高频系统设计题： Design Tiny URL： coderpad.io/ABCDEFG

自学： [www.GOOGLE.COM](http://www.google.com) hit 这个网站后发生了什么

Consistent Hashing：Horizontal Sharding / Partition 其实是一个意思。 1 2 3 存在一台机器上，4 5 6 存在 另 一台机器上

工作上 Vertical Sharding 不太会 解决 问题

* Horizontal Sharding 和 Partition 是一个意思：
  + 当一个表更大的时候，用consistent hashing来解决：
* 为什么要Consistent Hashing：
  + % n 的方法是最简单的Hash 算法。（机器1： 0 3 6 9， 机器2： 1 4 7 10， 机器三： 2 5 8 11。 直接模3 ）但这种方法在机器数n变成n+1的时候，每个key % n +1 结果和之前基本不一样，导致了数据的大迁移 -- 所以称之为 inconsistent hashing （可能会有75%的数据移动）。最简单的 hash function。
  + 一个简单的consistent hashing：将key % 360 （一个圆环，固定的数目），将360分配给n台机器，每个机器负责一段区间，区间分配信息记录在 web server上。 （根据 wikipedia， only K / n data migration，k是总data 量， n 是机器数量， consistent hashing 可用于 distributed caching） 新加一台机器的时候，从表中选择一个位置插入（相邻两个区间 和 最大），分走相邻两台机器的一部分数据（n = 2 to 3，只有 ⅓ 的数据移动）。
    - 缺陷：
      * **不均分（3变4时无法均匀分布 not evenly distributed）**；迁移机器负载很大（所有的迁移数据都从2台迁移机器负责，并不是均匀分配任务量）
      * 迁移压力过大： 新机器数目只从老的两台机器获取，老的两台机器负载很大。活不是大家分着干，而是只有两台干
  + 更实用的consistent hashing：将整个Hash区间看做环，这个环的大小从0-359变为0-2^64 - 1, 想象成 环上 有 2 ^ 64 个点， 现在把**机器**和**数据**都看做环上的点
    - hash(机器ip/unique name) = 0- 2^64 - 1 -- **UUID算法 保证**几乎不发生碰撞 （怎么把字符串 变成一个 2^64 数字）
    - hash(数据的 key user\_id) = 0 - 2^64 - 1，ｍａｐ到顺时针的下一个机器要数据 （其实是 下一个 virtual node 要 数据）
    - virtual node 概念： 每个机器node变成1000个点，均匀的撒在圆圈中（有1000个分身）。怎么生成？MAC- 1，MAC-2.。。。。MAC-1000。用虚拟节点来让一台机器负责**不连续的区间。 1000 个 小区间**
    - hash（user\_id）后寻找顺时针下一个virtual node，每台机器能分到的数目就比较均匀
    - 怎么确定1000个分身？肯定是越大越好，但是越大的话，数据结构就越大（trade off）
    - 怎么去顺时针找到下一个virtual node机器？

