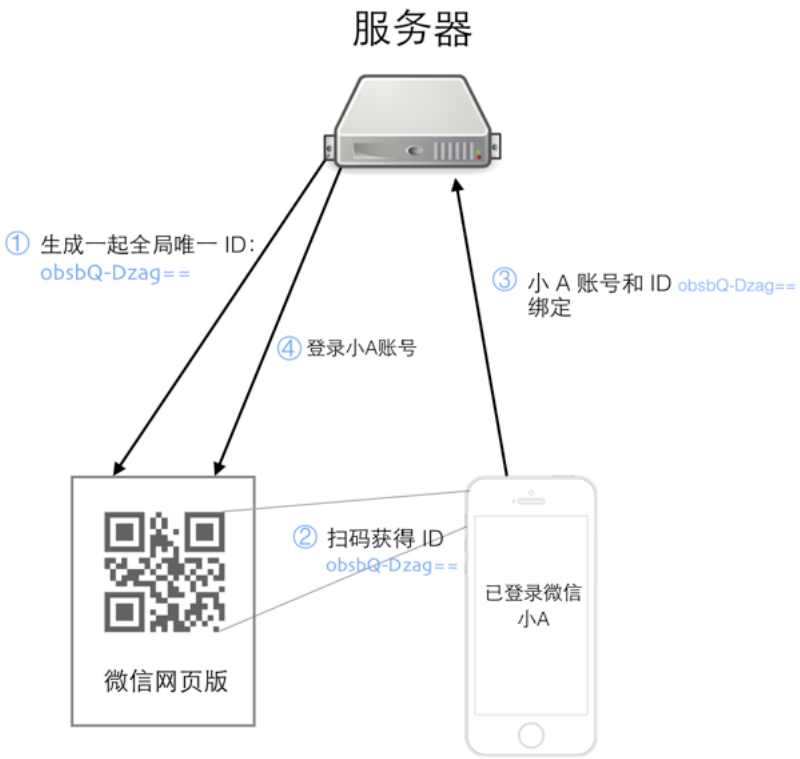
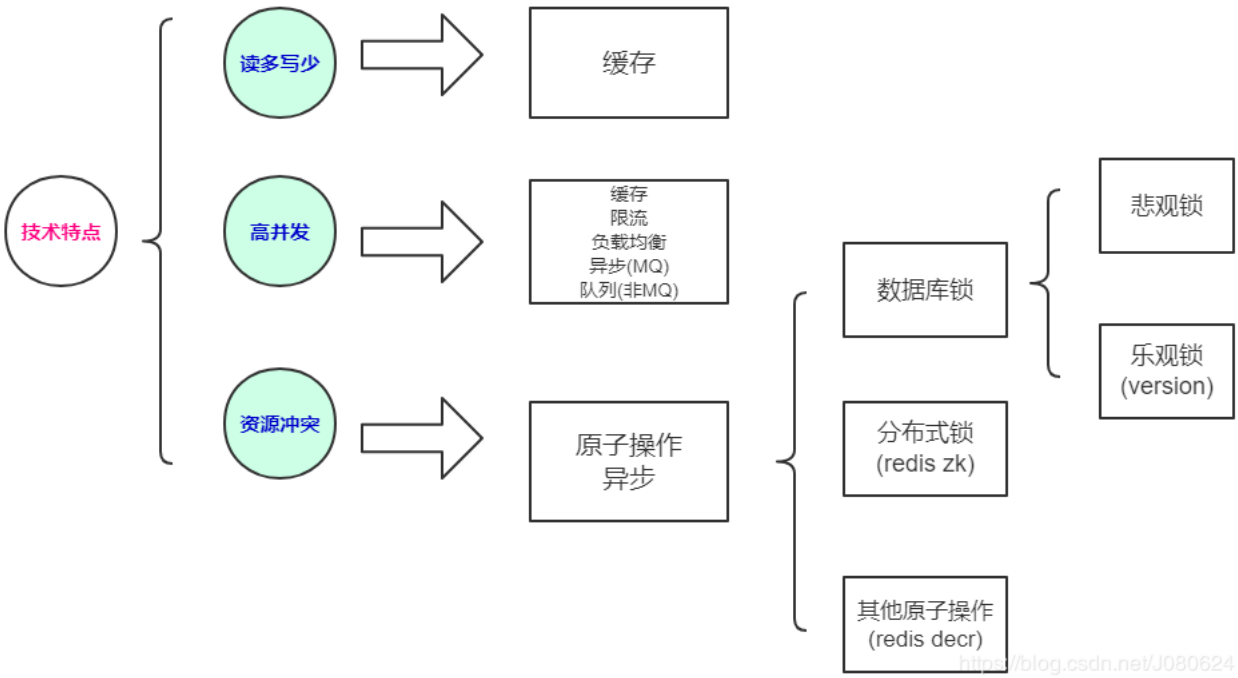
https://github.com/donnemartin/system-design-primer/blob/master/README-zh-Hans.md#%E5%A6%82%E4%BD%95%E5%A4%84%E7%90%86%E4%B8%80%E4%B8%AA%E7%B3%BB%E7%BB%9F%E8%AE%BE%E8%

