我为什么要使用哈希

15 OCTOBER 2015

什么是哈希（Hash）

本来这里不应该出现这一节的，因为实际上大家应该都知道什么是哈希。不过有时候为了文章的完整性，我这里就稍微教条性地说明一下吧。ヽ(◕ฺ◡◔)ﾉ

> 散列（英语：*Hashing*），通常音译作哈希，是电脑科学中一种对资料的处理方法，通过某种特定的函数、算法将要检索的项与用来检索的索引关联起来，生成一种便于搜索的数据结构。也译为散列。
>
> *-- From 散列, Wikipedia*

实际上通俗的说法就是把某种状态或者资料给映射到某个值上的操作。

本酱大概就解释到这里了，至于哈希的进一步认知包括冲突的产生和解决等，如果米娜桑不了解的话还请自行学习咕。ଘ੭ଘ

引子——子树问题

这个不是我在实践中遇到的问题，而是当年去某不作恶的大厂面试时候遇到的问题，觉得比较经典，所以就拿出来了。ᕕ(ᐛ)ᕗ

问题描述

给定一棵二叉树，假设每个节点的数据只有左右子节点，自身并不存储数据。请找出两两完全相等的子树们。

> 有兴趣的童鞋可以自己先思考一下。(ง'ω')ง

我的做法

实际上我也不知道自己的做法是不是正确做法，不过既然通过了那一轮面试，想来也不会偏差到哪去喵。ლ(╹ε╹ ლ)

做法大概如下：

1. 后序遍历一遍整棵树。
2. 对于遍历到每一个节点，都获取到左右子节点的哈希值，然后将其拼接重新计算出自身的哈希值，并返回给父亲节点。

至于哈希值怎么算，方法有很多。最简单的就是设叶子节点一个哈希值，比如是 `md5("")`，然后每次非叶子节点的哈希值就用 `md5(LEFT_HASH + RIGHT_HASH)` 来计算。大家也可以自己随便想一种方法来做就好了。

很多人可能不解了，明明是用 `md5`，这篇文章是讲哈希，有毛线关系。(ノ°O°)ノ┻┻

实际上 `md5` 就是一种哈希算法，而且是非常经典的哈希算法。

> 典型的哈希算法包括 *MD2*、*MD4*、*MD5* 和 *SHA-1* 等。当然不局限于这些，对于数字来说，取模也算是哈希算法，对于字符串状态转整数状态哈希来说还有诸如 *BKDR*、*ELF* 等等。

如果大家想多了解一些字符串转数字哈希的算法，可以参考一下 BYVoid 的这篇《各种字符串Hash函数比较》，或者想直接在 Node.js 里面使用的小伙伴们可以光顾下这个包——bling-hashes。

初步的轮廓已经明晰了，说白了就是将每个节点的哈希全算出来，如果是父亲节点就用子节点的哈希拼接起来再哈希一遍。σ`∀´)σ

把这些哈希算出来之后放在一个散列表里面待查。如果一个算出来的哈希跟之前已有的哈希值相等，那么就是说这个节点跟那个节点为根节点的子树有可能完全相等。

> 注意：**有可能**完全相等。
>
> 注意：只是**有可能**完全相等。
>
> 注意：重要的事情说三遍，只是**有可能**完全相等。

哈希是存在着一定的冲突概率的，所以说两个相等的哈希所检索到的源不一定一样，所以我们根据这些计算到的哈希建立哈希表，然后把表中同哈希值的子树再两两同时遍历一遍以检验是否相等。

1. 同时递归，取两个子树的根节点。
2. 后序遍历，看看每个节点是不是都一样存在（或者不存在）左子节点以及存在（或者不存在）右子节点。
3. 循环往复一直到两两遍历完整棵树得到验证结果。如果半路有一个节点的左右子节点状态不一样就可以直接跳出递归返回 `false`。

至此为止，我们可以看出大概是两大步——**计算各子树的哈希值**和**验证各同哈希子树的相等性**。不过稍微变通一下，我们就可以在计算出哈希值的时候就去跟以前的对比了。

