课程目标

1、通过对本章内容的学习，了解设计模式的由来。

2、介绍设计模式能帮我们解决哪些问题。

3、剖析工厂模式的历史由来及应用场景。

内容定位

不用设计模式并非不可以，但是用好设计模式能帮助我们更好地解决实际问题，设计模式最重要的是解耦。设计模式天天都在用，但自己却无感知。我们把设计模式作为一个专题，主要是学习设计模式是如何总结经验的，把经验为自己所用。学设计模式也是锻炼将业务需求转换技术实现的一种非常有效的方式。

回顾软件设计原则

在讲设计模式之前，我们一定要先了解软件设计原则。现在先来回顾一下软件设计七大原则：

设计原则	解释
开闭原则	对扩展开放，对修改关闭。
依赖倒置原则	通过抽象使各个类或者模块不相互影响，实现松耦合。
单一职责原则	一个类、接口、方法只做一件事。
接口隔离原则	尽量保证接口的纯洁性，客户端不应该依赖不需要的接口。
迪米特法则	又叫最少知道原则，一个类对其所依赖的类知道得越少越好。

里氏替换原则	子类可以扩展父类的功能但不能改变父类原有的功能。
合成复用原则	尽量使用对象组合、聚合，而不使用继承关系达到代码复用的目的。

为什么我们的课程要从设计模式开始？

写出优雅的代码

先来看一段很多年以前写的代码：

```
public void setCurForm(Gw_exammingForm curForm,String parameters)throws BaseException {
```

```
      JSONObject jsonObj = new JSONObject(parameters);
      //
      if(jsonObj.getString("examinationPaper_id")!= null &&
(!jsonObj.getString("examinationPaper_id").equals("")))
         curForm.setExaminationPaper_id(jsonObj.getLong("examinationPaper_id"));
      //
      if(jsonObj.getString("leavTime") != null && (!jsonObj.getString("leavTime").equals("")))
         curForm.setLeavTime(jsonObj.getInt("leavTime"));
      //
      if(jsonObj.getString("organization_id")!= null &&
(!jsonObj.getString("organization_id").equals("")))
         curForm.setOrganization_id(jsonObj.getLong("organization_id"));
      //
      if(jsonObj.getString("id")!= null && (!jsonObj.getString("id").equals("")))
         curForm.setId(jsonObj.getLong("id"));
      //
      if(jsonObj.getString("examroom_id")!= null &&
(!jsonObj.getString("examroom_id").equals("")))
         curForm.setExamroom_id(jsonObj.getLong("examroom_id"));
      //
      if(jsonObj.getString("user_id")!= null && (!jsonObj.getString("user_id").equals("")))
         curForm.setUser_id(jsonObj.getLong("user_id"));
      //
      if(jsonObj.getString("specialtyCode")!= null &&
(!jsonObj.getString("specialtyCode").equals("")))
         curForm.setSpecialtyCode(jsonObj.getLong("specialtyCode"));
      //
      if(jsonObj.getString("postionCode")!= null &&
(!jsonObj.getString("postionCode").equals("")))
         curForm.setPostionCode(jsonObj.getLong("postionCode"));
      //
      if(jsonObj.getString("gradeCode")!= null && (!jsonObj.getString("gradeCode").equals("")))
         curForm.setGradeCode(jsonObj.getLong("gradeCode"));
      //

      curForm.setExamStartTime(jsonObj.getString("examStartTime"));
      //
      curForm.setExamEndTime(jsonObj.getString("examEndTime"));
      //
      if(jsonObj.getString("single_selectionImpCount")!= null &&
           (!jsonObj.getString("single_selectionImpCount").equals("")))
         curForm.setSingle_selectionImpCount(jsonObj.getInt("single_selectionImpCount"));
      //
      if(jsonObj.getString("multi_selectionImpCount")!= null &&
```

```
            (!jsonObj.getString("multi_selectionImpCount").equals("")))
          curForm.setMulti_selectionImpCount(jsonObj.getInt("multi_selectionImpCount"));
      //

      if(jsonObj.getString("judgementImpCount")!= null &&
            (!jsonObj.getString("judgementImpCount").equals("")))
         curForm.setJudgementImpCount(jsonObj.getInt("judgementImpCount"));
      //
      if(jsonObj.getString("examTime")!= null && (!jsonObj.getString("examTime").equals("")))
         curForm.setExamTime(jsonObj.getInt("examTime"));
      //
      if(jsonObj.getString("fullScore")!= null && (!jsonObj.getString("fullScore").equals("")))
         curForm.setFullScore(jsonObj.getLong("fullScore"));
      //

      if(jsonObj.getString("passScore")!= null && (!jsonObj.getString("passScore").equals("")))
         curForm.setPassScore(jsonObj.getLong("passScore"));
      //
      curForm.setUserName(jsonObj.getString("user_name"));
      //
      if(jsonObj.getString("score")!= null && (!jsonObj.getString("score").equals("")))
         curForm.setScore(jsonObj.getLong("score"));
      //
      curForm.setResult(jsonObj.getString("result"));
      curForm.setIsPassed(jsonObj.getString("result"));
      //

      if(jsonObj.getString("single_ok_count")!= null &&
(!jsonObj.getString("single_ok_count").equals("")))
         curForm.setSingle_ok_count(jsonObj.getInt("single_ok_count"));
      //
      if(jsonObj.getString("multi_ok_count")!= null &&
(!jsonObj.getString("multi_ok_count").equals("")))
         curForm.setMulti_ok_count(jsonObj.getInt("multi_ok_count"));
      //
      if(jsonObj.getString("judgement_ok_count")!= null &&
(!jsonObj.getString("judgement_ok_count").equals("")))
         curForm.setJudgement_ok_count(jsonObj.getInt("judgement_ok_count"));
}
```

优化之后的代码：

```
public class ExamPaper extends Gw_exammingForm{

   private String examinationPaperId;//
```

```
   private String leavTime;//
   private String organizationId;//
   private String id;//
   private String examRoomId;//
   private String userId;//
   private String specialtyCode;//
   private String postionCode;//
   private String gradeCode;//
   private String examStartTime;//
   private String examEndTime;//
   private String singleSelectionImpCount;//
   private String multiSelectionImpCount;//
   private String judgementImpCount;//
   private String examTime;//
   private String fullScore;//
   private String passScore;//
   private String userName;//
   private String score;//
   private String resut;//
   private String singleOkCount;//
   private String multiOkCount;//
   private String judgementOkCount;//

}

public void setCurFormpublic void setCurForm(Gw_exammingForm curForm,String parameters)throws
BaseException {
   try {
      JSONObject jsonObj = new JSONObject(parameters);
      ExamPaper examPaper = JSONObject.parseObject(parameters,ExamPaper.class);

      curForm = examPaper;

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

}
```

