

59 | 模板模式（下）：模板模式与Callback回调函数有何区别和联系？

王争 · 设计模式之美



模板模式（下）

上一节课中，我们学习了模板模式的原理、实现和应用。它常用在框架开发中，通过提供功能扩展点，让框架用户在不修改框架源码的情况下，基于扩展点定制化框架的功能。除此之外，模板模式还可以起到代码复用的作用。


复用和扩展是模板模式的两大作用，实际上，还有另外一个技术概念，也能起到跟模板模式相同的作用，那就是**回调**（Callback）。今天我们今天就来了解一下，回调的原理、实现和应用，以及它跟模板模式的区别和联系。

话不多说，让我们正式开始今天的学习吧！

回调的原理解析

相对于普通的函数调用来说，回调是一种双向调用关系。A 类事先注册某个函数 F 到 B 类，A 类在调用 B 类的 P 函数的时候，B 类反过来调用 A 类注册给它的 F 函数。这里的 F 函数就是“回调函数”。A 调用 B，B 反过来又调用 A，这种调用机制就叫作“回调”。

A 类如何将回调函数传递给 B 类呢？不同的编程语言，有不同的实现方法。C 语言可以使用函数指针，Java 则需要使用包裹了回调函数的类对象，我们简称为回调对象。这里我用 Java 语言举例说明一下。代码如下所示：

 复制代码

```
1 public interface ICallback {
2     void methodToCallback();
3 }
4
5 public class BClass {
6     public void process(ICallback callback) {
7         //...
8         callback.methodToCallback();
9         //...
10    }
11 }
12
13 public class AClass {
14     public static void main(String[] args) {
15         BClass b = new BClass();
16         b.process(new ICallback() { //回调对象
17             @Override
18             public void methodToCallback() {
19                 System.out.println("Call back me.");
20             }
21         });
22     }
23 }
```

上面就是 Java 语言中回调的典型代码实现。从代码实现中，我们可以看出，回调跟模板模式一样，也具有复用和扩展的功能。除了回调函数之外，BClass 类的 process() 函数中的逻辑都可以复用。如果 ICallback、BClass 类是框架代码，AClass 是使用框架的客户端代码，我们可以通过 ICallback 定制 process() 函数，也就是说，框架因此具有了扩展的能力。

实际上，回调不仅可以应用在代码设计上，在更高层次的架构设计上也比较常用。比如，通过三方支付系统来实现支付功能，用户在发起支付请求之后，一般不会一直阻塞到支付结果返回，而是注册回调接口（类似回调函数，一般是一个回调用的 URL）给三方支付系统，等三方支付系统执行完成之后，将结果通过回调接口返回给用户。

回调可以分为同步回调和异步回调（或者延迟回调）。同步回调指在函数返回之前执行回调函数；异步回调指的是在函数返回之后执行回调函数。上面的代码实际上是同步回调的实现方式，在 `process()` 函数返回之前，执行完回调函数 `methodToCallback()`。而上面支付例子是异步回调的实现方式，发起支付之后不需要等待回调接口被调用就直接返回。从应用场景上来看，同步回调看起来更像模板模式，异步回调看起来更像观察者模式。

应用举例一：JdbcTemplate

Spring 提供了很多 Template 类，比如，`JdbcTemplate`、`RedisTemplate`、`RestTemplate`。尽管都叫作 `xxxTemplate`，但它们并非基于模板模式来实现的，而是基于回调来实现的，确切地说应该是同步回调。而同步回调从应用场景上很像模板模式，所以，在命名上，这些类使用 Template（模板）这个单词作为后缀。

这些 Template 类的设计思路都很相近，所以，我们只拿其中的 `JdbcTemplate` 来举例分析一下。对于其他 Template 类，你可以阅读源码自行分析。

在前面的章节中，我们也多次提到，Java 提供了 JDBC 类库来封装不同类型的数据库操作。不过，直接使用 JDBC 来编写操作数据库的代码，还是有点复杂的。比如，下面这段是使用 JDBC 来查询用户信息的代码。

