

课程目标

- 1、掌握原型模式和建造者模式的应用场景
- 2、掌握原型模式的浅克隆和深克隆的写法。
- 3、掌握建造者模式的基本写法。
- 4、了解克隆是如何破坏单例的。
- 5、了解原型模式的优、缺点
- 6、掌握建造者模式和工厂模式的区别。

内容定位

- 1、已了解并掌握工厂模式的人群。
- 2、已了解并掌握单例模式。
- 3、听说过原型模式，但不知道应用场景的人群。

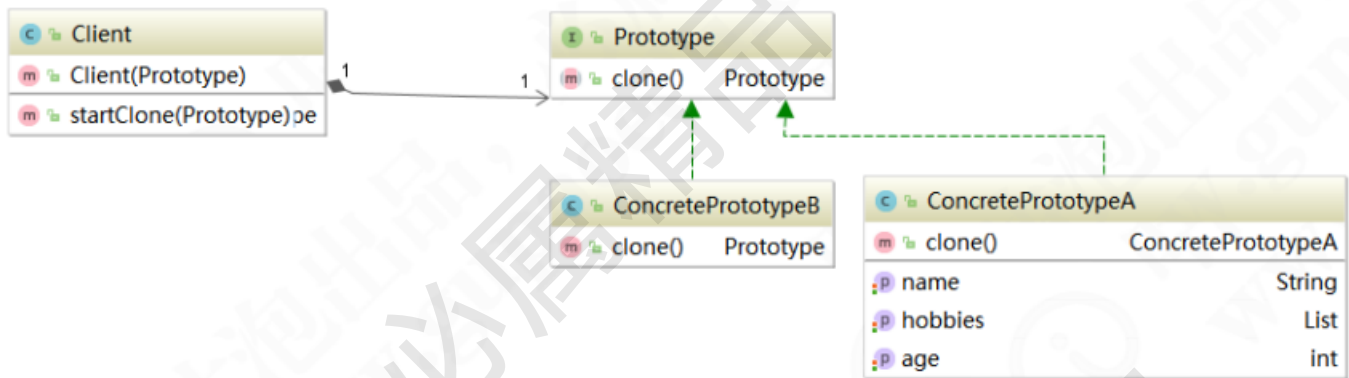
原型模式

原型模式 (Prototype Pattern) 是指原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象，属于创建型模式。

官方原文: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

原型模式的核心在于拷贝原型对象。以系统中已存在的一个对象为原型，直接基于内存二进制流进行拷贝，无需再经历耗时的对象初始化过程（不调用构造函数），性能提升许多。当对象的构建过程比较耗时时，可以利用当前系统中已存在的对象作为原型，对其进行克隆（一般是基于二进制流的复制），

躲避初始化过程，使得新对象的创建时间大大减少。下面，我们来看看原型模式类结构图：



从 UML 图中，我们可以看到，原型模式 主要包含三个角色：

客户(Client)：客户类提出创建对象的请求。

抽象原型(Prototype)：规定拷贝接口。

具体原型 (Concrete Prototype)：被拷贝的对象。

注：对不通过 new 关键字，而是通过对象拷贝来实现创建对象的模式就称作原型模式。

原型模式的应用场景

你一定遇到过大篇幅 getter、setter 赋值的场景。例如这样的代码：

```

import lombok.Data;

@Data
public class ExamPaper{

    // 省略属性设计
    ...

    public ExamPaper copy(){
        ExamPaper examPaper = new ExamPaper();
        //剩余时间
        examPaper.setLeavTime(this.getLeavTime());
        //单位主键
        examPaper.setOrganizationId(this.getOrganizationId());
        //考试主键
        examPaper.setId(this.getId());
        //用户主键
        examPaper.setUserId(this.getUserId());
        //专业
        examPaper.setSpecialtyCode(this.getSpecialtyCode());
        //岗位
        examPaper.setPostionCode(this.getPostionCode());
        //等级
        examPaper.setGradeCode(this.getGradeCode());
    }
  
```

```

//考试开始时间
examPaper.setExamStartTime(this.getExamStartTime());
//考试结束时间
examPaper.setExamEndTime(this.getExamEndTime());
//单选题重要数量
examPaper.setSingleSelectionImpCount(this.getSingleSelectionImpCount());
//多选题重要数量
examPaper.setMultiSelectionImpCount(this.getMultiSelectionImpCount());
//判断题重要数量
examPaper.setJudgementImpCount(this.getJudgementImpCount());
//考试时间
examPaper.setExamTime(this.getExamTime());
//总分
examPaper.setFullScore(this.getFullScore());
//及格分
examPaper.setPassScore(this.getPassScore());
//学员姓名
examPaper.setUserName(this.getUserName());
//分数
examPaper.setScore(this.getScore());

//单选答对数量
examPaper.setSingleOkCount(this.getSingleOkCount());
//多选答对数量
examPaper.setMultiOkCount(this.getMultiOkCount());
//判断答对数量
examPaper.setJudgementOkCount(this.getJudgementOkCount());

return examPaper;
}
}

```

代码非常工整，命名非常规范，注释也写的很全面，其实这就是原型模式的需求场景。但是，大家觉得这样的代码优雅吗？我认为，这样的代码属于纯体力劳动。那原型模式，能帮助我们解决这样的问题。

原型模式主要适用于以下场景：

- 1、类初始化消耗资源较多。
- 2、new 产生的一个对象需要非常繁琐的过程（数据准备、访问权限等）
- 3、构造函数比较复杂。
- 4、循环体中生产大量对象时。

在 Spring 中，原型模式应用得非常广泛。例如 `scope= "prototype"`，在我们经常用的 `JSON.parseObject()` 也是一种原型模式。

原型模式的通用写法

一个标准的原型模式代码，应该是这样设计的。先创建原型 `IPrototype` 接口：

```
public interface IPrototype<T> {
    T clone();
}
```

创建具体需要克隆的对象 ConcretePrototype

```
public class ConcretePrototype implements IPrototype {

    private int age;
    private String name;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

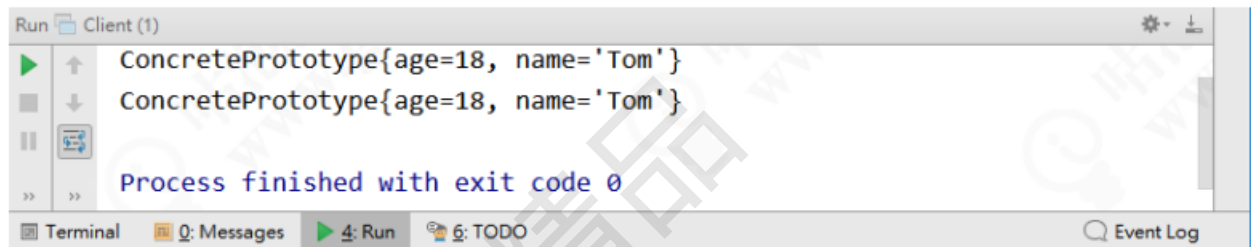
    @Override
    public ConcretePrototype clone() {
        ConcretePrototype concretePrototype = new ConcretePrototype();
        concretePrototype.setAge(this.age);
        concretePrototype.setName(this.name);
        return concretePrototype;
    }

    @Override
    public String toString() {
        return "ConcretePrototype{" +
            "age=" + age +
            ", name='" + name + '\'' +
            '}';
    }
}
```

