Paddle 中动态图调用流程、Alloctor与显存管理

目录

动态图调用流程

以动态图场景下 matmul 为例子，下面展示了 Python API -> Launch Kernel 的调用流程。

</> Python

```
1 # API 位置
```

```
# python/paddle/tensor/linalg.py:176
def matmull(x, y, transpose_x=False, transpose_y=False, name=None):
    # 调用 _C_ops.matmul()

# OPS 调用位置，这里是动静统一接口
# paddle/fluid/pybind/ops_api.cc
static PyObject *matmul(PyObject *self, PyObject *args, PyObject *kwargs)
{
  if (egr::Controller::Instance().GetCurrentTracer() == nullptr) {
    VLOG(6) << "Call static_api_matmul";
    return static_api_matmul(self, args, kwargs); # 静态图 api
  } else {
    VLOG(6) << "Call eager_api_matmul";
    return eager_api_matmul(self, args, kwargs);  # 动态图 api
  }
}

# 动态图 op_func
# paddle/fluid/pybind/eager_op_function.cc
PyObject * eager_api_matmul(PyObject *self, PyObject *args, PyObject
*kwargs) {
    ...
    // Call dygraph function
    decltype(::matmul_ad_func(x,y,transpose_x,transpose_y)) ad_func_out =
::matmul_ad_func(x,y,transpose_x,transpose_y);

# 动态图前向 API 调用
#
paddle/fluid/eager/api/generated/eager_generated/forwards/dygraph_functio
ns.cc
TEST_API paddle::Tensor matmul_ad_func(const paddle::Tensor& x, const
paddle::Tensor& y, bool transpose_x, bool transpose_y) {
    ...
    // Forward API Call
    auto api_result = paddle::experimental::matmul(x_tmp, y_tmp,
transpose_x, transpose_y);

# PHI 的上层接口
# paddle/phi/api/include/api.h
PADDLE_API Tensor matmul(const Tensor& x, const Tensor& y, bool
transpose_x = false, bool transpose_y = false);

# PHI 算子 launch
# paddle/phi/api/lib/api.cc
```

```
PADDLE_API Tensor matmul(const Tensor& x, const Tensor& y, bool transpose_x, bool transpose_y) {
    # 根据 layout、backend、data_type 选择 Kernel
    auto kernel_result = phi::KernelFactory::Instance().SelectKernelOrThrowError(
      "matmul", {kernel_backend, kernel_layout, kernel_data_type}, true);
     ...
    # 获取 device ctx 和 输入 tensor、准备输出 tensor
    auto* dev_ctx = GetDeviceContextByBackend(actual_kernel_backend);

    auto input_x = PrepareData(x, GetKernelInputArgDef(kernel.InputAt(0), actual_kernel_backend), {}, kernel_result.is_stride_kernel);
    auto input_y = PrepareData(y, GetKernelInputArgDef(kernel.InputAt(1), actual_kernel_backend), {}, kernel_result.is_stride_kernel);
    ...
    Tensor api_output;
    auto kernel_out = SetKernelOutput(&api_output);

    # 执行 Infer Meta
    phi::MatmulInferMeta(MakeMetaTensor(*input_x), MakeMetaTensor(*input_y), transpose_x, transpose_y, &meta_out);

    # kernel launch
    (*kernel_fn)(*dev_ctx, *input_x, *input_y, transpose_x, transpose_y, kernel_out);

```

在调到 C++ 层 API 时会调用 PrepareData() 准备输入数据，这里会将输入 tensor Cast 成 DenseTensor，但是看起来并没有直接调用 Allocator。

```
</>                                                                    C++

// paddle/phi/api/lib/data_transform.cc:430
paddle::optional<phi::DenseTensor> PrepareData(
    const paddle::optional<Tensor>& input,
    const phi::TensorArgDef& target_args_def,
    const TransformFlag& transform_flag,
    bool is_stride_kernel) {
  if (input) {
    return {*PrepareData(
        *input, target_args_def, transform_flag, is_stride_kernel)};
  }
  return paddle::none;
}
```

```
13
```

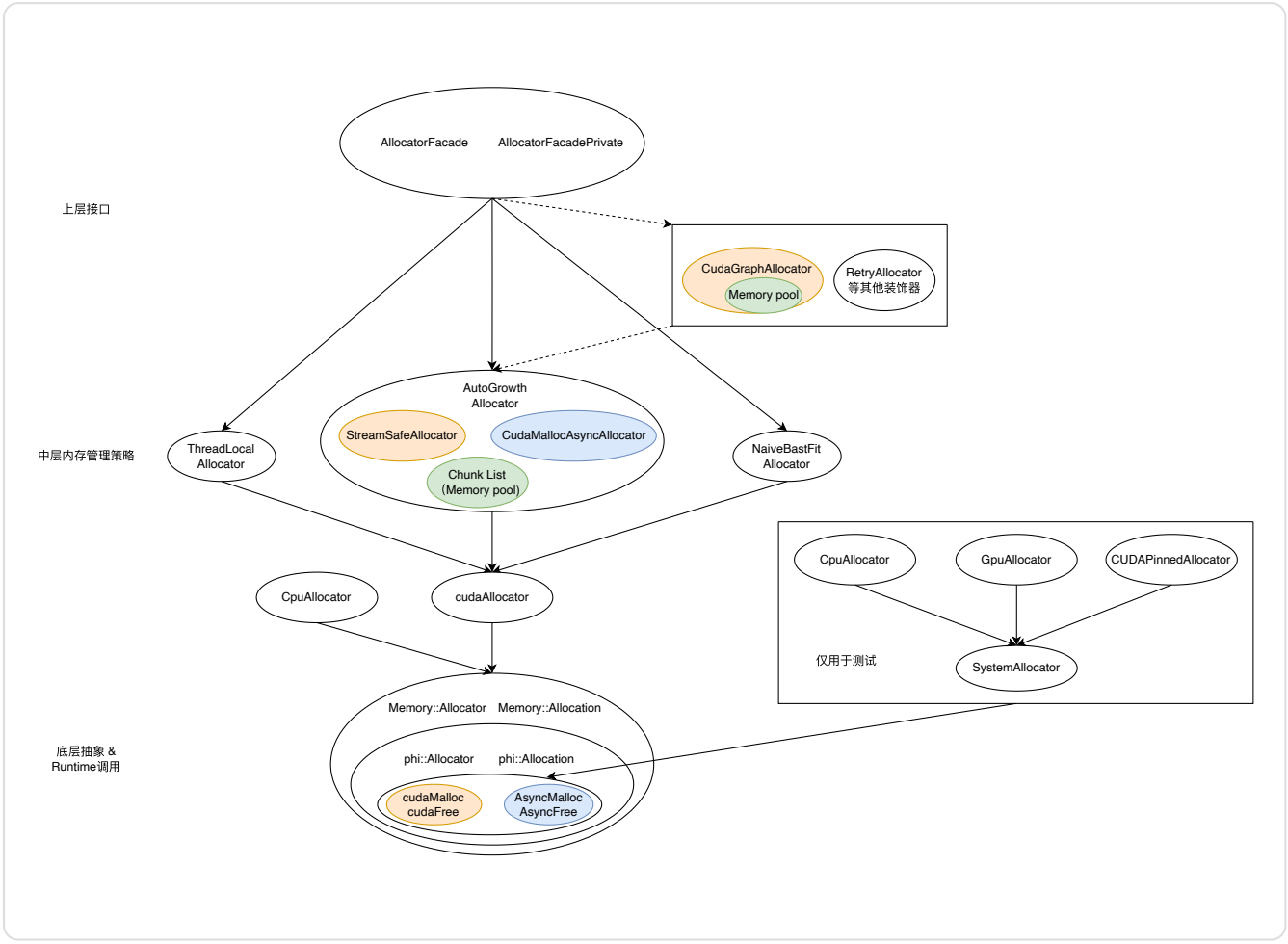
输出的显存最终在 Launch 的 Kernel 中通过 `dev_ctx.template Alloc<T>(out)` 创建，后续会经过一系列 Device contex、Stream 相关的查询，找到能正确使用的 Allocator，调用 AllocatorFacade，完成内存分配。

