高性能短链设计

捡田螺的小男孩 2022-07-29 07:47 发表于广东

以下文章来源于码海 ， 作者码海

码海

一起进阶，一起牛逼！

程序员田螺

专注分享后端面试题，包括计算机网络、MySql数据库、Redis缓存、操作系统、Java后...

11篇原创内容

公众号

前言

今天，我们来谈谈如何设计一个高性能短链系统，短链系统设计看起来很简单，但每个点都能展开很多知识点，也是在面试中非常适合考察侯选人的一道设计题，本文将会结合我们生产上稳定运行两年之久的高性能短链系统给大家简单介绍下设计这套系统所涉及的一些思路，希望对大家能有一些帮助。

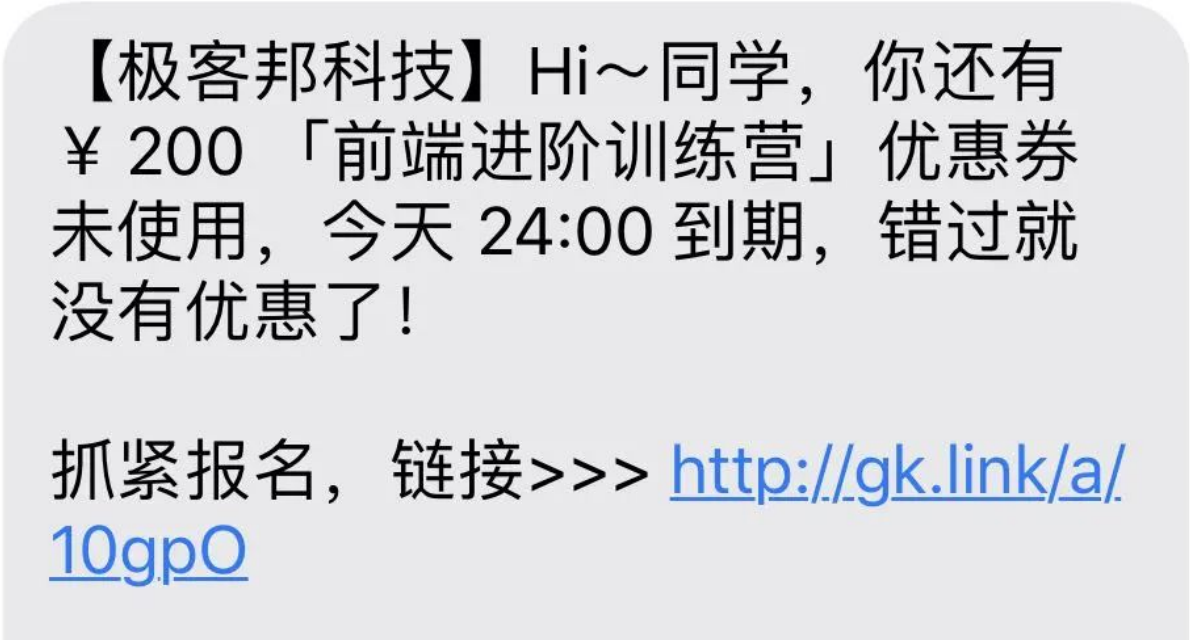
本文将会从以下几个方面来讲解，每个点包含的信息量都不少，相信大家看完肯定有收获

- 短链有啥好处，用长链不香吗
- 短链跳转的基本原理
- 短链生成的几种方法
- 高性能短链的架构设计

注：里面涉及到不少布隆过滤器，*snowflake* 等技术，由于不是本文重点，所以建议大家看完后再自己去深入了解，不然展开讲篇幅会很长

短链有啥好处，用长链不香吗

来看下以下极客时间发我的营销短信，点击下方蓝色的链接（短链）



浏览器的地址栏上最终会显示一条如下的长链。

那么为啥要用短链表示，直接用长链不行吗，用短链的话有如下好外

