JavaEE

1 Hibernate入门简介

1.1 Hibernate环境搭建

1. 加入hibernate框架相关的jar包(lib/required)
2. 编写hibernate的配置文件：默认文件名称 hibernate.cfg.xml

- 创建src/hibernate.cfg.xml (hibernate.properties)
- xml文件编写规则（.dtd或者.xsd），使用dtd文件
- 如果xml文件中不能智能提示，则需要配置dtd文件
- 配置<session-factory>节点：如数据库连接信息

1.1.1 配置文件

- 采用mysql-connector-java-`8.0.13`.jar,xml无问题,可能是此包版本问题

```
<!-- 采用mysql-connector-java-8.0.13 -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<!-- session-factory代表一个数据库：配置property和mapping -->
```

```
    <session-factory>
        <!-- 方言：指定hibernate要生成何种数据库的SQL语句 -->
        <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>

        <!-- 数据库连接 -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/test?characterEncoding=UTF-8&amp;serverTimezone=UTC&amp;useSSL=false</property>
        <property name="connection.username">root</property>
        <property name="connection.password">密码</property>

        <!-- 打印SQL语句 -->
        <property name="show_sql">true</property>
        <!-- 自动生成表结构 -->
        <property name="hbm2ddl.auto">update</property>
        <!-- mapping映射(xml或注解在哪) -->
        <mapping class="com.bfs.entity.User"/>
    </session-factory>
</hibernate-configuration>
```

1.1.2 编写实体类（不需要建表）

- 在类上使用注解：@Entity，@Table
- 在属性上使用注解：@Column

```
@Entity
@Table(name = "tb_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // 自增
    private Long id;
    // String类型映射到数据库字段的长度，默认是255
    @Column(length = 50)
    private String name;
    private int age;
    // 省略setter、getter方法
}
```

1.1.3 测试

- 增加一条记录
- 按主键查询

```
public class Test3 {
    @Test
    public void test1() {
```

```
        Session s = HibernateUtil.openSession();
        Transaction tx = s.beginTransaction();

        try {
            User u = new User();
            u.setName("李四");
            u.setAge(20);
            s.saveOrUpdate(u);
            tx.commit();
        } catch (Exception e) {
            tx.rollback();
        } finally {
            HibernateUtil.closeSession();
        }
    }

    @Test
    public void query1() {
        Session s = HibernateUtil.openSession();

        String hql = "from User where age>=?1";
        List<User> list = s.createQuery(hql,
User.class).setParameter(1, 20).list();
        for(User u : list) {
            System.out.println(u);
        }

        HibernateUtil.closeSession();
    }

    @Test
    public void query2() {
        Session s = HibernateUtil.openSession();
        String sql = "select * from tb_user where age>=?0";
        List<User> list = s.createNativeQuery(sql,
User.class).setParameter(0, 20).list();
        for(User u : list) {
            System.out.println(u);
        }

        HibernateUtil.closeSession();
    }

    @Test
    public void query3() {
        Session s = HibernateUtil.openSession();
        CriteriaBuilder cb = s.getCriteriaBuilder();
        CriteriaQuery<User> query = cb.createQuery(User.class);
        Root<User> root = query.from(User.class);
        query.select(root);
```

```
52          query.where(cb.ge(root.get("age"), 20));
53          List<User> list = s.createQuery(query).list();
54          for(User u : list) {
55              System.out.println(u);
56          }
57
58          HibernateUtil.closeSession();
59      }
60      @AfterClass
61      public static void destory() {
62          HibernateUtil.destory();
63      }
64  }
65
```

1.2 Hibernate核心概念

1.2.1 常用类

1. SessionFactory

- 它表示一个数据库（连接信息）
- 它是重要级对象，创建比较耗时
- **线程安全**的，可以全局共享（static）
- 一般不需要关闭

2. Session

- 它表示一次连接（包装Connection对象）
- 它是轻量级对象，用完需要关闭
- **线程不安全**的，但在线程内部**应该尽可能共享**（ThreadLocal）

3. Transaction

- 事务（事务提交、事务回滚）
- **增删改操作，需要使用事务**
- 查询操作，可以不使用事务

1.2.2 SessionFactory创建方式

1. hibernate3.x和hibernate5.x中支持

```
1  SessionFactory factory = new
   Configuration().configure().buildSessionFactory();
```

2. hibernate4.x和hibernate5.x中支持

```
ServiceRegistry registry = new
StandardServiceRegistryBuilder().configure().build();
Metadata metadata = new
MetadataSources(registry).buildMetadata();
SessionFactory factory = metadata.buildSessionFactory();
```

1.2.3 HibernateUtil工具类

1. 初始化SessionFactory
2. 尽可能复用Session
3. ThreadLocal的用法

```
public class HibernateUtil {

    private static SessionFactory factory;
    static {
        factory = new
Configuration().configure().buildSessionFactory();
    }
    // 线程局部变量
    private static ThreadLocal<Session> session = new
ThreadLocal<>();
    /**
     * 打开Session：线程不安全的，线程内部尽可能共享（线程变量）
     *  1) 先从线程变量中获得
     *  2) 如果没有再从factory中获得，并放在线程变量中
     * */
    public static Session openSession() {
        Session s = session.get();
        if(s==null || !s.isOpen()) {
            s = factory.openSession();
            session.set(s);
        }
        return s;
    }

    /**
     * 关闭Session
     * */
    public static void closeSession() {
        Session s = session.get();
        session.set(null);

        if(s!=null && s.isOpen()) {
            s.close();
        }
    }
    /**
     * 关闭SessionFactory
```

```
     * */
    public static void destory() {
        if(factory.isOpen()) {
            factory.close();
        }
    }
}
```

1.2.4 基本注解

1.2.4.1 @Table

1. @Table(name="xx")：指定映射的表名

- 如果不指定默认表名为类名
- **表名尽量不要有大写字母**
- oracle：不区分大小写（自动变成大写）
- mysql：默认window不区分大小写，但linux区分大小写
- 不可用'-'等等

2. @Table(indexes={@Index(columnList="name,age")}) 执行索引

- **索引：对查询性能影响很大**
- 备注：通常直接在数据库表中增加索引

1.2.4.2 @Entity

表明实体

```
@Entity
@Table(name = "my_user", indexes = { @Index(columnList = "name"),
@Index(columnList = "age") })
public class User {
```

1.2.4.3 @Column

默认所有属性上都有@Column，即所有属性都对应到数据库的某个字段

- @Column(name="")：映射到数据库的字段名称
- @Column(length=255)：数据库字段的字符长度（字符串类型默认长度为255）
- @Column(nullable=false, unique=true)：非空约束，唯一性约束(**会增加索引**)

```
// unique唯一性约束， 会增加索引
@Column(length=255, nullable=false, unique=false)
private String name;
```

1.2.4.4 @Transient

表示属性，但是不映射到数据库的字段

```
// 临时属性，数据库不存储该值
@Transient
private int myage;
```

1.2.5 主键生成策略

1.2.5.1 @Id

指定某属性是主键，一般使用String或Long， Integer类型

主键生成策略默认assigned(手工赋值)

1.2.5.2 @GeneratedValue生成策略

- 如果不指定生成策略，则使用AUTO
- 使用JPA自带的生成策略：strategy=共有4种取值
  - IDENTITY：自增,最常用（mysql， sqlserver等）
  - AUTO：等同SEQUENCE（如果数据库不支持序列，则通过表模拟）,**oracle中，使用自定义sequence生成策略**

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Long id;
```

1.2.5.3 自定义(hibernate提供)生成策略

1.2.5.3.1 @SequenceGenerator+@GeneratedValue

```
/**
 * oracle数据库表进行映射：
 *  主键ID自增：使用sequence，并且每张表一个sequence
 *  使用自定义生成器@SequenceGenerator
 */
@Entity
@Table(name = "my_user2")
@SequenceGenerator(name = "seq_user", sequenceName = "S_USER", allocationSize = 1) // 定义生成器
public class User2 {
    @Id
    @GeneratedValue(generator = "seq_user")
    private Long id;
```

1.2.5.3.2 @GenericGenerator+@GeneratedValue

strategy取值情况

1. native： 等同于strategy=IDENTITY

2. sequence：等同于@SequenceGenerator

```
@Entity
@Table(name = "my_user3")
@GenericGenerator(name="mynative", strategy="native") //也可以在其它类中使用
public class User3 {

    @Id
    @GeneratedValue(generator = "mynative")
    private Long id;
```

3. uuid：主键是字符串，使用hibernate生成uuid

```
@Entity
@Table(name = "my_user4")
@GenericGenerator(name="myuuid", strategy="uuid") //也可以在其它类中使用
public class User4 {

    @Id
    @GeneratedValue(generator = "myuuid")
    @Column(length = 36)
    private String id;
```

4. guid：主键是字符串，使用数据库生成uuid (select uuid())

```
@Entity
@Table(name = "my_user5")
@GenericGenerator(name="myguid", strategy="guid")
public class User5 {

    @Id
    @GeneratedValue(generator = "myguid")  //可以使用其它类定义的生成器
    @Column(length = 36)
    private String id;
```

5. assigned：不使用主键生成，需要手工赋值
6. foreign：在一对一关联关系时，可能用到

1.2.5.4 uuid和guid的区别

1. uuid：由hibernate生成，有一定的顺序， 推荐使用
2. guid：由数据库生成，select uuid()，无序

1.2.5.5 属性使用基本类型还是包装类型

1. 主键要使用包装类型：Long/Integer
2. 普通属性，使用包装类型(Integer)总体要比基本类型(int)好

- Integer：数据库字段可以为空

> 建议使用
> 查询条件：select * from tb where status is not null and status!=1

- int：数据库字段不能为空（不能将null赋值给int）

> int类型属性有默认值0
> 查询条件：不需要判断是否为空

1.2.5.6 主键使用整型还是字符串

1. 主键自增（整型）：
    - 优点：占空间小，性能好一点，有顺序，比较直观
    - 不足：分布式多库部署存在问题（合库非常麻烦）
2. uuid（字符串型）：
    - 优点：兼容所有数据库，适合分布式多库部署
    - 不足：相比整型而言，占空间大一点，性能差一点，无顺序，不直观

1.2.5.7 主键策略的选择

1. mysql, sqlserver:

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

1. oracle:

```
@SequenceGenerator(name = "seq_user", sequenceName = "S_USER",
allocationSize = 1)
@GeneratedValue(generator = "seq_user")
注意：一般每张表定义一个生成器
```

1. 通用uuid:

```
@GenericGenerator(name="myuuid", strategy="uuid")
@GeneratedValue(generator = "myuuid")
注意：uuid可以实体间共享
```

2 Hibernate关联关系映射

2.1 关联关系

- **一对多**：一个班有多个学生
- **多对一**：一个学生只属于一个班级
- **一对一**：一个人只有一张身份证（一张身份证只属于一个人）
- **多对多**：一个学生选择多门课程，一门课程有多个学生（数据库中要拆分成2个多对一）

2.2 单向关联关系和双向关联关系

- **双向关系**：通常使用双向关联（关系不紧密时，可使用单向关系）
- **单向关系**：
  - 优先使用单向的多对一
  - 单向的一对多：性能较差，由“一”的一方来维护双方关系

2.3 多向一对多

1. `@OneToMany(mappedBy="clazz")`

- one:指自身类(本类),many:指属性,前面是自身类,后面的是所标注的属性
- mappedBy="属性类的属性名":在**自身类**(班级)中**放弃维护**与**被标注属性**(学生)的关系,**参照**属性类**(学生)中的clazz属性**,one放弃维护与many的关系
- 在c.students.add(学生)的时候不发SQL。因为设置mappedBy="xx"，自身类不维护关联关系

```java
// one指自身(本类) many指属性, 前面是自身 后面的是属性
// 在班级中放弃维护与学生的关系,**参照学生中的clazz属性**
@OneToMany(mappedBy="clazz")
private Set<Student> students = new HashSet<>();
```

2. `@ManyToOne @JoinColumn(name = "clazz_id")`

- @ManyToOne:many是自身类(学生),one是属性(clazz)
- @JoinColumn(name = "clazz_id"):在**自身类**(学生)中加入一列(id) 来代替clazz对象的存储

3. 级联关系 cascade

- 类型

> - CascadeType.ALL
> - CascadeType.PERSIST

- CascadeType.MERGE
- CascadeType.REMOVE 级联删除
- CascadeType.REFRESH
- CascadeType.DETACH

- 在@OneToMany @ManyToOne都要加上(多向一对多)

```
@OneToMany(mappedBy = "comment" ,cascade = CascadeType.REMOVE)
private Set<Reply> replys = new HashSet<>();

@ManyToOne(cascade = CascadeType.REMOVE)
@JoinColumn(name = "comment_id")
private Comment comment;
```

2.4 单向多对一

在Clazz类中不设置学生属性(不维护学生属性)

```
// Student类
@ManyToOne
@JoinColumn(name = "clazz_id")
private Clazz clazz;
```

2.5 单向一对多

```
// Clazz类
@OneToMany  //维护关联关系
@JoinColumn(name = "clazz_id")
private Set<Student> students = new HashSet<>();
```

2.6 一对一

2.6.1 基于外键的一对一

特殊的多对一， 多方 的外键增加唯一性约束

- 两边**都**使用@OneToOne，如果一边没有指定mappedBy="xxx"
- 那么相当于省略了：@JoinColumn(name="xx")

1. Person

```
@Entity
@Table(name = "tb_person")
public class Person {

```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // 姓名
    @Column(length = 50)
    private String name;

    // 拥有一张身份证
    @OneToOne
    @JoinColumn(name="card_id", unique=true)// tb_person多一列
    private IdCard card;
}
```

2. IdCard

```
@Entity
@Table(name = "tb_idcard")
public class IdCard {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // 号码
    @Column(length = 20)
    private String sno;

    // 只属于一个人
    @OneToOne(mappedBy="card")
    private Person person;
}
```

3. 测试文件

```
public class T1 {

    @Test
    public void t1() {
        Session session = HibernateUtil.openSession();
        Transaction tx = session.beginTransaction();

        Person p = new Person();
        p.setName("张三");

        IdCard card = new IdCard();
        card.setSno("1234");

// 设置关联关系
        p.setCard(card); //有用

```

```
17  // 下面语句没有作用，不发SQL。因为设置mappedBy="xx"，不维护关联关系
18          card.setPerson(p);
19
20  // 保存对象：注意顺序
21          session.save(card);
22          session.save(p);
23
24          tx.commit();
25          HibernateUtil.closeSession();
26      }
27
28      @Test
29      public void t2() {
30          Session session = HibernateUtil.openSession();
31
32          String hql = "from Person";
33          List<Person> list = session.createQuery(hql,
    Person.class).list();
34          for(Person p : list) {
35              System.out.println(p.getName());
36
37              if(p.getCard() != null) {
38                  System.out.println(p.getCard().getSno());
39              }
40              System.out.println("===========");
41          }
42
43          HibernateUtil.closeSession();
44      }
45  }
```

2.6.2 共享主键的一对一

- 主对象的ID自增,@PrimaryKeyJoinColumn
- 从对象的ID不能自增，需要参照主对象的ID

1. Person

```
1   @Entity
2   @Table(name = "tb2_person")
3   public class Person2 {
4
5       // 主对象的ID自增
6       @Id
7       @GeneratedValue(strategy = GenerationType.IDENTITY)
8       private Long id;
9
10      // 姓名
11      @Column(length = 50)
12      private String name;
```

```
13
14      // 拥有一张身份证
15      @OneToOne
16      @PrimaryKeyJoinColumn //默认是@JoinColumn
17      private IdCard card;
18  }
```

