Linux常用命令

netstat -anp | grep pid 查看进程占用端口

netstat -na|grep ESTABLISHED|wc -l

netstat -an会打印系统当前网络链接状态，而grep ESTABLISHED 提取出已建立连接的信息。 然后wc -l统计。

netstat -tan |grep ^tcp |awk '{++a[$6]} END{for (i in a) print i, a[i]}' 统计TCP不同状态连接

cat /proc/sys/net/ipv4/ip\_local\_port\_range 查看可用端口号

mkdir -p 级联创建目录

find / -name src 在根目下查找名为src的目录

yum install -y unzip zip 安装压缩解压

cat -n test.log |grep‘关键字’ 查询关键字行号

cat -n test.log |tail -n +行号|head -n 20 查看关键字行号后20行

useradd -s /bin/bash -d /home/prd -m chenyao -p chenyao 创建用户

更改文件句柄数：

vi /etc/security/limits.conf

\* soft nofile 65535

\* hard nofile 65535

ulimit -n

curl ifconfig.me 查看外网IP

关闭linux防火墙：

/usr/sbin/sestatus -v

更改/etc/selinux/config SELINUX=disabled

Reboot

systemctl disable firewalld

systemctl stop firewalld

setenforce 0

linux修改java运行环境：

export JAVA\_HOME="/usr/local/java/jdk1.8.0\_171"

export PATH="$PATH:$JAVA\_HOME/bin"

export JRE\_HOME="$JAVA\_HOME/jre"

export CLASSPATH=".:$JAVA\_HOME/lib:$JRE\_HOME/lib"

修改完成后，使用source /etc/profile 命令进行更新；

Tomcat中文路径问题

修改系统编码

vi /etc/profile

export LC\_ALL="zh\_CN.UTF-8"

export LANG="zh\_CN.UTF-8"

source /etc/profile

<init-param>

<param-name>listings</param-name>

<param-value>true</param-value>

</init-param>

vi

shift+g 最后一行 gg第一行

搜索字符串 /字符串 下一个，按“n”

ctrl+b 向上移动一屏；

ctrl+f 向下移动一屏；

Docker

cd /home/zbjf/images/report-service/

查询镜像：docker images

强制移除镜像：docker rmi -f <镜像名>

docker rmi -f reportservice

构建镜像:docker build -t <镜像名> <Dockerfile路径>

docker build -t reportservice .

查询容器：docker ps

停止容器：docker stop {id}

后台运行(-d)、并暴露端口(-p):docker run -d -p 127.0.0.1:33301:22 <镜像名>

docker run -d -p 80:8080 reportservice

进入容器

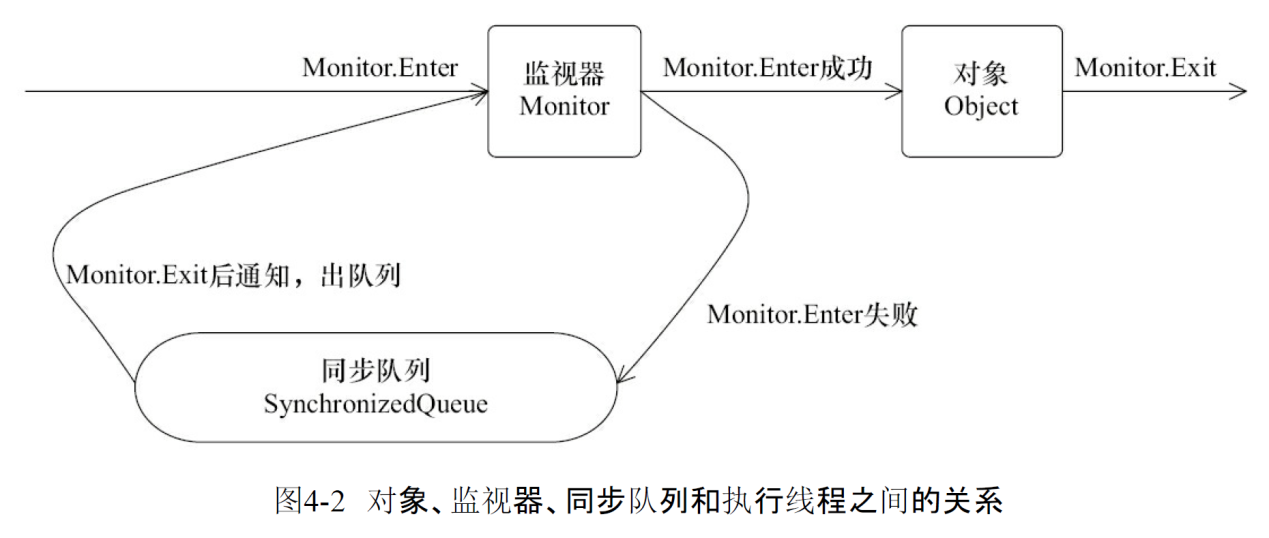
docker ps

docker exec -it passport /bin/bash

生成镜像

docker commit passport cdh64.zbjf.com:5000/test/passport:1127

synchronize锁的原理



简单线程池



线程池的本质就是使用了一个线程安全的工作队列连接工作者线程和客户端线程，客户端线程将任务放入工作队列后便返回，而工作者线程则不断地从工作队列上取出工作并执行。当工作队列为空时，所有的工作者线程均等待在工作队列上，当有客户端提交了一个任务之后会通知任意一个工作者线程，随着大量的任务被提交，更多的工作者线程会被唤醒。

OOM

JVM内存使用量监控:

jstat -gcutil [pid] 2000（ms）

导出JVM 中线程的运行状况：

jstack -F [pid] >jstack.log

导出整个JVM 中内存信息：

jmap -dump:format=b,file=文件名 [pid]

jmap -dump:live,format=b,file=/data1/heap.hprof 8520

查看整个JVM内存状态:

jmap -heap [pid]

查看JVM堆中对象详细占用情：

jmap -histo [pid]

jmap -histo:live [pid] | head -50

高CPU占用

使用top -p pid  -H  查看针对每一个线程占用CPU情况进行查询

其次将需要的线程ID转换为16进制格式：printf "%x\n" tid

最后打印线程的堆栈信息：jstack pid |grep tid -A 50

JAVA\_OPTS="-Xms4096m -Xmx4096m -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:-HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=./java\_pid.hprof"

Es启动报错问题

cat /etc/sysctl.conf | grep vm.max\_map\_count

echo "vm.max\_map\_count=262144" >>/etc/sysctl.conf

sysctl -p使设置生效

cat /etc/security/limits.conf

echo "用户名 hard nofile 65536" >> /etc/security/limits.conf

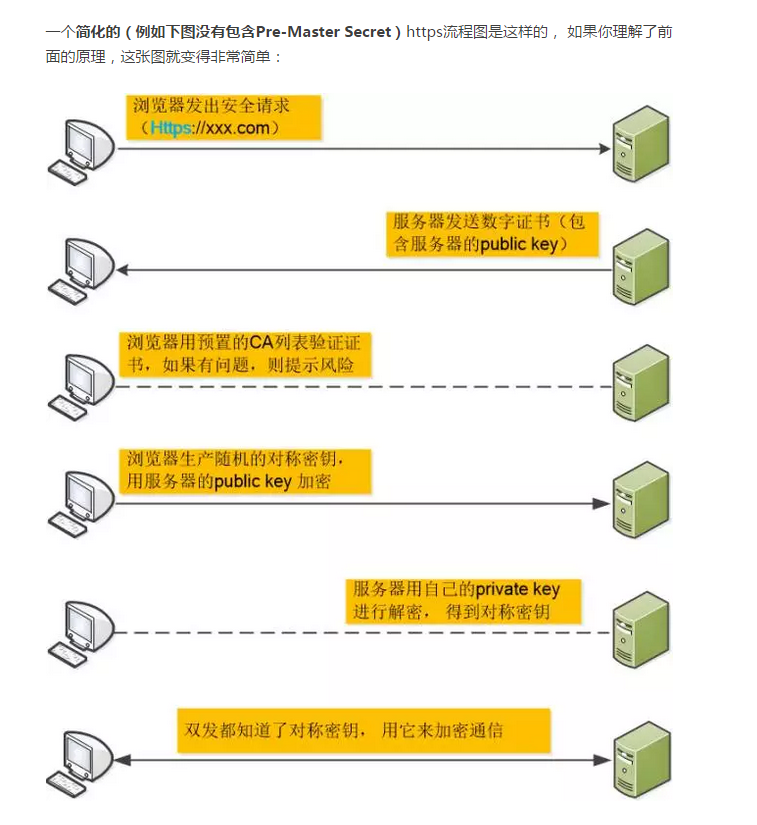
echo "用户名 soft nofile 65536" >> /etc/security/limits.conf

ulimit -Hn

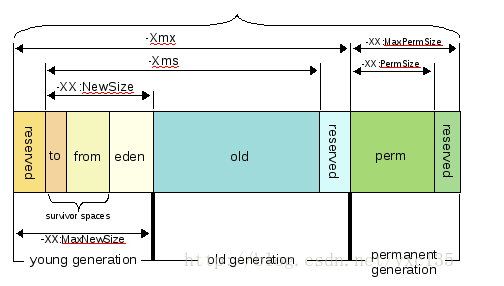
添加账户 useradd XXX

root给XXX账户赋权限，chown -R XXX /目录

https



Jvm参数



-Xmx300m 　　　　　　最大堆大小

-Xms300m 　　　　　　　　初始堆大小

-Xmn100m 　 　　　　　　年轻代大小

-XX:SurvivorRatio=8 　　　　　　Eden区与Survivor区的大小比值，设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10

