# Spring核心知识

Spring是一个开源框架，Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为J2EE应用程序开发提供集成的框架。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是控制反转(IoC)和面向切面(AOP)。简单来说，Spring是一个分层的JavaSE/EEfull-stack(一站式)轻量级开源框架。

为什么说Spring是一个一站式的轻量级开源框架呢？EE开发可分成三层架构，针对JavaEE的三层结构，每一层Spring都提供了不同的解决技术。

• WEB层：SpringMVC

• 业务层：Spring的IoC

• 持久层：Spring的JDBCTemplate(Spring的JDBC模板，ORM模板用于整合其他的持久层框架)

从上面的简要介绍中，我们要知道Spring的核心有两部分：

• IoC：控制反转。

举例来说，在之前的操作中，比方说有一个类，我们想要调用类里面的方法(不是静态方法)，就要创建类的对象，使用对象调用方法实现。对于Spring来说，Spring创建对象的过程，不是在代码里面实现的，而是交给Spring来进行配置实现的。

AOP：面向切面编程。

# SpringAOP原理

## AOP编程技术

### 什么是AOP编程

AOP: Aspect Oriented Programming 面向切面编程。  
　　面向切面编程(也叫面向方面)：Aspect Oriented Programming(AOP),是目前软件开发中的一个热点。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。  
　　AOP是OOP的延续，是（Aspect Oriented Programming）的缩写，意思是面向切面（方面）编程。  
　　主要的功能是：**日志记录，性能统计，安全控制，事务处理，异常处理**等等。  
　　主要的意图是：将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改  变这些行为的时候不影响业务逻辑的代码。

　　可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP实际是GoF设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP可以说也是这种目标的一种实现。

假设把应用程序想成一个立体结构的话，OOP的利刃是纵向切入系统，把系统划分为很多个模块（如：用户模块，文章模块等等），而AOP的利刃是横向切入系统，提取各个模块可能都要重复操作的部分（如：权限检查，日志记录等等）。由此可见，AOP是OOP的一个有效补充。

注意：AOP不是一种技术，实际上是编程思想。凡是符合AOP思想的技术，都可以看成是AOP的实现。

**Aop，** aspect object programming 面向切面编程

功能： 让关注点代码与业务代码分离！

#### 关注点

关注点,重复代码就叫做关注点；

#### 切面

关注点形成的类，就叫切面(类)！

面向切面编程，就是指 对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

#### 切入点

执行目标对象方法，动态植入切面代码。

可以通过切入点表达式，指定拦截哪些类的哪些方法； 给指定的类在运行的时候植入切面类代码。

### AOP底层实现原理

#### 代理设计模式

##### 什么是代理模式

通过代理控制对象的访问,可以详细访问某个对象的方法，在这个方法调用处理，或调用后处理。既(AOP微实现) ,AOP核心技术面向切面编程。



##### 代理模式应用场景

SpringAOP、事物原理、日志打印、权限控制、远程调用、安全代理 可以隐蔽真实角色

##### 代理的分类

静态代理(静态定义代理类)

动态代理(动态生成代理类)

Jdk自带动态代理

Cglib 、javaassist（字节码操作库）

##### 静态代理

###### 什么是静态代理

由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了。

###### 静态代理代码

|  |
| --- |
| **publicinterface** IUserDao {  **void** save();  }  **publicclass** UserDao **implements** IUserDao {  **publicvoid** save() {  System.***out***.println("已经保存数据...");  }  }  代理类  **publicclass** UserDaoProxy**implements** IUserDao {  **private** IUserDao target;  **public** UserDaoProxy(IUserDao iuserDao) {  **this**.target = iuserDao;  }  **publicvoid** save() {  System.***out***.println("开启事物...");  target.save();  System.***out***.println("关闭事物...");  }  } |

##### 动态代理

###### 什么是动态代理

1.代理对象,不需要实现接口

2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)

3.动态代理也叫做:JDK代理,接口代理

###### JDK动态代理

1)原理：是根据类加载器和接口创建代理类（此代理类是接口的实现类，所以必须使用接口 面向接口生成代理，位于java.lang.reflect包下）

2)实现方式：

1. 通过实现InvocationHandler接口创建自己的调用处理器 IvocationHandler handler = new InvocationHandlerImpl(…);

2. 通过为Proxy类指定ClassLoader对象和一组interface创建动态代理类Class clazz = Proxy.getProxyClass(classLoader,new Class[]{…});

3. 通过反射机制获取动态代理类的构造函数，其参数类型是调用处理器接口类型Constructor constructor = clazz.getConstructor(new Class[]{InvocationHandler.class});

4. 通过构造函数创建代理类实例，此时需将调用处理器对象作为参数被传入Interface Proxy = (Interface)constructor.newInstance(new Object[] (handler));

**缺点：jdk动态代理，必须是面向接口，目标业务类必须实现接口**

|  |
| --- |
| // 每次生成动态代理类对象时,实现了InvocationHandler接口的调用处理器对象  **publicclass** InvocationHandlerImpl **implements** InvocationHandler {  **private** Object target;// 这其实业务实现类对象，用来调用具体的业务方法  // 通过构造函数传入目标对象  **public** InvocationHandlerImpl(Object target) {  **this**.target = target;  }  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  Object result = **null**;  System.***out***.println("调用开始处理");  result = method.invoke(target, args);  System.***out***.println("调用结束处理");  **return**result;  }  **publicstaticvoid** main(String[] args) **throws** NoSuchMethodException, SecurityException, InstantiationException,  IllegalAccessException, IllegalArgumentException, InvocationTargetException {  // 被代理对象  IUserDao userDao = **new** UserDao();  InvocationHandlerImpl invocationHandlerImpl = **new** InvocationHandlerImpl(userDao);  ClassLoader loader = userDao.getClass().getClassLoader();  Class<?>[] interfaces = userDao.getClass().getInterfaces();  // 主要装载器、一组接口及调用处理动态代理实例  IUserDao newProxyInstance = (IUserDao) Proxy.*newProxyInstance*(loader, interfaces, invocationHandlerImpl);  newProxyInstance.save();  }  } |

###### CGLIB动态代理

原理：利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

#### 什么是CGLIB动态代理

使用cglib[Code Generation Library]实现动态代理，并不要求委托类必须实现接口，底层采用asm字节码生成框架生成代理类的字节码

#### CGLIB动态代理相关代码

|  |
| --- |
| **publicclass** CglibProxy **implements** MethodInterceptor {  **private**ObjecttargetObject;  // 这里的目标类型为Object，则可以接受任意一种参数作为被代理类，实现了动态代理  **public**Object getInstance(Objecttarget) {  // 设置需要创建子类的类  **this**.targetObject = target;  Enhancer enhancer = **new** Enhancer();  enhancer.setSuperclass(target.getClass());  enhancer.setCallback(**this**);  **return**enhancer.create();  }  **public**Object intercept(Objectobj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {  System.***out***.println("开启事物");  Objectresult = proxy.invoke(targetObject, args);  System.***out***.println("关闭事物");  // 返回代理对象  **return**result;  }  **publicstaticvoid** main(String[] args) {  CglibProxy cglibProxy = **new** CglibProxy();  UserDao userDao = (UserDao) cglibProxy.getInstance(**new** UserDao());  userDao.save();  }  } |

#### CGLIB动态代理与JDK动态区别

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

Spring中。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP

2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

JDK动态代理只能对实现了接口的类生成代理，而不能针对类 。  
CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法 。  
因为是继承，所以该类或方法最好不要声明成final ，final可以阻止继承和多态。

### AOP编程使用

#### 注解版本实现AOP

|  |
| --- |
| <aop:aspectj-autoproxy></aop:aspectj-autoproxy>开启事物注解权限  @Aspect 指定一个类为切面类  @Pointcut("execution(\* com.itmayiedu.service.UserService.add(..))") 指定切入点表达式  @Before("pointCut\_()") 前置通知: 目标方法之前执行  @After("pointCut\_()") 后置通知：目标方法之后执行（始终执行）  @AfterReturning("pointCut\_()") 返回后通知：执行方法结束前执行(异常不执行)  @AfterThrowing("pointCut\_()") 异常通知: 出现异常时候执行  @Around("pointCut\_()") 环绕通知：环绕目标方法执行  @Component  @Aspect  **publicclass** AopLog {  // 前置通知  @Before("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** begin() {  System.***out***.println("前置通知");  }  //  // 后置通知  @After("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** commit() {  System.***out***.println("后置通知");  }  // 运行通知  @AfterReturning("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** returning() {  System.***out***.println("运行通知");  }  // 异常通知  @AfterThrowing("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** afterThrowing() {  System.***out***.println("异常通知");  }  // 环绕通知  @Around("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  System.***out***.println("环绕通知开始");  proceedingJoinPoint.proceed();  System.***out***.println("环绕通知结束");  }  } |

#### XML方式实现AOP

|  |
| --- |
| Xml实现aop编程：  1）引入jar文件【aop 相关jar， 4个】  2）引入aop名称空间  3）aop 配置  \* 配置切面类（重复执行代码形成的类）  \* aop配置  拦截哪些方法 / 拦截到方法后应用通知代码  <beansxmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>  <!-- dao实例 -->  <beanid=*"userService"*class=*"com.itmayiedu.service.UserService"*></bean>  <!-- 切面类 -->  <beanid=*"aop"*class=*"com.itmayiedu.aop2.AopLog2"*></bean>  <!-- Aop配置 -->  <aop:config>  <!-- 定义一个切入点表达式：拦截哪些方法 -->  <aop:pointcutexpression=*"execution(\* com.itmayiedu.service.UserService.\*(..))"*  id=*"pt"*/>  <!-- 切面 -->  <aop:aspectref=*"aop"*>  <!-- 环绕通知 -->  <aop:aroundmethod=*"around"*pointcut-ref=*"pt"*/>  <!-- 前置通知：在目标方法调用前执行 -->  <aop:beforemethod=*"begin"*pointcut-ref=*"pt"*/>  <!-- 后置通知： -->  <aop:aftermethod=*"after"*pointcut-ref=*"pt"*/>  <!-- 返回后通知 -->  <aop:after-returningmethod=*"afterReturning"*  pointcut-ref=*"pt"*/>  <!-- 异常通知 -->  <aop:after-throwingmethod=*"afterThrowing"*  pointcut-ref=*"pt"*/>  </aop:aspect>  </aop:config>  </beans>  **publicclass** AopLog2 {  // 前置通知  **publicvoid** begin() {  System.***out***.println("前置通知");  }  //  // 后置通知  **publicvoid** commit() {  System.***out***.println("后置通知");  }  // 运行通知  **publicvoid** returning() {  System.***out***.println("运行通知");  }  // 异常通知  **publicvoid** afterThrowing() {  System.***out***.println("异常通知");  }  // 环绕通知  **publicvoid** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  System.***out***.println("环绕通知开始");  proceedingJoinPoint.proceed();  System.***out***.println("环绕通知结束");  }  } |

### AOP编程应用场景

日志记录，性能统计，安全控制，事务处理，异常处理

## Spring事务使用

### 事务基本特性

#### 原子性（Atomicity）

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

#### 一致性（Consistency）

　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

#### 隔离性（Isolation）

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

　　关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

#### 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

### 事务控制分类

#### 编程式事务控制

自己手动控制事务，就叫做编程式事务控制。

Jdbc代码：

Conn.setAutoCommite(false); // 设置手动控制事务

Hibernate代码：

Session.beginTransaction(); // 开启一个事务

【细粒度的事务控制： 可以对指定的方法、指定的方法的某几行添加事务控制】

(比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚.)

#### 声明式事务控制

Spring提供了对事务的管理, 这个就叫声明式事务管理。

Spring提供了对事务控制的实现。用户如果想用Spring的声明式事务管理，只需要在配置文件中配置即可； 不想使用时直接移除配置。这个实现了对事务控制的最大程度的解耦。

Spring声明式事务管理，核心实现就是基于Aop。

【粗粒度的事务控制： 只能给整个方法应用事务，不可以对方法的某几行应用事务。】

(因为aop拦截的是方法。)

Spring声明式事务管理器类：

Jdbc技术：DataSourceTransactionManager

Hibernate技术：HibernateTransactionManager

# 手写Spring事务框架

## 编程事务实现

### 概述

所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，spring推荐使用TransactionTemplate。

### 案例

#### 使用编程事务实现手动事务

使用编程事务实现，手动事务 begin、commit、rollback

|  |
| --- |
| @Component  **publicclass** TransactionUtils {  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  // 开启事务  **public** TransactionStatus begin() {  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionAttribute());  **return**transaction;  }  // 提交事务  **publicvoid** commit(TransactionStatus transactionStatus) {  dataSourceTransactionManager.commit(transactionStatus);  }  // 回滚事务  **publicvoid**rollback(TransactionStatus transactionStatus) {  dataSourceTransactionManager.rollback(transactionStatus);  }  }  @Service  **publicclass** UserService {  @Autowired  **private** UserDao userDao;  @Autowired  **private** TransactionUtils transactionUtils;  **publicvoid** add() {  TransactionStatus transactionStatus = **null**;  **try** {  transactionStatus = transactionUtils.begin();  userDao.add("wangmazi", 27);  **int**i = 1 / 0;  System.***out***.println("我是add方法");  userDao.add("zhangsan", 16);  transactionUtils.commit(transactionStatus);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (transactionStatus != **null**) {  transactionStatus.rollbackToSavepoint(transactionStatus);  }  }  }  } |

#### AOP技术封装手动事务

|  |
| --- |
| @Component  @Aspect  **publicclass** AopTransaction {  @Autowired  **private** TransactionUtils transactionUtils;  // // 异常通知  @AfterThrowing("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** afterThrowing() {  System.***out***.println("程序已经回滚");  // 获取程序当前事务进行回滚  TransactionAspectSupport.*currentTransactionStatus*().setRollbackOnly();  }  // 环绕通知  @Around("execution(\* com.itmayiedu.service.UserService.add(..))")  **publicvoid** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  System.***out***.println("开启事务");  TransactionStatus begin = transactionUtils.begin();  proceedingJoinPoint.proceed();  transactionUtils.commit(begin);  System.***out***.println("提交事务");  }  } |

#### 使用事务注意事项

事务是程序运行如果没有错误,会自动提交事物,如果程序运行发生异常,则会自动回滚。

如果使用了try捕获异常时.一定要在catch里面手动回滚。

事务手动回滚代码

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

## 声明事务实现

### 概述

管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。

显然声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式。

声明式事务管理使业务代码不受污染，一个普通的POJO对象，只要加上注解就可以获得完全的事务支持。和编程式事务相比，声明式事务唯一不足地方是，后者的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。但是即便有这样的需求，也存在很多变通的方法，比如，可以将需要进行事务管理的代码块独立为方法等等。

