## 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）

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>

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。

#### 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

#### 3.5Redis主从同步

#### 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

#### 4.1Php内存回收机制（感觉会常碰见）

#### 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

## 5.Nginx相关

#### 5.1介绍nginx的工作原理

## 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）

#### 进程间通信有哪些

#### 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

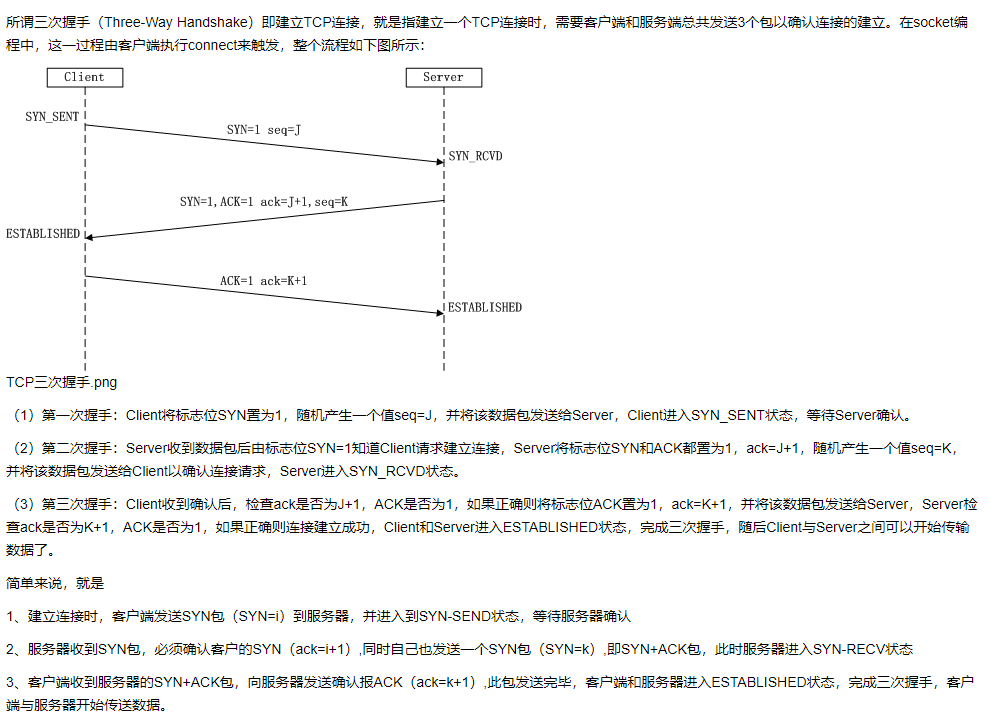
