大型网站设计架构

> 读 *大型网站设计架构 核心原理与案例分析* 思考体会

一个好的网站架构需要考虑哪些东西：

1. 性能。性能是评判网站架构最核心的标准，这会直观的反应到用户的实际体验上，如果不能带给用户良好的体验，那么一切都是空谈。
2. 可用性。故障无处不在：硬件故障、程序 BUG、自然灾害...高可用意味着你的网站在大多数情况下（即使某台服务器出现故障）都能正常提供服务，一个高可用的大型网站至少要保证"两个九"，即一年中 99% 的时间都是可以正常访问的。
3. 伸缩性。伸缩性是指网站的资源（例如服务器数量）能够随时调整，一个良好的伸缩性是指当一个网站资源增加时，其性能也会得到线性增长。
4. 扩展性。扩展性指的是功能扩展，任何一个产品都需要不断升级迭代，推出新的功能，一个良好的扩展性意味着网站设计业务之间耦合度低，新增业务对已有业务不会造成太大的影响。否则，任何一次升级可能都需要程序员加班加点检查各处的依赖，整个网站臃肿不堪，产品迭代效率缓慢，在市场上只能是处于被淘汰的那一方。
5. 安全性。一个大型网站必然会遭受到各种各样的攻击，如果一个网站随时会被攻击瘫痪，试问谁会使用这个网站？

网站架构的系统层次如下图所示：

图 A.1　网站系统架构层次

想要设计一个良好的架构，必须要考虑到图中几点层次架构：

1. 前端架构：展现给客户，由游览器渲染。
2. 应用层架构：供前端访问的业务服务。
3. 服务层架构：基础服务，如一些中间件等；或抽离出的可复用的基础服务，如 SSO 登录模块。
4. 存储层架构：持久化服务，如数据库。
5. 后台架构：离线任务，如定时任务、搜索任务、数据仓库等。
6. 数据采集与监控：统一的解决方案，采集数据、监控告警。
7. 安全架构：保护网站免遭攻击及数据泄露。

高性能

评价标准

用户最直观的感受就是 RT 响应时间，即请求一个页面、接口到收到答复并处理（渲染）的时间，任何一个网站在上线前都必须将 RT P99 控制在一个可接受的范围内。

对于程序员而言，关注的可能更多是 QPS（每秒查询数）、TPS（每秒事务数）、最大并发数量、性能计数器（进程数、CPU利用率、内存使用等）等指标。

通常我们会通过发压机对不同并发数目下判断系统的总体承受能力。

并 发 数	响应时间（ms）	TPS	错误率（%）	Load	内 存（GB）	备　注
10	500	20	0	5	8	性能测试
20	800	30	0	10	10	性能测试
30	1000	40	2	15	14	性能测试
40	1200	45	20	30	16	负载测试
60	2000	30	40	50	16	压力测试
80	超时	0	100	不详	不详	压力测试

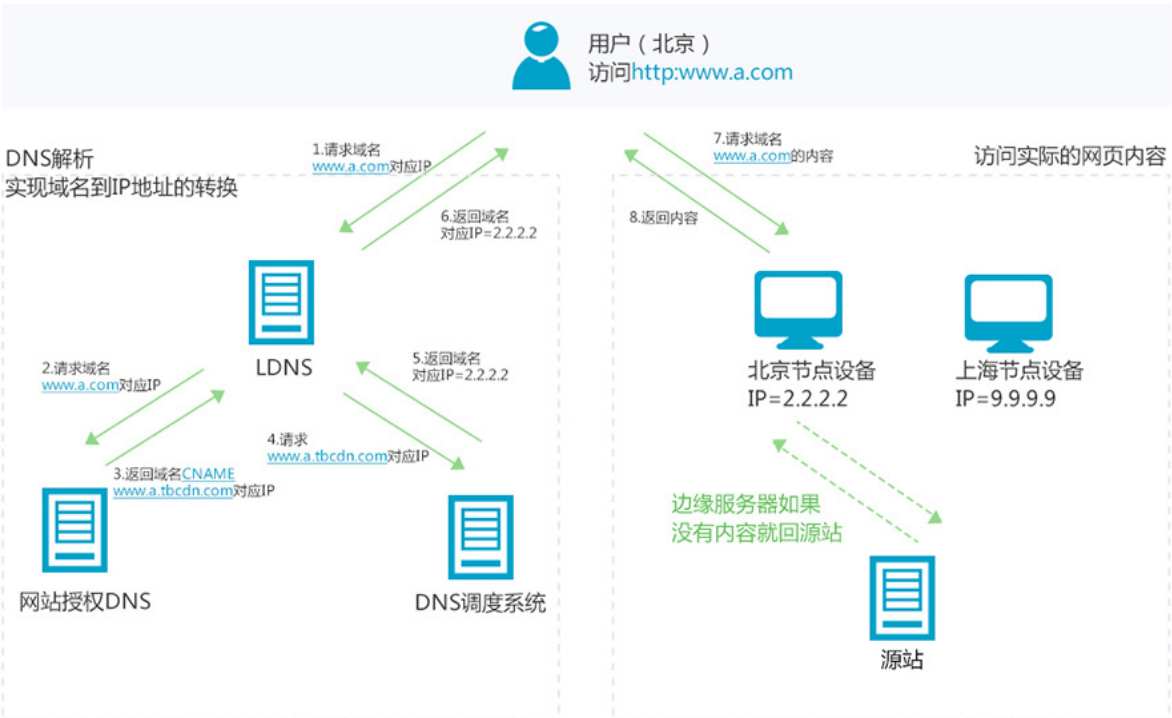
> 并发数发压并不仅仅只是开 N 个线程不停发起请求，每个线程两次请求之间应该还会有些间隔，这个间隔被称为思考时间。
>
> QPS 指每秒查询数，通常仅仅针对 Get 请求；TPS 指每秒事务数，不单单针对查询，是 QPS 的超集。

