参与北美期货交易业务清算环节，中间件kafka ELK集群搭建与升级。

面向服务的架构。

订单购物车模块。

将项目中注解和xml混合，改为为完全为注解。方便阅读  
将. 配置类运用是，开启注解，并且指定扫描路径。  
类上注解，告诉容器这个类需要被扫描  
类变量加注解，是依赖注入，容器为我们创建对象，  
构造方法。

畅读spring，mybatice hibernate官方文档。

1.git关注来源项目，喜欢读源码  
CSDN上分享自己的学习经验。虚拟机搭建分布式集群，maven私服，git私服，CICD环境。  
2.具备较好的英文阅读能力和自学能力，

3.保持看官网：畅读spring，mybatice hibernate官方文档。  
4.看源码：看过SSM，常用集合，常用数据类型，juc底层源码  
5.看原理：了解设计模式，JVM，tomacat，mysql底层，ngnix。  
6.工具：熟悉性能测试工具jemeter和jvm调优化工具。  
7.中间件：了解，并有elk，zookper，double，redis.，Kafka，fastDFS，经验

熟悉常用的垃圾收集器，具有实际jvm调优实战经验。

1.1spring结合策略模式，简化多个if，完成单点登录系统，用户名和密码缓存问题.  
1.2spring的单例模式，线程安全问题，juc，nio，jvm的参数调优，工厂模式的运用。  
2.完成单点登录模块，用户名和密码缓(AOP)存问题，redis限制登录功能  
3.订单状态，状态模式  
4.搭建北美地区期货交易清算环节kafka，从activemq上的移植。  
5.网站的全文搜索功能，Elsticse.

Double 到spring cloud的移植。

将已有项目优化为SOA的分布式项目.( spring cloud)

博客网站  
ssm  
tfs  
ngnix  
docker  
tomacat  
redis  
active mq  
elk  
支付。.

java多线程编程主要是两个包  
java.lang包，主要是线程基础类  
java.util包，主要是线程基础类和并发工具类  
对juc和jni有自己的理解.

对spring源码有深入研究，能够对spring进行二次扩展.

静态代理，代理对象代理单一对象执行行为，！代理对象本身在编译期就写好；1.目标对象添加行为需要修改代理对象，代理代理对象2.为不同对象代理，需要写不同的代理对象。  
动态代理，代理对象工厂为不同对象生成相应的代理对象，执行相应操作。！代理对象在运行期生成，1目标对象添加行为，无需对代理工厂做操作，2，对多个对象代理，无需修改代理工厂。

------------------------------------------------------------------------------

JVM

最小最大内存抖动，消耗资源。

多个线程waiting同一个condition，就是死锁。

Top。

 top -Hp进程编号。

1. Vmthread，就看频繁gc，看gc日志，如何定位，回收了，但是每次回收很少，就是问题。  
   2.业务线程，就用jstack  
   3.有内存回收不掉，就是内存泄露。定位问题不难，解决问题南。
2. 生产环境，执行jmap会stop all world，会占用系统资源，jmap128g，不被允许。
3. 测试环境压测，  
   复制到生产环境，执行jmap。
4. 堆内存转储命令，再用工具
5. 1. 负载均衡时，tcp copy把请求发到测试环境，  
   2.生产环境用检测工具，会影响系统性能。在测试环境压测。  
   3.或者设置系统参数，在内存爆时，自动执行堆内存转储。
6. 阿里工具也会影响系统性能，Redfie在线修改class文件
7. java调优一般在测试环境进行。
8. 阿里来源java调优工具。还有压测工具。
9. 线程池导致的堆回收问题
10. 1.设置参数，方便定位问题，设置最佳参数。  
    2.看gc日志，定位问题  
    3. jvm调优工具，定位问题。

虚拟机栈：可以理解为由多个线程栈构成的集合。

一个线程就是开辟一个线程栈空间，线程多会OOM  
循环调用导致线程太大，也会OOM

Xss256k

Spring

构造函数带参实例化的bean，出现循环依赖时没无法解决。

Hashmap 在扩容的时候，才会存在线程安全问题

根据字符串的字符的asII码相加可以得到哈希码。

哈希算法的实质是散列存储。

提高hashmap的实现分为put 和get.

每个entry最起码包含key，value, index,next这几个元素。

Put 是根据key 计算哈希码，对数组长度取模得到index

，通过头插法链表的形式解决hash冲突。

Get 是根据key 计算哈希码，对数组长度取模得到index，遍历链表，当key相同时，取出该元素。

Jdk 1.7之前是数组+链表实现，1.8之后是在元素大于8时为红黑树。

红黑树的优势体现在查询速度比链表。

Kafka storm redis ELK应用开发与升级维护。，

分布式实时计算,高性能高可用API

主要基于springboot与mybatise.

1.APAC EMEA NAN 的dev uat prod cob集群的优化、部署、维护与升级。

2.应用开发

ETL任务处理主要包含数据拉取，持久化，推送，解析，转换，验证, 抽取，给外部提供访问本地数据的API。

参与ETL任务处理与任务进程web可视化

负责ETL数据拉取，持久化，给外部提供访问本地数据的API。

1.1负责单点登录模块的，用户信息的管理，用户请求拦截，token的制作与验证，eraka注册中心管理微服务，

自定义注解，拦截请求token的验证与生成。

1.2SSL Web可视化工具，包含服务端证书制作，客户端证书导入与更新至keystore，证书信息管理，定期证书有效性验证的与预警。

1.3.作为客户端以restful形式拉取外部数据到本地：每日有百万级别数据从外部拉取，验证，转换，并持久化到数据库。为防止单个请求体过大，或者返回是数据量过大影响运行内存，以及改善任务耗时问题，采用线程池管理多个任务，多线程发送请求并验证返回结果，并放入阻塞队列，多个线程从队列拿数据并转换，最后持久化到数据库。

将生产环境JVM运行时log和堆栈镜像输出到文件，分析JVM运行状况，调整JVM参数。

实现多个配置文件松耦合,代码层面定制备案的加载顺序，灵活运用benfactorypostprocessor, beanpostprocessor,beanfacory。

定制化线程池，在定义批量数据处理接口，在多线程任务中执行批量数据接口，接口的实现包括批量请求，批量数据转换，批量数据验证，批量数据解析等。

此模块为公共模块，为提高代码可扩展性，广泛采用泛型（定义泛型的接口和类，定义函数的参数，采用最顶层的接口引用和泛型类型，提高可扩展性），使用设计模式包括，工厂模式，动态代理模式，单例模式（线程池），生产者消费者模式（由多个生产者和消费者线程操作的缓存队列，多线程采用线程池管理，多个线程是否完整执行都会与父类变量进行与操作，父类变量为线程安全变量）。

基于单例模式设计数据库连接池。

1．定义service接口，类型为泛型，接口声明default方法，多个同名方法为重载关系，

2．这些方法被不同runable的实现类A所调用，类型为泛型。

3.1定义缓存队列元素泛型类Q,类型为泛型，包含生产者，指针计算器p，和队列元素为S，其类型为泛型。

3.2定义缓存类B， 将LinkedblockingQueue作为实体变量，类型为Q，定义线程安全队列计数器count.

3.3定义消费者/生产者类coumsumer/producer继承父类A,他们都会操作缓存B。

4.定义业务类继承生产者/消费者，实现service接口,重写service接口中的default方法。

1. 定义抽象类P，类中定义抽象方法A，类中定义实体方法B调用抽象方法A。
2. 子类C继承抽象父类P，重写父类抽象方法B为b。
3. 创建子类C的实现指向父类P的引用，子类调用父类的实体方法A，会去调用方法b。

子类的实现指向父类的应用----多态

子类的实现调用父类的同名方法时，若子类重写了父类的方法，则会去调用子类的方法。若子类没有该方法，回去调用父类的方法。

服务端返回的Token是1小时有效，保证每次每次token有效，将token放入valiate 变量中，每45分钟刷新一次，多个https请求共享。对于返回无效token的请求，catch后重新那最新token进行请求。

10000多个tog，需要通过https post请求取数据，单个tag返回数据大，有内存溢出和频繁GC的风险，分批次请求数据，放入有界阻塞队列，队列满时，等待，设计了10个线程，每个线程单次请求10个tag，每个线程循环100次，用thredLocal记录各个线程处理了多少个tag，继承linkListQue，添加valiate 变量，记录有多少个数据队列，消费端记录取出多少个数据。

三者相等。

1.4.作为服务端应对外部以restful形式访问本地数据：包含双向验证，用户单点登录，数据授权。数据表多为数据库中先前已有的表，这些表的数据现在有了新的用途。

设计用户表，角色表，权限表等。

并优化JVM tomact应对客户端高并发请求，单次请求体和单次请求数据量大等问题。

基于AOP，当用户访问特定的函数时，决定是否授权，解决用户数据授权问题，

基于AOP, 并实现Methordinterpretor 类，当某些方法被访问时，将其结果缓存进redis。

自定义注解，运用反射实现拦截特定请求，服务端保存token至redis，自定义加密算法。用户seesionid和cookie的，

在应对数据库访问压力层面，采用redis作为mybatise的二级缓存，再用druid作为数据库连接池。

(理解session会话

客户端存放的cookie包含session id和服务端session，

数据库session和JDBC,druid数据库连接池，数据库优化

)

1.5.为改善大文件传输，大量数据高效验证与转换，多形式持久化和多样化数据源带来的挑战，设计基于NIO Reactor模式的数据处理服务器，提供给客户端脚本和jar包（未来将约定数据传输格式，并向客户端提API），服务端与客户端采用socket套接字进行数据传输，传输过程web可视化。用户导入我们提供的maven坐标，启动脚本，向脚本中传参可向服务段发送请求。当有外部新数据需要上传到本地，客户端无需变动，或者服务端只需写相应的数据质量验证等服务。客户端的数据表现形式当前只支持为文件，服务端数据表现形式为磁盘文件，kafka topic，HDFS，数据库表。

1.4.1.外部从server拉取文件：在指定时间段，启动脚本并传参，客户端进程开始监听传参定目录文件（该文件会有其他业务团队写入请求），读取文件并发给服务端做验证，验证通过后，sever从磁盘，kafka topic，HDFS，数据库表中取数据并发送给客户端。  
1.4.2. 外部向server推送文件：启动特定主机客户端，监听特定目录，当目录内文件满足要求时，服务端告诉客户端该调用服务端什么样的服务(rpc)去做数据质量验证，服务端对数据验证通过后传输给server，server端将数据持久化到磁盘，kafka topic，HDFS，数据库表。

. 1.6 ELK 日志分析，数据源包括kafka,redis和一些服务。

3.熟悉maven和git进行项目管理，

1.多模块三级复合maven项目构建，maven仓库和私服配置，依赖冲突解决，制定profile，依赖和插件管理、升级，使用maven-resource,maven-assambely,maven-dependncy-copy的定制化打包等。