更好地重构项目

平时我们写的代码虽然满足了需求但往往不利于项目的开发与维护，以下面的 JDBC 代码为例：

```
public void save(Student stu){
    String sql="INSERT INTO t_student(name,age) VALUES(?,?)";
    Connection conn=null;
    Statement st=null;
    try{
        // 1.
        Class.forName("com.mysql.jdbc.Driver");
        // 2.
        conn=DriverManager.getConnection("jdbc:mysql:///jdbcdemo","root","root");
        // 3.
        PreparedStatement ps=conn.prepareStatement(sql);
        ps.setObject(1,stu.getName());
        ps.setObject(2,stu.getAge());
        // 4.   SQL
        ps.executeUpdate();
        // 5.
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        try{
            if(st!=null)
                st.close();
        }catch(SQLException e){
            e.printStackTrace();
        }finally{
            try{
                if(conn!=null)
                    conn.close();
            }catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
}

//
public void delete(Long id){
    String sql="DELETE  FROM t_student WHERE id=?";
    Connection conn=null;
    Statement st=null;
    try{
        // 1.
        Class.forName("com.mysql.jdbc.Driver");
        // 2.
```

```
        conn=DriverManager.getConnection("jdbc:mysql:///jdbcdemo","root","root");
        // 3.
        PreparedStatement ps=conn.prepareStatement(sql);
        ps.setObject(1,id);
        // 4.   SQL
        ps.executeUpdate();
        // 5.
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        try{
            if(st!=null)
                st.close();
        }catch(SQLException e){
            e.printStackTrace();
        }finally{
            try{
                if(conn!=null)
                    conn.close();
            }catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
}

//
public void update(Student stu){
    String sql="UPDATE t_student SET name=?,age=? WHERE id=?";
    Connection conn=null;
    Statement st=null;
    try{
        // 1.
        Class.forName("com.mysql.jdbc.Driver");
        // 2.
        conn=DriverManager.getConnection("jdbc:mysql:///jdbcdemo","root","root");
        // 3.
        PreparedStatement ps=conn.prepareStatement(sql);
        ps.setObject(1,stu.getName());
        ps.setObject(2,stu.getAge());
        ps.setObject(3,stu.getId());
        // 4.   SQL
        ps.executeUpdate();
        // 5.
```

```
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        try{
            if(st!=null)
                st.close();
        }catch(SQLException e){
            e.printStackTrace();
        }finally{
            try{
                if(conn!=null)
                    conn.close();
            }catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
}
```

上述代码中功能没问题，但是代码重复的太多，因此我们可以进行抽取，把重复的代码放到一个工具类 JdbcUtil 里。

```
//
public class JdbcUtil {
    private JdbcUtil() { }
    static {
        // 1.
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static Connection getConnection() {
        try {
            // 2.
            return DriverManager.getConnection("jdbc:mysql:///jdbcdemo", "root", "root");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
```

```
    }

    //
    public static void close(ResultSet rs, Statement st, Connection conn) {
        try {
            if (rs != null)
                rs.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (st != null)
                    st.close();
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                try {
                    if (conn != null)
                        conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

在实现类中直接调用工具类 JdbcUtil 中的方法即可

```
//
public void save(Student stu) {
    String sql = "INSERT INTO t_student(name,age) VALUES(?,?)";
    Connection conn = null;
    PreparedStatement ps=null;
    try {
        conn = JDBCUtil.getConnection();
        // 3.
        ps = conn.prepareStatement(sql);
        ps.setObject(1, stu.getName());
        ps.setObject(2, stu.getAge());
        // 4.   SQL
        ps.executeUpdate();
        // 5.
```

```
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtil.close(null, ps, conn);
    }

}

//
public void delete(Long id) {
    String sql = "DELETE  FROM t_student WHERE id=?";
    Connection conn = null;
    PreparedStatement ps = null;
    try {
        conn=JDBCUtil.getConnection();
        // 3.
        ps = conn.prepareStatement(sql);
        ps.setObject(1, id);
        // 4.   SQL
        ps.executeUpdate();
        // 5.
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtil.close(null, ps, conn);
    }

}

//
public void update(Student stu) {
    String sql = "UPDATE t_student SET name=?,age=? WHERE id=?";
    Connection conn = null;
    PreparedStatement ps = null;
    try {
        conn=JDBCUtil.getConnection();
        // 3.
        ps = conn.prepareStatement(sql);
        ps.setObject(1, stu.getName());
        ps.setObject(2, stu.getAge());
        ps.setObject(3, stu.getId());
        // 4.   SQL
        ps.executeUpdate();
        // 5.
```

```
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtil.close(null, ps, conn);
    }

}

public Student get(Long id) {
    String sql = "SELECT * FROM t_student WHERE id=?";
    Connection conn = null;
    Statement st = null;
    ResultSet rs = null;
    PreparedStatement ps=null;
    try {
        conn = JDBCUtil.getConnection();
        // 3.
        ps = conn.prepareStatement(sql);
        ps.setObject(1, id);
        // 4.   SQL
        rs = ps.executeQuery();
        if (rs.next()) {
            String name = rs.getString("name");
            int age = rs.getInt("age");
            Student stu = new Student(id, name, age);
            return stu;
        }
        // 5.
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtil.close(rs, ps, conn);
    }
    return null;
}

public List<Student> list() {
    List<Student> list = new ArrayList<>();
    String sql = "SELECT * FROM t_student ";
    Connection conn = null;
    Statement st = null;
    ResultSet rs = null;
    PreparedStatement ps=null;
    try {
```

```
        conn=JDBCUtil.getConnection();
        // 3.
        ps = conn.prepareStatement(sql);
        // 4.   SQL
        rs = ps.executeQuery();
        while (rs.next()) {
            long id = rs.getLong("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            Student stu = new Student(id, name, age);
            list.add(stu);
        }
        // 5.
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtil.close(rs, ps, conn);
    }
    return list;
}
```

虽然完成了重复代码的抽取，但数据库中的账号密码等直接显示在代码中，不利于后期账户密码改动的维护，我们可以建立个 db.propertise 文件用来存储这些信息

```
driverClassName =com.mysql.jdbc.Driver
url =jdbc:mysql:///jdbcdemo
username =root
password =root
```

只需在工具类 JdbcUtil 中获取里面的信息即可

```
static {
    // 1.
    try {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        InputStream inputStream = loader.getResourceAsStream("db.properties");
        p = new Properties();
        p.load(inputStream);
        Class.forName(p.getProperty("driverClassName"));
    } catch (Exception e) {
        e.printStackTrace();
```

```
    }
}

public static Connection getConnection() {
    try {
        // 2.
        return DriverManager.getConnection(p.getProperty("url"), p.getProperty("username"),
        p.getProperty("password"));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

