中山大學
SUN YAT-SEN UNIVERSITY

简历简要补充图片

肖霖畅

个人概述

- 肖霖畅
  - 年龄：24岁
  - 硕士 中山大学 计算机学院(软件学院) 计算机技术 [2022.09 - 2025.06]
  - 本科 中山大学 计算机学院(软件学院) 软件工程 [2018.09 - 2022.06]
  - 个人主页：https://xlcbingo1999.github.io/

- 专业技能
  - **编程语言**: 熟悉**Golang**, Python, C/C++, Swift/Objective-C; 了解Go的goroutine调度机制等底层原理
  - **开发工具**: 了解**Kubernetes, Docker**等相关基础; 熟悉Linux的使用和维护; 熟悉Git等工具

大纲

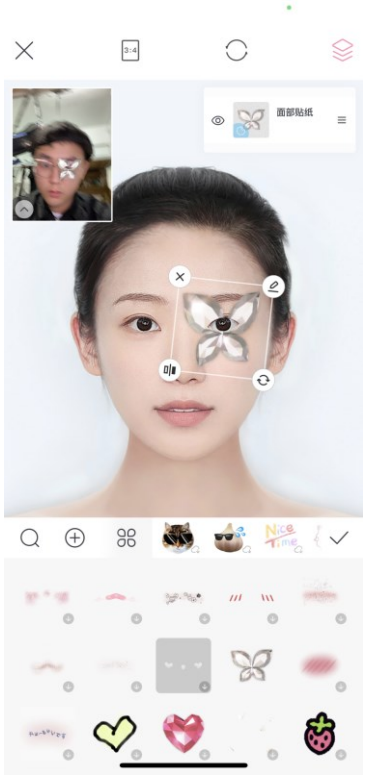
1. **实习经历** **(2021.05 - 2021.11)**
2. 基于Kubernetes的容器化AI云平台-DeepAI 和 研究论文 (2021.11-至今)
3. 基于异构超算的多元任务运行时系统 (2023.12-至今)

实习 – 上线业务界面示例

- 搭建 多媒体相册SDK （支持多媒体文件的增删，修图修视频，预览等功能）
- 参与 自定义风格平台 的搭建 （支持用户自定义风格并在拍照时实时应用风格）
- 搭建"拍摄-存草稿-修图-分享-回流"的社交链路全流程UI逻辑

