2021技术随手记

2021年2月18日 14:59

1. **进程，线程及通信方式**

   https://www.php.cn/php-ask-453612.html 进程和线程有点 主从关系一样的，线程共享进程的资源

   进程间通信方式：1. 管道 2. 信号 3. 消息队列 Posix 消息队列 4. 共享内存 5. 套接口

2. **Php 查看扩展路径及查看进程和占用**

   php -i | grep -i extension_dir 或者 phpinfo 来看一下 或者 echo ini_get('extension_dir');

   ps -ef | grep "php-fpm" # 查看进程

   top | grep "php-fpm" # 查看占用

3. **Php 底层数组实现方式**

   linked list + hashtable 双向链表是中间映射表，用来存放索引和具体存储位置，然后具体存储位置对应的具体值在 hashtable 中

   PHP7数组实现的升级:

   PHP7通过链地址法来解决哈希冲突，只不过PHP5的链表是真实物理存在的链表，链表中bucket间的上下游是通过真实存在的指针来维护，而PHP7的链表其实是一种逻辑上的链表，所有的bucket都分配在一段连续的数组内存中，且不再通过指针维护上下游关系，每一个bucket只维护一个bucket在数组中的索引(由于内存是连续的，通过索引可以快速定位到bucket)，即可完成链表上的bucket遍历。

   扩容的过程为：

   如果已删除元素所占比例达到阈值，则会移除已被逻辑删除的 Bucket，然后将后面的 Bucket 向前补上空缺的 Bucket，因为 Bucket 的下标发生了变动，所以还需要更改每个元素在中间映射表中储存的实际下标值。

   如果未达到阈值，PHP 则会申请一个大小是原数组两倍的新数组，并将旧数组中的数据复制到新数组中，因为数组长度发生了改变，所以 key-value 的映射关系需要重新计算，这个步骤为重建索引。

4. **写时复制：**

   如果两个变量是相同的值，则共享同一块内存，而那块内存的 is_ref =1 refcount = 1 后者被引用一次 +1，为 0 的时候被销毁，相当于资源延迟分配。

   垃圾回收：

   不会立即回收，会放入缓冲区（一个双向链表），然后默认到了 10000 个开始回收，先将 refcount -1 ， =0 则进行回收

5. **解决内存溢出：**

   1、要增加PHP可用内存大小

   2、对数组进行分批处理，将用过的变量及时销毁；

3、尽可能减少静态变量的使用；

4、数据库操作完成后，要马上关闭连接。

5、可以使用 memory_get_usage () 函数，获取当前占用内存 根据当前使用的内存来调整程序

引申：

① unset () 函数只能在变量值占用内存空间超过 256 字节时才会释放内存空间

② 有当指向该变量的所有变量（如引用变量）都被销毁后，才会释放内存

③ unset 被引用的变量只会解除引用关系，不会销毁该变量

6. **Php7 新特性**

https://www.php.net/manual/zh/migration70.new-features.php 简要总结就是：

?? 运算符（NULL 合并运算符）

组合比较符 ó

函数返回值类型声明 : type

define 可以定义常量数组

标量类型声明

use 批量声明

匿名类，支持用 new class 来实例化一个匿名类，『用后即焚』

闭包（Closure）（匿名函数）增加了一个 call 方法

相同命名空间类一次性导入

Php7 底层优化：

① ZVAL 结构体优化，占用由24字节降低为16字节

② 内部类型 zend_string，结构体成员变量采用 char 数组，不是用 char*

③ PHP 数组实现由 hashtable 变为 zend array ，减少内存占用，同时 bucket 在连续内存空间

④ 函数调用机制，改进函数调用机制，通过优化参数传递环节，减少了一些指令

7. **Php 排序二维数组** array_multisort + array_column 就行

```php
<?php
$user_list = [
    ['name' => '张三', 'age' => 28],
    ['name' => '赵六', 'age' => 21],
    ['name' => '王五', 'age' => 20],
    ['name' => '李四', 'age' => 21]
];

array_multisort(array_column($user_list, 'age'), SORT_ASC, $user_list);

var_dump($user_list);
```

9. **缓存的应用场景：**

① 数据不需要强一致性

② 读多写少，并且读取得数据重复性较高

10. **Php 异步执行脚本：**

① popen 调用脚本，缺点是无法跨越，不能传参，会产生进程，高并发了会创建大量进程

② curl 方式，最小响应超时时间是 1s，也受限

③ fsockopen 打开一个网络连接或者 unix 套接字连接，原理和 http 一致，支持毫秒级超时处理

④ 引入 swoole

https://www.php.cn/php-weizijiaocheng-469392.html

11. **浅析OOP思想：**

面向对象程序设计（英语：Object-oriented programming，缩写：OOP）是种具有对象概念的编程典范，同时也是一种程序开发的抽象方针。它可能包含数据、属性、代码与方法。对象则指的是类（class）的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性，对象里的程序可以访问及经常修改对象相关连的数据。在面向对象程序编程里，计算机程序会被设计成彼此相关的对象。向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想。

对象的产生：① 以原型为基础 ② 以类为基础

12. **常见状态码**

一二三四五原则:（即一：消息系列；二：成功系列； 三：重定向系列；四：请求错误系列；五：服务器端错误系列。

301 状态码是永久移动

302 是临时移动

304 如果请求头中带有 If-None-Match 或 If-Modified-Since，则会到源服务器进行有效性校验，**如果源服务器资源没有变化，则会返回304，同时不返回内容；如果有变化，则返回200；**

500 代码 文件权限 资源有问题

501 请求方法服务器不支持

502 网关错误，例如得到了一个无效响应一类的就会出现这种错误

503 超载或者维护模式

504 网关超时，即在指定时间内没有正确的响应

499 客户端关闭请求，例如客户端请求 1s 内结束，php 未返回结果，或者找不到要请求的地址，会返回 499 错误

完整笔记 http系统状态码及nginx状态码

13. **Iptolong long2ip 注意转换成整形的时候负数问题**

```
function IP2Long($ip) {

    $ips = explode('.', $ip);

    if(count($ips) != 4) {

        return false;

    }

    return ($ips[0] << 24) + ($ips[1] << 16) + ($ips[2] << 8) +
```

```
$ips[3];
}


function Long2IP($int) {
    $ip1 = $ipint >> 24;
    $ip2 = ($ipint >> 16) & 255;
    $ip3 = ($ipint >> 8) & 255;
    $ip4 = $ipint & 255;
    return $ip1.'.'.$ip2.'.'.$ip3.'.'.$ip4;
}
```

14. **验证邮箱有效性：**
    filter_var($email, FILTER_VALIDATE_EMAIL)；验证IP也是有这样的情况
    /^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/ 验证邮箱
    ^[A-Za-z0-9\u4e00-\u9fa5]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$ 更完整的验证邮箱正则，例如 hxd.work@qq.com
    ^[A-Za-z0-9-_\u4e00-\u9fa5]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+$ 如果可以出现 -_ 就在正则中加入，例如 hxd-work@qq.com
15. **php链式调用：**
    ① 使用魔法函数__call结合call_user_func来实现
    ② 使用魔法函数__call结合call_user_func_array来实现
    ③ 不使用魔法函数__call来实现，修改 __call() 为 trim 重点在于，返回$this指针，方便调用后者函数。
16. **不使用第三个变量来交换两个变量的值**
    ① 两个为 数字时

```php
<?php
/**
 * 双方变量为数字时，可用交换方法五
 * 使用加减运算符，相当于数学运算了
*/
$a = 1; // a变量原始值
$b = 2; // b变量原始值
echo '交换之前 $a 的值: '.$a.', $b 的值: '.$b,'<br>'; // 输出原始值
$a=$a+$b; // $a $b和值
$b=$a-$b; // 不解释..
$a=$a-$b; // 不解释..
echo '交换之后 $a 的值: '.$a.', $b 的值: '.$b,'<br>'; // 输出结果值
```

    ② 两个为字符串时

```php
<?php
/**
 * 双方变量为字符串或者数字时，可用交换方法四
 * 使用异或运算
 */
$a = "This is A"; // a变量原始值
$b = "This is B"; // b变量原始值
echo '交换之前 $a 的值: '.$a.', $b 的值: '.$b,'<br>'; // 输出原始值

/**
 * 原始二进制:
 *
$a:01010100011010000110100101110011001000000110100101110011001000000100000
1
 *
$b:01010100011010000110100101110011001000000110100101110011001000000100001
0
 *
 * 下面主要使用按位异或交换，具体请参照下列给出的二进制过程，
 */

$a=$a^$b; // 此刻
$a:00000000000000000000000000000000000000000000000000000000000000000000000
011
$b=$b^$a; // 此刻
$b:01010100011010000110100101110011001000000110100101110011001000000100000
1
$a=$a^$b; // 此刻
$a:01010100011010000110100101110011001000000110100101110011001000000100001
0
echo '交换之后 $a 的值: '.$a.', $b 的值: '.$b,'<br>'; // 输出结果值
```

