifies exceptions that should not trigger the fallback method.
ObservableExecutionMode observableExecutionMode() default ObservableExecutionMode.EAGER;

```

```

HystrixException[] raiseHystrixExceptions() default {};
String defaultFallback() default "";
Fallback Method
Provides a graceful degradation path when the primary service call fails.

```

If a call [fails](#), [times out](#), or [the circuit breaker is open](#), a fallback method is executed.

The return value of the fallback method [will replace the return value](#) of the original method.

In this example, if the callExternalService method fails, the fallbackMethod will be executed.

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @HystrixCommand(fallbackMethod = "fallbackMethod")
    public String callExternalService() {
        return restTemplate.getForObject("http://external-service/api", String.class);
    }

    public String fallbackMethod() {
        return "Fallback response";
    }
}
```

Circuit Breaker and Timeout

Prevent the system from repeatedly calling failed services or services that take too long to respond.

When [failures reach a certain threshold](#), the [circuit breaker trips](#) and [subsequent calls fail immediately without making an actual call to the service](#).

By defining [timeouts](#) and [managing retries](#), Hystrix ensures that slow services do not degrade the performance of the entire system.

This example sets a timeout of 2 seconds for the method execution and configures the circuit breaker with specific properties.

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @HystrixCommand(fallbackMethod = "fallbackMethod", commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "2000"),
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "5000"),
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "50")
    })
    public String callExternalService() {
        return restTemplate.getForObject("http://external-service/api", String.class);
    }

    public String fallbackMethod() {
```

```

        return "Fallback response";
    }
}

```

Bulkhead Pattern

Isolates failures in different parts of a system to prevent cascading failures.

By using separate thread pools or semaphore isolation for different services or components, it limits the impact of failure to a specific component.

```

import org.springframework.stereotype.Service;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;

@Service
public class BulkheadService {

    @HystrixCommand(fallbackMethod = "fallbackMethod", threadPoolProperties = {
        @HystrixProperty(name = "coreSize", value = "10"),
        @HystrixProperty(name = "maxQueueSize", value = "10")
    })
    public String performOperation() throws InterruptedException {
        // Simulate a service call
        Thread.sleep(1000);
        return "Service response";
    }

    public String fallbackMethod() {
        return "Fallback due to bulkhead isolation";
    }
}

```

Request Cache

This example enables request caching for the callExternalService method.

```

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @HystrixCommand(fallbackMethod = "fallbackMethod", commandProperties = {
        @HystrixProperty(name = "requestCache.enabled", value = "true")
    })
    public String callExternalService(String id) {
        return restTemplate.getForObject("http://external-service/api/" + id, String.class);
    }

    public String fallbackMethod(String id) {
        return "Fallback response for id: " + id;
    }
}

```

Request Collapsing

Reduces the number of requests sent to a backend service by batching multiple requests together.

Hystrix can batch requests together and send them **as a single request**, thus optimizing resource utilization.

This example uses request collapsing to batch multiple requests into a single request.

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCollapser;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.concurrent.Future;

@Service
public class MyService {

    @HystrixCollapser(batchMethod = "getMultiple", scope = com.netflix.hystrix.HystrixCollapser.Scope.GLOBAL)
    public Future<String> getSingle(String id) {
        // This method will be collapsed into a batch request
        return null;
    }

    @HystrixCommand
    public List<String> getMultiple(List<String> ids) {
        // This method executes the batch request
        return restTemplate.getForObject("http://external-service/api?ids=" + String.join(",", ids),
List.class);
    }
}
```

HystrixProperty

```
package com.netflix.hystrix.contrib.javanica.annotation;
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface HystrixProperty
```

The `@HystrixProperty` annotation is used to define individual properties. Multiple `@HystrixProperty` annotations can be combined within a single `@HystrixCommand` to set multiple properties.

```
String name();
String value();
```

Execution Properties

execution.isolation.strategy

Define how a Hystrix command is isolated from other commands to ensure that failures or delays in one command do not affect the overall system.

either THREAD or SEMAPHORE (The default value is THREAD).

- Thread Isolation

Each Hystrix command runs in its own thread from a thread pool.

This means that each command is isolated from others, and failures or long-running operations in one thread **do not affect other threads**.

This isolation strategy is suitable for scenarios where the command's execution is time-consuming or involves blocking operations.

This isolates the command from the rest of the application. If the command times out or the thread pool is full,

Hystrix can immediately return a fallback.

- **Semaphore Isolation**

Commands executed with the semaphore isolation strategy run on the thread that invoked the command.

Each HystrixCommand can have its own semaphore isolation settings, including its own number of permits.

The semaphore has a fixed number of permits, and each request to the Hystrix command will attempt to acquire a permit.

If all permits are used, subsequent commands will be rejected until a permit becomes available.

This strategy is suitable for scenarios where commands are light-weight and you want to limit the number of concurrent commands without spawning new threads.

execution.timeout.enabled

Whether timeouts are enabled (the default value is true).

execution.isolation.thread.timeoutInMilliseconds

Timeout value for command execution. (The default value is 1000 milliseconds).

execution.isolation.thread.interruptOnTimeout

Whether to interrupt execution on timeout (the default value is true).

execution.isolation.thread.interruptOnCancel

Whether to interrupt execution on cancel (the default value is false).

execution.isolation.semaphore.maxConcurrentRequests

Maximum concurrent requests when using semaphore isolation.

Thread Pool Properties

coreSize

Core number of threads in the thread pool.

Hystrix uses a thread pool to manage these threads. Each command execution gets a thread from this pool.

The thread pool in Hystrix is specifically used in the THREAD isolation strategy, not the semaphore isolation strategy.

maximumSize

Maximum number of threads in the thread pool.

maxQueueSize

Maximum queue size for the thread pool.

queueSizeRejectionThreshold

Queue size at which requests are rejected.

keepAliveTimeMinutes

Time for keeping idle threads alive.

metrics.rollingStats.timeInMilliseconds

Duration of statistical rolling window for thread pool metrics.

metrics.rollingStats.numBuckets

Number of buckets in the statistical rolling window for thread pool metrics.

Circuit Breaker Properties

circuitBreaker.enabled

Whether the circuit breaker is enabled.

circuitBreaker.requestVolumeThreshold

This property defines the minimum number of requests that must be made within a rolling window before the circuit breaker will consider tripping based on the error rate.

It helps prevent the circuit breaker from opening due to a small number of requests, which could lead to unnecessary openings based on insufficient data.

circuitBreaker.sleepWindowInMilliseconds

This property specifies how long the circuit breaker **stays open** after it has been tripped. During this period, **all requests will be rejected**.

States: A circuit breaker has three states:

Closed:

Requests are allowed to pass through. If the failure rate **exceeds a threshold**, the circuit breaker transitions to the open state.

Open:

Requests are **blocked**, and the system waits for a predefined period (the `sleepWindowInMilliseconds`) before transitioning to the half-open state.

After a predefined period (the sleep window), the circuit breaker transitions to the **half-open** state.

Half-Open:

The circuit breaker allows a single test request to pass through to the service.

Success:

If the request is successful, the circuit breaker **transitions back to the Closed state**.

Failure:

If the request fails, the circuit breaker **transitions back to the Open state**.

circuitBreaker.errorThresholdPercentage

This property determines the **percentage of requests that must fail** (i.e., return an error) for the circuit breaker to trip.

The failure percentage is calculated only if the number of requests meets or exceeds the `circuitBreaker.requestVolumeThreshold`.

circuitBreaker.forceOpen

Forces the circuit breaker **open**, preventing requests.

circuitBreaker.forceClosed

Forces the circuit breaker closed, allowing requests.

Request Context Properties

requestCache.enabled

Whether request caching is enabled.

The cache is stored in memory and is scoped to the duration of a single request.

Avoid Redundant Execution:

When **multiple commands with the same parameters are invoked** within the same request (such as a single HTTP request),

the `requestCache` allows Hystrix to return the cached result of the first command execution instead of executing the command again.

Cache Results: Hystrix

can cache the results of a command execution within the same request.

If the same command is invoked multiple times within the same request (e.g., from the same HTTP request or thread context),

the results will be fetched from the cache rather than re-executed.

Cache Key:

The cache is keyed by the command's input parameters.

If a command is invoked with the same parameters, Hystrix will use the cached result for that command.

Clearing the Cache:

The request cache is cleared **at the end of each request**. This means that the cache is **not persistent across different requests**.

Custom Cache Key:

For more complex scenarios, you can implement custom caching logic using `HystrixRequestContext` to manage how and when cache entries are cleared.

requestLog.enabled

Whether request logging is enabled.

Collapse Properties

`collapser.maxRequestsInBatch`

Maximum number of requests in a batch.

`collapser.timerDelayInMilliseconds`

Time window for batching requests.

`collapser.requestCache.enabled`

Whether request caching is enabled for collapser.

Metrics Properties

`metrics.rollingStats.timeInMilliseconds`

Duration of statistical rolling window.

`metrics.rollingStats.numBuckets`

Number of buckets in the statistical rolling window.

`metrics.rollingPercentile.enabled`

Whether rolling percentiles are enabled.

`metrics.rollingPercentile.timeInMilliseconds`

Duration of percentile rolling window.

`metrics.rollingPercentile.numBuckets`

Number of buckets in the percentile rolling window.

`metrics.rollingPercentile.bucketSize`

Size of each bucket in the percentile rolling window.

Other

熔断器原理

滑动窗口设计目的：上面提到的断路器需要的时间窗口**请求量**和**错误率**这两个统计数据，都是指**固定时间长度内的统计数据**，断路器的目标，就是根据这些统计数据来预判并决定系统下一步的行为

执行过程：Hystrix 通过**滑动窗口**来对数据进行“平滑”统计，默认情况下，一个滑动窗口包含 **10 个桶(Bucket)**，每个桶时间宽度是 1 秒，负责 1 秒的数据统计。

滑动窗口**包含的总时间**以及其中的**桶数量**都是可以配置的。

上图的每个小矩形代表一个桶，可以看到，每个桶都记录着 **1 秒内**的四个指标数据：**成功量、失败量、超时量和拒绝量**

这里的拒绝量指的就是上面流程图中【信号量/线程池资源检查】中被拒绝的流量。10 个桶合起来是一个完整的滑动窗口，所以计算**一个滑动窗口的总数据需要将 10 个桶的数据加起来**。

OKHttp

The OkHttp library is a popular HTTP client used for making HTTP requests in Java applications, including Spring Boot applications. It is known for its efficiency, ease of use, and powerful features.

It can be used as the underlying HTTP client in Feign for making API calls.

Configuration

pom.xml

First, add the OkHttp dependency to your pom.xml if you are using Maven:

```
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>okhttp</artifactId>
    <version>4.9.2</version>
</dependency>
```

Create an OkHttpClient Bean

Next, create an OkHttpClient bean in your Spring Boot application. This bean can be used throughout your application to make HTTP requests.

```
import okhttp3.OkHttpClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OkHttpConfig {

    @Bean
    public OkHttpClient okHttpClient() {
        return new OkHttpClient.Builder()
            .build();
    }
}

@Configuration
@ConditionalOnClass(Feign.class)
@AutoConfigureBefore(FeignAutoConfiguration.class)
public class OkHttpConfig {
    @Bean
    public okhttp3.OkHttpClient client(OkHttpClientFactory httpClientFactory,
                                         ConnectionPool connectionPool,
                                         FeignHttpClientProperties httpClientProperties) {
        Boolean followRedirects = httpClientProperties.isFollowRedirects();
        Integer connectTimeout = httpClientProperties.getConnectionTimeout();
        Boolean disableSslValidation = httpClientProperties.isDisableSslValidation();
        this.okHttpClient = httpClientFactory.createBuilder(disableSslValidation)
            .connectTimeout(connectTimeout, TimeUnit.MILLISECONDS)
            .followRedirects(followRedirects).connectionPool(connectionPool)
            .addInterceptor(new MyOkhttpInterceptor())
            .build();
        return this.okHttpClient;
    }
}
```

Use OkHttpClient to Make HTTP Requests

You can now inject the OkHttpClient bean into your service classes and use it to make HTTP requests.

```
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.io.IOException;

@Service
public class HttpService {

    private final OkHttpClient okHttpClient;

    @Autowired
    public HttpService(OkHttpClient okHttpClient) {
        this.okHttpClient = okHttpClient;
    }

    public String makeGetRequest(String url) throws IOException {
        Request request = new Request.Builder()
            .url(url)
            .build();
```

```

        try (Response response = okHttpClient.newCall(request).execute()) {
            if (!response.isSuccessful()) {
                throw new IOException("Unexpected code " + response);
            }
            return response.body().string();
        }
    }
}

```

Example Controller Class

You can create a controller class to handle HTTP requests and use the `HttpService` to make external HTTP requests.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ApiController {

    private final HttpService httpService;

    @Autowired
    public ApiController(HttpService httpService) {
        this.httpService = httpService;
    }

    @GetMapping("/fetch")
    public String fetch(@RequestParam String url) {
        try {
            return httpService.makeGetRequest(url);
        } catch (IOException e) {
            e.printStackTrace();
            return "Error: " + e.getMessage();
        }
    }
}

```

Ribbon load balancer

A client-side load balancer that works with Feign [to distribute load](#) across multiple instances of a microservice.

It is often used in conjunction with [Spring Cloud](#) to provide load balancing for microservices.

However, as of Spring Cloud 2020.0.0 (codename: Ilford), Ribbon is in maintenance mode and the Spring team recommends using [Spring Cloud LoadBalancer](#) as an alternative.

Core

Workflow

Spring Boot Initialization

When a Spring Boot application starts, it performs classpath scanning and loads configuration properties.

The `@SpringBootApplication` annotation triggers [component scanning](#) and [auto-configuration](#).

If you've annotated your application with `@RibbonClient`, Spring Boot will load the Ribbon configuration with `RibbonClientConfiguration`, `RibbonAutoConfiguration`.

The `RibbonConfiguration` class defines the Ribbon load balancer configuration.

```

@Configuration
public class RibbonConfiguration {

    @Bean
    public IRule ribbonRule() {
        return new AvailabilityFilteringRule(); // or any other rule implementation
    }
}

```

Creating the Load Balancer

Ribbon creates a `ILoadBalancer` instance, typically `ZoneAwareLoadBalancer`, which manages the list of servers and applies the load balancing strategy.

```
ILoadBalancer loadBalancer = new ZoneAwareLoadBalancer<>();
```

Setting Up RestTemplate with Ribbon

If you have defined a `RestTemplate` bean with the `@LoadBalanced` annotation, Spring Boot configures the `RestTemplate` to use Ribbon for client-side load balancing.

```
@Configuration
public class RestTemplateConfig {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

The `@LoadBalanced` annotation ensures that the `RestTemplate` is intercepted and configured to use Ribbon's `LoadBalancerInterceptor`.

```
@Bean
public LoadBalancerInterceptor ribbonInterceptor(ILoadBalancer loadBalancer, ServerIntrospector
serverIntrospector, LoadBalancerClientFactory loadBalancerClientFactory) {
    return new LoadBalancerInterceptor(loadBalancer, serverIntrospector, loadBalancerClientFactory);
}
```

RestTemplate Request Execution

When the `RestTemplate` makes a request to a service, Ribbon intercepts the request and determines which server to use based on the load balancing strategy.

```
@Service
public class RibbonService {

    private final RestTemplate restTemplate;

    public RibbonService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public String getDataFromService() {
        return restTemplate.getForObject("http://ribbon-client/service-endpoint", String.class);
    }
}
```

The `LoadBalancerInterceptor` class intercepts the request and uses the `ILoadBalancer` to choose a server.

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {

    private final ILoadBalancer loadBalancer;

    public LoadBalancerInterceptor(ILoadBalancer loadBalancer) {
        this.loadBalancer = loadBalancer;
    }

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body, ClientHttpRequestExecution
execution) throws IOException {
        Server server = loadBalancer.chooseServer();
        URI originalUri = request.getURI();
        String serviceUri = String.format("%s://%s:%s%s", originalUri.getScheme(), server.getHost(),
server.getPort(), originalUri.getPath());
        HttpRequest newRequest = new HttpRequestWrapper(request) {
            @Override
            public URI getURI() {
                return URI.create(serviceUri);
            }
        };
        return execution.execute(newRequest, body);
    }
}
```

```
}
```

Load Balancer Chooses a Server

The `ILoadBalancer` uses the configured rule (e.g., `RoundRobinRule`, `AvailabilityFilteringRule`) to choose a server from the list of available servers.

```
public class RoundRobinRule extends AbstractLoadBalancerRule {  
  
    private AtomicInteger nextServerCyclicCounter;  
  
    public RoundRobinRule() {  
        nextServerCyclicCounter = new AtomicInteger(0);  
    }  
  
    @Override  
    public Server choose(ILoadBalancer lb, Object key) {  
        if (lb == null) {  
            return null;  
        }  
        List<Server> allServers = lb.getAllServers();  
        int serverCount = allServers.size();  
        int nextServerIndex = incrementAndGetModulo(serverCount);  
        return allServers.get(nextServerIndex);  
    }  
  
    private int incrementAndGetModulo(int modulo) {  
        for (;;) {  
            int current = nextServerCyclicCounter.get();  
            int next = (current + 1) % modulo;  
            if (nextServerCyclicCounter.compareAndSet(current, next)) {  
                return next;  
            }  
        }  
    }  
}
```

Sending the Request:

The `RestTemplate` sends the request to the chosen server, and the response is returned to the calling service.

Configuration

pom.xml

```
<dependencyManagement>  
    <dependencies>  
        <dependency>  
            <groupId>org.springframework.cloud</groupId>  
            <artifactId>spring-cloud-dependencies</artifactId>  
            <version>Greenwich.SR6</version> <!-- Or the appropriate version for your project -->  
            <type>pom</type>  
            <scope>import</scope>  
        </dependency>  
    </dependencies>  
</dependencyManagement>
```

Enable Ribbon

Annotate your Spring Boot application or configuration class with `@RibbonClient`.

```
@SpringBootApplication  
{@RibbonClient(name = "ribbon-client", configuration = RibbonConfiguration.class)  
public class RibbonApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(RibbonApplication.class, args);  
    }  
}
```

Beans

```

@Configuration
public class RibbonConfiguration {

    @Bean
    public IRule ribbonRule() {
        return new AvailabilityFilteringRule(); // or any other rule implementation
    }

    // Define a RestTemplate bean and use it in your services.
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

}

```

application.yml

```

ribbon-client:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8081,localhost:8082,localhost:8083
    NLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
    ConnectTimeout: 3000
    ReadTimeout: 3000

```

Using Ribbon with RestTemplate

Use the RestTemplate to make requests.

```

@Service
public class RibbonService {

    private final RestTemplate restTemplate;

    public RibbonService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public String getDataFromService() {
        return restTemplate.getForObject("http://ribbon-client/service-endpoint", String.class);
    }
}

```

Load Balancing Strategies

RoundRobinRule

Distributes requests evenly across all available servers **in a circular order**.

Use Case

Suitable for scenarios where an even distribution of requests is desired.

Example

If you have three servers, the first request goes to server A, the second to server B, the third to server C, the fourth back to server A, and so on.

RandomRule

Randomly selects a server from the list of available servers.

Use Case

Useful for scenarios where you want a random distribution of requests to avoid any potential bias.

Example

Each request is sent to a randomly chosen server.

RetryRule

Combines a specified rule (e.g., RoundRobinRule) with retry logic. If a request fails, it retries on other servers.

Use Case

Helpful when you want to ensure a higher success rate by retrying failed requests on different servers.

Example

Uses RoundRobinRule, but if a request to server A fails, it retries on server B.

WeightedResponseTimeRule

Assigns a weight to each server **based on its response time**. Servers with faster response times get more requests.

Use Case

Suitable for scenarios where response time is a critical factor.

Example

If server A has a response time of 50ms, server B has 100ms, and server C has 200ms, server A will receive more requests than B and C.

BestAvailableRule

Chooses the server **with the least concurrent requests**.

Use Case

Ideal for minimizing the load on heavily used servers and reducing response time.

Example

If server A has 2 concurrent requests, server B has 1, and server C has 3, the next request goes to server B.

AvailabilityFilteringRule

Filters out servers that are deemed "**circuit tripped**" or **have high concurrent connections**, and then uses a specified rule (e.g., RoundRobinRule) on the remaining servers.

Use Case

Useful for improving availability by avoiding problematic servers.

Example

Excludes servers that have failed health checks or have high load, and balances the load among the remaining servers.

ZoneAvoidanceRule

Combines Zone-aware load balancing with AvailabilityFilteringRule. It **avoids zones with poor performance** and selects the best server within the preferred zone.

Use Case

Useful for multi-zone deployments where zone performance varies.

Example

Prefers servers in the same zone with better performance and fewer failures.

Spring Cloud LoadBalancer

Configuration

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Configure Load Balancer

Spring Cloud LoadBalancer can be used with **RestTemplate** or **WebClient**. Here's how to configure each:

Using RestTemplate

Define a RestTemplate Bean with Load Balancer

Use the `@LoadBalanced` annotation to make `RestTemplate` aware of the load balancer.

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Use RestTemplate in Your Service

Inject and use `RestTemplate` to make load-balanced requests.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    @Autowired
    private RestTemplate restTemplate;

    public String getData() {
        // Make a request to the load-balanced service
        return restTemplate.getForObject("http://my-service/data", String.class);
    }
}
```

Using WebClient

Define a WebClient Bean with Load Balancer

Configure `WebClient` with `LoadBalancerExchangeFilterFunction`.

```
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.cloud.client.loadbalancer.LoadBalancerExchangeFilterFunction;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class AppConfig {

    @Bean
    public WebClient.Builder webClientBuilder(LoadBalancerClient loadBalancerClient) {
        return WebClient.builder()
            .filter(new LoadBalancerExchangeFilterFunction(loadBalancerClient));
    }
}
```

Use WebClient in Your Service

Inject and use `WebClient` to make load-balanced requests.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class MyService {

    @Autowired
    private WebClient.Builder webClientBuilder;
```

```

        public Mono<String> getData() {
            WebClient webClient = webClientBuilder.build();
            return webClient.get()
                .uri("http://my-service/data")
                .retrieve()
                .bodyToMono(String.class);
        }
    }
}

```

Define Application Properties

Make sure you have the service discovery properties configured in your `application.properties` or `application.yml`.

