## 模板模式与Callback回调函数有何区别和联系？

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。

### 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。

### 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？