架构实战营模块5

第5课：接口高可用

李运华

前阿里资深技术专家（P9）

教学目标

1. 掌握接口级别高可用设计的架构模式和技巧

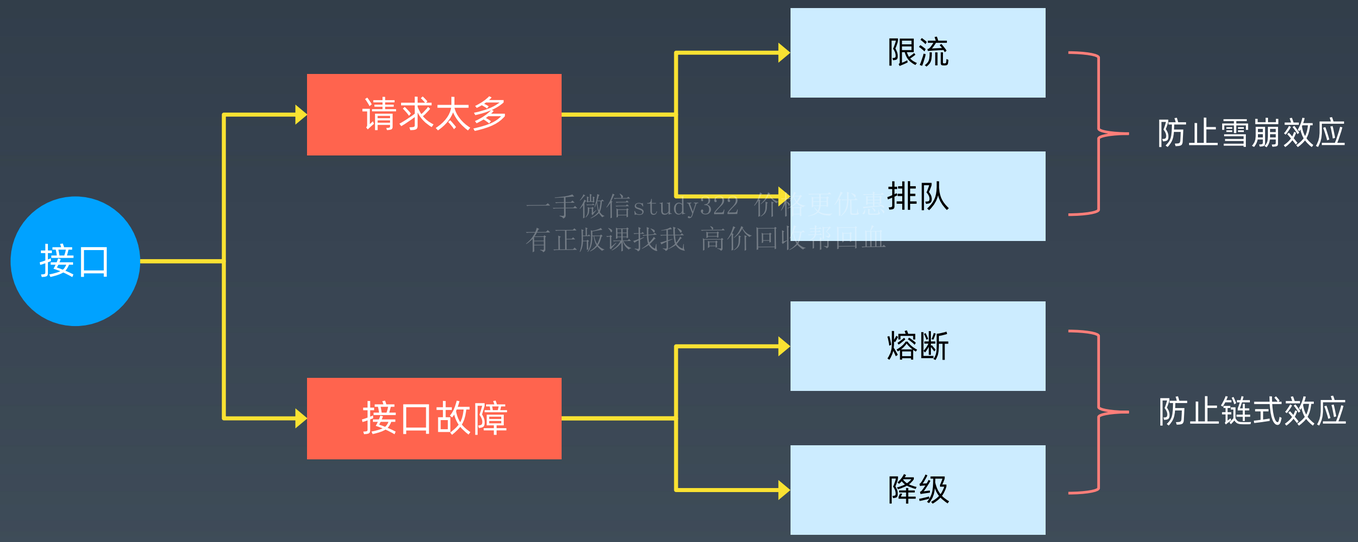
架构和代码共同决定系统质量！

目录

1 接口高可用整体框架

接口高可用整体框架

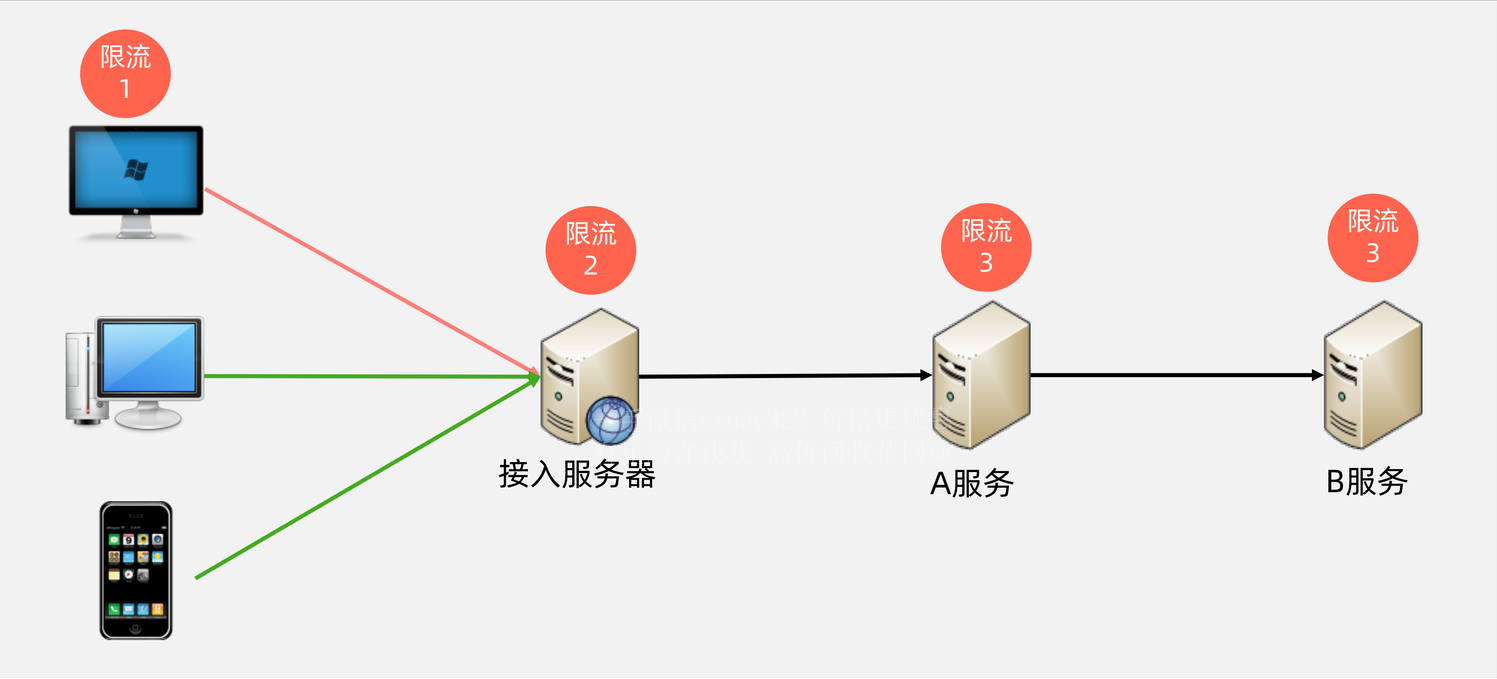
雪崩效应：请求量超过系统处理能力后导致系统性能螺旋快速下降。
链式效应：某个故障引起后续一连串的故障。



