FRAMEWORK

一图胜百文

(重点在原理而非配置)

[1. MYBATIS 框架核心基础 1-3](#_Toc18428645)

[1.1. MYBATIS 架构分析 1-3](#_Toc18428646)

[1.1.1. Mybatis 应用架构(重点) 1-3](#_Toc18428647)

[1.1.2. Mybatis 产品架构 1-4](#_Toc18428648)

[1.1.3. MyBatis 技术架构 1-5](#_Toc18428649)

[1.2. MYBATIS 环境配置实践 1-7](#_Toc18428650)

[1.2.1. 初始化数据环境 1-7](#_Toc18428651)

[1.2.2. 创建并配置项目 1-7](#_Toc18428652)

[1.2.3. 项目环境单元测试 1-9](#_Toc18428653)

[1.3. MYBATIS业务应用快速实践 1-10](#_Toc18428654)

[1.3.1. 基于业务实现 1-10](#_Toc18428655)

[1.3.2. 基本业务进阶实现 1-16](#_Toc18428656)

[1.3.3. 会话工厂对象创建增强(了解) 1-19](#_Toc18428657)

[1.4. MYBATIS 应用原理进阶分析 1-20](#_Toc18428658)

[1.4.1. 会话工厂创建分析(了解) 1-20](#_Toc18428659)

[1.4.2. 会话对象应用分析 (了解) 1-21](#_Toc18428660)

[1.4.3. 基于Mapper接口会话(了解) 1-22](#_Toc18428661)

[1.4.4. 缓存应用实现过程分析(掌握) 1-23](#_Toc18428662)

[2. SPRING 框架核心进阶 2-25](#_Toc18428663)

[1.1. SPRING 框架架构分析 2-25](#_Toc18428664)

[1.1.1. Spring 框架应用架构 2-25](#_Toc18428665)

[1.1.2. Spring 框架产品架构 2-25](#_Toc18428666)

[1.1.3. Spring 框架技术架构 2-26](#_Toc18428667)

[1.2. SPRING框架快速实践 (注解方式-脱离文档) 2-27](#_Toc18428668)

[1.2.1. Spring项目创建及配置 2-27](#_Toc18428669)

[1.2.2. Spring基本测试环境创建 2-28](#_Toc18428670)

[1.2.3. Spring项目基本业务实现 2-29](#_Toc18428671)

[1.2.4. Spring项目课堂练习分析及实现 2-30](#_Toc18428672)

[1.3. SPRING IOC 应用原理进阶分析 2-32](#_Toc18428673)

[1.3.1. Spring IOC设计思想分析 2-32](#_Toc18428674)

[1.3.2. Spring Bean 工厂的初始化 2-32](#_Toc18428675)

[1.3.3. Spring 中的两大map对象分析 2-32](#_Toc18428676)

[1.3.4. Spring 中两大bean对象分析(了解) 2-33](#_Toc18428677)

[1.3.5. Spring 中两大bean对象描述方式 2-34](#_Toc18428678)

[2.1. Spring 依赖注入实践增强 2-35](#_Toc18428679)

[2.1.1. Spring 中Bean对象依赖注入分析(重点) 2-35](#_Toc18428680)

[2.1.2. Spring 中Bean对象依赖注入实践 2-36](#_Toc18428681)

[3. SPRING 框架MVC模块进阶 3-38](#_Toc18428682)

[3.1. Spring MVC 设计思想 3-38](#_Toc18428683)

[3.1.1. Spring MVC 设计思想分析 3-38](#_Toc18428684)

[3.1.2. Spring MVC 核心对象及流程分析 3-38](#_Toc18428685)

[3.2. Spring MVC 快速实践 3-39](#_Toc18428686)

[3.2.1. xml方式配置实现(脱离文档) 3-39](#_Toc18428687)

[3.2.2. 注解方式配置实现（脱离文档） 3-42](#_Toc18428688)

[3.3. Spring MVC 请求响应处理增强分析 3-45](#_Toc18428689)

[3.3.1. 请求处理增强分析及实现 3-45](#_Toc18428690)

[3.3.2. 响应处理增强分析实现 3-46](#_Toc18428691)

[2. SSM 综合案例实现(注解方式) 3-47](#_Toc18428692)

[2.1. 业务描述 3-47](#_Toc18428693)

[2.2. 初始化化数据环境 3-47](#_Toc18428694)

[2.3. 项目创建及配置 3-48](#_Toc18428695)

[2.4. 项目资源整合实现 3-49](#_Toc18428696)

[2.4.1. 配置架构分析及实现 3-49](#_Toc18428697)

[2.4.2. 整合连接池对象 3-51](#_Toc18428698)

[2.4.3. 整合mybatis框架 3-51](#_Toc18428699)

[2.4.4. 整合Spring MVC 模块 3-52](#_Toc18428700)

[2.5. 业务设计分析及实现 3-52](#_Toc18428701)

[2.5.1. 业务架构 3-52](#_Toc18428702)

[2.5.2. POJO实现 3-53](#_Toc18428703)

[2.5.3. Dao实现 3-54](#_Toc18428704)

[2.5.4. Service实现 3-54](#_Toc18428705)

[2.5.5. Controller实现 3-55](#_Toc18428706)

[2.6. 项目部署及运行分析 3-56](#_Toc18428707)

[3. FRAMEWORK总结分析 3-56](#_Toc18428708)

[3.1. 重点难点 3-56](#_Toc18428709)

[3.2. FAQ分析 3-57](#_Toc18428710)

[3.3. BUG 分析 3-57](#_Toc18428711)

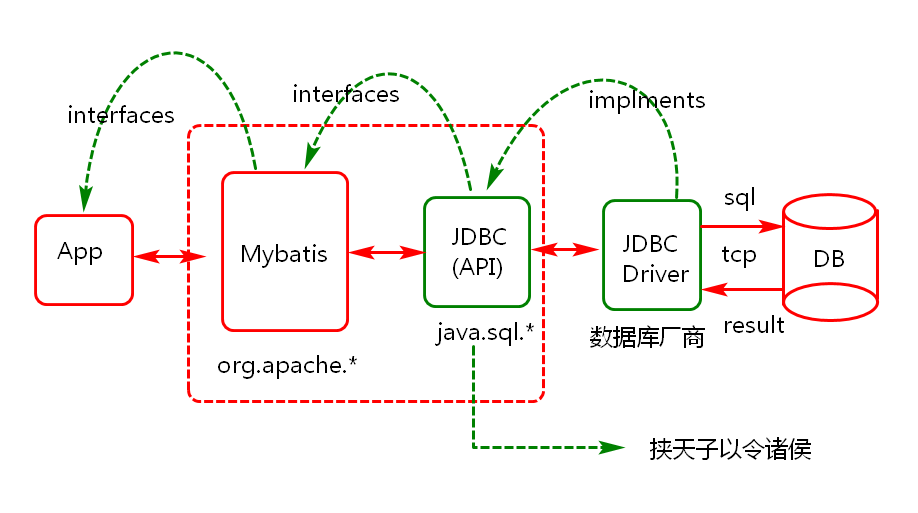
# MYBATIS 框架核心基础

## MYBATIS 架构分析

### Mybatis 应用架构(重点)

谈谈对mybatis的应用架构的理解？

MyBatis 是一个优秀的持久层框架，实现了对JDBC操作（标准API）的封装，主要用于传统简化JDBC操作中的一些相对繁琐的步骤，例如参数的映射，结果集的映射（数据库中记录存储到内存中的对象中）等。



为何使用mybatis实现数据持久层应用？

1. 第一稳定，灵活（动态SQL），功能强大(池，日志，缓存)
2. 学习以及使用成本低
3. 解耦，SQL的可维护性，可复用性比较高。

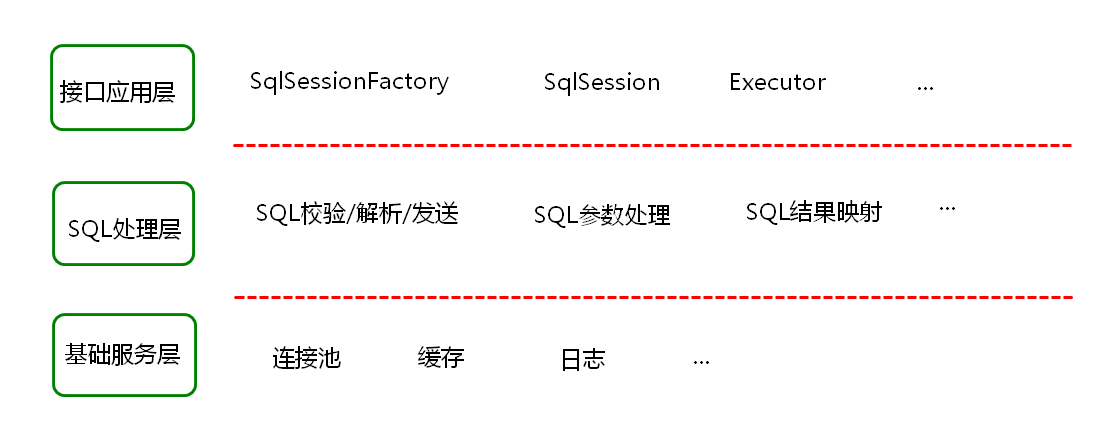
说明:mybatis官网参考

1. [mybatis.org/mybatis-3](http://www.mybatis.org/mybatis-3)
2. github.com/mybatis
3. <http://www.mybatis.org/spring/>

### Mybatis 产品架构

说说对mybatis的产品架构的理解？

所有框架都要解决一些共性问题（持久化），都是一种半成品，mybatis也不例外，它作为一种框架，它要解决相关问题，如何解决问题？采用怎样的架构解决问题，这是我们要学习的一个点。

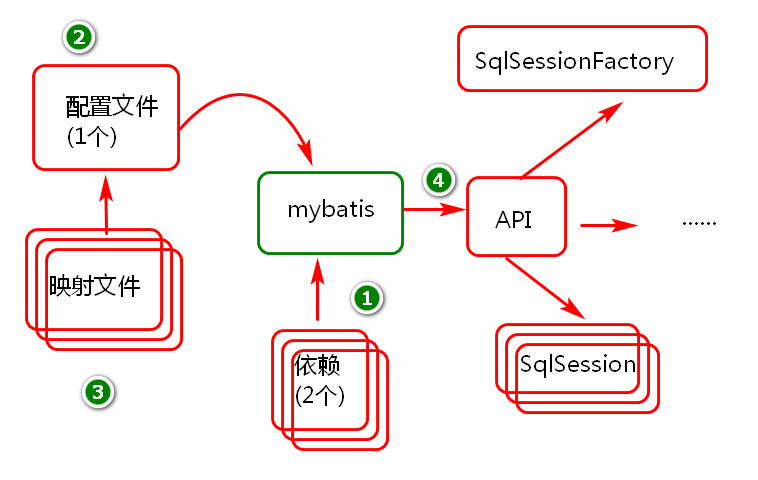


思考:mybatis作为一个持久层框架,应该解决哪些功能性问题?

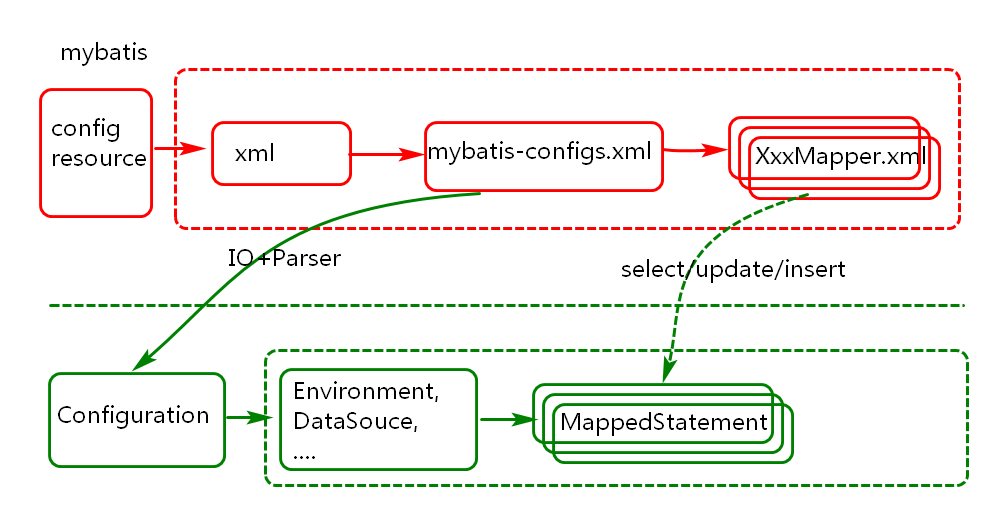
1. 会话功能 (SqlSession)
2. 会话语言 (SQL,动态SQL)
3. 会话协议 (TCP)
4. 用户体验 (连接池,缓存,日志)

### MyBatis 技术架构

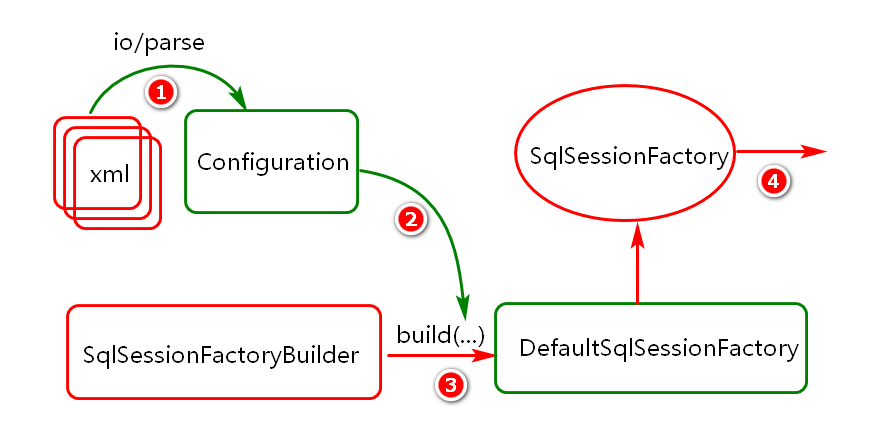
Mybatis 框架"构成"分析,如下图所示:



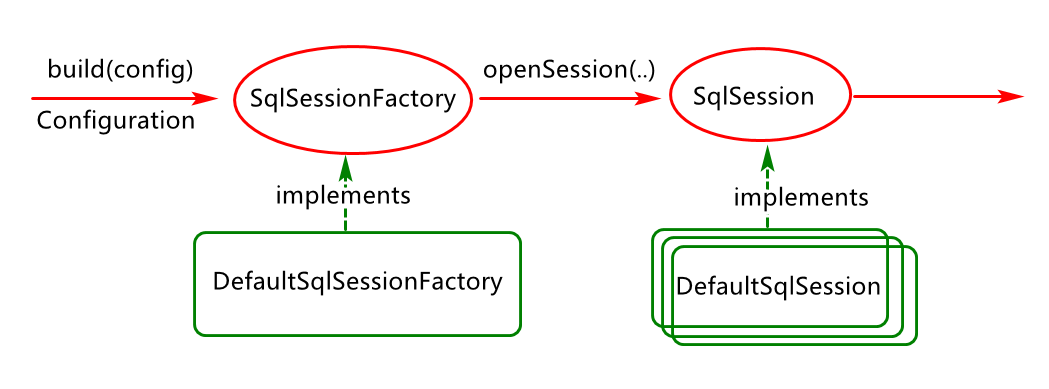
资源配置技术架构,如下图所示:



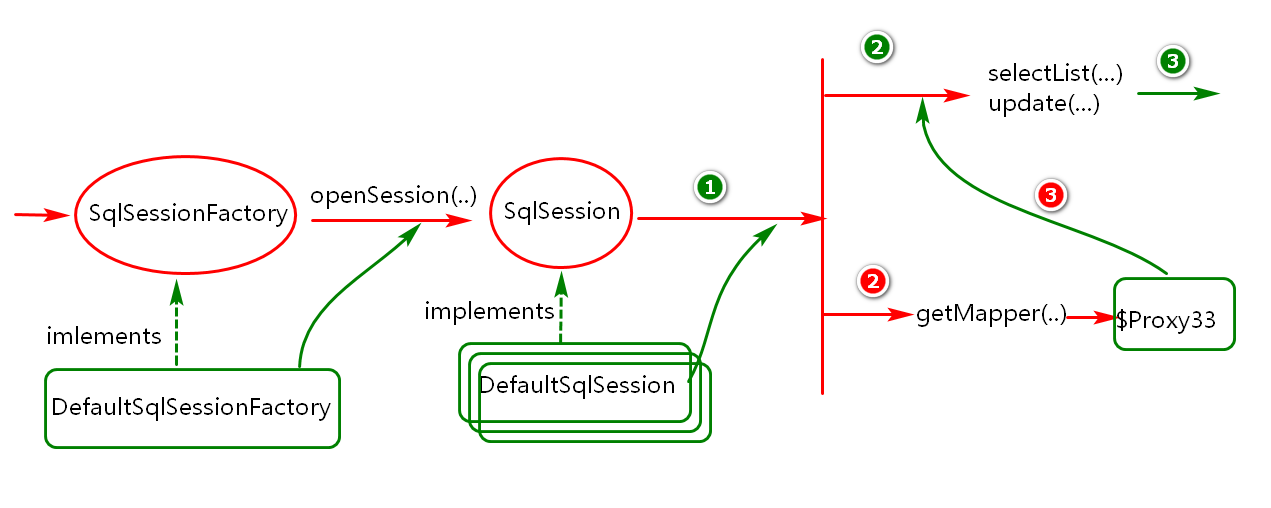
会话工厂对象创建(说说SqlSessionFactory对象的创建过程),如下图所示:



会话对象创建？(说说sqlsession对象创建过程),如下图所示:



会话对象应用方式,如下图所示:



## MYBATIS 环境配置实践

### 初始化数据环境

本次数据初始化直接在mysql的命令行执行:

1. 启动系统命令行控制台
2. 登陆数据库:mysql -u root -p
3. 设置客户端编码：set names utf8
4. 导入数据：source d:/xxx.sql

说明：查询时，假如有中文要显示，可先执行set names gbk.

