**PHP 解决高并发**

我们通常衡量一个Web系统的吞吐率的指标是QPS（Query Per Second，每秒处理请求数），解决每秒数万次的高并发场景，这个指标非常关键。举个例子，我们假设处理一个业务请求平均响应时间为100ms，同时，系统内有20台Apache的Web服务器，配置MaxClients为500个（表示Apache的最大连接数目）。

那么，我们的Web系统的理论峰值QPS为（理想化的计算方式）：

20\*500/0.1 = 100000 （10万QPS）

咦？我们的系统似乎很强大，1秒钟可以处理完10万的请求，5w/s的秒杀似乎是“纸老虎”哈。实际情况，当然没有这么理想。在高并发的实际场景下，机器都处于高负载的状态，在这个时候平均响应时间会被大大增加。

普通的一个p4的服务器每天最多能支持大约10万左右的IP，如果访问量超过10W那么需要专用的服务器才能解决，如果硬件不给力 软件怎么优化都是于事无补的。主要影响服务器的速度

有：网络-硬盘读写速度-内存大小-cpu处理速度。

就Web服务器而言，Apache打开了越多的连接进程，CPU需要处理的上下文切换也越多，额外增加了CPU的消耗，然后就直接导致平均响应时间增加。因此上述的MaxClient数目，要根据CPU、内存等硬件因素综合考虑，绝对不是越多越好。可以通过Apache自带的abench来测试一下，取一个合适的值。然后，我们选择内存操作级别的存储的Redis，在高并发的状态下，存储的响应时间至关重要。网络带宽虽然也是一个因素，不过，这种请求数据包一般比较小，一般很少成为请求的瓶颈。负载均衡成为系统瓶颈的情况比较少，在这里不做讨论哈。

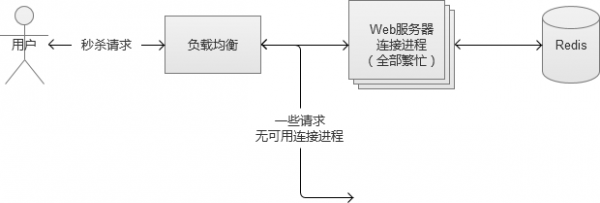
那么问题来了，假设我们的系统，在5w/s的高并发状态下，平均响应时间从100ms变为250ms（实际情况，甚至更多）：

20\*500/0.25 = 40000 （4万QPS）

于是，我们的系统剩下了4w的QPS，面对5w每秒的请求，中间相差了1w。

举个例子，高速路口，1秒钟来5部车，每秒通过5部车，高速路口运作正常。突然，这个路口1秒钟只能通过4部车，车流量仍然依旧，结果必定出现大塞车。（5条车道忽然变成4条车道的感觉）

同理，某一个秒内，20\*500个可用连接进程都在满负荷工作中，却仍然有1万个新来请求，没有连接进程可用，系统陷入到异常状态也是预期之内。



其实在正常的非高并发的业务场景中，也有类似的情况出现，某个业务请求接口出现问题，响应时间极慢，将整个Web请求响应时间拉得很长，逐渐将Web服务器的可用连接数占满，其他正常的业务请求，无连接进程可用。

更可怕的问题是，是用户的行为特点，系统越是不可用，用户的点击越频繁，恶性循环最终导致“雪崩”（其中一台Web机器挂了，导致流量分散到其他正常工作的机器上，再导致正常的机器也挂，然后恶性循环），将整个Web系统拖垮。

3. 重启与过载保护

如果系统发生“雪崩”，贸然重启服务，是无法解决问题的。最常见的现象是，启动起来后，立刻挂掉。这个时候，最好在入口层将流量拒绝，然后再将重启。如果是redis/memcache这种服务也挂了，重启的时候需要注意“预热”，并且很可能需要比较长的时间。