2.Git冲突的解决，基于master创建release/dev分支。

基于master创建自己是work分支

基于work分支创建test分支，

修改work分支，从work分支拉取自己的改动到test分支，从dev分支拉取别人的改动到test分支。

从将test分支merge进dev分支。部署dev分支进行集成测试。

修改work分支，从work分支拉取自己的改动到test分支，从dev分支拉取别人的改动到test分支。从将test分支merge进dev分支。部署dev分支进行集成测试

将work分支merge进release分支。

1. dev环境下，熟悉linux在线项目调试，shell/java代码编辑，打包，热部署。
2. 理解多态，泛型，强制类型转换，提高代码高可用和可扩展性
3. 理解java基本集合，JUC, IO的底层实现原理，并依据需要加以运用。
4. 理解反射，注解，动态代理，并熟悉基本运用。

熟悉单例模式，数据库配置

工厂模式，

生产者消费者模式，消息队列

代理模式，动态代理对不可修改的类进行增强，

面向切面编程，将热点访问数据缓存至redis, 控制不同用户访问表的权限。

3.熟悉jvm运行时内存，能解释锁和和多线程安全的原理，依据需求解决项目中的并发问题。

根据需要设置jvm参数，输出jvm运行时日志，内存映像的持节化，善于分析jvm日志，并对参数适当调整。

4.理解spring bean生命周期，IOC和AOP的运用，了解spring中常用的设计模式。

5.mybatise源码的理解与sql的基本优化。

6.熟悉spring mvc ,redis，,HDFS,springcloud，kafak和ELK, storm基本使用和原理。

topic是逻辑概念，一种topic对应多个partition，即为不同消息，partition在物理上，对应一个文件夹，下面有多个segment ，对应文件。  
一个消费者组，是由多个具有相同group id 的实例组成，可以是线程或者进程，他们共同消费同种topic。一个partition可被多个消费者组消费，但是最多被同一个消费者组下的一个消费者消费。消费者组下的一个消费者有可能消费同一种topic下的多个partition，或者多种topic下的，不同partition。  
1.理想状态下，topic下partition数量和消费者组下实例个数相同，那么一个消费者刚好被分配消费一个partition上的数据。  
2一种.topic的下的 partition多余消费者数量，那么每一个消费者至少被分配到一个partition来消费。有的消费者可能会被分配到多个partition用于消费。  
3.一种topic下partition少于，消费者组消费者数量时，一个消费者最多被分配一个partition可消费，有的消费者会处于闲置状态，无partition可消费。

，一个topic通常包含多个分区，分布在不同broker上，比如一个topic包含两个两个partition，每个broker上都包含两个partition。  
partition即为分区，可分为主分区和副本。消息者组每次都是消费主分区。

partition数量与副本，ack，broker数量，偏移量，index

基于Kafka storm redis ELK从应用开发、维护与优化，确保数据不丢失，架构高可用。

1.工具开发

基于微服务的web工具，模拟上游发UAT/prod消息的工具。通过restful API 从上游拉取数据，生产到Kafka指定topic或者storm，用于prod环境部署后的验证或者UAT测试。

UAT和DEV部署java 版的webSocket客户端，向webSocket服务端推送不同服务状态信息，广播给所有JS版的webSocket客户端。

2.webSocket接收上游数据，更新至redis,kafka,数据库。

1）对延时要求较高的数据流向storm，期中一部分经过storm处理后，更新至redis。

2）被多个服务消费或对历史数据有重复消费需求的数据流向kafka。

3. 为下游提供服务

1）对于A类客户端，客户向我们提供配置星系，提供实时缓存API，API缓存采用ecach以及大对象,服务端缓存采用redis,采用通过websocket实现服务端缓存与本地缓存的近实时通知。

2）对于B类客户提供web可视化界面，Ecach作为一级缓存，redis作为二级缓存，webSock向web客户端推送数据更新。

3) 对于C类客户，redis作为缓存，提供restful访问的入口给客户端。证书+token；以及订阅kafka topic的形式消费历史数据。

4.数据分析平台

1.1 Spring AOP+kafka+ Logstash +ES+日志收集系统开发，及维护升级。

在不改已有代码的前提下，加入Spring AOP的代码， 拦截日志，推送给kafka

* 1. Filebeat +kafka+Logstash +ES
  2. kafka+kafkaconnetor+ES
  3. meitricBeat收集中间件指标。

Redis写数据速度被kafka快

对于批量数据，大对象，服务端将数据更新至redis的同时，向kafka发送更新通知，客户监听kafka通知，客户端向redis拉取最新数据至本地缓存。

对于小数据，小对象，服务端将数据更新至redis的同时，向kafka发送数据本身，客户端消费kafka数据。

Producer挂掉的原因

Producer默认发送的数据大于默认的buffer，需要调整

超过Timeout的时间Producer挂掉

代码重构与优化

将业务代码与非业务代码分离，分离后只有通用代码与业务代码。

引入spring,mybatise,storm,redis,kafka.

采用泛型而不用Object，对于通用模块定义变量类型定义为父类，使得代码可扩展。

对于查询较多的采用MAP

对于不重复的数据用set

对于可能重复数据，采用数据，插入频繁的数据采用链表数据，对查询较多的数据采用普通数据。

对于遍历采用，迭代器，而不采用增强for循环和普通for循环。

# 1.Java基础

## JVM

## Linux常用监控命令

Linux top命令下JVM的虚拟内存与物理内存分析

JVM内存泄漏

## 1.1多线程

## 1.2锁

## 1.3java编程

11、hashcode 和 equals

底层实现，什么场景需要重写他们

hashCode和equale函数用来判断对象是否相同,

equals通常用来比较两个对象的内容是否相等，==用来比较两个对象的地址是否相等

14、Object toString 方法常用的地方，为什么要重写该方法

常用在对象模型类

因为假如User是一个用户的对象，如果User.toString();结果是不正常的，因为User对象中可能有多个属性，如年龄，姓名等，这个toString后无法知道具体的是那个属性转换为字符串

##### 集合

13、hashmap put 方法存放的时候怎么判断是否是重复的

先比较key的hashCode，再比较相等或equals的，所以重写hashCode()和equals()方法即可实现添加重复元素。

###### 15、Set 和 List 区别？

Set（集）：集合中的对象不按特定方式排序，并且没有重复对象。它的有些实现类能对集合中的对象按特定方式排序。

List（列表）：集合中的对象按索引位置排序，可以有重复对象，允许按照对象在集合中的索引位置检索对象。

###### 16、ArrayList 和 LinkedList 区别，如果存取相同的数据，ArrayList 和 LinkedList 谁占用空间更大？那个插入数据更快，分别用于什么场景

 ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构

ArrayList 继承AbstractList

LinkedList 继承AbstractSequentialList

ArrayList 采用的是数组形式来保存对象的，这种方式将对象放在连续的位置中，所以最大的缺点就是插入删除时非常麻烦

LinkedList 采用的将对象存放在独立的空间中，而且在每个空间中还保存下一个链接的索引 但是缺点就是查找非常麻烦 要丛第一个索引开始

对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针

对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据，若要从数组中删除或插入某一个对象，需要移动后段的数组元素，从而会重新调整索引顺序,调整索引顺序会消耗一定的时间，相反,LinkedList是使用链表实现的,若要从链表中删除或插入某一个对象,只需要改变前后对象的引用即可

###### TreeSet HashSet LinkedHashSet

TreeSet 对存入对数据有什么要求呢？

TreeSet集合是用来对象元素进行排序的,同样他也可以保证元素的唯一

22、TreeSet 底层源码有看过吗？

TreeSet的底层实现是TreeMap

public TreeSet(Comparator<? super E> comparator) {

    this(new TreeMap<>(comparator));

  }

HashSet 是不是线程安全的？为什么不是线程安全的？

说白了，HashSet就是限制了功能的HashMap，所以了解HashMap的实现原理

###### 25、Concurrenthashmap 是怎么做到线程安全的？

ConcurrentHashMap的大部分操作和HashMap是相同的，例如初始化，扩容和链表向红黑树的转变等。但是，在ConcurrentHashMap中，大量使用了U.compareAndSwapXXX

的方法，这个方法是利用一个CAS算法实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。这个算法的基本思想就是不断地去比较当前内存中的变量值与你指定的

一个变量值是否相等，如果相等，则接受你指定的修改的值，否则拒绝你的操作。因为当前线程中的值已经不是最新的值，你的修改很可能会覆盖掉其他线程修改的结果。这一

点与乐观锁，SVN的思想是比较类似的。

同时，在ConcurrentHashMap中还定义了三个原子操作，用于对指定位置的节点进行操作。这三种原子操作被广泛的使用在ConcurrentHashMap的get和put等方法中，

正是这些原子操作保证了ConcurrentHashMap的线程安全。

在ConcurrentHashMap没有出现以前，jdk使用hashtable来实现线程安全，但是hashtable是将整个hash表锁住，所以效率很低下。

ConcurrentHashMap将数据分别放到多个Segment中，默认16个，每一个Segment中又包含了多个HashEntry列表数组，

对于一个key，需要经过三次hash操作，才能最终定位这个元素的位置，这三次hash分别为：

对于一个key，先进行一次hash操作，得到hash值h1，也即h1 = hash1(key)；

将得到的h1的高几位进行第二次hash，得到hash值h2，也即h2 = hash2(h1高几位)，通过h2能够确定该元素的放在哪个Segment；

将得到的h1进行第三次hash，得到hash值h3，也即h3 = hash3(h1)，通过h3能够确定该元素放置在哪个HashEntry。

每一个Segment都拥有一个锁，当进行写操作时，只需要锁定一个Segment，而其它Segment中的数据是可以访问的。

26、HashTable 你了解过吗？

Hashtable既不支持Null key也不支持Null value。Hashtable的put()方法的注释中有说明

Hashtable是线程安全的，

Hashtable是线程安全的，它的每个方法中都加入了Synchronize方法，效率比较低

Hashtable默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。

Hashtable在计算元素的位置时需要进行一次除法运算，而除法运算是比较耗时的。

##### 多态

##### 接口与抽象类

##### 泛型

Finally

1.当 try 代码块执行完成， finally 代码块就可以执行，哪怕没有发生异常。

2.finally是在catch执行之后，catch中的return之前执行的。

3. finally当中抛出异常或者return, atch中的return或者不捕获或者抛出的异常就会被忽略。

<https://blog.csdn.net/w605283073/article/details/103841999>

Exception的分类

10、Object 类中的方法

1，构造函数

3，wait(),wait(long),wait(long,int),notify(),notifyAll()

4，toString()和getClass,

5，clone()

