
KubeCon
CloudNativeCon
OPEN SOURCE SUMMIT
China 2023

Kubernetes生产环境的容器热迁移

郎叶楠 & 刘华，Tencent

热迁移的历史

2011

- Checkpoint Restore In Userspace项目启动

2015

- 1月 Kubernetes社区成员开启了一个持续至今的讨论：
  Pod lifecycle checkpointing · Issue #3949
- 7月 Kubernetes v1.0.0版本发布，不支持热迁移

2018

- Google borg在LPC会议上介绍了内部使用热迁移的经验

Tencent

2021

- 4月 热迁移项目启动

2022

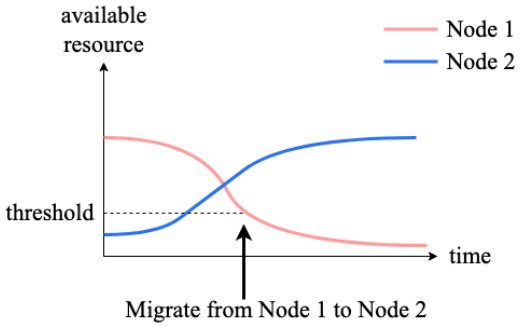
- 2月 热迁移正式上线，日均迁移20k次

热迁移首次落地在K8s生产环境

为什么我们需要热迁移

目标：降低离线训练成本

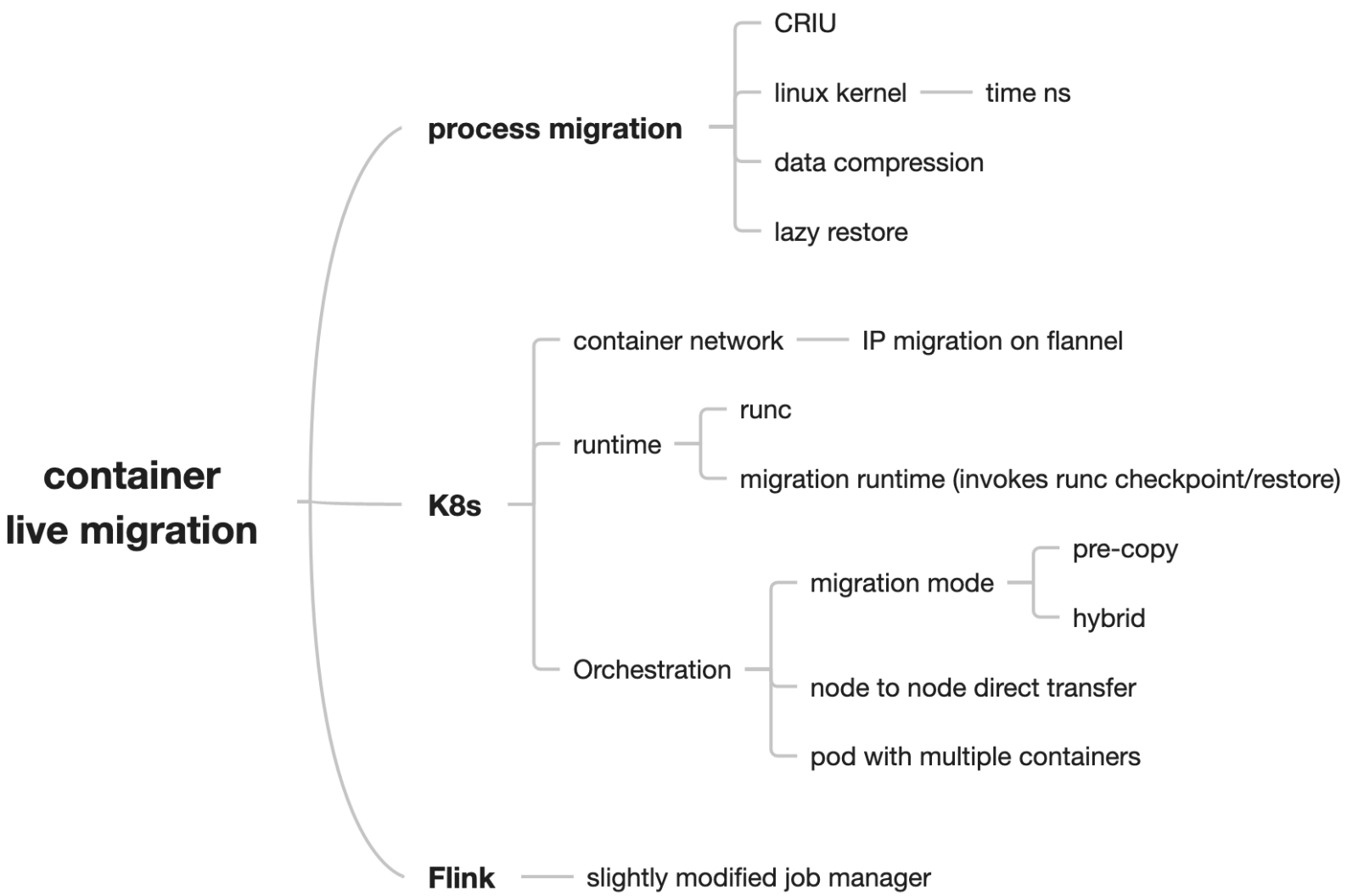
- 算法团队基于Flink开发了离线训练平台
- 对延迟不敏感
- 占用了很大一部分资源



手段：充分利用低价资源

- 价格低的同时稳定性也低，如混部资源、竞价实例
- 希望作业一直运行在高性能的节点上
- Flink task manager单点重调度的开销太大
- 通过热迁移技术实现**低开销无损重调度**