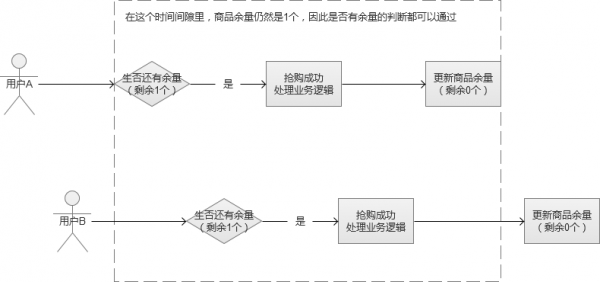
秒杀和抢购的场景，流量往往是超乎我们系统的准备和想象的。这个时候，过载保护是必要的。如果检测到系统满负载状态，拒绝请求也是一种保护措施。在前端设置过滤是最简单的方式，但是，这种做法是被用户“千夫所指”的行为。更合适一点的是，将过载保护设置在CGI入口层，快速将客户的直接请求返回

高并发下的数据安全

我们知道在多线程写入同一个文件的时候，会存现“线程安全”的问题（多个线程同时运行同一段代码，如果每次运行结果和单线程运行的结果是一样的，结果和预期相同，就是线程安全的）。如果是MySQL数据库，可以使用它自带的锁机制很好的解决问题，但是，在大规模并发的场景中，是不推荐使用MySQL的。秒杀和抢购的场景中，还有另外一个问题，就是“超发”，如果在这方面控制不慎，会产生发送过多的情况。我们也曾经听说过，某些电商搞抢购活动，买家成功拍下后，商家却不承认订单有效，拒绝发货。这里的问题，也许并不一定是商家奸诈，而是系统技术层面存在超发风险导致的。

1. 超发的原因

假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。（同文章前面说的场景）



在上面的这个图中，就导致了并发用户B也“抢购成功”，多让一个人获得了商品。这种场景，在高并发的情况下非常容易出现。

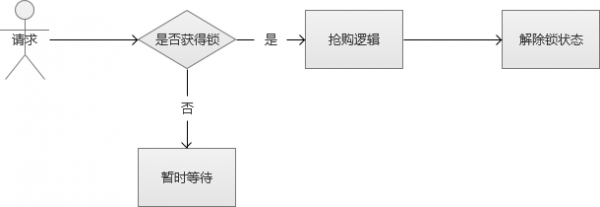
优化方案1：将库存字段number字段设为unsigned，当库存为0时，因为字段不能为负数，将会返回false

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | <?php  //优化方案1：将库存字段number字段设为unsigned，当库存为0时，因为字段不能为负数，将会返回false  include('./mysql.php');  $username = 'wang'.rand(0,1000);  //生成唯一订单  function build\_order\_no(){    return date('ymd').substr(implode(NULL, array\_map('ord', str\_split(substr(uniqid(), 7, 13), 1))), 0, 8);  }  //记录日志  function insertLog($event,$type=0,$username){      global $conn;      $sql="insert into ih\_log(event,type,usernma)      values('$event','$type','$username')";      return mysqli\_query($conn,$sql);  }  function insertOrder($order\_sn,$user\_id,$goods\_id,$sku\_id,$price,$username,$number)  {        global $conn;        $sql="insert into ih\_order(order\_sn,user\_id,goods\_id,sku\_id,price,username,number)        values('$order\_sn','$user\_id','$goods\_id','$sku\_id','$price','$username','$number')";       return  mysqli\_query($conn,$sql);  }  //模拟下单操作  //库存是否大于0  $sql="select number from ih\_store where goods\_id='$goods\_id' and sku\_id='$sku\_id' ";  $rs=mysqli\_query($conn,$sql);  $row = $rs->fetch\_assoc();    if($row['number']>0){//高并发下会导致超卖        if($row['number']<$number){          return insertLog('库存不够',3,$username);        }        $order\_sn=build\_order\_no();        //库存减少        $sql="update ih\_store set number=number-{$number} where sku\_id='$sku\_id' and number>0";        $store\_rs=mysqli\_query($conn,$sql);        if($store\_rs){            //生成订单            insertOrder($order\_sn,$user\_id,$goods\_id,$sku\_id,$price,$username,$number);            insertLog('库存减少成功',1,$username);        }else{            insertLog('库存减少失败',2,$username);        }    }else{        insertLog('库存不够',3,$username);    }  ?> |