### XML实现声明

### 注解版本声明

|  |
| --- |
| <beansxmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>  <!-- 开启注解 -->  <context:component-scanbase-package=*"com.itmayiedu"*></context:component-scan>  <!-- 1. 数据源对象: C3P0连接池 -->  <beanid=*"dataSource"*class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <propertyname=*"driverClass"*value=*"com.mysql.jdbc.Driver"*></property>  <propertyname=*"jdbcUrl"*value=*"jdbc:mysql://localhost:3306/test"*></property>  <propertyname=*"user"*value=*"root"*></property>  <propertyname=*"password"* value=*"root"*></property>  </bean>  <!-- 2. JdbcTemplate工具类实例 -->  <beanid=*"jdbcTemplate"*class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <propertyname=*"dataSource"*ref=*"dataSource"*></property>  </bean>  <!-- 配置事物 -->  <beanid=*"dataSourceTransactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <propertyname=*"dataSource"*ref=*"dataSource"*></property>  </bean>  <!-- 开启注解事物 -->  <tx:annotation-driventransaction-manager=*"dataSourceTransactionManager"*/>  </beans> |

用法

|  |
| --- |
| @Transactional  **publicvoid** add() {  userDao.add("wangmazi", 27);  **int**i = 1 / 0;  System.***out***.println("我是add方法");  userDao.add("zhangsan", 16);  } |

## 手写Spring注解版本事务

### 注解

Jdk1.5新增新技术，注解。很多框架为了简化代码，都会提供有些注解。可以理解为插件，是代码级别的插件，在类的方法上写：@XXX，就是在代码上插入了一个插件。

注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。

**注解分类：内置注解(也成为元注解jdk 自带注解)、自定义注解（Spring框架）**

##### 什么是内置注解

（1） @SuppressWarnings   再程序前面加上可以在javac编译中去除警告--阶段是SOURCE  
（2） @Deprecated   带有标记的包，方法，字段说明其过时----阶段是SOURCE  
（3）@Overricle   打上这个标记说明该方法是将父类的方法重写--阶段是SOURCE

###### @Overricle 案例演示

|  |
| --- |
| **@Override**  **public String toString() {**  **returnnull;**  **}** |

###### @Deprecated案例演示

|  |
| --- |
| **new Date().parse("");** |

###### @SuppressWarnings  案例演示

|  |
| --- |
| @SuppressWarnings({ "all" })  **publicvoid** save() {  java.util.List list = **new** ArrayList();  } |

##### 实现自定义注解

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：  
[@Target](mailto:1.@Target)

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

1. CONSTRUCTOR:用于描述构造器
2. FIELD:用于描述域
3. LOCAL\_VARIABLE:用于描述局部变量
4. METHOD:用于描述方法
5. PACKAGE:用于描述包
6. PARAMETER:用于描述参数
7. TYPE:用于描述类、接口(包括注解类型) 或enum声明

[**2.@Retention**](mailto:2.@Retention)

表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）  
**3.@Documented  
4.@Inherited**

使用@interface 定义注解。

|  |
| --- |
| @Target(value = { ElementType.***METHOD*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public@interface**AddAnnotation {  **int** userId() **default**  **t** 0;  String userName() **default**"默认名称";  String[]arrays();  }  反射读取注解信息  **publicstaticvoid** main(String[] args) **throws** ClassNotFoundException {  ClassclassInfo = Class.*forName*("com.itmayiedu.entity.User");  // 获取到所有方法  Method[] methods = classInfo.getDeclaredMethods();  **for** (Method method : methods) {  System.***out***.println(method);  AddAnnotationdeclaredAnnotation = method.getDeclaredAnnotation(AddAnnotation.**class**);  **if** (declaredAnnotation == **null**) {  // 结束本次循环  **continue**;  }  // 获取userId  **int**userId = declaredAnnotation.userId();  System.***out***.println("userId:" + userId);  // 获取userName  String userName = declaredAnnotation.userName();  System.***out***.println("userName:" + userName);  // 获取arrays  String[] arrays = declaredAnnotation.arrays();  **for** (String str : arrays) {  System.***out***.println("str:" + str);  }  }  } |

### 自定义事务注解

|  |
| --- |
| //编程事务（需要手动begin 手动回滚手都提交）  @Component()  @Scope("prototype") // 设置成原型解决线程安全  **publicclass** TransactionUtils {  **private** TransactionStatus transactionStatus;  // 获取事务源  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  // 开启事务  **public** TransactionStatus begin() {  transactionStatus = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionAttribute());  **return**transactionStatus;  }  // 提交事务  **publicvoid** commit(TransactionStatus transaction) {  dataSourceTransactionManager.commit(transaction);  }  // 回滚事务  **publicvoid** rollback() {  System.***out***.println("rollback");  dataSourceTransactionManager.rollback(transactionStatus);  }  }  **注解类**  @Autowired  **private**TransactionUtilstransactionUtils;  @AfterThrowing("execution(\* com.itmayiedu.service.\*.\*.\*(..))")  **publicvoid** afterThrowing() **throws** NoSuchMethodException, SecurityException {  // isRollback(proceedingJoinPoint);  System.***out***.println("程序发生异常");  // TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();  // TransactionStatus currentTransactionStatus =  // TransactionAspectSupport.currentTransactionStatus();  // System.out.println("currentTransactionStatus:" +  // currentTransactionStatus);  transactionUtils.rollback();  }  // // 环绕通知 在方法之前和之后处理事情  @Around("execution(\* com.itmayiedu.service.\*.\*.\*(..))")  **publicvoid** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  // 调用方法之前执行  TransactionStatus transactionStatus = begin(proceedingJoinPoint);  proceedingJoinPoint.proceed();// 代理调用方法 注意点： 如果调用方法抛出异常不会执行后面代码  // 调用方法之后执行  commit(transactionStatus);  }  **public** TransactionStatus begin(ProceedingJoinPoint pjp) **throws** NoSuchMethodException, SecurityException {  // // 判断是否有自定义事务注解  ExtTransactiondeclaredAnnotation = getExtTransaction(pjp);  **if** (declaredAnnotation == **null**) {  **returnnull**;  }  // 如果有自定义事务注解，开启事务  System.***out***.println("开启事务");  TransactionStatus transactionStatu = transactionUtils.begin();  **return**transactionStatu;  }  **publicvoid** commit(TransactionStatus transactionStatu) {  **if** (transactionStatu != **null**) {  // 提交事务  System.***out***.println("提交事务");  transactionUtils.commit(transactionStatu);  }  }  **public**ExtTransaction getExtTransaction(ProceedingJoinPoint pjp) **throws** NoSuchMethodException, SecurityException {  // 获取方法名称  String methodName = pjp.getSignature().getName();  // 获取目标对象  Class<?>classTarget = pjp.getTarget().getClass();  // 获取目标对象类型  Class<?>[] par = ((MethodSignature) pjp.getSignature()).getParameterTypes();  // 获取目标对象方法  Method objMethod = classTarget.getMethod(methodName, par);  // // 判断是否有自定义事务注解  ExtTransactiondeclaredAnnotation = objMethod.getDeclaredAnnotation(ExtTransaction.**class**);  **if** (declaredAnnotation == **null**) {  System.***out***.println("您的方法上,没有加入注解!");  **returnnull**;  }  **return**declaredAnnotation;  }  // 回滚事务  **publicvoid** isRollback(ProceedingJoinPoint pjp) **throws** NoSuchMethodException, SecurityException {  // // 判断是否有自定义事务注解  ExtTransactiondeclaredAnnotation = getExtTransaction(pjp);  **if** (declaredAnnotation != **null**) {  System.***out***.println("已经开始回滚事务");  // 获取当前事务 直接回滚  TransactionAspectSupport.*currentTransactionStatus*().setRollbackOnly();  **return**;  }  }  **使用自定义注解**  @ExtTransaction  **publicvoid** add() {  userDao.add("test001", 20);  **int**i = 1 / 0;  System.***out***.println("################");  userDao.add("test002", 21);  } |

## Spring事物传播行为

Spring中事务的定义：

Propagation（key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。）有以下选项可供使用：

PROPAGATION\_REQUIRED—如果当前有事务，就用当前事务，如果当前没有事务，就新建一个事务。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。//如果外层方法没有事务，就会以非事务进行执行。

PROPAGATION\_MANDATORY--支持当前事务，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。

--- 如果当前有事务，就是以非事务进行执行

PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

默认传播行为为REQUIRED

# SpringIOC原理

## XML技术

### 什么是XML

它是可扩展标记语言（Extensible Markup Language，简称XML），是一种标记语言。

XML 全称为可扩展的标记语言。主要用于描述数据和用作配置文件。

XML 文档在逻辑上主要由一下 5 个部分组成：

XML 声明：指明所用 XML 的版本、文档的编码、文档的独立性信息

文档类型声明：指出 XML 文档所用的 DTD

元素：由开始标签、元素内容和结束标签构成

注释：以结束，用于对文档中的内容起一个说明作用

处理指令：通过处理指令来通知其他应用程序来处理非 XML 格式的数据，格式为

XML 文档的根元素被称为文档元素，它和在其外部出现的处理指令、注释等作为文档实体的子节点，根元素本身和其内部的子元素也是一棵树。

### XML样例

|  |
| --- |
| **<?xml version="1.0" encoding="UTF-8"?>**  **<students>**  **<student1 id="001">**  **<微信公众号>@残缺的孤独</微信公众号>**  **<学号>20140101</学号>**  **<地址>北京海淀区</地址>**  **<座右铭>要么强大，要么听话</座右铭>**  **</student1>**  **<student2 id="002">**  **<新浪微博>@残缺的孤独</新浪微博>**  **<学号>20140102</学号>**  **<地址>北京朝阳区</地址>**  **<座右铭>在哭泣中学会坚强</座右铭>**  **</student2>**  **</students>** |

**<?xml version="1.0" encoding="UTF-8"?>**

作用[xml文件](https://www.baidu.com/s?wd=xml%E6%96%87%E4%BB%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YdnjK9rjbvuWfLPAN9Ph7W0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHR4rjR1rHc4nWTYP10krj0Y)头部要写的话，说明了xml的版本和编码，utf-8一般是[网络传输](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E4%BC%A0%E8%BE%93&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YdnjK9rjbvuWfLPAN9Ph7W0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHR4rjR1rHc4nWTYP10krj0Y)用的编码

### XML解析方式？

Dom4j、Sax、Pull

### Dom4j与Sax区别

 dom4j不适合大文件的解析，因为它是一下子将文件加载到内存中，所以有可能出现内存溢出，sax是基于事件来对xml进行解析的，所以他可以解析大文件的xml，也正是因为如此，所以dom4j可以对xml进行灵活的增删改查和导航，而sax没有这么强的灵活性，所以sax经常是用来解析大型xml文件，而要对xml文件进行一些灵活（crud）操作就用dom4j。

### 使用dom4j解析xml

解析XML过程是通过获取Document对象，然后继续获取各个节点以及属性等操作，因此获取Document对象是第一步，大体说来，有三种方式：

1.自己创建Document对象

|  |
| --- |
| **Documentdocument = DocumentHelper.*createDocument*();**  **Element root = document.addElement("students");** |

其中students是根节点，可以继续添加其他节点等操作。

2.自己创建Document对象

|  |
| --- |
| **// 创建SAXReader对象**  **SAXReaderreader = newSAXReader();**  **// 读取文件转换成Document**  **Documentdocument = reader.read(newFile("XXXX.xml"));** |

3.读取XML文本内容获取Document对象

|  |
| --- |
| **String xmlStr = "<students>......</students>";**  **Documentdocument = DocumentHelper.*parseText*(xmlStr);** |

### 解析xml代码

Xml配置:

|  |
| --- |
| **<?xml version="1.0" encoding="UTF-8"?>**  **<students>**  **<student1 id="001">**  **<微信公众号>每特学院</微信公众号>**  **<学号>20140101</学号>**  **<地址>北京海淀区</地址>**  **<座右铭>要么强大，要么听话</座右铭>**  **</student1>**  **<student2 id="002">**  **<新浪微博>蚂蚁课堂</新浪微博>**  **<学号>20140102</学号>**  **<地址>北京朝阳区</地址>**  **<座右铭>在哭泣中学会坚强</座右铭>**  **</student2>**  **</students>** |

Java代码

|  |
| --- |
| **publicstaticvoid** main(String[] args) **throws** SAXException, DocumentException {  XmlUtils xmlUtils = **new** XmlUtils();  xmlUtils.test001();  }  **publicvoid** test001() **throws** DocumentException {  SAXReader saxReader = **new** SAXReader();  Document read = saxReader.read(getClassPath("student.xml"));  // 获取根节点  Element rootElement = read.getRootElement();  *getNodes*(rootElement);  }  **public** InputStream getClassPath(String xmlPath) {  InputStream resourceAsStream = getClass().getClassLoader().getResourceAsStream(xmlPath);  **return**resourceAsStream;  }  **publicstaticvoid** getNodes(Element rootElement) {  System.***out***.println("获取当前名称:" + rootElement.getName());  // 获取属性信息  List<Attribute>attributes = rootElement.attributes();  **for** (Attribute attribute : attributes) {  System.***out***.println("属性:" + attribute.getName() + "---" + attribute.getText());  }  // 获取属性value  String value = rootElement.getTextTrim();  **if** (!StringUtils.*isEmpty*(value)) {  System.***out***.println("value:" + value);  }  // 使用迭代器遍历,继续遍历子节点  Iterator<Element>elementIterator = rootElement.elementIterator();  **while** (elementIterator.hasNext()) {  Element next = elementIterator.next();  *getNodes*(next);  } |

注意:

this.getClass().getClassLoader().getResourceAsStream(xmlPath) 获取当前项目路径xmlfsfs

## XML与JSON区别

Xml是重量级数据交换格式，占宽带比较大。

JSON是轻量级交换格式，xml占宽带小。

所有很多互联网公司都会使用json作为数据交换格式

很多银行项目，大多数还是在使用xml。

## 什么是SpringIOC

spring ioc指的是控制反转，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。交由Spring来管理这些，实现解耦

