# [PHP 进阶之路 - 亿级 pv 网站架构实战之性能压榨](https://segmentfault.com/a/1190000010455076)

* [性能优化](https://segmentfault.com/t/%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96/blogs)

* [php](https://segmentfault.com/t/php/blogs)

 12.6k 次阅读  ·  读完需要 44 分钟

123

本博客并非全部原创，其实是一个知识的归纳和汇总，里面我引用了很多网上、书上的内容。也给出了相关的链接。

****本文涉及的知识点比较多，大家可以根据关键字去搜索相关的内容和购买相应的书籍进行系统的学习。不对的地方大家予以批评指正。****

有人给我留言说，亿级 PV 就别写文章了，随便用几个开源软件就能搞定了，只要不犯什么大错。我不以为然，如果你利用了相同的思想，使用了更高性能的基础服务，也许就能支持更多的流量并发，节约更多的服务器，优化的思路才是重点。

本内容的视频分享见我的直播

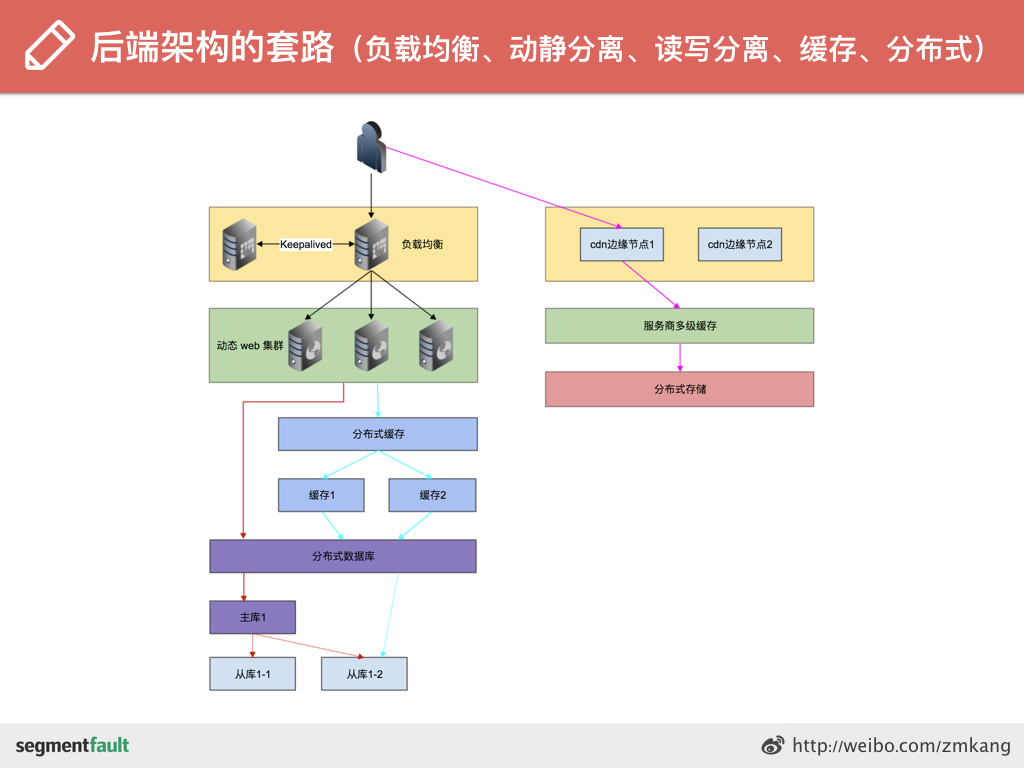
* [PHP 进阶之路 - 亿级 pv 网站架构的技术细节与套路](https://segmentfault.com/ls/1650000011318558)
* [PHP 进阶之路 - 亿级pv网站架构实战之性能压榨](https://segmentfault.com/ls/1650000011318558" \t "https://segmentfault.com/a/_blank)

# 性能优化的原则

1. 性能优化是建立在对业务的理解之上的
2. 性能优化与架构、业务相辅相成、密不可分的

# 性能优化的引入

我们先看一张简单的 web 架构图



从上到下从用户的浏览器到最后的数据库，那么我们说先前端的优化。

# 前端优化

雅虎军规：[http://www.cnblogs.com/paul-3...](http://www.cnblogs.com/paul-3/p/5999722.html" \t "https://segmentfault.com/a/_blank)  