测试代码：

```
public static void main(String[] args) {
    //创建原型对象
    ConcretePrototype prototype = new ConcretePrototype();
    prototype.setAge(18);
    prototype.setName("Tom");
    System.out.println(prototype);
    //拷贝原型对象
    ConcretePrototype cloneType = prototype.clone();
    System.out.println(cloneType);
}
```

运行结果：



这时候，有小伙伴就问了，原型模式就这么简单吗？对，就是这么简单。在这个简单的场景之下，看上去操作好像变复杂了。但如果有几百个属性需要复制，那我们就可以一劳永逸。但是，上面的复制过程是我们自己完成的，在实际编码中，我们一般不会浪费这样的体力劳动，JDK 已经帮我们实现了一个现成的 API，我们只需要实现 `Cloneable` 接口即可。来改造一下代码，修改 `ConcretePrototype` 类：

```
public class ConcretePrototype implements Cloneable {  
    private int age;  
    private String name;  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public ConcretePrototype clone() {  
        try {  
            return (ConcretePrototype)super.clone();  
        } catch (CloneNotSupportedException e) {  
            e.printStackTrace();  
            return null;  
        }  
    }  
  
    @Override  
    public String toString() {  
        return "ConcretePrototype{" +  
            "age=" + age +  
            ", name='" + name + '\'' +  
            '}';  
    }  
}
```

重新运行，也会得到同样的结果。有了 JDK 的支持再多的属性复制我们也能轻而易举地搞定了。下面我们

再来做一个测试，给 `ConcretePrototype` 增加一个个人爱好的属性 `hobbies`：

```
@Data
public class ConcretePrototype implements Cloneable {

    private int age;
    private String name;
    private List<String> hobbies;

    @Override
    public ConcretePrototype clone() {
        try {
            return (ConcretePrototype)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "ConcretePrototype{" +
            "age=" + age +
            ", name='" + name + '\'' +
            ", hobbies=" + hobbies +
            '}';
    }
}
```

修改客户端测试代码：

```
public static void main(String[] args) {
    //创建原型对象
    ConcretePrototype prototype = new ConcretePrototype();
    prototype.setAge(18);
    prototype.setName("Tom");
    List<String> hobbies = new ArrayList<String>();
    hobbies.add("书法");
    hobbies.add("美术");
    prototype.setHobbies(hobbies);
    System.out.println(prototype);
    //拷贝原型对象
    ConcretePrototype cloneType = prototype.clone();
    cloneType.getHobbies().add("技术控");

    System.out.println("原型对象：" + prototype);
    System.out.println("克隆对象：" + cloneType);
}
```

我们给，复制后的克隆对象新增一项爱好，发现原型对象也发生了变化，这显然不符合我们的预期。因为我们希望克隆出来的对象应该和原型对象是两个独立的对象，不应该再有联系了。从测试结果分析来看，应该是 `hobbies` 共用了一个内存地址，意味着复制的不是值，而是引用的地址。这样的话，如果

我们修改任意一个对象中的属性值，prototype 和 cloneType 的 hobbies 值都会改变。这就是我们常说的浅克隆。只是完整复制了值类型数据，没有赋值引用对象。换言之，所有的引用对象仍然指向原来的对象，显然不是我们想要的结果。那如何解决这个问题呢？下面我们来看深度克隆继续改造。

使用序列化实现深度克隆

在上面的基础上我们继续改造，来看代码，增加一个 deepClone()方法：

```
/**
 * Created by Tom.
 */
@Data
public class ConcretePrototype implements Cloneable,Serializable {

    private int age;
    private String name;
    private List<String> hobbies;

    @Override
    public ConcretePrototype clone() {
        try {
            return (ConcretePrototype)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    public ConcretePrototype deepClone(){
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(bos);
            oos.writeObject(this);

            ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(bis);

            return (ConcretePrototype)ois.readObject();
        } catch (Exception e){
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "ConcretePrototype{" +
            "age=" + age +
            ", name='" + name + '\'' +
            ", hobbies=" + hobbies +
            '}';
    }
}
```

来看客户端调用代码：

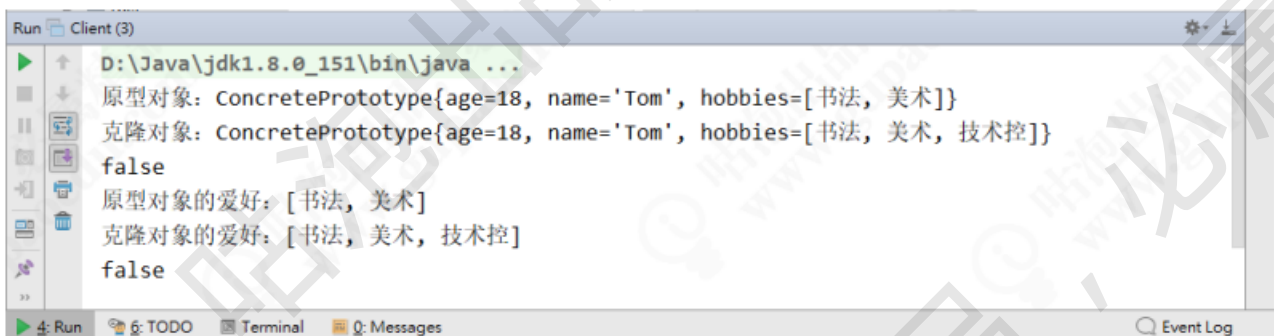
```
public static void main(String[] args) {
    //创建原型对象
    ConcretePrototype prototype = new ConcretePrototype();
    prototype.setAge(18);
    prototype.setName("Tom");
    List<String> hobbies = new ArrayList<String>();
    hobbies.add("书法");
    hobbies.add("美术");
    prototype.setHobbies(hobbies);

    //拷贝原型对象
    ConcretePrototype cloneType = prototype.deepCloneHobbies();
    cloneType.getHobbies().add("技术控");

    System.out.println("原型对象: " + prototype);
    System.out.println("克隆对象: " + cloneType);
    System.out.println(prototype == cloneType);

    System.out.println("原型对象的爱好: " + prototype.getHobbies());
    System.out.println("克隆对象的爱好: " + cloneType.getHobbies());
    System.out.println(prototype.getHobbies() == cloneType.getHobbies());
}
```

