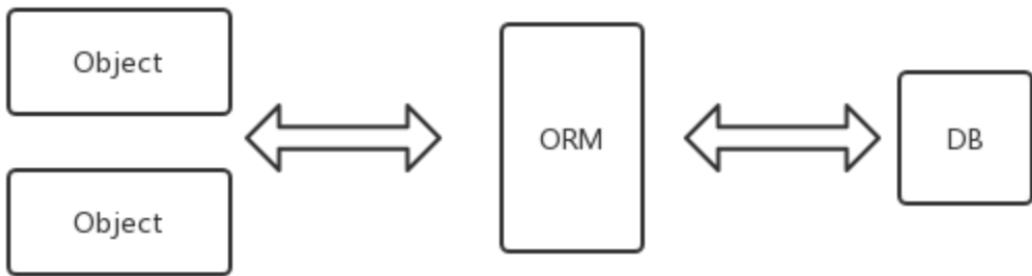


MyBatis

基本介绍

ORM (Object Relational Mapping) : 对象关系映射，指的是持久化数据和实体对象的映射模式，解决面向对象与关系型数据库存在的互不匹配的现象



MyBatis:

- MyBatis 是一个优秀的基于 Java 的持久层框架，它内部封装了 JDBC，使开发者只需关注 SQL 语句本身，而不需要花费精力去处理加载驱动、创建连接、创建 Statement 等过程。
- MyBatis 通过 XML 或注解的方式将要执行的各种 Statement 配置起来，并通过 Java 对象和 Statement 中 SQL 的动态参数进行映射生成最终执行的 SQL 语句。
- MyBatis 框架执行 SQL 并将结果映射为 Java 对象并返回。采用 ORM 思想解决了实体和数据库映射的问题，对 JDBC 进行了封装，屏蔽了 JDBC 底层 API 的调用细节，使我们不用操作 JDBC API，就可以完成对数据库的持久化操作。

MyBatis 官网地址: <http://www.mybatis.org/mybatis-3/>

参考视频: <https://space.bilibili.com/37974444/>

基本操作

相关API

Resources: 加载资源的工具类

- `InputStream getResourceAsStream(String fileName)`: 通过类加载器返回指定资源的字节流
 - 参数 fileName 是放在 src 的核心配置文件名: MyBatisConfig.xml

SqlSessionFactoryBuilder: 构建器，用来获取 SqlSessionFactory 工厂对象

- `SqlSessionFactory build(InputStream is)`: 通过指定资源的字节输入流获取 SqlSession 工厂对象

SqlSessionFactory: 获取 SqlSession 构建者对象的工厂接口

- `SqlSession openSession()`: 获得 SqlSession 构建者对象，并开启手动提交事务
- `SqlSession openSession(boolean)`: 获得 SqlSession 构建者对象，参数为 true 开启自动提交事务

SqlSession: 构建者对象接口，用于执行 SQL、管理事务、接口代理

- SqlSession 代表和数据库的一次会话，用完必须关闭
- SqlSession 和 Connection 一样都是非线程安全，每次使用都应该去获取新的对象

注: `update` 数据需要提交事务，或开启默认提交

SqlSession 常用 API:

方法	说明
List selectList(String statement, Object parameter)	执行查询语句，返回List集合
T selectOne(String statement, Object parameter)	执行查询语句，返回一个结果对象
int insert(String statement, Object parameter)	执行新增语句，返回影响行数
int update(String statement, Object parameter)	执行删除语句，返回影响行数
int delete(String statement, Object parameter)	执行修改语句，返回影响行数
void commit()	提交事务
void rollback()	回滚事务
T getMapper(Class cls)	获取指定接口的代理实现类对象
void close()	释放资源

映射配置

映射配置文件包含了数据和对象之间的映射关系以及要执行的 SQL 语句，放在 src 目录下

命名：StudentMapper.xml

- 映射配置文件的文件头：

```
<?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">
```

- 根标签：
 - ：核心根标签
 - namespace：属性，名称空间
- 功能标签：
 - <select>：查询功能标签
 - ：新增功能标签
 - ：修改功能标签
 - ：删除功能标签
 - id：属性，唯一标识，配合名称空间使用
 - resultType：指定结果映射对象类型，和对应的方法的返回值类型（全限定名）保持一致，但是如果返回值是 List 则和其泛型保持一致
 - parameterType：指定参数映射对象类型，必须和对应的方法的参数类型（全限定名）保持一致
 - statementType：可选 STATEMENT, PREPARED 或 CALLABLE, 默认值：PREPARED
 - STATEMENT：直接操作 SQL，使用 Statement 不进行预编译，获取数据：\$
 - PREPARED：预处理参数，使用 PreparedStatement 进行预编译，获取数据：#
 - CALLABLE：执行存储过程，CallableStatement
- 参数获取方式：
 - SQL 获取参数：#{属性名}

```
<mapper namespace="StudentMapper">
    <select id="selectById" resultType="student" parameterType="int">
        SELECT * FROM student WHERE id = #{id}
    </select>
</mapper>
```

强烈推荐官方文档：<https://mybatis.org/mybatis-3/zh/sqlmap-xml.html>

核心配置

核心配置文件包含了 MyBatis 最核心的设置和属性信息，如数据库的连接、事务、连接池信息等

命名：MyBatisConfig.xml

- 核心配置文件的文件头：

```
<?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">
```

- 根标签：
 - : 核心根标签
- 引入连接配置文件：
 - : 引入数据库连接配置文件标签
 - resource: 属性, 指定配置文件名

```
<properties resource="jdbc.properties"/>
```

- 调整设置
 - : 可以改变 Mybatis 运行时行为
- 起别名：
 - : 为全类名起别名的父标签
 - : 为全类名起别名的子标签
 - type: 指定全类名
 - alias: 指定别名
 - : 为指定包下所有类起别名的子标签, 别名就是类名, 首字母小写

```
<!--起别名-->
<typeAliases>
    <typeAlias type="bean.Student" alias="student"/>
    <package name="com.seazean.bean"/>
        <!--二选一-->
</typeAliases>
```

- : 自带别名:

别名	数据类型
string	java.lang.String
long	java.lang.Lang
int	java.lang.Integer
double	java.lang.Double
boolean	java.lang.Boolean
....

- 配置环境, 可以配置多个标签
 - : 配置数据库环境标签, default 属性指定哪个 environment
 - : 配置数据库环境子标签, id 属性是唯一标识, 与 default 对应
 - : 事务管理标签, type 属性默认 JDBC 事务
 - : 数据源标签
 - type 属性: POOLED 使用连接池 (MyBatis 内置), UNPOOLED 不使用连接池
 - : 数据库连接信息标签。
 - name 属性取值: driver, url, username, password
 - value 属性取值: 与 name 对应
- 引入映射配置文件
 - : 引入映射配置文件标签
 - : 引入映射配置文件子标签
 - resource: 属性指定映射配置文件的名称
 - url: 引用网路路径或者磁盘路径下的 sql 映射文件
 - class: 指定映射配置类
 - : 批量注册

参考官方文档: <https://mybatis.org/mybatis-3/zh/configuration.html>

#{}和\${}

#{}: 占位符, 传入的内容会作为字符串加上引号, 以预编译的方式传入, 将 sql 中的 #{} 替换为 ? 号, 调用 PreparedStatement 的 set 方法来赋值, 有效的防止 SQL 注入, 提高系统安全性

\${}: 拼接符, 传入的内容会直接替换拼接, 不会加上引号, 可能存在 sql 注入的安全隐患

- 能用 #{} 的地方就用 #{}, 不用或少用 \${}
- 必须使用 \${} 的情况:
 - 表名作参数时, 如: `SELECT * FROM ${tableName}`
 - order by 时, 如: `SELECT * FROM t_user ORDER BY ${columnName}`
- sql 语句使用 #{}, properties 文件内容获取使用 \${}

日志文件

在日常开发过程中, 排查问题时需要输出 MyBatis 真正执行的 SQL 语句、参数、结果等信息, 就可以借助 log4j 的功能来实现执行信息的输出。

- 在核心配置文件根标签内配置 log4j

```
<!--配置LOG4J-->
<settings>
    <setting name="logImpl" value="log4j"/>
</settings>
```

- 在 src 目录下创建 log4j.properties

```
# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

#输出到日志文件
#log4j.appender.file=org.apache.log4j.FileAppender
#log4j.appender.file.File=../logs/iaask.log
#log4j.appender.file.layout=org.apache.log4j.PatternLayout
#log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %l %m%n
```

- pom.xml

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.21</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.21</version>
</dependency>
```

代码实现

- 实体类

```
public class Student {
    private Integer id;
    private String name;
    private Integer age;
    ....
}
```

- StudentMapper

```
public interface StudentMapper {
    //查询全部
    public abstract List<Student> selectAll();
    //根据id查询
}
```

```

public abstract Student selectById(Integer id);

//新增数据
public abstract Integer insert(Student stu);

//修改数据
public abstract Integer update(Student stu);

//删除数据
public abstract Integer delete(Integer id);
}

```

- config.properties

```

driver=com.mysql.jdbc.Driver
url=jdbc:mysql://192.168.2.184:3306/db1
username=root
password=123456

```

- MyBatisConfig.xml

```

<?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">

<!--核心根标签-->
<configuration>
    <!--引入数据库连接的配置文件-->
    <properties resource="jdbc.properties"/>

    <!--配置LOG4J-->
    <settings>
        <setting name="logImpl" value="log4j"/>
    </settings>

    <!--起别名-->
    <typeAliases>
        <typeAlias type="bean.Student" alias="student"/>
        <!--<package name="bean"/>-->
    </typeAliases>

    <!--配置数据库环境，可以多个环境，default指定哪个-->
    <environments default="mysql">
        <!--id属性唯一标识-->
        <environment id="mysql">
            <!--事务管理，type属性，默认JDBC事务-->
            <transactionManager type="JDBC"></transactionManager>
            <!--数据源信息    type属性连接池-->
            <dataSource type="POOLED">
                <!--property获取数据库连接的配置信息-->
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>

    <!--引入映射配置文件-->
    <mappers>
        <!--mapper引入指定的映射配置 resource属性执行的映射配置文件的名称-->
        <mapper resource="StudentMapper.xml"/>
    </mappers>
</configuration>

```

- StudentMapper.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="StudentMapper">
    <select id="selectAll" resultType="student">
        SELECT * FROM student
    </select>

    <select id="selectById" resultType="student" parameterType="int">
        SELECT * FROM student WHERE id = #{id}
    </select>

    <insert id="insert" parameterType="student">

```

```

    INSERT INTO student VALUES (#{id},#{name},#{age})
</insert>

<update id="update" parameterType="student">
    UPDATE student SET name = #{name}, age = #{age} WHERE id = #{id}
</update>

<delete id="delete" parameterType="student">
    DELETE FROM student WHERE id = #{id}
</delete>

</mapper>

```

- 控制层测试代码：根据 id 查询

```

@Test
public void selectById() throws Exception{
    //1.加载核心配置文件
    InputStream is = Resources.getResourceAsStream("MyBatisConfig.xml");

    //2.获取SqlSession工厂对象
    SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(is);

    //3.通过工厂对象获取SqlSession对象
    SqlSession sqlSession = ssf.openSession();

    //4.执行映射配置文件中的sql语句，并接收结果
    Student stu = sqlSession.selectOne("StudentMapper.selectById", 3);

    //5.处理结果
    System.out.println(stu);

    //6.释放资源
    sqlSession.close();
    is.close();
}

```

- 控制层测试代码：新增功能

```

@Test
public void insert() throws Exception{
    //1.加载核心配置文件
    //2.获取SqlSession工厂对象
    //3.通过工厂对象获取SqlSession对象
    SqlSession sqlSession = sqlSessionFactory.openSession(true);

    //4.执行映射配置文件中的sql语句，并接收结果
    Student stu = new Student(5, "周七", 27);
    int result = sqlSession.insert("StudentMapper.insert", stu);

    //5.提交事务
    //sqlSession.commit();

    //6.处理结果
    System.out.println(result);

    //7.释放资源
    sqlSession.close();
    is.close();
}

```

批量操作

三种方式实现批量操作：

- 标签属性：这种方式属于全局批量

```

<settings>
    <setting name="defaultExecutorType" value="BATCH"/>
</settings>

```

defaultExecutorType：配置默认的执行器

- SIMPLE 就是普通的执行器（默认，每次执行都要重新设置参数）
- REUSE 执行器会重用预处理语句（只预设置一次参数，多次执行）
- BATCH 执行器不仅重用语句还会执行批量更新（只针对修改操作）

- SqlSession 会话内批量操作：

```

public void testBatch() throws IOException{
    SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

    // 可以执行批量操作的sqlSession
    SqlSession openSession = sqlSessionFactory.openSession(ExecutorType.BATCH);
    long start = System.currentTimeMillis();
    try{
        EmployeeMapper mapper = openSession.getMapper(EmployeeMapper.class);
        for (int i = 0; i < 10000; i++) {
            mapper.addEmp(new Employee(UUID.randomUUID().toString().substring(0, 5), "b", "1"));
        }
        openSession.commit();
        long end = System.currentTimeMillis();
        // 批量: (预编译sql一次=>设置参数==>10000次==>执行1次 (类似管道))
        // 非批量: (预编译sql=设置参数=执行) ==> 10000 耗时更多
        System.out.println("执行时长: " + (end - start));
    }finally{
        openSession.close();
    }
}

```

- Spring 配置文件方式 (applicationContext.xml) :

```

<!--配置一个可以进行批量执行的sqlSession -->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg name="sqlSessionFactory" ref="sqlSessionFactoryBean"/>
    <constructor-arg name="executorType" value="BATCH"/>
</bean>

```

```

@.Autowired
private SqlSession sqlSession;

```

代理开发

代理规则

分层思想：控制层（controller）、业务层（service）、持久层（dao）

调用流程：



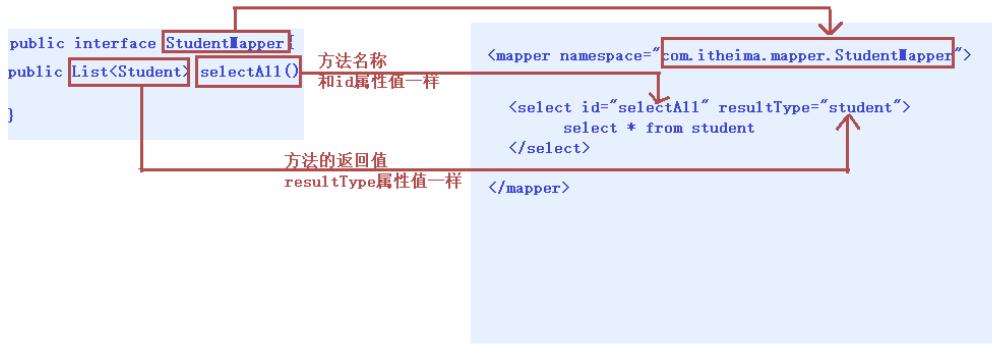
传统方式实现 DAO 层，需要写接口和实现类。采用 Mybatis 的代理开发方式实现 DAO 层的开发，只需要编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象

接口开发方式：

1. 定义接口
2. 操作数据库，MyBatis 框架根据接口，通过动态代理的方式生成代理对象，负责数据库的操作

Mapper 接口开发需要遵循以下规范：

- Mapper.xml 文件中的 namespace 与 DAO 层 mapper 接口的全类名相同
- Mapper.xml 文件中的增删改查标签的 id 属性和 DAO 层 Mapper 接口方法名相同
- Mapper.xml 文件中的增删改查标签的 parameterType 属性和 DAO 层 Mapper 接口方法的参数相同
- Mapper.xml 文件中的增删改查标签的 resultType 属性和 DAO 层 Mapper 接口方法的返回值相同



实现原理

通过动态代理开发模式，只编写一个接口不写实现类，通过 `getMapper()` 方法最终获取到 `MapperProxy` 代理对象，而这个代理对象是 MyBatis 使用了 JDK 的动态代理技术生成的

动态代理实现类对象在执行方法时最终调用了 `MapperMethod.execute()` 方法，这个方法中通过 switch case 语句根据操作类型来判断是新增、修改、删除、查询操作，最后一步回到了 MyBatis 最原生的 `SqlSession` 方式来执行增删改查

- 代码实现：

```

public Student selectById(Integer id) {
    Student stu = null;
    SqlSession sqlSession = null;
    InputStream is = null;
    try{
        //1.加载核心配置文件
        is = Resources.getResourceAsStream("MyBatisConfig.xml");

        //2.获取sqlsession工厂对象
        SqlSessionFactory s = new SqlSessionFactoryBuilder().build(is);

        //3.通过工厂对象获取sqlsession对象
        sqlSession = s.openSession(true);

        //4.获取StudentMapper接口的实现类对象
        StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);

        //5.通过实现类对象调用方法，接收结果
        stu = mapper.selectById(id);
    } catch (Exception e) {
        e.getMessage();
    } finally {
        //6.释放资源
        if(sqlSession != null) {
            sqlSession.close();
        }
        if(is != null) {
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    //7.返回结果
    return stu;
}

```

结果映射

相关标签

- ：返回结果映射对象类型，和对应方法的返回值类型保持一致，但是如果返回值是 List 则和其泛型保持一致
- ：返回一条记录的 Map，key 是列名，value 是对应的值，用来配置字段和对象属性的映射关系标签，结果映射（和 resultType 二选一）
 - id 属性：唯一标识
 - type 属性：实体对象类型
 - autoMapping 属性：结果自动映射

内的核心配置文件标签：

- ：配置主键映射关系标签
- ：配置非主键映射关系标签
 - column 属性：表中字段名称
 - property 属性：实体对象变量名称
- ：配置被包含单个对象的映射关系标签，嵌套封装结果集（多对一、一对多）
 - property 属性：被包含对象的变量名，要进行映射的属性名
 - javaType 属性：被包含对象的数据类型，要进行映射的属性的类型（Java 中的 Bean 类）
 - select 属性：加载复杂类型属性的映射语句的 ID，会从 column 属性指定的列中检索数据，作为参数传递给目标 select 语句
- ：配置被包含集合对象的映射关系标签，嵌套封装结果集（一对多、多对多）
 - property 属性：被包含集合对象的变量名
 - ofType 属性：集合中保存的对象数据类型
- ：鉴别器，用来判断某列的值，根据得到某列的不同值做出不同自定义的封装行为

自定义封装规则可以将数据库中比较复杂的数据类型映射为 JavaBean 中的属性

嵌套查询

子查询：

```
public class Blog {  
    private int id;  
    private String msg;  
    private Author author;  
    // set + get  
}
```

```
<resultMap id="blogResult" type="Blog" autoMapping = "true">  
    <association property="author" column="author_id" javaType="Author" select="selectAuthor"/>  
</resultMap>  
  
<select id="selectBlog" resultMap="blogResult">  
    SELECT * FROM BLOG WHERE ID = #{id}  
</select>  
  
<select id="selectAuthor" resultType="Author">  
    SELECT * FROM AUTHOR WHERE ID = #{id}  
</select>
```

循环引用：通过缓存解决

```
<resultMap id="blogResult" type="Blog" autoMapping = "true">  
    <id column="id" property="id"/>  
    <collection property="comment" ofType="Comment">  
        <association property="blog" javaType="Blog" resultMap="blogResult"/><!--y-->  
    </collection>  
</resultMap>
```

多表查询

一对一

一对一实现：

- 数据准备

```
CREATE TABLE person(  
    id INT PRIMARY KEY AUTO_INCREMENT,
```

```

        name VARCHAR(20),
        age INT
    );
    INSERT INTO person VALUES (NULL,'张三',23),(NULL,'李四',24),(NULL,'王五',25);

CREATE TABLE card(
    id INT PRIMARY KEY AUTO_INCREMENT,
    number VARCHAR(30),
    pid INT,
    CONSTRAINT cp_fk FOREIGN KEY (pid) REFERENCES person(id)
);
    INSERT INTO card VALUES (NULL,'12345',1),(NULL,'23456',2),(NULL,'34567',3);

```

- bean类

```

public class Card {
    private Integer id;      //主键id
    private String number;   //身份证号
    private Person p;        //所属人的对象
    .....
}

public class Person {
    private Integer id;      //主键id
    private String name;     //人的姓名
    private Integer age;     //人的年龄
}

```

- 配置文件 OneToOneMapper.xml, MyBatisConfig.xml 需要引入 (可以把 bean 包下起别名)

```

<?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="OneToOneMapper">

    <!--配置字段和实体对象属性的映射关系-->
    <resultMap id="oneToOne" type="card">
        <!--column 表中字段名称, property 实体对象变量名称-->
        <id column="cid" property="id" />
        <result column="number" property="number" />
        <!--
            association: 配置被包含对象的映射关系
            property: 被包含对象的变量名
            javaType: 被包含对象的数据类型
        -->
        <association property="p" javaType="bean.Person">
            <id column="pid" property="id" />
            <result column="name" property="name" />
            <result column="age" property="age" />
        </association>
    </resultMap>

    <select id="selectAll" resultMap="oneToOne"> <!--SQL-->
        SELECT c.id cid,number,pid,NAME,age FROM card c,`person` p WHERE c.pid=p.id
    </select>
</mapper>

```

- 核心配置文件 MyBatisConfig.xml

```

<!-- mappers引入映射配置文件 -->
<mappers>
    <mapper resource="one_to_one/OneToOneMapper.xml"/>
    <mapper resource="one_to_many/OneToManyMapper.xml"/>
    <mapper resource="many_to_many/ManyToManyMapper.xml"/>
</mappers>

```

- 测试类

```

public class Test01 {
    @Test
    public void selectAll() throws Exception{
        //1.加载核心配置文件
        InputStream is = Resources.getResourceAsStream("MyBatisConfig.xml");

        //2.获取sqlSession工厂对象
        SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(is);

        //3.通过工厂对象获取sqlSession对象
        SqlSession sqlSession = ssf.openSession(true);
    }
}

```

```

//4.获取OneToOneMapper接口的实现类对象
OneToOneMapper mapper = sqlSession.getMapper(OneToOneMapper.class);

//5.调用实现类的方法，接收结果
List<Card> list = mapper.selectAll();

//6.处理结果
for (Card c : list) {
    System.out.println(c);
}

//7.释放资源
sqlSession.close();
is.close();
}
}

```

一对多

一对多实现：

- 数据准备

```

CREATE TABLE classes(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(20)
);
INSERT INTO classes VALUES (NULL,'程序一班'),(NULL,'程序二班')

CREATE TABLE student(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(30),
    age INT,
    cid INT,
    CONSTRAINT cs_fk FOREIGN KEY (cid) REFERENCES classes(id)
);
INSERT INTO student VALUES (NULL,'张三',23,1),(NULL,'李四',24,1),(NULL,'王五',25,2);

```

- bean类

```

public class Classes {
    private Integer id;      //主键id
    private String name;     //班级名称
    private List<Student> students; //班级中所有学生对象
    .....
}
public class Student {
    private Integer id;      //主键id
    private String name;     //学生姓名
    private Integer age;     //学生年龄
}

```

- 映射配置文件

```

<mapper namespace="OneToManyMapper">
    <resultMap id="oneToMany" type="bean.Classes">
        <id column="cid" property="id"/>
        <result column="cname" property="name"/>

        <!--collection：配置被包含的集合对象映射关系-->
        <collection property="students" ofType="bean.Student">
            <id column="sid" property="id"/>
            <result column="sname" property="name"/>
            <result column="sage" property="age"/>
        </collection>
    </resultMap>
    <select id="selectAll" resultMap="oneToMany"> <!--SQL-->
        SELECT c.id cid,c.name cname,s.id sid,s.name sname,s.age sage FROM classes c,student s WHERE
        c.id=s.cid
    </select>
</mapper>

```

- 代码实现片段

```

//4.获取OneToManyMapper接口的实现类对象
OneToManyMapper mapper = sqlSession.getMapper(OneToManyMapper.class);

//5.调用实现类的方法，接收结果
List<Classes> classes = mapper.selectAll();

//6.处理结果
for (Classes cls : classes) {
    System.out.println(cls.getId() + "," + cls.getName());
    List<Student> students = cls.getStudents();
    for (Student student : students) {
        System.out.println("\t" + student);
    }
}

```

多对多

学生课程例子，中间表不需要 bean 实体类

- 数据准备

```

CREATE TABLE course(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(20)
);
INSERT INTO course VALUES (NULL,'语文'),(NULL,'数学');

CREATE TABLE stu_cr(
    id INT PRIMARY KEY AUTO_INCREMENT,
    sid INT,
    cid INT,
    CONSTRAINT sc_fk1 FOREIGN KEY (sid) REFERENCES student(id),
    CONSTRAINT sc_fk2 FOREIGN KEY (cid) REFERENCES course(id)
);
INSERT INTO stu_cr VALUES (NULL,1,1),(NULL,1,2),(NULL,2,1),(NULL,2,2);

```

- bean类

```

public class Student {
    private Integer id;      //主键id
    private String name;     //学生姓名
    private Integer age;     //学生年龄
    private List<Course> courses; // 学生所选择的课程集合
}
public class Course {
    private Integer id;      //主键id
    private String name;     //课程名称
}

```

- 配置文件

```

<mapper namespace="ManyToManyMapper">
    <resultMap id="manyToMany" type="Bean.Student">
        <id column="sid" property="id"/>
        <result column="sname" property="name"/>
        <result column="sage" property="age"/>

        <collection property="courses" ofType="Bean.Course">
            <id column="cid" property="id"/>
            <result column="cname" property="name"/>
        </collection>
    </resultMap>
    <select id="selectAll" resultMap="manyToMany"> <!--SQL-->
        SELECT sc.sid,s.name sname,s.age sage,sc.cid,c.name cname FROM student s,course c,stu_cr sc WHERE
        sc.sid=s.id AND sc.cid=c.id
    </select>
</mapper>

```

鉴别器

需求：如果查询结果是女性，则把部门信息查询出来，否则不查询；如果是男性，把 last_name 这一列的值赋值

```
<!--
    column: 指定要判断的列名
    javaType: 列值对应的java类型
-->
<discriminator javaType="string" column="gender">
    <!-- 女生 -->
    <!-- resultType不可缺少，也可以使用resultMap -->
    <case value="0" resultType="com.bean.Employee">
        <association property="dept"
            select="com.dao.DepartmentMapper.getDeptById"
            column="d_id">
        </association>
    </case>
    <!-- 男生 -->
    <case value="1" resultType="com.bean.Employee">
        <id column="id" property="id"/>
        <result column="last_name" property="lastName"/>
        <result column="gender" property="gender"/>
    </case>
</discriminator>
```

延迟加载

两种加载

立即加载：只要调用方法，马上发起查询

延迟加载：在需要用到数据时才进行加载，不需要用到数据时就不加载数据，延迟加载也称懒加载

优点：先从单表查询，需要时再从关联表去关联查询，提高数据库性能，因为查询单表要比关联查询多张表速度要快，节省资源

坏处：只有当需要用到数据时，才会进行数据库查询，这样在大批量数据查询时，查询工作也要消耗时间，所以可能造成用户等待时间变长，造成用户体验下降

核心配置文件：

标签名	描述	默认值
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载，特定关联关系中可通过设置 fetchType 属性来覆盖该项的开关状态。	false
aggressiveLazyLoading	开启时，任一方法的调用都会加载该对象的所有延迟加载属性。否则每个延迟加载属性会按需加载（参考 lazyLoadTriggerMethods）	false

```
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

assocation

分布查询：先按照身份 id 查询所属人的 id、然后根据所属人的 id 去查询人的全部信息，这就是分步查询

- 映射配置文件 OneToOneMapper.xml

一对一映射：

- column 属性表示给要调用的其它的 select 标签传入的参数
- select 属性表示调用其它的 select 标签
- fetchType="lazy" 表示延迟加载（局部配置，只有配置了这个地方才会延迟加载）

```
<mapper namespace="OneToOneMapper">
    <!--配置字段和实体对象属性的映射关系-->
    <resultMap id="oneToOne" type="card">
        <id column="id" property="id" />
        <result column="number" property="number" />
        <association property="p" javaType="bean.Person">
```

```

        column="pid"
        select="one_to_one.PersonMapper.findPersonById"
        fetchType="lazy">
        <!--需要配置新的映射文件-->
    </association>
</resultMap>

<select id="selectAll" resultMap="oneToOne">
    SELECT * FROM card <!--查询全部，负责根据条件直接全部加载-->
</select>
</mapper>

```

- PersonMapper.xml

```

<mapper namespace="one_to_one.PersonMapper">
    <select id="findPersonById" parameterType="int" resultType="person">
        SELECT * FROM person WHERE id=#{pid}
    </select>
</mapper>

```

- PersonMapper.java

```

public interface PersonMapper {
    User findPersonById(int id);
}

```

- 测试文件

```

public class Test01 {
    @Test
    public void selectAll() throws Exception{
        InputStream is = Resources.getResourceAsStream("MyBatisConfig.xml");
        SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(is);
        SqlSession sqlSession = ssf.openSession(true);
        OneToOneMapper mapper = sqlSession.getMapper(OneToOneMapper.class);
        // 调用实现类的方法，接收结果
        List<Card> list = mapper.selectAll();

        // 不能遍历，遍历就是相当于使用了该数据，需要加载，不遍历就是没有使用。

        // 释放资源
        sqlSession.close();
        is.close();
    }
}

```

collection

同样在一对多关系配置的 结点中配置延迟加载策略，结点中也有 select 属性和 column 属性

- 映射配置文件 OneToManyMapper.xml

一对多映射：

- column 是用于指定使用哪个字段的值作为条件查询
- select 是用于指定查询账户的唯一标识（账户的 dao 全限定类名加上方法名称）

```

<mapper namespace="OneToManyMapper">
    <resultMap id="oneToMany" type="bean.Classes">
        <id column="id" property="id"/>
        <result column="name" property="name"/>

        <!--collection: 配置被包含的集合对象映射关系-->
        <collection property="students" ofType="bean.Student">
            <id column="id" />
            <select column="name" property="name">
                one_to_one.StudentMapper.findStudentByCid
            </select>
        </collection>
    </resultMap>
    <select id="selectAll" resultMap="oneToMany">
        SELECT * FROM classes
    </select>
</mapper>

```

- StudentMapper.xml

```

<mapper namespace="one_to_one.StudentMapper">
    <select id="findPersonByCid" parameterType="int" resultType="student">
        SELECT * FROM person WHERE cid=#{id}
    </select>
</mapper>

```

注解开发

单表操作

注解可以简化开发操作，省略映射配置文件的编写

常用注解：

- @Select("查询的 SQL 语句")：执行查询操作注解
- @Insert("插入的 SQL 语句")：执行新增操作注解
- @Update("修改的 SQL 语句")：执行修改操作注解
- @Delete("删除的 SQL 语句")：执行删除操作注解

参数注解：

- @Param：当 SQL 语句需要多个（大于1）参数时，用来指定参数的对应规则

核心配置文件配置映射关系：

```

<mappers>
    <package name="使用了注解的Mapper接口所在包"/>
</mappers>
<!!--或者-->
<mappers>
    <mapper class="包名.Mapper名"></mapper>
</mappers>

```

基本增删改查：

- 创建 Mapper 接口

```

package mapper;
public interface StudentMapper {
    //查询全部
    @Select("SELECT * FROM student")
    public abstract List<Student> selectAll();

    //新增数据
    @Insert("INSERT INTO student VALUES (#{id},#{name},#{age})")
    public abstract Integer insert(Student student);

    //修改操作
    @Update("UPDATE student SET name=#{name},age=#{age} WHERE id=#{id}")
    public abstract Integer update(Student student);

    //删除操作
    @Delete("DELETE FROM student WHERE id=#{id}")
    public abstract Integer delete(Integer id);
}

```

- 修改 MyBatis 的核心配置文件

```

<mappers>
    <package name="mapper"/>
</mappers>

```

- bean类

```

public class Student {
    private Integer id;
    private String name;
    private Integer age;
}

```

- 测试类

```

@Test
public void selectAll() throws Exception{
    //1.加载核心配置文件
    InputStream is = Resources.getResourceAsStream("MyBatisConfig.xml");

    //2.获取SqlSession工厂对象
    SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(is);

    //3.通过工厂对象获取sqlSession对象
    SqlSession sqlSession = ssf.openSession(true);

    //4.获取StudentMapper接口的实现类对象
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);

    //5.调用实现类对象中的方法，接收结果
    List<Student> list = mapper.selectAll();

    //6.处理结果
    for (Student student : list) {
        System.out.println(student);
    }

    //7.释放资源
    sqlSession.close();
    is.close();
}

```

多表操作

相关注解

实现复杂关系映射之前我们可以在映射文件中通过配置 来实现，使用注解开发后，可以使用 @Results 注解，@Result 注解，@One 注解，@Many 注解组合完成复杂关系的配置

注解	说明
@Results	代替 标签，注解中使用单个 @Result 注解或者 @Result 集合 使用格式：@Results({ @Result(), @Result() })或@Results({ @Result() })
@Result	代替<id> 和 标签，@Result 中属性介绍： column: 数据库的列名 property: 封装类的变量名 one: 需要使用 @One 注解 (@Result(one = @One)) Many: 需要使用 @Many 注解 (@Result(many= @Many))
@One(一对一)	代替 标签，多表查询的关键，用来指定子查询返回单一对象 select: 指定调用 Mapper 接口中的某个方法 使用格式：@Result(column="", property="", one=@One(select=""))
@Many(多对一)	代替 标签，多表查询的关键，用来指定子查询返回对象集合 select: 指定调用 Mapper 接口中的某个方法 使用格式：@Result(column="", property="", many=@Many(select=""))

一对一

身份证对人

- PersonMapper 接口

```

public interface PersonMapper {
    //根据id查询
    @Select("SELECT * FROM person WHERE id=#{id}")
    public abstract Person selectById(Integer id);
}

```

- CardMapper接口

```

public interface CardMapper {
    //查询全部
    @Select("SELECT * FROM card")
    @Results({
        @Result(column = "id",property = "id"),
        @Result(column = "number",property = "number"),
    })
}

```

```

@Result(
    property = "p",          // 被包含对象的变量名
    javaType = Person.class,  // 被包含对象的实际数据类型
    column = "pid",           // 根据查询出的card表中的pid字段来查询person表
    /*
        one、@One 一对一固定写法
        select属性：指定调用哪个接口中的哪个方法
    */
    one = @One(select = "one_to_one.PersonMapper.selectById")
)
})
public abstract List<Card> selectAll();
}

```

- 测试类 (详细代码参考单表操作)

```

//1.加载核心配置文件
//2.获取sqlSession工厂对象
//3.通过工厂对象获取SqlSession对象

//4.获取StudentMapper接口的实现类对象
CardMapper mapper = sqlSession.getMapper(CardMapper.class);
//5.调用实现类对象中的方法，接收结果
List<Card> list = mapper.selectAll();

```

一对多

班级和学生

- StudentMapper接口

```

public interface StudentMapper {
    //根据cid查询student表 cid是外键约束列
    @Select("SELECT * FROM student WHERE cid=#{cid}")
    public abstract List<Student> selectByCid(Integer cid);
}

```

- ClassesMapper接口

```

public interface ClassesMapper {
    //查询全部
    @Select("SELECT * FROM classes")
    @Results({
        @Result(column = "id", property = "id"),
        @Result(column = "name", property = "name"),
        @Result(
            property = "students", //被包含对象的变量名
            javaType = List.class, //被包含对象的实际数据类型
            column = "id",           //根据id字段查询student表
            many = @Many(select = "one_to_many.StudentMapper.selectByCid")
        )
    })
    public abstract List<Classes> selectAll();
}

```

- 测试类

```

//4.获取StudentMapper接口的实现类对象
ClassesMapper mapper = sqlSession.getMapper(ClassesMapper.class);
//5.调用实现类对象中的方法，接收结果
List<Classes> classes = mapper.selectAll();

```

多对多

学生和课程

- SQL 查询语句

```

SELECT DISTINCT s.id,s.name,s.age FROM student s,stu_cr sc WHERE sc.sid=s.id
SELECT c.id,c.name FROM stu_cr sc,course c WHERE sc.cid=c.id AND sc.sid=#{id}

```

- CourseMapper 接口

```

public interface CourseMapper {
    //根据学生id查询所选课程
    @Select("SELECT c.id,c.name FROM stu_cr sc,course c WHERE sc.cid=c.id AND sc.sid=#{id}")
    public abstract List<Course> selectBySid(Integer id);
}

```

- StudentMapper 接口

```

public interface StudentMapper {
    //查询全部
    @Select("SELECT DISTINCT s.id,s.name,s.age FROM student s,stu_cr sc WHERE sc.sid=s.id")
    @Results({
        @Result(column = "id",property = "id"),
        @Result(column = "name",property = "name"),
        @Result(column = "age",property = "age"),
        @Result(
            property = "courses",      //被包含对象的变量名
            javaType = List.class,    //被包含对象的实际数据类型
            column = "id",           //根据查询出的student表中的id字段查询中间表和课程表
            many = @Many(select = "many_to_many.CourseMapper.selectBySid")
        )
    })
    public abstract List<Student> selectAll();
}

```

- 测试类

```

//4.获取StudentMapper接口的实现类对象
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
//5.调用实现类对象中的方法，接收结果
List<Student> students = mapper.selectAll();

```

缓存机制

缓存概述

缓存：缓存就是一块内存空间，保存临时数据

作用：将数据源（数据库或者文件）中的数据读取出来放到缓存中，再次获取时直接从缓存中获取，可以减少和数据库交互的次数，提升程序的性能

缓存适用：

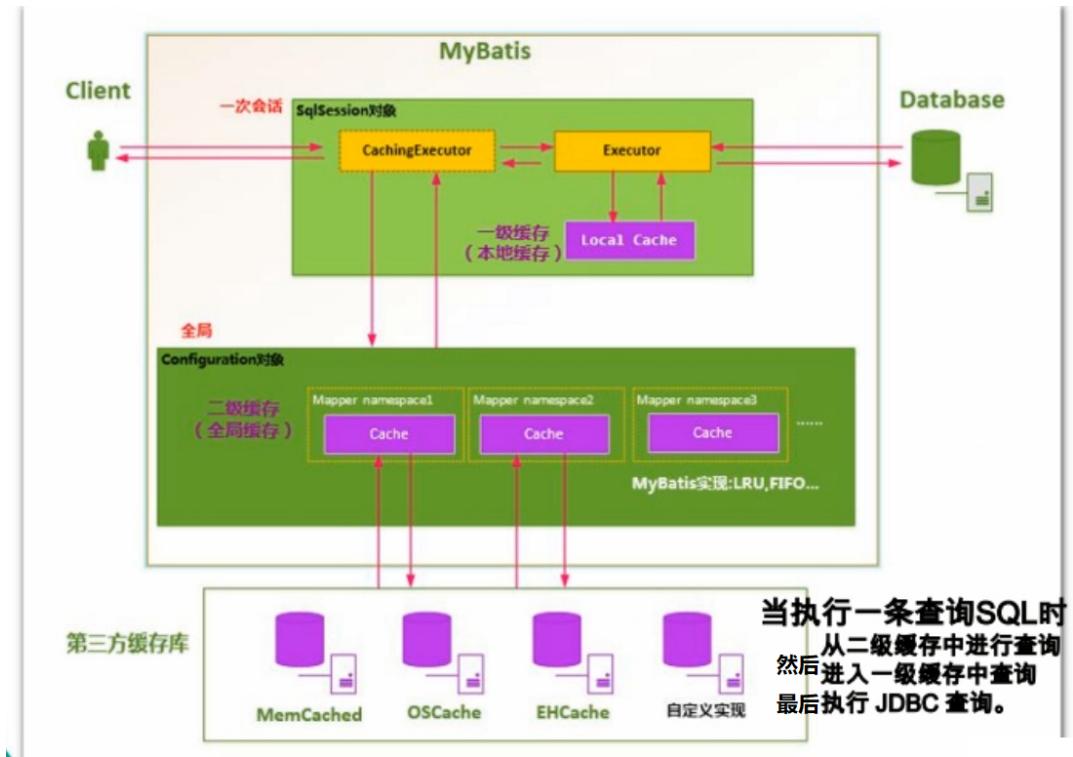
- 适用于缓存的：经常查询但不经常修改的，数据的正确与否对最终结果影响不大的
- 不适用缓存的：经常改变的数据，敏感数据（例如：股市的牌价，银行的汇率，银行卡里面的钱）等等

缓存类别：

- 一级缓存：SqlSession 级别的缓存，又叫本地会话缓存，自带的（不需要配置），一级缓存的生命周期与 SqlSession 一致。在操作数据库时需要构造 SqlSession 对象，在对象中有一个数据结构（HashMap）用于存储缓存数据，不同的 SqlSession 之间的缓存数据区域是互相不影响的
- 二级缓存：mapper（namespace）级别的缓存，二级缓存的使用，需要手动开启（需要配置）。多个 SqlSession 去操作同一个 Mapper 的 SQL 可以共用二级缓存，二级缓存是跨 SqlSession 的

开启缓存：配置核心配置文件中 `cache` 标签

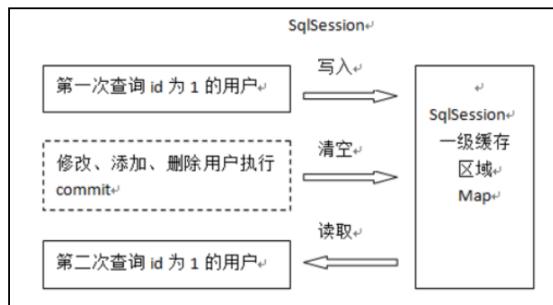
- `cacheEnabled: true` 表示全局性地开启所有映射器配置文件中已配置的任何缓存，默认 `true`



参考文章: <https://www.cnblogs.com/yocean/p/7342498.html>

一级缓存

一级缓存是 SqlSession 级别的缓存



工作流程: 第一次发起查询用户 id 为 1 的用户信息, 先去找缓存中是否有 id 为 1 的用户信息, 如果没有, 从数据库查询用户信息, 得到用户信息, 将用户信息存储到一级缓存中; 第二次发起查询用户 id 为 1 的用户信息, 先去找缓存中是否有 id 为 1 的用户信息, 缓存中有, 直接从缓存中获取用户信息。

一级缓存的失效:

- SqlSession 不同
- SqlSession 相同, 查询条件不同时 (还未缓存该数据)
- SqlSession 相同, 手动清除了级缓存, 调用 `sqlsession.clearCache()`
- SqlSession 相同, 执行 commit 操作或者执行插入、更新、删除, 清空 SqlSession 中的一级缓存, 这样做的目的为了让缓存中存储的是最新的信息, 避免脏读

Spring 整合 MyBatis 后, 一级缓存作用:

- 未开启事务的情况, 每次查询 Spring 都会创建新的 SqlSession, 因此一级缓存失效
- 开启事务的情况, Spring 使用 ThreadLocal 获取当前资源绑定同一个 SqlSession, 因此此时一级缓存是有效的

测试一级缓存存在

```
public void testFirstLevelCache(){
    //1. 获取sqlSession对象
    SqlSession sqlSession = SqlSessionFactoryUtils.openSession();
    //2. 通过sqlSession对象获取UserDao接口的代理对象
    UserDao userDao1 = sqlSession.getMapper(UserDao.class);
    //3. 调用UserDao接口的代理对象的findById方法获取信息
    User user1 = userDao1.findById(1);
    System.out.println(user1);
```

```

//sqlSession.clearCache() 清空缓存

UserDao userDao2 = sqlSession.getMapper(UserDao.class);
User user = userDao.findById(1);
System.out.println(user2);

//4. 测试两次结果是否一样
System.out.println(user1 == user2); //true

//5. 提交事务关闭资源
SqlSessionFactoryUtils.commitAndClose(sqlSession);
}

```

二级缓存

基本介绍

二级缓存是 mapper 的缓存，只要是同一个命名空间 (namespace) 的 SqlSession 就共享二级缓存的内容，并且可以操作二级缓存

作用：作用范围是整个应用，可以跨线程使用，适合缓存一些修改较少的数据

工作流程：一个会话查询数据，这个数据就会被放在当前会话的一级缓存中，如果会话关闭或提交一级缓存中的数据会保存到二级缓存

二级缓存的基本使用：

- 在 MyBatisConfig.xml 文件开启二级缓存，**cacheEnabled** 默认值为 true，所以这一步可以省略不配置

```

<!--配置开启二级缓存-->
<settings>
    <setting name="cacheEnabled" value="true"/>
</settings>

```

- 配置 Mapper 映射文件

`<cache>` 标签表示当前这个 mapper 映射将使用二级缓存，区分的标准就看 mapper 的 namespace 值

```

<mapper namespace="dao.UserDao">
    <!--开启user支持二级缓存-->
    <cache eviction="FIFO" flushInterval="6000" readOnly="" size="1024"/>
    <cache></cache> <!--则表示所有属性使用默认值-->
</mapper>

```

eviction (清除策略) :

- LRU - 最近最少使用：移除最长时间不被使用的对象，默认
- FIFO - 先进先出：按对象进入缓存的顺序来移除它们
- SOFT - 软引用：基于垃圾回收器状态和软引用规则移除对象
- WEAK - 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象

flushInterval (刷新间隔) : 可以设置为任意的正整数，默认情况是不设置，也就是没有刷新间隔，缓存仅仅会在调用语句时刷新

size (引用数目) : 缓存存放多少元素，默认值是 1024

readOnly (只读) : 可以被设置为 true 或 false

- 只读的缓存会给所有调用者返回缓存对象的相同实例，因此这些对象不能被修改，促进了性能提升
- 可读写的缓存会（通过序列化）返回缓存对象的拷贝，速度上会慢一些，但是更安全，因此默认值是 false

type: 指定自定义缓存的全类名，实现 Cache 接口即可

- 要进行二级缓存的类必须实现 java.io.Serializable 接口，可以使用序列化方式来保存对象。

```

public class User implements Serializable{}

```

相关属性

- select 标签的 useCache 属性

映射文件中的 `<select>` 标签中设置 `useCache="true"` 代表当前 statement 要使用二级缓存（默认）

注意：如果每次查询都需要最新的数据 sql，要设置成 `useCache=false`，禁用二级缓存

```
<select id="findAll" resultType="user" useCache="true">
    select * from user
</select>
```

2. 每个增删改标签都有 flushCache 属性，默认为 true，代表在执行增删改之后就会清除一、二级缓存，保证缓存的一致性；而查询标签

默认值为 false，所以查询不会清空缓存

3. localCacheScope：本地缓存作用域，中的配置项，默认值为 SESSION，当前会话的所有数据保存在会话缓存中，设置为 STATEMENT 禁用一级缓存

源码解析

事务提交二级缓存才生效：DefaultSqlSession 调用 commit() 时会回调 executor.commit()

- CachingExecutor#query(): 执行查询方法，查询出的数据会先放入 entriesToAddOnCommit 集合暂存

```
// 从二级缓存中获取数据，获取不到去一级缓存获取
List<E> list = (List<E>) tcm.getObject(cache, key);
if (list == null) {
    // 回调 BaseExecutor#query
    list = delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
    // 将数据放入 entriesToAddOnCommit 集合暂存，此时还没放入二级缓存
    tcm.putObject(cache, key, list);
}
```

- commit(): 事务提交，**清空一级缓存，放入二级缓存**，二级缓存使用 TransactionalCacheManager (tcm) 管理

```
public void commit(boolean required) throws SQLException {
    // 首先调用 BaseExecutor#commit 方法，【清空一级缓存】
    delegate.commit(required);
    tcm.commit();
}
```

- TransactionalCacheManager#commit: 查询出的数据放入二级缓存

```
public void commit() {
    // 获取所有的缓存事务，挨着进行提交
    for (TransactionalCache txCache : transactionalCaches.values()) {
        txCache.commit();
    }
}
```

```
public void commit() {
    if (clearOnCommit) {
        delegate.clear();
    }
    // 将 entriesToAddOnCommit 中的数据放入二级缓存
    flushPendingEntries();
    // 清空相关集合
    reset();
}
```

```
private void flushPendingEntries() {
    for (Map.Entry<Object, Object> entry : entriesToAddOnCommit.entrySet()) {
        // 将数据放入二级缓存
        delegate.putObject(entry.getKey(), entry.getValue());
    }
}
```

增删改操作会清空缓存：

- update(): CachingExecutor 的更新操作

```
public int update(MappedStatement ms, Object parameterObject) throws SQLException {
    flushCacheIfRequired(ms);
    // 回调 BaseExecutor#update 方法，也会清空一级缓存
    return delegate.update(ms, parameterObject);
}
```

- flushCacheIfRequired(): 判断是否需要清空二级缓存

```
private void flushCacheIfRequired(MappedStatement ms) {
    Cache cache = ms.getCache();
    // 判断二级缓存是否存在，然后判断标签的 flushCache 的值，增删改操作的 flushCache 属性默认为 true
    if (cache != null && ms.isFlushCacheRequired()) {
        // 清空二级缓存
        tcm.clear(cache);
    }
}
```

自定义

自定义缓存

```
<cache type="com.domain.something.MyCustomCache"/>
```

type 属性指定的类必须实现 org.apache.ibatis.cache.Cache 接口，且提供一个接受 String 参数作为 id 的构造器

```
public interface Cache {
    String getId();
    int getSize();
    void putObject(Object key, Object value);
    Object getObject(Object key);
    boolean hasKey(Object key);
    Object removeObject(Object key);
    void clear();
}
```

缓存的配置，只需要在缓存实现中添加公有的 JavaBean 属性，然后通过 cache 元素传递属性值，例如在缓存实现上调用一个名为 setCacheFile(String file) 的方法：

```
<cache type="com.domain.something.MyCustomCache">
    <property name="cacheFile" value="/tmp/my-custom-cache.tmp"/>
</cache>
```

- 可以使用所有简单类型作为 JavaBean 属性的类型，MyBatis 会进行转换。
- 可以使用占位符（如 \${cache.file}），以便替换成为配置文件属性中定义的值

MyBatis 支持在所有属性设置完毕之后，调用一个初始化方法，如果想要使用这个特性，可以在自定义缓存类里实现 org.apache.ibatis.builder.InitializingObject 接口

```
public interface InitializingObject {
    void initialize() throws Exception;
}
```

注意：对缓存的配置（如清除策略、可读或可读写等），不能应用于自定义缓存

对某一命名空间的语句，只会使用该命名空间的缓存进行缓存或刷新，在多个命名空间中共享相同的缓存配置和实例，可以使用 cache-ref 元素来引用另一个缓存

```
<cache-ref namespace="com.someone.application.data.SomeMapper"/>
```

构造语句

动态 SQL

基本介绍

动态 SQL 是 MyBatis 强大特性之一，逻辑复杂时，MyBatis 映射配置文件中，SQL 是动态变化的，所以引入动态 SQL 简化拼装 SQL 的操作

DynamicSQL 包含的标签：

- if
- where
- set
- choose (when、otherwise)
- trim
- foreach

各个标签都可以进行灵活嵌套和组合

OGNL: Object Graphic Navigation Language (对象图导航语言) , 用于对数据进行访问

参考文章: <https://www.cnblogs.com/ysoccean/p/7289529.html>

where

: 条件标签, 有动态条件则使用该标签代替 WHERE 关键字, 封装查询条件

作用: 如果标签返回的内容是以 AND 或 OR 开头的, 标签内会剔除掉

表结构:

栏位	索引	外键	触发器	选项	注释	SQL 预览
名						
id						1
username						
sex						
birthday						
address						

if

基本格式:

```
<if test="条件判断">  
    查询条件拼接  
</if>
```

我们根据实体类的不同取值, 使用不同的 SQL 语句来进行查询。比如在 id 如果不为空时可以根据 id 查询, 如果username 不为空时还要加入用户名作为条件, 这种情况在我们的多条件组合查询中经常会碰到。

- UserMapper.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="mapper.UserMapper">  
    <select id="selectCondition" resultType="user" parameterType="user">  
        SELECT * FROM user  
        <where>  
            <if test="id != null ">  
                id = #{id}  
            </if>  
            <if test="username != null ">  
                AND username = #{username}  
            </if>  
            <if test="sex != null ">  
                AND sex = #{sex}  
            </if>  
        </where>  
    </select>  
</mapper>
```

- MyBatisConfig.xml, 引入映射配置文件

```
<mappers>  
    <!--mapper引入指定的映射配置 resource属性执行的映射配置文件的名称-->  
    <mapper resource="UserMapper.xml"/>  
</mappers>
```

- DAO 层 Mapper 接口

```

public interface UserMapper {
    //多条件查询
    public abstract List<User> selectCondition(Student stu);
}

```

- 实现类

```

public class DynamicTest {
    @Test
    public void selectCondition() throws Exception{
        //1.加载核心配置文件
        InputStream is = Resources.getResourceAsStream("MyBatisConfig.xml");

        //2.获取sqlSession工厂对象
        SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(is);

        //3.通过工厂对象获取SqlSession对象
        SqlSession sqlSession = ssf.openSession(true);

        //4.获取StudentMapper接口的实现类对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);

        User user = new User();
        user.setId(2);
        user.setUsername("李四");
        //user.setSex(男); AND 后会自动剔除

        //5.调用实现类的方法，接收结果
        List<Student> list = mapper.selectCondition(user);

        //6.处理结果
        for (User user : list) {
            System.out.println(user);
        }

        //7.释放资源
        sqlSession.close();
        is.close();
    }
}

```

set

: 进行更新操作的时候，含有 set 关键词，使用该标签

```

<!-- 根据 id 更新 user 表的数据 -->
<update id="updateUserById" parameterType="com.ys.po.User">
    UPDATE user u
    <set>
        <if test="username != null and username != ''">
            u.username = #{username},
        </if>
        <if test="sex != null and sex != ''">
            u.sex = #{sex}
        </if>
    </set>
    WHERE id=#{id}
</update>

```

- 如果第一个条件 username 为空，那么 sql 语句为： update user u set u.sex=? where id=?
- 如果第一个条件不为空，那么 sql 语句为： update user u set u.username = ?, u.sex = ? where id=?

choose

假如不想用到所有的查询条件，只要查询条件有一个满足即可，使用 choose 标签可以解决此类问题，类似于 Java 的 switch 语句

标签： ,

```

<select id="selectUserByChoose" resultType="user" parameterType="user">
    SELECT * FROM user
    <where>

```

```

<choose>
    <when test="id != '' and id != null">
        id=#{id}
    </when>
    <when test="username != '' and username != null">
        AND username=#{username}
    </when>
    <otherwise>
        AND sex=#{sex}
    </otherwise>
</choose>
</where>
</select>

```

有三个条件，id、username、sex，只能选择一个作为查询条件

- 如果 id 不为空，那么查询语句为：select * from user where id=?
- 如果 id 为空，那么看 username 是否为空
 - 如果不为空，那么语句为：select * from user where username=?
 - 如果 username 空，那么查询语句为 select * from user where sex=?

trim

trim 标记是一个格式化的标记，可以完成 set 或者是 where 标记的功能，自定义字符串截取

- prefix: 给拼串后的整个字符串加一个前缀，trim 标签体中是整个字符串拼串后的结果
- prefixOverrides: 去掉整个字符串前面多余的字符
- suffix: 给拼串后的整个字符串加一个后缀
- suffixOverrides: 去掉整个字符串后面多余的字符

改写 if + where 语句：

```

<select id="selectUserByUsernameAndSex" resultType="user" parameterType="com.ys.po.User">
    SELECT * FROM user
    <trim prefix="where" prefixOverrides="and | or">
        <if test="username != null">
            AND username=#{username}
        </if>
        <if test="sex != null">
            AND sex=#{sex}
        </if>
    </trim>
</select>

```

改写 if + set 语句：

```

<!-- 根据 id 更新 user 表的数据 -->
<update id="updateUserById" parameterType="com.ys.po.User">
    UPDATE user u
    <trim prefix="set" suffixOverrides=",">
        <if test="username != null and username != ''">
            u.username = #{username},
        </if>
        <if test="sex != null and sex != ''">
            u.sex = #{sex},
        </if>
    </trim>
    WHERE id=#{id}
</update>

```

foreach

基本格式：

```

<foreach>: 循环遍历标签。适用于多个参数或者的关系。
<foreach collection="" open="" close="" item="" separator="">
    获取参数
</foreach>

```

属性：

- collection: 参数容器类型，(list-集合， array-数组)

- open: 开始的 SQL 语句
- close: 结束的 SQL 语句
- item: 参数变量名
- separator: 分隔符

需求: 循环执行 sql 的拼接操作, `SELECT * FROM user WHERE id IN (1,2,5)`

- UserMapper.xml片段

```
<select id="selectByIds" resultType="user" parameterType="list">
    SELECT * FROM student
    <where>
        <foreach collection="list" open="id IN(" close=")" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

- 测试代码片段

```
//4.获取StudentMapper接口的实现类对象
UserMapper mapper = sqlSession.getMapper(UserMapper.class);

List<Integer> ids = new ArrayList<>();
Collections.addAll(list, 1, 2);
//5.调用实现类的方法,接收结果
List<User> list = mapper.selectByIds(ids);

for (User user : list) {
    System.out.println(user);
}
```

SQL片段

将一些重复性的 SQL 语句进行抽取, 以达到复用的效果

格式:

```
<sql id="片段唯一标识">抽取的SQL语句</sql>      <!--抽取标签-->
<include refid="片段唯一标识"/>          <!--引入标签-->
```

使用:

```
<sql id="select">SELECT * FROM user</sql>

<select id="selectByIds" resultType="user" parameterType="list">
    <include refid="select"/>
    <where>
        <foreach collection="list" open="id IN(" close=")" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

逆向工程

MyBatis 逆向工程, 可以针对单表自动生成 MyBatis 执行所需要的代码 (mapper.java、mapper.xml、pojo...)

generatorConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!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="testTables" targetRuntime="MyBatis3">
        <commentGenerator>
            <!-- 是否去除自动生成的注释 true: 是 : false: 否 -->
            <property name="suppressAllComments" value="true" />
        </commentGenerator>
    </context>
</generatorConfiguration>
```

```

</commentGenerator>
<!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->
<jdbcConnection driverClass="com.mysql.jdbc.Driver"
    connectionURL="jdbc:mysql://localhost:3306/mybatisrelation" userId="root"
    password="root">
</jdbcConnection>

<!-- 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>
    <property name="forceBigDecimals" value="false" />
</javaTypeResolver>

<!-- targetProject:生成PO类的位置！！ -->
<javaModelGenerator targetPackage="com.ys.po"
    targetProject=".src">
    <!-- enableSubPackages:是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="false" />
    <!-- 从数据库返回的值被清理前后的空格 -->
    <property name="trimStrings" value="true" />
</javaModelGenerator>
<!-- targetProject:mapper映射文件生成的位置！！ -->
<sqlMapGenerator targetPackage="com.ys.mapper"
    targetProject=".src">
    <property name="enableSubPackages" value="false" />
</sqlMapGenerator>
<!-- targetPackage: mapper接口生成的位置，重要！！ -->
<javaClientGenerator type="XMLMAPPER"
    targetPackage="com.ys.mapper"
    targetProject=".src">
    <property name="enableSubPackages" value="false" />
</javaClientGenerator>
<!-- 指定数据库表，要生成哪些表，就写哪些表，要和数据库中对应，不能写错！ -->
<table tableName="items"></table>
<table tableName="orders"></table>
<table tableName="orderdetail"></table>
<table tableName="user"></table>
</context>
</generatorConfiguration>

```

生成代码:

```

public void testGenerator() throws Exception{
    List<String> warnings = new ArrayList<String>();
    boolean overwrite = true;
    //指向逆向工程配置文件
    File configFile = new File(GeneratorTest.class.
        getResource("/generatorConfig.xml").getFile());
    ConfigurationParser cp = new ConfigurationParser(warnings);
    Configuration config = cp.parseConfiguration(configFile);
    DefaultShellCallback callback = new DefaultShellCallback(overwrite);
    MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,
        callback, warnings);
    myBatisGenerator.generate(null);
}

```

参考文章: <https://www.cnblogs.com/yocean/p/7360409.html>

构建 SQL

基础语法

MyBatis 提供了 org.apache.ibatis.jdbc.SQL 功能类，专门用于构建 SQL 语句

方法	说明
SELECT(String... columns)	根据字段拼接查询语句
FROM(String... tables)	根据表名拼接语句
WHERE(String... conditions)	根据条件拼接语句
INSERT_INTO(String tableName)	根据表名拼接新增语句
INTO_VALUES(String... values)	根据值拼接新增语句
UPDATE(String table)	根据表名拼接修改语句
DELETE_FROM(String table)	根据表名拼接删除语句

增删改查注解：

- @SelectProvider: 生成查询用的 SQL 语句
- @InsertProvider: 生成新增用的 SQL 语句
- @UpdateProvider: 生成修改用的 SQL 语句注解
- @DeleteProvider: 生成删除用的 SQL 语句注解。
 - type 属性: 生成 SQL 语句功能类对象
 - method 属性: 指定调用方法

基本操作

- MyBatisConfig.xml 配置

```
<!-- mappers 引入映射配置文件 -->
<mappers>
  <package name="mapper"/>
</mappers>
```

- Mapper 类

```
public interface StudentMapper {
    // 查询全部
    @SelectProvider(type = ReturnSql.class, method = "getSelectAll")
    public abstract List<Student> selectAll();

    // 新增数据
    @InsertProvider(type = ReturnSql.class, method = "getInsert")
    public abstract Integer insert(Student student);

    // 修改操作
    @UpdateProvider(type = ReturnSql.class, method = "getUpdate")
    public abstract Integer update(Student student);

    // 删除操作
    @DeleteProvider(type = ReturnSql.class, method = "getDelete")
    public abstract Integer delete(Integer id);
}
```

- ReturnSQL 类

```
public class ReturnSql {
    // 定义方法，返回查询的sql语句
    public String getSelectAll() {
        return new SQL() {
            {
                SELECT("*");
                FROM("student");
            }
            .toString();
        }
    }

    // 定义方法，返回新增的sql语句
    public String getInsert(Student stu) {
        return new SQL() {
            {
                INSERT_INTO("student");
                INTO_VALUES("#{id},#{name},#{age}");
            }
            .toString();
        }
    }
}
```

```

}

//定义方法，返回修改的sql语句
public String getUpdate(Student stu) {
    return new SQL() {
        {
            UPDATE("student");
            SET("name=#{name}", "age=#{age}");
            WHERE("id=#{id}");
        }
        .toString();
    }
}

//定义方法，返回删除的sql语句
public String getDelete(Integer id) {
    return new SQL() {
        {
            DELETE_FROM("student");
            WHERE("id=#{id}");
        }
        .toString();
    }
}
}

```

- 功能实现类

```

public class sqlTest {
    @Test //查询全部
    public void selectAll() throws Exception{
        //1.加载核心配置文件
        InputStream is = Resources.getResourceAsStream("MyBatisConfig.xml");

        //2.获取SqlSession工厂对象
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);

        //3.通过工厂对象获取SqlSession对象
        SqlSession sqlSession = sqlSessionFactory.openSession(true);

        //4.获取StudentMapper接口的实现类对象
        StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);

        //5.调用实现类对象中的方法，接收结果
        List<Student> list = mapper.selectAll();

        //6.处理结果
        for (Student student : list) {
            System.out.println(student);
        }

        //7.释放资源
        sqlSession.close();
        is.close();
    }

    @Test //新增
    public void insert() throws Exception{
        //1 2 3 4获取StudentMapper接口的实现类对象
        StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);

        //5.调用实现类对象中的方法，接收结果 ->6 7
        Student stu = new Student(4, "赵六", 26);
        Integer result = mapper.insert(stu);
    }

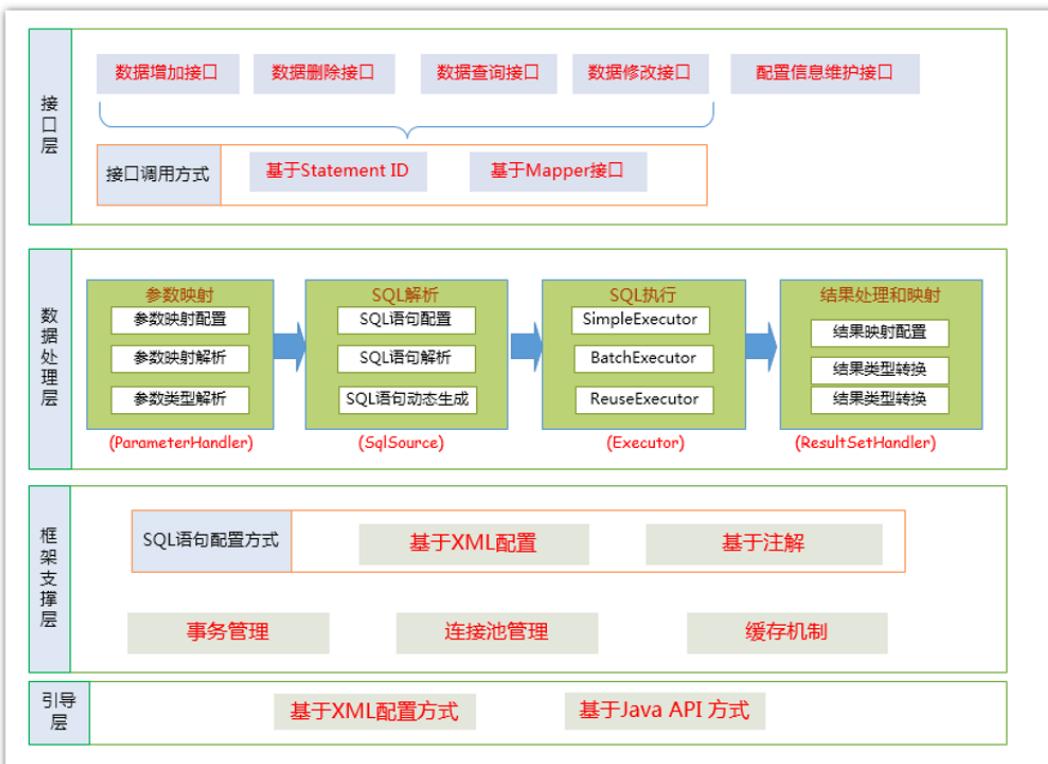
    @Test //修改
    public void update() throws Exception{
        //1 2 3 4 5调用实现类对象中的方法，接收结果 ->6 7
        Student stu = new Student(4, "赵六wq", 36);
        Integer result = mapper.update(stu);
    }

    @Test //删除
    public void delete() throws Exception{
        //1 2 3 4 5 6 7
        Integer result = mapper.delete(4);
    }
}

```

运行原理

运行机制



MyBatis 运行过程:

- 加载 MyBatis 全局配置文件，通过 XPath 方式解析 XML 配置文件，首先解析核心配置文件，**标签中配置属性项有 defaultExecutorType**，用来配置指定 Executor 类型，将配置文件的信息填充到 Configuration 对象。最后解析映射器配置的映射文件，并构建 **MappedStatement** 对象填充至 Configuration，将解析后的映射器添加到 mapperRegistry 中，用于获取代理
- 创建一个 DefaultSqlSession 对象，根据参数创建指定类型的 Executor，二级缓存默认开启，把 Executor 包装成缓存执行器
- DefaultSqlSession 调用 getMapper()，通过 JDK 动态代理获取 Mapper 接口的代理对象 MapperProxy
- 执行 SQL 语句：
 - MapperProxy.invoke() 执行代理方法，通过 MapperMethod#execute 判断执行的是增删改查中的哪个方法
 - 查询方法调用 sqlSession.selectOne()，从 Configuration 中获取执行者对象 MappedStatement，然后 Executor 调用 executor.query 开始执行查询方法
 - 首先通过 CachingExecutor 去二级缓存查询，查询不到去一级缓存查询，最后去数据库查询并放入一级缓存
 - Configuration 对象根据

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    try {
        // 当前方法是否是属于 Object 类中的方法
        if (Object.class.equals(method.getDeclaringClass())) {
            return method.invoke(this, args);
        // 当前方法是否是默认方法
        } else if (isDefaultMethod(method)) {
            return invokeDefaultMethod(proxy, method, args);
        }
    } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
    }
    // 包装成一个 MapperMethod 对象并初始化该对象
    final MapperMethod mapperMethod = cachedMapperMethod(method);
}
```

```
// 【根据 switch-case 判断使用的什么类型的 SQL 进行逻辑处理】，此处分析查询语句的查询操作
return mapperMethod.execute(sqlSession, args);
}
```

sqlSession.selectOne(String, Object): 查询数据

```
public Object execute(SqlSession sqlSession, Object[] args) {
    //.....
    // 解析传入的参数
    Object param = method.convertArgsToSqlCommandParam(args);
    result = sqlSession.selectOne(command.getName(), param);
}
// DefaultSqlSession.selectList(String, Object)
public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
    // 获取执行者对象
    MappedStatement ms = configuration.getMappedStatement(statement);
    // 开始执行查询语句，参数通过 wrapCollection() 包装成集合类
    return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
}
```

Executor#query():

- CachingExecutor.query(): 先执行 CachingExecutor 去二级缓存获取数据

```
public class CachingExecutor implements Executor {
    private final Executor delegate;           // 包装了 BaseExecutor, 二级缓存不存在数据调用 BaseExecutor 查询
}
```

- MappedStatement.getBoundSql(parameterObject): 把 parameterObject 封装成 BoundSql
构造函数中有: this.parameterObject = parameterObject
 - boundSql= BoundSql (id=220)
 - additionalParameters= HashMap<K,V> (id=223)
 - metaParameters= MetaObject (id=224)
 - parameterMappings= ArrayList<E> (id=226)
 - parameterObject= Integer (id=96)
 - sql= "select id,last_name lastName,email,gender from tbl_employee where id = ?" (id=228)

- CachingExecutor.createCacheKey(): 创建缓存对象
- ms.getCache(): 获取二级缓存
- tcm.getObject(cache, key): 尝试从二级缓存中获取数据

- BaseExecutor.query(): 二级缓存不存在该数据，调用该方法
 - localCache.getObject(key): 尝试从本地缓存（一级缓存）获取数据

- BaseExecutor.queryFromDatabase(): 缓存获取数据失败，开始从数据库获取数据，并放入本地缓存

- SimpleExecutor.doQuery(): 执行 query
 - configuration.newStatementHandler(): 创建 StatementHandler 对象
 - 根据

```
public <E> List<E> query(Statement statement, ResultHandler resultHandler) {
    // 获取 SQL 语句
    String sql = boundSql.getSql();
    statement.execute(sql);
    // 通过 ResultSetHandler 对象封装结果集，映射成 JavaBean
    return resultSetHandler.handleResultSets(statement);
}
```

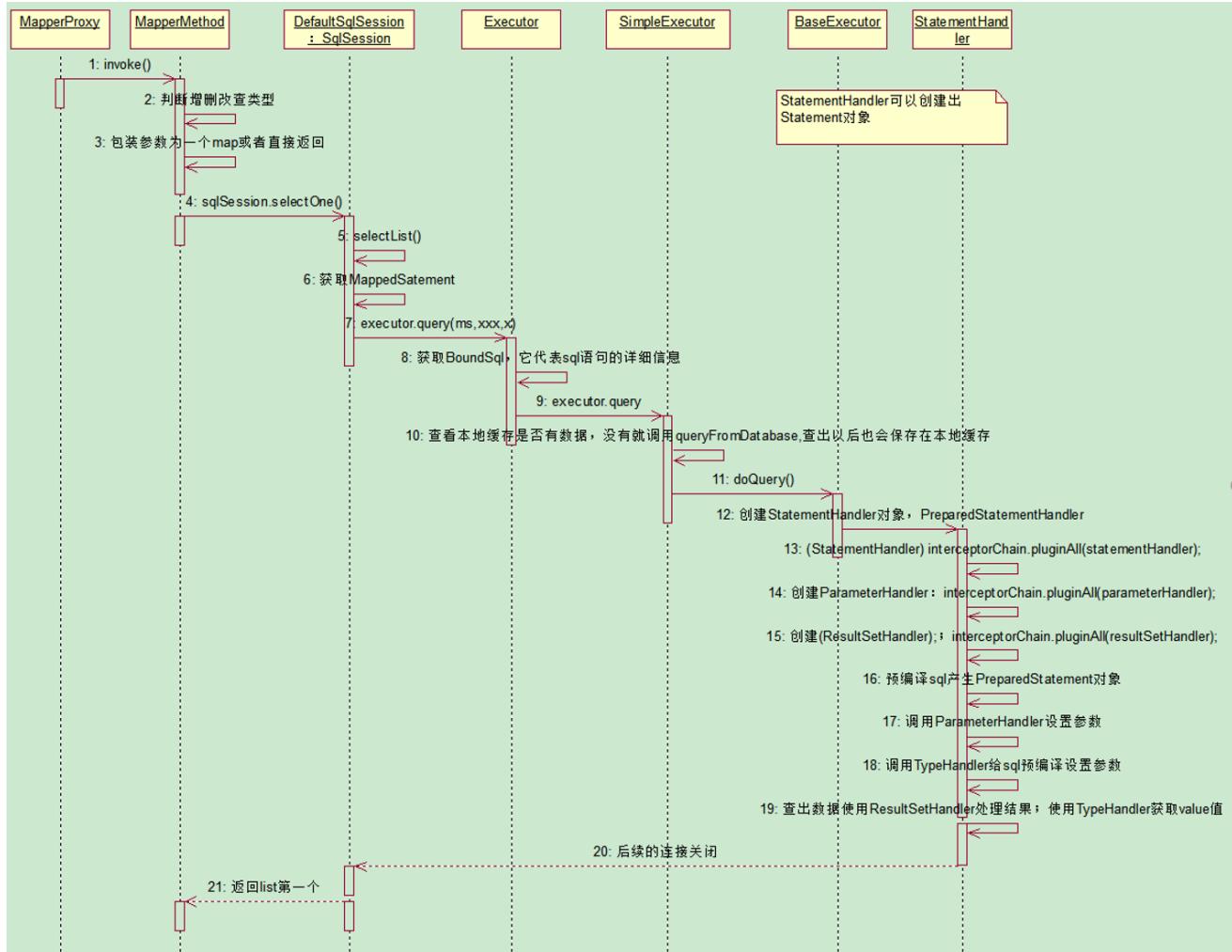
- resultSetHandler.handleResultSets(statement): 处理结果集
 - handleResultSet(rsw, resultMap, multipleResults, null): 底层回调
 - handleRowvalues(): 逐行处理数据，根据是否配置了 属性选择是否使用简单结果集映射
 - 首先判断数据是否被限制行数，然后进行结果集的映射
 - 最后将数据存入 ResultHandler 对象，底层就是 List 集合

```

public class DefaultResultHandler implements ResultHandler<Object> {
    private final List<Object> list;
    public void handleResult(ResultContext<?> context) {
        list.add(context.getResultObject());
    }
}

