四、面试参考池

a.Java基础

- 多线程，线程池的参数，实际使用场景
- 锁，多种锁机制，结合具体场景
- 容器，HashMap，LinkedList，TreeMap

b.JVM的熟悉

- GC机制
- 内存模型
- 调参．B
- 定位问题的方式和案例．B
- 可达性分析是什么，怎么做，那些是root节点

c.框架中间件的使用和理解

- MQ，存储，实现，消费，性能，可靠性，延迟．B
- Redis，zset、跳表、数据结果、过期机制、淘汰策略、备份、线程模型
- 注册中心，dubbo zk，eureka，consule，发现机制
- 熔断、降级、限流机制，sentienl，hystrix　B
- 定时调度　A
- springboot spring mvc的请求处理流程，容器扩展点
- es 深度分页，优化，复杂查询语句，索引，分片　B
- 分布式锁
- 负载均衡．B
- dubbo

d.数据库/缓存

- 联合索引
- 索引原理，b+和hash
- SQL考察
- 事务
- 分库分表 B
- 缓存的过期和淘汰机制，多级缓存的数据一致性 B

e.设计模式

设计模式找熟悉并且在目标项目可能用到或聊到的

不要一个个遍历设计模式，从一个模式的使用，深入下去

f.代码题

1.阻塞队列的实现 A

2.延迟队列的实现 A

3.LRU本地缓存的实现 B

4.数组三个数和为指定目标数值的三个数

5.topN

6.排序算法的一种

7.一万以内的质数

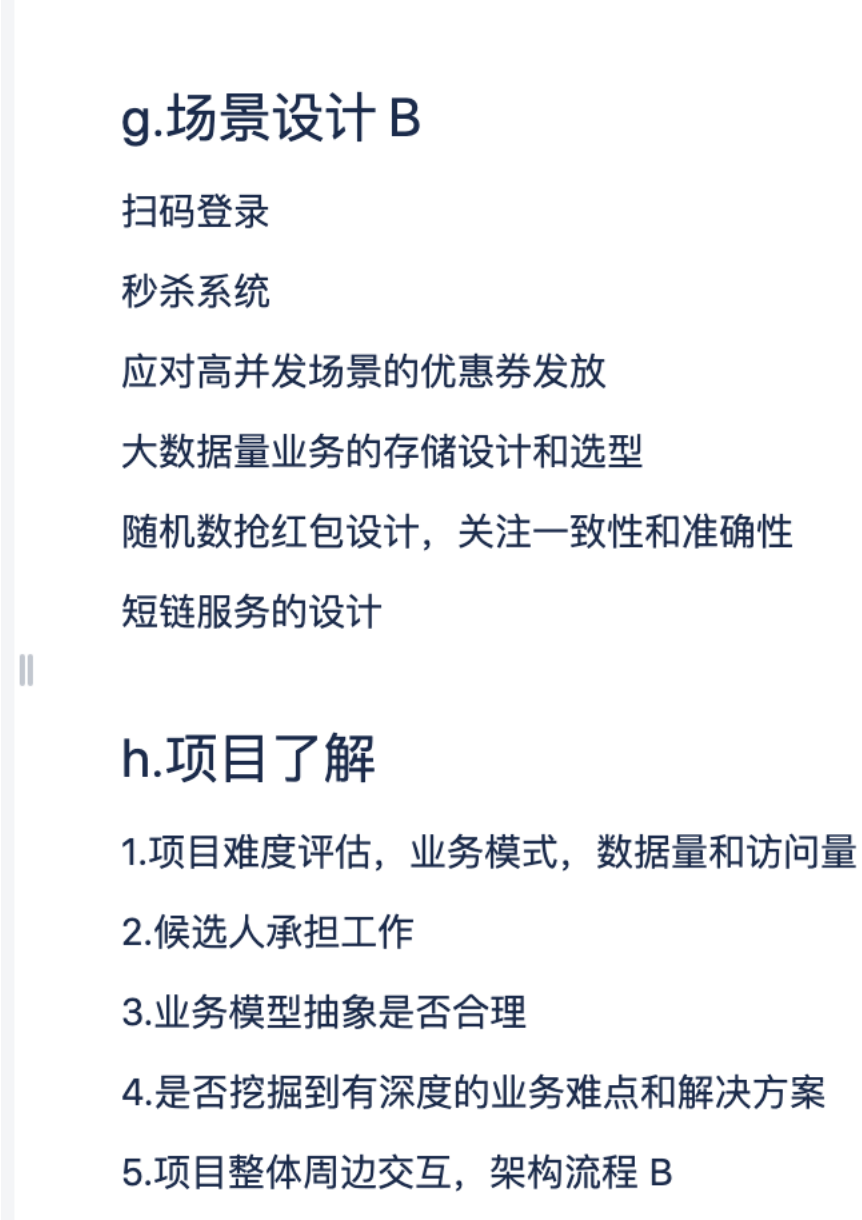
8.二叉树遍历

9.单链表翻转

10.二分查找

11.两个链表公共节点

12.字符串查找



Java基础
多线程
https://github.com/xbox1994/Java-Interview/blob/master/MD/Java%E5%9F%BA%E7%A1%80-%E5%A4%9A%E7%BA%BF%E7%A8%8B.md

