耦合：程序之间的依赖応

包括：

类之间依赖关系

方法之间依赖关系

解耦：

降低程序之间的依赖応

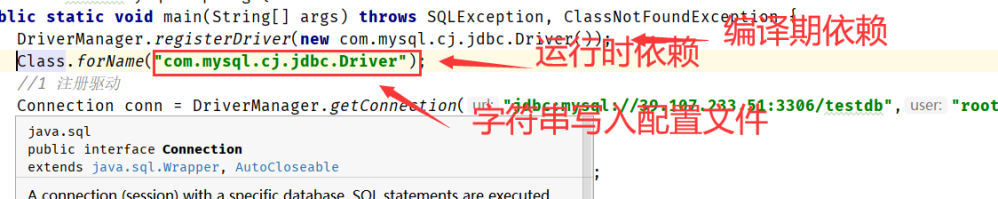
实际操作：

尽量做到；编译期不依赖，运行时才依赖

思路：

1、使用反射创建对象，避免使用NEW

2、通过读取配置文件来获取要创建的对象全限定类名



**工厂方式解耦**

**//单例模式适用于大量重复使用的组件，如业务层service及DAO层对象 使用单利要注意成员变量。**

单例模式（单例多线程的）

     优点：共享一个实例，内存占用少，GC开销小

     缺点：共享资源存在线程安全问题

**//多利对象**

多例模式（单线程的）

    优点：不存在线程安全问题（静态共享资源除外）

    缺点：多个实例，内存占用多，GC开销大

**package** li.factory;  
  
**import** java.io.IOException;  
**import** java.util.Enumeration;  
**import** java.util.HashMap;  
**import** java.util.Map;  
**import** java.util.Properties;  
**import** java.util.Set;  
  
  
**public class** JavaBeanFactory {  
*//定义配置文件存放全类名和名字只的关系***private static** Properties *properties*;  
*//创建一个容器 一个容器用来存放对象 实现制对象复用 不是每次调用服务都新生成 减少资源消耗***public static** Map *beans* = **new** HashMap();  
  
  
*//static***static** {  
*properties* = **new** Properties();  
**try** {  
*//创建properties文件来配置 需要创建的类的对象 需要的全限定类名 在spring里 默认是application.xml  
properties*.load(JavaBeanFactory.**class**.getClassLoader().getResourceAsStream(**"application.properties"**));  
System.***out***.println(**"properties.get(\"accountService\") = "** + *properties*.get(**"accountService"**));  
System.***out***.println(**"properties.get(\"accountDao\") = "** + *properties*.get(**"accountDao"**));  
  
*iniBeans*(*beans*);*//主动初始化对象 ，当配置文件一旦读取则立即根据配置文件创建 Bean容器*} **catch** (IOException e) {  
e.printStackTrace();  
**throw new** ExceptionInInitializerError(**"初始化失败！！！"**);  
} **catch** (IllegalAccessException e) {  
e.printStackTrace();  
} **catch** (InstantiationException e) {  
e.printStackTrace();  
} **catch** (ClassNotFoundException e) {  
e.printStackTrace();  
}  
}  
  
*/\*\*  
\*  
\** ***@param beans*** *Bean容器  
\* 主动初始化对象 ，当配置文件一旦读取则立即 遍历配置文件 根据配置文件创建 Bean容器  
\*/***private static void** iniBeans(Map beans) **throws** ClassNotFoundException, IllegalAccessException, InstantiationException {  
  
Set<Object> keys = *properties*.keySet();  
Object bean = **null**;  
Class clazz = **null**;  
String beanName = **""**;  
String classPath = **""**;  
  
**for** (Object key : keys){  
beanName=(String)key;  
classPath = (String) *properties*.get(beanName);  
clazz = Class.*forName*(classPath);  
bean = clazz.newInstance(); *//通过反射 调用默认构造方法 创建对象*beans.put(beanName,bean);  
  
}  
}  
  
*//这是懒加载的模式 当该方法被调用阶段才去查询或者创建需要的对象***public static** Object getBean(String beanName) {  
Object bean = **null**;  
*//判断容器已经存在该对象就获取直接返回没有的话 重新创建一个*bean = *beans*.get(beanName);  
**if**(bean != **null**){  
System.***out***.println(beanName+**"--已经存在于容器之中 直接返回 = "** + bean);  
**return** bean;  
}  
  
String classPath = (String) *properties*.get(beanName);  
*//System.out.println( beanName + "===============>" + classPath);*Class clazz = **null**;  
**try** {  
clazz = Class.*forName*(classPath);  
bean = clazz.newInstance(); *//通过反射 调用默认构造方法 创建对象*} **catch** (ClassNotFoundException e) {  
e.printStackTrace();  
} **catch** (IllegalAccessException e) {  
e.printStackTrace();  
} **catch** (InstantiationException e) {  
e.printStackTrace();  
}  
  
*beans*.put(beanName,bean);  
  
System.***out***.println(beanName+**"--在容器之中 没有找到已经生成 并置于容器中 = "** + bean);  
  
**return** bean;  
}  
  
  
}