1、链接变短，在对内容长度有限制的平台发文，可编辑的文字就变多了

最典型的就是微博，限定了只能发 140 个字，如果一串长链直接怼上去，其他可编辑的内容就所剩无几了，用短链的话，链接长度大大减少，自然可编辑的文字多了不少。

再比如一般短信发文有长度限度，如果用长链，一条短信很可能要拆分成两三条发，本来一条一毛的短信费变成了两三毛，何苦呢。另外用短链在内容排版上也更美观。

2、我们经常需要将链接转成二维码的形式分享给他人，如果是长链的话二维码密集难识别，短链就不存在这个问题了,如图示

长链，密集难识别

短链，易识别

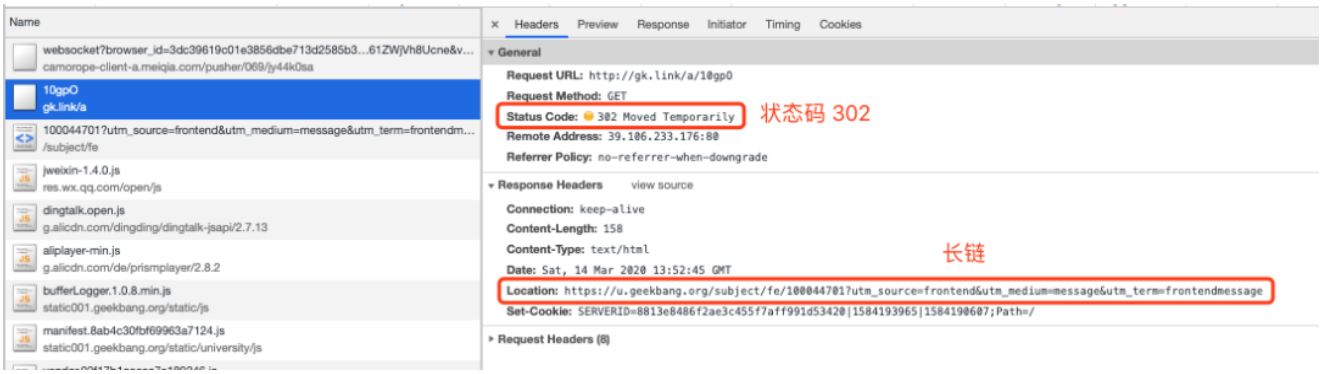
3、链接太长在有些平台上无法自动识别为超链接

如图示，在钉钉上，就无法识别如下长链接，只能识别部分，用短地址无此问题

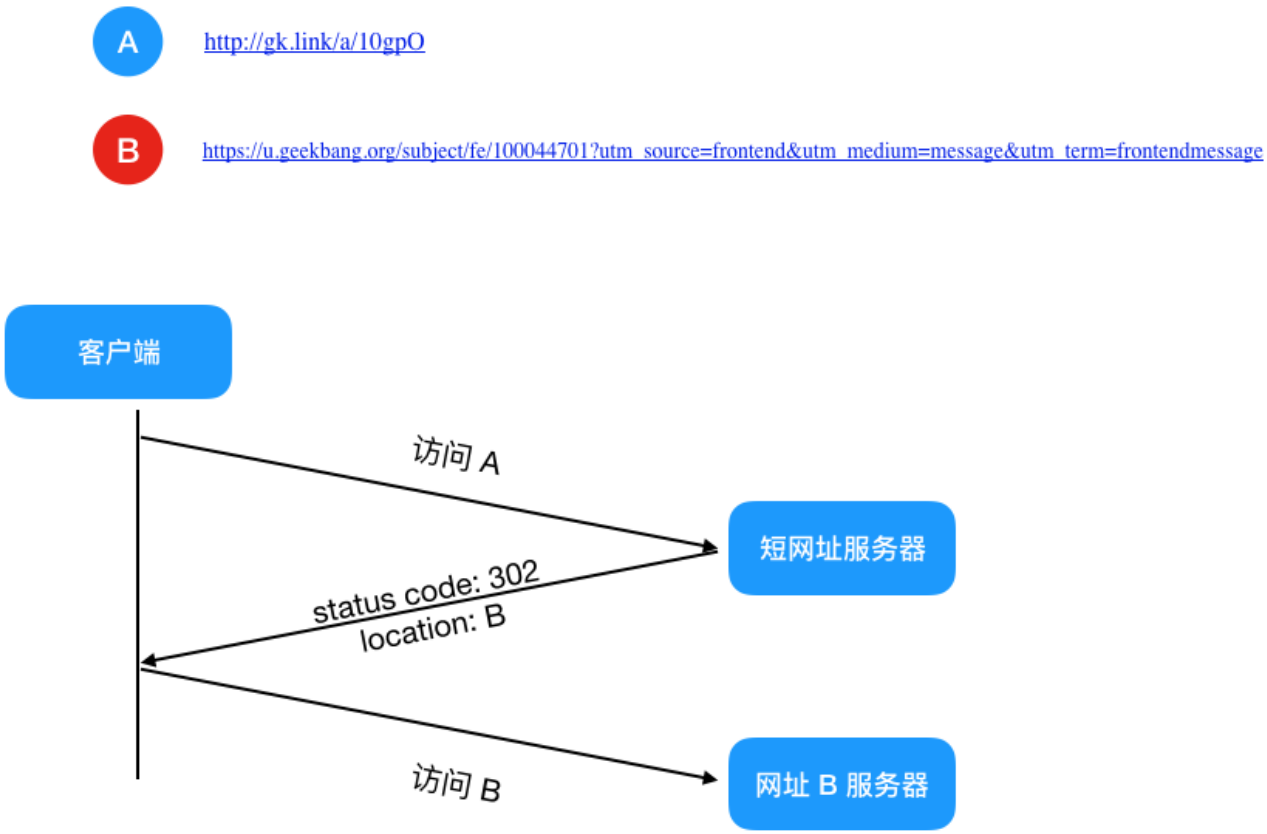
```
http://[redacted]./kibana/app/kibana#/discover?_g=()&_a=(columns:!
(_source),index:AWTQMc5qiAA8G3rW_XZ5,interval:auto,query:(language:lucene,query:''),sort:!
('@timestamp',desc))
```