抽取到这里貌似已经完成，但在实现类中，依然存在部分重复代码，在 DML 操作中，除了 SQL 和设置值的不同，其他都相同，将相同的抽取出去，不同的部分通过参数传递进来，无法直接放在工具类中，这时我们可以创建一个模板类 JdbcTemplate，创建一个 DML 和 DQL 的模板来进行对代码的重构。

```
//
public static List<Student> query(String sql,Object...params){
    List<Student> list=new ArrayList<>();
    Connection conn = null;
    PreparedStatement ps=null;
    ResultSet rs = null;
    try {
        conn=JDBCUtil.getConnection();
        ps=conn.prepareStatement(sql);
        //
        for (int i = 0; i < params.length; i++) {
            ps.setObject(i+1, params[i]);
        }
        rs = ps.executeQuery();
        while (rs.next()) {
            long id = rs.getLong("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            Student stu = new Student(id, name, age);
            list.add(stu);
        }
```

```
        // 5.
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtil.close(rs, ps, conn);
    }
    return list;
}
```

实现类直接调用方法即可。

```
//
public void save(Student stu) {
    String sql = "INSERT INTO t_student(name,age) VALUES(?,?)";
    Object[] params=new Object[]{stu.getName(),stu.getAge()};
    JdbcTemplate.update(sql, params);
}

//
public void delete(Long id) {
    String sql = "DELETE FROM t_student WHERE id = ?";
    JdbcTemplate.update(sql, id);
}

//
public void update(Student stu) {
    String sql = "UPDATE t_student SET name = ?,age = ? WHERE id = ?";
    Object[] params=new Object[]{stu.getName(),stu.getAge(),stu.getId()};
    JdbcTemplate.update(sql, params);
}

public Student get(Long id) {
    String sql = "SELECT * FROM t_student WHERE id=?";
    List<Student> list = JDBCTemplate.query(sql, id);
    return list.size()>0? list.get(0):null;
}

public List<Student> list() {
    String sql = "SELECT * FROM t_student ";
    return JDBCTemplate.query(sql);
}
```

这样重复的代码基本就解决了，但又个很严重的问题就是这个程序 DQL 操作中只能处理 Student 类和 t_student 表的相关数据，无法处理其他类如：Teacher 类和 t_teacher 表。不同表（不同的对象），不同的表就应该有不同列，不同列处理结果集的代码就应该不一样，处理结果集的操作只有 DAO 自己最清楚，也就是说，处理结果的方法压根就不应该放在模板方中，应该由每个 DAO 自己来处理。因此我们可以创建一个 IRowMapper 接口来处理结果集

```
public interface IRowMapper {
    //
    List rowMapper(ResultSet rs) throws Exception;
}
```

DQL 模板类中调用 IRowMapper 接口中的 handle 方法，提醒实现类去自己去实现 mapping 方法

```
public static List<Student> query(String sql,IRowMapper rsh, Object...params){
    List<Student> list = new ArrayList<>();
    Connection conn = null;
    PreparedStatement ps=null;
    ResultSet rs = null;
    try {
        conn = JdbcUtil.getConnection();
        ps = conn.prepareStatement(sql);
        //
        for (int i = 0; i < params.length; i++) {
            ps.setObject(i+1, params[i]);
        }
        rs = ps.executeQuery();
        return rsh.mapping(rs);
        // 5.
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JdbcUtil.close(rs, ps, conn);
    }
    return list ;
```

```
}
```

实现类自己去实现 IRowMapper 接口的 mapping 方法，想要处理什么类型数据在里面定义即可

```
public Student get(Long id) {
    String sql = "SELECT * FROM t_student WHERE id = ?";
    List<Student> list = JdbcTemplate.query(sql,new StudentRowMapper(), id);
    return list.size()>0? list.get(0):null;
}
public List<Student> list() {
    String sql = "SELECT * FROM t_student ";
    return JdbcTemplate.query(sql,new StudentRowMapper());
}
class StudentRowMapper implements IRowMapper{
    public List mapping(ResultSet rs) throws Exception {
        List<Student> list=new ArrayList<>();
        while(rs.next()){
            long id = rs.getLong("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            Student stu=new Student(id, name, age);
            list.add(stu);
        }
        return list;
    }
}
```

好了，基本已经大功告成了，但是 DQL 查询不单单只有查询学生信息（List 类型），还可以查询学生数量，这时就要通过泛型来完成

```
public interface IRowMapper<T> {
    //
    T mapping(ResultSet rs) throws Exception;
}
```

```
public static <T> T query(String sql,IRowMapper<T> rsh, Object...params){
    Connection conn = null;
```

```
    PreparedStatement ps=null;
    ResultSet rs = null;
    try {
        conn = JdbcUtil.getConnection();
        ps = conn.prepareStatement(sql);
        //
        for (int i = 0; i < params.length; i++) {
            ps.setObject(i+1, params[i]);
        }
        rs = ps.executeQuery();
        return rsh.mapping(rs);
        // 5.
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JdbcUtil.close(rs, ps, conn);
    }
    return null;
}
```

StudentRowMapper 类：

```
class StudentRowMapper implements IRowMapper<List<Student>>{
    public List<Student> mapping(ResultSet rs) throws Exception {
        List<Student> list=new ArrayList<>();
        while(rs.next()){
            long id = rs.getLong("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            Student stu=new Student(id, name, age);
            list.add(stu);
        }
        return list;
    }
}
```

这样不仅可以查询 List，还可以查询学生数量：

```
public Long getCount(){
    String sql = "SELECT COUNT(*) total FROM t_student";
    Long totalCount = (Long) JdbcTemplate.query(sql,
            new IRowMapper<Long>() {
```

```
            public Long mapping(ResultSet rs) throws Exception {
                Long totalCount = null;
                if(rs.next()){
                    totalCount = rs.getLong("total");
                }
                return totalCount;
            }
        });
    return totalCount;
}
```

好了，重构设计已经完成，好的代码能让我们以后维护更方便，因此学会对代码的重构是非常重要的。

经典框架都在用设计模式解决问题

Spring 就是一个把设计模式用得淋漓尽致的经典框架，其实从类的命名就能看出来，我来一一列举：

设计模式名称	举例
工厂模式	BeanFactory
装饰器模式	BeanWrapper
代理模式	AopProxy
委派模式	DispatcherServlet
策略模式	HandlerMapping
适配器模式	HandlerAdapter
模板模式	JdbcTemplate
观察者模式	ContextLoaderListener

需要特别声明的是，设计模式从来都不是单个设计模式独立使用的。在实际应用中，通常是多个设计模式混合使用 ,你中有我 ,我中有你。我们的课程中 ,会围绕 Spring 的 IOC、AOP、MVC、JDBC 这样的思路展开，根据其设计类型来设计讲解顺序：

类型	名称	英文
创建型模式	工厂模式	Factory Pattern
	单例模式	Singleton Pattern
	原型模式	Prototype Pattern
结构型模式	适配器模式	Adapter Pattern
	装饰器模式	Decorator Pattern
	代理模式	Proxy Pattern
行为性模式	策略模式	Strategy Pattern
	模板模式	Template Pattern
	委派模式	Delegate Pattern
	观察者模式	Observer Pattern

