JavaEE平台技术
Spring框架核心

邱明 博士

厦门大学信息学院

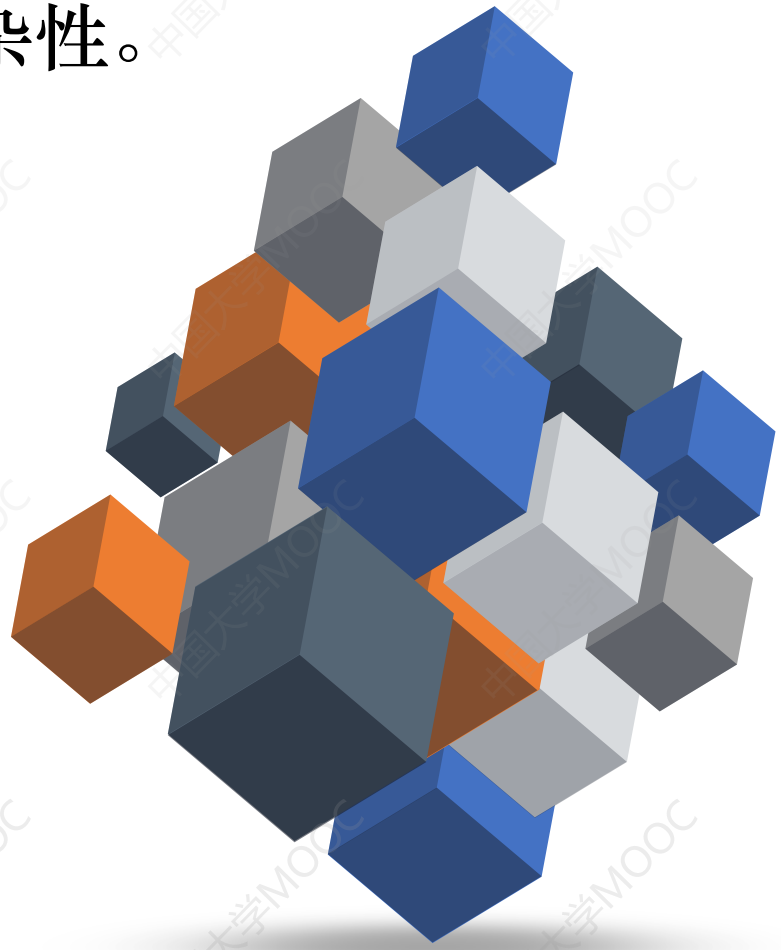
mingqiu@xmu.edu.cn

提纲

- 软件模块化
- Spring技术栈
- Servlet
- Spring容器
- 对象生命周期
- 控制反转（IoC）

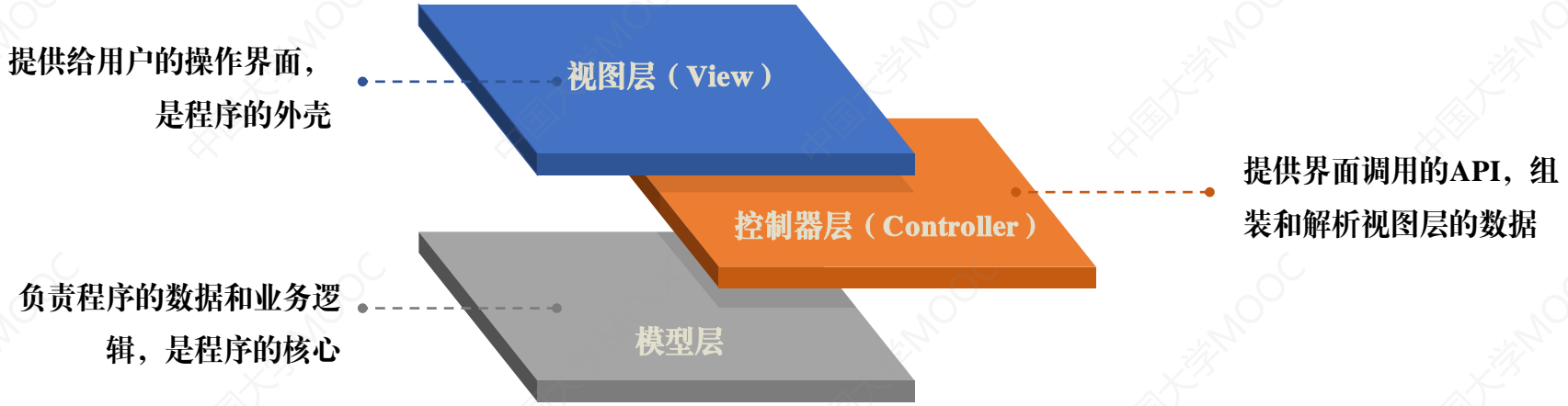
1.1 软件模块化

- 把一个程序分割成一些不同的部分，可以在某种程度上减少它的复杂性。



1.1 软件模块化

- Model-View-Controller将软件用户界面和业务逻辑分离



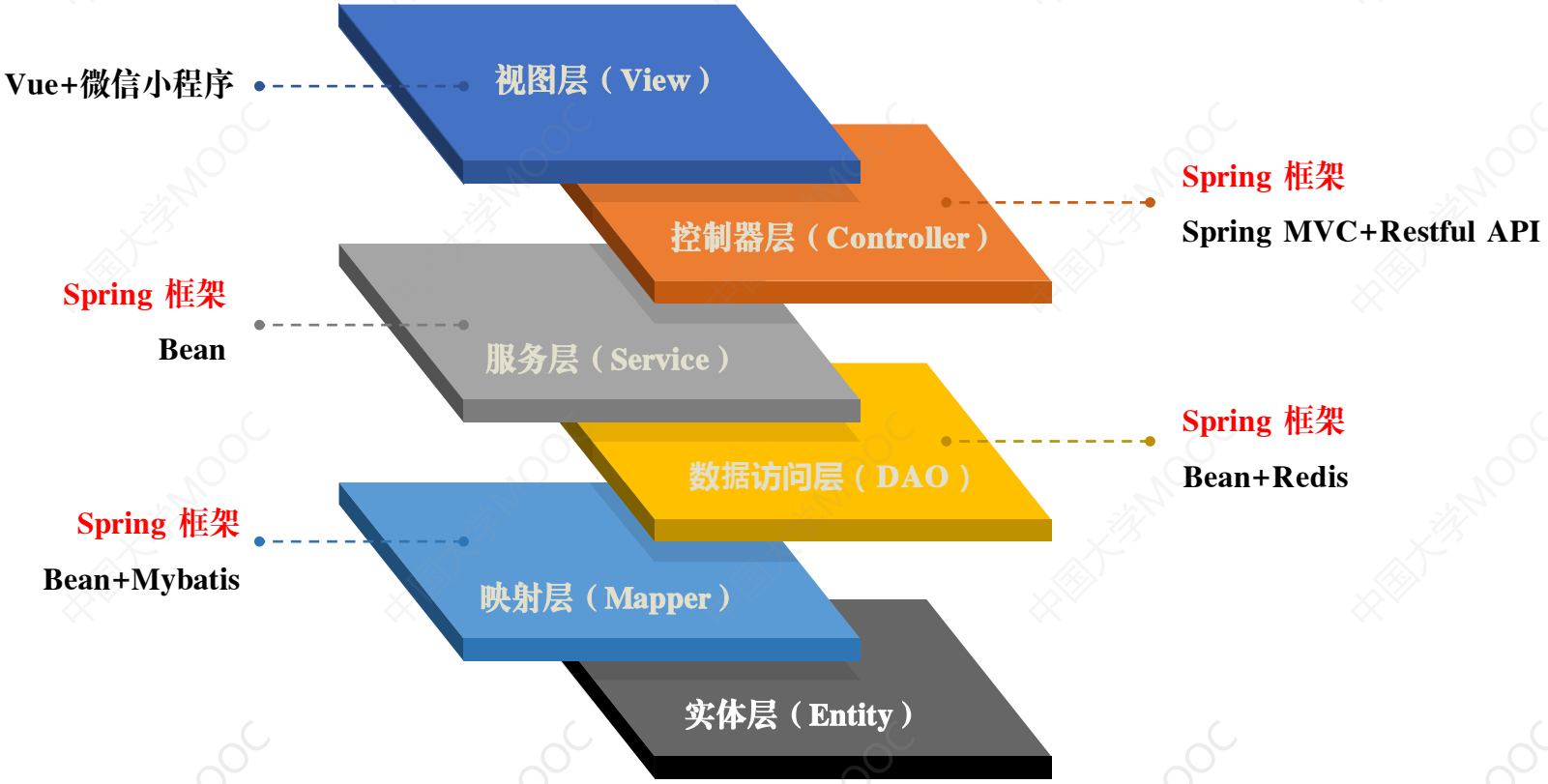
1.1 软件模块化

- 每一层承担特定的职能，高层依赖于低层