1.扫码登录

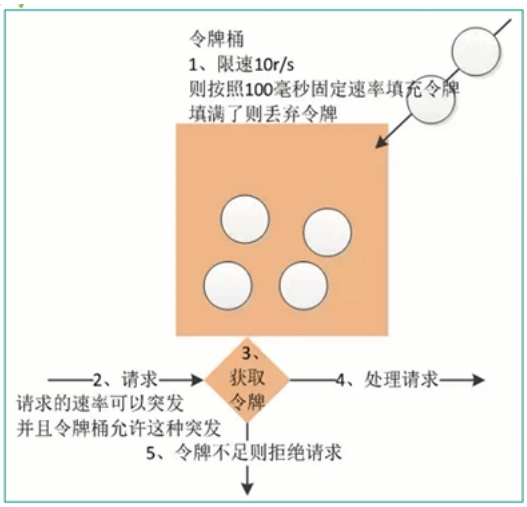


2.秒杀系统

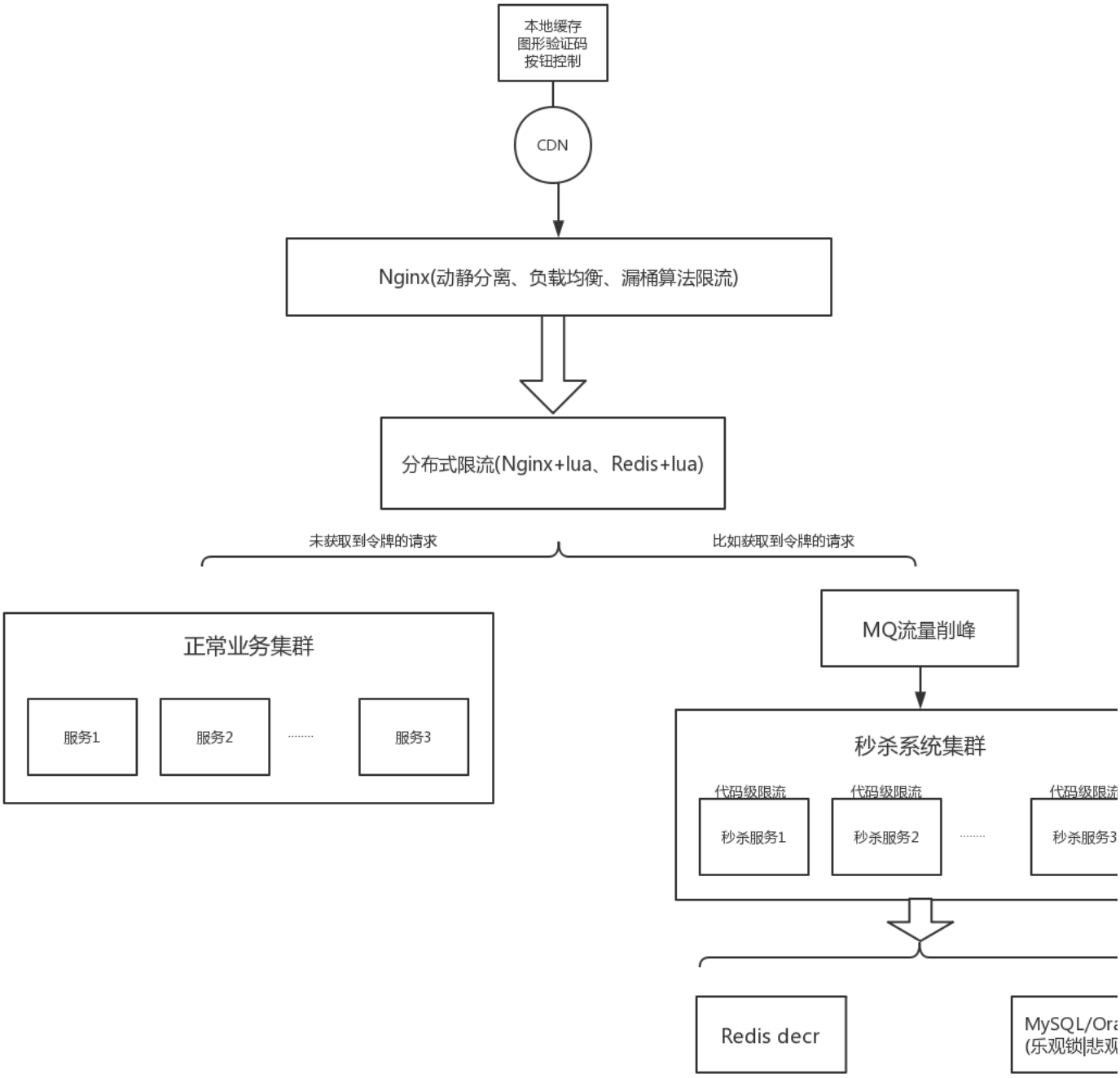
https://blog.csdn.net/J080624/article/details/86545536