## 减少 http 请求数

1. 图片、css、script等等这些都会增加http请求数，减少这些元素的数量就能减少响应时间。

把多个JS、CSS在可能的情况下写进一个文件，页面里直接写入图片也是不好的做法，应该写进CSS里，小图拼合后利用 background 来定位。

1. 现在很多 icon 都是直接做成字体，矢量高清，也减少网络请求数
2. 现在的前端框架都会通过组件的方式开发，最后打包生成一个 js 或者 两个 js 文件 + 一个 css 或者两个 css 文件。

## 利用浏览器缓存

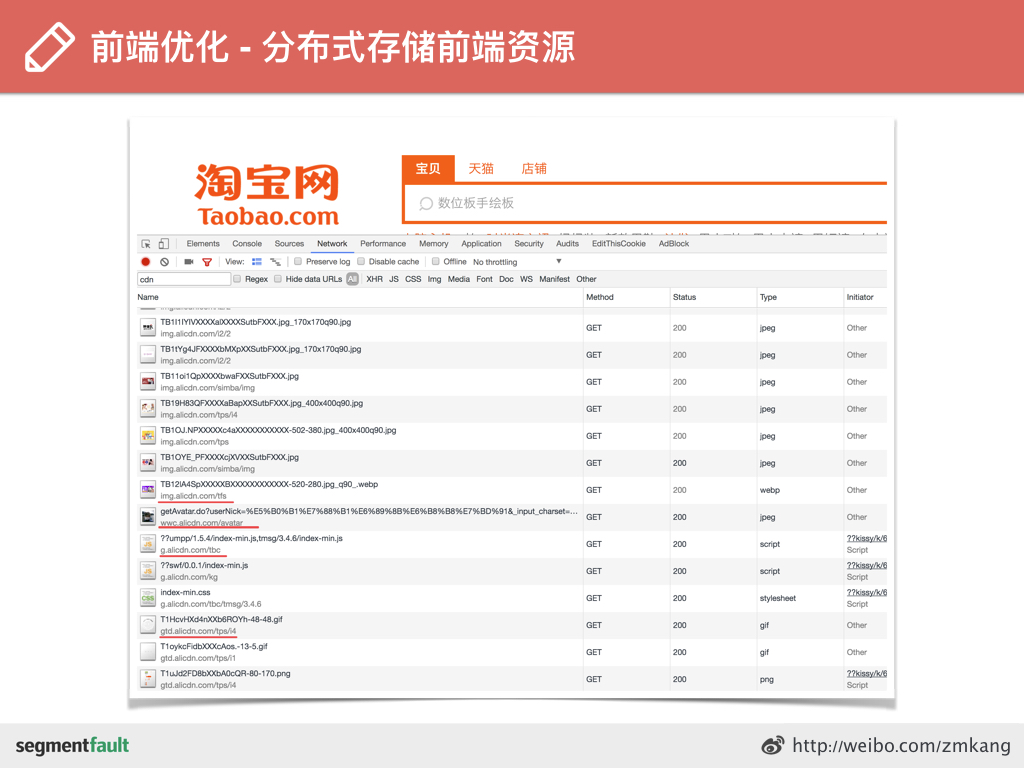
expires,cache-control,last-modified,etag  
[http://blog.csdn.net/eroswang...](http://blog.csdn.net/eroswang/article/details/8302191" \t "https://segmentfault.com/a/_blank)  
防止缓存，比如资源更新了，原来的做法是?v=xxxx 现在前端的打包工作可以能会生成 /v1.2.0/xxx.js

## 使用分布式存储前端资源

接地气利用 cdn 存储前端资源

## 多域名访问资源

* 原因一：浏览器对同一域名的并行请求数有上限，多个域名则支持更多并行请求
* 原因二：使用同一域名的时候无用的 cookie 简直是噩梦



## 数据压缩

1. 开启gzip
2. 前端资源本身的压缩，js/css 打包编译（去掉空格，语意简化）图片资源的压缩等。

## 优化首屏展示速度

1. 资源的按需加载，延时加载 [https://mengkang.net/229.html](https://mengkang.net/229.html" \t "https://segmentfault.com/a/_blank)
2. 图片的懒加载，淘宝的商品介绍太多图，用户点击进来又有多少人一直往下看图的呢？