多媒体相册SDK

自定义风格平台

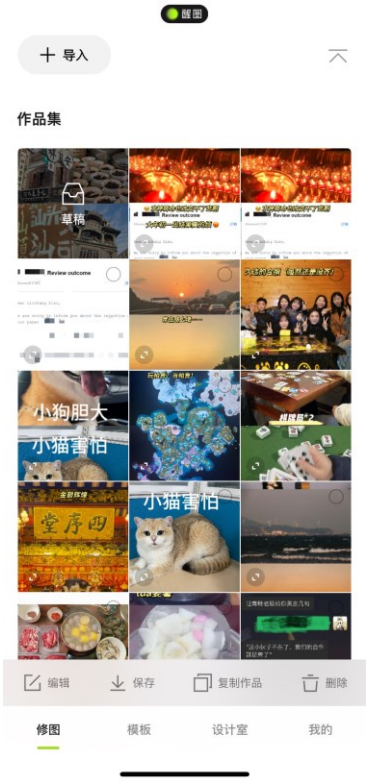


业务贡献: 可复用的多媒体相册SDK

- **高内聚低耦合**的多媒体相册SDK
  - 从主业务中抽离，避免和大量老代码发生耦合
  - 以组件的形式向上提供服务，支持其他应用接入，避免重写轮子

相机APP

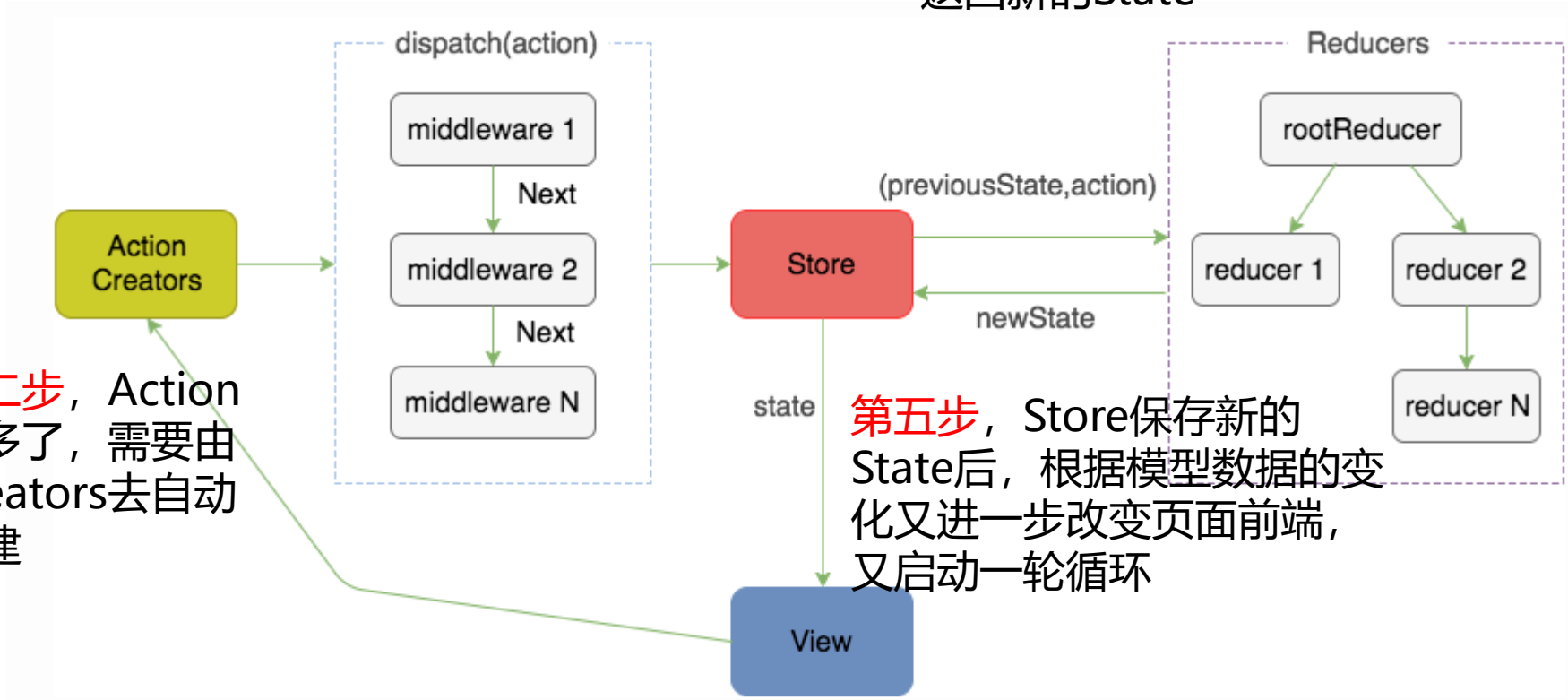
修图APP



方案贡献: Redux模式(管理异步高交互数据流)

