结构型

结构型设计模式包括:
装饰模式、
外观模式、
组成模式（别名：合成模式）、
适配器模式（别名：包装器模式模式）、
桥接模式、
享元模式、
代理模式,
一共7种。

1.装饰模式

意图：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更加灵活。

解释：在不改变原封装的前提下，为对象动态的添加新功能的模式。
在OC中，Category和Delegate（或者说是委托）可以看作是其实现方式。
Delegate机制使一个对象可以被另一个对象修改或者扩展功能，而不需要为这个对象生成子类，这满足装饰模式的思想。

2.外观模式

意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

解释：举例说明，一般使用的第三方SDK或者开源框架时，他们一般都会有一个管理manager类，这个manager类提供这个大部分甚至所有的功能设置及使用，我们这些使用者只需要和这个manager打交道即可，而不必与这个开源框架里的每个细分子系统的manager打交道。

3.组成模式（合成模式）

意图：将对象组合成树形结构以表示“部分-整体”的层次结构，组合模式使得用户对单

个对象和组合对象的使用具有一致性。

解释：举例来说，iOS中的视图层次结构树的设计就满足组成模式的思想。

4.适配器模式（别名：包装器模式）

意图：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口欧不兼容而不能一起工作的那些类可以一起工作。

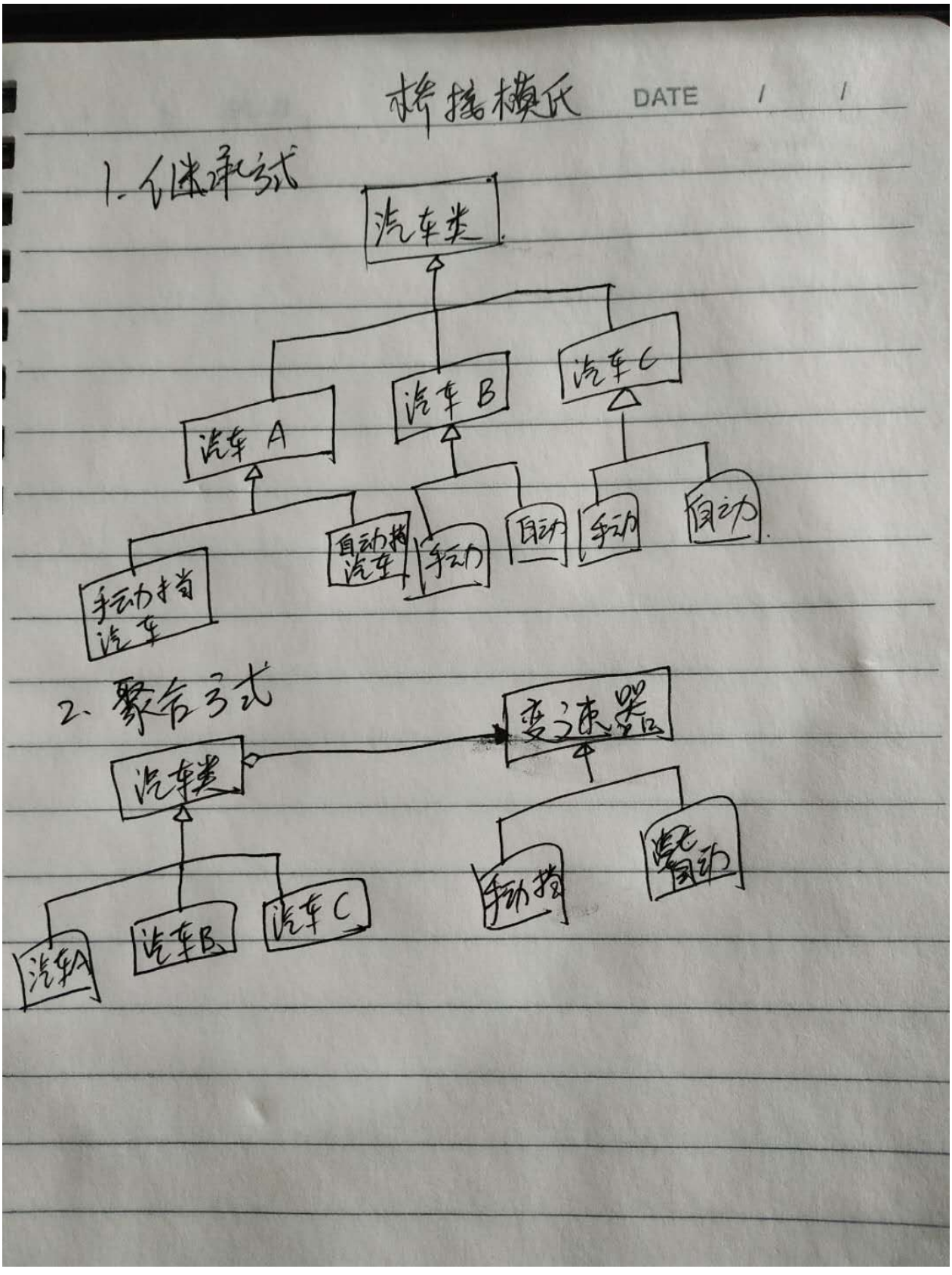
解释：适配器模式有类适配器和对象适配器两种。通过类继承来适配两个或两组接口的称为类适配器。OC中可以通过协议来实现多继承目的。通过对象的组合来适配两个或两组接口的被称为对象适配器。这是一个类适配的实例，一个示例让你明白适配器模式 - CSDN博客 (类适配器)。

