本科毕业设计（论文）

基于 Java 平台的 ORM 框架的设计与实现

于海强

燕 山 大 学

2017 年 6 月

本科毕业设计（论文）

基于Java平台的ORM框架的设计与实现

学　　院：信息科学与工程学院（软件学院）

专　　业：软件工程

学生 姓名：于海强

学　　号：130120010200

指导 教师：王开宇

答辩 日期：

燕山大学毕业设计（论文）任务书

学院：信息科学与工程学院（软件学院）　　系级教学单位：　软件工程系

<table>
<tr><td>学号</td><td>130120010200</td><td>学生姓名</td><td>于海强</td><td>专 业
班 级</td><td>软件工程</td></tr>
<tr><td rowspan="4">题目</td><td>题目名称</td><td colspan="4">Java 平台 ORM 框架的设计与实现</td></tr>
<tr><td>题目性质</td><td colspan="4">1.理工类：工程设计 （　　）；工程技术实验研究型（　　）；
理论研究型（　　）；计算机软件型（ ∨ ）；综合型（　　）。
2.文管类（　　）；3.外语类（　　）；4.艺术类（　　）。</td></tr>
<tr><td>题目类型</td><td colspan="4">1.毕业设计（ ∨ ）　　2.论文（　　）</td></tr>
<tr><td>题目来源</td><td colspan="4">科研课题（　　）　生产实际（ ∨ ）　自选题目（　　）</td></tr>
<tr><td>主要内容</td><td colspan="5">功能实现包括：CRUD 及结果集映射，基于对象生成修改数据表，数据表自动映射实体类（代码生成），内部添加监听器监听容器生命周期，aop 实现切面拦截，事务控制， 查询工具类增强单表查询效率。Ioc 提供依赖注入功能降低内部耦合性</td></tr>
<tr><td>基本要求</td><td colspan="5">实现对象关系映射的，在灵活性与开发效率（使用简洁性）两者间取得平衡，使开发人员快速的上手数据库编程，又可以针对 应用进行定制性的再次开发，实现对 sql 的灵活控制，同时保证系统的高稳定性，伸缩性，要求降低系统内部模块依赖，对开发人员提供友好的异常反馈。以及完善的使用文档</td></tr>
<tr><td>参考资料</td><td colspan="5">《Spring mvc 源代码分析与实现》</td></tr>
<tr><td>周次</td><td>1—4 周</td><td>5—8 周</td><td>9—12 周</td><td>13—16 周</td><td>17 周</td></tr>
</table>

应完成的内容	功能细化分析，确定功能实现优先级，同时核心功能 ioc 模块的编写	实现增删查改功能，要求通过对 JavaBean 的注解，生成相应 sql 语句，及结果集的封装	数据表映射为 JavaBean，及 JavaBean 映射相应数据表，及查询工具类等功能	Aop，事务处理相关功能实现，同时重构内部结构，引入监听器	最终文档，ppt 整理，及毕设答辩准备
指导教师：王开宇 职称：副教授　　2017 年 3 月 3 日					

注：周次完成内容请指导老师根据课题内容自主合理安排。

摘要

Java 平台 ORM 框架通过对 jdbc 的抽象, 用面向对象的思想操纵数据库,分离 sql 语句与 Java 代码,简化开发,使应用的数据库访问层具有高可用性,高可扩展性,高可维护性,具体内容而言,已完成多数据源的管理,在 Session 级别提供多数据源访问,数据库事件订阅通知监控数据操作性能,事务性支持,针对增加.删除.更新操作自动生成 sql 语句,提供面向对象操作,对于简单的查询提供单表工具查询,多表复杂查询提供多种返回结果封装.JavaBean 和 scheme 的双向映射提供完全的工具支持.针对开发中扩展性差,回归测试慢的问题,有针对性开发了 Ioc,和 Aop 框架,降低了模块间的耦合,同时提供了回归测试工具,利用往期版本的 diff,将改变的测试样例可视化展示出来,大大提供回归测试效率,同时保证了准确性.这套工具也可以开放出来供客户端使用.

关键词: ORM;Java;JDBC,回归测试;

Abstract

Abstract Java platform ORM framework based on JDBC abstraction, controls the database with the object-oriented thought, separate SQL statements and Java code, simplify the development, the application of the database access layer with high availability, high scalability, high maintainability, in terms of specific content, has completed the management of multiple data sources, multiple data access is provided in the Session level, event subscription notification database performance monitoring data, transactional support, aimed at increasing. Delete. Updates automatically generated SQL statements, provide object-oriented operation, single table query tools to provide to the simple query, more complex query table offers a variety of results returned encapsulation. JavaBean and scheme of bidirectional mapping to provide a complete tool support. In view of the development of poor extensibility, the problem of low regression testing targeted the development of the Ioc, and the Aop framework, reduces the coupling between modules, and provides the regression testing tool, using the past version of the diff, will change the test sample visualization display, provide a regression test efficiency greatly, at the same time to ensure the accuracy. This tool can also open up for the client to use

Keywords ORM;Java;JDBC;Regression Testing;

目 录

第1章 绪论

1.1 选题的背景、目的和意义

Java 平台以强大的跨平台特性,简单的语法,丰富的类库非常适合于编写大型网络应用,其中数据持久化部分,应用层可以针对 jdbc 编程,避免了不同平台数据库的不同访问方式,但是数据库基于关系数学的特性和应用程序面向对象的编程思想存在不同.需要 orm 在 jdbc 和应用层做映射关联,封装对数据库的访问使应用层可以用面向对象的思想去操纵数据库.

……

1.1.1 选题背景

数据库访问层作为企业级应用至关重要的一环,对程序的健壮性,扩展性,性能,及可维护性都提出更高的要求,原生的 jdbc 仅仅提供了最底层的数据库访问接口,仅仅使用 jdbc 开发,不仅效率低下,而且扩展性,健壮性都难以保证.所以 java 平台有众多的 orm 框架提供了优秀的设计,考虑到应用开发中等通用性需求,做了很多工具封装.

……

1.1.2 选题目的和意义

通过对 Java 平台原有 jdbc 接口,进行轻量封装,简化基本数据检索,更新操作的代码量,通过回调机制将数据操作异常统一管理起来,针对多数据源,统一管理,提供对数据库访问的监控,sql 调优,针对基本数据操作,提供 Sql 自动生成,以及 scheme 和 Pojo 间的双向映射.提高数据访问层开发效率.

……

1.2 国内外研究现状

目前 java 平台主流开源数据持久层框架有 hibernate,mybatis.两种更多是互相补充关系，MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取

结果集。MyBatis 可以对配置和原生 Map 使用简单的 XML 或注解，将接口和 Java 的 POJOs 映射成数据库中的记录。而 Hibernate 不仅负责从 Java 类到数据库表的映射（还包括从 Java 数据类型到 SQL 数据类型的映射），还提供了面向对象的数据查询检索机制，从而极大地缩短的手动处理 SQL 和 JDBC 上的开发时间。

1.3 本文研究内容

本文先通过从 Java ORM 的背景,国内外研究现状,最终目的等方面大致介绍了这个课题,而后.将主要讲述项目最终实现的成果,基本功能介绍,使用介绍,以及完成的与现有主流 orm 框架不同的功能特性,包括使用方式,设计理念等.由于本项目的研究意义,将主要篇幅用于讲解项目整体设计思想,理念,重要功能点的实现原理,包括本项目中 Ioc,Aop 部分的实现,将作为重要内容讲解,其他简单功能将一带而过.

