PHP面试之道

idcpj

目　录

目录

常见经典面试题

什么是引用变量?在PHP当中,用什么符号定义引用变量?
要求写出 jquery中,可以处理AJAX的几种方法。
写出尽可能多的 Linux命令
写出三种以上 MYSQL数据库存储引擎的名称
编写一个在线留言本,实现用户的在线留言功能
谈谈你对MVC的认识介绍几种目前比较流行的MVC框架
请写出常见的排序算法
PHP如何解决网站大流量与高并发的问题

第一章PHP基础知识考察点

什么是引用变量?在PHP当中,用什么符号定义引用变量?

1. PHP的引用变量的概念及定义方式
   概念: 在PHP中引用意味着用不同的名字访问同一个变量内容。
   定义方式: 使用&符号
2. 延伸:PHP引用变量的原理
   普通赋值

```
//定义一个变量
$a= range(8,1988)
//定义变量b,将a变量的值赋值给b
// COW机制 Copy On Write
$b=$a;  //此时, $b和$a指向同一个内存地址,只有当$a改变时,$b才指向新的地址
//对a进行修改
$a= range(8,1999)
```

引用赋值

```
$b=&$a
```

$a和$b永远指向一个变量地址

回话控制

cookie

保存本地中

```
//赋值
setcookie($name, $value, $expire, $path, $domain, $seure);
//取值
$_COOKIE($name);
//删除
setcookie("TestCookie", "", time() - 10000 );


//设置数组
setcookie("user[three]", "cookiethree");
setcookie("user[two]", "cookietwo");
setcookie("user[one]", "cookieone");
```

session

保存在服务器中

```
//开始
session_start();
//赋值
$_SESSION['favcolor'] = 'green';

//清空session
$_SESSION=[];
//删除sesion及cookie中的session_id
session_destroy();
```

```
//删除
unset($_SESSION["newsession"]);

//以下设置表示,在超过1440的sesison,有百分之一的概率删除session,
ini_set('session.gc_probability', 1);
ini_set('session.gc_divisor', 100);
ini_set('session.gc_maxlifetime', 1440); //设置过期时间
```

传递session id

可在cookie被禁用时,也可以使用session_id
常量SID 如果开启了 cookie SID 就为空 如果没开启 SID 等价于 - PHPSESSIONID=session_id()的值,
如何传递 session_id 参考

```
< a href="1.php?<? php echo SID ;?>">下个页面</a>
//等价
< a href="1.php?<?php echo session_name() ;?>= <?php session_id() ;?>">下个页</a>
```

Session存储方式

如果有部署了多个服务器,那么 session_id 不能存在文件中,而是通过 `session_set_save_handler` 设置 存入 `MYSQL、 Memcache、 Redis` 中
php 官网有存储在sql 中的类实现方法

常用设计模式

常见设计模式:工厂模式、单例模式、注册树模式、适配器模式
观察者模式和策略模式

字符串的定义方式

1. 单引号
2. 双引号
3. heredoc 类似双引号

```
$str = <<<EoT
...
EoT
```

5. newdoc 类似单引号

```
$str = <<<'EoT'
```

```
...
'EoT'
```

知识点延伸

浮点数不能进行比较运算

浮点转为 cpu 会损耗,

```
$a=0.8;
$b=0.1;

if ($a+$b==0.8){
    echo $a+$b;
}
```

布尔类型

FALSE的七种情况
整型0、浮点00、布尔 false、空字符串、0字符串、空数组、NULL

数组类型

超全局数组

`$GLOBALS、$_GET、$_POST、$_REQUEST、$_SESSION、 $_COOKIE、$_SERVER、$_FILES、$_E`
$GLOBALS:包含所有已上所有元素

```
$_SERVER['SERVER_ADDR']  //获取服务器的 ip
$_SERVER['REMOTE_ADDR']   //用户 ip

$_SERVER['SERVER_NAME']  //获取服务器的域名  www.example.com
$_SERVER[ REQUEST_TIME'] //请求时间
$_SERVER['QUERY_STRING']  //?后的参数 $_GET 参数
$_SERVER['HTTP_REFERER']  //从哪一页进来
$_SERVER['HTTP_USER_AGENT']  //浏览器信息
$_SERVER['REQUEST_URI']   // 域名后的所有信息 /demo/index.php?name=cpj&age=12
$_SERVER['PATH_INFO'] // 网址路径信息 ,一般用于路由
```

三种情况 为NULL

直接赋值为NULL、未定义的变量、 unset销毁的变量

常量

const、 define
const更快,是语言结构, define是函数
const 可以在类中定义常量,define 不行

优先级

递增减 > ! > 算数运算符 > 大小比较 > (不)相等比较 > 引用 > 位运算符(^) > 位运算符(|) >

短路作用：|| 和 && 与 and 和 or 的优先级不同；

```
//短路作用
$a = true || $b == 3; // $b 不会执行； 相当于$a =( true || $b == 3);
$b = false && $a == 1; // $b 为false 此表达式不会执行
//优先级问题：
$a = false || true;  // $a = true;
$b = false or true; //  $b = false,整体是true；
```

真题

```
$a = 0;
$b = 0;
if($a = 3 > 0 || $b = 3 > 0){  //短路  $a=((3>0)||$b=3>0) 执行3>0后就短路,所以$ a=true
    $a++;
    $b++;
    echo $a;   //$a = 1 (true);
    echo $b;   //$b = 1;
}
```

流程控制

if与 elseif

把 if 范围小的放前面

switch ...case

控制表达式只能是 整型,浮点型和字符串
continue语句作用到 switch的作用类似于 break

break

break 的数字代表跳出几次循环

```
for ($i = 0; $i < 3; $i++){
    echo '外层循环'.$i.' 开始'."\n";
    for ($j = 0; $j < 2; $j++){
```

```
        if ($i == 1){
            break 2;    //使用break 2直接跳出2层循环
        }
        echo '内层循环'.$i.'-'.$j."\n";
    }
    echo '外层循环'.$i.' 结束'."\n";
}
```

函数

函数引用

从函数返回一个引用,必须在函数声明和指派返回值给一个变量时都使用引用运算符 `&`

```
function  myFunc(){
    static $b= 10;
    return $b;
}
$a = myFunc();
$a = & myFunc();  //$a 与$ b 互为应用
$a = 100;
echo myFunc();  //100
```

打印函数

```
//打印一个字符串
print()
//打印一个或多个字符串
echo()

//格式化输出
$num = 2.12;
$d='123a';
printf("%.1f---%d",$num,$d); //2.1---123

//返回格式化值
$num = 2.12;
$d='123a';
echo sprintf("%.1f---%d",$num,$d); //2.1---123
```

正则

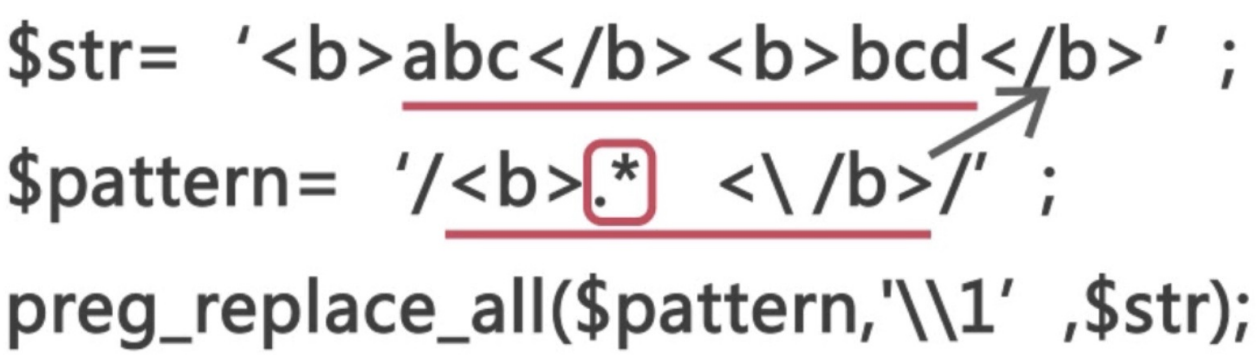
```
正则表达式的作用:分割、查找、匹配、替换字符串
分隔符:正斜线(/)、hash符号(#) 以及取反符号(~)
```

```
通用原子:\d、\D、\w、\W、\s、\S
元字符: * ? A $ + {n } {n,} {n,m} [] () [^]| [-]
模式修正符: i m e s U x A D u
```

