基础

1. spring支持IOC（控制反转）、AOP（面向切面编程）
2. **IoC（Inversion of Control:控制反转）** 是一种设计思想，而不是一个具体的技术实现。IoC 的思想就是将原本在程序中手动创建对象的控制权，交由 Spring 框架来管理
3. **Bean** 代指的就是那些被 IoC 容器所管理的对象。

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

4. 声明为Bean的注解
    1. Component：通用的注解，可标注任意类为 `Spring` 组件
    2. Repository：对应持久层即 Dao 层，主要用于数据库相关操作
    3. Service：对应服务层
    4. Controller：对应 Spring MVC 控制层
5. @component 和 @Bean 的区别是什么?
    - `@Component` 注解作用于**类**，而`@Bean`注解作用于**方法**
    - `@Component`通常是通过**类路径扫描**来自动侦测以及自动装配到 Spring 容器中

      `@Bean` 注解通常是我们在**标有该注解的方法中定义产生这个 bean**,`@Bean`告诉了 Spring 这是某个类的实例，当我需要用它的时候还给我
    - `@Bean` 注解比 `@Component` 注解的自定义性更强，而且很多地方我们只能通过 `@Bean` 注解来注册bean

      ```
      @Bean
      public OneService getService(status) {
          case (status)  {
              when 1:
                      return new serviceImpl1();
              when 2:
                      return new serviceImpl2();
              when 3:
                      return new serviceImpl3();
          }
      }
      //此例子无法使用@Component实现
      ```

6. 进入@Bean的注解有哪些
    1. `@Autowired`
    2. `@Resource`
    3. `@Inject`

7. @Autowired 和 @Resource 的区别是什么?
    - `@Autowired`默认byType；`@Resource`默认byName
    - `@Autowired` 是 Spring 提供的注解，`@Resource` 是 JDK 提供的注解
    - `@Autowired` **支持在构造函数**、方法、字段和**参数**上使用。`@Resource` 主要用于**字段和方法**上的注入，不支持在构造函数或参数上使用。

    `@Autowired` 属于 Spring 内置的注解，**默认的注入方式为`byType`**（根据类型进行匹配）,会优先根据接口类型去匹配并注入 Bean （接口的实现类）。**当一个接口存在多个实现类的话**，`byType`这种方式就无法正确注入对象。这种情况下，**注入方式会变为 `byName`**（根据名称进行匹配）。

    ```
    //假设有SmsService 接口有两个实现类：SmsServiceImpl1和 SmsServiceImpl2，且它们都已经被 Spring 容器所管理

    // 报错，byName 和 byType 都无法匹配到 bean
    @Autowired
    private SmsService smsService;
    // 正确注入方式1： SmsServiceImpl1 对象对应的 bean
    @Autowired
    private SmsService smsServiceImpl1;
    // 正确注入方式2： SmsServiceImpl1 对象对应的 bean
    // smsServiceImpl1 就是我们上面所说的名称
    @Autowired
    @Qualifier(value = "smsServiceImpl1")
    private SmsService smsService;
    ```

    `@Resource`属于 JDK 提供的注解，**默认注入方式为 byName**。如果无法通过名称匹配到对应的 Bean 的话，注入方式会变为byType

8. Bean的作用域

    singleton : IoC 容器中只有唯一的 bean 实例。Spring 中的 bean 默认都是单例的，是对单例设计模式的应用。

    prototype : 每次获取都会创建一个新的 bean 实例。也就是说，连续 `getBean()` 两次，得到的是不同的 Bean 实例。

    request （仅 Web 应用可用）: 每一次 HTTP 请求都会产生一个新的 bean（请求 bean），该 bean 仅在当前 HTTP request 内有效。

    session （仅 Web 应用可用）: 每一次来自新 session 的 HTTP 请求都会产生一个新的 bean（会话 bean），该 bean 仅在当前 HTTP session 内有效。

    application/global-session （仅 Web 应用可用）： 每个 Web 应用在启动时创建一个 Bean（应用 Bean），该 bean 仅在当前应用启动时间内有效。

    websocket （仅 Web 应用可用）： 每一次 WebSocket 会话产生一个新的 bean

9. Bean是否线程安全

    几乎所有场景的 Bean 作用域都是使用默认的 singleton ，重点关注 singleton 作用域即可。

    singleton 作用域下，IoC 容器中只有唯一的 bean 实例，可能会存在资源竞争问题。（prototype 作用域下，不存在线程安全问题）如果这个 bean 是有状态的话，那就存在线程安全问题。**不过，大部分 Bean 实际都是无状态**（没有定义可变的成员变量，比如 Dao、Service）