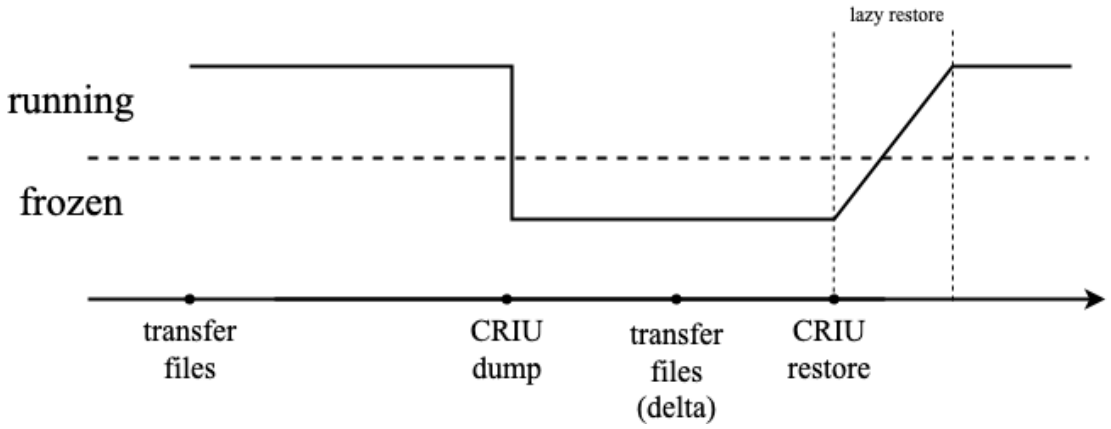
热迁移实现概述



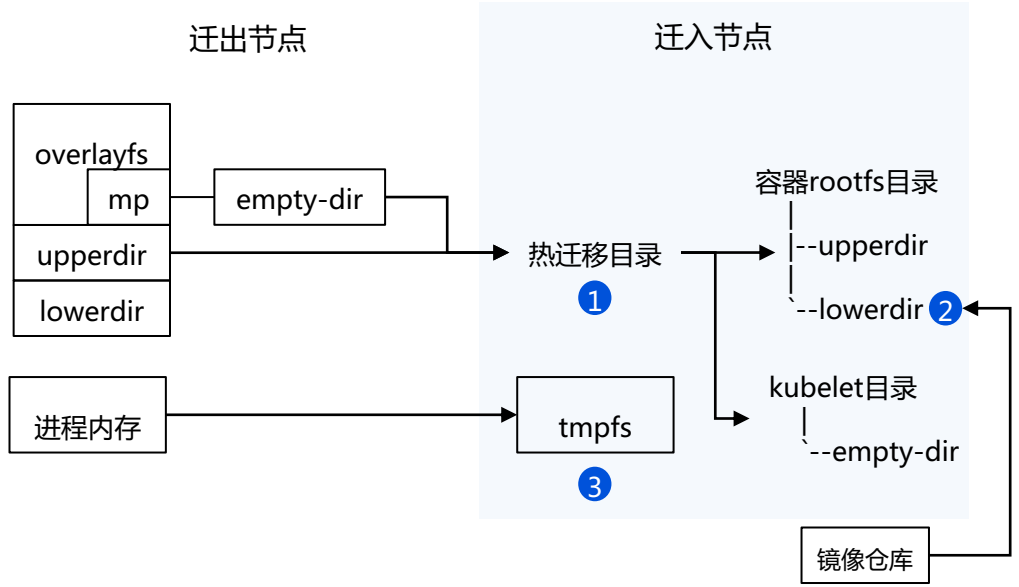
实现效果

- 不需要修改K8s代码
- 不要求高版本K8s
  （我们用的是v1.18版本）
- 支持迁移原生deployment和statefulset

进程迁移



- CRIU的迭代迁移模式性能不稳定，所以我们选择一次性dump整个进程
- 文件增量传输：分两次传输进程的文件系统，第二次只传输文件增量，降低进程冻结时间
- 优化ghost file的处理：不限制大小，支持增量传输
- dump加速：数据压缩 + 并发传输
- lazy restore：先快速恢复进程，再按需加载内存