运行程序，我们发现得到了我们期望的结果：



```
Run Client (3)
D:\Java\jdk1.8.0_151\bin\java ...
原型对象: ConcretePrototype{age=18, name='Tom', hobbies=[书法, 美术]}
克隆对象: ConcretePrototype{age=18, name='Tom', hobbies=[书法, 美术, 技术控]}
false
原型对象的爱好: [书法, 美术]
克隆对象的爱好: [书法, 美术, 技术控]
false
```

克隆破坏单例模式

如果我们克隆的目标的对象是单例对象，那意味着，深克隆就会破坏单例。实际上防止克隆破坏单例解决思路非常简单，禁止深克隆便可。要么你我们的单例类不实现 `Cloneable` 接口；要么我们重写 `clone()` 方法，在 `clone` 方法中返回单例对象即可，具体代码如下：

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return INSTANCE;
```



```
}
```

原型模式在源码中的应用

先来 JDK 中 Cloneable 接口：

```
public interface Cloneable {
}
```

接口定义还是很简单的，我们找源码其实只需要找到看哪些接口实现了 Cloneable 即可。来看

ArrayList 类的实现。

```
public Object clone() {
    try {
        ArrayList<> v = (ArrayList<>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}
```

我们发现方法中只是将 List 中的元素循环遍历了一遍。这个时候我们再思考一下，是不是这种形式就是深克隆呢？其实用代码验证一下就知道了，继续修改 ConcretePrototype 类，增加一个

deepCloneHobbies()方法：

```
public class ConcretePrototype implements Cloneable,Serializable {

    ...

    public ConcretePrototype deepCloneHobbies(){
        try {
            ConcretePrototype result = (ConcretePrototype)super.clone();
            result.hobbies = (List)((ArrayList)result.hobbies).clone();
            return result;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    ...
}
```

修改客户端代码：

```
public static void main(String[] args) {
    ...

    //拷贝原型对象
    ConcretePrototype cloneType = prototype.deepCloneHobbies();
    ...

}
```

运行也能得到期望的结果。但是这样的代码，其实是硬编码，如果在对象中声明了各种集合类型，那每种情况都需要单独处理。因此，深克隆的写法，一般会直接用序列化来操作。

原型模式的优缺点

优点:

- 1、性能优良，Java 自带的 原型模式 是基于内存二进制流的拷贝，比直接 new 一个对象性能上提升了许多。
- 2、可以使用深克隆方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起来，简化了创建对象的过程，以便在需要的时候使用(例如恢复到历史某一状态)，可辅助实现撤销操作。

缺点:

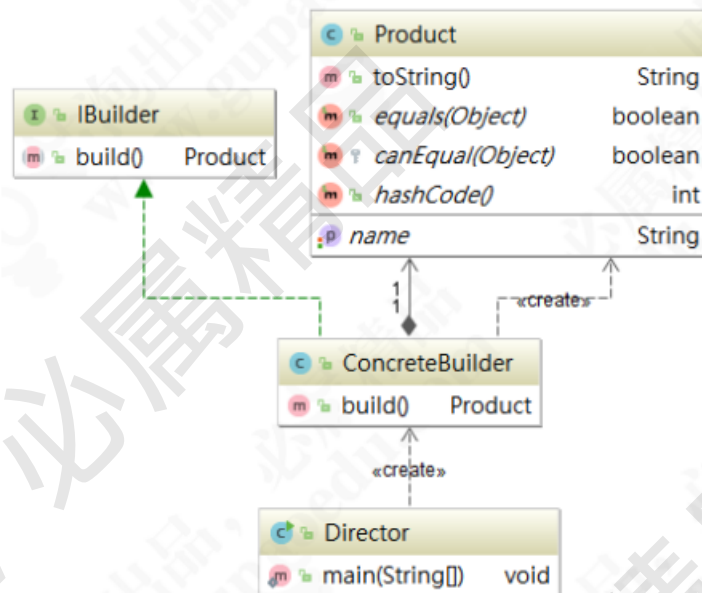
- 1、需要为每一个类配置一个克隆方法。
- 2、克隆方法位于类的内部，当对已有类进行改造的时候，需要修改代码，违反了开闭原则。
- 3、在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来会比较麻烦。因此，深拷贝、浅拷贝需要运用得当。

建造者模式

建造者模式 (Builder Pattern) 是将一个复杂对象的构建过程与它的表示分离，使得同样的构建过程可以创建不同的表示，属于创建型模式。使用建造者模式对于用户而言只需指定需要建造的类型就可以获得对象，建造过程及细节不需要了解。

官方原文: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

建造者模式适用于创建对象需要很多步骤，但是步骤的顺序不一定固定。如果一个对象有非常复杂的内部结构（很多属性），可以将复杂对象的创建和使用进行分离。先来看一下建造者模式的类图：



建造者模式的设计中主要有四个角色：

- 1、产品（Product）：要创建的产品类对象
- 2、建造者抽象（Builder）：建造者的抽象类，规范产品对象的各个组成部分的建造，一般由子类实现具体的建造过程。
- 3、建造者（ConcreteBuilder）：具体的 Builder 类，根据不同的业务逻辑，具体化对象的各个组成部分的创建。
- 4、调用者（Director）：调用具体的建造者，来创建对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

建造者模式的应用场景

建造者模式适用于一个具有较多的零件的复杂产品的创建过程，由于需求的变化，组成这个复杂产品的各个零件经常猛烈变化，但是它们的组合方式却相对稳定。

建造者模式适用于以下几种场景：

- 1 相同的方法，不同的执行顺序，产生不同的结果时
- 2 多个部件或零件，都可以装配到一个对象中，但是产生的结果又不相同。

- 3 产品类非常复杂，或者产品类中的调用顺序不同产生不同的作用。
- 4 当初始化一个对象特别复杂，参数多，而且很多参数都具有默认值时。

建造者模式的基本写法

我们还是以课程为例，一个完整的课程需要由 PPT 课件、回放视频、课堂笔记、课后作业组成，但是这些内容的设置顺序可以随意调整，我们用建造者模式来代入理解一下。首先我们创建一个需要构造的产品类 **Course**：

```
import lombok.Data;

/**
 * Created by Tom.
 */
@Data
public class Course {

    private String name;
    private String ppt;
    private String video;
    private String note;

    private String homework;

    @Override
    public String toString() {
        return "CourseBuilder{" +
            "name='" + name + '\'' +
            ", ppt='" + ppt + '\'' +
            ", video='" + video + '\'' +
            ", note='" + note + '\'' +
            ", homework='" + homework + '\'' +
            '}';
    }
}
```

然后创建建造者类 **CourseBuilder**，将复杂的构造过程封装起来，构造步骤由用户决定：