-XX:+UseG1GC 　　　　　　使用 G1 (Garbage First) 垃圾收集器

-XX:MaxTenuringThreshold=14 　　提升年老代的最大临界值(tenuring threshold). 默认值为 15[每次GC，增加1岁，到15岁如果还要存活，放入Old区]

-XX:ParallelGCThreads=8 　　设置垃圾收集器在并行阶段使用的线程数[一般设置为本机CPU线程数相等，即本机同时可以处理的个数，设置过大也没有用]

-XX:ConcGCThreads=8 　　　　并发垃圾收集器使用的线程数量

-XX:+DisableExplicitGC　　　　　　　　　　禁止在启动期间显式调用System.gc()

-XX:+HeapDumpOnOutOfMemoryError OOM时导出堆到文件

-XX:HeapDumpPath=d:/a.dump 　　 导出OOM的路径

-XX:+PrintGCDetails 　　　　 打印GC详细信息

-XX:+PrintGCTimeStamps 　　　 打印CG发生的时间戳

-XX:+PrintHeapAtGC 　　　　　 每一次GC前和GC后，都打印堆信息

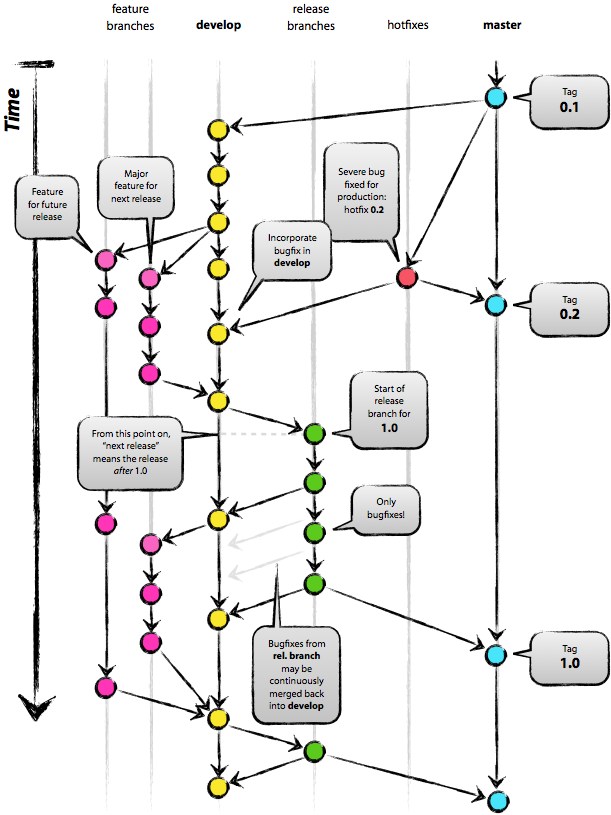
-XX:+TraceClassLoading 　　　 监控类的加载

-XX:+PrintClassHistogram 　　　　　 按下Ctrl+Break后，打印类的信息

-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

Git Flow



主分支 master 永远是可用的稳定版本  
开发分支 develop 是正在测试但未上线的版本

功能和特性开发在 feature 分支进行，完成后合并到 develop 进行测试  
线上紧急的 Bug 修正开 hotfix 分支进行，修正完成后合并到 master

命名

master  
develop  
feature/xxx  
hotfix/xxx

release/版本号  
版本release之前打对应版本的tag

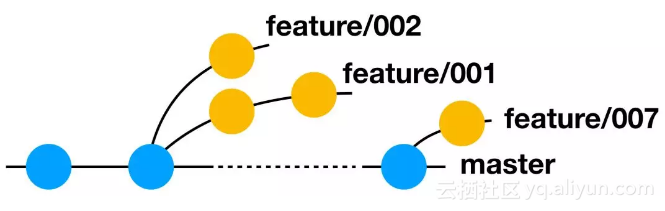
<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

The overall flow of Gitflow is:

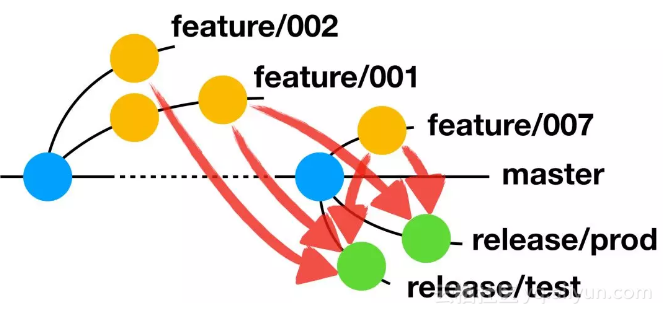
1. A develop branch is created from master
2. A release branch is created from develop
3. Feature branches are created from develop
4. When a feature is complete it is merged into the develop branch
5. When the release branch is done it is merged into developand master
6. If an issue in master is detected a hotfix branch is created from master
7. Once the hotfix is complete it is merged to both developand master

