提纲

ComputeTask

ComputeTask是ignite关于基于内存的MapReduce的简单抽象。通常当用户需要对job到节点的调度（job-to-node mapping）进行细粒度控制的时候，或者需要自定义fail-over逻辑的时候，才是用ComputeTask，对于其他情况下，则应该使用Distributed Closures（参考这里）。

通过实现ComputeTask接口中的map(...)和reduce(...)方法，就可以定义一个Task，当然也可以自定义实现Result(...)方法。

```
public interface ComputeTask<T, R> extends Serializable {
    # 该方法用于将task拆分成一系列的jobs。
    # @param arg: task参数
    # @param subgrid:  可供当前tak执行的节点集合
    # @return:  jobs到它所对应的节点的映射
    @Nullable public Map<? extends ComputeJob, ClusterNode> map(List<ClusterNode> subgrid,
@Nullable T arg) throws IgniteException;

    # 这是一个异步回调，每当接收到某个节点的执行结果的时候就会被调用，由该方法决定是等待更多的节点的执行结果，
    # 还是在当前所有已经接收到的结果上执行reduce，还是将当前的job failover到其它节点上
    # @param res：接收到的执行结果
    # @param rcvd：之前已经接收到的所有的results，如果task所在的类上添加了ComputeTaskNoResultCache这个注释，则@rcvd为空
    # @return：返回ComputeJobResultPolicy，决定如何处理后续的job results
    public ComputeJobResultPolicy result(ComputeJobResult res, List<ComputeJobResult> rcvd) throws
IgniteException;

    # 该方法用于对当前已经接收到的所有的results上执行聚合，如果某些jobs执行失败且无法正常failover到其它节点上，则@results中
    # 会包含这个失败的job的执行结果，否则的话，@results中将不会包含失败的job的执行结果
    @ param results：所有已经接收到的jobs result，如果task所在的类上添加了ComputeTaskNoResultCache这个注释，则@results为空
    @ return:  聚合后的task result
    @Nullable public R reduce(List<ComputeJobResult> results) throws IgniteException;
}
```

ComputeTaskAdapter

ComputeTaskAdapter实现了ComputeTask接口，并提供了一个默认的关于result(...)方法的实现：如果一个job抛出了异常，则返回FAILOVER policy，否则返回WAIT policy（以等待所有jobs完成）。

ComputeTaskSplitAdapter

ComputeTaskSplitAdapter继承自ComputeTaskAdapter，它自动负责job到node的映射，ComputeTaskSplitAdapter非常适合于同构的环境，因为同构环境中，在执行jobs到nodes的映射时，所有的节点都适合于任意一个job（This adapter is especially useful in homogeneous environments where all nodes are equally suitable for executing jobs and the mapping step can be done implicitly.）。

ComputeJob

在ComputeTask中创建的所有的jobs都实现了ComputeJob接口，ComputeJob中的execute()方法定义了job的逻辑，并且返回job的执行结果，ComputeJob中的cancel()方法定义了当取消某个job时的逻辑。

ComputeJobAdapter

ComputeJobAdapter实现了ComputeJob接口，并且提供了一个空的cancel()方法。

ComputeLoadBalancer

ComputeLoadBalancer的作用

有时候在定义Ignite任务的时候，会有类似如下的语句：

```
public class MyFooBarTask extends ComputeTaskAdapter<String, String> {
    // Inject load balancer.
    @LoadBalancerResource
    ComputeLoadBalancer balancer;

    ...
}
```

这里的ComputeLoadBalancer起到什么作用呢?

ComputeLoadBalancer用于根据负载均衡策略查找到最适合的节点，从代码内部来讲，它是通过查询org.apache.ignite.spi.loadbalancing.LoadBalancingSpi来找到最合适的节点的。

ComputeLoadBalancer的使用

ComputeLoadBalancer的使用方式有以下3中：

- 如果用户直接通过实现ComputeTask来定义任务，则直接在Task内部使用；
- 如果用户通过继承ComputeTaskAdapter类来定义任务，则直接在Task内部使用；
- 如果用户通过继承ComputeTaskSplitAdapter类来定义任务，则ComputeLoadBalancer会被隐式的使用；

