PHP 开发手册

前 言

本书《php 开发手册》是根据《Java 开发手册》和网络搜索汇编而成。整理这个是因为看到阿里巴巴提供的开发手册觉得 php 缺少一个类似的手册。以此整理出来手册供 phper 查阅。

本手册以 php 开发者为中心视角，划分为**编程规约、异常日志、单元测试、安全规约、MySQL 数据库、工程结构、设计规约**七个维度，再根据内容特征，细分成若干二级子目录。

另外，依据约束力强弱及故障敏感性，规约依次分为强制、推荐、参考三大类。在延伸信息中，

“说明”对规约做了适当扩展和解释；

“正例”提倡什么样的编码和实现方式；

“反例”说明需要提防的雷区，以及真实的错误案例。

github 地址： https://github.com/songangweb/php_development_document
以便获取最新的文档

另：欢迎留言,加 qq 群交流,共同完善此手册。
qq 群号:691025612

PHP 开发手册

目录

一、编程规约

（一）命名风格

1. 【强制】代码中的命名均不能以符号与数字开始，也不能以符号与数字结束。
   反例：_name / __name / name_ / name$ / name__2

2. 【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。
   说明：正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，纯拼音命名方式更要避免采用。
   正例：renminbi / alibaba / taobao / youku / hangzhou 等国际通用的名称，可视同英文。
   反例：DaZhePromotion() [打折] / getPingfenByName() [评分] / $某变量 = 3

3. 【强制】类名使用 UpperCamelCase 风格。
   正例：PhpServerlessPlatform / TaskReply
   反例：phpserverlessplatform / taskREPLY

4. 【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase 风格，必须遵从驼峰形式。
   正例：getHttpMessage() / $inputUserId

5. 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。
   正例：MAX_STOCK_COUNT / CACHE_EXPIRED_TIME
   反例：MAX_COUNT / EXPIRED_TIME

6. 【强制】抽象类命名使用 Abstract 或 Base 开头；异常类命名使用 Exception 结尾；测试类命名以它要测试的类的名称开始，以 Test 结尾。

7. 【强制】避免在子父类的成员变量之间、或者不同代码块的局部变量之间采用完全相同的命名，使可读性降低。

   说明：子类、父类成员变量名相同，即使是 public 类型的变量也是能够通过编译，而局部变量在同一方法内的不同代码块中同名也是合法的，但是要避免使用。对于非 setter/getter 的参数名称也要避免与成员变量名称相同。
   反例：
   class ConfusingName {

```
        public $age;
        // 非 setter/getter 的参数名称，不允许与本类成员变量同名
        public getData() {
            if(condition) {
                $money = 531;

                // ...
            }
                for ($i = 0; $i < 10; $i++) {
                    // 在同一方法体中，不允许与其它代码块中的 money 命名相同
                    $money = 615;
                    // ...
                }
            }
        }
    class Son extends ConfusingName {
        // 不允许与父类的成员变量名称相同
        public $age;
    }
```

8. **【强制】杜绝完全不规范的缩写，避免望文不知义。**

反例：AbstractClass“缩写”命名成 AbsClass；condition“缩写”命名成 condi，此类随意缩写严重降低了代码的可阅读性。

9. **【推荐】如果类的对象名已经告诉你了后,就不需要在类中的方法名中,重复定义**

反例：

```
class Car {
    public $carMake;
    public $carModel;
    public $carColor;
    //...
}
```

正例：

```
class Car {
    public $Make;
    public $Model;
    public $Color;
    //...
}
```

10. 【推荐】为了达到代码自解释的目标，任何自定义编程元素在命名时，使用尽量完整的单词组合来表达其意。

正例：表达原子更新的类名为：AtomicReferenceFieldUpdaterController。

反例：$a 的随意命名方式。

11. 【推荐】在常量与变量的命名时，表示类型的名词放在词尾，以提升辨识度。

正例：$startTime / $workQueue / $nameList / TERMINATED_THREAD_COUNT

反例：$startedAt / $QueueOfWork / $listName / COUNT_TERMINATED_THREAD

12. 【推荐】如果模块、接口、类、方法使用了设计模式，在命名时需体现出具体模式。

说明：将设计模式体现在名字中，有利于阅读者快速理解架构设计理念。

正例： public class OrderFactory;

public class LoginProxy;

public class ResourceObserver;

13. 【推荐】接口和实现类的命名有两套规则：

1)【强制】对于 Service 和 DAO 类，基于 SOA 的理念，暴露出来的服务一定是接口，内部的实现类用 Impl 的后缀与接口区别。

正例：CacheServiceImpl 实现 CacheService 接口。

2)【推荐】如果是形容能力的接口名称，取对应的形容词为接口名（通常是－able 的形容词）。

正例：AbstractTranslator 实现 Translatable 接口。

14. 【参考】枚举类名带上 Enum 前缀或后缀，枚举成员名称需要全大写，单词间用下划线隔开。

说明：枚举其实就是特殊的类，域成员均为常量，且构造方法被默认强制是私有。

正例：枚举名字为 ProcessStatusEnum / EnumProcessStatus

成员名称：SUCCESS / UNKNOWN_REASON。

15. 【参考】Service/DAO 层方法命名规约规约：

1) 获取单个对象的方法用 get 做前缀。

2）获取多个对象的方法用 list 做前缀，复数形式结尾如：listObjects。

3）获取统计值的方法用 count 做前缀。

4）插入的方法用 save/insert 做前缀。

5）删除的方法用 remove/delete 做前缀。

6）修改的方法用 update 做前缀。

（二） 常量定义

1. 【强制】不允许任何魔法值（即未经预先定义的常量）直接出现在代码中。

反例：$key = "Id#songang_" + tradeId;

```
cache.put($key, value);
// 缓存 get 时，由于在代码复制时，漏掉下划线，导致缓存击穿而出现问题
```