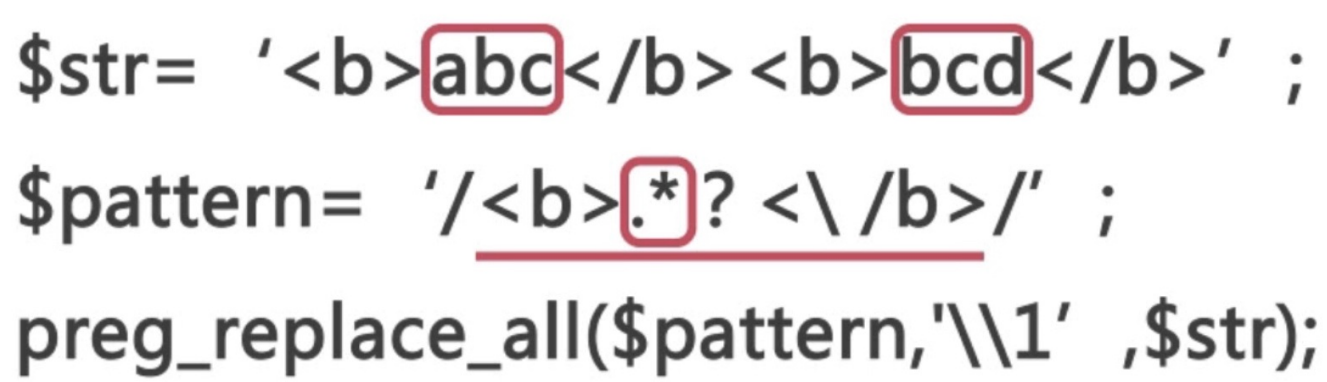
后向引用

```
$str='<b>abc</b>';
Pattern= '/<b>(.*)<\/b>/';
preg_replace($pattern, '\\1', $str);
```

贪婪模式



用 `.*?` 取消贪婪模式



用 `U` 取消贪婪
`$pattern = '/<b>.*</b>/U'`

常用函数

```
preg_match(),
preg_match_all(),
Preg_replace(),
preg_split()
```

中文匹配
UTF-8汉字编码范围是 `0x4e00-0x9fa5`,
ANSI(gb2312)环境下, `0xb0-0xf7`, `0xa1-0xfe`

UTF-8要使用u模式修正符使模式字符串被当成UTF-8,
ANSI(gb2312)环境下,要使用chr将Asc码转换为字符

```
$str='中文'
Pattern ='/[\xt{4e001-\x{t9fa5}]+/u';
```

匹配 img 的 src 值

```
$str='<img alt="测试" id="aaa"  src="aaa.jpg" />';
$pattern='/<img. *?src="(.*?).*?\/?>/i'  // .*? 为取消贪婪
Preg_match( $Pattern, $str, $match);
```

文件操作

文件读取,写入

```
模式 描述
r    打开文件为只读。文件指针在文件的开头开始。
w    打开文件为只写。删除文件的内容或创建一个新的文件，如果它不存在。文件指针在文件的开头开始。
a    打开文件为只写。文件中的现有数据会被保留。文件指针在文件结尾开始。创建新的文件，如果文件不存在。
x    创建新文件为只写。返回 FALSE 和错误，如果文件已存在。
r+   打开文件为读/写、文件指针在文件开头开始。
w+   打开文件为读/写。删除文件内容或创建新文件，如果它不存在。文件指针在文件开头开始。
a+   打开文件为读/写。文件中已有的数据会被保留。文件指针在文件结尾开始。创建新文件，如果它不存在。
x+   创建新文件为读/写。返回 FALSE 和错误，如果文件已存在。
```

打开
`fopen("webdictionary.txt", "r")`

读取

```
fread() 读取文件
fgets()  读取一行
fgetc()  读取一个字符
```

关闭
`fclose()`

file_get_content() 与 file_put_content() 性能更好

其他

```
file()  以数组形式读取字符串
```

目录操作函数

```
名称相关: basename()、 dirname()、 patino()
目录读取: opendir()、 readdir()、 closedir()、 rewinder()
目录删除: rmdir()  //只有目录中没有文件,才可以删除
目录创建: mkdir()
```

其他函数

```
文件大小: filesize()  //目录文件大小,需要遍历每个文件
目录大小: disk_free_space()  //磁盘可用空间、 disk_total_space()  //总磁盘空间
文件拷贝:copy()
删除文件: unlink()
文件类型: filetype()
重命名文件或者目录: rename()
文件截取: truncate()
文件属性: file_exists()、 is_readable()、 is_writable()、
is_executable()、 filectime()、 fileatime()、 fillemtime()
```

考题范围

文件操作模式,目录的遍历,目录的删除

```
//在文件开头写入 hello word
$path = 'demo.txt';
$handle = fopen($path, 'r');
$content = fread($handle, filesize($path));
$content = "hello word" . $content;
fclose($handle);
//写入
$handle = fopen($path, 'w');
fwrite($handle, $content);
fclose($handle);
```

目录的遍历

```
function loopdir($dir){
    $handle = opendir($dir);
    while(false!==($file=readdir($handle))){
        if ($file !='.' &&     $file!='.'){
           echo $file."\n";
           if (filetype($dir.'/'.$file)=='dir'){
               loopdir($dir.'.'.$file);
           }
        }
    }
}
loopdir($dir);
```

第二章 Javascript、 jquery、AJAX基础知识考察点

javascript

数据类型

字符串、数字、布尔、数组、对象、Null、 Undefined
Javascript变量均为对象。当您声明一个变量时,就创建了一个新的对象。

函数

- 无默认值
- 函数内部声明的变量(使用var)是局部变量
- 在函数外声明的变量是全局变量,所有脚本和函数都能访问它(与 php 不同, php 外部变量也不可再函数 内部使用)

内置对象

Number

```
var p=3.14;
var mynum=new Number(value);
var my_Num= Number(value);
```

String

```
var str='This is String';
var str=new String(s);
```

```
var str=String(s);
```

Array

```
var arr=new Array ();
var arr=new Array(size)
var arr=new Array(el, e2, e3,.en);
```

Date

```
var date=new Date ();
```

Math

```
var pi_value= Math.PI
var sqrt_value=Math sqrt(9);  //9 开根
```

Regexp

```
/pattern/attributes
new Regexp(pattern, attributes)
```

Window对象

`Window、 Navigator、 Screen、 History、 Location`

DOM对象

`Document、Element、Attr、 Event`

第三章 Linux基础知识考察点

常用命令

系统安全

`sudo、su、 chmod、setfac`

进程管理

`w、top、ps、kill、 pkill,、 pstree、 killall`

用户管理

`id、 usermod、 useradd、 groupadd、 userdel`

文件系统

`mount、 umount、fsck、df、du`

网络应用

`curl、 telnet、mail、 elinks`

网络测试

`ping、 netstat、host`

网络配置

`hostname、 ifconfig`

软件包管理

`yum、rpm、apt-get`

文件内容查看

`head、tail、less、more`

文件处理

`touch、 unlink、 rename、ln、cat`

目录操作

`cd、mv、rm、pwd、tree、cp、ls`

文件权限属性

`setfac、 chmod、 chown、 chgrp`

文件传输

`ftp、scp`

定时任务

crontab命令

`crontab -e`
`* * * * * 命令(分时日月周)`

at命令

一次性执行命令

```
at 2: 00 tomorrow
at>/home/Jason/do job
at> Ctrl+D
```

shell 命令

脚本执行方式

1. 赋予权限,直接执行,
   `chmod+ x test.sh;/test.sh`
2. 调用解释器使得脚本执行,
   `bash、csh、csh、ash、bsh、ksh等等`

编写基础

开头用#!指定脚本解释器
`#!/bin/sh`

第四章 MYSQL基础知识考察点

字段类型

整数类型

`TINYINT、 SMALLINT、 MEDIUMINT、INT、 BIGINT`
属性: `UNSIGNED`
长度:可以为整数类型指定宽度,例如:INT(11)、对大多数应用
是没有意义的,它不会限制值的合法范围,只会影响显示字符的个
数

> int(3) 可以存入值为1234567 ,且不会切断,
> 设置长度是为了 如果设置了 `Zerofill` 存入123 会变为 0123,
> 所以 int(0) 也是具有意义的

实数类型

`FLOAT、 DOUBLE、 DECIMAL`
DECIMAL可存储比 BIGINT还大的整数;可以用于存储精确的小数,相当于存为字符串
FLOAT和 DOUBLE类型支持使用标准的浮点进行近似计算

