46-建造者模式：详解构造函数、set方法、建造者模式三种对象创建方式

上两节课中，我们学习了工厂模式，讲了工厂模式的应用场景，并带你实现了一个简单的DI容器。今天，我们再来学习另外一个比较常用的创建型设计模式，**Builder模式**，中文翻译为**建造者模式**或者**构建者模式**，也有人叫它**生成器模式**。

实际上，建造者模式的原理和代码实现非常简单，掌握起来并不难，难点在于应用场景。比如，你有没有考虑过这样几个问题：直接使用构造函数或者配合set方法就能创建对象，为什么还需要建造者模式来创建呢？建造者模式和工厂模式都可以创建对象，那它们两个的区别在哪里呢？

话不多说，带着上面两个问题，让我们开始今天的学习吧！

为什么需要建造者模式？

在平时的开发中，创建一个对象最常用的方式是，使用new关键字调用类的构造函数来完成。我的问题是，什么情况下这种方式就不适用了，就需要采用建造者模式来创建对象呢？你可以先思考一下，下面我通过一个例子来带你看一下。

假设有这样一道设计面试题：我们需要定义一个资源池配置类ResourcePoolConfig。这里的资源池，你可以简单理解为线程池、连接池、对象池等。在这个资源池配置类中，有以下几个成员变量，也就是可配置项。现在，请你编写代码实现这个ResourcePoolConfig类。

成员变量	解释	是否必填?	默认值
name	资源名称	是	没有
maxTotal	最大总资源数量	否	8
maxIdle	最大空闲资源数量	否	8
minIdle	最小空闲资源数量	否	0

极客时间

只要你稍微有点开发经验，那实现这样一个类对你来说并不是件难事。最常见、最容易想到的实现思路如下代码所示。因为maxTotal、maxIdle、minIdle不是必填变量，所以在创建ResourcePoolConfig对象的时候，我们通过往构造函数中，给这几个参数传递null值，来表示使用默认值。

```
public class ResourcePoolConfig {
  private static final int DEFAULT_MAX_TOTAL = 8;
  private static final int DEFAULT_MAX_IDLE = 8;
  private static final int DEFAULT_MIN_IDLE = 0;

  private String name;
  private int maxTotal = DEFAULT_MAX_TOTAL;
  private int maxIdle = DEFAULT_MAX_IDLE;
  private int minIdle = DEFAULT_MIN_IDLE;

  public ResourcePoolConfig(String name, Integer maxTotal, Integer maxIdle, Integer minIdle) {
    if (StringUtils.isBlank(name)) {
```

```
      throw new IllegalArgumentException("name should not be empty.");
    }
    this.name = name;

    if (maxTotal != null) {
      if (maxTotal <= 0) {
        throw new IllegalArgumentException("maxTotal should be positive.");
      }
      this.maxTotal = maxTotal;
    }

    if (maxIdle != null) {
      if (maxIdle < 0) {
        throw new IllegalArgumentException("maxIdle should not be negative.");
      }
      this.maxIdle = maxIdle;
    }

    if (minIdle != null) {
      if (minIdle < 0) {
        throw new IllegalArgumentException("minIdle should not be negative.");
      }
      this.minIdle = minIdle;
    }
  }
  //...省略getter方法...
}
```

现在，ResourcePoolConfig只有4个可配置项，对应到构造函数中，也只有4个参数，参数的个数不多。但是，如果可配置项逐渐增多，变成了8个、10个，甚至更多。如果继续沿用现在的设计思路，那构造函数的参数列表会变得很长，代码在可读性和易用性上都会变差。在使用构造函数的时候，我们就容易搞错各参数的顺序，传递进错误的参数值，导致非常隐蔽的bug。不仅如此，函数参数太多，写出来的代码，可读性也非常差。

```
// 参数太多，导致可读性差、参数可能传递错误
ResourcePoolConfig config = new ResourcePoolConfig("dbconnectionpool", 16, null, 8, null, false , true, 10,
```