第2章　功能特性

2.1　基本功能

2.1.1　Jdbc 基本功能封装

具体功能:通过配置 Javabean 和 scheme 映射的元数据

通过 Session 当前操作的上下文，提供 insert,query,update,delete 等 DDL 操作支持，同时针对查询提供多类型的结果映射.

代码示例：

```
//初始化 SessionFactory
  SessionFactory sessionFactory = new SessionFactory(“muppet.xml”);
//开启一个会话，默认会话属于当前线程
Session session = sessionFactory.getSession();
     //删除 t_note 表中 value = 2 的记录
String deleteContidition = " value = ? ";
Object[] conditionValue = new Object[]{"2"};
session.delete(Note.class, deleteContidition, conditionValue);
    //删除 t_note 表中主键为 2 的记录
      session.deleteByPrimaryKey(Note.class, 2);
//insert 添加操作
Note note = new Note();
note.setId(10);
note.setPassword(String.valueOf(10));
note.setUser_id(String.valueOf(10));
note.setUsername(String.valueOf(10));
note.setValue(String.valueOf(10));
session.insert(note);
//query 查询操作
//查找主键为 2 的记录
```

```
//返回值为 Note 类型
session.queryById(Note.class, 2)

//单表普通查询，返回值为 ist<Note>类型
session.query(Note.class, " user_id > ? and user_id < ?", new Object[]{2,4})
//自定义 sql 语句，复杂查询。
//返回类型 List<Map<String,Object>>
session.query("select * from t_note ", null)
//查询单个记录.返回类型 Map<String.Object>
session.queryOne("select * from t_note where password = ? or user_id = ?"
                        , new Object[]{3,4})
  //更新操作
  //根据指定查询条件更新相关记录
session.update(note, " pk = ?", new Object[]{2});
  //根据主键更新相关记录
session.updateByPrimaryKey(note)
```

2.1.2 Javabean 向 scheme 映射

入口 xml 文件中配置，将自动解析 javabean 元配置信息生成 scheme

```
<buildtable value="true"></buildtable>
```

Javabean 配置如下：

```
@Table(tablename="t_note")
public class Note {
        //配置映射的列名，是否为空，默认值，类型，长度等
        @Column(columnname="pk",cannull=false
        ,type=@Type(type=SqlType.INT,length=10),defaultvalue="0")
        private int id;
        @Column(columnname="value"
                ,type=@Type(type=SqlType.VARCHAR,length=255))
        private String value;
        @Column(columnname="username"
```

```
                ,type=@Type(type=SqlType.VARCHAR,length=255))
        private String username;
        @Column(columnname="password")
        private String password;
        private String user_id;
}
```

2.1.3 Scheme 向 Javabean 的映射

配置 xml

```
//用来定义哪些表需要被自动生成
<table tableName="tb_bloghouse" domainObjectName="BlogHouse"></table>
//自定义生成的 Javabean 放到哪个包下
<generate packageName="cn.bronzeware.test1"></generate>
    AutoGenerateUtil.generate("muppet.xml");
```

2.2 高级特性

2.2.1 单表查询工具

SessionFactory 是 muppet 的入口类,它的初始化需要配置了数据源,以及一些其他信息的 xml 文件作为输入.初始化完成后可以使用 SessionFactory 创建 session. Session 是最基本的访问数据库的接口,在单表查询工具类中,也是通过 session,我们可以

```
SessionFactory factroy = new SessionFactory("muppet.xml");//创建
```

sessionfactory，传入参数是配置文件的路径，实例配置文件见下

```
        Session session = factroy.getSession(false);//创建 Session,false
```

代表关闭自动提交

```
        Criteria criteria = session.createCriteria(Note.class);//创建工
```

具查询

```
        //criteria.andPropEqual("id", 37);//相等
        //criteria.andPropGreater("id", 36);//大于
        //criteria.andPropGreaterEq("id", 39);//大于等于
```

```
//criteria.andPropGreater("id", 36).order("user_id",
false).order("id", false);//排序，链式调用
//criteria.andPropLess("id", 37);//小于
criteria.andPropGreater("user_id", 30);//大于
criteria.andPropLike("username", "%yuhai1%");//like
criteria.or(criteria1);//or,连接两个查询，做 or 连接
System.out.print(criteria.list());//查询结果，返回结果集
```

2.2.2 事务操作支持

```
SessionFactory factroy = new SessionFactory("muppet.xml");//创建
sessionfactory，传入参数是配置文件的路径，实例配置文件见下
   Session session = factroy.getSession(false);//创建
Session,false 代表关闭自动提交
   Transaction transaction = null;
   try{
         transaction = session.beginTransaction();//开始事务
         ///执行增删查改方法。
         transaction.commit();//提交事务
   }catch(SQLException e){
         if(transaction!=null){
            transaction.rollback();//回滚事务
         }
   }finally{
         session.close();//关闭 session
   }
```

和其他操作相同,依然需要先初始化 SessionFactory,通过 session 工厂获取具体的 Session,需要说明的是默认情况下,通过 getSession 方法获取的 Session 是开启自动提交操作的,也就是通过 session 执行的每一次 ddl 语句都是不能回滚的撤销的,对于某些简单的操作,这是没有问题的,但是对于复杂的业务流程,如果同时存在多次更新操作,需要事务的支持.如果需要事务管理,那么首先应该通过 getSession(false)操作,关闭自动提交,

这样通过 session 的 beginTransaction 操作开启事务.通过 try catch 捕获出现异常的操作,在捕获器里回滚事务,在 try 代码块最后提交事务,最后关闭 session,释放数据库连接.这是整个完整的事务操作流程,可以看到这个过程还是很麻烦的.所以我们通过回调机制,重新设计,提供了一个工具,在 session 工厂中,可以通过 transactionOperationCallback 方法获取 session,transaction,这样整个 session 的生命周期都已经托管,也不用执行事务提交操作,session 关闭,回滚,因为 session 工厂已经帮你完成了开启 session,开启事务,提交事务,关闭事务,关闭 session 等操作,另外如果你需要返回结果集,可以直接在 TransactionExecute 中返回,通过在 transactionOperationCallback 返回值接受即可.

```
sessionFactory.transactionOperationCallback(new
TransactionExecute() {
public Object execute(Session session, Transaction transaction) {
            Criteria<Note> criteria =
        session.createCriteria(Note.class);
        criteria.andPropGreater("user_id", 0);
         Logger.println("查询结果个数： " +
         criteria.select("value,pk,password,user_id")
               .list().size());
          return null;
          }
      });
```

为了方便测试,Session 工厂额外提供了 transactionOperationCallbackTest 方法,这个方法接口同上,但是不同之处在于无论是否抛出异常,最终执行完毕都将会进行回滚,如果你需要检查数据库操作执行结果应该在结束之前,查询,或者你也可以在内部自己选择去提交,但是这个方法会保证如果你没有执行任何数据操作,将回滚操作.这是为了方便于测试.

2.2.3 多数据源管理

当应用业务复杂之时,需要针对具体业务做垂直划分,直接方案就是分库,这时一个应用系统需要分库,这就需要 orm 框架提供多数据源管理,例如一个业务

方法可能需要对两个数据库进行操作,或者不同业务需要针对不同数据源操作.muppet 提供了对这种业务场景提供了支持,你可以通过在配置文件中定义不同数据源,同时为不同数据源定义不同的 key,这样客户端可以通过 key 值获取相应的数据库连接.例如我们配置了一个 key 为 main 的数据源,这样我们通过 sessionFactory.getSession("main")就可以获取"main"数据源连接,同时 muppet 还提供了为不同数据源配置不同的映射,即映射到不同的 bean,这样在框架初始化时就可以访问数据源,检查具体的数据源下映射的 pojo.