短链跳转的基本原理

从上文可知，短链好处多多，那么它是如何工作的呢。我们在浏览器抓下包看看



可以看到请求后，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



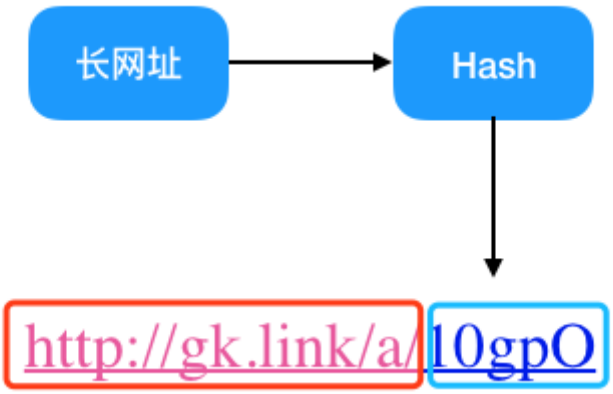
主要步骤就是访问短网址后重定向访问 B，那么问题来了，301 和 302 都是重定向，到底该用哪个，这里需要注意一下 301 和 302 的区别

- 301，代表 永久重定向，也就是说第一次请求拿到长链接后，下次浏览器再去请求短链的话，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，这样在 server 层面就无法获取到短网址的点击数了，如果这个链接刚好是某个活动的链接，也就无法分析此活动的效果。所以我们一般不采用 301。
- **302**，代表 临时重定向，也就是说每次去请求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 统计点击数，所以虽然用 302 会给 server 增加一点压力，但在数据异常重要的今天，这点代码是值得的，所以推荐使用 302！

短链生成的几种方法

1、哈希算法

怎样才能生成短链，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，那么长链怎么才能映射成一串字母呢，哈希函数不就用来干这事的吗，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，相信肯定有很多人说用 MD5，SHA 等算法，其实这样做有点杀鸡用牛刀了，而且既然是加密就意味着性能上会有损失，我们其实不关心反向解密的难度，反而更关心的是哈希的运算速度和冲突概率。

能够满足这样的哈希算法有很多，这里推荐 Google 出品的 MurmurHash 算法，MurmurHash 是一种非加密型哈希函数，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（实际上性能是 MD5 等加密算法的十倍以上），也正是由于

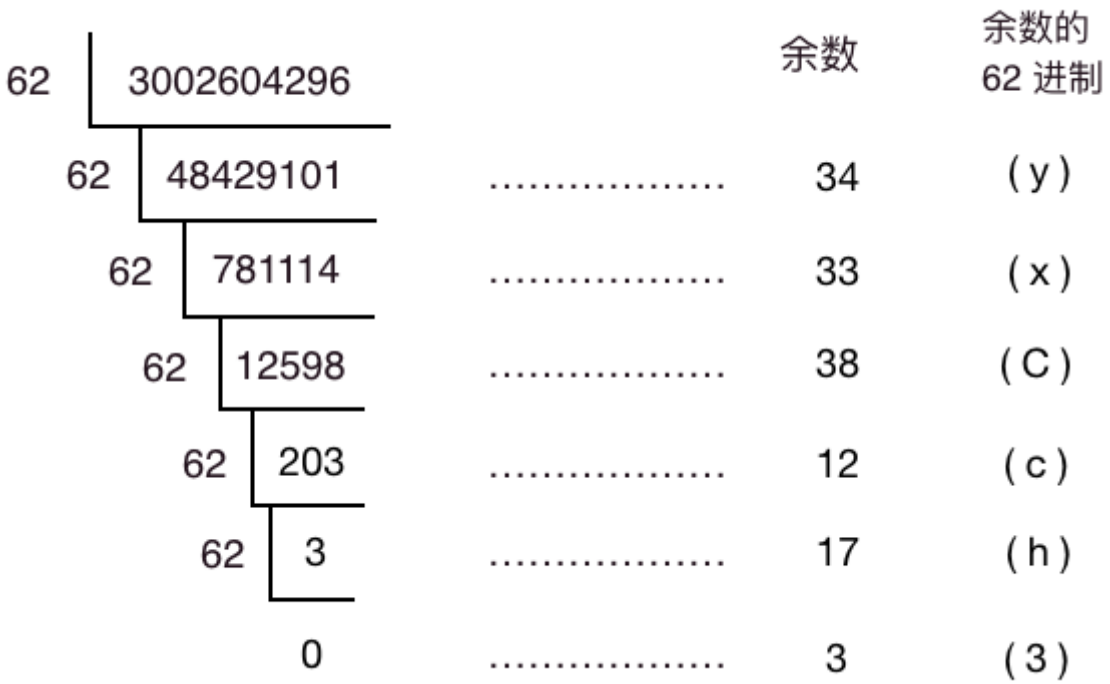
它的这些优点，所以虽然它出现于 2008，但目前已经广泛应用到 Redis、MemCache、Cassandra、HBase、Lucene 等众多著名的软件中。