1.1 软件模块化

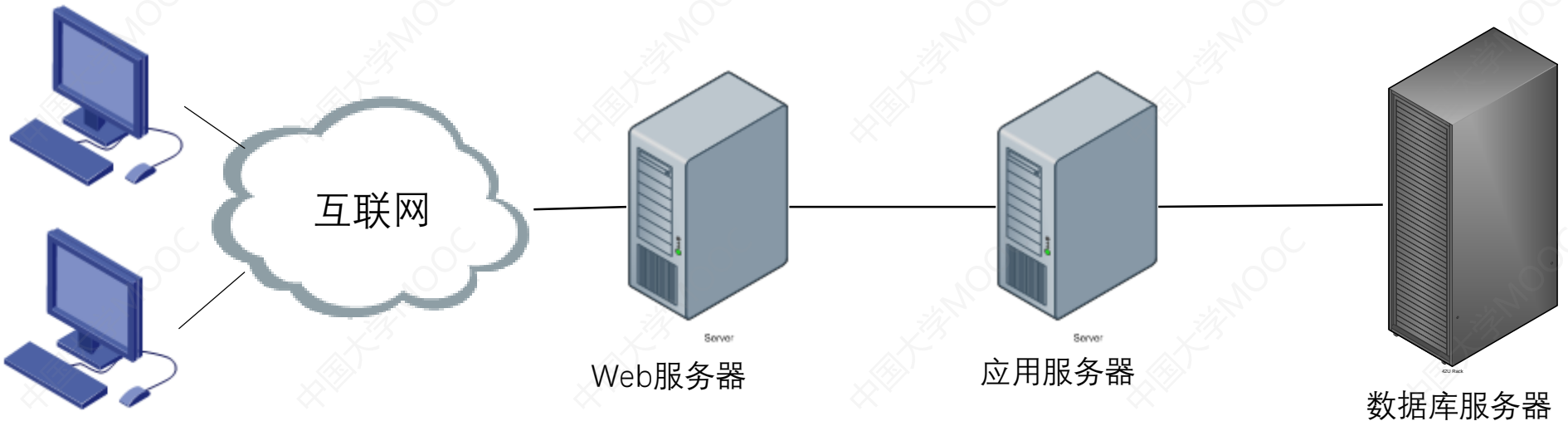
- 多层体系结构的优点
  - 结构简单，便于不同技能的程序员分工负责不同的层
  - 便于测试，每一层都可以独立测试
  - 变更可控，可以把代码的变更控制在一层之内，不会影响其他的层

1.1 软件模块化

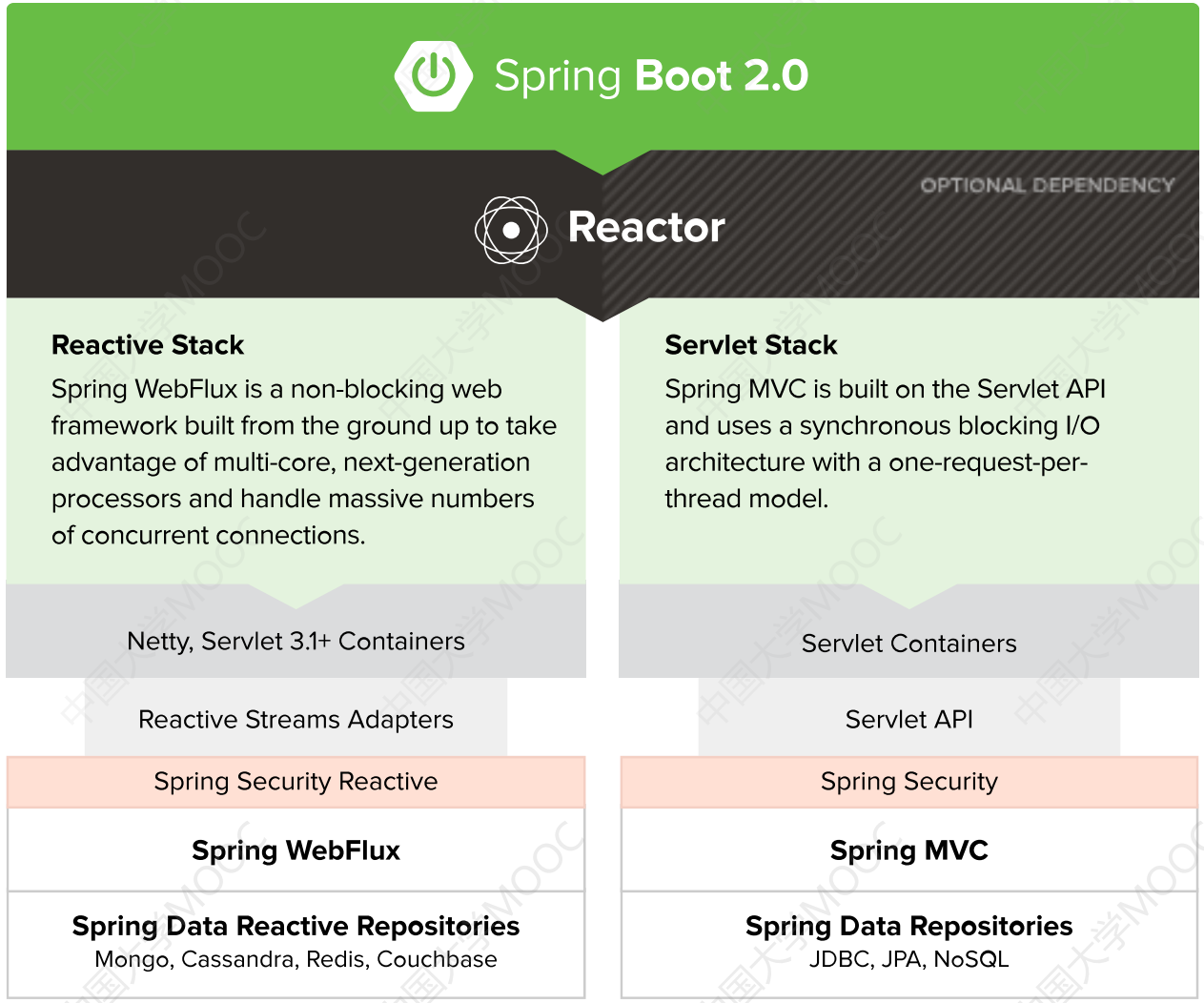

Vue+微信小程序
视图层（View）
Spring 框架
Spring MVC+Restful API
控制器层（Controller）
Spring 框架
Bean
服务层（Service）
Spring 框架
Bean+Redis
数据访问层（DAO）
Spring 框架
Bean+Mybatis
映射层（Mapper）
实体层（Entity）

