**<https://blog.csdn.net/dianxin113/article/details/108443540>**

短链系统设计看起来很简单，但如何设计一个高性能短链系统呢，这也是面试中非常常见的一道设计题。

首先，为什么要用短链？

短链跳转的基本原理是什么？

短链生成的几种方法你知道吗？

高性能短链的架构如何设计？

1. 短链的好处

短链现在基本上属于各大公司发短信必备技术了，比如CSDN发我的验证码短信，下方蓝色链接明显是短链。

而当我点开浏览器之后，发现这个短链已经跳转到其根本的长链接上去了。

那么为啥要先转成短链表示呢，显而易见：

1、url链接变短，节省空间而且更美观

最典型的就是微博和短信，如果一个参数很多路径很深的长链接发出去的话，基本上你这个微博或短信就没法再添加其他文字信息了。另外用短链在内容排版上也更美观，大家肯定不想看到长长的一串各种参数的url。

2、短链生成的二维码密集度更低，更容易被识别。如图示：

3、链接太长无法自动识别

最常见的就是发个kibana长链接在QQ微信或钉钉上中间断开了，很不友好，当然这种内部使用的基本上也懒得去转成短链。

2. 短链跳转原理

短链跳转到长链，“跳转”两个字就让我想到了302重定向。那么它是不是和我猜想的一样呢，我们可以抓个包看看

果然如我猜测的那样，短链请求后返回了状态码 302（重定向）与 location 值为长链的响应，然后浏览器会再请求这个长链以得到最终的响应,整个交互流程图如下：

注：A：生成的短链 ；B：对应的长链

熟悉计算机网络的大家都知道301和302都有重定向的功能，那这里为什么选择302呢？

这也是面试官经常会问到的一个小问题。

所以这里就需要了解 301 和 302 的区别：

301永久重定向，即浏览器只需要第一次请求拿到长链接后，下次再去访问这个短链就不会向短网址服务器请求了，而是直接从浏览器的缓存里拿。这样可以提高浏览器的访问速度，但是也有一个问题，就是如果我们想统计活动链接的访问数据的话就无从下手了。或者是这个活动结束了我想删除访问入口，但由于浏览器缓存可能还会导致有用户访问到。所以我们一般不使用301。

302 临时重定向，即每次访问短链都会去请求短网址服务器（除非响应中用 Cache-Control 或 Expired 暗示浏览器缓存），这样就便于 server 数据监控，所以虽然用 302 会给 server 增加一点压力，但明显是利大于弊的。

3. 短链生成方法

既然知道了短链的好处和跳转原理，那如何来根据长链生成短链呢？这才是本文的重点。

仔细观察上例中的短链，显然它是由固定短链域名 + 长链映射成的一串字母组成，这听起来是不是很熟悉？哈希函数不就用来干这事的吗。

1、哈希算法

MurmurHash算法

短链的域名每个公司肯定都有固定的域名，我们要做的生成后面的那一小串字母即可，于是我们有了以下设计思路

那么这个哈希函数该怎么取呢，第一印象肯定有很多人说用 MD5，SHA 等算法，这些哈希算法也可以但是没必要，大材小用了，而且加密哈希会有性能上的损失，对于短链系统来说我们更关心的是哈希的运算速度和冲突概率。

这里推荐 一种非加密型的hash算法—— MurmurHash 算法，适用于一般的哈希检索操作。与其它流行的哈希函数相比，对于规律性较强的 key，MurmurHash 的随机分布特征表现更良好。

非加密意味着着相比 MD5，SHA 这些函数它的性能肯定更高，正由于性能好，目前已经广泛应用到 Redis、MemCache、Cassandra、HBase、Lucene 等众多著名的软件中。

MurmurHash 提供了两种长度的哈希值，32 bit，128 bit，为了让网址尽可通地短，我们一般选择 32 bit 的哈希值就够用了，32 bit 能表示的最大值近 43 亿。

对长链做 MurmurHash 计算，得到的哈希值是十进制的，为了变得更短一些可以转换成62进制。比如我们把hash值为3002604296转为 62 进制可缩短它的长度：

于是我们有 (3002604296)10 = (3hcCxy)10，一下从 10 位缩短到了 6 位！

注：这里对MurmurHash不做过多分析，感兴趣的同学可以自己去研究。

哈希冲突

既然是哈希函数，不可避免地会产生哈希冲突（尽管概率很低），该怎么解决呢。

我们知道既然访问访问短链能跳转到长链，那么两者之前这种映射关系一定是要保存起来的，可以用 Redis 或 Mysql 等，这里我们选择用 Mysql 来存储。表结构如下所示

CREATE TABLE `short\_url\_map` (

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

`lurl` varchar(160) DEFAULT NULL COMMENT '长地址',

`surl` varchar(10) DEFAULT NULL COMMENT '短地址',

`ctime` int(11) DEFAULT NULL COMMENT '创建时间',

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

于是我们有了以下设计思路。

将长链（lurl）经过 MurmurHash 后得到短链。

再根据短链去 short\_url\_map 表中查找看是否存在相关记录

如果不存在，则将长链与短链对应关系插入数据库中存储。

如果存在，说明已经有相关记录了，此时在长串上拼接一个自定义好的字段，比如「DUPLICATE」，然后再对接接的字段串「lurl + DUPLICATE」做第一步hash操作，如果最后还是重复则继续拼接直到不重复，查询时根据短链取出长链的时候把这些自定义好的字符串移除即是原来的长链。

这样一个初步的短链系统就可以实现了，但明显还是有优化的地方，插入一条记录居然要经过两次 sql 操作（根据短链查记录，将长短链对应关系插入数据库中），如果在高并发下，显然会成为瓶颈。