</> C++

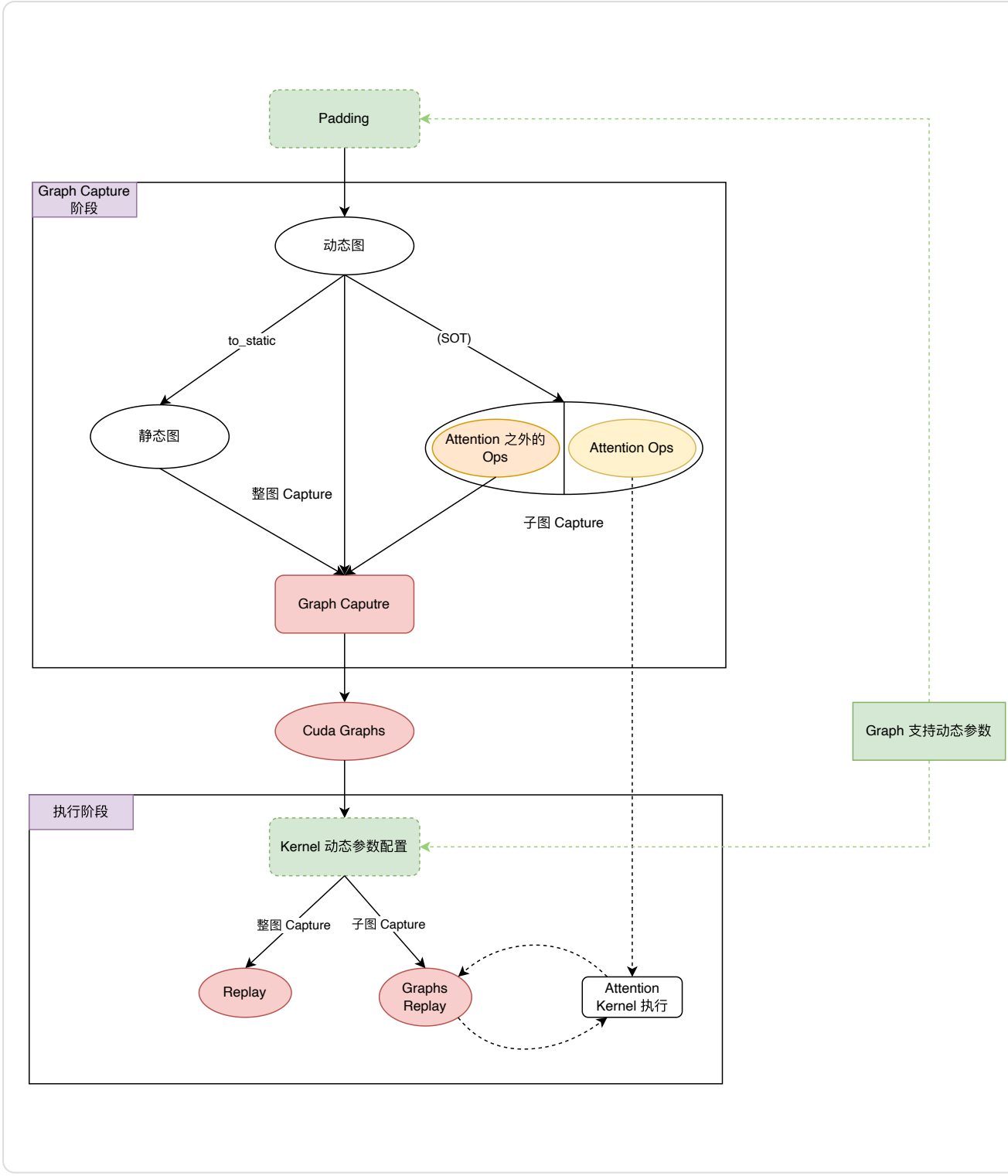
```cpp
// paddle/phi/backends/gpu/gpu_context.h
class PADDLE_API GPUContext : public DeviceContext,
                              public TypeInfoTraits<DeviceContext,
GPUContext> {}

// paddle/phi/core/device_context.h
class PADDLE_API DeviceContext {
    virtual void* Alloc(TensorBase*,
                        DataType dtype,
                        size_t requested_size = 0,
                        bool pinned = false,
                        bool fake_alloc = false) const;
    template <typename T>
    TEST_API T* Alloc(TensorBase* tensor,
                      size_t requested_size = 0,
                      bool pinned = false) const;
    ConstructDevCtx
    template <typename DevCtx>
    inline std::unique_ptr<DeviceContext> CreateDeviceContext(
        const phi::Place& p,
        bool disable_setting_default_stream_for_allocator,
        int stream_priority) {
     // 构造 DeviceCtx时构造
    }

}
// 回到 Densor tensor
class TEST_API DenseTensor : public TensorBase,
                             public TypeInfoTraits<TensorBase,
DenseTensor> {
    void* AllocateFrom(Allocator* allocator,
                       DataType dtype,
                       size_t requested_size = 0,
                       bool fake_alloc = false) override;
}
```

