

13 | 实时统计：链路跟踪实时计算中的实用算法

2022-11-21 徐长龙 来自北京



天下无鱼

<https://shikey.com/>

《高并发系统实战课》

[课程介绍 >](#)



讲述：徐长龙

时长 14:15 大小 13.03M



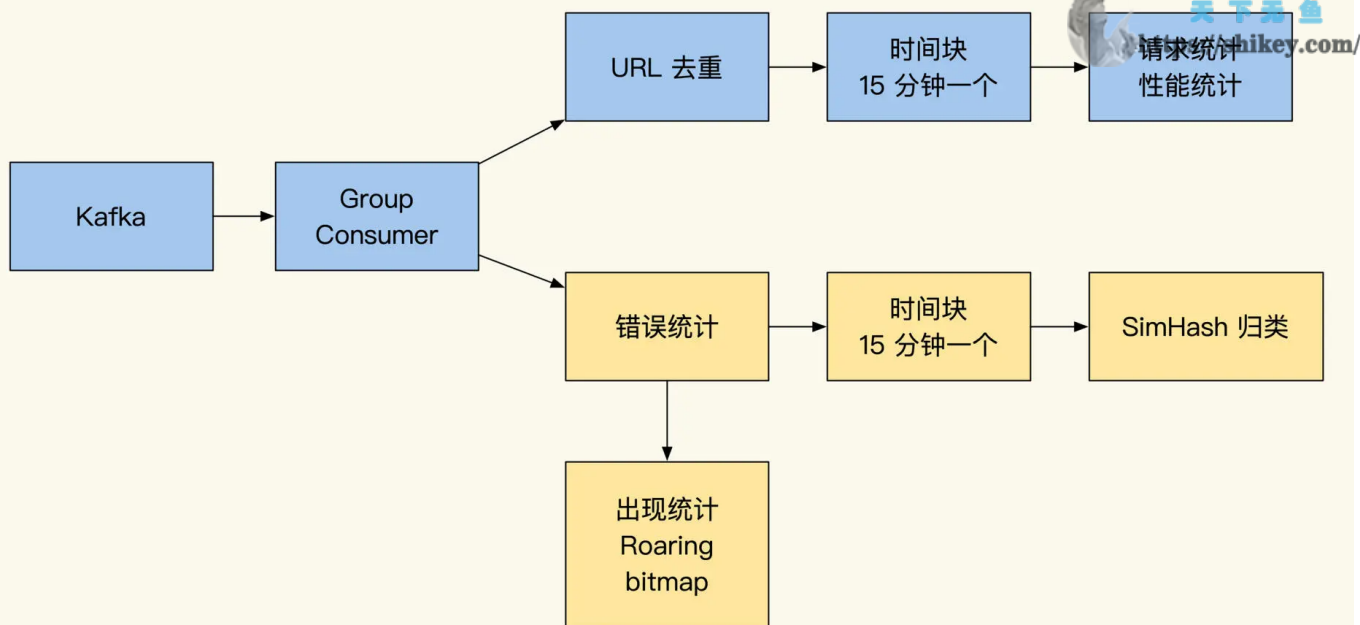
你好，我是徐长龙。

前几节课我们了解了 ELK 架构，以及如何通过它快速实现一个定制的分布式链路跟踪系统。不过 ELK 是一个很庞大的体系，使用它的前提是我们至少要有性能很好的三台服务器。

如果我们的数据量很大，需要投入的服务器资源就更多，之前我们最大一次的规模，投入了大概 2000 台服务器做 ELK。但如果我们的服务器资源很匮乏，这种情况下，要怎样实现性能分析统计和监控呢？