2. 悲观锁思路

解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。

悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。



虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

优化方案2：使用MySQL的事务，锁住操作的行

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | <?php  //优化方案2：使用MySQL的事务，锁住操作的行  include('./mysql.php');  //生成唯一订单号  function build\_order\_no(){    return date('ymd').substr(implode(NULL, array\_map('ord', str\_split(substr(uniqid(), 7, 13), 1))), 0, 8);  }  //记录日志  function insertLog($event,$type=0){      global $conn;      $sql="insert into ih\_log(event,type)      values('$event','$type')";      mysqli\_query($conn,$sql);  }  //模拟下单操作  //库存是否大于0  mysqli\_query($conn,"BEGIN");  //开始事务  $sql="select number from ih\_store where goods\_id='$goods\_id' and sku\_id='$sku\_id' FOR UPDATE";//此时这条记录被锁住,其它事务必须等待此次事务提交后才能执行  $rs=mysqli\_query($conn,$sql);  $row=$rs->fetch\_assoc();  if($row['number']>0){      //生成订单      $order\_sn=build\_order\_no();      $sql="insert into ih\_order(order\_sn,user\_id,goods\_id,sku\_id,price)      values('$order\_sn','$user\_id','$goods\_id','$sku\_id','$price')";      $order\_rs=mysqli\_query($conn,$sql);      //库存减少      $sql="update ih\_store set number=number-{$number} where sku\_id='$sku\_id'";      $store\_rs=mysqli\_query($conn,$sql);      if($store\_rs){        echo '库存减少成功';          insertLog('库存减少成功');          mysqli\_query($conn,"COMMIT");//事务提交即解锁      }else{        echo '库存减少失败';          insertLog('库存减少失败');      }  }else{    echo '库存不够';      insertLog('库存不够');      mysqli\_query($conn,"ROLLBACK");  }  ?> |

3. FIFO队列思路

那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉哈。



然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。

4. 文件锁的思路

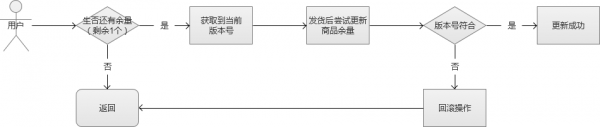
对于日IP不高或者说并发数不是很大的应用，一般不用考虑这些！用一般的文件操作方法完全没有问题。但如果并发高，在我们对文件进行读写操作时，很有可能多个进程对进一文件进行操作，如果这时不对文件的访问进行相应的独占，就容易造成数据丢失

优化方案4：使用非阻塞的文件排他锁

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | <?php  //优化方案4：使用非阻塞的文件排他锁  include ('./mysql.php');  //生成唯一订单号  function build\_order\_no(){    return date('ymd').substr(implode(NULL, array\_map('ord', str\_split(substr(uniqid(), 7, 13), 1))), 0, 8);  }  //记录日志  function insertLog($event,$type=0){      global $conn;      $sql="insert into ih\_log(event,type)      values('$event','$type')";      mysqli\_query($conn,$sql);  }  $fp = fopen("lock.txt", "w+");  if(!flock($fp,LOCK\_EX | LOCK\_NB)){      echo "系统繁忙，请稍后再试";      return;  }  //下单  $sql="select number from ih\_store where goods\_id='$goods\_id' and sku\_id='$sku\_id'";  $rs =  mysqli\_query($conn,$sql);  $row = $rs->fetch\_assoc();  if($row['number']>0){//库存是否大于0      //模拟下单操作      $order\_sn=build\_order\_no();      $sql="insert into ih\_order(order\_sn,user\_id,goods\_id,sku\_id,price)      values('$order\_sn','$user\_id','$goods\_id','$sku\_id','$price')";      $order\_rs =  mysqli\_query($conn,$sql);      //库存减少      $sql="update ih\_store set number=number-{$number} where sku\_id='$sku\_id'";      $store\_rs =  mysqli\_query($conn,$sql);      if($store\_rs){        echo '库存减少成功';          insertLog('库存减少成功');          flock($fp,LOCK\_UN);//释放锁      }else{        echo '库存减少失败';          insertLog('库存减少失败');      }  }else{    echo '库存不够';      insertLog('库存不够');  }  fclose($fp);   ?> |

