开源之夏

eKuiper

规则级别的 CPU 用量统计

申请人：皮昌钱

2024 年 06 月 04 日

目录

1. 主要目标 3
2. 实现 4
   2.1. profile 库 4
   2.2. eKuiper rule CPU 用量统计 5
3. 安排 7
4. 写在最后 7

1. 主要目标

eKuiper 中有以下几个重要的基本概念：规则，流，表。其中主要的与本项目相关的概念是规则。

一个规则代表了一个流处理流程，定义了从将数据输入流的数据源到各种处理逻辑，再到将数据输入到外部系统的动作。

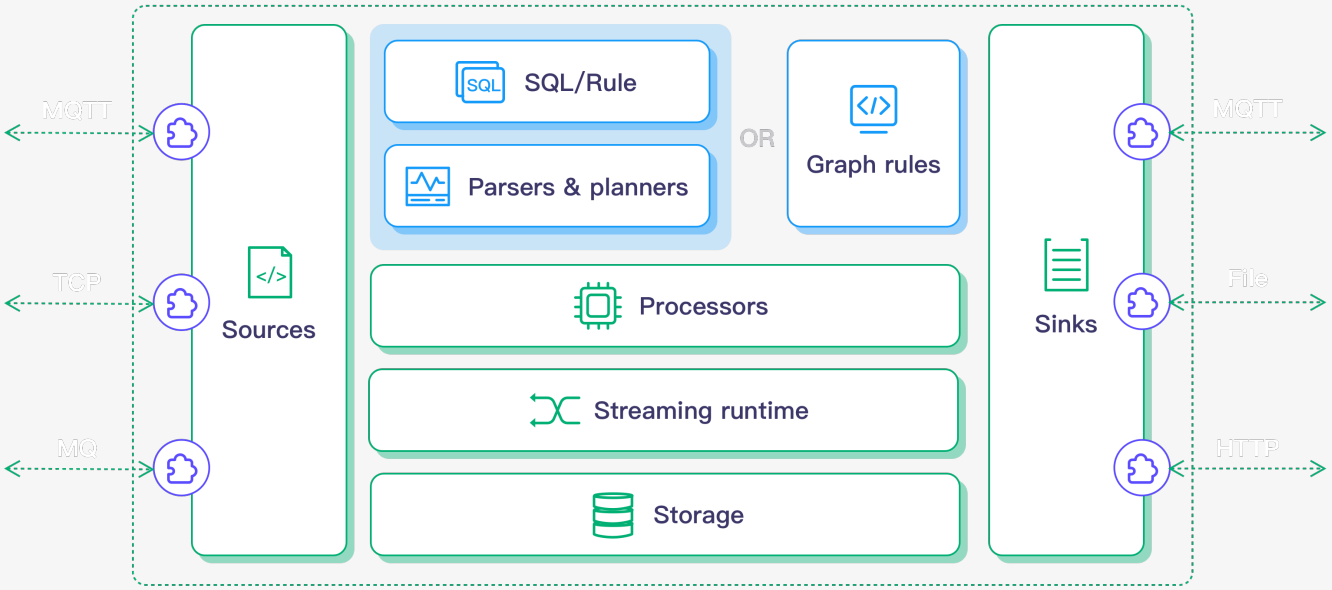

图 1 eKuiper 架构

在 eKuiper 中，可以部署多个流式规则来进行数据的处理。但是当 eKuiper 中部署多条规则时，我们无法通过目前的 go profile 来快速得知每条规则的 CPU 用量，以及规则与规则之间的 CPU 用量对比，从而来快速定位 eKuiper 资源消耗问题。

本项目的主要目标便是解决上述问题，通过以下方法完成：

- 开发兼容的、接口友好的、可独立的 profile 库
  - 兼容：引入这一库不对项目本身造成冲突
  - 接口友好：提供的 profile 接口易使用、易理解
  - 可独立：该库可以独立成一个小的开源库，为其他项目提供更加高级的 profile 功能
- 对于 eKuiper 项目中流式规则的代码使用 `pprof.SetGoroutineLabels` 埋点采集性能数据，使用开发的 profile 库和编写的性能数据处理函数提取关键数据，prometheus 开源库采集提取出的关键数据。用 prometheus 提供的可视化工具或者其他开源可视化工具进行单条规则的性能观察和多条规则之间的性能对比