AoneFlow

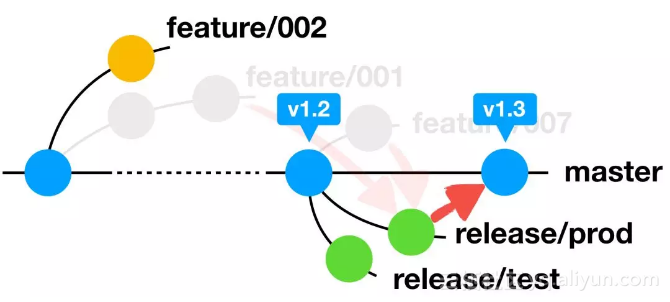
**规则一，开始工作前，从主干创建特性分支。**



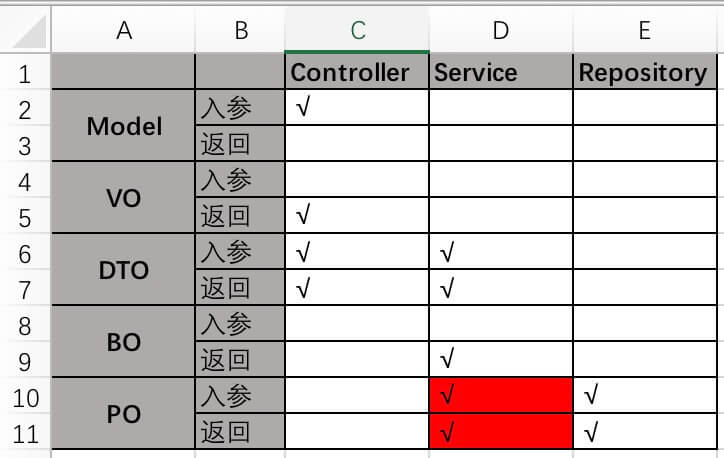
**规则二，通过合并特性分支，形成发布分支。**



**规则三，发布到线上正式环境后，合并相应的发布分支到主干，在主干添加标签，同时删除该发布分支关联的特性分支。**

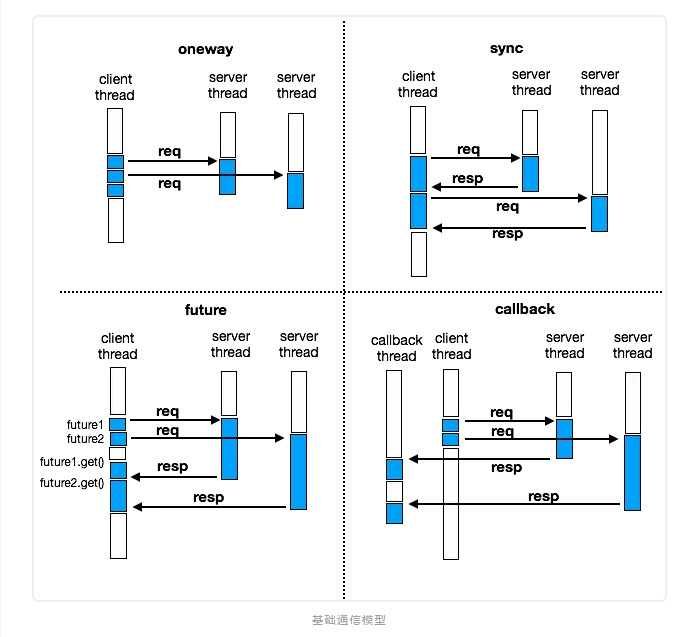


领域模型



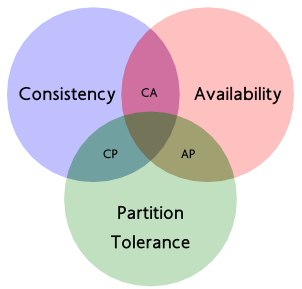
实际项目可以简化，使用 VO + DTO + PO

基础通信模型



CAP理论

一个分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容错性（P：Partition tolerance）这三个基本需求，最多只能同时满足其中两项。



BASE理论

Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）

BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

设计模式

模板方法

装饰者

代理

防缓存击穿

**public** String get(String key) {

String value = redis.get(key);

**if** (value == **null**) { // 代表缓存值过期

// 设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db

**if** (redis.setnx(key\_mutex, 1, 3 \* 60) == 1) { // 代表设置成功

value = db.get(key);

redis.set(key, value, expire\_secs);

redis.del(key\_mutex);

} **else** { // 这个时候代表同时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值即可

sleep(50);

get(key); // 重试

}

} **else** {

**return** value;

}

}

KeepAlive 常见异常

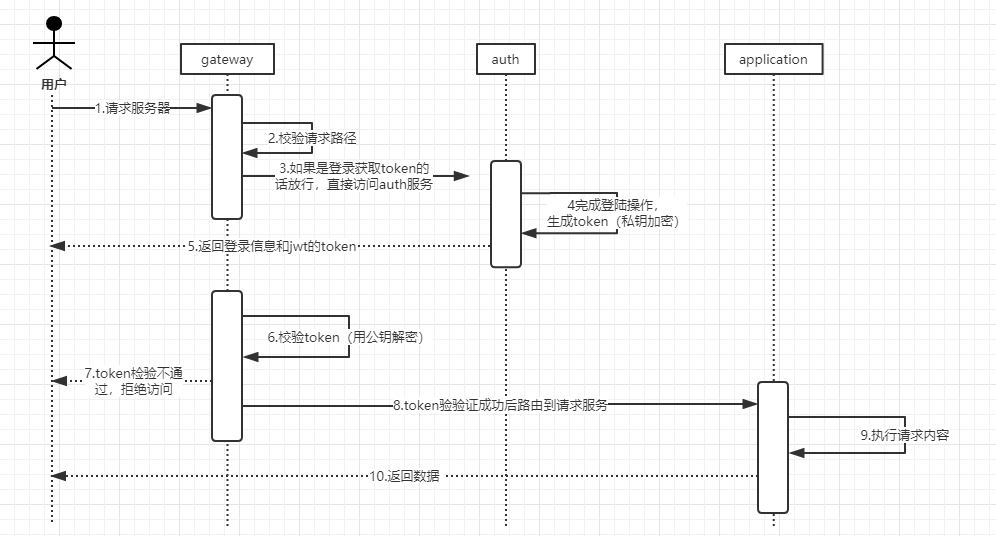
启用 TCP KeepAlive 的应用程序，一般可以捕获到下面几种类型错误

1.ETIMEOUT 超时错误，在发送一个探测保护包经过 (tcpkeepalivetime + tcpkeepaliveintvl \* tcpkeepaliveprobes)时间后仍然没有接收到 ACK 确认情况下触发的异常，套接字被关闭 java java.io.IOException:Connectiontimedout

2.EHOSTUNREACH host unreachable(主机不可达)错误，这个应该是 ICMP 汇报给上层应用的。 java java.io.IOException:Noroute to host

3.链接被重置，终端可能崩溃死机重启之后，接收到来自服务器的报文，然物是人非，前朝往事，只能报以无奈重置宣告之。 java java.io.IOException:Connectionresetbypeer

微服务网关鉴权



JWT 的原理

服务器认证以后，生成一个 JSON 对象，发回给用户，就像下面这样。

{

"姓名": "张三",

"角色": "管理员",

"到期时间": "2018年7月1日0点0分"}

以后，用户与服务端通信的时候，都要发回这个 JSON 对象。服务器完全只靠这个对象认定用户身份。为了防止用户篡改数据，服务器在生成这个对象的时候，会加上签名（详见后文）。

服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展。

特点：

有效使用 JWT，可以降低服务器查询数据库的次数。

JWT 的最大缺点是，由于服务器不保存 session 状态，因此无法在使用过程中废止某个 token，或者更改 token 的权限。也就是说，一旦 JWT 签发了，在到期之前就会始终有效，除非服务器部署额外的逻辑。

JWT 本身包含了认证信息，一旦泄露，任何人都可以获得该令牌的所有权限。为了减少盗用，JWT 的有效期应该设置得比较短。对于一些比较重要的权限，使用时应该再次对用户进行认证。

为了减少盗用，JWT 不应该使用 HTTP 协议明码传输，要使用 HTTPS 协议传输。

缓存

**什么是“Cache Aside Pattern”？**

**答**：旁路缓存方案的经验实践，这个实践又分**读实践，写实践**。

**对于读请求**

先读cache，再读db

如果，cache hit，则直接返回数据

如果，cache miss，则访问db，并将数据set回缓存

**对于写请求**

淘汰缓存，而不是更新缓存

先操作数据库，再淘汰缓存

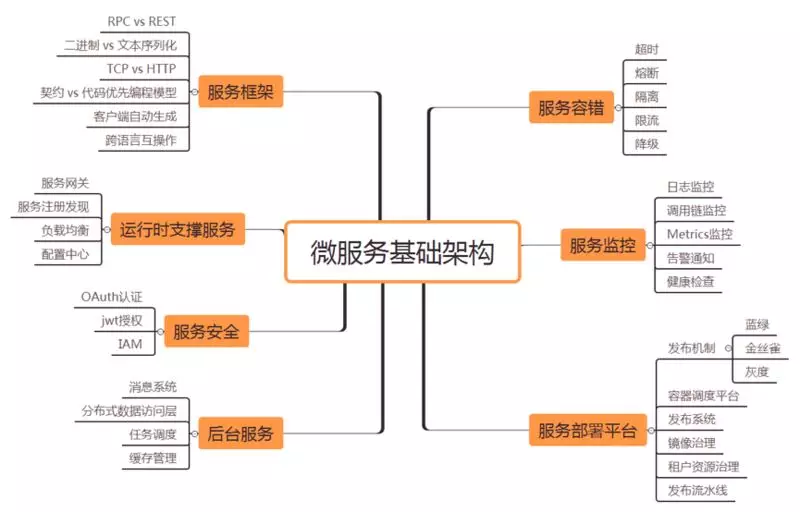
问题：

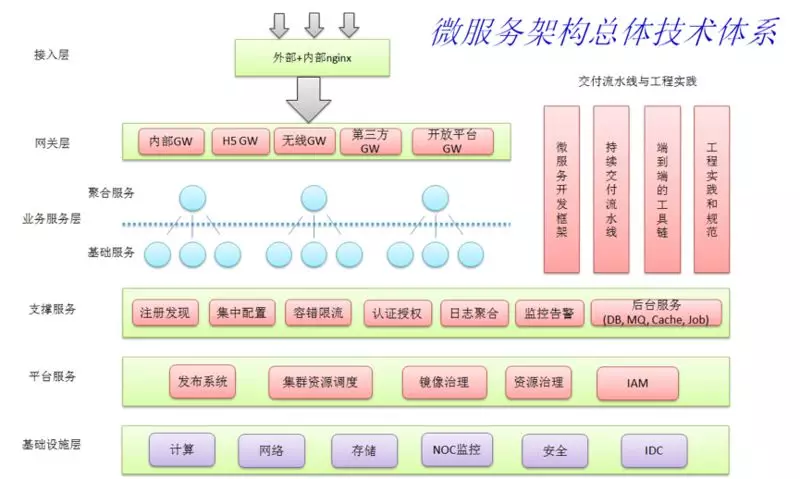
写请求，数据库操作成功，淘汰缓存失败，是否应该回滚数据库？

不应该回滚，利用缓存过期时间兜底，业务上需要允许数据短期不一致

微服务架构

[https://mp.weixin.qq.com/s?\_\_biz=MzI4MTY5NTk4Ng==&mid=2247489150&amp;idx=1&amp;sn=0e754da0ecaa16e16722df27dae26a26&source=41#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzI4MTY5NTk4Ng==&mid=2247489150&amp;idx=1&amp;sn=0e754da0ecaa16e16722df27dae26a26&source=41" \l "wechat_redirect)