或者 str_replace 处理

```php
<?php
$a = "This is A"; // a变量原始值
$b = "This is B"; // b变量原始值
echo '交换之前 $a 的值: '.$a.', $b 的值: '.$b,'<br>'; // 输出原始值
$a .= $b; // 将$b的值追加到$a中
$b = str_replace($b, "", $a); // 在$a(原始$a+$b)中，将$b替换为空，则余下的返回值为$a
$a = str_replace($b, "", $a); // 此时，$b为原始$a值，则在$a(原始$a+$b)中将$b(原始$a)替换为空，则余下的返回值则为原始$b,交换成功
echo '交换之后 $a 的值: '.$a.', $b 的值: '.$b,'<br>'; // 输出结果值
```

17. **Strtoupper/strtolower 遇到中文会乱码**

    ① 需要手动分割字符串，然后 ord 函数判断是否是单词，是则大小写转换，中文则不处理

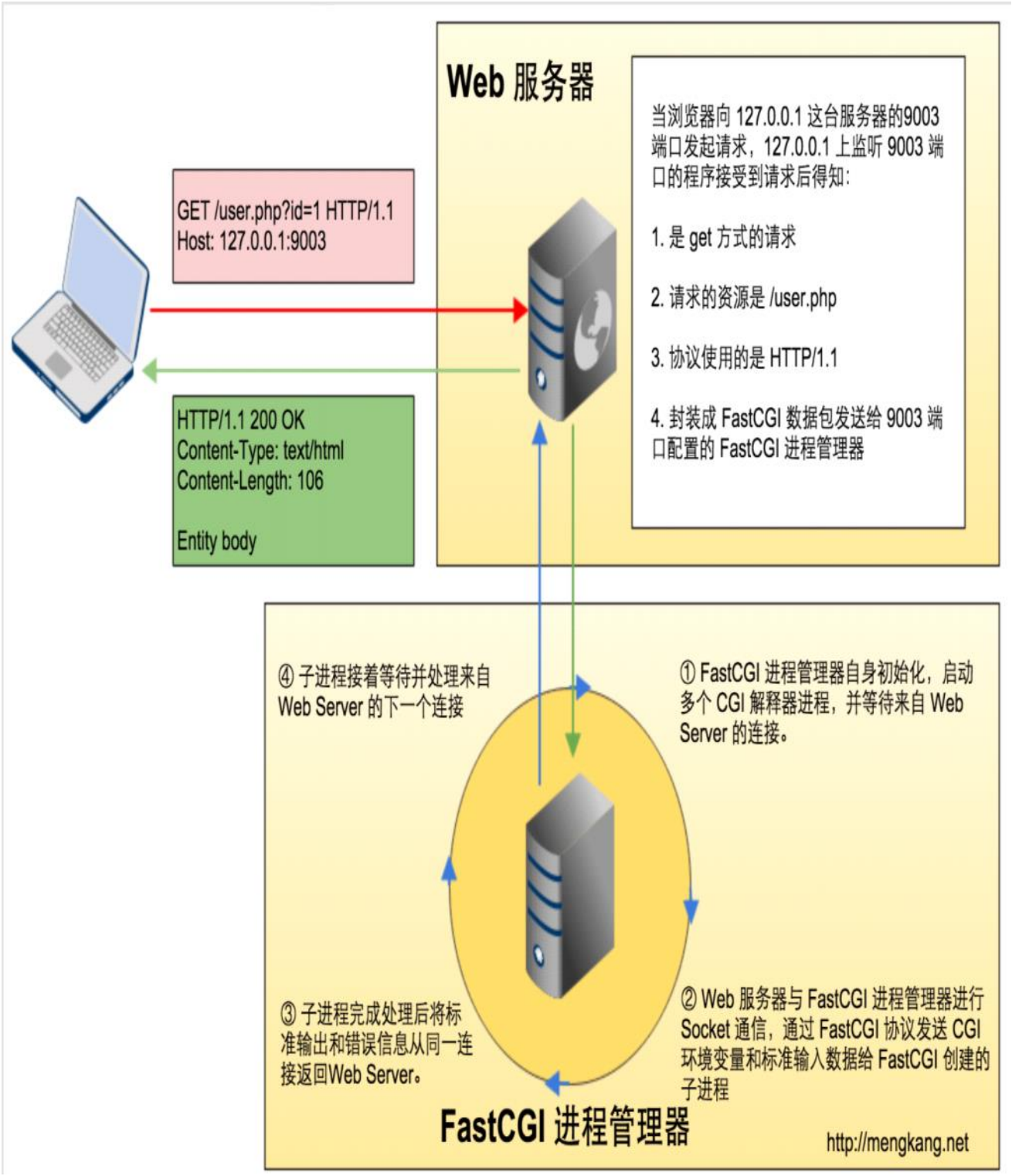
    ② mb_convert_case 函数中有可选参数，直接能处理这种情况

18. **Php-fpm 和 NGINX 通信机制**

CGI：是 Web Server 与 Web Application 之间数据交换的一种协议。
FastCGI：同 CGI，是一种通信协议，但比 CGI 在效率上做了一些优化。
PHP-CGI：是 PHP （Web Application）对 Web Server 提供的 CGI 协议的接口程序。
PHP-FPM：是 PHP（Web Application）对 Web Server 提供的 FastCGI 协议的接口程序，额外还提供了相对智能一些任务管理。
CGI就是规定要传哪些数据，以什么样的格式传递给后方处理这个请求的协议，例如URL、查询字符串、POST数据、HTTP header，缺点是每次请求都有启动和退出操作，不适合并发场景
Fastcgi 是常驻类型的，不需要每次去激活了
FastCGI程序会 先启动一个master，解析配置环境，初始化执行环境，然后再启动多个worker。当请求过来时，master会传递给一个worker，然后立即可以接受下一个请求。

首先要说的是：**fastcgi是一个协议，php-fpm实现了这个协议。**

大家都知道，PHP的解释器是php-cgi。php-cgi只是个CGI程序，他自己本身只能解析请求，返回结果，不会进程管理，所以就出现了一些能够调度php-cgi进程的程序，php-fpm就是这样的一个东西。它克服了php-cgi变更php.ini配置后，需重启php-cgi才能让新的php-ini生效，不可以平滑重启，直接杀死php-cgi进程，php就不能运行了的问题。修改php.ini之后，php-cgi进程的确没办法平滑重启的。php-fpm对此的处理机制是新的worker用新的配置，已经存在的worker处理完手上的活就可以歇着了，通过这种机制来平滑过度。



19. **Explain 后需要关注的信息：**

列名	备注
type	本次查询表联接类型，从这里可以看到本次查询大概的效率
key	最终选择的索引，如果没有索引的话，本次查询效率通常很差
key_len	本次查询用于结果过滤的索引实际长度，参见另一篇分享（FAQ系列-解读EXPLAIN执行计划中的key_len）
rows	预计需要扫描的记录数，预计需要扫描的记录数**越小越好**
Extra	额外附加信息，主要确认是否出现 **Using filesort、Using temporary** 这

	两种情况

首先看下 **type** 有几种结果，分别表示什么意思：

类型	备注
ALL	执行**full table scan**，这是**最差**的一种方式
index	执行**full index scan**，并且可以通过索引完成结果扫描并且直接从索引中取的想要的结果数据，也就是可以避免**回表**，比ALL略好，因为索引文件通常比全部数据要来的小
range	利用索引进行范围查询，比index略好
index_subquery	子查询中可以用到索引
unique_subquery	子查询中可以用到唯一索引，效率比 index_subquery 更高些
index_merge	可以利用**index merge**特性用到多个索引，提高查询效率
ref_or_null	表连接类型是ref，但进行扫描的索引列中可能包含NULL值
fulltext	全文检索
ref	基于索引的等值查询，或者表间等值连接
eq_ref	表连接时基于主键或非NULL的唯一索引完成扫描，比ref略好
const	基于主键或唯一索引唯一值查询，最多返回一条结果，比eq_ref略好
system	查询对象表只有一行数据，这是最好的情况

上面几种情况，从上到下一次是**最差到最好。**

再来看下Extra列中需要注意出现的几种情况：

关键字	备注
Using filesort	将用外部排序而不是按照索引顺序排列结果，数据较少时从内存排序，否则需要在磁盘完成排序，代价非常高，**需要添加合适的索引**
Using temporary	需要创建一个临时表来存储结果，这通常发生在对没有索引的列进行GROUP BY时，或者ORDER BY里的列不都在索引里，**需要添加合适的索引**
Using index	表示MySQL使用覆盖索引避免全表扫描，不需要再到表中进行二次查找数据，这是比较好的结果之一。注意不要和type中的index类型混淆
Using where	通常是进行了全表引扫描后再用WHERE子句完成结果过滤，**需要添加合适的索引**
Impossible WHERE	对Where子句判断的结果总是false而不能选择任何数据，例如where 1=0，无需过多关注

Select tables optimized away	使用某些聚合函数来访问存在索引的某个字段时，优化器会通过索引直接一次定位到所需要的数据行完成整个查询，例如MIN()\MAX()，这种也是比较好的结果之一