2. 实现

2.1. profile 库

设计的 profile 库架构图如下：

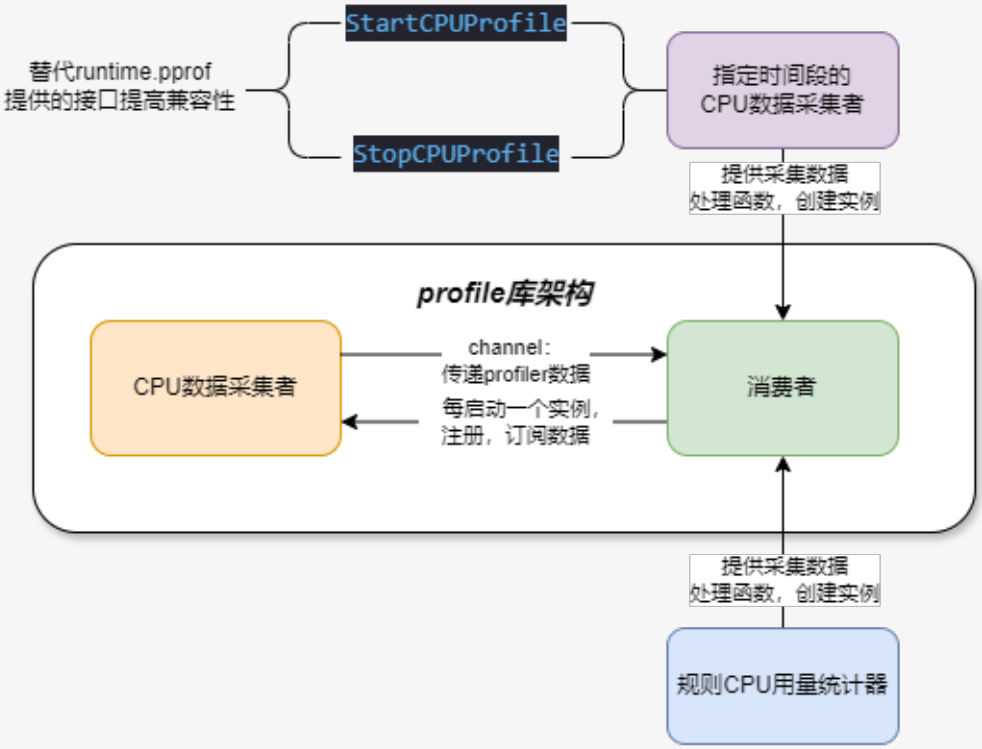

图 2 profile 库架构

当程序启动时，通过 profile 库提供的 `StartCPUProfiler` 启动 CPU 数据采集者协程，该协程定时采集程序的 profile 数据（可配置采集间隔时间和每次采集的时间窗口），每次采集之后会将数据通过 channel 发送给注册的消费者。

项目通过引入 profile 库，添加消费者实例来使用 profile 库的功能。添加消费者实例必须通过设置回调函数来设置消费者处理数据的方式。添加一个新的消费者实例并使用 profile 库提供的 `StartConsume` 启动消费者首先会在 CPU 数据采集者处注册，然后启动消费者协程等待 CPU 数据采集者。成功收到数据后，消费者会调用回调函数来处理数据。（对应图 2 的下部）。

profile 库中的 CPU 数据采集者是通过 `runtime/pprof` 库提供的 `StopCPUProfile` 和 `StartCPUProfile` 来实现指定时间段性能采集。为了保证兼容性，防止引入该库的项目调用 `runtime/pprof` 库提供的 `StopCPUProfile` 和 `StartCPUProfile` 而产生的混乱，profile 库应该提供替代的接口（名称不变），这两个接口的实现仍是通过消费者来实现（对应图 2 的上部）。

另外，go 还提供了 web 查看程序性能的方法，可以通过下面的代码来使用这一功能：

```
import _ "net/http/pprof"

func main() {
  go func() {
    log.Println(http.ListenAndServe(addr, mux))
  }()
  // 程序逻辑
}
```

大致的实现原理是引入 net/http/pprof 意味着程序启动会先执行库中的 init 函数，该库的 init 函数中的 http.HandleFunc("/debug/pprof/profile", Profile) 设置的网络接口的处理函数同样使用了 runtime/pprof 库提供的 StopCPUProfile 和 StartCPUProfile。所以需要在 profile 库提供同样的功能，但是需要将 /debug/pprof/profile 的处理函数用上文实现的替代接口实现。

可以在线查看 profile 库的 demo 实现。

2.2. eKuiper rule CPU 用量统计

在 eKuiper 中，每条规则有规则 id，在 go 中的上下文（contex）中记录规则 id，这样做能够让每个协程都能通过上下文获取到启动自身的规则 id。

在整个规则的代码逻辑中，通过 pprof.SetGoroutineLabels(ctx context.Context) 埋点设置来采集每条规则的性能数据，以下是埋点的代码：

```
ctx = pprof.WithLabels(ctx, pprof.Labels("rule", ruleID))
pprof.SetGoroutineLabels(ctx)
```

eKuiper rule CPU 用量统计首先引入 2.1 实现的 profile 库，然后添加消费者实例，为消费者实例提供数据处理回调函数。对于 eKuiper 项目来说，处理数据要关注标签为 rule 的数据，并且需要对每条规则的 CPU 用量单独统计。对于统计出的数据，可以定义对应的 prometheus 收集器并注册到 prometheus 中，这样便能够在 prometheus 中查看每条规则的 CPU 用量情况，并且能够通过编写 prometheus 的查询语句 PromQl 来对比多条规则之间的 CPU 用量。