Paddle Allocator 架构梳理

Allocator 整体架构：



CUDA Graph 路径实现：



1. Device 显存分配

Paddle 将 GUP 显存分配和相关的参数配置封装在了 `paddle::platform::RecordedGpuMallocHelper` 中。这里只看GPU相关代码，实现了 `Malloc` 、 `Free` 和 `MallocAsync` 、 `FreeAsync` 两套逻辑，其中 `Async` 的两个接口是新增的，基本没被业务使用验证过。一般情况下 Naive 的 Allocator 会更快，因为它一次性开辟的大块显存，后续完全由框架做内存管理，不需要与 CUDA 交互。而 `Async` 的接口在

Stream级别有更细粒度的控制，理论上在特定场景下会有更高的性能和显存利用率、更少的碎片。

Sync接口	Cuda Runtime API	
gpuError_t Malloc(void **ptr, size_t size, bool malloc_managed_memory = false)	cudaMalloc(ptr, size) 和 cudaMallocManaged(ptr, size)	gpuError_t
void Free(void *ptr, size_t size)	cudaFree(ptr)	void FreeAsync(v

下面简单对比下两个接口的差异，Sync Malloc 除了要传指针和大小之外，还需要传一个 `malloc_managed_memory` bool 值，用来控制是否使用动态统一内存编址来管理内存。

MllocAsync 接口支持了 Stream-Ordered Allocator，需要在这里传入需要开辟的 Stream 参数或指定内存池。需要注意的是这里会忽略当前的 Device context，驱动程序会默认使用 Stream 所在的 Device context。

</> C++

```
// paddle/phi/core/platform/device/gpu/gpu_info.cc

class RecordedGpuMallocHelper {
 public:
   // Malloc
   gpuError_t Malloc(void **ptr,
                     size_t size,
                     bool malloc_managed_memory = false) {
     ...
         phi::backends::gpu::CUDAGraphCaptureModeGuard
capture_mode_guard;
        if (UNLIKELY(malloc_managed_memory)) {
          result = cudaMallocManaged(ptr, size);
        } else {
          result = cudaMalloc(ptr, size);
        }
     ...
   }

   // MallocAsync
   gpuError_t MallocAsync(void **ptr, size_t size, gpuStream_t stream) {
       PADDLE_ENFORCE_GPU_SUCCESS(
          cudaDeviceGetDefaultMemPool(&memPool_, dev_id_));  // 使用的是
Device 的 Default memory pool
       ...
       gpuError_t result;
```

```
25          result = cudaMallocAsync(ptr, size, stream);
26          ...
27      }
28
29      // Free
30      void Free(void *ptr, size_t size) {
31          ...
32          auto err = cudaFree(ptr);
33          ...
34      }
35
36      // Async Free
37      void FreeAsync(void *ptr, size_t size, gpuStream_t stream) {
38          ...
39          auto err = cudaFreeAsync(ptr, stream);
40          ...
41       }
```

所有的 Malloc & Free 封装在 RecordedGpuMallocHelper 内，并通过 Instance 出来的 单例调用各接口。

</> C++

```
1  gpuError_t RecordedGpuMalloc(void **ptr, size_t size, int dev_id, bool
   malloc_managed_memory) {
2    return RecordedGpuMallocHelper::Instance(dev_id)->Malloc(
3        ptr, size, malloc_managed_memory);
4  }
5
6  void RecordedGpuFree(void *p, size_t size, int dev_id) {
7    return RecordedGpuMallocHelper::Instance(dev_id)->Free(p, size);
8  }
9
10 gpuError_t RecordedGpuMallocAsync(void **ptr, size_t size, int dev_id,
   gpuStream_t stream) {
11   return RecordedGpuMallocHelper::Instance(dev_id)->MallocAsync(
12       ptr, size, stream);
13 }
14
15 void RecordedGpuFreeAsync(void *p, size_t size, int dev_id, gpuStream_t
   stream) {
16   return RecordedGpuMallocHelper::Instance(dev_id)->FreeAsync(p, size,
   stream);
17 }
```

2. 基础 Allocator

Paddle 为了兼容多硬件和多种显存分配策略， 对 Alloctor 进行了多层抽象。最底层定义在 `phi namespace` 下，有两个类 `Allocation` 和 `Allocator` 。其中 `Allocation` 表示已经分配的一块内存，封装了内存块的地址、大小和释放逻辑。 `Allocation` 由 `Allocator` 创建，本质上是一个带有 `Deleter` 的智能指针，只记录内存地址并不直接持有资源。

`Allocator` 是内存分配策略的最底层抽象接口，定义如何分配和释放内存的纯虚接口。

总结： `Allocator` 通过 `Allocate()` 方法创建 `Allocation` 对象， `Allocation` 通过 `deleter_` 实现内存释放，两者共同协作完成内存的生命周期管理。

</> C++

```cpp
// paddle/phi/core/allocator.h

class Allocation { // final
 public:
  using Place = phi::Place;
  using DeleterFnPtr = void (*)(Allocation*);

  Allocation() = default;
  Allocation(void* data, size_t size, const Place& place)
      : ptr_(data), size_(size), place_(place) {}
  Allocation(void* data, size_t size, DeleterFnPtr deleter, const Place& place)
      : ptr_(data), size_(size), deleter_(deleter), place_(place) {}

  Allocation(Allocation&& other) noexcept { swap(*this, other); } // 支持移动语义
  Allocation& operator=(Allocation&& other) noexcept {
    swap(*this, other);
    return *this;
  }

  virtual ~Allocation() {
    if (deleter_) {
      deleter_(this);
    }
  }

  void* ptr() const noexcept { return ptr_; }

