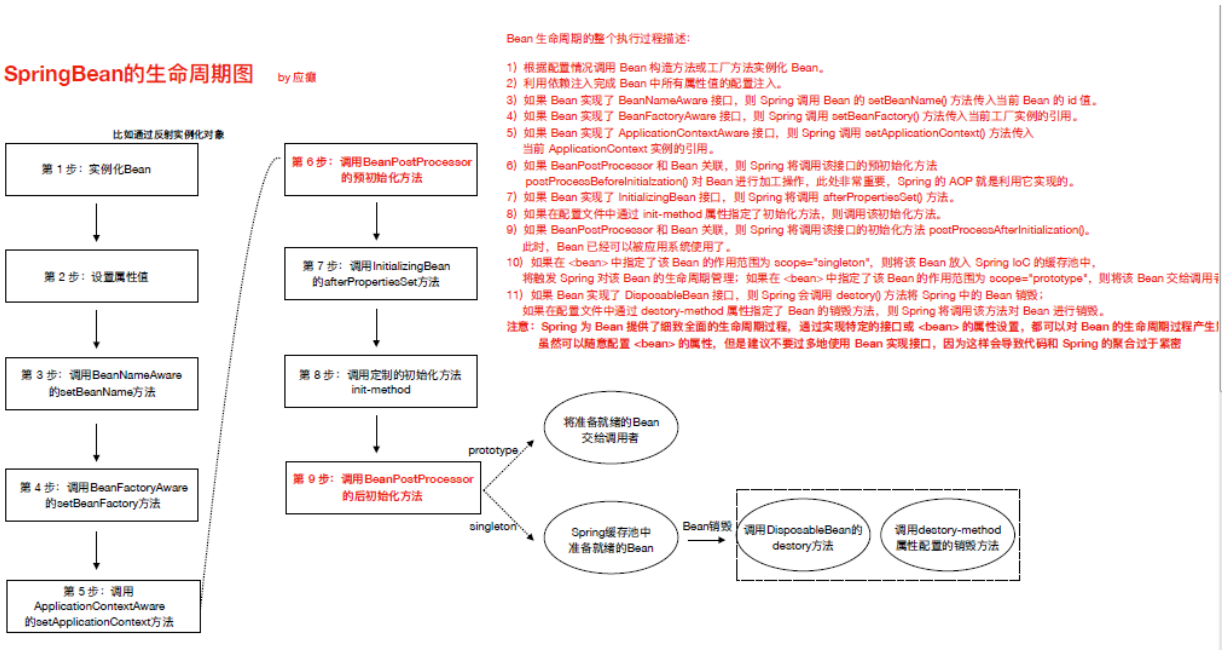