字符串类型

`VARCHAR、CHAR、TEXT、BLOB`
VARCHAR类型用于存储可变长字符串,它比定长类型更节省空间
VARCHAR使用1或2个额外字节记录字符串的长度,列长度小于
255字节,使用1个字节表示,否则用2个

> `char` 定长 ,如 `char(10)` 不管你存入多长, 都占用10 个字节,用于存储定长的字符串
> `varchar` 与 `char` 会切断长度长度的字符串 ,超过255字节的只能用 `varchar` 或者 `text` ;
> 如果存储值很短或定长,可以选择 `char`
> 能用 `varchar` 的地方不用 `text` ;

枚举

```
create table user_sex( sex enum('M','F'));
```

有时可以使用枚举代替常用的字符串类型
把不重复的集合存储成一个预定义的集合
非常紧凑,把列表值压缩到一个或两个字节
部存储的是整数

日期和时间类型

尽量使用 `TIMESTAMP` ,比 `DATETIME` 空间效率高

用整数保存时间戳的格式通常不方便处理
如果需要存储微秒,可以使用 bigint存储

```
date    1000-01-01 ~ 9999-12-31
datetime    1000-01-01 00:00:00 ~ 9999-12-31 23:59:59
timestamp    1970-01-01 00:00:01  ~ 2038-01-19 03:14:07
```

数据表引擎

Innodb表引擎

默认事务型引擎,最重要最广泛的存储引擎,性能非常优秀
数据存储在共享表空间,可以通过配置分开
对主键查询的性能高于其他类型的存储引擎
内部做了很多优化,从磁盘读取数据时自动在内存构建hash索引
插入数据时自动构建插入缓冲区
通过一些机制和工具支持真正的热备份
支持崩溃后的安全恢复
支持行级锁
支持外键

MYISAM表引擎

51版本前, MYISAM是默认的存储引擎
拥有全文索引、压缩、空间函数
表锁,不支持事务和行级锁,不支持崩溃后的安全恢复
表存储在两个文件,MYD和MYI
设计简单,某些场景下性能很好

其他表引擎

Archive、 Blackhole、csV、 Memory

锁机制

表锁是日常开发当中常见的问题,当多个查询同一时刻进行数据修改时,就会产生并发控制的问题

共享锁(读锁)和排他锁(写锁)

读锁

共享的,不堵塞,多个用户可以同时读一个资源,互不干扰

写锁

排他的,一个写锁会阻塞其他的写锁和读锁,这样可以只允许一人进行写入,防止其他用户读取正在写入的资源。

锁粒度

- 表锁,系统性能开销最小,会锁定整张表, MYISAM使用表锁
- 行锁,最大程度地支持并发处理,但是也带来了最大的锁开销,Innodb实现行级锁

事务处理

MYSQL提供事务处理的表引|擎, Inno DB
服务器层不管理事务,由下层的引擎实现,所以同一个事务中,使
用多种存储引擎不靠谱
在非事务的表上执行事务操作 MYSQL不会发出提醒,也不会报错

存储过程

为以后的使用而保存的一条或多条 MYSQL语句的集合
存储过程就是有业务逻辑和流程的集合
可以在存储过程中创建表,更新数据,删除等等

触发器

提供给程序员和数据分析员来保证数据完整性的一种方法,它是与
表事件相关的特殊的存储过程

使用场景

可通过数据库中的相关表实现级联更改
实时监控某张表中的某个字段的更改而需要做出相应的处理
某些业务编号的生成等
滥用会造成数据库及应用程序的维护困难

> MySQL 视图、函数、存储过程和触发器 简书 详解

索引的基础和类型

索引对性能的影响

大大减少服务器需要扫描的数据量

帮助服务器避免排序和临时表
将随机I/O变顺序I/O
大大提高查询速度,降低写的速度、占用磁盘

索引的使用场景

对于非常小的表,大部分情况下全表扫描效率更高
特大型的表,建立和使用索引的代价将随之增长,可以使用分区技术来解决

索引的类型

索引||有很多种类型,都是实现在存储引擎层的

- 普通索引:最基本的索引,没有任何约束限制
- 唯一索引:与普通索引类似,但是具有唯一性约束
- 主键索引:主键可以与外键构成参照完整性约束,防止数据不一致
- 组合索引:将多个列组合在一起创建索引,可以覆盖多个列
- 外键索引:只有 Innodb类型的表才可以使用外键索引,保证数据的一致性、完整性和实现级联操作
- 全文索引: MYSQL自带的全文索引只能用于 MYISAM,并且只能对英文进行全文检索

创建索引的原则

1. 最适合索引的列是出现在 WHERE子句中的列,或连接子句中的列而不是出现在 SELECT关键字后的列
2. 索引列的基数越大,索引的效果越好
3. 对字符串进行索引,应该制定一个前缀长度,可以节省大量的索引空间
4. 根据情况创建复合索引,复合索引可以提高查询效率
5. 避免创建过多索引,索引||会额外占用磁盘空间,降低写操作效率
6. 主键尽可能选择较短的数据类型,可以有效减少索引的磁盘占用提高查询效率

索引的注意事项

1. 复合索引遵循前缀原则

```
创建的索引有顺序,非顺序的索引不生效
KEY(a, b, c)  //创建 a b c 索引
WHERE a=l and b=2 and c= 3 //生效
WHERE a=1 and b=2   //生效
WHERE a= 1  //生效

WHERE b=2 andc= 3   //无效
WHERE a= 1 and c=3      //无效
```

2. like查询,%在前,则素以失效,可以使用全文索引
3. column is null可以使用索引

`where name=null //也是可以使用索引`

4. 如果 MYSQL估计使用索引比全表扫描更慢,会放弃使用索引
   `如果只有100 条数据 where id 1>and id<100 会自动转为全表索引`
5. 如果or前的条件中的列有索引,后面的没有,索引都不会被用到
   `where name='cpj' or age='12'`
6. 列类型是字符串,查询时一定要给值加引|号,否则索引失效

```
name varchar(16)
存了 "100"
Where name =100  //虽然能得到值 但是没有使用索引
```

六种关联查询

交叉连接 CROSS JOIN

```
SELECT* FROM A,B(C)或者
SELECT* FROM A CROSS JOIN B(CROSS JOIN C
//没有任何关联条件,结果是笛卡尔积,结果集会很大,没有意义,很少使用
```

内连接 INNER JOIN

```
SELECT* FROM A, B WHERE A id=Bd或者
SELECT FROM A INNER JOIN B ON A.id=Bid
```

内连接分为三类

- 等值连接: ON Aid=Bid
- 不等值连接: ON A id>B.d
- 自连接: `SELECT* FROM A T1 INNER JOIN A T2 on T1.id=T2. pid`
-

外连接 LEFT JOIN/ RIGHT JOIN

左外连接

LEFT OUTER JOIN,以左表为主,先查询出左表,按照ON后的关联条件匹配右表,没有匹配到的用NULL填充,可以简写成 LEFT JOIN

右外连接

RIGHT OUTER JOIN,以右表为主,先查询出右表,按照ON后的关联条件匹配左表,没有匹配到的用NULL填

充,可以简写成 RIGHT JOIN

联合查询 UNION与 UNION ALL

`SELECT* FROM A UNION SELECT* FROM B UNION··`
就是把多个结果集集中在一起, UNION前的结果为基准,需要注意的是联合查询的列数要相等,相同的记录行会合并
UNION ALL不会合并重复的结果集

嵌套语句

用一条SQL语句的结果作为另外一条SQL语句的条件
`SELECT FROM A WHERE id IN(SELECT id FROM B)`

真题

1. 更新 b 表 的 c1 c2 到 A 表的 c1 c2

```
A(id, sex, par, cl, c2)
B(id, age, cl, c2)
//方法一
update a, b set A.cl= B.cl, A.c2=B.c2
where A.id = B.id and B.age >50
//方法二
update a inner join B on Aid B id
set A.c1=B.c1,A.c2= B.c2
where Bage >50
```

mysql查询优化

分析SQL查询慢的方法

1. 记录慢查询日志
   分析查询日志,不要直接打开慢查询日志进行分析,这样比较浪费
   时间和精力,可以使用pt- query- digest工具进行分析
2. 使用 show profile
   `set profiling=1` ;开启,服务器上执行的所有语句会检测消耗的时
   间,存到临时表中

```
set profiling=1;   //开启 profile
show profiles
show profile for query临时表ID
```

demo

```
mysql > set profiling=1;s
mysql > select from a;
mysql> show profiles;
```

