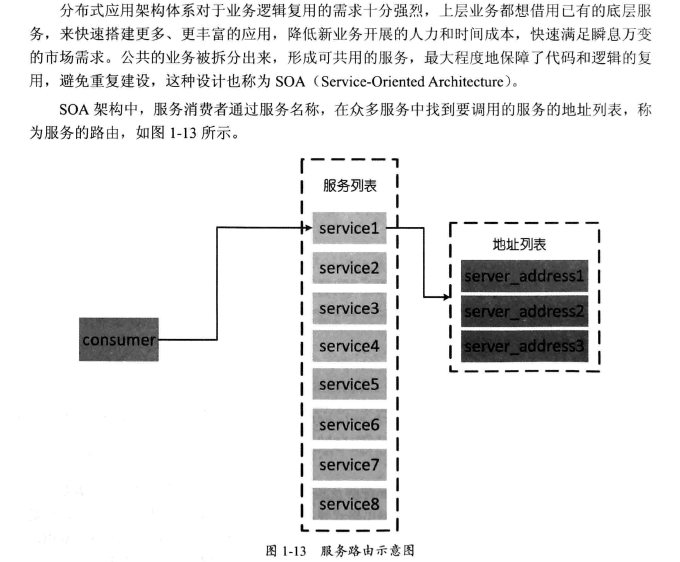
redis实现分布式锁

1.面向服务的体系架构SOA

RPC即Remote Process Call即远程过程调用，实现方式有RMI,WebService。

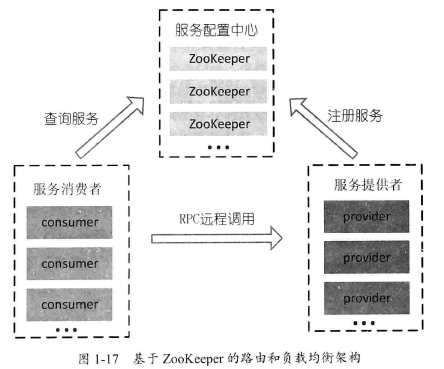
单台服务器的处理能力受硬件成本的限制，不可能无限制地提升。RPC将原来的本地调用转变为调用远端的服务器上的方法，给系统的处理能力和吞吐量带来了近似于无限制提升的可能，这是系统发展到一定阶段必须性的变革，也是实现分布式计算的基础。

将对象转换为二进制流的过程称为对象的序列化。将二进制流恢复为对象的过程称为对象的反序列化。序列化工具: Protocal Buffers,Hessian,java内置的（效率低）。



单点故障的解决方案就是机器集群，哪怕是负载均衡机器也要多台，以保证一台挂了另一台还能用。

基于ZooKeeper的持久和非持久节点，我们能够近乎实时地感知后到后端服务器的状态（上线，下线，宕机）。通过集群间zab协议，使得服务配置信息能够保持一致。而ZooKeeper本身容错特性和leader选举机制，能保障我们方便地进行扩容。通过ZooKeeper来实现服务动态注册，机器上线与下线的动态感知，扩容方便，容错性好，且无中心化结构能够解决之前使用负载均衡设备所带来的单点故障问题，只有当配置信息更新时才会去ZooKeeper上获取最新的服务地址列表，其他时候使用本地缓存即可。基于ZooKeeper的服务配置中心的搭建。



负载均衡算法包括轮询法，随机法，源地址哈希法，加权轮询法，加权随机法，最小连接法等，根据具体使用场景选取对应的算法。

1.轮询法

将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

2.随机法

通过系统随机函数，根据后端服务器列表的大小值来随机选取其中一台进行访问。

3.源地址哈希法

源地址哈希的思想是获取客户端访问的IP地址值，通过哈希函数计算得到一个数值，用该数值对服务器列表的大小时行取模运算，得到的结果便是要访问的服务器的序号。

4.加权轮询

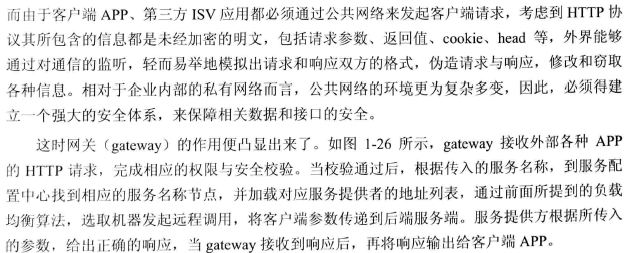
不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不尽相同。给配置高，负载低的机器配置更高的权重，让其处理更多的请求，而低配置，负载高的机器，则给其分配较低的权重，降低其系统负载，加权轮询能很好的处理这一问题，并将请求顺序且按照权重分配到后端。

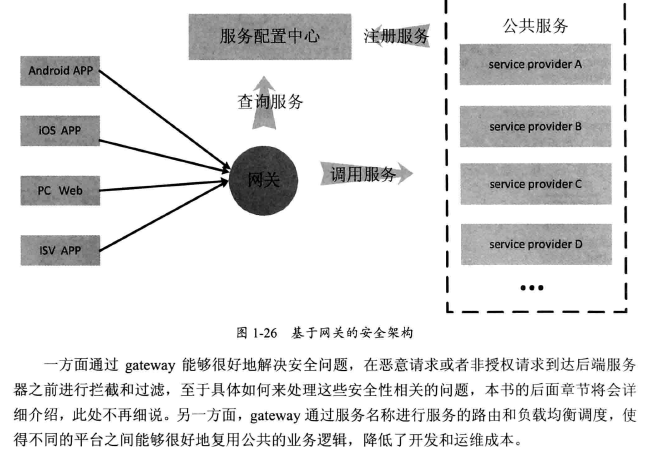
5.加权随机

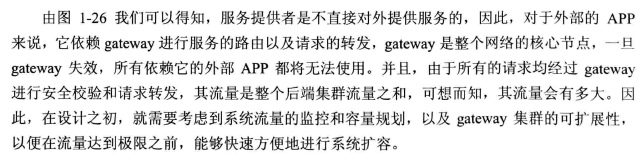
与加权轮询法类似，加权随机法也根据后端服务器不同的配置和负载情况，配置不同的权重。

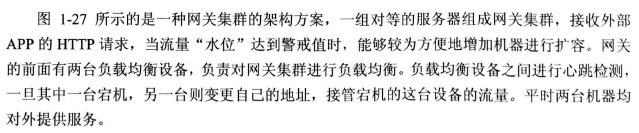
6.最小连接数

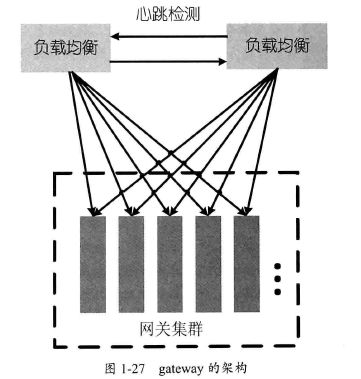
最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它正是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前请求，尽可能地提高后端服务器的利用效率，将负载合理地分流到每一台机器。

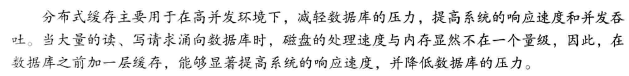


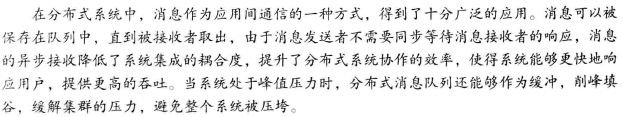


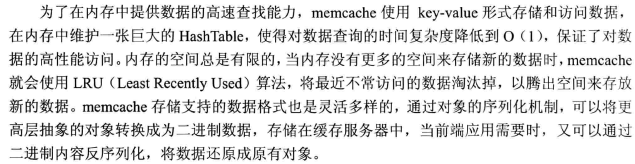






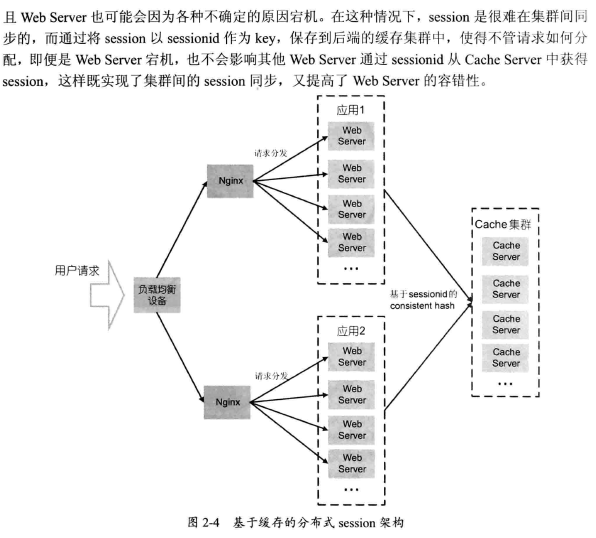






分布式session

对于系统可靠性要求较高的用户，可以将session持久化到DB中，这样可以保证宕机时会话不易丢失，但缺点就是系统的整体吞吐将受到很大的影响。可以将session统一存储集群上，如memcache,这样可以保证较高的读，写性能，这一点对于并发量大的系统来说非常重要，并且从安全性考虑，session毕竟是有有效期的，使用缓存存储，也便于利用缓存的失效机制。缺点，一旦缓存重启，里面保存的会话也就丢失了，需要用户重新建立会话。

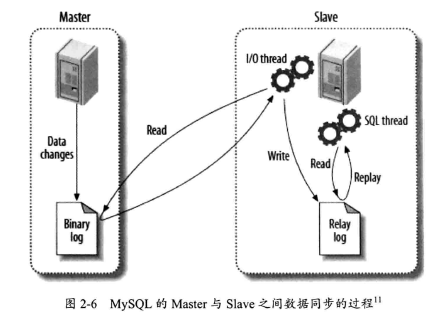


memcached-session-manager是一个开源的高可用的Tomcat session解决方案，它支持Stickey模式和Non-Sticky模式。Sticky模式表示每次请求都会被映射到同一台后端Web Server,直到该Web Server宕机，这样session可先存放在服务器本地，等到请求处理完成再同步到后端memcache服务器；而当Web Server宕机时，请求被映射到其他Web Server，这时候，其他Web Server可以从后端memcache中恢复session.对于Non-Sticky模式来说，请求每次映射的后端Web Server是不确定的，当请求到来时，从memcache中加载session;当请求处理完成时，将session再写回memcache.

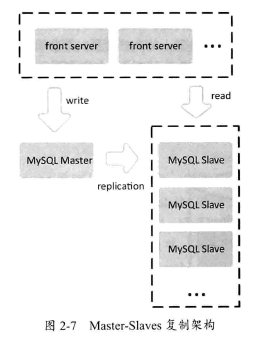
Non-Sticky需要配置Tomcat中的conf/context.xml文件中配置SessionManager.