> 解决办法：两种
>
> 1. 在 Bean 中尽量避免定义可变的成员变量。
> 2. 在类中定义一个 `ThreadLocal` 成员变量，将需要的可变成员变量保存在 `ThreadLocal` 中（推荐的一种方式）

10. Bean的生命周期
    1. **创建**Bean的实例：使用 Java 反射 API 来创建 Bean 的实例
    2. Bean的属性**赋值**/填充：为 Bean **设置相关属性和依赖**，例如 *`@Autowired` 等注解注入的对象*、*`@Value` 注入的值*、*`setter` 方法或构造函数注入依赖和值*、*`@Resource` 注入的各种资源*
    3. Bean**初始化**：
        1. 如果实现了其他 `*.Aware` 接口，就调用相应的方法。如：实现了 `BeanNameAware` 接口，调用 `setBeanName()` 方法
        2. 实现了 `InitializingBean` 接口，执行 `afterPropertiesSet()` 方法
        3. 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法和 `postProcessAfterInitialization()` 方法
    4. **销毁**Bean：销毁并不是说要立马把 Bean 给销毁掉，而是把 Bean 的销毁方法先记录下来

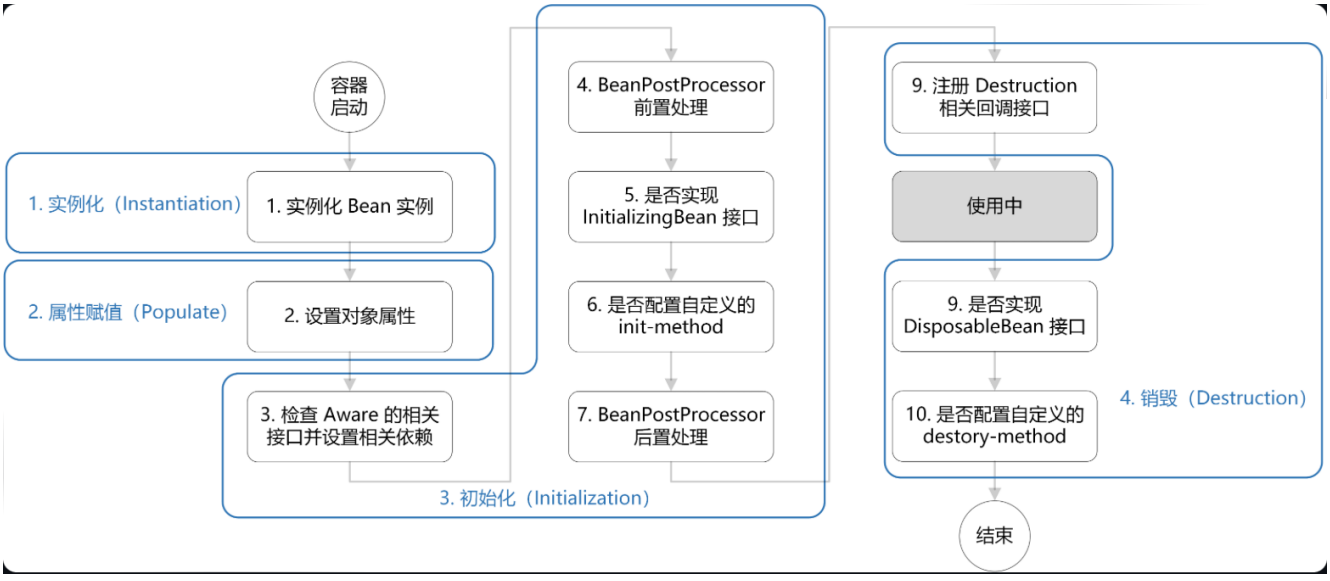
> 整体上可以简单分为四步：**实例化 —> 属性赋值 —> 初始化 —> 销毁**。
>
> 初始化这一步涉及到的步骤比较多，包含 **`Aware` 接口的依赖注入**、**`BeanPostProcessor` 在初始化前后的处理**以及 **`InitializingBean` 和 `init-method`** 的初始化操作。
>
> 销毁这一步会注册相关销毁回调接口，最后**通过 `DisposableBean` 和 `destory-method` 进行销毁**。