如何优化？

给短链字段 surl 加上唯一索引即可

当长链经过 MurmurHash 得到短链后，直接将长短链对应关系插入 db 中，如果 db 里不含有此短链的记录，则插入成功，如果包含了则说明违反了唯一性索引则会插入失败，此时再重新hash插入直到成功即可。

因为 MurmurHash 发生冲突的概率是非常低的，基本上不太可能发生，所以这种方案是可以接受的。当然如果在数据量很大的情况下，冲突的概率会增大，此时我们可以加布隆过滤器来进行优化。

用所有生成的短网址构建布隆过滤器，当一个新的长链生成短链后，先将此短链在布隆过滤器中进行查找，如果不存在，说明 db 里不存在此短网址，可以插入！

综上，总体的设计思路如下

用哈希算法生成的短链其实已经能满足我们的业务需求，除了哈希算法还有别的方法可以生产短链吗？

当然有，短链说到底其实可以看做一个唯一ID，这就变成了如何设计生产全局唯一ID的题目了。

2、全局唯一ID生成算法

我们可以维护一个 ID 自增生成器【唯一ID】，主要有以下四种获取 id 的方法

1、类 uuid

UUID是指在一台机器上生成的数字，它保证对在同一时空中的所有机器都是唯一的，但这种方式生成的 id 比较长，且无序，在插入 db 时可能会频繁导致页分裂，影响插入性能。

2、Redis

用 Redis 是个不错的选择，性能好，单机可支撑 10 w+ 请求，足以应付大部分的业务场景，说如果一台机器扛不住可以设置多台。不过用 Redis 这种方案，需要考虑持久化（短链 ID 总不能一样吧），灾备，成本有点高。

3、Snowflake

用 Snowflake 也是个不错的选择， Snowflake 依赖于系统时钟的一致性。如果某台机器的系统时钟回拨，有可能造成 ID 冲突，或者 ID 乱序。snowflake算法及其改进算法在生成全局唯一ID上已被很多场景使用，但要保证时钟的一致性，系统设计略复杂。

4、Mysql 自增主键

这种方式使用简单，扩展方便，所以我们使用 Mysql 的自增主键来作为短链的 id。简单总结如下：

那么问题来了，如果用 Mysql 自增 id 作为短链 ID，在高并发下，db 的写压力会很大，这种情况该怎么办呢。

我们可以提前生成这些自增 id 存起来，用的时候直接去用就好了！

方案如下：

设计一个专门的发号表，每插入一条记录，为短链 id 预留 （主键 id \* 1000 - 999） 到 (主键 id \* 1000) 的号段，如下

发号表：url\_sender\_num

如图示：tmp\_start\_num 代表短链的起始 id，tmp\_end\_num 代表短链的终止 id。

当长链转短链的请求打到某台机器时，先看这台机器是否分配了短链号段，未分配就往发号表插入一条记录，则这台机器将为短链分配范围在 tmp\_start\_num 到 tmp\_end\_num 之间的 id。从 tmp\_start\_num 开始分配，一直分配到 tmp\_end\_num，如果发号 id 达到了 tmp\_end\_num，说明这个区间段的 id 已经分配完了，则再往发号表插入一条记录就又获取了一个发号 id 区间。

画外音：思考一下这个自增短链 id 在机器上该怎么实现呢， 可以用 redis, 不过更简单的方案是用 AtomicLong，单机上性能不错，也保证了并发的安全性，当然如果并发量很大，AtomicLong 的表现就不太行了，可以考虑用 LongAdder，在高并发下表现更加优秀。

整体设计图如下

解决了发号器问题，我们还需要一个短链和长链对应的映射表，短链ID为主键，为了防止同一个长链生成多个不同的短链，可以在长链上加索引。但因为长链字符串较长可以考虑将其md5后存储。

表设计如下

CREATE TABLE `short\_url\_map` (

`id`int(11) unsigned NOT NULL AUTO\_INCREMENT COMMENT'短链 id',

`lurl`varchar(10) DEFAULT NULL COMMENT'长链',

`md5`char(32) DEFAULT NULL COMMENT'长链md5',

`ctime`int(11) DEFAULT NULL COMMENT'创建时间',

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

当然了，数据量如果很大的话，后期就需要分区或分库分表了。分库分表又是另一个沉重的话题啦。

4. 高性能短链架构设计

对于一些高并发高QPS的秒杀活动，光是上面这是肯定是扛不住的，考虑到这种情况，可以考虑引入 openResty。

它是一个基于 Nginx 与 Lua 的高性能 Web 平台，由于 Nginx 的非阻塞 IO 模型，使用 openResty 可以轻松支持 100 w + 的并发数，一般情况下你只要部署一台即可，不过为了避免单点故障，两台为宜，同时 openResty 也自带了缓存机制，集成了 redis 这些缓存模块，也可以直接连 mysql。不需要再通过业务层连这些中间件，性能自然会高不少

如图示，使用 openResty 省去了业务层这一步，直达缓存层与数据库层，也提升了不少性能。

5. 总结

本文对短链设计方案作了详细地剖析，旨在给大家提供几种不同的短链设计思路，文中涉及到挺多像布隆过滤器，openResty ，snowflake等技术，文中没有展开讲，感兴趣的话大家可以再详细了解一下。

再比如文中提到的 Mysql 页分裂也需要对底层使用的 B+ tree 数据结构，操作系统按页获取等知识有比较详细地了解，这些每一个展开说都可以再搞一篇长文了，让我们下次再聊吧。

————————————————

版权声明：本文为CSDN博主「羁士」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/dianxin113/article/details/108443540