解决这个问题的办法你应该也已经想到了，那就是用set()函数来给成员变量赋值，以替代冗长的构造函数。我们直接看代码，具体如下所示。其中，配置项name是必填的，所以我们把它放到构造函数中设置，强制创建类对象的时候就要填写。其他配置项maxTotal、maxIdle、minIdle都不是必填的，所以我们通过set()函数来设置，让使用者自主选择填写或者不填写。

```
public class ResourcePoolConfig {
  private static final int DEFAULT_MAX_TOTAL = 8;
  private static final int DEFAULT_MAX_IDLE = 8;
  private static final int DEFAULT_MIN_IDLE = 0;

  private String name;
  private int maxTotal = DEFAULT_MAX_TOTAL;
  private int maxIdle = DEFAULT_MAX_IDLE;
  private int minIdle = DEFAULT_MIN_IDLE;
```

```
  public ResourcePoolConfig(String name) {
    if (StringUtils.isBlank(name)) {
      throw new IllegalArgumentException("name should not be empty.");
    }
    this.name = name;
  }

  public void setMaxTotal(int maxTotal) {
    if (maxTotal <= 0) {
      throw new IllegalArgumentException("maxTotal should be positive.");
    }
    this.maxTotal = maxTotal;
  }

  public void setMaxIdle(int maxIdle) {
    if (maxIdle < 0) {
      throw new IllegalArgumentException("maxIdle should not be negative.");
    }
    this.maxIdle = maxIdle;
  }

  public void setMinIdle(int minIdle) {
    if (minIdle < 0) {
      throw new IllegalArgumentException("minIdle should not be negative.");
    }
    this.minIdle = minIdle;
  }
  //...省略getter方法...
}
```

接下来，我们来看新的ResourcePoolConfig类该如何使用。我写了一个示例代码，如下所示。没有了冗长的函数调用和参数列表，代码在可读性和易用性上提高了很多。

```
// ResourcePoolConfig使用举例
ResourcePoolConfig config = new ResourcePoolConfig("dbconnectionpool");
config.setMaxTotal(16);
config.setMaxIdle(8);
```

至此，我们仍然没有用到建造者模式，通过构造函数设置必填项，通过set()方法设置可选配置项，就能实现我们的设计需求。如果我们把问题的难度再加大点，比如，还需要解决下面这三个问题，那现在的设计思路就不能满足了。

- 我们刚刚讲到，name是必填的，所以，我们把它放到构造函数中，强制创建对象的时候就设置。如果必填的配置项有很多，把这些必填配置项都放到构造函数中设置，那构造函数就又会出现参数列表很长的问题。如果我们把必填项也通过set()方法设置，那校验这些必填项是否已经填写的逻辑就无处安放了。
- 除此之外，假设配置项之间有一定的依赖关系，比如，如果用户设置了maxTotal、maxIdle、minIdle其中一个，就必须显式地设置另外两个；或者配置项之间有一定的约束条件，比如，maxIdle和minIdle要小于等于maxTotal。如果我们继续使用现在的设计思路，那这些配置项之间的依赖关系或者约束条件的校验逻辑就无处安放了。
- 如果我们希望ResourcePoolConfig类对象是不可变对象，也就是说，对象在创建好之后，就不能再修改内部的属性值。要实现这个功能，我们就不能在ResourcePoolConfig类中暴露set()方法。

为了解决这些问题，建造者模式就派上用场了。

我们可以把校验逻辑放置到Builder类中，先创建建造者，并且通过set()方法设置建造者的变量值，然后在使用build()方法真正创建对象之前，做集中的校验，校验通过之后才会创建对象。除此之外，我们把ResourcePoolConfig的构造函数为了private私有权限，只能通过建造者来创建ResourcePoolConfig类对象，并且ResourcePoolConfig没有提供任何set()方法，这样创建出来的对象就是不可变对象了。

