如何成为一枚高贵的 PHP 攻城狮

语言特色

> PHP确实是一门特色的语言(仅次于Python)。

面向对象

自PHP5开始，PHP具有完整的对象模型，包括类，抽象类，接口，继承，构造函数，反射，克隆和异常等。以下是PHP5.3-PHP7.2中关于面向对象的新特性:

- 命名空间(PHP5.3,更好的组织类)
- 性状 (PHP5.4,解决了多继承场景的问题)
- 匿名类(PHP7.0)
- 导入一组类(PHP7.0)
- 类常量的可见性(PHP7.1)
- 多异常捕获处理 (PHP7.1)
- 允许重写抽象方法(PHP7.2)

函数式

与JAVA不同的是，PHP将函数视为”一等公民”，如果你偏爱函数式编程的话，PHP也不会让你失望。以下是PHP5.3-PHP7.2中关于函数式的新特性:

- 闭包(PHP5.3)
- 闭包支持$this(PHP5.4)
- Closure::call() (PHP7.0)

异步编程

众所周知,PHP是一门单线程同步语言。在PHP5.5之前，如果想要提高性能只能通过多进程或者多线程(pthreads拓展)的方式来编写代码。但在PHP5.5官方推出的新特性Generator，异步成为了可能。以下是PHP5.3-PHP7.2中关于异步编程的新特性:

- Generator(PHP5.5)

- SplDoublyLinkedList 双链表
- SplStack 栈
- SplQueue 队列
- SplHeap 堆
- SplMaxHeap 最大堆
- SplMinHeap 最小堆
- SplPriorityQueue 优先队列
- SplFixedArray 固定数组
- SplObjectStorage 对象容器

迭代器

- AppendIterator 附加迭代器
- ArrayIterator 数组迭代器
- CachingIterator 缓存迭代器
- CallbackFilterIterator — 回调过滤迭代器
- DirectoryIterator — 目录迭代器
- EmptyIterator — 空迭代器
- FilesystemIterator — 文件系统迭代器
- FilterIterator — 过滤迭代器
- GlobIterator — Glob迭代器
- InfiniteIterator — 无限迭代器
- IteratorIterator — 迭代器迭代器
- LimitIterator — 限定迭代器
- MultipleIterator — 多重迭代器
- NoRewindIterator — 无倒回迭代器
- ParentIterator — 父迭代器
- RecursiveArrayIterator — 递归数组迭代器
- RecursiveCachingIterator — 递归缓存迭代器
- RecursiveCallbackFilterIterator — 递归回调过滤迭代器
- RecursiveDirectoryIterator — 递归目录迭代器
- RecursiveFilterIterator — 递归过滤迭代器
- RecursiveIteratorIterator — 递归迭代迭代器
- RecursiveRegexIterator — 递归正则迭代器
- RecursiveTreeIterator — 递归树状结构迭代器
- RegexIterator — 正则迭代器

接口

- Countable —可计算接口
- OuterIterator —外部迭代器接口
- RecursiveIterator —递归迭代器接口
- SeekableIterator —可查找迭代器接口

函数

- class_implements — 返回指定的类实现的所有接口。
- class_parents —返回指定类的父类
- class_uses —返回指定类使用的性状
- iterator_apply —为迭代器中每个元素调用一个用户自定义函数
- iterator_count —计算迭代器中元素的个数
- iterator_to_array —将迭代器中的元素拷贝到数组
- spl_autoload_call —尝试调用所有已注册的__autoload()函数来装载请求类
- spl_autoload_extensions —注册并返回spl_autoload函数使用的默认文件扩展名
- spl_autoload_functions —返回所有已注册的__autoload()函数。
- **spl_autoload_register** —注册给定的函数作为 __autoload 的实现
- spl_autoload_unregister — 注销已注册的__autoload()函数
- spl_autoload —__autoload()函数的默认实现
- spl_classes — 返回所有可用的SPL类
- spl_object_hash —返回指定对象的hash id

文件处理

- SplFileInfo —SplFileInfo类为单个文件的信息提供了高层次面向对象的接口。
- SplFileObject —SplFileObject为文件提供了一个面向对象接口。
- SplTempFileObject —SplFileObject为临时文件提供了一个面向对象接口。

各种类及接口

- **ArrayObject** —ArrayObject类允许对象作为数组使用
- SplAbserver —SplAbserver 接口与SplSubject一起使用以实现观察者设计模式。
- SplSubject —SplSubject接口与SplObserver一起使用以实现观察者设计模式。

最好的语言为何被怼

> 某MM：你能让这个论坛的人都吵起来，我今晚就跟你走。
> 某软件工程师：PHP是最好的语言！
> 某论坛真的就炸锅了，各种吵架……
> 某MM：服了你了，我们走吧，你想干啥都行。
> 某软件工程师：今天不行，我一定要说服他们，PHP必须是最好的语言……

编程语言的鄙视链中PHP总是被黑的最惨的那一个。但为什么一部分人叫嚣着PHP是世界上最好的语言，而一部分人各种怼PHP呢？几个原因：

由于一开始PHP的初衷仅仅只是一个工具而已，再加上Lerdorf尽管是个非常优秀的程序员，但是语言设计方面不是他所专长，导致PHP的有非常多的历史包袱。

- 混乱的函数命名，Linux C风格与驼峰命名共存
- 被误用的字符串连接符(.)
- 混乱的参数顺序
- 缺乏对多线程的支持

由于PHP始终如一的向下兼容，PHP仍会背着这些包袱努力向前。

无知者的偏见

PHP确实如上述所说存在各种各样的问题，但PHP社区与PHP的开发者们一直在努力着。然而有些无知者从未真正了解过PHP，甚至都没有使用过，仅仅凭借某些社区论坛上一些跟风的言论就认定PHP是一门差劲的语言实在令人无奈。甚至我的某位大龄移动端同事至今还认为PHP的变量命名是 `var` 。