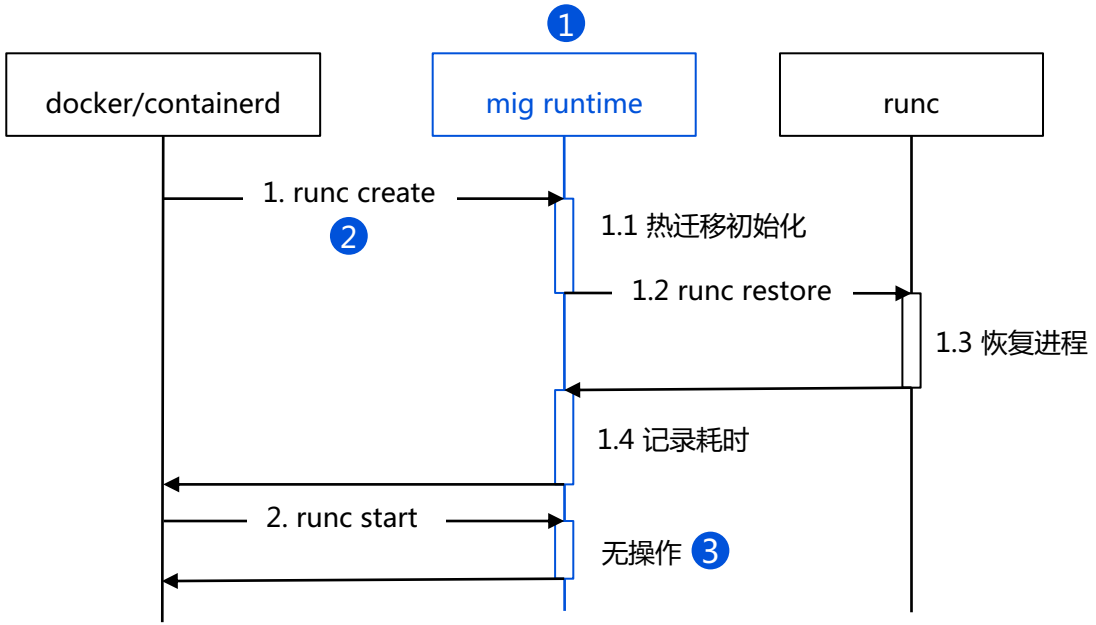


1. 使用rename提升文件系统恢复速度
2. lowerdir来自容器镜像，所以从镜像仓库恢复
3. 进程内存被直接存储在tmpfs

运行时

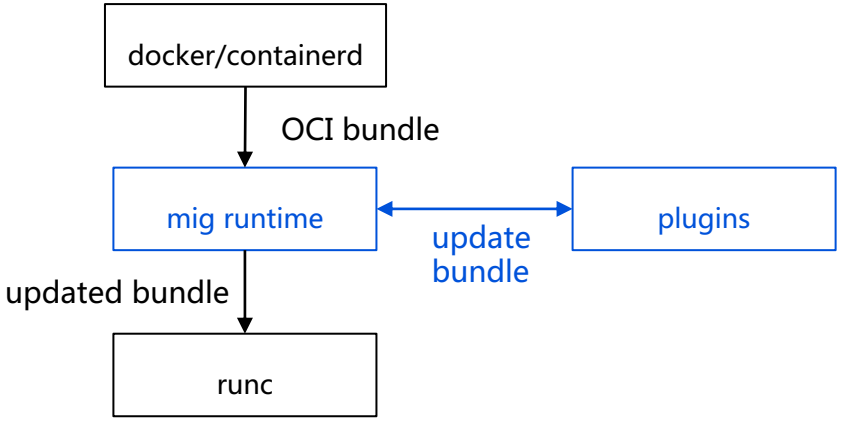
将容器创建替换为容器恢复

不需要修改kubelet/runtime代码



1. mig runtime可以拦截runc调用并根据OCI bundle中的env判断是创建新容器还是恢复容器
2. 需要恢复容器时，将runc create替换为runc restore
3. runc-restore后进程已恢复完毕，所以拦截runc start并直接返回

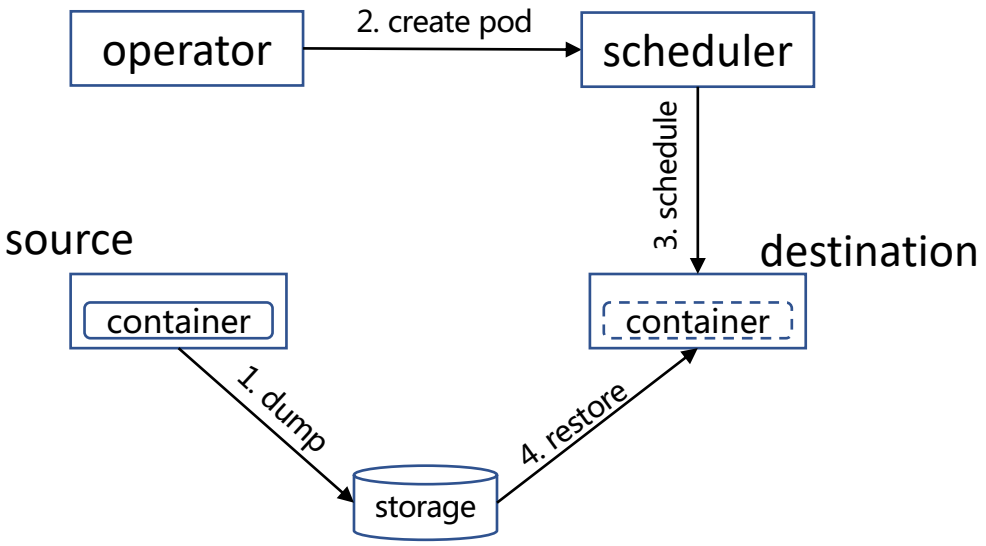
运行时扩展



- K8s只使用了OCI bundle的部分功能（如部分hooks），在bundle层面进行扩展可以使用全部功能
- OCI bundle更贴近操作系统，在此层面进行扩展更灵活也更简单
- 我们基于mig runtime开发了一系列服务于大数据和AI的插件，如设备管理、隔离性提升、日志管理等

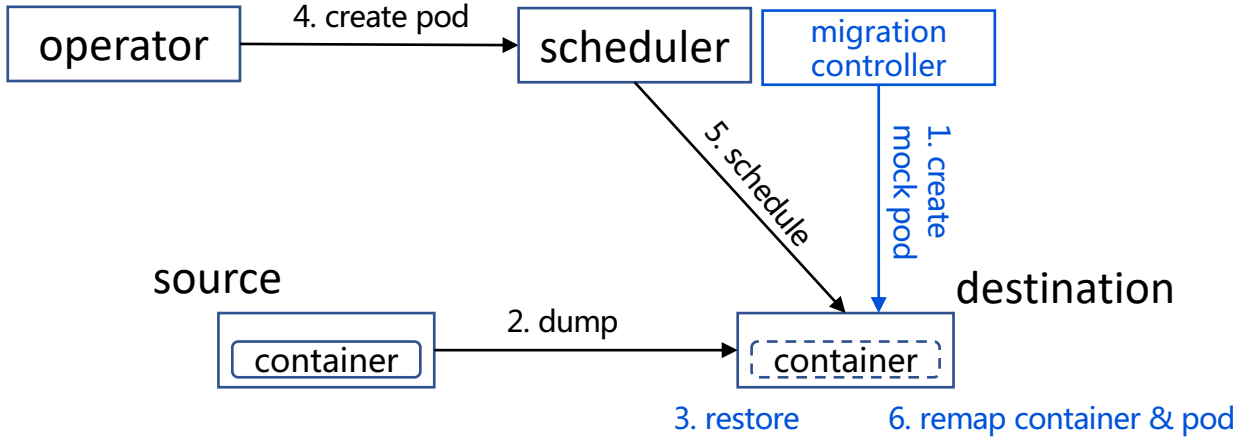
容器编排

两次传输方案



- 作业暂停时间 = $t_{dump} + t_{create} + t_{schedule} + t_{restore}$
- t_{dump}和$t_{restore}$受本机带宽和远端存储性能影响，发生大规模迁移时远端存储可能成为瓶颈
- t_{create}和$t_{schedule}$ 波动较大，在集群繁忙时可能耗时较长
- 新旧Pod存在依赖关系（必须删除旧Pod才会创建新Pod），所以无法实现hybrid迁移模式

点对点直传方案



- 我们在不修改K8s代码的前提下实现了容器与Pod的重映射能力，可以将已存在的容器映射到其他Pod上
- 得益于重映射能力，进程在dump完成后可以立即开始restore
- restore操作完全使用本地数据进行，耗时极短，所以作业暂停时间仅取决于dump进程内存的耗时（进程文件在dump中仅需传输增量）
- 由于operator和scheduler操作都在restore操作之后，所以二者性能对进程暂停时间无影响，这使得热迁移在繁忙集群依旧性能稳定
- 重映射能力解除了新旧Pod间的依赖关系（二者可共存任意长时间），我们基于此实现了hybrid迁移模式