corePoolSize - 池中所保存的线程数，包括空闲线程。

maximumPoolSize-池中允许的最大线程数。

keepAliveTime - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

unit - keepAliveTime 参数的时间单位。

workQueue - 执行前用于保持任务的队列。此队列仅保持由 execute方法提交的 Runnable任务。

threadFactory - 执行程序创建新线程时使用的工厂。

handler - 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序。

ThreadPoolExecutor是Executors类的底层实现。

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.**newFixedThreadPool**

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

4.**newScheduledThreadPool**

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

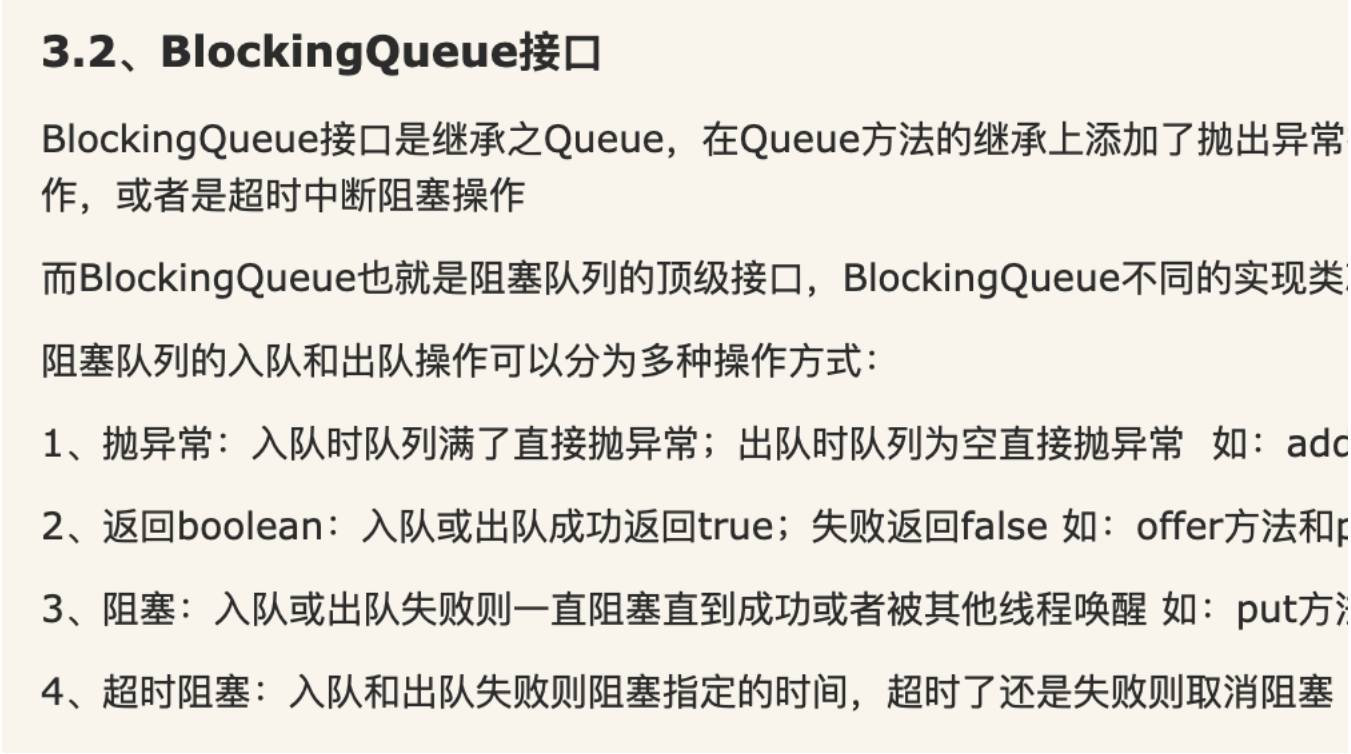
基本思想：

- 高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换。
- 并发不高、任务执行时间长的业务要区分开：
  - IO密集型的任务，因为IO操作并不占用CPU，可以加大线程池中的线程数目，让CPU处理更多的业务
  - CPU密集型任务，线程池中的线程数设置得少一些，减少线程上下文的切换。
- 并发高、业务执行时间长，在于整体架构的设计，能否使用中间件对任务进行拆分和解耦。

1.实现方式:Thread、Runnable、Callable
2.线程可以获得更大的吞吐量，但是开销很大，线程栈空间的大小、切换线程需要的时间，所以用到线程池进行重复利用，当线程使用完毕之后就放回线程池，避免创建与销毁的开销。
3.通信方式:

等待通知机制 wait()、notify()、join()、interrupted()
并发工具synchronized、lock、CountDownLatch、CyclicBarrier、Semaphore

容器&集合
HashMap并发情况下造成死循环的原因 https://www.cnblogs.com/wfq9330/p/9023892.html
https://hadyang.github.io/interview/docs/java/collection/HashMap/
