InversionOfControl

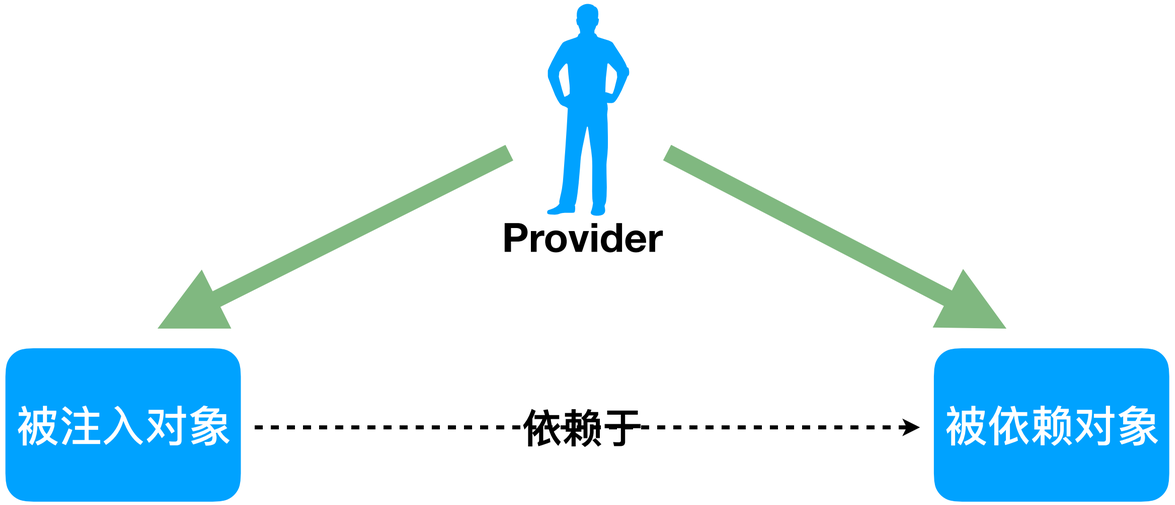
IoC

IoC-ServiceProvider

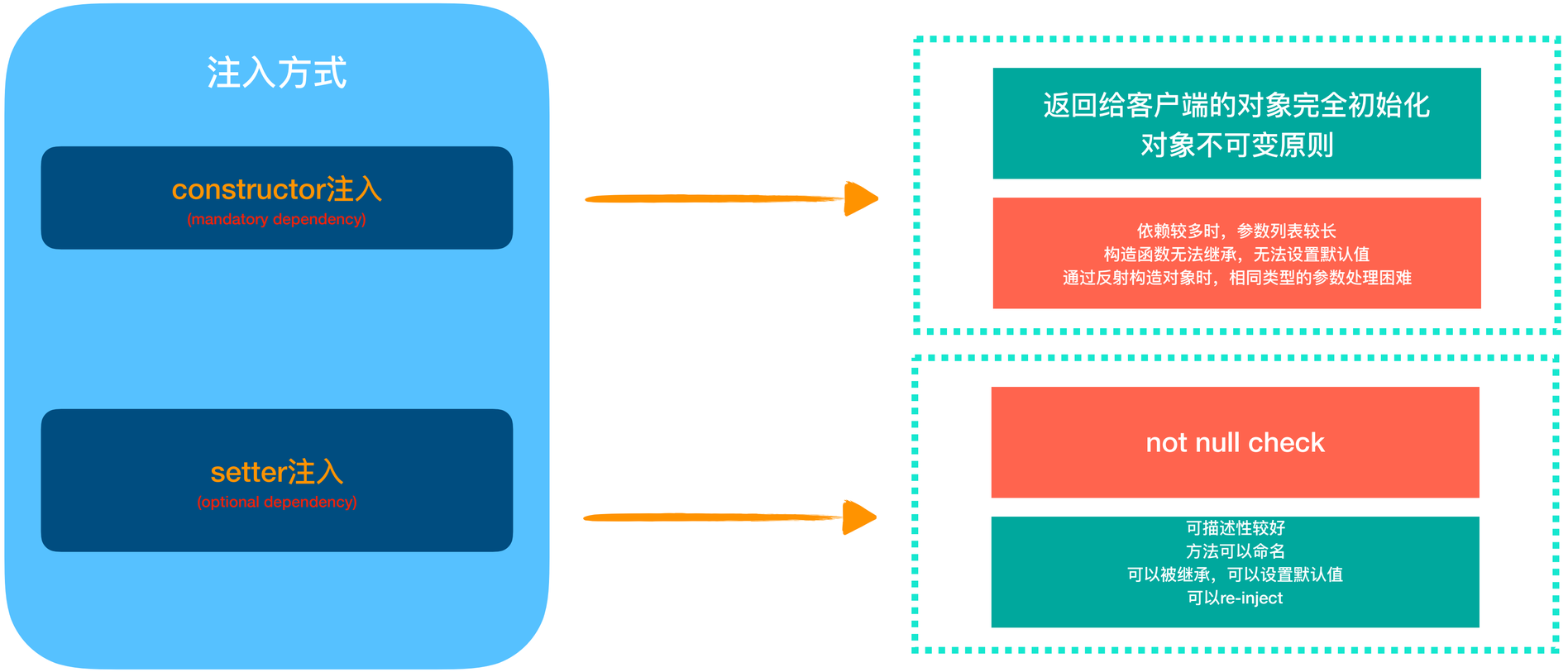
职责

对象创建
(解耦合，将创建逻辑从客户端代码分离)