画外音：这里有个小插曲，*MurmurHash* 成名后，作者拿到了 *Google* 的 *offer*，所以多做些开源的项目，说不定成名后你也能不经意间收到 *Google* 的 *offer* ^_^。

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，我们选择 32 bit 的哈希值，32 bit 能表示的最大值近 43 亿，对于中小型公司的业务而言绰绰有余。对上文提到的极客长链做 MurmurHash 计算，得到的哈希值为 3002604296，于是我们现在得到的短链为 固定短链域名+哈希值 = http://gk.link/a/3002604296

如何缩短域名？

有人说人这个域名还是有点长，还有一招，3002604296 得到的这个哈希值是十进制的，那我们把它转为 62 进制可缩短它的长度，10 进制转 62 进制如下：



于是我们有 $(3002604296)_{10} = (3hcCxy)_{10}$，一下从 10 位缩短到了 6 位！于是现在得到了我们的短链为 http://gk.link/a/3hcCxy

画外音：*6* 位 *62* 进制数可表示 *568* 亿的数，应付长链转换绰绰有余

如何解决哈希冲突的问题？

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。表结构应该如下所示

```
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',
  `gmt_create` 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。

1. 将长链（lurl）经过 MurmurHash 后得到短链。
2. 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，将长链与短链对应关系插入数据库中，存储。
3. 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」做第一步操作，如果最后还是重复呢，再拼一个字段串啊，只要到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。

以上步骤显然是要优化的，插入一条记录居然要经过两次 sql 查询（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。

画外音：一般数据库和应用服务（只做计算不做存储）会部署在两台不同的 *server* 上，执行两条 *sql* 就需要两次网络通信，这两次网络通信与两次 *sql* 执行是整个短链系统的性能瓶颈所在！

所以该怎么优化呢

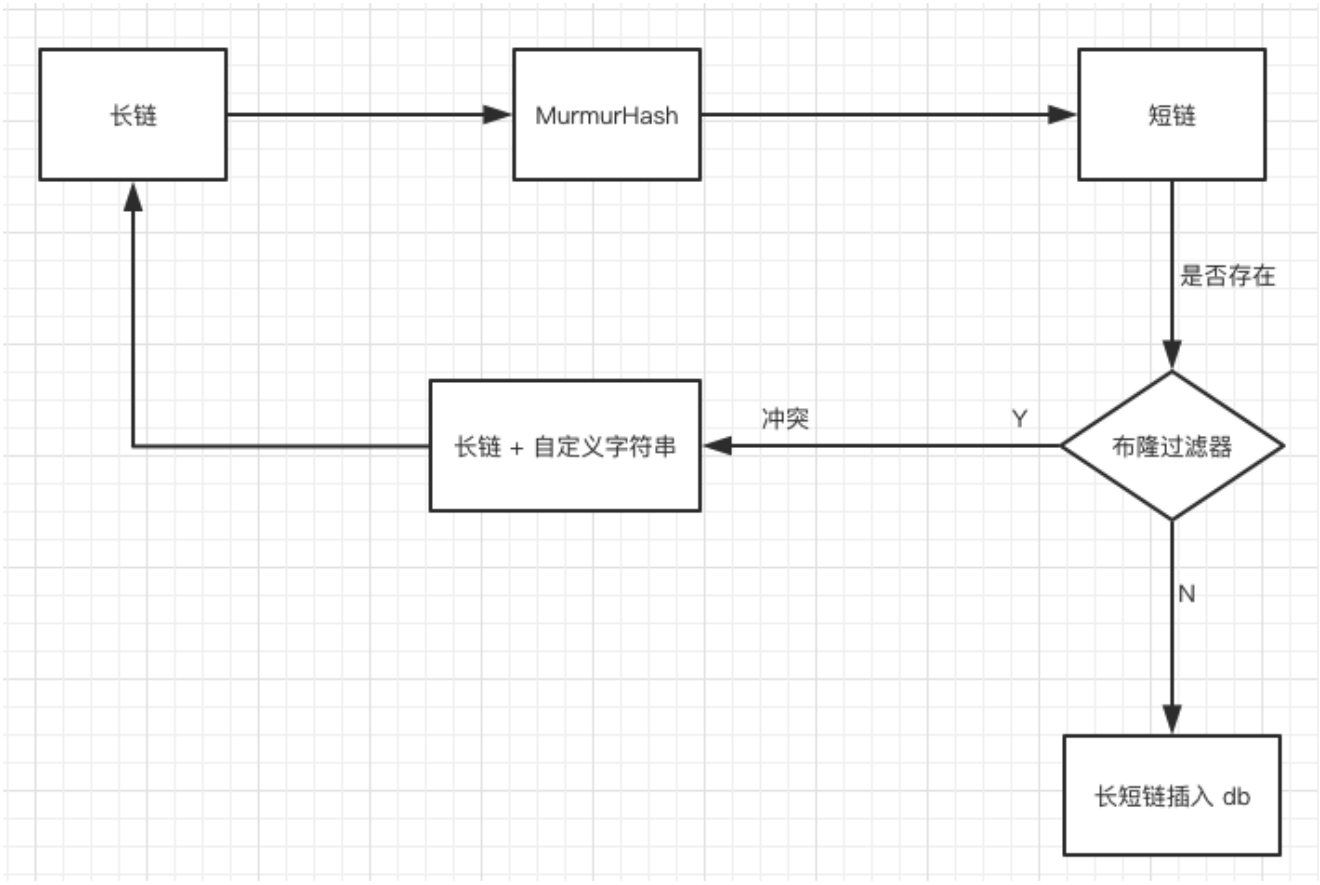
1. 首先我们需要给短链字段 surl 加上唯一索引
2. 当长链经过 MurmurHash 得到短链后，直接将长短链对应关系插入 db 中，如果 db 里不含有此短链的记录，则插入，如果包含了，说明违反了唯一性索引，此时只要给长链再加上我们上文说的自定义字段「DUPLICATE」,重新 hash 再插入即可，看起来在违反唯一性索引的情况下是多执行了步骤，但我们要知道 MurmurHash 发生冲突的概率是非常低的，基本上不太可能发生，所以这种方案是可以接受的。

