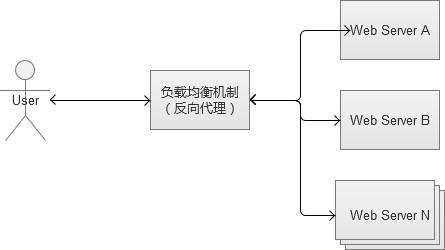
# [亿级Web系统搭建——单机到分布式集群](http://kb.cnblogs.com/page/509402/)

当一个Web系统从日访问量10万逐步增长到1000万，甚至超过1亿的过程中，Web系统承受的压力会越来越大，在这个过程中，我们会遇到很多的问题。为了解决这些性能压力带来问题，我们需要在Web系统架构层面搭建多个层次的缓存机制。在不同的压力阶段，我们会遇到不同的问题，通过搭建不同的服务和架构来解决。

****Web负载均衡****

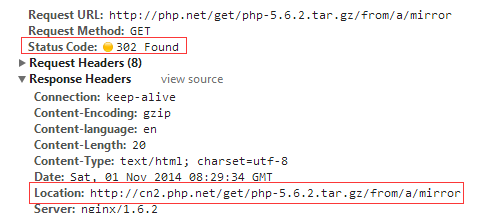
　　Web负载均衡（Load Balancing），简单地说就是给我们的服务器集群分配“工作任务”，而采用恰当的分配方式，对于保护处于后端的Web服务器来说，非常重要。



　　负载均衡的策略有很多，我们从简单的讲起哈。

****1. HTTP重定向****

　　当用户发来请求的时候，Web服务器通过修改HTTP响应头中的Location标记来返回一个新的url，然后浏览器再继续请求这个新url，实际上就是页面重定向。通过重定向，来达到“负载均衡”的目标。例如，我们在下载PHP源码包的时候，点击下载链接时，为了解决不同国家和地域下载速度的问题，它会返回一个离我们近的下载地址。重定向的HTTP返回码是302，如下图：



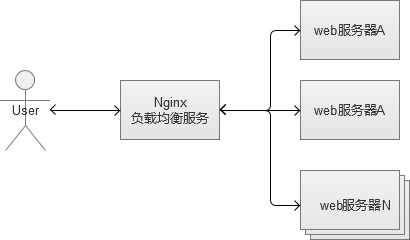
　　如果使用PHP代码来实现这个功能，方式如下：



　　这个重定向非常容易实现，并且可以自定义各种策略。但是，它在大规模访问量下，性能不佳。而且，给用户的体验也不好，实际请求发生重定向，增加了网络延时。

****2. 反向代理负载均衡****

　　反向代理服务的核心工作主要是转发HTTP请求，扮演了浏览器端和后台Web服务器中转的角色。因为它工作在HTTP层（应用层），也就是网络七层结构中的第七层，因此也被称为“七层负载均衡”。可以做反向代理的软件很多，比较常见的一种是Nginx。



　　Nginx是一种非常灵活的反向代理软件，可以自由定制化转发策略，分配服务器流量的权重等。反向代理中，常见的一个问题，就是Web服务器存储的session数据，因为一般负载均衡的策略都是随机分配请求的。同一个登录用户的请求，无法保证一定分配到相同的Web机器上，会导致无法找到session的问题。

　　解决方案主要有两种：

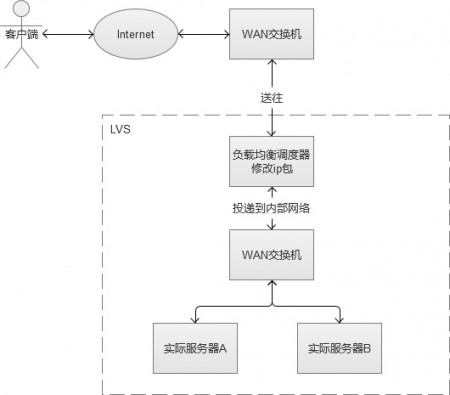
　　1. 配置反向代理的转发规则，让同一个用户的请求一定落到同一台机器上（通过分析cookie），复杂的转发规则将会消耗更多的CPU，也增加了代理服务器的负担。