1.1 软件模块化

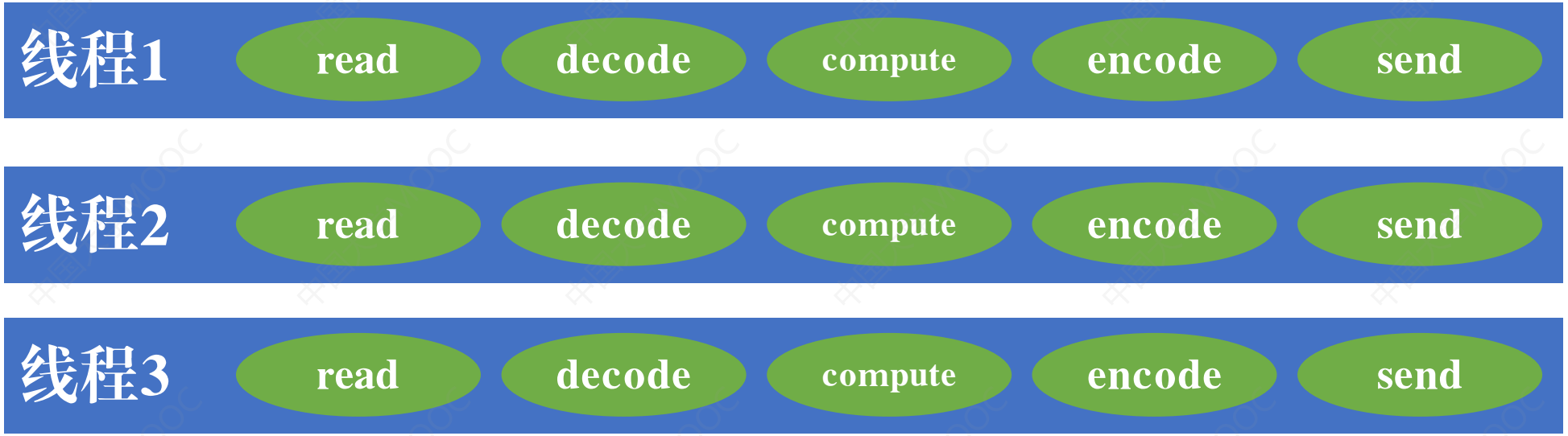
- 网络结构



1.2.Spring技术栈

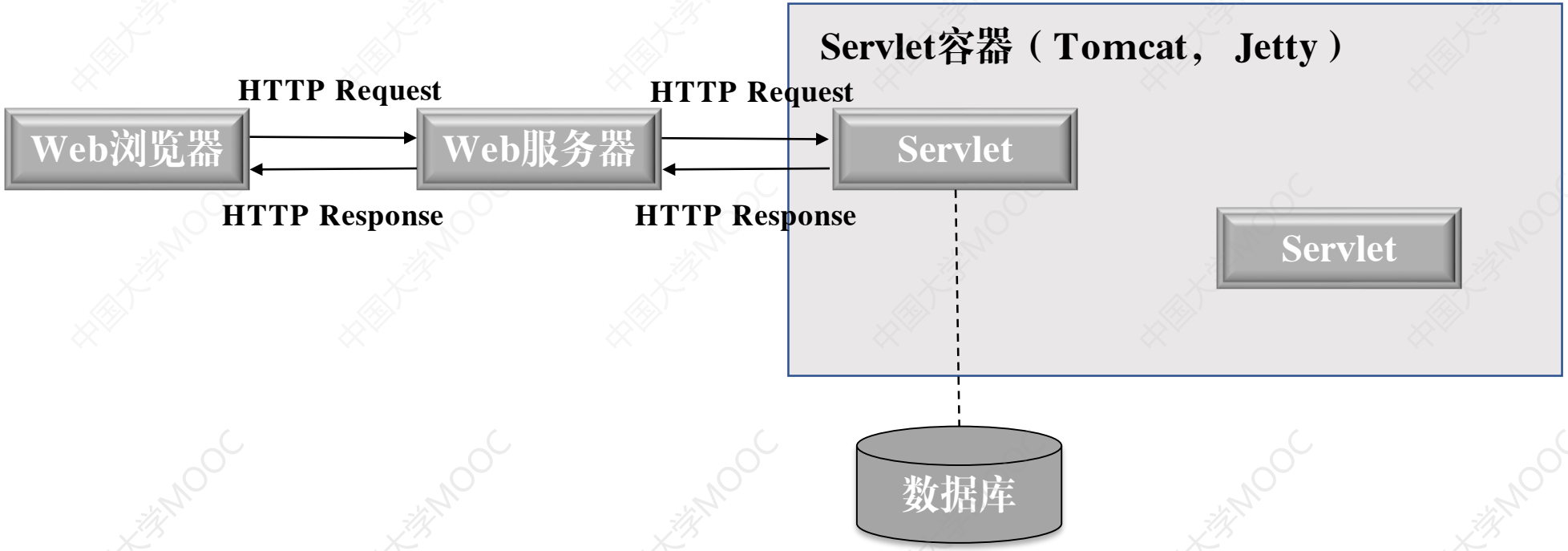


1.2.Spring技术栈



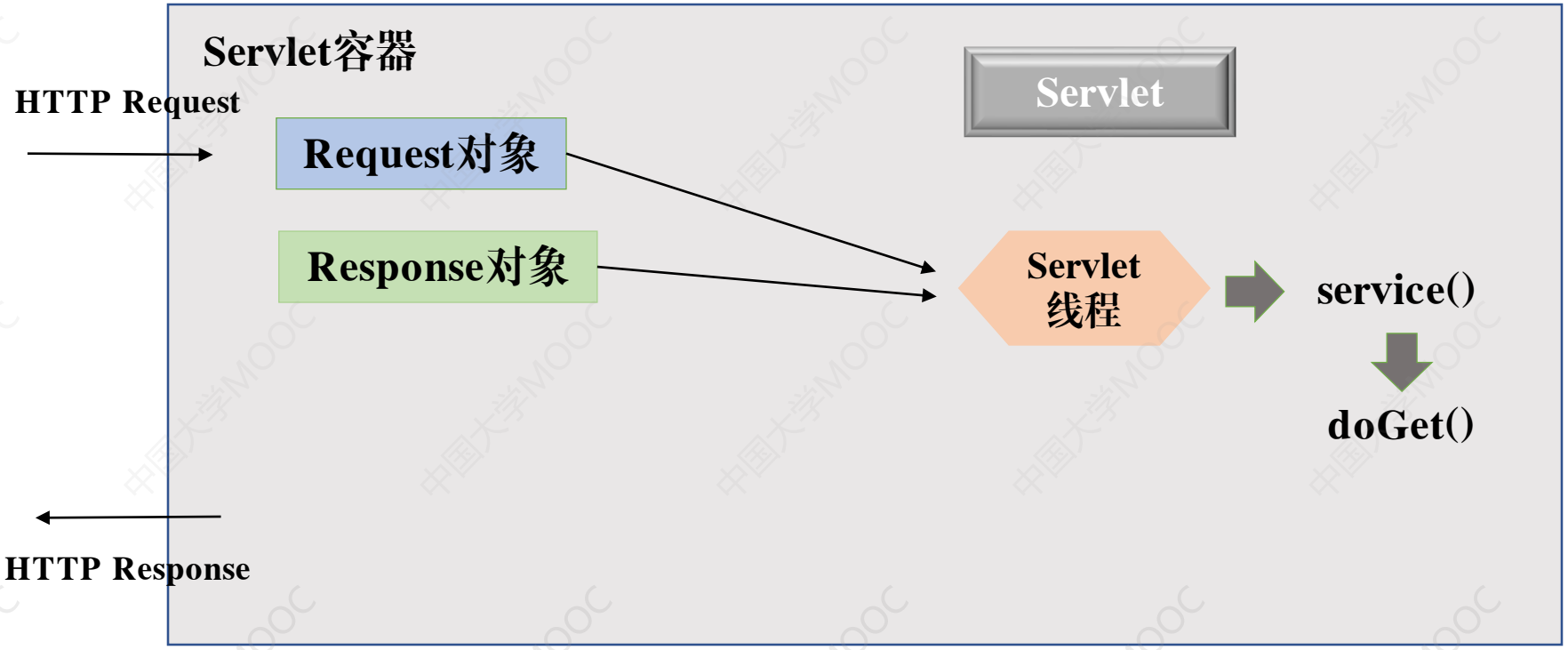
1.3. Servlet

- Servlets
  - 运行在 Web 服务器或应用服务器上的Java程序，它可以收集来自前端的用户输入数据，以动态生成网页的方式呈现来自数据库的记录，



1.3. Servlet

- Servlet的工作机制