20. **Php-fpm 与 php 交互**

Php-fpm 运行的三种模式：

Static 模式最简单，直接启动配置的固定数量的进程，但是灵活性不够高

Ondemand 模式相对 static 模式比较复杂，会根据请求量的增加动态增加，但是处理完请求后不会立即释放，而是由定时事件定时的检测空闲到一定时间的进程才会释放

Dynamic 模式类似于 ondemand 模式，但进程的回收机制不同于 ondemand 模式，会根据 idle 数量进行增加和减少worker数量

Php-fpm 运行的逻辑：

Fpm 的实现就是创建一个 master 进程，在 master 进程中创建 worker pool 并监听 socket，然后 fork出多个子进程（work），这些 worker 在启动后阻塞在 fcgi_accept_request() 上，各自 accept 请求，**有请求到达后 worker 开始读取请求数据，读取完成后开始处理然后再返回，在这期间是不会接收其它请求的，也就是说 fpm 的子进程同时只能响应一个请求，只有把这个请求处理完成后才会 accept 下一个请求，所以有多少子进程就能同时处理多少请求。**

Fpm 工作流程：

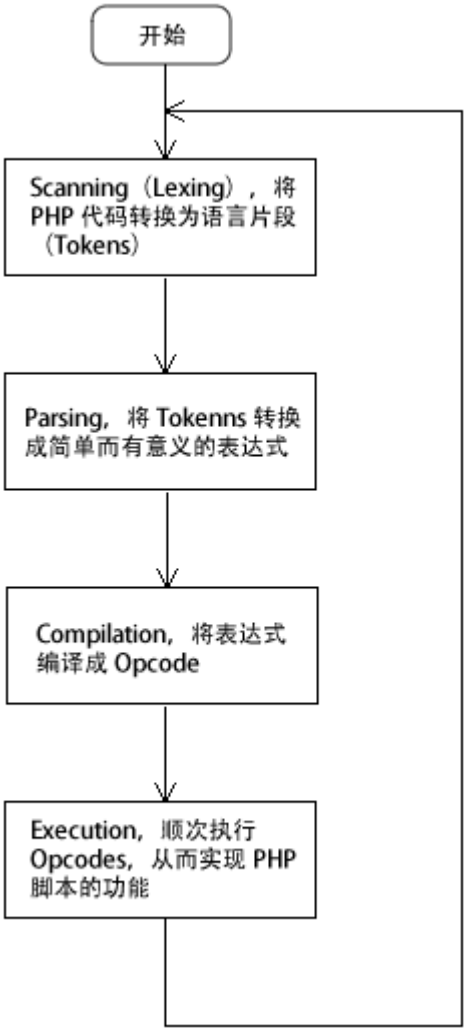
① FastCGI 进程管理器自身初始化，启动多个 CGI 解释器进程，并等待来自 Web Server 的连接。

② Web 服务器与 FastCGI 进程管理器进行 Socket 通信，通过 FastCGI 协议发送 CGI 环境变量和标准输入数据给 CGI 解释器进程。

③ CGI 解释器进程完成处理后将标准输出和错误信息从同一连接返回 Web Server。

④ CGI 解释器进程接着等待并处理来自 Web Server 的下一个连接。

Nginx 与 php-fpm 有两种通信方式：tcp socket 和 unix socket，unix 不需要经过网络协议栈，不需要打包拆包，计算校验和，维护序号和应答，只是将应用层数据从一个进程拷贝到另一个进程，减少不必要的 tcp 开销，高并发时性能不稳定，tcp 模式可以保证通信的正确性和完整性，效率可以通过负载均衡等优化。



一段PHP代码会经过词法解析、语法解析等阶段，会被翻译成一个个指令（opcode），然后 zend 虚拟机会顺序执行这些指令。PHP 本身是用C语言实现的，因此最终调用的也是C语言的函数，实际上我们可以把 PHP 看做一个C语言开发的软件。Opcode 是php执行的最基本单位

21. **数据库触发器 trigger**

触发器是一种特殊的存储过程，它被分配给某个特定的表，触发器都是自动调用的。当一特定的表数据被插入，更新或删除时，数据库需要执行一定的动作，触发器是确保数据完整性和一致性的基本有效的方法。

```
use 数据库名
create/alter trigger 触发器名
on 表名
for insert / delete /update
as
触发器要执行的操作
go

# enable/disable/drop trigger 触发器名
```

应用场景有：数据检查-例如周末禁止添加员工，安全性确认-例如年龄不能调低，数据备份

22. **数据库存储过程**

存储过程是一个预编译的SQL语句，执行效率高；存储过程代码放在数据库中，直接调

用，无需网络通信；安全性高，需要有一定权限的用户才行；可以重复使用
缺点是：物理迁移困难

23. **数据库连接池实现原理**
    连接池的作用就是为了提高性能，将已经创建好的连接保存在池中，当有请求来时，直接使用已经创建好的连接对 Server 端进行访问。这样 省略了创建连接和销毁连接的过程，从而提高性能。
24. **Redis 常见应用场景**
    首页热点新闻/商品，避免频繁读取数据库　bitmap 用来记录连续签到/登录情况　新闻阅读量的计数器

    ***Bitmap** 来做签到的方案：*
    Key 为 sign:userid:date 例如 sign:1000:202102 每个月一个 key，签到则将N天设置为 1，从0 开始，例如 bitset key 10 1，即202102月的第 10+ 1天签到了
    获取第一天签到使用 bitpos key 1 命令来获取。
    获取整月签到的次数 bitcount key 即是获取整月里 1 的数量，bitcount key [start] [end] 来进行范围查询。

    HyperLogLog 数据结构使用的场景为，面对海量的数据，如果要做精确度要求不高的去重统计，则推荐使用HyperLogLog，而不是set。因为HyperLogLog总共只占用12K的内存。而set会随着数据增长而线性增长。可以将 hhl 看做退化版的布隆过滤器

    布隆过滤器 比HHL多了一个 contains 方法，用于判断一个值是否已经存在。

    最新新闻列表 lpush 就行，然后读取　简单的消息发布系统 pubsub　sortedset 来做排行榜
25. **Linux 查看系统信息的基础命令**

```
系统
# uname -a        # 查看内核/操作系统/CPU信息
# head -n 1 /etc/issue  # 查看操作系统版本
# cat /proc/cpuinfo    # 查看CPU信息
# hostname        # 查看计算机名
# free -m        # 查看内存使用量和交换区使用量

资源
# df -h         # 查看各分区使用情况
# du -sh <目录名>     # 查看指定目录的大小
# grep MemTotal /proc/meminfo  # 查看内存总量
# grep MemFree /proc/meminfo  # 查看空闲内存量
# uptime         # 查看系统运行时间、用户数、负载
# cat /proc/loadavg    # 查看系统负载

网络
```

```
# ifconfig        # 查看所有网络接口的属性
# iptables -L     # 查看防火墙设置
# route -n        # 查看路由表
# netstat -lntp     # 查看所有监听端口
# netstat -antp     # 查看所有已经建立的连接
# netstat -s      # 查看网络统计信息

用户
# w           # 查看活动用户
# id <用户名>      # 查看指定用户信息
# last          # 查看用户登录日志
# cut -d: -f1 /etc/passwd  # 查看系统所有用户
# cut -d: -f1 /etc/group  # 查看系统所有组
# crontab -l      # 查看当前用户的计划任务

进程
# ps -ef         # 查看所有进程
# top          # 实时显示进程状态
```

其中查看文件信息的命令差别：

stat命令一般用于查看文件的状态信息。stat命令的输出信息比ls命令的输出信息要更详细。

wc命令一般用于统计文件的信息，比如文本的行数，文件所占的字节数。

du命令一般用于统计文件和目录所占用的空间大小。

ls 命令一般用于查看文件和目录的信息，包括文件和目录权限、拥有者、所对应的组、文件大小、修改时间、文件对应的路径等等信息。

df命令用于显示包含每个文件名参数的文件系统上可用的磁盘空间量，默认磁盘空间以1K块为最小单位。

netstat -ntlp | grep 端口号 查看端口被占用情况

lsof filename 查看文件被进程占用情况

26. **Find grep 命令区别**

    grep命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。

    Find 从指定的起始目录开始，递归地搜索其各个子目录，查找满足寻找条件的文件并对之采取相关的操作

    简单是：**grep是查找匹配条件的行，find是搜索匹配条件的文件**

27. **Awk**

    Awk 内置变量：

变量	用法
$0	当前记录（这个变量中存放着整个行的内容）

$1~$n	当前记录的第n个字段，字段间由FS分隔
FS	输入字段分隔符 默认是空格或\t
NF	当前记录中的字段个数，就是有多少列
NR	已经读出的记录数，就是行号，从1开始，如果有多个文件话，这个值也是不断累加中。
FNR	当前记录数，与NR不同的是，这个值会是各个文件自己的行号
RS	输入的记录分隔符， 默认为换行符
OFS	输出字段分隔符， 默认也是空格
ORS	输出的记录分隔符，默认为换行符
FILENAME	当前输入文件的名字

