## 1.算法相关

#### 1.1快排

以下是一种场景，不会破坏原有的数组顺序，也是可以的；

在原有数组的基础上排序需要改写函数，传入头尾“指针”以及“数组地址”。

function quick\_sort($arr)  
{  
//判断参数是否是一个数组  
if(!is\_array($arr)) return false;  
//递归出口:数组长度为1，直接返回数组  
$length=count($arr);  
if($length<=1) return $arr;  
//数组元素有多个,则定义两个空数组  
$left=$right=array();  
//使用for循环进行遍历，把第一个元素当做比较的对象  
for($i=1;$i<$length;$i++)  
{  
//判断当前元素的大小  
if($arr[$i]<$arr[0]){  
$left[]=$arr[$i];  
}else{  
$right[]=$arr[$i];  
}  
}  
//递归调用  
$left=quick\_sort($left);  
$right=quick\_sort($right);  
//将所有的结果合并  
return array\_merge($left,array($arr[0]),$right);

#### 1.2二分查找

function getValue4($num,$arr,$start = 0,$end = 100){  
//采用二分法查找  
$middle = floor(($end + $start) / 2);  
  
//判断  
if($arr[$middle] == $num){  
//已经找到了，递归的出口  
return $middle + 1;  
}elseif($arr[$middle] < $num){  
//要查找的元素在数组的后半段  
$start = $middle + 1;  
//边界值  
if($start >= $end){  
//没有找到，但是已经超出边界值，递归出口  
return false;  
}  
//调用自己去查找：递归点  
return getValue4($num,$arr,$start,$end); //getValue4($num,$arr,51,100)  
}else{  
//要查找的元素在数组的前半段  
$end = $middle - 1;  
//判断边界值  
if($end < 0)return false;  
  
//调用自己：递归点  
return getValue4($num,$arr,$start,$end); //getValue4($num,$arr,0,49)  
}  
  
//都没有找到  
return false;  
}

#### 1.3桶排序了解一下

#### 1.4堆排序了解一下

#### 1.5最短路径优化

#### 推荐看一下《啊哈，算法》快速成型，两小时基本能搞定大部分问题。

https://blog.csdn.net/ahalei/article/list/2

## 2.mysql相关

#### 2.1sql注入

SQL注入，就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。具体来说，它是利用现有应用程序，将（恶意的）SQL命令注入到后台数据库引擎执行的能力，它可以通过在Web表单中输入（恶意）SQL语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行SQL语句。

#### 2.2mysql存储引擎（b+树了解一下）

myisam属于堆表，数据写入一直累积(concurrent\_insert参数设置为2);此时写入性能比innodb好，但是无论是主键查询还是非主键查询，都不可避免的需要二次io(除非能索引内完成字段记录返回)

innodb属于聚簇索引组织表(主键索引即数据),由于必须保证索引有序，在写入时需要找到合适的位置进行插入，有时候还会导致二叉树调整所以写入性能比myisam要差；但是主键查找时只需要一次io即可返回数据，非主键索引查询时如果在索引内不能完成查询记录返回则需要第二次的io

Myisam与innodb引擎,索引文件的区别：

innodb的次索引指向对主键的引用。

myisam的次索引和主索引都指向物理行。

注意: innodb来说,

1: 主键索引 既存储索引值,又在叶子中存储行的数据

2: 如果没有主键, 则会Unique key做主键

3: 如果没有unique,则系统生成一个内部的rowid做主键.

4: 像innodb中,主键的索引结构中,既存储了主键值,又存储了行数据,这种结构称为"聚簇索引"

聚簇索引

优势: 根据主键查询条目比较少时,不用回行(数据就在主键节点下)

劣势: 如高性能索引策略

对于innodb而言,因为节点下有数据文件,因此节点的分裂将会比较慢.

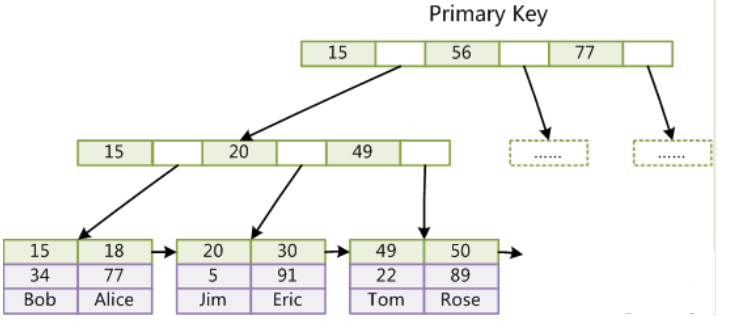
对于innodb的主键,尽量用整型,而且是递增的整型.

如果是无规律的数据,将会产生的页的分裂,影响速度.果碰到不规则数据插入时,造成频繁的页分裂.

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。

虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好做法，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

1）MyISAM：  
每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。  
MyISAM表格可以被压缩，而且它们支持全文搜索。不支持事务，而且也不支持外键。如果事物回滚将造成不完全回滚，不具有原子性。在进行updata时进行表锁，并发量相对较小。如果执行大量的SELECT，MyISAM是更好的选择。  
MyISAM的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而Innodb是索引和数据是紧密捆绑的，没有使用压缩从而会造成Innodb比MyISAM体积庞大不小  
MyISAM缓存在内存的是索引，不是数据。而InnoDB缓存在内存的是数据，相对来说，服务器内存越大，InnoDB发挥的优势越大。

优点：查询数据相对较快，适合大量的select，可以全文索引。  
缺点：不支持事务，不支持外键，并发量较小，不适合大量update

2）InnoDB：（参数说明：[Mysql存储引擎之Innodb重要参数说明](http://www.cnblogs.com/kevingrace/p/6133818.html)）  
这种类型是事务安全的。.它与BDB类型具有相同的特性,它们还支持外键。InnoDB表格速度很快。具有比BDB还丰富的特性,因此如果需要一个事务安全的存储引擎，建议使用它。在update时表进行行锁，并发量相对较大。如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。  
优点：支持事务，支持外键，并发量较大，适合大量update  
缺点：查询数据相对较快，不适合大量的select  
对于支持事物的InnoDB类型的表，影响速度的主要原因是AUTOCOMMIT默认设置是打开的，而且程序没有显式调用BEGIN 开始事务，导致每插入一条都自动Commit，严重影响了速度。可以在执行sql前调用begin，多条sql形成一个事物（即使autocommit打开也可以），将大大提高性能。

3）memory，表存在内存里面，可以用hash索引

#### 2.3Mysql数据事务隔离级别

一、事务的基本要素（ACID）

1、原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位。

2、一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如A向B转账，不可能A扣了钱，B却没收到。

3、隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

4、持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

#### 2.4事务隔离级别

1）read uncommitted : 读取尚未提交的数据 ：哪个问题都不能解决

2）read committed：读取已经提交的数据 ：可以解决脏读 ---- oracle默认的

3）repeatable read：重读读取：可以解决脏读 和 不可重复读 ---mysql默认的

4）serializable：串行化：可以解决 脏读 不可重复读 和 幻读---相当于锁表

**1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据**

**2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。**

**3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。**

**小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表**

#### 2.5MySQL优化

优化方案：

SQL和索引 > 数据库表结构 > 系统配置 > 硬件   
但成本从低到高。

<https://blog.csdn.net/sinat_23080035/article/details/52802569>

很详细。

#### 2.6MySQL防注入

应对方法：

1. mysql\_escape\_string() 转义特殊字符（(PHP 4 >= 4.3.0, PHP 5)）(mysql\_real\_escape\_string必须先链接上数据库，否则会报错)

下列字符受影响：

\x00 //对应于ascii字符的NULL

\n //换行符且回到下一行的最前端

\r //换行符

\ //转义符

'

"

\x1a //16进制数

如果成功，则该函数返回被转义的字符串。如果失败，则返回 false。

1. addslashes(): 函数返回在预定义字符之前添加反斜杠的字符串 (stripslashes()实现字符串还原)

预定义的字符有：

单引号（'）

双引号（"）

反斜杠（\）

NULL

1. prepared  statements(预处理机制)

<?php

$mysqli = new mysqli("example.com", "user", "password", "database");

if ($mysqli->connect\_errno) {

echo "Failed to connect to MySQL: (" . $mysqli->connect\_errno . ") " . $mysqli->connect\_error;

}

/\* Non-prepared statement \*/

if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {

echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;

}

/\* Prepared statement, stage 1: prepare \*/

if (!($stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)"))) {

echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;

}

/\* Prepared statement, stage 2: bind and execute \*/

$id = 1;

if (!$stmt->bind\_param("i", $id)) {

echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;

}

if (!$stmt->execute()) {

echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;

}

?>

#### 2.7索引原理及优化慢查询案例

<https://www.cnblogs.com/bypp/p/7755307.html>

（这个讲的很好啊，建议看明白）

不幸碰见这种题，就算哔了狗吧。内容很多，主要是索引的原理。

很可能会问不同数据库之间的区别。

#### 2.8大量数据查询处理案例

1、应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。   
  
2、对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。   
  
3、应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：   
select id from t where num is null   
可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：   
select id from t where num=0   
  
4、尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：   
select id from t where num=10 or num=20   
可以这样查询：   
select id from t where num=10   
union all   
select id from t where num=20   
  
5、下面的查询也将导致全表扫描：(不能前置百分号)   
select id from t where name like ‘%c%’   
若要提高效率，可以考虑全文检索。   
  
6、in 和 not in 也要慎用，否则会导致全表扫描，如：   
select id from t where num in(1,2,3)   
对于连续的数值，能用 between 就不要用 in 了：   
select id from t where num between 1 and 3   
  
7、如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：   
select id from t where num=@num   
可以改为强制查询使用索引：   
select id from t with(index(索引名)) where num=@num   
  
8、应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
select id from t where num/2=100   
应改为:   
select id from t where num=100\*2   
  
9、应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
select id from t where substring(name,1,3)=’abc’–name以abc开头的id   
select id from t where datediff(day,createdate,’2005-11-30′)=0–’2005-11-30′生成的id   
应改为:   
select id from t where name like ‘abc%’   
select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′   
  
10、不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。   
  
11、在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使 用，并且应尽可能的让字段顺序与索引顺序相一致。   
  
12、不要写一些没有意义的查询，如需要生成一个空表结构：   
select col1,col2 into #t from t where 1=0   
这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：   
create table #t(…)   
  
