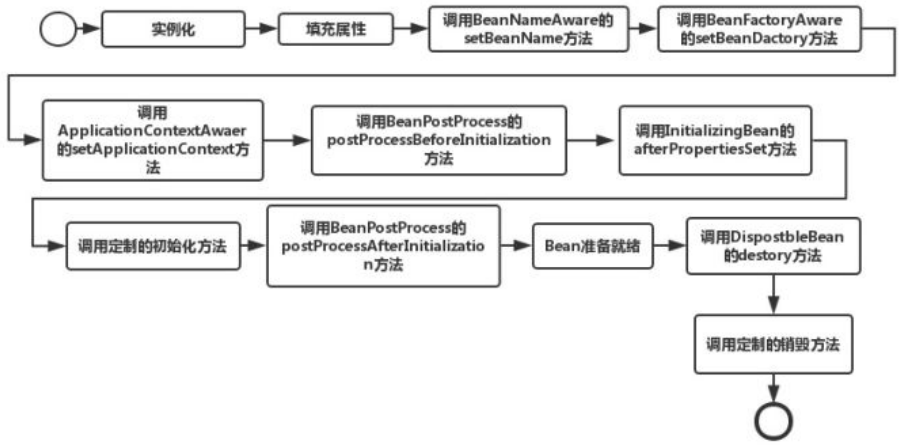
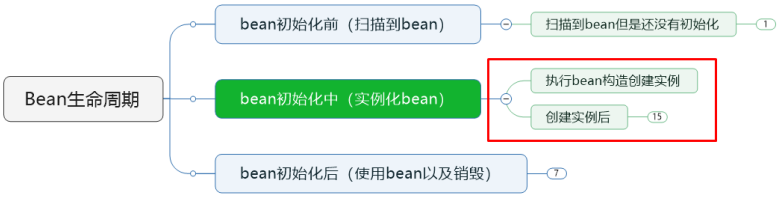
1. Spring中Bean的生命周期。





Instance实例化-》设置属性值-》调用BeanNameAware的setBeanName方法-》调用BeanPostProsessor的预初始化方法-》调用InitializationBean的afterPropertiesSet()的方法-》调用定制的初始化方法callCustom的init-method-》调用BeanPostProsessor的后初始化方法-》Bean可以使用了 -》 容器关闭-》 调用DisposableBean的destroy方法-》调用定制的销毁方法CallCustom的destroy-method。

1、Spring对Bean进行实例化（相当于程序中的new Xx()）

2、Spring将值和Bean的引用注入进Bean对应的属性中

3、如果Bean实现了BeanNameAware接口，Spring将Bean的ID传递给setBeanName()方法（实现BeanNameAware清主要是为了通过Bean的引用来获得Bean的ID，一般业务中是很少有用到Bean的ID的）

4、如果Bean实现了BeanFactoryAware接口，Spring将调用setBeanDactory(BeanFactory bf)方法并把BeanFactory容器实例作为参数传入。（实现BeanFactoryAware 主要目的是为了获取Spring容器，如Bean通过Spring容器发布事件等）

5、如果Bean实现了ApplicationContextAwaer接口，Spring容器将调用setApplicationContext(ApplicationContext ctx)方法，把y应用上下文作为参数传入.(作用与BeanFactory类似都是为了获取Spring容器，不同的是Spring容器在调用setApplicationContext方法时会把它自己作为setApplicationContext 的参数传入，而Spring容器在调用setBeanDactory前需要程序员自己指定（注入）setBeanDactory里的参数BeanFactory)

7、如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessBeforeInitialization（预初始化）方法（作用是在Bean实例创建成功后对进行增强处理，如对Bean进行修改，增加某个功能）7.如果Bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法，作用与在配置文件中对Bean使用init-method声明初始化的作用一样，都是在Bean的全部属性设置成功后执行的初始化方法。