### 创建并配置项目

1. 打开IDE并进行配置(新的工作区)
2. 配置工作区编码(utf-8)
3. 配置maven环境(本地库,私服配置)
4. 创建maven项目并添加依赖

项目名称:CGB-MYBATIS-01

打包方式:jar包

组id:com.cy

MySQL驱动依赖(假如驱动版本与当前数据库不一致可能会有问题)

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.17</version>

</dependency>

Mybatis 框架依赖（参考官方 mybatis.org/mybatis-3）

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.5.2</version>

</dependency>

Junit单元测试依赖

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

说明:学了spring boot以后还有一种测试方式

1. 配置项目

在src/main/resources目录下创建mybatis-configs.xml配置文件如下：

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<!-- mybatis 核心配置 -->

<configuration>

<!-- 配置初始化环境(连接) -->

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<!-- 使用mybatis自带连接池 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql:///dbgoods?serverTimezone=GMT"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"123456"*/>

</dataSource>

</environment>

</environments>

</configuration>

其模板：参考官方配置实现（Getting start ）

### 项目环境单元测试

在测试包中，创建测试类，测试是否可以获取与数据库的连接

**public** **class** TestBase {

/\*\*

\* 借助此对象创建SqlSession(通过此对象

\* 实现与数据库之间的会话)

\*/

**protected** SqlSessionFactory factory;

/\*\*

\* 此方会在@Test注解修饰的方法之前执行,

\* 通常用于做一些初始化操作(方法名自己定义)

\*/

@Before

**public** **void** init()**throws** IOException{

InputStream in=

Resources.*getResourceAsStream*("mybatis-configs.xml");

factory=**new** SqlSessionFactoryBuilder().build(in);

//系统底层建造者模式构建工厂对象(此对象构建过程相对复杂)

System.***out***.println(factory);

}

@Test

**public** **void** testSqlSessionConnection(){

SqlSession session=factory.openSession();

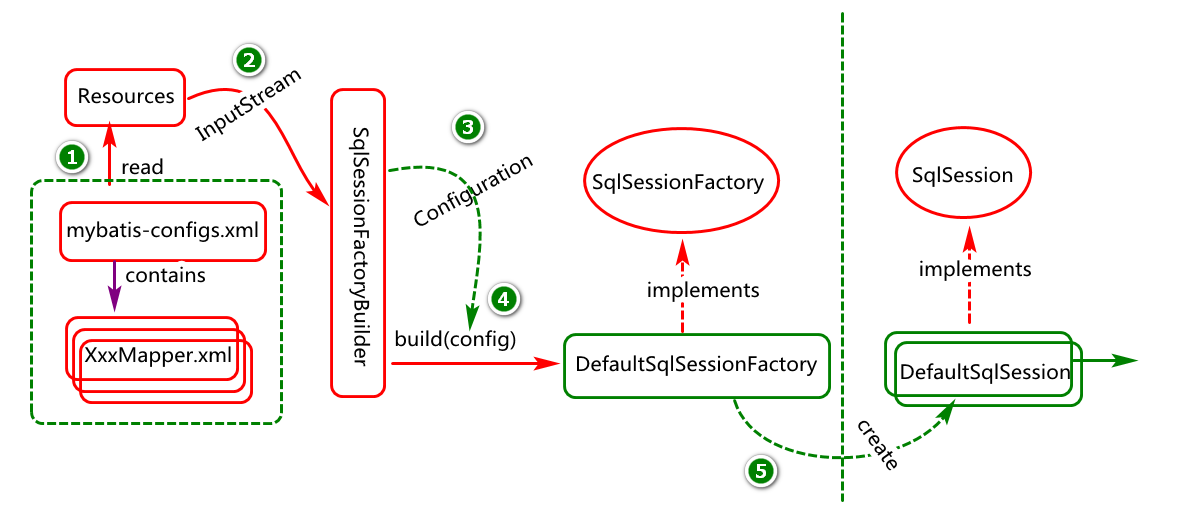
Connection conn=session.getConnection();

System.***out***.println(conn);

}

}

API 创建过程分析:



## MYBATIS业务应用快速实践

参考官网：[www.mybatis.org/mybatis-3](http://www.mybatis.org/mybatis-3) 实现对商品模块数据的CRUD操作。

### 基于业务实现

* **添加业务实现(以添加商品信息为例进行实现)**

第一步：编写pojo对象(com.cy.pj.goods.pojo.Goods),

借助此对象存储内存中的数据,最后将内存中的数据持久化到数据库.

**package** com.cy.pj.goods.pojo;

**import** java.io.Serializable;

**import** java.util.Date;

/\*\*

\* POJO:借助此对象在内存中封装商品信息

\*/

**public** **class** Goods **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 6239917530570596544L;

**private** Long id;

**private** String name;

**private** String remark;

**private** Date createdTime;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getRemark() {

**return** remark;

}

**public** **void** setRemark(String remark) {

**this**.remark = remark;

}

**public** Date getCreatedTime() {

**return** createdTime;

}

**public** **void** setCreatedTime(Date createdTime) {

**this**.createdTime = createdTime;

}

}

第二步:编写映射文件(mapper/GoodsMapper.xml),实现内存中对象(一般是pojo对象)与数据库表的映射(例如属性与字段进行映射)

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"com.cy.pj.goods.dao.GoodsDao"*>

<insert id=*"insertObject"*

parameterType=*"com.cy.pj.goods.pojo.Goods"*>

insert into tb\_goods

(id,name,remark,createdTime)

values

(#{id},#{name},#{remark},now())

</insert>

</mapper>

第三步:注册映射文件(mybatis-configs.xml)，在配置文件中添加如下语句：

<mappers>

<mapper resource=*"mapper/GoodsMapper.xml"*/>

</mappers>

第四步:定义单元测试类(TestGoodsDao)及方法进行单元测试

**package** com.test;

**import** org.apache.ibatis.session.SqlSession;

**import** org.junit.Test;

**import** com.cy.pj.goods.pojo.Goods;

**public** **class** TestGoodsDao01 **extends** TestBaseWithXml {

@Test

**public** **void** testInsertObject() {

//1.创建Goods对象

Goods g=**new** Goods();

g.setId(14L);

g.setName("IO");

g.setRemark("IO...");

//2.将Goods对象写入到数据库

//2.1获取SqlSession对象

SqlSession session=

sqlSessionFactory.openSession();

//2.2将对象持久化

String statement="com.cy.pj.goods.dao.GoodsDao.insertObject";

**int** rows=session.insert(statement, g);

System.***out***.println("insert.rows=+"+rows);

//2.3提交事务

session.commit();

//2.4释放资源

session.close();

}

* **修改业务实现(以修改商品信息为例进行实现)**

第一步：修改映射文件(mapper/GoodsMapper.xml),添加update元素实现信息修改操作语句的定义.

<update id=*"updateObject"*>

update tb\_goods

set name=#{name},

remark=#{remark}

where id=#{id}

</update>

第二步:修改单元测试类(TestGoodsDao01),添加修改测试方法.

@Test

**public** **void** testUpdateObject() {

//1.创建Goods对象

Goods g=**new** Goods();

g.setId(14L);

g.setName("Thread");

g.setRemark("Thread...");

//2.将Goods对象写入到数据库

//2.1获取SqlSession对象

SqlSession session=

sqlSessionFactory.openSession();

//2.2将对象持久化

String statement="com.cy.pj.goods.dao.GoodsDao.updateObject";

**int** rows=session.update(statement, g);

System.***out***.println("update.rows=+"+rows);

//2.3提交事务

session.commit();

//2.4释放资源

session.close();

}

}

1. 删除业务实现(以删除作者信息为例进行实现)

step01:修改映射文件(mapper/GoodsMapper.xml),添加delete元素,实现基于id进行删除操作语句的定义.

<delete id=*"deleteById"*>

delete from tb\_goods

where id=#{id}

</delete>

step02:修改单元测试类(TestGoodsDao01),添加删除测试方法

@Test

**public** **void** testDeleteById() {

//1.获取session对象

SqlSession session =

sqlSessionFactory.openSession();

//2.执行删除操作

String statement=

"com.cy.pj.goods.dao.GoodsDao.deleteById";

**try** {

**int** rows=session.delete(statement, 10);

System.***out***.println("delete.rows="+rows);

//3.提交事务

session.commit();

//4.释放资源

}**finally** {

session.close();

}

}

1. 查询业务实现(以查询某个商品信息为例进行实现)

step01:修改映射文件(mapper/GoodsMapper.xml),添加select元素,实现基于id查询作者信息语句的定义.

<select id=*"selectById"*

resultType=*"com.cy.pj.goods.pojo.Goods"*>

select id,name,remark,createdTime

from tb\_goods

where id=#{id}

</select>

step02:修改单元测试类(TestGoodsDao01),添加基于id执行查询的操作

@Test

**public** **void** testSelectById() {

//1.获取session对象

SqlSession session =

sqlSessionFactory.openSession();

//2.执行删除操作

String statement=

"com.cy.pj.goods.dao.GoodsDao.selectById";

**try** {

Goods g=session.selectOne(statement,11);

System.***out***.println(g);

//3.提交事务

session.commit();

//4.释放资源

}**finally** {

session.close();

}

}

1. 分页查询实现(以查询多个商品信息为例进行实现)

step01:修改映射文件(mapper/GoodsMapper.xml),添加select元素,实现基于分页条件查询作者信息语句的定义.

<select id=*"findPageObjects"*

resultType=*"com.cy.pj.goods.pojo.Goods"*>

select \*

from tb\_goods

limit #{startIndex},#{pageSize}

</select>

step02:修改单元测试类(TestGoodsDao01),添加基于分页条件执行查询.

@Test

**public** **void** testFindPageObjects() {

//1.获取session对象

SqlSession session =

sqlSessionFactory.openSession();

//2.执行删除操作

String statement=

"com.cy.pj.goods.dao.GoodsDao.findPageObjects";

**try** {

Map<String,Object> map=**new** HashMap<String,Object>();

map.put("startIndex", 0);

map.put("pageSize", 3);

List<Goods> list=

session.selectList(statement,map);

System.***out***.println(list);

//3.提交事务

session.commit();//connection.commint

//4.释放资源

}**finally** {

session.close();

}

}

­­­­­­­

1. 删除多条数据(以删除商品信息为例进行删除,借助动态sql)

step01:修改映射文件(mapper/GoodsMapper.xml),添加delete元素,实现基于多个ID删除作者信息的语句的定义.

<delete id=*"deleteObjects"*>

delete from tb\_goods

<where>

<!-- foreach元素用于迭代一个数组或集合 -->

<foreach collection=*"array"*

item=*"id"*>

or id=#{id}

</foreach>

</where>

</delete>

step02:修改单元测试类(TestGoodsDao01),添加基于多个ID执行删除的操作.

@Test

**public** **void** testDeleteObjects() {

//1.获取session对象

SqlSession session =

sqlSessionFactory.openSession();

//2.执行删除操作

String statement=

"com.cy.pj.goods.dao.GoodsDao.deleteObjects";

**try** {

**int** rows=session.delete(statement, **new** Object[] {13,3});

System.***out***.println("delete.rows="+rows);

//3.提交事务

session.commit();//connection.commint

//4.释放资源

}**finally** {

session.close();

}}

### 基本业务进阶实现

基于接口方式实现数据访问操作

1. 分页查询实现(以查询多个作者信息为例进行实现)

第一步: 创建一个接口GoodsDao 定一个分页查询方法

**package** com.cy.pj.goods.dao;

**import** java.util.List;

**import** org.apache.ibatis.annotations.Param;

**import** com.cy.pj.goods.pojo.Goods;

**public** **interface** GoodsDao {

/\*\*

\* 执行数据的删除操作

\* **@param** ids

\* **@return**

\*/

**int** deleteObjects(**int**... ids);

}

第二步: 创建一个单元测试类TestGoodsDao02对分页查询进行单元测试

**public** **class** TestGoodsDao02 **extends** TestBaseWithXml {

@Test

**public** **void** testDeleteObjects() {

//1.获取session对象

SqlSession session =

sqlSessionFactory.openSession();

**try** {

//2.执行删除操作

GoodsDao gDao=

session.getMapper(GoodsDao.**class**);

**int** rows=gDao.deleteObjects(11,12);

System.***out***.println("delete.rows="+rows);

//3.提交事务

//session.commit();//connection.commint

//4.释放资源

}**finally** {

session.close();

}

}

}

1. 基于多个id进行删除 (以删除作者信息为例进行实现)

第一步：修改GoodsDao接口,添加deleteObjects方法

**int** deleteObjects(**int**... ids);

第二步: 修改单元测试类,添加testDeleteObjects方法实现单元测试.

@Test

**public** **void** testDeleteObjects() {

//1.获取session对象

SqlSession session =

sqlSessionFactory.openSession();

**try** {

//2.执行删除操作

GoodsDao gDao=

session.getMapper(GoodsDao.**class**);

**int** rows=gDao.deleteObjects(11,12);

System.***out***.println("delete.rows="+rows);

//3.提交事务

//session.commit();//connection.commint

//4.释放资源

}**finally** {

session.close();

}

}

3）基于注解(Annotation)方式做业务实现?

第一步：在GoodsDao接口中添加统计记录行数的方法：getRowCount()

int getRowCount();

第二步：在GoodsDao接口的统计记录方法上添加@Select注解，定义查询语句。

@Select("select count(\*) from tb\_goods")

int getRowCount();

第三步：在TestGoodsDao02测试类中添加测试方法。

@Test

**public** **void** testGetRowCount() {

SqlSession session=

sqlSessionFactory.openSession();

**try** {

GoodsDao dao=

session.getMapper(GoodsDao.**class**);

**int** rowCount=dao.getRowCount();

System.***out***.println(rowCount);

session.commit();

}**finally** {

session.close();

}

}

### 会话工厂对象创建增强(了解)

基于非XMl方式构建SqlSessionFactory对象?

**package** com.test;

**import** com.cy.pj.goods.dao.GoodsDao;

/\*\*

\* Building SqlSessionFactory without XML

\*/

**public** **class** TestBaseWithJava {

**protected** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** init() {

//1.构建数据源对象

PooledDataSource dataSource=**new** PooledDataSource();

dataSource.setDriver("com.mysql.cj.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///dbgoods?serverTimezone=GMT&characterEncoding=utf8");

dataSource.setUsername("root");

dataSource.setPassword("root");

//2.创建事务管理工厂

JdbcTransactionFactory transactionFactory=

**new** JdbcTransactionFactory();

//3.创建一个环境对象

Environment env= **new** Environment("development",

transactionFactory, dataSource);

//4.创建配置对象

Configuration config=

**new** Configuration(env);

config.addMapper(GoodsDao.**class**);

//5.创建sqlSessionFactory对象

sqlSessionFactory=

**new** SqlSessionFactoryBuilder()

.build(config);

}

@Test

**public** **void** testConnection(){

Connection conn=

sqlSessionFactory.openSession()

.getConnection();

System.***out***.println(conn);

}

}

测试实现

**package** com.test;

**import** org.apache.ibatis.session.SqlSession;

**import** org.junit.Test;

**import** com.cy.pj.goods.dao.GoodsDao;

/\*\*

\* 基于接口方式进行测试实现

\*/

**public** **class** TestGoodsDao03

**extends** TestBaseWithJava {

@Test

**public** **void** testGetRowCount() {

SqlSession session=

sqlSessionFactory.openSession();

**try** {

GoodsDao dao=

session.getMapper(GoodsDao.**class**);

**int** rowCount=dao.getRowCount();

System.***out***.println(rowCount);

session.commit();

}**finally** {

session.close();