接口高可用架构本质上是“丢车保帅”策略，业务或者用户体验会部分有损！

2 限流

限流



用户请求全流程各个环节都可以限流：

1. 请求端限流：发起请求的时候就进行限流，被限流的请求实际上并没有发给后端服务器；
2. 接入端限流：接到业务请求的时候进行限流，避免业务请求进入实际的业务处理流程；
3. 服务限流：单个服务的自我保护措施，处理能力不够的时候丢弃新的请求。

限流具体实现方式

请求端限流

【常见手段】
1. 限制请求次数，例如按钮变灰）；
2. 嵌入简单业务逻辑，例如生成随机数。

【优缺点】
1. 实现简单；
2. 流量本地就控制住了；
3. 防君子不防小人（脚本）。

接入端限流

【常见手段】
1. 限制同一用户请求频率；
2. 随机抛弃无状态请求，例如限流浏览请求，不限流下单请求。

【优缺点】
1. 实现复杂；
2. 可以防刷；
3. 限流阈值可能需要人工判断。

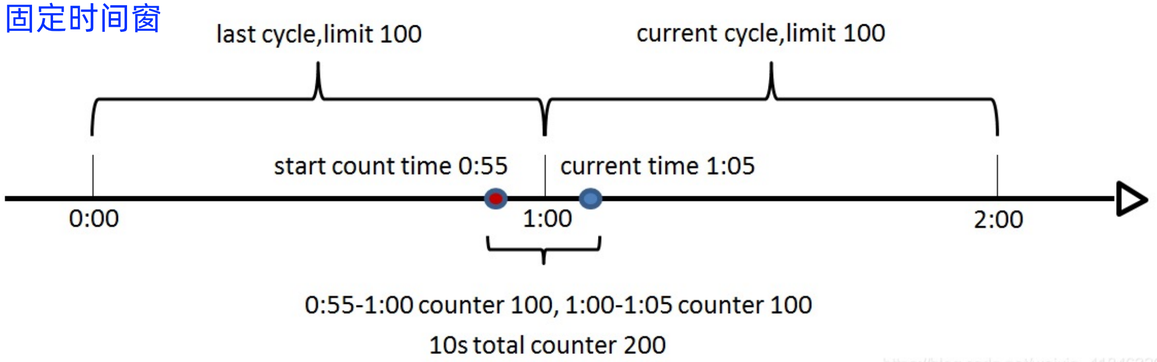
服务限流

【常见手段】
1. 根据处理能力，丢弃无法处理的请求。

【优缺点】
1. 实现简单；
2. 处理能力难以精准配置。

限流算法 - 固定 & 滑动 时间窗

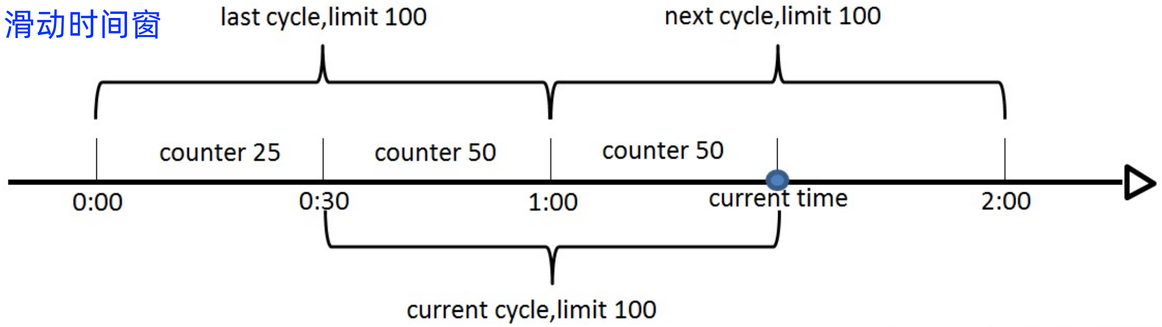
固定时间窗



【设计原理】

1. 统计固定时间周期内的请求量，超过阈值则限流；
2. 存在临界点问题，如图中的红蓝两点对应的时间范围。