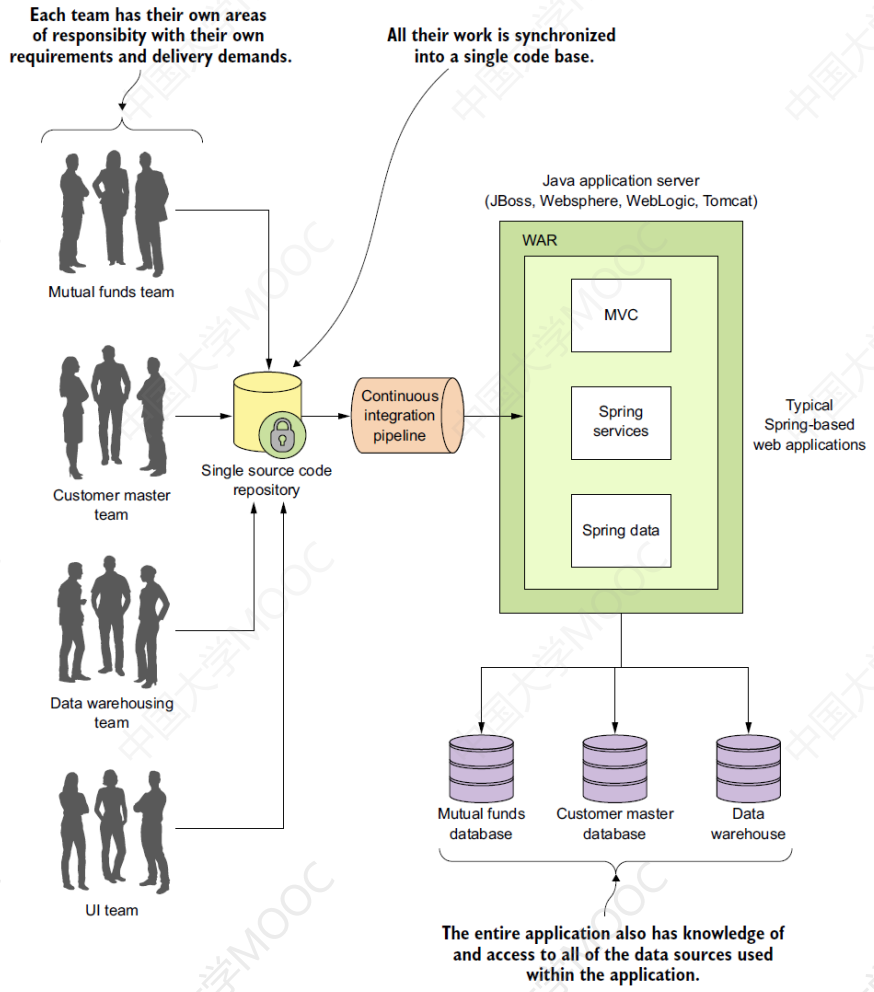


1.3. Servlet

- Servlets的作用
  - 通讯功能
  - Servlet对象的生命周期管理
  - 多线程支持
  - 安全性支持

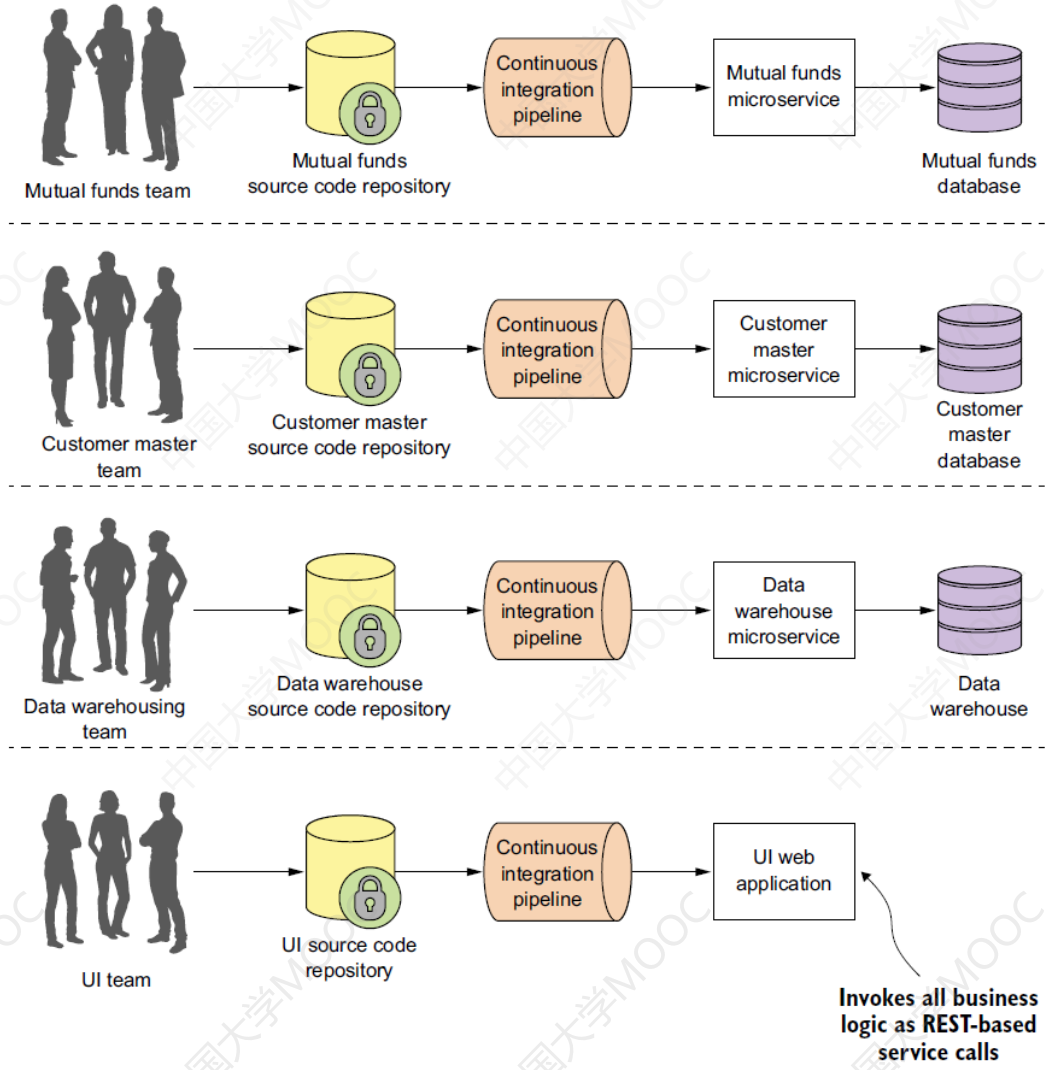
2.Spring 5.0 概览

- 大怪兽型应用



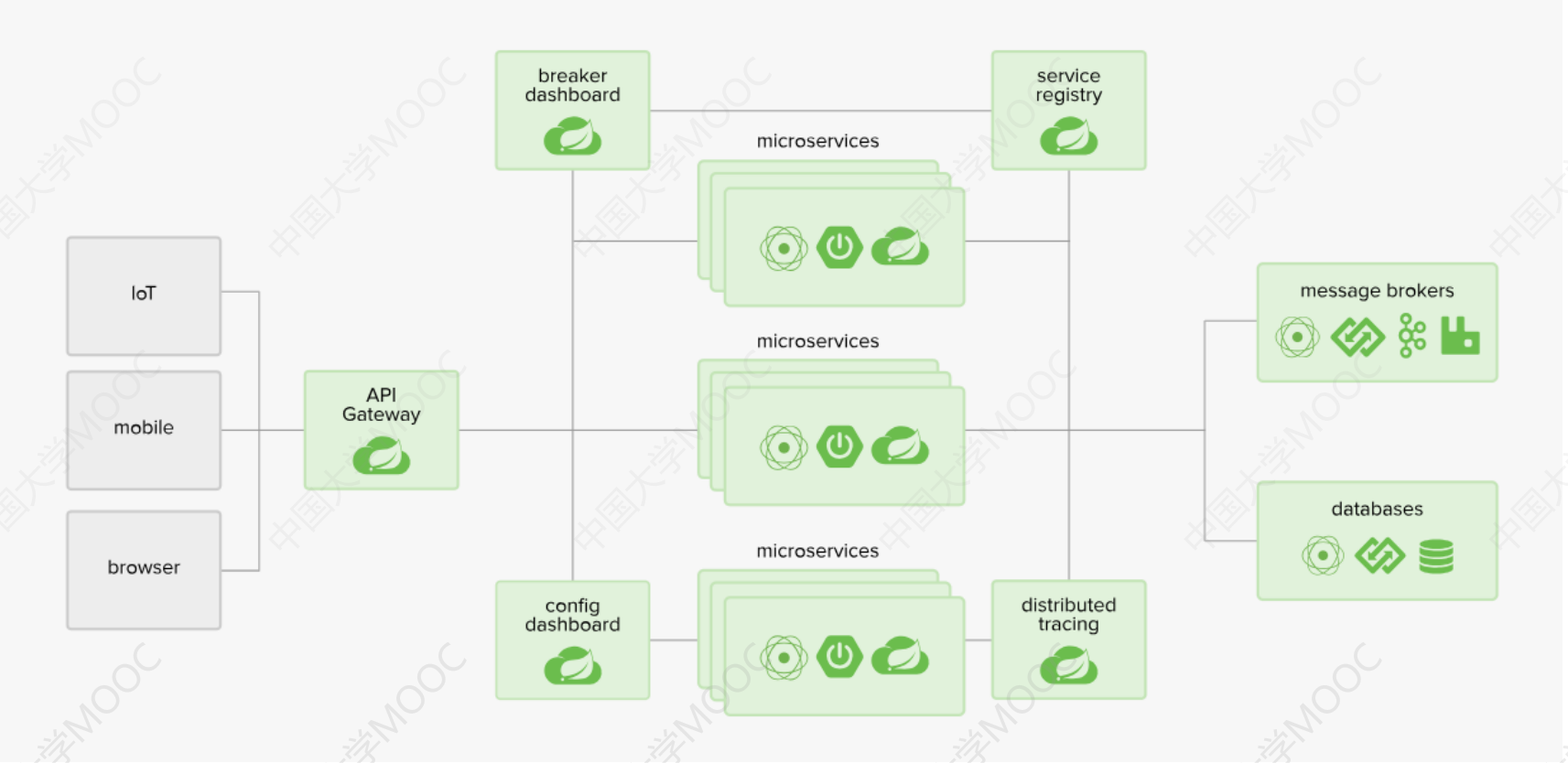
2.Spring 5.0 概览

- 微服务体系结构应用



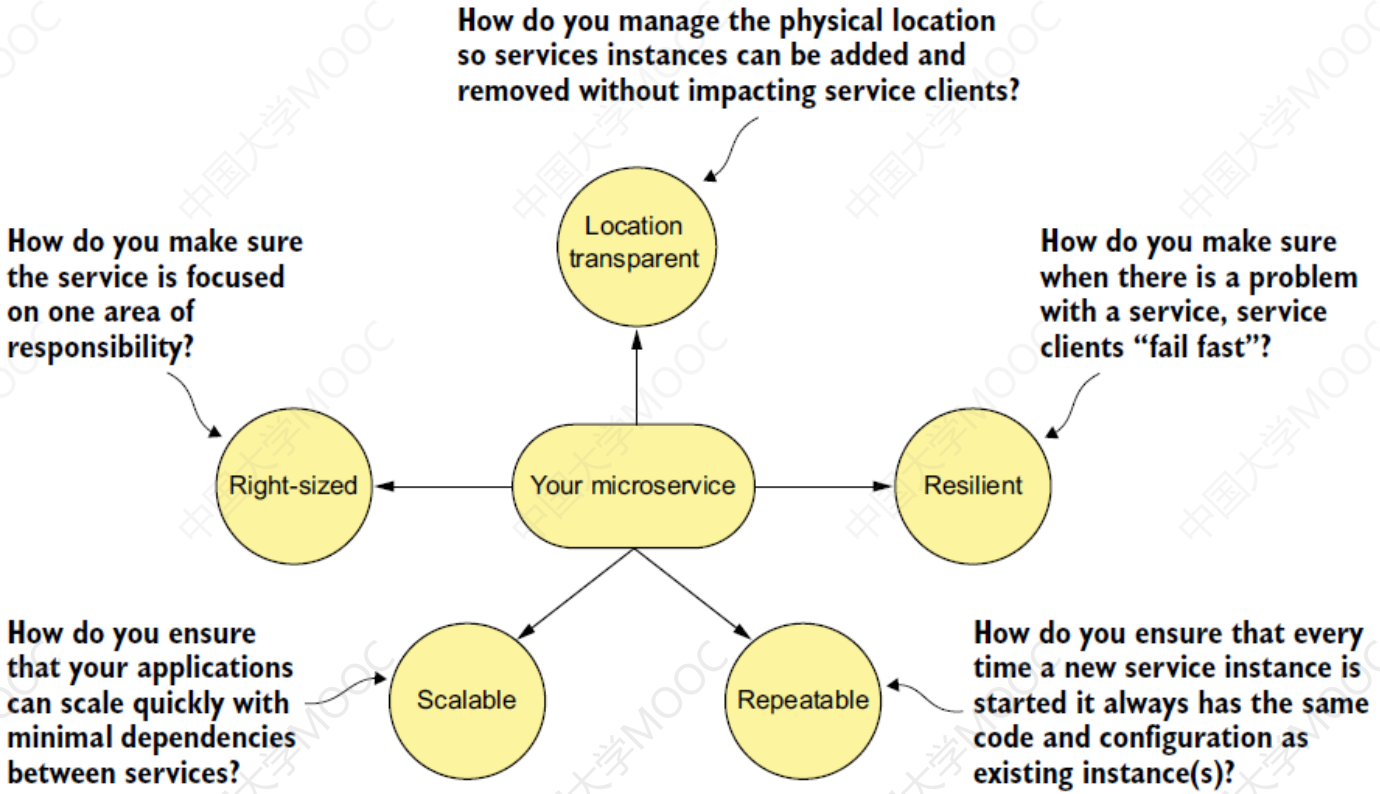
