模板方法模式

1.模板方法模式介绍

在面向对象开发过程中，通常会遇到这样的一个问题，我们知道一个算法所需要的关键步骤，并且确定了这些步骤的执行顺序，但是，某些步骤的具体实现是未知的，或者说某些步骤地实现是会随着环境的变化而改变的，例如，执行程序的流程大致如下：

1. 检查代码的正确性；
2. 链接相关的类库；
3. 编译相关代码；
4. 执行程序；

对于不同的程序设计语言，上述4个步骤都是不一样的，但是，他们的执行流程是固定的，这类问题的解决方案就是模板方法模式。

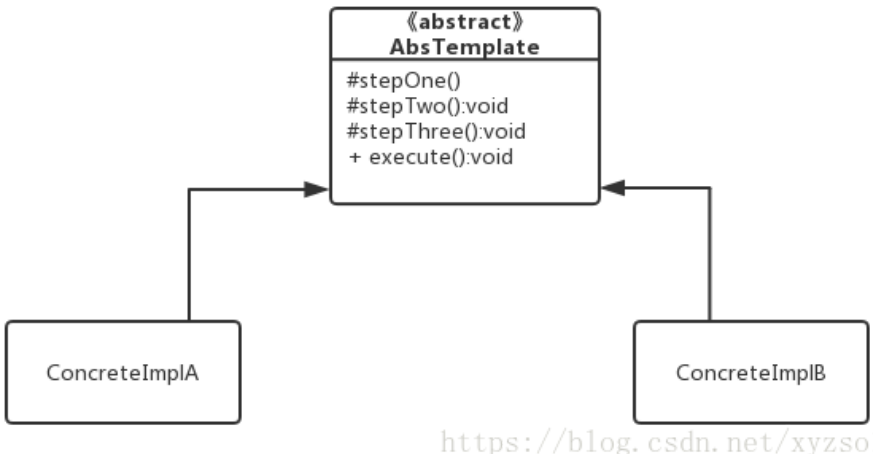
2.模板方法模式的定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中，使得自雷可以不改变一个算法的结构即可重定义该算法的步骤。

3.模板方法模式的使用场景

1. 多个子类有共有的方法，并且逻辑基本相同时；
2. 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现；
3. 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过构子函数约束其行为。

4.模板方法的UML类图



角色介绍：

- AbsTemplate:抽象类，定义一套算法框架。
- ConcreteImplA:具体实现类A。
- ConcreteImplB:具体实现类B。

5.模板方法模式简单实现示例

模板方法实际上是封装一个固定流程，就像是一套执行模板一样，第一步该做什么，第二步该做什么都已经在抽象类重定义好了，而子类可以有不同的算法实现，在框架不被修改的情况下实现某些步骤的算法替换，下面以打开计算机这个动作简单演示一下模板方法。打开计算机的整个过程都是固定的，首先启动计算机电源，计算机检查自身状态没有问题时将进入操作系统，对用户进行验证之后即可登录计算机，下面我们使用模板方法来模拟一下这个过程：

```
package templatemethod;

public abstract class AbstractComputer {
        protected void powerOn() {
                System.out.println("开启电源");
        }

        protected void checkHardware() {
                System.out.println("硬件检查");
        }

        protected void loadOS() {
                System.out.println("载入操作系统");
        }

        protected void login() {
                System.out.println("小白的计算机无验证，直接进入系统");
        }

        /*
         * 启动计算机方法，步骤固定未开启电源、系统检查、加载操作系统、用户登录。该方法为final， 防止算法框架被复写
         */
        public final void startUp() {
                System.out.println("-----开机 START-------");
                powerOn();
                checkHardware();
                loadOS();
                login();
                System.out.println("----开机END------");
        }
}
```

```
package templatemethod;

public class CoderComputer extends AbstractComputer {

        @Override
        protected void login() {
                System.out.println("程序员只需要进行用户和密码验证就可以了");
        }

}
```

```
package templatemethod;

public class MilitartComputer extends AbstractComputer {
```

```
        @Override
    protected void checkHardware() {
            super.checkHardware();
            System.out.println("检查硬件防火墙");
    }


    @Override
    protected void login() {
            System.out.println("进行指纹识别等浮渣的用户验证");
    }

}
```

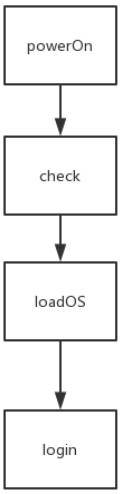
```
package templatemethod;

public class Test {
        public static void main(String[] args) {
         AbstractComputer computer=new CoderComputer();
         computer.startUp();
         computer=new MilitartComputer();
         computer.startUp();
        }
}
```

输出结果如下：

```
-----开机 START-------
开启电源
硬件检查
载入操作系统
程序员只需要进行用户和密码验证就可以了
----开机END------
-----开机 START-------
开启电源
硬件检查
检查硬件防火墙
载入操作系统
进行指纹识别等浮渣的用户验证
----开机END------
```

通过上面的例子可以看出，在startUp方法中有一些固定的步骤，依次为开启电源、检查硬件、加载系统、用户登录4个步骤，这4个步骤是计算机开机过程中不会变动，如图



但是，不同用户的这几个步骤的实现可能各不相同，因此，子类需要覆写相应的方法来进行自定义处理，这里需要注意的是startUp为final方法，这样就保证了逻辑流程不能被子类修改，子类只能够改变某一步中具体实现，这样就保证了这个逻辑流程的稳定性。startUp中这几个算法步骤我们可以称为一个套路，也称为模板方法，这也是模板方法的由来。。