**设计模式报告**

**健康打卡模块**

作者

3170104767 | 闫宇

3150100162 | 秦瀚翔

3160103826 | 贾奕斐

目录

[1 设计模式调查 3](#_Toc39392045)

[1.1 概述 3](#_Toc39392046)

[1.2 设计模式调查与分析 3](#_Toc39392047)

[1.2.1 创建型设计模式 3](#_Toc39392048)

[1.2.2 行为型设计模式 3](#_Toc39392049)

[1.3 总结 5](#_Toc39392050)

[2 健康打卡模块子系统体系结构分析 7](#_Toc39392051)

[2.1 系统关键质量 7](#_Toc39392052)

[2.2 系统体系结构 7](#_Toc39392053)

[2.3 在关键质量属性中选择的策略（tactic） 7](#_Toc39392054)

[2.3.1 性能（performance） 7](#_Toc39392055)

[2.3.2 使用性（usability） 7](#_Toc39392056)

[2.3.3 服务获得性（availability） 8](#_Toc39392057)

[3 健康打卡子系统设计模式分析 9](#_Toc39392058)

[3.1 总体设计模式 9](#_Toc39392059)

[3.1.1 UI 9](#_Toc39392060)

[3.1.2 整体 9](#_Toc39392061)

[3.2 简单工厂（Abstract Factory Pattern） 12](#_Toc39392062)

[3.3 命令模式（Command Pattern） 13](#_Toc39392063)

[3.4 责任链模式（Chain of Responsibility Pattern） 14](#_Toc39392064)

[3.5 模板方法模式（Template Method Pattern） 15](#_Toc39392065)

[3.6 观察者模式（Observer Pattern) 16](#_Toc39392066)

[3.7 React特有的设计模式 17](#_Toc39392067)

[3.7.1 函数式编程（Functional Programming） 17](#_Toc39392068)

[3.7.2 组件化设计(Component) 18](#_Toc39392069)

[4 其他可用设计模式分析 20](#_Toc39392070)

[4.1 状态模式(State Pattern) 20](#_Toc39392071)

[4.1.1 概述 20](#_Toc39392072)

[4.1.2 应用 20](#_Toc39392073)

[4.1.3 总结 21](#_Toc39392074)

[4.2 外观模式（Façade Pattern） 21](#_Toc39392075)

[4.2.1 概述 21](#_Toc39392076)

[4.2.2 应用 21](#_Toc39392077)

[4.2.3 总结 22](#_Toc39392078)

[4.3 中介者模式（Mediator） 22](#_Toc39392079)

[4.3.1 概述 22](#_Toc39392080)

[4.3.2 应用 22](#_Toc39392081)

[4.3.3 总结 23](#_Toc39392082)

[5 参考 24](#_Toc39392083)

# 1 设计模式调查

## 1.1 概述

在软件设计中，合适的设计模式可以帮助软件设计者更好的进行软件设计，使得设计变得规范。

设计模式包含很多方面比如在一类问题中的常见错误，针对这类问题的解决方案等，并且它可以使得设计者在设计时能够避免一些将会引起问题的紧耦合，增强软件设计的适应变化的能力。

设计模式通常分为创建型、结构型与行为型三种模式大类。下本也将详细写到，针对其中几种设计模式进行调查后进行的分析与总结。

## 1.2 设计模式调查与分析

### 1.2.1 创建型设计模式

创建型设计模式中我们调查了简单工厂设计模式

**（1）简单工厂模式**

简单工厂模式（Simple Factory Pattern）: 定义一个工厂类，他可以根据参数的不同，返回不同类的实例，被创建的实例通常都具有共同的父类。

简单工厂模式中包含的角色如下：

1、Factory(工厂):核心部分，负责实现创建所有产品的内部逻辑，工厂类可以被外界直接调用，创建所需对象；

2、Product(抽象类产品)：工厂类所创建的所有对象的父类，封装了产品对象的公共方法，所有的具体产品为其子类对象；

3、Concrete Product(具体产品)：简单工厂模式的创建目标，所有被创建的对象都是某个具体类的实例。它要实现抽象产品中声明的抽象方法(有关抽象类)。 分析：

在简单工厂模式中用于被创建实例的方法通常为静态(static)方法,因此简单工厂模式又被成为静态工厂方法(Static Factory Method)。需要什么，只需要传入一个正确的参数，就可以获取所需要的对象，而无需知道其实现过程

### 1.2.2 行为型设计模式

**（1）命令模式**

命令模式(Command Pattern）：该模式将一个请求封装成一个对象，从而让 你使用不同的请求把客户端参数化，同时对请求排队或者记录请求日志，并提供命令的撤销和恢复功能。

命令模式中包含的角色及其相应的职责如下：

1、Command（命令）：声明执行操作的接口。有java 接口或者抽象类来实现。

2、Concrete Command（具体命令）：将一个接收者对象绑定于一个动作；调用接收者相应的操作，以实现命令角色声明的执行操作的接口。

3、Client（客户）：创建一个具体命令对象（并可以设定它的接收者）。

4、Invoker（请求者）：调用命令对象执行这个请求。

5、Receiver（接收者）：知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

分析：

在命令模式中，我们不仅仅将命令直接封装起来提供调用，而是加入了调用者和接受者两个角色，使得一条命令将分步完成，降低耦合度，提高灵活性。并 且将命令进行多层逻辑封装，可以重用底层封装，且确保了一定的扩展性，对于客户端来说不需要知道复杂的逻辑也很便捷。

**（2）责任链模式**

责任链模式（Chain of Responsibility Pattern）：为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。

