## spring

轻量，容器，集成功能，IOC耦合，AOP切面，

### 自动装配

springboot在启动时通过springAplication.run()，首先构造springAplication对象，构造的时候会加载spring.factories配置文件（初始化器和监听器完全限定名）， 调用run方法有一个刷新容器的过程，过程中会去解析配置文件里配置类 里面会有一些条件注解，会根据引入的jar包、自动注入一些bean，这就是自动装配的

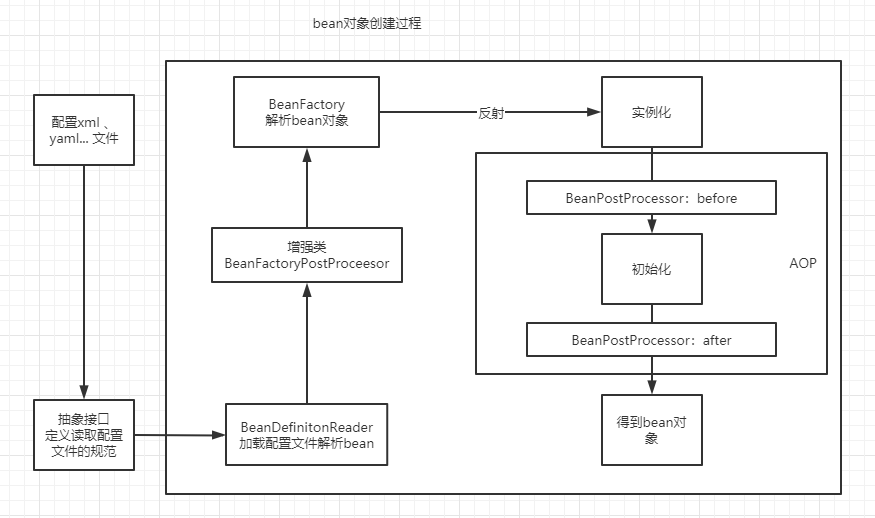
### bean创建过程

实例化：在堆中开辟一块空间，全是默认值

初始化：给属性完成赋值操作

初始化1：填充属性，赋值

初始化2：调用具体的初始化方法



**bean生命周期**

### spring循环依赖

创建一个单例对象的时候需要另外一个单例对象

构造器循环依赖没办法解决

属性循环依赖解决方法：三级缓存

一级缓存：singletonObjects

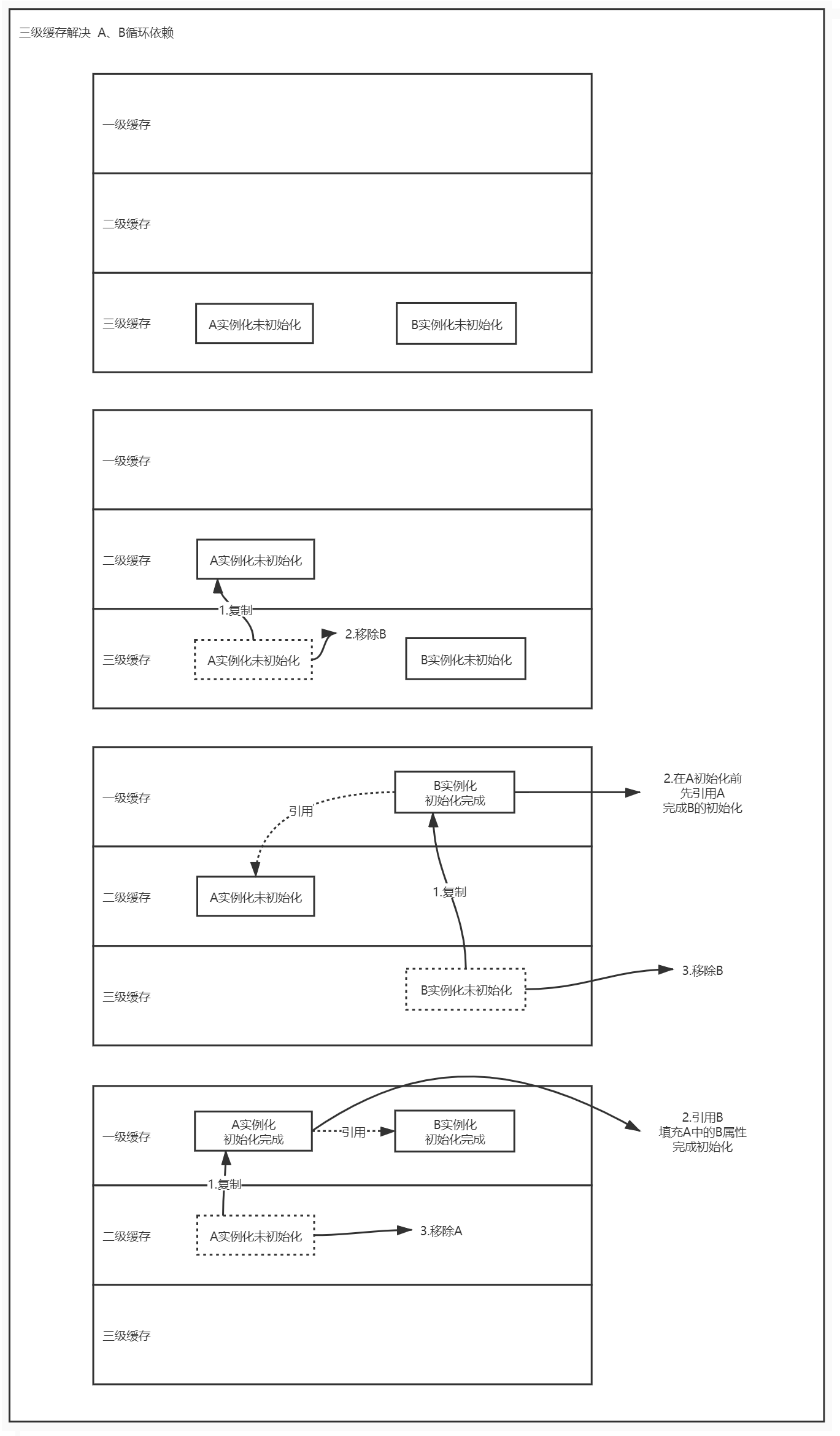
二级缓存：earlySingletonObjects

三级缓存：singletonFactories

