通用规范

基本约定

源码文件

1、所有源码文件均必须使用无BOM的UTF8编码格式。

2、纯PHP代码源文件，只使用`<?php`标签，省略标签`?>`。

缩进

使用Tab缩进，每个Tab长度设定为4个空格。

末尾空格

末尾空格，需删除掉。

关键字 和 true / false / null

PHP的所有关键字均使用小写，包括boolean值：true，false，null。

花括号

所有左花括号，都不换行，皆紧跟前一行代码，并且下一行，一定不能是空行，如下：

```
    public function getStatusName() {
        $status = self::getStatus();
        return $status[$this->status];
    }
```

例外情况：某些框架自动生成的代码，不必特意去修改。

数组

数组请使用新的写法：[]，不再使用array()。

命名规范

命名采用驼峰命名法。

1、 类名所有单词的首字母均大写。

2、 方法名和变量名，第一个单词首字母小写，其它单词首字母大写。

3、 类文件的文件名(不包含.php文件后缀)与类名一致。

4、 变量命名除非特定场景外，不使用下划线前缀。

5、 命名空间、文件夹尽量使用一个单词来命名。

例外情况：

1、由于数据库的命名规则有自己的传统习惯和要求，而一般ORM框架在实现ActiveRecord模式时，为了方便，ActiveRecord模型类的成员变量直接来自数据库表字段名。这种场景下，允许不遵循驼峰命名。

2、常量定义，所有字符均大写，单词之间使用下划线分割。

建议：

命名应该在保证能够准确表达所命名对象的含义的前提下，尽量简洁。特别是可以结合上下文环境，简化命名。比如在OrderModel类中使用create()而不使用createOrder()。

注意事项：

正式环境使用Linux系统，对文件名称大小写是敏感的，因此必须特别注意大小写一致。

代码注释

1、 注释必须遵守 phpDocument规范。

2、 类文件，必须注释，说明文件用途、作者。

3、 类的所有公有方法和变量必须进行注释，说明用途、参数信息、返回信息、异常。

4、 注释应该尽量简洁。

如下类似getXXX()方法，只有返回值，并且无需太多说明即可理解的情况下，则只需要一行注释即可：

```
1
2  /**
3   * @return string 返回状态名称
4   */
5  public function getStatusName() {
6  }
```

类似的，变量的注释，如果无需太多说明即可理解的情况下，则只需要一行注释即可：

```
1  /**
2   * @var integer 状态-冻结
3   */
4  const STATUS_1100 = 1100;
```

一个完整的注释的例子：

```php
/**
 * 描述类用途
 * @author 作者
 */

/**
 * 描述类用途
 * @author 作者
 */
class Example {
    /**
     * @var integer 一个常量
     */
    const A_CONSTANT = 1000;

    /**
     * @return string 返回名称
     */
    public function geName() {
    }

    /**
     * 方法描述
     * @param integer $param1 参数1描述
     * @param integer $param2 参数2描述
     * @return string 返回值描述
     */
    public function foo($param1, $param2) {
    }
}
```

代码风格

命名空间(Namespace) 和 导入(Use)声明

1、 命名空间(namespace)的声明后面必须有一行空行。

2、 所有的导入(use)声明必须放在命名空间(namespace)声明的下面。

3、 一句声明中，必须只有一个导入(use)关键字。

4、 在导入(use)声明代码块后面必须有一行空行。

5、 不要保留无用的use声明。

6、 use声明，尽量按照命名空间排序。

如下:

```php
namespace common\models\tenant\base\product;

use common\models\Error;
use common\models\tenant\LogicModel;

class ProductModel extends LogicModel {

}
```

类

1、 继承(extends)和实现(implement)必须和类名写在同一行。

```
class ProductModel extends LogicModel {

}
```

2、 成员变量，必须声明可见性，即public、private、protected，不能省略，也不能使用旧的var的写法。

3、 成员方法，必须声明可见性，即public、private、protected，不能省略。

如果有多个参数，逗号需紧跟参数名后，逗号后面加一个空格。

如果参数有默认值，“=”左右各有一个空格分开。

方法与方法之间必须留一个空行。

4、 abstract和final关键字，它们必须放在可见性声明(public、private、protected)的前面。而static关键字，则必须放在可见性声明的后面。

```
abstract class ClassName {
    abstract protected function zim();

    final public static function bar() {

    }

    private static $foo;
}
```