## SpringIOC原理

使用反射机制+XML技术

### 手写SpringIOCXML版本

|  |
| --- |
| /\*\*  \* 手写Spring专题 XML方式注入bean  \*  \* **@author**作者:余胜军  \*  \*/  **publicclass** ClassPathXmlApplicationContext {  // xml路径地址  **private** String xmlPath;  **public** ClassPathXmlApplicationContext(String xmlPath) {  **this**.xmlPath = xmlPath;  }  **public** Object getBean(String beanId) **throws** Exception {  // 1. 读取配置文件  List<Element>elements = readerXml();  **if** (elements == **null**) {  **thrownew** Exception("该配置文件没有子元素");  }  // 2. 使用beanId查找对应的class地址  String beanClass = findXmlByIDClass(elements, beanId);  **if** (StringUtils.*isEmpty*(beanClass)) {  **thrownew** Exception("未找到对应的class地址");  }  // 3. 使用反射机制初始化，对象  Class<?>forName = Class.*forName*(beanClass);  **return**forName.newInstance();  }  // 读取配置文件信息  **public** List<Element> readerXml() **throws** DocumentException {  SAXReader saxReader = **new** SAXReader();  **if** (StringUtils.*isEmpty*(xmlPath)) {  **new** Exception("xml路径为空...");  }  Document read = saxReader.read(getClassXmlInputStream(xmlPath));  // 获取根节点信息  Element rootElement = read.getRootElement();  // 获取子节点  List<Element>elements = rootElement.elements();  **if** (elements == **null** || elements.isEmpty()) {  **returnnull**;  }  **return**elements;  }  // 使用beanid查找该Class地址  **public** String findXmlByIDClass(List<Element>elements, String beanId) **throws** Exception {  **for** (Element element : elements) {  // 读取节点上是否有value  String beanIdValue = element.attributeValue("id");  **if** (beanIdValue == **null**) {  **thrownew** Exception("使用该beanId为查找到元素");  }  **if** (!beanIdValue.equals(beanId)) {  **continue**;  }  // 获取Class地址属性  String classPath = element.attributeValue("class");  **if** (!StringUtils.*isEmpty*(classPath)) {  **return**classPath;  }  }  **returnnull**;  }  // 读取xml配置文件  **public** InputStream getClassXmlInputStream(String xmlPath) {  InputStream resourceAsStream = getClass().getClassLoader().getResourceAsStream(xmlPath);  **return**resourceAsStream;  }  } |

### 手写SpringIOC注解版本

|  |
| --- |
| /\*\*  \* 手写Spring专题注解版本注入bean  \*  \* **@author**作者:余胜军  \*  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  **publicclass** ClassPathXmlApplicationContext {  // 扫包范围  **private** String packageName;  ConcurrentHashMap<String, Object>initBean = **null**;  **public** ClassPathXmlApplicationContext(String packageName) {  **this**.packageName = packageName;  }  // 使用beanID查找对象  **public** Object getBean(String beanId) **throws** Exception {  // 1.使用反射机制获取该包下所有的类已经存在bean的注解类  List<Class>listClassesAnnotation = findClassExisService();  **if** (listClassesAnnotation == **null** || listClassesAnnotation.isEmpty()) {  **thrownew** Exception("没有需要初始化的bean");  }  // 2.使用Java反射机制初始化对象  initBean = initBean(listClassesAnnotation);  **if** (initBean == **null** || initBean.isEmpty()) {  **thrownew** Exception("初始化bean为空!");  }  // 3.使用beanID查找查找对应bean对象  Object object = initBean.get(beanId);  // 4.使用反射读取类的属性,赋值信息  attriAssign(object);  **return**object;  }  // 使用反射读取类的属性,赋值信息  **publicvoid** attriAssign(Object object) **throws** IllegalArgumentException, IllegalAccessException {  // 1.获取类的属性是否存在获取bean注解  Class<? **extends** Object>classInfo = object.getClass();  Field[] declaredFields = classInfo.getDeclaredFields();  **for** (Field field : declaredFields) {  // 属性名称  String name = field.getName();  // 2.使用属性名称查找bean容器赋值  Object bean = initBean.get(name);  **if** (bean != **null**) {  // 私有访问允许访问  field.setAccessible(**true**);  // 给属性赋值  field.set(object, bean);  **continue**;  }  }  }  // 使用反射机制获取该包下所有的类已经存在bean的注解类  **public** List<Class> findClassExisService() **throws** Exception {  // 1.使用反射机制获取该包下所有的类  **if** (StringUtils.*isEmpty*(packageName)) {  **thrownew** Exception("扫包地址不能为空!");  }  // 2.使用反射技术获取当前包下所有的类  List<Class<?>>classesByPackageName = ClassUtil.*getClasses*(packageName);  // 3.存放类上有bean注入注解  List<Class>exisClassesAnnotation = **new** ArrayList<Class>();  // 4.判断该类上属否存在注解  **for** (Class classInfo : classesByPackageName) {  ExtServiceextService = (ExtService) classInfo.getDeclaredAnnotation(ExtService.**class**);  **if** (extService != **null**) {  exisClassesAnnotation.add(classInfo);  **continue**;  }  }  **return**exisClassesAnnotation;  }  // 初始化bean对象  **public** ConcurrentHashMap<String, Object> initBean(List<Class>listClassesAnnotation)  **throws** InstantiationException, IllegalAccessException {  ConcurrentHashMap concurrentHashMap = **new** ConcurrentHashMap<String, Object>();  **for** (Class classInfo : listClassesAnnotation) {  // 初始化对象  Object newInstance = classInfo.newInstance();  // 获取父类名称  String beanId = *toLowerCaseFirstOne*(classInfo.getSimpleName());  concurrentHashMap.put(beanId, newInstance);  }  **return**concurrentHashMap;  }  // 首字母转小写  **publicstatic** String toLowerCaseFirstOne(String s) {  **if** (Character.*isLowerCase*(s.charAt(0)))  **return**s;  **else**  **return** (**new** StringBuilder()).append(Character.*toLowerCase*(s.charAt(0))).append(s.substring(1)).toString();  }  } |

### 常用反射工具类

|  |
| --- |
| **publicclass** ClassUtil {  /\*\*  \* 取得某个接口下所有实现这个接口的类  \*/  **publicstatic** List<Class> getAllClassByInterface(Classc) {  List<Class>returnClassList = **null**;  **if** (c.isInterface()) {  // 获取当前的包名  String packageName = c.getPackage().getName();  // 获取当前包下以及子包下所以的类  List<Class<?>>allClass = *getClasses*(packageName);  **if** (allClass != **null**) {  returnClassList = **new** ArrayList<Class>();  **for** (Classclasses : allClass) {  // 判断是否是同一个接口  **if** (c.isAssignableFrom(classes)) {  // 本身不加入进去  **if** (!c.equals(classes)) {  returnClassList.add(classes);  }  }  }  }  }  **return**returnClassList;  }  /\*  \* 取得某一类所在包的所有类名不含迭代  \*/  **publicstatic** String[] getPackageAllClassName(String classLocation, String packageName) {  // 将packageName分解  String[] packagePathSplit = packageName.split("[.]");  String realClassLocation = classLocation;  **int**packageLength = packagePathSplit.length;  **for** (**int**i = 0; i<packageLength; i++) {  realClassLocation = realClassLocation + File.***separator*** + packagePathSplit[i];  }  File packeageDir = **new** File(realClassLocation);  **if** (packeageDir.isDirectory()) {  String[] allClassName = packeageDir.list();  **return**allClassName;  }  **returnnull**;  }  /\*\*  \* 从包package中获取所有的Class  \*  \* **@param** pack  \* **@return**  \*/  **publicstatic** List<Class<?>> getClasses(String packageName) {  // 第一个class类的集合  List<Class<?>>classes = **new** ArrayList<Class<?>>();  // 是否循环迭代  **boolean**recursive = **true**;  // 获取包的名字并进行替换  String packageDirName = packageName.replace('.', '/');  // 定义一个枚举的集合并进行循环来处理这个目录下的things  Enumeration<URL>dirs;  **try** {  dirs = Thread.*currentThread*().getContextClassLoader().getResources(packageDirName);  // 循环迭代下去  **while** (dirs.hasMoreElements()) {  // 获取下一个元素  URL url = dirs.nextElement();  // 得到协议的名称  String protocol = url.getProtocol();  // 如果是以文件的形式保存在服务器上  **if** ("file".equals(protocol)) {  // 获取包的物理路径  String filePath = URLDecoder.*decode*(url.getFile(), "UTF-8");  // 以文件的方式扫描整个包下的文件并添加到集合中  *findAndAddClassesInPackageByFile*(packageName, filePath, recursive, classes);  } **elseif** ("jar".equals(protocol)) {  // 如果是jar包文件  // 定义一个JarFile  JarFile jar;  **try** {  // 获取jar  jar = ((JarURLConnection) url.openConnection()).getJarFile();  // 从此jar包得到一个枚举类  Enumeration<JarEntry>entries = jar.entries();  // 同样的进行循环迭代  **while** (entries.hasMoreElements()) {  // 获取jar里的一个实体可以是目录和一些jar包里的其他文件如META-INF等文件  JarEntry entry = entries.nextElement();  String name = entry.getName();  // 如果是以/开头的  **if** (name.charAt(0) == '/') {  // 获取后面的字符串  name = name.substring(1);  }  // 如果前半部分和定义的包名相同  **if** (name.startsWith(packageDirName)) {  **int**idx = name.lastIndexOf('/');  // 如果以"/"结尾是一个包  **if** (idx != -1) {  // 获取包名把"/"替换成"."  packageName = name.substring(0, idx).replace('/', '.');  }  // 如果可以迭代下去并且是一个包  **if** ((idx != -1) || recursive) {  // 如果是一个.class文件而且不是目录  **if** (name.endsWith(".class") && !entry.isDirectory()) {  // 去掉后面的".class" 获取真正的类名  String className = name.substring(packageName.length() + 1, name.length() - 6);  **try** {  // 添加到classes  classes.add(Class.*forName*(packageName + '.' + className));  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  }  }  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  **return**classes;  }  /\*\*  \* 以文件的形式来获取包下的所有Class  \*  \* **@param** packageName  \* **@param** packagePath  \* **@param** recursive  \* **@param** classes  \*/  **publicstaticvoid** findAndAddClassesInPackageByFile(String packageName, String packagePath, **finalboolean**recursive,  List<Class<?>>classes) {  // 获取此包的目录建立一个File  File dir = **new** File(packagePath);  // 如果不存在或者也不是目录就直接返回  **if** (!dir.exists() || !dir.isDirectory()) {  **return**;  }  // 如果存在就获取包下的所有文件包括目录  File[] dirfiles = dir.listFiles(**new** FileFilter() {  // 自定义过滤规则如果可以循环(包含子目录) 或则是以.class结尾的文件(编译好的java类文件)  **publicboolean** accept(File file) {  **return** (recursive&&file.isDirectory()) || (file.getName().endsWith(".class"));  }  });  // 循环所有文件  **for** (File file : dirfiles) {  // 如果是目录则继续扫描  **if** (file.isDirectory()) {  *findAndAddClassesInPackageByFile*(packageName + "." + file.getName(), file.getAbsolutePath(), recursive,  classes);  } **else** {  // 如果是java类文件去掉后面的.class 只留下类名  String className = file.getName().substring(0, file.getName().length() - 6);  **try** {  // 添加到集合中去  classes.add(Class.*forName*(packageName + '.' + className));  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  }  }  }  } |

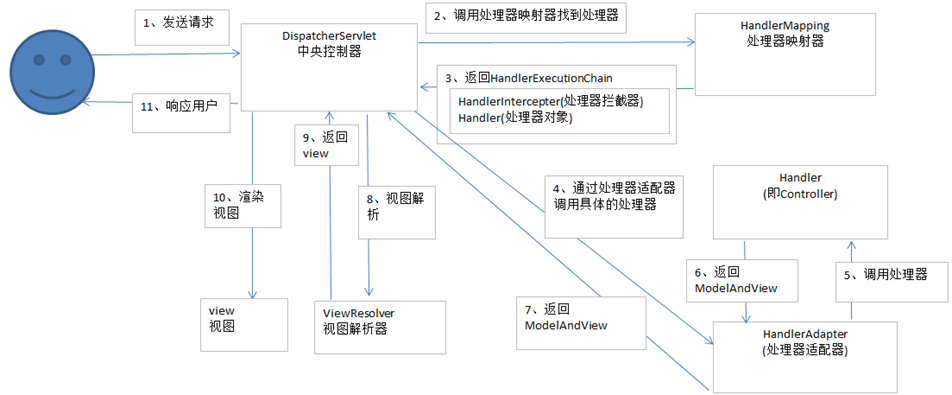
### Maven相关依赖

|  |
| --- |
| <dependencies>  <!-- 引入Spring-AOP等相关Jar -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjrt</artifactId>  <version>1.6.1</version>  </dependency>  <dependency>  <groupId>aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.5.3</version>  </dependency>  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.1\_2</version>  </dependency>  <!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->  <dependency>  <groupId>com.mchange</groupId>  <artifactId>c3p0</artifactId>  <version>0.9.5.2</version>  </dependency>  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.37</version>  </dependency>  <!-- https://mvnrepository.com/artifact/dom4j/dom4j -->  <dependency>  <groupId>dom4j</groupId>  <artifactId>dom4j</artifactId>  <version>1.6.1</version>  </dependency>  <!-- https://mvnrepository.com/artifact/commons-lang/commons-lang -->  <dependency>  <groupId>commons-lang</groupId>  <artifactId>commons-lang</artifactId>  <version>2.6</version>  </dependency>  </dependencies> |

### SpringIOC容器核心接口

# SpringMVC原理

### SpringMVC的运行流程



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。

### 回顾Servet知识

#### 什么是Servlet

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

使用 Servlet，您可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页。

Java Servlet 通常情况下与使用 CGI（Common Gateway Interface，公共网关接口）实现的程序可以达到异曲同工的效果。但是相比于 CGI，Servlet 有以下几点优势：

性能明显更好。

Servlet 在 Web 服务器的地址空间内执行。这样它就没有必要再创建一个单独的进程来处理每个客户端请求。

Servlet 是独立于平台的，因为它们是用 Java 编写的。

服务器上的 Java 安全管理器执行了一系列限制，以保护服务器计算机上的资源。因此，Servlet 是可信的。

Java 类库的全部功能对 Servlet 来说都是可用的。它可以通过 sockets 和 RMI 机制与 applets、数据库或其他软件进行交互。

#### 什么是Servlet生命周期

### Servlet 生命周期：

Servlet 加载—>实例化—>服务—>销毁。

**init（）：**

在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。

**service（）：**

它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个HttpServlet对象，该对象的Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。

**destroy（）：**

仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

### 手写SpringMVC思路

#### 1.web.xml加载

 为了读取web.xml中的配置，我们用到ServletConfig这个类，它代表当前Servlet在web.xml中的配置信息。通过web.xml中加载我们自己写的MyDispatcherServlet和读取配置文件。

#### 2、初始化阶段

  在前面我们提到DispatcherServlet的initStrategies方法会初始化9大组件，但是这里将实现一些SpringMVC的最基本的组件而不是全部，按顺序包括：