常用命令：

```
# 打印每一行的第二和第三个字段
awk '{print $2, $3}' file

# 统计文件的行数
awk ' END {print NR}' file

# 对 awk 处理的行做过滤
awk 'NR < 5' #行号小于5
awk 'NR==1,NR==4 {print}' file #行号等于1和4的打印出来
awk '/linux/' #包含linux文本的行（可以用正则表达式来指定，超级强大）
awk '!/linux/' #不包含linux文本的行

# 使用 -F 来设置定界符（默认为空格）
awk -F: '{print $NF}' /etc/passwd

# awk 实现head 命令
awk 'NR<=10{print}' filename

# 实现tail命令
awk '{buffer[NR%10] = $0;} END{for(i=0;i<11;i++){ \
print buffer[i %10]} } ' filename

# 查询访问最频繁的100个请求，主要是各种参数都包含了
grep -v ".php"  access.log | awk '{print $7}' | sort |uniq -c | sort -rn | head -n 100

# 查询访问 100 次以上的 ip
awk '{print $1}' access.log | sort -n |uniq -c |awk '{if($1 >100) print $0}'|sort -rn

# 查询指定 ip 访问最多的 100 个页面
grep '112.97.250.255' access.log |awk '{print $7}'|sort |uniq -c |sort -rn |head -n 100

# 查询最近 1000 条请求访问最多的地址
```

```
33 tail -1000 access.log |awk '{print $7}'|sort|uniq -c|sort -nr|less
34
35 # 按每秒统计请求数，显示top 100 的时间点 cut是截取 14-21 位，分钟为14-18 小时为 14-15
36 awk '{print $4}' access.log |cut -c 14-21|sort|uniq -c|sort -nr|head -n 100
```

28. **Which 和 whereis 区别**

    Which 是用来查找系统 PATH 目录下的可执行文件。说白了就是查找那些我们已经安装好的可以直接执行的命令，which 是基于 path 目录查找的。

    Whereis 这个命令可以用来查找二进制（命令）、源文件、man文件。Whereis 是基于索引数据库的，locate也是基于数据库的，find 是基于硬盘文件的

29. **负载均衡的几种实现方式及原理**

    ① ip负载均衡，相当于多一到N次重定向，过程

    ② DNS 负载均衡，DNS支持一个域名多个ip地址了

    ③ 反向代理负载均衡，NGINX 根据一定规则进行请求分发

    ④ F5硬件级别

    ⑥ CDN 对于静态文件的负载均衡

    负载均衡构建在原有网络结构之上，它提供了一种透明且廉价有效的方法扩展服务器和网络设备的带宽、加强网络数据处理能力、增加吞吐量、提高网络的可用性和灵活性。

30. **数据库主从复制的原理，会不会延迟，会该怎样解决**

    三个要点：网络延迟，master 负载 slave 负载 slave 对数据安全性的要求

    原理 ① master 将数据改变记录到 binlog 中 ② slave 启动一个io线程，从指定位置开始同步 binlog ③ 读取到 master 数据的更新，slave 写入到 replaylog 中，然后开始重放数据

    Tps 是事务数/秒 qps 是每秒查询率

    延迟原因：主库的 tps 并发较高时，产生的 ddl 超过 slave 的执行，或者网络延迟较大，也有可能是从库性能很差

    解决：减少网络延迟，关闭 slave 的 sync_binlog 设置成大点就行，累计多次事务之后刷盘 innodb_flush_log_at_trx_commit = 2 事务提交之后刷盘，slave 上也可以关闭这个，缺点是意外断电了会丢失数据

31. **如何保障数据的可用性，即使被删库了也能恢复到分钟级别。你会怎么做。**

    数据库集群方案就行，删掉主库了会自动选举从库，业务保持稳定，然后就是精细化的备份

32. **数据库链接过多的原因和解决方案**

    原因：① 配置的 max_connections 数量太少，修改配置或者 set global max_connections=xxx 就行 ② sleep 的链接回收太慢，修改 wait_timeout 就行，调小点加速回收 ③ 使用连接池

33. **502 504错误的原因**

    502 是无效响应，① nginx 无法与 php-fpm 进行连接，检查 php-fpm 是否启动 ② 脚本执行超时，然后 php-fpm 终止了执行和worker进程，也可能是高并发情况下，超过了最大子进程数量

max_execution_time request_terminate_timeout max_children 这三个配置相关
504 是 php 脚本的执行时间超过了 nginx 的等待时间，可能由 502 升级成为 504，和以下的 nginx 配置相关
fastcgi_connect_timeout 60;
fastcgi_read_timeout 300;
fastcgi_send_timeout 300;

34. **从输入 url 到页面展现经历了哪些**

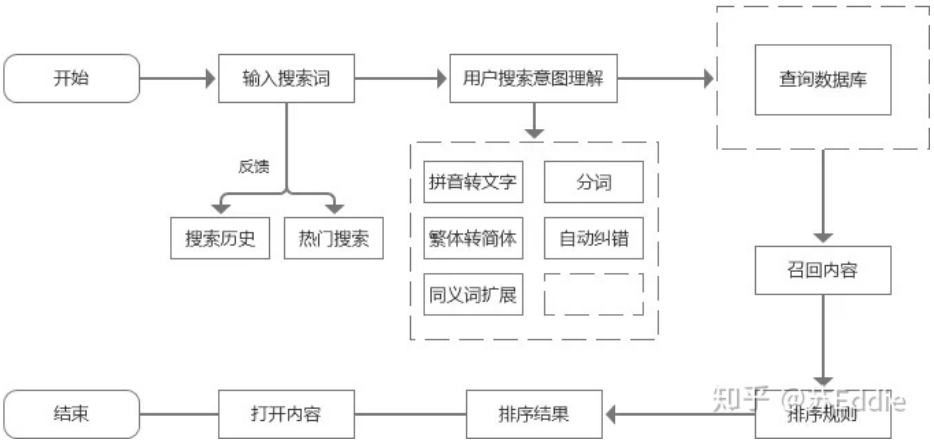
DNS 解析:将域名解析成 IP 地址
TCP 连接：TCP 三次握手
发送 HTTP 请求
服务器处理请求并返回 HTTP 报文
浏览器解析渲染页面
断开连接：TCP 四次挥手

35. **Redis 在具体业务中的应用**

考试部分，redis 存储某本书的数据，存储一小时，提升在课堂测验部分的查询性能
验证码收取部分，存储用完即销毁的验证码数据 存储当天 ip 手机号 的请求次数

36. **搜索解决方案**

① 简单的直接 like 搜索就行，例如数量较小的表，在后台管理中的操作，也可以是 mysql 的全文搜索。
② 电商系列可以使用elastic search，结合 jieba 分词这样的工具，分词查找，按相关性，热度排序，重点关注是怎样用户输入的词，然后再是搜索结果。
曾经有系统是有独立进程从数据库读取数据，保存到elastic search，会有少许延迟，同时每小时有脚本来校验数据同步的完整性。



37. **性能调优方案**

前端 - 后端 - 网络 分治解决。

① 基础的关注慢 sql，针对 sql 进行优化，没有高并发情况下，sql 一般是优化的入手点
② 再是elk监控方案，给每个请求一个唯一的 request_id，监控响应慢的接口，对接口可以打印sql，打印每个方法/调用的执行时间，找到慢的点，优化代码，使用缓存等手段，提升这段代码的执行。

③ 如果高并发的读，读频率搞的数据放到缓存中，当成热数据，提高查询响应
④ 网络层面就是负载均衡，一些高耗时的统计类任务 ，拆到额外的机器执行，不要影响正式业务
⑤ 让系统方便横向扩展，必要时加机器，加配置解决
⑥ 网络方面风控，拦截恶意流量，避免给业务代理多余的压力

https://coolshell.cn/articles/7490.html **耗子的文章**

38. **魔术方法**

_call()当调用不存在的方法时会自动调用的方法
__autoload()在实例化一个尚未被定义的类是会自动调用次方法来加载类文件
__set()当给未定义的变量赋值时会自动调用的方法
__get()当获取未定义变量的值时会自动调用的方法
__construct()构造方法，实例化类时自动调用的方法
__destroy()销毁对象时自动调用的方法
__unset()当对一个未定义变量调用unset()时自动调用的方法
__isset()当对一个未定义变量调用isset()方法时自动调用的方法
__clone()克隆一个对象
__tostring()当输出一个对象时自动调用的方法

39. **数据库 MVCC 是怎样的**

MVCC，Multi-Version Concurrency Control，多版本并发控制。MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问；在编程语言中实现事务内存。
MVCC 提供了时点（point in time）一致性视图。MVCC 并发控制下的读事务一般使用时间戳或者事务 ID去标记当前读的数据库的状态（版本），读取这个版本的数据。
读、写事务相互隔离，不需要加锁。读写并存的时候，写操作会根据目前数据库的状态，创建一个新版本，并发的读则依旧访问旧版本的数据。
MVCC 就是 同一份数据临时保留多版本的一种方式，进而实现并发控制。

MVCC 在不同的隔离级别下的差别：

在事务隔离级别为RC和RR级别下， InnnoDB存储引擎使用的才是多版本并发控制。然而，对于快照数据的定义却不相同。在RC事务隔离级别下，对于快照数据（undo端数据），总是读取被锁定行的最新的一份快照数据。而在RR事务隔离级别下，对于快照数据，多版本并发控制总是读取事务开始时的行数据。

