

——高性能PHP应用开发之

## 第2讲 反射机制和设计模式

# 目录

## CONTENTS

1 / 反射API

---

2 / 设计模式

---

# 目录

## CONTENTS

### 1 / 反射API

---













### 2 / 设计模式

---

# 反射API简介

反射是指在PHP运行状态中，扩展分析PHP程序，导出或提取出关于类、方法、属性、参数、注释等的详细信息。这种**动态获取的信息以及动态调用对象的方法**的功能称为反射API。反射API主要使用场景：自动加载插件，自动生成文档，甚至可用来扩充PHP语言。

反射API由若干类组成，可帮助我们用来访问程序的元数据或者同相关的注释交互。借助反射API我们可以获取诸如类实现了那些方法，创建一个类的实例（**不同于用new创建**），调用一个方法（不同于常规调用），传递参数，动态调用类的静态方法等等功能。

- ⊕  Reflection
- ⊕  ReflectionClass
- ⊕  ReflectionZendExtension
- ⊕  ReflectionExtension
- ⊕  ReflectionFunction
- ⊕  ReflectionFunctionAbstract
- ⊕  ReflectionMethod
- ⊕  ReflectionObject
- ⊕  ReflectionParameter
- ⊕  ReflectionProperty
- ⊕  Reflector
- ⊕  ReflectionException

# Reflection类

Reflection类只提供了两个静态方法，以查看当前PHP中可使用反射API方法。

- ✓ export(): 导出一个类或方法的详细信息。
- ✓ getModifierNames(): 取得修饰符的名字。

可以使用 **Reflection::export( new ReflectionExtension( 'reflection' ) );** 列出当前可用的完整反射API类声明。

```
Reflection {  
  
    /* 方法 */  
    public static string export ( Reflector $reflector [, bool $return = false ] )  
    public static array getModifierNames ( int $modifiers )  
}
```

# ReflectionFunction类

ReflectionFunction返回一个函数的相关信息。

```
// 构造方法
public object __construct(string name)
// 导出该函数的详细信息
public static string export()
// 取得函数名
public string getName()
// 测试是否为系统内部函数
public bool isInternal()
//测试是否为用户自定义函数
public bool isUserDefined()
//取得文件名，包括路径名
public string getFileName()
//取得定义函数的起始行
public int getStartLine()
//取得定义函数的结束行
public int getEndLine()
//取得函数的注释
public string getDocComment()
```

```
//取得静态变量
public array getStaticVariables()
//调用该函数，通过参数列表传参数
public mixed invoke(mixed* args)
//调用该函数，通过数组传参数
public mixed invokeArgs(array args)
//测试该函数是否返回引用
public bool returnsReference()
//取得该方法所需的参数，返回值为对象数组
public ReflectionParameter[] getParameters()
//取得该方法所需的参数个数
public int getNumberOfParameters()
//取得该方法所需的参数个数
public int getNumberOfRequiredParameters()
```

# ReflectionClass类

ReflectionClass返回一个类的相关信息。

```
// 构造方法：类名
public object __construct(string name)
//导出该类的详细信息
public static string export()
//取得类名或接口名
public string getName()
//测试该类是否为系统内部类
public bool isInternal()
//测试该类是否为用户自定义类
public bool isUserDefined()
//测试该类是否被实例化过
public bool isInstantiable()
//测试该类是否有特定的常量
public bool hasConstant(string name)
//测试该类是否有特定的方法
public bool hasMethod(string name)
//测试该类是否有特定的属性
public bool hasProperty(string name)
```

```
//取得定义该类的文件名，包括路径名
public string getFileName()
//取得定义该类的开始行
public int getStartLine()
//取得定义该类的结束行
public int getEndLine()
//取得该类的注释
public string getDocComment()
//取得该类的构造函数信息
public ReflectionMethod getConstructor()
//取得该类的某个特定的方法信息
public ReflectionMethod getMethod(string name)
//取得该类的所有的方法信息
public ReflectionMethod[] getMethods()
//取得某个特定的属性信息
public ReflectionProperty getProperty(string name)
//取得该类的所有属性信息
public ReflectionProperty[] getProperties()
```



