反应式流在大规模场景下的应用

0. Agenda

1. 自我介绍
2. 常见并发模型
3. 场景和挑战
4. 反应式流的应用
5. 社区相关思考

1. 自我介绍

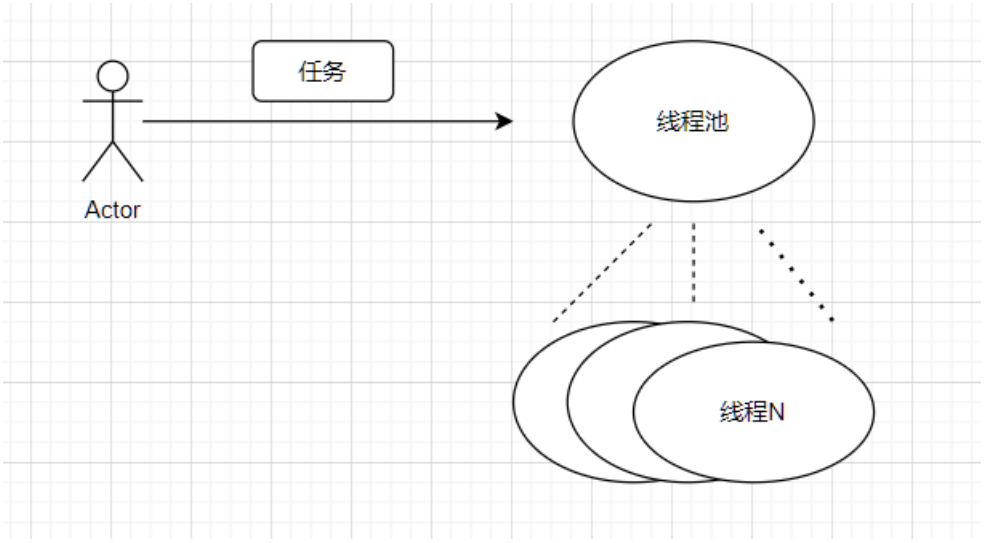
1. 何品，目前就职于淘宝 2016 –> ???
   参与过的项目：
   a. 淘宝内容平台建设，内容化建设，参与GraphQL的内部实现 TQL
   b. 淘宝反应式架构升级项目组
   c. 淘宝消息平台——手淘消息
   d. 消息基础——支持直播互动等
2. 参与翻译书籍：《Netty 实战》、《Scala 实用指南》、《反应式设计模式》
3. 参与贡献项目：Akka、Netty、Scala 等
4. 喜欢游戏比如《异度神剑》、《王者荣耀》、阅读、骑摩托

能和大家分享，很荣幸

2. 常见并发模型

1. 基于线程池

最普通的并发方式，通过一个线程池来管理一组线程。不同用途的线程池进行了隔离。



优势

1. 可以根据不同的业务和使用场景进行隔离。
2. 学习门槛较低

劣势

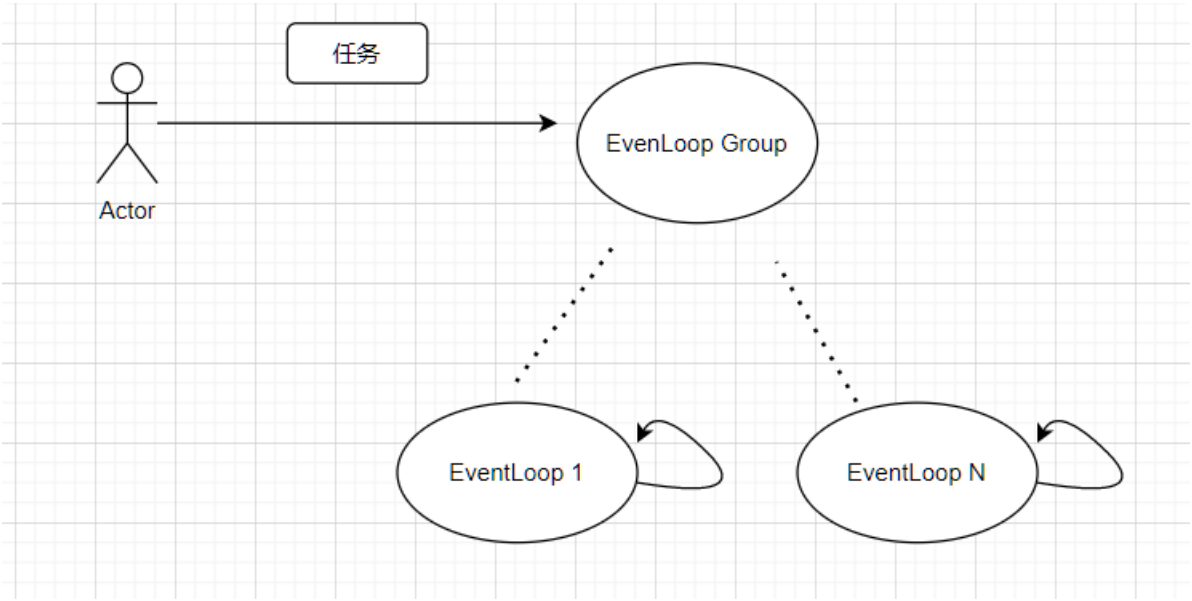
1. 线程较重，大规模上下文切换和线程影响性能。

代表

1. hsf 、tomcat等

2. 基于 EventLoop 模型

在普通的线程池模型上进行改进后的模型，把单个线程或者线程池看做一个单独的EventLoop



优势

1. 单个EventLoop 优先执行本EventLoop 内部的任务，更具线程亲和性
2. 一般要求EventLoop 执行的任务是全异步代码，性能高。

劣势

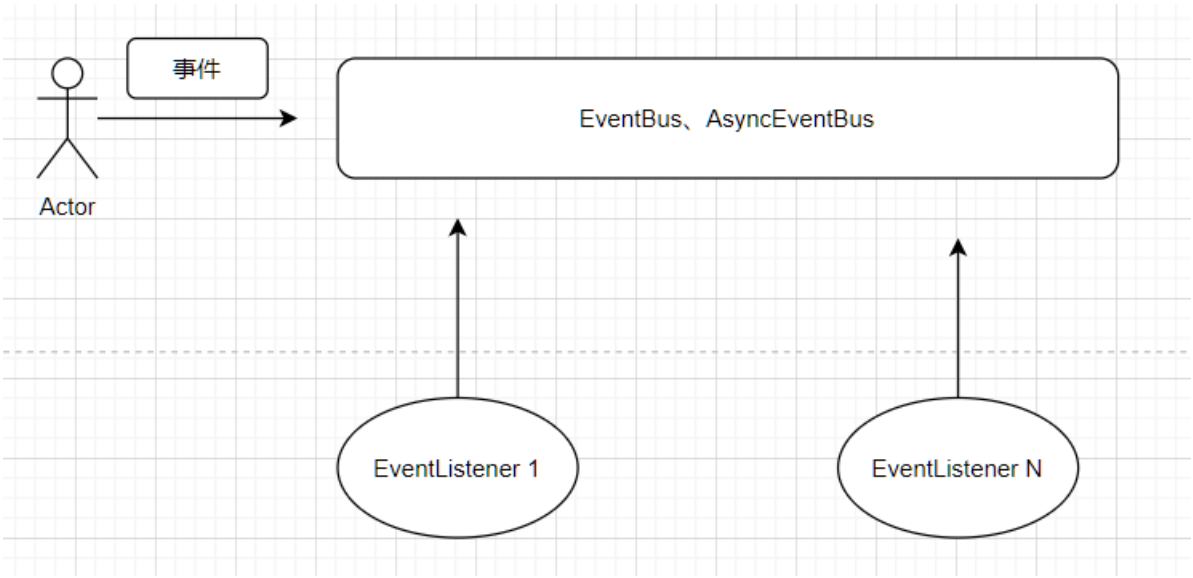
1. EventLoop 对阻塞代码不友好，学习成本高。
2. 如果配合为基于回调的代码，则容易形成 callback hell