数据库访问压力太大，可以按业务进行数据库拆分，如以前在一个DB里有news,user,post,comment表，现在按这些表来拆分数据库每一个表放到一个单一的数据库里去。

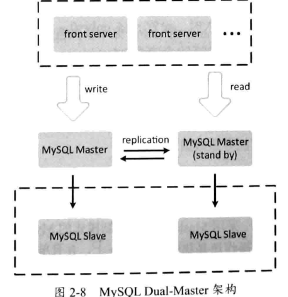
随着访问量的不断增加，拆分后的某个库压力越来越大，这时就需要集群了。使用mysql的replication(复制）策略来对系统进行扩展。主从。可以将一台mysql数据库服务器中的数据复制到其他mysql数据库服务器上。当各台数据库服务器上都包含相同数据时，前端 应用通过访问mysql集群中任意一台服务器，都能够读取相同的数据。要实现数据库的复制，需要开户Master服务器端的Binary log。数据复制的过程实际上就是Slave从master获取binary log,然后再在本地镜像的执行日志中记录的操作。由于复制过程是异步的，因此Master和Slave之间的数据有可能存在延迟的现象，此时只能够保证数据最终的一致性。

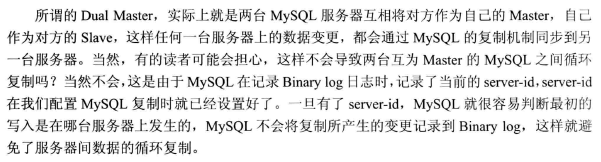


Master执行数据写入操作，数据的更新通过Binary log同步到Slave 集群，而对于数据读取的请求，则交由Slave来处理，这样Slave集群可以分担数据库读的压力，并且读，写分享还保障了数据能够达到最终一致性。

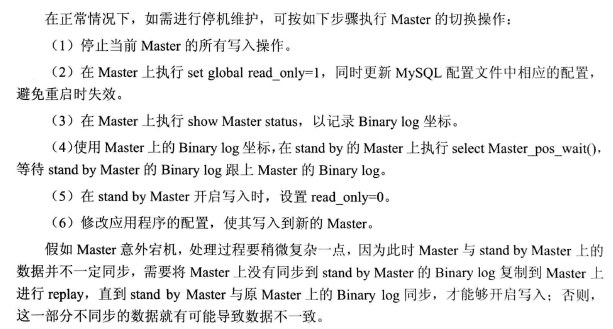


Master-Slaves复制架构存在一个问题，即所谓的单点故障。当Master宕机时，系统将无法写入，而在某些特定的场景下，也可能需要Master停机，以便进行系统维护，优化和升级。同样的道理，Master停机将导致整个系统都无法写入，直到Master恢复。为了尽可能地降低系统停止写入的时间，最佳的方式就是采用Dual-Master架构，双主架构，Master-Master架构。



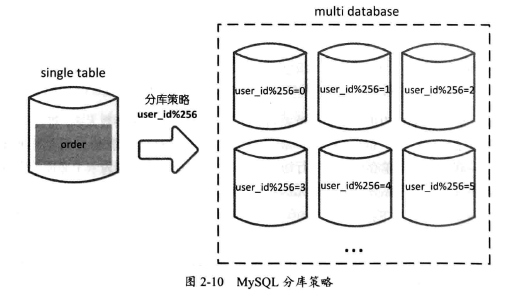


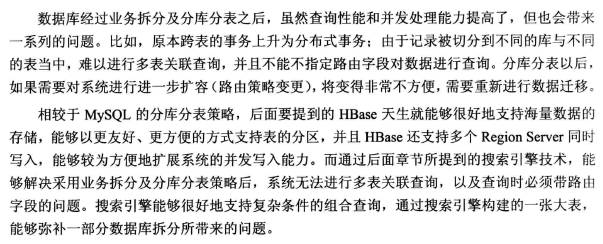
我们仅开启一台Master的写入，另一台Master仅仅stand by 或者作为读库开放，这样可以避免数据写入的冲突，防止数据不一致的情况发生。



对于访问极为频繁且数据量巨大的单表来说，我们首先要做的就是减少单表的记录条数，以便减少数据查询所需要的时间，提高数据库的吞吐，这就是所谓的分表。在分表之前，首先需要选择适当的分表策略，使得数据能够较为均衡地分布到多张表中，并且不影响正常的查询。

分表能够解决单表数据量达大带来的查询效率下降的问题，却无法给数据库的并发处理能力带来质的提升。面对高并发的读写访问，当数据库Master服务器无法承载写操作压力时，不管如何扩展Slave服务器，都没有用。此应该对数据库进行拆分，从而提高数据库写入能力，即所谓的分库。与分且策略相似，分库也可以采用通过一个关键字段取模的方式，来对数据访问进行路由。





垂直扩展一般是加内存，升cpu，硬件上面的升级。

Solo支持通过Data Schema来定义字段，类型和设置文本分析，使得用户可以通过http post请求，向服务器提交Document，生成索引，以及进行索引的更新和删除操作。对于复杂的查询条件，solr提供了一整套表达式查询语言，能够更方便地实现包括字段匹配，模糊查询，分组统计等功能；

对于常规的Web攻击手段，如xss,crsf,sql注入等，防范措施相对来说比较容易，如xss的防范需要转义掉输入的尖括号，防止crsf攻击需要将cookie设置为httponly，以及增加session相关的Hash token码，sql注入的防范需要将分号等字符转义。

考虑到数据在网络传输过程中的安全性，防止数中途篡改和被第三方监听拦截，这需要从系统架构的体系上来进行保障。如，采用摘要认证来防止 信息篡改，采用签名认证来验证通信双方的合法性，通过https协议来保障通信过程中数据不被第三方监听和截获。

OAuth协议的出现，使得用户在不需要泄露自己用户名密码的情况下，能够完成对第三方应用的授权，而第三方应用得到用户授权后，便可以在一定时间内访问到用户授权的数据。

常见的WEB攻击手段：

xss:跨站脚本攻击指的是攻击者在网页中嵌入恶意脚本程序，当用户打开该网页时，脚本程序便开始在客户端的浏览器上执行，以盗取客户端cookie,用户名和密码，下载执行病毒木马程序，甚至是获取客户端admin权限等。

csrf:csrf攻击的全称是跨站请求伪造(cross site request forgery),是一种对网站的恶意利用，尽管听起来跟xss跨站脚本攻击有点相似，但事实上csrf与xss差别很大，xss利用的是站点内的信任用户，而csrf则是通过伪装来自受信任用户的请求来利用受信任的网站。你可以这么理解csrf攻击：攻击者盗用了你的身份，以你的名义向第三方网站发送恶意请求。crsf能做的事情包括利用你的身份发邮件，发短信，进行交易转帐等，甚至盗取你的账号。



csrf防范：

1.将cookie设置为HttpOnly,这样脚本就无法读取到cookie信息，避免了攻击者伪造cooke的情况出现。

response.setHeader("Set-Cookie","cookiename=cookievalue;HttpOnly");

2.增加token,在请求中放入攻击者所不能伪造的信息，并且该信息不存在于cookie之中，鉴于此，系统开发人员可以在http请求中以参数的形式加入一个随便产生的token,并在服务端进行token校验，如果请求中没有token或者token内容不正确，则认为是csrf攻击而拒绝该请求。

3.通过Referer识别，根据Http协议，在Http头中有一个字段叫Referer，它记录了该http请求的来源地址。

SQL注入的防范：

1.使用预编译语句 PreparedStatement；2.使用ORM框架；3.避免密码明文存放；4.处理好相应的异常；

**常用安全算法：**

数字摘要也称消息摘要，它是一个唯一对应一个消息或文本的固定长度的值，它由一个单向Hash函数对消息进行计算而产生。如果消息在传递的途中改变了，接收者通过收到的消息采用相同的Hash重新计算，新产生的摘要与原摘要进行比较，就可知道消息是否被篡改了，因此消息摘要能够验证消息的完整性。消息摘要采用单向Hash函数，将需要计算的内容“摘要”成固定长度的串，这个串也称为数字指纹。

1.MD5:长度为128位

public static byte[] md5(String content) {

MessageDigest md = MessageDigest.getInstance("MD5"); //取到算法实例

byte[] bytes = md.digest(content.getBytes("utf8");

return bytes;

}

2.SHA:安全散列算法。总长度160.把上面的"MD5"换成"SHA-1"即可。

3.十六进制编码

4.Base64：可逆。

{

BASE64Encoder base64Encoder = new BASE64Encoder();

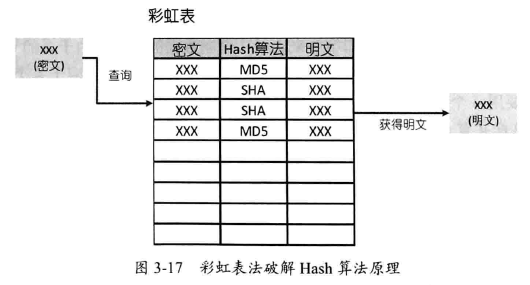
return base64Encoder.decodeBuffer("....");

BASE64Decoder base64Decoder = new BASE64Decoder();

return base64Decoder.decodeBuffer("....");

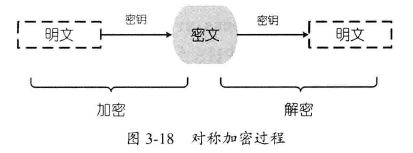
}

5.彩虹表破解Hash算法，彩虹表法是一种破解哈希算法的技术，从原理上来说能够对任何一种Hash算法进行攻击。它就是明文与密文的一种映射。也就是用的穷举法。字典大小都从几百m到几十GB不等。



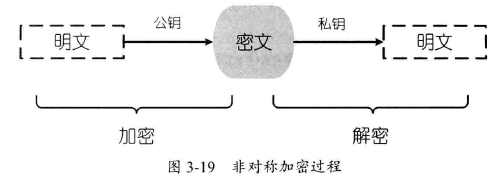
对称加密算法：

对称加密算法是应用较早的加密算法，技术成熟。在对称加密算法中，数据发送方将明文（原始数据）和加密密钥一起经过特种加密算法处理后，生成复杂的加密密文进行发送，数据接收方收到密文后，若想读取原文，则需要使用加密使用的密钥及相同算法的逆算法对加密的密文进行解密，才能使其恢复成可读明文。在对称加密算法中，使用的密钥只有一个，发送和接收双方都使用这个密钥对数据进行加密和解密，这就要求加密和解密方事先都必须知道加密的密钥。DES，AES

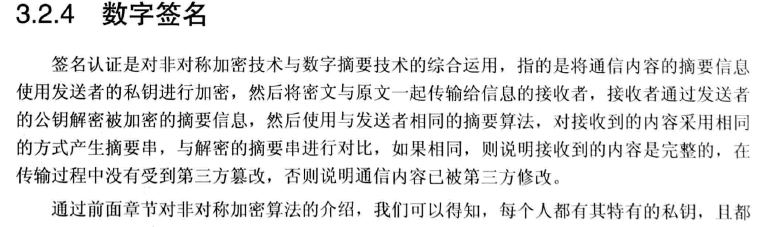


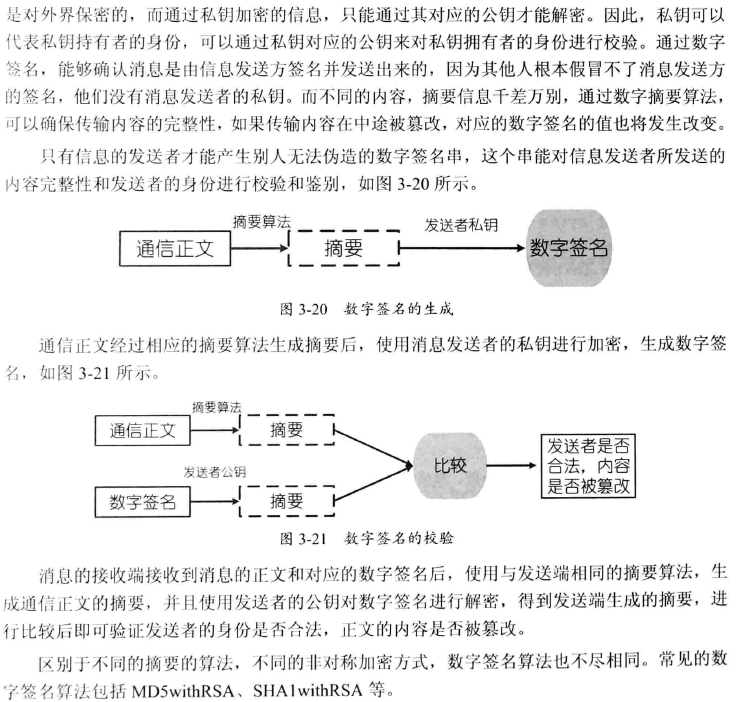
非对称加密算法

非对称加密算法又称为公开密钥加密算法，它需要两个密钥，一个公钥，一个私钥。公钥与私钥需要配对使用，如果用公钥对数据进行加密，只有用对应的私钥才能进行解密，而如果使用私钥对数据进行加密，那么只有用对应的公钥才能解密。加密和解密使用的是两个不同的密钥，所以称为非对称加密算法。RSA



