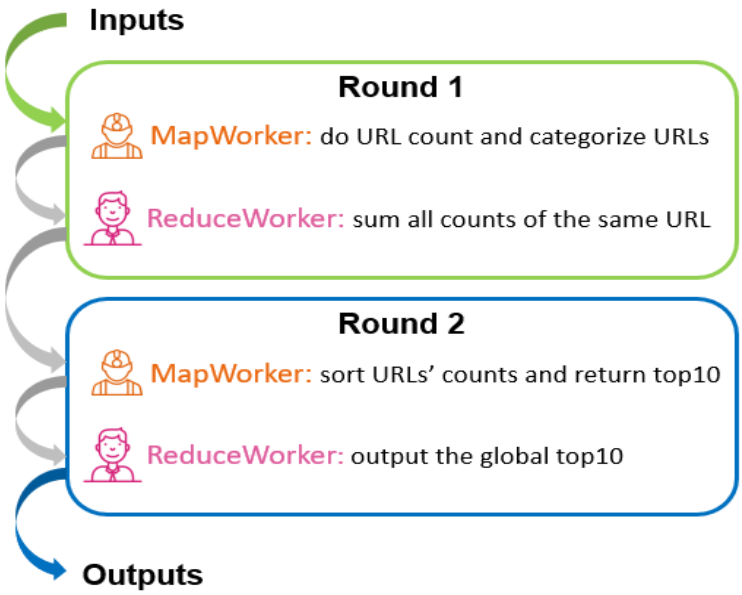
1. Ideas

基本思路如下图所示。



URLTop10的统计分为两轮进行:

- > Round 1: 统计每个URL出现的次数

  MapPhase:

  1. `master`将input data分成**nMap**个dataChunk, 然后形成**nMap**个task分发给worker
  2. 每个`worker`收到task后，分别统计自己负责的dataChunk中每个URL出现的次数，将结果写入KeyValue，其中Key为URL，Value为对应的URL在该dataChunk中出现的次数
  3. 每个`worker`根据Key进行分类，把KeyValue写到相应的文件中(共**nReduce**个文件)

  ReducePhase:

  1. `master`将MapPhase中**nMap**个worker输出的**nMap*nReduce**个文件形成**nReduce**个task分发给worker
  2. 每个`worker`收到task后，将**nMap**个文件中，相同Key下的所有Value拣到一起，相加，得到同一个URL在所有dataChunks中的count
  3. 每个`worker`统计完自己负责的部分URL的Global Count后，将结果写入一个文件中

- > Round 2: 根据每个URL的Count，算出10 most frequent URLs

  MapPhase:

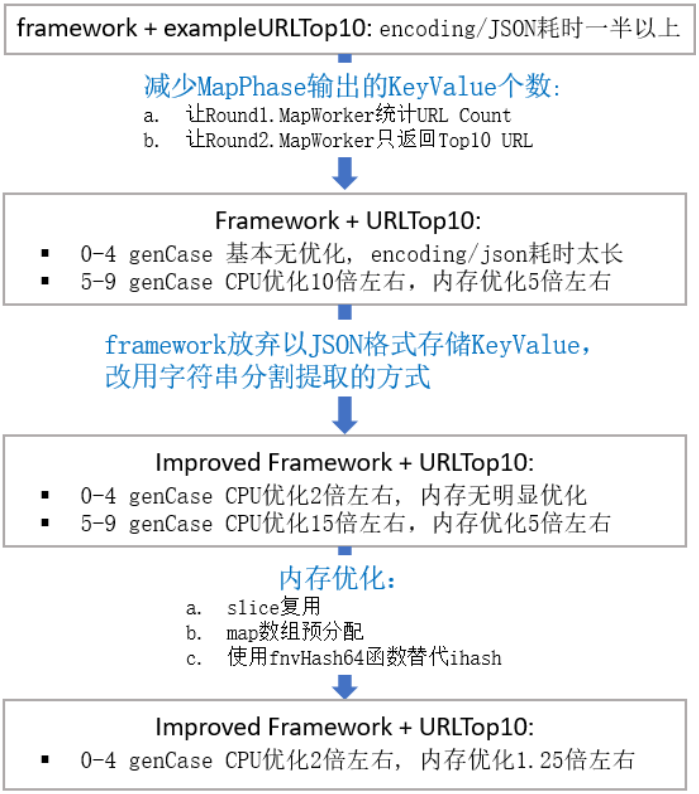
  1. `master`将Round 1 ReducePhase 输出的**nReduce**个文件，形成**nReduce**个task分发给worker
  2. 每个`worker`收到task后，分别对自己负责的dataChunk中的URL Count排序，得到该dataChunk中top10 most frequent URLs，形成含有10个元素的[]KeyValue，其中Key为URL，Value为URL Count
  3. 每个`worker`根据Key进行分类，把KeyValue写到相应的文件中(共**nReduce**个文件, 此时**nReduce**为1)