代表

1. netty、vert.x

3. 基于 EventBus 模型

采用事件监听器的方式，对各种显式的线程池、EventLoopGroup等进行解耦



优势

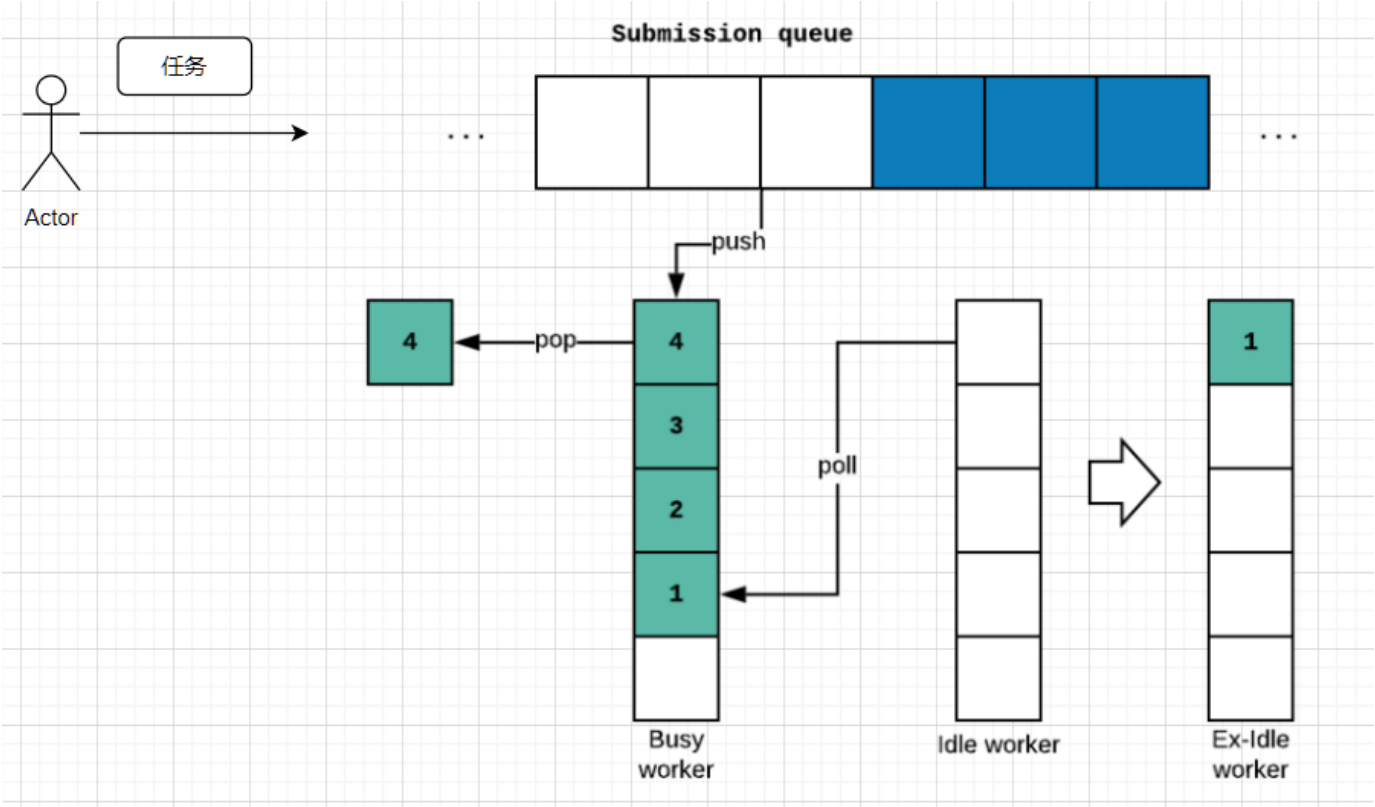
1. 使用起来较为简单，避免了异步代码带来的编码上的复杂性

劣势

1. 在分析代码的执行逻辑的时候，需要大量的分析和图示，不容易从代码直观的看出实际的执行流程
2. 容易出现因为错误造成的循环，且难以分析和定位

4. 基于 work stealing 的执行模型

根据work stealing的方式，充分利用多核心CPU的能力



优势

1. 可以充分利用多核心的CPU能力避免闲置

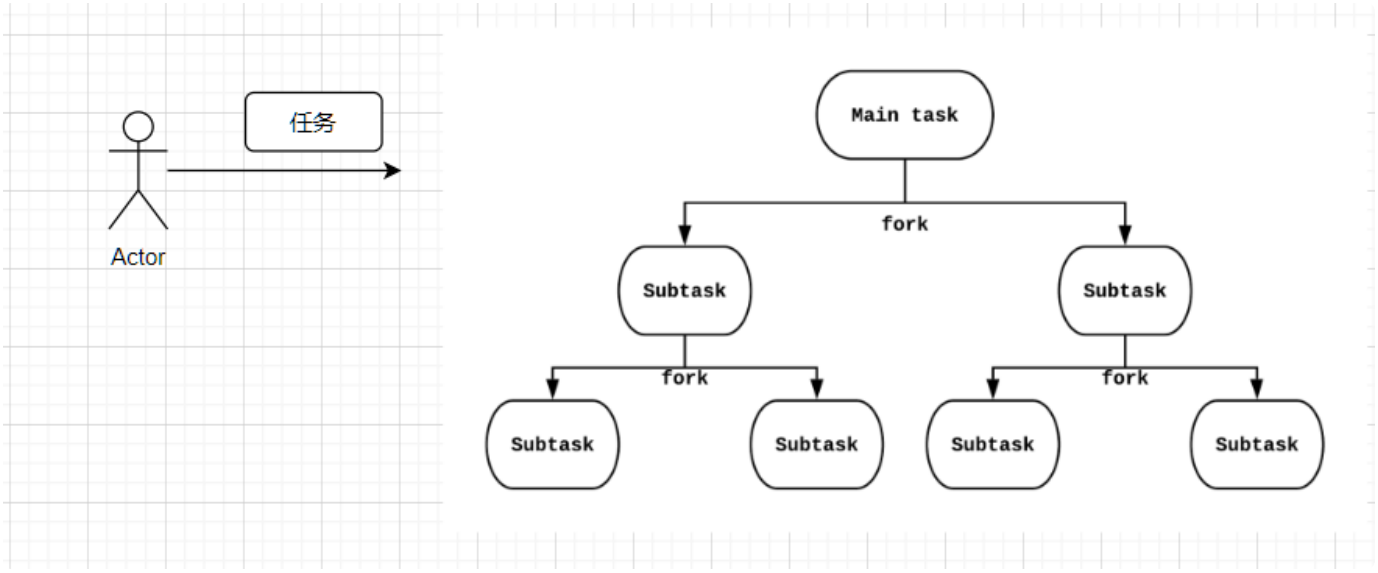
劣势

1. 仅对全CPU bounds的任务类型效果明显
2. 如果自己实现难度较高，一般为ForkJoinPool 的 FIFO 模式

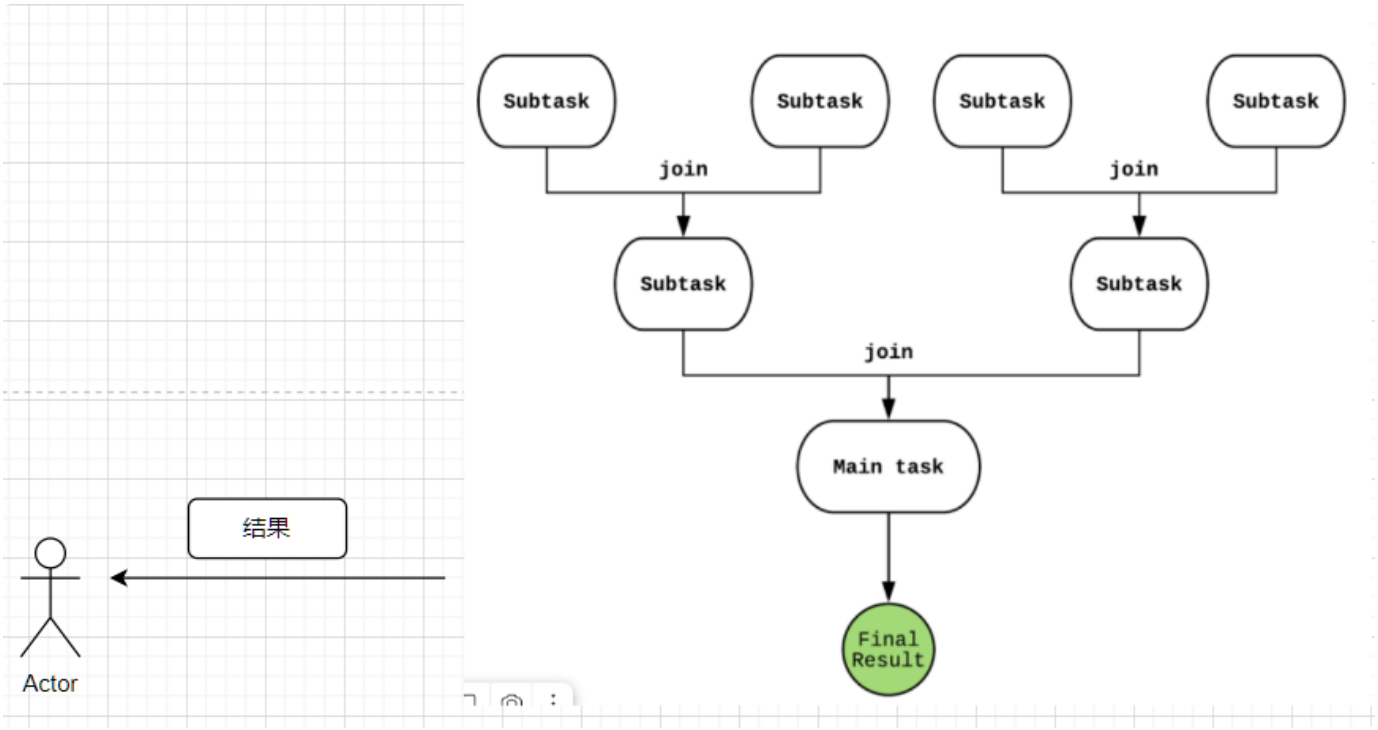
5. Fork Join 模型

一种典型的map、reduce 逻辑，一般需要特殊的线程池支持来达到更好的执行效果

任务拆分过程：



结果归并过程：



优势

1. 对于有明显 map reduce 类型的任务非常友好，一般需要配合ForkJoinPool 的LIFO模式进行使用

劣势

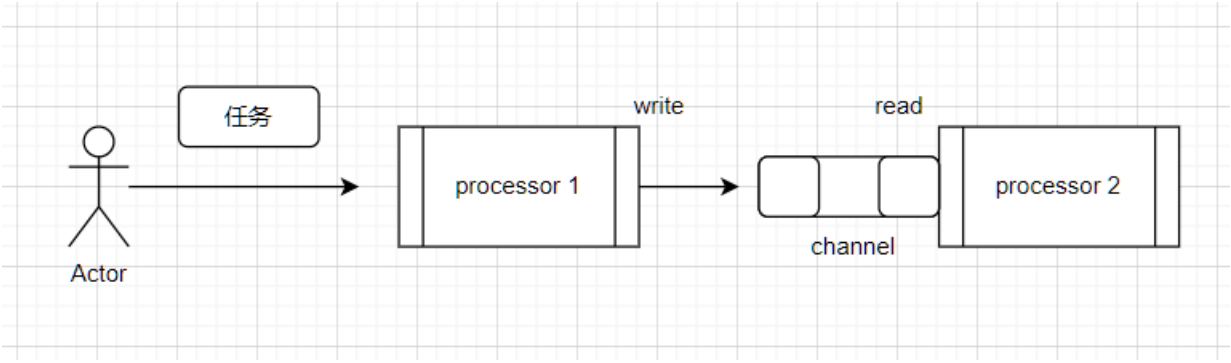
1. 适用场景有限，且实现难度较高

2. 使用难度较高

6. 基于CSP模型

基于 CSP 的模型可以用来实现并发，且在有直接支持的语言上更加简单

Do not communicate by sharing memory; instead, share memory by communicating.



