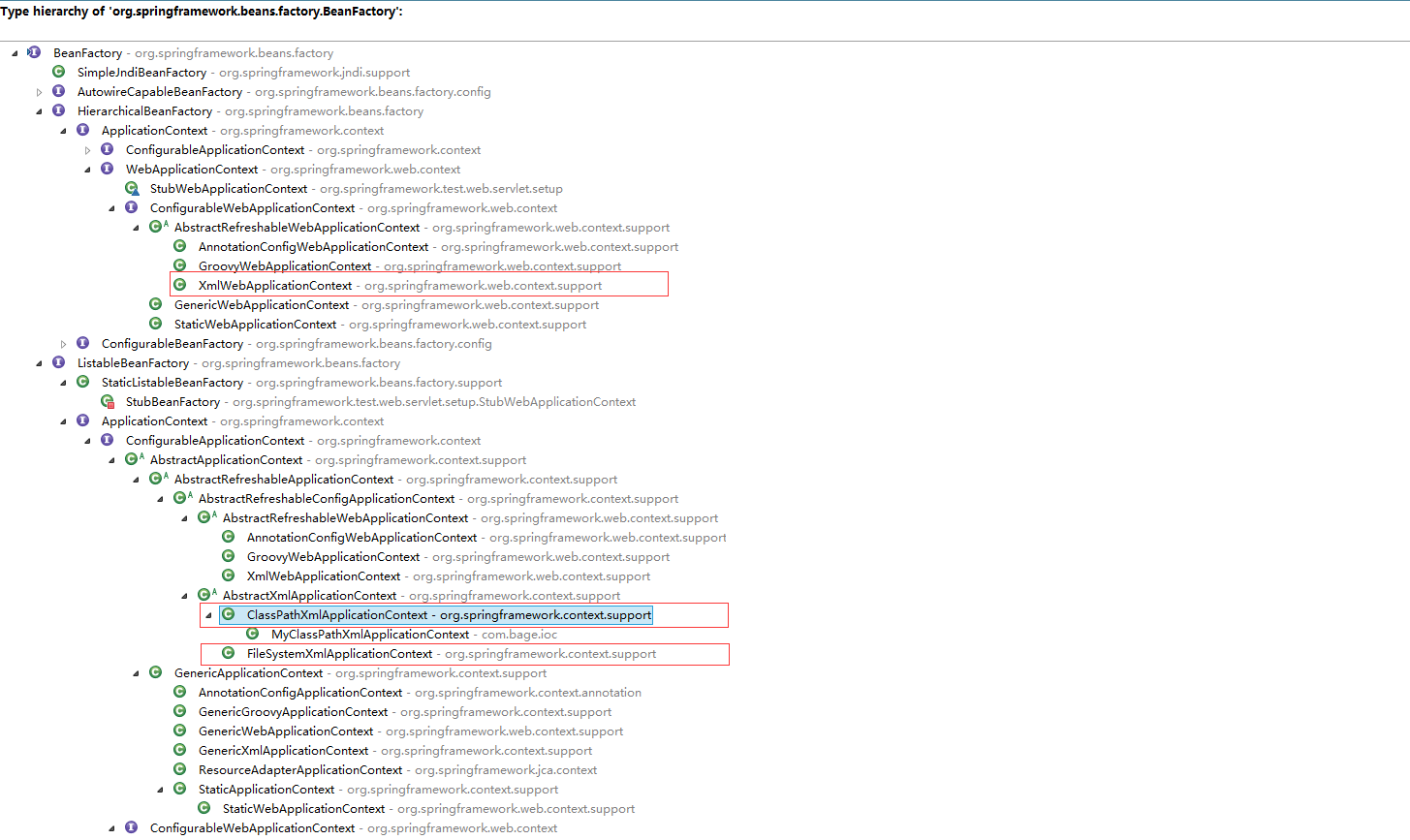
# Spring学习笔记

## Ioc的设计

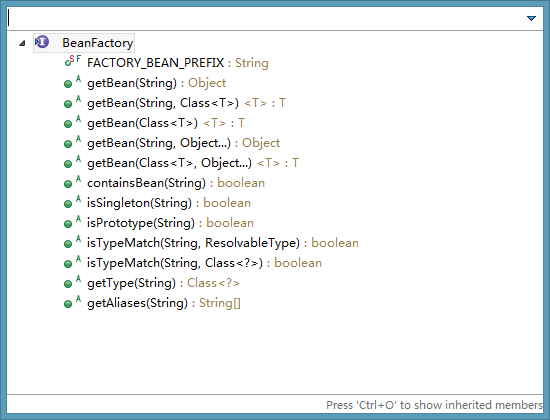
#### 整体结构



#### BeanFactory

<https://www.cnblogs.com/aspirant/p/9082858.html>

定义最基本的IOC的功能，定义最基本的IOC容器的最基本形式。



负责生产和管理bean。

BeanFactory是IOC容器的核心接口，它的职责包括：实例化Bean、定位、配置应用程序中的对象及建立这些对象间的依赖。

BeanFactory只是个接口，并不是IOC容器的具体实现，但是Spring容器给出了很多种实现，如 DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext等，其中XmlBeanFactory就是常用的一个，该实现将以XML方式描述组成应用的对象及对象间的依赖关系。它为其他具体的IOC容器提供了最基本的规范，例如DefaultListableBeanFactory，XmlBeanFactory，ApplicationContext 等具体的容器都是实现了BeanFactory，再在其基础之上附加了其他的功能。

#### IoC容器启动过程

<https://www.cnblogs.com/ITtangtang/p/3978349.html>

• 第一个过程是Resource资源定位。这个Resouce指的是BeanDefinition的资源定位。这个过程就是容器找数据的过程，就像水桶装水需要先找到水一样。

• 第二个过程是BeanDefinition的载入过程。这个载入过程是把用户定义好的Bean表示成Ioc容器内部的数据结构，而这个容器内部的数据结构就是BeanDefition。