2. IdCard

```
1   @Entity
2   @Table(name = "tb2_idcard")
3   public class IdCard2 {
4
5       // 从对象的ID不能自增，需要参照主对象的ID
6       @Id
7       @GeneratedValue(generator = "fk")
8       @GenericGenerator(
9           name="fk",
10          strategy="foreign",
11          parameters=@Parameter(name="property", value="person")
12      )
13      private Long id;
14
15      // 号码
16      @Column(length = 20)
17      private String sno;
18
19      // 只属于一个人
20      @OneToOne(mappedBy = "card")
21      private Person person;
22  }
```

2.7 多对多

2.7.1 生成中间表

1. Student

```
1   @Entity
2   @Table(name = "tbx_student")
3   public class Student {
4
5       @Id
6       @GeneratedValue(strategy = GenerationType.IDENTITY)
7       private Long id;
8
9       // 姓名
10      @Column(length = 50)
11      private String name;
```

```
12
13      // 有多门课
14      // 查询学生的课程：根据学生ID去中间表，和student_id匹配，再用查
    询到course_id和课程ID查询
15      @ManyToMany
16      @JoinTable(name="tbx_student_course",
17              joinColumns= @JoinColumn(name="student_id"),
18              inverseJoinColumns=@JoinColumn(name="course_id"))
19      private Set<Course> courses = new HashSet<>();
20  }
```

2. Course

```
1   @Entity
2   @Table(name = "tbx_course")
3   public class Course {
4
5       @Id
6       @GeneratedValue(strategy = GenerationType.IDENTITY)
7       private Long id;
8
9       // 名称
10      @Column(length = 50)
11      private String name;
12
13      // 有多个学生，放弃维护关联关系（不发SQL）
14      @ManyToMany(mappedBy = "courses")
15      private Set<Student> studnets = new HashSet<>();
16  }
```

3. test

```
1   // 设置关联关系
2   s1.getCourses().add(c1); //有用
3   s1.getCourses().add(c2); //有用
4   s2.getCourses().add(c2); //有用
5
6   // 下面语句没有作用，不发SQL。因为设置mappedBy="xx"，不维护关联关系
7   c1.getStudnets().add(s1);
8   c2.getStudnets().add(s2);
```

2.7.2 拆分2个一对多(手动中间表)

1. Student

```
1   @Entity
2   @Table(name = "tbx2_student")
3   public class Student {
4
5       @Id
6       @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    // 姓名
    @Column(length = 50)
    private String name;

    // 有多个成绩
    @OneToMany(mappedBy = "student")
    private Set<Score> cjs = new HashSet<>();
}
```

2. Course

```
@Entity
@Table(name = "tbx2_course")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // 名称
    @Column(length = 50)
    private String name;

    /// 有多个成绩
        @OneToMany(mappedBy = "course")
    private Set<Score> cjs = new HashSet<>();
}
```

3. Score(中间表)

```
/**
 * 成绩表
 */
@Entity
@Table(name = "tbx2_score")
public class Score {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // 属于一个学生
    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student2 student;

    // 属于一门课程
    @ManyToOne
    @JoinColumn(name = "course_id")
```

```
20      private Course2 course;
21
22      // 分值
23      private int val;
24  }
```

3 Hibernate增删改查

3.1 分类

1. 增加：save(), saveOrUpdate()
2. 修改：update(), saveOrUpdate()
3. 删除：delete() //按ID删除
4. 查询：
    - 按ID查询：get()/load()
    - HQL查询（JPQL）：推荐使用（类似SQL）
    - 条件查询：不推荐使用（特殊繁琐）
    - 原生SQL查询：特殊场合下使用

3.2 HQL

3.2.1 HQL查询语言

HQL类似于SQL，但完全不同，HQL是面向对象的

1. 查询结果集
2. DML风格查询：

    增加： `insert into tb(x, y) select xx, yy from tb2`

    修改： `update tb set x=xx`

    删除： `delete from tb`

3.2.2 HQL语句的大小写敏感问题

1. 类名和属性名：大小写敏感，如：Student
2. 其它：大小写不敏感，如：select，SELECT，Select，SElect

3.2.3 HQL占位符

1. 位置占位符：`?1,?2,...`

- setParameter(1, 25)

2. 命名占位符: `:xxx`

- setParameter("age", 25)

```
Session session = HibernateUtil.openSession();

//String hql = "from User where age>?1";    //位置占位符
//String hql = "from User u where u.age>?1";
String hql = "select u from User u where u.age>?1";
List<User> list = session.createQuery(hql,
User.class).setParameter(1, 25).list();
for(User u : list) {
    System.out.println(u);
}

// 命名占位符 :xx
String hql2 = "from User where age>:age and name like ?1";
List<User> list2 = session.createQuery(hql2, User.class)
    .setParameter("age", 25)
    .setParameter(1, "test%").list();
for(User u : list2) {
    System.out.println(u);
}
HibernateUtil.closeSession();

```

3.3 增加 save(), saveOrUpdate()

3.3.1 save()

```
User u = new User();
u.setName("张三");
u.setAge(20);
session.save(u); //u.id的值由insert语句获得的，发SQL语句
```

3.3.2 saveOrUpdate()

数据库中存在有这个id的对象为更新,不存在为save

```
User4 u = new User4();
u.setName("张三");
u.setAge(20);
session.saveOrUpdate(u); //不发SQL，u.id的值由Hibernate生成
```

3.3.3 批量操作

```
for(int i=1; i<=20; i++) {
    // 方式1：通过ID构造
    //Clazz clazz = new Clazz();
    //clazz.setId(i%2+1L);

    // 方式2：从数据库查询
    Clazz clazz = session.get(Clazz.class, i%2+1L);

    // 学生
    Student stu = new Student();
    stu.setName("test_" + i);
    stu.setAge(20 + i);
    stu.setClazz(clazz); //保证clazz的id不为null
    session.save(stu);
}
```

3.4 修改 update(), saveOrUpdate()

3.4.1 update()

```
Clazz c = session.get(Clazz.class, 1L);
c.setName("17-5");
session.update(c);
```

3.4.2 批量操作

```
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();

// HQL的批量修改
String hql = "update Student set name=?1 where age>=?2";
session.createQuery(hql).setParameter(1, "测试").setParameter(2,
38).executeUpdate();

tx.commit();
HibernateUtil.closeSession();
```

3.5 删除 delete()

3.5.1 通过ID构造对象进行删除

```
Clazz c = new Clazz();
c.setId(1L);
session.delete(c);
```

3.5.2 查询数据库获得对象

```
Clazz c = session.get(Clazz.class, 1L);
session.delete(c);
```

3.5.3 HQL语句(批量)删除

```
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();

// HQL的批量删除
String hql = "delete from Student where age>=?1";
session.createQuery(hql).setParameter(1, 38).executeUpdate();

tx.commit();
HibernateUtil.closeSession();
```

3.6 查询

3.6.1 原生SQL查询

```
/**
* 原生SQL查询
* */
@Test
    public void query_sql() {
    Session session = HibernateUtil.openSession();

    // 位置占位符和命名占位符
    String sql = "select * from tb_user where age>?1";
    List<User> list = session.createNativeQuery(sql,
User.class).setParameter(1, 25).list();
    for(User u : list) {
        System.out.println(u);
    }


    String sql2 = "select * from tb_user where age>:age";
    List<User> list2 = session.createNativeQuery(sql2,
User.class).setParameter("age", 28).list();
    for(User u : list2) {
        System.out.println(u);
    }

    HibernateUtil.closeSession();
}
```

3.6.2 Hibernate的条件查询（已过时）

```java
/**
* Hibernate的条件查询（已过时）
* */
@Test
    public void query_critera() {
    Session session = HibernateUtil.openSession();

    List<User> list =
session.createCriteria(User.class).add(Restrictions.gt("age",
25)).list();
    for(User u : list) {
        System.out.println(u);
    }

    HibernateUtil.closeSession();
}
```

3.6.3 HQL单表查询

- 创建查询语句：`session.createQuery(hql, Student.class)`
- 设置参数：`session.setParameter(1, 30)`
- 得到查询列表：`session.list()`
- 结果数组：`session.uniqueResult()`：得到 `Object[] arr;`
- 按照ID查询：`session.get(类型.class, 1L);`：如果数据库没有该记录，则返回null，用到该对象时抛NullPointerException

1. List

```java
//String hql = "from Student where age>=?1";
//String hql = "from Student s where s.age>=?1";
String hql = "select s from Student s where s.age>=?1";

List<Student> list = session.createQuery(hql,
Student.class).setParameter(1, 30).list();
```

2. List<Object[]>

```java
// 查询部分字段：每条记录是Object[]
String hql = "select id, name, age from Student where age>=?1";

List<Object[]> list = session.createQuery(hql,
Object[].class).setParameter(1, 30).list();
```

3. List

```
// 查询部分字段：每条记录是 List
String hql = "select new list(id, name, age) from Student where age>=?1";

List<List> list = session.createQuery(hql, List.class).setParameter(1, 30).list();
```

4. List

```
// 查询部分字段：每条记录是 List
String hql = "select new map(id as id, name as name, age as age) from Student where age>=?1";

List<Map> list = session.createQuery(hql, Map.class).setParameter(1, 30).list();
```

3.6.4 HQL连接查询

1. 逗号关联(存在笛卡尔积)

```
// 存在笛卡尔积
// String hql = "from Student, Clazz";
String hql = "select s, c from Student s, Clazz c";

List<Object[]> list = session.createQuery(hql, Object[].class).list();
```

2. 逗号关联(解决笛卡尔积)

```
// 模仿sql，解决笛卡尔积（糟糕方式）
String hql = "from Student s, Clazz c where s.clazz.id=c.id and c.name=?1 and s.age<?2";

List<Object[]> list = session.createQuery(hql, Object[].class).setParameter(1, "17-1班").setParameter(2, 30).list();
```

3. 对象隐式关联

```
// 隐式的关联：某班级中年龄<=30
String hql = "from Student where clazz.name=?1 and age<?2";

List<Student> list = session.createQuery(hql, Student.class).setParameter(1, "17-1班").setParameter(2, 30).list();
```

4. 对象显式关联

```
// 显式的关联：连接查询
// 内连接：join，外连接（左连接：left join、右连接：right join）
// sql的连接查询：select x.*, y.* from tb_student x join tb_clazz on x.clazz_id=y.id where ...
String hql = "from       Student s left join s.clazz c     where c.name=?1 and s.age<=?2";

// form前省略：select s, c
List<Object[]> list = session.createQuery(hql, Object[].class).setParameter(1, "17-1班").setParameter(2, 30).list();
```

5. 内连接

```
// 连接查询（内连接）
String hql = "select s from Student s join s.clazz c where c.name=?1 and s.age<=?2";

List<Student> list = session.createQuery(hql, Student.class).setParameter(1, "17-1班").setParameter(2, 30).list();
```

3.6.5 HQL分页和排序

- `setFirstResult(0)` //从第几条记录开始
- `setMaxResults(5)` //每页大小

```
// 排序和分页
// 先按age降序排，如果age相同则再按name升序排
String hql = "from Student order by age desc, name asc";
List<Student> list = session.createQuery(hql, Student.class)
                    .setFirstResult(0)   //从第几条记录开始，  从0开始算
                    .setMaxResults(5)    //每页大小
                    .list();
```

3.6.6 HQL迫切连接查询

- join：连接查询，返回Object[]
- join fetch：迫切连接查询，返回单个对象,性能优化

```
// join：连接查询，返回Object[]
// join fetch：迫切连接查询，返回单个对象
// 迫切连接查询：join fetch（性能优化）
String hql = "from Student s join fetch s.clazz c where c.name=?1
and s.age<=?2";
// 返回Student集合，并将clazz对象填充到student中

List<Student> list = session.createQuery(hql,
Student.class).setParameter(1, "17-1班").setParameter(2,
30).list();
```

3.6.7 聚合查询

一般与分组group by 一起使用

- count()
- avg()
- max()
- min()
- sum()

```
// 错误：String hql = "select * from Student";
String hql = "select count(*), min(age), max(age), avg(age),
sum(age) from Student";
Object[] arr = session.createQuery(hql,
Object[].class).uniqueResult();
System.out.println(Arrays.toString(arr));
```

3.6.8 查询函数（日期）

1. JPQL标准函数

- concat(c1, c2, ...)：字符串拼接

```
// concat(c1, c2, c3)：字符串拼接
String hql = "select concat(name, '_', age) from Student";
List<String> list = session.createQuery(hql,
String.class).list();
```

- substring(c1, begin, len)：begin从1开始的

```
// substring(c1, begin, len)：begin从1开始的
String hql = "select substring(name, 2, 2) from Student";
List<String> list = session.createQuery(hql,
String.class).list();
```

- upper(c), lower(c)
- trim(c), length(c)
- abs(c), mod(c), sqrt(c)

- current_date(), current_time()

```
// 时间相关函数current_date(), current_time(), current_timestamp()
String hql = "select current_date(), current_time(), name from Student";
List<Object[]> list = session.createQuery(hql, Object[].class).list();
```

2. HQL函数

- cast(c as string)：类型转换
- extract(year from c)：抽取时间
- year(c), month(c), day(c)
- hour(c), minute(c), second(c)

```
// year(字段), month(), day(), hour(), minute(), second()
String hql = "select year(current_date), month(current_date()), name from Student";
List<Object[]> list = session.createQuery(hql, Object[].class).list();
```

- str(c)：转成字符串

3. 集合处理表达式

- size(集合)

```
// 查找没有学生的班级
String hql = "from Clazz where size(students)=0";
List<Clazz> list = session.createQuery(hql, Clazz.class).list();
```

- maxelement(集合), minelement(集合)
- [some|exists|all|any] elements(集合)

4. case表达式

> CASE {operand} WHEN {test_value} THEN {match_result} ELSE {miss_result} EN

5. nullif表达式

> select nullif(p.nick, p.name) from xxx

3.6.9 子查询

- where xx in(select xx from tb)

```
// HQL中，如果条件是对象，实质上指对象的id
String hql = "from Student where clazz in (select c from Clazz c)";
List<Student> list = session.createQuery(hql, Student.class).list();
```

3.6.10 分组查询

- group by 对象 ==> group by 对象.id
- group by 对象.属性 having 条件
- where和having的区别
  - where：在分组之前，过滤条件（不可以使用聚合函数）
  - having：在分组之后，过滤条件（可以使用聚合函数）

1. example1

```
String hql = "select s.clazz, count(*), min(s.age), max(s.age) from Student s group by s.clazz";
List<Object[]> list = session.createQuery(hql, Object[].class).list();
```

2. example2

```
String hql = "select s.clazz, count(*), min(s.age), max(s.age) from Student s "
    + "group by s.clazz having max(s.age)>=?1";
List<Object[]> list = session.createQuery(hql, Object[].class).setParameter(1, 40).list();
```