优势

1. 在有直接支持的编程语言中使用起来非常简单

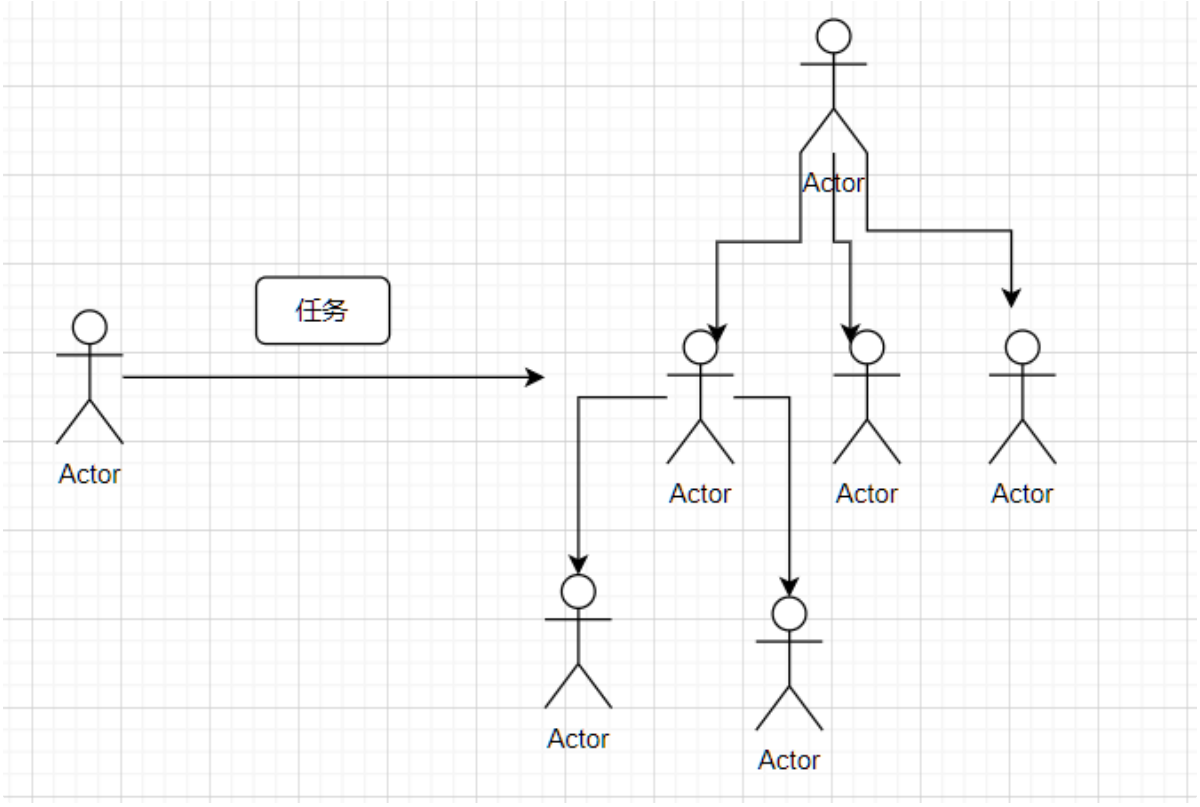
代表：Golang、Clojure、Kotlin

劣势

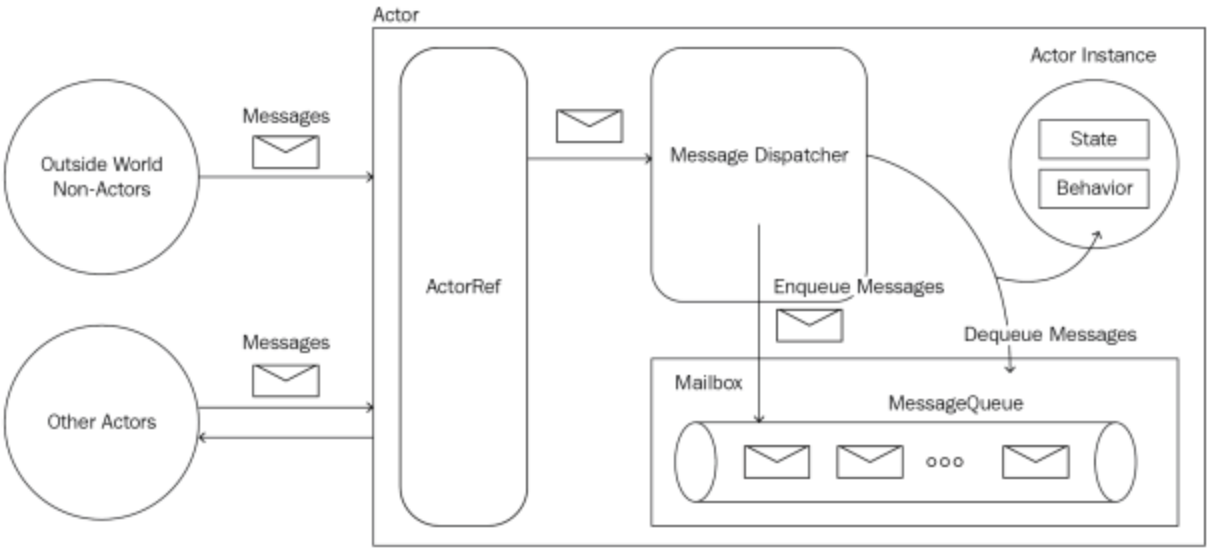
1. channel 无法天然支持分布式
2. 高度调度依赖于 scheduler

7. 基于 Actor 模型

类似 CSP 模型，但是基于全异步的消息来进行驱动



单个Actor实现内部解析：



优势

1. 天然支持分布式，且fire and forget 模式符合人类社会沟通方式

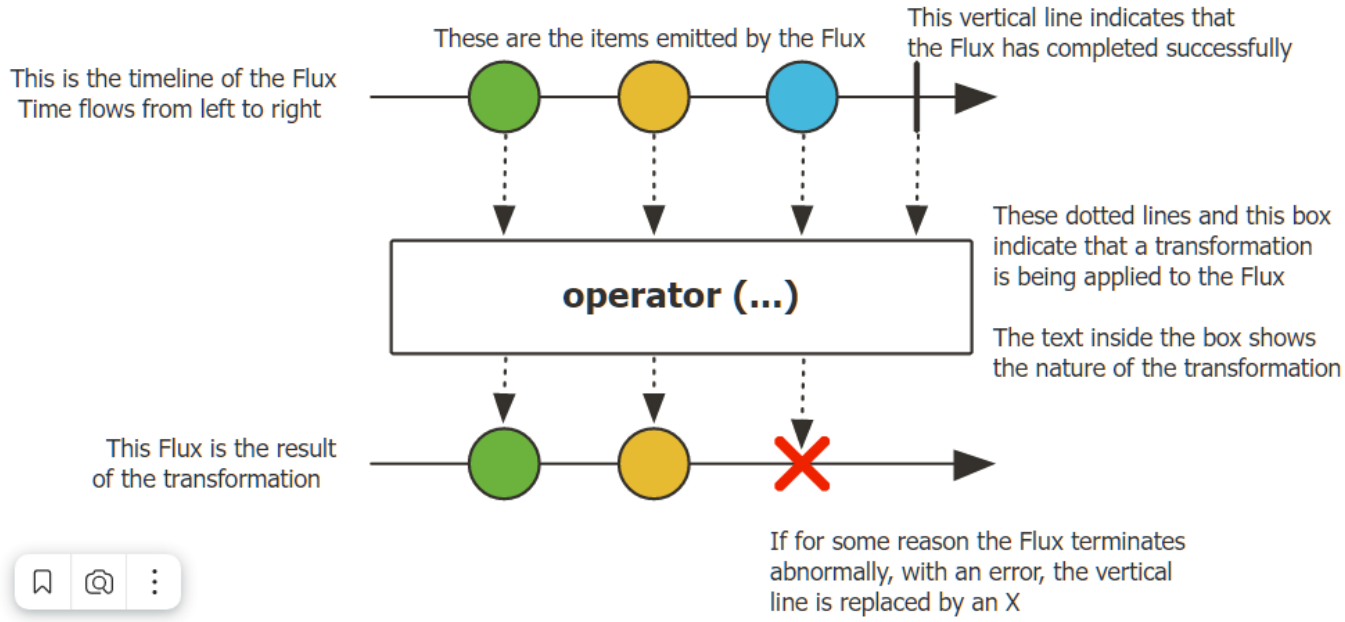
劣势

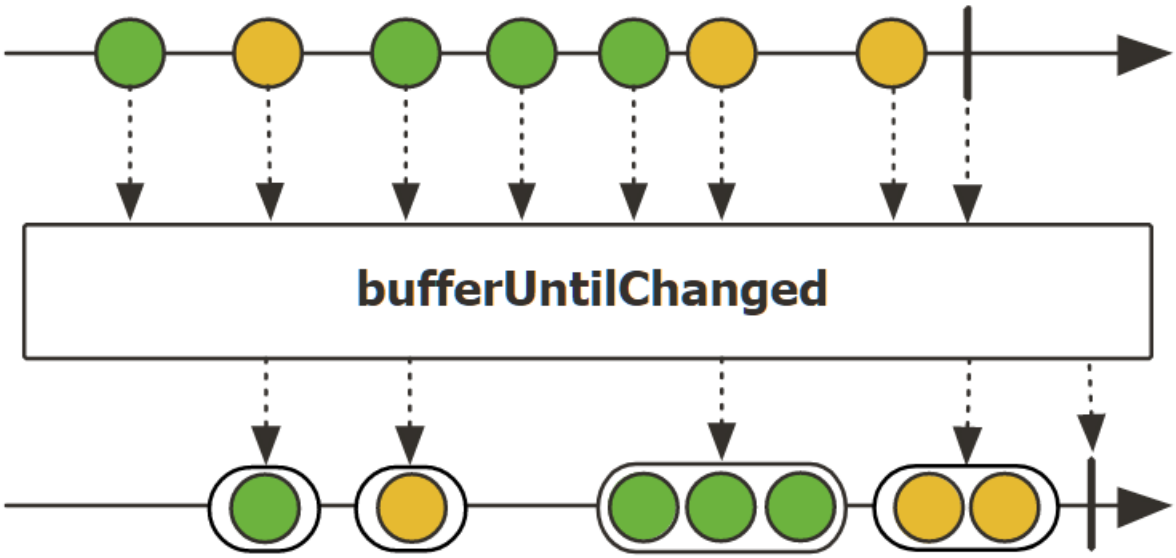
1. 无法清晰分析出代码的执行流程，不如CSP

2. 基于消息的模型通常无法较好的进行类型安全的编程（已部分解决）

8. 基于 Reactive 的函数式

基于 Reactor、RxJava 等声明式编程框架将并发控制和执行流程进行了精炼





```java
        Flux<List<AtomicInteger>> test =
                Flux.range(1, 100)
                    .map(AtomicInteger::new) // wrap integer with object to test gc
                    .map(retainedDetector::tracked)
                    .concatWith(Mono.error(new Throwable("expected")))
                    .bufferUntilChanged()
                    .take(50);
```