  ReducePhase:

1. `master`将MapPhase中**nMap**个worker输出的**nMap*nReduce**个文件形成**nReduce**个task分发给worker(此时**nReduce**为1)
2. 每个`worker`收到task后，将**nMap**个文件中所有URL Count排序，得到Global top10 most frequent URLs
3. 每个`worker`将Global top10 most frequent URLs写到一个文件中

2. PProf

整体优化思路流程如下图所示。

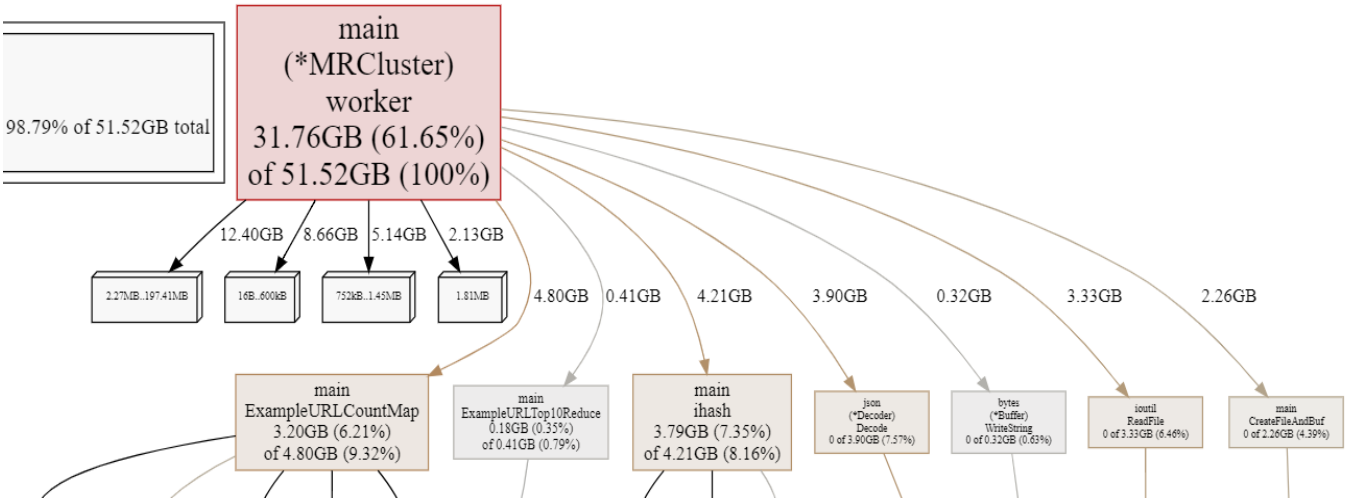


> Phase 1: framework+ExampleURLTop10 -- encoding/JSON耗时太长

完成framework后，用`make test_example`测试了一下，pprof CPU的结果如下图所示。（为了方便，测试了所有的data scale，但genCase只测试了两个，一个随机从0~4个genCase里抽取，另一个随机从5~9个genCase里抽取）

```
Duration: 2.13mins, Total samples = 11.90mins (558.05%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 237.50s, 33.27% of 713.92s total
Dropped 443 nodes (cum <= 3.57s)
Showing top 20 nodes out of 140
      flat  flat%   sum%        cum   cum%
    10.56s  1.48%  1.48%    653.94s 91.60%  main.(*MRCluster).worker
     2.05s  0.29%  1.77%    347.34s 48.65%  encoding/json.(*Decoder).Decode
     1.63s  0.23%  1.99%    234.55s 32.85%  encoding/json.(*decodeState).unmarshal
     3.71s  0.52%  2.51%    227.07s 31.81%  encoding/json.(*decodeState).value
    16.20s  2.27%  4.78%    224.04s 31.38%  encoding/json.(*decodeState).object
     3.92s  0.55%  5.33%    130.96s 18.34%  encoding/json.(*Encoder).Encode
    34.70s  4.86% 10.19%    107.04s 14.99%  encoding/json.(*Decoder).readValue
     1.66s  0.23% 10.43%    106.83s 14.96%  encoding/json.(*encodeState).marshal
     0.34s 0.048% 10.47%    103.72s 14.53%  runtime.systemstack
    34.45s  4.83% 15.30%     88.56s 12.40%  encoding/json.(*decodeState).scanWhile
     2.58s  0.36% 15.66%     88.13s 12.34%  encoding/json.(*encodeState).reflectValue
     5.95s  0.83% 16.49%     76.71s 10.74%  encoding/json.(*decodeState).literalStore
    64.13s  8.98% 25.48%        70s  9.81%  encoding/json.stateInString
     2.83s   0.4% 25.87%     68.08s  9.54%  encoding/json.ptrEncoder.encode
    10.88s  1.52% 27.40%     63.12s  8.84%  encoding/json.structEncoder.encode
    14.73s  2.06% 29.46%     59.50s  8.33%  runtime.mallocgc
    26.65s  3.73% 33.19%     57.51s  8.06%  runtime.scanobject
         0     0% 33.19%     53.60s  7.51%  runtime.gcBgMarkWorker
         0     0% 33.19%     53.59s  7.51%  runtime.gcBgMarkWorker.func2
     0.53s 0.074% 33.27%     53.59s  7.51%  runtime.gcDrain
```