```

```
  // Returns the size of this memory buffer, i.e., ptr() + size() - 1 is the last valid element.
  size_t size() const noexcept { return size_; }

  void* operator->() const noexcept { return ptr_; }
  operator bool() const noexcept { return ptr_; }
  const Place& place() const noexcept { return place_; }
  DeleterFnPtr deleter() const noexcept { return deleter_; }

 protected:
  friend void swap(Allocation& a, Allocation& b) noexcept;
  void* ptr_{nullptr};
  size_t size_{};
  DeleterFnPtr deleter_{nullptr};
  Place place_;
};

class Allocator {
 public:
  using DeleterType = std::function<void(Allocation*)>;   // Allocation Deleter
  using AllocationPtr = std::unique_ptr<Allocation, DeleterType>;

  virtual ~Allocator() = default;
  virtual AllocationPtr Allocate(size_t bytes_size) = 0;  // 开辟的内存由 Allocation 记录

  virtual bool IsAllocThreadSafe() const { return false; } // 是否是线程安全的
};
```

`phi::Allocation` 和 `phi::Allocator` 又派生出了 `paddle::memory::allocation::Allocator` 和 `paddle::memory::allocation::Allocation`，后续所有的内存管理策略都是在这两个类的基础上完成的。这里还引入了 Alloctor 装饰器的概念，用来在基础功能上实现灵活的内存分配策略，Paddle 中是通过继承和组合实现的，因此在进行内存分配时要记录下装饰器链，释放时逆序释放。

这里还定义了实现内存对齐、预填充、static_cast 等工具方法

</> C++

```
// paddle/phi/core/memory/allocation/allocator.h

class Allocation : public phi::Allocation {
```

```
 public:
  Allocation(void* ptr, size_t size, phi::Place place)
      : phi::Allocation(ptr, size, place), base_ptr_(ptr) {}
  Allocation(void* ptr, void* base_ptr, size_t size, const phi::Place&
place)
      : phi::Allocation(ptr, size, place), base_ptr_(base_ptr) {}

  void* base_ptr() const { return base_ptr_; }

 private:
  inline void RegisterDecoratedAllocator(Allocator* allocator) {  // 插入
Allocator 到装饰器栈顶
    decorated_allocators_.emplace_back(allocator);
  }
  inline void PopDecoratedAllocator() { decorated_allocators_.pop_back();
} // 逆序释放时使用
  inline Allocator* TopDecoratedAllocator() {
    return decorated_allocators_.back();
  }

 private:
  void* base_ptr_;  // the point that directly requested from system

  static constexpr size_t kReserveAllocatorNum = 8;
  using DecoratedAllocatorStack =
      framework::InlinedVector<Allocator*, kReserveAllocatorNum>;
  DecoratedAllocatorStack decorated_allocators_;

  friend class Allocator;
};


// Base interface class of memory Allocator.
class Allocator : public phi::Allocator {
 public:
  static void AllocationDeleter(phi::Allocation* allocation) {
    Allocator* allocator =
        static_cast<Allocation*>(allocation)->TopDecoratedAllocator();
    allocator->Free(allocation);
  }

  // Allocate an allocation.
  // size may be 0, but it would be too complex if we handle size == 0
```

```
  // in each Allocator. So we handle size == 0 inside AllocatorFacade
  // in our design.
  AllocationPtr Allocate(size_t size) override {
// 内存开辟入口，返回显存块
    auto ptr = AllocateImpl(size);
    static_cast<Allocation*>(ptr)->RegisterDecoratedAllocator(this);
// 当前 Alloctor 入栈
    return FillValue(AllocationPtr(ptr, AllocationDeleter));
// 调用 memset 预填充 (Paddle这里调用的是cudaMemset而非Async)
  }

  void Free(phi::Allocation* allocation) {
    static_cast<Allocation*>(allocation)->PopDecoratedAllocator();
// 逆序出栈释放内存
    FreeImpl(allocation);
  }

  uint64_t Release(const phi::Place& place) { return ReleaseImpl(place);
}

 protected:
  virtual phi::Allocation* AllocateImpl(size_t size) = 0;
  virtual void FreeImpl(phi::Allocation* allocation);
  virtual uint64_t ReleaseImpl(const phi::Place& place UNUSED) { return
0; }
};
```

3. 不同内/显存管理策略

AllocatorStrategy 中有三种分配策略：

- NaiveBestFit
- AutoGrowth
- ThreadLocal

分配策略可以通过 FLAG 配置，能够控制底层 Allocator 的选择。

</> C++

```
// paddle/phi/core/memory/allocation/allocator_strategy.cc
static AllocatorStrategy GetStrategyFromFlag() {
  if (FLAGS_allocator_strategy == "naive_best_fit") {
    return AllocatorStrategy::kNaiveBestFit;
  }
  if (FLAGS_allocator_strategy == "auto_growth") {
```

