86-开源实战四（下）：总结Spring框架用到的11种设计模式

上一节课，我们讲解了Spring中支持扩展功能的两种设计模式：观察者模式和模板模式。这两种模式能够帮助我们创建扩展点，让框架的使用者在不修改源码的情况下，基于扩展点定制化框架功能。

实际上，Spring框架中用到的设计模式非常多，不下十几种。我们今天就总结罗列一下它们。限于篇幅，我不可能对每种设计模式都进行非常详细的讲解。有些前面已经讲过的或者比较简单的，我就点到为止。如果有什么不是很懂的地方，你可以通过阅读源码，查阅之前的理论讲解，自己去搞定它。如果一直跟着我的课程学习，相信你现在已经具备这样的学习能力。

话不多说，让我们正式开始今天的学习吧！

适配器模式在Spring中的应用

在Spring MVC中，定义一个Controller最常用的方式是，通过@Controller注解来标记某个类是Controller类，通过@RequesMapping注解来标记函数对应的URL。不过，定义一个Controller远不止这一种方法。我们还可以通过让类实现Controller接口或者Servlet接口，来定义一个Controller。针对这三种定义方式，我写了三段示例代码，如下所示：

```
// 方法一：通过@Controller、@RequestMapping来定义
@Controller
public class DemoController {
    @RequestMapping("/employname")
    public ModelAndView getEmployeeName() {
        ModelAndView model = new ModelAndView("Greeting");
        model.addObject("message", "Dinesh");
        return model;
    }
}

// 方法二：实现Controller接口 + xml配置文件:配置DemoController与URL的对应关系
public class DemoController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) throws Exception {
        ModelAndView model = new ModelAndView("Greeting");
        model.addObject("message", "Dinesh Madhwal");
        return model;
    }
}

// 方法三：实现Servlet接口 + xml配置文件:配置DemoController类与URL的对应关系
public class DemoServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOExcepti
    this.doPost(req, resp);
  }

  @Override
  protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOExcept
    resp.getWriter().write("Hello World.");
  }
}
```

在应用启动的时候，Spring容器会加载这些Controller类，并且解析出URL对应的处理函数，封装成

Handler对象，存储到HandlerMapping对象中。当有请求到来的时候，DispatcherServlet从HanderMapping中，查找请求URL对应的Handler，然后调用执行Handler对应的函数代码，最后将执行结果返回给客户端。

但是，不同方式定义的Controller，其函数的定义（函数名、入参、返回值等）是不统一的。如上示例代码所示，方法一中的函数的定义很随意、不固定，方法二中的函数定义是handleRequest()、方法三中的函数定义是service()（看似是定义了doGet()、doPost()，实际上，这里用到了模板模式，Servlet中的service()调用了doGet()或doPost()方法，DispatcherServlet调用的是service()方法）。DispatcherServlet需要根据不同类型的Controller，调用不同的函数。下面是具体的伪代码：

```
Handler handler = handlerMapping.get(URL);
if (handler instanceof Controller) {
  ((Controller)handler).handleRequest(...);
} else if (handler instanceof Servlet) {
  ((Servlet)handler).service(...);
} else if (hanlder 对应通过注解来定义的Controller) {
  反射调用方法...
}
```

从代码中我们可以看出，这种实现方式会有很多if-else分支判断，而且，如果要增加一个新的Controller的定义方法，我们就要在DispatcherServlet类代码中，对应地增加一段如上伪代码所示的if逻辑。这显然不符合开闭原则。

实际上，我们可以利用是适配器模式对代码进行改造，让其满足开闭原则，能更好地支持扩赞。在第51节课中，我们讲到，适配器其中一个作用是“统一多个类的接口设计”。利用适配器模式，我们将不同方式定义的Controller类中的函数，适配为统一的函数定义。这样，我们就能在DispatcherServlet类代码中，移除掉if-else分支判断逻辑，调用统一的函数。