2.Spring 5.0 概览

- Spring Cloud



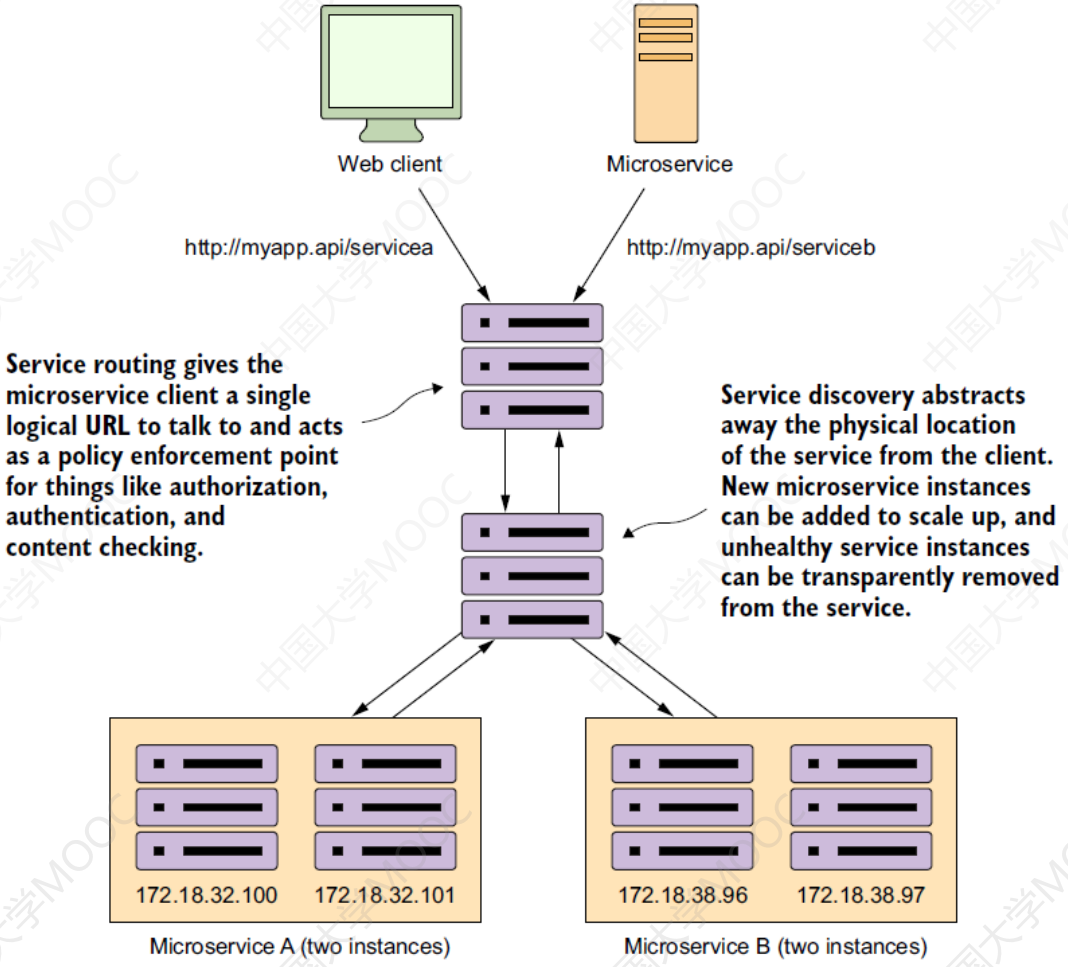
2.Spring 5.0 概览

- 微服务体系结构应用



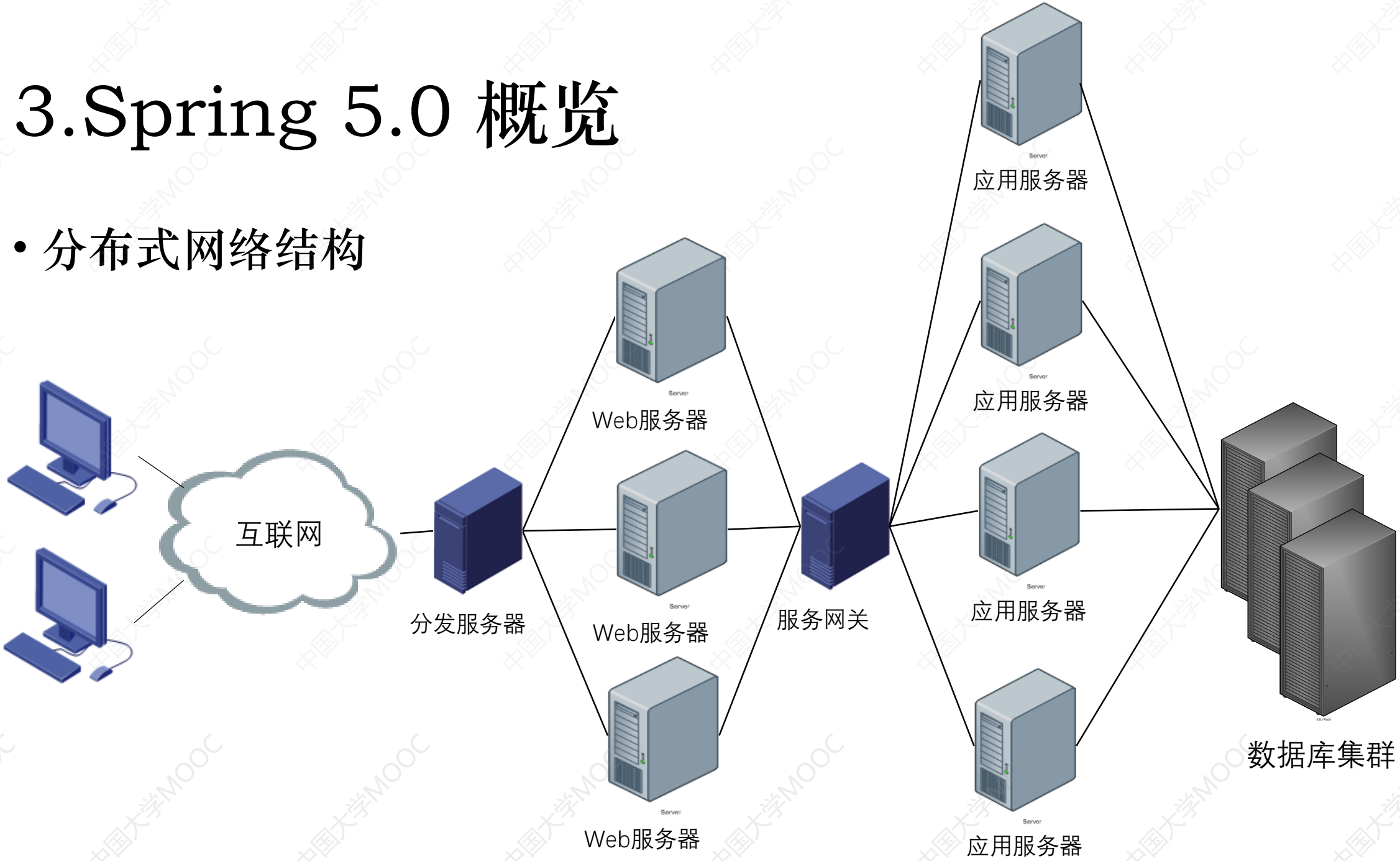
3.Spring 5.0 概览

- 服务路由



3.Spring 5.0 概览

- 分布式网络结构

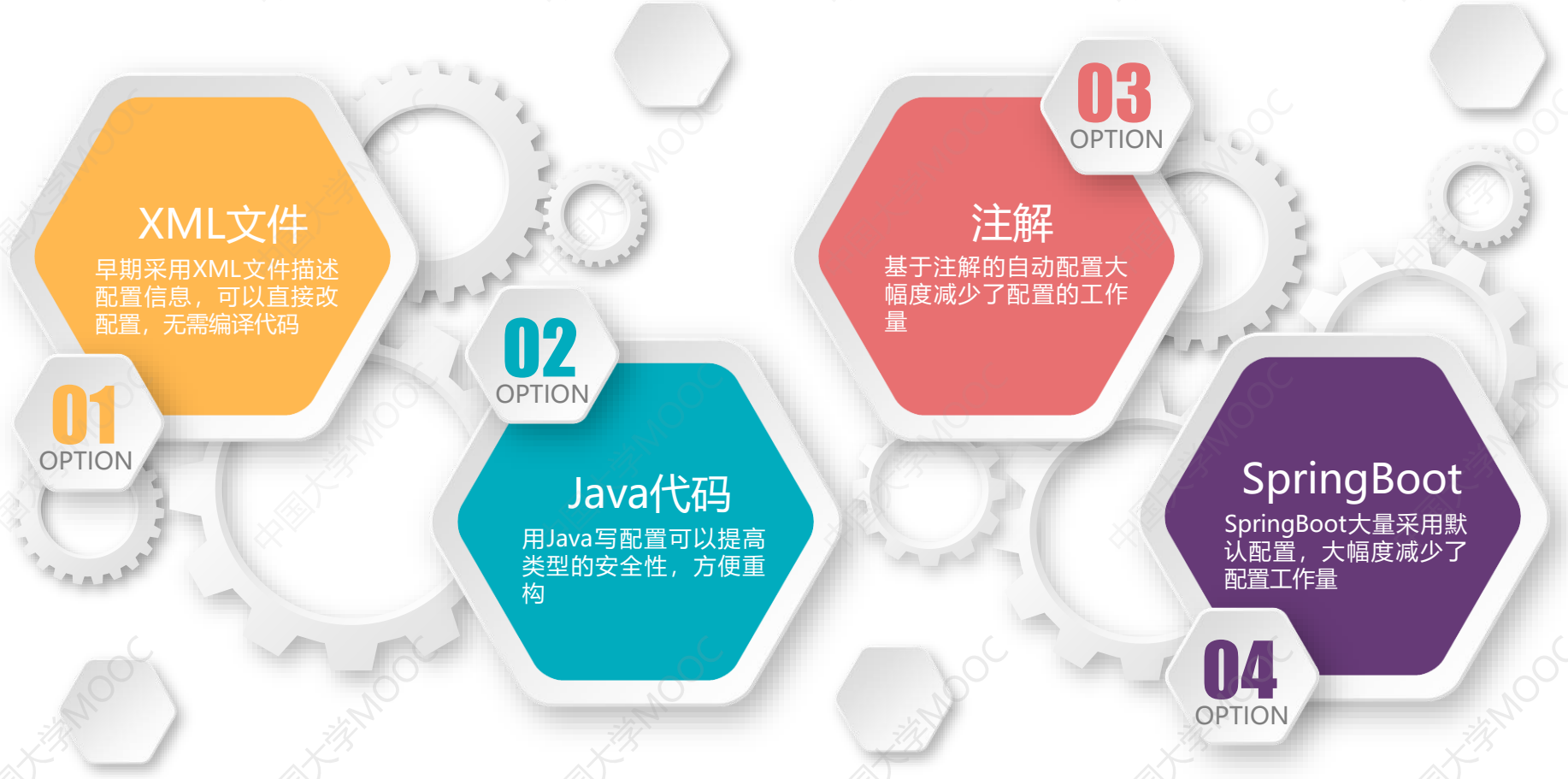


3. Spring容器

- Spring容器来负责创建对象并把对象关联起来提供服务。
- 容器提供了公共服务
- 容器依赖于配置信息

3. Spring容器

- Spring框架的配置



3. Spring容器

- Spring Bean对象的注解
  - @Component
    - id - Bean对象的名称，id属性名称理论上可以任意命名，默认为类名且首字母小写
    - scope - singleton（默认值），prototype，request，session, global-session
  - @Controller, @Service， @Repository与@Component含义相同，分别用于标识Controller层，Service层，DAO层的Bean对象