当时我只有两台 4 核 8G 服务器，所以我用了一些巧妙的算法，实现了本来需要大量服务器并行计算，才能实现的功能。这节课，我就给你分享一下这些算法。

我先把实时计算的整体结构图放出来，方便你建立整体印象。



实时计算的整体结构图

从上图可见，我们实时计算的数据是从 **Kafka** 拉取的，通过进程实时计算统计 **Kafka** 的分组消费。接下来，我们具体看看这些算法的思路和功用。

URL 去参数聚合

做链路跟踪的小伙伴都会很头疼 **URL** 去参数这个问题，主要原因是很多小伙伴会使用 **RESTful** 方式来设计内网接口。而做链路跟踪或针对 **API** 维度进行统计分析时，如果不做整理，直接将这些带参数的网址录入到统计分析系统中是不行的。

同一个 **API** 由于不同的参数无法归类，最终会导致网址不唯一，而成千上万个“不同”网址的 **API** 汇总在一起，就会造成统计系统因资源耗尽崩掉。除此之外，同一网址不同的 **method** 操作在 **RESTful** 中实际也是不同的实现，所以同一个网址并不代表同一个接口，这更是给归类统计增加了难度。

为了方便你理解，这里举几个 **RESTful** 实现的例子：

- **GET** geekbang.com/user/1002312/info 获取用户信息
- **PUT** geekbang.com/user/1002312/info 修改用户信息

- **DELETE** `geekbang.com/user/1002312/friend/123455` 删除用户好友

可以看到我们的网址中有参数，虽然是同样的网址，但是 **GET** 和 **PUT** 方法代表的意义并不一样，这个问题在使用 **Prometheus**、**Trace** 等工具时都会出现。

一般来说，碰到这种问题，我们都会先整理数据，再录入到统计分析系统当中。我们有两种常用方式来对 **URL** 去参数。

第一种方式是**人工配置替换模板**，也就是人工配置出一个 **URL** 规则，用来筛选出符合规则的日志并替换掉关键部分的参数。

我一般会用一个类似 **Trie Tree** 保存这个 **URL** 替换的配置列表，这样能够提高查找速度。但是这个方式也有缺点，需要人工维护。如果开发团队超过 200 人，列表需要时常更新，这样维护起来会很麻烦。

复制代码

```
1 类Radix tree效果:
2 /user
3 - /*
4 - - /info
5 - - - :GET
6 - - - :PUT
7 - - /friend
8 - - - /*
9 - - - - :DELETE
```

具体实现是将网址通过 **/** 进行分割，逐级在前缀搜索树查找。

我举个例子，比如我们请求 **GET /user/1002312/info**，使用树进行检索时，可以先找到 **/user** 根节点。然后在 **/user** 子节点中继续查找，发现有元素 **/***（代表这里替换）而且同级没有其他匹配，那么会被**记录为这里可替换**。然后需要继续查找 **/*** 下子节点 **/info**。到这里，网址已经完全匹配。

在网址更深一层是具体请求 **method**，我们找到 **GET** 操作，即可完成这个网址的配置匹配。然后，直接把 **/*** 部分的 **1002312** 替换成固定字符串即可，替换的效果如下所示：

复制代码

1 GET /user/1002312/info 替换成 /user/replaced/info



天下无鱼

http://shixue.com/


另一种方式是数据特征筛选，这种方式虽然会有误差，但是实现简单，无需人工维护。这个方法是我推崇的方式，虽然这种方式有可能有失误，但是确实比第一种方式更方便。

具体请看后面的演示代码：

复制代码

```
1 //根据数据特征过滤网址内参数
2 function filterUrl($url)
3 {
4     $urlArr = explode("/", $url);
5
6     foreach ($urlArr as $urlIndex => $urlItem) {
7         $totalChar = 0; //有多少字母
8         $totalNum = 0; //有多少数值
9         $totalLen = strlen($urlItem); //总长度
10
11         for ($index = 0; $index < $totalLen; $index++) {
12             if (is_numeric($urlItem[$index])) {
13                 $totalNum++;
14             } else {
15                 $totalChar++;
16             }
17         }
18
19         //过滤md5 长度32或64 内容有数字 有字符混合 直接认为是md5
20         if (($totalLen == 32 || $totalLen == 64) && $totalChar > 0 && $totalNum > 0) {
21             $urlArr[$urlIndex] = "*md*";
22             continue;
23         }
24
25         //字符串 data 参数是数字和英文混合 长度超过3(回避v1/v2一类版本)
26         if ($totalLen > 3 && $totalChar > 0 && $totalNum > 0) {
27             $urlArr[$urlIndex] = "*data*";
28             continue;
29         }
30
31         //全是数字在网址中认为是id一类， 直接进行替换
32         if ($totalChar == 0 && $totalNum > 0) {
33             $urlArr[$urlIndex] = "*num*";
34             continue;
35         }
36     }
37     return implode("/", $urlArr);
38 }
```