}

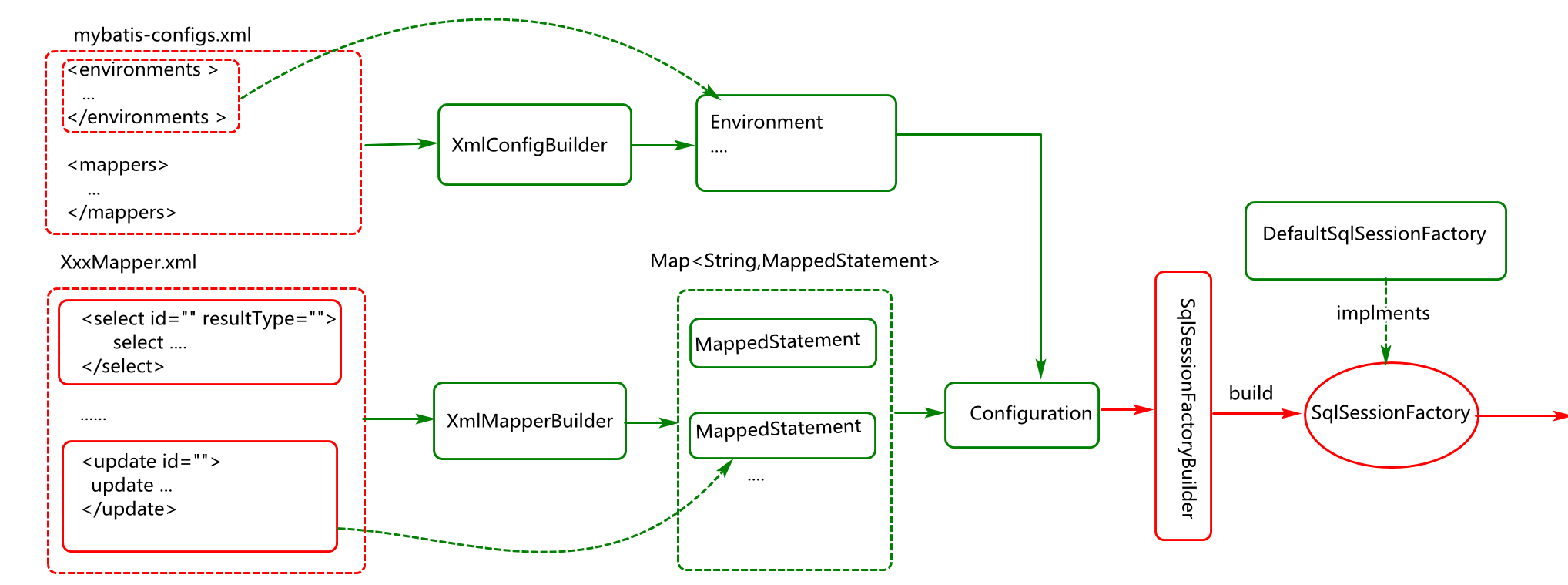
}

}

## MYBATIS 应用原理进阶分析

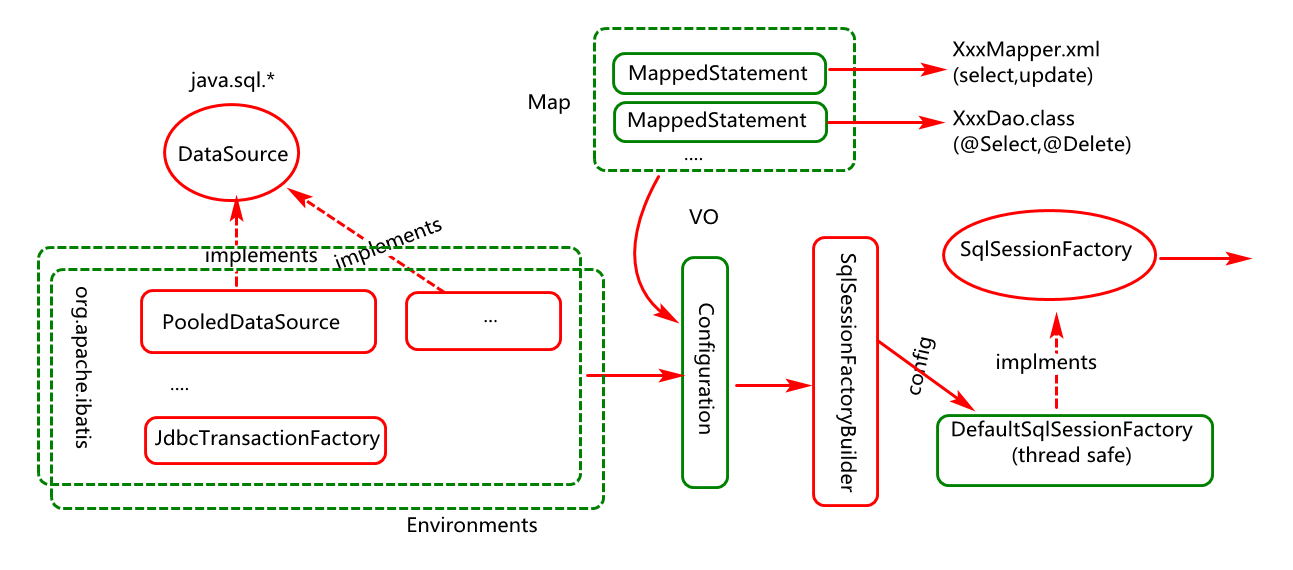
### 会话工厂创建分析(了解)

会话工厂(SqlSessionFactory)创建过程分析:基于xml方式



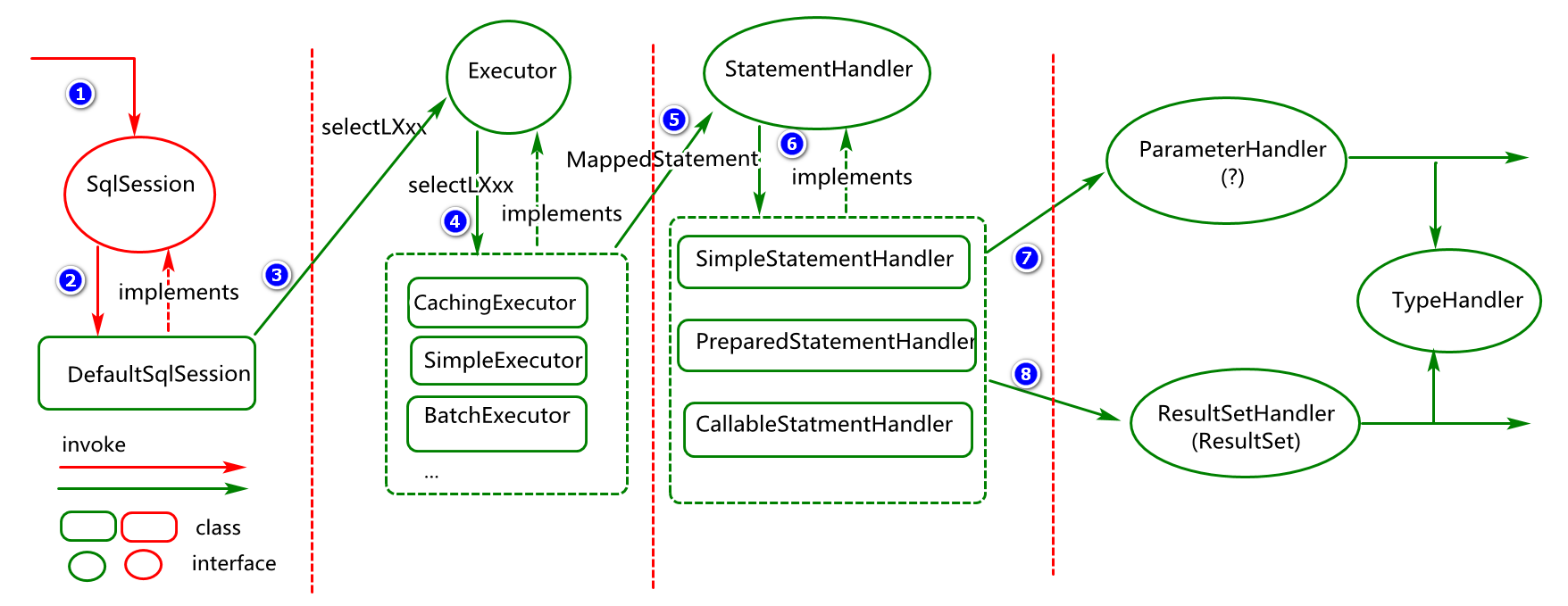
* + - 1. 通过IO读取配置文件
      2. 解析IO数据并进行封装(所有信息都会存储到Configuration对象)
      3. 基于配置对象创建SqlSessionFactory对象

会话工厂(SqlSessionFactory)创建过程分析:不基基于xml方式



### 会话对象应用分析 (了解)

说说mybatis对jdbc的封装过程（了解）？（SqlSession应用增强分析）



1)封装了连接获取过程？（Executor->JDBCTransaction🡪Connection）

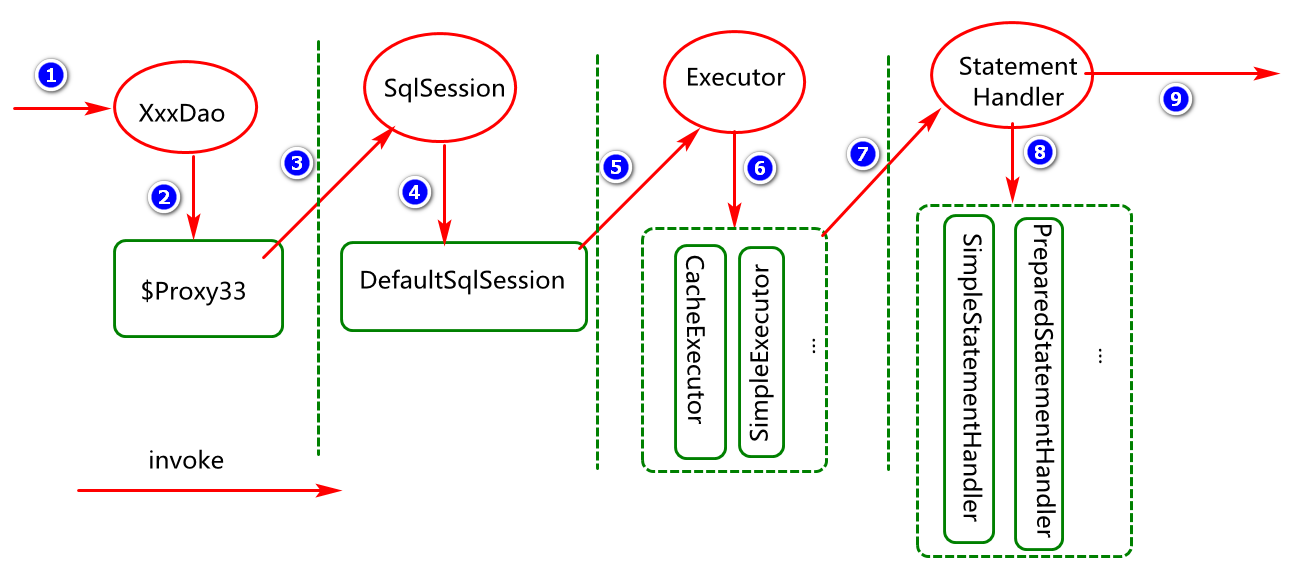
2)封装了Statement对象创建过程？(connection)

3)封装了sql的发送过程，参数的处理过程，结果的映射过程。

### 基于Mapper接口会话(了解)

例如：XxxMapper xm=session.getMapper(XxxDao.class);

当获取到mapper接口对应的实现类对象以后可以基于实现类底层执行sql操作。



FAQ?

1. 不同通过getMapper方式获取指定接口实现类是否可以基于mybatis实现数据访问操作?(可以)
2. 基于Mapper接口访问数据库库的优势,劣势?(解耦,可读性好但是性能相对较差,相对于直接通过sqlsession的selectList,update方法而言).

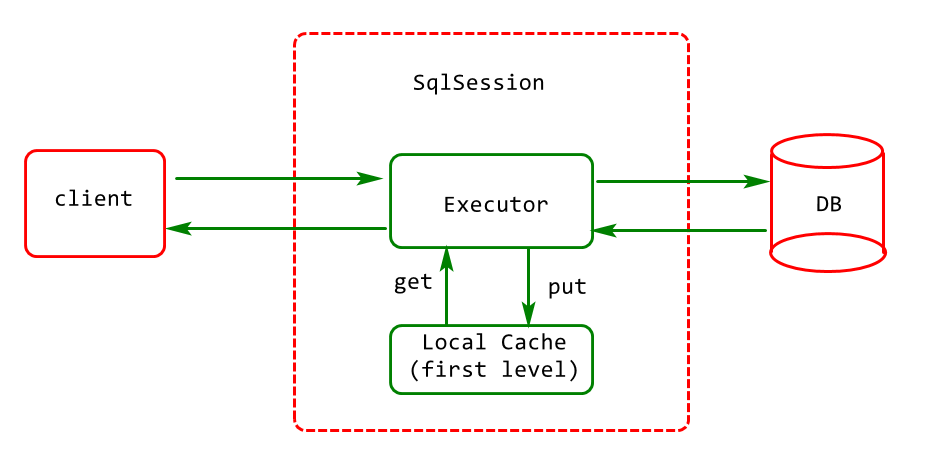
### 缓存应用实现过程分析(掌握)

Mybatis 框架提供一种缓存机制，通过缓存的应用来提高查询性能,但可能会有一定的不一致(脏读)问题。Mybatis中的缓存提供一级缓存和二级缓存的实现,默认都是开启状态（可参考官网）。

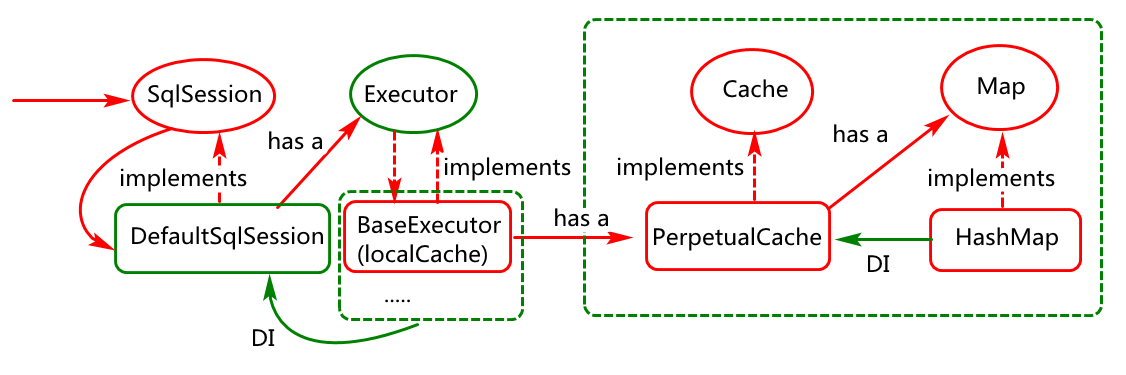
1.Mybatis 一级缓存

Mybatis中的一级缓存有时又称为SqlSession级缓存，SqlSession关闭时一级缓存失效。在同一个SqlSession内部多次执行同一个查询，后续的查询会从此缓存取数据.一级缓存结构分析如下:

一级缓存基本架构应用分析:



一级缓存代码架构分析:

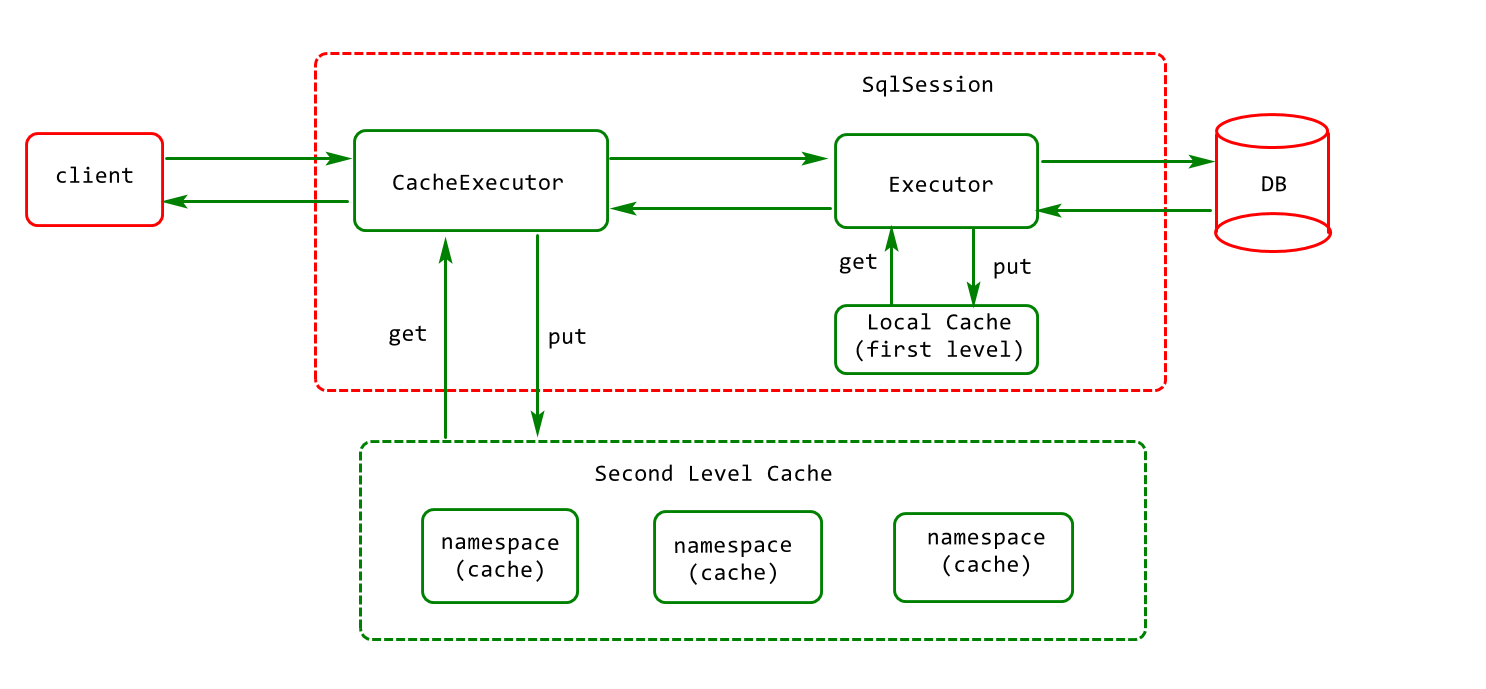


说明：一级缓存的实现可查看BaseExecutor类中的localCache属性。

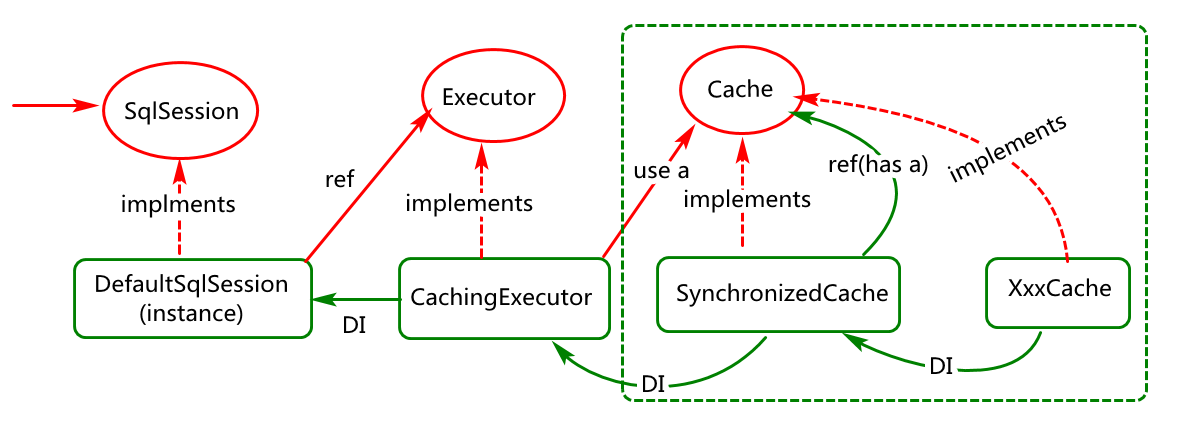
2.MyBatis二级缓存

Mybatis中的二级缓存有时又称跨session缓存，可在多个SqlSession间共享数据,假如要使用二级缓存，可在对应的mapper文件中借助cache元素进行配置(可参考官方映射文件配置),二级缓存架构分析如下:

二级缓存基本应用架构:



二级缓存代码结构分析:



说明：

1)二级缓存的实现可查看CachingExecutor类中的query方法.