```
/**
 * Created by Tom.
 */
public class CourseBuilder{

    private Course course = new Course();

    public void addName(String name) {
        course.setName(name);
    }

    public void addPPT(String ppt) {
        course.setPpt(ppt);
    }

    public void addVideo(String video) {
        course.setVideo(video);
    }
}
```

```

    }

    public void addNote(String note) {
        course.setNote(note);
    }

    public void addHomework(String homework) {
        course.setHomework(homework);
    }

    public Course build() {
        return course;
    }
}

```

编写测试类：

```

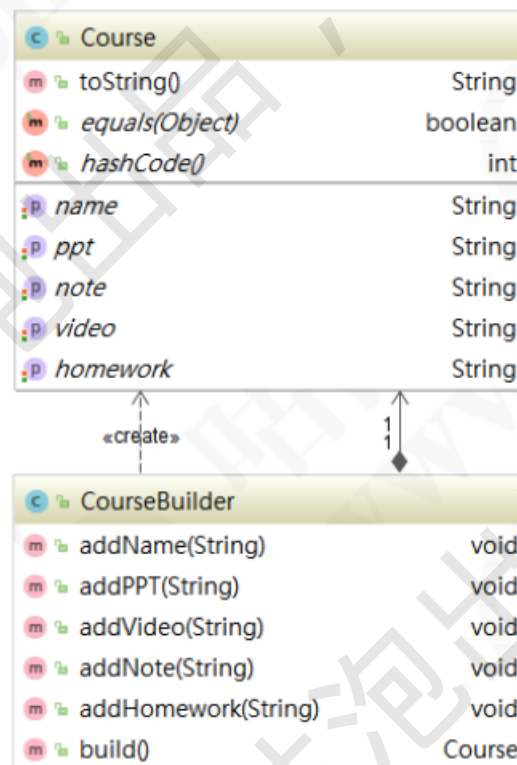
public static void main(String[] args) {
    CourseBuilder builder = new CourseBuilder();

    builder.addName("设计模式");
    builder.addPPT("【PPT 课件】");
    builder.addVideo("【回放视频】");
    builder.addNote("【课堂笔记】");
    builder.addHomework("【课后作业】");

    System.out.println(builder.build());
}

```

来看一下类结构图：



建造者模式的链式写法

在平时的应用中，建造者模式通常是采用链式编程的方式构造对象，下面我们来演示代码，修改 CourseBuilder 类，将 Course 变为 CourseBuilder 的内部类：

```
import lombok.Data;

/**
 * Created by Tom.
 */
public class CourseBuilder {

    @Data
    public class Course {

        private String name;
        private String ppt;
        private String video;
        private String note;

        private String homework;

        @Override
        public String toString() {
            return "CourseBuilder{" +
                "name='" + name + '\'' +
                ", ppt='" + ppt + '\'' +
                ", video='" + video + '\'' +
                ", note='" + note + '\'' +
                ", homework='" + homework + '\'' +
                '}';
        }
    }
}
```

然后，将构造步骤添加进去，每完成一个步骤，都返回 this：

```
import lombok.Data;

/**
 * Created by Tom.
 */
public class CourseBuilder {
    private Course course = new Course();

    public CourseBuilder addName(String name) {
        course.setName(name);
        return this;
    }

    public CourseBuilder addPPT(String ppt) {
        course.setPpt(ppt);
        return this;
    }

    public CourseBuilder addVideo(String video) {
        course.setVideo(video);
        return this;
    }
}
```



```

public CourseBuilder addNote(String note) {
    course.setNote(note);
    return this;
}

public CourseBuilder addHomework(String homework) {
    course.setHomework(homework);
    return this;
}

public Course build() {
    return this.course;
}

@Data
public class Course {

    ...

}
}

```

客户端使用：

```

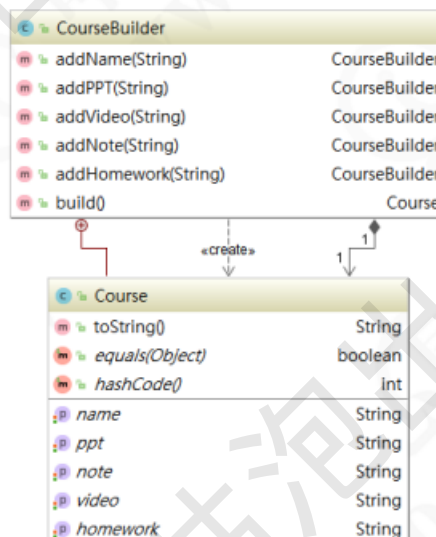
public static void main(String[] args) {
    CourseBuilder builder = new CourseBuilder()
        .addName("设计模式")
        .addPPT("【PPT 课件】")
        .addVideo("【回放视频】")
        .addNote("【课堂笔记】")
        .addHomework("【课后作业】");

    System.out.println(builder.build());
}

```

这样写法是不是很眼熟，好像在哪见过呢？后面我们分析建造者模式在源码中的应用大家就会明白。

接下来，我们再来看一下类图的变化：



建造者模式应用案例

下面我们再来看一个实战案例，这个案例参考了开源框架 JPA 的 SQL 构造模式。是否记得我们在构造 SQL 查询条件的时候，需要根据不同的条件来拼接 SQL 字符串。如果查询条件复杂的时候，我们 SQL 拼接的过程也会变得非常复杂，从而给我们的代码维护带来非常大的困难。因此，我们用建造者类 QueryRuleSqlBuilder 将复杂的构造 SQL 过程进行封装，用 QueryRule 对象专门保存 SQL 查询时的条件，最后根据查询条件，自动生成 SQL 语句。来看代码，先创建 QueryRule 类：

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

/**
 * QueryRule,主要功能用于构造查询条件
 *
 * @author Tom
 */
public final class QueryRule implements Serializable
{
    private static final long serialVersionUID = 1L;
    public static final int ASC_ORDER = 101;
    public static final int DESC_ORDER = 102;
    public static final int LIKE = 1;
    public static final int IN = 2;
    public static final int NOTIN = 3;
    public static final int BETWEEN = 4;
    public static final int EQ = 5;
    public static final int NOTEQ = 6;
    public static final int GT = 7;
    public static final int GE = 8;
    public static final int LT = 9;
    public static final int LE = 10;
    public static final int ISNULL = 11;
    public static final int ISNOTNULL = 12;
    public static final int ISEMPY = 13;
    public static final int ISNOTEMPTY = 14;
    public static final int AND = 201;
    public static final int OR = 202;
    private List<Rule> ruleList = new ArrayList<Rule>();
    private List<QueryRule> queryRuleList = new ArrayList<QueryRule>();
    private String propertyName;

    private QueryRule() {}

    private QueryRule(String propertyName) {
        this.propertyName = propertyName;
    }

    public static QueryRule getInstance() {
        return new QueryRule();
    }

    /**
     * 添加升序规则