<http://www.cnblogs.com/SirSmith/p/4985571.html>





JDK提供了一系列的故障跟踪和排查工具jstat,jmap,jstack,可查看内存空间使用情况，GC执行情况，线程堆栈，堆中对象等。

查看日志常用命令

查看行号: cat -n 文件

分页显示文件: more 文件

显示文件尾: tail -n2 文件 //显示文件最后2行

显示文件头: head -n2 文件 //显示文件前2行

字符统计: wc -l 文件 // -l统计行数，-c显示文件的字节数，-L查看最长的行的长度

查看重复出现的行: sort 文件 | uniq -c //排序后统计，-c显示每行内容出现的次数，-u只会显示出现一次的行，-d显示重复出现的行;

如果只知道文件名称，不知道路径或者需要查找一个文件的路径，可以使用find

#find /home/xxx -name "要查找的文件";

#find . -print 递归打印当前目录的所有文件

使用whereis能够方便地定位到文件系统中可执行文件的位置。

url访问 curl

以Tomcat下的access.log为例

查看请求访问量: cat access.log | cut -fl -d " " | sort | uniq -c | sort -k -n -r | head 10

页面访问量排名前10的url:cat access.log | cut -f4 -d " " | sort | uniq -c | sort -k -l -n -r | head -10

在Linux系统中，可以通过top和uptime命令来查看系统的load值，特定时间间隔内运行行列中的平均线程数。

uptime查看系统的load; 查看cpu利用率 :top | grep Cpu;

磁盘剩余: df -h ; 查看具体目录所占用的空间: du -d l -h /home/xxx;

系统网络状况: sar -n DEV 1

磁盘I/O: iostat -d -k; 内存使用: free -m

如果要查询我们的web应用是否正常，可以通过curl指令定时访问应用中预留的自检url,可以实时地感知到应用的健康状态，一旦系统无响应或者响应超时，即可输出报警信息，以被相应的监控调度系统捕捉到，第一时间通知开发和运维人员进行处理。

通过curl执行自检效果:

curl http://www.xxx.com/selfcheck/check.html

在xxx.com上在的服务器上有一个check.html，返回值为ok.正常情况下check.html返回ok,如服务器故障就不能正常返回了，就报错了。

可以通过shell实现应用监控的脚本：

#!/bin/bash

#xxx@gmail.com

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/lib

CURL=/usr/local/bin/curl

url\_for\_check="http://www.xxx.com/selfcheck/check.html"

result=`$CURL -s $url\_for\_check`

if [[ $result = \*ok\* ]];then

echo 'check success'

else

echo 'check error'

fi

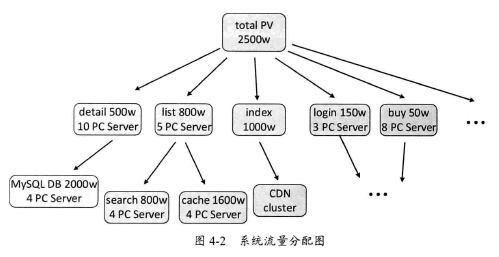
exit 0

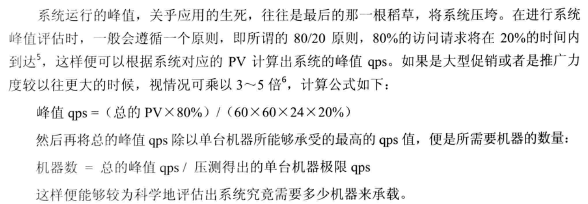
首先导出curl库的地址。

同样可以进行业务检测。

**容量评估和应用水位**

新系统上线之前，都要进行一些压力评估来确定一些合理的部署方案。业务方给出总的访问量，即总pv.uv以后，再逐一细化，推导出罗到每一个独立的系统，接口上的流量大概是多少，这样一来，每个子系统所承载的量也就清晰了。子系统将根据落到系统上的总的访问量，来评估机器的数量，网络的带宽和技术实现方式 。





**网站架构调整：**

1.独立部署，避免不同的系统之间相互争夺共享资源（比如cpu,内存，磁盘等)

2.web服务器集群，实现可伸缩性；

3.部署分布式缓存系统，使查询操作尽可能在缓存命中，降低数据库的负载压力。

4.数据库实施读/写分离，实现HA(High Availability,高可用性）架构；

5.利用CDN回事系统响应；可以将静态资源数据（图片，音频，视频，脚本，HTML网页)缓存在CDN节点上。CDN越来越便宜了。

6.业务垂直化，降低耦合，从而实现分而治之的管理。

拆系统之业务垂直化：

由于业务逻辑全部耦合在一起，并且部署在同一个web容器中，必然会导致系统中不同业务之间的耦合过于紧密，除了扩展和维护困难，在生产环境中极有可能会因为某个业务功能不可用而影响系统整体服务不可用，因此在这个阶段主要需要解决的问题就是降低业务耦合，实现高内聚低耦合，提升系统容错性，避免牵一发而动全身的风险。

注意：拆分粒度越细，耦合越小，容错性越好，每个业务子系统的职责就越清晰。但是如果拆分粒度过细，维护成本将是一个不小的挑战。

为什么需要实现服务化架构：

多元化的业务需求下，子系统中一定会存在较多的共享业务，这些共享业务肯定会被重复建设，产生较多的冗余业务代码。就是把大家都要用到的业务功能从他们的业务中抽离出来做成一个单一的共享的服务。

<http://blog.csdn.net/csdnxingyuntian/article/details/54948783>

**服务拆分粒度之微服务：**

一般来说建议大家以子系统为维度进行服务拆分，也可以在此基础上对服务的读/写进行分离，或者拆分出单独的服务，这样服务的拆分粒度大小适中，所付出的成本与实际收益会相对更加均衡，否则粒度过细不仅会导致维护成本提升，而且系统出现问题时定位和排查问题困难，更重要的是很难梳理服务之间的依赖关系。

**服务化与RPC协议:**

RPC由客户端和服务端两部分构成，和在同一个进程空间内执行本地方法调用相比，rpc的实现细节会相对复杂不少。简单来说，服务提供方所提供的方法需要由服务调用方以网络的形式进行远程调用，因为这个过程也称为rpc请求，服务提供方根据服务调用方提供的参数执行指定的服务方法，执行完后再将执行结果响应给服务调用方，这样一次rpc调用就完成了。rpc实现方案有很多，如rmi,webservice,hessian及finagle等。不同的rpc实现对序列化和反序列化的处理也不相同，比如将对象序列化成xml/json等文本格式尽管具备良好的可读性，扩展性和通用性，但却过于笨重，不仅报文体积大，解析过程也异常缓慢，因此在一些特别注重性能的场景下，采用二进制协议更合适。

PRC调用三个步骤:

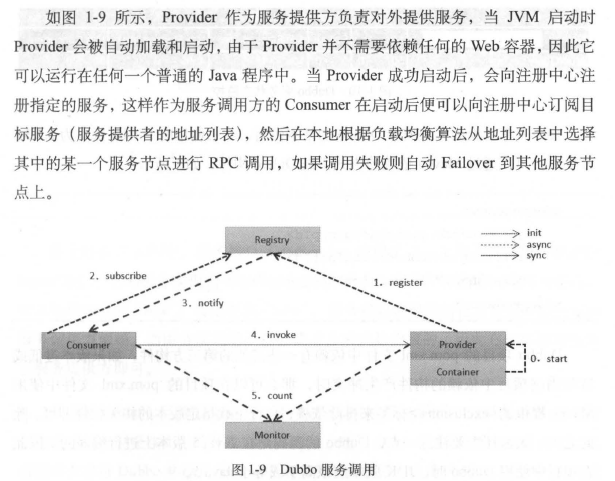
.底层的网络通信协议处理

.解决寻址问题

.请求/响应过程中参数的序列化和反序列化工作。

其实rpc的本质 就是屏蔽以上复杂的底层处理细节。

1.使用阿里分布式服务框架Dubbo实现服务化（设计精良，使用简单，技术文档丰富，还可二次开发）



不建议在生产环境中将超时时间设置得太长，否则如果应用获取不到会话，又长时间不肯返回，那么一定会对业务产生较大的影响。但将超时时间设置得太短又可能适得其反，因此超时时间究竟应该如何设置需要根据实际的业务场景而定，大家可以在日常的压测过程中仔细评估。比较简单的业务执行时间较短，可以将超时时间设置短一点，但对于复杂业务而言，则需要将超时时间适当地设置得长一点。

Failover（故障转移）

Dubbo不仅是一个RPC框架，更是一个服务治理框架：服务的动态注册与发现、服务的扩容评估、服务的升/降级处理。

为什么要引入注册中心：当服务变得越来越多时，如果把服务的调用地址(URL)配置在服务调用方，那么url的配置管理将变得非常麻烦，因此引入注册中心的目的就是实现服务的动态注册和发现，让服务的位置更加透明，这样服务调用方将得到解脱，并且在客户端实现负载均衡和Failover将会大大降低对硬件负载均衡器的依赖，从而减少成本。所以需要进行监控。

服务化后的分布式事务问题：

业界的技术难题，没有通用的解决方案，没有高效的实现手段。

分布式系统为什么需要进行流量管制

分布式系统应对高并发，大流量的常规手段：扩容(集群)、动静分离、缓存、服务降级(如暂时关闭评论)、限流。

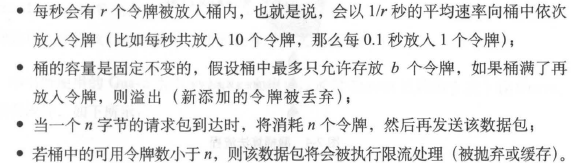
限流的具体方案：

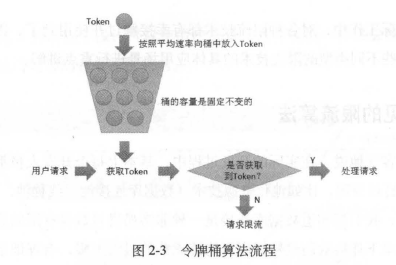
根据自己的业务来限流。

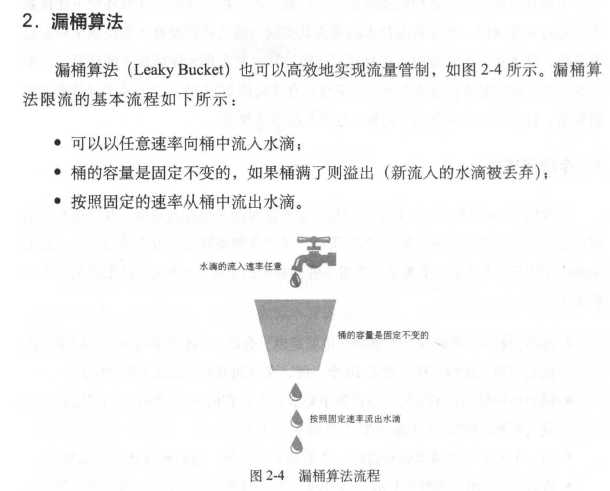
常见的限流算法：比如池化资源技术(数据库连接池，线程池，对象池等)，以数据库为例，我们都乱箭数据库连接是一种非常昂贵且数量有限的底层资源，为了避免并发环境中连接数超过数据库所能够承载的最大上限，合理地运用连接池技术，可以有效限制单个进程内能够申请到的最大连接数，确保在并发环境下连接数不会超过资源阈值。