可以明显的看出encoding/JSON花的时间实在是太多了，虽然runtime.systemstack花的时间也不少.....然后看了一下内存的分配，如下图所示。内存的分配也非常不合理。先尝试优化encoding/JSON耗时太长的问题。



> Phase 2: framework+URLTop10 -- 对URL分布离散的0~4 genCase，优化基本无效; 5~9 genCase CPU优化近10倍，内存优化近5倍

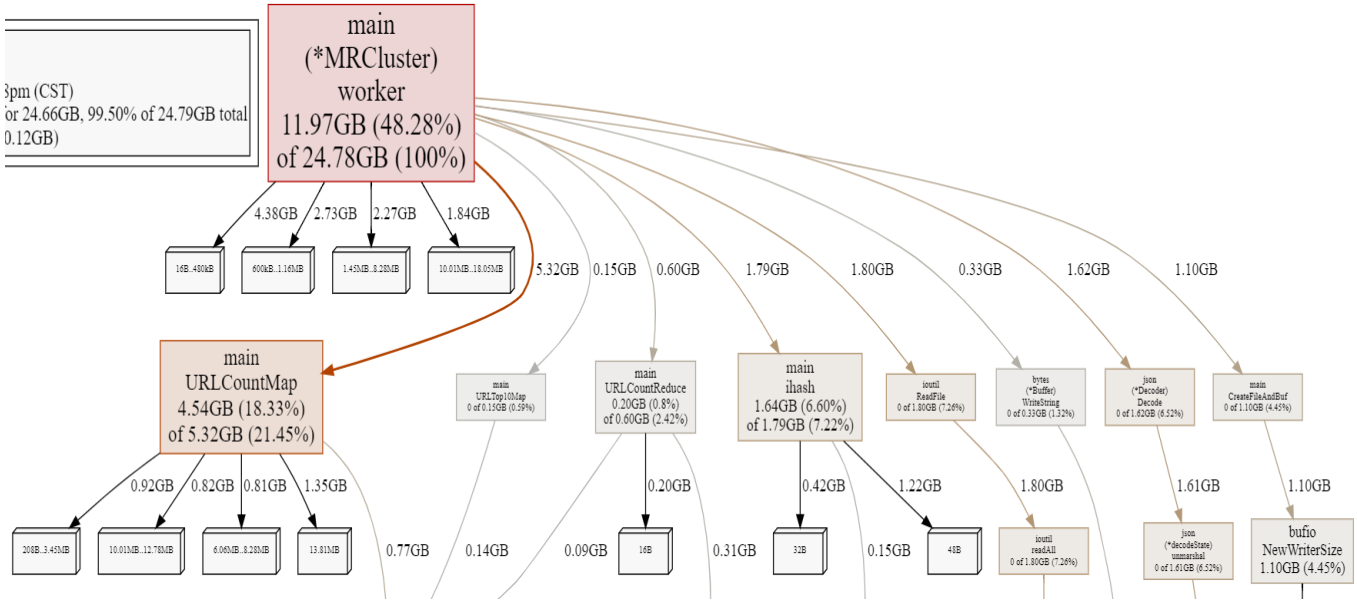
采用Idea所述的mapF和reduceF后，随机从0~4和5~9 genCase中各随机抽取一个case对所有data scale进行pprof分析，得到结果如下。