 复制代码

```
1 public class JdbcDemo {
2     public User queryUser(long id) {
3         Connection conn = null;
4         Statement stmt = null;
5         try {
6             //1.加载驱动
7             Class.forName("com.mysql.jdbc.Driver");
8             conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/demo", "xzg
9
10            //2.创建statement类对象，用来执行SQL语句
11            stmt = conn.createStatement();
12
13            //3.ResultSet类，用来存放获取的结果集
14            String sql = "select * from user where id=" + id;
15            ResultSet resultSet = stmt.executeQuery(sql);
16
17            String eid = null, ename = null, price = null;
18
```

```

19     while (resultSet.next()) {
20         User user = new User();
21         user.setId(resultSet.getLong("id"));
22         user.setName(resultSet.getString("name"));
23         user.setTelephone(resultSet.getString("telephone"));
24         return user;
25     }
26 } catch (ClassNotFoundException e) {
27     // TODO: log...
28 } catch (SQLException e) {
29     // TODO: log...
30 } finally {
31     if (conn != null)
32         try {
33             conn.close();
34         } catch (SQLException e) {
35             // TODO: log...
36         }
37     if (stmt != null)
38         try {
39             stmt.close();
40         } catch (SQLException e) {
41             // TODO: log...
42         }
43 }
44 return null;
45 }
46
47 }

```

queryUser() 函数包含很多流程性质的代码，跟业务无关，比如，加载驱动、创建数据库连接、创建 statement、关闭连接、关闭 statement、处理异常。针对不同的 SQL 执行请求，这些流程性质的代码是相同的、可以复用的，我们不需要每次都重新敲一遍。


针对这个问题，Spring 提供了 JdbcTemplate，对 JDBC 进一步封装，来简化数据库编程。使用 JdbcTemplate 查询用户信息，我们只需要编写跟这个业务有关的代码，其中包括，查询用户的 SQL 语句、查询结果与 User 对象之间的映射关系。其他流程性质的代码都封装在了 JdbcTemplate 类中，不需要我们每次都重新编写。我用 JdbcTemplate 重写了上面的例子，代码简单了很多，如下所示：

```

1 public class JdbcTemplateDemo {
2     private JdbcTemplate jdbcTemplate;
3
4     public User queryUser(long id) {
5         String sql = "select * from user where id="+id;
6         return jdbcTemplate.query(sql, new UserRowMapper()).get(0);
7     }
8
9     class UserRowMapper implements RowMapper<User> {
10         public User mapRow(ResultSet rs, int rowNum) throws SQLException {
11             User user = new User();
12             user.setId(rs.getLong("id"));
13             user.setName(rs.getString("name"));
14             user.setTelephone(rs.getString("telephone"));
15             return user;
16         }
17     }
18 }

```

那 JdbcTemplate 底层具体是如何实现的呢？我们来看一下它的源码。因为 JdbcTemplate 代码比较多，我只摘抄了部分相关代码，贴到了下面。其中，JdbcTemplate 通过回调的机制，将不变的执行流程抽离出来，放到模板方法 execute() 中，将可变的部分设计成回调 StatementCallback，由用户来定制。query() 函数是对 execute() 函数的二次封装，让接口用起来更加方便。

 复制代码

```

1 @Override
2 public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws DataAccessException {
3     return query(sql, new RowMapperResultSetExtractor<T>(rowMapper));
4 }
5
6 @Override
7 public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws Data
8     Assert.notNull(sql, "SQL must not be null");
9     Assert.notNull(rse, "ResultSetExtractor must not be null");
10    if (logger.isDebugEnabled()) {
11        logger.debug("Executing SQL query [" + sql + "]");
12    }
13
14    class QueryStatementCallback implements StatementCallback<T>, SqlProvider {
15        @Override
16        public T doInStatement(Statement stmt) throws SQLException {
17            ResultSet rs = null;

```

```


18     try {
19         rs = stmt.executeQuery(sql);
20         ResultSet rsToUse = rs;
21         if (nativeJdbcExtractor != null) {
22             rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
23         }
24         return rse.extractData(rsToUse);
25     }
26     finally {
27         JdbcUtils.closeResultSet(rs);
28     }
29 }
30 @Override
31 public String getSql() {
32     return sql;
33 }
34 }
35
36 return execute(new QueryStatementCallback());
37 }
38
39 @Override
40 public <T> T execute(StatementCallback<T> action) throws DataAccessException {
41     Assert.notNull(action, "Callback object must not be null");
42
43     Connection con = DataSourceUtils.getConnection(getDataSource());
44     Statement stmt = null;
45     try {
46         Connection conToUse = con;
47         if (this.nativeJdbcExtractor != null &&
48             this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {
49             conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
50         }
51         stmt = conToUse.createStatement();
52         applyStatementSettings(stmt);
53         Statement stmtToUse = stmt;
54         if (this.nativeJdbcExtractor != null) {
55             stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
56         }
57         T result = action.doInStatement(stmtToUse);
58         handleWarnings(stmt);
59         return result;
60     }
61     catch (SQLException ex) {
62         // Release Connection early, to avoid potential connection pool deadlock
63         // in the case when the exception translator hasn't been initialized yet.
64         JdbcUtils.closeStatement(stmt);
65         stmt = null;
66         DataSourceUtils.releaseConnection(con, getDataSource());

