

# CS209 Final Review(10+)

## Lecture 10 Reflection + Annotation

### Reflection

- 反射用于在运行时检查或修改方法、类和接口的行为。
  - 检查类的所有字段和方法
  - 调用对象的方法
  - 访问另一个类中的私有字段。

如果读取了一个文件 这个文件里面的东西和类型是位置的，如果总是用 instanceof 就很繁琐并且违反了OOP的原则

```
1  Object x = ...;
2  if (x instanceof Shape) {
3      Shape s = (Shape) x;
4      g2.draw(s);
5  }
6  Class<String> cls1 = String.class; // for known classes
7  Class<String> cls2 = Class.forName("java.lang.String"); //using full
package name
8  Class cls3 = x.getClass(); // 运行时的类型 如果Animal animal = new dog();
GetClass是dog
9  // 有了类就可以有(e.g., name, fields, constructors, methods)
```

### GetName的特例

- byte: [B char: [C double: [D float: [F int: [I long: [J
- class or interface: [Lname; short: [S boolean: [Z
- 比如 String[][] 就是 [[Ljava.lang.String; int[][] 就是 [[I

### JVM 中的 Class 实例:

- JVM 为每个数据类型（包括基本类型、类、接口等）创建一个唯一的 Class 类实例。
- "java.lang.String" "java.util.Random" "java.lang.Runnable"
- 通过 Class 实例可以获取类的详细信息：
  - name : "java.lang.String" package : "java.lang" Interface
  - super : "java.lang.Object" Field Method

### JVM自动创建并且无法直接实例化 Class 对象：

- 当JVM加载类（.class文件）时，会自动创建一个 Class 类的实例，只会创建一个 Only One
- Class 类没有 public constructor。只有JVM可以创建 Class 对象，不能通过 new 手动实例化

```
1 Class cls = new Class(String); // 不允许
```

## AccessibleObject是Field Method Constructor的共同父类

### a. Field

```

1 Field f = String.class.getDeclaredField("value");
2 // 如果去掉Declared使用 getField() 只会返回public的字段 但是会返回继承的
  public字段
3 // Declared会获取除了继承的字段之外的所有字段 包括private protected default
4 System.out.println(f.getName());           // 输出字段名称: value
5 System.out.println(f.getType());           // 输出字段类型: [B (byte
  数组)
6
7 int m = f.getModifiers();                  // 获取字段的修饰符
8 System.out.println(Modifier.isFinal(m));   // 判断字段是否是 final:
  true
9 System.out.println(Modifier.isPrivate(m)); // 判断字段是否是
  private: true
10
11 Field field = Animal.getDeclaredField("Name");
12 field.setAccessible(true)
13 field.set(instance, "test"); // 如果是static private就用set(null, "test")

```

### b. Method

```

1 String s = "Hello Java";
2 Method m = String.class.getMethod("substring", int.class); // 还需要提供参
  数列表
3 m.invoke(s, 6); // Java
4 Method m2 = Integer.class.getMethod("parseInt", String.class);
5 m2.invoke(null, "12345").getClass().getName(); // java.lang.Integer
6 // 如果是static Method 前面就是null

```

也可以通过 method.setAccessible(true); 来 invoke 非 public 方法

### c. Constructor

```
1 Class cls = Student.class; // 获取 Student 类的 Class 对象
2 Constructor constructor = cls.getConstructor(String.class, int.class);
3 // 获取指定构造函数
4 Student std = (Student) constructor.newInstance("Alice", 15);
5 // 调用构造函数，创建对象
```

## Utility

- a. 在 JUnit 测试中的应用：自动发现和运行测试方法 操作私有字段和私有方法
- b. 在 Spring 框架中的应用

**依赖注入：**例如，根据类的构造函数参数类型来实例化依赖对象。

**动态识别依赖：**通过反射获取构造函数的参数类型或注解，确定需要注入的对象类型。

**运行时创建对象：**使用反射动态创建对象，并将这些对象注入到其他类的属性中。

## Problems

- a. **安全风险 (Risky)** 反射允许直接访问私有字段和方法，包括 final 字段。这可能破坏类的封装性
- b. **缺乏编译期类型安全：**验证方法是否存在、参数类型是否匹配等步骤延后到了运行时
- c. **重构时容易出错 (Causes Bugs When Refactoring) :** 如方法重命名或字段更改 High Coupling
- d. **性能较低 (Performance is Slower)**

## Annotation

- o **Java注解以 @ 开头：**
- o **Java注解是附加到程序实体上的元数据**，附加到类、接口、字段和方法上，用于提供额外的信息。
- o **Java注解不会改变程序的语义**
- o **注解与注释不同：**注解不能简单地看作注释 (comments)，因为它们可能会影响编译器或运行时对程序的处理方式。例如，`@Override` 注解会告诉编译器这是一个重写方法，如果方法签名不正确，编译器会报错。

## Purpose

- a. **编译器指令 (Compiler instructions)** `@Override` `@Deprecated`  
`@SuppressWarnings`
- b. **构建时指令 (Build-time instructions)** `@Getter` `@Setter` `@Table` `@Generated`

c. 运行时指令 (Runtime instructions) @Test @Test @Controller @Service  
@Autowired

a. 预定义注解 (Predefined annotations)

i. 内置注解 (Built-in annotations) : 这是Java语言本身提供的注解类型，用于编译器提示或运行时功能

- @Override @Deprecated : 标记某个元素已过时。
- @SuppressWarnings({"unchecked", "deprecation"})
  - deprecation (过时) : 表示某些代码或方法已经被标记为过时。
  - unchecked (未检查) : 通常在与泛型出现之前的旧代码交互时产生，例如使用原始类型 (raw type) 时。
- @SafeVarargs 注解用于告诉编译器：该方法的可变参数操作是安全的，抑制相关警告。

ii. 元注解 (Meta-annotations) :

元注解是用于描述其他注解的注解，例如控制自定义注解的行为。常见元注解包括：

- @Retention : 指定注解的生命周期 (如源代码、字节码、运行时)。
  - a. RetentionPolicy.SOURCE
    - 定义：注解只保留在 源码 中，编译后被丢弃（不会保留在 .class 文件中）。
    - 特点：仅对源码有效，主要用于编译器的检查或提示。无法通过反射在运行时获取。
    - 典型用途：用于 编译器级别的注解，如 @Override 和 @SuppressWarnings 。
  - b. RetentionPolicy.CLASS
    - 定义：注解会被保留到 字节码文件 (.class) 中，但运行时不可见（JVM 会忽略）。
    - 特点：编译器记录注解，但运行时无法访问。是默认的保留策略（如果没有 @Retention ）。
    - 典型用途：用于编译器或工具在构建阶段对字节码的分析。
  - c. RetentionPolicy.RUNTIME
    - 定义：注解会被保留到运行时，且可以通过反射获取。
    - 特点：编译器会记录注解，JVM 在运行时也会保留它们。可以结合反射实现动态行为。
    - 典型用途：用于 运行时注解，如 Spring 的 @Autowired 、JPA 的 @Entity 。

- 可以用于类型自动检查 **注解本身不会自动执行检查逻辑**, 需要结合反射手动实现逻辑

```

1  @Range(min = 1, max = 100)
2  public int age;