性能优化

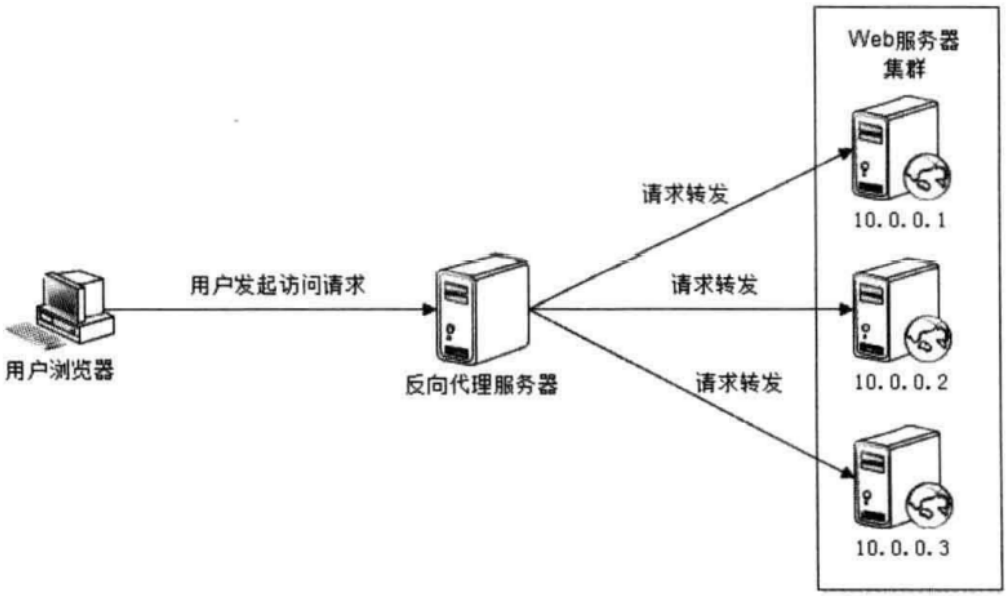
性能优化可以考虑从前端优化、后端优化、以及存储优化。

- 前端优化
  - CDN 加速。使用内容分发网络存储静态资源，开启 CDN 并不需要修改大量源代码，而是通过配置源站域名 CNAME，将源站域名指向 CDN 加速域名，由 CDN 全局负载均衡调度一个最优的 CDN 服务器给用户，用户直接请求 CDN 代理服务器。

    > 在某些前后端同源的项目下，请求后台接口同样会走 CDN 服务器代理，但动态资源不需要缓存，可以约定 Content-Type 类型决定是否需要缓存。



- 反向代理。使用 Nginx 或其他高性能反向代理服务器进行代理，降低单个服务器压力，缺点是反向代理服务器压力可能会比较大。



- 资源压缩。可以将传输内容、页面资源等资源压缩，减少网络传输时间。
- 资源缓存。游览器也可以缓存某些资源。
- 减少 HTTP 请求。HTTP 1.0 版本是无状态的，任何一次请求都需要打开额外的链接，可以考虑将多个请求合并。

- 后端优化
  - 使用缓存。缓存可以减少对数据库或其他存储服务的压力，但是单台服务器上的缓存可能会存在各种各样的问题，例如：带有状态、内存不足等，因此可以考虑使用分布式缓存集群，如 MemCache、Redis 等。
  - 使用集群。使用集群可以降低单台服务器的压力，使系统整体负载能力增强，并且多台廉价服务器比一台昂贵的高性能服务器划算得多。
  - 异步。某些后台任务可以使用 MQ 完成异步操作，后台提交任务后可以立即响应前端，例如下单后扣减数据库成功后需要做一些其他操作、发送消息通知等。
  - 代码优化。针对特定语言进行特定优化。

- 存储层优化
  - 机械硬盘 VS 固态硬盘（SSD）。机械硬盘通过磁头转动读取信息，顺序读取会比随机读取快上几百倍。固态硬盘没有机械装置，数据被记忆在硅晶体中，可以像内存一样做到随机读取。
  - B+ 树 VS LSM 树。B+ 树以稳定高效的读取速率得到人们的青睐，但 B+ 树在读取或写入时对不同的页面可能会产生大量的随机读写；LSM 是一种顺序的日志合并树，写入读取总是顺序的，当数据较多时会将多个段合并以去除重复数据（类比 AOF 重写），写入非常快，但读取可能不断遍历段以读取值，稳定性不如 B+ 树。在 SSD 技术日益成熟的情况下，也许 B+ 树还会继续大放光彩。

- RAID（冗余廉价磁盘阵列） VS HDFS（分布式文件系统）。RAID 是利用多个磁盘形成条带并发写入获得性能，**HDFS 可以看作是在服务器集群层面上类似实现了 RAID 功能。**HDFS 将文件分割为多个 block，写入会并发写入多个块，而读取也会并发读取然后合并。NameNode 中存储了文件块的一些元数据，NameNode 也负责分配 block，做注册中心、负载均衡使用。