刚刚讲了大致的设计思路，我们再具体看下Spring的代码实现。

Spring定义了统一的接口HandlerAdapter，并且对每种Controller定义了对应的适配器类。这些适配器类包括：AnnotationMethodHandlerAdapter、SimpleControllerHandlerAdapter、SimpleServletHandlerAdapter等。源码我贴到了下面，你可以结合着看下。

```
public interface HandlerAdapter {
  boolean supports(Object var1);

  ModelAndView handle(HttpServletRequest var1, HttpServletResponse var2, Object var3) throws Exception;

  long getLastModified(HttpServletRequest var1, Object var2);
}

// 对应实现Controller接口的Controller
public class SimpleControllerHandlerAdapter implements HandlerAdapter {
  public SimpleControllerHandlerAdapter() {
  }

  public boolean supports(Object handler) {
    return handler instanceof Controller;
```

```
  }

  public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) thro
    return ((Controller)handler).handleRequest(request, response);
  }

  public long getLastModified(HttpServletRequest request, Object handler) {
    return handler instanceof LastModified ? ((LastModified)handler).getLastModified(request) : -1L;
  }
}

// 对应实现Servlet接口的Controller
public class SimpleServletHandlerAdapter implements HandlerAdapter {
  public SimpleServletHandlerAdapter() {
  }

  public boolean supports(Object handler) {
    return handler instanceof Servlet;
  }

  public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) thro
    ((Servlet)handler).service(request, response);
    return null;
  }

  public long getLastModified(HttpServletRequest request, Object handler) {
    return -1L;
  }
}

//AnnotationMethodHandlerAdapter对应通过注解实现的Controller,
//代码太多了，我就不贴在这里了
```

在DispatcherServlet类中，我们就不需要区分对待不同的Controller对象了，统一调用HandlerAdapter的handle()函数就可以了。按照这个思路实现的伪代码如下所示。你看，这样就没有烦人的if-else逻辑了吧?

```
// 之前的实现方式
Handler handler = handlerMapping.get(URL);
if (handler instanceof Controller) {
  ((Controller)handler).handleRequest(...);
} else if (handler instanceof Servlet) {
  ((Servlet)handler).service(...);
} else if (hanlder 对应通过注解来定义的Controller) {
  反射调用方法...
}

// 现在实现方式
HandlerAdapter handlerAdapter = handlerMapping.get(URL);
handlerAdapter.handle(...);
```

策略模式在Spring中的应用

我们前面讲到，Spring AOP是通过动态代理来实现的。熟悉Java的同学应该知道，具体到代码实现，Spring支持两种动态代理实现方式，一种是JDK提供的动态代理实现方式，另一种是Cglib提供的动态代理实现方式。

前者需要被代理的类有抽象的接口定义，后者不需要（这两种动态代理实现方式的更多区别请自行百度研究吧）。针对不同的被代理类，Spring会在运行时动态地选择不同的动态代理实现方式。这个应用场景实际上就是策略模式的典型应用场景。

我们前面讲过，策略模式包含三部分，策略的定义、创建和使用。接下来，我们具体看下，这三个部分是如何体现在Spring源码中的。

在策略模式中，策略的定义这一部分很简单。我们只需要定义一个策略接口，让不同的策略类都实现这一个策略接口。对应到Spring源码，AopProxy是策略接口，JdkDynamicAopProxy、CglibAopProxy是两个实现了AopProxy接口的策略类。其中，AopProxy接口的定义如下所示：

```
public interface AopProxy {
  Object getProxy();
  Object getProxy(ClassLoader var1);
}
```

在策略模式中，策略的创建一般通过工厂方法来实现。对应到Spring源码，AopProxyFactory是一个工厂类接口，DefaultAopProxyFactory是一个默认的工厂类，用来创建AopProxy对象。两者的源码如下所示：

```
public interface AopProxyFactory {
  AopProxy createAopProxy(AdvisedSupport var1) throws AopConfigException;
}

public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {
  public DefaultAopProxyFactory() {
  }

  public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
    if (!config.isOptimize() && !config.isProxyTargetClass() && !this.hasNoUserSuppliedProxyInterfaces(conf
      return new JdkDynamicAopProxy(config);
    } else {
      Class<?> targetClass = config.getTargetClass();
      if (targetClass == null) {
        throw new AopConfigException("TargetSource cannot determine target class: Either an interface or a
      } else {
        return (AopProxy)(!targetClass.isInterface() && !Proxy.isProxyClass(targetClass) ? new ObjenesisCgl
      }
    }
  }

  //用来判断用哪个动态代理实现方式
  private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
    Class<?>[] ifcs = config.getProxiedInterfaces();
    return ifcs.length == 0 || ifcs.length == 1 && SpringProxy.class.isAssignableFrom(ifcs[0]);
  }
}
```