依赖绑定
(识别业务对象之间的依赖关系)
(将对象的依赖项绑定在对象中)
(保证对象在使用时处于就绪状态)



IoC-ServiceProvider



实现方式：直接编码/配置文件(xml配置依赖关系)/元信息管理(juice/spring-annotation/javaconfig)

Bean(1)-xml

```xml
<beans> //对所有的bean设置公用属性，default-init-method/default-destroy-method/default-lazy-init等，也可设置description等基本属性
    <bean id="pojoBean" name="outBean" class="cn.deepclue.pojoClass"> // init-method/destroy-method/lazy-init/depend-on/autowired/scope
        <!-- 构造函数注入-->
        <constructor-arg index="1" value="intValue" />
        <constructor-arg index="0" value="stringValue-1" /> //同类型的构造函数参数可以用index来区分
        <constructor-arg index="2" value="stringValue-2" />
        <!--
        <constructor-arg type="int" value="intValue" /> //原生类型直接注入
        <constructor-arg type="java.lang.String" value="stringValue" />
        -->
        <constructor-arg ref="anotherBean" /> //引用类型

        <!-- setter属性注入 -->
        <property name="refProp" ref="refPropBean"/>
        <property name="stringProp" value="stringValue" />
        <property name="idrefProp">
            <idref bean="idRefPropBean"/> //配置解析阶段检测bean是否存在，无需在运行时检测
        </property>

        <!-- 集合注入 -->
        <property name="alist">
            <list>
                <value> value1 </value>
                <!--
                <ref bean="cRefBean" />
                -->
            </list>
        </property>
        <property name="amap">
            <map>
                <entry key="key">
                    <value> somevalue </value>
                    <!--
                    <ref bean="cRefBean" />
                    -->
                </entry>
            </map>
        </property>
        <property name="aset">
            <set>
                <value> value1 </value>
                <!--
                <ref bean="cRefBean" />
                -->
            </set>
        </property>
    </bean>
</beans>
```

```xml
<beans>
    <!-- 静态工厂方法注入 -->
    <bean id="serviceBean" class="cn.deepclue.staticServiceFactoryCls" factory-method="getInstance">
            <constructor-args ref="factoryArgsBean" />
    </bean>
    <bean id="compoundServiceBean" class="compoundServiceCls" >
        <property name="serviceInstance" ref="serviceBean" />
    </bean>


    <!-- 方法注入 method injection-->
    <bean id="myCommand" class="cn.deepclue.CustomCommandCls" scope="prototype">
        <!-- inject dependencies here -->
    </bean>>
    <bean id="commandManager" class="cn.deepclue.CommandManagerCls">
            <lookup-method name="createCommand" bean="myCommand"/>
    </bean>

    <!-- 跨scope注入 -->
    <bean id="userPreference" class="cn.deepclue.userPreferenceCls" scope="session">
        <aop:scoped-proxy/>
    </bean>
    <bean id="userService" class="cn.deepclue.userService">
        <property name="userPreference" ref="userPreference"/>
    </bean>

    <!-- 联名 -->
    <alias name="systemA-datasource" alias="mainAppDataSource" />
    <alias name="systemB-datasource" alias="secondaryAppDataSource" />
</beans>
```