6，finalize()用于在垃圾回收

## 1.4Java 8新特性

# 2.算法

## 2.1链表

## 2.2树

## 2.3二维数组

## 2.4Java集合

## 2.5设计模式

### 2.5.1静态代理：

一个代理对象，写死了代理啥,在编译器就写好了代理的操作。

代理对象为目标对象执行指定的操作。

小明想买美国的鞋子：通过A代理（专门代购美国鞋子的人），去美国买鞋子。

小红想买韩国的化妆品：自己去美国买

小红想买美国的化妆品：通过B代理（专门代购韩国化妆品人），去韩国买化妆品。

### 2.5.2动态代理：

代理对象工厂：专门生成代理对象的工厂。

代理对象工厂根据目标对象，在"程序运行期间"自动生成代理对象，代理对象为目标对象执行相应的操作。

小红 找代理公司，代理公司为小红分配，专门代购韩国化妆品的人（代理），去韩国买些。

小明 找代理公司，代理公司为小明分配，专门代购美国鞋子的人（代理），去韩国买化妆品。

1.创建目标对象

2.代理对象工厂，并生成代理对象：实现InvocationHandler接口，重写其的invoke方法；调用Proxy类的静态方法newProxyInstance方法。

目标对象管理器（写带参构造函数，用于接受目标对象，一个类来实现InvocationHandler接口的类，重写了它的invoke方法）

生成代理对象：将吸收接受目标对象的handler传入 Proxy类的静态方法newProxyInstance，返回一个接口的代理实例，生成代理对象。

3.代理对象代替目标对象执行相应操作。

# 3框架

## 3.1Spring springboot

##### 反射的机制

大家都知道，要让Java程序能够运行，那么就得让Java类要被Java虚拟机加载。Java类如果不被Java虚拟机加载，是不能正常运行的。现在我们运行的所有的程序都是在编译期的时候就已经知道了你所需要的那个类的已经被加载了。

Java的反射机制是在编译并不确定是哪个类被加载了，而是在程序运行的时候才加载、探知、自审。使用在编译期并不知道的类。这样的特点就是反射

反射机制通过void setAccessible(boolean flag)方法可以得到一个类的private的方法和属性，使用这些private的方法和属性

## springMVC

## 3.2Mybastise

1).一条sq|执行流程: 连接，缓存，解析器，预处理器，优化器，执行计划，执行引擎。优化我们考虑每个阶段的处理方案。从连接池，到redis，到sq|语句，索引正确建立，读写分离，分库分表。再到业务优化，分流，等等

2).建立索性和图，主键，select update delete语句优化，避免使用select\*，尽量使用索性字段作为查询字段，join代替嵌套查询，where.代理having，

如果不是主键也会底层给你一-个主键索引每次走其他索引的时候也是找到主键索引通过主键索引去查询的.。

使用数据库连接池，数据库连接可被反复使用，提高效率。

少用or。多用union。少用in。尽量不要嵌套太多层左右链接。因为每-一个链接都要遍历一边数据。层次越多遍历越慢。还有-些其它的

最主要的还是查询，更新删除操作导致死锁和性能消耗的情况较少，查询是最提现程序员数据库操作能力的，最主要就是防止索引失效，防止索性失效最主要的就是防止临时表的自动创建。  
这些，explain 然后慢慢优化SQL语句。.

数据页分裂，索引页分裂这些会问到吗?就是数据从少到多，进行数据页分裂再到聚簇索引的出现，再到索引的数据结构这一块，会问这么深吗?

Mysq|索引分为两种，主键索引最快，因为他没有回表操作，当你已普通索引进行查找时会进行一次回表，从头开始再查一遍主建. 索引然后获取到数据，主建索引在没有的情况下Mysql会自动帮你创建，索引优化最好查询的时候能少回表就少  
  
因为B-Three在数据量大的  
  
情况下容易变成单向链表所以Mysq|采用了B+Three结构是跟节点页节点数据节点，数据节点上下关联行程双向链表结构，采用二分算法进行查询数据

3).Mybatis有二级缓存，为什么还要用redis? ? ?

1. Mybatis - -级缓存作用域是session,  
  
session commit之后缓存就失效了。  
  
2. Mybatis二级缓存作用域是sessionfactory,该缓存是以namespace为单位的( 也就是一个Mapper.xml文件)，不同namespace下的操作互不影响。  
  
3.所有对数据表的改变操作都会刷新缓存。但是一般不要用二级缓存，例如在  
  
UserMapper.xml中有大多数针对user表的操作。但是在另-一个XXXMapper.xml中，还有针对user单表的操作。这会导致user在两个命名空间下的数据不-致。  
  
4.如果在UserMapper.xml中做了刷新缓存的操作，在XXXMapper.xml中缓存仍然有效，如果有针对user的单表查询，使用缓存的结果可能会不正确，读到脏数据。  
  
5. Redis很好的解决了这个问题，而且比之一、二级缓存的好处很多，Redis 可以搭建在其他服务器\_上，缓存容量可扩展。Redis 可以灵活的使用在需要缓存的数据.上，比如一些热，点数据，统计点赞啊,中央缓存

## 3.3springcloud

# 4缓存与持久化

## IO

1. 内核为每个进程分配想通虚拟内存地址空间，如何做到内核数据共享，而进程各自的用户态数据互补干扰，如何进行地址映射的？
2. 我们都知道，程序最终是要运行在物理内存之中的，所以虽然每个进程的虚拟地址空间是一样的，但是不同的页表将会映射到不同的物理地址。
3. mysql和kafak基于硬盘存储，有磁盘io性能阻塞要求，而elisticsearch 和redis基于缓存它是没由磁盘IO阻塞，但有网卡与内核态数据与用户态数据交互的阻塞。  
   内核太空间与用户态空间，地址映射，IO设备之间的关系。channel如何完成数据流动的。

**通道（ Channel）：**由 java.nio.channels 包定义的。 Channel 表示 IO 源与目标打开的连接。Channel 类似于传统的“流”。只不过 Channel本身不能直接访问数据， Channel 只能与Buffer 进行交互。

从通道到buffer，是一种0拷贝，也就是数据不经历由磁盘到内核态，再由内核态到用户态，直接用DMA处理器负责将数据读入buffer，  
所以通道数据到buffer为读入，buffer数据到通道为写出。

与kafka 0拷贝技术分析  
解释redis读写为何没有io阻塞，可以用单线程读写，而网络数据传输，是有io要求的，在6.0之后用多线程。

进程是资源分配的基本单位，linux中一切皆为文件。

vfs虚拟文件系统，管理硬件设备资源，

vm虚拟内存技术，进程虚拟内存地址（内核态和用户态空间）到物理地址空间的映射mmp关系。

内核：虚拟文件系统VFS（）和VM虚拟内存技术，内存映射mmap （Memory Mapping ,缩写 mmap）以及直接内存地址DMA（Direct Memory Access）

1.虚拟文件系统将外部设备（网卡，磁盘，物理内存）无差异化地抽象为文件，linux中一切皆为文件，  
2.内核负责为进程分配虚拟内存，和物理内存，  
3.内核与外部设备进行数据交换，内核态数据可被用户态访问。所以进程等待Io资源，可写，可读状态耗时，就有了阻塞。

内核空间：Linux自身使用的空间；主要提供进程调度、内存分配、连接硬件资源等功能程序逻辑使用。

用户空间：提供给各个程序进程的空间；用户空间不具有访问内核空间资源的权限，如果应用程序需要使用到内核空间的资源，则需要通过系统调用来完成：从用户空间切换到内核空间，完成相关操作后再从内核空间切换回用户空间。

内核通过虚拟文件系统管理磁盘，一般用户进程无法直接访问IO资源，需要通过系统调用才能访问。

上层程序的IO操作、不是物理设备级别的读写，而是缓存的复制。而内核缓冲区和物理设备之间的数据交换则是由操作系统的Kernel来完成。

网卡(接收到data） --- > 内核 ---> 应用程序（读取）  
网卡电流高低信号被转换为二进制数据，再被linux内核识别。

链路层  
网络层  
传输层  
都是受内核管理的。

IO 可以简单分为磁盘 IO 和 网络 IO ,磁盘 IO 相对于网络 IO 速度会快一点。

JAVA 对 NIO （nio new Io 非阻塞io）抽象为 Channel , Channel 又可以分为 FileChannel （磁盘 io）和 SocketChannel （网络 io）

磁盘和网卡，缓存都为CPU外围设备，在用户程序读写数据时，缓存充当内核缓冲区（磁盘缓冲区）和socket缓冲区（网卡缓冲区）

### read & write与0拷贝

#### 阻塞IO读写

read: 把数据从内核缓冲区复制到进程缓冲区。

write: 把数据从进程缓冲区复制到内核缓冲区。

#### 非阻塞IO读写

从通道到bufffer

Buffer到通道。

大白话解释，零拷贝就是没有把数据从一个存储区域拷贝到另一个存储区域。但是没有数据的复制，怎么可能实现数据的传输呢？其实我们在java NIO、netty、kafka遇到的零拷贝，并不是不复制数据，而是减少不必要的数据拷贝次数，从而提升代码性能

* 零拷贝的好处

减少或避免不必要的CPU数据拷贝，从而释放CPU去执行其他任务

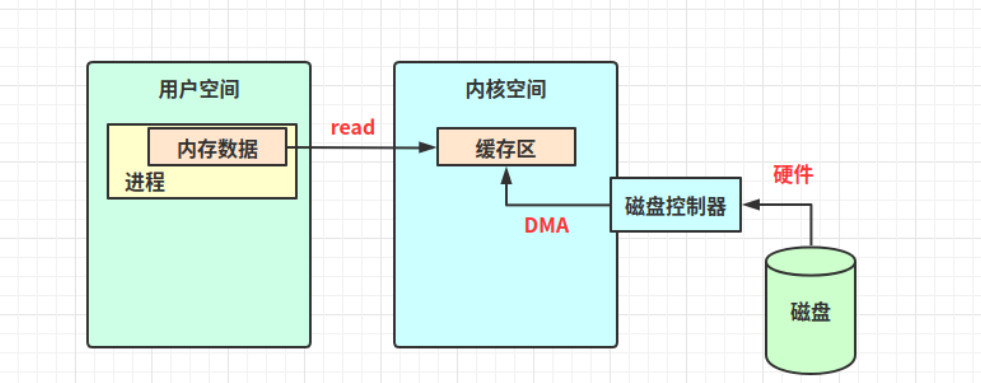
零拷贝机制能减少用户空间和操作系统内核空间的上下文切换

减少内存的占用

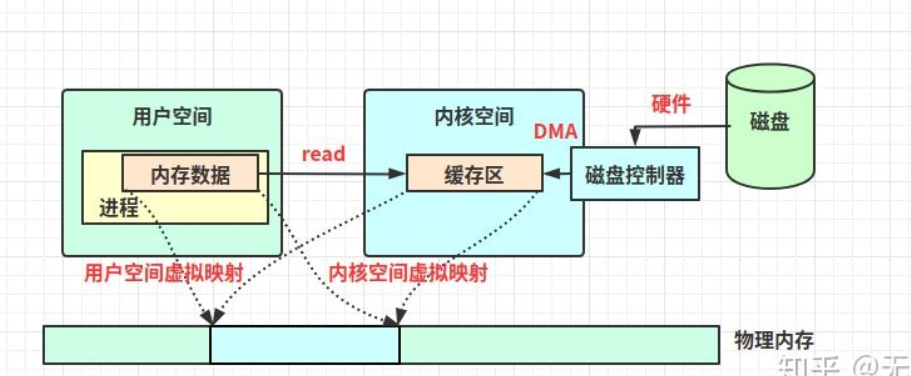
* mmap+write 实现的零拷贝
* sendfile 实现的零拷贝
* 带有DMA收集拷贝功能的sendfile实现的零拷贝
* java提供的零拷贝方式