• 第三个过程是向IOC容器注册这些BeanDefinition的过程，这个过程就是将前面的BeanDefition保存到HashMap中的过程。

#### ApplicationContext

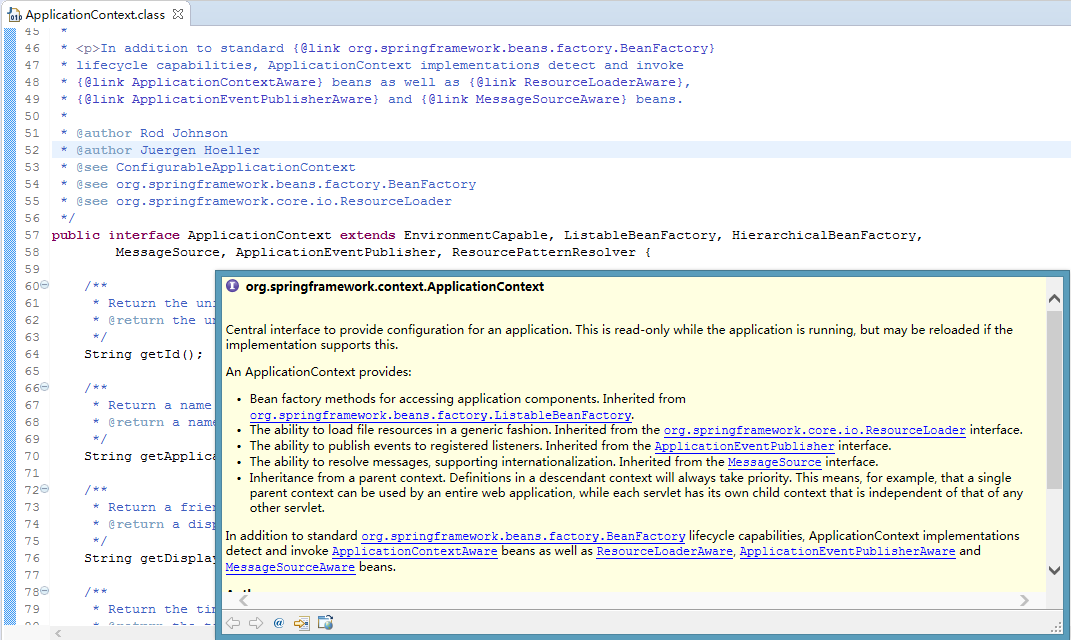
##### 附加功能

<https://blog.csdn.net/lxyso/article/details/45446757>

<https://www.cnblogs.com/aspirant/p/9082858.html>

ApplicationContext实现了BeanFactory接口。还提供了更完整的框架功能。

* 利用MessageSource进行国际化
* 强大的事件机制(Event)
* 底层资源的访问



##### 常用实现

<https://www.cnblogs.com/henuyuxiang/p/3919457.html>

ClassPathXmlApplicationContext ：编译目录下

XmlWebApplicationContext ： web应用上下文

FileSystemXmlApplicationContext ： 文件系统目录下，绝对路径，否则默认在当前进程根目录下

## 请求过程

#### 请求过程

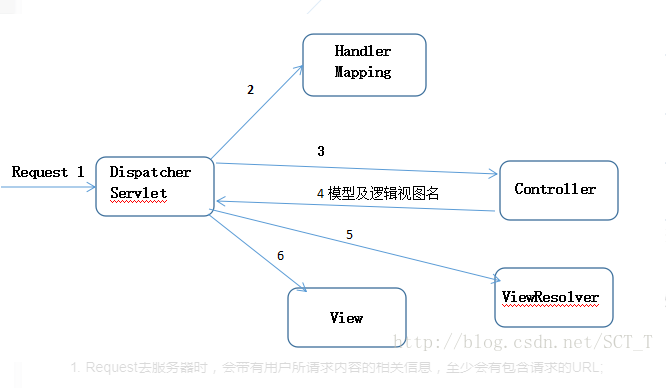
<https://blog.csdn.net/SCT_T/article/details/53998216?utm_source=blogxgwz0>

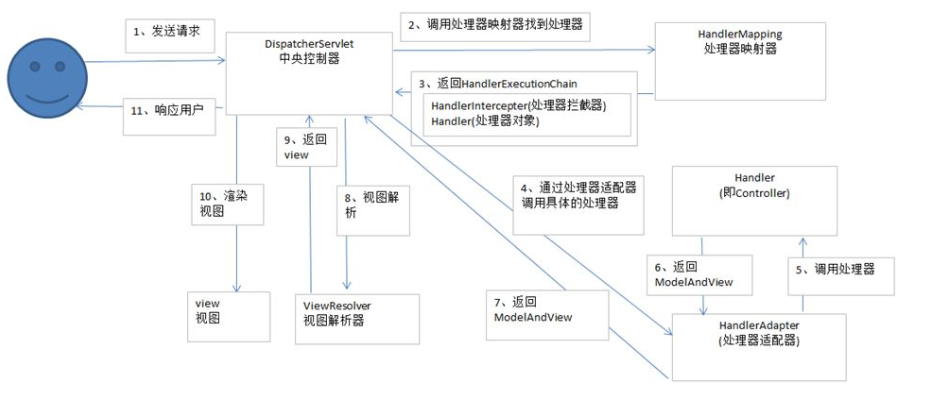
<https://www.cnblogs.com/dreamworlds/p/5396112.html>

<https://www.cnblogs.com/java1024/p/8556519.html>

<https://www.cnblogs.com/glorywzm/p/6503141.html>

##### 结构图







##### 各个过程

* Request去服务器时，会带有用户所请求内容的相关信息，至少会有包含请求的URL;
* DispatcherServlet 查询HandlerMapping来确定将Request发送给哪个Controller处理；
* 选定了合适的Controller，DispatcherServlet会将Request发送过去，等待Controller对请求的处理。
* Controller在完成逻辑处理后，会产生一些信息，需要返回给用户并且要在浏览器上显示的。这些信息称为模型（Model） ,而这些信息的加工格式化者就是HTML,信息需要发送一个视图（通常是JSP）。
* Controller所做的最后一件事就是将模型数据打包，并且标出用于渲染输出的视图名称，发还给DispatcherServlet.
* ViewResolver根据DispatcherServlet的逻辑名查找真正的视图。

