**Ioc实现原理**

**Ioc**:控制反转容器,提供了另外一种模块协作的方式,它通过静态配置或者运行期注册的方式将组件或者一个POJO,普通Java对象管理起来,被托管的bean组件的生命周期由容器统一负责,并且Ioc最强大的功能在于它不仅能管理所有组件的生命周期,还能通过配置组件之间或类之间的依赖关系,自动完成被托管组件的依赖,可以说,只要配置好组件的初始化参数,依赖条件,那么以后在运行期的任何时候,你都可以直接向容器获取该组件,这样的好处是什么呢? 又是为了解决什么痛点呢?为什么在ORM框架里又要引入IOC的概念呢?

**某些对象是业务焦点.或者说是整个系统的入口点,以及负责业务较多的部分,为了实现高效的迭代,高度的可伸缩性,这个组件需要依赖很多其他组件完成业务操作,例如本系统的cn.bronzeware.muppet.core.Context,完成了Session生命周期的管理,Criteria的管理,事物操作,测试帮助的支持,等内聚性不高的功能,需要依赖于其他组件.作为对外提供服务的入口类,xml配置文件的入口,它的初始化将完成整个orm框架的初始化,初始化中,需要产生系统内部使用的临时数据结构,中间数据结构,这些数据结构,组件往往又是其他组件的输入,为了避免传递具体的数据结构,如果B组件的初始化依赖于A组件生成的数据结果,那么我们会向B组件传递已完成初始化的A组件引用.**

**传递方式无外乎setter注入,构造函数注入,等**

**Setter注入和构造器参数注入**

**setter注入的局限性.我们希望一个类在完成实例化之后应该是一个标准的状态完整的类,而setter很明显是在实例化之后调用的,那么对象实例化之后并没有提供一个状态完整的类,如果一个类的实例化阶段就依赖于其他组件,很明显,使用setter把实例化过程放到setter方法中,这样类的实例化逻辑就太多于复杂. 另外如果一个组件被很多组件所依赖,那么依赖一次,就需要为期setter一次.有了ioc之后只需要静态注册,或者动态注册一次即可被托管.**

**构造器参数注入没有上述setter注入的问题,它在实例化阶段就已经收到了全部的所依赖的组件,那么构造器有什么其他问题呢?问题在于Java等静态语言的局限,如果后期变更,或者添加新的依赖,那么就需要频繁修改方法签名.那么有没有一种比较好的方式呢?可以传递一个参数,所有的组件都向它获取,有,这个参数就是ioc容器,只要在各个以来组建中传递此ioc容器,各个组件之间本来n的平方的依赖,就变成n个组件对ioc的依赖,依赖关系统一由ioc管理,每一次依赖关系的修改,只需要需求方内部决定即可.所谓的控制反转也就是这个概念,将控制权转移,将所有组件的控制权放在Ioc容器里,其他组件请求某种服务.**

**简单来说,痛点就是依赖关系错综复杂,并且缺乏管理.ioc是为了管理依赖关系,但并不能降低组件本身之间的依赖程度,例如B依赖A,引入ioc之后,B还是依赖于A,只是获取方式变成了向ioc容器申请,这种申请可以通过静态配置方式固化,可视化,方便部署与修改.**

Muppet中IOC的具体实现:

**首先定义BeanFactory.bean工厂,提供基本的获取操作.**

/\*\*

\* <p>BeanFactory定义了获取bean的所有方式,通过自定义的beanName,bean的类型,已经通过混合获取bean

\* 通过这三种方式,你可以获取到已经注册进BeanFactory的所有bean.

\* <h3>当该bean没有被BeanFacory托管时,按定义的三种获取规则将无法返回bean实例给调用者.这时将抛出

\* {@link SuchBeanNotFoundException }异常.

\* <p>

\*

\* @see {@link BaseBeanFactory} {@link ApplicationContext}

\* @throws SuchBeanNotFoundException

\* @author yuhaiqiang email: yuhaiqiangvip@sina.com

\*/

public interface BeanFactory {

/\*\*

\* <p>推荐使用的获取 bean实例的方式,通过 该种方式获取bean实例,避免了对象的强制类型转换.