### 直接内存访问（Direct Memory Access）（DMA）

直接内存访问：DMA允许外设设备和内存存储器之间直接进行IO数据传输，其过程不需要CPU的参与。无论是磁盘数据到内核是磁盘缓冲区还是网卡数据到内核缓冲区都是DMA完成数据拷贝的。



* 进程发起read请求，内核先检查内核空间缓冲区是否存在进程所需数据，如果已经存在，则直接copy数据到进程的内存区。如果没有，系统则向磁盘请求数据，通过DMA写入内核的read缓冲冲区，接着再将内核缓冲区数据copy到进程的内存区
* 进程发起write请求，则是把进程的内存区数据copy到内核的write缓冲区，然后再通过DMA把内核缓冲区数据刷回磁盘或者网卡中。
* 虚拟内存：现代操作系统都使用虚拟内存，有如下两个好处
  + 一个以上的虚拟地址可以指向同一个物理内存地址
  + 虚拟内存空间可大于实际可用的物理地址
* 利用第一点特性可以把内核空间地址和用户空间的虚拟地址映射到同一个物理地址，这样DMA就可以填充(读写)对内核和用户空间进程同时可见的缓冲区了；大致如下



，

一个进程启动后，内核会为起分配一个虚拟内存空间，看似连续，实际可能是不连续，用户进程申请的内存只有实际用时，内核才会为它分配物理空间。

我们都知道，在linux的内存管理机制中，采用了虚拟内存管理机制。linux内核的虚拟地址空间共4G,分为两部分内核空间（3G~4G）以及用户空间（0~3G）。内核空间对于每个进程而言都是可见的，而用户空间对于每个用户而言是独占的，其他用户不可见。  
内核空间虚拟地址与用户虚拟空间映射到同一块物理地址时，就回NIO。  
kafka用到的0拷贝技术，是网卡与内核读buffer进行交换数据后，直接拷贝到内核写buffer，不经过用户态。

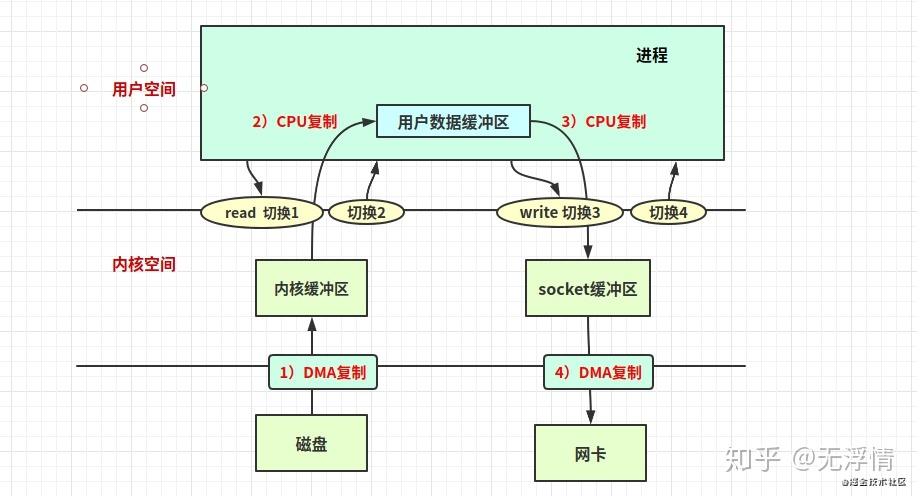
### 传统的 I/O ,发送本地数据到远端模型

基于传统IO, java在linux系统上，读取一个磁盘文件，并发送到远程端的服务。

#include <unistd>

ssize\_t write(int filedes, void \*buf, size\_t nbytes);

ssize\_t read(int filedes, void \*buf, size\_t nbytes);



图中的内核缓存其实可以理解为磁盘缓存区，与网卡缓冲区对应。

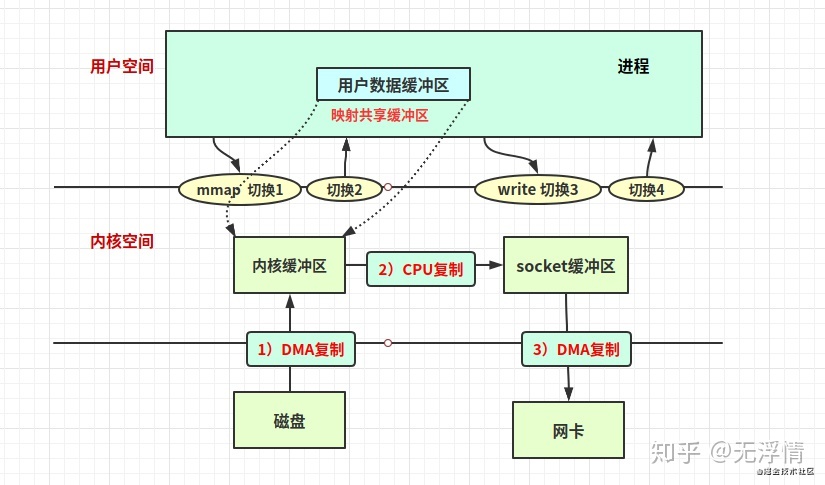
* 1）发出read系统调用（所谓是系统调用都是调用linux内核函数），会导致用户空间到内核空间的上下文切换，然后再通过DMA将文件中的数据从磁盘上读取到内核空间缓冲区
* 2）接着将内核空间缓冲区的数据拷贝到用户空间进程内存，然后read系统调用返回。而系统调用的返回又会导致一次内核空间到用户空间的上下文切换
* 3）write系统调用，则再次导致用户空间到内核空间的上下文切换，将用户空间的进程里的内存数据复制到内核空间的socket缓冲区（也是内核缓冲区，不过是给socket使用的），然后write系统调用返回，再次触发上下文切换
* 4）至于socket缓冲区到网卡的数据传输则是独立异步的过程，也就是说write系统调用的返回并不保证数据被传输到网卡

**「一共有四次用户空间与内核空间的上下文切换。四次数据copy，分别是两次CPU数据复制，两次DMA数据复制」**

### mmap+write实现的零拷贝

#include <sys/mman.h>

void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset)



此类情况：用户数据缓冲区和内核数据缓冲区的虚拟地址指向同一物理内存地址。

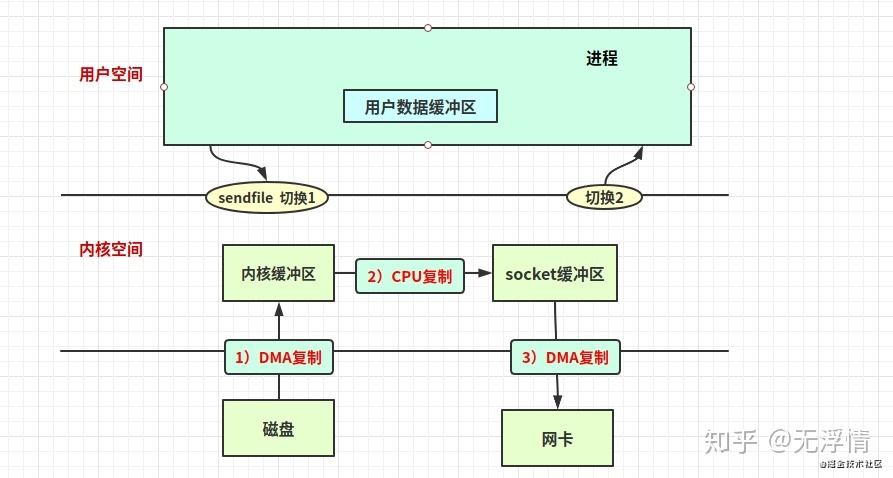
* 1）发出mmap系统调用，导致用户空间到内核空间的上下文切换。然后通过DMA引擎将磁盘文件中的数据复制到内核空间缓冲区
* 2）mmap系统调用返回，导致内核空间到用户空间的上下文切换
* 3）这里不需要将数据从内核空间复制到用户空间，因为用户空间和内核空间共享了这个缓冲区
* 4）发出write系统调用，导致用户空间到内核空间的上下文切换。将数据从内核空间缓冲区复制到内核空间socket缓冲区；write系统调用返回，导致内核空间到用户空间的上下文切换
* 5）异步，DMA引擎将socket缓冲区中的数据copy到网卡

**「通过mmap实现的零拷贝I/O进行了4次用户空间与内核空间的上下文切换，以及3次数据拷贝；其中3次数据拷贝中包括了2次DMA拷贝和1次CPU拷贝」**

### sendfile实现的零拷贝

#include <sys/sendfile.h>

ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count);

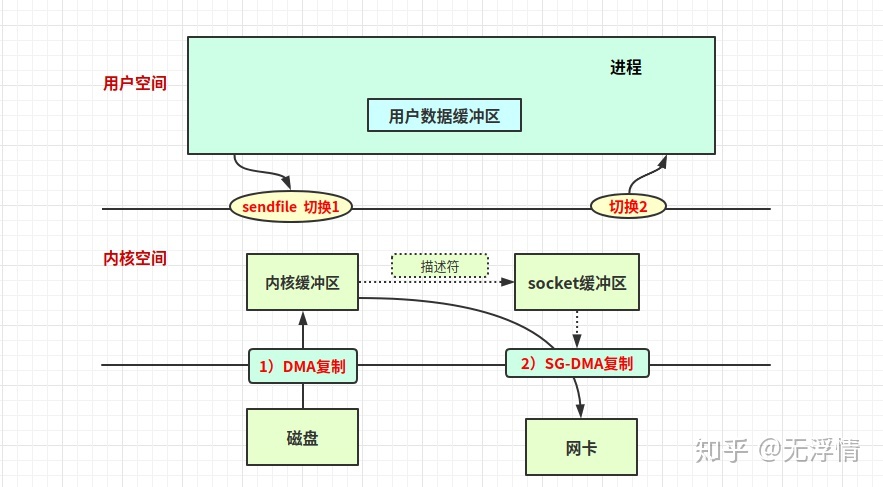


* 1）发出sendfile系统调用，导致用户空间到内核空间的上下文切换，然后通过DMA引擎将磁盘文件中的内容复制到内核空间缓冲区中，接着再将数据从内核空间缓冲区复制到socket相关的缓冲区
* 2）sendfile系统调用返回，导致内核空间到用户空间的上下文切换。DMA异步将内核空间socket缓冲区中的数据传递到网卡