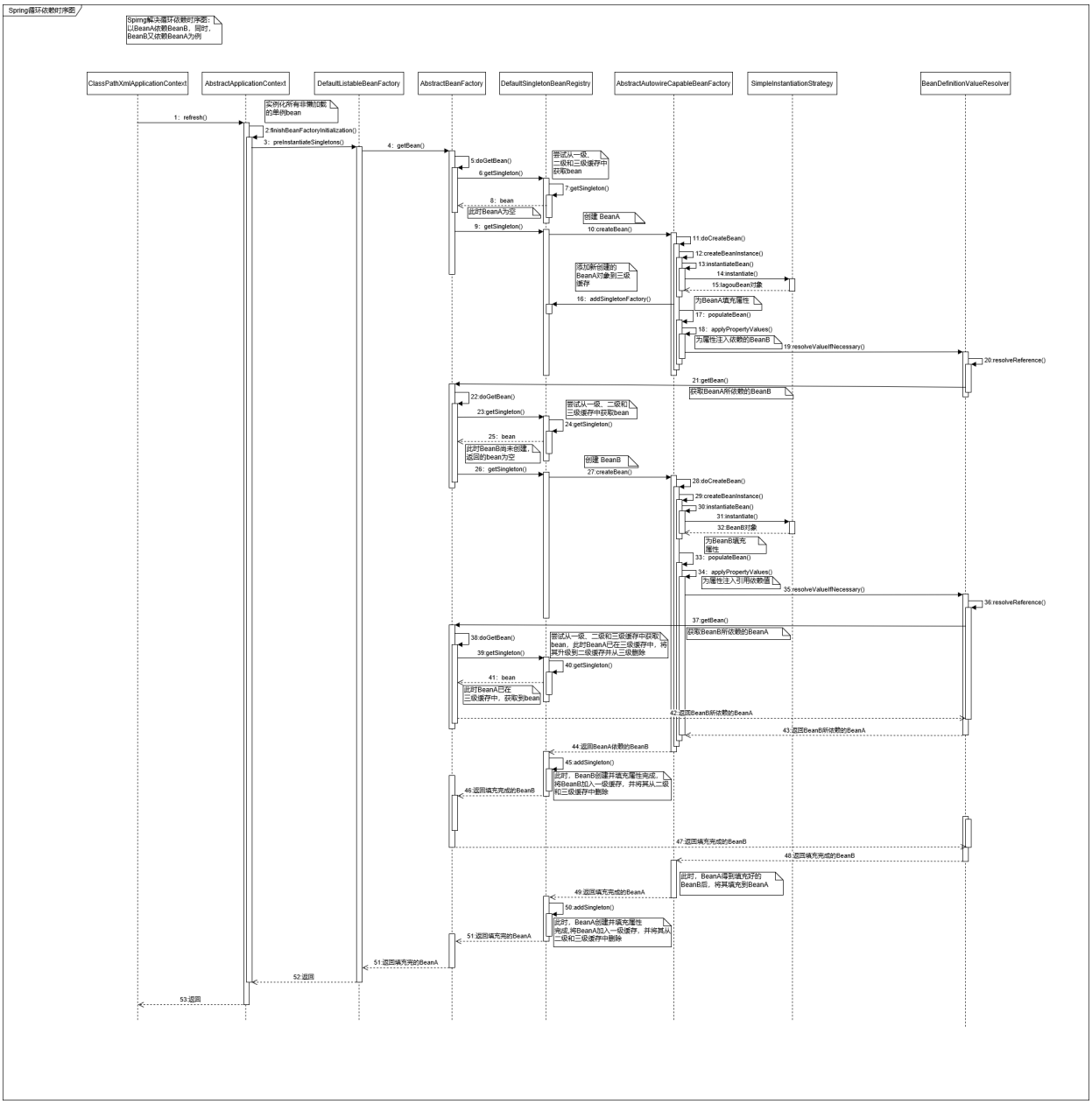
1、SpringBean的生命周期

