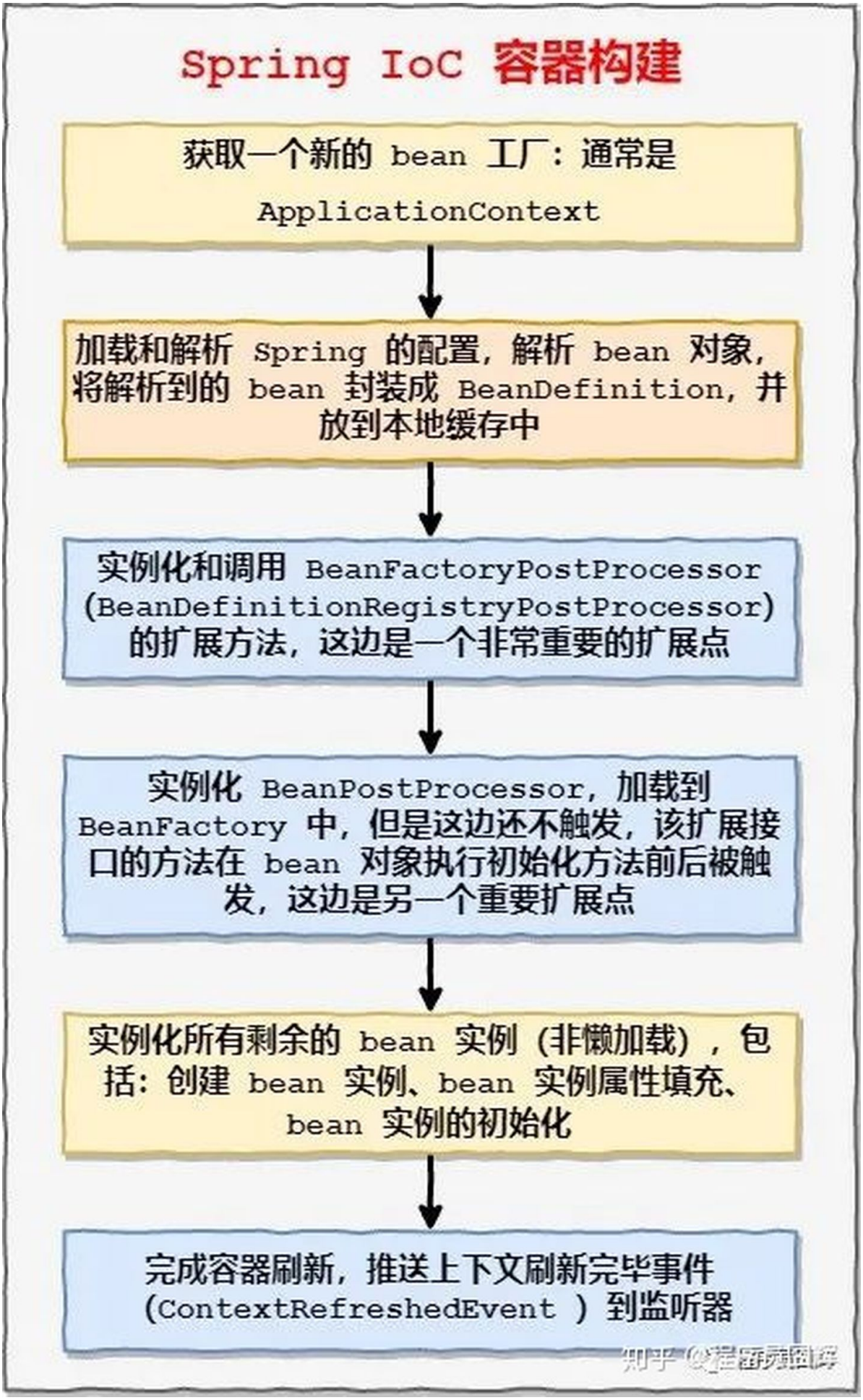