首先是`0~4 genCase`的分析。由于0~4 genCase产生的URL分布比较离散，而URLTop10采取的优化主要是在Round1.MapPhase统计URL Count，合并相同的URL从而减小输出JSON文件大小和在Round2.MapPhase对URL Count进行排序，输出局部的Top10 URL上。极端地，如果每个URL只出现一次，那优化就完全没用了，甚至还会因为Round1.MapPhase在统计每个URL Count时的map操作而变慢。(如果先分拣URL，让reduce worker去做count, map操作会更快)。

pprof CPU的输出如下图所示。相比于ExampleURLTop10基本无优化。

```
Duration: 53.88s, Total samples = 6.75mins (752.20%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 142.40s, 35.14% of 405.26s total
Dropped 399 nodes (cum <= 2.03s)
Showing top 20 nodes out of 137
      flat  flat%   sum%        cum   cum%
     4.45s  1.10%  1.10%    363.93s 89.80%  main.(*MRCluster).worker
     1.05s  0.26%  1.36%    180.89s 44.64%  encoding/json.(*Decoder).Decode
     0.67s  0.17%  1.52%    121.59s 30.00%  encoding/json.(*decodeState).unmarshal
     1.84s  0.45%  1.98%    117.42s 28.97%  encoding/json.(*decodeState).value
     7.89s  1.95%  3.92%    115.98s 28.62%  encoding/json.(*decodeState).object
     0.27s 0.067%  3.99%     74.68s 18.43%  runtime.systemstack
    16.21s  4.00%  7.99%     55.74s 13.75%  encoding/json.(*Decoder).readValue
    17.40s  4.29% 12.28%     48.54s 11.98%  encoding/json.(*decodeState).scanWhile
    17.48s  4.31% 16.60%     48.01s 11.85%  runtime.mapassign_faststr
     1.02s  0.25% 16.85%     42.97s 10.60%  encoding/json.(*Encoder).Encode
     1.47s  0.36% 17.21%     42.28s 10.43%  main.URLCountMap
    15.61s  3.85% 21.06%     40.38s  9.96%  runtime.scanobject
    35.10s  8.66% 29.72%     39.53s  9.75%  encoding/json.stateInString
     2.85s   0.7% 30.43%     38.50s  9.50%  encoding/json.(*decodeState).literalStore
         0     0% 30.43%     36.66s  9.05%  runtime.gcBgMarkWorker
         0     0% 30.43%     36.65s  9.04%  runtime.gcBgMarkWorker.func2
     0.48s  0.12% 30.55%     36.65s  9.04%  runtime.gcDrain
     7.73s  1.91% 32.45%     35.94s  8.87%  runtime.mallocgc
     0.51s  0.13% 32.58%     35.50s  8.76%  encoding/json.(*encodeState).marshal
    10.37s  2.56% 35.14%     31.55s  7.79%  runtime.gcWriteBarrier
```