5. 乐观锁思路

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。



有很多软件和服务都“乐观锁”功能的支持，例如Redis中的watch就是其中之一。通过这个实现，我们保证了数据的安全。

优化方案5：Redis中的watch

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | <?php  $redis = new redis();   $result = $redis->connect('127.0.0.1', 6379);   echo $mywatchkey = $redis->get("mywatchkey");  /\*    //插入抢购数据   if($mywatchkey>0)   {       $redis->watch("mywatchkey");    //启动一个新的事务。      $redis->multi();     $redis->set("mywatchkey",$mywatchkey-1);     $result = $redis->exec();     if($result) {        $redis->hSet("watchkeylist","user\_".mt\_rand(1,99999),time());        $watchkeylist = $redis->hGetAll("watchkeylist");          echo "抢购成功！<br/>";          $re = $mywatchkey - 1;          echo "剩余数量：".$re."<br/>";          echo "用户列表：<pre>";          print\_r($watchkeylist);     }else{        echo "手气不好，再抢购！";exit;     }   }else{       // $redis->hSet("watchkeylist","user\_".mt\_rand(1,99999),"12");       //  $watchkeylist = $redis->hGetAll("watchkeylist");          echo "fail！<br/>";          echo ".no result<br/>";          echo "用户列表：<pre>";        //  var\_dump($watchkeylist);   }\*/  $rob\_total = 100;   //抢购数量  if($mywatchkey<=$rob\_total){      $redis->watch("mywatchkey");      $redis->multi(); //在当前连接上启动一个新的事务。      //插入抢购数据      $redis->set("mywatchkey",$mywatchkey+1);      $rob\_result = $redis->exec();      if($rob\_result){           $redis->hSet("watchkeylist","user\_".mt\_rand(1, 9999),$mywatchkey);          $mywatchlist = $redis->hGetAll("watchkeylist");          echo "抢购成功！<br/>";            echo "剩余数量：".($rob\_total-$mywatchkey-1)."<br/>";          echo "用户列表：<pre>";          var\_dump($mywatchlist);      }else{            $redis->hSet("watchkeylist","user\_".mt\_rand(1, 9999),'meiqiangdao');          echo "手气不好，再抢购！";exit;      }  }  ?> |

PHP解决网站大数据大流量与高并发

第一个要说的就是数据库，首先要有一个很好的架构，查询尽量不用\* 避免相关子查询 给经常查询的添加索引 用排序来取代非顺序存取，如果条件允许 ，一般MySQL服务器最好安装在Linux操作系统中 。关于apache和nginx在高并发的情况下推荐使用nginx，ginx是Apache服务器不错的替代品。nginx内存消耗少 官方测试能够支撑5万并发连接，在实际生产环境中跑到2～3万并发连接数。php方面不需要的模块尽量关闭，使用memcached，Memcached 是一个高性能的分布式内存对象缓存系统，不使用数据库直接从内存当中调数据，这样大大提升了速度，iiS或Apache启用GZIP压缩优化网站，压缩网站内容大大节省网站流量。

第二，禁止外部的盗链。

外部网站的图片或者文件盗链往往会带来大量的负载压力，因此应该严格限制外部对  
于自身的图片或者文件盗链，好在目前可以简单地通过refer来控制盗链，Apache自  
己就可以通过配置来禁止盗链，IIS也有一些第三方的ISAPI可以实现同样的功能。当  
然，伪造refer也可以通过代码来实现盗链，不过目前蓄意伪造refer盗链的还不多，  
可以先不去考虑，或者使用非技术手段来解决，比如在图片上增加水印。