3. 使用 show status
   `show status`会返回一些计数器, `show global status`查看服务器级别的所有计数有时根据这些计数,可以猜测出哪些操作代价较高或者消耗时间多
4. 使用 show processlist
   观察是否有大量线程处于不正常
5. 使用 explain
   分析单条SQL语句
   `mysql> explain select * from fq_order\G;`

优化查询过程中的数据访问

- 访问数据太多导致查询性能下降
- 确定应用程序是否在检索大量超过需要的数据,可能是太多行或列
- 确认 MYSQL服务器是否在分析大量不必要的数据行
- 查询不需要的记录,使用 limit解决
- 多表关联返回全部列,指定Aid,A.name,Bage
- 总是取出全部列, `SELECT*`会让优化器无法完成索引覆盖扫描的优化
- 重复查询相同的数据,可以缓存数据,下次直接读取缓存

是否在扫描额外的记录

使用 explain来进行分析,如果发现查询需要扫描大量的数据但只返回少数的行,可以通过如下技巧去优化:
使用索引覆盖扫描,把所有用的列都放到索引中,这样存储引擎不需要回表获取对应行就可以返回结果

一个复杂查询好与多个简单查询

- MysαL内部每秒能扫描内存中上百万行数据,相比之下,响应数据给客户端就要慢得多
- 使用尽可能少的查询是好的,但是有时将一个大的查询分解为多个小的查询是很有必要

切分查询

将一个大的查询分为多个小的相同的查询一次性删除1000万的数据要比一次删除1万,暂停一会的方案更加损耗服务器开销

分解关联查询

- 可以将一条关联语句分解成多条SQL来执行
- 让缓存的效率更高
- 执行单个查询可以减少锁的竞争
- 在应用层做关联可以更容易对数据库进行拆分
- 查询效率会有大幅提升
- 较少冗余记录的查询

优化 count()查询

- `count(*)`中的`*`会忽略所有的列,直接统计所有列数,因此不要使用 count(列名)
- MYISAM中,没有任何 WHERE条件的`count(*)`非常快,当有 WHERE条件, MYISAM的 count统计不一定比其他表引擎快
- 可以使用 explain查询近似值,用近似值替代 count()
- 增加汇总表
- 使用缓存

优化关联查询

- 确定ON或者 USING子句的列上有索引
- 确保 GROUP BY和 ORDER BY中只有一个表中的列,这样 MYSQL才有可能使用索引

优化子查询

尽可能使用关联查询来替代

优化 GROUP BY和 DISTINCT

- 这两种查询均可使用索引来优化,是最有效的优化方法
- 关联查询中,使用标识列进行分组的效率会更高
- 如果不需要 ORDER BY,进行 GROUP BY时使用 ORDER BY NULL, MYSQL不会再进行文件排序
- WITH ROLLUP超级聚合,可以挪到应用程序处理
  > 分类聚合后的结果进行汇总 参考用法简书

优化UNON查询

UNION ALL的效率高于 UNION

分区表的原理

对用户而言,分区表是一个独立的逻辑表,但是底层 MYSQL将其分成了多个物理子表,这对用户来说是透明的,每一个分区表都会使用一个独立的表文件 .

创建表时使用 partition by子句定义每个分区存放的数据,执行查询时,优化器会根据分区定义过滤那些没有我们需要数据的分区,这样查询只需要查询所需数据在的分区即可

分区的主要目的是将数据按照一个较粗的粒度分在不同的表中,这样可以将相关的数据存放在一起,而且如果想一次性删除整个分区的数据也很方便

适用场景

1. 表非常大,无法全部存在内存,或者只在表的最后有热点数据,其他都是历史数据
2. 分区表的数据更易维护,可以对独立的分区进行独立的操作
3. 分区表的数据可以分布在不同的机器上,从而高效使用资源
4. 可以使用分区表来避免某些特殊的瓶颈
5. 可以备份和恢复独立的分区

限制

1. 一个表最多只能有1024个分区
2. 5.1版本中,分区表表达式必须是整数,5.5可以使用列分区
3. 分区字段中如果有主键和唯一索引列,那么主键列和唯一列都必须包含进来
4. 分区表中无法使用外键约束
5. 需要对现有表的结构进行修改
6. 所有分区都必须使用相同的存储引擎
7. 分区函数中可以使用的函数和表达式会有一些限制
8. 某些存储引擎不支持分区
9. 对于 MYISAM的分区表,不能使用 load index into cache
10. 对于 MYISAM表,使用分区表时需要打开更多的文件描述符

分库分表的原理

工作原理

通过一些HASH算法或者工具实现将一张数据表垂直或者水平进行物理切分

适用场景

1. 单表记录条数达到百万到千万级别时
2. 解决表锁的问题

分表方式

水平分割

表很大,分割后可以降低在查询时需要读的数据和索引的页数,同时也降低了索引的层数,提高查询速度

使用场景

1. 表中的数据本身就有独立性,例如表中分别记录各个地区的数据或者不同时期的数据,特别是有些数据常用,有些不常用
2. 需要把数据存放在多个介质上

水平分表缺点

1. 给应用增加复杂度,通常查询时需要多个表名,查询所有数据都需 `UNION` 操作
2. 在许多数据库应用中,这种复杂性会超过它带来的优点,查询时会增加读一个索引层的磁盘次数

垂直分表

把主键和一些列放在一个表,然后把主键和另外的列放在另一个表中,就是多一张详情表

使用场景

1. 如果一个表中某些列常用,而另外一些列不常用
2. 可以使数据行变小,一个数据页能存储更多数据,查询时减少I/O次数

垂直分表缺点

1. 管理冗余列,查询所有数据需要JoIN操作

MYSQL的复制原理及负载均衡

MYSQL主从复制工作原理

在主库上把数据更改记录到二进制日志
从库将主库的日志复制到自己的中继日志
从库读取中继日志中的事件,将其重放到从库数据中

MYSQL主从复制解决的问题

数据分布:随意停止或开始复制,并在不同地理位置分布数据备份
负载均衡:降低单个服务器的压力
高可用和故障切换:帮助应用程序避免单点失败
升级测试:可以使用更高版本的 MYSQL作为从库

解题方法

充分掌握分区和分表的工作原理和适用场景,在面试中,此类题通

常比较灵活,会给一些现有公司遇到问题的场景,大家可以根据分区和分表以及Mysα复制、负载均衡的适用场景来根据情况进行回答。

安全性

安全注入

1. 使用预处理语句防SQL注入
2. 写入数据库的数据要进行特殊字符的转义
3. 查询错误信息不要返回给用户,将错误记录到日志

MYSQL的其他安全设置

1. 定期做数据备份
2. 不给查询用户root权限,合理分配权限
3. 关闭远程访问数据库权限
4. 修改root口令,不用默认口令,使用较复杂的口令
5. 删除多余的用户
6. 改变root用户的名称
7. 限制一般用户浏览其他库
8. 限制用户对数据文件的访问权限

第五章简单程序设计题考察点

留言板

PDO

```
$pdo =  new PDO($dsn, $username, $password, $attr);
$sql ='select id, title, content FROM message where user name=:user name
$stmt= $pdo->prepare($sql);
$stmt->execute([:user_name=>$user namel);
$result= $stmt->fetchAll(PDO: FETCH-ASSOC);
```