```

```

    * @param propertyName
    * @return
    */
    public QueryRule addAscOrder(String propertyName) {
        this.ruleList.add(new Rule(ASC_ORDER, propertyName));
        return this;
    }

    /**
     * 添加降序规则
     * @param propertyName
     * @return
     */
    public QueryRule addDescOrder(String propertyName) {
        this.ruleList.add(new Rule(DESC_ORDER, propertyName));
        return this;
    }

    public QueryRule andIsNull(String propertyName) {
        this.ruleList.add(new Rule(ISNULL, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andIsNotNull(String propertyName) {
        this.ruleList.add(new Rule(ISNOTNULL, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andIsEmpty(String propertyName) {
        this.ruleList.add(new Rule(ISEMPY, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andIsNotEmpty(String propertyName) {
        this.ruleList.add(new Rule(ISNOTEMPTY, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andLike(String propertyName, Object value) {
        this.ruleList.add(new Rule(LIKE, propertyName, new Object[] { value }).setAndOr(AND));
        return this;
    }

    public QueryRule andEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(EQ, propertyName, new Object[] { value }).setAndOr(AND));
        return this;
    }

    public QueryRule andBetween(String propertyName, Object... values) {
        this.ruleList.add(new Rule(BETWEEN, propertyName, values).setAndOr(AND));
        return this;
    }

    public QueryRule andIn(String propertyName, List<Object> values) {
        this.ruleList.add(new Rule(IN, propertyName, new Object[] { values }).setAndOr(AND));
        return this;
    }

    public QueryRule andIn(String propertyName, Object... values) {
        this.ruleList.add(new Rule(IN, propertyName, values).setAndOr(AND));
        return this;
    }
}

```

```

public QueryRule andNotIn(String propertyName, List<Object> values) {
    this.ruleList.add(new Rule(NOTIN, propertyName, new Object[] { values }).setAndOr(AND));
    return this;
}

public QueryRule orNotIn(String propertyName, Object... values) {
    this.ruleList.add(new Rule(NOTIN, propertyName, values).setAndOr(OR));
    return this;
}

public QueryRule andNotEqual(String propertyName, Object value) {
    this.ruleList.add(new Rule(NOTEQ, propertyName, new Object[] { value }).setAndOr(AND));
    return this;
}

public QueryRule andGreaterThan(String propertyName, Object value) {
    this.ruleList.add(new Rule(GT, propertyName, new Object[] { value }).setAndOr(AND));
    return this;
}

public QueryRule andGreaterEqual(String propertyName, Object value) {
    this.ruleList.add(new Rule(GE, propertyName, new Object[] { value }).setAndOr(AND));
    return this;
}

public QueryRule andLessThan(String propertyName, Object value) {
    this.ruleList.add(new Rule(LT, propertyName, new Object[] { value }).setAndOr(AND));
    return this;
}

public QueryRule andLessEqual(String propertyName, Object value) {
    this.ruleList.add(new Rule(LE, propertyName, new Object[] { value }).setAndOr(AND));
    return this;
}

public QueryRule orIsNull(String propertyName) {
    this.ruleList.add(new Rule(ISNULL, propertyName).setAndOr(OR));
    return this;
}

public QueryRule orIsNotNull(String propertyName) {
    this.ruleList.add(new Rule(ISNOTNULL, propertyName).setAndOr(OR));
    return this;
}

public QueryRule orIsEmpty(String propertyName) {
    this.ruleList.add(new Rule(ISEMPY, propertyName).setAndOr(OR));
    return this;
}

public QueryRule orIsNotEmpty(String propertyName) {
    this.ruleList.add(new Rule(ISNOTEMPTY, propertyName).setAndOr(OR));
    return this;
}

public QueryRule orLike(String propertyName, Object value) {
    this.ruleList.add(new Rule(LIKE, propertyName, new Object[] { value }).setAndOr(OR));
    return this;
}

public QueryRule orEqual(String propertyName, Object value) {

```

```

        this.ruleList.add(new Rule(EQ, propertyName, new Object[] { value }).setAndOr(OR));
        return this;
    }

    public QueryRule orBetween(String propertyName, Object... values) {
        this.ruleList.add(new Rule(BETWEEN, propertyName, values).setAndOr(OR));
        return this;
    }

    public QueryRule orIn(String propertyName, List<Object> values) {
        this.ruleList.add(new Rule(IN, propertyName, new Object[] { values }).setAndOr(OR));
        return this;
    }

    public QueryRule orIn(String propertyName, Object... values) {
        this.ruleList.add(new Rule(IN, propertyName, values).setAndOr(OR));
        return this;
    }

    public QueryRule orNotEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(NOTEQ, propertyName, new Object[] { value }).setAndOr(OR));
        return this;
    }

    public QueryRule orGreaterThan(String propertyName, Object value) {
        this.ruleList.add(new Rule(GT, propertyName, new Object[] { value }).setAndOr(OR));
        return this;
    }

    public QueryRule orGreaterEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(GE, propertyName, new Object[] { value }).setAndOr(OR));
        return this;
    }

    public QueryRule orLessThan(String propertyName, Object value) {
        this.ruleList.add(new Rule(LT, propertyName, new Object[] { value }).setAndOr(OR));
        return this;
    }

    public QueryRule orLessEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(LE, propertyName, new Object[] { value }).setAndOr(OR));
        return this;
    }

    public List<Rule> getRuleList() {
        return this.ruleList;
    }

    public List<QueryRule> getQueryRuleList() {
        return this.queryRuleList;
    }

    public String getPropertyName() {
        return this.propertyName;
    }

    protected class Rule implements Serializable {
        private static final long serialVersionUID = 1L;
        private int type; //规则的类型
        private String property_name;
        private Object[] values;
        private int andOr = AND;
    }

```

```

public Rule(int paramInt, String paramString) {
    this.property_name = paramString;
    this.type = paramInt;
}

public Rule(int paramInt, String paramString,
    Object[] paramArrayOfObject) {
    this.property_name = paramString;
    this.values = paramArrayOfObject;
    this.type = paramInt;
}

public Rule setAndOr(int andOr){
    this.andOr = andOr;
    return this;
}

public int getAndOr(){
    return this.andOr;
}

public Object[] getValues() {
    return this.values;
}

public int getType() {
    return this.type;
}

public String getPropertyName() {
    return this.property_name;
}
}
}

```

然后，创建 QueryRuleSqlBuilder 类：

```

package com.gupaoedu.vip.pattern.builder.sql;

/**
 * 根据 QueryRule 自动构建 sql 语句
 * @author Tom
 *
 */
public class QueryRuleSqlBuilder {
    private int CURR_INDEX = 0; //记录参数所在的位置
    private List<String> properties; //保存列名列表
    private List<Object> values; //保存参数值列表
    private List<Order> orders; //保存排序规则列表

    private String whereSql = "";
    private String orderSql = "";
    private Object [] valueArr = new Object[]{};
    private Map<Object,Object> valueMap = new HashMap<Object,Object>();

    /**
     * 或得查询条件
     * @return
     */
    private String getWhereSql(){
        return this.whereSql;
    }
}