策略模式的典型应用场景，一般是通过环境变量、状态值、计算结果等动态地决定使用哪个策略。对应到Spring源码中，我们可以参看刚刚给出的DefaultAopProxyFactory类中的createAopProxy()函数的代码实现。其中，第10行代码是动态选择哪种策略的判断条件。

组合模式在Spring中的应用

上节课讲到Spring“再封装、再抽象”设计思想的时候，我们提到了Spring Cache。Spring Cache提供了一套抽象的Cache接口。使用它我们能够统一不同缓存实现（Redis、Google Guava…）的不同的访问方式。Spring中针对不同缓存实现的不同缓存访问类，都依赖这个接口，比如：EhCacheCache、GuavaCache、NoOpCache、RedisCache、JCacheCache、ConcurrentMapCache、CaffeineCache。Cache接口的源码如下所示：

```
public interface Cache {
  String getName();
  Object getNativeCache();
  Cache.ValueWrapper get(Object var1);
  <T> T get(Object var1, Class<T> var2);
  <T> T get(Object var1, Callable<T> var2);
  void put(Object var1, Object var2);
  Cache.ValueWrapper putIfAbsent(Object var1, Object var2);
  void evict(Object var1);
  void clear();

  public static class ValueRetrievalException extends RuntimeException {
    private final Object key;

    public ValueRetrievalException(Object key, Callable<?> loader, Throwable ex) {
      super(String.format("Value for key '%s' could not be loaded using '%s'", key, loader), ex);
      this.key = key;
    }

    public Object getKey() {
      return this.key;
    }
  }

  public interface ValueWrapper {
    Object get();
  }
}
```

在实际的开发中，一个项目有可能会用到多种不同的缓存，比如既用到Google Guava缓存，也用到Redis缓存。除此之外，同一个缓存实例，也可以根据业务的不同，分割成多个小的逻辑缓存单元（或者叫作命名空间）。

为了管理多个缓存，Spring还提供了缓存管理功能。不过，它包含的功能很简单，主要有这样两部分：一个是根据缓存名字（创建Cache对象的时候要设置name属性）获取Cache对象；另一个是获取管理器管理的所有缓存的名字列表。对应的Spring源码如下所示：

```
public interface CacheManager {
  Cache getCache(String var1);
  Collection<String> getCacheNames();
}
```

刚刚给出的是CacheManager接口的定义，那如何来实现这两个接口呢？实际上，这就要用到了我们之前讲过的组合模式。

我们前面讲过，组合模式主要应用在能表示成树形结构的一组数据上。树中的结点分为叶子节点和中间节点两类。对应到Spring源码，EhCacheManager、SimpleCacheManager、NoOpCacheManager、RedisCacheManager等表示叶子节点，CompositeCacheManager表示中间节点。

叶子节点包含的是它所管理的Cache对象，中间节点包含的是其他CacheManager管理器，既可以是CompositeCacheManager，也可以是具体的管理器，比如EhCacheManager、RedisManager等。

我把CompositeCacheManger的代码贴到了下面，你可以结合着讲解一块看下。其中，getCache()、getCacheNames()两个函数的实现都用到了递归。这正是树形结构最能发挥优势的地方。

```
public class CompositeCacheManager implements CacheManager, InitializingBean {
  private final List<CacheManager> cacheManagers = new ArrayList();
  private boolean fallbackToNoOpCache = false;

  public CompositeCacheManager() {
  }

  public CompositeCacheManager(CacheManager... cacheManagers) {
    this.setCacheManagers(Arrays.asList(cacheManagers));
  }

  public void setCacheManagers(Collection<CacheManager> cacheManagers) {
    this.cacheManagers.addAll(cacheManagers);
  }

  public void setFallbackToNoOpCache(boolean fallbackToNoOpCache) {
    this.fallbackToNoOpCache = fallbackToNoOpCache;
  }

  public void afterPropertiesSet() {
    if (this.fallbackToNoOpCache) {
      this.cacheManagers.add(new NoOpCacheManager());
    }

  }

  public Cache getCache(String name) {
    Iterator var2 = this.cacheManagers.iterator();

    Cache cache;
    do {
      if (!var2.hasNext()) {
        return null;
      }

      CacheManager cacheManager = (CacheManager)var2.next();
      cache = cacheManager.getCache(name);
    } while(cache == null);

    return cache;
  }

  public Collection<String> getCacheNames() {
    Set<String> names = new LinkedHashSet();
    Iterator var2 = this.cacheManagers.iterator();
```

```
    while(var2.hasNext()) {
      CacheManager manager = (CacheManager)var2.next();
      names.addAll(manager.getCacheNames());
    }

    return Collections.unmodifiableSet(names);
  }
}
```