　　2. 将session这类的信息，专门用某个独立服务来存储，例如redis/memchache，这个方案是比较推荐的。

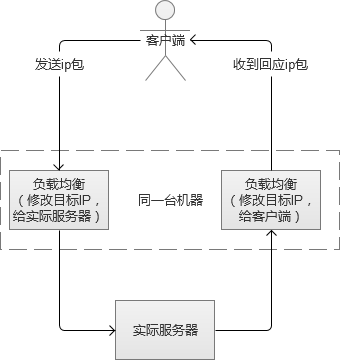
　　反向代理服务，也是可以开启缓存的，如果开启了，会增加反向代理的负担，需要谨慎使用。这种负载均衡策略实现和部署非常简单，而且性能表现也比较好。但是，它有“单点故障”的问题，如果挂了，会带来很多的麻烦。而且，到了后期Web服务器继续增加，它本身可能成为系统的瓶颈。

****3. IP负载均衡****

　　IP负载均衡服务是工作在网络层（修改IP）和传输层（修改端口，第四层），比起工作在应用层（第七层）性能要高出非常多。原理是，他是对IP层的数据包的IP地址和端口信息进行修改，达到负载均衡的目的。这种方式，也被称为“四层负载均衡”。常见的负载均衡方式，是LVS（Linux Virtual Server，Linux虚拟服务），通过IPVS（IP Virtual Server，IP虚拟服务）来实现。



　　在负载均衡服务器收到客户端的IP包的时候，会修改IP包的目标IP地址或端口，然后原封不动地投递到内部网络中，数据包会流入到实际Web服务器。实际服务器处理完成后，又会将数据包投递回给负载均衡服务器，它再修改目标IP地址为用户IP地址，最终回到客户端。



　　上述的方式叫LVS-NAT，除此之外，还有LVS-RD（直接路由），LVS-TUN（IP隧道），三者之间都属于LVS的方式，但是有一定的区别，篇幅问题，不赘叙。

　　IP负载均衡的性能要高出Nginx的反向代理很多，它只处理到传输层为止的数据包，并不做进一步的组包，然后直接转发给实际服务器。不过，它的配置和搭建比较复杂。

****4. DNS负载均衡****

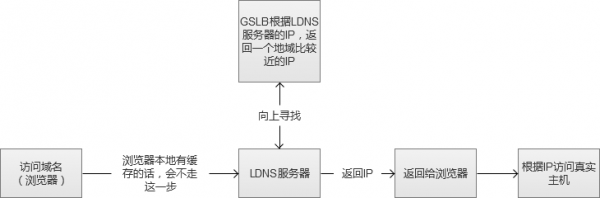
　　DNS（Domain Name System）负责域名解析的服务，域名url实际上是服务器的别名，实际映射是一个IP地址，解析过程，就是DNS完成域名到IP的映射。而一个域名是可以配置成对应多个IP的。因此，DNS也就可以作为负载均衡服务。



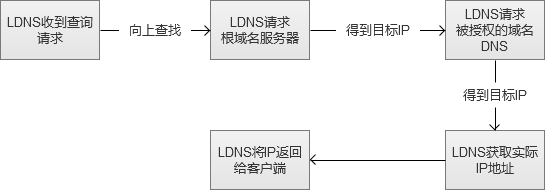
　　这种负载均衡策略，配置简单，性能极佳。但是，不能自由定义规则，而且，变更被映射的IP或者机器故障时很麻烦，还存在DNS生效延迟的问题。

****5. DNS/GSLB负载均衡****

　　我们常用的CDN（Content Delivery Network，内容分发网络）实现方式，其实就是在同一个域名映射为多IP的基础上更进一步，通过GSLB（Global Server Load Balance，全局负载均衡）按照指定规则映射域名的IP。一般情况下都是按照地理位置，将离用户近的IP返回给用户，减少网络传输中的路由节点之间的跳跃消耗。

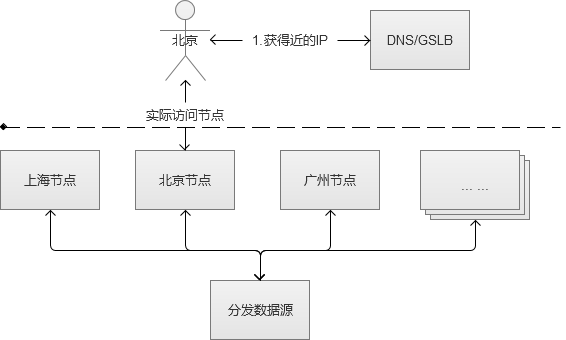


　　图中的“向上寻找”，实际过程是LDNS（Local DNS）先向根域名服务（Root Name Server）获取到顶级根的Name Server（例如.com的），然后得到指定域名的授权DNS，然后再获得实际服务器IP。



