模板方法设计模式(template method)

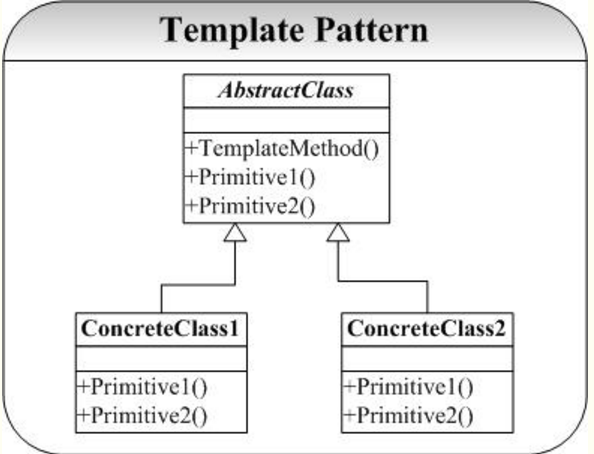
# Template Method模板方法设计模式

在面向对象系统的分析与设计过程中经常会遇到这样一种情况：**对于某一个业务逻辑（算法实现）在不同的对象中有不同的细节实现，但是逻辑（算法）的框架（或通用的应用算法）是相同的。**Template Method提供了这种情况的一个实现框架。

Template Method模式是采用继承的方式实现这一点：将逻辑（算法）框架放在抽象基类中，并定义好细节的接口，子类中实现细节。

**Strategy模式**解决的是和Template Method模式类似的问题，但是Strategy模式是将逻辑（算法）封装到一个类中，并采取组合（委托）的方式解决这个问题。

**解决这个问题可以采取两种模式来解决，一是Template模式，二是Strategy模式**。本文给出的是Template Method模式。一个通用的Template Method模式的结构图为：



Template Method模式实际上就是利用**面向对象中多态的概念**实现算法实现细节和高层接口的松耦合。可以看到Template Method模式采取的是继承方式实现这一点的，由于继承是一种强约束性的条件，因此也给Template Method模式带来一些许多不方便的地方。

Template Method模式的实现关键是将通用算法（逻辑）封装起来，而将算法细节让子类实现（多态）。唯一注意的是我们将原语操作（细节算法）定义为受保护（Protected）成员，对外部只提供模板方法供调用。

Template模式是很简单模式，但是也应用很广的模式。Template Method是采用继承的方式实现算法的异构，其关键点就是将通用算法封装在抽象基类中，并将不同的算法细节放到子类中实现。

**Template Method模式获得一种反向控制结构效果，**这也是面向对象系统的分析和设计中一个原则----**DIP（依赖倒置：Dependency Inversion Principles）**。**其含义就是父类调用子类的操作（高层模块调用低层模块的操作），低层模块实现高层模块声明的接口**。这样控制权在父类（高层模块），低层模块反而要依赖高层模块。

# 模板方法设计模式的应用

模板方法设计模式在各个框架中使用相当频繁，如Dubbo中：

Dubbo中的AbstractProxyInvoker抽象类：

*/\*\*  
 \* InvokerWrapper  
 \*/*public abstract class AbstractProxyInvoker<T> implements Invoker<T> {  
 private final T proxy;  
 private final Class<T> type;  
 private final URL url;  
  
 public AbstractProxyInvoker(T proxy, Class<T> type, URL url) {  
 this.proxy = proxy;  
 this.type = type;  
 this.url = url;  
 }  
 @Override  
 public void destroy() {  
 }  
  
 // *TODO Unified to AsyncResult?* @Override  
 public Result invoke(Invocation invocation) throws RpcException {  
 RpcContext rpcContext = RpcContext.*getContext*();  
 try {  
 **// 同样采用了模板设计模式，留给实现类去实现，如在JdkProxyFactory和JavassistProxyFactory的getInvoker方法中，  
 // 分别实现了doInovker方法  
 Object obj = doInvoke(proxy, invocation.getMethodName(), invocation.getParameterTypes(), invocation.getArguments());**  
 if (RpcUtils.*isFutureReturnType*(invocation)) {  
 return new AsyncRpcResult((CompletableFuture<Object>) obj);  
 }else {  
 return new RpcResult(obj);  
 }  
 } catch (InvocationTargetException e) {if (rpcContext.isAsyncStarted() && !rpcContext.stopAsync()) {  
 logger.error("Provider async started, but got an exception from the original method, cannot write the exception back to consumer because an async result may have returned the new thread.", e);  
 }  
 return new RpcResult(e.getTargetException());  
 } catch (Throwable e) {  
 throw new RpcException("Failed to invoke remote proxy method " + invocation.getMethodName() + " to " + getUrl() + ", cause: " + e.getMessage(), e);  
 }  
 }  
 // 模板设计模式，让实现类去实现该doInvoke方法  
 **protected** abstract Object doInvoke(T proxy, String methodName, Class<?>[] parameterTypes, Object[] arguments) throws Throwable;

}

AbstractProxyInvoker抽象类中实现了invoke方法的逻辑，而针对具体的调用细节抽象成doInvoke方法，让实现类去实现。invoke为public供外部调用，而doInvoke为protected，仅供内部即invoke方法调用。