第三步（可选），修改Store前，由各部分中间件[日志、错误处理、API请求、路由等]进行预处理

第四步，Store收到Action后，发送给Reducers，经过判断逻辑后返回新的State



第二步，Action太多了，需要由Creators去自动创建

第五步，Store保存新的State后，根据模型数据的变化又进一步改变页面前端，又启动一轮循环

第一步，用户操作页面，需要更新页面的模型数据，用户不应该直接操作后端状态，因此用Action发送

大纲

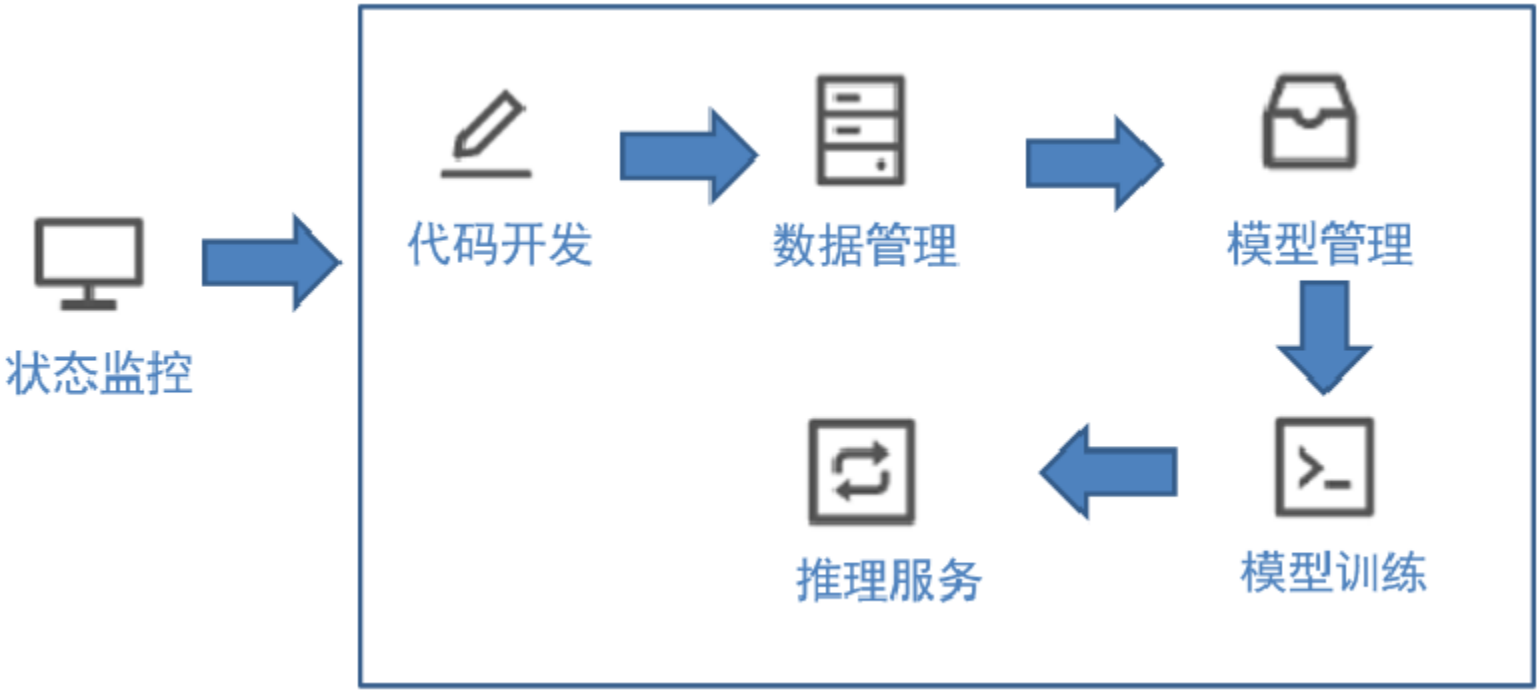
1. 实习经历 (2021.05 - 2021.11)

2. **基于Kubernetes的容器化AI云平台-DeepAI 和 研究论文** (2021.11-至今)