Bean(1)-xml

内部bean

除了外围bean，无法注入其他bean，也无法被其他bean直接访问

Lazy-init

disable pre-instantiation

method-injection

跨作用域注入

循环依赖

Depend-on

保障隐含依赖情形下的正确初始化

Autowired

避免二义性

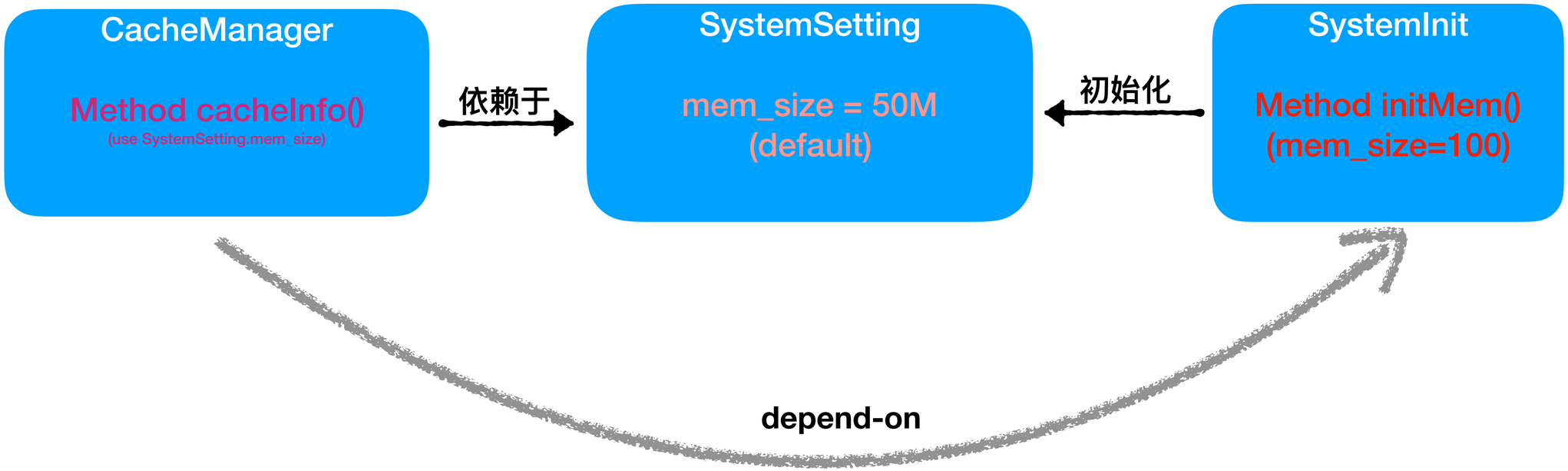
什么类型的pojo对象不适合用容器管理

domain object

Singleton-scope的bean随容器初始化而初始化，其余作用域的bean需要时初始化

Bean(1)-xml

Depend-on的使用场景



bean依赖关系不明显，需要显式指定，保障正确的初始化顺序

Bean(2)-scope

singleton

per-container
per-bean

stateless

application

request

websocket

session

global
request

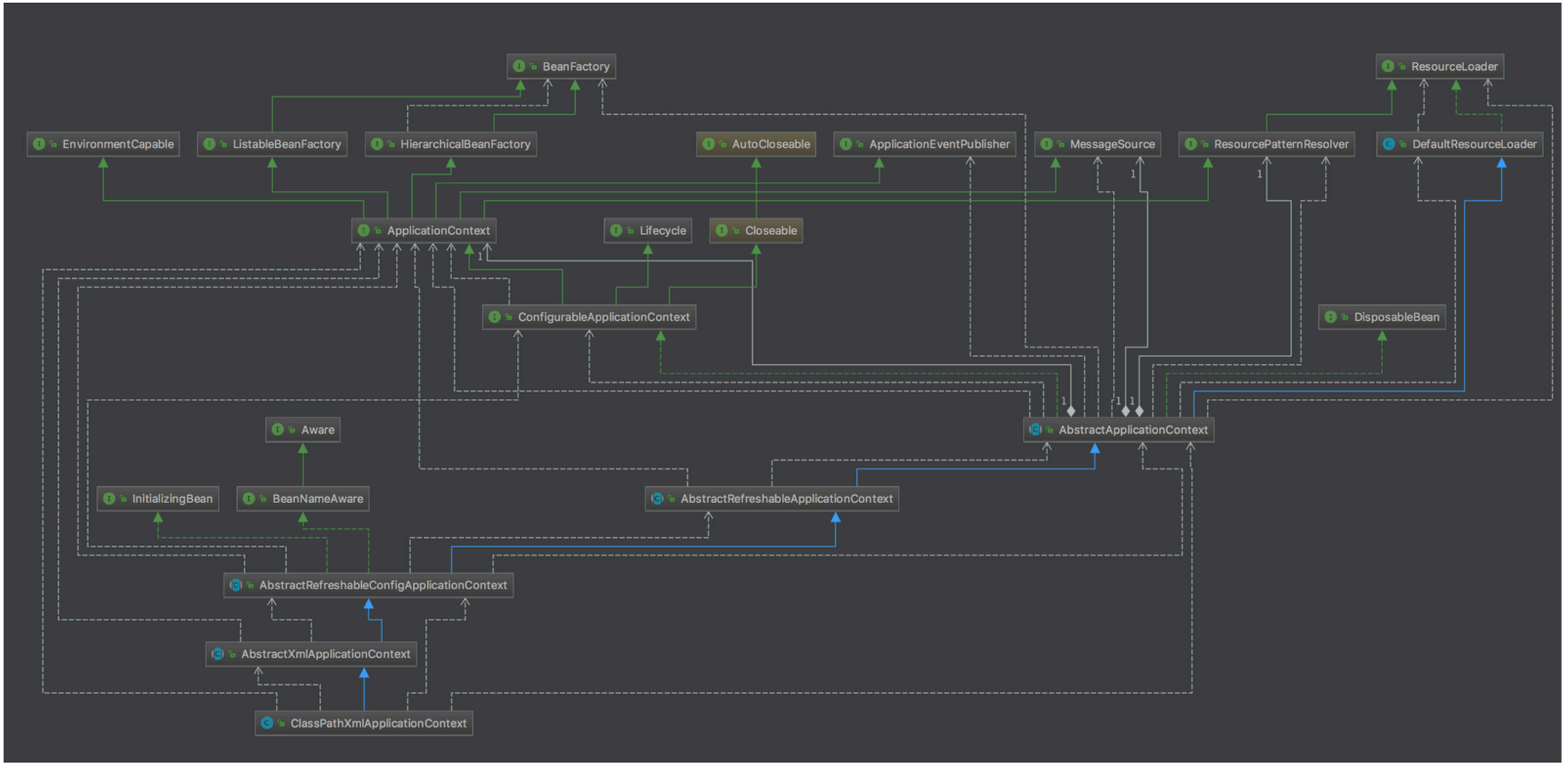
prototype

per-call
per-bean

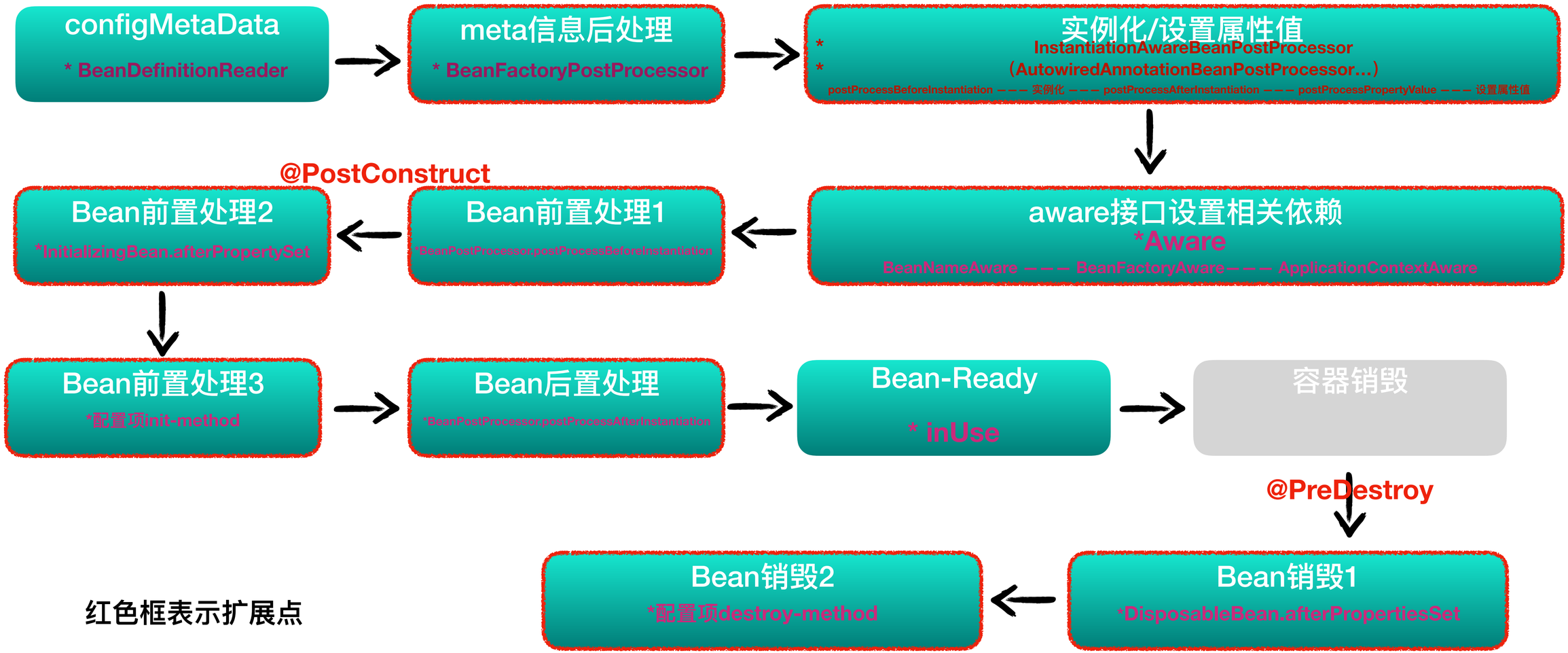
Stateful

Bean(3)-lifecycle

镇楼！！！



Bean(3)-lifecycle



红色框表示扩展点

预加载 ————> 实例化————>初始化————>使用————>销毁

Bean(3)-lifecycle

扩展点：BeanFactoryPostProcessor接口

应用场景

允许在容器实例化对象之前，对注册到容器中的BeanDefinition所保存的信息作相应修改；配合Order接口可以定义不同的BeanFactoryPostProcessor的执行顺序。

系统实现

- PropertyPlaceholderConfiguer
- PropertyOverrideConfiguer
- CustomEditorConfigurer

将配置
中的占位符替换为实际值
${jdbc.url}

重写bean默认值/转换值
max_conn重写为20/解密字符串替换原值

借助PropertyEditor
完成String对象到具体对象的转换，可注册自定义PropertyEditor

Bean(3)-lifecycle

扩展点：Aware接口

应用场景

实现该接口的类可以取得Spring容器中的相应资源