```

- **@Target** : 指定注解可以应用于哪些元素 (如类、方法、字段)。
  - a. **@Target(ElementType.METHOD)** : 可以用在方法上。
  - b. **@Target({ElementType.TYPE, ElementType.METHOD})** : 可以用在类、接口或枚举上。
- **@Documented** : 标记注解是否会包含在Javadoc中。
  - **Javadoc** 是 Java 提供的一种工具, 用于根据代码中的注释生成 API 文档。

### b. 自定义注解 (Custom annotations)

- 开发者可以定义自己的注解以满足特定需求。
  - 注解方法不能有参数 不能抛出异常。
  - 注解方法返回的值只能是合法的数据类型。 (自己定义的 **Class** 不可以)
    - 基本类型 (如 **int**、**boolean**) **String** 枚举类型 **Class<?>**
    - **Class** 类型。 其他注解。 上述类型的数组。

```

1  [Access Specifier] @interface <AnnotationName> {
2      DataType <MethodName>() [default value];
3  }

```

```

1  @Retention(RetentionPolicy.RUNTIME) // 注解的生命周期 (运行时可用)
2  @Target({ElementType.METHOD, ElementType.TYPE}) // 限定作用范围
3  public @interface MyAnnotation {
4      String value(); // 必须元素 (没有默认值)
5      int version() default 1; // 可选元素 (有默认值)
6      // 一个注释可以没有必须元素 也可以没有可选元素 按需求而定
7  }
8  @MyAnnotation(value = "Custom Annotation Example", version = 2)
9  public class MyClass {
10     @MyAnnotation("Method Annotation") // 只有当必须元素数量等于1的时候, 可以
11     // 不带名字
12     public void myMethod() {
13         System.out.println("Using custom annotation");
14     }
15 }

```

```
13      }
14  }
```

## Lecture 11 Java EE + Servlet + JDBC + JPA

### Java EE

**Java EE** (顶层) : 基于 Java SE (Standard Edition, 标准版) 构建, 提供额外的企业级功能。

- 数据库访问: JDBC、JPA
- Web 服务: Servlets、XML 处理
- 远程方法调用: RMI
- 消息服务: JMS
- 企业级功能: Enterprise Beans (EJB)

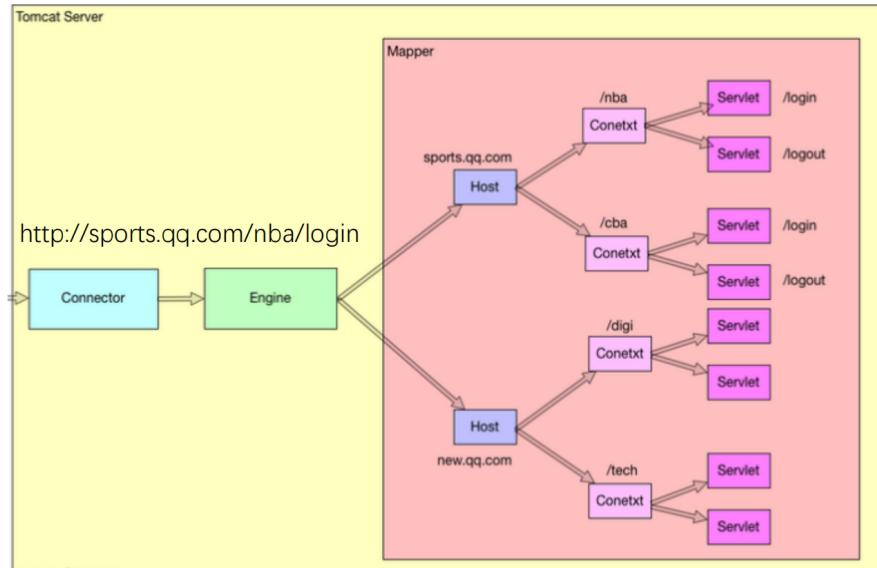
**Java SE** (中层) : 包含日常使用的核心 API, 例如: `java.lang` `java.io` Java SE 是 Java 平台的基础, 支持通用的 Java 应用开发。

**Java ME** (底层) : 针对小型设备 (如手机、嵌入式设备) 的 Java 版本。

### Java EE (Enterprise Edition)

- 是一组针对企业级应用开发的标准和规范, 包括 API 定义 (如 Servlet) 和行为描述 (如生命周期管理)
- **Oracle Glassfish** (*Reference Implementations*): 完整支持 Java EE 的 Servlet 和 JSP 规范。
- **Apache Tomcat**:
  - 专注于 Servlet 和 JSP 的轻量级容器, 广泛应用于小型和中型 Web 应用。
  - 浏览器 (Browser) 向 HTTP 服务器发送 HTTP 请求。如果请求是静态内容, **Apache HTTP Server** 会直接响应。如果请求需要逻辑处理, 则转发到 **Tomcat**, 运行 Servlet 或 JSP。Tomcat 与数据库交互, 返回结果, 最终通过 HTTP Server 响应用户。
  - `Connector`:
    - **Connector** 封装了不同的网络协议 (如 HTTP、HTTPS)。**Connector** 将接收到的网络请求转化为 Java 容器可识别的 `Request` 对象, 并传递给 **Container**。容器处理完成后, **Connector** 负责将处理结果封装为网络协议格式, 并返回给客户端。
    - **Connector** 还负责处理底层 Socket 连接
  - `Container`
    - **Engine**: 每个 Tomcat `Service` 只有一个 `Engine`。表示一个完整的请求处理流程。接收来自 `Connector` 的请求。将请求分发到正确的虚拟主机 (`Host`)。
    - **Host**: 根据请求中的域名, 将请求路由到对应的 `Context` (上下文)。

- **Context**: 对应一个主机上不同的Application，将请求进一步路由到具体的 **Wrapper** (封装器)。
- **Wrapper (封装器)** 对应一个Application上不同的Servlet
  - **Wrapper** 是对单个 **Servlet** 的封装。将请求交给特定的 **Servlet** 实现类，处理具体的 HTTP 请求。



## Multitiered Applications

### a. Client Tier

特性	应用程序客户端 (Application Client)	Web 客户端 (Web Client)
运行环境	客户端机器上的独立程序	浏览器
界面类型	图形用户界面 (GUI)	网页 (HTML, JSP)
复杂性	可直接访问业务逻辑，较复杂	轻量级用户交互，较简单
典型使用场景	企业级桌面应用 (如银行内部系统)	在线服务或电商平台

### b. Web Tier

- 处理Client Tier和Business Tier之间的交互
  - 客户端（如浏览器）发送请求到 Web 层。Web 层中的 Servlet 处理请求逻辑，调用业务层组件（如 JavaBeans 或 Enterprise Beans）处理逻辑。
  - Servlet 将处理结果（如 HTML、JSON）传递给 JSP 页面，用于生成动态页面。
  - JSP (JavaServer Pages) 返回生成的 HTML 页面给客户端浏览器。

### c. Business Tier

- 处理业务逻辑，将核心功能封装在组件中，支持数据访问、事务管理和安全性。

- **Enterprise JavaBean (EJB, 企业 Java Bean)** 是一种服务器端的软件组件，用于封装业务逻辑提供 事务管理 远程调用 消息驱动 内置的安全机制 生命周期管理 以及 负载均衡等 功能。

#### d. Data Tier

- Aka. **企业信息系统层 (Enterprise Information Systems, EIS)** 提供持久化数据存储，为业务层提供与数据相关的服务。业务层通过特定接口访问以获取或存储数据。