3. example3

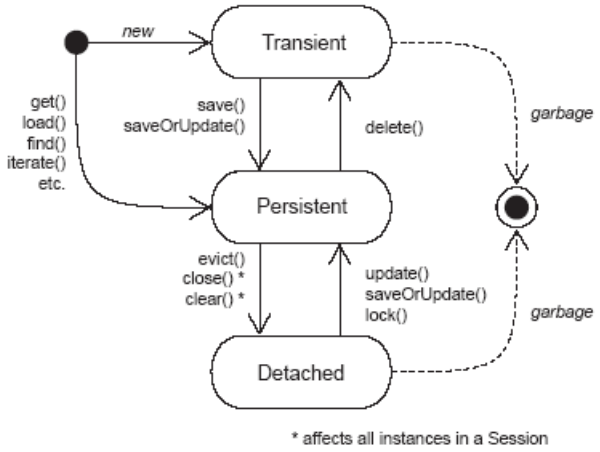
```
// age>=30的学生，再进行分组
String hql = "select s.clazz, count(*), min(s.age), max(s.age) from Student s "
    + "where age>=30 group by s.clazz having max(s.age)>=?1";
List<Object[]> list = session.createQuery(hql, Object[].class).setParameter(1, 40).list();
```

3.7 对象的生命周期

3.7.1 生命周期状态

1. 瞬时状态(Transient)：使用new操作符得到的对象，没有和数据库表进行关联。(数据库中没有与之对应的纪录)

2. 持久状态(Persist)：持久对象是任何具有数据库标识的实例，它由Session统一管理。它们的状态在事务结束时同数据库进行同步。（数据库中有与之对应的纪录，并受session管理） **对持久状态对象的修改，会自动同步到数据库**，同步的同时(set方法)**不发sql**,在 `commit()/flush()` 时候**统一**发sql
3. 脱管状态(Detached)：Session关闭或调用clear()或evict()，不受Session的管理。（数据库中有与之对应的纪录，但不受session管理）




```java
@Test
public void t1() {
    Session session = HibernateUtil.openSession();
    Transaction tx = session.beginTransaction();

    Clazz clazz = new Clazz();
    clazz.setName("18-1");
    // 此时，clazz对象是瞬时状态（id是null）

    session.save(clazz);
    // 此时，clazz对象是持久状态（id有值）

    tx.commit();
    HibernateUtil.closeSession();
}
@Test
public void t2() {
    Session session = HibernateUtil.openSession();
    Transaction tx = session.beginTransaction();

    Clazz clazz = session.get(Clazz.class, 2L);
    // 此时，clazz对象是持久状态（id有值）

    clazz.setName("abcd");
    // 注意：对持久状态对象的改变，会自动同步到数据库（可能导致严重问题）

    // session.saveOrUpdate(clazz); //可选

```

```
        tx.commit();
        HibernateUtil.closeSession();
    }
```

3.7.2 函数

1. session.clear()：将所有对象从session中逐出

```
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();

Clazz clazz = session.get(Clazz.class, 2L);
// 此时，clazz对象是持久状态（id有值）

// 强制清空session，不进行刷新session（即不与数据库同步）
session.clear();
// 原session中的所有对象都变成脱管状态

// 此时，clazz对象是脱管状态，数据库中有与之对应的纪录但不受session管理
clazz.setName("1123");
// 不自动同步到数据库

// 通过update()重新变成持久状态
// session.saveOrUpdate(clazz);

tx.commit();
session.close();
```

2. session.evict(obj)：只将该对象从session中逐出，不影响其它对象

```
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();

Clazz clazz = session.get(Clazz.class, 2L);
// 此时，clazz对象是持久状态（id有值）

clazz.setName("aaaaaa");

// saveOrUpdate()此时不会发SQL语句
session.saveOrUpdate(clazz);

// 将clazz对象逐出session
session.evict(clazz);

tx.commit();
session.close();
```

3. session.flush()：强制刷新session，与数据库同步，发SQL语句

```
Session session = HibernateUtil.openSession();
```

```
Transaction tx = session.beginTransaction();

Clazz clazz = session.get(Clazz.class, 2L);
// 此时，clazz对象是持久状态（id有值）

clazz.setName("aaaaaa");

// 同步session中对象状态到数据库
session.flush(); //发SQL语句

// 将clazz对象逐出session
session.evict(clazz);

tx.commit();
session.close();
```

3.7.3 hibernate什么时候会SQL语句

1. `save()` 需要获得ID值,当**有ID不会发SQL**

```
User u = new User();
u.setName("张三");
u.setAge(20);
session.save(u); //u.id的值由insert语句获得的，发SQL语句

User4 u = new User4();
u.setName("张三");
u.setAge(20);
session.saveOrUpdate(u); //不发SQL，u.id的值由Hibernate生成

User5 u = new User5();
u.setName("张三");
u.setAge(20);
session.save(u); //u.id的值由数据库select uuid()生成，发SQL
```

2. `commit()` 前调用 `flush()` 方法，刷新session，与数据库同步

```
Clazz clazz = session.get(Clazz.class, 1L);
// 此时，clazz对象是持久状态（id有值）
clazz.setName("aaaaaa"); // 不发sql
// 同步session中对象状态到数据库
session.flush(); //发SQL语句
clazz.setName("bbbb");
tx.commit();//发SQL
```

3. `commit()` 提交事务时(之前没有 `flush()` ,或者 `flush()` 后没有改变对象)

`delete()` , `update()` , `save()有ID时` 不会发SQL,而是等到 `commit()` 统一发

```java
Clazz clazz = session.get(Clazz.class, 1L);
// 此时，clazz对象是持久状态（id有值）
clazz.setName("aaaaaa"); // 不发sql
// 同步session中对象状态到数据库
session.flush(); //发SQL语句
//clazz.setName("bbbb");
tx.commit();//不发SQL
```

4. `get()`方法查询数据库

3.8 其他注解

3.8.1 @Where（只映射满足条件的纪录）

用于类上,用于集合属性上

```java
@Entity
@Table(name = "tb_clazz")
@Where(clause = "delFlag is null")
public class Clazz {
```

3.8.2 @OrderBy

用于集合属性上,按数据库表字段进行排序（简单排序）

@OrderBy("**所标注属性的字段** desc, 所标注属性的字段 asc")

```java
@OrderBy("age desc, name asc")
private Set<Student> students = new HashSet<>();
```

3.8.3 @SortComparator

用于集合属性上,按业务逻辑进行Java排序（复杂排序）

```java
@SortComparator(StudentComparator.class)
private Set<Student> students = new HashSet<>();
```

4 Hibernate缓存

4.1 懒加载与迫切加载

4.1.1 get() load()

1. get()

- get()：立即发SQL，返回Clazz类型对象

- 如果数据库没有该记录，则返回null，用到该对象时抛NullPointerException

```
// get()：立即发SQL，返回Clazz类型对象
// 如果数据库没有该记录，则返回null，用到该对象时抛NullPointerException
Clazz clazz = session.get(Clazz.class, 1L);
```

2. load()

- load()：不立即发SQL（延迟加载），返回一个代理对象
- 在必要的时候，才发SQL查询（查询到的结果放在对象的target属性上）如DB没有该记录，仍然会返回代理对象，用到该对象时抛ObjectNotFoundException

```
// load()：不立即发SQL（延迟加载），返回一个代理对象
// 在必要的时候，才发SQL查询（查询到的结果放在对象的target属性上）
// 如果数据库没有该记录，仍然会返回代理对象，用到该对象时抛ObjectNotFoundException
Clazz clazz = session.load(Clazz.class, 1L);
```

4.1.2 懒加载(延迟加载)

- **一对多**：**默认使用延迟加载**（懒加载），在必要的时候，会发SQL加载
- 代表函数：`load()`
- 注解：`@OneToMany(fetch=FetchType.LAZY)` 不要改变"一对多"中默认的懒加载，如果确实需要立即加载，使用HQL中的join fetch

```
@OneToMany(mappedBy = "clazz", fetch=FetchType.LAZY)// 默认LAZY
private Set<Student> students = new HashSet<>();

// 不要改变"1对多"中默认的懒加载，如果确实需要立即加载，使用HQL中的join fetch
// join fetch：迫切连接
String hql = "select distinct c from Clazz c left join fetch c.students";
List<Clazz> list = session.createQuery(hql, Clazz.class).list();
```

4.1.3 迫切加载

- **多对一：默认使用迫切加载**
- 代表函数：`get()`
- 注解：`@ManyToOne(fetch = FetchType.EAGER)`

```
Student student = session.get(Student.class, 1L);//立即加载关联对象
```

4.2 一级缓存（session级缓存）

- 持久化对象，受session管理（持久化对象是放在session缓存中），所有查询出来的实体对象，**一个一个**地都放在session缓存中（**不是集合对象**），缓存只在同一个sessoin中有效
- **get()和load()会先优先从一级缓存中取值**
- session.close(), session.clear(), session.evict(obj)会影响session缓存

1. example1: load()使用缓存

```
Session session = HibernateUtil.openSession();

// 发SQL
Clazz clazz = session.get(Clazz.class, 1L);
System.out.println(clazz.getName());

// 获得session缓存中的数据
Set keys = session.getStatistics().getEntityKeys();
System.out.println(keys);

// 不发SQL：使用缓存
Clazz clazz2 = session.load(Clazz.class, 1L);
System.out.println(clazz2.getName());

HibernateUtil.closeSession();
```

2. example2: clear()影响缓存

```
// 清空session缓存
session.clear();

// 发SQL
Clazz clazz2 = session.load(Clazz.class, 1L);
System.out.println(clazz2.getName());
```

3. example3: 个题对象形式存在

```
// 一个一个地都放在session缓存中（不是集合对象)
//查询到的结果集list中的每个实体对象，都会放在session缓存
String hql = "from Clazz";
List<Clazz> list = session.createQuery(hql, Clazz.class).list();
System.out.println(list.size());
// 查询到的结果集list中的每个实体对象，都会放在session缓存


// 获得session缓存中的数据
Set keys = session.getStatistics().getEntityKeys();
System.out.println(keys);

// 不发SQL：使用缓存
Clazz clazz2 = session.load(Clazz.class, 1L);
```

```
System.out.println(clazz2.getName());
```

4. example4：迫切加载的对象也会被缓存

```
// 加载学生，会立即加载班级
Student student = session.get(Student.class, 1L);
System.out.println(student.getName());

// 获得session缓存中的数据
Set keys = session.getStatistics().getEntityKeys();
System.out.println(keys);

// 不发SQL：使用缓存
Clazz clazz2 = session.load(Clazz.class, 2L);
System.out.println(clazz2.getName());
```

4.3 二级缓存（sessionFactory级缓存）

4.3.1 ehcache基本知识

- 二级缓存常用ehcache缓存，是一个线程级缓存，即运行在JVM中
- **持久化状态对象**，会放入二级缓存
- get()和load()会**先使用session缓存**，**再使用二级缓存**，如果缓存没有则**查询数据库**

4.3.2 ehcache开启步骤

1. 放入ehcache的jar包
2. 编写ehcache.xml配置文件
3. 在hibernate.cfg.xml中开启二级缓存
4. 在实体类上或实体集合属性上使用@Cache注解

4.3.3 ehcache.xml配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
    <diskStore path="../ehcache/hibernate"/>
    <defaultCache maxElementsInMemory="10000" eternal="false"
timeToIdleSeconds="600" timeToLiveSeconds="3600"
        overflowToDisk="true" diskSpoolBufferSizeMB="30"
maxElementsOnDisk="10000000"
        diskPersistent="false"
diskExpiryThreadIntervalSeconds="600"/>

    <!-- hibernate内置的缓存区域 -->
```

```
10      <cache name="org.hibernate.cache.internal.StandardQueryCache"
    maxElementsInMemory="50" eternal="false"
    timeToIdleSeconds="3600" timeToLiveSeconds="7200"
    overflowToDisk="true"/>
11      <cache
    name="org.hibernate.cache.internal.UpdateTimestampsCache"
    maxElementsInMemory="5000" eternal="true"
     overflowToDisk="true"/>
12
13      <!-- 实体对象定制缓存，没定制的实体使用默认设置 对类使用-->
14      <cache name="com.bfs.entity.Student"
    maxElementsInMemory="1000" eternal="false"
    timeToIdleSeconds="600" timeToLiveSeconds="7200"
    overflowToDisk="true"/>
15      <cache name="com.bfs.entity.Clazz" maxElementsInMemory="1000"
    eternal="false" timeToIdleSeconds="600" timeToLiveSeconds="7200"
    overflowToDisk="true"/>
16      <!-- 实体对象定制缓存，没定制的实体使用默认设置 对集合属性使用--
    >
17      <cache name="com.bfs.entity.Clazz.students"
    maxElementsInMemory="1000" eternal="false"
    timeToIdleSeconds="600" timeToLiveSeconds="7200"
    overflowToDisk="true"/>
18
19      <!-- 手动使用缓存 -->
20      <cache name="mycache" maxElementsInMemory="1000"
    eternal="false" timeToIdleSeconds="600" timeToLiveSeconds="7200"
    overflowToDisk="true"/>
21
22  </ehcache>
23
```

4.3.4 hibernate.cfg.xml中开启二级缓存

```
1  <!-- 开启二级缓存 -->
2  <property name="cache.use_second_level_cache">true</property>
3  <!-- 开始查询缓存(开始二级缓存就开启查询缓存)       -->
4  <property name="cache.use_query_cache">true</property>
5  <!-- 设置使用的二级缓存为ehcache       -->
6  <property
   name="cache.region.factory_class">org.hibernate.cache.ehcache.int
   ernal.EhcacheRegionFactory</property>
```

4.3.5 在实体类上或实体集合属性上使用@Cache注解

```
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
```

- READ_WRITE：缓存可读写
- 标注的类或者集合属性必须**在XML中注册开启**

1. 实体类

```
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Clazz {
```

2. 属性集合

```
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
private Set<Student> students = new HashSet<>();
```

4.4 查询缓存（属于二级缓存）

4.4.1 介绍

- 缓存**查询语句**和**对应的结果集**
  - key：查询语句（条件参数不同，是不同的key）
  - value：查询到的结果集（一般是集合）
- 开启查询缓存：`setCacheable(true)`
  - 先使用查询缓存(属于二级)，如果没有则查询数据库，并将查询结果放入查询缓存

4.4.2 example

```
String hql = "from Clazz";
List<Clazz> list = session.createQuery(hql,
Clazz.class).setCacheable(true).list();
System.out.println(list);

String hql2 = "select c from Clazz c";
List<Clazz> list2 = session.createQuery(hql2,
Clazz.class).setCacheable(true).list();
System.out.println(list2);
```

4.5 缓存机制的其他问题

- 缓存的目的，查询时不用去数据库查询而直接从内存读入
- 缓存的数据和数据库的纪录必须保持一致，或者缓存不可用

4.5.1 一级缓存溢出

如果循环插入的数据量比较多时，需要及时清空session缓存

save(s)之后，s对象是持久状态的，它存储在session缓存中,session缓存的内存，可能内存溢出

```java
for(int i=1; i<=1000000; i++) {
    Student s = new Student();
    s.setName("test_" + i);
    session.saveOrUpdate(s);
    // 如果循环插入的数据量比较多时，需要及时清空session缓存
    // save(s)之后，s对象是持久状态的，它存储在session缓存中
    // session缓存的内存，可能内存溢出

    if(i%1000 == 0) {
        session.flush(); //强制同步数据库
        session.clear(); //清空session缓存，避免内存溢出
    }
}
```

4.5.2 save()/update()/delete()只影响缓存中单个实体

1. 一个session中，get()同一个ID时，不会去二级缓存查找