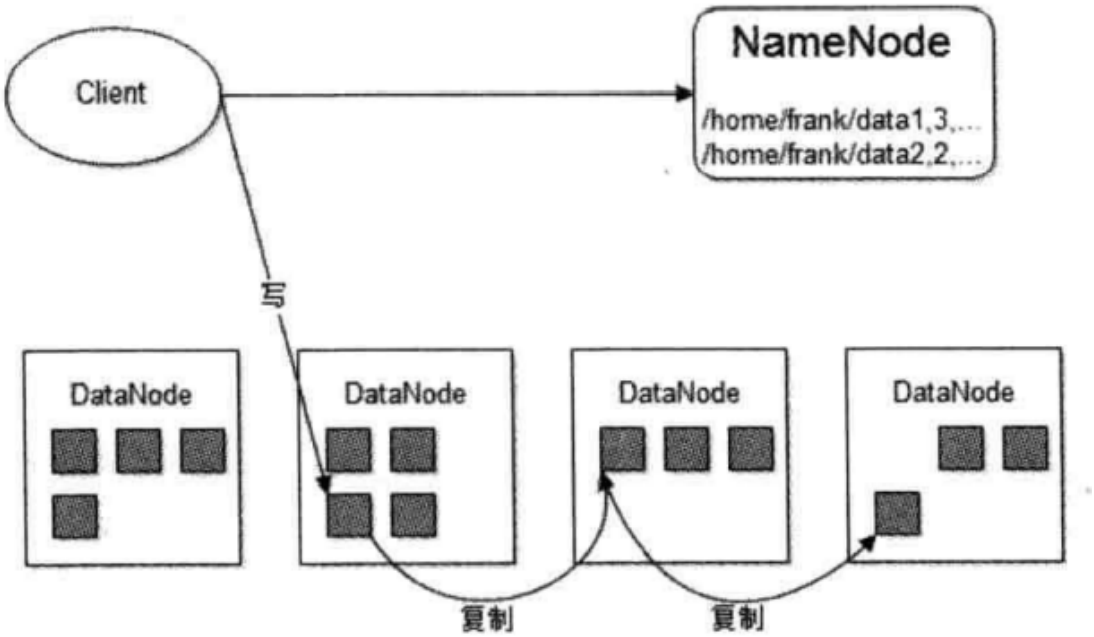


图 4.23　HDFS 架构原理图

高可用

高可用设计

高可用意味着技术任意一台服务器宕机，网站需要保持可用性。CAP 理论告诉我们：P（分区容忍性）是必须要保证的，而 C（数据一致性）和 A（可用性）只能保证一个。

通俗的介绍一下 CAP 理论：

一个分布式系统里面，节点组成的网络本来应该是连通的。然而可能因为一些故障，使得有些节点之间不连通了，整个网络就分成了几块区域。数据就散布在了这些不连通的区域中。这就叫分区。

当你一个数据项只在一个节点中保存，那么分区出现后，和这个节点不连通的部分就访问不到这个数据了。这时分区就是无法容忍的。

提高分区容忍性的办法就是一个数据项复制到多个节点上，那么出现分区之后，这一数据项就可能分布到各个区里，容忍性就提高了。

然而，要把数据复制到多个节点，就会带来一致性的问题，就是多个节点上面的数据可能是不一致的。要保证一致，每次写操作就都要等待全部节点写成功，如果某个节点宕机，

写操作就会无限等待，这又会带来可用性的问题。

如果不考虑可用性，数据一致性又得不到保障。

总的来说就是，数据存在的节点越多，分区容忍性越高，但要复制更新的数据就越多，一致性就越难保证。为了保证一致性，更新所有节点数据所需要的时间就越长，可用性就会降低。这就是 CAP 理论。

例如现在比较流行的 Raft 算法是 AP 的，一旦有半数以上节点宕机，数据就无法正确写入，导致不可用；但 Raft 算法不要求数据同步到所有节点，仅仅需要同步到半数以上节点，因此可用性在大多数时候能够得到保证。Raft 算法是当下最流行的一致性与共识的解决方案。

