小时到分钟 - 一步步优化巨量关键词的匹配

问题由来

前些天工作中遇到一个问题：

> 有 60万 条短消息记录日志，每条约 50 字，5万 关键词，长度 2-8 字，绝大部分为中文。要求将这 60万 条记录中包含的关键词全部提取出来并统计各关键词的命中次数。

本文完整介绍了我的实现方式，看我如何将需要运行十小时的任务优化到十分钟以内。虽然实现语言是 PHP，但本文介绍的更多的思想，应该能给大家一些帮助。

原始 - grep

设计

一开始接到任务的时候，我的小心思立刻转了起来， `日志 + 关键词 + 统计` ，我没有想到自己写代码实现，而是首先想到了 linux 下常用的日志统计命令 `grep` 。

`grep` 命令的用法不再多提，使用 `grep 'keyword' | wc -l` 可以很方便地进行统计关键词命中的信息条数，而php的 `exec()` 函数允许我们直接调用 linux 的 shell 命令，虽然这样执行危险命令时会有安全隐患。

代码

上伪代码：

```
foreach ($word_list as $keyword) {
    $count = intval(exec("grep '{$keyword}' file.log | wc -l"));
    record($keyword, $count);
}
```

在一台老机器上跑的，话说老机器效率真的差，跑了6小时。估计最新机器2-3小时吧，后面的优化都使用的新机器，而且需求又有变动，正文才刚刚开始。

原始，原始在想法和方法。

进化 - 正则

设计

交了差之后，第二天产品又提出了新的想法，说以后想把某数据源接入进来，消息以数据流的形式传递，而不再是文件了。而且还要求了消息统计的实时性，一下把我想把数据写到文件再统计的想法也推翻了，为了方案的可扩展

性，现在的统计对象不再是一个整体，而是要考虑拿n个单条的消息来匹配了。

这时，略懵的我只好祭出了最传统的工具- 正则 。正则的实现也不难，各个语言也都封装好了正则匹配函数，重点是模式(pattern)的构建。

当然这里的模式构建也不难， /keyword1|keword2|.../ ，用 | 将关键词连接起来即可。

正则小坑

这里介绍两个使用中遇到的小坑：

- 正则模式长度太长导致匹配失败：

  PHP 的正则有回溯限制，以防止消耗掉所有的进程可用堆栈，最终导致 php 崩溃。太长的模式会导致 PHP 检测到回溯过多，中断匹配，经测试默认设置时最大模式长度为 32000 字节 左右。

  php.ini 内 pcre.backtrack_limit 参数为最大回溯次数限制，默认值为 1000000，修改或 php.ini 或在脚本开始时使用 ini_set('pcre.backtrack_limit', n); 将其设置为一个较大的数可以提高单次匹配最大模式长度。当然也可以将关键词分批统计（我用了这个=_=）。

- 模式中含有特殊字符导致大量warning：

  匹配过程中发现 PHP 报出大量 warning： unknown modifier 乱码 ，仔细检查发现关键词中有 / 字符，可以使用 preg_quote() 函数过滤一遍关键词即可。

代码

上伪代码：

```
$end = 0;
$step = 1500;
$pattern = array();
// 先将pattern 拆成多个小块
while ($end < count($word_list)) {
    $tmp_arr = array_slice($word_list, $end, $step);
    $end += $step;
    $item = implode('|', $tmp_arr);
    $pattern[] = preg_quote($item);
}

$content = file_get_contents($log_file);
$lines = explode("\n", $content);
foreach ($lines as $line) {
    // 使用各小块pattern分别匹配
    for ($i = 0; $i < count($pattern); $i++) {
        preg_match_all("/{$pattern[$i]}/", $line, $match);
    }
    $match = array_unique(array_filter($match));
    dealResult($match);
}
```

为了完成任务，硬着头皮进程跑了一夜。当第二天我发现跑了近十个小时的时候内心是崩溃的。。。太慢了，完全达不到使用要求，这时，我已经开始考虑改换方法了。

当产品又改换了关键词策略，替换了一些关键词，要求重新运行一遍，并表示还会继续优化关键词时，我完全否定了现有方案。绝对不能用关键词去匹配信息，这样一条一条用全部关键词去匹配，效率实在是不可忍受。