应用层代码将会与spring耦合到一起，所以在应用层不推荐使用

系统实现

- BeanNameAware
- BeanClassLoaderAware
- BeanFactoryAware
- ApplicationContextAware
- ResoureLoaderAware
- MessageSourceAware
- ApplicationEventPublisherAware

- 获取该Bean在ioc容器中名字
- 获取加载该Bean的ClassLoader
- 获取该bean所在的BeanFactory
- 获取该bean所在的ApplicationContext
- 获取resourceLoader使用资源
- 获取国际化支持
- 获取上下文事件发布器

Bean(3)-lifecycle

扩展点：BeanPostProcessor接口

应用场景

允许用户覆盖容器默认或者定义自己的初始化逻辑/依赖解析逻辑的等一系列行为。通过实现order接口，可定义多个processor。

系统实现

ApplicationContextAwareProcessor

AutowiredAnnotationBeanPostProcessor

一系列Aware接口的实现

解析注解信息（@Autowired@Resource@Value等在spring内部就是通过BeanPostProcessor实现的）

Bean(3)-lifecycle

扩展点：InitializaBean/init-method/Desposablebean/destry-method

Init

设置对象属性，调用BeanPostProcessor之后，在某些业务场景下（不符合业务逻辑），该bean不一定处于可以直接使用的状态，实现了InitializingBean接口，将会完成一些初始化工作（调用afterPropertySet方法）。Spring内部广泛使用这种方式，业务系统中则提供init-method的配置项，避免与spring直接耦合。

Destroy

同Init形成对比，提供了自定义销毁逻辑的扩展点。典型的应用场景是数据库连接池的使用，在系统退出后，连接池应该关闭，释放资源。

Bean(4)-annotation

@Required
应用于setter-method
标示的属性必须在配置期间完成填充
建议用@Autowired的requried属性替代

@Autowired
应用于setter-method/method(args)/field/collection等
默认行为是配置期间完成填充，可以通过@Autowired(required=false)更改