      * 第一种 做法： Binary Search in sorted array （找第一个出现的 >= current 机器value）。（120， 149， 180， 250， 320）， 给定 190，会找到 250。这是在没有挂掉点的情况下
      * 如果机器会随时增加和删除，可以用TreeMap来实现（红黑树）NoSQL 数据库 在 做 sharding 的 时候，就是 用 TreeMap 来实现的。 只需要 把 虚拟节点 存 进来就好。-- 这个方法更优
      * What happened when you add a new machine？ 先把一台机器算出1000个分身，然后映射到对应的位置，需要做数据迁移，把顺时针的下面一个virtual node的所有节点的hash重新算一下，如果属于新的节点，则把它copy过去
        + 迁移的数据是不是 连续的？Big Data 那节课 会 讲，做 Lintcode 上 consistent hashing 的 题目！
      * 虚拟节点 并不是 真的 机器，它只是 放到这个环上，更均匀一点
* 数据库 和 系统设计 会 问 （Consistent Hashing 和 Replica）Replica数据备份：
  + Backup和Replica的区别是什么？
    - Backup一般是周期性的，离线复制。每天晚上备份，通常只是恢复到之前的某个时期点，不分摊读的压力。backup 是在 replica 中的某一个 下线之后，才上线 （云备胎）
    - Replica是实时的，在数据写入的时候，会以复制品的形式存为多份。当数据丢失的时候，可以马上通过其他的复制品恢复。读的话可能是到 replica 读 的。 replica 可以分摊读的压力： 如果有3个 request，可以分摊压力
  + MySQL Replica
    - 以MySQL为代表的数据库，通常有自带的 Master / Slave 方式来进行replica。一般是一台master 配 两个 slave， slave 分摊读 的 压力，但是 不写到 slave 中。 slave 只是 从 master 上 通过 WAL 同步数据。master 每次 告诉 slave 我有 哪些log，而不是 直接 把数据同步。所以 slave上的 数据是有延迟的。slave 上数据可能会不一致 （只要有 replica就一定会有不一致性）。data 不一致性 是 从application level 克服的。比如 点赞 这个 feature 就 不需要 那么精确，就可以从 slave读。如果应用层需要精确的数据，就从 master 去读。
    - Write Ahead Log(WAL) ： SQL数据库的任何操作，都以log形式做一个记录，
    - 从应用层去控制 到底读Master还是读Slave，如果读多少人like，则读Slave就可以，如果读critical info，则从master
    - 那么多Write Ahead Log，每隔一段时间会有一个check point，得知当前的值，而不是具体的operation
      * A 1- >2
      * B 3->4 t1（checkpoint）
      * 下次 同步，只从 checkpoint 来进行 同步
    - Master 挂了怎么办：SQL型数据库 自带的。Slave被激活，告诉Master我在了，Master每次有任何操作都通知Slave来读Log。Slave的数据有延迟，并非是复制data，而是复制record。Master挂了后，将一台Slave 升级成Master，接受读+写，可能会造成一定程度的数据丢失和不一致
  + Cassandra的Replica怎么存：来了一个数据，根据consistent hashing把数据存储到顺时针的3台机器上
  + SQL自带Replica方式是Master / Slave, SQL 不自带 consistent hashing。需要手动 constent hashing 做 Replica方式也可以在 Consistent Hashing环上顺时针存三份
  + NoSQL：NoSQL就是 在 sharding 和 replica 上 帮你偷懒。自带的Replica就是在ocnsistent hashing环上顺时针存三份，不需要手动存，NoSQL帮你实现了**Sharding**和**Replica**
* **Design Tiny URL**（高频题）系统设计中的 **战斗机**
  + 某同学：先来个Load Balancer，后面一堆Web Server，然后memcached，最底层NoSQL，搞定！！ -- 这种回答不好。因为并没有答出中点，没提出可能会遇到的问题。没有想到必须要用NoSQL，先别用 distributed system。 了解feature
  + e.g. bit.ly or goo.gl
  + Scenario:
    - 提问，问清楚条件。QPS，capacity
      * 1.long URL -> short URL。网页中给定Loooooooong URL，返回 shortURL
      * 2.网页中给定short URL 还原 Long URL， 并跳转(http response code： 301 永久变换为 另外的网址。302 是临时 变换为 另外的网址，直接跳转)至真实网址。浏览器跳转
      * QPS： 100M Daily Active User， 每个用户平均每天生成0.1条URL，
        + 写操作：100M \* 0.1 / 86400 ~ 100
        + Peak Write： 100 \* 2 = 200
        + Read: 假设每个用户每天点 1个 url，那么 Average Read QPS 就是 100 M \* 1 / 86400 == 1K
        + Peak Read： 2K
        + 流量比较小，一台MySQL就可以解决
      * Storage：
        + 100M \* 0.1 ~ 10M
        + 每条URL 按照 100Byte，一共1G，1TB硬盘可以用 3年，1台电脑足够OK
        + 所以这题不需要分布式，一台电脑就可以搞定
    - 可行解：Service + Storage
      * URLService
        + encode short to long
        + decode long to short
    - Storage： SQL or NoSQL
      * 是否需要支持transaction? 银行交易？如果不需要，就用 SQL 和 NoSQL都行
      * 是否需要丰富的SQL Query （secondary index）， short 去 query long，long query short。SQL的话可以secondary index。如果noSQL需要两张表，但也可以实现
      * 是否想偷懒？
        + 大多数Web Framework与SQL兼容得很好
        + SQL比NoSQL少些很多代码
      * 是否需要Sequential ID