一个高可用的网站必须包含如下几种措施：

- 数据备份。备份是最让人安心的，备份分为冷备和热备，冷备是指定期将数据归档、备份；热备是指实时进行数据同步。在大型网站中，冷备与热备都是必要的，现在大多数存储层服务如 Mysql 都提供解决方案。
- 服务集群。使用集群即使一台服务器宕机其他服务器也能正常工作。使用集群的前提是服务必须是无状态的，如果某台服务器带有状态，例如保存了用户的购物车，当负载均衡调度其他服务器后，其他服务器并没有保存相关信息。解决方案通常有：
  - 使用 Cookie 或 JWT 让用户保存信息。
  - 利用 Hash 算法绑定服务器，例如一致性哈希算法。
  - 使用分布式缓存（中心化）。
- 故障检测、自动恢复。集群中任意服务器宕机后，要能够检测到故障，并自动进行转移。常用的解决方案有心跳检测、服务器定时推送健康状态，一旦检测到不可用，如果是主从集群，则会自动进行选举；如果是服务集群，则负载均衡调度器将其从列表中移除。
- 服务切分。切分的思想在于避免一个服务出现问题而影响全局，例如淘宝曾经秒杀服务就曾引起整个淘宝业务不可用。纵向切分指按照业务模块将系统分割成不同的业务，不同的业务分离部署，具有不同的优先级；横向切分指将服务分层，如应用层、服务层、存储层。
- 异步调用。使用 MQ 进行异步调用可以保证尽管消费者宕机，消息也不会丢失。如果有多个消费者，也可以避免一个消费者宕机从而导致所有消费者都无法处理。
- 服务降级。分为主动降级和被动降级，主动降级是指提前关闭某些功能，例如淘宝双十一时，提前关闭一些功能以让出服务器资源；被动降级是指当服务不可用时，主动关闭服务防止更多压力达到服务器上或进行降级服务，例如某图片服务器宕机，降级策略会选择将本地一些预留的图片返回。被动降级通常只是暂时的，可以指定一个降级窗口期。
- 数据监控。最好不要等到故障后再来恢复，一个好的网站应该要能够预知到故障发送并提前做出处理。数据监控包括服务端监控与用户数据监控，服务端监控包括监控服务器性能负载，并实时告警；用户数据监听包括收集用户行为、游览器日志收集等，对危险的用户让其进行验证才能够操作。

网站发布

除了服务器的故障之外，发布过程中也可能会导致一些问题，如合并冲突、误操作线上数据，因此一个**自动化的发布流程**也是非常重要的。发布过程中，任意一环都必须要严格遵循，就像火车一样，必须要到达某个站点，每一站都需要检查，不合格的项目下车，最终火车到达终点时，项目也就正常发布了。

一个经典发布过程通常包括：

1. 开发环境程序员自测。
2. 开发分支合入测试分支，冲突检测、代码检测。
3. 测试分支自动化集成测试（例如 Go 的单元测试、Java 的单元测试）。
4. 测试环境自动部署，安全测试、QA 测试，卡点确认。
5. 测试分支合入主干分支，冲突检测、代码检测。
6. 主干分支自动化集成测试。
7. 线上预览环境自动部署，线上预览环境测试，卡点确认。
8. 创建 release 发版分支（或标记 Tag），禁止代码合入。
9. 线上环境发布。