@Primary
应用于bean
by-type会产生多个候选bean
标示首选bean

@lazy

@PostConstruct
@PreDestroy
应用于method
类比于init-method/destroy-method
InitializingBean/Desposablebean

@Resoure
应用于field/setter-method
by-name会产生多个候选bean
用该注解对候选bean进行筛选，选定最终的注入bean，采用名称匹配

@Qualifier
应用于bean
by-type会产生多个候选bean
用该注解对候选bean进行筛选，选定最终的注入bean，默认是采用名称匹配

@Scope

1：@Autowired是by-type匹配，结合@Qualifier可以实现by-name匹配
2：@Resource是by-name匹配，如果不设定name，才会by-type匹配。
只能应用在field和setter-method上。
因此在constructor和method上采用@Autowired和@Qualifier的搭配。

Bean(4)-annotation

@ComponentScan
自动扫描注册

@ComponentScan(basePackage = “cn.deepclue”,
includeFilters = @Filter(type = FilterType.REGEX, pattern = “.*Stub.*Repository”),
excludeFilters = @Filter(Repository.class))

@Service
结合ComponentScan自动注册
服务层注解

@Repository
结合ComponentScan自动注册
持久层注解

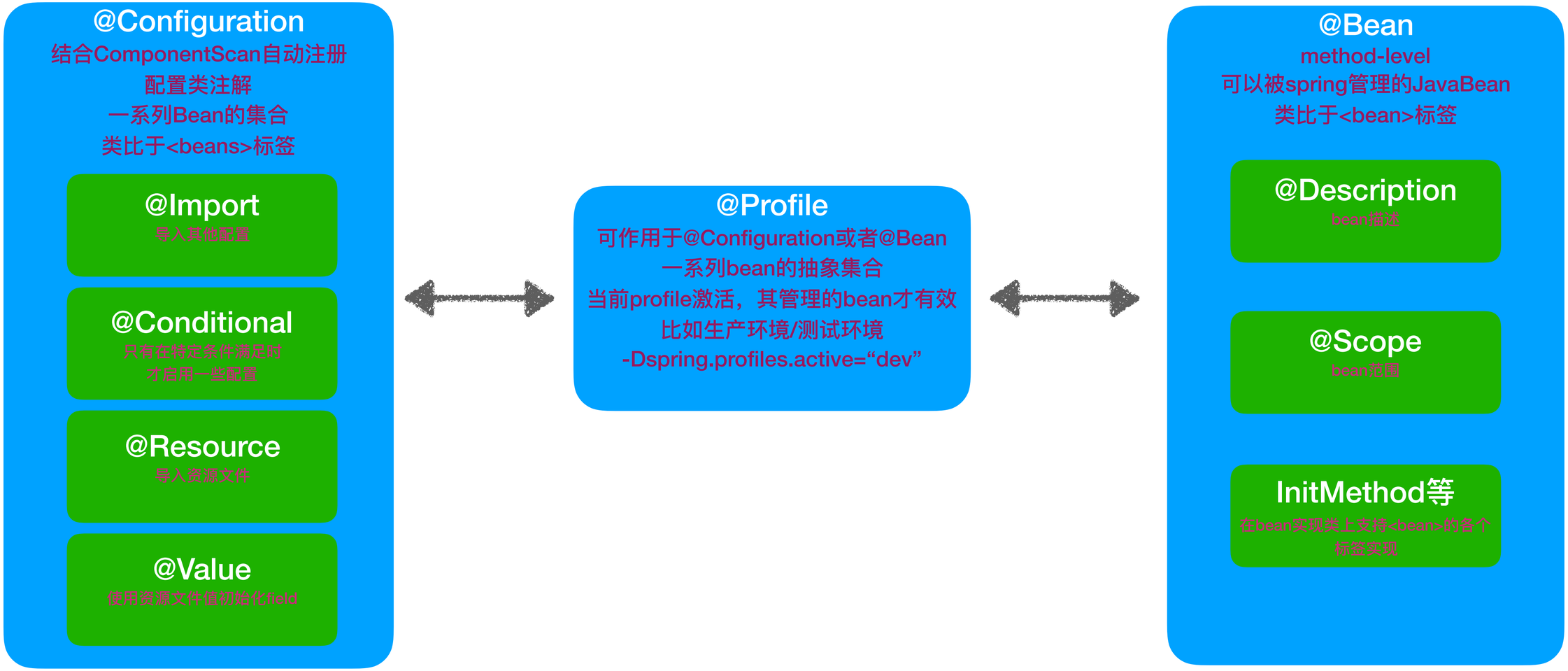
@Controller
结合ComponentScan自动注册
展示层注解

@Component
结合ComponentScan自动注册
通用元注解

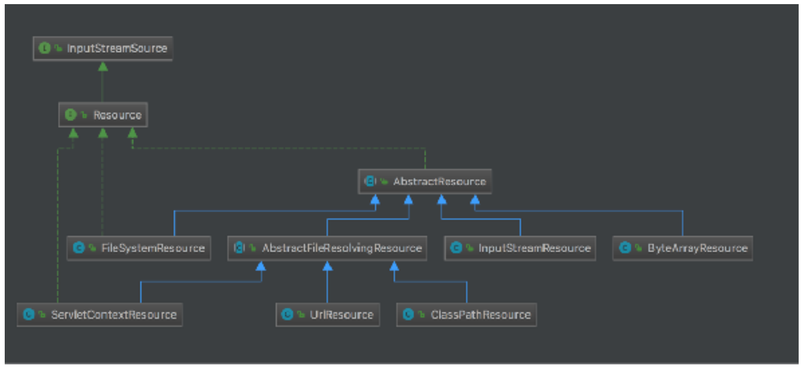
@Configuration
结合ComponentScan自动注册
配置类注解

