59-模板模式（下）：模板模式与Callback回调函数有何区别和联系？

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）。今天我们今天就来看一下，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：

```
public interface ICallback {
  void methodToCallback();
}

public class BClass {
  public void process(ICallback callback) {
    //...
    callback.methodToCallback();
    //...
  }
}

public class AClass {
  public static void main(String[] args) {
    BClass b = new BClass();
    b.process(new ICallback() { //回调对象
      @Override
      public void methodToCallback() {
        System.out.println("Call back me.");
      }
    });
  }
}
```

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()。而上面支付的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。

```
public class JdbcDemo {
  public User queryUser(long id) {
    Connection conn = null;
    Statement stmt = null;
    try {
      //1.加载驱动
      Class.forName("com.mysql.jdbc.Driver");
      conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/demo", "xzg", "xzg");

      //2.创建statement类对象，用来执行SQL语句
      stmt = conn.createStatement();

      //3.ResultSet类，用来存放获取的结果集
      String sql = "select * from user where id=" + id;
      ResultSet resultSet = stmt.executeQuery(sql);

      String eid = null, ename = null, price = null;

      while (resultSet.next()) {
        User user = new User();
        user.setId(resultSet.getLong("id"));
        user.setName(resultSet.getString("name"));
        user.setTelephone(resultSet.getString("telephone"));
        return user;
      }
    } catch (ClassNotFoundException e) {
      // TODO: log...
    } catch (SQLException e) {
      // TODO: log...
    } finally {
```

```
      if (conn != null)
        try {
          conn.close();
        } catch (SQLException e) {
          // TODO: log...
        }
      if (stmt != null)
        try {
          stmt.close();
        } catch (SQLException e) {
          // TODO: log...
        }
    }
    return null;
  }

}
```

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：

```
public class JdbcTemplateDemo {
  private JdbcTemplate jdbcTemplate;

  public User queryUser(long id) {
    String sql = "select * from user where id="+id;
    return jdbcTemplate.query(sql, new UserRowMapper()).get(0);
  }

  class UserRowMapper implements RowMapper<User> {
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
      User user = new User();
      user.setId(rs.getLong("id"));
      user.setName(rs.getString("name"));
      user.setTelephone(rs.getString("telephone"));
      return user;
    }
  }
}
```

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。

```
@Override
public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws DataAccessException {
 return query(sql, new RowMapperResultSetExtractor<T>(rowMapper));
}

@Override
public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws DataAccessException {
 Assert.notNull(sql, "SQL must not be null");
 Assert.notNull(rse, "ResultSetExtractor must not be null");
 if (logger.isDebugEnabled()) {
  logger.debug("Executing SQL query [" + sql + "]");
 }

 class QueryStatementCallback implements StatementCallback<T>, SqlProvider {
  @Override
  public T doInStatement(Statement stmt) throws SQLException {
   ResultSet rs = null;
   try {
    rs = stmt.executeQuery(sql);
    ResultSet rsToUse = rs;
    if (nativeJdbcExtractor != null) {
     rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
    }
    return rse.extractData(rsToUse);
   }
   finally {
    JdbcUtils.closeResultSet(rs);
   }
  }
  @Override
  public String getSql() {
   return sql;
  }
 }

 return execute(new QueryStatementCallback());
}

@Override
public <T> T execute(StatementCallback<T> action) throws DataAccessException {
 Assert.notNull(action, "Callback object must not be null");

 Connection con = DataSourceUtils.getConnection(getDataSource());
 Statement stmt = null;
 try {
  Connection conToUse = con;
  if (this.nativeJdbcExtractor != null &&
    this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {
   conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
  }
  stmt = conToUse.createStatement();
  applyStatementSettings(stmt);
  Statement stmtToUse = stmt;
  if (this.nativeJdbcExtractor != null) {
   stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
  }
  T result = action.doInStatement(stmtToUse);
  handleWarnings(stmt);
  return result;
 }
 catch (SQLException ex) {
  // Release Connection early, to avoid potential connection pool deadlock
  // in the case when the exception translator hasn't been initialized yet.
```

```
   JdbcUtils.closeStatement(stmt);
   stmt = null;
   DataSourceUtils.releaseConnection(con, getDataSource());
   con = null;
   throw getExceptionTranslator().translate("StatementCallback", getSql(action), ex);
  }
  finally {
   JdbcUtils.closeStatement(stmt);
   DataSourceUtils.releaseConnection(con, getDataSource());
  }
 }
```

应用举例二：setClickListener()

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。

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

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：

```
public class ShutdownHookDemo {

  private static class ShutdownHook extends Thread {
```

```
    public void run() {
      System.out.println("I am called during shutting down.");
    }
  }

  public static void main(String[] args) {
    Runtime.getRuntime().addShutdownHook(new ShutdownHook());
  }

}
```