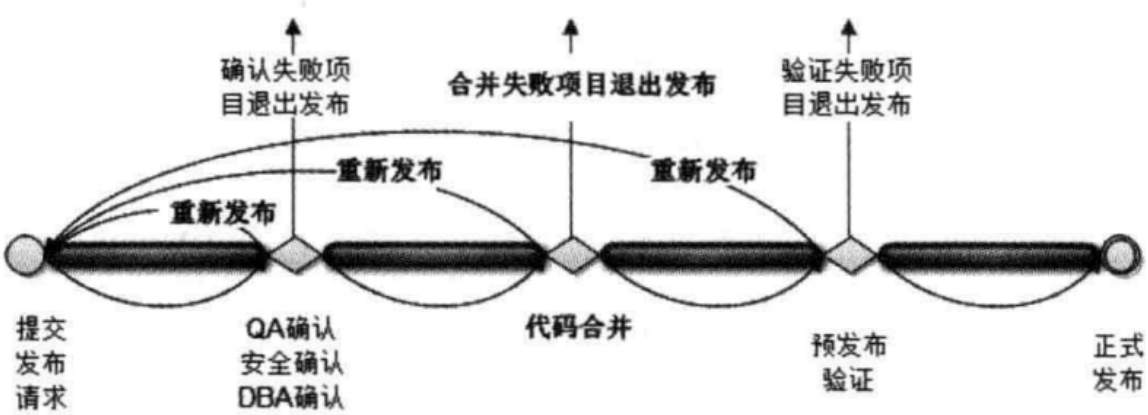


图 5.17　网站火车发布模型

线上发布时常用的发布方式为 **灰度发布（金丝雀发布）**，即新版本流量与旧版本流量并行，持续观察，持续增加新版本流量占比，最终完全发布。

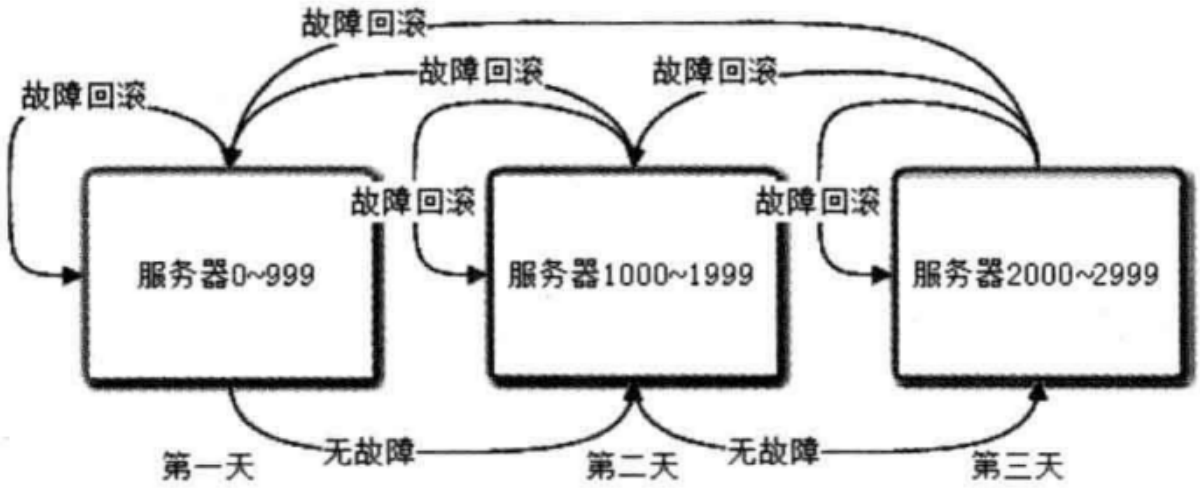


图 5.18　网站灰度发布模型

当然也可以在前端设置一个开关，让用户自行选择体验新旧版本，代价是比较复杂，前端需要维护两套 API。

可伸缩

可伸缩意味着服务器资源可以伸缩调整，同时系统整体处理整体随着线性变化。

通常通过集群方式完成，前面也讲过，应用服务器需设置成无状态的，然后由负载均衡统一调度。

负载均衡方式

- HTTP 重定向。客户端请求负载均衡服务器，服务器根据负载均衡算法返回 302 临时重定向状态码诱导用户重新发起请求，好处是无需响应资源，负载均衡服务器压力不算大，缺点是客户需要发起两次 HTTP 请求，实践中用的不多。
- DNS 域名解析。可以通过为一个域名配置多个 IP 来完成负载均衡的目的。通常应用服务器可能经常会发生变化，因此 DNS 配置也会经常改变，所以在实践中，可能还会加一层抽象，即 DNS -> 反向代理服务器 -> 应用服务器。
- 反向代理。可以配置反向代理服务器进行负载均衡，如 Nginx 可以完成这一目的。
- IP/TCP 层负载均衡。上述负载均衡是运行在 HTTP 层面的，一般称为 7 层负载均衡。IP/TCP 负载均衡一般称为 4 层负载均衡，指当请求到达主机时，通过修改 IP 或端口进行转发，可以类比 NAT 转发，四层负载均衡通常在内核层完成，速度较快。

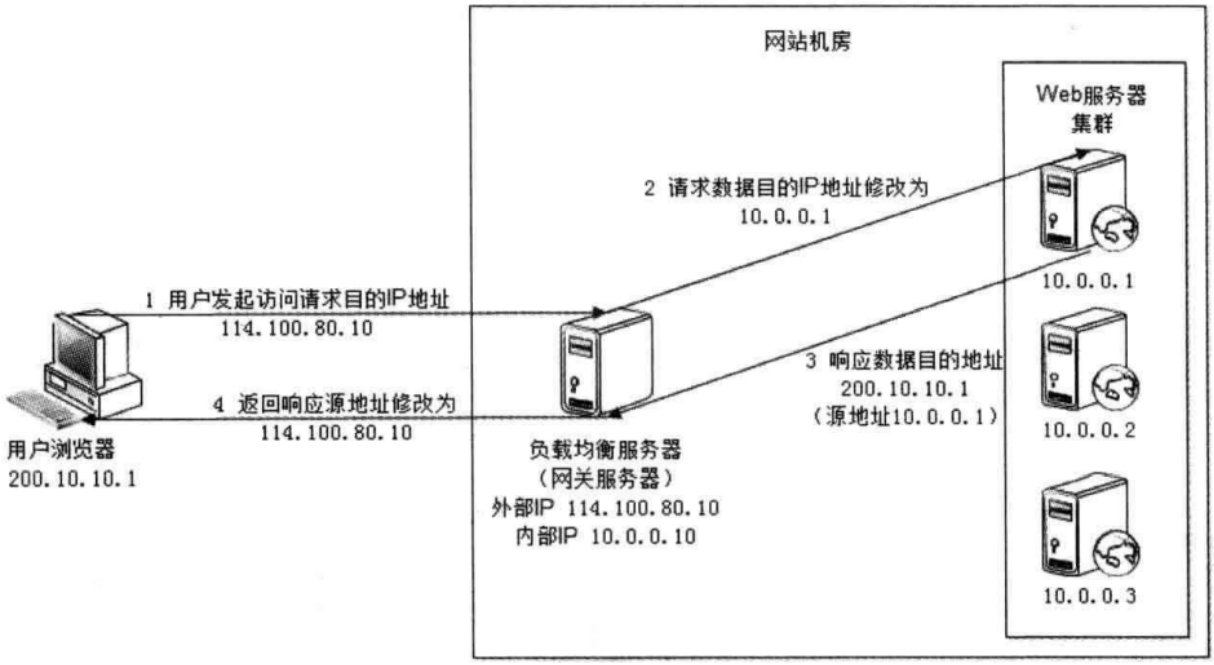


图 6.8　IP 负载均衡原理

- 链路层负载均衡（三级负载均衡）。指仅修改 MAC 地址，而不修改 IP、PORT 进行负载均衡。这种方式需要利用到**逻辑链路捆绑**技术，将多个物理网络逻辑捆绑成一个局域网链路，通过 MAC 地址便可寻址。**而由于源 IP 没有改变，所以服务器的响应会直接答复客户，而不会经过 LVS 负载均衡代理。**