\* <p>{@code getBean(BeanFactory.class)}

\* @return clazz类型的bean实例.

\* @throws SuchBeanNotFoundException 当容器没有托管该类型的实例时,抛出此异常

\* @param clazz 类型

\*/

<T> T getBean(Class<T> clazz);

/\*\*

\* <p>首先通过beanName去寻找相应bean实例,如果存在将其强转为 T 类型.此时会检查 获取的bean实例是否

\* 是T类型如果不匹配,那么将按照 {@link #getBean(Class)}去获取,如果依然获取不到将抛出 {@link SuchBeanNotFoundException}

\* <p>如果通过beanName获取不到响应的bean实例,那么将按照 {@link #getClass()}方式去获取bean实例.如果获取不到

\* 将抛出 {@link SuchBeanNotFoundException}

\* <h3>不推荐通过该方式获取bean实例

\* @throws SuchBeanNotFoundException 按照获取规则获取不到bean实例时

\* @param beanName 通过beanName获取

\* @param clazz 期待bean的类型

\* @return T类型 指定beanName的bean

\*/

<T> T getBean(String beanName, Class<T> clazz);

/\*\*

\* <p>通过指定beanName获取对应的bean.需要注意,通过这种方式,调用方无法知道bean的类型,需要一个

\* 强制类型转换

\* <h3>按照获取规则无法获取指定bean时,将抛出 {@link SuchBeanNotFoundException}

\* @throws SuchBeanNotFoundException

\* @param beanName 指定的beanName

\* @return 指定beanName的实例

\*/

Object getBean(String beanName);

/\*\*

\* 获取当前beanFactory托管的所有bean实例.

\* @return 返回对象实例数组.如果没有托管bean.那么将返回空数组

\*/

Object[] getBeans();

}

**BaseBeanFactory的实现**

/\*\*

\* <p>{@link BeanFactory}接口的基本实现,提供了四种获取bean的方式.

\* <li>{@link #getBean(Class)}

\* <li>{@link #getBean(String)}

\* <li>{@link #getBean(String, Class)}

\* <li>{@link #getBeans()}

\* 具体请参考 {@link BeanFactory}接口

\* <h3>除实现BeanFactory接口外, {@link BaseBeanFactory}还提供了注册服务.

\* <p>{@link #registerBean(Class, Object)}注册Class类型的bean实例.

\* <p>{@link #registerBean(Object)} 通过获取object对象的类型,注册bean实例

\* <p> {@link #registerBean(String, Object)} 按照给定beanName注册bean实例

\*

\* @see ApplicationContext

\*

\* @throws SuchBeanNotFoundException

\* @author yuhaiqiang email: yuhaiqiangvip@sina.com

\* @time 17/2/12.

\*/

public class BaseBeanFactory implements BeanFactory{

public <T> T getBean(Class<T> clazz) {

if(clazzMap.containsKey(clazz)){

return (T)clazzMap.get(clazz);

}

else{

for(Map.Entry<Class<?>, Object> entry:clazzMap.entrySet()){

if(entry.getValue().getClass().equals(clazz)){

return (T)entry.getValue();

}

}

}

throw new SuchBeanNotFoundException(clazz);

}

public boolean hasBean(String beanName){

try{

this.getBean(beanName);

return true;

} catch (SuchBeanNotFoundException e){

return false;

}

}

public <T> boolean hasBean(Class<T> beanClazz){

try{

this.getBean(beanClazz);

return true;

}catch (SuchBeanNotFoundException e){

return false;

}

}

@Override

public <T> T getBean(String beanName, Class<T> clazz) {

if(this.hasBean(beanName)){

Object bean = getBean(beanName);

if(bean.getClass().equals(clazz)){

return (T)bean;

}

else if(this.hasBean(clazz) == false){

return (T)bean;

}

else{

//如果存在beanClass的bean

throw new SuchBeanNotFoundException(beanName, clazz);