- Aware 接口能让 Bean 能拿到 Spring 容器资源

  `BeanNameAware`：注入当前 bean 对应 beanName；

  `BeanClassLoaderAware`：注入加载当前 bean 的 ClassLoader；

  `BeanFactoryAware`：注入当前 `BeanFactory` 容器的引用



AOP

术语	含义
目标(Target)	被通知的对象
代理(Proxy)	向目标对象应用通知之后创建的代理对象
连接点(JoinPoint)	目标对象的所属类中，定义的所有方法均为连接点
切入点(Pointcut)	被切面拦截 / 增强的连接点（切入点一定是连接点，连接点不一定是切入点）
通知(Advice)	增强的逻辑 / 代码，也即拦截到目标对象的连接点之后要做的事情
切面(Aspect)	切入点(Pointcut)+通知(Advice)
Weaving(织入)	将通知应用到目标对象，进而生成代理对象的过程动作

1. **Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。**

   Spring AOP 已经集成了 AspectJ。当切面太多的话，最好选择 AspectJ ，它比 Spring AOP 快很多

2. 通知类型

   Before（前置通知）

   After （后置通知）

   AfterReturning（返回通知）

   AfterThrowing（异常通知）

   Around （环绕通知）

3. 多个切面的执行顺序

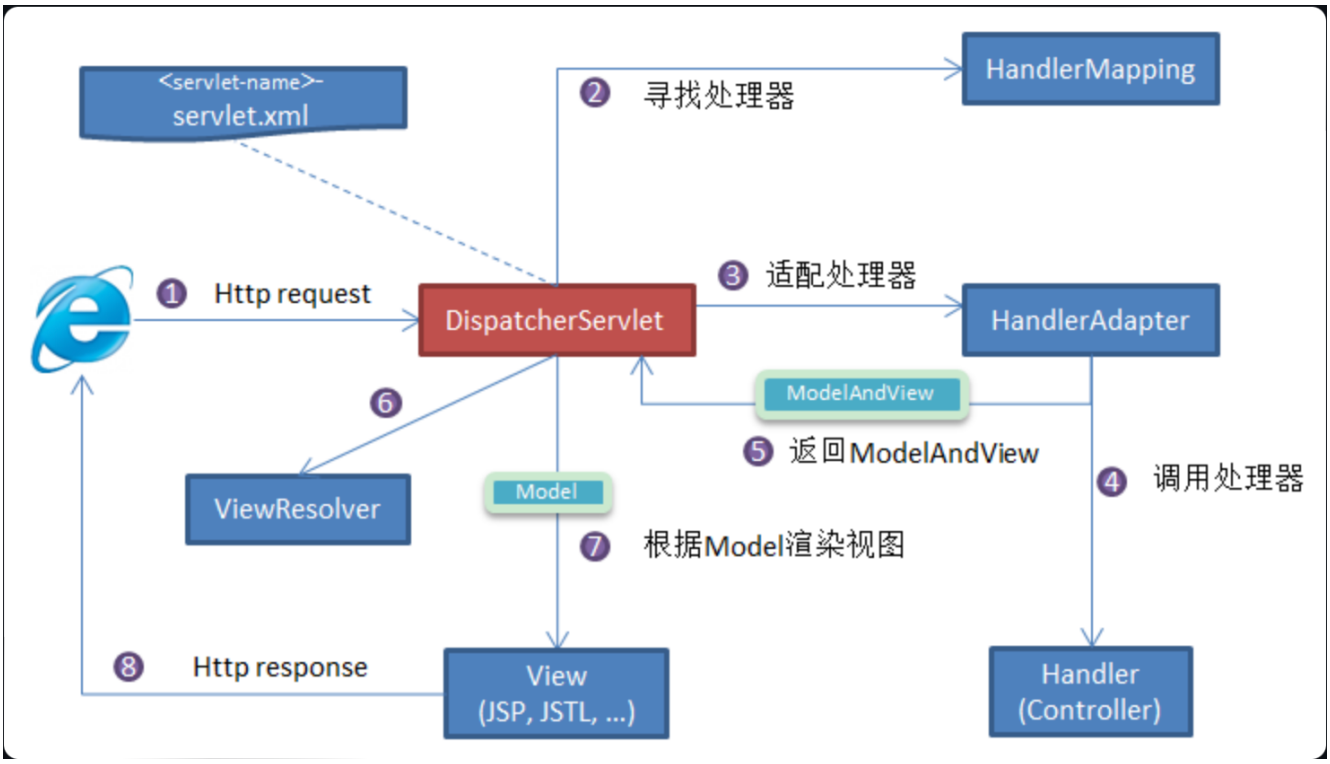
   1. @Order：值越小优先级越高。
   2. 切面实现`Ordered`接口，重写`getOrder`方法。

```
@Component
@Aspect
public class LoggingAspect implements Ordered {//
    @Override
    public int getOrder() {
        // 返回值越小优先级越高
        return 1;
    }
}
```