```

my-service:
  ribbon:
    listOfServers: http://localhost:8081,http://localhost:8082

```

Configure Service Registration (Optional)

If you're using a service registry like Eureka, you might need to configure your services to register and discover instances.

```

# Eureka client configuration
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

```

Monitor

Spring Cloud Sleuth

Core

Spring Cloud Sleuth is a [distributed tracing solution](#) that helps in monitoring and troubleshooting microservices-based applications.

It provides capabilities to trace requests across different services, enabling better visibility into the flow of requests and performance bottlenecks.

Key Components

Trace ID

A unique identifier assigned to each request that allows tracking through various services.

All logs related to that request [can be linked using this ID](#).

Span ID

Represents a [single operation](#) within a trace.

Each operation has its own Span ID, which helps in detailing the duration and attributes of that operation.

Span

A Span is a logical unit of work [representing a single operation](#).

It contains information like start time, end time, and other metadata about the operation being executed.

Sampler

A component that determines [whether a request should be traced or not](#).

It can be configured to sample all requests or a specific percentage of them.

Brave

The underlying library used by Sleuth for tracing. It provides the tracing API and implementations for managing traces and spans.

Logging

Sleuth automatically enriches logs with trace and span IDs, making it easier to correlate logs with requests.

Integration with Tracing Systems

Sleuth can integrate with distributed tracing systems like Zipkin or OpenTelemetry for visualizing traces and managing collected data.

Workflow

Sleuth adds [trace and span IDs](#) to your logs to track the flow of requests.

Each request is assigned [a unique trace ID](#), and each segment of a request [is given a span ID](#).

These IDs are propagated across microservices, ensuring that all related log entries can be easily correlated.

Example log output:

```
[traceId, spanId] INFO --- c.e.DemoApplication: Started DemoApplication in 5.4 seconds
```

Usage Scenarios

Performance Monitoring

By tracking the time taken for each operation, developers can identify slow services and optimize them.

Error Diagnosis

Helps in pinpointing where failures occur in a request flow by providing a complete trace of how the request was handled

Distributed Context Propagation

Automatically propagates trace context across service boundaries, ensuring that trace information is consistent across services.

Version

Spring Cloud Sleuth was removed from the release train in [Spring Cloud 2022.0](#), and its functionality was integrated into [Micrometer Tracing](#).

Configuration

Add Dependencies

If you're using Spring Cloud, make sure to add the appropriate Spring Cloud BOM in your `pom.xml`

For Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

For Gradle

```
implementation 'org.springframework.cloud:spring-cloud-starter-sleuth'
implementation 'org.springframework.cloud:spring-cloud-starter-sleuth:3.1.11'
```

```
implementation 'io.micrometer:micrometer-tracing'
```

```
implementation 'io.micrometer:micrometer-tracing-brave' // If you want to continue using Brave
```

webflux:

```
implementation 'org.springframework.cloud:spring-cloud-sleuth-starter-reactor-netty:3.2.1'
```

Enable Sleuth

In your Application class, add the `@EnableSleuth` annotation to enable Sleuth tracing:

```
@SpringBootApplication
@EnableSleuth
public class MyApplication {

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

Configuration Properties (Optional)

You can customize Sleuth's behavior by setting properties in your application's configuration.

For example, to change the default tracer name:

```
spring:
  application:
    name: my-web-flux-app
  sleuth:
    sampler:
      probability: 1.0 # Set sampling rate to 100% for all traces
    tracer:
      name: my-custom-tracer
```

Use Sleuth in Your Application

Inject Span: In your controllers or services, you can inject the Span object to manually start, stop, or tag spans.

```
@RestController
public class MyController {

  @Autowired
  private SpanTracer spanTracer;

  @GetMapping("/hello")
  public String hello() {
    Span span = spanTracer.currentSpan();
    span.tag("http.method", "GET");
    span.tag("http.path", "/hello");
    return "Hello, World!";
  }
}
```

Run Your Application

Start your Spring Boot application. Sleuth will automatically add tracing information to your logs, such as trace IDs and span IDs.

Using Tracing in Code

Inject Span

You can inject `Tracer` or `Span` in your services to create custom spans if needed:

```
@RestController
public class MyController {

  @Autowired
  private SpanTracer spanTracer;

  @GetMapping("/hello")
  public String hello() {
    Span span = spanTracer.currentSpan();
    span.tag("http.method", "GET");
    span.tag("http.path", "/hello");
    return "Hello, World!";
  }
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import brave.Tracer;
import brave.Span;
```

```

@Service
public class MyService {
    @Autowired
    private Tracer tracer;

    public void myMethod() {
        Span newSpan = tracer.nextSpan().name("my-span").start();
        try (Tracer.SpanInScope ws = tracer.withSpanInScope(newSpan)) {
            // Your logic here
        } finally {
            newSpan.finish();
        }
    }
}

```

Use Annotations

Sleuth provides annotations like `@Span` and `@NewSpan` to automatically create and manage spans.

```

@RestController
public class MyController {

    @GetMapping("/hello")
    @Span(name = "helloEndpoint")
    public String hello() {
        return "Hello, World!";
    }
}

```

Log Spans

Use the `Span` object to log information about the current span.

```

@RestController
public class MyController {

    @Autowired
    private SpanTracer spanTracer;

    @GetMapping("/hello")
    public String hello() {
        Span span = spanTracer.currentSpan();
        span.logEvent("message", "Hello, World!");
        return "Hello, World!";
    }
}

```

View Traces

Once you've enabled Sleuth, you can view the generated traces in a distributed tracing system like Zipkin or Jaeger. You'll need to configure your application to send traces to your chosen system.

Log Trace Information

Ensure your logging framework is configured to include trace information.

If you're using Logback, you can configure your `'logback-spring.xml'` like this:

```

<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg %X{X-B3-Traceld} %X{X-B3-
SpanId}%n</pattern>

```

```
</encoder>
</appender>
<root level="INFO">
    <appender-ref ref="STDOUT" />
</root>
</configuration>
```

[spring-cloud-sleuth-core]

```
package org.springframework.cloud.sleuth.annotation;
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Target({ElementType.METHOD})
public @interface ContinueSpan
```

Mark a method or a class to continue the current tracing span automatically.

This is especially useful in scenarios where you want to propagate the tracing context without manually managing spans in your code.

Asynchronous Context

When using this annotation in an asynchronous context, the span will automatically propagate across threads.

```
String log() default "";
```

Usage

```
import org.springframework.cloud.sleuth.annotation.ContinueSpan;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    @ContinueSpan
    public void myMethod() {
        // This method continues the current span
        // Your business logic here
    }
}
```

micrometer

Add Dependencies

```
implementation 'io.projectreactor:reactor-core-micrometer'
```

Use the @SpanName Annotation

Apply the @SpanName annotation to your controller methods or service methods to provide a custom span name:

Plugins

maven-source-plugin

在指定的生命周期执行后生成源码 jar 包

classes		2019/12/10 11:37	文件夹	
maven-archiver		2019/12/10 11:37	文件夹	
maven-status		2019/12/10 11:37	文件夹	
test-classes		2019/12/10 11:37	文件夹	
project03-1.0-SNAPSHOT.jar		2019/12/10 11:37	WinRAR 压缩文件	3 KB
project03-1.0-SNAPSHOT-sources.jar		2019/12/10 11:37	WinRAR 压缩文件	1 KB
project03-1.0-SNAPSHOT-test-sources...		2019/12/10 11:37	WinRAR 压缩文件	1 KB

pow.xml >>

```

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <goals>
                <goal>jar</goal>
            </goals>
            <phase>generate-test-resources</phase>      运行到这个生命周期执行
        </execution>
    </executions>
</plugin>

```

maven-resource-plugin

负责将正式与测试用到的资源文件导出到指定位置。还负责用资源文件中的参数替换 pom 文件中对应的占位符

```

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>3.2.0</version>
    <configuration>
        <encoding>UTF-8</encoding>
        <useDefaultDelimiters>true</useDefaultDelimiters>      <!-- 当这里为 true, 那么 resource 文件夹下的配置文件, 比如
application.yml 这些文件里面的${}包起来的内容就可以被 pom 文件中 profiles 标签下的对应名称部分行替换了 -->
    </configuration>
    <executions>      <!-- 复制资源和 bin 文件夹 -->
        <execution>
            <id>copy-resources</id>
            <phase>package</phase>
            <goals>
                <goal>copy-resources</goal>
            </goals>
            <configuration>
                <resources>
                    <resource>
                        <directory>src/main/resources</directory>    <!-- 文件来源 -->
                    </resource>
                </resources>
                <outputDirectory>${project.build.directory}/pack/resources</outputDirectory>    <!-- 资源文件的输出目录 -->
            </configuration>
        </execution>
    </executions>
</plugin>

```

```

</execution>
<execution>
    <id>copy-bin</id>
    <phase>package</phase>
    <goals>
        <goal>copy-resources</goal>
    </goals>
    <configuration>
        <resources>
            <resource>
                <directory>src/main/bin</directory>      <!-- 文件来源 -->
            </resource>
        </resources>
        <outputDirectory>${project.build.directory}/pack/bin</outputDirectory>      <!-- 资源文件的输出目录 -->
    </configuration>
</execution>
</executions>
</plugin>

```

maven-antrun-plugin

This plugin allows you to run Ant scripts within a Maven build, which can be helpful for [performing custom tasks](#) that Maven doesn't directly support.

Usage

Running custom scripts.

Performing tasks like copying files, modifying configuration files, or running shell commands.

Integrating existing Ant-based build tasks into a Maven project.

```

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>3.1.0</version>
            <executions>
                <execution>
                    <id>run-ant</id>
                    <phase>generate-sources</phase>
                    <!--
                        Specifies the Maven build phase during which the plugin will execute,
                        e.g., generate-sources, compile, or package.
                    -->
                    <configuration>
                        <tasks>
                            <!--
                                Contains standard Ant tasks, such as echo, mkdir, copy, or other custom Ant
                                tasks.
                            -->
                            These tasks run during the specified phase.
                        <!--
                            <echo message="Running Ant tasks in Maven build..." />
                            <mkdir dir="${project.build.directory}/custom-dir" />
                            <copy file="src/main/resources/sample.txt"
                                todir="${project.build.directory}/custom-dir" />
                        </tasks>
                    </configuration>
                    <goals>
                        <goal>run</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

```

        </execution>
    </executions>
</plugin>
</plugins>
</build>

```

maven-enforcer-plugin

The maven-enforcer-plugin is a powerful tool for Maven projects used to enforce certain rules and constraints to maintain consistency and quality across projects.

It helps catch potential issues early in the build process by checking for things like Java version, banned dependencies, transitive dependency conflicts, and more.

Env Check

The maven-enforcer-plugin can enforce checks on both the local environment and the global environment by validating the project's build environment and dependencies.

```

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-enforcer-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <id>enforce-rules</id>
                    <goals>
                        <goal>enforce</goal>
                    </goals>
                    <configuration>
                        <rules>
                            <!-- Example rules -->
                            <requireJavaVersion>
                                <version>[11,)</version>
                                <!-- Require Java 11 or newer -->
                            </requireJavaVersion>
                            <banDuplicateClasses/>
                            <requireMavenVersion>
                                <version>[3.6,</version>
                                <!-- Require Maven 3.6 or newer -->
                            </requireMavenVersion>
                        </rules>
                        <failFast>true</failFast>
                            <!-- Stop on first rule violation -->
                            <reactorModuleConvergence />
                            <!--
                                Enforces that all modules within a multi-module Maven project
                                must have consistent versions for dependencies or artifacts that
                                are shared between them.
                            -->
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>

```

maven-surefire-plugin

The maven-surefire-plugin is a popular Maven plugin used to run unit tests in a Maven project.

It is primarily used during the test phase of the build lifecycle to execute tests written in Java.

The plugin integrates with various testing frameworks such as JUnit, TestNG, and others, allowing developers to run tests and generate reports.

Annotation @Disabled not works

The Maven Surefire Plugin is responsible for running your tests.

If you're using an older version of the plugin, it might not fully support JUnit 5 annotations like @Disabled.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M9</version>
      <configuration>
        <includes>
          <include>**/*Test.java</include>
        </includes>
        <test>com.example.MyTest</test>
        <!-- Specify which tests to run. -->

        <environmentVariables>
          <MY_ENV_VAR>my-value</MY_ENV_VAR>
        </environmentVariables>

        <systemPropertyVariables>
          <environment>dev</environment>
        </systemPropertyVariables>
        <!-- Set system properties for the tests. -->

        <excludes>
          <exclude>**/IntegrationTest.java</exclude>
        </excludes>
        <parallel>methods</parallel>
        <!-- Defines the parallel mode (e.g., methods, classes, both) -->

        <threadCount>4</threadCount>
        <!-- Number of threads for parallel execution. -->

        <forkCount>2</forkCount>
        <!-- Number of JVM instances for running tests. -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

maven-compiler-plugin

概念

用于编译项目代码

pow.xml

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.0</version>
  <configuration>
    <source>1.8</source>
      Specify JDK version for Maven compilation.
      // 1.8 1.9 11 17
    <target>1.8</target>
```

```
<encoding>UTF-8</encoding>
</configuration>
</plugin>
```

maven-assembly-plugin

支持定制化打包方式，例如 apache 项目的打包方式

允许用户将项目输出与它的依赖项、模块、站点文档、和其他文件一起组装成一个可分发的归档文件。说白了就是：结构定制化的打包。

pom.xml

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <descriptors>
      <descriptor>${basedir}/src/main/assembly/assembly.xml</descriptor>    <!-- 这个是 assembly 所在位置；${basedir}是指项目的根路径 -->
    </descriptors>
    <finalName>${project.artifactId}</finalName>    <!-- 打包解压后的目录名； ${project.artifactId}是指：项目的artifactId-->
    <outputDirectory>${project.build.directory}/release</outputDirectory>    <!-- 打包压缩包位置-->
    <encoding>UTF-8</encoding>    <!-- 打包编码 -->
  </configuration>
  <executions>
    <execution> <!-- 配置执行器 -->
      <id>make-assembly</id>
      <phase>package</phase>    <!-- 绑定到 package 生命周期阶段上 -->
      <goals>
        <goal>single</goal>    <!-- 只运行一次 -->
      </goals>
    </execution>
  </executions>
</plugin>
```

assembly.xml

maven-jar-plugin

maven 默认打包插件【springboot 默认使用该方式打包】，用来创建 project jar

仅负责将源码打成 jar 包，不能独立运行。

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.2.0</version>
  <configuration>
    <includes>    <!-- 打包时包含的文件配置，在这个工程中，只打包 com 文件夹 -->
      <include>
        **/com/**
      </include>
    </includes>
```

```

<archive>
    <manifest>
        <addClasspath>true</addClasspath>      <!-- 配置加入依赖包 -->
        <classpathPrefix>lib/</classpathPrefix>
        <useUniqueVersions>false</useUniqueVersions>
        <mainClass>com.ccccit.springdockerserver.SpringDockerServerApplication</mainClass>      <!-- Spring Boot 启动
类(自行修改) -->
    </manifest>
    <manifestEntries>
        <Class-Path>resources/</Class-Path>          <!-- 外部资源路径加入 manifest.mf 的 Class-Path -->
    </manifestEntries>
</archive>
<outputDirectory>${project.build.directory}/pack/</outputDirectory>      <!-- jar 输出目录 -->
</configuration>
</plugin>

```

```

<build>
    <plugins>
        <plugin>
            <!-- Build an executable JAR -->
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.1.0</version>
            <configuration>
                <archive>
                    <manifest>
                        <!-- <addClasspath>true</addClasspath>-->
                        <!-- <classpathPrefix>classes/</classpathPrefix>-->
                        <mainClass>com.simijar.Main</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.6.0</version>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
                <archive>
                    <manifest>
                        <mainClass>com.simijar.Main</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>

```

```
<executions>
  <execution>
    <id>make-assembly</id>
    <phase>package</phase>
    <goals>
      <goal>single</goal>
    </goals>
  </execution>
</executions>
</plugin>
</plugins>
</build>
```

maven-shade-plugin

maven-dependency-plugin

负责将各种依赖打包。也可以根据你的设置，将所打的依赖 jar 包输出到指定位置。

maven-dependency-plugin 最大的用途是帮助分析项目依赖，dependency:list 能够列出项目最终解析到的依赖列表，dependency:tree 能进一步的描绘项目依赖树。

maven-dependency-plugin 还有很多目标帮助你操作依赖文件，例如 dependency:copy-dependencies 能将项目依赖从本地 Maven 仓库复制到某个特定的文件夹下面。

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.2.0</version>
  <executions>      <!-- 复制依赖 -->
    <execution>
      <id>copy-dependencies</id>      <!-- 这里的 id 可以随意写，与下面的 goal 无名称上的必然联系 -->
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/pack/lib</outputDirectory>      <!-- 依赖包 输出目录 -->
      </configuration>
    </execution>
  </executions>
</plugin>
```

spring-boot-maven-plugin

Core

The spring-boot-maven-plugin is a tool provided by Spring Boot that simplifies the process of building and running Spring Boot applications in a Maven environment.

Key Features

Package Applications

The plugin packages Spring Boot applications into an executable JAR or WAR file with all dependencies bundled, making the application easy to run and deploy.

Repackage JARs

It can convert a standard JAR or WAR to an executable format by repackaging it with necessary dependencies and setting up the Spring Boot loader.

Run Applications

The plugin enables you to run your application directly using Maven, simplifying testing and development.

Dependency Management

It includes support for dependency management specific to Spring Boot, which helps in aligning dependency versions and ensures compatibility with the Spring Boot version in use.

pow.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <fork>true</fork>
        A new virtual machine will be created for execution when Maven compiles.
        Without this configuration, devtools will not take effect. This newly created JVM is the fork here. It will
        compile a little slower, but the isolation is very good
        <verbose>true</verbose>
        <skip>true</skip>
        Prevent the plugin from running during a specific Maven build. This is particularly useful when you want to
        skip running or repackaging the Spring Boot application as part of certain profiles or stages in your build
        pipeline.
        The default value is false
      <executable>C:\Program Files\Java\jdk1.6.0_33</executable>
      <compilerVersion>1.3</compilerVersion>
      <meminitial>128m</meminitial>
      <maxmem>1024m</maxmem>
      <compilerArgs>
        <arg>-XX:MaxPermSize=256m</arg>
      </compilerArgs>
      <includeSystemScope>true</includeSystemScope>
      <mainClass>com.saidake.ConsumerApplication</mainClass>
      <outputDirectory>/simi</outputDirectory>
      Where to place the jar file
      <finalName>app</finalName>
```

```
The output jar file name  
</configuration>  
<dependencies>  
  <dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>springloaded</artifactId>  
    <version>1.2.6.RELEASE</version>  
  </dependency>  
</dependencies>  
</plugin>  
  
</plugins>  
</build>
```

executions / execution / goals

spring-boot:build-image

Package an application into an OCI image using a buildpack, forking the lifecycle to make sure that package ran.

This goal is suitable for command-line invocation.

If you need to configure a goal execution in your build, use build-image-no-fork instead.

Spring-boot:build-image-no-fork

Package an application into an OCI image using a buildpack, but without forking the lifecycle. This goal should be used when configuring a goal execution in your build. To invoke the goal on the command-line, use build-image instead.

Spring-boot:build-info

Generate a build-info.properties file based on the content of the current MavenProject.

Spring-boot:help

Display help information on spring-boot-maven-plugin. Call mvn spring-boot:help -Ddetail=true -Dgoal=<goal-name> to display parameter details.

Spring-boot:process-aot

Invoke the AOT engine on the application.

Spring-boot:process-test-aot

Invoke the AOT engine on tests.

Spring-boot:repackage

Repackage existing JAR and WAR archives so that they can be executed from the command line using java -jar. With layout=NONE can also be used simply to package a JAR with nested dependencies (and no main class, so not

executable).

Spring-boot:run

Run an application in place.

Required		
parametersName	Type	Default
classesDirectory	File	\${project.build.outputDirectory}

Spring-boot:start

Start a spring application. Contrary to the run goal,
this does not block and allows other goals to operate on the application.

This goal is typically used in integration test scenario where the application is started before a test suite and stopped after.

Spring-boot:stop

Stop an application that has been started by the “start” goal. Typically invoked once a test suite has completed.

Spring-boot:test-run

Run an application in place using the test runtime classpath. The main class that will be used to launch the application is determined as follows: The configured main class, if any. Then the main class found in the test classes directory, if any. Then the main class found in the classes directory, if any.

configuration

addResources

Add maven resources to the classpath directly,
this allows live in-place editing of resources.

Duplicate resources are removed from target/classes to prevent them from appearing twice
if ClassLoader.getResources() is called.

Please consider adding spring-boot-devtools to your project instead as it provides this feature and many more.

Name	addResources
Type	boolean
Default value	false
User property	spring-boot.run.addResources
Since	1.0.0

classesDirectory

Directory containing the classes and resource files that should be used to run the application.