JPA JTA

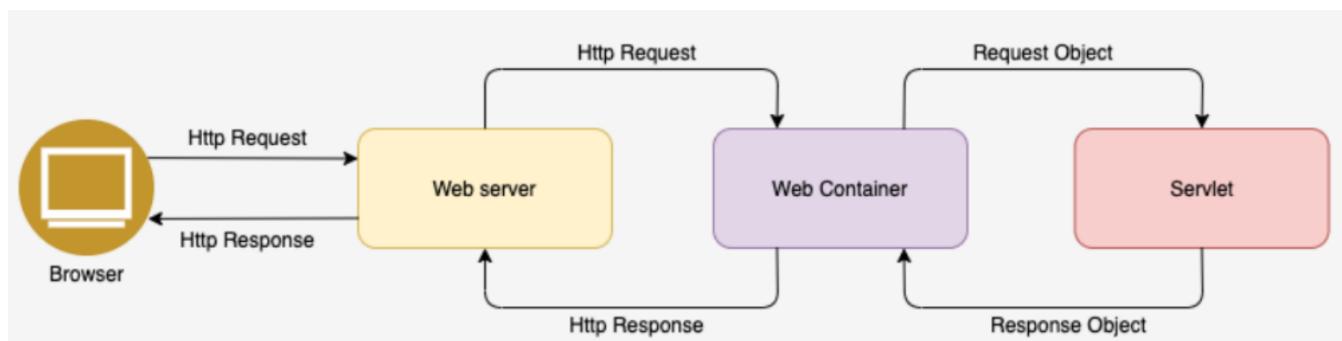
## Container

- 不仅负责承载组件，还负责为组件提供运行时环境和一系列服务（如安全、事务管理、生命周期管理等）。

容器类型	主要用途	运行位置	管理的组件
EJB 容器	处理复杂的业务逻辑，支持事务、远程调用和分布式系统。	Java EE 服务器	Session Beans, Message-Driven Beans
Web 容器	处理 HTTP 请求，生成动态网页内容，提供前端与后端交互支持。	Java EE 服务器	Servlet, JSP 页面
Application Client 容器	运行桌面客户端程序，允许客户端直接与服务器交互，提供 GUI。	客户端设备	桌面客户端应用程序
Applet 容器	在浏览器中运行 Java 小程序（已逐步弃用），提供交互式网页内容。	客户端设备（浏览器）	Applet

## Servlet

- **Servlet** 是 Java EE Web 应用程序的核心组件，主要用于处理客户端请求和生成动态内容。通过 **URL** 路径，**Web Client** 将任务交给对应的 **servlets**
- **Servlet** 无法直接处理原始 HTTP 请求，而是依赖 **Web Container** 提供的运行环境和服务。



- **加载和初始化:** 容器加载 **Servlet** 类，调用 **init()** 方法完成初始化。
- **请求处理:**

- 对于每个客户端请求，容器调用 `service()` 方法处理请求。容器会为每个请求分配一个线程。
- `service()` 方法根据请求类型调用相应的 `doGet()` 或 `doPost()`。
- 所有请求共享一个 `Servlet` 实例确保 `Servlet` 是线程安全的。`Servlet` 可以同时处理多个请求
- 销毁：**当容器决定卸载 `Servlet` 时，调用 `destroy()` 方法进行资源清理。

## Hierarchy

```
Servlet -----> GenericServlet -> HttpServlet -> MyServlet
```

- `GenericServlet` 是一个通用的、与协议无关的 `Servlet`。
  - `HttpServlet` 是 `GenericServlet` 的子类，添加了与 HTTP 协议相关的方法和字段
- 核心方法 `service()`：**

- `HttpServlet` (抽象类) 重写了 `service()` 方法。
- 该方法会根据客户端请求的 HTTP 方法（如 GET、POST、PUT、DELETE 等），自动分发到对应的处理方法（如 `doGet()` 等）。但是对于具体的 `doGet()` 方法，用户需要自己定义，实现业务逻辑。

```

1  @Override
2  public void doGet(HttpServletRequest request, HttpServletResponse
response) {
3      response.setContentType("text/html");
4      PrintWriter out = response.getWriter();
5      out.println("<html><body>");
6      out.println("<h1>" + message + "</h1>");
7      out.println("</body></html>");
8  }
```

- 通过 `@WebServlet`，声明 `Servlet` 类并指定 URL 映射路径，简化了传统的 `web.xml` 配置。
- `@WebServlet(name = "hello", value = "/hello-servlet")` 用来修饰 `Servlet` 类

## Web Server & Web Container

- Web Client** 是顾客，发出“点菜”（请求）。发起HTTP请求，与Web Tier交互
- Web Server** 是服务员，接收点菜请求并分配工作：直接端菜（静态内容），或将菜送到后厨加工。属于Web Tier，处理静态资源请求，分发动态请求给Web Container

- **Web Container** 是后厨，负责根据请求“做菜”（处理业务逻辑并生成动态内容）。属于Web Tier，运行服务端组件(Servlet, JSP)并可能调用Business Tier中的复杂逻辑

## Reference Implementation(RI)

- Java EE full-fledged: **Oracle Glassfish** **JBoss AS** **IBM WebSphere**
- Servlet & JSP: **Oracle Glassfish** **Apache Tomcat** **Eclipse Jetty** **Resin**
- EJB: **Oracle Glassfish** **Apache TomEE and OpenEJB** **BuzzyBeans**
- JPA: **EclipseLink** **OpenJPA** **Hibernate**

## JDBC & JPA

### File Systems vs Database

- **使用文件系统的场景**: 需要直接访问文件，例如图片、音频、视频或日志文件。需要和版本控制工具集成。数据无固定结构，且对查询性能要求不高。
- **使用数据库的场景**: 需要存储和管理关联的结构化数据（如用户信息、订单数据）。需要高效查询、统计分析和复杂的关系操作。需要支持高并发、多用户访问，以及事务处理。