```java
Clazz clazz = session.get(Clazz.class, 1L);
session.clear();  //清空session缓存，但不影响二级缓存
// 插入一条新记录
Clazz c = new Clazz();
c.setName("17-11");
session.saveOrUpdate(c);  //此时，会发SQL（id是数据库自增）
session.clear();
// 一个session中，get()同一个ID时，不会去二级缓存查找
session = HibernateUtil.openSession();
clazz = session.get(Clazz.class, 1L);
```

2. 验证

```java
Session session = HibernateUtil.openSession();
Clazz clazz = session.get(Clazz.class, 1L);
System.out.println("========================: " +
clazz.getId());
HibernateUtil.closeSession();

// 更新一条记录
session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
Clazz c2 = session.get(Clazz.class, 2L);
c2.setName("aaa");
session.saveOrUpdate(c2); //非必要
tx.commit();  //如果session中对象和数据库中对象不一致，则发SQL
HibernateUtil.closeSession();

// 二级缓存中数据有效：不发SQL
session = HibernateUtil.openSession();
clazz = session.get(Clazz.class, 1L);  //二级缓存（Clazz#1有效）
```

```
System.out.println("========================: " +
clazz.getId());
HibernateUtil.closeSession();
```

4.5.3 update,delete语句进行批量操作会影响整个实体缓存

update, delete语句进行批量操作时，会缓存整个实体缓存（全部不可用）

```
Session session = HibernateUtil.openSession();
Clazz clazz = session.get(Clazz.class, 1L);
System.out.println("========================: " +
clazz.getId());
HibernateUtil.closeSession();

// 更新一条记录
session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
String hql = "update Clazz set name=?1 where id=2";  //将Clazz相
关缓存对象全部标识不可用
session.createQuery(hql).setParameter(1,
"bbbb").executeUpdate();
tx.commit();
HibernateUtil.closeSession();

// 二级缓存中数据无效，会发SQL
session = HibernateUtil.openSession();
clazz = session.get(Clazz.class, 1L); //必须去数据库查询
System.out.println("========================: " +
clazz.getId());
HibernateUtil.closeSession();
```

5 Spring XML

5.1 面向对象设计原则

1. 开闭原则（The Open-Closed Principle , OCP）
    - 对扩展开放
    - 对修改关闭
2. 里氏替换原则（Liskov Substitution Principle , LSP）
    - 子类应当可以替换基类并出现在基类能够出现的任何地方
    - 子类可以扩展父类的功能，但不能改变父类原有的功能
3. 迪米特原则（最少知道原则）（Law of Demeter , LoD）
    - 降低类之间的耦合，尽量减少对其他类的依赖

    - 是否可以减少public方法和属性，是否可以修改为private等
4. 单一职责原则
    - 只能让一个类/接口/方法有且仅有一个职责
    - 所谓一个类的一个职责是指引起该类变化的一个原因
5. 接口分隔原则（Interface Segregation Principle ，ISP）
    - 一个类对一个类的依赖应该建立在最小的接口上
    - 建立单一接口，不要建立庞大臃肿的接口
    - 尽量细化接口，接口中的方法尽量少

> 单一职责强调的是接口、类、方法的职责是单一的，强调职责
> 接口分隔原则主要是约束接口，针对抽象、整体框架

6. 依赖倒置原则（Dependency Inversion Principle ，DIP）
    - 高层模块不应该依赖于低层模块，二者都应该依赖于抽象
    - 抽象不应该依赖于细节，细节应该依赖于抽象
    - 针对接口编程，不要针对实现编程
7. 组合/聚合复用原则（Composite/Aggregate Reuse Principle ，CARP）
    - 尽量使用组合/聚合，不要使用类继承。
8. 关联关系与依赖关系区分

- 关联关系：一般关联、组合、聚合，has a

```
class A {
   private B b;  //关联
}
```

- 依赖关系

```
class A {
   public void test(B b) {}
}
```

- 继承关系 is a

5.2 控制反转IoC和依赖注入DI

1. IoC和DI是在不同的角度讲同一件事件，一般使用IoC
2. IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。
3. 依赖注入的三种方式：

- 构造器注入
- setter方法注入

- 注解注入
- 依据现有bean注入

4. 只有spring容器创建的，才会依赖注入，new的不会注入

```
UserService userService = new UserService();
// userService对象不是spring容器创建的，那么无法依赖注入
System.out.println(userService.getUserDao());

ApplicationContext atx = new
ClassPathXmlApplicationContext("beans2.xml");
UserService userService2 = atx.getBean(UserService.class);
// userService2对象是spring容器创建的，会依赖注入
System.out.println(userService2.getUserDao());
```

5.3 Spring环境搭建

1. 加入Spring框架相关的jar包
2. 编写Spring的配置文件
    - xml方式
    - 注解方式
3. 实例化ApplicationContext,从spring容器中获得对象
4. Bean的配置

5.4 Spring核心技术

1. IoC：控制反转

- 依赖注入（DI）：注入依赖对象(属性)
- 控制反转和依赖注入是在不同的角度讲同一件事件，一般使用控制反转

2. AOP：面向切面编程
   动态代理

5.5 Spring历史

> spring 1.2
> spring 2.0：xml配置
> spring 2.5：引入注解配置
> spring 3.x：mvc的改进

> spring 5.x: 新特性

5.6 Spring XML配置文件

5.6.1 配置文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    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-4.1.xsd">
<!-- bean1 -->
    <bean id="userService" class="com.bfs.serive.UserService"/>
<!-- bean2 -->
    <bean id="userService" class="com.kzw.service.UserService">
        <!-- 注入依赖对象 -->
        <property name="dao" ref="userDao"/>
    </bean>
<!-- bean3 -->
    <bean id="ctime" class="java.util.Date" scope="prototype" primary="true"/>
<!-- bean4 -->
    <bean id="df" class="java.text.SimpleDateFormat">
        <constructor-arg value="yyyy-MM-dd HH:mm:ss"/>
    </bean>
    <bean id="date" class="java.util.Date" scope="prototype"/>
    <bean id="str" factory-bean="df" factory-method="format">
        <constructor-arg ref="date"/>
    </bean>

</beans>
```

5.6.2 XML配置介绍

5.6.2.1 bean的定义

```xml
<!-- 对类注入 -->
<bean id="userDao" class="com.kzw.dao.impl.UserDaoImpl"/>
```

- Spring容器，是一个map。map_key是bean的ID，map_value是根据class生成的对象

  默认是根据无参构造器new出来的，所以一般需要**提供无参的构造方法**

- 相当于 `new XX()`,class必须是具体的实现类，一般要求有无参构造方法

5.6.2.2 scope作用域

```
<!-- Date ctime = new Date() -->
<!-- scope="prototype" 每次返回不同对象 -->
<bean id="ctime" class="java.util.Date" scope="prototype"
primary="true"/>
```

- `scope="singleton"`：单例模式,默认值，每次返回**同一个对象**
- `scope="prototype"`：原型模式,每次**返回**一个**不同**的**对象**，struts2中的action必须使用该方式
- `scope="request/session"`：不常用，用于web项目中

5.6.2.3 value与ref

- value：指传入一个字符串型的值（使用属性编辑器进行类型转换）
- ref：指定引用的另一个bean的id

1. `<property name="xx" value="字符串值"/>`
2. `<property name="xx" ref="引用bean的id"/>`
   或者：

```
<property name="xx">
    <ref bean="引用bean的id"/>
</property>
```

3. 使用内嵌bean：

```
<property name="xx">
    <bean class="类型"/>
</property>
```

5.6.2.4 依赖对象的注入

```
<bean id="userService" class="com.kzw.service.UserService">
    <!-- 对类的属性注入依赖对象 -->
    <property name="dao" ref="userDao"/>
</bean>
```

- 相当于 `setXx()`
- 注入方式
  - setter注入,对于属性
  - 构造器注入,对于类
- 依据现有bean注入,对于类
  - 注解
- 注入形式
  - ref：通过ID引用另一个bean

- value：设置一个值（字符串或数值等）

5.6.2.5 带参构造器注入

```
<!--1-->
<!--
  DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
  Date date = new Date();
  String str = df.format(date);
 -->
<!-- 带参构造器注入 -->
<bean id="df" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-MM-dd HH:mm:ss"/>
</bean>
<!-- 无参构造器注入 -->
<bean id="date" class="java.util.Date" scope="prototype"/>

<!--2-->
<!-- 通过构造器方式注入：调用某带参的构造方法 -->
<bean id="user2" class="com.kzw.bean.User" primary="true">
    <constructor-arg value="100"/>
    <constructor-arg value="李四"/>
</bean>
```

5.6.2.6 依据现有bean普通方法注入

1. xml1

```
<!-- String str = df.format(date); -->
<bean id="df" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-MM-dd HH:mm:ss"/>
</bean>

<bean id="date" class="java.util.Date" scope="prototype"/>
<bean id="str" factory-bean="df" factory-method="format">
    <constructor-arg ref="date"/>
</bean>
```

2. xml2

```
<!-- 使用普通方法返回对象（不带参数） -->
<bean id="factory" class="com.kzw.factory.Factory2"/>
<bean id="user5" factory-bean="factory"
      factory-method="getInstanse"/>
<!-- 使用普通方法返回对象（带参数） -->
<bean id="user6" factory-bean="factory"
      factory-method="getInstanse">
    <constructor-arg value="200"/>
    <constructor-arg value="李四22"/>
</bean>
```

3. java

```java
public class Factory2 {

    public User getInstanse() {
        return new User(2L, "李四");
    }

    public User getInstanse(Long id, String name) {
        return new User(id, name);
    }
}
```

5.6.2.7 使用静态方法注入对象

- 在**无现有**bean情况下调动方法注入,注意要为**static方法**

1. xml

```xml
<!-- 使用静态方法返回对象（不带参数） -->
<bean id="user3" class="com.kzw.factory.Factory1"
      factory-method="getInstanse"/>
<!-- 使用静态方法返回对象（带参数） -->
<bean id="user4" class="com.kzw.factory.Factory1"
      factory-method="getInstanse">
    <constructor-arg value="100"/>
    <constructor-arg value="张三11"/>
</bean>
```

2. java

```java
public class Factory1 {

    public static User getInstanse() {
        return new User(1L, "张三");
    }

    public static User getInstanse(Long id, String name) {
        return new User(id, name);
    }
}
```

5.6.2.8 内嵌bean注入

```xml
<bean id="d" class="java.util.Date" scope="prototype"/>
<bean id="df" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-MM-dd HH:mm:ss"/>
</bean>
<bean id="str" factory-bean="df" factory-method="format"
scope="prototype">
    <constructor-arg ref="d"/>
```

```
</bean>


<!-- 内嵌bean -->
<bean id="str2" factory-bean="df" factory-method="format"
scope="prototype">
    <constructor-arg>
        <!-- 定义局部bean，不需要配置id -->
        <bean class="java.util.Date"/>
    </constructor-arg>
</bean>
```

5.6.2.9 集合属性注入

1. xml

```
<!-- 集合属性的注入 -->
<bean id="mybean" class="com.kzw.bean.MyBean">
    <property name="array">
        <array>
            <value>111</value>
            <value>111</value>
            <value>111</value>
        </array>
    </property>

    <property name="list">
        <list>
            <value>aaa</value>
            <value>aaa</value>
            <value>aaa</value>
        </list>
    </property>

    <property name="set">
        <list>
            <value>aaa</value>
            <value>aaa</value>
            <value>aaa</value>
        </list>
    </property>

    <property name="map">
        <map>
            <entry key="id" value="100"/>
            <entry key="name" value="张三"/>
            <entry key="birthday" value-ref="date"/>
        </map>
    </property>

```

```
    <property name="props">
        <props>
            <prop key="id">200</prop>
            <prop key="name">李四</prop>
            <prop key="age">30</prop>
        </props>
    </property>

    <property name="props2">
        <value>
            id=300
            name=\u738b\u4e94
            age=30
        </value>
    </property>


    <property name="cls" value="com.kzw.bean.User"/>

 </bean>
```

2. java

```
public class MyBean {
    private String[] array;
    private List<String> list;
    private Set<String> set;
    private Map<String, Object> map;
    private Properties props;
    private Properties props2;
    private Class<?> cls;
```

5.6.3 bean的生命周期

- init-method="方法名"：在bean实例化并依赖注入完成之后，调用该方法
- destroy-method="方法名"：在spring容器关闭之前，调用该方法
- 使用注解

1. xml文件

```
<!-- 通过setter方式注入：  要求有无参构造方法 -->
<bean id="user1" class="com.kzw.bean.User" init-method="init" destroy-method="close" lazy-init="true">
    <property name="id" value="1" />
    <property name="name" value="张三" />
</bean>
```

2. java文件

```java
private Long id;
private String name;
private Date ctime;

public void init() {
    System.out.println("初始化: " + id + ", " + name);
}

public void close() {
    System.out.println("销毁: " + id);
}
```

5.6.4 属性编辑器

5.6.4.1 作用

将字符串类型转换成属性需要的类型

```xml
<!-- 识别value -->
<property name="age" value="20"/>
<property name="date" value="2020-04-06 10:20"/>
<!-- value="1,张三,20" ==> User(id=1, name="张三", age=30) -->
<property name="card" value="200,1122,北京"/>
```

5.6.4.2 步骤

1. 编写属性编辑器
2. 注册属性编辑器

5.6.4.3 属性编辑器方式1--PropertyEditorSupport

1. XML文件

```xml
<bean
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <!-- 对注册器对象的属性进行map类型注入 -->
    <property name="customEditors">
        <map>
            <entry key="java.util.Date"
                   value="com.kzw.editor.MyDateEditor" />
            <entry key="com.kzw.bean.Card"
                   value="com.kzw.editor.MyCardEditor"/>
        </map>
    </property>

</bean>
```

2. MyDateEditor文件

```
public class MyDateEditor extends PropertyEditorSupport {
    /**
     * 处理字符串，并包装成需要的某类型对象
     */
    @Override
    public void setAsText(String text) throws
IllegalArgumentException {
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd
HH:mm");
        try {
            Date date = df.parse(text);
            // 包装成对象，并进行赋值，然后spring将此对象注入
            this.setValue(date);

        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

}
```

3. MyCardEditor文件

```
public class MyCardEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws
IllegalArgumentException {

        String[] arr = text.split(",");

        Card c = new Card();
        c.setId(Long.parseLong(arr[0]));
        c.setSno(arr[1]);
        c.setAddr(arr[2]);

        setValue(c);
    }

}
```

5.6.4.4 属性编辑器方式2--PropertyEditorRegistrar

1. XML文件

```xml
<!-- 注册属性编辑器 -->
<bean
class="org.springframework.beans.factory.config.CustomEditorConf
igurer">
    <property name="propertyEditorRegistrars">
        <array>
            <bean class="com.kzw.editor.MyEditorRegistrar">
                <!-- 对注册器对象的属性进行注入 -->
                <property name="format"
                          value="yyyy-MM-dd HH:mm"/>
            </bean>
        </array>
    </property>
</bean>
```

2. MyEditorRegistrar文件

```java
public class MyEditorRegistrar implements
PropertyEditorRegistrar {

    private String format = "yyyy-MM-dd HH:mm:ss";

    @Override
    public void registerCustomEditors(
                                    PropertyEditorRegistry
registry) {

        // 注册Date类型的属性编辑器
        DateFormat df = new SimpleDateFormat(format);
        registry.registerCustomEditor(
                        Date.class, new CustomDateEditor(df,
true));

        // 注册Card类型的属性编辑器
        registry.registerCustomEditor(
                                      Card.class, new
MyCardEditor());
    }

    public void setFormat(String format) {
        this.format = format;
    }

}