13、很多时候用 exists 代替 in 是一个好的选择：   
select num from a where num in(select num from b)   
用下面的语句替换：   
select num from a where exists(select 1 from b where num=a.num)   
  
14、并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。   
  
15、索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。   
  
16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。   
  
17、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会 逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。   
  
18、尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。   
  
19、任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。  
  
20、尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。   
  
21、避免频繁创建和删除临时表，以减少系统表资源的消耗。   
  
22、临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使 用导出表。   
  
23、在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。   
  
24、如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。   
  
25、尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。   
  
26、使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。   
  
27、与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。   
  
28、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。   
  
29、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。   
  
30、尽量避免大事务操作，提高系统并发能力。

## 3.Redis相关

#### 3.1数据类型：

字符串，链表，哈希，集合，有序集合

注意不同场景下的使用数据类型的选择。

#### 3.2Redis单线程为什么快？

1、完全基于内存

Redis是纯内存数据库，相对于读写磁盘，读写内存的速度就不是几倍几十倍了，一般，hash查找可以达到每秒百万次的数量级。

2、多路复用IO

“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗）。可以直接理解为：单线程的原子操作，避免上下文切换的时间和性能消耗；加上对内存中数据的处理速度，很自然的提高redis的吞吐量。

这里很可能衍生多路复用io是啥？

#### 3.3redis和memcached的区别（总结）

观点一：

1、Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等；

2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储；

3、虚拟内存--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘；

4、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10；

5、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从；

6、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）；

7、灾难恢复--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复；

8、Redis支持数据的备份，即master-slave模式的数据备份；

#### 3.4Redis怎么实现持久化

Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化），另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

RDB存在哪些优势呢？

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢？

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF的优势有哪些呢？

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

AOF的劣势有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。

4、常用配置

RDB持久化配置

Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

AOF持久化配置

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no #从不同步。高效但是数据不会被持久化。

#### 3.5Redis主从同步

https://www.cnblogs.com/lukexwang/p/4711977.html

#### 3.6怎么去设计一个可配置的访问权限控制，利用Redis

## 4.Php相关

[PHP 底层的运行机制与原理](https://www.cnblogs.com/terryguan/p/6691855.html)

PHP说简单，但是要精通也不是一件简单的事。我们除了会使用之外，还得知道它底层的工作原理。

PHP是一种适用于web开发的动态语言。具体点说，就是一个用C语言实现包含大量组件的软件框架。更狭义点看，可以把它认为是一个强大的UI框架。

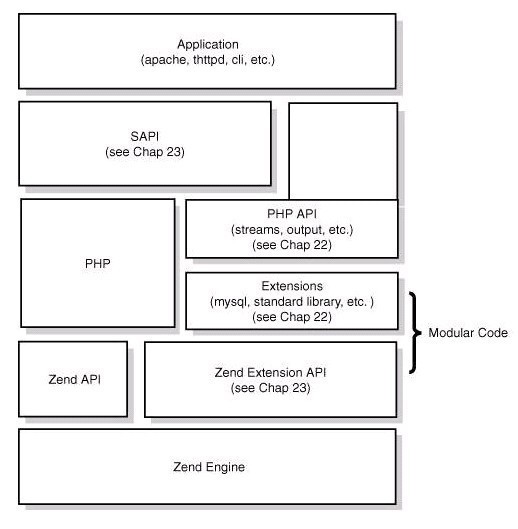
了解PHP底层实现的目的是什么？动态语言要像用好首先得了解它，内存管理、框架模型值得我们借鉴，通过扩展开发实现更多更强大的功能，优化我们程序的性能。

**1. PHP的设计理念及特点**

* 多进程模型：由于PHP是多进程模型，不同请求间互不干涉，这样保证了一个请求挂掉不会对全盘服务造成影响，当然，随着时代发展，PHP也早已支持多线程模型。
* 弱类型语言：和C/C++、Java、C#等语言不同，PHP是一门弱类型语言。一个变量的类型并不是一开始就确定不变，运行中才会确定并可能发生隐式或显式的类型转换，这种机制的灵活性在web开发中非常方便、高效，具体会在后面PHP变量中详述。
* 引擎(Zend)+组件(ext)的模式降低内部耦合。
* 中间层(sapi)隔绝web server和PHP。
* 语法简单灵活，没有太多规范。缺点导致风格混杂，但再差的程序员也不会写出太离谱危害全局的程序。

**2. PHP的四层体系**

PHP的核心架构如下图：



从图上可以看出，PHP从下到上是一个4层体系：

* Zend引擎：Zend整体用纯C实现，是PHP的内核部分，它将PHP代码翻译（词法、语法解析等一系列编译过程）为可执行opcode的处理并实现相应的处理方法、实现了基本的数据结构（如hashtable、oo）、内存分配及管理、提供了相应的api方法供外部调用，是一切的核心，所有的外围功能均围绕Zend实现。
* Extensions：围绕着Zend引擎，extensions通过组件式的方式提供各种基础服务，我们常见的各种内置函数（如array系列）、标准库等都是通过extension来实现，用户也可以根据需要实现自己的extension以达到功能扩展、性能优化等目的（如贴吧正在使用的PHP中间层、富文本解析就是extension的典型应用）。
* Sapi：Sapi全称是Server Application Programming Interface，也就是服务端应用编程接口，Sapi通过一系列钩子函数，使得PHP可以和外围交互数据，这是PHP非常优雅和成功的一个设计，通过sapi成功的将PHP本身和上层应用解耦隔离，PHP可以不再考虑如何针对不同应用进行兼容，而应用本身也可以针对自己的特点实现不同的处理方式。
* 上层应用：这就是我们平时编写的PHP程序，通过不同的sapi方式得到各种各样的应用模式，如通过webserver实现web应用、在命令行下以脚本方式运行等等。

如果PHP是一辆车，那么车的框架就是PHP本身，Zend是车的引擎（发动机），Ext下面的各种组件就是车的轮子，Sapi可以看做是公路，车可以跑在不同类型的公路上，而一次PHP程序的执行就是汽车跑在公路上。因此，我们需要：性能优异的引擎+合适的车轮+正确的跑道。

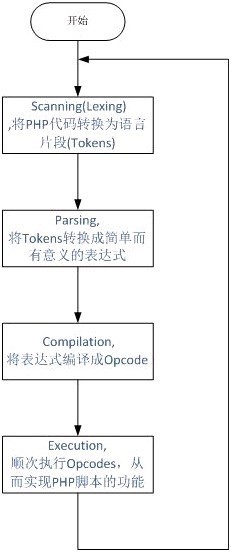
**3. Sapi**

如前所述，Sapi通过通过一系列的接口，使得外部应用可以和PHP交换数据并可以根据不同应用特点实现特定的处理方法，我们常见的一些sapi有：

* apache2handler：这是以apache作为webserver，采用mod\_PHP模式运行时候的处理方式，也是现在应用最广泛的一种。
* cgi：这是webserver和PHP直接的另一种交互方式，也就是大名鼎鼎的fastcgi协议，在最近今年fastcgi+PHP得到越来越多的应用，也是异步webserver所唯一支持的方式。
* cli：命令行调用的应用模式

**4. PHP的执行流程&opcode**

我们先来看看PHP代码的执行所经过的流程。



从图上可以看到，PHP实现了一个典型的动态语言执行过程：拿到一段代码后，经过词法解析、语法解析等阶段后，源程序会被翻译成一个个指令(opcodes)，然后ZEND虚拟机顺次执行这些指令完成操作。PHP本身是用C实现的，因此最终调用的也都是C的函数，实际上，我们可以把PHP看做是一个C开发的软件。

PHP的执行的核心是翻译出来的一条一条指令，也即opcode。

Opcode是PHP程序执行的最基本单位。一个opcode由两个参数(op1,op2)、返回值和处理函数组成。PHP程序最终被翻译为一组opcode处理函数的顺序执行。

常见的几个处理函数：

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ZEND\_ASSIGN\_SPEC\_CV\_CV\_HANDLER : 变量分配 （$a=$b）  ZEND\_DO\_FCALL\_BY\_NAME\_SPEC\_HANDLER：函数调用  ZEND\_CONCAT\_SPEC\_CV\_CV\_HANDLER：字符串拼接 $a.$b  ZEND\_ADD\_SPEC\_CV\_CONST\_HANDLER: 加法运算 $a+2  ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST：判断相等 $a==1  ZEND\_IS\_IDENTICAL\_SPEC\_CV\_CONST：判断相等 $a===1 |

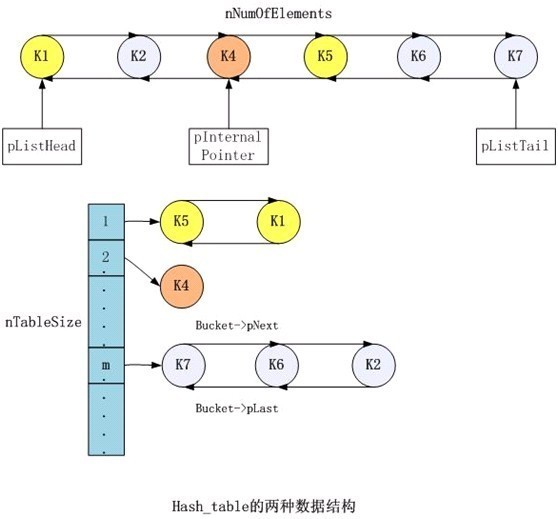
**5. HashTable — 核心数据结构**

HashTable是zend的核心数据结构，在PHP里面几乎并用来实现所有常见功能，我们知道的PHP数组即是其典型应用，此外，在zend内部，如函数符号表、全局变量等也都是基于hash table来实现。

PHP的hash table具有如下特点：

* 支持典型的key->value查询
* 可以当做数组使用
* 添加、删除节点是O（1）复杂度
* key支持混合类型：同时存在关联数组合索引数组
* Value支持混合类型：array (“string”,2332)
* 支持线性遍历：如foreach

