27. 图解多线程在支付系统中的应用案例及常见误区

大家好，我是隐墨星辰，深耕境内/跨境支付架构设计十余年。

今天花点时间简单聊下多线程在支付系统中的应用案例以及常见的误区。一方面在支付系统经常要使用多线程，每次面试也都会问多线程的问题，但90%以上的候选人都回答得不怎么好，另一方面是碰到过一些线上的坑需要避开。

1. 前言

在支付系统中，使用多线程和线程池是一种常见且有效的方式来提升系统的处理能力和响应速度。尽管网络上关于多线程的文章很多，但很少有文章能够精准地贴合线上系统的实际需求。今天的文章旨在结合支付系统的实际场景，探讨多线程技术的合理应用以及常见的误区。

文章内容聚焦在场景应用，具体参数和多线程原理，请大家参考官方文档即可。

2. 线程池的合理配置

对于线程池的配置，存在一个普遍的误解：即简单地根据任务类型（CPU密集型或IO密集型）来确定线程池的大小。比如网上有很多文章，都写了类似这样的说明：“对于CPU密集型任务，线程池大小设置为CPU核心数加1；对于IO密集型任务，则可以设置更多，比如CPU核心数的两倍”。

实际上，线程池的配置需要更加精细地考虑任务的具体特性、系统的响应时间要求以及资源的可用性。

- **CPU密集型任务**：对于完全在内存中运算的任务，建议将线程池大小设置为与CPU核心数相等，以最大化CPU的利用率，减少线程切换带来的损耗。

  比如数据加解密服务，纯计算，不依赖外部的IO资源。

- **IO密集型任务**：这类任务的关键在于合理预估系统并发需求和单个任务的平均耗时，从而计算出合适的线程池大小。公式可以简化为：线程池大小 = 系统预期最大并发数 * 单个任务平均耗时。

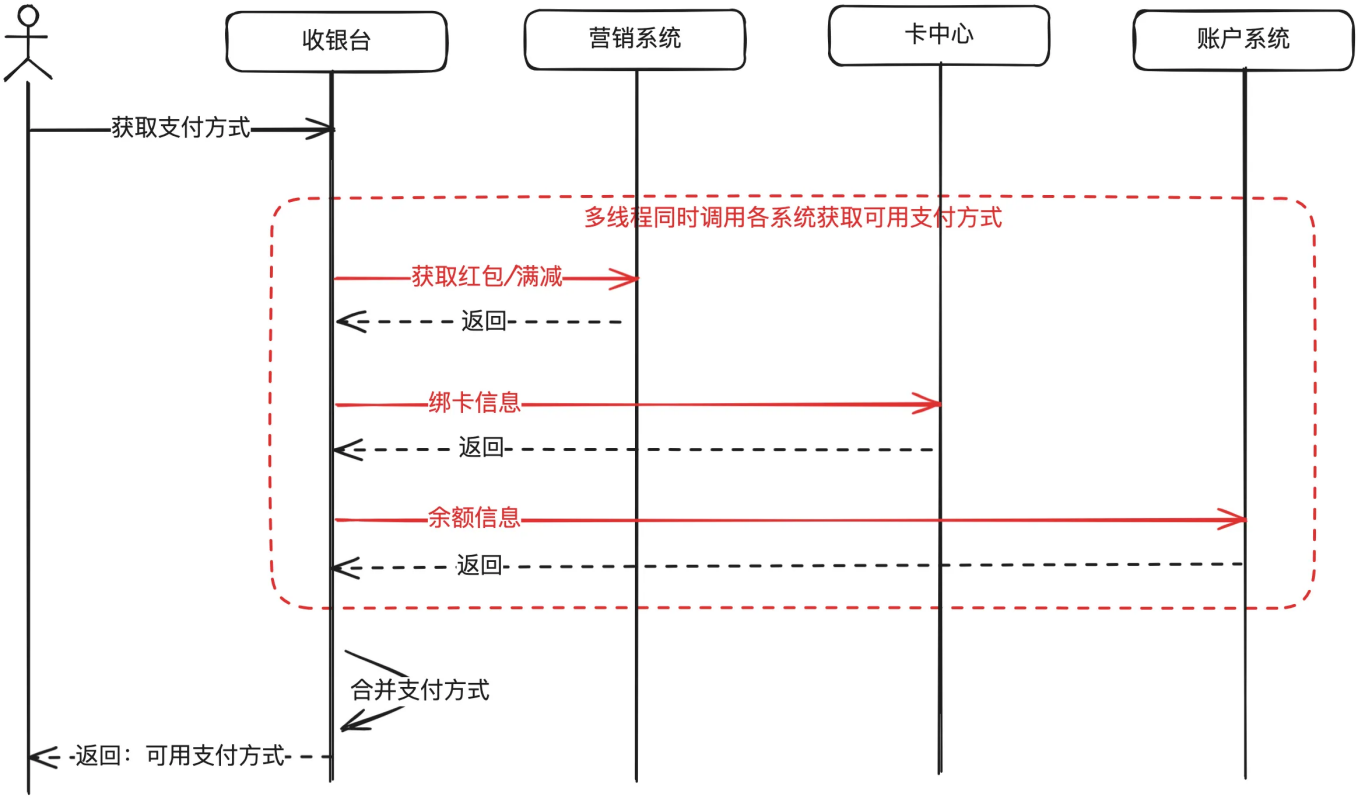
  比如支付系统对接了很多外部银行通道，这种涉及外部银行通道的交互通常需要几百到几千毫秒，就是属于典型的IO密集型任务。如果一个支付请求平均耗时500ms，并发需要有40QPS，那么线程池大小应该是：40 * 0.5 = 20。如果是一个请求平均耗时达到1000ms，同样需要有40QPS，那么线程池大小应该是：40 * 1 = 40。