第三，控制大文件的下载。

大文件的下载会占用很大的流量，并且对于非SCSI硬盘来说，大量文件下载会消耗  
CPU，使得网站响应能力下降。因此，尽量不要提供超过2M的大文件下载，如果需要  
提供，建议将大文件放在另外一台服务器上。

第四，使用不同主机分流主要流量

将文件放在不同的主机上，提供不同的镜像供用户下载。比如如果觉得RSS文件占用  
流量大，那么使用FeedBurner或者FeedSky等服务将RSS输出放在其他主机上，这  
样别人访问的流量压力就大多集中在FeedBurner的主机上，RSS就不占用太多资源了

第五，使用不同主机分流主要流量  
将文件放在不同的主机上，提供不同的镜像供用户下载。比如如果觉得RSS文件占用流量大，那么使用FeedBurner或者FeedSky等服务将RSS输出放在其他主机上，这样别人访问的流量压力就大多集中在FeedBurner的主机上，RSS就不占用太多资源了。

第六，使用流量分析统计软件。  
在网站上安装一个流量分析统计软件，可以即时知道哪些地方耗费了大量流量，哪些页面需要再进行优化，因此，解决流量问题还需要进行精确的统计分析才可以。比如：Google Analytics（Google分析）。

高并发和高负载的约束条件：硬件、部署、操作系统、Web 服务器、PHP、MySQL、测试

部署：服务器分离、数据库集群和库表散列、镜像、负载均衡

负载均衡分类： 1)、DNS轮循 2）代理服务器负载均衡 3）地址转换网关负载均衡 4）NAT负载均衡 5）反向代理负载均衡 6）混合型负载均衡

部署方案1：

适用范围：静态内容为主体的网站和应用系统；对系统安全要求较高的网站和应用系统。

Main Server：主服务器

承载程序的主体运行压力，处理网站或应用系统中的动态请求；

将静态页面推送至多个发布服务器；

将附件文件推送至文件服务器；

安全要求较高，以静态为主的网站，可将服务器置于内网屏蔽外网的访问。

DB Server：数据库服务器

承载数据库读写压力；

只与主服务器进行数据量交换，屏蔽外网访问。

File/Video Server：文件/视频服务器

承载系统中占用系统资源和带宽资源较大的数据流；

作为大附件的存储和读写仓库；

作为视频服务器将具备视频自动处理能力。

发布服务器组：

只负责静态页面的发布，承载绝大多数的Web请求；

通过Nginx进行负载均衡部署。

部署方案2：

适用范围：以动态交互内容为主体的网站或应用系统；负载压力较大，且预算比较充足的网站或应用系统；

Web服务器组：

Web服务无主从关系，属平行冗余设计；

通过前端负载均衡设备或Nginx反向代理实现负载均衡；

划分专用文件服务器/视频服务器有效分离轻/重总线；

每台Web服务器可通过DEC可实现连接所有数据库，同时划分主从。

数据库服务器组：

相对均衡的承载数据库读写压力；

通过数据库物理文件的映射实现多数据库的数据同步。

共享磁盘/磁盘阵列

将用于数据物理文件的统一读写

用于大型附件的存储仓库

通过自身物理磁盘的均衡和冗余，确保整体系统的IO效率和数据安全；

方案特性：

通过前端负载均衡，合理分配Web压力；

通过文件/视频服务器与常规Web服务器的分离，合理分配轻重数据流；

通过数据库服务器组，合理分配数据库IO压力；

每台Web服务器通常只连接一台数据库服务器，通过DEC的心跳检测，可在极短时间内自动切换至冗余数据库服务器；

磁盘阵列的引入，大幅提升系统IO效率的同时，极大增强了数据安全性。

Web服务器：

Web服务器很大一部分资源占用来自于处理Web请求，通常情况下这也就是Apache产生的压力，在高并发连接的情况下，Nginx是Apache服务器不错的替代品。Nginx (“engine x”) 是俄罗斯人编写的一款高性能的 HTTP 和反向代理服务器。在国内，已经有新浪、搜狐通行证、网易新闻、网易博客、金山逍遥网、金山爱词霸、校内网、YUPOO相册、豆瓣、迅雷看看等多家网站、 频道使用 Nginx 服务器。