无限分类

id	title	pid	path
1	服装	0	0-1
2	上衣	1	0-1-2
3	长袖	2	0-1-2-3

方法一 : 根据 pid

方法二 : 根据 path

第六章PHP框架基础知识考察点

第七章算法、逻辑思维考察点

算法的概念

基本概念
个问题可以有多种算法,每种算法都不同的效率一个算法具有五个特征:有穷性、确切性、输入项、输出项、可行性

时间复杂度和空间复杂度的概念

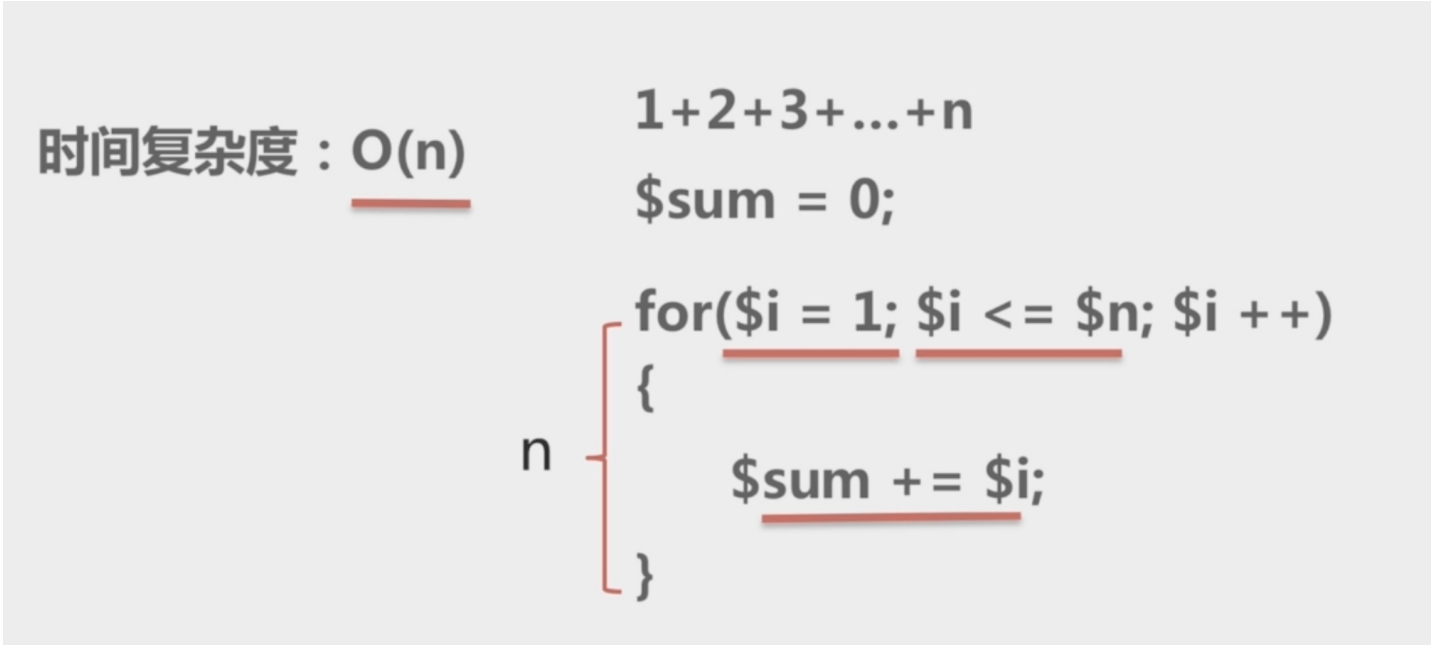
算法评定
算法分析的目的在于选择合适算法和改进算法一个算法的评价主要从时间复杂度和空间复杂度来考虑

时间复杂度

执行算法所需要的计算工作量。一般来说,计算机算法是问题规模n的函数f(n),算法的时间复杂度也因此记做`T(n)=O(f(n))`
问题的规模n越大,算法执行的时间的增长率与f(n)的增长率正相关,称作渐进时间复杂度(Asymptotic Time Complexity)

时间复杂度计算方式

o(n^2)、O(1)、O(n)?
如果 n 的数未知 那么复杂度就是 `O(n)`,
如果 n 个数,但是支取前三个,则不用 `O(3)`,而是用 `O(1)`,括号里为常数则多为1



- 得出算法的计算次数公式
- 用常数1来取代所有时间中的所有加法常数
- 在修改后的运行次数函数中,只保留最高阶项
  如 `O(n^2+n+1)` 则只要 `O(n^2)`
- 如果最高阶存在且不是1,则去除与这个项相乘的常数

举例

常数阶: `O(1)`
线性阶: `O(n)`
平(立)方阶: `O(n^2)` 或 `o(n^3)`

```
由于是循环了n 的平方 所以其复杂度 为 O(n^2)
for($i=1;$i<=$n;$i++){
    for($j=1;$j<=$n;$j++){
        $sum +=$
    }
}
```

特殊平方阶: `o(n^2/2+n/2)->o(nA2)`

```
// 其复杂度 为 `O(n^2+n+1) -> O(n^2)
for(){
    for(){}
}
for(){}
```

```
echo $n
```

对数阶`O(log(2n))`;

```
while($n >=1)(
    $n=$n/2
}

n(2^m)=1  -> 2^m=2  -> m=log(2n)  -> O(log(2n))
```

常见时间复杂度:常数阶、线性阶、平方阶、立方阶、对数阶、
nlog2n阶、指数阶
`O(1)>O(log2n)>O(n)>O(nlog2n)>O(n^2)>(n^3)>O(2^n)>O(n!)>O(n^n)`

空间复杂度

算法需要消耗的内存空间,记作`S(n)=O(f(n))`
包括程序代码所占用的空间,输入数据所占用的空间和辅助变量所占用的空间这三个方面
计算和表示方法与时间复杂度类似,一般用复杂度的渐近性来表示
有时用空间换取时间
冒泡排序的元素交换,空间复杂度O(1),因为要加的数永远只有一个

排序算法

冒泡排序

原理:两两相邻的数进行比较,如果反序就交换,否则不交换时间复杂度:最坏`(O(n^2),`平均`(O(n^2)`
如: 对,1,2,3,4,5,6进行 排序

直接插入排序

原理
每次从无序表中取出第一个元素,把它插入到有序表的合适位置,使有序表仍然有序
时间复杂度:最坏(O(n^2),平均(O(n^2)
空间复杂度:O(1)

希尔排序

原理
把待排序的数据根据增量分成几个子序列,对子序列进行插入排序,直到增量为1,直接进行插入排序;增量的排序,一般是数组的长度的一半,再变为原来增量的一半,直到增量为1
时间复杂度:最差(O(n^2)),平均(O(n(log2n)
空间复杂度:O(1)

选择排序

原理
每次从待排序的数据元素中选出最小(或最大)的一个元素,存放在序列的起始位置,直到全部待排序的数据元素排完
时间复杂度:最坏(O(n^2),平均(O(n^2)
空间复杂度:O(1)

快速排序

原理
通过一趟排序将要排序的数据分割成独立的两部分,其中一部分的所有数据都比另外一部分的所有数据都要小,然后再按照此方法对这两部分数据分别进行快速排序,整个排序过程可以递归完
成
时间复杂度:最差(O(n^2),平均(O(n(log2n)
空间复杂度:最差(O(n),平均(O(log2n)

堆排序

原理
把待排序的元素按照大小在二叉树位置上排列,排序好的元素要满足:父节点的元素要大于等于子节点;这个过程叫做堆化过程,如果根节点存放的是最大的数,则叫做大根堆,如果是最小,就叫小根堆,可以把根节点拿出来,然后再堆化,循环到最后一个节点
时间复杂度:最差(O(n(log2n),平均(O(nl(og2n)
空间复杂度:O(1)

归并排序

原理
将两个(或两个以上)有序表合并成一个新的有序表,即把待排序序列分为若干个有序的子序列,再把有序的子序列合并为整体有序序列
时间复杂度:最差O(n(log2n),平均(O(n(log2n)
空间复杂度:O(n)

查找算法

二分查找

原理:从数组的中间元素开始,如果中间元素正好是要查找的元素,搜索结束,如果某一个特定元素大于或者小于中间元素,则在数组大于或者小于中间元素的那一半中查找,而且跟开始一样从中间开始比较,如果某一步骤数组为空,代表找不到。
时间复杂度:最差(O(log2n),平均(O(log2n)

空间复杂度:迭代(O(1))、递归(O(log2n))

顺序查找

原理:按一定的顺序检查数组中每一个元素,直到找到所要寻找的特定值为止。
时间复杂度:最差(O(n),平均(O(n))
空间复杂度:O(1)

第八章数据结构

常见数据结构

Array-数组

数组,最简单而且应用最广泛的数据结构之特性:使用连续的内存来存储、数组中的所有元素必须是相同的类型或类型的衍生(同质数据结构)、元素可以通过下标直接访问

Linkedlist-链表

链表,线性表的一种,最基本、最简单,也是最常用的数据结构特性:元素之间的关系是一对一的关系(除了第一个和最后一个元素,其他元素都是首尾相接)、顺序存储结构和链式存储结构两种存储方式

Stack-栈

栈,和队列相似,一个带有数据存储特性的数据结构特性:存储数据是先进后出的、栈只有一个出口,只能从栈顶部增加和移除元素

Heap-堆

堆,一般情况下,堆叫二叉堆,近似完全二叉树的数据
结构特性:子节点的键值或者索引!总是小于它的父节点、每个节点的左右子树又是一个二叉堆、根节点最大的堆叫最大堆或者大根堆、最小的叫最小堆或者小根堆

list

线性表,由零个或多个数据元素组成的有限序列
特性:线性表是一个序列、0个元素构成的线性表是空表、第一个元素无先驱、最后一个元素无后继、其他

元素都只有一个先驱和后继、有长度,长度是元素个数,长度有限

doubly-linked-list

双向链表
特性:每个元素都是一个对象,每个对象有一个关键字key和两个指针(next和prev)

qlueue

队列
特性:先进先出(FIFO)、并发中使用、可以安全将对象从一个任务传给另一个任务

set

集合
特性:保存不重复元素

map

字典
特性:关联数组、也被叫做字典或者键值对

graph

图
特性:通常使用邻接矩阵和邻接表表示、前者易实现但是对于稀疏
矩阵会浪费较多空间、后者使用链表的方式存储信息但是对于图搜
索时间复杂度较高

第九章高并发和大流量解决方案

我们说的高并发是什么?