}

}else{

//如果不存在相应key值的bean但是存在相应class的bean

if(this.hasBean(clazz)){

T bean = this.getBean(clazz);

return bean;

}

else{

//如果均不存在则抛出异常

throw new SuchBeanNotFoundException(beanName ,clazz);

}

}

}

@Override

public Object getBean(String beanName) {

if(beanNameMap.containsKey(beanName)){

return beanNameMap.get(beanName);

}else{

throw new SuchBeanNotFoundException(beanName);

}

}

public Object registerBean(String beanName,Object object){

Object oldObject = null;

if(beanNameMap.containsKey(beanName)){

oldObject = beanNameMap.get(beanName);

}

beanNameMap.put(beanName, object);

return oldObject;

}

public Object registerBean(Object object){

Class clazz = object.getClass();

return registerBean(clazz, object);

}

public Object registerBean(Class clazz, Object object){

Object oldObject = null;

if(clazzMap.containsKey(clazz)){

oldObject = clazzMap.get(clazz);

}

clazzMap.put(clazz, object);

return oldObject;

}

@Override

public Object[] getBeans() {

Object[] beanNameObjects = beanNameMap.values().toArray();

Object[] clazzObjects = clazzMap.values().toArray();

Object[] result = new Object[beanNameObjects.length + clazzObjects.length];

System.arraycopy(beanNameObjects,0,result,0,beanNameObjects.length);

System.arraycopy(clazzObjects, 0, result, beanNameObjects.length, clazzObjects.length);

return result ;

}

}

BeanFactory与BaseBeanFactory仅仅是提供了按照beanName,Class注册和获取bean的基本操作,核心数据结构也仅仅是HashMap,BeanFactory并没有关心其他的逻辑,例如这个bean是如何配置的,是如何初始化的,如何完成依赖关系的管理.BeanFactory仅仅作为bean工厂,完成一些简单的事情.

**那么bean生命周期的管理,配置依赖的管理,以及bean实例的初始化,如何完成呢?**

ApplicationContext接口的定义

/\*\*

\* <p>ApplicationContext继承了BeanFactory接口,对外提供获取服务.

\* 并且自定义了注册方法,判断方法.ApplicationContext提供了完整的bean

\* 存取服务

\* <p>ApplicationContext接口是IOC暴露在外的少数接口之一,你可以通过传递该实例 ,完成bean之间依赖关系的管理,当你想传递某个组件时就将改组件注册进容器,容器 托管后,任何想获取该组件的类都可以通过ApplicationContext获取.

\* <p>因此.bean之间本来显式的bean依赖配置,通过ApplicationContext管理起来.依赖

\* 关系统统都指向了ApplicationContext

\* <h3>ApplicationContext 并不像Spring 中ApplicationContext那样作为一个只读的

\* 接口,它对外提供注册服务.

\* @see BeanFactory

\* @see BaseBeanFactory

\* @time 17/2/12.

\*/

public interface ApplicationContext extends BeanFactory{

public Object registerBean(String beanName,Object object);

public Object registerBean(Object object);

public Object registerBean(Class clazz, Object object);

public <T> T containsBean(Class<T> clazz);

}

这样的设计或许与Spring有很大不同,Spring中ApplicationContext是一个只读的接口,只提供了get操作,并没有提供注册,(如果想要运行时注册,需要借助其他接口).但是我们期待的ioc需要管理复杂初始化逻辑的对象,所以需要提供运行时注册功能.