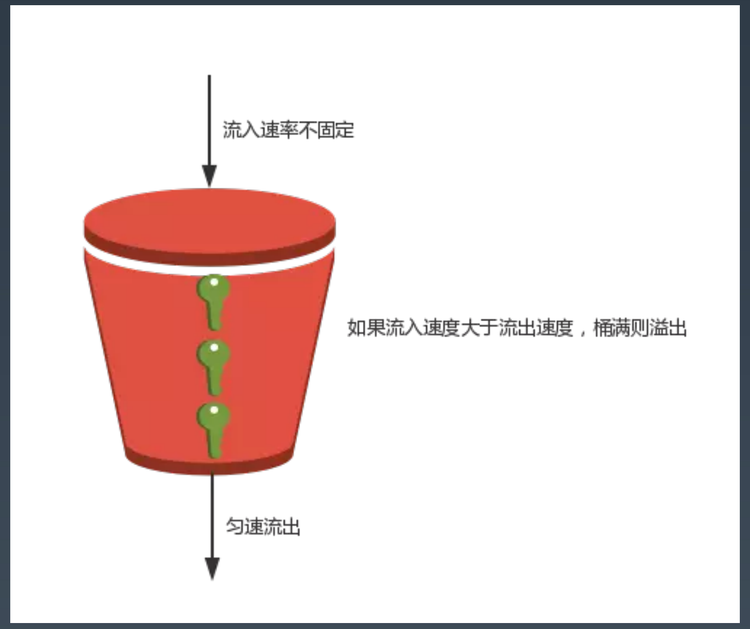
滑动时间窗



【设计原理】

1. 统计滑动时间周期内的请求量，超过阈值则限流；
2. 判断比较准确，但实现稍微复杂。

限流算法 - 漏桶



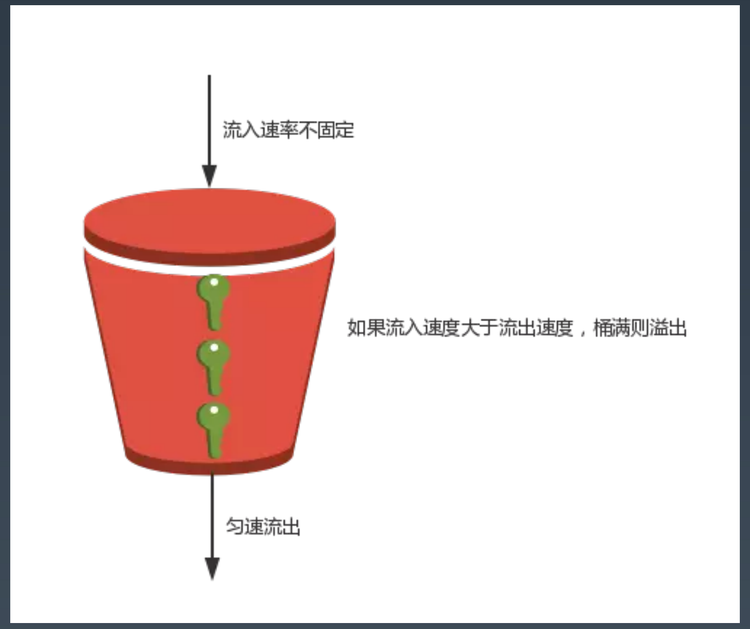
【基本原理】
请求放入“桶”（消息队列等），业务处理单元（线程/进程/服务)从桶里拿请求处理，桶满则丢弃新的请求。

【技术本质】
总量控制，桶大小是设计关键。

【优缺点】
1. 桶大小动态调整比较困难，例如 Java BlockingQueue；
2. 无法控制流出速度（处理速度）；
3. 突发流量时丢弃的请求较少。

【应用场景】
瞬时高并发流量，例如0点签到，整点秒杀。

漏桶算法变种 - 写缓冲(Buffer)



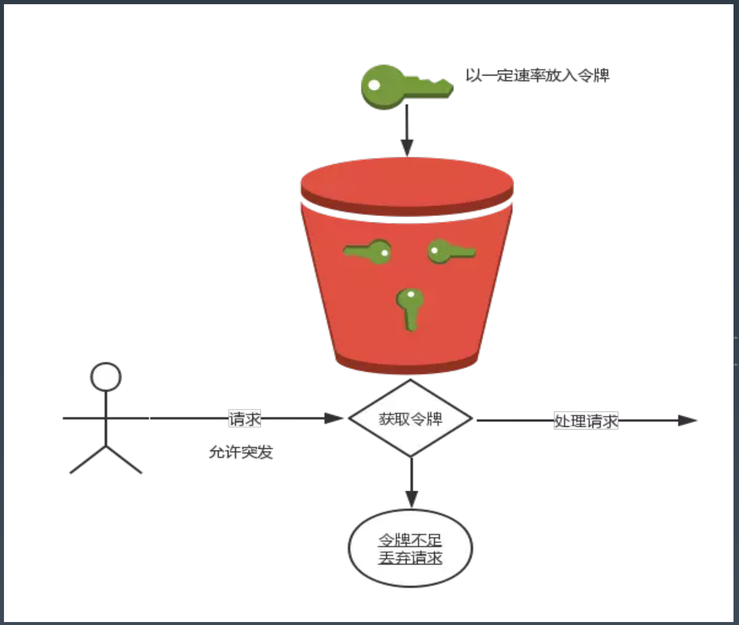
【基本原理】
如果漏桶的容量无限（例如用 Kafka 消息队列），则漏桶可以用来做写缓冲。

【技术本质】
同步改异步，缓冲所有请求，慢慢处理。

【应用场景】
高并发写入请求，例如热门微博评论。

为什么看微博的请求可以丢弃，而写评论请求却全部缓冲起来？

限流算法 - 令牌桶



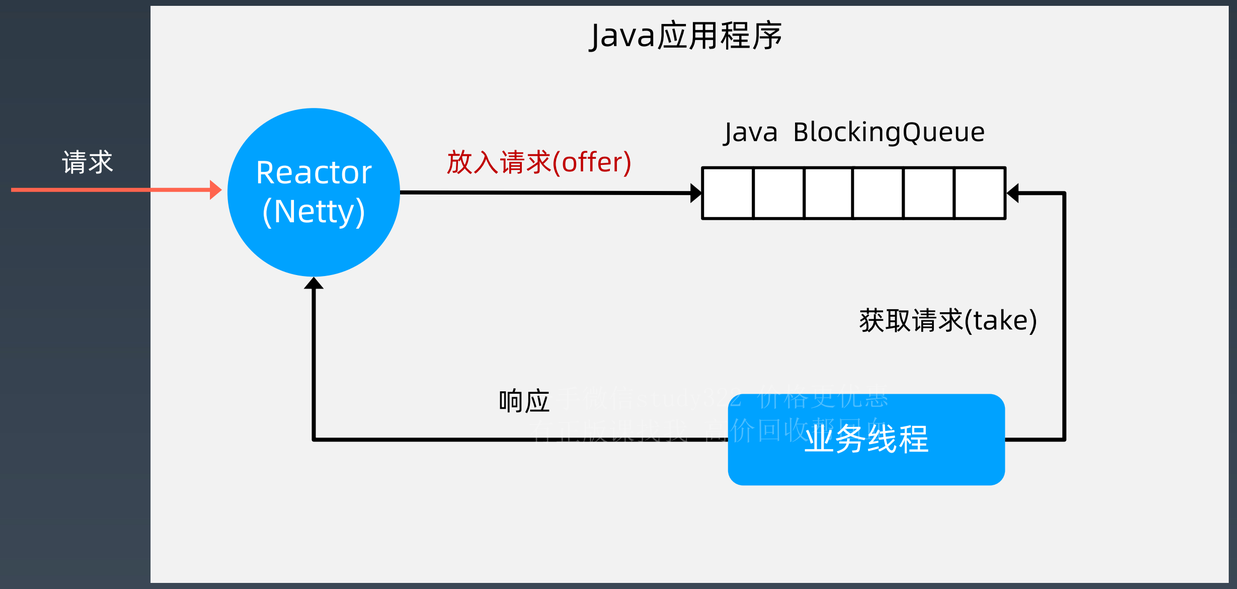
【基本原理】
某个处理单元按照速率将令牌放入“桶”（消息队列等），业务处理单元收到请求后需要获取令牌，获取不到就丢弃请求。