40. **Php 数组解决hash冲突**

哈希表，顾名思义，即将不同的关键字映射到不同单元的一种数据结构。而将不同关键字映射到不同单元的方法就叫做哈希函数，冲突解决方案：

链接法

即当不同的关键字映射到同一单元时，在同一单元内使用链表来保存这些关键字。

开放寻址法

即当插入数据时，如果发现关键字被映射到的单元存在数据了，说明发生了冲突，就继续寻找下一个单元，直到找到可用单元为止。

而因为开放寻址法方案属于占用其他关键字映射单元的位置，所以后续的关键字更容易出现哈希冲突，因此容易出现性能下降。

php解决哈希冲突的方式是使用了链接法，所以php数组是由哈希表+双向链表实现

41. **Array_map 与 array_reduce array_walk array_fliter 区别**

    array_reduce($arr , callable $callback) 使用回调函数迭代地将数组简化为单一的值。

    array_map(callback $callback , $arr) 返回用户自定义函数作用后的数组。回调函数接受的参数数目应该和传递给 array_map() 函数的数组数目一致。此函数返回的是新数组，可以同时处理多个数组

    Array_walk 遍历处理，但不返回新数组，只改变现有的数组，walk 只可以处理一个数组，

    Array_filter 过滤掉输入数组中的元素，产生新数组

42. **Redis 分布式锁**

    特性

    - 互斥性： 同一时刻只能有一个线程持有锁
    - 可重入性： 同一节点上的同一个线程如果获取了锁之后能够再次获取锁
    - 锁超时：和J.U.C中的锁一样支持锁超时，防止死锁
    - 高性能和高可用： 加锁和解锁需要高效，同时也需要保证高可用，防止分布式锁失效
    - 具备阻塞和非阻塞性：能够及时从阻塞状态中被唤醒

    使用 set key value [EX seconds][PX milliseconds][NX|XX] 命令 (正确做法)

    Get key 获取锁， del key 删除锁

43. **Redis 事务**

命令	描述
MULTI	标记一个事务块的开始
EXEC	执行所有事务块内的命令
DISCARD	取消事务，放弃执行事务块内的所有命令
WATCH	监视一个（或多个）key，**如果在事务执行之前这个（或多个）key被其他命令所改动，那么事务将被打断**
UNWATCH	取消 WATCH 命令对所有 keys 的监视

    提交/放弃事务之后，会自动 unwatch，无需手动 unwatch

    Redis 不支持事务回滚机制，某个命令出现了错误，不会影响前后的命令执行。

44. **Laravel 注入原理**

    Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思

想。Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

DI—Dependency Injection，即“依赖注入”：组件之间依赖关系由容器在运行期决定，形象的说，即**由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。**通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。**一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。**

其实**IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。**

所有类需要提起在容器中登记，在运行需要的时候在提供，所有类的创建销毁都由容器控制。控制反转IoC(Inversion of Control)是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方

laravel 容器包含**控制反转和依赖注入（DI），使用起来就是，先把对象 bind 好，需要时可以直接使用 make 来取就好。这种由外部负责其依赖需求的行为，我们可以称其为 “控制反转（IoC）”**

依赖注入原理其实就是利用类方法反射，取得参数类型，然后利用容器构造好实例。然后再使用回调函数调起。注入对象构造函数不能有参数，否则会报错。

容器是个超级工厂模式，真正的 IoC 容器会根据类的依赖需求，自动在注册、绑定的一堆实例中搜寻符合的依赖需求，并自动注入到构造函数参数中去。自动搜寻依赖需求的功能，是通过反射（Reflection）实现的，恰好的，php 完美的支持反射机制

反射是什么：

面向对象编程中对象被赋予了自省的能力，而这个自省的过程就是反射。

反射，直观理解就是根据到达地找到出发地和来源。比如，一个光秃秃的对象，我们可以仅仅通过这个对象就能知道它所属的类、拥有哪些方法。

反射是指在PHP运行状态中，扩展分析PHP程序，导出或提出关于类、方法、属性、参数等的详细信息，包括注释。这种动态获取信息以及动态调用对象方法的功能称为反射API。

在平常开发中，用到反射的地方不多：一个是对对象进行调试，另一个是获取类的信息。在MVC和插件开发中，使用反射很常见，但是反射的消耗也很大，在可以找到替代方案的情况下，就不要滥用。

PHP有Token函数，可以通过这个机制实现一些反射功能。从简单灵活的角度讲，使用已经提供的反射API是可取的。

很多时候，善用反射能保持代码的优雅和简洁，但反射也会破坏类的封装性，因为反射可以使本不应该暴露的方法或属性被强制暴露了出来，这既是优点也是缺点。

```php
$student=new person();

$obj = new ReflectionClass('person');
$className = $obj->getName();
foreach($obj->getProperties() as $v)
{
    $Properties[$v->getName()] = $v;
}
foreach($obj->getMethods() as $v)
{
    $Methods[$v->getName()] = $v;
}
```

45. **三个小算法题：**

54张扑克牌，一半黑的，一半红的，随机抽两张，一黑一红的概率是多少。

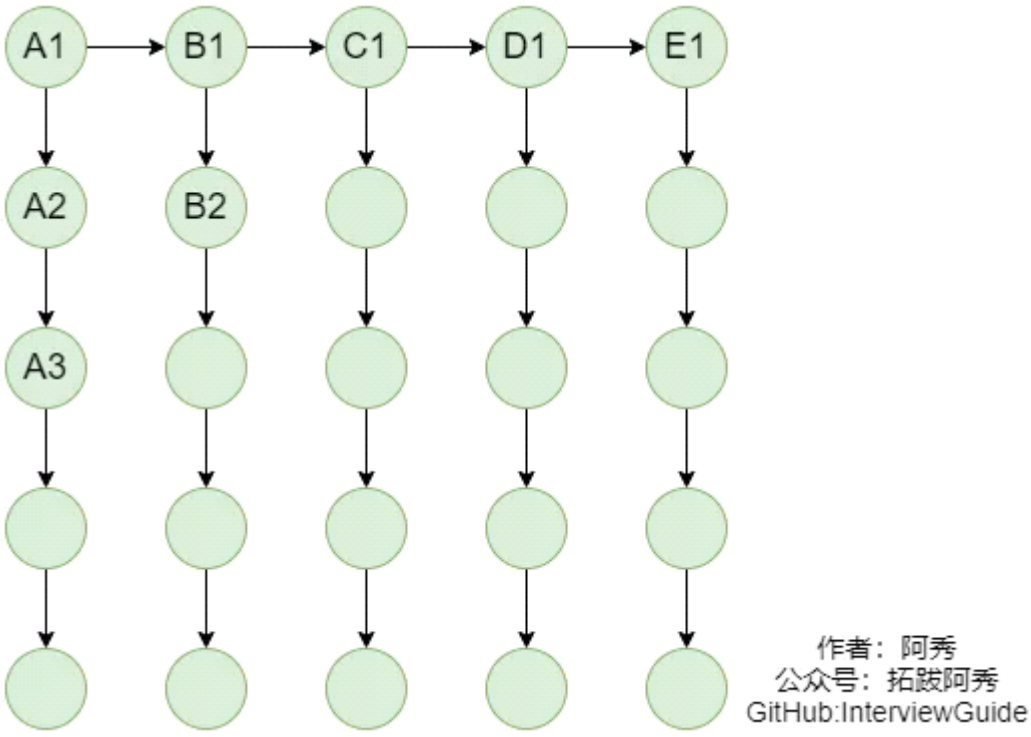
解析：第一张红 1/2 第二张黑 27/53，第一张黑第二张红概率一样，则1/2 * 27/53 * 2 = 27/53

还有个一堆小面包，第一个拿走一半外加半个，第二个人拿走剩下的一半外加半个，第三个人拿走剩下的一半加半个，正好拿完，问那堆原来有几个?

解析：第3人拿前有：0.5×2=1（个）第2人拿前有：（1＋0.5）×2=3（个）

第1人拿前有：（3＋0.5）×2=7（个）则原来有：7个

25匹马5条跑道找最快的3匹马，需要跑几次?