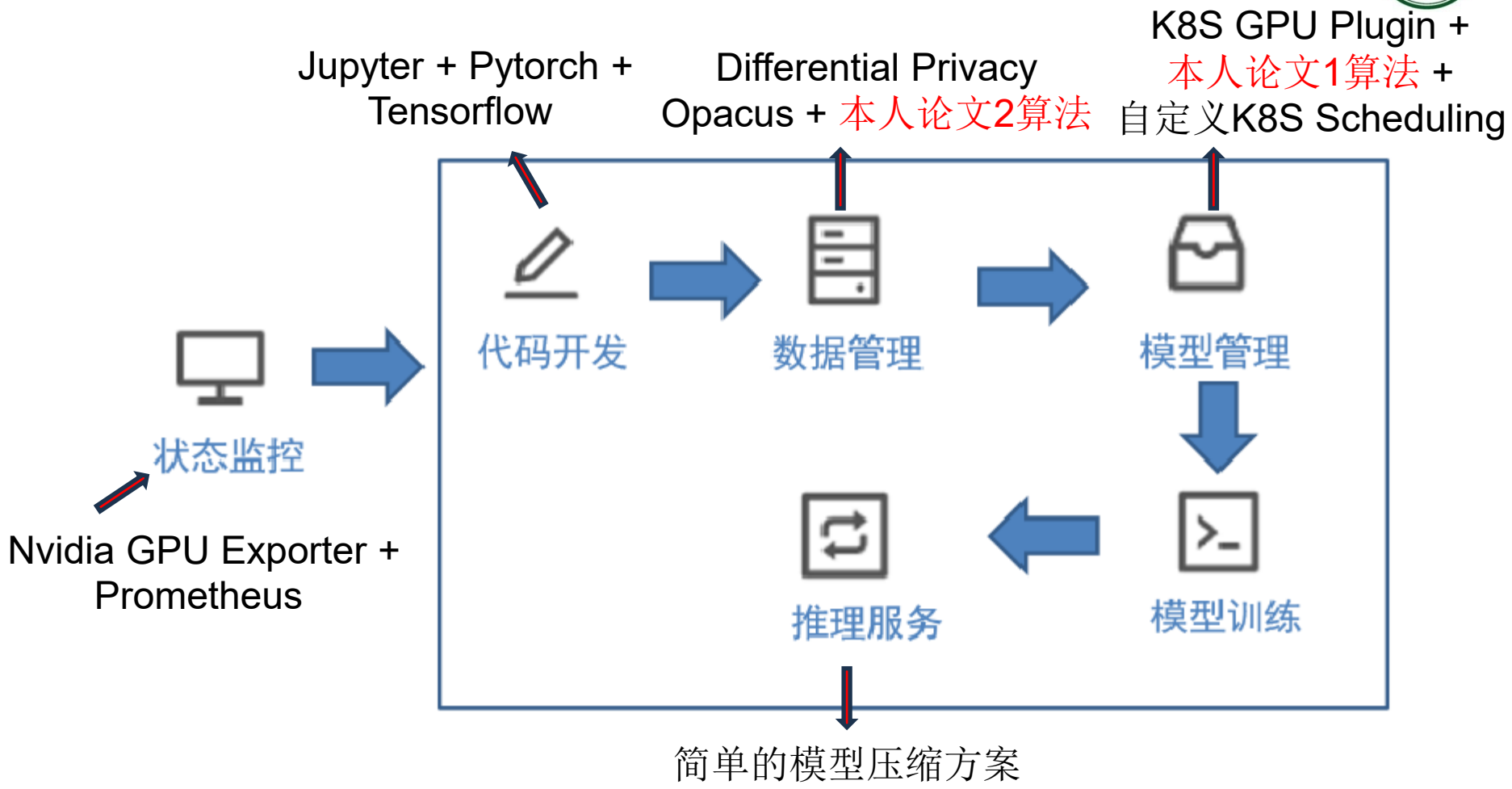
3. 基于异构超算的多元任务运行时系统 (2023.12-至今)