2.2.4 数据访问的管理性能监控

Muppet 针对数据库访问做了事件订阅分发机制,可以通过监听数据库事件,达到统计连接频率,连接寿命等指标,具体时间类型包括,启动时连接成功,启动时连接失败,启动连接关闭失败,启动连接关闭成功,普通连接成功,普通连接失败,普通连接关闭失败,普通连接关闭成功,表不存在事件,等这些事件类型还在扩展,目的是通过事件类型,将业务中分散的,时间跨度长的异常,元信息,等统计起来,例如应用中数据库连接过于频繁,具体调用量,如果统计连接时间,单位时间连接数,可以有针对性的调整数据源配置等. 客户端通过监听具体的事件,统计相关指标. 优化应用性能.

2.3 本章小结

本章主要介绍项目实现的功能及特性,通过示例代码详细的说明了 muppet 如何使用,能解决的业务场景,通过讲解基本功能,包括通过配置 pojo 到 scheme 的映射,我们可以通过 pojo 直接完成数据库操作,使开发人员基本告别了简单 Sql 语句的编写,调试,通过对查询结果多种类型的封装,可以满足客户端的多种需要,另外介绍了 pojo 和 scheme 的双向映射,高级特性包括单表查询工具,事务操作支持,以及利用回调机制简化了 Session 的使用,最后介绍多数据源管理,及性能监控等特性的应用场景,及使用方式.

第3章　容器及基本工具的设计

3.1　控制反转容器

3.1.1　控制反转容器的介绍

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?

3.1.2　引入控制反转容器前的痛点

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.

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.

3.1.3　注入方式的不同

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.

3.1.4 应用上下文具体实现

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.

3.1.5 Bean 实例化器的具体实现

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.

3.2 面向切面拦截部分的设计

3.2.1 aop 的场景及解决的的问题

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.

3.2.2 Aop 和自动化回归测试工具的设计

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.

4. 反射工具类,使用 cglib 动态生成 bean 实例.至此 Aop 中生成代理类的逻辑完成.

Aop 的主体工作已经完成,剩下的需要 TestFrameworkApplicationContext 继承 ApplicationContext 完成测试框架内部注解的解析,通过我们之前声明的子类重写扫描方法,可以扩展 bean 托管规则,将添加 Test 注解的类型添加进来.剩下我们值需要实现拦截器方法就可以了.具体的拦截器编写就很简单了,通过获取方法返回值,重定向错误输出,标准输出等.工作.然后通过检查给定路径下的文件,解析出结果,这样通过比较当前版本和之前版本的不同,检查当前版本输出,修改 bug,这样可以重复利用之前人工复查的结果,假设我们当前测试用例有 100 个,新添加了 20 个,这样一共 120 个测试样例,我只需要人工比较这新增 20 个测试样例的结果就可以,剩下已经存在的 100 个测试样例,可以通过和之前版本做比较,如果不同说明本次修改存在问题,因为我们的假设是以往版本的输出是正确的标准的,基于

这样的假设,如果我们比较出不同,就说明本次版本迭代可能你引入 bug,这时只需要核查不同的测试样例,追踪出现的问题.

3.3 动态代理部分的封装

3.3.1 代理和动态代理

代理中分被代理和代理，很多情况下，我们想控制和目标对象的交互,代理有两个基本要素：1）代理持有被代理对象，被代理类是被代理类控制的，2）外部要想和目标对象有交互，必须先经过代理对象。这就是代理模式要完成的目标。这个目标不会凭空实现，下面说明下如何实现这个目标

用一个日志记录例子说明下：

如果我想记录这个类中方法的调用日志，即何时调用了这个类的哪个方法

```
//目标接口即被代理的接口
interface TargetInterface{
    public void target1();
    public void target2();
}
//有一个类实现了这个接口，现在我们要针对这个类的对象进行代理
Class TargetClass implements TargetInterface{
    public void target1(){
    System.out.println("target1 调用");
    }
    public void target2(){
    }
}
```

如果我想获取一个实现了 TargetInterface 接口的 对象，怎么办

```
public TargetInterface getTargetInterface(){
  TargetInterface targetClass = new TargetClass();
  return targetClass;
}
```

于是返回了一个 TargetClass 对象,这个对象实现了 TargetInterface 接口

```
//为了代理它，我必须也得装成 TargetInterface 一样。
//先实现这个接口，但是至于具体请求转发给 TargetClass 类对象实现
Class ProxyClass implements TargetInterface{
    private TargetInterface targetClass;
        public ProxyClass(TargetInterface targetClass){
        this.targetClass = targetClass;
    }
    public void target1(){
        System.out.println("调用了 target1 方法");
        this.targetClass.target1();
    }
    public void target2(){
        this.targetClass.target2();
    }
}
```

现在我要实现代理 TargetClass 对象的目标

```
public TargetInterface getTargetInterface(){
    TargetInterface targetClass = new TargetClass();//被代理对象已
    经出现
    return  proxyClass = new ProxyClass(targetClass);//代理对象也出
    现，同时代理对象将被代理对象控制起来
}
```

通过此方法获取到的 targetInterface 接口实例，已经不是 TargetClass 类型对象了，而是 ProxyInterface 类型对象，但是外部看不到，他们只知道他们获取了一个 TargetInterface 类型实例，而不在乎获取的对象到底是 TargetClass,还是 TargetProxy 类型，但是 ProxyClass 却是代理了 TargetClass.现在 TargetClass 的方法访问都想经过 ProxyClass 这一关。本来是 TargetClass 声明我实现了 TargetInterface,现在 ProxyClass 也声明自己实现了 TargetInterface 解耦,但是它是在获取 TargetClass 对象后，将其封装起来，实现了对代理对象的控制，但是对于外界这一切都是透明的，

因为他们都实现了 TargetInterface 接口。这就是代理：也是静态代理，静态的原因是任何一个接口我们都需要提供一个代理类来实现，但是我们看到这些代码都是非常简单的。先不提动态代理，通过这个例子，我们也可以总结下，

代理模式的弱点：

代理对象通过持有被代理的引用，控制记录被代理方法的调用，但是，代理对象并不知道被代理对象内部实现的逻辑，不清楚被代理对象在方法体中做了什么，所以注定代理对象在其方法体中只能做一些与被代理对象中的逻辑无关的一些操作，如果逻辑有关，那么代理对象如何知道呢？因为不知道所以只能做一些无关的操作，例如，日志记录，权限拦截，访问控制，异常拦截。等通用性功能。也就是说代理模式更多实现的功能点是与业务关联不大的操作。也就是与接口关联不大的操作，那么如何实现对所有接口的代理呢？

3.3.2 动态代理

动态代理可以一次代理许多许多接口，但是前提是这些 接口被代理的原因是相同的，也就是都是为了记录日志，权限控制。

如何实现动态代理

jdk 的实现

jdk 中提供了动态代理的支持，但是功能有限

示例代码如下；

```
class DefaultBindInvocationHandler implements InvocationHandler{
        public Object bind(Object target)
        throws NullPointerException,IllegalArgumentException{
        this.target = (target);//持有被代理对象
/**
*是否实现了接口，如果没有实现接口，jdk 是无法代理的，这个方法的第二个参数
*要求不能为空，所以需要检验下被代理对象是否实现了接口，如果没有实现，那么
*生成对类的代理。对类的代理将在后面细说。
*/
  if(isHaveInterface(target)){
```

```
            return Proxy.newProxyInstance(target.getClass().getClassLoader()
            , target.getClass().getInterfaces(), this);
        }
        else{
            return createProxy(target.getClass());
        }
    }
        public Object invoke(Object proxy, Method method, Object[] args)throws Throwable {
            before(proxy,method,args);
            Object result = method.invoke(target, args);
            after(proxy,method,args);
            return result;
        }
    }
```