5、 私有、保护成员变量尽量放置在类代码的末尾。

关于私有、保护成员变量的放置位置，基本上有两种风格，一种放置在类的头部，一种放置在类的末尾。

我们更倾向于设计和编码应该面向用户(即使用者和维护人员)，即让用户更容易阅读、理解、使用和维护代码。而对于类来说，接口(这里指的是暴露给用户使用的公有成员变量或方法)是用户优先关注的，因此接口应该放置在更明显的位置，这有助于用户更快了解类的用途。私有和保护成员变量，属于类的内部实现细节，只有在需要对类的实现进行详细了解时，才需要关注，因此没有必要放置在头部显著位置。

6、 方法调用

方法和左括号(之间不应该有空格，对于参数，在分隔的逗号和下一个参数之间要有相同的空格分离，最后一个参数和右括号之间不能有空格。如下：

```
$result = foo($param1, $param2, $param3);
```

控制结构

1、 if、else、elseif

{}是必须的，即使分支下面只有一行代码。

if 、elseif与圆括号之间有一个空格，圆括号与花括号{之间有一个空格。

else、elseif紧跟if代码块的}后面，不另起一行。

else 左右各一个空格。

```php
<?php

if ($expr1) {

} elseif ($expr2) {

} else {

}
```

2、 switch、case

switch 与圆括号之间有一个空格，圆括号与花括号 { 之间有一个空格。

case相对switch有一个缩进，case下的代码块相对case有一个缩进。

```php
<?php

switch ($expr) {
    case 0:
        echo 'First case, with a break';
    break;
    case 1:
        echo 'Second case, which falls through';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;
}
```

3、 while、do while

while 与圆括号之间有一个空格，圆括号与 { 之间有一个空格。

do 与 { 之间有一个空格。

do while 结构中，while紧跟do代码块的}后面，不另起一行。

```php
<?php

while ($expr) {

}

do {

} while ($expr);
```

4、 for

for 与圆括号之间有一个空格，二元操作符 "="、"<" 左右各有一个空格，圆括号与 { 之间有一个空格。

```php
<?php

for ($i = 0; $i < 10; $i++) {

}
```

5、 foreach

foreach 与圆括号之间有一个空格，"=>" 左右各有一个空格，圆括号与 { 之间有一个空格。

```php
<?php

foreach ($iterable as $key => $value) {

}
```

6、 try catch

try 右边有一个空格。

catch紧跟try代码块的}后面，不另起一行。

catch与圆括号之间有一个空格，圆括号与 { 之间有一个空格。

```php
<?php

try {

} catch (ExceptionType $e) {

}
```

空行

1、 所有左花括号 { 都不换行，并且 { 紧挨着的下方，一定不是空行。

```php
<?php

if ($expr) {
    some code;
}
```

2、 同级代码（缩进相同）的注释（行注释/块注释）前面，必须有一个空行。

```php
<?php

if ($expr) {
    // 注释1
    $var1 = foo();

    // 注释2
    $var2 = foo();
}
```

3、 方法之间有一个空行。

4、 namespace语句、use语句、class语句之间有一个空行。

5、 return语句

如果 return 语句之前只有一行PHP代码，return 语句之前可以不需要空行。

如果 return 语句之前有至少二行PHP代码，return 语句之前加一个空行。

建议：逻辑紧密相关的代码，应适当增加空行与其它代码隔开，以便阅读。

行注释

// 后面需要加一个空格。

如果 // 前面有非空字符，则 // 前面需要加一个空格。

异常及错误处理

异常的使用规范

何时使用及使用什么异常

1、底层代码的参数或环境检查，其作用相当于C中的断言，只是用于确保方法被正确调用。也就是如果这些异常被抛出，就意味着方法被错误调用，属于代码错误或者环境未满足要求。这类异常直接使用yii\base\Exception。

2、正常情况下不可能发生的灾难性错误，比如数据库连接失败、操作失败，文件读写失败。这类错误一旦发生，则意味着当前执行流程无法继续，通常也难以自动采取有效措施修复问题。这类异常直接使用yii\base\Exception。

