1. 代码架构

快手 | K-Code

主模块

class KcodeAlertAnalysisImpl

- Collection<String> alarmMonitor(String path, Collection<String> alertRules)
  返回所有的触发的报警的数据
- Collection<String> getLongestPath(String caller, String responder, String time, String type)
  返回报警点影响的最长调用链路

读取与解析模块

class ReadParse

- Map<String, Map<String, Map<Long, SortedMap<Long, MyArray>>>> compute()
  返回解析数据文件后得到的存储表

class MyArray 存放调用结果的数据结构

- void add(boolean isSuccess, int timeCost);
  向数据结构中添加一次调用的结果
- double getSuccessRate()
  返回调用成功率
- int getP99()
  返回耗时的P99值

有向图模块

class Digraph<T> 使用邻接表实现的有向图

- void addEdge(T v, T w);
  向由向图中添加一条 v->w 的边
- List<List<T>> getLongestPaths(T v)
  返回有向图中以v为起点的所有最长路径

警报规则模块

class Rule

- static Rule compile(String line)
  返回根据警报规则字符串构造的Rule对象
- Collection<Result> apply(
  Map<String,Map<String,Map<Long,SortedMap<Long, MyArray>>>> callerMap)
  将规则应用于存储表，得到触发警报格则的结果

class RuleApplyTask extends RecursiveTask<Collection<Rule.Result>>

- Collection<Rule.Result> compute()
  多线程计算所有警报规则触发得到的结果

2. 核心思路

快手 | K-Code

2.1 监控数据报警点检查

数据解析 → 数据存储 → 报警点检查

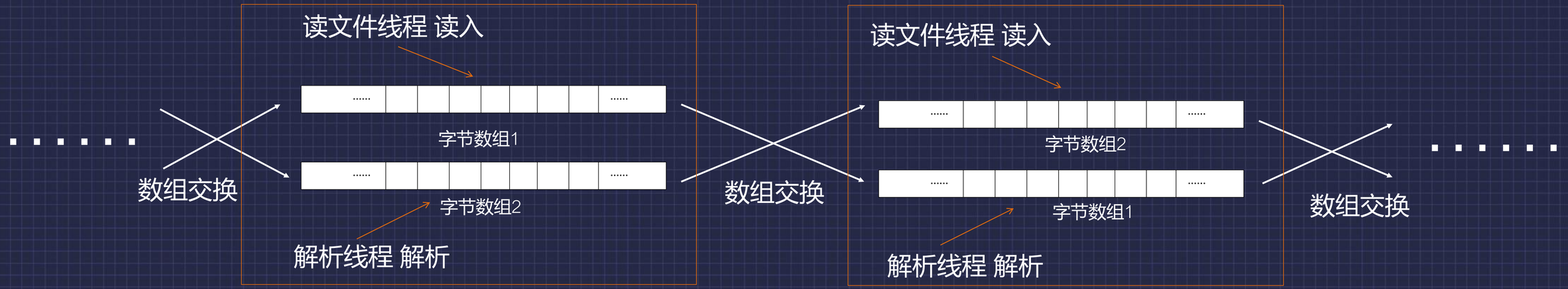
- **数据解析**

```
主调服务名,主调方IP,被调服务名,被调方IP,结果,耗时(ms), 调用时间戳(ms)
serviceA,172.17.60.2,serviceB,172.17.60.3,true,20,1592454780000「2020-06-18 12:33」
serviceA,172.17.60.2,serviceB,172.17.60.3,true,30,1592454780000「2020-06-18 12:33」
serviceA,172.17.60.2,serviceB,172.17.60.3,true,50,1592454900000「2020-06-18 12:34」

serviceZ,172.17.60.8,serviceA,172.17.60.2,true,420,1592454780000「2020-06-18 12:33」
serviceZ,172.17.60.8,serviceC,172.17.60.4,true,120,1592454780000「2020-06-18 12:33」
```