DeepAI的项目简介和承担工作

- 一站式AI任务平台，用户可以用平台提供的环境进行**AI任务代码在线开发**；
- 平台负责**调度任务**并分配训练资源，并支持将训练模型部署为推理服务；
- 平台支持根据训练任务的价值和数据的隐私预算**主动获取数据**；
- 平台依据整体资源水位实现**集群资源的弹性动态调整**
- **承担工作：后端开发+计算资源调度算法设计实现+数据调度算法设计实现**

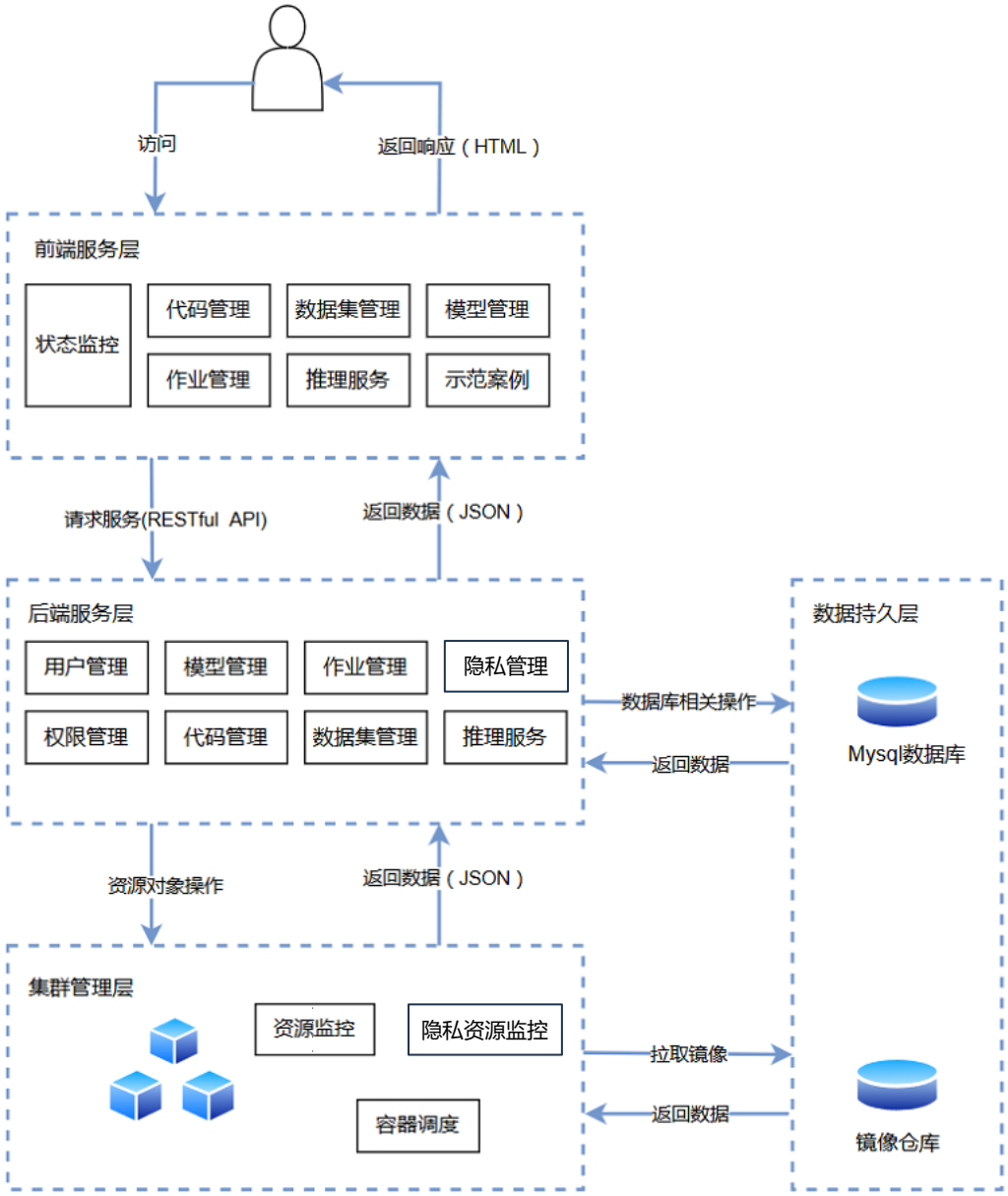


