# HHP设计思想

# 特点

## 纯面向对象

本框架及其应用模块，没有任何除“类”以外的全局数据：没有用define定义常量，甚至连配置文件都是用return而没有用变量（参见设计思想/配置文件章节）。因为“类”，顾名思义还有分类的意思，那么任何全局变量都可以归类作为一个“类”的常量。

建议开发者不要使用全局数据。

## 使用名字空间

本框架及其应用模块所有类均有名字空间。其好处：

1. 所有高级语言(java、c++等)都支持名字空间（老的PHP程序员可能觉得名字空间没多大用处，但大方向总是没错的）。
2. 每个类的依赖关系，在文件一开始处的use集合就指明了。
3. 可以通过autoload方式，实现lazy load一个类的文件，对于PHP这种进程式的语言来说，提高了效率。
4. 其他好处跟其他语言里的名字空间一样。

建议开发者也使用名字空间。

## 禁止使用include

参见：[明确的“模块之间依赖关系](#_明确的模块之间依赖关系)”。

## 系统单位：模块

我们认为模块是一个应用程序的基本组成单位，比模块更小的单位就只有类！模块与模块之间只有依赖（调用）的关系，没有从属（父子）关系。

为什么这么设计呢？我们知道函数是现代语言中最小的可复用单元，之上是“类”，但“类”提供的功能有限：她只提供了某一类别的功能，因为我们不可能把类设计得更加复杂，因为这样增加了耦合关系。但现实中我们往往需要复用更加复杂的功能，比如想要一个用户管理功能，包括注册、登录、添加用户、用户列表、删除用户等。这时，我们需要编写不同的类来提供一个更加完整的功能，而这个集合可能（也最好能够）被放入其他应用程序中。我们把这个集合叫模块。我们不把模块的集合再作定义，主要是不想系统中的定义太多，耦合增加。

当然，模块与模块之间的文件存储位置可以是从属的，即一个模块的代码放入另一个模块文件夹里，但其功能从属关系并不是父子的。

## 明确的“模块之间依赖关系”

上面说了模块之间的依赖关系只有调用了，且模块很可能被拷贝到其他应用程序中，那么如何得知该模块的依赖关系就显得很重要了。如果不知道该模块所依赖的其他模块，当该模块被拷贝到其他应用程序之后，可能依赖的模块不存在了而丧失某些功能。这对于像PHP这种解释执行的语言来说，是其一个缺点，不像编译性可以用工具查看其依赖（比如linux上的so文件，可以用ldd命令查看她依赖的其他库），且在程序启动时，会检查依赖，如果依赖不存在，程序根本无法启动。

### 解决方案

1. 在模块配置文件中指明该模块需要用到的模块及路径。
2. 应用模块，**除了启动程序外，其他地方禁止使用include，用“use 名字空间”代替**。

### 处理过程

当要用到某个文件中定义的类时，由于没有include该类的定义，php解释器就会调用注册的\_\_autoload函数（即框架的autoload函数），autoload函数会根据类的名字空间解析出该类对应的模块，再在模块配置文件中找到该模块对应的路径，再调用include语句把类定义文件include进来。这样该类的定义就找到了。

注：每个类都有其名字空间，且必须是模块对应的路径，比如“用户”模块有个类Login在user/model/Login.php里，那么class Login的名字空间就应该是user\model。当要用到user\model\Login时，就直接用use user\model\Login就OK了。

# 阅读顺序

从下面开始，顺序阅读。

# 背景

好多框架太复杂，而且没有设计文档，造成学习成本高，用起来不自由。

# 目的

设计一个框架，目的肯定是为了解决一些问题的。本框架就是为了解决：MVC架构（多重MVC框架）、常用类的集合（服务）、模块之间的依赖（事件机制）等。

还有个目的就是把设计思想传播出去，使得每个用框架的人都能成为框架的主人，随心所欲地用。

# 设计思想

## 需要解决的问题

设计一个框架的目的，就是解决遇到的问题，要不然框架设计时，就会盲目，不知道从哪儿入手开始设计。这也符合软件设计的原则--从接口入手。下面就列举了问题：

### 路由（找到Controller::Action）

ControllerRouter这个不多讲了，就是怎么把一个请求解析对应到一个Controller的一个方法（方法我们成为Action），并执行。

所以，框架就是一个ControllerRouter。

### 层次结构（嵌套MVC）

#### 嵌套MVC简介

上面解决了Controller的路由问题，但是我们通常所说的MVC架构（即把MVC三个层次分开，使开发者更专注，系统结构更清晰）的三个层次Model、Controller、View远远不够，在实际使用中，根据业务逻辑的需要，这三个层次的每一层都可以再次划分出MVC三个层次。

##### 示例：ControllerRouter调用Controller

要求：框架由统一的index.php调用所有的Controller。

处理过程：ControllerRouter根据用户的请求，决定调用哪个类的哪个方法，即用户访问的是ControllerRouter，ControllerRouter知道要调用哪个类，所以ControllerRouter和Controller又组成了MVC里的一个MC。