```



```

}

/**
 * 获得排序条件
 * @return
 */
private String getOrderSql(){
    return this.orderSql;
}

/**
 * 获得参数值列表
 * @return
 */
public Object [] getValues(){
    return this.valueArr;
}

/**
 * 获取参数列表
 * @return
 */
private Map<Object,Object> getValueMap(){
    return this.valueMap;
}

/**
 * 创建 SQL 构造器
 * @param queryRule
 */
public QueryRuleSqlBuilder(QueryRule queryRule) {
    CURR_INDEX = 0;
    properties = new ArrayList<String>();
    values = new ArrayList<Object>();
    orders = new ArrayList<Order>();
    for (QueryRule.Rule rule : queryRule.getRuleList()) {
        switch (rule.getType()) {
            case QueryRule.BETWEEN:
                processBetween(rule);
                break;
            case QueryRule.EQ:
                processEqual(rule);
                break;
            case QueryRule.LIKE:
                processLike(rule);
                break;
            case QueryRule.NOTEQ:
                processNotEqual(rule);
                break;
            case QueryRule.GT:
                processGreaterThan(rule);
                break;
            case QueryRule.GE:
                processGreaterEqual(rule);
                break;
            case QueryRule.LT:
                processLessThan(rule);
                break;
            case QueryRule.LE:
                processLessEqual(rule);
                break;
            case QueryRule.IN:
                processIN(rule);

```

```

        break;
    case QueryRule.NOTIN:
        processNotIN(rule);
        break;
    case QueryRule.ISNULL:
        processIsNull(rule);
        break;
    case QueryRule.ISNOTNULL:
        processIsNotNull(rule);
        break;
    case QueryRule.ISEMPY:
        processIsEmpty(rule);
        break;
    case QueryRule.ISNOTEMPTY:
        processIsNotEmpty(rule);
        break;
    case QueryRule.ASC_ORDER:
        processOrder(rule);
        break;
    case QueryRule.DESC_ORDER:
        processOrder(rule);
        break;
    default:
        throw new IllegalArgumentException("type " + rule.getType() + " not supported.");
    }
}
//拼装 where 语句
appendWhereSql();
//拼装排序语句
appendOrderSql();
//拼装参数值
appendValues();
}

/**
 * 去掉 order
 *
 * @param sql
 * @return
 */
private String removeOrders(String sql) {
    Pattern p = Pattern.compile("order\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_INSENSITIVE);
    Matcher m = p.matcher(sql);
    StringBuffer sb = new StringBuffer();
    while (m.find()) {
        m.appendReplacement(sb, "");
    }
    m.appendTail(sb);
    return sb.toString();
}

/**
 * 去掉 select
 *
 * @param sql
 * @return
 */
private String removeSelect(String sql) {
    if (sql.toLowerCase().matches("from\\s+")) {
        int beginPos = sql.toLowerCase().indexOf("from");
        return sql.substring(beginPos);
    } else {
        return sql;
    }
}

```

```

    }
}

/**
 * 处理 like
 * @param rule
 */
private void processLike(QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    Object obj = rule.getValues()[0];

    if (obj != null) {
        String value = obj.toString();
        if (!StringUtils.isEmpty(value)) {
            value = value.replace('*', '%');
            obj = value;
        }
    }
    add(rule.getAndOr(), rule.getPropertyName(), "like", "%" + rule.getValues()[0] + "%");
}

/**
 * 处理 between
 * @param rule
 */
private void processBetween(QueryRule.Rule rule) {
    if ((ArrayUtils.isEmpty(rule.getValues()))
        || (rule.getValues().length < 2)) {
        return;
    }
    add(rule.getAndOr(), rule.getPropertyName(), "", "between", rule.getValues()[0], "and");
    add(0, "", "", "", rule.getValues()[1], "");
}

/**
 * 处理 =
 * @param rule
 */
private void processEqual(QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    add(rule.getAndOr(), rule.getPropertyName(), "=", rule.getValues()[0]);
}

/**
 * 处理 <>
 * @param rule
 */
private void processNotEqual(QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    add(rule.getAndOr(), rule.getPropertyName(), "<>", rule.getValues()[0]);
}

/**
 * 处理 >
 * @param rule
 */
private void processGreaterThan(

```

```

        QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    add(rule.getAndOr(),rule.getPropertyName(),">",rule.getValues()[0]);
}

/**
 * 处理>=
 * @param rule
 */
private void processGreaterEqual(
    QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    add(rule.getAndOr(),rule.getPropertyName(),">=",rule.getValues()[0]);
}

/**
 * 处理<
 * @param rule
 */
private void processLessThen(QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    add(rule.getAndOr(),rule.getPropertyName(),"<",rule.getValues()[0]);
}

/**
 * 处理<=
 * @param rule
 */
private void processLessEqual(
    QueryRule.Rule rule) {
    if (ArrayUtils.isEmpty(rule.getValues())) {
        return;
    }
    add(rule.getAndOr(),rule.getPropertyName(),"<=",rule.getValues()[0]);
}

/**
 * 处理 is null
 * @param rule
 */
private void processIsNull(QueryRule.Rule rule) {
    add(rule.getAndOr(),rule.getPropertyName(),"is null",null);
}

/**
 * 处理 is not null
 * @param rule
 */
private void processIsNotNull(QueryRule.Rule rule) {
    add(rule.getAndOr(),rule.getPropertyName(),"is not null",null);
}

/**
 * 处理 <>'
 * @param rule
 */
private void processIsNotEmpty(QueryRule.Rule rule) {

```

```

        add(rule.getAndOr(),rule.getPropertyName(),"<>","'");
    }

    /**
     * 处理 '='
     * @param rule
     */
    private void processIsEmpty(QueryRule.Rule rule) {
        add(rule.getAndOr(),rule.getPropertyName(),"=", "'");
    }

    /**
     * 处理 in 和 not in
     * @param rule
     * @param name
     */
    private void inAndNotIn(QueryRule.Rule rule,String name){
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        if ((rule.getValues().length == 1) && (rule.getValues()[0] != null)
            && (rule.getValues()[0] instanceof List)) {
            List<Object> list = (List) rule.getValues()[0];

            if ((list != null) && (list.size() > 0)){
                for (int i = 0; i < list.size(); i++) {
                    if(i == 0 && i == list.size() - 1){
                        add(rule.getAndOr(),rule.getPropertyName(),"",name + " (" ,list.get(i),")");
                    }else if(i == 0 && i < list.size() - 1){
                        add(rule.getAndOr(),rule.getPropertyName(),"",name + " (" ,list.get(i),"";
                    }
                    if(i > 0 && i < list.size() - 1){
                        add(0,"","","","list.get(i),"";
                    }
                    if(i == list.size() - 1 && i != 0){
                        add(0,"","","","list.get(i),")");
                    }
                }
            }
        } else {
            Object[] list = rule.getValues();
            for (int i = 0; i < list.length; i++) {
                if(i == 0 && i == list.length - 1){
                    add(rule.getAndOr(),rule.getPropertyName(),"",name + " (" ,list[i],")");
                }else if(i == 0 && i < list.length - 1){
                    add(rule.getAndOr(),rule.getPropertyName(),"",name + " (" ,list[i],"";
                }
                if(i > 0 && i < list.length - 1){
                    add(0,"","","","list[i],"";
                }
                if(i == list.length - 1 && i != 0){
                    add(0,"","","","list[i],")");
                }
            }
        }
    }

    /**
     * 处理 not in
     * @param rule
     */
    private void processNotIN(QueryRule.Rule rule){