# nginx 优化

  
分为下面三个部分来

## nginx 本身配置的优化

1. worker\_processes auto 设置多少子进程
2. worker\_cpu\_affinity 亲缘性绑定
3. worker\_rlimit\_nofile 65535 worker 进程打开的文件描述符的最大数
4. worker\_connections 65535 子进程最多处理的连接数
5. epoll 多路复用
6. sendfile on 是对文件I/O的系统调用的一个优化，系统api
7. 如果是反向代理web服务器，需要配置fastcgi相关的参数
8. 数据返回开启gzip压缩
9. 静态资源使用 http 缓存协议
10. 开启长连接 keepalive\_timeout

fastcgi\_connect\_timeout 300;

fastcgi\_send\_timeout 300;

fastcgi\_read\_timeout 300;

fastcgi\_buffer\_size 64k;

fastcgi\_buffers 4 64k;

fastcgi\_busy\_buffers\_size 128k;

fastcgi\_temp\_file\_write\_size 256k;

gzip on;

gzip\_min\_length 1k;

gzip\_buffers 4 16k;

gzip\_http\_version 1.0;

gzip\_comp\_level 2;

gzip\_types text/plain application/x-javascript text/css application/xml text/javascript application/json;

gzip\_vary on;

gzip\_proxied expired no-cache no-store private auth;

gzip\_disable "MSIE [1-6]\.";

location ~ .\*\.(gif|jpg|jpeg|png|bmp|swf)$

{

expires 30d;

}

[http://mailman.nginx.org/pipe...](http://mailman.nginx.org/pipermail/nginx/2010-March/019451.html" \t "https://segmentfault.com/a/_blank)

## tcp/ip 网络协议配置的优化

1. /proc/sys/net/ipv4/tcp\_tw\_recycle 1 开启TCP连接中TIME-WAIT sockets的快速回收,保证tcp\_timestamps = 1
2. /proc/sys/net/ipv4/tcp\_tw\_reuse 1 允许将TIME-WAIT sockets重新用于新的TCP连接 [https://mengkang.net/564.html](https://mengkang.net/564.html" \t "https://segmentfault.com/a/_blank)
3. /proc/sys/net/ipv4/tcp\_syncookies 0 是否需要关闭洪水抵御 看自己业务，比如秒杀，肯定需要关闭了
4. /proc/sys/net/ipv4/tcp\_max\_tw\_buckets 180000 否则经常出现 time wait bucket table overflow
5. tcp\_nodelay on 小文件快速返回，我之前通过网络挂载磁盘出现找不到的情况
6. tcp\_nopush on

tcp\_tw\_recycle 快速回收可能导致丢包的问题 [https://mengkang.net/1144.html](https://mengkang.net/1144.html" \t "https://segmentfault.com/a/_blank)

## linux 系统的优化

除了上面的网络协议配置也是在系统基础之外，为了配合nginx自己里面的设定需要做如下修改

1. /proc/sys/net/core/somaxconn 65535
2. ulimit -a 65535

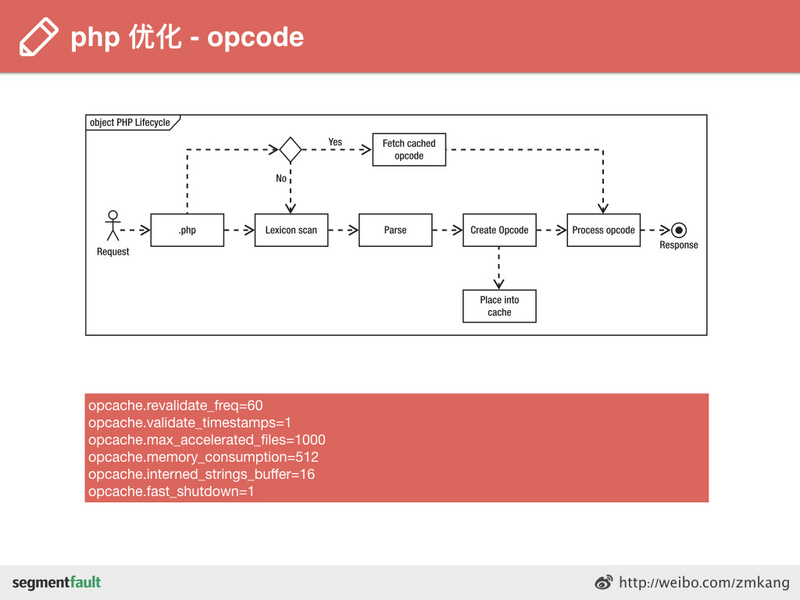
更多详细的优化配置说明：[http://os.51cto.com/art/20140...](http://os.51cto.com/art/201404/434930.htm" \t "https://segmentfault.com/a/_blank)