规范还是规范

> 无规矩不成方圆，社会如此，团队如此，工程如此。

从我们还很小的时候，爸妈就开始教育我们什么可以做，什么不可以做。比如”吃饭前要洗手”,”不能在公共场合大声喧哗”。等我们进入学校，乃至步入社会，我们明白总有各种条条款款约束着我们，可能是组织层面的，可能是道德层面的，也可能是法律层面的。我们要成为一个受他人尊重，被社会认可的人就必须坚守这些规范。

世界上没有完美的事物，即使是再优秀的规范也不是完美的。好的规范可以减少人与人之间的协作成本但同时也会阻碍灵活性，如何把握规范约束的度值得推敲。

在没有PSR规范之前，代码命名，文件命名，自动加载都没有统一的标准，各个项目都自我约定规范或处于无规范状态。后来PHPFIG为了实现组件的互操作性，并为实现最佳编程和测试实践提供规范而制定了PSR规范。PHPFIG的成员包含了许多知名的开源项目作者，例如CakePHP作者，Composer作者，Drupal作者，Phalcon作者，Symfony作者，Yii作者等，详细成员链接请点击此处。PSR并不是官方规范，然而众多知名开源项目作者参与并制定使之成为PHP项目的事实规范。

PSR规范目前已有9个已实施的规范，8个处于草案阶段的规范，1个被废弃的规范共18个。以下是根据序号排列的规范：

- PSR-0 自动加载规范（已废弃）
- PSR-1 基础编码规范（已实施）
- PSR-2 编码风格指南（已实施）
- PSR-3 日志接口（已实施）

已实施的规范

PSR-1 基础编码规范

PSR-1仅仅只对文件，类与方法进行了规范。

- 文件必须只使用 `<?php` 或者 `<?=`
- 文件必须是无BOMUTF-8编码
- 文件要不要不声明类，方法，常量等，要不用来生成输出，改变.ini设定等
- 命名空间与类必须遵守PSR-4(**PSR-0已废弃**)
- 类名的命名格式必须是**大驼峰**，如： `MyClass`
- 常量的命名格式必须是**大写+下划线链接**，如： `MY_CONSTANTS`
- 方法的命名格式必须是**小驼峰**，如： `myFunction`

PSR-2 编码风格指南

PSR-2是对PSR-1的拓展与延伸，在缩进，行的字符数限制，空格，空行，类与方法的可见性等方面进行了规范。

- 代码必须遵守PSR-1规范
- 缩进必须是四个空格，而不能是 `TAB`
- 对行的字符数并没有硬限制，但是软限制在120个字符。建议80个字符以内。
- 每个 `namespace` 命名空间声明语句和 `use` 声明语句块后面，必须插入一个空白行。范例如下:

```
namespace Vendor\Package;

use FooInterface;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;
```


- 类的开始花括号 `{` 必须在函数声明后自成一行，结束花括号 `}` 也必须写在函数主体后自成一行。范例如下：

```
//class declare
class Foo
{
    //class body
}
```

- 方法的开始花括号 `{` 必须在方法声明后自成一行，结束花括号 `}` 也必须写在方法主体后自成一行。范例如下：

```
class Foo
{
    //method declare
```

```
    public function sampleMethod($a, $b = null)
    {
    }

    //abstract必须在访问修饰符前面
    abstract public function sampleAbstractMethod();

    //static必须声明在访问修饰符后面
    final public static function sampleFinalStaticMethod()
    {
    }
}
```

- 控制结构的关键字后必须要有一个空格符，而调用方法或函数时则一定不能有。范例如下：
- 控制结构的开始花括号 { 必须写在声明的同一行，而结束花括号 } 必须写在主体后自成一行。范例如下：
- 控制结构的开始左括号（后和结束右括号）前，都一定不能有空格符。范例如下：

```
class Foo
{
    public function sampleMethod($a, $b = null)
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }
```


PSR-3 日志接口

PSR-3定义了一个通用的日志接口，为了使日志类库以简单通用的方式，通过接收一个Psr\Log\LoggerInterface 对象，来记录日志信息。

实现了PSR-3规范的类库

- monolog Total Downloads Latest Stable Version Reference Status

PSR-4 自动加载规范

PSR-4定义了关于由文件路径自动加载对应类的相关规范。

4. 自动加载器（autoloader）的实现一定不能抛出异常、一定不能触发任一级别的错误信息以及不应该有返回值

例子

下表展示了符合规范完整类名、命名空间前缀和文件基目录所对应的文件路径。

完整类名	命名空间前缀	基础目录	文件路径
\Acme\Log\Writer\File_Writer	Acme\Log\Writer	./acme-log-writer/lib/	./acme-log-writer/lib/File_Writer.php
\Aura\Web\Response\Status	Aura\Web	/path/to/aura-web/src/	/path/to/aura-web/src/Response/Status.php
\Symfony\Core\Request	Symfony\Core	./vendor/Symfony/Core/	./vendor/Symfony/Core/Request.php
\Zend\Acl	Zend	/usr/includes/Zend/	/usr/includes/Zend/Acl.php

关于本规范的实现，可参考相关实例

PSR-6 缓存接口

缓存是提升项目性能的常规手段，同样也是大多数框架或类库的常规特性之一。这导致不同的框架或类库存在不同的级别的对于缓存的解决方案。这种差异使得开发人员不得不学习多个系统，甚至这些系统都不能提供给他们所需的功能。此外缓存库的开发人员也在面临一个抉择:是只支持少数的框架，还是编写大量的适配器去适配各式各样的系统。

PSR-6将解决这个问题，库和框架开发人员可以指望缓存系统按照预期的方式工作，而缓存系统的开发人员只需要实现一组接口，而不是一个完整的适配器。

PSR-7 HTTP消息接口

HTTP消息是Web开发的基础。Web浏览器和HTTP客户端（如cURL）创建发送到Web服务器的HTTP请求消息，它提供HTTP响应消息。服务器端代码接收HTTP请求消息，并返回HTTP响应消息。

HTTP消息通常由终端用户中抽象出来，但作为开发人员，我们通常需要知道它们是如何构造的，以及如何访问或操作它们以执行我们的任务，无论是对HTTP API请求，还是处理传入请求。