```

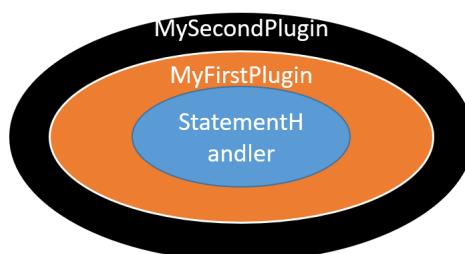
- `return collapseSingleResultList(multipleResults)`：可能存在多个结果集的情况
- `localCache.putObject(key, list)`：放入一级（本地）缓存
- `return list.get(0)`：返回结果集的第一个数据



插件使用

插件原理

实现原理：插件是按照插件配置顺序创建层层包装对象，执行目标方法的之后，按照逆向顺序执行（栈）



在四大对象创建时：

- 每个创建出来的对象不是直接返回的，而是 `interceptorChain.pluginAll(parameterHandler)`
- 获取到所有 Interceptor（插件需要实现的接口），调用 `interceptor.plugin(target)` 返回 target 包装后的对象

- 插件机制可以使用插件为目标对象创建一个代理对象，代理对象可以拦截到四大对象的每一个执行

```

@Intercepts(
{
    @Signature(type=StatementHandler.class,method="parameterize",args=java.sql.Statement.class)
})
public class MyFirstPlugin implements Interceptor{

    //intercept: 拦截目标对象的目标方法的执行
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("MyFirstPlugin...intercept:" + invocation.getMethod());
        //动态的改变一下sql运行的参数：以前1号员工，实际从数据库查询11号员工
        Object target = invocation.getTarget();
        System.out.println("当前拦截到的对象：" + target);
        //拿到：StatementHandler==>ParameterHandler==>parameterObject
        //拿到target的元数据
        MetaObject metaObject = SystemMetaObject.forObject(target);
        Object value = metaObject.getValue("parameterHandler.parameterObject");
        System.out.println("sql语句用的参数是：" + value);
        //修改完sql语句要用的参数
        metaObject.setValue("parameterHandler.parameterObject", 11);
        //执行目标方法
        Object proceed = invocation.proceed();
        //返回执行后的返回值
        return proceed;
    }

    // plugin: 包装目标对象的，为目标对象创建一个代理对象
    @Override
    public Object plugin(Object target) {
        //可以借助 Plugin 的 wrap 方法来使用当前 Interceptor 包装我们目标对象
        System.out.println("MyFirstPlugin...plugin:mybatis将要包装的对象" + target);
        Object wrap = Plugin.wrap(target, this);
        //返回为当前target创建的动态代理
        return wrap;
    }

    // setProperties: 将插件注册时的property属性设置进来
    @Override
    public void setProperties(Properties properties) {
        System.out.println("插件配置的信息：" + properties);
    }
}

```

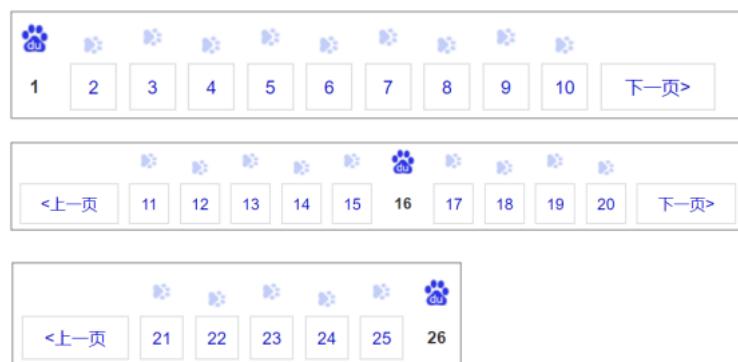
核心配置文件：

```

<!--plugins: 注册插件-->
<plugins>
    <plugin interceptor="mybatis.dao.MyFirstPlugin">
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
    </plugin>
</plugins>

```

分页插件



- 分页可以将很多条结果进行分页显示。如果当前在第一页，则没有上一页。如果当前在最后一页，则没有下一页，需要明确当前是第几页，这一页中显示多少条结果。
- MyBatis 是不带分页功能的，如果想实现分页功能，需要手动编写 LIMIT 语句，不同的数据库实现分页的 SQL 语句也是不同，手写分页成本较高。
- PageHelper：第三方分页助手，将复杂的分页操作进行封装，从而让分页功能变得非常简单

分页操作

开发步骤:

1. 导入 PageHelper 的 Maven 坐标
2. 在 MyBatis 核心配置文件中配置 PageHelper 插件

注意: 分页助手的插件配置在通用 Mapper 之前

```
<plugins>
    <plugin interceptor="com.github.pagehelper.PageInterceptor">
        <!-- 指定方言 -->
        <property name="dialect" value="mysql"/>
    </plugin>
</plugins>.....</mappers>
```

3. 与 MySQL 分页查询页数计算公式不同

```
static <E> Page<E> startPage(int pageNum, int pageSize) : pageNum第几页, pageSize页面大小
```

```
@Test
public void selectAll() {
    //第一页: 显示2条数据
    PageHelper.startPage(1,2);
    List<Student> students = sqlSession.selectList("StudentMapper.selectAll");
    for (Student student : students) {
        System.out.println(student);
    }
}
```

参数获取

PageInfo构造方法:

- o `PageInfo<Student> info = new PageInfo<>(list)` : list 是 SQL 执行返回的结果集合, 参考上一节

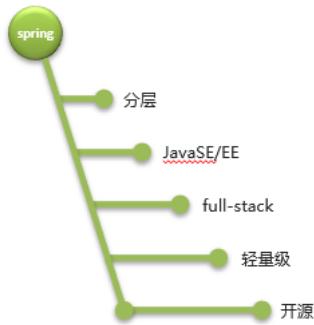
PageInfo相关API:

1. `startPage()`: 设置分页参数
2. `PageInfo`: 分页相关参数功能类。
3. `getTotal()`: 获取总条数
4. `getPages()`: 获取总页数
5. `getPageNum()`: 获取当前页
6. `getPageSize()`: 获取每页显示条数
7. `getPrePage()`: 获取上一页
8. `getNextPage()`: 获取下一页
9. `isFirstPage()`: 获取是否是第一页
10. `isLastPage()`: 获取是否是最后一页

Spring

概述

Spring 是分层的 JavaSE/EE 应用 full-stack 轻量级开源框架



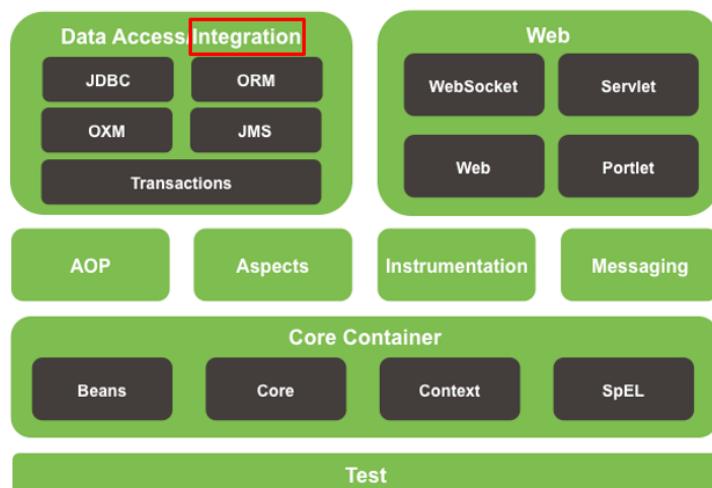
Spring 优点:

- 方便解耦，简化开发
- 方便集成各种框架
- 方便程序测试
- AOP 编程难过的支持
- 声明式事务的支持
- 降低 JavaEE API 的使用难度

体系结构:

Spring 体系结构

- 底层是核心容器
 - Beans
 - Core
 - Context
 - SpringEL 表达式
- 中间层技术
 - AOP
 - Aspects
- 应用层技术
 - 数据访问与数据集成
 - Web 集成
 - Web 实现
- 基于 Test 测试



参考视频: <https://space.bilibili.com/37974444>

IoC

基本概述

- IoC (Inversion Of Control) 控制反转，Spring 反向控制应用程序所需要使用的外部资源
- Spring 控制的资源全部放置在 Spring 容器中，该容器称为 IoC 容器（存放实例对象）
- 官方网站: <https://spring.io/> → Projects → spring-framework → LEARN → Reference Doc



- 桀合 (Coupling) : 代码编写过程中所使用技术的结合紧密度，用于衡量软件中各个模块之间的互联程度
- 内聚 (Cohesion) : 代码编写过程中单个模块内部各组成部分间的联系，用于衡量软件中各个功能模块内部的功能联系
- 代码编写的目标: 高内聚，低耦合。同一个模块内的各个元素之间要高度紧密，各个模块之间的相互依存度不紧密

入门项目

模拟三层架构中表现层调用业务层功能

- 表现层：UserApp 模拟 UserServicelet（使用 main 方法模拟）
- 业务层：UserService

步骤：

1. 导入 spring 坐标 (5.1.9.release)

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
```

2. 编写业务层与表现层（模拟）接口与实现类 service.UserService, service.impl.UserServiceImpl

```
public interface UserService {
    //业务方法
    void save();
}

public class UserServiceImpl implements UserService {
    public void save() {
        System.out.println("user service running...");
    }
}
```

3. 建立 Spring 配置文件：resources/applicationContext.xml (名字一般使用该格式)

4. 配置所需资源 (Service) 为 Spring 控制的资源

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 1. 创建 spring 控制的资源 -->
    <bean id="userService" class="service.impl.UserServiceImpl"/>
</beans>
```

5. 表现层 (App) 通过 Spring 获取资源 (Service 实例)

```
public class UserApp {
    public static void main(String[] args) {
        //2. 加载配置文件
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        //3. 获取资源
        UserService userService = (UserService) ctx.getBean("userService");
        userService.save(); //user service running...
    }
}
```



XML开发

bean

基本属性

标签: 标签, 的子标签

作用: 定义 Spring 中的资源, 受此标签定义的资源将受到 Spring 控制

格式:

```
<beans>
    <bean />
</beans>
```

基本属性

- id: bean 的名称, 通过 id 值获取 bean (首字母小写)
- class: bean 的类型, 使用完全限定类名
- name: bean 的名称, 可以通过 name 值获取 bean, 用于多人配合时给 bean 起别名

```
<bean id="beanId" name="beanName1,beanName2" class="ClassName"></bean>
```

```
ctx.getBean("beanId") == ctx.getBean("beanName1") == ctx.getBean("beanName2")
```

作用范围

作用: 定义 bean 的作用范围

格式:

```
<bean scope="singleton"></bean>
```

取值:

- singleton: 设定创建出的对象保存在 Spring 容器中, 是一个单例的对象
- prototype: 设定创建出的对象保存在 Spring 容器中, 是一个非单例 (原型) 的对象
- request、session、application、websocket: 设定创建出的对象放置在 web 容器对应的位置

Spring 容器中 Bean 的线程安全问题:

- 原型 Bean, 每次创建一个新对象, 线程之间并不存在 Bean 共享, 所以不会有线程安全的问题
 - 单例 Bean, 所有线程共享一个单例实例 Bean, 因此是存在资源的竞争, 如果单例 Bean 是一个无状态 Bean, 也就是线程中的操作不会对 Bean 的成员执行查询以外的操作, 那么这个单例 Bean 是线程安全的
- 解决方法: 开发人员来进行线程安全的保证, 最简单的办法就是把 Bean 的作用域 singleton 改为 prototype

生命周期

作用: 定义 bean 对象在初始化或销毁时完成的工作

格式:

```
<bean init-method="init" destroy-method="destroy"></bean>
```

取值: bean 对应的类中对应的具体方法名

实现接口的方式实现初始化, 无需配置文件配置 init-method:

- 实现 InitializingBean, 定义初始化逻辑
- 实现 DisposableBean, 定义销毁逻辑

注意事项:

- 当 scope="singleton" 时, Spring 容器中有且仅有一个对象, init 方法在创建容器时仅执行一次
- 当 scope="prototype" 时, Spring 容器要创建同一类型的多个对象, init 方法在每个对象创建时均执行一次
- 当 scope="singleton" 时, 关闭容器 (.close()) 会导致 bean 实例的销毁, 调用 destroy 方法一次
- 当 scope="prototype" 时, 对象的销毁由垃圾回收机制 GC 控制, destroy 方法将不会被执行

bean 配置:

```
<!--init-method和destroy-method用于控制bean的生命周期-->
<bean id="userService3" scope="prototype" init-method="init" destroy-method="destroy" class="service.impl.userServiceImpl"/>
```

业务层实现类:

```
public class UserServiceImpl implements UserService{
    public UserServiceImpl(){
        System.out.println(" constructor is running...");
```

```

}

public void init(){
    System.out.println("init....");
}

public void destroy(){
    System.out.println("destroy....");
}

public void save() {
    System.out.println("user service running...");
}
}

```

测试类:

```
UserService userService = (UserService)ctx.getBean("userService3");
```

创建方式

- 静态工厂

作用: 定义 bean 对象创建方式, 使用静态工厂的形式创建 bean, 兼容早期遗留系统的升级工作

格式:

```
<bean class="FactoryClassName" factory-method="factoryMethodName"></bean>
```

取值: 工厂 bean 中用于获取对象的静态方法名

注意事项: class 属性必须配置成静态工厂的类名

bean 配置:

```
<!--静态工厂创建 bean-->
<bean id="userService4" class="service.UserServiceFactory" factory-method="getService"/>
```

工厂类:

```

public class UserServiceFactory {
    public static UserService getService(){
        System.out.println("factory create object...");
        return new UserServiceImpl();
    }
}

```

测试类:

```
UserService userService = (UserService)ctx.getBean("userService4");
```

- 实例工厂

作用: 定义 bean 对象创建方式, 使用实例工厂的形式创建 bean, 兼容早期遗留系统的升级工作

格式:

```
<bean factory-bean="factoryBeanId" factory-method="factoryMethodName"></bean>
```

注意事项:

- 使用实例工厂创建 bean 首先需要将实例工厂配置 bean, 交由 Spring 进行管理
- factory-bean 是实例工厂的 Id

bean 配置:

```
<!--实例工厂创建 bean, 依赖工厂对象对应的 bean-->
<bean id="factoryBean" class="service.UserServiceFactory2"/>
<bean id="userService5" factory-bean="factoryBean" factory-method="getService"/>
```

工厂类:

```

public class UserServiceFactory2 {
    public UserService getService(){
        System.out.println(" instance factory create object...");
        return new UserServiceImpl();
    }
}

```

获取Bean

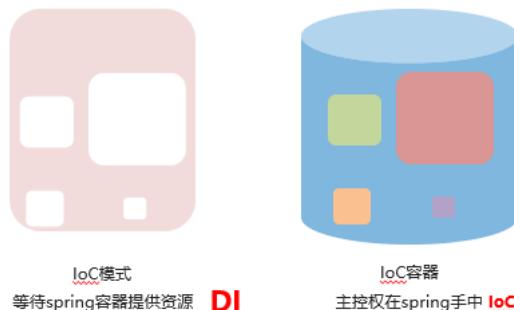
ApplicationContext 子类相关API:

方法	说明
String[] getBeanDefinitionNames()	获取 Spring 容器中定义的所有 JavaBean 的名称
BeanDefinition getBeanDefinition(String beanName)	返回给定 bean 名称的 BeanDefinition
String[] getBeanNamesForType(Class<?> type)	获取 Spring 容器中指定类型的所有 JavaBean 的名称
Environment getEnvironment()	获取与此组件关联的环境

DI

依赖注入

- IoC (Inversion Of Control) 控制翻转, Spring 反向控制应用程序所需要使用的外部资源
- DI (Dependency Injection) 依赖注入, 应用程序运行依赖的资源由 Spring 为其提供, 资源进入应用程序的方式称为注入, 简单说就是利用反射机制为类的属性赋值的操作



IoC 和 DI 的关系: IoC 与 DI 是同一件事站在不同角度看待问题

set 注入

标签: 标签, 的子标签

作用: 使用 set 方法的形式为 bean 提供资源

格式:

```

<bean>
    <property />
    <property />
    .....
</bean>

```

基本属性:

- name: 对应 bean 中的属性名, 要注入的变量名, 要求该属性必须提供可访问的 set 方法 (严格规范此名称是 set 方法对应名称, 首字母必须小写)
- value: 设定非引用类型属性对应的值, **不能与 ref 同时使用**
- ref: 设定引用类型属性对应 bean 的 id , 不能与 value 同时使用

```
<property name="propertyName" value="PropertyValue" ref="beanId"/>
```

代码实现:

- DAO 层: 要注入的资源

```
public interface UserDao {  
    public void save();  
}
```

```
public class UserDaoImpl implements UserDao{  
    public void save(){  
        System.out.println("user dao running...");  
    }  
}
```

- Service 业务层

```
public interface UserService {  
    public void save();  
}
```

```
public class UserServiceImpl implements UserService {  
    private UserDao userDao;  
    private int num;  
  
    //1.对需要进行注入的变量添加set方法  
    public void setUserDao(UserDao userDao) {  
        this.userDao = userDao;  
    }  
  
    public void setNum(int num) {  
        this.num = num;  
    }  
  
    public void save() {  
        System.out.println("user service running..." + num);  
        userDao.save();  
        bookDao.save();  
    }  
}
```

- 配置 applicationContext.xml

```
<!--2.将要注入的资源声明为bean-->  
<bean id="userDao" class="dao.impl.UserDaoImpl"/>  
  
<bean id="userService" class="service.impl.UserServiceImpl">  
    <!--3.将要注入的引用类型的变量通过property属性进行注入，-->  
    <property name="userDao" ref="userDao"/>  
    <property name="num" value="666"/>  
</bean>
```

- 测试类

```
public class UserApp {  
    public static void main(String[] args) {  
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");  
        UserService userService = (UserService) ctx.getBean("userService");  
        userService.save();  
    }  
}
```

构造注入

标签: 标签, 的子标签

作用: 使用构造方法的形式为 bean 提供资源, 兼容早期遗留系统的升级工作

格式:

```
<bean>  
    <constructor-arg />  
    ....<!--一个bean可以有多个constructor-arg标签-->  
</bean>
```

属性:

- name: 对应bean中的构造方法所携带的参数名
- value: 设定非引用类型构造方法参数对应的值, 不能与 ref 同时使用
- ref: 设定引用类型构造方法参数对应 bean 的 id , 不能与 value 同时使用
- type: 设定构造方法参数的类型, 用于按类型匹配参数或进行类型校验

- o index: 设定构造方法参数的位置, 用于按位置匹配参数, 参数 index 值从 0 开始计数

```
<constructor-arg name="argsName" value="argsValue" />
<constructor-arg index="arg-index" type="arg-type" ref="beanId"/>
```

代码实现:

- o DAO 层: 要注入的资源

```
public class UserDaoImpl implements UserDao{
    private String username;
    private String pwd;
    private String driver;

    public UserDaoImpl(String driver, String username, String pwd) {
        this.driver = driver;
        this.username = username;
        this.pwd = pwd;
    }
    public void save(){
        System.out.println("user dao running..." +username+ " " +pwd+ " " +driver);
    }
}
```

- o Service 业务层: 参考 set 注入

- o 配置 applicationContext.xml

```
<bean id="userDao" class="dao.impl.UserDaoImpl">
    <!--使用构造方法进行注入, 需要保障注入的属性与bean中定义的属性一致-->
    <!--一致指顺序一致或类型一致或使用index解决该问题-->
    <constructor-arg index="2" value="123"/>
    <constructor-arg index="1" value="root"/>
    <constructor-arg index="0" value="com.mysql.jdbc.driver"/>
</bean>

<bean id="userService" class="service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="num" value="666"/>
</bean>
```

方式二: 使用 UserServiceImpl 的构造方法注入

```
<bean id="userService" class="service.impl.UserServiceImpl">
    <constructor-arg name="userDao" ref="userDao"/>
    <constructor-arg name="num" value="666666"/>
</bean>
```

- o 测试类: 参考 set 注入

集合注入

标签: , 或 标签的子标签

作用: 注入集合数据类型属性

格式:

```
<property>
    <list></list>
</property>
```

代码实现:

- o DAO 层: 要注入的资源

```
public interface BookDao {
    public void save();
}
```

```
public class BookDaoImpl implements BookDao {
    private ArrayList al;
    private Properties properties;
    private int[] arr;
    private HashSet hs;
    private HashMap hm;

    public void setAl(ArrayList al) {
```

```

        this.al = al;
    }

    public void setProperties(Properties properties) {
        this.properties = properties;
    }

    public void setArr(int[] arr) {
        this.arr = arr;
    }

    public void setHs(HashSet hs) {
        this.hs = hs;
    }

    public void setHm(HashMap hm) {
        this.hm = hm;
    }

    public void save() {
        System.out.println("book dao running...");
        System.out.println("ArrayList:" + al);
        System.out.println("Properties:" + properties);
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
        System.out.println("HashSet:" + hs);
        System.out.println("HashMap:" + hm);
    }
}

```

- Service 业务层

```

public class UserServiceImpl implements UserService {
    private BookDao bookDao;

    public UserServiceImpl() {}

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    public void save() {
        System.out.println("user service running..." + num);
        bookDao.save();
    }
}

```

- 配置 applicationContext.xml

```

<bean id="userService" class="service.impl.UserServiceImpl">
    <property name="bookDao" ref="bookDao"/>
</bean>

<bean id="bookDao" class="dao.impl.BookDaoImpl">
    <property name="al">
        <list>
            <value>seazean</value>
            <value>66666</value>
        </list>
    </property>
    <property name="properties">
        <props>
            <prop key="name">seazean666</prop>
            <prop key="value">666666</prop>
        </props>
    </property>
    <property name="arr">
        <array>
            <value>123456</value>
            <value>66666</value>
        </array>
    </property>
    <property name="hs">
        <set>
            <value>seazean</value>
            <value>66666</value>
        </set>
    </property>
    <property name="hm">
        <map>
            <entry key="name" value="seazean6666" />
        </map>
    </property>
</bean>

```

```
        <entry key="value" value="6666666666"/>
    </map>
</property>
</bean>
```

P

标签: p:propertyName, p:propertyName-ref

作用：为 bean 注入属性值

格式：

```
<bean p:propertyName="PropertyValue" p:propertyName-ref="beanId"/>
```

开启 p 命令空间：开启 Spring 对 p 命令空间的支持，在 beans 标签中添加对应空间支持。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>
```

实例：

```
<bean  
    id="userService"  
    class="service.impl.UserServiceImpl"  
    p:userDao-ref="userDao"  
    p:bookDao-ref="bookDao"  
    p:num="10"  
>
```

SpEL

Spring 提供了对 EL 表达式的支持，统一属性注入格式

作用：为 bean 注入属性值

格式：

```
<property value="EL">
```

注意：所有属性值不区分是否引用类型，统一使用value赋值

所有格式统一使用 value="#{}"

- 常量 #{}{10} #{{3.14}} #{{2e5}} #{{'it'}}
 - 引用 bean #{{beanId}}
 - 引用 bean 属性 #{{beanId.propertyName}}
 - 引用 bean 方法 beanId.methodName().methodName()
 - 引用静态方法 T(java.lang.Math).PI
 - 运算符支持 #{{3 lt 4 == 4 ge 3}}
 - 正则表达式支持 #{{user.name matches'[a-z]{6,}'}}
 - 集合支持 #{{!likes[3]}}

实例：

```
<bean id="userService" class="service.impl.UserServiceImpl">
    <property name="userDao" value="#{userDao}"/>
    <property name="bookDao" value="#{bookDao}"/>
    <property name="num" value="#{6666666666}"/>
</bean>
```

prop

Spring 提供了读取外部 properties 文件的机制，使用读取到的数据为 bean 的属性赋值

操作步骤：

1. 准备外部 properties 文件
2. 开启 context 命名空间支持

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
    ">
```

3. 加载指定的 properties 文件

```
<context:property-placeholder location="classpath:data.properties" />
```

4. 使用加载的数据

```
<property name="propertyName" value="${propertiesName}" />
```

- 注意：如果需要加载所有的 properties 文件，可以使用 `*.properties` 表示加载所有的 properties 文件
- 注意：读取数据使用 `${propertiesName}` 格式进行，其中 `propertiesName` 指 properties 文件中的属性名

代码实现：

- data.properties

```
username=root
pwd=123456
```

- DAO 层：注入的资源

```
public interface UserDao {
    public void save();
}

public class UserDaoImpl implements UserDao{
    private String userName;
    private String password;

    public void setUserName(String userName) {
        this.userName = userName;
    }
    public void setPassword(String password) {
        this.password = password;
    }

    public void save(){
        System.out.println("user dao running..." +userName+ " "+password);
    }
}
```

- Service 业务层

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
    public void save() {
        System.out.println("user service running...");
        userDao.save();
    }
}
```

- applicationContext.xml

```

<context:property-placeholder location="classpath:*.properties"/>

<bean id="userDao" class="dao.impl.UserDaoImpl">
    <property name="userName" value="${username}"/>
    <property name="password" value="${pwd}"/>
</bean>

<bean id="userService" class="service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
</bean>

```

- 测试类

```

public class UserApp {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService) ctx.getBean("userService");
        userService.save();
    }
}

```

import

标签：，标签的子标签

作用：在当前配置文件中导入其他配置文件中的项

格式：

```

<beans>
    <import />
</beans>

```

属性：

- resource：加载的配置文件名

```
<import resource="config2.xml"/>
```

Spring 容器加载多个配置文件：

- applicationContext-book.xml

```

<bean id="bookDao" class="dao.impl.BookDaoImpl">
    <property name="num" value="1"/>
</bean>

```

- applicationContext-user.xml

```

<bean id="userDao" class="dao.impl.UserDaoImpl">
    <property name="userName" value="${username}"/>
    <property name="password" value="${pwd}"/>
</bean>

<bean id="userService" class="service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="bookDao" ref="bookDao"/>
</bean>

```

- applicationContext.xml

```

<import resource="applicationContext-user.xml"/>
<import resource="applicationContext-book.xml"/>

<bean id="bookDao" class="com.seazean.dao.impl.BookDaoImpl">
    <property name="num" value="2"/>
</bean>

```

- 测试类

```

new ClassPathXmlApplicationContext("applicationContext-user.xml", "applicationContext-book.xml");
new ClassPathXmlApplicationContext("applicationContext.xml");

```

Spring 容器中的 bean 定义冲突问题

- 同 id 的 bean，后定义的覆盖先定义的
- 导入配置文件可以理解为将导入的配置文件复制粘贴到对应位置，程序执行选择最下面的配置使用

- 导入配置文件的顺序与位置不同可能会导致最终程序运行结果不同
-

三方资源

Druid

第三方资源配置

- pom.xml

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.16</version>
</dependency>
```

- applicationContext.xml

```
<!--加载druid资源-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://192.168.2.185:3306/spring_db"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</bean>
```

- App.java

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
DruidDataSource datasource = (DruidDataSource) ctx.getBean("dataSource");
System.out.println(datasource);
```

Mybatis

Mybatis 核心配置文件消失

- 环境 environment 转换成数据源对象
- 映射 Mapper 扫描工作交由 Spring 处理
- 类型别名交由 Spring 处理

DAO 接口不需要创建实现类，MyBatis-Spring 提供了一个动态代理的实现 **MapperFactoryBean**，这个类可以让直接注入数据映射器接口到 service 层 bean 中，底层将会动态代理创建类

整合原理：利用 Spring 框架的 SPI 机制，在 META-INF 目录的 spring.handlers 中给 Spring 容器中导入 NamespaceHandler 类

- NamespaceHandler 的 init 方法注册 bean 信息的解析器 MapperScannerBeanDefinitionParser
- 解析器在 Spring 容器创建过程中去解析 **mapperScanner** 标签，解析出的属性填充到 MapperScannerConfigurer 中
- MapperScannerConfigurer 实现了 BeanDefinitionRegistryPostProcessor 接口，重写 postProcessBeanDefinitionRegistry() 方法，可以扫描到 MyBatis 的 Mapper

注解开发

注解驱动

XML

启动注解扫描，加载类中配置的注解项：

```
<context:component-scan base-package="packageName1,packageName2"/>
```

说明：

- 在进行包扫描时，会对配置的包及其子包中所有文件进行扫描，多个包采用，隔开
- 扫描过程是以文件夹递归迭代的形式进行的
- 扫描过程仅读取合法的 Java 文件
- 扫描时仅读取 Spring 可识别的注解
- 扫描结束后会将可识别的有效注解转化为 Spring 对应的资源加入 IoC 容器
- 从加载效率上来说注解优于 XML 配置文件

注解：启动时使用注解的形式替代 xml 配置，将 Spring 配置文件从工程中消除，简化书写

缺点：为了达成注解驱动的目的，可能会将原先很简单的书写，变的更加复杂。XML 中配置第三方开发的资源是很方便的，但使用注解驱动无法在第三方开发的资源中进行编辑，因此会增大开发工作量

```

<bean
    id="userService"
    class="UserServiceImpl"
    scope="prototype"
    init-method="init"
    destroy-method="destroy"
/>

```

```

@Component("userService")
@Scope("prototype")
public class UserServiceImpl implements ....{
    @PostConstruct
    public void init() {
        System.out.println("init...");
    }
    @PreDestroy
    public void destroy() {
        System.out.println("destroy...");
    }
}

```

纯注解

注解配置类

名称: @Configuration、@ComponentScan

类型: 类注解

作用: 设置当前类为 Spring 核心配置加载类

格式:

```

@Configuration
@ComponentScan({"scanPackageName1","scanPackageName2"})
public class SpringConfigClassName{
}

```

说明:

- 核心配合类用于替换 Spring 核心配置文件, 此类可以设置空的, 不设置变量与属性
- bean 扫描工作使用注解 @ComponentScan 替代, 多个包用 {} 和 , 隔开

加载纯注解格式上下文对象, 需要使用 AnnotationConfigApplicationContext

```

@Configuration
public class SpringConfig {
    @Bean
    public Person person() {
        return new Person("lisi", 20);
    }
}

public class MainTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
            AnnotationConfigApplicationContext(SpringConfig.class);
        //方式一: 名称对应类名
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);

        //方式二: 名称对应方法名
        Person bean1 = (Person) applicationContext.getBean("person1");

        //方法三: 指定名称@Bean("person2")
    }
}

```

扫描器

组件扫描过滤器

开发过程中, 需要根据需求加载必要的 bean, 排除指定 bean

名称: @ComponentScan

类型: 类注解

作用：设置 Spring 配置加载类扫描规则

格式：

```
@ComponentScan(  
    value = {"dao", "service"},           //设置基础扫描路径  
    excludeFilters = {  
        @ComponentScan.Filter(  
            type= FilterType.ANNOTATION,  
            classes = Service.class)      //设置过滤器  
    }  
)
```

属性：

- includeFilters：设置包含性过滤器
- excludeFilters：设置排除性过滤器
- type：设置过滤器类型

基本注解

设置 bean

名称：@Component、@Controller、@Service、@Repository

类型：类注解，写在类定义上方

作用：设置该类为 Spring 管理的 bean

格式：

```
@Component  
public class className{}
```

说明：@Controller、@Service、@Repository 是@Component 的衍生注解，功能同@Component

属性：

- value（默认）：定义 bean 的访问 id

作用范围

名称：@Scope

类型：类注解，写在类定义上方

作用：设置该类作为 bean 对应的 scope 属性

格式：

```
@Scope  
public class className{}
```

相关属性

- value（默认）：定义 bean 的作用域，默认为 singleton，非单例取值 prototype

生命周期

名称：@PostConstruct、@PreDestroy

类型：方法注解，写在方法定义上方

作用：设置该类作为 bean 对应的生命周期方法

示例：

```
//定义bean，后面添加bean的id  
@Component("userService")  
//定义bean的作用域  
@Scope("singleton")  
public class UserServiceImpl implements UserService {  
    //初始化  
    @PostConstruct  
    public void init(){  
        System.out.println("user service init...");  
    }  
}
```

```
    }
    //销毁
    @PreDestroy
    public void destroy(){
        System.out.println("user service destroy...");
    }
}
```

一个对象的执行顺序: Constructor >> @Autowired (注入属性) >> @PostConstruct (初始化逻辑)

加载资源

名称: @Bean

类型: 方法注解

作用: 设置该方法的返回值作为 Spring 管理的 bean

格式:

```
@Bean("dataSource")
public DruidDataSource createDataSource() { return ..... }
```

说明:

- 因为第三方 bean 无法在其源码上进行修改, 使用 @Bean 解决第三方 bean 的引入问题
- 该注解用于替代 XML 配置中的静态工厂与实例工厂创建 bean, 不区分方法是否为静态或非静态
- @Bean 所在的类必须被 Spring 扫描加载, 否则该注解无法生效

相关属性

- value (默认) : 定义 bean 的访问 id
 - initMethod: 声明初始化方法
 - destroyMethod: 声明销毁方法
-

属性注入

基本类型

名称: @Value

类型: 属性注解、方法注解

作用: 设置对应属性的值或对方法进行传参

格式:

```
//@Value("${jdbc.username}")
@Value("root")
private String username;
```

说明:

- value 值仅支持非引用类型数据, 赋值时对方法的所有参数全部赋值
- value 值支持读取 properties 文件中的属性值, 通过类属性将 properties 中数据传入类中
- value 值支持 SpEL
- @value 注解如果添加在属性上方, 可以省略 set 方法 (set 方法的目的是为属性赋值)

相关属性:

- value (默认) : 定义对应的属性值或参数值
-

自动装配

属性注入

名称: @Autowired、@Qualifier

类型: 属性注解、方法注解

作用: 设置对应属性的对象、对方法进行引用类型传参

格式:

```
@Autowired(required = false)
@Qualifier("userDao")
private UserDao userDao;
```

说明:

- @Autowired 默认按类型装配，指定 @Qualifier 后可以指定自动装配的 bean 的 id

相关属性:

- required: 为 true (默认) 表示注入 bean 时该 bean 必须存在，不然就会注入失败抛出异常；为 false 表示注入时该 bean 存在就注入，不存在就忽略跳过

注意: 在使用 @Autowired 时，首先在容器中查询对应类型的 bean，如果查询结果刚好为一个，就将该 bean 装配给 @Autowired 指定的数据，如果查询的结果不止一个，那么 @Autowired 会根据名称来查找，如果查询的结果为空，那么会抛出异常

解决方法: 使用 required = false

优先注入

名称: @Primary

类型: 类注解

作用: 设置类对应的 bean 按类型装配时优先装配

范例:

```
@Primary  
public class ClassName{}
```

说明:

- @Autowired 默认按类型装配，当出现相同类型的 bean，使用 @Primary 提高按类型自动装配的优先级，多个 @Primary 会导致优先级设置无效
-

注解对比

名称: @Inject、@Named、@Resource

- @Inject 与 @Named 是 JSR330 规范中的注解，功能与 @Autowired 和 @Qualifier 完全相同，适用于不同架构场景
- @Resource 是 JSR250 规范中的注解，可以简化书写格式

@Resource 相关属性

- name: 设置注入的 bean 的 id
- type: 设置注入的 bean 的类型，接收的参数为 Class 类型

@Autowired 和 @Resource之间的区别:

- @Autowired 默认是按照类型装配注入，默认情况下它要求依赖对象必须存在（可以设置 required 属性为 false）
 - @Resource 默认按照名称装配注入，只有当找不到与名称匹配的 bean 才会按照类型来装配注入
-

静态注入

Spring 容器管理的都是实例对象，@Autowired 依赖注入的都是容器内的对象实例，在 Java 中 static 修饰的静态属性（变量和方法）是属于类的，而非属于实例对象

当类加载器加载静态变量时，Spring 上下文尚未加载，所以类加载器不会在 Bean 中正确注入静态类

```
@Component  
public class TestClass {  
    @Autowired  
    private static Component component;  
  
    // 调用静态组件的方法  
    public static void testMethod() {  
        component.callTestMethod();  
    }  
}  
// 编译正常，但运行时报java.lang.NullPointerException，所以在调用testMethod()方法时，component变量还没被初始化
```

解决方法:

- @Autowired 注解到类的构造函数上，Spring 扫描到 Component 的 Bean，然后赋给静态变量 component

```

@Component
public class TestClass {
    private static Component component;

    @Autowired
    public TestClass(Component component) {
        TestClass.component = component;
    }

    public static void testMethod() {
        component.callTestMethod();
    }
}

```

- @Autowired 注解到静态属性的 setter 方法上
- 使用 @PostConstruct 注解一个方法，在方法内为 static 静态成员赋值
- 使用 Spring 框架工具类获取 bean，定义成局部变量使用

```

public class TestClass {
    // 调用静态组件的方法
    public static void testMethod() {
        Component component = ApplicationContextUtil.getBean("component");
        component.callTestMethod();
    }
}

```

参考文章: <http://jessehzx.top/2018/03/18/spring-autowired-static-field/>

文件读取

名称: @PropertySource

类型: 类注解

作用: 加载 properties 文件中的属性值

格式:

```

@PropertySource(value = "classpath:filename.properties")
public class className {
    @Value("${propertiesAttributeName}")
    private String attributeName;
}

```

说明:

- 不支持 * 通配符，加载后，所有 Spring 控制的 bean 中均可使用对应属性值，加载多个需要用 {} 和 , 隔开

相关属性

- value (默认) : 设置加载的 properties 文件名
- ignoreResourceNotFound: 如果资源未找到，是否忽略，默认为 false

加载控制

依赖加载

@DependsOn

- 名称: @DependsOn
- 类型: 类注解、方法注解
- 作用: 控制 bean 的加载顺序，使其在指定 bean 加载完毕后再加载
- 格式:

```

@DependsOn("beanId")
public class className {
}

```

- 说明:

- 配置在方法上，使 @DependsOn 指定的 bean 优先于 @Bean 配置的 bean 进行加载
- 配置在类上，使 @DependsOn 指定的 bean 优先于当前类中所有 @Bean 配置的 bean 进行加载

- 配置在类上，使 @DependsOn 指定的 bean 优先于 @Component 等配置的 bean 进行加载
- 相关属性
 - value (默认)：设置当前 bean 所依赖的 bean 的 id

@Order

- 名称: @Order
- 类型: 配置类注解
- 作用: 控制配置类的加载顺序，值越小越先加载
- 格式:

```
@order(1)
public class springConfigClassName {  
}
```

@Lazy

- 名称: @Lazy
- 类型: 类注解、方法注解
- 作用: 控制 bean 的加载时机，使其延迟加载，获取的时候加载
- 格式:

```
@Lazy
public class className {  
}
```

应用场景

@DependsOn

- 微信订阅号，发布消息和订阅消息的 bean 的加载顺序控制（先开订阅，再发布）
- 双 11 活动，零点前是结算策略 A，零点后是结算策略 B，策略 B 操作的数据为促销数据，策略 B 加载顺序与促销数据的加载顺序

@Lazy

- 程序灾难出现后对应的应急预案处理是启动容器时加载时机

@Order

- 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

整合资源

导入

名称: @Import

类型: 类注解

作用: 导入第三方 bean 作为 Spring 控制的资源，这些类都会被 Spring 创建并放入 ioc 容器

格式:

```
@Configuration
@Import(OtherClassName.class)
public class className {  
}
```

说明:

- @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
- 在被导入的类中可以继续使用 @Import 导入其他资源
- @Bean 所在的类可以使用导入的形式进入 Spring 容器，无需声明为 bean

Druid

- 加载资源

```

@Component
public class JDBCConfig {
    @Bean("dataSource")
    public static DruidDataSource getDataSource() {
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://192.168.2.185:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("123456");
        return ds;
    }
}

```

- 导入资源

```

@Configuration
@ComponentScan(value = {"service", "dao"})
@Import(JDBCConfig.class)
public class SpringConfig {
}

```

- 测试

```

DruidDataSource dataSource = (DruidDataSource) ctx.getBean("dataSource");
System.out.println(dataSource);

```

JUnit

Spring 接管 Junit 的运行权，使用 Spring 专用的 Junit 类加载器，为 Junit 测试用例设定对应的 Spring 容器

注意：

- 从 Spring5.0 以后，要求 Junit 的版本必须是4.12及以上
- Junit 仅用于单元测试，不能将 Junit 的测试类配置成 Spring 的 bean，否则该配置将会被打包进入工程中

test / java / service / UserServiceTest

```

//设定spring专用的类加载器
@RunWith(SpringJUnit4ClassRunner.class)
//设定加载的spring上下文对应的配置
@ContextConfiguration(classes = SpringConfig.class)
public class UserServiceTest {
    @Autowired
    private AccountService accountService;
    @Test
    public void testFindByID() {
        Account account = accountService.findById(1);
        Assert.assertEquals("Mike", account.getName());
    }
}

```

pom.xml

```

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>

```

IoC 原理

核心类

BeanFactory

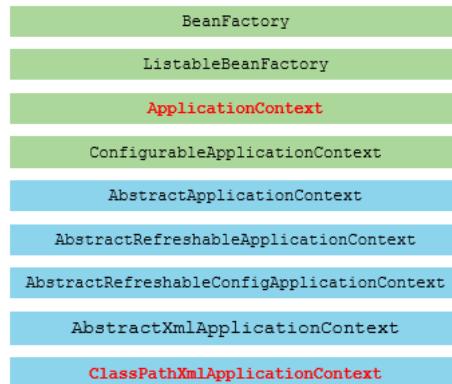
ApplicationContext:

1. ApplicationContext 是一个接口，提供了访问 Spring 容器的 API
2. ClassPathXmlApplicationContext 是一个类，实现了上述功能
3. ApplicationContext 的顶层接口是 BeanFactory
4. BeanFactory 定义了 bean 相关的最基本操作
5. ApplicationContext 在 BeanFactory 基础上追加了若干新功能

ApplicationContext 和 BeanFactory 对比:

- BeanFactory 和 ApplicationContext 是 Spring 的两大核心接口，都可以当做 Spring 的容器
- BeanFactory 是 Spring 里面最底层的接口，是 IoC 的核心，定义了 IoC 的基本功能，包含了各种 Bean 的定义、加载、实例化，依赖注入和生命周期管理。ApplicationContext 接口作为 BeanFactory 的子类，除了提供 BeanFactory 所具有的功能外，还提供了更完整的框架功能：
 - 继承 MessageSource，因此支持国际化
 - 资源文件访问，如 URL 和文件（ResourceLoader）。
 - 载入多个（有继承关系）上下文（即加载多个配置文件），使得每一个上下文都专注于一个特定的层次，比如应用的 web 层
 - 提供在监听器中注册 bean 的事件
- BeanFactory 创建的 bean 采用延迟加载形式，只有在使用到某个 Bean 时（调用 getBean），才对该 Bean 进行加载实例化（Spring 早期使用该方法获取 bean），这样就不能提前发现一些存在的 Spring 的配置问题；ApplicationContext 是在容器启动时，一次性创建了所有的 Bean，容器启动时，就可以发现 Spring 中存在的配置错误，这样有利于检查所依赖属性是否注入
- ApplicationContext 启动后预载入所有的单实例 Bean，所以程序启动慢，运行时速度快
- 两者都支持 BeanPostProcessor、BeanFactoryPostProcessor 的使用，但两者之间的区别是：BeanFactory 需要手动注册，而 ApplicationContext 则是自动注册

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



BeanFactory 的成员属性:

```
String FACTORY_BEAN_PREFIX = "&";
```

- 区分是 FactoryBean 还是创建的 Bean，加上 & 代表是工厂，getBean 将会返回工厂
- FactoryBean: 如果某个 bean 的配置非常复杂，或者想要使用编码的形式去构建它，可以提供一个构建该 bean 实例的工厂，这个工厂就是 FactoryBean 接口实现类，FactoryBean 接口实现类也是需要 Spring 管理
 - 这里产生两种对象，一种是 FactoryBean 接口实现类（IOC 管理），另一种是 FactoryBean 接口内部管理的对象
 - 获取 FactoryBean 接口实现类，使用 getBean 时传的 beanName 需要带 & 开头
 - 获取 FactoryBean 内部管理的对象，不需要带 & 开头

BeanFactory 的基本使用:

```
Resource res = new ClassPathResource("applicationContext.xml");
BeanFactory bf = new XmlBeanFactory(res);
UserService userService = (UserService)bf.getBean("userService");
```

FactoryBean

FactoryBean: 对单一的 bean 的初始化过程进行封装，达到简化配置的目的

FactoryBean 与 BeanFactory 区别:

- FactoryBean: 封装单个 bean 的创建过程，就是工厂的 Bean
- BeanFactory: Spring 容器顶层接口，定义了 bean 相关的获取操作

代码实现:

- FactoryBean，实现类一般是 MapperFactoryBean，创建 DAO 层接口的实现类

```
public class EquipmentDaoImplFactoryBean implements FactoryBean {
    @Override //获取Bean
    public Object getObject() throws Exception {
        return new EquipmentDaoImpl();
```

```

    }

    @Override //获取bean的类型
    public Class<?> getObjectType() {
        return null;
    }

    @Override //是否单例
    public boolean isSingleton() {
        return false;
    }
}

```

- MapperFactoryBean 继承 SqlSessionDaoSupport，可以获取 SqlSessionTemplate，完成 MyBatis 的整合

```

public abstract class SqlSessionDaoSupport extends DaoSupport {
    private SqlSessionTemplate sqlSessionTemplate;
    // 获取 sqlSessionTemplate 对象
    public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
        if (this.sqlSessionTemplate == null || sqlSessionFactory != this.sqlSessionTemplate.getSqlSessionFactory()) {
            this.sqlSessionTemplate = createSqlSessionTemplate(sqlSessionFactory);
        }
    }
}

```

过滤器

数据准备

- DAO 层 UserDao、AccountDao、BookDao、EquipmentDao

```

public interface UserDao {
    public void save();
}

```

```

@Component("userDao")
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("user dao running...");
    }
}

```

- Service 业务层

```

public interface UserService {
    public void save();
}

```

```

@Service("userService")
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;//.....BookDao等

    public void save() {
        System.out.println("user service running...");
        userDao.save();
    }
}

```

过滤器

名称: TypeFilter

类型: 接口

作用: 自定义类型过滤器

示例:

- config / filter / MyTypeFilter

```

public class MyTypeFilter implements TypeFilter {
    @Override
    /**
     * metadataReader:读取到的当前正在扫描的类的信息
     * metadataReaderFactory:可以获取到任何其他类的信息
     */
    //加载的类满足要求，匹配成功
    public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {
        //获取当前类注解的信息
        AnnotationMetadata am = metadataReader.getAnnotationMetadata();
        //获取当前正在扫描的类的类信息
        ClassMetadata classMetadata = metadataReader.getClassMetadata();
        //获取当前类资源（类的路径）
        Resource resource = metadataReader.getResource();

        //通过类的元数据获取类的名称
        String className = classMetadata.getClassName();
        //如果加载的类名满足过滤器要求，返回匹配成功
        if(className.equals("service.impl.UserServiceImpl")){
            //返回true表示匹配成功，返回false表示匹配失败。此处仅确认匹配结果，不会确认是排除还是加入，排除/加入由配置项决定，与此处无关
            return true;
        }
        return false;
    }
}

```

- o SpringConfig

```

@Configuration
//设置排除bean，排除的规则是自定义规则（FilterType.CUSTOM），具体的规则定义为MyTypeFilter
@ComponentScan(
    value = {"dao", "service"},
    excludeFilters = @ComponentScan.Filter(
        type= FilterType.CUSTOM,
        classes = MyTypeFilter.class
    )
)
public class springConfig {
}

```

导入器

bean 只有通过配置才可以进入 Spring 容器，被 Spring 加载并控制

- o 配置 bean 的方式如下：

- XML 文件中使用 标签配置
- 使用 @Component 及衍生注解配置

导入器可以快速高效导入大量 bean，替代 @Import({a.class,b.class})，无需在每个类上添加 @Bean

名称： ImportSelector

类型： 接口

作用： 自定义bean导入器

- o selector / MyImportSelector

```

public class MyImportSelector implements ImportSelector{
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        // 1.编程形式加载一个类
        // return new String[]{"dao.impl.BookDaoImpl"};

        // 2.加载import.properties文件中的单个类名
        // ResourceBundle bundle = ResourceBundle.getBundle("import");
        // String className = bundle.getString("className");

        // 3.加载import.properties文件中的多个类名
        ResourceBundle bundle = ResourceBundle.getBundle("import");
        String className = bundle.getString("className");
        return className.split(",");
    }
}

```

- o import.properties

```

#2.加载import.properties文件中的单个类名
@classname=dao.impl.BookDaoImpl

#3.加载import.properties文件中的多个类名
@classname=dao.impl.BookDaoImpl,dao.impl.AccountDaoImpl

#4.导入包中的所有类
path=dao.impl.*

```

- SpringConfig

```

@Configuration
@ComponentScan({"dao", "service"})
@Import(MyImportSelector.class)
public class SpringConfig {
}

```

注册器

可以取代 ComponentScan 扫描器

名称: ImportBeanDefinitionRegistrar

类型: 接口

作用: 自定义 bean 定义注册器

- registrar / MyImportBeanDefinitionRegistrar

```

public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    /**
     * AnnotationMetadata:当前类的注解信息
     * BeanDefinitionRegistry:BeanDefinition注册类，把所有需要添加到容器中的bean调用registerBeanDefinition手工注册进来
     */
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        //自定义注册器
        //1.开启类路径bean定义扫描器，需要参数bean定义注册器BeanDefinitionRegistry，需要制定是否使用默认类型过滤器
        ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(registry, false);
        //2.添加包含性加载类型过滤器（可选，也可以设置为排除性加载类型过滤器）
        scanner.addIncludeFilter(new TypeFilter() {
            @Override
            public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {
                //所有匹配全部成功，此处应该添加实际的业务判定条件
                return true;
            }
        });
        //设置扫描路径
        scanner.addExcludeFilter(tf); //排除
        scanner.scan("dao", "service");
    }
}

```

- SpringConfig

```

@Configuration
@Import(MyImportBeanDefinitionRegistrar.class)
public class SpringConfig {
}

```

处理器

通过创建类继承相应的处理器的接口，重写后置处理的方法，来实现拦截 Bean 的生命周期来实现自己自定义的逻辑

BeanPostProcessor: bean 后置处理器，bean 创建对象初始化前后进行拦截工作的

BeanFactoryPostProcessor: beanFactory 的后置处理器

- 加载时机：在 BeanFactory 初始化之后调用，来定制和修改 BeanFactory 的内容；所有的 bean 定义已经保存加载到 beanFactory，但是 bean 的实例还未创建
- 执行流程：

- ioc 容器创建对象
- invokeBeanFactoryPostProcessors(beanFactory): 执行 BeanFactoryPostProcessor

- 在 BeanFactory 中找到所有类型是 BeanFactoryPostProcessor 的组件，并执行它们的方法
- 在初始化创建其他组件前面执行

BeanDefinitionRegistryPostProcessor:

- 加载时机：在所有 bean 定义信息将要被加载，但是 bean 实例还未创建，优先于 BeanFactoryPostProcessor 执行；利用 BeanDefinitionRegistryPostProcessor 给容器中再额外添加一些组件
- 执行流程：
 - ioc 容器创建对象
 - refresh() → invokeBeanFactoryPostProcessors(beanFactory)
 - 从容器中获取到所有的 BeanDefinitionRegistryPostProcessor 组件
 - 依次触发所有的 postProcessBeanDefinitionRegistry() 方法
 - 再来触发 postProcessBeanFactory() 方法

监听器

基本概述

ApplicationListener：监听容器中发布的事件，完成事件驱动模型开发

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

所以监听 ApplicationEvent 及其下面的子事件

应用监听器步骤：

- 写一个监听器 (ApplicationListener实现类) 来监听某个事件 (ApplicationEvent及其子类)
- 把监听器加入到容器 @Component
- 只要容器中有相关事件的发布，就能监听到这个事件；
 - ContextRefreshedEvent：容器刷新完成（所有 bean 都完全创建）会发布这个事件
 - ContextClosedEvent：关闭容器会发布这个事件
- 发布一个事件： applicationContext.publishEvent()

```
@Component
public class MyApplicationListener implements ApplicationListener<ApplicationEvent> {
    //当容器中发布此事件以后，方法触发
    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        System.out.println("收到事件：" + event);
    }
}
```

实现原理

ContextRefreshedEvent 事件：

- 容器初始化过程中执行 initApplicationEventMulticaster()：初始化事件多播器
 - 先去容器中查询 id = applicationEventMulticaster 的组件，有直接返回
 - 没有就执行 this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory) 并且加入到容器中
 - 以后在其他组件要派发事件，自动注入这个 applicationEventMulticaster
- 容器初始化过程执行 registerListeners() 注册监听器
 - 从容器中获取所有监听器： getBeanNamesForType(ApplicationListener.class, true, false)
 - 将 listener 注册到 ApplicationEventMulticaster
- 容器刷新完成： finishRefresh() → publishEvent(new ContextRefreshedEvent(this))

发布 ContextRefreshedEvent 事件：

- 获取事件的多播器（派发器）：getApplicationEventMulticaster()
- multicastEvent 派发事件
 - 获取到所有的 ApplicationListener
 - 遍历 ApplicationListener
 - 如果有 Executor，可以使用 Executor 异步派发 Executor executor = getTaskExecutor()
 - 没有就同步执行 listener 方法 invokeListener(listener, event)，拿到 listener 回调 onApplicationEvent

容器关闭会发布 ContextClosedEvent

注解实现

注解：@EventListener

基本使用：

```
@Service
public class UserService{
    @EventListener(classes={ApplicationEvent.class})
    public void listen(ApplicationEvent event){
        System.out.println("UserService。。监听到的事件：" + event);
    }
}
```

原理：使用 EventListenerMethodProcessor 处理器来解析方法上的 @EventListener，Spring 扫描使用注解的方法，并为之创建一个监听对象

SmartInitializingSingleton 原理：afterSingletonsInstantiated()

- IOC 容器创建对象并 refresh()
- finishBeanFactoryInitialization(beanFactory)：初始化剩下的单实例 bean
 - 先创建所有的单实例 bean：getBean()
 - 获取所有创建好的单实例 bean，判断是否是 SmartInitializingSingleton 类型的，如果是就调用 afterSingletonsInstantiated()

AOP

基本概述

AOP (Aspect Oriented Programming)：面向切面编程，一种编程范式，指导开发者如何组织程序结构

AOP 弥补了 OOP 的不足，基于 OOP 基础之上进行横向开发：

- uOOP 规定程序开发以类为主体模型，一切围绕对象进行，完成某个任务先构建模型
- uAOP 程序开发主要关注基于 OOP 开发中的共性功能，一切围绕共性功能进行，完成某个任务先构建可能遇到的所有共性功能（当所有功能都开发出来也就没有共性与非共性之分），将软件开发由手工制作走向半自动化/全自动化阶段，实现“插拔式组件体系结构”搭建

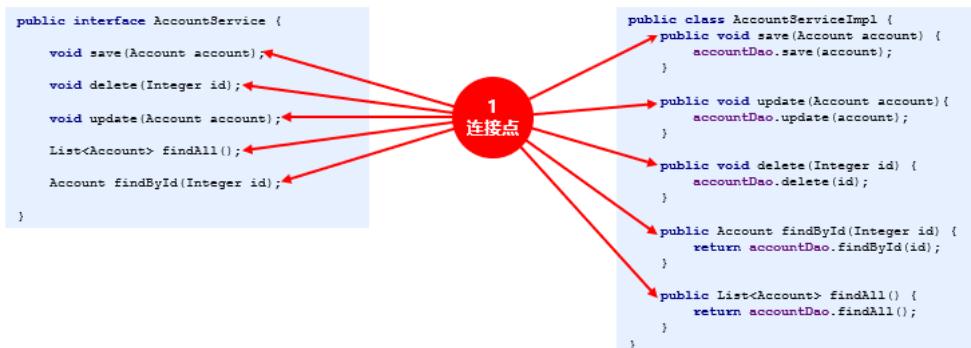
AOP 作用：

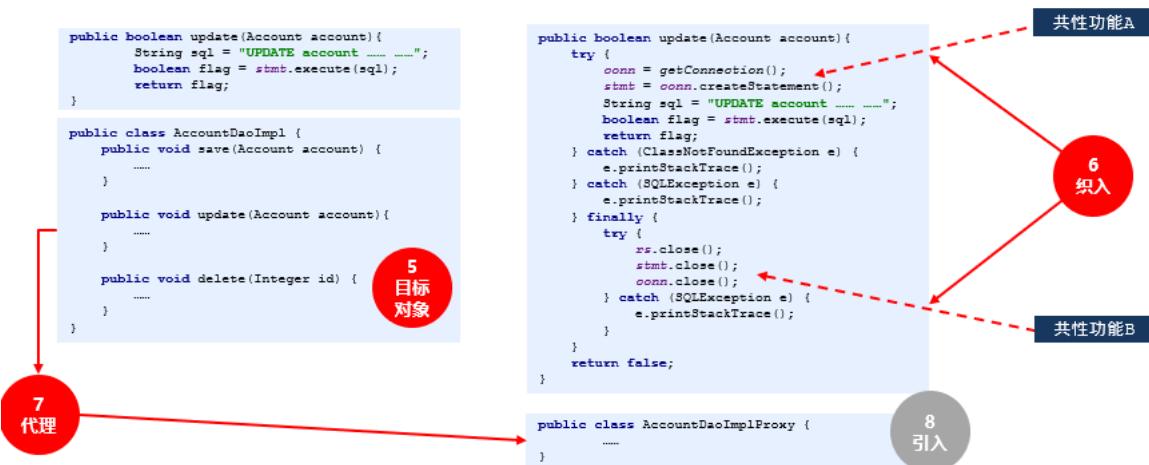
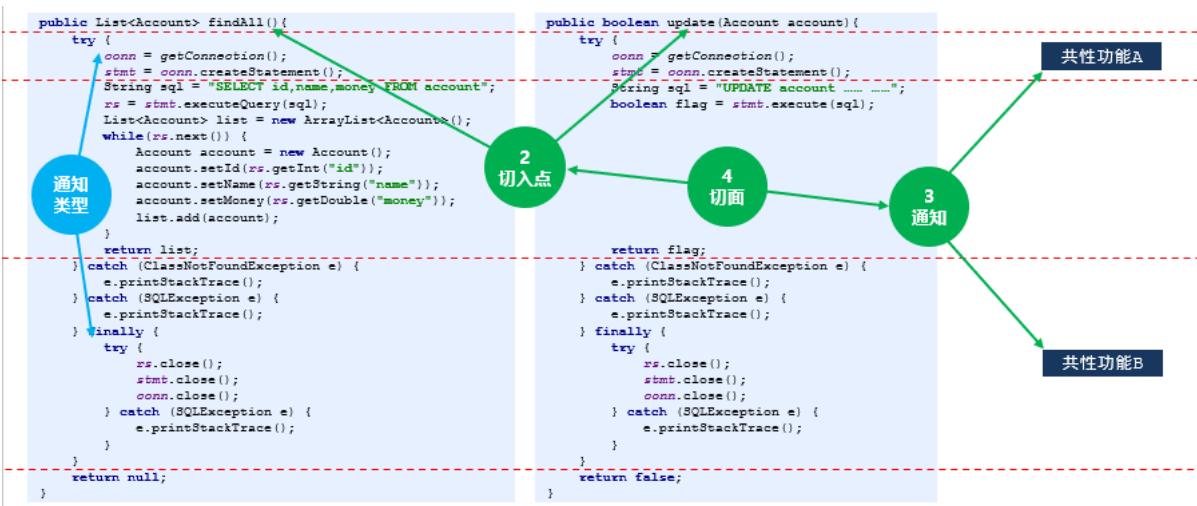
- 提高代码的可重用性
- 业务代码编码更简洁
- 业务代码维护更高效
- 业务功能扩展更便捷

核心概念

概念详解

- Joinpoint (连接点)：就是方法
- Pointcut (切入点)：就是挖掉共性功能的方法
- Advice (通知)：就是共性功能，最终以一个方法的形式呈现
- Aspect (切面)：就是共性功能与挖的位置的对应关系
- Target (目标对象)：就是挖掉功能的方法对应的类产生的对象，这种对象是无法直接完成最终工作的
- Weaving (织入)：就是将挖掉的功能回填的动态过程
- Proxy (代理)：目标对象无法直接完成工作，需要对其进行功能回填，通过创建原始对象的代理对象实现
- Introduction (引入/引介)：就是对原始对象无中生有的添加成员变量或成员方法





入门项目

开发步骤：

- 开发阶段
 - 制作程序
 - 将非共性功能开发到对应的目标对象类中，并制作成切入点方法
 - 将共性功能独立开发出来，制作成通知
 - 在配置文件中，声明切入点
 - 在配置文件中，声明切入点与通知间的关系（含通知类别），即切面
- 运行阶段（AOP 完成）
 - Spring 容器加载配置文件，监控所有配置的切入点方法的执行
 - 当监控到切入点方法被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置将通知对应的功能织入，完成完整的代码逻辑并运行

1. 导入坐标 pom.xml

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>

```

2. 业务层抽取通用代码 service / UserServiceImpl

```

public interface UserService {
    public void save();
}

```

```

public class UserServiceImpl implements UserService {
    @Override
    public void save() {
        //System.out.println("共性功能");
        System.out.println("user service running...");
    }
}

```

aop.AOPAdvice

```

//1.制作通知类，在类中定义一个方法用于完成共性功能
public class AOPAdvice {
    //共性功能抽取后职称独立的方法
    public void function(){
        System.out.println("共性功能");
    }
}

```

3.把通知加入spring容器管理，配置aop applicationContext.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           https://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/aop
           https://www.springframework.org/schema/aop/spring-aop.xsd
       ">
    <!--原始Spring控制资源-->
    <bean id="userService" class="service.impl.UserServiceImpl"/>
    <!--2.配置共性功能成功spring控制的资源-->
    <bean id="myAdvice" class="aop.AOPAdvice"/>
    <!--3.开启AOP命名空间：beans标签内-->
    <!--4.配置AOP-->
    <aop:config>
        <!--5.配置切入点-->
        <aop:pointcut id="pt" expression="execution(* *..*(..))"/>
        <!--6.配置切面（切入点与通知的关系）-->
        <aop:aspect ref="myAdvice">
            <!--7.配置具体的切入点对应通知中那个操作方法-->
            <aop:before method="function" pointcut-ref="pt"/>
        </aop:aspect>
    </aop:config>
</beans>

```

4. 测试类

```

public class App {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService) ctx.getBean("userService");
        userService.save(); //先输出共性功能，然后 user service running...
    }
}

```

XML开发

AspectJ

Aspect (切面) 用于描述切入点与通知间的关系，是 AOP 编程中的一个概念

AspectJ 是基于 java 语言对 Aspect 的实现

AOP

config

标签: [aop:config](#), 的子标签

作用: 设置 AOP

格式:

```
<beans>
    <aop:config>.....</aop:config>
    <aop:config>.....</aop:config>
    <!--一个beans标签中可以配置多个aop:config标签-->
</beans>
```

pointcut

标签: [aop:pointcut](#), 归属于 aop:config 标签和 aop:aspect 标签

作用: 设置切入点

格式:

```
<aop:config>
    <aop:pointcut id="pointcutId" expression="....."/>
    <aop:aspect>
        <aop:pointcut id="pointcutId" expression="....."/>
    </aop:aspect>
</aop:config>
```

说明:

- 一个 aop:config 标签中可以配置多个 aop:pointcut 标签, 且该标签可以配置在 aop:aspect 标签内

属性:

- id : 识别切入点的名称
- expression : 切入点表达式

aspect

标签: [aop:aspect](#), aop:config 的子标签

作用: 设置具体的 AOP 通知对应的切入点 (切面)

格式:

```
<aop:config>
    <aop:aspect ref="beanId">.....</aop:aspect>
    <aop:aspect ref="beanId">.....</aop:aspect>
    <!--一个aop:config标签中可以配置多个aop:aspect标签-->
</aop:config>
```

属性:

- ref : 通知所在的 bean 的 id

Pointcut

切入点

切入点描述的是某个方法

切入点表达式是一个快速匹配方法描述的通配格式, 类似于正则表达式

表达式

格式：

```
关键字(访问修饰符 返回值 包名.类名.方法名(参数)异常名)
```

示例：

```
//匹配UserService中只含有一个参数的findById方法  
execution(public User service.UserService.findById(int))
```

格式解析：

- 关键字：描述表达式的匹配模式（参看关键字列表）
- 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

关键字：

- execution：匹配执行指定方法
- args：匹配带有指定参数类型的方法
- within、this、target、@within、@target、@args、@annotation、bean、reference pointcut等

通配符：

- *：单个独立的任意符号，可以独立出现，也可以作为前缀或者后缀的匹配符出现

```
//匹配com.seazean包下的任意包中的UserService类或接口中所有find开头的带有一个任意参数的方法  
execution(public * com.seazean.*.UserService.find*(*))
```

- ..：多个连续的任意符号，可以独立出现，常用于简化包名与参数

```
//匹配com包下的任意包中的userService类或接口中所有名称为findById参数任意数量和类型的方法  
execution(public User com..UserService.findById(..))
```

- +：专用于匹配子类类型

```
//匹配任意包下的Service结尾的类或者接口的子类或者实现类  
execution(* *..*Service+.*(..))
```

逻辑运算符：

- &&：连接两个切入点表达式，表示两个切入点表达式同时成立的匹配
- ||：连接两个切入点表达式，表示两个切入点表达式成立任意一个的匹配
- !：连接单个切入点表达式，表示该切入点表达式不成立的匹配

示例：

```
execution(* *(..)) //前三个都是匹配全部  
execution(* *..*(..))  
execution(* *..*.*(..))  
execution(public * *..*.*(..))  
execution(public int *..*.*(..))  
execution(public void *..*.*(..))  
execution(public void com..*.*(..))  
execution(public void com..service.*.*(..))  
execution(public void com.seazean.service.*.*(..))  
execution(public void com.seazean.service.User*.*(..))  
execution(public void com.seazean.service.*Service.*(..))  
execution(public void com.seazean.service.UserService.*(..))  
execution(public User com.seazean.service.UserService.find*(..)) //find开头  
execution(public User com.seazean.service.UserService.*Id(..)) //I  
execution(public User com.seazean.service.UserService.findById(..))  
execution(public User com.seazean.service.UserService.findById(int))  
execution(public User com.seazean.service.UserService.findById(int,int))  
execution(public User com.seazean.service.UserService.findById(int,*))  
execution(public User com.seazean.service.UserService.findById())  
execution(List com.seazean.service.*Service+.*findAll(..))
```

配置方式

XML 配置规则：

- 企业开发命名规范严格遵循规范文档进行
- 先为方法配置局部切入点，再抽取类中公共切入点，最后抽取全局切入点
- 代码走查过程中检测切入点是否存在越界性包含
- 代码走查过程中检测切入点是否存在非包含性进驻
- 设定 AOP 执行检测程序，在单元测试中监控通知被执行次数与预计次数是否匹配（不绝对正确：加进一个不该加的，删去一个不该删的相当于结果不变）

- 设定完毕的切入点如果发生调整务必进行回归测试

```
<aop:config>
    <!--1.配置公共切入点-->
    <aop:pointcut id="pt1" expression="execution(* *(..))"/>
    <aop:aspect ref="myAdvice">
        <!--2.配置局部切入点-->
        <aop:pointcut id="pt2" expression="execution(* *(..))"/>
        <!--引用公共切入点-->
        <aop:before method="logAdvice" pointcut-ref="pt1"/>
        <!--引用局部切入点-->
        <aop:before method="logAdvice" pointcut-ref="pt2"/>
        <!--3.直接配置切入点-->
        <aop:before method="logAdvice" pointcut="execution(* *(..))"/>
    </aop:aspect>
</aop:config>
```

Advice

通知类型

AOP 的通知类型共5种：前置通知，后置通知、返回后通知、抛出异常后通知、环绕通知

before

标签: [aop:before](#), aop:aspect的子标签

作用：设置前置通知

- **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
- 应用：数据校验

格式：

```
<aop:aspect ref="adviceId">
    <aop:before method="methodName" pointcut="execution(* *(..))"/>
    <!--一个aop:aspect标签中可以配置多个aop:before标签-->
</aop:aspect>
```

基本属性：

- method: 在通知类中设置当前通知类别对应的方法
- pointcut: 设置当前通知对应的切入点表达式，与pointcut-ref属性冲突
- pointcut-ref: 设置当前通知对应的切入点id，与pointcut属性冲突

after

标签: [aop:after](#), aop:aspect的子标签

作用：设置后置通知

- **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
- 应用：现场清理

格式：

```
<aop:aspect ref="adviceId">
    <aop:after method="methodName" pointcut="execution(* *(..))"/>
    <!--一个aop:aspect标签中可以配置多个aop:after标签-->
</aop:aspect>
```

基本属性：

- method: 在通知类中设置当前通知类别对应的方法
- pointcut: 设置当前通知对应的切入点表达式，与pointcut-ref属性冲突
- pointcut-ref: 设置当前通知对应的切入点id，与pointcut属性冲突

after-r

标签: [aop:after-returning](#), aop:aspect的子标签

作用：设置返回后通知

- **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
- 应用：返回值相关数据处理

格式：

```
<aop:aspect ref="adviceId">
    <aop:after-returning method="methodName" pointcut="execution(* *(..))"/>
    <!--一个aop:aspect标签中可以配置多个aop:after-returning标签-->
</aop:aspect>
```

基本属性：

- method：在通知类中设置当前通知类别对应的方法
- pointcut：设置当前通知对应的切入点表达式，与pointcut-ref属性冲突
- pointcut-ref：设置当前通知对应的切入点id，与pointcut属性冲突
- returning：设置接受返回值的参数，与通知类中对应方法的参数一致

after-t

标签：[aop:after-throwing](#), aop:aspect的子标签

作用：设置抛出异常后通知

- **抛出异常后通知**：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
- 应用：对原始方法中出现的异常信息进行处理

格式：

```
<aop:aspect ref="adviceId">
    <aop:after-throwing method="methodName" pointcut="execution(* *(..))"/>
    <!--一个aop:aspect标签中可以配置多个aop:after-throwing标签-->
</aop:aspect>
```

基本属性：

- method：在通知类中设置当前通知类别对应的方法
- pointcut：设置当前通知对应的切入点表达式，与pointcut-ref属性冲突
- pointcut-ref：设置当前通知对应的切入点id，与pointcut属性冲突
- throwing：设置接受异常对象的参数，与通知类中对应方法的参数一致

around

标签：[aop:around](#), aop:aspect的子标签

作用：设置环绕通知

- **环绕通知**：在原始方法执行前后均有对应执行执行，还可以阻止原始方法的执行
- 应用：功能强大，可以做任何事情

格式：

```
<aop:aspect ref="adviceId">
    <aop:around method="methodName" pointcut="execution(* *(..))"/>
    <!--一个aop:aspect标签中可以配置多个aop:around标签-->
</aop:aspect>
```

基本属性：

- method：在通知类中设置当前通知类别对应的方法
- pointcut：设置当前通知对应的切入点表达式，与pointcut-ref属性冲突
- pointcut-ref：设置当前通知对应的切入点id，与pointcut属性冲突

环绕通知的开发方式（参考通知顺序章节）：

- 环绕通知是在原始方法的前后添加功能，在环绕通知中，存在对原始方法的显式调用

```
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    Object ret = pjp.proceed();
    return ret;
}
```

- 环绕通知方法相关说明：

- 方法须设定 Object 类型的返回值，否则会拦截原始方法的返回。如果原始方法返回值类型为 void，通知方法也可以设定返回值类型为 void，最终返回 null
- 方法需在第一个参数位置设定 ProceedingJoinPoint 对象，通过该对象调用 proceed() 方法，实现对原始方法的调用。如省略该参数，原始方法将无法执行
- 使用 proceed() 方法调用原始方法时，因无法预知原始方法运行过程中是否会出现异常，强制抛出 Throwable 对象，封装原始方法中可能出现的异常信息

通知顺序

当同一个切入点配置了多个通知时，通知会存在运行的先后顺序，该顺序以通知配置的顺序为准。

- AOPAdvice

```
public class AOPAdvice {
    public void before(){
        System.out.println("before...");
```

```

    }
    public void after(){
        System.out.println("after...");
    }
    public void afterReturning(){
        System.out.println("afterReturning...");
    }
    public void afterThrowing(){
        System.out.println("afterThrowing...");
    }
    public Object around(ProceedingJoinPoint pjp) {
        System.out.println("around before...");
        //对原始方法的调用
        Object ret = pjp.proceed();
        System.out.println("around after..." + ret);
        return ret;
    }
}

```

- o applicationContext.xml 顺序执行

```

<aop:config>
    <aop:pointcut id="pt" expression="execution(* *..*(..))"/>
    <aop:aspect ref="myAdvice">
        <aop:before method="before" pointcut-ref="pt"/>
        <aop:after method="after" pointcut-ref="pt"/>
        <aop:after-returning method="afterReturning" pointcut-ref="pt"/>
        <aop:after-throwing method="afterThrowing" pointcut-ref="pt"/>
        <aop:around method="around" pointcut-ref="pt"/>
    </aop:aspect>
</aop:config>

```

获取数据

参数

第一种方式:

- o 设定通知方法第一个参数为 JoinPoint，通过该对象调用 getArgs() 方法，获取原始方法运行的参数数组