2)二级缓存底层采用装饰模式进行设计,例如:

new SynchronizedCache (new LoggingCache(…));

1. 二级缓存中可以存储序列化对象(readWrite属性值为false时)
2. 使用二级缓存可在映射文件中借助<Cache/>标签进行声明,也可在接口中借助注解@CacheNamespace对接口进行描述.
3. 当映射配置有注解方式,也有xml方式时可通过缓存引用进行配置,例如.

@CacheNamespaceRef(name="com.cy.pj.googs.dao.GoodsDao ")

# SPRING 框架核心进阶

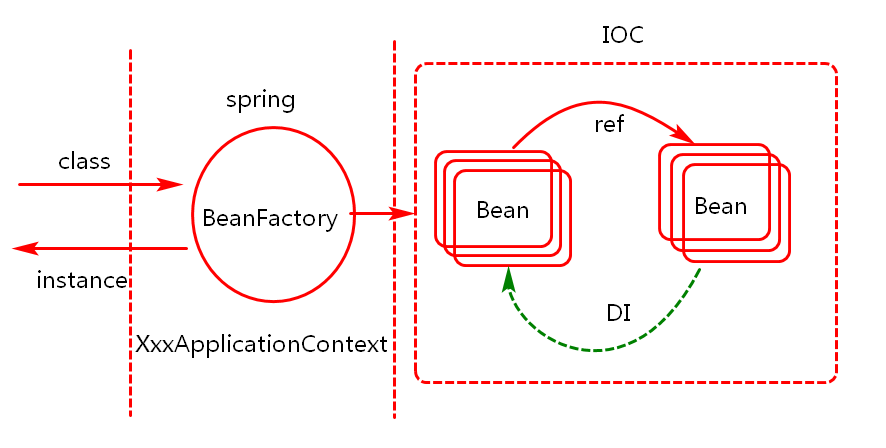
## SPRING 框架架构分析

### Spring 框架应用架构

Spring 官网资源:spring.io/projects

Spring 是一个”资源整合”框架,通过spring可将很多资源(例如连接池,

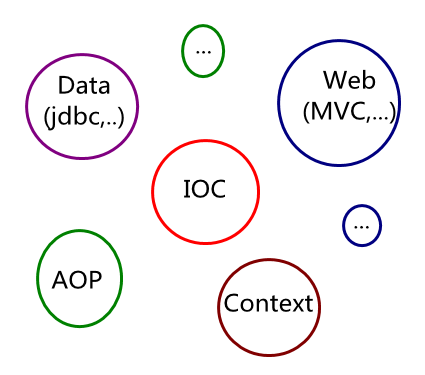
mybatis,...)等整合在一起,对外提供相关服务(例如,秒杀服务,支付服务,...)。



说明:spring 框架中一切资源的整合都源于IOC模块，IOC要实现对象生命周期的管理，对象依赖关系的管理。

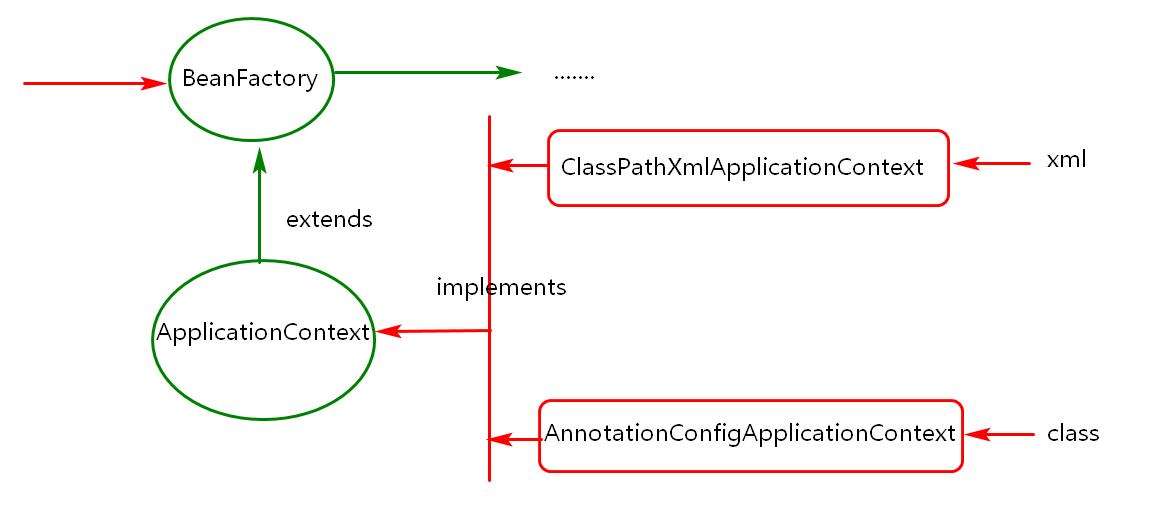
### Spring 框架产品架构

产品架构主要从这个框架对外提供的服务（功能）进行理解.

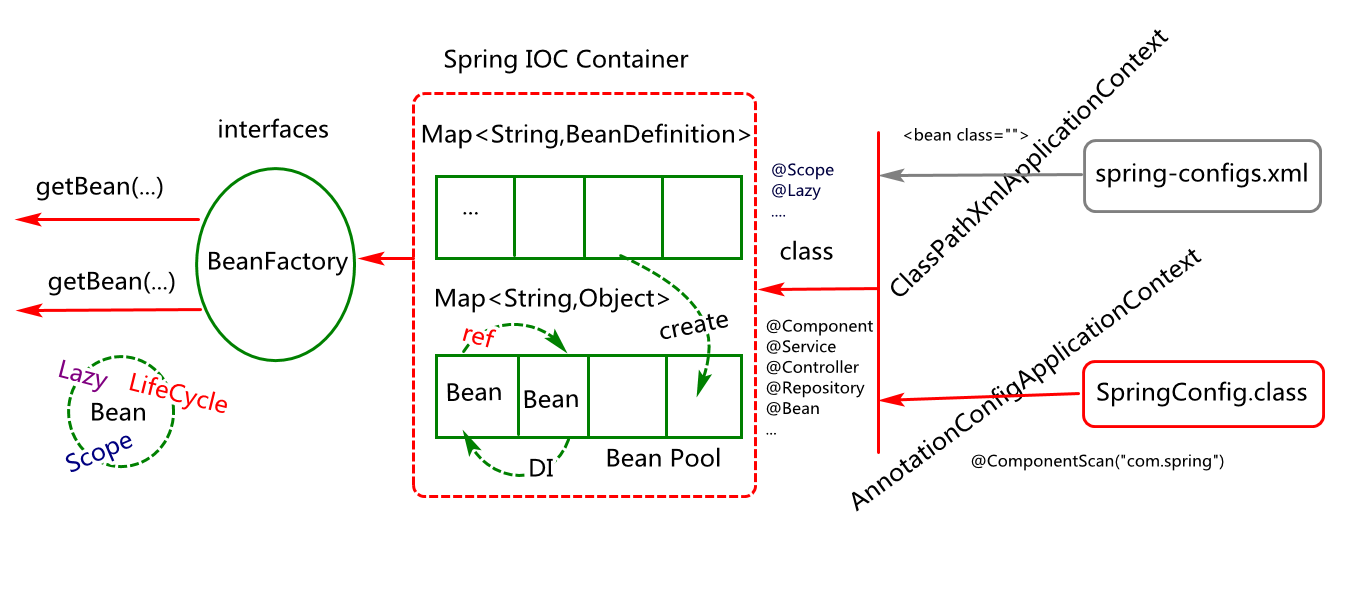


### Spring 框架技术架构

IOC API基础架构(Spring 工厂对象分析)



IOC(控制反转) 容器初始化过程分析.



## SPRING框架快速实践 (注解方式-脱离文档)

### Spring项目创建及配置

1. 创建maven项目(jar包项目)
2. 项目名称 CGB-SPRING-01
3. 组id:com.cy
4. 打包方式:jar
5. 添加项目依赖(spring-context)

添加spring依赖

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.9.RELEASE</version>

</dependency>

添加junit依赖

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

### Spring基本测试环境创建

添加spring配置类(SpringConfig)

**package** com.cy.spring.config;

**import** org.springframework.context.annotation.ComponentScan;

/\*\*

\* **@ComponentScan** 用于告诉spring容器从

\* 从指定包进行bean的扫描

\*/

@ComponentScan("com.cy.spring.beans")

**public** **class** SpringConfig {//spring-configs.xml

}

定义测试基类

**package** com.spring;

**import** org.junit.After;

**import** org.junit.Before;

**import** org.junit.Test;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**public** **class** TestBase {

**protected** AnnotationConfigApplicationContext ctx;

@Before

**public** **void** init() {

ctx=**new** AnnotationConfigApplicationContext(

SpringConfig.**class**);

}

@After

**public** **void** close() {

ctx.close();

}

@Test

**public** **void** testCtx() {

System.***out***.println(ctx);

}

}

### Spring项目基本业务实现

业务描述,创建一个DefaultCache对象然后将此对象交给Spring容器管理.

创建DefaultCache类,并明确此类交给spring管理.

**package** com.cy.spring.beans;

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

**import** org.springframework.context.annotation.Lazy;

**import** org.springframework.context.annotation.Scope;

**import** org.springframework.stereotype.Component;

/\*\*

**@Component** 注解用于告诉spring容器

请将这个类交给spring管理.

**@Lazy** 用于告诉spring容器此对象要延迟加载

**@Scope** 用于告诉spring容器此bean的作用域

1)singleton (单例作用域-默认,会存储到池中)

2)prototype (多例作用域,每次获取都创建新对象)

\*/

@Lazy

@Scope("singleton")

@Component //@Controller,@Service,...

**public** **class** DefaultCache {

**public** DefaultCache() {

System.***out***.println("DefaultCache()");

}

@PostConstruct //告诉spring 此对象初始化时执行init方法

**public** **void** init() {

System.***out***.println("init()");

}

@PreDestroy//告诉spring 此对象销毁时执行close方法

**public** **void** close() {

System.***out***.println("close()");

}

}

定义测试类,从spring容器中获取bean对象

**package** com.cy.test;

**import** org.junit.Assert;

**import** org.junit.Test;

**import** com.cy.spring.beans.DefaultCache;

**public** **class** TestCache **extends** TestBase {

@Test

**public** **void** testDefaultCahce() {

DefaultCache cache01=

ctx.getBean("defaultCache",DefaultCache.**class**);

Assert.*assertNotEquals*(**null**, cache01);

DefaultCache cache02=

ctx.getBean("defaultCache",DefaultCache.**class**);

Assert.*assertNotEquals*(**null**, cache02);

System.***out***.println(cache01==cache02);

}

}

### Spring项目课堂练习分析及实现

1.整合第三方连接池DRUID.

1. 步骤分析:

step01:添加依赖(数据库驱动依赖,连接池依赖)

step02:对druid连接池进行配置

step03:对连接池进行单元测试.

1. 关键代码实现

添加mysql依赖

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.17</version>

</dependency>

添加Druid依赖

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.19</version>

</dependency>

定义数据源配置类

**package** com.cy.spring.beans;

**import** javax.sql.DataSource;

@Configuration

**public** **class** DataSourceConfig {

@Bean(value="druid",initMethod="init",destroyMethod="close")

**public** DataSource newDruid() {

// System.out.println("newDruid()");

DruidDataSource ds=**new** DruidDataSource();

//ds.setDriverClassName("com.mysql.cj.jdbc.Driver");

ds.setUrl("jdbc:mysql:///test?serverTimezone=GMT");

ds.setUsername("root");

ds.setPassword("root");

ds.setMaxWait(10000);

//....

**return** ds;

}

}

创建单元测试类

**package** com.cy.test;

**import** javax.sql.DataSource;

**import** org.junit.Assert;

**import** org.junit.Test;

**import** com.zaxxer.hikari.HikariDataSource;

**public** **class** TestDataSource **extends** TestBase{

@Test

**public** **void** testDruidDataSource()**throws** Exception {

DataSource ds=

ctx.getBean("druid",DataSource.**class**);

Assert.*assertNotEquals*(**null**, ds);

System.***out***.println(ds.getConnection());

}

}

2.整合第三方连接池HiKariCP连接池(扩展).

## SPRING IOC 应用原理进阶分析

### Spring IOC设计思想分析

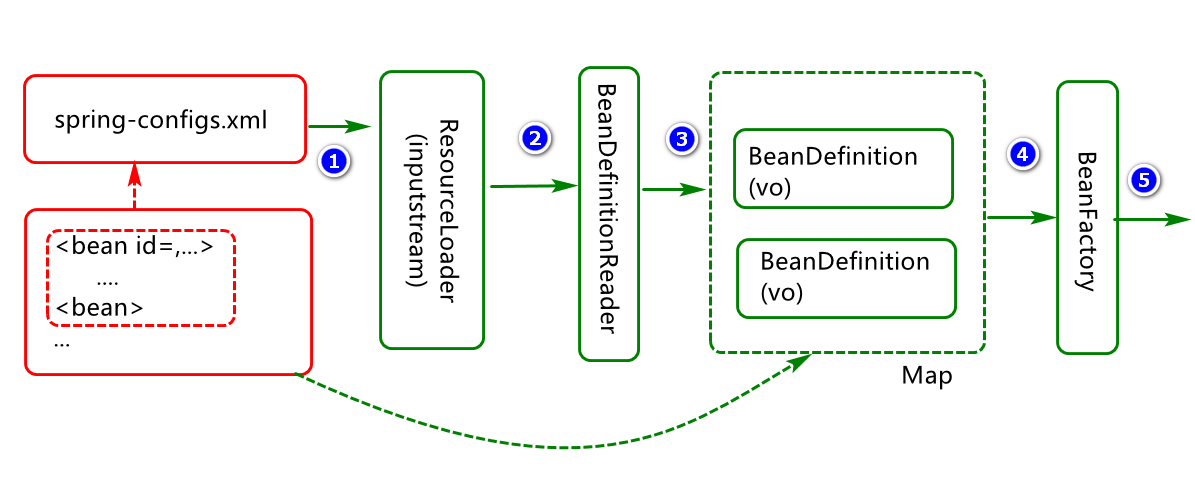
IOC 是一种设计思想，称之为控制反转。基于这种思想实现对象创建，对象的科学管理以及应用时的解耦(借助DI机制实现)。Spring框架核心就是基于这种机制进行了完美实现。

说明：

1. 控制反转探讨的是什么？谁控制谁的问题（spring控制对象的创建管理）
2. 生活中的IOC的实现？(例如股票操盘手,父母包办婚姻）

### Spring Bean 工厂的初始化

基于xml方式实现Spring中Bean工厂的初始化.



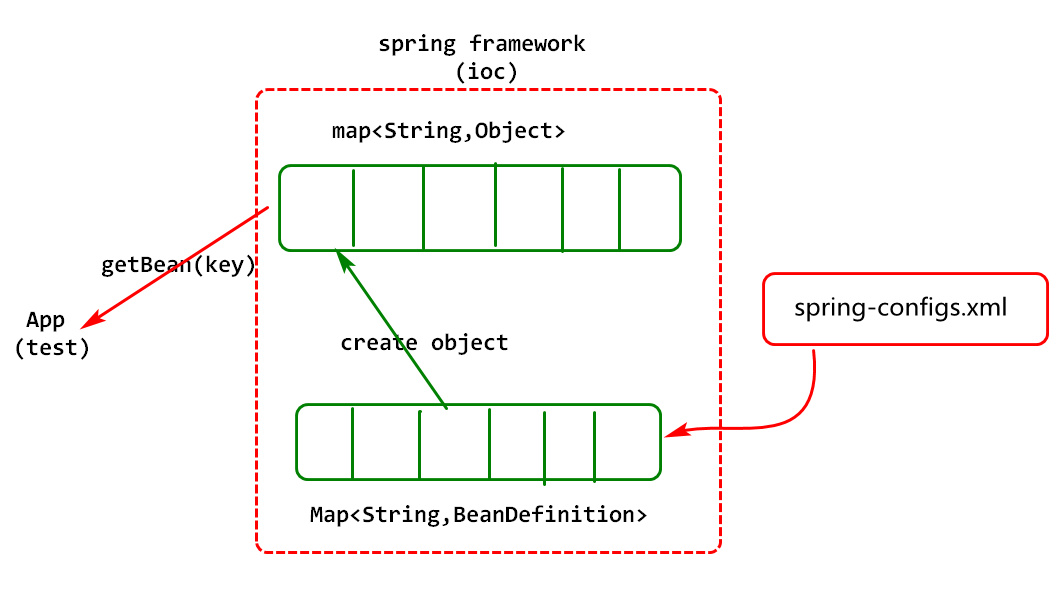
说明:Spring 中的Bean工厂会基于Bean对象描述,创建bean的实例,并可以有选择性的对实例对象进行管理(例如单例作用域的对象).