https://hadyang.github.io/interview/docs/java/collection/Concurrenthashmap/
https://hadyang.github.io/interview/docs/java/collection/BlockQueue/
TreeMap底层原理使用红黑树
https://www.cnblogs.com/jackion5/p/11173721.html
阻塞队列



实现原理：通过可重入锁ReenTrantLock+Condition 来实现多线程之间的同步效果
优先队列:小顶堆的数据结构

leftNo = parentNo*2+1

rightNo = parentNo*2+2

parentNo = (nodeNo-1)/2

https://www.cnblogs.com/Elliott-Su-Faith-change-our-life/p/7472265.html

GC机制
https://ricstudio.top/archives/jvm_gc_knowledge
https://ricstudio.top/archives/jvm_memory_structure
https://www.jianshu.com/p/76959115d486
GC调参
http://xstarcd.github.io/wiki/Java/JVM_GC.html
https://github.com/vipcolud/monitor
https://www.cnblogs.com/QG-whz/p/9647614.html
https://cloud.tencent.com/developer/article/1346305
https://blog.csdn.net/weiquanaishiyao/article/details/83578999
https://blog.csdn.net/hellozhxy/article/details/80487097

了解G1
G1的第一篇paper（附录1）发表于2004年，在2012年才在jdk1.7u4中可用。oracle官方计划在jdk9中将G1变成默认的垃圾收集器，以替代CMS。为何oracle要极力推荐G1呢，G1有哪些优点?
首先，G1的设计原则就是简单可行的性能调优
开发人员仅仅需要声明以下参数即可：

-XX:+UseG1GC -Xmx32g -XX:MaxGCPauseMillis=200

其中-XX:+UseG1GC为开启G1垃圾收集器，-Xmx32g 设计堆内存的最大内存为32G，-XX:MaxGCPauseMillis=200设置GC的最大暂停时间为200ms。如果我们需要调优，在内存大小一定的情况下，我们只需要修改最大暂停时间即可。
其次，G1将新生代，老年代的物理空间划分取消了。