pprof mem的输出结果如下图所示。相比于ExampleURLTop10基本无优化。

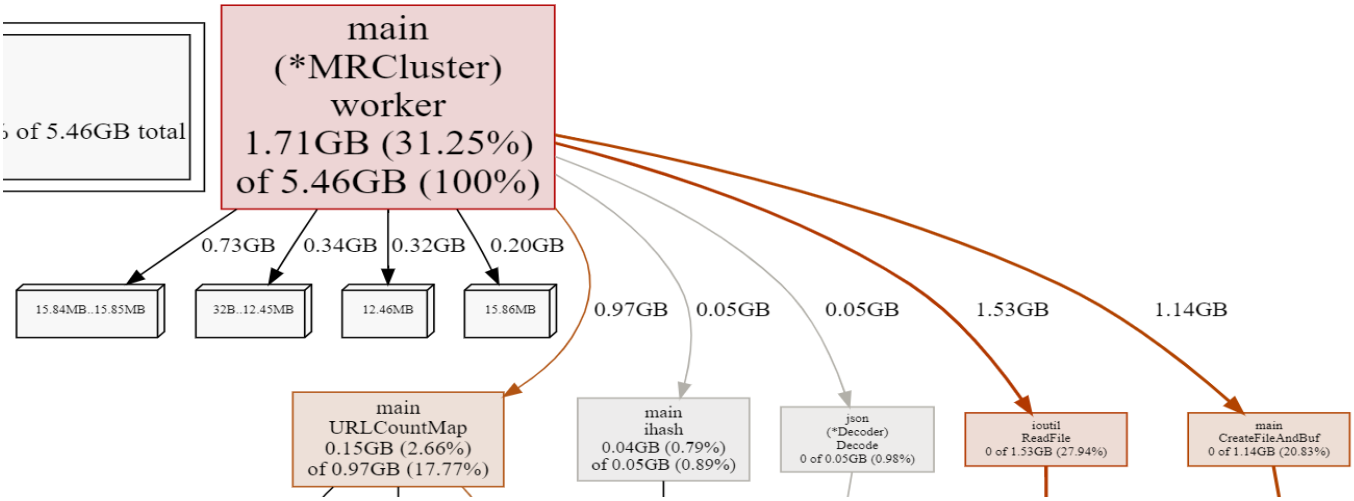


然后是5~9 genCase的分析。由于5~9 genCase的URL分布是biased的，MapPhase所做的优化就比较明显了。

pprof CPU的输出如下图所示。相比于ExampleURLTop10优化了将近10倍。由于我在windows上测的结果，所以runtime.cgocall耗时就比较明显了。

```
Duration: 4.96s, Total samples = 31.51s (635.70%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 15.10s, 47.92% of 31.51s total
Dropped 256 nodes (cum <= 0.16s)
Showing top 20 nodes out of 118
      flat  flat%   sum%        cum   cum%
     0.13s  0.41%  0.41%     29.60s 93.94%  main.(*MRCluster).worker
    12.08s 38.34% 38.75%     12.09s 38.37%  runtime.cgocall
     0.44s  1.40% 40.15%      9.84s 31.23%  main.URLCountMap
         0     0% 40.15%      7.92s 25.13%  syscall.Syscall
         0     0% 40.15%      7.91s 25.10%  main.SafeClose
         0     0% 40.15%      7.57s 24.02%  internal/poll.(*FD).Close
         0     0% 40.15%      7.57s 24.02%  internal/poll.(*FD).decref
         0     0% 40.15%      7.57s 24.02%  internal/poll.(*FD).destroy
         0     0% 40.15%      7.57s 24.02%  os.(*File).Close
         0     0% 40.15%      7.57s 24.02%  os.(*file).close
         0     0% 40.15%      7.57s 24.02%  syscall.CloseHandle
     0.07s  0.22% 40.37%      4.46s 14.15%  encoding/json.(*Decoder).Decode
     1.61s  5.11% 45.48%      3.80s 12.06%  runtime.mapassign_faststr
     0.01s 0.032% 45.51%      3.47s 11.01%  runtime.systemstack
         0     0% 45.51%      3.16s 10.03%  io/ioutil.ReadFile
         0     0% 45.51%      3.10s  9.84%  strings.Split
     0.45s  1.43% 46.94%      3.10s  9.84%  strings.genSplit
     0.05s  0.16% 47.10%      3.05s  9.68%  encoding/json.(*decodeState).unmarshal
     0.09s  0.29% 47.38%      2.98s  9.46%  encoding/json.(*decodeState).value
     0.17s  0.54% 47.92%      2.93s  9.30%  encoding/json.(*decodeState).object
```

pprof mem的输出结果如下图所示。相比于ExampleURLTop10内存优化了将近5倍。



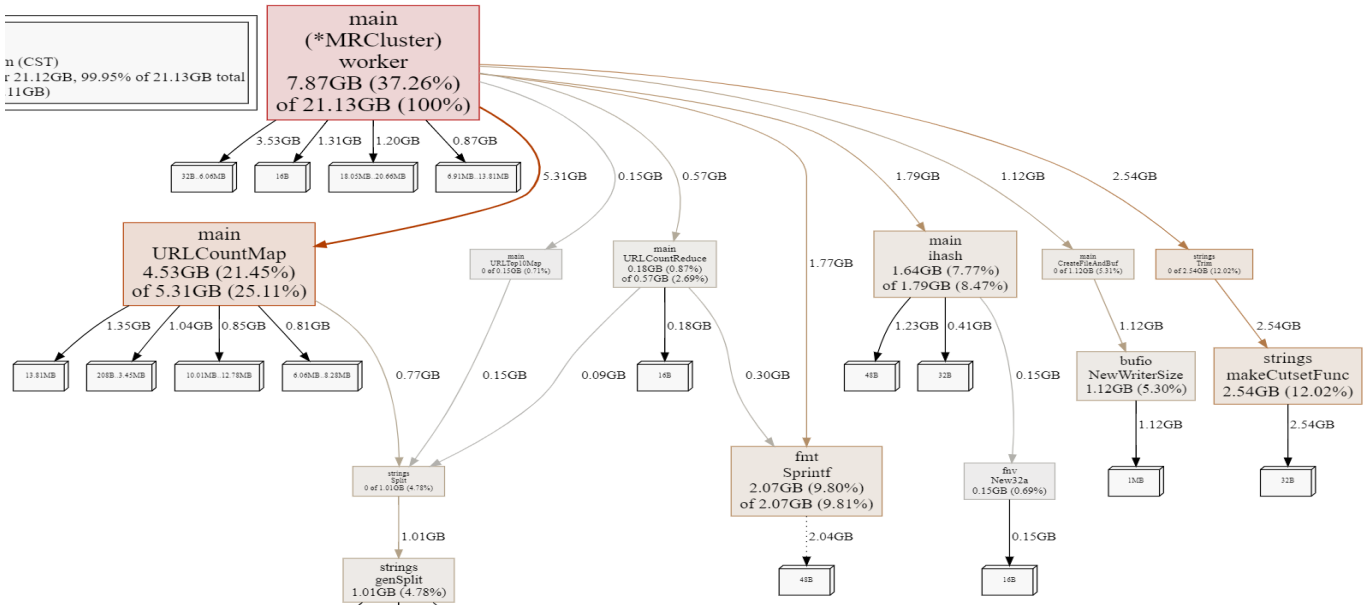
> Phase 3: Improved framework + URLTop10 -- 0~4 genCase CPU优化近2倍，内存无明显优化; 5~9 genCase CPU优化将近15倍，内存优化近5倍