5.桥接模式

意图：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

解释：桥接模式的做法是将变化部分（也就是实现部分）抽象出来，使变化部分与抽象部分（可以说是接口定义部分）分离开来，从而将多个维度的变化彻底分离。

举例来说，一个汽车抽象类可以派生出多个具体的汽车类，每个具体汽车类下面又可以派生出手动挡汽车类和自动挡汽车类，这种继承方式的结构图如下：



从上图中可以看到，每一种汽车品牌下面都需要创建一个具体类，假如再添加一个汽车品牌类，则再需要添加其手动挡和自动挡的子类。这样，类的数目太多，不利于维护和扩展。

假如使用桥接模式，将变速器抽象出来，再派生出手动挡和自动挡子类，然后汽车品牌类再去使用手动挡或者自动挡类。这样就可以将汽车品牌这一维度与汽车变速器这

一维度进行隔离，两者可以单独变化。

桥接模式就是把两个角色之间的继承关系改为聚合关系，从而使二者可以各自独立的变化。把原来在基类的实现化细节抽象出来，再构造到一个实现化的结构中，然后把原来的基类改造成一个抽象化的等级结构，这样就实现了系统在多个维度上的独立变化。

桥接模式与适配器模式的区别：
桥接模式的目的是将接口部分和实现部分分离，从而对它们可以较为容易也相对独立的加以改变。而适配器模式的目的是使用适配器来兼容现有类或对象与另外一个类或对象。

6.享元模式

意图：运用共享技术有效地支持大量细粒度的对象。

解释：这个模式就很理解了，iOS中tableViewCell和collectionViewCell的复用机制就和享元模式的思想一致。使用一个Cell的时候我们先去看看tableView的复用池中有没有可以重用的cell，如果有就用这个可以重用的cell，只有在没有的时候才去创建一个Cell。

7.代理模式

意图：为其他对象提供一种代理以控制对这个对象的访问。

解释：通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要复杂性。
代理对象起到中介作用，连接客户端和目标对象。

例子：电脑桌面的快捷方式。电脑对某个程序提供一个快捷方式（代理对象），快捷方式连接客户端和程序，客户端通过操作快捷方式就可以直接打开那个程序。

OC中的NSProxy类的设计就满足代理模式的思想。
使用NSProxy时需要继承NSProxy，然后重写下面这两个方法

```
- methodSignatureForSelector:
- forwardInvocation:
```