```

```
67     con = null;
68     throw getExceptionTranslator().translate("StatementCallback", getSql(action), e
69 }
70 finally {
71     JdbcUtils.closeStatement(stmt);
72     DataSourceUtils.releaseConnection(con, getDataSource());
73 }
74 }
```

应用举例二：setOnClickListener()

在客户端开发中，我们经常给控件注册事件监听器，比如下面这段代码，就是在 Android 应用开发中，给 Button 控件的点击事件注册监听器。

 复制代码

```
1 Button button = (Button)findViewById(R.id.button);
2 button.setOnClickListener(new OnClickListener() {
3     @Override
4     public void onClick(View v) {
5         System.out.println("I am clicked.");
6     }
7 });
```

从代码结构上来看，事件监听器很像回调，即传递一个包含回调函数（onClick()）的对象给另一个函数。从应用场景上来看，它又很像观察者模式，即事先注册观察者（OnClickListener），当用户点击按钮的时候，发送点击事件给观察者，并且执行相应的 onClick() 函数。

我们前面讲到，回调分为同步回调和异步回调。这里的回调算是异步回调，我们往 setOnClickListener() 函数中注册好回调函数之后，并不需要等待回调函数执行。这也印证了我们前面讲的，异步回调比较像观察者模式。

应用举例三：addShutdownHook()

Hook 可以翻译成“钩子”，那它跟 Callback 有什么区别呢？


网上有人认为 Hook 就是 Callback，两者说的是一回事儿，只是表达不同而已。而有人觉得 Hook 是 Callback 的一种应用。Callback 更侧重语法机制的描述，Hook 更加侧重应用场景的描述。我个人比较认可后面一种说法。不过，这个也不重要，我们只需要见了代码能认识，遇到场景会用就可以了。

Hook 比较经典的应用场景是 Tomcat 和 JVM 的 shutdown hook。接下来，我们拿 JVM 来举例说明一下。JVM 提供了 `Runtime.addShutdownHook(Thread hook)` 方法，可以注册一个 JVM 关闭的 Hook。当应用程序关闭的时候，JVM 会自动调用 Hook 代码。代码示例如下所示：

 复制代码

```
1 public class ShutdownHookDemo {
2
3     private static class ShutdownHook extends Thread {
4         public void run() {
5             System.out.println("I am called during shutting down.");
6         }
7     }
8
9     public static void main(String[] args) {
10         Runtime.getRuntime().addShutdownHook(new ShutdownHook());
11     }
12
13 }
```

我们再来看 `addShutdownHook()` 的代码实现，如下所示。这里我只给出了部分相关代码。

 复制代码

```
1 public class Runtime {
2     public void addShutdownHook(Thread hook) {
3         SecurityManager sm = System.getSecurityManager();
4         if (sm != null) {
5             sm.checkPermission(new RuntimePermission("shutdownHooks"));
6         }
7         ApplicationShutdownHooks.add(hook);
8     }
9 }
10
11 class ApplicationShutdownHooks {
```



```

12  /* The set of registered hooks */
13  private static IdentityHashMap<Thread, Thread> hooks;
14  static {
15      hooks = new IdentityHashMap<>();
16      } catch (IllegalStateException e) {
17          hooks = null;
18      }
19  }
20
21  static synchronized void add(Thread hook) {
22      if(hooks == null)
23          throw new IllegalStateException("Shutdown in progress");
24
25      if (hook.isAlive())
26          throw new IllegalArgumentException("Hook already running");
27
28      if (hooks.containsKey(hook))
29          throw new IllegalArgumentException("Hook previously registered");
30
31      hooks.put(hook, hook);
32  }
33
34  static void runHooks() {
35      Collection<Thread> threads;
36      synchronized(ApplicationShutdownHooks.class) {
37          threads = hooks.keySet();
38          hooks = null;
39      }
40
41      for (Thread hook : threads) {
42          hook.start();
43      }
44      for (Thread hook : threads) {
45          while (true) {
46              try {
47                  hook.join();
48                  break;
49              } catch (InterruptedException ignored) {
50              }
51          }
52      }
53  }
54  }