性能对比

Pod重建耗时

百分位数	重启耗时（秒）
10th	7
50th	9
75th	13
90th	17.3
99th	32.66

Pod重建耗时：create Pod、调度、启动pause容、执行init容器、启动容器的总耗时

PS：Pod重建耗时不包括终止容器和删除Pod的耗时，因为这两种耗时在一定程度上可由被优化掉

pre-copy模式耗时(mem: 10G)

百分位数	总耗时（秒）	业务中断（秒）
10th	24	11
50th	31	17
75th	38	21
90th	45	26
99th	63	37

总耗时：从触发热迁移到所有热迁移流程结束的耗时

业务中断：从进程被冻结（执行CRIU dump）到进程在新节点启动完成的耗时

hybrid模式耗时

内存大小	业务中断(s)	pre-dump time(s)	lazy restore time(s)
1	**0.7**	2	1.1
16	**1.3**	15.8	16.6
32	**1.4**	24.8	35.4
64	**2.9**	59.25	73.5

业务中断：从进程被冻结（执行CRIU dump）到进程在新节点启动完成的耗时

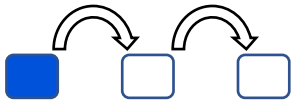
pre-dump time：执行criu pre-dump的累计耗时

lazy restore time：进程恢复后完成所有内存加载的耗时

热迁移与云原生

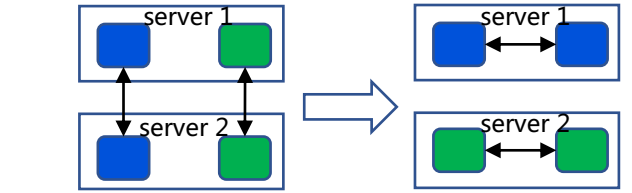
重调度应当是云原生的核心能力

场景1：spot instance



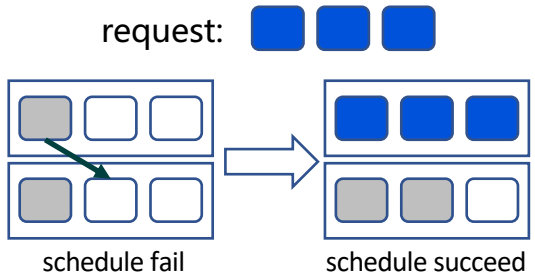
- 在不稳定的资源上为作业提供稳定的运行环境
- 当作业资源被回收时，可迁移至其他节点

场景2：拓扑优化



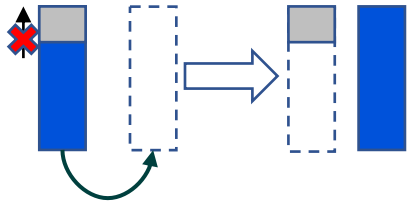
- 根据作业实际运行情况优化拓扑结构，不必完全依赖预测与规划
- 可不断调整作业拓扑至最优状态，不再是“一锤子买卖”