如果要使得ComputeLoadBalancer生效，则必须满足：

- 设置LoadBalancingSpi
- 没有设置AffinityKeyMapped annotation（参考这里）

LoadBalancingSpi

LoadBalancingSpi提供下一个最适合的node来执行下一个job（注意：ignite中的job和task的概念和flink中的刚好相反，在ignite中一个task被拆分为多个job）。

ignite中原生提供了以下几个LoadBalancingSpi：

- org.apache.ignite.spi.loadbalancing.roundrobin.RoundRobinLoadBalancingSpi（这是ignite中默认使用的LoadBalancingSpi）
- org.apache.ignite.spi.loadbalancing.adaptive.AdaptiveLoadBalancingSpi
- org.apache.ignite.spi.loadbalancing.weightedrandom.WeightedRandomLoadBalancingSpi

RoundRobinLoadBalancingSpi

可以在spring xml配置文件中按照如下方式配置RoundRobinLoadBalancingSpi：

```
    <property name="loadBalancingSpi">
        <bean class="org.apache.ignite.spi.loadbalancing.roundrobin.RoundRobinLoadBalancingSpi">
            <property name="perTask" value="false"/>
        </bean>
    </property>
```

AdaptiveLoadBalancingSpi

该SPI大概率的将更多的任务分发到性能更好的节点上执行，它提供了一个可插拔的算法（AdaptiveLoadProbe）用于在任意时刻计算节点的负载，它提供了一个AdaptiveLoadProbe接口，用户可以实现该接口来自定义节点负载探测逻辑。默认的AdaptiveCpuLoadProbe被使用，它根据每个节点上的平均CPU来执行job调度。

可以在spring xml配置文件中按照如下方式配置AdaptiveLoadBalancingSpi：

```
<property name="loadBalancingSpi">
    <bean class="org.apache.ignite.spi.loadBalancing.adaptive.AdaptiveLoadBalancingSpi">
        <property name="loadProbe">
            <!--除了AdaptiveProcessingTimeLoadProbe以外，还可以设置AdaptiveCpuLoadProbe，或者
AdaptiveJobCountLoadProbe-->
            <bean
class="org.apache.ignite.spi.loadBalancing.adaptive.AdaptiveProcessingTimeLoadProbe">
                <property name="useAverage" value="true"/>
            </bean>
        </property>
    </bean>
</property>
```

关于AdaptiveLoadBalancingSpi的配置中会用到loadProbe，ignite提供了3个原生的loadProbe：

- AdaptiveCpuLoadProbe 使用节点的平均CPU负载或者实时CPU负载，默认采用平均CPU负载；可以通过setUseAverage(boolean)来决定使用平均计数还是实时计数；
- AdaptiveJobCountLoadProbe 使用节点上处于活跃或者等待状态的job计数来进行负载均衡，节点计数可以是平均的，也可以是实时的；可以通过setUseAverage(boolean)来决定使用平均计数还是实时计数；
- AdaptiveProcessingTimeLoadProbe 使用节点上所有job的总的处理时间来进行负载均衡，总的处理时间可以是平均的，也可以是实时的；可以通过setUseAverage(boolean)来决定使用平均计数还是实时计数；

WeightedRandomLoadBalancingSpi

该SPI可以选择性的为每个节点设置权重，虽然在调度任务的时候，是随机选择节点的，但是具有较大权重的节点将会大概率被调度较多个job。可以通过setNodeWeight(int)来设置节点的权重，默认的，每个节点具有相等的权重（DFLT_NODE_WEIGHT = 10）。

可以在spring xml配置文件中按照如下方式配置WeightedRandomLoadBalancingSpi：

```
<property name="loadBalancingSpi">
    <bean
class="org.apache.ignite.spi.loadBalancing.weightedrandom.WeightedRandomLoadBalancingSpi">
        <property name="useWeights" value="true"/>
        <property name="nodeWeight" value="10"/>
    </bean>
</property>
```

Distributed Task Session

每个Task执行过程中都对应了一个distributed task session，由接口ComputeTaskSession来定义。Task session对于task和所有率属于该task的jobs可见，因此在task上或者其中的某个job上设置的属性可以被率属于该task的其它jobs访问。在属性被设置的时候，Task session会接收到通知，Task session也可以等待某个属性被设置。对于Task和其对应的所有的jobs来说，属性设置的顺序是一致的。

Per-Node Shared State

Ignite在每个节点上提供了一个并发的node-local-map，就跟Thread Local类似，这个map只在节点本地保存。这主要用于在同一个节点上的不同jobs之间共享状态。