Name	classesDirectory
Type	java.io.File

Default value \${project.build.outputDirectory}

User property

Since 1.0.0

tomcat7-maven-plugin

pom.xml >>

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.1</version>
      <configuration>
        <port>41200</port>
        <path>/</path>
      </configuration>
    </plugin>
  </plugins>
</build>
```

javafx-maven-plugin

```
<dependency>
  <groupId>de.roskenet</groupId>
  <artifactId>springboot-javafx-support</artifactId>
  <version>2.1.6</version>
</dependency>

<dependency>
  <groupId>de.roskenet</groupId>
  <artifactId>springboot-javafx-test</artifactId>
  <version>1.3.0</version>
  <scope>test</scope>
</dependency>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>com.zenjava</groupId>
      <artifactId>javafx-maven-plugin</artifactId>
      <version>8.7.0</version>
      <configuration>
        <mainClass>com.saidake.SdkApplication</mainClass>
      </configuration>
    </plugin>
  </plugins>
```

```
</build>
```

cds-maven-plugin

Security

jjwt

Configuration

Add Dependencies

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Create a JWT Utility Class

Create a utility class to generate and validate JWT tokens:

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Component
public class JwtUtil {

    private String SECRET_KEY = "secret"; // Use a more secure secret in production

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }
}
```

```

}

public String generateToken(String username) {
    Map<String, Object> claims = new HashMap<>();
    return createToken(claims, username);
}

private String createToken(Map<String, Object> claims, String subject) {
    return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new
Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact();
}

public Boolean validateToken(String token, String username) {
    final String extractedUsername = extractUsername(token);
    return (extractedUsername.equals(username) && !isTokenExpired(token));
}
}

```

Create a JWT Filter

This filter will intercept requests and check if the JWT token is present and valid:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {
        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUsername(jwt);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

```

```

        if (jwtUtil.validateToken(jwt, userDetails.getUsername())) {
            UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new
UsernamePasswordAuthenticationToken(
                userDetails, null, userDetails.getAuthorities());
            usernamePasswordAuthenticationToken
                .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
        }
    }
    chain.doFilter(request, response);
}
}

```

[jjwt]

Jwts

```
package io.jsonwebtoken;
```

```
public final class Jwts
```

```

public static JwtParser parser() {
    return new DefaultJwtParser();
}

public static JwtBuilder builder() {
    return new DefaultJwtBuilder();
}
```

JwtBuilder

```
package io.jsonwebtoken;
```

```
public interface JwtBuilder extends ClaimsMutator<JwtBuilder>
```

```
JwtBuilder setClaims(Claims claims);
```

```
Claims setSubject(String sub);
```

```
Claims setIssuedAt(Date iat);
```

```
Claims setExpiration(Date exp);
```

```
JwtBuilder signWith(SignatureAlgorithm alg, String base64EncodedSecretKey);
```

```
JwtBuilder signWith(SignatureAlgorithm alg, byte[] secretKey);
```

JwtParser

```
package io.jsonwebtoken;
```

```
public interface JwtParser
```

```
JwtParser setSigningKey(byte[] key);
```

```
JwtParser setSigningKey(String base64EncodedKeyBytes);
```

```
Jwt<Header, Claims> parseClaimsJwt(String claimsJwt) throws ExpiredJwtException, UnsupportedJwtException,
MalformedJwtException, SignatureException, IllegalArgumentException;
```

Jwt

```
package io.jsonwebtoken;
```

```
public interface Jwt<H extends Header, B>
```

```
H getHeader();
```

```
B getBody();
```

Claims

```
package io.jsonwebtoken;  
public interface Claims extends Map<String, Object>, ClaimsMutator<Claims>
```

In the context of JWT (JSON Web Tokens), **claims** are pieces of information asserted about an entity (typically, the user) and encoded within the JWT.

Claims are essentially key-value pairs that are included in the payload of the JWT. These claims can contain information such as user details, permissions, or any other data needed for the token's purpose.

jasypt-spring-boot-starter

为项目中所有密码加密

```
public final class BasicTextEncryptor implements TextEncryptor 基础文本加密  
public String encrypt(String message) 加密字符串  
public String decrypt(String encryptedMessage) 解析加密字符串  
public void setPassword(String password) 设置加密解密密码
```

jasypt

Configuration

```
jasypt:  
  encryptor:  
    password: wangmaox #加密的密钥，自定义即可  
    algorithm: PBEWithMD5AndDES #指定解密算法
```

custom-config:

```
  infos:  
    email: ENC(gqtN4w5o5JrJR0armxigJ+L2HCfPYBVP3Q3rx7ImjDaluwJA7eMRvw==)  
      # Jasypt 加密，格式为 ENC(加密结果)  
      # ENC() 就是它的标识，程序启动的时候，会自动解密其中的内容，如果解密失败，则会报错。
```

Property Encrypt

```
public static String getPassword(InputStream propertiesFile, String property, String key) throws IOException  
  
  BasicTextEncryptor encryptor = new BasicTextEncryptor();  
  encryptor.setPassword(key);  
  Properties encProps = new EncryptableProperties(encryptor);  
  encProps.load(propertiesFile);  
  return encProps.getProperty(property);  
}
```

[jasypt]

BasicTextEncryptor

```
package org.jasypt.util.text;  
public final class BasicTextEncryptor implements TextEncryptor PBEWithMD5AndDES  
  
  public String encrypt(String message) Directly encrypt  
  public void setPassword(String password)
```

StrongTextEncryptor

(PBEWithMD5AndTripleDES)

AES256TextEncryptor

(PBEWithHMACSHA512AndAES_256)

EncryptableProperties

```
package org.jasypt.properties;  
public final class EncryptableProperties extends Properties
```

public String **getProperty**(String key) Use the given encryptor to decode mapping value.

oauth2

Core

How Does the OAuth2 Resource Server Validate the Token?

Receive the Token

The resource server receives an access token **in the HTTP request**, usually in the Authorization header as a Bearer token.

Token Introspection

The resource server introspects the token to validate it. There are two common ways to do this:

Introspection Endpoint

The resource server can **call the authorization server's introspection endpoint** to validate the token.

This endpoint **returns metadata** about the token, including its validity, the associated user, and the scopes granted.

Locally Verify the Token

If the token is a JSON Web Token (JWT), the resource server can validate the token locally by verifying the signature, issuer, audience, expiration, and other claims.

Using the Introspection Endpoint

Configuration

The resource server is configured with the URL of the authorization server's introspection endpoint.

HTTP Request

The resource server makes an HTTP POST request to the introspection endpoint with the access token.

Example:

```
POST /introspect HTTP/1.1  
Host: auth-server.com  
Authorization: Basic <client_id:client_secret>  
Content-Type: application/x-www-form-urlencoded  
  
token=<access_token>
```

Response Handling

The authorization server responds with a JSON object indicating the token's validity and associated metadata.

Example Response:

```
{  
  "active": true,  
  "scope": "read write",  
  "username": "jdoe",  
  "exp": 1615279052,  
  "sub": "jdoe",  
  "aud": "resource-server",  
  "iss": "auth-server.com"  
}
```

Validation

The resource server checks **the active field** to determine if the token is valid.

It also verifies the token's **scope**, **audience**, and **expiration** to ensure the request is authorized.

Locally Verifying JWT Tokens

Configuration

The resource server is configured with the public key or a URL to fetch the public key of the authorization server to verify the JWT signature.

Decode the Token

The resource server **decodes** the JWT, which typically has three parts: header, payload, and signature.

Example JWT:

```
header.payload.signature
```

Verify the Signature

The resource server uses the public key to verify the token's signature.

Validate Claims

The resource server checks the standard claims such as iss (issuer), aud (audience), and exp (expiration time) to ensure the token is valid and was issued for the intended resource.

Scope and Permissions

The resource server verifies that the token has **the necessary scopes** and **permissions** to access the requested resource.

Configuration

pom.xml

Include the necessary dependencies for Spring Security and OAuth 2.0 support.

```
<!-- Spring Security Starter -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<!-- OAuth 2.0 Resource Server Starter -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

application.yml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/auth/realm/your-realm
            # This is the URI of the authorization server that issues the JWT tokens.
            # It can be a custom authorization server or an external provider like Keycloak, Okta, or Auth0.
```

Implement Security Configuration

Create a security configuration class to define security rules for your resource server.

Validate Tokens

The oauth2ResourceServer().jwt() method in the security configuration ensures **that incoming requests are authenticated using JWT tokens** issued by the authorization server.

The issuer URI specified in the configuration will be used to validate the tokens.

Example Scenario

User Authentication:

A user logs into **a client application** (e.g., a web app) via **the authorization server**.

The authorization server issues an access token.

Accessing Resource Server:

The client application includes the access token in the request to the resource server.

The resource server validates the token by checking its issuer, signature, and expiry.

Enforcing Permissions:

The resource server checks the scopes and permissions within the token to ensure the user is authorized to perform the requested action.

Define a SecurityFilterChain bean

```
@Configuration
@EnableWebSecurity
public class OAuth2ResourceServerSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .antMatchers("/public/**").permitAll() // Public endpoints
                    .anyRequest().authenticated() // Secured endpoints
            )
            .oauth2ResourceServer((oauth2ResourceServer) ->
                oauth2ResourceServer.jwt(
                    (jwt) -> jwt.decoder(jwtDecoder())
                )
            );
        return http.build();
    }

    @Bean
    public JwtDecoder jwtDecoder() {
        return NimbusJwtDecoder.withPublicKey(this.key).build();
    }
}
```

Extend WebSecurityConfigurerAdapter

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .antMatchers("/public/**").permitAll() // Public endpoints
                    .anyRequest().authenticated() // Secured endpoints
            )
    }
}
```

rsa

依赖:

```
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcpk15on</artifactId>
    <version>1.58</version>
    <scope>compile</scope>
</dependency>
```

spring-security-acl (jar)

访问权限控制

config >

```
@EnableGlobalMethodSecurity(prePostEnabled = true) 开启全局安全验证
```

```
public class AclConfig extends GlobalMethodSecurityConfiguration {
```

```
    private static final Logger logger = LoggerFactory.getLogger(AclConfig.class);
```

```
}
```

```
public class AclAuthorizationStrategyImplWrapper implements AclAuthorizationStrategy { 其会用来在对 Acl 进行某些操作  
时检查当前用户是否具有对应的权限
```

所谓权限，就是一个字符串。一般不会重复，所谓权限检查，就是查看用户权限列表中是否含有匹配的字符串

在 Security 提供的 UserDetailsService 默认实现 JdbcDaoImpl 中，角色和权限都存储在 authorities 表中

```
private final GrantedAuthority gaGeneralChanges;  
private final GrantedAuthority gaModifyAuditing;  
private final GrantedAuthority gaTakeOwnership;  
private SidRetrievalStrategy sidRetrievalStrategy = new SidRetrievalStrategyImpl();
```

```
public AclAuthorizationStrategyImplWrapper(GrantedAuthority... auths) {  
    Assert.isTrue(auths != null && (auths.length == 3 || auths.length == 1),  
        "权限不能为空");  
    if (auths.length == 3) {  
        gaTakeOwnership = auths[0];  
        gaModifyAuditing = auths[1];  
        gaGeneralChanges = auths[2];  
    } else {  
        gaTakeOwnership = gaModifyAuditing = gaGeneralChanges = auths[0];  
    }  
}
```

controller >

```
@PreAuthorize("hasRole('ADMIN')") 权限验证 hasRole("ADMIN")。那它实际上查询的是用户权限集合中是否存在字符串"ROLE_ADMIN"
```

Spring Security 的 Acl 功能需要使用到四张数据库表，分别为 acl_sid, acl_class, acl_object_identity, acl_entry

acl_sid id 主键 sid 字符串类型的 sid (varchar) principal 是否用户 (Boolean)

表 acl_sid 是用来保存 Sid 的。一种是基于用户的 Sid，叫 PrincipalSid；另一种是基于 GrantedAuthority 的 Sid，叫 GrantedAuthoritySid

sid 字段存放的是用户名或者是 GrantedAuthority 的字符串表示，principal 字段是用来区分对应的 Sid 是用户还是 GrantedAuthority 的

Acl 中对象的权限是用来授予给 Sid 的，Sid 有用户和 GrantedAuthority 之分，所以我们的对象权限是可以用来授予给用户或 GrantedAuthority 的。

acl_class id 主键 class 对象类型的全限定名

表 acl_class 是用来保存对象类型的，字段 class 中保存的是对应对象的全限定名。Acl 需要使用它来区分不同的对象类型

`acl_object_identity` `id` 主键 `object_id_class` 关联 `acl_class`, 表示对象类型 (number) `object_id_identity` 对象的主键
(唯一 number)

`parent_object` 父对象的 id, 关联 `acl_object_identity` (number) `owner_sid` 拥有者的 sid, 关联 `acl_sid` (number)
 `entries_inheriting` 是否继承父对象的权限, 继承删除权限等 (boolean)

表 `acl_object_identity` 是用来存放需要进行访问控制的对象的信息的。其保存的信息有对象的拥有者、对象的类型、对象的主键、对象的父对象和是否继承父对象的权限。

`acl_entry` `id` 主键 `acl_object_identity` 对应 `acl_object_identity` 的 id `ace_order` 所属 Acl 的权限顺序 (number) `sid`
对应 `acl_sid` 的 id (number)

`mask` 权限对应的掩码 (number) `granting` 是否授权 (boolean)

`audit_success` 暂未发现其作用, Acl 中有一个更新其值的方法, 但未见被调用。 (boolean) `audit_failure` 认证失败
(boolean)

表 `acl_entry` 是用于存放具体的权限信息的, 其描述的就是某个主体 (Sid) 对某个对象 (`acl_object_identity`) 是否
(granting) 拥有某种权限 (mask)。

当同一对象 `acl_object_identity` 在 `acl_entry` 表中拥有多条记录时, 就会使用 `ace_order` 来标记对应的顺序,
其对应于往 Acl 中插入 `AccessControlEntry` 时的位置, 在进行权限判断时也是依靠 `ace_order` 的顺序来进行的,
`ace_order` 越小的越先进行判断。`ace` 是 Access Control Entry 的简称。

AclAuthorizationStrategy

其构造需要接收一个或三个 `GrantedAuthority` 参数, 控制 Acl 操作的权限, (用户名, 密码, 权限)

更改 Acl 对应对象的所有者需要的权限。

更改 Acl 中包含的某个 `AccessControlEntry` 的 audit 信息 (对应 `acl_entry` 表中的 `is_audit_success` 和 `is_audit_failure` 字段)
需要的权限

更改其它如增、删、改 Acl 中所包含的 `AccessControlEntry` 等需要的权限。

这些权限的鉴定是我们在操作 Acl 时由 Spring Security Acl 内部进行判断的, 我们只需要在这里定义就好。

当 Acl 对应的所有者对 Acl 进行操作时, 不管其是否拥有指定需要的权限, 除了改变 audit 信息之外的所有操作默认都是被
允许的。

当只有一个参数时表示三者共用一个 `GrantedAuthority`。

spire.xls.free

EXCEL 图片删除工具包, 支持 excel 图片删除, 插入等操作

Servers

Eureka

Core

Eureka is a service registry designed specifically for microservice architectures.

It acts as a central hub where microservices can register themselves and discover each other.

This enables services to dynamically locate and communicate with their peers, ensuring a resilient and scalable system.

Workflow

Spring Boot Initialization

When a Spring Boot application starts, it performs classpath scanning and loads configuration properties.

The `@SpringBootApplication` annotation triggers component scanning and auto-configuration.

If the `spring-cloud-starter-netflix-eureka-client` dependency is included, the `EurekaClientAutoConfiguration` class is loaded.

The `EurekaClientAutoConfiguration` class configures a Eureka client in the Spring Boot application.

It creates and configures a `DiscoveryClient` bean if the application is annotated with `@EnableEurekaClient` or
`@EnableDiscoveryClient`.

The `EurekaInstanceConfigBean` provides the instance-specific configuration for Eureka, such as `hostname`, lease
renewal interval, lease expiration duration, etc.

The `EurekaClientConfigBean` holds the client-specific configuration, such as the Eureka server URL, fetch registry interval, and other settings.

Application Registration with Eureka

During startup, the `DiscoveryClient` registers the application instance with the Eureka server using the `EurekaHttpClient` class.

The `register()` method of `EurekaHttpClient` sends a POST request to the Eureka server to register the instance.

```
public EurekaHttpResponse<Void> register(InstanceInfo info) {  
    // Send POST request to Eureka server to register instance  
}
```

Request Process in Eureka

Service Registration

The `DiscoveryClient` periodically sends heartbeats to the Eureka server to renew the registration using the `renew()` method of `EurekaHttpClient`.

```
public EurekaHttpResponse<InstanceInfo> renew(String appName, String id) {  
    // Send PUT request to renew registration  
}
```

Service Discovery

When a service wants to discover other services, it uses the `DiscoveryClient` to fetch the registry from Eureka.

The `getInstancesByVipAddress()` method retrieves instances by their VIP address (logical name).

Service Request Process:

When a microservice makes a request to another service, it first fetches the service instance information from Eureka using the `DiscoveryClient`.

```
@Autowired  
private DiscoveryClient discoveryClient;  
  
public void callService() {  
    List<ServiceInstance> instances = discoveryClient.getInstances("SERVICE-NAME");  
    if (instances != null && !instances.isEmpty()) {  
        // Use the instance information to make a request  
        ServiceInstance instance = instances.get(0);  
        String url = instance.getUri().toString() + "/endpoint";  
        // Make HTTP request to the service  
    }  
}
```

Configuration

Eureka Server

Add Dependencies

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

Enable Eureka Server

Annotate your main application class with `@EnableEurekaServer`:

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}

```

Configure Application Properties

```

spring.application.name=eureka-server
server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

eureka.instance.hostname=localhost

```

Basic Configuration

`eureka.instance.hostname=<host-name>`

Specifies the hostname that will be used for accessing the Eureka Server.

`eureka.instance.hostname=my-eureka-server`

`eureka.client.register-with-eureka=<boolean>`

Indicates whether the Eureka Server itself should register with another Eureka Server. Typically set to false for the standalone server.

`eureka.client.register-with-eureka=false`

`eureka.client.fetch-registry=<boolean>`

Specifies whether the Eureka Server should fetch the registry information [from other Eureka servers](#). Set to false for standalone servers.

`eureka.client.fetch-registry=false`

Server Configuration

`eureka.server.enable-self-preservation=<boolean>`

Enables or disables the self-preservation mode, which helps the server to remain operational in case of network issues by not expiring instances quickly.

`eureka.server.enable-self-preservation=true`

`eureka.server.eviction-interval-timer-in-ms=<milliseconds>`

Specifies the interval (in milliseconds) at which the server checks for expired instances and removes them from the registry.

`eureka.server.eviction-interval-timer-in-ms=6000`

Instance Configuration

`eureka.instance.lease-renewal-interval-in-seconds=<seconds>`

Specifies the interval (in seconds) at which the client sends heartbeats to the server to indicate that it is still alive.

`eureka.instance.lease-renewal-interval-in-seconds=30`

`eureka.instance.lease-expiration-duration-in-seconds=<seconds>`

Specifies the duration (in seconds) after which the server expires the instance if no heartbeats are received.

`eureka.instance.lease-expiration-duration-in-seconds=90`

`eureka.instance.prefer-ip-address=<boolean>`

If true, the Eureka Server will use the IP address instead of the hostname for registration.

`eureka.instance.prefer-ip-address=true`

Peer Awareness

`eureka.client.service-url.defaultZone=<URL>`

Specifies the list of Eureka servers to which this server will connect. Useful in a multi-zone setup.

```
eureka.client.service-url.defaultZone=http://peer1:8761/eureka/,http://peer2:8761/eureka/  
eureka.client.availability-zones=<zone-list>
```

Configures the availability zones for the Eureka server.

```
eureka.client.availability-zones.default=zone1,zone2
```

Advanced Configuration

```
eureka.client.initial-instance-info-replication-interval-seconds=<seconds>
```

Interval (in seconds) at which the Eureka server replicates initial instance information to peer servers.

```
eureka.client.initial-instance-info-replication-interval-seconds=40
```

```
eureka.client.registry-fetch-interval-seconds=<seconds>
```

Frequency (in seconds) at which the client fetches the registry information from the server.

```
eureka.client.registry-fetch-interval-seconds=30
```

```
eureka.client.registry-refresh-single-vip-address=<VIP-address>
```

VIP address for the registry refresh.

```
eureka.client.registry-refresh-single-vip-address=my-vip-address
```

Security Configuration

```
eureka.server.enable-basic-auth=<boolean>
```

Enable basic authentication for Eureka server endpoints.

```
eureka.server.enable-basic-auth=true
```

```
eureka.client.basic-auth-username=<username>
```

Username for basic authentication.

```
eureka.client.basic-auth-username=user
```

```
eureka.client.basic-auth-password=<password>
```

Password for basic authentication.

```
eureka.client.basic-auth-password=pass
```

```
eureka.dashboard.path=<path>
```

Customizes the path to the Eureka dashboard.

```
eureka.dashboard.path=/custom-eureka
```

Run the Eureka Server

Run the application, and the Eureka server will be available at <http://localhost:8761>.

Eureka Client

Create a Spring Boot Project for Eureka Client

Again, use Spring Initializr to create a new Spring Boot project. Add the following dependencies:

Add Dependencies in `pom.xml` or `build.gradle`:

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

Enable Eureka Client

Annotate your main application class with `@EnableEurekaClient` (optional as Spring Boot will automatically register the application with Eureka):

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication
@EnableEurekaClient
public class EurekaClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaClientApplication.class, args);
    }
}

```

Configure Application Properties

```

spring.application.name=eureka-client
server.port=8080

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

```

Configuration (Optional)

If you prefer to configure `EurekaInstanceConfigBean` programmatically in a Spring Boot `@Configuration` class, it might look like this:

In Netflix's Eureka, a data center refers to the location or environment where the instance is deployed, such as Amazon Web Services (AWS) or a private on-premise data center.

Eureka uses an interface called `DataCenterInfo` to represent the type of data center. The most common implementations are:

Amazon AWS (`AmazonInfo`)

Used when running services on AWS EC2 instances. It contains additional details like AWS instance ID, availability zone, and more.

MyOwn (Basic)

Default implementation for non-AWS environments, representing a private data center or a local setup.

```

import org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;

@Configuration
public class EurekaConfig {

    @Bean
    public EurekaInstanceConfigBean eurekaInstanceConfig(Environment environment) {
        EurekaInstanceConfigBean config = new EurekaInstanceConfigBean();
        config.setPreferIpAddress(true);
        config.setIpAddress("192.168.1.100");
        config.setNonSecurePort(8080);
        config.setLeaseRenewalIntervalInSeconds(10);
        config.setLeaseExpirationDurationInSeconds(30);
        config.getMetadataMap().put("instanceId", environment.getProperty("spring.application.name")
            + ":" + environment.getProperty("spring.application.instance_id"));

        // Create AmazonInfo instance for AWS
        AmazonInfo amazonInfo = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
        config.setDataCenterInfo(amazonInfo);

        return config;
    }
}

```

Run the Eureka Client

Run the application, and it will register itself with the Eureka server.