优势

1. 对逻辑和并发控制等进行了高度的精炼，信息更加内聚

劣势

1. 信息的高度浓缩造成的理解困难
2. 函数式带来的使用成本

代表

Akka、Reactor、RxJava、FS2 等

9. 虚拟线程

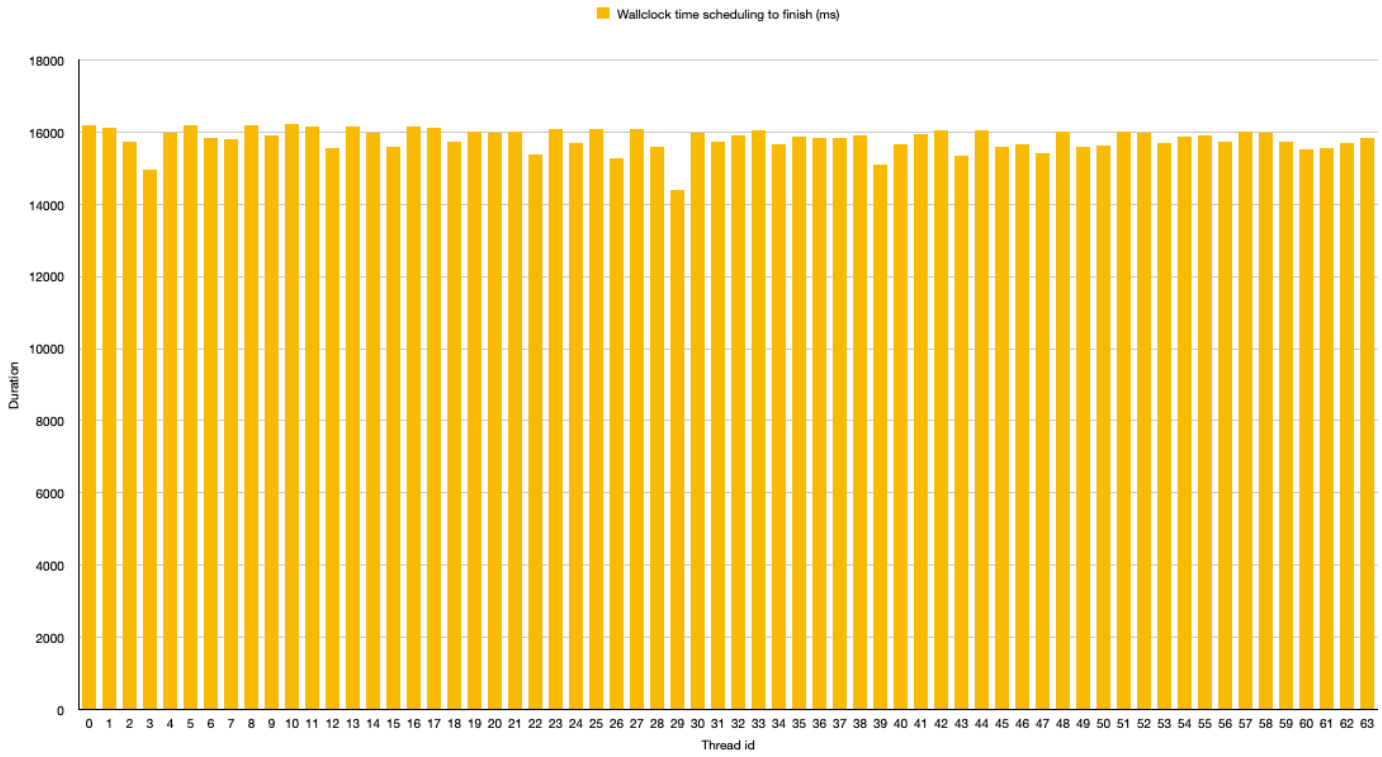
https://github.com/openjdk/jdk/commit/9583e3657e43cc1c6f2101a64534564db2a9bd84

https://openjdk.org/jeps/425

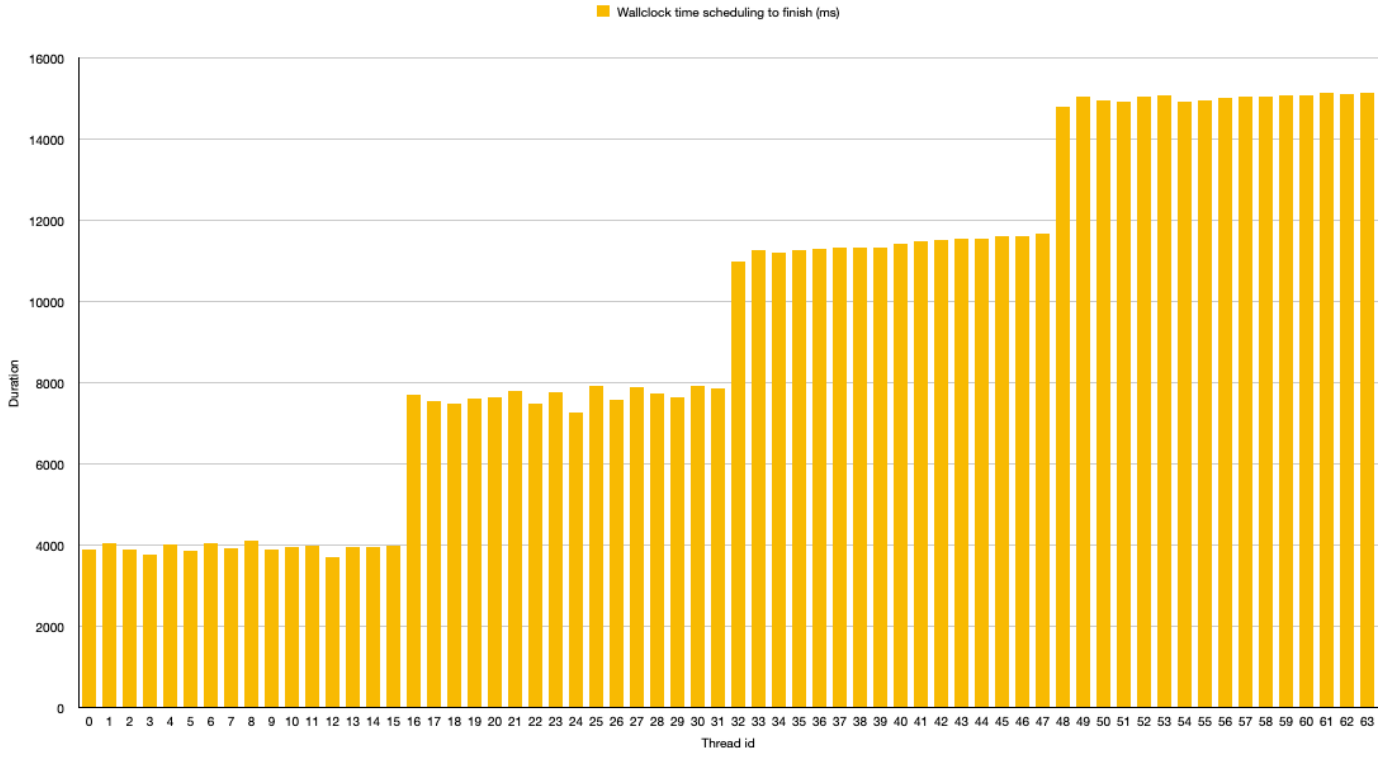
Java | 复制代码

```java
public class LoomTest {

  public static long blackHole;

  public static void main(String[] args) throws Exception {
    ExecutorService executor = Executors.newCachedThreadPool();
                          //换成
Executors::newVirtualThreadPerTaskExecutor()

    for(int i = 0; i < 64; i++) {
      final Instant start = Instant.now();
      final int id = i;

      executor.submit(() -> {
        BigInteger res = BigInteger.ZERO;

        for(int j = 0; j < 100_000_000; j++) {
          res = res.add(BigInteger.valueOf(1L));
        }

        blackHole = res.longValue();

        System.out.println(id + ";" +
            Duration.between(start, Instant.now()).toMillis());
      });
    }

    executor.shutdown();
    executor.awaitTermination(1, TimeUnit.HOURS);
  }
}
```