2. 【推荐】不要使用一个常量类维护所有常量，要按常量功能进行归类，分开维护。

说明：大而全的常量类，杂乱无章，使用查找功能才能定位到修改的常量，不利于理解和维护。

正例：缓存相关常量放在类 CacheConsts 下；系统配置相关常量放在类 ConfigConsts 下。

（三） 代码格式

1. 【强制】IDE 的 text file encoding 设置为 UTF-8; IDE 中文件的换行符使用 Unix 格式，不要使用 Windows 格式。

2. 【强制】如果是大括号内为空，则简洁地写成{}即可，大括号中间无需换行和空格；如果是非空代码块则：

1） 左大括号前不换行。
2） 左大括号后换行。
3） 右大括号前换行。
4） 右大括号后还有 else 等代码则不换行；表示终止的右大括号后必须换行。

3. 【强制】左小括号和字符之间不出现空格；同样，右小括号和字符之间也不出现空格；而左大括号前需要空格。详见第 5 条下方正例提示。

反例：if (空格 a == b 空格)

4. 【强制】if/for/while/switch/do 等保留字与括号之间都必须加空格。

5. 【强制】任何二目、三目运算符的左右两边都需要加一个空格。

说明：运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号等。

6. 【强制】采用 4 个空格缩进，禁止使用 tab 字符。

说明：如果使用 tab 缩进，必须设置 1 个 tab 为 4 个空格。IDEA 设置 tab 为 4 个

空格时，请勿勾选 Use tab character；而在 eclipse 中，必须勾选 insert spaces for tabs。

正例： （涉及 1-5 点）

```
public main() {
    // 缩进 4 个空格
    $say = "hello";
    // 运算符的左右必须有一个空格
    $flag = 0;
    // 关键词 if 与括号之间必须有一个空格，括号内的 f 与左括号，0 与右括号不需要空格
    if ($flag == 0) {
        echo $say;
    }
    // 左大括号前加空格且不换行；左大括号后换行
    if ($flag == 1) {
        echo "world";
        // 右大括号前换行，右大括号后有 else，不用换行
    } else {
        echo "ok";
        // 在右大括号后直接结束，则必须换行
    }
}
```

7. **【强制】注释的双斜线与注释内容之间有且仅有一个空格。**

正例：

```
// 这是示例注释，请注意在双斜线之后有一个空格
$db = new DbOrm();
```

8. **【强制】在进行类型强制转换时，右括号与强制转换值之间不需要任何空格隔开。**

正例：

```
$foo = '1';
$bar = (int)$foo + 1;
```

9. **【强制】单行字符数限制不超过 120 个，超出需要换行，换行时遵循如下原则：**

1）第二行相对第一行缩进 4 个空格，从第三行开始，不再继续缩进，参考示例。

2）运算符与下文一起换行。

3）方法调用的点符号与下文一起换行。

4）方法调用中的多个参数需要换行时，在逗号后进行。

5）在括号前不要换行，见反例。

正例：

```
$db = new DbOrm();
// 超过 120 个字符的情况下，换行缩进 4 个空格，点号和方法名称一起换行
```

```
$db->table('user')
->where('age',20)
->where('age',21)
->where('age',22)
->where('age',23)
->select();
```

反例：

```
$db = new DbOrm();
// 超过 120 个字符的情况下，不要在括号前换行
$db->table('user')->where('age',20)->where('age',21)->where
('age',22)->where('age',23)->select();
// 参数很多的方法调用可能超过 120 个字符，不要在逗号前换 1 行
method($args1, $args2, $args3, ...
, $argsX);
```

10. 【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格。

正例：下例中实参的 $args1，后边必须要有一个空格。

```
method($args1, $args2, $args3);
```

11. 【推荐】 函数参数(限制为 2 个或更少)

说明：限制函数参数个数极其重要,因为参数越多,可扩展性越男,如果必须的话,建议封装为对象,或传递数组

反例：

```
function createMenu(string $title, string $body, string $buttonText, bool $cancellable):
void {
        // ...
 }
```

12. 【推荐】 不要用标示作为函数的参数

说明： 一个函数应该只做一件事,把不同标示的代码拆分到多个函数里

正例：

```
function createFile(string $name): void
{
    touch($name);
}

function createTempFile(string $name): void
{
    touch('./temp/'.$name);
}
```

反例：

```
function createFile(string $name, bool $temp = false): void
```

```
{
    if ($temp) {
        touch('./temp/'.$name);
    } else {
        touch($name);
    }
}
```

13. 【推荐】单个方法的总行数不超过 80 行。

说明：除注释之外的方法签名、左右大括号、方法内代码、空行、回车及任何不可见字符的总行数不超过
80 行。

正例：代码逻辑分清红花和绿叶，个性和共性，绿叶逻辑单独出来成为额外方法，使主干代码更加清；共性逻辑抽取成为共性方法，便于复用和维护。

14. 【推荐】没有必要增加若干空格来使变量的赋值等号与上一行对应位置的等号对齐。

正例：

```
$one = 1;
$two = '2L';
$three = '3F';
$sb = 'sb';
```

反例：

```
$one     = 1;
$two     = '2L';
$three   = '3F';
$sb      = 'sb';
```

说明：增加 $three 这个变量，如果需要对齐，则给 $one、$two、$sb 都要增加几个空格，在变量比较多的情况下，是非常累赘的事情。

15. 【推荐】不同逻辑、不同语义、不同业务的代码之间插入一个空行分隔开来以提升可读性。

说明：任何情形，没有必要插入多个空行进行隔开。

（四）OOP 规约

1. 【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

2. 【强制】相同参数类型，相同业务含义，才可以使用可变参数，避免使用Object。

   说明：可变参数必须放置在参数列表的最后。（提倡同学们尽量不用可变参数编程）
   正例：function display($name, $tag, ...$args){...}

3. 【强制】不能使用过时的类或方法。
   说明：如果接口提供方既然明确是过时接口，那么有义务同时提供新的接口；作为调用方来说，有义务去考证过时方法的新实现是什么。