责任链模式中包含的角色及其相应的职责如下：

1、Handler（抽象处理者）：定义出一个处理请求的接口。如果需要，接口可以定义 出一个方法以设定和返回对下家的引用。这个角色通常由一个Java抽象类或者Java接口实现。上图中Handler类的聚合关系给出了具体子类对下家的引用，抽象方法handleRequest()规范了子类处理请求的操作。

　　2、Concrete Handler（具体处理者）：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

分析：

很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

**（3）模板方法模式**

模板方法模式（Template Method Pattern）：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

模板方法模式中包含的角色及其相应的职责如下：

1、Abstract Class（抽象类）：用来定义算法骨架和原语操作，在这个类里面，还可以提供算法中通用的实现

2、Concrete Class（具体实现类）：用来实现算法骨架中的某些步骤，完成跟特定子类相关的功能。

分析：

通过把不变行为搬移到超类。去除子类中的反复代码来体现它的优势。模板方法模式就是提供了一个非常好的代码复用平台，当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就会在子类中反复出现。我们通过模板方法模式把这些行为搬移到单一的地方。这样就帮助子类摆脱反复的不变行为的纠缠。

**（4）观察者模式**

观察者模式(Observer Pattern):该模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

观察者模式中包含的角色及其相应的职责如下：

1、Subject（抽象主题）:抽象主题角色把所有对观察者对象的引用保存在一个集合（比如Vector）里，然后每个主题都可以有任何数量的观察者。抽象主题提供的接口有：增加观察者对象、删除观察者对象、通知所有的观察者对象、获取观察者数量、设置改变。

　　2、Concrete Subject（具体主题）: 将有关状态存入具体主题对象，在具体主题的内部状态改变时，给所有注册过的观察者发送通知。

　　3、Observer（抽象观察者）:为所有的具体观察者定义一个接口，订阅的主题改变时更新自己，这个接口叫做更新接口。

　　4、Concrete Observer（具体观察者）:存储与主题自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态向协调。如有需要，具体观察者角色可以保持一个指向具体主题对象的引用。

分析：

一个软件系统中常常要求在某一个对象的状态发生变化的时候，某些和它关联的其他的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计时需要使这些低耦合的对象之间能够维持行为的协调一致，保证高度的协作。观察者模式是满足这一要求的各种设计方案中最重要的一种。

## 1.3 总结

对所分析的5种设计方法的优缺点及应用的总结：

**简单工厂模式**

优缺点：

1、良好的封装性

2、解耦框架

3、扩展性

应用：

调用者清楚应该使用那个具体工厂的服务。

**命令模式**

优缺点：

1、比较容易设计一个命令队列

2、允许接受请求的一方是否处理请求

3、可以容易的实现对请求的添加和删除

4、可能会导致系统有过多的具体命令类

应用：

需要支持命令的撤销和恢复操作。

**责任链模式**

优缺点：

1、将请求的发送者和接收者解耦

2、可以简化你的对象，不需要知道链的结构

3、通过改变链内的成员或调动他们的次序，允许你动态地新增或删除责任

4、不能保证请求一定会被执行

5、可能不容观察运行时的特征

应用：

当算法牵涉到一种链型运算，而且不希望吹过程中有过多的循环和条件选择语句，并且希望比较容易的扩充文法。

**模板方法模式**

优缺点：

1、把不变的行为搬移到超类

应用：

通过模板方法可以把不变和可变的混合行为搬到单一地方，摆脱纠缠。

**观察者模式**

优缺点：

1、观察者和被观察者可以独立改变

2、松耦合导致代码关系不明显

3、广播时会有效率问题

应用：

对一个对象状态的更新，需要对其他对象同步更新，而且其他对象的数量动态可变时可用观察者模式。

# 2 健康打卡模块子系统体系结构分析

## 2.1 系统关键质量

依据本子系统的软件需求分析说明书（SRS），本子系统的关键质量属性（QA）为性能（performance）、使用性（usability）和服务获得性 （availability）。其中在性能方面，操作响应时间有明确的下界需求：

1、界面显示在0.5s以内

2、用户操作响应时间在0.5s以内

## 2.2 系统体系结构

客户端：浏览网页主要采用IE7.0以上浏览器或者Chrome。使用react库中的视图组件，简化客户端的设计，增加客户端的美观性。

JavaScript脚本：客户端静态页面中，各种文本框与按键的操作均能触发脚本函数，脚本通过fetch（），与服务器进行异步交互。并且对象接受服务器反馈信息后，能通过脚本函数对客户端静态页面实现重新渲染。

JavaScript访问/修改数据库：此模块通过url参数方式从客户端得到数据与命令，并且对其进行安全检测，而后按要求访问/修改数据库，并且返回操作结果或是查询结果。

## 2.3 在关键质量属性中选择的策略（tactic）

### 2.3.1 性能（performance）

（1）Tactic 1

Introduce Concurrency（并发），毫无疑问，引入并发是提高性能必不可少的策略.