spring中bean的创建是一个复杂的过程，并且在创建的过程中提供了一系列扩展接口供使用者扩展，俗话说，“数无形时少直觉，形无数时难入微”，意思就是说只有数据没有图形很难有直观深刻的体会，只有图形没有深入追究的数据则很难真正理解其背后的真实原理，所以，以图形的形式记录SpringBean生命周期的整个过程则要更加直观，而在整理图形的过程中势必要亲自阅览源码，弄清原理。SpringBean的整个生命周期如下图所示：



2、SpringBean的循环依赖

Spring中循环依赖是指两个或多个被spring所管理的bean相互依赖，形成了环形。Spring采用了三级缓存的机制很好地解决了循环依赖问题，以BeanA依赖于BeanB，同时BeanB又依赖于BeanA为例，通过阅读追踪源码，解决循环依赖问题的调用时序图如下：


Spring循环依赖时序图
Spring解决循环依赖时序图：以BeanA依赖BeanB，同时，BeanB又依赖BeanA为例
ClassPathXmlApplicationContext
AbstractApplicationContext
DefaultListableBeanFactory
AbstractBeanFactory
DefaultSingletonBeanRegistry
AbstractAutowireCapableBeanFactory
SimpleInstantiationStrategy
BeanDefinitionValueResolver
实例化所有非懒加载的单例bean
1：refresh()
2:finishBeanFactoryInitialization()
3：preInstantiateSingletons()
4：getBean()
5:doGetBean()
6:getSingleton()
尝试从一级、二级和三级缓存中获取bean
7:getSingleton()
8：bean
此时BeanA为空
9：getSingleton()
创建 BeanA
10:createBean()
11:doCreateBean()
12:createBeanInstance()
13:instantiateBean()
14:instantiate()
15:lagouBean对象
添加新创建的BeanA对象到三级缓存
16：addSingletonFactory()
为BeanA填充属性
17：populateBean()
18：applyPropertyValues()
为属性注入依赖的BeanB
19:resolveValueIfNecessary()
20:resolveReference()
21:getBean()
获取BeanA所依赖的BeanB
22:doGetBean()
23:getSingleton()
尝试从一级、二级和三级缓存中获取bean
24:getSingleton()
25：bean
此时BeanB尚未创建，返回的bean为空
26：getSingleton()
创建 BeanB
27:createBean()
28:doCreateBean()
29:createBeanInstance()
30:instantiateBean()
31:instantiate()
32:Bean对象
为BeanB填充属性
33：populateBean()
34：applyPropertyValues()
为属性注入引用依赖值
35:resolveValueIfNecessary()
36:resolveReference()
37:getBean()
获取BeanB所依赖的BeanA
38:doGetBean()
39:getSingleton()
尝试从一级、二级和三级缓存中获取bean，此时BeanA已在三级缓存中，将其升级到二级缓存并从三级删除
40:getSingleton()
41：bean
此时BeanA已在二级缓存中，获取到bean
42:返回BeanB所依赖的BeanA
43:返回BeanB所依赖的BeanA
44:返回BeanA依赖的BeanB
45:addSingleton()
此时，BeanB创建并填充属性完成，将BeanB加入一级缓存，并将其从二级和三级缓存中删除
46:返回填充完成的BeanB
47:返回填充完成的BeanB
48:返回填充完成的BeanB
此时，BeanA得到填充好的BeanB后，将其填充到BeanA
49:返回填充完成的BeanA
50:addSingleton()
此时，BeanA创建并填充属性完成，将BeanA加入一级缓存，并将其从二级和三级缓存中删除
51:返回填充完的BeanA
51:返回填充完的BeanA
52:返回
53:返回

3、Spring Aop

名词	解释
JoinPoint(连接点)	它是指可以把增强代码加入到业务主线中的点，方法开始时，结束时，正常执行完成时，方法异常时等，这些特殊的点称为连接点，项目中每个方法都有连接点，连接点是一种候选点
PointCut(切入点)	指定AOP思想想要影响的方法是哪些，描述那些感兴趣的方法
Advice(通知/增强)	切面类中用于提供增强逻辑的方法，分类有：前置通知(@Before)、后置通知(@AfterReturning)、异常通知(@AfterThrowing)、最终通知(@After)、环绕通知(@Around)
Target(目标对象)	要被代理的目标对象
Proxy(代理)	对目标对象进行增强后的代理对象
Weaving(织入)	把增强逻辑应用到目标对象来创建代理对象的过程，spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入
Aspect(切面)	把增强代码定义到一个类中，这个类就是一个切面，切面=切入点+增强逻辑