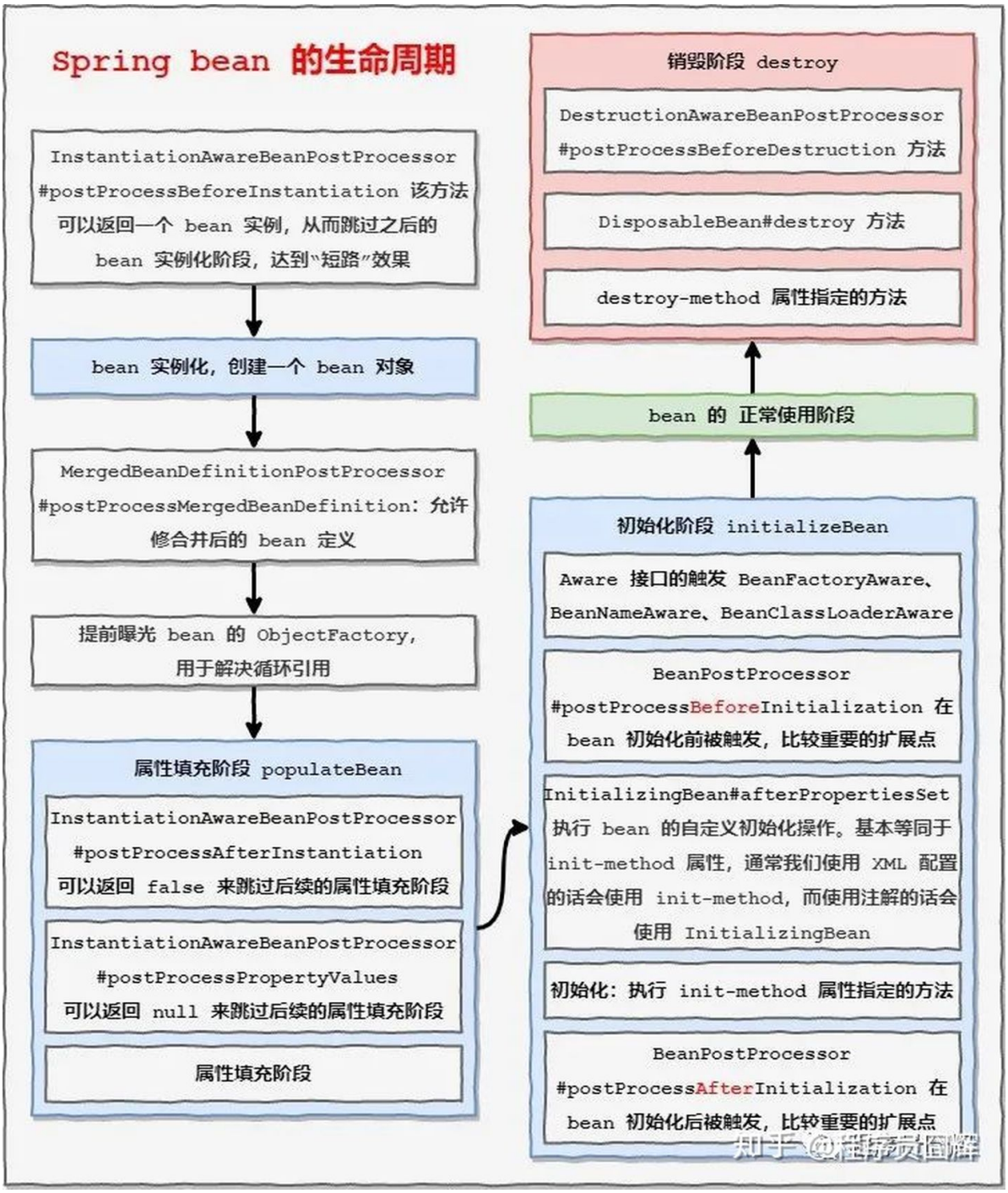
> Spring IoC 的容器构建流程（8分）