每个HTTP请求消息都有特定的形式：

```
POST /path HTTP/1.1
Host: example.com

foo=bar&baz=bat
```

- guzzlehttp/psr7: Total Downloads
- oscarotero/psr7-middlewares Total Downloads
- zendframework/zend-diactoros Total Downloads

PSR-11 容器接口

PSR-11是依赖注入容器的通用接口，旨在规范框架或库通过容器获取对象与参数。

实现了PSR-11的类库

- PHP-DI/PHP-DI Total Downloads

PSR-13 超媒体链接

无论是HTML内容还是不同的API格式内容，超媒体链接正日益成为WEB重要的部分。然而，没有单一的通用超媒体格式，也没有一种通用的方式来表示格式之间的链接。

PSR-13旨在为PHP开发人员提供一种简单、通用的、独立于使用的序列化格式的方式来表示超媒体链接。进而允许系统将超媒体链接的响应序列化为一个或多个有线格式而独立于决定这些链接应该是什么的过程。

PSR-16 简单缓存

PSR-16旨在规范一个用于缓存项和高速缓存驱动程序的简单但可拓展的接口。

框架太多的痛苦

> 既然有语言之争，那么框架之争也变得理所应当，作为一门专注于Web领域的语言，PHP有着太多的框架。有时候你是否会觉得学不完的框架很让人苦恼？

我们先来看下有哪些主流框架(按照Github Star 降序排列)

Laravel

Laravel是目前PHP语言中最火的框架，没有之一。以优雅为目标，用了许许多多php的新特性，活跃的社区，丰富的生态环境。最新LTS Laravel5.5 已经要求环境必须在php7.0以上了。可以说是PHP界最时尚的框架了。

Phalcon是PHP界最快的MVC框架，采用Zend Extension的形式进行开发。然而由于对PHP开发而言太过黑盒，且开发难度较大，所以Phalcon团队又开发出一门语言Zephir 来编写Phalcon，大家可以尝试一下。

Slim

Slim是一款麻雀虽小五脏俱全的框架，非常建议新手对其源码进行研究。如果编写小项目，采用Slim会非常的得心应手。

Cakephp

CakePHP在国内用的人数不是太多，但这并不影响作为老牌框架的地位。

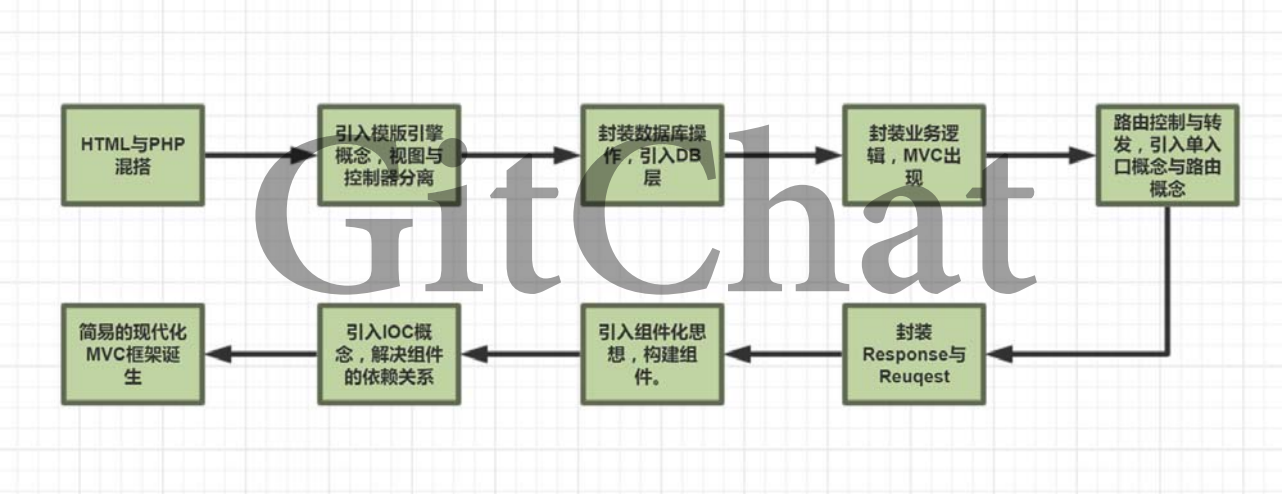
Lumen

Lumen是一款专注于构建微服务架构和 API 应用的框架。

除了以上框架之外，还有鸟哥的Yaf,国产的ThinkPHP与PHP官方的Zend Framework也比较知名。

框架的演化

那么多的框架，学完何时是个头啊?所以我们要求同存异。首先我们来审视下一个框架是如何演化的。



可以看出一个框架最基本的功能都在这张图中。下一步我们来学习这些通过框架具体有哪些。

通用组件

- FastRoute
- klein

Controller

Controller组件主要用于处理业务逻辑，并与Model组件或View组件进行交互。

Model
PHP的Model一般是充血模型，Modle组件封装了具体的业务模型逻辑并实现了ORM，屏蔽了SQL细节，可以让开发像操作对象一样操作数据库。

相关的开源组件：

- doctrine2
- laravel-mongodb
- Propel2

View

View组件一般是一个模板引擎。

相关的开源组件：

- Twig
- smarty

ErrorHander

ErrorHander组件主要用来捕获异常与错误信息并进行相关的处理。

相关的开源组件：

- collision

IOC Container

IOC Container 通过反射来解决类与类之间的依赖关系。

相关的开源组件：

- PHP-DI

没有注释是可耻的

对于第一种说辞，其实是强调代码的**自说明性**，好的代码会自己说话。然后在这个**自说明**仅仅强调了这个代码的逻辑作用，却无法附带更多的信息(在下面的注释类型中会讲解)。其次对于大部分并不能很好的书写确切、精炼、自说明性的代码的开发而言，这个言论更像是借口。