8、如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessAfterInitialization（后初始化）方法（作用与7的一样，只不过7是在Bean初始化前执行的，而这个是在Bean初始化后执行的，时机不同 ）

9、经过以上的工作后，Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁

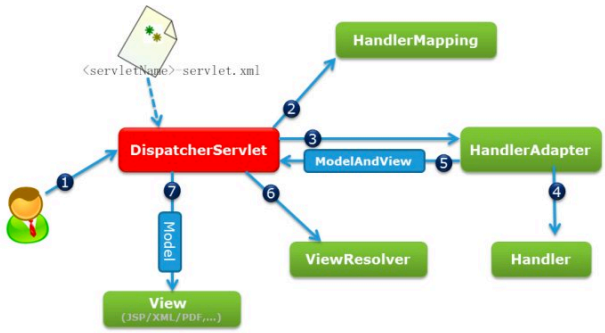
10、如果Bean实现了DispostbleBean接口，Spring将调用它的destory方法，作用与在配置文件中对Bean使用destory-method属性的作用一样，都是在Bean实例销毁前执行的方法。

2. SpringBoot项目启动时执行特定的方法：

我们可以通过实现ApplicationRunner和CommandLineRunner，来实现，他们都是在SpringApplication 执行之后开始执

行的。

3. SpringMVC处理请求的流程。



1、请求解析和匹配DipatcherServlet路径：客户端发出一个http请求给web服务器，web服务器对http请求进行解析，如果匹配DispatcherServlet的请求映射路径（在web.xml中指定），web容器将请求转交给DispatcherServlet.

2、匹配处理器Handler：DipatcherServlet接收到这个请求之后将根据请求的信息（包括URL、Http方法、请求报文头和请求参数Cookie等）以及HandlerMapping的配置找到处理请求的处理器（Handler）。

3-4、处理器进行处理：DispatcherServlet根据HandlerMapping找到对应的Handler,将处理权交给Handler（Handler将具体的处理进行封装），再由具体的HandlerAdapter对Handler进行具体的调用。

5、处理器返回逻辑视图ModelAndView对象给DispatcherServlet：Handler对数据处理完成以后将返回一个ModelAndView()对象给DispatcherServlet。

6、Dispatcher通过ViewResolver将逻辑视图转化为正式视图view：Handler返回的ModelAndView()只是一个逻辑视图并不是一个正式的视图，DispatcherSevlet通过ViewResolver将逻辑视图转化为真正的视图View。

7、Dispatcher通过model解析出ModelAndView()中的参数进行解析最终展现出完整的view并返回给客户端。

4. Spring AOP解决了什么问题？怎么实现的?

a. https://blog.csdn.net/moreevan/article/details/11977115/

b. 作用：AOP技术，利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

c. 实现：

i. 一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

ii. 二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

d. 使用场景：

1. Authentication 权限
2. Caching 缓存
3. Context passing 内容传递
4. Error handling 错误处理
5. Lazy loading　懒加载
6. Debugging　　调试
7. logging, tracing, profiling and monitoring　记录跟踪　优化　校准
8. Performance optimization　性能优化
9. Persistence　　持久化
10. Resource pooling　资源池
11. Synchronization　同步
12. Transactions 事务

e. 几个概念：

1.切面（Aspect）：对象操作过程中的截面，这可能是AOP中最为关键的术语。切面所要做的事就是专注于各自领域的逻辑实现，这样可以使得开发逻辑更加清晰，更加适合专业的分工合作。由于切面的隔离性，降低了耦合，这样就可以在不同的应用中将各个切面组合使用，从而是代码重用性大大增加。