将25匹马分成ABCDE5组，假设每组的排名就是A1>A2>A3>A4>A5,用边相连，这里比赛5次

第6次，每组的第一名进行比赛，可以找出最快的马，这里假设A1>B1>C1>D1>E1

D1，E1肯定进不了前3，直接排除掉

第7次，B1 C1 A2 B2 A3比赛，可以找出第二，第三名
所以最少比赛需要7次

46. **Php 读取文件夹和创建文件夹**

```php
<?php

/**
* 遍历指定文件夹下的文件。
*
* @param string $directory
*
* @return array
*/
function dir_list($directory)
{
    static $array = [];

    $dir = dir($directory);
    while ($file = $dir->read()) {
        if (is_dir("$directory/$file") && $file !== '.' && $file !==
'..') {
            dir_list("$directory/$file");
        } else {
            if ($file !== '.' && $file !== '..') {
                $array[] = $file;
            }
        }
    }

    return $array;
}
```

```php
<?php

/**
* 遍历指定文件夹下的文件。
*
* @param string $directory
*
* @return array
*/
function dir_list(string $directory): array
{
    static $array = [];

    $dir = scandir($directory);
    foreach ($dir as $file) {
        if (is_dir("$directory/$file") && $file !== '.' && $file !==
'..') {
            dir_list("$directory/$file");
        } else {
            if ($file !== '.' && $file !== '..') {
                $array[] = $file;
            }
        }
    }
```

```
    return $array;
}
```

创建文件夹

```
// $dir = 'A/B/C/D/E'
function Directory($dir) {
    return is_dir ($dir) || (Directory(dirname($dir)) && mkdir($dir,
0777));
}

// mkdir 函数自身支持递归创建
mkdir(string $pathname,int $mode =
0777,bool $recursive = false,resource $context = ?):bool
```

47. **PHP 生存周期**

SAPI运行PHP都经过下面几个阶段:

1、模块初始化阶段（module init）：

这个阶段主要进行php框架、zend引擎的初始化操作。这个阶段一般是在SAPI启动时执行一次，对于FPM而言，就是在fpm的master进行启动时执行的。php加载每个扩展的代码并调用其模块初始化例程（MINIT），进行一些模块所需变量的申请,内存分配等。

2、请求初始化阶段（request init）：

当一个页面请求发生时，在请求处理前都会经历的一个阶段。对于fpm而言，是在worker进程accept一个请求并读取、解析完请求数据后的一个阶段。在这个阶段内，SAPI层将控制权交给PHP层，PHP初始化本次请求执行脚本所需的环境变量。

3、php脚本执行阶段

php代码解析执行的过程。Zend引擎接管控制权，将php脚本代码编译成opcodes并顺次执行

4、请求结束阶段（request shutdown）：

请求处理完后就进入了结束阶段，PHP就会启动清理程序。这个阶段，将flush输出内容、发送http响应内容等，然后它会按顺序调用各个模块的RSHUTDOWN方法。RSHUTDOWN用以清除程序运行时产生的符号表，也就是对每个变量调用unset函数。

5、模块关闭阶段（module shutdown）：

该阶段在SAPI关闭时执行，与模块初始化阶段对应，这个阶段主要是进行资源的清理、php各模块的关闭操作，同时，将回调各扩展的module shutdown钩子函数。这是发生在所有请求都已经结束之后，例如关闭fpm的操作。（这个是对于CGI和CLI等SAPI，没有“下一个请求”，所以SAPI立刻开始关闭。）

48. **MYSQL 几个count的区别**

COUNT(常量) 和 COUNT(*) 表示的是直接查询符合条件的数据库表的行数。

而COUNT(列名)表示的是查询符合条件的列的值不为NULL的行数。

Count(ID) 是遍历整张表的主键索引

49. **Redis 哨兵机制**

1）Sentinel(哨兵) 进程是用于监控 Redis 集群中 Master 主服务器工作的状态

2）在 Master 主服务器发生故障的时候，可以实现 Master 和 Slave 服务器的切换，保

证系统的高可用（High Availability）

监控(Monitoring)：

1）哨兵(sentinel) 会不断地检查你的 Master 和 Slave 是否运作正常。

2）提醒(Notification)：当被监控的某个Redis节点出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。（使用较少）

3）自动故障迁移(Automatic failover)：当一个 Master 不能正常工作时，哨兵(sentinel) 会开始一次自动故障迁移操作。具体操作如下：

（1）它会将失效 Master 的其中一个 Slave 升级为新的 Master, 并让失效 Master 的其他Slave 改为复制新的 Master。

（2）当客户端试图连接失效的 Master 时，集群也会向客户端返回新 Master 的地址，使得集群可以使用现在的 Master 替换失效 Master。

（3）Master 和 Slave 服务器切换后，Master 的 redis.conf、Slave 的 redis.conf 和 sentinel.conf 的配置文件的内容都会发生相应的改变，即 Master 主服务器的 redis.conf 配置文件中会多一行 slaveof 的配置，sentinel.conf 的监控目标会随之调换。

50. **B+ 树的优点**

1、B+树的**层级更少**：相较于B树B+每个非叶子节点存储的关键字数更多，树的层级更少所以查询数据更快；

2、B+树**查询速度更稳定**：B+所有关键字数据地址都存在叶子节点上，所以每次查找的次数都相同所以查询速度要比B树更稳定;

3、B+树**天然具备排序功能**：B+树所有的叶子节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比B树高。

4、B+树**全节点遍历更快**：B+树遍历整棵树只需要遍历所有的叶子节点即可，，而不需要像B树一样需要对每一层进行遍历，这有利于数据库做全表扫描。

B树相对于B+树的优点是，如果经常访问的数据离根节点很近，而B树的非叶子节点本身存有关键字其数据的地址，所以这种数据检索的时候会要比 B+ 树快。

51. **各种区别**

Cookie 和 session 区别

- 作用范围不同，Cookie 保存在客户端（浏览器），Session 保存在服务器端。
- 存取方式的不同，Cookie 只能保存 ASCII，Session 可以存任意数据类型，一般情况下我们可以在 Session 中保持一些常用变量信息，比如说 UserId 等。
- 有效期不同，Cookie 可设置为长时间保持，比如我们经常使用的默认登录功能，Session 一般失效时间较短，客户端关闭或者 Session 超时都会失效。
- 隐私策略不同，Cookie 存储在客户端，比较容易遭到不法获取，早期有人将用户的登录名和密码存储在 Cookie 中导致信息被窃取；Session 存储在服务端，安全性相对 Cookie 要好一些。
- 存储大小不同， 单个 Cookie 保存的数据不能超过 4K，Session 可存储数据远高于 Cookie。

Cookie 被禁用之后：

第一种方案，每次请求中都携带一个 SessionID 的参数，也可以 Post 的方式提交，也可以在请求的地址后面拼接 xxx?SessionID=123456…。

第二种方案，Token 机制。Token 机制多用于 App 客户端和服务器交互的模式，也可以用于 Web 端做用户状态管理。

MYISAM 和 Innodb 差别

1. **InnoDB 支持事务**，MyISAM 不支持事务。这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一；
2. **InnoDB 支持外键**，而 MyISAM 不支持。对一个包含外键的 InnoDB 表转为 MYISAM 会失败；
3. **InnoDB 是聚集索引，MyISAM 是非聚集索引。**聚簇索引的文件存放在主键索引的叶子节点上，因此 InnoDB 必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而 MyISAM 是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
4. **InnoDB 不保存表的具体行数**，执行 select count(*) from table 时需要全表扫描。而MyISAM 用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；
5. **InnoDB 最小的锁粒度是行锁，MyISAM 最小的锁粒度是表锁。**一个更新语句会锁住整张表，导致其他查询和更新都会被阻塞，因此并发访问受限。这也是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一；
6. 每个 MyISAM 在磁盘上存储成三个文件。分别为：**表定义文件、数据文件、索引文件。**InnoDB 是单文件，文件大小受操作系统的限制
7. **MyISAM：不支持 fulltext 索引，innodb 支持**
8. MyISAM 支持支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。InnoDB 需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。
9. 可移植性，MyISAM 是存储在文件中，迁移很方便，InnoDB 有各种日志，免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

Tcp 和 udp 的区别：

1. **tcp是面向连接的，而udp是无连接就发送数据的**，tcp 的三次握手，四次挥手，所以 tcp 面向链接，而udp就是一个数据报带着目的ip和端口，直接发送。
2. **tcp传输时是可靠的，udp传输时是不可靠的。**tcp有个以字节为单位的滑动窗口，它把要发送的数据都以字节形式存储在这个滑动窗口当中。每次发送完窗口的数据后，都会先保留数据，只有当收到对方的数据确认收到信号时再清除这些数据，如果超时没有

收到确认信号的话就要重传。这就是tcp的确认重传机制。

3. **udp 适合快速传输不太重要但实时性要求比较高的大数据**，比如实时的视频传输，而 tcp 适合传输可靠的内容。

4. **基于tcp的应用层协议：http,https,ftp,telnet 基于udp ： dns,tftp**

5. **一个tcp数据包报头的大小是20字节，udp数据报报头是8个字节。**TCP报头中包含序列号，ACK号，数据偏移量，保留，控制位，窗口，紧急指针，可选项，填充项，校验位，源端口和目的端口。而UDP报头只包含长度，源端口号，目的端口，和校验和。

6. **TCP有流量控制。**在任何用户数据可以被发送之前，TCP需要三数据包来设置一个套接字连接。TCP处理的可靠性和拥塞控制。另一方面，UDP不能进行流量控制。