*/\*\* Cache of singleton objects: bean name to bean instance.* 一级缓存 *\*/*private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);  
  
*/\*\* Cache of singleton factories: bean name to ObjectFactory.* 三级缓存 *\*/*private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);  
  
*/\*\* Cache of early singleton objects: bean name to bean instance.* 二级缓存 *\*/*private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

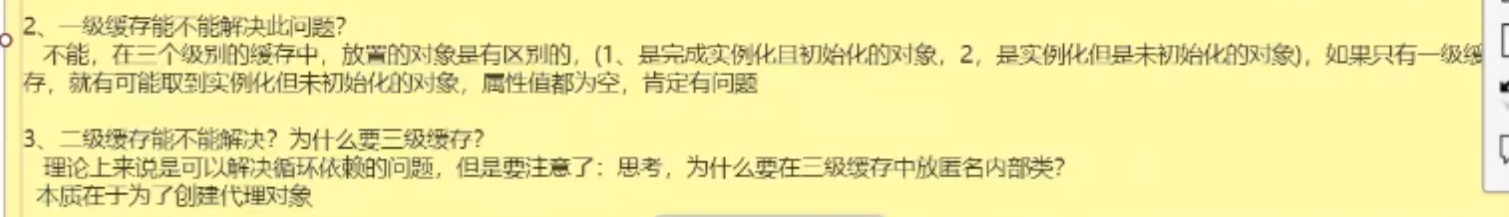
假设AB相互循环（流程图）**流程图可能还存在问题**





1. 为什么使用构造器注入属性的时候不能解决循环依赖问题？

原因在于三级缓存是实例化和初始化分开来操作，将提前实例化好的对象提前暴靠出去，供别人调用，而使用构造器的时候，必须要调用构造方法了，没有构造方法无法完成对象的实例化操作，也就无法创建对象，那么永远会陷入到死循环中。