3、Http异常，包括常见的404等错误。这类异常使用yii\web\HttpException。目前系统中仅允许使用404、400、403错误。

404表示页面不存在，一般情况下，由框架抛出。此外对直接体现在URL地址中的GET参数，验证失败时可以抛出404错误。

除上述情形之外的GET参数或POST参数验证失败，请使用400错误。

但需要牢记异常的使用场景，仅当GET参数或POST参数是由系统自行生成的情况，验证失败才使用异常，这种情况下通常意味着客户自行修改了参数信息或编码错误，属于意外情况。如果参数由客户输入，则应该使用正常的错误提示，而不应该抛出异常。

4、业务逻辑错误产生的异常。一般情况，业务逻辑错误，不应该抛出异常，而是返回错误信息。

异常的捕获

如果没有特殊情况，不应该主动捕获异常，异常信息应由框架统一捕获并处理。

异常信息的显示

系统对于上述异常的显示，采用如下的方式：

1、对于Exception及PHP Exception异常，同PHP错误一样，统一报Http 500错误。该页面，只提示系统发生错误，但不显示任何具体的错误信息，以避免暴露系统的内部实现细节。

2、对于HttpException异常，根据code的不同，使用不同的错误页面。

3、对于业务逻辑错误产生的异常，均使用相同的错误页面。该页面会显示异常的message信息。

错误信息设置规范

代码返回的错误信息，应当尽量对用户友好，即容易阅读和理解。

当需要对来自不同逻辑层次的错误进行拼接的时候，请统一使用小括号，如下：

```
1 return $this->setError('库中记录删除失败('. $model->getError().')');
```

安全规范

对客户数据进行验证

来自客户端提交的数据，包括POST参数、GET参数、cookie等，都认为是不安全的，必须进行严格的验证。

防止JavaScript 注入攻击

对于来自用户的输入信息，在输出时，应该尽量使用文本框、下拉框等表单控件来展示，如果直接输出，则需要对可能存在的HTML标签或JS代码进行转义后输出。

如果需要原样输出带有HTML标签的文本，则需要在输入的时候做严格的过滤。

防止SQL注入攻击

数据库操作必须使用预编译方式，避免SQL注入。

YII项目规范

基本约定

命名

1、 AR模型：与表名一致，如user表，对应的AR模型类为User。

2、 表单模型：使用Form后缀，如User表单模型类为UserForm。

3、 model模型：使用Model后缀，如UserModel。

4、 service模型：使用Service后缀，如user管理模块对应的Service模型类为UserService。

5、 过滤器类：使用Filter后缀，如权限验证过滤器类为：PrivilegeFilter。

6、 AR模型中的成员变量，因为实现的原因，必须保证跟数据库字段命名一致，允许使用非驼峰命名。除此之外，除非特别说明的例外情况，否则均必须使用驼峰命名。

代码注释

Controller类的所有action方法，必须注释方法用途。

Service类允许不做注释。

Form表单类允许不做注释。

其它未特别说明的场景，均需要按规范进行注释。

代码风格

安全规范

对客户数据进行验证

来自客户端提交的数据，包括POST参数、GET参数、cookie等，都认为是不安全的，必须进行严格的验证。

1、输入型的数据，如通过表单中的文本框输入，通过GET参数提交或cookie提交的数据。这些数据需要通过表单模型验证其格式是否合法。

2、对于富文本编辑框，必须使用HtmlPurifierFilter对内容进行过滤，剔除js脚本，并确保html标签正确闭合。

3、对于没有任何约束，允许输入任意字符的字段，也应该在表单模型中明确声明。

4、选择型的数据，如通过表单中的下拉框、单选、复选框选择的数据。这些数据通常来自其它数据表或有固定的取值，表单模型通常难以进行验证。因此通常会被忽略掉。这部分数据，同样应该严格验证和过滤。如果是固定取值，则应该通过表单模型的验证规则，验证取值是否在设定的范围内。如果来自其它数据表，则应该验证对应的记录是否存在并且有效。