　　CDN在Web系统中，一般情况下是用来解决大小较大的静态资源（html/Js/Css/图片等）的加载问题，让这些比较依赖网络下载的内容，尽可能离用户更近，提升用户体验。

　　例如，我访问了一张imgcache.gtimg.cn上的图片（腾讯的自建CDN，不使用qq.com域名的原因是防止http请求的时候，带上了多余的cookie信息），我获得的IP是183.60.217.90。



　　这种方式，和前面的DNS负载均衡一样，不仅性能极佳，而且支持配置多种策略。但是，搭建和维护成本非常高。互联网一线公司，会自建CDN服务，中小型公司一般使用第三方提供的CDN。

****Web系统的缓存机制的建立和优化****

　　刚刚我们讲完了Web系统的外部网络环境，现在我们开始关注我们Web系统自身的性能问题。我们的Web站点随着访问量的上升，会遇到很多的挑战，解决这些问题不仅仅是扩容机器这么简单，建立和使用合适的缓存机制才是根本。

　　最开始，我们的Web系统架构可能是这样的，每个环节，都可能只有1台机器。



　　我们从最根本的数据存储开始看哈。

****一、 MySQL数据库内部缓存使用****

　　MySQL的缓存机制，就从先从MySQL内部开始，下面的内容将以最常见的InnoDB存储引擎为主。

　　1. 建立恰当的索引

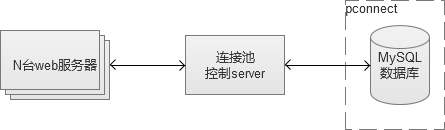
　　最简单的是建立索引，索引在表数据比较大的时候，起到快速检索数据的作用，但是成本也是有的。首先，占用了一定的磁盘空间，其中组合索引最突出，使用需要谨慎，它产生的索引甚至会比源数据更大。其次，建立索引之后的数据insert/update/delete等操作，因为需要更新原来的索引，耗时会增加。当然，实际上我们的系统从总体来说，是以select查询操作居多，因此，索引的使用仍然对系统性能有大幅提升的作用。

　　2. 数据库连接线程池缓存

　　如果，每一个数据库操作请求都需要创建和销毁连接的话，对数据库来说，无疑也是一种巨大的开销。为了减少这类型的开销，可以在MySQL中配置thread\_cache\_size来表示保留多少线程用于复用。线程不够的时候，再创建，空闲过多的时候，则销毁。



　　其实，还有更为激进一点的做法，使用pconnect（数据库长连接），线程一旦创建在很长时间内都保持着。但是，在访问量比较大，机器比较多的情况下，这种用法很可能会导致“数据库连接数耗尽”，因为建立连接并不回收，最终达到数据库的max\_connections（最大连接数）。因此，长连接的用法通常需要在CGI和MySQL之间实现一个“连接池”服务，控制CGI机器“盲目”创建连接数。



　　建立数据库连接池服务，有很多实现的方式，PHP的话，我推荐使用swoole（PHP的一个网络通讯拓展）来实现。

　　3. Innodb缓存设置（innodb\_buffer\_pool\_size）

　　innodb\_buffer\_pool\_size这是个用来保存索引和数据的内存缓存区，如果机器是MySQL独占的机器，一般推荐为机器物理内存的80%。在取表数据的场景中，它可以减少磁盘IO。一般来说，这个值设置越大，cache命中率会越高。

　　4. 分库/分表/分区。

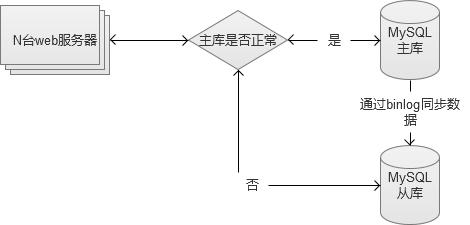
　　MySQL数据库表一般承受数据量在百万级别，再往上增长，各项性能将会出现大幅度下降，因此，当我们预见数据量会超过这个量级的时候，建议进行分库/分表/分区等操作。最好的做法，是服务在搭建之初就设计为分库分表的存储模式，从根本上杜绝中后期的风险。不过，会牺牲一些便利性，例如列表式的查询，同时，也增加了维护的复杂度。不过，到了数据量千万级别或者以上的时候，我们会发现，它们都是值得的。