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！

画外音：布隆过滤器是一种非常省内存的数据结构，长度为 *10* 亿的布隆过滤器，只需要 *125 M* 的内存空间。

综上，如果用哈希函数来设计，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，接下来我们再来看看如何用自增序列的方式来生成短链

2、自增序列算法

我们可以维护一个 ID 自增生成器，比如 1，2，3 这样的整数递增 ID，当收到一个长链转短链的请求时，ID 生成器为其分配一个 ID，再将其转化为 62 进制，拼接到短链域名后面就得到了最终的短网址，那么这样的 ID 自增生成器该如何设计呢。如果在低峰期发号还好，高并发下，ID 自增生成器的的 ID 生成可能会系统瓶颈，所以它的设计就显得尤为重要。

主要有以下四种获取 id 的方法

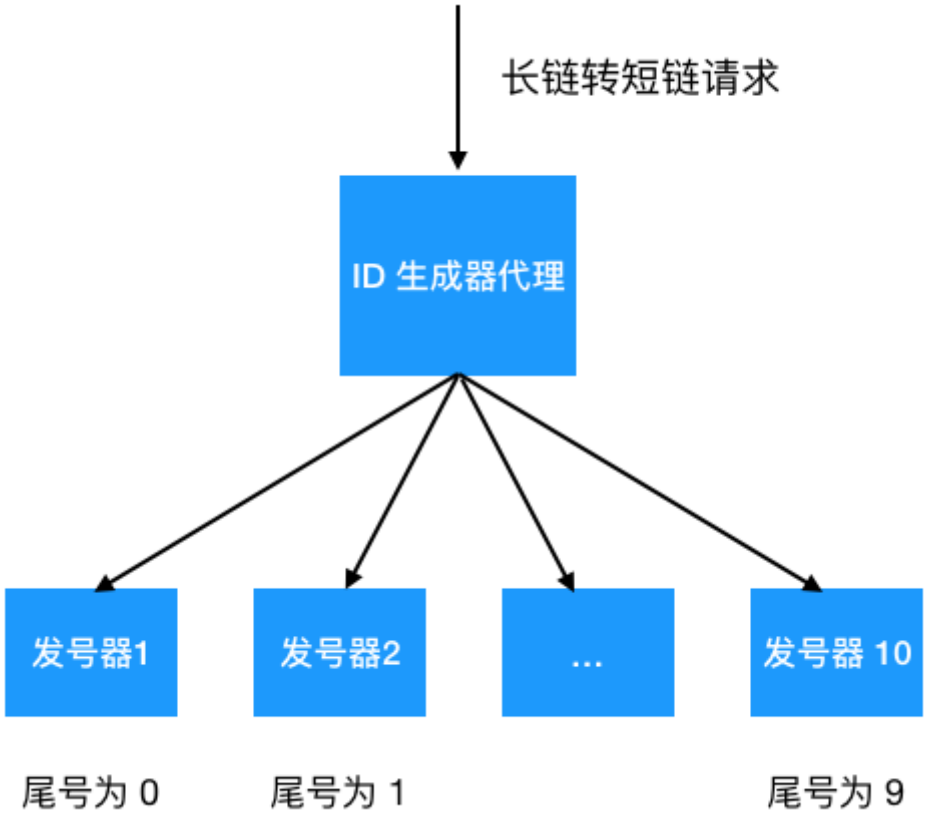
1、类 uuid

简单地说就是用 **UUID uuid = UUID.randomUUID();** 这种方式生成的 UUID，UUID(Universally Unique Identifier)全局唯一标识符,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，但有人说如果一台机器扛不住呢，可以设置多台嘛，比如我布置 10 台机器，每台机器分别只生

成尾号0，1，2，...9 的 ID, 每次加 10 即可，只要设置一个 ID 生成器代理随机分配给发号器生成 ID 就行了。