```

5.6.5 获取bean

- 初始化Spring容器

`ApplicationContext atx = new ClassPathXmlApplicationContext("xx.xml");`

- 获得bean

`atx.getBean(UserService.class);`

`atx.getBean("userService", UserService.class);`

```
// 初始化Spring容器
ApplicationContext atx = new ClassPathXmlApplicationContext("beans.xml");

// 根据类型获得bean
UserService service = atx.getBean(UserService.class);
service.save();// 调用方法

// 根据ID获得bean
UserService service2 = atx.getBean("userService", UserService.class);
service2.save();// 调用方法
```

使用ApplicationContext.getBean() 获取bean:

1. 根据**类型**获得bean:

- 比较方便
- 如果**同一个类型有多个bean**定义时，会**报错**（可以指定primary="true"）

2、根据**id**获得bean:

- id是唯一，所以能返回一个对象
- 如果声明转换类型不一致，会报错

6 Spring 注解

6.1 依赖注入注解

注意：需要开启依赖注入注解功能

`<context:annotation-config/>`

- @Value：注入字符串，使用属性编辑器进行类型转换
- @Autowired：spring提供，只根据类型进行注入(**类型兼容**,接口,继承等等)
- @Resource：jdk提供，先根据属性名在spring容器中根据id查找bean，如果没有则根据类型查找

- @Resource可能需要查2次
- 根据属性名查找到的bean，如果bean的类型和属性的类型不一样，会报错

6.2 Bean定义注解

注意：需要开启组件自动扫描

```
<context:component-scan base-package="com.bfs" />
```

扫描指定路径下的所有类中的 @Component，@Repository，@Service，@Controller多个路径以逗号隔开,被扫描到的成为Spring中的Bean,**如果里面有依赖注入注解,便会进行依赖注入**

- @Repository：通常用在DAO类上
- @Service：通常用于Service类上
- @Controller：通常用于Action类上
- @Component：用于其它情况
  这四个注解，功能(定义Bean)是一样的。在spring mvc中，@Controller有特殊用途

1. xml

```
<!-- 开启依赖注入的注解 -->
<context:annotation-config/>
<!-- 开启bean定义的注解：组件自动扫描-->
<context:component-scan base-package="com.kzw2" />

<!-- 注册属性编辑器 -->
<bean
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <array>
            <bean class="com.kzw.editor.MyEditorRegistrar">
                <property name="format" value="yyyy-MM-dd HH:mm"/>
            </bean>
        </array>
    </property>
</bean>
```

2. java

```
@Component
public class User {

    @Value("1")
    private Long id;
```

```
6
7       @Value("张三")
8       private String name;
9
10      @Value("2020-04-06 10:20")
11      private Date ctime;
12
13      @Autowired
14      private Card card;
```

6.3 Spring配置注解(完全使用注解)

6.3.1 Spring配置相关注解

- @Configuration：表示这是一个spring配置文件
- @ComponentScan：开启组件自动扫描（扫描4个注解），**默认basePackages为当前类所在目录(即./*)**
- @Bean：定义一个bean，**bean的id为方法名称**
  注意：@Bean定义的方法，在Spring容器初始化调用的
- @Scope：bean**默认是单例模式**，指定原型模式 @Scope("prototype")
- @Qualifier：微调器，根据id指定一个bean
- @DependsOn：该bean定义要依赖于另一个bean的定义
- @PostConstruct：该对象创建完成之后调用，相当于 init-method
- @PreDestroy：该对象销毁时调用，相当于 destroy-method.
- @Primary：根据类型查找时，优先级最高
- @PropertySource("classpath:/jdbc.properties") :加载属性文件

6.3.2 依赖注入方式

6.3.2.1 方法的参数，会自动依赖注入

注意：根据参数注入，若是根据类型寻找的,当有多个的时候,如果没有@Primary会报错

1. 根据类型

```java
@Bean
@Scope("prototype")
public Date ctime(){
    return new Date();
}
@Bean
@Scope("prototype")
@Primary
public Date ctime2(){
    return new Date();
}
@Bean
public User user2(Date date){
```

2. 根据id

```java
// 根据id注入bean
@Bean
public String bean2(@Qualifier("user2") User user) {
    System.out.println("bean2: " + user);
    return "bean2";
}
```

6.3.2.2 调用另一个标注了@Bean的方法

```java
@Bean // bean的id=user1
public User user1() {xxx;}

User user1 = user1(); // 依赖注入
```

6.3.3 注解形式的配置文件

6.3.3.1 基础配置和依赖注入

```java
/**
 * Spring的配置文件（注解方式）
 */
@Configuration
@ComponentScan("com.kzw")
public class AppConfig {

    @Autowired
    private UserService userService;

    @Bean // bean的id=user1
    @Primary // 根据类型查找时，优先级最高
    public User user1() {
        System.out.println("初始化：user1");
        User u = new User(2L, "李四");
        u.setCtime(new Date());
```

```
17
18          userService.save(u);
19
20          return u;// 注意要返回
21      }
22
23      @Bean
24      @Scope("prototype")
25      public Date ctime() {
26          System.out.println("初始化：ctime");
27          return new Date();
28      }
29
30      // 依赖注入方式1：方法的参数，会自动依赖注入
31      @Bean
32      public User user2(Date date) {
33          System.out.println("初始化：user2");
34          User u = new User(3L, "王五");
35          u.setCtime(date);
36          return u;
37      }
38
39      // 依赖注入方式2：调用另一个标注了@Bean的方法
40      @Bean
41      public User user3() {
42          Date date1 = ctime(); // 依赖注入
43          Date date2 = ctime();
44          System.out.println(date1 == date2); // false，原型模式
45
46          User user1 = user1(); // 依赖注入
47          User user2 = user1();
48          System.out.println(user1 == user2); // true，单例模式
49
50          User u = new User(4L, "小刘");
51          u.setCtime(date1);
52          return u;
53      }
54
55      /**
56       * 使用参数方式注入一个UserService类型的bean
57       * 调用方法方式注入一个User类型对象
58       */
59      @Bean
60      @DependsOn("bean2")
61      public String bean1(UserService userService) {
62          // 此外并不是调用方法，而是获得一个id=user3的bean
63          User user = user3();
64          System.out.println("bean1: " + user);
65          userService.save(user);
66
```

```
67          return "bean1";
68      }
69
70      // 根据id注入bean
71      @Bean
72      public String bean2(@Qualifier("user2") User user) {
73          System.out.println("bean2: " + user);
74          return "bean2";
75      }
76
77  }
78
```

6.3.3.2 其他注解

```
1   @Configuration
2   @PropertySource("classpath:/jdbc.properties") // 加载属性文件
3   public class AppConfig2 {
4
5       // 方式1：注入Environment对象
6       @Autowired
7       private Environment env;
8
9       @Bean
10      public String test1() {
11          String username = env.getProperty("jdbc.username");
12          System.out.println("username: " + username);
13          return "test1";
14      }
15      // 为了可以使用@Value("${xxx}")
16      @Bean
17      public static PropertySourcesPlaceholderConfigurer
18
     placeholderConfigurer() {
19
20          return new PropertySourcesPlaceholderConfigurer();
21      }
22
23      // 方式2：使用@Value("${xxx}"),但是必须加上的配置
24      @Value("${jdbc.password}")
25      private String password;
26
27      @Bean
28      public String test2() {
29          System.out.println("test2, password: " + password);
30          return "test2";
31      }
32
33      @Bean                   // 属性注解
```

```
34      public String test3(@Value("${jdbc.username}") String
    username)
35      {
36          System.out.println("test3, username: " + username);
37          return "test2";
38      }
39
40  }
```

6.3.3.3 注解配置的属性编辑器

6.3.3.3.1 FormattingConversionService+自带(默认)属性编辑器

- 此处bean的id=conversionService,不能随便写
- 该bean在spring内部通过bean的id查找
- 有多个属性编辑器,conversionService.addFormatter 继续添加

```
1   /**
2   * 属性编辑器: 方式1
3   * 此处bean的id=conversionService，不能随便写
4   * 原因: 在spring内部通过bean的id查找
5   */
6   @Bean
7   public FormattingConversionService conversionService() {
8       FormattingConversionService conversionService =
9           new DefaultFormattingConversionService(false);
10
11      conversionService.addFormatter(
12          new DateFormatter("yyyy-MM-dd HH:mm"));
13
14      return conversionService;
15  }
```

6.3.3.3.2 CustomEditorConfigurer+自定义属性编辑器

- 此处bean的id可以随便写,但是方法必须为static
- 该bean在spring内部通过bean的类型查找
- 有多个属性编辑器,map.put(目标类型.class,自定义属性编辑器.class); 继续添加

```
1   // 属性编辑器: 方式2
2   // 必须为static方法
3   @Bean
4   // 必须static
5   public static CustomEditorConfigurer editorConfigurer() {
6       Map<Class<?>, Class<? extends PropertyEditor>> map =
7           new HashMap<>();
8       //map.put(目标类型.class,自定义属性编辑器.class);
9       map.put(Date.class, MyDateEditor.class);
10
11      CustomEditorConfigurer cfg = new CustomEditorConfigurer();
```

```
        cfg.setCustomEditors(map);

        return cfg;
    }
```

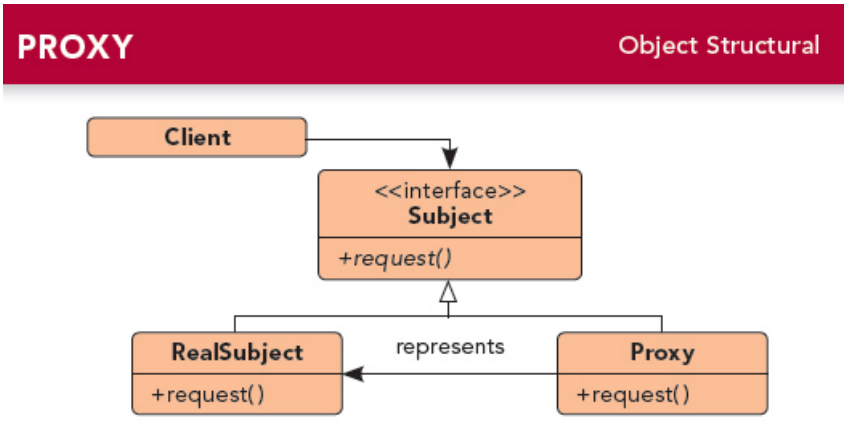
7 Spring aop

7.1 代理简介

7.1.1 目标对象和代理对象

目标对象：被代理的对象
代理对象：经过代理后的对象

7.1.2 静态代理：代理设计模式



代理类是自己定义的

```
package proxy_static;

// 代理对象类
public class AppleProxyComputer implements Computer {
    // 目标对象
    private Computer target;

    public AppleProxyComputer(Computer target) {
        this.target = target;
    }
    @Override
    public int sell(int price, int num) {
        System.out.println("代理商，做宣传...");

        // 调用目标对象的方法
        int total = target.sell(price, num);

```

```
18          System.out.println("代理商，送键盘鼠标...");
19          return total;
20      }
21  }
22
```

7.1.3 动态代理：代理类在JVM中动态创建的

- **JDK动态代理**：使用JDK中的Proxy类，动态创建代理对象（前提：**目标对象类必须实现了业务接口**），优先使用JDK的动态代理
  - 原理：根据目标类对象实现的接口，动态创建一个类也实现这些接口（目标对象类和代理对象类是兄弟关系，都实现相同接口）
- **CGLIB动态代理**：第三方提供，动态创建代理对象。（目标对象类**可以不实现任何接口**）
  - 原理：**动态创建目标对象类的子类**（目标对象类是代理对象类的父类）

7.1.4 jdk动态代理

1. 代理类

```
1  package proxy_dync.jdk;
2
3  import java.lang.reflect.InvocationHandler;
4  import java.lang.reflect.Method;
5  import java.lang.reflect.Proxy;
6
7  public class ProxyComputerBean implements InvocationHandler {
8
9      // 目标对象
10     private Computer target;
11
12     /**
13      * 动态创建一个代理对象 使用JDK动态代理：Proxy
14      */
15     public Computer create(Computer target) {
16         this.target = target;
17         // 根据目标对象动态创建代理对象
18         return (Computer)
   Proxy.newProxyInstance(target.getClass().getClassLoader(),
   target.getClass().getInterfaces(),
19                 this);
20     }
21     /**
22      * 通过代理对象调用方法时，会被拦截，执行下面的方法
23      */
24     @Override
25     public Object invoke(Object proxy, Method method, Object[]
   args) throws Throwable {
```

```
        Object ret = null;

        // 环绕通知
        try {
            System.out.println("调用之前：代理商做宣传...");  //前置通知
            // 调用目标对象的方法
            ret = method.invoke(target, args);

            System.out.println("调用之后：代理商送键盘鼠标...");//后置通知
        } catch (Exception e) {
            System.out.println("异常时调用"); //异常通知
        } finally {
            System.out.println("最终调用：送货上门"); //最终通知
        }
        return ret;
    }

}

```

2. 测试类

```
package proxy_dync.jdk;

public class T2 {

    public static void main(String[] args) {

        Computer computer = new LenovoComputer();
        int total = computer.sell(8000, 2);
        System.out.println(total);

        System.out.println("====================");

        // 代理目标对象
        Computer computer2 = new ProxyComputerBean().create(computer);
        System.out.println(computer2.getClass());

        // 通过代理对象，调用方法
        int total2 = computer2.sell(8000, 2);

        System.out.println(total2);
    }
}

```

7.1.5 cglib

1. 代理类

```java
package proxy_dync.cglib;

import java.lang.reflect.Method;

import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.InvocationHandler;

public class ProxyComputerBean implements InvocationHandler {

    // 目标对象
    private Object target;

    /**
     * 动态创建一个代理对象 使用第三方组件CGLIB动态代理
     */
    @SuppressWarnings("unchecked")
    public <T> T create(Object target) {
        this.target = target;

        // 根据目标对象动态创建代理对象
        Enhancer enhancer = new Enhancer();

enhancer.setClassLoader(target.getClass().getClassLoader());
        enhancer.setSuperclass(target.getClass()); // 生成目标对象类的子类
        enhancer.setCallback(this);

        return (T) enhancer.create();
    }

    /**
     * 通过代理对象调用方法时，会被拦截，执行下面的方法
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

        Object ret = null;

        // 环绕通知
        try {
            System.out.println("调用之前：代理商做宣传..."); // 前置通知

            // 调用目标对象的方法
            ret = method.invoke(target, args);

```

```
            System.out.println("调用之后：代理商送键盘鼠标...");
// 后置通知

        } catch (Exception e) {
            System.out.println("异常时调用"); // 异常通知
        } finally {
            System.out.println("最终调用：送货上门"); // 最终通知
        }

        return ret;
    }

}
```

2. 测试类

```
package proxy_dync.cglib;

public class T3 {

    // 使用CGLIB动态代理
    public static void main(String[] args) {

        HuaweiComputer computer = new HuaweiComputer();
        int total = computer.sell(8000, 2);
        System.out.println(total);

        System.out.println("====================");

        // 代理目标对象
        HuaweiComputer computer2 = new
ProxyComputerBean().create(computer);
        System.out.println(computer2.getClass());

        // 通过代理对象，调用方法
        int total2 = computer2.sell(8000, 2);

        System.out.println(total2);
    }
}
```