我们用建造者模式重新实现了上面的需求，具体的代码如下所示：

```
public class ResourcePoolConfig {
  private String name;
  private int maxTotal;
  private int maxIdle;
  private int minIdle;

  private ResourcePoolConfig(Builder builder) {
    this.name = builder.name;
    this.maxTotal = builder.maxTotal;
    this.maxIdle = builder.maxIdle;
    this.minIdle = builder.minIdle;
  }
  //...省略getter方法...

  //我们将Builder类被设计成了ResourcePoolConfig的内部类。
  //我们也可以将Builder类设计成独立的非内部类ResourcePoolConfigBuilder。
  public static class Builder {
    private static final int DEFAULT_MAX_TOTAL = 8;
    private static final int DEFAULT_MAX_IDLE = 8;
    private static final int DEFAULT_MIN_IDLE = 0;

    private String name;
    private int maxTotal = DEFAULT_MAX_TOTAL;
    private int maxIdle = DEFAULT_MAX_IDLE;
    private int minIdle = DEFAULT_MIN_IDLE;

    public ResourcePoolConfig build() {
      // 校验逻辑放到这里来做，包括必填项校验、依赖关系校验、约束条件校验等
      if (StringUtils.isBlank(name)) {
        throw new IllegalArgumentException("...");
      }
      if (maxIdle > maxTotal) {
        throw new IllegalArgumentException("...");
      }
      if (minIdle > maxTotal || minIdle > maxIdle) {
        throw new IllegalArgumentException("...");
      }

      return new ResourcePoolConfig(this);
    }

    public Builder setName(String name) {
      if (StringUtils.isBlank(name)) {
        throw new IllegalArgumentException("...");
      }
      this.name = name;
      return this;
    }

    public Builder setMaxTotal(int maxTotal) {
```

```
      if (maxTotal <= 0) {
        throw new IllegalArgumentException("...");
      }
      this.maxTotal = maxTotal;
      return this;
    }

    public Builder setMaxIdle(int maxIdle) {
      if (maxIdle < 0) {
        throw new IllegalArgumentException("...");
      }
      this.maxIdle = maxIdle;
      return this;
    }

    public Builder setMinIdle(int minIdle) {
      if (minIdle < 0) {
        throw new IllegalArgumentException("...");
      }
      this.minIdle = minIdle;
      return this;
    }
  }
}

// 这段代码会抛出IllegalArgumentException，因为minIdle>maxIdle
ResourcePoolConfig config = new ResourcePoolConfig.Builder()
        .setName("dbconnectionpool")
        .setMaxTotal(16)
        .setMaxIdle(10)
        .setMinIdle(12)
        .build();
```

实际上，使用建造者模式创建对象，还能避免对象存在无效状态。我再举个例子解释一下。比如我们定义了一个长方形类，如果不使用建造者模式，采用先创建后set的方式，那就会导致在第一个set之后，对象处于无效状态。具体代码如下所示：

```
Rectangle r = new Rectange(); // r is invalid
r.setWidth(2); // r is invalid
r.setHeight(3); // r is valid
```

为了避免这种无效状态的存在，我们就需要使用构造函数一次性初始化好所有的成员变量。如果构造函数参数过多，我们就需要考虑使用建造者模式，先设置建造者的变量，然后再一次性地创建对象，让对象一直处于有效状态。

实际上，如果我们并不是很关心对象是否有短暂的无效状态，也不是太在意对象是否是可变的。比如，对象只是用来映射数据库读出来的数据，那我们直接暴露set()方法来设置类的成员变量值是完全没问题的。而且，使用建造者模式来构建对象，代码实际上是有点重复的，ResourcePoolConfig类中的成员变量，要在Builder类中重新再定义一遍。

与工厂模式有何区别?

从上面的讲解中，我们可以看出，建造者模式是让建造者类来负责对象的创建工作，上一节课中讲到的工厂模式，是由工厂类来负责对象创建的工作，那它们之间有什么区别呢?

实际上，工厂模式是用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。建造者模式是用来创建一种类型的复杂对象，通过设置不同的可选参数，“定制化”地创建不同的对象。

网上有一个经典的例子很好地解释了两者的区别。

顾客走进一家餐馆点餐，我们利用工厂模式，根据用户不同的选择，来制作不同的食物，比如披萨、汉堡、沙拉。对于披萨来说，用户又有各种配料可以定制，比如奶酪、西红柿、起司，我们通过建造者模式根据用户选择的不同配料来制作披萨。