### Spring 中的两大map对象分析

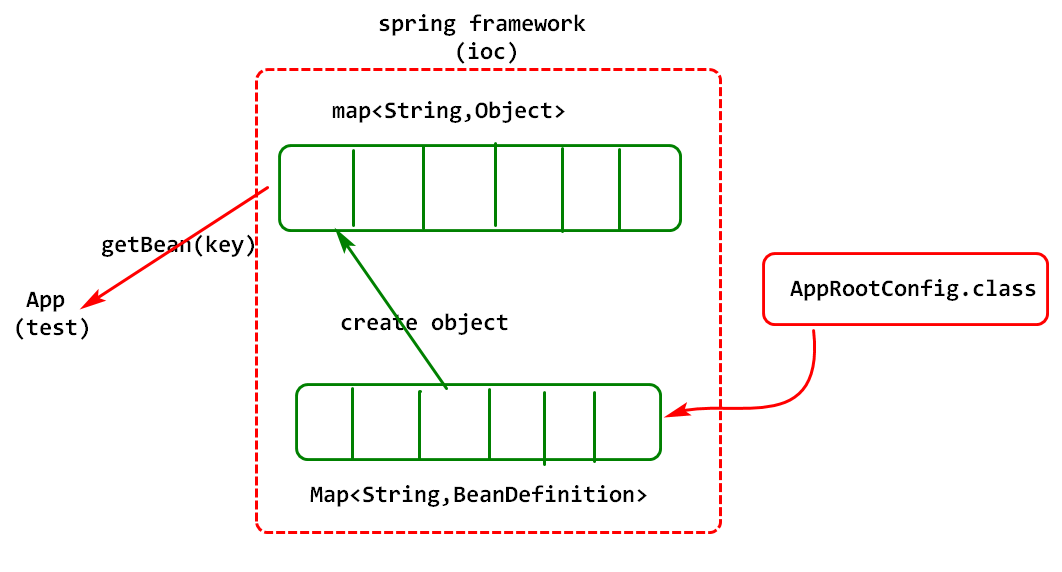
1.如何理解Spring中的两大map对象(存储对象的两个容器)？

1. 一个map用于存储bean的配置信息(工厂的原材料)
2. 一个map用于存储bean的实例信息(工厂中的成品对象)

基于xml配置文件实现:



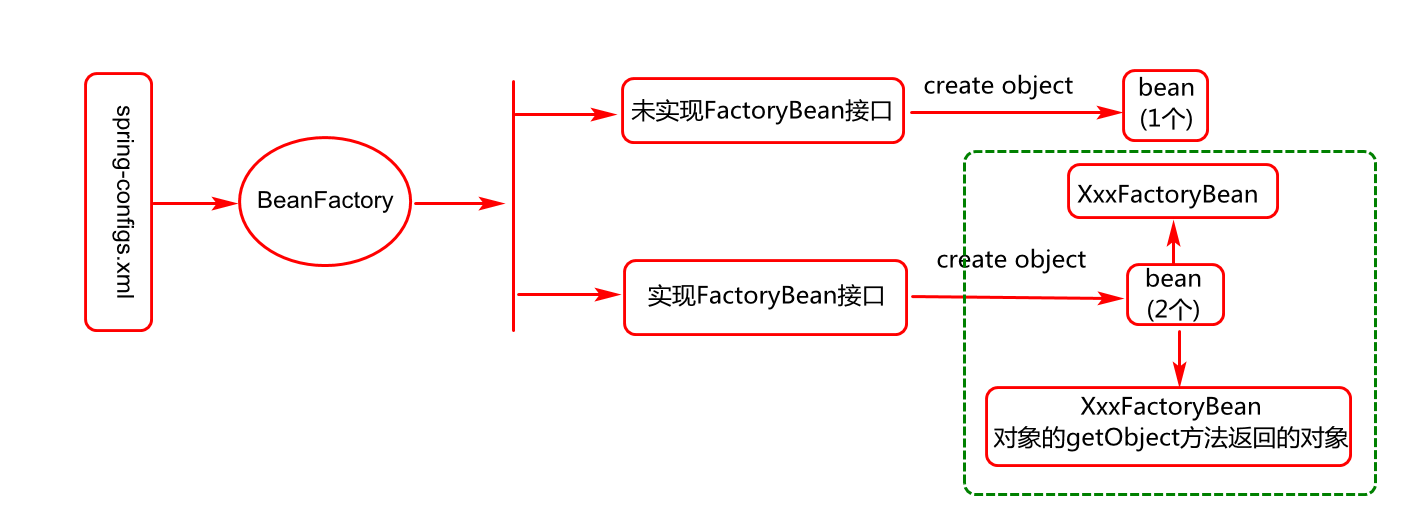
基于注解配置实现:



### Spring 中两大bean对象分析(了解)

Bean对象创建

1. 未实现FactoryBean接口(直接构造方法);
2. 实现FactoryBean接口（调用FactoryBean对象的getObject方法）



说明：一般在创建一些相对复杂的工厂对象时，通常会写一个工厂bean对象，

然后基于工厂bean对象创建具体的工厂对象，例如SqlSessionFactoryBean,

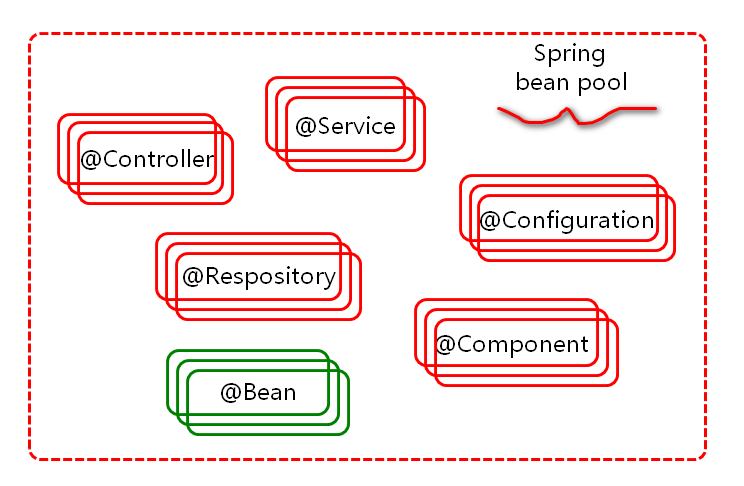
ShiroFilterFactoryBean，ProxyFactoryBean等。

### Spring 中两大bean对象描述方式

Bean 对象的描述

1. xml方式 (例如<bean id=”factory” class=”com.beans.Factory”>)
2. annotation方式（@Service,@Controller，@Configuration，@Bean，..）

Spring 中用于描述这是一个Bean对象的相关注解如下:



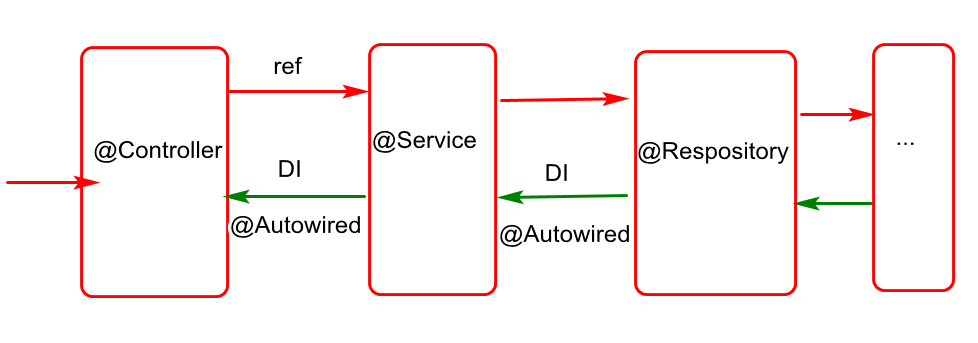
说明:无论使用如上图中的哪个注解对Bean进行描述,对Spring而言都认为是一样的Bean.

## Spring 依赖注入实践增强

### Spring 中Bean对象依赖注入分析(重点)

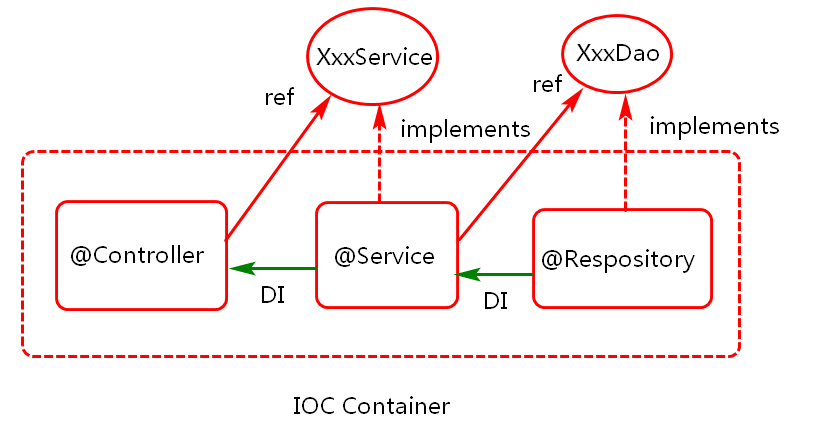
依赖注入(DI)是一种设计思想,简单点就是对象通过set或构造方法直接为对象属性赋值,在Spring框架中为这种注入基于反射技术提供一种自动实现方式.例如:

IOC 依赖注入在项目中的应用实现：



实际项目中为了解耦和,对象之间通常会通过接口进行通讯,也就是

说对象要耦合与接口,例如2

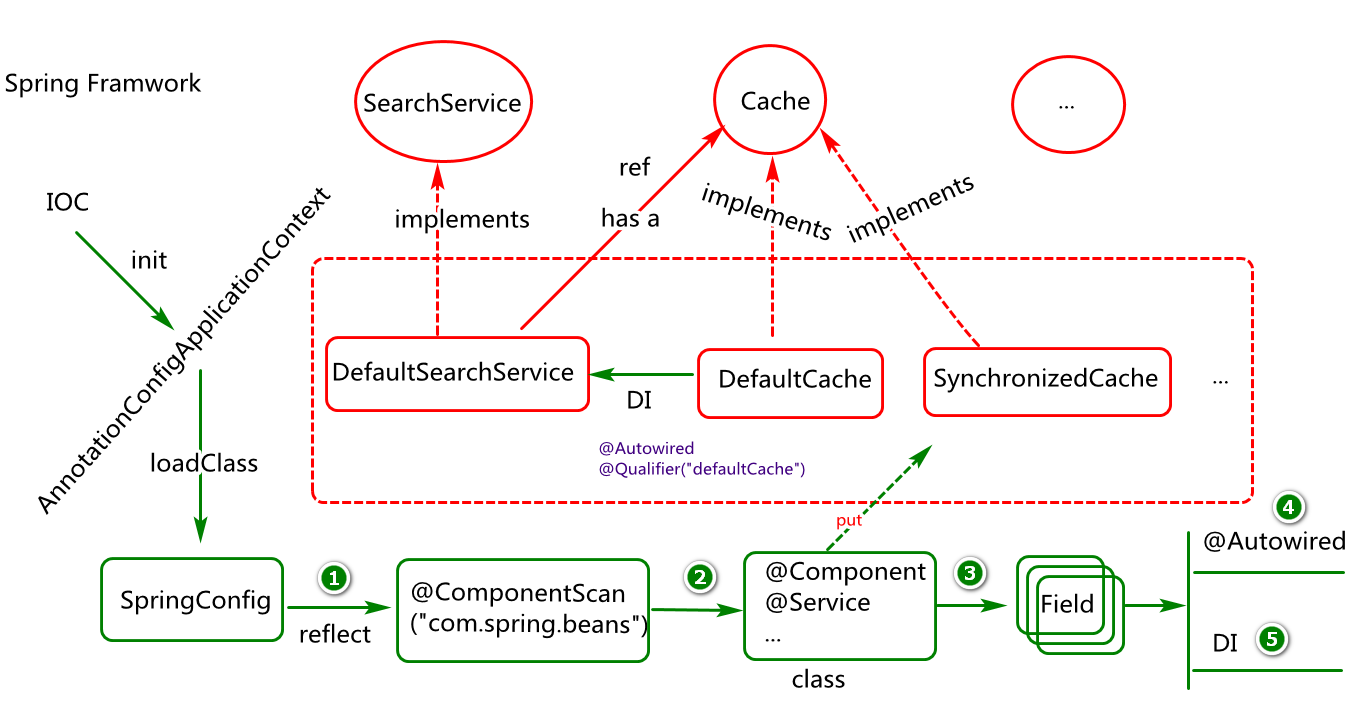


个人认为：IOC的核心是对象生命周期管理(资源管理)以及

依赖注入（资源协同）；

### Spring 中Bean对象依赖注入实践

在1.2小节基础上,基于如下设计,进行代码实践分析及实现。



核心步骤分析：

Step01:创建Cache接口以及对应的实现DefaultCache

Step02:创建SearchService接口以及对应的实现类。

Step03:将DefaultCache对象注入给DefaultSearchService。

Step03:创建SynchronizedCache测试@Autowired注解。

关键代码分析及实现：

Cache接口定义

**public** **interface** Cache {

}

DefaultCache类定义

@Lazy

@Component

**public** **class** DefaultCache **implements** Cache{

**public** DefaultCache() {

System.***out***.println("DefaultCache()");

}

}

SynchronizedCache定义

@Component

**public** **class** SynchronizedCache **implements** Cache {

}

SearchService接口定义

**public** **interface** SearchService {

}

SearchService实现类定义

@Service

**public** **class** DefaultSearchService **implements** SearchService {

/\*\***@Autowired** 可以修饰属性，set方法，构造方法等

\* 默认按照属性类型，方法参数类型为对象属性注入值,

\* 假如相同类型的对象有多个，还会按属性名或方法参

\* 数名等进行查找。

\* **@Qualifier** 配合@Autowired，用于指定要注入的对

\* 象的名字。

\*/

@Autowired

@Qualifier("defaultCache")

**private** Cache cache;

@Override

**public** String toString() {

**return** "DefaultSearchService [cache=" + cache + "]";

}

}

测试实现

**public** **class** TestSearchService **extends** TestBase {

@Test

**public** **void** testSearchService() {

DefaultSearchService ds=

ctx.getBean("defaultSearchService",

DefaultSearchService.**class**);

System.***out***.println(ds);

}

}

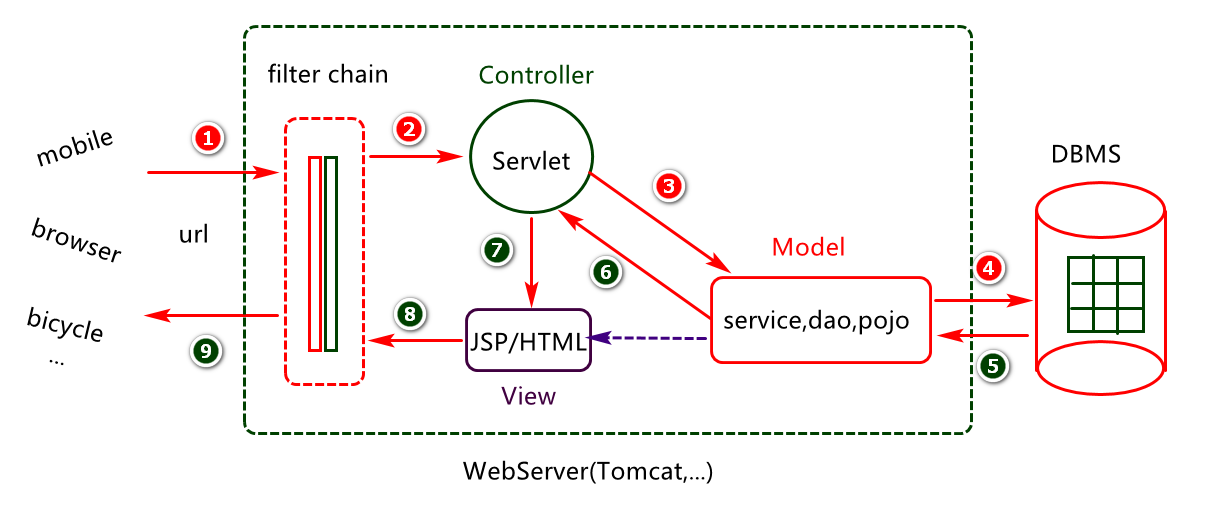
# SPRING 框架MVC模块进阶

## Spring MVC 设计思想

### Spring MVC 设计思想分析

MVC是一种分层架构设计思想，目的是基于对象职责上的不同，进行分层设计，实现各司其职，各尽所能，以提高代码的可维护性，可扩展性。

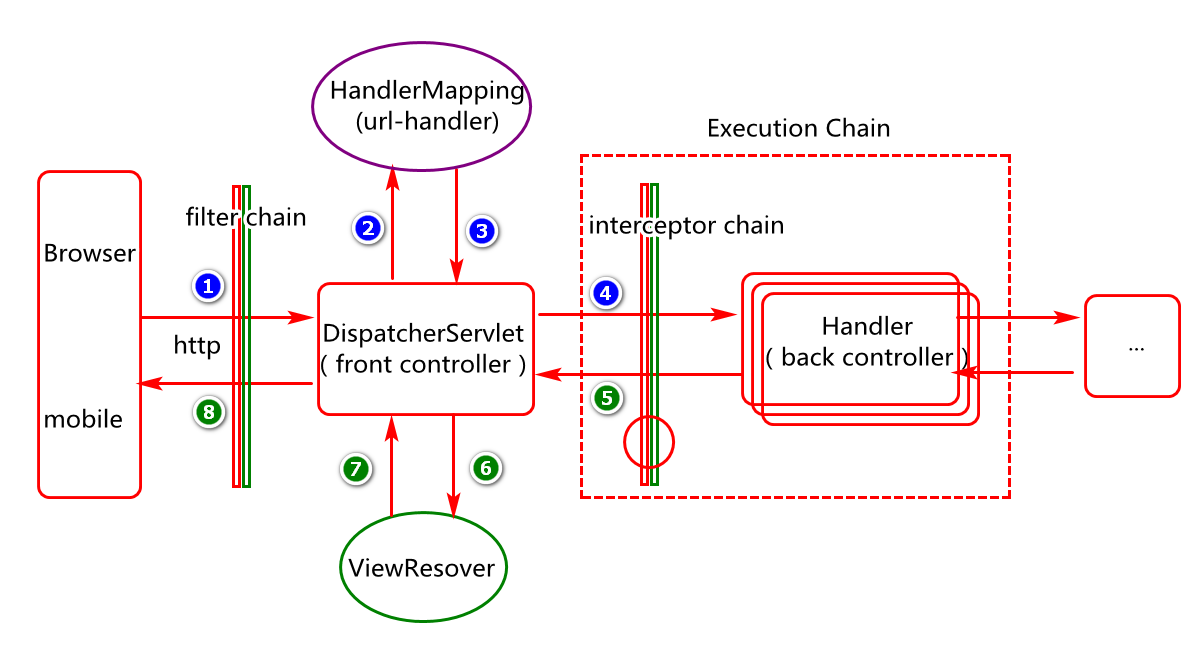
1. 生活中的MVC:正规饭店(菜单,服务员,厨师)
2. 程序中的MVC:(html,jsp)/servlet/(service,dao))



SPRING MVC是Spring 框架中的一个WEB模块, 是基于MVC设思想的一种完美实现。

### Spring MVC 核心对象及流程分析

Spring MVC 处理流程及其核心组件对象构成，如下图所示：



FAQ：说说spring MVC中的核心组件？

1. DipatcherServlet:前端控制器(web服务器启动时加载)
2. HandlerMapping:注册中心(负责存储url到后端控制器的映射)
3. HandlerInterceptor:拦截器(请求到handler之间的拦截)
4. Handler:后端处理器(又称之为Controller)
5. ViewResolver:视图解析器(负责是视图页面进行解析)

## Spring MVC 快速实践

### xml方式配置实现(脱离文档)

1. 创建maven项目
2. 项目名称: CGB-SPRINGMVC-01
3. 组id: com.cy
4. 打包方式: war包方式
5. 配置并初始化项目环境

1)生成web.xml(配置spring mvc 前端控制器)

2)设置项目的运行时环境(选择tomcat,提供servlet支持)

3)设置项目编码方式 utf-8

4)设置统一编译环境 JDK8

5)添加项目依赖:spring-webmvc

6)添加spring mvc配置文件并进行配置:spring-configs.xml

7)web.xml中配置spring mvc前端控制器(DispatcherServlet)

8)部署项目,启动tomcat测试 (假如tomcat正常启动,则没问题)

说明:如上配置可参考后面内容中的关键代码分享.