此步重点对0~4 genCase进行优化，由前面可知，50%左右的时间都花在了encoding/json上，而需要json编码的只是很简单的一个KeyValue结构体，其实没必要用json编码的。于是此步取消了encoding/json，而采用字符串的分割形式来对MapPhase输出的结果进行读写。0~4 genCase CPU和内存开销比5~9 genCase高太多，所以接下来重点关注0~4 genCase的优化。

0~4 genCase中随机抽取一个gen函数对所有data scale进行测试，pprof CPU的结果如下图所示。可以看到map操作和内存分配占了绝大部分时间。

```
Duration: 24.34s, Total samples = 3.01mins (740.74%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 70.59s, 39.14% of 180.33s total
Dropped 224 nodes (cum <= 0.90s)
Showing top 20 nodes out of 135
      flat  flat%   sum%        cum   cum%
    10.96s  6.08%  6.08%    165.60s 91.83%  main.(*MRCluster).worker
    16.28s  9.03% 15.11%     43.33s 24.03%  runtime.mapassign_faststr
    11.76s  6.52% 21.63%     42.24s 23.42%  runtime.mallocgc
     1.54s  0.85% 22.48%     34.43s 19.09%  main.URLCountMap
     0.06s 0.033% 22.51%     27.86s 15.45%  runtime.systemstack
     0.93s  0.52% 23.03%     22.90s 12.70%  fmt.Sprintf
     0.70s  0.39% 23.42%     21.29s 11.81%  strings.Trim
     5.01s  2.78% 26.20%     15.63s  8.67%  runtime.scanobject
     0.08s 0.044% 26.24%     13.22s  7.33%  runtime.growWork_faststr
     2.87s  1.59% 27.83%     13.12s  7.28%  runtime.evacuate_faststr
    12.81s  7.10% 34.94%     12.82s  7.11%  runtime.cgocall
     0.91s   0.5% 35.44%     12.51s  6.94%  runtime.newobject
     1.04s  0.58% 36.02%     12.44s  6.90%  strings.TrimFunc
     1.61s  0.89% 36.91%     12.13s  6.73%  runtime.slicebytetostring
     2.33s  1.29% 38.20%     11.81s  6.55%  fmt.(*pp).doPrintf
     0.95s  0.53% 38.73%     11.34s  6.29%  strings.makeCutsetFunc
     0.69s  0.38% 39.11%     10.96s  6.08%  main.ihash
         0     0% 39.11%     10.78s  5.98%  runtime.gcBgMarkWorker
         0     0% 39.11%     10.77s  5.97%  runtime.gcBgMarkWorker.func2
     0.06s 0.033% 39.14%     10.77s  5.97%  runtime.gcDrain
```

相应的pprof mem的结果如下图所示。内存只有优化了四五个G左右。其中`URLCountMap`和`fmt.Sprintf`还有`ihash`的耗内存情况比较突出。