场景3：资源碎片



- 按需动态调整资源分布，满足不同应用的资源需求

场景4：垂直扩容

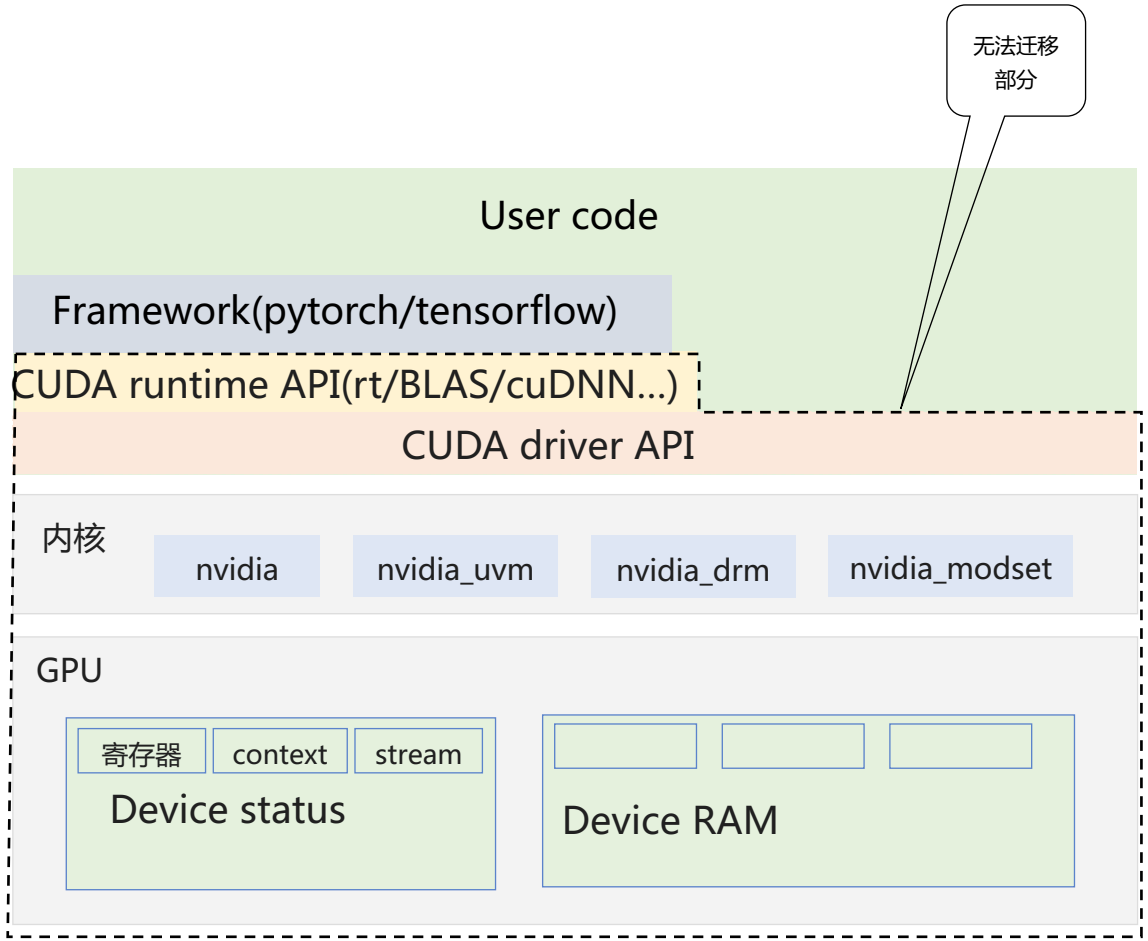


- 本机资源不足以扩容时，迁移到其他节点进行扩容

基于热迁移的低开销重调度

- 利用热迁移技术避免作业重启，降低重调度开销
- 与计算引擎深度结合，提供更通用的热迁移能力
- 通过GPU热迁移实现最优拓扑，提升训练速度

GPU热迁移的难点-以CUDA应用为例



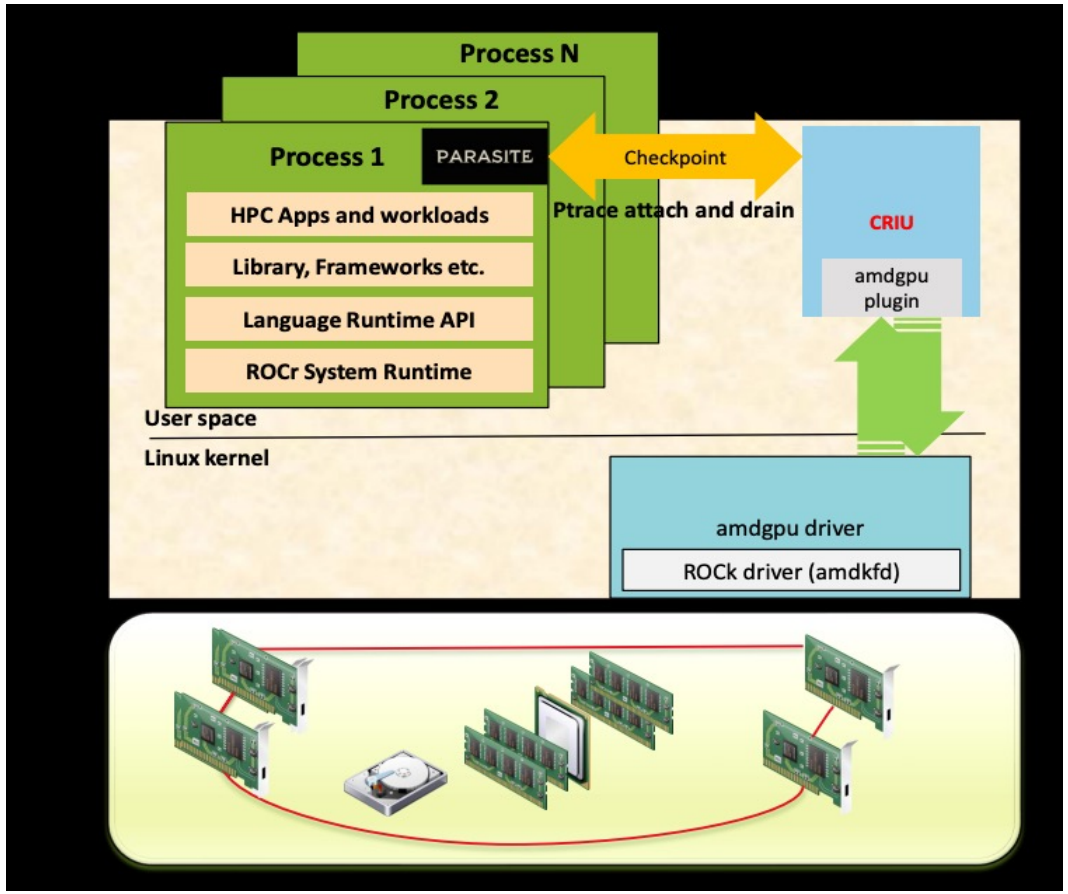
GPU相关软件栈

GPU作为直通设备，迁移存在难题

- GPU硬件状态的导入/导出难题
- CUDA driver层映射显存到应用地址空间
- 设备文件的迁移
  - /dev/nvidiactl
  - /dev/nvidia#num
- 厂商的支持程度
  - NVIDIA vGPU支持热迁移
    - License
    - 虚拟机
  - CUDA不支持checkpoint/restore
  - AMD ROCm在尝试支持