对于第二种说辞，是强调了错误注释的危害而认为不应该写注释，这个逻辑就像是**努力了可能没结果所以不努力一样。**

对于第三种说辞，是强调了书写注释的成本太过高昂，然而对比不写注释带来的沟通成本、再次阅读理解的时间与精力成本而言，这点成本显然是值得的。

注释的作用

注释不仅仅只是为了解释代码，它还有其它的作用，以下是注释的几种作用：

重复代码

重复性的注释只是把代码用不同的文字再去描述一边，除了给读者增加阅读量之外并没有提供更多有用的信息。

解释代码

解释性注释通常用来解释复杂、歧义的代码块。通常是由于代码含糊不清才会体现出这类注释的价值。如果代码是因为复杂才需要解释的话，最好的方式是改进代码，也就是前辈所说的”好的代码不需要注释”。

代码标记

标记性的注释并非有意留在代码中的，它是用来提示开发者有些工作没有完成。例如: `//todo` 。

概述代码

概述代码是把若干行代码用一两句话概述，因为读者读注释肯定比读代码快。概述性质的注释对于那些要修改你代码的人很有用。

代码意图说明

目的性注释用来指明一段代码的意图，它指出要解决的问题而非解决的方法。

传达代码无法表述的信息

某些信息不能通过代码来表示，但是又必须包含在源代码中。这种注释包括:

- 版权声明、版本号、作者等杂项信息
- 代码设计相关的注意事项
- 参考链接
- 诸如PHPDoc,JavaDoc,JsDoc等编辑器要求有的注释

这个 @author 用来表明谁创建了这个结构元素(**指文件、类、接口、性状、方法、函数、常量、变量**)或者对它进行了重要的修改。此标记还可以包含电子邮件地址。

- 例子

```
/**
 * @author My Name
 * @author My Name <my.name@example.com>
 */
```

@copyright

这个 @copyright 标签被用来记录结构元素的版权信息

- 语法

```
@copyright <description>
```

- 描述

@copyright 定义了谁拥有”结构元素” 的版权信息，除非另有说明，否则使用这个标记的版权适用于它应用的”结构元素”和它的子元素。

描述的格式受每个项目的编码标准的控制。建议提及著作权和涉及的组织所涵盖的年份。

- 例子

```
/**
 * @copyright 1997-2005 The PHP Group
 */
```


@deprecated

@deprecated 标记用于指明哪些“结构元素”被弃用，并在以后的版本中删除。

- 语法

```
@deprecated [<"Semantic Version">][:<"Semantic Version">] [<description>]
```

- 描述

@deprecated 表明在将来的版本中将删除相关的“结构元素”，因为它已经过时或者不推荐使用它的用法

```
/**
 * @deprecated
 *
 * @deprecated 1.0.0:2.0.0
 * @see \New\Recommended::method()
 *
 * @deprecated 1.0.0
 *
 * @deprecated :2.0.0
 *
 * @deprecated No longer used by internal code and not recommended.
 *
 * @deprecated 1.0.0 No longer used by internal code and not recommended.
 */
```

@method

`@method` 允许一个类知道哪些“魔法”方法是可调用的。

- 语法

```
@method [return type] [name]([type] [parameter], [...]) [description]
```

- 描述

  `@method` 标记用于类包含 `__call()` 魔术方法并定义一些明确用途的情况。

  有一个子类，它的父类有一个 `__call()` 来拥有动态的getter或setter来预定义的属性。子类知道哪些getter和setter需要存在，但是却依赖父类的 `__call()` 方法来提供它。在这种情况下，子类将为每个魔术setter或getter方法使用 `@method` 标记。

  `@method` 标记允许作者通过在签名中包含这些类型来指定传递参数和返回值的类型。

  当预期的方法没有返回值时，可以省略返回类型，在这种情况下默认返回类型为 `void`;。

  `@method` 标签不能在没有与类或接口相关联的PHPDoc中使用。

- 例子

```
class Parent
{
    public function __call()
    {
        <...>
```

```
@param ["Type"] [name] [<description>]
```

- 描述

  指明函数或方法的参数的类型和功能的时候可以使用 `@param` 标记。当提供时，它必须包含一个“类型”来指示预期的内容；另一方面，描述是可选的，但建议使用。对于复杂的结构，如选择阵列，建议使用“内联phpDoc”描述选项数组。

  `@param` 可以有多行的描述，不需要明确的界定。

  建议在每个函数和方法使用这个标记。

- 例子

```
/**
 * Counts the number of items in the provided array.
 *
 * @param mixed[] $items Array structure to count the elements of.
 *
 * @return int Returns the number of elements.
 */
function count(array $items)
{
    <...>
}
```

下面的示例演示了用一个“内联phpDoc”的来标记一个具有两个元素：’required’ 和 ‘label’ 的可选数组。


```
/**
 * Initializes this class with the given options.
 *
 * @param array $options {
 *     @var bool   $required Whether this element is required
 *     @var string $label    The display name for this element
 * }
 */
public function __construct(array $options = array())
{
    <...>
}
```

@property

```
class Parent
{
    public function __get()
    {
        <...>
    }
}

/**
 * @property string $myProperty
 */
class Child extends Parent
{
    <...>
}
```

@return

`@return` 标记用来指明函数或方法的返回值

- 语法

```
@return <"Type"> [description]
```

- 描述

  当指明函数或者方法的返回值时可以使用 `@return` ,当使用 `@return` 的时候一定要包含”Type”来指定返回什么类型；另一方面的描述是可选的，但在复杂的返回结构，如关联数组时，是推荐的。

  `@return` 标签可能有一个多行描述，不需要显式的限制。

  建议在每个函数和方法中使用这个标记。唯一的例外是：正如任何一个项目的编码标准所定义的那样，**函数与方法并没有返回值**：这时 `@return` 可以省略。

- 例子

```
/**
 * @return int Indicates the number of items.
 */
function count()
{
    <...>
}
```