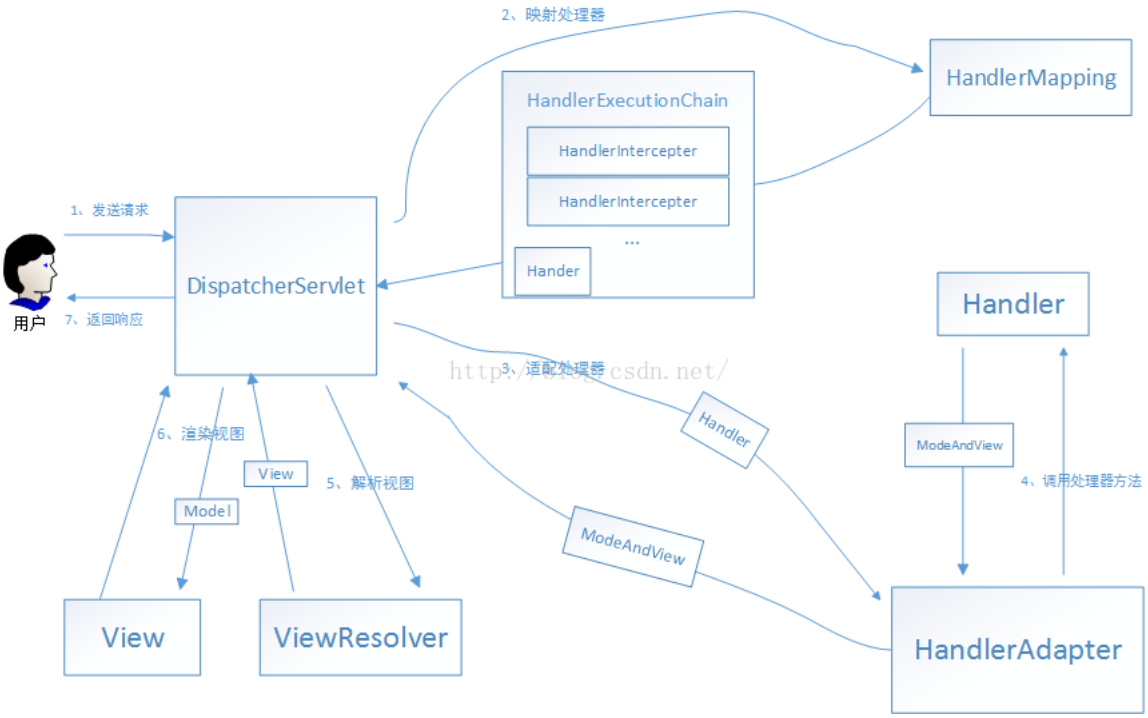
这样我们再也不用单独的空间对每个代进行设置了，不用担心每个代内存是否足够

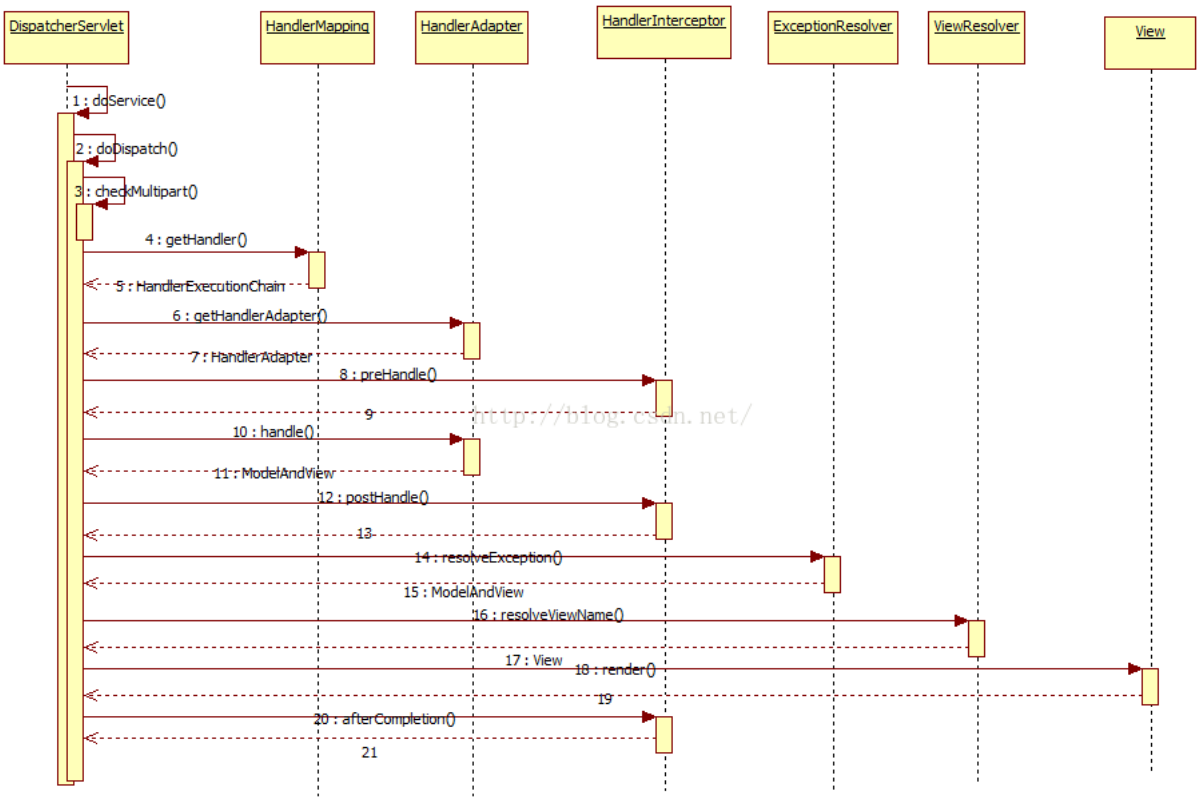
中间件
Consul强一致性(C)带来的是：gossip协议
服务注册相比Eureka会稍慢一些。因为Consul的raft协议要求必须过半数的节点都写入成功才认为注册成功
Leader挂掉时，重新选举期间整个consul不可用。保证了强一致性但牺牲了可用性。
Eureka保证高可用(A)和最终一致性：
服务注册相对要快，因为不需要等注册信息replicate到其他节点，也不保证注册信息是否replicate成功
当数据出现不一致时，虽然A, B上的注册信息不完全相同，但每个Eureka节点依然能够正常对外提供服务，这会出现查询服务信息时如果请求A查不到，但请求B就能查到。如此保证了可用性但牺牲了一致性。
其他方面，eureka就是个servlet程序，跑在servlet容器中; Consul则是go编写而成。

ES深度分页
https://www.cnblogs.com/1ning/p/10132703.html
https://www.cnblogs.com/jpfss/p/10815172.html

Search After实时滚动查询
https://blog.csdn.net/ctwy291314/article/details/82754652

Spring mvc处理流程





组件	说明
DispatcherServlet	本质上是一个 HttpServlet，Servlet 容器会把请求委托给 入口，负责协调各个组件工作