Nginx的优势：

高并发连接：官方测试能够支撑5万并发连接，在实际生产环境中跑到2～3万并发连接数。

内存消耗少：在3万并发连接下，开启的10个Nginx 进程才消耗150M内存（15M\*10=150M）。

内置的健康检查功能：如果 Nginx Proxy 后端的某台 Web 服务器宕机了，不会影响前端访问。

策略：相对于老牌的Apache，我们选择Lighttpd和Nginx这些具有更小的资源占用率和更高的负载能力的web服务器。

Mysql：

MySQL本身具备了很强的负载能力，MySQL优化是一项很复杂的工作，因为这最终需要对系统优化的很好理解。大家都知道数据库工作就是大量的、 短时的查询和读写，除了程序开发时需要注意建立索引、提高查询效率等软件开发技巧之外，从硬件设施的角度影响MySQL执行效率最主要来自于磁盘搜索、磁盘IO水平、CPU周期、内存带宽。

　　根据服务器上的硬件和软件条件进行MySQl优化。MySQL优化的核心在于系统资源的分配，这不等于无限制的给MySQL分配更多的资源。在MySQL配置文件中我们介绍几个最值得关注的参数：

改变索引缓冲区长度(key\_buffer)

改变表长(read\_buffer\_size)

设定打开表的数目的最大值(table\_cache)

对缓长查询设定一个时间限制(long\_query\_time)

如果条件允许 ，一般MySQL服务器最好安装在Linux操作系统中，而不是安装在FreeBSD中。  
策略： MySQL优化需要根据业务系统的数据库读写特性和服务器硬件配置，制定不同的优化方案，并且可以根据需要部署MySQL的主从结构。

PHP：

1、加载尽可能少的模块；

2、如果是在windows平台下，尽可能使用IIS或者Nginx来替代我们平常用的Apache；

3、安装加速器（都是通过缓存php代码预编译的结果和数据库结果来提高php代码的执行速度）  
eAccelerator，eAccelerator是一个自由开放源码php加速器，优化和动态内容缓存，提高了性能php脚本的缓存性能，使得PHP脚本在编译的状态下，对服务器的开销几乎完全消除。

Apc：Alternative PHP Cache（APC）是 PHP 的一个免费公开的优化代码缓存。它用来提供免费，公开并且强健的架构来缓存和优化 PHP 的中间代码。

memcache：memcache是由Danga Interactive开发的，高性能的，分布式的内存对象缓存系统，用于在动态应用中减少数据库负载，提升访问速度。主要机制是通过在内存里维护一个统 一的巨大的hash表，Memcache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

Xcache：国人开发的缓存器，

策略： 为PHP安装加速器。

代理服务器（缓存服务器）：

Squid Cache（简称为Squid）是一个流行的自由软件（GNU通用公共许可证）的代理服务器和Web缓存服务器。Squid有广泛的用途，从作为网页服务器的前置cache服务器缓存相关请求来提高Web服务器的速度，到为一组人共享网络资源而缓存万维网，域名系统和其他网络搜索，到通过过滤流量帮助网络安全，到局域网通过代理网。Squid主要设计用于在Unix一类系统运行。

策略：安装Squid 反向代理服务器，能够大幅度提高服务器效率。

压力测试：压力测试是一种基本的质量保证行为，它是每个重要软件测试工作的一部分。压力测试的基本思路很简单：不是在常规条件下运行手动或自动测试，而是在计算机数量较少或系统资源匮乏的条件下运行测试。通常要进行压力测试的资源包括内部内存、CPU 可用性、磁盘空间和网络带宽等。一般用并发来做压力测试。  
压力测试工具：webbench，ApacheBench等

漏洞测试：在我们的系统中漏洞主要包括：sql注入漏洞，xss跨站脚本攻击等。安全方面还包括系统软件，如操作系统漏洞，mysql、apache等的漏洞，一般可以通过升级来解决。

漏洞测试工具：Acunetix Web Vulnerability Scanner