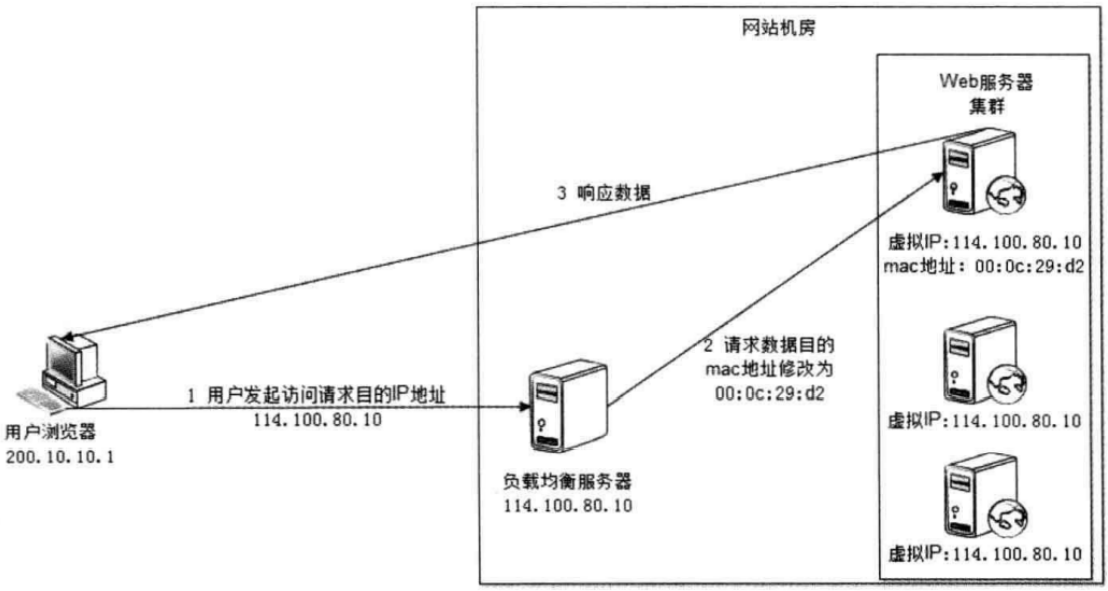


图 6.9　数据链路层负载均衡原理

一致性哈希算法

对于某些带状态的集群服务器，必须要将请求导入到特定的服务器上，否则将无法正常服务，例如带有 session 状态的应用服务器，按 Key 分区的缓存服务器等等...

这时如果新增一台服务器，很可能会导致原先请求落在其他服务器上，典型的哈希算法 `index = hash % n` 会导致 `n / (n + 1)` 错误率发生。

此时可以使用一致性哈希算法，可参考: 分布式系统中的分区问题，一致性哈希算法只会导致少量的请求或数据迁移。

但是在某些时候，少量的数据迁移可能也会导致性能问题，尤其是对于数据库而言。

例如如果对数据库进行了分片操作，将一个表中的数据拆分到多个数据库中去，现在需要新增一个数据库服务器，就算使用了一致性哈希算法，我们也需要去遍历**一个**数据库中的表的全部数据，重新计算哈希并确定是否需要迁移。

这种遍历、计算、迁移操作是十分费时的，而且还需要锁住表。如果一致性哈希中虚拟节点较多的话，需要遍历的表可能会更多。

想要避免部分数据迁移的话，办法是在加一层抽象，**提前规划好未来可能最多的服务器数目**，然后创建分区，在数据库中可以体现为 `schema`，分区的数量要不能小于未来最多的服务器数目，然后一个数据库可以保存多个分区。

由于多了一层抽象，需要一个额外的**注册中心**来记录分区所在的数据库，由于分区数目是固定的，从行记录到分区可以简单的采用取余算法路由，分区到数据库需要查注册中心，最后前往对应数据库搜索对应分区，拿到数据。

这样当我们新增一个数据库时，可以将分区作为一个整体进行**批量迁移**，可以直接同步数据快照，这比数据部分迁移快几百倍，数据库通常也自带 `schema` 同步功能。然后仅需修改注册中心的记录即可。

这是市面上绝大多数分布式数据库的做法。

> 分布式数据库虽然提供较高的伸缩性以及不错的性能，但却无法进行 Join 操作、也无法保障事务，通常需要在业务有意识的避免事务操作。

可扩展

六大设计原则中最重要的一点原则是 **开闭原则，指对扩展开放而对修改关闭**，这意味着当我们需要修改功能或新增功能时，因尽可能的通过扩展的形式嵌入，而尽可能的不修改或是少量修改之前的代码。不要大量修改旧的源代码永远是一句忠告，你并不知道这可能会引起什么样的后果。