**「通过sendfile实现的零拷贝I/O使用了2次用户空间与内核空间的上下文切换，以及3次数据的拷贝。其中3次数据拷贝中包括了2次DMA拷贝和1次CPU拷贝」**

### 带有DMA收集拷贝功能的sendfile实现的零拷贝

* 从Linux 2.4版本开始，操作系统提供scatter和gather的SG-DMA方式，直接从内核空间缓冲区中将数据读取到网卡，无需将内核空间缓冲区的数据再复制一份到socket缓冲区



* 1）发出sendfile系统调用，导致用户空间到内核空间的上下文切换。通过DMA引擎将磁盘文件中的内容复制到内核空间缓冲区
* 2）这里没把数据复制到socket缓冲区；取而代之的是，相应的描述符信息被复制到socket缓冲区。该描述符包含了两种的信息：A)内核缓冲区的内存地址、B)内核缓冲区的偏移量
* 3）sendfile系统调用返回，导致内核空间到用户空间的上下文切换。DMA根据socket缓冲区的描述符提供的地址和偏移量直接将内核缓冲区中的数据复制到网卡

**「带有DMA收集拷贝功能的sendfile实现的I/O使用了2次用户空间与内核空间的上下文切换，以及2次数据的拷贝，而且这2次的数据拷贝都是非CPU拷贝。这样一来我们就实现了最理想的零拷贝I/O传输了，不需要任何一次的CPU拷贝，以及最少的上下文切换」**

### java提供的零拷贝方式

* java NIO的零拷贝实现是基于mmap+write方式
* FileChannel的map方法产生的MappedByteBuffer FileChannel提供了map()方法，该方法可以在一个打开的文件和MappedByteBuffer之间建立一个虚拟内存映射，MappedByteBuffer继承于ByteBuffer；该缓冲器的内存是一个文件的内存映射区域。map方法底层是通过mmap实现的，因此将文件内存从磁盘读取到内核缓冲区后，用户空间和内核空间共享该缓冲区。用法如下
* public void main(String[] args){
* try {
* FileChannel readChannel = FileChannel.open(Paths.get("./cscw.txt"), StandardOpenOption.READ);
* FileChannel writeChannel = FileChannel.open(Paths.get("./siting.txt"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);
* MappedByteBuffer data = readChannel.map(FileChannel.MapMode.READ\_ONLY, 0, 1024 \* 1024 \* 40);
* //数据传输
* writeChannel.write(data);
* readChannel.close();
* writeChannel.close();
* }catch (Exception e){
* System.out.println(e.getMessage());
* }
* }
* FileChannel的transferTo、transferFrom 如果操作系统底层支持的话，transferTo、transferFrom也会使用相关的零拷贝技术来实现数据的传输。用法如下

public void main(String[] args) {

try {

FileChannel readChannel = FileChannel.open(Paths.get("./cscw.txt"), StandardOpenOption.READ);

FileChannel writeChannel = FileChannel.open(Paths.get("./siting.txt"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);

long len = readChannel.size();

long position = readChannel.position();

//数据传输

readChannel.transferTo(position, len, writeChannel);

//效果和transferTo 一样的

//writeChannel.transferFrom(readChannel, position, len, );

readChannel.close();

writeChannel.close();

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

### Swap机制，

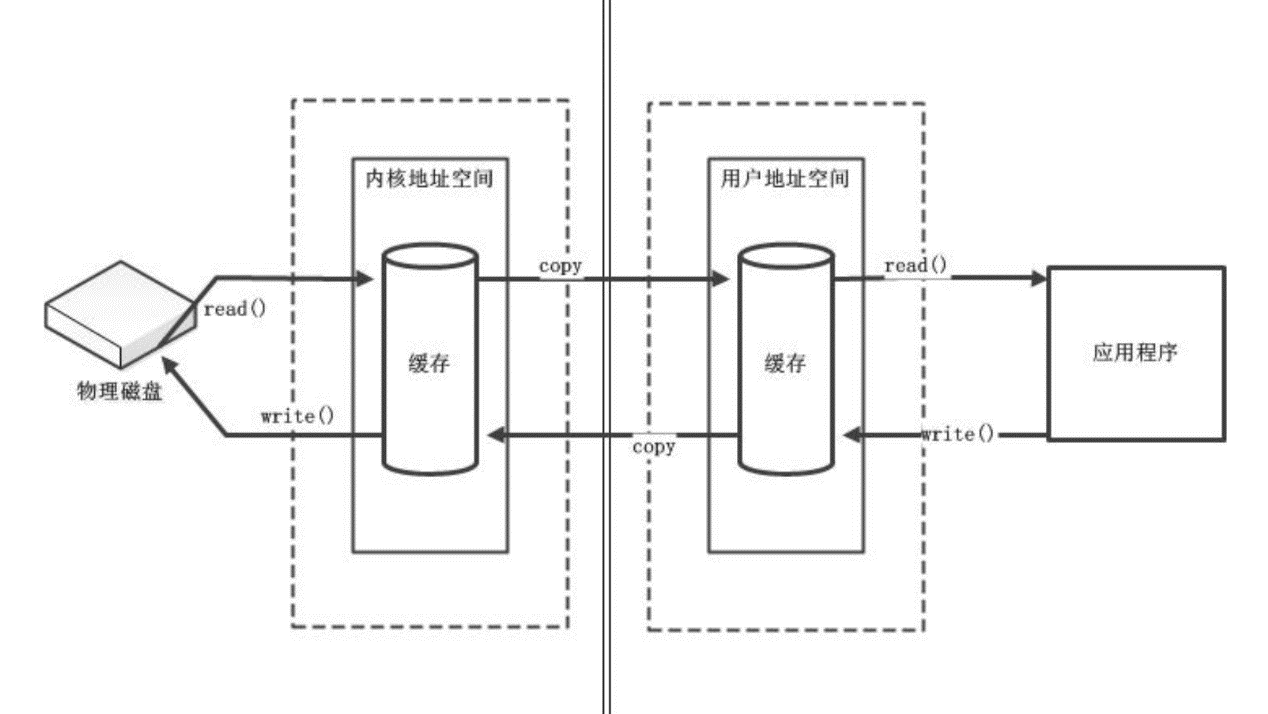
当内存不够用的时候，就会触发 swap,会根据 LRU 将最近使用频率比较低的虚拟页对应物理也淘汰掉，写入到磁盘中去，淘汰掉一部分物理内存中的数据，然后对对应的虚拟页设置为 0，然后将磁盘上的数据再加载到内存中去。

就是把一块磁盘或者一个本地文件，当作内存使用，它包括啷个过程：  
换出，把进程暂时不用的内存数据保存到磁盘中，在释放内存给其他进程使用。  
换入，当进程再次访问内存时，从磁盘读取数据到内存中。

之所以有阻塞是因为，进程想要访问IO资源必须通过内核，对于单核cpu来说，在某一个时刻，内核态的缓存只能被一个进程所访问，内核态数据是所有进程公用的，进程自己的数据确实私有的。

Linux的的核心是linux内核，linux

1. https://www.cnblogs.com/dengchj/p/5410421.html
2. <https://www.jianshu.com/p/486b0965c296>
3. IO流
4. socket网络流
5. 字符流和字节流
6. 在linux文件Io和网络Io基础上，理解nysql的B+树索引结构，以及Kafkay页缓存与0拷贝，redis单线程IO读写多路复用，与6.0后网络io多线程  
   在TCP/IP基础上理解http与websocket。
7. IO读写  
     
   2、缓冲区  
     
         缓冲区的目的：减少频繁地与设备之间的物理交换。  
     
         Linux系统中，操作系统只有一个内核缓冲区，而每个用户，有自己的独立缓冲区，叫进程缓冲区。  
     
   3、四种IO模型  
     
         同步阻塞IO（Blocking IO）：  
     
              【优点】阻塞等待数据期间，用户线程挂起，不会占用CPU资源。  
     
              【缺点】为每个连接配置一个线程，在并发量大的情况下，内存、线程切换的开销会非常巨大。  
     
              【高并发场景下不可用】 【Socket默认模式】  
     
         同步非阻塞IO（Non-blocking IO）  
     
              【优点】 应用程序不断地进行IO系统调用，轮询数据是否准备好，在内核等待数据地过程中立即返回，在内核缓冲区有数据时才开始阻塞，实时性较好。  
     
              【缺点】 不断地轮询，占用大量CPU时间，效率低下。  
     
              【高并发场景下不可用】  
     
         IO多路复用（IO Multiplexing）： 避免轮询等待问题。  
     
              【特点】 涉及两种系统调用，一种select/epoll，另一种是IO操作。select先查询注册的socket连接对应的文件描述符（轮询），若可读则用户线程阻塞read读取。  
     
              【优点】 与一个线程维护一个连接的阻塞IO模型相比，select/epoll可以一个选择器同时处理成千上万个连接。不必创建大量的线程，减少系统开销。  
     
              【缺点】 select/epoll是阻塞式的，整个读写过程是阻塞的。  
     
              【New IO技术应用】【Netty】  
     
         异步IO（Asynchronous IO）：彻底解除线程的阻塞。  
     
              【特点】 在内核等待和复制数据阶段都不需要阻塞，用户只注册一个IO操作完成的回调函数。  
     
              【缺点】 应用程序仅需要事件注册和接收，其余工作留给操作系统。
8. JVM虚拟机和linux之间的存地址映射，以及多线程IO是如何完成的。
9. 通道（ Channel）：由 java.nio.channels 包定义的。 Channel 表示 IO 源与目标打开的连接。Channel 类似于传统的“流”。只不过 Channel本身不能直接访问数据， Channel 只能与Buffer 进行交互。
10. JVM以一个进程（Process）的身份运行在Linux系统上，了解Linux与进程的内存关系，是理解JVM与Linux内存的关系的基础。



当应用程序读文件的时候，数据需要从先从磁盘读取到内核空间(第一次读写，没有 page cache 缓存数据)，在从内核空间 copy 到用户空间，这样应用程序才能使用读到的数据。当一个文件的全部数据都在内核的 Page Cache 上时，就不用再从磁盘读了，直接从内核空间 copy 到用户空间去了。