通过这两种方式，可以很方便地将我们的网址替换成后面这样：

- GET geekbang.com/user/**1002312**/info => geekbang.com/user/*num*/info_GET  天下无鱼
<https://shikey.com/>
- PUT geekbang.com/user/**1002312**/info => geekbang.com/user/*num*/info_PUT
- DELETE geekbang.com/user/**1002312**/friend/**123455** =>
geekbang.com/user/*num*/friend/*num*_DEL

经过过滤，我们的 API 列表是不是清爽了很多？这时再做 API 进行聚合统计分析的时候，就会更加方便了。

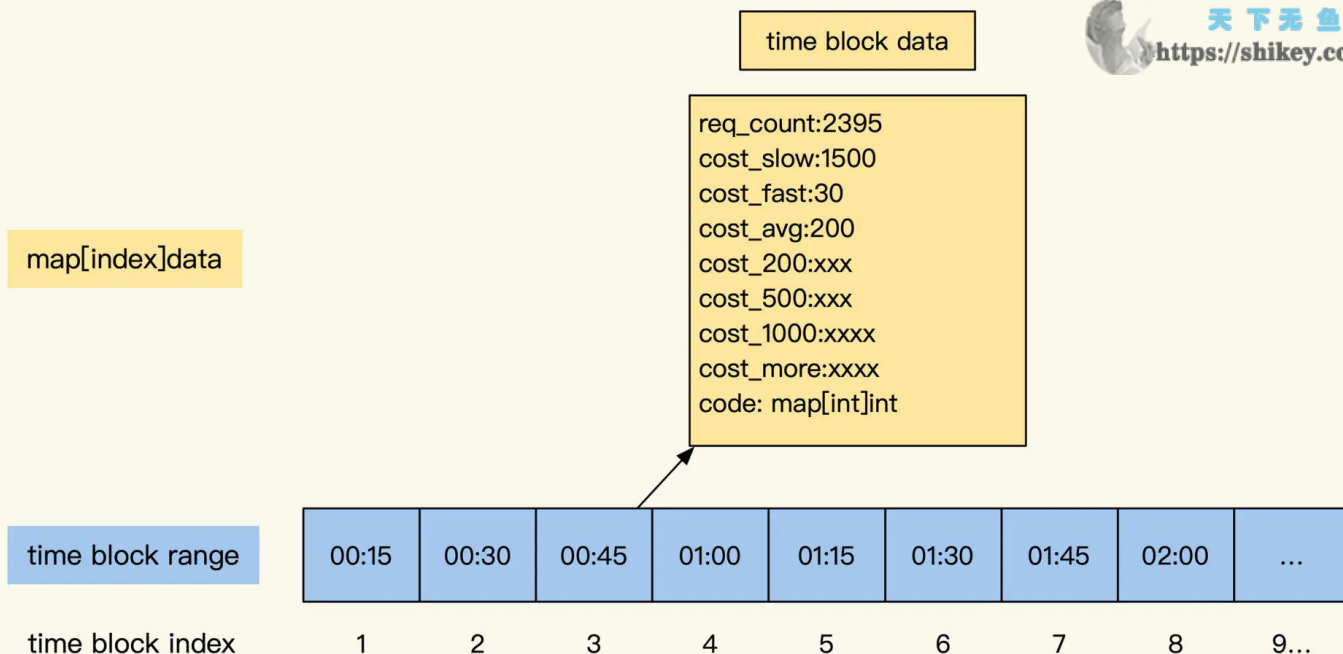
时间分块统计

将 URL 去参数后，我们就可以对不同的接口做性能统计了，这里我用的是时间块方式实现。这么设计，是因为我的日志消费服务可用内存是有限的（只有 8G），而且如果保存太多数据到数据库的话，实时更新效率会很低。