Handler	处理器，本质上是由实现 Controller 接口的类、实现 Htt @RequestMapping 的方法等封装而成的对象
HandlerMapping	内部维护了一些 <访问路径, 处理器> 映射，负责为请求找
HandlerAdapter	处理器的适配器。Spring 中的处理器的实现多变，比如用 现 HttpRequestHandler 接口，也可以用 @RequestMap 导致 Spring 不知道怎么调用用户的处理器逻辑。所以这里 器去调用处理器的逻辑
ViewResolver	用于将视图名称解析为视图对象 View。
View	在视图对象用于将模板渲染成 html 或其他类型的文件。比 成 html。

分布式锁
https://www.javazhiyin.com/31246.html
注意： Redis 从2.6.12版本开始 set 命令支持 NX 、 PX 这些参数来达到 setnx 、 setex 、 psetex 命令的效果。
文档参见： http://doc.redisfans.com/string/set.html

NX： 表示只有当锁定资源不存在的时候才能 SET 成功。利用 Redis 的原子性，保证了只有第一个请求的线程才能获得锁，而之后的所有线程在锁定资源被释放之前都不能获得锁。
PX： expire 表示锁定的资源的自动过期时间，单位是毫秒。具体过期时间根据实际场景而定

Redis 从2.6.0开始通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值，
文档参见： http://doc.redisfans.com/script/eval.html