目前的公平性问题：



使用虚拟线程：



说明：已修复，但是还没有合并到 jdk 19

优势

1. 非常简单的使用，直接怼线程池 1 即可

2. 可以支持超大规模的虚拟线程
3. 原生的并发包、网络、文件等对虚拟线程的支持，极大的提高了性能和未来的空间

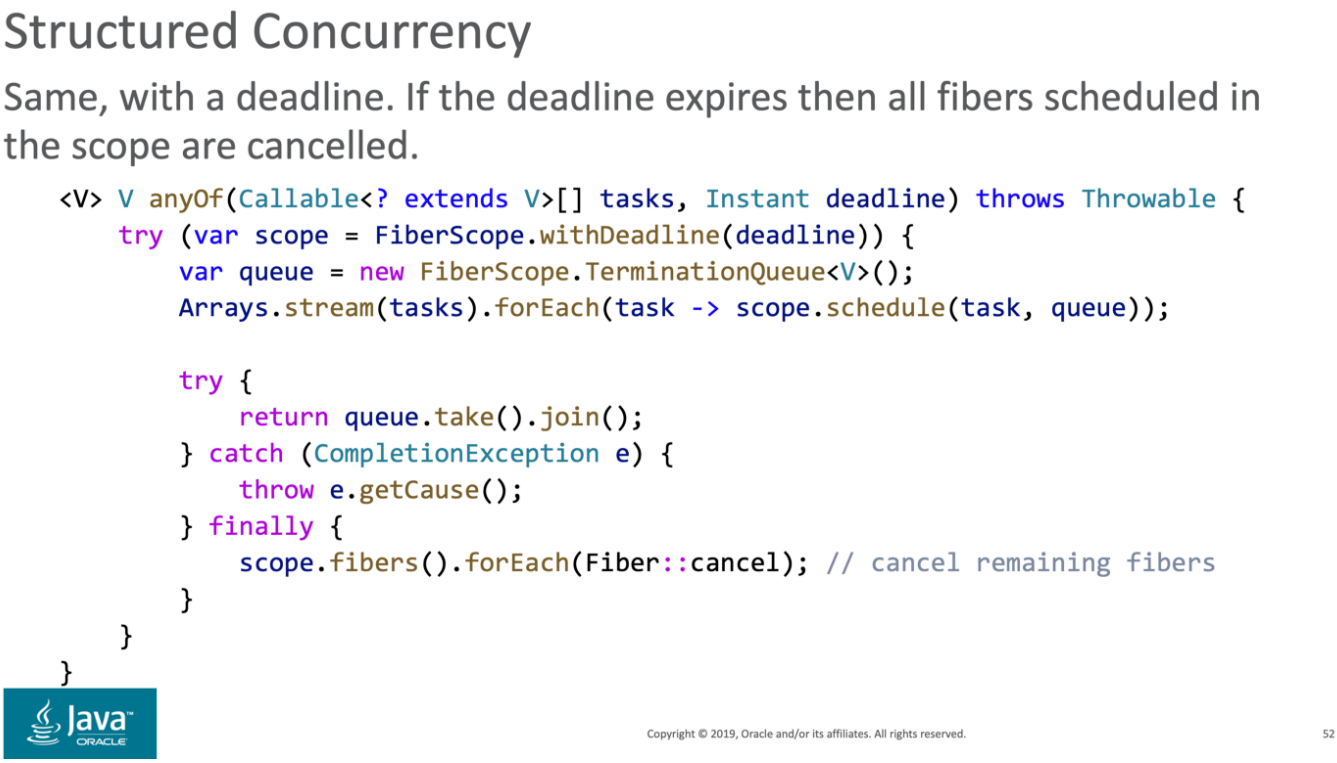
劣势

1. 原生的虚拟线程不如框架方便使用

10. 结构化并发

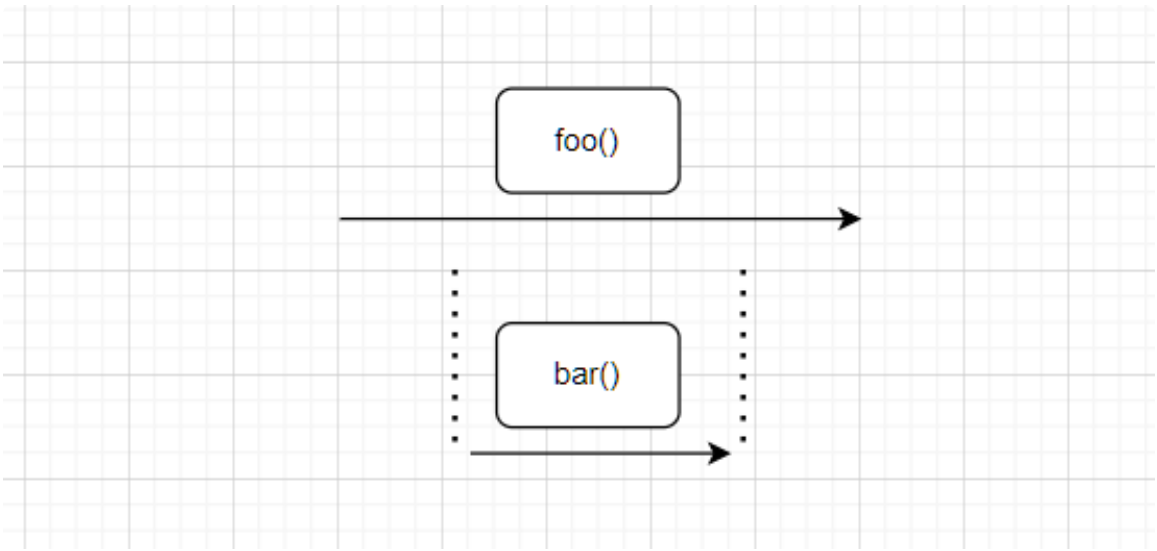
并发代码中生命周期的一致性、错误和结果的延续性等

https://openjdk.org/jeps/428



```java
<T> T race(List<Callable<T>> tasks, Instant deadline) throws ExecutionException {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<T>()) {
        for (var task : tasks) {
            scope.fork(task);
        }
        scope.joinUntil(deadline);
        return scope.result(); // Throws if none of the forks completed successfully
    }
}
```

```java
<T> List<T> runAll(List<Callable<T>> tasks) throws Throwable {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        List<Future<T>> futures = tasks.stream().map(scope::fork).toList();
        scope.join();
        scope.throwIfFailed(e -> e);  // Propagate exception as-is if any fork fails
        // Here, all tasks have succeeded, so compose their results
        return futures.stream().map(Future::resultNow).toList();
    }
}
```



Features

405: Record Patterns (Preview)
422: Linux/RISC-V Port
424: Foreign Function & Memory API (Preview)
425: Virtual Threads (Preview)
426: Vector API (Fourth Incubator)
427: Pattern Matching for switch (Third Preview)
428: Structured Concurrency (Incubator)