# php 优化

## 升级到 php7

注意有很多函数和扩展被废弃，比如mysql相关的，有风险，做好测试再切换。

## opcode 缓存

  
php 5.5 之后好像就内置了吧，需要在php.ini里添加如下配置

opcache.revalidate\_freq=60

opcache.validate\_timestamps=1

opcache.max\_accelerated\_files=1000

opcache.memory\_consumption=512

opcache.interned\_strings\_buffer=16

opcache.fast\_shutdown=1

1. opcache.revalidate\_freq

这个选项用于设置缓存的过期时间（单位是秒），当这个时间达到后，opcache会检查你的代码是否改变，如果改变了PHP会重新编译它，生成新的opcode，并且更新缓存。

1. opcache.validate\_timestamps

当这个选项被启用（设置为1），PHP会在opcache.revalidate\_freq设置的时间到达后检测文件的时间戳（timestamp）。

1. opcache.max\_accelerated\_files

这个选项用于控制内存中最多可以缓存多少个PHP文件。

1. opcache.memory\_consumption

你可以通过调用opcachegetstatus()来获取opcache使用的内存的总量

1. opcache.interned\_strings\_buffer

字符串opcache的复用，单位为MB

1. opcache.fast\_shutdown=1

开启快速停止续发事件，依赖于Zend引擎的内存管理模块

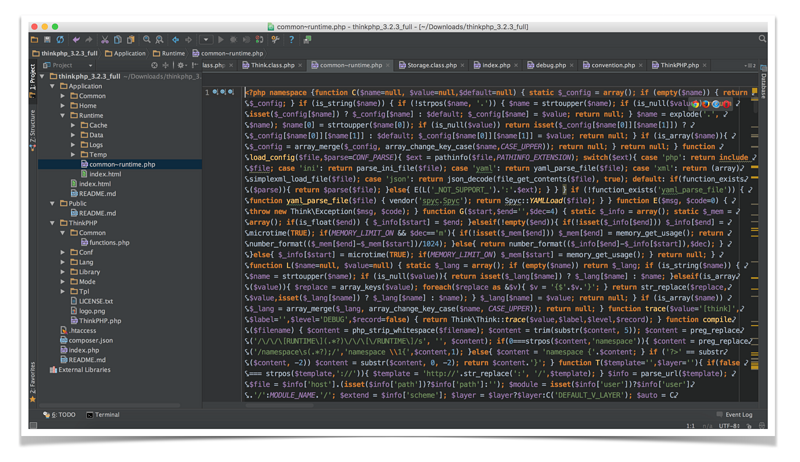
## php7 hugepage 的使用

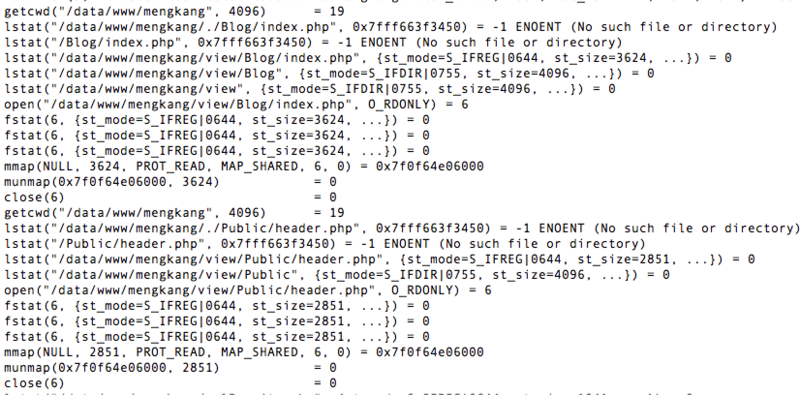
Hugepage 的作用：间接提高虚拟地址和内存地址转换过程中查表的TLB缓存命中率

opcache.huge\_code\_pages=1

鸟哥博客详细介绍：[http://www.laruence.com/2015/...](http://www.laruence.com/2015/10/02/3069.html" \t "https://segmentfault.com/a/_blank)