****二、 MySQL数据库多台服务搭建****

　　1台MySQL机器，实际上是高风险的单点，因为如果它挂了，我们Web服务就不可用了。而且，随着Web系统访问量继续增加，终于有一天，我们发现1台MySQL服务器无法支撑下去，我们开始需要使用更多的MySQL机器。当引入多台MySQL机器的时候，很多新的问题又将产生。

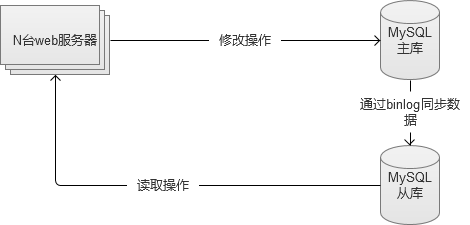
　　1. 建立MySQL主从，从库作为备份

　　这种做法纯粹为了解决“单点故障”的问题，在主库出故障的时候，切换到从库。不过，这种做法实际上有点浪费资源，因为从库实际上被闲着了。



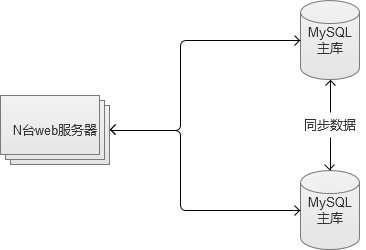
　　2. MySQL读写分离，主库写，从库读。

　　两台数据库做读写分离，主库负责写入类的操作，从库负责读的操作。并且，如果主库发生故障，仍然不影响读的操作，同时也可以将全部读写都临时切换到从库中（需要注意流量，可能会因为流量过大，把从库也拖垮）。



　　3. 主主互备。

　　两台MySQL之间互为彼此的从库，同时又是主库。这种方案，既做到了访问量的压力分流，同时也解决了“单点故障”问题。任何一台故障，都还有另外一套可供使用的服务。



　　不过，这种方案，只能用在两台机器的场景。如果业务拓展还是很快的话，可以选择将业务分离，建立多个主主互备。

****三、 MySQL数据库机器之间的数据同步****

　　每当我们解决一个问题，新的问题必然诞生在旧的解决方案上。当我们有多台MySQL，在业务高峰期，很可能出现两个库之间的数据有延迟的场景。并且，网络和机器负载等，也会影响数据同步的延迟。我们曾经遇到过，在日访问量接近1亿的特殊场景下，出现，从库数据需要很多天才能同步追上主库的数据。这种场景下，从库基本失去效用了。

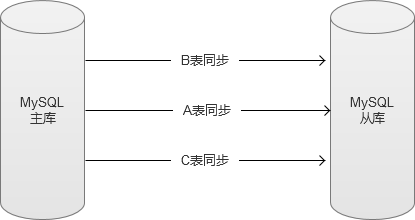
　　于是，解决同步问题，就是我们下一步需要关注的点。

　　1. MySQL自带多线程同步

　　MySQL5.6开始支持主库和从库数据同步，走多线程。但是，限制也是比较明显的，只能以库为单位。MySQL数据同步是通过binlog日志，主库写入到binlog日志的操作，是具有顺序的，尤其当SQL操作中含有对于表结构的修改等操作，对于后续的SQL语句操作是有影响的。因此，从库同步数据，必须走单进程。

　　2. 自己实现解析binlog，多线程写入。

　　以数据库的表为单位，解析binlog多张表同时做数据同步。这样做的话，的确能够加快数据同步的效率，但是，如果表和表之间存在结构关系或者数据依赖的话，则同样存在写入顺序的问题。这种方式，可用于一些比较稳定并且相对独立的数据表。



　　国内一线互联网公司，大部分都是通过这种方式，来加快数据同步效率。还有更为激进的做法，是直接解析binlog，忽略以表为单位，直接写入。但是这种做法，实现复杂，使用范围就更受到限制，只能用于一些场景特殊的数据库中（没有表结构变更，表和表之间没有数据依赖等特殊表）。