### JDBC (Java Database Connectivity)

- 直接使用原生 **SQL** 并且还需要 **map between Java object's data and relational DB**，很麻烦

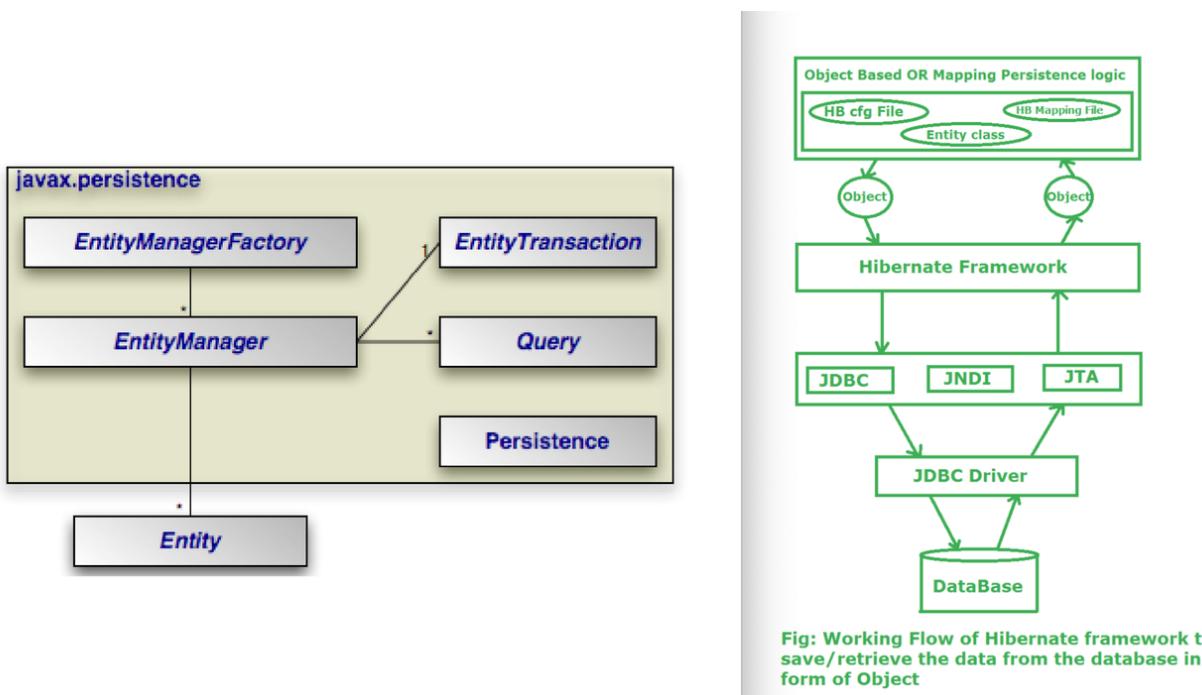
### JPA(Java Persistence API一种接口规范) 使用**Object-Relational Mapping (ORM)**

- **Implementation**: **Eclipselink (RI)** **Hibernate** **Apache OpenJPA** 都是JPA的子类所以可以轻松切换（使用多态）
  - **Hibernate** 可以跨不同的数据库软件使用，而无需修改代码，仅需要调整配置。并且持久化操作基于对象而不是原始类型（primitive types）。

### Primary Components of JPA

- **EntityManagerFactory**: 是一个工厂类，用于创建 **EntityManager**。线程安全，在应用程序启动时初始化，只需要创建一次。管理实体的生命周期及其持久化上下文。
- **EntityManager**: 是一个轻量级的接口，负责与数据库交互。主要用于管理实体对象的增删改查操作。在具体操作中，它提供了对实体的持久化（**persist()**）、移除（**remove()**）、查询（**find()**）等方法。
- **EntityTransaction**: 管理事务的生命周期（开启、提交、回滚）。
- **Query**: 用于执行数据库查询（包括 JPQL 或原生 SQL）。可以从数据库中检索实体对象或原始数据。
- **Persistence**: 提供静态方法，用于创建和管理 **EntityManagerFactory**。

- **Entity**: 数据库中表的抽象，每一个实体对象映射为数据库中的一行记录。通过注解（如 `@Entity`）定义，表示类是可持久化的。



## Lecture 12 The Spring Framework + SpringBoot

### The Spring Framework

#### Core Idea

- 控制反转 (IoC, Inversion of Control) 将对象或程序的一部分控制权交给一个容器或框架。
  - 使用方式: 开发者需要将自己的行为插入框架中的某些位置，可以通过继承框架的类或将自己的类作为插件。然后，框架的代码会在合适的时机调用我们自定义的代码。
- 依赖注入 (DI, Dependency Injection) 是实现控制反转的一种方式。
  - 在 Spring 中的实现: Spring 框架通过依赖注入机制动态地将所需的依赖对象注入到类中，而无需手动实例化这些对象。
  - 原来的方法无法轻松替换 Wheel 或 Battery 以进行单元测试，因为它们的实例化在 Car 内部固定死
    - 低耦合: Car 类不需要知道 Wheel 和 Battery 的具体实现，只需要依赖接口或抽象类。
    - 易于测试: 可以在测试时轻松替换 Wheel 或 Battery 为模拟对象 (Mock)。

```

1  class Car {
2      private Wheel wheel = new NepaliRubberWheel();
3      private Battery battery = new ExcideBattery();
4  }

```

```

5  class Car {
6      private Wheel wheel; private Battery battery;
7      public Car(Wheel wheel, Battery battery) {
8          this.wheel = wheel; this.battery = battery;
9      }
10     public void setWheel(Wheel wheel) {this.wheel = wheel;}
11     public void setBattery(Battery battery) {this.battery = battery;}
12 }

```

## Annotations

- @Component** **类级别的注释** IoC 容器会扫描带有 `@Component` 注解的类，并将其实例化为 Bean。  
**实现：**扫描类：Spring 会扫描项目中的类（通常通过包路径配置），**实例化对象：**自动为带 `@Component` 的类创建实例。**注入依赖：**将依赖注入到需要的地方。
- @Autowired** 通过 IoC 容器在运行时自动将需要的 Bean 注入到指定的属性、构造函数或方法中。  
**实现：**分为3种类型 `Setter` 方法注入 构造函数注入 字段注入 (不符合OOP原则 破坏 Private 封装)

```

1 @Component
2 public class Car {
3     @Autowired private Engine engine;
4     @Autowired public Car(Engine engine) {this.engine = engine;}
5     @Autowired public void setEngine(Engine engine) {this.engine =
6         engine;}

```

- @Configuration** 用于标记一个类为 **配置类**。配置类的作用是**定义并实例化 Bean**，取代传统的 XML 配置文件。  
**特性：**配置类可以包含多个 `@Bean` 方法，每个方法定义一个 Bean。Spring 启动时会扫描 `@Configuration` 类。Spring 调用 `engine()` 方法，将其返回的对象注册为 Bean，供 IoC 使用。当其他组件需要一个 `Engine` 类型的 Bean 时，Spring 会返回这个对象。
- @ComponentScan** 配合 `@Configuration` 使用，用于指定 Spring 容器需要扫描的包路径。
  - 容器会扫描这些路径下的类，查找 `@Component`、`@Service`、`@Repository` 等注解，并自动将它们注册为 Bean

```
1  @Configuration  
2  @ComponentScan(basePackages = "com.example")  
3  public class AppConfig {  
4      @Bean public Engine engine() {return new Engine("V8");}  
5  }
```

v. **@Bean** : `@Bean` 是一个**方法级别的**注解，用于显式声明一个 Bean，并将其交由 Spring IoC 容器管理。

- **特性**: 方法返回的对象被 Spring 管理，可以被注入到其他组件中。
- `@Component` : 是类级别的注解，`@Bean` : 是方法级别的注解，尤其适用于复杂初始化逻辑或需要注入特定参数时。