DeepAI的核心任务流



- 论文测试的模拟系统均复用DeepAI代码, 避免了重复造轮子, 另一方面推动了模拟算法的快速落地

DeepAI的系统架构



大纲

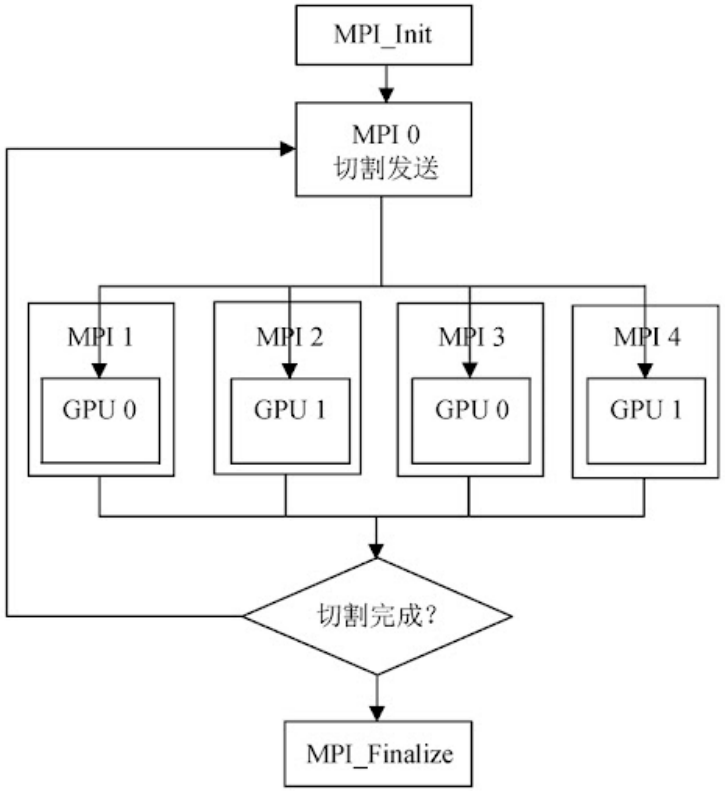
1. 实习经历 (2021.05 - 2021.11)

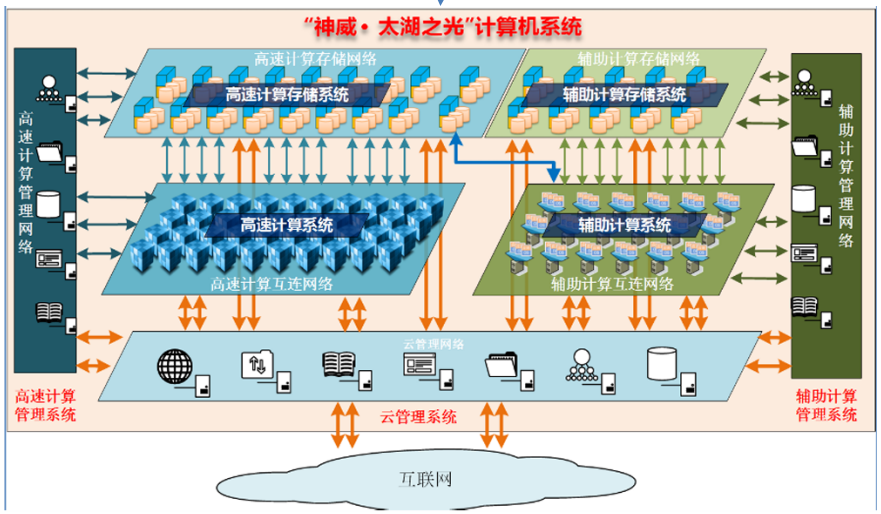
2. 基于Kubernetes的容器化AI云平台-DeepAI 和 研究论文 (2021.11-至今)