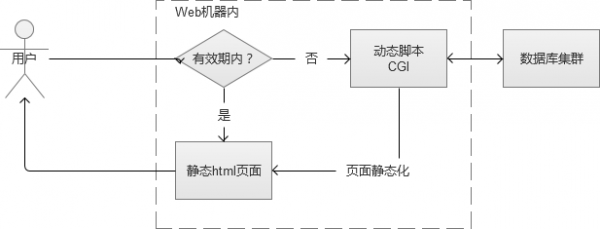
****四、 在Web服务器和数据库之间建立缓存****

　　实际上，解决大访问量的问题，不能仅仅着眼于数据库层面。根据“二八定律”，80%的请求只关注在20%的热点数据上。因此，我们应该建立Web服务器和数据库之间的缓存机制。这种机制，可以用磁盘作为缓存，也可以用内存缓存的方式。通过它们，将大部分的热点数据查询，阻挡在数据库之前。



****1. 页面静态化****

　　用户访问网站的某个页面，页面上的大部分内容在很长一段时间内，可能都是没有变化的。例如一篇新闻报道，一旦发布几乎是不会修改内容的。这样的话，通过CGI生成的静态html页面缓存到Web服务器的磁盘本地。除了第一次，是通过动态CGI查询数据库获取之外，之后都直接将本地磁盘文件返回给用户。

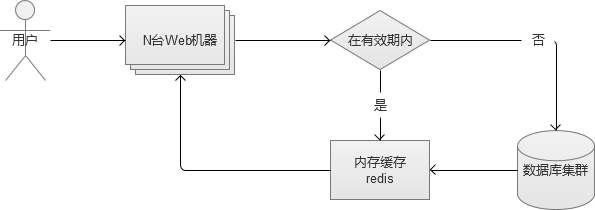


　　在Web系统规模比较小的时候，这种做法看似完美。但是，一旦Web系统规模变大，例如当我有100台的Web服务器的时候。那样这些磁盘文件，将会有100份，这个是资源浪费，也不好维护。这个时候有人会想，可以集中一台服务器存起来，呵呵，不如看看下面一种缓存方式吧，它就是这样做的。

****2. 单台内存缓存****

　　通过页面静态化的例子中，我们可以知道将“缓存”搭建在Web机器本机是不好维护的，会带来更多问题（实际上，通过PHP的apc拓展，可通过Key/value操作Web服务器的本机内存）。因此，我们选择搭建的内存缓存服务，也必须是一个独立的服务。

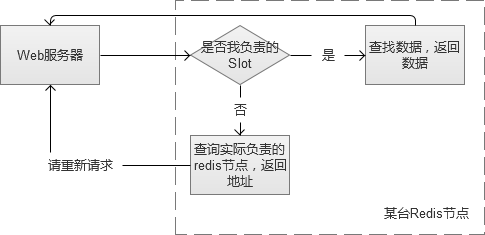
　　内存缓存的选择，主要有redis/memcache。从性能上说，两者差别不大，从功能丰富程度上说，Redis更胜一筹。



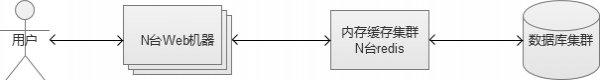
****3. 内存缓存集群****

　　当我们搭建单台内存缓存完毕，我们又会面临单点故障的问题，因此，我们必须将它变成一个集群。简单的做法，是给他增加一个slave作为备份机器。但是，如果请求量真的很多，我们发现cache命中率不高，需要更多的机器内存呢？因此，我们更建议将它配置成一个集群。例如，类似redis cluster。

　　Redis cluster集群内的Redis互为多组主从，同时每个节点都可以接受请求，在拓展集群的时候比较方便。客户端可以向任意一个节点发送请求，如果是它的“负责”的内容，则直接返回内容。否则，查找实际负责Redis节点，然后将地址告知客户端，客户端重新请求。



　　对于使用缓存服务的客户端来说，这一切是透明的。



　　内存缓存服务在切换的时候，是有一定风险的。从A集群切换到B集群的过程中，必须保证B集群提前做好“预热”（B集群的内存中的热点数据，应该尽量与A集群相同，否则，切换的一瞬间大量请求内容，在B集群的内存缓存中查找不到，流量直接冲击后端的数据库服务，很可能导致数据库宕机）。