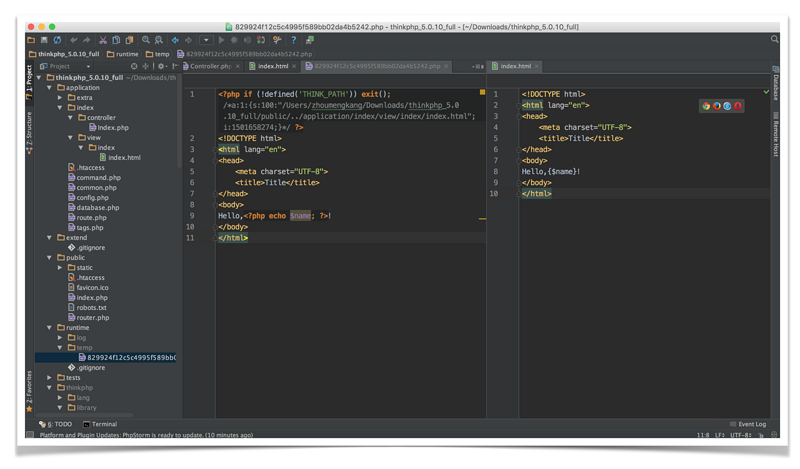
## 代码伪编译

以thinkphp为例，它会把框架基础组件（必须用到的组件）合并压缩到一个文件中，不仅减少了文件目录查找，文件打开的系统调用。  




通过stracephp-fpm子进程，可以清楚系统调用的过程，在我上面例子中有打开一个文件有12次系统调用（只是举例，我这里相对路径设置的原因导致多了两次文件查找）。如果有10个文件，那就是120次，优化的效果可能不是那么明显，但是这是一种思路。  
顺便说下 set\_include\_path能不用就不要用，上面的demo的截图里面找不到目录就是证明。

## 模板编译

  
模板把它们自定义的语法，最后转换成php语法，这样方便解析。而不是每次都解析一遍。

## xhprof 查找性能瓶颈

我的截图一直上传不成功，正好社区有这样的博客，推荐下 [https://segmentfault.com/a/11...](https://segmentfault.com/a/1190000003509917)

# 业务优化

## 非侵入式扩展开发

比如原来有一个model，叫问答，现在需要开发一个有奖问答，需要支持话题打赏，里面多了很多功能。这个时候应该利用面向对象的继承的特性。而不是做下面的开发

**<?phpclass** **AskModel** {

**public** **function** **detail**($id){

$info = 从数据库查询到该问题的信息;

// 逻辑1

// 逻辑2

}

}

**<?phpclass** **AskModel** {

**public** **function** **detail**($id){

$info = 从数据库查询到该问题的信息;

// 逻辑1

**if**($info['type'] == 2){

//...

}**else**{

}

// 逻辑2

**if**($info['type'] == 2){

//...

}**else**{

}

}

}

这样逻辑多了，子类型多了，逻辑判断就非常重复，程序运行起来低效可能是一方面，更多的是不可维护性。

业务和架构不分家，架构是建立在对业务的理解之上的。再放下[上次直播](https://segmentfault.com/l/1500000009978736" \t "https://segmentfault.com/a/_blank)的PPT （sf故障无法传图，等会补吧）

## 异步思想

举例：

1. 处理邮件发送。
2. gearman 图片裁剪。
3. 页面上 ajax 加载动态数据。
4. 图片的懒加载，双击图片看大图。
5. sf 上通过websocket 通知你有新的消息，但是并没有告诉你有什消息，点击消息图标才会去异步请求具体的消息。

这些都是异步的思想。能分步走就分步走，能不能请求的就不请求。

## 静态化

专题页面，比如秒杀页面，为了应对更大的流量、并发。而且更新起来也比较方便。

## 业务解耦

比如刚刚上面说的专题页面，还有必要走整个框架的一套流程吗？进来引用一大堆的文件，初始化一大堆的东西？是不是特别低效呢？所以需要业务解耦，专题页面如果真要框架（可以首次访问之后生成静态页面）也应该是足够轻量级的。不能与传统业务混为一谈。