Zend hash table实现了典型的hash表散列结构，同时通过附加一个双向链表，提供了正向、反向遍历数组的功能。其结构如下图：



可以看到，在hash table中既有key->value形式的散列结构，也有双向链表模式，使得它能够非常方便的支持快速查找和线性遍历。

* 散列结构：Zend的散列结构是典型的hash表模型，通过链表的方式来解决冲突。需要注意的是zend的hash table是一个自增长的数据结构，当hash表数目满了之后，其本身会动态以2倍的方式扩容并重新元素位置。初始大小均为8。另外，在进行key->value快速查找时候，zend本身还做了一些优化，通过空间换时间的方式加快速度。比如在每个元素中都会用一个变量nKeyLength标识key的长度以作快速判定。
* 双向链表：Zend hash table通过一个链表结构，实现了元素的线性遍历。理论上，做遍历使用单向链表就够了，之所以使用双向链表，主要目的是为了快速删除，避免遍历。Zend hash table是一种复合型的结构，作为数组使用时，即支持常见的关联数组也能够作为顺序索引数字来使用，甚至允许2者的混合。
* PHP关联数组：关联数组是典型的hash\_table应用。一次查询过程经过如下几步（从代码可以看出，这是一个常见的hash查询过程并增加一些快速判定加速查找。）：

PHP

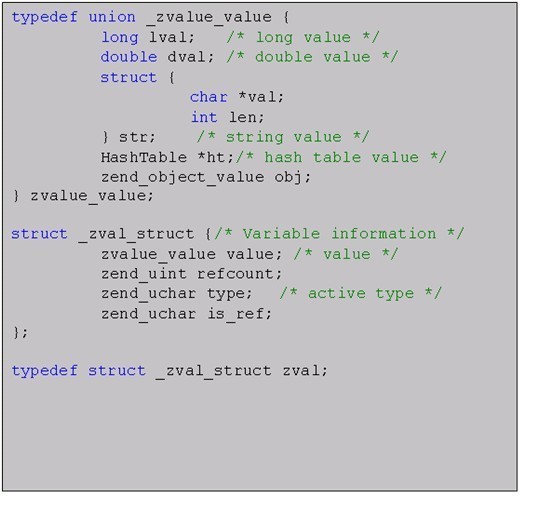
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | getKeyHashValue h;  index = n & nTableMask;  Bucket \*p = arBucket[index];  while (p) {  if ((p->h == h) & (p->nKeyLength == nKeyLength)) {  RETURN p->data;  }  p=p->next;  }  RETURN FALTURE; |

* PHP索引数组：索引数组就是我们常见的数组，通过下标访问。例如 $arr[0]，Zend HashTable内部进行了归一化处理，对于index类型key同样分配了hash值和nKeyLength(为0)。内部成员变量nNextFreeElement就是当前分配到的最大id，每次push后自动加一。正是这种归一化处理，PHP才能够实现关联和非关联的混合。由于push操作的特殊性，索引key在PHP数组中先后顺序并不是通过下标大小来决定，而是由push的先后决定。例如 $arr[1] = 2; $arr[2] = 3;对于double类型的key，Zend HashTable会将他当做索引key处理

**6. PHP变量**

PHP是一门弱类型语言，本身不严格区分变量的类型。PHP在变量申明的时候不需要指定类型。PHP在程序运行期间可能进行变量类型的隐示转换。和其他强类型语言一样，程序中也可以进行显示的类型转换。PHP变量可以分为简单类型(int、string、bool)、集合类型(array resource object)和常量(const)。以上所有的变量在底层都是同一种结构 zval。

Zval是zend中另一个非常重要的数据结构，用来标识并实现PHP变量，其数据结构如下：



Zval主要由三部分组成：

* type：指定了变量所述的类型（整数、字符串、数组等）
* refcount&is\_ref：用来实现引用计数(后面具体介绍)
* value：核心部分，存储了变量的实际数据

Zvalue是用来保存一个变量的实际数据。因为要存储多种类型，所以zvalue是一个union，也由此实现了弱类型。

PHP变量类型和其实际存储对应关系如下：

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5 | IS\_LONG -> lvalue  IS\_DOUBLE -> dvalue  IS\_ARRAY -> ht  IS\_STRING -> str  IS\_RESOURCE -> lvalue |

引用计数在内存回收、字符串操作等地方使用非常广泛。PHP中的变量就是引用计数的典型应用。Zval的引用计数通过成员变量is\_ref和ref\_count实现，通过引用计数，多个变量可以共享同一份数据。避免频繁拷贝带来的大量消耗。

在进行赋值操作时，zend将变量指向相同的zval同时ref\_count++，在unset操作时，对应的ref\_count-1。只有ref\_count减为0时才会真正执行销毁操作。如果是引用赋值，则zend会修改is\_ref为1。

PHP变量通过引用计数实现变量共享数据，那如果改变其中一个变量值呢？当试图写入一个变量时，Zend若发现该变量指向的zval被多个变量共享，则为其复制一份ref\_count为1的zval，并递减原zval的refcount，这个过程称为“zval分离”。可见，只有在有写操作发生时zend才进行拷贝操作，因此也叫copy-on-write(写时拷贝)

对于引用型变量，其要求和非引用型相反，引用赋值的变量间必须是捆绑的，修改一个变量就修改了所有捆绑变量。

整数、浮点数是PHP中的基础类型之一，也是一个简单型变量。对于整数和浮点数，在zvalue中直接存储对应的值。其类型分别是long和double。

从zvalue结构中可以看出，对于整数类型，和c等强类型语言不同，PHP是不区分int、unsigned int、long、long long等类型的，对它来说，整数只有一种类型也就是long。由此，可以看出，在PHP里面，整数的取值范围是由编译器位数来决定而不是固定不变的。

对于浮点数，类似整数，它也不区分float和double而是统一只有double一种类型。

在PHP中，如果整数范围越界了怎么办？这种情况下会自动转换为double类型，这个一定要小心，很多trick都是由此产生。

和整数一样，字符变量也是PHP中的基础类型和简单型变量。通过zvalue结构可以看出，在PHP中，字符串是由由指向实际数据的指针和长度结构体组成，这点和c++中的string比较类似。由于通过一个实际变量表示长度，和c不同，它的字符串可以是2进制数据（包含），同时在PHP中，求字符串长度strlen是O(1)操作。

在新增、修改、追加字符串操作时，PHP都会重新分配内存生成新的字符串。最后，出于安全考虑，PHP在生成一个字符串时末尾仍然会添加

常见的字符串拼接方式及速度比较：

假设有如下4个变量：$strA=‘123’; $strB = ‘456’; $intA=123; intB=456;

现在对如下的几种字符串拼接方式做一个比较和说明：

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $res = $strA.$strB和$res = “$strA$strB”  这种情况下，zend会重新malloc一块内存并进行相应处理，其速度一般  $strA = $strA.$strB  这种是速度最快的，zend会在当前strA基础上直接relloc，避免重复拷贝  $res = $intA.$intB  这种速度较慢，因为需要做隐式的格式转换，实际编写程序中也应该注意尽量避免  $strA = sprintf (“%s%s”,$strA.$strB);  这会是最慢的一种方式，因为sprintf在PHP中并不是一个语言结构，本身对于格式识别和处理就需要耗费比较多时间，另外本身机制也是malloc。不过sprintf的方式最具可读性，实际中可以根据具体情况灵活选择。 |

PHP的数组通过Zend HashTable来天然实现。

foreach操作如何实现？对一个数组的foreach就是通过遍历hashtable中的双向链表完成。对于索引数组，通过foreach遍历效率比for高很多，省去了key->value的查找。count操作直接调用HashTable->NumOfElements，O(1)操作。对于’123’这样的字符串，zend会转换为其整数形式。$arr[‘123’]和$arr[123]是等价的

资源类型变量是PHP中最复杂的一种变量，也是一种复合型结构。

PHP的zval可以表示广泛的数据类型，但是对于自定义的数据类型却很难充分描述。由于没有有效的方式描绘这些复合结构，因此也没有办法对它们使用传统的操作符。要解决这个问题，只需要通过一个本质上任意的标识符（label）引用指针，这种方式被称为资源。

在zval中，对于resource，lval作为指针来使用，直接指向资源所在的地址。Resource可以是任意的复合结构，我们熟悉的mysqli、fsock、memcached等都是资源。

如何使用资源：

* 注册：对于一个自定义的数据类型，要想将它作为资源。首先需要进行注册，zend会为它分配全局唯一标示。
* 获取一个资源变量：对于资源，zend维护了一个id->实际数据的hash\_tale。对于一个resource，在zval中只记录了它的id。fetch的时候通过id在hash\_table中找到具体的值返回。
* 资源销毁：资源的数据类型是多种多样的。Zend本身没有办法销毁它。因此需要用户在注册资源的时候提供销毁函数。当unset资源时，zend调用相应的函数完成析构。同时从全局资源表中删除它。

资源可以长期驻留，不只是在所有引用它的变量超出作用域之后，甚至是在一个请求结束了并且新的请求产生之后。这些资源称为持久资源，因为它们贯通SAPI的整个生命周期持续存在，除非特意销毁。很多情况下，持久化资源可以在一定程度上提高性能。比如我们常见的mysql\_pconnect ,持久化资源通过pemalloc分配内存，这样在请求结束的时候不会释放。  
对zend来说，对两者本身并不区分。

PHP中的局部变量和全局变量是如何实现的？对于一个请求，任意时刻PHP都可以看到两个符号表(symbol\_table和active\_symbol\_table)，其中前者用来维护全局变量。后者是一个指针，指向当前活动的变量符号表，当程序进入到某个函数中时，zend就会为它分配一个符号表x同时将active\_symbol\_table指向a。通过这样的方式实现全局、局部变量的区分。

获取变量值：PHP的符号表是通过hash\_table实现的，对于每个变量都分配唯一标识，获取的时候根据标识从表中找到相应zval返回。

函数中使用全局变量：在函数中，我们可以通过显式申明global来使用全局变量。在active\_symbol\_table中创建symbol\_table中同名变量的引用，如果symbol\_table中没有同名变量则会先创建。

#### 4.1Php内存回收机制（感觉会常碰见）

https://www.cnblogs.com/impy/p/7850955.html

#### 4.2Php是怎么实现实时类型的判断

#### 4.3Php怎么启动

#### 4.4Php-fpm是做啥的

有一些工具一直在用，却从来也不知道这些东西是什么。

现在想想还是梳理一下比较好。所以看到我觉得不错的定义就抄过来了。

首先，CGI是干嘛的？CGI是为了保证web server传递过来的数据是标准格式的，方便CGI程序的编写者。

web server（比如说nginx）只是内容的分发者。比如，如果请求/index.html，那么web server会去文件系统中找到这个文件，发送给浏览器，这里分发的是静态数据。好了，如果现在请求的是/index.php，根据配置文件，nginx知道这个不是静态文件，需要去找PHP解析器来处理，那么他会把这个请求简单处理后交给PHP解析器。Nginx会传哪些数据给PHP解析器呢？url要有吧，查询字符串也得有吧，POST数据也要有，HTTP header不能少吧，好的，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。仔细想想，你在PHP代码中使用的用户从哪里来的。