（3）Tactic 2

Reduce overhead（减少开销），减少开销包括减少计算开销和减少通信开销 两方面。

应用：一些格式匹配的工作可以在浏览器端直接进行验证， 而无须在服务器端进行，以减少网络通信所带来的开销。

### 2.3.2 使用性（usability）

（1）Tactic 1

Cancel（取消），取消策略几乎应用在了所有的体系结构设计中。

### 2.3.3 服务获得性（availability）

#### 2.3.3.1 概述

衡量指标：a = MTBF /（MTBF + MTTR）

MTBF：Mean time between failure，即平均故障时间。

MTTR：Mean time to recover，即平均恢复时间。

#### 2.3.3.2 策略

（1）Tactic 1

Exception Detection（异常检测），异常检测包括操作系统异常，常数异常等情况。

（4）Ttactic 4

State Resynchronization（状态同步），在被动冗余的情况下，状态的同步操作一般由主服务器定期更新备份服务器的数据来完成。

# 3 健康打卡子系统设计模式分析

## 3.1 总体设计模式

### 3.1.1 UI

App使用了以React Native为基础的Expo平台作为UI，不少人认为React Native采用了MVVM设计模式或是MVC设计模式。其实仅仅只考虑React Native，它并不属于这两种模式。实际上，无论是MVVM，还是MVC，或者是MV什么，关键在于分层，或者说是任务的分离。

React Native所基于React最初的提出是为了方便创建交互式UI。React采用组件化的设计，创建拥有各自状态的组件，再由这些组件构成更加复杂的UI。React组件接收输入的数据并返回需要展示的内容。当组件的状态数据改变时，组件会重新渲染对应的标记。并且React Native的渲染方法是使用了原生代码实现，可以保证App的运行效率。

实际上，我们可以这样理解：React Native所基于的React是数据驱动的，它的内容包括了数据、函数和结果。对于React，数据是应用程序的状态，函数是React组件，结果是UI。没有单独的视图，也不存在分层。有人将React Native的设计模式称为“函数化的前端架构(Functional Front-End Architecture)”，从本质上讲，它是一个以拥有状态的函数实现的UI(The UI as a function of its state)（Yassine Elouafi, 2015）。这些将在3.7中详细介绍。

当然并不是说React Native无法实现MVC设计模式，查阅资料得知React Native的创建者Facebook提供了Flux这一模式，实现了分层。不仅如此，Flux为了解决MVC中双向交流（Bidirectional Communication）带来的难以维护和debug的问题，还在传统MVC模式上进行了一些改进。

React Native正处于发展期，技术迭代非常快。Redux是基于Flux设计模式的最出名的技术，然而近年来似乎并没有受到开发者们的青睐。随着第三方的Navigation组件的兴起，Facebook停止维护Navigation这一重要组件，而Navigation这一组件对于手机App开发也可以说是必要的。又因为Redux将状态参数分离，给第三方Navigation的开发带来了很大的困难，最有名的、也是目前官方推荐的React Navigation在2018年秋季发布时表示不再特意考虑Redux的兼容性，并且认为如果不清楚自己到底要利用Redux干什么，使用Redux反而会带来额外的不必要的问题。而根据项目的统一要求，我们也并没有在前端利用Redux实现MVC设计模式，否则将会给项目整合徒增困难。

### 3.1.2 整体

虽然单独考虑前端的话，不属于MVC设计模式，但是如果从整体上来看我们采用了MVC的设计模式。

MVC模式（Model-View-Controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC模式最早由Trygve Reenskaug在1978年提出，是施乐帕罗奥多研究中心（Xerox PARC）在20世纪80年代为程序语言Smalltalk发明的一种软件设计模式。MVC模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式通过对复杂度的简化，使程序结构更加直观。软件系统通过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。专业人员可以通过自身的专长分组。

Expo + Express的构架中蕴含着MVC的思想，但是也并不是严格按照MVC的标准进行布局的，下面从实际情况从模型、视图和控制器进行分析：

1. 模型（Model）用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。Model有对数据直接访问的权力，例如对数据库的访问。Model不依赖View和Controller，也就是说，Model不关心它会被如何显示或是如何被操作。但是Model中数据的变化一般会通过一种刷新机制被公布。

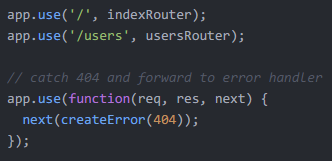
在项目中，我们使用MySQL作为数据库。基础模型的设计非常灵活，甚至不存在类或者额外的模型定义，采用了JSON的格式进行规范化处理，从数据库获取必要的信息后储存在JSON中。

如下图所示，我们与MySQL数据库建立连接后，可以直接使用SQL语句从数据库获取JSON数据，直接返回给UI端，也就是View层，用于显示。



1. 控制器（Controller）作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

Controller主要依赖express服务器端的路由控制实现。框架基于模块和操作的方式进行访问，由于该框架的应用采用单一入口文件来执行，因此所有的模块和操作都通过URL的参数来访问和执行。这样一来，访问会变成由URL的参数来统一解析和调度。



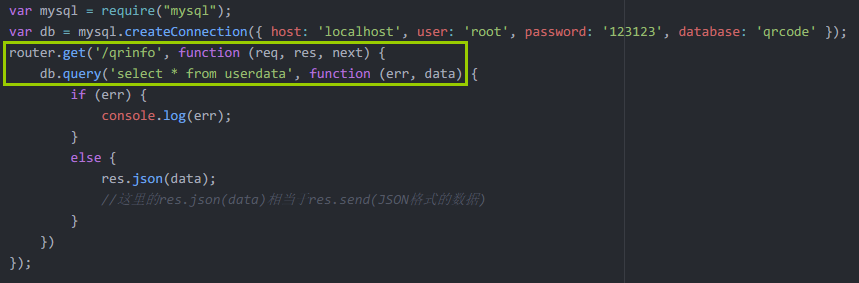
定义的模块和操作如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 行为 | 模块 | 操作 | 普通用户 | 管理员用户 |
| 获取二维码信息 | healthcode | qrinfo | 1 | 1 |
| 获取健康信息 | healthcode | healthinfo | 1 | 1 |
| 提交每日打卡 | dailyreport | submit | 1 | 1 |