进化，需求和实现的进化

觉醒 - 拆词

设计

我终于开始意识到要拿信息去关键词里对比。如果我用关键词为键建立一个 hash 表，用信息里的词去 hash 表里查找，如果查到就认为匹配命中，这样不是能达到 O(1) 的效率了么？

可是一条短消息，我如何把它拆分为刚好的词去匹配呢，分词？分词也是需要时间的，而且我的关键词都是些无语义的词，构建词库、使用分词工具又是很大的问题，最终我想到 拆词 。

为什么叫拆词呢，我考虑以蛮力将一句话拆分为 所有可能的 词。如 我是好人 就可以拆成 我是、是好、好人、我是好、是好人、我是好人 等词，我的关键词长度为 2-8，所以可拆词个数会随着句子长度迅速增加。不过，可以用标点符号、空格、语气词（如 的、是 等）作为分隔将句子拆成小短语再进行拆词，会大大减少拆出的词量。

其实分词并没有完整实现就被后一个方法替代了，只是一个极具实现可能的构想，写这篇文章时用伪代码实现了一下，供大家参考，即使不用在匹配关键词，用在其他地方也是有可能的。

代码

```
$str_list = getStrList($msg);
foreach ($str_list as $str) {
    $keywords = getKeywords($str);
    foreach ($keywords as $keyword) {
        // 直接通过PHP数组的哈希实现来进行快速查找
        if (isset($word_list[$keyword])) {
            record($keyword);
        }
    }
}
/**
 * 从消息中拆出短句子
 */
function getStrList($msg) {
    $str_list = array();
    $seperators = array(', ', '。', '的', ...);

    $words = preg_split('/(?<!^)(?!$)/u', $msg);
    $str = array();
    foreach ($words as $word) {
        if (in_array($word, $seperators)) {
            $str_list[] = $str;
            $str = array();
        } else {
```

```
                $str[] = $word;
            }
        }

        return array_filter($str_list);
    }

    /**
     * 从短句中取出各个词
     */
    function getKeywords($str) {
        if (count($str) < 2) {
            return array();
        }

        $keywords = array();
        for ($i = 0; $i < count($str); $i++) {
            for ($j = 2; $j < 9; $j++) {
                $keywords[] = array_slice($str, $i, $j); // todo 限制一下不要超过数组最大
            }
        }

        return $keywords;
```

结果

我们知道一个 `utf-8` 的中文字符要占用三个字节，为了拆分出包含中英文的每一个字符，使用简单的 `split()` 函数是做不到的。

这里使用了 `preg_split('/(?<!^)(?!$)/u', $msg)` 是通过正则匹配到两个字符之间的''来将两个字符拆散，而两个括号里的 `(?<!^)(?!$)` 是分别用来限定捕获组不是第一个，也不是最后一个（不使用这两个捕获组限定符也是可以的，直接使用 `//` 作为模式会导致拆分结果在前后各多出一个空字符串项）。 捕获组的概念和用法可见我之前的博客 PHP正则中的捕获组与非捕获组

由于没有真正实现，也不知道效率如何。估算每个短句长度约为 10 字左右时，每条短消息约50字左右，会拆出 200 个词。虽然它会拆出很多无意义的词，但我相信效率绝不会低，由于其 hash 的高效率，甚至我觉得会可能比终极方法效率要高。

最终没有使用此方案是因为它对句子要求较高，拆词时的分隔符也不好确定，最重要的是它不够优雅。。。这个方法我不太想去实现，统计标识和语气词等活显得略为笨重，而且感觉拆出很多无意义的词感觉效率浪费得厉害。

觉醒，意识和思路的觉醒

终级 - Trie树

trie树

于是我又来找谷哥帮忙了，搜索大量数据匹配，有人提出了 使用 trie 树的方式，没想到刚学习的 trie 树的就派上了用场。我上上篇文章刚介绍了 trie 树，在空间索引 - 四叉树 里 字典树 这一小节，大家可以查看一下。