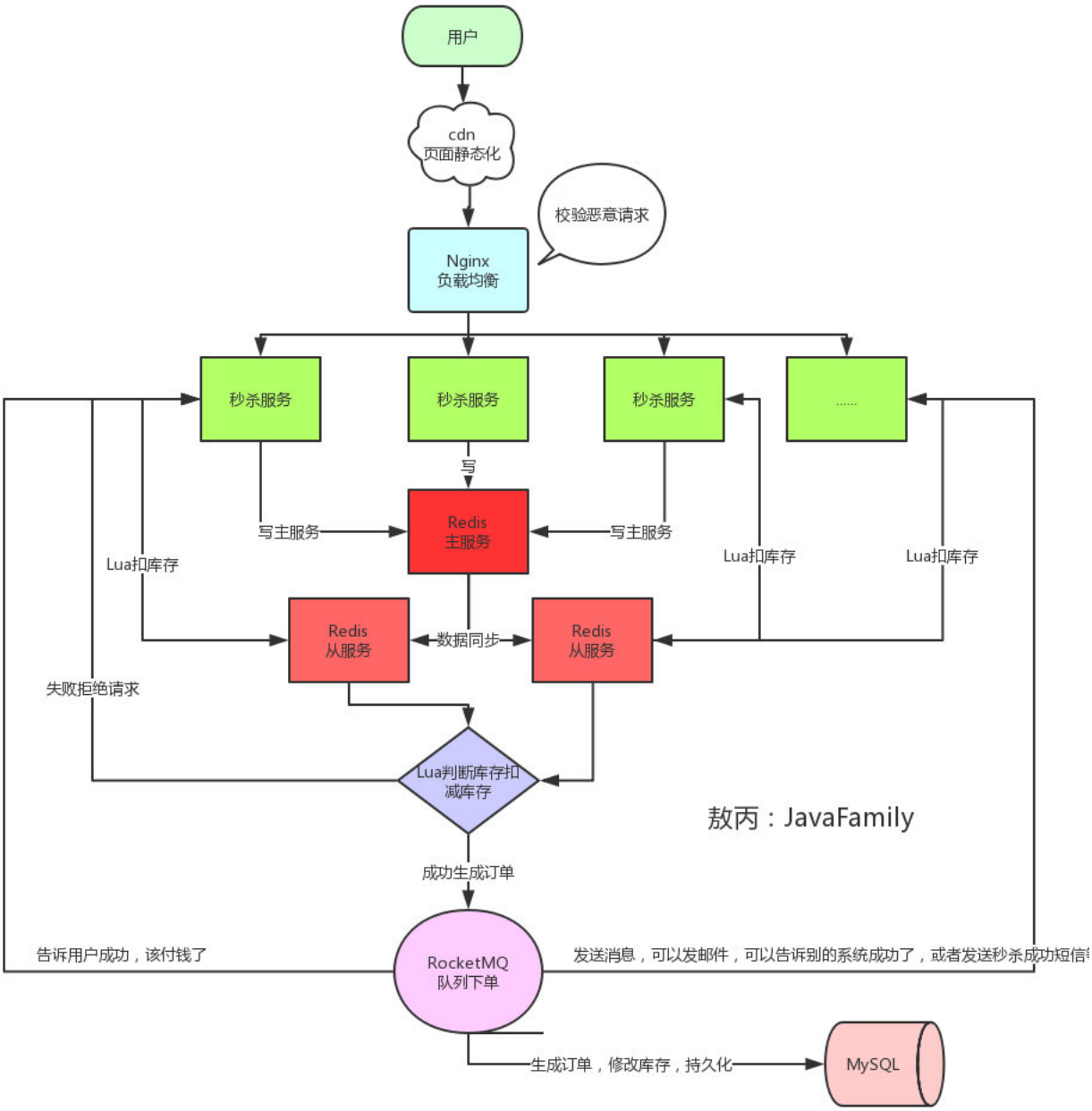


削峰利器-异步MQ、队列

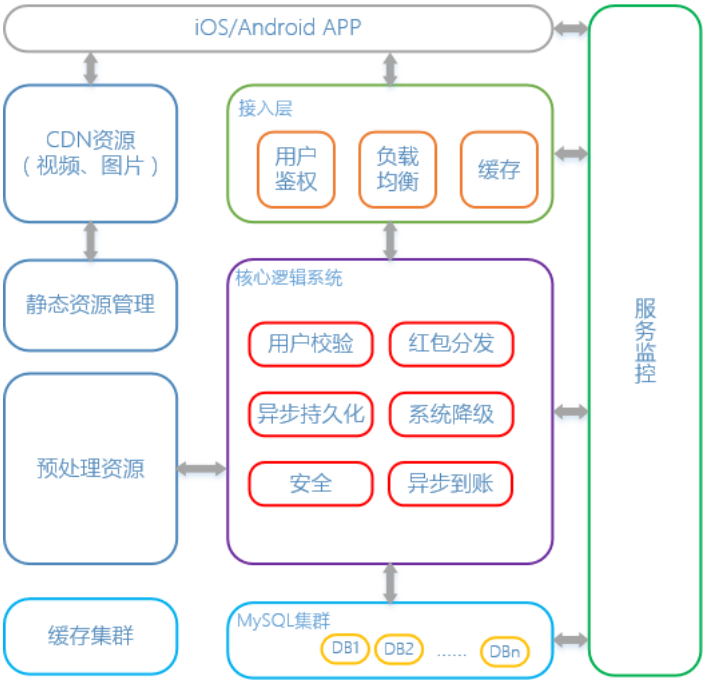

令牌桶
1、限速10r/s
则按照100毫秒固定速率填充令牌
填满了则丢弃令牌
2、请求
请求的速率可以突发
并且令牌桶允许这种突发
3、
获取
令牌
4、处理请求
5、令牌不足则拒绝请求


