ROS2基础架构

- callback：最小调度实体
- node：由应用产生，是一个callback的集合，是executor的最小分配单元。每个应用由一组node构成
- executor：在CPU上执行的、OS层面调度的实体，例如线程。本文章中每个executor被分配到一个CPU核上，对每个核执行FIFO调度，优先级1~99。executor内的callback调度与普通优先级实时调度不同，executor有两种独特的callback调度策略。callback的优先级由类型决定，timer最高，依次为subscription、service、client、waitable，所有callback被不可抢占式执行；executor通过rmw层通信，更新所有非timer callback在各自队列中的准备状态，这个更新会在所有队列为空的时候发生。这种延迟的callback就绪状态的更新使得非timer callback的优先级分配无效（？），并且让数据链以一个类轮询的方式执行。

  executor被分配了node之后，便开始spin。新来的数据会存储在middleware层，直到被callback使用
- chain：由一组callback组成，有先后顺序。对于时间触发的任务，起始callback是timer；事件触发的任务，起始callback可以是常规callback。周期性的任务也可以被认为是timer callback。chain的优先级由设计者给定。一个chain被绑定到一个executor上，一个executor可以执行多个chain。一个chain也可以被分开绑定到多个executor上。
- 问题：ROS2在执行executor时不区分chain-level callback；executor被CFS调度算法执行，导致无法优先考虑执行紧急chain的executor，OS级别的优先级分配可以给executor分配优先级，但这无法解决chain的自我干扰。
- 过载处理机制：用于处理timer callback错过周期。机制产生于timer callback开始执行时（通过执行rcl_timer_call函数）。首先next_call_time变量增加 timer callback的周期 到 其当前值，新的值表示下一次触发timer callback的时间。如果next_call_time在当前时间之前，那么会重复增加直到指向最早的未来时间。错过的timer任务被跳过。由于过载，对端到端时延的最长延迟为一个timer callback的周期，这是因为实际链的施放时间 实际上是由timer callback任务周期 和 实例链的开始执行时间 决定的。

PiCAS调度方案

- 目标：高优先级链应该比低优先级链先执行；如果链的两个实例被同一个CPU执行，链的前一个实例应该在下一个实例开始执行之前 执行完毕，这是为了避免同一个链的实例之间的自干扰。
- TH1：如果链的两个实例被同一个CPU执行，链的前一个实例保证在下一个实例开始执行之前 执行完毕 当以下条件能被满足时成立：（1）链的数据依赖下游callback的优先级比上游callback高（2）链的下游callback在比上游callback优先级更高的executor上执行
- 优先级分配策略
  - chains运行在同一个executor内：
    - 一个chain中的普通callbacks：按chain中的顺序，越靠后优先级越高
    - 一个chain中的普通callbacks和timer callbacks：普通callback按chain中的顺序，越靠后优先级越高；且timer callback的优先级比所有普通callback优先级低
    - 多个chain中的普通callbacks：高优先级的chain中的所有callback优先级要比低优先级的chain中的所有callback高。chain内的callback优先级按前述方式处理
    - 多个chain中的普通callbacks和timer callbacks：高优先级的chain中的timer callback优先级要比低优先级的chain中的timer callback高。单个chain中的普通callbacks和timer callbacks的优先级按前述方式处理。

    - chains运行在不同executor（不同executor绑定到同一个CPU上）：
        - executor内的callback优先级遵循前述策略
        - 一个chain绑定到单CPU：包含数据依赖上游的callback的executor的优先级低于包含下游callback的executor
        - 多个chain绑定到单CPU： 包含高优先级的chain的executor的优先级高于包含低优先级的chain的executor
- node分配（到executor）、executor映射（到CPU）算法
    - 策略：将与同一个chain关联的nodes尽可能分配到同一个CPU上
    - 维护一个所有待分配的node集合N^*，将N^*中的node按其包含的最高优先级callback按降序排列
    - 对于当前未被分配的最高优先级chain C，选择出N^*中与C关联的所有的node的集合N
    - 尝试找出空闲executor e
        - 情况A：有可用executor e，把N分配给e，e分配给一个可用CPU，判断CPU可用意味着原CPU利用率加上e利用率不超过1。如果没有可用CPU，从N中移除包含最低优先级callback的node n（移回N^*），再寻找可用CPU，如此循环直到N中只有一个node，如果此时仍然没有可用CPU，转到情况C。对于找到的可用CPU，将N分配到满足executor优先级策略的最高优先级executor，将此executor分配到最低利用率的CPU。如果找不到满足策略的executor，转到情况C。
        - 情况B：没有可用executor，找出一个对N可行的 已经被绑定到CPU核P_k的 非空e_m，判断可用意味着原CPU利用率加上N利用率不超过1。如果没有可用executor，从N中移除包含最低优先级callback的node n（移回N^*），再寻找可用executor，如此循环直到N中只有一个node，如果此时仍然没有可用executor，转到情况C。对于找到的可用executor，将N分配到满足callback、executor优先级策略的、最低利用率的executor。如果找不到满足策略的executor，转到情况C。
        - 情况C：有两种情形会导致情况C：第一，找到了可用CPU，但是不满足executor优先级策略，这种情况下，把这个CPU上所有executor合并成单executor，这样策略被满足；第二：所有CPU利用率都超过1，这种情况下直接把N分配到利用率最低的CPU