7.2 aop

7.2.1 概念

7.2.1.1 aop

Spring AOP（面向切面编程）的核心概念：

- 切面：一个关注点的模块化，可能横切多个对象
- 切入点：匹配连接点的断言，通过表达式匹配要**拦截哪些方法**
- 连接点：拦截到的方法，在spring aop中表示方法的执行
- 通知：在连接点特定位置上执行的方法，通知根据位置不同，包含：前置通知、后置通知、异常通知、最终通知、环绕通知
- 目标对象：被代理的对象，被切面横切的对象
- 代理对象：经过aop代理的对象，由aop框架动态生成的对象

注意：如果一个类中存在一个方法被切入点表达式匹配，那么spring中该对象是一个代理对象

7.2.1.2 通知类型

1、前置通知：在连接点方法执行之前调用，@Before
2、后置通知：在连接点方法执行正常返回之后调用，@AfterReturning
3、异常通知：在连接点方法执行非正常返回之后调用，@AfterThrowing
4、最终通知：在连接点方法执行（正常或非正常）退出时调用，@After
5、环绕通知：自定义通知，@Around

```
前置通知 --> 目标对象方法调用 --> 最终通知   --> 后置通知
                              --> 异常通知    --> 后置通知
```

7.2.1.3 切入点表达式

1、execution：指定到方法
2、within：指定到类（类中所有方法）
3、@annotation：标注了某注解的所有方法
4、多个切入点组合：&&，||，！

```
例1：execution(public * com.kzw.service..(..))
    匹配com.kzw.service包下所有类的public方法
例2：execution(public * com.kzw.service...(..))
    匹配com.kzw.service包以及所有子包下所有类的public方法
例3：with(com.kzw.service.)
    匹配com.kzw.service包下所有类的所有方法
例4：with(com.kzw.service..)
    匹配com.kzw.service包以及所有子包下所有类的所有方法
例5：@annotation(com.kzw.Log)
    匹配标了@Log注解的所有方法
```

```java
@Pointcut("@annotation(com.kzw.anno.Log)")
    public void pc1() {
}
```

> 例6: @Pointcut("pc1() && pc2() || pc3()")

7.2.2 example

1. LogAspect

```
// 注解定义
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD })
public @interface Log {

    String value() default "admin";
}

// 切面
@Aspect
@Component
@Order(10)  //值越小，优先级越高
public class LogAspect {

    // 可以在切面中注入spring中的对象
    // 在web环境下，可以注入HttpServletRequest对象
    @Autowired
    private UserService userService;

    // 定义切入点
    @Pointcut("@annotation(com.kzw.anno.Log)")
    public void pc1() {
    }

    /**
     * 哪个用户在什么时间访问了哪个方法，参数是什么，返回值是什么，方法执行了多长时间
     * 如何获得用户：1)通过session获得， 2)通过线程变量获得
     */
    @Around("pc1() && @annotation(log)")
    public Object doLog(ProceedingJoinPoint pjp, Log log) throws Throwable {

        // 获得代理的原目标对象
        System.out.println(pjp.getTarget());
        // userService.save(); //userService是一个代理对象
        userService.findById(1);

        String uname = log.value();
        String time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());
        String md = pjp.getSignature().toLongString();
        String args = Arrays.toString(pjp.getArgs());
```

```
41          long start = System.currentTimeMillis();
42
43          Object ret = null;
44          try {
45              // 调用目标对象的方法
46              ret = pjp.proceed();
47
48          } catch (Exception e) {
49              ret = String.format("调用异常，异常信息：%s（%s）",
    e.getClass().getSimpleName(), e.getMessage());
50              throw new Throwable(e);
51          } finally {
52              // 保存日志
53              long end = System.currentTimeMillis();
54              String msg = String.format("[%s]：用户(%s)调用方法
    [%s]，参数：%s，返回值：%s，执行时长：%d",
55                      time, uname, md, args, ret+"", end-start);
56              System.out.println(msg);
57          }
58
59          return ret;
60      }
61
62      // 前置通知
63      @Before("pc1()")
64      public void test1() {
65          System.out.println("Log，前置通知。。。");
66      }
67
68  }
69
```

2. myaspect

```
1   @Aspect
2   @Component
3   @Order(2)
4   public class MyAspect {
5
6       // 定义切入点
7       @Pointcut("execution(public * *(..))")
8       public void pc1() {
9       }
10
11      @Pointcut("execution(* com.kzw.service..*.*(..))")
12      public void pc2() {
13      }
14
15      @Pointcut("pc1() && pc2()")
16      public void pc3() {
```

```
17      }
18
19      // 前置通知
20      @Before("pc3()")
21      public void test1() {
22          System.out.println("My 前置通知。。。");
23      }
24
25      // 后置通知：可以获得方法返回值
26      @AfterReturning(pointcut = "pc3()", returning = "ret")
27      public void test2(JoinPoint jp, Object ret) {
28          System.out.println("后置通知，方法返回值：" + ret);
29      }
30
31      // 异常通知：可以获得方法的异常信息
32      @AfterThrowing(pointcut = "pc3()", throwing = "ex")
33      public void test3(Throwable ex) {
34          System.out.println("后置通知，异常信息：" +
    ex.getClass());
35      }
36
37      // 最终通知：正常或非正常退出时调用
38      @After("pc3()")
39      public void test4() {
40          System.out.println("最终通知...");
41      }
42
43      /**
44       * 环绕通知：自定义通知
45       * 1、方法需要返回值
46       * 2、方法需要参数：ProceedingJoinPoint
47       * 3、方法需要抛出异常
48       */
49      @Around("pc3()")
50      public Object test5(ProceedingJoinPoint pjp) throws
    Throwable {
51
52          Object ret = null;
53          try {
54              System.out.println("相当于前置通知");
55
56              // 调用目标对象的方法
57              ret = pjp.proceed();
58
59              System.out.println("相当于后置通知");
60          } catch (Exception e) {
61              System.out.println("相当于异常通知");
62              throw new Throwable(e); //需要将异常继续向外抛，避免异
    常被吞掉
63          } finally {
```

```
64                 System.out.println("相当于最终通知");
65             }
66
67             return ret;
68         }
69
70     }
```

8 Spring link DB

8.1 jdbc

8.1.1 Spring事务管理器:@Transactional

1. @Transactional注解：（该类对象会被代理）基于AOP实现
   - 可以用在类上，表示该类所有的public方法都应用该注解
   - 可以用在方法上，可以覆盖类上的@Transactional注解配置
   - @EnableTransactionManagement开启

```
1   @Configuration
2   @ComponentScan
3   @EnableTransactionManagement
4   //开启@Transactional注解，需要事务管理器
5   @PropertySource("classpath:/jdbc.properties")
6   public class AppConfig {
```

2. 属性readOnly：只读锁，默认值false
   readOnly="true"：优化性能，但该方法中不能执行增删改的操作
3. 并不是所有的异常都会回滚，默认只有抛出运行时异常才事务会**回滚**，抛出受检查异常(Exception及其子类)事务会**提交**
   - rollbackFor={指定回滚异常类}
   - noRollbackFor={指定提交异常类}

8.1.2 使用spring jdbc

1. 配置文件

```
1   // jdbc配置文件
2
3   jdbc.driver=com.mysql.cj.jdbc.Driver
4   jdbc.url=jdbc:mysql://localhost:3306/test?characterEncoding=UTF-
    8&serverTimezone=UTC
5   jdbc.username=root
```

```
jdbc.password=123456


@Configuration
@ComponentScan
@EnableTransactionManagement
//开启@Transactional注解，需要事务管理器
@PropertySource("classpath:/jdbc.properties")
public class AppConfig {
    @Autowired
    private Environment env;

    // 数据源
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new
            DriverManagerDataSource();
    ds.setDriverClassName(env.getProperty("jdbc.driver"));
        ds.setUrl(env.getProperty("jdbc.url"));
        ds.setUsername(env.getProperty("jdbc.username"));
        ds.setPassword(env.getProperty("jdbc.password"));
        return ds;
    }
     //事务管理器
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new
           DataSourceTransactionManager(dataSource());
    }

    // JdbcTemplate：线程安全的，可以使用单例模式
    @Bean
    public JdbcTemplate jdbcTemplate() {
        // 注入依赖对象
        return new JdbcTemplate(dataSource());
    }
}
```

2. userdao

```
/**
 * 基于Spring JdbcTemplate实现
 */
@Repository
@Transactional
public class UserDao1 {

    @Autowired
    private JdbcTemplate jdbcTemplate;

```

```
11      public void save(User user) throws Exception {
12          String sql = "insert into tb_user values(null,?,?)";
13          // 默认jdbc是自动提交事务的
14          jdbcTemplate.update(sql, user.getName(), user.getAge());
15
16          // 抛出运行时异常
17          throw new RuntimeException("模拟运行时异常");
18      }
19
20
21      public void save2(User user) throws Exception {
22          String sql = "insert into tb_user values(null,?,?)";
23          // 默认jdbc是自动提交事务的
24          jdbcTemplate.update(sql, user.getName(), user.getAge());
25
26          // 抛出受检查异常
27          throw new Exception("模拟受检查异常");
28      }
29
30      // 让运行时异常提交（所有异常事务都提交）
31      @Transactional(noRollbackFor = RuntimeException.class)
32      public void save3(User user) throws Exception {
33          String sql = "insert into tb_user values(null,?,?)";
34          // 默认jdbc是自动提交事务的
35          jdbcTemplate.update(sql, user.getName(), user.getAge());
36
37          // 抛出运行时异常
38          throw new RuntimeException("模拟运行时异常");
39      }
40
41
42      // 让受检查异常回滚（所有异常事务都回滚）
43      @Transactional(rollbackFor = Exception.class)
44      public void save4(User user) throws Exception {
45          String sql = "insert into tb_user values(null,?,?)";
46          // 默认jdbc是自动提交事务的
47          jdbcTemplate.update(sql, user.getName(), user.getAge());
48
49          // 抛出受检查异常
50          throw new Exception("模拟受检查异常");
51      }
52
53
54
55      @Transactional(readOnly = true)
56      public List<User> findAll() {
57          String sql = "select * from tb_user";
58          return jdbcTemplate.queryForList(sql, User.class);
59      }
60  }
```

8.1.3 使用dbutil

1. dbutils默认@Transactional无效,需要提供一个**具有事务感知**的**数据源代理**

```
@Configuration
@ComponentScan
@EnableTransactionManagement  //开启@Transactional注解，需要事务管理器
@PropertySource("classpath:/jdbc.properties")
public class AppConfig {
@Autowired
private Environment env;


    // 数据源
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new
            DriverManagerDataSource();
        ds.setDriverClassName(env.getProperty("jdbc.driver"));
        ds.setUrl(env.getProperty("jdbc.url"));
        ds.setUsername(env.getProperty("jdbc.username"));
        ds.setPassword(env.getProperty("jdbc.password"));
        return ds;
    }
    //事务管理器,dbutil,和spring自带jdbc都要用事务管理器
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(dataSource());
    }

    // QueryRunner，必须提供一个事务感知的数据源代理
    // 否则，@Transactional注解无效
    // 事务感知的数据源代理


    @Bean
    public TransactionAwareDataSourceProxy dataSourceProxy() {
        return new
TransactionAwareDataSourceProxy(dataSource());
    }

    @Bean
    public QueryRunner queryRunner() {
        return new QueryRunner(dataSourceProxy());// 这里用代理
    }
}
```

2. dbutils 执行SQL 时，会抛出SQLException（这是受检查异常），要在方法上加上 `throws SQLException` ,类上加上 `@Transactional(rollbackFor =`

SQLException.class)

```java
@Repository
@Transactional(rollbackFor = SQLException.class)
public class UserDao2 {
    @Autowired
    private QueryRunner runner;
    // 会回滚
    public void save2(User user)  throws SQLException{
        String sql = "insert into tb_user"+
                                        "values(null,?,?)";
        // 默认jdbc是自动提交事务的
        runner.update(sql, user.getName(), user.getAge());

        // 抛出受检查异常,结合@Transactional导致回滚
        throw new SQLException("模拟受检查sql异常");
    }
    // 不会回滚,dbutils正常执行发出的SQLException不会导致回滚
    public void save3(User user)  throws SQLException{
        String sql = "insert into tb_user"+
                                        "values(null,?,?)";
        // 默认jdbc是自动提交事务的
        runner.update(sql, user.getName(), user.getAge());
    }
}
```

8.2 Sring JPA

8.2.1 对应关系

1. Spring集成JPA（hibernate）

EntityManagerFactory == hibernate的SessionFactory

EntityManager == hibernate的Session

2. SpringBoot：spring通过注解集成JPA，而不是直接集成hibernate

8.2.2 层次

- entity：实体类
- dao：底层的数据库交互层

- service：封装dao层,服务层实现更复杂的用户逻辑

Service层是否需要?

> - DAO层只针对单个表的增删改查，不能处理复杂的业务，事务一般不加在DAO方法上

- Service：对应真实的业务逻辑，事务应该加在Service方法上Service类一般会引用多个DAO类，在一个方法中对多个表进行操作

8.2.3 code

1. AppConfig

```
@Configuration
@ComponentScan
@EnableTransactionManagement    //事务注解，需要事务管理器
@EnableJpaRepositories  //开启JPA Repository功能
@PropertySource("classpath:/jdbc.properties")
public class AppConfig {

    @Autowired
    private Environment env;

    // 数据源
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(env.getProperty("jdbc.driver"));
        ds.setUrl(env.getProperty("jdbc.url"));
        ds.setUsername(env.getProperty("jdbc.username"));
        ds.setPassword(env.getProperty("jdbc.password"));
        return ds;
    }

    // JPA中的EntityManagerFactory，相当于hibernate中的SessionFactory
    @Bean
    public EntityManagerFactory entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emb = new LocalContainerEntityManagerFactoryBean();

        // 数据源、映射文件路径、jpa属性
        emb.setDataSource(dataSource());
        emb.setPackagesToScan("com.kzw.entity");

        // hibernate属性：方言，show_sql，hbm2ddl.auto
        Properties prop = new Properties();
```

```
        prop.setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQL5InnoDBDialect");
        prop.setProperty("hibernate.show_sql", "true");
        prop.setProperty("hibernate.hbm2ddl.auto", "update");
        emb.setJpaProperties(prop);

        // 指定JPA具体的提供商
        emb.setJpaVendorAdapter(new
HibernateJpaVendorAdapter());

        // 进行初始化
        emb.afterPropertiesSet();
        return emb.getObject();
    }

    // JPA的事务管理器
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new
JpaTransactionManager(entityManagerFactory());
    }
}
```

2. user

```
@Entity
@Table(name = "tb_user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(length = 50)
    private String name;

    private int age;
```