## Spring IoC 容器 (Spring IoC Container)

**定义**: Spring IoC 容器负责**Instantiate**、**Configure**、**Assemble**和**Manage**对象（即 Bean）。

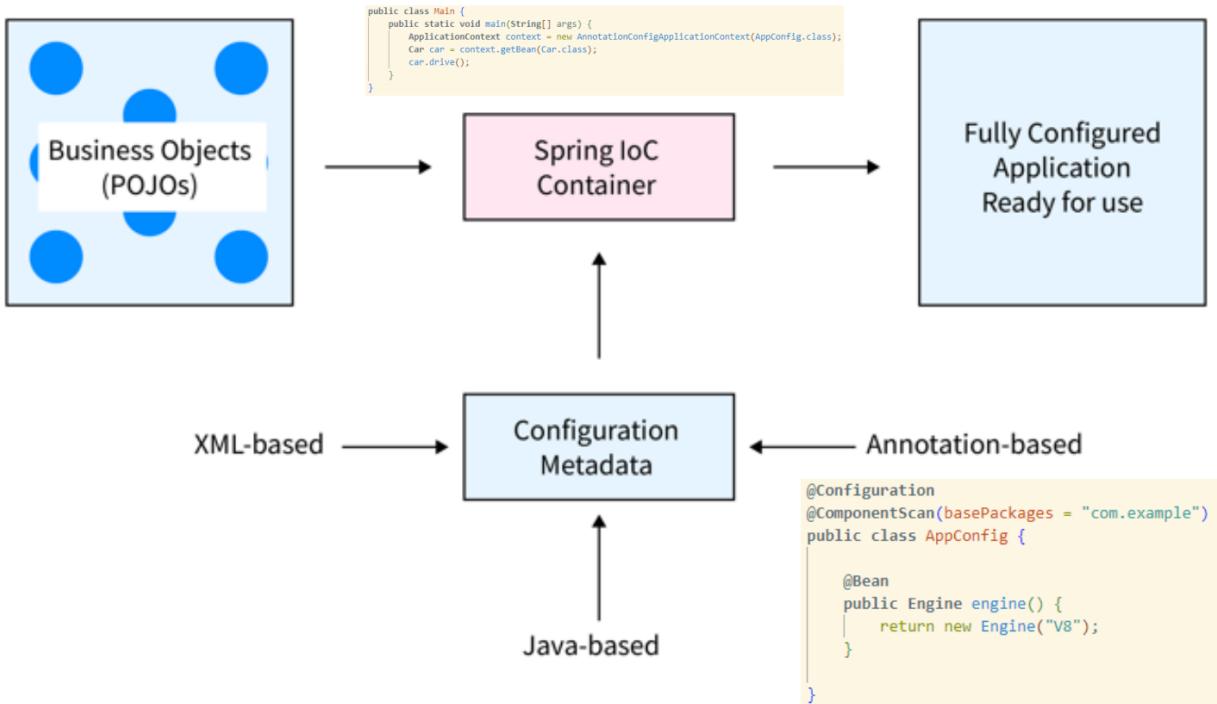
**主要接口**: `ApplicationContext` 提供了丰富的功能，如 Bean 的加载、依赖注入、事件发布等。

```
1  ApplicationContext context = new  
2  AnnotationConfigApplicationContext(AppConfig.class);  
3  Car car = context.getBean(Car.class);  
4  car.drive();
```

- 使用 `AnnotationConfigApplicationContext` 初始化 IoC 容器，并加载 `AppConfig` 配置类。
- 通过 `context.getBean(Car.class)` 从容器中获取 `Car` 实例。
- Spring 自动注入 `Car` 类所需的依赖（如 `Engine`）。通过获取的 `Car` 实例调用其方法（如 `drive()`）

### 工作流程：

- **加载配置元数据**: 容器根据注解（如 `@Configuration`）或 XML 配置文件加载应用的配置元数据。
  - **Configuration Metadata**: `Annotation-based` `Java-based` `XML-based`
- **实例化 Bean**: 容器创建所有的 Bean 实例，并根据依赖关系注入它们所需的依赖。
- **管理生命周期**: 容器负责 Bean 的初始化和销毁过程，开发者无需手动管理。



## Spring AOP (Aspect-Oriented Programming, 面向切面编程)

- 横切关注点 (Cross-Cutting Concerns) 被抽离到切面 (Aspect) 中，与业务逻辑解耦。
- 通过切面 (Aspect) 增强现有代码的功能，而**无需修改原始代码**。
- 横切关注点是指那些**分散在多个模块中的逻辑**，例如日志记录、安全检查、事务管理等。

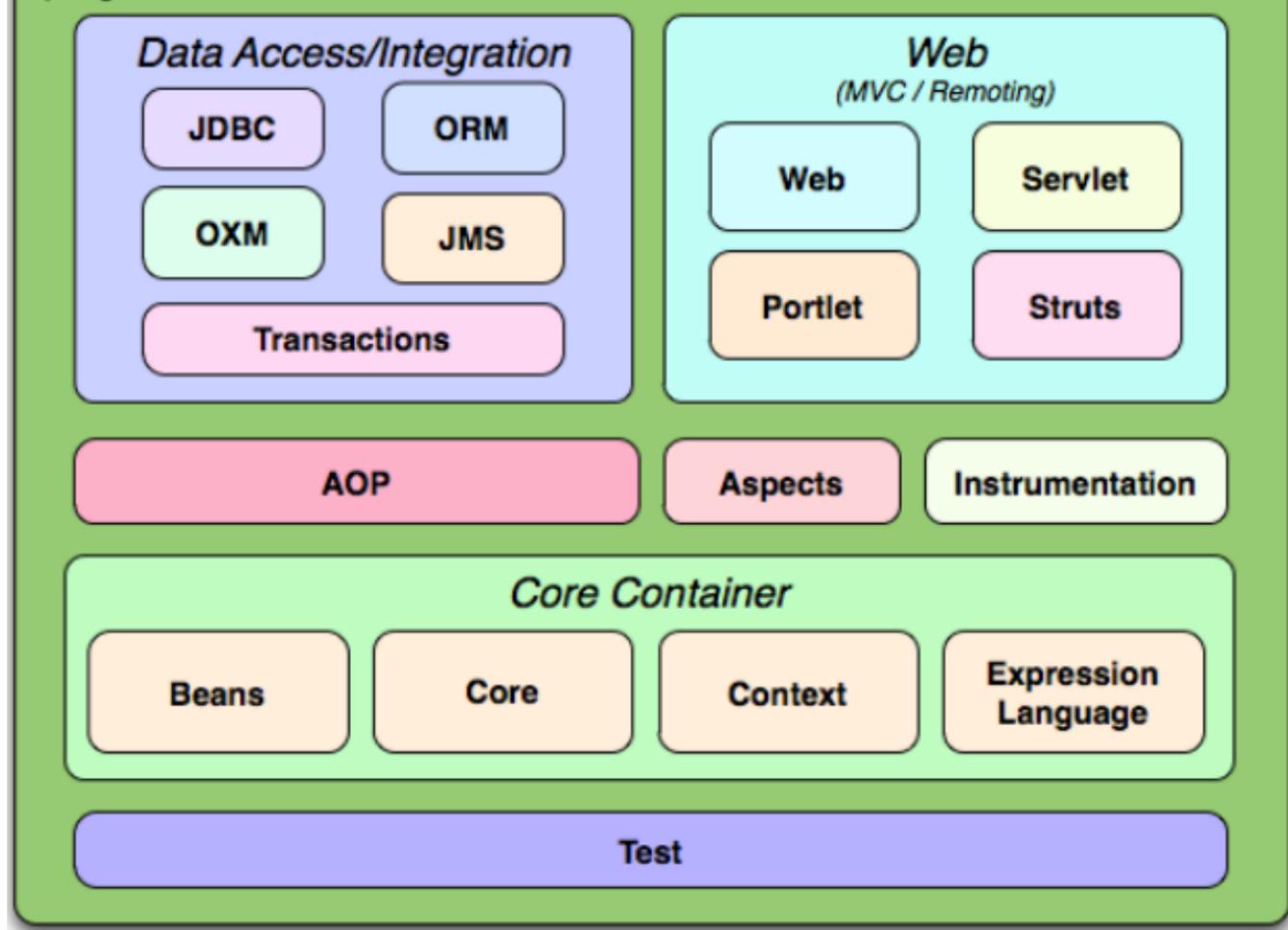
## Spring MVC (Spring Model-View-Controller)