@ throw 标记可以用来指示“结构元素”抛出特定类型的错误。
此标记提供的类型必须是异常类。

此标记用于在您的文档中显示可能发生的错误，以及在何种情况下可能发生错误。建议提供描述异常抛出的原因的说明。

建议在需要抛出异常的地方都使用此标记，即使它具有相同的类型。通过记录每一个事件，创建一个详细的视图，并且消费者知道要检查哪些错误。

- 例子

```
/**
 * Counts the number of items in the provided array.
 *
 * @param mixed[] $array Array structure to count the elements of.
 *
 * @throws InvalidArgumentException if the provided argument is not of type
 *     'array'.
 *
 * @return int Returns the number of elements.
 */
function count($items)
{
    <...>
}
```

@var

您可以使用 @ var 标记来记录以下“结构元素”的“类型”:

- 常量，包括类常量活全局常量
- 属性
- 变量，包括全局变量或局部变量

- 语法

```
@var ["Type"] [element_name] [<description>]
```

- 描述

@var 标记定义了由一个常量、属性或变量的值表示的数据类型。

在类型不明确或未知的情况下，每个常量或属性定义或变量应该在包含 @var 标记的DocBlock之前。任何其他变量可能在前面加上一个包含 @var 标记的DocBlock。

```
class Foo
{
  /** @var string|null Should contain a description */
  protected $description = null;

  public function setDescription($description)
  {
      // there should be no docblock here
      $this->description = $description;
  }
}
```

另一个例子是明确地声明foreach中的变量。许多IDE会使用这些信息来自动完成对item的声明。

```
/** @var \Sqlite3 $sqlite */
foreach($connections as $sqlite) {
    // there should be no docblock here
    $sqlite->open('/my/database/path');
    <...>
}
```

同样可以用于复合声明

```
class Foo
{
  protected
      /**
       * @var string Should contain a description
       */
      $name,
      /**
       * @var string Should contain a description
       */
      $description;
}
```


或者常量

```
class Foo
{
  const
      /**
       * @var string Should contain a description
```

- 单元测试:是将一个完整的类或者函数从完整的系统中隔离出来进行测试。
- 集成测试:是对两个或更多的类库、类进行的联合测试。这种测试通常在有了两个可以测试的类就可以开始了，并且一直持续到系统开发完成。
- 回归测试:是指重复执行以前的测试用例，以便在原先通过了相同测试集合的软件中查找缺陷。
- 系统测试:是在最终的配置下运行整个软件。以便测试安全、性能、资源消耗、时方面的问题以及其它无法在低级集成上测试的问题。

为什么要写测试用例

- 更好的保障代码质量
- 避免重复性的测试，节省时间
- 梳理业务逻辑

测试先行还是测试后行

有时候开发会很困惑，到底是先编写测试用例还是编写完代码之后再编写测试用例呢？答案是测试现行，原因如下：

- 在写代码之前编写测试用例并不会比写代码之后编写测试用例花的时间更长。
- 假设你首先编写测试用例，你会更早的发现缺陷，更早的去修正他们。
- 首先编写测试用例会让你更早的去思考需求与设计，更好的去编写高质量代码。
- 在编写代码之前写测试用例可以更早的暴露需求上的问题。

不写测试用例的项目真的比写测试用例的项目开发快吗？

似乎大部分开发都意识到编写测试用例对于项目开发的益处。然而对于时间成本的考量又是那些不愿编写测试用例的开发的顾虑所在。

那么我们来思考编写测试用例的作用—**发现缺陷并修正**，这个缺陷可能是需求上的，也可以是具体编码上的。我们知道缺陷发现的越早，修正的成本(时间与精力)越低廉。需求上的缺陷导致的成本又高于编码上的缺陷。如果通过编写测试用例来提早发现需求上的缺陷并修正，远比项目即将交付时发现缺陷修正花费更少的时间。对比可能出现的返工或重新的需求修正与评估，编写测试用例的花费的时间成本反而更少。

相关的测试框架

PHPUnit

PHPUnit是xUnit家族的一员，更是最有名的PHP测试框架。

Behat

Behat 是一款BDD(行为驱动测试)框架。

团队对个人的情感影响

就仿佛有了一个安全的港湾，你可以在外面努力的闯荡，但当你遇到问题时，遇到挫折时，团队一定可以给予你精神上的鼓舞。

团队对个人的能力性格上的补全

在团队中每个人在技能、特长、知识、年龄、性格等方面具有互补性，你可以通过同事的视角获取一个不一样的观念来补全你能力上性格上的短板。

朝着目标前进

每个团队一定都会有一个既是团队又是个人的目标，朝着这个目标努力的奋斗的路途中学会信任，学会理解，学会牺牲。

方法论与 'X'DD

> 假设世界上有那么一个人，他拥有无限的精力，最高的效率，最快的速度以及最完备的知识体系，那么他做任何事的效率都是100%。然而世界上并没有这么一个人，但我们希望通过各自方法淬炼团队，使团队的工作效率无限趋向于这个人。

方法论

方法论是什么

方法论是一种以解决问题为目标的理论体系或系统，通常涉及对问题阶段、任务、工具、方法技巧的论述。方法论会对一系列具体的方法进行分析研究、系统总结并最终提出较为一般性的原则。