这背后的原理也很简单：1）线程切换是需要消耗资源的。2）线程在等待IO资源时，CPU是空转的。

我们使用多线程的目的就是尽可能压榨CPU的运算能力，尽可能多的完成任务。如果是CPU密集型的，就减少线程切换，如果是IO密集型的，就在等待IO资源时，让出CPU给其它线程。

3. 典型案例及核心代码实现

3.1. 最短时间内获取尽可能多的可用支付方式



每个用户在支付系统中绑定了很多支付方式，比如不同的银行卡，还有内部的余额，红包等。每次渲染收银台之前，都需要去获取这些支付系统，汇总后展示给用户。

一种方式就是串行去获取，但这明显会影响用户体验。最优的方案当然是使用多线程并行获取。

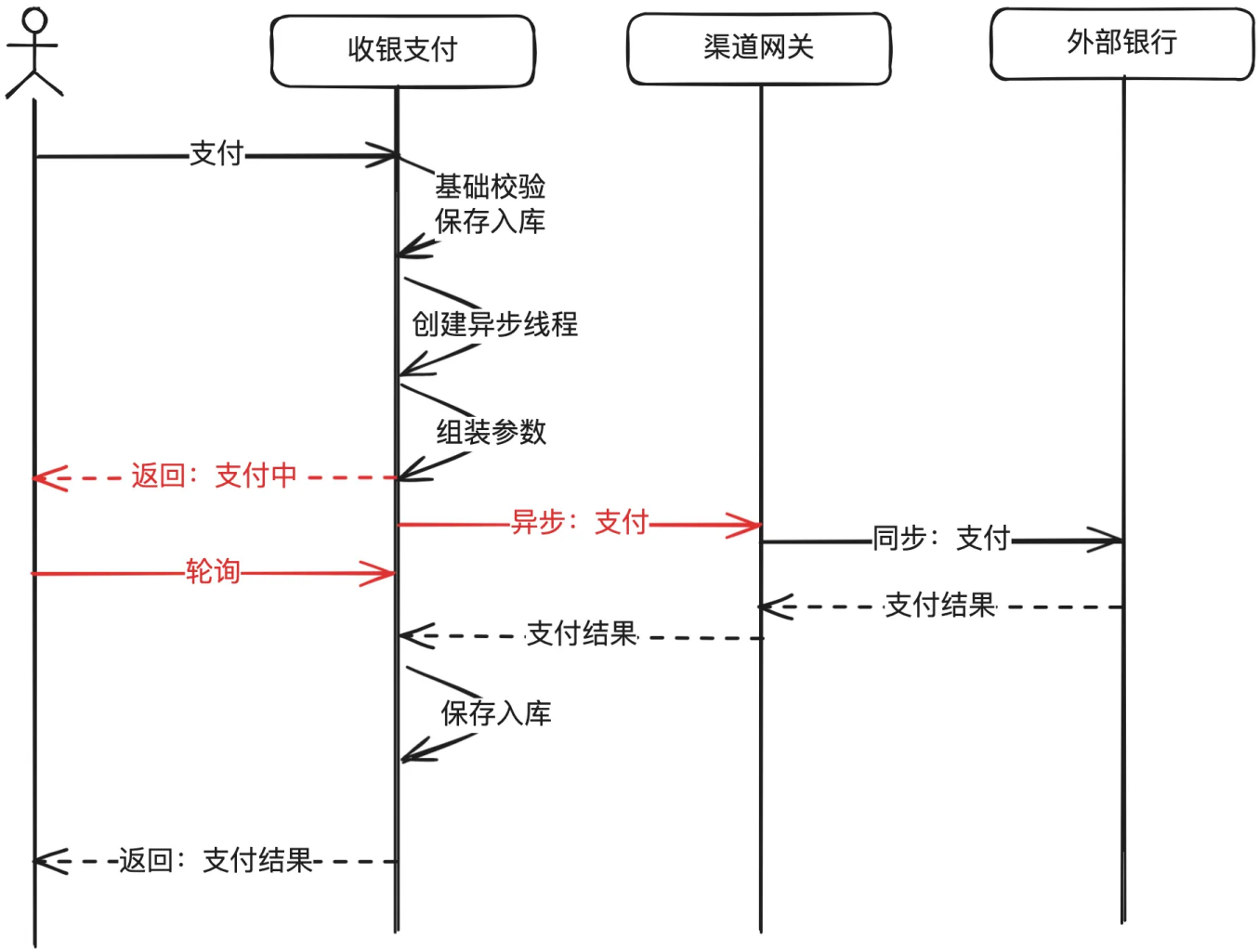
下面是一个JAVA伪代码示例，真实场景需要考虑超时、异常处理等。

```java
ExecutorService executor = Executors.newFixedThreadPool(20);

List<Callable<PaymentMethodTask>> tasks = new ArrayList<>();
// 填充任务列表
for (PaymentMethod method : methods) {
    tasks.add(() -> method.queryAvailableMethod());
}
try {
    List<Future<PaymentMethodTask>> results = executor.invokeAll(tasks);
    // 处理结果
} catch (InterruptedException e) {
    // 异常处理
}