SQL: Sequential auto increment 有个锁。

NoSQL: Non-sequential。

取决于算法是什么

* + - * NoSQL QPS性能更高
      * SQL需要自己写sharding/replica 的code
      * 基于此，tiny URL 用SQL 和 NoSQL都可以 （NoSQL就是需要建两张表，因为不支持secondary index， NoSQL不怎么需要 scalability。sequential ID 是否需要取决于 算法，下面讨论算法：：：）
    - 算法是什么：
      * 将long URL转化为 6位的URL。为什么是6位？
        + 设计一个 Hashing Function 将 long URL 转化为 Short URL。难以设计一个没有冲突的哈希算法。
      * 第二种算法： 随机生成6位的ShortURL，在HashTable里找如果没有用过，就使用该URL，创建 对应关系 （short URL vs. Long URL），否则再生成一个。
        + 优点：实现简单。很多网站都是这么做的。机票Confirmation Number
        + 缺点： 网址越多的话，short URL生成速度会越来越慢 ，出现冲突的概率会越来越高，数据库查重复越来越慢(航司的Confirmation Number就是这么干的）
      * 算法3：进制转换Base62
        + 0-9，a-z， A-Z一共62个number，当做62进制来算
        + 每个short url对应到一个整数，该整数对应数据库表的Primary Key - Sequential ID
        + 6位 = 62^6 = 57B = 570亿个组合。62^5 ~ 900 Million。 62^7 = 3500 Billion
        + 其实就是 根据 Sequencial ID 生成 Short URL （10进制转化成 62 进制，不够6位就补0）。ShortURL to Sequencial ID （62进制 to 10进制）
        + idToShortURL（）， shortURLToID()
        + 优点： 效率高 （O（6）常数级别），
        + 缺点：（依赖于全局sequential自增Id瓶颈被全局id卡死了）。需要访问数据库的中心节点得到自增I
        + 可能有security concern：因为算法暴露在外了，大家都可以试各种 short URL。但是由于 short URL 本身 就是 public 的 nature。 如果是confirmation number的话，可能算法2更好一点，因为有了confirmation number 就 能登机了。
        + 不要特别倾向于 哪种方法： 分别讨论每个方法的优点和 缺点
      * 真的关心效率问题吗？ 2K QPS 很简单。算法2和算法3都可以。一秒钟 插入 200条数据，用 哪个 方法都可以。
        + 随机生成 vs. 进制转化：
        + 两种都可以，分别描述trade off
    - 基于随机生成的方法：
      * 对于SQLdatabase，需要shortUrl column 和 longURL column。分别要对 shortUrl 和 LongURL 进行 index
      * 对于 NoSQL， 需要两张表，以 Cassandra为例：
        + 根据Long 查询 short： row\_key = longURL, column key = short URL, value = null / timestamp
        + row\_key = SHORTurl, columnKey = longURL, value = null or timestamp,因为大部分 nNoSQL不支持 secondary index
      * scenario， browser hit short URL，返回 301 然后redirect
    - 基于进制转换的方法：
      * 因为需要auto increment id，所以只能用 SQL， 需要 id column 和 longUrl (index 这个 column），不需要存 short key， 只需要 存 id，可以根据id 去换算成 shortURL
      * 最大区别就是 从出具库 那边 return的是 id
      * 要分别比较优劣，答出来能给 weak hire
    - Scale -- 答好才能hire
      * 两种问法
      * 第一种：How to Shard？
        + How to Replica，
        + 可以把 consistent hashing 念一遍，用什么做key是比较讲究的
      * 第二种： How to reduce response response time?
        + 点了短网址如何快速转到长网址
        + goal： 提高读的速度还是 写的速度？ 提高读的速度（读密集应用，只要给人用的读密集，给机器用的是写密集）
        + 哪些地方可以improve？

MemCacheD （cache aside）：可以用Cache Aside的方法，先去memcached里面去检索（存储 short URL 和 longURL 的pair），然后再去数据库检索

利用地理位置信息进行提速Location based： 不同地区使用不同web server，不同用户根据DNS （Domain Name Service）解析到不同地区的服务器。 （[www.XXX.COM对应的是](about:blank) 192.168.x.x 这个 ip 地址 是存在 DNS server 上，google有个8.8.8.8服务器，它会存储hostname and ip pair）访问8.8.8.8美国DNS服务器，告知去哪里访问。中国去 中国的服务器，美国去美国的服务器。DNS不是全球的中心节点。每次访问会优先访问 美国DNS服务器，GoDaddy 上更新 DNS IP需要 24 小时 或48小时 来让globally知道 IP 变了