上面的定义明显不是我们通常所言的并发,在互联网时代,所讲的并发、高并发,通常是指并发访问。也就是在某个时间点,有多少个访问同时到来
通常如果一个系统的日PV在千万以上,有可能是一个高并发的系统

高并发的问题,我们具体该关心什么?

- QPS
  每秒钟请求或者查询的数量,在互联网领域,指每秒响应请求数(指HTTP请求);一个页面中可能有多个http 请求
  `(总PV数*80%)/(6小时秒数*20%)=峰值每秒请求数(QPS)`
- 吞吐量
  单位时间内处理的请求数量(通常由QPS与并发数决定)
- 响应时间
  从请求发出到收到响应花费的时间。例如系统处理一个HTTP请求需要100ms,这个100ms就是系统的响应时间
- PV
  综合浏览量(Page view),即页面浏览量或者点击量,个访客在24小时内访问的页面数量
- UV
  独立访客(Unique Visitor),即一定时间范围内相同访客多次访问网站,只计算为1个独立访客
- 带宽
  计算带宽大小需关注两个指标,峰值流量和页面的平均大小
  `日网站带宽=PV/统计时间(换算到秒)*平均页面大小(单位KB)*8`

常用性能测试工具

ab、wrk、 http_load、 Web bench、 Siege、 Apache Jmeter

ab 概念

全称是 apache benchmark,是 apache官方推出的工具刨建多个并发访问线程,模拟多个访问者同时对某一URL地址进行访问。它的测试目标是基于URL的,因此,它既可以用来测试apache的负载压力,也可以测试nginx、 lighthttp、 tomcat、IS等其它Web服务器的压力

ab的使用

模拟并发请求100次,总共请求5000次
`ab -c 100 -n 5000 www.demo.com`

注意事项

测试机器与被测试机器分开
不要对线上服务做压力测试
观察测试工具ab所在机器,以及被测试的前端机的cPU,内存,网络等都不超过最高限度的75%（`top` 命令)
demo

```
> ab -c 100 -n 100
>
Concurrency Level:      100
Time taken for tests:   55.923 seconds
Complete requests:      1000
Failed requests:        3
   (Connect: 0, Receive: 0, Length: 3, Exceptions: 0)
Total transferred:      2958950 bytes
HTML transferred:       2495491 bytes
Requests per second:    17.88 [#/sec] (mean)    # 这里是QPS数,每秒最高请求数,越高越好
Time per request:       5592.287 [ms] (mean)
Time per request:       55.923 [ms] (mean, across all concurrent requests)
Transfer rate:          51.67 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.1      0       5
Processing:   408 5430 1947.2   5391   11044
Waiting:        0 5422 1953.9   5391   11043
Total:        413 5430 1946.6   5391   11044

Percentage of the requests served within a certain time (ms)
  50%   5391            ## 平均页面响应时间 5391毫秒
  66%   6127
  75%   6684
  80%   7049
```

```
  90%   8073
  95%   8741
  98%   9532
  99%  10089
 100%  11044 (longest request)  ## 最长页响应时间
```

QPS 划分

QPS达到50

可以称之为小型网站,一般的服务器就可以应付,无需优化

QPS达到100

假设关系型数据库的每次请求在0.01秒完成
假设单页面只有一个SQL查询,那么100QPS意味着1秒钟完成100
次请求,但是此时我们并不能保证数据库查询能完成100次

方案:数据库缓存层、数据库的负载均衡

QPS达到800

假设我们使用百兆带宽,意味着网站出口的实际带宽是8M左右假设每个页面只有10K,在这个并发条件下,百兆带宽已经吃完

方案:CDN加速、负载均衡

QPS达到1000

假设使用 Memcache缓存数据库查询数据,每个页面对Memcache的请求远大于直接对DB的请求
Memcache的悲观并发数在2W左右,但有可能在之前内网带宽已经吃光,表现出不稳定

方案:静态HTML缓存

QPS达到2000

这个级别下,文件系统访问锁都成为了灾难

方案:做业务分离,分布式存储

数据库的优化

> msql 优化的详细方法

数据库的优化

数据库的缓存(memcache 缓存,redis 缓存等)
分库分表、分区操作
读写分离
负载均衡

数据表数据类型优化

字段使用什么样的数据类型更合适

tinyint (0-255) smallint , bigint
char,vachar
enum 特定、固定的分类可以使用enUm存储,效率更快
IP地址的存储 //用 php 的 `ip2long('192.168.1.38'); //3232235814`
对字段进行 not null 这样,存储的字段就不会有 null 值 ,只有空值

索引优化

索引不是越多越好,在合适的字段上创建合适的索引
复合索引的前缀原则
复合索引的前缀原则
lke查询%的问题(% 在前如: `%name` 则索引失败)
全表扫描优化
or条件索引使用情况

字符串类型索引失效的问题(如字符串类型的字段必须要加引号查询)

SQL语句的优化

优化查询过程中的数据访问
优化长难句的查询语句
优化特定类型的查询语句
使用 Limit
返回列不用*
变复杂为简单
切分查询(如删大量数据时,可分多次删除)
分解关联查询
优化 count() (如对统计数据单独存放在一个字段,而不是进行 count() 统计)
优化关联查询
优化子查询
优化 Group by和 distinct
优化 limit和 union

存储引擎优化

尽量使用 Inno DB存储引擎

数据表结构设计的优化

分区操作

通过特定的策略对数据表进行物理拆分
对用户透明
partition by
对新建表进行分区

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
```

```
);
```

对已有表进行分区

```
ALTER TABLE user PARTITION BY RANGE (id)
(
PARTITION p_Apr VALUES LESS THAN (2),
PARTITION p_May VALUES LESS THAN (4),
PARTITION p_Dec VALUES LESS THAN MAXVALUE
);
```

分库分表

水平拆分
垂直拆分

数据库服务器架构的优化

- 主从复制
- 读写分离
- 双主热备
- 负载均衡:
  通过LVS的三种基本模式实现负载均衡
  My Cat数据库中间件实现负载均衡

流量优化-防盗链

防盗链处理

- 盗链概念
  盗链是指在自己的页面上展示一些并不在自己服务器上的内容获得他人服务器上的资源地址,绕过别人的资源展示页面,直接在自己的页面上向最终用户提供此内容
  常见的是小站盗用大站的图片、音乐、视频、软件等资源
  通过盗链的方法可以减轻自己服务器的负担,因为真实的空间和流量均是来自别人的服务器
- 防盗链概念
  防止别人通过一些技术手段绕过本站的资源展示页面,盗用本站的
  资源,让绕开本站资源展示页面的资源链接失效
  可以大大减轻服务器及带宽的压力
- 工作原理
  通过 Referer或者签名,网站可以检测目标网页访问的来源网页,如果是资源文件,则可以跟踪到显示它的网页地址。一旦检测到来源不是本站即进行阻止或者返回指定的页面
- Referer 方式防盗链
  Nginx模块 `ngx_http_referer_module` 用于阻挡来源非法的域名请求
  Nginx指令 `valid referers` ,全局变量 `$invalid referer`
  `valid_referers none I blocked I server_names,string`