```

3.2. 同步受理异步处理



用户提交支付请求后，如果是外部银行通道，通常耗时都需要几百到几千毫秒，如果全链路都是同步接口，那么整个系统的线程很快就被消耗完，且一旦外部银行出现响应慢的情况，极其容易出现雪崩现象。

所以我们在调用外部银行扣款时，通常都使用“同步受理异步处理”的方案。简单地说，就是先受理用户的请求，做基础校验，校验通过后，保存到DB，发起一个异步线程请求到外部银行，然后马上返回给用户，前端再发起定时轮询结果。

下面是一段JAVA伪代码，实际使用时需要加上各种异常处理。

```java
public class PaymentProcessor {
    private ExecutorService executor;
    private GatewayService gatewayService;

    public PaymentProcessor() {
        // 根据要求配置线程池：核心线程数20，最大线程数50，队列长度100
        this.executor = new ThreadPoolExecutor(20, 50, 60L, TimeUnit.SECONDS, new ArrayBlockingQueue<>(100));
    }

    public void pay(PaymentRequest request) {
        validate(request);
        save(request);

        // 异步执行任务：发送支付请求给外部银行
        executor.submit(() -> {
            try {
                gatewayService.sendToChannel(request);
            } catch (Exception e) {
                // 异常处理
            }
        });
    }
}
```