【技术本质】
速率控制，令牌产生速度是设计关键。

【优缺点】
1. 可以动态调整处理速度；
2. 突发流量的时候可能丢弃很多请求；
3. 实现相对复杂。

【典型应用场景】
1. 控制访问第三方服务的速度；
2. 控制自己的处理速度。

Java 限流的漏桶算法简单示例

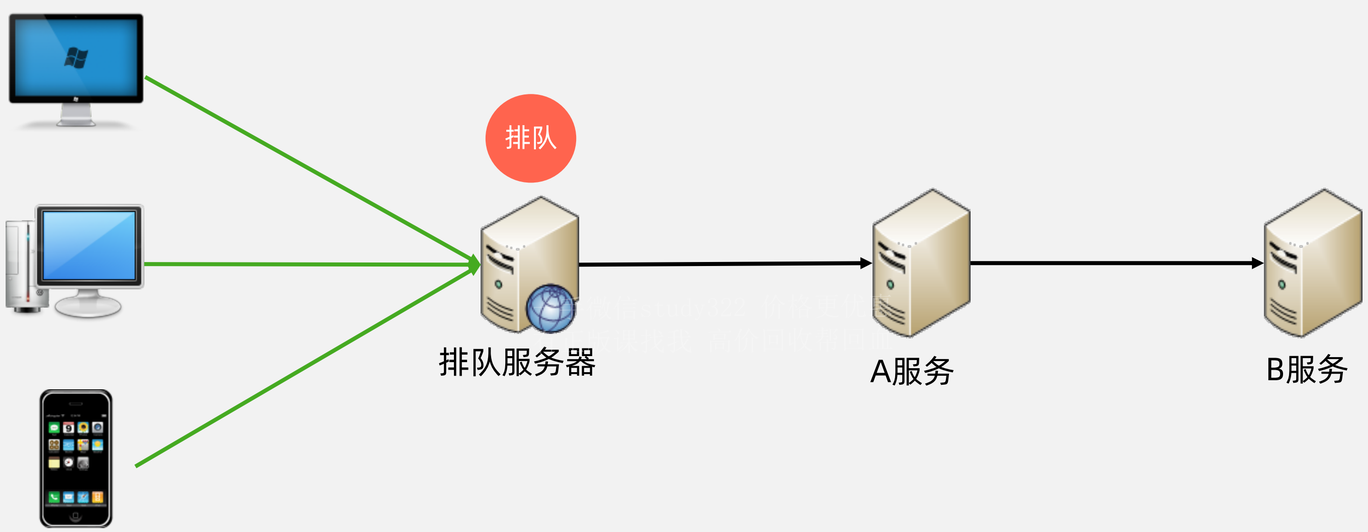


【设计关键】
1. 业务线程和 IO 线程分离，通过队列传递请求；
2. BlockingQueue 的长度配置，太长没作用，太短浪费。

如果是 Tomcat、SpringBoot 这类框架怎么办？

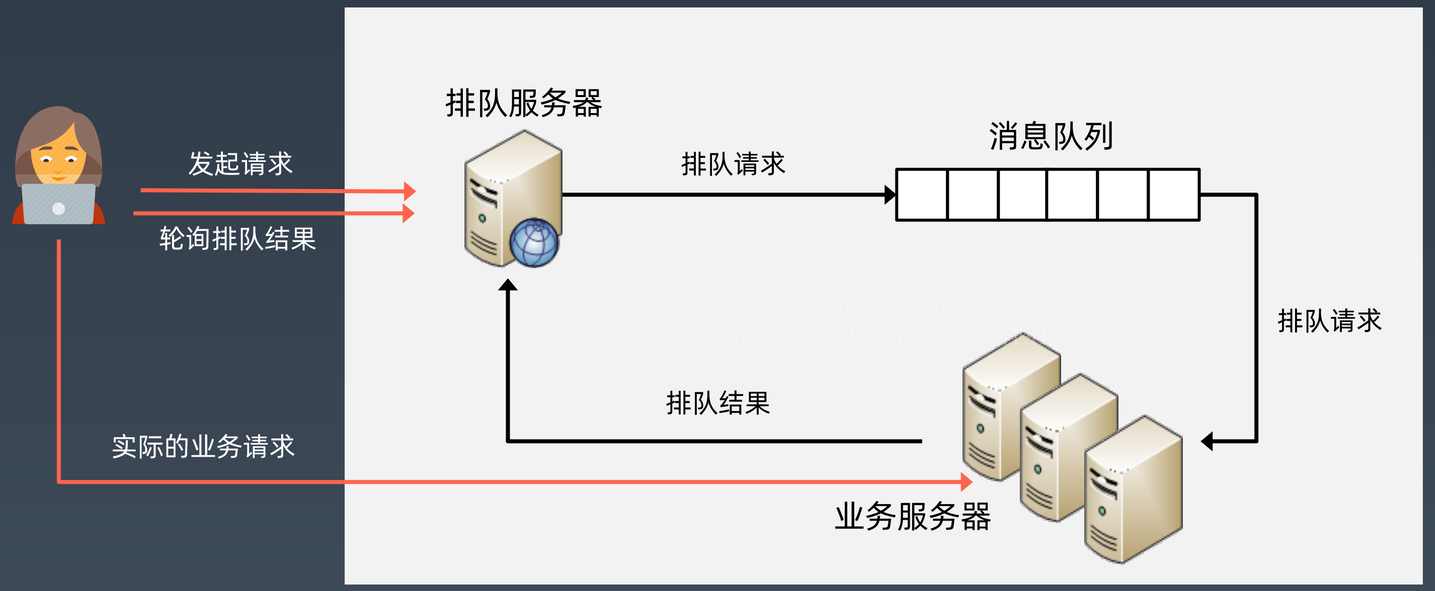
3 排队

排队



基本原理：收到请求后并不同步处理，而是将请求放入队列，系统根据能力异步处理。