实际上，我们也不要太学院派，非得把工厂模式、建造者模式分得那么清楚，我们需要知道的是，每个模式为什么这么设计，能解决什么问题。**只有了解了这些最本质的东西，我们才能不生搬硬套，才能灵活应用，甚至可以混用各种模式创造出新的模式，来解决特定场景的问题。**

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

建造者模式的原理和实现比较简单，重点是掌握应用场景，避免过度使用。

如果一个类中有很多属性，为了避免构造函数的参数列表过长，影响代码的可读性和易用性，我们可以通过构造函数配合set()方法来解决。但是，如果存在下面情况中的任意一种，我们就要考虑使用建造者模式了。

- 我们把类的必填属性放到构造函数中，强制创建对象的时候就设置。如果必填的属性有很多，把这些必填属性都放到构造函数中设置，那构造函数就又会出现参数列表很长的问题。如果我们把必填属性通过set()方法设置，那校验这些必填属性是否已经填写的逻辑就无处安放了。
- 如果类的属性之间有一定的依赖关系或者约束条件，我们继续使用构造函数配合set()方法的设计思路，那这些依赖关系或约束条件的校验逻辑就无处安放了。
- 如果我们希望创建不可变对象，也就是说，对象在创建好之后，就不能再修改内部的属性值，要实现这个功能，我们就不能在类中暴露set()方法。构造函数配合set()方法来设置属性值的方式就不适用了。

除此之外，在今天的讲解中，我们还对比了工厂模式和建造者模式的区别。工厂模式是用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。建造者模式是用来创建一种类型的复杂对象，可以通过设置不同的可选参数，“定制化”地创建不同的对象。

课堂讨论

在下面的ConstructorArg类中，当isRef为true的时候，arg表示String类型的refBeanId，type不需要设置；当isRef为false的时候，arg、type都需要设置。请根据这个需求，完善ConstructorArg类。

```
public class ConstructorArg {
```

```
    private boolean isRef;
    private Class type;
    private Object arg;
    // TODO: 待完善...
  }
```

欢迎留言和我分享你的想法，如果有收获，你也可以把这篇文章分享给你的朋友。

精选留言：

- 相逢是缘 2020-02-17 17:40:18
  打卡
  一、使用场景：
  1）类的构造函数必填属性很多，通过set设置，没有办法校验必填属性
  2）如果类的属性之间有一定的依赖关系，构造函数配合set方式，无法进行依赖关系和约束条件校验
  3）需要创建不可变对象，不能暴露set方法。
  （前提是需要传递很多的属性，如果属性很少，可以不需要建造者模式）
  二、实现方式：
  把构造函数定义为private，定义public static class Builder 内部类，通过Builder 类的set方法设置属性，调用build方法创建对象。

  三、和工厂模式的区别：
  1）工厂模式：创建不同的同一类型对象（集成同一个父类或是接口的一组子类），由给定的参数来创建哪种类型的对象；
  2）建造者模式：创建一种类型的复杂对象，通过很多可设置参数，“定制化”的创建对象 [3赞]

- 黄林晴 2020-02-17 22:12:03
  打卡~
  最近半年用的最多的就是Builder模式了 [2赞]

- webmin 2020-02-17 13:53:31
  public class ConstructorArg {
  private boolean isRef;
  private Class type;
  private Object arg;

  public boolean isRef() {
  return isRef;
  }

  public Class getType() {
  return type;
  }

  public Object getArg() {
  return arg;
  }

  private ConstructorArg(Builder builder) {
  this.isRef = builder.isRef;

```
this.type = builder.type;
this.arg = builder.arg;
}

public static class Builder {
private boolean isRef;
private Class type;
private Object arg;

public ConstructorArg build() {
if(isRef && type != null) {
throw new IllegalArgumentException("...");
}

if (!isRef && type == null) {
throw new IllegalArgumentException("...");
}

if (this.isRef && (arg != null && arg.getClass() != String.class)) {
throw new IllegalArgumentException("...");
}

if (!this.isRef && arg == null) {
throw new IllegalArgumentException("...");
}

return new ConstructorArg(this);
}

public Builder setRef(boolean ref) {
if(ref && this.type != null) {
throw new IllegalArgumentException("...");
}
this.isRef = ref;
return this;
}

public Builder setType(Class type) {
if (this.isRef || type == null) {
throw new IllegalArgumentException("...");
}
this.type = type;
return this;
}

public Builder setArg(Object arg) {
if (this.isRef && (arg != null && arg.getClass() != String.class)) {
throw new IllegalArgumentException("...");
}
```

if (!this.isRef && arg == null) {
throw new IllegalArgumentException("...");
}
this.arg = arg;
return this;
}
}
} [2赞]