当web server收到/index.php这个请求后，会启动对应的CGI程序，这里就是PHP的解析器。接下来PHP解析器会解析php.ini文件，初始化执行环境，然后处理请求，再以规定CGI规定的格式返回处理后的结果，退出进程。web server再把结果返回给浏览器。

好了，CGI是个协议，跟进程什么的没关系。那fastcgi又是什么呢？Fastcgi是用来提高CGI程序性能的。

提高性能，那么CGI程序的性能问题在哪呢？"PHP解析器会解析php.ini文件，初始化执行环境"，就是这里了。标准的CGI对每个请求都会执行这些步骤（不闲累啊！启动进程很累的说！），所以处理每个时间的时间会比较长。这明显不合理嘛！那么Fastcgi是怎么做的呢？首先，Fastcgi会先启一个master，解析配置文件，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。这样就避免了重复的劳动，效率自然是高。而且当worker不够用时，master可以根据配置预先启动几个worker等着；当然空闲worker太多时，也会停掉一些，这样就提高了性能，也节约了资源。这就是fastcgi的对进程的管理。

那PHP-FPM又是什么呢？是一个实现了Fastcgi的程序，被PHP官方收了。

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（皇上，臣妾真的做不到啊！）所以就出现了一些能够调度php-cgi进程的程序，比如说由lighthttpd分离出来的spawn-fcgi。好了PHP-FPM也是这么个东东，在长时间的发展后，逐渐得到了大家的认可（要知道，前几年大家可是抱怨PHP-FPM稳定性太差的），也越来越流行。

fastcgi是一个协议，php-fpm实现了这个协议

对。

有的说，php-fpm是fastcgi进程的管理器，用来管理fastcgi进程的

对。php-fpm的管理对象是php-cgi。但不能说php-fpm是fastcgi进程的管理器，因为前面说了fastcgi是个协议，似乎没有这么个进程存在，就算存在php-fpm也管理不了他（至少目前是）。 有的说，php-fpm是php内核的一个补丁

以前是对的。因为最开始的时候php-fpm没有包含在PHP内核里面，要使用这个功能，需要找到与源码版本相同的php-fpm对内核打补丁，然后再编译。后来PHP内核集成了PHP-FPM之后就方便多了，使用--enalbe-fpm这个编译参数即可。

有的说，修改了php.ini配置文件后，没办法平滑重启，所以就诞生了php-fpm

是的，修改php.ini之后，php-cgi进程的确是没办法平滑重启的。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。

还有的说PHP-CGI是PHP自带的FastCGI管理器，那这样的话干吗又弄个php-fpm出

不对。php-cgi只是解释PHP脚本的程序而已。

#### 4.5php有哪些魔术方法，魔术变量

从**php5以后的版本，类就可以使用魔术方法了**。php规定以两个下划线开头的方法都保留为魔术方法，所以建议大家**函数名最好不用\_\_开 头，除非是为了重载已有的魔术方法。**

**目前php已有的魔术方法有**

\_\_construct，\_\_destruct，\_\_call，\_\_get，\_\_set，\_\_isset，\_\_unset，\_\_sleep，\_\_wakeup，\_\_toString，\_\_set\_state 和 \_\_clone。

**\_\_construct和\_\_destruct**

\_\_construct和\_\_destruct是**类的构造函数和析构函数**，这个大家经常会用到，相信大家都很熟悉，这里就不多说了。

**\_\_sleep和\_\_wakeup**

\_\_sleep和\_\_wakeup是**序列化类的时候调用的**。当序列化对象时，php将试图在序列动作之前调用该对象的成员函数\_\_sleep()， 当使用unserialize() 恢复对象时， 将调用\_\_wakeup()。

**\_\_toString**

\_\_toString是**对象被转为string时调用**的**必须有返回值即:return xxx ; 否则报错**，例如

<?php

class Str

{

private $str;

public function \_\_construct($str) {

$this->str = $str;

}

public function \_\_toString() {

return $this->str;

}

}

$class = new Str('Hello');

echo $class; // 这里对象被转为了string，所以调用了\_\_toString

?>

上例将输出 Hello

**\_\_set\_state**

\_\_set\_state是当用**var\_export()来导出类的时候调用的**，这个魔术函数只有一个参数，这个参数是一个数组，用来指定 export的时候得到的属性。一般很少用到。

**\_\_call, \_\_get和\_\_set**

\_\_call, \_\_get和\_\_set 这三个魔术方法是**最常用的**，**当调用类中不存在的方法时就会调用\_\_call**，而**\_\_get和\_\_set则是访问和设置类不存在的成员变量时调用的。**   
这三个的函数原型如下：

mixed \_\_call(string $name, array $arguments)

void \_\_set(string $name, mixed $value)

mixed \_\_get(string $name)

**\_\_call的例子：**

<?php

class Caller

{

public function \_\_call( $method , $args )

{

echo "Method $method called:/n" ;

print\_r($args );

}

}

$foo = new Caller();

$foo ->test(1, 2);

?>

上例将输出：   
Method test called:   
Array   
(   
[0] => 1   
[1] => 2   
)

**\_\_get 和 \_\_set 的例子：**

<?php

class a

{

public $c = 0;

public $arr = array();

public function \_\_set($k, $v)

{

echo $k . "/n";

echo $v . "/n";

$this->arr[$k] = $v;

}

public function \_\_get($k)

{

echo "The value of $k is " . $this->arr[$k];

}

}

$a = new a;

$a->b = 1; // 成员变量b不存在，所以会调用\_\_set

$a->c = 2; // 成员变量c是存在的，所以不调用\_\_set，无任何输出

$d = $a->b; // 成员变量b不存在，所以会调用\_\_get

?>

上例将输出：   
b   
1   
The value of b is 1

**\_\_isset和\_\_unset**

\_\_isset和\_\_unset这两个与\_\_get和\_\_set其实原理是差不多的，他们的原型如下：

bool \_\_isset(string $name)

void \_\_unset(string $name)

举个例子：

<?php

class a

{

public $c = 3;

public $arr = array ( 'a' => 1, 'b' => 2);

public function \_\_isset( $k )

{

return isset( $this ->arr[ $k ]);

}

public function \_\_unset( $k )

{

unset($this ->arr[ $k ]);

}

}

$a = new a;

var\_dump(isset($a ->a)); // 成员变量a不存在，所以调用\_\_isset，返回true

var\_dump(isset($a ->c)); // 成员变量c是存在的，没有调用\_\_isset，同样返回true

unset($a ->b); // 成员变量b不存在，调用\_\_unset

var\_dump($a );

?>

上例将输出：

bool(true)

bool(true)

object(a)#1 (2) {

["c"]=>int(3)

["arr"]=>array(1) {

["a"]=>int(1)

}

}

**\_\_clone**

类复制（clone）的时候，如果有定义\_\_clone这个魔术方法就会调用它。

举例如下：

<?php

class a

{

public function \_\_clone()

{

echo "object cloned" ;

}

}

$a = new a;

$b = $a ; // $b只是$a的引用， 不是克隆，所以不调用\_\_clone，没任何输出。

$c = clone $a ; // 调用了\_\_clone，将输出 object cloned

?>

上例将输出：

object cloned

参考: <http://php.net/manual/zh/language.oop5.magic.php>

**PHP的魔术常量**

定义: 以两个\_开头和结尾的常量为魔术常量   
注意: 魔术常量不区分大小写

<?php

$file1 = \_\_FILE\_\_;

$file2 = \_\_file\_\_;

var\_dump($file1);

var\_dump($file2);

?>

结果是:   
string(36) “F:\Apache\www\temp\php\_demo\temp.php”   
string(36) “F:\Apache\www\temp\php\_demo\temp.php”

***\_LINE\_***

文件中的当前行号。

***\_FILE\_***

文件的完整路径和文件名。如果用在被包含文件中，则返回被包含的文件名。自 PHP 4.0.2 起，**FILE** 总是包含一个绝对路径（如果是符号连接，则是解析后的绝对路径），而在此之前的版本有时会包含一个相对路径。

***\_DIR\_***

文件所在的目录。如果用在被包括文件中，则返回被包括的文件所在的目录。它等价于 dirname(**FILE**)。除非是根目录，否则目录中名不包括末尾的斜杠。（PHP 5.3.0中新增）

***\_FUNCTION\_***

函数名称（PHP 4.3.0 新加）。自 PHP 5 起本常量返回该函数被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写字母的。

***\_CLASS\_***

类的名称（PHP 4.3.0 新加）。自 PHP 5 起本常量返回该类被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写字母的。类名包括其被声明的作用区域（例如 Foo\Bar）。注意自 PHP 5.4 起 **CLASS** 对 trait 也起作用。当用在 trait 方法中时，**CLASS** 是调用 trait 方法的类的名字。

***\_NAMESPACE\_***

当前命名空间的名称（区分大小写）。此常量是在编译时定义的（PHP 5.3.0 新增）。

***\_TRAIT\_***

Trait 的名字（PHP 5.4.0 新加）。自 PHP 5.4 起此常量返回 trait 被定义时的名字（区分大小写）。Trait 名包括其被声明的作用区域（例如 Foo\Bar）。

***\_METHOD\_***

类的方法名（PHP 5.0.0 新加）。返回该方法被定义时的名字（区分大小写）。

## 5.Nginx相关

#### 5.1介绍nginx的工作原理

#### 5.2 Nginx+Php-fpm运行原理详解

**一、代理与反向代理**

**现实生活中的例子**

1、正向代理：访问google.com



如上图，因为google被墙，我们需要vpn翻墙才能访问google.com。

vpn对于“我们”来说，是可以感知到的（我们连接vpn）vpn对于”google服务器”来说，是不可感知的(google只知道有http请求过来)。

**对于人来说可以感知到，但服务器感知不到的服务器，我们叫他正向代理服务器。**

2、反向代理：通过反向代理实现负载均衡



如上图，我们访问baidu.com的时候，baidu有一个代理服务器，通过这个代理服务器，可以做负载均衡，路由到不同的server。

此代理服务器,对于“我们”来说是不可感知的(我们只能感知到访问的是百度的服务器，不知道中间还有代理服务器来做负载均衡)。

此代理服务器，对于”server1 server2 server3”是可感知的(代理服务器负载均衡路由到不同的server)   
**对于人来说不可感知，但对于服务器来说是可以感知的，我们叫他反向代理服务器**