核心的构建流程如下，也就是 refresh 方法的核心内容：



> Spring bean 的生命周期（10分）

bean 的生命周期主要有以下几个阶段，深色底的5个是比较重要的阶段。



> BeanFactory 和 FactoryBean 的区别（6分）

BeanFactory：Spring 容器最核心也是最基础的接口，本质是个工厂类，用于管理 bean 的工厂，最核心的功能是加载 bean，也就是 getBean 方法，通常我们不会直接使用该接口，而是使用其子接口。

FactoryBean：该接口以 bean 样式定义，但是它不是一种普通的 bean，它是个工厂 bean，实现该接口的类可以自己定义要创建的 bean 实例，只需要实现它的 getObject 方法即可。

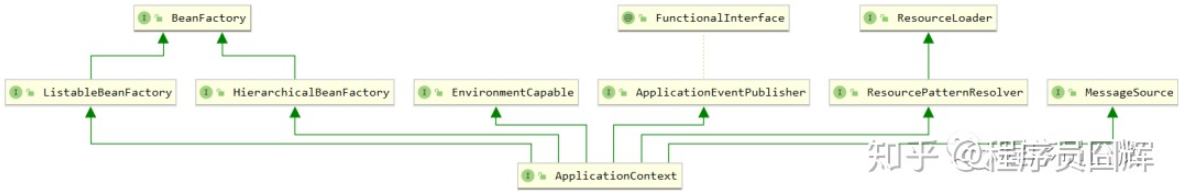
FactoryBean 被广泛应用于 Java 相关的中间件中，如果你看过一些中间件的源码，一定会看到 FactoryBean 的身影。

一般来说，都是通过 FactoryBean#getObject 来返回一个代理类，当我们触发调用时，会走到代理类中，从而可以在代理类中实现中间件的自定义逻辑，比如：RPC 最核心的几个功能，选址、建立连接、远程调用，还有一些自定义的监控、限流等等。

> BeanFactory 和 ApplicationContext 的区别（6分）

BeanFactory：基础 IoC 容器，提供完整的 IoC 服务支持。

ApplicationContext：高级 IoC 容器，BeanFactory 的子接口，在 BeanFactory 的基础上进行扩展。包含 BeanFactory 的所有功能，还提供了其他高级的特性，比如：事件发布、国际化信息支持、统一资源加载策略等。正常情况下，我们都是使用的 ApplicationContext。



这边以电话来举个简单的例子：

我们家里使用的 “座机” 就类似于 BeanFactory，可以进行电话通讯，满足了最基本的需求。

而现在非常普及的智能手机，iPhone、小米等，就类似于 ApplicationContext，除了能进行电话通讯，还有其他很多功能：拍照、地图导航、听歌等。

> Spring 的 AOP 是怎么实现的（5分）

本质是通过动态代理来实现的，主要有以下几个步骤。

1、获取增强器，例如被 Aspect 注解修饰的类。

2、在创建每一个 bean 时，会检查是否有增强器能应用于这个 bean，简单理解就是该 bean 是否在该增强器指定的 execution 表达式中。如果是，则将增强器作为拦截器参数，使用动态代理创建 bean 的代理对象实例。

3、当我们调用被增强过的 bean 时，就会走到代理类中，从而可以触发增强器，本质跟拦截器类似。

> **多个AOP的顺序怎么定（6分）**

通过 Ordered 和 PriorityOrdered 接口进行排序。PriorityOrdered 接口的优先级比 Ordered 更高，如果同时实现 PriorityOrdered 或 Ordered 接口，则再按 order 值排序，值越小的优先级越高。

> Spring 的 AOP 有哪几种创建代理的方式（9分）

Spring 中的 AOP 目前支持 JDK 动态代理和 Cglib 代理。

通常来说：如果被代理对象实现了接口，则使用 JDK 动态代理，否则使用 Cglib 代理。另外，也可以通过指定 proxyTargetClass=true 来实现强制走 Cglib 代理。