1.令牌桶算法：

令牌桶算法主要用于限制流量的平均注入速率，并且还允许出现一定程序上的突发流量。Nginx的限流模块就是一个典型的采用令牌算法的实现。







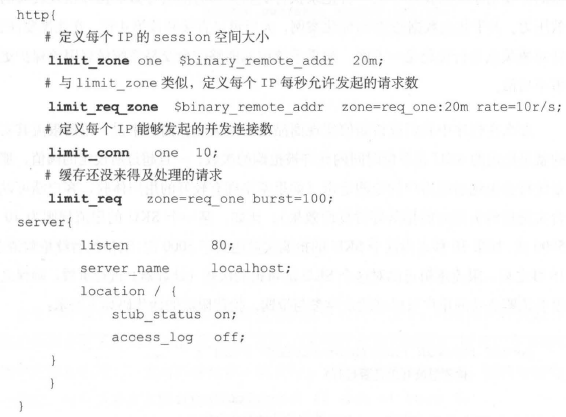
从本质上说，令牌桶算法和漏桶算法都可以用于在高并发，大流量场景下对流量实施管制，让系统的负载片于比较均衡的水位，不会因为峰值流量过大，导致系统被击垮。这两种算法的限流方向是截然相反的。令牌桶算法限制是是流量的平均流入速率，并且可以允许出现一定程序上的突发流量，当桶中令牌数量不足扣减时，新的请求将被执行限流处理；而漏桶算法限制的是流量的流出速率，而不是流入速率，并且这种流出速率还是保持固定不变的，不允许像令牌桶算法那样出现突发流量，当流入的水滴超过桶的容量时，新的请求将被执行限流处理。

使用Google的Guava实现平均速率限流。本地缓存领域除了可以使用EhCache，Guava Cache也是一个非常不错的替代方案。

**使用Nginx实现接入层限流**

Nginx是一个开源的高性能的Http服务器，同时也可以作为一个反向代理服务器，甚至还可以作为一个IMAP/POP3/SMTP服务器。由于Nginx拥有强悍的并发处理能力，因此许多互联网企业都将Nginx部署在接入层，将其作为反向代理服务器来使用，负责请求的负载均衡和分发等工作。

Nginx自带限流模块：



参数limit\_req\_zone用于设置每个IP在单位时间内所允许的请求数，值"rate=10r/s"表示每个IP每秒只允许发10个请求。如果每秒每个ip发起的请求数超过10个应该怎么办？参数limit\_req的作用类似于缓冲区，用于缓存还没有来得及进行处理的请求，值"burst=100"表示缓存的请求数为100，如果缓冲区也缓存不下时，新的请求将会被拒绝和抛弃。

使用计数器算法实现商品抢购限流：

除了令牌桶，漏桶等限流算法，计数器算法也可以达到不错的限流效果。抢购为例，当用户成功下单后便需要扣减指定的库存数量，但是大量的并发请求对数据库中同一行记录执行更新操作必然会导致数据库出现较大的负载压力。那么程序中如何实现 商品抢购限流呢？抢购限流其实指的就是指定的sku在单位时间内允许被抢购的次数，一旦超过所设定的阈值，那么系统将会拒绝后续用户的抢购请求（如果希望拥有较好的用户体验，客户端可以配合实施抢购失败时的排队等待页面效果）。比如，某一个sku的限流规则为10秒5000次，如果10秒之内这个sku的抢购次数达到了5000次，则拒绝后续抢购请求，10秒之后，限流逻辑可以对这个sku的可抢购次数（计数器）进行重置，确保之前没抢购成功的用户可以继续正常参与抢购。抢购的伪代码如下：

public boolean limit(List<String> skus) {

if (检测限流开关是否打开) {

if(null != skus && !skus.isEmpty()){

/\*key为sku,value为可抢购的剩余次数\*/

Map<String, Integer> skuMap = 获取被限流的sku名单;

synchronized(skuMap){

try {

for(String sku : skus){

/\*检查目标sku是否包含在限流名单中\*/

if (skuMap.containsKey(sku)) {

while\_skus.add(sku);

/\*如果抢购次数达到阈值，则不允许下单\*/

if (0 >= skuMap.get(sku)) {

return false;

}

}

}

for (String sku : while\_skus){

/\*扣减可抢购次数\*/

skuMap.put(sku, skuMap.get(sku)-1);

}

} finally {

if (!while\_skus.isEmpty()) {

while\_skus.clear()

}

}

}

}

}

return true;

}

在生产环境中，可将限流规则配置在配置中心内，以便后续对限流开关进行动态关停和变更不同sku的限流规则。可抢购次数的扣减操作，可以在redis等集中式容器中进行，也可以直接在本地进行。

基于时间分片的消峰方案

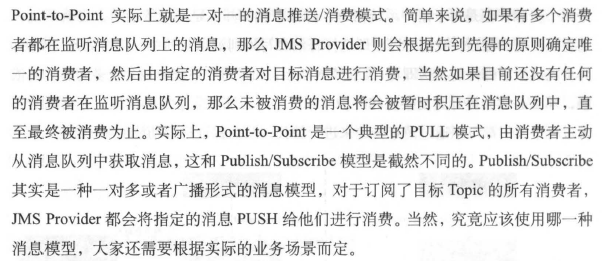
1.活动分时段进行实现消峰，一天分多次抢购

2.通过答题验证实现消峰。如12306的验证

3.异步调用需求，Fork/Join框架可以进行多核并行计算。

使用MQ是为了实现特定场景下的流量消峰，以及通过消息传递来实现异步调用等操作。Apache下的ActiveMQ,Kafka,阿里的RocketMQ以及HornetQ,RabbitMQ,ZeroMQ。建议规模较小的网站使用ActiveMQ这种轻量级的MQ产品（甚至也可以使用Redis提供的Publish/Subscribe模型).而KafKa,RocketMQ天生就是为互联网场景下拥有高并发，大流量的分布式系统量身打造的。因为在这种规模的消息体量上，我们重点需要考虑的是MQ产品的吞吐量，可用性及扩展性等多个方面的问题。

JMS主要由JMS Provider,Provider及Consumer三个角色构成，其中JMS Provider的主要任务是负责消息路由和消息传递，而Provider(消息生产者）和Consumer(消息消费者),其实属于JMS客户端，只是前者负责消息队列写入消息，而后者负责订阅消息。有点对点 和 发布/订阅模型。



RocketMQ特点:

支持顺序消息

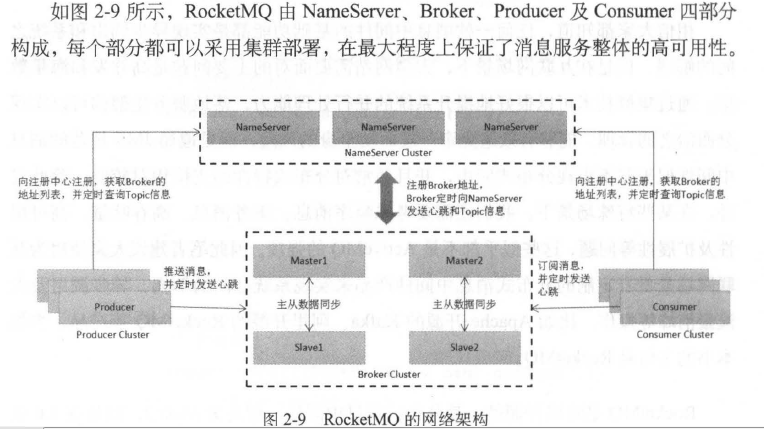
支持事务消息

支持集群与广播模式

亿级消息堆积能力

完善的分布式特性

支持push与pull两种消息订阅模式



NameServer其实就是一个注册中心，Broker,Producer及Consumer在启动时都会向其进行注册，它的主要功能是负责客户端的寻址操作。NameServer集群中的各个节点之间都是无状态的，由于NameServer并不会存在特别频繁的读/写操作，几乎没有负载压力，因此在生产环境中部署NameServer集群时并不需要添加太多的节点。

Broker为消息服务端，提供消息的管理，存储及分发等功能。一个Topic会被均匀分布到集群环境中的所有Broker节点上，在RocketMQ中实际负责消息存储的是Queue，它包含在Topic内部。换句话说，Queue是消息存储的最小逻辑单元。如果生产环境中某个Topic的消息量特别大，那么应该为Broker集群添加更多的节点来降低单个Broker节点的负载压力。当每个Broker节点启动时会和NameServer集群中的所有节点建立长连接，并定时轮询发送心跳和topic信息，由NameServer负责定时检查当前的存活连接，如果在指定的时间内没有接收到心跳，那么这个Broker节点的会话连接将会被NameServer主动关闭。

Producer为消息生产者，用于向Broker推送消息；而Consumer为消息消费者，负责消费Broker中的消息。单个Producer和Consumer在启动时会在集群环境中的某一个NameServer节点建立长连接，定时轮询Topic信息，如果当前所连接的NameServer节点宕机，则自动重连到其他集群节点上。除此之外，Producer和Consumer还会定时向Broker发送心跳，由Broker负责定时检查当前的存活连接，如果在指定时间内没有接收到心跳，那么客户端的会话连接将被Broker主动关闭。

Broker的部署：

单Master模式；多Master集群模式；多Master/Slave异步复制模式；多Master/Slave同步双写模式；

同步双写类似于Mysql数据库的半同步复制功能，同一份数据只有在主从都成功写入后，才会返回给Producer,这样即使Master宕机，Comsumer仍然可以从Slave上订阅还未被消费的消息，并且主从之间的消息是无延迟的，但性能相对较低。异步对性能的影响小，但会导致主从之间出现一定数据不一致的窗口期，但通常消息的延迟都是毫秒级别。