****4. 减少数据库“写”****

　　上面的机制，都实现减少数据库的“读”的操作，但是，写的操作也是一个大的压力。写的操作，虽然无法减少，但是可以通过合并请求，来起到减轻压力的效果。这个时候，我们就需要在内存缓存集群和数据库集群之间，建立一个修改同步机制。

　　先将修改请求生效在cache中，让外界查询显示正常，然后将这些sql修改放入到一个队列中存储起来，队列满或者每隔一段时间，合并为一个请求到数据库中更新数据库。

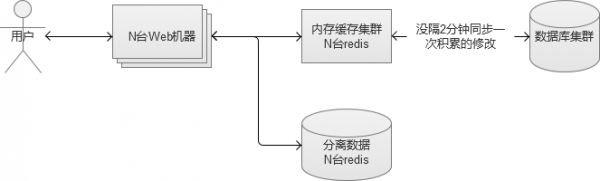


　　除了上述通过改变系统架构的方式提升写的性能外，MySQL本身也可以通过配置参数innodb\_flush\_log\_at\_trx\_commit来调整写入磁盘的策略。如果机器成本允许，从硬件层面解决问题，可以选择老一点的RAID（Redundant Arrays of independent Disks，磁盘列阵）或者比较新的SSD（Solid State Drives，固态硬盘）。

****5. NoSQL存储****

　　不管数据库的读还是写，当流量再进一步上涨，终会达到“人力有穷时”的场景。继续加机器的成本比较高，并且不一定可以真正解决问题的时候。这个时候，部分核心数据，就可以考虑使用NoSQL的数据库。NoSQL存储，大部分都是采用key-value的方式，这里比较推荐使用上面介绍过Redis，Redis本身是一个内存cache，同时也可以当做一个存储来使用，让它直接将数据落地到磁盘。

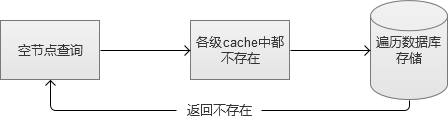
　　这样的话，我们就将数据库中某些被频繁读写的数据，分离出来，放在我们新搭建的Redis存储集群中，又进一步减轻原来MySQL数据库的压力，同时因为Redis本身是个内存级别的Cache，读写的性能都会大幅度提升。



　　国内一线互联网公司，架构上采用的解决方案很多是类似于上述方案，不过，使用的cache服务却不一定是Redis，他们会有更丰富的其他选择，甚至根据自身业务特点开发出自己的NoSQL服务。

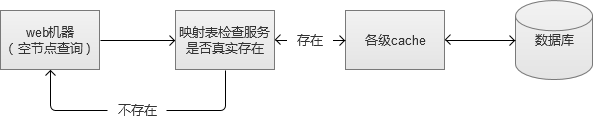
****6. 空节点查询问题****

　　当我们搭建完前面所说的全部服务，认为Web系统已经很强的时候。我们还是那句话，新的问题还是会来的。空节点查询，是指那些数据库中根本不存在的数据请求。例如，我请求查询一个不存在人员信息，系统会从各级缓存逐级查找，最后查到到数据库本身，然后才得出查找不到的结论，返回给前端。因为各级cache对它无效，这个请求是非常消耗系统资源的，而如果大量的空节点查询，是可以冲击到系统服务的。



　　在我曾经的工作经历中，曾深受其害。因此，为了维护Web系统的稳定性，设计适当的空节点过滤机制，非常有必要。

　　我们当时采用的方式，就是设计一张简单的记录映射表。将存在的记录存储起来，放入到一台内存cache中，这样的话，如果还有空节点查询，则在缓存这一层就被阻挡了。



****异地部署（地理分布式）****

　　完成了上述架构建设之后，我们的系统是否就已经足够强大了呢？答案当然是否定的哈，优化是无极限的。Web系统虽然表面上看，似乎比较强大了，但是给予用户的体验却不一定是最好的。因为东北的同学，访问深圳的一个网站服务，他还是会感到一些网络距离上的慢。这个时候，我们就需要做异地部署，让Web系统离用户更近。