考虑再三，我选择分时间块来保存周期时间块内的统计，将一段时间内的请求数据在内存中汇总统计。

为了更好地展示，我将每天 24 小时，按 15 分钟一个时间块来划分，而每个时间块内都会统计各自时间段内的接口数据，形成数据统计块。

这样，一天就会有 96 个数据统计块（计算公式是： $6400 \text{ 秒} / (15 \text{ 分钟} * 60 \text{ 秒}) = 96$ ）。如果 API 有 200 个，那么我们内存中保存的一天的数据量就是 19200 条（ $96 * 200 = 19200$ ）。



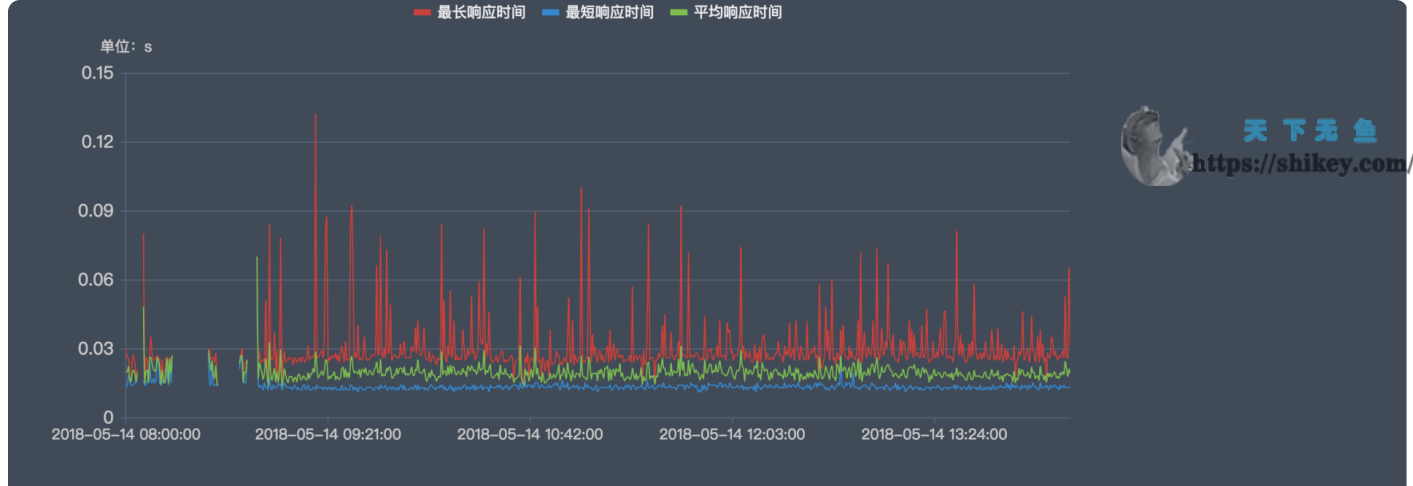
时间块结构

假设我们监控的系统有 200 个接口，就能推算出一年的统计数据量为 700w 条左右。如果有需要，我们可以让这个粒度更小一些。

事实上，市面上很多 metrics 监控的时间块粒度是 3~5 秒一个，直到最近几年出现 OLAP 和时序数据库后，才出现秒级粒度性能统计。而粒度越小监控越细致，粒度过大只能看到时段内的平均性能表现。

