技术文档 | Apollo Cyber调度器

Apollo Cyber RT框架核心理念基于组件，通过组件实现有预先设定的“输入”、“输出”。实际上，在框架中，每个组件代表一个专用的算法模块。框架可根据所有预定义的组件生成有向无环图 (DAG)。

在运行时刻，框架把融合好的传感器数据和预定义的组件打包在一起形成用户级轻量任务，之后，框架的调度器即可根据资源可用性和任务优先级来派发这些任务。

以下，ENJOY

Apollo Cyber RT是百度自研得无人车计算任务实时并行计算框架，该框架作为Apollo3.5的一部分已经开放给开发者们。Cyber RT的一个重要模块是Scheduler模块，它是RT框架的核心部分。

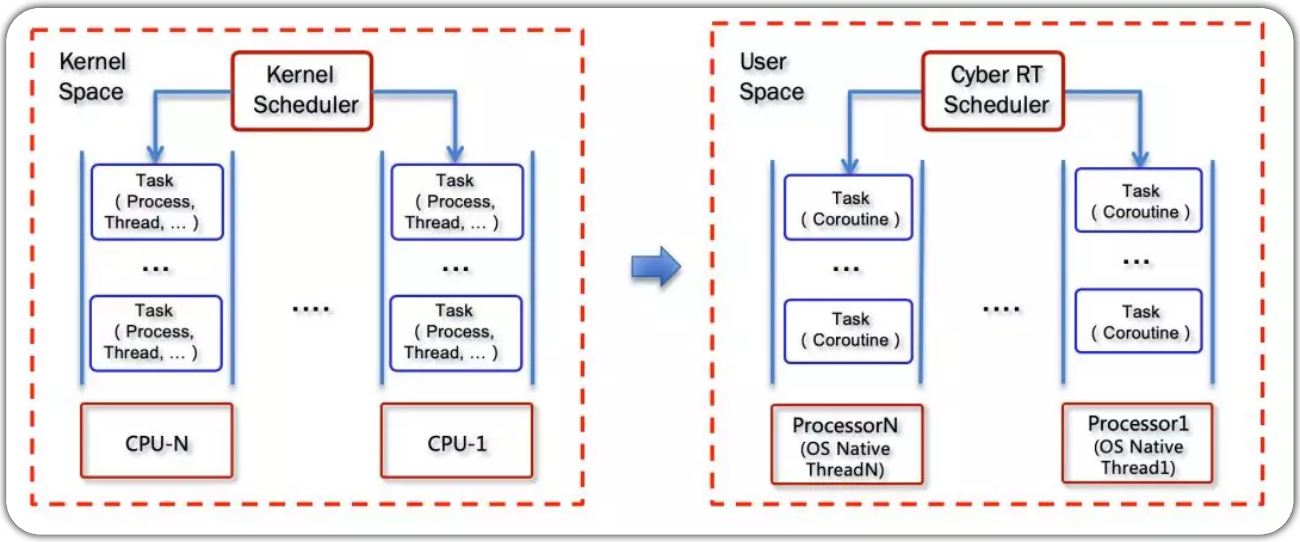
调度器解决了什么问题

Scheduler将Kernel调度、任务(进程、线程)从内核空间搬到了用户空间，在用户空间实现了对车载计算任务对计算资源依赖的统一调配。对于算法业务而言，他们只需要关注自己的无人车感知、规划等算法本身实现，而无需关注并发、同步，甚至是操作系统锁、同步、资源调配等细节问题，从而大大提高了并行算法任务的编写。

调度器设计

整体设计

Cyber RT Scheduler的整体设计如下：



1、Cyber RT Sched通过pthread_setschedparam、pthread_setaffinity_np等接口，将Native Thread绑定到不同的cpu上，尽量做到Per-Cpu Per-Thread。