漏桶
1、流入水滴
流入速率任意
2、如果流入速率过
快，超过了桶的容量，
则直接丢弃水滴
3、按照常量速率
流出水滴
https://blog.csdn.net/J080624


本地缓存
图形验证码
按钮控制
CDN
Nginx(动静分离、负载均衡、漏桶算法限流)
分布式限流(Nginx+lua、Redis+lua)
未获取到令牌的请求
比如获取到令牌的请求
正常业务集群
服务1
服务2
服务3
MQ流量削峰
秒杀系统集群
代码级限流
秒杀服务1
代码级限流
秒杀服务2
代码级限流
秒杀服务3
Redis decr
MySQL/Ora
(乐观锁|悲观



https://www.ibm.com/developerworks/cn/web/wa-design-small-and-good-kill-system/index.html



抢红包的并发请求处理

春节整点时刻，同一个红包会被成千上万的人同时请求，如何控制并发请求，确保红包会且仅会被一个用户抢到?

做法一：使用加锁操作先占有锁资源，再占有红包。

可以使用分布式全局锁的方式（各种分布式锁组件或者数据库锁），先申请 lock 该红包资源且成功后再做后续操作。优点是不会出现脏数据问题，某一个时刻只有一个应用线程持有 lock，红包只会被至多一个用户抢到，数据一致性有保障。缺点是，所有请求同一时刻都在抢红包 A，下一个时刻又都在抢红包 B，并且只有一个抢成功，其他都失败，效率很低。

做法二：单独开发请求排队调度模块。

排队模块接收用户的抢红包请求，以 FIFO 模式保存下来，调度模块负责 FIFO 队列的动态调度，一旦有空闲资源，便从队列头部把用户的访问请求取出后交给真正提供服务的模块处理。优点是，具有中心节点的统一资源管理，对系统的可控性强，可深度定制。缺点是，所有请求流量都会有中心节点参与，效率必然会比分布式无中心系统低，并且，中心节点也很容易成为整个系统的性能瓶颈。

做法三：巧用 Redis 特性，使其成为分布式序号生成器（我们最终采用的做法）。

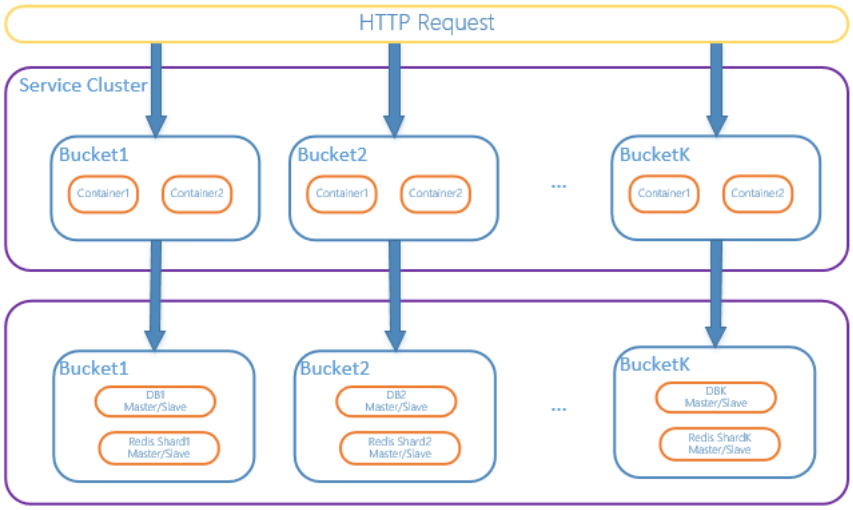
前文已经提到，红包系统所使用的红包数据都是预先生成好的，我们使用数字 ID 来标识，这个 ID 是全局唯一的，所有围绕红包的操作都使用这个 ID 作为数据的关联项。在实际的请求流量过来时，我们采用了"分组"处理流量的方式，如下图 3 所示。