ApplicationContext的具体实现

以按照类型查找这一简单方式来介绍

public <T> T getBean(Class<T> clazz) {

// 如果是单例类那么可以从容器中获取

T t = null;

try {

t = beanFactory.getBean(clazz);

if (!isSingleton(clazz)) {

t = beanInitializor.initializeBean(clazz);

refreshBean(t);

}

} catch (SuchBeanNotFoundException e) {

Object object = null;

try {

object = beanInitializor.initializeBean(clazz);

if(Utils.empty(object)){

//如果initializeBean返回null,说明是接口，抛出异常是初始化失败。

throw e;

}

} catch (InitializeException e1) {

Logger.error(e.getMessage());

throw e1;

}

//this.registerBean(object);

this.registerBean(clazz, object);

return (T) object;

}

return t;

}

1. 按照类型在beanFactory中查找.如果查找到的类型不是单例生命周期,那么就交由beanInitializer bean实例化器实例化该类型的实例,如果是单例的生命周期那么就直接返回该bean实例.
2. 如果抛出SuchBeanNotFoundException,那么就通过beanInitialiter实例化器初始化该类型实例,如果返回为空,说明是该类型是接口,无法找到对应实现类型,将抛出异常.
3. 如果实例化成功,那么将注册将该bean实例以该类型注册进Context中.并且返回该类型实例

GetBean(Class<T> clazz);方法不是一个功能单一的方法,它的目的是获取一个bean实例,如果不存在甚至尝试去加载它.这很明显违反了单一职责.ApplicationContext在实现中调用了BaseBeanFactory的简约getBean方法,采用获取bean,如果不存在就去加载它的目的就是为了更好的体验,为什么一定要静态配置bean,为什么动态注册bean只能注册已经实例化好的bean,为什么get的时候要提醒自己之前已经将它实例化,这个get方法更像是一种懒加载策略,只有你get但是get不到的时候,才去加载它,总之如果你想要简约的getBean方法,只能去求助于BaseBeanFactroy了,后面,我打算提供一个抽象级别更高于ApplicationContext,但低于BeanFactory的实现,它完成生命周期的管理,但是并不像 get it,If Null Initialize It.这样”蛮横”

ApplicationContex中其他查找规则的getBean方法,思路都是类似的,如果get不到,尝试去加载它.

ApplicationContext注册bean是如果做到的.

@Override

public Object registerBean(Class clazz, Object object) {

refreshBean(object);

beforeRegister(clazz, object);

Object result = ((BaseBeanFactory) beanFactory)

.registerBean(clazz, object);

afterRegister(clazz, object, result);

return result;

}

注册bean之前,现刷新一下它,装配它所需要的依赖.

然后设计保护方法在registerBean方法前后做增强,before,与after都可以被子类重写.

AbstractApplicationiContext实现中,before,与after方法前后仅仅做了日志处理,推荐子类在实现before方法,after方法时,调用父类的方法.

实际上,后面在做自动化回归测试时,就是利用了before,after方法做了一次bean统计.

ApplicationContext的初始化逻辑:

private void initialize(){

//获取自动扫描包的目录

autoScanPackage = (String[]) config.getProperty(ApplicationConfig.AUTO\_SCAN\_PACKAGE\_KEY);

if(Utils.notEmpty(autoScanPackage)){

//获取包下所有的类

List<Class<?>> clazzList = ReflectUtil.getClasses(autoScanPackage);

//System.out.println(autoScanPackage);

//ArrayUtil.println(clazzList);

//获取所有添加Component注解的类

clazzList = baseScanComponentClass(clazzList);

//获取所有单例类

//clazzList = this.singletonBeans(clazzList);

publishConfigBeanClassEvent(clazzList);

//初始化所有 singleton Class实例

List list = initializeBeans(clazzList);

//ArrayUtil.println(list);

//将componentExecutor和ApplicationContext放进去

//自动装配

autowireds(list);

//解析aware和capable

awareAndCapable();

isRefresh = true;

}else{

throw new BeanInitializationException("bean initialization has error happend , can not found autowird configs, please check");

}

}

ApplicationContex中的getBean,和registerBean作为一个方法,已经可以对外提供服务,但是AutowiredApplicationContext的实现中,它的目的是扫描一个包下所有的添加指定注解的Class,在它的实现中,

1. 需要获取需要扫描的包名
2. 描包下需要被托管的类型,
3. 实例化所有需要被托管的类型,
4. 刷新该实例,注册进容器,
5. 修改容器状态.