Eureka 服务注册流程

服务提供者启动时向 EurekaServer 注册自己的信息，每 30 秒向 EurekaServer 发送心跳报告健康状态（eureka 会更新记录服务列表信息，心跳不正常会被剔除，消费者就可以拉取到最新的信息）

消费者根据服务名称从 EurekaServer 拉取可用服务列表，基于服务列表做负载均衡，从中选中一个服务后发起远程调用

自我保护机制：如果 Eureka Server 在 15 分钟内有超过 85% 的 Eureka Client 都没有正常的发送心跳过来（单机模式时尤为明显），那么 Eureka Server 就认为注册中心与客户端出现了网络故障，Eureka Server 自动进入自我保护机制。

eureka 这是会将不可用的服务暂时断开，并期望能够在接下来一段时间内接收到心跳信号，而不是直接剔除

阈值计算：Renews threshold = $\text{floor}[2 \times n \times 0.85]$ Eureka Server 期望每分钟收到客户端实例续约的阈值（n 为服务数量）

Renews threshold = $\text{floor}[2 \times (n+1) \times 0.85]$ 当 eureka-server 不注册自身时，Eureka Server 期望每分钟收到客户端实例续约的阈值

Renews (last min) = $\text{floor}[2 \times n]$ Eureka Server 最后 1 分钟收到客户端实例续约的总数
[spring-cloud-netflix-eureka-client]

EurekaInstanceConfigBean

```
package org.springframework.cloud.netflix.eureka;
@ConfigurationProperties("eureka.instance")
public class EurekaInstanceConfigBean implements CloudEurekaInstanceConfig, EnvironmentAware
    Configure how a secure application is registered with Eureka:
    Registering a secure application: To allow an app to be contacted over HTTPS, set the following flags in the
    EurekaInstanceConfig:
        eureka.instance.nonSecurePortEnabled=false
        eureka.instance.securePortEnabled=true

    public void setDataCenterInfo(DataCenterInfo dataCenterInfo)
```

Using Eureka on AWS

If the application is planned to be deployed to an AWS cloud, the Eureka instance must be configured to be AWS-aware. You can do so by customizing the [EurekaInstanceConfigBean](#) as follows:

```
@Bean
@Profile("!default")
public EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {
    EurekaInstanceConfigBean b = new EurekaInstanceConfigBean(inetUtils);
    AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
    b.setDataCenterInfo(info);
    return b;
}
```

[eureka-client]

EurekaClientConfig

```
package com.netflix.discovery;
```

```
public interface EurekaClientConfig    eureka client 配置
boolean shouldUseDnsForFetchingServiceUrls()  是否使用 DNS 方式获取 Eureka-Server URL 地址
                                                converters
```

CodecWrapperBase

```
package com.netflix.discovery.converters.wrappers;
public interface CodecWrapperBase 编码包装器
String codecName();          编码名称
boolean support(MediaType var1);  是否支持媒体类型
```

EncoderWrapper

```
package com.netflix.discovery.converters.wrappers;
public interface EncoderWrapper extends CodecWrapperBase
<T> String encode(T var1) throws IOException;  编码
<T> void encode(T var1, OutputStream var2) throws IOException;  编码
```

shared

ConfigClusterResolver

```
package com.netflix.discovery.shared.resolver.aws;
public class ConfigClusterResolver implements ClusterResolver<AwsEndpoint>
public List<AwsEndpoint> getClusterEndpoints()  获取集群端点，遍历 zone 和 eureka 地址的 map，添加所有断点到
List<AwsEndpoint> 中
```

RedirectingEurekaHttpClient

```
package com.netflix.discovery.shared.transport.decorator;
public class RedirectingEurekaHttpClient extends EurekaHttpClientDecorator 跳转 eureka 客户端
protected <R> EurekaHttpResponse<R> execute(RequestExecutor<R> requestExecutor) 执行
```

endpoint

EndpointUtils

```
package com.netflix.discovery.endpoint;
public class EndpointUtils 端点工具
public static Map<String, List<String>> getServiceUrlsMapFromConfig(EurekaClientConfig clientConfig, String instanceZone,
boolean preferSameZone) 从配置中获取所有 zone 和对应的 eureka 地址
{'defaultZone': 'http://127.0.0.1:8761/eureka/'}
```

DiscoveryJerseyProvider

```
package com.netflix.discovery.provider;
public class DiscoveryJerseyProvider implements MessageBodyWriter<Object>, MessageBodyReader<Object>  发现
Jersey 提供者
public DiscoveryJerseyProvider(EncoderWrapper jsonEncoder, DecoderWrapper jsonDecoder)
```

[jersey-server]

spi

```
package com.sun.jersey.spi.container;
public interface WebApplication extends Traceable    web 应用程序
                                                    server
package com.sun.jersey.server.impl.application;
public final class WebApplicationImpl implements WebApplication    web 应用实现
private void _initiate(final ResourceConfig rc, final IoCComponentProviderFactory _provider)    初始化jersey web 应用程序
```

Nacos

Core

Nacos (which stands for Naming and Configuration Service) is an open-source project by Alibaba designed to provide dynamic service discovery, configuration management, and service management.

It helps microservices and distributed systems manage and maintain their configurations and services more efficiently.

Components

Nacos Server:

The central server component where services register themselves and clients query for service information.

It also stores configuration data and provides APIs for service discovery and configuration management.

Nacos Client:

The client-side component **that integrates with your microservices**. It handles the registration of services, service discovery, and fetching configuration data.

Clients can be integrated into different programming languages and frameworks, including Java, Go, and Python.

Nacos Console:

A web-based user interface for managing services and configurations.

Provides visual tools to view service health, perform service operations, and manage configurations.

Nacos Election Mechanism

Raft Consensus Algorithm:

Raft is a consensus algorithm that is designed to be easier to understand than other consensus algorithms like Paxos.

It manages **a replicated log** and ensures that multiple nodes in a distributed system **can agree on a common state** even in the presence of failures.

In the context of Nacos, Raft ensures that all configuration changes and service registrations are consistently replicated across all nodes in the cluster.

Roles in Raft:

Leader:

The node that is responsible for **managing the log** and **replicating entries to the follower nodes**. There is only one leader in the cluster at any time.

Followers:

Nodes that replicate the log entries from the leader. They **acknowledge the entries to the leader** to ensure consistency.

Candidates:

Nodes that are trying to become the leader. They **start an election process** if they do not hear from the current leader within a specified timeout.

Leader Election:

When a node starts, it begins as a follower. If it does not receive any messages from the leader within the election timeout, it **transitions to a candidate and starts an election**.

The candidate node **sends out RequestVote messages to all other nodes** in the cluster. If it receives a majority of votes, it becomes the leader.

Once a leader is elected, it **starts sending heartbeat messages (AppendEntries)** to the followers to maintain its leadership.

Term and Voting:

Raft operates in terms of discrete time periods called terms. Each term **begins with an election**. Terms are numbered with **consecutive integers**.

Each node maintains a **currentTerm variable** that increases monotonically.

When a candidate requests votes, it **includes its term number**. Nodes will vote for **at most one candidate per term**, on a first-come-first-served basis.

If a node receives a message with a higher term than its current term, it **updates its term** and **reverts to follower state**.

Log Replication:

The leader is responsible for **accepting log entries from clients** and **replicating them to the followers**.

Once an entry is replicated to a majority of the nodes, it is considered committed.

Followers append the log entries to their logs and send acknowledgments to the leader.

Handling New Configurations vs. Updates

New Configurations:

Initial Fetch:

When an application first starts, it **requests configurations from Nacos**. This can be done via an API call to fetch the current configurations.

Initialization:

New configurations are loaded and applied to the application. The configuration center **sends a full configuration set** if it's the first fetch.

Configuration Updates:

Delta Notification:

When a configuration is updated, Nacos **sends a notification to applications**. This notification includes the changes (delta) rather than the entire configuration.

Efficient Updates:

The application receives the update, processes only the changed parts, and applies these changes.

Configuration

Add Dependencies

```
<dependencies>
    <!-- Spring Cloud Alibaba Nacos Discovery -->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>2021.1</version>
    </dependency>

    <!-- Spring Cloud Alibaba Nacos Config -->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
        <version>2021.1</version>
    </dependency>

    <!-- Other dependencies -->
</dependencies>
```

Configure Nacos Properties

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
      config:
        server-addr: 127.0.0.1:8848
```

```

file-extension: yaml

# Optional: Customize the service name and group
spring:
  application:
    name: my-service
  cloud:
    nacos:
      discovery:
        group: MY_GROUP

```

Enable Nacos Discovery and Configuration

Annotate your Spring Boot application class with `@EnableDiscoveryClient` to enable service discovery and `@EnableNacosConfig` to enable Nacos configuration (if needed).

```

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableDiscoveryClient
public class NacosDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(NacosDemoApplication.class, args);
    }
}

```

Using Nacos Configuration

To use configuration properties from Nacos, you can annotate your configuration class with `@RefreshScope` and use `@Value` or `@ConfigurationProperties` to bind the properties.

Example with `@Value`

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RefreshScope
public class ConfigController {

    @Value("${my.config.value:default}")
    private String configValue;

    @GetMapping("/config")
    public String getConfigValue() {
        return configValue;
    }
}

```

Example with `@ConfigurationProperties`

```

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.stereotype.Component;

@Component
@RefreshScope
@ConfigurationProperties(prefix = "my.config")
public class MyConfigProperties {

    private String value;

    // Getter and Setter
    public String getValue() {
        return value;
    }
}

```

```
public void setValue(String value) {  
    this.value = value;  
}  
}
```

Running the Application

Make sure Nacos server is running and start your Spring Boot application.

Your application will register itself with Nacos and can use Nacos for configuration management.

Nacos Dashboard:

You can access the Nacos dashboard at <http://127.0.0.1:8848/nacos> to view and manage services and configurations.

Dynamic Refresh:

Use the `@RefreshScope` annotation to enable dynamic refresh of properties when configurations are updated in Nacos.

Spring Cloud Bus

Spring Cloud Bus is a powerful tool in the Spring Cloud ecosystem that allows you to propagate state changes across a distributed system.

It integrates with Spring Cloud Config to broadcast configuration changes, but it can also be used for other messaging needs in a microservices architecture.

Core

Spring Cloud Bus uses a message broker (such as RabbitMQ or Kafka) to broadcast state changes across microservices.

When a configuration change or any other event occurs, the change is broadcasted to all instances of the application.

You can configure a [Spring Cloud Gateway](#) to work with [Spring Cloud Bus](#) in the same application. While [Spring Cloud Gateway](#) and [Spring Cloud Bus](#) serve different purposes,

they can coexist and complement each other in a microservices architecture.

Default Exchanges and Queues

RabbitMQ

Default Configuration:

Exchange: `spring-cloud-bus`

This is a default direct exchange used by Spring Cloud Bus.

Queue: `spring-cloud-bus`

This queue is bound to the `spring-cloud-bus` exchange.

Routing Key

Typically not used directly by Spring Cloud Bus but is associated with message routing in RabbitMQ.

Automatic Setup:

When you include `spring-cloud-starter-bus-amqp`, Spring Cloud Bus will create and bind these default exchanges and queues if they do not already exist.

You generally do not need to configure these manually unless you want to customize their behavior.

If you have created a custom RabbitMQ exchange for your business, Spring Cloud Bus will not automatically use it for propagating events unless you specifically configure it to do so.

Kafka

Default Configuration:

Topic: `springCloudBus`

This is the default topic used by Spring Cloud Bus for event propagation.

Automatic Setup:

When using `spring-cloud-starter-bus-kafka`, Spring Cloud Bus will use the default topic for messaging.

You do not need to manually create this topic unless you want to customize it.

Workflow

Initial Setup

Microservices are configured to listen for updates via Spring Cloud Bus. They are connected to a message broker.

Change Event

An administrator changes configuration properties in a configuration server.

Trigger Refresh

An application triggers a configuration refresh event using the /actuator/bus-refresh endpoint.

Broadcast

The event is broadcasted to the message broker.

Propagation

All services subscribed to the message broker receive the refresh event.

Update

Each service retrieves the updated configuration and applies it.

Configuration

Spring Cloud Bus Server

Add Dependencies

For RabbitMQ:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

For Kafka:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>
```

To trigger a configuration refresh and broadcast it to all clients, use the /actuator/bus-refresh endpoint. This requires the

Actuator dependency:

Add the Actuator dependency:

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

Configure the Message Broker

Configure the message broker connection settings and enable Spring Cloud Bus.

For RabbitMQ:

```
spring:
  cloud:
    bus:
      enabled: true
      rabbitmq:
        host: localhost
        port: 5672
        username: guest
        password: guest
      rabbitmq:
        exchanges:
          bus: spring-cloud-bus
```

For Kafka:

```
spring:
```

```

cloud:
  bus:
    enabled: true
  kafka:
    bootstrap-servers: localhost:9092
  properties:
    key:
      serializer: org.apache.kafka.common.serialization.StringSerializer
    value:
      serializer: org.apache.kafka.common.serialization.StringSerializer

```

Publishing Events

Manually

Trigger a Refresh Event via HTTP Request:

```
curl -X POST http://localhost:8080/actuator/bus-refresh
```

This will send a refresh event **to all services subscribed to the bus**, causing them to refresh their configuration.

Bus Server Behavior:

When you trigger a refresh (e.g., using curl -X POST http://localhost:8080/actuator/bus-refresh), the Spring Cloud Bus infrastructure automatically publishes a RefreshRemoteApplicationEvent to the message broker. This action is handled internally by Spring Cloud Bus, and **you do not need to configure custom event publishers** for this.

Event Broadcast:

The RefreshRemoteApplicationEvent is broadcast to all other services connected to the same message broker. This broadcasting is managed by the message broker (like RabbitMQ or Kafka) and Spring Cloud Bus.

Custom Event

To publish events, use the ApplicationEventPublisher to broadcast changes. For example, to refresh configuration:

```

import org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Service;

@Service
public class RefreshService {

    private final ApplicationEventPublisher publisher;

    public RefreshService(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public void triggerRefresh() {
        publisher.publishEvent(new RefreshRemoteApplicationEvent(this, "my-app", "default"));
    }
}

```

Spring Cloud Bus Client

Add Dependencies

For RabbitMQ:

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

```

For Kafka:

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>

```

```
</dependency>
```

Configuration

Configure the connection to the message broker to match the server's configuration.

For RabbitMQ:

```
spring:
  cloud:
    bus:
      enabled: true
    rabbitmq:
      host: localhost
      port: 5672
      username: guest
      password: guest
```

For Kafka:

```
spring:
  cloud:
    bus:
      enabled: true
    kafka:
      bootstrap-servers: localhost:9092
      properties:
        key:
          deserializer: org.apache.kafka.common.serialization.StringDeserializer
        value:
          deserializer: org.apache.kafka.common.serialization.StringDeserializer
```

Listening for Events

Implement listeners to handle events received from the bus.

```
import org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.stereotype.Component;

@Component
public class EventListener implements ApplicationListener<RefreshRemoteApplicationEvent> {

    @Override
    public void onApplicationEvent(RefreshRemoteApplicationEvent event) {
        // Handle the refresh event
        System.out.println("Received refresh event: " + event.getMessage());
    }
}
```

You can also listen to events published on the bus by using the `@EventListener` annotation:

```
import org.springframework.context.event.EventListener;
import org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent;
import org.springframework.stereotype.Component;

@Component
public class BusEventListener {

    @EventListener
    public void handleRefreshEvent(RefreshRemoteApplicationEvent event) {
        // Handle the refresh event here
        System.out.println("Received refresh event: " + event);
    }
}
```

Add Dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Enable Config Server

In the main application class, annotate it with `@EnableConfigServer`.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

Configure Application Properties

```
server:
  port: 8888
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-repo/config-repo
```

Add Dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Configure Application Properties

Old Approach with `bootstrap.yml`

```
spring:
  application:
    name: my-service
```

```

cloud:
  config:
    uri: http://localhost:8888
New Approach with spring.config.import
spring:
  config:
    import: optional:configserver:http://myconfigserver.com
      optional: This prefix indicates that the import is optional. If the specified configuration source is not available, the application will continue to start without it, and no error will be thrown.
      configserver: This refers to the type of external configuration source. In this case, it's a Config Server.
      http://myconfigserver.com: This is the URL of the Config Server from which the configurations are to be imported.

```

The `spring.config.import` parameter is used in Spring Boot to import configuration from external sources. This feature was introduced in [Spring Boot 2.4](#) and provides a more flexible way to manage configuration by allowing external configuration sources to be included.

Use Config Values in Your Application

You can now use the properties defined in your Config Server repository in your Spring Boot application.

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyController {

  @Value("${my.custom.property}")
  private String customProperty;

  @GetMapping("/property")
  public String getProperty() {
    return customProperty;
  }
}

```

Refresh Properties

Bean Scope

The `@RefreshScope` annotation works at the bean level. Therefore, the entire bean will be reloaded when a refresh is triggered, not just the individual fields annotated with `@Value`.

State Loss

Because the entire bean is re-instantiated, any state that is not tied to properties (e.g., data stored in non-`@Value` fields) will be lost during the refresh.

It's essential to design your beans accordingly if they hold important state.

Performance

Frequent refreshing of beans, especially complex ones, could have performance implications.

Use `@RefreshScope` judiciously, especially in high-load applications.

RefreshScope

Use `@RefreshScope` on any bean that should be refreshed when the configuration properties change.

```

@RefreshScope
@ConfigurationProperties(prefix = "my.config")
public class MyConfigProperties {
  private String someProperty;
  // getters and setters
}

```

```

@Component
@RefreshScope
public class MyConfigBean {

    @Value("${my.config.someProperty}")
    private String someProperty;

    public String getSomeProperty() {
        return someProperty;
    }
}

```

Trigger a Refresh

Use the `/actuator/refresh` endpoint to refresh the properties dynamically

The refresh action causes the beans annotated with `@RefreshScope` to be re-instantiated, and properties injected via `@Value` or `@ConfigurationProperties` are reloaded with the latest values from the property sources (like configuration files, environment variables, or the Spring Cloud Config Server).

```
curl -X POST http://localhost:8080/actuator/refresh
```

Spring Cloud Gateway

Core

Limit Access

Custom Filters

If you need more flexibility, you can create custom filters to limit access based on custom logic.

```

import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;

@Component
public class CustomRateLimiterGatewayFilterFactory extends
AbstractGatewayFilterFactory<CustomRateLimiterGatewayFilterFactory.Config> {

    public CustomRateLimiterGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        return (exchange, chain) -> {
            // Custom rate limiting logic
            if (isRateLimitExceeded(exchange)) {
                exchange.getResponse().setStatus(HttpStatus.TOO_MANY_REQUESTS);
                return exchange.getResponse().setComplete();
            }
            return chain.filter(exchange);
        };
    }

    private boolean isRateLimitExceeded(ServerWebExchange exchange) {
        // Implement your custom rate limiting logic here
        // Return true if rate limit is exceeded, otherwise false
        return false;
    }

    public static class Config {
        // Configuration properties for the filter
    }
}

```

Using Key Resolvers

You can also use key resolvers to specify the keys for rate limiting. For example, you can limit based on the user ID or IP address.

```
import org.springframework.cloud.gateway.filter.ratelimit.KeyResolver;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;

@Component
public class UserKeyResolver implements KeyResolver {

    @Override
    public Mono<String> resolve(ServerWebExchange exchange) {
        return Mono.just(exchange.getRequest().getRemoteAddress().getAddress().getHostAddress());
    }
}
```

Configuration

Add Dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<!-- Spring Data Redis (Optional for limiting access)-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

Configure Application Properties

```
server:
  port: 8080

spring:
  application:
    name: gateway-service

cloud:
  gateway:
    routes:
      - id: user-service
        uri: lb://USER-SERVICE
        predicates:
          - Path=/user/**
        filters:
          - RewritePath=/user/(?<segment>.*), /${segment}
          - CustomFilter           # Custom filter
          - name: CustomRateLimiter # Custom Limiter (Optional)
          - name: RequestRateLimiter # (Optional)
```

```

args:
  key-resolver: "#{@userKeyResolver}" # Key Resolver
  redis-rate-limiter:
    replenishRate: 10 # The number of requests per second allowed.
    burstCapacity: 20 # The maximum number of requests allowed in a burst.
- id: order-service
  uri: lb://ORDER-SERVICE
    This URI indicates that the gateway should use load balancing to find an available instance of the service.
predicates:
  - Path=/order/** 
filters:
  - RewritePath=/order/(?<segment>.*), /${segment}
  - name: CircuitBreaker
    args:
      name: myCircuitBreaker
      fallbackUri: forward:/fallback
        Provides a fallback route if the primary service instances are unavailable.

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/

```

Enable Discovery Client and Gateway

In your main Spring Boot application class, enable the Discovery Client and Spring Cloud Gateway by using the appropriate annotations.