* 加载配置文件
* 扫描用户配置包下面所有的类
* 拿到扫描到的类，通过反射机制，实例化。并且放到ioc容器中(Map的键值对 beanName-bean) beanName默认是首字母小写
* 初始化HandlerMapping，这里其实就是把url和method对应起来放在一个k-v的Map中,在运行阶段取出

#### 3、运行阶段

  每一次请求将会调用doGet或doPost方法，所以统一运行阶段都放在doDispatch方法里处理，它会根据url请求去HandlerMapping中匹配到对应的Method，然后利用反射机制调用Controller中的url对应的方法，并得到结果返回。按顺序包括以下功能：

* 异常的拦截
* 获取请求传入的参数并处理参数
* 通过初始化好的handlerMapping中拿出url对应的方法名，反射调用

### 手写SpringMVC基本实现

|  |
| --- |
| /\*\*  \* 手写SpringMVC框架作者:每特教育-余胜军<br>  \*  \* **@QQ644064779** 1.自定义DispatcherServlet<br>  \* 2.servletinit()方法初始化###只会执行一次<br>  \* ######2.1获取当前包下所有的类<br>  \* ######2.2初始化当前包下所有的类,使用Java反射机制初始化对象存放在SpringMVC容器中key(beanId)-  \* value( 当前实例对象) <br>  \* ######2.3初始化HandlerMapping方法,将url和方法对应上<br>  \* ########2.3.1使用Java反射技术读取类的信息,存放在map集合中key为url请求地址,value为对应方法  \* <br>  \* ########2.3.2使用Java反射技术读取类的信息,存放在map集合中key为url请求地址,value为对应实例对象  \* <br>  \* 3.servlet get或者post请求<br>  \* ######## 3.1.1获取请求地址,使用Java反射技术找到对应的方法和实例对象进行执行<br>  \*/  **publicclass**ExtDispatcherServlet**extends** HttpServlet {  // mvc bean key=beanid ,value=对象  **private** ConcurrentHashMap<String, Object>mvcBeans = **new** ConcurrentHashMap<String, Object>();  // mvc请求方法 key=requestUrl,value=对象  **private** ConcurrentHashMap<String, Object>mvcBeanUrl = **new** ConcurrentHashMap<String, Object>();  // mvc请求方法 key=requestUrl,value=方法  **private** ConcurrentHashMap<String, String>mvcMethodUrl = **new** ConcurrentHashMap<String, String>();  /\*\*  \* 初始化自定义SpringMVC容器  \*/  **publicvoid** init() **throws** ServletException {  **try** {  // 1.获取当前包下所有的类  List<Class<?>>classes = ClassUtil.*getClasses*("com.itmayiedu.ext.controller");  // 2.初始化当前包下所有的类,使用Java反射机制初始化对象存放在SpringMVC容器中key(beanId)-value(  // 当前实例对象)  findClassMVCBeans(classes);  // 3.初始化HandlerMapping方法,将url和方法对应上  handlerMapping(mvcBeans);  } **catch** (Exception e) {  }  }  // 2.初始化当前包下所有的类,使用Java反射机制初始化对象存放在SpringMVC容器中key(beanId)-value(  // 当前实例对象)  **publicvoid** findClassMVCBeans(List<Class<?>>classes)  **throws** ClassNotFoundException, InstantiationException, IllegalAccessException {  mvcBeans = **new** ConcurrentHashMap<String, Object>();  **for** (Class<?>classInfo : classes) {  ExtControllerextController = classInfo.getDeclaredAnnotation(ExtController.**class**);  **if** (extController != **null**) {  // 默认类名小写作为bean的名称  String beanId = ClassUtil.*toLowerCaseFirstOne*(classInfo.getSimpleName());  mvcBeans.put(beanId, ClassUtil.*newInstance*(classInfo));  }  }  }  // 3.初始化HandlerMapping方法,将url和方法对应上  **publicvoid** handlerMapping(ConcurrentHashMap<String, Object>mvcBeans) {  // 遍历mvc bean对象  **for** (Map.Entry<String, Object>entry : mvcBeans.entrySet()) {  // springmvc注入object对象  Object mvcObject = entry.getValue();  // 判断类上是否有@ExtRequestMapping注解  Class<? **extends** Object>classInfo = mvcObject.getClass();  String requestBaseUrl = **null**;  ExtRequestMappingclassExtRequestMapping = classInfo.getAnnotation(ExtRequestMapping.**class**);  **if** (classExtRequestMapping != **null**) {  requestBaseUrl = classExtRequestMapping.value();  }  // 遍历当前类的所有方法,判断方法上是否有注解  Method[] declaredMethods = classInfo.getDeclaredMethods();  **for** (Method method : declaredMethods) {  ExtRequestMappingmethodExtRequestMapping = method.getDeclaredAnnotation(ExtRequestMapping.**class**);  **if** (methodExtRequestMapping != **null**) {  String httpRequestUrl = methodExtRequestMapping.value();  mvcBeanUrl.put(requestBaseUrl + httpRequestUrl, mvcObject);  mvcMethodUrl.put(requestBaseUrl + httpRequestUrl, method.getName());  }  }  }  }  @Override  **protectedvoid** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  doPost(req, resp);  }  @Override  **protectedvoid** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **try** {  doDispatch(req, resp);  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  **publicvoid** doDispatch(HttpServletRequest req, HttpServletResponse resp) **throws** IOException, ServletException {  // 1.获取请求url地址  String requestUrl = req.getRequestURI();  // 2.使用请求url查找对应mvc控制器bean  Object object = mvcBeanUrl.get(requestUrl);  **if** (object == **null**) {  resp.getWriter().println("http ext not found controller 404");  **return**;  }  // 3.获取对应的请求方法  String methodName = mvcMethodUrl.get(requestUrl);  **if** (StringUtils.*isEmpty*(methodName)) {  resp.getWriter().println("http ext not found Method 404");  **return**;  }  // 4.使用java反射技术执行方法  Class<? **extends** Object>classInfo = object.getClass();  String resultPage = (String) methodInvoke(classInfo, object, methodName);  // 5.视图展示  viewdisplay(resultPage, req, resp);  }  // 执行方法  **public** Object methodInvoke(Class<? **extends** Object>classInfo, Object object, String methodName) {  **try** {  Method method = classInfo.getMethod(methodName);  Object result = method.invoke(object);  **return**result;  } **catch** (Exception e) {  e.printStackTrace();  **returnnull**;  }  }  // 视图展示  **publicvoid** viewdisplay(String pageName, HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  // 获取后缀信息  String suffix = ".jsp";  // 页面目录地址  String prefix = "/";  req.getRequestDispatcher(prefix + pageName + suffix).forward(req, res);  }  } |

## 数据库连接池原理

### 基本原理

在内部对象池中，维护一定数量的数据库连接，并对外暴露数据库连接的获取和返回方法。

如外部使用者可通过getConnection方法获取数据库连接，使用完毕后再通过releaseConnection方法将连接返回，注意此时的连接并没有关闭，而是由连接池管理器回收，并为下一次使用做好准备。

### 线程池作用

 ①资源重用

    由于数据库连接得到重用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，增进了系统环境的平稳性（减少内存碎片以级数据库临时进程、线程的数量）

  ②更快的系统响应速度

    数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池内备用。此时连接池的初始化操作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应时间。

  ③新的资源分配手段

    对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接的配置，实现数据库连接技术。

  ④统一的连接管理，避免数据库连接泄露

   在较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用的连接，从而避免了常规数据库连接操作中可能出现的资源泄露

### 常用数据库连接池

#### C3P0

C3P0是一个开放源代码的JDBC连接池，它在lib目录中与Hibernate一起发布,包括了实现jdbc3和jdbc2扩展规范说明的Connection 和Statement 池的DataSources 对象。

#### BoneCP

BoneCP 是一个开源的快速的 JDBC 连接池。BoneCP很小，只有四十几K(运行时需要log4j和Google Collections的支持，这二者加起来就不小了)，而相比之下 C3P0 要六百多K。另外个人觉得 BoneCP 有个缺点是，JDBC驱动的加载是在连接池之外的，这样在一些应用服务器的配置上就不够灵活。当然，体积小并不是 BoneCP 优秀的原因，BoneCP 到底有什么突出的地方呢，请看看性能测试报告。池，Tomcat的数据源使用的就是DBCP。目前 DBCP 有两个版本分别是 1.3 和 1.4。1.3 版

#### DBCP

DBCP (Database Connection Pool)是一个依赖Jakarta commons-pool对象池机制的数据库连接本对应的是 JDK 1.4-1.5 和 JDBC 3，而1.4 版本对应 JDK 1.6 和 JDBC 4。因此在选择版本的时候要看看你用的是什么 JDK 版本了，功能上倒是没有什么区别。

#### Proxool

Proxool是一个Java SQL Driver驱动程序，提供了对你选择的其它类型的驱动程序的连接池封装。可以非常简单的移植到现存的代码中。完全可配置。快速，成熟，健壮。可以透明地为你现存的JDBC驱动程序增加连接池功能。

#### DBCP与C3P0区别

dbcp没有自动的去回收空闲连接的功能  c3p0有自动回收空闲连接功能  两者主要是对数据连接的处理方式不同！C3P0提供最大空闲时间，DBCP提供最大连接数。 前者当连接超过最大空闲连接时间时，当前连接就会被断掉。DBCP当连接数超过最大连接数时，所有连接都会被断

### 纯手写数据库连接池

#### 读取外部配置信息

|  |
| --- |
| //外部配置文件信息  **publicclass** DbBean {  /\* 链接属性 \*/  **private** String driverName = "com.mysql.jdbc.Driver";  **private** String url = "jdbc:mysql://localhost:3306/test";  **private** String userName = "root";  **private** String password = "root";  **private** String poolName = "thread01";// 连接池名字  **privateint**minConnections = 1; // 空闲池，最小连接数  **privateint**maxConnections = 10; // 空闲池，最大连接数  **privateint**initConnections = 5;// 初始化连接数  **privatelong**connTimeOut = 1000;// 重复获得连接的频率  **privateint**maxActiveConnections = 100;// 最大允许的连接数，和数据库对应  **privatelong**connectionTimeOut = 1000 \* 60 \* 20;// 连接超时时间，默认20分钟  **public** String getDriverName() {  **return**driverName;  }  **publicvoid** setDriverName(String driverName) {  **this**.driverName = driverName;  }  **public** String getUrl() {  **return**url;  }  **publicvoid** setUrl(String url) {  **this**.url = url;  }  **public** String getUserName() {  **return**userName;  }  **publicvoid** setUserName(String userName) {  **this**.userName = userName;  }  **public** String getPassword() {  **return**password;  }  **publicvoid** setPassword(String password) {  **this**.password = password;  }  **public** String getPoolName() {  **return**poolName;  }  **publicvoid** setPoolName(String poolName) {  **this**.poolName = poolName;  }  **publicint** getMinConnections() {  **return**minConnections;  }  **publicvoid** setMinConnections(**int**minConnections) {  **this**.minConnections = minConnections;  }  **publicint** getMaxConnections() {  **return**maxConnections;  }  **publicvoid** setMaxConnections(**int**maxConnections) {  **this**.maxConnections = maxConnections;  }  **publicint** getInitConnections() {  **return**initConnections;  }  **publicvoid** setInitConnections(**int**initConnections) {  **this**.initConnections = initConnections;  }  **publiclong** getConnTimeOut() {  **return**connTimeOut;  }  **publicvoid** setConnTimeOut(**long**connTimeOut) {  **this**.connTimeOut = connTimeOut;  }  **publicint** getMaxActiveConnections() {  **return**maxActiveConnections;  }  **publicvoid** setMaxActiveConnections(**int**maxActiveConnections) {  **this**.maxActiveConnections = maxActiveConnections;  }  **publiclong** getConnectionTimeOut() {  **return**connectionTimeOut;  }  **publicvoid** setConnectionTimeOut(**long**connectionTimeOut) {  **this**.connectionTimeOut = connectionTimeOut;  }  } |

#### 创建数据库连接池

|  |
| --- |
| /\*\*  \* 数据库连接池<br>  \*  \* 1.初始化<br>  \* ####线程池核心容器空闲线程数、活动线程数<br>  \* ###构造函数 1.1.1初始化线程,存放在空闲线程池中<br>  \* 2.获取连接<br>  \* ####1.判断存在线程数是否大于最大线程如果大于最大线程数,则进行等待...<br>  \* ####2.判断空闲线程数是否大于0 如果空闲线程数<0，创建新的连接<br>  \* ####3.如果空闲线程数>0，则获取当前空闲线程,存入在活动线程集合中<br>  \* 3.释放连接<br>  \* ####3.1.1.判断空闲线程数是否大于最大线程数<br>  \* ####3.1.2.如果空闲线程数小于最大线程数,将该连接收回到空闲线程集合中<br>  \* ####3.1.3.删除该连接对应的活动线程集合数据<br>  \* <br>  \*  \*  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** ConnectionPool **implements** IConnectionPool {  // 空闲线程集合  **private** List<Connection>freeConnection = **new** Vector<Connection>();  // 活动线程集合  **private** List<Connection>activeConnection = **new** Vector<Connection>();  // 记录线程总数  **privatestaticint***connCount* = 0;  **private** DbBean dbBean;  **public** ConnectionPool(DbBean dbBean) {  **this**.dbBean = dbBean;  init();  }  **publicvoid** init() {  **try** {  **for** (**int**i = 0; i<dbBean.getInitConnections(); i++) {  Connection newConnection = newConnection();  **if** (newConnection != **null**) {  // 添加到空闲线程中...  freeConnection.add(newConnection);  }  }  } **catch** (Exception e) {  }  }  // 创建新的Connection  **private** Connection newConnection() {  **try** {  **if** (dbBean == **null**) {  **returnnull**;  }  Class.*forName*(dbBean.getDriverName());  Connection connection = DriverManager.*getConnection*(dbBean.getUrl(), dbBean.getUserName(),  dbBean.getPassword());  *connCount*++;  **return**connection;  } **catch** (Exception e) {  **returnnull**;  }  }  **public** Connection getConnection() {  // \* ####1.判断活动线程数是否大于最大线程如果大于最大线程数,则进行等待...<br>  Connection connection = **null**;  **try** {  **if** (*connCount*<dbBean.getMaxActiveConnections()) {  // 还有活动线程可以使用  // \* ####2.判断空闲线程数是否大于0 如果空闲线程数<0，创建新的连接<br>  **if** (freeConnection.size() > 0) {  connection = freeConnection.remove(0);// 等于freeConnection.get(0);freeConnection.remove(0);  } **else** {  // 创建新的连接  connection = newConnection();  }  **boolean**available = isAvailable(connection);  **if** (available) {  activeConnection.add(connection);  } **else** {  *connCount*--;// i--操作  connection = getConnection();// 递归调用getConnection方法  }  } **else** {  // 大于最大线程数,进行等待,重新获取连接  wait(dbBean.getConnTimeOut());  connection = getConnection();// 递归调用getConnection方法  }  } **catch** (Exception e) {  e.printStackTrace();  }  // \* ####3.如果空闲线程数>0，则获取当前空闲线程,存入在活动线程集合中<br>  **return**connection;  }  // 判断连接是否可用  **publicboolean** isAvailable(Connection connection) {  **try** {  **if** (connection == **null** || connection.isClosed()) {  **returnfalse**;  }  } **catch** (Exception e) {  // **TODO**: handle exception  }  **returntrue**;  }  **publicvoid** releaseConnection(Connection connection) {  **try** {  **if** (connection == **null**) {  **return**;  }  **if** (isAvailable(connection)) {  // 判断空闲线程数是否大于最大线程数  **if** (freeConnection.size() <dbBean.getMaxConnections()) {  freeConnection.add(connection);  } **else** {  // 空闲线程数已经满了  connection.close();  }  activeConnection.remove(connection);  *connCount*--;  notifyAll();  }  } **catch** (Exception e) {  }  }  } |