访问请求被负载均衡器分发到每个 Service Cluster 的分组 Bucket，一个分组 Bucket 包含若干台应用容器、独立的数据库和 Redis 节点。Redis 节点内存储的是这个分组可以分发的红包 ID 号段，利用 Redis 特性实现红包分发，各服务节点通过 Redis 原语获取当前 拆到的红包。这种做法的思路是，Redis 本身是单进程工作模型，来自分布式系统各个节点的操作请求天然的被 Redis Server 做了一个同步队列，只要每个请求执行的足够快，这个队列就不会引起阻塞及请求超时。而本例中我们使用了 DECR 原语，性能上是可以满足需求的。Redis 在这里相当于是充当一个分布式序号发生器的功能，分发红包 ID。

此外，落地数据都持久化在独立的数据库中，相当于是做了水平分库。某个分组内处理的请求，只会访问分组内部的 Redis 和数据库，和其他分组隔离开。

整个处理流程核心的思想是，**分组的方式使得整个系统实现了高内聚，低耦合的原则，能将数据流量分而治之，提升了系统的可伸缩性，当面临更大流量的需求时，通过线性扩容的方法，即可应对。**并且当单个节点出现故障时，影响面能够控制在单个分组内部，系统也就具有了较好的隔离性。

图 3. 系统部署逻辑视图



系统容量评估，借助数据优化，过载保护

由于是首次开展活动，我们缺乏实际的运营数据，一切都是摸着石头过河。所以从项目伊始，我们便强调对系统各个层次的预估，既包括了活动参与人数、每个 APP 界面上的功能点潜在的高峰流量值、后端请求的峰值、缓存系统请求峰值和数据库读写请求峰值等，还包括了整个业务流程和服务基础设施中潜在的薄弱环节。后者的难度更大因为很难量化。此前我们连超大流量的全链路性能压测工具都较缺乏，所以还是有很多实践的困难的。

在这里内心真诚的感谢开源社区的力量，在我们制定完系统的性能指标参考值后，借助如 wrk 等优秀的开源工具，我们在有限的资源里实现了对整个系统的端到端全链路压测。实测中，我们的核心接口在单个容器上可以达到 20,000 以上的 QPS，整个服务集群在 110,000 以上的 QPS 压力下依然能稳定工作。

正是一次次的全链路压测参考指标，帮助我们了解了性能的基准，并以此做了代码设计层面、容器层面、JVM 层面、MySQL 数据库层面、缓存集群层面的种种优化，极大的提升了系统的可用性。具体做法限于篇幅不在此赘述，有兴趣的读者欢迎交流。

此外，为了确保线上有超预估流量时系统稳定，我们做了过载保护。超过性能上限阈值的流量，系统会快速返回特定的页面结果，将此部分流量清理掉，保障已经接受的有效流量可以正常处理。