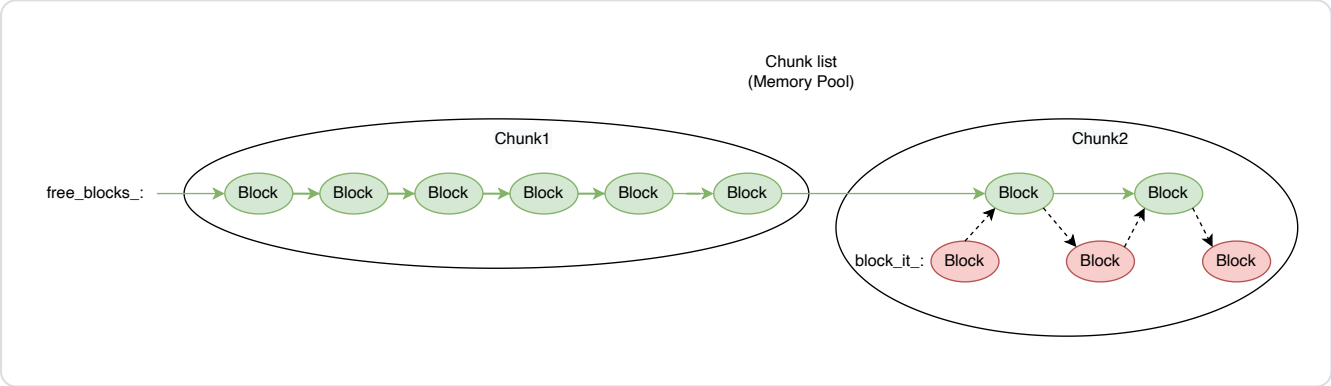
```
    return AllocatorStrategy::kAutoGrowth;
  }
  if (FLAGS_allocator_strategy == "thread_local") {
    return AllocatorStrategy::kThreadLocal;
  }
  PADDLE_THROW(...);
}
```

下面重点看 AutoGrowth 策略，其他分配策略都不支持多流。 auto_growth 策略下能够通过 Flag 选择 StreamSafeAllocator 和 CudaMallocAsyncAllocator 这两个重要 Allocator。

3.1 AutoGrowth V1(V2 略)

V2主要用来解决显存碎片问题，未开发完，这里略过。理论上 MallocAsync 中使用了 Cuda 的显存池，其中的碎片重映射功能解决部分碎片问题。

3.1.1 AutoGrowth 策略中的显存池



AutoGrowthBestFitAllocator 中引入了 BlockAllocation、Chunk、Block 概念，其中 BlockAllocation 是对 Block 的一层包装，Chunk 表示一块儿较大的内存，Block 则表示一小块儿内存。Chunk 内部会维护一个列表，记录 Chunk 中的所有 Block，如图中 Chunk2 中的虚线链表所示。在 AutoGrowth Allocator 中会维护一个 Chunk list，记录 Allocator 管理的所有 Allocation，同时还会维护一个 Block list，用来管理所有的能够使用的 Block。

AutoGrowth Allocator 中的显存池就由这些数据结构实现， Chunk size的配置要在执行 `InitAutoGrowthCUDAAllocator(place, stream)` 初始化时才会根据 Flag 配置。

</> C++

```
protected:
 struct Block {
   Block(void *ptr, size_t size, bool is_free, Chunk *chunk)
       : ptr_(ptr), size_(size), is_free_(is_free), chunk_(chunk) {}

   void *ptr_;
   size_t size_;
```

```
    bool is_free_;
    Chunk *chunk_;  // which chunk it is from
  };

  struct Chunk {
    explicit Chunk(DecoratedAllocationPtr allocation)
        : allocation_(std::move(allocation)) {}

    DecoratedAllocationPtr allocation_;
    List<Block> blocks_;
  };

  struct BlockAllocation : public Allocation {
    explicit BlockAllocation(const List<Block>::iterator &it)
        : Allocation(it->ptr_,
                     it->chunk_->allocation_->base_ptr(),
                     it->size_,
                     it->chunk_->allocation_->place()),
          block_it_(it) {}

    List<Block>::iterator block_it_;
  };

  using BlockIt = List<Block>::iterator;

  std::shared_ptr<Allocator> underlying_allocator_;
  std::map<std::pair<size_t, void *>, BlockIt> free_blocks_;
  std::list<Chunk> chunks_;
```

3.1.2 AutoGrowthBestFitAllocator

Allocate 时首先会尝试从 free block 链表中寻找能够满足分配请求的block，找不到时会尝试开辟一块儿新的Chunk，并加入到显存池的管理中。

</> C++

```
phi::Allocation *AutoGrowthBestFitAllocator::AllocateImpl(
    size_t unaligned_size) {
  // event record
  size_t size = AlignedSize(unaligned_size + extra_padding_size_,
alignment_);   // 内存对齐后的 size

  std::lock_guard<SpinLock> guard(spinlock_);
```

```
  auto iter = free_blocks_.lower_bound(std::make_pair(size, nullptr)); // 寻找最小的能满足分配请求的 block
  BlockIt block_it;
  if (iter != free_blocks_.end()) {  // 显存池能满足
    1. free_blocks_ 中删除被使用的 block iterator
    2. 计算满足分配需求后剩余的显存块大小 remaining_size
    3. remaining_size 大于 0 时会使用剩余的显存块创建一个新的 block 并插入 free_blocks_和所在Chunk的 block list 中
  } else {  // 显存池不能满足，开辟(Malloc)新的显存
    1. 尝试释放所有 Chunk 中空闲的 block，被 FLAGS_free_when_no_cache_hit 控制
    2. 计算需要新开辟的显存的大小，最小开辟一个新 chunk：std::max(size, chunk_size_)
    3. 直接调用底层 Allocator 的 Allocate 接口开辟新 Chunk，并添加到 Chunk list 中
        - BadAlloc 时尝试释放空闲 Chunk 后再次 Allocate，失败会直接报错
    4. 计算满足分配需求后剩余的显存块大小 remaining_size
    5. remaining_size 大于 0 时会使用剩余的显存块创建一个新的 block 并插入 free_blocks_和所在Chunk的 block list 中
  }
  // 更新管理的 memory size
  return new BlockAllocation(block_it);  // 使用 block 的 iterator 构造 BlockAllocation 返回
}
```

Free 当前的 Block 时，会查看同一个 Chunk 中当前 Block 的前一个和后一个 Block 是不是 Free 的，然后尝试将当前 Block 与前后 Block 进行合并，以减少碎片。理论上当一个 Chunk 中所有的显存都被回收时，Chunk 中有且只有一个block，且状态会是 Free，这时如果调用了 `FreeIdleChunks()` 就会将整个 Chunk 释放。

</> C++

```
void AutoGrowthBestFitAllocator::FreeImpl(phi::Allocation *allocation) {
  // event record
  std::lock_guard<SpinLock> guard(spinlock_);
  1. 获取 Allocation/Block 所在的 Chunk
  2. 尝试与 Chunk 中前一个 Block 合并
  3. 尝试与 Chunk 中后一个 Block 合并
  4. 当前 Block 插入 free_blocks_
  5. 析构 Allocation
  6. 尝试释放空闲的 Chunk，由 FLAGS_free_idle_chunk 控制
}
```