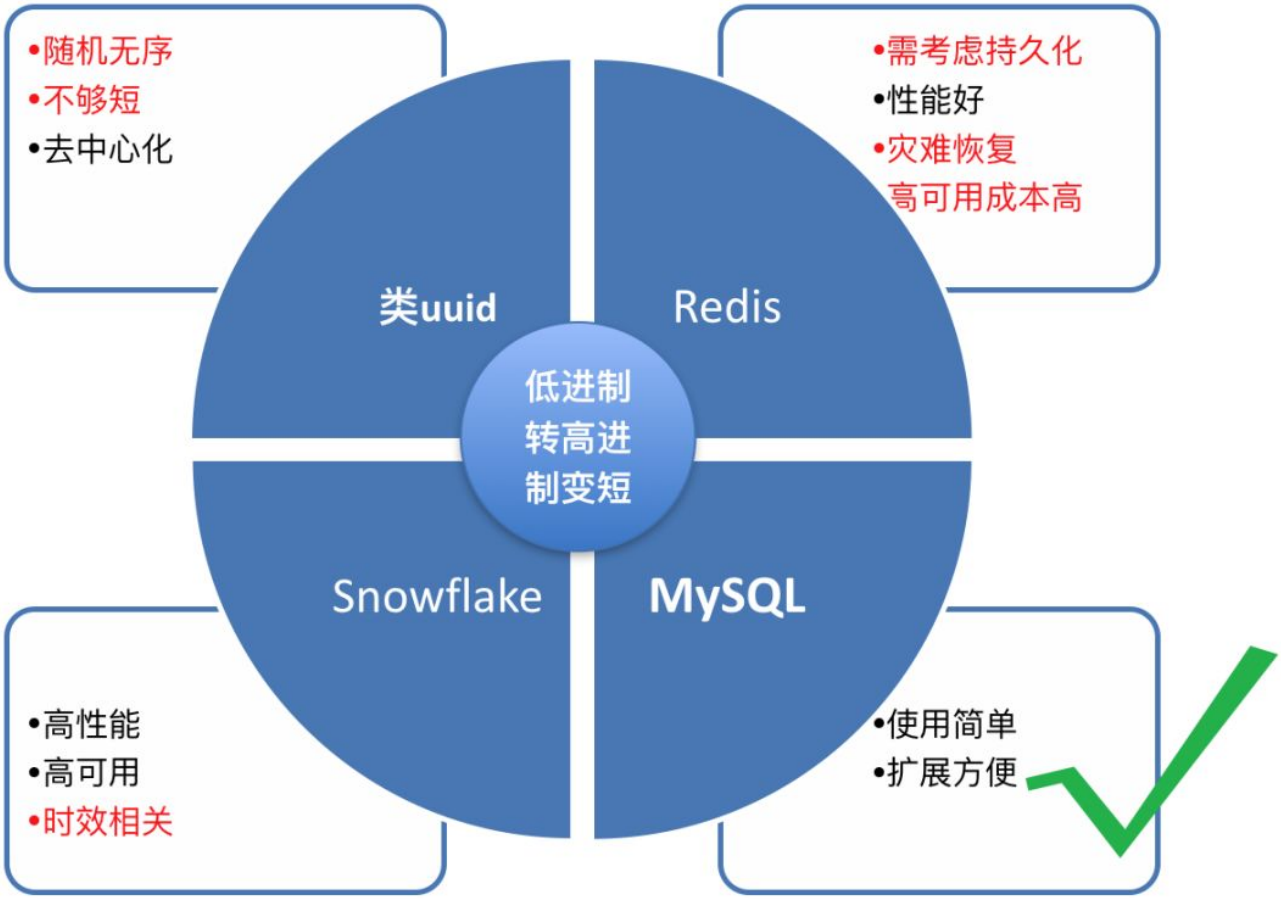
不过用 Redis 这种方案，需要考虑持久化（短链 ID 总不能一样吧），灾备，成本有点高。

3、Snowflake

用 Snowflake 也是个不错的选择，不过 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。

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。

考虑一下，一定要在用到的时候去生成 id 吗，是否可以提前生成这些自增 id ?

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

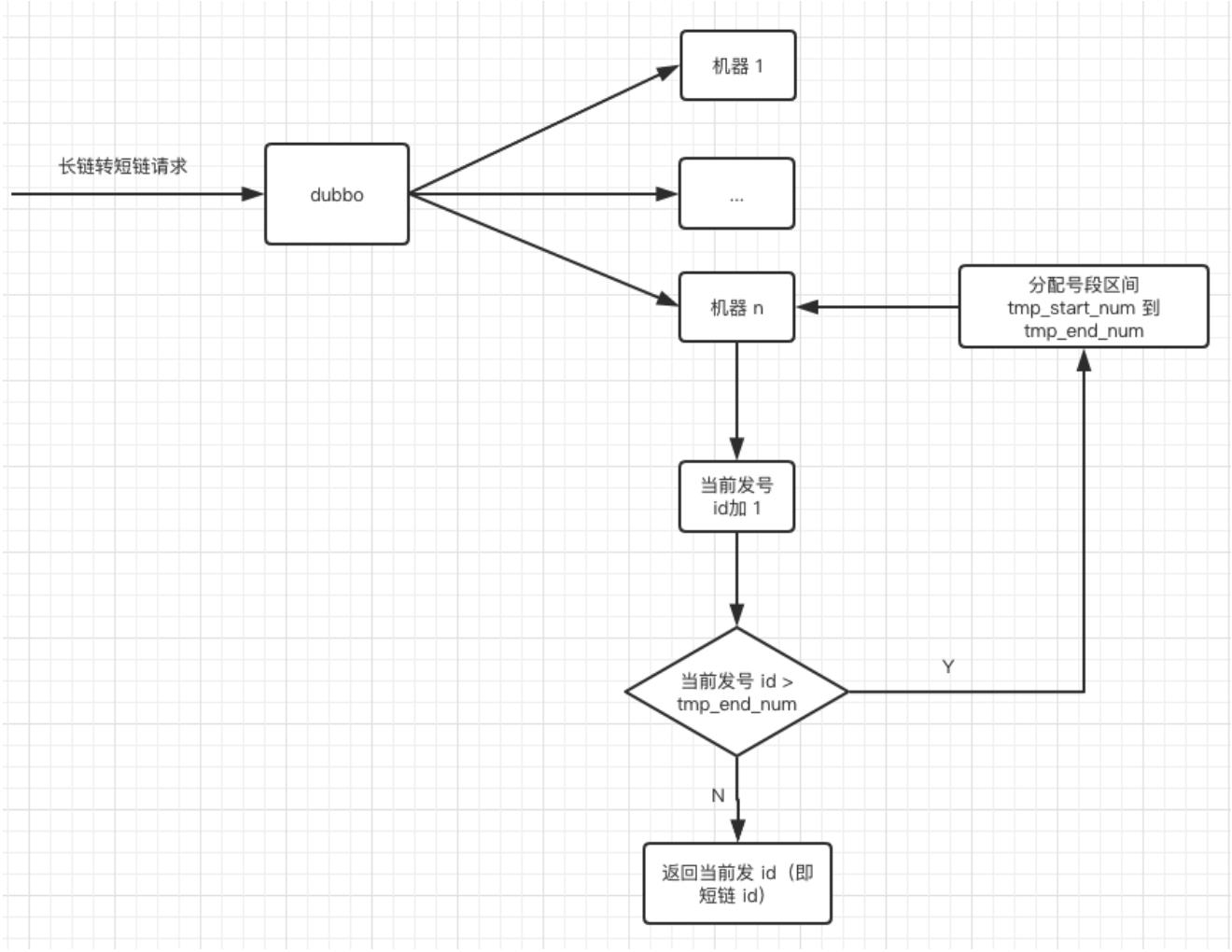
id	gmt_create	tmp_start_num	tmp_end_num
1	1536389934	1	1000
2	1536389967	1001	2000
3	1536389999	2001	3000
4	1536390028	3001	4000
5	1536390055	4001	5000

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



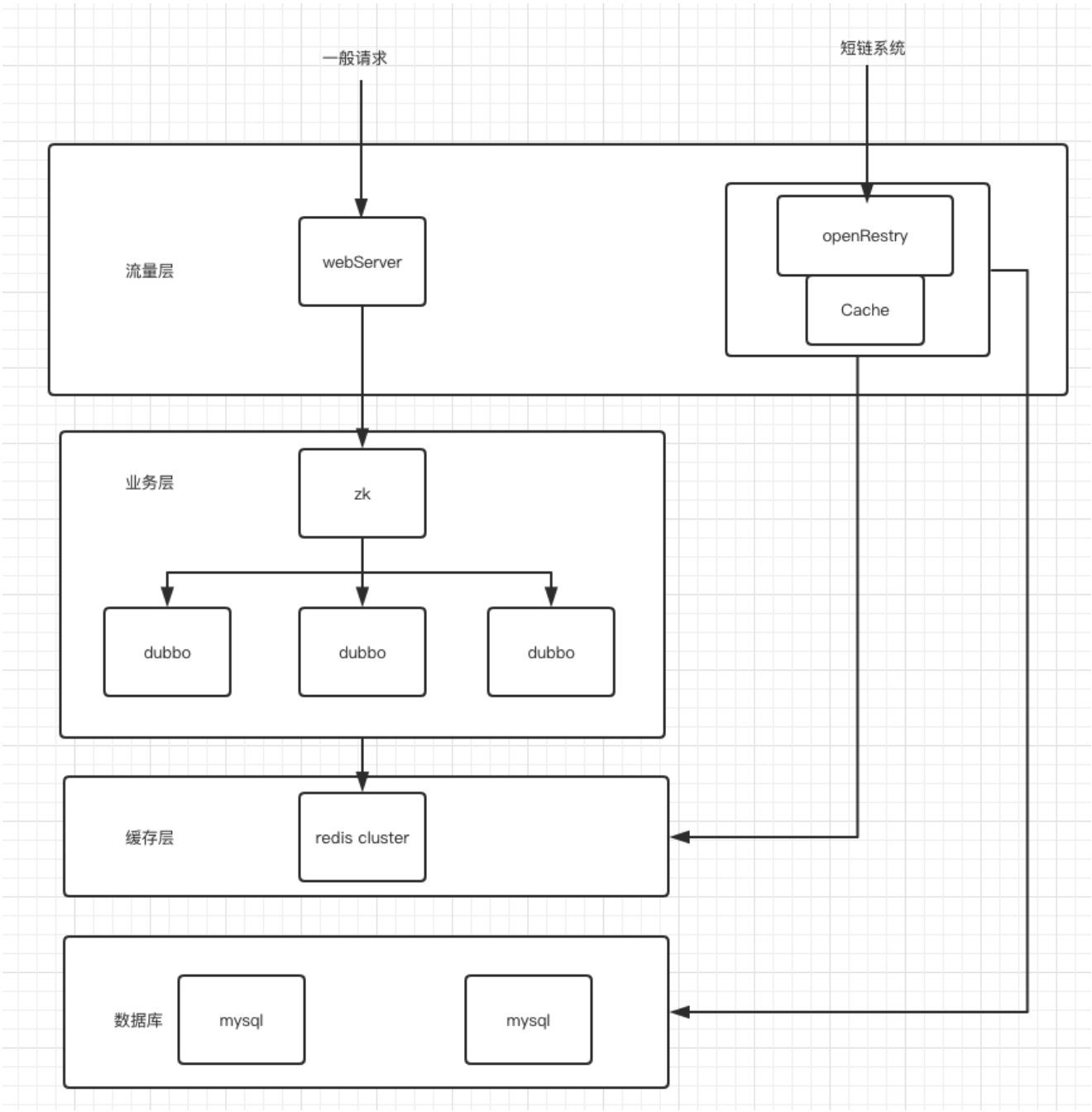
解决了发号器问题，接下来就简单了，从发号器拿过来的 id ，即为短链 id，接下来我们再创建一个长短链的映射表即可， 短链 id 即为主键，不过这里有个需要注意的地方，我们可能需要防止多次相同的长链生成不同的短链 id 这种情况，这就需要每次先根据长链来查找 db 看是否存在相关记录，一般的做法是给长链加索引，但这样的话索引的空间会很大，所以我们可以对长链适当的压缩，比如 md5，再对长链的 md5 字段做索引，索引就会小很多。这样只要根据长链的 md5 去表里查是否存在相同的记录即可。所以我们设计的表如下

```
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',
  `gmt_create`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。

高性能短链的架构设计

在电商公司，经常有很多活动，秒杀，抢红包等等，在某个时间点的 QPS 会很高，考虑到这种情况，我们引入了 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。

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 等技术，文中没有展开讲，建议大家回头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，相信大家各个知识点都吃透后会收获不小。

程序员田螺

专注分享后端面试题，包括计算机网络、MySql数据库、Redis缓存、操作系统、Java后...

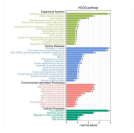
11篇原创内容

公众号

喜欢此内容的人还喜欢

【单细胞高级绘图】07.KEGG富集结果展示

TOP生物信息



我们是如何记录图片的?

Tecvan

干货|最全Web 渗透测试信息搜集-CheckList

编码安全研究