装饰器模式在Spring中的应用

我们知道，缓存一般都是配合数据库来使用的。如果写缓存成功，但数据库事务回滚了，那缓存中就会有脏数据。为了解决这个问题，我们需要将缓存的写操作和数据库的写操作，放到同一个事务中，要么都成功，要么都失败。

实现这样一个功能，Spring使用到了装饰器模式。TransactionAwareCacheDecorator增加了对事务的支持，在事务提交、回滚的时候分别对Cache的数据进行处理。

TransactionAwareCacheDecorator实现Cache接口，并且将所有的操作都委托给targetCache来实现，对其中的写操作添加了事务功能。这是典型的装饰器模式的应用场景和代码实现，我就不多作解释了。

```
public class TransactionAwareCacheDecorator implements Cache {
  private final Cache targetCache;

  public TransactionAwareCacheDecorator(Cache targetCache) {
    Assert.notNull(targetCache, "Target Cache must not be null");
    this.targetCache = targetCache;
  }

  public Cache getTargetCache() {
    return this.targetCache;
  }

  public String getName() {
    return this.targetCache.getName();
  }

  public Object getNativeCache() {
    return this.targetCache.getNativeCache();
  }

  public ValueWrapper get(Object key) {
    return this.targetCache.get(key);
  }

  public <T> T get(Object key, Class<T> type) {
    return this.targetCache.get(key, type);
  }

  public <T> T get(Object key, Callable<T> valueLoader) {
    return this.targetCache.get(key, valueLoader);
  }

  public void put(final Object key, final Object value) {
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
```

```
      TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
        public void afterCommit() {
          TransactionAwareCacheDecorator.this.targetCache.put(key, value);
        }
      });
    } else {
      this.targetCache.put(key, value);
    }
  }

  public ValueWrapper putIfAbsent(Object key, Object value) {
    return this.targetCache.putIfAbsent(key, value);
  }

  public void evict(final Object key) {
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
      TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
        public void afterCommit() {
          TransactionAwareCacheDecorator.this.targetCache.evict(key);
        }
      });
    } else {
      this.targetCache.evict(key);
    }

  }

  public void clear() {
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
      TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
        public void afterCommit() {
          TransactionAwareCacheDecorator.this.targetCache.clear();
        }
      });
    } else {
      this.targetCache.clear();
    }
  }
}
```

工厂模式在Spring中的应用

在Spring中，工厂模式最经典的应用莫过于实现IOC容器，对应的Spring源码主要是BeanFactory类和ApplicationContext相关类（AbstractApplicationContext、ClassPathXmlApplicationContext、FileSystemXmlApplicationContext…）。除此之外，在理论部分，我还带你手把手实现了一个简单的IOC容器。你可以回过头去再看下。

在Spring中，创建Bean的方式有很多种，比如前面提到的纯构造函数、无参构造函数加setter方法。我写了一个例子来说明这两种创建方式，代码如下所示：

```
public class Student {
  private long id;
  private String name;

  public Student(long id, String name) {
    this.id = id;
    this.name = name;
```

```
  }

  public void setId(long id) {
    this.id = id;
  }

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

// 使用构造函数来创建Bean
<bean id="student" class="com.xzg.cd.Student">
    <constructor-arg name="id" value="1"/>
    <constructor-arg name="name" value="wangzheng"/>
</bean>

// 使用无参构造函数+setter方法来创建Bean
<bean id="student" class="com.xzg.cd.Student">
    <property name="id" value="1"></property>
    <property name="name" value="wangzheng"></property>
</bean>
```