例如获取二维码信息时，使用fetch方法访问下面这个url：

<http://server_address:port/healthcode/qrinfo>

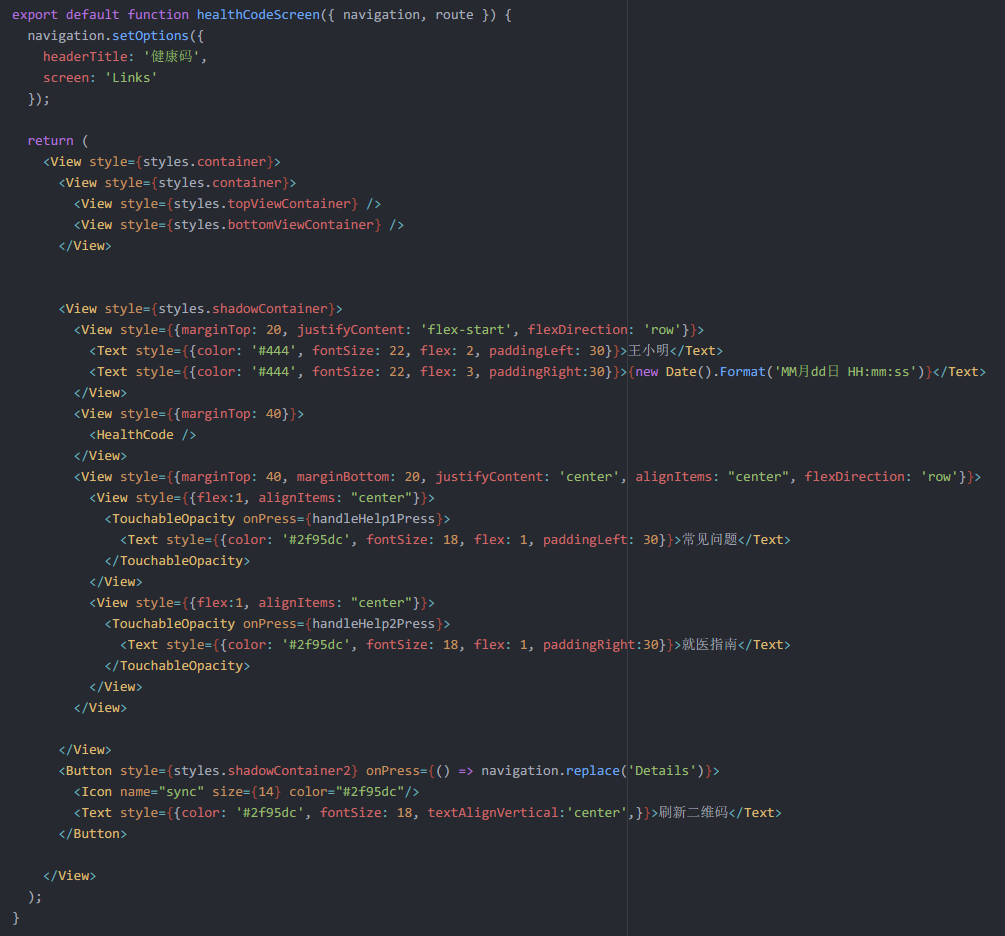
控制器就会通过中间件执行select操作查询数据库，获取生成健康码所需信息，并把结果返回给相应的视图。



1. 视图（View）能够实现数据有目的的显示。在View中一般没有程序上的逻辑，但是要实现View的刷新。

这里，我们将UI整体作为View。正如上文提到的，React Native的核心思想是组件化，而其组件是拥有状态的函数，也就是说React Native中界面作为函数出现，其返回的结果就是JSX或者ES6表示的UI。Controller将输出的数据通过JSON的方式传给View的函数，而具体的输出工作则交由View来实现。

健康码界面如下图所示：



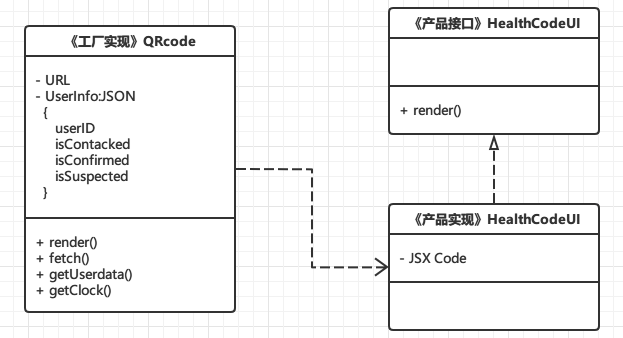
## 3.2 简单工厂（Abstract Factory Pattern）

目的/问题：创建一个对象常常需要复杂的过程，所以不适合包含在一个复合对象中。创建对象可能会导致大量的重复代码，可能会需要复合对象访问不到的信息，也可能提供不了足够级别的抽象，还可能并不是复合对象概念的一部分。工厂方法模式通过定义一个单独的创建对象的方法来解决这些问题。由子类实现这个方法来创建具体类型的对象。

简介：简单工厂模式（Simple Factory Pattern）属于类的创新型模式，又叫静态工厂方法模式（Static Factory Method Pattern）是通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。简单工厂模式中包含的角色及其相应的职责如下：