1. beanFactory和FactoryBean区别

都是用来生产对象的，beanFactory用于通过工厂批量生产有固定模板的相同对象，factoryBean用于生产某一个特殊独立，具有复杂属性的一些对象

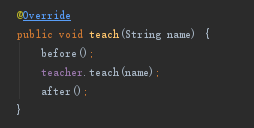
3. ApplicationContext和BeanFactory区别

### springIOC

依赖注入，控制反转，是spring的核心概念之一，给我们提供了一个IOC容器通过xml配置或者注解扫描帮助我们创建管理对象

### springAOP

静态代理：拿一个代理类写一个代理方法调用业务方法，在业务方法前后分别执行需要代码



动态代理：在静态代理的基础上加泛型设计出对所有对象都能用的代理类

1. springAOP静态代理:

定义切点

@Pointcut("@annotation(com.xgn.nz.smartcampus.ui.common.log.SysLog)")

定义切面

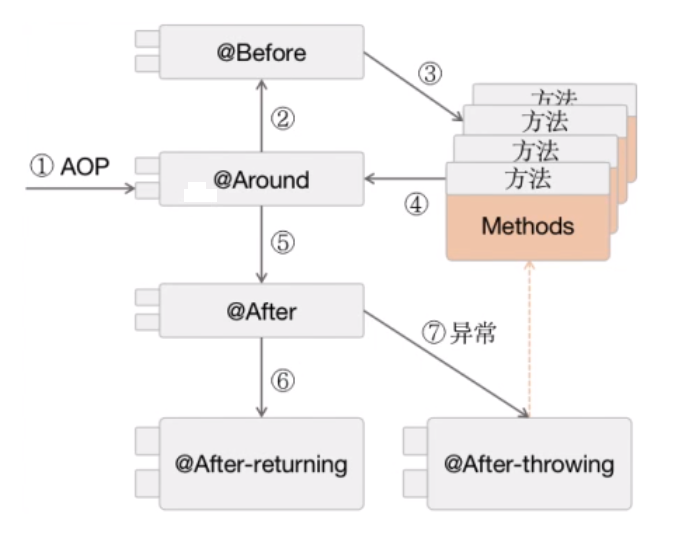
//切面 配置通知  
@AfterReturning(pointcut = "logPoinCut()", returning = "returnValue")  
public void saveSysLog(JoinPoint joinPoint, Object returnValue) {

2.jdk动态代理： invoke

*/\*\*  
 \* JDK动态代理  
 \* 实现InvocationHandler接口  
 \*/*public class DyProxy<T> implements InvocationHandler {  
 private T target;  
 public DyProxy(T target) {  
 this.target = target;  
 }  
 public T getProxy() {  
 return (T) Proxy.*newProxyInstance*(  
 target.getClass().getClassLoader(), target.getClass().getInterfaces(), this  
 );  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 before();  
 Object rs = method.invoke(this.target, args);//业务方法  
 after();  
 return null;  
 }  
 void before() { System.*out*.println("before.....动态代理......."); }  
 void after() { System.*out*.println("after......动态代理......"); }  
 public static void main(String[] args) {  
 ITeacher teacher = new DyProxy<ITeacher>(new TeacherImpl()).getProxy();  
 teacher.teach("动态代理"); }}