2.连接点：程序运行过程中的某个阶段点，如某个方法的调用，或者异常的抛出。

3.处理逻辑（Advice）：在某一个连接点所采用的处理逻辑，处理逻辑的调用模式通常有三种：

a.Around 在连接点前后插入预处理过程和后处理过程。

b.Before 仅在连接点钱出入处理过程。

b.Throw 在连接点抛出异常时进行异常处理。

Advice有的译为“通知”，说法不一，我侧重于“处理逻辑”出自于Spring开发指南一文。

4.切点（PointCut）：一系列连接点的集合，它指明处理逻辑将在合适触发。

5. Spring事务的传播属性是怎么回事？它会影响什么？

a. 七个事传播属性：

1. PROPAGATION_REQUIRED -- 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
2. PROPAGATION_SUPPORTS -- 支持当前事务，如果当前没有事务，就以非事务方式执行。
3. PROPAGATION_MANDATORY -- 支持当前事务，如果当前没有事务，就抛出异常。
4. PROPAGATIONREQUIRESNEW -- 新建事务，如果当前存在事务，把当前事务挂起。
5. PROPAGATIONNOTSUPPORTED -- 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
6. PROPAGATION_NEVER -- 以非事务方式执行，如果当前存在事务，则抛出异常。
7. PROPAGATIONNESTED -- 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATIONREQUIRED类似的操作。

b. 五个隔离级别：

1. ISOLATION_DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.
2. 另外四个与JDBC的隔离级别相对应：
3. ISOLATION*READ*UNCOMMITTED 这是事务最低的隔离级别，它充许别外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻读。
4. ISOLATION*READ*COMMITTED 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻读。
5. ISOLATION*REPEATABLE*READ 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免不可重复读。
6. ISOLATION_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻读。

c. 关键词：

1、幻读：事务1读取记录时事务2增加了记录并提交，事务1再次读取时可以看到事务2新增的记录；

2、不可重复读：事务1读取记录时，事务2更新了记录并提交，事务1再次读取时可以看到事务2修改后的记录；

3、脏读：事务1更新了记录，但没有提交，事务2读取了更新后的行，然后事务T1回滚，现在T2读取无效。

6. Spring中BeanFactory和FactoryBean有什么区别?

a. BeanFactory，以Factory结尾，表示它是一个工厂类(接口)，用于管理Bean的一个工厂。在Spring中，BeanFactory是IOC容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

b. FactoryBean以Bean结尾，表示它是一个Bean，不同于普通Bean的是：它是实现了FactoryBean接口的Bean，根据该Bean的ID从BeanFactory中获取的实际上是FactoryBean的getObject()返回的对象，而不是FactoryBean本身，如果要获取FactoryBean对象，请在id前面加一个&符号来获取。

7. Spring框架中IOC的原理是什么?

(1). IoC（Inversion of Control）是指容器控制程序对象之间的关系，而不是传统实现中，由程序代码直接操控。控制权由应用代码中转到了外部容器，控制权的转移是所谓反转。对于Spring而言，就是由Spring来控制对象的生命周期和对象之间的关系；IoC还有另外一个名字——“依赖注入（Dependency Injection）”。从名字上理解，所谓依赖注入，即组件之间的依赖关系由容器在运行期决定，即由容器动态地将某种依赖关系注入到组件之中。

(2). 在Spring的工作方式中，所有的类都会在spring容器中登记，告诉spring这是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。

(3). 在系统运行中，动态的向某个对象提供它所需要的其他对象。

(4). 依赖注入的思想是通过反射机制实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中。总而言之，在传统的对象创建方式中，通常由调用者来创建被调用者的实例，而在Spring中创建被调用者的工作由Spring来完成，然后注入调用者，即所谓的依赖注入or控制反转。 注入方式有两种：依赖注入和设置注入；IoC的优点：降低了组件之间的耦合，降低了业务对象之间替换的复杂性，使之能够灵活的管理对象。

8. spring的依赖注入有哪几种方式

在Spring容器中为一个bean配置依赖注入有四种方式：

1. 使用属性的setter方法注入 这是最常用的方式；
2. 使用构造器注入；

3. 使用Filed注入（用于注解方式）.