3.2 NaiveBestFit

略

3.3 ThreadLocal

略

4. 几个重要 Allocator

4.1 StreamSafeAllocator

StreamSafeAllocator 的最大特点是实现了不同 Stream 之间安全共享显存块。

4.1.1 Allocation

StreamSafeAllocator 同样有一个配套的显存管理类 StreamSafeCUDAAllocation，StreamSafeCUDAAllocation 中会记录该显存块所属的 Stream 以及需要使用该显存块的其他 Stream。

- 借用登记是通过 RecordStream 实现的，会在记录的 Stream 上创建并插入一个 CUDA Event，并维护到 map 中。
- 在该显存块回收时，需要保证所有使用该显存块的 Stream 都已经用完，即记录在 Map 中的所有 Event 都已经完成。这会通过 Allocation的CanBeFreed 接口进行判断。

另外 StreamSafeCUDAAllocation 还实现了 `RecordStreamWithNoGraphCapturing()` 接口，用来记录处在 Capture 状态的 Stream，暂时还不知道在哪里会被用到。

```C++
void RecordGraphCapturingStreams();
void RecordStreamWithNoGraphCapturing(gpuStream_t stream);

 std::set<gpuStream_t> graph_capturing_stream_set_;
```

4.1.2 Allocator

`StreamSafeCUDAAllocator` 在构造时会传入一个 `in_cuda_graph_capturing` 的 bool 值用来标识是否处于 Cuda Graph Capture状态。向上看在 AllocatorFacade 中这个bool 值实际上是 `allow_free_idle_chunk` （默认是 true），向下看在 StreamSafeAllocator 内这个bool值最终会影响显存的 Release，在 Caputre 期间不会 Release 显存。也就是说 StreamSafeAllocator 对 CUDA Graph 的支持只有延迟释放显存。

下面是 `StreamSafeCUDAAllocator` 的内存申请和释放逻辑：

```C++
```

```
phi::Allocation* StreamSafeCUDAAllocator::AllocateImpl(size_t size) {
  // record event
  1. 尝试释放unfreed_allocations_中处于暂缓释放状态的显存块
  2. 调用底层 Allocator (AutoGrowth) 开辟显存
      + BadAlloc: 释放 Device 上的空闲内存后，重新尝试开辟
  3. 构造 StreamSafeCUDAAllocation，并使其与当前 Allocator 所在的 Device 的 Default stream 绑定
}

void StreamSafeCUDAAllocator::FreeImpl(phi::Allocation* allocation) {
  // record event
  1. 调用 CanBeFreed() 判断 allocation 能否释放
      + 能没有依赖，直接释放
      + 有 Event 依赖的情况下，先暂存入 unfreed_allocations_延缓释放
}

uint64_t StreamSafeCUDAAllocator::ReleaseImpl(const phi::Place& place) {
  // record event
  1. Allocator 处于 Cuda Graph Capture 状态时不释放显存
  2. 释放 place 上所有的 StreamSafe Allocator 的可释放显存块
}
```

4.2 CudaMallocAsyncAllocator

4.2.1 Allocation

CudaMallocAsynAllocation 最大的特点是将所有的挂起等待释放的显存块儿全都放在了一个独立的 NonBlocking Stream 上去管理，这样能减少全局的 Event 依赖。并且该 free_stream_是通过 lazy 的模式创建的。

</> C++

```
size_t CUDAMallocAsyncAllocation::Free() {
   1. 无 Stream 借用，可释放状态
       + 直接释放
       + 将 malloc_stream 添加到 Graph 捕获列表
   2. 暂时无法释放
       + 同步 malloc_stream 和 free_stream
       + 同步所有借用 Stream 和 free_stream
       + 释放
       + 将 free_stream 添加到 Graph 捕获列表
}
```

4.2.2 Allocator

CudaMallocAsyncAllocator 在 CUDA Graph Capture 之前会先调用 `ClearFreeStream` 清理一下 free_stream 中所有的 chunk。同时设置了一个释放 Memory 节流阈值，hold 的显存超过阈值时会自动释放。同时由于 AsyncAllocator 使用的是 Cuda 的 memory pool，每次同步是也会释放空闲的显存，显存管理相对合理。

首先看下 `FreeAllocation()` 和 `AllocateImpl()` 的显存操作

```C++

void CUDAMallocAsyncAllocator::FreeAllocation(
    CUDAMallocAsyncAllocation* allocation) {
    1. 直接 Free 显存块
    2. 更新 pending_release_size_
}

phi::Allocation* CUDAMallocAsyncAllocator::AllocateImpl(size_t size) {
    LazyInitializeCudaFreeStream();
    MallocThrottling();
    // record event
    1. 尝试申请显存
    2. 开辟成功
       2.1 构造 allocation
       2.2 处于 Capture 状态
           + 记录 allocation 所属的 Graph
           + 设置回调函数，在 Graph 析构时再释放显存块
    3. 开辟失败
       + BadAlloc
}
```

CudaMallocAsyncAllocator 中还对 CUDA Graph 的显存块做了隔离，专门用来处理 Memory 申请、释放在 Graph Capture 内外的组合情况。

	Malloc 在 Graph 外	Malloc 在 Graph 内
Free 在 Graph 内	Capture 结束时释放	立即释放
Free 在 Graph 外	立即释放	Graph 析构时释放

```C++
void CUDAMallocAsyncAllocator::FreeImpl(phi::Allocation* phi_allocation)
{
    1. 处在 Caputure 状态
```

```
        1.1 判断 allocation 是否是在当前 Graph 中创建的
            + 不是：添加回调，capture 结束后释放
            + 是：  立即释放
    2. 没有处在 Capture 状态
        + 是当前 Graph 内创建的，直接释放
    3. 其他情况直接释放
}
```

4.3 CUDAGraphAllocator

CUDAGraphAllocator 是 CUDAGraph 专用的 Allocator，其有一个私有的显存池，用来隔离 Caputre 时的外部显存调用和与 CUDA Runtime 交互的同步操作（底层用的是 cudaMalloc 而非 Async 的接口）。