单独看一下URLCountMap的内存和CPU情况。可以看出用map做URL Count操作是非常昂贵的。

```
4.53GB     5.31GB (flat, cum) 25.11% of Total
     .          .     36:}
     .          .     37:
     .          .     38:// URLCountMap is the map function in the first round
     .          .     39:// URLCountMap split content into components and do url count ( partial count)
     .          .     40:func URLCountMap(filename string, contents string) []KeyValue {
     .   791.72MB     41:   lines := strings.Split(string(contents), "\n")
     .          .     42:   mapCounter := make(map[string]int)
     .          .     43:
     .          .     44:   for _, l := range lines {
     .          .     45:           l = strings.TrimSpace(l)
     .          .     46:           if len(l) == 0 {
     .          .     47:                   continue
     .          .     48:           }
3.29GB     3.29GB     49:           mapCounter[l] += 1
     .          .     50:   }
     .          .     51:
1.24GB     1.24GB     52:   kvs := make([]KeyValue, 0, len(mapCounter))
     .          .     53:   for k, v := range mapCounter {
     .          .     54:           kvs = append(kvs, KeyValue{k, strconv.Itoa(v)})
     .          .     55:   }
     .          .     56:
     .          .     57:   return kvs
```

```
    .          .     36:}
    .          .     37:
    .          .     38:// URLCountMap is the map function in the first round
    .          .     39:// URLCountMap split content into components and do url count ( partial count)
    .          .     40:func URLCountMap(filename string, contents string) []KeyValue {
    .      2.32s     41:   lines := strings.Split(string(contents), "\n")
    .          .     42:   mapCounter := make(map[string]int)
    .          .     43:
  120ms      120ms     44:   for _, l := range lines {
   30ms     2.86s     45:           l = strings.TrimSpace(l)
   10ms       10ms     46:           if len(l) == 0 {
    .          .     47:                   continue
    .          .     48:           }
 1.03s     25.63s     49:           mapCounter[l] += 1
    .          .     50:   }
    .          .     51:
    .      490ms     52:   kvs := make([]KeyValue, 0, len(mapCounter))
   80ms        2s     53:   for k, v := range mapCounter {
  270ms        1s     54:           kvs = append(kvs, KeyValue{k, strconv.Itoa(v)})
    .          .     55:   }
    .          .     56:
    .          .     57:   return kvs
    .          .     58:}
    .          .     59:
```

> Phase 4: 内存预分配和底层分配空间复用 -- 内存优化了1.25倍左右

优化的点包括：

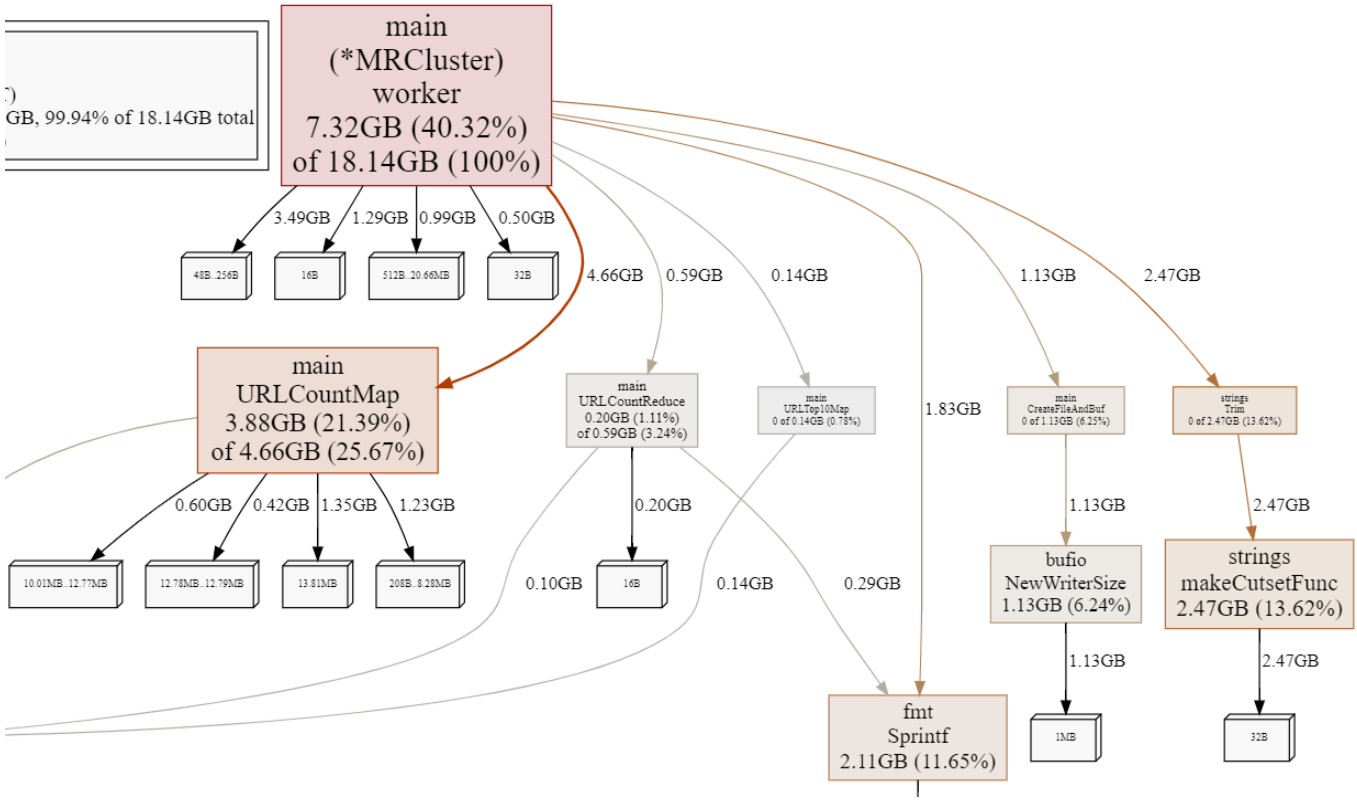
1. `bufio.Read(content)`[]byte类型的content复用
2. `map预分配空间`预分配空间大小为len(lines)/4
3. `ihash函数换成了fnvHash64`