2.cglib

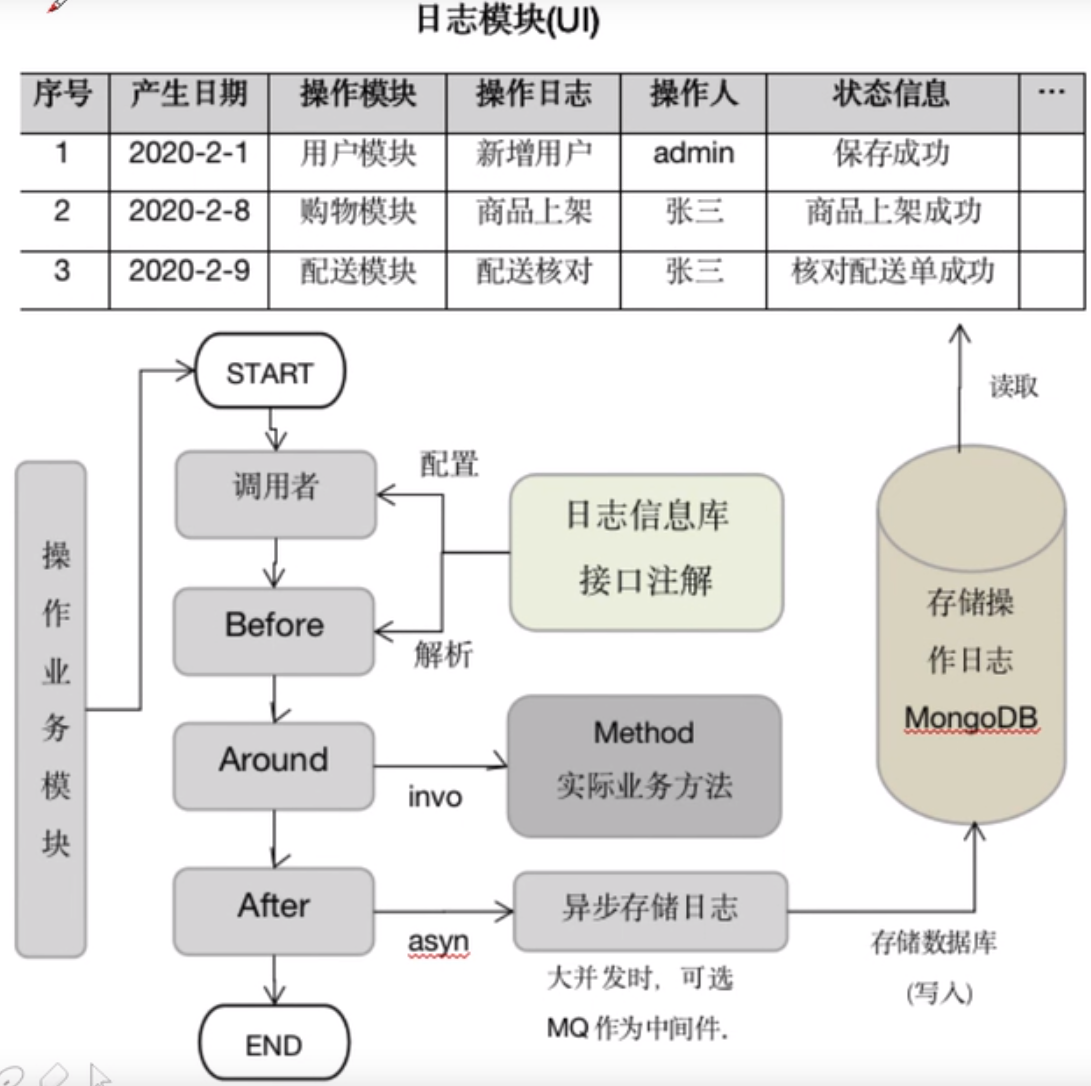
public class CGLibProxy<T> implements MethodInterceptor {  
 private T target;  
 public CGLibProxy(T target) { this.target = target; }  
 public T getProxy(){  
 return (T)Enhancer.*create*(this.target.getClass(),this);  
 }  
 @Override  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 before();  
 Object rs = methodProxy.invokeSuper(o, objects);  
 after();  
 return rs;  
 }  
 void before() {  
 System.*out*.println("before.....CGLibProxy.......");  
 }  
 void after() {  
 System.*out*.println("after......CGLibProxy......");  
 }  
 public static void main(String[] args) {  
 ProductImpl product = new CGLibProxy<ProductImpl>(new ProductImpl()).getProxy();  
 product.show("CGLib动态代理");  
 }  
}



定义切点

@Around 中间可以手动调用methods

案例：



### springBoot

#### 简述：

将SpringMvc基本大部分的配置改为注解和约定的方式，通过这种方式简化了原有的配置，要再配置的更细则通过编写Java配置类，或通过properties或者.yam配置文件的方式去进行更细的配置

#### 自动装配

springboot 自动装配

在启动时加载：spring.factories里面有自动配置类的完全限定名，但是不一定都加载，根据pom导入了对应的start启动器，才会加载对应启动器进行自动装配。