我们想要获取 TargetClass 对象的代理怎么办呢？

```
TargetClass targetClassObject = new TargetClassObject();
DefaultBindInvocationHandler proxyHandler = new DefaultBindInvocationHandler();
proxyHandler.bind(targetClassObject);
```

其他需要代理的类只要 要被代理的原因相同，反映到代码就是 invoke 方法的逻辑满足这个类被代理的需要，那么他就可以使用这个类进行代理。invoke 方法在执行目标类方法前，执行了 before,之后执行了 after，用于前置后置增强。如何有需要特殊的增强，额外的增强，可以新增子类重写 before，after，实现增强，同时也可以重写 invoke 方法。但是 jdk 代理存在的遗憾的是当类没有实现任何接口时，jdk 就不能代理了而且 jdk 代理的限制在于，我代理一个对象也许就是拦截对象的所有方法，为什么你要求我一定要实现一个接口呢，如果我要拦截私有方法难道就不行吗？很明显，jdk 代理无法完成这个目标。于是 cglib 给我们提供了这种方便。

cglib 代理

cglib 是专门针对类进行代理的，它通过一种运行时生成字节吗文件，的技术，动态生成类，生成字节吗，实现继承某个类，实现某个接口，最大限度的

实现了动态生成类技术，Spring 基于 cglib 甚至可以获取到方法体上参数的名字。在大

量的框架都使用了 cglib 动态生成类技术。

cglib 生成代理类很容易

```
Enhancer enhancer = new Enhancer();   //启动容器
enhancer.setSuperclass(targetClass);// 设置代理目标，被代理的类的 Class 对象
enhancer.setCallback(this);// 设置回调 ，要求参数为实现了 MethodInterceptor 接口的类
enhancer.setClassLoader(targetClass.getClassLoader());//指定加载器为加载目标类的加载器。（不能指定为加载本类的加载器）
return enhancer.create();///创建一个对象
```

这样一个继承了 targetClass 类型的对象就创建成功了。

3.3.3 Muppet 对代理的封装

一切都应准备就绪，但是如何提供一个工具类，以后更加便捷的操作呢？首先明确一下，我们不能完全抛弃 jdk 代理，毕竟 jdk 代理更加快速一些，而且我们希望我们尽量与 cglib 的 耦合弱一点，最好将用到 cglib 的地方控制在一个静态的值域中，而不能随业务扩展随着扩展。也就是我们要封装动态代理。

1.考虑输入。无怪乎，被代理对象，代理增强器，在 jdk 中是 InvocationHandler，在 cglib 中是 MethodInterceptor。两个接口。

2.代理增强器是随着代理逻辑增多，而增多的，如何包装 cglib 的增强器，使之出现的地方有限而最少呢？

我们是这样设计的。

1）/**

*首先定义绑定代理增强接口，何为绑定代理增强呢，因为我们一般在获取代理对象时都是通过代理增强器类的绑定操作，

*绑定要被代理的对象，然后返回代理对象。不如我们就定义一绑定接口。同时让他继承 InvocationHandler 实现 jdk 代理

*这个接口是为 jdk 动态代理服务的

*

```
interface BindInvocationHandler extends InvocationHandler{


/**
```

```
* 绑定被代理对象，返回代理对象
*/
public Object bind(Object target)throws
NullPointerException,IllegalArgumentException;

/**
*绑定被代理对象，返回代理对象，根据指定的接口返回实现指定接口的代
理
*/
public <T> T bind(Object target,Class<T> targetInterface)throws
NullPointerException,IllegalArgumentException;
}
```

2）定义默认绑定代理增强类，这个类实现了绑定代理增强，和 cglib 代理增强。注定是最复杂的环节，

```
class DefaultBindInvocationHandler implements BindInvocationHandler
,MethodInterceptor{}
```

先明确一下这个类的功能

2.1）实现 InvocationHandler 接口

这个接口只需要实现一个方法

```
public Object invoke(Object proxy, Method method, Object[] args);
```

参数分别为 生成的代理对象，即将执行的方法对象，方法中参数

看看我们实现的方法体

```
before(proxy,method,args,null);//默认是空方法体
Object result = method.invoke(target, args);//执行被代理对象方法
after(proxy,method,args,null);//默认是空方法体
return result;
```

很简单，before，和 after 都是 protected 类型方法，很明显是用来子类 继承，按照模板方法，将行为转移到子类实现，

2.2）实现 MethodIntercept 接口

这个接口也只有一个方法与 invoke 类似

```
@Override
public Object intercept(Object proxy, Method method, Object[] params,
MethodProxy methodProxy) throws Throwable {
    before(proxy,method,params,methodProxy);
    Object  result = methodProxy.invokeSuper(proxy, params);
    after(proxy,method,params,methodProxy);
    return result;
}
```

平白无奇，同样是通过 methodProxy 方法实现被代理对象方法执行。只不过 before 方法和 after 方法第四个参数不为空

2.3）jdk 绑定被代理对象同时返回代理对象

```
public Object bind(Object target)
                throws NullPointerException,IllegalArgumentException{
    this.target = (target);
  if(isHaveInterface(target)){//判断有没有实现接口，如果实现了，那么按
                        照 jdk 动态代理，否则按照 cglib 代理。
      return Proxy.newProxyInstance(target.getClass().getClassLoader()
        , target.getClass().getInterfaces(), this);
  }
  else{
        return createProxy(target.getClass());
  }
}
```

先绑定被代理对象，然后，创建代理对象，返回代理对象，同样平淡无奇

2.4）cglib 绑定被代理对象，同时返回代理对象。

为了区分和 jdk 的区别，方法名改为 getClassProxy，表示是对类的代理，而不是 jdk 的对接口的代理

```
public <T> T getClassProxy(T target){
    this.target = target;
    return  (T) createProxy(target.getClass());
```

```
  }
    private final <T> T createProxy(Class<T> targetClass) {
    Enhancer enhancer = new Enhancer();  //启动容器
    enhancer.setSuperclass(targetClass);// 设置代理目标，被代理的类的
    Class 对象
    enhancer.setCallback(this);// 设置回调  ，要求参数为实现了
    MethodInterceptor 接口的类
    enhancer.setClassLoader(targetClass.getClassLoader());//指定加载
    器为加载目标类的加载器.(不能指定为加载本类的加载器）
    return enhancer.create();///创建一个对象
}
```

这段代码在上面 已经解释过了，即使平淡，也不平淡，不平淡的地方是 this，在设置回调类时是设置 this,如果是本类对象调用，那么 this，当然是本类，也就是代理对象的方法调用都会引起 intercept 方法的调用。但是一旦子类重写了 intercept 方法，那回调类将转移到子类对象，也就是子类的实现的 intercept 方法会被调用，而正是我们希望看到的结果。即我们创建的代理对象随着子类重写 intercept 方法，代理增强也不同

2.5）到这我们可以新增子类，DefaultBindInvocationHandler 做一些前后置增强了，但是我们的需求止于此了吗

首先新增子类，对于子类来说功能实现选择 较多，可以重写 before，after,也可以重写 invoke，但是每次增加一个代理加强

都需要继承一个父类，是不是太过于多余，限制太多。耦合性太强，我们期待什么呢？

像这样定一个代理增强、

```
public class ClosedInvocationHandler implements InvocationHandler{
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        if(Closed.class.isInstance(proxy)){
            Closed close = (Closed)proxy;
```

```
            if(close.hasClosed()){
                throw new IllealInvokeException("已经关闭，没有权限访问
            ");
            }else{
                return method.invoke(proxy, args);
            }
        }
        return method.invoke(proxy, args);
    }
}
```