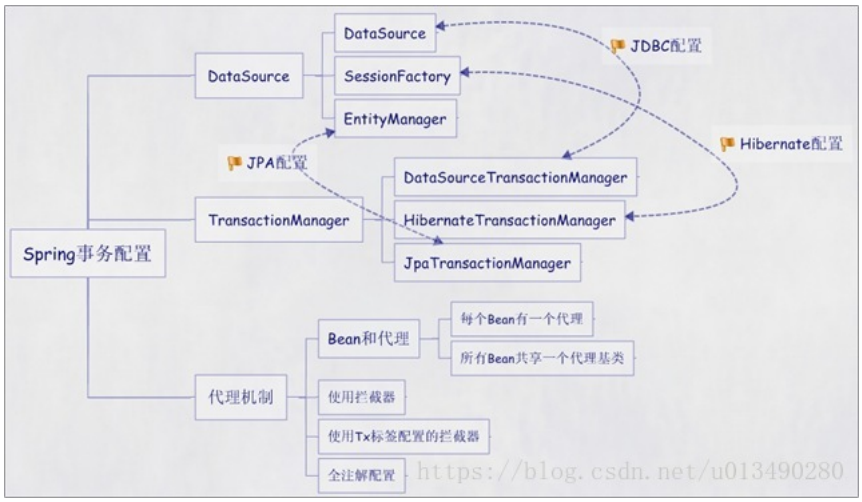
```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

dubbo
https://dubbo.apache.org/zh-cn/index.html

SQL
https://www.jianshu.com/p/3f27a6dced16
创建一张总总表

```
create table total(
select a.s_id as s_id,a.s_name as s_name,a.s_age as s_age,a.s_sex as s_sex,
b.c_id as c_id,b.score as score,c.t_id as t_id,d.t_name as t_name
from student a
left join
score  b on a.s_id=b.s_id
left join
course c on b.c_id=c.c_id
left join
teacher d on c.t_id=d.t_id
);
select * from total;
```

事务



分库分表的本质
https://blog.csdn.net/qq_36625757/article/details/90477131
分表：单表是数据达到几百万甚至上千万，SQL执行的速度就会变慢
分库：一般我们经验而言，最多支撑到并发 2000，一定要扩容了，而且一个健康的单库并发值你最好保持在每秒 1000 左右，不要太大

	分库分表前	分库分表
并发支撑情况	MySQL 单机部署，扛不住高并发	MySQL
磁盘使用情况	MySQL 单机磁盘容量几乎撑满	拆分为多
SQL 执行性能	单表数据量太大，SQL 越跑越慢	单表数据

中间件：sharding-jdbc：当当开源的，属于 client 层方案。确实之前用的还比较多一些，因为 SQL 语法支持也比较多，没有太多限制，而且目前推出到了 2.0 版本，支持分库分表、读写分离、分布式 id 生成、柔性事务（最大努力通知型事务、TCC 事务）。而且确实之前使用的公司会比较多一些（这个在官网有登记使用的公司，可以看到从 2017 年一直到现在，是有不少公司在用的），目前社区也还一直在开发和维护，还算是比较活跃，算是一个现在也可以选择的方案。
https://shardingsphere.apache.org/document/current/cn/overview/

熔断，限流，降级
超时：目的是保护消费方服务，超时时间的选取，一般看provider正常响应时间是多少，再追加一个buffer即可。
重试：对于provider这种偶尔抖动，可以设置合理的重试次数
幂等：在重试的前提下 需要保证服务提供方的幂等性
熔断：和重试相对的是熔断，重试是为了解决provider这种偶尔抖动，如果是provider持续的超时时间过长重试会使得消费方服务变得性能更差，需要采取熔断有损的提供服务
限流：provider有时候也要防范来自consumer的流量突变问题

qps限流：限制每秒处理请求数不超过阈值。
并发限流：限制同时处理的请求数目。Java 中的 Semaphore 是做并发限制的好工具，特别适用于资源有效的场景。
单机限流：Guava 中的 RateLimiter。
集群限流：TC 提供的 common-blocking 组件提供此功能。
算法：
漏桶算法：漏桶算法思路很简单，水（请求）先进入到漏桶里，漏桶以一定的速度出水，当水流入速度过大会直接溢出，可以看出漏桶算法能强行限制数据的传输速率。
令牌桶算法：对于很多应用场景来说，除了要求能够限制数据的平均传输速率外，还要求允许某种程度的突发传输。这时候漏桶算法可能就不合适了，令牌桶算法更为适合。
令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。
在 Guava 的 RateLimiter 中，使用的就是令牌桶算法，允许部分突发流量传输。在其源码里，可以看到能够突发传输的流量等于 maxBurstSeconds * qps。