软件研发中的方法论

软件研发是一门工程学，为了使团队可以更高效更准确的完成某一个任务，许多前辈提出了以下几种方法论。

- 瀑布流开发
- 迭代模型
- 螺旋式开发
- 敏捷开发

- 瀑布流开发

- 是一种严格线性的、按阶段顺序的、逐步细化的开发模式

适用场景：

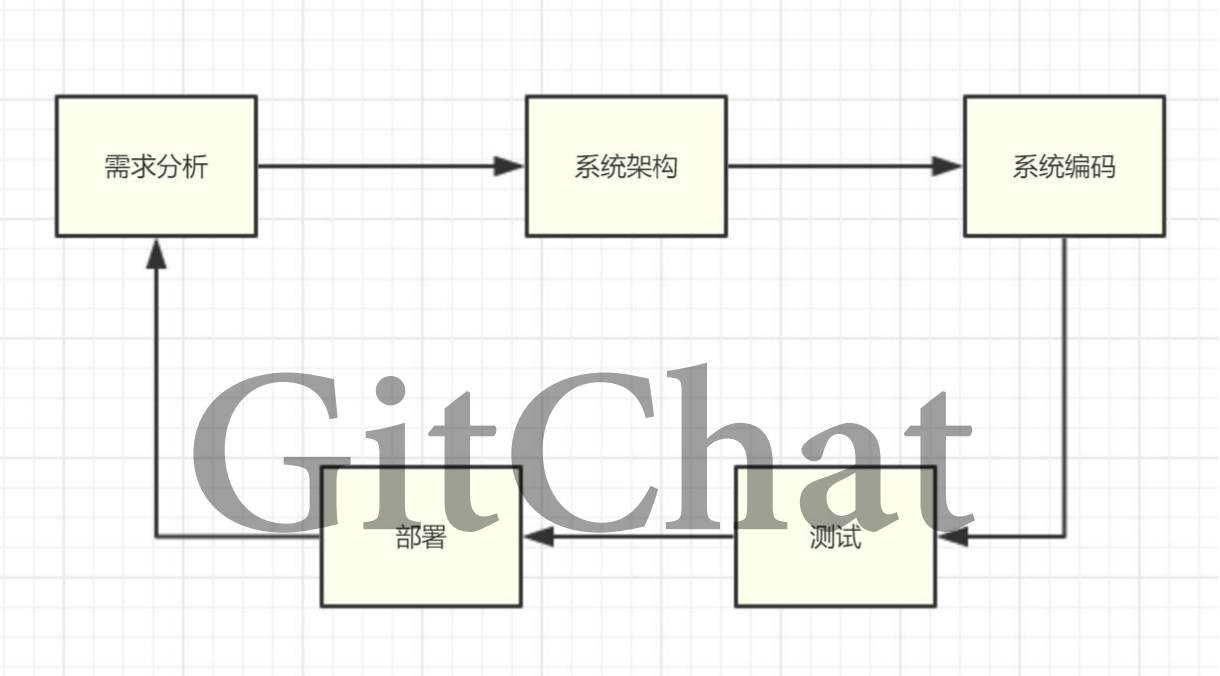
- 产品路线明确，且技术方案成熟的项目
- 比起开发效率更在乎质量的项目
- 研发能力较弱时，采用瀑布式开发流程更容易把控

缺陷：

- 项目初期，需求往往不明确，很难在初期就把需求敲定，影响后期环节
- 开发过程中需产生大量文档，增加工作量
- 由于开发模型是线性的，用户只有等到整个过程的末期才能见到开发成果，从而增加了开发的风险。

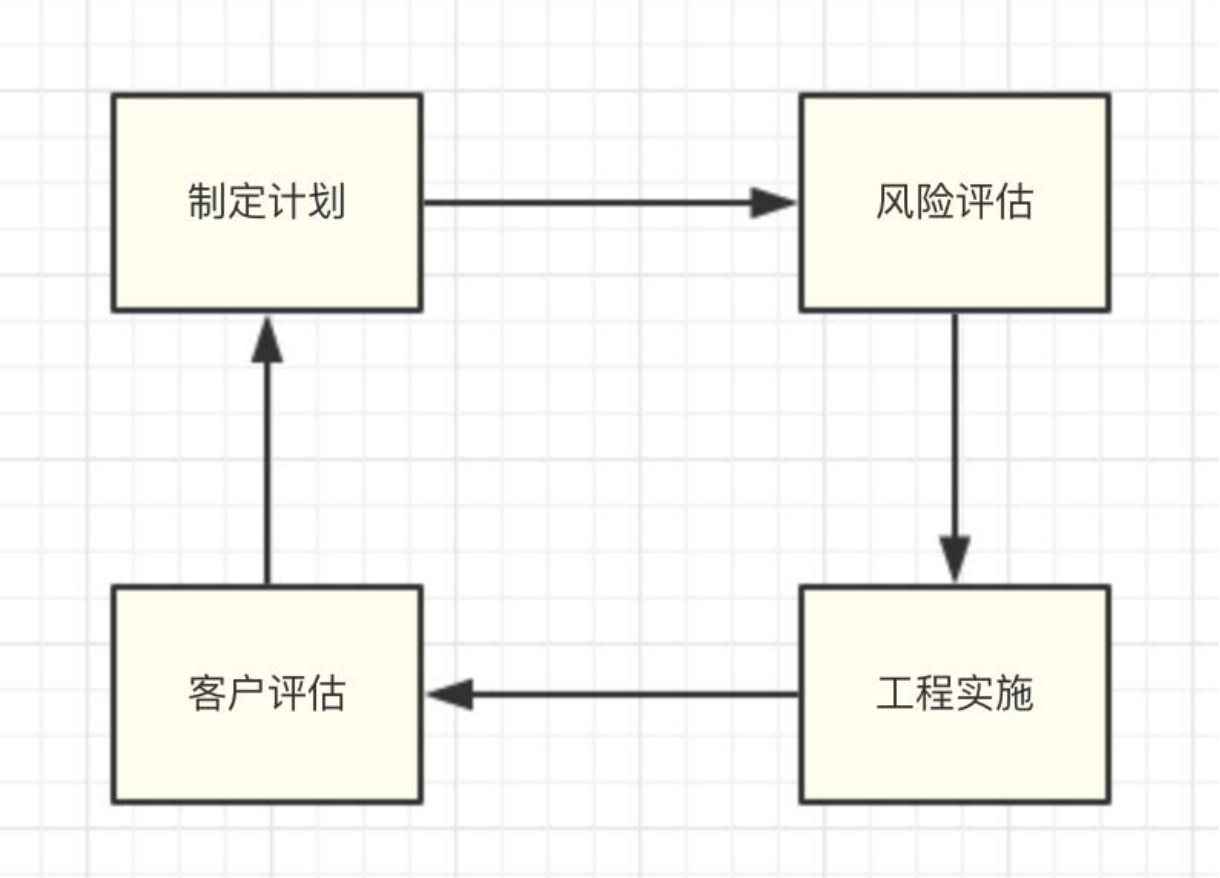
- 迭代模型

迭代模型则是通过将大的需求拆建成一个个周期进行开发，以减少风险。如下图，一个周期为一次迭代。



特点：

- 不要求一次性地开发出完整的软件系统，将软件开发视为一个逐步获取用广需求、完善软件产品的过程
- 降低风险，即使失误损失的也只是一次迭代周期的成本
- 开发人员清楚问题的焦点所在，效率更高
- 用户需求无需一开始就完全界定