```

从代码中我们可以发现，有关 Hook 的逻辑都被封装到 ApplicationShutdownHooks 类中了。当应用程序关闭的时候，JVM 会调用这个类的 runHooks() 方法，创建多个线程，并发地

执行多个 Hook。我们在注册完 Hook 之后，并不需要等待 Hook 执行完成，所以，这也算是一种异步回调。

模板模式 VS 回调

回调的原理、实现和应用到此就都讲完了。接下来，我们从应用场景和代码实现两个角度，来对比一下模板模式和回调。

从应用场景上来看，同步回调跟模板模式几乎一致。它们都是在一个大的算法骨架中，自由替换其中的某个步骤，起到代码复用和扩展的目的。而异步回调跟模板模式有较大差别，更像是观察者模式。

从代码实现上来看，回调和模板模式完全不同。回调基于组合关系来实现，把一个对象传递给另一个对象，是一种对象之间的关系；模板模式基于继承关系来实现，子类重写父类的抽象方法，是一种类之间的关系。

前面我们也讲到，组合优于继承。实际上，这里也不例外。在代码实现上，回调相对于模板模式会更加灵活，主要体现在下面几点。

像 Java 这种只支持单继承的语言，基于模板模式编写的子类，已经继承了一个父类，不再具有继承的能力。

回调可以使用匿名类来创建回调对象，可以不用事先定义类；而模板模式针对不同的实现都要定义不同的子类。

如果某个类中定义了多个模板方法，每个方法都有对应的抽象方法，那即便我们只用到其中的一个模板方法，子类也必须实现所有的抽象方法。而回调就更加灵活，我们只需要往用到的模板方法中注入回调对象即可。

还记得上一节课的课堂讨论题目吗？看到这里，相信你应该有了答案了吧？

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。

今天，我们重点介绍了回调。它跟模板模式具有相同的作用：代码复用和扩展。在一些框架、类库、组件等的设计中经常会用到。

相对于普通的函数调用，回调是一种双向调用关系。A 类事先注册某个函数 F 到 B 类，A 类在调用 B 类的 P 函数的时候，B 类反过来调用 A 类注册给它的 F 函数。这里的 F 函数就是“回调函数”。A 调用 B，B 反过来又调用 A，这种调用机制就叫作“回调”。

回调可以细分为同步回调和异步回调。从应用场景上来看，同步回调看起来更像模板模式，异步回调看起来更像观察者模式。回调跟模板模式的区别，更多的是在代码实现上，而非应用场景上。回调基于组合关系来实现，模板模式基于继承关系来实现，回调比模板模式更加灵活。

课堂讨论

对于 Callback 和 Hook 的区别，你有什么不同的理解吗？在你熟悉的编程语言中，有没有提供相应的语法概念？是叫 Callback，还是 Hook 呢？

欢迎留言和我分享你的想法。如果有收获，欢迎你把这篇文章分享给你的朋友。

AI智能总结

回调函数和模板模式是常用的软件设计模式，它们都具有复用和扩展的功能。回调函数是一种双向调用关系，A类事先注册某个函数F到B类，A类在调用B类的P函数的时候，B类反过来调用A类注册给它的F函数。而模板模式则是通过提供功能扩展点，让框架用户在不修改框架源码的情况下，基于扩展点定制化框架的功能，同时也可以起到代码复用的作用。

回调函数和模板模式在实际应用中有着相似的功能，但也有一些区别。回调函数可以分为同步回调和异步回调，而模板模式则更侧重于提供一个框架的基本结构，让用户根据需要进行扩展和定制。在实际应用中，回调函数常用于事件监听器的注册，比如在Android应用开发中给控件注册事件监听器，而模板模式常用于框架开发中，提供功能扩展点。

举例来说，Spring框架中的JdbcTemplate就是基于回调函数实现的，它通过回调的机制，将不变的执行流程抽离出来，放到模板方法中，将可变的部分设计成回调，由用户来定制。另外，在客户端开发中，给控件注册事件监听器也是一种回调的应用，比如在Android应用开发中给Button控件的点击事件注册监听器。

总的来说，回调函数和模板模式都是在软件设计中常用的技术手段，它们各自在不同的场景下发挥着重要的作用，帮助开发者实现代码复用和功能扩展。回调和模板模式的区别主要在于实现方式和应用场景，而它们都是为了实现代码复用和功能扩展而存在的重要设计模式。

全部留言 (90)