> JDK 动态代理和 Cglib 代理的区别（9分）

1、JDK 动态代理本质上是实现了被代理对象的接口，而 Cglib 本质上是继承了被代理对象，覆盖其中的方法。

2、JDK 动态代理只能对实现了接口的类生成代理，Cglib 则没有这个限制。但是 Cglib 因为使用继承实现，所以 Cglib 无法代理被 final 修饰的方法或类。

3、在调用代理方法上，JDK 是通过反射机制调用，Cglib是通过FastClass 机制直接调用。FastClass 简单的理解，就是使用 index 作为入参，可以直接定位到要调用的方法直接进行调用。

4、在性能上，JDK1.7 之前，由于使用了 FastClass 机制，Cglib 在执行效率上比 JDK 快，但是随着 JDK 动态代理的不断优化，从 JDK 1.7 开始，JDK 动态代理已经明显比 Cglib 更快了。

> JDK 动态代理为什么只能对实现了接口的类生成代理（7分）

根本原因是通过 JDK 动态代理生成的类已经继承了 Proxy 类，所以无法再使用继承的方式去对类实现代理。

> Spring 的事务传播行为有哪些（6分）

1、REQUIRED：Spring 默认的事务传播级别，如果上下文中已经存在事务，那么就加入到事务中执行，如果当前上下文中不存在事务，则新建事务执行。

2）REQUIRES_NEW：每次都会新建一个事务，如果上下文中有事务，则将上下文的事务挂起，当新建事务执行完成以后，上下文事务再恢复执行。

3）SUPPORTS：如果上下文存在事务，则加入到事务执行，如果没有事务，则使用非事务的方式执行。

4）MANDATORY：上下文中必须要存在事务，否则就会抛出异常。

5）NOT_SUPPORTED ：如果上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

6）NEVER：上下文中不能存在事务，否则就会抛出异常。

7）NESTED：嵌套事务。如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

> **Spring 的事务隔离级别（6分）**

Spring 的事务隔离级别底层其实是基于数据库的，Spring 并没有自己的一套隔离级别。

DEFAULT：使用数据库的默认隔离级别。

READ_UNCOMMITTED：读未提交，最低的隔离级别，会读取到其他事务还未提交的内容，存在脏读。

READ_COMMITTED：读已提交，读取到的内容都是已经提交的，可以解决脏读，但是存在不可重复读。

REPEATABLE_READ：可重复读，在一个事务中多次读取时看到相同的内容，可以解决不可重复读，但是存在幻读。

SERIALIZABLE：串行化，最高的隔离级别，对于同一行记录，写会加写锁，读会加读锁。在这种情况下，只有读读能并发执行，其他并行的读写、写读、写写操作都是冲突的，需要串行执行。可以防止脏读、不可重复度、幻读，没有并发事务问题。