1. 工厂角色（Creator）：这是简单工厂模式的核心，由它负责创建所有的类的内部逻辑。当然工厂类必须能够被外界调用，创建所需要的产品对象。
2. 抽象产品角色（Product）：简单工厂模式所创建的所有对象的父类，注意，这里的父类可以是接口也可以是抽象类，它负责描述所有实例所共有的公共接口。
3. 具体产品角色（Concrete Product）：简单工厂所创建的具体实例对象这些具体的产品往往都拥有共同的父类。

示例：例如我们生成健康码的UI，因为生成的对象包含的元素比较复杂，我们使用简单工厂模式来实现。以此来降低对象之间的耦合度，另外工厂模式是依靠抽象架构的，它把实例化产品的任务交由实现类完成，扩展性比较好。如下图所示：



## 3.3 命令模式（Command Pattern）

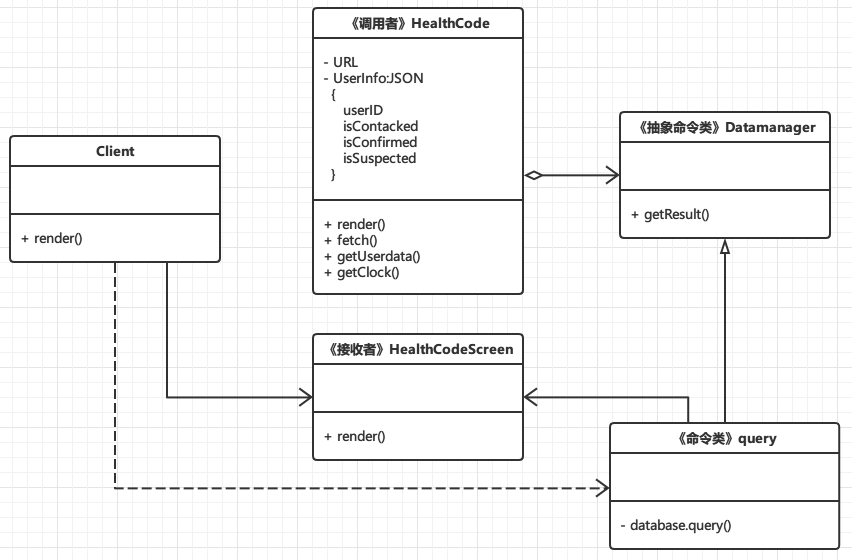
目的/问题：它尝试以对象来代表实际行动。命令对象可以把行动(action)及其参数封装起来，于是这些行动可以被：重复多次；取消（如果该对象有实现的话）；取消后又再重做

简介：命令模式将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

顾名思义，命令模式就是对命令的封装，首先来看一下命令模式类图中的基本结构：

1. Command类：是一个抽象类，类中对需要执行的命令进行声明，一般来说要对外公布一个execute方法用来执行命令。
2. ConcreteCommand类：Command类的实现类，对抽象类中声明的方法进行实现。
3. Client类：最终的客户端调用类。
4. Invoker类：调用者，负责调用命令。
5. Receiver类：接收者，负责接收命令并且执行命令。

示例：我们仍然以健康码为例，这里我们最终的目的是执行一条SQL语句，并将这一结果返回给健康码界面。当客户端需要获取健康码时，调用者HealthCode类通过DataManager向服务器发送一个请求，而这个请求由服务器端的命令类来实现，并且执行一个SQL，接收者将返回的信息生成健康码。命令模式作为一种行为类模式，首先做到了对命令的封装，然后要做到低耦合，进而提高项目的灵活性。当然这一核心功能还是借助MySQL来实现的，否则我们每一次操作数据库都需要实现一个操作数据库的方法。

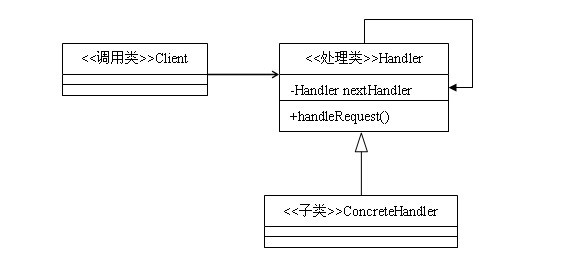


## 3.4 责任链模式（Chain of Responsibility Pattern）

目的/问题：传统代码的业务逻辑是这样的，方法有两个参数：整数i和一个请求request，根据i的值来决定由谁来处理request，如果i==1，由Handler1来处理，如果i==2，由Handler2来处理，以此类推；或者采用if…else if…else的结构。然而这会导致两个问题：

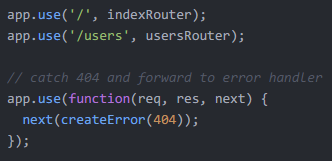
1. 代码臃肿：实际应用中的判定条件通常不是这么简单地判断是否为1或者是否为2，也许需要复杂的计算，也许需要查询数据库等等，这就会有很多额外的代码，如果判断条件再比较多，那么这个if…else…语句基本上就没法看了。
2. 耦合度高：如果我们想继续添加处理请求的类，那么就要继续添加else if判定条件；另外，这个条件判定的顺序也是写死的，如果想改变顺序，那么也只能修改这个条件语句。

介绍：责任链模式使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。类图如下所示：



示例：express服务器端的路由控制的实现就是责任链模式。框架基于模块和操作的方式进行访问，由于该框架的应用采用单一入口文件来执行，因此所有的模块和操作都通过URL的参数来访问和执行。这样一来，访问会变成由URL的参数来统一解析和调度。

例如，如下图所示，URL匹配到URL则由相应的中间件类来处理，或者最后没有可匹配路由则返回404(http状态中用于表示page not found)信息。在实际使用过程中也要注意不能把通配路由放在最前面，否则无论访问什么URL都会返回404信息。