4. 常见误区

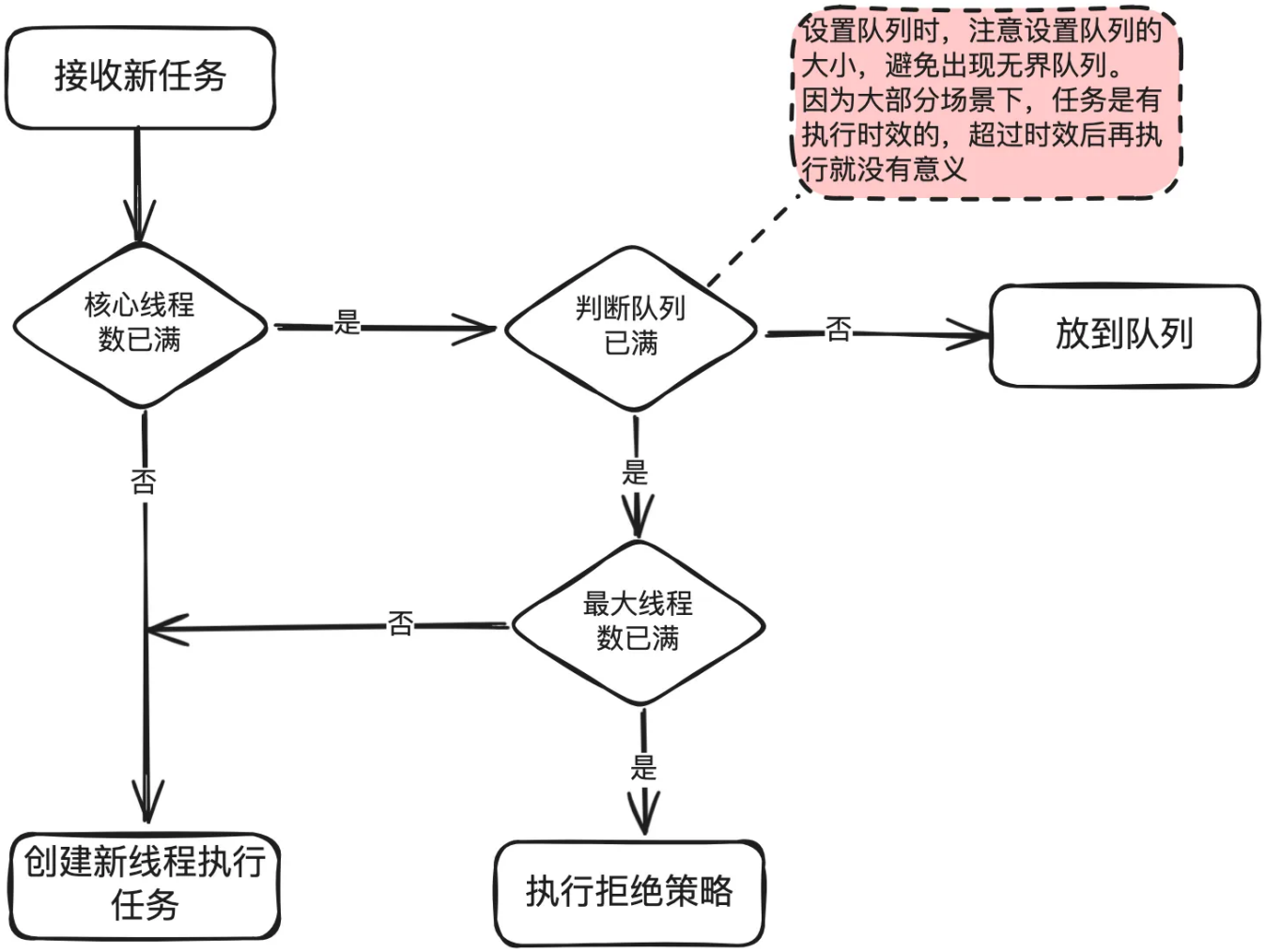
4.1. IO密集型任务线程池大小设置为CPU核数的2倍

前面第2节有说到，哪怕是IO密集型，我们也不能简单设置为CPU核数的2倍，我们仍然要考虑任务执行耗时，系统设计的最大并发数是多少等因数。

建议为：系统预期最大并发任务数 * 单任务平均耗时。

注意，这个耗时是指等待外部资源的耗时，不是CPU运算耗时。比如外发银行后，等待外部银行返回的过程，就是等待时间，基本不消耗CPU资源。

4.2. 为什么设置了最大线程数不生效



曾经有同学使用ThreadPoolExecutor，设置了核心线程数，最大线程数，但是线上出现很多超时未处理的任务，但是请求数没有超过最大线程数。排查很久才发现虽然设置了最大线程数，但是没有设置队列大小（LinkedBlockingQueue），那么它会默认为Integer.MAX_VALUE，这基本上可以认为是无界队列，也就是请求全部放到了队列中。

所以大家如果使用ThreadPoolExecutor来配置线程池，最好是根据自己的诉求，把参数设置完整，包括核心线程数，最大线程数，队列大小，拒绝策略等。比如有些业务超时后已经没有意义，那就把队列放小点，拒绝策略为直接拒绝。

具体的请参考JAVA官方文档。

4.3. 直接new线程

因为简单，有些同学喜欢直接new线程。的确，这种方式在简单场景下是没有问题的，但是复杂场景下是很容易出问题，且不好排查，建议不要养成这样的习惯。如果场景真的非常简单，也建议使用创建固定大小线程池来做，比如ExecutorService executor = Executors.newFixedThreadPool(n)。

5. 结束语

多线程在支付系统中的应用非常广泛，可以显著提高处理速度和系统吞吐量，但同时还需要我们合理配置线程池大小和策略，同时要从实际出发，避免直接照抄网上一些不符合实际情况的文章，避开一些常见的陷阱。

这是《图解支付系统设计与实现》专栏系列文章中的第（27）篇。

深耕境内/跨境支付架构设计十余年，欢迎关注并星标公众号“隐墨星辰”，和我一起深入解码支付系统的方方面面。

专栏系列文章PDF合集不定时更新，欢迎关注我的公众号“隐墨星辰”，留言“PDF”获取。

有个支付系统设计与实现讨论群，添加个人微信（yinmon_sc）备注666进入。

隐墨星辰 个人微信

10年顶尖境内/跨境支付公司架构经验

著有《图解支付系统设计与实现》

备注666进支付讨论群