# mysql

## [mysql 各数据类型的 大小及长度](https://www.cnblogs.com/ghjbk/p/6681470.html)

数字型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **大小** | **范围（有符号）** | **范围（无符号）** | **用途** |
| TINYINT | 1 字节 | (-128，127) | (0，255) | 小整数值 |
| SMALLINT | 2 字节 | (-32 768，32 767) | (0，65 535) | 大整数值 |
| MEDIUMINT | 3 字节 | (-8 388 608，8 388 607) | (0，16 777 215) | 大整数值 |
| INT或INTEGER | 4 字节 | (-2 147 483 648，2 147 483 647) | (0，4 294 967 295) | 大整数值 |
| BIGINT | 8 字节 | (-9 233 372 036 854 775 808，9 223 372 036 854 775 807) | (0，18 446 744 073 709 551 615) | 极大整数值 |
| FLOAT | 4 字节 | (-3.402 823 466 E+38，1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38) | 0，(1.175 494 351 E-38，3.402 823 466 E+38) | 单精度 浮点数值 |
| DOUBLE | 8 字节 | (1.797 693 134 862 315 7 E+308，2.225 073 858 507 201 4 E-308)，0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308) | 0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308) | 双精度 浮点数值 |
| DECIMAL | 对DECIMAL(M,D) ，如果M>D，为M+2否则为D+2 | 依赖于M和D的值 | 依赖于M和D的值 | 小数值 |

字符类型

|  |  |  |
| --- | --- | --- |
| CHAR | 0-255字节 | 定长字符串 |
| VARCHAR | 0-255字节 | 变长字符串 |
| TINYBLOB | 0-255字节 | 不超过 255 个字符的二进制字符串 |
| TINYTEXT | 0-255字节 | 短文本字符串 |
| BLOB | 0-65 535字节 | 二进制形式的长文本数据 |
| TEXT | 0-65 535字节 | 长文本数据 |
| MEDIUMBLOB | 0-16 777 215字节 | 二进制形式的中等长度文本数据 |
| MEDIUMTEXT | 0-16 777 215字节 | 中等长度文本数据 |
| LOGNGBLOB | 0-4 294 967 295字节 | 二进制形式的极大文本数据 |
| LONGTEXT | 0-4 294 967 295字节 | 极大文本数据 |

枚举集合

ENUM （最多65535个成员）                              64KB  
SET  （最多64个成员）                                 64KB

时间类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **大小 (字节)** | **范围** | **格式** | **用途** |
| DATE | 3 | 1000-01-01/9999-12-31 | YYYY-MM-DD | 日期值 |
| TIME | 3 | '-838:59:59'/'838:59:59' | HH:MM:SS | 时间值或持续时间 |
| YEAR | 1 | 1901/2155 | YYYY | 年份值 |
| DATETIME | 8 | 1000-01-01 00:00:00/9999-12-31 23:59:59 | YYYY-MM-DD HH:MM:SS | 混合日期和时间值 |
| TIMESTAMP | 8 | 1970-01-01 00:00:00/2037 年某时 | YYYYMMDD HHMMSS | 混合日期和时间值，时间戳 |

## mysql数据容量

select

table\_schema as '数据库',

sum(table\_rows) as '记录数',

sum(truncate(data\_length/1024/1024/1024, 2)) as '数据容量(G)',

sum(truncate(index\_length/1024/1024/1024, 2)) as '索引容量(G)'

from information\_schema.tables

group by table\_schema

order by sum(data\_length) desc, sum(index\_length) desc;

select

table\_schema as '数据库',

table\_name as '表名',

table\_rows as '记录数',

truncate(data\_length/1024/1024/1024, 2) as '数据容量(G)',

truncate(index\_length/1024/1024/1024, 2) as '索引容量(G)'

from information\_schema.tables

where table\_schema='indonesia\_yj\_event\_tracking'

order by data\_length desc, index\_length desc;

## mysql查看优化后的语句

EXPLAIN EXTENDED

select t.\* from (select \* from indonesia.t\_account\_manage where user\_id = 3 ) t LEFT JOIN indonesia.t\_loan\_record b on t.user\_id = b.user\_id ;

SHOW WARNINGS;



## Mysql查询优化跟踪

# 1. 打开optimizer trace功能 (默认情况下它是关闭的):

SET optimizer\_trace="enabled=on";

# 2. 这里输入你自己的查询语句

SELECT ...;

# 3. 从OPTIMIZER\_TRACE表中查看上一个查询的优化过程

SELECT \* FROM information\_schema.OPTIMIZER\_TRACE;

# 4. 可能你还要观察其他语句执行的优化过程，重复上边的第2、3步 ... # 5. 当你停止查看语句的优化过程时，把optimizer trace功能关闭

SET optimizer\_trace="enabled=off";

问题：如果binlog写完盘以后发生crash，这时候还没给客户端答复就重启了。等客户端再重连进来，发现事务已经提交成功了，这是不是bug？

回答：不是。

你可以设想一下更极端的情况，整个事务都提交成功了，redo log commit完成了，备库也收到binlog并执行了。但是主库和客户端网络断开了，导致事务成功的包返回不回去，这时候客户端也会收到“网络断开”的异常。这种也只能算是事务成功的，不能认为是bug。

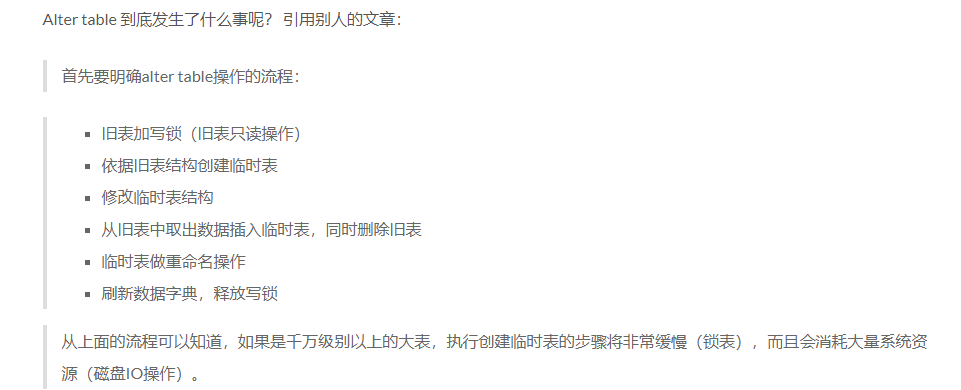
实际上数据库的crash-safe保证的是：

如果客户端收到事务成功的消息，事务就一定持久化了；

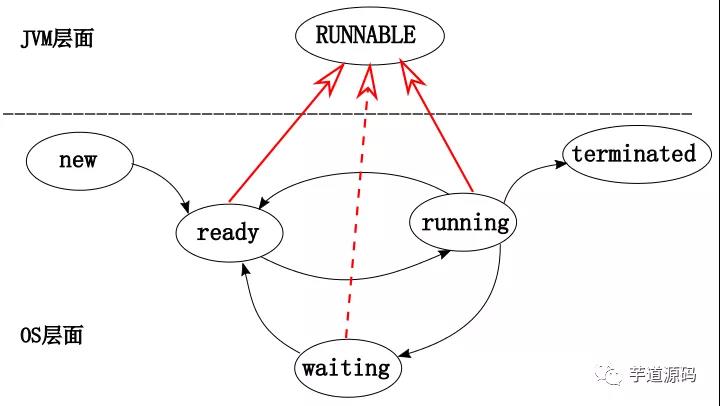
如果客户端收到事务失败（比如主键冲突、回滚等）的消息，事务就一定失败了；

如果客户端收到“执行异常”的消息，应用需要重连后通过查询当前状态来继续后续的逻辑。此时数据库只需要保证内部（数据和日志之间，主库和备库之间）一致就可以了。

## mysql修改表结构



NoSQL选型

Jvm与OS线程状态区别

千万级用户量的压力预估

这个假设这个网站预估的用户数是1000万，那么根据28法则，每天会来访问这个网站的用户占到20%，也就是200万用户每天会过来访问。

通常假设平均每个用户每次过来会有30次的点击，那么总共就有6000万的点击（PV）。

每天24小时，根据28法则，每天大部分用户最活跃的时间集中在（24小时 \* 0.2）≈ 5小时内，而大部分用户指的是（6000万点击 \* 0.8 ≈ 5000万点击）