特点：

- 设计上的灵活性,可以在项目的各个阶段进行变更设计上的灵活性
- 以小的分段来构建大型系统,使成本计算变得简单容易
- 客户始终参与每个阶段的开发,保证了项目不偏离正确方向以及项目的可控性。
- 随着项目推进,客户始终掌握项目的最新信息,从而他或她能够和管理层有效地交互。
- 客户认可这种公司内部的开发方式带来的良好的沟通和高质量的产品。

适用场景

- 大型的高风险项目
- 新近开发，需求不明确的项目

- 敏捷开发

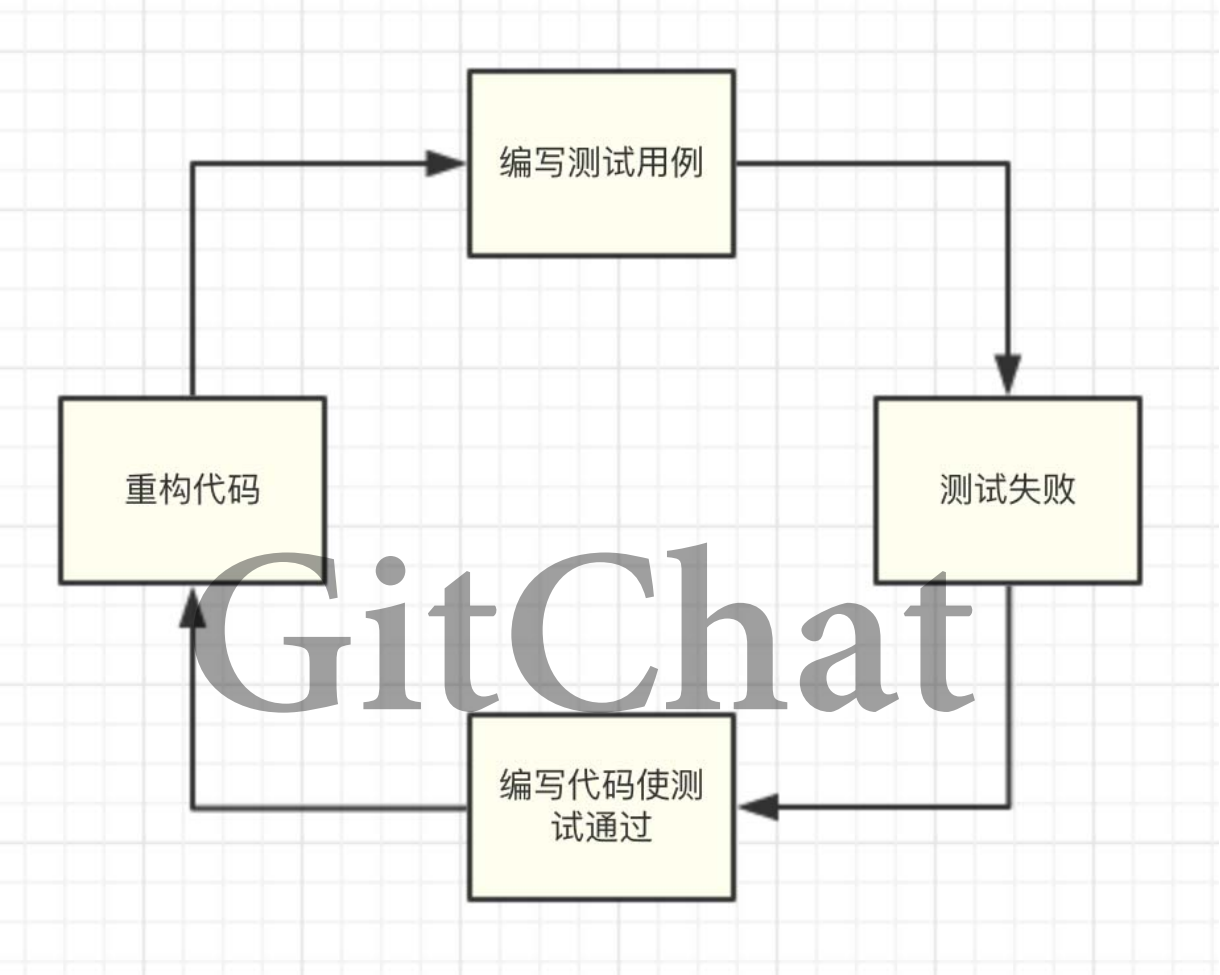
敏捷开发以用户的需求进化为核心，采用迭代、循序渐进的方法进行软件开发。在敏捷开发中，软件项目在构建初期被切分成多个子项目，各个子项目的成果都经过测试，具备可视、可集成和可运行使用的特征。换言之，就是把一个大项目分为多个相互联系，但也可独立运行的小项目，并分别完成，在此过程中软件一直处于可使用状

某某驱动开发,大家都听说住，这个章节为大家讲解最常见的驱动开发。

TDD

测试驱动开发(Test-driven development),是一种不同于传统软件开发流程的新型的开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。

- 步骤



1. 新增一个测试
2. 运行所有的测试（有时候只需要运行一个或一部分），发现新增的测试不能通过
3. 做一些小小的改动，尽快地让测试程序可运行，为此可以在程序中使用一些不合情理的方法