#### 测试运行结果

|  |
| --- |
| **publicclass** Test001 {  **publicstaticvoid** main(String[] args) {  DBThread dBThread = **new** DBThread();  **for** (**int**i = 1; i<= 3; i++) {  Thread thread = **new** Thread(dBThread, "用户线程" + i);  thread.start();  }  }  }  **class** DBThread **implements** Runnable {  **publicvoid** run() {  **for** (**int**i = 0; i< 10; i++) {  Connection connection = ConnectionPoolManager.*getConnection*();  System.***out***.println(Thread.*currentThread*().getName() + ",connection:" + connection);  ConnectionPoolManager.*releaseConnection*(connection);  }  }  } |

# 纯手写Mybatis框架

### 接口层-和数据库交互的方式

MyBatis和数据库的交互有两种方式：

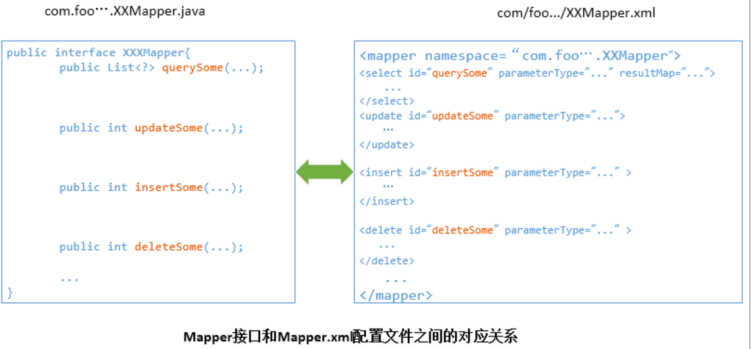
使用传统的MyBatis提供的API；

使用Mapper接口；

### 使用Mapper接口

MyBatis 将配置文件中的每一个<mapper> 节点抽象为一个 Mapper 接口：

这个接口中声明的方法和<mapper> 节点中的<select|update|delete|insert> 节点项对应，即<select|update|delete|insert> 节点的id值为Mapper 接口中的方法名称，parameterType 值表示Mapper 对应方法的入参类型，而resultMap 值则对应了Mapper 接口表示的返回值类型或者返回结果集的元素类型。



根据MyBatis 的配置规范配置好后，通过SqlSession.getMapper(XXXMapper.class)方法，MyBatis 会根据相应的接口声明的方法信息，通过动态代理机制生成一个Mapper 实例，我们使用Mapper接口的某一个方法时，MyBatis会根据这个方法的方法名和参数类型，确定Statement Id，底层还是通过SqlSession.select("statementId",parameterObject);或者SqlSession.update("statementId",parameterObject); 等等来实现对数据库的操作，MyBatis引用Mapper 接口这种调用方式，纯粹是为了满足面向接口编程的需要。（其实还有一个原因是在于，面向接口的编程，使得用户在接口上可以使用注解来配置SQL语句，这样就可以脱离XML配置文件，实现“0配置”）。

### 数据处理层

数据处理层可以说是MyBatis的核心，从大的方面上讲，它要完成两个功能：

通过传入参数构建动态SQL语句；

SQL语句的执行以及封装查询结果集成List<E>；

### 框架支撑层

#### 事务管理机制

事务管理机制对于ORM框架而言是不可缺少的一部分，事务管理机制的质量也是考量一个ORM框架是否优秀的一个标准。

#### 连接池管理机制

由于创建一个数据库连接所占用的资源比较大，对于数据吞吐量大和访问量非常大的应用而言，连接池的设计就显得非常重要。

#### 缓存机制

为了提高数据利用率和减小服务器和数据库的压力，MyBatis 会对于一些查询提供会话级别的数据缓存，会将对某一次查询，放置到SqlSession 中，在允许的时间间隔内，对于完全相同的查询，MyBatis会直接将缓存结果返回给用户，而不用再到数据库中查找。

#### SQL语句的配置方式

传统的MyBatis 配置SQL语句方式就是使用XML文件进行配置的，但是这种方式不能很好地支持面向接口编程的理念，为了支持面向接口的编程，MyBatis 引入了Mapper接口的概念，面向接口的引入，对使用注解来配置SQL语句成为可能，用户只需要在接口上添加必要的注解即可，不用再去配置XML文件了，但是，目前的MyBatis 只是对注解配置SQL语句提供了有限的支持，某些高级功能还是要依赖XML配置文件配置SQL 语句。

### 引导层

引导层是配置和启动MyBatis配置信息的方式。MyBatis 提供两种方式来引导MyBatis ：基于XML配置文件的方式和基于Java API 的方式。

### 主要构件及其相互关系

从MyBatis代码实现的角度来看，MyBatis的主要的核心部件有以下几个：

SqlSession：作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能；

Executor：MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护；

StatementHandler：封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。

ParameterHandler：负责对用户传递的参数转换成JDBC Statement 所需要的参数；

ResultSetHandler：负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；

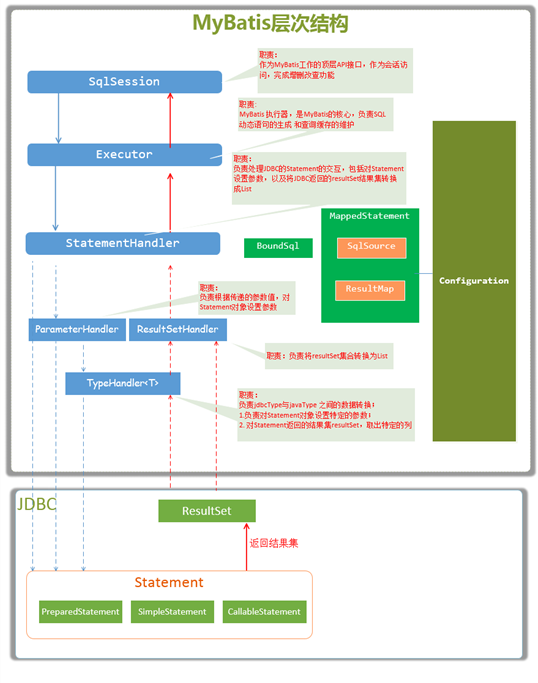
TypeHandler：负责java数据类型和jdbc数据类型之间的映射和转换；

MappedStatement：MappedStatement维护了一条<select|update|delete|insert>节点的封装；

SqlSource：负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回；

BoundSql：表示动态生成的SQL语句以及相应的参数信息；

Configuration：MyBatis所有的配置信息都维持在Configuration对象之中；



### 纯手写Mybatis代码

#### 定义JDBCUtils

|  |
| --- |
| **publicfinalclass**JDBCUtils {  **privatestatic** String *connect*;  **privatestatic** String *driverClassName*;  **privatestatic** String *URL*;  **privatestatic** String *username*;  **privatestatic** String *password*;  **privatestaticboolean***autoCommit*;  /\*\* 声明一个 Connection类型的静态属性，用来缓存一个已经存在的连接对象 \*/  **privatestatic** Connection *conn*;  **static** {  *config*();  }  /\*\*  \* 开头配置自己的数据库信息  \*/  **privatestaticvoid** config() {  /\*  \* 获取驱动  \*/  *driverClassName* = "com.mysql.jdbc.Driver";  /\*  \* 获取URL  \*/  *URL* = "jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8";  /\*  \* 获取用户名  \*/  *username* = "root";  /\*  \* 获取密码  \*/  *password* = "root";  /\*  \* 设置是否自动提交，一般为false不用改  \*/  *autoCommit* = **false**;  }  /\*\*  \* 载入数据库驱动类  \*/  **privatestaticboolean** load() {  **try** {  Class.*forName*(*driverClassName*);  **returntrue**;  } **catch** (ClassNotFoundException e) {  System.***out***.println("驱动类 " + *driverClassName* + " 加载失败");  }  **returnfalse**;  }  /\*\*  \* 专门检查缓存的连接是否不可以被使用，不可以被使用的话，就返回 true  \*/  **privatestaticboolean** invalid() {  **if** (*conn* != **null**) {  **try** {  **if** (*conn*.isClosed() || !*conn*.isValid(3)) {  **returntrue**;  /\*  \* isValid方法是判断Connection是否有效,如果连接尚未关闭并且仍然有效，则返回 true  \*/  }  } **catch** (SQLException e) {  e.printStackTrace();  }  /\*  \* conn既不是 null 且也没有关闭，且 isValid 返回 true，说明是可以使用的 ( 返回 false )  \*/  **returnfalse**;  } **else** {  **returntrue**;  }  }  /\*\*  \* 建立数据库连接  \*/  **publicstatic** Connection connect() {  **if** (*invalid*()) { /\* invalid为true时，说明连接是失败的 \*/  /\* 加载驱动 \*/  *load*();  **try** {  /\* 建立连接 \*/  *conn* = DriverManager.*getConnection*(*URL*, *username*, *password*);  } **catch** (SQLException e) {  System.***out***.println("建立 " + *connect* + " 数据库连接失败 , " + e.getMessage());  }  }  **return***conn*;  }  /\*\*  \* 设置是否自动提交事务  \*\*/  **publicstaticvoid** transaction() {  **try** {  *conn*.setAutoCommit(*autoCommit*);  } **catch** (SQLException e) {  System.***out***.println("设置事务的提交方式为 : " + (*autoCommit* ? "自动提交" : "手动提交") + " 时失败: " + e.getMessage());  }  }  /\*\*  \* 创建 Statement 对象  \*/  **publicstatic** Statement statement() {  Statement st = **null**;  *connect*();  /\* 如果连接是无效的就重新连接 \*/  *transaction*();  /\* 设置事务的提交方式 \*/  **try** {  st = *conn*.createStatement();  } **catch** (SQLException e) {  System.***out***.println("创建 Statement 对象失败: " + e.getMessage());  }  **return**st;  }  /\*\*  \* 根据给定的带参数占位符的SQL语句，创建 PreparedStatement 对象  \*  \* **@param** SQL  \* 带参数占位符的SQL语句  \* **@return**返回相应的 PreparedStatement 对象  \*/  **privatestatic** PreparedStatement prepare(String SQL, **boolean**autoGeneratedKeys) {  PreparedStatement ps = **null**;  *connect*();  /\* 如果连接是无效的就重新连接 \*/  *transaction*();  /\* 设置事务的提交方式 \*/  **try** {  **if** (autoGeneratedKeys) {  ps = *conn*.prepareStatement(SQL, Statement.***RETURN\_GENERATED\_KEYS***);  } **else** {  ps = *conn*.prepareStatement(SQL);  }  } **catch** (SQLException e) {  System.***out***.println("创建 PreparedStatement 对象失败: " + e.getMessage());  }  **return**ps;  }  **publicstatic** ResultSet query(String SQL, List<Object>params) {  **if** (SQL == **null** || SQL.trim().isEmpty() || !SQL.trim().toLowerCase().startsWith("select")) {  **thrownew** RuntimeException("你的SQL语句为空或不是查询语句");  }  ResultSet rs = **null**;  **if** (params.size() > 0) {  /\* 说明有参数传入，就需要处理参数 \*/  PreparedStatement ps = *prepare*(SQL, **false**);  **try** {  **for** (**int**i = 0; i<params.size(); i++) {  ps.setObject(i + 1, params.get(i));  }  rs = ps.executeQuery();  } **catch** (SQLException e) {  System.***out***.println("执行SQL失败: " + e.getMessage());  }  } **else** {  /\* 说明没有传入任何参数 \*/  Statement st = *statement*();  **try** {  rs = st.executeQuery(SQL); // 直接执行不带参数的 SQL 语句  } **catch** (SQLException e) {  System.***out***.println("执行SQL失败: " + e.getMessage());  }  }  **return**rs;  }  **privatestatic** Object typeof(Object o) {  Object r = o;  **if** (o**instanceof** java.sql.Timestamp) {  **return**r;  }  // 将 java.util.Date 转成 java.sql.Date  **if** (o**instanceof** java.util.Date) {  java.util.Date d = (java.util.Date) o;  r = **new** java.sql.Date(d.getTime());  **return**r;  }  // 将 Character 或 char 变成 String  **if** (o**instanceof** Character || o.getClass() == **char**.**class**) {  r = String.*valueOf*(o);  **return**r;  }  **return**r;  }  **publicstaticboolean** execute(String SQL, Object... params) {  **if** (SQL == **null** || SQL.trim().isEmpty() || SQL.trim().toLowerCase().startsWith("select")) {  **thrownew** RuntimeException("你的SQL语句为空或有错");  }  **boolean**r = **false**;  /\* 表示执行 DDL 或 DML 操作是否成功的一个标识变量 \*/  /\* 获得被执行的 SQL 语句的前缀 \*/  SQL = SQL.trim();  SQL = SQL.toLowerCase();  String prefix = SQL.substring(0, SQL.indexOf(" "));  String operation = ""; // 用来保存操作类型的变量  // 根据前缀确定操作  **switch** (prefix) {  **case**"create":  operation = "create table";  **break**;  **case**"alter":  operation = "update table";  **break**;  **case**"drop":  operation = "drop table";  **break**;  **case**"truncate":  operation = "truncate table";  **break**;  **case**"insert":  operation = "insert :";  **break**;  **case**"update":  operation = "update :";  **break**;  **case**"delete":  operation = "delete :";  **break**;  }  **if** (params.length> 0) { // 说明有参数  PreparedStatement ps = *prepare*(SQL, **false**);  Connection c = **null**;  **try** {  c = ps.getConnection();  } **catch** (SQLException e) {  e.printStackTrace();  }  **try** {  **for** (**int**i = 0; i<params.length; i++) {  Object p = params[i];  p = *typeof*(p);  ps.setObject(i + 1, p);  }  ps.executeUpdate();  *commit*(c);  r = **true**;  } **catch** (SQLException e) {  System.***out***.println(operation + " 失败: " + e.getMessage());  *rollback*(c);  }  } **else** { // 说明没有参数  Statement st = *statement*();  Connection c = **null**;  **try** {  c = st.getConnection();  } **catch** (SQLException e) {  e.printStackTrace();  }  // 执行 DDL 或 DML 语句，并返回执行结果  **try** {  st.executeUpdate(SQL);  *commit*(c); // 提交事务  r = **true**;  } **catch** (SQLException e) {  System.***out***.println(operation + " 失败: " + e.getMessage());  *rollback*(c); // 回滚事务  }  }  **return**r;  }  /\*  \*  \* @param SQL 需要执行的 INSERT 语句  \*  \* @param autoGeneratedKeys 指示是否需要返回由数据库产生的键  \*  \* @param params 将要执行的SQL语句中包含的参数占位符的参数值  \*  \* @return 如果指定 autoGeneratedKeys 为 true 则返回由数据库产生的键；如果指定 autoGeneratedKeys  \* 为 false 则返回受当前SQL影响的记录数目  \*/  **publicstaticint** insert(String SQL, **boolean**autoGeneratedKeys, List<Object>params) {  **int**var = -1;  **if** (SQL == **null** || SQL.trim().isEmpty()) {  **thrownew** RuntimeException("你没有指定SQL语句，请检查是否指定了需要执行的SQL语句");  }  // 如果不是 insert 开头开头的语句  **if** (!SQL.trim().toLowerCase().startsWith("insert")) {  System.***out***.println(SQL.toLowerCase());  **thrownew** RuntimeException("你指定的SQL语句不是插入语句，请检查你的SQL语句");  }  // 获得被执行的 SQL 语句的前缀 ( 第一个单词 )  SQL = SQL.trim();  SQL = SQL.toLowerCase();  **if** (params.size() > 0) { // 说明有参数  PreparedStatement ps = *prepare*(SQL, autoGeneratedKeys);  Connection c = **null**;  **try** {  c = ps.getConnection(); // 从 PreparedStatement 对象中获得它对应的连接对象  } **catch** (SQLException e) {  e.printStackTrace();  }  **try** {  **for** (**int**i = 0; i<params.size(); i++) {  Object p = params.get(i);  p = *typeof*(p);  ps.setObject(i + 1, p);  }  **int**count = ps.executeUpdate();  **if** (autoGeneratedKeys) { // 如果希望获得数据库产生的键  ResultSet rs = ps.getGeneratedKeys(); // 获得数据库产生的键集  **if** (rs.next()) { // 因为是保存的是单条记录，因此至多返回一个键  var = rs.getInt(1); // 获得值并赋值给var变量  }  } **else** {  var = count; // 如果不需要获得，则将受SQL影像的记录数赋值给var变量  }  *commit*(c);  } **catch** (SQLException e) {  System.***out***.println("数据保存失败: " + e.getMessage());  *rollback*(c);  }  } **else** { // 说明没有参数  Statement st = *statement*();  Connection c = **null**;  **try** {  c = st.getConnection(); // 从 Statement 对象中获得它对应的连接对象  } **catch** (SQLException e) {  e.printStackTrace();  }  // 执行 DDL 或 DML 语句，并返回执行结果  **try** {  **int**count = st.executeUpdate(SQL);  **if** (autoGeneratedKeys) { // 如果企望获得数据库产生的键  ResultSet rs = st.getGeneratedKeys(); // 获得数据库产生的键集  **if** (rs.next()) { // 因为是保存的是单条记录，因此至多返回一个键  var = rs.getInt(1); // 获得值并赋值给var变量  }  } **else** {  var = count; // 如果不需要获得，则将受SQL影像的记录数赋值给var变量  }  *commit*(c); // 提交事务  } **catch** (SQLException e) {  System.***out***.println("数据保存失败: " + e.getMessage());  *rollback*(c); // 回滚事务  }  }  **return**var;  }  /\*\* 提交事务 \*/  **privatestaticvoid** commit(Connection c) {  **if** (c != **null**&& !*autoCommit*) {  **try** {  c.commit();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  /\*\* 回滚事务 \*/  **privatestaticvoid** rollback(Connection c) {  **if** (c != **null**&& !*autoCommit*) {  **try** {  c.rollback();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  /\*\*  \* 释放资源  \*\*/  **publicstaticvoid** release(Object cloaseable) {  **if** (cloaseable != **null**) {  **if** (cloaseable**instanceof** ResultSet) {  ResultSet rs = (ResultSet) cloaseable;  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **if** (cloaseable**instanceof** Statement) {  Statement st = (Statement) cloaseable;  **try** {  st.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **if** (cloaseable**instanceof** Connection) {  Connection c = (Connection) cloaseable;  **try** {  c.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  }  } |