CUDAGraphAllocator 类中除了隔离的显存池，在显存开辟和释放接口上没有做特殊处理。

</> C++

```cpp
class CUDAGraphAllocator
    : public Allocator,
      public std::enable_shared_from_this<CUDAGraphAllocator> {
 private:

  class PrivateAllocation : public Allocation {  // Graph 私有的显存池
   public:
    PrivateAllocation(CUDAGraphAllocator* allocator,
                      DecoratedAllocationPtr underlying_allocation)
        : Allocation(underlying_allocation->ptr(),
                     underlying_allocation->base_ptr(),
                     underlying_allocation->size(),
                     underlying_allocation->place()),
          allocator_(allocator->shared_from_this()),
          underlying_allocation_(std::move(underlying_allocation)) {}

   private:
    std::shared_ptr<Allocator> allocator_;
    DecoratedAllocationPtr underlying_allocation_;
  };

  explicit CUDAGraphAllocator(std::shared_ptr<Allocator> allocator)  // 构造函数，underlying_allocator_用来实现装饰器
      : underlying_allocator_(std::move(allocator)) {}

 public:
```

```
  ~CUDAGraphAllocator() override = default;

  static std::shared_ptr<Allocator> Create(
      const std::shared_ptr<Allocator>& allocator) {
    return std::shared_ptr<Allocator>(new CUDAGraphAllocator(allocator));
  }

 protected:
  phi::Allocation* AllocateImpl(size_t size) override {     // 从私有的显存池中开辟显存
    VLOG(10) << "Allocate " << size << " for CUDA Graph";
    return new PrivateAllocation(this,
                                 static_unique_ptr_cast<Allocation>(
                                     underlying_allocator_->Allocate(size)));
  }

  void FreeImpl(phi::Allocation* allocation) override {
    VLOG(10) << "delete for CUDA Graph";
    delete allocation;
  }

 private:
  std::shared_ptr<Allocator> underlying_allocator_;
};
```

Paddle 将 Capture 期间相对敏感的 Release 操作没有放在 CUDAGraph 中，而是放在了底层的 StreamSafeAllocator 中。在 Cuda Graph 中不会 Release 显存来避免显存重用问题，这中策略会带来一定程度上的显存浪费和碎片问题。

```
</>                                                                    C++

uint64_t StreamSafeCUDAAllocator::ReleaseImpl(const phi::Place& place) {
  if (UNLIKELY(in_cuda_graph_capturing_)) {
    VLOG(7) << "Memory release forbidden in CUDA Graph Capturing";
    return 0;
  }
  ...
}
```

4.3 Allocator 的上层接口: AllocatorFacade

4.3.1 AllocatorFacadePrivate

Paddle底层没有显式的 Memory Pool 类，显存池的管理过 Allocation 及其子类实现，例如服务于 AutoGrowth 策略的 BlockAllocation。AllocatorFacadePrivate 承担了显存池的上层封装和一部分 Allocator 管理功能。AllocatorFacadePrivate 构造时会根据 Flag 以及配置的显存管理策略选择合适的 Allocator。构造函数如下：

```C++
explicit AllocatorFacadePrivate(bool allow_free_idle_chunk = true){
 // 初始化 StreamSafeAllocator/CudaMallocAsyncAllocator
 strategy_ = GetAllocatorStrategy();
 switch (strategy_) {
     case AllocatorStrategy::kNaiveBestFit: {
         InitNaiveBestFitCPUAllocator();
         for (int dev_id = 0; dev_id < platform::GetGPUDeviceCount();
++dev_id) { // 为每个 Device 创建一个 NBFC Allocator
             InitNaiveBestFitCUDAAllocator(phi::GPUPlace(dev_id));
         }
         InitNaiveBestFitCUDAPinnedAllocator();
         break;
     }
     case AllocatorStrategy::kAutoGrowth: {
         InitNaiveBestFitCPUAllocator();
         allow_free_idle_chunk_ = allow_free_idle_chunk;
         for (int dev_id = 0; dev_id < platform::GetGPUDeviceCount();
++dev_id) { // 为每个 Device 创建一个 Cuda Allocator
             InitAutoGrowthCUDAAllocator(phi::GPUPlace(dev_id),
allow_free_idle_chunk_); // 然后包装成 AutoGrowthBestFitAllocator 存放在
allocators_中
         }
         auto_growth_allocators_ = allocators_;

         if (FLAGS_use_cuda_malloc_async_allocator) {  // 根据 Flag 选择使
用的上层 Allocator 装饰器
             ...
             WrapCUDAMallocAsyncAllocatorForDefault();   // 使用 Async
Allocator
             is_cuda_malloc_async_allocator_used_ = true;
          } else {
         if (FLAGS_use_stream_safe_cuda_allocator) {
               if (LIKELY(!IsCUDAGraphCapturing())) {
                 WrapStreamSafeCUDAAllocatorForDefault();    // 使用
StreamSafe Allocator
               }
               is_stream_safe_cuda_allocator_used_ = true;
```

```
            }
        }

        InitNaiveBestFitCUDAPinnedAllocator();
        break;
    }
    case AllocatorStrategy::kThreadLocal: {
        InitNaiveBestFitCPUAllocator();
        for (int dev_id = 0; dev_id < platform::GetGPUDeviceCount();
++dev_id) {
            InitThreadLocalCUDAAllocator(phi::GPUPlace(dev_id));
        }
        InitNaiveBestFitCUDAPinnedAllocator();
        break;
    }
    default: { PADDLE_THROW(...)}
  } // switch

  InitZeroSizeAllocators();
  InitSystemAllocators();

  if (FLAGS_gpu_allocator_retry_time > 0) { // 如果设置了 retry，会再装饰一层
RetryAllocator
      WrapCUDARetryAllocator(FLAGS_gpu_allocator_retry_time);
  }
  // No need to wrap CUDAGraphAllocator for StreamSafeCUDAAllocator
  if (!is_stream_safe_cuda_allocator_used_ &&
UNLIKELY(IsCUDAGraphCapturing())) {  // 这里没看太明白，这么写会导致所有
Allocator 都包一层 CudaGraph 装饰器吗?
      WrapCUDAGraphAllocator();
   }
}
```