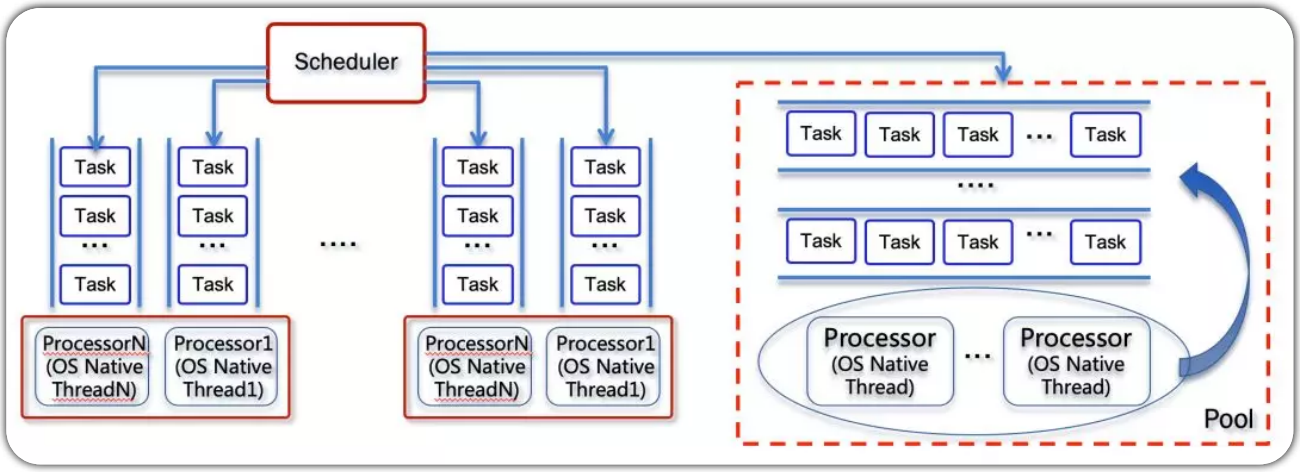
2、在Cyber RT Scheduler角度，OS的Native Thread可以类比为物理CPU，我们将cyber中封装Native Thread的类称为Processor。

3、在OS中，内核负责调度用户创建的Native Thread到CPU上运行。而从Cyber RT Scheduler角度看，cyber RT通过栈上下文切换的方式，让上层空间创建的Croutine在Processor上有序运行。

4、类似于kernel sched， Cyber RT Sched也是Run Queue Per-Processor的实现方式。

调度策略

新的调度模型的好处是，调度框架可以按照具体业务需求，更合理的设计算法任务资源分配和优先级，比如，我们无人车的算法任务可以基于任务之间的相互关联关系，组成一张有向无环图，当前设计下，调度器可以将有直接依赖关系的任务绑定到同一块cpu或相邻cpu上，这样会极大提高cache locality。而不同的任务可以放到完全不同的cpu上，这样可以保证关键算法任务得到足够的cpu资源，进而他们可以对车辆保持实时控制。



我们提供了两种调度策略，一种是编排策略(CHOREO POLICY)，这种策略实现了与业务紧耦合的任务编排调度算法，算法任务绑Processor、优先级、执行顺序等都会被严格控制，这是一种高确定性的调度策略,理想状态下，所有任务都会被确定性安排；另一种是CLASSIC POOL(CLASSIC POLICY)， Pool中采用传统的轮询策略，任务会被放到一个多优先级有序队列中有序执行，这是一种近似公平的调度策略，对车上计算任务来说，调度器希望右边的池子越小越好，这个策略是为暂时无法编排的任务提供的。