4. 静态、实例工厂的方法注入

9. 用Spring如何实现一个切面?

```
@Aspect
@Component
public class LockAspect {
    @Pointcut("@annotation(xx.xx)")
    public void pointcut() {

    }

    @Around("pointcut()")
     public Object arount(ProceedingJoinPoint point) throws Throwable {
       ...
       try {
         return point.proceed();
       } catch (Exception e) {
         throw e;
       } finally {
         ...
       }
     }
}
```

10. Spring如何实现数据库事务?

使用@Transactional注解或在配置文件里面配置

11. Spring加载次序Classloader

1、先构造函数—>然后是bean的set方法注入—>InitializingBean的afterPropertiesSet方法—>init-method方法；

2、InitializingBean接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候会执行该方法。系统则是先调用afterPropertiesSet方法，然后在调用init-method中指定的方法。

3、Spring装配Bean的过程：

1. 实例化；
2. 设置属性值；
3. 如果实现了BeanNameAware接口，调用setBeanName设置Bean的ID或者Name；
4. 如果实现BeanFactoryAware接口，调用setBeanFactory 设置BeanFactory；
5. 如果实现ApplicationContextAware，调用setApplicationContext设置ApplicationContext
6. 调用BeanPostProcessor的预先初始化方法；
7. 调用InitializingBean的afterPropertiesSet()方法；
8. 调用定制init-method方法；
9. 调用BeanPostProcessor的后初始化方法；

4、Spring容器关闭过程：

1. 调用DisposableBean的destroy()；
2. 调用定制的destroy-method方法；

12. 框架的优缺点SpringMVC,Struts2等...

1、Struts2是类级别拦截，参数为类中所有方法共有，一个Action对应一个request上下文，SpringMVC是方法级别拦截，参数为对应方法所有；

2、由于Struts2需要针对每个request进行封装，把request，session等servlet生命周期的变量封装成一个一个Map，供给每个Action使用，并保证线程安全，所以在原则上，是比较耗费内存的。

3、拦截器实现机制上，Struts2有以自己的interceptor机制，SpringMVC用的是独立的AOP方式，这样导致Struts2的配置文件量还是比SpringMVC大。

4、SpringMVC的入口是servlet，而Struts2是filter

5、SpringMVC集成了Ajax，使用非常方便，只需一个注解@ResponseBody就可以实现，然后直接返回响应文本即可，而Struts2拦截器集成了Ajax，在Action中处理时一般必须安装插件或者自己写代码集成进去，使用起来也相对不方便。

6、SpringMVC开发效率和性能高于Struts2。

7、SpringMVC配置少，零配置。

13. IOC控制反转与DI依赖注入：

1、IOC控制反转：是一种将对象交给容器去控制的设计思想，松耦合，方便单元测试，增加功能重用性；

2、DI依赖注入：组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

3、AOP面向切面编程：面向切面编程（AOP）完善spring的依赖注入（DI）。

14. AOP开发：

1. 导入aop依赖包：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>4.3.12.RELEASE</version>
</dependency>
```

2. 创建切面类：

```
/**
 * 日志切面类
 *
 * @author xuan
 * @date 2018/11/1
 */
@Aspect
public class LogAspects {
    /**
     * 公共的切入点表达式
     * 1、本类引用
     * 2、其他的切面引用
     *
     * @author xuan
     * @date 2018/11/1
     */
    @Pointcut(value = "execution(public int com.test.aop.MathCalculator.*(..))")
    public void pointCut(){}

    /**
     * 前置通知
     *
     * 在目标方法之前切入，切入点表达式（指在哪个方法切入）
     * joinPoint参数一定要出现在参数列表第一位，放在后面会报错
     *
     * @author xuan
     * @date 2018/11/1
     */
    @Before(value = "pointCut()")
    public void logStart(JoinPoint joinPoint) {
            String methodName = joinPoint.getSignature().getName();
            Object[] args = joinPoint.getArgs();
            System.out.println(methodName + "运行。。。参数列表是：" + Arrays.toString(args));
    }