视图将数据渲染输出，并且通过这个输出将响应对象传递给客户端。

⑴ 用户发送请求至前端控制器DispatcherServlet

⑵ DispatcherServlet收到请求调用HandlerMapping处理器映射器。

⑶ 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

⑷ DispatcherServlet通过HandlerAdapter处理器适配器调用处理器

⑸ 执行处理器(Controller，也叫后端控制器)。

⑹ Controller执行完成返回ModelAndView

⑺ HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

⑻ DispatcherServlet将ModelAndView传给ViewReslover视图解析器

⑼ ViewReslover解析后返回具体View

⑽ DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。

⑾ DispatcherServlet响应用户。

1.用户发送请求到DispatchServlet

2.DispatchServlet根据请求路径查询具体的Handler

3.HandlerMapping返回一个HandlerExcutionChain给DispatchServlet

　HandlerExcutionChain：Handler和Interceptor集合

4.DispatchServlet调用HandlerAdapter适配器

5.HandlerAdapter调用具体的Handler处理业务

6.Handler处理结束返回一个具体的ModelAndView给适配器

ModelAndView:model-->数据模型，view-->视图名称

7.适配器将ModelAndView给DispatchServlet

8.DispatchServlet把视图名称给ViewResolver视图解析器

9.ViewResolver返回一个具体的视图给DispatchServlet

10.渲染视图

11.展示给用户

#### 事务

<https://www.cnblogs.com/yixianyixian/p/8372832.html>

<https://www.jdon.com/concurrent/acid-database.html>

##### 四大特性

事务有四个特性：ACID

**原子性（Atomicity）**

事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。

**一致性（Consistency）**

一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，在现实中的数据不应该被破坏。如果一个操作触发辅助操作（级联，触发器），这些也必须成功，否则交易失败。如果系统是由多个节点组成，一致性规定所有的变化必须传播到所有节点。