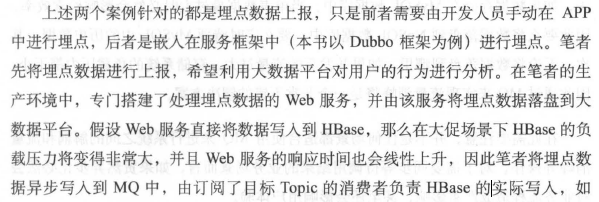
MQ的主要目的：

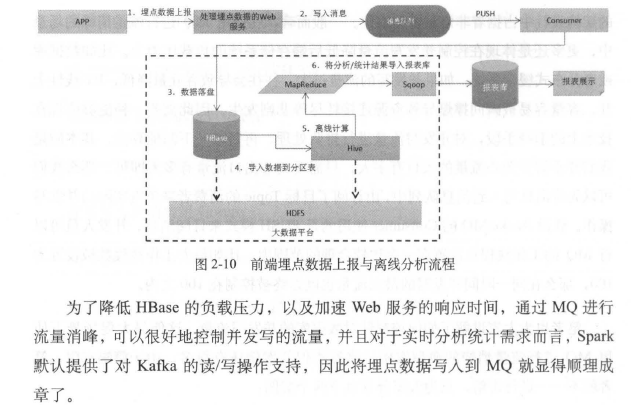
1.通过异步调用可以很好地解决分布式环境下系统之间的耦合问题，减少系统中一些非必需的依赖调用，降低复杂度和维护成本，从而保证进程 功能的单一性；在互联网场景下，大型网站主要需要面对的问题是高并发和海量数据，那么为了避免流量过大对系统产生较大冲击，引发系统出现雪崩现象，利用MQ来实现流量消峰可以让流量可控，使之有条不絮地对系统进行访问操作，最大程序上保护系统的稳定运行。

案例：

1.前端埋点数据上报消峰案例;

2.分布式调用跟踪系统的埋点数据上报消峰案例。





如果数据存储在Mysql中，当单表数据量过大时必然会影响检索效率，因此要么将数据落盘到NoSql数据库中，要么定时清理mysql中的历史数据；其次，无论将数据落盘到哪里，如果并发写的流量过大，存储系统的负载压力都不小，因此通过mq来实现流量消峰是一个非常不错的解决方案。

Zookeeper实现一个分布式配置管理平台；

在分布式环境中，由于对业务进行垂直划分，并且系统往往都是采用集群部署的，那么集群环境中的每一个节点都持有同一份配置文件，一旦配置信息发生改变，就意味着集群环境中的所有配置文件都需要做出相应的调整，而随着系统拆分的粒度越来越细，维护成本将会大大提升，并且配置出错的可能性也随之增加，因此需要一种集中式资源配置的形式，以让所有的集群节点共享同一份配置信息。降此之外，我们在某些场景下希望配置是可以在运行时发生变更的，系统必须在不重启的情况下获取到这些变更后的数据信息，并及时作出相应的调整策略。

集中式资源管理平台好处：

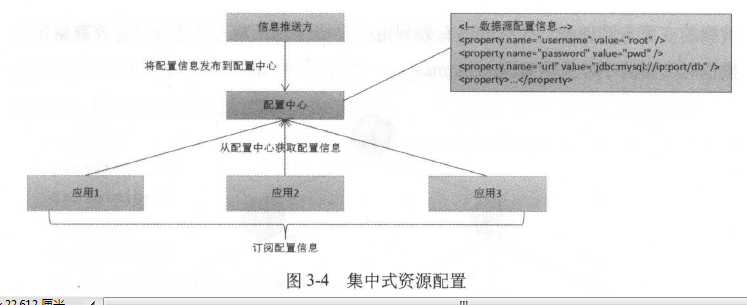
配置信息统一管理；

动态获取/更新配置信息；

降低运维人员的维护成本；

降低配置出错率；

分布式配置管理服务的本持就是典型的发布/订阅模式，获取配置信息的一方为订阅方，发布配置信息的一方为推送方；

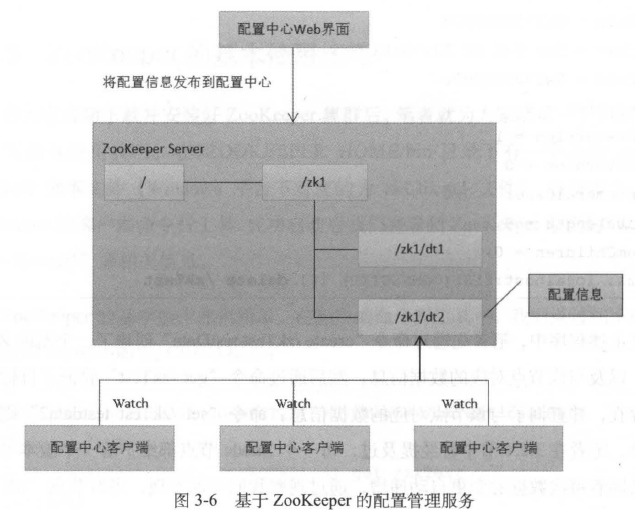


Zookeeper提供配置管理（数据发布/订阅)，分布式协调/通知，分布工锁及统一命名等服务，利用Zookeeper提供的一系列接口，能够非常方便地实现一致性，Leader选举，以及分布式配置管理平台等功能。

Zookeeper的数据模型中，每个子节点叫做Znode,每个Znode都有一个全局唯一的路径作为标示，进行数据的读/写操作。

Znode的不同类型（持久节点和瞬时节点）决定了其不同的行为特性。如果需要实现Leader选举，分布式锁等场景，那么瞬时节点无疑是最好的选择。每一个Znode节点都维护着一个版本号，版本号会随着每次数据的变更自动递增，如果能够有效利用这个特性，开发人员便可以有效避免并发操作时带来的不一致性问题。

我们将信息发布到Znode目录上后，由客户端负责信息订阅，一旦配置信息发生变更，所有的Watch目标Znode的ZooKeeper客户端都会感知到，那么当重新获取配置信息后，我们便可以在程序中执行一些自定义的逻辑处理（比如开关控制，动态更新系统的配置信息等操作）；



客户端需要考虑：

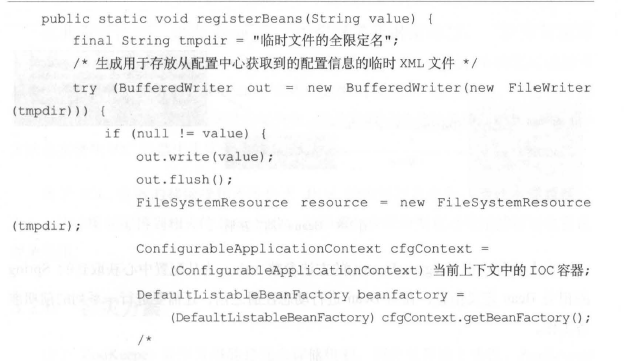
信息配置管理界面；

订阅Znode中的配置信息；

容灾机制；

**从配置中心获取Spring的Bean定义实现Bean的动态注册**

因为spring中ioc中的Bean是单例的，所以如果配置发生变化的话，我们需要重新注册Bean.





1.输出Spring相关Bean的定义信息到本地临时文件，通过FileSystemResource读取。将ApplicationContext的实例强制向下转换为ConfigurableApplicationContext，这样可以通过调用getBeanFactory()方法来获取DefaultListableBeanFactory实例。可以通过ApplicationContextAware接口，重写该接口的setApplicationContext()就去后，可以在IOC初始化时得到ApplicationContext实例。实例化XmlBeanDefinitionReader后，可以通过调用loadBeanDefinitions()方法从之前输出到本地的xml文件中加载相关的Bean定义，并由该方法调用Spring的其他方法完成如下操作：

解析Bean定义；

销毁当前IOC容器中与Bean定义中相同beanName的Bean实例；

Bean动态注册；

如果在Bean定义中配置了数据源信息，一定要将属性"destroy-method"设置为"close"，否则重新注册Bean后，之前持有的底层连接资源将无法得到释放。

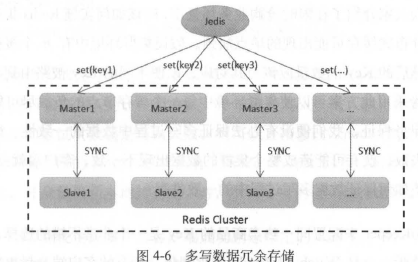
Hibernate中的二级缓存默认使用的就是Ehcache.

LRU最近最少使用，FIFO先进先出;LFU较少使用

Ehcache是本地缓存肯定会和jvm共享内存，可能会导致java.lang.OutOfMemoryError.

Redis集群多写多读方案

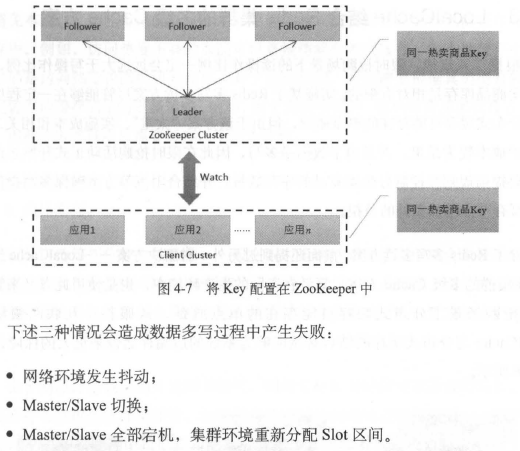
假设某一个热卖商品的Key为"itemid123",经过路由后，数据会被存储到集群环境中的某一个缓存节点上。我们把多写和多读这两个步骤拆分开来看，多写就是指在限时抢购活动正式开始之前，先对某一个热卖商品的key进行加工和计算。假设集群环境中有n个缓存节点，那么加工和计算后的key也应有n个，在最理想的情况下，每一个key都应该路由到一个指定的缓存节点上，其实多写操作就是为了实现数据的冗余存储。



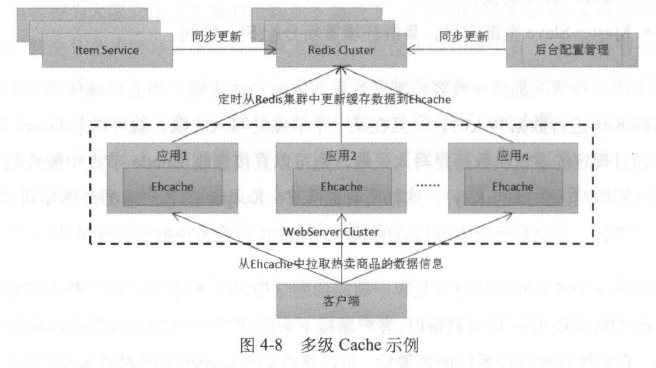
多写过程中要保证数据的一致性；

假设集群环境中有n个缓存节点，那么加工和计算后的key的数量应该与其对应，以便不同的key被路由到不同的缓存节点上。尽管采用此方案可以使集群环境中每一个缓存节点的负载尽可能均衡，但是由于缺少事务保证，没办法保证多写过程中数据的一致性；因此一旦某个节点写入失败，就有可能造成整个集群的数据出现不一致，客户端会出现脏读。

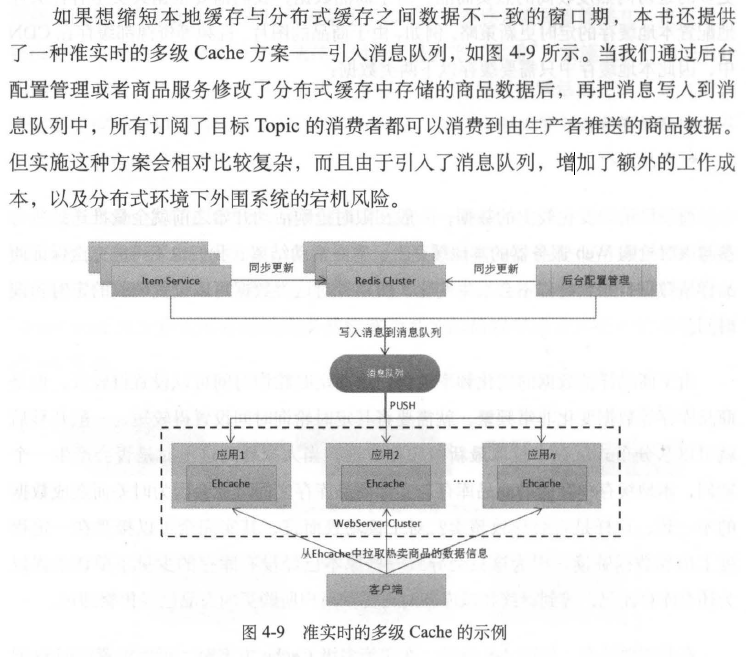
使用ZooKeeper来配置同一热点商品的Key是一个非常不错的选择，此时客户端对其进行监听，一旦Watch到Znode发生变化，所有的客户端全量更新本地持有的Key即可。