#### 状态码：300-500之间的状态码

#### TCP协议

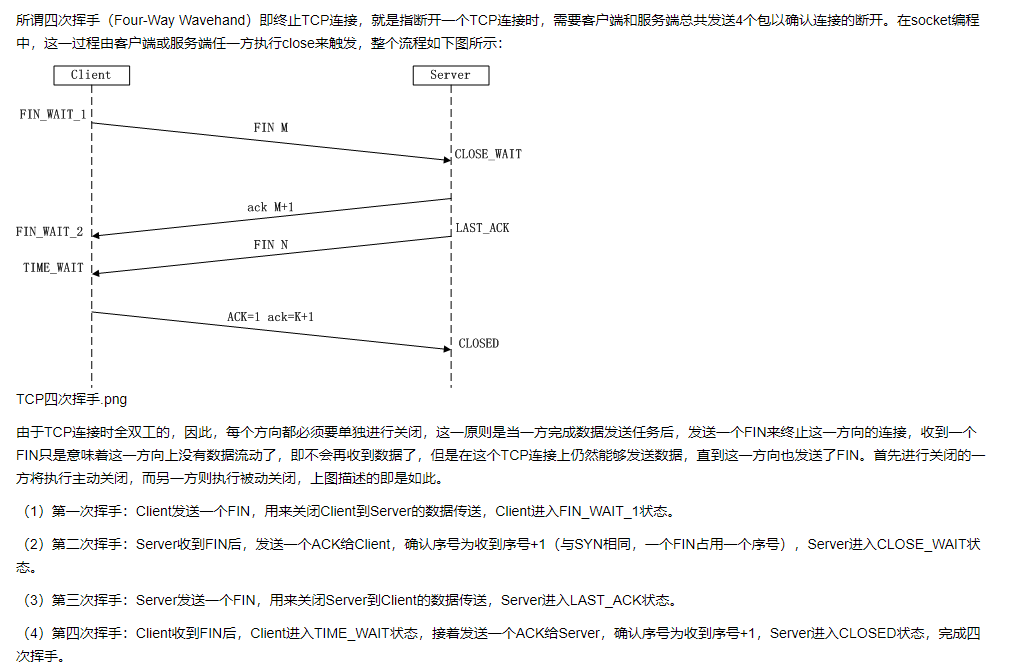
参考

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

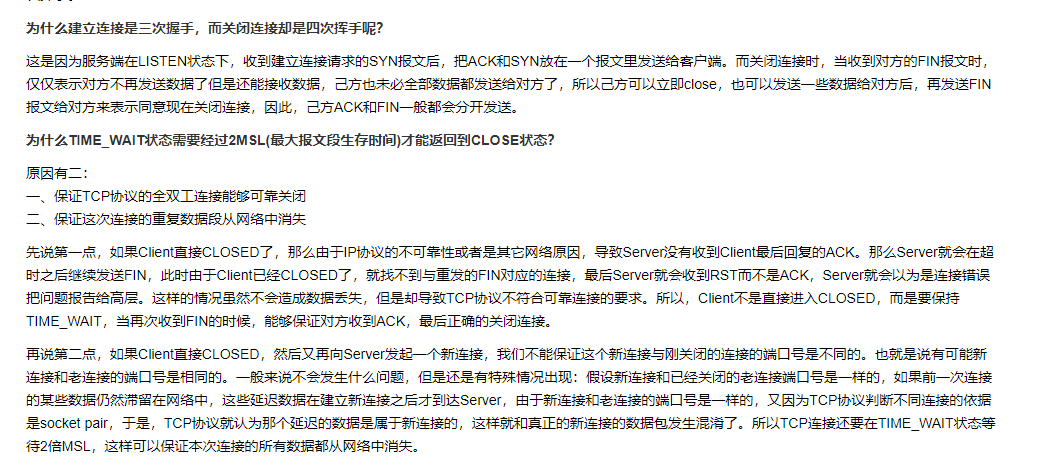
###### 三次握手，



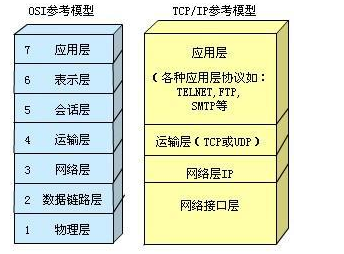
###### 四次挥手

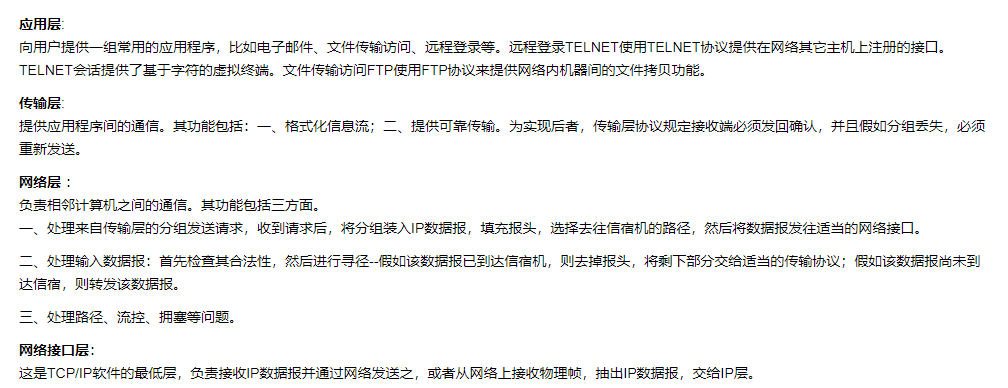


三次握手，四次挥手的原因



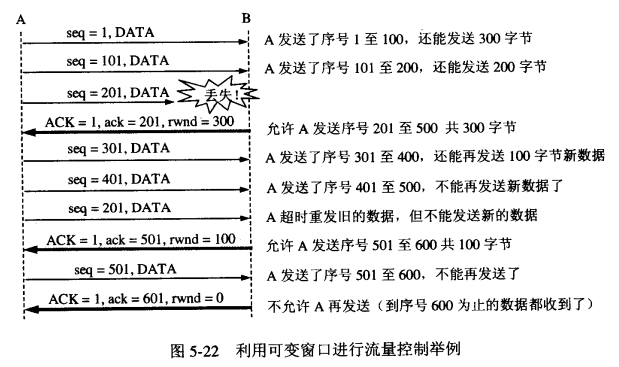
###### 七层模型，对应的协议

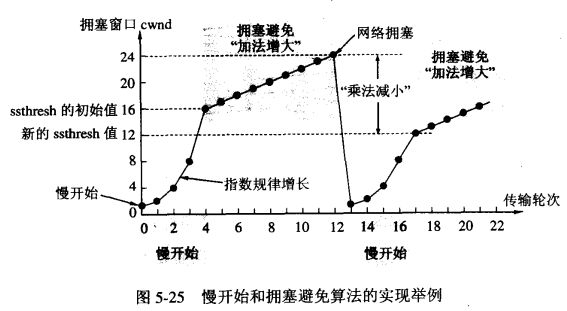




###### 慢启动，自适应（tcp协议C语言接触比较多，当年在国师面前卖弄了一下），感觉这个说出来会是加分项，并不是所有人都还记得。

<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。