Fine-grained Resource Configuration

Motivation

The current resource request for container does not consider the resource usages in specific UDF. It assumes that the UDF should just create short-life small objects and these can be recycled quickly by JVM, so most of the resources will be controlled and managed by framework. It is suitable for most of the scenarios and can work well in performance.

For some special cases, the UDF may consume much resources in different ways. For example. the user may load big dictionary in memory or define the large internal cache in UDF, and this kind of memory will not be released and recycled. So it will result in exceeding the maximum resource limit of container and even decrease performance.

To avoid the above bad cases, this proposal provides an option to give users the possibility to define the resource usages if needed. And the container resource request should also consider this part of resource in UDF in order to get the best performance.

Core Changes

- The container resource that RM requests from the cluster is composed of cpu cores and memory currently. The cpu cores in container resource is the same with that in slot resource profile from JM requests to RM, and the memory in container should consider extra framework memory except for the total memory in slot resource profile.

- The framework memory is mainly used by network buffer pool and memory manager services in TM, and it is occupied along with TM startup no matter with running tasks. Currently this part of memory size and type(heap or direct) can be configured according to job scale. For future improvements, it can be configured and deterministic in task level, so considered together in slot resource profile from JM requests to RM.

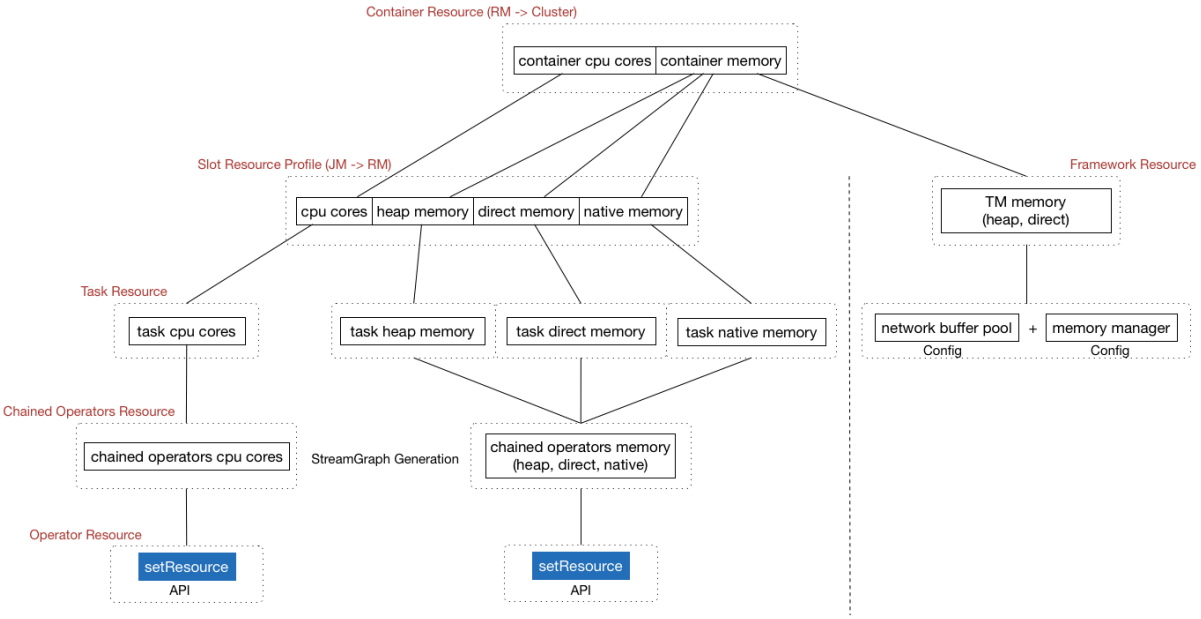
- The cpu cores in slot resource profile refers to the cpu usages by running tasks, and they can be calculated from chained operators when generating job graph based on individual operator cpu cores setted by **setResource** API.

- The memory in slot resource profile is expanded to different types(heap, direct, native) and also refers to the memory usages by running tasks.

- The task memory can be calculated from chained operators which mainly concern about memory usages in states and UDF. Users can estimate the state size by **setResource** API, and the speicific state backend provides the method to give the

best suitable memory usages in different types based on the state size. For UDF memory usages, it can also be setted by **setResource** API.

- Apart from resource, The JVM options attached with container should be supported and could also be configured in job level.



Public Interfaces

On the API level, the main objective is to give users the options to expose different resources usages for operators concerned about UDF implementation. In order to define the resource and consider dynamic expansion in an easy way, we introduce the **ResourceSpec** class to describe the different resource factors and provide some basic construction methods for resource group.

The **ResourceSpec** can be setted onto the internal transformation in DataStream and base operator in DataSet separately, the similar way with other existing properties like parallelism, name, etc. In order to support resource resize for container, the **ResourceSpec** will be setted as minimum and maximum in the API for future improvements.

ResourceSpec

```
public class ResourceSpec implements Serializable {

    public ResourceSpec(
            double cpuCores,
            long heapMemoryInMB,
            long directMemoryInMB,
            long nativeMemoryInMB,
            long stateSizeInMB) {
```

```
    }

    // without state
    public ResourceSpec(
            double cpuCores,
            long heapMemoryInMB,
            long directMemoryInMB,
            long nativeMemoryInMB) {
    }

    //only need cpu and java heap memory as common
    public ResourceSpec(double cpuCores, long heapMemoryInMB) {
    }
}
```

SingleOutputStreamOperator(DataStream)

```
public SingleOutputStreamOperator<T> setResource(
            ResourceSpec minResource, ResourceSpec maxResource) {
    transformation.setResource(minResource, maxResource);
    return this;
}

public SingleOutputStreamOperator<T> setResource(ResourceSpec resource) {
    transformation.setResource(resource, resource);
    return this;
}
```

Operator(DataSet)

```
public O setResource(ResourceSpec minResource, ResourceSpec maxResource) {
    this.minResource = minResource;
    this.maxResource = maxResource;
    O returnType = (O) this;
    return returnType;
}

public O setResource(ResourceSpec resource) {
    this.minResource = resource;
    this.maxResource = resource;
    O returnType = (O) this;
    return returnType;
}
```