    @After(value = "pointCut()")
    public void logEnd() {
            System.out.println("除法结束。。。");
    }
```

```
40
41      @AfterReturning(value = "pointCut()", returning = "result")
42      public void logReturn(JoinPoint joinPoint, Object result) {
43              String methodName = joinPoint.getSignature().getName();
44              System.out.println(methodName + "正常返回。。。计算结果：" + result);
45      }
46
47      @AfterThrowing(value = "pointCut()", throwing = "exception")
48      public void logException(JoinPoint joinPoint, Exception exception){
49              String methodName = joinPoint.getSignature().getName();
50              System.out.println(methodName + "异常，异常信息：" + exception);
51      }
52
53  }
```

> 通知方法类型：
>
> 前置通知（logStart）：在目标方法运行之前运行；
> 后置通知（logEnd）：在目标方法运行结束之后运行；
> 返回通知（logReturn）：在目标方法正常返回之后运行；
> 异常通知（logException）：在目标方法出现异常是运行；
> 环绕通知（动态代理）：手动推荐目标方法运行（joinPoint.procced()）。

3. 将切面类和目标类都加入到容器中，并开启基于注解的aop动态代理

```
1  public class MathCalculator {
2      public int div(int i, int j) {
3              System.out.println("MathCalculator.div.....");
4              return i/j;
5      }
6
7  }
8
9
10 /**
11  * aop
12  *  在程序运行期间动态的将某段代码切入到指定方法指定位置进行运行的编程方式
13  * '@EnableAspectJAutoProxy'：开启基于注解的aop动态代理
14  *
15  * @author xuan
16  * @date 2018/11/1
17  */
18 @Configuration
19 @EnableAspectJAutoProxy
20 public class MainConfigOfAOP {
21
22     /**
23      * 业务逻辑类加入容器中
24      *
25      * @author xuan
26      * @date 2018/11/1
27      */
28     @Bean
29     public MathCalculator calculator() {
```

```
            return new MathCalculator();
    }

    /**
     * 切面类加入容器中
     *
     * @author xuan
     * @date 2018/11/1
     */
    @Bean
    public LogAspects logAspects(){
            return new LogAspects();
    }
}
```

4. 测试：

```
@Test
public void test01() {
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfi
    MathCalculator calculator = applicationContext.getBean(MathCalculator.class);
    calculator.div(1, 0);
}
```

15. 事务配置：

1. @EnableTransactionManagement开启事务配置功能；
2. 容器中配置DataSource、JdbcTemplate、PlatformTransactionManager三个Bean实例；
3. 使用@Transactional注解开启事务。

```
@EnableTransactionManagement
@Configuration
@ComponentScan("com.test.tx")
public class TxConfig {
    @Bean
    public DataSource dataSource() {
            DruidDataSource dataSource = new DruidDataSource();
            dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/xxx?useSSL=false");
            dataSource.setUsername("xxx");
            dataSource.setPassword("xxx");
            dataSource.setDriverClassName("com.mysql.jdbc.Driver");
            return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
            return new JdbcTemplate(dataSource());
    }