调度器使用

调度系统调度策略主要分为classic策略和choreography策略。这两个策略都需要进行一定的配置，包括task优先级、绑定processor，以及processor本身分组、绑cpu、设定优先级等。

classic策略是一个较为通用的调度策略，这个策略配置简单，只需要按照业务需求配置好任务优先级即可保证车载算法正常运行。

choreography策略是严格的Per-Processor Per Runqueue设计，需要对车载算法DAG充分了解，才能配置出高性能的调度策略。

调度器配置文件是pb格式，所以配置相对比较方便。在这里将sched配置关键字段陈述出来，开发者可以按照自己的需求对调度器进行配置。

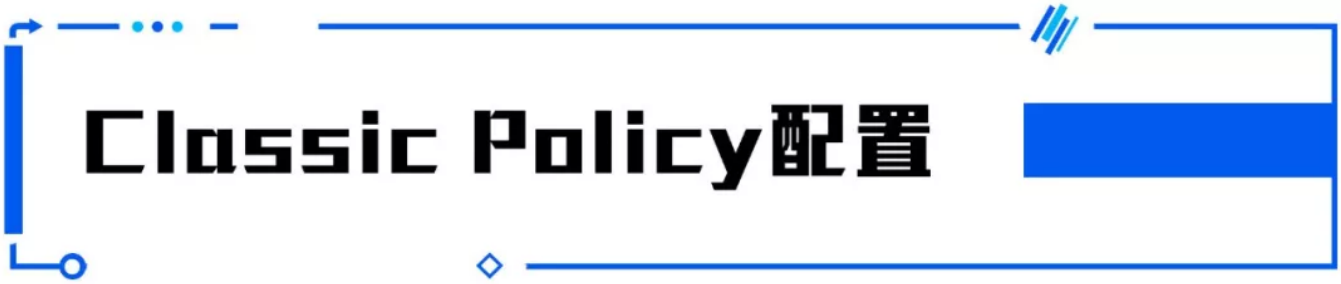

```
syntax = "proto2";
package apollo.cyber.proto;

message InnerThread {
  optional string name = 1;
  optional string cpuset = 2;
  optional string policy = 3;
  optional uint32 prio = 4 [default = 1];
}

message ChoreographyTask {
  optional string name = 1;                          /*Task名字*/
  optional int32 processor = 2;                      /*Task将要放到的Processor index*/
  optional uint32 prio = 3 [default = 1];            /*Task优先级*/
}

message ChoreographyConf {
  optional uint32 choreography_processor_num = 1;    /*处理编排任务的线程数量*/
  optional string choreography_affinity = 2;         /*处理编排任务的线程的亲和性设置*/
  optional string choreography_processor_policy = 3; /*处理编排任务的线程的调度策略*/
  optional int32 choreography_processor_prio = 4;    /*处理编排任务的线程的优先级*/
  optional string choreography_cpuset = 5;           /*分配处理编排任务的cpu详情*/
  optional uint32 pool_processor_num = 6;            /*处理未编排任务的线程数量*/
  optional string pool_affinity = 7;                 /*处理未编排任务的线程的亲和性设置*/
  optional string pool_processor_policy = 8;         /*处理未编排任务的线程的调度策略*/
  optional int32 pool_processor_prio = 9;            /*处理未编排任务的线程的优先级*/
```

```
27    optional string pool_cpuset = 10;                  /*处理未编排任务的cpu*/
28    repeated ChoreographyTask tasks = 11;
29    repeated InnerThread threads = 12;
30  }
```




```
1   syntax = "proto2";
2   package apollo.cyber.proto;
3
4   message ClassicTask {
5     optional string name = 1;                        /*Task名字*/
6     optional uint32 prio = 2 [default = 1];          /*Task优先级*/
7   }
8
9   message SchedGroup {
10    optional uint32 processor_num = 1;               /*指定分配调度线程个数*/
11    optional string affinity = 2;                    /*指定分配哪些cpu*/
12    optional string cpuset = 3;                      /*指定调度线程与cpu的亲和性*/
13    optional string processor_policy = 4;            /*指定调度线程的调度策略*/
14    optional int32 processor_prio = 5 [default = 0]; /*指定调度线程的优先级*/
15    repeated ClassicTask tasks = 6;
16  }
17
18  message ClassicConf {
19    repeated SchedGroup groups = 1;
20  }
```