最新 精选



Java 垒墙工程师

2020-08-25

回调的方式是不是打乱了系统调用的层次？相互依赖，依赖关系变得复杂

作者回复: 有点，不过权衡利弊吧，没有绝对的完美。

共 2 条评论 >



1



吴小智

2020-03-18

模板方法和回调应用场景是一致的，都是定义好算法骨架，并对外开放扩展点，符合开闭原则；两者的区别是代码的实现上不同，模板方法是通过继承来实现，是自己调用自己；回调是类之间的组合。

共 1 条评论 >



109



L!en6o

2020-03-19

曾经重构代码对这模板模式和callback就很疑惑。个人觉得callback更加灵活，适合算法逻辑较少的场景，实现一两个方法很舒服。比如Guava 的Futures.addCallback 回调 onSuccess onFailure方法。而模板模式适合更加复杂的场景，并且子类可以复用父类提供的方法，根据场景判断是否需要重写更加方便。

共 4 条评论 >



77



小晏子

2020-03-18

callback和hook不是一个层面的东西，callback是程序设计方面的一种技术手段，是编程语言层面的东西，hook是通过这种技术手段实现的功能扩展点，其基本原理就是callback。比如windows api中提供的各种事件通知机制，其本身是windows开放给用户可以扩展自己想要的功能的扩展点，而实现这些功能的手段是callback。

只要编程语言支持传递函数作为参数，都可以支持callback设计，比如c, go, javascript, python等。另外一些框架中提供的功能扩展点我们称之为hook，比如vue在其实例生命周期中提供的各种hook函数。

共 2 条评论>

👍 59



剑八

2020-07-05

模板方法就是定义一个流程，每个流程结点可变的就是一个抽象spi，由不同实现去现。解决的是一个复用与扩展问题。复用的是这个流程本身以及某些结点可以是默认实现。扩展的是有些结点是可以有不同实现的场景。

回调是一种交互方式，由调用者告诉被调用者：你做完后还要做一个事情，这个事情是什么。然后被调用者做完后就可以做这个指定的事情。回调倒不用强制和模板方法概念合在一起。

共 5 条评论>

👍 43



柠檬C

2020-03-19

个人看法：模板模式关注点还是在类与对象上，通过继承与多态实现算法的扩展
回调关注点在方法上，虽然在java语言中不得不以匿名内部类的形式出现，但本质是将方法当做参数一样传递，有点函数式编程的意思了

共 2 条评论>

👍 19



pedro

2020-03-18

callback应该偏语言层面，hook偏业务层面，二者一个是概念，一个是具体的落地方式。

共 2 条评论>

👍 12



写代码的

2020-08-30

虽然模板模式和回调很像，甚至和可以互相替换，但是为了让它们的功能和名称更契合，我觉得按照这样原则来使用这两种方法是不是会更好些：如果预留的扩展点必须实现，因为这些扩展点包含和这个类本身相关的关键功能性代码，不实现的话这个类就是个半成品，无法使用，那么使用模板模式，因为模板模式使用的抽象类可以在与语言层面强制这些扩展点必须被实现；如果预留的扩展点可以不实现，或者这些扩展点的实现逻辑甚至可以和这个类完全无关，那么就使用回调，回调使用的组合关系恰好可以让类和扩展点的实现进行解耦，比如按钮上的事件回调，回调中的逻辑和按钮这个类本身的功能并没有什么关系，甚至回调可以传 null。

共 2 条评论>

👍 10



辣么大

2020-03-24

什么是“回调”？A注册一个函数到B，B执行某个函数时，会调用A注册的这个函数。
我见过的应用一般完全结束（关闭，收尾等）时用hook，其他情况用callback或者on...listener。这种区别更多是语意上的，不是实现上的。

共 1 条评论 >

👍 6



前端西瓜哥

2020-03-19

Callback 是在一个方法的执行中，调用嵌入的其他方法的机制，能很好地起到代码复用和框架扩展的作用。在 JavaScript 中，因为函数可以直接作为另一个函数的参数，所以能经常看到回调函数的身影，比如定时器 `setTimeout(callback, delay)`、Ajax 请求成功或失败对应的回调函数等。不过如果滥用回调的话，会在某些场景下会因为嵌套过多导致回调地狱。

Hook 本质上也是回调，但它往往和一些场景性的行为绑定在一起。在浏览器环境中，我们可以通过 `img.onload = func1` 来让图片在加载完后执行函数 `func1`，某种意义上算是一种 Hook。此外在 js 的 vue 框架中，也提供了组件生命周期的 Hook，比如 `beforeDestory` 钩子函数会在组件即将被销毁前执行，常用于销毁一些当前组件才会用到的定时器。



👍 6