```

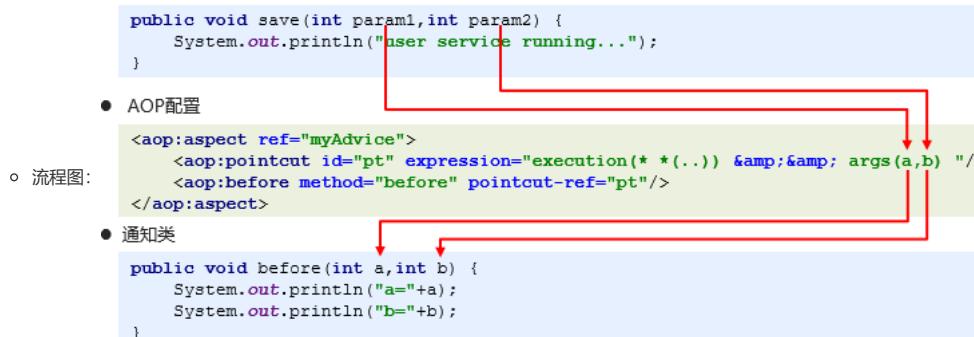
public void before(JoinPoint jp) throws Throwable {
    Object[] args = jp.getArgs();
}

```

- o 所有的通知均可以获取参数，环绕通知使用 ProceedingJoinPoint.getArgs() 方法

第二种方式:

- o 设定切入点表达式为通知方法传递参数（锁定通知变量名）



- o 解释:

- & 代表并且 &
- 输出结果: a = param1 b = param2

第三种方式:

- o 设定切入点表达式为通知方法传递参数（改变通知变量名的定义顺序）

```
public void save(int param1,int param2) {  
    System.out.println("user service running...");  
}
```

AOP配置

```
<aop:aspect ref="myAdvice">  
    <aop:pointcut id="pt" expression="execution(* *(..)) && args(a,b)" />  
    <aop:before method="before" pointcut-ref="pt" arg-names="b,a"/>  
</aop:aspect>  
  
通知类  
public void before(int a,int b) {  
    System.out.println("a=" + a);  
    System.out.println("b=" + b);  
}
```

解释：输出结果 a = param2 b = param1

改变参数转入通知的顺序

返回值

环绕通知和返回后通知可以获取返回值，后置通知不一定，其他类型获取不到

第一种方式：适用于返回后通知 (after-returning)

- 设定返回值变量名
- 原始方法：

```
public class UserServiceImpl implements UserService {  
    @Override  
    public int save() {  
        System.out.println("user service running...");  
        return 100;  
    }  
}
```

◦ AOP 配置：

```
<aop:aspect ref="myAdvice">  
    <aop:pointcut id="pt" expression="execution(* *(..))"/>  
    <aop:after-returning method="afterReturning" pointcut-ref="pt" returning="ret"/>  
</aop:aspect>
```

◦ 通知类：

```
public class AOPAdvice {  
    public void afterReturning(Object ret) {  
        System.out.println("return:" + ret);  
    }  
}
```

第二种：适用于环绕通知 (around)

- 在通知类的方法中调用原始方法获取返回值
- 原始方法：

```
public class UserServiceImpl implements UserService {  
    @Override  
    public int save() {  
        System.out.println("user service running...");  
        return 100;  
    }  
}
```

◦ AOP 配置：

```
<aop:aspect ref="myAdvice">  
    <aop:pointcut id="pt" expression="execution(* *(..))" />  
    <aop:around method="around" pointcut-ref="pt" />  
</aop:aspect>
```

◦ 通知类：

```

public class AOPAdvice {
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        Object ret = pjp.proceed();
        return ret;
    }
}

```

- 测试类:

```

public class App {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService) ctx.getBean("userService");
        int ret = userService.save();
        System.out.println("app....." + ret);
    }
}

```

异常

环绕通知和抛出异常后通知可以获取异常，后置通知不一定，其他类型获取不到

第一种：适用于返回后通知 (after-throwing)

- 设定异常对象变量名
- 原始方法

```

public class UserServiceImpl implements UserService {
    @Override
    public void save() {
        System.out.println("user service running...");
        int i = 1/0;
    }
}

```

- AOP 配置

```

<aop:aspect ref="myAdvice">
    <aop:pointcut id="pt" expression="execution(* *(..)) "/>
    <aop:after-throwing method="afterThrowing" pointcut-ref="pt" throwing="t"/>
</aop:aspect>

```

- 通知类

```

public void afterThrowing(Throwable t){
    System.out.println(t.getMessage());
}

```

第二种：适用于环绕通知 (around)

- 在通知类的方法中调用原始方法捕获异常
- 原始方法：

```

public class UserServiceImpl implements UserService {
    @Override
    public void save() {
        System.out.println("user service running...");
        int i = 1/0;
    }
}

```

- AOP 配置：

```

<aop:aspect ref="myAdvice">
    <aop:pointcut id="pt" expression="execution(* *(..)) "/>
    <aop:around method="around" pointcut-ref="pt" />
</aop:aspect>

```

- 通知类：try.....catch.....捕获异常后，ret为null

```

public Object around(ProceedingJoinPoint pjp) throws Throwable {
    Object ret = pjp.proceed(); //对此处调用进行try.....catch.....捕获异常，或抛出异常
    /* try {
        ret = pjp.proceed();
    } catch (Throwable throwable) {
        System.out.println("around exception..." + throwable.getMessage());
    }*/
    return ret;
}

```

- 测试类

```
userservice.delete();
```

获取全部

- UserService

```

public interface userservice {
    public void save(int i, int m);

    public int update();

    public void delete();
}

```

```

public class userServiceImpl implements userService {
    @Override
    public void save(int i, int m) {
        System.out.println("user service running..." + i + "," + m);
    }

    @Override
    public int update() {
        System.out.println("user service update running...");
        return 100;
    }

    @Override
    public void delete() {
        System.out.println("user service delete running...");
        int i = 1 / 0;
    }
}

```

- AOPAdvice

```

public class AOPAdvice {
    public void before(JoinPoint jp){
        //通过JoinPoint参数获取调用原始方法所携带的参数
        Object[] args = jp.getArgs();
        System.out.println("before..."+args[0]);
    }

    public void after(JoinPoint jp){
        Object[] args = jp.getArgs();
        System.out.println("after..."+args[0]);
    }

    public void afterReturning(Object ret){
        System.out.println("afterReturning..."+ret);
    }

    public void afterThrowing(Throwable t){
        System.out.println("afterThrowing..."+t.getMessage());
    }

    public Object around(ProceedingJoinPoint pjp) {
        System.out.println("around before...");
        Object ret = null;
        try {
            //对原始方法的调用
            ret = pjp.proceed();
        } catch (Throwable throwable) {
            System.out.println("around...exception...."+throwable.getMessage());
        }
    }
}

```

```

        }
        System.out.println("around after..."+ret);
        return ret;
    }
}

```

- o applicationContext.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
    ">
<bean id="userService" class="service.impl.UserServiceImpl"/>
<bean id="myAdvice" class="aop.AOPAdvice"/>

<aop:config>
    <aop:pointcut id="pt" expression="execution(* *..*(..))"/>
    <aop:aspect ref="myAdvice">
        <aop:before method="before" pointcut="pt"/>
        <aop:around method="around" pointcut-ref="pt"/>
        <aop:after method="after" pointcut="pt"/>
        <aop:after-returning method="afterReturning" pointcut-ref="pt" returning="ret"/>
        <aop:after-throwing method="afterThrowing" pointcut-ref="pt" throwing="t"/>
    </aop:aspect>
</aop:config>
</beans>

```

- o 测试类

```

public class App {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService) ctx.getBean("userService");
        //        userService.save(666, 888);
        //        int ret = userService.update();
        //        System.out.println("app....." + ret);
        userService.delete();
    }
}

```

注解开发

AOP注解

AOP注解简化XML:

```

<!--开启AOP注解驱动支持-->
<aop:aspectj-autoproxy/>
<!--原始的功能要配置成spring控制的资源-->
<bean id="userService" class="....."/>
<!--抽取的功能要配置成spring控制的资源-->
<bean id="myAdvice" class="com.itheima.aop.AopAdvice"/>
<!--aop配置-->
<aop:config>
    <!--配置切入点-->
    <aop:aspect ref="myAdvice">
        <!--配置方法-->
        <aop:pointcut id="pt" expression="execution(* *..*(..))"/>
        <!--切入点和通知的关系-->
        <aop:before method="before" pointcut-ref="pt"/>
    </aop:aspect>
</aop:config>

```

```

@Component
@Aspect
public class AopAdvice {

    @Pointcut("execution(* *..*(..))")
    public void pt() {
    }

    @Before("pt()")
    public void before() {
        System.out.println("before");
    }
}

```

注意事项:

1. 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
2. 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
3. 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

4. 可以在通知类型注解后添加参数，实现 XML 配置中的属性，例如 after-returning 后的 returning 属性

启动注解

XML

开启 AOP 注解支持：

```
<aop:aspectj-autoproxy/>
<context:component-scan base-package="aop,config,service"/><!--启动Spring扫描-->
```

开发步骤：

1. 导入坐标（伴随 spring-context 坐标导入已经依赖导入完成）
2. 开启 AOP 注解支持
3. 配置切面 @Aspect
4. 定义专用的切入点方法，并配置切入点 @Pointcut
5. 为通知方法配置通知类型及对应切入点 @Before

纯注解

注解：@EnableAspectJAutoProxy

位置：Spring 注解配置类定义上方

作用：设置当前类开启 AOP 注解驱动的支持，加载 AOP 注解

格式：

```
@Configuration
@ComponentScan("com.seazean")
@EnableAspectJAutoProxy
public class SpringConfig { }
```

基本注解

Aspect

注解：@Aspect

位置：类定义上方

作用：设置当前类为切面类

格式：

```
@Aspect
public class AopAdvice { }
```

Pointcut

注解：@Pointcut

位置：方法定义上方

作用：使用当前方法名作为切入点引用名称

格式：

```
@Pointcut("execution(* *(..))")
public void pt() { }
```

说明：被修饰的方法忽略其业务功能，格式设定为无参无返回值的方法，方法体内空实现（非抽象）

Before

注解: @Before

位置: 方法定义上方

作用: 标注当前方法作为前置通知

格式:

```
@Before("pt()")
public void before(JoinPoint joinPoint){
    //joinPoint.getArgs();
}
```

注意: 多个参数时, JoinPoint参数一定要在第一位

After

注解: @After

位置: 方法定义上方

作用: 标注当前方法作为后置通知

格式:

```
@After("pt()")
public void after(){
}
```

AfterR

注解: @AfterReturning

位置: 方法定义上方

作用: 标注当前方法作为返回后通知

格式:

```
@AfterReturning(value="pt()", returning = "result")
public void afterReturning(Object result) {
}
```

特殊参数:

- returning : 设定使用通知方法参数接收返回值的变量名

AfterT

注解: @AfterThrowing

位置: 方法定义上方

作用: 标注当前方法作为异常后通知

格式:

```
@AfterThrowing(value="pt()", throwing = "t")
public void afterThrowing(Throwable t){
}
```

特殊参数:

- 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

Around

注解: @Around

位置: 方法定义上方

作用: 标注当前方法作为环绕通知

格式:

```
@Around("pt()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    Object ret = pjp.proceed();
    return ret;
}
```

执行顺序

AOP 使用 XML 配置情况下，通知的执行顺序由配置顺序决定，在注解情况下由于不存在配置顺序的概念，参照通知所配置的方法名字符串对应的编码值顺序，可以简单理解为字母排序

- 同一个通知类中，相同通知类型以方法名排序为准

```
@Before("aop.AOPPointcut.pt()")
public void aop001Log() {}

@Before("aop.AOPPointcut.pt()")
public void aop002Exception() {}
```

- 不同通知类中，以类名排序为准

- 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
@Aspect
@Order(1) //先执行
public class AOPAdvice2 { }
```

```
@Component
@Aspect
@Order(2)
public class AOPAdvice1 { //默认执行此通知 }
```

AOP 原理

静态代理

装饰者模式 (Decorator Pattern)：在不惊动原始设计的基础上，为其添加功能

```
public class UserServiceDecorator implements UserService {
    private UserService userService;

    public UserServiceDecorator(UserService userService) {
        this.userService = userService;
    }

    public void save() {
        //原始调用
        userService.save();
        //增强功能（后置）
        System.out.println("后置增强功能");
    }
}
```

Proxy

JDKProxy 动态代理是针对对象做代理，要求原始对象具有接口实现，并对接口方法进行增强，因为代理类继承 Proxy

静态代理和动态代理的区别：

- 静态代理是在编译时就已经将接口、代理类、被代理类的字节码文件确定下来
- 动态代理是程序在运行后通过反射创建字节码文件交由 JVM 加载

```
public class UserServiceJDKProxy {
    public static UserService createUserServiceJDKProxy(UserService userService) {
        UserService service = (UserService) Proxy.newProxyInstance(
            userService.getClass().getClassLoader(), //获取被代理对象的类加载器
            userService.getClass().getInterfaces(), //获取被代理对象实现的接口
            new InvocationHandler() { //对原始方法执行进行拦截并增强
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                    if (method.getName().equals("save")) {
                        System.out.println("前置增强");
                        Object ret = method.invoke(userService, args);
                        return ret;
                    }
                }
            }
        );
    }
}
```

```

        System.out.println("后置增强");
        return ret;
    }
    return null;
}
);
return service;
}
}

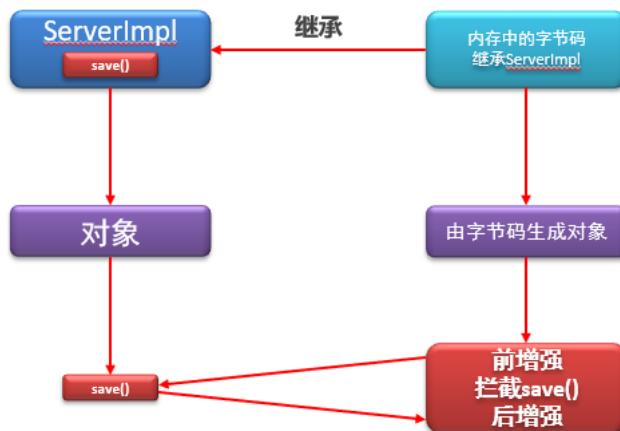
```

CGLIB

CGLIB (Code Generation Library) : Code 生成类库

CGLIB 特点:

- CGLIB 动态代理**不限定**是否具有接口，可以对任意操作进行增强
- CGLIB 动态代理无需要原始被代理对象，动态创建出新的代理对象
- CGLIB **继承被代理类**，如果代理类是 final 则不能实现



- CGLIB 类

- JDKProxy 仅对接口方法做增强，CGLIB 对所有方法做增强，包括 Object 类中的方法 (toString、hashCode)
- 返回值类型采用多态向下转型，所以需要设置父类类型

需要对方法进行判断是否是 save，来选择性增强

```

public class UserServiceImplCglibProxy {
    public static UserService createUserServiceCglibProxy(Class cls){
        //1. 创建Enhancer对象（可以理解为内存中动态创建了一个类的字节码）
        Enhancer enhancer = new Enhancer();

        //2. 设置Enhancer对象的父类是指定类型UserServerImpl
        enhancer.setSuperclass(cls);

        //3. 设置回调方法
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object o, Method m, Object[] args, MethodProxy mp) throws Throwable {
                //o是被代理出的类创建的对象，所以使用MethodProxy调用，并且是调用父类
                //通过调用父类的方法实现对原始方法的调用
                Object ret = methodProxy.invokeSuper(o, args);
                //后置增强内容，需要判断是否都是save方法
                if (method.getName().equals("save")) {
                    System.out.println("I love Java");
                }
                return ret;
            }
        });
        //使用Enhancer对象创建对应的对象
        return (UserService)enhancer.create();
    }
}

```

- Test类

```

public class App {
    public static void main(String[] args) {
        UserService userService = UserServiceCglibProxy.createUserServiceCglibProxy(UserServiceImpl.class);
        userService.save();
    }
}

```

代理选择

Spring 可以通过配置的形式控制使用的代理形式，Spring 会先判断是否实现了接口，如果实现了接口就使用 JDK 动态代理，如果没有实现接口则使用 CGLIB 动态代理，通过配置可以修改为使用 CGLIB

- XML 配置

```

<!--XML配置AOP-->
<aop:config proxy-target-class="false"></aop:config>

```

- XML 注解支持

```

<!--注解配置AOP-->
<aop:aspectj-autoproxy proxy-target-class="false"/>

```

- 注解驱动

```

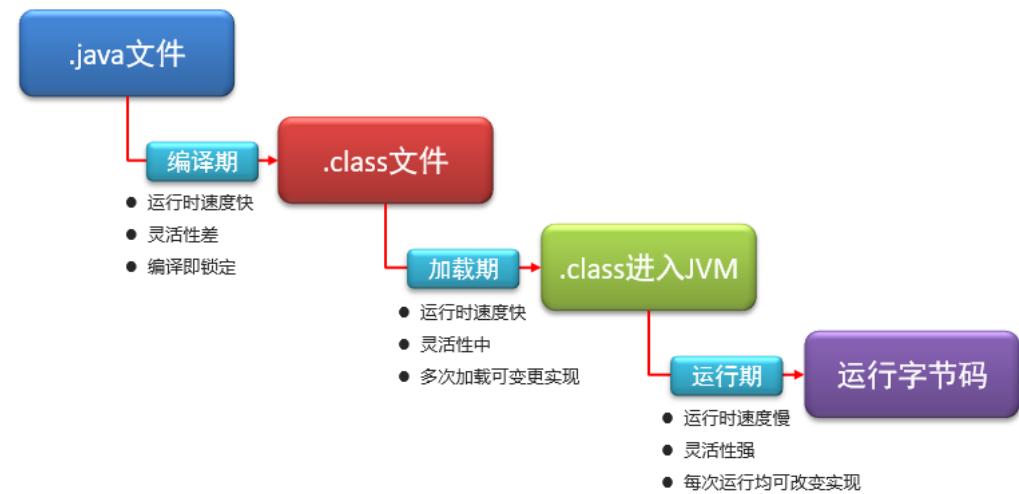
//修改为使用 cglb 创建代理对象
@EnableAspectJAutoProxy(proxyTargetClass = true)

```

- JDK 动态代理和 CGLIB 动态代理的区别：

- JDK 动态代理只能对实现了接口的类生成代理，没有实现接口的类不能使用。
- CGLIB 动态代理即使被代理的类没有实现接口也可以使用，因为 CGLIB 动态代理是使用继承被代理类的方式进行扩展
- CGLIB 动态代理是通过继承的方式，覆盖被代理类的方法来进行代理，所以如果方法是被 final 修饰的话，就不能进行代理

织入时机



事务

事务机制

事务介绍

事务：数据库中多个操作合并在一起形成的操作序列，事务特征（ACID）

作用：

- 当数据库操作序列中个别操作失败时，提供一种方式使数据库状态恢复到正常状态（A），保障数据库即使在异常状态下仍能保持数据一致性（C）（要么操作前状态，要么操作后状态）
- 当出现并发访问数据库时，在多个访问间进行相互隔离，防止并发访问操作结果互相干扰（I）

Spring 事务一般加到业务层，对应着业务的操作，Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，Spring 是无法提供事务功能的，Spring 只提供统一事务管理接口

Spring 在事务开始时，根据当前环境中设置的隔离级别，调整数据库隔离级别，由此保持一致。程序是否支持事务首先取决于数据库，比如 MySQL，如果是 InnoDB 引擎，是支持事务的；如果 MySQL 使用 MyISAM 引擎，那从根上就是不支持事务的

保证原子性：

- 要保证事务的原子性，就需要在异常发生时，对已经执行的操作进行回滚
- 在 MySQL 中，恢复机制是通过回滚日志（undo log）实现，所有事务进行的修改都会先记录到这个回滚日志中，然后再执行相关的操作。如果执行过程中遇到异常的话，直接利用回滚日志中的信息将数据回滚到修改之前的样子即可
- 回滚日志会先于数据持久化到磁盘上，这样保证了即使遇到数据库突然宕机等情况，当用户再次启动数据库的时候，数据库还能够通过查询回滚日志来回滚将之前未完成的事务

隔离级别

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

- TransactionDefinition.ISOLATION_DEFAULT：使用后端数据库默认的隔离级别，MySQL 默认采用的 REPEATABLE_READ 隔离级别，Oracle 默认采用的 READ_COMMITTED 隔离级别。
- TransactionDefinition.ISOLATION_READ_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
- TransactionDefinition.ISOLATION_READ_COMMITTED：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
- TransactionDefinition.ISOLATION_REPEATABLE_READ：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- TransactionDefinition.ISOLATION_SERIALIZABLE：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别

MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读）

分布式事务：允许多个独立的事务资源（transactional resources）参与到一个全局的事务中。事务资源通常是关系型数据库系统，但也可以是其他类型的资源，全局事务要求在其中的所有参与的事务要么都提交，要么都回滚，这对于事务原有的 ACID 要求又有了提高

在使用分布式事务时，InnoDB 存储引擎的事务隔离级别必须设置为 SERIALIZABLE

传播行为

事务传播行为是为了解决业务层方法之间互相调用的事务问题，也就是方法嵌套：

- 当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。
- 例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行

```
//外层事务 Service A 的 aMethod 调用内层 Service B 的 bMethod
class A {
    @Transactional(propagation=propagation.xxx)
    public void aMethod {
        B b = new B();
        b.bMethod();
    }
}
class B {
    @Transactional(propagation=propagation.xxx)
    public void bMethod {}
}
```

支持当前事务的情况：

- TransactionDefinition.PROPAGATION_REQUIRED：如果当前存在事务则加入该事务；如果当前没有事务则创建一个新的事务
 - 内外层是相同的事务，在 aMethod 或者在 bMethod 内的任何地方出现异常，事务都会被回滚
 - 工作流程：
 - 线程执行到 serviceA.aMethod() 时，其实是执行的代理 serviceA 对象的 aMethod
 - 首先执行事务增强器逻辑（环绕增强），提取事务标签属性，检查当前线程是否绑定 connection 数据库连接资源，没有就调用 datasource.getConnection()，设置事务提交为手动提交 autocommit(false)
 - 执行其他增强器的逻辑，然后调用 target 的目标方法 aMethod() 方法，进入 serviceB 的逻辑
 - serviceB 也是先执行事务增强器的逻辑，提取事务标签属性，但此时会检查到线程绑定了 connection，检查注解的传播属性，所以调用 DataSourceUtils.getConnection(datasource) 共享该连接资源，执行完相关的增强和 SQL 后，发现事务并不是当前方法开启的，可以直接返回上层

- serviceA.aMethod() 继续执行，执行完增强后进行提交事务或回滚事务
- TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行
- TransactionDefinition.PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常

不支持当前事务的情况：

- TransactionDefinition.PROPAGATIONQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起
 - 内外层是不同的事务，如果 bMethod 已经提交，如果 aMethod 失败回滚，bMethod 不会回滚
 - 如果 bMethod 失败回滚，ServiceB 抛出的异常被 ServiceA 捕获，如果 B 抛出的异常是 A 会回滚的异常，aMethod 事务需要回滚，否则仍然可以提交
- TransactionDefinition.PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起
- TransactionDefinition.PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常

其他情况：

- TransactionDefinition.PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务（两个事务没有关系）来运行
 - 如果 ServiceB 异常回滚，可以通过 try-catch 机制执行 ServiceC
 - 如果 ServiceB 提交，ServiceA 可以根据具体的配置决定是 commit 还是 rollback
 - 应用场景：在查询数据的时候要向数据库中存储一些日志，系统不希望存日志的行为影响到主逻辑，可以使用该传播

required：必须的、supports：支持的、mandatory：强制的、nested：嵌套的

超时属性

事务超时，指一个事务所允许执行的最长时间，如果超过该时间限制事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒，默认值为 -1

只读属性

对于只有读取数据查询的事务，可以指定事务类型为 readonly，即只读事务；只读事务不涉及数据的修改，数据库会提供一些优化手段，适合用在有多条数据库查询操作的方法中

读操作为什么需要启用事务支持：

- MySQL 默认对每一个新建立的连接都启用了 autocommit 模式，在该模式下，每一个发送到 MySQL 服务器的 SQL 语句都会在一个单独的事务中进行处理，执行结束后会自动提交事务，并开启一个新的事务
 - 执行多条查询语句，如果方法加上了 @Transactional 注解，这个方法执行的所有 SQL 会被放在一个事务中，如果声明了只读事务的话，数据库就会去优化它的执行，并不会带来其他的收益。如果不加 @Transactional，每条 SQL 会开启一个单独的事务，中间被其它事务修改了数据，比如在前条 SQL 查询之后，后条 SQL 查询之前，数据被其他用户改变，则这次整体的统计查询将会出现读数据不一致的状态
-

核心对象

事务对象

J2EE 开发使用分层设计的思想进行，对于简单的业务层转调数据层的单一操作，事务开启在业务层或者数据层并无太大差别，当业务中包含多个数据层的调用时，需要在业务层开启事务，对数据层中多个操作进行组合并归属于同一个事务进行处理

Spring 为业务层提供了整套的事务解决方案：

- PlatformTransactionManager
 - TransactionDefinition
 - TransactionStatus
-

PTM

PlatformTransactionManager，平台事务管理器实现类：

- DataSourceTransactionManager 适用于 Spring JDBC 或 MyBatis
- HibernateTransactionManager 适用于 Hibernate3.0 及以上版本
- JpaTransactionManager 适用于 JPA
- JdoTransactionManager 适用于 JDO
- JtaTransactionManager 适用于 JTA

管理器：

- JPA (Java Persistence API) Java EE 标准之一，为 POJO 提供持久化标准规范，并规范了持久化开发的统一 API，符合 JPA 规范的开发可以在不同的 JPA 框架下运行

非持久化一个字段：

```

static String transient1; // not persistent because of static
final String transient2 = "Satish"; // not persistent because of final
transient String transient3; // not persistent because of transient
@Transient
String transient4; // not persistent because of @Transient

```

◦ JDO (Java Data Object) 是 Java 对象持久化规范，用于存取某种数据库中的对象，并提供标准化 API。JDBC 仅针对关系数据库进行操作，JDO 可以扩展到关系数据库、XML、对象数据库等，可移植性更强

◦ JTA (Java Transaction API) Java EE 标准之一，允许应用程序执行分布式事务处理。与 JDBC 相比，JDBC 事务则被限定在一个单一的数据库连接，而一个 JTA 事务可以有多个参与者，比如 JDBC 连接、JDO 都可以参与到一个 JTA 事务中

此接口定义了事务的基本操作：

方法	说明
TransactionStatus getTransaction(TransactionDefinition definition)	获取事务
void commit(TransactionStatus status)	提交事务
void rollback(TransactionStatus status)	回滚事务

Definition

TransactionDefinition 此接口定义了事务的基本信息：

方法	说明
String getName()	获取事务定义名称
boolean isReadOnly()	获取事务的读写属性
int getIsolationLevel()	获取事务隔离级别
int getTimeout()	获取事务超时时间
int getPropagationBehavior()	获取事务传播行为特征

Status

TransactionStatus 此接口定义了事务在执行过程中某个时间点上的状态信息及对应的状态操作：

方法	说明
boolean isNewTransaction()	获取事务是否处于新开始事务状态
void flush()	刷新事务状态
boolean isCompleted()	获取事务是否处于已完成状态
boolean hasSavepoint()	获取事务是否具有回滚储存点
boolean isRollbackOnly()	获取事务是否处于回滚状态
void setRollbackOnly()	设置事务处于回滚状态

编程式

控制方式

编程式、声明式 (XML) 、声明式 (注解)

环境准备

银行转账业务

- 包装类

```
public class Account implements Serializable {
    private Integer id;
    private String name;
    private Double money;
    ....
}
```

- DAO层接口: AccountDao

```
public interface AccountDao {
    //入账操作 name:入账用户名 money:入账金额
    void inMoney(@Param("name") String name, @Param("money") Double money);

    //出账操作 name:出账用户名 money:出账金额
    void outMoney(@Param("name") String name, @Param("money") Double money);
}
```

- 业务层接口提供转账操作: AccountService

```
public interface AccountService {
    //转账操作 outName:出账用户名 inName:入账用户名 money:转账金额
    public void transfer(String outName, String inName, Double money);
}
```

- 业务层实现提供转账操作: AccountServiceImpl

```
public class AccountServiceImpl implements AccountService {
    private AccountDao accountDao;
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
    @Override
    public void transfer(String outName, String inName, Double money) {
        accountDao.inMoney(outName, money);
        accountDao.outMoney(inName, money);
    }
}
```

- 映射配置文件: dao / AccountDao.xml

```
<mapper namespace="dao.AccountDao">
    <update id="inMoney">
        UPDATE account SET money = money + #{money} WHERE name = #{name}
    </update>

    <update id="outMoney">
        UPDATE account SET money = money - #{money} WHERE name = #{name}
    </update>
</mapper>
```

- jdbc.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://192.168.2.185:3306/spring_db
jdbc.username=root
jdbc.password=1234
```

- 核心配置文件: applicationContext.xml

```
<context:property-placeholder location="classpath:*.properties"/>

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${jdbc.driver}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>

<bean id="accountService" class="service.impl.AccountServiceImpl">
    <property name="accountDao" ref="accountDao" />
</bean>

<bean class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="typeAliasesPackage" value="domain" />
</bean>
<!--扫描映射配置和Dao-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="dao" />

```

```
</bean>
```

- 测试类

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("ap...xml");
AccountService accountService = (AccountService) ctx.getBean("accountService");
accountService.transfer("Jock1", "Jock2", 100d);
```

编程式

编程式事务就是代码显式的给出事务的开启和提交

- 修改业务层实现提供转账操作：AccountServiceimpl

```
public void transfer(String outName, String inName, Double money){
    //1. 创建事务管理器,
    DataSourceTransactionManager dstm = new DataSourceTransactionManager();
    //2. 为事务管理器设置与数据层相同的数据源
    dstm.setDataSource(dataSource);
    //3. 创建事务定义对象
    TransactionDefinition td = new DefaultTransactionDefinition();
    //4. 创建事务状态对象, 用于控制事务执行, 【开启事务】
    TransactionStatus ts = dstm.getTransaction(td);
    accountDao.inMoney(inName, money);
    int i = 1/0;      //模拟业务层事务过程中出现错误
    accountDao.outMoney(outName, money);
    //5. 提交事务
    dstm.commit(ts);
}
```

- 配置 applicationContext.xml

```
<!--添加属性注入-->
<bean id="accountService" class="service.impl.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

AOP改造

- 将业务层的事务处理功能抽取出来制作成 AOP 通知，利用环绕通知运行期动态织入

```
public class TxAdvice {
    private DataSource dataSource;
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Object tx(ProceedingJoinPoint pjp) throws Throwable {
        //开启事务
        PlatformTransactionManager ptm = new DataSourceTransactionManager(dataSource);
        //事务定义
        TransactionDefinition td = new DefaultTransactionDefinition();
        //事务状态
        TransactionStatus ts = ptm.getTransaction(td);
        //pjp.getArgs() 标准写法, 也可以不加, 同样可以传递参数
        Object ret = pjp.proceed(pjp.getArgs());

        //提交事务
        ptm.commit(ts);

        return ret;
    }
}
```

- 配置 applicationContext.xml，要开启 AOP 空间

```
<!--修改bean的属性注入-->
<bean id="accountService" class="service.impl.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"/>
</bean>
```

```

<!--配置AOP通知类，并注入dataSource-->
<bean id="txAdvice" class="aop.TxAdvice">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!--使用环绕通知将通知类织入到原始业务对象执行过程中-->
<aop:config>
    <aop:pointcut id="pt" expression="execution(* *..transfer(..))"/>
    <aop:aspect ref="txAdvice">
        <aop:around method="tx" pointcut-ref="pt"/>
    </aop:aspect>
</aop:config>

```

- 修改业务层实现提供转账操作：AccountServiceImpl

```

public class AccountServiceImpl implements AccountService {
    private AccountDao accountDao;
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
    @Override
    public void transfer(String outName, String inName, Double money) {
        accountDao.inMoney(outName, money);
        //int i = 1 / 0;
        accountDao.outMoney(inName, money);
    }
}

```

声明式

XML

tx使用

删除 TxAdvice 通知类，开启 tx 命名空间，配置 applicationContext.xml

```

<!--配置平台事务管理器-->
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!--定义事务管理的通知类-->
<tx:advice id="txAdvice" transaction-manager="txManager">
    <!--定义控制的事务-->
    <tx:attributes>
        <tx:method name="transfer" read-only="false"/>
    </tx:attributes>
</tx:advice>

<!--使用aop:advisor在AOP配置中引用事务专属通知类，底层invoke调用-->
<aop:config>
    <aop:pointcut id="pt" expression="execution(* service.*Service.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pt"/>
</aop:config>

```

- aop:advice 与 aop:advisor 区别
 - aop:advice 配置的通知类可以是普通 Java 对象，不实现接口，也不使用继承关系
 - aop:advisor 配置的通知类必须实现通知接口，底层 invoke 调用
 - MethodBeforeAdvice
 - AfterReturningAdvice
 - ThrowsAdvice

pom.xml 文件引入依赖：

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>

```

tx配置

advice

标签: tx:advice, beans 的子标签

作用: 专用于声明事务通知

格式:

```
<beans>
    <tx:advice id="txAdvice" transaction-manager="txManager">
    </tx:advice>
</beans>
```

基本属性:

- id: 用于配置 aop 时指定通知器的 id
- transaction-manager: 指定事务管理器 bean

attributes

类型: tx:attributes, tx:advice 的子标签

作用: 定义通知属性

格式:

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
    </tx:attributes>
</tx:advice>
```

method

标签: tx:method, tx:attribute 的子标签

作用: 设置具体的事务属性

格式:

```
<tx:attributes>
    <!--标准格式-->
    <tx:method name="*" read-only="false"/>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="find*" read-only="true"/>
</tx:attributes>
<aop:pointcut id="pt" expression="execution(* service.*Service.*(..))"/><!--标准-->
```

说明: 通常事务属性会配置多个, 包含 1 个读写的全事务属性, 1 个只读的查询类事务属性

属性:

- name: 待添加事务的方法名表达式 (支持 * 通配符)
- read-only: 设置事务的读写属性, true 为只读, false 为读写
- timeout: 设置事务的超时时长, 单位秒, -1 为无限长
- isolation: 设置事务的隔离级别, 该隔离级设定是基于 Spring 的设定, 非数据库端
- no-rollback-for: 设置事务中不回滚的异常, 多个异常使用 , 分隔
- rollback-for: 设置事务中必回滚的异常, 多个异常使用 , 分隔
- propagation: 设置事务的传播行为

注解

开启注解

XML

标签: tx:annotation-driven

归属: beans 标签

作用: 开启事务注解驱动, 并指定对应的事务管理器

范例:

```
<tx:annotation-driven transaction-manager="txManager"/>
```

纯注解

名称: @EnableTransactionManagement
类型: 类注解, Spring 注解配置类上方
作用: 开启注解驱动, 等同 XML 格式中的注解驱动
范例:

```
@Configuration
@ComponentScan("com.seazean")
@PropertySource("classpath:jdbc.properties")
@Import({JDBCConfig.class, MyBatisConfig.class, TransactionManagerConfig.class})
@EnableTransactionManagement
public class SpringConfig { }
```

```
public class TransactionManagerConfig {
    @Bean
    public PlatformTransactionManager getTransactionManager(@Autowired DataSource dataSource){ //自动装配
        return new DataSourceTransactionManager(dataSource);
    }
}
```

配置注解

名称: @Transactional
类型: 方法注解, 类注解, 接口注解
作用: 设置当前类/接口中所有方法或具体方法开启事务, 并指定相关事务属性
范例:

```
@Transactional(
    readonly = false,
    timeout = -1,
    isolation = Isolation.DEFAULT,
    rollbackFor = {ArithmaticException.class, IOException.class},
    noRollbackFor = {},
    propagation = Propagation.REQUIRES_NEW
)
public void addAccount{}
```

说明:

- @Transactional 注解只有作用到 public 方法上事务才生效
- 不推荐在接口上使用 @Transactional 注解
原因: 在接口上使用注解, 只有在使用基于接口的代理 (JDK) 时才会生效, 因为注解是不能继承的, 这就意味着如果正在使用基于类的代理 (CGLIB) 时, 那么事务的设置将不能被基于类的代理所识别
- 正确的设置 @Transactional 的 rollbackFor 和 propagation 属性, 否则事务可能会回滚失败
- 默认情况下, 事务只有遇到运行期异常和 Error 会导致事务回滚, 但是在遇到检查型 (Checked) 异常时不会回滚
 - 继承自 RuntimeException 或 error 的是非检查型异常, 比如空指针和索引越界, 而继承自 Exception 的则是检查型异常, 比如 IOException, ClassNotFoundException, RuntimeException 本身继承 Exception
 - 非检查型类异常可以不用捕获, 而检查型异常则必须用 try 语句块把异常交给上级方法, 这样事务才能有效

事务不生效的问题

- 情况 1: 确认创建的 MySQL 数据库表引擎是 InnoDB, MyISAM 不支持事务
- 情况 2: 注解到 protected, private 方法上事务不生效, 但不会报错
原因: 理论上而言, 不用 public 修饰, 也可以用 aop 实现事务的功能, 但是方法私有化让其他业务无法调用
AopUtils.canApply: methodMatcher.matches(method, targetClass) --true--> return true
TransactionAttributeSourcePointcut.matches(), AbstractFallbackTransactionAttributeSource 中 getTransactionAttribute 方法调用了其本身的 computeTransactionAttribute 方法, 当加了事务注解的方法不是 public 时, 该方法直接返回 null, 所以造成增强不匹配

```
private TransactionAttribute computeTransactionAttribute(Method method, Class<?> targetClass) {
    // Don't allow no-public methods as required.
    if (allowPublicMethodsonly() && !Modifier.isPublic(method.getModifiers())) {
        return null;
    }
}
```

- 情况 3: 注解所在的类没有被加载成 Bean

- 情况 4：在业务层捕捉异常后未向上抛出，事务不生效

原因：在业务层捕捉并处理了异常（try..catch）等于把异常处理掉了，Spring 就不知道这里有错，也不会主动去回滚数据，推荐做法是在业务层统一抛出异常，然后在控制层统一处理

- 情况 5：遇到检测异常时，也无法回滚

原因：Spring 的默认的事务规则是遇到运行异常（RuntimeException）和程序错误（Error）才会回滚。想针对检测异常进行事务回滚，可以在 @Transactional 注解里使用 rollbackFor 属性明确指定异常

- 情况 6：Spring 的事务传播策略在**内部方法**调用时将不起作用，在一个 Service 内部，事务方法之间的嵌套调用，普通方法和事务方法之间的嵌套调用，都不会开启新的事务，事务注解要加到调用方法上才生效

原因：Spring 的事务都是使用 AOP 代理的模式，动态代理 invoke 后会调用原始对象，而原始对象在去调用方法时是不会触发拦截器，就是一个方法调用本对象的另一个方法，所以事务也就无法生效

```
@Transactional
public int add(){
    update();
}

//注解添加在update方法上无效，需要添加到add()方法上
public int update(){}
```

- 情况 7：注解在接口上，代理对象是 CGLIB

使用注解

- Dao 层

```
public interface AccountDao {
    @Update("update account set money = money + #{money} where name = #{name}")
    void inMoney(@Param("name") String name, @Param("money") Double money);

    @Update("update account set money = money - #{money} where name = #{name}")
    void outMoney(@Param("name") String name, @Param("money") Double money);
}
```

- 业务层

```
public interface AccountService {
    //对当前方法添加事务，该配置将替换接口的配置
    @Transactional(
        readOnly = false,
        timeout = -1,
        isolation = Isolation.DEFAULT,
        rollbackFor = {},//java.lang.ArithmetricException.class, IOException.class
        noRollbackFor = {},
        propagation = Propagation.REQUIRED
    )
    public void transfer(String outName, String inName, Double money);
}
```

```
public class AccountServiceImpl implements AccountService {
    @Autowired
    private AccountDao accountDao;
    public void transfer(String outName, String inName, Double money) {
        accountDao.inMoney(outName,money);
        //int i = 1/0;
        accountDao.outMoney(inName,money);
    }
}
```

- 添加文件 Spring.config、Mybatis.config、JDBCConfig (参考ioc_Mybatis)、TransactionManagerConfig

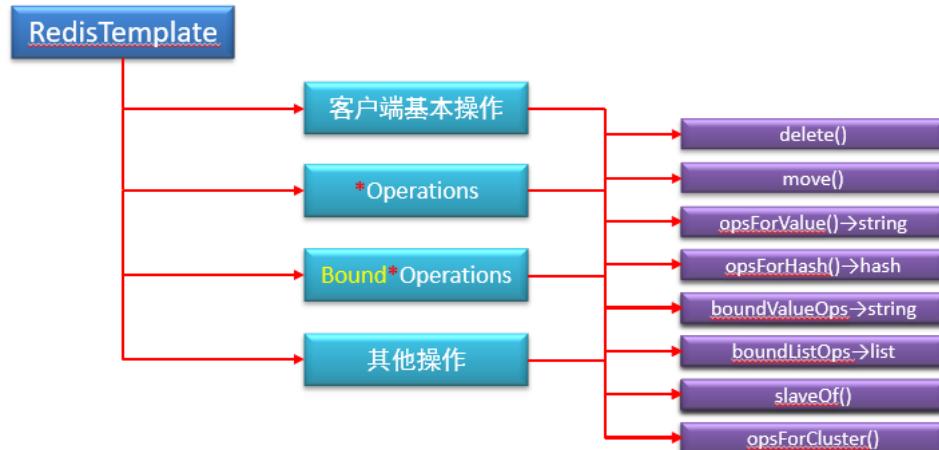
```
@Configuration
@ComponentScan({"", "", ""})
@PropertySource("classpath:jdbc.properties")
@Import({JDBCConfig.class, MyBatisConfig.class})
@EnableTransactionManagement
public class SpringConfig {
```

模板对象

Spring 模板对象: TransactionTemplate、JdbcTemplate、RedisTemplate、RabbitTemplate、JmsTemplate、HibernateTemplate、RestTemplate

- JdbcTemplate: 提供标准的 sql 语句操作API
- NamedParameterJdbcTemplate: 提供标准的具名 sql 语句操作API
- RedisTemplate:

```
public void changeMoney(Integer id, Double money) {  
    redisTemplate.opsForValue().set("account:id:" + id, money);  
}  
public Double findMonyById(Integer id) {  
    Object money = redisTemplate.opsForValue().get("account:id:" + id);  
    return new Double(money.toString());  
}
```



原理

XML

三大对象:

- BeanDefinition: 是 Spring 中极其重要的一个概念, 存储了 bean 对象的所有特征信息, 如是否单例、是否懒加载、factoryBeanName 等, 和 bean 的关系就是类与对象的关系, 一个不同的 bean 对应一个 BeanDefinition
- BeanDefinitionRegistry: 存放 BeanDefinition 的容器, 是一种键值对的形式, 通过特定的 Bean 定义的 id, 映射到相应的 BeanDefinition, BeanFactory 的实现类同样继承 BeanDefinitionRegistry 接口, 拥有保存 BD 的能力
- BeanDefinitionReader: 读取配置文件, XML 用 Dom4j 解析, 注解用 IO 流加载解析

程序:

```
BeanFactory bf = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));  
UserService userService1 = (UserService)bf.getBean("userService");
```

源码解析:

```
public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) {  
    super(parentBeanFactory);  
    this.reader.loadBeanDefinitions(resource);  
}  
public int loadBeanDefinitions(Resource resource) {  
    //将 resource 包装成带编码格式的 EncodedResource  
    //EncodedResource 中 getReader() 方法, 调用java.io包下的 转换流 创建指定编码的输入流对象  
    return loadBeanDefinitions(new EncodedResource(resource));  
}
```

- XmlBeanDefinitionReader.loadBeanDefinitions(): 把 Resource 解析成 BeanDefinition 对象

- currentResources = this.resourcesCurrentlyBeingLoaded.get(): 拿到当前线程已经加载过的所有 EncodedResource 资源, 用 ThreadLocal 保证线程安全
- if (currentResources == null): 判断 currentResources 是否为空, 为空则进行初始化
- if (!currentResources.add(encodedResource)): 如果已经加载过该资源会报错, 防止重复加载
- inputSource = new InputSource(inputStream): 资源对象包装成 InputSource, InputSource 是 SAX 中的资源对象, 用来进行 XML 文件的解析
- return doLoadBeanDefinitions(): 加载返回
- currentResources.remove(encodedResource): 加载完成移除当前 encodedResource

- `resourcesCurrentlyBeingLoaded.remove()` : ThreadLocal 为空时移除元素, 防止内存泄露
- `XmlBeanDefinitionReader.doLoadBeanDefinitions(inputSource, resource)` : 真正的加载函数
 - `Document doc = doLoadDocument(inputSource, resource)` : 转换成有层次结构的 Document 对象
 - `getEntityResolver()` : 获取用来解析 DTD、XSD 约束的解析器
 - `getValidationModeForResource(resource)` : 获取验证模式
 - `int count = registerBeanDefinitions(doc, resource)` : 将 Document 解析成 BD 对象, 注册 (添加) 到 BeanDefinitionRegistry 中, 返回新注册的数量
 - `createBeanDefinitionDocumentReader()` : 创建 DefaultBeanDefinitionDocumentReader 对象
 - `getRegistry().getBeanDefinitionCount()` : 获取解析前 BeanDefinitionRegistry 中的 bd 数量
 - `registerBeanDefinitions(doc, readerContext)` : 注册 BD
 - `this.readerContext = readerContext` : 保存上下文对象
 - `doRegisterBeanDefinitions(doc.getDocumentElement())` : 真正的注册 BD 函数
 - `doc.getDocumentElement()` : 拿出顶层标签
 - `return getRegistry().getBeanDefinitionCount() - countBefore` : 返回新加入的数量
 - `DefaultBeanDefinitionDocumentReader.doRegisterBeanDefinitions()` : 注册 BD 到 BR
 - `createDelegate(getReaderContext(), root, parent)` : beans 是标签的解析器对象
 - `delegate.isDefaultNamespace(root)` : 判断 beans 标签是否是默认的属性
 - `root.getAttribute(PROFILE_ATTRIBUTE)` : 解析 profile 属性
 - `preProcessXml(root)` : 解析前置处理, 自定义实现
 - `parseBeanDefinitions(root, this.delegate)` : 解析 beans 标签中的子标签
 - `parseDefaultElement(ele, delegate)` : 如果是默认的标签, 用该方法解析子标签
 - 判断标签名称, 进行相应的解析
 - `processBeanDefinition(ele, delegate)` :
 - `delegate.parseCustomElement(ele)` : 解析自定义的标签
 - `postProcessXml(root)` : 解析后置处理
 - `DefaultBeanDefinitionDocumentReader.processBeanDefinition()` : 解析 bean 标签并注册到注册中心
 - `delegate.parseBeanDefinitionElement(ele)` : 解析 bean 标签封装为 BeanDefinitionHolder
 - `if (!StringUtil.hasText(beanName) && !aliases.isEmpty())` : 条件一成立说明 name 没有值, 条件二成立说明别名有值
 - beanName = aliases.remove(0) : 拿别名列表的第一个元素当作 beanName
 - `parseBeanDefinitionElement(ele, beanName, containingBean)` : 解析 bean 标签
 - `parseState.push(new BeanEntry(beanName))` : 当前解析器的状态设置为 BeanEntry
 - class 和 parent 属性存在一个, parent 是作为父标签为了被继承
 - `createBeanDefinition(className, parent)` : 设置了 class 的 GenericBeanDefinition 对象
 - `parseBeanDefinitionAttributes()` : 解析 bean 标签的属性
 - 接下来解析子标签
 - `beanName = this.readerContext.generateBeanName(beanDefinition)` : 生成 className + # + 序号的名称赋值给 beanName
 - `return new BeanDefinitionHolder(beanDefinition, beanName, aliases)` : 包装成 BeanDefinitionHolder
 - `registerBeanDefinition(bdHolder, getReaderContext().getRegistry())` : 注册到容器
 - `beanName = definitionHolder.getBeanName()` : 获取 beanName
 - `this.beanDefinitionMap.put(beanName, beanDefinition)` : 添加到注册中心
 - `getReaderContext().fireComponentRegistered()` : 发送注册完成事件

说明: 源码部分的笔记不一定适合所有人阅读, 作者采用流水线式去解析重要的代码, 解析的结构类似于树状, 如果视觉疲劳可以去网上参考一些博客和流程图学习源码。

IOC

容器启动

Spring IOC 容器是 ApplicationContext 或者 BeanFactory, 使用多个 Map 集合保存单实例 Bean, 环境信息等资源, 不同层级有不同的容器, 比如整合 SpringMVC 的父子容器 (先看 Bean 部分的源码解析再回看容器)

ClassPathXmlApplicationContext 与 AnnotationConfigApplicationContext 差不多:

```
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
    this();
    register(annotatedClasses); // 解析配置类, 封装成一个 BeanDefinitionHolder, 并注册到容器
    refresh(); // 加载刷新容器中的 Bean
}
```

```

public AnnotationConfigApplicationContext() {
    // 注册 Spring 的注解解析器到容器
    this.reader = new AnnotatedBeanDefinitionReader(this);
    // 实例化路径扫描器，用于对指定的包目录进行扫描查找 bean 对象
    this.scanner = new ClassPathBeanDefinitionScanner(this);
}

```

AbstractApplicationContext.refresh():

- prepareRefresh(): 刷新前的预处理
 - this.startupDate = System.currentTimeMillis()：设置容器的启动时间
 - initPropertySources()：初始化一些属性设置，可以自定义个性化的属性设置方法
 - getEnvironment().validateRequiredProperties()：检查环境变量
 - earlyApplicationEvents= new LinkedHashSet<ApplicationEvent>()：保存容器中早期的事件
- obtainFreshBeanFactory(): 获取一个全新的 BeanFactory 接口实例，如果容器中存在工厂实例直接销毁
 - refreshBeanFactory()：创建 BeanFactory，设置序列化 ID、读取 BeanDefinition 并加载到工厂
 - if (hasBeanFactory()): applicationContext 内部拥有一个 beanFactory 实例，需要将该实例完全释放销毁
 - destroyBeans()：销毁原 beanFactory 实例，将 beanFactory 内部维护的单实例 bean 全部清掉，如果哪个 bean 实现了 Disposable接口，还会进行 bean destroy 方法的调用处理
 - this.singletonsCurrentlyInDestruction = true：设置当前 beanFactory 状态为销毁状态
 - String[] disposableBeanNames：获取销毁集合中的 bean，如果当前 bean 有析构函数就会在销毁集合
 - destroyingSingleton(disposableBeanNames[i])：遍历所有的 disposableBeans，执行销毁方法
 - removesingleton(beanName)：清除三级缓存和 registeredSingletons 中的当前 beanName 的数据
 - this.disposableBeans.remove(beanName)：从销毁集合中清除，每个 bean 只能 destroy 一次
 - destroyBean(beanName, disposableBean)：销毁 bean
 - dependentBeanMap 记录了依赖当前 bean 的其他 bean 信息，因为依赖的对象要被回收了，所以依赖当前 bean 的其他对象都要执行 destroySingleton，遍历 dependentBeanMap 执行销毁
 - bean.destroy()：解决完成依赖后，执行 DisposableBean 的 destroy 方法
 - this.dependenciesForBeanMap.remove(beanName)：保存当前 bean 依赖了谁，直接清除
 - 进行一些集合和缓存的清理工作
 - closeBeanFactory()：将容器内部的 beanFactory 设置为空，重新创建
 - beanFactory = createBeanFactory()：创建新的 DefaultListableBeanFactory 对象
 - beanFactory.setSerializationId(getId())：进行 ID 的设置，可以根据 ID 获取 BeanFactory 对象
 - customizeBeanFactory(beanFactory)：设置是否允许覆盖和循环引用
 - loadBeanDefinitions(beanFactory)：加载 BeanDefinition 信息，注册 BD注册到 BeanFactory 中
 - this.beanFactory = beanFactory：把 beanFactory 填充至容器中
 - getBeanFactory(): 返回创建的 DefaultListableBeanFactory 对象，该对象继承 BeanDefinitionRegistry
 - prepareBeanFactory(beanFactory): BeanFactory 的预准备工作，向容器中添加一些组件
 - setBeanClassLoader(getClassloader())：给当前 bf 设置一个类加载器，加载 bd 的 class 信息
 - setBeanExpressionResolver()：设置 EL 表达式解析器
 - addPropertyEditorRegistrar：添加一个属性编辑器，解决属性注入时的格式转换
 - addBeanPostProcessor()：添加后处理器，主要用于向 bean 内部注入一些框架级别的实例
 - ignoreDependencyInterface()：设置忽略自动装配的接口，bean 内部的这些类型的字段 不参与依赖注入
 - registerResolvableDependency()：注册一些类型依赖关系
 - addBeanPostProcessor()：将配置的监听者注册到容器中，当前 bean 实现 ApplicationListener 接口就是监听器事件
 - beanFactory.registerSingleton()：添加一些系统信息
 - postProcessBeanFactory(beanFactory): BeanFactory 准备工作完成后进行的后置处理工作，扩展方法
 - invokeBeanFactoryPostProcessors(beanFactory): 执行 BeanFactoryPostProcessor 的方法
 - processedBeans = new HashSet<>()：存储已经执行过的 BeanFactoryPostProcessor 的 beanName
 - if (beanFactory instanceof BeanDefinitionRegistry)：当前 BeanFactory 是 bd 的注册中心，bd 全部注册到 bf
 - for (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors)：遍历所有的 bf 后置处理器
 - if (postProcessor instanceof BeanDefinitionRegistryPostProcessor)：是 Registry 类的后置处理器
 - registryProcessor.postProcessBeanDefinitionRegistry(registry)：向 bf 中注册一些 bd
 - registryProcessors.add(registryProcessor)：添加到 BeanDefinitionRegistryPostProcessor 集合
 - regularPostProcessors.add(postProcessor)：添加到 BeanFactoryPostProcessor 集合
 - 逻辑到这里已经获取到所有 BeanDefinitionRegistryPostProcessor 和 BeanFactoryPostProcessor 接口类型的后置处理器
 - 首先回调 BeanDefinitionRegistryPostProcessor 类的后置处理方法 postProcessBeanDefinitionRegistry()
 - 获得了 PriorityOrdered (主排序接口) 接口的 bdrpp，进行 sort 排序，然后全部执行并放入已经处理过的集合
 - 再执行实现了 Ordered (次排序接口) 接口的 bdrpp
 - 最后执行没有实现任何优先级或者是顺序接口 bdrpp，boolean reiterate = true 控制 while 是否需要再次循环，循环内是查找并执行 bdrpp 后处理器的 registry 相关的接口方法，接口方法执行以后会向 bf 内注册 bd，注册的 bd 也有可能是 bdrpp 类型，所以需要该变量控制循环
 - processedBeans.add(ppName)：已经执行过的后置处理器存储到该集合中，防止重复执行
 - invokeBeanFactoryPostProcessors()：bdrpp 继承了 BeanFactoryPostProcessor，有 postProcessBeanFactory 方法
 - 执行普通 BeanFactoryPostProcessor 的相关 postProcessBeanFactory 方法，按照主次无次序执行

- if (processedBeans.contains(ppName)) : 会过滤掉已经执行过的后置处理器
- beanFactory.clearMetadataCache() : 清除缓存中合并的 Bean 定义, 因为后置处理器可能更改了元数据

以上是 BeanFactory 的创建及预准备工作, 接下来进入 Bean 的流程

- registerBeanPostProcessors(beanFactory): **注册 Bean 的后置处理器**, 为了干预 Spring 初始化 bean 的流程, 这里仅仅是向容器中注入而非使用
 - beanFactory.getBeanNamesForType(BeanPostProcessor.class) : 获取配置中实现了 BeanPostProcessor 接口类型
 - int beanProcessorTargetCount : 后置处理器的数量, 已经注册的 + 未注册的 + 即将要添加的一个
 - beanFactory.addBeanPostProcessor(new BeanPostProcessorChecker()): 添加一个检查器
 - BeanPostProcessorChecker.postProcessAfterInitialization(): 初始化后的后处理器方法
 - !(bean instanceof BeanPostProcessor) : 当前 bean 类型是普通 bean, 不是后置处理器
 - !isInfrastructureBean(beanName) : 成立说明当前 beanName 是用户级别的 bean 不是 Spring 框架的
 - this.beanFactory.getBeanPostProcessorCount() < this.beanPostProcessorTargetCount : BeanFactory 上面注册后处理器数量 < 后处理器数量, 说明后处理框架尚未初始化完成
 - for (String ppName : postProcessorNames) : 遍历 PostProcessor 集合, 根据实现不同的顺序接口添加到不同集合
 - sortPostProcessors(priorityOrderedPostProcessors, beanFactory) : 实现 PriorityOrdered 接口的后处理器排序
 - registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors) : **注册到 beanFactory 中**
- 接着排序实现 Ordered 接口的后置处理器, 然后注册普通的 (没有实现任何优先级接口) 后置处理器
- 最后排序 MergedBeanDefinitionPostProcessor 类型的处理器, 根据实现的排序接口, 排序完注册到 beanFactory 中
- beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext)) : 重新注册 ApplicationListenerDetector 后处理器, 用于在 Bean 创建完成后检查是否属于 ApplicationListener 类型, 如果是就把 Bean 放到监听器容器中保存起来
- initMessageSource(): 初始化 MessageSource 组件, 主要用于做国际化功能, 消息绑定与消息解析
 - if (beanFactory.containsLocalBean(MESSAGE_SOURCE_BEAN_NAME)) : 容器是否含有名称为 messageSource 的 bean
 - beanFactory.getBean(MESSAGE_SOURCE_BEAN_NAME, MessageSource.class) : 如果有证明用户自定义了该类型的 bean, 获取后直接赋值给 this.messageSource
 - dms = new DelegatingMessageSource() : 容器中没有就新建一个赋值
- initApplicationEventMulticaster(): **初始化事件传播器**, 在注册监听器时会用到
 - if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) : 条件成立说明用户自定义了事件传播器, 可以实现 ApplicationEventMulticaster 接口编写自己的事件传播器, 通过 bean 的方式提供给 Spring
 - 如果有就直接从容器中获取; 如果没有则创建一个 SimpleApplicationEventMulticaster 注册
- onRefresh(): 留给用户去实现, 可以硬编码提供一些组件, 比如提供一些监听器
- registerListeners(): 注册通过配置提供的 Listener, 这些**监听器**最终注册到 ApplicationEventMulticaster 内
 - for (ApplicationListener<?> listener : getApplicationListeners()) : 注册编码实现的监听器
 - getBeanNamesForType(ApplicationListener.class, true, false) : 注册通过配置提供的 Listener
 - multicastEvent(earlyEvent) : **发布前面步骤产生的事件 applicationEvents**
 - Executor executor = getTaskExecutor() : 获得线程池, 有线程池就异步执行, 没有就同步执行
- finishBeanFactoryInitialization(): **实例化非懒加载状态的单实例**
 - beanFactory.freezeConfiguration() : **冻结配置信息**, 就是冻结 BD 信息, 冻结后无法再向 bf 内注册 bd
 - beanFactory.preInstantiateSingletons() : 实例化 non-lazy-init singletons
 - for (String beanName : beanNames) : 遍历容器内所有的 beanDefinitionNames
 - getMergedLocalBeanDefinition(beanName) : 获得与父类合并后的对象 (Bean → 获取流程部分详解此函数)
 - if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) : BD 对应的 Class 满足非抽象、单实例, 非懒加载, 需要预先实例化
 - if (isFactoryBean(beanName)) : BD 对应的 Class 是 factoryBean 对象
 - getBean(FACTORY_BEAN_PREFIX + beanName) : 获得工厂 FactoryBean 实例本身
 - isEagerInit : 控制 FactoryBean 内部管理的 Bean 是否也初始化
 - getBean(beanName) : **初始化 Bean, 获得 Bean 详解此函数**
 - getBean(beanName) : 不是工厂 bean 直接获取
 - for (String beanName : beanNames) : 检查所有的 Bean 是否实现 SmartInitializingSingleton 接口, 实现了就执行 afterSingletonsInstantiated(), 进行一些创建后的操作
- finishRefresh(): 完成刷新后做的一些事情, 主要是启动生命周期
 - clearResourceCaches() : 清空上下文缓存
 - initLifecycleProcessor() : **初始化和生命周期有关的后置处理器**, 容器的生命周期
 - if (beanFactory.containsLocalBean(LIFECYCLE_PROCESSOR_BEAN_NAME)) : 成立说明自定义了生命周期处理器
 - defaultProcessor = new DefaultLifecycleProcessor() : Spring 默认提供的生命周期处理器
 - beanFactory.registerSingleton() : 将生命周期处理器注册到 bf 的一级缓存和注册单例集合中
 - getLifecycleProcessor().onRefresh() : 获得该**生命周期后置处理器回调 onRefresh()**, 调用 startBeans(true)
 - lifecycleBeans = getLifecycleBeans() : 获得到所有实现了 Lifecycle 接口的对象包装到 Map 内, key 是 beanName, value 是 Lifecycle 对象
 - int phase = getPhase(bean) : 获得当前 Lifecycle 的 phase 值, 当前生命周期对象可能依赖其他生命周期对象的执行结果, 所以需要 phase 决定执行顺序, 数值越低的优先执行
 - LifecycleGroup group = phases.get(phase) : 把 phsae 相同的 Lifecycle 存入 LifecycleGroup
 - if (group == null) : group 为空则创建, 初始情况下是空的
 - group.add(beanName, bean) : 将当前 Lifecycle 添加到当前 phase 值一样的 group 内

- `collections.sort(keys)`：从小到大排序，按优先级启动
- `phases.get(key).start()`：遍历所有的 Lifecycle 对象开始启动
- `doStart(this.lifecycleBeans, member.name, this.autoStartupOnly)`：底层调用该方法启动
 - `bean = lifecycleBeans.remove(beanName)`：确保 Lifecycle 只被启动一次，在一个分组内被启动了在其他分组内就看不到 Lifecycle 了
 - `dependenciesForBean = getBeanFactory().getDependenciesForBean(beanName)`：获取当前即将被启动的 Lifecycle 所依赖的其他 beanName，需要先启动所依赖的 bean，才能启动自身
 - `if (...)`：传入的参数 autoStartupOnly 为 true 表示启动 isAutoStartUp 为 true 的 SmartLifecycle 对象，不会启动普通的生命周期的对象；false 代表全部启动
 - `bean.start()`：调用启动方法
- `publishEvent(new ContextRefreshedEvent(this))`：发布容器刷新完成事件
- `liveBeansView.registerApplicationContext(this)`：暴露 Mbean

补充生命周期 stop() 方法的调用

- `DefaultLifecycleProcessor.stop()`：调用 `DefaultLifecycleProcessor.stopBeans()`
 - 获取到所有实现了 Lifecycle 接口的对象并按 phase 数值分组
 - `keys.sort(Collections.reverseOrder())`：按 phase 降序排序 Lifecycle 接口，最先启动的最晚关闭（责任链？）
 - `phases.get(key).stop()`：遍历所有的 Lifecycle 对象开始停止
 - `latch = new CountDownLatch(this.smartMemberCount)`：创建 CountDownLatch，设置 latch 内部的值为当前分组内的 smartMemberCount 的数量
 - `countDownBeanNames = Collections.synchronizedSet(new HashSet<>())`：保存当前正在处理关闭的 smartLifecycle 的 BeanName
 - `for (LifecycleGroupMember member : this.members)`：处理本分组内需要关闭的 Lifecycle
 - `doStop(this.lifecycleBeans, member.name, latch, countDownBeanNames)`：真正的停止方法
 - `getBeanFactory().getDependentBeans(beanName)`：获取依赖当前 Lifecycle 的其他对象的 beanName，因为当前的 Lifecycle 即将要关闭了，所有的依赖了当前 Lifecycle 的 bean 也要关闭
 - `countDownBeanNames.add(beanName)`：将当前 SmartLifecycle beanName 添加到 countDownBeanNames 集合内，该集合表示正在关闭的 SmartLifecycle
 - `bean.stop()`：调用停止的方法

获取 Bean

单实例：在容器启动时创建对象

多实例：在每次获取的时候创建对象

获取流程：获取 Bean 时先从单例池获取，如果没有则进行第二次获取，并带上工厂类去创建并添加至单例池

Java 启动 Spring 代码：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
UserService userService = (UserService) context.getBean("userService");
```

`AbstractBeanFactory doGetBean()`：获取 Bean，`context.getBean()` 追踪到此

- `beanName = transformedBeanName(name)`：name 可能是一个别名，重定向出来真实 beanName；也可能是一个 & 开头的 name，说明要获取的 bean 实例对象，是一个 FactoryBean 对象（IOC 原理 → 核心类）
 - `BeanFactoryUtils.transformedBeanName(name)`：判断是哪种 name，返回截取 & 以后的 name 并放入缓存
 - `transformedBeanNameCache.computeIfAbsent`：缓存是并发安全集合，key == null || value == null 时 put 成功
 - do while 循环一直去除 & 直到不再含有 &
 - `canonicalName(name)`：aliasMap 保存别名信息，其中的 do while 逻辑是迭代查找，比如 A 别名叫做 B，但是 B 又有别名叫 C，aliasMap 为 {"C": "B", "B": "A"}，get(C) 最后返回的是 A
- `DefaultSingletonBeanRegistry.getSingleton()`：第一次获取从缓存池获取（循环依赖详解此代码）
 - 缓存中有数据进行 `getObjectForBeanInstance()` 获取可使用的 Bean（本节结束部分详解此函数）
 - 缓存中没有数据进行下面的逻辑进行创建
- `if(isPrototypeCurrentlyInCreation(beanName))`：检查 bean 是否在原型（Prototype）正在被创建的集合中，如果是就报错，说明产生了循环依赖，原型模式解决不了循环依赖

原因：先加载 A，把 A 加入集合，A 依赖 B 去加载 B，B 又依赖 A，去加载 A，发现 A 在正在创建集合中，产生循环依赖

- `markBeanAsCreated(beanName)`：把 bean 标记为已经创建，防止其他线程重新创建 Bean

- `mbd = getMergedLocalBeanDefinition(beanName)`：获取合并父 BD 后的 BD 对象，BD 是直接继承的，合并后的 BD 信息是包含父类的 BD 信息
 - `this.mergedBeanDefinitions.get(beanName)`：从缓存中获取
 - `if(bd.getParentName() == null)`：beanName 对应 BD 没有父 BD 就不用处理继承，封装为 RootBeanDefinition 返回
 - `parentBeanName = transformedBeanName(bd.getParentName())`：处理父 BD 的 name 信息
 - `if(!beanName.equals(parentBeanName))`：一般情况父子 BD 的名称不同
 - `pbd = getMergedBeanDefinition(parentBeanName)`：递归调用，最终返回父 BD 的父 BD 信息
 - `mbd = new RootBeanDefinition(pbd)`：按照父 BD 信息创建 RootBeanDefinition 对象
 - `mbd.overrideFrom(bd)`：子 BD 信息覆盖 mbd，因为是要以子 BD 为基准，不存在的才去父 BD 寻找（类似 Java 继承）
 - `this.mergedBeanDefinitions.put(beanName, mbd)`：放入缓存

◦ `checkMergedBeanDefinition()`：判断当前 BD 是否为抽象 BD，抽象 BD 不能创建实例，只能作为父 BD 被继承

◦ `mbd.getDependson()`：获取 bean 标签 depends-on

◦ `if(dependson != null)`：遍历所有的依赖加载，解决不了循环依赖

`isDependent(beanName, dep)`：判断循环依赖，出现循环依赖问题报错

▪ 两个 Map：`<bean name="A" depends-on="B" ...>`

▪ `dependentBeanMap`：记录依赖了当前 beanName 的其他 beanName（谁依赖我，我记录谁）

▪ `dependenciesForBeanMap`：记录当前 beanName 依赖的其它 beanName

▪ 以 B 为视角 `dependentBeanMap {"B": {"A"}}`，以 A 为视角 `dependenciesForBeanMap {"A": {"B"}}`

▪ `canonicalName(beanName)`：处理 bean 的 name

▪ `dependentBeans = this.dependentBeanMap.get(canonicalName)`：获取依赖了当前 bean 的 name

▪ `if (dependentBeans.contains(dependentBeanName))`：依赖了当前 bean 的集合中是否有该 name，有就产生循环依赖

▪ 进行递归处理所有的引用：假如 `<bean name="A" dp="B"> <bean name="B" dp="C"> <bean name="C" dp="A">`

```
dependentBeanMap={A:{C}, B:{A}, C:{B}}
// C 依赖 A          判断谁依赖了 C      递归判断          谁依赖了 B
isDependent(C, A) → C#dependentBeans={B} → isDependent(B, A); → B#dependentBeans={A} //返回true
```

`registerDependentBean(dep, beanName)`：把 bean 和依赖注册到两个 Map 中，注意参数的位置，被依赖的在前

`getBean(dep)`：先加载依赖的 Bean，又进入 `doGetBean()` 的逻辑

◦ `if (mbd.isSingleton())`：判断 bean 是否是单例的 bean

`getSingleton(String, ObjectFactory<?>)`：第二次获取，传入一个工厂对象，这个方法更倾向于创建实例并返回

```
sharedInstance = getSingleton(beanName, () -> {
    return createBean(beanName, mbd, args); //创建, 跳转生命周期
    //lambda表达式, 调用了ObjectFactory的getObject()方法, 实际回调接口实现的是createBean()方法进行创建对象
});
```

▪ `singletonObjects.get(beanName)`：从一级缓存检查是否已经被加载，单例模式复用已经创建的 bean

▪ `this.singletonsCurrentlyInDestruction`：容器销毁时会设置这个属性为 true，这时就不能再创建 bean 实例了

▪ `beforeSingletonCreation(beanName)`：检查构造注入的依赖，构造参数注入产生的循环依赖无法解决

`!this.singletonsCurrentlyInCreation.add(beanName)`：将当前 beanName 放入到正在创建中单实例集合，放入成功说明没有产生循环依赖，失败则产生循环依赖，进入判断条件内的逻辑抛出异常

原因：加载 A，向正在创建集合中添加了 {A}，根据 A 的构造方法实例化 A 对象，发现 A 的构造方法依赖 B，然后加载 B，B 构造方法的参数依赖于 A，又去加载 A 时来到当前方法，因为创建中集合已经存在 A，所以添加失败

▪ `singletonObject = singletonFactory.getObject()`：创建 bean (生命周期部分详解)

▪ 创建完成以后，Bean 已经初始化好，是一个完整的可使用的 Bean

▪ `afterSingletonCreation(beanName)`：从正在创建中的集合中移出

▪ `addSingleton(beanName, singletonObject)`：添加一级缓存单例池中，从二级三级缓存移除

`bean = getObjectForBeanInstance`：单实例可能是普通单实例或者 FactoryBean，如果是 FactoryBean 实例，需要判断 name 是带 & 还是不带 &，带 & 说明 `getBean` 获取 FactoryBean 对象，否则是获取 FactoryBean 内部管理的实例

▪ 参数 name 是未处理 & 的 name，beanName 是处理过 & 和别名后的 name

▪ `if(BeanFactoryUtils.isFactoryDereference(name))`：判断 `doGetBean` 中参数 name 前是否带 &，不是处理后的

▪ `if(!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name))`：Bean 是普通单实例或者是 FactoryBean 就可以直接返回，否则进入下面的获取 FactoryBean 内部管理的实例的逻辑

▪ `getCachedObjectForFactoryBean(beanName)`：尝试到缓存获取，获取到直接返回，获取不到进行下面逻辑

▪ `if (mbd == null && containsBeanDefinition(beanName))`：Spring 中有当前 beanName 的 BeanDefinition 信息

`mbd = getMergedLocalBeanDefinition(beanName)`：获取合并后的 BeanDefinition

▪ `mbd.isSynthetic()`：默认值是 false 表示这是一个用户对象，如果是 true 表示是系统对象

▪ `object = getobjectFromFactoryBean(factory, beanName, !synthetic)`：从工厂内获取实例

▪ `factory.isSingleton() && containsSingleton(beanName)`：工厂内部维护的对象是单实例并且一级缓存存在该 bean

▪ 首先去缓存中获取，获取不到就使用工厂获取然后放入缓存，进行循环依赖判断

◦ `else if (mbd.isPrototype())`：bean 是原型的 bean

`beforePrototypeCreation(beanName)`：当前线程正在创建的原型对象 beanName 存入 `prototypesCurrentlyInCreation`

▪ `curVal = this.prototypesCurrentlyInCreation.get()`：获取当前线程的正在创建的原型类集合

▪ `this.prototypesCurrentlyInCreation.set(beanName)`：集合为空就把当前 beanName 加入

▪ `if (curVal instanceof String)`：已经有线程相关原型类创建了，把当前的创建的加进去

`createBean(beanName, mbd, args)`：创建原型类对象，不需要三级缓存

`afterPrototypeCreation(beanName)`：从正在创建中的集合中移除该 beanName，与 `beforePrototypeCreation` 逻辑相反

◦ `convertIfNecessary()`：依赖检查，检查所需的类型是否与实际 bean 实例的类型匹配

◦ `return (T) bean`：返回创建完成的 bean

生命周期

四个阶段

Bean 的生命周期：实例化 instantiation，填充属性 populate，初始化 initialization，销毁 destruction

AbstractAutowireCapableBeanFactory.createBean(): 进入 Bean 生命周期的流程

- resolvedClass = resolveBeanClass(mbd, beanName)：判断 mbd 中的 class 是否已经加载到 JVM，如果未加载则使用类加载器将 beanName 加载到 JVM 中并返回 class 对象
- if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null)：条件成立封装 mbd 并把 resolveBeanClass 设置到 bd 中
 - 条件二：mbd 在 resolveBeanClass 之前是否有 class
 - 条件三：mbd 有 className
- bean = resolveBeforeInstantiation(beanName, mbdToUse)：实例化前的后置处理器返回一个代理实例对象（不是 AOP）
 - 自定义类继承 InstantiationAwareBeanPostProcessor，重写 postProcessBeforeInstantiation 方法，方法逻辑为创建对象
 - 并配置文件 <bean class="intefacePackage.MyInstantiationAwareBeanPostProcessor"> 导入为 bean
 - 条件成立，短路操作，直接 return bean
- Object beanInstance = doCreateBean(beanName, mbdToUse, args)：Do it

AbstractAutowireCapableBeanFactory.doCreateBean(beanName, RootBeanDefinition, Object[] args): 创建 Bean

- BeanWrapper instanceWrapper = null：Spring 给所有创建的 Bean 实例包装成 BeanWrapper，内部最核心的方法是获取实例，提供了一些额外的接口方法，比如属性访问器
- instanceWrapper = this.factoryBeanInstanceCache.remove(beanName)：单例对象尝试从缓存中获取，会移除缓存
- createBeanInstance()：缓存中没有实例就进行创建实例（逻辑复杂，下一小节详解）
- if (!mbd.postProcessed)：每个 bean 只进行一次该逻辑
 - applyMergedBeanDefinitionPostProcessors()：后置处理器，合并 bd 信息，接下来要属性填充了

AutowiredAnnotationBeanPostProcessor.postProcessMergedBeanDefinition()：后置处理逻辑 (@Autowired)

- metadata = findAutowiringMetadata(beanName, beanType, null)：提取当前 bean 整个继承体系内的 @Autowired、@Value、@Inject 信息，存入一个 InjectionMetadata 对象，保存着当前 bean 信息和要自动注入的字段信息