4. 【强制】浮点数之间的等值判断，可以把浮点数转化为字符串进行比较。
   说明：

```
$a = 0.58 * 100;
$b = 58;
var_dump($a);// 输出 float 58
var_dump($b);// 输出 int 58
var_dump($a == $b); // bool(false)
var_dump(intval($a)); // int 57
var_dump(floatval($b) == $a);// bool(false)
var_dump(floatval($b)); // float 58
var_dump(strval($b) == strval($a)); // bool(true)
```

5. 【强制】为了防止精度损失，使用下列函数。
   说明：

```
bcadd — 将两个高精度数字相加
bccomp — 比较两个高精度数字，返回-1, 0, 1
bcdiv — 将两个高精度数字相除
bcmod — 求高精度数字余数
bcmul — 将两个高精度数字相乘
bcpow — 求高精度数字乘方
bcpowmod — 求高精度数字乘方求模，数论里非常常用
bcscale — 配置默认小数点位数，相当于就是 Linux bc 中的”scale=”
bcsqrt — 求高精度数字平方根
bcsub — 将两个高精度数字相减
```

6. 【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在 init 方法。

7. 【推荐】 类内方法定义的顺序依次是：公有方法或保护方法 > 私有方法 > getter / setter 方法。

说明：公有方法是类的调用者和维护者最关心的方法，首屏展示最好；保护方法虽然只是子类关心，也可能是“模板设计模式”下的核心方法；而私有方法外部一般不需要特别关心，是一个黑盒实现；因为承载的信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体最后。

8. 【推荐】setter 方法中，参数名称与类成员变量名称一致，this.成员名 = 参数名。在 getter/setter 方法中，不要增加业务逻辑，增加排查问题的难度。

反例：

```
function __get () {
    $this->data + 100;
}
```

9. 【推荐】循环体内，字符串的连接方式，使用 .= 方法进行扩展。

说明：下例中，每次循环都会 创建出一个临时字符串，然后进行拼接操作,造成内存资源浪费。

反例：

```
$str = "start";
for (i = 0; i < 100; i++) {
    $str = $str + "hello";
}
```

10. 【推荐】类成员与方法访问控制从严：

1） 如果不允许外部直接通过 new 来创建对象，那么构造方法必须是 private。
2） 类非 static 成员变量并且与子类共享，必须是 protected。
3） 类非 static 成员变量并且仅在本类使用，必须是 private。
4） 类 static 成员变量如果仅在本类使用，必须是 private。
5） 若是 static 成员变量，考虑是否为 final。
6） 类成员方法只供类内部调用，必须是 private。
7） 类成员方法只对继承类公开，那么限制为 protected。

说明：任何类、方法、参数、变量，严控访问范围。过于宽泛的访问范围，不利于模块解耦。思考：如果是一个 private 的方法，想删除就删除，可是一个 public 的 service 成员方法或成员变量，删除一下，不得手心冒点汗吗？变量像自己的小孩，尽量在自己的视线内，变量作用域太大，无限制的到处跑，那么你会担心的。

（五）控制语句

1. **【强制】在一个 switch 块内，每个 case 要么通过 continue/break/return 等来终止，要么注释说明程序将继续执行到哪一个 case 为止；在一个 switch 块内，都必须包含一个 default 语句并且放在最后，即使它什么代码也没有。**

说明：注意 break 是退出 switch 语句块，而 return 是退出方法体。

2. **【强制】在 if/else/for/while/do 语句中必须使用大括号。**

说明：即使只有一行代码，避免采用单行的编码方式：if (condition) statements;

3. **【强制】在高并发场景中，避免使用”等于”判断作为中断或退出的条件。**

说明：如果并发控制没有处理好，容易产生等值判断被“击穿”的情况，使用大于或小于的区间判断条件来代替。

反例：判断剩余奖品数量等于 0 时，终止发放奖品，但因为并发处理错误导致奖品数量瞬间变成了负数，这样的话，活动无法终止。

4. **【推荐】表达异常的分支时，少用 if-else 方式，这种方式可以改写成**

```
if (condition) {
    ...
    return obj;
}
// 接着写 else 的业务逻辑代码;
```

说明：如果非使用 if()...else if()...else...方式表达逻辑，避免后续代码维护困难，【强制】请勿超过 3 层。

正例：

```
function isShopOpen(string $day){
if (empty($day)) {
        return false;
   }
   $openingDays = [
        'friday', 'saturday', 'sunday'
   ];
    return in_array(strtolower($day), $openingDays, true);
}
```

反例：

```
function isShopOpen($day){
 if ($day) {
```

```
        if (is_string($day)) {
            $day = strtolower($day);
            if ($day === 'friday') {
                return true;
            } elseif ($day === 'saturday') {
                return true;
            } elseif ($day === 'sunday') {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

5. 【推荐】方法体中避免嵌套太深和提前返回

正例：

```
function fibonacci(int $n): int
{
    if ($n === 0 || $n === 1) {
        return $n;
    }
    if ($n > 50) {
        throw new \Exception('Not supported');
    }
    return fibonacci($n - 1) + fibonacci($n - 2);
}
```

反例：

```
function fibonacci(int $n)
{
    if ($n < 50) {
        if ($n !== 0) {
            if ($n !== 1) {
                return fibonacci($n - 1) + fibonacci($n - 2);
            } else {
                return 1;
            }
        } else {
            return 0;
```

```
            }
        } else {
            return 'Not supported';
        }
    }
```

6. 【推荐】不要让代码需要多次翻译,增加理解困难度

正例：

```
$locations = ['Austin', 'New York', 'San Francisco'];
foreach ($locations as $location) {
    doStuff();
    doSomeOtherStuff(); // ... // ... // ...
    dispatch($location);
}
```

反例 ：