防止JavaScript 注入攻击

对于来自用户的输入信息，在输出时，应该尽量使用文本框、下拉框等表单控件来展示，如果直接输出，则必须使用Html::encode()编码后输出。

对于必须原样输出的信息，比如富文本信息，应该在输入保存的时候，使用HtmlPurifierFilter进行过滤。

防止SQL注入攻击

在AR中禁止使用类似下面直接拼接SQL的代码：

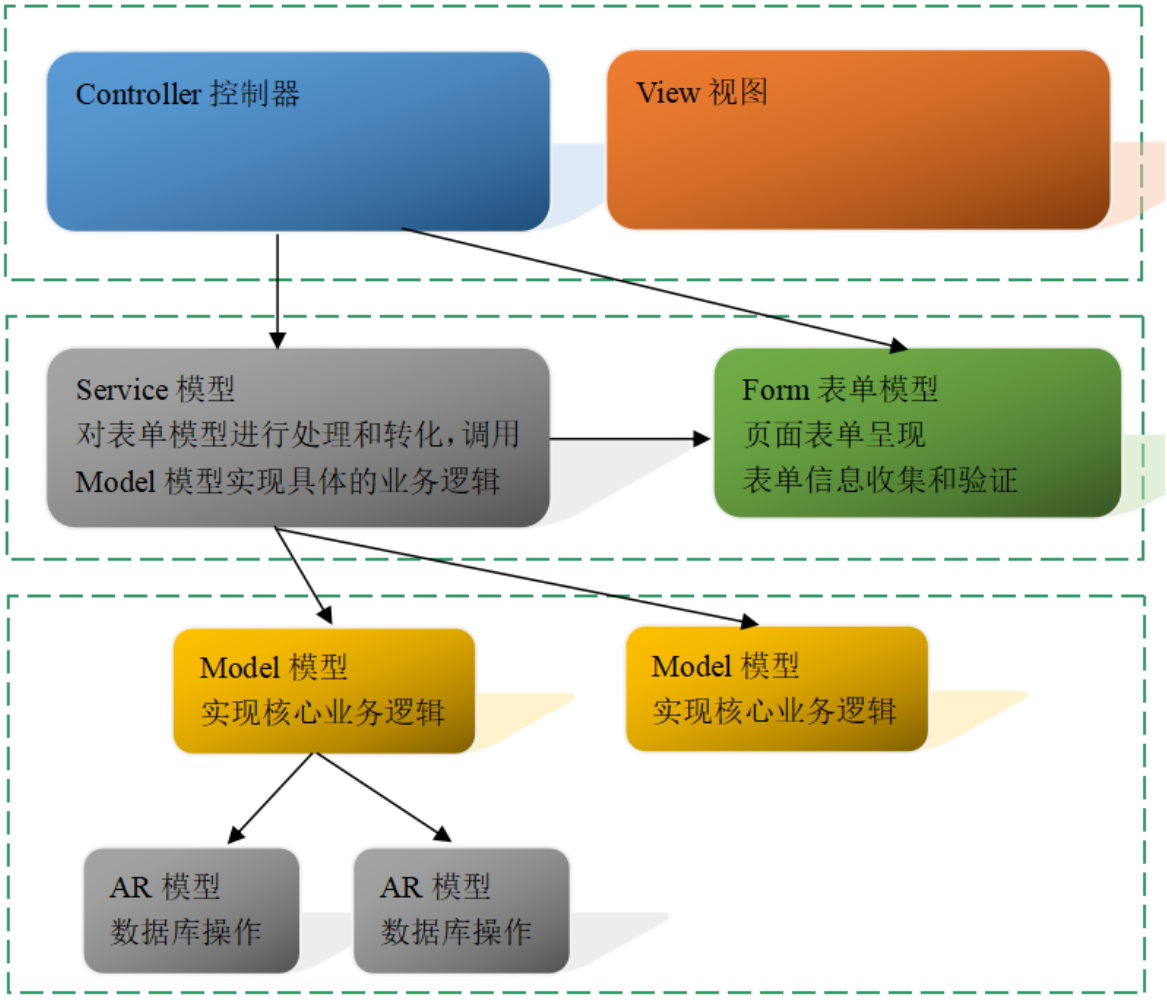
```
1 $sql = "SELECT * FROM table WHERE id=$id";
```

这种方式，如果拼接的参数来自表单，并且未被严格过滤，则容易被注入，存在安全隐患。

应该尽量使用使用绑定方式，如下：

```
1 $sql = "SELECT * FROM table WHERE id = :id";
2 $params = [':id' => $id];
3 $command = $db->createCommand($sql, $params);
```