**总结**

说白了：“正向”、“反向”是相对于人的感知来说的。人能感受到的代理就是正向代理，人感受不到的代理就是反向代理。

**二、初识Nginx与Php-fpm**

**Nginx是什么**

Nginx (“engine x”) 是一个高性能的HTTP和反向代理服务器，也是一个IMAP/POP3/SMTP服务器。

**Php-fpm是什么**

1、cgi、fast-cgi协议

cgi的历史

早期的webserver只处理html等静态文件，但是随着技术的发展，出现了像php等动态语言。   
webserver处理不了了，怎么办呢？那就交给php解释器来处理吧！   
交给php解释器处理很好，但是，php解释器如何与webserver进行通信呢？

为了解决不同的语言解释器(如php、python解释器)与webserver的通信，于是出现了cgi协议。只要你按照cgi协议去编写程序，就能实现语言解释器与webwerver的通信。如php-cgi程序。

fast-cgi的改进

有了cgi协议，解决了php解释器与webserver通信的问题，webserver终于可以处理动态语言了。但是，webserver每收到一个请求，都会去fork一个cgi进程，请求结束再kill掉这个进程。这样有10000个请求，就需要fork、kill php-cgi进程10000次。

有没有发现很浪费资源？

于是，出现了cgi的改良版本，fast-cgi。fast-cgi每次处理完请求后，不会kill掉这个进程，而是保留这个进程，使这个进程可以一次处理多个请求。这样每次就不用重新fork一个进程了，大大提高了效率。

2、php-fpm是什么

php-fpm即php-Fastcgi Process Manager.   
php-fpm是 FastCGI 的实现，并提供了进程管理的功能。   
进程包含 master 进程和 worker 进程两种进程。   
master 进程只有一个，负责监听端口，接收来自 Web Server 的请求，而 worker 进程则一般有多个(具体数量根据实际需要配置)，每个进程内部都嵌入了一个 PHP 解释器，是 PHP 代码真正执行的地方。

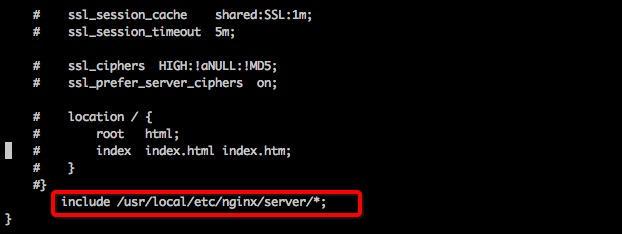
**三、Nginx如何与Php-fpm结合**

上面我们说了，Nginx不只有处理http请求的功能，还能做反向代理。Nginx通过反向代理功能将动态请求转向后端Php-fpm。

**下面我们来配置一个全新的Nginx+Php-fpm**

**1、配置nginx.conf文件**

进入nginx目录下，编辑 nginx.conf文件。   
如图，在nginx.conf最后一行，添加include文件



**2、添加对应的server**

进入上面include的路径，添加一个server.



**下面我们解释一下配置项的含义：**

server {

listen 80; #监听80端口，接收http请求

server\_name www.example.com; #就是网站地址

root /usr/local/etc/nginx/www/huxintong\_admin; # 准备存放代码工程的路径

#路由到网站根目录www.example.com时候的处理

location / {

index index.php; #跳转到www.example.com/index.php

autoindex on;

}

#当请求网站下php文件的时候，反向代理到php-fpm

location ~ \.php$ {

include /usr/local/etc/nginx/fastcgi.conf; #加载nginx的fastcgi模块

fastcgi\_intercept\_errors on;

fastcgi\_pass 127.0.0.1:9000; #nginx fastcgi进程监听的IP地址和端口

}

}

总而言之：当我们访问www.example.com的时候，处理流程是这样的：

www.example.com

|

|

Nginx

|

|

路由到www.example.com/index.php

|

|

加载nginx的fast-cgi模块

|

|

fast-cgi监听127.0.0.1:9000地址

|

|

www.example.com/index.php请求到达127.0.0.1:9000

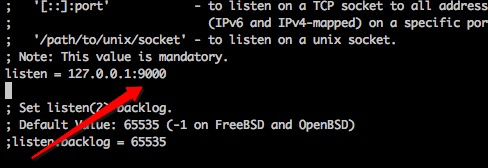
|

|

等待处理...

**下面我们启用php的php-fpm来处理这个请求**

打开php-fpm.conf文件，我们看到如下配置：



即:php-fpm模块监听127.0.0.1:9000端口，等待请求到来去处理。

**四、总结**

nginx与php-fpm的结合，完整的流程是这样的。

www.example.com

|

|

Nginx

|

|

路由到www.example.com/index.php

|

|

加载nginx的fast-cgi模块

|

|

fast-cgi监听127.0.0.1:9000地址

|

|

www.example.com/index.php请求到达127.0.0.1:9000

|

|

php-fpm 监听127.0.0.1:9000

|

|

php-fpm 接收到请求，启用worker进程处理请求

|

|

php-fpm 处理完请求，返回给nginx

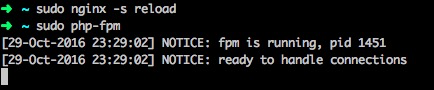
|

|

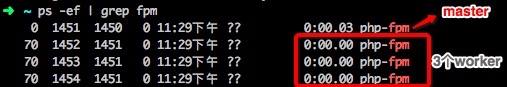
nginx将结果通过http返回给浏览器

**五、效果展示**

**1、启动nginx与php-fpm模块**



**启动成功，我们查看php-fpm进程**



如上图，有一个master进程，3个worker进程。

## 6.Rabbitmq相关

#### Rabbitmq消息怎么实现持久化

## Laravel相关

#### 为什么用laravel

#### Laravel中加载一个类的过程/原理

## 7.设计模式相关

了解两三个设计模式

单例模式

## 8.Linux相关

#### 经常使用的命令（查看内存使用情况，ps，nohup）

#### 进程线程是啥？

答：首先得知道什么是进程什么是线程？

我的理解是进程是指在系统中正在运行的一个应用程序；程序一旦运行就是进程，或者更专业化来说：进程是指程序执行时的一个实例。

线程是进程的一个实体。

进程——资源分配的最小单位，线程——程序执行的最小单位。

线程进程的区别体现在几个方面：

第一：因为进程拥有独立的堆栈空间和数据段，所以每当启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这对于多进程来说十分“奢侈”，系统开销比较大，而线程不一样，线程拥有独立的堆栈空间，但是共享数据段，它们彼此之间使用相同的地址空间，共享大部分数据，比进程更节俭，开销比较小，切换速度也比进程快，效率高，但是正由于进程之间独立的特点，使得进程安全性比较高，也因为进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。一个线程死掉就等于整个进程死掉。

第二：体现在通信机制上面，正因为进程之间互不干扰，相互独立，进程的通信机制相对很复杂，譬如管道，信号，消息队列，共享内存，套接字等通信机制，而线程由于共享数据段所以通信机制很方便。。

3.属于同一个进程的所有线程共享该进程的所有资源，包括文件描述符。而不同过的进程相互独立。

4.线程又称为轻量级进程，进程有进程控制块，线程有线程控制块；

5.线程必定也只能属于一个进程，而进程可以拥有多个线程而且至少拥有一个线程；

第四：体现在程序结构上，举一个简明易懂的列子：当我们使用进程的时候，我们不自主的使用if else嵌套来判断pid，使得程序结构繁琐，但是当我们使用线程的时候，基本上可以甩掉它，当然程序内部执行功能单元需要使用的时候还是要使用，所以线程对程序结构的改善有很大帮助。

进程与线程的选择取决以下几点：

1、需要频繁创建销毁的优先使用线程；因为对进程来说创建和销毁一个进程代价是很大的。

2、线程的切换速度快，所以在需要大量计算，切换频繁时用线程，还有耗时的操作使用线程可提高应用程序的响应

3、因为对CPU系统的效率使用上线程更占优，所以可能要发展到多机分布的用进程，多核分布用线程；

4、并行操作时使用线程，如C/S架构的服务器端并发线程响应用户的请求；

5、需要更稳定安全时，适合选择进程；需要速度时，选择线程更好。

#### 进程间通信、线程见通信有哪些

进程间的通信，它的数据空间的独立性决定了它的通信相对比较复杂，需要通过操作系统。以前进程间的通信只能是单机版的，现在操作系统都继承了基于套接字（socket）的进程间的通信机制。这样进程间的通信就不局限于单台计算机了，实现了网络通信。

进程的通信机制主要有：管道、有名管道、消息队列、信号量、共享空间、信号、套接字。

1.信号

信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过POSIX实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。信号事件的发生有两个来源：硬件来源(比如我们按下了键盘或者其它硬件故障)；软件来源。

信号分为可靠信号和不可靠信号，实时信号和非实时信号。

进程有三种方式响应信号：

•忽略信号

•捕捉信号

•执行缺省操作

2.信号量

信号量也可以说是一个计数器，常用来处理进程或线程同步的问题，特别是对临界资源的访问同步问题。临界资源：为某一时刻只能由一个进程或线程操作的资源，当信号量的值大于或等于0时，表示可以供并发进程访问的临界资源数，当小于0时，表示正在等待使用临界资源的进程数。更重要的是，信号量的值仅能由PV操作来改变。

3.消息队列

消息队列是存放在内核中的消息链表，每个消息队列由消息队列标识符标识，于管道不同的是，消息队列存放在内核中，只有在内核重启时才能删除一个消息队列，内核重启也就是系统重启，同样消息队列的大小也是受限制的。

4.共享内存

共享内存就是分配一块能被其他进程访问的内存。共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。首先说下在使用共享内存区前，必须通过系统函数将其附加到进程的地址空间或说为映射到进程空间。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到 进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互 斥锁和信号量都可以。采用共享内存通信的一个显而易 见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而 共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就 解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存 中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