1）尽量将请求拦截在系统上游 2）读多写少的常用多使用缓存

总体架构节点如下

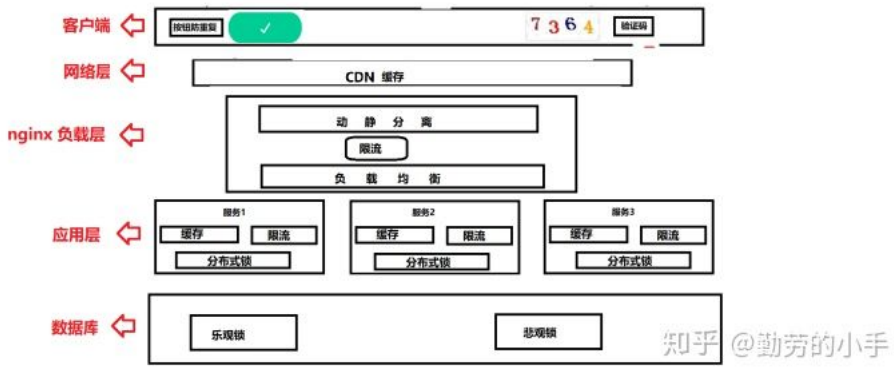
1、客户端： 按钮防重复点击、验证码防机器人

2、网络层：CDN缓存静态资源

3、负载层：Nginx： 动静分离、限流、负债均衡

4、应用层：缓存、限流、分布式锁

5、数据层：数据库乐观锁和数据库本身事务锁



3.应对高并发场景的优惠券发放

总体思路：请求+队列+异步发放卷码
1 用户请求，缓存查看是否有发放记录，有直接返回。
2 缓存读取判断发放记录>100W,直接返回码已经发完。
3 还有剩余码，请求用户信息放入队列，发放记录+1，缓存存已经发放用户标识，返回领取成功。
4 异步服务读取队列处理相应数据操作。

案例三：优惠券领取
需求：领取优惠券时，优惠券有限制“每天总领取张数”，“用户总共可领取该优惠券的张数”，必须很好的控制这个数量，保证优惠券发放准确。
分析：优惠券领取之前，肯定得判断优惠券是否已达到总领取张数，该用户是否已达到自身最大可领取张数。如用数据库记录的话，查出来数量的时候有可能其他线程已经进行了更新，容易发超。
解决方案：redis的自增incr是原子性操作，增加之后返回自增之后的值，那么可以将优惠券领取作为key，领取张数作为值，同理，将用户领取作为key，领取张数作为值，每次领取之前进行各自的自增操作，然后进行临界值判断，若超出，则提醒用户，并进行自减操作。
另，因为我们公司的优惠券券号是提前生成好的，所以也会以队列的形式存放在redis里。利用lpop等操作保证券码被唯一使用。

以上是自己在工作中使用到的控制并发采用的方法，总结了一下，主要是以下几点：
1.分布式服务，资源数据存放位置唯一，可在该方面寻求解决方法
数据库的唯一索引、强检验、id自增、乐观锁，redis的原子操作等
2.分布式服务，可通过方案设计将并发访问变成串行访问，很好的避免并发
以上内容是自己摸索总结，有问题希望看客指正，有好的方法的也欢迎留言分享！

4.大数据量业务的存储设计和选型

https://zhuanlan.zhihu.com/p/42776873
https://juejin.im/post/58ffd93a5c497d0058154162

	Mysql	HBase	ES
存储方式	行存储 适用于OLTP业务	列族存储 平衡了OLTP、OLAP业务	索引存储 适用于各种检索业务
扩展性	单机，扩展性较差	水平扩展	水平扩展
事务	支持	不支持	不支持
一致性	strong consistency	strong consistency timeline consistency	可配置
secondary index	支持	不支持	支持
全文检索	支持 但很鸡肋	不支持	支持

5.随机数抢红包设计，关注一致性和准确性

```
import java.text.DecimalFormat;
import java.util.Random;
import java.util.Scanner;


/**
* 模拟微信抢红包功能
*/
public class RedBags { //创建一个RedBags类
    public static void main(String[] args) {
        System.out.println("----------模拟微信抢红包----------\n");
        Scanner sc = new Scanner(System.in); //控制台输入
        System.out.print("请输入要装入红包的总金额（元）：");
        double total = sc.nextDouble(); //输入“红包的总金额”
        System.out.print("请输入红包的个数（个）：");
        int bagsnum = sc.nextInt(); //输入“红包的个数”
        double min = 0.01; //初始化“红包的最小金额”
        Random random = new Random(); //创建随机数对象random
        DecimalFormat df = new DecimalFormat("###,###,###.##"); //创建DecimalFormat类的对象df，并设置格式
        for (int i = 1; i < bagsnum; i++) { //设置“循环”，边界值是红包数-1
        //保证后面至少有(bagsnum - i) 个 最小金额可以发
            double safe = (total - (bagsnum - i) * min) / (bagsnum - i); //通过公式模拟数学中的离散模型
            double money = (double) random.nextInt((int) ((safe - min) * 100)) / 100 + min; //根据离散模型得到每个红包的金额
            total = total - money; //替换total的值
            String temp = df.format(money); //调用format()方法，对数字money格式化
            System.out.println("第" + i + "个红包：" + temp + "元"); //输出结果
```

```
        }
        String left = df.format(total); //调用format()方法，对数字total格式化
        //输出最后一个红包
        System.out.println("第" + bagsnum + "个红包：" + left + "元"); //输出结果
        sc.close(); //关闭控制台输入
    }
}
```