TODO: 从CPU和mem的结果来看，最应该优化的是fmt.Sprintf，字符串拼接函数，考虑过使用strings.Join、bytes.Buffer等替代方案，但最后会导致bufio.WriteString()写到文件的内容为空。

0~4 genCase随机抽取一个case对所有dataScale进行测试。pprof CPU的结果如下图所示。

```
Duration: 24.34s, Total samples = 3.01mins (740.74%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 70.59s, 39.14% of 180.33s total
Dropped 224 nodes (cum <= 0.90s)
Showing top 20 nodes out of 135
      flat  flat%   sum%        cum   cum%
    10.96s  6.08%  6.08%    165.60s 91.83%  main.(*MRCluster).worker
    16.28s  9.03% 15.11%     43.33s 24.03%  runtime.mapassign_faststr
    11.76s  6.52% 21.63%     42.24s 23.42%  runtime.mallocgc
     1.54s  0.85% 22.48%     34.43s 19.09%  main.URLCountMap
     0.06s 0.033% 22.51%     27.86s 15.45%  runtime.systemstack
     0.93s  0.52% 23.03%     22.90s 12.70%  fmt.Sprintf
     0.70s  0.39% 23.42%     21.29s 11.81%  strings.Trim
     5.01s  2.78% 26.20%     15.63s  8.67%  runtime.scanobject
     0.08s 0.044% 26.24%     13.22s  7.33%  runtime.growWork_faststr
     2.87s  1.59% 27.83%     13.12s  7.28%  runtime.evacuate_faststr
    12.81s  7.10% 34.94%     12.82s  7.11%  runtime.cgocall
     0.91s   0.5% 35.44%     12.51s  6.94%  runtime.newobject
     1.04s  0.58% 36.02%     12.44s  6.90%  strings.TrimFunc
     1.61s  0.89% 36.91%     12.13s  6.73%  runtime.slicebytetostring
     2.33s  1.29% 38.20%     11.81s  6.55%  fmt.(*pp).doPrintf
     0.95s  0.53% 38.73%     11.34s  6.29%  strings.makeCutsetFunc
     0.69s  0.38% 39.11%     10.96s  6.08%  main.ihash
         0     0% 39.11%     10.78s  5.98%  runtime.gcBgMarkWorker
         0     0% 39.11%     10.77s  5.97%  runtime.gcBgMarkWorker.func2
     0.06s 0.033% 39.14%     10.77s  5.97%  runtime.gcDrain
```

相应的pprof mem的结果如下图所示。


GB, 99.94% of 18.14GB total
main
(*MRCluster)
worker
7.32GB (40.32%)
of 18.14GB (100%)
3.49GB
1.29GB
0.99GB
0.50GB
48B..256B
16B
512B..20.66MB
32B
4.66GB
0.59GB
0.14GB
1.13GB
2.47GB
main
URLCountMap
3.88GB (21.39%)
of 4.66GB (25.67%)
main
URLCountReduce
0.20GB (1.11%)
of 0.59GB (3.24%)
main
URLTop10Map
0 of 0.14GB (0.78%)
1.83GB
main
CreateFileAndBuf
0 of 1.13GB (6.25%)
strings
Trim
0 of 2.47GB (13.62%)
0.60GB
0.42GB
1.35GB
1.23GB
10.01MB..12.77MB
12.78MB..12.79MB
13.81MB
208B..8.28MB
0.20GB
0.10GB
16B
0.14GB
0.29GB
1.13GB
2.47GB
bufio
NewWriterSize
1.13GB (6.24%)
strings
makeCutsetFunc
2.47GB (13.62%)
1.13GB
2.47GB
fmt
Sprintf
2.11GB (11.65%)
1MB
32B