Spring MVC

1. 核心组件有哪些
   - **`DispatcherServlet`：核心的中央处理器**，负责接收请求、分发，并给予客户端响应。
   - **`HandlerMapping`：处理器映射器，根据 URL 去匹配**查找能处理的`Handler`，并会将请求涉及到的拦截器和`Handler`一起封装。

- **`HandlerAdapter`**：**处理器适配器**，根据 `HandlerMapping` 找到的 `Handler` ，**适配执行对应的 `Handler`**；
- **`Handler`**：**请求处理器，处理实际请求**的处理器。返回ModelAndView
- **`ViewResolver`**：**视图解析器**，根据 `Handler` 返回的逻辑视图 / 视图，**解析并渲染真正的视图**，并传递给 `DispatcherServlet` 响应客户端



Spring环依赖

Spring 框架通过使用三级缓存（其实就是三个Map）来解决这个问题，确保即使在循环依赖的情况下也能正确创建 Bean。

一级缓存（singletonObjects）：存放**最终形态**的 Bean（已经实例化、属性填充、初始化），单例池，为“Spring 的单例属性”而生。一般情况我们获取 Bean 都是从这里获取的，但是并不是所有的 Bean 都在单例池里面，例如原型 Bean 就不在里面。

二级缓存（earlySingletonObjects）：存放**过渡** Bean（半成品，尚未属性填充），也就是三级缓存中 `ObjectFactory` 产生的对象，与三级缓存配合使用的，可以防止 AOP 的情况下，每次调用 `ObjectFactory#getObject()` 都是会产生新的代理对象的。

三级缓存（singletonFactories）：存放 `ObjectFactory`，`ObjectFactory` 的 `getObject()` 方法（最终调用的是 `getEarlyBeanReference()` 方法）可以**生成原始 Bean 对象或者代理对象**（如果 Bean 被 AOP 切面代理）。**三级缓存只会对单例 Bean 生效**。

> 创建过程：
>
> 先去 **一级缓存** `singletonObjects` 中获取，存在就返回；
>
> 如果不存在或者对象正在创建中，于是去 **二级缓存** `earlySingletonObjects` 中获取；

> 如果还没有获取到，就去 **三级缓存** `singletonFactories` 中获取，通过执行 `ObjectFacotry` 的 `getObject()` 就可以获取该对象，获取成功之后，从三级缓存移除，并将该对象加入到二级缓存中
>
> 流程**举例**：A包括B，B包括A。创建A，缺B。去创建B，但A也没好。去**三级缓存**中调用`getObject()`方法获取**A的前期暴露对象**（由`getEarlyBeanReference()`生成，然后把**前期暴露对象放入二级缓存**，然后B借用它来创建）

> 在没有 AOP 的情况下，确实可以只使用一级和三级缓存来解决循环依赖问题。当涉及到 AOP 时，二级缓存就显得非常重要了，因为它确保了即使在 Bean 的创建过程中有多次对早期引用的请求，也始终只返回同一个代理对象，从而避免了同一个 Bean 有多个代理对象的问题

@Lazy

如果一个 **Bean 没有被标记为懒加载**，那么它会在 Spring IoC 容器**启动的过程中被创建和初始化**。如果一个 Bean **被标记为懒加载**，那么它不会在 Spring IoC 容器启动时立即实例化，而是在**第一次被请求时才创建**。这可以帮助**减少应用启动时的初始化时间**，也**可以解决循环依赖问题**（但不建议）

事务

1. 编程式事物
2. 声明式事务

Spring 事务中哪几种事务传播行为：

`TransactionDefinition.PROPAGATION_REQUIRED`

1. `REQUIRED`：默认，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
2. `REQUIRES_NEW`：如果当前存在事务，则把当前事务**挂起**。即一定会新开启自己的事务
3. `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**事务来运行
4. `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**
5. `SUPPORTS`、`NOT_SUPPORTED`、`NEVER`：这三种导致事务不会发生回滚
    1. 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
    2. 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。**
    3. NEVER：以非事务方式运行，如果当前存在事务，则抛出异常