```
none:" Referer"来源头部为空的情况
blocked:" Referer"来源头部不为空,但是里面的值被代理或者防火墙删除了,这些值都不以http:/或者 ht
tps://开头.
server names:“ Referer"来源头部包含当前的 server names
```

```
location ~ .*\.(gifljpglpnglflvlswfrarlzips)${
    valid_referers none blocked imooc.com * imooc.com:
    if($invalid_referer){
        #return 403:
        rewrite ^/ http://www.imooc.com/403.jpg;
    }
}
```

- 加密签名
  伪造 Referer:可以使用加密签名解决
  使用第三方模块 `HttpaccesskeyModule` 实现 Nginx防盗链

```
accesskey on off模块开关
accesskey_hashmethod md5 | sha-1签名加密方式
accesskey_arg GET参数名称
```

```
accesskey signature加密规则
```

```
location ~ .*\.(gifljpglpnglflvlswfrarlzips)${
    accesskey on;
    accesskey_hashmethod md5;
    accesskey_arg sign;
    accesskey_signature"pwd11111$remote_addr;  # pwd11111 加客户端 ip
}
```

```
$sign= md5('jason'. $_SERVER['remote_addr'];
echo '<img src=”./logo_new,png?sign=. $sign.'">';
```

CDN加速

CDN的工作原理

传统模式

入域名发起请求-->解析域名获取肛-->对应的服务器-->服务器响应并返

使用CDN访问

用户发起请求-->智能DNS的解析(根据IP判断地理位置、接入网类型、选择路由最短和负载最轻的服务器)->取得缓存服务器IP->把内容返回给用户(如果缓存中有)->向源站发起请求->将结果返回给用户-->将结果存入缓存服务器

场景

站点或者应用中大量静态资源的加速分发,例如:CSs,JS图片和HTML,大文件下载,直播网站等

实现

BAT等都有提供CDN服务
可用VS做4层负载均衡
可用 Nginx, Varnish, Squid, Apache Trafficserver做7层负载均衡和 cache
使用 squid反向代理,或者 Nginx等的反向代理

独立图片服务器的部署

独立的必要性

- 分担Web服务器的I/O负载将耗费资源的图片服务分离出来,提高服务器的性能和稳定性
- 能够专门对图片服务器进行优化-为图片服务设置有针对性的缓存方案,减少带宽成本,提高访问速度
- 提高网站的可扩展性-通过增加图片服务器,提高图片吞吐能力

采用独立域名

> 并非二级域名

原因

- 同一域名下浏览器的并发连接数有限制,突破浏览器连接数的限制
- 由于 cookie的原因,对缓存不利,大部分 Web cache都只缓存不带 cookie的请求,导致每次的图片请求都不能命中 cache

独立后的问题

如何进行图片上传和图片同步

NFS共享方式
利用FTP同步(php 可以操作 ftp)

动态语言静态化

什么是动态语言静态化

将现有PHP等动态语言的逻辑代码生成为静态HTML文件,用户访问动态脚本重定向到静态HTML文件的过程。
对实时性要求不高的页面

为什么要静态化

动态脚本通常会做逻辑计算和数据查询,访问量越大,服务器压力越大
访问量大时可能会造成CPU负载过高,数据库服务器压力过大

静态化的实现方式

使用模板引擎

可以使用 Smarty的缓存机制生成静态HTML缓存文件

```
$smarty-> cache dir=$RooT."/ cache";/缓存目录
$smarty-> caching=true;//是否开启缓存
$smarty-> cache_lifetime="3600";/缓存时间
$smarty-> display(string template, string cache_id[, string compile_id]]):
$smarty-> clear_all_cache();//清除所有缓存
$smarty-> clear_cache(" file.html");/清除指定的缓存
$smarty-> clear_cache( 'article.htm',$art_id);//清除同一个模板下的指定缓存号的缓存
```

利用ob系列的函数

```
ob_start():打开输出控制缓冲
ob_get_contents0:返回输出缓冲区内容
ob_clean():清空输出缓冲区
ob_end_flush0:冲刷出(送出)输出缓冲区内容并关闭缓冲
```

```
ob_start():
```

```
//输出到页面的HTML代码
...
ob_get_contents();
ob end flush();
fopen(); //写入
```

实现页面静态化,并且当内容改变时,主动缓存新内容,且如果有$_ GET参数时候,带参数的静态化页面

```
<?php
$id = $_GET['id'];
if (empty($id)) {
    $id = '';
}
$cache_name = md5(__FILE__) . '-' . $id . '.html';
$cache_lifetime = 3600;

if (@filectime(__FILE__) <= @filectime($cache_name) && file_exists($cache_name) && $
cache_lifetime+@filectime($cache_name) > time()) {
    include $cache_name;
    exit;
}
ob_start();
?>

<b>This is  My script <?php echo $id; ?></b>

<?php
$content = ob_get_contents();
ob_end_flush();
file_put_contents($cache_name, $content);
?>
```

动态语言的并发处理

什么是进程、线程、协程

进程(Process)

是计算机中的程序关于某数据集合上的一次运行活动,是系统进行资源分配和调度的基本单位,是操作系统结构的基础,进程是一个“执行中的程序

进程的三态模型:多道程序系统中,进程在处理器上交替运行,状态不断地发生变化

- 运行
  运行:当一个进程在处理机上运行时,则称该进程处于运行状态。处于此状态的进程的数目小于等于处理器的数目,对于单处理机系统,处于运行状态的进程只有一个。在没有其他进程可以执行时(如所有进程都在阻塞状态),通常会自动执行系统的空闲进程。
- 就绪
  就绪:当一个进程获得了除处理机以外的一切所需资源,一旦得到处理机即可运行,则称此进程处于就绪状态。就绪进程可以按多个优先级来划分队列。例如,当一个进程由于时间片用完而进入就绪状态时,排入低优先级队列;当进程由I/O操作完成而进入就绪状态时,排入高优先级队列。
- 阻塞
  阻塞:也称为等待或睡眠状态,一个进程正在等待某一事件发生(例如请求Io而等待I/o完成等)而暂时停止运行,这时即使把处理机分配给进程也无法运行,故称该进程处于阻塞状态。

进程的五态模型:对于一个实际的系统,进程的状态及其转换更为复杂
新建态、活跃就绪/静止就绪、运行、活跃阻塞/静止阻塞、终止态

线程

线程,有时被称为轻量级进程 (Lightweight Process,LWP),是程序执行流的最小单元。

线程是进程中的一个实体,是被系统独立调度和分派的基本单位,线程自己不拥有系统资源,只拥有一点儿在运行中必不可少的资源但它可与同属一个进程的其它线程共享进程所拥有的全部资源。

一个线程可以刨建和撤消另一个线程,同一进程中的多个线程之间可以并发执行。

线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元,是系统独立调度和分派cPU的基本单位指运行中的程序的调度单位。

在单个程序中同时运行多个线程完成不同的工作,称为多线程。

每一个程序都至少有一个线程,若程序只有一个线程,那就是程序本身。

线程的状态:就绪、阻塞、运行

协程

协程是一种用户态的轻量级线程,协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时,将寄存器上下文和栈保存到其他地方,在切回来的时候,恢复先前保存的寄存器上下文和栈,直接操作栈则基本没有内核切换的开销,可以不加锁的访问全局变量,所以上下文的切换非常快。

线程与进程的区别

1. 线程是进程内的一个执行单元,进程内至少有一个线程,它们共享进程的地址空间,而进程有自己独立的地址空间
2. 进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源
3. 线程是处理器调度的基本单位但进程不是
4. 二者均可并发执行
5. 每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口,但是线程不能够独立执行,必须依存在应用程序中,由应用程序提供多个线程执行控制

线程与协程的区别

1. 一个线程可以多个协程,一个进程也可以单独拥有多个协程
2. 线程进程都是同步机制,而协程则是异步
3. 协程能保留上一次调用时的状态,每次过程重入时,就相当于进入上一次调用的状态

什么是多进程、多线程

多进程

同一个时间里,同一个计算机系统中如果允许两个或两个以上的进程处于运行状态,这就是多进程(如同时听歌,玩游戏)
多开一个进程,多分配一份资源,进程间通讯不方便

多线程

线程就是把一个进程分为很多片,每一片都可以是一个独立的流程
与多进程的区别是只会使用一个进程的资源,线程间可以直接通信

同步阻塞模型

多进程

最早的服务器端程序都是通过多进程、多线程来解决并发IO的问题一个请求创建一个进程,然后子进程进入循环同步堵塞地与客户端连接进行交互,收发处理数据

多线程

用多线程模式实现非常简单,线程中可以直接向某一个客户端连接发送数据

缺点

这种模型严重依赖进程的数量解决并发问题
启动大量进程会带来额外的进程调度消耗

异步非阻塞模型

- 现在各种高并发异步IO的服务器程序都是基于epol!实现的
- Io复用异步非阻塞程序使用经典的 Reactor模型, Reactor顾名思义就是反应堆的意思,它本身不处理任何数据收发。只是可以监视一个 socket句柄的事件变化

Reactor有4个核心的操作

1. add添加 socket监听到 reactor
2. set修改事件监听,可以设置监听的类型,如可读、可写
3. de从 reactor中移除,不再监听事件
4. callback,事件发生后对应的处理逻辑,一般在add/set时制定

Nginx:多线程 Reactor
Swoole:多线程 Reactor+多进程 Worker

PHP并发编程实践

PHP的 Swoole扩展

PHP的异步、并行、高性能网络通信引擎,使用纯C语言编写,提供了PHP语言的异步多线程服务器,异步TcP/UDP网络客户端,异步 MYSQL,异步 Redis,数据库连接池, Asynctask,消息队列,毫秒定时器,异步文件读写,异步DNS查询

除了异步Io的支持之外, Swoole为PHP多进程的模式设计了多个并发数据结构和IPC通信机制,可以大大简化多进程并发编程的工作。

Swoole2.0支持了类似Go语言的协程,可以使用完全同步的代码实现异步程序

Swoole的异步 MYSQL实现