工厂模式详解

工厂模式的历史由来

在现实生活中我们都知道，原始社会自给自足（没有工厂）、农耕社会小作坊（简单工厂，民间酒坊）、工业革命流水线(工厂方法，自产自销)、现代产业链代工厂(抽象工厂，富士康)

原始社会自给自足　　农耕社会小作坊

工厂流水线生产　　现代产业链代工厂

我们的项目代码同样也是由简而繁一步一步迭代而来，但对于调用者来说确是越来越简单化。

简单工厂模式

简单工厂模式（Simple Factory Pattern）是指由一个工厂对象决定创建出哪一种产品类的实例，但它不属于 GOF，23 种设计模式（参考资料：http://en.wikipedia.org/wiki/Design_Patterns#Patterns_by_Type）。简单工厂适用于工厂类负责创建的对象较少的场景，且客户端只需要传入工厂类的参数，对于如何创建对象的逻辑不需要关心。

接下来我们来看代码，还是以课程为例。咕泡学院目前开设有 Java 架构、大数据、人工智能等课程，已经形成了一个生态。我们可以定义一个课程标准 ICourse 接口：

```
public interface ICourse {
    /** 录制视频 */
    public void record();
}
```

创建一个 Java 课程的实现 JavaCourse 类：

```
public class JavaCourse implements ICourse {
    public void record() {
        System.out.println("录制 Java 课程");
    }
```

```
}
```

看客户端调用代码，我们会这样写：

```
public static void main(String[] args) {
    ICourse course = new JavaCourse();
    course.record();
}
```

看上面的代码，父类 ICourse 指向子类 JavaCourse 的引用，应用层代码需要依赖 JavaCourse，如果业务扩展，我继续增加 PythonCourse 甚至更多，那么我们客户端的依赖会变得越来越臃肿。因此，我们要想办法把这种依赖减弱，把创建细节隐藏。虽然目前的代码中，我们创建对象的过程并不复杂，但从代码设计角度来讲不易于扩展。现在，我们用简单工厂模式对代码进行优化。先增加课程 PythonCourse 类：

```
public class PythonCourse implements ICourse {
    public void record() {
        System.out.println("录制 Python 课程");
    }
}
```

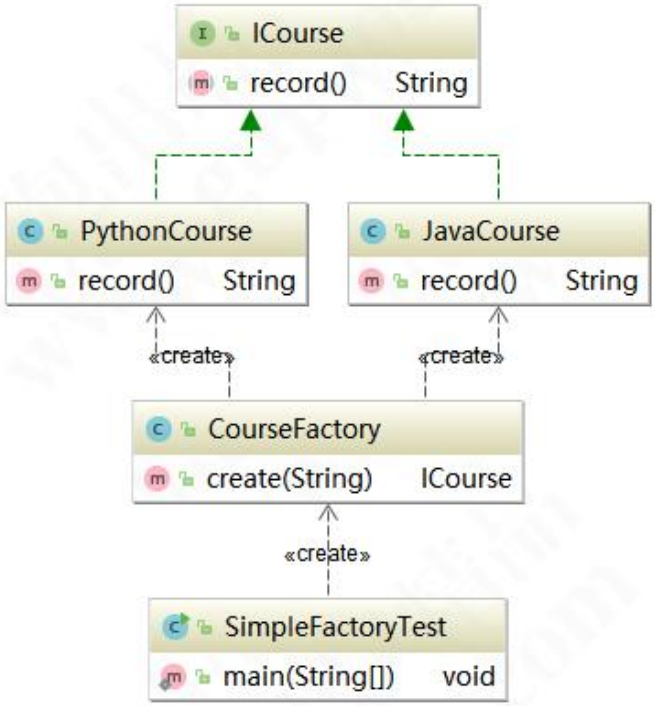
创建 CourseFactory 工厂类：

```
public class CourseFactory {
    public ICourse create(String name){
        if("java".equals(name)){
            return new JavaCourse();
        }else if("python".equals(name)){
            return new PythonCourse();
        }else {
            return null;
        }
    }
}
```

修改客户端调用代码：

```
public class SimpleFactoryTest {
    public static void main(String[] args) {
        CourseFactory factory = new CourseFactory();
        factory.create("java");
    }
}
```

当然，我们为了调用方便，可将 factory 的 create()改为静态方法，下面来看一下类图：



客户端调用是简单了 ,但如果我们业务继续扩展 ,要增加前端课程 ,那么工厂中的 create() 就要根据产品链的丰富每次都要修改代码逻辑。不符合开闭原则。因此，我们对简单工厂还可以继续优化，可以采用反射技术：

```
public class CourseFactory {
    public ICourse create(String className){
        try {
            if (!(null == className || "".equals(className))) {
                return (ICourse) Class.forName(className).newInstance();
            }
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }
}
```

修改客户端调用代码：

```
public static void main(String[] args) {
        CourseFactory factory = new CourseFactory();
        ICourse course =
```

```
factory.create("com.gupaoedu.vip.pattern.factory.simplefactory.JavaCourse");
        course.record();
}
```

优化之后，产品不断丰富不需要修改 CourseFactory 中的代码。但是，有个问题是，方法参数是字符串，可控性有待提升，而且还需要强制转型。我们再修改一下代码：

```
public ICourse create(Class<? extends ICourse> clazz){
    try {
        if (null != clazz) {
            return clazz.newInstance();
        }
    }catch (Exception e){
        e.printStackTrace();
    }
    return null;
}
```

优化客户端代码：

```
public static void main(String[] args) {
    CourseFactory factory = new CourseFactory();
    ICourse course = factory.create(JavaCourse.class);
    course.record();
}
```

再看一下类图：

简单工厂模式在 JDK 源码也是无处不在，现在我们来举个例子，例如 Calendar 类，看 Calendar.getInstance()方法，下面打开的是 Calendar 的具体创建类：

```
private static Calendar createCalendar(TimeZone zone,
                                       Locale aLocale)
{
```

```
    CalendarProvider provider =
        LocaleProviderAdapter.getAdapter(CalendarProvider.class, aLocale)
                             .getCalendarProvider();
    if (provider != null) {
        try {
            return provider.getInstance(zone, aLocale);
        } catch (IllegalArgumentException iae) {
            // fall back to the default instantiation
        }
    }

    Calendar cal = null;

    if (aLocale.hasExtensions()) {
        String caltype = aLocale.getUnicodeLocaleType("ca");
        if (caltype != null) {
            switch (caltype) {
            case "buddhist":
            cal = new BuddhistCalendar(zone, aLocale);
                break;
            case "japanese":
                cal = new JapaneseImperialCalendar(zone, aLocale);
                break;
            case "gregory":
                cal = new GregorianCalendar(zone, aLocale);
                break;
            }
        }
    }
    if (cal == null) {
        // If no known calendar type is explicitly specified,
        // perform the traditional way to create a Calendar:
        // create a BuddhistCalendar for th_TH locale,
        // a JapaneseImperialCalendar for ja_JP_JP locale, or
        // a GregorianCalendar for any other locales.
        // NOTE: The language, country and variant strings are interned.
        if (aLocale.getLanguage() == "th" && aLocale.getCountry() == "TH") {
            cal = new BuddhistCalendar(zone, aLocale);
        } else if (aLocale.getVariant() == "JP" && aLocale.getLanguage() == "ja"
                   && aLocale.getCountry() == "JP") {
            cal = new JapaneseImperialCalendar(zone, aLocale);
        } else {
            cal = new GregorianCalendar(zone, aLocale);
        }
```

```
    }
    return cal;
}
```