ReflectUtil.getClassProxy(new A(),new ClosedInvocationHandler());这样我们只需要定义一个实现了 jdk 接口的实现类，实现了与 cglib 解耦，同时屏蔽了 jdk 代理与 cglib 接口的区别。同时不需要代理增强不需要额外定义绑定操作。让客户端不关心代理实现的细节，将关注点放在定义代理增强上。为此我们定义了一个标准绑定增强类

```
class StandardBindInvocationHandler extends
                        DefaultBindInvocationHandler{
private InvocationHandler handler;
public StandardBindInvocationHandler(InvocationHandler handler){
        this.handler = handler;
}
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    before(proxy,method,args,null);
    Object result = handler.invoke(target, method, args);
    after(proxy,method,args,null);
    return result;
}
@Override
public Object intercept(Object proxy, Method method, Object[] params,
```

```
MethodProxy methodProxy) throws Throwable {
    before(proxy,method,params,methodProxy);
    Object object = handler.invoke(target, method, params);
    after(proxy,method,params,methodProxy);
    return object;
    }
}
```

这个类可以看到是包内可见，并不对包外开放，也就是在其他地方定义代理增强完全不需要知道这么一个类的存在。这个类其实本身就是代理模式的应用，代理的接口就是 InvocationHandler 接口，在初始化这个类对象的时候需要指定一个代理增强对象。然后可以看到不论是 jdk 代理，还是 cglib 代理 ，都将请求转发到这个代理对象，但是有一需要注意，在传给代理增强类参数中有一丝变化

```
Object object = handler.invoke(target, method, params);//在 jdk 中，第
```

一个参数是生成的代理对象，但是在我们这里传给他的事被代理对象，也就是用户输入的类。这一点在定义代理增强类时需要注意，我觉得这很好，对于客户端调用，他不关注生成的代理对象是什么玩意，他只关注我被代理的对象什么样，然后指定特定的逻辑 处理。

所以在我们刚才自定的代理增强类中。

```
public class ClosedInvocationHandler implements InvocationHandler{
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        if(Closed.class.isInstance(proxy)){
            Closed close = (Closed)proxy;
            if(close.hasClosed()){
                throw new IllealInvokeException("已经关闭，没有权限
            访问");
            }else{
                return method.invoke(proxy, args);
            }
```

```
            }
             return method.invoke(proxy, args);
    }
}
```

我们可以直接调用method.invoke(proxy,args);在使用原生jdk代理使绝对不可以的，因为这句话的意思 是执行指定对象的这个方法，因为代理对象执行了一个方法，引起invoke调用,在invoke方法中在对代理对象的方法进行调用，那同样会引起invoke方法的调用，正确是调用被代理对象的这个方法，而我们给自定义代理增强传入的参数就是被代理对象。所以很方便使用。

那么其他的黑箱是啥呢，当然是ReflectUtil的getClassProxy方法

```
StandardBindInvocationHandler handler = new
StandardBindInvocationHandler(invocationHandler);
return handler.getClassProxy(target);
```

现在梳理下方法调用

1.创建StandardBindInvocationHandler，指定了代理增强，这个代理增强是我们的特定的业务实现，实现接口为InvocationHandler接口

2.调用standardBindInvocatinoHandler的getClassProxy，StandardBindInvocationHandler本身没有重写，所以还是父类的方法

```
this.target = target;
return  (T)
createProxy(target.getClass(),constructorArgsClazzs,constructorArgs
Values);
```

3。绑定被代理对象，注意这个target是保护类型，也就是子类可以访问，也就是在StandardBingInvocationHandler中的invoke,intercept方法中的target参数就是这个target,而它的赋值就是在这里。

4.然后createProxy方法就是我们之前说的

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(targetClass);// 设置代理目标
```

```
    enhancer.setCallback(this);// 设置回调
    enhancer.setClassLoader(targetClass.getClassLoader());

    return (T)enhancer.create();
```

同样传神的是 this，因此代理对象的默认代理增强是子类对象，也就是代理对象上的方法调用请求 将调用 StandardBingInvocationHandler 类的 intercept 方法，然后。

5.当调用 intercept 时，intercept 会将请求转发给用户自定义的 InvocationHandler 实现上,也就是 ClosedInvocationHandler 类型的代理增强。至此，整个流程结束缘起 ReflectUtil.创建代理类交给了 DefaultBindInvocationHandler 类，回调处理交给了子类的实现，子类的实现又把请求转给了用户的自定义请求。由于采用 jdk 自定义的 InvocationHandler 接口，所以实现了和 cglib 的解耦。jdk 的代理同样一个思路，都是面向对象的简单使用。最后的调用变成这样

```
public class ClosedInvocationHandler implements InvocationHandler{
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        if(Closed.class.isInstance(proxy)){
            Closed close = (Closed)proxy;
            if(close.hasClosed()){
            throw new IllealInvokeException("已经关闭，没有权限访问");
            }else{
            return method.invoke(proxy, args);
            }
        }
        return method.invoke(proxy, args);
    }
}
```

对类代理 ReflectUtil.getClassProxy(new A(),new ClosedInvocationHandler());
对接口代理 ReflectUtil.getProxy(new A(),new ClosedInvocationHandler());
解耦，关注应该关注的地方，使用简洁，是我们追求的目标。

3.4 本章小结

ioc 和 aop 的引入都是有实际的场景需要,ioc 是各模块之间耦合性太强,各模块显式依赖过多,通过 ioc 将对象实例间的依赖关系统一管理起来,除此依赖 ioc 提供的 bean 托管机制显得更强大,Aop 的引入就可以证明,aop 需要依赖于 ioc 存在,需要 ioc 给他提供 bean 的管理,同时 aop 通过动态代理,将一些工作分离出来,同意针对一个切面进行编程,这样分离机制十分有效,而且通过针对 整个切面编程,用最少的时间,最少的代码完成了以前无法想象的任务量,可以说引入了 aop 和 ioc 之后,muppet 需要任何其他工具功能,或者扩展新的功能都相比以前容易很多,可维护性提高,协同开发的难度降低,这都是不小的进步.

第4章

4.1 表序

4.2 本章小结(必须有)

第5章　项目中常见错误实践及改进

5.1　对象属性间的平行依赖

```
Public class Context{
    Private A a;
    Private B b;
}
```

属性 a 和 b 我们认为他们是“平行的”。现在他们没有存在互相依赖

```
Public class Context{
    Private A a = new A();
    Private B b = new B(a);
}
```

现在呢，可以认为 b 平行依赖于 a.这就是对象属性间的平行依赖。这种情形是没有问题的。

```
Public class Context{
    Private A a;
    Private B b = new B(a);//此时 B 如果实例化，那么 a 为 null
 Public Context(A a){
    This.a = a;
}
}
```

错误出现了，属性 b 的实例化要优先于构造方法的，所以 B 接受了一个 null。至于 NullPointerException 什么时候触发，没人能预测，最好是在 B 构造方法中触发，这时你可能还会恍然大悟，传入了一个 null 值，但是如果是在 B 的常规方法调用触发，你可能需要花点时间来排查空指针异常了。

记住：实例属性的实例代码块要优于构造方法执行。

建议：不要依赖 Java 的对象初始化顺序，尽量将属性初始化放到构造方法中。尤其类比较复杂，存在“对象属性的平行依赖”时。但是我们还是喜欢声明属性的同时将其初始化（顺便设置个 final 常量）。毕竟这样编码很爽，可读性也比较高。这一点 C++规范的很好，属性声明时不得初始化，只能在构造函数中初始

化。所以我们经常看见 C++超长的只做了简单赋值操作的构造函数。

5.2 构造函数陷阱

“构造函数陷阱”：构造方法中调用可被重写的方法