```
$l = ['Austin', 'New York', 'San Francisco'];
for ($i = 0; $i < count($l); $i++) {
    $li = $l[$i];
    doStuff();
    doSomeOtherStuff();
    // ... // ... // ... // Wait, what is `$li` for again?
    dispatch($li);
}
```

7. 【推荐】不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明：很多 if 语句内的逻辑表达式相当复杂，与、或、取反混合运算，甚至各种方法纵深调用，理解成本非常高。如果赋值一个非常好理解的布尔变量名字，则是件令人爽心悦目的事情。

正例：

```
// 伪代码如下
$existed = ($a != null) && (...) || (...);
if ($existed) {
    ...
}
```

反例:

```
if ( ($a != null) && (...) || (...)) {
    ...
}
```

8. 【推荐】不要在其它表达式（尤其是条件表达式）中，插入赋值语句。

说明：赋值点类似于人体的穴位，对于代码的理解至关重要，所以赋值语句需要清晰地

单独成为一行。

反例：

```
$threshold = ($count = 6+1) - 1;
```

9. 【推荐】循环体中的语句要考量性能，以下操作尽量移至循环体外处理，如定义对象、变量、获取数据库连接，进行不必要的 try-catch 操作（这个 try-catch 是否可以移至循环体外）。

10. 【推荐】避免采用取反逻辑运算符。

说明：取反逻辑不利于快速理解，并且取反逻辑写法必然存在对应的正向逻辑写法。

正例：使用 if (x < 628) 来表达 x 小于 628。

反例：使用 if (!(x >= 628)) 来表达 x 小于 628。

11. 【推荐】接口入参保护，这种场景常见的是用作批量操作的接口。

12. 【参考】下列情形，需要进行参数校验：

1） 调用频次低的方法。

2） 执行时间开销很大的方法。此情形中，参数校验时间几乎可以忽略不计，但如果因为参数错误导致中间执行回退，或者错误，那得不偿失。

3） 需要极高稳定性和可用性的方法。

4） 对外提供的开放接口，不管是 RPC/API/HTTP 接口。

5） 敏感权限入口。

13. 【参考】下列情形，不需要进行参数校验：

1） 极有可能被循环调用的方法。但在方法说明里必须注明外部参数检查要求。

2） 底层调用频度比较高的方法。毕竟是像纯净水过滤的最后一道，参数错误不太可能到底层才会暴露问题。一般 DAO 层与 Service 层都在同一个应用中，部署在同一台服务器中，所以 DAO 的参数校验，可以省略。

3） 被声明成 private 只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参数已经做过检查或者肯定不会有问题，此时可以不校验参数。

（六）注释规约

1. 【强制】类、类属性、类方法的注释必须使用 phpdoc 规范，使用/**内容*/格式，不得使用// xxx 方式。

2. 【强制】所有的抽象方法（包括接口中的方法）必须要用 phpdoc 注释、除了返回值、参数、异常说明外，还必须指出该方法做什么事情，实现什么功能。
   说明：对子类的实现要求，或者调用注意事项，请一并说明。

3. 【强制】所有的类都必须添加创建者和创建日期。

4. 【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/* */注释，注意与代码对齐。

5. 【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

6. 【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持英文原文即可。
   反例：“TCP 连接超时”解释成“传输控制协议连接超时”，理解反而费脑筋。

7. 【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。
   说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严重滞后，就失去了导航的意义。

8. 【参考】谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。如果无用，则删除。
   说明：代码被注释掉有两种可能性：1）后续会恢复此段代码逻辑。2）永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（代码仓库已然保存了历史代码）。

9. 【参考】对于注释的要求：第一、能够准确反映设计思想和代码逻辑；第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看的，使其能够快速接替自己的工作。

10. 【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

反例：