还有一个大家经常使用的 logback，我们可以看到 LoggerFactory 中有多个重载的方法 getLogger()：

```
public static Logger getLogger(String name) {
    ILoggerFactory iLoggerFactory = getILoggerFactory();
    return iLoggerFactory.getLogger(name);
}

public static Logger getLogger(Class clazz) {
    return getLogger(clazz.getName());
}
```

简单工厂也有它的缺点：工厂类的职责相对过重，不易于扩展过于复杂的产品结构。

工厂方法模式

工厂方法模式（Fatory Method Pattern）是指定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类，工厂方法让类的实例化推迟到子类中进行。在工厂方法模式中用户只需要关心所需产品对应的工厂，无须关心创建细节，而且加入新的产品符合开闭原则。

工厂方法模式主要解决产品扩展的问题，在简单工厂中，随着产品链的丰富，如果每个课程的创建逻辑有区别的话，工厂的职责会变得越来越多，有点像万能工厂，并不便于维护。根据单一职责原则我们将职能继续拆分，专人干专事。Java 课程由 Java 工厂创建，Python 课程由 Python 工厂创建，对工厂本身也做一个抽象。来看代码，先创建 ICourseFactory 接口：

```
public interface ICourseFactory {
    ICourse create();
}
```

在分别创建子工厂，JavaCourseFactory 类：

```
import com.gupaoedu.vip.pattern.factory.ICourse;
import com.gupaoedu.vip.pattern.factory.JavaCourse;
```

```
public class JavaCourseFactory implements ICourseFactory {
    public ICourse create() {
        return new JavaCourse();
    }
}
```

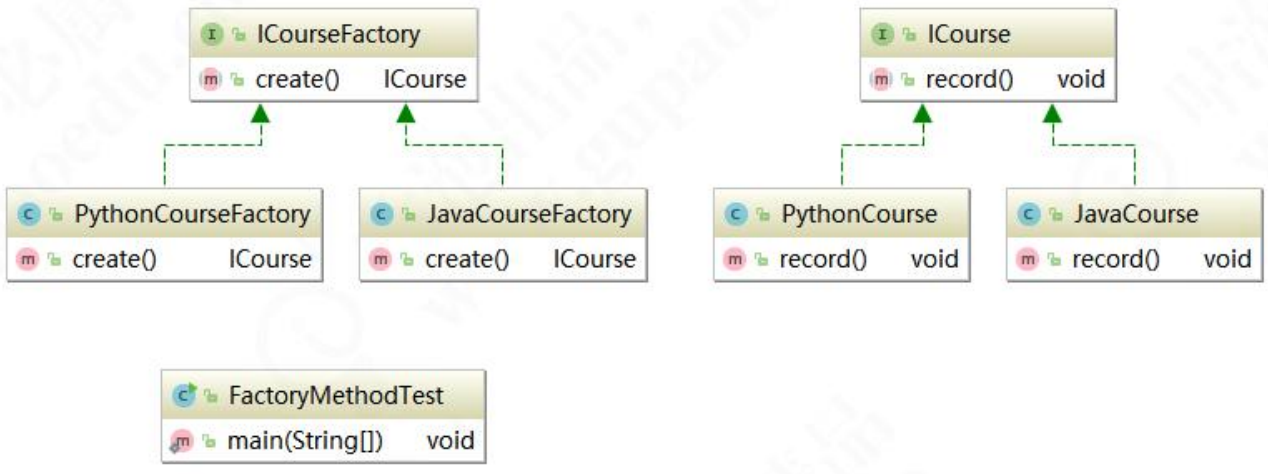
PythonCourseFactory 类：

```
import com.gupaoedu.vip.pattern.factory.ICourse;
import com.gupaoedu.vip.pattern.factory.PythonCourse;

public class PythonCourseFactory implements ICourseFactory {
    public ICourse create() {
        return new PythonCourse();
    }
}
```

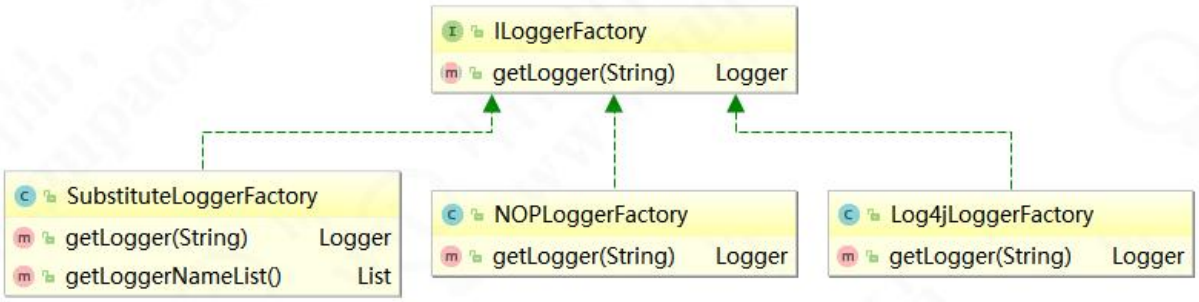
看测试代码：

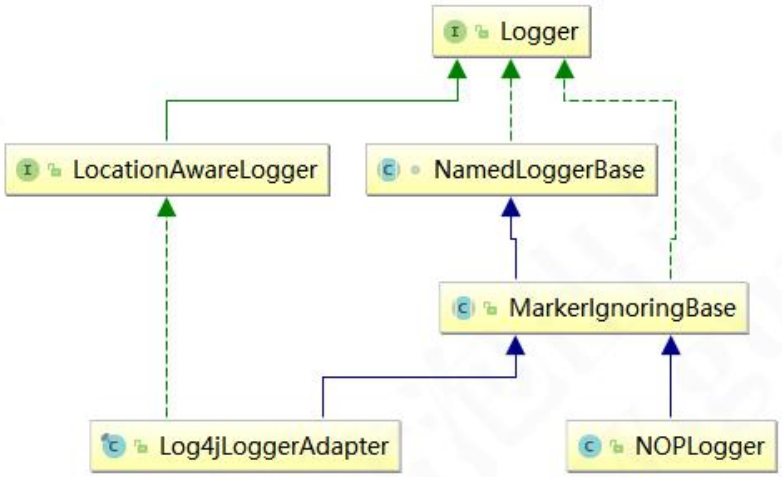
```
public static void main(String[] args) {
    ICourseFactory factory = new PythonCourseFactory();
    ICourse course = factory.create();
    course.record();

    factory = new JavaCourseFactory();
    course = factory.create();
    course.record();
}
```