```
Public class A{
    Public A(){
    ....//初始化操作
    Function();//
    ....初始化操作
    }
    Public void function(){
    .....
    }
}
```

如果 A 的子类重写了 function 方法，那么 A 类构造方法执行的就是其子类的 function 实现，如果 A 类设计时没有考虑到这种情况，那么 A 的初始化就存在很大风险

所以要将 function 设为 private，或 final

建议：构造方法内不要使用 public 方法，如果使用必须要考虑 子类可能会重写该方法，影响父类的初始化过程。建议将存在子类重写需求的逻辑抽象出一个方法设置为 protected， public 方法设置为 final 调用该 protected 方法。这样 public 对外服务，和 protected 提供重写 比较清晰。

另外记住 private ,final 方法中 调用 public 方法也要想到子类可能会重写这个 public 方法偶。

下面还会讲解构造方法中 执行 public 方法还有多坑。

5.3 构造函数与重载带来的空指针异常

```
Public class Context{
    Public A(){//构造方法
    ....
```

```
    Register();
    ....
    }
    Public void register(){
        beforeRegister();
        .....
        afterRegister();
    }
    Protected void beforeRegister(){
    }
    Protected void afterRegister(){
    }
}
```

现在 Context 需要向外提供注册功能.但是实例化时，需要先注册一些服务。注册操作前后会　执行 before 方法和 after 方法，子类可以重写，向提供扩展注册功能

于是乎，子类重写了 before,after 功能，蛋疼的空指针异常又出现了。

为什么呢？因为

```
Public class ChildContext{
    Private Map<String,Object> map = new HashMap<>();
    Protected void beforeRegister(){
    ......
    Map.put();
    .....
    }
    Protected void afterRegister(){
    ....
    //
    ...
    }
```

}

子类重写了注册的 before 和 after 方法目的是 监听注册功能，维护了额外的数据结构（）。但是正是这个额外的数据结构触发了空指针异常。原因正是父类的构造方法中执行了子类重写的方法,而子类的重写方法中使用了自己的属性就像 beforeRegister()方法中使用了 map 一样。但是 map 还没有初始化呢,还没有初始化呢.

解决思路有好几种：

1. 最简单的思路是将子类中的变量 map 声明为 static，不就先于父类构造方法执行了吗？的确解决了这个 bug，但引入新的 bug，我们提到过慎用 static 变量因为他属于整个类。不单单属于一个对象。那意味着所有的对象实例都共用这个 static map, 这是正确的逻辑吗？你考虑了吗？这个坑我还就犯过...最最重要的是，这个 static map 在该进行垃圾回收时没法被回收。没准哪个时刻出现了 Out of Memory 内存溢出，服务器宕机，然后查到了这个 root 根节点大对象正是 static map.导致不可挽回的但本可避免的过失

２．有一个争议的实现方法

父类构造方法中 先调用 beforeInitialize 同时 beforeInitialize()，afterInitialize()方法供子类重写，这时子类就可以把属性初始化需求放到 beforeInitialize()方法中。实现了父类对子类的依赖，实现子类属性 先于父类初始化但是我们倒转了依赖。破坏了子类，父类初始化顺序。

３．我们可以在 beforeRegister 中 判断属性变量是否为 null，如果为 null 就初始化它。简单粗暴，但是我没有用这种方法。因为我想试试第二种方法到底会有多坑

４．还有好多方法...但是归根结底，我们是在构造方法中调用可重写方法倒置了子父类的依赖，让父类依赖于子类，与 Java 面向对象的设计理念相冲突，才会出现这么多问题。

建议：尽量不要在构造方法中调用可以可被重写的方法 Public,protected 方法尽可能少的出现在构造方法中”构造函数陷阱“：构造方法中调用可被的重写方法

5.4 父类对子类的依赖

面向对象的设计中，子类可以访问到父类方法，属性，但是父类无法访问子

类属性，本应该是子类依赖于父类。但是 设计模式中模板方法 的思路带来了父类对子类的依赖。（模板方法：父类的方法中调用了方法 A,但是方法 A 由子类具体实现）模板设计虽然由父类定义了逻辑处理的流程，子类只是填空。但是父类的处理逻辑中包括了不可预测的子类实现。模板方法要求父类充分考虑子类可能的具体实现，考虑哪种实现是正确的，符合要求的。模板的设计需要高度的抽象。之前已经谈到了，构造方法中尽量不要使用模板方法的设计。

另外：充分考虑子类通过重写对父类处理逻辑的干扰，经常使用 private 和 final

5.5 静态属性.静态方法带来的麻烦

静态属性属于对象共有，静态方法通过类名即可调用，有何麻烦？

例子：

封装数据库配置获取数据库连接。我们使用了 dbcp 连接池，需要提供 url.password,username 等等...于是将 username,password,url 等写死在 DataBaseUtil 类的属性中。同时提供 getConnection 静态方法。

DataBaseUtil.getConnection()，可以在任何地方随时调用。用起来很爽。

需求变更一：

将配置信息转移到 xml 中。

现在 DataBaseUtil.getConnection()变更。需要传递 xml 文件路径。但是 DatabaseUtil 调用太多，需改需要改动地方太多。

临时方案:

在程序启动时，设置 xml 路径到 DataBaseUtil 静态属性中。这样保证所有的 getConnection 调用之前都设置了 xml

需求变更二：

数据库配置再次变动，需要配置多数据源，例如主从配置等。

我们需要通过数据源 key 获取数据库连接.

所以我们不得不提供 DataBaseUtil.getConnection(“main”);

当然，我们还可以提供其他蹩脚方案，DataBaseUtil.getConnection()方法获取默认数据源连接。

分析一下.DataBaseUtil.getConnection 的主要逻辑：

1．负责 xml 文件读取.解析.配置数据源数据库连接池

2．获取默认数据源连接

DataBaseUtil 的逻辑由简单的属性配置,到 xml 配置管理,到后来的多数据源配置管理....

DataBaseUtil 已经远远地脱离了当初设计它的初衷...面对需求的变更，DataBaseUtil 的应对很保守很被动。而我认为造成无能为力的原因，就是 DataBaseUtil 提供了 getConnection 静态方法.

假设我们从一开始将 DataBaseUtil.getConnection 改为实例方法，同时 DataBaseUtil 构造方法参数列表要求传入 xml 路径，完成 xml 读取解析，以及多数据源管理。

例如获取主数据源，我们只需要在构造方法中提供参数，即设置当前 DataBaseUtil 对象为主数据源。通过 DataBaseUtil 实例对象的任何 getConnection 调用前都已经完成 xml 解析，数据源管理。

getConnection 不需要再关心 xml,多数据源这些过程。

DataBaseUtil 实例的创建可以由工厂负责，在任何需要获取数据库链接的地方，我们都可以传递 DataBaseUtil 实例引用，使用者不需要了解实例的创建过程。

就像《Clean Code》（《代码整洁之道》）作者 Martin 提到过的：

“软件系统应将启动过程和启动过程之后的运行逻辑分离开，在启动过程中构建应用对象，配置内部模块互相依赖的关系。”

显然静态的 getConnection 将启动过程和启动后的运行逻辑混在了一起，系统维护难度增加，当把 DataBaseUtil.getConnection 设置为实例调用,我们面对需求变化更加自由，将对象实例的构建过程和运行时逻辑分离开。在对象构造时完成复杂的配置逻辑。那么启动后的方法调用更加的轻松，关注点更少。

关于静态方法的反思：

之前提到，静态方法通过类名.方法名即可访问。这只是表面上的简洁，更深层次上，静态方法意味：它没有“状态”可言，客户端不需要关注他的状态，你随时可以访问，它突破了对象实例构造方法这唯一入口，它可以在任何该类对象实例初始化前对外提供服务。

如果一个类包含了 n 个静态方法，他就相当于提供了 n+1 个入口，类的设计者除

了需要考虑构造方法这一入口之外，还需要考虑是否需要在静态方法中完成 额外的系统准备工作。就像 DataBaseUtil.getConnection 还需要完成 xml,数据源管理，然后才能返回数据库连接。

此时静态方法给类引入了更大的复杂性。

关于静态方法使用的建议

1).不要因为静态方法通过类名调用的方便，就优先考虑静态方法。

2)静态方法应侧重完成工具功能，不应处理 复杂逻辑，复杂的处理流程。当静态方法中不建议访问系统业务资源，例如 xml 配置读取.

3)静态方法应该侧重于处理，参数列表中的参数.例如 function(A a, B b);静态方法应该针对参数 a,b 作处理。

4)以上几个例子证明，静态方法相比实例方法扩展性弱。如果静态方法实现依赖的数据结构变化频繁，不建议使用静态方法。

5)...使用静态方法时，多问自己一句，它与系统其他部分有关吗？他是否与其他模块存在明显边界？

5.6 太多的无状态类

无状态类：我们可以把对象的属性视为内部状态，当一个对象没有属性，或者属性不可修改只读，我们可以把它视为无状态类。例如常见的 service，dao,事件处理 action 等等

我们习惯了无状态类，并托管给 ioc 管理。很多时候我们设计时更倾向于设计无状态类，但是不经意间自己抗下太多的包袱

错误示例：

需求：将一个 List<Map>对象持久化存储时时，我们抽象了接口

