**Probe Scheduler System Design**

* **需求分析**

类似0-1背包问题。输入：系统每天可用的最大资源为T，每个探针Pi每天消耗的负载/资源为Fi，贡献值为Vi。输出：找出一个最优的探针组合P[x...y]，使得贡献值sum(Vi)最大，且sum（Fi） <= T。

本问题不就要找到最大的贡献值，更要找到贡献值最大的探针组合。

（假设）探针Pi最多有上万个，系统资源T有上千个。从任务调度器的角度看，探针和系统资源可以动态增减，但本次重点考虑找出贡献值最大的路径的算法，所以暂不考虑。

Inputs:

Maximum system load: 13

|  |  |  |
| --- | --- | --- |
| **P** | **F** | **V** |
| p1 | 3 | 8 |
| p2 | 4 | 7 |
| p3 | 5 | 9 |
| p4 | 7 | 5 |
| p5 | 6 | 6 |
| p6 | 5 | 6 |

Outputs:

Probes to be executed: p1, p2, p3

Total execution times: 12

Total expected alarm contribution value: 24

* **系统设计**
* **算法设计**
* **贪心算法？**

在本应用场景中，一个探针的资源贡献与负载消耗接近正比关系，即消耗越多贡献值也越大，因此尽可能耗尽所有的资源也就意味着贡献值接近最大，所选出的路径接近最佳路径，但如何尽可能耗尽资源？按照负载倒序排列，认为前N个即为性价比较高方案，sum(N)<=T。也可以比较前几个组合以找到更有组合，类似通过优化比较基准数、有序片段来优化快速排序。

* 优点：算法简单，速度快，可以找出接近最优的路径
* 缺点：可能不是最佳路径
* 复杂度：主要取决于排序算法，可以使用红黑树边插入边按照负载排序，则：
* 时间复杂度O（m），m为最优方案的路径长度，远小于n；空间复杂度O（n），n为探针数
* **动态规划算法？**

思路类似递归，把一个大问题划分成若干个相似的小问题去解决。如果要计算第i个探针Pi后的最大贡献值Vi，只需知道前一个探针P(i-1)的最大贡献值V(i-1)、V(剩余资源数)，即可以算出Vi。可以借助最大贡献值数组dpTable存储递归过程数据，大大减少递归/循环次数。

关键算法1：状态转移方程Vi，对于每个探针Pi，要么分配，或不分配

1. 如果不分配Pi，例如资源不足，则Vi=V(i-1)，即前一次的最大贡献值；
2. 如果加入Pi，则Vi = max(V(i-1), Pi的贡献值 + V(剩余资源数))

关键算法2：推算最优探针列表，根据（Pi，Fi）

* 如果dpTable[i-1][Fi] == dpTable[i][Fi]，则P(i)不在路径上，继续求（P(i-1)，Fi）
* 如果dpTable[i-1][Fi] != dpTable[i][Fi]，则P(i)在路径上，继续求（P(i-1)，Fi）

主要数据结构：

* 探针负载情况 = ProbeLoad = {探针名，负载值，贡献值}
* 探针负载列表 = vector<vector<ProbeLoad>>，也可以用map直接不允许重名
* 最大贡献值表 = dpTable =[探针数][负载数]，dpTable[Pi][Fi] = 每个探针在每个负载切片下的最大贡献值，初始值为0

算法分析

* 优点：可以准确找到最大贡献值，及其路径
* 缺点：算法相对复杂
* 复杂度：时间复杂度O（P\*T），空间复杂度O（P\*T）

算法优化：

dpTable可以优化成滚动的2行，也可以直接用一位数组存储资源切片列表，因为V(i)只与V(i-1)有关。

* **OO程序设计**

1. 算法可能会有多个，所以需要封装变化，可以使用策略模式
2. 问题域中有调度器的概念，且探针任务P貌似比较耗时，可以参考Yarn或Mesos，不过本项目不考虑。
3. 类图如下，详细见 doc/ProbeScheduler.oom 用Power Designer软件打开



* **数据模型设计**

使用CSV格式存储探针负载数据，每行{name，load，value}，资源容量由用户程序指定，计算结果输出到屏幕。

* **目录结构**
* **源码目录结构**

/

doc/ 文档目录

src/ 源码目录

scheduler/ 探针调度器模块

test/ 测试程序，测试数据目录可以由用户指定

util/ 工具模块

安装目录结构

/

include/ 头文件

scheduler/

util/

bin/ 可执行文件，库文件

lib/ 库文件

test/ 测试目录，主要存放默认测试数据

* **编译测试**

how to complie when we clone source code?

1. config Linux env

export NET\_BRAIN\_BUILD=/home/ec2-user/NetBrain/build

export JMAKE\_PATH=/home/ec2-user/NetBrain

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:NET\_BRAIN\_ROOT/lib:.

2.compile and install

cd NetBrain/src

make

make install

3. run test cases

cd $NET\_BRAIN\_BUILD/test

./testAll -h

* **测试用例**
* 边界值：
* 探针负载为0，即P=0
* 系统资源为0，即T=0
* 探针负载<0
* 系统资源<0
* 探针负载>10万
* 系统资源>10万
* 读取探针负载文件错误，可以交由CVS文件读取工具模块测试
* 场景测试
* 【正常-小规模】初始化探针负载数据，P=3，T=5
* 【扩展-动态扩容】P=（3 -->30），T=（5-->50），暂不测试
* 【异常-动态缩容】T=（30-->10），P=（50-->10），暂不测试
* 测试数据
* 探针数据存在CSV文件中，小批量测试数据由人工写入，性能测试数据由程序随机生成，可以限定探针个数，以及每个探针的负载和贡献值取值范围。
* 对于不同算法，要确保数据相同。所以，当性能测试数据文件不存在时，程序自动生成，否则直接加载。
* **压力测试**
* 测试环境
* AWS Linux 2, CPU=2.4GHz, 内存=1GB，gcc version 7.3.1
* Win10，CPU=1.6\*2GHz，内存=8G，VC++2019
* 思路：探针数与资源数成正比关系，首先测试10:1，成比例增加，比较不同的算法。假设探针数为P，资源数为T，动态规划二位数组版为ValueMax，贪心算法为Greedy
* P=10K，T=1K，Greedy
* P=100K，T=10K，Greedy
* P=1000K，T=100K，Greedy
* P=10K，T=1K，ValueMax
* P=100K，T=10K，ValueMax
* P=1000K，T=100K，ValueMax
* **跨平台测试**
* AWS Linux 2
* Win 10
* **测试报告**
* 报告总结：
* Greedy算法，非常接近最优探针组合，所选探针负载 / 资源容量 > 90%，时间复杂度接近 O(1)，空间复杂度 O(探针数)，因为探针列表在始终维持倒序排列。而探针的初始化、增加和减少都不频繁。
* ValueMax，准确找出贡献值最大的探针组合，时间复杂度 O(P\*T) = O(探针数\*资源数)，空间复杂度 =