业界的尝试-AMD ROCm

AMD和CRIU社区合作完成GPU作业的Checkpoint/restore

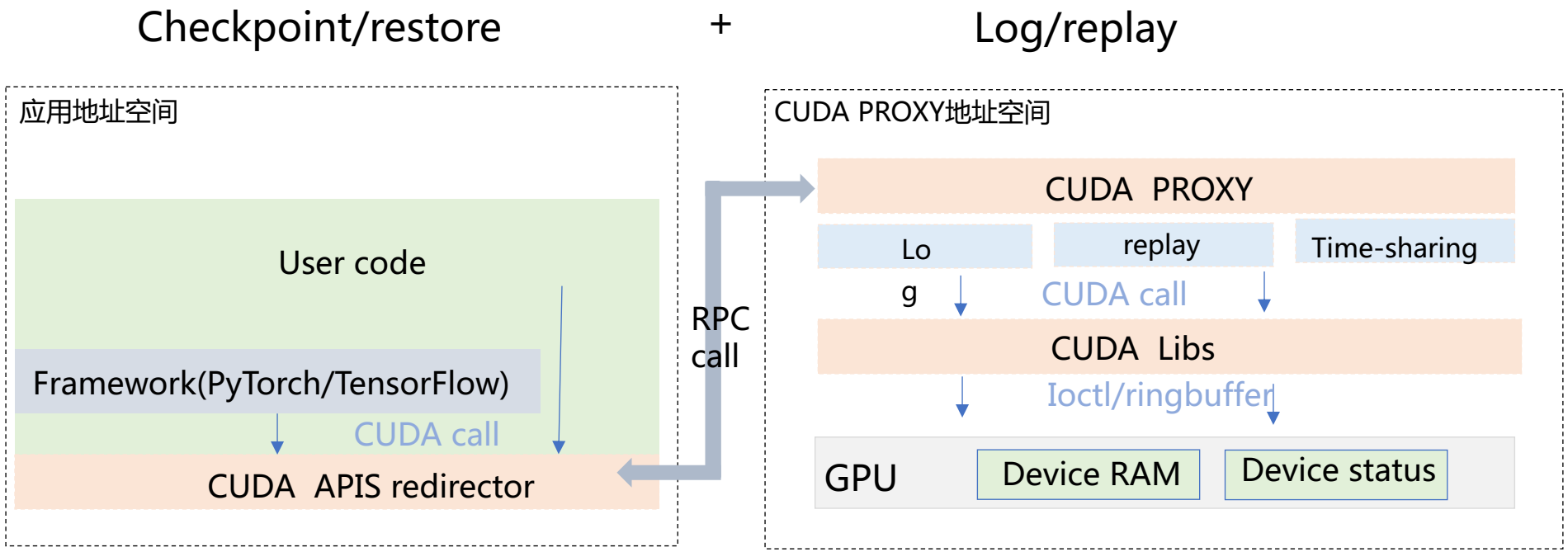


AMD ROCm CR流程

- CRIU：增加3个HOOK以导出/恢复GPU状态
  - CR_PLUGIN_HOOK__RESUME_DEVICES_LATE
  - CR_PLUGIN_HOOK__HANDLE_DEVICE_VMA
  - CR_PLUGIN_HOOK__UPDATE_VMA_MAP
- AMDGPU plugin
  - 链接CRIU和KFD内核模块
  - 保存GPU设备文件
  - Checkpoint/restore GPU状态
- KFD增加GPU资源导出
  - Memory
  - Queues
  - Events
  - Topology
- 扩展IOCT接口
  - CRIU_PAUSE
  - CRIU_PROCESS_INFO
  - CRIU_DUMPER
  - CRIU_RESTORER
  - CRIU_RESUME

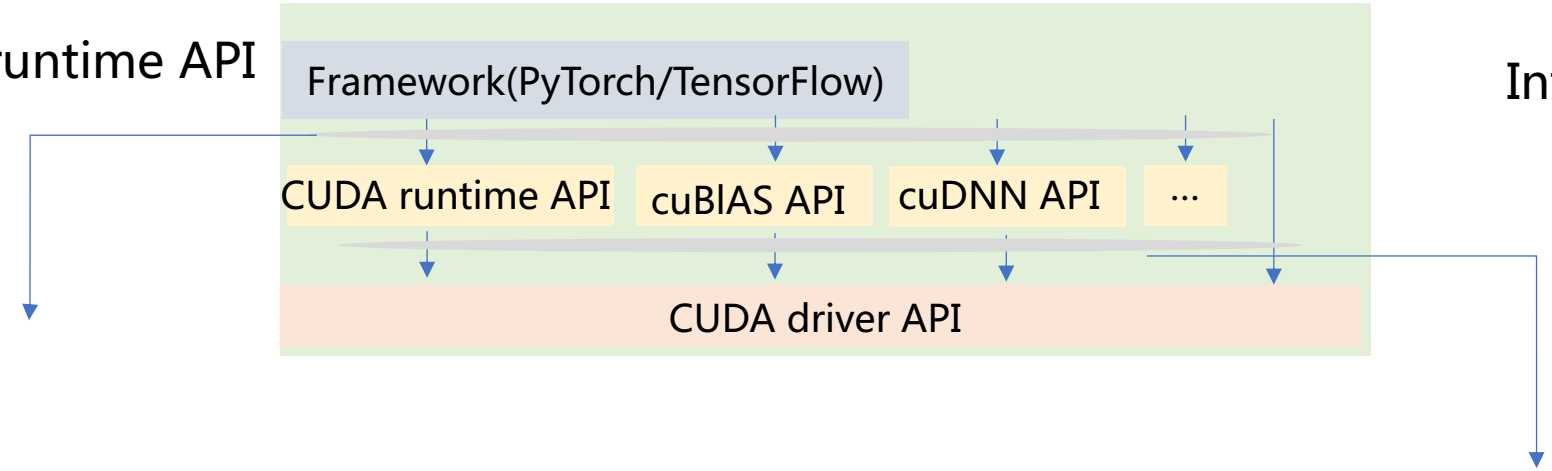
解决方案-GPU和CPU状态分而治之

分离GPU侧和GPU侧状态，分而治之



解决方案-拦截runtime API or driver API

Intecept runtime API



Intercept driver API

pros：
1. 所有的API均是公开的，无私有API

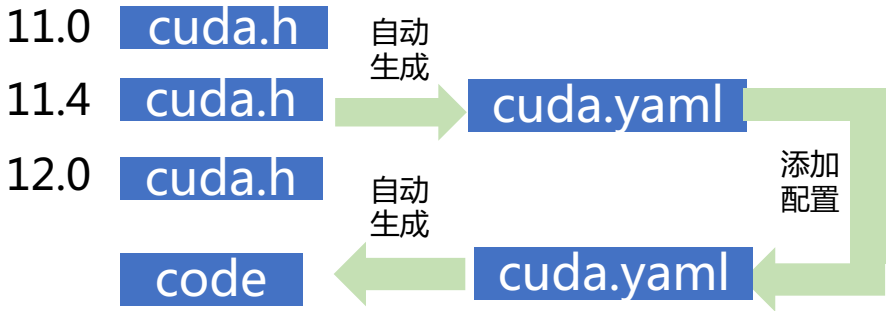
cons：
1. API数量大(包含driver API)
2. API由用户的镜像决定，且变化快
3. 所有程序均需要重新编译（libcudart）
4. 存在C/C++两种类型的API

pros：
1. API稳定性高，和NVIDIA内核模块绑定
2. API数量少, 总量大约400附近
3. 无需重新编译程序
4. 均为C类型API

cons：
1. cuGetExportTable
2. 大量未公开的API接口

生成redirect APIS-开放 & 封闭

开放部分：自动生成




```yaml
- !CudaFunc
name: cuMemcpyHtoD_v2
stage: 2
is_macro: false
record_api: false
multi_version: false
args:
- !FuncArg
  name: dstDevice
  type: CUdeviceptr
  rpc_type: ptr
- !FuncArg
  name: srcHost
  type: const void*
  rpc_type: mem_data
  rpc_type_binds: ByteCount
- !FuncArg
  name: ByteCount
  type: size_t
```

```yaml
- !CudaFunc
  name: cuMemAlloc_v2
  stage: 2
  is_macro: false
  has_return: true
  record_api: true
  multi_version: false
  args:
  - !FuncArg
    name: dptr
    type: CUdeviceptr*
    rpc_type: ptr_result
    rpc_type_binds: ''
    resource_type: ''
    resource_map: ''
  - !FuncArg
    name: bytesize
    type: size_t
```