也就是说，在5小时内会有5000万点击进来。

换算下来，在那5小时的活跃访问期内，大概每秒钟会有3000左右的请求量，然后这5小时中可能又会出现大量用户集中访问的高峰时间段。

比如在集中半个小时内大量用户涌入形成高峰访问。根据线上经验，一般高峰访问是活跃访问的2~3倍。假设我们按照3倍来计算，那么5小时内可能有短暂的峰值会出现每秒有10000左右的请求。

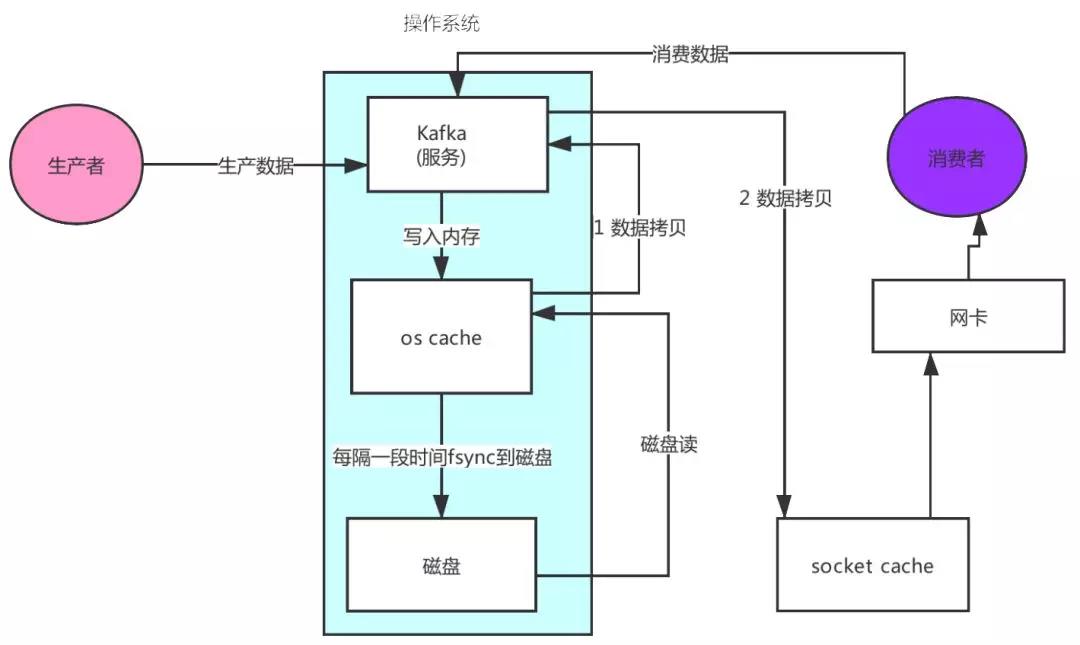
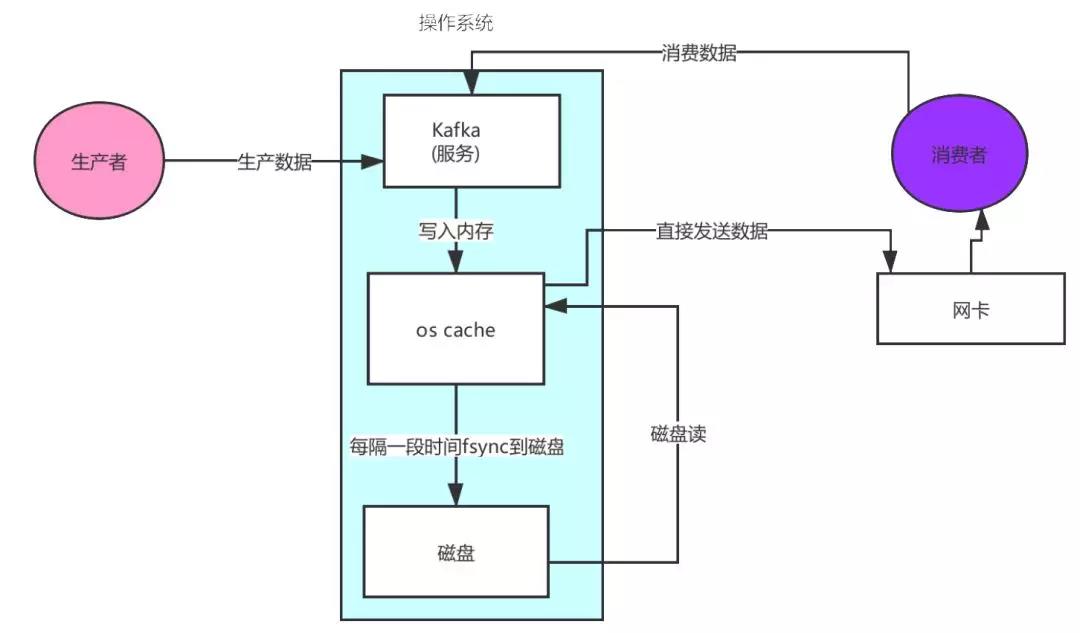
Kafka 性能好在什么地方？