- 王涛 2020-02-17 12:05:21
最近，在学习本专栏的过程中。
逐渐体会到“知其然，更知其所以然”的感觉
[2赞]

- 大牛凯 2020-02-17 03:01:34
感觉课后思考题的arguments逻辑判断相对复杂，再加上想尝试一下建造者模式，所以抛砖引玉，给一个Python的尝试，有一个困惑是对于Python这种语言，如何避免对用户暴露在调用builder之前build方法之前初始化的ConstructorArg对象？

class ConstructorArg(object):
def __init__(self, isRef): #如何避免暴露这个实例对象？
self.isRef = isRef
self.type = None
self.arg = None

def __repr__(self):
string = f"isRef: {self.isRef}\n" + \
f"type: {self.type}\n" + \
f"arg: {self.arg}\n"
return string

class ConstructorArgBuilder(object):
def __init__(self, isRef):
self.constructorArg = ConstructorArg(isRef)

def addType(self, typeObj):
self.constructorArg.type = typeObj
return self

def addArg(self, argObj):
self.constructorArg.arg = argObj
return self

def build(self):
if self.constructorArg.isRef:
if self.constructorArg.type is None:
raise Exception("type cannot be None when isRef")
elif not isinstance(self.constructorArg.type, str):
raise Exception("type must be string when isRef")
elif not self.constructorArg.isRef:

```
if self.constructorArg.type is None:
raise Exception("type cannot be None when not isRef")
if self.constructorArg.arg is None:
raise Exception("arg cannot be None when not isRef")
return self.constructorArg [2赞]
```

- 啦啦啦 2020-02-17 22:11:42

用php实现了一个

```
<?php
class ResourcePoolConfig{
private $maxTotal;
private $maxIdle;
private $minIdle;
public function __construct(build $build)
{
$this->maxTotal = $build->maxTotal;
$this->maxIdle = $build->maxIdle;
$this->minIdle = $build->minIdle;
echo 'maxTotal'.$this->maxTotal;
echo '<br />';
echo 'maxIdle'.$this->maxIdle;
echo '<br />';
echo 'minIdle'.$this->minIdle;
echo '<br />';
}
}

class Builder{
public $maxTotal;
public $maxIdle;
public $minIdle;
public function noodleValidate(){
if($this->maxIdle>$this->maxTotal){
throw new Exception("maxIdle抛出异常");
}
if($this->minIdle>$this->maxTotal){
throw new Exception("minIdle抛出异常");
}
}

public function setMaxTotal($value=20){
$this->maxTotal = $value;
return $this;
}

public function setMaxIdle($value=10){
$this->maxIdle = $value;
return $this;
}
```

```
public function setMinIdle($value=10){
$this->minIdle = $value;
return $this;
}
}

$b = new Builder();
$b->setMaxTotal(5)->setMaxIdle()->setMinIdle()->noodleValidate();
new ResourcePoolConfig($b); [1赞]
```

- PHP是世界上最好的需要 2020-02-17 16:07:27
  总感觉只是把校验逻辑挪到了bulder 里。没直观的感觉到builder 的有点呢。老师~ [1赞]