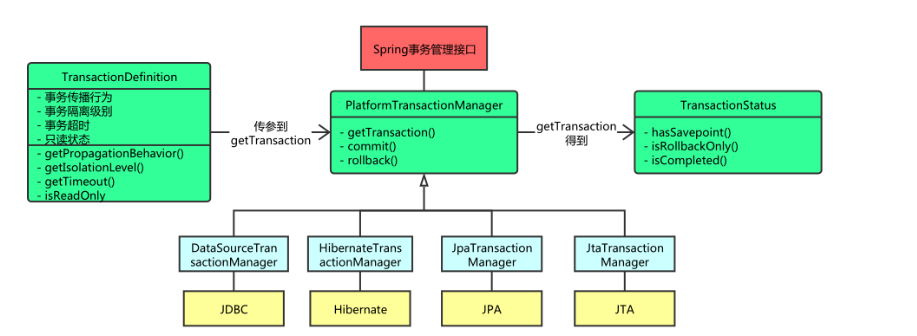
**隔离性（Isolation）**

可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。

**持久性（Durability）**

一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

##### 相关接口



##### 传播行为

Propagation：宣传; 传播，传输，蔓延

| **传播行为** | **含义** |
| --- | --- |
| PROPAGATION\_REQUIRED  propagation\_required | 表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务 |
| PROPAGATION\_SUPPORTS  propagation\_supports | 表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行 |
| PROPAGATION\_MANDATORY  propagation\_mandatory(强制的; 命令的) | 表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常 |
| PROPAGATION\_REQUIRED\_NEW  propagation\_required\_new | 表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager |
| PROPAGATION\_NOT\_SUPPORTED  propagation\_not\_supported | 表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager |
| PROPAGATION\_NEVER  propagation\_never | 表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常 |
| PROPAGATION\_NESTED  propagation\_nested | 表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION\_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务 |

##### 隔离级别

<http://www.cnblogs.com/s-b-b/p/5845096.html>

**READ COMMITTED（提交读）**

大多数数据库系统的默认隔离级别是READ COMMITTED，这种隔离级别就是一个事务的开始，只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。这种级别会出现读取旧数据的现象

**READ UNCIMMITTED（未提交读）/ 脏读（Dirty reads）**

脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。这就是事务还没提交，而别的事务可以看到他其中修改的数据的后果，也就是脏读。

**REPEATABLE READ（可重复读）**

REPEATABLE READ解决了脏读的问题，该级别保证了每行的记录的结果是一致的，也就是上面说的读了旧数据的问题，但是却无法解决另一个问题，幻行，顾名思义就是突然蹦出来的行数据。指的就是某个事务在读取某个范围的数据，但是另一个事务又向这个范围的数据去插入数据，导致多次读取的时候，数据的行数不一致。

**SERIALIZABLE（可串行化）**

SERIALIZABLE是最高的隔离级别，它通过强制事务串行执行（注意是串行），避免了前面的幻读情况，由于他大量加上锁，导致大量的请求超时，因此性能会比较底下，再特别需要数据一致性且并发量不需要那么大的时候才可能考虑这个隔离级别

#### Bean生命周期

<https://www.cnblogs.com/kenshinobiy/p/4652008.html>

<https://www.cnblogs.com/redcool/p/6397398.html>

实现：/study-spring/src/main/java/com/bage/lifecircle/LifeCircleBean.java

1. 实例化一个Bean，也就是我们通常说的new

2. 按照Spring上下文对实例化的Bean进行配置，也就是IOC注入，使用依赖注入，Spring按照Bean定义信息配置Bean所有属性

3. 如果这个Bean实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的是Spring配置文件中Bean的ID

4. 如果这个Bean实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()，传递的是Spring工厂本身（可以用这个方法获取到其他Bean）

5. 如果这个Bean实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文，该方式同样可以实现步骤4，但比4更好，以为ApplicationContext是BeanFactory的子接口，有更多的实现方法

6. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor经常被用作是Bean内容的更改，并且由于这个是在Bean初始化结束时调用After方法，也可用于内存或缓存技术

7. 如果这个Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法

8. 如果这个Bean关联了BeanPostProcessor接口，将会调用postAfterInitialization(Object obj, String s)方法

注意：以上工作完成以后就可以用这个Bean了，那这个Bean是一个single的，所以一般情况下我们调用同一个ID的Bean会是在内容地址相同的实例

9. 当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean接口，会调用其实现的destroy方法

10. 最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法