1. Spring MVC基础业务实现
2. 定义Controller类
3. 包名:com.cy.pj.search.controller
4. 类名:SearchController

代码如下:

@Controller

@RequestMapping("/search/")

**public** **class** SearchController {

}

其中:@RequestMapping修饰类时用于定义请求路径

1. 添加Controller方法

方法1:此方法用于向客户端返回一个页面

@RequestMapping("doSearchUI")

**public** String doSearchUI() {

**return** "search";

}

其中:对于返回值search需要在/WEB-INF/pages/目录下有对应的

search.html页面.

方法2:此方法用于向客户端返回一个json格式的字符串

@RequestMapping("doSearch")

@ResponseBody

**public** Object doSearch(String key) {

Map<String,Object> map=**new** HashMap<String,Object>();

map.put("state", 1);

map.put("message","hello everyone");

**return** map;//{"state":1,"message":"hello everyone"}

}

其中:

a)@ResponseBody修饰方法时,Spring MVC可以将方法的返回值以指定格式进行进行输出,例如JSON格式字符串.

b)doSearch方法进行测试时,需要在项目中添加如下依赖:

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.9.3</version>

</dependency>

1. 部署并启动服务测试.
2. 关键代码分享:

spring-configs.xml

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

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

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/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>

<context:component-scan base-package=*"com.cy.controller"*/>

<!-- Enable MVC Configuration:启用默认bean对象(注解驱动) -->

<mvc:annotation-driven/>

<!-- 启用 DefaultServletHttpRequestHandler对静态进行处理 -->

<mvc:default-servlet-handler/>

<!-- 配置视图解析器(ViewResolver)对象 -->

<bean id=*"viewResolver"* class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<!-- Set DI (默认找对象中的set方法)-->

<property name=*"Prefix"* value=*"/WEB-INF/pages/"*/>

<property name=*"Suffix"* value=*".html"*/>

</bean>

</beans>

Web.xml

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* version=*"2.5"*>

<display-name>CGB-SPRINGMVC-01</display-name>

<!-- 配置前端控制器 -->

<servlet>

<servlet-name>frontController</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-configs.xml</param-value>

</init-param>

<!-- tomcat 启动时则初始化servlet,数字越小优先级越高 -->

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>frontController</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

SearchController

**package** com.cy.pj.search.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.ResponseBody;

@Controller

@RequestMapping("/search/")

**public** **class** SearchController {

@RequestMapping("doSearchUI")

**public** String doSearchUI() {

**return** "search";

}///WEB-INF/pages/search.html

@RequestMapping("doSearch")

@ResponseBody

**public** Object doSearch(String key) {

Map<String,Object> map=**new** HashMap<String,Object>();

map.put("state", 1);

map.put("message","hello everyone");

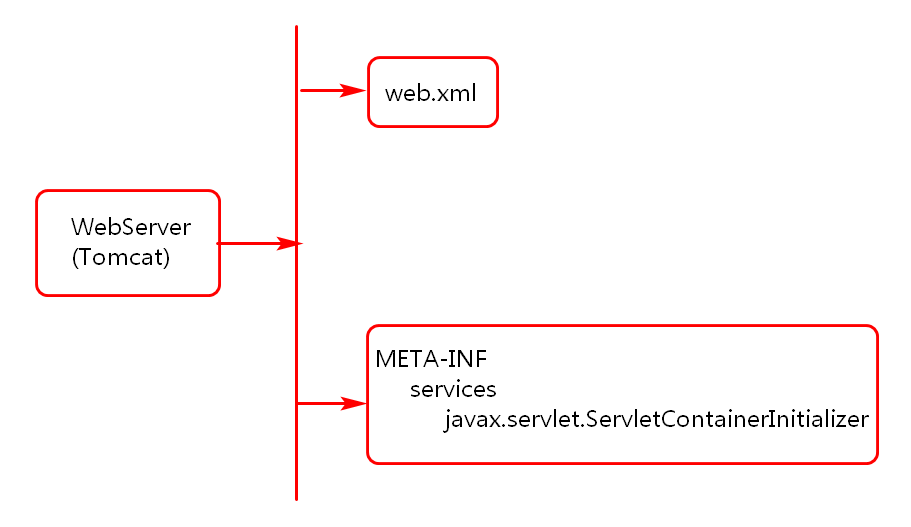
**return** map;//{"state":1,"message":"hello everyone"}

}

}

### 注解方式配置实现（脱离文档）

Tomcat 启动加载方式:



1. 创建maven项目
2. 项目名称: CGB-SPRINGMVC-02
3. 组id: com.cy
4. 打包方式: war包方式
5. 配置并初始化项目环境(重点对xml方式进行重构)

1)配置maven war包插件(忽略web.xml)

2)设置项目的运行时环境(选择tomcat)

3)设置项目编码方式 utf-8

4)设置统一编译环境 JDK8

5)添加项目依赖:spring-webmvc

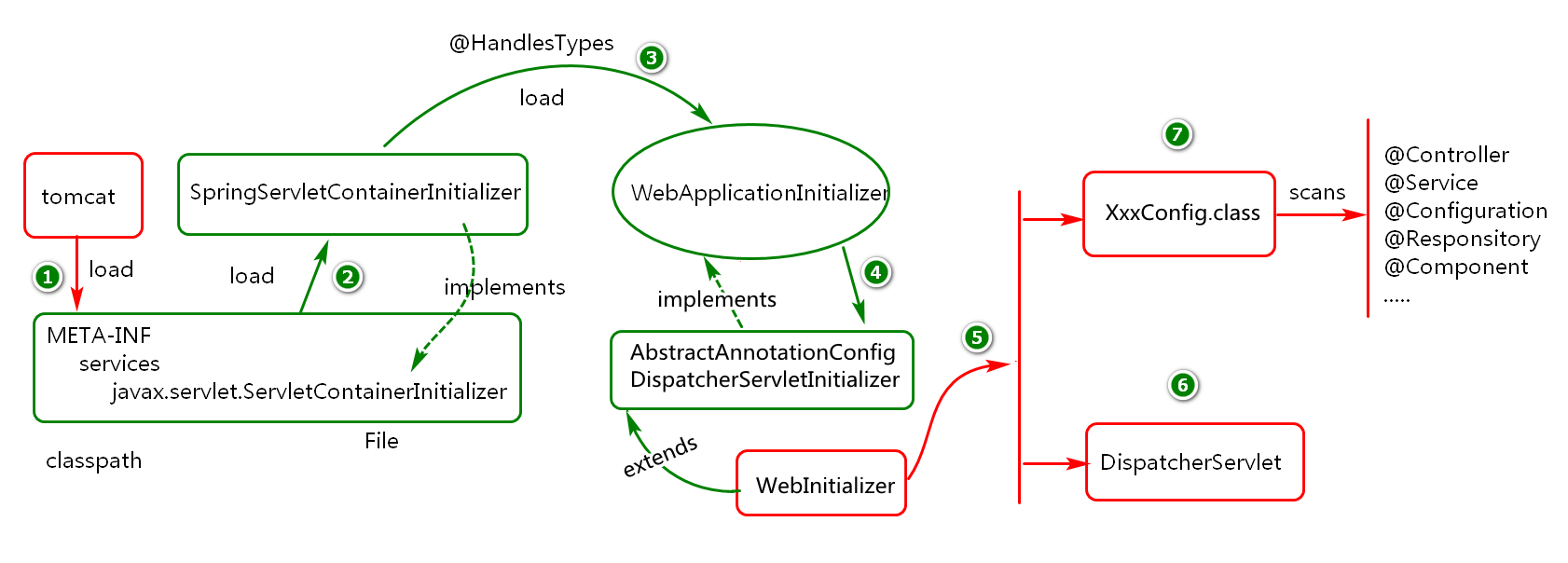
6)添加spring mvc配置类:SpringWebConfig类

7)创建WebInitializer类配置spring mvc(例如前端控制器)。

8)部署项目,启动tomcat测试 (假如tomcat正常启动,则没问题)

1. Spring MVC基础业务实现 (参考xml方式业务代码实现)

4.Spring mvc 注解方式应用分析:



其中META-INF目录可在sping-web.jar中进行查看.

关键代码分析:

SpringWebConfig类

@Configuration

@ComponentScan("com.cy.pj.search.controller")

@EnableWebMvc //<mvc:annotation-driven/>

**public** **class** SpringWebConfig **implements** WebMvcConfigurer{

//<mvc:default-servlet-handler/>

@Override

**public** **void** configureDefaultServletHandling(

DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

@Override

**public** **void** configureViewResolvers(

ViewResolverRegistry registry) {

registry.jsp("/WEB-INF/pages/",".html");

}

}

WebInitializer类

**public** **class** WebInitializer **extends**

AbstractAnnotationConfigDispatcherServletInitializer {

//Service,Repository

@Override

**protected** Class<?>[] getRootConfigClasses() {

System.***out***.println("getRootConfigClasses()");

**return** **null**;

}

//View,Controller

@Override

**protected** Class<?>[] getServletConfigClasses() {

System.***out***.println("getServletConfigClasses()");

**return** **new** Class[] {SpringWebConfig.**class**};

}

@Override

**protected** String[] getServletMappings() {

System.***out***.println("getServletMappings()");

**return** **new** String[] {"/"};

}

}

## Spring MVC 请求响应处理增强分析

所有MVC框架的重点都在请求和响应数据的处理上。

### 请求处理增强分析及实现(重点)

Spring MVC请求处理主要从如下几个方面进行考虑：

1. 请求路径(普通方式，rest方式) ：404
2. 请求方式(Get请求，Post请求,…):405
3. 请求参数(直接量，POJO对象，MAP对象):400

代码分析如下：

@Controller

@RequestMapping("/request/")

**public** **class** RequestHandleController {

//===请求url定义

//普通URL定义实现(多个url可以对应同一个资源)

@RequestMapping(value={"doHandleUrl","doWelcomeUI"})

**public** String doHandleUrl() {

**return** "welcome";

}

//REST风格的URL，其url格式为{a}/{b}/{c}

//假如希望方法参数获取url中{}表达式内部的值

//可以使用@PathVariable对参数进行修饰

@RequestMapping("{module}/{page}")

**public** String doMoudleUrl(

@PathVariable String module,

@PathVariable String page) {

**return** module+"/"+page;

}

//===请求方式定义

//@RequestMapping(value="type",method=RequestMethod.GET)

@GetMapping("type")

//@PostMapping("type")

@ResponseBody

**public** String doRequestType() {

**return** "request type";

}

//=====请求参数处理======

@GetMapping("param")

@ResponseBody

**public** String doRequestParam(

RequestWrapper rw,//pojo

@RequestParam(required=**false**) String msg,

@DateTimeFormat(pattern="yyyy/MM/dd")Date begin) {

**return** "request parameter handle msg="+msg+",begin="+begin+",rw="+rw.toString();

}

}

### 响应处理增强分析实现（重点）

Spring MVC响应处理主要从如下几个方面进行考虑：

1. 响应方式(转发forward，重定向redirect)
2. 响应数据封装(ModelAndView,Model,Map,Pojo)
3. 响应数据转换（将对象序列化为JSON格式字符串）

代码分析如下：

@Controller

@RequestMapping("/resp/")

**public** **class** ResponseHandleController {

@RequestMapping("doResponseUI")

**public** String doResponseUI() {

**return** "response";

}

@RequestMapping("type")

**public** String doResponseType() {

//转发(客户端请求一次)

**return** "forward:doResponseUI";//address

//重定向(客户端两次请求)

//return "redirect:doResponseUI";//address

}

@RequestMapping("doDataConvert")

@ResponseBody

**public** String doDataConvert()**throws** Exception {

Map<String,Object> map=**new** HashMap<String,Object>();

map.put("id", 100);

map.put("msg", "hello jackson");

//借助jackson中的API将对象转换json格式的字符串

ObjectMapper om=**new** ObjectMapper();

**return** om.writeValueAsString(map);

}

}

# SSM框架综合案例实现(注解方式)

## 业务描述

通过spring整合mybatis完成商品数据库中数据的查询,并在客户端以json格式对数据进行输出?

思路:

1. 创建maven web项目(打包方式war):CGB-SSM-01
2. 配置项目?(运行时环境,编码,编译版本)
3. 添加依赖?(mysql,druid,mybatis,junit,spring-webmvc,..)
4. 整合资源?(连接池,mybatis,spring mvc..)

## 初始化化数据环境

数据库脚本分析:

**drop** database if **exists** dbgoods;

**create** database dbgoods **default** **character** **set** utf8;

use dbgoods;

**create** **table** tb\_goods(

id bigint auto\_increment,

name **varchar**(200) **not** **null**,

remark text,

createdTime datetime,

**primary** **key** (id)

)engine=InnoDB;

**insert** **into** tb\_goods **values** (**null**,'A','A...',now());

**insert** **into** tb\_goods **values** (**null**,'B','B...',now());

**insert** **into** tb\_goods **values** (**null**,'C','C...',now());

说明:可以将如上脚本写到db-goods.sql文件中,然后去执行完成完成初始化.

## 项目创建及配置

1. 创建maven web项目:CGB-SSM-01
2. 在pom.xml添加war包插件,修改其配置(这样项目可以不写web.xml)

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.6</version>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

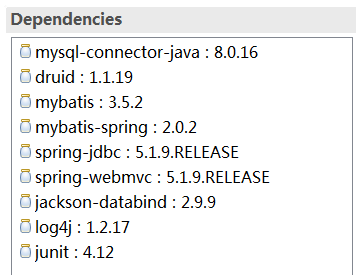
</configuration>

</plugin>

</plugins>

</build>

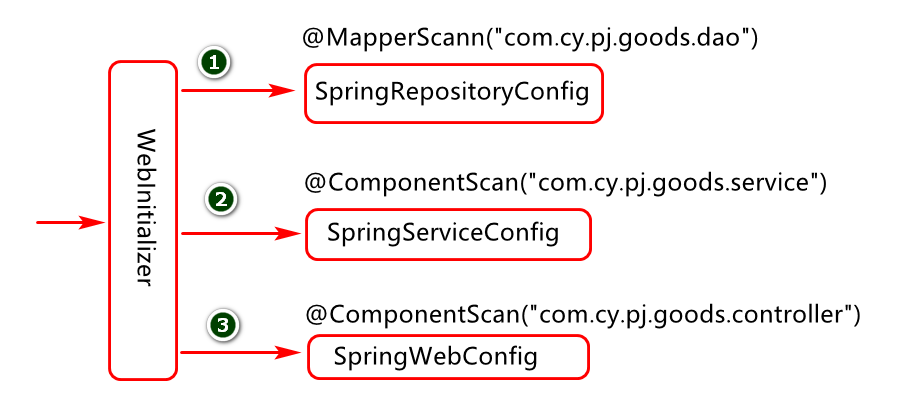
1. 配置项目(设置运行时环境-tomcat,编译版本JDK1.8)
2. 添加项目依赖(参考如下图中的依赖)



## 项目资源整合实现

### 配置架构分析及实现

配置架构分析:



其中:

1. SpringRepositoryConfig 负责数据层配置
2. SpringServiceConfig负责业务层配置
3. SpringWebConfig负责请求处理层配置(Spring MVC中的V,C)
4. WebInitializer 负责启动初始化.