```
$db =  new Swoole\MYSQL;
$server =  array('host'=>",'user=>'','password'=>",database=>",);
$db->connect($server, function(Sdb, Result){
    $db->query ("show tables", function(Swoole\MYSQL $db, $result){
        //do some thing
    });
});
```

并发处理

消息队列

- 场景说明
  用户注册后,需要发注册邮件和注册短信
- 串行方式
  将注册信息写入数据库成功后,发送注册邮件,再发送注册短信
- 并行方式
  将注册信息写入数据库成功后,发送注册郎件的同时发送注册短信
- 消息队列方式
  将注册信息写入数据库成功后,将成功信息写入队列,此时直接返回成功给用户,写入队列的时间非常短,可以忽略不计,然后异步发送邮件和短信

应用解耦

- 场景说明
  用户下单后,订单系统需要通知库存系统。
  假如库存系统无法访问,则订单减库存将失败,从而导致订单失败订单系统与库存系统耦合
- 引用队列
  用户下单后,订单系统完成持久化处理,将消息写入消息队列,返回用户订单下单成功,订阅下单的消息,采用拉/推的方式,获取下单信息,库存系统根据下单信息,进行库存操作

流量削锋

- 应用场景
  秒杀活动,流量瞬时激增,服务器压力大。
  用户发起请求,服务器接收后,先写入消息队列。假如消息队列长度超过最大值,则直接报错或提示用户
  后续程序读取消息队列再做处理
  控制请求量
  缓解高流量

日志处理

- 应用场景
  解决大量日志的传输,日志采集程序将程序写入消息队列,然后通过日志处理程序的订阅消费日志

消息通讯

- 应用场景
  聊天室
  多个客户端订阅同一主题,进行消息发布和接收

常见消息队列产品

Kafka、 Activemq、 Zeros、 Rabbitmq、 Redis等

接口的并发请求

`curl_multi_init`

php 端可同时调用多个接口

数据库缓存

简介

MysαL等一些常见的关系型数据库的数据都存储在磁盘当中,在高并发场景下,业务应用对 MYSQL产生的增、删、改、查的操作造成巨大的I/O开销和查询压力,这无疑对数据库和服务器都是一种巨大的压力,为了解决此类问题,缓存数据的概念应运而生。

优点

极大地解决数据库服务器的压力
提高应用数据的响应速度

常见的缓存形式

内存缓存,文件缓存

启用 MYSQL查询缓存

极大地降低CPU使用率

`query_cache_ type`

查询缓存类型,有0、1、2三个取值。

- 0则不使用查询缓存。
- 1表示始终使用查询缓存。

```
//对某一条不进行缓存
SELECT SQL_NO_ CACHE* FROM my_table WHERE condition
```

- 2表示按需使用查询缓存。

```
//在需要缓存时,添加SQL_CACHE
SELECT SQL_CACHE * FROM my_table WHERE condition;
```

`query_ cache_size`

默认情况下 query_cache_size为0,表示为查询缓存预留的内存为0,无法使用查询缓存
`SET GLOBAL query cache size =134217728;`

注意事项

查询缓存可以看做是SQL文本和查询结果的映射
第二次查询的SQL和第一次查询的SQL完全相同,则会使用缓存
`SHOW STATUS LIKE 'Qcache_hits;` 查看命中次数
表的结构或数据发生改变时,查询缓存中的数据不再有效

清理缓存

`FLUSH QUERY CACHE;` ||/清理查询缓存内存碎片
`RESET QUERY CACHE;` /从查询缓存中移出所有查询
`FLUSH TABLES;` //关闭所有打开的表,同时该操作将会清空查询缓存中的内容

使用 Memcache缓存查询数据

对于大型站点,如果没有中间缓存层,当流量打入数据库层时,即便有之前的几层为我们挡住一部分流量,但是在大并发的情况下,还是会有大量请求涌入数据库层,这样对于数据库服务器的压力冲击很大,响应速度也会下降,因此添加中间缓存层很有必要。

工作原理

Memcache是一个高性能的分布式的内存对象缓存系统,通过在内存里维护一个统一的巨大的hash表,它能够用来存储各种格式的数据,包括图像、视频、文件以及数据库检索的结果等。简单的说就是将数据调用到内存,然后从内存中读取,从而大大提高读取速度

方法

获取: `get(key)`
设置: `set(key,val, expire)`
删除: `delete(key)`

使用 Redis缓存查询数据

与 Memcache的区别

- 性能相差不大
- Redis在2.0版本后增加了自己的VM特性,突破物理内存的限制,Memcache可以修改最大可用内存采用LRU算法
- Redis,依赖客户端来实现分布式读写
- Memcache本身没有数据冗余机制
- Redis支持(快照、AOF),依赖快照进行持久化,aof增强了可靠性的同时,对性能有所影响
- Memcache不支持持久化,通常做缓存,提升性能;
- Memcache在并发场景下,用cas保证一致性, redis事务支持比较弱,只能保证事务中的每个操作连续执行较弱,只能保证事务中的每个操作连续执行
- Redis支持多种类的数据类型
- Redis用于数据量较小的高性能操作和运算上
- Memcache用于在动态系统中减少数据库负载,提升性能;适合做缓存,提高性能

Web服务器的负载均衡-ngnix 反向 代理

七层负载均衡的实现

基于URL等应用层信息的负载均衡
Nginx的 proxy是它一个很强大的功能,实现了7层负载均衡

Nginx负载均衡

内置策略、扩展策略
内置策略: IP Hash、加权轮询
扩展策略:fair策略、通用hash、一致性hash

内置策略

- 加权轮询策略
  首先将请求都分给高权重的机器,直到该机器的权值降到了比其他机器低,才开始将请求分给下一个高权重的机器
  当所有后端机器都down掉时, Nginx会立即将所有机器的标志位清成初始状态,以避免造成所有的机器都处在 timeout的状态
- IP Hash
  Nginx内置的另一个负载均衡的策略,流程和轮询很类似,只是其中的算法和具体的策略有些变化
  IP Hash算法是一种变相的轮询算法

扩展策略

- fair策略
  根据后端服务器的响应时间判断负载情况,从中选出负载最轻的机器进行分流
- 通用Hash、一致性Hash策略
  通用hash比较简单,可以以 Nginx内置的变量为key进行hash,致性hash采用了 Nginx内置的一致性hash环,支持 memcache

```
Nginx配置
http{
    upstream imooc_cluster {
```

```
        server 121.42.68.3:8001 weight=10;// 加权中
        server 121.42.69.3:8002 weight=9;
        #server 121.42.68.3:8003;
        #server 121.42.68.4;
    }
    server {
        listen 80;
        location / {
            proxy_pass http: //imooc_cluster;
        }
    }
}
```

四层负载均衡的实现

通过报文中的目标地址和端口,再加上负载均衡设备设置的服务器选择方式,决定最终选择的内部服务器

LVS实现服务器集群负载均衡有三种方式,NAT,DR和TUN