第一步中,获取扫描包名,,只是借助于ApplicationConfig,它负责与外界输入打交道,所有的配置信息都可以向它获取.扫描包下需要被托管的类型,AutowiredApplicationContext只会扫描添加Component注解的类型,但是当我们又扩展需求时,又当如何呢,例如我们需要管理其他注解的类型,或者其他配置信息的类型呢,或者其他过滤条件呢?

这时我们提炼出来一种机制.提供模版方法给子类实现,当容器初始化时,会将子类和父类扫描逻辑扫描出来的结果,合并然后托管到beanFactory.稍后可以看到自动化回归工具中,实现了这个方法,将Test注解添加到我们托管规则中,这个机制有效的扩展了我们的托管规则.子类可以自由的扩展拓展规则.

**Bean实例化部分:**

Bean实例化部分作为ioc容器极其重要的一部分,因为ioc容器就是声称解放new操作,它就像一个工厂一样,你给定一个模版,他给你new出来无数的实例,大部分的对象实例化是非常容器的,java提供的强大的反射机制,通过Class对象实例,即可以实例化出该类型对象实例,当类型提供了默认构造方法,即无参数的构造方法时,这一切都简单易行,当类型的构造方法需要构造参数呢?这样一来,bean的实例化就很复杂了,考虑muppet orm 框架内部使用的 简化这个功能,重新定义一下需求.我们需要它能按照构造方法参数类型自动到ioc框架去获取该类型实例,不能按照beanName进行引用,就像Spring做到的那样.实际实现过程中具体包括一下几个步骤:

1. 检查类型是否是接口,如果是接口直接返回null,这也是唯一返回null的地方
2. 检查类型的所有构造器,是否存在默认构造器,或者存在DefaultConstructor注解的构造器,如果存在,那么将其视为默认构造器.
3. 解析默认构造器之后,我们需要解析默认构造器的参数,获得参数数组.
4. 按照参数数组定义的顺序到ApplicationContext中去getBean,获取相关类型的bean实例,如果该类型bean实例还没有被初始化,那么继续去实例化………,为了方便理解,可以把参数数组实例化的过程理解为 数据结构中树的深度优先遍历问题.被实例化的类型作为树的节点,它的构造参数数组即是它的孩子,按照深度优先遍历,遍历孩子节点,如果节点已经初始化,那么返回实例,如果没有被初始化,将以孩子节点作为根节点,继续遍历,直至遍历整棵树.
5. “实例化死锁”的问题:当遍历一棵子树时,如果子树中的节点出现了到根节点的最短路径中出现的节点,那么我们认为出现了bean实例化时的循环依赖问题,简单来说,如果实例化A类时,A类的构造的参数的实例化反需要依赖于A类,这样就出现了实例化死锁问题,谁也没法先被实例化.这个问题可以通过一个链表记录当前遍历过程的栈.每次实例化新的类型时,检查当前类型是否在正在被实例化的栈中,避免出现死锁问题.
6. 默认构造方法的参数数组已经实例化完成之后,即可完成bean的实例化.实际代码中还会需要在bean实例化时是托管给拦截器管理器,所谓的拦截器管理器就是负责解析aop拦截类,生成bean的代理逻辑,根据cglib动态代理生成bean类型的代理bean.

Bean实例化器,实例化一个bean主要分以上六个步骤,如果任何一个步骤出错,将抛出InitializeException初始化异常.bean实例化器负责了容器中被托管的bean的实例化,被容器托管的bean实例,都是通过它实例化的,同时也能正确的被代理,但是通过容器的注册服务,动态注册进容器的bean实例,目前还不能被代理,如果他们需要被代理,那就需要在实例化一遍.所以考虑需要被代理的bean实例,最好通过静态注册,通过容器实例化bean.这也是我们推荐的方式.

至此注册服务,获取服务,bean实例化服务部分等等,ioc模块的核心部分的原理已经梳理清楚了.不过我们还留下了一个问题 ,那就是bean实例化的最后一步交给拦截器管理器去实例化.