#### ****①顺序写****

Kafka 的设计中，数据其实是存储在磁盘上面，一般来说，会把数据存储在内存上面性能才会好。

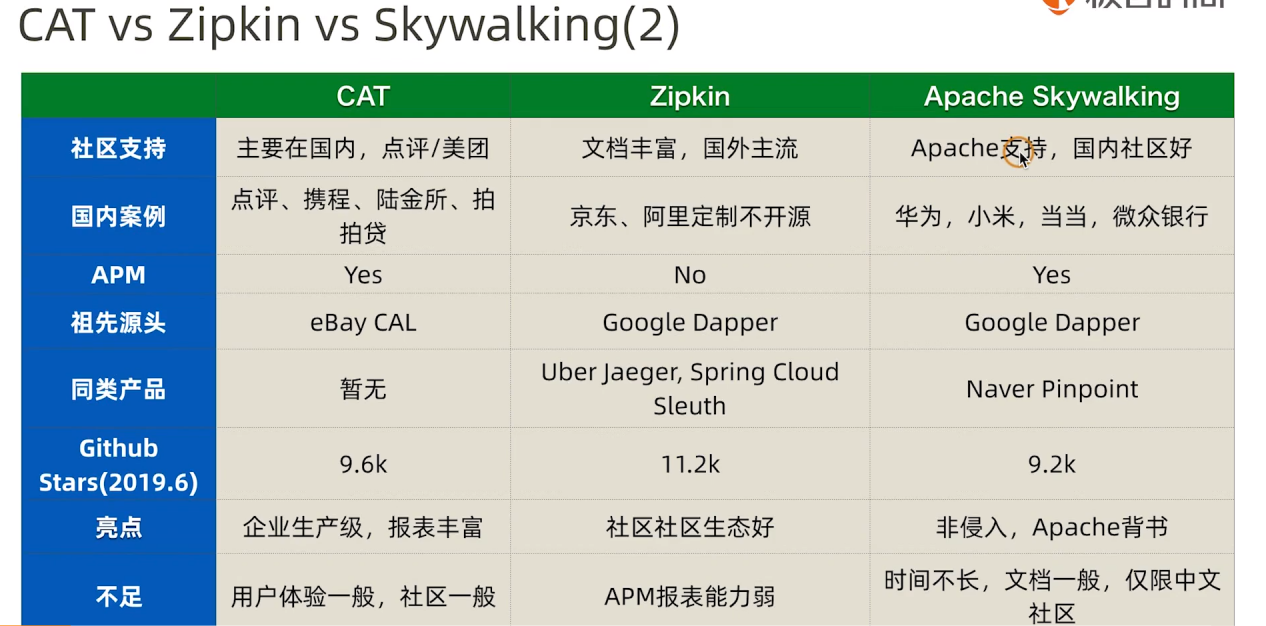
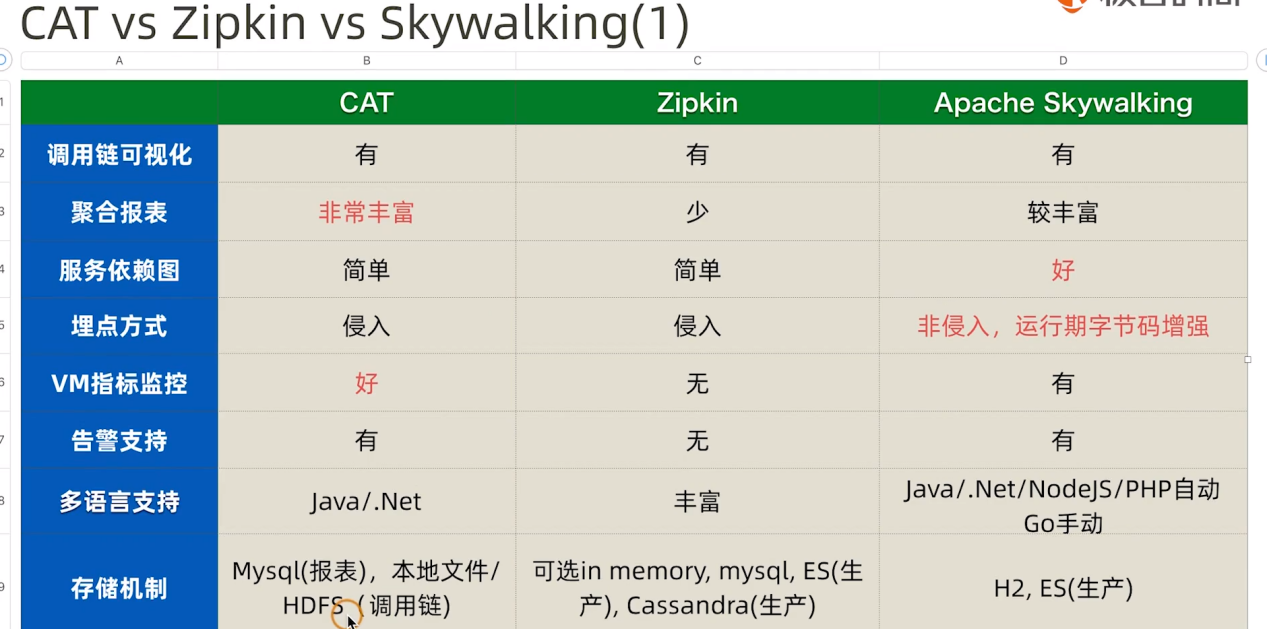
但是 Kafka 用的是顺序写，追加数据是追加到末尾，磁盘顺序写的性能极高，在磁盘个数一定，转数达到一定的情况下，基本和内存速度一致。

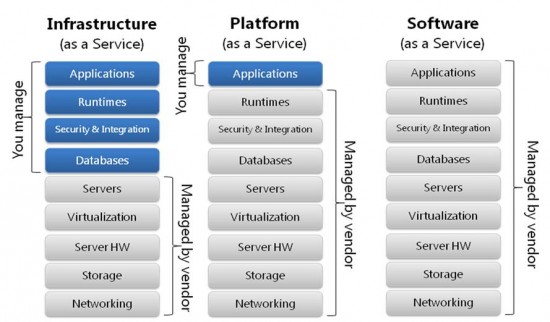
#### ****②零拷贝****

数据的拷贝从内存拷贝到 Kafka 服务进程那块，又拷贝到 Socket 缓存那块，整个过程耗费的时间比较高。  
Kafka 利用了 Linux 的 sendFile 技术（NIO），省去了进程切换和一次数据拷贝，让性能变得更好。

IO操作分两个阶段  
1、等待数据准备好(读到内核缓存)  
2、将数据从内核读到用户空间(进程空间)  
一般来说1花费的时间远远大于2。  
1上阻塞2上也阻塞的是同步阻塞IO  
1上非阻塞2阻塞的是同步非阻塞IO，Reactor就是这种模型  
1上非阻塞2上非阻塞是异步非阻塞IO，Proactor模型就是这种模型

微服务生产就绪

调用链产品比对

云计算IaaS、PaaS、SaaS

应用分层

* 开放接口层：可直接封装Service方法暴露成RPC接口；通过Web封装成http接口；进行网关安全控制、流量控制等。
* 终端显示层：各个端的模板渲染并执行显示的层。当前主要是velocity渲染，JS渲染，JSP渲染，移动端展示等。
* Web层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
* Service层：相对具体的业务逻辑服务层。
* Manager层：通用业务处理层，它有如下特征： - 1） 对第三方平台封装的层，预处理返回结果及转化异常信息； - 2） 对Service层通用能力的下沉，如缓存方案、中间件通用处理； - 3） 与DAO层交互，对多个DAO的组合复用。
* DAO层：数据访问层，与底层MySQL、Oracle、Hbase等进行数据交互。
* 外部接口或第三方平台：包括其它部门RPC开放接口，基础平台，其它公司的HTTP接口。

Kafka、RabbitMQ、RocketMQ等消息中间件的对比

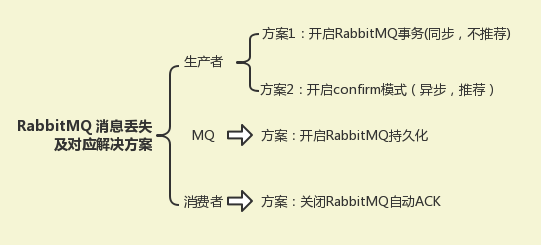
有关测试结论：

Kafka的吞吐量高达17.3w/s，不愧是高吞吐量消息中间件的行业老大。这主要取决于它的队列模式保证了写磁盘的过程是线性IO。此时broker磁盘IO已达瓶颈。

RocketMQ也表现不俗，吞吐量在11.6w/s，磁盘IO %util已接近100%。RocketMQ的消息写入内存后即返回ack，由单独的线程专门做刷盘的操作，所有的消息均是顺序写文件。

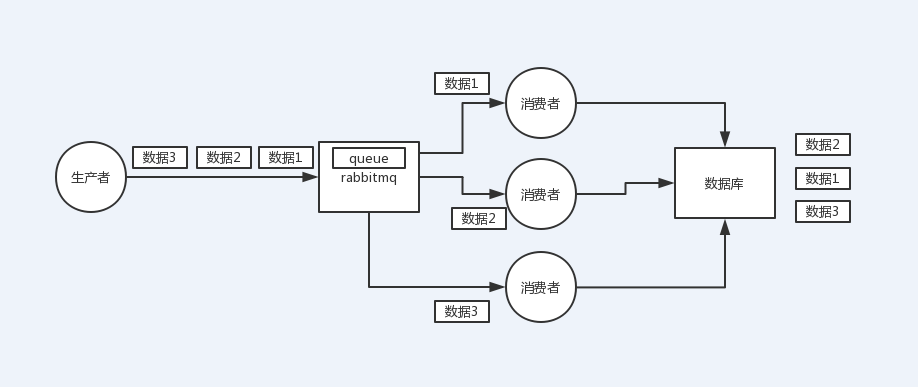
RabbitMQ的吞吐量5.95w/s，CPU资源消耗较高。它支持AMQP协议，实现非常重量级，为了保证消息的可靠性在吞吐量上做了取舍。我们还做了RabbitMQ在消息持久化场景下的性能测试，吞吐量在2.6w/s左右。