6.短连接服务设计

对每一个核心组件进行详细深入的分析。举例来说，如果你被问到设计一个 url 缩写服务，开始讨论：
* 生成并储存一个完整 url 的 hash
    * MD5 和 Base62
    * Hash 碰撞
    * SQL 还是 NoSQL
    * 数据库模型
* 将一个 hashed url 翻译成完整的 url
    * 数据库查找
* API 和面向对象设计

场景
短链接服务就是将一段长的URL转换为短的URL，比如利用新浪微博的短链接生成器，可将一段长的URL（http://blog.csdn.net/poem_qianmo/article/details/52344732）转换为一段短的URL（http://t.cn/RtFFvic），用户通过访问短链接即可重定向到原始的URL。
整个交互流程如下：
1. 用户访问短链接：http://t.cn/RtFFvic
2. 短链接服务器t.cn收到请求，根据URL路径RtFFvic获取到原始的长链接：http://blog.csdn.net/poem_qianmo/article/details/52344732
3. 服务器返回302状态码，将响应头中的Location设置为：http://blog.csdn.net/poem_qianmo/article/details/52344732
4. 浏览器重新向http://blog.csdn.net/poem_qianmo/article/details/52344732发送请求
5. 返回响应
设计要点
* 短链接生成算法
(1)利用放号器，初始值为0，对于每一个短链接生成请求，都递增放号器的值，再将此值转换为62进制（a-zA-Z0-9），比如第一次请求时放号器的值为0，对应62进制为a，第二次请求时放号器的值为1，对应62进制为b，第10001次请求时放号器的值为10000，对应62进制为sBc。
(2)将短链接服务器域名与放号器的62进制值进行字符串连接，即为短链接的URL，比如：t.cn/sBc。
* 重定向过程
生成短链接之后，需要存储短链接到长链接的映射关系，即sBc -> URL，浏览器访问短链接服务器时，根据URL Path取到原始的链接，然后进行302重定向。映射关系可使用K-V存储，比如Redis或Memcache。
优化方案
* 算法优化
采用以上算法，对于同一个原始URL，每次生成的短链接是不同的，这样就会浪费存储空间，因为需要存储多个短链接到同一个URL的映射，如果能将相同的URL映射成同一个短链接，这样就可以节省存储空间了。
（1）方案1：查表
每次生成短链接时，先在映射表中查找是否已有原始URL的映射关系，如果有，则直接返回结果。很明显，这种方式效率很低。
（2）方案2：使用LRU本地缓存，空间换时间
使用固定大小的LRU缓存，存储最近N次的映射结果，这样，如果某一个链接生成的非常频繁，则可以在LRU缓存中找到结果直接返回，这是存储空间和性能方面的折中。
* 可伸缩和高可用
如果将短链接生成服务单机部署，缺点一是性能不足，不足以承受海量的并发访问，二是成为系统单点，如果这台机器宕机则整套服务不可 用，为了解决这个问题，可以将系统集群化，进行“分片”。
在以上描述的系统架构中，如果发号器用Redis实现，则Redis是系统的瓶颈与单点，因此，利用数据库分片的设计思想，可部署多个发号器实例，每个实例负责特定号段的发号，比如部署10台Redis，每台分别负责号段尾号为0-9的发号，注意此时发号器的步长则应该设置为10（实例个数）。
另外，也可将长链接与短链接映射关系的存储进行分片，由于没有一个中心化的存储位置，因此需要开发额外的服务，用于查找短链接对应的原始链接的存储节点，这样才能去正确的节点上找到映射关系。