隔离级别

`TransactionDefinition.ISOLATION_DEFAULT`

1. `DEFAULT`：默认级别
2. `READ_UNCOMMITTED`：**最低隔离级别**，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**
3. `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
4. `REPEATABLE_READ`：多次读取结果一致，**可以阻止脏读和不可重复读**
5. `SERIALIZABLE`：**最高隔离级别**，都可以阻止

> 脏读：读取了另一个事务未提交的数据
>
> 幻读：第二次读取的结果与第一次读取的结果不一致（**新插入导致**）
>
> 不可重复读：第二次读取的结果与第一次读取的结果不一致（**数据修改导致**）

异常

`@Transactional` 注解默认回滚策略是只有在遇到`RuntimeException`(运行时异常) 或者 `Error` 时才会回滚事务，而不会回滚 `Checked Exception` （受检查异常）

如果想要修改默认的回滚策略，可以使用 `@Transactional` 注解的 `rollbackFor` 和 `noRollbackFor` 属性来指定哪些异常需要回滚。

```
@Transactional(rollbackFor = Exception.class)
public void someMethod() {
}//所有异常都回滚

@Transactional(noRollbackFor = CustomException.class)
public void someMethod() {
}//指定的异常不会回滚
```

Spring Security

控制访问请求权限的方法

`permitAll()`：无条件允许任何形式访问，不管你登录还是没有登录。

`anonymous()`：允许**匿名**访问，也就是没有登录才可以访问。

`denyAll()`：无条件决绝任何形式的访问。

`authenticated()`：只允许**已认证**的用户访问。（仅仅身份验证）

`fullyAuthenticated()`：只允许**已经登录**或者通过 remember-me 登录的用户访问。

- 比`authenticated()`更严格。完全身份验证意味着用户不仅仅通过了身份验证，还必须通过了其他安全检查，例如输入了密码、输入了验证码等

`hasRole(String)`：只允许**指定角色**访问。

`hasAnyRole(String)`：指定一个或者多个角色，满足其一的用户即可访问。

`hasAuthority(String)`：只允许具有**指定权限**的用户访问

`hasAnyAuthority(String)`：指定一个或者多个权限，满足其一的用户即可访问。

`hasIpAddress(String)`：只允许**指定 ip** 的用户访问。

总之：匿名、已认证、已登陆、指定角色、指定权限、指定IP

密码加密

加密算法实现类的接口是 `PasswordEncoder` ，如果你想要自己实现一个加密算法的话，也需要实现 `PasswordEncoder` 接口。

```
public interface PasswordEncoder {
    // 加密也就是对原始密码进行编码
    String encode(CharSequence var1);
    // 比对原始密码和数据库中保存的密码
    boolean matches(CharSequence var1, String var2);
    // 判断加密密码是否需要再次进行加密，默认返回 false
    default boolean upgradeEncoding(String encodedPassword) {
        return false;
    }
}//三个必须实现的接口
//官方推荐使用基于 bcrypt 强哈希函数的加密算法实现类
```

Spring注解

@SpringApplication

`@Configuration` + `@EnableAutoConfiguration` + `@ComponentScan`

- **`@EnableAutoConfiguration`**：启用 SpringBoot 的**自动配置**机制
- **`@ComponentScan`**：扫描**被 `@Component` (`@Repository`,`@Service`,`@Controller`)注解**的 bean，注解默认会扫描该类所在的包下所有的类。
- **`@Configuration`**：允许在 Spring 上下文中注册额外的 bean 或导入其他配置类

SpringBean相关

@Autowired

自动导入对象到类中，被注入进的类同样要被 Spring 容器管理。

- 默认按照`type`注入

@Component

通用注解

`@Repository`、`@Service`、`@Controller` 其实都是`@Component`。

@ResController

`@Controller`+`@ResponseBody`

`@Controller`：返回的是一个页面，基本用在MVC中

`@ResController`：返回的是Json或xml形式的数据

@Scope

```
@Bean
@Scope("singleton")
public Person personSingleton() {
    return new Person();
}
```

singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。

prototype : 每次请求都会创建一个新的 bean 实例。

request : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。

session : 每一个 HTTP Session（用户会话，从登录到退出） 会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。