我还想说一个题外话，近两年出现了 influxDB 或 Prometheus，用它们来保存数据也可以，但这些方式都需要硬件投入和运维成本，你可以结合自身业务情况来权衡。

我们看一下，在 15 分钟为一段的时间块里，统计了 URL 的哪些内容？



时间	请求次数	最短时间	最长时间	平均时间	5%(s)	25%(s)	50%(s)	75%(s)	95%(s)	操作
8:00 ~ 9:00	558	0.012	0.084	0.021	0.013	0.015	0.02	0.025	0.029	<button>分时统计</button> <button>性能排行</button>

如上图，每个数据统计块内聚合了以下指标：

- 累计请求次数
- 最慢耗时
- 最快耗时
- 平均耗时
- 耗时个数，图中使用的是 ELK 提供的四分位数分析（如果拿不到全量数据来计算四分位数，也可以设置为：小于 200ms、小于 500ms、小于 1000ms、大于 1 秒的请求个数统计）
- 接口响应 http code 及对应的响应个数（如：{"200":1343,"500":23,"404": 12, "301":14}）

把这些指标展示出来，主要是为了分析这个接口的性能表现。看到这里，你是不是有疑问，监控方面我们大费周章去统计这些细节，真的有意义么？

的确，大多数情况下我们 API 的表现都很好，个别的特殊情况才会导致接口响应很慢。不过监控系统除了对大范围故障问题的监控，细微故障的潜在问题也不能忽视。尤其是大吞吐量的服务器，更难发现这种细微的故障。

我们只有在监控上支持对细微问题的排查，才能提前发现这些小概率的故障。这些小概率的故障在极端情况下会导致集群的崩溃。因此提前发现、提前处理，才能保证我们线上系统面对大流量并发时不至于突然崩掉。

错误日志聚类

监控统计请求之后，我们还要关注错误的日志。说到故障排查的难题，还得说说错误日志聚类这个方式。



我们都知道，平时常见的线上故障，往往伴随着大量的错误日志。在海量警告面前，我们一方面要获取最新的错误消息，同时还不能遗漏个别重要但低频率出现的故障。

因为资源有限，内存里无法存放太多的错误日志，所以日志聚类的方案是个不错的选择，通过日志聚合，对错误进行分类，给用户排查即可。这样做，在发现错误的同时，还能够提供错误的范本来加快排查速度。

我是这样实现日志错误聚合功能的：直接对日志做近似度对比计算，并加上一些辅助字段作为修正。这个功能可以把个别参数不同、但同属一类错误的日志聚合到一起，方便我们快速发现的低频故障。

通过这种方式实现的错误监控还有额外的好处，有了它，无需全站统一日志格式标准，就能轻松适应各种格式的日志，这大大方便了对不同系统的监控。

说到这，你是不是挺好奇实现细节的？下面是 github.com/mfonda/simhash 提供的 simhash 文本近似度样例：

 复制代码

```
1 package main
2 import (
3     "fmt"
4     "github.com/mfonda/simhash"
5 )
6 func main() {
7     var docs = [][]byte{
8         []byte("this is a test phrass"), //测试字符串1
9         []byte("this is a test phrass"), //测试字符串2
10        []byte("foo bar"), //测试字符串3
11    }
12    hashes := make([]uint64, len(docs))
13    for i, d := range docs {
14        hashes[i] = simhash.Simhash(simhash.NewWordFeatureSet(d)) //计算出测试字符串
15        fmt.Printf("Simhash of %s: %x\n", d, hashes[i])
16    }
17    //测试字符串1 对比 测试字符串2
18    fmt.Printf("Comparison of 0 1 : %d\n", simhash.Compare(hashes[0], hashes[1])
```



```
19 //测试字符串1 对比 测试字符串3
```

```
20 fmt.Printf("Comparison of 0 2 : %d\n", simhash.Compare(hashes[0], hashes[2])
```

```
21 }
```

```
22
```