实际上，除了这两种创建Bean的方式之外，我们还可以通过工厂方法来创建Bean。还是刚刚这个例子，用这种方式来创建Bean的话就是下面这个样子：

```
public class StudentFactory {
  private static Map<Long, Student> students = new HashMap<>();

  static{
    map.put(1, new Student(1,"wang"));
    map.put(2, new Student(2,"zheng"));
    map.put(3, new Student(3,"xzg"));
  }

  public static Student getStudent(long id){
    return students.get(id);
  }
}

// 通过工厂方法getStudent(2)来创建BeanId="zheng""的Bean
<bean id="zheng" class="com.xzg.cd.StudentFactory" factory-method="getStudent">
    <constructor-arg value="2"></constructor-arg>
</bean>
```

其他模式在Spring中的应用

前面的几个模式在Spring中的应用讲解的都比较详细，接下来的几个模式，大部分都是我们之前讲过的，这里只是简单总结一下，点到为止，如果你对哪块有遗忘，可以回过头去看下理论部分的讲解。

SpEL，全称叫Spring Expression Language，是Spring中常用来编写配置的表达式语言。它定义了一系列的语法规则。我们只要按照这些语法规则来编写表达式，Spring就能解析出表达式的含义。实际上，这就是我们前面讲到的解释器模式的典型应用场景。

因为解释器模式没有一个非常固定的代码实现结构，而且Spring中SpEL相关的代码也比较多，所以这里就不带你一块阅读源码了。如果感兴趣或者项目中正好要实现类似的功能的时候，你可以再去阅读、借鉴它的代码实现。代码主要集中在spring-expresssion这个模块下面。

前面讲到单例模式的时候，我提到过，单例模式有很多弊端，比如单元测试不友好等。应对策略就是通过IOC容器来管理对象，通过IOC容器来实现对象的唯一性的控制。实际上，这样实现的单例并非真正的单例，它的唯一性的作用范围仅仅在同一个IOC容器内。

除此之外，Spring还用到了观察者模式、模板模式、职责链模式、代理模式。其中，观察者模式、模板模式在上一节课已经详细讲过了。

实际上，在Spring中，只要后缀带有Template的类，基本上都是模板类，而且大部分都是用Callback回调来实现的，比如JdbcTemplate、RedisTemplate等。剩下的两个模式在Spring中的应用应该人尽皆知了。职责链模式在Spring中的应用是拦截器（Interceptor），代理模式经典应用是AOP。

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。

我们今天提到的设计模式有11种，它们分别是适配器模式、策略模式、组合模式、装饰器模式、工厂模式、单例模式、解释器模式、观察者模式、模板模式、职责链模式、代理模式，基本上占了23种设计模式的一半。这还只是我所知道的，实际上，Spring用到的设计模式可能还要更多。你看，设计模式并非“花拳绣腿”吧，它在实际的项目开发中，确实有很多应用，确实可以发挥很大的作用。

还是那句话，对于今天的内容，你不需要去记忆哪个类用到了哪个设计模式。你只需要跟着我的讲解，把每个设计模式在Spring中的应用场景，搞懂就可以了。看到类似的代码，能够立马识别出它用到了哪种设计模式；看到类似的应用场景，能够立马反映出要用哪种模式去解决，这样就说明你已经掌握得足够好了。

课堂讨论

我们前面讲到，除了纯构造函数、构造函数加setter方法和工厂方法之外，还有另外一个经常用来创建对象的模式，Builder模式。如果我们让Spring支持通过Builder模式来创建Bean，应该如何来编写代码和配置呢？你可以设计一下吗？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 悟光 2020-05-20 09:14:15
  尝试了一下，xml配置未找到直接调用build方法的配置，用构造器注入
  类：

  ```
  public class Student {
  private long id;
  private String name;

  private Student(Builder builder) {
  this.id =builder.id;
  this.name = builder.name;
  ```

```
}

public String getName() {
return name;
}


public static class Builder {
private long id;
private String name;
public Student build() {
if (StringUtils.isEmpty(name)){
throw new IllegalArgumentException("name is empty");
}
return new Student(this);
}

public void setId(long id) {
this.id = id;
}

public void setName(String name) {
this.name = name;
}
}
}
```
配置：
```
<bean id="build" class="cn.gitv.rt.advertisv5.utils.Student.Builder" >
<property name="name" value="aa"/>
<property name="id" value="2"/>
</bean>
<bean id="student" class="cn.gitv.rt.advertisv5.utils.Student">
<constructor-arg ref="build"/>
</bean>
```
2、“实际上，我们可以利用是适配器模式对代码进行改造，让其满足开闭原则，能更好地支持扩赞”。这一句应该 “赞” 敲串行了。