> 六大设计原则：开闭原则、接口隔离原则、单一职责原则、里氏替换原则、依赖倒置原则、迪米特法则。

因此网站必须要设计成可扩展的，而可扩展的最核心的要素就是低耦合，只有业务与业务之间耦合程度低，扩展功能才会显得相对容易。

想要完成这个目的最重要的就是拆分与通信：

- 拆分
  - 纵向拆分（竖切）：想象一刀纵向切割，然后切割出多个业务子模块。
  - 横向拆分（横切）：想象一刀横向切割，切分出多个层级，下层可供上层复用。

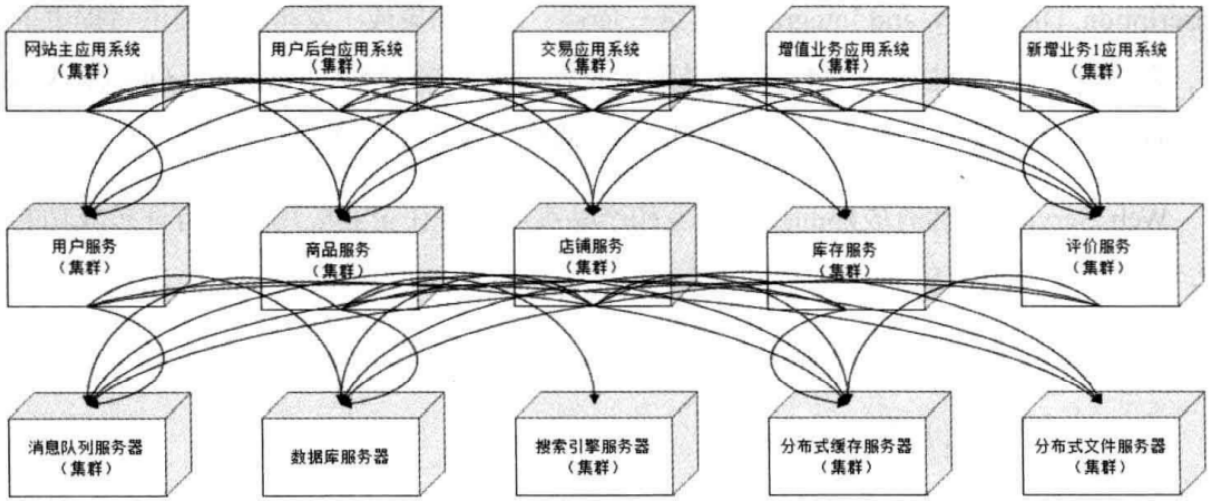
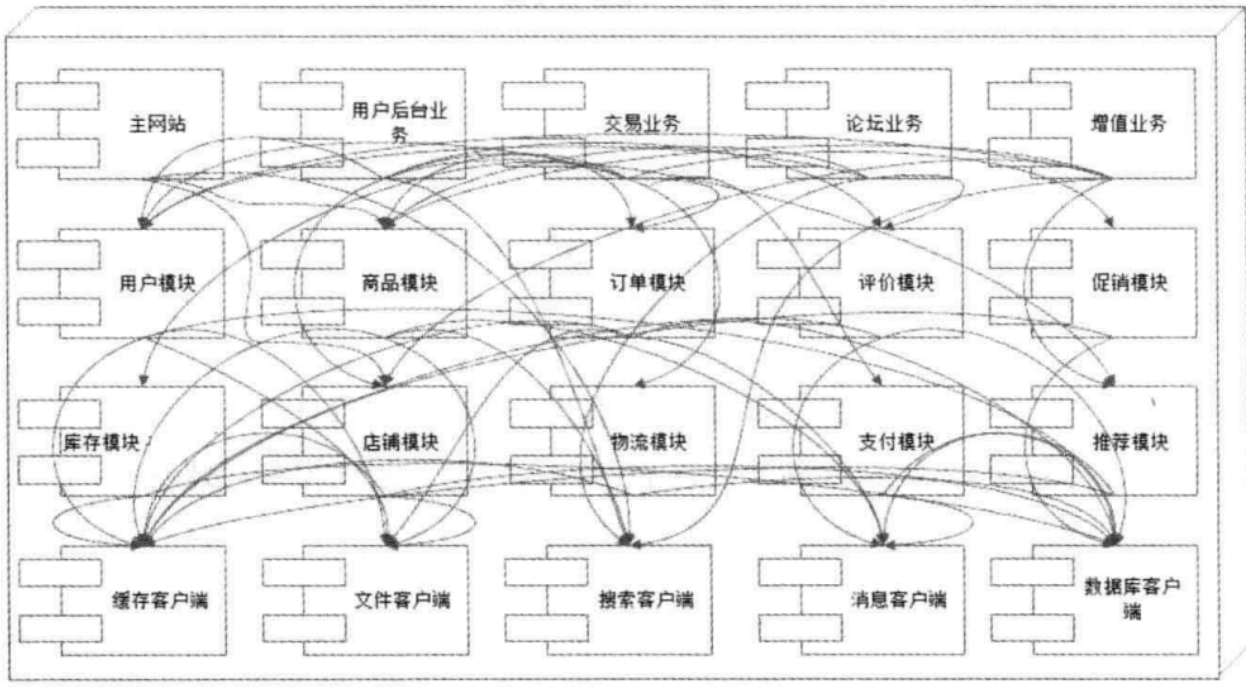


图 7.4　业务及模块拆分独立部署的分布式服务架构

拆分之后横向的服务（一行）之间往往不会互相影响，无论是新增功能还是扩展功能都不会对同层的服务造成太大的影响。而纵向的服务之间存在相互调用，因此升级扩展需要谨慎些，但只要保持接口不变，各层之间还是不会造成太大的影响。

试想，如果是未拆分、高耦合的业务，功能修改、维护起来是非常麻烦的，这通常体现在：

1. 编译、部署困难。构建一个巨型应用耗时多。
2. 分支复杂。多个团队共同维护一个分支，可能会造成大量冲突。
3. 底层服务压力大。如果未进行横向切分，各个服务均调用存储层，很容易导致数据库崩掉。
4. 扩展功能复杂。可能一不小心就踩雷了。