监控数据片段示例

双线程 边读边解析



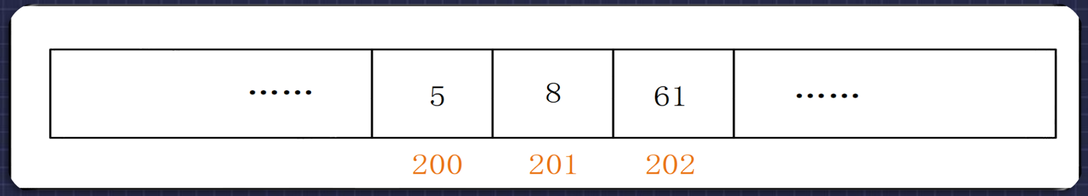
- **数据存储**

数据存储表：使用嵌套的Map，key依次为caller，responder，ip地址对，分钟，Value为存放耗时与成功次数的数据结构

```
private Map<String, Map<String, Map<Long, SortedMap<Long, MyArray>>>> callerMap = new HashMap<>();
```

存放耗时与成功次数的数据结构MyArray：

1. 使用计数数组来存储调用耗时（自动 动态扩容）
2. 由后向前遍历计数数组可快速得到P99
3. 使用两个变量来存储成功次数和失败次数



计数数组示例

- **报警点检查**

报警规则举例：2，ALL，serviceC，P99，2>，300ms

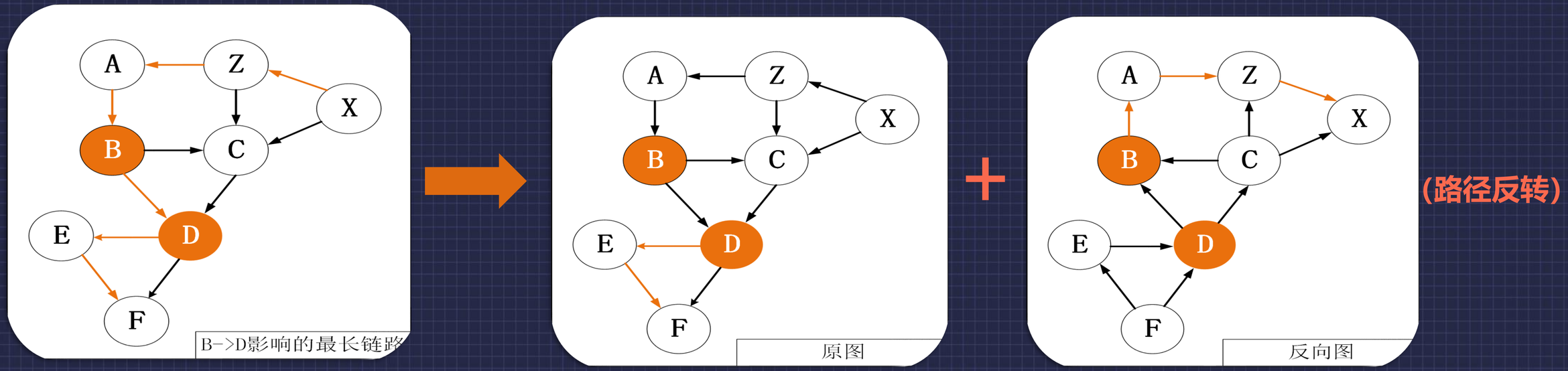
调用关系举例：serviceB,172.17.60.3,serviceC,172.17.60.4

输出结果：

2,2020-06-18 12:34,serviceB,172.17.60.3,serviceC,172.17.60.4,380ms
2,2020-06-18 12:35,serviceB,172.17.60.3,serviceC,172.17.60.4,404ms
2,2020-06-18 12:38,serviceB,172.17.60.3,serviceC,172.17.60.4,330ms