5.管道

管道传递数据是单向性的，只能从一方流向另一方，也就是一种半双工的通信方式；只用于有亲缘关系的进程间的通信，亲缘关系也就是父子进程或兄弟进程；没有名字并且大小受限，传输的是无格式的流，所以两进程通信时必须约定好数据通信的格式。管道它就像一个特殊的文件，但这个文件之存在于内存中，在创建管道时，系统为管道分配了一个页面作为数据缓冲区，进程对这个数据缓冲区进行读写，以此来完成通信。其中一个进程只能读一个只能写，所以叫半双工通信，为什么一个只能读一个只能写呢?因为写进程是在缓冲区的末尾写入，读进程是在缓冲区的头部读取，他们各自 的数据结构不同，所以功能不同。

6.命名管道

命名管道(NamedPipe)是服务器进程和一个或多个客户进程之间通信的单向或双向管道。不同于匿名管道的是：命名管道可以在不相关的进程之间和不同计算机之间使用，服务器建立命名管道时给它指定一个名字，任何进程都可以通过该名字打开管道的另一端，根据给定的权限和服务器进程通信。命名管道提供了相对简单的编程接口，使通过网络传输数据并不比同一计算机上两进程之间通信更困难，不过如果要同时和多个进程通信它就力不从心了。

命名管道不同与管道只能在具有亲缘关系的进程间通信了。它提供了一个路径名与之关联，有了自己的传输格式。

命名管道和管道的不同之处还有一点是,有名管道是个设备文件，存储在文件系统中，没有亲缘关系的进程也可以访问，但是它要按照先进先出的原则读取数据。同样也是单双工的。

7.套接字

套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机间的进程通信。

二、线程间通信

线程间通信：由于多线程共享地址空间和数据空间，所以多个线程间的通信是一个线程的数据可以直接提供给其他线程使用，而不必通过操作系统（也就是内核的调度）。

1.锁机制

包括互斥锁、条件变量、读写锁；

互斥锁提供了以排他方式防止数据结构被并发修改的方法。

使用条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

读写锁允许多个线程同时读共享数据，而对写操作是互斥的。

2.信号量机制(Semaphore)

包括无名线程信号量和命名线程信号量

3.信号机制(Signal)

类似进程间的信号处理

线程间的通信目的主要是用于线程同步。所以线程没有像进程通信中的用于数据交换的通信机制。

三、linux中进程间通信和线程间通信的区别

1.linux中的进程，是有fork()系统调用创建的，进程间都有独立的地址空间，他们之间不能直接通信，必须通过一些IPC进程进程间通信机制来完成。常见的IPC有：PIPE，命名管道，信号，共享内存以及socket等；

2.linux中的线程，是clone()系统调用创建的,一个进程下的线程间是共享内存空间的，故线程A可以之间访问线程B中定义的变量，但是必须注意并发的情况；

3.另：“线程上下文”的规模要远远小于进程上下文

四、进程/线程间同步机制

临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）四种方式

1.临界区

通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。

2.互斥量

采用互斥对象机制。 只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享 .互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

3.信号量

它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目 .信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。

PV操作及信号量的概念都是由荷兰科学家E.W.Dijkstra提出的。信号量S是一个整数，S大于等于零时代表可供并发进程使用的资源实体数，但S小于零时则表示正在等待使用共享资源的进程数。

P操作申请资源：

（1）S减1；

（2）若S减1后仍大于等于零，则进程继续执行；

（3）若S减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。

V操作 释放资源：

（1）S加1；

（2）若相加结果大于零，则进程继续执行；

（3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。

4.事 件

通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作 .

5.总结

1. 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。

2.互斥量（Mutex），信号灯（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用WaitForSingleObject来等待进程和线程退出。

3.通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号灯对象可以说是一种资源计数器。

#### Grep命令（不包含某某，查出来一个多少行）

#### Scp命令

#### sort命令

#### uniq命令

## 9.网络通信相关

#### 正向代理和反向代理了解

#### 负载均衡的机制

#### 网络协议

#### HTTP协议

#### 一个HTTP请求发送都有哪些内容

#### 状态码：300-500之间的状态码

对网站管理工作者来说有个词不陌生，HTTP状态码，它是用以表示网页服务器HTTP响应状态的3位数字代码。状态码的第一个数字代表了响应的五种状态之一。

1XX系列：指定客户端应相应的某些动作，代表请求已被接受，需要继续处理。由于 HTTP/1.0 协议中没有定义任何 1xx 状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送 1xx 响应。

2XX系列：代表请求已成功被服务器接收、理解、并接受。这系列中最常见的有200、201状态码。

200状态码：表示请求已成功，请求所希望的响应头或数据体将随此响应返回

201状态码：表示请求成功并且服务器创建了新的资源，且其 URI 已经随Location 头信息返回。假如需要的资源无法及时建立的话，应当返回 '202 Accepted'

202状态码：服务器已接受请求，但尚未处理

3XX系列：代表需要客户端采取进一步的操作才能完成请求，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的 Location 域中指明。这系列中最常见的有301、302状态码。

301状态码：被请求的资源已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。

302状态码：请求的资源临时从不同的URI响应请求，但请求者应继续使用原有位置来进行以后的请求

304自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。 如果网页自请求者上次请求后再也没有更改过，您应将服务器配置为返回此响应(称为 If-Modified-Since HTTP 标头)。

4XX系列：表示请求错误。代表了客户端看起来可能发生了错误，妨碍了服务器的处理。常见有：401、404状态码。

401状态码：请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403状态码：服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。

404状态码：请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂时的还是永久的。假如服务器知道情况的话，应当使用410状态码来告知旧资源因为某些内部的配置机制问题，已经永久的不可用，而且没有任何可以跳转的地址。404这个状态码被广泛应用于当服务器不想揭示到底为何请求被拒绝或者没有其他适合的响应可用的情况下。

5xx系列：代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。常见有500、503状态码。

500状态码：服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。

503状态码：由于临时的服务器维护或者过载，服务器当前无法处理请求。通常，这个是暂时状态，一段时间会恢复

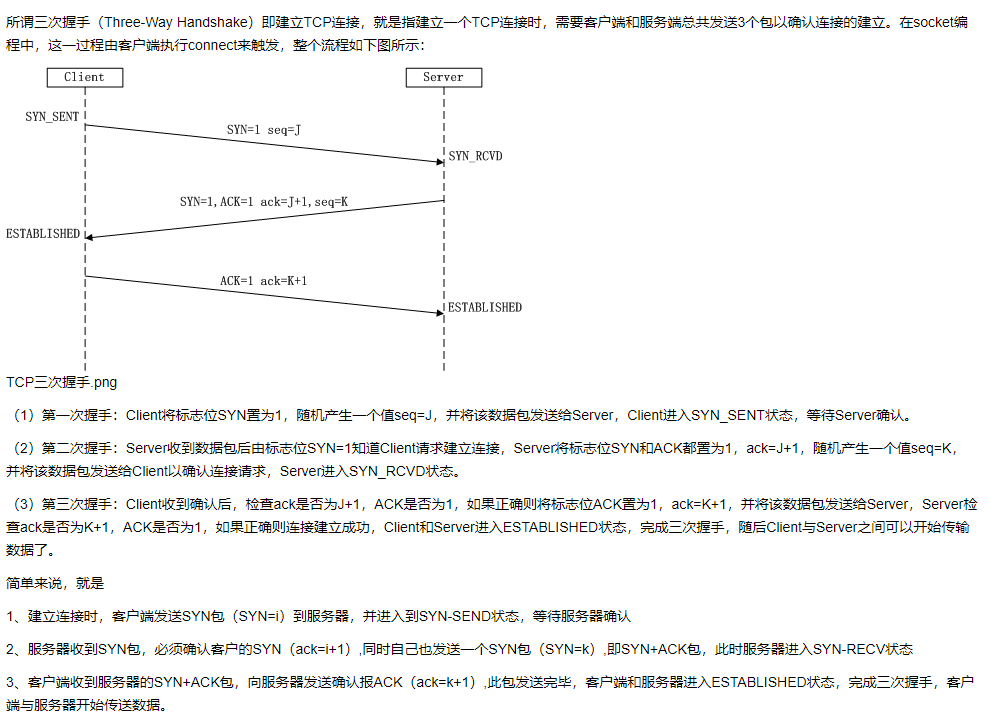
了解基本SEO状态码，是SEO优化人员必备知识。HTTP状态码是服务器和客户端之间交流信息的语言。通过查看网站日志的HTTP码，我们可以清楚查看搜索引擎在网站的爬取情况。