```
// put elephant into fridge
put($elephant, $fridge);
```

方法名 put，加上两个有意义的变量名 $elephant 和 $fridge，已经说明了这是在干什么，语义清晰的代码不需要额外的注释。

11. 【参考】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描，经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。

1） 待办事宜（TODO）：（标记人，标记时间，[预计处理时间]）表示需要实现，但目前还未实现的功能。

2） 错误，不能工作（FIXME）：(标记人，标记时间，[预计处理时间]) 在注释中用 FIXME 标记某代码是错误的，而且不能工作，需要及时纠正的情况。

（七）其它

1. 【推荐】不要在视图模板中加入任何复杂的逻辑。

说明：根据 MVC 理论，视图的职责是展示，不要抢模型和控制器的活。

2. 【推荐】及时清理不再使用的代码段或配置信息。

说明：对于垃圾代码或过时配置，坚决清理干净，避免程序过度臃肿，代码冗余。

正例：对于暂时被注释掉，后续可能恢复使用的代码片断，在注释代码上方，统一规定使用三个斜杠(///)来说明注释掉代码的理由。

二、异常日志

（一） 异常处理

1. 【强制】异常不要用来做流程控制，条件控制。

说明：异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式要低很多。

2. 【强制】catch 时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的 catch 尽可能进行区分异常类型，再做对应的异常处理。

说明：对大段代码进行 try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。

正例：用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。

3. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。

4. 【强制】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回滚事务。

5. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。

说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

6. 【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回 null 值。

说明：本手册明确防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 null 的情况。

7. 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：

1） 数据库的查询结果可能为 null。
2） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。
3） 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。

4） 对于 Session 中获取的数据，建议进行 NPE 检查，避免空指针。

8. 【参考】对于公司外的 http/api 开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess()方法、“错误码”、“错误简短信息”。

说明：关于 RPC 方法返回方式使用 Result 方式的理由：
使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。
如果不加栈信息，只是 new 自定义异常，加入自己的理解的 error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

9. 【参考】避免出现重复的代码（Don't Repeat Yourself），即 DRY 原则。

说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。
正例：一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取：
private checkParam() {...}

（二）日志规约

1. 【强制】所有日志文件至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。网络运行状态、安全相关信息、系统监测、管理后台操作、用户敏感操作需要留存相关的网络日志不少于 6 个月。

2. 【强制】应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：

3. appName_logType_logName.log 。 logType: 日志类型， 如 stats/monitor/access 等；logName:日志描述。这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。
说明：推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。
正例：force-web 应用中单独监控时区转换异常，如：force_web_timeZoneConvert.log

4. 【强制】避免重复打印日志，浪费磁盘空间。

5. 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息,可以快速定位。

正例：各类参数或者对象+ "_" + 报错信息

6. 【强制】谨慎地记录日志。生产环境禁止输出 trace,debug 日志；有选择地输出 info 日志；如果使用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题,避免把服务器磁盘撑爆,并记得及时删除这些观察日志。

说明：大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请思考:这些日志真的有人看吗？看到这条日志你能做什么？能不能给问题排查带来好处？

7. 【推荐】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。如非必要，请不要在此场景打出 error 级别，避免频繁报警。

说明：注意日志输出的级别，error 级别只记录系统逻辑出错、异常或者重要的错误信息。

8. 【推荐】尽量用英文来描述日志错误信息，如果日志中的错误信息用英文描述不清楚的话使用中文描述即可，否则容易产生歧义。【强制】国际化团队或海外部署的服务器由于字符集问题，使用全英文来注释和描述日志错误信息。

三、单元测试

(一) 单元测试

1. 【强制】好的单元测试必须遵守 AIR 原则。

说明：单元测试在线上运行时，感觉像空气（AIR）一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。

A：Automatic（自动化）
I：Independent（独立性）
R：Repeatable（可重复）

2. 【强制】单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。

3. 【强制】保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间决不能互相调用，也不能依赖执行的先后次序。

反例：method2 需要依赖 method1 的执行，将执行结果作为 method2 的输入。

4. 【强制】单元测试是可以重复执行的，不能受到外界环境的影响。

说明：单元测试通常会被放到持续集成中，每次有代码 check in 时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等）有依赖，容易导致持续集成机制的不可用。

5. 【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。

说明：只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的交互逻辑，那是集成测试的领域。