LocalCache结合Redis集群的多级Cache方案。可以用来解决限时抢购场景下分布式缓存可能存在的单点瓶颈。可以给后端存储带来更大的保障。

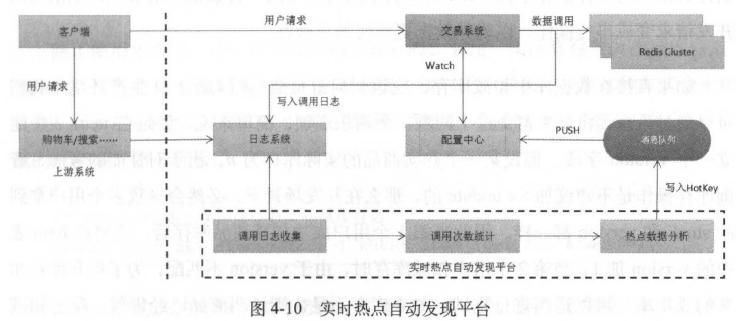


由于本地缓存会共享同一个jvm进程内的heap空间，因此为了避免本地缓存可能带来的种种弊端，不建议把所有的商品信息都缓存到本地的缓存中，如访问热度不高的商品可以直接访问分布式缓存，而本地缓存中存储更多的是访问热度较高的热卖商品。对于商品数据，我们需要根据其类型有针对性地配置本地缓存的定时更新策略。如，由于商品的图片，视频等资源都缓存在CDN中，因此本地缓存中只需要缓存商品详情，商品库存。商品详情等变化较少的数据，一般在限时抢购活动开始之前就全量推送到所有参与限时抢购Web服务器的本地缓存上，直到活动结束。在实施多级Cache方案时，可以设置定时轮询时间，定时从分布式缓存中拉取最新的缓存数据更新到本地缓存中。但千万不要设置 为本地缓存设置 TTL策略，因为这种做法会产生一个问题： 当用户流量过大时，一旦本地缓存的TTL过期，由于本地缓存中没有数据，Web服务器便会直接从分布式缓存中拉取数据并更新到本地缓存上。试想一下，如果大量请求都无法在本地缓存命中，那么这些请求便会全部落到分布式缓存上，然后大量的更新请求会对本地缓存重复写入，这会导致程序的吞吐量严重下降，尤其是变化频率较大的商品库存数据。

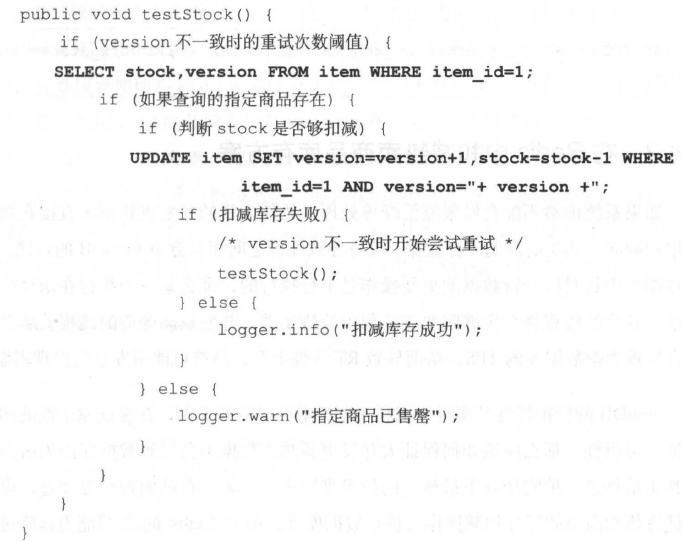


**实时热点自动发现方案**

如何配置热点Key,



Mysql为了保证原子性，InnoDB引擎默认会对同一行数据记录加锁，把前端的并发请求变成串行操作，以确保数据更新时的正确性。如果直接在数据库中扣减库存，应该如何避免商品超卖呢？在生产环境中我们可以通过乐观锁机制来避免这个问题，所谓乐观锁，简单来说，就是在item表中建立一个version字段。所设某一个热卖商品的实际库存为n,出于对性能的考虑，查询库存操作是不建立加for update的，那么在并发场景下，必然会导致多个用户拿到的stock和version都一样。因此当第1个用户成功扣减商品库存后，需要将item表中的version加1；当第2个用户扣减库存时，由于version不匹配，为了提升库存扣减的成功率，可以适当进行重试，如果库存不足，则说明商品卖光了，反之扣减version继续加1。



InnoDB的行锁特性其实是一把利与弊都很明显，在保证原子性的同时降低了可用性。那么应该如何保证大并发更新热点数据不会导致数据库沦为瓶颈，这其实就是秒杀，抢购场景下最核心的技术难题之一。可以尝试将热卖商品的库存扣减操作转移至数据库外。由于Redis的读/写能力远胜过任何类型的关系数据库，因此在Redis中实现库存扣减是一个不错的替代方案。这样数据库中存储的商品库存可以被理解为实际库存，而Reids中存储的商品库存则为实时库存。但在并发较大的情况下，直接在Redis中扣减库存一定会导致商品出现超卖现象，如何避免？引入分布式锁来避免超卖。

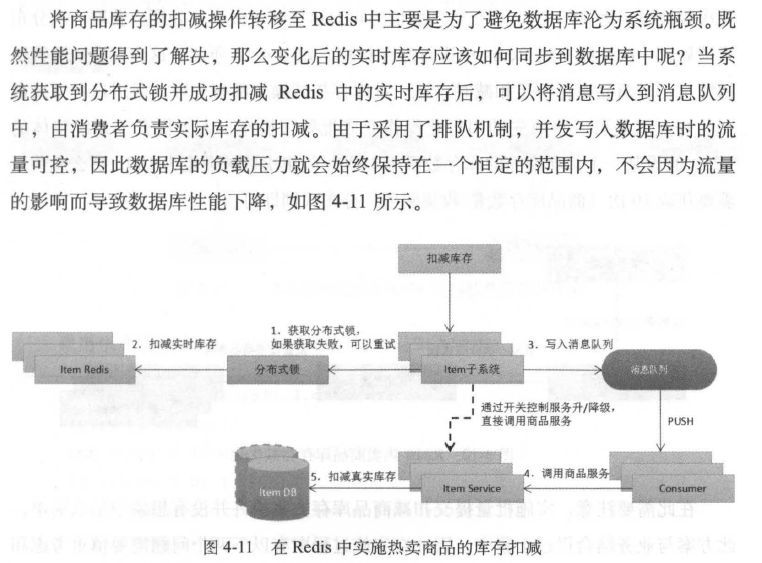
分布式锁必须满足：

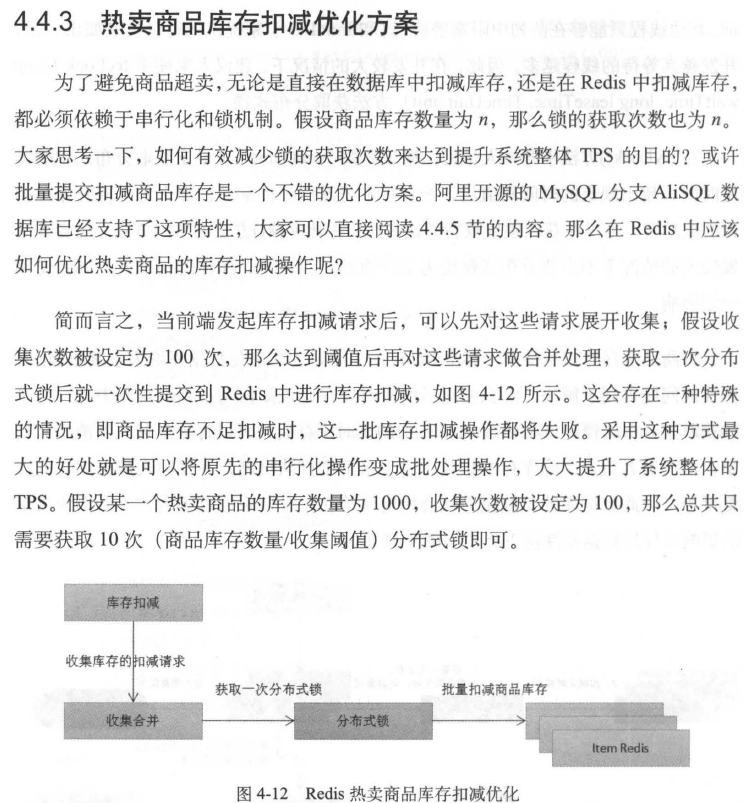
1.在任何情况下分布式锁都不能沦为系统瓶颈。

2.不能产生死锁。

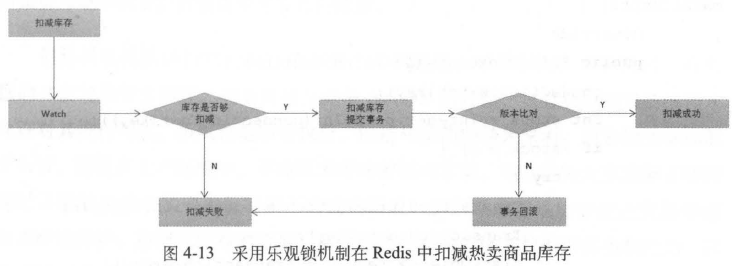
3.支持锁重入。

分布式锁除了ZooKeeper外还有基于ZooKeeper和Redis实现的分布式锁 Redisson，它算得上一款崭新的Redis客户端API,它支持丰富的数据类型，并且是线程安全的，底层使用Netty 4进行网络通信。





在Redis中扣减热卖商品的库存，除了可以使用分布式锁来保证数据的一致性，我们还有其他更优的解决方案，毕竟分布式锁存在较大的性能开销，因此为了能够更好地提升系统的吞吐量，可以考虑采用Redis提供的Watch命令来实现乐观锁。在并发环境下，通过Watch命令对目标Key进行标记后，当事务提交时，如果监控到目标Key对应的值已经发生了改变，那么就意味着版本号发生了改变，因此这一次的事务提交操作就需要进行回滚。



采用乐观锁在Redis 中扣减库存的伪代码如下所示：

Jedis jedis = jedisPool . getResource ();

jedis . watch(key) ;

int stock= Integer . parseint(jedis . get(key)) ;

if (stock > 0) {

Transaction transaction= jedis.multi();

Transaction.decr (key );

List<Object> result = transaction.exec();

if (result != null && !result . isEmpty()) {

System.out.println("抢购成功");

}

jedis.unwatch();

}

jedisPool.returnResource(jedis);

在应用层实现排队机制，将热卖商品的库存扣减操作设置队列串行执行，这样便可以降低大促场景下针对同一热卖商品并发写的流量，虽然对于集群环境中的单个节点而言，库存的扣减操作是串行执行的，但是从整体上来看仍然是并发执行的，由集群环境中的节点数量决定并发写的流量。

但是单机排队串行写的执行效率似乎并不高效，尽管数据库中针对同一行数据的并发写操作多线程之间需要相互竞争InnoDB 的行锁，只有获取到锁的线程才允许对其进行写人。虽然也是串行执行，但是从压测的结果来看，前者的效率远低于后者，因此在生产环境中，不建议采用这种解决方案。

**数据库分库分表案例**

分库分表中间件有淘宝的TDDL,Shark,MyCat.