~~global-session~~：Spring5中已经没有了

@Configuration

一般用来声明配置类，**可以使用 `@Component` 注解替代**，不过使用`@Configuration`注解声明配置类更加语义化

HTTP请求相关类型

- GET、POST、PUT、DELETE、PATCH

1. `@GetMapping("users")`等价于`@RequestMapping(value="/users",method=RequestMethod.GET)`
2. `@PostMapping("users")` 等价于`@RequestMapping(value="/users",method=RequestMethod.POST)`
3. `@PutMapping("/users/{userId}")` 等价于 `@RequestMapping(value="/users/{userId}",method=RequestMethod.PUT)`
4. `@DeleteMapping("/users/{userId}")`
5. `@PatchMapping("/profile")`：PUT 不够用了之后才用 PATCH

前后端传值：参数中

@PathVariable和@RequestParam

```
@GetMapping("/klasses/{klassId}/teachers")
public List<Teacher> getKlassRelatedTeachers(
         @PathVariable("klassId") Long klassId,
         @RequestParam(value = "type", required = false) String type ) {
...
}
```

@RequetBody

用于读取 Request 请求（可能是 POST,PUT,DELETE,GET 请求）的 body 部分并且**Content-Type 为 application/json** 格式的数据，接收到数据之后会自动将数据绑定到 Java 对象上去。系统会使用`HttpMessageConverter`或者自定义的`HttpMessageConverter`将请求的 body 中的 **json 字符串转换为 java 对象**。

读取配置信息

```yaml
# 配置文件
wuhan2020: 2020年初武汉爆发了新型冠状病毒，疫情严重，但是，我相信一切都会过去！武汉加油！中国加油！

my-profile:
  name: Guide哥
  email: koushuangbwcx@163.com

library:
  location: 湖北武汉加油中国加油
  books:
    - name: 天才基本法
      description: 二十二岁的林朝夕在父亲确诊阿尔茨海默病这天，得知自己暗恋多年的校园男神裴之即将出国深造的消息——对方考取的学校，恰是父亲当年为她放弃的那所。
    - name: 时间的秩序
      description: 为什么我们记得过去，而非未来？时间“流逝”意味着什么？是我们存在于时间之内，还是时间存在于我们之中？卡洛·罗韦利用诗意的文字，邀请我们思考这一亘古难题——时间的本质。
    - name: 了不起的我
      description: 如何养成一个新习惯？如何让心智变得更成熟？如何拥有高质量的关系？ 如何走出人生的艰难时刻？
```

@Value

```java
@Value("${wuhan2020}")
String wuhan2020;
```

@ConfigurationProperties

- 像使用普通的 Spring bean 一样，将其注入到类中使用。（配置文件中写一个类的实例并注入）

```java
@Component
@ConfigurationProperties(prefix = "library")
class LibraryProperties {
    @NotEmpty
    private String location;
    private List<Book> books;

    @Setter
    @Getter
    @ToString
    static class Book {
        String name;
        String description;
    }
  省略getter/setter
  ......
}

```

@PropertySource

`@PropertySource`读取指定 properties 文件

```
@Component
@PropertySource("classpath:website.properties")
class WebSite {
    @Value("${url}")
    private String url;

  省略getter/setter
  ......
}
```

参数校验

- `@NotEmpty` 被注释的字符串的不能为 null 也不能为空
- `@NotBlank` 被注释的字符串非 null，并且必须包含一个非空白字符
- `@Null` 被注释的元素必须为 null
- `@NotNull` 被注释的元素必须不为 null
- `@AssertTrue` 被注释的元素必须为 true
- `@AssertFalse` 被注释的元素必须为 false
- `@Pattern(regex=,flag=)`被注释的元素必须符合指定的正则表达式
- `@Email` 被注释的元素必须是 Email 格式。
- `@Min(value)`被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@Max(value)`被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@DecimalMin(value)`被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@DecimalMax(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@Size(max=, min=)`被注释的元素的大小必须在指定的范围内
- `@Digits(integer, fraction)`被注释的元素必须是一个数字，其值必须在可接受的范围内
- `@Past`被注释的元素必须是一个过去的日期
- `@Future` 被注释的元素必须是一个将来的日期
- ……

验证请求体：RequestBody

同时需要在方法的参数中加上 `@RequestBody @Valid`

验证请求参数：Path Variables 和 Request Parameters

- **在类上加上 `@Validated` 注解**

异常处理

Controller层

1. `@ControllerAdvice` :注解定义**全局异常处理类**
2. `@ExceptionHandler` :注解声明**异常处理方法**

```
@ControllerAdvice
@ResponseBody
public class GlobalExceptionHandler {

    /**
     * 请求参数异常处理
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?>
handleMethodArgumentNotValidException(MethodArgumentNotValidException ex,
HttpServletRequest request) {
        ......
    }
}
```