#### 定义MyInvocationHandlerMbatis

|  |
| --- |
| /\*\*  \* 功能说明:手写mybatis框架注解版本<br>  \* 1.使用动态代理技术,获取接口方法上的sql语句<br>  \* 2.根据不同的SQL语句<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** MyInvocationHandlerMbatis **implements** InvocationHandler {  /\*\*  \* 这个就是我们要代理的真实对象  \*/  **private** Object subject;  /\*\*  \* 构造方法，给我们要代理的真实对象赋初值  \*  \* **@param** subject  \*/  **public** MyInvocationHandlerMbatis(Object subject) {  **this**.subject = subject;  }  /\*\*  \* 该方法负责集中处理动态代理类上的所有方法调用。调用处理器根据这三个参数进行预处理或分派到委托类实例上反射执行  \*  \* **@param** proxy  \* 代理类实例  \* **@param** method  \* 被调用的方法对象  \* **@param** args  \* 调用参数  \* **@return**  \* **@throws** Throwable  \*/  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  // 判断方法上是否有ExtInsert注解  ExtInsertextInsert = method.getAnnotation(ExtInsert.**class**);  **if** (extInsert != **null**) {  **return** insertSQL(extInsert, method, args);  }  // 判断方法上注解类型  ExtSelectextSelect = method.getAnnotation(ExtSelect.**class**);  **if** (extSelect != **null**) {  **return** selectMybatis(extSelect, method, args);  }  **returnnull**;  }  **publicint** insertSQL(ExtInsertextInsert, Method method, Object[] args) {  // 获取注解上的sql  String insertSql = extInsert.value();  System.***out***.println("sql:" + insertSql);  // 获取方法上的参数  Parameter[] parameters = method.getParameters();  // 将方法上的参数存放在Map集合中  ConcurrentHashMap<Object, Object>parameterMap = getExtParams(parameters, args);  // 获取SQL语句上需要传递的参数  String[] sqlParameter = SQLUtils.*sqlInsertParameter*(insertSql);  List<Object>parameValues = **new** ArrayList<>();  **for** (**int**i = 0; i<sqlParameter.length; i++) {  String str = sqlParameter[i];  Object object = parameterMap.get(str);  parameValues.add(object);  }  // 将SQL语句替换为？号  String newSql = SQLUtils.*parameQuestion*(insertSql, sqlParameter);  System.***out***.println("newSql:" + newSql);  // 调用jdbc代码执行  **int**insertResult = JDBCUtils.*insert*(newSql, **false**, parameValues);  **return**insertResult;  }  **public** Object selectMybatis(ExtSelectextInsert, Method method, Object[] args) **throws** SQLException {  **try** {  // 获取查询SQL语句  String selectSQL = extInsert.value();  // 将方法上的参数存放在Map集合中  Parameter[] parameters = method.getParameters();  // 获取方法上参数集合  ConcurrentHashMap<Object, Object>parameterMap = getExtParams(parameters, args);  // 获取SQL传递参数  List<String>sqlSelectParameter = SQLUtils.*sqlSelectParameter*(selectSQL);  // 排序参数  List<Object>parameValues = **new** ArrayList<>();  **for** (**int**i = 0; i<sqlSelectParameter.size(); i++) {  String parameterName = sqlSelectParameter.get(i);  Object object = parameterMap.get(parameterName);  parameValues.add(object.toString());  }  // 变为?号  String newSql = SQLUtils.*parameQuestion*(selectSQL, sqlSelectParameter);  System.***out***.println("执行SQL:" + newSql + "参数信息:" + parameValues.toString());  // 调用JDBC代码查询  ResultSet rs = JDBCUtils.*query*(newSql, parameValues);  // 获取返回类型  Class<?>returnType = method.getReturnType();  **if** (!rs.next()) {  // 没有查找数据  **returnnull**;  }  // 向上移动  rs.previous();  // 实例化对象  Object newInstance = returnType.newInstance();  **while** (rs.next()) {  **for** (String parameterName : sqlSelectParameter) {  // 获取集合中数据  Object value = rs.getObject(parameterName);  // 查找对应属性  Field field = returnType.getDeclaredField(parameterName);  // 设置允许私有访问  field.setAccessible(**true**);  // 赋值参数  field.set(newInstance, value);  }  }  **return**newInstance;  } **catch** (Exception e) {  e.printStackTrace();  }  **returnnull**;  }  **private** ConcurrentHashMap<Object, Object> getExtParams(Parameter[] parameters, Object[] args) {  // 获取方法上参数集合  ConcurrentHashMap<Object, Object>parameterMap = **new** ConcurrentHashMap<>();  **for** (**int**i = 0; i<parameters.length; i++) {  // 参数信息  Parameter parameter = parameters[i];  ExtParamextParam = parameter.getDeclaredAnnotation(ExtParam.**class**);  // 参数名称  String paramValue = extParam.value();  // 参数值  Object oj = args[i];  parameterMap.put(paramValue, oj);  }  **return**parameterMap;  }  } |

#### 定义SqlSession

|  |
| --- |
| /\*\*  \* 获取SqlSession对象<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** SqlSession {  // 获取getMapper  **publicstatic**<T> T getMapper(Class<T>clas)  **throws** IllegalArgumentException, InstantiationException, IllegalAccessException {  **return**(T) Proxy.*newProxyInstance*(clas.getClassLoader(), **new** Class[] { clas },  **new** MyInvocationHandlerMbatis(clas));  }  } |

#### 定义SQLUtils

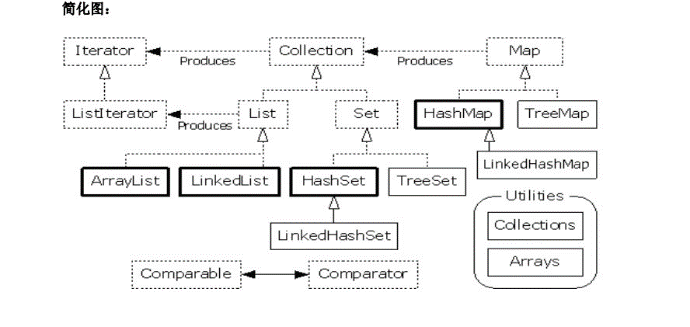
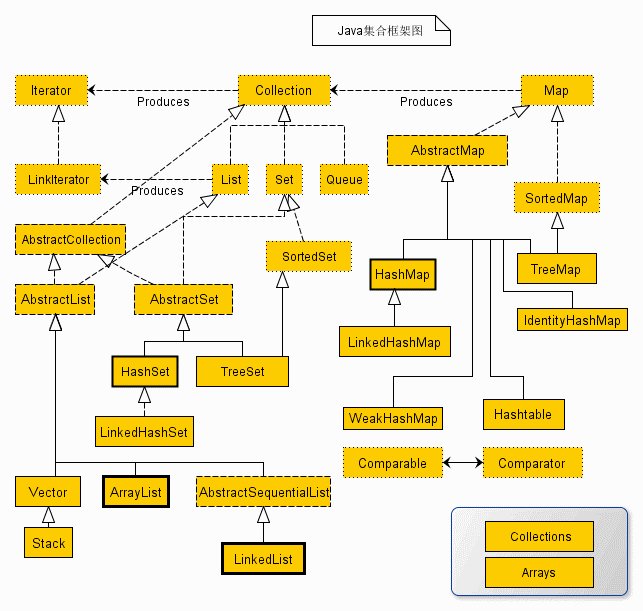
|  |
| --- |
| /\*\*  \* SQL拼接<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** SQLUtils {  /\*\*  \*  \* 获取Insert语句后面values 参数信息<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param** sql  \* **@return**  \*/  **publicstatic** String[] sqlInsertParameter(String sql) {  **int**startIndex = sql.indexOf("values");  **int**endIndex = sql.length();  String substring = sql.substring(startIndex + 6, endIndex).replace("(", "").replace(")", "").replace("#{", "")  .replace("}", "");  String[] split = substring.split(",");  **return**split;  }  /\*\*  \*  \* 获取select 后面where语句作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param** sql  \* **@return**  \*/  **publicstatic** List<String> sqlSelectParameter(String sql) {  **int**startIndex = sql.indexOf("where");  **int**endIndex = sql.length();  String substring = sql.substring(startIndex + 5, endIndex);  String[] split = substring.split("and");  List<String>listArr = **new** ArrayList<>();  **for** (String string : split) {  String[] sp2 = string.split("=");  listArr.add(sp2[0].trim());  }  **return**listArr;  }  /\*\*  \* 将SQL语句的参数替换变为?<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param** sql  \* **@param** parameterName  \* **@return**  \*/  **publicstatic** String parameQuestion(String sql, String[] parameterName) {  **for** (**int**i = 0; i<parameterName.length; i++) {  String string = parameterName[i];  sql = sql.replace("#{" + string + "}", "?");  }  **return**sql;  }  **publicstatic** String parameQuestion(String sql, List<String>parameterName) {  **for** (**int**i = 0; i<parameterName.size(); i++) {  String string = parameterName.get(i);  sql = sql.replace("#{" + string + "}", "?");  }  **return**sql;  }  **publicstaticvoid** main(String[] args) {  // String sql = "insert into user(userName,userAge)  // values(#{userName},#{userAge})";  // String[] sqlParameter = sqlInsertParameter(sql);  // for (String string : sqlParameter) {  // System.out.println(string);  // }  List<String>sqlSelectParameter = SQLUtils  .*sqlSelectParameter*("select \* from User where userName=#{userName} and userAge=#{userAge} ");  **for** (String string : sqlSelectParameter) {  System.***out***.println(string);  }  }  } |

#### 运行效果

|  |
| --- |
| UserDao mapper = SqlSession.*getMapper*(UserDao.**class**);  **int**insertUser = mapper.insertUser(18, "余胜军");  System.***out***.println("影响行数:" + insertUser);  User user = mapper.selectUser("Xiaoxin", 21);  System.***out***.println("查询结果:" + user.getUserName() + "," + user.getUserAge()); |