- **Model (模型)**：负责数据、逻辑和规则的管理。
- **View (视图)**：负责数据的可视化显示。
- **Controller (控制器)**：负责接收用户输入并将其转化为对模型或视图的指令。
- **REST 支持**: 提供对 RESTful Web 服务的强大支持，便于构建 API。

### 过程

1. 用户发送 HTTP 请求首先被 **DispatcherServlet** (Spring 的前端控制器，中心控制器) 拦截。
2. **DispatcherServlet** (中心控制器) 将请求分发给适当的控制器 (具体的 **Controller**)。
3. **Controller** 调用服务层处理业务需求，并返回包含数据的模型 (**Model**) 以及逻辑视图给 **DispatcherServlet**。
4. 前端控制器 (**DispatcherServlet**) 将视图渲染任务分配给视图解析器 (**View**) 并返回响应

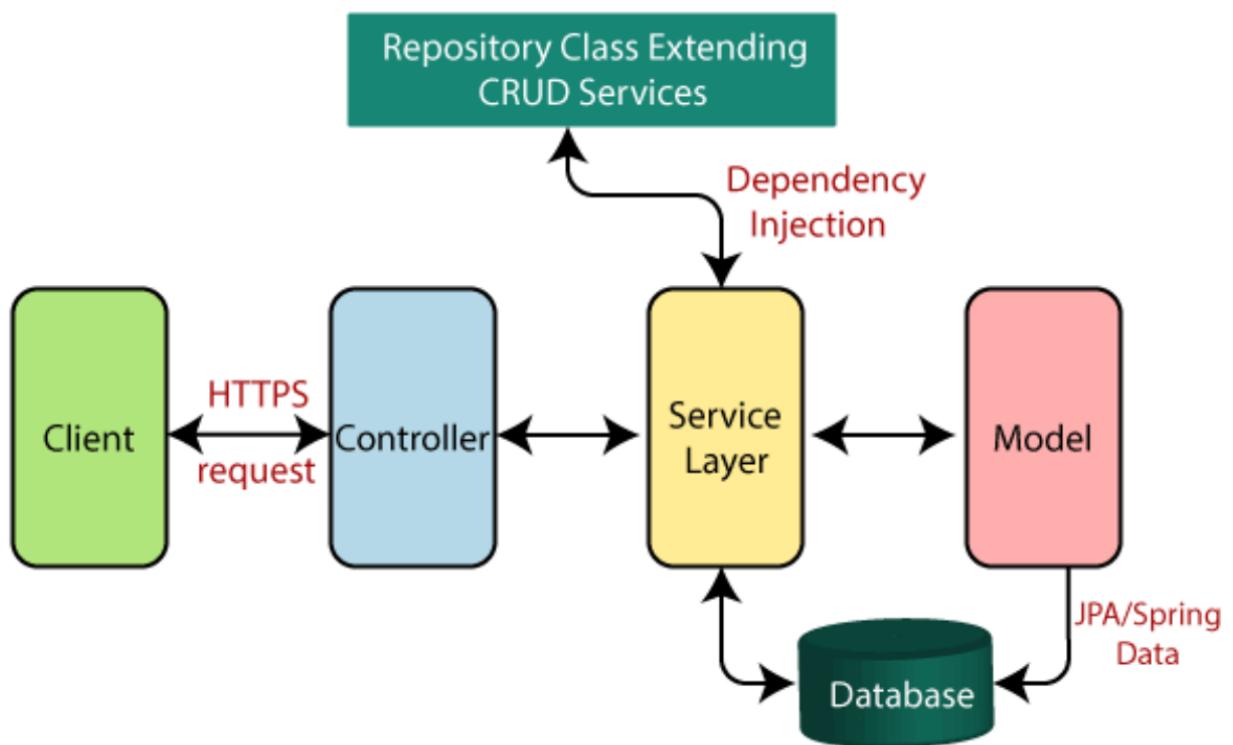
## Spring Framework Runtime



## Spring Boot

- Spring Boot 简化并自动化了配置过程（自动生成默认的 **POM (Project Object Model)**）大大加速了 Spring 应用的创建和部署。"**Convention over Configuration**"（约定优于配置）原则
- Maven 工作流程：
  - 读取 `pom.xml` 文件 构建项目 检查本地仓库查看是否已经存在项目所需的依赖项 `artifact`，如果没有，从中央仓库获取依赖，将依赖存入本地仓库，生成目标文件（例如 `.jar` 或 `.war`），生成的文件通常会存储在 `target` 文件夹下。
- 通过 **Spring Initializer** 启动，  
**Opinionated**（有主观的默认配置）：提供合理的默认配置。例如 Spring Web 默认使用嵌入式 **Tomcat**。  
**Customized**（可定制的）：允许开发者选择和添加自己的依赖（**Postgre** **Thymeleaf**）
- 配置工具
  - **spring-boot-starter-parent** 是 Spring Boot 提供的一种父 POM，用于简化依赖管理和项目构建配置(提供默认的 `plugin` `dependencies` `version` `management properties`)

- `spring-boot-starter-web` 传递式地引入所有与web开发相关的依赖
- `spring-boot-starter-parent` 和 `spring-boot-starter-web` 都属于artifact
- `application.properties` 文件是 Spring Boot 应用配置的主要方式。 (配置内容包括 `Core Properties` `Data Properties` `Transaction & Data Migration`)
- `@SpringBootApplication` 一个注释代表下面3个注释
  - `@Configuration` 指定该类是一个配置类，可注册额外的 Bean 或导入其他配置。
  - `@ComponentScan` 自动扫描该类所在包及其子包中的组件 (如 `@Controller`, `@Service` )。
  - `@EnableAutoConfiguration` 启用 Spring Boot 的自动配置机制，根据类路径中的依赖自动加载相关配置。



- Model 通常指一个Java中的ORM对象，一个 `@Entity` 的类，而Database就是 Postgre 中的具体数据库

**POJO(Plain Ordinary Java Object)的特点：**没有继承特定类或实现特定接口 (如 `extends` 或 `implements` )。没有使用特定注解。只包含私有字段和公共方法，一个 `public` 的无参数构造器，无特定的行为或逻辑 可选的 `toString`、`equals` 和 `hashCode` 方法

## View Controller Service Repository

- **View:** 接受从 `Controller` 传递过来的Model
- **Controller**

- `@Controller`：标记一个类为控制器类，专用于处理 Web 请求，与 `@RequestMapping` 一起使用
- `@RequestMapping` 定义处理的 URL 路径（如 `/list`）
- `Model`：用于传递数据到视图模板，例如将 `students` 列表传递到 `index.html`。  
`Controller` 通过设置一个 `Model` 来传递信息给 `View`
- **Service**
  - **业务逻辑处理：** `@Service` 注解代表提供具体的业务功能，例如查询数据、数据转换和验证。
  - **与 Repository 层交互：** 调用仓储层完成数据库操作（`CRUD`）。

## Repository

**数据访问：**负责与数据库交互。提供数据的增删改查（CRUD）功能。已经支持了一些默认的函数可以直接使用，如果需要自定义，方法前缀名如果为（`findBy` `queryBy` `countBy` `deleteBy`）只需要定义了就会自动生成实现。