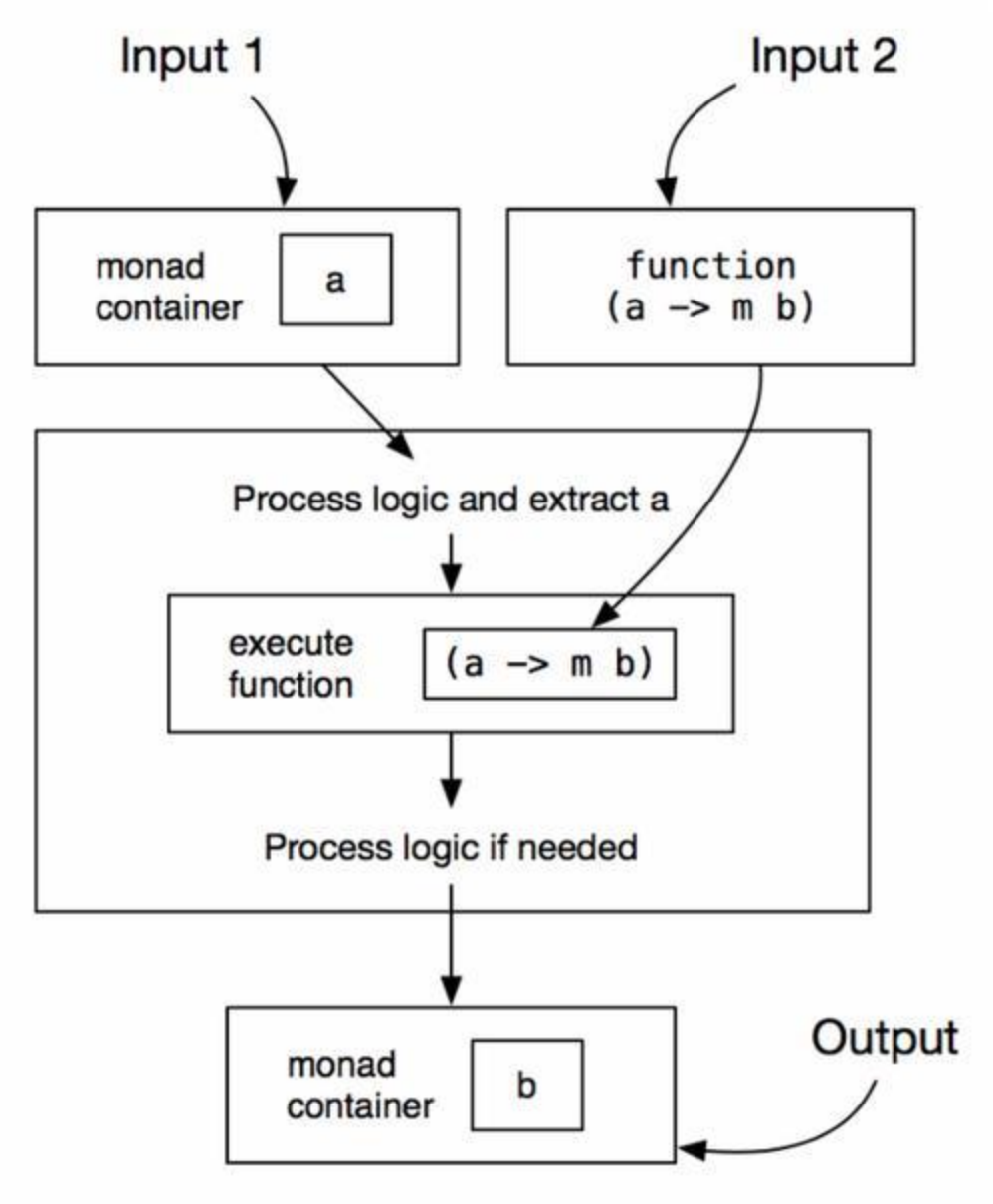
优势

1. 原生支持结构化并发，更好的处理并发、异常等，使用简单

劣势

1. API不如使用 Reactor 、RxJava 等框架丰富

11. 基于IOMonad



优势

1. 支持高纬度的优化、lazy、高度抽象
2. 性能好

劣势

1. 函数式有较高的理解成本

大部分情况用好 Java 自带的就可以了

代表

Cats-Effect、ZIO

个人认为：

高级用法：结合 IOMonad + 反应式流
普通用法：CompletionStage + 线程池 + BlockingQueue
未来的普通用法：CompletionStage + 虚拟线程池 + BlockingQueue