6. 【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。

说明：新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。

7. 【强制】单元测试代码必须写在测试目录,如：src/test，不允许写在业务代码目录下。

8. 【推荐】单元测试的基本目标：语句覆盖率达到 70%；核心模块的语句覆盖率和分支覆盖率都要达到 100%

说明：在工程规约的应用分层中提到的 DAO 层，Manager 层，可重用度高的 Service，都应该进行单元测试。

9. 【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。

B：Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。

C：Correct，正确的输入，并得到预期的结果。

D：Design，与设计文档相结合，来编写单元测试。

E：Error，强制错误信息输入（如：非法数据、异常流程、业务允许外等），并得到预期的结果。

10. 【推荐】对于数据库相关的查询，更新，删除等操作，不能假设数据库里的数据是存在的，或者直接操作数据库把数据插入进去，请使用程序插入或者导入数据的方式来准备数据。

反例：删除某一行数据的单元测试，在数据库中，先直接手动增加一行作为删除目标，

但是这一行新增数据并不符合业务插入规则，导致测试结果异常。

11. 【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者对单元测试产生的数据有明确的前后缀标识。

正例：在企业智能事业部的内部单元测试中，使用 ENTERPRISE_INTELLIGENCE_UNIT_TEST_的前缀来标识单元测试相关代码。

12. 【推荐】对于不可测的代码在适当的时机做必要的重构，使代码变得可测，避免为了达到测试要求而书写不规范测试代码。

13. 【推荐】在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好覆盖所有测试用例。

14. 【推荐】单元测试作为一种质量保障手段，在项目提测前完成单元测试，不建议项目发布后补充单元测试用例。

15. 【参考】为了更方便地进行单元测试，业务代码应避免以下情况：

构造方法中做的事情过多。
存在过多的全局变量和静态方法。存在过多的外部依赖。
存在过多的条件语句。

16. 【参考】不要对单元测试存在如下误解：

那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。单元测试代码是多余的。系统的整体功能与各单元部件的测试正常与否是强相关的。单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障。

四、安全规约

（一）安全规约

1.【强制】隶属于用户个人的页面或者功能必须进行权限控制校验。
说明：防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容、修改他人的订单。

2.【强制】用户敏感数据禁止直接展示，必须对展示数据进行脱敏。
说明：中国大陆个人手机号码显示为:137****0969，隐藏中间 4 位，防止隐私泄露。
3.【强制】用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入，禁止字符串拼接 SQL 访问数据库。
4.【强制】用户请求传入的任何参数必须做有效性验证。
说明：忽略参数校验可能导致：
page size 过大导致内存溢出
恶意 order by 导致数据库慢查询任意重定向
SQL 注入
反序列化注入
正则输入源串拒绝服务 ReDoS
说明：代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。
5.【强制】禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。
6.【强制】表单、AJAX 提交必须执行 CSRF 安全验证。
说明：CSRF(Cross-site request forgery)跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户不知情的情况下对数据库中用户参数进行相应修改。
7.【强制】在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放的机制，如数量限制、疲劳度控制、验证码校验，避免被滥刷而导致资损。
说明：如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

8.【推荐】发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过滤等风控策略。

五、MySQL 数据库

(一) 建表规约

1. 【强制】表达是与否概念的字段，必须使用 is_xxx 的方式命名，数据类型是 unsigned tinyint（1 表示是，0 表示否）。

说明：任何字段如果为非负数，必须是 unsigned。
注意：数据库表示是与否的值，使用 tinyint 类型，坚持 is_xxx 的命名方式是为了明确其取值含义与取值范围。
正例：表达逻辑删除的字段名 is_deleted，1 表示删除，0 表示未删除。

2. 【强制】表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只出现数字。数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。

说明：MySQL 在 Windows 下不区分大小写，但在 Linux 下默认是区分大小写。因此，数据库名、表名、字段名，都不允许出现任何大写字母，避免节外生枝。

正例：aliyun_admin，rdc_config，level3_name

反例：AliyunAdmin，rdcConfig，level_3_name

3. 【强制】表名不使用复数名词。

说明：表名应该仅仅表示表里面的实体内容，不应该表示实体数量，对应于 DO 类名也是单数形式，符合表达习惯。

4. 【强制】禁用保留字，如 desc、range、match、delayed 等，请参考 MySQL 官方保留字。

5. 【强制】主键索引名为 pk_字段名；唯一索引名为 uk_字段名；普通索引名则为 idx_字段名。

说明：pk_ 即 primary key；uk_ 即 unique key；idx_ 即 index 的简称。

6. 【强制】小数类型为 decimal，禁止使用 float 和 double。

说明：在存储的时候，float 和 double 都存在精度损失的问题，很可能在比较值的时候，得到不正确的结果。如果存储的数据范围超过 decimal 的范围，建议将数据拆成整数和小数并分开存储。

7. 【强制】如果存储的字符串长度几乎相等，使用 char 定长字符串类型。

8. 【强制】varchar 是可变长字符串，不预先分配存储空间，长度不要超过 5000，如果存储长度大于此值，定义字段类型为 text，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

9. 【强制】表必备三字段：id, create_time, update_time。

说明：其中 id 必为主键，类型为 bigint unsigned、单表时自增、步长为 1。create_time, update_time 的类型均为 datetime 类型。

10. 【推荐】表的命名最好是遵循“业务名称_表的作用”。

正例：alipay_task / force_project / trade_config

11. 【推荐】库名与应用名称尽量一致。

12. 【推荐】如果修改字段含义或对字段表示的状态追加时，需要及时更新字段注释。

13. 【推荐】字段允许适当冗余，以提高查询性能，但必须考虑数据一致。冗余字段应遵循：

1） 不是频繁修改的字段。

2） 不是 varchar 超长字段，更不能是 text 字段。

3） 不是唯一索引的字段。

正例：商品类目名称使用频率高，字段长度短，名称基本一不变，可在相关联的表中冗余存储类目名称，避免关联查询。