3. dao

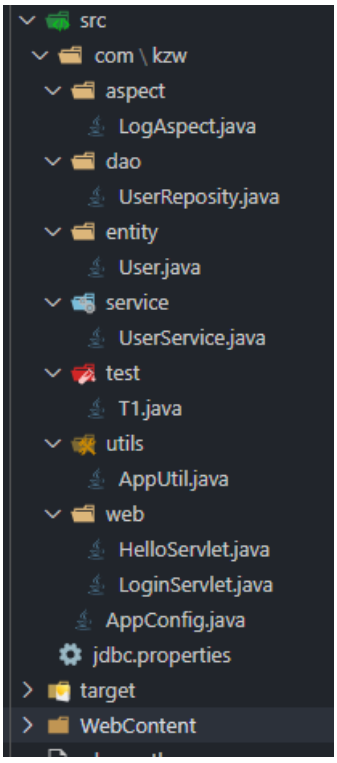
```
/**
 * 针对实体User进行增删改查（分页），不需要提供接口实现类
 * 该接口的实现类，通过Spring AOP生成动态代理对象
 * spring-data-jpa组件
 * */
public interface UserReposity extends JpaRepository<User, Long>
{

    // 查询方式1：根据方法名称自动构造查询条件（语义理解）
    // 只适用于简单场景
    public List<User> findByNameLike(String name);
```

```
11
12      public List<User> findByNameAndAge(String name, int age);
13
14      // 查询方式2：使用@Query注解指定HQL或原生SQL
15      // 位置占位符，从1开始
16      @Query("from User where name like ?1 order by age desc")
17      public List<User> find1(String name);
18
19      @Query("from User where age>=?2 and name like ?1")
20      public List<User> find2(String name, int age);
21
22      // 基于原生SQL
23      @Query(value="select * from tb_user where age>=?2 and name
    like ?1", nativeQuery=true)
24      public List<User> find3(String name, int age);
25
26
27      // DML风格的查询：增删改，必须使用@Modifying
28      @Modifying
29      @Query("update User set age=age+?2 where age>=?1")
30      public void incAge(int age, int inc);
31
32      // 分页
33      @Query("from User where name like ?1")
34      public List<User> find1(String name, Pageable pageable);
35  }
36
```

4. service

```
1   @Service
2   @Transactional // 事务应该放在service层中
3   public class UserService {
4
5       @Autowired
6       private UserReposity userDao;
7
8       public void incAge(int age, int inc) {
9           userDao.incAge(age, inc);
10      }
11
12      @Transactional(readOnly = true)
13      public List<User> find1(String name) {
14          return userDao.find1(name);
15      }
16
17      public User findById(Long id) {
18          return userDao.getOne(id);
19      }
20  }
```

9 spring web



9.1 步骤

在Web项目中集成Spring和JPA步骤：

1. 构建web项目
2. 集成spring框架：构建一个基于web的ApplicationContext
3. 在spring中集成JPA（原来方式集成即可）
4. 如何在web项目中(如servlet)获得spring中的bean

9.2 xml配置-集成spring

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">

    <!-- 集成Spring框架 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.kzw.AppConfig</param-value>
```

```
    </context-param>
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- 字符集过滤器 -->
    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- OpenEntityManagerInViewFilter -->
    <!-- 保证页面渲染完毕之后,再关闭session -->
    <filter>
        <filter-name>OpenEntityManagerInViewFilter</filter-name>
        <filter-class>org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>OpenEntityManagerInViewFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- 将request绑定到当前线程中,线程中所有对象都能访问,
     可以通过依赖注入方式获取 -->
    <filter>
        <filter-name>RequestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
    </filter>
```

```
    <filter-mapping>
        <filter-name>RequestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

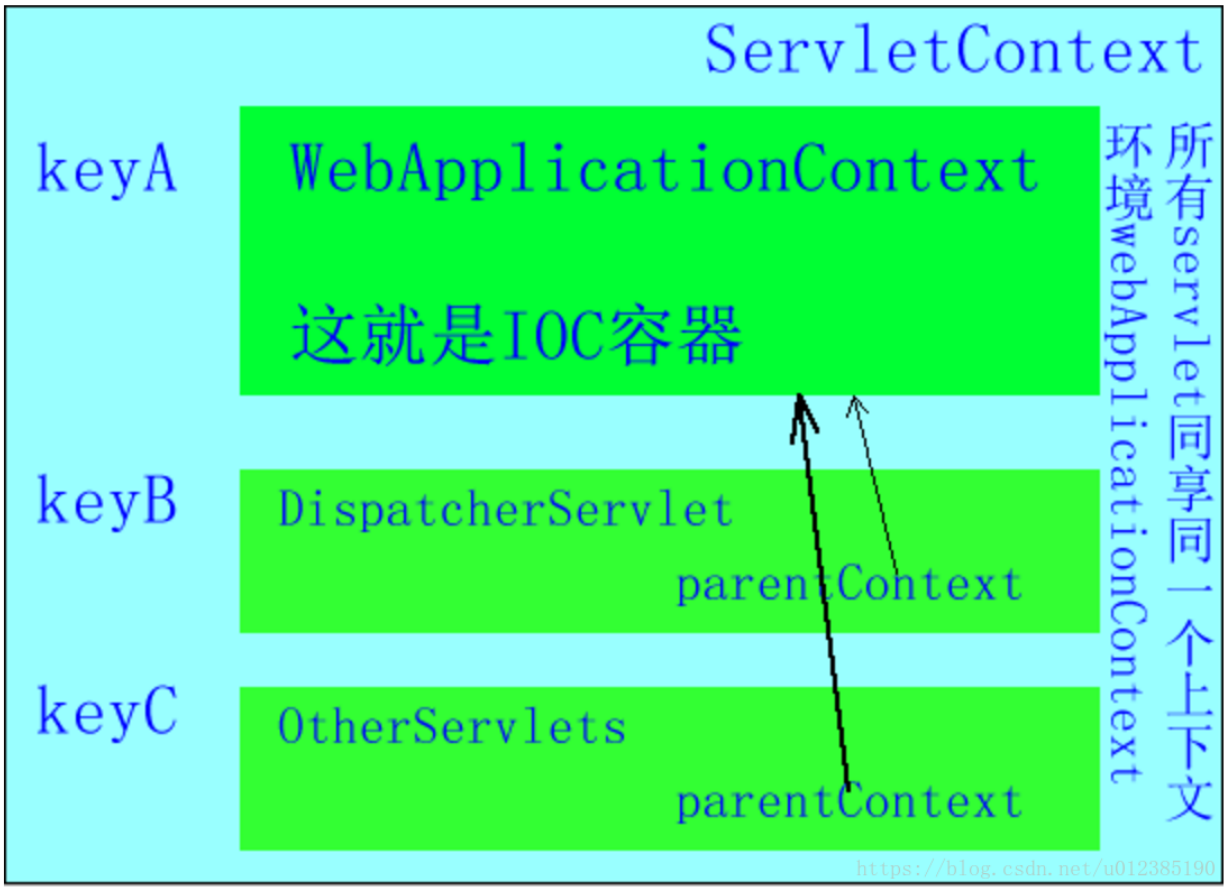
9.3 sundry

Spring WebApplicationContext

1. 可以进行依赖注入吗?

```

//该类交给spring管理之后，才能使用 @Resource, @Autowired 进行依赖注入
//若交给spring管理,servlet便无法使用,servlet放入servlet容器才有效
@WebServlet("/hello")
@SuppressWarnings("serial")
public class HelloServlet extends HttpServlet {

    // 不能直接使用注解进行依赖注入
    private UserService userService =
        AppUtil.getBean(UserService.class);
```

- tomcat启动时，下面组件启动顺序：servlet < fitler < listener,使用listener先初始化ServletContext(作为公共环境容器存放公共信息),然后创建WebApplicationContext(Web应用上下文)并以键值对形式存放与ServletContext中。



- WebApplicationContext(spring容器)实现类:
  - XmlWebApplicationContext
  - AnnotationConfigWebApplicationContext
- WebApplicationContext包含servlet的上下文环境,但是他不能包括servlet对象(把servlet交给spring管理)

2. 如何在servlet中获得spring中的bean:

一定不要使用 new AnnotationConfigApplicationContext(AppConfig.class);因为xml已经配置了一个同类型的容器,再创建非常耗费资源

解决方法如下:

- 应该使用工具类: WebApplicationContextUtils.getWebApplicationContext(

```
1                   req.getServletContext());
```

必须提供ServletContext对象作为参数。

```java
@Override
protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException {
    // 获得WebApplicationContext实例对象
    WebApplicationContext atx =
WebApplicationContextUtils.getWebApplicationContext(req.getServl
etContext());
    UserService userService2 = atx.getBean(UserService.class);
    // 方式1输出
    System.out.println(userService2);
    // 方式2输出
    System.out.println(userService);

    User user = new User("张三" ,20);
    userService.save(user);
}
```

- 工具类：实现接口 ApplicationContextAware

```java
@Component
public class AppUtil implements ApplicationContextAware {
    private static ApplicationContext context;
    @Override
    public void setApplicationContext(ApplicationContext atx)
throws BeansException {
        context = atx;
    }
    /**
     * 获得Bean
     */
    public static <T> T getBean(String id, Class<T> cls) {
        return context.getBean(id, cls);
    }
    public static <T> T getBean(Class<T> cls) {
        return context.getBean(cls);
    }
}
```

10 Spring MVC

10.1 xml方式

10.1.1 introduction

1. Spring MVC中，@Controller注解有特殊用途，此时不能替换成@Component，@Service，@Repository

spring配置文件中：不能扫描@Controller注解

springmvc配置文件中：只扫描@Controller注解

2. spring mvc中，@Controller有什么用?

springmvc扫描到@Controller注解，会继续扫描该类中的其它注解，如@RequestMapping

3. spring mvc拦截所有请求：

- 可以直接访问JSP页面（相当于不拦截）
- 其它请求（包含静态文件：css,js,jpg）默认被拦截，被当作普通请求

10.1.2 spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

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-4.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.1.xsd">

    <!-- 配置与spring相关的组件 -->
    <!-- 开启组件自动扫描: @Component, @Service, @Repository，不能扫描@Controller -->
    <context:component-scan base-package="com.kzw">
        <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>
</beans>
```

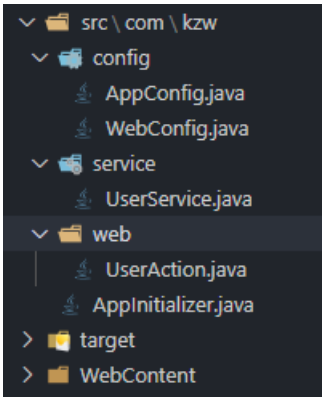
10.1.3 spring-mvc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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-4.1.xsd
        http://www.springframework.org/schema/context
```

```
9           http://www.springframework.org/schema/context/spring-
    context-4.1.xsd
10          http://www.springframework.org/schema/mvc
11          http://www.springframework.org/schema/mvc/spring-mvc-
    4.1.xsd">
12
13      <!-- 配置与spring mvc相关的组件 -->
14      <!-- 开启组件自动扫描: 只扫描@Controller -->
15      <context:component-scan base-package="com.kzw">
16          <context:include-filter type="annotation"
    expression="org.springframework.stereotype.Controller"/>
17      </context:component-scan>
18      <!-- 启用注解 -->
19      <mvc:annotation-driven />
20
21
22      <!-- 配置视图解析器: 将视图名称 ==> 真实页面 -->
23      <mvc:view-resolvers>
24          <mvc:jsp prefix="/pages/" suffix=".jsp"/>
25      </mvc:view-resolvers>
26
27      <!-- 配置静态资源访问 -->
28      <!--  请求映射: /img/aa/b.jpg ==> /static/img/aa/b.jpg
29                    /imgs/a.jpg ==> /static/img/a.jpg
30
31      <mvc:resources location="/static/" mapping="/**"/>
32      -->
33      <mvc:resources location="/static/img/" mapping="/img/**"/>
34      <mvc:resources location="/static/img/" mapping="/imgs/**"/>
35      <mvc:resources location="/static/css/" mapping="/css/**"/>
36
37      <!-- 服务器上的本地目录: file:///path 或者 file:path -->
38      <mvc:resources location="file:///C:/var/"
    mapping="/upload/**"/>
39
40      <!-- 重定向 -->
41      <mvc:redirect-view-controller redirect-url="/user/t1"
    path="/"/>
42
43  </beans>
```

10.2 注解方式

10.2.1 step



基于注解的spring mvc环境搭建：

1、去掉web.xml，使用一个web启动器代替

2、基于注解方式，编写spring配置类

3、基于注解方式，编写spring mvc配置类

4、在spring配置类中集成JPA，在spring mvc配置类中编写与web相关的内容

10.2.2 code

1. AppInitializer

```
/**
 * web启动器，取代web.xml，web程序的入口
 */
public class AppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    /**
     * 集成spring框架，指定spring配置类
     */
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { AppConfig.class };
    }

    /**
     * 集成springmvc框架，指定springmvc配置类
     */
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { WebConfig.class };
    }

    /**
     * DispatcherServlet要拦截的请求:
     * /代表: 所有url都被DisptacherServlet处理
     */
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
```

```
        //所有url都被DisptacherServlet处理
    }

    /**
     * 注册其它的servlet, filter, listener
     */
    @Override
    public void onStartup(ServletContext sc) throws
ServletException {
        super.onStartup(sc);

        // 1) 注册监听器：RequestContextListener, 将请求对象绑定到
当前线程
        // 如果需要给监听器设置初始化参数，则使用
sc.setInitParameter()
        sc.addListener(RequestContextListener.class);

        // 2) 注册过滤器：CharacterEncodingFilter
        // 过滤器设置初始化参数，并设置拦截url，即<url-pattern>
        Dynamic filter = sc.addFilter("EncodingFilter",
CharacterEncodingFilter.class);
        filter.setInitParameter("encoding", "UTF-8");
        filter.addMappingForUrlPatterns(null, false, "/*");
    }

    // 基于servlet3.x的文件上传配置
    @Override
    protected void
customizeRegistration(javax.servlet.ServletRegistration.Dynamic
registration) {
        // 指定文件上传的临时目录
        String loc = System.getProperty("java.io.tmpdir");
        MultipartConfigElement config = new
MultipartConfigElement(loc);
        registration.setMultipartConfig(config);
    }

}
```

2. AppConfig

```
/**
 * Spring的配置文件
 * */
@Configuration
@ComponentScan(basePackages = "com.kzw", excludeFilters =
@Filter(type = FilterType.ANNOTATION, value = Controller.class))
public class AppConfig {

}
```

3. WebConfig

- spring与springmvc 配置文件中的Bean会合并在一起,均交给spring容器管理

```
/**
 * Spring MVC的配置文件
 */
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.kzw", includeFilters =
@Filter(type = FilterType.ANNOTATION, value = Controller.class))
public class WebConfig implements WebMvcConfigurer {

    // 静态资源目录
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry
registry) {

registry.addResourceHandler("/**").addResourceLocations("/static
/");

registry.addResourceHandler("/upload/**").addResourceLocations("
file:///C:/var/");
    }

    // 视图到请求url的映射
    @Override
    public void addViewControllers(ViewControllerRegistry
registry) {
        // 重定向映射
        registry.addRedirectViewController("/", "/user/t1");
        // 普通映射：将url映射到某视图名称

registry.addViewController("/user/t3").setViewName("test3");
    }

    // 视图解析器
    @Override
    public void configureViewResolvers(ViewResolverRegistry
registry) {
        registry.jsp("/pages/", ".jsp");
    }

    // 文件上传解析器
    @Bean
    public MultipartResolver multipartResolver() {
        // 使用servlet 3.x的文件上传
        return new StandardServletMultipartResolver();
    }

```

```
}
```

4. userservice

```
@Service
public class UserService {

    @Autowired
    private HttpServletRequest request;// request可以依赖户转入

    public void save() {
        System.out.println("当前请求: " +
request.getRequestURI());
        System.out.println("保存用户信息");
    }
}
```