3. 场景和挑战

目前负责实时发布订阅服务，支持的场景有：

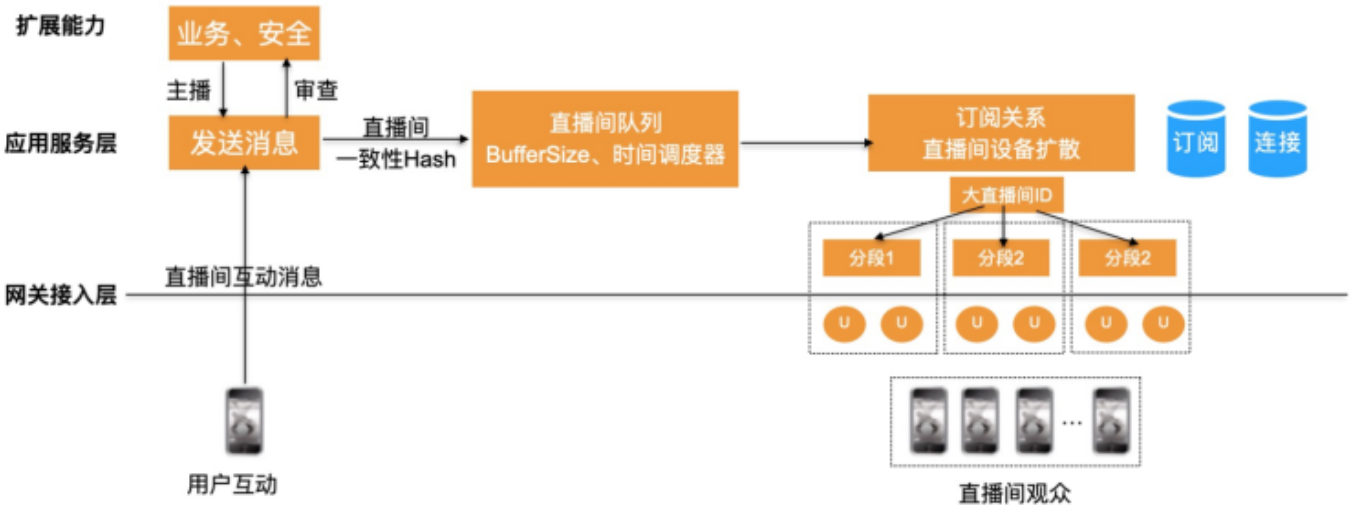
以及喵糖：

挑战

1. 海量 Topic 的伸缩性
2. 热点 Topic 的压力
3. 消息的差异化流控和优先级
4. 消息的实时性

4. 反应式流的应用

1.0 架构

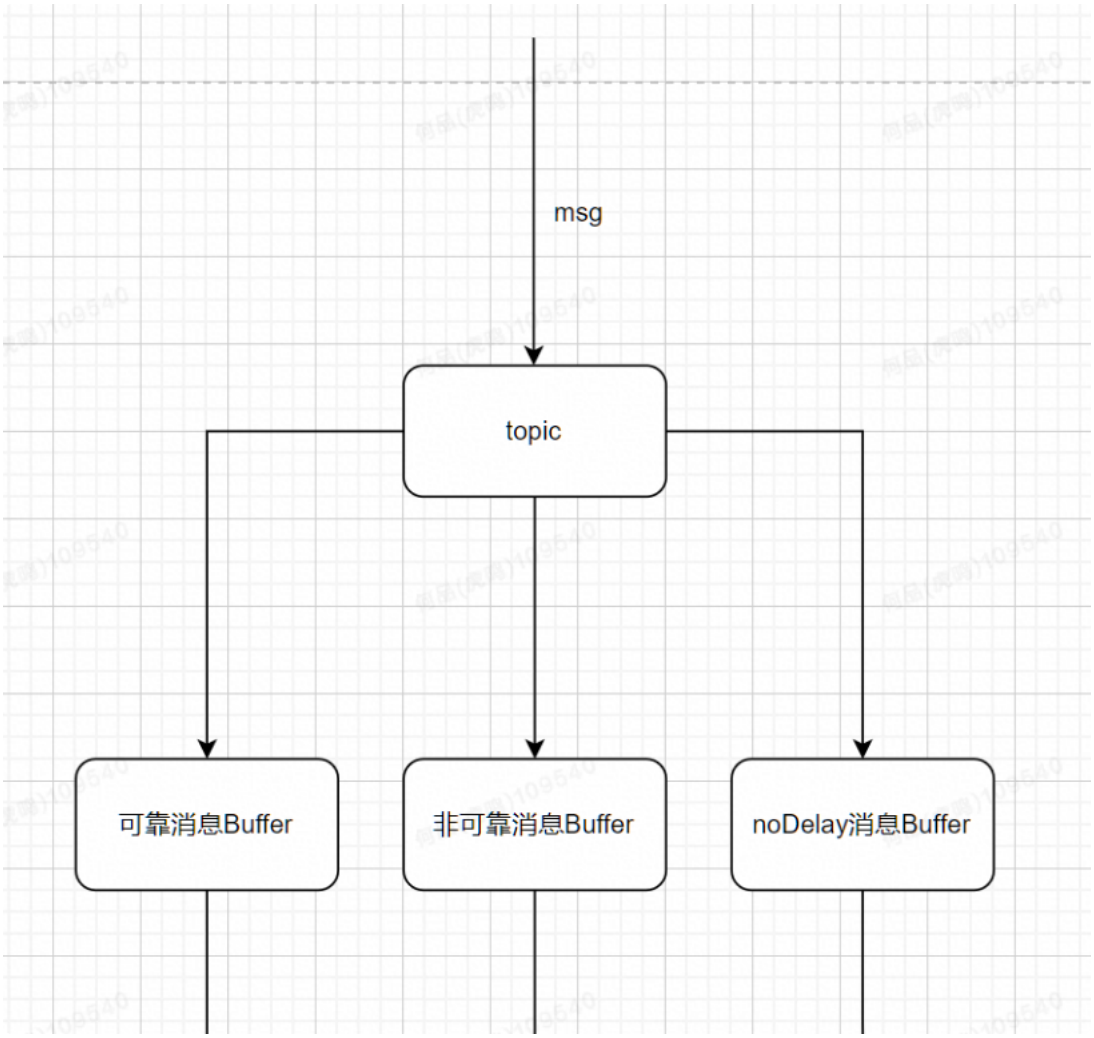


在 1.0 链路中我们核心使用了 RxJava 来支持相关能力, 后迁移到 Reactor， 遇到了很多问题：

1. 对机器资源的大量消耗、cpu高、实时性差
2. 调度有问题，造成消息挤压，从而冲击下游后，造成回压，FGC
3. 各种类型的消息分级粒度太粗、软隔离做的不彻底

问题1： 隔离粒度太粗

解法：进行更细粒度的拆分，拆分为多个队列（逻辑队列）



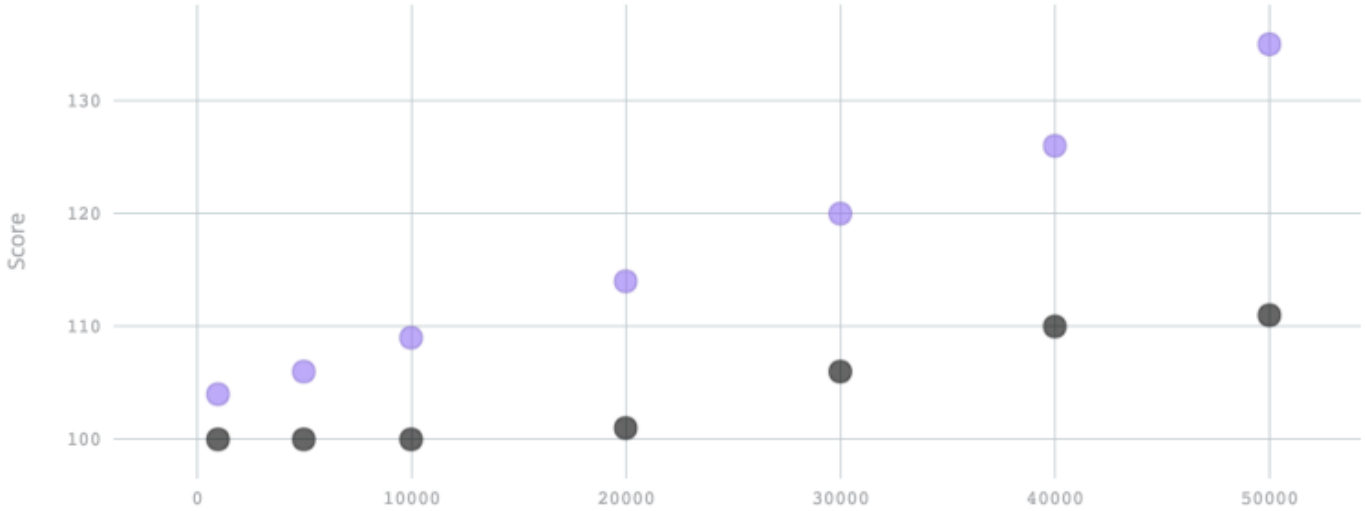
问题2：大规模逻辑队列调度问题

cpu高、调度时间片不准、冲击下游等等....

- Improve Flux.timeout efficiency：https://github.com/reactor/reactor-core/issues/2845 ，因为Scheduler 使用的是Java默认的，造成不支持海量房间的调度。
- Explore support for multiproducer Sinks.Many with heavy contention #2850，https://github.com/reactor/reactor-core/issues/2850 就是 cpu 打高。而我这边因为一致性hash到一台机器，就多个 hsf provider 线程会往 热门房间写消息。
- bufferTimeout 内部的同步关键字造成在热门直播间的锁。
- 不支持双流、多流多优先级合并
- 不支持流的流控，比如最多每秒通过多少个消息，当然可以配合transform操作符结合sentinel的sentinel-reactor-adapter 来进行。
- 自定义一个操作符的实现比较复杂，不需要自己手动处理各种同步和多线程的问题，感兴趣的可以参考下：https://github.com/reactor/reactor-core/blob/main/reactor-core/src/main/java/reactor/core/publisher/FluxBufferTimeout.java