天下无鱼

<https://shikey.com/>

看完代码，我再给你讲讲这里的思路。

我们可以用一个常驻进程，持续做 **group consumer** 消费 **Kafka** 日志信息，消费时每当碰到错误日志，就需要通过 **simhash** 将其转换成 **64 位 hash**。然后，通过和已有错误类型的列表进行遍历对比，日志长度相近且海明距离（**simhash.compare** 计算结果）差异不超过 **12 个 bit** 差异，就可以归为一类。

请注意，由于算法的限制，**simhash** 对于小于 **100 字** 的文本误差较大，所以需要我们实际测试下具体的运行情况，对其进行微调。文本特别短时，我们需要一些其他辅助来去重。注意，同时 **100 字** 以下要求匹配度大于 **80%**，**100 字** 以上则要大于 **90%** 匹配度。

最后，除了日志相似度检测以外，**也可以通过生成日志的代码文件名、行数以及文本长度来辅助判断**。由于是模糊匹配，这样能够减少失误。

接下来，我们要把归好类的错误展示出来。

具体步骤是这样的：如果匹配到当前日志属于已有某个错误类型时，就保存错误第一次出现的日志内容，以及错误最后三次出现的日志内容。

我们需要在归类界面查看错误的最近发生时间、次数、开始时间、开始错误日志，同时可以通过 **Trace ID** 直接跳转到 **Trace** 过程渲染页面。（这个做法对排查问题很有帮助，你可以看看我在 [🔗 Java 单机开源版](#) 中的实现，体验下效果。）

事实上，错误去重还有很多的优化空间。比方说我们内存中已经统计出上千种错误类型，那么每次新进的错误日志的 **hash**，就需要和这 **1000 个** 类型挨个做对比，这无形浪费了我们大量的 **CPU** 资源。

对于这种情况，网上有一些简单的小技巧，比如将 **64 位 hash** 分成两段，先对比前半部分，如果近似度高的话再对比后半部分。

这类技巧叫日志聚合，但行业里应用得比较少。

云厂商也提供了类似功能，但是很少应用于错误去重这个领域，相信这里还有潜力可以挖掘，算力充足的情况下行业常用 K-MEANS 或 DBSCAN 算法做日志聚合，有兴趣的小伙伴可以再深挖下。



bitmap 实现频率统计

我们虽然统计出了错误归类，但是这个错误到底发生了多久、线上是否还在持续产生报错？这些问题还是没解决。

若是在平时，我们会将这些日志一个个记录在 OLAP 类的统计分析系统中，按时间分区来汇总聚合这些统计。但是，这个方式需要大量的算力支撑，我们没有那么多资源，还有别的方式来表示么？