14. 【推荐】单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。

说明：如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表。

15. 【参考】合适的字符存储长度，不但节约数据库表空间、节约索引存储，更重要的是提升检索速度。

正例：如下表，其中无符号值可以避免误存负数，且扩大了表示范围。

对象	年龄区间	类型	字节	表示范围
人	150 岁之内	tinyint unsigned	1	无符号值：0 到 255
龟	数百岁	smallint unsigned	2	无符号值：0 到 65535
恐龙化石	数千万年	int unsigned	4	无符号值：0 到约 42.9 亿
太阳	约 50 亿年	bigint unsigned	8	无符号值：0 到约 10 的 19 次方

（二）索引规约

1. **【强制】业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引。**

说明：不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明显的；另外，即使在应用层做了非常完善的校验控制，只要没有唯一索引，根据墨菲定律，必然有脏数据产生。

2. **【强制】超过三个表禁止 join。需要 join 的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引。**

说明：即使双表 join 也要注意表索引、SQL 性能。

3. **【强制】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度即可。**

说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分度会高达 90%以上，可以使用 count(distinct left(列名，索引长度))/count(*)的区分度来确定。

4. **【强制】页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。**

说明：索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。

5. **【推荐】如果有 order by 的场景，请注意利用索引的有序性。order by 最后的字段是组合索引的一部分，并且放在索引组合顺序的最后，避免出现 file_sort 的情况，影响查询性能。**

正例：where a=? and b=? order by c；索引：a_b_c

反例：索引如果存在范围查询，那么索引有序性无法利用，如：WHERE a>10 ORDER BY b；索引 a_b 无法排序。

6. **【推荐】利用覆盖索引来进行查询操作，避免回表。**

说明：如果一本书需要知道第 11 章是什么标题，会翻开第 11 章对应的那一页吗？目录浏览一下就好，这个目录就是起到覆盖索引的作用。

正例：能够建立索引的种类分为主键索引、唯一索引、普通索引三种，而覆盖索引只是一种查询的一种效果，用 explain 的结果，extra 列会出现：using index。

7. **【推荐】利用延迟关联或者子查询优化超多分页场景。**

说明：MySQL 并不是跳过 offset 行，而是取 offset+N 行，然后返回放弃前 offset 行，返回 N 行，那当 offset 特别大的时候，效率就非常的低下，要么控制返回的总页数，要

么对超过特定阈值的页数进行 SQL 改写。

正例：先快速定位需要获取的 id 段，然后再关联：

SELECT a.* FROM 表 1 a, (select id from 表 1 where 条件 LIMIT 100000,20) b where a.id=b.id

8. 【推荐】SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好。

说明：

1） consts 单表中最多只有一个匹配行（主键或者唯一索引），在优化阶段即可读取到数据。

2） ref 指的是使用普通的索引（normal index）。

3） range 对索引进行范围检索。

反例：explain 表的结果，type=index，索引物理文件全扫描，速度非常慢，这个 index 级别比较 range 还低，与全表扫描是小巫见大巫。

9. 【推荐】建组合索引的时候，区分度最高的在最左边。

正例：如果 where a=? and b=? ，如果 a 列的几乎接近于唯一值，那么只需要单建 idx_a 索引即可。

说明：存在非等号和等号混合时，在建索引时，请把等号条件的列前置。如：where c>? and d=? 那么即使 c 的区分度更高，也必须把 d 放在索引的最前列，即索引 idx_d_c。

10. 【推荐】防止因字段类型不同造成的隐式转换，导致索引失效。

11. 【参考】创建索引时避免有如下极端误解：

1） 宁滥勿缺。认为一个查询就需要建一个索引。

2） 宁缺勿滥。认为索引会消耗空间、严重拖慢记录的更新以及行的新增速度。

3） 抵制惟一索引。认为业务的惟一性一律需要在应用层通过“先查后插”方式解决。

（三）SQL 语句

1. 【强制】不要使用 count(列名)或 count(常量)来替代 count(*)，count(*)是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明：count(*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

2. 【强制】count(distinct col) 计算该列除 NULL 之外的不重复行数，注意 count(distinct col1, col2) 如果其中一列全为 NULL，那么即使另一列有不同的值，也返回为 0。

3. 【强制】当某一列的值全是 NULL 时，count(col)的返回结果为 0，但 sum(col)的返回结果为 NULL，因此使用 sum()时需注意 NPE 问题。

正例：使用如下方式来避免 sum 的 NPE 问题：SELECT IFNULL(SUM(column), 0) FROM table;

4. 【强制】使用 ISNULL()来判断是否为 NULL 值。

说明：NULL 与任何值的直接比较都为 NULL。

1） NULL<>NULL 的返回结果是 NULL，而不是 false。

2） NULL=NULL 的返回结果是 NULL，而不是 true。

3） NULL<>1 的返回结果是 NULL，而不是 true。

5. 【强制】代码中写分页查询逻辑时，若 count 为 0 应直接返回，避免执行后面的分页语句。

6. 【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

说明：以学生和成绩的关系为例，学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

7. 【强制】禁止使用存储过程，存储过程难以调试和扩展，更没有移植性。

8. 【强制】数据订正（特别是删除、修改记录操作）时，要先 select，避免出现误删除，确认无误才能执行更新语句。

9. 【推荐】in 操作能避免则避免，若实在避免不了，需要仔细评估 in 后边的集合元素数量，控制在 1000 个之内。

10. 【参考】如果有国际化需要，所有的字符存储与表示，均以 utf-8 编码，注意字符统计函数的区别。

说明：

SELECT LENGTH("轻松工作"); 返回为 12
SELECT CHARACTER_LENGTH("轻松工作"); 返回为 4
如果需要存储表情，那么选择 utf8mb4 来进行存储，注意它与 utf-8 编码的区别。

11. 【参考】TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少，但 TRUNCATE 无事务且不触发 trigger，有可能造成事故，故不建议在开发代码中使用此语句。
说明：TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同。

（四）ORM 映射

1. 【强制】在表查询中，一律不要使用 * 作为查询的字段列表，需要哪些字段必须明确写明。
说明：1）增加查询分析器解析成本。