## 3.5 模板方法模式（Template Method Pattern）

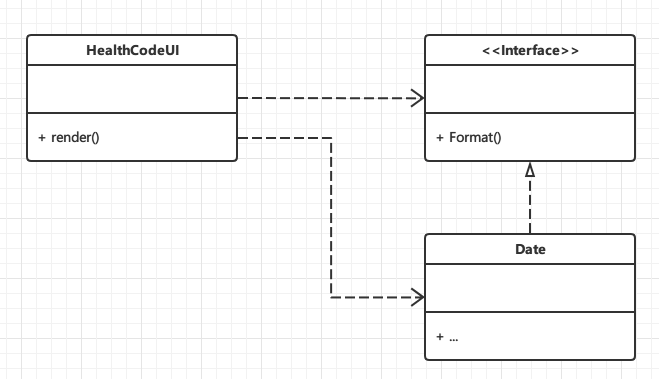
目的/问题：使用模板方法模式可以解决重复代码、代码臃肿问题，并且有利于代码分工。

介绍：模板方法模式定义一个操作中算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变算法的结构即可重定义该算法中的某些特定步骤。一个复杂的任务，由公司中的项目主负责人将主要的逻辑写好，然后把那些看上去比较简单的方法写成抽象的，交给其他的同事去开发。这种分工方式在编程人员水平层次比较明显的公司中经常用到。

示例：例如在表示健康码界面表示时间时，我们对Date类进行扩展使其能够按照一定的格式转化为字符串，这样方便UI类直接调用：



类图如下所示，我们定义了一个接口实现了Format的方法，HealthCode界面类可以直接使用：



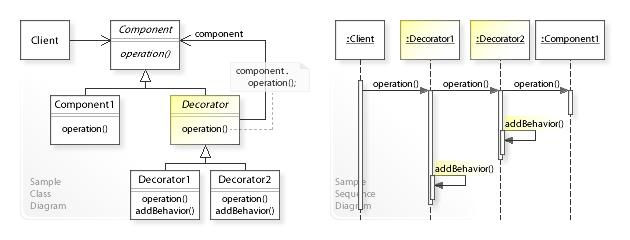
## 3.6 观察者模式（Observer Pattern)

目标/问题：一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其他的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作。观察者模式是满足这一要求的各种设计方案中最重要的一种。

简介：观察者模式是对象的行为模式，又叫发布订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependent)模式。

观察者模式定义了一种一对多的依赖关系（如果需要），让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

示例：用的React Native所构建的组件都是使用这一模式的：React采用组件化的设计，创建拥有各自状态的组件，再由这些组件构成更加复杂的UI。React组件接收输入的数据并返回需要展示的内容。当组件的状态数据改变时，组件会重新渲染对应的标记。并且React Native的渲染方法是使用了原生代码实现，可以保证App的运行效率。



## 3.7 React特有的设计模式

除了上文提到的整体的设计模式和著名的GoF设计模式，我想谈一谈React Native所基于的React特有的设计模式。正如上文提到过的，有人将React Native的设计模式称为“函数化的前端架构(Functional Front-End Architecture)”，从本质上讲，它是一个以拥有状态的函数实现的UI(The UI as a function of its state)。

### 3.7.1 函数式编程（Functional Programming）

问题/目的：

1. 提高了代码的可读性
2. 提高了代码的灵活性
3. 有助于代码的重用

简介：在函数是编程中，函数是第一类对象（First class citizens），意思是说一个函数，既可以作为其它函数的参数（输入值），也可以从函数中返回（输入值），被修改或者被分配给一个变量。并且尽可能保证参数不变性，并不是说参数不可变，而是用新的替代旧的。

示例：如下图所示，我们定义了一个fetchData的函数，用于从服务器获取健康码信息，除此之外，我们可以看到一些函数直接作为对象存在，这里有函数用于改变state（这一示例并不够精妙，使用函数式编程能够实现很多意想不到的操作，这里只能当作参考）。另外还可以利用函数对象生成比较复杂的内容并返回，或是用函数套函数，这些在React Native都是常规操作。让代码更加灵活，这一点我们在开发过程中深有体会，一个语句可能有很多很多写法。最新的React Native标准中随着React's Hooks API的引入，推荐用function作为一个组件，取代了之前的class地位。据官方文档所说，function组件更为强大，将函数是第一类对象的思想进一步扩展（超越了class）。