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class GatewayServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayServiceApplication.class, args);
    }
}

```

Configure Services

Ensure that your backend services (like `USER-SERVICE` and `ORDER-SERVICE`) are registered with Eureka Server, so that Spring Cloud Gateway can route requests to them.

`application.yml` for Backend Services:

```

server:
  port: 8081

spring:
  application:
    name: user-service

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/

```

Run the Application

Start your Eureka Server, backend services, and Spring Cloud Gateway application.

You should now be able to route requests through the gateway to your backend services.

Example Request

Assuming your gateway service is running on `localhost:8080`:

Access the user service: <http://localhost:8080/user/some-endpoint>

Access the order service: <http://localhost:8080/order/some-endpoint>

Temporal

Core

Components

Worker

Temporal workers are responsible for **polling tasks** and **executing workflows/activities**.

The worker **continuously listens on a queue for tasks**.

Once started, they continuously poll the Temporal server for tasks until stopped

You don't need to stop and restart the worker for each task. It keeps polling in the background.

Key Features:

- Can scale horizontally by adding more worker processes.

- Binds to specific task queues to process workflows or activities.

Workflow

A single instance of a workflow, identified by a workflow ID and run ID.

Key Features:

- Maintains its unique state and lifecycle.

- Can be restarted, queried, or signaled.

WorkflowMethod

A **WorkflowMethod** is the entry point for a Temporal workflow. It represents the main logic of the workflow, orchestrating activities and handling state transitions.

Deterministic:

- Must execute deterministically, as the workflow execution is replayed based on its event history.

Long-Running:

- Supports processes that can span seconds, hours, days, or even years.

Stateful:

- Maintains state across failures and restarts.

No Side Effects:

- Cannot perform external I/O or other non-deterministic operations directly. All such operations must be delegated to activities.

Activity

Activities are the units of work executed by workers. They perform side-effecting operations like database writes, HTTP calls, or other I/O-bound tasks.

Key Features:

- Stateless by nature but can maintain local state.

- Supports retries and timeouts.

- Executed in the language runtime and environment of the worker.

Workflow Stub

A proxy object that represents a workflow instance.

You can call methods on this stub, and Temporal handles execution.

Child Workflows Stub

Executed within the context of a parent workflow.

Allow decomposition of complex workflows into smaller, reusable units.

WorkflowOptions

Specifies configurations like taskQueue, workflowId, searchAttributes, and more.

Search Attributes

Allows attaching key-value pairs to workflows for searching and filtering in Temporal's UI or CLI.

Task Queue

A logical routing mechanism for workflows and activities. Workers listen on task queues to process tasks.

Key Features:

- Ensures that specific workflows or activities are routed to appropriate workers.

- Decouples workflows from worker implementations.

Temporal Server

The core of the Temporal architecture, it manages workflow and activity executions, state persistence, and task distribution.

Key Features:

- Handles state persistence in a durable backend (e.g., MySQL, PostgreSQL, Cassandra).

- Manages retries, heartbeats, and failure recovery.

- Provides APIs for starting workflows, querying states, and sending signals.

Signal

A mechanism to asynchronously send external inputs to a running workflow.

Key Features:

- Allows real-time interaction with workflows.

- Used to modify the behavior of a workflow or provide additional data.

Workflow Events

Workflow Start Event

This event indicates the start of the workflow execution. It is triggered when a new workflow is initiated, either manually or through an automatic trigger.

Activity Task Started Event

This event is generated when an activity task is dispatched to a worker and the activity starts executing.

Activity Task Completed Event

This event occurs when an activity completes successfully. It contains the result or output of the activity execution.

Activity Task Failed Event

This event is triggered if an activity fails due to an error, such as an exception being thrown in the activity code.

Activity Task Timed Out Event

This event is generated when an activity exceeds the configured timeout, and it is considered to have timed out.

Timer Started Event

A timer started event occurs when a timer is set within the workflow, which will trigger after a specific duration or when a specific event occurs.

Timer Fired Event

This event is generated when the timer set by the workflow has expired and the associated logic is triggered.

Signal Received Event

This event occurs when a workflow receives a signal, which could be used to notify the workflow about some external event, such as a user action or another system event.

Query Received Event

This event happens when a query is received by the workflow. Queries allow external callers to retrieve the current state or data of the workflow.

Workflow Completed Event

This event is triggered when the workflow finishes successfully, either by returning a result or completing all tasks and

transitions.

Workflow Failed Event

This event occurs when the workflow fails due to an unhandled error or an internal failure within the workflow logic.

Workflow Canceled Event

This event is generated when the workflow is explicitly canceled by an external entity or another workflow.

Workflow Terminated Event

This event occurs when a workflow is forcefully terminated, often by the Temporal system itself due to an unhandled issue or for cleanup.

Workflow Continued As New Event

This event occurs when a workflow continues with a new execution, essentially restarting with a new execution ID, often after a long-running workflow has passed a certain checkpoint or milestone.

TemporalBootstrap

```
package com.sap.sbnc.framework.temporal.common;
public class TemporalBootstrap

public static <T> T createWorkflowStub(String workflowId, String taskQueue, WorkflowClient workflowClient,
TemporalContext temporalContext, SearchAttributes searchAttributes, Class<T> workflowClass) {
    workflowId:
        • A unique identifier for the workflow instance.
        • This is used to associate a stub with an existing workflow or a new one.
    taskQueue:
        • The name of the task queue to which the workflow tasks are assigned.
        • Temporal workers polling this queue will execute the workflow's tasks.
    workflowClient:
        • An instance of WorkflowClient, used to communicate with the Temporal service.
        • This client is required for creating workflow stubs.
    temporalContext:
        • Likely a custom or user-defined object encapsulating Temporal-related configurations or context.
        • Its exact role would depend on its implementation, but it might include settings like timeouts or logging
            configurations.
    searchAttributes:
        • An instance of SearchAttributes, which is used to attach metadata to the workflow.
        • This metadata can be indexed and queried later using Temporal's advanced search capabilities.
    workflowClass:
        • A Class<T> object representing the workflow interface (or class) to be stubbed.
        • Temporal requires a strongly-typed reference to the workflow's definition.

public static <T> T createActivityStub(String taskQueue, TemporalContext temporalContext, Class<T> activityClass) {
```

Create stubs that allow workflows to invoke Temporal activities.

Activities are the components of a Temporal application that handle external operations or business logic.

This method binds the activity invocation to a specific task queue and integrates it with Temporal's workflow execution environment.

WorkflowClient

```
package io.temporal.client;
public interface WorkflowClient
```

```
static <A1, A2, A3> WorkflowExecution start(Functions.Proc3<A1, A2, A3> workflow, A1 arg1, A2 arg2, A3 arg3) {  
    Starts a Temporal workflow asynchronously with the provided method and arguments, returning a WorkflowExecution  
    object to track the workflow's execution.  
<A1, A2, A3>: Specifies the types of the three arguments that the workflow method accepts.
```

Workflow

```
package io.temporal.workflow;  
public final class Workflow
```

```
public static <T> T newChildWorkflowStub(Class<T> workflowInterface, ChildWorkflowOptions options)  
Create a stub for a child workflow. Child workflows are workflows started from within another workflow, enabling  
hierarchical workflow structures.  
ChildWorkflowOptions:

- Provides fine-grained control over the execution of the child workflow.

```

```
public static <T> T newContinueAsNewStub(Class<T> workflowInterface)  
Create a special stub for invoking the continue-as-new functionality within a workflow.  
This feature allows a workflow to terminate its current execution and start a new execution of the same or a different  
workflow, effectively "restarting" itself with a fresh state.
```

```
public static void await(Supplier<Boolean> unblockCondition)  
Pause a workflow's execution until a specified condition becomes true.
```

```
public static <R> R newExternalWorkflowStub(Class<? extends R> workflowInterface, String workflowId)  
Create a stub for an external workflow, enabling workflows to invoke other workflows as part of their execution.  
This method allows you to communicate with other running workflows without directly modifying their state.
```

Configuration

Reference: ChatGPT

Add Temporal Dependencies

Maven:

```
<dependency>  
    <groupId>io.temporal</groupId>  
    <artifactId>temporal-sdk</artifactId>  
    <version>1.22.0</version> <!-- Use the latest version -->  
</dependency>
```

Gradle:

```
implementation 'io.temporal:temporal-sdk:1.22.0' // Use the latest version
```

Setup Temporal

Temporal workers are responsible for **polling tasks** and **executing workflows/activities**. Create a Spring-managed worker:

TemporalWorkerConfig

```
import io.temporal.worker.WorkerFactory;  
import io.temporal.worker.Worker;
```

```

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class TemporalWorkerConfig {

    @Bean
    public WorkflowServiceStubs workflowServiceStubs() {
        return WorkflowServiceStubs.newInstance(
            WorkflowServiceStubsOptions.newBuilder()
                .setTarget("127.0.0.1:7233") // Replace with your Temporal server address
                .build()
        ); // Connect to the Temporal server
    }

    @Bean
    public WorkflowClient workflowClient(WorkflowServiceStubs serviceStubs) {
        return WorkflowClient.newInstance(serviceStubs);
    }

    @Bean
    public WorkerFactory workerFactory(WorkflowClient workflowClient) {
        return WorkerFactory.newInstance(workflowClient);
    }

    @Bean
    public Worker workflowWorker(WorkerFactory workerFactory) {
        Worker worker = workerFactory.newWorker("MyTaskQueue"); // Specify the task queue
        worker.registerWorkflowImplementationTypes(MyWorkflowImpl.class); // Register workflows
        worker.registerActivitiesImplementations(new MyActivityImpl()); // Register activities
        workerFactory.start();
        return worker;
    }
}

```

Define Workflow and Activities

Temporal workflows are interfaces annotated with `@WorkflowInterface`, and activities are interfaces annotated with `@ActivityInterface`.

MyWorkflow

Workflow Interface

```

import io.temporal.workflow.WorkflowInterface;
import io.temporal.workflow.WorkflowMethod;

@WorkflowInterface
public interface MyWorkflow {
    @WorkflowMethod
    String processTask(String input);
}

```

MyWorkflowImpl

Workflow Implementation

```

import io.temporal.workflow.Workflow;

public class MyWorkflowImpl implements MyWorkflow {

    private final MyActivity activities = Workflow.newActivityStub(
        MyActivity.class,
        ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofMinutes(1))
            .build()
    )
}

```

```
);

@Override
public String processTask(String input) {
    return activities.performTask(input); // Call the activity
}
}
```

MyActivity Activity Interface

```
import io.temporal.activity.ActivityInterface;
import io.temporal.activity.ActivityMethod;

@ActivityInterface
public interface MyActivity {
    @ActivityMethod
    String performTask(String input);
}

MyActivityImpl
Activity Implementation
public class MyActivityImpl implements MyActivity {
    @Override
    public String performTask(String input) {
        return "Processed: " + input;
    }
}
```

Trigger Workflow Execution

Use the WorkflowClient to start workflows in your Spring Boot service.

WorkflowService

```
import io.temporal.client.WorkflowClient;
import io.temporal.client.WorkflowOptions;
import org.springframework.stereotype.Service;

@Service
public class WorkflowService {

    private final WorkflowClient workflowClient;

    public WorkflowService(WorkflowClient workflowClient) {
        this.workflowClient = workflowClient;
    }

    public String startWorkflow(String input) {
        MyWorkflow workflow = workflowClient.newWorkflowStub(
            MyWorkflow.class,
            WorkflowOptions.newBuilder()
                .setTaskQueue("MyTaskQueue") // Must match the task queue from the worker
                .build()
        );
        return workflow.processTask(input); // Start the workflow
    }
}
```

Temporal Server Setup

To use Temporal, you need to run a Temporal server. Use Docker to set up a local instance:

```
docker run -d --name temporal \
-p 7233:7233 \
temporalio/auto-setup:latest
```

Example Workflow Trigger in Controller

Create a Spring Boot controller to trigger the workflow:

WorkflowController

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class WorkflowController {

    private final WorkflowService workflowService;

    public WorkflowController(WorkflowService workflowService) {
        this.workflowService = workflowService;
    }

    @GetMapping("/start-workflow")
    public String startWorkflow(@RequestParam String input) {
        return workflowService.startWorkflow(input);
    }
}
```

Temporal Docker Compose File

```
temporal-ui:
  container_name: temporal-ui
  depends_on:
    - temporal
  environment:
    - TEMPORAL_ADDRESS=temporal:7233
    - TEMPORAL_CORS_ORIGINS=http://localhost:3000
  image: temporalio/ui:${TEMPORAL_UI_VERSION}
  networks:
    - temporal-network
  ports:
    - 8080:8080
```

7233: Temporal Server Communication Port

This is the port Temporal UI uses to communicate with the Temporal server.

Do not modify this unless you've specifically configured the Temporal server to use a different port. By default, the Temporal server listens on port 7233.

3000: CORS Origin

The TEMPORAL_CORS_ORIGINS environment variable specifies the origins allowed for cross-origin resource sharing (CORS).

In this case, it's set to http://localhost:3000, which is typically used if you have another frontend (e.g., a React app) interacting with the Temporal server.

8080: Temporal UI Port

This is the port Temporal UI listens on within the container and maps to the host port.

Modify the first number (8080) **if you want to expose the UI on a different host port.**

[temporal-serviceclient]

WorkflowServiceStubs

```
package io.temporal.serviceclient;
public interface WorkflowServiceStubs
    extends ServiceStubs<WorkflowServiceGrpc.WorkflowServiceBlockingStub,
    WorkflowServiceGrpc.WorkflowServiceFutureStub>
```

```
static WorkflowServiceStubs newLocalServiceStubs()  
static WorkflowServiceStubs newServiceStubs(WorkflowServiceStubsOptions options)  
[temporal-sdk]
```

WorkerFactory

```
package io.temporal.worker;  
public final class WorkerFactory  
public synchronized void start()  
    Start all registered workers so they can begin polling the Temporal service for workflow and activity tasks.  
public Worker newWorker(String taskQueue)
```

Binding to Task Queue

When you call this method with a specific taskQueue name, it creates a worker that listens for tasks assigned to that queue.

Task Execution

The worker is then capable of executing tasks that are pushed onto the task queue.

This allows the worker to process those tasks as they become available.

```
[temporal-testing]
```

TestWorkflowEnvironment

```
package io.temporal.testing;  
public interface TestWorkflowEnvironment extends Closeable  
void registerDelayedCallback(Duration delay, Runnable r);  
Register a callback (Runnable) that will be executed after a specified delay (Duration).  
This is useful for simulating delayed actions or events in your workflow tests.
```

Immediate Execution

Zuul

Core

Zuul is an edge service that provides dynamic routing, monitoring, resiliency, security, and more.

It's an open-source project maintained by [Netflix](#) and is part of the [the Netflix OSS suite](#).

Zuul acts as a front door for all requests from devices and web sites to the backend of a web application.

Key Features of Zuul

Routing

Zuul dynamically routes requests to different backend services.

This makes it easy to hide the complexity of your microservices architecture and provide a single entry point for clients.

Load Balancing

Zuul can balance requests across multiple instances of a service, ensuring that no single instance is overwhelmed.

Security

Zuul can handle authentication and authorization, protecting backend services from unauthorized access.

Resiliency

Zuul can implement fallback logic, rate limiting, and retries to ensure high availability and fault tolerance.

Monitoring

Zuul can provide detailed metrics on the requests it handles, including response times, success rates, and more.

How Zuul Works

Zuul operates as a gateway or edge service. It sits between clients and your backend services.

When a client sends a request, Zuul intercepts it, applies various filters, and then routes it to the appropriate backend service.

The response from the backend service is then passed back through Zuul to the client.

Zuul Filters

Zuul uses filters to perform various tasks. Filters are small units of work that can modify the incoming request, route it, and modify the outgoing response. Zuul has four types of filters:

Pre Filters

Execute [before the request is routed to the backend service](#). They can be used for tasks like authentication and logging.

Post Filters

Execute [after the request has been routed to the backend service](#). They can be used to modify the response before it is sent back to the client.

Routing Filters

Handle the [routing](#) of the request to the backend service. They can modify the request path or parameters.

Error Filters

Execute [when an error occurs](#) during the handling of the request. They can be used to handle and log errors.

Rate Limiting in Zuul

We can implement rate limiting in Zuul by using custom filters or integrating with third-party libraries like Bucket4j for more advanced rate limiting capabilities.

Below are the two main approaches:

Using Custom Filters

You can create a custom Zuul filter to implement rate limiting. Here is how you can do it:

```
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletRequest;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

@Component
public class RateLimitFilter extends ZuulFilter {

    private final ConcurrentHashMap<String, AtomicInteger> rateLimitingMap = new ConcurrentHashMap<>();
    private static final long TIME_FRAME = TimeUnit.MINUTES.toMillis(1); // 1 minute
    private static final int LIMIT = 10; // 10 requests per minute

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }
}
```

```

@Override
public Object run() throws ZuulException {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();

    String clientIp = request.getRemoteAddr();
    AtomicInteger requestCount = rateLimitingMap.computeIfAbsent(clientIp, k -> new AtomicInteger(0));

    long currentTime = System.currentTimeMillis();
    if (requestCount.get() == 0 || currentTime - requestCount.get() > TIME_FRAME) {
        rateLimitingMap.put(clientIp, new AtomicInteger(1));
    } else {
        if (requestCount.incrementAndGet() > LIMIT) {
            ctx.setResponseStatusCode(429);
            ctx.setSendZuulResponse(false);
            ctx.setResponseBody("Rate limit exceeded");
            return null;
        }
    }
    return null;
}

```

Integrating with Bucket4j

Bucket4j is a Java rate-limiting library that can be integrated with Zuul for more sophisticated rate limiting.

Dependencies

```

<dependency>
    <groupId>com.github.vladimir-bukhtoyarov</groupId>
    <artifactId>bucket4j-core</artifactId>
    <version>6.2.0</version>
</dependency>

```

Custom Filter with Bucket4j

```

import com.github.bucket4j.Bandwidth;
import com.github.bucket4j.Bucket;
import com.github.bucket4j.Bucket4j;
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletRequest;
import java.time.Duration;

@Component
public class Bucket4jRateLimitFilter extends ZuulFilter {

    private final Bucket bucket;

    public Bucket4jRateLimitFilter() {
        Bandwidth limit = Bandwidth.simple(10, Duration.ofMinutes(1));
        this.bucket = Bucket4j.builder().addLimit(limit).build();
    }

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 1;
    }
}

```

```

@Override
public boolean shouldFilter() {
    return true;
}

@Override
public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();

    if (bucket.tryConsume(1)) {
        // Allowed request
        return null;
    } else {
        // Rate limit exceeded
        ctx.setResponseStatusCode(429);
        ctx.setSendZuulResponse(false);
        ctx.setResponseBody("Rate limit exceeded");
        return null;
    }
}
}

```

Configuration

Add Zuul Dependency

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<!-- Spring Data Redis Dependency (Optional for limiting access) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>

```

Enable Zuul Proxy

Annotate your main application class with `@EnableZuulProxy` to enable Zuul proxy functionality.

```

package com.example.zuulgateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
@EnableZuulProxy
public class ZuulGatewayApplication {

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

```

Configure Application Properties

```

server:
  port: 8080

```

```

eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
  instance:
    hostname: localhost
  service-url:
    defaultZone: http://localhost:8761/eureka/

zuul:
  routes:
    service1:
      path: /service1/**
      serviceId: service1
    service2:
      path: /service2/**
      serviceId: service2

```

This configuration routes requests to `/service1/**` to the service registered with Eureka as `service1` and `/service2/**` to the service registered as `service2`.

Register Services with Eureka

Ensure that your backend services are also Spring Boot applications and that they are registered with Eureka. You need to add the Eureka client dependency and configure Eureka in those applications similarly.

Run Zuul Gateway and Services

Start your Eureka server, Zuul gateway application, and backend services. Zuul will automatically route the requests based on your configuration.

Elasticsearch

Configuration

Add Dependencies

First, you need to add the necessary dependencies to your pom.xml file if you're using Maven. Include the Spring Data Elasticsearch starter and Elasticsearch client dependencies:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-high-level-client</artifactId>
  <version>7.17.0</version>
</dependency>
<!-- Substitute of the elasticsearch-rest-high-level-client -->
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-client</artifactId>
  <version>8.10.1</version> <!-- Use the latest version -->
</dependency>

```

Configure Elasticsearch

```

spring:
  elasticsearch:
    uris: http://localhost:9200
    username: elastic

```

```
password: your_password
```

Define an Entity

Create a Java class to represent the data you want to store in Elasticsearch. Annotate it with @Document to map it to an Elasticsearch index:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.Document;

@Document(indexName = "products")
public class Product {

    @Id
    private String id;
    private String name;
    private String description;
    private Double price;

    // Getters and setters
}
```

Create a Repository Interface

Define a repository interface that extends ElasticsearchRepository to perform CRUD operations on your data:

```
import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProductRepository extends ElasticsearchRepository<Product, String> {
    List<Product> findByName(String name);
}
```

Use the Repository in a Service

You can now inject the repository into a service class and use it to interact with Elasticsearch:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public void saveProduct(Product product) {
        productRepository.save(product);
    }

    public List<Product> findProductsByName(String name) {
        return productRepository.findByName(name);
    }

    public Iterable<Product> findAllProducts() {
        return productRepository.findAll();
    }

    public void deleteProduct(String id) {
        productRepository.deleteById(id);
    }
}
```

[elasticsearch]

SortBuilders

```
package org.elasticsearch.search.sort;
```

```

public class SortBuilders
public static GeoDistanceSortBuilder geoDistanceSort(String fieldName, double lat, double lon)
    The method is used to sort search results by geographical distance from a specific point.
    This is commonly used in scenarios like finding the nearest places or users.
public static GeoDistanceSortBuilder geoDistanceSort(String fieldName, GeoPoint... points)
public static GeoDistanceSortBuilder geoDistanceSort(String fieldName, String... geohashes)
public static ScoreSortBuilder scoreSort()
public static FieldSortBuilder fieldSort(String field)
public static FieldSortBuilder pitTiebreaker()
public static ScriptSortBuilder scriptSort(Script script, ScriptSortBuilder.ScriptSortType type)

```

[elasticsearch-high-level-rest-client]

RestHighLevelClient

```

package org.elasticsearch.client;
public class RestHighLevelClient implements Closeable
public final SearchResponse search(SearchRequest searchRequest, RequestOptions options) throws IOException
    SearchResponse searchResponse = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);

```

[elasticsearch-rest-client]

RestClient

```

package org.elasticsearch.client;
public class RestClient implements Closeable

```

Amazon S3

Configuration

Add Dependencies

```

<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-s3</artifactId>
    <version>1.13.1</version>
</dependency>

```

Configure AWS Credentials:

Environment Variables

Set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables.

Java System Properties

Set the `aws.accessKeyId` and `aws.secretAccessKey` system properties.

Credential Profile:

Create a credential profile in your `.aws/credentials` file.