# 纯手写集合框架

## 集合框架介绍



说明：对于以上的框架图有如下几点说明

1.所有集合类都位于java.util包下。Java的集合类主要由两个接口派生而出：Collection和Map，Collection和Map是Java集合框架的根接口，这两个接口又包含了一些子接口或实现类。

2. 集合接口：6个接口（短虚线表示），表示不同集合类型，是集合框架的基础。

3. 抽象类：5个抽象类（长虚线表示），对集合接口的部分实现。可扩展为自定义集合类。

4. 实现类：8个实现类（实线表示），对接口的具体实现。

5. Collection 接口是一组允许重复的对象。

6. Set 接口继承 Collection，集合元素不重复。

7. List 接口继承 Collection，允许重复，维护元素插入顺序。

8. Map接口是键－值对象，与Collection接口没有什么关系。

9.Set、List和Map可以看做集合的三大类：

List集合是有序集合，集合中的元素可以重复，访问集合中的元素可以根据元素的索引来访问。

Set集合是无序集合，集合中的元素不可以重复，访问集合中的元素只能根据元素本身来访问（也是集合里元素不允许重复的原因）。

Map集合中保存Key-value对形式的元素，访问时只能根据每项元素的key来访问其value。

## 纯手写List框架

List集合代表一个有序集合，集合中每个元素都有其对应的顺序索引。List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。

List接口继承于Collection接口，它可以定义一个允许重复的有序集合。因为List中的元素是有序的，所以我们可以通过使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。

List接口为Collection直接接口。List所代表的是有序的Collection，即它用某种特定的插入顺序来维护元素顺序。用户可以对列表中每个元素的插入位置进行精确地控制，同时可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。实现List接口的集合主要有：ArrayList、LinkedList、Vector、Stack。

### ArrayList底层实现原理

1. Arraylist底层基于数组实现

|  |
| --- |
| **private** Object[] elementData; |

1. Arraylist底层默认数组初始化大小为10个object数组

|  |
| --- |
| **public** ExtArraylist() **throws** Exception {  **this**(10);  }  **public** ExtArraylist(**int**initialCapacity) **throws** Exception {  **if** (initialCapacity< 0) {  **thrownew** IllegalArgumentException("初始容量不能小于0 " + initialCapacity);  }  elementData = **new** Object[initialCapacity];  } |

1. 添加元素后大于当前数组的长度，则进行扩容,将数组的长度增加原来数组的一半。

|  |
| --- |
| // 增大数组空间  **privatevoid** grow(**int**minCapacity) {  // overflow-conscious code  **int**oldCapacity = elementData.length;  **int**newCapacity = oldCapacity + (oldCapacity>> 1); // 在原来容量的基础上加上  // oldCapacity/2  **if** (newCapacity - minCapacity< 0)  newCapacity = minCapacity; // 最少保证容量和minCapacity一样  **if** (newCapacity - ***MAX\_ARRAY\_SIZE***> 0)  newCapacity = *hugeCapacity*(minCapacity); // 最多不能超过最大容量  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.*copyOf*(elementData, newCapacity);  } |

### Vector底层实现原理

Vector是线程安全的，但是性能比ArrayList要低。

ArrayList，Vector主要区别为以下几点：

（1）：Vector是线程安全的，源码中有很多的synchronized可以看出，而ArrayList不是。导致Vector效率无法和ArrayList相比；

（2）：ArrayList和Vector都采用线性连续存储空间，当存储空间不足的时候，ArrayList默认增加为原来的50%，Vector默认增加为原来的一倍；

（3）：Vector可以设置capacityIncrement，而ArrayList不可以，从字面理解就是capacity容量，Increment增加，容量增长的参数。

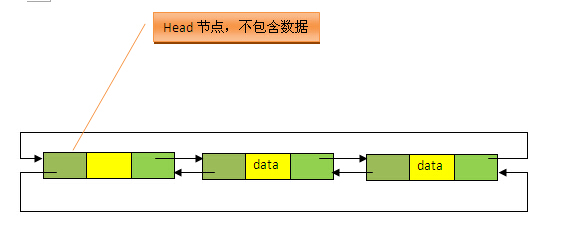
|  |
| --- |
| **privatevoid** grow(**int** minCapacity) {  // overflow-conscious code  **int** oldCapacity = elementData.length;  **int** newCapacity = oldCapacity + (oldCapacity >> 1); //扩充的空间增加原来的50%（即是原来的1.5倍）  **if** (newCapacity - minCapacity < 0) //如果容器扩容之后还是不够，那么干脆直接将minCapacity设为容器的大小  newCapacity = minCapacity;  **if** (newCapacity - MAX\_ARRAY\_SIZE> 0) //如果扩充的容器太大了的话，那么就执行hugeCapacity  newCapacity = hugeCapacity(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.copyOf(elementData, newCapacity);  } |

|  |
| --- |
| /\*\*  \* 自定List接口<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param**<E>  \*/  **publicinterface** ExtList<E> {  **publicvoid** add(E object);  **publicvoid** add(**int**index, E object);  **public** Object remove(**int**index);  **publicboolean** remove(E object);  **publicint** getSize();  **public** Object get(**int**index);  }  /\*\*  \* 纯手写ArrayList<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** ExtArrayList<E>**implements** ExtList<E> {  // 保存ArrayList中数据的数组  **privatetransient** Object[] elementData;  // ArrayList实际数量  **privateint**size;  **public** ExtArrayList() {  // 默认初始容量为10  **this**(10);  }  **public** ExtArrayList(**int**initialCapacity) {  **if** (initialCapacity< 0) {  **thrownew** IllegalArgumentException("Illegal Capacity: " + initialCapacity);  }  // 初始化数组容量  elementData = **new** Object[initialCapacity];  }  // 添加方法实现  **publicvoid** add(Object object) {  ensureExplicitCapacity(size + 1);  elementData[size++] = object;  }  **publicvoid** add(**int**index, Object object) {  rangeCheck(index);  ensureExplicitCapacity(size + 1);  System.*arraycopy*(elementData, index, elementData, index + 1, size - index);  elementData[index] = object;  size++;  }  **publicvoid** ensureExplicitCapacity(**int**minCapacity) {  // 如果存入的数据,超出了默认数组初始容量就开始实现扩容  **if** (size == elementData.length) {  // 获取原来数组的长度 2  **int**oldCapacity = elementData.length;  // oldCapacity >> 1 理解成 oldCapacity/2 新数组的长度是原来长度1.5倍  **int**newCapacity = oldCapacity + (oldCapacity>> 1); // 3  **if** (newCapacity<minCapacity) {  // 最小容量比新容量要小的,则采用初始容量minCapacity  newCapacity = minCapacity;  }  // System.out.println("oldCapacity:" + oldCapacity + ",newCapacity:"  // + newCapacity);  elementData = Arrays.*copyOf*(elementData, newCapacity);  }  }  // public void add(Object object) {  // elementData[size++] = object;  // // 如果存入的数据,超出了默认数组初始容量就开始实现扩容  // if (size == elementData.length) {  // // 新的数组容量  // int newCapacity = 2 \* size;  // // 实现数组扩容  // Object[] newObjct = new Object[newCapacity];  // for (int i = 0; i < elementData.length; i++) {  // newObjct[i] = elementData[i];  // }  // elementData = newObjct;  // }  // }  // 获取数据  **public** Object get(**int**index) {  rangeCheck(index);  **return**elementData[index];  }  **public** Object remove(**int**index) {  Object object = get(index);  **int**numMoved = elementData.length - index - 1;  **if** (numMoved> 0)  System.*arraycopy*(elementData, index + 1, elementData, index, numMoved);  elementData[--size] = **null**;  **return**object;  }  **publicboolean** remove(E object) {  **for** (**int**i = 0; i<elementData.length; i++) {  Object element = elementData[i];  **if** (element.equals(object)) {  remove(i);  **returntrue**;  }  }  **returnfalse**;  }  **privatevoid** rangeCheck(**int**index) {  **if** (index>= size) {  **thrownew** IndexOutOfBoundsException("数组越界啦!");  }  }  **publicint** getSize() {  **return**size;  }  } |

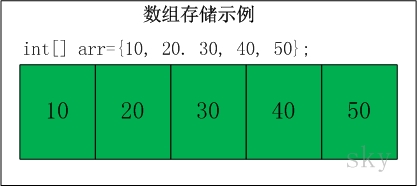
### LinkeList原理

LinkedList 和 ArrayList 一样，都实现了 List 接口，但其内部的数据结构有本质的不同。LinkedList 是基于链表实现的（通过名字也能区分开来），所以它的插入和删除操作比 ArrayList 更加高效。但也是由于其为基于链表的，所以随机访问的效率要比 ArrayList 差。

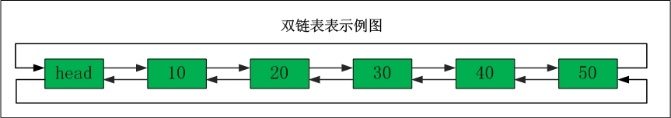
#### LinkedList数据结构原理

LinkedList底层的数据结构是基于双向循环链表的，且头结点中不存放数据,如下：   
   
既然是双向链表，那么必定存在一种数据结构——我们可以称之为节点，节点实例保存业务数据，前一个节点的位置信息和后一个节点位置信息，如下图所示：   


#### 数组和链表结构对比



数组　是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要快速访问数据，很少插入和删除元素，就应该用数组。



链表　中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起，每个结点包括两个部分：一个是存储 数据元素 的　数据域，另一个是存储下一个结点地址的 指针。

　如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表。

##### 内存存储区别

数组从栈中分配空间, 对于程序员方便快速,但自由度小。

链表从堆中分配空间, 自由度大但申请管理比较麻烦.

##### 逻辑结构区别

数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费。

链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项）

##### 总结

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(1)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：

　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；

链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

### 相关代码

|  |
| --- |
| **package** com.itmayiedu;  /\*\*  \* 纯手写 LinkeList<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*  \* **@param**<E>  \*/  **publicclass** LinkeList<E> {  // 第一个元素  **private** Node first;  // 最后一个元素  **private** Node last;  // 实际存放在长度  **privateint**size;  **class** Node {  // 上一个节点  Node prev;  // 节点内容  Object object;  // 下一个节点  Node next;  }  **publicvoid** add(E e) {  // 创建新的节点  Node node = **new** Node();  // 节点内容  node.object = e;  **if** (first == **null**) {  // // 上一个节点  // node.prev = null;  // // 下一个节点  // node.next = null;  // 第一个元素和最后一个元素都是为node  first = node;  } **else** {  // 存放上一个节点内容  node.prev = last;  // 设置上一个节点的next为当前节点  last.next = node;  }  last = node;  size++;  }  **publicvoid** add(**int**index, E e) {  // 1.循环遍历到当前index位置Node  // 2.新增当前节点  Node newNode = **new** Node();  newNode.object = e;  // 获取原来的节点  LinkeList<E>.Node olNode = getNode(index);  // 获取原来上一个节点  LinkeList<E>.Node olNodePrev = olNode.prev;  // 4.新增节点的上一个还是当前Node节点的上一个节点,下一个就是原来的节点  // 原来上一个节点变为当前节点  olNode.prev = newNode;  **if** (olNodePrev == **null**)  first = newNode;  **else**  // 原来上一个节点的下一个节点变为当前节点  olNodePrev.next = newNode;  // 新节点的下一个节点为原来节点  newNode.next = olNode;  size++;  }  **public** E get(**int**index) {  LinkeList<E>.Node node = getNode(index);  **return**(E) node.object;  }  **public** Node getNode(**int**index) {  Node node = **null**;  **if** (first != **null**) {  node = first;  **for** (**int**i = 0; i<index; i++) {  node = node.next;  }  }  **return**node;  }  **publicvoid** remove(**int**index) {  checkElementIndex(index);  LinkeList<E>.Node node = getNode(index);  **if** (node != **null**) {  LinkeList<E>.Node prevNode = node.prev;  LinkeList<E>.Node nextNode = node.next;  // 设置上一个节点的next为当前删除节点的next  **if** (prevNode != **null**) {  prevNode.next = nextNode;  }  // 判断是否是最后一个节点  **if** (nextNode != **null**) {  nextNode.prev = prevNode;  }  }  size--;  }  **publicstaticvoid** main(String[] args) {  LinkeList<String>linkeList = **new** LinkeList<String>();  linkeList.add("张三");  linkeList.add("李四");  linkeList.add("wawngma");  linkeList.add("wangai");  linkeList.add(2, "ll");  **for** (**int**i = 0; i<linkeList.size; i++) {  System.***out***.println(linkeList.get(i));  }  }  **privateboolean** isElementIndex(**int**index) {  **return**index>= 0 &&index<size;  }  **privatevoid** checkElementIndex(**int**index) {  **if** (!isElementIndex(index))  **thrownew** IndexOutOfBoundsException("越界啦!");  }  } |

## 纯手写Map框架

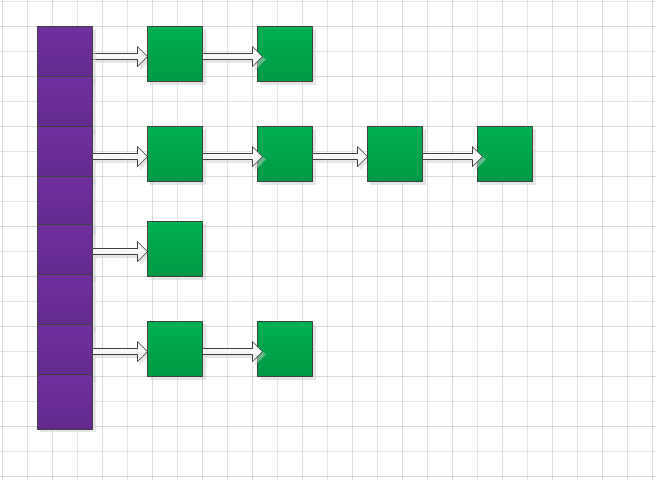
HashMap的介绍

HashMap的实现原理

从底层结构、put和get方法、hash数组索引、扩容机制等几个方面来分析HashMap的实现原理：

### 底层结构

HashMap的底层结构是由数组+链表构成的。



数组（紫色）：hash数组（桶），数组元素是每个链表的头节点

链表（绿色）：解决hash冲突，不同的key映射到了数组的同一索引处，则形成链表。

### put和get方法

put()方法大概过程如下：

如果添加的key值为null，那么将该键值对添加到数组索引为0的链表中，不一定是链表的首节点。

如果添加的key不为null，则根据key计算数组索引的位置：

**数组索引处存在链表，则遍历该链表，如果发现key已经存在，那么将新的value值替换旧的value值**

数组索引处不存在链表，将该key-value添加到此处，成为头节点

get()方法的大概过程：   
1. 如果key为null，那么在数组索引table[0]处的链表中遍历查找key为null的value   
2. 如果key不为null，根据key找到数组索引位置处的链表，遍历查找key的value，找到返回value，若没找到则返回null