RabbitMQ消息防丢

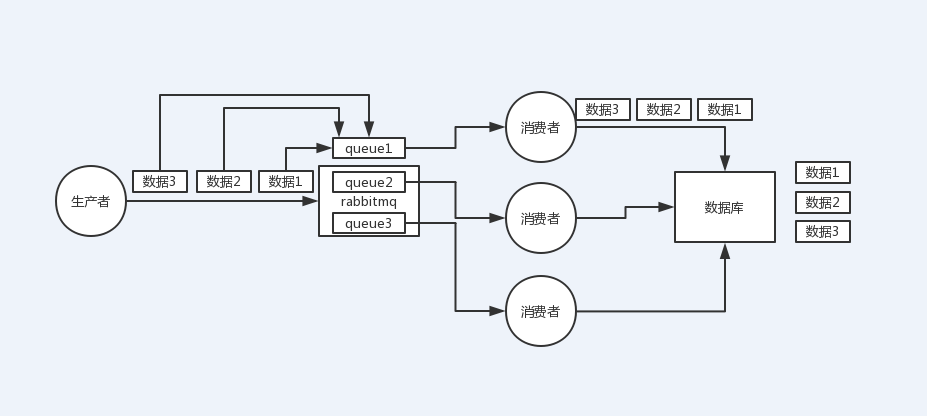


如何保证消息顺序

## RabbitMQ

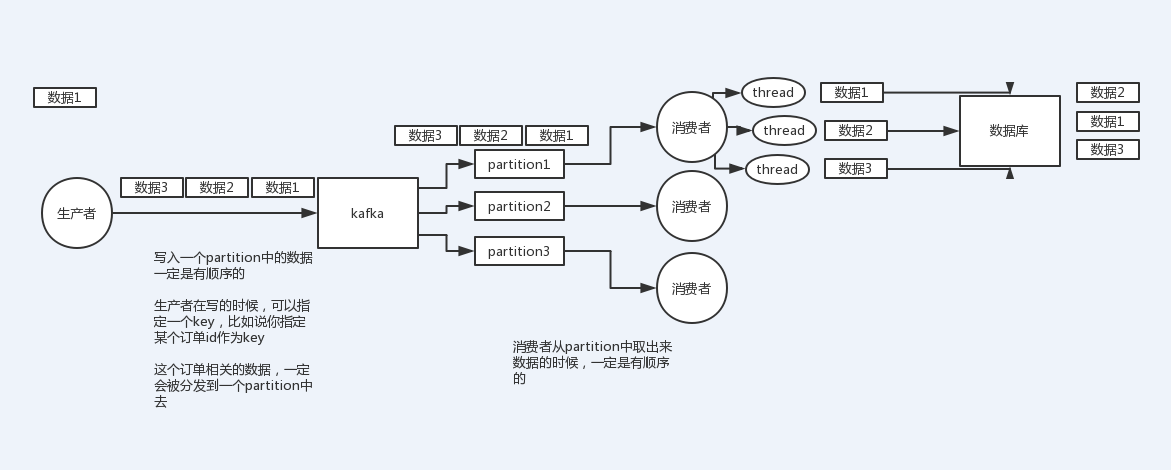


RabbitMQ消息顺序错乱的场景：数据1、2、3按顺序发到一个queue，多个消费者消费同一个queue

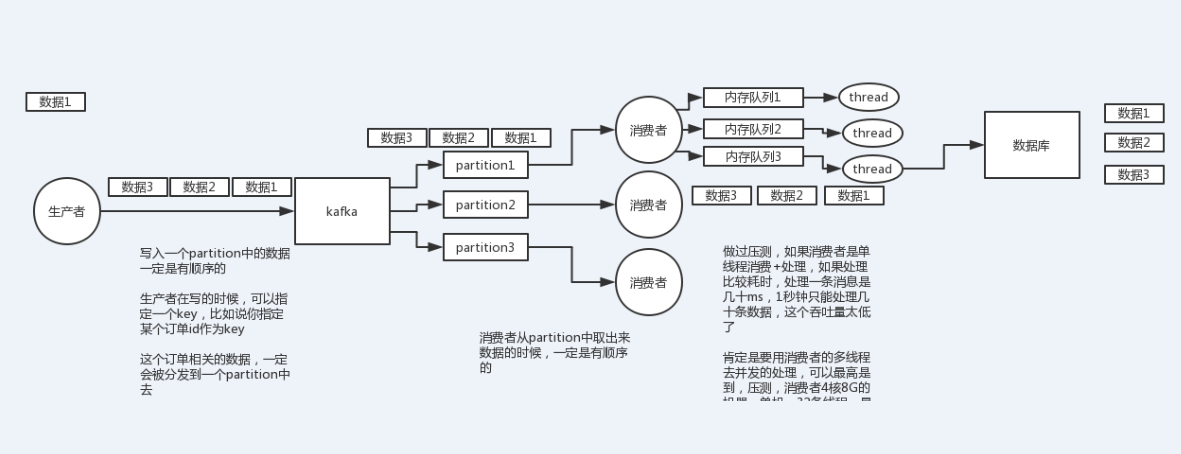


拆分为多个queue，每个queue由一个consumer消费；或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理

## Kafka



　　1个topic，3个partition，3个consumer，每个消费者消费一个partition，需要保证顺序的消息都放入同一个partiton，但是如果一个消费者开启多个线程来处理，还是无法保证消息的顺序性。



解决办法：每个消费者内部设置多个内存队列，对消息的key做hash，将需要保证顺序的消息映射到同一个内存队列中，每个线程负责处理一个内存队列

隔离级别

| **隔离级别** | **脏读** | **不可重复读** | **幻读** |
| --- | --- | --- | --- |
| READ UNCOMMITTED | Possible | Possible | Possible |
| READ COMMITTED | Not Possible | Possible | Possible |
| REPEATABLE READ | Not Possible | Not Possible | Possible |
| SERIALIZABLE | Not Possible | Not Possible | Not Possible |

Spring事务传播机制

1.propagation\_required(支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择)

2.propagation\_supports(支持当前事务，如果当前没有事务，就以非事务方式执行)

3.propagation\_mandatory(支持当前事务，如果当前没有事务，就抛出异常)

4.propagation\_requires\_new(新建事务，如果当前存在事务，把当前事务挂起)

5.propagation\_not\_supported(以非事务方式执行操作，如果当前存在事务，就把当前事务挂起)

6.propagation\_never(以非事务方式执行，如果当前存在事务，则抛出异常)

7.propagation\_nested(如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作)

mysql锁分析

show processlist;

SELECT \* FROM information\_schema.INNODB\_TRX;

SELECT \* FROM information\_schema.INNODB\_LOCKs;

SELECT \* FROM information\_schema.INNODB\_LOCK\_waits;



Insert into a select from b

1.在RR模式下：在按主键进行排序或不排序时，b表S行锁  
2.在RR模式下：在按非主键进行排序时，b表S表锁  
3.在RC模式下：无论按主键还是非主键排序，b表不加锁

微服务划分

拆分的大原则是当一块业务不依赖或极少依赖其它服务，有独立的业务语义，为超过2个的其他服务或客户端提供数据，那么它就应该被拆分成一个独立的服务模块。

****单一职责原则****

* 意思是每个微服务只需要实现自己的业务逻辑就可以了，比如订单管理模块，它只需要处理订单的业务逻辑就可以了，其它的不必考虑。

****服务自治原则****

* 意思是每个微服务从开发、测试、运维等都是独立的，包括存储的数据库也都是独立的，自己就有一套完整的流程，我们完全可以把它当成一个项目来对待。不必依赖于其它模块。

****轻量级通信原则****

* 首先是通信的语言非常的轻量，第二，该通信方式需要是跨语言、跨平台的，之所以要跨平台、跨语言就是为了让每个微服务都有足够的独立性，可以不受技术的钳制。

****接口明确原则****

* 由于微服务之间可能存在着调用关系，为了尽量避免以后由于某个微服务的接口变化而导致其它微服务都做调整，在设计之初就要考虑到所有情况，让接口尽量做的更通用，更灵活，从而尽量避免其它模块也做调整。