因此拆分是必要的，而拆分带来的坏处就是维护负载、通信负载，抛开维护成本，网络通信将成为系统的主要性能瓶颈。

1. 选择一个好的 RPC 通信协议十分重要。
2. 如果可以异步响应，使用 MQ 会是一个好的选择，MQ 不仅降低了延迟（异步），生产者也无需关注消费者，二者完全没有耦合，非常容易扩展。

除此之外，选择一个好的开发框架也十分重要，一个好的开发框架可以很容易的让你扩展一些功能。

安全性

常见网站应用攻击

1. XSS 攻击（跨站脚本攻击）。此类攻击通常表现为攻击者诱导用户点进某一链接，而此链接中内嵌了一段 JS 代码，导致用户游览器加载并执行脚本。其原理为 HTML 是一种超文本标记语言，通过将一些字符特殊地对待来区别文本和标记，如果动态页面中插入的内容含有一段JavaScript脚本时，这些脚本程序就将会在用户浏览器中执行。

   例如如果你在发布文章时在文章中写了一段 JS 代码，你的文章被渲染时可能就会加载这段 JS 恶意程序。

   为了避免 XSS 攻击，消毒几乎最有效的办法，将用户输入的特殊字符进行转义存储，例如 `<` 被转移为 `<` 等。

2. 注入攻击。注入攻击分为 SQL 注入和 OS 注入，这二者原理是一样的。举个简单的 OS 注入的例子，某网站接受用户上传文件并进行 MD5 校验，程序代码为

`os.systen("md5 " + file_name)`，如果用户上传的文件名为 `$(rm -rf .)`，那么实际执行的命令就会变成 `md5 $(rm -rf .)`，这回导致服务器被删库。

因此在执行命令时一定要小心判断，将用户的参数进行参数绑定，当成一个字符串处理，Shell 脚本中单引号可以无视特殊字符。

3. CSRF 跨域攻击。CSRF 利用的是游览器自带的 Cookie 的机制，攻击者迫使用户访问攻击者的服务器，由于游览器自带 Cookie 访问，这样攻击者就得到的用户的 Cookie 信息，十分危险。

   解决方案通常有：添加 Reference 头部字段、使用 Token 而不是 Cookie、要求用户进行验证、开启同源策略，不允许跨站访问（即使前后端分离部署，也可以通过路径进行反向代理）。

数据安全

数据加密除了使用 HTTPS 等安全的协议对数据传输加密外，在数据库中对敏感数据也应该加盐散列存储。

除此之外，尽量不要将错误内容回显给用户，以防止信息泄露。

除了数据泄露之外，一个安全的网站还应该做到信息过滤与反垃圾，传统的过滤方式为加 N 层过滤器，根据不同的规则进行匹配，但是这么做可能会导致后续规则膨胀，难以管理。

最新的垃圾识别技术是采用机器学习进行数据分类，根据历史已有的数据集进行训练，提取出特征，进行判断。

举个简单的例子，根据历史的数据而言，文本“加入我们”出现在垃圾邮件的概率为 20%，出现在非垃圾邮件中的概率为 5%，这就是一个简单的分类模型，然后对检测邮件提取特征文本，例如提取到 “加入我们” 这个特征值，再结合其他特征值以及相关概率进行判断，判断其是否是垃圾邮件。

机器学习可能会存在误判，但比基于规则的过滤方式高效的多，而且比较全面。

秒杀系统架构设计

秒杀场景的冲击在于：

1. 突然骤增的流量、服务器负载增加、带宽增加
2. 对现有网站造成冲击
3. 超买超卖问题

秒杀系统架构设计可以归纳为如下几点：

- 秒杀系统独立部署，避免对其他业务造成冲击

- 页面静态化，如评论、点赞量等数据冻结，下单地址使用默认地址或允许之后修改，不去请求其他服务，防止造成影响，静态页面使用 CDN 加速。
- 主动降级，提前关闭一些非核心功能。
- 下单按钮关闭，等到秒杀开始时开放，开放后禁止用户连续点击。
- 为防止用户使用爬虫下单，每次下单都需要带上一个随机数参数，此随机数通过请求服务器生成，服务器禁止一些不存在的随机数请求。服务器可以预先缓存一些随机数，从随机数池中分发。
- 提前返回结果，告知秒杀结束，例如如果只有 10 个秒杀名额，在请求进入订单处理系统之前，先经过一个拦截器，拦截器进行简单计数，如果已经存在 100 个请求了，则直接返回秒杀结束，避免大量请求打到服务器、数据库中。
- 预减缓存、内存标记、异步处理。Redis 等缓存服务器性能比数据库高得多，并且也提供原子操作，可以提前缓存库存数目，请求到达服务器时，先预减缓存，如果缓存中库存不足，则直接返回，并在位图中打上标记，下次首先根据位图判断库存是否充足。

  > 这个逻辑核心在于秒杀最终成功的人只占一小部分，而大多数人秒杀会失败。使用缓存和内存标记可以防止大量请求走到下一步读数据库的逻辑。
  >
  > 如果用户取消支付，需要对 Redis 进行补偿并解除内存标记。

- 异步削峰。可以考虑将下单请求交由 MQ 处理，后端开放查询接口供前端轮询，前端不断轮询并显示 “排队中”，后端可以利用 MQ 进行削峰处理，降低压力。
- 用户标记。一旦用户下单成功，则禁止用户再次下单；当然也可以禁止用户同一时间下单多次，例如使用 Redis 标记用户，并配置一个过期时间。