Create an Amazon S3 Client

```

@Configuration
public class AmazonS3Config {

```

```

@Value(value = "classpath:application-${spring.profiles.active:default}.properties")
private Resource propertySource;

@Autowired
private Environment environment;

@Value("${s3.accessKeyId}")
private String awsKeyId;

@Value("${s3.secretAccessKey}")
private String accessKey;

@Value("${s3.endPoint}")
private String endPoint;

@Value("${s3.region}")
private String region;

@Value(value = "${http.client.ssl.key-store}")
private String serverSSLKeyStore;

public void setTrustStore() throws KeyStoreException, NoSuchAlgorithmException, CertificateException,
IOException, UnrecoverableKeyException, KeyManagementException {
    KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
    InputStream is = new FileInputStream(serverSSLKeyStore);
    String storePassword = CashManagementUtil.getPassword(propertySource.getInputStream(),
            "http.client.ssl.key-store-password", environment.getProperty("KEYSTORE_ENCRYPT_KEY"));
    ks.load(is, storePassword.toCharArray());

    KeyManagerFactory kmf = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    char[] kp = storePassword.toCharArray();
    kmf.init(ks, kp);

    SSLContext sslContext = SSLContext.getInstance("TLS");

    TrustManagerFactory tmf = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
    tmf.init(ks);

    sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

    SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();
    HttpsURLConnection.setDefaultSSLSocketFactory(sslSocketFactory);
    SSLContext.setDefault(sslContext);
}

@Bean
public AmazonS3 awsS3Client() throws UnrecoverableKeyException, KeyManagementException,
KeyStoreException, NoSuchAlgorithmException, CertificateException, IOException {
    //A. credential configuration
    AWS Credentials awsCreds = new BasicAWSCredentials(awsKeyId, accessKey);
    //A. endpoint configuration
    EndpointConfiguration ec = new EndpointConfiguration(endPoint, region);
    //A. client configuration
    ClientConfiguration cc = new ClientConfiguration();
    cc.setSignerOverride("AWSS3V4SignerType");
    return AmazonS3ClientBuilder.standard().withClientConfiguration(cc).withEndpointConfiguration(ec)
        .withCredentials(new AWSStaticCredentialsProvider(awsCreds)).build();
}

AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

```

Perform S3 Operations

List buckets

```
ListBucketsRequest listBucketsRequest = new ListBucketsRequest();
ListBucketsResult listBucketsResult = s3Client.listBuckets(listBucketsRequest);
for (Bucket bucket : listBucketsResult.getBuckets()) {
    System.out.println("Bucket Name: " + bucket.getName());
}
```

Create a bucket

```
CreateBucketRequest createBucketRequest = new CreateBucketRequest("my-bucket-name");
s3Client.createBucket(createBucketRequest);
```

Upload an object

```
PutObjectRequest putObjectRequest = new PutObjectRequest("my-bucket-name", "my-file.txt", new File("my-
file.txt"));
s3Client.putObject(putObjectRequest);
```

Download an object

```
GetObjectRequest getObjectRequest = new GetObjectRequest("my-bucket-name", "my-file.txt");
s3Client.getObject(getObjectRequest, new File("downloaded-file.txt"));
```

Delete an object:

```
DeleteObjectRequest deleteObjectRequest = new DeleteObjectRequest("my-bucket-name", "my-file.txt");
s3Client.deleteObject(deleteObjectRequest);
```

Amazon
awssdk

AWS SDK

AWS SDK v1: The group ID is `com.amazonaws`.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk</artifactId>
    <version>1.x.x</version>
</dependency>
```

AWS SDK v2: The group ID is `software.amazon.awssdk`.

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>dynamodb</artifactId> <!-- Or other AWS service SDKs -->
    <version>2.x.x</version>
</dependency>
```

[netty-nio-client]

NettyNioAsyncHttpClient

```
package software.amazon.awssdk.http.nio.netty;
public final class NettyNioAsyncHttpClient implements SdkAsyncHttpClient
    software.amazon.awssdk.http.async.SdkAsyncHttpClient
```

```
public static Builder builder()
```

Create a NettyNioAsyncHttpClient Builder.

```
    SdkAsyncHttpClient httpClient = NettyNioAsyncHttpClient.builder()
        .maxConcurrency(dynamodbMaxConcurrency)
        .connectionAcquisitionTimeout(Duration.ofMillis(dynamodbConnectionTimeout))
        .build();
```

```
public static SdkAsyncHttpClient create()
```

[http-client-spi]

SdkAsyncHttpClient

```
package software.amazon.awssdk.http.async;  
@Immutable  
@ThreadSafe  
@SdkPublicApi  
public interface SdkAsyncHttpClient extends SdkAutoCloseable  
  
    Makes asynchronous HTTP requests to AWS services.  
    It is part of the SDK's infrastructure that allows developers to interact with AWS services without blocking the execution  
    of their applications,  
    enabling more efficient handling of I/O operations.  
    software.amazon.awssdk.http.nio.netty.NettyNioAsyncHttpClient
```

[aws-java-sdk-core]

NotThreadSafe

```
package com.amazonaws.annotation;  
@Documented  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.CLASS)  
public @interface NotThreadSafe  
  
    Documenting annotation to indicate a class is not thread-safe and should not be used in a multi-threaded context.
```

DefaultAWSCredentialsProviderChain

```
package com.amazonaws.auth;  
public class DefaultAWSCredentialsProviderChain extends AWSCredentialsProviderChain
```

AWS DynamoDB

Core

Amazon DynamoDB is a fully managed NoSQL database service provided by AWS, designed for high availability, scalability, and performance.

It is suitable for applications that require consistent, single-digit millisecond response times at any scale.

Key Components

Tables

The primary structure for storing data. Each table consists of **items** (records) and **attributes** (fields).

Items

Individual records in a table, similar to rows in relational databases. Each item is uniquely identified by a primary key.

Attributes

The data fields within an item. Attributes can be of various types, including strings, numbers, and binary data.

DynamoDB attribute names are **case-sensitive**.

This means that when you define your entity class using annotations like @DynamoDbAttribute,

the attribute name provided in the annotation must exactly match the attribute name in your DynamoDB table.

String

Represents textual data. (e.g., Name = "Alice")

A string is a sequence of Unicode characters.

Strings are used in partition keys, sort keys, and any other attribute you define.

Strings can be up to 400KB in size and must be UTF-8 encoded.

Number

Represents numeric data. (e.g., Age = 30)

Numbers can be integers or floating-point values.

DynamoDB uses a high-precision arithmetic library to support up to 38 digits of precision.

Binary

Represents raw binary data (like images, documents, or other files).

The maximum size of a binary attribute is 400KB.

Primary Key

Uniquely identifies each item in a table. There are two types:

Partition Key (also known as the HASH key)

A single attribute that determines the partition for the item.

Sort Key (also known as the RANGE key)

An additional key attribute used to sort items with the same partition key.

Composite Key

Together, the partition key and sort key uniquely identify an item in a table, forming what is called a composite primary key.

This means two items can have the same partition key, but their sort keys must differ to maintain uniqueness.

Indexes

Allow for more flexible querying:

Global Secondary Index (GSI)

Allows querying on non-primary key attributes.

Local Secondary Index (LSI)

Allows querying on non-primary key attributes while using the same partition key.

Streams

Capture changes to items in a table (inserts, updates, deletes) and can be used for triggering actions in other AWS services.

Provisioned and On-Demand Capacity

Provisioned

You can configure the amount of read and write capacity units for your table to handle expected traffic. You need to specify:

ReadCapacityUnits (RCU)

The number of reads per second that your table is provisioned to handle.

WriteCapacityUnits (WCU)

The number of writes per second that your table is provisioned to handle.

On-Demand

Automatically adjusts capacity based on workload, suitable for unpredictable traffic.

DynamoDB Accelerator (DAX)

An in-memory caching service for DynamoDB that reduces response times from milliseconds to microseconds.

Transactions

Support for multi-item, all-or-nothing operations, ensuring atomicity across multiple items.

Data Types

Supports various data types, including scalar types (e.g., String, Number), document types (e.g., List, Map), and sets.

Configuration

Add Dependencies

Add the DynamoDB and AWS SDK dependencies to your pom.xml (for Maven):

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-bom</artifactId>
    <version>1.12.201</version> <!-- Check for the latest version -->
    <scope>import</scope>
    <type>pom</type>
</dependency>
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>bom</artifactId>
    <version>2.17.173</version> <!-- Check for the latest version -->
    <scope>import</scope>
    <type>pom</type>
</dependency>

<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>dynamodb</artifactId>
</dependency>
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>dynamodb-enhanced</artifactId>
</dependency>
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-dynamodb</artifactId>
</dependency>
```

For Gradle, add these dependencies in your build.gradle:

```
implementation 'software.amazon.awssdk:dynamodb:2.20.0'
implementation 'org.springframework.cloud:spring-cloud-starter-aws:2.2.6.RELEASE'
```

Configure AWS Credentials

You need to set up AWS credentials for accessing DynamoDB. You can provide them in different ways:

- Environment Variables
Set `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
- AWS Credentials File
Located in `~/.aws/credentials`:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```
- Spring Boot Application Properties
In your application.properties or application.yml, add:

```
cloud.aws.credentials.accessKey=YOUR_ACCESS_KEY
cloud.aws.credentials.secretKey=YOUR_SECRET_KEY
cloud.aws.region.static=YOUR_REGION
```

Configuration

Create a Spring @Configuration class that initializes the DynamoDB client:

Low level Configuration (v1)

```
@Bean
public DynamoDBMapper dynamoDBMapper(AmazonDynamoDB client) {
    return new DynamoDBMapper(client);
}

private AmazonDynamoDB getLocalDynamoDBClient() {
    DynamoDBConfig config = DynamoDBConfig
        .builder()
        .domain(domain)
        .endpoint(host)
        .build();
    return new DefaultDynamoDBClientFactory(config).lowLevelClient();
}
```

Enhanced Dynamodb Client

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbAsyncTable;
import software.amazon.awssdk.regions.Region;

@Configuration
public class DynamoDBConfig {

    @Bean
    public DynamoDbClient dynamoDbClient() {
        return DynamoDbClient.builder()
            .region(Region.US_EAST_1) // Specify your region
            .build();
    }

    // An altrnative for the DynamoDBClient with better performance.
    @Bean
    public DynamoDbAsyncClient dynamoDbAsyncClient() {
        // Build DynamoDBAsyncClient with region and credentials
        DynamoDbAsyncClientBuilder builder = DynamoDbAsyncClient.builder();

        // Set the region where DynamoDB is deployed
        builder.region(Region.US_EAST_1); // Change to your region

        // Use DefaultCredentialsProvider to automatically pick up AWS credentials
        builder.credentialsProvider(DefaultCredentialsProvider.create());
    }
}
```

```

        // Optionally customize settings like HTTP configuration, retries, etc.
        return builder.build();
    }
    @Bean
    public DynamoDbEnhancedAsyncClient dynamoEnhancedClient(DynamoDbAsyncClient dynamoClient) {
        return DynamoDbEnhancedAsyncClient.builder().dynamoDbClient(dynamoClient).build();
    }
    @Bean
    public DynamoDbAsyncTable<Product> productTable(DynamoDbEnhancedAsyncClient enhancedAsyncClient) {
        return enhancedAsyncClient.table("Product", TableSchema.fromBean(Product.class));
    }
}

```

Create an Entity and Repository for DynamoDB

Define your entity class to map to a DynamoDB table. You can use annotations to define table and attribute names:

```

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName = "YourTableName")
public class YourEntity {

    private String id;
    private String name;

    @DynamoDBHashKey(attributeName = "id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

Create a repository interface for the entity, using Spring Data DynamoDB:

```

import org.socialsignin.spring.data.dynamodb.repository.EnableScan;
import org.springframework.data.repository.CrudRepository;

@EnableScan
public interface YourEntityRepository extends CrudRepository<YourEntity, String> {
}

```

Service Layer for Business Logic

Create a service class to interact with DynamoDB through the repository:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service

```

```

public class YourEntityService {

    @Autowired
    private YourEntityRepository repository;

    public YourEntity save(YourEntity entity) {
        return repository.save(entity);
    }

    public Optional<YourEntity> findById(String id) {
        return repository.findById(id);
    }

    public void delete(String id) {
        repository.deleteById(id);
    }
}

```

[sdk-core]

SdkIterable

```

package software.amazon.awssdk.core.pagination.sync;
@SdkPublicApi
public interface SdkIterable<T> extends Iterable<T>
default Stream<T> stream()

```

[dynamodb]

DynamoDbAsyncClient

```

package software.amazon.awssdk.services.dynamodb;
public interface DynamoDbAsyncClient extends SdkClient

```

DynamoDbAsyncClient is an asynchronous service client that implements the SdkAsyncHttpClient interface in the AWS SDK for Java

The DynamoDBAsyncClient is an asynchronous client for interacting with Amazon DynamoDB.

It allows for non-blocking operations, making it ideal for high-performance and scalable applications that interact with DynamoDB.

This client leverages Java's CompletableFuture API, allowing for asynchronous calls that do not block the calling thread while waiting for a response.

```
static DynamoDbAsyncClientBuilder builder()
```

Create a DynamoDbAsyncClient.

```

DynamoDbAsyncClient dynamoDbAsyncClient = DynamoDbAsyncClient.builder()
    .region(optionalRegion.orElse(Region.CN_NORTHWEST_1))
    .httpClient(httpClient)
    .build();

```

[dynamodb-enhanced]

DynamoDbEnhancedClient [CORE]

```

package software.amazon.awssdk.enhanced.dynamodb;
@SdkPublicApi
@ThreadSafe
public interface DynamoDbEnhancedClient extends DynamoDbEnhancedResource

```

```
default BatchGetResultPagelitable batchGetItem(BatchGetItemEnhancedRequest request)
```

Allows batch retrieval of multiple items from one or more tables in a single request.

```
default BatchWriteResult batchWriteItem(BatchWriteItemEnhancedRequest request)
```

Allows batch write operations (put and delete) for multiple items in a single request.

DynamoDbTable [CORE]

```
package software.amazon.awssdk.enhanced.dynamodb;
```

```
@SdkPublicApi
```

```
@ThreadSafe
```

```
public interface DynamoDbTable<T> extends MappedTableResource<T>
```

```
    software.amazon.awssdk.enhanced.dynamodb.model.Pagelitable
```

```
    software.amazon.awssdk.core.pagination.sync.SdkIterable
```

```
default Pagelitable<T> scan()
```

Scans the table and returns all items.

```
    List<TestPerson> testPersonList= testPersonTable.scan().items().stream().toList();
```

```
default Pagelitable<T> scan(ScanEnhancedRequest request)
```

Performs a scan with additional options such as filters.

```
    ScanEnhancedRequest scanRequest = ScanEnhancedRequest.builder().limit(10).build();
```

```
    PageIterable<TestPerson> scanResults = dynamoDbTable.scan(scanRequest);
```

```
default void putItem(T item)
```

Inserts a new item into the table or replaces an existing item with the same key.

```
    dynamoDbTable.putItem(testPerson);
```

```
default void putItem(PutItemEnhancedRequest<T> request)
```

Allows customization of the putItem operation, such as adding a condition to only insert the item if it doesn't already exist.

```
    dynamoDbTable.putItem(PutItemEnhancedRequest.builder(TestPerson.class).item(testPerson).build());
```

```
default T getItem(Key key)
```

Retrieves a single item from the table by its primary key.

```
    Key key = Key.builder().partitionValue("S001").build();
```

```
    TestPerson person = dynamoDbTable.getItem(key);
```

```
default T getItem(GetItemEnhancedRequest request)
```

Retrieves an item based on a customized request with additional parameters.

```
    GetItemEnhancedRequest<TestPerson> request = GetItemEnhancedRequest.builder().key(key).build();
```

```
    TestPerson person = dynamoDbTable.getItem(request);
```

```
default T updateItem(T item)
```

Updates an item in the table. If the item doesn't exist, it creates a new one.

Fields that are null in the item parameter object will not be updated in the existing item in the DynamoDB table.

```
default T updateItem(UpdateItemEnhancedRequest<T> request)
```

Allows customization of the updateItem operation, such as adding conditions or specifying attributes to update.

```
default T deleteItem(Key key)
```

Deletes an item from the table by its key.

```
default T deleteItem(DeleteItemEnhancedRequest request)
```

Deletes an item with additional options like conditional deletion.

```
    dynamoDbTable.deleteItem(DeleteItemEnhancedRequest.builder(TestPerson.class).key(key).build());
```

```
default Pagelitable<T> query(QueryEnhancedRequest request)
```

Queries the table for items that match the provided key condition expression.

In DynamoDB, a **Query** operation is always focused on retrieving items based on the primary key (which consists of the partition key and optionally the sort key) or secondary index keys (if you've defined secondary indexes).

You cannot use the Query operation to query on arbitrary fields other than the partition and sort key.

```
    QueryEnhancedRequest queryRequest = QueryEnhancedRequest.builder()
```

```

    .queryConditional(QueryConditional.keyEqualTo(Key.builder().partitionValue("S001").build()))
    .build();
    PageIterable<TestPerson> results = dynamoDbTable.query(queryRequest);
default Pagelerable<T> query(Consumer<QueryEnhancedRequest.Builder> requestConsumer)
    public PageIterable<TestPerson> queryTestPersons(DynamoDbTable<TestPerson> testPersonTable, String
testPersonIdPrefix) {
        return testPersonTable.query(request -> request.queryConditional(
            QueryConditional.keyEqualTo(Key.builder().partitionValue(testPersonIdPrefix).build())
            .limit(10) // Limit to 10 results per page
        );
    }
}

```

DynamoDbAsyncTable [CORE]

package software.amazon.awssdk.enhanced.dynamodb;
public interface **DynamoDbAsyncTable**<T> extends MappedTableResource<T>

The DynamoDbAsyncTable is part of the AWS SDK for Java v2, specifically for [the asynchronous interaction](#) with Amazon DynamoDB.

It provides a more idiomatic way to work with DynamoDB tables using [the enhanced client](#), allowing you to interact with DynamoDB asynchronously using non-blocking I/O.

This is particularly useful for handling high-throughput and large-scale operations in a scalable and efficient manner.

Key Concepts

Asynchronous Operations

DynamoDbAsyncTable uses [CompletableFuture](#) to execute non-blocking calls.

This helps in improving throughput by not blocking threads while waiting for a response from DynamoDB.

Enhanced Client

The DynamoDbEnhancedAsyncClient is a [higher-level abstraction](#) over the raw DynamoDB SDK and provides a more user-friendly API for table-based operations.

Type-Safe Operations

You can define your data model as a POJO (Plain Old Java Object), and the enhanced client [will map that model to and from DynamoDB items](#).

```

default CompletableFuture<T> getItem(GetItemEnhancedRequest request)
default CompletableFuture<T> getItem(Consumer<GetItemEnhancedRequest.Builder> requestConsumer)
default CompletableFuture<T> getItem(Key key)
default CompletableFuture<T> getItem(T keyItem)
default CompletableFuture<Void> putItem(T item)
default CompletableFuture<T> updateItem(T item)
default CompletableFuture<T> deleteItem(Key key)
default CompletableFuture<T> deleteItem(T keyItem)

```

Define the Model Class

You need to define a POJO that represents your DynamoDB table structure. This class should be annotated [to map its fields to DynamoDB attributes](#).

```

import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.DynamoDbBean;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.DynamoDbPartitionKey;

@DynamoDbBean
public class Product {

    private String id;
    private String name;
}

```

```

private double price;

@DynamoDbPartitionKey
public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}
}

```

Configure DynamoDbAsyncTable

In your Spring Boot configuration, set up the [DynamoDbAsyncClient](#) and [DynamoDbAsyncTable](#) using the enhanced client.

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbAsyncTable;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;

@Configuration
public class DynamoDbConfig {

    @Bean
    public DynamoDbAsyncClient dynamoDbAsyncClient() {
        return DynamoDbAsyncClient.builder().build();
    }

    @Bean
    public DynamoDbEnhancedAsyncClient dynamoDbEnhancedAsyncClient(DynamoDbAsyncClient dynamoDbAsyncClient) {
        return DynamoDbEnhancedAsyncClient.builder()
            .dynamoDbClient(dynamoDbAsyncClient)
            .build();
    }

    @Bean
    public DynamoDbAsyncTable<Product> productTable(DynamoDbEnhancedAsyncClient enhancedAsyncClient) {
        return enhancedAsyncClient.table("Product", TableSchema.fromBean(Product.class));
    }
}

```

Performing Operations with DynamoDbAsyncTable

Now that you have your `DynamoDbAsyncTable` set up, you can perform various operations like saving, updating, deleting, and querying items.

Inserting an Item (Put)

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbAsyncTable;

```

```

import java.util.concurrent.CompletableFuture;

@Service
public class ProductService {

    private final DynamoDbAsyncTable<Product> productTable;

    @Autowired
    public ProductService(DynamoDbAsyncTable<Product> productTable) {
        this.productTable = productTable;
    }

    public CompletableFuture<Void> addProduct(Product product) {
        return productTable.putItem(product);
    }
}

```

Retrieving an Item (Get)

```

public CompletableFuture<Product> getProduct(String id) {
    return productTable.getItem(r -> r.key(k -> k.partitionValue(id)));
}

```

Updating an Item

```

public CompletableFuture<Void> updateProduct(Product product) {
    return productTable.updateItem(product);
}

```

Deleting an Item

```

public CompletableFuture<Void> deleteProduct(String id) {
    return productTable.deleteItem(r -> r.key(k -> k.partitionValue(id)));
}

```

Using CompletableFuture for Asynchronous Handling

```

public CompletableFuture<Void> addAndFetchProduct(Product product) {
    return addProduct(product).thenCompose(aVoid -> getProduct(product.getId()))
        .thenAccept(fetchedProduct -> {
            System.out.println("Fetched Product: " + fetchedProduct.getName());
        });
}

```

DynamoDbEnhancedAsyncClient

```

package software.amazon.awssdk.enhanced.dynamodb;
public interface DynamoDbEnhancedAsyncClient extends DynamoDbEnhancedResource

```

This enhanced client is used to work with your **POJO** (in this case, **Product**) and interact with **DynamoDB** in a type-safe manner.