基于配置架构,创建配置类:

数据层配置类

**package** com.cy.pj.common.config;

@Configuration

@MapperScan("com.cy.pj.goods.dao")//扫描dao

**public** **class** SpringRepositoryConfig {

}

业务层配置类:

**package** com.cy.pj.common.config;

@Configuration

@ComponentScan("com.cy.pj.goods.service")

**public** **class** SpringServiceConfig {

//.....

}

控制层配置类

**package** com.cy.pj.common.config;

@ComponentScan("com.cy.pj.goods.controller")

@EnableWebMvc

@Configuration

**public** **class** SpringWebConfig **implements** WebMvcConfigurer{

}

WebInitializer 启动类配置:

**package** com.cy.pj.common.config;

//-->web.xml

**public** **class** WebInitializer **extends**

AbstractAnnotationConfigDispatcherServletInitializer {

//Service,Repository

@Override

**protected** Class<?>[] getRootConfigClasses() {

System.***out***.println("getRootConfigClasses()");

**return** **new** Class[] {SpringRepositoryConfig.**class**,

SpringServiceConfig.**class**};

}

//View,Controller

@Override

**protected** Class<?>[] getServletConfigClasses() {

System.***out***.println("getServletConfigClasses()");

**return** **new** Class[] {SpringWebConfig.**class**};

}

@Override

**protected** String[] getServletMappings() {

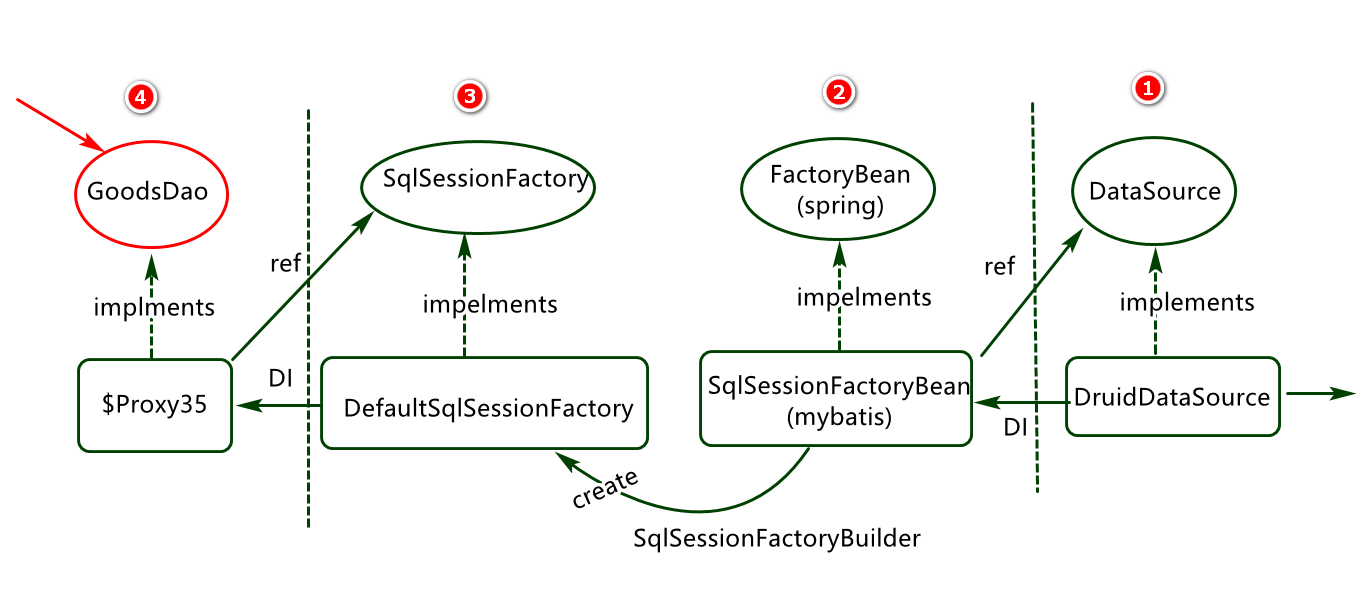
System.***out***.println("getServletMappings()");

**return** **new** String[] {"/"};

}

}

整合过程分析:(顺序从右到左)



### 整合连接池对象

在SpringRepositoryConfig 类中添加dataSource()方法,在方法中创建

DruidDataSource对象.并将此对象交给spring管理.

@Bean(value="druid",initMethod="init",destroyMethod="close")

**public** DruidDataSource dataSource() {

DruidDataSource ds=**new** DruidDataSource();

ds.setUrl("jdbc:mysql:///dbgoods?serverTimezone=GMT%2B8");

ds.setUsername("root");

ds.setPassword("root");

**return** ds;

}

说明:@Bean注解要配合@Configuration注解使用,@Configuration用于描述类.

### 整合mybatis框架

在SpringRepositoryConfig类中添加创建SqlSessionFactory对象创建

的方法:

@Bean("sqlSessionFactory")

**public** SqlSessionFactory newSqlSessionFactory(

DataSource dataSource)

**throws** Exception {

//构建SqlSessionFactoryBean对象

SqlSessionFactoryBean factoryBean =

**new** SqlSessionFactoryBean();

factoryBean.setDataSource(dataSource);

//调用FactoryBean的getObject方法创建SqlSessionFactory

//底层会使用SqlSessionFactoryBuilder创建

**return** factoryBean.getObject();

}

参考:www.mybatis.org/spring

### 整合Spring MVC 模块

在Spring MVC配置类中添加视图解析配置,默认servlet处理等配置.

**package** com.cy.pj.common.config;

@Configuration

@ComponentScan("com.cy.pj.goods.controller")

@EnableWebMvc //<mvc:annotation-driven/>

**public** **class** SpringWebConfig **implements** WebMvcConfigurer{

//<mvc:default-servlet-handler/>

@Override

**public** **void** configureDefaultServletHandling(

DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

@Override

**public** **void** configureViewResolvers(

ViewResolverRegistry registry) {

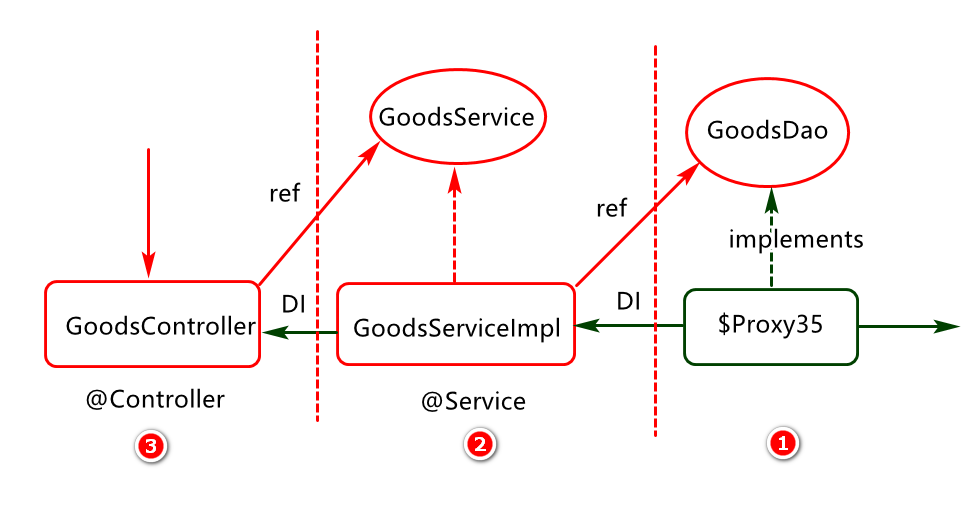
registry.jsp("/WEB-INF/pages/",".html");

}

}

## 业务设计分析及实现

### 业务架构



### POJO实现

定义pojo对象Goods用于封装数据:

**package** com.cy.pj.goods.pojo;

**import** java.io.Serializable;

**import** java.util.Date;

**public** **class** Goods **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 690138036951052829L;

**private** Long id;

**private** String name;

**private** String remark;

**private** Date createdTime;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getRemark() {

**return** remark;

}

**public** **void** setRemark(String remark) {

**this**.remark = remark;

}

**public** Date getCreatedTime() {

**return** createdTime;

}

**public** **void** setCreatedTime(Date createdTime) {

**this**.createdTime = createdTime;

}

@Override

**public** String toString() {

**return** "Goods [id=" + id + ", name=" + name + ", remark=" + remark + ", createdTime=" + createdTime + "]";

}

}

### Dao实现

定义GoodsDao接口,代码如下:

**package** com.cy.pj.goods.dao;

**import** org.apache.ibatis.annotations.Select;

**public** **interface** GoodsDao {

@Select("select \* from tb\_goods")

List<Goods> findGoods();

}

在SpringRepositoryConfig类上添加@MapperScan注解,实现对指定包下Dao接口的扫描:

@Configuration

@MapperScan("com.cy.pj.goods.dao")//扫描dao

**public** **class** SpringRepositoryConfig {

//....

}

编写单元测试类TestGoodsDao,然后getRowCount方法进行测试

**public** **class** TestGoodsDao **extends** TestBase{

@Test

**public** **void** testFindGoods() {

GoodsDao dao=ctx.getBean("goodsDao",GoodsDao.**class**);

List<Goods> list=dao.findGoods();

**for**(Goods g:list) {

System.***out***.println(g);

}

}

}

### Service实现

创建GoodsService接口,实现具体业务:

**package** com.cy.pj.goods.service;

**import** java.util.List;

**import** com.cy.pj.goods.pojo.Goods;

**public** **interface** GoodsService {

List<Goods> findGoods();

}

创建接口实现类:

**package** com.cy.pj.goods.service.impl;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.cy.pj.goods.dao.GoodsDao;

**import** com.cy.pj.goods.pojo.Goods;

**import** com.cy.pj.goods.service.GoodsService;

@Service

**public** **class** GoodsServiceImpl **implements** GoodsService {

@Autowired

**private** GoodsDao goodsDao;

@Override

**public** List<Goods> findGoods() {

//...

List<Goods> list=goodsDao.findGoods();

//...

**return** list;

}

}

编写Service配置类(需要在容器启动时加载),对service对象进行扫描

**package** com.cy.pj.common.config;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.cy.pj.goods.service")

**public** **class** SpringServiceConfig {

//.....

}

编写测试类执行单元测试:

**public** **class** TestGoodsService **extends** TestBase{

@Test

**public** **void** testFindGoods() {

GoodsService gs=

ctx.getBean("goodsServiceImpl", GoodsService.**class**);

List<Goods> list=gs.findGoods();

**for**(Goods g:list) {

System.***out***.println(g);

}

}

}

### Controller实现

编写controller,用于处理客户端的请求:

**package** com.cy.pj.goods.controller;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.ResponseBody;

**import** com.cy.pj.goods.pojo.Goods;

**import** com.cy.pj.goods.service.GoodsService;

@Controller

@RequestMapping("/goods/")

**public** **class** GoodsController {

@Autowired

**private** GoodsService goodsService;

@RequestMapping("doFindGoods")

@ResponseBody

**public** List<Goods> doFindGoods(){

**return** goodsService.findGoods();

}//json 串:spring mvc 启动API将对象转换为JSON串

}

## 项目部署及运行分析

项目部署到tomcat,然后启动进行访问测试.

<http://localhost/CGB-SSM-01/goods/doFindGoods>

# FRAMEWORK总结分析

## 重点难点

1. MyBatis基本架构及核心API应用
2. MyBatis 实现数据访问的两种基本方式
3. Mybatis 中缓存应用?(一级缓存,二级缓存)
4. Spring 中的IOC模块设计原理及Bean对象管理
5. Spring 中MVC分层设计思想及核心组件作用.
6. …

## FAQ分析

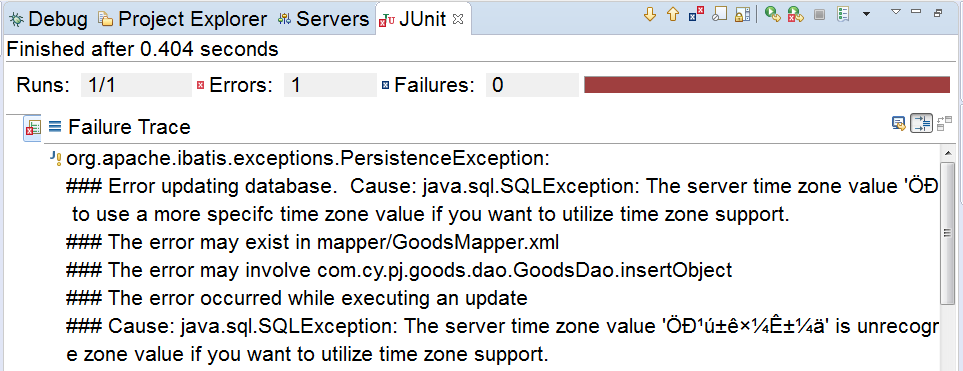
1. 如何理解mybatis框架?(优势,劣势)
2. 如何理解spring框架?(核心优势:资源整合,优化资源使用,解耦)
3. 如何理解 @Autowired 这个注解？

@Autowired注解用于修饰类中属性，构造方法，set方法，用于告诉spring框架请按照类型(属性类型，构造方法参数类型，set方法参数类型)为属性注入值。假如spring容器提供了多个相同类型对象，此时还可以按照名字（属性名，set方法参数名，构造方法参数名）查找匹配对象。

4………….

## BUG 分析

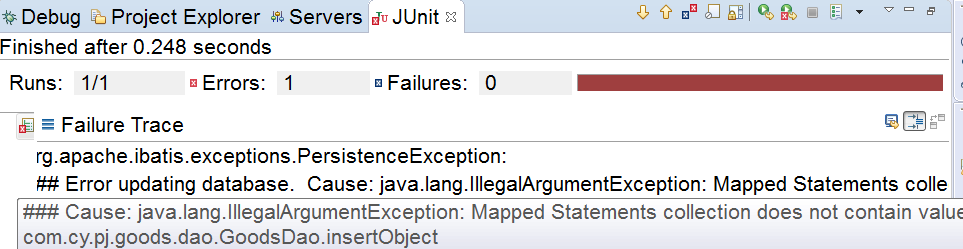
* 时区配置问题



解决方案：

1. 要么降低数据库驱动版本
2. 要么在url后追加serverTimezone=GMT

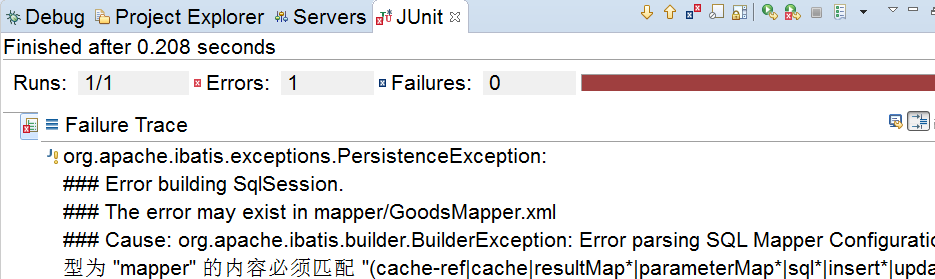
* 映射语句不存在



解决方案：

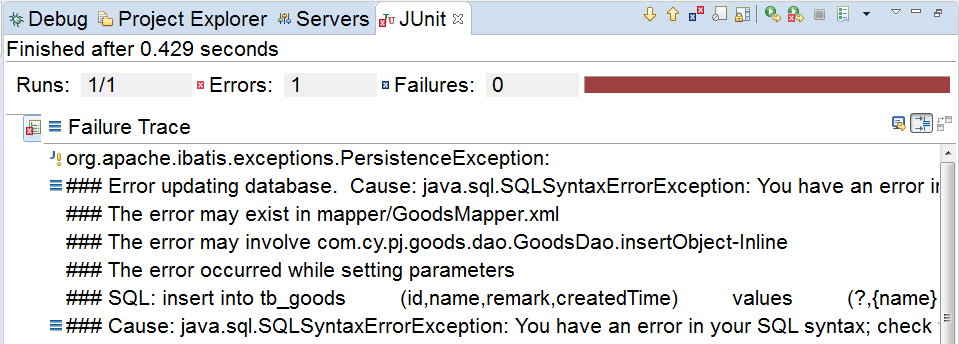
1. 检查映射文件是否注册(映射文件是否在配置文件中已配置)
2. 检查statement语句与映射文件中namespace+元素id是否相同。