关系型数据库常见的性能瓶颈主要有两个:

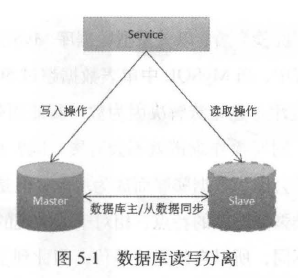
. 大量的并发读／写操作，导致单库出现难以承受的负载压力；

．单表存储数据量过大，导致检索效率低下。

数据库读写分离

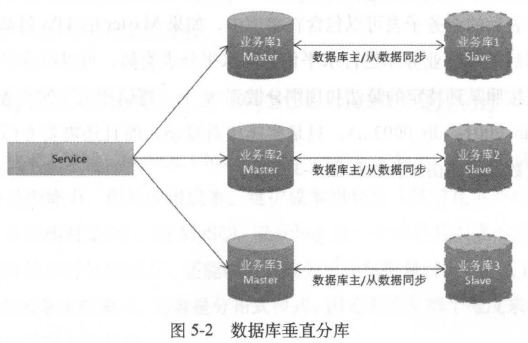
读写分离状态（生产环境一般会采用一主一从或一主多从），由Master 负责写操作，而Slave作为备库，不会开放写操作，但可以允许读操作，主从之间保持数据同步即可。根据二八法则， 80%的数据库操作是读操作，剩下的20%则为写操作。读写分离后，可以大大提升单库无法支撑的负载压力.Master 与Slave 数

据库之间数据同步是会存在一定延迟的，因此在写人Master 之前最好将同一份数据落到缓存中，以避免高并发情况下，从Slave 中获取不到指定数据的情况发生。



数据库垂直分库

随着用户规模的线性上升，系统瓶颈一定会暴露。因此到了这个阶段， DBA 会开始对数据库进行垂直分库。所谓垂直分库，就是企业根据自身业务的垂直划分，将原本冗余在单库中的数据表拆分到不同的业务库中，实现分而治之的数据管理和读／写操作.



以关系型数据库MySQL 为例，由于单一业务的数据信息仍然落盘在单表中，当MySQL 中单表数据超过500 万行时，读操作就会逐渐成为瓶颈，哪怕索引重建， 也无法解决因为数据膨胀而带来的检索效率低下等问题。大家可能会有疑问，对写操作难道就不会有影响吗？由于写是顺序写，所以基本上数据库的写人操作不会因为数据膨胀而成为瓶颈，但是读操作一定会存在上限。这其实也是RDBMS 等类型数据库的特点，相对于非常规的NoSQL 数据库而言，由于双方的底层存储架构不同，所以自然无法等价。因此到了这个阶段， DBA 就需要在垂直分库的基础上实施水平分表。

数据库水平分库与水平分表

当大家对数据库的读写分离和垂直分库等概念有了一定的了解后，笔者再为大家介绍互联网场景下关系型数据库中应对高并发、单表数据量过大的最终解决方案。水平分表就是将原本冗余在单库中的单个业务表拆分为n 个“逻辑相关”的业务子表（如tab\_OOOO 、tab\_OOOI 、tab 0002 ... ），不同的业务子表各自负责存储不同区间的数据，对外形成一个整体，这就是大家常说的Sharding 操作。

水平分表后的业务子表可以包含在单库中，如果Master 的TPS 过高，则还可以对垂直分库后的单一业务库进行水平化。

在生产环境中实施分库分表操作，主要是为了解决高并发场景下单库的性能瓶颈，并充分利用分布式的威力提升数据库的读／写性能。假设后续业务表中的数据量又一次达到存储阑值并对性能产生影响时，DBA只需要再次对现有的业务库和业务表横向扩容，并迁移数据即可。目前，一些大型的互联网电商企业内部都对分库分表场景有着广泛的应用和实施经验。

**MySQL Sharding 与MySQL Cluster 的区别**

MySQL Cluster 只是一个数据库集群，其优势只是扩展了数据库的并行处理能力，但是使用成本、维护成本相当高（目前在生产环境中使用者较少，而且实施相对复杂）。而MySQL Sharding 是一个成熟且实惠的方案，不仅可以提升数据库的并行处理能力，还能够解决因为单表数据量过大所产生的检索瓶颈。简而言之，前者是集群模式，后者是分布式模式，因此无论从哪个维度来看， Shardin都是互联网当下最好的选择。

Sharding 中间件

一旦数据库实施分库分表后，开发人员就必须思考两个问题。首先，必须明确定义SQL 语句中的Shard Key （路由条件），这非常重要，因为路由维度直接决定了数据的落盘位置。其次，应该如何根据所定义的Shard Key进行数据路由，这需要定义一套特定的路由算法和规则。当这两个问题成功解决后，就能够让任何一条SQL 语句的读／写操作准确定位到具体的业务库和业务表中去执行开发人员预期的操作。



**Shark**

Shark 是一款采用Apache License 2.0 开源协议的分布式MySQL 分库分表中间件。Sharding领域的一站式解决方案，具备丰富、灵活的路由算法支持，能够方便DBA实现库和表的水平扩容，以及有效降低数据的迁移成本。Shark 每天为不同的企业、业务提供过亿级别的SQL 读／写服务。

· 完善的技术文档支持；

· 动态数据源的无缝切换：

· 丰富、灵活的分布式路由算法支持：

·非Proxy 架构， 应用直连数据库，降低外围系统依赖所带来的看机风险3

．业务零侵入，配置简单

·站在巨人的肩膀上（ SpringJDBC 、基于Druid 的Sqlparser 完成SQL 解析），

执行性能高效、稳定：

·提供多机SequenceID 的API 支持，解决多机SequenceID 难题：

·默认支持基于ZooKeeper 、Redis3.x Cluster 作为集中式资源配置中心：

· 基于Velocity 模板引擎渲染内容，支持SQL 语句独立配置和动态拼接，与业

务逻辑代码解稠；

·提供内置验证页面，方便开发、测试及运维人员对执行后的SQL 进行验证；

·提供自动生成配置文件的API 支持，降低配置信息出错率。

Shark 支持两类分库分表模式，如下所示：

· 单库多表模式$

· 多库多表模式。

**多机Sequence ID 解决方案**

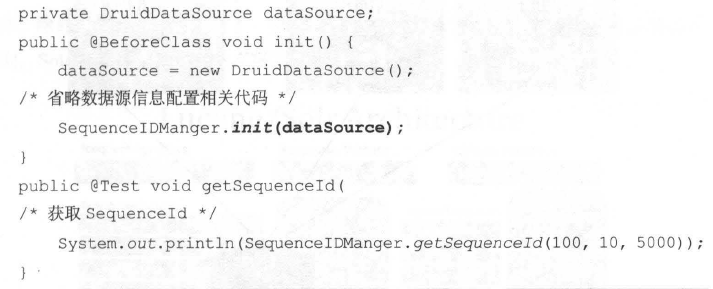
如果只是考虑生成D 的唯一性而忽略掉连续性，开发人员可以利用UUID 、物理机器IP、随机值、时间戳等不同的维度共同生成唯一的ID 。既然要兼顾生成ID的唯一性和连续性，那么依赖一个独立的外围单点系统来负责完成则不失为一个可取的方案。

Shark 内部已经提供了生成Sequence~D 的API （底层支持数据库和ZooKeeper 作为申请SequenceID 的存储系统），开发人员只需要调用即可.

使用Shark 提供的DDL 语句在单库中创建所需的数据表，如下所示：



Shark 的ID 生成器也支持配置数据库连接池



Shark 提供生成SequenceID 的API 只是封装了ID 的生成逻辑，真正保证唯一性

和连续性的还是单点数据库（想要提升容灾能力和出，可搭建Master/Slave 模式） 。

上述程序中的getSequenceid(int idcNum,int type,long memData）方法包含3 个参数，

第1 个参数是IDC 机房编码，用于区分不同的IDC 机房；第2 个参数是业务类别， 2

位数字长度；第3 个参数是向数据库申请的ID 缓存数。最终这段代码会返回一个数

据类型为long、数值长度为19 位的SequenceID。数值长度为19 位的SequenceID 已

经能够满足绝大多数企业后续多年的使用了。

或许当大家看到这里时，会有一头雾水的感觉，接下来笔者就为大家阐述Shark

生成SequenceID 的原理。如果每一个应用系统中都依赖Shark ，也就意味着每一个

应用系统都间接集成一个ID 生成器，并统一指定一个单点存储系统作为存储介质，

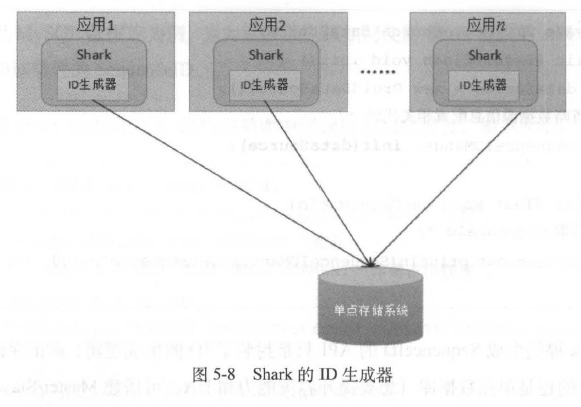
如图5-8 所示。数据库主要负责存储当前ID 序列的最大值，每次每个ID 生成器从数

据库中申请ID 时，都会通过行锁机制来确保并发环境下数据的一致性。或许大家会

产生一个疑问，是否存在性能瓶颈？如果每生成一个ID 都去数据库申请必然性能低

下，但是Shark 的田生成器一次会从数据库中取出一段ID ，然后缓存在本地，这样

就不用每次都去数据库中申请了。



想要提升Shark 生成SequenceID 的性能，可以调整getSequenceidO方法中的第3

个参数，该数值越大ID 生成效率越高，但是一旦B 生成器取了一段D 后，系统突

然若机了，那么这一组B 将会废弃，并且后续可向数据库申请的有限D 缓存数将

越来越少（ SequenceID 的最大值不可超过二进制位数64 位long 类型的最大值

9223372036854775807 ），因此这需要大家仔细权衡，笔者建议将其值设置在“5000 ～

10000”较为合理。

**使用Solr 满足多维度的复杂条件查询**

关于分布式事务

目前较常见的三种分布式系统中的一致性协议如下：

两阶段提交协议（ two phase commit protocol,2PC)；

三阶段提交协议（ three phase commit protocol,3PC)；

Paxos 协议。

由于事务时间、资源锁定时间延长，会导致资源等待时间变长，无关紧要的数据丢失就丢失了，不一定需要分布式事务。

数据库的HA(高可用)方案

集群也是一种HA方案；

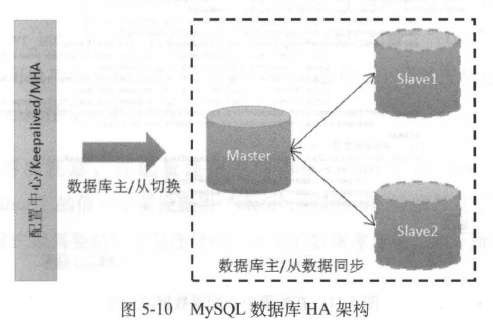
如果Tomcat 这类Web 容器需要部署集群，前端还需要引人反向代理服务器来分