	UDP	TCP
是否连接	无连接	面向连接
是否可靠	不可靠传输，不使用流量控制和拥塞控制	可靠传输，使用流量控制和拥塞控制
连接对象个数	支持一对一，一对多，多对一和多对多交互通信	只能是一对一通信
传输方式	面向报文，udp 直接发送，如果报文过长，udp只会记录位置，不合并不拆分，直接发送	面向字节流，tcp有缓冲区，如果发送的报文太长，会被切分短点再发
流量控制	无	有，滑动窗口控制
首部开销	首部开销小，仅8字节	首部最小20字节，最大60字节
适用场景	适用于实时应用（IP电话、视频会议、直播等）	适用于要求可靠传输的应用，例如文件传输

Http 和 https 的区别：

1、https协议需要到ca申请证书，一般免费证书很少，需要交费。

2、http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

Include 和 require 区别

这两者是语言结构，不是函数，他们都可以直接引用参数，而不是括号内引用参数

include在用时加载，一般放在代码段中，出错时继续执行下面的代码

require一般放在脚本最前面，会一开始就读取，出错时停止运行代码

_once 是已加载的不加载

Epoll select poll 区别

(1)select==>时间复杂度O(n)
它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。**所以select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。最多支持 1024 个fd。
(2)poll==>时间复杂度O(n)
poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，**但是它没有最大连接数的限制，原因是它是基于链表来存储的.** poll的实现和select非常相似，只是描述fd集合的方式不同，poll 使用 pollfd 结构而不是 select 的 fd_set 结构，其他的都差不多。
(3)epoll==>时间复杂度O(1)
epoll可以理解为event poll，不同于忙轮询和无差别轮询，**epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）**
select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。**

总结：

（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

Linux 系统中，把一切都看做是文件，当进程打开现有文件或创建新文件时，内核向进程返回一个文件描述符，**文件描述符就是内核为了高效管理已被打开的文件所创建的索引，用来指向被打开的文件，所有执行I/O操作的系统调用都会通过文件描述符。**

New self 和 new static 的区别：

- new self()和new static()的区别只有在继承中才能体现出来，如果没有任何继承，那么这两者是没有区别的。

- 在继承中，new self()返回的实例是万年不变的，无论谁去调用，都返回同一个类的实例，而new static()则是由调用者决定的。

Git reset 和 git revert 区别

git reset 会失去后面的提交，而 git revert 是通过反做的方式重新创建一个新的提交，而保留原有的提交，git reset 之后需要 git push -f 强制提交，不建议使用

Do while while foreach for 区别

Do while 和while类似，do while 会不管条件真假先执行一次，while 条件为真才执行，foreach 循环为先读取整块数据，然后再循环，而 for 主要用于限制循环次数例如循环数组，while 是移动内部指针，foreach 是对数组副本进行操作，而 foreach 在读操作比较快，在写操作比较慢，因为 php 的 **引用计数写时复制** 的特性

Mysql 事务中脏读和幻读的区别：

脏读（Dirty Read）： 脏读是指一个事务读到了另一个未提交事务修改过的数据。

幻读（Phantom Read）： 所谓幻读，指的是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务**再次读取该范围的记录时**，会读取到之前没有读到的数据。

事务的四个隔离级别：Read Uncommitted（读取未提交内容），Read Committed（读取提交内容），Repeatable Read（可重读），Serializable（可串行化），**其中未提交读会产生脏读，未提交读、提交读、可重复读会产生幻读情况**

Isset empty gettype is_null 区别

Expression	gettype()	empty()	is_null()	isset()	bool : if($x)
$x = "";	string	true	false	true	false
$x = null;	NULL	true	true	false	false
var $x;	NULL	true	true	false	false
$x is undefined	NULL	true	true	false	false
$x = array();	array	true	false	true	false
$x = array('a', 'b');	array	false	false	true	true
$x = false;	bool	true	false	true	false
$x = true;	bool	false	false	true	true
$x = 1;	int	false	false	true	true
$x = 42;	int	false	false	true	true
$x = 0;	int	true	false	true	false

$x = -1;	int	false	false	true	true
$x = "1";	string	false	false	true	true
$x = "0";	string	true	false	true	false
$x = "-1";	string	false	false	true	true
$x = "php";	string	false	false	true	true
$x = "true";	string	false	false	true	true
$x = "false";	string	false	false	true	true

isset 检测一个变量是否有值，包含 false 0 空字符串，不可以为 null
Empty 检查是否有空值 空字符串 0 null false，有则返回 true 非空或者非零值则返回 false

52. **Redis 主从同步原理**

Slave 初始化中是全量同步，

- 从服务器连接主服务器，发送SYNC命令；
- 主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；
- 主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；
- 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；
- 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；
- 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；

全量之后是增量同步：指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。

额外信息：

1）一个master可以有多个slave，slave也可以有多个slave，组成树状结构
2）主从同步不会阻塞master，但是会阻塞slave。也就是说当一个或多个slave与master进行初次同步数据时，master可以继续处理client发来的请求。相反slave在初次同步数据时则会阻塞不能处理client的请求；
3）主从同步可以用来提高系统的可伸缩性，我们可以用多个slave专门处理client的读请求，也可以用来做简单的数据冗余或者只在slave上进行持久化从而提升集群的整体性能。

Redis Sentinal 着眼于高可用，在 master 宕机时会自动将 slave 提升为 master，继续提供服务。
Redis Cluster 着眼于扩展性，在单个 redis 内存不足时，使用 Cluster 进行分片存储。

53. **redo undo 日志和事务执行中崩溃的重放**

redo log 包括两部分：一个是内存中的日志缓冲(redo log buffer)，另一个是磁盘上的日志文件(redo logfile)。

mysql 每执行一条 DML（增删改） 语句，先将记录写入 redo log buffer，后续某个时间点再一次性将多个操作记录写到 redo log file。这种 **先写日志，再写磁盘** 的技术就是 MySQL wal(预写式日志)，在计算机操作系统中，用户空间(user space)下的缓冲区数据一般情况下是无法直接写入磁盘的，中间必须经过操作系统内核空间(kernel space)缓冲区(OS Buffer)。

因此， redo log buffer 写入 redo logfile 实际上是先写入 OS Buffer ，然后再通过系统调用 fsync() 将其刷到 redo log file中。

undo-log 保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读。**在事务开始之前**，将当前事务的版本生成undo-log，undo也会产生redo日志来保证undo-log的可靠性。

undo log和redo log记录物理日志不一样，它是逻辑日志。**可以认为当delete一条记录时，undo log中会记录一条对应的insert记录，反之亦然，当update一条记录时，它记录一条对应相反的update记录。**

在恢复时，对于已经COMMIT的事务使用redo log进行重做，对于没有COMMIT的事务，使用undo log进行回滚。

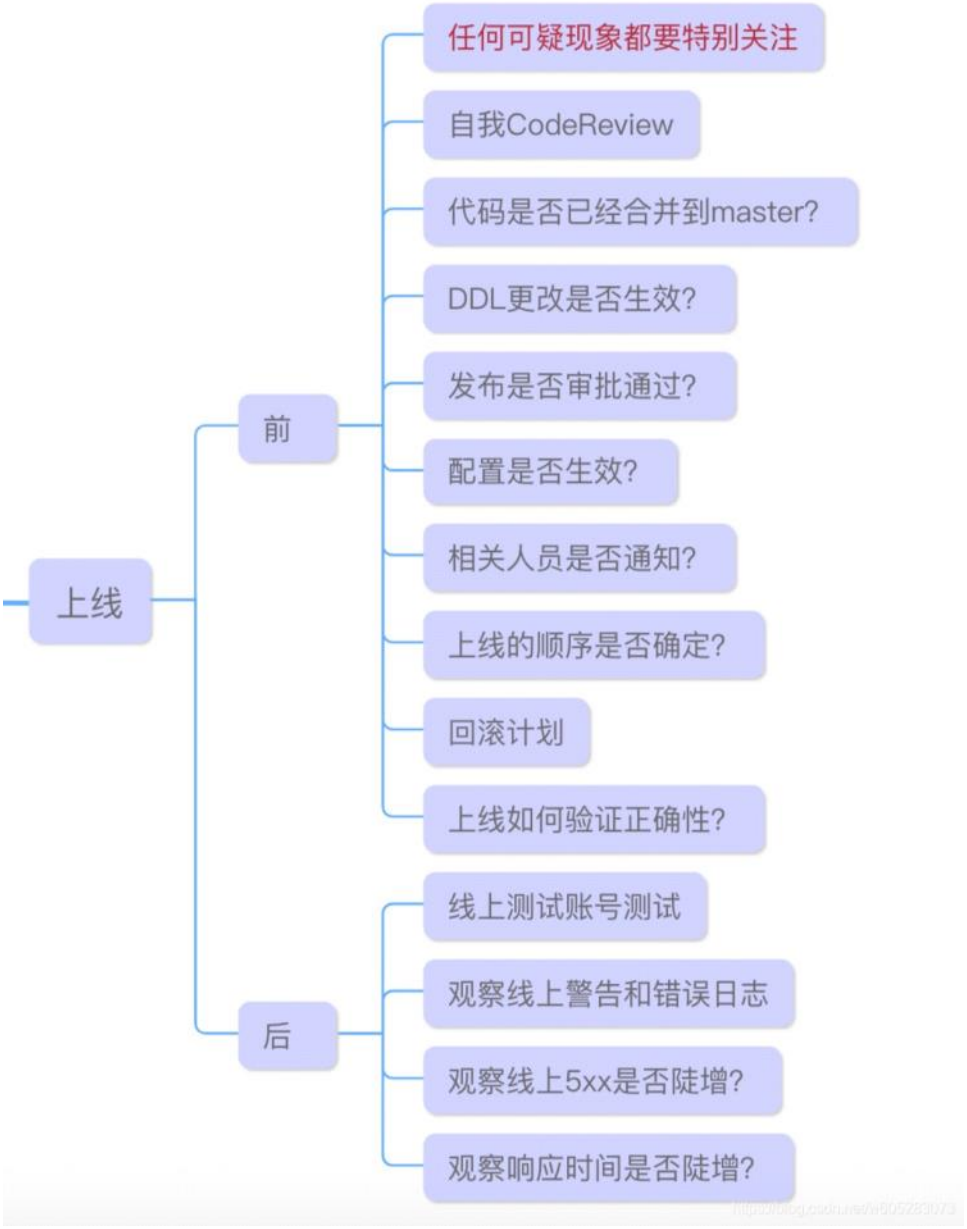
54. **Php 长网址转成短网址**

```php
<?php
//短网址生成算法
class ShortUrl {

  //字符表
  public static $charset =
```

```
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

  public static function encode($url)
  {
    $key = 'abc'; //加盐
    $urlhash = md5($key . $url);
    $len = strlen($urlhash);

    //将加密后的串分成4段，每段4字节，对每段进行计算，一共可以生成四组短连接
    for ($i = 0; $i < 4; $i++) {
      $urlhash_piece = substr($urlhash, $i * $len / 4, $len / 4);

      //将分段的位与0x3fffffff做位与，0x3fffffff表示二进制数的30个1，即30位以后的加密串都归零
      //此处需要用到hexdec()将16进制字符串转为10进制数值型，否则运算会不正常
      $hex = hexdec($urlhash_piece) & 0x3fffffff;

      //域名根据需求填写
      $short_url = "http://t.cn/";

      //生成6位短网址
      for ($j = 0; $j < 6; $j++) {

        //将得到的值与0x0000003d,3d为61，即charset的坐标最大值
        $short_url .= self::$charset[$hex & 0x0000003d];

        //循环完以后将hex右移5位
        $hex = $hex >> 5;
      }

      $short_url_list[] = $short_url;
    }

    return $short_url_list;
  }
}

$url = "http://www.sunbloger.com/";
$short = ShortUrl::encode($url);
print_r($short);
```