cuda.yaml内容示例

封闭部分：逆向工程

Libcudart, libblas, libcudnn使用了大量的隐藏函数

1. 通过cuGetExportTable获取函数指针块
2. 需要时直接调用函数指针，不调用cuda暴露API

```
CUresult cuGetExportTable(const void **ppExportTable,
                          const CUuuid *pExportTableId);

static CUuuid uuids[] = {
    {0x6b, 0xd5, 0xfb, 0x6c, 0x5b, 0xf4, 0xe7, 0x4a, 0x89, 0x87, 0xd9, 0x39, 0x12, 0xfd, 0x9d, 0xf9},
    {0xa0, 0x94, 0x79, 0x8c, 0x2e, 0x74, 0x2e, 0x74, 0x93, 0xf2, 0x8, 0x0, 0x20, 0xc, 0xa, 0x66},
    {0x42, 0xd8, 0x5a, 0x81, 0x23, 0xf6, 0xcb, 0x47, 0x82, 0x98, 0xf6, 0xe7, 0x8a, 0x3a, 0xec, 0xdc},
    {0xc6, 0x93, 0x33, 0x6e, 0x11, 0x21, 0xdf, 0x11, 0xa8, 0xc3, 0x68, 0xf3, 0x55, 0xd8, 0x95, 0x93},
    {0xd4, 0x8, 0x20, 0x55, 0xbd, 0xe6, 0x70, 0x4b, 0x8d, 0x34, 0xba, 0x12, 0x3c, 0x66, 0xe1, 0xf2},
    {...},
}
```

解决方案-log/replay哪些API

无需Log/replay的API/状态

- 寄存器等临时的状态
  - 使用cudaDeviceSynchronize排空活动kernel
  - 迁移时无活动kernel，可以只迁移显存资源
- 过程类API
  - cuLaunchKernel
  - cuMemcpy.*

需要Log/replay的API

- 资源的创建/删除API
  - cuCtxCreate/cuCtxDestroy
  - cuMemAlloc/cuMemFree
  - …
- 初始化API/状态改变API
  - cuInit
  - cuCtxSetCurrent
  - …

解决方案-log/replay 变与不变

Log/replay的核心点对CUDA资源API进行分类，区分可变、不可变的资源

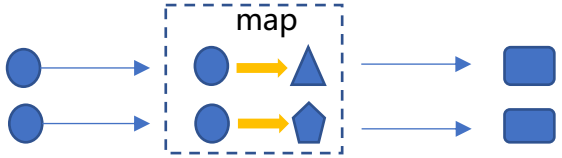
Opaque to Application

资源：

Context	Module
Stream	Function
Event	

- 应用无需理解这些资源
- CUDA CALL中作为参数传递到CUDA LIBS

应对策略：replay时做一次映射



Visible to Application

资源：

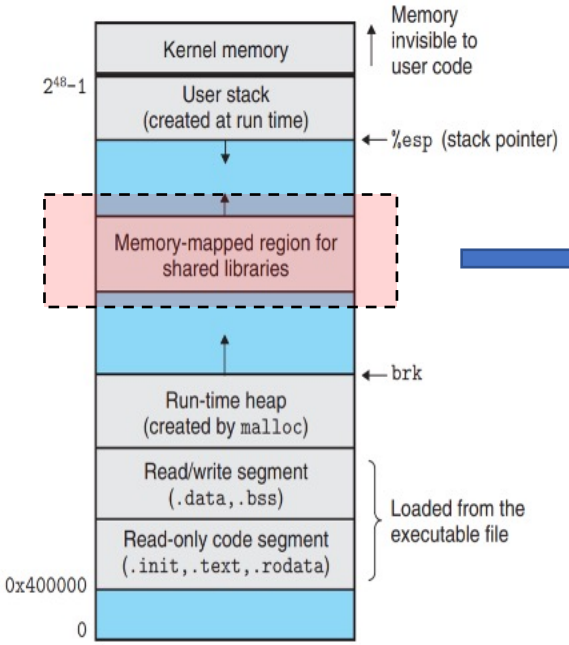
CUdeviceptr

- PyTorch/TensorFlow自己管理显存
  - 启动时分配显存，并池化管理
  - Memcpy/launchkenel传递部分内存，对地址++--
  - 无法保证replay后地址冲突问题，无法做映射