**抽象化数据操作：**使用 Spring Data JPA，开发者可以通过接口直接操作数据库，而无需手动编写 SQL 语句。

**支持分页和排序：**通过内置方法实现对数据的分页（paginate）和排序操作。

```

1 List<User> usersByName = userRepository.findByName("Alice");
2 List<User> usersByEmail =
3     userRepository.queryByEmail("bob@example.com");
4 long activeUsersCount = userRepository.countByStatus("ACTIVE");
5 userRepository.deleteByName("Bob");
6 boolean exists = userRepository.existsByEmail("alice@example.com");

```

`CommandLineRunner` 是一个函数式接口通，只有一个抽象方法 `run(String... args)`，过 `@Bean` 定义，用于在应用启动时执行一次性逻辑，例如数据库初始化

```

1 @Bean
2 public CommandLineRunner commandLineRunner(StudentService service) {
3     return args -> {service.addStudents();};
4 }

```

## RESTful Web Service

- **MVC controller:** 使用视图技术（如 Thymeleaf）返回 HTML 页面。返回 HTML 格式的完整页面内容。

- REST controller 以对象形式返回数据，Spring Boot 会自动将其序列化为 JSON 格式。
- @RestController 是 @Controller 和 @ResponseBody 的组合注解。标记类为 RESTful 控制器。自动将返回值序列化为 JSON 或 XML 格式。
- @RequestMapping 定义该控制器 URL 前缀 @RequestMapping("/api/students")

```

1  @GetMapping
2  public List<Student> getStudentsByEmail(@RequestParam(value = "email")
3      Optional<String> email) {
4      if (email.isPresent()) {
5          return studentService.findByEmailLike(email.get());
6      }
7      return studentService.getStudents();
8  }
9  @PutMapping(path = "/{studentId}")
10 public void updateStudent(@PathVariable("studentId") Long studentId,
11                           @RequestParam(required = false) String name,
12                           @RequestParam(required = false) String email) {
13     studentService.updateStudent(studentId, name, email);
14 }
```

- @GetMapping 是 @RequestMapping(method = RequestMethod.GET) 的简写。
- @RequestMapping(method = RequestMethod.PUT, path = "/{studentId}") 是 Put 的完整版
- @RequestMapping 既可以是类级别的注释，也可以是方法级别的注释，会从前往后把类和方法的路径拼接

## Lecture 13 Test + Log

### Testing

- 测试类必须至少包含一个测试方法，不能是抽象类，且必须有一个构造函数(无则生成一个默认构造器)，**不需要是 public 的，但不能是 private**。
- 测试方法
  - i. 带有下列关键字 @Test @RepeatedTest @ParameterizedTest  
@TestFactory @TestTemplate @ParameterizedTest  
**@ParameterizedTest 中 candidate 会依次取 strings 中的所有值进行测试**

```
1  @ParameterizedTest
```

```

2 @ValueSource(strings = { "racecar", "radar", "able was I ere I saw
3 elba" })
4 void palindromes(String candidate) {
5     assertTrue(StringUtils.isPalindrome(candidate));

```

- ii. Test Methods可以在当前测试类中声明。也可以从父类或者接口中继承。测试方法不能返回值（`@TestFactory` 是例外）

- o `Assertions`

- i. `assertEquals(expected, actual)`：断言实际值与预期值相等。多次使用单个断言，不使用`assertAll()`，如果某个断言失败，测试方法会立即中断，后续代码不会执行。
- ii. `assertAll()`：方法用于同时执行多个断言，即使其中某些断言失败，也会继续执行其他断言。所有失败的断言会被收集，并最终作为一个 `MultipleFailuresError` 报告。

```

1 assertAll("Should return address of Oracle's headquarter",
2         () -> assertEquals("Redwood Shores", address.getCity()),
3         () -> assertEquals("Oracle Parkway", address.getStreet()),
4         () -> assertEquals("500", address.getNumber())

```

- o `Assumptions` 和 `Assertions` 的对比

`Assertions`：验证实际结果是否符合预期结果。用于测试代码的正确性。断言失败会导致测试方法被标记为 **失败 (failed)**。

`Assumptions`：验证前提条件是否满足。用于跳过当前不适用的测试方法（例如：环境条件不符合）。假设失败不会标记测试为失败，而是直接 **中止 (aborted)**。一般用来测试环境变量是否正确

```

1 assumeTrue("CI".equals(System.getenv("ENV")));
2 assumeTrue("DEV".equals(System.getenv("ENV")),
3             () -> "Aborting test: not on developer workstation");
4 assumingThat("CI".equals(System.getenv("ENV")),
5             () -> {assertEquals(2, calculator.divide(4, 2));}); // 只有前提对了才
会进行assert

```

## 生命周期方法

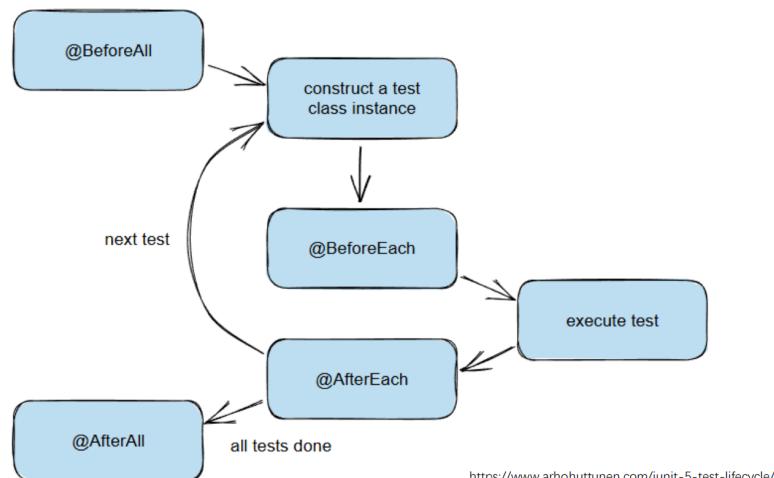
- o `@BeforeAll` `@AfterAll` 两者都必须是 `static`

- `@BeforeEach` `@AfterEach` 大部分情况不能是 `static` (如果使用`PER_METHOD`模式就可以)