更好的长短网址转换方案是**长网址插入数据库，得到id，十进制的 id 转成 62 进制的值，保存起来对应关系，可逆可查**

55. **代码发布流程**



在一台机器发布，其他的 rsync 同步代码

56. **常见设计模式**

单例模式解决的是如何在整个项目中创建唯一对象实例的问题，避免重复创建(实例化)对象，已经有现成的实例就用现成的。减少资源的浪费(因为创建多个实例，浪费内存，完全没必要)，单件模式保证了每时每刻引用的都是同一个实例。最常用的地方是数据库连接。

- 必须拥有一个访问级别为private的构造函数，用于阻止类被随意实例化
- 必须拥有一个保存类的实例的静态变量
- 必须拥有一个访问这个实例的公共静态方法，该方法通常被命名为getInstance()
- 必须拥有一个私有的空的clone方法，防止实例被克隆复制

```
class Single{
    public static $_instance;   // 私有化静态属性
    private function __construct()   //私有化构造方法
```

```
    {
    }

    private function __clone()    // 私有化克隆方法，禁止克隆
    {
    }

    public static function getInstance()  //公有化静态方法
    {
        if(!self::$_instance) {
            self::$_instance = new self();
        }
        return self::$_instance;
    }

    public function sayHi()
    {
        echo "Hi \n";
    }
}

$single= Single::getInstance();
$single->sayHi();
```

工厂模式 是一种类，它具有为您创建对象的某些方法。工厂模式解决的是如何不通过new建立实例对象的方法，您可以使用工厂类创建对象，而不直接使用 new。这样，如果您想要更改所创建的对象类型，只需更改该工厂即可。使用该工厂的所有代码会自动更改。

```
/**
 * 抽象出一个人的接口
 * Interface Person
 */
interface Person{
    public function showInfo();
}

/**
 * 一个继承于抽象人接口的学生类
 * Class Student
```

```
 */
class Student implements Person{
    public function showInfo()
    {
        echo "这是一个学生 \n";
    }
}
/**
 * 一个继承于抽象人接口的老师类
 * Class Teacher
 */
class Teacher implements Person{
    public function showInfo()
    {
        echo "这是一个老师 \n";
    }
}
/**
 * 人类工厂
 * Class PersonFactory
 */
class PersonFactory{
    public static function factory($person_type)
    {
        // 将传入的类型首字母大写
        $class_name = ucfirst($person_type);
       if (class_exists($class_name)) {
            return new $class_name;
        } else {
            throw new Exception("类：$class_name不存在",1);
        }
    }
}
// 需要一个学生
$student= PersonFactory::factory('student');
echo $student->showInfo();
// 需要一个老师的时候
$teacher= PersonFactory::factory('teacher');
echo $teacher->showInfo();
```

适配器模式：将各种截然不同的函数接口封装成统一的API，首先定义一个接口(有几个

方法，以及相应的参数)。然后，有几种不同的情况，就写几个类实现该接口。将完成相似功能的函数，统一成一致的方法。

策略模式：将一组特定的行为和算法封装成类，以适应某些特定的上下文环境，用意是对一组算法的封装。动态的选择需要的算法并使用。

57. **实现一个数据结构，要求 set(index, val)、get(index)、setAll(val)三种操作时间复杂度都是O（1）**

重点是set value 的时候记录一下计数器的值，setAll 的时候修改计数器的值，这样查到具体值后，比较当前计数器值和保存的计数器值是否一致，选择返回全局值还是当前值

```
$node = new Node();
$node->set(0, 2);
$node->setAll(3);
return $node->get(0);



class Node {
    private static int $counter = 0;  // 计数器，记录是否被修改过
    private static array $array;      // 数组存储
    private static int $globalValue;  // 全局的值

    public function set($index, $value)
    {
        self::$array[$index] = [self::$counter, $value];
    }


    public function get($index)
    {
        $tmp = self::$array[$index][0];
        $value = self::$array[$index][1];
        if ($tmp == self::$counter) return $value;
        return self::$globalValue;
    }


    public function setAll($value)
    {
        self::$counter++;
        self::$globalValue = $value;
    }
}
```

58. **Redis 缓存击穿，缓存穿透，缓存雪崩区别及解决方案**

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，造成了类似攻击行为

缓存击穿是大批量的请求在访问一个key，这个key失效的瞬间，请求打到了数据库

缓存雪崩是大批量的请求在访问大批量的key，这些key同时失效，所有请求打到数据库，造成数据库无法响应。

避免雪崩是给key加一个随机生存时间，例如都是 3分钟，给他们加一个 random_int(1,30) 这样的时间，不会同时失效，或者热点数据长期有效，至少过完高并发的这几天再失效。

避免穿透是接口层增加校验，比如用户鉴权校验，参数做校验，不合法的参数直接代码return，同时nginx层面限制ip请求频率。也可以借助布隆过滤器，从其中判断key存不存在，不存在直接return；

避免击穿最简单是让key长期有效。

59. **PHP 查找两个有序数组的相同元素**

还是双指针的经典妙用

```
public function findTheSameItems($arr1,$arr2)
{
    $size1 = count($arr1);
    $size2 = count($arr2);
    $i = $j = 0;
    $result = [];
    while (true) {//移动值较小的
        if ($arr1[$i] > $arr2[$j])
            $j++;
        elseif ($arr1[$i] < $arr2[$j])
            $i++;
        else {
            $result[] = $arr1[$i];
            $j++;
        }


        if ($i == $size1 || $j == $size2)
            break;
    }

    return $result;
}
```

配上二分查找的套路：

```
# 二分查找
function binarySearch(array $arr, $target)
{
    $low = 0;
    $high = count($arr) - 1;
    while ($low <= $high) {
        $mid = floor(($low + $high) / 2);
        # 找到元素
        if ($arr[$mid] == $target) return $mid;
        # 中间元素比目标大，查找左边
        if ($arr[$mid] > $target) $high = $mid - 1;
        # 中间元素比右边小，查找右边
        if ($arr[$mid] < $target) $low = $mid + 1;
    }

    #查找失败
    return false;
}
```

借助二分查找实现开根号计算：

```
function mySqrt($x) {
    if ($x <= 1) return $x;
    $left = 1;
    $right = $x - 1;
    while ($left <= $right) {
```

```
            $mid = $left + (($right - $left) >> 1);
            if ($mid > $x / $mid) {
                $right = $mid - 1;
            } else if ($mid < $x / $mid) {
                if ($mid + 1 > $x / ($mid + 1)) return $mid;
                $left = $mid + 1;
            } else {
                return $mid;
            }
        }

        return -1; // only for return a value
    }
```

60.