> Spring 的事务隔离级别是如何做到和数据库不一致的？（5分）

比如数据库是可重复读，Spring 是读已提交，这是怎么实现的？

Spring 的事务隔离级别本质上还是通过数据库来控制的，具体是在执行事务前先执行命令修改数据库隔离级别，命令格式如下：

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
```

> Spring 事务的实现原理（8分）

Spring 事务的底层实现主要使用的技术：AOP（动态代理） + ThreadLocal + try/catch。

动态代理：基本所有要进行逻辑增强的地方都会用到动态代理，AOP 底层也是通过动态代理实现。

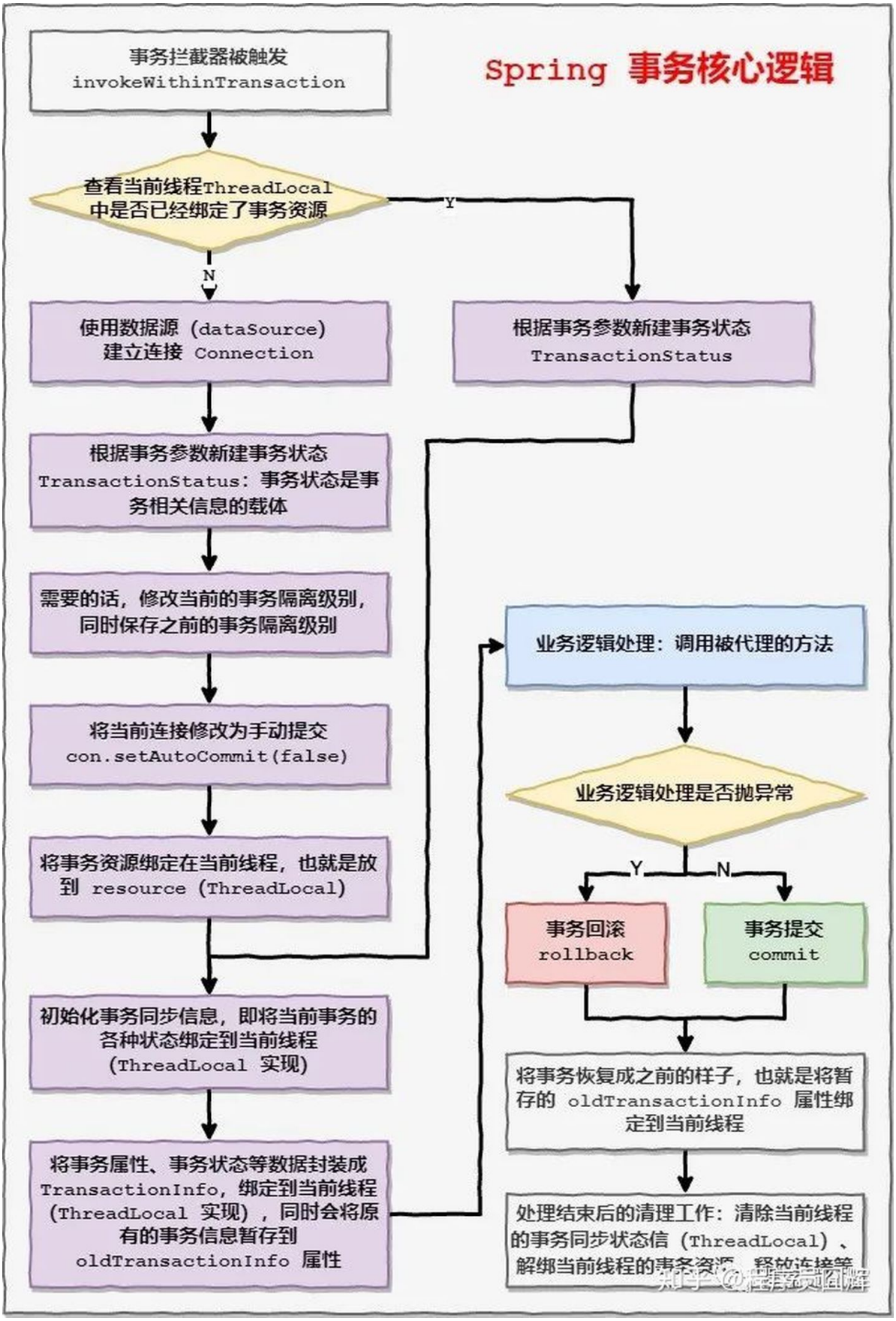
ThreadLocal：主要用于线程间的资源隔离，以此实现不同线程可以使用不同的数据源、隔离级别等等。

try/catch：最终是执行 commit 还是 rollback，是根据业务逻辑处理是否抛出异常来决定。

Spring 事务的核心逻辑伪代码如下：