每个地理位置的web server有一个mem cached。然后有一个 中心 的 Shared Database。这样会快很多，因为 client browser 和 web server 很近， 但是web server 和 DB server 的通信效率会比较高，会比从 中国 browser直接访问美国服务器快很多

* + - * 如果Read 、write request 变得 很多，需要多台服务器搞定怎么办？如何 sharding （分布在 不同的 机器上）
        + Consistent Hashing用什么做sharding key？ 用short还是long URL，还是ID？
        + 如果用Long URL做sharding Key： given LongURL，可以找到了对应机器。但是主流需求是short to long，如果现在只知道 abEMc0, 不知道long是什么，怎么知道存在哪台机器？只能广播，就没有sharding的意义了。 所以用 Long URL 做 sharding key 一定不对！
        + 如果用ID（shortURL）做sharding Key，对于short URL to Long时，可以找到机器。 不过Long URL就不知道怎么查询 是否存在 short URL 了，怎么看有没有存在某台机器上，只能广播了。。不过只有写的时候才需要，写比较少，并且写的需求比读的需求少一个数量级。 （最核心的问题是，没有办法用 LongURL 直接 通过 一个 HashFunction 生成一个 不重复 的shortURL）
        + 怎么解决不广播的问题:： 把shortURL的前头加一位作为sharding Key，这个key 又 Hash（Long\_URL）%62得到，前提是 database的数目不能超过62，这样 不管是given long url 还是 given short url 都能够得到机器的位置
        + 另一种优化： 广播建立的假设是，每次都要判断 去重。是否一定要建立一个假设，一个 long URL 必须唯一对应一个short URL。Google其实做的是可以允许重复，之前假设Given Long，都要看看是否short已经生成过了，如果这个假设不成立，也就是说，不管三七二十一，来了一个新的都随机生成一个short URL，LongURL 对应short URL 1 和 short URL 2。则Long URL不需要广播，用随机数的方法，随机生成一个 shortURL，然后就知道了对应哪台机器，看一下这台机器上如果没有 重复，就使用这个 short URL 作为 long URL 的pair。在这里用 随机 生成的方式会 更简单，带来的问题就是 一个 longURL被生成了很多分shortURL。
        + 怎么避免过度的生成 short URL： Web 和 memcacheD和DB如果在不同机器，则network成为瓶颈。如果cache和Web Server 在同一台机器， local cache存一些热门long URL，直接从 内存中 拿。如果有100台 web server，一个 请求 到了 web server 1， 热门的 long URL 和 short URL 在cache 中，DB sharding 成 中间的 四台DB server，则可以避免将一个long生成过多的short URL
      * 继续优化：
        + web server 与 database 之间的 通信
        + centralized DB sets and web server 之间通信较慢

e.g. web server in china to viist dabase server in US

* + - * + 按照用户习惯进行优化：

中国用户访问中国网址。

StackOverflow大概率是美国或欧洲的节点，将用户比较经常访问的网站做一张表。

以第一位sharding key判断是中国网站还是美国的网站。0 表示中国，1表示 美国，放在 sharding key 的第一位，将DB按地域划分 如果 是 0 就访问 DB USA，1就访问 DB China。DB-USA， DB-CHN。还可以做Replica，将远端的replica只能读，如果写的话，就写的慢一点。 Sharding 对 读的优化比较明显，Replica 对读的优化比较有限，因为可能会 inconsistent

* + - 是否需要长久保留short URL?
      * Bitley: 过段时间就会失效了。用户体验 不好
      * 应该要长久保存。但如果网址本身已经失效了，则不一定需要保存
    - Short URL 被疯狂点击 怎么办：。
* 当点击 Google.COM 是 发生了什么？
  + 访问 离当前机器最近的DNS ， 逐渐找到 [www.GOOGLE.COM](http://www.google.com) 对应的 ip是什么
  + browser 向 IP 发送 request
  + server listens to 80 for http and 443 for https
    - Http Server type： Apache / uNICORN. gUNICORN, uWSGI
    - Web Application Framework： Django， Ruby on Rails， NodeJS
  + 根据path 找到 对应的 model
  + organize data and return Html 返回
  + 复杂的网站，会 首先hit load balancer，然后返回对应的web server ip