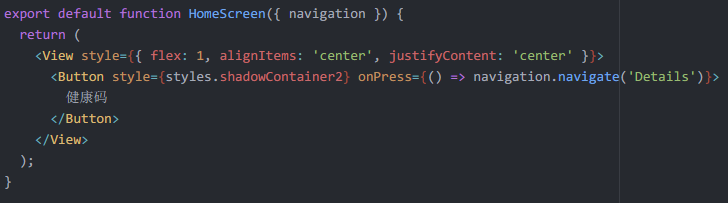


### 3.7.2 组件化设计(Component)

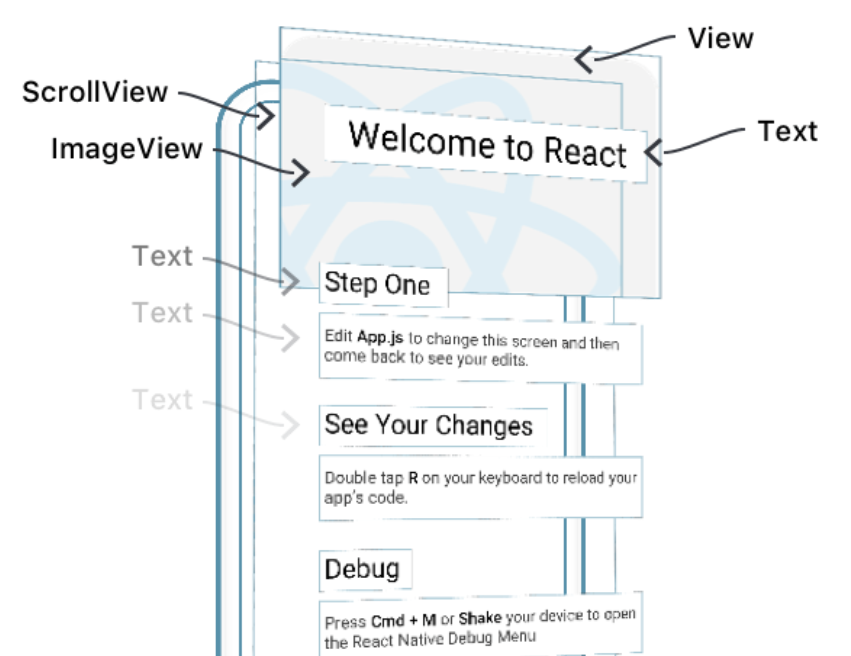
目的/问题：在以往的开发中，会把UI页面所有的复杂控件作为一个单一的整体进行开发，由于控件之间需要进行通信，因此不同的组件之间的耦合度会很多，由于开发一个控件的时候要考虑到控件与控件之间的联系，这么一来，随意一个复杂控件的代码思想就很复杂。

简介：React的出现，很好地解决了这个问题，React的核心思想就是组件化，它会把页面中的组件一个个进行分解，按照功能要求，大组件可以分解成小组件，小组件还可以分解成更小更简单的组件，各个组件维护自己的状态和UI，当状态发生改变时，会自动重现渲染整个组件，多个组件一起协作共同构成了React应用。组件化的开发思想中，可以将组件对应一个类，类与类之间的组合实现了解耦，而且类中将界面逻辑和业务逻辑统一在一起又实现了聚合，逻辑很直观。

示例：如主界面一个function就是一个组件：



其中由很多小组件构成，我们可以参考React Native官方网站的这张图来更加直观的理解React Native的组件：



我们结合函数化编程和组件化设计能够实现一些具有特殊功能的组件，例如闪烁、实时刷新、计时器等等。

# 4 其他可用设计模式分析

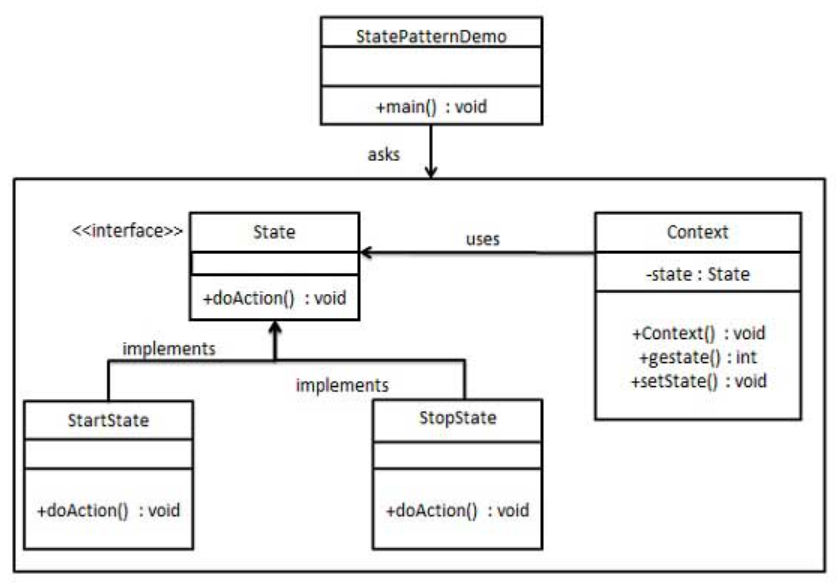
## 4.1 状态模式(State Pattern)

### 4.1.1 概述

在状态模式中，类行为根据其状态而变化。这种类型的设计模式属于行为模式。在状态模式中，我们创建代表各种状态的对象以及上下文对象，其行为随其状态对象的变化而变化。

实现：我们将创建一个定义操作的State接口和实现State接口的具体状态类。 Context类是承载状态的类。演示类StatePatternDemo将使用Context和状态对象根据所处状态的类型来演示Context行为的变化。

* 环境类（Context）: 定义客户感兴趣的接口。维护一个 ConcreteState 子类的实例，这个实例定义当前状态。
* 抽象状态类（State）: 定义一个接口以封装与 Context 的一个特定状态相关的行为。
* 具体状态类（ConcreteState）: 每一子类实现一个与 Context 的一个状态相关的行为。



### 4.1.2 应用

根据用户是否确诊，健康打卡模块处于不同的状态。在用户没有确诊的状态下，每日健康打卡界面隔离开启，信息界面关闭，用户无法进入隔离信息界面填写信息。而在用户确诊并隔离的状态下，用户允许进入隔离信息界面填写信息，但无法进入每日健康打卡界面。使用状态模式可以使模块的行为取决于它的状态，不同状态下的处理不依赖于其他状态，独立变化。