技术本质：请求缓存+ 同步改异步 + 请求端轮询。
应用场景：秒杀、抢购。

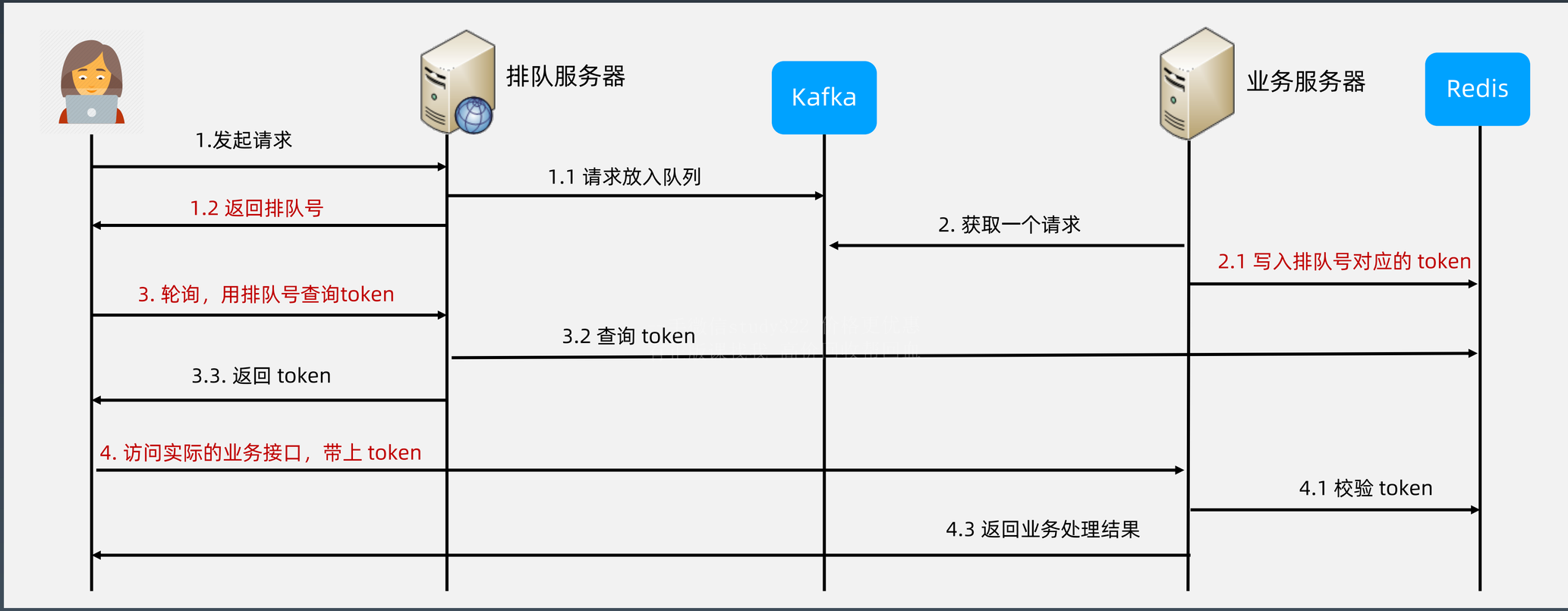
排队的架构示意图



【设计关键】

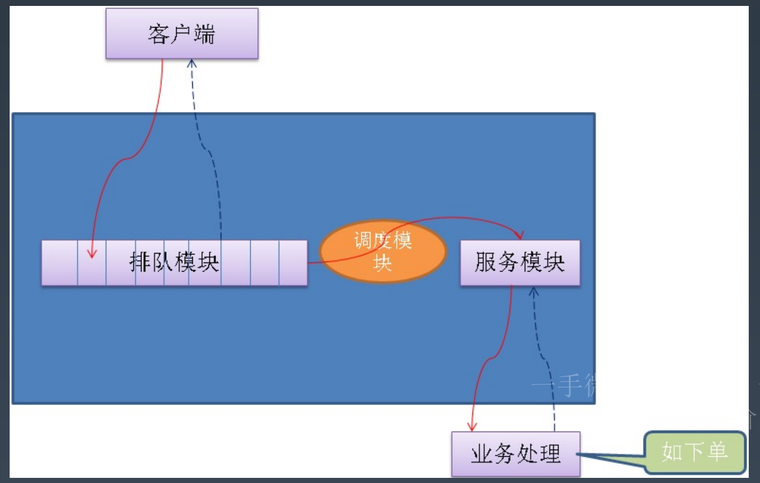
1. 如何设计异步处理流程；
2. 如何保证用户体验（前端、客户端交互）。

排队的具体实现方案示例



为什么要用 token，直接用排队号不就可以了么？

1号店双十一秒杀排队



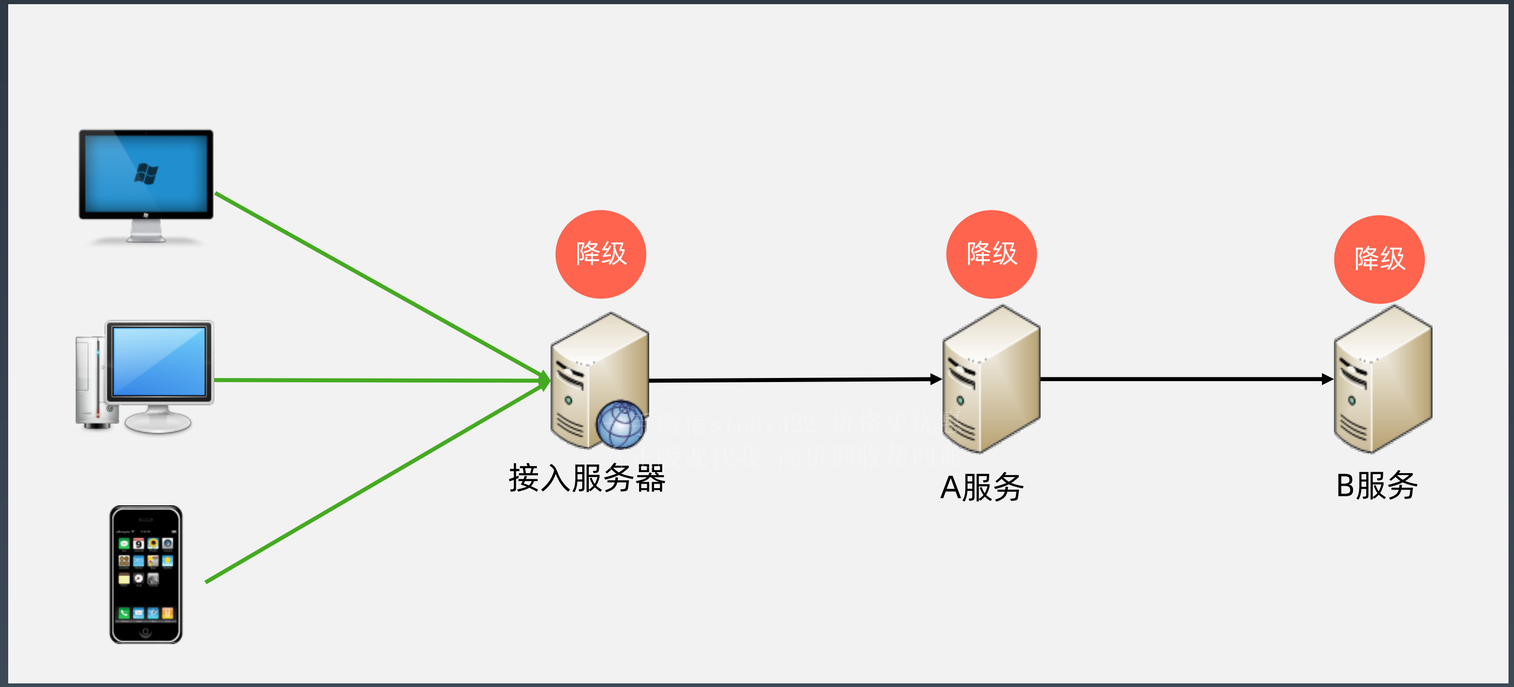
排队模块：
负责接收用户的抢购请求，将请求以先入先出的方式保存下来。每一个参加秒杀活动的商品保存一个队列，队列的大小可以根据参与秒杀的商品数量（或加点余量）自行定义。