JPA相关：

JAVA持久化API

表创建以及主键

`@Entity`声明一个类对应一个数据库实体。

`@Table` 设置表名

`@Id`：声明一个字段为主键。

> 使用`@Id`声明之后，我们还需要定义主键的生成策略。我们可以**使用 `@GeneratedValue` 指定主键生成策略**。@GeneratedValue`注解默认使用的策略是`GenerationType.AUTO。
>
> **@GeneratedValue(strategy = GenerationType.IDENTITY)等价于**通过 `@GenericGenerator`声明一个主键策略，然后 `@GeneratedValue`使用这个策略

```
@Entity
@Table(name = "role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) //IDENTITY常用于MYSQL
    private Long id;
```

设置字段类型

`@Column` 声明字段。

```java
@Column(columnDefinition = "tinyint(1) default 1")
private Boolean enabled;

@Column(name = "user_name", nullable = false, length=32)
private String userName;
```

指定不持久化

`@Transient`：声明不需要与数据库映射的字段，在保存的时候不需要保存进数据库

其他

`@Lob`:声明某个字段为大字段。

可以使用**枚举类型**的字段，不过枚举字段要用`@Enumerated`注解修饰。

1. `@CreatedDate`: 表示该字段为创建时间字段，在这个实体被 insert 的时候，会设置值
2. `@CreatedBy` :表示该字段为创建人，在这个实体被 insert 的时候，会设置值

   `@LastModifiedDate`、`@LastModifiedBy`同理。
3. `@EnableJpaAuditing`：开启 JPA 审计功能。

`@Modifying` 注解提示 JPA 该操作是修改操作,注意还要配合`@Transactional`注解使用

- `@OneToOne` 声明一对一关系
- `@OneToMany` 声明一对多关系
- `@ManyToOne` 声明多对一关系
- `@ManyToMany` 声明多对多关系

事务

`@Transactional`

Json处理

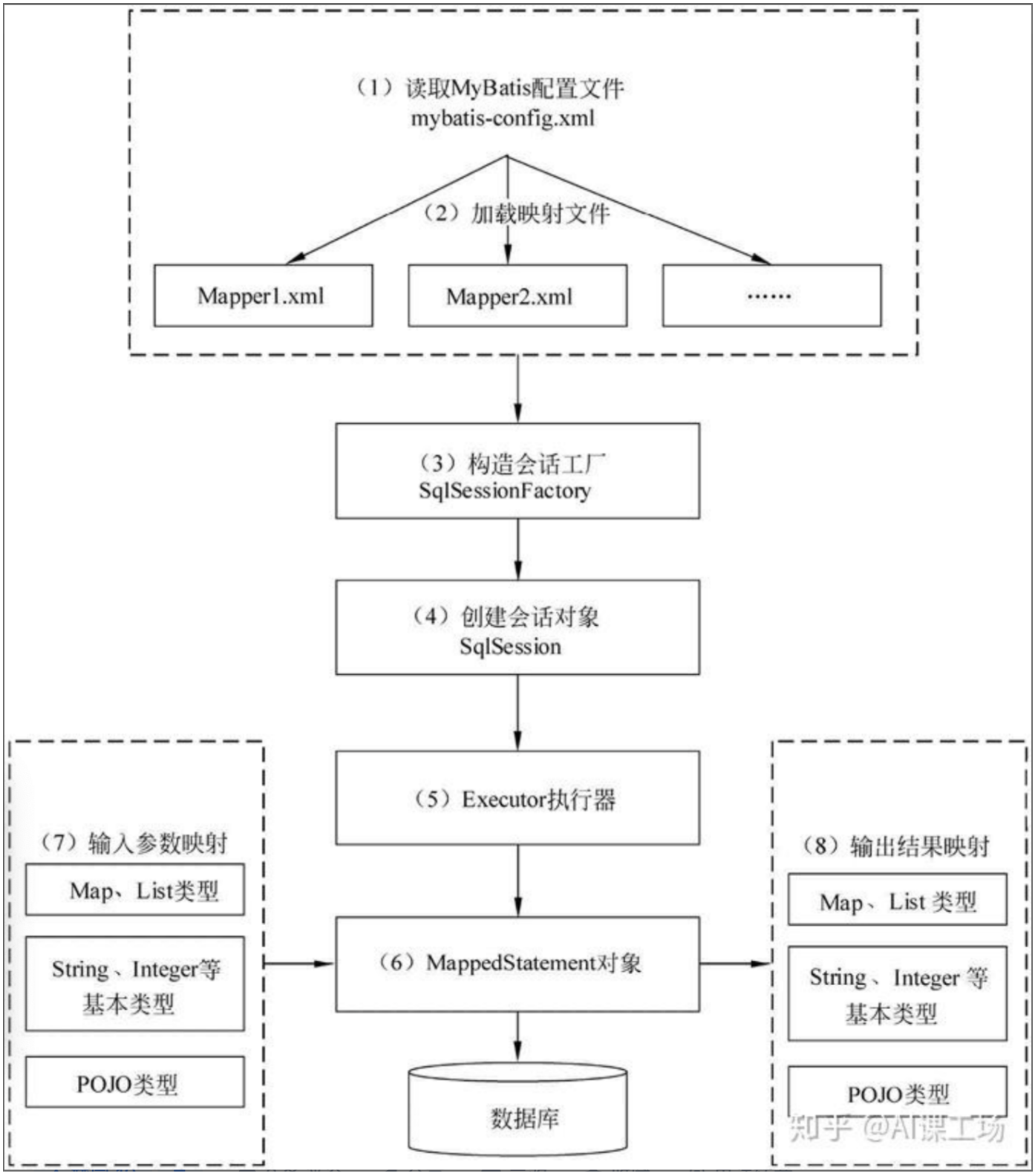
`@JsonIgnoreProperties` 作用在类上用于过滤掉特定字段不返回或者不解析