剪枝

实际上上面的做法还有一个优化的方案，不过跟哈希相关性已经基本上很小了。不过还是跟**解决冲突**有一丢丢的关系的，没兴趣的童

鞋也可以直接跳过了。(๑•́ ₃ •̀๑)

由于子树哈希值是存在一定的冲突概率的，所以两个同哈希的子树不一定相同。那么我们如果能一眼看出这样的两棵子树是不相等的，就可以省略验证这一个递归的步骤了。

这里有一种最显而易见的情况我们是可以忽略省略步骤的，那就是深度。

如果两棵子树两两完全相等，那么说明这俩基佬的深度（或者说高度）是一样的，如果连深度都不一样了还如何愉快搞基——所以说如果有两个相等哈希值的子树的深度不一样的话可以直接略过验证步骤了。

那么就可以这么做：

1. 设所有叶子节点的深度为 `0`，然后每往上一层加一。
2. 遇到左右子节点深度不一样的父节点时，取深度大的那个子节点深度去加一。

以上步骤在遍历计算哈希的时候顺便也做了，这样就多了一个验证标记了。

所以差不多就这样了，浅尝辄止。(˘·з·)

引子的小结

就上述的场景来说，哈希非常好地将一个非常复杂的状态转化成一个可以检索的状态。本来毫无头绪的一个问题使用了哈希之后就完全变成了一个检索加验证的过程了。

报告图问题

这个问题就是我在大搜车中确实遇到的场景了。大家也不需要知道什么是报告图，就当它是一个代号了。

问题描述

> 要做的事情大概就是说给定一个报告，我们根据报告的各个细节选定各种图层然后揉成一团叠加在一起形成最后一个结果图。

其实本来就有个系统在做这件事情的——每来一个报告就生成一张图，然后存储好之后给前端使用。

我做的事情是将逻辑迁移到另一套计算密集型任务集中处理系统中去。(´艸`)

其实生成这样一张图片的逻辑是 CPU 计算密集型的逻辑，所以比较耗费资源和时间的，那么我们就能在这上面做点手脚优化一下。

优化方法

首先我们要知道的是，有哪些图层是固定的，所以其实这算半个排列组合的问题了。

不过我们也知道排列组合的增长性非常快，更何况我这里有约 100 个图层选择，所以可能性非常多，一下子全生成好不可能。

那么就可以用哈希和懒惰的思想来实现了。(ˇωˇ人)

虽然报告是有无限种可能的，但是把报告转成图层数据之后，拥有完全一样的图层数据的报告就可以用同一张图片了，这样就可以大大节省空间和时间了。

其实大概的步骤非常简单：

1. 把图层数据计算成哈希。（比如把所有图层文件路径用某种符号拼接，再用 `md5` 计算一下）
2. 去数据库查找这个哈希主键存不存在。
    - 如果存在则验证源图层数据域当前图层数据是否吻合。
        - 如果不吻合则按某种算法重新计算哈希，继续步骤2。
        - **如果吻合则可以直接拿着这个数据返回了，跳出计算。**
    - 如果不存在就说明当前数据库还没有这个图层情况的报告图生成，那么就执行生成报告图逻辑。
3. 报告图生成之后，将其存入数据库中。
    - 计算出这个报告图图层数据的哈希，去数据库查存不存在。
        - 如果不存在则说明哈希不冲突，能用，直接用这个哈希存进去。
        - 如果存在则说明哈希冲突，那么按某种算法重新计算哈希，继续上面的步骤直到不冲突为止。

如果大家想知道“按某种算法重新生成哈希”里面“某种算法”的话可以看看下面的瞎狗眼的说明了。(ﾉ◕ヮ◕)ﾉ*:･ﾟ✧

> 其实很简单，把图层数据的这个字符串加某个固定字符当小尾巴，如果哈希还是冲突则继续加这个小尾巴，直到计算出来的哈希不冲突为止。
>
> 比如我就用了这字符当小尾巴——□（麻将牌中的蘭）。(♛‿♛)