#### TCP协议

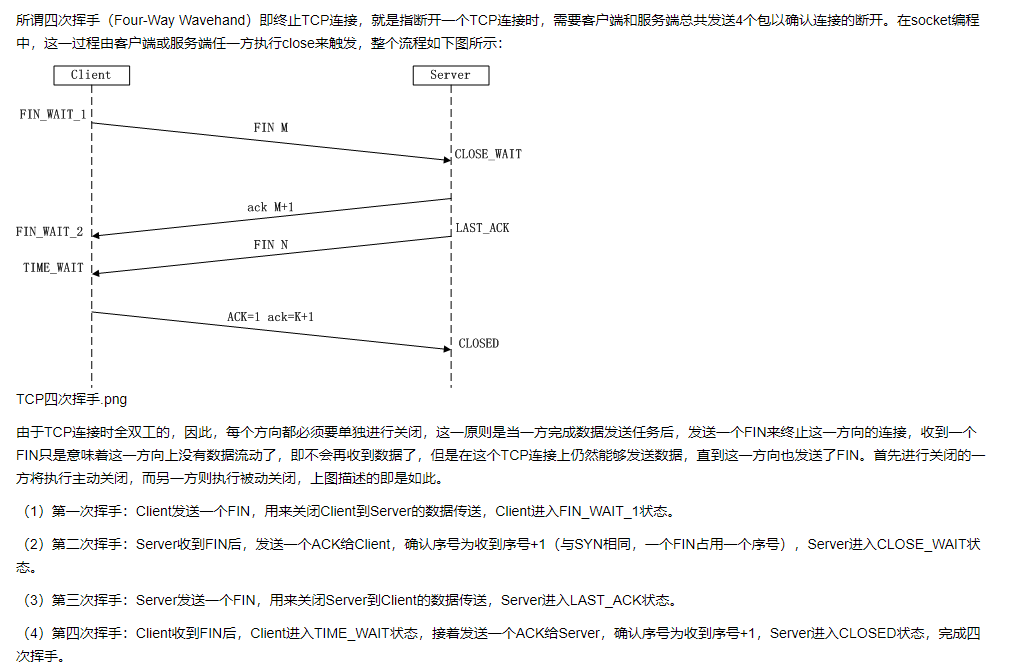
参考

<https://www.cnblogs.com/buxiangxin/p/8336022.html>

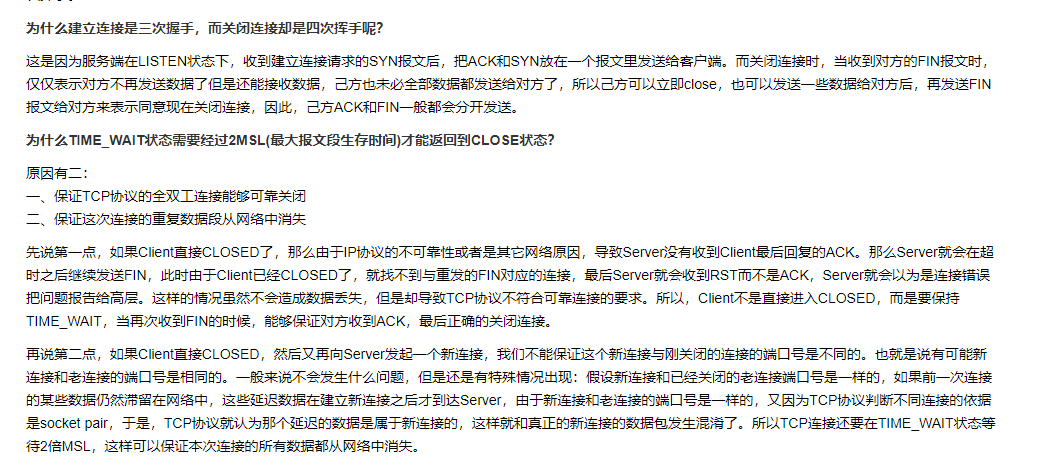
###### 三次握手，



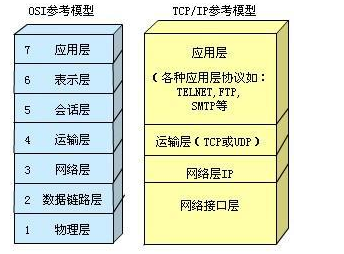
###### 四次挥手

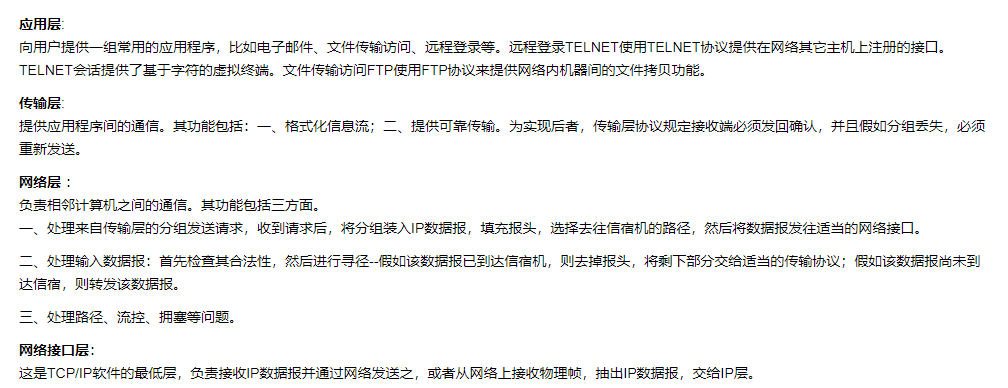


三次握手，四次挥手的原因



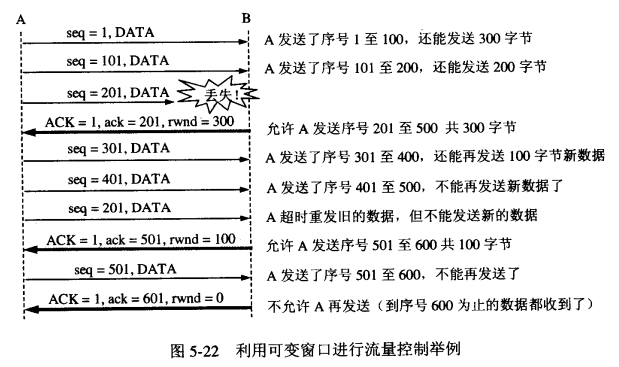
###### 七层模型，对应的协议

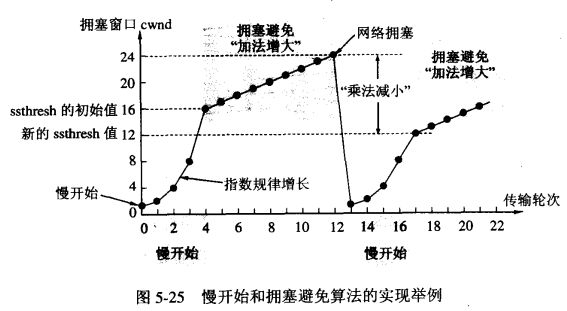




###### 慢启动，自适应（tcp协议C语言接触比较多，当年在国师面前卖弄了一下），感觉这个说出来会是加分项，并不是所有人都还记得。

<https://www.cnblogs.com/woaiyy/p/3554182.html>讲解很清晰，两个图理解一下





## 10.零碎知识点

empty(‘0’) 返回true

正则表达式

|  |  |  |  |
| --- | --- | --- | --- |
| **变量** | **empty** | **is\_null** | **isset** |
| $a=”” | true | false | true |
| $a=null | true | true | false |
| var $a | true | true | false |
| $a=array() | true | false | true |
| $a=false | true | false | true |
| $a=15 | false | false | true |
| $a=1 | false | false | true |
| $a=0 | true | false | true |
| $a=”0” | true | false | true |
| $a=”true” | false | false | true |
| $a=”false” | false | false | true |

互联网分布式架构设计，提高系统并发能力的方式

方法论上主要有两种：垂直扩展（Scale Up）与水平扩展（Scale Out）。

垂直扩展：提升单机处理能力。垂直扩展的方式又有两种：

（1）增强单机硬件性能，例如：增加CPU核数如32核，升级更好的网卡如万兆，升级更好的硬盘如SSD，扩充硬盘容量如2T，扩充系统内存如128G；

（2）提升单机架构性能，例如：使用Cache来减少IO次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

不管是提升单机硬件性能，还是提升单机架构性能，都有一个致命的不足：单机性能总是有极限的。所以互联网分布式架构设计高并发终极解决方案还是水平扩展。

水平扩展：只要增加服务器数量，就能线性扩充系统性能。水平扩展对系统架构设计是有要求的，如何在架构各层进行可水平扩展的设计，以及互联网公司架构各层常见的水平扩展实互联网分层架构中，各层次水平扩展的实践又有所不同：

（1）反向代理层可以通过“DNS轮询”的方式来进行水平扩展；

（2）站点层可以通过nginx来进行水平扩展；

（3）服务层可以通过服务连接池来进行水平扩展；

（4）数据库可以按照数据范围，或者数据哈希的方式来进行水平扩展；践，是本文重点讨论的内容。

常见性能优化策略分类

代码

之所以把代码放到第一位，是因为这一点最容易引起技术人员的忽视。很多技术人员拿到一个性能优化的需求以后，言必称缓存、异步、JVM等。实际上，第一步就应该是分析相关的代码，找出相应的瓶颈，再来考虑具体的优化策略。有一些性能问题，完全是由于代码写的不合理，通过直接修改一下代码就能解决问题的，比如for循环次数过多、作了很多无谓的条件判断、相同逻辑重复多次等。

数据库

数据库的调优，总的来说分为以下三部分：

SQL调优

这是最常用、每一个技术人员都应该掌握基本的SQL调优手段（包括方法、工具、辅助系统等）。这里以MySQL为例，最常见的方式是，由自带的慢查询日志或者开源的慢查询系统定位到具体的出问题的SQL，然后使用explain、profile等工具来逐步调优，最后经过测试达到效果后上线。这方面的细节，可以参考MySQL索引原理及慢查询优化。

架构层面的调优

这一类调优包括读写分离、多从库负载均衡、水平和垂直分库分表等方面，一般需要的改动较大，但是频率没有SQL调优高，而且一般需要DBA来配合参与。那么什么时候需要做这些事情？我们可以通过内部监控报警系统（比如Zabbix），定期跟踪一些指标数据是否达到瓶颈，一旦达到瓶颈或者警戒值，就需要考虑这些事情。通常，DBA也会定期监控这些指标值。

连接池调优

我们的应用为了实现数据库连接的高效获取、对数据库连接的限流等目的，通常会采用连接池类的方案，即每一个应用节点都管理了一个到各个数据库的连接池。随着业务访问量或者数据量的增长，原有的连接池参数可能不能很好地满足需求，这个时候就需要结合当前使用连接池的原理、具体的连接池监控数据和当前的业务量作一个综合的判断，通过反复的几次调试得到最终的调优参数。

缓存

分类

本地缓存（HashMap/ConcurrentHashMap、Ehcache、Guava Cache等），缓存服务（Redis/Tair/Memcache等）。

使用场景

什么情况适合用缓存？考虑以下两种场景：

•短时间内相同数据重复查询多次且数据更新不频繁，这个时候可以选择先从缓存查询，查询不到再从数据库加载并回设到缓存的方式。此种场景较适合用单机缓存。

•高并发查询热点数据，后端数据库不堪重负，可以用缓存来扛。

选型考虑

•如果数据量小，并且不会频繁地增长又清空（这会导致频繁地垃圾回收），那么可以选择本地缓存。具体的话，如果需要一些策略的支持（比如缓存满的逐出策略），可以考虑Ehcache；如不需要，可以考虑HashMap；如需要考虑多线程并发的场景，可以考虑ConcurentHashMap。

•其他情况，可以考虑缓存服务。目前从资源的投入度、可运维性、是否能动态扩容以及配套设施来考虑，我们优先考虑Tair。除非目前Tair还不能支持的场合（比如分布式锁、Hash类型的value），我们考虑用Redis。