Repeated update error

If you need to update multiple fields of the same item, ensure that each item is only updated in its own transaction. This way, you avoid the conflict of multiple operations on the same item.

Expression

```

package software.amazon.awssdk.enhanced.dynamodb;
@SdkPublicApi
@ThreadSafe
public final class Expression

```

It is used to represent a condition or filter expression that can be applied during DynamoDB operations, such as Query, Scan, Update, or PutItem.

This class allows you to define logical conditions based on DynamoDB item attributes, such as comparing attributes or checking if attributes exist.

To create an Expression, you need to provide three main components:

expression

The condition or filter logic (a string).

expressionValues

A map of attribute values that are used in the expression.

expressionNames

A map of attribute name placeholders (if needed, to avoid reserved words).

```
@NotThreadSafe
```

```
public static final class Builder
```

```
    public Builder expression(String expression)
```

```
    public Builder putExpressionName(String key, String value)
```

If your attribute names conflict with DynamoDB's reserved keywords, you can use expressionNames to provide aliases

The # symbol is used to define attribute name placeholders, known as expression attribute names.

The : symbol is used to define attribute value placeholders, known as expression attribute values.

```
    Map<String, String> expressionNames = new HashMap<>();  
    expressionNames.put("#name", "name"); // Avoid conflict with reserved keyword "name"
```

```
    Expression expression = Expression.builder()  
        .expression("#name = :value")  
        .expressionNames(expressionNames)  
        .expressionValues(Map.of(":value", AttributeValue.builder().s("John").build()))  
        .build();
```

```
    public Builder putExpressionValue(String key, AttributeValue value)
```

```
        Expression filterExpression = Expression.builder()  
            .expression("age = :age")  
            .putExpressionValue(":age", AttributeValue.builder().n(String.valueOf(age)).build())  
            .build();
```

```
    public Builder expressionValues(Map<String, AttributeValue> expressionValues)
```

```
        Expression filterExpression = Expression.builder()  
            .expression("age = :age")  
            .expressionValues(expressionValues)  
            .build();
```

Key

```
package software.amazon.awssdk.enhanced.dynamodb;
```

```
public final class Key
```

```
public static Builder builder()
```

Returns a new builder that can be used to construct an instance of this class.

```
public Builder partitionValue(String partitionValue)
```

String value to be used for the partition key. The string will be converted into an AttributeValue of type S.

```
public Builder sortValue(String sortValue)
```

String value to be used for the sort key. The string will be converted into an AttributeValue of type S.

```
package software.amazon.awssdk.enhanced.dynamodb;
```

model

Pagelterable

```
package software.amazon.awssdk.enhanced.dynamodb.model;  
@SdkPublicApi  
@ThreadSafe  
public interface Pagelterable<T> extends SdkIterable<Page<T>>  
default SdkIterable<T> items()
```

QueryConditional

```
package software.amazon.awssdk.enhanced.dynamodb.model;  
@SdkPublicApi  
@ThreadSafe  
public interface QueryConditional
```

TransactUpdateItemEnhancedRequest

```
package software.amazon.awssdk.enhanced.dynamodb.model;  
@SdkPublicApi  
@ThreadSafe  
public class TransactUpdateItemEnhancedRequest<T>
```

TransactWriteItemsEnhancedRequest

```
package software.amazon.awssdk.enhanced.dynamodb.model;  
@SdkPublicApi  
@ThreadSafe  
public final class TransactWriteItemsEnhancedRequest  
public static Builder builder()
```

```
public <T> Builder addUpdateItem(MappedTableResource<T> mappedTableResource,  
TransactUpdateItemEnhancedRequest<T> request)
```

(annotations)

DynamoDbBean

```
package software.amazon.awssdk.enhanced.dynamodb.mapper.annotations;  
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@SdkPublicApi  
public @interface DynamoDbBean
```

Indicates that the class is a DynamoDB entity. Fields in the class are automatically mapped to attributes in the DynamoDB table unless you customize the attribute name.

DynamoDbPartitionKey

```
package software.amazon.awssdk.enhanced.dynamodb.mapper.annotations;  
@Target({ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@BeanTableSchemaAttributeTag(BeanTableSchemaAttributeTags.class)  
@SdkPublicApi  
public @interface DynamoDbPartitionKey
```

Designate a field as **the partition key** (also known as **the hash key**) in a DynamoDB table.

Partition Key

The partition key **uniquely identifies a record** within a DynamoDB table. It is a mandatory attribute for every table.

Composite Key

If your table uses **a composite key (partition key and sort key)**, `@DynamoDbPartitionKey` marks the field as the partition key, while the sort key is marked with `@DynamoDbSortKey`.

Required for Queries

The partition key is used in query operations to retrieve specific items or groups of items that share the same partition key.

Usage

The Order class has both a partition key (`orderId`) and a sort key (`customerId`), meaning **multiple orders can share the same orderId** but must have unique `customerId` values.

```
@DynamoDbBean
public class Order {

    private String orderId;
    private String customerId;
    private String orderDate;

    @DynamoDbPartitionKey
    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    @DynamoDbSortKey
    public String getCustomerId() {
        return customerId;
    }

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getOrderDate() {
        return orderDate;
    }

    public void setOrderDate(String orderDate) {
        this.orderDate = orderDate;
    }
}
```

DynamoDbSortKey

```
package software.amazon.awssdk.enhanced.dynamodb.mapper.annotations;
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@BeanTableSchemaAttributeTag(BeanTableSchemaAttributeTags.class)
@SdkPublicApi
public @interface DynamoDbSortKey
```

Mark a field in a class as **the sort key** (also known as **the range key**) in a DynamoDB table.

Sort Key

Used in tables that have a composite primary key, which consists of a partition key (designated by `@DynamoDbPartitionKey`) and a sort key.

The combination of these two keys uniquely identifies an item in the table.

Required for Composite Keys

It is mandatory to have a sort key if your table uses a composite key.

The sort key allows for sorting and querying based on a range of values within the partition key.

Queries

You can query a DynamoDB table not just by partition key but also by sort key, allowing you to retrieve ranges of items that share the same partition key.

DynamoDbAttribute

```
package software.amazon.awssdk.enhanced.dynamodb.mapper.annotations;
```

```
@Target({ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@SdkPublicApi  
public @interface DynamoDbAttribute
```

Maps a field in a Java class to a corresponding attribute in a DynamoDB table.

By default, the field name is used as the attribute name in DynamoDB, but you can customize the attribute name by providing a specific value in the annotation.

```
String value();
```

[auth] (v2)

DefaultCredentialsProvider

```
package software.amazon.awssdk.auth.credentials;  
public final class DefaultCredentialsProvider implements AwsCredentialsProvider, SdkAutoCloseable,  
ToCopyableBuilder<Builder, DefaultCredentialsProvider>
```

DefaultCredentialsProvider checks various sources for AWS credentials in the following order:

Environment Variables:

If these are set, the provider uses them.

```
AWS_ACCESS_KEY_ID  
AWS_SECRET_ACCESS_KEY  
AWS_SESSION_TOKEN (optional, if using temporary security credentials)
```

Java System Properties:

These system properties can be set using -D flags when starting the JVM.

```
aws.accessKeyId  
aws.secretKey  
aws.sessionToken (optional)
```

AWS Profile Configuration File:

Typically located at `~/.aws/credentials` or `~/.aws/config`.

The default profile is used unless specified otherwise using `AWS_PROFILE` or the `aws.profile` system property.

Amazon ECS Container Credentials:

When running in an Amazon ECS container, credentials are retrieved from the ECS task role via the container metadata service.

Instance Profile Credentials (EC2 or Lambda):

When running on **EC2 or Lambda**, credentials are fetched from the instance's IAM role **via the EC2 metadata service**.

StaticCredentialsProvider

```
package software.amazon.awssdk.auth.credentials;  
@SdkPublicApi  
public final class StaticCredentialsProvider implements AwsCredentialsProvider  
    DynamoDbClient dynamoDbClient = DynamoDbClient.builder()  
        .credentialsProvider(StaticCredentialsProvider.create(AwsBasicCredentials.create("test",  
            "test")))  
        .region(Region.US_EAST_1) // Specify your desired region here  
        .endpointOverride(URI.create("http://192.168.127.128:4566")) // Local DynamoDB endpoint  
        .build();
```

AwsCredentialsProvider

```
package software.amazon.awssdk.auth.credentials;  
@FunctionalInterface  
@SdkPublicApi  
public interface AwsCredentialsProvider  
    AwsCredentials resolveCredentials();
```

[regions] (v2)

DefaultAwsRegionProviderChain

```
package software.amazon.awssdk.regions.providers;  
public final class DefaultAwsRegionProviderChain extends AwsRegionProviderChain
```

The DefaultAwsRegionProviderChain is part of the AWS SDK and is responsible for **automatically determining the AWS region** to be used by the client.

It **checks various locations in a predefined order** to identify the region.

This is useful when you want your application to run in different environments (like local development, EC2 instances, or Lambda) without explicitly setting the region in your code.

The DefaultAwsRegionProviderChain checks for the AWS region in the following order:

- AWS_REGION environment variable

- It checks if the environment variable AWS_REGION is set. This is typically configured on EC2 or ECS instances.

- aws.region system property

- It checks if the Java system property aws.region is set.

- AWS profile configuration file

- It looks for the region in the AWS credentials/config file, typically located in `~/.aws/config`.

- EC2 instance metadata

- If running on an EC2 instance, it retrieves the region **from the instance's metadata**.

- Default region

- If none of the above are set, it may fall back to a default or throw an error, depending on the service's configuration.

[aws-java-sdk-dynamodb] (v1)

DynamoDBTable

```
package com.amazonaws.services.dynamodbv2.datamodeling;
```

```
@DynamoDB
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Inherited
public @interface DynamoDBTable
```

The `@DynamoDBTable` annotation is used in [AWS SDK for Java v1](#) to map a Java class to a DynamoDB table.

This annotation is part of the [DynamoDB Object Persistence Model](#), which simplifies interaction with DynamoDB by mapping Java objects to database records.

```
String tableName();
```

DynamoDBHashKey

```
package com.amazonaws.services.dynamodbv2.datamodeling;
@DynamoDB
@DynamoDBKeyed(KeyType.HASH)
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface DynamoDBHashKey
```

This annotation is used to mark a field as the primary key (partition key) of a DynamoDB table.

Every DynamoDB table must have a hash key, which uniquely identifies an item in the table.

The `attributeName` parameter specifies the attribute name in the DynamoDB table.

```
String attributeName() default "";
```

Usage

```
@DynamoDBTable(tableName = "ProductCatalog")
public class Product {

    private String id;

    @DynamoDBHashKey(attributeName = "Id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

DynamoDBRangeKey

```
package com.amazonaws.services.dynamodbv2.datamodeling;
@DynamoDB
@DynamoDBKeyed(KeyType.RANGE)
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface DynamoDBRangeKey
```

This annotation is used to mark a field as the range key (sort key) of a DynamoDB table that uses a composite primary key (hash + range).

A range key works with the hash key to uniquely identify an item. While the hash key determines the partition, the range key sorts items within that partition.

Like `@DynamoDBHashKey`, the `attributeName` parameter is used to specify the attribute name in the table.

Usage

```
@DynamoDBTable(tableName = "ProductCatalog")
public class Product {
```

```

private String id;
private String name;

@DynamoDBHashKey(attributeName = "Id")
public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

@DynamoDBRangeKey(attributeName = "Name")
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}

```

DynamoDBAttribute

```

package com.amazonaws.services.dynamodbv2.datamodeling;
@DynamoDB
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface DynamoDBAttribute

```

This annotation is used to map any other field in a class to an attribute in a DynamoDB table.

The attributeName parameter is optional, but you can use it to specify a different attribute name than the field name in the Java class.

```

String attributeName() default "";
String mappedBy() default "";

Usage
@DynamoDBTable(tableName = "ProductCatalog")
public class Product {

    private String id;
    private String name;
    private String description;

    @DynamoDBHashKey(attributeName = "Id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    @DynamoDBRangeKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Description")
    public String getDescription() {
        return description;
    }
}

```

```

    }

    public void setDescription(String description) {
        this.description = description;
    }
}

```

DynamoDBMapper

```

package com.amazonaws.services.dynamodbv2.datamodeling;
public class DynamoDBMapper extends AbstractDynamoDBMapper

```

DynamoDBMapper is a **high-level object-to-document** (or object-to-item) **mapper** for Amazon DynamoDB.

It provides a simplified API for saving, querying, loading, and deleting objects stored in DynamoDB tables.

It allows developers to interact with DynamoDB in an object-oriented manner, automatically handling serialization and deserialization between DynamoDB items and Java objects.

Key Features

Object-Oriented Access

You can map Java objects to DynamoDB tables and columns, allowing developers to work with tables as if they were just Java classes.

Automatic Serialization/Deserialization

Handles the transformation between Java objects and DynamoDB's native item types (such as `String`, `Number`, `Map`, `List`).

Batch Operations

Supports batch save, load, and delete operations.

Query and Scan

Provides methods to execute queries and scans on DynamoDB tables.

```

public <T> PaginatedQueryList<T> query(Class<T> clazz, DynamoDBQueryExpression<T> queryExpression)
public <T> void save(T object)
public <T> T load(Class<T> clazz, Object hashKey)

```

Load an object from DynamoDB.

```

DynamoDBMapper mapper = new DynamoDBMapper(dynamoDBClient);
Long hashKey = 105L;
double rangeKey = 1.0d;
TestClass obj = mapper.load(TestClass.class, hashKey, rangeKey);
obj.getIntegerAttribute().add(42);
mapper.save(obj);
mapper.delete(obj);

```

AWS Lambda

Core

AWS Lambda is a **serverless compute service** provided by Amazon Web Services (AWS) that allows you to run code in response to events without provisioning or managing servers.

It automatically scales your application by running code in response to triggers, making it suitable for various use cases such as data processing, web applications, and real-time file processing.

Key Components

Functions

The core component of Lambda. A function is a piece of code written in a supported language (e.g., Python, Node.js, Java) that performs a specific task.

Triggers

Events that invoke Lambda functions. Triggers can come from various sources, such as:

AWS Services

S3 (object uploads), DynamoDB (table changes), SNS (notifications), and many others.

HTTP Requests

API Gateway can trigger Lambda functions via RESTful APIs.

Execution Role

An IAM role **that grants permissions to the Lambda function** to access other AWS services (e.g., S3, DynamoDB).

This role defines what resources the function can interact with.

Environment Variables

Key-value pairs that you can define **to pass configuration settings to your function at runtime**.

This allows you to manage configurations without changing the code.

Concurrency

The number of instances of a function that can run simultaneously.

AWS manages scaling based on incoming request volume, but you can set limits on concurrency.

Memory and Timeout Settings

You can configure **the amount of memory** allocated to a Lambda function (from 128 MB to 10 GB) and the maximum execution time (up to 15 minutes) for each invocation.

Deployment Packages

The code and dependencies packaged together to create a Lambda function.

These packages can be uploaded directly or stored in Amazon S3.

Layers

A way to manage code and dependencies across multiple Lambda functions.

Layers allow you to include additional libraries or custom runtimes that can be reused.

Monitoring and Logging

AWS Lambda integrates with Amazon CloudWatch for logging and monitoring. You can view logs, set up alerts, and track performance metrics.

VPC Integration

Lambda can be configured to access resources **within a Virtual Private Cloud (VPC)**, allowing it to interact with private services securely.

Configuration

Producer

Prerequisites

AWS Account

Ensure you have an AWS account and the necessary permissions to create and manage Lambda functions.

Spring Boot Project

Set up a Spring Boot project using your preferred method (e.g., Spring Initializr).

Add Dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-function-web</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-aws-lambda</artifactId>
</dependency>
```

For Gradle:

```
implementation 'org.springframework.cloud:spring-cloud-function-adapter-aws:4.0.2'

implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    AWS Lambda Core dependency
implementation 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    AWS Lambda Log4j2 Adapter (optional, for logging with Log4j2)
implementation 'software.amazon.awssdk:lambda:2.20.96'
    AWS SDK for Lambda (Optional, if you want to invoke AWS Lambda from your Java code)
implementation 'software.amazon.awssdk:s3:2.20.96'
    AWS SDK for S3 (if your Lambda function interacts with S3)
```

Create a Spring Function

Define a Spring function that encapsulates your business logic:

```
package com.example.lambda;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import java.util.function.Function;

@Configuration
public class FunctionConfiguration {

    @Bean
    public Function<String, String> greet() {
        return name -> "Hello, " + name + "!";
    }

    @Bean
    public Function<Integer, String> square() {
        return number -> "Square: " + (number * number);
    }
}
```

Implement a Lambda Handler

Old Version

Create a handler class that will be invoked by AWS Lambda:

The function beans in a Spring Boot application using Spring Cloud Function **will** be automatically mapped to the **LambdaHandler** based on the parameter types of the function beans.

```
package com.example.lambda;

import org.springframework.cloud.function.adapter.aws.SpringBootRequestHandler;

public class LambdaHandler extends SpringBootRequestHandler<Object, Object> {
    // No additional code needed; Spring Cloud handles the mapping
}
```

Spring Cloud Function 3.x and 4.x

However, using this class isn't mandatory, as AWS Lambda can directly invoke your function through Spring Cloud Function's auto-configuration.

Starting with Spring Cloud Function 3.x+, you do not need to configure the `FunctionInvoker` explicitly in most cases. Spring Cloud Function's AWS adapter is designed to automatically route incoming AWS Lambda events to your Spring Boot-defined functions, making it much simpler and more function-centric.

```
package com.example;

import org.springframework.cloud.function.adapter.aws.FunctionInvoker;

public class MyLambdaHandler extends FunctionInvoker {
    // This will route AWS Lambda requests to your Spring Boot functions
}
```

Package and Deploy

Create a Deployment ZIP File

AWS Lambda requires a ZIP file containing your JAR and dependencies. You can create the ZIP file with this command:

```
zip -j my-lambda-project.zip build/libs/my-lambda-project-0.0.1-SNAPSHOT.jar
```

Or once your function is defined, package the application as a fat JAR (or deployable ZIP) using the shadow plugin, for example:

```
./gradlew build shadowJar
```

Create and Deploy the Lambda Function

Deploy Using AWS Management Console

```
aws lambda create-function \
--function-name MyLambdaFunction \
--zip-file fileb://my-lambda-project.zip \
--handler com.example.lambda.MyLambdaHandler \
--runtime java17 \
--role arn:aws:iam::123456789012:role/lambda-role
```

Configure Lambda Function

Set the handler to `yourPackage.LambdaApplication::helloWorld`.

Choose the appropriate runtime (e.g., Java 11).

Configure any necessary environment variables or IAM roles.

When deploying your Lambda function, set the handler in the AWS Lambda configuration to:

```
com.example.MyLambdaHandler
```

Alternatively, if no handler class is provided, Spring Cloud Function will automatically expose the function defined in your Spring Boot app (e.g., uppercase) as the Lambda entry point.

Test

Test your Lambda function by invoking it from the AWS Management Console or using the AWS CLI.

Additional Considerations

Event-Driven Functions

If your Lambda function is triggered by events (e.g., S3 object uploads, API Gateway requests), configure the appropriate

event source mapping.

Concurrency

Adjust the maximum concurrency limit for your Lambda function to handle varying workloads.

Custom Runtime

For more control, consider using a custom runtime to tailor the execution environment.

Serverless Framework

Explore the Serverless Framework for simplified deployment and management of serverless applications, including Lambda functions.

Consumer

Configure AWS Credentials

You need to set up AWS credentials for accessing DynamoDB. You can provide them in different ways:

- Environment Variables
Set `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
- AWS Credentials File
Located in `~/.aws/credentials`:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```
- Spring Boot Application Properties
In your `application.properties` or `application.yml`, add:

```
cloud.aws.credentials.accessKey=YOUR_ACCESS_KEY
cloud.aws.credentials.secretKey=YOUR_SECRET_KEY
cloud.aws.region.static=YOUR_REGION
```

Add Dependencies

```
implementation 'software.amazon.awssdk:lambda:2.20.20' // Use the latest version available
```

Invoke the Lambda Function

In your Spring Boot application, you can use the `LambdaClient` class provided by the AWS SDK to invoke the Lambda function.

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.core.SdkBytes;
import org.springframework.stereotype.Service;

@Service
public class LambdaService {

    private final LambdaClient lambdaClient;

    public LambdaService() {
        this.lambdaClient = LambdaClient.builder()
            .region(Region.US_EAST_1) // Specify the AWS region
            .build();
    }

    public String invokeLambdaFunction(String functionName, String payload) {
        // Create an InvokeRequest
        InvokeRequest invokeRequest = InvokeRequest.builder()
            .functionName(functionName)
            .payload(SdkBytes.fromUtf8String(payload))
            .build();
    }
}
```

```

    // Invoke the Lambda function
    InvokeResponse invokeResponse = lambdaClient.invoke(invokerRequest);

    // Get the response from the Lambda function
    String response = invokeResponse.payload().asUtf8String();

    return response; // Return the Lambda function's output
}
}

```

Use the Service to Invoke the Lambda Function

Once the LambdaService is created, you can inject it into a controller or another service to invoke the Lambda function.