`@JsonIgnore`一般用于类的属性上，作用和上面的`@JsonIgnoreProperties` 一样。

`@JsonFormat`一般用来格式化 json 数据。

Mybatis

执行流程



#{}和${}的区别

#{}方式能够很大程度防止sql注入(安全)，${}方式无法防止Sql注入

#{} 是占位符，做预编译处理，MyBatis在处理 #{ } 时，会将 SQL 中的 #{ } 编译为 ?，对应的变量自动加单引号

${} 是拼接符，即字符串替换

XML映射文件中有哪些标签

`<select>、<insert>、<update>、<delete>`

> 在 MyBatis 中，每一个 `<select>`、 `<insert>`、 `<update>`、 `<delete>` 标签，都会被解析为一个 `MappedStatement` 对象。

`<resultMap>、 <parameterMap>、 <sql>、 <include>、 <selectKey>`

加上动态 sql 的 9 个标签， **`trim|where|set|foreach|if|choose|when|otherwise|bind`**

> `<sql>` 为 sql 片段标签，通过 `<include>` 标签引入 sql 片段
>
> `<selectKey>` 为不支持自增的主键生成策略标签

Dao接口的工作原理是什么？Dao 接口里的方法，参数不同时，方法能重载

最佳实践中，通常一个 xml 映射文件，都会写一个 Dao 接口与之对应。Dao 接口就是人们常说的 `Mapper` 接口

接口的**全限名**，就是映射文件中的 `namespace` 的值；接口的**方法名**，就是映射文件中 `MappedStatement` 的 id 值；接口方法内的**参数**，就是传递给 sql 的参数。当调用接口方法时，**接口全限名+方法名**拼接字符串作为 key 值，可唯一定位一个 `MappedStatement`

> Dao 接口里的方法可以重载，但是 Mybatis 的 xml 里面的 ID 不允许重复。
>
> **Mybatis 的 Dao 接口可以有多个重载方法，但是多个接口对应的映射必须只有一个，否则启动会报错。**
>
> 重载需要满足其一：
>
> 1. 仅有一个无参方法和一个有参方法
> 2. 多个有参方法时，参数数量必须一致。且使用相同的 `@Param` ，或者使用 `param1` 这种。
>
> ```
> Person queryById();
> Person queryById(@Param("id") Long id);
> Person queryById(@Param("id") Long id, @Param("name") String name);
> ```

```
public interface StuMapper {
 List<Student> getAllStu();
 List<Student> getAllStu(@Param("id") Integer id);
}
```

```
<select id="getAllStu" resultType="com.pojo.Student">
  select * from student
  <where>
    <if test="id != null">
      id = #{id}
    </if>
  </where>
</select>
```