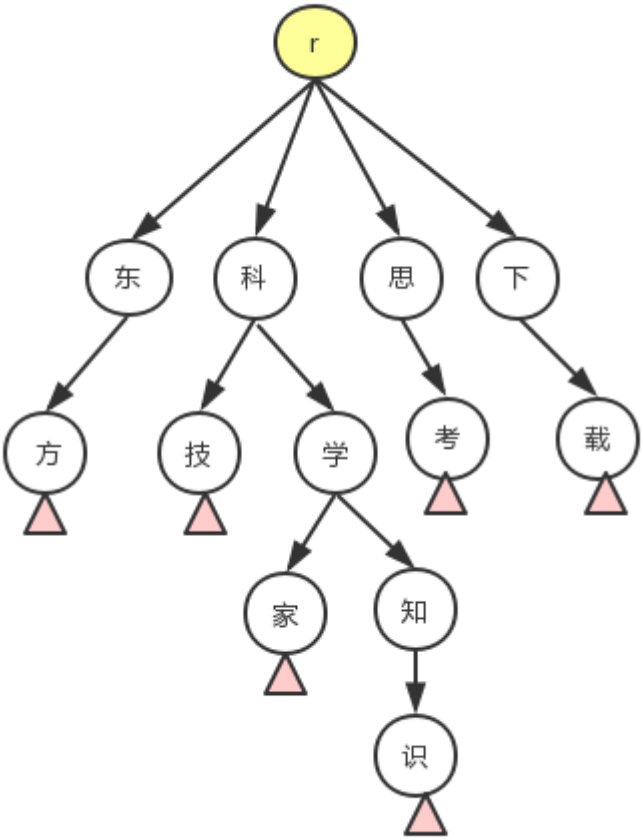
当然也为懒人复制了一遍我当时的解释(看过的可以跳过这一小节了)。

> 字典树，又称前缀树或 trie 树，是一种有序树，用于保存关联数组，其中的键通常是字符串。与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。

我们可以类比字典的特性：我们在字典里通过拼音查找晃(huang)这个字的时候，我们会发现它的附近都是读音为 huang 的，可能是声调有区别，再往前翻，我们会看到读音前缀为 huan 的字，再往前，是读音前缀为 hua 的字... 取它们的读音前缀分别为 h qu hua huan huang 。我们在查找时，根据 abc...xyz 的顺序找到 h 前缀的部分，再根据 ha he hu 找到 hu 前缀的部分...最后找到 huang ，我们会发现，越往后其读音前缀越长，查找也越精确，这种类似于字典的树结构就是字典树，也是前缀树。

设计

那么 trie 树怎么实现关键字的匹配呢? 这里以一幅图来讲解 trie 树匹配的过程。



其中要点：

构造trie树

1. 将关键词用上面介绍的 preg_split() 函数拆分为单个字符。如 科学家 就拆分为 科、学、家 三个字符。
2. 在最后一个字符后添加一个特殊字符 ` ，此字符作为一个关键词的结尾（图中的粉红三角），以此字符来标识查到了一个关键词（不然，我们不知道匹配到 科、学 两个字符时算不算匹配成功）。

3. 检查根部是否有第一个字符(科)节点，如果有了此节点，到 步骤4 。 如果还没有，在根部添加值为 科 的节点。
4. 依次检查并添加 学、家 两个节点。
5. 在结尾添加 ` 节点，并继续下一个关键词的插入。

匹配

然后我们以 这位科学家很了不起！ 为例来发起匹配。

- 首先我们将句子拆分为单个字符 这、位、... ；
- 从根查询第一个字符 这 ，并没有以这个字符开头的关键词，将字符“指针”向后移，直到找到根下有的字符节点 科 ；
- 接着在节点 科 下寻找值为 学 节点，找到时，结果子树的深度已经到了2，关键词的最短长度是2，此时需要在 学 结点下查找是否有 ` ，找到意味着匹配成功，返回关键词，并将字符“指针”后移，如果找不到则继续在此结点下寻找下一个字符。
- 如此遍历，直到最后，返回所有匹配结果。

代码

完整代码我已经放到了GitHub上：Trie-GitHub-zhenbianshu，这里放上核心。

首先是数据结构树结点的设计，当然它也是重中之重：

```
 $node = array(
    'depth' => $depth, // 深度，用以判断已命中的字数
    'next' => array(
        $val => $node, // 这里借用php数组的哈希底层实现，加速子结点的查找
        ...
    ),
);
```