2）无用字段增加网络消耗，尤其是 text 类型的字段。

2. 【强制】更新数据表记录时，必须同时更新记录对应的 update_time 字段值为当前时间。

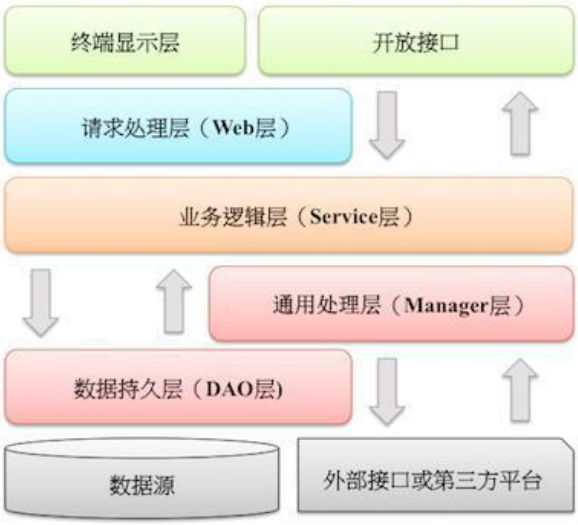
3. 【推荐】不要写一个大而全的数据更新接口。不管是不是自己的目标更新字段，都进行 update table set c1=value1,c2=value2,c3=value3; 这是不对的。执行 SQL 时，不要更新无改动的字段，一是易出错；二是效率低；三是增加 binlog 存储。

4. 【参考】事务不要滥用。事务会影响数据库的 QPS，另外使用事务的地方需要考虑各方面的回滚方案，包括缓存回滚、搜索引擎回滚、消息补偿、统计修正等。

六、工程结构

（一）应用分层

1. 【推荐】图中默认上层依赖于下层，箭头关系表示可直接依赖，如：开放接口层可以依赖于 Web 层，也可以直接依赖于 Service 层，依此类推：



- 开放接口层：可直接封装 Service 方法暴露成 RPC 接口；通过 Web 封装成 http 接口；进行网关安全控制、流量控制等。
- 终端显示层：各个端的模板渲染并执行显示的层。当前主要是 velocity 渲染，JS 渲染，JSP 渲染，移动端展示等。
- Web 层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
- Service 层：相对具体的业务逻辑服务层。
- Manager 层：通用业务处理层，它有如下特征：
  1） 对第三方平台封装的层，预处理返回结果及转化异常信息。
  2） 对 Service 层通用能力的下沉，如缓存方案、中间件通用处理。
  3） 与 DAO 层交互，对多个 DAO 的组合复用。
- DAO 层：数据访问层，与底层 MySQL、Oracle、Hbase 等进行数据交互。
- 外部接口或第三方平台：包括其它部门 RPC 开放接口，基础平台，其它公司的 HTTP 接口

2. 【参考】(分层异常处理规约) 在 DAO 层，产生的异常类型有很多，无法用细粒度的异常进行 catch，使用 catch(Exception e)方式，并 throw new DAOException(e)，不需要打印日志，因为日志在 Manager/Service 层一定需要捕获并打印到日志文件中去，如果同台服务器再打日志，浪费性能和存储。在 Service 层出现异常时，必须记录出错日志到磁盘，尽可能带上参数信息，相当于保护案发现场。如果 Manager 层与 Service 同机部署，日志方式与 DAO 层处理一致，如果是单独部署，则采用与 Service 一致的处理方式。Web 层绝不应该继续往上抛异常，因为已经处于顶层，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面，加上用户容易理解的错误提示信息。开放接口层要将异常处理成错误码和错误信息方式返回。

3. 【参考】分层领域模型规约：

DO（Data Object）：此对象与数据库表结构一一对应，通过 DAO 层向上传输数据源对象。

DTO（Data Transfer Object）：数据传输对象，Service 或 Manager 向外传输的对象。

BO（Business Object）：业务对象，由 Service 层输出的封装业务逻辑的对象。

AO（Application Object）：应用对象，在 Web 层与 Service 层之间抽象的复用对象模型，极为贴近展示层，复用度不高。

VO（View Object）：显示层对象，通常是 Web 向模板渲染引擎层传输的对象。

Query：数据查询对象，各层接收上层的查询请求。注意超过 2 个参数的查询封装，禁止使用 Map 类来传输。

七、设计规约

（一）设计规约

1. 【强制】存储方案和底层数据结构的设计获得评审一致通过，并沉淀成为文档。

说明：有缺陷的底层数据结构容易导致系统风险上升，可扩展性下降，重构成本也会因历史数据迁移和系统平滑过渡而陡然增加，所以，存储方案和数据结构需要认真地进行设计和评审，生产环境提交执行后，需要进行 double check。

正例：评审内容包括存储介质选型、表结构设计能否满足技术方案、存取性能和存储空间能否满足业务发展、表或字段之间的辩证关系、字段名称、字段类型、索引等；数据结构变更（如在原有表中新增字段）也需要进行评审通过后上线。

2. 【强制】在需求分析阶段，如果与系统交互的 User 超过一类并且相关的 User Case 超过 5 个，使用用例图来表达更加清晰的结构化需求。

3. 【强制】如果某个业务对象的状态超过 3 个，使用状态图来表达并且明确状态变化的各个触发条件。

说明：状态图的核心是对象状态，首先明确对象有多少种状态，然后明确两两状态之间是否存在直接转换关系，再明确触发状态转换的条件是什么。

正例：淘宝订单状态有已下单、待付款、已付款、待发货、已发货、已收货等。比如已下单与已收货这两种状态之间是不可能有直接转换关系的。

4. 【强制】如果系统中某个功能的调用链路上的涉及对象超过 3 个，使用时序图来表达并且明确各调用环节的输入与输出。

说明：时序图反映了一系列对象间的交互与协作关系，清晰立体地反映系统的调用纵深链路。

5. 【强制】如果系统中模型类超过 5 个，并且存在复杂的依赖关系，使用类图来表达并且明确类之间的关系。

说明：类图像建筑领域的施工图，如果搭平房，可能不需要，但如果建造蚂蚁 Z 空间大楼，肯定需要详细的施工图。

6. 【强制】如果系统中超过 2 个对象之间存在协作关系，并且需要表示复杂的处理流程，使用活动图来表示。

说明：活动图是流程图的扩展，增加了能够体现协作关系的对象泳道，支持表示并发等。

7. 【推荐】需求分析与系统设计在考虑主干功能的同时，需要充分评估异常流程与业务边界。

反例：用户在淘宝付款过程中，银行扣款成功，发送给用户扣款成功短信，但是支付宝入款时由于断网演练产生异常，淘宝订单页面依然显示未付款，导致用户投诉。

8. 【推荐】类在设计与实现时要符合单一原则。

说明：单一原则最易理解却是最难实现的一条规则，随着系统演进，很多时候，忘记了类设计的初衷。

9. 【推荐】谨慎使用继承的方式来进行扩展，优先使用聚合/组合的方式来实现。

说明：不得已使用继承的话，必须符合里氏代换原则，此原则说父类能够出现的地方子类一定能够出现，比如，“把钱交出来”，钱的子类美元、欧元、人民币等都可以出现。

10. 【推荐】系统设计时，根据依赖倒置原则，尽量依赖抽象类与接口，有利于扩展与维护。

说明：低层次模块依赖于高层次模块的抽象，方便系统间的解耦。

11. 【推荐】系统设计时，注意对扩展开放，对修改闭合。

说明：极端情况下，交付线上生产环境的代码都是不可修改的，同一业务域内的需求变化，通过模块或类的扩展来实现。

12. 【推荐】系统设计阶段，共性业务或公共行为抽取出来公共模块、公共配置、公共类、公共方法等，避免出现重复代码或重复配置的情况。

说明：随着代码的重复次数不断增加，维护成本指数级上升。

13. 【推荐】避免如下误解：敏捷开发 = 讲故事 + 编码 + 发布。

说明：敏捷开发是快速交付迭代可用的系统，省略多余的设计方案，摒弃传统的审批流程，但核心关键点上的必要设计和文档沉淀是需要的。

反例：某团队为了业务快速发展，敏捷成了产品经理催进度的借口，系统中均是勉强能运行但像面条一样的代码，可维护性和可扩展性极差，一年之后，不得不进行大规模重构，得不偿失。

14. 【参考】系统设计主要目的是明确需求、理顺逻辑、后期维护，次要目的用于指导编码。

说明：避免为了设计而设计，系统设计文档有助于后期的系统维护和重构，所以设计结果需要进行分类归档保存。

15. 【参考】设计的本质就是识别和表达系统难点，找到系统的变化点，并隔离变化点。

说明：世间众多设计模式目的是相同的，即隔离系统变化点。

16. 【参考】系统架构设计的目的：

确定系统边界。确定系统在技术层面上的做与不做。

确定系统内模块之间的关系。确定模块之间的依赖关系及模块的宏观输入与输出。

确定指导后续设计与演化的原则。使后续的子系统或模块设计在规定的框架内继续演化。确定非功能性需求。非功能性需求是指安全性、可用性、可扩展性等。

17. 【参考】在做无障碍产品设计时，需要考虑到：

所有可交互的控件元素必须能被 tab 键聚焦，并且焦点顺序需符合自然操作逻辑。用于登陆校验和请求拦截的验证码均需提供图形验证以外的其它方式。

自定义的控件类型需明确交互方式。