## 分布式以及 soa

说业务优化，真的不得不提架构方面的东西，业务解耦之后，就有了分布式和soa，因为这在[上次分享](https://segmentfault.com/l/1500000009978736)中已经都说过了，就不多说了。  
只说下 soa 自定义 socket 传输协议。  
https://segmentfault.com/img/remote/1460000010158196  
最重要的就是在自定义头里面强调body\_len，注意设置为紧凑型，才能保证跨平台性 具体说明：[https://mengkang.net/586.html](https://mengkang.net/586.html" \t "https://segmentfault.com/a/_blank)

# Mysql 优化

数据索引相关的文章网上很多了，不足的地方大家补充。

## 表设计 - 拥抱 innodb

现在大多数情况都会使用innodb类型了。具体原因是 mysql 专家给的意见。  
我自己对 mysql 的优化不了解，每一个细分领域都是一片汪洋，每个人的时间精力是有限的，所以大家也不用什么都非要深入去研究，往往是一些计算机基础更为重要。  
参考这份ppt [https://static.mengkang.net/u...](https://static.mengkang.net/upload/file/20170806/1501990586261700.pdf" \t "https://segmentfault.com/a/_blank)

## 表设计 - 主键索引

1. innodb 需要一个主键，主键不要有业务用途，不要修改主键。
2. 主键最好保持顺序递增，随机主键会导致聚簇索引树频繁分裂，随机I/O增多，数据离散，性能下降。

举例：  
之前项目里有些索引是article\_id + tag\_id 联合做的主键，那么这种情况下，就是业务了属性了。主键也不是顺序递增，每插入新的数据都有可能导致很大的索引变动（了解下数据库b+索引的原理）

## 表设计 - 字段选择

1. 能选短整型，不选长整型。比如一篇文章的状态值，不可能有超过100种吧，不过怎么扩展，没必要用int了。
2. 能选 char 就避免 varchar，比如图片资源都有一个hashcode，固定长度20位，那么就可以选char了。
3. 当使用 varchar 的时候，长度够用就行，不要滥用。
4. 大文本单独分离，比如文章的详情，单独出一张表。其他基本信息放在一张表里，然后关联起来。
5. 冗余字段的使用，比如文章的详情字段，增加一个文章markdown解析之后的字段。

## 索引优化

大多数情况下，索引扫描要比全表扫描更快，性能更好。但也不是绝对的，比如需要查找的数据占了整个数据表的很大比例，反而使用索引更慢了。

1. 没有索引的更新，可能会导致全表数据都被锁住。所以更新的时候要根据索引来做。
2. 联合索引的使用
3. explain 的使用

联合索引“最左前缀”，查询优化器还会帮你调整条件表达式的顺序，以匹配组合索引的要求。

**CREATE** **TABLE** `test` (

`id` int(11) **unsigned** **NOT** NULL AUTO\_INCREMENT,

`a` int(10) **unsigned** **NOT** NULL,

`b` int(10) **unsigned** **NOT** NULL,

`c` int(10) **unsigned** **NOT** NULL,

PRIMARY **KEY** (`id`),

**KEY** `index\_abc` (`a`,`b`,`c`)

) **ENGINE**=**InnoDB** **DEFAULT** **CHARSET**=utf8;

能使用到索引

**explain** **select** \* **from** **test** **where** a=1;**explain** **select** \* **from** **test** **where** a=1 **and** b=2;**explain** **select** \* **from** **test** **where** a=1 **and** b=2 **and** c=3;**explain** **select** \* **from** **test** **where** a=1 **and** b **in** (2,3) **and** c=3;**explain** **select** \* **from** **test** **where** a=1 **and** b=2 **order** **by** c **desc**;

不能使用索引

**explain** **select** \* **from** **test** **where** a=1 **and** b **in** (2,3) **order** **by** c **desc**;**explain** **select** \* **from** **test** **where** b=2;

索引更详细讲解 [https://mengkang.net/1302.html](https://mengkang.net/1302.html" \t "https://segmentfault.com/a/_blank)

explain 搜到一篇不错的: [http://blog.csdn.net/woshiqjs...](http://blog.csdn.net/woshiqjs/article/details/24135495" \t "https://segmentfault.com/a/_blank)  
很重要的参数type,key,extra