### Pagecache

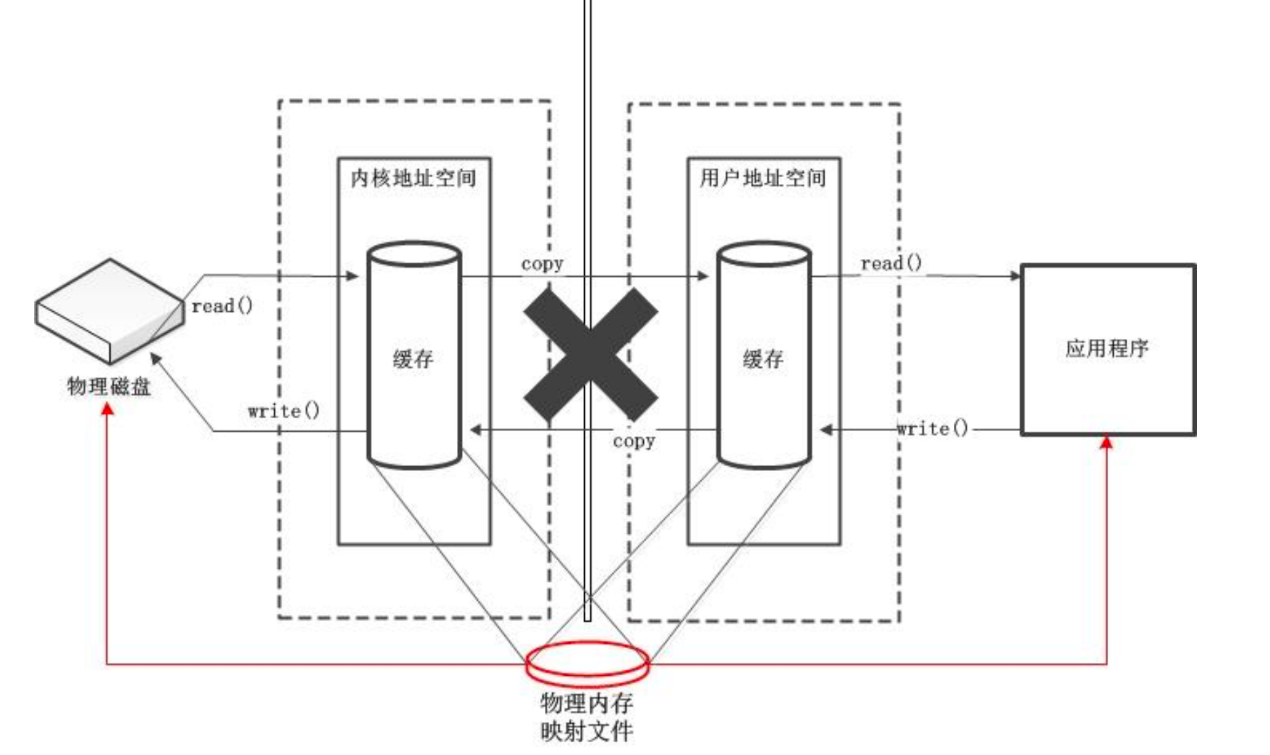
<https://blog.csdn.net/Wanguyunxiaodaniu/article/details/107302344>

应用程序对一个文件写数据时，先将要写的数据 copy 到内核 的 page cache，然后调用 fsync 将数据从内核落盘到文件上（只要调用返回成功，数据就不会丢失）。或者不调用 fsync 落盘，应用程序的数据只要写入到 内核的 pagecache 上，写入操作就算完成了，数据的落盘交由 内核 的 Io 调度程序在适当的时机来落盘（突然断电会丢数据，MySQL 这样的程序都是自己维护数据的落盘的）。

### Memory Mapping

我们可以看到数据的读写总会经过从用户空间与内核空间的 copy ,如果能把这个 copy 去掉，效率就会高很多，这就是 mmap （内存映射）。将用户空间和内核空间的内存指向同一块物理内存。内存映射 英文为 Memory Mapping ,缩写 mmap。对应系统调用 mmap

这样在用户空间读写数据，实际操作的也是内核空间的，减少了数据的 copy 。

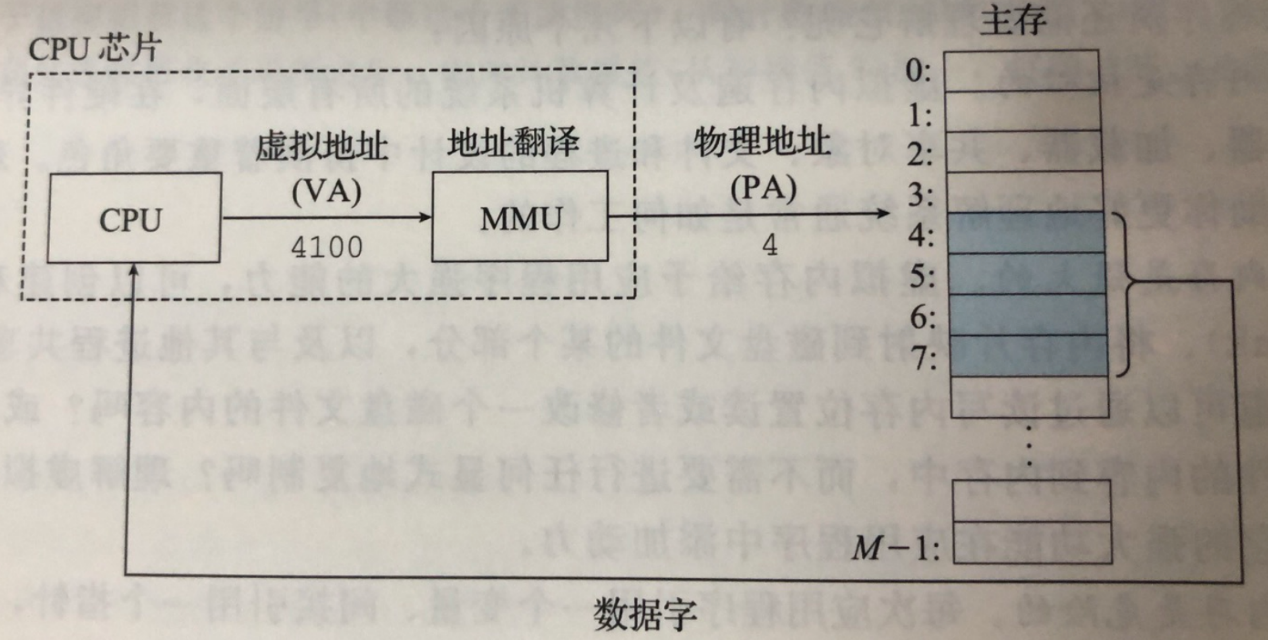


怎么实现的呢，简单来说就是 linux 中进程的地址是虚拟地址，cpu 会将虚拟地址映射到物理内存的物理地址上。mmap 实际是将用户进程的某块虚拟地址与内核空间的某块虚拟地址映射到同一块物理内存上，已达到减少数据的 copy 。

用户程序调用系统调用 mmap 之后的数据的读写都不需要调用系统调用 read 和 write 了。

计算机的主存可以看做是由 M 个连续字节组成的数组，每个字节都有一个唯一物理地址 (Physical Address)。

### Cpu 使用的 虚拟寻址 （VA,Virtual Address）来查找物理地址。



CPU 会将进程使用的 虚拟地址 通过 CPU 上的硬件 内存管理单元 (Memory Management Unit MMU) 的进行地址翻译找到物理主存中的物理地址，从而获取数据。

当进程加载之后，系统会为进程分配一个虚拟地址空间，当虚拟地址空间中的某个 虚拟地址 被使用的时候，就会将其先映射到主存上的 物理地址。

上图是cpu通过虚拟地址读取4个字节示意图。cpu在加载虚拟地址（VA）之前，MMU（地址管理单元）会将VA转换成PA（物理地址），然后在执行正常的读取操作。

### 虚拟内存VM（管理内存的工具）

<https://blog.csdn.net/bdss58/article/details/84307036>

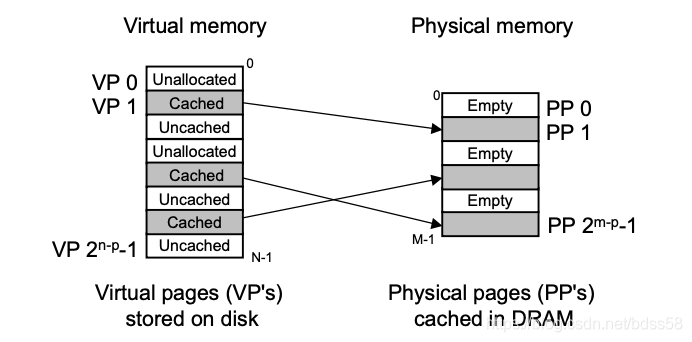
在现代操作系统中，进程之间共享使用cpu和内存，但是内存资源有限，为了更加高效地使用内存，现代操作系统提供一个内存抽象—虚拟内存。虚拟内存巧妙地利用内存，地址转换，磁盘文件和操作系统内核来为每一个进程提供足够大的统一的私有地址空间。虚拟内存提供三个重要的能力：1）将内存当作磁盘的缓存，在内存中只保留常用数据，必要时从内存和磁盘之间交换数据。2）简化内存管理，为每个进程提供统一的地址空间。3）保护进程的内存空间不受其他进程影响。虚拟内存是计算机系统里的一个很棒的方案。它的实现对于程序员来说是透明的，但是程序员为什么需要了解它的工作机制呢？1）虚拟内存的位置比较核心，它渗透到计算机系统的方方面面，比如硬件异常设计，汇编器，加载器，共享对象文件，文件和进程。2）虚拟内存功能非常强大，它能给应用程序提供创建销毁一块内存并将其映射到磁盘文件的能力，还能与其他进程共享内存。比如，你能通过读写特定内存地址来改变磁盘文件的内容，还可以在不需要copy操作的情况下，将磁盘文件的内容加载到内存里。