现在再来看一下类图：



再来看看 logback 中工厂方法模式的应用，看看类图就 OK 了：





工厂方法适用于以下场景：

1、创建对象需要大量重复的代码。

2、客户端（应用层）不依赖于产品类实例如何被创建、实现等细节。

3、一个类通过其子类来指定创建哪个对象。

工厂方法也有缺点：

1、类的个数容易过多，增加复杂度。

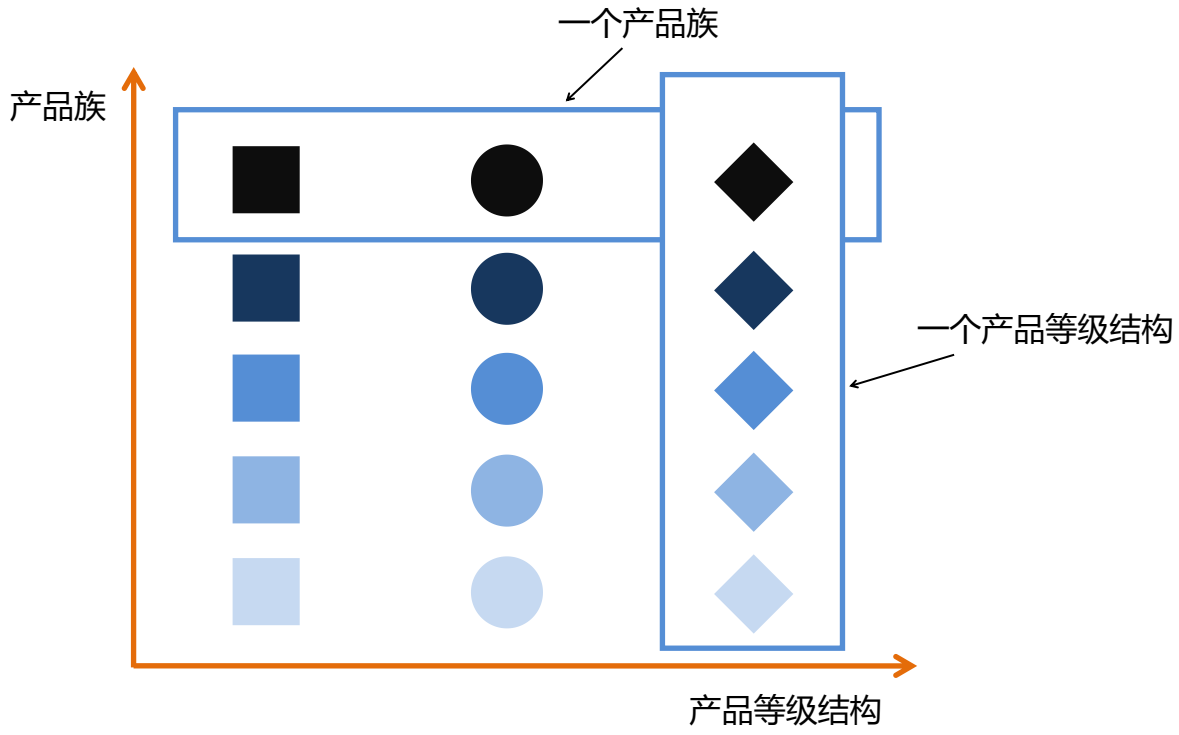
2、增加了系统的抽象性和理解难度。

抽象工厂模式

抽象工厂模式（Abastract Factory Pattern）是指提供一个创建一系列相关或相互依赖

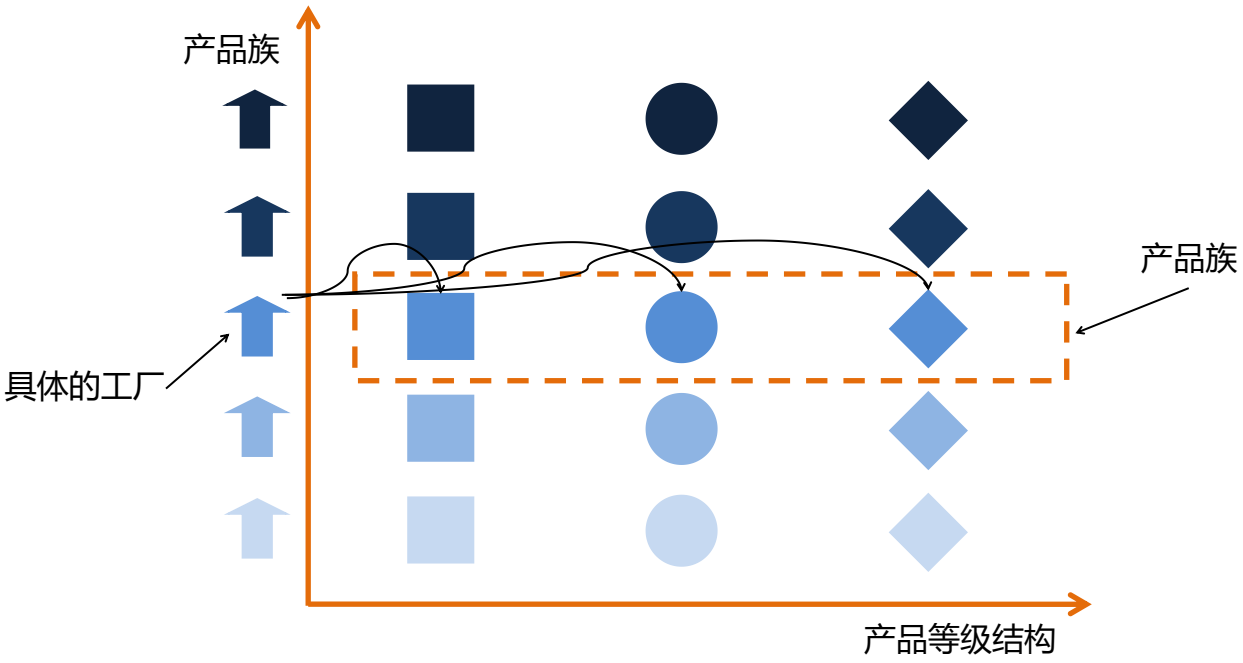
对象的接口，无须指定他们具体的类。客户端（应用层）不依赖于产品类实例如何被创建、实现等细节，强调的是一系列相关的产品对象（属于同一产品族）一起使用创建对象需要大量重复的代码。需要提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

讲解抽象工厂之前，我们要了解两个概念产品等级结构和产品族，看下面的图：



从上图中看出有正方形，圆形和菱形三种图形，相同颜色深浅的就代表同一个产品族，相同形状的代表同一个产品等级结构。同样可以从生活中来举例，比如，美的电器生产多种家用电器。那么上图中，颜色最深的正方形就代表美的洗衣机、颜色最深的圆形代表美的空调、颜色最深的菱形代表美的热水器，颜色最深的一排都属于美的品牌，都是美的电器这个产品族。再看最右侧的菱形，颜色最深的我们指定了代表美的热水器，那么第二排颜色稍微浅一点的菱形，代表海信的热水器。同理，同一产品结构下还有格力热水器，格力空调，格力洗衣机。