```
private final Class<?> targetClass; //当前 bean
private final Collection<InjectedElement> injectedElements; //要注入的信息集合
```

- metadata = buildAutowiringMetadata(clazz)：查询当前 clazz 感兴趣的注解信息
 - ReflectionUtils.dowithLocalFields()：提取字段的注解信息
 - findAutowiredAnnotation(field)：代表感兴趣的注解就是那三种注解，获取这三种注解的元数据
 - ReflectionUtils.dowithLocalMethods()：提取方法的注解信息
 - do{} while (targetClass != null && targetClass != Object.class)：循环从父类中解析，直到 Object 类
 - this.injectionMetadataCache.put(cacheKey, metadata)：存入缓存

mbd.postProcessed = true：设置为 true，下次访问该逻辑不会再进入

- earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences && isSingletonCurrentlyInCreation(beanName)：单例、解决循环引用、是否在单例正在创建集中

```
if (earlySingletonExposure) {
    // 【放入三级缓存一个工厂对象，用来获取提前引用】
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
    // Lambda 表达式，用来获取提前引用，循环依赖部分详解该逻辑
}
```

- populateBean(beanName, mbd, instanceWrapper)：属性填充，依赖注入，整体逻辑是先处理标签再处理注解，填充至 pvs 中，最后通过 apply 方法最后完成属性依赖注入到 BeanWrapper
 - if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName))：实例化的后置处理器，默认返回 true，可以自定义类继承 InstantiationAwareBeanPostProcessor 修改后置处理方法的返回值为 false，使 continueWithPropertyPopulation 为 false，会导致直接返回，不进行属性的注入
 - if (!continueWithPropertyPopulation)：自定义方法返回值会造成该条件成立，逻辑为直接返回，不进行依赖注入
 - PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null)：处理依赖注入逻辑开始
 - mbd.getResolvedAutowireMode() == ?：根据 bean 标签配置的 autowire 判断是 BY_NAME 或者 BY_TYPE
 - autowireByName(beanName, mbd, bw, newPvs)：根据字段名称去获取依赖的 bean，还没注入，只是添加到 pvs
 - propertyNames = unsatisfiedNonSimpleProperties(mbd, bw)：bean 实例中有该字段和该字段的 setter 方法，但是在 bd 中没有 property 属性
 - 拿到配置的 property 信息和 bean 的所有字段信息
 - pd.getWriteMethod() != null：当前字段是否有 set 方法，配置类注入的方式需要 set 方法
 - !isExcludedFromDependencyCheck(pd)：当前字段类型是否在忽略自动注入的列表中
 - !pvs.contains(pd.getName())：当前字段不在 xml 或者其他方式的配置中，也就是 bd 中不存在对应的 property
 - !BeanUtils.isSimpleProperty(pd.getPropertyType())：是否是基本数据类型和内置的几种数据类型，基本数据类型不允许自动注入
 - if (containsBean(propertyName))：BeanFactory 中存在当前 property 的 bean 实例，说明找到对应的依赖数据

- `getBean(propertyName)`：拿到 `propertyName` 对应的 bean 实例
- `pvs.add(propertyName, bean)`：填充到 pvs 中
- `registerDependentBean(propertyName, beanName)`：添加到两个依赖 Map (`dependsOn`) 中
- `autowireByType(beanName, mbd, bw, newPvs)`：根据字段类型去查找依赖的 bean
 - `desc = new AutowireByTypeDescriptor(methodParam, eager)`：依赖描述信息
 - `resolveDependency(desc, beanName, autowiredBeanNames, converter)`：根据描述信息，查找依赖对象，容器中没有对应的实例但是有对应的 BD，会调用 `getBean(Type)` 获取对象
- `pvs = newPvs`：`newPvs` 是处理了依赖数据后的 pvs，所以赋值给 pvs
- `hasInstAwareBpps`：表示当前是否有 `InstantiationAwareBeanPostProcessors` 的后置处理器 (Autowired)
- `pvsToUse = ibp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName)`：`@Autowired` 注解的注入，这个传入的 pvs 对象，最后原封不动的返回，不会添加东西
 - `findAutowiringMetadata()`：包装着当前 bd 需要注入的注解信息集合，**三种注解的元数据**，直接缓存获取
 - `InjectionMetadata.InjectedElement.inject()`：遍历注解信息解析后注入到 Bean，方法和字段的注入实现不同
- 以字段注入为例：
 - `value = resolveFieldValue(field, bean, beanName)`：处理字段属性值
 - `value = beanFactory.resolveDependency()`：解决依赖
 - `result = doResolveDependency()`：**真正处理自动注入依赖的逻辑**
 - `Object shortcut = descriptor.resolveShortcut(this)`：默认返回 null
 - `Object value = getAutowireCandidateResolver().getSuggestedValue(descriptor)`：**获取 @Value 的值**
 - `converter.convertIfNecessary(value, type, descriptor.getTypeDescriptor())`：如果 value 不是 null，就直接进行类型转换返回数据
 - `matchingBeans = findAutowireCandidates(beanName, type, descriptor)`：如果 value 是空说明字段是引用类型，**获取 @Autowired 的 Bean**

```
// addCandidateEntry() -> Object beanInstance = descriptor.resolveCandidate()
public Object resolveCandidate(String beanName, Class<?> requiredType, BeanFactory beanFactory) throws BeansException {
    // 获取 bean
    return beanFactory.getBean(beanName);
}
```
- `ReflectionUtils.makeAccessible(field)`：修改访问权限
- `field.set(bean, value)`：获取属性访问器为此 field 对象赋值
- `applyPropertyValues()`：将所有解析的 `PropertyValues` 的注入至 `BeanWrapper` 实例中 (深拷贝)
 - `if (pvs.isEmpty())`：注解 `@Autowired` 和 `@Value` 标注的信息在后置处理的逻辑注入完成，此处为空直接返回
 - 下面的逻辑进行 XML 配置的属性的注入，首先获取转换器进行数据转换，然后获取 `WriteMethod (set)` 方法进行反射调用，完成属性的注入
- `initializeBean(String, Object, RootBeanDefinition)`：初始化，分为配置文件和实现接口两种方式
 - `invokeAwareMethods(beanName, bean)`：根据 bean 是否实现 Aware 接口执行初始化的方法
 - `wrappedBean = applyBeanPostProcessorsBeforeInitialization`：初始化前的后置处理器，可以继承接口重写方法
 - `processor.postProcessBeforeInitialization()`：执行后置处理的方法，默认返回 bean 本身
 - `if (current == null) return result`：重写方法返回 null，会造成后置处理的短路，直接返回
 - `invokeInitMethods(beanName, wrappedBean, mbd)`：反射执行初始化方法
 - `isInitializingBean = (bean instanceof InitializingBean)`：初始化方法的定义有两种方式，一种是自定义类实现 `InitializingBean` 接口，另一种是配置文件配置 `<bean id="..." class="..." init-method="init"/>`
 - `isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))`：
 - 条件一：当前 bean 是不是实现了 `InitializingBean`
 - 条件二：`InitializingBean` 接口中的方法 `afterPropertiesSet`，判断该方法是否是容器外管理的方法
 - `if (mbd != null && bean.getClass() != NullBean.class)`：成立说明是配置文件的方式
 - `if(!(接口条件)) 表示如果通过接口实现了初始化方法的话，就不会在调用配置类中 init-method 定义的方法`
 - `((InitializingBean) bean).afterPropertiesSet()`：调用方法
 - `invokeCustomInitMethod`：执行自定义的方法
 - `initMethodName = mbd.getInitMethodName()`：获取方法名
 - `Method initMethod = ()`：根据方法名获取到 init-method 方法
 - `methodToInvoke = ClassUtils.getInterfaceMethodIfPossible(initMethod)`：将方法转成从接口层面获取
 - `ReflectionUtils.makeAccessible(methodToInvoke)`：访问权限设置成可访问
 - `methodToInvoke.invoke(bean)`：反射调用初始化方法，以当前 bean 为角度去调用
 - `wrappedBean = applyBeanPostProcessorsAfterInitialization`：初始化后的后置处理器
 - `AbstractAutoProxyCreator.postProcessAfterInitialization()`：如果 Bean 被子类标识为要代理的 bean，则使用配置的拦截器**创建代理对象**，AOP 部分詳解
 - 如果不存在循环依赖，创建动态代理 bean 在此处完成；否则真正的创建阶段是在属性填充时获取提前引用的阶段，**循环依赖詳解**，源码分析：

```
// 该集合用来避免重复将某个 bean 生成代理对象,
```

```

private final Map<Object, Object> earlyProxyReferences = new ConcurrentHashMap<>(16);

public Object postProcessAfterInitialization(@Nullable Object bean, String bn) {
    if (bean != null) {
        // cacheKey 是 beanName 或者加上 &
        Object cacheKey = getCacheKey(bean.getClass(), beanName);y
        if (this.earlyProxyReferences.remove(cacheKey) != bean) {
            // 去提前代理引用池中寻找该key, 不存在则创建代理
            // 如果存在则证明被代理过, 则判断是否是当前的 bean, 不是则创建代理
            return wrapIfNecessary(bean, bn, cacheKey);
        }
    }
    return bean;
}

```

- o `if (earlySingletonExposure)`: 是否允许提前引用

`earlySingletonReference = getSingleton(beanName, false)`: 从二级缓存获取实例，放入一级缓存是在 `doGetBean` 中的 `sharedInstance = getSingleton()` 逻辑中，此时在 `createBean` 的逻辑还没有返回，所以一级缓存没有

`if (earlySingletonReference != null)`: 当前 bean 实例从二级缓存中获取到了，说明产生了循环依赖，在属性填充阶段会提前调用三级缓存中的工厂生成 Bean 的代理对象（或原始实例），放入二级缓存中，然后使用原始 bean 继续执行初始化

- `if (exposedObject == bean)`: 初始化后的 bean == 创建的原始实例，条件成立的两种情况：当前的真实实例不需要被代理；当前实例存在循环依赖已经被提前代理过了，初始化时的后置处理器直接返回 bean 原实例

`exposedObject = earlySingletonReference`: 把代理后的 Bean 传给 exposedObject 用来返回，因为只有代理对象才封装了拦截器链，main 方法中用代理对象调用方法时会进行增强，代理是对原始对象的包装，所以这里返回的代理对象中含有完整的原实例（属性填充和初始化后的），是一个完整的代理对象，返回后外层方法会将当前 Bean 放入一级缓存

- `else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName))`: 是否有其他 bean 依赖当前 bean，执行到这里说明是不存在循环依赖、存在增强代理的逻辑，也就是正常的逻辑

- `dependentBeans = getDependentBeans(beanName)`: 取到依赖当前 bean 的其他 beanName
- `if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean))`: 判断 dependentBean 是否创建完成
 - `if (!this.alreadyCreated.contains(beanName))`: 成立当前 bean 尚未创建完成，当前 bean 是依赖 exposedObject 的 bean，返回 true
- `return false`: 创建完成返回 false
- `actualDependentBeans.add(dependentBean)`: 创建完成的 dependentBean 加入该集合
- `if (!actualDependentBeans.isEmpty())`: 条件成立说明有依赖于当前 bean 的 bean 实例创建完成，但是当前的 bean 还没创建完成返回，依赖当前 bean 的外部 bean 持有的是不完整的 bean，所以需要报错

- o `registerDisposableBeanIfNecessary`: 判断当前 bean 是否需要注册析构函数回调，当容器销毁时进行回调

- `if (!mbd.isPrototype() && requiresDestruction(bean, mbd))`
 - 如果是原型 prototype 不会注册析构回调，不会回调该函数，对象的回收由 JVM 的 GC 机制完成
 - `requiresDestruction()`:
 - `DisposableBeanAdapter.hasDestroyMethod(bean, mbd)`: bd 中定义了 DestroyMethod 返回 true
 - `hasDestructionAwareBeanPostProcessors()`: 后处理器框架决定是否进行析构回调
- `registerDisposableBean()`: 条件成立进入该方法，给当前单实例注册回调适配器，适配器内根据当前 bean 实例是继承接口（DisposableBean）还是自定义标签来判定具体调用哪个方法实现

- o `this.disposableBeans.put(beanName, bean)`: 向销毁集合添加实例

创建实例

`AbstractAutowireCapableBeanFactory.createBeanInstance(beanName, RootBeanDefinition, Object[] args)`

- o `resolveBeanClass(mbd, beanName)`: 确保 Bean 的 Class 真正的被加载
- o 判断类的访问权限是不是 public，不是进入下一个判断，是否允许访问类的 non-public 的构造方法，不允许则报错
- o `Supplier<?> instanceSupplier = mbd.getInstanceSupplier()`: 获取创建实例的函数，可以自定义，没有进入下面的逻辑
- o `if (mbd.getFactoryMethodName() != null)`: 判断 bean 是否设置了 factory-method 属性，优先使用
 - ，设置了该属性进入 factory-method 方法创建实例
- o `resolved = false`: 代表 bd 对应的构造信息是否已经解析成可以反射调用的构造方法
- o `autowireNecessary = false`: 是否自动匹配构造方法
- o `if(mbd.resolvedConstructorOrFactoryMethod != null)`: 获取 bd 的构造信息转化成反射调用的 method 信息
 - method 为 null 则 resolved 和 autowireNecessary 都为默认值 false
 - `autowireNecessary = mbd.constructorArgumentsResolved`: 构造方法有参数，设置为 true
- o **bd 对应的构造信息解析完成，可以直接反射调用构造方法了：**
 - `return autowireConstructor(beanName, mbd, null, null)`: 有参构造，根据参数匹配最优的构造器创建实例
 - `return instantiateBean(beanName, mbd)`: 无参构造方法通过反射创建实例
 - `SimpleInstantiationStrategy.instantiate()`: 真正用来实例化的函数（无论如何都会走到这一步）
 - `if (!bd.hasMethodOverrides())`: 没有方法重写覆盖

```

    BeanUtils.instantiateClass(constructorToUse) : 调用 Constructor.newInstance() 实例化
    ■ instantiateWithMethodInjection(bd, beanName, owner) : 有方法重写采用 CGLIB 实例化
    ■ BeanWrapper bw = new BeanWrapperImpl(beanInstance) : 包装成 BeanWrapper 类型的对象
    ■ return bw : 返回实例

○ ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName) : @Autowired 注解, 对应的后置处理器
AutowiredAnnotationBeanPostProcessor 逻辑
    ■ 配置了 lookup 的相关逻辑
    ■ this.candidateConstructorsCache.get(beanClass) : 从缓存中获取构造方法, 第一次获取为 null, 进入下面逻辑
    ■ rawCandidates = beanClass.getDeclaredConstructors() : 获取所有的构造器
    ■ Constructor<?> requiredConstructor = null : 唯一的选项构造器, @Autowired(required = "true") 时有值
    ■ for (Constructor<?> candidate : rawCandidates) : 遍历所有的构造器:
        ann = findAutowiredAnnotation(candidate) : 有三种注解中的一个会返回注解的属性
        ■ 遍历 this.autowiredAnnotationTypes 中的三种注解:

            this.autowiredAnnotationTypes.add(Autowired.class); //!!!!!!!
            this.autowiredAnnotationTypes.add(value.class);
            this.autowiredAnnotationTypes.add(...ClassUtils.forName("javax.inject.Inject"));

    ■ AnnotatedElementUtils.getMergedAnnotationAttributes(anno, type) : 获取注解的属性
    ■ if (attributes != null) return attributes : 任意一个注解属性不为空就注解返回
    if (anno == null) : 注解属性为空
    ■ userClass = ClassUtils.getUserClass(beanClass) : 如果当前 beanClass 是代理对象, 方法上就已经没有注解了, 所以获取原始的用户类型重新获取该构造器上的注解属性 (事务注解失效也是这个原理)
    if (anno != null) : 注解属性不为空了
        ■ required = determineRequiredStatus(anno) : 获取 required 属性的值
            ■ !anno.containsKey(this.requiredParameterName) || : 判断属性是否包含 required, 不包含进入后面逻辑
            ■ this.requiredParameterValue == anno.getBoolean(this.requiredParameterName) : 获取属性值返回
        ■ if (required) : 代表注解 @Autowired(required = true)
            ■ if (!candidates.isEmpty()) : true 代表只能有一个构造方法, 构造集合不是空代表可选的构造器不唯一, 报错
            ■ requiredConstructor = candidate : 把构造器赋值给 requiredConstructor
        ■ candidates.add(candidate) : 把当前构造方法添加至 candidates 集合
    if(candidate.getParameterCount() == 0) : 当前遍历的构造器的参数为 0 代表没有参数, 是默认构造器, 赋值给 defaultConstructor
    ■ candidateConstructors = candidates.toArray(new Constructor<?>[0]) : 将构造器转成数组返回

○ if(ctors != null) : 条件成立代表指定了构造方法数组
    mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR : 标签内 autowireMode 的属性值, 默认是 no, AUTOWIRE_CONSTRUCTOR 代表选择最优的构造方法
    ■ mbd.hasConstructorArgumentValues() : bean 信息中是否配置了构造参数的值
    ■ !ObjectUtils.isEmpty(args) : getBean 时, 指定了参数 arg

○ return autowireConstructor(beanName, mbd, ctors, args) : 选择最优的构造器进行创建实例 (复杂, 不建议研究)
    ■ beanFactory.initBeanWrapper(bw) : 向 BeanWrapper 中注册转换器, 向工厂中注册属性编辑器
    ■ Constructor<?> constructorToUse = null : 实例化反射构造器
        ArgumentsHolder argsHolderToUse : 实例化时真正去用的参数, 并持有对象
            ■ rawArguments 是转换前的参数, arguments 是类型转换完成的参数
            ■ Object[] argsToUse : 参数实例化时使用的参数
        ■ Object[] argsToResolve : 表示构造器参数做转换后的参数引用
        ■ if (constructorToUse != null && mbd.constructorArgumentsResolved):
            ■ 条件一成立说明当前 bd 生成的实例不是第一次, 缓存中有解析好的构造器方法可以直接拿来反射调用
            ■ 条件二成立说明构造器参数已经解析过了
        ■ argsToUse = resolvePreparedArguments() : argsToResolve 不是完全解析好的, 还需要继续解析
        ■ if (constructorToUse == null || argsToUse == null) : 条件成立说明缓存机制失败, 进入构造器匹配逻辑
        ■ Constructor<?>[] candidates = chosenCtors : chosenCtors 只有在构造方法上有 autowire 三种注解时才有数据
        ■ if (candidates == null) : candidates 为空就根据 beanClass 是否允许访问非公开的方法来获取构造方法
        ■ if (candidates.length == 1 && explicitArgs == null && !mbd.hasConstructorArgumentValues()) : 默认无参数
            bw.setBeanInstance(instantiate()) : 使用无参构造器反射调用, 创建出实例对象, 设置到 BeanWrapper 中去
        ■ boolean autowiring : 需要选择最优的构造器
        ■ args = mbd.getConstructorArgumentValues() : 获取参数值
            resolvedValues = new ConstructorArgumentValues() : 获取已经解析后的构造器参数值
                ■ final Map<Integer, ValueHolder> indexedArgumentValues : key 是 index, value 是值
                ■ final List<ValueHolder> genericArgumentValues : 没有 index 的值

```

```

minNrofArgs = resolveConstructorArguments(..., resolvedValues) : 从 bd 中解析并获取构造器参数的个数
    ■ valueResolver.resolveValueIfNecessary() : 将引用转换成真实的对象
    ■ resolvedValueHolder.setSource(valueHolder) : 将对象填充至 ValueHolder 中
    ■ resolvedValues.addIndexedArgumentValue() : 将参数值封装至 resolvedValues 中
    ■ AutowireUtils.sortConstructors(candidates) : 排序规则 public > 非公开的 > 参数多的 > 参数少的
    ■ int minTypeDiffWeight = Integer.MAX_VALUE : 值越低说明构造器参数列表类型和构造参数的匹配度越高
    ■ Set<Constructor<?>> ambiguousConstructors : 模棱两可的构造器，两个构造器匹配度相等时放入
    ■ for (Constructor<?> candidate : candidates) : 遍历筛选出 minTypeDiffWeight 最低的构造器
    ■ Class<?>[] paramTypes = candidate.getParameterTypes() : 获取当前处理的构造器的参数类型
    ■ if() : candidates 是排过序的，当前筛选出来的构造器的优先级一定是优先于后面的 constructor
    ■ if (paramTypes.length < minNrofArgs) : 需求的小于给的，不匹配
    ■ int typeDiffWeight : 获取匹配度
        ■ mbd.isLenientConstructorResolution() : true 表示 ambiguousConstructors 允许有数据，false 代表不允许有数据，有数据就报错
            (LenientConstructorResolution: 宽松的构造函数解析)
        ■ argsHolder.getTypeDifferenceWeight(paramTypes) : 选择参数转换前和转换后匹配度最低的，循环向父类中寻找该方法，直到寻找到 Object 类
    ■ if (typeDiffWeight < minTypeDiffWeight) : 条件成立说明当前循环处理的构造器更优
    ■ else if (constructorToUse != null && typeDiffWeight == minTypeDiffWeight) : 当前处理的构造器的计算出来的 DiffWeight 与上一次筛选出来的最
        优构造器的值一致，说明有模棱两可的情况
    ■ if (constructorToUse == null) : 未找到可以使用的构造器，报错
    ■ else if (ambiguousConstructors != null && !mbd.isLenientConstructorResolution()) : 模棱两可有数据，LenientConstructorResolution ==
        false，所以报错
    ■ argsHolderToUse.storeCache(mbd, constructorToUse) : 匹配成功，进行缓存，方便后来者使用该 bd 实例化
    ■ bw.setBeanInstance(instantiate(beanName, mbd, constructorToUse, argsToUse)) : 匹配成功调用 instantiate 创建出实例对象，设置到
        BeanWrapper 中去
○ return instantiateBean(beanName, mbd) : 默认走到这里

```

循环依赖

循环引用

循环依赖：是一个或多个对象实例之间存在直接或间接的依赖关系，这种依赖关系构成一个环形调用

Spring 循环依赖有四种：

- DependsOn 依赖加载【无法解决】（两种 Map）
- 原型模式 Prototype 循环依赖【无法解决】（正在创建集合）
- 单例 Bean 循环依赖：构造参数产生依赖【无法解决】（正在创建集合，getSingleton() 逻辑中）
- 单例 Bean 循环依赖：setter 产生依赖【可以解决】

解决循环依赖：提前引用，提前暴露创建中的 Bean

- Spring 先实例化 A，拿到 A 的构造方法反射创建出来 A 的早期实例对象，这个对象被包装成 ObjectFactory 对象，放入三级缓存
- 处理 A 的依赖数据，检查发现 A 依赖 B 对象，所以 Spring 就会去根据 B 类型到容器中去 getBean(B)，这里产生递归
- 拿到 B 的构造方法，进行反射创建出来 B 的早期实例对象，也会把 B 包装成 ObjectFactory 对象，放到三级缓存，处理 B 的依赖数据，检查发现 B 依赖了 A 对象，然后 Spring 就会去根据 A 类型到容器中去 getBean(A.class)
- 这时从三级缓存中获取到 A 的早期对象进入属性填充

循环依赖的三级缓存：

```

//一级缓存：存放所有初始化完成单实例 bean，单例池，key是beanName，value是对应的单实例对象引用
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

//二级缓存：存放实例化未进行初始化的 Bean，提前引用池
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

/** Cache of singleton factories: bean name to ObjectFactory. 3 */
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

```

- 为什么需要三级缓存？

- 循环依赖解决需要提前引用动态代理对象，AOP 动态代理是在 Bean 初始化后的后置处理中进行，这时的 bean 已经是成品对象。因为需要提前进行动态代理，三级缓存的 ObjectFactory 提前产生需要代理的对象，把提前引用放入二级缓存
- 如果只有二级缓存，提前引用就直接放入了一级缓存，然后 Bean 初始化完成后又会放入一级缓存，产生数据覆盖，导致提前引用的对象和一级缓存中的并不是同一个对象
- 一级缓存只能存放完整的单实例，为了保证 Bean 的生命周期不被破坏，不能将未初始化的 Bean 暴露到一级缓存
- 若存在循环依赖，后置处理不创建代理对象，真正创建代理对象的过程是在 getBean(B) 的阶段中
- 三级缓存一定会创建提前引用吗？
 - 出现循环依赖就会去三级缓存获取提前引用，不出现就不会，走正常的逻辑，创建完成直接放入一级缓存
 - 存在循环依赖，就创建代理对象放入二级缓存，如果没有增强方法就返回 createBeanInstance 创建的实例，因为 addSingletonFactory 参数中传入了实例化的 Bean，在 singletonFactory.getObject() 中返回给 singletonObject，所以存在循环依赖就一定会使用工厂，但是不一定创建的是代理对象，不需要增强就是原始对象

- wrapIfNecessary 一定创建代理对象吗？ (AOP 动态代理部分有源码解析)
 - 存在增强器会创建动态代理，不需要增强就不需要创建动态代理对象
 - 存在循环依赖会提前增强，初始化后不需要增强
- 什么时候将 Bean 的引用提前暴露给第三级缓存的 ObjectFactory 持有？
 - 实例化之后，依赖注入之前

```
createBeanInstance -> addSingletonFactory -> populateBean
```

源码解析

假如 A 依赖 B, B 依赖 A

- 当 A 创建实例后填充属性前，执行：

```
addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean))
```

```
// 添加给定的单例工厂以构建指定的单例
protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        // 单例池包含该Bean说明已经创建完成，不需要循环依赖
        if (!this.singletonObjects.containsKey(beanName)) {
            //加入三级缓存
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            // 从二级缓存移除，因为三个Map中都是一个对象，不能同时存在！
            this.registeredSingletons.add(beanName);
        }
    }
}
```

- 填充属性时 A 依赖 B，这时需要 getBean(B)，也会把 B 的工厂放入三级缓存，接着 B 填充属性时发现依赖 A，去进行第一次 getSingleton(A)

```
public Object getSingleton(String beanName) {
    return getSingleton(beanName, true); //为true代表允许拿到早期引用。
}
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // 在一级缓存中获取 beanName 对应的单实例对象。
    Object singletonObject = this.singletonObjects.get(beanName);
    // 单实例确实尚未创建；单实例正在创建，发生了循环依赖
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            // 从二级缓存获取
            singletonObject = this.earlySingletonObjects.get(beanName);
            // 二级缓存不存在，并且允许获取早期实例对象，去三级缓存查看
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    // 从三级缓存获取工厂对象，并得到 bean 的提前引用
                    singletonObject = singletonFactory.getObject();
                    // 【缓存升级】，放入二级缓存，提前引用池
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    // 从三级缓存移除该对象
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

- 从三级缓存获取 A 的 Bean: singletonFactory.getObject()，调用了 lambda 表达式的 getEarlyBeanReference 方法：

```
public Object getEarlyBeanReference(Object bean, String beanName) {
    Object cacheKey = getCacheKey(bean.getClass(), beanName);
    // 【向提前引用代理池 earlyProxyReferences 中添加该 Bean，防止对象被重新代理】
    this.earlyProxyReferences.put(cacheKey, bean);
    // 创建代理对象，createProxy
    return wrapIfNecessary(bean, beanName, cacheKey);
}
```

- B 填充了 A 的提前引用后会继续初始化直到完成，返回原始 A 的逻辑继续执行

AOP

注解原理

@EnableAspectJAutoProxy: AOP 注解驱动，给容器中导入 AspectJAutoProxyRegistrar

```
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {
    // 是否强制使用 CGLIB 创建代理对象
    // 配置文件方式: <aop:aspectj-autoproxy proxy-target-class="true"/>
    boolean proxyTargetClass() default false;

    // 将当前代理对象暴露到上下文内，方便代理对象内部的真实对象拿到代理对象
    // 配置文件方式: <aop:aspectj-autoproxy expose-proxy="true"/>
    boolean exposeProxy() default false;
}
```

AspectJAutoProxyRegistrar 在用来向容器中注册 **AnnotationAwareAspectJAutoProxyCreator**，以 BeanDefiantion 形式存在，在容器初始化时加载。AnnotationAwareAspectJAutoProxyCreator 间接实现了 InstantiationAwareBeanPostProcessor, Order 接口，该类会在 Bean 的实例化和初始化的前后起作用
工作流程：创建 IOC 容器，调用 refresh() 刷新容器， registerBeanPostProcessors(beanFactory) 阶段，通过 getBean() 创建 AnnotationAwareAspectJAutoProxyCreator 对象，在生命周期的初始化方法中执行回调 initBeanFactory() 方法初始化注册三个工具类：BeanFactoryAdvisorRetrievalHelperAdapter、ReflectiveAspectJAdvisorFactory、BeanFactoryAspectJAdvisorsBuilderAdapter

后置处理

Bean 初始化完成的执行后置处理器的方法：

```
public Object postProcessAfterInitialization(@Nullable Object bean, String bn) {
    if (bean != null) {
        // cachekey 是【beanName 或者加上 & 的 beanName】
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        if (this.earlyProxyReferences.remove(cacheKey) != bean) {
            // 去提前代理引用池中寻找该 key，不存在则创建代理
            // 如果存在则证明被代理过，则判断是否是当前的 bean，不是则创建代理
            return wrapIfNecessary(bean, bn, cachekey);
        }
    }
    return bean;
}
```

AbstractAutoProxyCreator.wrapIfNecessary(): 根据通知创建动态代理，没有通知直接返回原实例

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
    // 条件一般不成立，很少使用 TargetSourceCreator 去创建对象 BeforeInstantiation 阶段，doCreateBean 之前的阶段
    if (StringUtils.hasLength(beanName) && this.targetSourcedBeans.contains(beanName)) {
        return bean;
    }
    // advisedBeans 集合保存的是 bean 是否被增强过了
    // 条件成立说明当前 beanName 对应的实例不需要被增强处理，判断是在 BeforeInstantiation 阶段做的
    if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
        return bean;
    }
    // 条件一：判断当前 bean 类型是否是基础框架类型，这个类的实例不能被增强
    // 条件二：shouldSkip 判断当前 beanName 是否是 .ORIGINAL 结尾，如果是就跳过增强逻辑，直接返回
    if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {
        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }

    // 【查找适合当前 bean 实例的增强方法】（下一节详解）
    Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
    // 条件成立说明上面方法查询到适合当前 clazz 的通知
    if (specificInterceptors != DO_NOT_PROXY) {
        this.advisedBeans.put(cacheKey, Boolean.TRUE);
        // 根据查询到的增强创建代理对象（下一节详解）
        // 参数一：目标对象
        // 参数二：beanName
        // 参数三：匹配当前目标对象 clazz 的 Advisor 数据
        Object proxy = createProxy(
            bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));
        // 保存代理对象类型
        this.proxyTypes.put(cacheKey, proxy.getClass());
        // 返回代理对象
        return proxy;
    }
}
```

```

    }
    // 执行到这里说明没有查到通知，当前 bean 不需要增强
    this.advisedBeans.put(cacheKey, Boolean.FALSE);
    // 【返回原始的 bean 实例】
    return bean;
}

```

获取通知

AbstractAdvisorAutoProxyCreator.getAdvisesAndAdvisorsForBean(): 查找适合当前类实例的增强，并进行排序

```

protected Object[] getAdvisesAndAdvisorsForBean(Class<?> beanClass, String beanName, @Nullable TargetSource targetSource) {
    // 查询适合当前类型的增强通知
    List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
    if (advisors.isEmpty()) {
        // 增强为空直接返回 null，不需要创建代理
        return DO_NOT_PROXY;
    }
    // 不是空，转成数组返回
    return advisors.toArray();
}

```

AbstractAdvisorAutoProxyCreator.findEligibleAdvisors():

- candidateAdvisors = findCandidateAdvisors(): 获取当前容器内可以使用（所有）的 advisor，调用的是 AnnotationAwareAspectJAutoProxyCreator 类的方法，每个方法对应一个 Advisor
 - advisors = super.findCandidateAdvisors(): 查询出 XML 配置的所有 Advisor 类型
 - advisorNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(): 通过 BF 查询出来 BD 配置的 class 中是 Advisor 子类的 BeanName
 - advisors.add(): 使用 Spring 容器获取当前这个 Advisor 类型的实例
 - advisors.addAll(...buildAspectJAdvisors()): 获取所有添加 @Aspect 注解类中的 Advisor
- buildAspectJAdvisors(): 构建的方法，把 Advice 封装成 Advisor
 - beanNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(this.beanFactory, Object.class, true, false): 获取出容器内 Object 所有的 beanName，就是全部的
 - for (String beanName : beanNames): 遍历所有的 beanName，判断每个 beanName 对应的 Class 是否是 Aspect 类型，就是加了 @Aspect 注解的类
 - factory = new BeanFactoryAspectInstanceFactory(this.beanFactory, beanName): 使用工厂模式管理 Aspect 的元数据，关联的真实 @Aspect 注解的实例对象
 - classAdvisors = this.advisorFactory.getAdvisors(factory): 添加了 @Aspect 注解的类的通知信息
 - aspectClass: @Aspect 标签的类的 class
 - for (Method method : getAdvisorMethods(aspectClass)): 遍历不包括 @Pointcut 注解的方法
 - Advisor advisor = getAdvisor(method, lazySingletonAspectInstanceFactory, advisors.size(), aspectName): 将当前 method 包装成 Advisor 数据
 - AspectJExpressionPointcut expressionPointcut = getPointcut(): 获取切点表达式
 - return new InstantiationModelAwarePointcutAdvisorImpl(): 把 method 中 Advice 包装成 Advisor，Spring 中每个 Advisor 内部一定是持有一个 Advice 的，Advice 内部最重要的数据是当前 method 和 aspectInstanceFactory，工厂用来获取实例
 - this.instantiatedAdvice = instantiateAdvice(this.declaredPointcut): 实例化 Advice 对象，逻辑是获取注解信息，根据注解的不同生成对应的 Advice 对象
 - advisors.addAll(classAdvisors): 保存通过 @Aspect 注解定义的 Advisor 数据
 - this.aspectBeanNames = aspectNames: 将所有 @Aspect 注解 beanName 缓存起来，表示提取 Advisor 工作完成
 - return advisors: 返回 Advisor 列表
 - eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, ...): 选出匹配当前类的增强
 - if (candidateAdvisors.isEmpty()): 条件成立说明当前 Spring 没有可以操作的 Advisor
 - List<Advisor> eligibleAdvisors = new ArrayList<>(): 存放匹配当前 beanClass 的 Advisors 信息
 - for (Advisor candidate : candidateAdvisors): 遍历所有的 Advisor
 - if (canApply(candidate, clazz, hasIntroductions)): 判断遍历的 advisor 是否匹配当前的 class，匹配就加入集合
 - if (advisor instanceof PointcutAdvisor): 创建的 advisor 是 InstantiationModelAwarePointcutAdvisorImpl 类型
 - PointcutAdvisor pca = (PointcutAdvisor) advisor: 封装当前 Advisor
 - return canApply(pca.getPointcut(), targetClass, hasIntroductions): 重载该方法
 - if (!pc.getClassFilter().matches(targetClass)): 类不匹配 Pointcut 表达式，直接返回 false
 - methodMatcher = pc.getMethodMatcher(): 获取 Pointcut 方法匹配器，类匹配进行类中方法的匹配
 - Set<Class<?>> classes: 保存目标对象 class 和目标对象父类超类的接口和自身实现的接口
 - if (!Proxy.isProxyClass(targetClass)): 判断当前实例是不是代理类，确保 class 内存储的数据包括目标对象的 class 而不是代理类的 class
 - for (Class<?> clazz : classes): 检查目标 class 和上级接口的所有方法，查看是否会被方法匹配器匹配，如果有一个方法匹配成功，就说明目标对象 AOP 代理需要增强

- `specificMethod = AopUtils.getMostSpecificMethod(method, targetClass)`: 方法可能是接口的，判断当前类有没有该方法
 - `return (specificMethod != method && matchesMethod(specificMethod))`: **类和方法的匹配**, 不包括参数
 - `extendAdvisors(eligibleAdvisors)`: 在 eligibleAdvisors 列表的索引 0 的位置添加 DefaultPointcutAdvisor, **封装了 ExposeInvocationInterceptor 拦截器**
 - `eligibleAdvisors = sortAdvisors(eligibleAdvisors)`: **对拦截器进行排序**, 数值越小优先级越高, 高的排在前面
 - 实现 Ordered 或 PriorityOrdered 接口, PriorityOrdered 的级别要优先于 Ordered, 使用 OrderComparator 比较器
 - 使用 @Order (Spring 规范) 或 @Priority (JDK 规范) 注解, 使用 AnnotationAwareOrderComparator 比较器
 - ExposeInvocationInterceptor 实现了 PriorityOrdered, 所以总是排在第一位, MethodBeforeAdviceInterceptor 没实现任何接口, 所以优先级最低, 排在最后
 - `return eligibleAdvisors`: 返回拦截器链
-

创建代理

`AbstractAutoProxyCreator.createProxy()`: 根据增强方法创建代理对象

- `ProxyFactory proxyFactory = new ProxyFactory()`: **无参构造 ProxyFactory**, 此处讲解一下两种有参构造方法:
 - `public ProxyFactory(Object target)`:

```
public ProxyFactory(Object target) {
    // 将目标对象封装成 SingletonTargetSource 保存到父类的字段中
    setTarget(target);
    // 获取目标对象 class 所有接口保存到 AdvisedSupport 中的 interfaces 集合中
    setInterfaces(ClassUtils.getAllInterfaces(target));
}
```

`ClassUtils.getAllInterfaces(target)` 底层调用 `getAllInterfacesForClassAsSet(java.lang.Class<?>, java.lang.ClassLoader)`:

- `if (clazz.isInterface() && isVisible(clazz, classLoader))`:
 - 条件一: 判断当前目标对象是接口
 - 条件二: 检查给定的类在给定的 ClassLoader 中是否可见
- `Class<?>[] ifcs = current.getInterfaces()`: 拿到自己实现的接口, 拿不到接口实现的接口
- `current = current.getSuperclass()`: 递归寻找父类的接口, 去获取父类实现的接口
- `public ProxyFactory(Class<?> proxyInterface, Interceptor interceptor)`:

```
public ProxyFactory(Class<?> proxyInterface, Interceptor interceptor) {
    // 添加一个代理的接口
    addInterface(proxyInterface);
    // 添加通知, 底层调用 addAdvisor
    addAdvice(interceptor);
}
```

- `addAdvisor(pos, new DefaultPointcutAdvisor(advice))`: Spring 中 Advice 对应的接口就是 Advisor, Spring 使用 Advisor 包装 Advice 实例
- `proxyFactory.copyFrom(this)`: 填充一些信息到 proxyFactory
- `if (!proxyFactory.isProxyTargetClass())`: 条件成立说明 proxyTargetClass 为 false (默认), 两种配置方法:
 - `<aop:aspectj-autoproxy proxy-target-class="true"/>`: 强制使用 CGLIB
 - `@EnableAspectJAutoProxy(proxyTargetClass = true)`

`if (shouldProxyTargetClass(beanClass, beanName))`: 如果 bd 内有 preserveTargetClass = true, 那么这个 bd 对应的 class **创建代理时必须使用 CGLIB**, 条件成立设置 proxyTargetClass 为 true

`evaluateProxyInterfaces(beanClass, proxyFactory)`: **根据目标类判定是否可以使用 JDK 动态代理**

- `targetInterfaces = ClassUtils.getAllInterfacesForClass()`: 获取当前目标对象 class 和父类的全部实现接口
- `boolean hasReasonableProxyInterface = false`: 实现的接口中是否有一个合理的接口
- `if (!isConfigurationCallbackInterface(ifc) && !isInternalLanguageInterface(ifc) && ifc.getMethods().length > 0)`: 遍历所有的接口, 如果有任何一个接口满足条件, 设置 hRPI 变量为 true
 - 条件一: 判断当前接口是否是 Spring 生命周期内会回调的接口
 - 条件二: 接口不能是 GroovyObject、Factory、MockAccess 类型的
 - 条件三: 找到一个可以使用的被代理的接口
- `if (hasReasonableProxyInterface)`: **有合理的接口, 将这些接口设置到 proxyFactory 内**
- `proxyFactory.setProxyTargetClass(true)`: **没有合理的代理接口, 强制使用 CGLIB 创建对象**
- `advisors = buildAdvisors(beanName, specificInterceptors)`: 匹配目标对象 clazz 的 Advisors, 填充至 ProxyFactory
- `proxyFactory.setPreFiltered(true)`: 设置为 true 表示传递给 proxyFactory 的 Advisors 信息做过基础类和方法的匹配
- `return proxyFactory.getProxy(getProxyClassLoader())`: 创建代理对象

```
public Object getProxy() {
    return createAopProxy().getProxy();
}
```

`DefaultAopProxyFactory.createAopProxy(AdvisedSupport config)`: 参数是一个配置对象, 保存着创建代理需要的生产资料, 会加锁创建, 保证线程安全

```
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
```

```

// 条件二为 true 代表强制使用 CGLIB 动态代理
if (config.isOptimize() || config.isProxyTargetClass() || 
    // 条件三：被代理对象没有实现任何接口或者只实现了 SpringProxy 接口，只能使用 CGLIB 动态代理
    hasNoUserSuppliedProxyInterfaces(config)) {
    Class<?> targetClass = config.getTargetClass();
    if (targetClass == null) {
        throw new AopConfigException("");
    }
    // 条件成立说明 target 【是接口或者是已经被代理过的类型】，只能使用 JDK 动态代理
    if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
        return new JdkDynamicAopProxy(config); // 使用 JDK 动态代理
    }
    return new ObjenesisCglibAopProxy(config); // 使用 CGLIB 动态代理
}
else {
    return new JdkDynamicAopProxy(config); // 【有接口的情况下只能使用 JDK 动态代理】
}
}

```

JdkDynamicAopProxy.getProxy(java.lang.ClassLoader): 获取JDK的代理对象

```

public JdkDynamicAopProxy(AdvisedSupport config) throws AopConfigException {
    // 配置类封装到 JdkDynamicAopProxy.advised 属性中
    this.advised = config;
}
public Object getProxy(@Nullable ClassLoader classLoader) {
    // 获取需要代理的接口数组
    Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, true);

    // 查找当前所有的需要代理的接口，看是否有 equals 方法和 hashCode 方法，如果有就做一个标记
    findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);

    // 该方法最终返回一个代理类对象
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    // classLoader: 类加载器 proxiedInterfaces: 生成的代理类，需要实现的接口集合
    // this JdkDynamicAopProxy 实现了 InvocationHandler
}

```

AopProxyUtils.completeProxiedInterfaces(this.advised, true): 获取代理的接口数组，并添加SpringProxy接口

- specifiedInterfaces = advised.getProxiedInterfaces(): 从 ProxyFactory 中拿到所有的 target 提取出来的接口
- if (specifiedInterfaces.length == 0): 如果没有实现接口，检查当前 target 是不是接口或者已经是代理类，封装到 ProxyFactory 的 interfaces 集合中
- addSpringProxy = !advised.isInterfaceProxied(SpringProxy.class): 判断目标对象所有接口中是否有 SpringProxy 接口，没有的话需要添加，这个接口标识这个代理类型是 Spring 管理的
 - addAdvised = !advised.isOpaque() && !advised.isInterfaceProxied(Advised.class): 判断目标对象的所有接口，是否已经有 Advised 接口
 - addDecoratingProxy = (decoratingProxy && !advised.isInterfaceProxied(DecoratingProxy.class)): 判断目标对象的所有接口，是否已经有 DecoratingProxy 接口
 - int nonUserIfcCount = 0: 非用户自定义的接口数量，接下来要添加上面的三个接口了
 - proxiedInterfaces = new Class<?>[specifiedInterfaces.length + nonUserIfcCount]: 创建一个新的 class 数组，长度是原目标对象提取出来的接口数量和 Spring 追加的数量，然后进行 System.arraycopy 拷贝到新数组中
 - int index = specifiedInterfaces.length: 获取原目标对象提取出来的接口数量，当作 index
 - if(addSpringProxy): 根据上面三个布尔值把接口添加到新数组中
 - return proxiedInterfaces: 返回追加后的接口集合

JdkDynamicAopProxy.findDefinedEqualsAndHashCodeMethods(): 查找在任何定义在接口中的 equals 和 hashCode 方法

- for (Class<?> proxiedInterface : proxiedInterfaces): 遍历所有的接口
- Method[] methods = proxiedInterface.getDeclaredMethods(): 获取接口中的所有方法
- for (Method method : methods): 遍历所有的方法
 - if (AopUtils.isEqualsMethod(method)): 当前方法是 equals 方法，把 equalsDefined 置为 true
 - if (AopUtils.isHashCodeMethod(method)): 当前方法是 hashCode 方法，把 hashCodeDefined 置为 true
 - if (this.equalsDefined && this.hashCodeDefined): 如果有一个接口中有这两种方法，直接返回

方法增强

main() 函数中调用用户方法，会进入代理对象的 invoke 方法

JdkDynamicAopProxy 类中的 invoke 方法是真正执行代理方法

```

// proxy: 代理对象，method: 目标对象的方法，args: 目标对象方法对应的参数
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object oldProxy = null;
    boolean setProxyContext = false;

    // advised 就是初始化 JdkDynamicAopProxy 对象时传入的变量
}

```

```

TargetSource targetSource = this.advised.targetSource;
Object target = null;

try {
    // 条件成立说明代理类实现的接口没有定义 equals 方法，并且当前 method 调用 equals 方法,
    // 就调用 JdkDynamicAopProxy 提供的 equals 方法
    if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
        return equals(args[0]);
    } //.....

    Object retval;
    // 需不需要暴露当前代理对象到 AOP 上下文内
    if (this.advised.exposeProxy) {
        // 【把代理对象设置到上下文环境】
        oldProxy = AopContext.setCurrentProxy(proxy);
        setProxyContext = true;
    }

    // 根据 targetSource 获取真正的代理对象
    target = targetSource.getTarget();
    Class<?> targetClass = (target != null ? target.getClass() : null);

    // 查找【适合该方法的增强】，首先从缓存中查找，查找不到进入主方法【下文详解】
    List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

    // 拦截器链是空，说明当前 method 不需要被增强
    if (chain.isEmpty()) {
        Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
        retval = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
    }
    else {
        // 有匹配当前 method 的方法拦截器，要做增强处理，把方法信息封装到方法调用器里
        MethodInvocation invocation =
            new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
        // 【拦截器链驱动方法，核心】
        retval = invocation.proceed();
    }

    Class<?> returnType = method.getReturnType();
    if (retval != null && retval == target &&
        returnType != Object.class && returnType.isInstance(proxy) &&
        !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
        // 如果目标方法返回目标对象，这里做个普通替换返回代理对象
        retval = proxy;
    }

    // 返回执行的结果
    return retval;
}
finally {
    if (target != null && !targetSource.isStatic()) {
        targetSource.releaseTarget(target);
    }
    // 如果允许了提前暴露，这里需要设置为初始状态
    if (setProxyContext) {
        // 当前代理对象已经完成工作，【把原始对象设置回上下文】
        AopContext.setCurrentProxy(oldProxy);
    }
}
}

```

this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass): 查找适合该方法的增强，首先从缓存中查找，获取通知时是从全部增强中获取适合当前类的，这里是**从当前类的中获取适合当前方法的增强**

- AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance()：向容器注册适配器，**可以将非 Advisor 类型的增强，包装成为 Advisor，将 Advisor 类型的增强提取出来对应的 MethodInterceptor**

- instance = new DefaultAdvisorAdapterRegistry()：该对象向容器中注册了 MethodBeforeAdviceAdapter、AfterReturningAdviceAdapter、ThrowsAdviceAdapter 三个适配器
- Advisor 中持有 Advice 对象

```

public interface Advisor {
    Advice getAdvice();
}

```

- advisors = config.getAdvisors()：获取 ProxyFactory 内部持有的增强信息
- interceptorList = new ArrayList<>(advisors.length)：拦截器列表有 5 个，1 个 ExposeInvocation 和 4 个增强器
- actualClass = (targetClass != null ? targetClass : method.getDeclaringClass())：真实的目标对象类型
- Boolean hasIntroductions = null：引介增强，不关心
- for (Advisor advisor : advisors)：遍历所有的 advisor 增强

- o `if (advisor instanceof PointcutAdvisor)`: 条件成立说明当前 Advisor 是包含切点信息的，进入匹配逻辑


```
pointcutAdvisor = (PointcutAdvisor) advisor; 转成可以获取到切点信息的接口
if(config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass))：当前代理被预处理，或者当前被代理的
class 对象匹配当前 Advisor 成功，只是 class 匹配成功
  ■ mm = pointcutAdvisor.getPointcut().getMethodMatcher()：获取切点的方法匹配器，不考虑引介增强
  ■ match = mm.matches(method, actualClass)：静态匹配成功返回 true，只关注于处理类及其方法，不考虑参数
  ■ if (match)：如果静态切点检查是匹配的，在运行的时候才进行动态切点检查，会考虑参数匹配（代表传入了参数）。如果静态匹配失败，直接不需要进行参数匹配，提高了工作效率
```
- o `interceptors = registry.getInterceptors(advisor)`: 提取出当前 advisor 内持有的 advice 信息


```
  ■ Advice advice = advisor.getAdvice()：获取增强方法
  ■ if (advice instanceof MethodInterceptor)：当前 advice 是 MethodInterceptor 直接加入集合
  ■ for (AdvisorAdapter adapter : this.adapters)：遍历三个适配器进行匹配（初始化时创建的），匹配成功创建对应的拦截器返回，以
MethodBeforeAdviceAdapter 为例
    if (adapter.supportsAdvice(advice))：判断当前 advice 是否是对应的 MethodBeforeAdvice
  interceptors.add(adapter.getInterceptor(advisor))：条件成立就往拦截器链中添加 advisor
    ■ advice = (MethodBeforeAdvice) advisor.getAdvice()：获取增强方法
    ■ return new MethodBeforeAdviceInterceptor(advice)：封装成 MethodBeforeAdviceInterceptor 返回
```
- o `interceptorList.add(new InterceptorAndDynamicMethodMatcher(interceptor, mm))`: 向拦截器链添加动态匹配器


```
interceptorList.addAll(Arrays.asList(interceptors))：将当前 advisor 内部的方法拦截器追加到 interceptorList
```
- o `interceptors = registry.getInterceptors(advisor)`: 进入 else 的逻辑，说明当前 Advisor 匹配全部 class 的全部 method，全部加入到 interceptorList
- o `return interceptorList`: 返回 method 方法的拦截器链

`RetVal = invocation.proceed(); 拦截器链驱动方法`

- o `if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1)`: 条件成立说明方法拦截器全部都已经调用过了（index 从 -1 开始累加），接下来需要执行目标对象的目标方法


```
return invokeJoinpoint()：调用连接点（目标）方法
```
- o `this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex)`: 获取下一个方法拦截器
- o `if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher)`: 需要运行时匹配


```
if (dm.methodMatcher.matches(this.method, targetClass, this.arguments))：判断是否匹配成功
  ■ return dm.interceptor.invoke(this)：匹配成功，执行方法
  ■ return proceed()：匹配失败跳过当前拦截器
```
- o `return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this)`: 一般方法拦截器都会执行到该方法，此方法内继续执行 proceed() 完成责任链的驱动，直到最后一个 MethodBeforeAdviceInterceptor 调用前置通知，然后调用 mi.proceed()，发现是最后一个拦截器就直接执行连接点（目标方法），return 到上一个拦截器的 mi.proceed() 处，依次返回到责任链的上一个拦截器执行通知方法

图示先从上往下建立链，然后从下往上依次执行，责任链模式

- o 正常执行：（环绕通知） → 前置通知 → 目标方法 → 后置通知 → 返回通知
- o 出现异常：（环绕通知） → 前置通知 → 目标方法 → 后置通知 → 异常通知
- o MethodBeforeAdviceInterceptor 源码：

```
public Object invoke(MethodInvocation mi) throws Throwable {
    // 先执行通知方法，再驱动责任链
    this.advice.before(mi.getMethod(), mi getArguments(), mi.getThis());
    // 开始驱动目标方法执行，执行完后返回到这，然后继续向上层返回
    return mi.proceed();
}
```

AfterReturningAdviceInterceptor 源码：没有任何异常处理机制，直接抛给上层

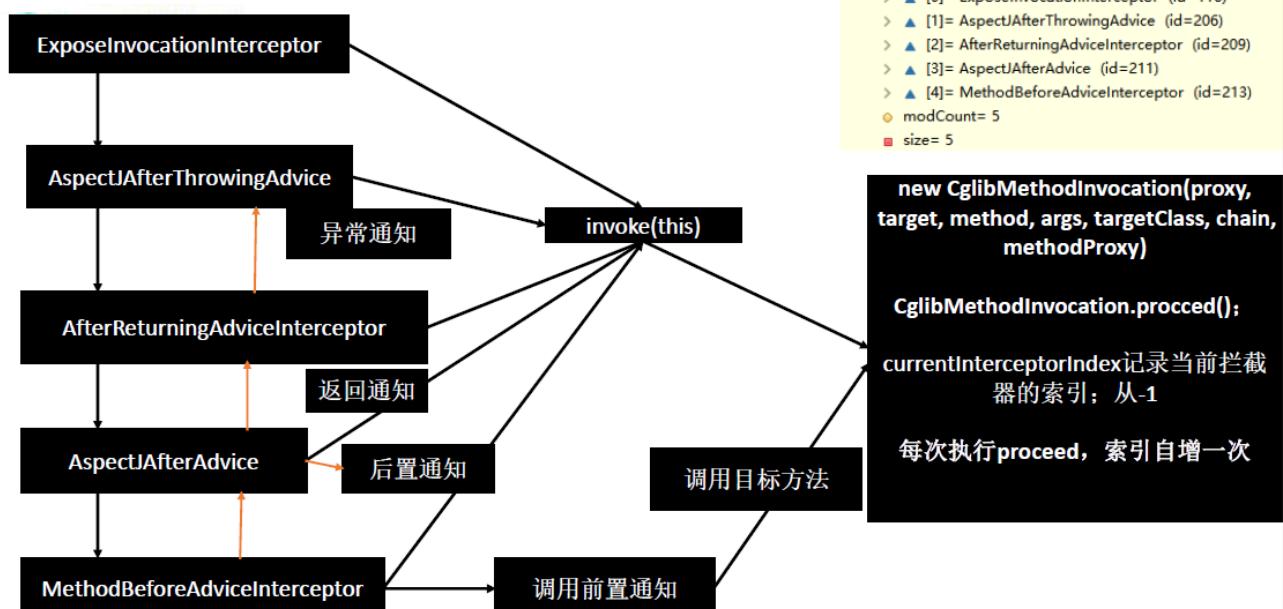
```
public Object invoke(MethodInvocation mi) throws Throwable {
    // 先驱动责任链，再执行通知方法
    Object retVal = mi.proceed();
    this.advice.afterReturning(retVal, mi.getMethod(), mi getArguments(), mi.getThis());
    return retVal;
}
```

AspectJAfterThrowingAdvice 执行异常处理：

```

public Object invoke(MethodInvocation mi) throws Throwable {
    try {
        // 默认直接驱动责任链
        return mi.proceed();
    }
    catch (Throwable ex) {
        // 出现错误才执行该方法
        if (shouldInvokeOnThrowing(ex)) {
            invokeAdviceMethod(getJoinPointMatch(), null, ex);
        }
        throw ex;
    }
}

```



参考视频：<https://www.bilibili.com/video/BV1gW411W7wy>.

事务

解析方法

标签解析

```
<tx:annotation-driven transaction-manager="txManager"/>
```

容器启动时会根据注解注册对应的解析器：

```

public class TxNamespaceHandler extends NamespaceHandlerSupport {
    public void init() {
        registerBeanDefinitionParser("advice", new TxAdviceBeanDefinitionParser());
        // 注册解析器
        registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());
        registerBeanDefinitionParser("jta-transaction-manager", new JtaTransactionManagerBeanDefinitionParser());
    }
    protected final void registerBeanDefinitionParser(String elementName, BeanDefinitionParser parser) {
        this.parsers.put(elementName, parser);
    }
}

```

获取对应的解析器 NamespaceHandlerSupport#findParserForElement:

```

private BeanDefinitionParser findParserForElement(Element element, ParserContext parserContext) {
    String localName = parserContext.getDelegate().getLocalName(element);
    // 获取对应的解析器
    BeanDefinitionParser parser = this.parsers.get(localName);
    // ...
    return parser;
}

```

调用解析器的方法对 XML 文件进行解析：

```

public BeanDefinition parse(Element element, ParserContext parserContext) {
    // 向Spring容器注册了一个 BD -> TransactionalEventListerFactory.class
    registerTransactionalEventListerFactory(parserContext);
    String mode = element.getAttribute("mode");
    if ("aspectj".equals(mode)) {
        // mode="aspectj"
        registerTransactionAspect(element, parserContext);
        if (ClassUtils.isPresent("javax.transaction.Transactional", getClass().getClassLoader())) {
            registerJtaTransactionAspect(element, parserContext);
        }
    } else {
        // mode="proxy", 默认逻辑, 不配置 mode 时
        // 用来向容器中注入一些 BeanDefinition, 包括事务增强器、事务拦截器、注解解析器
        AopAutoProxyConfigurer.configureAutoProxyCreator(element, parserContext);
    }
    return null;
}

```

注解解析

@EnableTransactionManagement 导入 TransactionManagementConfigurationSelector，该类给 Spring 容器中两个组件：

```

protected String[] selectImports(AdviceMode adviceMode) {
    switch (adviceMode) {
        // 导入 AutoProxyRegistrar 和 ProxyTransactionManagementConfiguration (默认)
        case PROXY:
            return new String[] {AutoProxyRegistrar.class.getName(),
                ProxyTransactionManagementConfiguration.class.getName()};
        // 导入 AspectJTransactionManagementConfiguration (与声明式事务无关)
        case ASPECTJ:
            return new String[] {determineTransactionAspectClass()};
        default:
            return null;
    }
}

```

AutoProxyRegistrar：给容器中注册 InfrastructureAdvisorAutoProxyCreator，利用后置处理器机制拦截 bean 以后包装并返回一个代理对象，代理对象中保存所有的拦截器，利用拦截器的链式机制依次进入每一个拦截器中进行拦截执行（就是 AOP 原理）

ProxyTransactionManagementConfiguration：是一个 Spring 的事务配置类，注册了三个 Bean：

- BeanFactoryTransactionAttributeSourceAdvisor：事务驱动，利用注解 @Bean 把该类注入到容器中，该增强器有两个字段：
- TransactionAttributeSource：解析事务注解的相关信息，真实类型是 AnnotationTransactionAttributeSource，构造方法中注册了三个注解解析器，解析 Spring、JTA、Ejb3 三种类型的事务注解
- TransactionInterceptor：事务拦截器，代理对象执行拦截器方法时，调用 TransactionInterceptor 的 invoke 方法，底层调用 TransactionAspectSupport.invokeWithinTransaction()，通过 PlatformTransactionManager 控制着事务的提交和回滚，所以事务的底层原理就是通过 AOP 动态织入，进行事务开启和提交

注解解析器 SpringTransactionAnnotationParser 解析 @Transactional 注解：

```

protected TransactionAttribute parseTransactionAnnotation(AnnotationAttributes attributes) {
    RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();
    // 从注解信息中获取传播行为
    Propagation propagation = attributes.getEnum("propagation");
    rbta.setPropagationBehavior(propagation.value());
    // 获取隔离级别
    Isolation isolation = attributes.getEnum("isolation");
    rbta.setIsolationLevel(isolation.value());
    rbta.setTimeout(attributes.getNumber("timeout").intValue());
    // 从注解信息中获取 readOnly 参数
    rbta.setReadOnly(attributes.getBoolean("readOnly"));
    // 从注解信息中获取 value 信息并且设置 qualifier，表示当前事务指定使用的【事务管理器】
    rbta.setQualifier(attributes.getString("value"));
    // 【存放的是 rollback 条件】，回滚规则放在这个集合
    List<RollbackRuleAttribute> rollbackRules = new ArrayList<>();
}

```

```

// 表示事务碰到哪些指定的异常才进行回滚，不指定的话默认是 RuntimeException/Error 非检查型异常来回滚
for (Class<?> rbRule : attributes.getClassArray("rollbackFor")) {
    rollbackRules.add(new RollbackRuleAttribute(rbRule));
}
// 与 rollbackFor 功能相同
for (String rbRule : attributes.getStringArray("rollbackForClassName")) {
    rollbackRules.add(new RollbackRuleAttribute(rbRule));
}
// 表示事务碰到指定的 exception 实现对象不进行回滚，否则碰到其他的class就进行回滚
for (Class<?> rbRule : attributes.getClassArray("noRollbackFor")) {
    rollbackRules.add(new NoRollbackRuleAttribute(rbRule));
}
for (String rbRule : attributes.getStringArray("noRollbackForClassName")) {
    rollbackRules.add(new NoRollbackRuleAttribute(rbRule));
}
// 设置回滚规则
rbta.setRollbackRules(rollbackRules);

return rbta;
}

```

驱动方法

TransactionInterceptor 事务拦截器的核心驱动方法：

```

public Object invoke(MethodInvocation invocation) throws Throwable {
    // targetClass 是需要被事务增强器增强的目标类, invocation.getThis() → 目标对象 → 目标类
    Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);
    // 参数一是目标方法，参数二是目标类，参数三是方法引用，用来触发驱动方法
    return invokeWithinTransaction(invocation.getMethod(), targetClass, invocation::proceed);
}

protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
                                         final InvocationCallback invocation) throws Throwable {

    // 事务属性源信息
    TransactionAttributeSource tas = getTransactionAttributeSource();
    // 提取 @Transactional 注解信息，txAttr 是注解信息的承载对象
    final TransactionAttribute txAttr = (tas != null ? tas.getTransactionAttribute(method, targetClass) : null);
    // 获取 Spring 配置的事务管理器
    // 首先会检查是否通过XML或注解配置 qualifier，没有就尝试去容器获取，一般情况下为 DatasourceTransactionManager
    final PlatformTransactionManager tm = determineTransactionManager(txAttr);
    // 权限定类名.方法名，该值用来做事务名称使用
    final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

    // 条件成立说明是【声明式事务】
    if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
        // 用来【开启事务】
        TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);

        Object retVal;
        try {
            // This is an 【around advice】: Invoke the next interceptor in the chain.
            // 环绕通知，执行目标方法（方法引用方式，invocation::proceed，还是调用 proceed）
            retVal = invocation.proceedWithInvocation();
        }
        catch (Throwable ex) {
            // 执行业务代码时抛出异常，执行回滚逻辑
            completeTransactionAfterThrowing(txInfo, ex);
            throw ex;
        }
        finally {
            // 清理事务的信息
            cleanupTransactionInfo(txInfo);
        }
        // 提交事务的入口
        commitTransactionAfterReturning(txInfo);
        return retVal;
    }
    else {
        // 编程式事务，省略
    }
}

```

开启事务

事务绑定

创建事务的方法:

```
protected TransactionInfo createTransactionIfNecessary(@Nullable PlatformTransactionManager tm,
                                                       @Nullable TransactionAttribute txAttr,
                                                       final String joinpointIdentification) {

    // If no name specified, apply method identification as transaction name.
    if (txAttr != null && txAttr.getName() == null) {
        // 事务的名称: 类的权限定名.方法名
        txAttr = new DelegatingTransactionAttribute(txAttr) {
            @Override
            public String getName() {
                return joinpointIdentification;
            }
        };
    }

    TransactionStatus status = null;
    if (txAttr != null) {
        if (tm != null) {
            // 通过事务管理器根据事务属性创建事务状态对象, 事务状态对象一般情况下包装着 事务对象, 当然也有可能是null
            // 方法上的注解为 @Transactional(propagation = NOT_SUPPORTED || propagation = NEVER) 时
            // 【下一节详解】
            status = tm.getTransaction(txAttr);
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("Skipping transactional joinpoint [" + joinpointIdentification +
                            "] because no transaction manager has been configured");
            }
        }
    }
    // 包装成一个上层的事务上下文对象
    return prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);
}
```

TransactionAspectSupport#prepareTransactionInfo: 为事务的属性和状态准备一个事务信息对象

- TransactionInfo txInfo = new TransactionInfo(tm, txAttr, joinpointIdentification); 创建事务信息对象
- txInfo.newTransactionStatus(status); 填充事务的状态信息
- txInfo.bindToThread(); 利用 ThreadLocal 把当前事务信息绑定到当前线程, 不同的事务信息会形成一个栈的结构
 - this.oldTransactionInfo = transactionInfoHolder.get(); 获取其他事务的信息存入 oldTransactionInfo
 - transactionInfoHolder.set(this); 将当前的事务信息设置到 ThreadLocalMap 中

事务创建

```
public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {
    // 获取事务的对象
    Object transaction = doGetTransaction();
    boolean debugEnabled = logger.isDebugEnabled();

    if (definition == null) {
        // Use defaults if no transaction definition given.
        definition = new DefaultTransactionDefinition();
    }
    // 条件成立说明当前是事务重入的情况, 事务中有 ConnectionHolder 对象
    if (isExistingTransaction(transaction)) {
        // a方法开启事务, a方法内调用b方法, b方法仍然加了 @Transactional 注解, 需要检查传播行为
        return handleExistingTransaction(definition, transaction, debugEnabled);
    }

    // 逻辑到这说明当前线程没有连接资源, 一个连接对应一个事务, 没有连接就相当于没有开启事务
    // 检查事务的延迟属性
    if (definition.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
        throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());
    }

    // 传播行为是 MANDATORY, 没有事务就抛出异常
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_MANDATORY) {
        throw new IllegalTransactionStateException();
    }
    // 需要开启事务的传播行为
```

```

    else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||
        definition.getPropagationBehavior() == TransactionDefinition.PROPAGATIONQUIRES_NEW ||
        definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
        // 什么也没挂起, 因为线程并没有绑定事务
        SuspendedResourcesHolder suspendedResources = suspend(null);
        try {
            // 是否支持同步线程事务, 一般是 true
            boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
            // 新建一个事务状态信息
            DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
            // 【启动事务】
            doBegin(transaction, definition);
            // 设置线程上下文变量, 方便程序运行期间获取当前事务的一些核心的属性, initSynchronization() 启动同步
            prepareSynchronization(status, definition);
            return status;
        }
        catch (RuntimeException | Error ex) {
            // 恢复现场
            resume(null, suspendedResources);
            throw ex;
        }
    }
    // 不支持事务的传播行为
    else {
        // Create "empty" transaction: no actual transaction, but potentially synchronization.
        boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
        // 创建事务状态对象
        // 参数 transaction 是 null 说明当前事务状态是未手动开启事, 线程上未绑定任何的连接资源, 业务程序执行时需要先去 datasource 获取的 conn, 是自动提交
        // 事务的, 不需要 Spring 再提交事务
        // 参数6 suspendedResources 是 null 说明当前事务状态未挂起任何事务, 当前事务执行到后置处理时不需要恢复现场
        return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);
    }
}

```

DataSourceTransactionManager#doGetTransaction: 真正获取事务的方法

- `DataSourceTransactionObject txObject = new DataSourceTransactionObject()`: 创建事务对象
- `txObject.setSavepointAllowed(isNestedAllowed())`: 设置事务对象是否支持保存点, 由事务管理器控制 (默认不支持)
- `ConnectionHolder conHolder = TransactionSynchronizationManager.getResource(obtainDataSource())`:
 - 从 ThreadLocal 中获取 conHolder 资源, 可能拿到 null 或者不是 null
 - 是 null: 举例

```

@Transaction
public void a() {...b.b()....}

```

- 不是 null: 执行 b 方法事务增强的前置逻辑时, 可以拿到 a 放进去的 conHolder 资源

```

@Transaction
public void b() {...}

```

- `txObject.setConnectionHolder(conHolder, false)`: 将 ConnectionHolder 保存到事务对象内, 参数二是 false 代表连接资源是上层事务共享的, 不是新建的连接资源
- `return txObject`: 返回事务的对象

DataSourceTransactionManager#doBegin: 事务开启的逻辑

- `txObject = (DataSourceTransactionObject) transaction`: 强转为事务对象
- 事务中没有数据库连接资源就要分配:
 - `Connection newCon = obtainDataSource().getConnection()`: 获取 JDBC 原生的数据库连接对象
 - `txObject.setConnectionHolder(new ConnectionHolder(newCon), true)`: 代表是新开启的事务, 新建的连接对象
- `previousIsolationLevel = dataSourceUtils.prepareConnectionForTransaction(con, definition)`: 修改连接属性
 - `if (definition != null && definition.isReadOnly())`: 注解 (或 XML) 配置了只读属性, 需要设置
 - `if (.definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT)`: 注解配置了隔离级别
 - `int currentIsolation = con.getTransactionIsolation()`: 获取连接的隔离级别
 - `previousIsolationLevel = currentIsolation`: 保存之前的隔离级别, 返回该值
 - `con.setTransactionIsolation(definition.getIsolationLevel())`: 将当前连接设置为配置的隔离级别
- `txObject.setPreviousIsolationLevel(previousIsolationLevel)`: 将 Conn 原来的隔离级别保存到事务对象, 为了释放 Conn 时重置回原状态
- `if (con.getAutoCommit())`: 默认会成立, 说明还没开启事务
 - `txObject.setMustRestoreAutoCommit(true)`: 保存 Conn 原来的事务状态
 - `con.setAutoCommit(false)`: 开启事务, JDBC 原生的方式
- `txObject.getConnectionHolder().setTransactionActive(true)`: 表示 Holder 持有的 Conn 已经手动开启事务了

- o `TransactionSynchronizationManager.bindResource(obtainDataSource(), txobject.getConnectionHolder())`: 将 ConnectionHolder 对象绑定到 ThreadLocal 内, 数据源为 key, 为了方便获取手动开启事务的连接对象去执行 SQL

事务重入

事务重入的核心处理逻辑:

```

private TransactionStatus handleExistingTransaction( TransactionDefinition definition,
                                                    Object transaction, boolean debugEnabled){
    // 传播行为是 PROPAGATION_NEVER, 需要以非事务方式执行操作, 如果当前事务存在则【抛出异常】
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NEVER) {
        throw new IllegalTransactionStateException();
    }
    // 传播行为是 PROPAGATION_NOT_SUPPORTED, 以非事务方式运行, 如果当前存在事务, 则【把当前事务挂起】
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NOT_SUPPORTED) {
        // 挂起事务
        Object suspendedResources = suspend(transaction);
        boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
        // 创建一个非事务的事务状态对象返回
        return prepareTransactionStatus(definition, null, false, newSynchronization, debugEnabled, suspendedResources);
    }
    // 开启新事物的逻辑
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATIONQUIRES_NEW) {
        // 【挂起当前事务】
        SuspendedResourcesHolder suspendedResources = suspend(transaction);
        // 【开启新事物】
    }
    // 传播行为是 PROPAGATION_NESTED, 嵌套事务
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
        // Spring 默认不支持内嵌事务
        // 【开启方式】: <property name="nestedTransactionAllowed" value="true">
        if (!isNestedTransactionAllowed()) {
            throw new NestedTransactionNotSupportedException();
        }

        if (useSavepointForNestedTransaction()) {
            // 为当前方法创建一个 TransactionStatus 对象,
            DefaultTransactionStatus status =
                prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);
            // 创建一个 JDBC 的保存点
            status.createAndHoldSavepoint();
            // 不需要使用同步, 直接返回
            return status;
        }
        else {
            // Usually only for JTA transaction, 开启一个新事务
        }
    }
}

// Assumably PROPAGATION_SUPPORTS or PROPAGATION_REQUIRED, 【使用当前的事务】
boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null);
}

```

挂起恢复

AbstractPlatformTransactionManager#suspend: 挂起事务, 并获得一个上下文信息对象