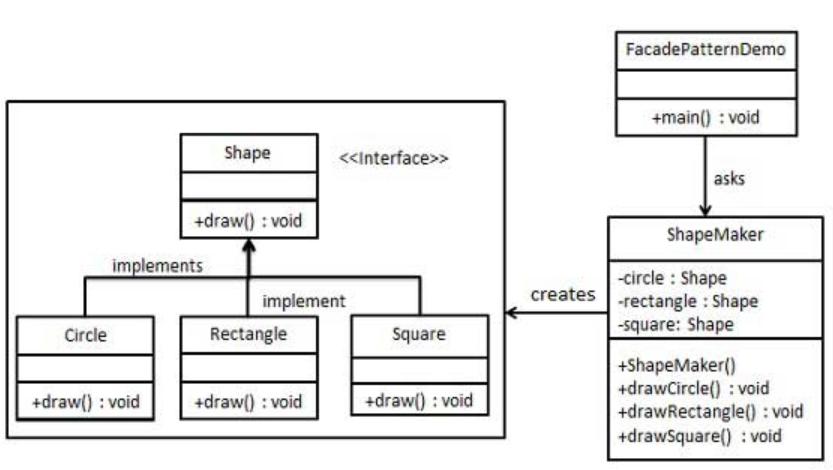
### 4.1.3 总结

对于不同状态下的用户操作，状态模式局部化了与特定状态相关的行为，并将不同状态的行为分割，简化新状态的添加。状态模式简化了条件判断的语句，使编码结构化，便于调试。为不同的状态引入独立的对象使得状态转换变得更加明确。但状态模式的结构与实现都较为复杂，使用不当将导致程序结构和代码的混乱。

## 4.2 外观模式（Façade Pattern）

### 4.2.1 概述

外观模式隐藏了系统的复杂性，并提供了到客户端的接口，客户端可以使用该接口访问系统。这种类型的设计模式属于结构模式，因为该模式向现有系统添加了接口以隐藏其复杂性。该模式涉及单个类，该类提供了客户端所需的简化方法，并将调用委托给现有系统类的方法。



### 4.2.2 应用

对于不同种类的用户（未隔离用户，隔离用户，管理员）提供统一的登录界面，通过统一的接口层将用户导向到不同的子模块。各种不同子功能使用单独的子模块完成，例如为未隔离用户的每日健康打卡以及被隔离用户的隔离信息填报分别定义单独的子类，共同实现一个每日信息填报接口。

### 4.2.3 总结

使用外观模式可以有效提高子模块独立性，简化整个组件的接口，将客户程序与子系统解耦，避免子程序的变化直接影响客户程序。在层次化结构中，可以使用外观模式定义每一层的入口。

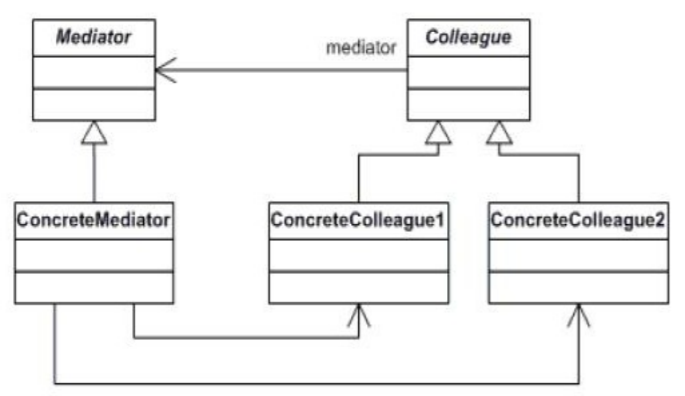
## 4.3 中介者模式（Mediator）

### 4.3.1 概述

中介者模式用于降低多个对象或类之间的通信复杂性。此模式提供了一个中介者类，该类通常处理不同类之间的所有通信，并通过松散耦合来支持代码的轻松维护。中介者模式属于行为模式类别。

中介者模式中包含的角色及其相应的职责如下：

* 抽象中介者（Mediator）角色：抽象中介者角色定义统一的接口用于各同事角色之间的通信。
* 具体中介者（Concrete Mediator）角色：具体中介者角色通过协调各同事角色实现协作行为。为此它要知道并引用各个同事角色。
* 同事（Colleague）角色：每一个同事角色都知道对应的具体中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。



### 4.3.2 应用

在健康打卡模块中，管理员用户可以作为具体中介者，未隔离用户和已隔离用户则作为两个具体同事。未隔离用户和已隔离用户继承自一个统一的抽象同事接口，管理员则继承自一个抽象中介者。

两类用户的扫码操作必须通过管理员协作。例如，未隔离用户想要进入特定场所，需要出示健康码，当管理员扫码验证用户健康状况后才能进入。同样，检查被隔离用户隔离信息，也需要管理员扫码验证。

### 4.3.3 总结

中介者模式主要用来封装行为，使行为的参与对象不必知道对方行为的实现细节。中介者类可能由于包含同事之间交互细节而变得非常复杂，使系统难以维护。因此使用中介者模式必须考虑对象之间引用是否复杂，对象是否有复用的要求，并在适当的情况下通过其他方法实现相同的目的。一旦使用中介者模式，对于中介者类应该仔细设计，避免它不能正常工作对其他类造成的影响。

# 5 参考

[1] Gamma E, Helm R, Johnson R, John V. 设计模式可复用面向对象的软件基础[M]. 机械工 业出版社, 2007.

[2] 维基百科 http://zh.wikipedia.org/wiki/Wikipedia

[3] 百度百科 http://baike.baidu.com/

[4] 百度文库 https://wenku.baidu.com/