```

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class LambdaController {

    private final LambdaService lambdaService;

    public LambdaController(LambdaService lambdaService) {
        this.lambdaService = lambdaService;
    }

    @GetMapping("/invokeLambda")
    public String invokeLambda(@RequestParam String functionName, @RequestParam String payload) {
        // Call the service method to invoke the Lambda function
        return lambdaService.invokeLambdaFunction(functionName, payload);
    }
}

```

Example Payload

Assuming your Lambda function takes a JSON payload, you can invoke it like this:

```
curl "http://localhost:8080/invokeLambda?functionName=myLambdaFunction&payload={"key":"value"}"
```

[spring-cloud-function-adapter-aws]

SpringBootRequestHandler

```

package org.springframework.cloud.function.adapter.aws;
public class SpringBootRequestHandler<E, O> extends AbstractSpringFunctionAdapterInitializer<Context> implements
RequestHandler<E, Object>

```

When AWS Lambda invokes your LambdaHandler, it will pass the input to the handler.

The handlexr **will look at the input type** to determine which function bean to call.

You do not need to define multiple SpringBootRequestHandler classes for different function beans in a Spring Cloud Function application.

You can use a single LambdaHandler that can route requests to different function beans based on the input type.

AWS IAM

AWS IAM (**Identity and Access Management**) is a service that allows you to securely control access to AWS resources. It provides a way to manage users, groups, and roles, as well as define permissions for these entities.

Components

Users

Individual people who can access AWS resources.

Groups

Collections of users that can be assigned permissions together.

Roles

Temporary security credentials that can be assumed by users or applications to access specific resources.

Policies

Documents that define permissions for users, groups, and roles. They can be managed at the account, group, or role level.

Benefits

Centralized management

Manage users, groups, and permissions from a single console.

Fine-grained control

Define granular permissions to control access to specific resources.

Enhanced security

Protect your AWS resources from unauthorized access.

Simplified administration

Use roles to delegate access without sharing credentials.

Compliance

Meet regulatory requirements by implementing strong access controls.

Common use cases

Managing user access

Granting permissions to employees or contractors to access specific AWS resources.

Delegating access

Assigning temporary permissions to applications or services to access AWS resources.

Implementing multi-factor authentication (MFA)

Adding an extra layer of security to user accounts.

Complying with regulations

Ensuring compliance with industry standards and regulations.

AWS S3

Core

Amazon S3 (Simple Storage Service) is a scalable object storage service provided by Amazon Web Services (AWS) for storing and retrieving any amount of data at any time.

It is designed for durability, availability, and performance, making it suitable for a wide range of use cases.

Key Components

Buckets

A container for storing objects in S3. Each bucket has a unique name within the AWS region.

Usage

You create buckets to organize and manage your data.

Objects

The fundamental entity stored in S3, consisting of the data itself, a unique identifier (key), and metadata.

Usage

Objects can be any type of file, such as images, videos, documents, or backups.

Key

A unique identifier for each object within a bucket. The key **is the name you assign to the object**.

Usage

Used to retrieve and manage objects in S3.

Regions

Geographic locations where AWS data centers are situated.

S3 buckets are created in specific regions.

Usage

Choosing a region can affect latency and compliance.

Storage Classes

Different types of storage options that optimize cost and performance based on usage patterns.

Types

S3 Standard

General-purpose storage for frequently accessed data.

S3 Intelligent-Tiering

Automatically moves data between two access tiers based on changing access patterns.

S3 Standard-IA

Infrequently accessed data with lower storage costs.

S3 Glacier

Low-cost storage for archiving data with retrieval times ranging from minutes to hours.

Permissions and Access Control

Bucket Policies

JSON-based policies to control access at the bucket level.

IAM Policies

Manage permissions for AWS users and roles.

Access Control Lists (ACLs)

Fine-grained permissions for individual objects.

Versioning

A feature that allows you to keep multiple versions of an object in a bucket.

Usage

Helps in data recovery and tracking changes over time.

Lifecycle Policies

Rules that automate the transition of objects between storage classes or the deletion of objects after a specified period.

Usage

Manage data lifecycle to optimize costs.

Event Notifications

Mechanisms to trigger notifications (e.g., to AWS Lambda, SNS, or SQS) based on object events like uploads or deletions.

Usage

Facilitates automated workflows based on changes in the bucket.

Data Transfer Acceleration

A feature that speeds up uploads to S3 by using Amazon CloudFront's globally distributed edge locations.

Usage

Reduces latency for large file uploads.

Set Up AWS Account

Create an AWS account if you don't have one.

Create an IAM user with programmatic access and attach the `AmazonS3FullAccess` policy (or a more restricted policy based on your needs).

Note down the Access Key ID and Secret Access Key.

Add Dependencies

Add the AWS SDK for S3 dependency to your `pom.xml` (for Maven) or `build.gradle` (for Gradle).

For Maven:

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-s3</artifactId>
    <version>1.12.300</version> <!-- Check for the latest version -->
</dependency>
```

For Gradle:

```
implementation 'com.amazonaws:aws-java-sdk-s3:1.12.300' // Check for the latest version
```

Configure AWS Credentials

You can configure AWS credentials in multiple ways. The simplest way during development is to create an `application.properties` or `application.yml` file.

In `application.properties`:

```
cloud.aws.credentials.access-key=YOUR_ACCESS_KEY
cloud.aws.credentials.secret-key=YOUR_SECRET_KEY
cloud.aws.region.static=us-west-2 # Set your desired region
```

Create an S3 Configuration Class

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class S3Config {
    @Bean
    public AmazonS3 s3client() {
        return AmazonS3ClientBuilder.standard()
            .withRegion("us-west-2") // Your desired region
            .build();
    }
}
```

Create a Service to Interact with S3

Create a service class that will handle S3 operations.

```
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.model.ObjectMetadata;
import com.amazonaws.services.s3.model.S3Object;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.io.InputStream;

@Service
public class S3Service {
```

```

private final AmazonS3 amazonS3;
private final String bucketName = "your-bucket-name";

@Autowired
public S3Service(AmazonS3 amazonS3) {
    this.amazonS3 = amazonS3;
}

public void uploadFile(String key, InputStream inputStream, ObjectMetadata metadata) {
    amazonS3.putObject(bucketName, key, inputStream, metadata);
}

public S3Object downloadFile(String key) {
    return amazonS3.getObject(bucketName, key);
}

public void deleteFile(String key) {
    amazonS3.deleteObject(bucketName, key);
}
}

```

Use the Service in Your Application

You can now inject `S3Service` into your controllers or other services and use it to perform S3 operations.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.io.InputStream;

@RestController
@RequestMapping("/api/s3")
public class S3Controller {
    @Autowired
    private S3Service s3Service;

    @PostMapping("/upload")
    public String uploadFile(@RequestParam String key, @RequestParam InputStream inputStream) {
        ObjectMetadata metadata = new ObjectMetadata();
        metadata.setContentLength(inputStream.available());
        s3Service.uploadFile(key, inputStream, metadata);
        return "File uploaded successfully!";
    }

    @GetMapping("/download/{key}")
    public S3Object downloadFile(@PathVariable String key) {
        return s3Service.downloadFile(key);
    }

    @DeleteMapping("/delete/{key}")
    public String deleteFile(@PathVariable String key) {
        s3Service.deleteFile(key);
        return "File deleted successfully!";
    }
}

```

[aws-java-sdk-s3]

AmazonS3

```

package com.amazonaws.services.s3;
public interface AmazonS3 extends S3DirectSpi

```

com.amazonaws.services.s3.internal.S3DirectSpi

```

public void setRegion(com.amazonaws.regions.Region region) throws IllegalArgumentException;

```

```
public ObjectListing listObjects(String bucketName) throws SdkClientException, AmazonServiceException;  
  
public void deleteObject(String bucketName, String key) throws SdkClientException, AmazonServiceException;
```

S3DirectSpi

```
PutObjectResult putObject(PutObjectRequest req);  
S3Object getObject(GetObjectRequest req);  
ObjectMetadata getObject(GetObjectRequest req, File dest);  
  
CompleteMultipartUploadResult completeMultipartUpload(CompleteMultipartUploadRequest req);  
InitiateMultipartUploadResult initiateMultipartUpload(InitiateMultipartUploadRequest req);  
UploadPartResult uploadPart(UploadPartRequest req);  
CopyPartResult copyPart(CopyPartRequest req);  
void abortMultipartUpload(AbortMultipartUploadRequest req);
```

AmazonS3ClientBuilder

```
package com.amazonaws.services.s3;  
@NotThreadSafe  
public final class AmazonS3ClientBuilder extends AmazonS3Builder<AmazonS3ClientBuilder, AmazonS3>  
  
public static AmazonS3ClientBuilder standard()  
public final Subclass withCredentials(AWSCredentialsProvider credentialsProvider)  
public final Subclass withClientConfiguration(ClientConfiguration config)
```

[s3]

S3AsyncClient

```
package software.amazon.awssdk.services.s3;  
public interface S3AsyncClient extends SdkClient  
  
static S3AsyncClientBuilder builder()  
    Create a builder that can be used to configure and create a S3AsyncClient.  
    S3AsyncClientBuilder builder = S3AsyncClient.builder()  
        .httpClient(s3LowConcurrencyHttpClient)  
        .region(Region.CN_NORTHWEST_1)  
        .serviceConfiguration(serviceConfiguration);
```

S3Configuration

```
package software.amazon.awssdk.services.s3;  
public final class S3Configuration implements ServiceConfiguration, ToCopyableBuilder<S3Configuration.Builder,  
S3Configuration>  
  
public static Builder builder()  
    Create a S3Configuration. Builder, used to create a S3Configuration.  
    S3Configuration serviceConfiguration = S3Configuration.builder()  
        .checksumValidationEnabled(false)  
        .chunkedEncodingEnabled(true)
```

```
.build();
```

Amazon SNS

Core

Amazon Simple Notification Service (SNS) is a **fully managed** messaging service that enables the decoupling of microservices, distributed systems, and serverless applications.

It allows you to send messages to various subscribers and endpoints, making it easier to build scalable applications.

Key Components

Topics

Central communication channels to which messages are published.

Subscribers can subscribe to these topics.

Subscriptions

Mechanisms for endpoints (like email, SMS, or HTTP) to receive messages published to a topic.

Each subscription can have different protocols.

Subscribes an endpoint (e.g., email, SMS, or Lambda) to a topic so that the endpoint receives messages from the topic.

Messages

The data sent through SNS. Messages can be up to 256 KB in size and can include JSON content for structured data.

Endpoints

The target for messages, which can be email addresses, phone numbers (for SMS), HTTP/HTTPS endpoints, or even other AWS services.

Message Filtering

Allows subscribers to receive only specific messages based on message attributes, enabling more targeted communication.

Dead Letter Queues (DLQs)

Used to handle messages that fail to be delivered to their endpoints after a certain number of attempts, improving reliability.

Mobile Push Notifications

SNS supports sending notifications to mobile devices through services like Apple Push Notification Service (APNS) and Google Cloud Messaging (GCM).

Integration with Other AWS Services

SNS integrates with services like AWS Lambda, Amazon SQS, and Amazon CloudWatch, allowing for more complex workflows and monitoring.

Configuration

Producer

Add Dependencies

Add the AWS SDK dependency for SNS in your `pom.xml`:

```
<dependency>
<groupId>com.amazonaws</groupId>
```

```
<artifactId>aws-java-sdk-sns</artifactId>
<version>1.12.XXX</version> <!-- Use the latest version -->
</dependency>
```

For Gradle:

```
implementation 'software.amazon.awssdk:sns'
implementation 'software.amazon.awssdk:auth'
```

Configure AWS Credentials

You can configure your AWS credentials using one of the following methods:

Environment Variables

Set `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.

AWS Credentials File

Store your credentials in `~/.aws/credentials`.

Configure SNS Client

Create a configuration class for the SNS client:

```
import com.amazonaws.services.sns.AmazonSNS;
import com.amazonaws.services.sns.AmazonSNSClientBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SnsConfig {

    @Bean
    public AmazonSNS amazonSNS() {
        return AmazonSNSClientBuilder.standard()
            .withRegion("us-east-1") // Set your region
            .build();
    }

    // An alternative for AmazonSNS.
    @Bean
    public AmazonSNSAsync amazonSNS() {
        return AmazonSNSAsyncClientBuilder.standard().build();
    }
}
```

Create an SNS Service

Create a service class to handle SNS operations:

```
import com.amazonaws.services.sns.AmazonSNS;
import com.amazonaws.services.sns.model.PublishRequest;
import com.amazonaws.services.sns.model.PublishResult;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SnsService {

    @Autowired
    private AmazonSNS amazonSNS;
```

```

public String publishMessage(String topicArn, String message) {
    PublishRequest publishRequest = new PublishRequest(topicArn, message);
    PublishResult publishResult = amazonSNS.publish(publishRequest);
    return publishResult.getMessageId();
}
}

```

Using the SNS Service

You can now use the `SnsService` in your controllers or other services:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class NotificationController {

    @Autowired
    private SnsService snsService;

    @PostMapping("/send-notification")
    public String sendNotification(@RequestBody NotificationRequest request) {
        return snsService.publishMessage(request.getTopicArn(), request.getMessage());
    }
}

class NotificationRequest {
    private String topicArn;
    private String message;

    // Getters and setters
}

```

Testing

Run your Spring Boot application and send a POST request to `/send-notification` with a JSON body like:

```
{
    "topicArn": "arn:aws:sns:us-east-1:123456789012:YourTopic",
    "message": "Hello from Spring Boot!"
}
```

Consumer

Configure AWS Credentials

You need to set up AWS credentials for accessing DynamoDB. You can provide them in different ways:

- Environment Variables
Set `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
- AWS Credentials File
Located in `~/.aws/credentials`:


```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```
- Spring Boot Application Properties
In your `application.properties` or `application.yml`, add:


```
cloud.aws.credentials.accessKey=YOUR_ACCESS_KEY
cloud.aws.credentials.secretKey=YOUR_SECRET_KEY
cloud.aws.region.static=YOUR_REGION
```

Add Dependencies

Add the required dependencies to your pom.xml if you're using Maven:

```
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>sqs</artifactId>
    <version>2.20.0</version> <!-- Use the latest version -->
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```

Create an SQS Consumer

Instead of implementing a separate service class, you can create a simple consumer method that you can invoke periodically, such as from a scheduled task. Here's a more straightforward approach:

Add a Scheduled Task:

You can use Spring's @Scheduled annotation to create a method that polls the SQS queue at regular intervals.

Polling Method:

Create a method within your main application class or any other suitable component.

```
package com.simil.consumer.consumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.sqs.SqsClient;
import software.amazon.awssdk.services.sqs.model.ReceiveMessageRequest;
import software.amazon.awssdk.services.sqs.model.Message;
import software.amazon.awssdk.services.sqs.model.DeleteMessageRequest;

import java.util.List;

@Component
@EnableScheduling
public class SqsConsumer {

    @Autowired
    private SqsClient sqsClient;

    @Value("${cloud.aws.sqs.queueUrl}") // Assuming you have set this in your application properties
    private String queueUrl;

    @Scheduled(fixedDelay = 5000) // Poll every 5 seconds
    public void pollQueue() {
        List<Message> messages = sqsClient.receiveMessage(ReceiveMessageRequest.builder()
            .queueUrl(queueUrl)
            .waitTimeSeconds(20) // Long polling
            .build()
            .messages());

        for (Message message : messages) {
            processMessage(message);
            deleteMessage(message);
        }
    }
}
```

```

private void processMessage(Message message) {
    // Handle the received message
    System.out.println("Received message: " +message.messageId()+" "+ message.body());
}

private void deleteMessage(Message message) {
    sqsClient.deleteMessage(DeleteMessageRequest.builder()
        .queueUrl(queueUrl)
        .receiptHandle(message.receiptHandle())
        .build());
}
}

```

Configure SQS Client

You also need to create an SqsClient bean and configure the queue URL in your Spring Boot application:

```

package com.simil.consumer.AAAConfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;
import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.sqs.SqsClient;

import java.net.URI;

@Configuration
public class AwsSqsConfig {

    @Bean
    public SqsClient sqsClient() {
        return SqsClient.builder()
            .credentialsProvider(StaticCredentialsProvider.create(
                AwsBasicCredentials.create("test", "test")
            ))
            .endpointOverride(URI.create("http://192.168.127.128:4566"))
            .region(Region.US_EAST_1) // Set your AWS region here
            .build();
    }

    @Bean
    public String queueUrl() {
        return "http://localhost:4566/000000000000/SimiQueue"; // Replace with your SQS queue URL
    }
}

```

Run Your Application

Once you run your Spring Boot application,
it will continuously poll the SQS queue for new messages and process them as they arrive.
The messages received will come from the SNS topic, as the SQS queue is subscribed to it.

[aws-java-sdk-sns]

AmazonSNS

```

package com.amazonaws.services sns;
public interface AmazonSNS

PublishResult publish(PublishRequest publishRequest);

```

PublishRequest

```
package com.amazonaws.services.sns.model;
public class PublishRequest extends com.amazonaws.AmazonWebServiceRequest implements Serializable, Cloneable

public PublishRequest withMessage(String message)
public PublishRequest withMessageAttributes(java.util.Map<String, MessageAttributeValue> messageAttributes)
public PublishRequest withTopicArn(String topicArn)
```

[sns]

SqsAsyncClient

```
package software.amazon.awssdk.services.sqs;
@Generated("software.amazon.awssdk:codegen")
@SdkPublicApi
@ThreadSafe
public interface SqsAsyncClient extends AwsClient

static SqsAsyncClientBuilder builder()
    Create a SqsAsyncClient.
        SqsAsyncClient.builder()
            .httpClient(sqsAsyncHttpClient)
            .build();
```

SqsClient

```
package software.amazon.awssdk.services.sqs;
@Generated("software.amazon.awssdk:codegen")
@SdkPublicApi
@ThreadSafe
public interface SqsClient extends AwsClient

static SqsClient create()
static SqsClientBuilder builder()
```

Amazon EC2

Amazon Elastic Compute Cloud (EC2) is a web service that provides resizable compute capacity in the cloud. It allows users to run virtual servers (instances) on-demand, making it easy to scale applications up or down based on needs.

Key Components

Instances

Virtual servers running applications.

EC2 offers various instance types optimized for different use cases (e.g., compute, memory, storage).

Amazon Machine Images (AMIs)

Pre-configured templates used to create EC2 instances.

An AMI includes the operating system, application server, and applications.

Login Account

Account `ec2-user` is the default user created when you launch an EC2 instance **using the Amazon Linux or Amazon Linux 2 AMIs**.

It has the necessary permissions to perform administrative tasks and is configured for easy use with SSH.

LocalStack EC2 instance

LocalStack will create a mock EC2 instance. However, this instance **doesn't exist in the real world**.

It won't have a functioning public IP or operating system for you to connect to.

Where to Find AMI IDs

AWS Console:

Navigate to the EC2 dashboard in the AWS Management Console.

Click on AMIs under the Images section in the left sidebar.

Here, you can search for AMIs **based on various criteria** (e.g., public/private, OS type, architecture).

AWS CLI:

You can also find AMI IDs using the AWS CLI.

For example, to list all the public AMIs for a specific operating system (e.g., Amazon Linux 2), you can run:

```
aws ec2 describe-images --owners amazon --filters "Name=name,Values=amzn2-ami-hvm-*-x86_64-gp2" --query "Images[*].[ImageId, Name]" --output table
```

This command will give you a table of available Amazon Linux 2 AMI IDs.

LocalStack:

If you're using LocalStack, you can use a dummy AMI ID, as LocalStack simulates AWS services.

A common practice is to use an AMI ID like `ami-0123456789abcdef0`,

as mentioned in your command. LocalStack doesn't require a valid AMI ID for running instances since it's a local environment.

Instance Types

Different configurations of CPU, memory, storage, and networking capabilities.

Types are categorized into families, such as **General Purpose**, **Compute Optimized**, and **Memory Optimized**.

Elastic Block Store (EBS)

A block storage service that provides persistent storage volumes for EC2 instances. EBS volumes can be attached to instances and used like hard drives.

Security Groups

Virtual firewalls that control inbound and outbound traffic to instances.

They define rules based on **IP address**, **port**, and **protocol**.

Key Pairs

Security credentials used to access EC2 instances.

A key pair consists of a **public key** (stored in AWS) and a **private key** (kept by the user).

Elastic Load Balancing (ELB)

A service **that automatically distributes incoming application traffic** across multiple EC2 instances, enhancing availability and fault tolerance.

Auto Scaling

A feature that automatically adjusts the number of EC2 instances in response to demand, ensuring optimal resource utilization and cost management.

VPC (Virtual Private Cloud)

A logically isolated network that allows you to launch AWS resources, including EC2 instances, in a virtual environment.

Monitoring and Management

Tools like Amazon CloudWatch provide monitoring for EC2 instances, including metrics such as CPU usage, memory usage, and network traffic.

Spot Instances

A purchasing option for EC2 instances that allows you to bid on unused capacity at potentially lower costs. Ideal for flexible and fault-tolerant applications.

Dedicated Hosts and Reserved Instances

Dedicated Hosts

Physical servers dedicated to your use, offering control over instance placement.

Reserved Instances

Provide significant savings for long-term usage by reserving instances for a specified term.

Configuration

Launch an EC2 Instance

To deploy your Spring Boot application on EC2, you'll need to launch an EC2 instance.

Create a Key Pair

```
aws ec2 create-key-pair --key-name SimiKeyPair --query 'KeyMaterial' --output text > SimiKeyPair.pem  
chmod 400 SimiKeyPair.pem
```

Launch the EC2 Instance

Note: LocalStack does not provide actual AMIs or security groups, so this part is primarily for demonstration.

```
aws --endpoint-url=http://localhost:4566 ec2 run-instances \  
--image-id ami-0123456789abcdef0 \ # Replace with a valid AMI ID for LocalStack (can be a dummy ID)  
--instance-type t2.micro \  
--key-name MyKeyPair \  
--security-group-ids sg-0123456789abcdef0 \ # Replace with a valid security group ID  
--count 1 \  
--associate-public-ip-address
```

Retrieve Instance Information

You can describe the instance to get its details:

Copy the Spring Boot JAR to the EC2 Instance

Use SCP to Copy the JAR

Once your EC2 instance is running, you'll want to copy the JAR file to it using SCP (Secure Copy Protocol):
Replace <public-ip> with the public IP address of your EC2 instance (if you are using a real EC2 instance).

```
scp -i MyKeyPair.pem target/my-spring-boot-app.jar ec2-user@<public-ip>/home/ec2-user/
```

SSH into the EC2 Instance

```
ssh -i MyKeyPair.pem ec2-user@<public-ip>
```

Run the Spring Boot Application

Once logged in, run your Spring Boot application:

Your Spring Boot application should now be running. You can access it through the EC2 instance's public IP address.

```
java -jar my-spring-boot-app.jar
```