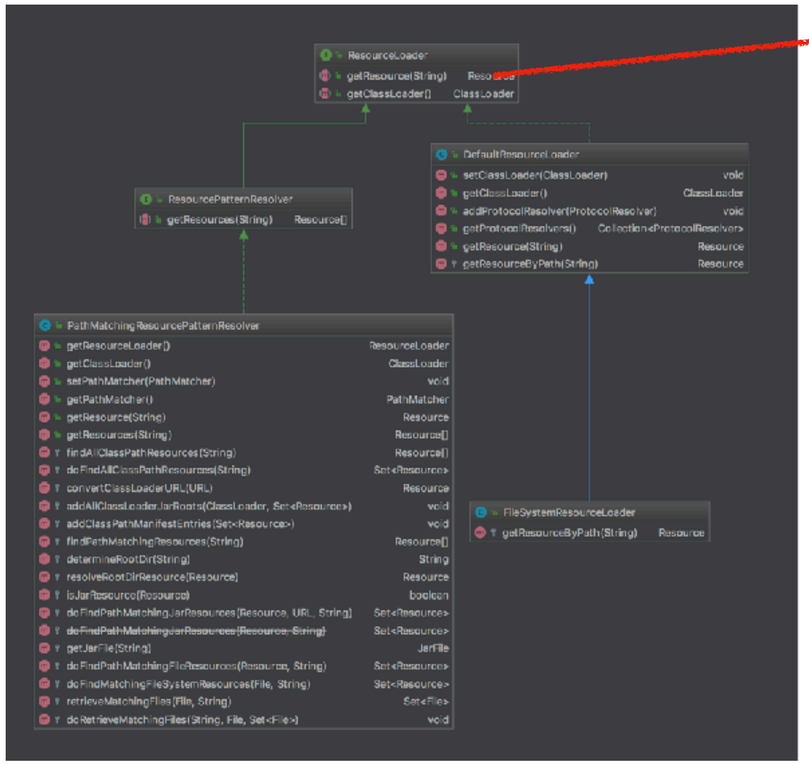
@Bean
method-level

Bean(4)-java config



ApplicationContext(1)-统一资源加载





URLResource
封装URL中资源
StringPrefix = ftp:http:file

ClassPathResource
封装classpath中资源
stringPrefix = classpath:

Resource对资源的统一抽象

Resource getResource(String)

ResourceLoader对资源进行定位

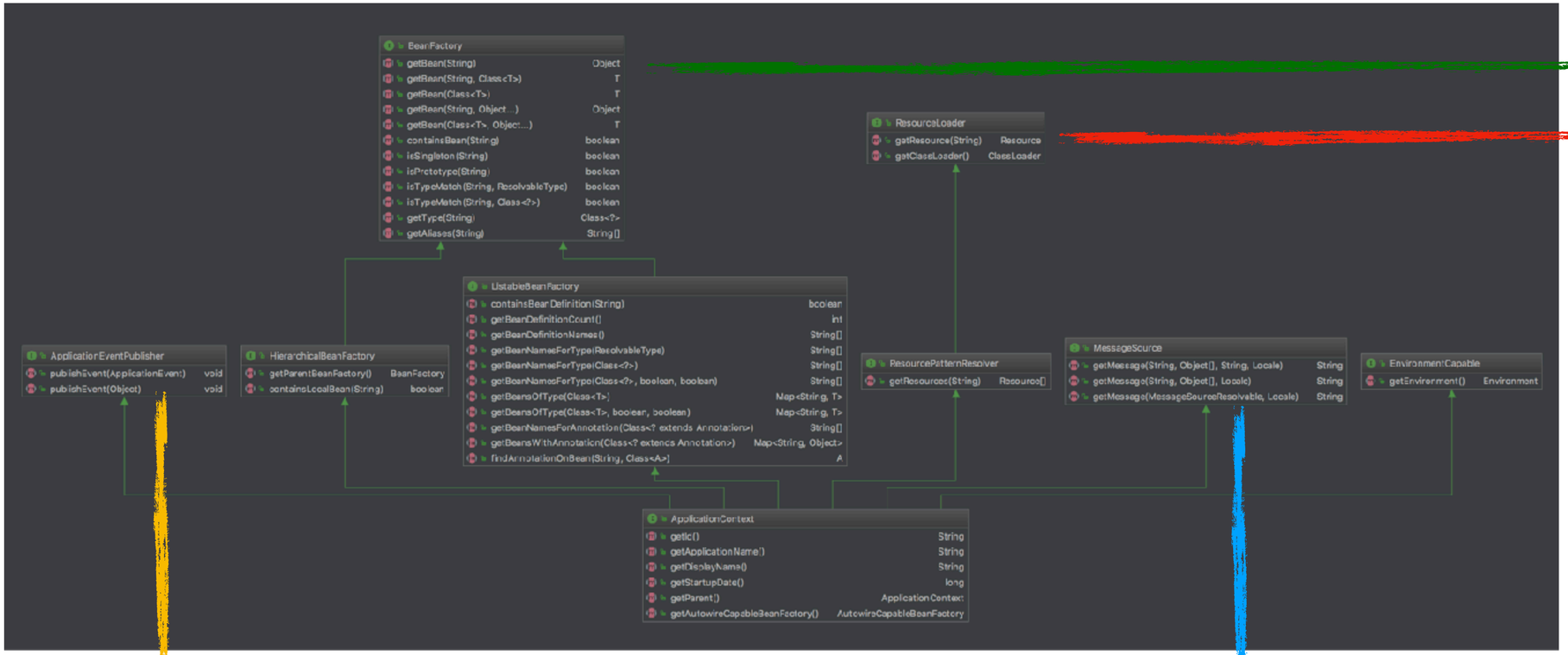
FileSystemResourceLoader
从文件系统获取资源

DefaultResourceLoader
默认的资源获取器

PathMatchingResourcePatternResolver
对ResourceLoader接口的扩展实现类，可以根据模式返回多个ResoureLoader