```

protected final SuspendedResourcesHolder suspend(@Nullable Object transaction) {
    // 事务是同步状态的
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
        List<Transactionsynchronization> suspendedSynchronizations = doSuspendSynchronization();
        try {
            Object suspendedResources = null;
            if (transaction != null) {
                // do it
                suspendedResources = doSuspend(transaction);
            }
            // 将上层事务绑定在线程上下文的变量全部取出来
            // ...
            // 通过被挂起的资源和上层事务的上下文变量, 创建一个【SuspendedResourcesHolder】返回
            return new SuspendedResourcesHolder(suspendedResources, suspendedSynchronizations,
                                                name, readOnly, isolationLevel, wasActive);
        } //...
    }
}

```

```

}
protected Object doSuspend(Object transaction) {
    DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
    // 将当前方法的事务对象 connectionHolder 属性置为 null, 不和上层共享资源
    // 当前方法有可能是不开启事务或者要开启一个独立的事务
    txObject.setConnectionHolder(null);
    // 【解绑在线程上的事务】
    return TransactionSynchronizationManager.unbindResource(obtainDataSource());
}

```

AbstractPlatformTransactionManager#resume: 恢复现场, 根据挂起资源去恢复线程上下文信息

```

protected final void resume(Object transaction, SuspendedResourcesHolder resourcesHolder) {
    if (resourcesHolder != null) {
        // 获取被挂起的事务资源
        Object suspendedResources = resourcesHolder.suspendedResources;
        if (suspendedResources != null) {
            // 绑定上一个事务的 ConnectionHolder 到线程上下文
            doResume(transaction, suspendedResources);
        }
        List<TransactionSynchronization> suspendedSynchronizations = resourcesHolder.suspendedSynchronizations;
        if (suspendedSynchronizations != null) {
            //....
            // 将线程上下文变量恢复为上一个事务的挂起现场
            doResumeSynchronization(suspendedSynchronizations);
        }
    }
}
protected void doResume(@Nullable Object transaction, Object suspendedResources) {
    // doSuspend 的逆动作, 【绑定资源】
    TransactionSynchronizationManager.bindResource(obtainDataSource(), suspendedResources);
}

```

提交回滚

回滚方式

```

protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txInfo, Throwable ex) {
    // 事务状态信息不为空进入逻辑
    if (txInfo != null && txInfo.getTransactionStatus() != null) {
        // 条件二成立 说明目标方法抛出的异常需要回滚事务
        if (txInfo.getTransactionAttribute() != null && txInfo.getTransactionAttribute().rollbackOn(ex)) {
            try {
                // 事务管理器的回滚方法
                txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
            }
            catch (TransactionSystemException ex2) {}
        }
        else {
            // 执行到这里, 说明当前事务虽然抛出了异常, 但是该异常并不会导致整个事务回滚
            try {
                // 提交事务
                txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
            }
            catch (TransactionSystemException ex2) {}
        }
    }
}
public boolean rollbackOn(Throwable ex) {
    // 继承自 RuntimeException 或 Error 的是【非检查型异常】, 才会归滚事务
    // 如果配置了其他回滚错误, 会获取到回滚规则 rollbackRules 进行判断
    return (ex instanceof RuntimeException || ex instanceof Error);
}

```

```

public final void rollback(TransactionStatus status) throws TransactionException {
    // 事务已经完成不需要回滚
    if (status.isCompleted()) {
        throw new IllegalTransactionStateException();
    }
    DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
    // 开始回滚事务
    processRollback(defStatus, false);
}

```

AbstractPlatformTransactionManager#processRollback: 事务回滚

- `triggerBeforeCompletion(status)`：用来做扩展逻辑，回滚前的前置处理
 - `if (status.hasSavepoint())`：条件成立说明当前事务是一个内嵌事务，当前方法只是复用了上层事务的一个内嵌事务
`status.rollbackToHoldSavepoint()`：内嵌事务加入事务时会创建一个保存点，此时恢复至保存点
 - `if (status.isNewTransaction())`：说明事务是当前连接开启的，需要去回滚事务
`doRollback(status)`：真正的回滚函数
 - `DataSourceTransactionObject txObject = status.getTransaction()`：获取事务对象
 - `Connection con = txObject.getConnectionHolder().getConnection()`：获取连接对象
 - `con.rollback()`：JDBC 的方式回滚事务
 - `else`：当前方法是共享的上层的事务，和上层使用同一个 Conn 资源，**共享的事务不能直接回滚，应该交给上层处理**
`doSetRollbackOnly(status)`：设置 `con.rollbackOnly = true`，线程回到上层事务 commit 时会检查该字段，然后执行回滚操作
 - `triggerAfterCompletion(status, TransactionSynchronization.STATUS_ROLLED_BACK)`：回滚的后置处理
 - `cleanupAfterCompletion(status)`：清理和恢复现场
-

提交方式

```
protected void commitTransactionAfterReturning(@Nullable TransactionInfo txInfo) {
    if (txInfo != null && txInfo.getTransactionStatus() != null) {
        // 事务管理器的提交方法
        txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
    }
}
```

```
public final void commit(TransactionStatus status) throws TransactionException {
    // 已经完成的事务不需要提交了
    if (status.isCompleted()) {
        throw new IllegalTransactionStateException();
    }
    DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
    // 条件成立说明是当前的业务强制回滚
    if (defStatus.isLocalRollbackOnly()) {
        // 回滚逻辑,
        processRollback(defStatus, false);
        return;
    }
    // 成立说明共享当前事务的【下层事务逻辑出错，需要回滚】
    if (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {
        // 如果当前事务还是事务重入，会继续抛给上层，最上层事务会进行真实的事务回滚操作
        processRollback(defStatus, true);
        return;
    }
    // 执行提交
    processCommit(defStatus);
}
```

AbstractPlatformTransactionManager#processCommit: 事务提交

- `prepareForCommit(status)`：前置处理
 - `if (status.hasSavepoint())`：条件成立说明当前事务是一个内嵌事务，只是复用了上层事务
`status.releaseHeldSavepoint()`：清理保存点，因为没有发生任何异常，所以保存点没有存在的意义了
 - `if (status.isNewTransaction())`：说明事务是归属于当前连接的，需要去提交事务
`doCommit(status)`：真正的提交函数
 - `Connection con = txObject.getConnectionHolder().getConnection()`：获取连接对象
 - `con.commit()`：JDBC 的方式提交事务
 - `doRollbackOnCommitException(status, ex)`：提交事务出错后进行回滚
 - `cleanupAfterCompletion(status)`：清理和恢复现场
-

清理现场

恢复上层事务：

```

protected void cleanupTransactionInfo(@Nullable TransactionInfo txInfo) {
    if (txInfo != null) {
        // 从当前线程的 ThreadLocal 获取上层的事务信息，将当前事务出栈，继续执行上层事务
        txInfo.restoreThreadLocalStatus();
    }
}
private void restoreThreadLocalStatus() {
    // Use stack to restore old transaction TransactionInfo.
    transactionInfoHolder.set(this.oldTransactionInfo);
}

```

当前层级事务结束时的清理：

```

private void cleanupAfterCompletion(DefaultTransactionStatus status) {
    // 设置当前方法的事务状态为完成状态
    status.setCompleted();
    if (status.isNewSynchronization()) {
        // 清理线程上下文变量以及扩展点注册的 sync
        TransactionSynchronizationManager.clear();
    }
    // 事务是当前线程开启的
    if (status.isNewTransaction()) {
        // 解绑资源
        doCleanupAfterCompletion(status.getTransaction());
    }
    // 条件成立说明当前事务执行的时候，【挂起了一个上层的事务】
    if (status.getSuspendedResources() != null) {
        Object transaction = (status.hasTransaction() ? status.getTransaction() : null);
        // 恢复上层事务现场
        resume(transaction, (SuspendedResourcesHolder) status.getSuspendedResources());
    }
}

```

DataSourceTransactionManager#doCleanupAfterCompletion：清理工作

- TransactionSynchronizationManager.unbindResource(obtainDataSource())：解绑数据库资源
- if (txobject.isMustRestoreAutoCommit())：是否恢复连接，Conn 归还到 DataSource，归还前需要恢复到申请时的状态
con.setAutoCommit(true)：恢复连接为自动提交
- DataSourceUtils.resetConnectionAfterTransaction(con, txobject.getPreviousIsolationLevel())：恢复隔离级别
- DataSourceUtils.releaseConnection(con, this.dataSource)：将连接归还给数据库连接池
- txobject.getConnectionHolder().clear()：清理 ConnectionHolder 资源

注解

Component

@Component 解析流程：

- 注解类启动容器的时，注册 ClassPathBeanDefinitionScanner 到容器，用来扫描 Bean 的相关信息

```

protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
    // 遍历指定的所有包，【这就相当于扫描了】
    for (String basePackage : basePackages) {
        // 读取当前包下的资源转换为 BeanDefinition，字节流的方式
        Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
        for (BeanDefinition candidate : candidates) {
            // 遍历，封装，类似于 XML 的解析方式，注册到容器中
            registerBeanDefinition(definationHolder, this.registry)
        }
    }
    return beanDefinitions;
}

```

- ClassPathScanningCandidateComponentProvider.findCandidateComponents()

```

public Set<BeanDefinition> findCandidateComponents(String basePackage) {
    if (this.componentsIndex != null && indexSupportsIncludeFilters()) {
        return addCandidateComponentsFromIndex(this.componentsIndex, basePackage);
    }
    else {
        return scanCandidateComponents(basePackage);
    }
}

```

```

private Set<BeanDefinition> scanCandidateComponents(String basePackage) {}

■ String packageSearchPath = ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX resolveBasePackage(basePackage) + '/' +
this.resourcePattern : 将 package 转化为 ClassLoader 类资源搜索路径 packageSearchPath, 例如: com.sea.spring.boot 转化为
classpath*:com/sea/spring/boot/**/*.class

■ resources = getResourcePatternResolver().getResources(packageSearchPath) : 加载路径下的资源

■ for (Resource resource : resources) : 遍历所有的资源

metadataReader = getMetadataReaderFactory().getMetadataReader(resource) : 获取元数据阅读器

if (isCandidateComponent(metadataReader)) : 当前类不匹配任何排除过滤器, 并且匹配一个包含过滤器, 返回 true

■ includeFilters 由 registerDefaultFilters() 设置初始值, 方法有 @Component, 没有 @Service, 因为 @Component 是 @Service 的元注解, Spring
在读取 @Service 时也读取了元注解, 并将 @Service 作为 @Component 处理

this.includeFilters.add(new AnnotationTypeFilter(Component.class))

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component // 拥有了 Component 功能
public @interface Service {}

candidates.add(sbd) : 添加到返回结果的 list

```

参考文章: <https://my.oschina.net/floor/blog/4325651>

Autowired

打开 @Autowired 源码, 注释上写 Please consult the javadoc for the AutowiredAnnotationBeanPostProcessor

AutowiredAnnotationBeanPostProcessor 间接实现 InstantiationAwareBeanPostProcessor, 就具备了实例化前后 (而不是初始化前后) 管理对象的能力, 实现了 BeanPostProcessor, 具有初始化前后管理对象的能力, 实现 BeanFactoryAware, 具备随时拿到 BeanFactory 的能力, 所以这个类具备一切后置处理器的能力

在容器启动, 为对象赋值的时候, 遇到 @Autowired 注解, 会用后置处理器机制, 来创建属性的实例, 然后再利用反射机制, 将实例化好的属性, 赋值给对象上, 这就是 Autowired 的原理

作用时机:

- Spring 在每个 Bean 实例化之后, 调用 AutowiredAnnotationBeanPostProcessor 的 postProcessMergedBeanDefinition() 方法, 查找该 Bean 是否有 @Autowired 注解, 进行相关元数据的获取
- Spring 在每个 Bean 调用 populateBean() 进行属性注入的时候, 即调用 postProcessProperties() 方法, 查找该 Bean 属性是否有 @Autowired 注解, 进行相关数据的填充

MVC

基本介绍

SpringMVC: 是一种基于 Java 实现 MVC 模型的轻量级 Web 框架

SpringMVC 优点:

- 使用简单
- 性能突出 (对比现有的框架技术)
- 灵活性强

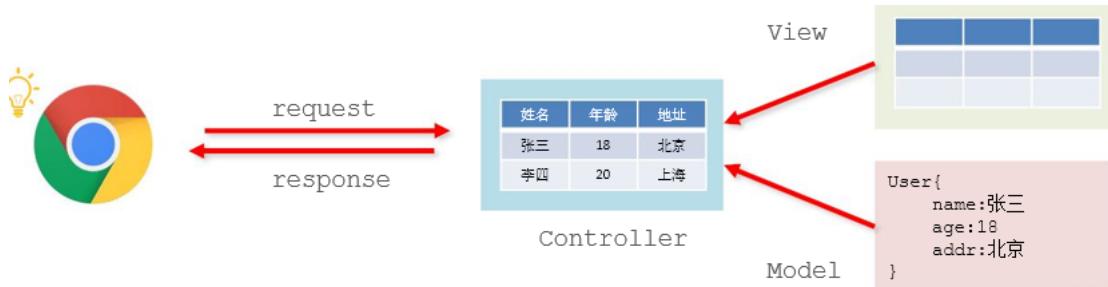
软件开发三层架构:

- 表现层: 负责数据展示
- 业务层: 负责业务处理
- 数据层: 负责数据操作



MVC (Model View Controller), 一种用于设计创建Web应用程序表现层的模式

- Model (模型) : 数据模型, 用于封装数据
- View (视图) : 页面视图, 用于展示数据
 - jsp
 - html
- Controller (控制器) : 处理用户交互的调度器, 用于根据用户需求处理程序逻辑
 - Servlet
 - SpringMVC



参考视频: <https://space.bilibili.com/37974444/>

基本配置

入门项目

流程分析:

- 服务器启动
 1. 加载 web.xml 中 DispatcherServlet
 2. 读取 spring-mvc.xml 中的配置, 加载所有 controller 包中所有标记为 bean 的类
 3. 读取 bean 中方法上方标注 @RequestMapping 的内容
- 处理请求
 1. DispatcherServlet 配置拦截所有请求 /
 2. 使用请求路径与所有加载的 @RequestMapping 的内容进行比对
 3. 执行对应的方法
 4. 根据方法的返回值在 webapp 目录中查找对应的页面并展示

代码实现:

- pom.xml 导入坐标

```

<modelVersion>4.0.0</modelVersion>

<groupId>demo</groupId>
<artifactId>spring_base_config</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
    <!-- servlet3.0规范的坐标 -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    
```

```

<scope>provided</scope>
</dependency>
<!--jsp坐标-->
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>
<!--spring的坐标-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
<!--springmvc的坐标-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
</dependencies>

<!--构建-->
<build>
    <!--设置插件-->
    <plugins>
        <!--具体的插件配置-->
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.1</version>
            <configuration>
                <port>80</port>
                <path>/</path>
            </configuration>
        </plugin>
    </plugins>
</build>

```

- 设定具体 Controller，控制层 java / controller / UserController

```

@Controller // @Component衍生注解
public class UserController {
    //设定当前方法的访问映射地址，等同于Servlet在web.xml中的配置
    @RequestMapping("/save")
    //设置当前方法返回值类型为string，用于指定请求完成后跳转的页面
    public String save(){
        System.out.println("user mvc controller is running ...");
        //设定具体跳转的页面
        return "success.jsp";
    }
}

```

- webapp / WEB-INF / web.xml，配置SpringMVC核心控制器，请求转发到对应的具体业务处理器Controller中（等同于Servlet配置）

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
          version="3.1">
    <!--配置Servlet-->
    <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!--加载Spring控制文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath*:spring-mvc.xml</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

- resources / spring-mvc.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- 扫描加载所有的控制类-->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <context:component-scan base-package="controller"/>
</beans>

```

加载控制

Controller 加载控制：SpringMVC 的处理器对应的 bean 必须按照规范格式开发，未避免加入无效的 bean 可通过 bean 加载过滤器进行包含设定或排除设定，表现层 bean 标注通常设定为 @Controller

- resources / spring-mvc.xml 配置

```

<context:component-scan base-package="com.seazean">
    <context:include-filter
        type="annotation"
        expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

- 静态资源加载（webapp 目录下的相关资源），spring-mvc.xml 配置，开启 mvc 命名空间

```

<!-- 放行指定类型静态资源配置方式-->
<mvc:resources mapping="/img/**" location="/img/" /> <!-- webapp/img/ 资源-->
<mvc:resources mapping="/js/**" location="/js/" />
<mvc:resources mapping="/css/**" location="/css/" />

<!-- SpringMVC 提供的通用资源配置放行方式，建议选择-->
<mvc:default-servlet-handler/>

```

- 中文乱码处理 SpringMVC 提供专用的中文字符过滤器，用于处理乱码问题。配置在 web.xml 里面

```

<!-- 乱码处理过滤器，与 Servlet 中使用的完全相同，差异之处在于处理器的类由 Spring 提供-->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

注解驱动

WebApplicationContext，生成 Spring 核心容器（主容器/父容器/根容器）

- 父容器：Spring 环境加载后形成的容器，包含 Spring 环境下的所有的 bean
- 子容器：当前 mvc 环境加载后形成的容器，不包含 Spring 环境下的 bean
- 子容器可以访问父容器中的资源，父容器不可以访问子容器的资源

EnableWebMvc 注解作用：

- 支持 ConversionService 的配置，可以方便配置自定义类型转换器
- 支持 @NumberFormat 注解格式化数字类型
- 支持 @DateTimeFormat 注解格式化日期数据，日期包括 Date、Calendar
- 支持 @Valid 的参数校验（需要导入 JSR-303 规范）
- 配合第三方 jar 包和 SpringMVC 提供的注解读写 XML 和 JSON 格式数据

纯注解开发：

- 使用注解形式转化 SpringMVC 核心配置文件为配置类 java / config / SpringMVCConfiguration.java

```

@Configuration
@ComponentScan(value = "com.seazean", includeFilters = @ComponentScan.Filter(
    type=FilterType.ANNOTATION,

```

```

        classes = {Controller.class} )
    }
//等同于<mvc:annotation-driven/>, 还不完全相同
@EnableWebMvc
public class SpringMVCConfiguration implements WebMvcConfigurer{
    //注解配置通用放行资源的格式 建议使用
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}

```

- 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 class ServletContainersInitConfig extends AbstractDispatcherServletInitializer {
    //创建Servlet容器时, 使用注解方式加载SPRINGMVC配置类中的信息,
    //并加载成WEB专用的ApplicationContext对象该对象放入了ServletContext范围,
    //在整个WEB容器中可以随时获取调用
    @Override
    protected WebApplicationContext createServletApplicationContext() {
        A.C.W.A ctx = new AnnotationConfigWebApplicationContext();
        ctx.register(SpringMVCConfiguration.class);
        return ctx;
    }

    //注解配置映射地址方式, 服务于SpringMVC的核心控制器DispatcherServlet
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    //乱码处理作为过滤器, 在servlet容器启动时进行配置
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        super.onStartup(servletContext);
        CharacterEncodingFilter cef = new CharacterEncodingFilter();
        cef.setEncoding("UTF-8");
        FilterRegistration.Dynamic registration = servletContext.addFilter("characterEncodingFilter", cef);
        registration.addMappingForUrlPatterns(EnumSet.of(
            DispatcherType.REQUEST,
            DispatcherType.FORWARD,
            DispatcherType.INCLUDE), false, "/*");
    }
}

```

请求映射

名称: @RequestMapping

类型: 方法注解、类注解

位置: 处理器类中的方法定义上方、处理器类定义上方

- 方法注解

作用: 绑定请求地址与对应处理方法间的关系

无类映射地址访问格式: <http://localhost/requestURL2>

```

@Controller
public class UserController {
    @RequestMapping("/requestURL2")
    public String requestURL2() {
        return "page.jsp";
    }
}

```

- 类注解

作用: 为当前处理器中所有方法设定公共的访问路径前缀

带有类映射地址访问格式, 将类映射地址作为前缀添加在实际映射地址前面: /user/requestURL1

最终返回的页面如果未设定绝对访问路径, 将从类映射地址所在目录中查找 webapp/user/page.jsp

```

@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/requestURL2")
    public String requestURL2() {
        return "page.jsp";
    }
}

```

- 常用属性

```

@RequestMapping(
    value="/requestURL3", //设定请求路径,与path属性、value属性相同
    method = RequestMethod.GET, //设定请求方式
    params = "name", //设定请求参数条件
    headers = "content-type=text/*", //设定请求消息头条件
    consumes = "text/*", //用于指定可以接收的请求正文类型(MIME类型)
    produces = "text/*" //用于指定可以生成的响应正文类型(MIME类型)
)
public String requestURL3() {
    return "/page.jsp";
}

```

基本操作

请求处理

普通类型

SpringMVC 将传递的参数封装到处理器方法的形参中，达到快速访问参数的目的

- 访问 URL: <http://localhost/requestParam1?name=seazean&age=14>

```

@Controller
public class UserController {
    @RequestMapping("/requestParam1")
    public String requestParam1(String name ,int age){
        System.out.println("name=" + name + ",age=" + age);
        return "page.jsp";
    }
}

```

```

<%@page pageEncoding="UTF-8" language="java" contentType="text/html;UTF-8" %>
<html>
<body>
    <h1>请求参数测试页面</h1>
</body>
</html>

```

@RequestParam 的使用:

- 类型: 形参注解
- 位置: 处理器类中的方法形参前方
- 作用: 绑定请求参数与对应处理方法形参间的关系
- 访问 URL: <http://localhost/requestParam2?userName=jock>

```

@RequestMapping("/requestParam2")
public String requestParam2(@RequestParam(
    name = "userName",
    required = true, //为true代表必须有参数
    defaultValue = "s") String name){
    System.out.println("name=" + name);
    return "page.jsp";
}

```

POJO类型

简单类型

当 POJO 中使用简单类型属性时，参数名称与 POJO 类属性名保持一致

- 访问 URL: <http://localhost/requestParam3?name=seazean&age=14>

```
@RequestMapping("/requestParam3")
public String requestParam3(User user){
    System.out.println("name=" + user.getName());
    return "page.jsp";
}
```

```
public class User {
    private String name;
    private Integer age;
    //....
}
```

参数冲突

当 POJO 类型属性与其他形参出现同名问题时，将被同时赋值，建议使用 @RequestParam 注解进行区分

- 访问 URL: <http://localhost/requestParam4?name=seazean&age=14>

```
@RequestMapping("/requestParam4")
public String requestParam4(User user, String age){
    System.out.println("user.age=" + user.getAge() + ",age=" + age); //14 14
    return "page.jsp";
}
```

复杂类型

当 POJO 中出现对象属性时，参数名称与对象层次结构名称保持一致

- 访问 URL: <http://localhost/requestParam5?address.province=beijing>

```
@RequestMapping("/requestParam5")
public String requestParam5(User user){
    System.out.println("user.address=" + user.getAddress().getProvince());
    return "page.jsp";
}
```

```
public class User {
    private String name;
    private Integer age;
    private Address address; //...
}
```

```
public class Address {
    private String province;
    private String city;
    private String address;
}
```

容器类型

POJO 中出现集合类型的处理方式

- 通过 URL 地址中同名参数，可以为 POJO 中的集合属性进行赋值，集合属性要求保存简单数据

访问 URL: <http://localhost/requestParam6?nick=jock1&nick=jockme&nick=zahc>

```

@RequestMapping("/RequestParam6")
public String requestParam6(User user){
    System.out.println("user=" + user);
    //user = User{name='null',age=null,nick={Jock1,Jockme,zahc}}
    return "page.jsp";
}

```

```

public class User {
    private String name;
    private Integer age;
    private List<String> nick;
}

```

- POJO 中出现 List 保存对象数据，参数名称与对象层次结构名称保持一致，使用数组格式描述集合中对象的位置访问 URL: [http://localhost/requestParam7?address\[0\].province=bj&addresses\[1\].province=tj](http://localhost/requestParam7?address[0].province=bj&addresses[1].province=tj)

```

@RequestMapping("/RequestParam7")
public String requestParam7(User user){
    System.out.println("user.addresses=" + user.getAddresses());
    //{Address{provice=bj,city='null',address='null'}},{Address{....}}
    return "page.jsp";
}

```

```

public class User {
    private String name;
    private Integer age;
    private List<Address> addresses;
}

```

- POJO 中出现 Map 保存对象数据，参数名称与对象层次结构名称保持一致，使用映射格式描述集合中对象位置
URL: [http://localhost/requestParam8?addressMap\['home'\].province=bj&addressMap\['job'\].province=tj](http://localhost/requestParam8?addressMap['home'].province=bj&addressMap['job'].province=tj)

```

@RequestMapping("/RequestParam8")
public String requestParam8(User user){
    System.out.println("user.addressMap=" + user.getAddressMap());
    //user.addressMap={home=Address{p=,c=,a=},job=Address{....}}
    return "page.jsp";
}

```

```

public class User {
    private Map<String,Address> addressMap;
    //....
}

```

数组集合

数组类型

请求参数名与处理器方法形参名保持一致，且请求参数数量 > 1个

- 访问 URL: <http://localhost/requestParam9?nick=Jockme&nick=zahc>

```

@RequestMapping("/RequestParam9")
public String requestParam9(String[] nick){
    System.out.println(nick[0] + "," + nick[1]);
    return "page.jsp";
}

```

集合类型

保存简单类型数据，请求参数名与处理器方法形参名保持一致，且请求参数数量 > 1个

- 访问 URL: <http://localhost/requestParam10?nick=Jockme&nick=zahc>

```

@RequestMapping("/RequestParam10")
public String requestParam10(@RequestParam("nick") List<String> nick){
    System.out.println(nick);
    return "page.jsp";
}

```

- 注意：SpringMVC 默认将 List 作为对象处理，赋值前先创建对象，然后将 nick 作为对象的属性进行处理。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 nick 属性，因此数据为空
解决方法：需要告知 SpringMVC 的处理器 nick 是一组数据，而不是一个单一属性。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

转换器

类型

开启转换配置：`<mvc:annotation-driven />`

作用：提供 Controller 请求转发，Json 自动转换等功能

如果访问 URL：<http://localhost/requestParam1?name=seazean&age=seazean>，会出现报错，类型转化异常

```
@RequestMapping("/requestParam1")
public String requestParam1(String name ,int age){
    System.out.println("name=" + name + ",age=" + age);
    return "page.jsp";
}
```

SpringMVC 对接收的数据进行自动类型转换，该工作通过 Converter 接口实现：

- 标量转换器
- 集合、数组相关转换器
- 默认转换器

日期

访问 URL：<http://localhost/requestParam11?date=2020/02/02>

```
ObjectToObjectConverter
@RequestMapping("/requestParam11")
public String requestParam11(Date date) {
    System.out.println("date=" + date);
    return "page.jsp";
}
```

如果访问 URL：<http://localhost/requestParam11?date=1999-09-09> 会报错，所以需要日期类型转换

- 声明自定义的转换格式并覆盖系统转换格式，配置 resources / spring-mvc.xml

```
<!--5.启用自定义Converter-->
<mvc:annotation-driven conversion-service="conversionService"/>
<!--1.设定格式类型Converter，注册为Bean，受SpringMVC管理-->
<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <!--2.自定义Converter格式类型设定，该设定使用的是同类型覆盖的思想-->
    <property name="formatters">
        <!--3.使用set保障相同类型的转换器仅保留一个，避免冲突-->
        <set>
            <!--4.设置具体的格式类型-->
            <bean class="org.springframework.format.datetime.DateFormatter">
                <!--5.类型规则-->
                <property name="pattern" value="yyyy-MM-dd"/>
            </bean>
        </set>
    </property>
</bean>
```

- @DateTimeFormat

类型：形参注解、成员变量注解

位置：形参前面 或 成员变量上方

作用：为当前参数或变量指定类型转换规则

```
public String requestParam12(@DateTimeFormat(pattern = "yyyy-MM-dd") Date date){
    System.out.println("date=" + date);
    return "page.jsp";
}
```

```
@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date date;
```

依赖注解驱动支持，xml 开启配置：

```
<mvc:annotation-driven />
```

自定义

自定义类型转换器，实现 Converter 接口或者直接容器中注入：

- 方式一：

```
public class WebConfig implements WebMvcConfigurer {
    @Bean
    public WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addFormatters(FormatterRegistry registry) {
                registry.addConverter(new Converter<String, Date>() {
                    @Override
                    public Date convert(String source) {
                        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
                        Date date = null;
                        //类型转换器无法预计使用过程中出现的异常，因此必须在类型转换器内部捕获,
                        //不允许抛出，框架无法预计此类异常如何处理
                        try {
                            date = df.parse(source);
                        } catch (ParseException e) {
                            e.printStackTrace();
                        }
                    }
                    return date;
                });
            }
        };
    }
}
```

- 方式二：

```
//本例中的泛型填写的是String, Date, 最终出现字符串转日期时, 该类型转换器生效
public class MyDateConverter implements Converter<String, Date> {
    //重写接口的抽象方法, 参数由泛型决定
    public Date convert(String source) {
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        Date date = null;
        //类型转换器无法预计使用过程中出现的异常, 因此必须在类型转换器内部捕获,
        //不允许抛出, 框架无法预计此类异常如何处理
        try {
            date = df.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
    return date;
}
```

配置 resources / spring-mvc.xml，注册自定义转换器，将功能加入到 SpringMVC 转换服务 ConverterService 中

```
<!--1.将自定义Converter注册为Bean, 受SpringMVC管理-->
<bean id="myDateConverter" class="converter.MyDateConverter"/>
<!--2.设定自定义Converter服务bean-->
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <!--3.注入所有的自定义Converter, 该设定使用的是同类型覆盖的思想-->
    <property name="converters">
        <!--4.set保障同类型转换器仅保留一个, 去重规则以Converter<S,T>的泛型为准-->
        <set>
            <!--5.具体的类型转换器-->
            <ref bean="myDateConverter"/>
        </set>
    </property>
</bean>

<!--开启注解驱动, 加载自定义格式化转换器对应的类型转换服务-->
<mvc:annotation-driven conversion-service="conversionService"/>
```

- 使用转换器

```
@RequestMapping("/requestParam12")
public String requestParam12(Date date){
    System.out.println(date);
    return "page.jsp";
}
```

响应处理

页面跳转

请求转发和重定向:

- 请求转发:

```
@Controller
public class UserController {
    @RequestMapping("/showPage1")
    public String showPage1() {
        System.out.println("user mvc controller is running ...");
        return "forward:/WEB-INF/page/page.jsp";
    }
}
```

- 请求重定向:

```
@RequestMapping("/showPage2")
public String showPage2() {
    System.out.println("user mvc controller is running ...");
    return "redirect:/WEB-INF/page/page.jsp";//不能访问WEB-INF下的资源
}
```

页面访问快捷设定 (InternalResourceViewResolver) :

- 展示页面的保存位置通常固定且结构相似，可以设定通用的访问路径简化页面配置，配置 spring-mvc.xml:

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/pages/">
    <property name="suffix" value=".jsp"/>
</bean>
```

- 简化

```
@RequestMapping("/showPage3")
public String showPage3() {
    System.out.println("user mvc controller is running...");
    return "page";
}

@RequestMapping("/showPage4")
public String showPage4() {
    System.out.println("user mvc controller is running...");
    return "forward:page";
}

@RequestMapping("/showPage5")
public String showPage5() {
    System.out.println("user mvc controller is running...");
    return "redirect:page";
}
```

- 如果未设定了返回值，使用 void 类型，则默认使用访问路径作页面地址的前缀后缀

```
//最简页面配置方式，使用访问路径作为页面名称，省略返回值
@RequestMapping("/showPage6")
public void showPage6() {
    System.out.println("user mvc controller is running ...");
}
```

数据跳转

ModelAndView 是 SpringMVC 提供的一个对象，该对象可以用作控制器方法的返回值（Model 同），实现携带数据跳转作用：

- 设置数据，向请求域对象中存储数据
- 设置视图，逻辑视图

代码实现：

- 使用 HttpServletRequest 类型形参进行数据传递

```
@Controller
public class BookController {
    @RequestMapping("/showPageAndData1")
    public String showPageAndData1(HttpServletRequest request) {
        request.setAttribute("name", "seazean");
        return "page";
    }
}
```

- 使用 Model 类型形参进行数据传递

```
@RequestMapping("/showPageAndData2")
public String showPageAndData2(Model model) {
    model.addAttribute("name", "seazean");
    Book book = new Book();
    book.setName("SpringMVC入门实战");
    book.setPrice(66.6d);
    //添加数据的方式，key对value
    model.addAttribute("book", book);
    return "page";
}
```

```
public class Book {
    private String name;
    private Double price;
}
```

- 使用 ModelAndView 类型形参进行数据传递，将该对象作为返回值传递给调用者

```
@RequestMapping("/showPageAndData3")
public ModelAndView showPageAndData3(ModelAndView modelAndView) {
    //ModelAndView mav = new ModelAndView(); 替换形参中的参数
    Book book = new Book();
    book.setName("SpringMVC入门案例");
    book.setPrice(66.66d);

    //添加数据的方式，key对value
    modelAndView.addObject("book", book);
    modelAndView.addObject("name", "Jockme");
    //设置页面的方式，该方法最后一次执行的结果生效
    modelAndView.setViewName("page");
    //返回值设定成 ModelAndView 对象
    return modelAndView;
}
```

- ModelAndView 扩展

```
// ModelAndView 对象支持转发的手工设定，该设定不会启用前缀后缀的页面拼接格式
@RequestMapping("/showPageAndData4")
public ModelAndView showPageAndData4(ModelAndView modelAndView) {
    modelAndView.setViewName("forward:/WEB-INF/page/page.jsp");
    return modelAndView;
}

// ModelAndView 对象支持重定向的手工设定，该设定不会启用前缀后缀的页面拼接格式
@RequestMapping("/showPageAndData5")
public ModelAndView showPageAndData5(ModelAndView modelAndView) {
    modelAndView.setViewName("redirect:page.jsp");
    return modelAndView;
}
```

JSON

注解：@ResponseBody

作用：将 Controller 的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到 Response 的 body 区。如果返回值是字符串，那么直接将字符串返回客户端；如果是一个对象，会将对象转化为 JSON，返回客户端

注意：当方法上面没有写 `ResponseBody`，底层会将方法的返回值封装为 `ModelAndView` 对象

- 使用 `HttpServletResponse` 对象响应数据

```
@Controller
public class AccountController {
    @RequestMapping("/showData1")
    public void showData1(HttpServletRequest response) throws IOException {
        response.getWriter().write("message");
    }
}
```

- 使用 `@ResponseBody` 将返回的结果作为响应内容（页面显示），而非响应的页面名称

```
@RequestMapping("/showData2")
@ResponseBody
public String showdata2(){
    return "[{'name':'Jock'}]";
}
```

- 使用 jackson 进行 json 数据格式转化

导入坐标：

```
<!--json相关坐标3个-->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.0</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.0</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.9.0</version>
</dependency>
```

```
@RequestMapping("/showData3")
@ResponseBody
public String showData3() throws JsonProcessingException {
    Book book = new Book();
    book.setName("SpringMVC入门案例");
    book.setPrice(66.66d);

    ObjectMapper om = new ObjectMapper();
    return om.writeValueAsString(book);
}
```

- 使用 SpringMVC 提供的消息类型转换器将对象与集合数据自动转换为 JSON 数据

```
//使用SpringMVC注解驱动，对标注@ResponseBody注解的控制器方法进行结果转换，由于返回值为引用类型，自动调用jackson提供的类型转换器进行格式转换
@RequestMapping("/showData4")
@ResponseBody
public Book showData4() {
    Book book = new Book();
    book.setName("SpringMVC入门案例");
    book.setPrice(66.66d);
    return book;
}
```

- 手工添加信息类型转换器

```

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
        </list>
    </property>
</bean>

```

- 使用 SpringMVC 注解驱动：

```

<!--开启springmvc注解驱动，对@ResponseBody的注解进行格式增强，追加其类型转换的功能，具体实现由MappingJackson2HttpMessageConverter进行-->
<mvc:annotation-driven/>

```

- 转换集合类型数据

```

@RequestMapping("/showData5")
@ResponseBody
public List showData5() {
    Book book1 = new Book();
    book1.setName("SpringMVC入门案例");
    book1.setPrice(66.66d);

    Book book2 = new Book();
    book2.setName("SpringMVC入门案例");
    book2.setPrice(66.66d);

    ArrayList al = new ArrayList();
    al.add(book1);
    al.add(book2);
    return al;
}

```

Restful

基本介绍

Rest (REpresentational State Transfer)：表现层状态转化，定义了资源在网络传输中以某种表现形式进行状态转移，即网络资源的访问方式

- 资源：把真实的对象数据称为资源，一个资源既可以是一个集合，也可以是单个个体；每一种资源都有特定的 URI（统一资源标识符）与之对应，如果获取这个资源，访问这个 URI 就可以，比如获取特定的班级 /class/12；资源也可以包含子资源，比如 /classes/classId/teachers 某个指定班级的所有老师
- 表现形式：资源是一种信息实体，它可以有多种外在表现形式，把资源具体呈现出来的形式比如 json、xml、image、txt 等等叫做它的“表现层/表现形式”
- 状态转移：描述的服务器端资源的状态，比如增删改查（通过 HTTP 动词实现）引起资源状态的改变，互联网通信协议 HTTP 协议，是一个无状态协议，所有的资源状态都保存在服务器端

访问方式

Restful 是按照 Rest 风格访问网络资源

- 传统风格访问路径：<http://localhost/user/get?id=1>
- Rest 风格访问路径：<http://localhost/user/1>

优点：隐藏资源的访问行为，通过地址无法得知做的是何种操作，书写简化

Restful 请求路径简化配置方式：`@RestController = @Controller + @ResponseBody`

相关注解：`@GetMapping` 注解是 `@RequestMapping` 注解的衍生，所以效果是一样的，建议使用 `@GetMapping`

- `@GetMapping("/poll") = @RequestMapping(value = "/poll", method = RequestMethod.GET)`

```

@RequestMapping(method = RequestMethod.GET)          // @GetMapping 就拥有了 @RequestMapping 的功能
public @interface GetMapping {
    @AliasFor(annotation = RequestMapping.class) // 与 RequestMapping 相通
    String name() default "";
}

```

- `@PostMapping("/push") = @RequestMapping(value = "/push", method = RequestMethod.POST)`

过滤器：`HiddenHttpMethodFilter` 是 SpringMVC 对 Restful 风格的访问支持的过滤器

代码实现：

- restful.jsp：

- 页面表单使用隐藏域提交请求类型，参数名称固定为 `_method`，必须配合提交类型 `method=post` 使用

- GET 请求通过地址栏可以发送，也可以通过设置 form 的请求方式提交
- POST 请求必须通过 form 的请求方式提交

```
<h1>restful风格请求表单</h1>
<!--切换请求路径为restful风格-->
<form action="/user" method="post">
    <!--隐藏域，切换为PUT请求或DELETE请求，但是form表单的提交方式method属性必须填写post-->
    <input name="_method" type="hidden" value="PUT"/>
    <input value="REST-PUT 提交" type="submit"/>
</form>
```

o java / controller / UserController

```
@RestController          //设置rest风格的控制器
@RequestMapping("/user/") //设置公共访问路径，配合下方访问路径使用
public class UserController {
    @GetMapping("/user")
    //@RequestMapping(value = "/user",method = RequestMethod.GET)
    public String getUser(){
        return "GET-张三";
    }

    @PostMapping("/user")
    //@RequestMapping(value = "/user",method = RequestMethod.POST)
    public String saveUser(){
        return "POST-张三";
    }

    @PutMapping("/user")
    //@RequestMapping(value = "/user",method = RequestMethod.PUT)
    public String putUser(){
        return "PUT-张三";
    }

    @DeleteMapping("/user")
    //@RequestMapping(value = "/user",method = RequestMethod.DELETE)
    public String deleteUser(){
        return "DELETE-张三";
    }
}
```

o 配置拦截器 web.xml

```
<!--配置拦截器，解析请求中的参数_method，否则无法发起PUT请求与DELETE请求，配合页面表单使用-->
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <servlet-name>DispatcherServlet</servlet-name>
</filter-mapping>
```

参数注解

Restful 开发中的参数注解

```
@GetMapping("{id}")
public String getMessage(@PathVariable("id") Integer id){}
```

使用 @PathVariable 注解获取路径上配置的具名变量，一般在有多个参数的时候添加

其他注解：

- @RequestHeader: 获取请求头
- @RequestParam: 获取请求参数 (指向号后的参数, url?a=1&b=2)
- @CookieValue: 获取 Cookie 值
- @RequestAttribute: 获取 request 域属性
- @RequestBody: 获取请求体 [POST]
- @MatrixVariable: 矩阵变量
- @ModelAttribute: 自定义类型变量

```
@RestController
@RequestMapping("/user/")
public class UserController {
```

```

//rest风格访问路径简化书写方式，配合类注解@RequestMapping使用
@RequestMapping("{id}")
public String restLocation2(@PathVariable Integer id){
    System.out.println("restful is running ....get:" + id);
    return "success.jsp";
}

//@RequestMapping(value = "{id}",method = RequestMethod.GET)
@GetMapping("{id}")
public String get(@PathVariable Integer id){
    System.out.println("restful is running ....get:" + id);
    return "success.jsp";
}

@PostMapping("{id}")
public String post(@PathVariable Integer id){
    System.out.println("restful is running ....post:" + id);
    return "success.jsp";
}

@PutMapping("{id}")
public String put(@PathVariable Integer id){
    System.out.println("restful is running ....put:" + id);
    return "success.jsp";
}

@DeleteMapping("{id}")
public String delete(@PathVariable Integer id){
    System.out.println("restful is running ....delete:" + id);
    return "success.jsp";
}
}

```

识别原理

表单提交要使用 REST 时，会带上 `_method=PUT`，请求过来被 `HiddenHttpMethodFilter` 拦截，进行过滤操作

`org.springframework.web.filter.HiddenHttpMethodFilter.doFilterInternal()`:

```

public class HiddenHttpMethodFilter extends OncePerRequestFilter {
    // 兼容的请求 PUT、DELETE、PATCH
    private static final List<String> ALLOWED_METHODS =
        Collections.unmodifiableList(Arrays.asList( HttpMethod.PUT.name(),
            HttpMethod.DELETE.name(), HttpMethod.PATCH.name()));
    // 隐藏域的名字
    public static final String DEFAULT_METHOD_PARAM = "_method";

    private String methodParam = DEFAULT_METHOD_PARAM;
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        HttpServletRequest requestToUse = request;
        // 请求必须是 POST,
        if ("POST".equals(request.getMethod()) && request.getAttribute(webutils.ERROR_EXCEPTION_ATTRIBUTE) == null) {
            // 获取标签中 name=_method 的 value 值
            String paramValue = request.getParameter(this.methodParam);
            if (StringUtil.hasLength(paramValue)) {
                // 转成大写
                String method = paramValue.toUpperCase(Locale.ENGLISH);
                // 兼容的请求方式
                if (ALLOWED_METHODS.contains(method)) {
                    // 包装请求
                    requestToUse = new HttpMethodRequestWrapper(request, method);
                }
            }
        }
        // 过滤器链放行的时候用wrapper。以后的方法调用getMethod是调用requestWrapper的
        filterChain.doFilter(requestToUse, response);
    }
}

```

Rest 使用客户端工具，如 Postman 可直接发送 put、delete 等方式请求不被过滤

改变默认的 `_method` 的方式：

```

@Configuration(proxyBeanMethods = false)
public class webConfig{
    //自定义filter
    @Bean
    public HiddenHttpMethodFilter hiddenHttpMethodFilter(){
        HiddenHttpMethodFilter methodFilter = new HiddenHttpMethodFilter();
        //通过set 方法自定义
        methodFilter.setMethodParam("_m");
        return methodFilter;
    }
}

```

Servlet

SpringMVC 提供访问原始 Servlet 接口的功能

- SpringMVC 提供访问原始 Servlet 接口 API 的功能，通过形参声明即可

```

@RequestMapping("/servletApi")
public String servletApi(HttpServletRequest request,
                         HttpServletResponse response, HttpSession session){
    System.out.println(request);
    System.out.println(response);
    System.out.println(session);
    request.setAttribute("name", "seazean");
    System.out.println(request.getAttribute("name"));
    return "page.jsp";
}

```

- Head 数据获取快捷操作方式

名称: @RequestHeader

类型: 形参注解

位置: 处理器类中的方法形参前方

作用: 绑定请求头数据与对应处理方法形参间的关系

范例:

```

快捷操作方式@RequestMapping("/headApi")
public String headApi(@RequestHeader("Accept-Language") String headMsg){
    System.out.println(headMsg);
    return "page";
}

```

- Cookie 数据获取快捷操作方式

名称: @CookieValue

类型: 形参注解

位置: 处理器类中的方法形参前方

作用: 绑定请求 Cookie 数据与对应处理方法形参间的关系

范例:

```

@RequestMapping("/cookieApi")
public String cookieApi(@CookieValue("JSESSIONID") String jsessionid){
    System.out.println(jsessionid);
    return "page";
}

```

- Session 数据获取

名称: @SessionAttribute

类型: 形参注解

位置: 处理器类中的方法形参前方

作用: 绑定请求Session数据与对应处理方法形参间的关系

范例:

```

@RequestMapping("/sessionApi")
public String sessionApi(@SessionAttribute("name") String name){
    System.out.println(name);
    return "page.jsp";
}

//用于在session中放入数据
@RequestMapping("/setSessionData")
public String setSessionData(HttpSession session){
    session.setAttribute("name", "seazean");
    return "page";
}

```

- Session 数据设置
名称: @SessionAttributes
类型: 类注解
位置: 处理器类上方
作用: 声明放入session范围的变量名称, 适用于Model类型数据传参
范例:

```

@Controller
//设定当前类中名称为age和gender的变量放入session范围, 不常用
@SessionAttributes(names = {"age", "gender"})
public class ServletController {
    //将数据放入session存储范围, Model对象实现数据set, @SessionAttributes注解实现范围设定
    @RequestMapping("/setSessionData2")
    public String setSessionDate2(Model model) {
        model.addAttribute("age", 39);
        model.addAttribute("gender", "男");
        return "page";
    }

    @RequestMapping("/sessionApi")
    public String sessionApi(@SessionAttribute("age") int age,
                           @SessionAttribute("gender") String gender){
        System.out.println(name);
        System.out.println(age);
        return "page";
    }
}

```

- spring-mvc.xml 配置

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd
                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.seazean"/>
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/page/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
    <mvc:annotation-driven/>
</beans>

```

运行原理

技术架构

组件介绍

核心组件:

- DispatcherServlet: 核心控制器, 是 SpringMVC 的核心, 整体流程控制的中心, 所有的请求第一步都先到达这里, 由其调用其它组件处理用户的请求, 它就是在 web.xml 配置的核心 Servlet, 有效的降低了组件间的耦合性
- HandlerMapping: 处理器映射器, 负责根据请求找到对应具体的 Handler 处理器, SpringMVC 中针对配置文件方式、注解方式等提供了不同的映射器来处理
- Handler: 处理器, 其实就是 Controller, 业务处理的核心类, 通常由开发者编写, 并且必须遵守 Controller 开发的规则, 这样适配器才能正确的执行。例如实现 Controller 接口, 将 Controller 注册到 IOC 容器中等
- HandlerAdapter: 处理器适配器, 根据映射器中找到的 Handler, 通过 HandlerAdapter 去执行 Handler, 这是适配器模式的应用
- View Resolver: 视图解析器, 将 Handler 中返回的逻辑视图 (ModelAndView) 解析为一个具体的视图 (View) 对象
- View: 视图, View 最后对页面进行渲染将结果返回给用户, SpringMVC 框架提供了很多的 View 视图类型, 包括: jstlView、freemarkerView、pdfView 等



优点:

- 与 Spring 集成，更好的管理资源
- 有很多参数解析器和视图解析器，支持的数据类型丰富
- 将映射器、处理器、视图解析器进行解耦，分工明确

工作原理

在 Spring 容器初始化时会建立所有的 URL 和 Controller 的对应关系，保存到 Map<URL, Controller> 中，这样 request 就能快速根据 URL 定位到 Controller：

- 在 Spring IOC 容器初始化完所有单例 bean 后
- SpringMVC 会遍历所有的 bean，获取 Controller 中对应的 URL (这里获取 URL 的实现类有多个，用于处理不同形式配置的 Controller)
- 将每一个 URL 对应一个 Controller 存入 Map<URL, Controller> 中

注意：将 @Controller 注解换成 @Component，启动时不会报错，但是在浏览器中输入路径时会出现 404，说明 Spring 没有对所有的 bean 进行 URL 映射

一个 Request 来了：

- 监听端口，获得请求：Tomcat 监听 8080 端口的请求处理，根据路径调用了 web.xml 中配置的核心控制器 DispatcherServlet，`DispatcherServlet#doDispatch` 是核心调度方法
- 首先根据 URI 获取 HandlerMapping 处理器映射器，RequestMappingHandlerMapping 用来处理 @RequestMapping 注解的映射规则，其中保存了所有 handler 的映射规则，最后包装成一个拦截器链返回，拦截器链对象持有 HandlerMapping。如果没有合适的处理请求的 HandlerMapping，说明请求处理失败，设置响应码 404 返回
- 根据映射器获取当前 handler，处理器适配器执行处理方法，适配器根据请求的 URL 去 handler 中寻找对应的处理方法：
 - 创建 ModelAndView (mav) 对象，用来填充数据，然后通过不同的参数解析器去解析 URL 中的参数，完成数据解析绑定，然后执行真正的 Controller 方法，完成 handle 处理
 - 方法执行完对返回值进行处理，没添加 @ResponseBody 注解的返回值使用视图处理器处理，把视图名称设置进入 mav 中
 - 对添加了 @ResponseBody 注解的 Controller 的按照普通的返回值进行处理，首先进行内容协商，找到一种浏览器可以接受（请求头 Accept）的并且服务器可以生成的数据类型，选择合适数据转换器，设置响应头中的数据类型，然后写出数据
 - 最后把 ModelAndView 和 ModelMap 中的数据封装到 ModelAndView 对象返回
- 视图解析，根据返回值创建视图，请求转发 View 实例为 InternalResourceView，重定向 View 实例为 RedirectView。最后调用 view.render 进行页面渲染，结果派发
 - 请求转发时请求域中的数据不丢失，会把 ModelAndView 的数据设置到请求域中，获取 Servlet 原生的 RequestDispatcher，调用 RequestDispatcher#forward 实现转发
 - 重定向会造成请求域中的数据丢失，使用 Servlet 原生方式实现重定向 HttpServletResponse#sendRedirect

调度函数

请求进入原生的 HttpServlet 的 doGet() 方法处理，调用子类 FrameworkServlet 的 doGet() 方法，最终调用 DispatcherServlet 的 doService() 方法，为请求设置相关属性后调用 doDispatch()，请求和响应的以参数的形式传入



```

// request 和 response 为 Java 原生的类
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    // 文件上传请求
    boolean multipartRequestParsed = false;
    // 异步管理器
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ...
    }
}

```

```

ModelAndView mv = null;
Exception dispatchException = null;

try {
    // 文件上传相关请求
    processedRequest = checkMultipart(request);
    multipartRequestParsed = (processedRequest != request);

    // 找到当前请求使用哪个 HandlerMapping (Controller 的方法) 处理, 返回执行链
    mappedHandler = getHandler(processedRequest);
    // 没有合适的处理请求的方式 HandlerMapping, 请求失败, 直接返回 404
    if (mappedHandler == null) {
        noHandlerFound(processedRequest, response);
        return;
    }

    // 根据映射器获取当前 handler 处理器适配器, 用来【处理当前的请求】
    HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
    // 获取发出此次请求的方式
    String method = request.getMethod();
    // 判断请求是不是 GET 方法
    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;
        }
    }
    // 拦截器链的前置处理
    if (!mappedHandler.applyPreHandle(processedRequest, response)) {
        return;
    }
    // 执行处理方法, 返回的是 ModelAndView 对象, 封装了所有的返回值数据
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

    if (asyncManager.isConcurrentHandlingStarted()) {
        return;
    }
    // 设置视图名字
    applyDefaultViewName(processedRequest, mv);
    // 执行拦截器链中的后置处理方法
    mappedHandler.applyPostHandle(processedRequest, response, mv);
} catch (Exception ex) {
    dispatchException = ex;
}

// 处理程序调用的结果, 进行结果派发
processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
}
//...
}

```

笔记参考视频: <https://www.bilibili.com/video/BV19K4y1L7MT>

请求映射

映射器

doDispatch() 中调用 getHandler 方法获取所有的映射器

总体流程:

- 所有的请求映射都在 HandlerMapping 中, **RequestMappingHandlerMapping** 处理 @RequestMapping 注解的映射规则
- 遍历所有的 HandlerMapping 看是否可以匹配当前请求, 匹配成功后返回, 匹配失败设置 HTTP 404 响应码
- 用户可以自定义的映射处理, 也可以给容器中放入自定义 HandlerMapping

访问 URL: <http://localhost:8080/user>

```

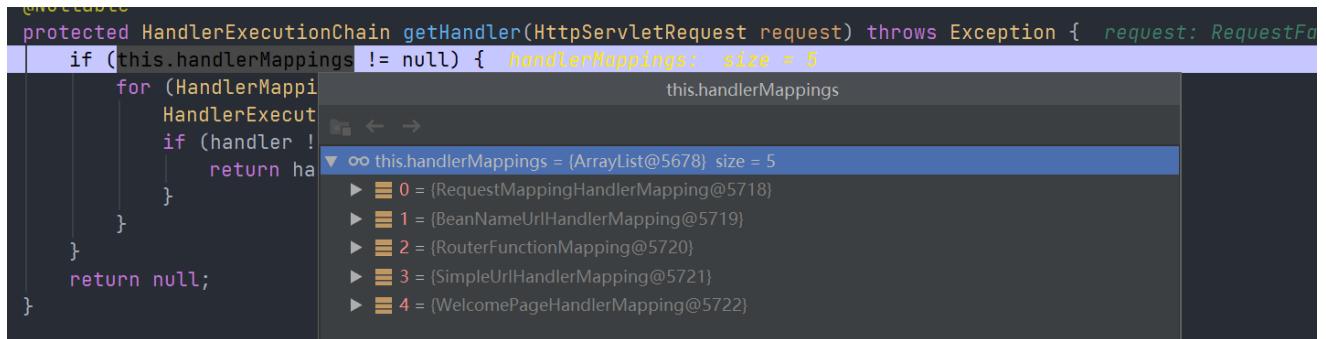
@GetMapping("/user")
public String getUser(){
    return "GET";
}
@PostMapping("/user")
public String postUser(){
    return "POST";
}
// ...

```

HandlerMapping 处理器映射器，保存了所有 @RequestMapping 和 handler 的映射规则

```
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    if (this.handlerMappings != null) {
        // 遍历所有的 HandlerMapping
        for (HandlerMapping mapping : this.handlerMappings) {
            // 尝试去每个 HandlerMapping 中匹配当前请求的处理
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}
```

```
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception { request: RequestFacade
    if (this.handlerMappings != null) { handlerMappings: size = 5
        for (HandlerMapping mapping : this.handlerMappings) {
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}
```



- `mapping.getHandler(request)`：调用 AbstractHandlerMapping#getHandler
 - `Object handler = getHandlerInternal(request)`：获取映射器，底层调用 RequestMappingInfoHandlerMapping 类的方法，又调用 AbstractHandlerMethodMapping#getHandlerInternal
 - `String lookupPath = initLookupPath(request)`：地址栏的 URI，这里的 lookupPath 为 /user
 - `this.mappingRegistry.acquireReadLock()`：加读锁防止其他线程并发修改
 - `handlerMethod = lookupHandlerMethod(lookupPath, request)`：获取当前 HandlerMapping 中的映射规则
 - `directPathMatches = this.mappingRegistry.getMappingsByDirectPath(lookupPath)`：获取当前的映射器与当前请求的 URI 有关的所有映射规则
 - `addMatchingMappings(directPathMatches, matches, request)`：匹配某个映射规则
 - `for (T mapping : mappings)`：遍历所有的映射规则
 - `match = getMatchingMapping(mapping, request)`：去匹配每一个映射规则，匹配失败返回 null
 - `matches.add(new Match())`：匹配成功后封装成匹配器添加到匹配集合中
 - `matches.sort(comparator)`：匹配集合排序
 - `Match bestMatch = matches.get(0)`：匹配完成只剩一个，直接获取返回对应的处理方法
 - `if (matches.size() > 1)`：当有多个映射规则符合请求时，报错
 - `return bestMatch.getHandlerMethod()`：返回匹配器中的处理方法
 - `executionChain = getHandlerExecutionChain(handler, request)`：为当前请求和映射器的构建一个拦截器链
 - `for (HandlerInterceptor interceptor : this.adaptedInterceptors)`：遍历所有的拦截器
 - `chain.addInterceptor(interceptor)`：把所有的拦截器添加到 HandlerExecutionChain 中，形成拦截器链
 - `return executionChain`：返回拦截器链，HandlerMapping 是链的 handler 成员属性

适配器

doDispatch() 中调用 `HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler())`

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    if (this.handlerAdapters != null) {
        // 遍历所有的 HandlerAdapter
        for (HandlerAdapter adapter : this.handlerAdapters) {
            // 判断当前适配器是否支持当前 handle
            if (adapter.supports(handler)) {
                return adapter;
            }
        }
    }
    return null;
}
```

```

        if (adapter.supports(handler)) {
            // 返回的是【RequestMappingHandlerAdapter】
            // AbstractHandlerMethodAdapter#supports -> RequestMappingHandlerAdapter
            return adapter;
        }
    }
    throw new ServletException();
}

```

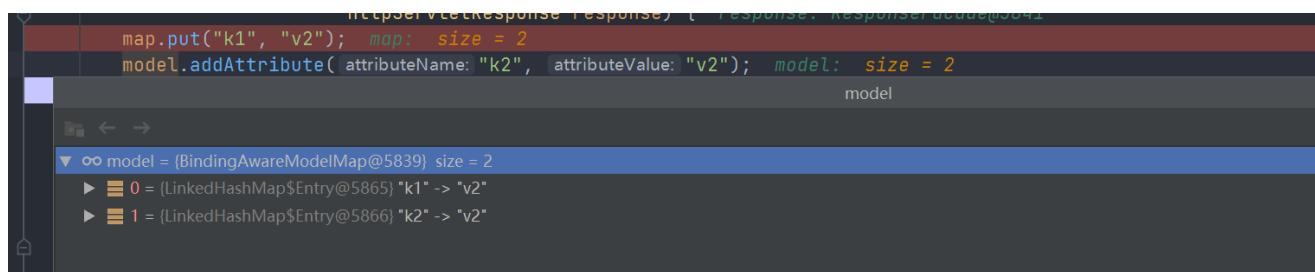
方法执行

实例代码：

```

@GetMapping("/params")
public String param(Map<String, Object> map, Model model, HttpServletRequest request) {
    map.put("k1", "v1");           // 都可以向请求域中添加数据
    model.addAttribute("k2", "v2"); // 它们两个都在数据封装在【BindingAwareModelMap】，继承自 LinkedHashMap
    request.setAttribute("m", "HelloWorld");
    return "forward:/success";
}

```



doDispatch() 中调用 `mv = ha.handle(processedRequest, response, mappedHandler.getHandler())` 使用适配器执行方法

`AbstractHandlerMethodAdapter#handle` → `RequestMappingHandlerAdapter#handleInternal` → `invokeHandlerMethod` :

```

protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
                                         HttpServletResponse response,
                                         HandlerMethod handlerMethod) throws Exception {
    // 封装成 SpringMVC 的接口，用于通用 web 请求拦截器，使能够访问通用请求元数据，而不是用于实际处理请求
    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        // WebDataBinder 用于【从 web 请求参数到 JavaBean 对象的数据绑定】，获取创建该实例的工厂
        WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
        // 创建 Model 实例，用于向模型添加属性
        ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);
        // 方法执行器
        ServletInvocableHandlerMethod invocableMethod = createInvocableHandlerMethod(handlerMethod);

        // 参数解析器，有很多
        if (this.argumentResolvers != null) {
            invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        }
        // 返回值处理器，也有很多
        if (this.returnValueHandlers != null) {
            invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
        }
        // 设置数据绑定器
        invocableMethod.setDataBinderFactory(binderFactory);
        // 设置参数检查器
        invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);

        // 新建一个 ModelAndViewContainer 并进行初始化和一些属性的填充
        ModelAndViewContainer mavContainer = new ModelAndViewContainer();

        // 设置一些属性

        // 【执行目标方法】
        invocableMethod.invokeAndHandle(webRequest, mavContainer);
        // 异步请求
        if (asyncManager.isConcurrentHandlingStarted()) {
            return null;
        }
        // 【获取 ModelAndView 对象，封装了 ModelAndViewContainer】
        return get ModelAndView(mavContainer, modelFactory, webRequest);
    }
    finally {

```

```
        webRequest.requestCompleted();
    }
}
```

ServletInvocableHandlerMethod#invokeAndHandle: 执行目标方法

- returnValue = invokeForRequest(webRequest, mavContainer, providedArgs) : 执行自己写的 controller 方法, 返回的就是自定义方法中 return 的值
- Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs) : 参数处理的逻辑, 遍历所有的参数解析器解析参数或者将 URI 中的参数进行绑定, 绑定完成后开始执行目标方法
 - parameters = getMethodParameters() : 获取此处理程序方法的方法参数的详细信息
 - Object[] args = new Object[parameters.length] : 存放所有的参数
 - for (int i = 0; i < parameters.length; i++) : 遍历所有的参数
 - args[i] = findProvidedArgument(parameter, providedArgs) : 获取调用方法时提供的参数, 一般是空
 - if (!this.resolvers.supportsParameter(parameter)) : 获取可以解析当前参数的参数解析器
 - return getArgumentResolver(parameter) != null : 获取参数的解析器是否为空
 - for (HandlerMethodArgumentResolver resolver : this.argumentResolvers) : 遍历容器内所有的解析器
 - if (resolver.supportsParameter(parameter)) : 是否支持当前参数
 - PathVariableMethodArgumentResolver#supportsParameter : 解析标注 @PathVariable 注解的参数
 - ModelMethodProcessor#supportsParameter : 解析 Map 和 Model 类型的参数, Model 和 Map 的作用一样
 - ExpressionValueMethodArgumentResolver#supportsParameter : 解析标注 @Value 注解的参数
 - RequestParamMapMethodArgumentResolver#supportsParameter : 解析标注 @RequestParam 注解
 - RequestPartMethodArgumentResolver#supportsParameter : 解析文件上传的信息
 - ModelAttributeMethodProcessor#supportsParameter : 解析标注 @ModelAttribute 注解或者不是简单类型
 - 子类 ServletModelAttributeMethodProcessor 是解析自定义类型 JavaBean 的解析器
 - 简单类型有 Void、Enum、Number、CharSequence、Date、URI、URL、Locale、Class
 - args[i] = this.resolvers.resolveArgument() : 开始解析参数, 每个参数使用的解析器不同
 - resolver = getArgumentResolver(parameter) : 获取参数解析器
 - return resolver.resolveArgument() : 开始解析
 - PathVariableMapMethodArgumentResolver#resolveArgument : @PathVariable, 包装 URI 中的参数为 Map
 - MapMethodProcessor#resolveArgument : 调用 mavContainer.getModel() 返回默认 BindingAwareModelMap 对象
 - ModelAttributeMethodProcessor#resolveArgument : 自定义的 JavaBean 的绑定封装, 下一小节详解
- return doInvoke(args) : 真正的执行 Controller 方法
 - Method method = getBridgedMethod() : 从 HandlerMethod 获取要反射执行的方法
 - ReflectionUtils.makeAccessible(method) : 破解权限
 - method.invoke(getBean(), args) : 执行方法, getBean 获取的是标记 @Controller 的 Bean 类, 其中包含执行方法

◦ 进行返回值的处理, 响应部分详解, 处理完成进入下面的逻辑

RequestMappingHandlerAdapter#getModelAndView: 获取 ModelAndView 对象

- modelFactory.updateModel(webRequest, mavContainer) : Model 数据升级到会话域 (请求域中的数据在重定向时丢失)
 - updateBindingResult(request, defaultModel) : 把绑定的数据添加到 BindingAwareModelMap 中
- if (mavContainer.isRequestHandled()) : 判断请求是否已经处理完成了
- ModelAndView model = mavContainer.getModel() : 获取包含 Controller 方法参数的 BindingAwareModelMap (本节开头)
- mav = new ModelAndView() : 把 ModelAndViewContainer 和 ModelAndView 中的数据封装到 ModelAndView
- if (!mavContainer.isViewReference()) : 是否是通过名称指定视图引用
- if (model instanceof RedirectAttributes) : 判断 model 是否是重定向数据, 如果是进行重定向逻辑
- return mav : 任何方法执行都会返回 ModelAndView 对象

参数解析

解析自定义的 JavaBean 为例, 调用 ModelAttributeMethodProcessor#resolveArgument 处理参数的方法, 通过合适的类型转换器把 URL 中的参数转换以后, 利用反射获取 set 方法, 注入到 JavaBean

- Person.java:

```
@Data
@Component //加入到容器中
public class Person {
    private String userName;
    private Integer age;
    private Date birth;
}
```

- Controller:

```

@RestController //返回的数据不是页面
public class ParameterController {
    // 数据绑定：页面提交的请求数据（GET、POST）都可以和对象属性进行绑定
    @GetMapping("/saveuser")
    public Person saveuser(Person person){
        return person;
    }
}

```

- 访问 URL: <http://localhost:8080/saveuser?userName=zhangsan&age=20>

进入源码: ModelAttributeMethodProcessor#resolveArgument

- name = ModelFactory.getNameForParameter(parameter) : 获取名字, 此例就是 person
- ann = parameter.getParameterAnnotation(ModelAttribute.class) : 是否有 ModelAttribute 注解
- if (mavContainer.containsAttribute(name)) : ModelAndView 中是否包含 person 对象
- attribute = createAttribute() : 创建一个实例, 空的 Person 对象
- binder = binderFactory.createBinder(webRequest, attribute, name) : Web 数据绑定器, 可以利用 Converters 将请求数据转成指定的数据类型, 绑定到 JavaBean 中
- bindRequestParameters(binder, webRequest) : 利用反射向目标对象填充数据
 - servletBinder = (ServletRequestDataBinder) binder: 类型强转
 - servletBinder.bind(servletRequest) : 绑定数据
 - mpvs = new MutablePropertyValues(request.getParameterMap()) : 获取请求 URI 参数中的 k-v 键值对
 - addBindValues(mpvs, request) : 子类可以用来为请求添加额外绑定值
 - doBind(mpvs) : 真正的绑定的方法, 调用 applyPropertyValues 应用参数值, 然后调用 setPropertyValues 方法
 - AbstractPropertyAccessor#setPropertyValues() :
 - List<PropertyValue> propertyValues : 获取到所有的参数的值, 就是 URI 上的所有的参数值
 - for (PropertyValue pv : propertyValues) : 遍历所有的参数值
 - set PropertyValue(pv) : 填充到空的 Person 实例中
 - nestedPa = getPropertyAccessorForPropertyName(propertyName) : 获取属性访问器
 - tokens = getPropertyNameTokens() : 获取元数据的信息
 - nestedPa.setPropertyValue(tokens, pv) : 填充数据
 - processLocalProperty(tokens, pv) : 处理属性
 - if (!Boolean.FALSE.equals(pv.conversionNecessary)) : 数据是否需要转换了
 - if (pv.isConverted()) : 数据已经转换过了, 转换了直接赋值, 没转换进行转换
 - oldValue = ph.getValue() : 获取未转换的数据
 - valueToApply = convertForProperty() : 进行数据转换
 - TypeConverterDelegate#convertIfNecessary : 进入该方法的逻辑
 - if (conversionService.canConvert(sourceTypeDesc, typeDescriptor)) : 判断能不能转换
 - GenericConverter converter = getConverter(sourceType, targetType) : 获取类型转换器
 - converter = this.converters.find(sourceType, targetType) : 寻找合适的转换器
 - sourceCandidates = getClassHierarchy(sourceType.getType()) : 原数据类型
 - targetCandidates = getClassHierarchy(targetType.getType()) : 目标数据类型
 - for (Class<?> sourceCandidate : sourceCandidates) {

//双重循环遍历, 寻找合适的转换器
 ■ for (Class<?> targetCandidate : targetCandidates) {
 - GenericConverter converter = getRegisteredConverter(..) : 匹配类型转换器
 - return converter : 返回转换器
 - conversionService.convert(newValue, sourceTypeDesc, typeDescriptor) : 开始转换
 - converter = getConverter(sourceType, targetType) : 获取可用的转换器
 - result = ConversionUtils.invokeConverter() : 执行转换方法
 - converter.convert() : 调用转换器的转换方法 (GenericConverter#convert)
 - return handleResult(sourceType, targetType, result) : 返回结果
 - ph.setValue(valueToApply) : 设置 JavaBean 属性 (BeanWrapperImpl.BeanPropertyHandler)
 - Method writeMethod : 获取写数据方法
 - Class<?> cls = getClass0() : 获取 Class 对象
 - writeMethodName = Introspector.SET_PREFIX + getBaseName() : set 前缀 + 属性名
 - writeMethod = Introspector.findMethod(cls, writeMethodName, 1, args) : 获取只包含一个参数的 set 方法
 - setWriteMethod(writeMethod) : 加入缓存
 - ReflectionUtils.makeAccessible(writeMethod) : 设置访问权限
 - writeMethod.invoke(getWrappedInstance(), value) : 执行方法

- `bindingResult = binder.getBindingResult()`：获取绑定的结果
- `mavContainer.addAllAttributes(bindingResultModel)`：把所有填充的参数放入 `ModelAndViewContainer`
- `return attribute`：返回填充后的 Person 对象

响应处理

响应数据

以 Person 为例：

```
@ResponseBody      // 利用返回值处理器里面的消息转换器进行处理，而不是视图
@GetMapping(value = "/person")
public Person getPerson() {
    Person person = new Person();
    person.setAge(28);
    person.setBirth(new Date());
    person.setUserName("zhangsan");
    return person;
}
```

直接进入方法执行完后的逻辑 `ServletInvocableHandlerMethod#invokeAndHandle`：

```
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
                             Object... providedArgs) throws Exception {
    // 【执行目标方法】，return person 对象
    Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
    // 设置状态码
    setResponseStatus(webRequest);

    // 判断方法是否有返回值
    if (returnValue == null) {
        if (isRequestNotModified(webRequest) || getResponseStatus() != null || mavContainer.isRequestHandled()) {
            disableContentCachingIfNecessary(webRequest);
            mavContainer.setRequestHandled(true);
            return;
        }
    } // 返回值是字符串
    else if (Stringutils.hasText(getResponseStatusReason())) {
        // 设置请求处理完成
        mavContainer.setRequestHandled(true);
        return;
    } // 设置请求没有处理完成，还需要进行返回值的逻辑
    mavContainer.setRequestHandled(false);
    Assert.state(this.returnValueHandlers != null, "No return value handlers");
    try {
        // 【返回值的处理】
        this.returnValueHandlers.handleReturnValue(
            returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
    }
    catch (Exception ex) {}
}
```

- 没有加 `@ResponseBody` 注解的返回数据按照视图处理的逻辑，`ViewNameMethodReturnValueHandler`（视图详解）
- 此例是加了注解的，返回的数据不是视图，`HandlerMethodReturnValueHandlerComposite#handleReturnValue`：

```
public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
                             ModelAndViewContainer mavContainer, NativeWebRequest webRequest) {
    // 获取合适的返回值处理器
    HandlerMethodReturnValueHandler handler = selectHandler(returnValue, returnType);
    if (handler == null) {
        throw new IllegalArgumentException();
    }
    // 使用处理器处理返回值（详解源码中的这两个函数）
    handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);
}
```

`HandlerMethodReturnValueHandlerComposite#selectHandler`：获取合适的返回值处理器

- `boolean isAsyncValue = isAsyncReturnValue(value, returnType)`：是否是异步请求
- `for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers)`：遍历所有的返回值处理器
 - `RequestBodyMethodProcessor#supportsReturnType`：处理标注 `@ResponseBody` 注解的返回值
 - `ModelAndViewMethodReturnValueHandler#supportsReturnType`：处理返回值类型是 `ModelAndView` 的处理器
 - `ModelAndViewResolverMethodReturnValueHandler#supportsReturnType`：直接返回 true，处理所有数据

RequestResponseBodyMethodProcessor#handleReturnValue: 处理返回值, 要进行内容协商

- `mavContainer.setRequestHandled(true)`: 设置请求处理完成
- `inputMessage = createInputMessage(webRequest)`: 获取输入的数据
- `outputMessage = createOutputMessage(webRequest)`: 获取输出的数据
- `writeWithMessageConverters(returnValue, returnType, inputMessage, outputMessage)`: 使用消息转换器进行写出
 - `if (value instanceof CharSequence)`: 判断返回的数据是不是字符类型
 - `body = value`: 把 value 赋值给 body, 此时 body 中就是自定义方法执行完后的 Person 对象
 - `if (isResourceType(value, returnType))`: 当前数据是不是流数据
 - `MediaType selectedMediaType`: 内容协商后选择使用的类型, 浏览器和服务器都支持的媒体(数据)类型
 - `MediaType contentType = outputMessage.getHeaders().getContentType()`: 获取响应头的数据
 - `if (contentType != null && contentType.isConcrete())`: 判断当前响应头中是否已经有确定的媒体类型
`selectedMediaType = contentType`: 前置处理已经使用了媒体类型, 直接继续使用该类型
 - `acceptableTypes = getAcceptableMediaTypes(request)`: 获取浏览器支持的媒体类型, 请求头字段
 - `this.contentNegotiationManager.resolveMediaTypes()`: 调用该方法
 - `for(ContentNegotiationStrategy strategy: this.strategies)`: 默认策略是提取请求头的字段的内容, 策略类为 HeaderContentNegotiationStrategy, 可以配置添加其他类型的策略
 - `List<MediaType> mediaTypes = strategy.resolveMediaTypes(request)`: 解析 Accept 字段存储为 List
 - `headerValueArray = request.getHeaderValues(HttpHeaders.ACCEPT)`: 获取请求头中 Accept 字段
 - `List<MediaType> mediaTypes = MediaType.parseMediaTypes(headerValues)`: 解析成 List 集合
 - `MediaType.sortBySpecificityAndQuality(mediaTypes)`: 按照相对品质因数 q 降序排序