分层及各层职责



控制器

控制器只负责连接视图和模型，不实现业务逻辑，只能通过调用service模型提供的接口完成业务逻辑操作。

视图

视图只负责做渲染，同样不实现业务逻辑。视图渲染过程中，所需要的所有数据，均应该由控制器传入。

表单模型

表单模型：用于构造表单、收集表单信息，并对表单信息进行验证。验证规则统一在表单模型中设置。

表单模型类只与控制器及Service类有关，它们均与具体的功能点及界面相关。因此不具有太大的通用性，通常放置在各自应用的models目录下。

Service模型类

Service模型类，用于提供完成具体业务功能的接口，这些接口由controller类的action方法调用，以完成具体的功能操作。或者也可以提供给SOAP或API接口使用。

Service模型类与功能相关，因此不同应用拥有各自的Service模型类。Service模型类文件，位于各自应用的models目录下面。

1、 Service模型类必须从Service类继承，其命名必须为XXXService。

2、 Service模型，直接为控制器服务，对表单模型数据进行加工，并调用底层的Model模型完成实际的业务逻辑操作。应该尽量避免在Service类中实现过于复杂的逻辑。

3、 Service类更偏向于过程，其大部分接口应该直接与控制器的action对应。除此之外只能包含少量get()等方法，或者私有方法(Service模型类不需要被继承)。

4、 Service模型类之间应该是相对独立的，尽可能避免调用其它的Service模型类。如果业务逻辑具有较强的通用性，可以被不同应用之间共享，则应该抽到底层的Model模型类中。

AR模型

AR模型，主要负责数据库相关操作，及与状态相关字段状态常量定义和状态名称接口。AR模型的设计和编码，必须遵循下面的规范：

1、AR模型，必须先创建相应的数据表，然后使用gii工具生成对应的模型类。gii会根据表结构，自动生成相应的验证规则、字段说明等代码，这可以减少工作量。

2、AR模型只负责数据库相关操作，不涉及任何业务规则。也就是AR可以提供类似下面的接口：

```
getXXX()
updateXXX()
deleteXXX()
saveXXX()
insertXXX()
isExistsXXX()
```

而不能提供类似下面的接口：

```
renew()
open()
close()
```

3、应该尽量避免在AR模型之外编写SQL语句。

4、AR模型类中针对所有表记录的操作，应该使用静态方法，针对当前记录的操作则使用非静态方法。

```
public static function getById($id)
public function getStatusName()
```

5、AR模型的验证规则，仅使用默认生成的规则，用于确保数据存储时，字段正确。除验证字段唯一外，不添加其它的验证规则。所有验证均由表单模型完成。

Model模型类

Model模型类，核心的业务逻辑，通过调用AR模型来完成数据库操作。

1、 Model模型类必须从LogicModel继承，其命名必须为XXXModel。

2、 Model模型类不涉及操作日志，操作日志必须在Service模型类中完成。

3、 Model模型类不依赖于任何的表单模型，其接口和实现中，均不得使用表单模型。

目录结构

案例

以下是服务商收货地址，相关的类文件组织结构：

```php
<?php
namespace common\models\distribution\system;

class DistributionReceiveAddress extends \common\models\ActiveRecord
{
}
```

```php
<?php
namespace app\models\system;

class AddressService extends Service
{
}
```

```php
<?php
namespace app\controllers\system;

class AddressController extends Controller
{
}
```

不良编码案例

接口设计

```php
<?php
namespace app\controllers\cart;

class ShoppingCartController extends Controller
{
    ...
    public function actionSettle() {
        ...
        $addressService = new AddressService();
```

```
        $addressProvider = $addressService->search();
        $address = $addressService->dealActiveData($addressProvider);
        ...
    }
    ...
}
```

```php
<?php
namespace app\models\system;

...

class AddressService extends Service
{
    ...
    /**
     * return Array()
     */
    public function dealActiveData($dataProvider) {
        $models = $dataProvider->getModels();
        $items = [ ];
        foreach($models as $address) {
            $region = Region::getById($address->district_id);

            $items['all'][$address->id] = $address->contact_name."
".$address->contact_tel." ".((isset($region['full_name'])) ?
str_replace("/","",$region['full_name']) : '')." ".$address->address;
            if($address->is_default
==DistributionReceiveAddress::DEFAULT_TRUE){
                $items['is_default'] = $address->id;
            }
        }
        return $items;
    }
    ...
}
```