比如：使用NSTimer时，由于NSTimer对象会强引用target，所以在target也强引用

NSTimer对象时，势必造成循环引用问题，所以可以在target和NSTimer对象之间设置一个代理，使用代理来打破引用循环。

```
// WeakProxy.h 文件
@interface WeakProxy : NSProxy

@property (weak,nonatomic,readonly)id target;

+ (instancetype)proxyWithTarget:(id)target;
- (instancetype)initWithTarget:(id)target;

@end

// WeakProxy.m 文件

@implementation WeakProxy

+ (instancetype)proxyWithTarget:(id)target {
    return [[self alloc] initWithTarget:target];
}

- (instancetype)initWithTarget:(id)target {
    _target = target;
    return self;
}


/*
 消息转发分为三大阶段

 第一阶段 + (BOOL)resolveClassMethod:(SEL)selector; 先征询消息接收者所属的类，看其是否能动态添加方法，以处理当前这个无法响应的 selector，这叫做 动态方法解析（dynamic method resolution）。如果运行期系统（runtime system） 第一阶段执行结束，接收者就无法再以动态新增方法的手段来响应消息，进入第二阶段。

 第二阶段 - (id)forwardingTargetForSelector:(SEL)aSelector; 看看有没有其他对象（备援接收者，replacement receiver）能处理此消息。如果有，运行期系统会把消息转发给那个对象，转发过程结束；如果没有，则启动完整的消息转发机制。

 第三阶段 - (void)forwardInvocation:(NSInvocation *)anInvocation; 完整的消息转发机制。运行期系统会把与消息有关的全部细节都封装到 NSInvocation 对象中，再给接收者最后一次机会，令其设法解决当前还未处理的消息。

 */

#pragma mark - 在消息转发机制中的第2步将消息转发给 target
- (id)forwardingTargetForSelector:(SEL)selector {
    return _target;
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)selector {
```

```
    // 如果代码走到了这一步，说明 - forwardingTargetForSelector: 方法返回的target为空，也就是弱引用的target为空，这个时候需要已经是走到了消息转发的第3步，这个时候可以将selector相对应的签名替换成一个 -init方法所对应的签名，通过这种手段来防止异常。
    // We only get here if `forwardingTargetForSelector:` returns nil.
    // In that case, our weak target has been reclaimed. Return a dummy method signature to keep `doesNotRecognizeSelector:` from firing.
    // We'll emulate the Obj-c messaging nil behavior by setting the return value to nil in `forwardInvocation:`, but we'll assume that the return value is `sizeof(void *)`.
    // Other libraries handle this situation by making use of a global method signature cache, but that seems heavier than necessary and has issues as well.
    // See https://www.mikeash.com/pyblog/friday-qa-2010-02-26-futures.html and https://github.com/steipete/PSTDelegateProxy/issues/1 for examples of using a method signature cache.
    return [NSObject instanceMethodSignatureForSelector:@selector(init)];
}

- (void)forwardInvocation:(NSInvocation *)invocation {
    // Fallback for when target is nil. Don't do anything, just return 0/NULL/nil.
    // The method signature we've received to get here is just a dummy to keep `doesNotRecognizeSelector:` from firing.
    // We can't really handle struct return types here because we don't know the length.
    void *nullPointer = NULL;
    [invocation setReturnValue:&nullPointer];
}

#pragma mark - 在消息转发机制中的第3步将消息转发给 target
/*
- (void)forwardInvocation:(NSInvocation *)invocation {
    SEL sel = [invocation selector];
    if ([self.target respondsToSelector:sel]) {
        [invocation invokeWithTarget:self.target];
    }
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector {
    return [self.target methodSignatureForSelector:aSelector];
}
*/

// 在viewController中使用,别忘记在 - dealloc 方法中调用NStimer对象的 - invalidate方法，及将NStimer对象设置为nil。
self.timer = [NSTimer scheduledTimerWithTimeInterval:1
                                               target:[WeakProxy proxyWithTarget:self]
                                             selector:@selector(timerInvoked:)
```

```
                                                    userInfo:nil
                                                     repeats:YES];
```

参考材料

1.《Objective-C编程之道：iOS设计模式解析》 书籍
2.《设计模式_可复用面向对象软件的基础》 书籍
3.代理模式中的WeakProxy的实现参考 SDWebImage