**Aop的引入**

项目开发到中期的时候,遇到的现实问题是,每一个小版本的迭代,除了功能开发之外,还需要做回归测试,设计测试样例,对功能做回归,避免当前的小版本对之前的系统侵入过大,导致不兼容问题,或者重大的bug,回归测试本身是非常重要的,但是由于功能越来越多,小版本,小功能的频繁添加,导致回归测试非常频繁,重要的是测试过程中,工作枯燥,重复性高,而且人工操作易于出错,时间较长,导致整个版本迭代的生命周期延长,在这种情况下,我打算将版本迭代后的回归测试过程自动化,具体目标,是针对以往版本的测试样例自动执行,并且在各个版本之间进行比较,为了避免测试输出检查过于复杂,只将最终的输出和以往版本做diff,比较,同时生成xml文件将输出结果持久化保存.

最终设计的方案如下:

使用ioc框架托管测试样例,同时测试样例需要像Junit框架那样,在测试方法上添加Test注解,这样,ioc框架管理所有测试样例,获取测试样例的方法后,一次性执行,同时在每一个方法执行前后,将控制台输出,错误输出,返回值等保存起来,写入到文件中,在做控制台重定向的过程中,采用的方案是Aop设计,对所有的bean实例做切面拦截.对每一次测试样例方法调用都做重定向.当然还可以做一些其他操作,例如执行时间统计等.只要通过ioc容器获取的bean实例都是可以切面拦截的.需要说明,Aop设计过程中,目前仅仅实现根据注解拦截,也就是拦截器可以注册需要拦截的注解方法,而不能像Spring Aop那样,可以根据异常拦截,根据全类名拦截等.

拦截器管理器.通过ioc容器静态注册方式托管的bean实例实际上都是由该接口实例化的.设计这个接口的意义在于ioc所需要实例化的需要管理的类应该是已经被代理的bean.当bean实例化完成之后的所有调用都是可以被捕捉到的,被拦截到的,而被代理的bean,它被代理的逻辑是什么,也就是是如何被代理的。这也是InterceptorManage接口需要考虑的问题.

拦截器管理器接口:

public interface InterceptorManage {

public Object intercept(Class targetClass, Class[] paramClazzs,Object[] params);

}

可以看到只有一个intercept方法,这个方法接受了要被代理的类型,参数名,参数列表等等,返回也仅仅是一个bean实例.那么在拦截器管理器的实现类中,到底做了哪些工作呢?

下面看一下拦截器管理器具体步骤

1. 扫描每一个Class文件时都会出发Bean\_Class\_Config事件,这样通过监听这个事件接口,这样可以将所有拦截器类上配置的拦截信息全部管理起来.例如目标拦截方法名,目标拦截注解名,拦截器类型,拦截器方法名,拦截器方法类型(Before,After,Around等),将这些拦截器元数据配置信息统一管理起来,保存在拦截器管理器中
2. 在实例化该实例时,拦截器管理器需要检查被实例化的类型.根据该类型上添加的注解找到处理该注解的拦截器列表.
3. 将拦截器列表与该类型传递给拦截器执行处理器.拦截器执行处理器是通过检查拦截器列表,针对具体的类型生成相应的拦截逻辑.简单来说,若有一个拦截器声明了前置拦截,另外一个拦截器声明了后置拦截,还有声明了环绕拦截,等这时拦截器执行处理器,需要管理这些拦截器,按照具体的顺序在拦截增强的方法中依次执行这些拦截逻辑,拦截器执行处理器首先整理出前置拦截器方法列表,环绕拦截器列表,后置拦截器列表等,待实际的拦截逻辑中依次执行这些拦截逻辑.

before(proxy, method, params, methodProxy);

Object result = around(proxy, method, params, methodProxy);

after(proxy, method, params, methodProxy, result);

其中before,around,after方法中都依次执行了拦截这个方法的拦截器方法,执行具体的拦截器方法时,需要像拦截器方法传递参数,传递当前执行方法的元数据,否则拦截方法不知道当前执行的方法是哪个方法,