* Mapper文件内容错误



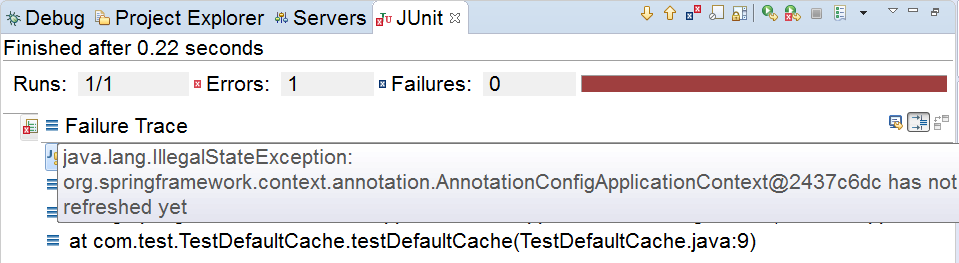
解决方案：检测配置文件中的语法，特殊字符。

* SQL语法错误



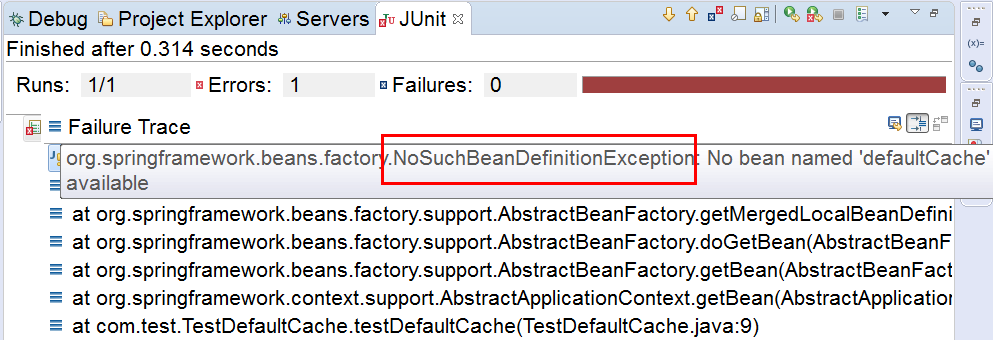
解决方案：基于映射检查SQL的具体语法。

* 问题:Spring容器初始化失败



解决方案：检查容器初始化是否读取了配置类或配置文件

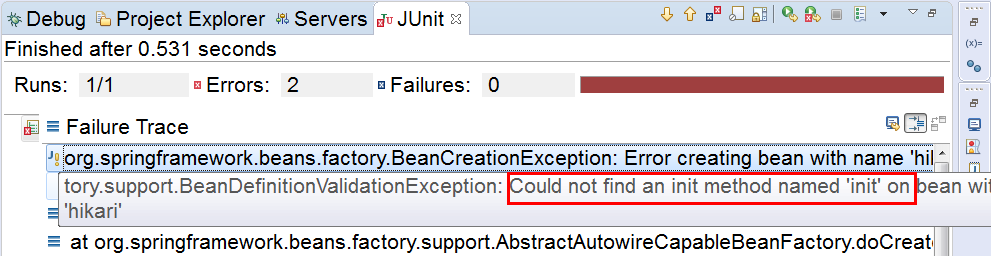
* 问题:没有找到对应的Bean对象：



解决方案：

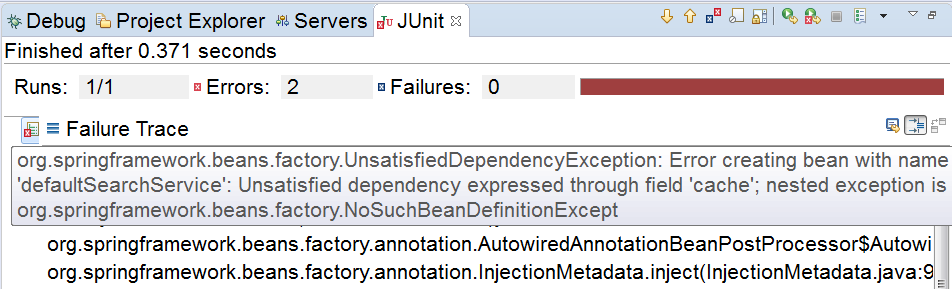
1. 对象所在的包是否在spring框架的扫描范围内
2. 对象对应的类上是否有spring指定的特定注解描述(例如@Component)
3. 从Spring容器获取Bean对象时指定的名字是否正确。

* 问题：整合HikariCP连接池错误



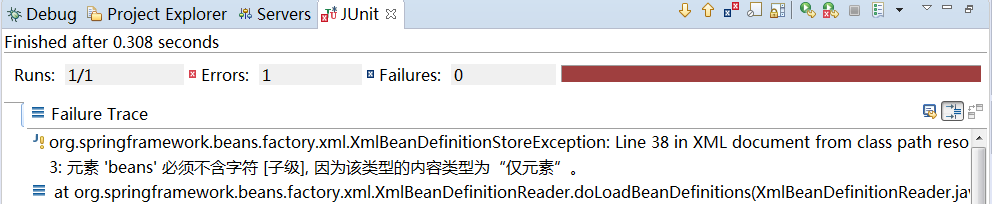
解决方案：HikariDataSource连接池对象不能指定init方法，它没有。

* 问题：defaultSearchService对象中无法为Cache属性注入值



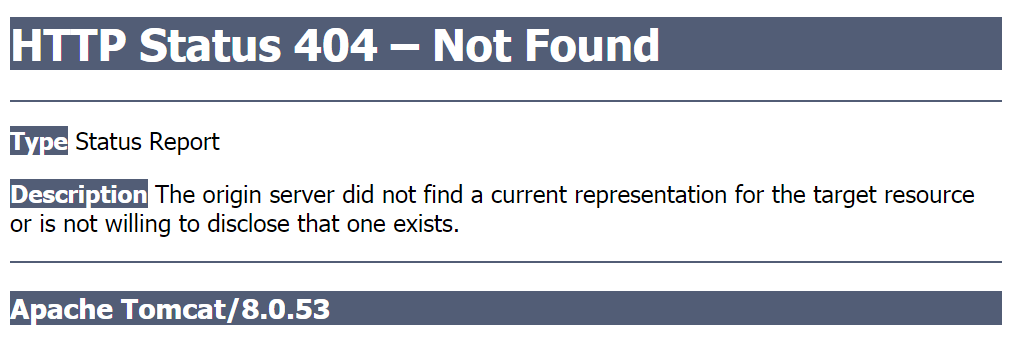
解决方案：检查spring容器中是否有Cache类型的对象。

* Spring配置文件编写错误



问题分析及解决方案：应该是配置文件内部不小心多了一些特殊字符，并且写在了标签外面，例如分号，引号等

* 请求资源不存在



问题分析及解决方案：404问题一般表示请求资源不存在，解决方案如下：

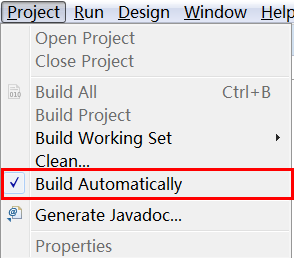
* + - 1. 确定tomcat服务启动ok
      2. 确定tomcat启动时加载的web.xml有前端控制器的配置（可选）
      3. 确定浏览器中的url资源访问路径是正确的。
      4. 确认要访问的资源配置是正确的(@Controller,包扫描)
      5. 确定部署目中有你对应的资源(资源不存在可能是编译有问题)
      6. 确定你的项目没有重命名。(重命名的项目可使用命名前方式访问)

如何确定你要访问的资源是否是存在的？查看tomcat启动日志？

-Dwtp.deploy=D:\CGB1902WORKSP\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps

资源不存在可能是项目没有正常编译，如何解决编译问题呢？

1)确保eclipse 中选中的自动编译



2）重新发布和部署项目，检测是否能够解决问题

3）尝试四大clean操作(先停止tomcat，从eclipse中移除项目)

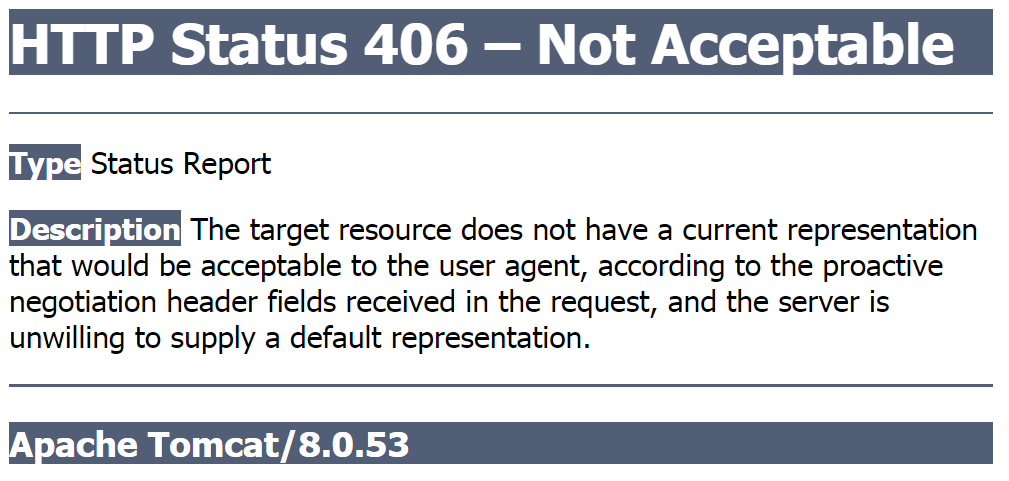
3.1)clean tomcat work目录

3.2)clean tomcat 的部署文件

3.3）maven clean 删除原先编译的class

3.4）project clean 重新编译项目代码

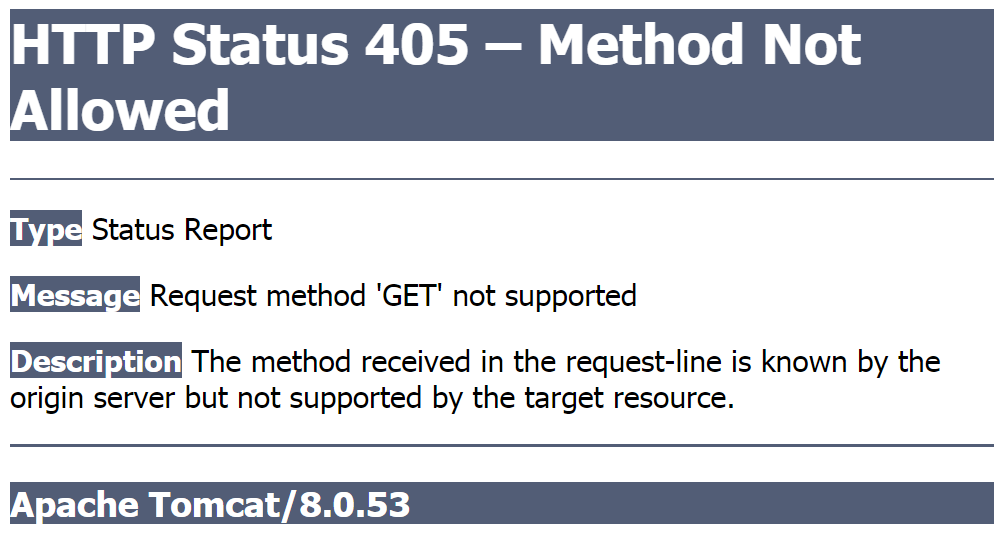
* 响应数据问题



问题分析：spring 响应数据客户端无法处理

解决方案：检测spring 配置（例如jackson依赖，默认bean的配置等）

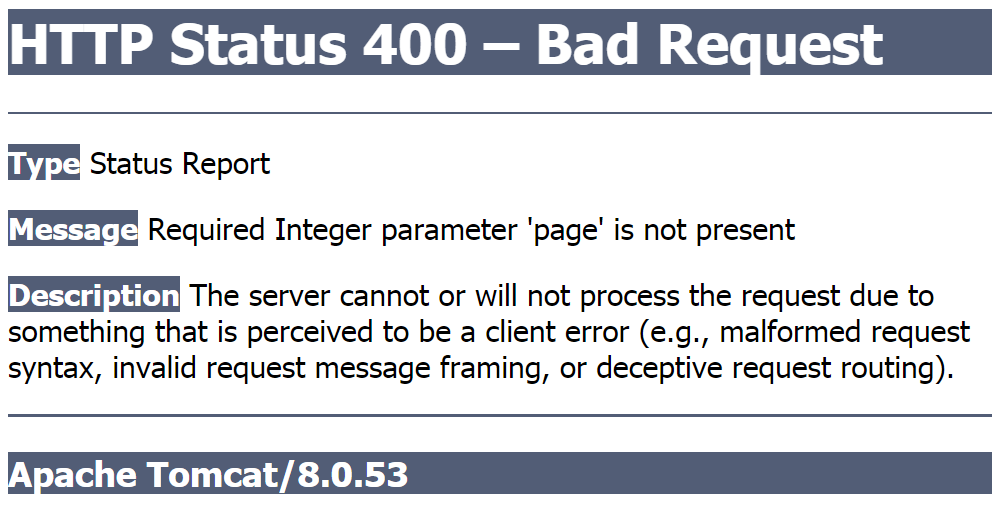
* 请求方式不匹配



问题分析：请求方式与服务端的处理方式不匹配。

解决方案：修改请求处理方式

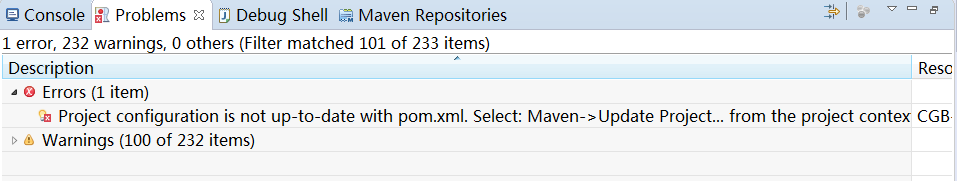
* 请求参数不匹配(例如个数，格式等)



问题分析：客户端请求参数与服务端控制层方法参数不匹配(

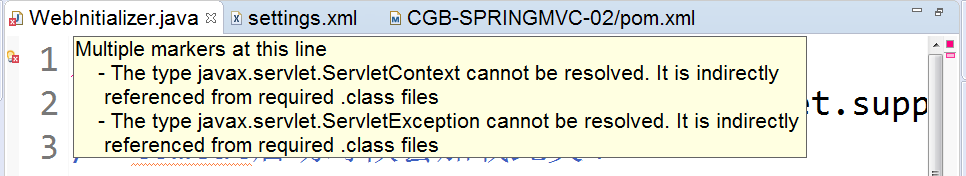
参数个数，参数格式)

* 项目有叹号或者红色叉号

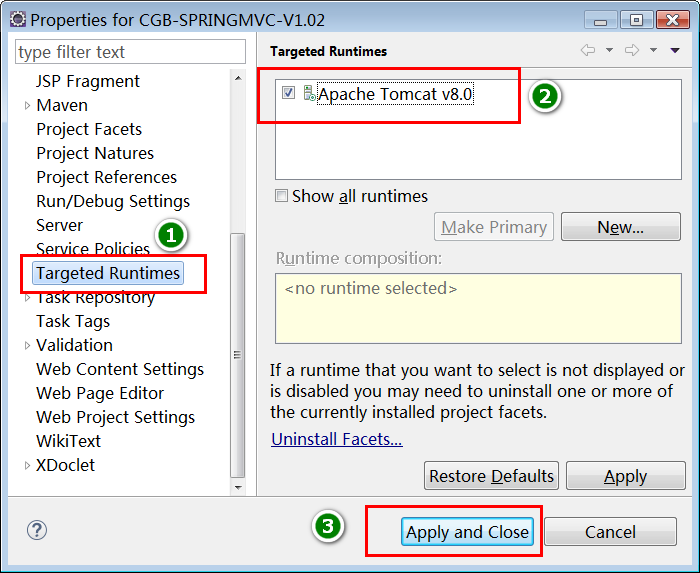


问题分析:先检测项目的problems，对象项目进行重新maven update操作

* 项目没有添加运行时依赖

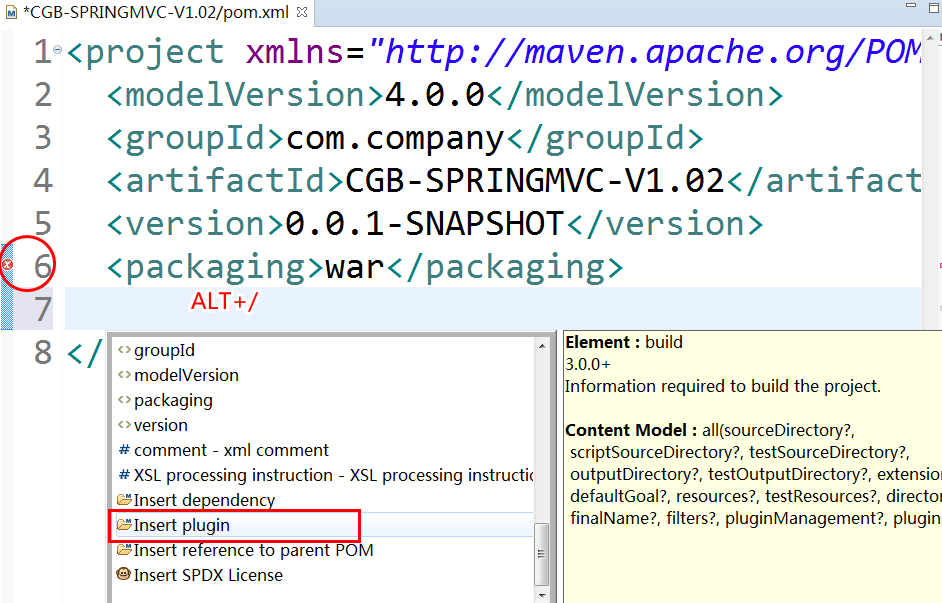


问题分析:项目没有添加运行时环境，右键项目,选择属性,然后添加项目的运行时环境

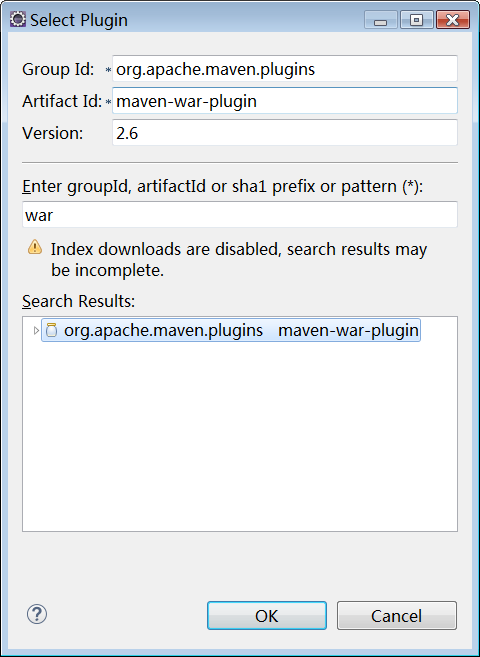


* 项目中没有web.xml，其处理方式

当项目中没有web.xml,又不想添加此web.xml时,可以修改war包插件



选择war包查找



案例如下:

<build>

<plugins>

<!-- 修改war包插件的配置 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.6</version>

<configuration>

<!-- 通过此配置可以没有web.xml -->

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

</plugins>

</build>