### type 最常见的

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

| **值** | **说明** |
| --- | --- |
| const | 通过索引直接找到一个匹配行，一般主键索引的时候 |
| ref | 没有主键索引或者唯一索引的条件索引，查询结果多行，在联合查询中很常见 |
| index | 利用到了索引，有可能有其它排序，where 或者 group by 等 |
| all | 全表扫描，没有使用到索引 |

### extra

如果有Using filesort或者Using temporary的话,就必须要优化了

## 收集慢查询

my.ini 配置里增加

long\_query\_time=2

log-slow-queries=/data/var/mysql\_slow.log

## 使用 nosql

redis 丰富的数据类型，非常适合配合mysql 做一些关系型的查询。比如一个非常复杂的查询列表可以将其插入zset 做排序列表，然后具体的信息，通过zset里面的纸去mysql 里面去查询。

# 缓存优化

## 多级缓存

1. 请求内缓存

static 变量存储，比如朋友圈信息流，在一次性获取20条信息的时候，有可能，点赞的人里面20条里面有30个人是重复的，他们点赞你的a图片也点赞了你的b图片，所以这时，如果能使用static数组来存放这些用户的基本信息就高效了些。

1. 本地缓存

请求结束了，下拉更新朋友圈，里面又出现了上面的同样的好友，还得重新请求一次。所以本地常驻内存的缓存就更高效了。

1. 分布式缓存

在A服务器上已经查询过了，在下拉更新的时候被分配到B服务器上了，难道同样的数据再查一次再存到B服务器的本地缓存里面吗，弄一个分布式缓存吧，这样防止了重复查询。但是多了网络请求这一步。

很多时候是三者共存的。

## 避免缓存的滥用

案例分析

用户积分更新

* + 比如用户的基本信息和积分混在一起，当用户登录的时候赠送积分。则需要更新用户的积分，这个时候更新整个用户的基本信息缓存么？
  + 所以这里也可以运用下面 hashes 分片的原则去更新

1. 礼物和主题绑定缓存

为了取数据方便把多个数据源混合缓存了，这种情况，相比大家可能都见过，这是灾难性的设计。

{

id:x,

title:x,

gift:{

id:x,

name:x,

img:x,

}

}

如果需要更新礼物的图片，那么所有用到过这个礼物的话题的缓存都要更新。

## redis 使用场景举例

由于比较基础基础好的老司机就可以忽略了，新人同学可以看下 [https://mengkang.net/356.html](https://mengkang.net/356.html" \t "https://segmentfault.com/a/_blank)

## redis 优化

1. 多实例化，更高效地利用服务器 cpu
2. 内存优化，官方意见 [https://redis.io/topics/memor...](https://redis.io/topics/memory-optimization" \t "https://segmentfault.com/a/_blank) 有点老
3. 尽可能的使用 hashes ，时间复杂度低，查询效率高。同时还节约内存。Instagram 最开始用string来存图片id=>uid的关系数据，用了21g，后来改为水平分割，图片id 1000 取模，然后将分片的数据存在一个hashse 里面，这样最后的内容减少了5g，四分之一基本上。

每一段使用一个Hash结构存储，由于Hash结构会在单个Hash元素在不足一定数量时进行压缩存储，所以可以大量节约内存。这一点在String结构里是不存在的。而这个一定数量是由配置文件中的hash-zipmap-max-entries参数来控制的。

# 服务器认知的提升

下面的内容，只能是让大家有一个大概的认识，了解一个优化的方向，具体的内容需要系统学习很多很多的知识。

## 多进程的优势

多进程有利于 CPU 计算和 I/O 操作的重叠利用。一个进程消耗的绝大部分时间都是在磁盘I/O和网络I/O中。  
如果是单进程时cpu大量的时间都在等待I/O，所以我们需要使用多进程。

## 减少上下文切换

为了让所有的进程轮流使用系统资源，进程调度器在必要的时候挂起正在运行的进程，同时恢复以前挂起的某个进程。这个就是我们常说的“上下文切换”。

关于上下文我之前写一个简单笔记 [https://mengkang.net/729.html](https://mengkang.net/729.html" \t "https://segmentfault.com/a/_blank)