4、事务

1、事务的四大特性

原子性（Atomicity） 原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

从操作的角度来描述，事务中的各个操作要么都成功要么都失败

一致性（Consistency） 事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

例如转账前A有1000，B有1000。转账后A+B也得是2000。

一致性是从数据的角度来说的，（1000，1000）（900，1100），不应该出现（900，1000）

隔离性（Isolation） 事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，

每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

比如：事务1给员工涨工资2000，但是事务1尚未被提交，员工发起事务2查询工资，发现工资涨了2000块钱，读到了事务1尚未提交的数据（脏读）

持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

2、事务的隔离级别

不考虑隔离级别，会出现以下情况：（以下情况全是错误的），也即为隔离级别在解决事务并发问题

脏读：一个线程中的事务读到了另外一个线程中**未提交**的数据。

不可重复读：一个线程中的事务读到了另外一个线程中已经提交的**update**的数据（前后内容不一样）

场景：

员工A发起事务1，查询工资，工资为1w，此时事务1尚未关闭

财务人员发起了事务2，给员工A张了2000块钱，**并且提交了事务**

员工A通过事务1再次发起查询请求，发现工资为1.2w，原来读出来1w读不到了，叫做不可重复读

虚读（幻读）：一个线程中的事务读到了另外一个线程中已经提交的insert或者delete的数据（前后条数不一样）

场景：

事务1查询所有工资为1w的员工的总数，查询出来了10个人，此时事务尚未关闭

事务2财务人员发起，新来员工，工资1w，向表中插入了2条数据，**并且提交了事务**

事务1再次查询工资为1w的员工个数，发现有12个人，见了鬼了

数据库共定义了四种隔离级别：

Serializable（串行化）：可避免脏读、不可重复读、虚读情况的发生。（串行化） 最高

Repeatable read（可重复读）：可避免脏读、不可重复读情况的发生。(幻读有可能发生) 第二

该机制下会对要update的行进行加锁

Read committed（读已提交）：可避免脏读情况发生。不可重复读和幻读一定会发生。 第三

Read uncommitted（读未提交）：最低级别，以上情况均无法保证。(读未提交) 最低

注意：级别依次升高，效率依次降低

MySQL的默认隔离级别是：REPEATABLE READ

查询当前使用的隔离级别：`select @@tx_isolation;`

设置MySQL事务的隔离级别：`set session transaction isolation level xxx;`（设置的是当前mysql连接会话的，并不是永久改变的）

3、事务的传播行为

事务往往在service层进行控制，如果出现service层方法A调用了另外一个service层方法B，A和B方法本身都已经被添加了事务控制，那么A调用B的时候，就需要进行事务的一些协商，这就叫做事务的传播行为。

A调用B，我们站在B的角度来观察来定义事务的传播行为

PROPAGATION_REQUIRED	**如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。**
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。**
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	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。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED	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，则执行与PROPAGATION_REQUIRED类似的操作。

4、Spring中事务的API

```
1  package org.springframework.transaction;
2
3  import org.springframework.lang.Nullable;
4
5  /**
6   * This is the central interface in Spring's transaction infrastructure.
7   * Applications can use this directly, but it is not primarily meant as API:
8   * Typically, applications will work with either TransactionTemplate or
9   * declarative transaction demarcation through AOP.
10  *
11  * <p>For implementors, it is recommended to derive from the provided
12  * {@link org.springframework.transaction.support.AbstractPlatformTransactionManager}
13  * class, which pre-implements the defined propagation behavior and takes care
14  * of transaction synchronization handling. Subclasses have to implement
15  * template methods for specific states of the underlying transaction,
16  * for example: begin, suspend, resume, commit.
17  *
18  * <p>The default implementations of this strategy interface are
19  * {@link org.springframework.transaction.jta.JtaTransactionManager} and
20  * {@link org.springframework.jdbc.datasource.DataSourceTransactionManager},
21  * which can serve as an implementation guide for other transaction strategies.
22  *
23  * @author Rod Johnson
24  * @author Juergen Hoeller
25  * @since 16.05.2003
26  * @see org.springframework.transaction.support.TransactionTemplate
27  * @see org.springframework.transaction.interceptor.TransactionInterceptor
```

```
 * @see org.springframework.transaction.interceptor.TransactionProxyFactoryBean
 */
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws Tr


    void commit(TransactionStatus status) throws TransactionException;


    void rollback(TransactionStatus status) throws TransactionException;

}

```

此接口是Spring的事务管理器核心接口。Spring本身并不支持事务实现，只是负责提供标准，应用底层支持什么样的事务，需要提供具体实现类。此处也是策略模式的具体应用。在Spring框架中，也为我们内置了一些具体策略，例如：DataSourceTransactionManager，HibernateTransactionManager等等。（和`HibernateTransactionManager`事务管理器在`spring-orm-5.1.12.RELEASE.jar`中）

Spring JdbcTemplate（数据库操作工具）、Mybatis（mybatis-spring.jar）————> DataSourceTransactionManager

Hibernate框架 ————————> HibernateTransactionManager

DataSourceTransactionManager 归根结底是横切逻辑代码，声明式事务要做的就是使用Aop（动态代理）来将事务控制逻辑织入到业务代码