这个类的实现。

ScheduledExecutorService vs Hashed Timer



解法：迁移到 Akka-Stream

- Akka-Stream 和时间相关的调度是基于 HashedWheelTimer 的，所以大量超时集的调度没有性能问题，换而言之，就是可以稳定调度大量的直播间。
- Akka-Stream 的 HashedWheelTimer 和执行线程池是分开的，不和 Reactor 一样混着的，不会相互影响。
- 支持多流按优先级合并，比如实现：“可靠消息和非可靠消息分离，优先处理可靠消息”，https://github.com/reactor/reactor-core/issues/2827 netflix 这边也有类似的需求。平替也很简单，把mergeComparing 改为 mergePrefered 即可。
- 支持无锁非阻塞的令牌桶, 不需要自己再写一个，使用throttle。
- 自定义实现一个处理节点也非常简单，文档也很齐全，只需要注册2个回调函数onPull和onPush，不需要额外处理任何的 线程、同步 和可见性问题。
- goupedWeightedWithin 比 Reactor 的 bufferTimeout 更加强大，可以实现按照“权重”和“个数”和“时间”的三重聚合逻辑，而 Reactor 只能实现基于“个数” 和 “时间” 的聚合逻辑，表现到结果上就是聚合效果更好了。
- 更少的使用同步关键字，性能更好

通知使用多个自定义 GraphStage 进行特殊的定义，

效果：

Before：

after：

机器为使用RxJava 时的 25%.

其他：2.0 架构更近一层

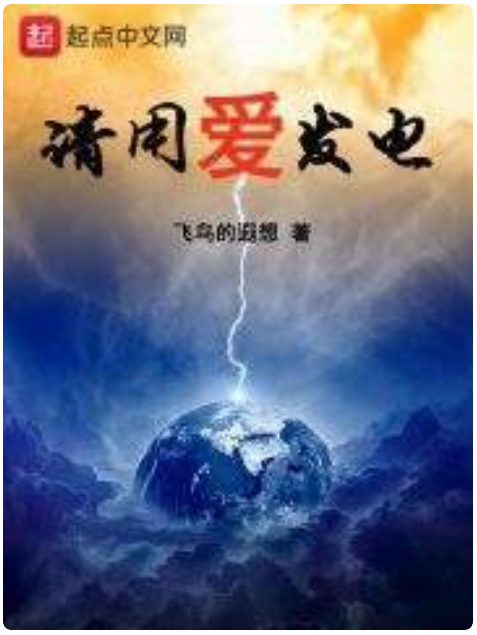
2.0 架构利用多团队协同，可以在提高实时性、稳定性的同时，进一步压缩成本，再降低40% 的机器，同时支持单房间 1500万设备 实时互动。

后续有机会再分享

5. 社区相关思考

1. 尽可能服用现在的轮子，避免重复造轮子
2. 欢迎贡献、Scala、Akka 等生态项目，你好我好大家好，早点下班


用愛發電大隊


起点中文网
请用爱发电
飞鸟的遐想 著