Job Scheduling

Task会被拆分为不同的jobs，并调度到相应的节点上执行，如果有多个jobs（可能来自于多个不同的tasks）被调度到同一个节点上，那么就需要确定这些job的执行顺序，**默认的，这些jobs将被提交到某个thread pool中，并且以随机的顺序被调度**，如果你需要更好的控制job的执行顺序，则需要开启CollisionSpi。Ignite提供了4种CollisionSpi：

- FifoQueueCollisionSpi jobs以FIFO的顺序被调度执行，并且允许#getParallelJobsNumber()数目的jobs并发执行，其它的jobs则进入等待队列。可以通过#setParallelJobsNumber(...)来设置并发执行的jobs的数目，如果调用了setParallelJobsNumber(1)，则所有的jobs依次执行，上一个job执行完毕之后下一个job才可以开始运行。通常jobs的执行并发度应该被设置为thread pool中的线程的数目。

可以在spring xml中按照如下方式配置FifoQueueCollisionSpi：

```
<bean id="grid.custom.cfg" class="org.apache.ignite.configuration.IgniteConfiguration"
singleton="true">
    ...
    <property name="collisionSpi">
        <bean class="org.apache.ignite.spi.collision.fifoqueue.FifoQueueCollisionSpi">
            <property name="parallelJobsNumber" value="1"/>
        </bean>
    </property>
    ...
</bean>
```

- PriorityQueueCollisionSpi 使用PriorityQueueCollisionSpi来为每个job设置优先级，高优先级的job将先于低优先级的job被调度。

Task的优先级可以通过TaskSession中的grid.task.priority属性来进行设置，如果没有设置该属性，则默认的优先级为0。下面的代码演示了如何设置Task的优先级：

```
public class MyUrgentTask extends ComputeTaskSplitAdapter<Object, Object> {
  // Auto-injected task session.
  @TaskSessionResource
  private ComputeTaskSession taskSes = null;

  @Override
  protected Collection<ComputeJob> split(int gridSize, Object arg) {
    ...
    // Set high task priority.
    taskSes.setAttribute("grid.task.priority", 10);

    List<ComputeJob> jobs = new ArrayList<>(gridSize);

    for (int i = 1; i <= gridSize; i++) {
      jobs.add(new ComputeJobAdapter() {
        ...
      });
    }
    ...

    // These jobs will be executed with higher priority.
    return jobs;
  }
}
```

和FifoQueueCollisionSpi类似，PriorityQueueCollisionSpi中也可以通过#setParallelJobsNumber(...)来设置并发执行的jobs的数目。

可以在spring xml中按照如下方式配置，PriorityQueueCollisionSpi：

```
<bean class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <property name="collisionSpi">
        <bean class="org.apache.ignite.spi.collision.priorityqueue.PriorityQueueCollisionSpi">
        <!-- Change the parallel job number if needed. Default is number of cores times 2.-->
            <property name="parallelJobsNumber" value="5"/>
        </bean>
    </property>
    ...
</bean>
```

- JobStealingCollisionSpi JobStealingCollisionSpi支持从over-utilized nodes中“steal” jobs到under-utilized node中。

JobStealingCollisionSpi必须和org.apache.ignite.spi.failover.jobstealing.JobStealingFailoverSpi一起使用，同时要开启job统计计数（job metrics update）。

可以在spring xml中按照如下方式配置JobStealingCollisionSpi：

```
<property name="collisionSpi">
    <bean class="org.apache.ignite.spi.collision.jobstealing.JobStealingCollisionSpi">
        <property name="activeJobsThreshold" value="100"/>
        <property name="waitJobsThreshold" value="0"/>
        <property name="messageExpireTime" value="1000"/>
        <property name="maximumStealingAttempts" value="10"/>
        <property name="stealingEnabled" value="true"/>
        <property name="stealingAttributes">
            <map>
                <entry key="node.segment" value="foobar"/>
            </map>
        </property>
    </bean>
</property>
```

- NoopCollisionSpi 在NoopCollisionSpi下，jobs一旦被调度到某个节点上，就在该节点上立即运行，这非常适合于具有大量短任务的情况。用户可以通过org.apache.ignite.configuration.IgniteConfiguration#setPublicThreadPoolSize(...)来设置可以并发执行的jobs的数目。