报告图的小结

在这种场景中，我把哈希拿来作检索某种报告图是否已经生成的用途。如果没有生成则生成一张，如果已经生成则直接拿已有的报告图去用。

至少比原来的来一张报告就生成一张图片来得快，并且省空间——相当于作冗余处理了。

事实上在很多的网盘系统中也有作冗余处理的。你以为你有多少多少 T 的空间，实际上相同的文件最终在网盘系统里面只存一份（不过排除备份的那些），而我相信做这些冗余判断的原理就是哈希了，SHA-1 也好 MD5 也好，反正就是这样。

> 上面网盘的冗余处理原理也只是我的猜测，我没在那些厂子里面工作过所以不能说就是就是这样子的。欢迎指正。。゜ヽ(゜´Д`)ﾉ゜。

唯一主键问题

这是我来这边工作后的另一个小插曲了，遇到一个主键生成的小需求。

问题描述

有一个数据要插入到数据库，所以要给它生成一个主键，但是需求比较奇葩，可能是历史遗留问题吧。(ㄴ^ㄴ)

- 非自增。
- 是一个全是数字的字符串。
- 不同类型的这个表的数据用不同的前缀，比如 `10`、`11`、`12` 等。
- 位数在十几位左右（不过在我这里就固定了）。

解决方案

如果是 `前缀 + 随机数` 的冲突概率会比较大的，所以还是用哈希来搞。

非常简单。首先前缀是固定的，我们就不管了，然后我根据这次进来的数据拼接成字符串（数据不会完全一样的），加上一点随机盐，然后用字符串哈希计算一遍，加上前导零，加上当前时间戳的后几位拼接起来，最后接上前缀就好了。

这个 `generate` 函数看起来就像这样子：

```
var bling = require("bling-hashes");
function generate(type, bodyParamStr) {
    var basePrefix;
    switch(type) {
        case 'foo': basePrefix = '10'; break;
        case 'bar': basePrefix = '11'; break;
        default: base_prefix = '00';
    }

    var date = moment();
    var hash = bling.bkdr(bodyParamStr + date.valueOf()).pad(10);
    hash = date.millisecond().pad(3) + hash;

    return basePrefix + hash;
};
```

> **注意：**这里的 *`bling`* 就是上面提到过的那个 *bling-hashes*，采用了 *`BKDR`* 算法来计算哈希。以及 *`Number.prototype.pad`* 函数是我邪恶得使用了 *SugarJs* 里面的函数，就是加上前导零的意思。如果受“千万不要修改原型链”影响较深地童鞋别学我哦。 *`bodyParamStr`* 是前端传过来的 ***Raw Form Data***，它看起来像 *`"data1=1&data2=2&..."`*。

最后得到的这个字符串是我们所要的主键了。。:.ﾟヽ(*´∀`)ﾉﾟ.:。

不过要注意的是，这个主键仍然又冲突的可能性，所以一旦冲突了（无论是自己检测到的还是插入数据库的时候疼了）就需要再生产一遍。就目前来说再生成的时候毫秒时间戳后三位会不一样，所以问题不大，允许存在的误差——毕竟不是那种分分钟集千万条的数据，肯定在 `int` 范围内。如果到时候真出问题了再改进。

主键的小结

这里的哈希是用在生成基本上没有碰撞的主键身上，感觉效果也是非常不错的——前提是你也有这种奇葩需求。

真·小结

本文大致介绍了哈希的几种用途，有可能是大家熟知的用途，也有可能是巧用，总之就是说了为什么我要用哈希。

在编程中，无论是实际用途还是自己玩玩的题目，多动动脑子就会出来一些“奇技淫巧”。哈希也好，别的东西也罢，反正都是为了解决问题的——千万别因为实际开发中通常性的“并没有什么卵用”而去忽视它们，虽然哈希已经是够常用的了。(๑•̀ω•́๑)