```
public void invokeWithinTransaction() {

 // 1.事务资源准备

 try {

 // 2.业务逻辑处理，也就是调用被代理的方法

    } catch (Exception e) {

 // 3.出现异常，进行回滚并将异常抛出

    } finally {

 // 现场还原：还原旧的事务信息

    }

 // 4.正常执行，进行事务的提交

 // 返回业务逻辑处理结果

}
```

详细流程如下图所示：



> Spring 怎么解决循环依赖的问题（9分）

Spring 是通过提前暴露 bean 的引用来解决的，具体如下。

Spring 首先使用构造函数创建一个 “不完整” 的 bean 实例（之所以说不完整，是因为此时该 bean 实例还未初始化），并且提前曝光该 bean 实例的 ObjectFactory（提前曝光就是将 ObjectFactory 放到 singletonFactories 缓存）.

通过 ObjectFactory 我们可以拿到该 bean 实例的引用，如果出现循环引用，我们可以通过缓存中的 ObjectFactory 来拿到 bean 实例，从而避免出现循环引用导致的死循环。

举个例子：A 依赖了 B，B 也依赖了 A，那么依赖注入过程如下。

- 检查 A 是否在缓存中，发现不存在，进行实例化
- 通过构造函数创建 bean A，并通过 ObjectFactory 提前曝光 bean A
- A 走到属性填充阶段，发现依赖了 B，所以开始实例化 B。
- 首先检查 B 是否在缓存中，发现不存在，进行实例化
- 通过构造函数创建 bean B，并通过 ObjectFactory 曝光创建的 bean B
- B 走到属性填充阶段，发现依赖了 A，所以开始实例化 A。
- 检查 A 是否在缓存中，发现存在，拿到 A 对应的 ObjectFactory 来获得 bean A，并返回。
- B 继续接下来的流程，直至创建完毕，然后返回 A 的创建流程，A 同样继续接下来的流程，直至创建完毕。

这边通过缓存中的 ObjectFactory 拿到的 bean 实例虽然拿到的是 “不完整” 的 bean 实例，但是由于是单例，所以后续初始化完成后，该 bean 实例的引用地址并不会变，所以最终我们看到的还是完整 bean 实例。

> Spring 能解决构造函数循环依赖吗（6分）

答案是不行的，对于使用构造函数注入产生的循环依赖，Spring 会直接抛异常。

为什么无法解决构造函数循环依赖?

上面解决逻辑的第一句话：“首先使用构造函数创建一个 “不完整” 的 bean 实例”，从这句话可以看出，构造函数循环依赖是无法解决的，因为当构造函数出现循环依赖，我们连 “不完整” 的 bean 实例都构建不出来。

> Spring 三级缓存（6分）

Spring 的三级缓存其实就是解决循环依赖时所用到的三个缓存。

singletonObjects：正常情况下的 bean 被创建完毕后会被放到该缓存，key：beanName，value：bean 实例。

singletonFactories：上面说的提前曝光的 ObjectFactory 就会被放到该缓存中，key：beanName，value：ObjectFactory。

earlySingletonObjects：该缓存用于存放 ObjectFactory 返回的 bean，也就是说对于一个 bean，ObjectFactory 只会被用一次，之后就通过

earlySingletonObjects 来获取，key：beanName，早期 bean 实例。

> @Resource 和 @Autowire 的区别（7分）

1、@Resource 和 @Autowired 都可以用来装配 bean

2、@Autowired 默认按类型装配，默认情况下必须要求依赖对象必须存在，如果要允许null值，可以设置它的required属性为false。

3、@Resource 如果指定了 name 或 type，则按指定的进行装配；如果都不指定，则优先按名称装配，当找不到与名称匹配的 bean 时才按照类型进行装配。

> @Autowire 怎么使用名称来注入（6分）

配合 @Qualifier 使用，如下所示：

```
@Component

public class Test {

 @Autowired

 @Qualifier("userService")

 private UserService userService;