# ReflectionClass类

ReflectionClass返回一个类的相关信息。

```
//取得该类所有常量信息
public array getConstants()
//取得该类特定常量信息
public mixed getConstant(string name)
//取得接口类信息
public ReflectionClass[] getInterfaces()
//测试该类是否为接口
public bool isInterface()
//测试该类是否为抽象类
public bool isAbstract()
//测试该类是否声明为final
public bool isFinal()
//取得该类的修饰符，返回值类型可能是个资源类型
public int getModifiers()
//通过Reflection::getModifierNames(
// $class->getModifiers())进一步读取
//测试传入的对象是否为该类的一个实例
public bool isInstance(stdclass object)

//创建该类实例
public stdclass newInstance(mixed* args)
//取得父类
public ReflectionClass getParentClass()
//测试传入的类是否为该类的父类
public bool isSubclassOf(ReflectionClass class)
//取得该类的所有静态属性
public array getStaticProperties()
//取得该类的静态属性值，若private，则不可访问
public mixed getStaticPropertyValue(string name [, mixed default])
//设置该类的静态属性值，若private，则不可访问，有悖封装原则
public void setStaticPropertyValue(string name, mixed value)
//取得该类的属性信息，不含静态属性
public array getDefaultProperties()
//测试是否实现了某个特定接口
public bool implementsInterface(string name)
```



# 其它反射类

类	描 述
Reflection	为类的摘要信息提供静态函数export()
ReflectionClass	类信息和工具
ReflectionMethod	类方法信息和工具
ReflectionParameter	方法参数信息
ReflectionProperty	类属性信息
ReflectionFunction	函数信息和工具
ReflectionExtension	PHP扩展信息
ReflectionException	错误类