再看下面的这张图，最左侧的小房子我们就认为具体的工厂，有美的工厂，有海信工厂，有格力工厂。每个品牌的工厂都生产洗衣机、热水器和空调。



通过上面两张图的对比理解，相信大家对抽象工厂有了非常形象的理解。接下来我们来看一个具体的业务场景而且用代码来实现。还是以课程为例，咕泡学院第三期课程有了新的标准，每个课程不仅要提供课程的录播视频，而且还要提供老师的课堂笔记。相当于现在的业务变更为同一个课程不单纯是一个课程信息，要同时包含录播视频、课堂笔记甚至还要提供源码才能构成一个完整的课程。在产品等级中增加两个产品 IVideo 录播视频和 INote 课堂笔记。

IVideo 接口：

```
public interface IVideo {
    void record();
}
```

INote 接口：

```
public interface INote {
    void edit();
}
```

然后创建一个抽象工厂 CourseFactory 类：

```
import com.gupaoedu.vip.pattern.factory.INote;
import com.gupaoedu.vip.pattern.factory.IVideo;
/**
 * 抽象工厂是用户的主入口
 * 在 Spring 中应用得最为广泛的一种设计模式
 * 易于扩展
 * Created by Tom.
 */
public interface CourseFactory {
    INote createNote();
    IVideo createVideo();
}
```

接下来，创建 Java 产品族，Java 视频 JavaVideo 类:

```
public class JavaVideo implements IVideo {
    public void record() {
        System.out.println("录制 Java 视频");
    }
}
```

扩展产品等级 Java 课堂笔记 JavaNote 类：

```
public class JavaNote implements INote {
    public void edit() {
        System.out.println("编写 Java 笔记");
    }
}
```

创建 Java 产品族的具体工厂 JavaCourseFactory:

```
public class JavaCourseFactory implements CourseFactory {
    public INote createNote() {
        return new JavaNote();
    }
    public IVideo createVideo() {
        return new JavaVideo();
    }
}
```

然后创建 Python 产品，Python 视频 PythonVideo 类：

```
public class PythonVideo implements IVideo {
    public void record() {
        System.out.println("录制 Python 视频");
    }
```

```
}
```

扩展产品等级 Python 课堂笔记 PythonNote 类：

```
public class PythonNote implements INote {
    public void edit() {
        System.out.println("编写 Python 笔记");
    }
}
```

创建 Python 产品族的具体工厂 PythonCourseFactory:

```
public class PythonCourseFactory implements CourseFactory {
    public INote createNote() {
        return new PythonNote();
    }
    public IVideo createVideo() {
        return new PythonVideo();
    }
}
```

来看客户端调用：

```
public static void main(String[] args) {
    JavaCourseFactory factory = new JavaCourseFactory();
    factory.createNote().edit();
    factory.createVideo().record();
}
```

上面的代码完整地描述了两个产品族 Java 课程和 Python 课程，也描述了两个产品等级视频和手记。抽象工厂非常完美清晰地描述这样一层复杂的关系。但是，不知道大家有没有发现，如果我们再继续扩展产品等级，将源码 Source 也加入到课程中，那么我们的代码从抽象工厂，到具体工厂要全部调整，很显然不符合开闭原则。因此抽象工厂也是有缺点的：

1、规定了所有可能被创建的产品集合，产品族中扩展新的产品困难，需要修改抽象工厂的接口。

2、增加了系统的抽象性和理解难度。

但在实际应用中，我们千万不能犯强迫症甚至有洁癖。在实际需求中产品等级结构升级

是非常正常的一件事情。我们可以根据实际情况，只要不是频繁升级，可以不遵循开闭原则。代码每半年升级一次或者每年升级一次又有何不可呢？

利用工厂模式重构的实践案例

还是演示课堂开始的 JDBC 操作案例，我们每次操作是不是都需要重新创建数据库连接，每次创建其实都非常耗费性能，消耗业务调用时间。我们利用工厂模式，将数据库连接预先创建好放到容器中缓存着，在业务调用时就只需现取现用。接下来我们来看这段代码：

Pool 抽象类：

```
package org.jdbc.sqlhelper;

import java.io.IOException;
import java.io.InputStream;
import java.sql.*;
import java.util.Properties;

/**
 * 自定义连接池 getInstance()返回 POOL 唯一实例,第一次调用时将执行构造函数
 * 构造函数 Pool()调用驱动装载 loadDrivers()函数;连接池创建 createPool()函数 loadDrivers()装载驱动
 * createPool()建连接池 getConnection()返回一个连接实例 getConnection(long time)添加时间限制
 * freeConnection(Connection con)将 con 连接实例返回到连接池 getnum()返回空闲连接数
 * getnumActive()返回当前使用的连接数
 *
 * @author Tom
 *
 */

public abstract class Pool {
   public String propertiesName = "connection-INF.properties";

   private static Pool instance = null; // 定义唯一实例

   /**
    * 最大连接数
    */
   protected int maxConnect = 100; // 最大连接数
```

```java
/**
 * 保持连接数
 */
protected int normalConnect = 10; // 保持连接数

/**
 * 驱动字符串
 */
protected String driverName = null; // 驱动字符串

/**
 * 驱动类
 */
protected Driver driver = null; // 驱动变量


/**
 * 私有构造函数,不允许外界访问
 */
protected Pool() {
   try
   {
      init();
      loadDrivers(driverName);
   }catch(Exception e)
   {
      e.printStackTrace();
   }
}

/**
 * 初始化所有从配置文件中读取的成员变量成员变量
 */
private void init() throws IOException {
   InputStream is = Pool.class.getResourceAsStream(propertiesName);
   Properties p = new Properties();
   p.load(is);
   this.driverName = p.getProperty("driverName");
   this.maxConnect = Integer.parseInt(p.getProperty("maxConnect"));
   this.normalConnect = Integer.parseInt(p.getProperty("normalConnect"));
}

/**
 * 装载和注册所有 JDBC 驱动程序
```

```
     * @param dri  接受驱动字符串
     */
    protected void loadDrivers(String dri) {
        String driverClassName = dri;
        try {
            driver = (Driver) Class.forName(driverClassName).newInstance();
            DriverManager.registerDriver(driver);
            System.out.println("成功注册 JDBC 驱动程序" + driverClassName);
        } catch (Exception e) {
            System.out.println("无法注册 JDBC 驱动程序:" + driverClassName + ",错误:" + e);
        }
    }

    /**
     * 创建连接池
     */
    public abstract void createPool();

    /**
     *
     * (单例模式)返回数据库连接池 Pool 的实例
     *
     * @param driverName 数据库驱动字符串
     * @return
     * @throws IOException
     * @throws ClassNotFoundException
     * @throws IllegalAccessException
     * @throws InstantiationException
     */
    public static synchronized Pool getInstance() throws IOException,
            InstantiationException, IllegalAccessException,
            ClassNotFoundException {

        if (instance == null) {
            instance.init();
            instance = (Pool) Class.forName("org.e_book.sqlhelp.Pool")
                    .newInstance();
        }
        return instance;
    }

    /**
     * 获得一个可用的连接,如果没有则创建一个连接,且小于最大连接限制
     * @return
```

```
     */
    public abstract Connection getConnection();

    /**
     * 获得一个连接,有时间限制
     * @param time 设置该连接的持续时间(以毫秒为单位)
     * @return
     */
    public abstract Connection getConnection(long time);

    /**
     * 将连接对象返回给连接池
     * @param con 获得连接对象
     */
    public abstract void freeConnection(Connection con);

    /**
     * 返回当前空闲连接数
     * @return
     */
    public abstract int getnum();

    /**
     * 返回当前工作的连接数
     * @return
     */
    public abstract int getnumActive();

    /**
     * 关闭所有连接,撤销驱动注册(此方法为单例方法)
     */
    protected synchronized void release() {
        // /撤销驱动
        try {
            DriverManager.deregisterDriver(driver);
            System.out.println("撤销 JDBC 驱动程序 " + driver.getClass().getName());
        } catch (SQLException e) {
            System.out
                    .println("无法撤销 JDBC 驱动程序的注册:" + driver.getClass().getName());
        }
    }
}
```