▼ Request Headers [View source](#)

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en-GB;q=0.8,en;q=0.7
```

- `producibleTypes = getProducibleMediaTypes(request, valueType, targetType)`: 服务器能生成的媒体类型
 - `request.getAttribute(HandlerMapping.PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE)`: 从请求域获取默认的媒体类型
 - `for (HttpMessageConverter<?> converter : this.messageConverters)`: 遍历所有的消息转换器
 - `converter.canWrite(valueClass, null)`: 是否支持当前的类型
 - `result.addAll(converter.getSupportedMediaTypes())`: 把当前 MessageConverter 支持的所有类型放入 result
- `List<MediaType> mediaTypesToUse = new ArrayList<>()`: 存储最佳匹配的集合
- 内容协商:

```
for (MediaType requestedType : acceptableTypes) { // 遍历所有浏览器能接受的媒体类型
    for (MediaType producibleType : producibleTypes) { // 遍历所有服务器能产出的
        if (requestedType.isCompatibleWith(producibleType)) { // 判断类型是否匹配, 最佳匹配
            // 数据协商匹配成功, 一般有多种
            mediaTypesToUse.add(getMostSpecificMediaType(requestedType, producibleType));
        }
    }
}
```

- `MediaType.sortBySpecificityAndQuality(mediaTypesToUse)`: 按照相对品质因数 q 排序, 降序排序, 越大的越好
- `for (MediaType mediaType : mediaTypesToUse)`: 遍历所有的最佳匹配, 选择一种赋值给选择的类型
- `selectedMediaType = selectedMediaType.removeQualityValue()`: 媒体类型去除相对品质因数
- `for (HttpMessageConverter<?> converter : this.messageConverters)`: 遍历所有的 HTTP 数据转换器
- `GenericHttpMessageConverter genericConverter: MappingJackson2HttpMessageConverter 可以将对象写为 JSON`
- `((GenericHttpMessageConverter) converter).canWrite()`: 判断转换器是否可以写出给定的类型
- `AbstractJackson2HttpMessageConverter#canWrite`
 - `if (!canWrite(mediaType))`: 是否可以写出指定类型
 - `MediaType.ALL.equalsTypeAndSubtype(mediaType)`: 是不是 */* 类型
 - `getSupportedMediaTypes()`: 支持 application/json 和 application/*+json 两种类型
 - `return true`: 返回 true
 - `objectMapper = selectObjectMapper(clazz, mediaType)`: 选择可以使用的 objectMapper
 - `causeRef = new AtomicReference<>()`: 获取并发安全的引用
 - `if (objectMapper.canSerialize(clazz, causeRef))`: objectMapper 可以序列化当前类
 - `return true`: 返回 true
- `body = getAdvice().beforeBodyWrite()`: 获取要响应的所有数据, 就是 Person 对象
- `addContentDispositionHeader(inputMessage, outputMessage)`: 检查路径
- `genericConverter.write(body, targetType, selectedMediaType, outputMessage)`: 调用消息转换器的 write 方法
- `AbstractGenericHttpMessageConverter#write`: 该类的方法

- `addDefaultHeaders(headers, t, contentType)`: 设置响应头中的数据类型
 - ▼ Response Headers View source
 - `Connection: keep-alive`
 - `Content-Type: application/json`
- `writeInternal(t, type, outputMessage)`: 数据写出为 JSON 格式
 - `Object value = object`: value 引用 Person 对象
 - `ObjectWriter objectWriter = objectMapper.writer()`: 获取 ObjectWriter 对象
 - `objectWriter.writeValue(generator, value)`: 使用 ObjectWriter 写出数据为 JSON

协商策略

开启基于请求参数的内容协商模式: (SpringBoot 方式)

```
spring.mvc.contentnegotiation:favor-parameter: true # 开启请求参数内容协商模式
```

发请求: <http://localhost:8080/person?format=json>, 解析 format

策略类为 ParameterContentNegotiationStrategy, 运行流程如下:

- `acceptableTypes = getAcceptableMediaTypes(request)`: 获取浏览器支持的媒体类型
- `mediaTypes = strategy.resolveMediaTypes(request)`: 解析请求 URL 参数中的数据
- `return resolveMediaTypeKey(webRequest, getMediaTypeKey(webRequest))`:
 - `getMediaTypeKey(webRequest)`:
 - `request.getParameter(getParameterName())`: 获取 URL 中指定的需求的数据类型
 - `getParameterName()`: 获取参数的属性名 format
 - `getParameter()`: 获取 URL 中 format 对应的数据
- `resolveMediaTypeKey()`: 解析媒体类型, 封装成集合

自定义内容协商策略:

```
public class webConfig implements WebMvcConfigurer {  
    @Bean  
    public WebMvcConfigurer webMvcConfigurer() {  
        return new WebMvcConfigurer() {  
            @Override // 自定义内容协商策略  
            public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {  
                Map<String, MediaType> mediaTypes = new HashMap<>();  
                mediaTypes.put("json", MediaType.APPLICATION_JSON);  
                mediaTypes.put("xml", MediaType.APPLICATION_XML);  
                mediaTypes.put("person", MediaType.parseMediaType("application/x-person"));  
                // 指定支持解析哪些参数对应的哪些媒体类型  
                ParameterContentNegotiationStrategy parameterStrategy = new ParameterContentNegotiationStrategy(mediaTypes);  
  
                // 请求头解析  
                HeaderContentNegotiationStrategy headStrategy = new HeaderContentNegotiationStrategy();  
  
                // 添加到容器中, 即可以解析请求头 又可以解析请求参数  
                configurer.strategies(Arrays.asList(parameterStrategy, headStrategy));  
            }  
  
            @Override // 自定义消息转换器  
            public void extendMessageConverters(List<HttpMessageConverter<?>> converters) {  
                converters.add(new GuiguMessageConverter());  
            }  
        };  
    }  
}
```

也可以自定义 `HttpMessageConverter`, 实现 `HttpMessageConverter` 接口重写方法即可

视图解析

返回解析

请求处理:

```
@GetMapping("/params")
public String param(){
    return "forward:/success";
//return "redirect:/success";
}
```

进入执行方法逻辑 ServletInvocableHandlerMethod#invokeAndHandle, 进入 this.returnValueHandlers.handleReturnValue :

```
public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
                             ModelAndViewContainer mavContainer, NativeWebRequest webRequest) {
    // 获取合适的返回值处理器: 调用 if (handler.supportsReturnType(returnType)) 判断是否支持
    HandlerMethodReturnValueHandler handler = selectHandler(returnValue, returnType);
    if (handler == null) {
        throw new IllegalArgumentException();
    }
    // 使用处理器处理返回值
    handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);
}
```

- o ViewNameMethodReturnValueHandler#supportsReturnType:

```
public boolean supportsReturnType(MethodParameter returnType) {
    Class<?> paramType = returnType.getParameterType();
    // 返回值是否是 void 或者是 CharSequence 字符序列, 这里是字符序列
    return (void.class == paramType || CharSequence.class.isAssignableFrom(paramType));
}
```

- o ViewNameMethodReturnValueHandler#handleReturnValue:

```
public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
                             ModelAndViewContainer mavContainer,
                             NativeWebRequest webRequest) throws Exception {
    // 返回值是字符串, 是 return "forward:/success"
    if (returnValue instanceof CharSequence) {
        String viewName = returnValue.toString();
        // 【把视图名称设置进入 ModelAndViewContainer 中】
        mavContainer.setViewName(viewName);
        // 判断是否是重定向数据 `viewName.startsWith("redirect:")`
        if (isRedirectViewName(viewName)) {
            // 如果是重定向, 设置是重定向指令
            mavContainer.setRedirectModelScenario(true);
        }
    }
    else if (returnValue != null) {
        // should not happen
        throw new UnsupportedOperationException();
    }
}
```

结果派发

doDispatch() 中的 processDispatchResult: 处理派发结果

```
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
                                    @Nullable HandlerExecutionChain mappedHandler,
                                    @Nullable ModelAndView mv,
                                    @Nullable Exception exception) throws Exception {
    boolean errorview = false;
    if (exception != null) {
    }
    // mv 是 ModelAndView
    if (mv != null && !mv.wasCleared()) {
        // 渲染视图
        render(mv, request, response);
        if (errorview) {
            webUtils.clearErrorRequestAttributes(request);
        }
    }
    else {}
}
```

DispatcherServlet#render:

- Locale locale = this.localeResolver.resolveLocale(request) : 国际化相关
- String viewName = mv.getViewName() : 视图名字, 是请求转发 forward:/success (响应数据解析并存入 ModelAndView)
- view = resolveViewName(viewName, mv.getModelInternal(), locale, request) : 解析视图
 - for (ViewResolver viewResolver : this.viewResolvers) : 遍历所有的视图解析器
 - view = viewResolver.resolveViewName(viewName, locale) : 根据视图名字解析视图, 调用内容协商视图处理器 ContentNegotiatingViewResolver 的方法
 - attrs = RequestContextHolder.getRequestAttributes() : 获取请求的相关属性信息
 - requestedMediaTypes = getMediaTypes(((ServletRequestAttributes) attrs).getRequest()) : 获取最佳匹配的媒体类型, 函数内进行了匹配的逻辑
 - candidateViews = getCandidateViews(viewName, locale, requestedMediaTypes) : 获取候选的视图对象
 - for (ViewResolver viewResolver : this.viewResolvers) : 遍历所有的视图解析器
 - view view = viewResolver.resolveViewName(viewName, locale) : 解析视图
 - AbstractCachingViewResolver#resolveViewName:
 - return view = createView(viewName, locale) : UrlBasedViewResolver#createView
- 请求转发: 实例为 InternalResourceView
 - if (viewName.startsWith(FORWARD_URL_PREFIX)) : 视图名字是否是 forward: 的前缀
 - forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length()) : 名字截取前缀
 - view = new InternalResourceView(forwardUrl) : 新建 InternalResourceView 对象并返回
 - return applyLifecycleMethods(FORWARD_URL_PREFIX, view) : Spring 中的初始化操作
- 重定向: 实例为 RedirectView
 - if (viewName.startsWith(REDIRECT_URL_PREFIX)) : 视图名字是否是 redirect: 的前缀
 - redirectUrl = viewName.substring(REDIRECT_URL_PREFIX.length()) : 名字截取前缀
 - RedirectView view = new RedirectView() : 新建 RedirectView 对象并返回
 - bestView = getBestView(candidateViews, requestedMediaTypes, attrs) : 选出最佳匹配的视图对象
- view.render(mv.getModelInternal(), request, response) : 页面渲染
 - mergedModel = createMergedOutputModel(model, request, response) : 把请求域中的数据封装到 model
 - prepareResponse(request, response) : 响应前的准备工作, 设置一些响应头
 - renderMergedOutputModel(mergedModel, getRequestToExpose(request), response) : 渲染输出的数据
 - getRequestToExpose(request) : 获取 Servlet 原生的方式

请求转发 InternalResourceView 的逻辑: 请求域中的数据不丢失

- exposeModelAsRequestAttributes(model, request) : 暴露 model 作为请求域的属性
 - model.forEach() : 遍历 Model 中的数据
 - request.setAttribute(name, value) : 设置到请求域中
- exposeHelpers(request) : 自定义接口
- dispatcherPath = prepareForRendering(request, response) : 确定调度分派的路径, 此例是 /success
- rd = getRequestDispatcher(request, dispatcherPath) : 获取 Servlet 原生的 RequestDispatcher 实现转发
- rd.forward(request, response) : 实现请求转发

重定向 RedirectView 的逻辑: 请求域中的数据会丢失

- targetUrl = createTargetUrl(model, request) : 获取目标 URL
- enc = request.getCharacterEncoding() : 设置编码 UTF-8
- appendQueryProperties(targetUrl, model, enc) : 添加一些属性, 比如 url + ?name=123&age=324
- sendRedirect(request, response, targetUrl, this.http10Compatible) : 重定向
 - response.sendRedirect(encodedURL) : 使用 Servlet 原生方法实现重定向

异步调用

请求参数

名称: @RequestBody

类型: 形参注解

位置: 处理器类中的方法形参前方

作用: 将异步提交数据转换成标准请求参数格式, 并赋值给形参

范例:

```

@Controller //控制层
public class AjaxController {
    @RequestMapping("/ajaxController")
    public String ajaxController(@RequestBody String message){
        System.out.println(message);
        return "page.jsp";
    }
}

```

- 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
 - POJO 中的属性如果请求数据中没有，属性值为 null
 - POJO 中没有的属性如果请求数据中有，不进行映射
- 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

```

@RequestMapping("/ajaxPojoToController")
//如果处理参数是POJO，且页面发送的请求数据格式与POJO中的属性对应，@RequestBody注解可以自动映射对应请求数据到POJO中
public String ajaxPojoToController(@RequestBody User user){
    System.out.println("controller pojo :" + user);
    return "page.jsp";
}

@RequestMapping("/ajaxListToController")
//如果处理参数是List集合且封装了POJO，且页面发送的数据是JSON格式，数据将自动映射到集合参数
public String ajaxListToController(@RequestBody List<User> userList){
    System.out.println("controller list :" + userList);
    return "page.jsp";
}

```

ajax.jsp

```

<%@page pageEncoding="UTF-8" language="java" contentType="text/html;UTF-8" %>

<a href="javascript:void(0);" id="testAjax">访问springmvc后台controller</a><br/>
<a href="javascript:void(0);" id="testAjaxPojo">传递Json格式POJO</a><br/>
<a href="javascript:void(0);" id="testAjaxList">传递Json格式List</a><br/>

<script type="text/javascript" src="${pageContext.request.contextPath}/js/jquery-3.3.1.min.js"></script>
<script type="text/javascript">
$(function () {
    //为id="testAjax"的组件绑定点击事件
    $("#testAjax").click(function(){
        //发送异步调用
        $.ajax({
            //请求方式：POST请求
            type:"POST",
            //请求的地址
            url:"ajaxController",
            //请求参数（也就是请求内容）
            data:'ajax message',
            //响应正文类型
            dataType:"text",
            //请求正文的MIME类型
            contentType:"application/text",
        });
    });

    //为id="testAjaxPojo"的组件绑定点击事件
    $("#testAjaxPojo").click(function(){
        $.ajax({
            type:"POST",
            url:"ajaxPojoToController",
            data:'{"name":"Jock", "age":39}',
            dataType:"text",
            contentType:"application/json",
        });
    });

    //为id="testAjaxList"的组件绑定点击事件
    $("#testAjaxList").click(function(){
        $.ajax({//.....
            data:'[{"name":"Jock", "age":39}, {"name":"Jockme", "age":40}]'});
    })
});
</script>

```

web.xml配置：请求响应章节请求中的web.xml配置

```
CharacterEncodingFilter + DispatcherServlet
```

spring-mvc.xml:

```
<context:component-scan base-package="controller, domain"/>
<mvc:resources mapping="/js/**" location="/js/" />
<mvc:annotation-driven/>
```

响应数据

注解: @ResponseBody

作用: 将 Java 对象转为 json 格式的数据

方法返回值为 POJO 时, 自动封装数据成 Json 对象数据:

```
@RequestMapping("/ajaxReturnJson")
@ResponseBody
public User ajaxReturnJson(){
    System.out.println("controller return json pojo...");
    User user = new User("Jockme",40);
    return user;
}
```

方法返回值为 List 时, 自动封装数据成 json 对象数组数据:

```
@RequestMapping("/ajaxReturnJsonList")
@ResponseBody
//基于Jackson技术, 使用@ResponseBody注解可以将返回的保存POJO对象的集合转成json数组格式数据
public List ajaxReturnJsonList(){
    System.out.println("controller return json list...");
    User user1 = new User("Tom",3);
    User user2 = new User("Jerry",5);

    ArrayList al = new ArrayList();
    al.add(user1);
    al.add(user2);
    return al;
}
```

AJAX 文件:

```
//为id="testAjaxReturnString"的组件绑定点击事件
$("#testAjaxReturnString").click(function(){
    //发送异步调用
    $.ajax({
        type:"POST",
        url:"ajaxReturnString",
        //回调函数
        success:function(data){
            //打印返回结果
            alert(data);
        }
    });
});

//为id="testAjaxReturnJson"的组件绑定点击事件
$("#testAjaxReturnJson").click(function(){
    $.ajax({
        type:"POST",
        url:"ajaxReturnJson",
        success:function(data){
            alert(data['name']+" , "+data['age']);
        }
    });
});

//为id="testAjaxReturnJsonList"的组件绑定点击事件
$("#testAjaxReturnJsonList").click(function(){
    $.ajax({
        type:"POST",
        url:"ajaxReturnJsonList",
        success:function(data){
            alert(data);
            alert(data[0]["name"]);
            alert(data[1]["age"]);
        }
    });
});
```

跨域访问

跨域访问：当通过域名 A 下的操作访问域名 B 下的资源时，称为跨域访问，跨域访问时，会出现无法访问的现象

环境搭建：

- 为当前主机添加备用域名
 - 修改 windows 安装目录中的 host 文件
 - 格式：ip 域名
- 动态刷新 DNS
 - 命令：ipconfig /displaydns
 - 命令：ipconfig /flushdns

跨域访问支持：

- 名称：@CrossOrigin
- 类型：方法注解、类注解
- 位置：处理器类中的方法上方或类上方
- 作用：设置当前处理器方法 / 处理器类中所有方法支持跨域访问
- 范例：

```
@RequestMapping("/cross")
@ResponseBody
//使用@Crossorigin开启跨域访问
//标注在处理器方法上方表示该方法支持跨域访问
//标注在处理器类上方表示该处理器类中的所有处理器方法均支持跨域访问
@CrossOrigin
public User cross(HttpServletRequest request){
    System.out.println("controller cross..." + request.getRequestURL());
    User user = new User("Jockme",36);
    return user;
}
```

- jsp 文件

```
<a href="javascript:void(0); id="testCross">跨域访问</a><br/>
<script type="text/javascript" src="${pageContext.request.contextPath}/js/jquery-3.3.1.min.js"></script>
<script type="text/javascript">
$(function () {
    //为id="testCross"的组件绑定点击事件
    $("#testCross").click(function (){
        //发送异步调用
        $.ajax({
            type:"POST",
            url:"http://127.0.0.1/cross",
            //回调函数
            success:function(data){
                alert("跨域调用信息反馈：" + data['name'] + " " + data['age']);
            }
        });
    });
});
```

拦截器

基本介绍

拦截器 (Interceptor) 是一种动态拦截方法调用的机制

作用：

1. 在指定的方法调用前后执行预先设定后的代码
2. 阻止原始方法的执行

核心原理：AOP 思想

拦截器链：多个拦截器按照一定的顺序，对原始被调用功能进行增强

拦截器和过滤器对比：

1. 归属不同：Filter 属于 Servlet 技术，Interceptor 属于 SpringMVC 技术
2. 拦截内容不同：Filter 对所有访问进行增强，Interceptor 仅针对 SpringMVC 的访问进行增强



处理方法

前置处理

原始方法之前运行:

```
public boolean preHandle(HttpServletRequest request,
                         HttpServletResponse response,
                         Object handler) throws Exception {
    System.out.println("preHandle");
    return true;
}
```

- 参数:
 - 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
 - handler: public String controller.InterceptorController.handleRun
 - handler.getClass(): org.springframework.web.method.HandlerMethod
- 返回值:
 - 返回值为 false, 被拦截的处理器将不执行

后置处理

原始方法运行后运行, 如果原始方法被拦截, 则不执行:

```
public void postHandle(HttpServletRequest request,
                        HttpServletResponse response,
                        Object handler,
                        ModelAndView modelAndView) throws Exception {
    System.out.println("postHandle");
}
```

参数:

- 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

异常处理

拦截器最后执行的方法, 无论原始方法是否执行:

```
public void afterCompletion(HttpServletRequest request,
                            HttpServletResponse response,
                            Object handler,
                            Exception ex) throws Exception {
    System.out.println("afterCompletion");
}
```

参数:

- 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

拦截配置

拦截路径:

- /**: 表示拦截所有映射
- /*: 表示拦截所有/开头的映射
- /user/*: 表示拦截所有 /user/ 开头的映射
- /user/add*: 表示拦截所有 /user/ 开头, 且具体映射名称以 add 开头的映射
- /user/*All: 表示拦截所有 /user/ 开头, 且具体映射名称以 All 结尾的映射

```
<mvc:interceptors>
    <!--开启具体的拦截器的使用, 可以配置多个-->
    <mvc:interceptor>
        <!--设置拦截器的拦截路径, 支持*通配-->
        <mvc:mapping path="/handleRun"/>
        <!--设置拦截排除的路径, 配置/**或/*, 达到快速配置的目的-->
        <mvc:exclude-mapping path="/b*"/>
        <!--指定具体的拦截器类-->
        <bean class="MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

拦截器链

责任链模式: 责任链模式是一种行为模式

特点: 沿着一条预先设定的任务链顺序执行, 每个节点具有独立的工作任务

优势:

- 独立性: 只关注当前节点的任务, 对其他任务直接放行到下一点
- 隔离性: 具备链式传递特征, 无需知晓整体链路结构, 只需等待请求到达后进行处理即可
- 灵活性: 可以任意修改链路结构动态新增或删减整体链路责任
- 解耦: 将动态任务与原始任务解耦

缺点:

- 链路过长时, 处理效率低下
- 可能存在节点上的循环引用现象, 造成死循环, 导致系统崩溃

拦截器链配置

- 当配置多个拦截器时, 形成拦截器链
- 拦截器链的运行顺序参照配置的先后顺序
- 当拦截器中出现对原始处理器的拦截, 后面的拦截器均终止运行
- 当拦截器运行中断, 仅运行配置在前面的拦截器的afterCompletion操作



源码解析

DispatcherServlet#doDispatch 方法中:

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    try {
        // 获取映射器以及映射器的所有拦截器(运行原理部分讲解了源码)
        mappedHandler = getHandler(processedRequest);
        // 前置处理, 返回 false 代表条件成立
        if (!mappedHandler.applyPreHandle(processedRequest, response)) {
            // 请求从这里直接结束
            return;
        }
        // 所有拦截器都返回 true, 执行目标方法
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
        // 倒序执行所有拦截器的后置处理方法
    }
```

```

        mappedHandler.applyPostHandle(processedRequest, response, mv);
    } catch (Exception ex) {
        //异常处理机制
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
}

```

HandlerExecutionChain#applyPreHandle: 前置处理

```

boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws Exception {
    //遍历所有的拦截器
    for (int i = 0; i < this.interceptorList.size(); i++) {
        HandlerInterceptor interceptor = this.interceptorList.get(i);
        //执行前置处理, 如果拦截器返回 false, 则条件成立, 不在执行其他的拦截器, 直接返回 false, 请求直接结束
        if (!interceptor.preHandle(request, response, this.handler)) {
            triggerAfterCompletion(request, response, null);
            return false;
        }
        this.interceptorIndex = i;
    }
    return true;
}

```

HandlerExecutionChain#applyPostHandle: 后置处理

```

void applyPostHandle(HttpServletRequest request, HttpServletResponse response, @Nullable ModelAndView mv)
    throws Exception {
    //倒序遍历
    for (int i = this.interceptorList.size() - 1; i >= 0; i--) {
        HandlerInterceptor interceptor = this.interceptorList.get(i);
        interceptor.postHandle(request, response, this.handler, mv);
    }
}

```

DispatcherServlet#triggerAfterCompletion 底层调用 HandlerExecutionChain#triggerAfterCompletion:

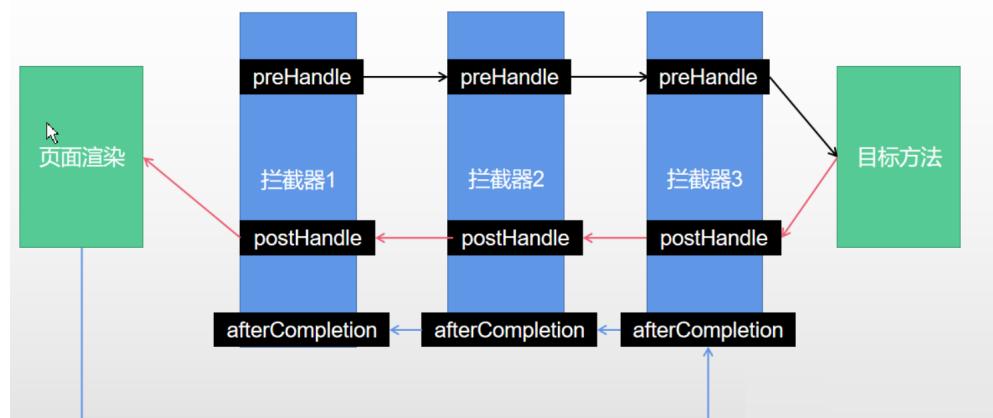
- 前面的步骤有任何异常都会直接倒序触发 afterCompletion
- 页面成功渲染有异常，也会倒序触发 afterCompletion

```

void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, @Nullable Exception ex) {
    //倒序遍历
    for (int i = this.interceptorIndex; i >= 0; i--) {
        HandlerInterceptor interceptor = this.interceptorList.get(i);
        try {
            //执行异常处理的方法
            interceptor.afterCompletion(request, response, this.handler, ex);
        }
        catch (Throwable ex2) {
            logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
        }
    }
}

```

拦截器的执行流程:



参考文章: <https://www.yuque.com/atguigu/springboot/vgzmgh#wtPLU>

自定义

- Controller层

```
@Controller
public class InterceptorController {
    @RequestMapping("/handleRun")
    public String handleRun() {
        System.out.println("业务处理器运行-----main");
        return "page.jsp";
    }
}
```

- 自定义拦截器需要实现 HandlerInterceptor 接口

```
//自定义拦截器需要实现HandlerInterceptor接口
public class MyInterceptor implements HandlerInterceptor {
    //处理器运行之前执行
    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) throws Exception {
        System.out.println("前置运行----a1");
        //返回值为false将拦截原始处理器的运行
        //如果配置多拦截器，返回值为false将终止当前拦截器后面配置的拦截器的运行
        return true;
    }

    //处理器运行之后执行
    @Override
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
                           ModelAndView modelAndView) throws Exception {
        System.out.println("后置运行----b1");
    }

    //所有拦截器的后置执行全部结束后，执行该操作
    @Override
    public void afterCompletion(HttpServletRequest request,
                               HttpServletResponse response,
                               Object handler,
                               Exception ex) throws Exception {
        System.out.println("完成运行----c1");
    }
}
```

说明：三个方法的运行顺序为 preHandle → postHandle → afterCompletion，如果 preHandle 返回值为 false，三个方法仅运行preHandle

- web.xml:

```
CharacterEncodingFilter + DispatcherServlet
```

- 配置拦截器: spring-mvc.xml

```
<mvc:annotation-driven/>
<context:component-scan base-package="interceptor,controller"/>
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/handleRun"/>
        <bean class="interceptor.MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

注意：配置顺序为先配置执行位置，后配置执行类

异常处理

处理器

异常处理器： **HandlerExceptionResolver** 接口

类继承该接口的以后，当开发出现异常后会执行指定的功能

```
@Component
public class ExceptionResolver implements HandlerExceptionResolver {
    @Override
    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;
    }
}
```

根据异常的种类不同，进行分门别类的管理，返回不同的信息：

```
public class ExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
                                         HttpServletResponse response,
                                         Object handler,
                                         Exception ex) {
        System.out.println("my exception is running ...." + ex);
        ModelAndView modelAndView = new ModelAndView();
        if( ex instanceof NullPointerException){
            modelAndView.addObject("msg", "空指针异常");
        }else if ( ex instanceof ArithmeticException){
            modelAndView.addObject("msg", "算数运算异常");
        }else{
            modelAndView.addObject("msg", "未知的异常");
        }
        modelAndView.setViewName("error.jsp");
        return modelAndView;
    }
}
```

模拟错误：

```
@Controller
public class Usercontroller {
    @RequestMapping("/save")
    @ResponseBody
    public String save(@RequestBody String name) {
        //模拟业务层发起调用产生了异常
        //    int i = 1/0;
        //    String str = null;
        //    str.length();

        return "error.jsp";
    }
}
```

注解开发

使用注解实现异常分类管理，开发异常处理器

@ControllerAdvice 注解：

- 类型：类注解
- 位置：异常处理器类上方
- 作用：设置当前类为异常处理器类
- 格式：

```
@Component
//声明该类是一个Controller的通知类，声明后该类就会被加载成异常处理器
@ControllerAdvice
public class ExceptionAdvice {
```

@ExceptionHandler 注解:

- 类型: 方法注解
- 位置: 异常处理器类中针对指定异常进行处理的方法上方
- 作用: 设置指定异常的处理方式
- 说明: 处理器方法可以设定多个
- 格式:

```
@Component
@ControllerAdvice
public class ExceptionAdvice {
    //类中定义的方法携带@ExceptionHandler注解的会被作为异常处理器，后面添加实际处理的异常类型
    @ExceptionHandler(NullPointerException.class)
    @ResponseBody
    public String doNullException(Exception ex){
        return "空指针异常";
    }

    @ExceptionHandler(Exception.class)
    @ResponseBody
    public String doException(Exception ex){
        return "all Exception";
    }
}
```

@ResponseStatus 注解:

- 类型: 类注解、方法注解
- 位置: 异常处理器类、方法上方
- 参数:
 - value: 出现错误指定返回状态码
 - reason: 出现错误返回的错误信息

解决方案

- web.xml

```
DispatcherServlet + CharacterEncodingFilter
```

- ajax.jsp

```
<%@page pageEncoding="UTF-8" language="java" contentType="text/html;UTF-8" %>

<a href="javascript:void(0); id="testException">点击</a><br/>

<script type="text/javascript" src="${pageContext.request.contextPath}/js/jquery-3.3.1.min.js"></script>
<script type="text/javascript">
$(function () {
    $("#testException").click(function(){
        $.ajax({
            contentType:"application/json",
            type:"POST",
            url:"save",
            /*通过修改参数，激活自定义异常的出现*/
            // name长度低于8位出现业务异常
            // age小于0出现业务异常
            // age大于100出现系统异常
            // age类型如果无法匹配将转入其他类别异常
            data:'{"name":"JockSuperMan","age": "-1"}',
            dataType:"text",
            //回调函数
            success:function(data){
                alert(data);
            }
        });
    });
});
</script>
```

- spring-mvc.xml

```
<mvc:annotation-driven/>
<context:component-scan base-package="com.seazean"/>
<mvc:resources mapping="/js/**" location="/js/" />
```

- o java / controller / UserController

```
@Controller
public class UserController {
    @RequestMapping("/save")
    @ResponseBody
    public List<User> save(@RequestBody User user) {
        System.out.println("user controller save is running ...");
        //对用户的非法操作进行判定，并包装成异常对象进行处理，便于统一管理
        if(user.getName().trim().length() < 8){
            throw new BusinessException("对不起，用户名长度不满足要求，请重新输入！");
        }
        if(user.getAge() < 0){
            throw new BusinessException("对不起，年龄必须是0到100之间的数字！");
        }
        if(user.getAge() > 100){
            throw new SystemException("服务器连接失败，请尽快检查处理！");
        }

        User u1 = new User("Tom",3);
        User u2 = new User("Jerry",5);
        ArrayList<User> al = new ArrayList<User>();
        al.add(u1);al.add(u2);
        return al;
    }
}
```

- o 自定义异常

```
//自定义异常继承RuntimeException，覆盖父类所有的构造方法
public class BusinessException extends RuntimeException {覆盖父类所有的构造方法}
```

```
public class SystemException extends RuntimeException {}
```

- o 通过自定义异常将所有的异常现象进行分类管理，以统一的格式对外呈现异常消息

```
@Component
@ControllerAdvice
public class ProjectExceptionAdvice {
    @ExceptionHandler(BusinessException.class)
    public String doBusinessException(Exception ex, Model m){
        //使用参数Model将要保存的数据传递到页面上，功能等同于 ModelAndView
        //业务异常出现的消息要发送给用户查看
        m.addAttribute("msg",ex.getMessage());
        return "error.jsp";
    }

    @ExceptionHandler(SystemException.class)
    public String doSystemException(Exception ex, Model m){
        //系统异常出现的消息不要发送给用户查看，发送统一的信息给用户看
        m.addAttribute("msg","服务器出现问题，请联系管理员！");
        return "error.jsp";
    }

    @ExceptionHandler(Exception.class)
    public String doException(Exception ex, Model m){
        m.addAttribute("msg",ex.getMessage());
        //将ex对象保存起来
        return "error.jsp";
    }
}
```

文件传输

上传下载

上传文件过程：

```
<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>
```



- 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

MultipartResolver

MultipartResolver接口：

- MultipartResolver 接口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 文件上传下载组件

文件上传下载实现：

- 导入坐标

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
```

- 页面表单 fileupload.jsp

```
<form method="post" action="/upload" enctype="multipart/form-data">
    <input type="file" name="file"><br>
    <input type="submit" value="提交">
</form>
```

- web.xml

```
DispatcherServlet + CharacterEncodingFilter
```

- 控制器

```
@PostMapping("/upload")
public String upload(@RequestParam("email") String email,
                     @RequestParam("username") String username,
                     @RequestPart("headerImg") MultipartFile headerImg) throws IOException {
    if(!headerImg.isEmpty()){
        //保存到文件服务器，OSS服务器
        String originalFilename = headerImg.getOriginalFilename();
        headerImg.transferTo(new File("H:\\cache\\\" + originalFilename));
    }
    return "main";
}
```

名称问题

MultipartFile 参数中封装了上传的文件的相关信息。

1. 文件命名问题，获取上传文件名，并解析文件名与扩展名

```
file.getOriginalFilename();
```

2. 文件名过长问题

3. 文件保存路径

```
ServletContext context = request.getServletContext();
String realPath = context.getRealPath("/uploads");
File file = new File(realPath + "/");
if(!file.exists()) file.mkdirs();
```

4. 重名问题

```
String uuid = UUID.randomUUID().toString().replace("-", "").toUpperCase();
```

```
@Controller
public class FileuploadController {
    @RequestMapping(value = "/fileupload")
    //参数中定义MultipartFile参数，用于接收页面提交的type=file类型的表单，表单名称与参数名相同
    public String fileupload(MultipartFile file,MultipartFile file1,MultipartFile file2, HttpServletRequest request) throws IOException {
        System.out.println("file upload is running ..." + file);
        // MultipartFile参数中封装了上传的文件的相关信息
        // System.out.println(file.getSize());
        // System.out.println(file.getBytes().length);
        // System.out.println(file.getContentType());
        // System.out.println(file.getName());
        // System.out.println(file.getOriginalFilename());
        // System.out.println(file.isEmpty());
        //首先判断是否是空文件，也就是存储空间占用为0的文件
        if(!file.isEmpty()){
            //如果大小在范围要求内正常处理，否则抛出自定义异常告知用户（未实现）
            //获取原始上传的文件名，可以作为当前文件的真实名称保存到数据库中备用
            String fileName = file.getOriginalFilename();
            //设置保存的路径
            String realPath = request.getServletContext().getRealPath("/images");
            //保存文件的方法，通常文件名使用随机生成策略产生，避免文件名冲突问题
            file.transferTo(new File(realPath,fileName));
        }
        //测试一次性上传多个文件
        if(!file1.isEmpty()){
            String fileName = file1.getOriginalFilename();
            //可以根据需要，对不同种类的文件做不同的存储路径的区分，修改对应的保存位置即可
            String realPath = request.getServletContext().getRealPath("/images");
            file1.transferTo(new File(realPath,fileName));
        }
        if(!file2.isEmpty()){
            String fileName = file2.getOriginalFilename();
            String realPath = request.getServletContext().getRealPath("/images");
            file2.transferTo(new File(realPath,fileName));
        }
        return "page.jsp";
    }
}
```

源码解析

StandardServletMultipartResolver 是文件上传解析器

DispatcherServlet#doDispatch:

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    // 判断当前请求是不是文件上传请求
    processedRequest = checkMultipart(request);
    // 文件上传请求会对 request 进行包装，导致两者不相等，此处赋值为 true，代表已经被解析
    multipartRequestParsed = (processedRequest != request);
}
```

DispatcherServlet#checkMultipart:

- o `if (this.multipartResolver != null && this.multipartResolver.isMultipart(request))`: 判断是否是文件请求
 - `StandardServletMultipartResolver#isMultipart`: 根据开头是否符合 multipart/form-data 或者 multipart/
- o `return this.multipartResolver.resolveMultipart(request)`: 把请求封装成 StandardMultipartHttpServletRequest 对象

开始执行 ha.handle() 目标方法进行数据的解析

- o `RequestPartMethodArgumentResolver#supportsParameter`: 支持解析文件上传数据

```

public boolean supportsParameter(MethodParameter parameter) {
    // 参数上有 @RequestPart 注解
    if (parameter.hasParameterAnnotation(RequestPart.class)) {
        return true;
    }
}

```

- RequestPartMethodArgumentResolver#resolveArgument: 解析参数数据，封装成 MultipartFile 对象
 - RequestPart requestPart = parameter.getParameterAnnotation(RequestPart.class)：获取注解的相关信息
 - String name = getPartName(parameter, requestPart)：获取上传文件的名字
 - Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument()：解析参数
 - List<MultipartFile> files = multipartRequest.getFiles(name)：获取文件的所有数据
- return doInvoke(args)：解析完成执行自定义的方法，完成上传功能

实用技术

校验框架

校验概述

表单校验保障了数据有效性、安全性

校验分类：客户端校验和服务端校验

- 格式校验
 - 客户端：使用 js 技术，利用正则表达式校验
 - 服务端：使用校验框架
- 逻辑校验
 - 客户端：使用ajax发送要校验的数据，在服务端完成逻辑校验，返回校验结果
 - 服务端：接收到完整的请求后，在执行业务操作前，完成逻辑校验

表单校验框架：

- JSR (Java Specification Requests)：Java 规范提案
- 303：提供bean属性相关校验规则
- JCP (Java Community Process)：Java 社区
- Hibernate 框架中包含一套独立的校验框架hibernate-validator
- 导入坐标：

```

<!-- 导入校验的jsr303规范-->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
<!-- 导入校验框架实现技术-->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.1.0.Final</version>
</dependency>

```

注意：

- tomcat7：搭配 hibernate-validator 版本 5..Final
- tomcat8.5：搭配 hibernate-validator 版本 6..Final

基本使用

开启校验

名称：@Valid、@Validated

类型：形参注解

位置：处理器类中的实体类类型的方法形参前方

作用：设定对当前实体类类型参数进行校验

范例：

```

@RequestMapping(value = "/addemployee")
public String addEmployee(@Valid Employee employee) {
    System.out.println(employee);
}

```

校验规则

名称: @NotNull

类型: 属性注解等

位置: 实体类属性上方

作用: 设定当前属性校验规则

范例: 每个校验规则所携带的参数不同, 根据校验规则进行相应的调整, 具体的校验规则查看对应的校验框架进行获取

```

public class Employee{
    @NotNull(message = "姓名不能为空")
    private String name;//员工姓名
}

```

错误信息

```

@RequestMapping(value = "/addemployee")
//Errors对象用于封装校验结果, 如果不满足校验规则, 对应的校验结果封装到该对象中, 包含校验的属性名和校验不通过返回的消息
public String addEmployee(@Valid Employee employee, Errors errors, Model model){
    System.out.println(employee);
    //判定Errors对象中是否存未通过校验的字段
    if(errors.hasErrors()){
        for(FieldError error : errors.getFieldErrors()){
            //将校验结果添加到Model对象中, 用于页面显示, 返回json数据即可
            model.addAttribute(error.getField(),error.getDefaultMessage());
        }
    }
    //当出现未通过校验的字段时, 跳转页面到原始页面, 进行数据回显
    return "addemployee.jsp";
}
return "success.jsp";
}

```

通过形参Errors获取校验结果数据, 通过Model接口将数据封装后传递到页面显示, 页面获取后台封装的校验结果信息

```

<form action="/addemployee" method="post">
    员工姓名: <input type="text" name="name"><span style="color:red">${name}</span><br/>
    员工年龄: <input type="text" name="age"><span style="color:red">${age}</span><br/>
    <input type="submit" value="提交">
</form>

```

多规则校验

- 同一个属性可以添加多个校验器

```

public class Employee{
    @NotBlank(message = "姓名不能为空")
    private String name;//员工姓名

    @NotNull(message = "请输入年龄")
    @Max(value = 60,message = "年龄最大值60")
    @Min(value = 18,message = "年龄最小值18")
    private Integer age;//员工年龄
}

```

- 三种判定空校验器的区别

表单数据	@NotNull	@NotEmpty	@NotBlank
String s = null;	false	false	false
String s = " " ;	true	false	false
String s = " " ;	true	true	false
String s = "Jock" ;	true	true	true

嵌套校验

名称: @Valid

类型: 属性注解

位置: 实体类中的引用类型属性上方

作用: 设定当前应用类型属性中的属性开启校验

范例:

```
public class Employee {  
    //实体类中的引用类型通过标注@Valid注解，设定开启当前引用类型字段中的属性参与校验  
    @Valid  
    private Address address;  
}
```

注意: 开启嵌套校验后, 被校验对象内部需要添加对应的校验规则

```
//嵌套校验的实体中，对每个属性正常添加校验规则即可  
public class Address implements Serializable {  
    @NotBlank(message = "请输入省份名称")  
    private String provinceName;//省份名称  
  
    @NotBlank(message = "请输入邮政编码")  
    @Size(max = 6,min = 6,message = "邮政编码由6位组成")  
    private String zipcode;//邮政编码  
}
```

分组校验

分组校验的介绍

- 同一个模块, 根据执行的业务不同, 需要校验的属性会有不同
 - 新增用户
 - 修改用户
- 对不同种类的属性进行分组, 在校验时可以指定参与校验的字段所属的组类别
 - 定义组 (通用)
 - 为属性设置所属组, 可以设置多个
 - 开启组校验

domain:

```
//用于设定分组校验中的组名, 当前接口仅提供字节码, 用于识别  
public interface GroupOne {  
}  
  
public class Employee{  
    @NotBlank(message = "姓名不能为空",groups = {GroupA.class})  
    private String name;//员工姓名  
  
    @NotNull(message = "请输入年龄",groups = {GroupA.class})  
    @Max(value = 60,message = "年龄最大值60")//不加Group的校验不生效  
    @Min(value = 18,message = "年龄最小值18")  
    private Integer age;//员工年龄  
  
    @Valid  
    private Address address;  
    //.....  
}
```

controller:

```
@Controller  
public class EmployeeController {  
    @RequestMapping(value = "/addemployee")  
    public String addEmployee(@Validated({GroupA.class}) Employee employee, Errors errors, Model m){  
        if(errors.hasErrors()){  
            List<FieldError> fieldErrors = errors.getFieldErrors();  
            System.out.println(fieldErrors.size());  
            for(FieldError error : fieldErrors){  
                m.addAttribute(error.getField(),error.getDefaultMessage());  
            }  
        }  
        return "addEmployee";  
    }  
}
```

```
        }
        return "addemployee.jsp";
    }
}
```

jsp:

```
<form action="/addemployee" method="post"><%--页面使用${}获取后台传递的校验信息--%>
    员工姓名: <input type="text" name="name"><span style="color:red">${name}</span><br/>
    员工年龄: <input type="text" name="age"><span style="color:red">${age}</span><br/>
    <%--注意，引用类型的校验未通过信息不是通过对对象进行封装的，直接使用对象名.属性名的格式作为整体属性字符串进行保存的，和使用者的属性传递方式有关，不具有通用性，仅适用于本案例--%>
    省: <input type="text" name="address.provinceName"><span style="color:red">${requestScope['address.provinceName']}</span><br/>
        <input type="submit" value="提交">
</form>
```

Lombok

Lombok 用标签方式代替构造器、getter/setter、toString() 等方法

引入依赖:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

下载插件: IDEA 中 File → Settings → Plugins, 搜索安装 Lombok 插件

常用注解:

```
@NoArgsConstructor      // 无参构造
@NoArgsConstructor      // 全参构造
@Data             // set + get
@ToString          // toString
@EqualsAndHashCode   // hashCode + equals
```

简化日志:

```
@Slf4j
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String handle01(@RequestParam("name") String name){
        log.info("请求进来了....");
        return "Hello, Spring!" + "你好: " + name;
    }
}
```

Boot

基本介绍

Boot介绍

SpringBoot 提供了一种快速使用 Spring 的方式，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率

SpringBoot 功能：

- 自动配置，自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素选择使用哪个配置，该过程是SpringBoot 自动完成的
- 起步依赖，起步依赖本质上是一个 Maven 项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能
- 辅助功能，提供了一些大型项目中常见的非功能性特性，如内嵌 web 服务器、安全、指标，健康检测、外部配置等

构建工程

普通构建:

1. 创建 Maven 项目
2. 导入 SpringBoot 起步依赖

```
<!--springboot 工程需要继承的父工程-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.8.RELEASE</version>
</parent>

<dependencies>
    <!--web 开发的起步依赖-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

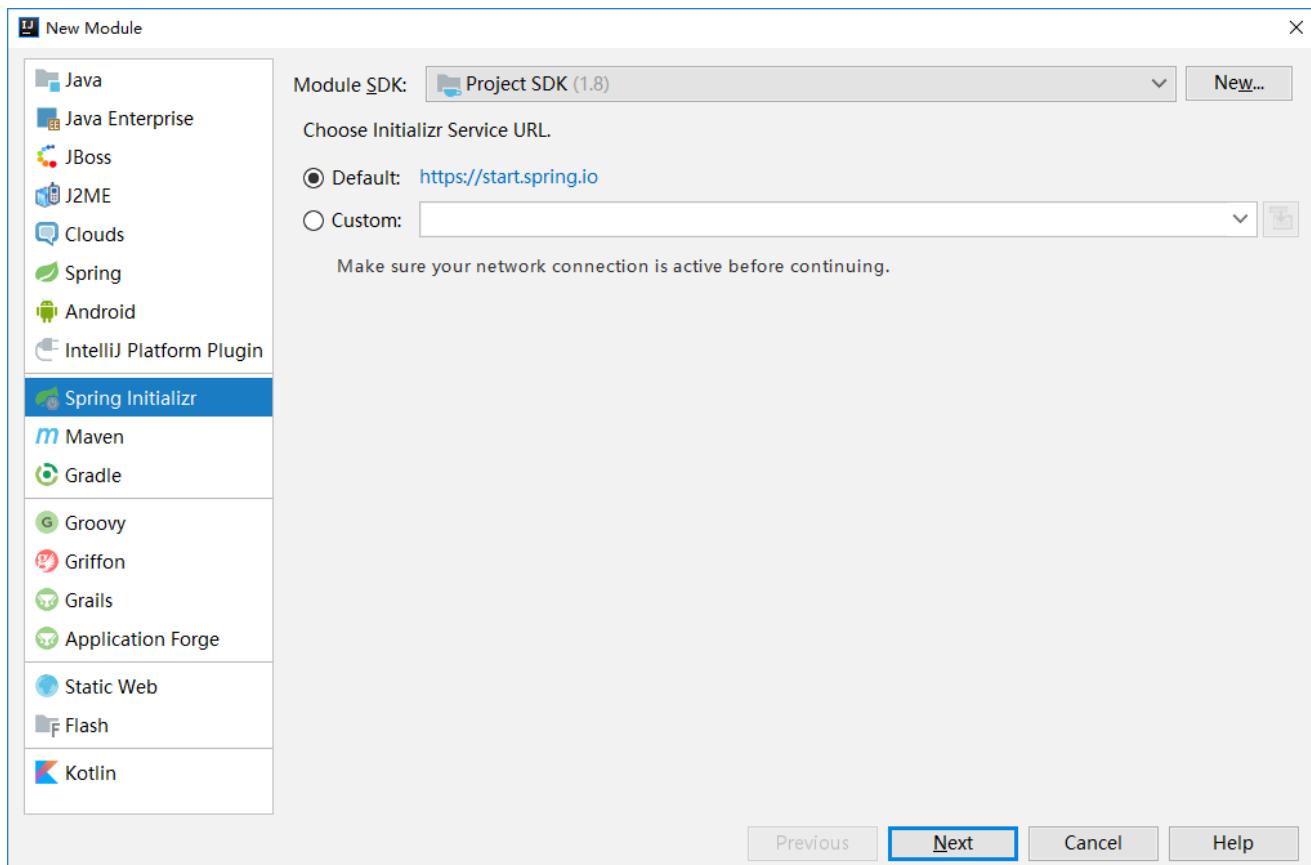
3. 定义 Controller

```
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello(){
        return " hello Spring Boot !";
    }
}
```

4. 编写引导类

```
// 引导类, SpringBoot项目的入口
@SpringBootApplication
public class HelloApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }
}
```

快速构建:



自动装配

依赖管理

在 spring-boot-starter-parent 中定义了各种技术的版本信息，组合了一套最优搭配的技术版本。在各种 starter 中，定义了完成该功能需要的坐标合集，其中大部分版本信息来自于父工程。工程继承 parent，引入 starter 后，通过依赖传递，就可以简单方便获得需要的 jar 包，并且不会存在版本冲突，自动版本仲裁机制。

底层注解

SpringBoot

@SpringBootApplication: 启动注解，实现 SpringBoot 的自动部署

- 参数 scanBasePackages: 可以指定扫描范围
- 默认扫描当前引导类所在包及其子包

假如所在包为 com.example.springbootenable，扫描配置包 com.example.config 的信息，三种解决办法：

1. 使用 @ComponentScan 扫描 com.example.config 包
2. 使用 @Import 注解加载类，这些类都会被 Spring 创建并放入 ioc 容器，默认组件的名字就是**全类名**
3. 对 @Import 注解进行封装

```
//1.@ComponentScan("com.example.config")
//2.@Import(UserConfig.class)
@EnableUser
@SpringBootApplication
public class SpringbootEnableApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(SpringbootEnableApplication.class, args);
        //获取Bean
        Object user = context.getBean("user");
        System.out.println(user);
    }
}
```

```
}
```

UserConfig:

```
@Configuration
public class UserConfig {
    @Bean
    public User user() {
        return new User();
    }
}
```

EnableUser 注解类:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(UserConfig.class)//@Import注解实现Bean的动态加载
public @interface EnableUser {
}
```

Configuration

@Configuration: 设置当前类为 SpringBoot 的配置类

- proxyBeanMethods = true: Full 全模式，每个 @Bean 方法被调用多少次返回的组件都是单实例的，默认值，类组件之间有依赖关系，方法会被调用得到之前单实例组件
- proxyBeanMethods = false: Lite 轻量级模式，每个 @Bean 方法被调用多少次返回的组件都是新创建的，类组件之间无依赖关系用 Lite 模式加速容器启动过程

```
@Configuration(proxyBeanMethods = true)
public class MyConfig {
    @Bean //给容器中添加组件。以方法名作为组件的 id。返回类型就是组件类型。返回的值，就是组件在容器中的实例
    public User user(){
        User user = new User("zhangsan", 18);
        return user;
    }
}
```

Condition

条件注解

Condition 是 Spring4.0 后引入的条件化配置接口，通过实现 Condition 接口可以完成有条件的加载相应的 Bean

注解: @Conditional

作用: 条件装配，满足 Conditional 指定的条件则进行组件注入，加上方法或者类上，作用范围不同

使用: @Conditional 配合 Condition 的实现类 (ClassCondition) 进行使用

ConditionContext 类 API:

方法	说明
ConfigurableListableBeanFactory getBeanFactory()	获取到 IOC 使用的 beanfactory
ClassLoader getClassLoader()	获取类加载器
Environment getEnvironment()	获取当前环境信息
BeanDefinitionRegistry getRegistry()	获取到 bean 定义的注册类

- ClassCondition:

```
public class ClassCondition implements Condition {
    /**
     * context 上下文对象。用于获取环境，IOC容器，ClassLoader对象
     * metadata 注解元对象。 可以用于获取注解定义的属性值
     */
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
```

```

//1.需求：导入Jedis坐标后创建Bean
//思路：判断redis.clients.jedis.Jedis.class文件是否存在
boolean flag = true;
try {
    Class<?> cls = Class.forName("redis.clients.jedis.Jedis");
} catch (ClassNotFoundException e) {
    flag = false;
}
return flag;
}
}

```

- UserConfig:

```

@Configuration
public class UserConfig {
    @Bean
    @Conditional(ClassCondition.class)
    public User user(){
        return new User();
    }
}

```

- 启动类:

```

@SpringBootApplication
public class SpringbootConditionApplication {
    public static void main(String[] args) {
        //启动SpringBoot应用，返回Spring的IOC容器
        ConfigurableApplicationContext context = SpringApplication.run(SpringbootConditionApplication.class, args);

        Object user = context.getBean("user");
        System.out.println(user);
    }
}

```

自定义注解

将类的判断定义为动态的，判断哪个字节码文件存在可以动态指定

- 自定义条件注解类

```

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(ClassCondition.class)
public @interface ConditionOnClass {
    String[] value();
}

```

- ClassCondition

```

public class ClassCondition implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata) {

        //需求：通过注解属性值value指定坐标后创建bean
        Map<String, Object> map = metadata.getAnnotationAttributes(
            ConditionOnClass.class.getName());
        //map = {value={属性值}}
        //获取所有的
        String[] value = (String[]) map.get("value");

        boolean flag = true;
        try {
            for (String className : value) {
                Class<?> cls = Class.forName(className);
            }
        } catch (Exception e) {
            flag = false;
        }
        return flag;
    }
}

```

- UserConfig

```
@Configuration
public class UserConfig {
    @Bean
    @ConditionOnClass("com.alibaba.fastjson.JSON")//JSON加载了才注册 user 到容器
    public User user(){
        return new User();
    }
}
```

- 测试 User 对象的创建

常用注解

SpringBoot 提供的常用条件注解：

@ConditionalOnProperty：判断配置文件中是否有对应属性和值才初始化 Bean

```
@Configuration
public class UserConfig {
    @Bean
    @ConditionalOnProperty(name = "it", havingValue = "seazean")
    public User user() {
        return new User();
    }
}
```

```
it=seazean
```

@ConditionalOnClass：判断环境中是否有对应类文件才初始化 Bean

@ConditionalOnMissingClass：判断环境中是否有对应类文件才初始化 Bean

@ConditionalOnMissingBean：判断环境中没有对应 Bean 才初始化 Bean

ImportRes

使用 bean.xml 文件生成配置 bean，如果需要继续复用 bean.xml，@ImportResource 导入配置文件即可

```
@ImportResource("classpath:beans.xml")
public class MyConfig {
    //...
}
```

```
<beans ...>
    <bean id="haha" class="com.lun.boot.bean.User">
        <property name="name" value="zhangsan"></property>
        <property name="age" value="18"></property>
    </bean>

    <bean id="hehe" class="com.lun.boot.bean.Pet">
        <property name="name" value="tomcat"></property>
    </bean>
</beans>
```

Properties

@ConfigurationProperties：读取到 properties 文件中的内容，并且封装到 JavaBean 中

配置文件：

```
mycar.brand=BYD
mycar.price=100000
```

JavaBean 类：

```

@Component //导入到容器内
@ConfigurationProperties(prefix = "mycar")//代表配置文件的前缀
public class Car {
    private String brand;
    private Integer price;
}

```

装配原理

启动流程

应用启动:

```

@SpringBootApplication
public class BootApplication {
    public static void main(String[] args) {
        // 启动代码
        SpringApplication.run(BootApplication.class, args);
    }
}

```

SpringApplication 构造方法:

- `this.resourceLoader = resourceLoader`: 资源加载器, 初始为 null
- `this.webApplicationType = webApplicationType.deduceFromClasspath()`: 判断当前应用的类型, 是响应式还是 Web 类
- `this.bootstrapRegistryInitializers = getBootstrapRegistryInitializersFromSpringFactories()`: 获取引导器
 - 去 `META-INF/spring.factories` 文件中找 `org.springframework.boot.Bootstrapper`
 - 寻找的顺序: classpath → spring-beans → boot-devtools → springboot → boot-autoconfigure
- `setInitializers(getSpringFactoriesInstances(ApplicationContextInitializer.class))`: 获取初始化器
 - 去 `META-INF/spring.factories` 文件中找 `org.springframework.context.ApplicationContextInitializer`
- `setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class))`: 获取监听器
 - 去 `META-INF/spring.factories` 文件中找 `org.springframework.context.ApplicationListener`
- `this.mainApplicationClass = deduceMainApplicationClass()`: 获取出 main 程序类

SpringApplication#run(String... args): 创建 IOC 容器并实现了自动装配

- `Stopwatch stopwatch = new Stopwatch()`: 停止监听器, 监控整个应用的启停
- `stopwatch.start()`: 记录应用的启动时间
- `bootstrapContext = createBootstrapContext()`: 创建引导上下文环境
 - `bootstrapContext = new DefaultBootstrapContext()`: 创建默认的引导类环境
 - `this.bootstrapRegistryInitializers.forEach()`: 遍历所有的引导器调用 `initialize` 方法完成初始化设置
- `configureHeadlessProperty()`: 让当前应用进入 headless 模式
- `listeners = getRunListeners(args)`: 获取所有 RunListener (运行监听器)
 - 去 `META-INF/spring.factories` 文件中找 `org.springframework.boot.SpringApplicationRunListener`
- `listeners.starting(bootstrapContext, this.mainApplicationClass)`: 遍历所有的运行监听器调用 `starting` 方法
- `applicationArguments = new DefaultApplicationArguments(args)`: 获取所有的命令行参数
- `environment = prepareEnvironment(listeners, bootstrapContext, applicationArguments)`: 准备环境
 - `environment = getOrCreateEnvironment()`: 返回或创建基础环境信息对象
 - `switch (this.webApplicationType)`: 根据当前应用的类型创建环境
 - `case SERVLET`: Web 应用环境对应 `ApplicationServletEnvironment`
 - `case REACTIVE`: 响应式编程对应 `ApplicationReactiveWebEnvironment`
 - `default`: 默认为 Spring 环境 `ApplicationEnvironment`
 - `configureEnvironment(environment, applicationArguments.getSourceArgs())`: 读取所有配置源的属性值配置环境
 - `ConfigurationPropertySources.attach(environment)`: 属性值绑定环境信息
 - `sources.addFirst(ATTACHED_PROPERTY_SOURCE_NAME,..)`: 把 `configurationProperties` 放入环境的属性信息头部
 - `listeners.environmentPrepared(bootstrapContext, environment)`: 运行监听器调用 `environmentPrepared()`, `EventPublishingRunListener` 发布事件通知所有的监听器当前环境准备完成
 - `DefaultPropertiesPropertySource.moveToEnd(environment)`: 移动 `defaultProperties` 属性源到环境中的最后一个源
 - `bindToSpringApplication(environment)`: 与容器绑定当前环境
 - `ConfigurationPropertySources.attach(environment)`: 重新将属性值绑定环境信息
 - `sources.remove(ATTACHED_PROPERTY_SOURCE_NAME)`: 从环境信息中移除 `configurationProperties`
 - `sources.addFirst(ATTACHED_PROPERTY_SOURCE_NAME,..)`: 把 `configurationProperties` 重新放入环境信息
- `configureIgnoreBeanInfo(environment)`: 配置忽略的 bean
- `printedBanner = printBanner(environment)`: 打印 SpringBoot 标志

- o `context = createApplicationContext()` : 创建 IOC 容器
 - `switch (this.webApplicationType)` : 根据当前应用的类型创建 IOC 容器
 - `case SERVLET`: Web 应用环境对应 AnnotationConfigServletWebServerApplicationContext
 - `case REACTIVE`: 响应式编程对应 AnnotationConfigReactiveWebServerApplicationContext
 - `default`: 默认为 Spring 环境 AnnotationConfigApplicationContext
 - o `context.setApplicationStartup(this.applicationStartup)` : 设置一个启动器
 - o `prepareContext()` : 配置 IOC 容器的基本信息
 - `postProcessApplicationContext(context)`: 后置处理流程
 - `applyInitializers(context)`: 获取所有的初始化器调用 `initialize()` 方法进行初始化
 - `listeners.contextPrepared(context)`: 所有的运行监听器调用 `environmentPrepared()` 方法, `EventPublishingRunListener` 发布事件通知 IOC 容器准备完成
 - `listeners.contextLoaded(context)`: 所有的运行监听器调用 `contextLoaded()` 方法, 通知 IOC 加载完成
 - o `refreshContext(context)` : 刷新 IOC 容器
 - Spring 的容器启动流程
 - `invokeBeanFactoryPostProcessors(beanFactory)` : 实现了自动装配
 - `onRefresh()` : 创建 WebServer 使用该接口
 - o `afterRefresh(context, applicationArguments)` : 留给用户自定义容器刷新完成后的处理逻辑
 - o `stopwatch.stop()` : 记录应用启动完成的时间
 - o `callRunners(context, applicationArguments)` : 调用所有 runners
 - o `listeners.started(context)` : 所有的运行监听器调用 `started()` 方法
 - o `listeners.running(context)` : 所有的运行监听器调用 `running()` 方法
 - 获取容器中的 ApplicationRunner、CommandLineRunner
 - `AnnotationAwareOrderComparator.sort(runners)` : 合并所有 runner 并且按照 @Order 进行排序
 - `callRunner()` : 遍历所有的 runner, 调用 run 方法
 - o `handleRunFailure(context, ex, listeners)` : 处理异常, 出现异常进入该逻辑
 - `handleExitCode(context, exception)` : 处理错误代码
 - `listeners.failed(context, exception)` : 运行监听器调用 `failed()` 方法
 - `reportFailure(getExceptionReporters(context), exception)` : 通知异常
-

注解分析

SpringBoot 定义了一套接口规范, 这套规范规定 SpringBoot 在启动时会扫描外部引用 jar 包中的 `META-INF/spring.factories` 文件, 将文件中配置的类型信息加载到 Spring 容器, 并执行类中定义的各种操作, 对于外部的 jar 包, 直接引入一个 starter 即可

`@SpringBootApplication` 注解是 `@SpringBootConfiguration`、`@EnableAutoConfiguration`、`@ComponentScan` 注解的集合

- o `@SpringBootApplication` 注解

```

@Inherited
@Configuration // 代表 @SpringBootApplication 拥有了该注解的功能
@EnableAutoConfiguration // 同理
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
// 扫描被 @Component (@Service,@Controller)注解的 bean, 容器中将排除TypeExcludeFilter 和 AutoConfigurationExcludeFilter
public @interface SpringApplication { }

```

- o `@SpringBootConfiguration` 注解:

```

@Configuration // 代表是配置类
@Indexed
public @interface SpringBootConfiguration {
    @AliasFor(annotation = Configuration.class)
    boolean proxyBeanMethods() default true;
}

```

`@AliasFor` 注解: 表示别名, 可以注解到自定义注解的两个属性上表示这两个互为别名, 两个属性其实是同一个含义相互替代

- o `@ComponentScan` 注解: 默认扫描当前类所在包及其子级包下的所有文件

- o `@EnableAutoConfiguration` 注解: 启用 SpringBoot 的自动配置机制

```

@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
}

```

- @AutoConfigurationPackage：将添加该注解的类所在的 package 作为自动配置 package 进行管理，把启动类所在的包设置一次，为了给各种自动配置的第三方库扫描用，比如带 @Mapper 注解的类，Spring 自身是不能识别的，但自动配置的 Mybatis 需要扫描用到，而 ComponentScan 只是用来扫描注解类，并没有提供接口给三方使用

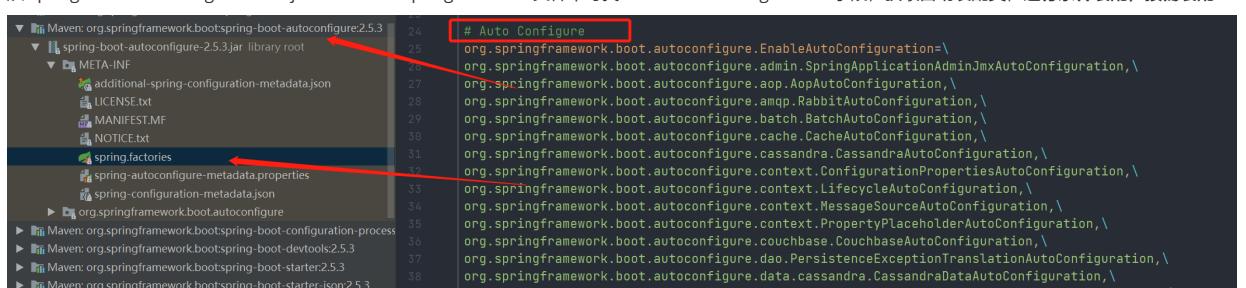
```
@Import(AutoConfigurationPackages.Registrar.class) // 利用 Registrar 给容器中导入组件
public interface AutoConfigurationPackage {
    String[] basePackages() default {}; // 自动配置包，指定了配置类的包
    Class<?>[] basePackageClasses() default {};
}
```

- register(registry, new PackageImports(metadata).getPackageNames().toArray(new String[0]))：注册 BD
 - new PackageImports(metadata).getPackageNames()：获取添加当前注解的类的所在包
 - registry.registerBeanDefinition(BEAN, new BasePackagesBeanDefinition(packageNames))：存放到容器中
 - new BasePackagesBeanDefinition(packageNames)：把当前主类所在的包名封装到该对象中
- @Import(AutoConfigurationImportSelector.class)：自动装配的核心类
 容器刷新时执行：invokeBeanFactoryPostProcessors() → invokeBeanDefinitionRegistryPostProcessors() → postProcessBeanDefinitionRegistry() → processConfigBeanDefinitions() → parse() → process() → processGroupImports() → getImports() → process() → AutoConfigurationImportSelector#getAutoConfigurationEntry()

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    // 获取注解属性，@SpringBootApplication 注解的 exclude 属性和 excludeName 属性
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    // 获取所有需要自动装配的候选项
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    // 去除重复的选项
    configurations = removeDuplicates(configurations);
    // 获得注解配置的排除的自动装配类
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    // 移除所有的配置的不需要自动装配的类
    configurations.removeAll(exclusions);
    // 过滤，条件装配
    configurations = getConfigurationClassFilter().filter(configurations);
    // 获得 AutoConfigurationImportListener 类的监听器调用 onAutoConfigurationImportEvent 方法
    fireAutoConfigurationImportEvents(configurations, exclusions);
    // 包装成 AutoConfigurationEntry 返回
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

AutoConfigurationImportSelector#getCandidateConfigurations：获取自动配置的候选项

- List<String> configurations = SpringFactoriesLoader.loadFactoryNames()：加载自动配置类
 - 参数一：getSpringFactoriesLoaderFactoryClass()：获取 @EnableAutoConfiguration 注解类
 - 参数二：getBeanClassLoader()：获取类加载器
 - factoryTypeName = factoryType.getName()：@EnableAutoConfiguration 注解的全类名
 - return loadSpringFactories(classLoaderToUse).getOrDefault()：加载资源
 - urls = classLoader.getResources(FACTORIES_RESOURCE_LOCATION)：获取资源类
 - FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories"：加载的资源的位置
 - return configurations：返回所有自动装配类的候选项
- 从 spring-boot-autoconfigure-2.5.3.jar/META-INF/spring.factories 文件中寻找 EnableAutoConfiguration 字段，获取自动装配类，进行条件装配，按需装配



装配流程

Spring Boot 通过 `@EnableAutoConfiguration` 开启自动装配，通过 `SpringFactoriesLoader` 加载 `META-INF/spring.factories` 中的自动配置类实现自动装配，自动配置类其实就是通过 `@Conditional` 注解按需加载的配置类，想要其生效必须引入 `spring-boot-starter-xxx` 包实现起步依赖

- SpringBoot 先加载所有的自动配置类 `xxxxAutoConfiguration`
- 每个自动配置类进行**条件装配**，默认都会绑定配置文件指定的值（`xxxProperties` 和配置文件进行了绑定）
- SpringBoot 默认会在底层配好所有的组件，如果用户自己配置了**以用户的优先**
- **定制化配置：**
 - 用户可以使用 `@Bean` 新建自己的组件来替换底层的组件
 - 用户可以去看这个组件是获取的配置文件前缀值，在配置文件中修改

以 `DispatcherServletAutoConfiguration` 为例：

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
// 类中的 Bean 默认不是单例
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
// 条件装配，环境中有 DispatcherServlet 类才进行自动装配
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {
    // 注册的 DispatcherServlet 的 BeanName
    public static final String DEFAULT_DISPATCHER_SERVLET_BEAN_NAME = "dispatcherServlet";

    @Configuration(proxyBeanMethods = false)
    @Conditional(DefaultDispatcherServletCondition.class)
    @ConditionalOnClass(ServletRegistration.class)
    // 绑定配置文件的属性，从配置文件中获取配置项
    @EnableConfigurationProperties(WebMvcProperties.class)
    protected static class DispatcherServletConfiguration {

        // 给容器注册一个 DispatcherServlet，起名字为 dispatcherServlet
        @Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
        public DispatcherServlet dispatcherServlet(WebMvcProperties webMvcProperties) {
            // 新建一个 DispatcherServlet 设置相关属性
            DispatcherServlet dispatcherServlet = new DispatcherServlet();
            // spring.mvc 中的配置项获取注入，没有就填充默认值
            dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());
            // .....
            // 返回该对象注册到容器内
            return dispatcherServlet;
        }

        @Bean
        // 容器中有这个类型组件才进行装配
        @ConditionalOnBean(MultipartResolver.class)
        // 容器中没有这个名字 multipartResolver 的组件
        @ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
        // 方法名就是 BeanName
        public MultipartResolver multipartResolver(MultipartResolver resolver) {
            // 给 @Bean 标注的方法传入了对象参数，这个参数就会从容器中找，因为用户自定义了该类型，以用户配置的优先
            // 但是名字不符合规范，所以获取到该 Bean 并返回到容器一个规范的名称：multipartResolver
            return resolver;
        }
    }
}

// 将配置文件中的 spring.mvc 前缀的属性与该类绑定
@ConfigurationProperties(prefix = "spring.mvc")
public class WebMvcProperties { }
```

事件监听

SpringBoot 在项目启动时，会对几个监听器进行回调，可以实现监听器接口，在项目启动时完成一些操作

`ApplicationContextInitializer`、`SpringApplicationRunListener`、`CommandLineRunner`、`ApplicationRunner`

- `MyApplicationRunner`

自定义监听器的启动时机：`MyApplicationRunner` 和 `MyCommandLineRunner` 都是当项目启动后执行，使用 `@Component` 放入容器即可使用

```
//当项目启动后执行run方法
@Component
public class MyApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("ApplicationRunner...run");
        System.out.println(Arrays.asList(args.getSourceArgs())); //properties配置信息
    }
}
```

- o MyCommandLineRunner

```
@Component
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("CommandLineRunner...run");
        System.out.println(Arrays.asList(args));
    }
}
```

- o MyApplicationContextInitializer 的启用要在 resource 文件夹下添加 META-INF/spring.factories

```
org.springframework.context.ApplicationContextInitializer=\ncom.example.springbootlistener.listener.MyApplicationContextInitializer
```

```
@Component
public class MyApplicationContextInitializer implements ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {
        System.out.println("ApplicationContextInitializer....initialize");
    }
}
```

- o MySpringApplicationRunListener 的使用要添加构造器

```
public class MySpringApplicationRunListener implements SpringApplicationRunListener {
    //构造器
    public MySpringApplicationRunListener(SpringApplication sa, String[] args) {

        @Override
        public void starting() {
            System.out.println("starting...项目启动中"); //输出SPRING之前
        }

        @Override
        public void environmentPrepared(ConfigurableEnvironment environment) {
            System.out.println("environmentPrepared...环境对象开始准备");
        }

        @Override
        public void contextPrepared(ConfigurableApplicationContext context) {
            System.out.println("contextPrepared...上下文对象开始准备");
        }

        @Override
        public void contextLoaded(ConfigurableApplicationContext context) {
            System.out.println("contextLoaded...上下文对象开始加载");
        }

        @Override
        public void started(ConfigurableApplicationContext context) {
            System.out.println("started...上下文对象加载完成");
        }

        @Override
        public void running(ConfigurableApplicationContext context) {
            System.out.println("running...项目启动完成, 开始运行");
        }

        @Override
        public void failed(ConfigurableApplicationContext context, Throwable exception) {
            System.out.println("failed...项目启动失败");
        }
    }
}
```

配置文件

配置方式

文件类型

SpringBoot 是基于约定的，很多配置都有默认值，如果想使用自己的配置替换默认配置，可以使用 application.properties 或者 application.yml (application.yaml) 进行配置

- 默认配置文件名称：application
- 在同一级目录下优先级为：properties > yaml > yaml

例如配置内置 Tomcat 的端口

- properties:

```
server.port=8080
```

- yaml:

```
server: port: 8080
```

- yaml:

```
server: port: 8080
```

加载顺序

所有位置的配置文件都会被加载，互补配置，**高优先级配置内容会覆盖低优先级配置内容**

扫描配置文件的位置按优先级**从高到底**:

- file:./config/：当前项目下的 /config 目录下
- file:./：当前项目的根目录，Project 工程目录
- classpath:/config/：classpath 的 /config 目录
- classpath:/：classpath 的根目录，就是 resources 目录

项目外部配置文件加载顺序：外部配置文件的使用是为了对内部文件的配合

- 命令行：在 package 打包后的 target 目录下，使用该命令

```
java -jar myproject.jar --server.port=9000
```

- 指定配置文件位置

```
java -jar myproject.jar --spring.config.location=e://application.properties
```

- 按优先级从高到底选择配置文件的加载命令

```
java -jar myproject.jar
```

yaml语法

基本语法：

- 大小写敏感
- **数据值前边必须有空格，作为分隔符**
- 使用缩进表示层级关系
- 缩进时不允许使用Tab键，只允许使用空格（各个系统 Tab 对应空格数目可能不同，导致层次混乱）
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- "#" 表示注释，从这个字符一直到行尾，都会被解析器忽略