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。

```
public class Runtime {
  public void addShutdownHook(Thread hook) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      sm.checkPermission(new RuntimePermission("shutdownHooks"));
    }
    ApplicationShutdownHooks.add(hook);
  }
}

class ApplicationShutdownHooks {
    /* The set of registered hooks */
    private static IdentityHashMap<Thread, Thread> hooks;
    static {
            hooks = new IdentityHashMap<>();
        } catch (IllegalStateException e) {
            hooks = null;
        }
    }

    static synchronized void add(Thread hook) {
        if(hooks == null)
            throw new IllegalStateException("Shutdown in progress");

        if (hook.isAlive())
            throw new IllegalArgumentException("Hook already running");

        if (hooks.containsKey(hook))
            throw new IllegalArgumentException("Hook previously registered");

        hooks.put(hook, hook);
    }

    static void runHooks() {
        Collection<Thread> threads;
        synchronized(ApplicationShutdownHooks.class) {
            threads = hooks.keySet();
            hooks = null;
        }

        for (Thread hook : threads) {
            hook.start();
        }
        for (Thread hook : threads) {
            while (true) {
                try {
                    hook.join();
```

```
                    break;
                } catch (InterruptedException ignored) {
                }
            }
        }
    }
}
```

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。

精选留言：

- 唔多志 2020-03-18 01:05:40
  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；两者的却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。 [9赞]

- 黄林晴 2020-03-18 08:12:23
  打卡
  回调接口如果定义了多个方法，不也需要全部实现吗

  课后思考:
  android 中有个hook 概念，多用于反射修改源码机制，进行插件化相关的开发 [3赞]

- 小晏子 2020-03-18 07:43:35
  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，是编程语言成面的东西，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，golang，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。 [3赞]

- L!en6o 2020-03-19 10:12:56
  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。 [2赞]

- iLeGeND 2020-03-19 00:44:52
  回调函数是不是只能在同一个jvm下的 程序之间才能实现 [1赞]

- Fstar 2020-03-19 00:10:33
  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。 [1赞]

- Frank 2020-03-18 21:52:16
  打卡 今日学习回调函数，收获如下: 回调是一种A调用B，B又回来调用A的一种机制。它有两种方式：同步回调和异步回调。它的功能与模版模式类似都是复用与扩展。回调采用的是组合方式，更加灵活。而模版模式采用的是继承，有单继承的局限，如果继承层次过深，后期不便于维护。自己在写JavaScript时，常常使用回调这种方式来完成需求，通过今日的学习，进一步加深了对回调机制的理解。 [1赞]

- pedro 2020-03-18 08:45:14
  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。 [1赞]

- 大头 2020-03-18 05:32:32
  java8支持参数传递，以及lambda的使用，也是对回掉的简化 [1赞]

- Rain 2020-03-19 23:45:14
  对于callback 和 hook 的提供意图来说，提供callback 的时候是希望在callback里面完成主要的工作。hook的目的则在于扩展。前者的提供者通常没我在默认实现，非常希望callback 完成具体任务，而hook是基本已经实现了大部分功能，如果需要特殊操作，那就在hook里面做。

- L🚲🐵 2020-03-19 13:52:50
  模板方法和回调应用场景一致, 两者的区别是代码实现上不一样, 模板方法是通过 继承来实现, 是自己调用自己, 回调是通过组合来实现, 是类之间的组合. java 中有 Callback的概念

- Jxin 2020-03-19 12:53:38
  1.callback是一个语法机制的命名，hook是一个应用场景的命名。但我认为两者换下语义更强。hook描述语法机制，指的就是添加钩子方法这么一种语法机制。callback描述应用场景，特指调用方需要被调用方回调自己的这种场景，这属于钩子方法的应用。大白话就是，我在用callback语法机制时，时常是做一些任务编排的事，跟回调这个语义并不贴切，让我觉得很别扭。

  2.java的jdbc其实操作数据库也有callback语法的应用。但现在都是用的orm框架，估摸也都忘了吧，不过也确实没有记忆的必要就是了。

- 柠檬C 2020-03-19 09:29:19
  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

- Michael 2020-03-19 08:14:33
  swift和OC的闭包也属于回调

- 花郎世纪 2020-03-19 03:00:00
  深度学习pytorch框架，提供hook去获取特征层数据

- 丁乐洪 2020-03-18 21:28:07
  模板类 与 模板模式 有啥关系，感觉干的是同类活

- Heaven 2020-03-18 20:29:46
  对于Java中的Callback,常见的还是异步回调,注册一个函数之后,无需等待返回了,可以进行下一步的工作,

仿佛就是种下了一个种子,等待开花结果
对于Hook,则像是一种具体的实现手段,而且常见于AOP的代理之中

- 徐旭 2020-03-18 20:15:27
  hook也是钩子吧，好像也可以用在上层直接调底层native层

- www.xnsms.com小鸟接码 2020-03-18 15:27:40
  打卡打卡.....滴,学生卡

- dongdong 2020-03-18 15:06:50
  行为模式什么时候更新