```

```

        inAndNotIn(rule,"not in");
    }

    /**
     * 处理 in
     * @param rule
     */
    private void processIN(QueryRule.Rule rule) {
        inAndNotIn(rule,"in");
    }

    /**
     * 处理 order by
     * @param rule 查询规则
     */
    private void processOrder(Rule rule) {
        switch (rule.getType()) {
            case QueryRule.ASC_ORDER:
                // propertyName 非空
                if (!StringUtils.isEmpty(rule.getPropertyName())) {
                    orders.add(Order.asc(rule.getPropertyName()));
                }
                break;
            case QueryRule.DESC_ORDER:
                // propertyName 非空
                if (!StringUtils.isEmpty(rule.getPropertyName())) {
                    orders.add(Order.desc(rule.getPropertyName()));
                }
                break;
            default:
                break;
        }
    }
}

/**
 * 加入到 sql 查询规则队列
 * @param andOr and 或者 or
 * @param key 列名
 * @param split 列名与值之间的间隔
 * @param value 值
 */
private void add(int andOr,String key,String split ,Object value){
    add(andOr,key,split,"",value,"");
}

/**
 * 加入到 sql 查询规则队列
 * @param andOr and 或则 or
 * @param key 列名
 * @param split 列名与值之间的间隔
 * @param prefix 值前缀
 * @param value 值
 * @param suffix 值后缀
 */
private void add(int andOr,String key,String split ,String prefix,Object value,String suffix){
    String andOrStr = (0 == andOr ? "" : (QueryRule.AND == andOr ? " and " : " or "));
    properties.add(CURR_INDEX, andOrStr + key + " " + split + prefix + (null != value ? " ? " : " ") + suffix);
    if(null != value){
        values.add(CURR_INDEX,value);
        CURR_INDEX ++;
    }
}
}

```



```

/**
 * 拼装 where 语句
 */
private void appendWhereSql(){
    StringBuffer whereSql = new StringBuffer();
    for (String p : properties) {
        whereSql.append(p);
    }
    this.whereSql = removeSelect(removeOrders(whereSql.toString()));
}

/**
 * 拼装排序语句
 */
private void appendOrderSql(){
    StringBuffer orderSql = new StringBuffer();
    for (int i = 0 ; i < orders.size(); i ++ ) {
        if(i > 0 && i < orders.size()){
            orderSql.append(",");
        }
        orderSql.append(orders.get(i).toString());
    }
    this.orderSql = removeSelect(removeOrders(orderSql.toString()));
}

/**
 * 拼装参数值
 */
private void appendValues(){
    Object [] val = new Object[values.size()];
    for (int i = 0; i < values.size(); i ++ ) {
        val[i] = values.get(i);
        valueMap.put(i, values.get(i));
    }
    this.valueArr = val;
}

public String builder(String tableName){
    String ws = removeFirstAnd(this.getWhereSql());
    String whereSql = ("".equals(ws) ? ws : (" where " + ws));
    String sql = "select * from " + tableName + whereSql;
    Object [] values = this.getValues();
    String orderSql = this.getOrderSql();
    orderSql = (StringUtils.isEmpty(orderSql) ? " " : (" order by " + orderSql));
    sql += orderSql;
    return sql;
}

private String removeFirstAnd(String sql){
    if(StringUtils.isEmpty(sql)){return sql;}
    return sql.trim().toLowerCase().replaceAll("^\\s*and", "") + " ";
}
}

```

创建 Order 类:

```

/**
 * sql 排序组件
 * @author Tom

```

```

*/
public class Order {
    private boolean ascending; //升序还是降序
    private String propertyName; //哪个字段升序, 哪个字段降序

    public String toString() {
        return propertyName + ' ' + (ascending ? "asc" : "desc");
    }

    /**
     * Constructor for Order.
     */
    protected Order(String propertyName, boolean ascending) {
        this.propertyName = propertyName;
        this.ascending = ascending;
    }

    /**
     * Ascending order
     *
     * @param propertyName
     * @return Order
     */
    public static Order asc(String propertyName) {
        return new Order(propertyName, true);
    }

    /**
     * Descending order
     *
     * @param propertyName
     * @return Order
     */
    public static Order desc(String propertyName) {
        return new Order(propertyName, false);
    }
}

```

编写测试代码:

```

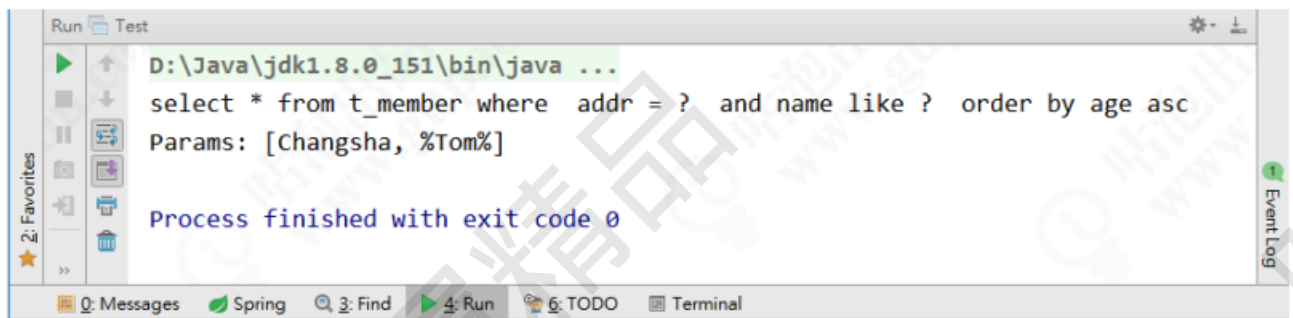
public static void main(String[] args) {
    QueryRule queryRule = QueryRule.getInstance();
    queryRule.addAscOrder("age");
    queryRule.andEqual("addr", "Changsha");
    queryRule.andLike("name", "Tom");
    QueryRuleSqlBuilder builder = new QueryRuleSqlBuilder(queryRule);

    System.out.println(builder.builder("t_member"));

    System.out.println("Params: " + Arrays.toString(builder.getValues()));
}