}
```

> @PostConstruct 修饰的方法里用到了其他 bean 实例，会有问题吗（5分）

该题可以拆解成下面3个问题：

1、@PostConstruct 修饰的方法被调用的时间

2、bean 实例依赖的其他 bean 被注入的时间，也可理解为属性的依赖注入时间

3、步骤2的时间是否早于步骤1：如果是，则没有问题，如果不是，则有问题

解析：

1、PostConstruct 注解被封装在 CommonAnnotationBeanPostProcessor中，具体触发时间是在 postProcessBeforeInitialization 方法，从 doCreateBean 维度看，则是在 initializeBean 方法里，属于初始化 bean 阶段。

2、属性的依赖注入是在 populateBean 方法里，属于属性填充阶段。

3、属性填充阶段位于初始化之前，所以本题答案为没有问题。

> bean 的 init-method 属性指定的方法里用到了其他 bean 实例，会有问题吗（5分）

该题同上面这题类似，只是将 @PostConstruct 换成了 init-method 属性。

答案是不会有问题。同上面一样，init-method 属性指定的方法也是在 initializeBean 方法里被触发，属于初始化 bean 阶段。

> 要在 Spring IoC 容器构建完毕之后执行一些逻辑，怎么实现（6分）

1、比较常见的方法是使用事件监听器，实现 ApplicationListener 接口，监听 ContextRefreshedEvent 事件。

2、还有一种比较少见的方法是实现 SmartLifecycle 接口，并且 isAutoStartup 方法返回 true，则会在 finishRefresh() 方法中被触发。

两种方式都是在 finishRefresh 中被触发，SmartLifecycle在ApplicationListener之前。

> Spring 中的常见扩展点有哪些（5分）

1、ApplicationContextInitializer

initialize 方法，在 Spring 容器刷新前触发，也就是 refresh 方法前被触发。

2、BeanFactoryPostProcessor

postProcessBeanFactory 方法，在加载完 Bean 定义之后，创建 Bean 实例之前被触发，通常使用该扩展点来加载一些自己的 bean 定义。

3、BeanPostProcessor

postProcessBeforeInitialization 方法，执行 bean 的初始化方法前被触发；
postProcessAfterInitialization 方法，执行 bean 的初始化方法后被触发。

4、@PostConstruct

该注解被封装在 CommonAnnotationBeanPostProcessor 中，具体触发时间是在 postProcessBeforeInitialization 方法。

5、InitializingBean

afterPropertiesSet 方法，在 bean 的属性填充之后，初始化方法（init-method）之前被触发，该方法的作用基本等同于 init-method，主要用于执行初始化相关操作。

6、ApplicationListener，事件监听器

onApplicationEvent 方法，根据事件类型触发时间不同，通常使用的 ContextRefreshedEvent 触发时间为上下文刷新完毕，通常用于 IoC 容器构建结束后处理一些自定义逻辑。

7、@PreDestroy

该注解被封装在 DestructionAwareBeanPostProcessor 中，具体触发时间是在 postProcessBeforeDestruction 方法，也就是在销毁对象之前触发。

8、DisposableBean

destroy 方法，在 bean 的销毁阶段被触发，该方法的作用基本等同于

destroy-method，主用用于执行销毁相关操作。

> Spring中如何让两个bean按顺序加载？（8分）

1、使用 @DependsOn、depends-on

2、让后加载的类依赖先加载的类

```
@Component

public class A {

 @Autowire

 private B b;

}
```

3、使用扩展点提前加载，例如：BeanFactoryPostProcessor

```
@Component

public class TestBean implements BeanFactoryPostProcessor {

 @Override

 public void postProcessBeanFactory(ConfigurableListableBeanFactory

          configurableListableBeanFactory) throws BeansException {
```

```
 // 加载bean
      beanFactory.getBean("a");
  }
}
```

> 使用 Mybatis 时，调用 DAO接口时是怎么调用到 SQL 的（7分）

详细的解析见：《面试题：mybatis 中的 DAO 接口和 XML 文件里的 SQL 是如何建立关系的？》

简单点说，当我们使用 Spring+MyBatis 时：

1、DAO接口会被加载到 Spring 容器中，通过动态代理来创建

2、XML中的SQL会被解析并保存到本地缓存中，key是SQL 的 namespace + id，value 是SQL的封装

3、当我们调用DAO接口时，会走到代理类中，通过接口的全路径名，从步骤2的缓存中找到对应的SQL，然后执行并返回结果