无限制增加进程数，则会增多 cpu 在各个进程间切换的次数。  
如果我们希望服务器支持较大的并发数，那么久要尽量减少上下文切换的次数，比如在nginx服务上nginx的子进程数不要超过cpu的核数。  
我们可以在压测的时候通过vmstat,nmon来监控系统上下文切换的次数。

## IOwait 不一定是 I/O 繁忙

**#** top

top - 09:40:40 up 565 days, 5:47, 2 users, load average: 0.03, 0.03, 0.00

Tasks: 121 total, 2 running, 119 sleeping, 0 stopped, 0 zombie

Cpu(s): 8.6%us, 0.3%sy, 0.0%ni, 90.7%id, 0.2%wa, 0.0%hi, 0.2%si, 0.0%st

一般情况下IOwait代表I/O操作的时间占（I/O操作的时间 + I/O和CPU时间）的比例。  
但是也时候也不准，比如nginx来作为web服务器，当我们开启很多nginx子进程，IOwait会很高，当再减少进程数到cpu核数附近时，IOwait会减少，监控网络流量会发现也增加。

## 多路复用 I/O 的使用

只要是提供socket服务，就可以利用多路复用 I/O 模型。  
需要补充的知识 [https://mengkang.net/726.html](https://mengkang.net/726.html" \t "https://segmentfault.com/a/_blank)

## 减少系统调用

strace 非常方便统计系统调用

# strace -c -p 23374

Process 23374 attached - interrupt to quit

^CProcess 23374 detached

% **time** seconds usecs/call calls errors **syscall**

------ ----------- ----------- --------- --------- ----------------

30.68 0.000166 0 648 poll

12.01 0.000065 0 228 munmap

11.65 0.000063 0 228 mmap

10.54 0.000057 0 660 recvfrom

10.35 0.000056 0 708 fstat

7.76 0.000042 0 252 **open**

6.10 0.000033 1 36 **write**

5.73 0.000031 0 72 24 access

5.18 0.000028 0 72 **read**

0.00 0.000000 0 276 **close**

0.00 0.000000 0 13 13 **stat**

0.00 0.000000 0 269 240 **lstat**

0.00 0.000000 0 12 rt\_sigaction

0.00 0.000000 0 12 rt\_sigprocmask

0.00 0.000000 0 12 pwrite

0.00 0.000000 0 48 setitimer

0.00 0.000000 0 12 **socket**

0.00 0.000000 0 12 **connect**

0.00 0.000000 0 12 **accept**

0.00 0.000000 0 168 sendto

0.00 0.000000 0 12 **shutdown**

0.00 0.000000 0 48 **fcntl**

0.00 0.000000 0 12 **flock**

0.00 0.000000 0 156 getcwd

0.00 0.000000 0 24 **chdir**

0.00 0.000000 0 24 **times**

0.00 0.000000 0 12 getuid

------ ----------- ----------- --------- --------- ----------------100.00 0.000541 4038 277 total

通过strace查看“系统调用时间”和“调用次数”来定位问题 [https://huoding.com/2013/10/0...](https://huoding.com/2013/10/06/288" \t "https://segmentfault.com/a/_blank)

## 自己构建web服务器

要想理解web服务器优化的原理，最好的办法是了解它的来龙去脉，实践就是最好的方式，我分为以下几个步骤：

1. 用 PHP 来实现一个动态 Web 服务器 [https://mengkang.net/491.html](https://mengkang.net/491.html" \t "https://segmentfault.com/a/_blank)
2. 简单静态 web 服务器（循环服务器）的实现 [https://mengkang.net/563.html](https://mengkang.net/563.html" \t "https://segmentfault.com/a/_blank)
3. 多进程并发的面向连接 Web 服务器的实践 [https://mengkang.net/571.html](https://mengkang.net/571.html" \t "https://segmentfault.com/a/_blank)
4. 简单静态 Select Web 服务器的实现 [https://mengkang.net/568.html](https://mengkang.net/568.html" \t "https://segmentfault.com/a/_blank)
5. I/O 多路复用 [https://mengkang.net/726.html](https://mengkang.net/726.html" \t "https://segmentfault.com/a/_blank)

上面是我的学习笔记，图片资源丢失了，大家可以根据相关关键词去搜搜相关的文章和书籍，更推荐大家去看书。