DBConnectionPool 数据库连接池：

```java
package org.jdbc.sqlhelper;

import java.io.IOException;
import java.io.InputStream;
import java.sql.*;
import java.util.*;
import java.util.Date;

/**
 * 数据库链接池管理类
 * @author Tom
 *
 */
public final class DBConnectionPool extends Pool {
   private int checkedOut; //正在使用的连接数
   /**
    * 存放产生的连接对象容器
    */
   private Vector<Connection> freeConnections = new Vector<Connection>(); //存放产生的连接对象容器

   private String passWord = null; // 密码

   private String url = null; // 连接字符串

   private String userName = null; // 用户名

   private static int num = 0;// 空闲连接数

   private static int numActive = 0;// 当前可用的连接数

   private static DBConnectionPool pool = null;// 连接池实例变量

   /**
    * 产生数据连接池
    * @return
    */
   public static synchronized DBConnectionPool getInstance()
   {
      if(pool == null)
      {
         pool = new DBConnectionPool();
      }
      return pool;
   }
```

```
/**
 * 获得一个 数据库连接池的实例
 */
private DBConnectionPool() {
   try
   {
      init();
      for (int i = 0; i < normalConnect; i++) { // 初始 normalConn 个连接
         Connection c = newConnection();
         if (c != null) {
            freeConnections.addElement(c); //往容器中添加一个连接对象
            num++; //记录总连接数
         }
      }
   }catch(Exception e)
   {
      e.printStackTrace();
   }
}
/**
 * 初始化
 * @throws IOException
 */
private void init() throws IOException
{
   InputStream is = DBConnectionPool.class.getResourceAsStream(propertiesName);
   Properties p = new Properties();
   p.load(is);
   this.userName = p.getProperty("userName");
   this.passWord = p.getProperty("passWord");
   this.driverName = p.getProperty("driverName");
   this.url = p.getProperty("url");
   this.driverName = p.getProperty("driverName");
   this.maxConnect = Integer.parseInt(p.getProperty("maxConnect"));
   this.normalConnect = Integer.parseInt(p.getProperty("normalConnect"));
}
/**
 * 如果不再使用某个连接对象时,可调此方法将该对象释放到连接池
 * @param con
 */
public synchronized void freeConnection(Connection con) {
   freeConnections.addElement(con);
   num++;
```

```
        checkedOut--;
        numActive--;
        notifyAll(); //解锁
    }

    /**
     * 创建一个新连接
     * @return
     */
    private Connection newConnection() {
        Connection con = null;
        try {
            if (userName == null) { // 用户,密码都为空
                con = DriverManager.getConnection(url);
            } else {
                con = DriverManager.getConnection(url, userName, passWord);
            }
            System.out.println("连接池创建一个新的连接");
        } catch (SQLException e) {
            System.out.println("无法创建这个 URL 的连接" + url);
            return null;
        }
        return con;
    }

    /**
     * 返回当前空闲连接数
     * @return
     */
    public int getnum() {
        return num;
    }

    /**
     * 返回当前连接数
     * @return
     */
    public int getnumActive() {
        return numActive;
    }


    /**
     * (单例模式)获取一个可用连接
```

```
     * @return
     */
    public synchronized Connection getConnection() {
        Connection con = null;
        if (freeConnections.size() > 0) { // 还有空闲的连接
            num--;
            con = (Connection) freeConnections.firstElement();
            freeConnections.removeElementAt(0);
            try {
                if (con.isClosed()) {
                    System.out.println("从连接池删除一个无效连接");
                    con = getConnection();
                }
            } catch (SQLException e) {
                System.out.println("从连接池删除一个无效连接");
                con = getConnection();
            }
        } else if (maxConnect == 0 || checkedOut < maxConnect) { // 没有空闲连接且当前连接小于最大允许值,最大值为 0 则不限制
            con = newConnection();
        }
        if (con != null) { // 当前连接数加 1
            checkedOut++;
        }
        numActive++;
        return con;
    }

    /**
     * 获取一个连接,并加上等待时间限制,时间为毫秒
     * @param timeout  接受等待时间(以毫秒为单位)
     * @return
     */
    public synchronized Connection getConnection(long timeout) {
        long startTime = new Date().getTime();
        Connection con;
        while ((con = getConnection()) == null) {
            try {
                wait(timeout); //线程等待
            } catch (InterruptedException e) {
            }
            if ((new Date().getTime() - startTime) >= timeout) {
                return null; // 如果超时,则返回
            }
```

```
        }
        return con;
    }

    /**
     * 关闭所有连接
     */
    public synchronized void release() {
        try {
            //将当前连接赋值到 枚举中
            Enumeration allConnections = freeConnections.elements();
            //使用循环关闭所用连接
            while (allConnections.hasMoreElements()) {
                //如果此枚举对象至少还有一个可提供的元素，则返回此枚举的下一个元素
                Connection con = (Connection) allConnections.nextElement();
                try {
                    con.close();
                    num--;
                } catch (SQLException e) {
                    System.out.println("无法关闭连接池中的连接");
                }
            }
            freeConnections.removeAllElements();
            numActive = 0;
        } finally {
            super.release();
        }
    }

    /**
     * 建立连接池
     */
    public void createPool() {

        pool = new DBConnectionPool();
        if (pool != null) {
            System.out.println("创建连接池成功");
        } else {
            System.out.println("创建连接池失败");
        }
    }
}
```