- 实战应用：使用反射API查询Redis类的方法和属性。

# 目录

## CONTENTS

1 / 反射API

---

2 / 设计模式

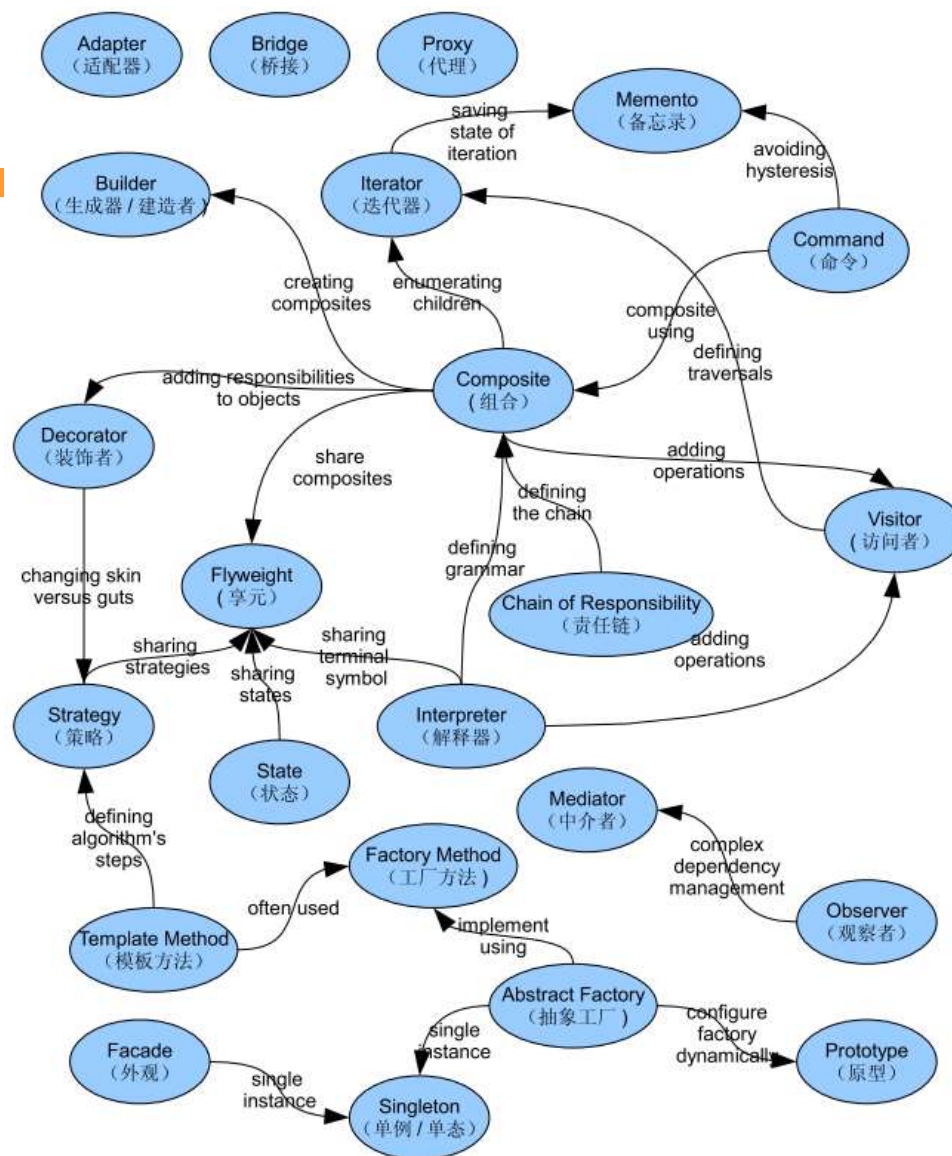
---

# 设计模式

设计模式 (Design pattern) 是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结；使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。总体来说设计模式分为三大类：

- ✓ 创建型模式，共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
- ✓ 结构型模式，共7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- ✓ 行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