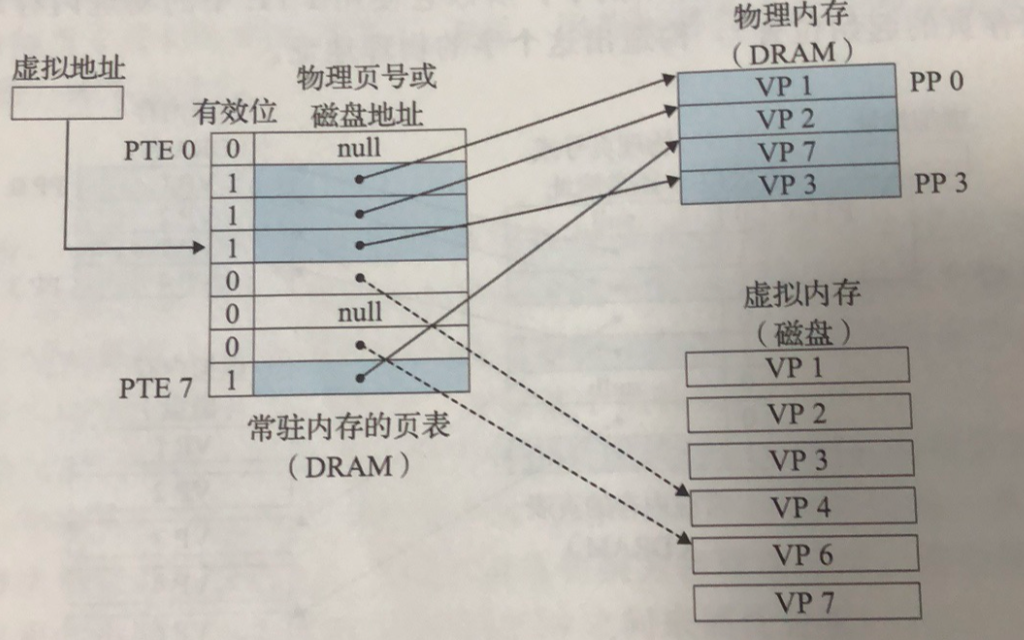
当多个进程需要共享数据的时候，只需要将其虚拟地址空间中的某些虚拟地址映射相同的物理地址即可。

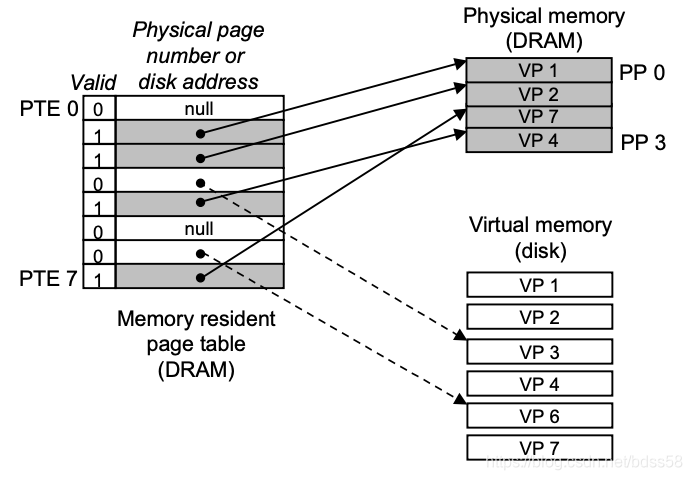
通常我们操作数据的时候，不会一个字节一个字节的操作，这样效率太低，通常都是连续访问某些字节。所以在内存管理的时候，将内存空间分割为页来管理，物理内存中有物理页（Physical Page），虚拟内存中有 Virtual Page 来管理。通常页的大小为 4KB。

系统通过 MMU 和 页表（Page Table） 来管理 虚拟页 和 物理也 的对应关系，页表就是页表条目（Page Table Entry,PTE）的数组。

从概念上来讲，虚拟内存一种存储在磁盘上的N个连续字节的数组。对应与数组中的每个字节都有其唯一的虚拟地址。数组中的部分字节内容缓存在内存中。内存与磁盘之间交换的时候以内存页（VPs）为最小单位，对应地，物理内存在分页，PPs。







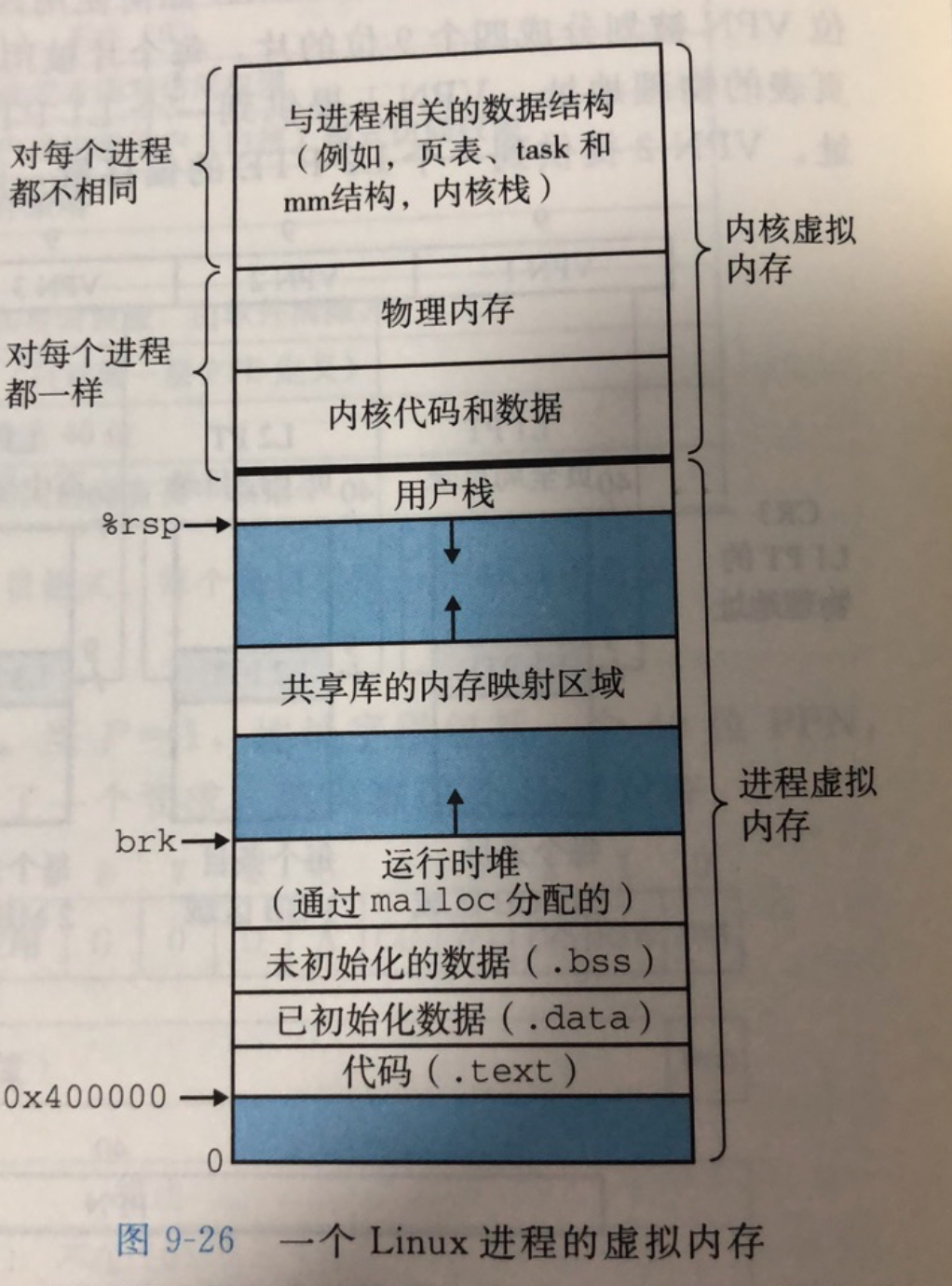
VM system 必须确认某个虚拟页（VP） 是否存储在DRAM中. 如果在，还要确定存储在那个物理内存页中（PP）

PTE 的有效为1时，标识数据在内存中，标识为 0 时，标识在磁盘上。

当访问的虚拟地址对应的数据不再物理内存上时，会有两种情况处理：

1、在内存够用的时候，会直接将虚拟页对应在磁盘上的数据加载到物理内存上，

### 进程的虚拟内存

Linux 会为每个进程分配一个单独的虚拟内存地址，

数据读写速度上 mmap 大于 ByteBuffer.allocateDirect 大于 ByteBuffer.allocate。

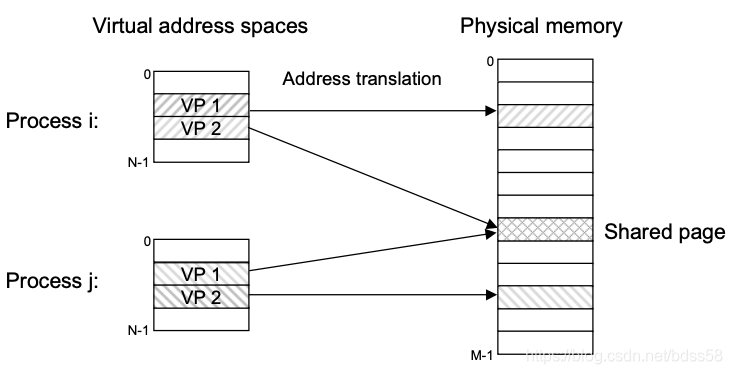
### Linux虚拟文件系统

https://blog.csdn.net/xyz9353/article/details/119304429

### VM作为内存管理工具

上面讲了那么多都是关于VM作为缓存工具的内容，其实有意思的是，在很多计算机系统里虚拟内存比物理内存要小，这样看起来就没必要使用缓存了。但是VM另有大用，它可以作为有效的内存管理工具来保护内存。

上面的内容都是假设有一张页表，将一个虚拟内存空间映射到物理内存空间，其实，操作系统为每个进程提供来单独的页表和单独的虚拟内存空间。请看下面的示意图。



#### 1.1 用户空间与内核空间

现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操作系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间。

#### 1.2 进程切换

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换。因此可以说，任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。

从一个进程的运行转到另一个进程上运行，这个过程中经过下面这些变化：

1. 保存处理机上下文，包括程序计数器和其他寄存器。
2. 更新PCB信息。
3. 把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列。
4. 选择另一个进程执行，并更新其PCB。
5. 更新内存管理的数据结构。
6. 恢复处理机上下文。

注：总而言之就是很耗资源，具体的可以参考这篇文章：[进程切换](https://links.jianshu.com/go?to=http%3A%2F%2Fguojing.me%2Flinux-kernel-architecture%2Fposts%2Fprocess-switch%2F)。

#### 1.3 进程的阻塞

正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。可见，进程的阻塞是进程自身的一种主动行为，也因此只有处于运行态的进程（获得CPU），才可能将其转为阻塞状态。当进程进入阻塞状态，是不占用CPU资源的。

#### 1.4 文件描述符fd

文件描述符（File descriptor）是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念。

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于UNIX、Linux这样的操作系统。

#### 1.5 缓存 IO与page cache

缓存 IO 又被称作标准 IO，大多数文件系统的默认 IO 操作都是缓存 IO。在 Linux 的缓存 IO 机制中，操作系统会将 IO 的数据缓存在文件系统的页缓存（ page cache ）中，也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。