```

这样一来, 我们的客户端代码就非常清朗, 来看运行结果:



建造者模式在源码中的体现

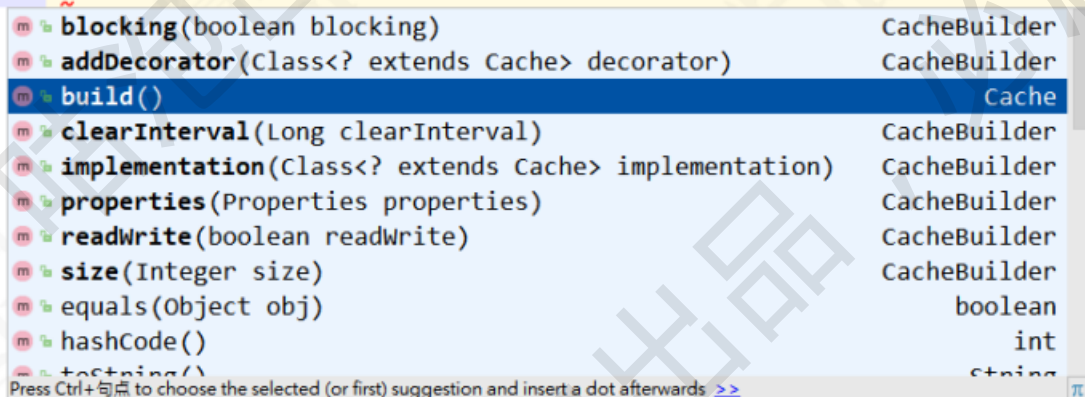
下面来看建造者模式在哪些源码中有应用呢？首先来看 JDK 的 `StringBuilder`，它提供 `append()` 方法，给我们开放构造步骤，最后调用 `toString()` 方法就可以获得一个构造好的完整字符串，源码如下：

```
public final class StringBuilder
extends AbstractStringBuilder
implements java.io.Serializable, CharSequence{
    ...

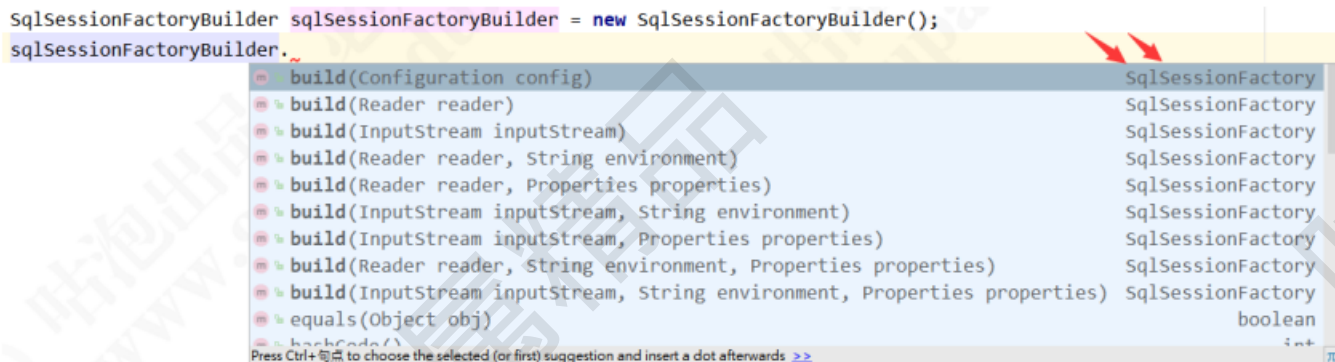
    public StringBuilder append(String str) {
        super.append(str);
        return this;
    }
    ...
}
```

在 MyBatis 中也有体现，比如 `CacheBuilder` 类，如下图：

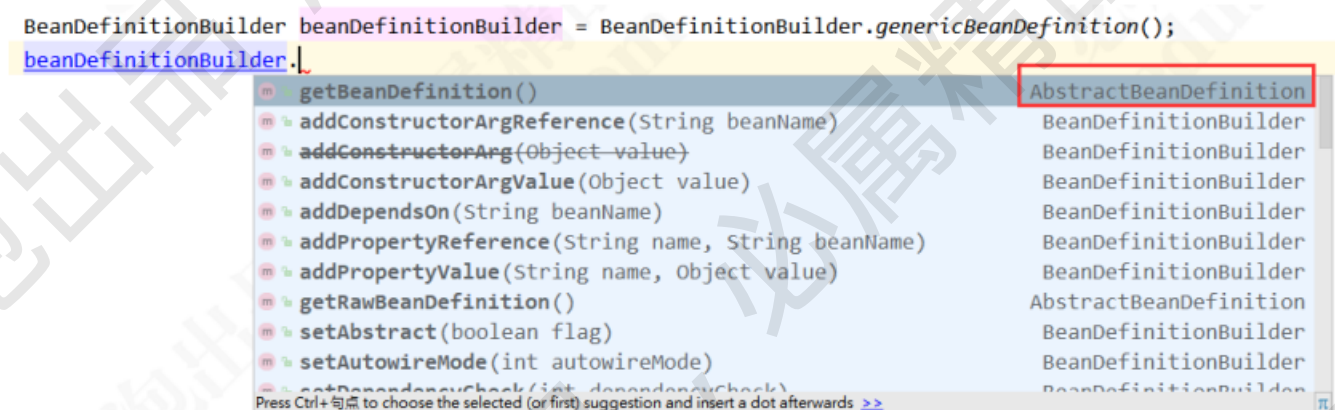
```
CacheBuilder cacheBuilder = new CacheBuilder( id: "" );
cacheBuilder.blocking(false);
cacheBuilder.
```



同样在 MyBatis 中，比如 `SqlSessionFactoryBuilder` 通过调用 `build()` 方法获得的是一个 `SqlSessionFactory` 类，如下图：



当然，在 Spring 中自然也少不了，比如 BeanDefinitionBuilder 通过调用 `getBeanDefinition()` 方法获得一个 BeanDefinition 对象，如下图：



建造者模式的优缺点

建造者模式的优点：

- 1、封装性好，创建和使用分离；
- 2、扩展性好，建造类之间独立、一定程度上解耦。

建造者模式的缺点：

- 1、产生多余的 Builder 对象；
- 2、产品内部发生变化，建造者都要修改，成本较大。

建造者模式和工厂模式的区别

同过前面的学习，我们已经了解建造者模式，那么它和工厂模式有什么区别你？

- 1、建造者模式更加注重方法的调用顺序，工厂模式注重于创建对象。

2、创建对象的力度不同，建造者模式创建复杂的对象，由各种复杂的部件组成，工厂模式创建出来的都一样。

3、关注重点不一样，工厂模式只需要把对象创建出来就可以了，而建造者模式中不仅要创建出这个对象，还要知道这个对象由哪些部件组成。

4、建造者模式根据建造过程中的顺序不一样，最终的对象部件组成也不一样。