调度模块：
负责排队模块到服务模块的动态调度，不断检查服务模块，一旦处理能力有空闲，就从排队队列头上把用户访问请求调入服务模块。

服务模块：
是负责调用真正业务处理服务，并返回处理结果，并调用排队模块的接口回写业务处理结果。
参考链接

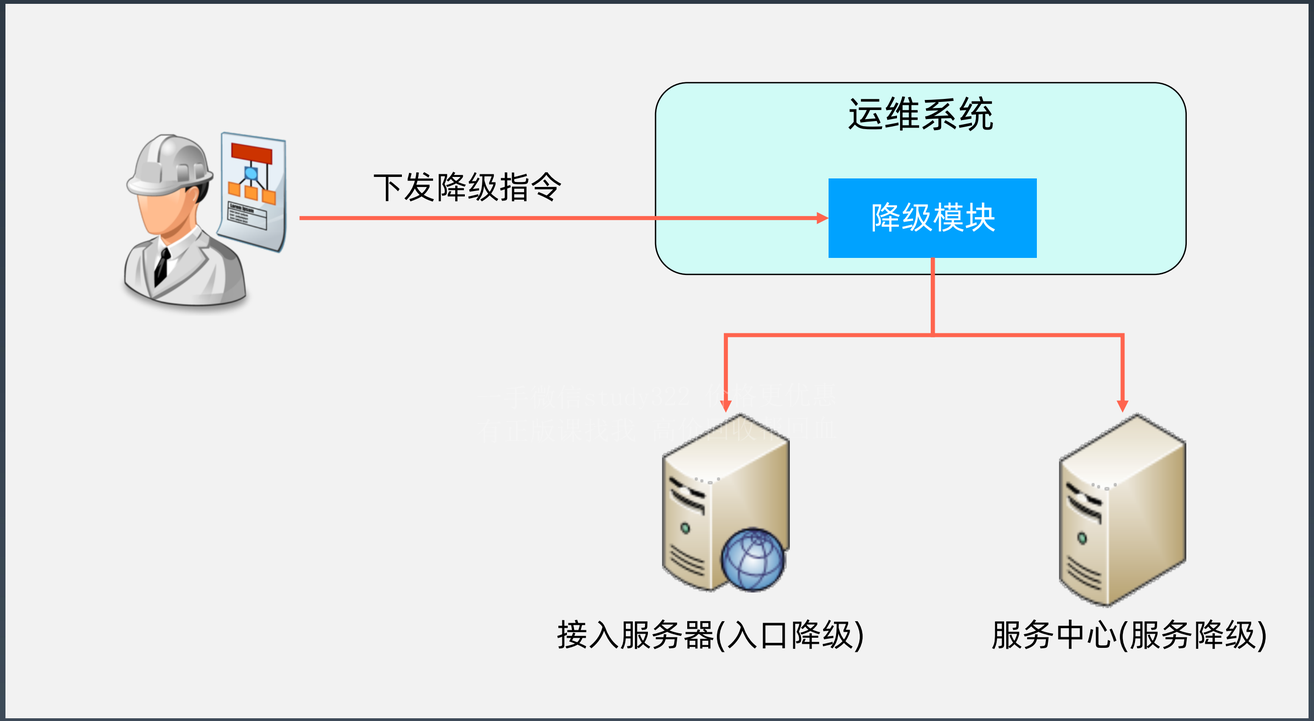
4 降级

降级



基本原理：直接停用某个接口或者 url，收到请求后直接返回错误（例如 HTTP 503）。
应用场景：故障应急，通常将非核心业务降级，保住核心业务，例如降级日志服务、升级服务等。