****一、 核心集中与节点分散****

　　有玩过大型网游的同学都会知道，网游是有很多个区的，一般都是按照地域来分，例如广东专区，北京专区。如果一个在广东的玩家，去北京专区玩，那么他会感觉明显比在广东专区卡。实际上，这些大区的名称就已经说明了，它的服务器所在地，所以，广东的玩家去连接地处北京的服务器，网络当然会比较慢。

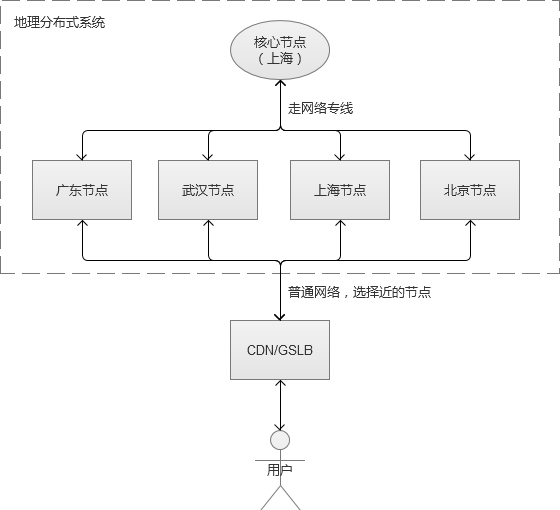
　　当一个系统和服务足够大的时候，就必须开始考虑异地部署的问题了。让你的服务，尽可能离用户更近。我们前面已经提到了Web的静态资源，可以存放在CDN上，然后通过DNS/GSLB的方式，让静态资源的分散“全国各地”。但是，CDN只解决的静态资源的问题，没有解决后端庞大的系统服务还只集中在某个固定城市的问题。

　　这个时候，异地部署就开始了。异地部署一般遵循：核心集中，节点分散。

　　1. 核心集中：实际部署过程中，总有一部分的数据和服务存在不可部署多套，或者部署多套成本巨大。而对于这些服务和数据，就仍然维持一套，而部署地点选择一个地域比较中心的地方，通过网络内部专线来和各个节点通讯。

　　2. 节点分散：将一些服务部署为多套，分布在各个城市节点，让用户请求尽可能选择近的节点访问服务。

　　例如，我们选择在上海部署为核心节点，北京，深圳，武汉，上海为分散节点（上海自己本身也是一个分散节点）。我们的服务架构如图：



　　需要补充一下的是，上图中上海节点和核心节点是同处于一个机房的，其他分散节点各自独立机房。

　　国内有很多大型网游，都是大致遵循上述架构。它们会把数据量不大的用户核心账号等放在核心节点，而大部分的网游数据，例如装备、任务等数据和服务放在地区节点里。当然，核心节点和地域节点之间，也有缓存机制。

****二、 节点容灾和过载保护****

　　节点容灾是指，某个节点如果发生故障时，我们需要建立一个机制去保证服务仍然可用。毫无疑问，这里比较常见的容灾方式，是切换到附近城市节点。假如系统的天津节点发生故障，那么我们就将网络流量切换到附近的北京节点上。考虑到负载均衡，可能需要同时将流量切换到附近的几个地域节点。另一方面，核心节点自身也是需要自己做好容灾和备份的，核心节点一旦故障，就会影响全国服务。

　　过载保护，指的是一个节点已经达到最大容量，无法继续接接受更多请求了，系统必须有一个保护的机制。一个服务已经满负载，还继续接受新的请求，结果很可能就是宕机，影响整个节点的服务，为了至少保障大部分用户的正常使用，过载保护是必要的。

　　解决过载保护，一般2个方向：

　　1. 拒绝服务，检测到满负载之后，就不再接受新的连接请求。例如网游登入中的排队。

　　2. 分流到其他节点。这种的话，系统实现更为复杂，又涉及到负载均衡的问题。

****小结****

　　Web系统会随着访问规模的增长，渐渐地从1台服务器可以满足需求，一直成长为“庞然大物”的大集群。而这个Web系统变大的过程，实际上就是我们解决问题的过程。在不同的阶段，解决不同的问题，而新的问题又诞生在旧的解决方案之上。

　　系统的优化是没有极限的，软件和系统架构也一直在快速发展，新的方案解决了老的问题，同时也带来新的挑战。