Resource和ResourceLoader都是bean，其他bean可以根据需要进行注入

ApplicationContext(1)-统一资源加载



Role1：BeanFactory

Role2：ResourceLoader

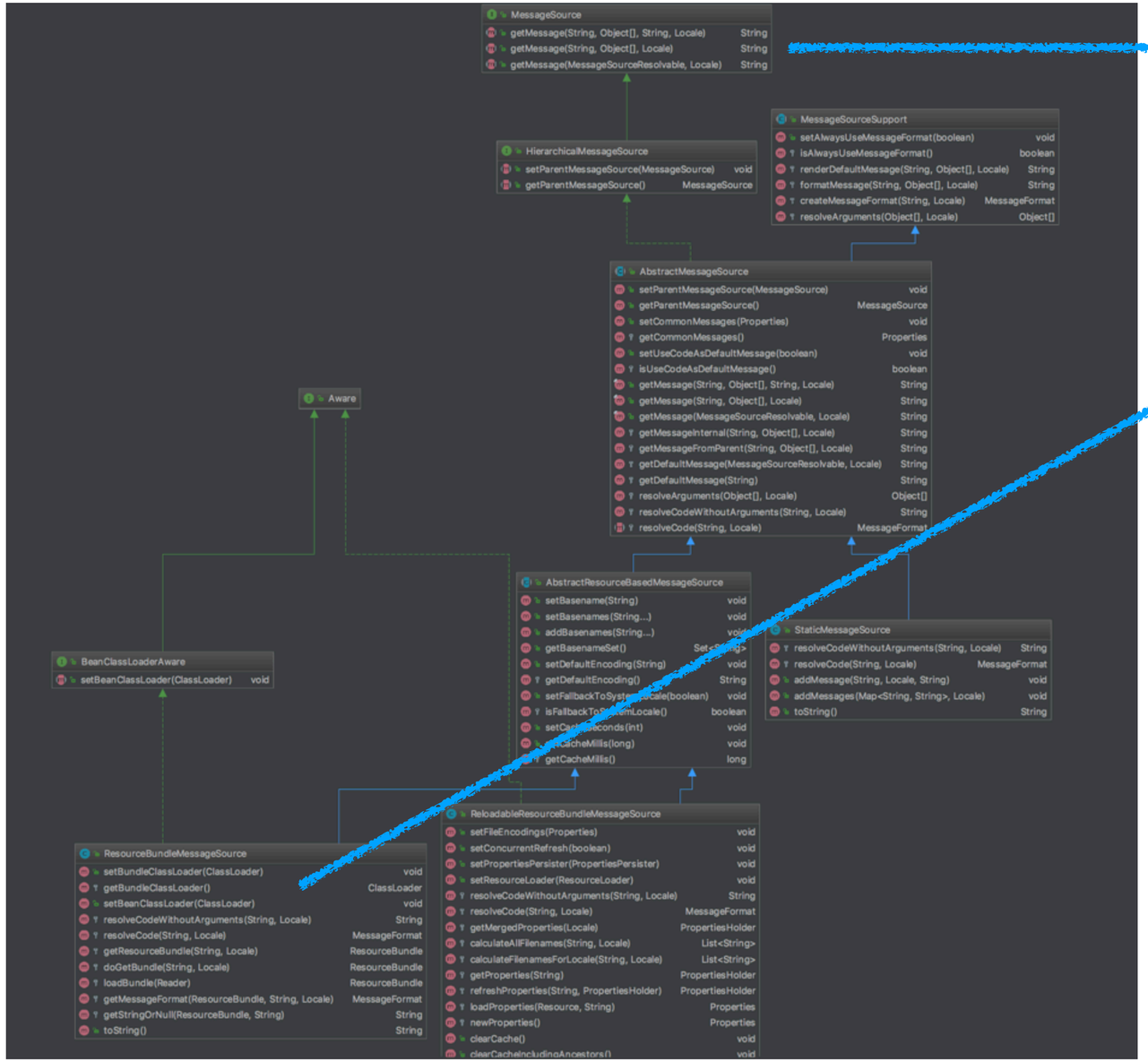
ApplicationContext本身就是ResourceLoader，
其本身定义的PropertyEditor来完成对Resource类型的转换，
不需要自己定义

```
ApplicationContext ctx = ...
ctx.getResource(...)
```