3. **基于异构超算的多元任务运行时系统** (2023.12-至今)

运行时系统概述

- **项目背景**：为解决应用多元化(MPI应用、UCX应用等等...)与超算环境异构化所带来的并行编程开发困难和运行效率不足的问题，设计实现一套支持AI等任务和不同超算体系结构的任务级运行时系统
- 运行时系统需要考虑：任务在哪里执行；数据在哪里放置

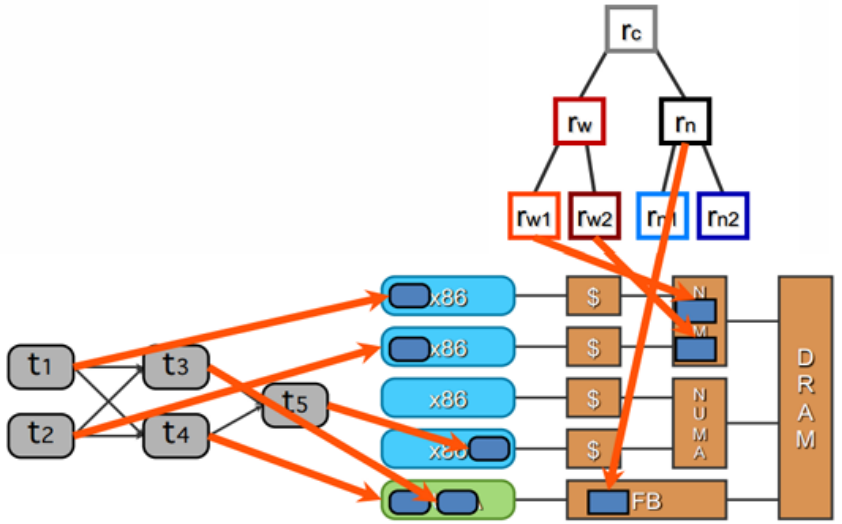




运行时系统概述

- 现有运行时系统的问题
  - 多元应用：需要用户利用多种语言、了解多种编程模式、需要用不同的同步原语显式处理并行代码的同步问题
  - 异构设备：不同架构的设备提供的软件栈不同，用户需要仔细研究不同架构再去需修改代码
  - 自定义调度问题：用户缺少在上层自己定义调度策略的方法

- 我们的运行时系统
  - 提供通用的数据抽象和任务接口
  - 中间层屏蔽底层设备的不同
  - 支持用户自定义调度算法



运行时系统的用户界面例: 并行斐波那契

```
enum TaskIDs {
  TOP_LEVEL_TASK_ID,
  FIBONACCI_TASK_ID,
  SUM_TASK_ID,
};
```

第一步，注册任务的唯一ID

```
int main(int argc, char **argv)
{
  Runtime::set_top_level_task_id(top_id: TOP_LEVEL_TASK_ID);

  {
    TaskVariantRegistrar registrar(task_id: TOP_LEVEL_TASK_ID, variant_name: "top_level");
    registrar.add_constraint(constraint: ProcessorConstraint(kind: Processor::LOC_PROC));
    Runtime::preregister_task_variant<top_level_task>(registrar, task_name: "top_level");
  }

  {
    TaskVariantRegistrar registrar(task_id: FIBONACCI_TASK_ID, variant_name: "fibonacci");
    registrar.add_constraint(constraint: ProcessorConstraint(kind: Processor::LOC_PROC));
    Runtime::preregister_task_variant<int, fibonacci_task>(registrar, task_name: "fibonacci");
  }

  {
    TaskVariantRegistrar registrar(task_id: SUM_TASK_ID, variant_name: "sum");
    registrar.add_constraint(constraint: ProcessorConstraint(kind: Processor::LOC_PROC));
    registrar.set_leaf(is_leaf: true);
    Runtime::preregister_task_variant<int, sum_task>(registrar, task_name: "sum");
  }

  // Callback for registering the inline mapper
  Runtime::add_registration_callback(callback: mapper_registration);

  return Runtime::start(argc, argv);
}
```

