容器追踪模块设计

容器追踪模块是基于进程追踪模块实现的，其数据结构为：

```
struct container_event {
        struct process_event process;
        unsigned long container_id;
        char container_name[50];
};
```

容器追踪模块由`container_tracker`实现

```
struct container_tracker : public tracker_with_config<container_env,
container_event>
{
  struct container_env current_env = { 0 };
  struct container_manager &this_manager;
  std::shared_ptr<spdlog::logger> container_logger;

  container_tracker(container_env env, container_manager &manager);
  void start_tracker();

  void fill_event(struct process_event &event);

  void init_container_table();

  void print_container(const struct container_event &e);

  void judge_container(const struct process_event &e);

  static int handle_event(void *ctx, void *data, size_t data_sz);
};
```

在container_tracker中，定义了一个judge_container函数，用于判断一个进程是否属于一个容器。函数如下所示：

```
void container_tracker::judge_container(const struct process_event &e)
{
```

```
  if (e.exit_event)
  {
    this_manager.mp_lock.lock();
    auto event = this_manager.container_processes.find(e.common.pid);
    // remove from map
    if (event != this_manager.container_processes.end())
    {
      event->second.process.exit_event = true;
      print_container(event->second);
      this_manager.container_processes.erase(event);
    }
    this_manager.mp_lock.unlock();
  }
  else
  {
    /* parent process exists in map */
    this_manager.mp_lock.lock();
    auto event = this_manager.container_processes.find(e.common.ppid);
    this_manager.mp_lock.unlock();
    if (event != this_manager.container_processes.end())
    {
      struct container_event con = { .process = e, .container_id =
(*event).second.container_id };
      strcpy(con.container_name, (*event).second.container_name);
      this_manager.mp_lock.lock();
      this_manager.container_processes[e.common.pid] = con;
      print_container(this_manager.container_processes[e.common.pid]);
      this_manager.mp_lock.unlock();
    }
    else
    {
      /* parent process doesn't exist in map */
      struct process_event p_event = { 0 };
      p_event.common.pid = e.common.ppid;
      fill_event(p_event);
      if ((p_event.common.user_namespace_id != e.common.user_namespace_id) ||
          (p_event.common.pid_namespace_id != e.common.pid_namespace_id) ||
          (p_event.common.mount_namespace_id != e.common.mount_namespace_id))
      {
        std::unique_ptr<FILE, int (*)(FILE *)> fp(popen("docker ps -q", "r"),
pclose);
        unsigned long cid;
        /* show all alive container */
        pid_t pid, ppid;
        while (fscanf(fp.get(), "%lx\n", &cid) == 1)
        {
```

```
47            std::string top_cmd = "docker top ", name_cmd = "docker inspect -f
   '{{.Name}}' ";
48            char hex_cid[20], container_name[50];
49            sprintf(hex_cid, "%lx", cid);
50            top_cmd += hex_cid;
51            name_cmd += hex_cid;
52            std::unique_ptr<FILE, int (*)(FILE *)> top(popen(top_cmd.c_str(),
   "r"), pclose),
53                name(popen(name_cmd.c_str(), "r"), pclose);
54            fscanf(name.get(), "/%s", container_name);
55            char useless[150];
56            /* delet the first row */
57            fgets(useless, 150, top.get());
58            while (fscanf(top.get(), "%*s %d %d %*[^\n]\n", &pid, &ppid) == 2)
59            {
60              this_manager.mp_lock.lock();
61              /* this is the first show time for this process */
62              if (this_manager.container_processes.find(pid) ==
   this_manager.container_processes.end())
63              {
64                struct container_event con = {
65                  .process = e,
66                  .container_id = cid,
67                };
68                strcpy(con.container_name, container_name);
69                this_manager.container_processes[pid] = con;
70                print_container(this_manager.container_processes[pid]);
71              }
72              this_manager.mp_lock.unlock();
73            }
74          }
75        }
76      }
77    }
78  }
79
```

代码整体作用

1. **检测进程是否退出**：如果进程退出，则从容器进程映射中移除，并打印容器信息。
2. **检测新进程是否属于某个容器**：如果是新启动的进程，判断其父进程是否已经在容器内。
3. **在容器内启动的进程**：将新进程与其父进程所属的容器相关联，并打印容器信息。
4. **未找到父进程的情况**：通过 Docker 命令查找当前运行的容器及其进程，更新容器进程映射并打印相关信息。

具体作用和流程细节

1. **进程退出事件**：
   - 如果 `e.exit_event` 为 `true`，表示当前事件是一个进程退出事件。
   - 获取全局容器进程映射（`this_manager.container_processes`）中的 `PID` 对应项。
   - 如果找到了对应的事件，则：
     - 标记该进程为退出状态。
     - 打印该容器信息。
     - 从映射中移除该事件。
2. **新进程事件**：
   - 如果 `e.exit_event` 为 `false`，表示当前事件是一个新进程事件。
   - 首先尝试通过父进程 ID（`ppid`）找到对应的父进程事件。
   - 如果找到了父进程事件，则：
     - 创建一个新的容器事件结构体，并将新进程与父进程所属的容器关联。
     - 打印该容器信息。
   - 如果没有找到父进程事件：
     - 构造一个新的 `process_event` 结构体，将父进程 ID 设为当前事件的父进程 ID。
     - 使用 `fill_event` 函数完善父进程事件信息。
     - 检查父进程的命名空间信息，如果与当前事件不同：
       - 使用 `docker ps -q` 命令获取所有正在运行的容器 ID。
       - 遍历每个容器，使用 `docker top` 和 `docker inspect` 来获取容器中的进程及其名称。
       - 如果找到新的进程，则将其与容器关联，并打印容器信息。