```
Interface XXXStorage{
Public void store(List<Map> ,  String path);
Public void resolve(String path);
}
```

于是，我们设计了 XmlXXXStorage,SerializeXXXStorage 具体实现。提供了 xml,和对象序列化方法存储。对于这两个类而言

由于两个参数，满足了 xml 和序列化存储的所有客户端输入要求.所以不

需要额外的状态，我们将其设计成单例类。并沾沾自喜，我们减少实例化对象次数，提高了性能

需求变更：

现在需要额外的存储方式，存储到数据库里

请问 String path 作何解？对于数据库存储有何意义。[非洲凝视]

一直都说，实现依赖于抽象，不得让抽象依赖于具体实现。

XXXStorage 接口中 store 参数列表过于具体.只需要提供 List<Map>,只需要告诉接口，我要存储什么东西即可，至于具体存储格式以及位置由具体实现类决定即可。

具体实现类需要的参数我们可以通过构造方法参数，或者 setter 方式传入。

避免让接口中的方法接受过多参数，避免具体实现的逻辑影响接口的设计。

我们还是通过将启动过程和启动后的运行逻辑分离开。这样具体实现类的设计有了更多的自由。但也因此不能将实现类设计成无状态的。

反思：

无状态类的设计总是简单的，线程安全的。但是普通的方法调用可能需要过多的参数。同时也干扰了我们接口的设计。

善于利用构造方法，可以简化我们的方法设计。

5.7 异常处理总结

当你在定义方法时，应该保证此方法的正确性，但是你不能保证你的方法的调用者，他们的输入数据，或者 使用环境的正确性，当客户端代码调用此方法时，我们应该向调用方声明可能出现的异常,这样当出现此种异常时，通过方法的文档，他们就清楚到底发生了什么，自己为什么错，合理的异常反馈是 代码库作者，代码库调用者 之间信息交流的通道，或者说是，代码库作者推卸责任证明自己没有错的挡箭牌

当你在开发初期，设计代码时，可能考虑不清楚可能出现什么异常（然后你没有定义，或者仅仅定义一两个），于是洋洋洒洒写了几周后，发现我需要给某某个方法声明抛出异常，因为用户的输入，太离奇古怪，或者这段代码依赖的环境太不稳定了，需要调用方捕获异常，并做异常处理，这时你抛出了新的异常，改了代码，发现项目中多了很多的编译错误，因为调用方并没有对异常做处理，这时你要求你的调用方作者改代码？当然答案在大部分情况下是否定的。

以上两种情况仅仅是开发过程中最常见的情景中的两个，还有很多，总之合理的异常声明是 逻辑分层，代码分工，沟通交流的有效保证。 我需要另外说明的是，当你的代码总是变动，总需要声明很多异常的时候，（三个异常就很多），你首先应该考虑是不是应该对这个方法进行重构了（提取方法，或者提取类），因为频繁的变动代码很大意义上说明，这个方法变动的元素太多，应该分离，或者 把变化部分提取出去。这才是正确合理的解决方案。而不是向用户抛出一堆异常。

那么我们如何向用户正确反馈异常呢？

首先我们们可以从 JDK 的设计学会一些，JDK 中所有的异常都继承自 Exception（除了 Throwable），当我们向上层（调用者）提供服务时，一个包应该是高内聚的，一个包下的类合作 完成一项大的任务，然后我们在这一项任务中抽象出来一个异常基类，例如 muppet 其中一个包是 cn.bronzeware.muppet.sqlgenerate 这个包专门负责生成 sql 语句的，那么你就可以定义一个异常基类 SqlGenerateException，那么以后这个包下的类，给包外调用方提供的核心方法抛出的异常都应该 SqlGenerateException 类型，我们可以把这个类声明为抽象异常基类

```
public class SqlGenerateException extends Exception{
        @Override
        public final String getMessage() {
            return "{Sql 语句生成出错->"+Message();
        }
        public abstract String Message();
}
```

我们将 getMessage 方法设为 final 不能重写,同时其子类必须重写 Message（）方法，我们在 getMessage 方法中有一个默认字符串叫做 “sql 语句生成出错”,这个字符串可以放关于异常基类的描述，然后这个字符串在加上 子类的 Message 方法利用模板方法设计模式,实现基类错误信息和子类错误信息的整合。

然后子类应该怎么写呢，

```
public class SelectSqlGenerateException extends SqlGenerateException{
        public SelectSqlGenerateException(String message){
```

```
        this.message = message;
    }
    public SelectSqlGenerateException(){
    }
    private String message = "";
    public String message(){
        return "Select 语句生成出错->"+message+"}";
    }
}
```

当在 SelectGenerate 生成 Select 语句时，如果出现关于生成语句的异常可以抛出 SelectGenerateException("这里在进一步指出 Select 语句具体出现什么异常" 这样当 SelectGenerate 类的调用者在获取到 SelectGenerateGenerate 异常时通过 getMessage()返回的字符串 是"{Sql 语句出错-> Select 语句出错> Select 语句定义错误，缺少逗号}" （举个例子） 那么这时，SelectGenerate 的调用者其实获取的异常类是 SelectSqlGenerateException 类型，它可以通过这种手段

获得具体的异常处理