    /**
     * 注册事务管理器
     *
     * @author xuan
     * @date 2018/11/3
     */
    @Bean
```

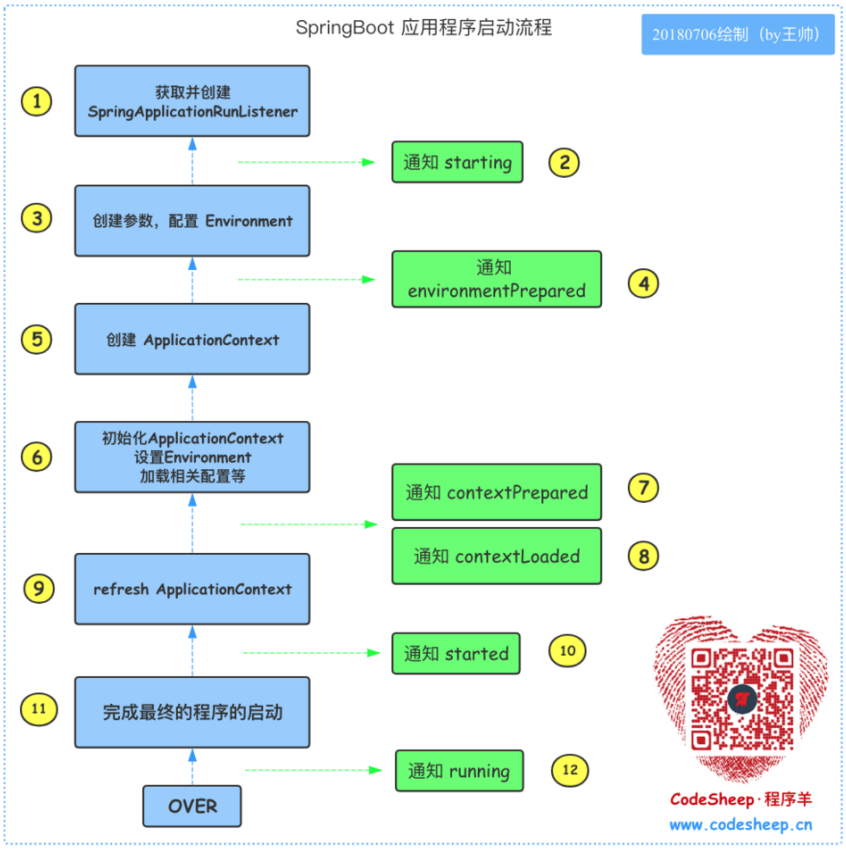
```
    public PlatformTransactionManager transactionManager() {
            return new DataSourceTransactionManager(dataSource());
    }
}

@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    /**
     * 添加用户
     *
     * @author xuan
     * @date 2018/11/3
     */
    @Transactional
    public void insertUser() {
            User user = new User("zhangsan", 27);
            userDao.insert(user);
    }
}
```

16. springboot的启动过程：



a. 通过 SpringFactoriesLoader加载 META-INF/spring.factories文件，获取并创建 SpringApplicationRunListener对象

b. 然后由 SpringApplicationRunListener来发出 starting 消息

c. 创建参数，并配置当前 SpringBoot 应用将要使用的 Environment

d. 完成之后，依然由 SpringApplicationRunListener来发出 environmentPrepared 消息

e. 创建 ApplicationContext

f. 初始化 ApplicationContext，并设置 Environment，加载相关配置等

g. 由 SpringApplicationRunListener来发出 contextPrepared消息，告知SpringBoot 应用使用的 ApplicationContext已准备OK

h. 将各种 beans 装载入 ApplicationContext，继续由 SpringApplicationRunListener来发出 contextLoaded 消息，告知 SpringBoot 应用使用的 ApplicationContext已装填OK

i. refresh ApplicationContext，完成IoC容器可用的最后一步

j. 由 SpringApplicationRunListener来发出 started 消息

k. 完成最终的程序的启动

l. 由 SpringApplicationRunListener来发出 running 消息，告知程序已运行起来了

17. spring事件的实现原理，写出常用的几个事件。

a. 事件机制：Spring中的事件机制是一个观察者模式的实现.观察者模式就是一个目标对象管理所有相依于它的观察者对象,并且在它本身的状态改变时主动发出通知.Spring的事件由ApplicationContext发布。

b. spring默认存在的事件：

1. ContextStartedEvent：ApplicationContext启动后触发的事件
2. ContextStoppedEvent：ApplicationContext停止后触发的事件
3. ContextRefreshedEvent：ApplicationContext初始化或刷新完成后触发的事件
4. ContextClosedEvent：ApplicationContext关闭后触发的事件