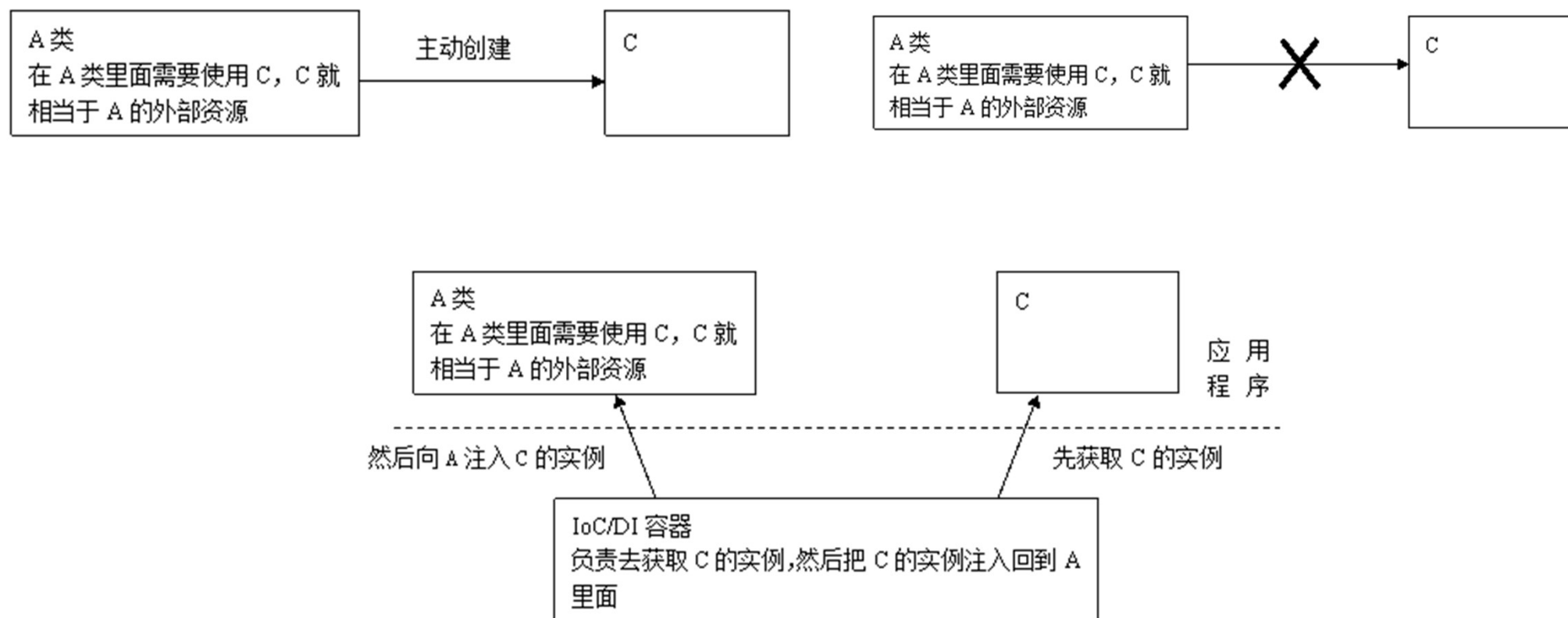
# 设计模式



# 依赖注入 (Dependency Inject)

- 简介：当一个类的实例需要另一个类的实例协助时，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。而采用依赖注入的方式，创建被调用者的工作不再由调用者来完成，因此叫**控制反转 (Inverse Of Control)**，**创建被调用者的实例的工作由IOC容器来完成**，然后注入调用者，因此也称为依赖注入。
- 使用场景：Laravel框架大量使用了依赖注入机制。
- 实现原理：创建**IOC容器**管理注入类，**闭包函数**。
- 参考代码：

# 依赖注入 (Dependency Inject)



# 依赖注入 (Dependency Inject)

```
class Di {  
    protected $_service = [];  
    public function set($name, $definition) {  
        $this->_service[$name] = $definition;  
    }  
    public function get($name) {  
        return call_user_func($this->_service[$name]);  
    }  
}
```

```
class Person {  
    protected $_name;  
    public function __construct($name) {  
        $this->_name = $name;  
    }  
    public function getName() {  
        return $this->_name;  
    }  
}
```

```
$di = new Di();  
$di->set('Person', function() {  
    return new Person('test');  
});  
$person = $di->get('Person');  
echo $person->getName();
```



# 依赖注入 (Dependency Inject)

```
class Di {  
    protected static $_service = [];  
    public static function bind($name, Callable $resolver) {  
        static::$_service[$name] = $resolver;  
    }  
    public static function make($name) {  
        if (isset(static::$_service[$name])) {  
            $resolver = static::$_service[$name];  
            return call_user_func($resolver);  
        }  
    }  
}  
  
Di::bind('person', function() {  
    return new Person('test');  
});  
echo Di::make('person')->getName();
```

# 简单工厂模式

- 基本思想：提供获取某个对象实例的一个接口，同时使调用代码避免确定实例化基类的步骤。工厂模式实际上就是建立一个统一的类实例化的函数接口，统一调用，统一控制。
- 应用场景：针对不同缓存类型，建立统一的构建工厂。
- 实现原理：分别创建各个具体类，创建工厂类提供静态方法实现各具体类的实例化。
- 参考代码：

```
Interface TransformType {  
    abstract public function write();  
}  
class Json implements TransformType { }  
class Xml implements TransformType { }
```

```
class Factory {  
    static public function create($className) {  
        $className = ucfirst(strtolower($className));  
        if (class_exists($className)) {  
            return new $className;  
        }  
        return null;  
    }  
}
```

# 单例模式

- 基本思想：通过提供自身共享实例的访问，单例设计模式用于限制特定对象只能被创建一次。
- 使用场景：数据库连接池的实例化。
- 实现原理：定义构造方法为private类型，定义静态属性保存已经实例化的对象。
- 参考代码：

```
class Singleton {  
    static private $_instance = null;  
    private function __construct(属性参数) {  
        // 初始化动作  
    }  
    static public function getInstance(属性参数) {  
        if (null == self::$_instance) {  
            self::$_instance = new self(属性参数);  
        }  
        return self::$_instance;  
    }  
}
```

# 观察者模式

- 基本思想：能够更便利地创建和查看目标对象状态的对象，并且提供和核心对象非耦合的置顶功能性。
- 使用场景：订单完成后，执行后续多种操作。
- 实现原理：每一种后续操作抽象成类中的方法，为目标类添加观察者，循环遍历观察者，执行观察者的方法。
- 参考代码：