```
catch (SqlGenerateException e) {
    if(e instanceof SelectSqlGenerateException){
        //做具体的处理
    }else if/其他处理
}
```

如果它需要针对具体的异常类型做处理，就可以用上述类型检查做特殊处理，否则如果不需要做特殊处理，再向上抛出

当有另外一个包下的类或者其他层次(Sql 语句层之上)的类 调用 SqlGenerate 层（sql 语句生成层）的服务时，他们本身也有一个主要的功能任务，这个功能任务，也需要与之对应的异常信息抽象基类，这个基类同样这样设计 muppet 中调用 Sql 语句层的包是 cn.bronzeware.muppet.context，是 Context 层，这个层负责数据操作，获取实体类信息（注解处理层），生成 Sql 操作 描述，生成 sql 语句，调用 JDBC 进行数据操作，封装结果集，主要是这几个功能，

他的异常基类是 ContextException()和 SqlGenerate 同样的设计，作为抽象基类 有一个模板方法 message(),其子类 SelectContextException 同样这样设计

```
public class ContextException extends Exception{
      private static final long serialVersionUID =
-2272596789720418995L;
      @Override
      public final String getMessage() {
          return "{数据操作 Context 层出错->"+Message();
      }
      public abstract String Message();
  }
  public class SelectContextException extends ContextException{
    public SelectContextException(String message){
        this.message = message;
    }
    public SelectContextException(){
    }
    private String message = "";
    public String message(){
        return "Select 操作出错->"+message+"}";
    }
}
```

当 SelectContext 中出现异常时，你可以抛出 SelectContext 异常，throw new SelectContextException("具体的 Select 操作可能出现的异常");此时可能出现什么情况，即 SelectContext 中出现的异常是因为 SqlGenerate 层出现了异常这时你应该捕获 SqlGenerateException 异常，捕获到这个异常后，可以这样操作

```
} catch (SqlGenerateException e) {
      throw new SelectContextException(e.getMessage());
}
```

应该是 "{数据操作 Context 层出错->Select 操作出错->{Sql 语句出错-> Select 语句出错> Select 语句定义错误，缺少逗号 }}"这样上一层知道了原来 SelectContext 层出错是因为 Sql 语句生成的错误，同时又是因为 Select 语句定义错误，缺少逗号这样的错误信息应该是很详细的，更详细的信息，需要你在合适的正确的地方,提供更详细的异常描述，然后做异常抛出，捕获，再抛出，再捕获等等这样,在长长的一套语句的最后(可以调整在最前,读者可以想一想)，总会有错误信息的最准确描述当然你依然可以声明第四级异常子类，（Exceptinon ,ContextException,SelectContextException）但是我认为三级足够了，首先，ContextException 中有最基本的描述，SelectContextException 中还会有描述，你还可以通过 SelectContextException 的构造方法在传入更具体的错误信息，我觉得足够了，当然你仍然有需求的话，考虑下是不是有必要在划分一个包，或者再声明一个与之前的包内的基类独立的另外一个包内基类。然后再做决定不迟。

5.8 本章小结

本章主要介绍了,在实践过程中所遇到的面向对象的问题,同时介绍了所进行的优化,改进措施,不能小瞧这些问题,例如构造方法中父类对于子类的依赖实际上说明的是不正确的抽象级别,当项目中充斥着这种不良的设计,代码的味道就会变,后续将会更难以扩展.同时为了简化接口设计难度,我们有时需要设计更多的状态类,将抽象的细节参数放在构造方法上.将通用参数放在接口中.做到面向抽象编程.

第6章　模板使用简单说明

6.1　内容编写

6.2　本章小结

结论

[单击此处输入结论内容(宋体、小四号字，行距最小值 22 磅)]

结论作为单独一章排列，但不加章号。

结论是对整个论文主要成果的归纳，要突出设计（论文）的创新点，以简练的文字对论文的主要工作进行评价，一般为 400～1000 字。

参考文献

[单击此处输入参考文献内容（宋体、小四号字，行距最小值 22 磅）]

参考文献是论文不可缺少的组成部分，它反映了论文的取材来源和广博程度。

注重引用近期发表的与论文工作直接有关的学术期刊类文献；

一般应在 15 篇以上，其中学术期刊类文献不少于 8 篇，外文文献不少于 3 篇，近三年的文献不少于 5 篇；

在论文正文中必须有参考文献的编号，参考文献的序号应按在正文中出现的顺序排列。

产品说明书、各类标准、各种报纸上刊登的文章及未公开发表的研究报告（著名的内部报告如 PB、AD 报告及著名大公司的企业技术报告等除外）不宜做为参考文献引用。但对于工程设计类论文，各种标准、规范和手册可作为参考文献。

引用网上参考文献时，应注明该文献的准确网页地址，网上参考文献不包含在上述规定的文献数量之内。

参考文献示例：

[1] 张国云. 计算机视觉与图像识别[M]. 北京:科学出版社, 2012，121-153.

[2] 何云峰, 周玲, 于俊清等. 基于局部特征聚合的图像检索方法[J]. 计算机学报, 2011,34(11):2224-2233.

[3] 周敬跃, 李伟文. 利用基元叶片理论单级跨音速轴流压气机特性[C]. 见: 中国工程热物理学术讨论会. 北京:工程热物理研究所, 1985,181-196

[4] 金宏. 导航系统的精度及容错性能的研究[D]. 北京：北京航空航天大学自动控制学科博士学位论文,1998:60-63.

[5] John K T, George S A. Alloy and micro structural design . London: Academic press Inc. LTD.1993，12(5):236-238

致谢

[单击此处输入致谢内容（宋体、小四号字,行距最小值 22 磅）]

对导师和给予指导或协助完成论文工作的组织和个人表示感谢。内容应简洁明了、实事求是，避免俗套。

附录 1 开题报告

[单击此处输入附录 1 的内容（宋体、小四号字,行距最小值 22 磅）]

开题报告不要出现签字和成绩等内容，只要正文。

附录 2 文献综述

[单击此处输入附录 2 的内容（宋体、小四号字,行距最小值 22 磅）]

文献综述不要出现签字等内容，只要正文。

附录 3 中期报告

[单击此处输入附录 3 的内容（宋体、小四号字,行距最小值 22 磅）]

中期报告不要出现签字和成绩等内容，只要正文。

附录 4 外文原文

[单击此处输入附录 4 的内容（宋体、小四号字,行距最小值 22 磅）]

附录 5 外文翻译

[单击此处输入附录 5 的内容（宋体、小四号字,行距最小值 22 磅）]

燕山大学毕业论文开题和中期考核评分表

开题考核：

开题考核（满分 20 分）			
社会发展背景的认识和选题意义（5 分）	国内外研究现状分析（5 分）	具备软件需求分析和技术选型的能力（10 分）	开题成绩

中期考核：

中期考核（30 分）			
软件设计方案的质量（15 分）	问题研究能力的考核（10 分）	项目管理能力的考核（5 分）	中期成绩

燕山大学毕业论文评审和答辩评分表

指导教师评分：

导师评分（满分 10 分）		
项目过程的管理意识（5 分）	自我学习意识和工作态度（5 分）	导师评分汇总

指导教师签字：_____ 年 月 日

毕业（设计）论文答辩：

答辩考核（满分 40 分）						
功能实现的完备度和性能达标情况（10 分）	软件测试能力（5 分）	创新和发展意识（5 分）	工作总结和综合表达能力（10 分）	报告和论文撰写质量（5 分）	外文资料阅读与翻译（5 分）	答辩成绩

答辩组组长签字：_____ 年 月 日

综合其开题考核、中期考核、导师评分、评阅评分、答辩成绩，

该本科生毕业（设计）论文的总成绩为：_____ 。

（☐优秀、☐良好、☐中等、☐及格、☐不及格）