顶层MVC归属：ControllerRouter之后才到Controller，所以，不管怎么说，都应该归属为Controller层。

##### 示例：显示用户信息

要求：对敏感信息如手机号、银行卡号等不能完全显示。

处理过程：Controller调用Model返回用户信息数据（可能是一个User对象），把数据交给View显示，但是中间需要另外一层：对数据的显示进行处理--我们把这层叫做数据过滤层：Filter。可以把Filter归为Model层，也可以归为View层，总之，Controller还是原来的Controller，View还是原来的View，只是Controller里多了调用相应的Filter处理Model返回的数据的功能。这时，Controller、Filter、View构成新的MVC。

顶层MVC归属：根据功能划分类别原则，因为Filter主要为View服务，所以我们一般把Filter归为View层。

##### 示例：检查访问权限

要求：只有拥有相应权限的人才能访问相应的Controller。

处理过程：在调用Controller之前，调用相应的Permission层来检查用户是否拥有访问权限，有框架决定访问哪个Permission（就像决定访问哪个Controller一样），而Permission决定要不要访问Controller，所以，Permission和Controller组成了MVC里的MC。

顶层MVC归属：同样，Permission之后才到Controller，所以……

#### 对象调用顺序

所以，在执行Controller之前，可能还会执行别的层次的一些类；在执行完Controller之后，还会执行别的层次的一些类。

我们用IExecuter接口来定义MVC的每个层次，配置文件中用preexecuters和later\_excuters指定（App其实也是一个IExecuter，只是没有明确写明）。这两个配置都是数组，存放继承IExecuter的类名。App会读取这两个值，依次执行其run函数。

### 每个Action都可以有不同配置文件

又有问题来了--试着考虑这样一个问题：把登录和添加用户的操作放同一个模块用户模块下，添加用户是需要鉴权的操作（至少应该判断是否登录），而登录操作不需要。所以登录操作需要的preexecuters可能就是array(‘hhp/InputChecker)，而添加用户操作需要的preexecuters是array(‘user/Login’ , ‘hhp/InputChecker)。

本框架的解决方法是：将要执行的Controller类里提供一方法，返回新的配置，配置中指明新的preexecuters。详情参看“配置文件”章节。

## 初始化

既然是纯面向对象了，那么当框架接受到一个请求时，应该立即启动一个对象去处理这个请求，再根据上面所说的，那么这个对象就是ControllerRouter了。但是，框架其实就是一个应用程序，应该叫App，既然这样，App就是ControllerRouter，实现了ControllerRouter的功能。

App由接受请求的文件实例化，并执行run函数，该函数参数为模块的配置（参见类说明文档）。

## 配置文件

框架分为三种配置：

1. 系统配置，由系统根目录下的config目录下的Config.php文件指定，config目录与hhp目录同级。
2. 模块配置，由每个模块根目录下的config目录下的Config.php文件指定。
3. 每个Controller的Action的配置：由Controller类提供getConfig($actionName)方法，App在执行所有preexecuters之前取得。

三种配置的配置项目和代表的意思完全一样，覆盖顺序为：Action->模块->系统。在App启动时，Action的配置会和模块的配置合并。可以在运行时刻通过App的实例的getConfig($configName)获取，该方法会先找合并后的模块配置，找不到就找系统配置。

配置文件是一个数组，用return返回。之所以没有用变量，是因为用变量就容易造成全局变量，而用return后，调用者只能用一个变量来接收，比如$conf = require\_once( ‘config.php’ )，让包含配置文件的人自己决定变量的范围。设计之初是想把配置作为一个类，但后来发现配置文件的可读性太差，还是改回了数组。

配置项的说明，参见《配置文件说明.doc》。

## 系统服务（Service）

很多公用的类，比如Log、DBClient，整个应用程序只需用相同的参数实例化一个对象就可以供大家一起用了，但又不能设计成单实例模型，因为不能阻止用另外的参数实例化第二个对象的用法。我们把这些对象称为服务，用ServiceManager进行管理（参见ServiceManager类说明）。为方便，App包装了ServiceManager的getService方法，在代码的任何位置（因为App可以在任何位置通过App::Instance取得），可通过getService获得相应的服务，但该服务必须在全局的配置文件的服务列表中指明，并制定其初始化参数。

## 事件驱动

当A依赖B时，B发生某种事件时，A需要知道，但B不可能显示地调用A。当然，可以用监听者模式，A实现B的监听者接口，B通过接口调用A，这样也挺好。但是要实现监听接口和调用接口，系统中会堆积很多接口，也很麻烦。所以，框架实现了比较通用的事件驱动模式，用EventManager进行管理。同样，为了方便，App包装了EventManager的trigger方法。只要trigger方法就能把事件发给所有监听者了，而监听者只需要在配置文件中配置好就OK了。

## 模块

系统的组成单位是模块；

一个模块可以部署在任意位置，但要框架找到她，必须在配置文件里指定路径，并设置为可用。

要使模块名在系统中唯一，才能被别的模块正确调用，但已经开发好的模块很可能已经重名，修改起来很不方便。所以，在系统配置文件中，可以为模块指定别名（如果不指定，别名就是模块名）。模块的别名在系统中唯一。模块自身用模块名作为名字空间的开始，调用者用别名作为名字空间的开始，框架负责解析对应。

每个模块都可以被用户直接请求，通过index.php完成调用。虽然每个模块的index.php都很相似，但为了从URL上把每个模块区分开，使路由更加简单，还是保留了这种方式，没有采用整个应用程序用一个index.php的方式。

# 为什不

为什么不采用消息机制

为什么不采用事件机制