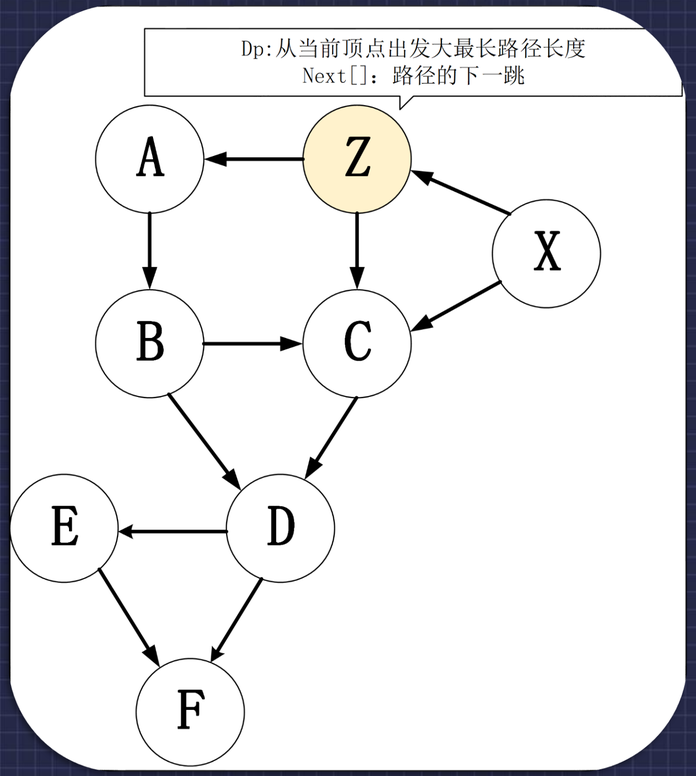
2.2 报警点最长调用链路求解算法

- 转化为求固定起点的最长路径



- 固定起点的最长路径算法（DFS+动态规划+回溯法）

- **固定起点的最长路径算法（DFS+动态规划+回溯法）**



- $dp[i]$：图中从顶点 i出发能构造的最长路径的长度
- $dp[i] = \max(dp[j]) + 1$（j为i的直接指向的顶点）
- 对于出度为0的顶点x，$dp[x]=1$

时间复杂度： O(E+V)　　E为有向图的边数，V为顶点数

空间复杂度： O(E+V)

构造路径：沿着路径查询顶点的next[]来得到下一跳顶点，利用回溯法构造出所有最长路径

3. 优化过程

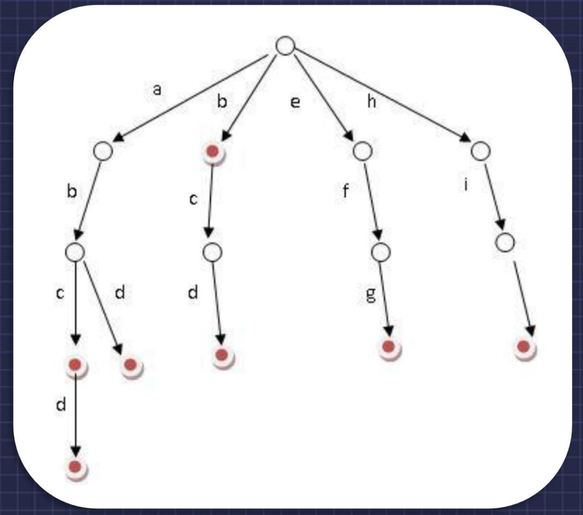
快手 | K-Code

3. 1 读取解析的优化

使用 前缀树 Trie 来减轻GC

```
主调服务名,主调方IP,被调服务名,被调方IP,结果,耗时(ms), 调用时间戳(ms)
serviceA,172.17.60.2,serviceB,172.17.60.3,true,20,1592454780000「2020-06-18 12:33」
serviceA,172.17.60.2,serviceB,172.17.60.3,true,30,1592454780000「2020-06-18 12:33」
serviceA,172.17.60.2,serviceB,172.17.60.3,true,50,1592454900000「2020-06-18 12:34」

serviceZ,172.17.60.8,serviceA,172.17.60.2,true,420,1592454780000「2020-06-18 12:33」
serviceZ,172.17.60.8,serviceC,172.17.60.4,true,120,1592454780000「2020-06-18 12:33」
```



Tire

计算触发警报规则的结果：使用ForkJoin并发框架多线程计算

将多个警报规则应用于存储表时，使用多线程加速，我们使用了ForkJoin并发框架来充分利用CPU多核

3. 2 最长链路查询加速

建立查找缓存哈希表

将报警规则所定义的查找点对应的最长链路缓存进哈希表，使得查询的时候可以直接查找哈希表

使用数组代替哈希表，使用自定义哈希函数

哈希函数：hashcode = hascaller.hashCode() ^ responder.hashCode() ^ time.hashCode() ^ type.hashCode()

数组索引计算: hashcode & (array.length-1)

冲突处理：若索引冲突，将冲突的位置标记为冲突位，使用一个hashmap存储冲突位的数据，查询时遇到冲突标记就转查hashmap，实际并未产生hash冲突，所以实际操作时并未创建hashmap

利用CPU cache

1. 动态选择查询数组的大小(使得不发生冲突或者冲突极少的 最小大小)

2. 后台使用一个额外线程不断得读取查询数组，来维持查询数组在cpu的三级cache
(经测试跑分更加稳定)