发请求，在互联网领域使用Nginx+ Tomcat 这种形式的集群模式非常常见，由Nginx

负责请求分发、负载均衡，以及Fai lover 等任务。数据库如果需要搭建队，也需要

一种机制保证主从的正常切换，保证Master 者机后S lave 能够开放读写权限，充当

新Master 的角色



基于配置中心实现主从切换z

基于Keepalived 实现主从切换；

基于MHA 实现主从切换。

**基于配置中心实现主从切换**

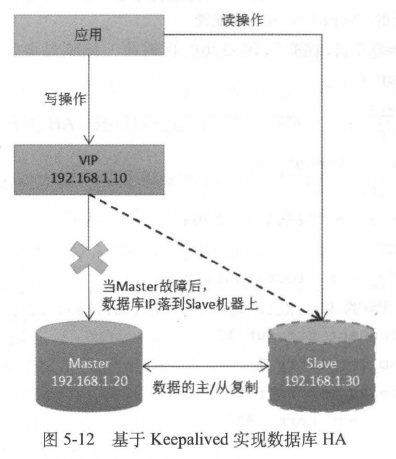
数据库实现主从切换通常有两种形式比较常见， 一种是通过监控系统发出告警后，运维人员手动修改数据源信息，另一种则是Master 实例发生故障后自动切换到Slave 库上。

<http://blog.csdn.net/wm5920/article/details/51863201>

基于Keepa lived 实现主从切换

Master机器和Slave机器上都装有Keepalived程序；

Master 机器和Slave 机器上的Keepalived 程序会相互发送心跳信号，确认对方状态是否存活， 一旦Master 实例出现故障，根据所配置属性“ delay\_loop ” 的值， Keepalived 会在指定的时间范围内定时检查数据库的健康状态，一旦发现异常， Master 机器上的Keepalived 会选择“自杀”，这时Slave 机器上的Keepalived 由于检测不到心跳，便会开始接管VIP 请求，这样所有的写人请求就会全部落到Slave 库上。



我们可以在Keepalived 的配置文件中通过属性“notify\_down”来指定当检测到

数据库实例宕机后需要执行的脚本（比如将Slave 库的读写权限打开）。在此需要注

意，从Master 实例若机到Slave 接管成为新Master 之前的这段时间，数据库暂时不

可用，如果这时有写人请求进来，有可能造成数据丢失。那么怎样才可以避免在数

据库主从切换的过程中不出现数据丢失呢？

在大部分情况下，为了缓解数据库的查询压力，数据都是优先插人到缓存服务

器中再插入到数据库的。那么我们可以通过指定Job 程序定时增量检查数据库与缓存

数据是否一致，如果不一致，则可以利用缓存数据对数据库实施补偿。而且，在某

些场景下，为了消峰我们采用异步模型将数据插入到数据库，比如先将数据写人到

消息队列，然后消费端消费到消息后再写入到数据库。因此，当消费消息后插入数

据到数据库失败时，可以通过failover 机制尝试多次等一系列手段来保证数据库主从

切换过程中数据尽可能不丢失。

**保障主从切换过程中的数据一致性**

在数据库主从切换的过程中，我们可以利用一系列的辅助手段来避免在数据库

不可用时造成数据的丢失，但是在数据库主从切换过程中，还存在一个棘手的问题，

那就是Master 实例若宕机后， Slave 变为新的Master 时，主从数据肯定不一致了，此

时如何保证主从数据的一致性呢？

在解决这个问题之前，我们首先回到主从同步这个问题上，在Master 的TPS 较

高的情况下，主从同步的延迟肯定是非常大的，因此为了避免数据库读写分离后应

用层无法从Slave 拉到实时数据，通常的做法是在写人Master 之前也将同一份数据

落到缓存中，以避免高并发情况下，从Slave 中获取不到指定数据的情况发生。

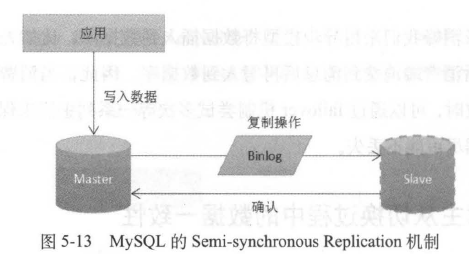
这是MySQL 在5.5 版本时开始提供的一个功能，半同步复制可以理

解为主从之间的强制数据同步，以保证主从数据的准实时性。简单来说，当事务提

交到Master 后， Master 会等待Slave 的回应，待Slave 回应收到Binlog （二进制日志）

后， Master 才会响应请求方已经完成了事务。在峰值流量较大的场景下，笔者不建

议开启这项功能，这会对TPS 产生一定的影响。



可以使用MySQL 在5.6 版本开始提供的GTID （全局事务ID ）特性。由于新Master 是之前的Slave ，而若机后的Master 在重启后可以作为Slave 存在，可以依靠GTID 特性来保证主从之间数据的最终一致性。

可以同步写入或异步写入，优先将数据写入到t\_order\_ buyer 表中会更好，因为哪怕后续t\_order\_ seller 表写人数据失败，至少用户还可以继续推动订单状态的流转（如对订单完成支付），反之如果用户看不到系统新生成的订单，那么对于商家来说似乎也是没有任何意义的。

**订单业务冗余表需求**

主要是为了高并发时的查询，满足订单业务分库分表后不同维度的查询。所以我们的做法是对同一份订单数据进行冗余存储，即我们需要维护两张订单表，一张是卖家订单表，另外一张是买家订单表。

#卖家订单表

t\_order\_seller(seller\_id,buyer\_id,order\_id);

#买家订单表

t\_order\_buyer(buyer\_id,seller\_id,order\_id);

卖家订单表以seller\_id为维度进行数据落盘，买家订单表则以buyer\_id为维度进行数据落盘，可以满足双方的订单查询 。

**冗余表的实现方案**

既然采用数据冗余方案可以解决和满足订单业务分库分表后不同维度的查询需求，接下来就为大家讲解冗余表需求的具体实现方案。实现订单数据的冗余存储，就是同时向t\_order\_ sell er 表和t\_order\_ buyer 表中插人同一份订单数据。数据的写入过程，可以分为以下两种形式：

· 数据同步写人。

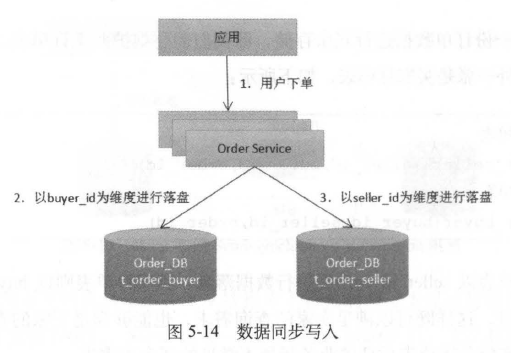
· 数据异步写人。

业务上实现订单数据的落盘，通常的做法是由接人层调用订单服务，然后由订

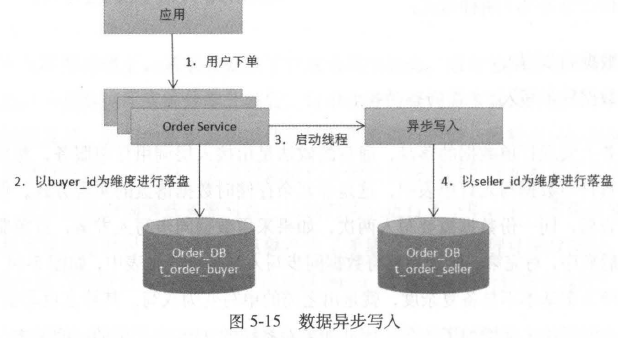
单服务将订单数据写人订单表中，这是非冗余存储时数据落盘的实现方式。但是实

现冗余表后，同一份数据需要写人两次，如果采用数据同步写入方案， 订单服务会

按照先后顺序，写完第一张表后再将数据同步写入到另外一张表中

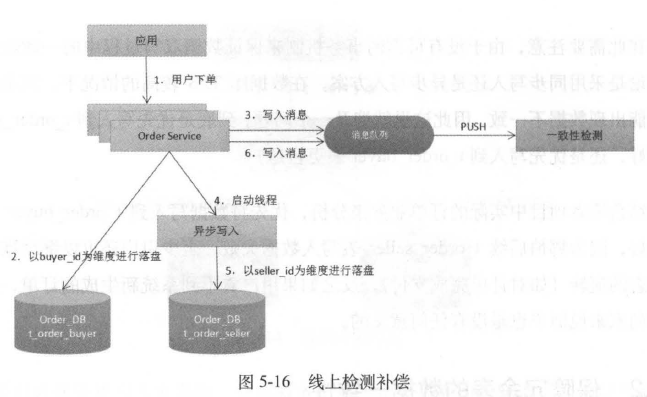


采用数据异步写人方案后，订单服务不再采用同步双写落盘数据，而是当服务成功将数据写人到第一张表后，通过异步模型将数据写入到第二张表中（比如启动一个异步线程负责写人，或者写人消息队列后由消费者负责写人），

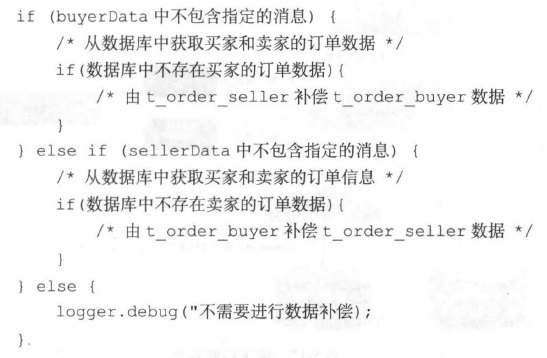


**保障冗余表的数据一致性**

在冗余表需求中，基于数据同步写人和数据异步写人两种方案，都可能在数据写人时产生失败，从而导致t\_order\_ seller 表和t\_order\_ buyer 表之间的数据产生不一致的情况发生，那么最终一致性方案应该如何实施呢？简单来说，当订单数据成功写人t\_order\_ buyer 表后，立即将消息写入到消息队列中，成功写人t\_order\_ seller 表后，再把相同的消息也写人到消息队列中，消费者消费到第1 条消息后，如果在指定的时间范围内没有消费到第2 条消息，就可以认为数据已经产生了不一致，需要执行数据补偿操作，这种补偿方案称为“线上检测补偿”



因此数据补偿之前， 需要优先执行**幕等**操作



如果你觉得“线上检测补偿”方案实施比较复杂， 并且还需要部署其他外围系统来协作，容易增加系统整体的岩机风险，笔者还提供了另外一种补偿方案，那就是基于增量日志扫描的线下检测。

当订单数据成功写人t\_order\_ buyer 表后，立即将数据写人到log l 中，成功写人t\_ order\_ seller 表后，再把相同的数据写人到log2 中，通过指定一个Job 程序不停地增量比对logl 和log2 ，如果出现数据不一致的情况，则进行数据补偿。

相对于“线上检测补偿”方案来说，“线下检测补偿”方案更易实现，但是数据不一致的窗口期相对也会更长，至于程序中到底应该使用哪一种数据补偿方案，则需要结合实际的业务场景而定。