5. useraction

- springmvc扫描到@Controller会
  - 交给spring管理
  - 继续扫描其内的注解

```
/**
 * 控制器类，即MVC中的C
 * @Controller 注解在此处有特殊用途
 * */
@Controller
public class UserAction {

    @Autowired
    private UserService userService;//

    /**
     * 请求映射：url ==> 方法t1
     * 返回视图名称：通过视图解析器查找真实页面
     * */
    @RequestMapping("/user/t1")
    public String t1() {
        System.out.println("hello world");
        userService.save();
        // 返回：  视图名称 ==> 真实页面: /pages/ + 视图名称 + .jsp
        return "test1";
    }

    @RequestMapping("/user/t2")
    public String t2(String name, Integer age, Model model) {
        System.out.println(String.format("name=%s, age=%d",
name, age));
        model.addAttribute("name", name);
```

```
27
28          return "user/test2";
29      }
30
31      @RequestMapping("/user/t4")
32      public String t4() {
33          return "test4";
34      }
35
36
37      /**
38       * 文件上传
39       * */
40      @RequestMapping("/file/upload")
41      public String upload(String uname, MultipartFile file1)
    throws Exception {
42
43          System.out.println(uname);
44
45          // 文件上传
46          if(!file1.isEmpty()) {
47              String fname = file1.getOriginalFilename();
48              System.out.println(fname); //原文件的名称
49              System.out.println(file1.getName()); //表单的name
50              System.out.println(file1.getSize());
51
52              // 保存文件到服务器
53              File dest = new File("C:/var", fname);
54              file1.transferTo(dest);
55          }
56
57          return "test1";
58      }
59  }
```

10.3 SpringMVC常用的注解:

1. @Controller

- 兼容spring@Controller 的功能

2. @RestController:
   等价于: @Controller + @ResponseBody

```
1  // 相当于每个方法上，都会增加@ResponseBody
2  @RestController
3  @RequestMapping("/h2")
4  public class HelloAction2 {
5
```

```
    // 进行页面跳转
    @RequestMapping("t1")
    public ModelAndView t1(@RequestBody String content) {
        ModelAndView mv = new ModelAndView("t6");
        mv.addObject("name", "张三");
        mv.addObject("age", 20);
        return mv;
    }

    // 没有页面跳转，输出t2
    @RequestMapping("t2")
    public String t2(Model model) {
        model.addAttribute("name", "张三");
        model.addAttribute("age", 20);
        return "t2";
    }

    @RequestMapping("t3")
    public Msg t3(String name, Integer age) {
        System.out.println(name + ", " + age);
        return new Msg();
    }
}
```

3. @ResponseBody:

   声明将方法返回的对象，转成JSON字符串

```
/**
* @ResponseBody:  使用默认的Jackson组件，将返回对象输出JSON字符串
需要了解Jackson相关的注解
*/
@ResponseBody
@RequestMapping("t8")
public User t8() {
    User u = new User();
    u.setName("张三");
    u.setAge(20);
    return u;
}

// Jackson是spring mvc默认的JSON组件
@ResponseBody
@RequestMapping("t9")
public Msg t9() {
    User u = new User();
    u.setName("张三");
    u.setAge(20);
    return new Msg(true, u);
}

```

```java
// 复杂场景下（存在对象引用），可以自定义JSON输出
@RequestMapping("t10")
public void t10(HttpServletResponse resp) {
    User u = new User();
    u.setName("张三");
    u.setAge(20);

    String json = Jackson.me().toJson(u);
    ResponseUtils.renderJson(resp, json);// 要引入老师的工具类
}
```

4. @RequestMapping:

- 定义在类上（可选）
- 定义在方法上（必选）

```java
@Controller
@RequestMapping("/hello")
public class HelloAction {
    // 不能直接访问 WEB-INF目录下的JSP页面
    @RequestMapping("/test1")
    public String t1() {
        return "test1";
    }
    // 请求url：/hello/test2
    // @RequestMapping(value = "/test2", method = RequestMethod.GET)
    @GetMapping("/test2")
    public String t2() {
        return "test1";
    }
    // 请求url：/hello/test2
    // @RequestMapping(value = "test2", method = RequestMethod.POST)
    @PostMapping("test2")
    public String t3() {
        return "test1";
    }
}
```

4. @RequestBody

- 主要用于请求内容类型为：application/json（参数：{id:1, name:"abc"}）
- 通常默认的请求内容类型(表单提交)：application/x-www-form-urlencoded（参数：id=1&name=abc）

```java
// 前端请求类型：application/json，传递json到后台
@RequestMapping("t4")
public Msg t4(@RequestBody String content) {
    System.out.println(content);
    Map<String, Object> map = Jackson.me().fromJson(content,
Map.class);
    System.out.println(map);
    return new Msg();
}
```

5. @Valid

- 表单对象进行验证（JSR303标准，hibernate-validator组件）

```java
public class User {

    @Pattern(regexp = "[a-zA-Z]{5,}", message = "至少5个英文字
母")
    private String name;

    @Range(min = 18, max = 60)
    private int age;

    @Future
    private Date ctime;
}
```

6. @InitBinder：参考**请求参数:command/form对象**
7. @RequestParam：
   提供默认值，或者请求中参数名称和方法的参数名称不一致时
8. @PathVariable：
   路径变量，可以美化url，/user/1
9. @RequestHeader：
   从请求header中获得相应的值
10. @CookieValue：
    从cookie中获得相应的值
11. @SessionAttributes
12. @ModelAttribute

10.4 控制器

10.4.1 常用参数

参数可以灵活使用，需要的时候加上即可，**参数的顺序（除了Errors或BindingResult必须紧跟在JavaBean对象之后）**是可以**随意**的。

1. HttpServletRequest、HttpServletResponse, HttpSession

```java
// 1. HttpServletRequest、HttpServletResponse, HttpSession
@RequestMapping("t1")
public String t1(HttpServletRequest req) {
    System.out.println(req.getParameter("id"));
    System.out.println(req.getRequestURI());

    return "test1";
}
@RequestMapping("t1_2")
public String t1_2(HttpServletRequest req, HttpServletResponse resp) {
    System.out.println(resp.getStatus());

    return "test1";
}
```

2. @RequestParam **或** 直接类型变量

- 使用@RequestParam，则默认请求默认有（required=true），url中该参数必须存在
- @RequestParam(defaultValue = "1")设置默认值
- @RequestParam(value = "xx")：xx要与url中参数匹配
- 如果**不加**@RequestParam，**函数参数名称要和url中的参数匹配**
- 获取请求参数的值，需要注意如果变量是基本类型(int, long)则必须保存该参数有正确的值

```java
// 2. @RequestParam或直接类型变量
// 获取请求参数的值，需要注意如果变量是基本类型(int, long)则必须保存该参数有正确的值
@RequestMapping("t2")
public String t2(Integer id, String name, int age) {
    // url参数要和函数的参数对应
    System.out.println(id + ", " + name + ", " + age);
    return "test1";
}

@RequestMapping("t2_2")
public String t2_2(Integer id, @RequestParam String name, @RequestParam(defaultValue = "1") int age) {
    System.out.println(id + ", " + name + ", " + age);
    return "test1";
}
```

3. @PathVariable

```java
// 3. 路径变量：@PathVariable
@RequestMapping("user/{id}")
public String t3(@PathVariable Long id) {
    System.out.println(id);

    return "test1";
}
```

4. @RequestHeader, @CookieValue

```java
// 4. @RequestHeader, @CookieValue
@RequestMapping("t4")
public String t4(@RequestHeader("User-Agent") String agent,
@CookieValue("JSESSIONID") String sessionId) {
    System.out.println(agent);
    System.out.println(sessionId);

    return "test1";
}
```

5. command/form对象（JavaBean）

- 简单情况

```java
// 5. command/form对象（JavaBean）
@RequestMapping("t5")
public String t5(User user) {// 简单类型,无冲突情况下,自动绑定
    System.out.println(user);
    return "test1";
}
```

- 复杂类型：@InitBinder

```java
// 日期类型：需要提供转换器、格式化器、属性编辑器
@RequestMapping("t1")
public void t1(Date ctime) {
    System.out.println(ctime);
}
// 只能应用于当前该的所有方法，进行数据绑定
@InitBinder
public void init(WebDataBinder binder) {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    binder.registerCustomEditor(Date.class, new
CustomDateEditor(df, true));
}
```

- 存在冲突：@InitBinder

```java
// 同时绑定多个对象
@RequestMapping("t2")
public void t2(User user, Student stu) {
    System.out.println(user.getName() + ", " + user.getAge());
```

```
    System.out.println(stu.getName() + ", " + stu.getSno());
}

// 存在关联关系
@RequestMapping("t3")
public void t3(Student stu) {
    System.out.println(stu.getName() + ", " + stu.getSno());
    System.out.println(stu.getUser().getName() + ", " +
stu.getUser().getAge());
}

// 绑定User类型对象
@InitBinder("user")
public void init1(WebDataBinder binder) {
    binder.setFieldDefaultPrefix("user.");
}

@InitBinder("student")
public void init2(WebDataBinder binder) {
    binder.setFieldDefaultPrefix("stu.");
}
```

6. Errors或BindingResult

- 参数绑定信息，可使用@Valid注解进行后端数据验证
- 存在多个BindingResult,只需跟在参数后面就行

```
// 7. Errors或BindingResult：必须紧跟在JavaBean之后，表示该
JavaBean绑定的结果
// 如果绑定失败，页面显示400错误：Bad Request
// 绑定的错误信息，存储在BindingResult对象中
@RequestMapping("t7_2")
public String t7_2(User user, BindingResult br) {
    System.out.println(user);
    // 如果绑定出错
    if(br.hasErrors()) {
        List<FieldError> errors = br.getFieldErrors();
        for(FieldError err : errors) {
            System.out.println(err.getField() + ": " +
err.getDefaultMessage());
        }
    }
    return "test1";
}
```

7. Map, Model, ModelMap（输出参数，代表数据）

```
// 用于传递数据，类似于request.setAttribute(),
session.setAttribute()
@RequestMapping("t6")
public String t6(Model model) {
```

```
    model.addAttribute("name", "张三");
    return "test1";
}

@RequestMapping("t6_2")
public String t6_2(Map<String, Object> map) {
    map.put("name", "李四");
    return "test1";
}

@RequestMapping("t6_3")
public String t6_3(ModelMap mm) {
    mm.addAttribute("name", "王五");
    mm.put("id", 100);
    return "test1";
}
```

8. SessionStatus

   用于清空session中的值，与@SessionAttributes()有关

9. RedirectAttributes: 例如 RedirectAttributes attr

- `attr.addAttribute("param", value);`

  这种方式就相当于重定向之后，在url后面拼接参数

  可以用`@RequestParam/request.getParameter`获得

- `attr.addFlashAttribute("param", value);`

  使用Flash传递数据,基于session

  可以用`@ModelAttribute/model,modelmap`获取

```
@RequestMapping("/redirect")
public String redirectTest(RedirectAttributes attr){
    attr.addAttribute("userName", "root");
    attr.addFlashAttribute("password","123456");
    return "redirect:/book/getbook";
}

@RequestMapping("/getbook")
@ResponseBody
public String getBook(ModelMap map,
@ModelAttribute("password") String password1,
@RequestParam(value="password",required=falses) String
password2,
HttpServletRequest request,
@RequestParam("userName") String userName,){
    System.out.println("userName : "+map.get("userName"))
    //null
    System.out.println("userName1 :"+
```

```
                        request.getAttribute("userName"));//null

        System.out.println("userName2 :" +
    request.getParameter("userName"));   //root
        System.out.println("userName3 : " + userName);  //root
        //取
        System.out.println("password : "+map.get("password"));
        // 123456
        System.out.println("password1 :" +
    request.getAttribute("password")); //null
        System.out.println("password2 :" +
    request.getParameter("password"));  //null
        System.out.println("password3 : " + password2);  //null
        System.out.println("password3 : " + password1);   // 123456

     RequestContextUtils.getInputFlashMap(request).get("password");
            //123456
        return "result";
    }
```

10.4.2 返回值

1. String / View:表示视图名称

- return "view"
  - string类型

```
// String类型：返回视图名称（服务器端跳转）
@RequestMapping("t1")
public String t1() {
return "t1";
}
```

  - void类型

```
// void：使用默认的视图名称（映射的url: hello/t5）
@RequestMapping("t5")
public void t5() {
    System.out.println("t5...");
}
```

- return "redirect:xx": attribute会填入url

```
// 页面跳转前缀：redirect（）
@RequestMapping("t2")
public String t2(Model model) {
    model.addAttribute("name", "张三");
    model.addAttribute("age", 20);
    return "redirect:/user/t1.jsp";
}
```

- return "forward:xx": attribute不会填入url

```java
// 页面跳转前缀：forward（）
@RequestMapping("t3")
public String t3(Model model) {
    model.addAttribute("name", "张三");
    model.addAttribute("age", 20);
    return "forward:/user/t1.jsp";
}
```

2. ModelAndView：返回Model和View

   在@RestController中，如果处理方法进行页面跳转，则需要返回ModelAndView对象

```java
// ModelAndView：早期的使用方式(一般不用)
// ModelAndView：主要用于@RestController中，需要页面跳转的请求映射
@RequestMapping("t6")
public ModelAndView t6() {
    ModelAndView mv = new ModelAndView("t6");
    mv.addObject("name", "张三");
    mv.addObject("age", 20);
    return mv;
}
```

3. Model，Map，ModelMap：返回数据

   使用默认的view名称（和请求url有关）

```java
// 返回model数据，使用默认的视图名称（因为没有指定view）
@RequestMapping("t7")
public Map<String, Object> t7() {
    Map<String, Object> map = new HashMap<>();
    map.put("name", "张三");
    map.put("age", 20);
    return map;
}

@RequestMapping("t7_2")
public void t7_2(Map<String, Object> map) {
    map.put("name", "张三");
    map.put("age", 20);
}

@RequestMapping("t7_3")
public User t7_3() {
    User u = new User();
    u.setName("张三");
    u.setAge(20);
    return u;
}
```

4. void：

- 使用默认的view名称（和请求url有关）
- 使用servlet api(*request, response*) 进行页面跳转或直接输出

```
// void：使用request, response进行页面跳转、输出JSON、文件下载等
@RequestMapping("t4")
public void t4(HttpServletRequest req, HttpServletResponse resp)
throws Exception {
    String path = req.getContextPath();
    resp.sendRedirect(path + "/user/t4.jsp?name=hello&age=20");
}
```

5. Java的对象：
   一般配合@ResponseBody使用，将对象渲染成JSON字符串（用于AJAX请求）

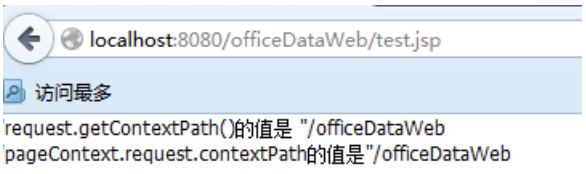
```
@RequestMapping("t3")
@ResponseBody
public Msg t3(String name, Integer age) {
    System.out.println(name + ", " + age);
    return new Msg();
}
```

10.4.3 页面跳转传参

1. *redirect , forward*
2. *request, response*
3. *ModelAndView*
4. *RedirectAttributes*

10.4.4 sundry

1. ${pageContext.request.contextPath}:项目路径



2. jsp页面可以放在 WEB-INF/目录下，静态资源文件不可以放在 WEB-INF/目录下,url不可直接访问
3. 400错误：控制器处理方法的参数绑定出错了

THE END