降级架构实现

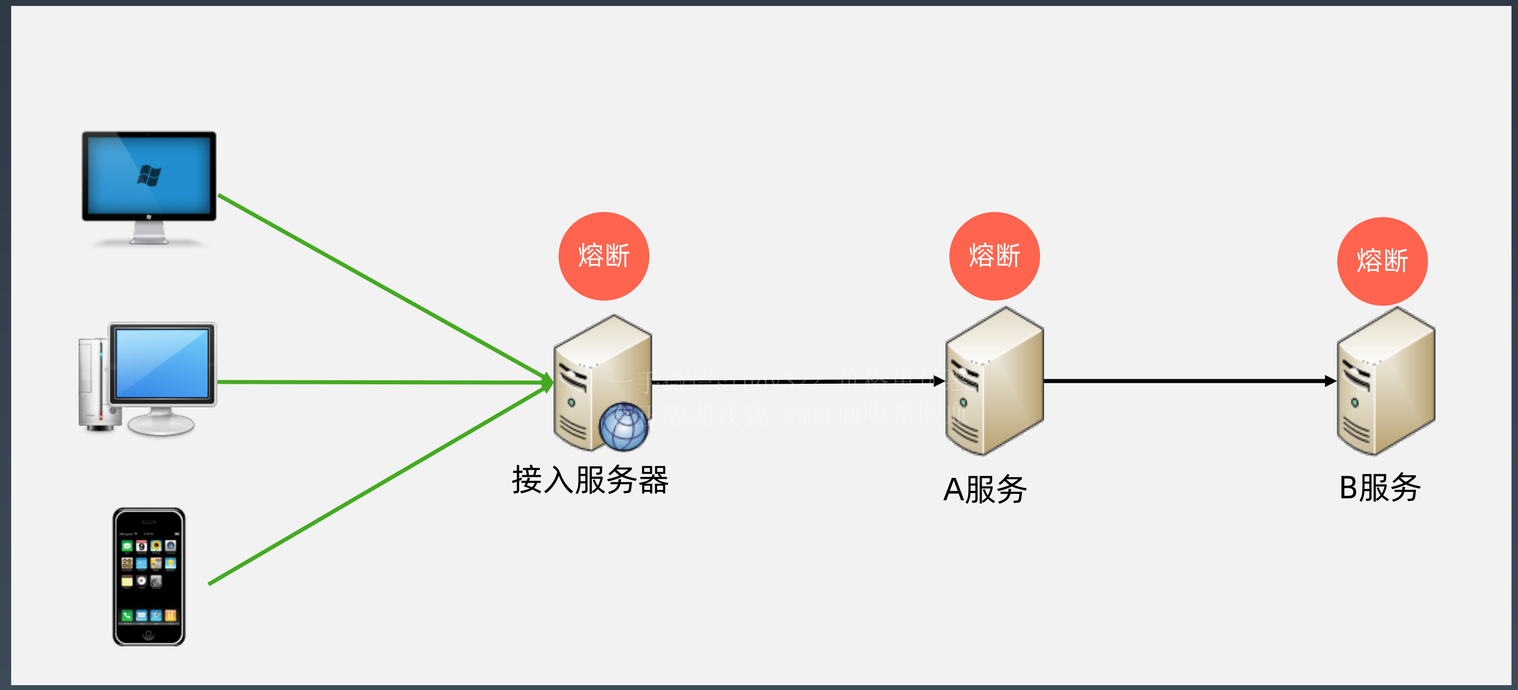


设计要点：

1. 独立系统操作降级，可以是独立的降级系统，也可以是嵌入到其它系统的降级功能；
2. 人工判断，人工执行，不要信 AIOps 之类的噱头。

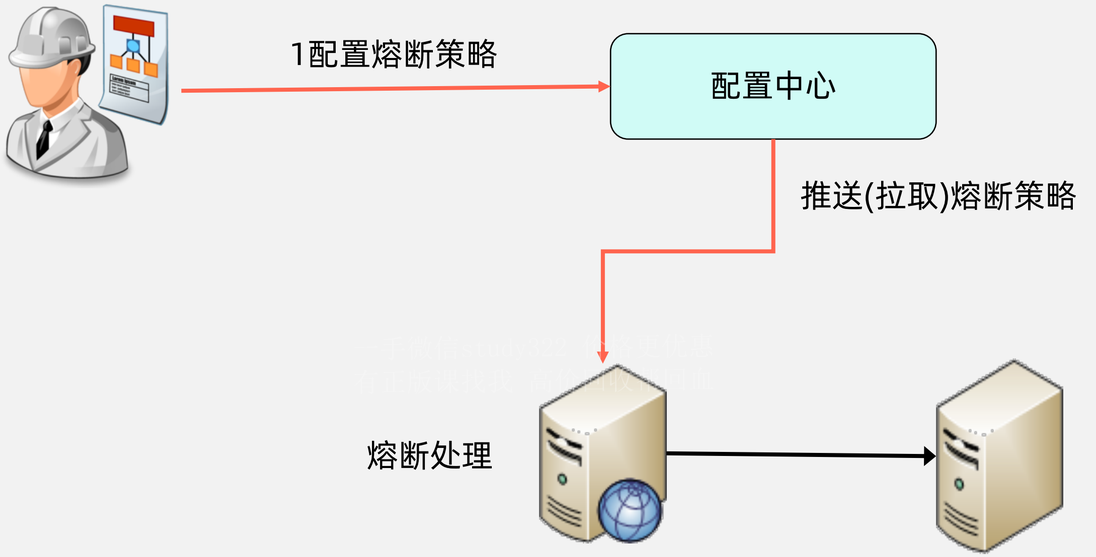
5 熔断

熔断



基本原理：下游接口故障的时候，一定时期内不再调用。
应用场景：服务自我保护，防止故障链式效应。

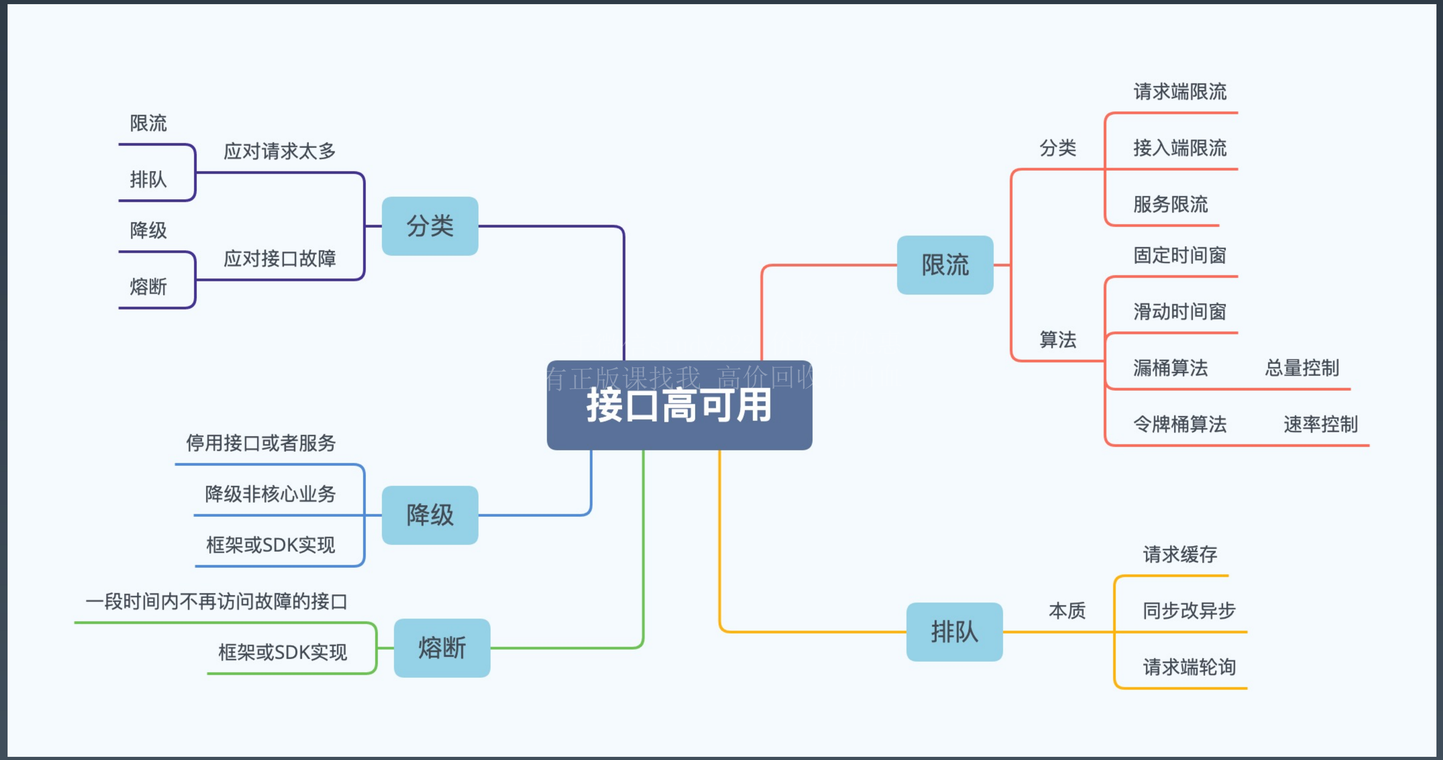
熔断架构实现



【实现细节】

1. 可以通过配置中心，也可以通过配置文件来配置熔断策略；
2. 熔断处理一般由框架或者 SDK 提供，例如 Dubbo + Hystrix；
3. 熔断策略一般按照失败次数、失败比例、响应时长等来确定。

本节思维导图


接口高可用
分类
限流
排队
应对请求太多
降级
熔断
应对接口故障
限流
分类
请求端限流
接入端限流
服务限流
算法
固定时间窗
滑动时间窗
漏桶算法
总量控制
令牌桶算法
速率控制
降级
停用接口或者服务
降级非核心业务
框架或SDK实现
熔断
一段时间内不再访问故障的接口
框架或SDK实现
排队
本质
请求缓存
同步改异步
请求端轮询

随堂测验

【判断题】

1. 限流是后端架构需要做的，和前端客户端无关。
2. 如果想保护下游依赖系统，可以用令牌桶算法进行限流。
3. 排队需要将用户同步请求改为异步请求，用户操作步骤增多，体验肯定会下降。
4. 降级需要人工判断是否降级和下发降级指令进行降级。
5. 熔断主要是框架或者 SDK 实现，架构设计能做的不多。

【思考题】

架构设计和代码共同决定系统质量，那么谁的影响更大一些？

Q&A

茶歇时间

八卦，趣闻，内幕……

THANKS