设计关键点

什么时候更新缓存？如何保障更新的可靠性和实时性？

更新缓存的策略，需要具体问题具体分析。这里以门店POI的缓存数据为例，来说明一下缓存服务型的缓存更新策略是怎样的？目前约10万个POI数据采用了Tair作为缓存服务，具体更新的策略有两个：

•接收门店变更的消息，准实时更新。

•给每一个POI缓存数据设置5分钟的过期时间，过期后从DB加载再回设到DB。这个策略是对第一个策略的有力补充，解决了手动变更DB不发消息、接消息更新程序临时出错等问题导致的第一个策略失效的问题。通过这种双保险机制，有效地保证了POI缓存数据的可靠性和实时性。

缓存是否会满，缓存满了怎么办？

对于一个缓存服务，理论上来说，随着缓存数据的日益增多，在容量有限的情况下，缓存肯定有一天会满的。如何应对？

① 给缓存服务，选择合适的缓存逐出算法，比如最常见的LRU。

② 针对当前设置的容量，设置适当的警戒值，比如10G的缓存，当缓存数据达到8G的时候，就开始发出报警，提前排查问题或者扩容。

③ 给一些没有必要长期保存的key，尽量设置过期时间。

缓存是否允许丢失？丢失了怎么办？

根据业务场景判断，是否允许丢失。如果不允许，就需要带持久化功能的缓存服务来支持，比如Redis或者Tair。更细节的话，可以根据业务对丢失时间的容忍度，还可以选择更具体的持久化策略，比如Redis的RDB或者AOF。

缓存被“击穿”问题

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑另外一个问题：缓存被“击穿”的问题。

•概念：缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

•

如何解决：业界比较常用的做法，是使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。类似下面的代码：

public String get(key) {

String value = redis.get(key);

if (value == null) { //代表缓存值过期

//设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db

if (redis.setnx(key\_mutex, 1, 3 \* 60) == 1) { //代表设置成功

value = db.get(key);

redis.set(key, value, expire\_secs);

redis.del(key\_mutex);

} else { //这个时候代表同时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值即可

sleep(50);

get(key); //重试

}

} else {

return value;

}

}

异步

使用场景

针对某些客户端的请求，在服务端可能需要针对这些请求做一些附属的事情，这些事情其实用户并不关心或者用户不需要立即拿到这些事情的处理结果，这种情况就比较适合用异步的方式处理这些事情。

作用

•缩短接口响应时间，使用户的请求快速返回，用户体验更好。

•避免线程长时间处于运行状态，这样会引起服务线程池的可用线程长时间不够用，进而引起线程池任务队列长度增大，从而阻塞更多请求任务，使得更多请求得不到技术处理。

•线程长时间处于运行状态，可能还会引起系统Load、CPU使用率、机器整体性能下降等一系列问题，甚至引发雪崩。异步的思路可以在不增加机器数和CPU数的情况下，有效解决这个问题。

常见做法

一种做法，是额外开辟线程，这里可以采用额外开辟一个线程或者使用线程池的做法，在IO线程（处理请求响应）之外的线程来处理相应的任务，在IO线程中让response先返回。

如果异步线程处理的任务设计的数据量非常巨大，那么可以引入阻塞队列BlockingQueue作进一步的优化。具体做法是让一批异步线程不断地往阻塞队列里扔数据，然后额外起一个处理线程，循环批量从队列里拿预设大小的一批数据，来进行批处理（比如发一个批量的远程服务请求），这样进一步提高了性能。

另一种做法，是使用消息队列（MQ）中间件服务，MQ天生就是异步的。一些额外的任务，可能不需要我这个系统来处理，但是需要其他系统来处理。这个时候可以先把它封装成一个消息，扔到消息队列里面，通过消息中间件的可靠性保证把消息投递到关心它的系统，然后让这个系统来做相应的处理。

比如C端在完成一个提单动作以后，可能需要其它端做一系列的事情，但是这些事情的结果不会立刻对C端用户产生影响，那么就可以先把C端下单的请求响应先返回给用户，返回之前往MQ中发一个消息即可。而且这些事情理应不是C端的负责范围，所以这个时候用MQ的方式，来解决这个问题最合适。

NoSQL

和缓存的区别

先说明一下，这里介绍的和缓存那一节不一样，虽然可能会使用一样的数据存储方案（比如Redis或者Tair），但是使用的方式不一样，这一节介绍的是把它作为DB来用。如果当作DB来用，需要有效保证数据存储方案的可用性、可靠性。

使用场景

需要结合具体的业务场景，看这块业务涉及的数据是否适合用NoSQL来存储，对数据的操作方式是否适合用NoSQL的方式来操作，或者是否需要用到NoSQL的一些额外特性（比如原子加减等）。

如果业务数据不需要和其他数据作关联，不需要事务或者外键之类的支持，而且有可能写入会异常频繁，这个时候就比较适合用NoSQL（比如HBase）。

比如，美团点评内部有一个对exception做的监控系统，如果在应用系统发生严重故障的时候，可能会短时间产生大量exception数据，这个时候如果选用MySQL，会造成MySQL的瞬间写压力飙升，容易导致MySQL服务器的性能急剧恶化以及主从同步延迟之类的问题，这种场景就比较适合用Hbase类似的NoSQL来存储。

JVM调优

什么时候调？

通过监控系统（如没有现成的系统，自己做一个简单的上报监控的系统也很容易）上对一些机器关键指标（gc time、gc count、各个分代的内存大小变化、机器的Load值与CPU使用率、JVM的线程数等）的监控报警，也可以看gc log和jstat等命令的输出，再结合线上JVM进程服务的一些关键接口的性能数据和请求体验，基本上就能定位出当前的JVM是否有问题，以及是否需要调优。

怎么调？

1.如果发现高峰期CPU使用率与Load值偏大，这个时候可以观察一些JVM的thread count以及gc count（可能主要是young gc count），如果这两个值都比以往偏大（也可以和一个历史经验值作对比），基本上可以定位是young gc频率过高导致，这个时候可以通过适当增大young区大小或者占比的方式来解决。

2.如果发现关键接口响应时间很慢，可以结合gc time以及gc log中的stop the world的时间，看一下整个应用的stop the world的时间是不是比较多。如果是，可能需要减少总的gc time，具体可以从减小gc的次数和减小单次gc的时间这两个维度来考虑，一般来说，这两个因素是一对互斥因素，我们需要根据实际的监控数据来调整相应的参数（比如新生代与老生代比值、eden与survivor比值、MTT值、触发cms回收的old区比率阈值等）来达到一个最优值。

3.如果发生full gc或者old cms gc非常频繁，通常这种情况会诱发STW的时间相应加长，从而也会导致接口响应时间变慢。这种情况，大概率是出现了“内存泄露”，Java里的内存泄露指的是一些应该释放的对象没有被释放掉（还有引用拉着它）。那么这些对象是如何产生的呢？为啥不会释放呢？对应的代码是不是出问题了？问题的关键是搞明白这个，找到相应的代码，然后对症下药。所以问题的关键是转化成寻找这些对象。怎么找？综合使用jmap和MAT，基本就能定位到具体的代码。

多线程与分布式

使用场景

离线任务、异步任务、大数据任务、耗时较长任务的运行\*\*，适当地利用，可达到加速的效果。

注意：线上对响应时间要求较高的场合，尽量少用多线程，尤其是服务线程需要等待任务线程的场合（很多重大事故就是和这个息息相关），如果一定要用，可以对服务线程设置一个最大等待时间。

常见做法

如果单机的处理能力可以满足实际业务的需求，那么尽可能地使用单机多线程的处理方式，减少复杂性；反之，则需要使用多机多线程的方式。

对于单机多线程，可以引入线程池的机制，作用有二：

•提高性能，节省线程创建和销毁的开销

•限流，给线程池一个固定的容量，达到这个容量值后再有任务进来，就进入队列进行排队，保障机器极限压力下的稳定处理能力在使用JDK自带的线程池时，一定要仔细理解构造方法的各个参数的含义，如core pool size、max pool size、keepAliveTime、worker queue等，在理解的基础上通过不断地测试调整这些参数值达到最优效果。

如果单机的处理能力不能满足需求，这个时候需要使用多机多线程的方式。这个时候就需要一些分布式系统的知识了。首先就必须引入一个单独的节点，作为调度器，其他的机器节点都作为执行器节点。调度器来负责拆分任务，和分发任务到合适的执行器节点；执行器节点按照多线程的方式（也可能是单线程）来执行任务。这个时候，我们整个任务系统就由单击演变成一个集群的系统，而且不同的机器节点有不同的角色，各司其职，各个节点之间还有交互。这个时候除了有多线程、线程池等机制，像RPC、心跳等网络通信调用的机制也不可少。后续我会出一个简单的分布式调度运行的框架。

度量系统（监控、报警、服务依赖管理）

严格来说，度量系统不属于性能优化的范畴，但是这方面和性能优化息息相关，可以说为性能优化提供一个强有力的数据参考和支撑。没有度量系统，基本上就没有办法定位到系统的问题，也没有办法有效衡量优化后的效果。很多人不重视这方面，但我认为它是系统稳定性和性能保障的基石。

关键流程

如果要设计这套系统，总体来说有哪些关键流程需要设计呢？

① 确定指标

② 采集数据

③ 计算数据，存储结果

④ 展现和分析

需要监控和报警哪些指标数据？需要关注哪些？

按照需求出发，主要需要二方面的指标：

1.接口性能相关，包括单个接口和全部的QPS、响应时间、调用量（统计时间维度越细越好；最好是，既能以节点为维度，也可以以服务集群为维度，来查看相关数据）。其中还涉及到服务依赖关系的管理，这个时候需要用到服务依赖管理系统

2.单个机器节点相关，包括CPU使用率、Load值、内存占用率、网卡流量等。如果节点是一些特殊类型的服务（比如MySQL、Redis、Tair），还可以监控这些服务特有的一些关键指标。

数据采集方式

通常采用异步上报的方式，具体做法有两种：第一种，发到本地的Flume端口，由Flume进程收集到远程的Hadoop集群或者Storm集群来进行运算；第二种，直接在本地运算好以后，使用异步和本地队列的方式，发送到监控服务器。

数据计算

可以采用离线运算（MapReduce/Hive）或者实时/准实时运算（Storm/Spark）的方式，运算后的结果存入MySQL或者HBase；某些情况，也可以不计算，直接采集发往监控服务器。

展现和分析

提供统一的展现分析平台，需要带报表（列表/图表）监控和报警的功能。