- Ken张云忠 2020-02-17 11:41:49

```
public class ConstructorArg {
private boolean isRef;
private Class type;
private Object arg;
public ConstructorArg(ConstructorArgBuilder builder) {
this.isRef = builder.isRef;
this.type = builder.type;
this.arg = builder.arg;
}
public static class ConstructorArgBuilder {
private boolean isRef;
private Class type;
private Object arg;
public ConstructorArg builder() {
if (arg == null) {
throw new IllegalArgumentException("arg should not be null");
}
if (isRef == true) {
if (arg instanceof String) {
type = String.class;
} else {
throw new IllegalArgumentException("when isRef is true,arg should be String type");
}
} else {
type = arg.getClass();
}
return new ConstructorArg(this);
}
public ConstructorArgBuilder(boolean isRef) {
this.isRef = isRef;
}
public ConstructorArgBuilder setArg(Object arg) {
this.arg = arg;
return this;
}
}
```

```
}
// use case
ConstructorArg constructorArg = new ConstructorArgBuilder(true).setArg(0).builder();
ConstructorArg constructorArg1 = new ConstructorArgBuilder(false).setArg(1).builder(); [1赞]
```

- Algo 2020-02-17 11:34:42

```
package com.acacia.leetcode.huahua.questions;

public class ConstructorArg {
private boolean isRef;
private Class type;
private Object arg;

private ConstructorArg(Builder builder){
this.isRef = builder.isRef;
this.type = builder.type;
this.arg = builder.arg;
}

public static class Builder{
private final static Class TYPE_STRING = String.class;
private final static Object ARG_STRING = "String";

private boolean isRef;
private Class type;
private Object arg;

public ConstructorArg build(){
if (isRef == true){
this.type = TYPE_STRING;
this.arg = ARG_STRING;
return new ConstructorArg(this);
}else {
return new ConstructorArg(this);
}
}

public Builder setRef(boolean ref) {
this.isRef = ref;
return this;
}

public Builder setType(Class type) {
// if (!type.isMemberClass()){
// throw new IllegalArgumentException("failed class name");
// }
this.type = type;
return this;
}
```

```
public Builder setArg(Object arg) {
if (arg == null){
throw new IllegalArgumentException("can not be null");
}
this.arg = arg;
return this;
}
}

public static void main(String[] args) {
ConstructorArg arg = new ConstructorArg.Builder().setRef(true).build();
System.out.println(arg.type);
System.out.println(arg.arg);
ConstructorArg arg1 = new ConstructorArg.Builder().setArg(false).setType(Integer.class).setArg(2).bui
ld();
System.out.println(arg1.type);
System.out.println(arg1.arg);
}
}
```

[1赞]

- cricket1981 2020-02-17 10:45:21

```
public class ConstructorArg {

private boolean isRef;
private Class type;
private Object arg;

private ConstructorArg(Builder builder) {
this.isRef = builder.isRef;
this.type = builder.type;
this.arg = builder.arg;
}

public boolean getIsRef() {
return isRef;
}

public Class getType() {
return type;
}

public Object getArg() {
return arg;
}

public static class Builder {
```

```
private boolean isRef;
private Class type;
private Object arg;

public ConstructorArg build() {
if (arg == null) {
throw new IllegalArgumentException("The value of parameter `arg` should be provided.");
}
if (!this.isRef && type == null) {
throw new IllegalArgumentException("The value of parameter `type` should be provided if `isRef`
== false.");
}
return new ConstructorArg(this);
}

public Builder setRef(boolean isRef) {
this.isRef = isRef;
return this;
}

public Builder setType(Class type) {
this.type = type;
return this;
}

public Builder setArg(Object arg) {
this.arg = arg;
return this;
}
}

}
```

用法：

```
ConstructorArg constructorArg1 = new ConstructorArg.Builder()
.setArg("rateCounter")
.setRef(true)
.build();

ConstructorArg constructorArg2 = new ConstructorArg.Builder()
.setArg("127.0.0.1")
.setType(String.class)
.setRef(false)
.build(); [1赞]
```

- 小晏子 2020-02-17 10:08:56

课后讨论题：

```
public class ConstructorArg {
```

```
private boolean isRef;
private Class type;
private Object arg;
// TODO: 待完善...

private ConstructorArg(Builder builder) {
this.isRef = builder.isRef;
this.type = builder.type;
this.arg = builder.arg;
}

public static class Builder {
private boolean isRef;
private Class type;
private Object arg;

public ConstructorArg build() {
if (!isRef) {
if (type == null || arg == null){
throw new IllegalArgumentException("type and arg must be set when isRef is false")
}
} else {
if (!(arg instanceof String)) {
throw new IllegalArgumentException("arg must be a String instance when isRef is true")
}
}
return new ConstructorArg(this);
}
}

public Builder setIsRef(boolean isRef){
this.isRef = isRef;
return this;
}

public Builder setType(Class type) {
this.type = type;
return this;
}

public Builder setObject(Object arg) {
this.arg = arg;
return this;
}
}

//use case
ConstructorArg ca = ConstructorArg.Builder()
.setIfRef(false)
.setType(Integer)
```