具体的实现逻辑、埋点的位置考量需要在项目完成期间完成。但是我编写了一个小的 demo 来测试我的 profile 库，这一 demo 会不断的并发执行两个任务——并行筛法求素数和并行归并排序。设置两个任务都是并行的目的是更加贴近项目（一条规则的执行流程包含多个协程）。demo 中的并行归并排序埋点代码如下：

```
func ParallelMergeSort(ctx context.Context, array []int, enableProfile
bool) {
```

```
    if enableProfile {
      pprof.SetGoroutineLabels(ctx)
    }

    wg := sync.WaitGroup{}
    wg.Add(2)
    // split array code ellipsis
    go func() {
      ParallelMergeSort(ctx, leftArray, enableProfile)
      wg.Done()
    }()
    go func() {
      ParallelMergeSort(ctx, rightArray, enableProfile)
      wg.Done()
    }()
    wg.Wait()
    // merged array code ellipsis
  }
```

可以看到每个协程都有调用 pprof.SetGoroutineLabels(ctx)来埋点统计性能信息。对应并行筛法求素数也做了同样的处理。demo 调用了 2.1 中的 profile 库并按照规则编写了数据处理函数统计了两个任务的 CPU 量。单个任务的数据查看和两个任务之间的数据对比，我使用了 go 中的 plot 库，使用两个任务的 CPU 占用量(百分比)生成折线图，并且可以通过访问网络接口实时监控折线图的变化，效果图如下：

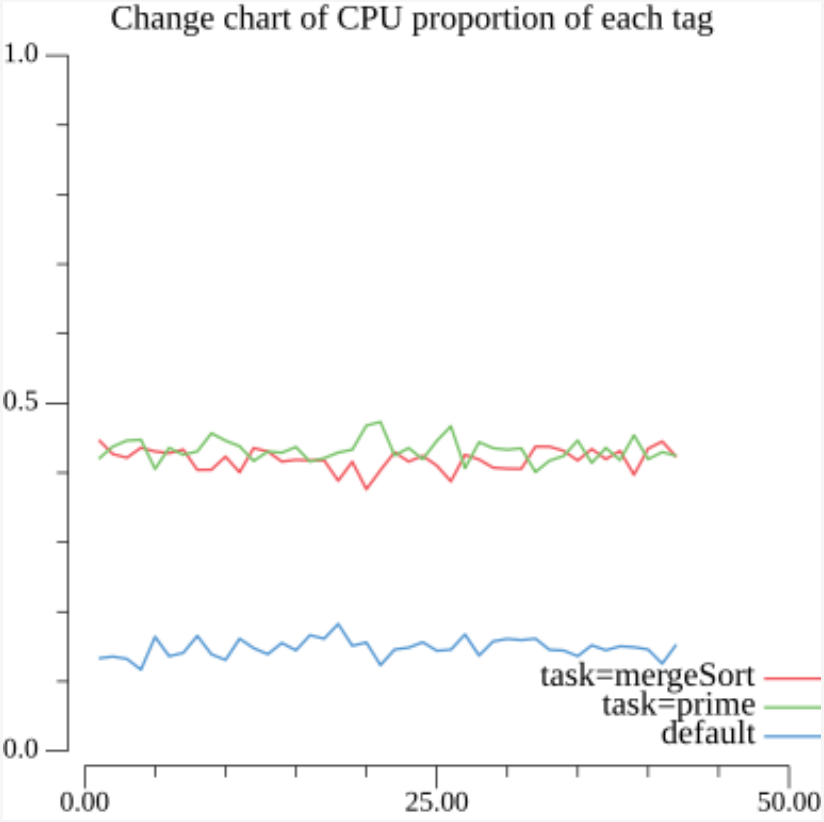

图 3 demo 监控图

3. 安排

- week1：与导师和社区沟通初步实现方案
- week2~3：阅读项目源码
- week4~8
    - 编写代码
    - 测试功能和兼容性
    - 收集社区反馈信息
    - 改进迭代
- week9：
    - 编写文档
    - 提交 PR，合并代码

4. 写在最后

我的 go 语言编程能力和项目经验足够胜任本项目，并且我对开源项目有着浓厚的兴趣。之所以选择本项目，是因为本项目与我的技术栈相匹配，并且我希望在实际项目中增强对分布式可观测性的理解。希望能够入选参与 eKuiper 开发来为开源社区贡献自己的一份力量。