```
server:  
  port: 8080  
  address: 127.0.0.1
```

数据格式：

- 纯量：单个的、不可再分的值

```
msg1: 'hello \n world' # 单引忽略转义字符  
msg2: "hello \n world" # 双识别转义字符
```

- 对象：键值对集合，Map、Hash

```
person:  
  name: zhangsan  
  age: 20  
# 行内写法  
person: {name: zhangsan}
```

注意：不建议使用JSON，应该使用yaml语法

- 数组：一组按次序排列的值，List、Array

```
address:  
  - beijing  
  - shanghai  
# 行内写法  
address: [beijing,shanghai]
```

```
allPerson #List<Person>  
  - {name:lisi, age:18}  
  - {name:wangwu, age:20}  
# 行内写法  
allPerson: [{name:lisi, age:18}, {name:wangwu, age:20}]
```

- 参数引用：

```
name: lisi  
person:  
  name: ${name} # 引用上边定义的name值
```

获取配置

三种获取配置文件的方式：

- 注解 @Value

```
@RestController  
public class HelloController {  
  @Value("${name}")  
  private String name;  
  
  @Value("${person.name}")  
  private String name2;  
  
  @Value("${address[0]}")  
  private String address1;  
  
  @Value("${msg1}")  
  private String msg1;  
  
  @Value("${msg2}")  
  private String msg2;  
  
  @RequestMapping("/hello")  
  public String hello(){  
    System.out.println("所有的数据");  
    return " hello Spring Boot !";  
  }  
}
```

- Environment 对象

```

@Autowired
private Environment env;

@RequestMapping("/hello")
public String hello() {
    System.out.println(env.getProperty("person.name"));
    System.out.println(env.getProperty("address[0]"));
    return " hello Spring Boot !";
}

```

- 注解 @ConfigurationProperties 配合 @Component 使用

注意：参数 prefix 一定要指定

```

@Component //不扫描该组件到容器内，无法完成自动装配
@ConfigurationProperties(prefix = "person")
public class Person {
    private String name;
    private int age;
    private String[] address;
}

```

```

@Autowired
private Person person;

@RequestMapping("/hello")
public String hello() {
    System.out.println(person);
    //Person{name='zhangsan', age=20, address=[beijing, shanghai]}
    return " hello Spring Boot !";
}

```

配置提示

自定义的类和配置文件绑定一般没有提示，添加如下依赖可以使用提示：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>

<!-- 下面插件作用是工程打包时，不将spring-boot-configuration-processor打进包内，让其只在编码的时候有用 -->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-configuration-processor</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>

```

Profile

@Profile：指定组件在哪个环境的情况下才能被注册到容器中，不指定，任何环境下都能注册这个组件

- 加了环境标识的 bean，只有这个环境被激活的时候才能注册到容器中，默认是 default 环境
- 写在配置类上，只有是指定的环境的时候，整个配置类里面的所有配置才能开始生效
- 没有标注环境标识的 bean 在，任何环境下都是加载的

Profile 的配置：

- profile 是用来完成不同环境下，配置动态切换功能

- **profile 配置方式**: 多 profile 文件方式, 提供多个配置文件, 每个代表一种环境
 - application-dev.properties/yml 开发环境
 - application-test.properties/yml 测试环境
 - application-pro.properties/yml 生产环境
- yaml 多文档方式: 在 yaml 中使用 --- 分隔不同配置

```
---
server:
  port: 8081
spring:
  profiles:dev
---

server:
  port: 8082
spring:
  profiles:test
---

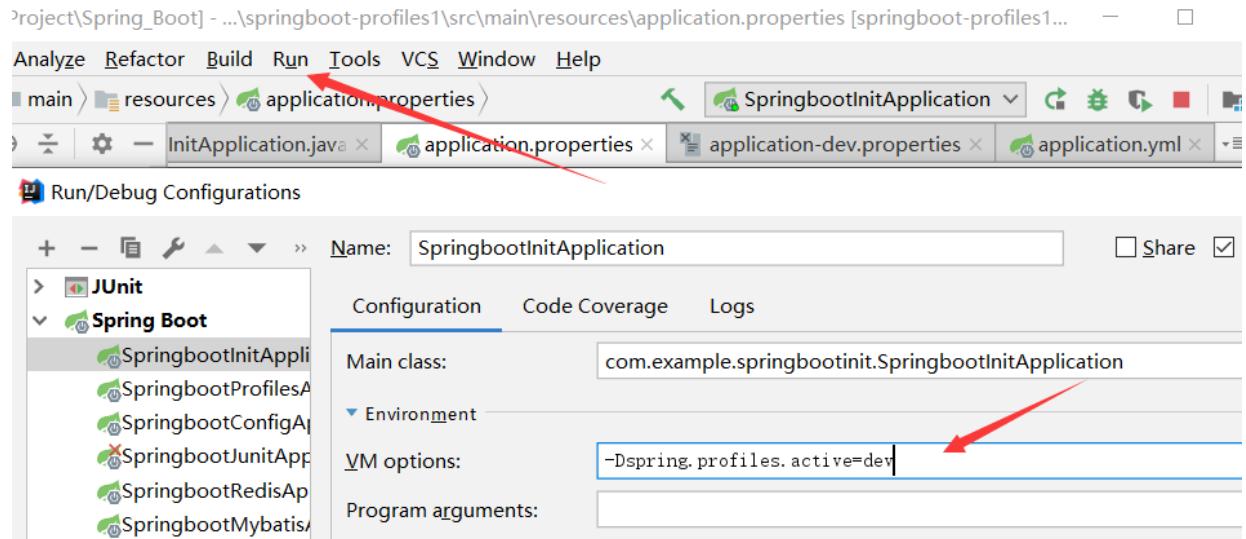
server:
  port: 8083
spring:
  profiles:pro
---
```

◦ profile 激活方式

- 配置文件: 在配置文件中配置: spring.profiles.active=dev

spring.profiles.active=dev

- 虚拟机参数: 在 VM options 指定: -Dspring.profiles.active=dev



- 命令行参数: java -jar xxx.jar --spring.profiles.active=dev

在 Program arguments 里输入, 也可以先 package

Web开发

功能支持

SpringBoot 自动配置了很多约定, 大多场景都无需自定义配置

- 内容协商视图解析器 ContentNegotiatingViewResolver 和 BeanName 视图解析器 BeanNameViewResolver
- 支持静态资源 (包括 webjars) 和静态 index.html 页支持
- 自动注册相关类: Converter、GenericConverter、Formatter
- 内容协商处理器: HttpMessageConverters
- 国际化: MessageCodesResolver

开发规范:

- 使用 @Configuration + WebMvcConfigurer 自定义规则, 不使用 @EnableWebMvc 注解
- 声明 WebMvcRegistrations 的实现类改变默认底层组件
- 使用 @EnableWebMvc + @Configuration + DelegatingWebMvcConfiguration 全面接管 SpringMVC

静态资源

访问规则

默认的静态资源路径是 classpath 下的，优先级由高到低为：/META-INF/resources、/resources、/static、/public 的包内，/ 表示当前项目的根路径

静态映射 /**，表示请求 / + 静态资源名 就直接去默认的资源路径寻找请求的资源

处理原理：静请求去寻找 Controller 处理，不能处理的请求就会交给静态资源处理器，静态资源也找不到就响应 404 页面

- 修改默认资源路径：

```
spring:  
  web:  
    resources:  
      static-locations:: [classpath:/haha/]
```

- 修改静态资源访问前缀，默认是 /**：

```
spring:  
  mvc:  
    static-path-pattern: /resources/**
```

访问 URL：<http://localhost:8080/resources/> + 静态资源名，将所有资源重定位到 /resources/

- webjar 访问资源：

```
<dependency>  
  <groupId>org.webjars</groupId>  
  <artifactId>jquery</artifactId>  
  <version>3.5.1</version>  
</dependency>
```

访问地址：<http://localhost:8080/webjars/jquery/3.5.1/jquery.js>，后面地址要按照依赖里面的包路径

欢迎页面

静态资源路径下 index.html 默认作为欢迎页面，访问 <http://localhost:8080> 出现该页面，使用 welcome page 功能不能修改前缀

网页标签上的小图标可以自定义规则，把资源重命名为 favicon.ico 放在静态资源目录下即可

源码分析

SpringMVC 功能的自动配置类 WebMvcAutoConfiguration：

```
public class WebMvcAutoConfiguration {  
  //当前项目的根路径  
  private static final String SERVLET_LOCATION = "/";  
}
```

- 内部类 WebMvcAutoConfigurationAdapter：

```
@Import(EnableWebMvcConfiguration.class)  
//绑定 spring.mvc、spring.web、spring.resources 相关的配置属性  
@EnableConfigurationProperties({ webMvcProperties.class, ResourceProperties.class, WebProperties.class })  
@Order(0)  
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer, ServletContextAware {  
  //有参构造器所有参数的值都会从容器中确定  
  public WebMvcAutoConfigurationAdapter(/*参数*/){  
    this.resourceProperties = resourceProperties.hasBeenCustomized() ? resourceProperties  
      : webProperties.getResources();  
    this.mvcProperties = mvcProperties;  
    this.beanFactory = beanFactory;  
    this.messageConvertersProvider = messageConvertersProvider;  
    this.resourceHandlerRegistrationCustomizer = resourceHandlerRegistrationCustomizerProvider.getIfAvailable();  
    this.dispatcherServletPath = dispatcherServletPath;  
    this.servletRegistrations = servletRegistrations;  
    this.mvcProperties.checkConfiguration();  
  }  
}
```

- ResourceProperties resourceProperties: 获取和 spring.resources 绑定的所有的值的对象
- WebMvcProperties mvcProperties: 获取和 spring.mvc 绑定的所有的值的对象
- ListableBeanFactory beanFactory: Spring 的 beanFactory
- HttpMessageConverters: 找到所有的 HttpMessageConverters
- ResourceHandlerRegistrationCustomizer: 找到 资源处理器的自定义器。
- DispatcherServletPath: 项目路径
- ServletRegistrationBean: 给应用注册 Servlet、Filter
- WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter.addResourceHandler(): 两种静态资源映射规则

```

public void addResourceHandlers(ResourceHandlerRegistry registry) {
    //配置文件设置 spring.resources.add-mappings: false, 禁用所有静态资源
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");//被禁用
        return;
    }
    //注册webjars静态资源的映射          映射          路径
    addResourceHandler(registry, "/webjars/**", "classpath:/META-INF/resources/webjars/");
    //注册静态资源路径的映射规则      默认映射 staticPathPattern = "/**"
    addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
        //staticLocations = CLASSPATH_RESOURCE_LOCATIONS
        registration.addResourceLocations(this.resourceProperties.getStaticLocations());
        if (this.servletContext != null) {
            ServletContextResource resource = new ServletContextResource(this.servletContext, SERVLET_LOCATION);
            registration.addResourceLocations(resource);
        }
    });
}

@ConfigurationProperties("spring.web")
public class WebProperties {
    public static class Resources {
        //默认资源路径, 优先级从高到底
        static final String[] CLASSPATH_RESOURCE_LOCATIONS = { "classpath:/META-INF/resources/",
                                                               "classpath:/resources/",
                                                               "classpath:/static/", "classpath:/public/" };
        private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;
        //可以进行规则重写
        public void setStaticLocations(String[] staticLocations) {
            this.staticLocations = appendSlashIfNecessary(staticLocations);
            this.customized = true;
        }
    }
}

```

- WebMvcAutoConfiguration.EnableWebMvcConfiguration.welcomePageHandlerMapping(): 欢迎页

```

//spring.web 属性
@EnableConfigurationProperties(WebProperties.class)
public static class EnableWebMvcConfiguration {
    @Bean
    public WelcomePageHandlerMapping welcomePageHandlerMapping(/*参数*/) {
        WelcomePageHandlerMapping welcomePageHandlerMapping = new WelcomePageHandlerMapping(
            new TemplateAvailabilityProviders(applicationContext),
            applicationContext, getWelcomePage(),
            //staticPathPattern = "**"
            this.mvcProperties.getStaticPathPattern());
        return welcomePageHandlerMapping;
    }
}
welcomePageHandlerMapping(/*参数*/) {
    //所以限制 staticPathPattern 必须为 ** 才能启用该功能
    if (welcomePage != null && "**".equals(staticPathPattern)) {
        logger.info("Adding welcome page: " + welcomePage);
        //重定向
        setRootViewName("forward:index.html");
    }
    else if (welcomeTemplateExists(templateAvailabilityProviders, applicationContext)) {
        logger.info("Adding welcome page template: index");
        setRootViewName("index");
    }
}

```

WelcomePageHandlerMapping, 访问 / 能访问到 index.html

Rest映射

开启 Rest 功能

```
spring:  
  mvc:  
    hiddenmethod:  
      filter:  
        enabled: true  #开启页面表单的Rest功能
```

源码分析，注入了 HiddenHttpMethodFilte 解析 Rest 风格的访问：

```
public class WebMvcAutoConfiguration {  
    @Bean  
    @ConditionalOnMissingBean(HiddenHttpMethodFilter.class)  
    @ConditionalOnProperty(prefix = "spring.mvc.hiddenmethod.filter", name = "enabled")  
    public OrderedHiddenHttpMethodFilter hiddenHttpMethodFilter() {  
        return new OrderedHiddenHttpMethodFilter();  
    }  
}
```

详细源码解析：SpringMVC → 基本操作 → Restful → 识别原理

Web 部分源码详解：SpringMVC → 运行原理

内嵌容器

SpringBoot 嵌入式 Servlet 容器，默认支持的 WebServe：Tomcat、Jetty、Undertow

配置方式：

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
    <exclusions>  
        <exclusion> <!--必须要把内嵌的 Tomcat 容器--&gt;<br/>            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-tomcat</artifactId>  
        </exclusion>  
    </exclusions>  
  </dependency>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-jetty</artifactId>  
  </dependency>
```

Web 应用启动，SpringBoot 导入 Web 场景包 tomcat，创建一个 Web 版的 IOC 容器：

- `SpringApplication.run(BootApplication.class, args)`：应用启动
- `ConfigurableApplicationContext.run()`：
 - `context = createApplicationContext()`：创建容器
 - `applicationContextFactory = ApplicationContextFactory.DEFAULT`
 - ```
ApplicationContextFactory DEFAULT = (webApplicationType) -> {
 try {
 switch (webApplicationType) {
 case SERVLET:
 // Servlet 容器，继承自 ServletWebServerApplicationContext
 return new AnnotationConfigServletWebServerApplicationContext();
 case REACTIVE:
 // 响应式编程
 return new AnnotationConfigReactiveWebServerApplicationContext();
 default:
 // 普通 Spring 容器
 return new AnnotationConfigApplicationContext();
 }
 } catch (Exception ex) {
 throw new IllegalStateException();
 }
}
```
  - `applicationContextFactory.create(this.webApplicationType)`：根据应用类型创建容器
  - `refreshContext(context)`：容器启动刷新

内嵌容器工作流程：

- Spring 容器启动逻辑中，在实例化非懒加载的单例 Bean 之前有一个方法 `onRefresh()`，留给子类去扩展，Web 容器就是重写这个方法创建 WebServer

```

protected void onRefresh() {
 //省略....
 createWebServer();
}

private void createWebServer() {
 ServletWebServerFactory factory = getWebServerFactory();
 this.webServer = factory.getWebServer(getSelfInitializer());
 createWebServer.end();
}

```

获取 WebServer 工厂 `ServletWebServerFactory`, 并且获取的数量不等于 1 会报错, Spring 底层有三种:

`TomcatServletWebServerFactory`、`JettyServletWebServerFactory`、`UndertowServletWebServerFactory`

- 自动配置类 `ServletWebServerFactoryAutoConfiguration` 导入了 `ServletWebServerFactoryConfiguration` (配置类), 根据条件装配判断系统中到底导入了哪个 Web 服务器的包, 创建出服务器并启动
- 默认是 web-starter 导入 tomcat 包, 容器中就有 `TomcatServletWebServerFactory`, 创建出 Tomcat 服务器并启动,

```

public TomcatWebServer(Tomcat tomcat, boolean autoStart, Shutdown shutdown) {
 // 初始化
 initialize();
}

```

初始化方法 `initialize` 中有启动方法: `this.tomcat.start()`

## 自定义

### 定制规则

```

@Configuration
public class MyWebMvcConfigurer implements WebMvcConfigurer {
 @Bean
 public WebMvcConfigurer webMvcConfigurer() {
 return new WebMvcConfigurer() {
 //进行一些方法重写, 来实现自定义的规则
 //比如添加一些解析器和拦截器, 就是对原始容器功能的增加
 };
 }
 //也可以不加 @Bean, 直接从这里重写方法进行功能增加
}

```

## 定制容器

`@EnableWebMvc`: 全面接管 SpringMVC, 所有规则全部自己重新配置

- `@EnableWebMvc + WebMvcConfigurer + @Bean` 全面接管SpringMVC
- `@Import(DelegatingWebMvcConfiguration.class)`, 该类继承 `WebMvcConfigurationSupport`, 自动配置了一些非常底层的组件, 只能保证 SpringMVC 最基本的使用

原理: 自动配置类 `WebMvcAutoConfiguration` 里面的配置要能生效, `WebMvcConfigurationSupport` 类不能被加载, 所以 `@EnableWebMvc` 导致配置类失效, 从而接管了 SpringMVC

```

@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
public class WebMvcAutoConfiguration {}

```

注意: 一般不适用此注解

## 数据访问

### JDBC

## 基本使用

导入 starter:

```
<!--导入 JDBC 场景-->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<!--导入 MySQL 驱动-->
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <!--版本对应你的 MySQL 版本<version>5.1.49</version>-->
</dependency>
```

单独导入 MySQL 驱动是因为不确定用户使用的什么数据库

配置文件:

```
spring:
 datasource:
 url: jdbc:mysql://192.168.0.107:3306/db1?useSSL=false # 不加 useSSL 会警告
 username: root
 password: 123456
 driver-class-name: com.mysql.jdbc.Driver
```

测试文件:

```
@Slf4j
@SpringBootTest
class Boot05WebAdminApplicationTests {

 @Autowired
 JdbcTemplate jdbcTemplate;

 @Test
 void contextLoads() {
 Long res = jdbcTemplate.queryForObject("select count(*) from account_tbl", Long.class);
 log.info("记录总数: {}", res);
 }
}
```

## 自动配置

DataSourceAutoConfiguration: 数据源的自动配置

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(dataSourceProperties.class)
public class DataSourceAutoConfiguration {

 @Conditional(PooledDataSourceCondition.class)
 @ConditionalOnMissingBean({ dataSource.class, XADataSource.class })
 @Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
 DataSourceConfiguration.Dhcp2.class, DataSourceConfiguration.OracleUcp.class })
 protected static class PooledDataSourceConfiguration {}

 // 配置项
 @ConfigurationProperties(prefix = "spring.datasource")
 public class DataSourceProperties implements BeanClassLoaderAware, InitializingBean {}
```

- 底层默认配置好的连接池是: **HikariDataSource**
- 数据库连接池的配置, 是容器中没有 **DataSource** 才自动配置的
- 修改数据源相关的配置: **spring.datasource**

相关配置:

- **DataSourceTransactionManagerAutoConfiguration**: 事务管理器的自动配置
- **JdbcTemplateAutoConfiguration**: **JdbcTemplate** 的自动配置
  - 可以修改这个配置项 **@ConfigurationProperties(prefix = "spring.jdbc")** 来修改 **JdbcTemplate**
  - **@AutoConfigureAfter(DataSourceAutoConfiguration.class)**: 在 **DataSource** 装配后装配
- **JndiDataSourceAutoConfiguration**: **jndi** 的自动配置
- **XADataSourceAutoConfiguration**: 分布式事务相关

## Druid

导入坐标:

```
<dependency>
 <groupId>com.alibaba</groupId>
 <artifactId>druid-spring-boot-starter</artifactId>
 <version>1.1.17</version>
</dependency>
```

```
@Configuration
@ConditionalOnClass(DruidDataSource.class)
@AutoConfigureBefore(DataSourceAutoConfiguration.class)
@EnableConfigurationProperties({DruidStatProperties.class, DataSourceProperties.class})
@Import({DruidSpringAopConfiguration.class,
 DruidStatViewServletConfiguration.class,
 DruidWebStatFilterConfiguration.class,
 DruidFilterConfiguration.class})
public class DruidDataSourceAutoConfigure {}
```

自动配置:

- 扩展配置项 `spring.datasource.druid`
- `DruidSpringAopConfiguration`: 监控 SpringBean, 配置项为 `spring.datasource.druid.aop-patterns`
- `DruidStatViewServletConfiguration`: 监控页的配置项为 `spring.datasource.druid.stat-view-servlet`, 默认开启
- `DruidWebStatFilterConfiguration`: Web 监控配置项为 `spring.datasource.druid.web-stat-filter`, 默认开启
- `DruidFilterConfiguration`: 所有 Druid 自己 filter 的配置

配置示例:

```
spring:
 datasource:
 url: jdbc:mysql://localhost:3306/db_account
 username: root
 password: 123456
 driver-class-name: com.mysql.jdbc.Driver

 druid:
 aop-patterns: com.atguigu.admin.* #监控SpringBean
 filters: stat,wall # 底层开启功能, stat (sql监控), wall (防火墙)

 stat-view-servlet: # 配置监控页功能
 enabled: true
 login-username: admin #项目启动访问: http://localhost:8080/druid, 账号和密码是 admin
 login-password: admin
 resetEnable: false

 web-stat-filter: # 监控web
 enabled: true
 urlPattern: /*
 exclusions: '*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*'

 filter:
 stat: # 对上面filters里面的stat的详细配置
 slow-sql-millis: 1000
 logSlowSql: true
 enabled: true
 wall:
 enabled: true
 config:
 drop-table-allow: false
```

配置示例: <https://github.com/alibaba/druid/tree/master/druid-spring-boot-starter>

配置项列表: <https://github.com/alibaba/druid/wiki/DruidDataSource%E9%85%8D%E7%BD%AE%E5%B1%9E%E6%80%A7%E5%88%97%E8%A1%A8>

# MyBatis

## 基本使用

导入坐标:

```
<dependency>
 <groupId>org.mybatis.spring.boot</groupId>
 <artifactId>mybatis-spring-boot-starter</artifactId>
 <version>2.1.4</version>
</dependency>
```

- 编写 MyBatis 相关配置: application.yml

```
配置mybatis规则
mybatis:
config-location: classpath:mybatis/mybatis-config.xml 建议不写
mapper-locations: classpath:mybatis/mapper/*.xml
configuration:
 map-underscore-to-camel-case: true

#可以不写全局配置文件，所有全局配置文件的配置都放在 configuration 配置项中即可
```

- 定义表和实体类

```
public class User {
 private int id;
 private String username;
 private String password;
}
```

- 编写 dao 和 mapper 文件/纯注解开发

dao: @Mapper 注解必须加, 使用自动装配的 package, 否则在启动类指定 @MapperScan() 扫描路径 (不建议)

```
@Mapper //必须加Mapper
@Repository
public interface UserXmlMapper {
 public List<User> findAll();
}
```

mapper.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.seazean.springbootmybatis.mapper.UserXmlMapper">
 <select id="findAll" resultType="user">
 select * from t_user
 </select>
</mapper>
```

- 纯注解开发

```
@Mapper
@Repository
public interface UserMapper {
 @Select("select * from t_user")
 public List<User> findAll();
}
```

## 自动配置

MybatisAutoConfiguration:

```
@EnableConfigurationProperties(MybatisProperties.class) //MyBatis配置项绑定类。
@AutoConfigureAfter({ DataSourceAutoConfiguration.class, MybatisLanguageDriverAutoConfiguration.class })
public class MybatisAutoConfiguration {
 @Bean
 @ConditionalOnMissingBean
 public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
 SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
 return factory.getObject();
 }

 @org.springframework.context.annotation.Configuration
```

```

@Import(AutoConfiguredMapperScannerRegistrar.class)
@ConditionalOnMissingBean({ MapperFactoryBean.class, MapperScannerConfigurer.class })
public static class MapperScannerRegistrarNotFoundConfiguration implements InitializingBean {}

@ConfigurationProperties(prefix = "mybatis")
public class MybatisProperties {}

```

- 配置文件: mybatis
  - 自动配置了 SqlSessionFactory
  - 导入 AutoConfiguredMapperScannerRegistrar 实现 @Mapper 的扫描
- 

## MyBatis-Plus

```

<dependency>
 <groupId>com.baomidou</groupId>
 <artifactId>mybatis-plus-boot-starter</artifactId>
 <version>3.4.1</version>
</dependency>

```

自动配置类: MybatisPlusAutoConfiguration

只需要 Mapper 继承 **BaseMapper** 就可以拥有 CRUD 功能

---

## Redis

### 基本使用

```

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

```

- 配置 redis 相关属性

```

spring:
 redis:
 host: 127.0.0.1 # redis 的主机 ip
 port: 6379

```

- 注入 RedisTemplate 模板

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringbootRedisApplicationTests {
 @Autowired
 private RedisTemplate redisTemplate;

 @Test
 public void testSet() {
 // 存入数据
 redisTemplate.boundValueOps("name").set("zhangsan");
 }
 @Test
 public void testGet() {
 // 获取数据
 Object name = redisTemplate.boundValueOps("name").get();
 System.out.println(name);
 }
}

```

---

### 自动配置

RedisAutoConfiguration 自动配置类

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RedisOperations.class)

```

```

@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {
 @Bean
 @ConditionalOnMissingBean(name = "redisTemplate")
 @ConditionalOnSingleCandidate(RedisConnectionFactory.class)
 public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
 RedisTemplate<Object, Object> template = new RedisTemplate<>();
 template.setConnectionFactory(redisConnectionFactory);
 return template;
 }

 @Bean
 @ConditionalOnMissingBean
 @ConditionalOnSingleCandidate(RedisConnectionFactory.class)
 public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory redisConnectionFactory) {
 StringRedisTemplate template = new StringRedisTemplate();
 template.setConnectionFactory(redisConnectionFactory);
 return template;
 }
}

```

- 配置项: `spring.redis`
- 自动导入了连接工厂配置类: `LettuceConnectionConfiguration`、`JedisConnectionConfiguration`
- 自动注入了模板类: `RedisTemplate<Object, Object>`、`StringRedisTemplate`, `k v`都是 `String` 类型
- 使用 `@Autowired` 注入模板类就可以操作 `redis`

## 单元测试

### Junit5

Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库, 由三个不同的子模块组成:

- JUnit Platform: 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: 提供了 JUnit5 的新的编程模型, 是 JUnit5 新特性的核心, 内部包含了一个测试引擎, 用于在 Junit Platform 上运行
- JUnit Vintage: JUnit Vintage 提供了兼容 JUnit4.x、Junit3.x 的测试引擎

注意: SpringBoot 2.4 以上版本移除了默认对 Vintage 的依赖, 如果需要兼容 Junit4 需要自行引入

```

@SpringBootTest
class Boot05WebAdminApplicationTests {
 @Test
 void contextLoads() { }
}

```

## 常用注解

JUnit5 的注解如下:

- `@Test`: 表示方法是测试方法, 但是与 JUnit4 的 `@Test` 不同, 它的职责非常单一不能声明任何属性, 拓展的测试将会由 Jupiter 提供额外测试, 包含 `org.junit.jupiter.api.Test`
- `@ParameterizedTest`: 表示方法是参数化测试
- `@RepeatedTest`: 表示方法可重复执行
- `@DisplayName`: 为测试类或者测试方法设置展示名称
- `@BeforeEach`: 表示在每个单元测试之前执行
- `@AfterEach`: 表示在每个单元测试之后执行
- `@BeforeAll`: 表示在所有单元测试之前执行
- `@AfterAll`: 表示在所有单元测试之后执行
- `@Tag`: 表示单元测试类别, 类似于 JUnit4 中的 `@Categories`
- `@Disabled`: 表示测试类或测试方法不执行, 类似于 JUnit4 中的 `@Ignore`
- `@Timeout`: 表示测试方法运行如果超过了指定时间将会返回错误
- `@ExtendWith`: 为测试类或测试方法提供扩展类引用

## 断言机制

### 简单断言

断言 (assertions) 是测试方法中的核心，用来对测试需要满足的条件进行验证，断言方法都是 org.junit.jupiter.api.Assertions 的静态方法。用来对单个值进行简单的验证：

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null

```
@Test
@DisplayName("simple assertion")
public void simple() {
 assertEquals(3, 1 + 2, "simple math");
 assertNull(null);
 assertNotNull(new Object());
}
```

### 数组断言

通过 assertArrayEquals 方法来判断两个对象或原始类型的数组是否相等

```
@Test
@DisplayName("array assertion")
public void array() {
 assertArrayEquals(new int[]{1, 2}, new int[] {1, 2});
}
```

### 组合断言

assertAll 方法接受多个 org.junit.jupiter.api.Executable 函数式接口的实例作为验证的断言，可以通过 lambda 表达式提供这些断言

```
@Test
@DisplayName("assert all")
public void all() {
 assertAll("Math",
 () -> assertEquals(2, 1 + 1),
 () -> assertTrue(1 > 0)
);
}
```

### 异常断言

Assertions.assertThrows(), 配合函数式编程就可以进行使用

```
@Test
 @DisplayName("异常测试")
 public void exceptionTest() {
 ArithmeticException exception = Assertions.assertThrows(
 //抛出断言异常
 ArithmeticException.class, () -> System.out.println(1 / 0)
);
 }
```

## 超时断言

Assertions.assertTimeout() 为测试方法设置了超时时间

```
@Test
 @DisplayName("超时测试")
 public void timeoutTest() {
 //如果测试方法时间超过1s将会异常
 Assertions.assertTimeout(Duration.ofMillis(1000), () -> Thread.sleep(500));
 }
```

## 快速失败

通过 fail 方法直接使得测试失败

```
@Test
 @DisplayName("fail")
 public void shouldFail() {
 fail("This should fail");
 }
```

## 前置条件

JUnit 5 中的前置条件 (assumptions) 类似于断言，不同之处在于不满足的断言会使得测试方法失败，而不满足的前置条件只会使得测试方法的执行终止，前置条件可以看成是测试方法执行的前提，当该前提不满足时，就没有继续执行的必要

```
@DisplayName("测试前置条件")
 @Test
 void testAssumptions(){
 Assumptions.assumeTrue(false, "结果不是true");
 System.out.println("111111");
 }
```

## 嵌套测试

JUnit 5 可以通过 Java 中的内部类和 @Nested 注解实现嵌套测试，从而可以更好的把相关的测试方法组织在一起，在内部类中可以使用 @BeforeEach 和 @AfterEach 注解，而且嵌套的层次没有限制

```
@DisplayName("A stack")
 class TestingStackDemo {

 Stack<Object> stack;

 @Test
 @DisplayName("is instantiated with new Stack()")
 void isInstantiatedWithNew() {
 assertNull(stack)
 }

 @Nested
```

```

 @DisplayName("when new")
 class WhenNew {
 @BeforeEach
 void createNewStack() {
 stack = new Stack<>();
 }

 @Test
 @DisplayName("is empty")
 void isEmpty() {
 assertTrue(stack.isEmpty());
 }

 @Test
 @DisplayName("throws EmptyStackException when popped")
 void throwsExceptionWhenPopped() {
 assertThrows(EmptyStackException.class, stack::pop);
 }
 }
}

```

## 参数测试

参数化测试是 JUnit5 很重要的一个新特性，它使得用不同的参数多次运行测试成为了可能

利用@ValueSource等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

- @ValueSource：为参数化测试指定入参来源，支持八大基础类以及 String 类型、Class 类型
- @NullSource：表示为参数化测试提供一个 null 的入参
- @EnumSource：表示为参数化测试提供一个枚举入参
- @CsvFileSource：表示读取指定 CSV 文件内容作为参数化测试入参
- @MethodSource：表示读取指定方法的返回值作为参数化测试入参（注意方法返回需要是一个流）

## 指标监控

### Actuator

每一个微服务在云上部署以后，都需要对其进行监控、追踪、审计、控制等，SpringBoot 抽取了 Actuator 场景，使得每个微服务快速引用即可获得生产级别的应用监控、审计等功能

```

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

```

暴露所有监控信息为 HTTP：

```

management:
 endpoints:
 enabled-by-default: true #暴露所有端点信息
 web:
 exposure:
 include: '*' #以web方式暴露

```

访问 [http://localhost:8080/actuator/\[beans/health/metrics/\]](http://localhost:8080/actuator/[beans/health/metrics/])

可视化界面：<https://github.com/codecentric/spring-boot-admin>

## Endpoint

默认所有的 Endpoint 除过 shutdown 都是开启的

```
management:
 endpoints:
 enabled-by-default: false #禁用所有的
 endpoint: #手动开启一部分
 beans:
 enabled: true
 health:
 enabled: true
```

端点:

ID	描述
<code>auditevents</code>	暴露当前应用程序的审核事件信息。需要一个 <code>AuditEventRepository</code> 组件
<code>beans</code>	显示应用程序中所有 Spring Bean 的完整列表
<code>caches</code>	暴露可用的缓存
<code>conditions</code>	显示自动配置的所有条件信息，包括匹配或不匹配的原因
<code>configprops</code>	显示所有 <code>@ConfigurationProperties</code>
<code>env</code>	暴露 Spring 的属性 <code>ConfigurableEnvironment</code>
<code>flyway</code>	显示已应用的所有 Flyway 数据库迁移。需要一个或多个 Flyway 组件。
<code>health</code>	显示应用程序运行状况信息
<code>httptrace</code>	显示 HTTP 跟踪信息，默认情况下 100 个 HTTP 请求-响应需要一个 <code>HttpTraceRepository</code> 组件
<code>info</code>	显示应用程序信息
<code>integrationgraph</code>	显示 Spring integrationgraph，需要依赖 <code>spring-integration-core</code>
<code>loggers</code>	显示和修改应用程序中日志的配置
<code>liquibase</code>	显示已应用的所有 Liquibase 数据库迁移，需要一个或多个 Liquibase 组件
<code>metrics</code>	显示当前应用程序的指标信息。
<code>mappings</code>	显示所有 <code>@RequestMapping</code> 路径列表
<code>scheduledtasks</code>	显示应用程序中的计划任务
<code>sessions</code>	允许从 Spring Session 支持的会话存储中检索和删除用户会话，需要使用 Spring Session 的基于 Servlet 的 Web 应用程序
<code>shutdown</code>	使应用程序正常关闭，默认禁用
<code>startup</code>	显示由 <code>ApplicationStartup</code> 收集的启动步骤数据。需要使用 <code>SpringApplication</code> 进行配置 <code>BufferingApplicationStartup</code>
<code>threaddump</code>	执行线程转储

应用程序是 Web 应用程序 (Spring MVC, Spring WebFlux 或 Jersey)，则可以使用以下附加端点:

ID	描述
<code>heapdump</code>	返回 <code>hprof</code> 堆转储文件。
<code>jolokia</code>	通过 HTTP 暴露 JMX bean (需要引入 <code>jolokia</code> , 不适用于 WebFlux)，需要引入依赖 <code>jolokia-core</code>
<code>logfile</code>	返回日志文件的内容 (如果已设置 <code>logging.file.name</code> 或 <code>logging.file.path</code> 属性)，支持使用 HTTP Range 标头来检索部分日志文件的内容。
<code>prometheus</code>	以 Prometheus 服务器可以抓取的格式公开指标，需要依赖 <code>micrometer-registry-prometheus</code>

常用 Endpoint:

- Health: 监控状况
- Metrics: 运行时指标
- Loggers: 日志记录

## 项目部署

SpringBoot 项目开发完毕后，支持两种方式部署到服务器：

- jar 包(官方推荐，默认)
- war 包

#### 更改 pom 文件中的打包方式为 war

- 修改启动类

```
@SpringBootApplication
public class SpringbootDeployApplication extends SpringBootServletInitializer {
 public static void main(String[] args) {
 SpringApplication.run(SpringbootDeployApplication.class, args);
 }

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder b) {
 return b.sources(SpringbootDeployApplication.class);
 }
}
```

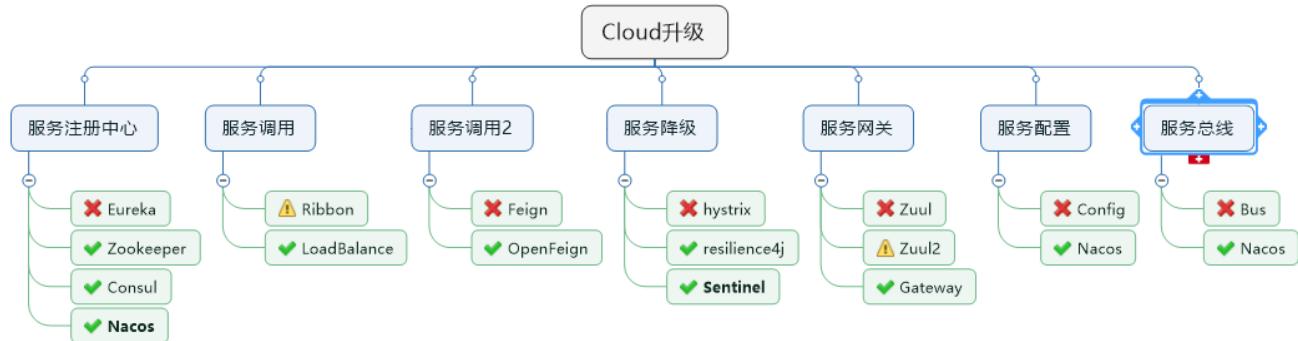
- 指定打包的名称

```
<packaging>war</packaging>
<build>
 <finalName>springboot</finalName>
 <plugins>
 <plugin>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-maven-plugin</artifactId>
 </plugin>
 </plugins>
</build>
```

# Cloud

## 基本介绍

SpringCloud 是分布式微服务的一站式解决方案，是多种微服务落地技术的集合体，俗称微服务全家桶



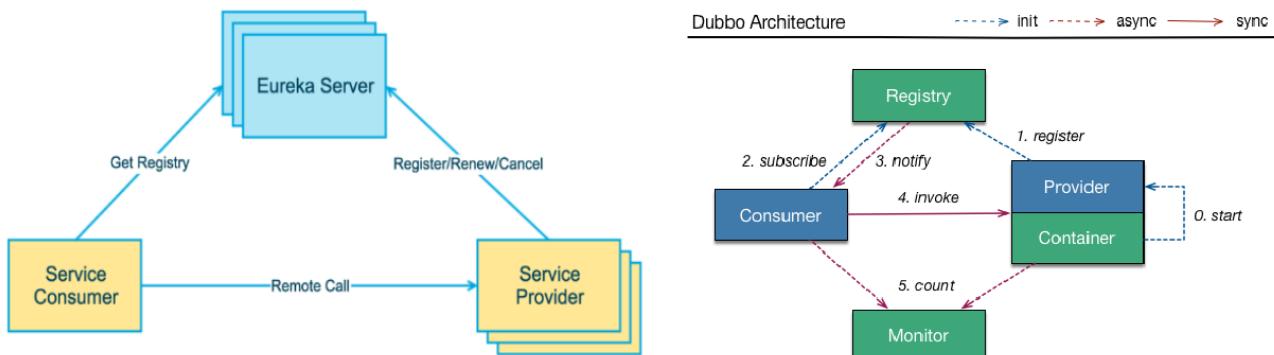
参考文档：[https://www.yuque.com/mrlinxi/pxvr4g/w cwd39](https://www.yuque.com/mrlinxi/pxvr4g/wcwd39)

## 服务注册

# Eureka

## 基本介绍

Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务治理。Eureka 采用了 CS(Client-Server) 的设计架构，Eureka Server 是服务注册中心，系统中的其他微服务使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接



- Eureka Server 提供服务注册服务：各个微服务节点通过配置启动后，会在 EurekaServer 中进行注册，EurekaServer 中的服务注册表中将会存储所有可用服务节点的信息，并且具有可视化界面
- Eureka Client 通过注册中心进行访问：用于简化 Eureka Server 的交互，客户端也具备一个内置的、使用轮询（round-robin）负载算法的负载均衡器。在应用启动后将会向 Eureka Server 发送心跳（默认周期为30秒），如果 Eureka Server 在多个心跳周期内没有接收到某个节点的心跳，将会从服务注册表中把这个服务节点移除（默认 90 秒）

## 服务端

服务器端主启动类增加 @EnableEurekaServer 注解，指定该模块作为 Eureka 注册中心的服务器

构建流程如下：

- 主启动类

```
@SpringBootApplication
@EnableEurekaServer // 表示当前是Eureka的服务注册中心
public class EurekaMain7001 {
 public static void main(String[] args) {
 SpringApplication.run(EurekaMain7001.class, args);
 }
}
```

- 修改 pom 文件

```
1.x: server跟client合在一起
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
2.x: server跟client分开
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- 修改 application.yml 文件

```
server:
 port: 7001

eureka:
 instance:
 hostname: localhost # eureka服务端的实例名称
 client:
 # false表示不向注册中心注册自己。
 register-with-eureka: false
 # false表示自己端就是注册中心，职责就是维护服务实例，并不需要去检索服务
 fetch-registry: false
 service-url:
 # 设置与 Eureka Server 交互的地址查询服务和注册服务都需要依赖这个地址。
```

```
defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

- 浏览器访问 <http://localhost:7001>

## 客户端

### 生产者

服务器端主启动类需要增加 @EnableEurekaClient 注解，表示这是一个 Eureka 客户端，要注册进 EurekaServer 中

- 主启动类：PaymentMain8001

```
@SpringBootApplication
@EnableEurekaClient
public class PaymentMain8001 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentMain8001.class, args);
 }
}
```

- 修改 pom 文件：添加一个 Eureka-Client 依赖

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- 写 yaml 文件

```
server:
 port: 8001

eureka:
 client:
 # 表示将自己注册进EurekaServer默认为true
 register-with-eureka: true
 # 表示可以从Eureka抓取已有的注册信息，默认为true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
 fetch-registry: true
 service-url:
 defaultZone: http://localhost:7001/eureka
 instance:
 instance-id: payment8001 # 只暴露服务名，不带有主机名
 prefer-ip-address: true # 访问信息有 IP 信息提示(鼠标停留在服务名称上时)
```

- 浏览器访问 <http://localhost:7001>

## 消费者

- 主启动类：PaymentMain8001

```
@SpringBootApplication
@EnableEurekaClient
@EnabledDiscoveryClient
public class PaymentMain8001 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentMain8001.class, args);
 }
}
```

- pom 文件同生产者

- 写 yaml 文件

```

server:
port: 80

微服务名称
spring:
application:
name: cloud-order-service
eureka:
client:
register-with-eureka: true
fetch-registry: true
service-url:
defaultZone: http://localhost:7001/eureka

```

- 浏览器访问 <http://localhost:7001>

## DS Replicas

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ORDER-SERVICE	n/a (1)	(1)	UP (1) - localhost:order-service:80
PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - localhost:payment-service:8001

## 集群构建

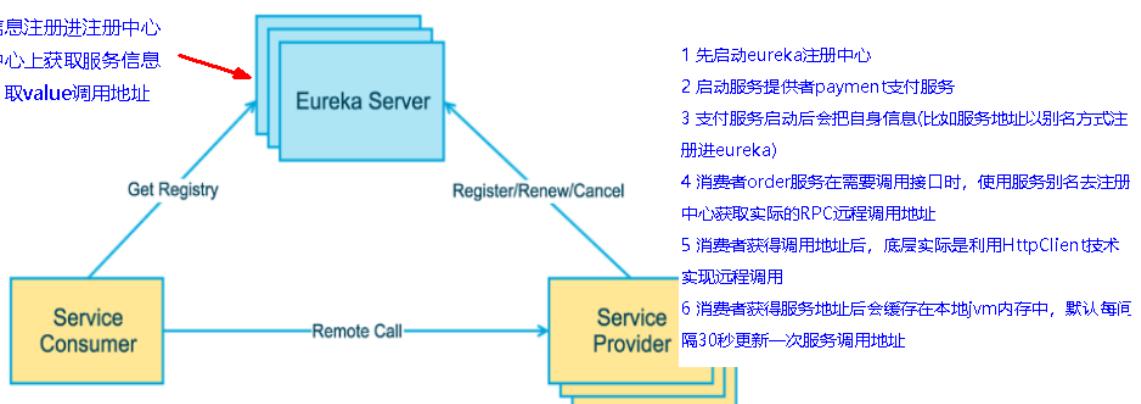
### 服务端

Server 端高可用集群原理：实现负载均衡和故障容错，互相注册，相互守望

服务注册：将服务信息注册进注册中心

服务发现：从注册中心上获取服务信息

实质：存key服务命 取value调用地址



多台 Eureka 服务器，每一台 Eureka 服务器需要有自己的主机名，同时各服务器需要相互注册

- Eureka1:

```

server:
port: 7001

eureka:
instance:
hostname: eureka7001.com
client:
register-with-eureka: false
fetch-registry: false
service-url:
设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址。
单机就是自己
defaultZone: http://eureka7001.com:7001/eureka/
集群指向其他eureka
#defaultZone: http://eureka7002.com:7002/eureka/
写成这样可以直接通过可视化页面跳转到7002
defaultZone: http://eureka7002.com:7002/

```

- Eureka2:

```

server:
port: 7002

eureka:
 instance:
 hostname: eureka7002.com
 client:
 register-with-eureka: false
 fetch-registry: false
 service-url:
 #写成这样可以直接通过可视化页面跳转到7001
 defaultZone: http://eureka7001.com:7001/

```

- 主启动类:

```

@SpringBootApplication
@EnableEurekaServer
public class EurekaMain7002 {
 public static void main(String[] args) {
 SpringApplication.run(EurekaMain7002.class, args);
 }
}

```

- 访问 <http://eureka7001.com:7001> 和 <http://eureka7002.com:7002>:



- RPC 调用: controller.OrderController

```

@RestController
@Slf4j
public class OrderController {
 public static final String PAYMENT_URL = "http://localhost:8001";

 @Autowired
 private RestTemplate restTemplate;

 // CommonResult 是一个公共的返回类型
 @GetMapping("/consumer/payment/get/{id}")
 public CommonResult<Payment> getPayment(@PathVariable("id") long id) {
 // 返回对象为响应体中数据转化成的对象，基本上可以理解为JSON
 return restTemplate.getForObject(PAYMENT_URL + "/payment/get/" + id, CommonResult.class);
 }
}

```

## 生产者

构建 PaymentMain8001 的服务集群

- 主启动类

```

@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
public class PaymentMain8002 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentMain8002.class, args);
 }
}

```

- 写 yaml 文件: 端口修改, 并且 spring.application.name 均为 cloud-payment-service

```

server:
port: 8002

spring:
application:
name: cloud-payment-service

```

```
eureka:
 client:
 # 表示将自己注册进EurekaServer默认为true
 register-with-eureka: true
 # 表示可以从Eureka抓取已有的注册信息，默認為true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
 fetch-registry: true
 service-url:
 defaultZone: http://localhost:7001/eureka
```

## 负载均衡

消费者端的 Controller

```
// public static final String PAYMENT_URL = "http://localhost:8001";
public static final String PAYMENT_URL = "http://localhost:8002";
```

由于已经建立了生产者集群，所以可以进行负载均衡的操作：

- Controller：只修改 PAYMENT\_URL 会报错，因为 CLOUD-PAYMENT-SERVICE 对应多个微服务，需要规则来判断调用哪个端口

```
public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";
```

- 使用 @LoadBlanced 注解赋予 RestTemplate 负载均衡的能力，增加 config.ApplicationConfig 文件：

```
@Configuration
public class ApplicationContextConfig {
 @Bean
 @LoadBalanced
 public RestTemplate getRestTemplate() {
 return new RestTemplate();
 }
}
```

## 服务发现

服务发现：对于注册进 Eureka 里面的微服务，可以通过服务发现来获得该服务的信息

- 主启动类增加注解 @EnableDiscoveryClient：

```
@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
public class PaymentMain8001 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentMain8001.class, args);
 }
}
```

- 修改生产者的 Controller

```
@RestController
@Slf4j
public class PaymentController {
 @Autowired
 private DiscoveryClient discoveryClient;

 @GetMapping(value = "/payment/discovery")
 public Object discovery() {
 List<String> services = discoveryClient.getServices();
 for (String service : services) {
 log.info("**** element:" + service);
 }

 List<ServiceInstance> instances = discoveryClient.getInstances("PAYMENT-SERVICE");
 for (ServiceInstance instance : instances) {
 log.info(instance.getServiceId() + "\t" + instance.getHost() + "\t" + instance.getPort());
 }
 return this.discoveryClient;
 }
}
```

## 自我保护

保护模式用于客户端和 EurekaServer 之间存在网络分区场景下的保护，一旦进入保护模式 EurekaServer 将会尝试保护其服务注册表中的信息，不在删除服务注册表中的数据，属于 CAP 里面的 AP 思想（可用性和分区容错性）

### System Status

Environment	test	Current time	2023-03-14T21:15:08 +0800
Data center	default	Uptime	00:43
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	3

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

如果一定时间内丢失大量该微服务的实例，这时 Eureka 就会开启自我保护机制，不会剔除该服务。因为这个现象可能是因为网络暂时不通，出现了 Eureka 的假死、拥堵、卡顿，客户端恢复后还能正常发送心跳

禁止自我保护：

- Server:

```
eureka:
 server:
 # 关闭自我保护机制，不可用的服务直接删除
 enable-self-preservation: false
 eviction-interval-timer-in-ms: 2000
```

- Client:

```
eureka:
 instance:
 # Eureka客户端向服务端发送心跳的时间间隔默认30秒
 lease-renewal-interval-in-seconds: 1
 # Eureka服务端在收到最后一次心跳后，90s没有收到心跳，剔除服务
 lease-expiration-duration-in-seconds: 2
```

## Consul

### 基本介绍

Consul 是开源的分布式服务发现和配置管理系统，采用 Go 语言开发，官网：<https://developer.hashicorp.com/consul>

- 提供了微服务系统中心的服务治理，配置中心，控制总线等功能
- 基于 Raft 协议，支持健康检查，同时支持 HTTP 和 DNS 协议支持跨数据中心的 WAN 集群
- 提供图形界面

下载 Consul 后，运行指令： `consul -version`

```
D:\Program Files\Java>consul -version
Consul v1.15.1
Revision 7c04b6a0
Build Date 2023-03-07T20:35:33Z
Protocol 2 spoken by default, understands 2 to 3 (....)
```

启动命令：

```
consul agent -dev
```

访问浏览器：<http://localhost:8500/>

中文文档：<https://www.springcloud.cc/spring-cloud-consul.html>

## 基本使用

无需 Server 端代码的编写

生产者:

- 引入 pom 依赖:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

- application.yml:

```
###consul 服务端口号
server:
 port: 8006

spring:
 application:
 name: consul-provider-payment
 ####consul注册中心地址
 cloud:
 consul:
 host: localhost
 port: 8500
 discovery:
 service-name: ${spring.application.name}
```

- 主启动类:

```
@SpringBootApplication
@EnableDiscoveryClient
public class PaymentMain8006 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentMain8006.class, args);
 }
}
```

消费者:

- application.yml:

```
###consul服务端口号
server:
 port: 80

spring:
 application:
 name: cloud-consumer-order
 ####consul注册中心地址
 cloud:
 consul:
 host: localhost
 port: 8500
 discovery:
 #hostname: 127.0.0.1
 service-name: ${spring.application.name}
```

- 主启动类: 同生产者

- 配置类:

```
@Configuration
public class ApplicationContextConfig {
 @Bean
 @LoadBalanced
 public RestTemplate getRestTemplate() {
 return new RestTemplate();
 }
}
```

- 业务类 Controller:

```

@RestController
@Slf4j
public class OrderConsulController {
 public static final String INVOKE_URL = "http://cloud-provider-pament";

 @Resource
 private RestTemplate restTemplate;

 @GetMapping("/consumer/payment/consul")
 public String paymentInfo() {
 return restTemplate.getForObject(INVOKE_URL, String.class);
 }
}

```

## 服务调用

### Ribbon

#### 基本介绍

SpringCloud Ribbon 是基于 Netflix Ribbon 实现的一套负载均衡工具，提供客户端的软件负载均衡算法和服务调用，Ribbon 客户端组件提供一系列完善的配置项如连接超时，重试等

官网：<https://github.com/Netflix/ribbon/wiki/Getting-Started>（已进入维护模式，未来替换为 Load Banlancer）

负载均衡 Load Balance (LB) 就是将用户的请求平摊的分配到多个服务上，从而达到系统的 HA (高可用)

#### 常见的负载均衡算法：

- 轮询：为请求选择健康池中的第一个后端服务器，然后按顺序往后依次选择
- 最小连接：优先选择连接数最少，即压力最小的后端服务器，在会话较长的情况下可以采取这种方式
- 散列：根据请求源的 IP 的散列 (hash) 来选择要转发的服务器，可以一定程度上保证特定用户能连接到相同的服务器，如果应用需要处理状态而要求用户能连接到和之前相同的服务器，可以采取这种方式

Ribbon 本地负载均衡客户端与 Nginx 服务端负载均衡区别：

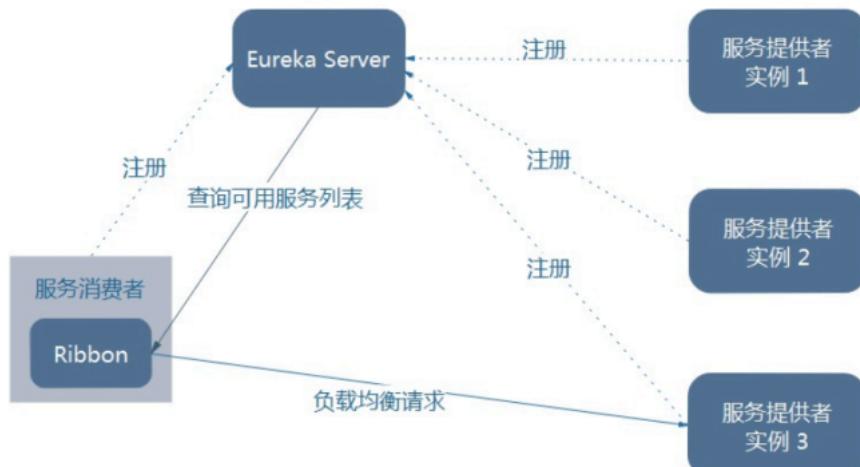
- Nginx 是服务器负载均衡，客户端所有请求都会交给 Nginx，然后由 Nginx 实现转发请求，即负载均衡是由服务端实现的
- Ribbon 本地负载均衡，在调用微服务接口时会在注册中心上获取注册信息服务列表，然后缓存到 JVM 本地，从而在本地实现 RPC 远程服务调用技术

集中式 LB 和进程内 LB 的对比：

- 集中式 LB：在服务的消费方和提供方之间使用独立的 LB 设施（如 Nginx），由该设施把访问请求通过某种策略转发至服务的提供方
- 进程内 LB：将 LB 逻辑集成到消费方，消费方从服务注册中心获知有哪些服务可用，然后从中选择出一个服务器，Ribbon 属于该类

## 工作流程

Ribbon 是一个软负载均衡的客户端组件

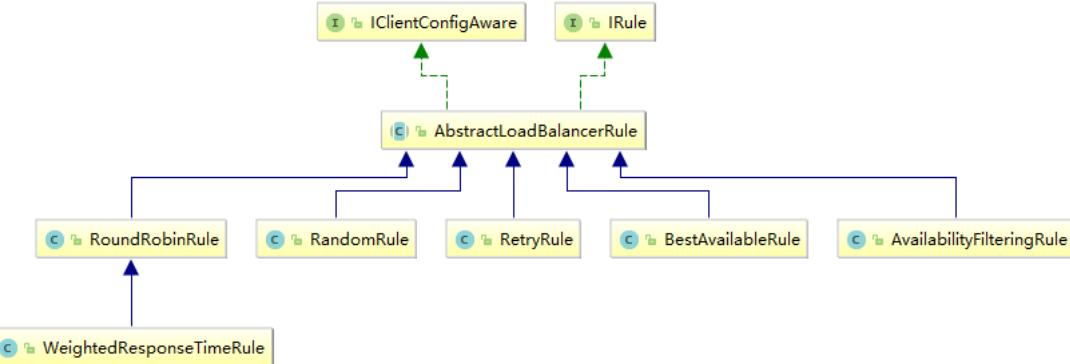


- 第一步先选择 EurekaServer，优先选择在同一个区域内负载较少的 Server
- 第二步根据用户指定的策略，再从 Server 取到的服务注册列表中选择一个地址

## 核心组件

Ribbon 核心组件 IRule 接口，主要实现类：

- RoundRobinRule：轮询
- RandomRule：随机
- RetryRule：先按照 RoundRobinRule 的策略获取服务，如果获取服务失败则在指定时间内会进行重试
- WeightedResponseTimeRule：对 RoundRobinRule 的扩展，响应速度越快的实例选择权重越大，越容易被选择
- BestAvailableRule：会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务
- AvailabilityFilteringRule：先过滤掉故障实例，再选择并发较小的实例
- ZoneAvoidanceRule：默认规则，复合判断 Server 所在区域的性能和 Server 的可用性选择服务器



注意：官方文档明确给出了警告，自定义负载均衡配置类不能放在 @ComponentScan 所扫描的当前包下以及子包下  
更换负载均衡算法方式：

- 自定义负载均衡配置类 MySelfRule：

```
@Configuration
public class MySelfRule {
 @Bean
 public IRule myRule() {
 return new RandomRule(); // 定义为随机负载均衡算法
 }
}
```

- 主启动类添加 @RibbonClient 注解

```
@SpringBootApplication
@EnableEurekaClient
// 指明访问的服务 CLOUD-PAYMENT-SERVICE，以及指定负载均衡策略
@RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration= MySelfRule.class)
public class OrderMain80 {
 public static void main(String[] args) {
 SpringApplication.run(OrderMain80.class, args);
 }
}
```

## OpenFeign

### 基本介绍

Feign 是一个声明式 WebService 客户端，能让编写 Web 客户端更加简单，只要创建一个接口并添加注解 @Feign 即可，可以与 Eureka 和 Ribbon 组合使用支持负载均衡，所以一般用在消费者端

OpenFeign 在 Feign 的基础上支持了 SpringMVC 注解，并且 @FeignClient 注解可以解析 @RequestMapping 注解下的接口，并通过动态代理的方式产生实现类，在实现类中做负载均衡和服务调用

优点：利用 RestTemplate 对 HTTP 请求的封装处理，形成了一套模板化的调用方法。但是对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以一个微服务接口上面标注一个 @Feign 注解，就可以完成包装依赖服务的调用

## 基本使用

@FeignClient("provider name") 注解使用规则：

- 声明的方法签名必须和 provider 微服务中的 controller 中的方法签名一致
- 如果需要传递参数，那么 @RequestParam 、 @RequestBody 、 @PathVariable 也需要加上

改造消费者服务

- 引入 pom 依赖：OpenFeign 整合了 Ribbon，具有负载均衡的功能

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

- application.yml：不将其注册到 Eureka 作为微服务

```
server:
 port: 80

eureka:
 client:
 # 表示不将其注入Eureka作为微服务，不作为Eureka客户端了，而是作为Feign客户端
 register-with-eureka: false
 service-url:
 # 集群版
 defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
```

- 主启动类：开启 Feign

```
@SpringBootApplication
@EnableFeignClients //不作为Eureka客户端了，而是作为Feign客户端
public class OrderOpenFeignMain80 {
 public static void main(String[] args) {
 SpringApplication.run(OrderOpenFeignMain80.class, args);
 }
}
```

- 新建 Service 接口：PaymentFeignService 接口和 @FeignClient 注解，完成 Feign 的包装调用

```
@Component
@FeignClient(value = "CLOUD-PAYMENT-SERVICE") // 作为一个Feign功能绑定的接口
public interface PaymentFeignService {
 @GetMapping(value = "/payment/get/{id}")
 public CommonResult<Payment> getPaymentById(@PathVariable("id") long id);

 @GetMapping("/payment/feign/timeout")
 public String paymentFeignTimeout();
}
```

- Controller：

```
@RestController
@Slf4j
public class OrderFeignController {
 @Autowired
 private PaymentFeignService paymentFeignService;

 @GetMapping("/consumer/payment/get/{id}")
 public CommonResult<Payment> getPayment(@PathVariable("id") long id) {
 // 返回对象为响应体中数据转化成的对象，基本上可以理解为JSON
 return paymentFeignService.getPaymentById(id);
 }

 @GetMapping("/consumer/payment/feign/timeout")
 public String paymentFeignTimeout() {
 // openfeign-ribbon，客户端一般默认等待1s
 return paymentFeignService.paymentFeignTimeout();
 }
}
```

## 超时问题

Feign 默认是支持 Ribbon, Feign 客户端的负载均衡和超时控制都由 Ribbon 控制

设置 Feign 客户端的超时等待时间:

```
ribbon:
 #指的是建立连接后从服务器读取到可用资源所用的时间
 ReadTimeout: 5000
 #指的是建立连接所用的时间，适用于网络状况正常的情况下，两端连接所用的时间
 ConnectTimeout: 5000
```

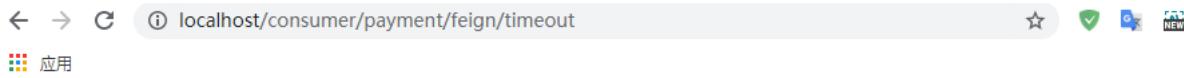
演示超时现象：OpenFeign 默认等待时间为 1 秒钟，超过后会报错

- 服务提供方 Controller:

```
@GetMapping("/payment/feign/timeout")
public String paymentFeignTimeout() {
 try {
 TimeUnit.SECONDS.sleep(3);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 return serverPort;
}
```

- 消费者 PaymentFeignService 和 OrderFeignController 参考上一小节代码

- 测试报错:



## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Sep 22 10:14:20 CST 2021

There was an unexpected error (type=Internal Server Error, status=500).

Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout

feign.RetryableException: Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout

(C:\Users\Seazean\Desktop\123\Cloud-OpenFeign超时错误.png)

## 日志级别

Feign 提供了日志打印功能，可以通过配置来调整日志级别，从而了解 Feign 中 HTTP 请求的细节

NONE	默认的，不显示任何日志
BASIC	仅记录请求方法、URL、响应状态码及执行时间
HEADERS	除了 BASIC 中定义的信息之外，还有请求和响应的头信息
FULL	除了 HEADERS 中定义的信息外，还有请求和响应的正文及元数据

配置在消费者端

- 新建 config.FeignConfig 文件：配置日志 Bean

```
@Configuration
public class FeignConfig {
 @Bean
 Logger.Level feignLoggerLevel() {
 return Logger.Level.FULL;
 }
}
```

- application.yml:

```
logging:
 level:
 # feign 日志以什么级别监控哪个接口
 com.atguigu.springcloud.service.PaymentFeignService: debug
```

- Debug 后查看后台日志

## 服务熔断

### Hystrix

#### 基本介绍

Hystrix 是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖会出现调用失败，比如超时、异常等，Hystrix 能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

断路器本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间地占用，避免了故障在分布式系统中的蔓延，乃至雪崩。

- 服务降级 Fallback：系统不可用时需要一个兜底的解决方案或备选响应，向调用方返回一个可处理的响应。
- 服务熔断 Break：达到最大服务访问后，直接拒绝访问。
- 服务限流 Flowlimit：高并发操作时严禁所有请求一次性过来拥挤，一秒钟 N 个，有序排队进行。

官方文档：<https://github.com/Netflix/Hystrix/wiki/How-To-Use>

## 服务降级

### 案例构建

生产者模块：

- 引入 pom 依赖：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

- 主启动类：开启 Feign

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker // 降级使用
public class PaymentHystrixMain8001 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentHystrixMain8001.class, args);
 }
}
```

- Controller：

```
@RestController
@Slf4j
public class PaymentController {
 @Resource
 private PaymentService paymentService;
 @Value("${server.port}")
 private String serverPort;

 // 正常访问
 @GetMapping("/payment/hystrix/ok/{id}")
 private String paymentInfo_ok(@PathVariable("id") Integer id) {
 return paymentService.paymentInfo_ok(id);
 }

 // 超时
 @GetMapping("/payment/hystrix/timeout/{id}")
 private String paymentInfo_Timeout(@PathVariable("id") Integer id) {
 // service 层有 Thread.sleep() 操作，保证超时
 return paymentService.paymentInfo_Timeout(id);
 }
}
```

- Service：

```
@Service
public class PaymentService {
 public String paymentInfo_ok(Integer id) {
```

```

 return "线程池: " + Thread.currentThread().getName() + "paymentInfo_OK, id: " + id;
 }

 public String paymentInfo_Timeout(Integer id) {
 int timeNumber = 3;
 try {
 TimeUnit.SECONDS.sleep(timeNumber);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 return "线程池: " + Thread.currentThread().getName() + " payment_Timeout, id: " + id;
 }
}

```

- jmeter 压测两个接口，发现接口 paymentInfo\_Ok 也变的卡顿

消费者模块：

- Service 接口：

```

@Component
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT")
public interface PaymentHystrixService {
 @GetMapping("/payment/hystrix/ok/{id}")
 public String paymentInfo_Ok(@PathVariable("id") Integer id);

 @GetMapping("/payment/hystrix/timeout/{id}")
 public String paymentInfo_Timeout(@PathVariable("id") Integer id);
}

```

- Controller：

```

@RestController
@Slf4j
public class OrderHystrixController {
 @Resource
 PaymentHystrixService paymentHystrixService;

 @GetMapping("/consumer/payment/hystrix/ok/{id}")
 public String paymentInfo_Ok(@PathVariable("id") Integer id) {
 return paymentHystrixService.paymentInfo_Ok(id);
 }

 @GetMapping("/consumer/payment/hystrix/timeout/{id}")
 public String paymentInfo_Timeout(@PathVariable("id") Integer id) {
 return paymentHystrixService.paymentInfo_Timeout(id);
 }
}

```

- 测试：使用的是 Feign 作为客户端，默认 1s 没有得到响应就会报超时错误，进行并发压测

- 解决：

- 超时导致服务器变慢（转圈）：超时不再等待
- 出错（宕机或程序运行出错）：出错要有兜底

## 降级操作

生产者端和消费者端都可以进行服务降级，使用 @HystrixCommand 注解指定降级后的方法

生产者端：主启动类添加新注解 @EnableCircuitBreaker，业务类（Service）方法进行如下修改，

```

// 模拟拥堵的情况
@HystrixCommand(fallbackMethod = "paymentInfo_TimeoutHandler", commandProperties = {
 // 规定这个线程的超时时间是3s，3s后就由fallbackMethod指定的方法“兜底”（服务降级）
 @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value = "3000")
})
public String paymentInfo_Timeout(Integer id) {
 // 超时或者出错
}

public String paymentInfo_TimeoutHandler(Integer id) {
 return "线程池: " + Thread.currentThread().getName() + " paymentInfo_TimeoutHandler, id: " + id;
}

```

服务降级的方法和业务处理的方法混杂在了一块，耦合度很高，并且每个方法配置一个服务降级方法

- 在业务类Controller上加 @DefaultProperties(defaultFallback = "method\_name") 注解
- 在需要服务降级的方法上标注 @HystrixCommand 注解，如果 @HystrixCommand 里没有指明 fallbackMethod，就默认使用 @DefaultProperties 中指明的降级服务

```

@RestController
@Slf4j
@DefaultProperties(defaultFallback = "payment_Global_FallbackMethod")
public class OrderHystrixController {
 @Resource
 PaymentHystrixService paymentHystrixService;

 @GetMapping("/consumer/payment/hystrix/ok/{id}")
 public String paymentInfo_OK(@PathVariable("id") Integer id) {
 return paymentHystrixService.paymentInfo_OK(id);
 }

 @HystrixCommand
 public String paymentInfo_Timeout(@PathVariable("id") Integer id) {
 return paymentHystrixService.paymentInfo_Timeout(id);
 }

 public String paymentTimeoutFallbackMethod(@PathVariable("id") Integer id) {
 return "fallback";
 }

 // 下面是全局fallback方法
 public String payment_Global_FallbackMethod() {
 return "Global fallback";
 }
}

```

客户端调用服务端，遇到服务端宕机或关闭等极端情况，为 Feign 客户端定义的接口添加一个服务降级实现类即可实现解耦

- application.yml: 配置文件中开启了 Hystrix

```

用于服务降级 在注解 @FeignClient中添加fallbackFactory属性值
feign:
 hystrix:
 enabled: true #在Feign中开启Hystrix

```

- Service: 统一为接口里面的方法进行异常处理，服务异常找 PaymentFallbackService，来统一进行服务降级的处理

```

@Component
@FeignClient(value = "PROVIDER-HYSTRIX-PAYMENT", fallback = PaymentFallbackService.class)
public interface PaymentHystrixService {

 @GetMapping("/payment/hystrix/ok/{id}")
 public String paymentInfo_OK(@PathVariable("id") Integer id);

 @GetMapping("/payment/hystrix/timeout/{id}")
 public String paymentInfo_Timeout(@PathVariable("id") Integer id);
}

```

- PaymentFallbackService:

```

@Component
public class PaymentFallbackService implements PaymentHystrixService {
 @Override
 public String paymentInfo_OK(Integer id) {
 return "-----PaymentFallbackService-paymentInfo_OK, fallback";
 }

 @Override
 public String paymentInfo_Timeout(Integer id) {
 return "-----PaymentFallbackService-paymentInfo_Timeout, fallback";
 }
}

```

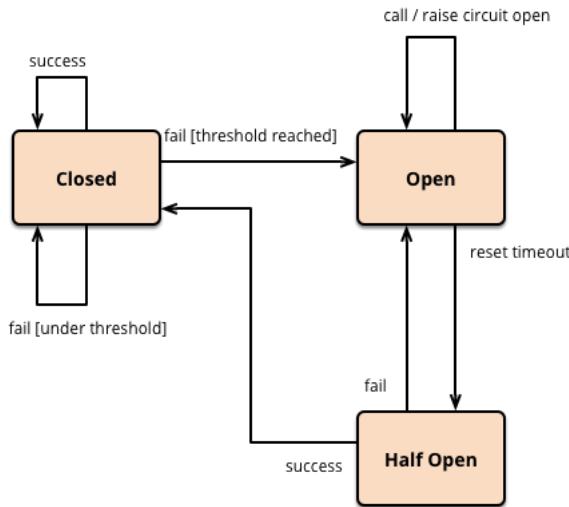
## 服务熔断

### 熔断类型

熔断机制是应对雪崩效应的一种微服务链路保护机制，当扇出链路的某个微服务出错不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应信息

Hystrix 会监控微服务间调用的状况，当失败的调用到一定阈值，缺省时 5 秒内 20 次调用失败，就会启动熔断机制；当检测到该节点微服务调用响应正常后（检测方式是尝试性放开请求），自动恢复调用链路

- 熔断打开：请求不再进行调用当前服务，再有请求调用时将不会调用主逻辑，而是直接调用降级 fallback。实现了自动的发现错误并将降级逻辑切换为主逻辑，减少响应延迟效果。内部设置时钟一般为 MTTR (Mean time to repair, 平均故障处理时间)，当打开时长达到所设时钟则进入半熔断状态
- 熔断关闭：熔断关闭不会对服务进行熔断，服务正常调用
- 熔断半开：部分请求根据规则调用当前服务，如果请求成功且符合规则则认为当前服务恢复正常，关闭熔断，反之继续熔断



## 熔断操作

涉及到断路器的四个重要参数：快照时间窗、请求总数阀值、窗口睡眠时间、错误百分比阀值

- circuitBreaker.enabled: 是否开启断路器
- metrics.rollingStats.timeInMilliseconds: 快照时间窗口，断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的 10 秒
- circuitBreaker.requestVolumeThreshold: 请求总数阀值，该属性设置在快照时间窗内（默认 10s）使断路器跳闸的最小请求数量（默认是 20），如果 10s 内请求数小于设定值，就算请求全部失败也不会触发断路器
- circuitBreaker.sleepWindowInMilliseconds: 窗口睡眠时间，短路多久以后开始尝试是否恢复进入半开状态，默认 5s
- circuitBreaker.errorThresholdPercentage: 错误百分比阀值，失败率达到多少后将断路器打开

```

//总的意思就是在n(10)毫秒内的时间窗口期内, m次请求中有p% (60%) 的请求失败了, 那么断路器启动
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
 @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),
 @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),
 @HystrixProperty(name = "circuitBreaker.sleepwindowInMilliseconds", value = "10000"),
 @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")
})
public String paymentCircuitBreaker(@PathVariable("id") Integer id) {
 if (id < 0) {
 throw new RuntimeException("*****id 不能负数");
 }
 String serialNumber = IdUtil.simpleUUID(); // 等价于UUID.randomUUID().toString()
 return Thread.currentThread().getName() + "\t" + "调用成功, 流水号: " + serialNumber;
}

```

- 开启：满足一定的阈值（默认 10 秒内超过 20 个请求次数）、失败率达到阈值（默认 10 秒内超过 50% 的请求失败）
- 关闭：一段时间之后（默认是 5 秒），断路器是半开状态，会让其中一个请求进行转发，如果成功断路器会关闭，反之继续开启