GetAllocator() 有两个重载，naive 的实现会直接根据传入的 place 返回 allocator_ 中对应 Device 的 Allocator 指针。带 stream 参数的 GetAllocator() 则会根据传入的 Stream 是否是 Device 的默认流判断是否需要创建一个新的 Allocator。

这么做主要的目的是为了隔离 Default Stream 与用户创建的自定义 Stream，使 Default Stream 相关的 Allocator 在初始化阶段就创建，用户自定义的 Allocator 延迟到 Get 使再创建，从而减少获取 Default Stream 和延迟创建 Allocator 的读写锁开销。这一收益是建立在用户未进行复杂的跨 Stream 显存分配操作的基础上的，**在多 Stream 支持完备的或面临更复杂的业务需求时可能要重新设计。**

```
</>                                                              C++
```

```
  const std::shared_ptr<Allocator>& GetAllocator(const phi::GPUPlace&
place, gpuStream_t stream, bool create_if_not_found = false) {
    if (LIKELY(!IsCUDAGraphCapturing())) {
      if (stream == GetDefaultStream(place)) {  // default stream 直接返回
预先创建的allocator_中的Allocator
        return GetAllocator(place, /* A non-zero num to choose allocator_
*/ 1);
      }
    }

    /* shared_lock_guard */ {
      std::shared_lock<std::shared_timed_mutex> lock_guard(
          cuda_allocator_mutex_);
      if (LIKELY(HasCUDAAllocator(place, stream))) {
        return cuda_allocators_[place][stream];       //
cuda_allocators_: (place -> map(stream -> Allocator)
      } else {
        PADDLE_ENFORCE_NE(create_if_not_found, false, ...);
      }
    }

    /* unique_lock_guard */ {
      std::unique_lock<std::shared_timed_mutex> lock_guard(   // 创建新
Allocator 的代码
          cuda_allocator_mutex_);
      InitCUDAAllocator(place, stream);
      return cuda_allocators_[place][stream];
    }
  }
```

因此在上层的数据结构中也将 Default Stream 和用户自定义 Stream 做了分离：

```
C++
allocators_ 存储默认流相关 ALlocator，初始化时就创建
cuda_allocators_ 存储非默认流相关 Allocator，Lazy 创建
```

另外一系列的 `InitxxxAllocator` 和 `WrapxxxAllocator` func 中分别封装了各类 Allocator 的初始化配置和装饰器方法。

4.3.2 AllocatorFacade

是所有 Allocator、 Allocation 和 AllocatorFacadePrivate 的最上层接口。

AllocatorFacade 中的重要 Func：

- GetAllocator() & GetAllocator(stream)：获取/创建实际使用的 Allocator。
- GetAutoGrowthAllocator()：
- GetZeroAllocator()：？
- GetBasePtr()：获取 AllocatorFacadePrivate 持有的 BasePtr
- GetPrivate()：返回/创建 AllocatorFacade 持有的 AllocatorFacadePrivate 显存池对象/CudaGraph显存池。当使用的不是 CudaMallocAsyncAllocator 且不在 Cuda Graph Capturing 期间时，会返回 Cuda Graph 的显存池，CudaMallocAsyncAllocator 不需要为每个 Graph 开启一个 Paddle 显存池，使用的是 CUDA 提供的 Graph memory pool。

- Release() & Release(stream)： 直接调用封装的 Allocator 的 Release()
- Alloc() & Alloc(stream)：直接调用封装的 Allocator 的 Allocate()
- AllocShared() & AllocShared(stream)：内部调用 Alloc
- EraseStream()：删除借用显存池的 Stream
- RecordStream()： 记录借用显存池的 Stream
- void AllocatorFacade::PrepareMemoryPoolForCUDAGraph(int64_t id)；将ID为 id 的显存池分配给 Cuda Graph
- void AllocatorFacade::RemoveMemoryPoolOfCUDAGraph(int64_t id)；回收 id 号显存池，引用计数为0的话释放显存池

上述两个 Graph 相关的显存池管理接口并没有设置 Graph 参数，是默认只会有一个 CUDA Graph 吗？CUDA Grpah 和 Allocator 的关系是怎么样的?

  - Paddle 中的显存管理与这里的显存池是分离的，cuda graph allocator 的显存池会封装底层的显存池，目的是为了隔离显存池，每个 allocator 会有一个对应的显存池。

AllocatorFacade 中的重要成员变量：

- AllocatorFacadePrivate* m_； 可以理解为持有的显存池
- std::unordered_map<int64_t, std::unique_ptr<AllocatorFacadePrivate>> cuda_graph_map_；字典的 key 是 Memory Pool 的ID，value 是对应显存池指针
- std::unordered_map<int64_t, int64_t> cuda_graph_ref_cnt_; 字典的 key 是 Memory Pool 的ID，value 是对应显存池被 Graph 使用的引用计数。这里有个疑问 Memory Pool 不应该是每个 Graph 专用的吗？引用计数的作用是什么？只是为了及时释放吗?
  - Caputre 的过程中应该是专用的，但显存池不会直接释放

发现的问题

1. 底层对 Device 没有做很好的抽象，例如：phi::Stream 没有对 gpuStream_t 做很好的抽象，导致底层的 Allocator 中出现了与比较多的与设备（GPU，XPU，CUSTOM_DEVICE）相关的冗余代码
2. 问题 1 进一步导致了代码中出现了大量 `#if defined #endif` 预处理指令，不规则的缩进导致代码可读性非常差
3. 理论上 StreamSafeAllocator 以及对应的显存池管理对 CUDA Graph 的支持并不完善，可能会出现未定义行为。（已与 @陈锐彪 确认）

参考文档

1. 飞桨框架v2.4发布新版动态图执行引擎！性能大幅度提升
2. 多stream安全Allocator方案设计
3. 动态图多stream支持
4. 飞桨 kernel 分发逻辑梳理
5. 算子库适配多流显存管理
6. 飞桨训练显存管理和分析
7. 【动静一致multi-stream支持】方案设计