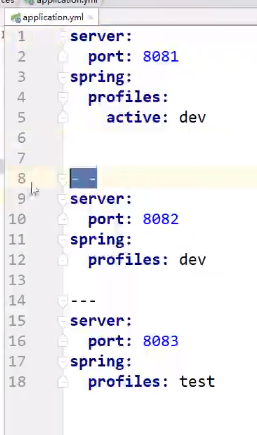
#### 启动流程

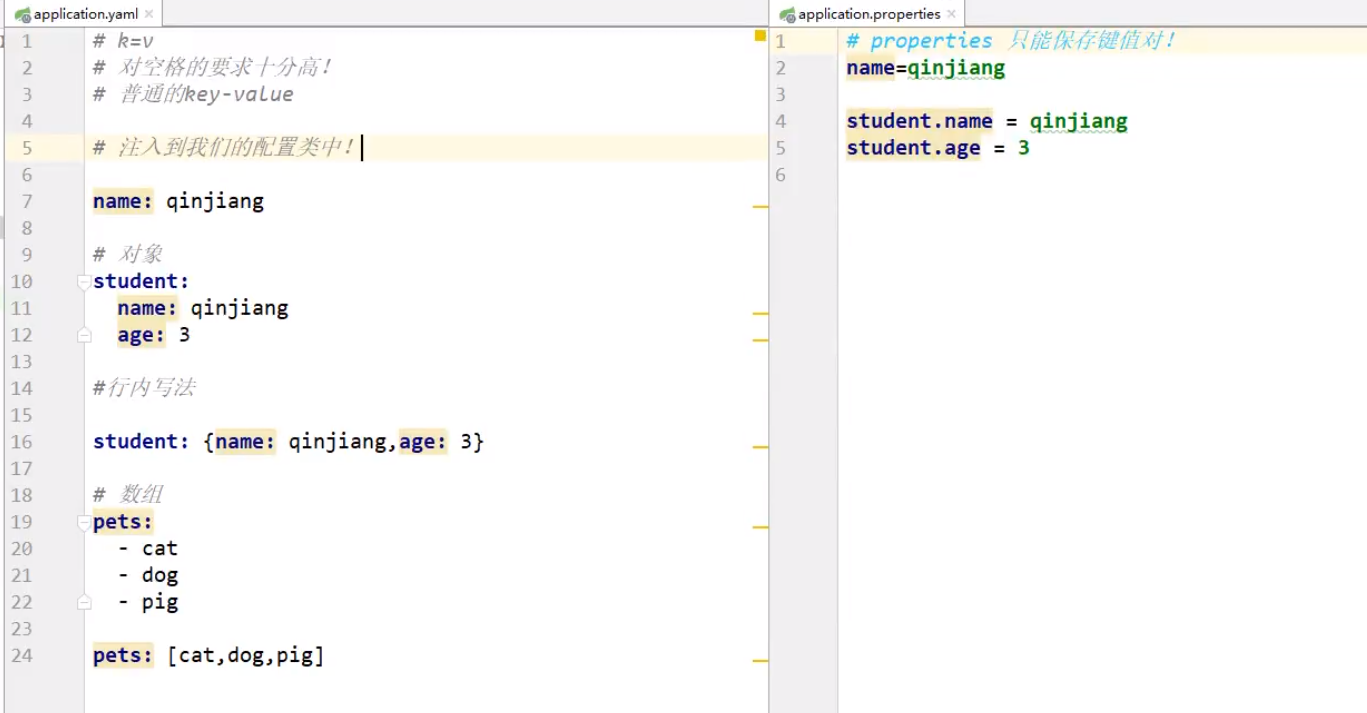
#### yaml配置

配置文件可以放在（优先级按照以下排序）



多版本配置，激活





#### JSR-303校验

@Validated //数据校验  
public class Person {

@Email(message = "是不是邮箱自己没数吗") /\*JSR303校验\*/  
private String email;

}



#### 源码解析

@SpringBootApplication  
public class DemoApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(DemoApplication.class, args);  
 }  
}

@SpringBootApplication

@SpringBootConfiguration：springboot的配置

@Configuration： spring配置类

@Component：说明这也是一个spring的组件

@EnableAutoConfiguration : 自动配置包

1SpringApplication.*run*(SpringAopApplication.class, args);

↓

*/\*\*  
 \* Static helper that can be used to run a {****@link*** *SpringApplication} from the  
 \* specified source using default settings.  
 \** ***@param*** *primarySource the primary source to load  
 \** ***@param*** *args the application arguments (usually passed from a Java main method)  
 \** ***@return*** *the running {****@link*** *ApplicationContext}  
 \*/*public static ConfigurableApplicationContext run(Class<?> primarySource, String... args) {  
 return *run*(new Class<?>[] { primarySource }, args);  
}