Replay时需要保持显存地址不变

如果保证replay显存地址不变

需要在replay后重建CUDA proxy的地址布局



Process memory layout

Big-size malloc

File based mmap

Anonymous mmap

CUDA related mapping
- ✓ malloc called by cuda libs
- ✓ GPU memory mmap

影响mmap区域布局的操作

Disable ASLR

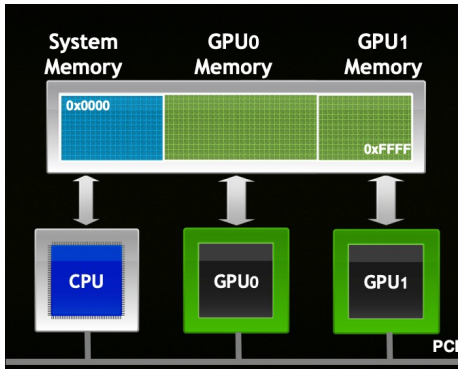
log/replay mmaps in order

如何log/replay

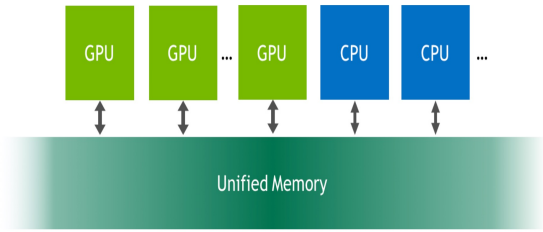
显/内存地址布局：从分离到统一

两种使用显存的方法

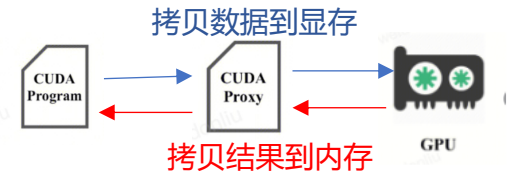
UVA
Unified Virtual Address



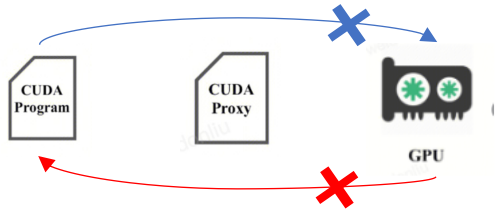
UVM
Unified Memory



1. cuMemAlloc/cuMemCpy.*



2. CUDA程序直接访问显存



CUDA程序无法直接访问显存

cudaHostAlloc获取Pinned memory

1. 加速CPU/GPU和GPU p2p数据传输 OK
2. ZERO COPY：GPU直接访问CPU侧内存地址 NOT OK
   1. 按需传输内存，性能损失巨大(10倍性能差距)
   2. PyTorch/TensorFlow均未使用

cudaMallocManaged获取unified memor

1. 分配的内存不指定位置
2. 访问地址触发PF，页粒度同步内存/显存
3. 应用无法控制数据位置，显存用尽时性能大幅下降
4. PyTorch/TensorFlow框架均未使用。

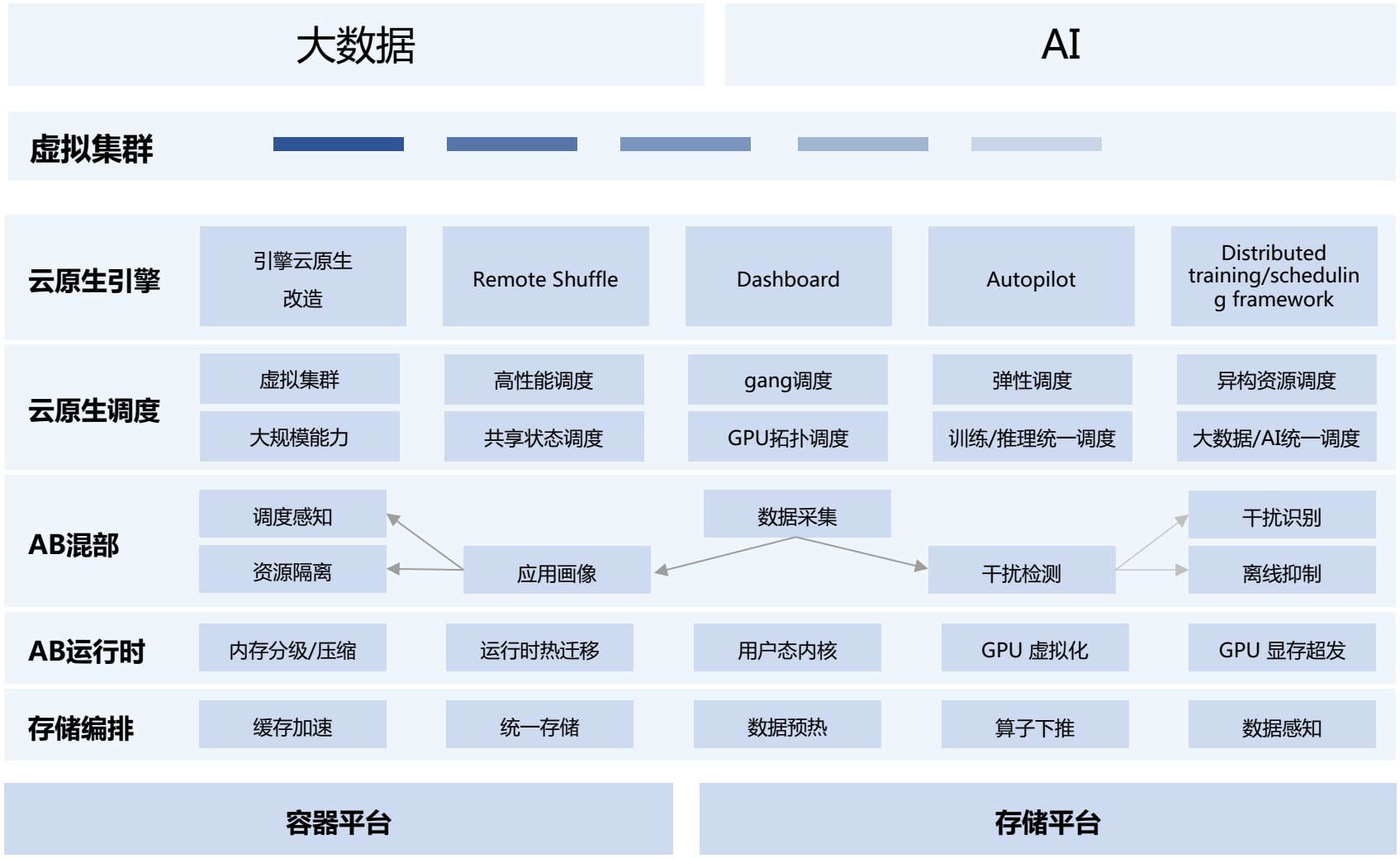
DEMO

进展：成功运行了https://github.com/pytorch/benchmark的benchmark

测试用例	本地执行(s)	RPC(s)	Fatbinary数	Kernel数
test_pytorch_CycleGAN_and_pix2pix_train_cuda	41.508	53.361s(++28%)	1064	43249
test_BERT_pytorch_train_cuda	6.639	14.771s(+122%)	820	34362

Demo: BERT 从节点1迁移到节点2

关于我们--峰峦云原生 AB 统一底座



AB云原生统一底座

复用峰峦作为云原生大数据底座的基础设施和能力

1. 大数据运行时扩展为AB运行时
2. 大规模集群能力
3. 弹性调度能力
4. 高可用能力
5. 虚拟集群能力

延续和增强GPU虚拟化能力，并加入更多底层能力，提升单机弹性调度能力

1. 时分复用+空分复用
2. 引入显存隔离、增强算力隔离能力
3. 单机节点资源的弹性管理

AI场景下的定制化能力

1. 利用空闲资源加速
2. 拓扑感知能力，包括GPU拓扑、网络拓扑等
3. GPU混部、GPU/CPU混部、潮汐调度
4. 自研AI调度器