# 观察者模式

```
class order {
    protected $observers = array(); // 存放观察容器
    //观察者新增
    public function addObserver($type, $observer) {
        $this->observers[$type][] = $observer;
    }
    //运行观察者
    public function obServer($type) {
        if (isset($this->observers[$type])) {
            foreach ($this->observers[$type] as $obser) {
                $a = new $obser;
                $a->update($this);
            }
        }
    }
    //下单购买流程
    public function create() {
        echo '购买成功';
        $this->obServer('buy'); // buy动作
    }
}
```

```
interface Afterorder {
    abstract public function update(Order $order);
}
class orderEmail implements Afterorder {
    public static function update($order) {
        echo '发送购买成功的邮件';
    }
}
class orderStatus implements Afterorder {
    public static function update($order) {
        echo '改变订单状态';
    }
}

$obj = new order;
$obj->addObserver('buy', 'orderEmail');
$obj->addObserver('buy', 'orderStatus');
$obj->create();
```

# SPL设计模式：观察者模式

观察者模式定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。一般来说，观察者模式包括如下四个角色：

- ✓ 抽象主题（Subject）角色：主题角色将所有对观察者对象的引用保存在一个集合中，每个主题可以有任意多个观察者。抽象主题提供了增加和删除观察者对象的接口。
- ✓ 抽象观察者（Observer）角色：为所有的具体观察者定义一个接口，在观察的主题发生改变时更新自己。
- ✓ 具体主题（ConcreteSubject）角色：存储相关状态到具体观察者对象，当具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个具体子类实现。
- ✓ 具体观察者（ConcreteObserver）角色：存储一个具体主题对象，存储相关状态，实现抽象观察者角色所要求的更新接口，以使得其自身状态和主题的状态保持一致。



# SPL设计模式：观察者模式

SPL实现了其中两个抽象角色：**SplObserver接口**和**SplSubject接口**。如果我们需要实现观察者模式，仅需要实现这两个接口即可。并且这两个接口定义在模块初始化的方法中

```
class Newspaper implements \SplSubject{
    private $name;
    private $observers = array();
    private $content;
    public function __construct($name) {
        $this->name = $name;
    }
    //add observer
    public function attach(\SplObserver $observer) {
        $this->observers[] = $observer;
    }
    //remove observer
    public function detach(\SplObserver $observer) {
        $key = array_search($observer,$this->observers, true);
        if($key){
            unset($this->observers[$key]);
        }
    }
}
```

```
//set breakouts news
public function breakOutNews($content) {
    $this->content = $content;
    $this->notify();
}
public function getContent() {
    return $this->content." ({$this->name})";
}
//notify observers(or some of them)
public function notify() {
    foreach ($this->observers as $value) {
        $value->update($this);
    }
}
```



# SPL设计模式：观察者模式

SPL实现了其中两个抽象角色：**SplObserver接口**和**SplSubject接口**。如果我们需要实现观察者模式，仅需要实现这两个接口即可。并且这两个接口定义在模块初始化的方法中

```
class Reader implements SplObserver{
    private $name;

    public function __construct($name) {
        $this->name = $name;
    }

    public function update(\SplSubject $subject) {
        echo $this->name.' is reading breakout news <b>'
            . $subject->getContent(). '</b><br>';
    }
}
```

```
$newspaper = new Newspaper('Newyork Times');
$allen = new Reader('Allen');
$jim = new Reader('Jim');
$linda = new Reader('Linda');
//add reader
$newspaper->attach($allen);
$newspaper->attach($jim);
$newspaper->attach($linda);
//remove reader
$newspaper->detach($linda);
//set break outs
$newspaper->breakOutNews('USA break down!');
```

# 外观（门面）模式

- 基本思想：用于为子系统的一组接口提供一个一致的界面。门面模式定义了一个高层接口，这个接口使得子系统更加容易使用：引入门面角色之后，用户只需要直接与门面角色交互，用户与子系统之间的复杂关系由门面角色来实现，从而降低了系统的耦合度。  
。
- 使用场景：不同缓存系统，提供统一的操作接口。
- 实现原理：门面类封装具体类的实际操作。
- 参考代码：

# 外观（门面）模式

```
class Facade {  
    protected $_driver = null;  
    public function __construct(CacheDriver $driver) {  
        $this->_driver = $driver;  
    }  
    public function get($name) {  
        return $this->_driver->get($name);  
    }  
}  
  
interface CacheDriver {  
    abstract public function get($name);  
}  
  
class Memcached implements CacheDriver { }
```

```
use \Facade as Cache;  
$cache = new Cache(new Memcached());  
echo $cache->get('html');
```

# 感谢聆听！

---

THANK YOU FOR YOUR ATTENTION