**共同特点：**返回值类型必须为 `void`。不能是 `private` (可以是 `public`、`protected` 或 `Default`)。

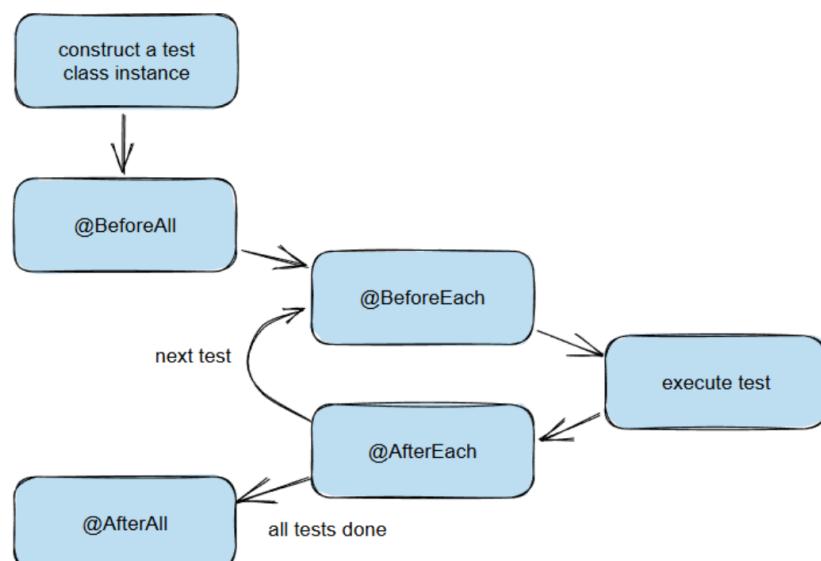
## 生命周期模式

- a. `PER_METHOD` 默认使用，JUnit 在执行每个测试方法前都会创建一个新的测试类实例。



- b. `PER_CLASS` 使用 `@TestInstance(Lifecycle.PER_CLASS)` 注解

- 在 `PER_CLASS` 模式下，`BeforeAll` 和 `AfterAll` 方法可以是非静态的。



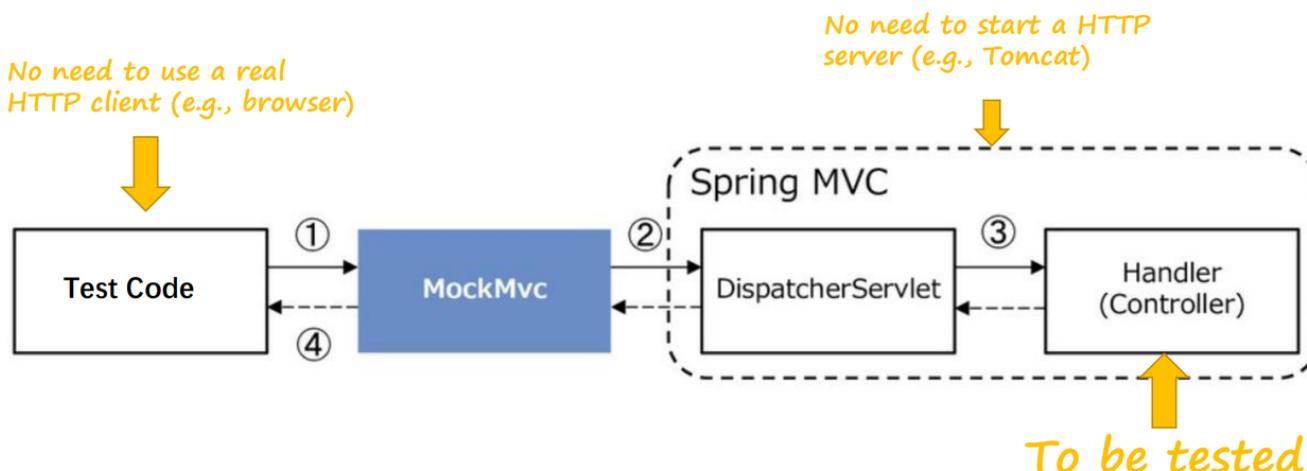
- `@Disabled`：禁用测试类或方法。`@EnabledForJreRange(min = JAVA_9, max = JAVA_11)`
- `@DisabledForJreRange(min = JAVA_9)`
- `@EnabledOnOs({LINUX, MAC})` `@DisabledOnOs(WINDOWS)`

**Spring Boot Testing** Spring Boot 默认你需要测试你的应用程序，会自动帮你加入 `test` 的依赖，**测试代码的目录结构与src的目录结构保持一致**。我们通过MockMvc来进行测试，真实的网络测试太麻烦

**Sanity Check:** `@SpringBootTest` 注解告诉Spring Boot查找主配置类（`@SpringBootApplication` 注解的类）并且使用主配置类来启动Spring应用的上下文（Application Context）。这样快速验证Spring上下文是否加载了需要的组件

**MockMvc:** 无需使用浏览器或其他HTTP客户端来发出请求 无需启动像Tomcat这样的嵌入式服务器

- a. 测试代码通过 `MockMvc` 模拟HTTP请求，设置URL、参数和请求类型（如GET或POST）。
- b. `MockMvc` 会拦截请求并将其转发给Spring MVC框架的核心组件 `DispatcherServlet`，无需依赖真实的网络层。`DispatcherServlet` 根据URL映射找到对应的控制器（Handler），并调用控制器的方法。
- c. `MockMvc` 接收控制器返回的HTTP响应，测试代码可以对响应内容、状态码等进行断言



<https://speakerdeck.com/rshindo/spring-fest-2020?slide=34>

- `@AutoConfigureMockMvc` 是 Spring Boot 提供的注解，用于在测试类中自动配置 MockMvc 实例。
- `.perform()` 用于构造和发送 HTTP 请求 `.andExpect()` 用于断言 HTTP 响应是否满足预期条件

```
1  @SpringBootTest
2  @AutoConfigureMockMvc
3  class StudentRestControllerTest {
4      @Autowired
5      private MockMvc mvc;
6      @Test
7      void getStudentsByEmail() throws Exception {
8          mvc.perform(get("/api/students/?email={email}", "jack"))
9              .andExpect(status().isOk())
10             .andExpect(jsonPath("$.*", hasSize(1)))
```

```
11         .andExpect(jsonPath("$.name").value("Jack"))
12         .andDo(print());
13     }
14 }
```

通过 `ObjectMapper` 来发送 `post` 或者 `put` 请求

```
1 Student stu = new Student("Jack", "jack@mail.com");
2 ObjectMapper mapper = new ObjectMapper();
3 String json = mapper.writeValueAsString(stu);
4 mvc.perform(post("/api/students/save")
5             .contentType(MediaType.APPLICATION_JSON)
6             .content(json))
7         .andExpect(status().isOk());
```

## Logging

- 使用了大量的 `println()`，可能会影响性能，所以转而使用 Logging
- 日志级别从“最少日志”到“最多日志”依次为：

**OFF -> FATAL -> ERROR -> WARN -> INFO -> DEBUG -> TRACE -> ALL**

- 如果日志级别为 **INFO**，则不会显示 **DEBUG** 和 **TRACE**，只会显示比自己更高级的

## Component

- **Logger (记录器)** 是一个命名实体，负责根据日志级别（如 DEBUG、INFO、WARN、ERROR 等）决定是否捕获日志。Logger 捕获日志后，会将日志消息传递给 **Appender (附加器)**。
- **Appender (附加器)** 负责将日志信息输出到各种目标的组件，决定日志输出目的地 `控制台` `Console` `文件File` `数据库Database` `远程服务器Remote Server`，内置和自定义 `Appender` `ConsoleAppender` `FileAppender` `SMTPAppender` `自定义`
- **Layout (布局或格式化器)** 负责将原始日志消息转化为人类可读的格式 `SimpleFormatter`（简单格式化器）`XMLFormatter`（XML 格式化器）`HTMLFormatter`（HTML 格式化器）
- `log4j.properties` 配置 `日志级别 (Log Levels)` `Appender (附加器) 的信息` `日志输出的目标文件名` `Layout (布局) 的格式化方式`
- **SLF4J (Simple Logging Facade for Java)**
  - 它是一个日志框架的抽象层，提供了一组统一的 API。支持多种日志框架（默认使用 `Logback`），包括：`java.util.logging` `Log4j` `Logback`

- 默认配置: Appender: `ConsoleAppender` Layout: `PatternLayoutEncoder`  
Level: `INFO`
- Spring Boot 默认日志功能不计入最终项目日志记录要求