.setObject(2)
.build(); [1赞]

- Frank 2020-02-17 09:53:11
  当在开发中需要创建一个具体的对象，如果必填属性很多，属性存在复杂的校验和属性之间存在依赖关系，可以使用建造者模式来避免类的构造函数参数列表过长，导致可读性差，不易使用的问题。建造者模式可以将类的构造方法私有，不提供类的setter方法，因此可以创建一个不变的对象。同时在某些场景下将属性构造好可以解决类的无效状态。
  使用思路：把校验逻辑放置到 Builder 类中，先创建建造者，并且通过 set() 方法设置建造者的变量值，然后在使用 build() 方法真正创建对象之前，做集中的校验，校验通过之后才会创建对象。除此之外，我们把主类的构造函数为了private 私有权限，只能通过建造者来创建 类对象，并且主类没有提供任何 set() 方法，这样创建出来的对象就是不可变对象了。 [1赞]

- javaadu 2020-02-17 09:37:08
  课堂讨论题：

```
/**
* 在下面的 ConstructorArg 类中，
* 当 isRef 为 true 的时候，arg 表示 String 类型的 refBeanId，type 不需要设置；
* 当 isRef 为 false 的时候，arg、type 都需要设置
*
* @author javaadu
*/
public class ConstructorArg {
private boolean isRef;
private Class type;
private Object arg;

private ConstructorArg(Builder builder) {
this.isRef = builder.isRef;
this.type = builder.type;
this.arg = builder.arg;
}

public static class Builder {
private boolean isRef;
private Class type;
private Object arg;

public ConstructorArg build() {
if (arg == null) {
throw new IllegalArgumentException("arg必须设置");
}
if (isRef) {
if (!(arg instanceof String)) {
throw new IllegalArgumentException("arg必须为String类型的对象");
}
} else {
if (type == null) {
throw new IllegalArgumentException("arg必须设置");
```

```
}
}

return new ConstructorArg(this)
}

public Builder setRef(boolean ref) {
isRef = ref;
return this;
}

public Builder setArg(Object arg) {
this.arg = arg;
return this;
}

public Builder setType(Class type) {
this.type = type;
return this;
}
}
} [1赞]
```

- Jeff.Smile 2020-02-17 07:03:51
  建造者模式核心代码:
  ```
  ResourcePoolConfig config = new ResourcePoolConfig.Builder()
  .setName("dbconnectionpool")
  .setMaxTotal(16)
  .setMaxIdle(10)
  .setMinIdle(12)
  .build(); [1赞]
  ```

- 朱晋君 2020-02-18 10:10:07
  课后思考题
  ```
  public class ConstructorArg {

  private boolean isRef;
  private Class type;
  private Object arg;

  private ConstructorArg(Builder builder){
  this.isRef = builder.isRef;
  this.type = builder.type;
  this.arg = builder.arg;
  }

  public static class Builder{

  private boolean isRef;
  private Class type;
  ```

```
private Object arg;

public Builder(boolean isRef){
this.isRef = isRef;
}

public ConstructorArg build(){
return new ConstructorArg(this);
}

public Builder setType(Class type) {
if (!isRef && null == type){
throw new IllegalArgumentException("type must be not null");
}
this.type = type;
return this;
}

public Builder setArg(Object arg) {
if (isRef && !(arg instanceof String)){
throw new IllegalArgumentException("arg must be type of String");
}

if (!isRef && null == arg){
throw new IllegalArgumentException("arg must be not null");
}
this.arg = arg;
return this;
}
}

public static void main(String args[]){
ConstructorArg constructorArg = new ConstructorArg.Builder(true).setArg("123").build();
}
}
```