## 工作流程

具体工作流程：

1. 创建 HystrixCommand（用在依赖的服务返回单个操作结果的时候）或 HystrixObservableCommand（用在依赖的服务返回多个操作结果的时候）对象
2. 命令执行，其中 HystrixComand 实现了下面前两种执行方式，而 HystrixObservableCommand 实现了后两种执行方式
  - execute(): 同步执行，从依赖的服务返回一个单一的结果对象，或是在发生错误的时候抛出异常
  - queue(): 异步执行，直接返回一个 Future 对象，其中包含了服务执行结束时要返回的单一结果对象
  - observe(): 返回 Observable 对象，代表了操作的多个结果，它是一个 Hot Observable（不论事件源是否有订阅者，都会在创建后对事件进行发布，所以对于 Hot Observable 的每个订阅者都有可能是从事件源的中途开始的，并可能只是看到了整个操作的局部过程）
  - toObservable(): 同样会返回 Observable 对象，也代表了操作的多个结果，但它返回的是一个 Cold Observable（没有订阅者的时候并不会发布事件，而是进行等待，直到有订阅者之后才发布事件，所以对于 Cold Observable 的订阅者，它可以保证从一开始看到整个操作的全部过程）
3. 若当前命令的请求缓存功能是被启用的，并且该命令缓存命中，那么缓存的结果会立即以 Observable 对象的形式返回
4. 检查断路器是否为打开状态，如果断路器是打开的，那么 Hystrix 不会执行命令，而是转接到 fallback 处理逻辑（第 8 步）；如果断路器是关闭的，检查是否有可用资源来执行命令（第 5 步）
5. 线程池/请求队列/信号量是否占满，如果命令依赖服务的专有线程池和请求队列，或者信号量（不使用线程池时）已经被占满，那么 Hystrix 也不会执行命令，而是转接到 fallback 处理逻辑（第 8 步）

6. Hystrix 会根据我们编写的方法来决定采取什么样的方式去请求依赖服务

- HystrixCommand.run(): 返回一个单一的结果，或者抛出异常
- HystrixObservableCommand.construct(): 返回一个 Observable 对象来发射多个结果，或通过 onError 发送错误通知

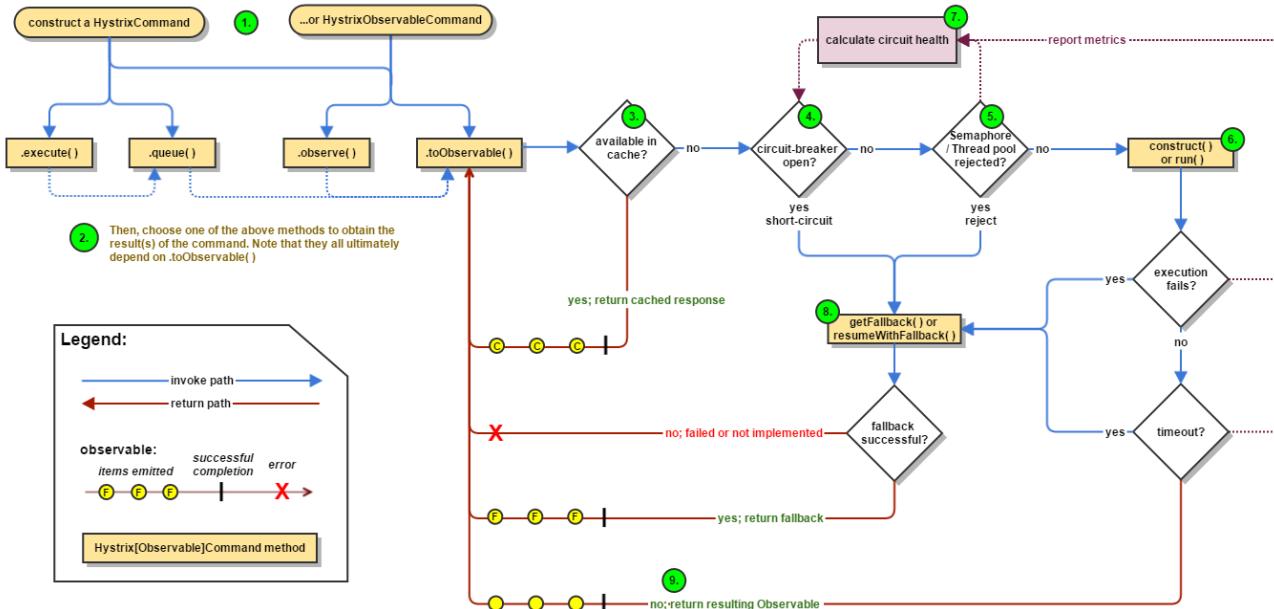
7. Hystrix 会将“成功”、“失败”、“拒绝”、“超时”等信息报告给断路器，而断路器会维护一组计数器来统计这些数据。断路器会使用这些统计数据来决定是否要将断路器打开，来对某个依赖服务的请求进行“熔断/短路”

8. 当命令执行失败的时候，Hystrix 会进入 fallback 尝试回退处理，通常也称该操作为“服务降级”，而能够引起服务降级情况：

- 第 4 步：当前命令处于“熔断/短路”状态，断路器是打开的时候
- 第 5 步：当前命令的线程池、请求队列或者信号量被占满的时候
- 第 6 步：HystrixObservableCommand.construct() 或 HystrixCommand.run() 抛出异常的时候

9. 当 Hystrix 命令执行成功之后，它会将处理结果直接返回或是以 Observable 的形式返回

注意：如果、没有为命令实现降级逻辑或者在降级处理逻辑中抛出了异常，Hystrix 依然会返回一个 Observable 对象，但是它不会发射任何结果数据，而是通过 onError 方法通知命令立即中断请求，并通过 onError() 方法将引起命令失败的异常发送给调用者



官方文档：<https://github.com/Netflix/Hystrix/wiki/How-it-Works>

## 服务监控

Hystrix 提供了准实时的调用监控（Hystrix Dashboard），Hystrix 会持续的记录所有通过 Hystrix 发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求多少成功，多少失败等，Netflix 通过 hystrix-metrics-event-stream 项目实现了对以上指标的监控，Spring Cloud 提供了 Hystrix Dashboard 的整合，对监控内容转化成可视化页面

- 引入 pom 依赖：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

- application.yml：只需要端口即可

```
server:
 port: 9001
```

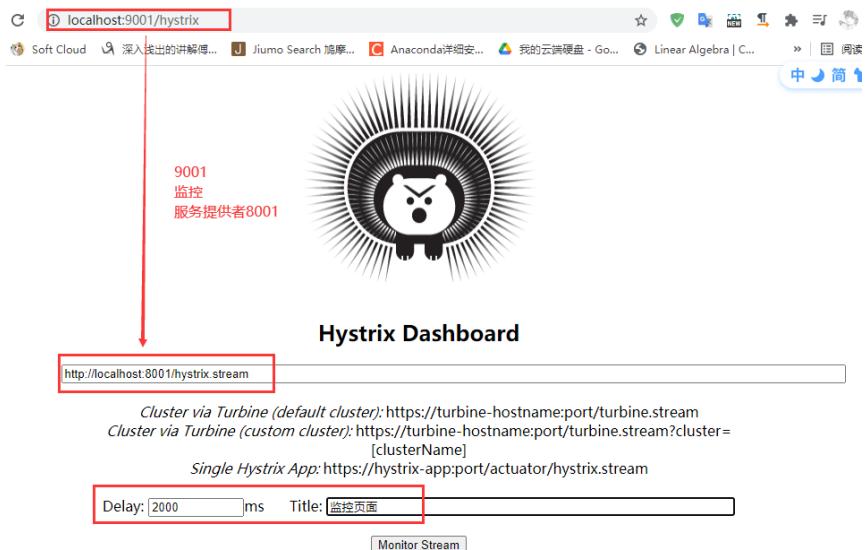
- 主启动类：

```
@SpringBootApplication
@EnableHystrixDashboard // 开启Hystrix仪表盘
public class HystrixDashboardMain9001 {
 public static void main(String[] args) {
 SpringApplication.run(HystrixDashboardMain9001.class, args);
 }
}
```

- 所有微服务（生产者）提供类 8001/8002/8003 都需要监控依赖配置

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

- 启动测试: <http://localhost:9001/hystrix>

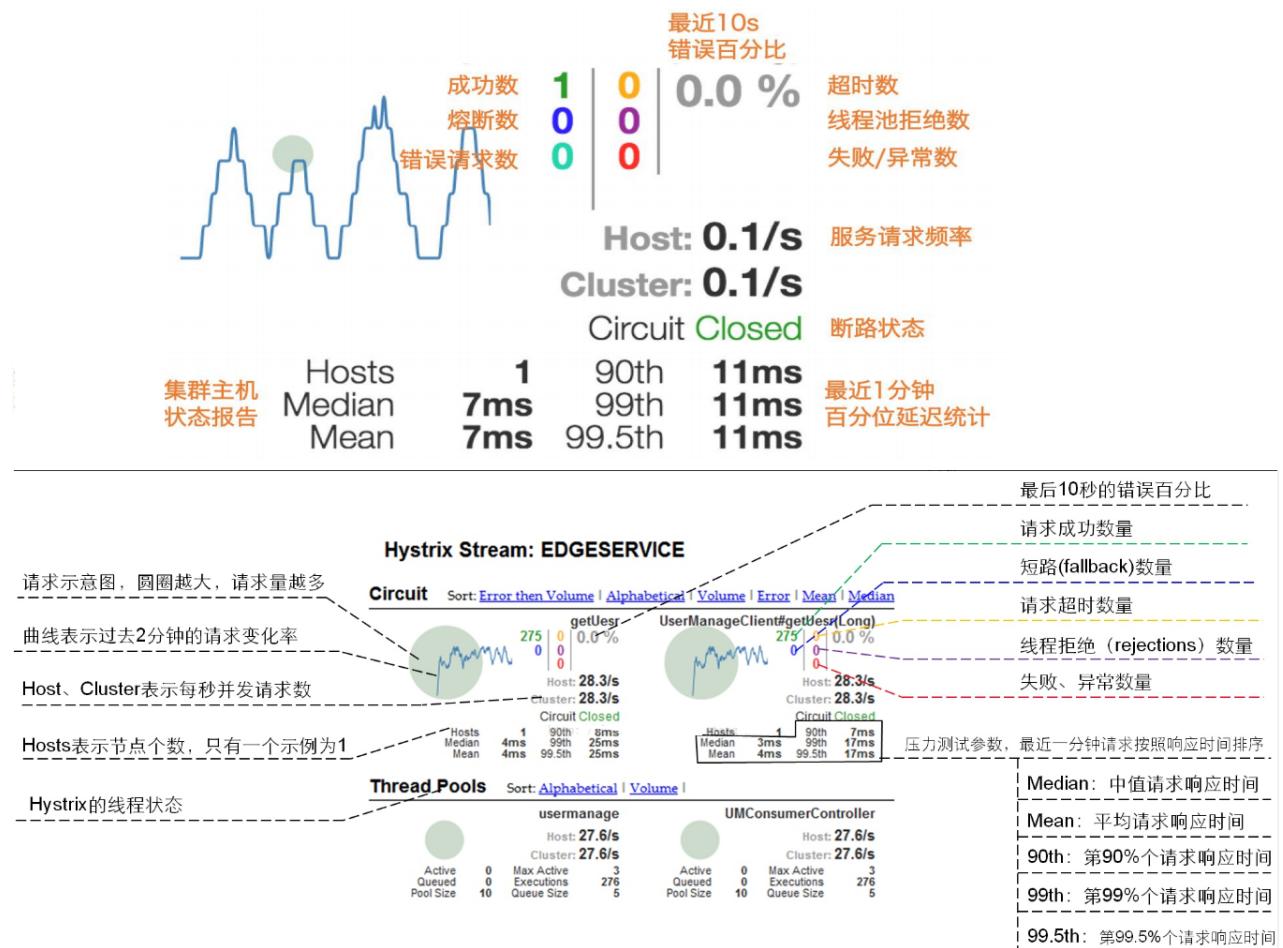


- 新版本 Hystrix 需要在需要监控的微服务端的主启动类中指定监控路径，不然会报错

```
@SpringBootApplication
@EnableEurekaClient // 本服务启动后会自动注册进eureka服务中
@EnableCircuitBreaker // 对hystrixR熔断机制的支持
public class PaymentHystrixMain8001 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentHystrixMain8001.class, args);
 }

 /**
 *=====
 *此配置是为了服务监控而配置，与服务容错本身无关，springcloud升级后的坑
 *ServletRegistrationBean因为springboot的默认路径不是"/hystrix.stream",
 *只要在自己的项目里配置上下面的servlet就可以了
 */
 @Bean
 public ServletRegistrationBean getServlet() {
 HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();
 ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);
 registrationBean.setLoadOnStartup(1);
 registrationBean.addUrlMappings("/hystrix.stream");
 registrationBean.setName("HystrixMetricsStreamServlet");
 return registrationBean;
 }
}
```

- 指标说明:



## 服务网关

### Zuul

SpringCloud 中所集成的 Zuul 版本，采用的是 Tomcat 容器，基于 Servlet 之上的一一个阻塞式处理模型，不支持任何长连接，用 Java 实现，而 JVM 本身会有第一次加载较慢的情况，使得 Zuul 的性能相对较差

官网： <https://github.com/Netflix/zuul/wiki>

## Gateway

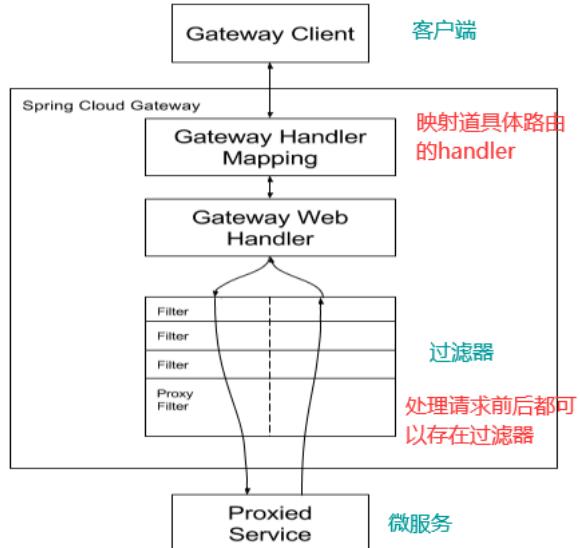
### 基本介绍

SpringCloud Gateway 是 Spring Cloud 的一个全新项目，基于 Spring 5.0+Spring Boot 2.0 和 Project Reactor 等技术开发的网关，旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。

- 基于 WebFlux 框架实现，而 WebFlux 框架底层则使用了高性能的 Reactor 模式通信框架 Netty (异步非阻塞响应式的框架)
- 基于 Filter 链的方式提供了网关基本的功能，例如：安全、监控/指标、限流等

Gateway 的三个核心组件：

- Route：路由是构建网关的基本模块，由 ID、目标 URI、一系列的断言和过滤器组成，如果断言为 true 则匹配该路由
- Predicate：断言，可以匹配 HTTP 请求中的所有内容（例如请求头或请求参数），如果请求参数与断言相匹配则进行路由
- Filter：指 Spring 框架中的 GatewayFilter 实例，使用过滤器可以在请求被路由前或之后（拦截）对请求进行修改



核心逻辑：路由转发 + 执行过滤器链

- 客户端向 Spring Cloud Gateway 发出请求，然后在 Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到 Gateway Web Handler
- Handler 通过指定的过滤器链来将请求发送到实际的服务执行业务逻辑，然后返回
- 过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前 (pre) 或之后 (post) 执行业务逻辑
- Filter 在 pre 类型的过滤器可以做参数校验、权限校验、流量监控、日志输出、协议转换等，在 post 类型的过滤器中可以做响应内容、响应头的修改、日志的输出、流量监控等

## 网关使用

### 配置方式

Gateway 网关路由有两种配置方式，分别为通过 yml 配置和注入 Bean

- 引入 pom 依赖：Gateway 不需要 spring-boot-starter-web 依赖，否在会报错，原因是底层使用的是 WebFlux 与 Web 冲突

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

- application.yml:

```
server:
 port: 9527

spring:
 application:
 name: cloud-gateway

eureka:
 instance:
 hostname: cloud-gateway-service
 client: #服务提供者provider注册进eureka服务列表内
 service-url:
 register-with-eureka: true
 fetch-registry: true
 defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka #集群版
```

- 主启动类（网关不需要业务类）：

```
@SpringBootApplication
@EnableEurekaClient
public class GatewayMain9527 {
 public static void main(String[] args) {
 SpringApplication.run(GatewayMain9527.class, args);
 }
}
```

- 以前访问 provider-payment8001 中的 Controller 方法，通过 localhost:8001/payment/get/id 和 localhost:8001/payment/lb，项目不想暴露 8001 端口号，希望在 8001 外面套一层 9527 端口：

```
server:
 port: 9527
```

```

spring:
 application:
 name: cloud-gateway
=====新增=====
cloud:
 gateway:
 routes:
 - id: payment_routh # payment_route #路由的ID, 没有固定规则但要求【唯一】，建议配合服务名
 uri: http://localhost:8001 #匹配后提供服务的路由地址
 predicates:
 - Path=/payment/get/** # 断言，路径相匹配的进行路由

 - id: payment_routh2 # payment_route#路由的ID, 没有固定规则但要求【唯一】，建议配合服务名
 uri: http://localhost:8001 #匹配后提供服务的路由地址
 predicates:
 - Path=/payment/lb/** # 断言，路径相匹配的进行路由

```

- uri + predicate 拼接就是具体的接口请求路径，通过 localhost:9527 映射的地址
- predicate 断言 <http://localhost:8001>下面有一个 /payment/get/\*\* 的地址，如果找到了该地址就返回 true，可以用 9527 端口访问，进行端口的适配
- \*\* 表示通配符，因为这是一个不确定的参数

## 注入Bean

通过 9527 网关访问到百度的网址 <https://www.baidu.com/>，在 config 包下创建一个配置类，路由规则是访问 /baidu 跳转到百度

```

@Configuration
public class GatewayConfig {
 // 配置了一个 id 为 path_route_cloud 的路由规则
 @Bean
 public RouteLocator customRouteLocator(RouteLocatorBuilder routeLocatorBuilder){
 // 构建一个路由器，这个routes相当于yml配置文件中的routes
 RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
 // 路由器的id是: path_route_cloud, 规则是访问/baidu , 将会转发到 https://www.baidu.com/
 routes.route("path_route_cloud",
 r -> r.path("/baidu").uri(" https://www.baidu.com")).build();
 return routes.build();
 }
}

```

## 动态路由

Gateway 会根据注册中心注册的服务列表，以注册中心上微服务名为路径创建动态路由进行转发，从而实现动态路由和负载均衡，避免出现一个路由规则仅对应一个接口方法，当请求地址很多时需要很大的配置文件

application.yml 开启动态路由功能

```

spring:
 application:
 name: cloud-gateway
 cloud:
 gateway:
 discovery:
 locator:
 enabled: true # 开启从注册中心动态创建路由的功能，利用微服务名进行路由
 routes:
 - id: payment_routh1 # 路由的ID, 没有固定规则但要求唯一，建议配合服务名
 uri: lb://cloud-payment-service #匹配后提供服务的路由地址
 predicates:
 - Path=/payment/get/** # 断言，路径相匹配的进行路由

 - id: payment_routh2 #路由的ID, 没有固定规则但要求唯一，建议配合服务名
 uri: lb://cloud-payment-service #匹配后提供服务的路由地址
 predicates:
 - Path=/payment/lb/** # 断言，路径相匹配的进行路由
 - After=2021-09-28T19:14:51.514+08:00[Asia/Shanghai]

```

lb:// 开头代表从注册中心中获取服务，后面是需要转发到的服务名称

## 断言类型

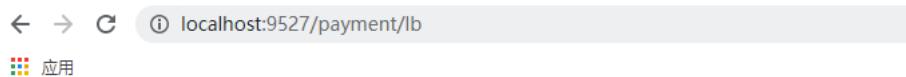
After Route Predicate: 匹配该断言时间之后的 URI 请求

- 获取时间:

```
public class TimeTest {
 public static void main(String[] args) {
 ZonedDateTime zbj = ZonedDateTime.now(); // 默认时区
 System.out.println(zbj); //2023-01-10T16:31:44.106+08:00[Asia/Shanghai]
 }
}
```

- 配置 yaml: 动态路由小结有配置

- 测试: 正常访问成功, 将时间修改到 2023-01-10T16:31:44.106+08:00[Asia/Shanghai] 之后访问失败



## Whitelabel Error Page

This application has no configured error view, so you are seeing this as a fallback.

Tue Sep 28 19:20:47 CST 2021

[f78a48b0] There was an unexpected error [type=Not Found, status=404].  
org.springframework.web.server.ResponseStatusException: 404 NOT FOUND  
at org.springframework.web.reactive.resource.ResourceWebHandler.lambda\$handle!  
Suppressed: reactor.core.publisher.FluxOnAssembly\$OnAssemblyException:  
Error has been observed at the following site(s):

常见断言类型:

- Before Route Predicate: 匹配该断言时间之前的 URI 请求
- Between Route Predicate: 匹配该断言时间之间的 URI 请求

```
- Between=2022-02-02T17:45:06.206+08:00[Asia/Shanghai],2022-03-25T18:59:06.206+08:00[Asia/Shanghai]
```

- Cookie Route Predicate: Cookie 断言, 两个参数分别是 Cookie name 和正则表达式, 路由规则会通过获取对应的 Cookie name 值和正则表达式去匹配, 如果匹配上就会执行路由

```
- Cookie=username, seazean # 只有发送的请求有 cookie, 而且有username=seazean这个数据才能访问, 反之404
```

- Header Route Predicate: 请求头断言

```
- Header=X-Request-Id, \d+ # 请求头要有 X-Request-Id 属性, 并且值为整数的正则表达式
```

- Host Route Predicate: 指定主机可以访问, 可以指定多个用 , 分隔开

```
- Host=*.seazean.com
```

- Method Route Predicate: 请求类型断言

```
- Method=GET # 只有 Get 请求才能访问
```

- Path Route Predicate: 路径匹配断言

- Query Route Predicate: 请求参数断言

```
- Query=username, \d+ # 要有参数名 username 并且值还要是整数才能路由
```

## Filter使用

Filter 链是同时满足一系列的过滤链, 路由过滤器可用于修改进入的 HTTP 请求和返回的 HTTP 响应, 路由过滤器只能指定路由进行使用, Spring Cloud Gateway 内置了多种路由过滤器, 都由 GatewayFilter 的工厂类来产生

配置文件: <https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.1.RELEASE/reference/html/#gatewayfilter-factories>

自定义全局过滤器: 实现两个主要接口 GlobalFilter, Ordered

```
@Component
@Slf4j
public class MyLogGatewayFilter implements GlobalFilter, Ordered {
```

```

@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
 log.info("*****Come in MyLogGatewayFilter: " + new Date());
 // 取出请求参数的uname对应的值
 String uname = exchange.getRequest().getQueryParams().getFirst("uname");
 // 如果 uname 为空，就直接过滤掉，不走路由
 if(uname == null){
 log.info("***** 用户名为 NULL 非法用户 o(╥﹏╥)o");
 }
 // 判断该请求不通过时：给一个回应，返回
 exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
 return exchange.getResponse().setComplete();
}

// 反之，调用下一个过滤器，也就是放行：在该环节判断通过的 exchange 放行，交给下一个 filter 判断
return chain.filter(exchange);
}

// 设置这个过滤器在Filter链中的加载顺序，数字越小，优先级越高
@Override
public int getOrder() {
 return 0;
}
}

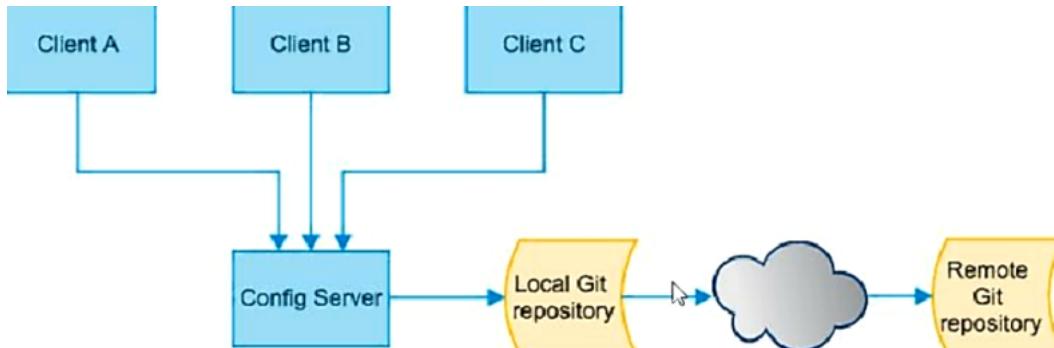
```

## 服务配置

### config

#### 基本介绍

SpringCloud Config 为微服务架构中的微服务提供集中化的外部配置支持（Git/GitHub），为各个不同微服务应用的所有环境提供了一个中心化的外部配置（Config Server）



SpringCloud Config 分为服务端和客户端两部分

- 服务端也称为分布式配置中心，是一个独立的微服务应用，连接配置服务器并为客户端提供获取配置信息，加密/解密等信息访问接口
- 客户端通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动时从配置中心获取和加载配置信息，配置服务器默认采用 Git 来存储配置信息，这样既有助于对环境配置进行版本管理，也可以通过 Git 客户端来方便的管理和访问配置内容

优点：

- 集中管理配置文件
- 不同环境不同配置，动态化的配置更新，分环境部署比如 dev/test/prod/beta/release
- 运行期间动态调整配置，服务向配置中心统一拉取配置的信息，**服务不需要重启即可感知到配置的变化并应用新的配置**
- 将配置信息以 Rest 接口的形式暴露

官网：<https://cloud.spring.io/spring-cloud-static/spring-cloud-config/2.2.1.RELEASE/reference/html/>

## 服务端

构建 Config Server 模块

- 引入 pom 依赖:

```
<!--springCloud Config Server-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

- application.yml:

```
server:
 port: 3344

spring:
 application:
 name: cloud-config-center #注册进Eureka服务器的微服务名
 cloud:
 config:
 server:
 git:
 # GitHub上面的git仓库名字 这里可以写https地址跟ssh地址, https地址需要配置username和 password
 uri: git@github.com:seazean/springcloud-config.git
 default-label: main
 search-paths:
 - springcloud-config # 搜索目录
 # username:
 # password:
 label: main # 读取分支,以前是master

#服务注册到eureka地址
eureka:
 client:
 service-url:
 defaultzone: http://localhost:7001/eureka,http://localhost:7002/eureka #集群版
```

search-paths 表示远程仓库下有一个叫做 springcloud-config 的, label 则表示读取 main 分支里面的内容

- 主启动类:

```
@SpringBootApplication
@EnableEurekaClient
@EnableConfigServer //开启SpringCloud的
public class ConfigCenterMain3344 {
 public static void main(String[] args) {
 SpringApplication.run(ConfigCenterMain3344.class, args);
 }
}
```

配置读取规则:

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

- label: 分支
- name: 服务名
- profile: 环境 (dev/test/prod)

比如: <http://localhost:3344/master/config-dev.yaml>

## 客户端

### 基本配置

配置客户端 Config Client, 客户端从配置中心 (Config Server) 获取配置信息

- 引入 pom 依赖:

```
<!--这里就是客户端的SpringCloud config 因为是客户端所以没有server-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

- bootstrap.yml: 系统级配置, 优先级更高, application.yml 是用户级的资源配置项

Spring Cloud 会创建一个 Bootstrap Context 作为 Spring 应用的 Application Context 的父上下文, 初始化的时候 Bootstrap Context 负责从外部源加载配置属性并解析配置, 这两个上下文共享一个从外部获取的 Environment, 为了配置文件的加载顺序和分级管理, 这里使用 bootstrap.yml

```
server:
 port: 3355 # 构建多个微服务, 3366 3377 等

spring:
 application:
 name: config-client
 cloud:
 #Config客户端配置
 config:
 label: main #分支名称 以前是master
 name: config #配置文件名称
 profile: dev #读取后缀名称
 # main分支上config-dev.yml的配置文件被读取 http://config-3344.com:3344/master/config-dev.yml
 uri: http://localhost:3344 # 配置中心地址

 #服务注册到eureka地址
 eureka:
 client:
 service-url:
 defaultZone: http://localhost:7001/eureka,http://localhost:7002/eureka
```

- 主启动类:

```
@SpringBootApplication
@EnableEurekaClient
public class ConfigClientMain3355 {
 public static void main(String[] args) {
 SpringApplication.run(ConfigClientMain3355.class, args);
 }
}
```

- 业务类: 将配置信息以 REST 窗口的形式暴露

```
@RestController
public class ConfigClientController {
 @Value("${config.info}")
 private String configInfo;

 @GetMapping("/configInfo")
 public String getConfigInfo() {
 return configInfo;
 }
}
```

## 动态刷新

分布式配置的动态刷新问题, 修改 GitHub 上的配置文件, Config Server 配置中心立刻响应, 但是 Config Client 客户端没有任何响应, 需要重启客户端

- 引入 pom 依赖:

```
<!--web/actuator这两个一般一起使用, 写在一起-->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- 修改 yml, 暴露监控端口: SpringBoot 的 actuator 启动端点监控 Web 端默认加载默认只有两个 info, health 可见的页面节点

```

management:
endpoints:
 web:
 exposure:
 include: "*" # 表示包含所有节点页面
 exclude: env,beans # 表示排除env、beans

```

- 业务类：加 @RefreshScope 注解

```

@RestController
@RefreshScope
public class ConfigClientController {
 // 从配置文件中取前缀为server.port的值
 @Value("${config.info}")
 private String configInfo;
 // config-{profile}.yml
 @GetMapping("/configInfo")
 public String getConfigInfo() {
 return configInfo;
 }
}

```

此时客户端还是没有刷新，需要发送 POST 请求刷新 3355: curl -X POST "http://localhost:3355/actuator/refresh"

引出问题：

- 在微服务多的情况下，每个微服务都需要执行一个 POST 请求，手动刷新成本太大
- 可否广播，一次通知，处处生效，大范围的实现自动刷新

解决方法：Bus 总线

## 服务消息

### Bus

#### 基本介绍

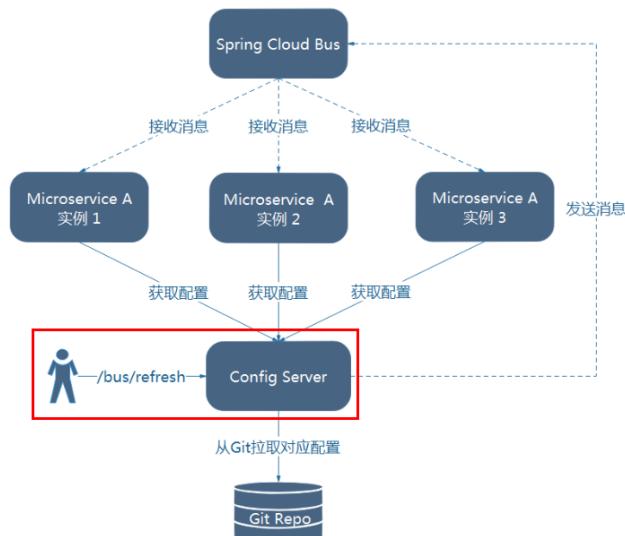
Spring Cloud Bus 能管理和传播分布式系统间的消息，就像分布式执行器，可用于广播状态更改、事件推送、微服务间的通信通道等

消息总线：在微服务架构的系统中，通常会使用轻量级的消息代理来构建一个共用的消息主题，并让系统中所有微服务实例都连接上来。由于该主题中产生的消息会被所有实例监听和消费，所以称为消息总线

基本原理：ConfigClient 实例都监听 MQ 中同一个 Topic (默认 springCloudBus)，当一个服务刷新数据时，会把信息放入 Topic 中，这样其它监听同一 Topic 的服务就能得到通知，然后去更新自身的配置

### 全局广播

利用消息总线接触一个服务端 ConfigServer 的 /bus/refresh 断点，从而刷新所有客户端的配置



改造 ConfigClient：

- 引入 MQ 的依赖：

```

<!--添加消息总线RabbitMQ支持-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

```

- yml 文件添加 MQ 信息:

```

server:
 port: 3344

spring:
 application:
 name: config-client #注册进Eureka服务器的微服务名
 cloud:
 # rabbitmq相关配置
 rabbitmq:
 host: localhost
 port: 5672
 username: guest
 password: guest

 # rabbitmq相关配置,暴露bus刷新配置的端点
management:
 endpoints: # 暴露bus刷新配置的端点
 web:
 exposure:
 include: 'bus-refresh'

```

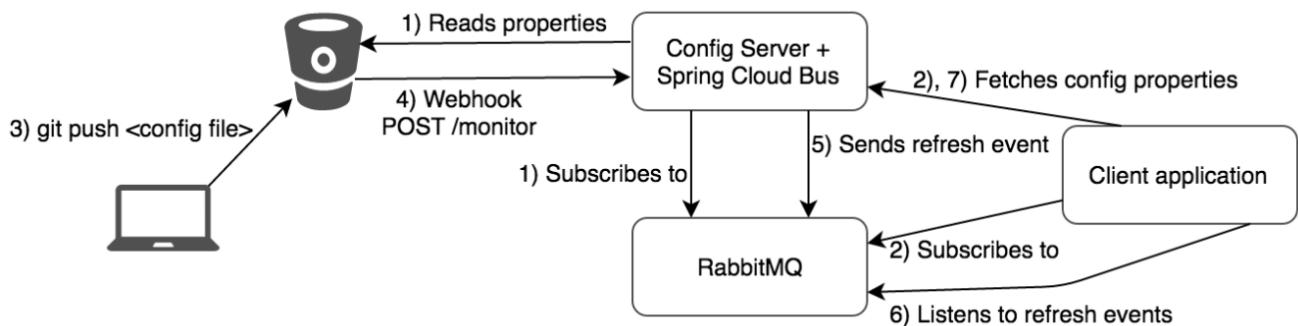
- 只需要调用一次 `curl -X POST "http://localhost:3344/actuator/bus-refresh"`, 可以实现全局广播

## 定点通知

动态刷新情况下, 只通知指定的微服务, 比如只通知 3355 服务, 不通知 3366, 指定具体某一个实例生效, 而不是全部

公式: `http://localhost:port/actuator/bus-refresh/{destination}`

`/bus/refresh` 请求不再发送到具体的服务实例上, 而是发给 Config Server 并通过 `destination` 参数类指定需要更新配置的服务或实例

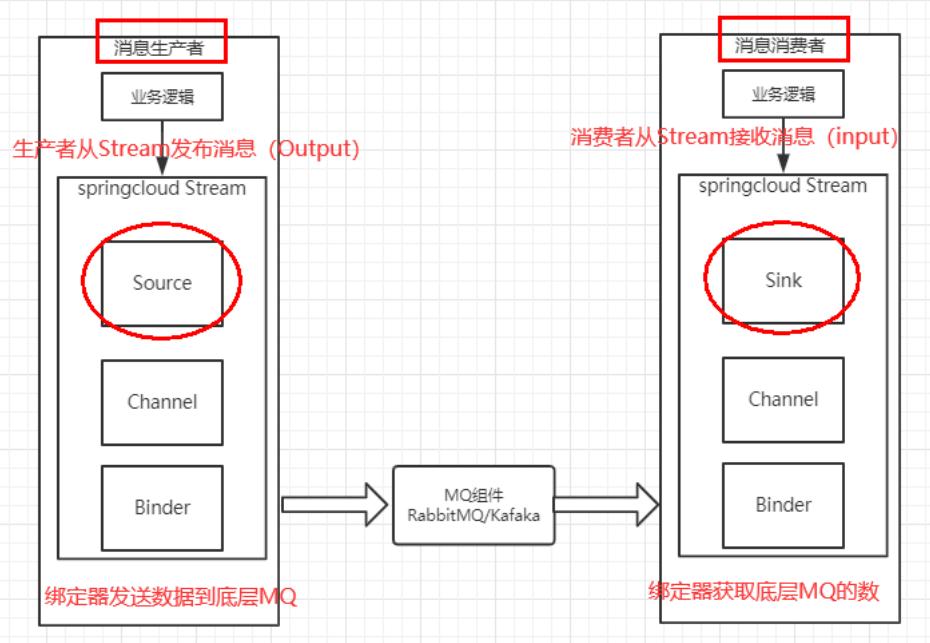


## Stream

### 基本介绍

Spring Cloud Stream 是一个构建消息驱动微服务的框架, 通过定义绑定器 Binder 作为中间层, 实现了应用程序与消息中间件细节之间的隔离, 屏蔽底层消息中间件的差异, 降低切换成本, 统一消息的编程模型

Stream 中的消息通信方式遵循了发布订阅模式, Binder 可以生成 Binding 用来绑定消息容器的生产者和消费者, Binding 有两种类型 Input 和 Output, Input 对应于消费者 (消费者从 Stream 接收消息), Output 对应于生产者 (生产者从 Stream 发布消息)



- Binder: 连接中间件
- Channel: 通道，是队列 Queue 的一种抽象，在消息通讯系统中实现存储和转发的媒介，通过 Channel 对队列进行配置
- Source、Sink: 生产者和消费者

中文手册: <https://m.wang1314.com/doc/webapp/topic/20971999.html>

## 基本使用

Binder 是应用与消息中间件之间的封装，目前实现了 Kafka 和 RabbitMQ 的 Binder，可以动态的改变消息类型（Kafka 的 Topic 和 RabbitMQ 的 Exchange），可以通过配置文件实现，常用注解如下：

- @Input: 标识输入通道，接收的消息通过该通道进入应用程序
- @Output: 标识输出通道，发布的消息通过该通道离开应用程序
- @StreamListener: 监听队列，用于消费者队列的消息接收
- @EnableBinding: 信道 Channel 和 Exchange 绑定

生产者发消息模块：

- 引入 pom 依赖: RabbitMQ

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

- application.yml:

```
server:
 port: 8801

spring:
 application:
 name: cloud-stream-provider
 cloud:
 stream:
 binders: # 在此处配置要绑定的rabbitmq的服务信息:
 defaultRabbit: # 表示定义的名称，用于于binding整合
 type: rabbit # 消息组件类型
 environment: # 设置rabbitmq的相关的环境配置
 spring:
 rabbitmq:
 host: localhost
 port: 5672
 username: guest
 password: guest
 bindings: # 服务的整合处理
 output: # 这个名字是一个通道的名称
 destination: studyExchange # 表示要使用的Exchange名称定义
 content-type: application/json # 设置消息类型，本次为json，文本则设置“text/plain”
 binder: defaultRabbit # 设置要绑定的消息服务的具体设置

eureka:
```

```

client: # 客户端进行Eureka注册的配置
 service-url:
 defaultZone: http://localhost:7001/eureka,http://localhost:7002/eureka
instance:
 lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）
 lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）
 instance-id: send-8801.com # 在信息列表时显示主机名称
 prefer-ip-address: true # 访问的路径变为IP地址

```

- 主启动类:

```

@SpringBootApplication
@EnableEurekaClient
public class StreamMQMain8801 {
 public static void main(String[] args) {
 SpringApplication.run(StreamMQMain8801.class, args);
 }
}

```

- 业务类: MessageChannel 的实例名必须是 output, 否则无法启动

```

// 可以理解为定义消息的发送管道Source对应output(生产者), sink对应input(消费者)
@EnableBinding(Source.class)
// @Service: 这里不需要, 不是传统的controller调用service。这个service是和rabbitMQ打交道的
// IMessageProvider 只有一个 send 方法的接口
public class MessageProviderImpl implements IMessageProvider {
 @Resource
 private MessageChannel output; // 消息的发送管道

 @Override
 public String send() {
 String serial = UUID.randomUUID().toString();

 // 创建消息, 通过output这个管道向消息中间件发消息
 this.output.send(MessageBuilder.withPayload(serial).build());
 System.out.println("****serial: " + serial);
 return serial;
 }
}

```

- Controller:

```

@RestController
public class SendMessageController {
 @Resource
 private IMessageProvider messageProvider;

 @GetMapping(value = "/sendMessage")
 public String sendMessage() {
 return messageProvider.send();
 }
}

```

消费者模块: 8802 和 8803 两个消费者

- application.yml: 只标注出与生产者不同的地方

```

server:
 port: 8802

spring:
 application:
 name: cloud-stream-consumer
 cloud:
 stream:
 # ...
 bindings: # 服务的整合处理
 input: # 这个名字是一个通道的名称
 # ...
 binder: { defaultRabbit } # 设置要绑定的消息服务的具体设置

eureka:
 # ...
 instance:
 # ...
 instance-id: receive-8802.com # 在信息列表时显示主机名称

```

- Controller:

```

@Component
@EnableBinding(Sink.class) // 理解为定义一个消息消费者的接收管道
public class ReceiveMessageListener {
 @Value("${server.port}")
 private String serverPort;

 @StreamListener(Sink.INPUT) //输入源：作为一个消息监听者
 public void input(Message<String> message) {
 // 获取到消息
 String messagestr = message.getPayload();
 System.out.println("消费者1号，----->接收到的消息：" + messagestr + "\t port: " + serverPort);
 }
}

```

## 高级特性

重复消费问题：生产者 8801 发送一条消息后，8802 和 8803 会同时收到 8801 的消息

解决方法：微服务应用放置于同一个 group 中，能够保证消息只会被其中一个应用消费一次。不同的组是可以全面消费的（重复消费），同一个组内的多个消费者会发生竞争关系，只有其中一个可以消费

```

bindings:
 input:
 destination: studyExchange
 content-type: application/json
 binder: { defaultRabbit }
 group: seazeann # 设置分组

```

消息持久化问题：

- 停止 8802/8803 并去除掉 8802 的分组 group: seazeann，8801 先发送 4 条消息到 MQ
- 先启动 8802，无分组属性配置，后台没有打出来消息，消息丢失
- 再启动 8803，有分组属性配置，后台打印出来了 MQ 上的消息

## Sleuth

### 基本介绍

Spring Cloud Sleuth 提供了一套完整的分布式请求链路跟踪的解决方案，并且兼容支持了 zipkin

在微服务框架中，一个客户端发起的请求在后端系统中会经过多次不同的服务节点调用来协同产生最后的请求结果，形成一条复杂的分布式服务调用链路，链路中的任何一个环节出现高延时或错误都会引起整个请求最后的失败，所以需要链路追踪

Sleuth 官网：<https://github.com/spring-cloud/spring-cloud-sleuth>

zipkin 下载地址：<https://repo1.maven.org/maven2/io/zipkin/java/zipkin-server/>

## 链路监控

Sleuth 负责跟踪整理，zipkin 负责可视化展示

```
java -jar zipkin-server-2.12.9-exec.jar # 启动 zipkin
```

访问 <http://localhost:9411/zipkin/> 展示交互界面

一条请求链路通过 Trace ID 唯一标识，Span 标识发起的请求信息

- Trace：类似于树结构的 Span 集合，表示一条调用链路，存在唯一 ID 标识
- Span：表示调用链路来源，通俗的理解 Span 就是一次请求信息，各个 Span 通过 ParentID 关联起来

服务生产者模块：

- 引入 pom 依赖：

```
<!--包含了sleuth+zipkin-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

- application.yml:

```
server:
 port: 8001

spring:
 application:
 name: cloud-payment-service
 zipkin:
 base-url: http://localhost:9411
 sleuth:
 sampler:
 #采样率值介于 0 到 1 之间, 1 则表示全部采集
 probability: 1
```

- 业务类:

```
@GetMapping("/payment/zipkin")
public String paymentZipkin() {
 return "hi ,i'am paymentzipkin server fall back, welcome to seazean";
}
```

服务消费者模块:

- application.yml:

```
server:
 port: 80

微服务名称
spring:
 application:
 name: cloud-order-service
 zipkin:
 base-url: http://localhost:9411
 sleuth:
 sampler:
 probability: 1
```

- 业务类:

```
@GetMapping("/comsumer/payment/zipkin")
public String paymentZipkin() {
 String result = restTemplate.getForObject("http://localhost:8001" + "/payment/zipkin/", String.class);
 return result;
}
```

## Alibaba

Spring Cloud Alibaba 致力于提供微服务开发的一站式解决方案，此项目包含开发分布式应用微服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务

- 服务限流降级：默认支持 WebServlet、WebFlux、OpenFeign、RestTemplate、Spring Cloud Gateway、Zuul、Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
- 服务注册与发现：适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持
- 分布式配置管理：支持分布式系统中的外部化配置，配置更改时自动刷新
- 消息驱动能力：基于 Spring Cloud Stream 为微服务应用构建消息驱动能力
- 分布式事务：使用 @GlobalTransactional 注解，高效并且对业务零侵入地解决分布式事务问题
- 阿里云对象存储：阿里云提供的海量、安全、低成本、高可靠的云存储服务
- 分布式任务调度：提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker (schedulerx-client) 上执行

官方文档：<https://github.com/alibaba/spring-cloud-alibaba/blob/master/README-zh.md>

官方手册：<https://spring-cloud-alibaba-group.github.io/github-pages/greenwich/spring-cloud-alibaba.html>

## Nacos

### 基本介绍

Nacos 全称 Dynamic Naming and Configuration Service，一个更易于构建云原生应用的动态服务发现、配置管理和服务的管理平台，Nacos = Eureka + Config + Bus

下载地址: <https://github.com/alibaba/nacos/releases>

启动命令：命令运行成功后直接访问 <http://localhost:8848/nacos>，默认账号密码都是 nacos

```
startup.cmd -m standalone # standalone 代表着单机模式运行，非集群模式
```

关闭命令：

```
shutdown.cmd
```

注册中心对比：C 一致性，A 可用性，P 分区容错性

注册中心	CAP 模型	控制台管理
Eureka	AP	支持
Zookeeper	CP	不支持
Consul	CP	支持
Nacos	AP	支持

切换模式: curl -X PUT '\$NACOS\_SERVER:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=CP'

官网: <https://nacos.io>

### 注册中心

Nacos 作为服务注册中心

服务提供者：

- 引入 pom 依赖：

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

- application.yml：

```
server:
 port: 9001

spring:
 application:
 name: nacos-payment-provider
 cloud:
 nacos:
 discovery:
 server-addr: localhost:8848 #配置Nacos地址，注册到Nacos
 # 做监控需要把这个全部暴露出来
management:
 endpoints:
 web:
 exposure:
 include: '*'
```

- 主启动类：注解是 EnableDiscoveryClient

```
@EnableDiscoveryClient
@SpringBootApplication
public class PaymentMain9001 {
 public static void main(String[] args) {
 SpringApplication.run(PaymentMain9001.class, args);
 }
}
```

- Controller：

```

@RestController
public class PaymentController {
 @Value("${server.port}")
 private String serverPort;

 @GetMapping(value = "/payment/nacos/{id}")
 public String getPayment(@PathVariable("id") Integer id) {
 return "nacos registry, serverPort: " + serverPort + "\t id" + id;
 }
}

```

- 管理后台服务:

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
nacos-payment-provider	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">删除</a>

- 新建一个模块端口是 9002，其他与 9001 服务一样，nacos-payment-provider 的实例数就变为 2

服务消费者:

- application.yml:

```

server:
 port: 83

spring:
 application:
 name: nacos-order-consumer
 cloud:
 nacos:
 discovery:
 server-addr: localhost:8848

消费者将要去访问的微服务名称(注册成功进nacos的微服务提供者)
service-url:
 nacos-user-service: http://nacos-payment-provider

```

- 主启动类:

```

@SpringBootApplication
@EnableDiscoveryClient
public class OrderNacosMain83 {
 public static void main(String[] args) {
 SpringApplication.run(OrderNacosMain83.class, args);
 }
}

```

- 业务类:

```

@Configuration
public class ApplicationContextBean {
 @Bean
 @LoadBalanced // 生产者集群状态下，必须添加，防止找不到实例
 public RestTemplate getRestTemplate() {
 return new RestTemplate();
 }
}

```

```

@RestController
@Slf4j
public class OrderNacosController {
 @Resource
 private RestTemplate restTemplate;
 // 从配置文件中读取 URL
 @Value("${service-url.nacos-user-service}")
 private String serverURL;

 @GetMapping("/consumer/payment/nacos/{id}")
 public String paymentInfo(@PathVariable("id") Long id) {

```

```
 String result = restTemplate.getForObject(serverURL + "/payment/nacos/" + id, String.class);
 return result;
 }
}
```

## 配置中心

### 基础配置

把配置文件写进 Nacos，然后再用 Nacos 做 config 这样的功能，直接从 Nacos 上抓取服务的配置信息

在 Nacos 中，dataId 的完整格式如下 \${prefix}-\${spring.profiles.active}.\${file-extension}

- prefix：默认为 spring.application.name 的值，也可以通过配置项 spring.cloud.nacos.config.prefix 来配置
- spring.profiles.active：当前环境对应的 profile，当该值为空时，dataId 的拼接格式变成 \${prefix}.\${file-extension}
- file-extension：配置内容的数据格式，可以通过配置项 spring.cloud.nacos.config.file-extension 来配置，目前只支持 properties 和 yaml 类型（不是 yaml）

构建流程：

- 引入 pom 依赖：

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

- 配置两个 yaml 文件：配置文件的加载是存在优先级顺序的，bootstrap 优先级高于 application

bootstrap.yaml：全局配置

```
nacos配置
server:
 port: 3377

spring:
 application:
 name: nacos-config-client
 cloud:
 nacos:
 discovery:
 server-addr: localhost:8848 #Nacos服务注册中心地址
 config:
 server-addr: localhost:8848 #Nacos作为配置中心地址
 file-extension: yaml #指定yaml格式的配置
${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}
```

application.yaml：服务独立配置，表示服务要去配置中心找名为 nacos-config-client-dev.yaml 的文件

```
spring:
 profiles:
 active: dev # 表示开发环境
```

- 主启动类：

```
@SpringBootApplication
@EnableDiscoveryClient
public class NacosConfigClientMain3377 {
 public static void main(String[] args) {
 SpringApplication.run(NacosConfigClientMain3377.class, args);
 }
}
```

- 业务类：@RefreshScope 注解使当前类下的配置支持 Nacos 的动态刷新功能

```
@RestController
@RefreshScope
public class ConfigClientController {
 @Value("${config.info}")
 private String configInfo;

 @GetMapping("/config/info")
 public String getConfigInfo() {
 return configInfo;
 }
}
```

- 新增配置，然后访问 <http://localhost:3377/config/info>

The screenshot shows the '新建配置' (Create Configuration) page in the Nacos UI. On the left, there's a sidebar with sections like '配置管理', '服务管理', and '集群管理'. The main area has fields for 'Data ID' (nacos-config-client-dev.yaml), 'Group' (DEFAULT\_GROUP), and 'Content' (YAML code). A red box highlights the 'Content' input field.

## 分类配置

分布式开发中的多环境多项目管理问题，Namespace 用于区分部署环境，Group 和 DataID 逻辑上区分两个目标对象

The screenshot shows the '命名空间' (Namespace) section in the Nacos UI. The sidebar has a '命名空间' section. The main area shows a list of namespaces: 'public' (selected), 'dev', and 'test'. A red arrow points from the '命名空间' section in the sidebar to the 'public' namespace in the list. The table below shows configuration details for each namespace.

Data ID	Group	归属应用:	操作
nacos-config-client-dev.yaml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
nacos-config-client-test.yaml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
nacos-config-client-info.yaml	DEV_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
nacos-config-client-info.yaml	TEST_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>

Namespace 默认 public，主要用来实现隔离，图示三个开发环境

Group 默认是 DEFAULT\_GROUP，Group 可以把不同的微服务划分到同一个分组里面去

## 加载配置

DataID 方案：指定 `spring.profile.active` 和配置文件的 DataID 来使不同环境下读取不同的配置

Group 方案：通过 Group 实现环境分区，在 config 下增加一条 Group 的配置即可

Namespace 方案：

```

server:
 port: 3377

spring:
 application:
 name: nacos-config-client
 cloud:
 nacos:
 discovery:
 server-addr: localhost:8848 #Nacos服务注册中心地址
 config:
 server-addr: localhost:8848 #Nacos作为配置中心地址
 file-extension: yaml #指定yaml格式的配置
 group: DEV_GROUP
 namespace: 95d44530-a4a6-4ead-98c6-23d192cee298

```

NACOS 1.1.4
命名空间
新建命名空间

▼ 配置管理

- [配置列表](#)
- [历史版本](#)
- [监听查询](#)

▼ 服务管理

命名空间名称	命名空间ID	配置数	操作
public(保留空间)		4 / 200	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>
dev	95d44530-a4a6-4ead-98c6-23d192cee298	0 / 200	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>
test	ec4699c7-16c8-443c-bc4d-7163a72f25d1	0 / 200	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>

## 集群架构

集群部署参考官方文档， Nacos 支持的三种部署模式：

- 单机模式：用于测试和单机使用
- 集群模式：用于生产环境，确保高可用
- 多集群模式：用于多数据中心场景

集群部署文档：<https://nacos.io/zh-cn/docs/v2/guide/admin/cluster-mode-quick-start.html>

默认 Nacos 使用嵌入式数据库 derby 实现数据的存储，重启 Nacos 后配置文件不会消失，但是多个 Nacos 节点数据存储存在一致性问题，每个 Nacos 都有独立的嵌入式数据库，所以 Nacos 采用了集中式存储的方式来支持集群化部署，目前只支持 MySQL 的存储

Windows 下 Nacos 切换 MySQL 存储：

- 在 Nacos 安装目录的 conf 目录下找到一个名为 `nacos-mysql.sql` 的脚本并执行
- 在 conf 目录下找到 `application.properties`，增加如下数据

```
spring.datasource.platform=mysql

db.num=1
db.url= jdbc:mysql://127.0.0.1:3306/nacos_config?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true
db.user=username
db.password=password
```

- 重新启动 Nacos，可以看到是个全新的空记录界面

Linux 参考：<https://www.yuque.com/mrlinxi/pxvr4g/rnahsn#dPvMy>

## Sentinel

### 基本介绍

Sentinel 是面向分布式、多语言异构化服务架构的流量治理组件

Sentinel 分为两个部分：

- 核心库（Java 客户端）不依赖任何框架/库，能够运行于 Java 8 及以上的版本的运行时环境
- 控制台（Dashboard）主要负责管理推送规则、监控、管理机器信息等

下载到本地，运行命令：`java -jar sentinel-dashboard-1.8.2.jar`（要求 Java8，且 8080 端口不能被占用），访问 <http://localhost:8080/>，账号密码均为 sentinel

官网：<https://sentinelguard.io>

下载地址：<https://github.com/alibaba/Sentinel/releases>

## 基本使用

构建演示工程：

- 引入 pom 依赖：

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

- application.yml: sentinel.transport.port 端口配置会在应用对应的机器上启动一个 HTTP Server, 该 Server 与 Sentinel 控制台做交互。比如 Sentinel 控制台添加了 1 个限流规则, 会把规则数据 Push 给 Server 接收, Server 再将规则注册到 Sentinel 中

```

server:
 port: 8401

spring:
 application:
 name: cloudalibaba-sentinel-service
 cloud:
 nacos:
 discovery:
 server-addr: localhost:8848 # Nacos 服务注册中心地址【需要启动Nacos8848】
 sentinel:
 transport:
 # 配置Sentinel dashboard地址
 dashboard: localhost:8080
 # 默认8719端口, 假如被占用会自动从8719开始依次+1扫描, 直至找到未被占用的端口
 port: 8719

 management:
 endpoints:
 web:
 exposure:
 include: '*'

```

- 主启动类:

```

@EnableDiscoveryClient
@SpringBootApplication
public class SentinelMainApp8401 {
 public static void main(String[] args) {
 SpringApplication.run(SentinelMainApp8401.class, args);
 }
}

```

- 流量控制 Controller:

```

@RestController
@Slf4j
public class FlowLimitController {
 @GetMapping("/testA")
 public String testA() {
 return "-----testA";
 }

 @GetMapping("/testB")
 public String testB() {
 return "-----testB";
 }
}

```

- Sentinel 采用懒加载机制, 需要先访问 <http://localhost:8401/testA>, 控制台才能看到

## 流控规则

流量控制规则 FlowRule: 同一个资源可以同时有多个限流规则

- 资源名 resource: 限流规则的作用对象, Demo 中为 testA
- 针对资源 limitApp: 针对调用者进行限流, 默认为 default 代表不区分调用来源
- 阈值类型 grade: QPS 或线程数模式
- 单机阈值 count: 限流阈值
- 流控模式 strategy: 调用关系限流策略
  - 直接: 资源本身达到限流条件直接限流
  - 关联: 当关联的资源达到阈值时, 限流自身
  - 链路: 只记录指定链路上的流量, 从入口资源进来的流量
- 流控效果 controlBehavior:
  - 快速失败: 直接失败, 抛出异常
  - Warm Up: 冷启动, 根据 codeFactory (冷加载因子, 默认 3) 的值, 从 count/codeFactory 开始缓慢增加, 给系统预热时间
  - 排队等待: 匀速排队, 让请求以匀速的方式通过, 阈值类型必须设置为 QPS, 否则无效



通过调用 `SystemRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则：

```
private void initSystemProtectionRule() {
 List<SystemRule> rules = new ArrayList<>();
 SystemRule rule = new SystemRule();
 rule.setHighestSystemLoad(10);
 rules.add(rule);
 SystemRuleManager.loadRules(rules);
}
```

详细内容参考文档：<https://sentinelguard.io/zh-cn/docs/flow-control.html>

## 降级熔断

Sentinel 熔断降级会在调用链路中某个资源出现不稳定状态时，对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都自动熔断（默认行为是抛出 `DegradeException`）

Sentinel 提供以下几种熔断策略：

- 资源名 resource：限流规则的作用对象，Demo 中为 testA
- 熔断策略 grade：
  - 慢调用比例 (SLOW\_REQUEST\_RATIO)：以慢调用比例作为阈值，需要设置允许的慢调用 RT（即最大的响应时间），请求的响应时间大于该值则统计为慢调用。当单位统计时长 (statIntervalMs) 内请求数目大于设置的最小请求数目，并且慢调用的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态 (HALF-OPEN 状态)，若接下来的一个请求响应时间小于设置的慢调用 RT 则结束熔断，若大于设置的慢调用 RT 则会再次被熔断
  - 异常比例 (ERROR\_RATIO)：当单位统计时长内请求数目大于设置的最小请求数目，并且异常的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态，若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断。异常比率的阈值范围是 [0.0, 1.0]，代表 0% - 100%
  - 异常数 (ERROR\_COUNT)：当单位统计时长内的异常数目超过阈值之后会自动进行熔断。经过熔断时长后熔断器会进入探测恢复状态，若接下来的一个请求成功完成（没有错误）则结束熔断，否则会再次被熔断
- 单机阈值 count：慢调用比例模式下为慢调用临界 RT；异常比例/异常数模式下为对应的阈值
- 熔断时长 timeWindow：单位为 s
- 最小请求数 minRequestAmount：熔断触发的最小请求数，请求数小于该值时即使异常比率超出阈值也不会熔断，默认 5
- 统计时长 statIntervalMs：单位统计时长
- 慢调用比例阈值 slowRatioThreshold：仅慢调用比例模式有效

新增熔断规则

资源名	/testA			
熔断策略	<input checked="" type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input type="radio"/> 异常数			
最大 RT	RT (毫秒)	比例阈值	取值 [0.0, 1.0]	
熔断时长	熔断时长(s)	s	最小请求数	5
统计时长	1000	ms		
<input type="button" value="新增并继续添加"/> <input type="button" value="新增"/> <input type="button" value="取消"/>				

注意异常降级仅针对业务异常，对 Sentinel 限流降级本身的异常 BlockException 不生效，为了统计异常比例或异常数，需要通过 `Tracer.trace(ex)` 记录业务异常或者通过 `@SentinelResource` 注解会自动统计业务异常

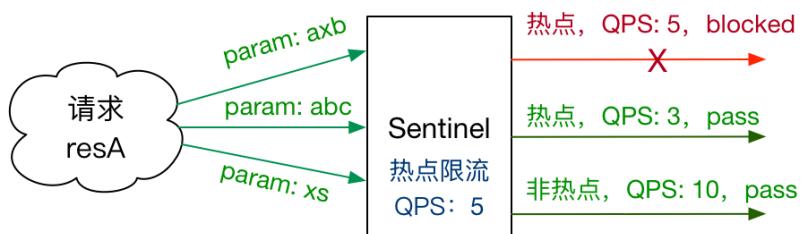
```
Entry entry = null;
try {
 entry = SphU.entry(resource);

 // Write your biz code here.
 // <><BIZ CODE>>
} catch (Throwable t) {
 if (!BlockException.isBlockException(t)) {
 Tracer.trace(t);
 }
} finally {
 if (entry != null) {
 entry.exit();
 }
}
```

详细内容参考文档: <https://sentinelguard.io/zh-cn/docs/circuit-breaking.html>

## 热点限流

热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流，Sentinel 利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控



引入 `@SentinelResource` 注解: <https://sentinelguard.io/zh-cn/docs/annotation-support.html>

- value: Sentinel 资源名，默认为请求路径，这里 value 的值可以任意写，但是约定与 Restful 地址一致
- blockHandler: 表示触发了 Sentinel 中配置的流控规则，就会调用兜底方法 `def_testHotKey`
- blockHandlerClass: 如果设置了该值，就会去该类中寻找 blockHandler 方法
- fallback: 用于在抛出异常的时候提供 fallback 处理逻辑

说明: fallback 对应服务降级（服务出错了需要有个兜底方法），blockHandler 对应服务熔断（服务不可用的兜底方法）

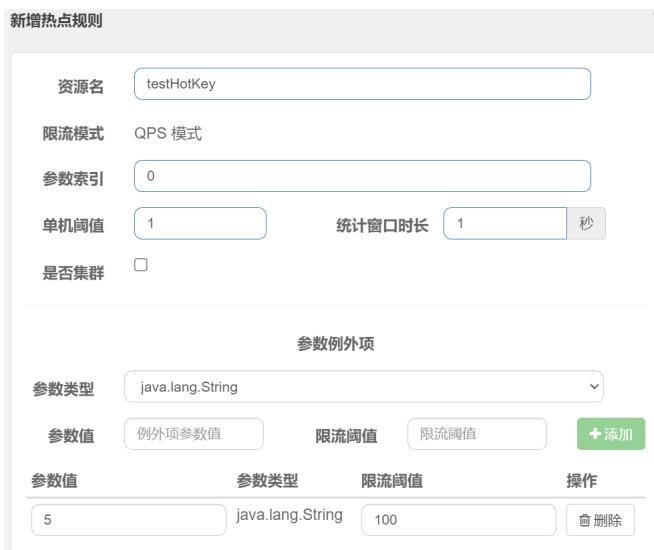
```

@GetMapping("/testHotKey")
@SentinelResource(value = "testHotKey", blockHandler = "del_testHotKey")
public String testHotKey(@RequestParam(value = "p1", required = false) String p1,
 @RequestParam(value = "p2", required = false) String p2) {
 return "-----testHotKey";
}

// 自定义的兜底方法，必须是 BlockException
public String del_testHotKey(String p1, String p2, BlockException e) {
 return "不用默认的兜底提示 Blocked by Sentinel(flow limiting), 自定义提示: del_testHotKeyo.";
}

```

图示设置 p1 参数限流，规则是 1s 访问 1 次，当 p1=5 时 QPS > 100，只访问 p2 不会出现限流 <http://localhost:8401/testHotKey?p2=b>



- 参数索引 paramIdx: 热点参数的索引，图中索引 0 对应方法中的 p1 参数
- 参数例外项 paramFlowItemList: 针对指定的参数值单独设置限流阈值，不受 count 阈值的限制，**仅支持基本类型和字符串类型**

说明：@SentinelResource 只管控制台配置规则问题，出现运行时异常依然会报错

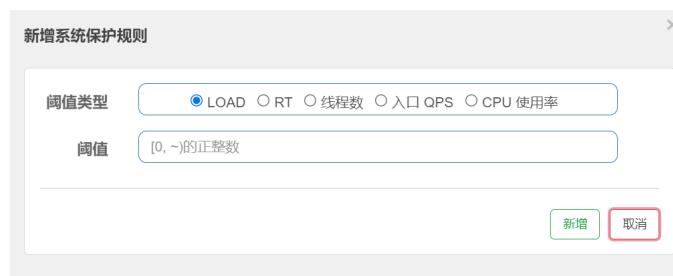
详细内容参考文档：<https://sentinelguard.io/zh-cn/docs/parameter-flow-control.html>

## 系统规则

Sentinel 系统自适应保护从整体维度对应用入口流量进行控制，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性

系统规则支持以下的阈值类型：

- Load (仅对 Linux/Unix-like 机器生效)：当系统 load1 超过阈值，且系统当前的并发线程数超过系统容量时才会触发系统保护，系统容量由系统的 `maxQps * minRt` 计算得出，设定参考值一般是 `CPU cores * 2.5`
- RT：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒
- 线程数：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护
- 入口 QPS：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护
- CPU usage：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0）



详细内容参考文档：<https://sentinelguard.io/zh-cn/docs/system-adaptive-protection.html>

## 服务调用

消费者需要进行服务调用

- 引入 pom 依赖:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

- application.yml: 激活 Sentinel 对 Feign 的支持

```
feign:
 sentinel:
 enabled: true
```

- 主启动类: 加上 @EnableFeignClient 注解开启 OpenFeign

- 业务类:

```
// 指明调用失败的兜底方法在PaymentFallbackService, 使用 fallback 方式是无法获取异常信息的,
// 如果想要获取异常信息, 可以使用 fallbackFactory 参数
@EnableFeignClient(value = "nacos-payment-provider", fallback = PaymentFallbackService.class)
public interface PaymentFeignService {
 // 去生产则服务中找相应的接口, 方法签名一定要和生产者中 controller 的一致
 @GetMapping(value = "/paymentSQL/{id}")
 public CommonResult<Payment> paymentSQL(@PathVariable("id") Long id);
}
```

```
@Component //不要忘记注解, 降级方法
public class PaymentFallbackService implements PaymentFeignService {
 @Override
 public CommonResult<Payment> paymentSQL(Long id) {
 return new CommonResult<>(444,"服务降级返回,没有该流水信息-----PaymentFallbackse
```

## 持久化

配置持久化:

- 引入 pom 依赖:

```
<!--SpringCloud alibaba sentinel-datasource-nacos 后续做持久化用到-->
<dependency>
 <groupId>com.alibaba.csp</groupId>
 <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

- 添加 Nacos 数据源配置:

```
server:
 port: 8401

spring:
 application:
 name: cloudalibaba-sentinel-service
 cloud:
 nacos:
 discovery:
 server-addr: localhost:8848 #Nacos服务注册中心地址
 sentinel:
 transport:
 dashboard: localhost:8080
 port: 8719
 # 关闭默认收敛所有URL的入口context, 不然链路限流不生效
 web-context-unify: false
 # filter:
 # enabled: false # 关闭自动收敛

 #持久化配置
 datasource:
 ds1:
 nacos:
 server-addr: localhost:8848
 dataId: cloudalibaba-sentinel-service
 groupId: DEFAULT_GROUP
```

```
data-type: json
rule-type: flow
```

## Seata

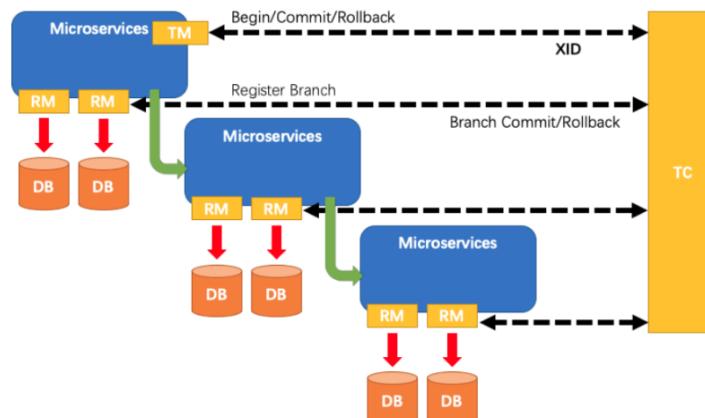
### 分布事物

一个分布式事务过程，可以用分布式处理过程的一 ID + 三组件模型来描述：

- XID (Transaction ID): 全局唯一的事务 ID，在这个事务 ID下的所有事务会被统一控制
- TC (Transaction Coordinator): 事务协调者，维护全局和分支事务的状态，驱动全局事务提交或回滚
- TM (Transaction Manager): 事务管理器，定义全局事务的范围，开始全局事务、提交或回滚全局事务
- RM (Resource Manager): 资源管理器，管理分支事务处理的资源，与 TC 交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚

典型的分布式事务流程：

1. TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID
2. XID 在微服务调用链路的上下文中传播（也就是在多个 TM, RM 中传播）
3. RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖
4. TM 向 TC 发起针对 XID 的全局提交或回滚决议
5. TC 调度 XID 下管辖的全部分支事务完成提交或回滚请求



### 基本配置

Seata 是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。

下载 seata-server 文件修改 conf 目录下的配置文件

- file.conf: 自定义事务组名称、事务日志存储模式为 db、数据库连接信息

**事务分组**: seata 的资源逻辑，可以按微服务的需要，在应用程序（客户端）对自行定义事务分组，每组取一个名字

```
1 service {
2 #transaction service group mapping
3 vgroup_mapping.my_test_tx_group = "default" 自定义事务组名称
4 #only support when registry.type=file, please don't set multiple addresses
5 default.groupList = "127.0.0.1:8091"
6 #disable seata
7 disableGlobalTransaction = false
8 }
9
10 ## transaction log store, only used in seata-server
11 store {
12 ## store mode: file, db
13 mode = "file"
14
15 ## file store property
16 file {
17 ## store location dir
18 dir = "sessionStore"
19 }
20
21 ## database store property
22 db {
23 ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc
24 dataSource = "dbcp"
25 ## mysql/oracle/h2/oceanbase etc.
26 db-type = "mysql"
27 driver-class-name = "com.mysql.jdbc.Driver"
28 url = "jdbc:mysql://127.0.0.1:3306/seata"
29 user = "mysql"
30 password = "mysql"
31 }
32 }
```

**填写 MySQL 配置信息**

- 数据库新建库 seata，建表 db\_store.sql 在 <https://github.com/seata/seata/tree/2.x/script/server/db> 目录里面
- registry.conf: 指明注册中心为 Nacos，及修改 Nacos 连接信息

```

1 registry {
2 # file、nacos、eureka、redis、zk、consul、etcd3、sofa
3 type = "nacos"
4
5 nacos {
6 serverAddr = "localhost:8848"
7 namespace =
8 cluster = "default"
9 }

```

启动 Nacos 和 Seata，如果 DB 报错，需要把 lib 文件夹下 mysql-connector-java-5.1.30.jar 删除，替换为自己 MySQL 连接器版本

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阀值	操作
cloudalibaba-sentinel-service	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
serverAddr	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

每页显示: 10 < 上一页 1 下一页 >

官网: <https://seata.io>

下载地址: <https://github.com/seata/seata/releases>

基本介绍: <https://seata.io/zh-cn/docs/overview/what-is-seata.html>

## 基本使用

两个注解:

- Spring 提供的本地事务: @Transactional
- Seata 提供的全局事务: @GlobalTransactional

搭建简单 Demo:

- 创建 UNDO\_LOG 表: SEATA AT 模式需要 UNDO\_LOG 表, 如果一个模块的事务提交了, Seata 会把提交了哪些数据记录到 undo\_log 表中, 如果这时 TC 通知全局事务回滚, 那么 RM 就从 undo\_log 表中获取之前修改了哪些资源, 并根据这个表回滚

```
-- 注意此处0.3.0+ 增加唯一索引 ux_undo_log
CREATE TABLE `undo_log` (
 `id` bigint(20) NOT NULL AUTO_INCREMENT,
 `branch_id` bigint(20) NOT NULL,
 `xid` varchar(100) NOT NULL,
 `context` varchar(128) NOT NULL,
 `rollback_info` longblob NOT NULL,
 `log_status` int(11) NOT NULL,
 `log_created` datetime NOT NULL,
 `log_modified` datetime NOT NULL,
 `ext` varchar(100) DEFAULT NULL,
 PRIMARY KEY (`id`),
 UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

- 引入 pom 依赖:

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
 <version>${spring-cloud-alibaba.version}</version>
</dependency>
```

- application.yml:

```
spring:
 application:
 name: seata-order-service
 cloud:
 alibaba:
 seata:
 # 自定义事务组名称需要与seata-server中file.conf中配置的事务组ID对应
 # vgroup_mapping.my_test_tx_group = "my_group"
 tx-service-group: my_group
 nacos:
 discovery:
 server-addr: localhost:8848
 datasource:
 driver-class-name: com.mysql.cj.jdbc.Driver
 url: jdbc:mysql://localhost:3306/seata_order?useUnicode=true&characterEncoding=UTF-8&useSSL=false&serverTimezone=UTC
 username: root
```

```
password: 123456
```

- 构建三个服务：

```
// 仓储服务
public interface StorageService {
 // 扣除存储数量
 void deduct(String commodityCode, int count);
}

// 订单服务
public interface OrderService {
 // 创建订单
 Order create(String userId, String commodityCode, int orderCount);
}

// 帐户服务
public interface AccountService {
 // 从用户账户中借出
 void debit(String userId, int money);
}
```

- 业务逻辑：增加 @GlobalTransactional 注解

```
public class OrderServiceImpl implements OrderService {
 @Resource
 private OrderDAO orderDAO;
 @Resource
 private AccountService accountService;

 @Transactional(rollbackFor = Exception.class)
 public Order create(String userId, String commodityCode, int orderCount) {
 int orderMoney = calculate(commodityCode, orderCount);
 // 账户扣钱
 accountService.debit(userId, orderMoney);

 Order order = new Order();
 order.userId = userId;
 order.commodityCode = commodityCode;
 order.count = orderCount;
 order.money = orderMoney;

 return orderDAO.insert(order);
 }
}
```

```
public class BusinessServiceImpl implements BusinessService {
 @Resource
 private StorageService storageService;
 @Resource
 private OrderService orderService;

 // 采购，涉及多服务的分布式事务问题
 @GlobalTransactional
 @Transactional(rollbackFor = Exception.class)
 public void purchase(String userId, String commodityCode, int orderCount) {
 storageService.deduct(commodityCode, orderCount);
 orderService.create(userId, commodityCode, orderCount);
 }
}
```

详细示例参考：<https://github.com/seata/seata-samples/tree/master/springcloud-nacos-seata>