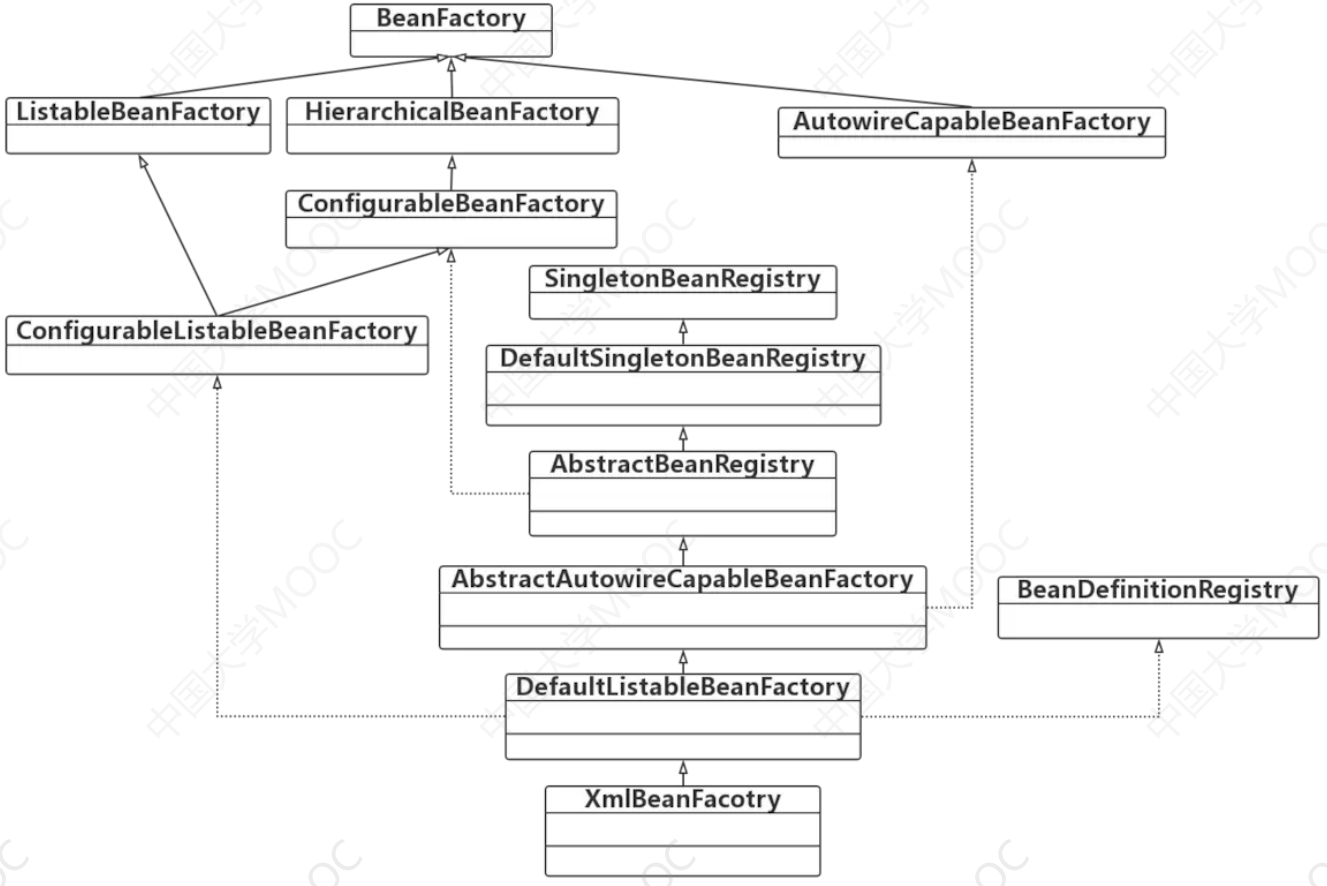
```
@Component("customerController")
@Scope("prototype")
```

3. Spring容器

- Spring拥有两种类型的容器
  - BeanFactory
    - BeanFactory负责读取bean配置信息，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的生命周期，每次获取对象时才会创建对象
  - ApplicationContext
    - ApplicationContext由BeanFactory派生而来， 同时也继承了容器的高级功能，如：MessageSource（国际化资源接口）、ResourceLoader（资源加载接口）、ApplicationEventPublisher（应用事件发布接口）等，提供了更多面向实际应用的功能。在容器启动时就会创建所有的对象

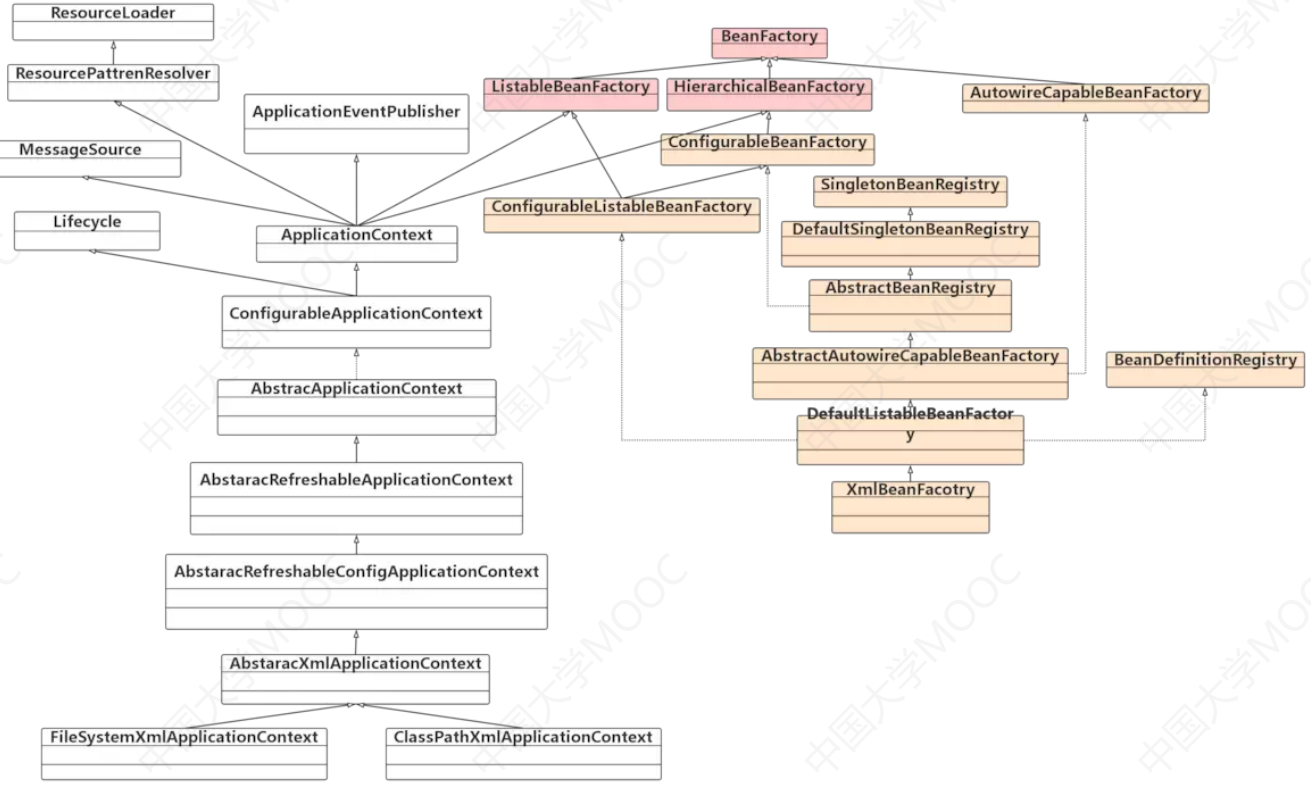
3. Spring容器

- BeanFactory类结构
  - BeanFactory: 主要的方法是getBean(StringbeanName)，从容器中返回特定名称的Bean。
  - ListableBeanFactory：定义了访问容器中Bean 基本信息的若干方法，如查看Bean 的个数、获取某一类型Bean 的配置名、查看容器中是否包括某一Bean 等方法；
  - HierarchicalBeanFactory：父子级联IoC 容器的接口，子容器可以通过接口方法访问父容器；
  - ConfigurableBeanFactory：增强了IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；
  - AutowireCapableBeanFactory：定义了将容器中的Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；
  - SingletonBeanRegistry：定义了允许在运行期间向容器注册单实例Bean 的方法；
  - BeanDefinitionRegistry：Spring 配置文件中每一个<bean>节点元素在Spring 容器里都通过一个BeanDefinition 对象表示，它描述了Bean 的配置信息。而BeanDefinitionRegistry 接口提供了向容器手工注册BeanDefinition 对象的方法。