↓

*/\*\*  
 \* Static helper that can be used to run a {****@link*** *SpringApplication} from the  
 \* specified sources using default settings and user supplied arguments.  
 \** ***@param*** *primarySources the primary sources to load  
 \** ***@param*** *args the application arguments (usually passed from a Java main method)  
 \** ***@return*** *the running {****@link*** *ApplicationContext}  
 \*/*public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {  
 return new SpringApplication(primarySources).run(args);  
}

↓

*/\*\*  
 \* Create a new {****@link*** *SpringApplication} instance. The application context will load  
 \* beans from the specified primary sources (see {****@link*** *SpringApplication class-level}  
 \* documentation for details. The instance can be customized before calling  
 \* {****@link*** *#run(String...)}.  
 \** ***@param*** *primarySources the primary bean sources  
 \** ***@see*** *#run(Class, String[])  
 \** ***@see*** *#SpringApplication(ResourceLoader, Class...)  
 \** ***@see*** *#setSources(Set)  
 \*/*public SpringApplication(Class<?>... primarySources) {  
 this(null, primarySources);  
}

↓

*/\*\*  
 \* Create a new {****@link*** *SpringApplication} instance. The application context will load  
 \* beans from the specified primary sources (see {****@link*** *SpringApplication class-level}  
 \* documentation for details. The instance can be customized before calling  
 \* {****@link*** *#run(String...)}.  
 \** ***@param*** *resourceLoader the resource loader to use  
 \** ***@param*** *primarySources the primary bean sources  
 \** ***@see*** *#run(Class, String[])  
 \** ***@see*** *#setSources(Set)  
 \*/*@SuppressWarnings({ "unchecked", "rawtypes" })  
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {  
 this.resourceLoader = resourceLoader;  
 Assert.*notNull*(primarySources, "PrimarySources must not be null");  
 this.primarySources = new LinkedHashSet<>(Arrays.*asList*(primarySources));// 启动类放到集合中  
 this.webApplicationType = WebApplicationType.*deduceFromClasspath*();

//从spring.factories文件中取出初始化器和监听器的完全限定名（包名+类名）放入集合

//初始化器7个，监听器11个  
 setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));//设置初始化  
 setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));//设置监听器

this.mainApplicationClass = deduceMainApplicationClass();//推断当前应用程序类  
}

SpringApplication对象创建完成

----------------------------------------------

run方法

public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {  
 return new (primarySources).run(args);  
}

↓

*/\*\*  
 \* Run the Spring application, creating and refreshing a new  
 \* {****@link*** *ApplicationContext}.  
 \** ***@param*** *args the application arguments (usually passed from a Java main method)  
 \** ***@return*** *a running {****@link*** *ApplicationContext}  
 \*/*public ConfigurableApplicationContext run(String... args) {  
 StopWatch stopWatch = new StopWatch();//启动计时  
 stopWatch.start();//启动计时开始  
 ConfigurableApplicationContext context = null;  
 Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();//异常报告  
 configureHeadlessProperty();//设置系统属性  
 SpringApplicationRunListeners listeners = getRunListeners(args);//获取注册监听器  
 listeners.starting();//启动监听器  
 try {  
 ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);//参数放入集合  
 ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);//加载环境数据  
 configureIgnoreBeanInfo(environment);//忽略一部分bean  
 Banner printedBanner = printBanner(environment);//打印spring图案  
 context = createApplicationContext();//创建上下文  
 exceptionReporters = getSpringFactoriesInstances(SpringBootExceptionReporter.class,  
 new Class[] { ConfigurableApplicationContext.class }, context);//获取异常报告器

//准备工作完成  
 prepareContext(context, environment, listeners, applicationArguments, printedBanner);//加载启动类进上下文  
 refreshContext(context);//刷新上下文  
 afterRefresh(context, applicationArguments);  
 stopWatch.stop();//启动计时结束  
 if (this.logStartupInfo) {  
 new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);  
 }  
 listeners.started(context);  
 callRunners(context, applicationArguments);  
 }  
 catch (Throwable ex) {  
 handleRunFailure(context, ex, exceptionReporters, listeners);  
 throw new IllegalStateException(ex);  
 }  
  
 try {  
 listeners.running(context);  
 }  
 catch (Throwable ex) {  
 handleRunFailure(context, ex, exceptionReporters, null);  
 throw new IllegalStateException(ex);  
 }  
 return context;  
}

### springMVC

