设计目标
DESIGN GOALS

设计考虑

1. 面向开发者（应用开发者、算法工程师）
   提供一个易用的c++ interface（api）为开发者实现提供便利
   能提供良好的抽象以满足软件开发中的可扩展性、低耦合度等要求
   能优雅得处理与第三方的依赖以及zensim框架下其他repo的关系

2. 面向并行计算架构（cpu多核、gpu众核）：
   host端并行：依赖openmp和simd
   device端并行：依赖cuda toolkit
   对常用parallel primitive提供实现（如foreach、scan、reduce、sort）

3. 基于数据的设计（data oriented design）：
   注重访存效率（优化cache使用、减少dram访问开销）
   数据可以方便地在不同memory space间移动和拷贝
   根据benchmark和实践经验，最理想且通用的layout是aosoa

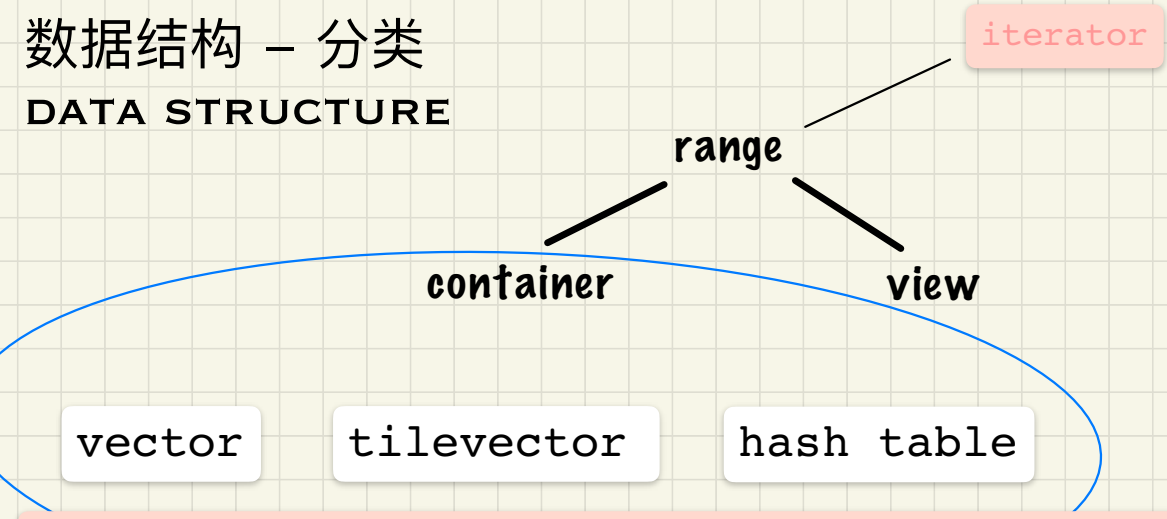
4. 业务为导向
   物理仿真计算本身高度data parallel，与2、3非常契合
   需提供仿真中所仰赖的线性系统解算
   （包含矩阵和向量的数据结构和常用的数值方法）

和2、3有关的数据类型

```
enum execspace_e { seq, openmp, cuda };
enum memsrc_e { host、device、um };
OmpExecutionPolicy、CudaExecutionPolicy
MemoryHandle { memsrc_e space; ProcId deviceid; };
```

数据结构 – 分类

DATA STRUCTURE



vector　tilevector　hash table

说明: 三种最基础容器，用于构建更复杂的数据结构。
构成: snode\ memory handle\ auxiliary data
snode, i.e. structure node (structure descriptor: see next page)
memory handle (allocation descriptor: memory space, e.g. cuda or host; location, e.g. which gpu device)
auxiliary data (e.g. a counter for hash table which stores the number of entries at the moment)
接口: 包含各类构造函数（RAII）、clone、resize (only for vector & tilevector)等基础维护功能
使用: 在各类计算后端（compute backend）中的使用须通过其proxy对象来访问，例如：

```
Vector<float> arr; // 其proxy对象类型是VectorProxy<execspace_e, Vector<float>>
computeOnDevice(proxy<execspace_e::cuda>(arr), ...); //在cuda内核中使用时得通过其cuda端的proxy对象
```

spatial / geometry acceleration

bvh
octree
hash buffer
particle list

spatial / geometry storage

sparse levelset
adaptive levelset
sparse grid
staggered grid

particle
mesh

说明：
以上是面向物理仿真开发的两类数据结构（根据目的划分）：空间/几何加速结构、空间/几何存储结构
区分标准是前者自身不会存储实际的物理仿真数据，而后者会

数据结构 – snode实现

DATA STRUCTURE

snode

`structure node`

记录
1. 总字节数
2. 每个通道的offset
3. 元素间的stride

访问：
by operator () overload

decoration：指定通道间以soa或aos形式布局

描述每个元素的所有属性在存储中的布局方式

domain：元素访问空间

多维度、编译期/运行时指定

channel type list：属性类型列表，如<float, int, child_snode, wrapt<UserDefinedType>>

snodes构成static hierarchy（静态层级结构）
对于每个snode，子结点可以是
1. 其他自定义的snode
2. 基础数据类型，如float、int、vec<T, Ns...>等
3. 用户自定义struct，如struct Foo { int a; char b; };

`structure node + buffer/pointer = structure instance`

描述一片连续存储空间的内在结构。
对于每个structure node，用户指定通道的编号（编译期/运行时）和元素编号（在domain内的）可以：
1. 获取所存储的数据
2. 得到某个structure instance，可进一步访问对应结构

存储维护

MEMORY MAINTENANCE

raw_allocator

最接近allocation api的一层allocator，根据memsrc_e区分

memsrc_e::host	malloc、free
memsrc_e::device	cudaMalloc、cudaFree
memsrc_e::um	cudaMallocManaged……

同时会记录每个allocation到一个map里

pmr allocator

advisor_allocator

上游allocator（upstream）必须是memsrc::um类型的raw_allocator，特点是在allocation完成后额外给予hint（比如设置preferred location等）

前述的包含容器在内的各类数据结构均从这些allocator里获取存储空间

std::pmr::pool_allocator

比较鲁棒和通用的存储分配器

upstream源头是raw_allocator

两类存储需求

通过结点系统维护的所有IObject，生命周期由开发者维护

通常存储于全局map结构中

执行结点计算时临时需求的临时存储空间

比如scan时需要额外分配一段存储空间来协助完成计算，用完即弃。可预先分配一个存储池从而避免频繁调用系统级存储操作api