这里我用了一个小技巧，就是在错误第一次产生后，每一秒用一个 bit 代表在 bitmap 中记录。

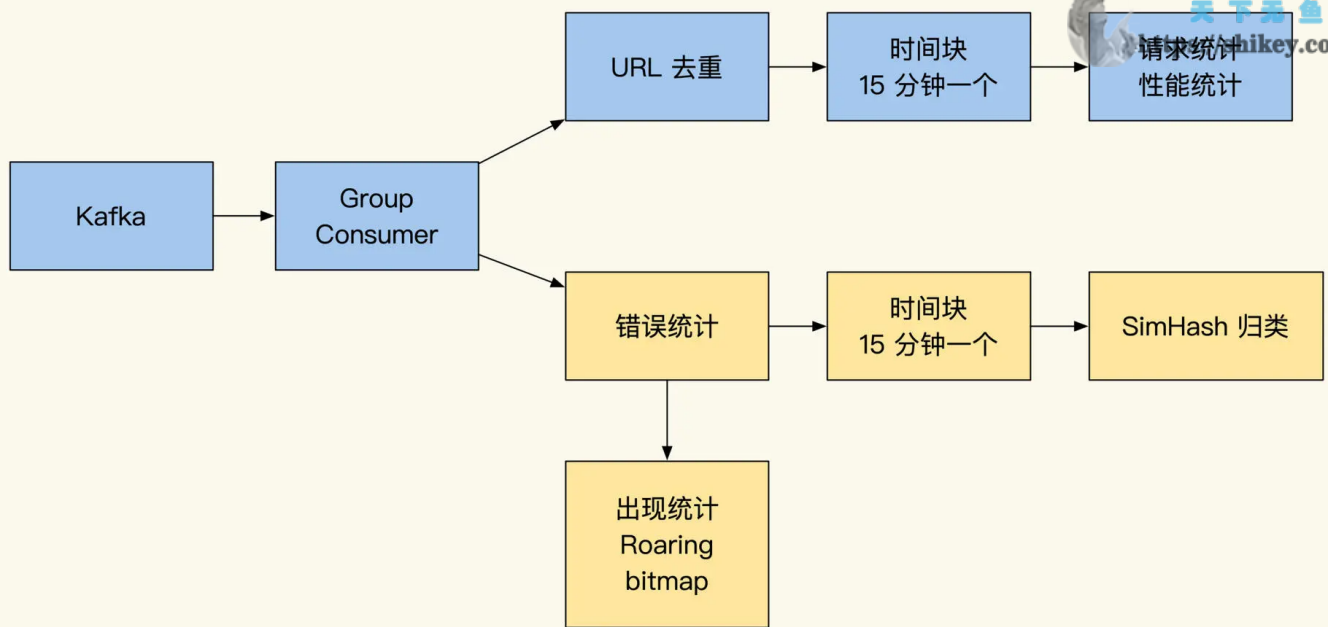
如果这个分钟内产生了同类错误，那么就记录为 1，以此类推，一天会用 86400 个 bit = 1350 个 uint64 来记录日志出现的频率周期。这样排查问题时，就可以根据 bit 反推什么时间段内有错误产生，这样用少量的内存就能快速实现频率周期的记录。

不过这样做又带来了一个新的问题——**内存浪费严重**。这是由于错误统计是按错误归类类型放在内存中的。一个新业务平均每天会有上千种错误，这导致我需要 1350x1000 个 int64 保存在内存中。

为了节省内存的使用，我将 bitmap 实现更换成 **Roraing bitmap**。它可以压缩 bitmap 的空间，对于连续相似的数据压缩效果更明显。事实上 bitmap 的应用不止这些，我们可以用它做很多有趣的标注，相对于传统结构可以节省更多的内存和存储空间。

总结

这节课我给你分享了四种实用的算法，这些都是我实践验证过的。你可以结合后面这张图来复习记忆。



为了解决参数不同给网址聚类造成的难题，可以通过配置或数据特征过滤方式对 URL 进行整理，还可以通过时间块减少统计的结果数据量。

为了梳理大量的错误日志，simhash 算法是一个不错的选择，还可以搭配 bitmap 记录错误日志的出现频率。有了这些算法的帮助，用少量系统资源，即可实现线上服务的故障监控聚合分析功能，将服务的工作状态直观地展示出来。

学完这节课，你有没有觉得，在资源匮乏的情况下，用一些简单的算法，实现之前需要几十台服务器的分布式服务才能实现的服务，是十分有趣的呢？

即使是现代，互联网发展这几年，仍旧有很多场景需要一些特殊的设计来帮助我们降低资源的损耗，比如：用 Bloom Filter 减少扫描次数、通过 Redis 的 hyperLogLog 对大量数据做大致计数、利用 GEO hash 实现地图分块分区统计等。如果你有兴趣，课后可以拓展学习一下 [Redis 模块](#) 的内容。

思考题

基于这节课讲到的算法和思路，SQL 如何做聚合归类去重？

欢迎你在留言区和我交流讨论，我们下节课见！



天下无鱼

<https://shikey.com/>

分享给需要的人，Ta购买本课程，你将得 18 元

生成海报并分享

赞 2

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇

12 | 引擎分片：Elasticsearch如何实现大数据检索？

下一篇

14 | 跳数索引：后起新秀ClickHouse

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。