第二步，设置根任务

第三步，注册每个任务和对应的函数名，给任务增加计算资源（即处理器的约束条件）

第四步，开始执行前的回调注册（自定义调度方案、注册运行时变量等）

第五步，启动并行程序执行

运行时系统的用户界面例: 并行斐波那契

```
void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx, Runtime *runtime) {
  int num_fibonacci = 15;
  const InputArgs &command_args = Runtime::get_input_args();
  for (int i = 1; i < command_args.argc; i++) {
    // Skip any legion runtime configuration parameters
    if (command_args.argv[i][0] == '-') {
      i++;
      continue;
    }

    num_fibonacci = atoi(nptr: command_args.argv[i]);
    assert(num_fibonacci >= 0);
    break;
  }
  printf(format: "Computing the first %d Fibonacci numbers...\n", num_fibonacci);

  std::vector<Future> fib_results;

  Future fib_start_time = runtime->get_current_time(ctx);
  std::vector<Future> fib_finish_times;

  for (int i = 0; i < num_fibonacci; i++) {
    TaskLauncher launcher(tid: FIBONACCI_TASK_ID, arg: TaskArgument(arg: &i, argsize: sizeof(i)));
    fib_results.push_back(x: runtime->execute_task(ctx, launcher));
    fib_finish_times.push_back(x: runtime->get_current_time(ctx, precondition: fib_results.back()));
  }

  for (int i = 0; i < num_fibonacci; i++) {
    int result = fib_results[i].get_result<int>();
    double elapsed = (fib_finish_times[i].get_result<double>() -
                      fib_start_time.get_result<double>());
    printf(format: "Fibonacci(%d) = %d (elapsed = %.9f s)\n", i, result, elapsed);
  }

  fib_results.clear();
}
```

返回Future

支持创建子任务

等待Future结果的返回并获取值

运行时系统的用户界面例: 并行斐波那契

```
int fibonacci_task(const Task *task,
                   const std::vector<PhysicalRegion> &regions,
                   Context ctx, Runtime *runtime) {
    assert(task->arglen == sizeof(int));
    int fib_num = *(const int *) task->args;
    if (fib_num == 0)
        return 0;
    if (fib_num == 1)
        return 1;

    // Launch fib-1
    const int fib1 = fib_num - 1;
    TaskLauncher t1(tid: FIBONACCI_TASK_ID, arg: TaskArgument(arg: &fib1, argsize: sizeof(fib1)));
    Future f1 = runtime->execute_task(ctx, launcher: t1);

    // Launch fib-2
    const int fib2 = fib_num - 2;
    TaskLauncher t2(tid: FIBONACCI_TASK_ID, arg: TaskArgument(arg: &fib2, argsize: sizeof(fib2)));
    Future f2 = runtime->execute_task(ctx, launcher: t2);

    TaskLauncher sum(tid: SUM_TASK_ID, arg: TaskArgument(arg: NULL, argsize: 0));
    sum.add_future(f: f1);
    sum.add_future(f: f2);
    Future result = runtime->execute_task(ctx, launcher: sum);

    return result.get_result<int>();
}

int sum_task(const Task *task,
             const std::vector<PhysicalRegion> &regions,
             Context ctx, Runtime *runtime) {
    assert(task->futures.size() == 2);
    Future f1 = task->futures[0];
    int r1 = f1.get_result<int>();
    Future f2 = task->futures[1];
    int r2 = f2.get_result<int>();

    return (r1 + r2);
}
```

斐波那契函数

求和