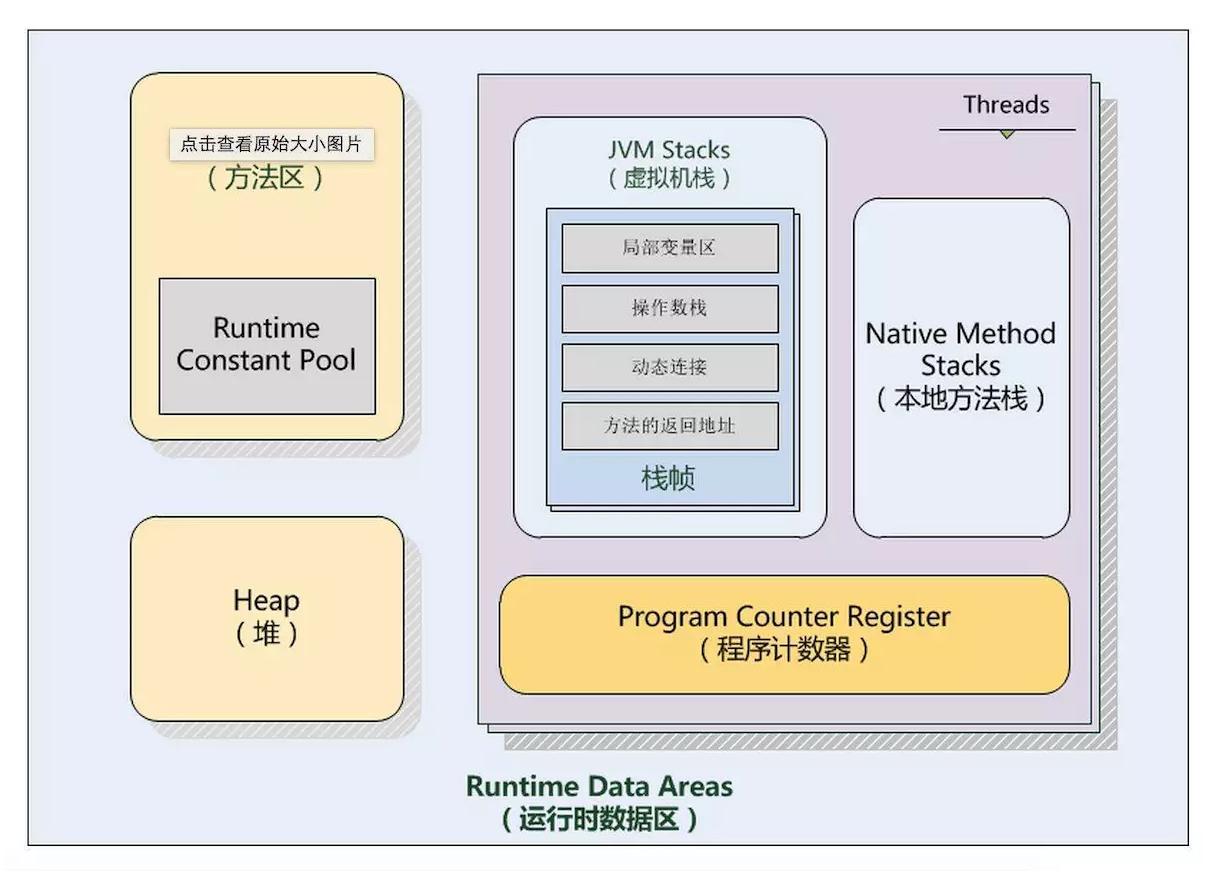
JVM 垃圾收集器

| 收集器 | 串行、并行or并发 | 新生代/老年代 | 算法 | 目标 | 适用场景 |
| --- | --- | --- | --- | --- | --- |
| ****Serial**** | 串行 | 新生代 | 复制算法 | 响应速度优先 | 单CPU环境下的Client模式 |
| ****Serial Old**** | 串行 | 老年代 | 标记-整理 | 响应速度优先 | 单CPU环境下的Client模式、CMS的后备预案 |
| ****ParNew**** | 并行 | 新生代 | 复制算法 | 响应速度优先 | 多CPU环境时在Server模式下与CMS配合 |
| ****Parallel Scavenge**** | 并行 | 新生代 | 复制算法 | 吞吐量优先 | 在后台运算而不需要太多交互的任务 |
| ****Parallel Old**** | 并行 | 老年代 | 标记-整理 | 吞吐量优先 | 在后台运算而不需要太多交互的任务 |
| ****CMS**** | 并发 | 老年代 | 标记-清除 | 响应速度优先 | 集中在互联网站或B/S系统服务端上的Java应用 |
| ****G1**** | 并发 | both | 标记-整理+复制算法 | 响应速度优先 | 面向服务端应用，将来替换CMS |

Elasticsearch、MongoDB、Hadoop选型

如果你仅仅想要通过关键字和简单的分析，那么Elasticsearch可以完成任务；如果你需要查询文档，并且包含更加复杂的分析过程，那么MongoDB相当适合；如果你有一个海量的数据，需要大量不同的复杂处理和分析，那么Hadoop提供了最为广泛的工具和灵活性。

JVM内存模型



加强：

1.分布式事务怎么保证一致性，事务消息，开源TCC

2.分库分表

3.分布式唯一id时钟回滚怎么解决

4.synchronized与ReentrantLock区别

Synchronized字节码层面的锁JVM实现的，java编译后会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。ReentrantLock是JDK实现的

Synchronized可以修飾方法也可以修饰代码块，ReentrantLock只能在方法里面用且需要手动解锁，

synchronized拿到锁只能等待，ReentrantLock拿不到锁可以通过代码让线程干别的事

5.mysql覆盖索引

Select索引中的字段不用回表

6.cms和g1收集器区别

Cms是老年代标记清除，g1年轻带用复制，老年代用标记整理

7.kafka挂了之后数据怎么移动，zk保存kafka的什么信息

进行rebalance，consumer去集群中找partition的replica消费

Zk保存broker，partition节点信息等

8.zk挂了之后

B+树与B-树的区别

1.B+树只在叶子节点存数据单一节点存储更多的索引元素，B-树非叶子节点也存数据，所以B+树更“矮胖”，B-树层级更多IO次数也就更多。

2.所有查询都要查找到叶子节点，查询性能稳定。B-树性能不稳定，最好的情况只查根节点，最坏查到叶子节点。

3.所有叶子节点形成有序链表，便于范围查询。

* 引导类加载器（Bootstrap ClassLoader）

加载java核心类库的C++代码

* 拓展类加载器（Extension ClassLoader）

加载jvm实现的一个拓展目录

* 应用类加载器（Application ClassLoader)

加载程序开发者自己编写的java类

什么是架构？

1 架构是为了应对软件系统复杂度而提出的一个解决方案

2 架构即(重要)决策，在解决方案中选择最优解

3 需求驱动架构，架起需求与设计实现的桥梁

4 对开发成本的考虑

架构师的职责

1.确认需求

需求文档完成后介入，对最终的需求评审和确认，提出需求不清晰和不完整的部分。

2.系统分解

抽离公共模块，分层复用等，考验代码功底。

3.技术选型

负责评估技术选型，出具方案，使用什么框架什么数据库等，之后由项目经理根据人力时间进度等实际情况权衡选择。

4.制定规范

出具技术规格说明书，给开发人员指明方向。

架构师是技术专家，不是业务专家。架构是演进的，架构师需要保证系统不但能够满足当前的需求，还能够快速应对将来的变化。

秒杀设计

从减少请求的角度：

1.资源静态化，把静态资源缓存到CDN，缓存到用户浏览器。

2.前端限流：按钮控制，秒杀前置灰，点击后设置间隔时间，限制点击频率

3.后端限流：网关层IP限流，UserId限流

4.业务上考虑

从扛住请求角度：

1.秒杀服务单独部署，提前租用流量机

2.使用负载均衡，比如Nginx能抗几万并发，tomcat抗几百并发，Nginx负载均衡到多台tomcat上，有条件可以用F5

3.使用缓存，库存预热到redis，这里可能会超卖，所以使用lua脚本进行原子性的库存扣减。

使用MQ削峰，如果库存很多，把从缓存到DB更改库存改成异步。

限流算法



自我介绍

13年毕业到现在差不多七年，最近两年在一家创业公司做架构。去的时候不到十个人，项目也是从0到1的阶段，一开始我负责支付这块接了一些银行和第三方支付。其中引入了一些开源中间件，比如注册中心，配置中心，调度中心。然后发现网关是个裸露的，任何请求都能进来，于是在这里做了统一token鉴权。有些第三方回调需要绕过，所以结合配置中心做了动态黑白名单。后面还做了些验签加密，结合redis lua脚本做了限流。工作中也发现了一些分布式事务不一致的问题，A服务执行成功，B服务执行失败，一开始人工处理效率低下。后来我通过本地消息表和rabbitmq消息确认机制做了个可靠消息的方案，结合ZK节点监听做了消息补偿，下游做幂等保证最终一致性。

最近一年作为项目负责人，带了六七个人做SaaS小贷平台，这里有点偏业务架构，会和产品业务聊需求，把需求转化成实现方案，再让开发来实现。一开始准备用多数据源做多租户的数据隔离，评估下来感觉每个租户一个数据库运维成本会比较大，前期量也不大，后来选择在表里面加商户标识，用mybatis拦截器对sql后面加条件带上租户标识来做，数据不做物理隔离，在一个库里面，节省运维成本。