### 扩容机制

先看一个例子，创建一个HashMap，初始容量默认为16，负载因子默认为0.75，那么什么时候它会扩容呢？

来看以下公式：

实际容量 = 初始容量 × 负载因子

1

计算可知，16×0.75=12，也就是当实际容量超过12时，这个HashMap就会扩容。

初始容量

当构造一个hashmap时，初始容量设为不小于指定容量的2的次方的一个数（new HashMap(5)， 指定容量为5，那么实际初始容量为8，2^3=8>5），且最大值不能超过2的30次方。

负载因子

负载因子是哈希数组在其容量自动增加之前可以达到多满的一种尺度。（时间与空间的折衷） 当哈希数组中的条目数超出了加载因子与初始容量的乘积时，则要对该哈希数组进行扩容操作（即resize）。

特点：

1\*16=16

0.75\*16=120.5\*16=8 **负载因子越小， 容易扩容—容易扩容的时候 产生新的空数组位置**

**三次扩容**

负载因子1 负载因子0.75 负载因子0.5

1.16\*2=32 16 1.16\*2=32 12 16\*2=32 8

2.32\*2=64 32 2.32\*2=64 24 32\*2=64 16

2.64\*2=12864 3.64\*2=128 48 64\*2=128 32

|  |  |  |
| --- | --- | --- |
| 存放数据10000万个>13毫秒 | 存放数据10000万个 13毫秒 | 存放数据10000万个 21毫秒 |
| 存放数据10000万个 76左右毫秒 | 存放数据10000万个 77左右毫秒 |  |

**负载因子越小，容易扩容，浪费空间，但查找效率高链表特别短 减少hash冲突**

**负载因子越大，不易扩容，对空间的利用更加充分，查找效率低（链表拉长）hash冲突比较多，链表比较长**

扩容过程

HashMap在扩容时，新数组的容量将是原来的2倍，由于容量发生变化，原有的每个元素需要重新计算数组索引Index，再存放到新数组中去，这就是所谓的rehash。

### eqauls方法和hashCode方法

1 如果两个对象相同，那么它们的hashCode值一定要相同。也告诉我们重写equals方法，一定要重写hashCode方法，也就是说hashCode值要和类中的成员变量挂上钩，对象相同–>成员变量相同—->hashCode值一定相同。

2 如果两个对象的hashCode相同，它们并不一定相同，这里的对象相同指的是用eqauls方法比较。

### 基于Linkedlist实现Map

#### 代码实现方式

|  |
| --- |
| /\*\*  \* 1.纯手写HasMap集合 完全采用Arraylist实现 缺点效率低 <br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** ExtArrayListMap<Key, Value> {  List<Entry<Key, Value>>tables = **new** ArrayList<Entry<Key, Value>>();  **publicvoid** put(Key key, Value value) {  // 判断key是否已经重复  EntryexistEntry = getEntry(key);  **if** (existEntry != **null**) {  existEntry.value = value;  **return**;  }  Entryentry = **new** Entry<Key, Value>(key, value);  tables.add(entry);  }  **public** Value get(String key) {  **for** (Entry<Key, Value>entry : tables) {  **if** (entry.key.equals(key)) {  **return**entry.value;  }  }  **returnnull**;  }  **publicvoid** remove(Key key) {  EntryexistEntry = getEntry(key);  **if** (existEntry != **null**) {  tables.remove(existEntry);  }  }  **public** Entry<Key, Value> getEntry(Key key) {  **for** (Entry<Key, Value>entry : tables) {  **if** (entry.key.equals(key)) {  **return**entry;  }  }  **returnnull**;  }  **publicstaticvoid** main(String[] args) {  ExtArrayListMap<String, String>extArrayListMap = **new** ExtArrayListMap<String, String>();  extArrayListMap.put("a", "aaaaa");  extArrayListMap.put("b", "bbb");  extArrayListMap.put("a", "aa");  // extArrayListMap.remove("b");  System.***out***.println(extArrayListMap.get("b"));  }  }  **class** Entry<Key, Value> {  **public** Entry(Key key, Value value) {  **this**.key = key;  **this**.value = value;  }  Key key;  Value value;  }  /\*\*  \* 纯手写HasMap集合(采用 LinkedList)集合实现<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** LinkedListMap<Key, Value> {  // 实际存放Map元素  LinkedList<Entry>[] tables = **new** LinkedList[998];  // 实际Map大小  **privateint**size;  **publicvoid** put(Object key, Object value) {  // 创建entry；  Entry newEntry = **new** Entry(key, value);  // hashCode  **int**hash = getHash(key);  // 判断是否已经在数组重存在  LinkedList<Entry>entrylinkedList = tables[hash];  **if** (entrylinkedList == **null**) {  // 数组中没有存放元素  LinkedList<Entry>linkedList = **new** LinkedList<>();  linkedList.add(newEntry);  tables[hash] = linkedList;  } **else** {  // hashCode 相同情况下存放在链表后面  **for** (Entry entry : entrylinkedList) {  **if** (entry.key.equals(key)) {  // hashCode相同对象也相同  entry.value = value;  } **else** {  // hashCode 相同，但是对象不同。  entrylinkedList.add(newEntry);  }  }  }  size++;  }  **publicint** getHash(Object key) {  **int**hashCode = key.hashCode();  **int**hash = hashCode % tables.length;  **return**hash;  }  **public** Value get(Object key) {  **return**(Value) getEntry(key).value;  }  **public** Entry getEntry(Object key) {  // hashCode  **int**hash = getHash(key);  LinkedList<Entry>listEntry = tables[hash];  **if** (listEntry != **null**) {  **for** (Entry entry : listEntry) {  **if** (entry.key.equals(key)) {  **return**entry;  }  }  }  **returnnull**;  }  **publicstaticvoid**main(String[] args) {  LinkedListMaplinkedListMap = **new** LinkedListMap<String, String>();  linkedListMap.put("a", "644");  linkedListMap.put("b", "123456");  linkedListMap.put("b", "123");  System.***out***.println(linkedListMap.get("b"));  }  }  **class** Entry {  **public** Entry(Object key, Object value) {  **this**.key = key;  **this**.value = value;  }  Object key;  Object value;  } |

### 基于JDK1.7版本实现HashMap

#### 创建Map接口

|  |
| --- |
| /\*\*  \* 纯手写Map集合<br>  \* 作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicinterface** ExtMap<K, V> {  // 向集合中插入数据  **public** V put(K k, V v);  // 根据k 从Map集合中查询元素  **public** V get(K k);  // 获取集合元素个数  **publicint** size();  **interface** Entry<K, V> {  K getKey();  V getValue();  V setValue(V value);  }  } |

#### 创建HashMap实现

|  |
| --- |
| /\*\*  \* 纯手写HashMap集合作者: 每特教育-余胜军<br>  \* 联系方式:QQ644064779|WWW.itmayiedu.com<br>  \*/  **publicclass** ExtHashMap<K, V>**implements** ExtMap<K, V> {  // 1.定义table 存放HasMap 数组元素默认是没有初始化容器懒加载  Node<K, V>[] table = **null**;  // 2. 实际用到table 存储容量大小  **int**size;  // 3.HashMap默认负载因子，负载因子越小，hash冲突机率越低，根据每个链表的个数  **float**DEFAULT\_LOAD\_FACTOR = 0.75f;  // 4.table默认初始大小 16  **staticint***DEFAULT\_INITIAL\_CAPACITY* = 16; // 16  **public**V put(K key, Vvalue) {  // 1.判断table 数组大小是否为空（如果为空的情况下，做初始化操作）  **if** (table == **null**) {  table = **new** Node[*DEFAULT\_INITIAL\_CAPACITY*];  }  // 2.判断数组是否需要扩容实际存储容量=负载因子0.75\*初始容量16 =12 相当于如果长度大于12的时候就需要开始数组扩容  **if** (size>= (DEFAULT\_LOAD\_FACTOR \* *DEFAULT\_INITIAL\_CAPACITY*)) {  System.***out***.println("开始扩容啦！！！！");  resize();  }  // 3.计算hash值指定下标位置  **int**index = getIndex(key, *DEFAULT\_INITIAL\_CAPACITY*);  Node<K, V>node = table[index];  **if** (node == **null**) {  // 没有发生hash冲突问题  node = **new** Node<K, V>(key, value, **null**);  size++;  } **else** {  Node<K, V>newNode = node;  **while** (newNode != **null**) {  // // 已经发生hash冲突问题key 直接添加(冲突node)到前面了不是往后面加  **if** (newNode.getKey().equals(key) || newNode.getKey() == key) {  // hashCodoe 相同，而且equals 相等情况说明是同一个对象修改值  // node.value = value;  **return**newNode.setValue(value);  } **else** {  // 继续添加，排在前面hascode取模余数相同 index 存放在链表或者hashCode 相同但是对象不同  // 新的node 的next 原来的node  **if** (newNode.next == **null**) {  node = **new** Node<K, V>(key, value, node);  size++;  }  }  newNode = newNode.next;  }  }  table[index] = node;  **returnnull**;  }  // hashMap数组扩容机制  **privatevoid** resize() {  // 1.定义新的数组元素  Node[] newTables = **new** Node[*DEFAULT\_INITIAL\_CAPACITY*<< 1];  // 2. 将老的table的key index重新计算下标  **for** (**int**i = 0; i<table.length; i++) {  // 老的Node节点  Node<K, V>oldNode = table[i];  **while** (oldNode != **null**) {  table[i] = **null**;  // 重新计算index 索引下标位置  K oldKey = oldNode.key;  **int**index = getIndex(oldKey, newTables.length);  // 老的next  Nodeoldnext = oldNode.next;  // 判断newTables数组中,是否存在下标相同，如果下标相同则存放在原来的.next  oldNode.next = newTables[index];  // 将原来的node赋值给新的node  newTables[index] = oldNode;  // 获取下一个节点,判断是否继续循环  oldNode = oldnext;  }  }  // 3.将newTable赋值给table;  table = newTables;  *DEFAULT\_INITIAL\_CAPACITY* = newTables.length;  newTables = **null**;// 将对象变为不可达对象  }  // hasMap数组扩容  // private void resize() {  //  // // 1.定于新数组（）按照两倍进行扩容 // 1.使用取模算法定位数组链表 DEFAULT\_INITIAL\_CAPACITY)\*2  // Node[] newTables = new Node[DEFAULT\_INITIAL\_CAPACITY << 1];  // // 2.hashCode不会发生改变.但是index会发生改变,重新计算index 重新取模  // for (int i = 0; i < table.length; i++) {  // Node<K, V> node = table[i];  // while (node != null) {  // table[i] = null;  // int index = getIndex(node.key, newTables.length);  // // 获取下一个节点  // ExtHashMap<K, V>.Node<K, V> next = node.next;  // // 判断是否有下一个节点  // node.next = newTables[index];  // // 将之前node 计算的inde下标存放在 newTables  // newTables[index] = node;  // // 判断是否继续循环  // node = next;  // }  //  // }  // // 将新的tables 赋值改老的table  // table = newTables;  // DEFAULT\_INITIAL\_CAPACITY = newTables.length;  // newTables = null;// 变为不可达对象  // }  // 测试方法.打印所有的链表元素  **void** print() {  **for** (**int**i = 0; i<table.length; i++) {  Node<K, V>node = table[i];  System.***out***.print("下标位置[" + i + "]");  **while** (node != **null**) {  System.***out***.print("[ key:" + node.getKey() + ",value:" + node.getValue() + "]");  node = node.next;  // if (node.next != null) {  // node = node.next;  // } else {  // // 结束循环  // node = null;  // }  }  System.***out***.println();  }  }  **publicint** getIndex(K k, **int**length) {  // System.out.println("k:" + k + ",hashCode=" + hashCode);  **int**index = k == **null** ? 0 : k.hashCode() % length;  **return**index;  }  **public**V get(K k) {  Node<K, V>node = getNode(table[getIndex(k, *DEFAULT\_INITIAL\_CAPACITY*)], k);  **return**node == **null** ? **null** : node.value;  }  **public** Node<K, V> getNode(Node<K, V>node, K k) {  **while** (node != **null**) {  **if** (node.getKey().equals(k)) {  **return**node;  }  node = node.next;  }  **returnnull**;  }  **publicint** size() {  **return**size;  }  // 定义节点  **class** Node<K, V>**implements** Entry<K, V> {  // 存放Map 集合 key  **private** K key;  // 存放Map 集合 value  **private** V value;  // 下一个节点Node  **private** Node<K, V>next;  **public** Node(K key, V value, Node<K, V>next) {  **super**();  **this**.key = key;  **this**.value = value;  **this**.next = next;  }  **public** K getKey() {  **returnthis**.key;  }  **public** V getValue() {  **returnthis**.value;  }  **public** V setValue(V value) {  // 设置新值的返回老的值  V oldValue = **this**.value;  **this**.value = value;  **return**oldValue;  }  }  } |

### HashMap 核心源码部分的分析

1. HashMap只允许一个为null的key。

2. HashMap的扩容：当前table数组的两倍

3. HashMap实际能存储的元素个数： capacity \* loadFactor

4. HashMap在扩容的时候，会重新计算hash值，并对hash的位置进行重新排列，因此，为了效率，尽量给HashMap指定合适的容量，避免多次扩容

## Java运算符

<< : 左移运算符，num << 1,相当于num乘以2

>> : 右移运算符，num >> 1,相当于num除以2

>>> : 无符号右移，忽略符号位，空位都以0补齐

## 数组拷贝

Arrays.copyOf功能是实现数组的复制，返回复制后的数组。参数是被复制的数组和复制的长度

System.arraycopy System.arraycopy方法：如果是数组比较大，那么使用System.arraycopy会比较有优势，因为其使用的是内存复制，省去了大量的数组寻址访问等时间

复制指定源数组src到目标数组dest。复制从src的srcPos索引开始，复制的个数是length，复制到dest的索引从destPos开始。

src:源数组； srcPos:源数组要复制的起始位置；

dest:目的数组； destPos:目的数组放置的起始位置； length:复制的长度。

注意：src and dest都必须是同类型或者可以进行转换类型的数组．

有趣的是这个函数可以实现自己到自己复制，比如：

int[] fun ={0,1,2,3,4,5,6};

System.arraycopy(fun,0,fun,3,3);

则结果为：{0,1,2,0,1,2,6};

实现过程是这样的，先生成一个长度为length的临时数组,将fun数组中srcPos

到srcPos+length-1之间的数据拷贝到临时数组中，再执行System.arraycopy(临时数组,0,fun,3,3).