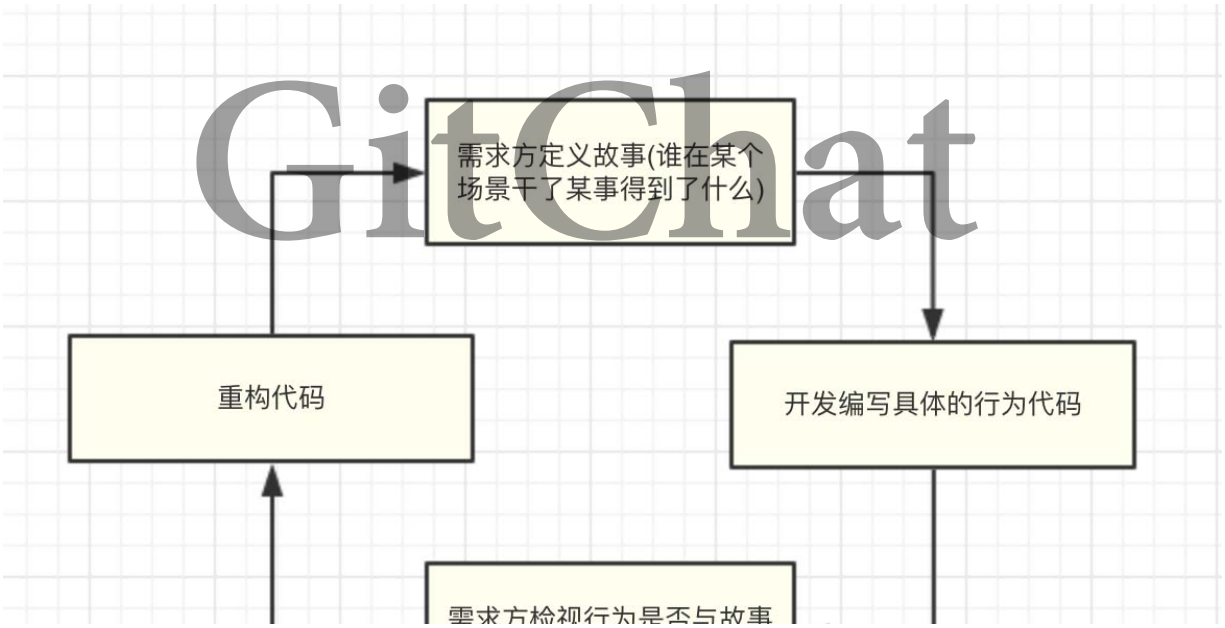
    - TDD提供了持续的回归测试，使我们拥有重构的勇气，因为代码的改动导致系统其他部分产生任何异常，测试都会立刻通知我们。完整的测试会帮助我们持续地跟踪整个系统的状态，因此我们就不需要担心会产生什么不可预知的副作用了。
    - TDD所产生的单元测试代码就是最完美的开发者文档，它们展示了所有的API该如何使用以及是如何运作的，而且它们与工作代码保持同步，永远是最新的。
    - TDD可以减轻压力、降低忧虑、提高我们对代码的信心、使我们拥有重构的勇气，这些都是快乐工作的重要前提。
    - 快速的提高了开发效率
- 相关的PHP测试框架
    - phpunit

BDD

行为驱动开发(Behavior Driven Development)是一种敏捷软件开发的技术，它鼓励软件项目中的开发者、QA和非技术人员或商业参与者之间的协作。BDD最初是由Dan North在2003年命名，它包括验收测试和客户测试驱动等的极限编程的实践，作为对测试驱动开发的回应。

与TDD不同的是,BDD并不是通过测试用例来规范约束开发者编写出质量更高、bug更少的代码，而是更加侧重行为设计,来解决需求与开发脱节的问题。

- 步骤



世界那么大，我想去看看

> 计算机科学的范畴非常的庞大，身为开发的我们有时候却不知道边界有多大。古人有云：不谋全局者，不足谋一域。计算机的世界那么大，你想看看吗？

计算机科学的主要领域（遵守ACM-2012计算分类系统）（By wiki）

大类	小类
电脑硬件	印刷电路板 外部设备 集成电路 超大规模集成电路 绿色计算 电子设计自动化
系统架构组织	计算机系统架构 嵌入式系统 实时计算
网络	网络传输协议 路由 网络拓扑 网络服务
软件组织	解释器 中间件 虚拟机 操作系统 软件质量
软件符号和工具	编程范型 编程语言 编译器 领域特定语言 软件框架 集成开发环境 软件配置管理 库
软件开发	软件开发过程 需求分析 软件设计 软件部署 软件维护 开源模式
计算理论	自动机 可计算性理论 计算复杂性理论 量子计算 数值计算方法 计算机逻辑 形式语义学
算法	算法分析 算法设计 随机化算法 计算几何
计算数学	离散数学 · 概率 · 统计学 · 数学软件 · 数理逻辑 · 集合论 · 数论 · 图论 · 类型论 · 范畴论 · 信息论 · 数值分析 · 数学分析
信息系统	数据库管理系统 电脑数据 企业信息系统 社会性软件 地理信息系统 决策支持系统 过程控制 数据挖掘 数字图书馆 系统平台 数字营销 万维网 信息检索
安全	密码学 形式化方法 入侵检测系统 网络安全 信息安全
人机交互	计算机辅助功能 · 用户界面 · 可穿戴计算机 · 普适计算 · 虚拟现实 · 聊天机器人
并发性	并发计算 并行计算 分布式计算 多线程 多元处理
人工智	自动推理 · 计算语言学 · 计算机视觉 · 进化计算 · 专家系统 · 自然语言处理 · 机器人学

小而美

致力于降低复杂度是软件开发的核心。人的大脑并没有办法去管理那么多的细节，但我们可以通过各自各样的工具与理念去处理这种复杂度。无论是体系构建还是业务编码，小而美一定是努力的方向。

自动化

高频低价值的工作是需要交给机器干的，我们需要做一些高杠杆，高附加值的工作。

创造世界

为什么每一门语言一开始都会运行 `hello world`？那是一个全新的世界在向你招手。想想你定义的变量，你定义的方法,你在以上帝的视角指定规则。就像”我的世界”一样，你在创造一个全新的世界。唯一不同的是”我的世界”是用的像素块，而你是用一种更高级更奇妙的方式去创造并享受其中。

感谢您的阅读，由于时间仓促，加之作者水平有限，有不足处，望广大读者积极指正。