3. Spring容器

- ApplicationContext类结构
  - ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了ApplicationListener 事件监听接口的Bean 可以接收到容器事件，并对事件进行响应处理。在ApplicationContext 抽象实现类AbstractApplicationContext 中，我们可以发现存在一个ApplicationEventMulticaster，它负责保存所有监听器，以便在容器产生上下文事件时通知这些事件监听 者。
  - MessageSource：为应用提供i18n 国际化消息访问的功能；
  - ResourcePatternResolver ： 所有ApplicationContext 实现类都实现了类似于PathMatchingResourcePatternResolver 的功能，可以通过带前缀的Ant 风格的资源文件路径装载Spring 的配置文件。
  - LifeCycle：该接口是Spring 2.0 加入的，该接口提供了start()和stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体Bean 实现，ApplicationContext 会将start/stop 的信息传递给容器中所有实现了该接口的Bean，以达到管理和控制JMX、任务调度等目的。
  - ConfigurableApplicationContext 扩展于ApplicationContext，它新增加了两个主要的方法：refresh()和close()，让ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用refresh()即可启动应用上下文，在已经启动的状态下，调用 refresh()则清除缓存并重新装载配置信息，而调用close()则可关闭应用上下文。这些接口方法为容器的控制管理带来了便利.



3. Spring容器

- Spring Bean的生命周期从创建容器开始，到容器销毁Bean为止。

Bean级生命周期接口

- BeanNameAware
- BeanFactoryAware
- ApplicationContextAware
- IntializingBean
- DisposableBean

只影响一个Bean的接口

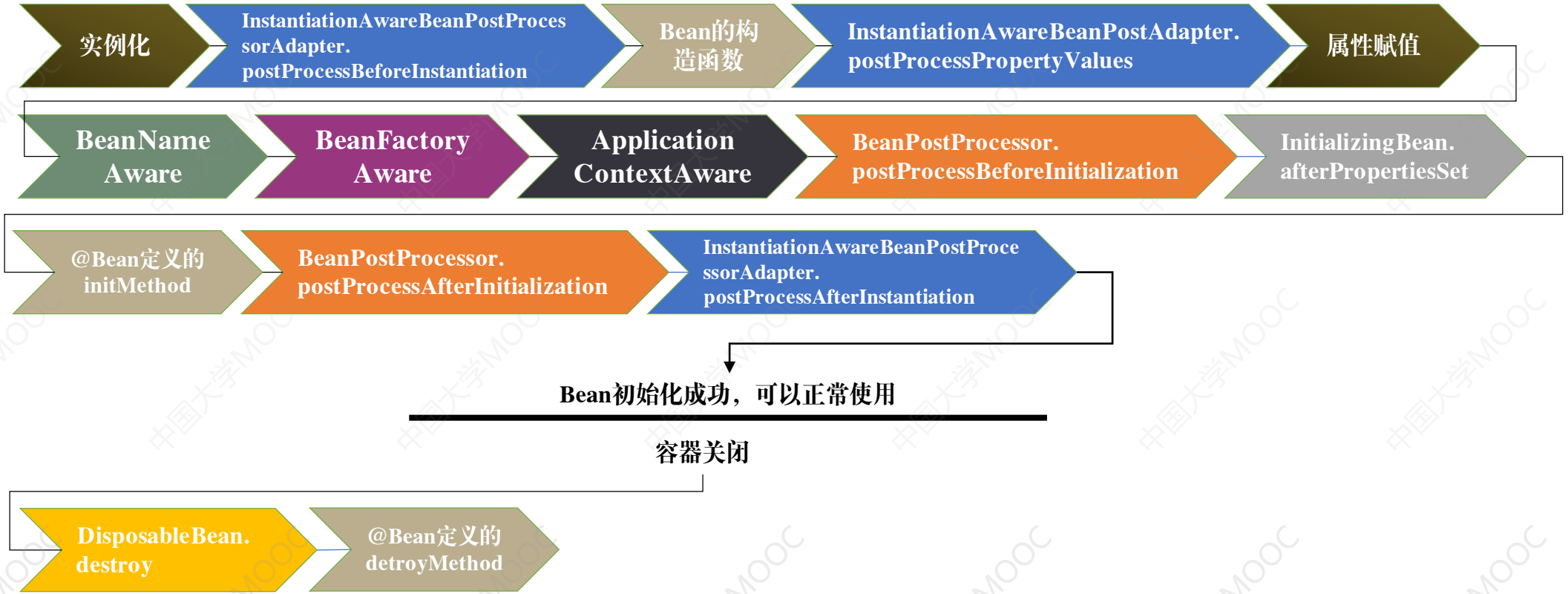
容器级生命周期接口

- InstantiationAwareBeanPostProcessorAdapter
- BeanPostProcessor

影响多个Bean的接口

3. Spring容器

- Bean的生命周期



3. Spring容器

- 无所不知的Aware

BeanNameAware	BeanFactoryAware	ApplicationContext Aware
• void setBeanName(String beanName)	• viod setBeanFactory (BeanFactory var1) throws BeansException	• void setApplicationContext (ApplicationContext context) throws BeansException
待对象实例化并设置属性之后调用该方法设置BeanName	待调用setBeanName之后调用该方法设置BeanFactory	获得ApplicationContext

3. Spring容器

- Bean级生命周期接口

InitializingBean

- void afterPropertiesSet() throws Exception;

属性赋值完成之后调用

DisposableBean

- void destroy() throws Exception;

关闭容器时调用

3. Spring容器

- 容器级生命周期接口

InstantiationAware BeanPost

- Object postProcessBeforeInstantiation (Class<?> beanClass, String beanName) throws BeansException;

在Bean对象实例化前调用

- boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException;

在Bean对象实例化后调用

- PropertyValues postProcessPropertyValues(PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeansException;

在（通过配置）设置某个属性前调用

BeanPostProcessor

- Object postProcessBeforeInitialization(Object o, String s) throws BeansException;

实例化完成前

- Object postProcessAfterInitialization(Object o, String s) throws BeansException;

全部实例化完成以后调用该方法

4. 控制反转

- 控制反转是指Bean对象之间的依赖不由它们自己管理，而是由Spring容器负责管理对象之间的依赖
- Spring容器采用依赖注入（DI）的方式实现控制反转

4. 控制反转

- 注解方式
  - @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会根据变量的名称来查找。

4. 控制反转

- @Autowired 标注在 Setter 方法上

```
@Component
public class Boss_Setter {

    private Car car;
    private Office office;

    @Autowired
    public void setCar(Car car){
        this.car = car;
    }

    @Autowired
    public void setOffice(Office office){
        this.office = office;
    }

}
```

4. 控制反转

- @Autowired 标注在构造方法上

```
@Component
public class Boss_constructor {
    private Car car;
    private Office office;

    @Autowired
    public Boss_constructor(Car car, Office office){
        this.car = car;
        this.office = office;
    }
}
```

4. 控制反转

- @Autowired 标注在属性上

```
@Component
public class Boss_property {
    @Autowired
    private Car car;
    @Autowired
    private Office office;
}
```