Role4：ApplicationEventPublisher

Role3：MessageSource

ApplicationContext(2)-国际化支持



MessageSource
国际化信息解析

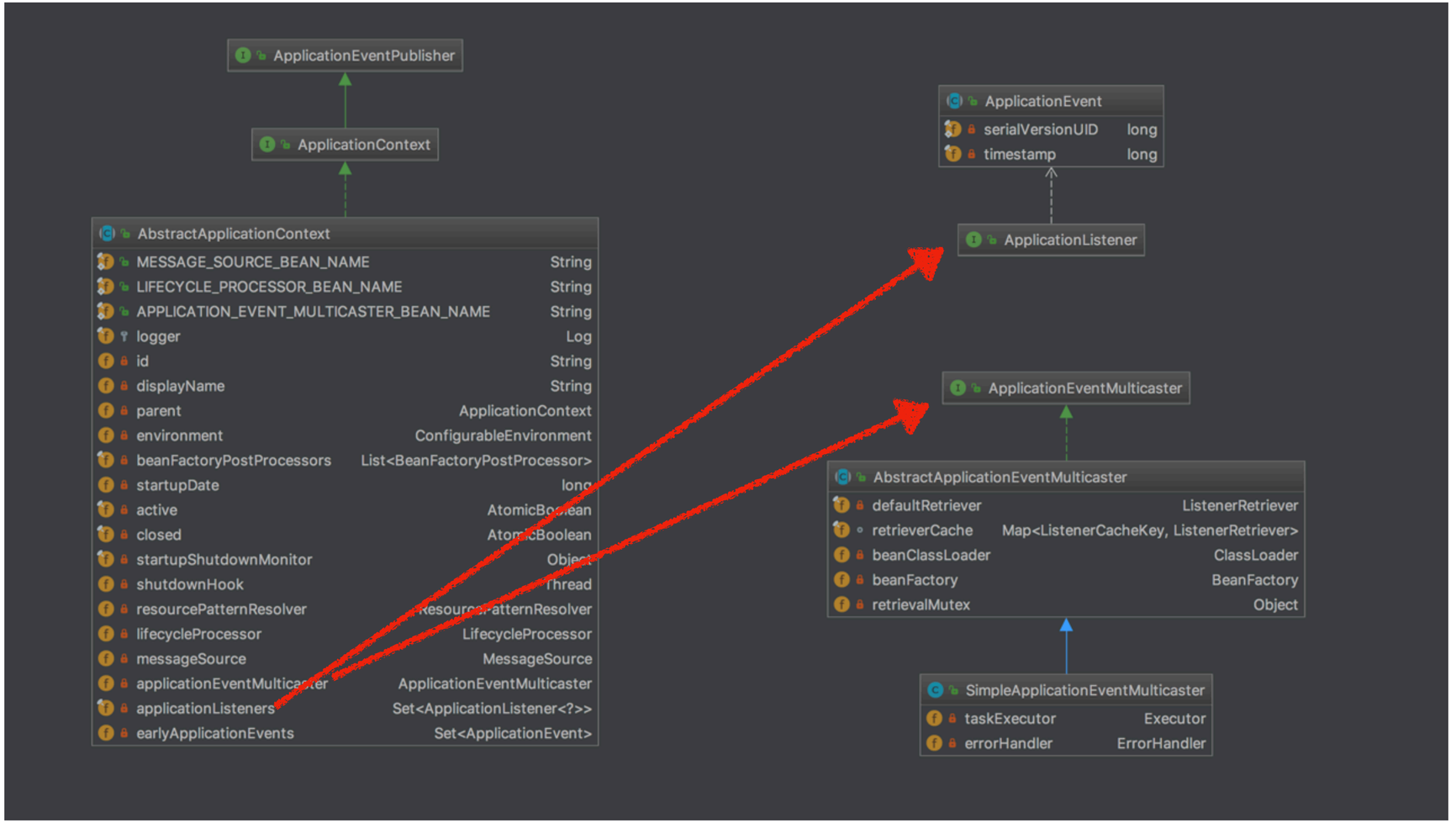
ResourceBundleMessageSource
依托于jdk-resourceBundle的messageSource实现
（最佳实践）

Locale设置国家/地区代码：Locale_CHINA/zh_CN
ResourceBundle保存特定于某个Locale的信息
各个Locale对应一个统一的baseName，比如
message_zh_CH.properties
message_en_US.properties

MessageSource也可以独立注入其他bean。
比如一个公用的validator需要国际化支持不同的提示信息，
则可以将ResourceBundleMessageSource注入，
以获取国际化信息。

Applicationcontext本身实现了messageSource接口，在启动过程中，会识别容器中实现了MessageSourceAware接口的bean定义，并将自身作为MessageSource实例注入到相关bean。因此业务如果某个模块需要国际化支持，最简单的方式就是是实现MessageSourceAware接口，然后注册到ApplicationContext容器，不过这样对ApplicationContext形成了依赖，显得Spring容器有一定的侵入性

ApplicationContext(3)-事件发布器



ApplicationEvent
事件的抽象

ApplicationListener
监听事件，对事件做出响应
(onApplicationEvent)

ApplicationEventMulticaster
发布事件并通知监听者

Application本身就是事件发布者，其发布者的角色委托给了ApplicationEventMulticaster

实现ApplicationEvent接口来自定义事件
实现ApplicationListener来定义监听器并定义事件处理逻辑
实现ApplicationEventPublisherAware接口来定义事件发布者

思考

innerBean的生命周期

如何解决循环依赖问题

工厂方法初始化的优势

跨scope注入-aopproxy/method inject

xml vs annotation vs java-config

@PostConstruct 与init-method 同时存在时的执行顺序

@Bean 方法设置为static有什么作用

@Configuration中注册@Bean vs @Component中注册@Bean

BeanFactoryPostProcessor跟BeanPostProcessor的区别

自定义的BeanPostProcessor/BeanFactoryPostProcessor内部能否使用@Autowired等注解

用xml和annotation同时实现了两个bean(id/name完全一样)，哪一个生效？都生效?