然后是树构建时子结点的插入：

```
// 这里要往节点内插入子节点，所以将它以引用方式传入
private function insert(&$node, $words) {
         if (empty($words)) {
            return;
        }
        $word = array_shift($words);
        // 如果子结点已存在，向子结点内继续插入
        if (isset($node['next'][$word])) {
            $this->insert($node['next'][$word], $words);
        } else {
            // 子结点不存在时，构造子结点插入结果
            $tmp_node = array(
                'depth' => $node['depth'] + 1,
                'next' => array(),
            );
            $node['next'][$word] = $tmp_node;
            $this->insert($node['next'][$word], $words);
```

```
        }
```

最后是查询时的操作：

```
    // 这里也可以使用一个全局变量来存储已匹配到的字符，以替换$matched
    private function query($node, $words, &$matched) {
            $word = array_shift($words);
            if (isset($node['next'][$word])) {
                // 如果存在对应子结点，将它放到结果集里
                array_push($matched, $word);
                // 深度到达最短关键词时，即可判断是否到词尾了
                if ($node['next'] > 1 && isset($node['next'][$word]['next']['``'])) {
                    return true;
                }
                return $this->query($node['next'][$word], $words, $matched);
            } else {
                $matched = array();
                return false;
            }
        }
```

结果

结果当然是喜人的，如此匹配，处理一千条数据只需要3秒左右。找了 Java 的同事试了下，Java 处理一千条数据只需要1秒。

这里来分析一下为什么这种方法这么快：

- 正则匹配：要用所有的关键词去信息里匹配匹配次数是 `key_len * msg_len` ，当然正则会进行优化，但基础这样，再优化效率可想而知。
- 而 trie 树效率最差的时候是 `msg_len * 9(最长关键词长度 + 1个特殊字符)` 次 hash 查找，即最长关键词类似 `AAA` ，信息内容为 `AAA...` 时，而这种情况的概率可想而知。

至此方法的优化到此结束，从每秒钟匹配 10 个，到 300 个，30 倍的性能提升还是巨大的。

终级，却不一定是终极

他径 - 多进程

设计

匹配方法的优化结束了，开头说的优化到十分钟以内的目标还没有实现，这时候就要考虑一些其他方法了。

我们一提到高效，必然想到的是 `并发` ，那么接下来的优化就要从并发说起。PHP 是单线程的（虽然也有不好用的多线程扩展），这没啥好的解决办法，并发方向只好从多进程进行了。

那么一个日志文件，用多个进程怎么读呢？这里当然也提供几个方案：

- 进程内添加日志行数计数器，各个进程支持传入参数 n，进程只处理第 `行数 % n = n` 的日志，这种 hack 的反向分布式我已经用得很熟练了，哈哈。

  这种方法需要进程传参数，还需要每个进程都分配读取整个日志的的内存，而且也不够优雅。

- 使用 linux 的 `split -l n file.log output_pre` 命令，将文件分割为每份为 n 行的文件，然后用多个进程去读取多个文件。

  此方法的缺点就是不灵活，想换一下进程数时需要重新切分文件。

- 使用 Redis 的 list 队列临时存储日志，开启多个进程消费队列。

  此方法需要另外向 Redis 内写入数据，多了一个步骤，但它扩展灵活，而且代码简单优雅。

最终使用了第三种方式来进行。

结果

这种方式虽然也会有瓶颈，最后应该会落在 Redis 的网络 IO 上。我也没有闲心开 n 个进程去挑战公司 Redis 的性能，运行 10 个进程三四分钟就完成了统计。即使再加上 Redis 写入的耗时，10分钟以内也妥妥的。

一开始产品对匹配速度已经有了小时级的定位了，当我 10 分钟就拿出了新的日志匹配结果，看到产品惊讶的表情，心里也是略爽的，哈哈~

他径，也能帮你走得更远

总结

解决问题的方法有很多种，我认为在解决各种问题之前，要了解很多种知识，即使只知道它的作用。就像一个工具架，你要先把工具尽量摆得多，才能在遇到问题时选取一个最合适的。接着当然要把这些工具用是纯熟了，这样才能使用它们去解决一些怪异问题。

工欲善其事，必先利其器，要想解决性能问题，掌握系统级的方法还略显不够，有时候换一种数据结构或算法，效果可能会更好。感觉自己在这方面还略显薄弱，慢慢加强吧，各位也共勉。