#### **缓存 IO 的缺点：**

数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

##### FileChannel 读写复制文件的用法

##### ByteBuffer 的介绍

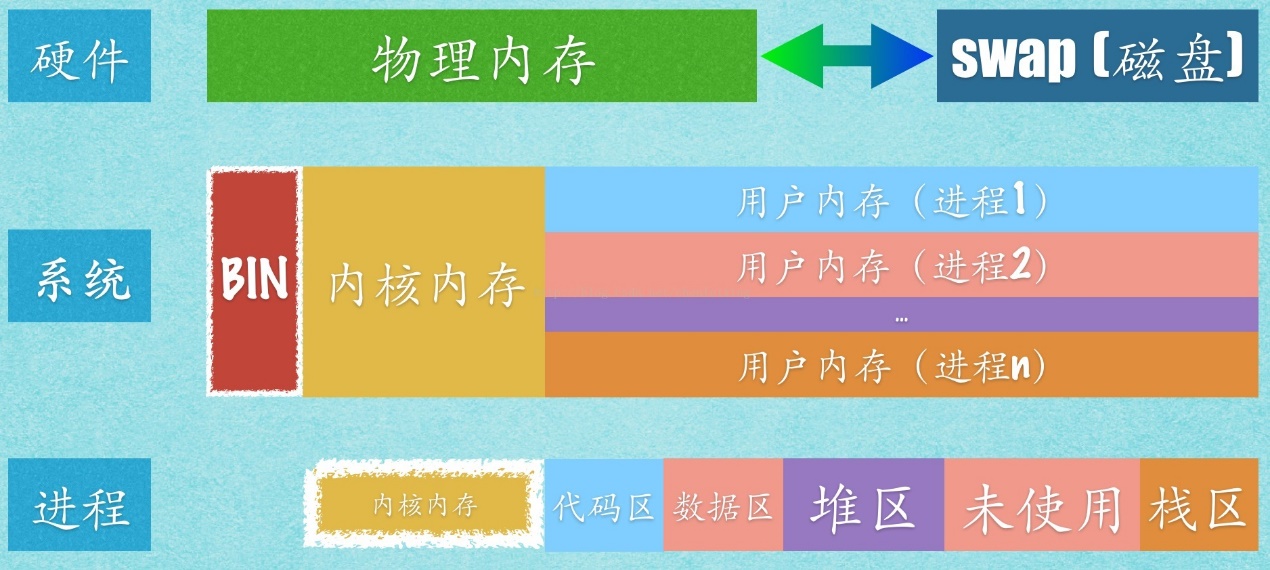
##### jvm 文件进程锁，FileLock

##### HeapByteBuffer ，DirectByteBuffer 和 mmap 谁的速度更快

##### 从 Linux 内核 中的 虚拟内存 、系统调用、文件描述符、Inode、Page Cache 、缺页异常讲述整个 IO 的过程

##### jvm 堆外的 DirectByteBuffer 的内存怎么回收

JVM以一个进程（Process）的身份运行在Linux系统上，了解Linux与进程的内存关系，是理解JVM与Linux内存的关系的基础。  
下图给出了硬件、系统、进程三个层面的内存之间的概要关系。



从硬件上看，Linux系统的内存空间由两个部分构成：物理内存和SWAP（位于磁盘）。物理内存是Linux活动时使用的主要内存区域；当物理内存不够使用时，Linux会把一部分暂时不用的内存数据放到磁盘上的SWAP中去，以便腾出更多的可用内存空间；而当需要使用位于SWAP的数据时，必须先将其换回到内存中。

从Linux系统上看，除了引导系统的BIN区，整个内存空间主要被分成两个部分：内核内存（Kernel space）、用户内存（User space）。用户内存是提供给各个进程主要空间，Linux给各个进程提供相同的虚拟内存空间；这使得进程之间相互独立，互不干扰。实现的方法是采用虚拟内存技术：给每一个进程一定虚拟内存空间，而只有当虚拟内存实际被使用时，才分配物理内存。如下图所示，对于32的Linux系统来说，一般将0～3G的虚拟内存空间分配做为用户空间，将3～4G的虚拟内存空间分配为内核空间；64位系统的划分情况是类似的。



从进程的角度来看，进程能直接访问的用户内存（虚拟内存空间）被划分为5个部分：代码区、数据区、堆区、栈区、未使用区。代码区中存放应用程序的机器代码，运行过程中代码不能被修改，具有只读和固定大小的特点。数据区中存放了应用程序中的全局数据，静态数据和一些常量字符串等，其大小也是固定的。堆是运行时程序动态申请的空间，属于程序运行时直接申请、释放的内存资源。栈区用来存放函数的传入参数、临时变量，以及返回地址等数据。未使用区是分配新内存空间的预备区域。

二、进程与JVM内存模型

JVM本质就是一个进程，因此其内存模型也有进程的一般特点。但是，JVM又不是一个普通的进程，其在内存模型上有许多崭新的特点，主要原因有两个：1.JVM将许多本来属于操作系统管理范畴的东西，移植到了JVM内部，目的在于减少系统调用的次数；2. Java NIO，目的在于减少用于读写IO的系统调用的开销。 JVM进程与普通进程内存模型比较如下图:



##### 1.用户内存

上图特别强调了JVM进程模型的代码区和数据区指的是JVM自身的，而非Java程序的。普通进程栈区，在JVM一般仅仅用做线程栈。JVM的堆区和普通进程的差别是最大的，下面具体详细说明：

首先是永久代。永久代本质上是Java程序的代码区和数据区。Java程序中类（class），会被加载到整个区域的不同数据结构中去，包括常量池、域、方法数据、方法体、构造函数、以及类中的专用方法、实例初始化、接口初始化等。这个区域对于操作系统来说，是堆的一个部分；而对于Java程序来说，这是容纳程序本身及静态资源的空间，使得JVM能够解释执行Java程序。

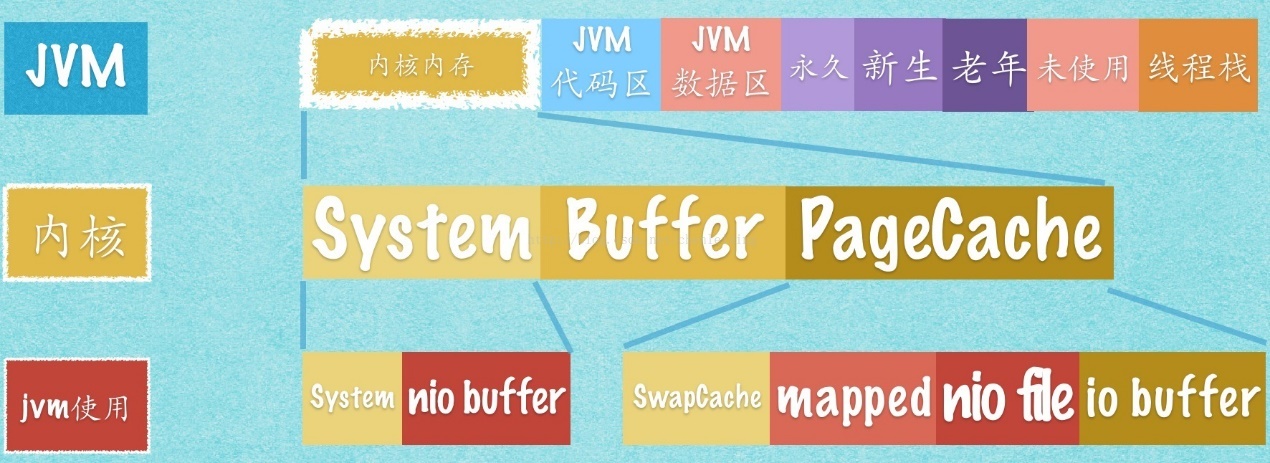
其次是新生代和老年代。新生代和老年代才是Java程序真正使用的堆空间，主要用于内存对象的存储；但是其管理方式和普通进程有本质的区别。  
普通进程在运行时给内存对象分配空间时，比如C++执行new操作时，会触发一次分配内存空间的系统调用，由操作系统的线程根据对象的大小分配好空间后返回；同时，程序释放对象时，比如C++执行delete操作时，也会触发一次系统调用，通知操作系统对象所占用的空间已经可以回收。  
JVM对内存的使用和一般进程不同。JVM向操作系统申请一整段内存区域（具体大小可以在JVM参数调节）作为Java程序的堆（分为新生代和老年代）；当Java程序申请内存空间，比如执行new操作，JVM将在这段空间中按所需大小分配给Java程序，并且Java程序不负责通知JVM何时可以释放这个对象的空间，垃圾对象内存空间的回收由JVM进行。

JVM的内存管理方式的优点是显而易见的，包括：第一，减少系统调用的次数，JVM在给Java程序分配内存空间时不需要操作系统干预，仅仅在Java堆大小变化时需要向操作系统申请内存或通知回收，而普通程序每次内存空间的分配回收都需要系统调用参与；第二，减少内存泄漏，普通程序没有（或者没有及时）通知操作系统内存空间的释放是内存泄漏的重要原因之一，而由JVM统一管理，可以避免程序员带来的内存泄漏问题。

最后是未使用区，未使用区是分配新内存空间的预备区域。对于普通进程来说，这个区域被可用于堆和栈空间的申请及释放，每次堆内存分配都会使用这个区域，因此大小变动频繁；对于JVM进程来说，调整堆大小及线程栈时会使用该区域，而堆大小一般较少调整，因此大小相对稳定。操作系统会动态调整这个区域的大小，并且这个区域通常并没有被分配实际的物理内存，只是允许进程在这个区域申请堆或栈空间。

##### 2.内核内存

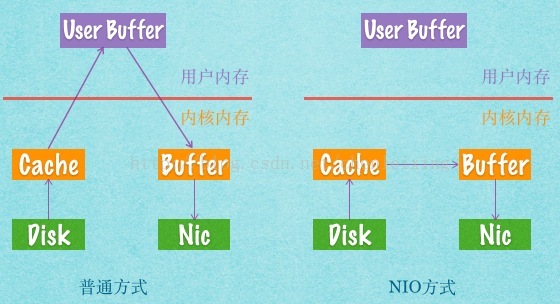
应用程序通常不直接和内核内存打交道，内核内存由操作系统进行管理和使用；不过随着Linux对性能的关注及改进，一些新的特性使得应用程序可以使用内核内存，或者是映射到内核空间。Java NIO正是在这种背景下诞生的，其充分利用了Linux系统的新特性，提升了Java程序的IO性能。



上图给出了Java NIO使用的内核内存在linux系统中的分布情况。nio buffer主要包括：nio使用各种channel时所使用的ByteBuffer、Java程序主动使用ByteBuffer.allocateDirector申请分配的Buffer。而在PageCache里面，nio使用的内存主要包括：FileChannel.map方式打开文件占用mapped、FileChannel.transferTo和FileChannel.transferFrom所需要的Cache（图中标示 nio file）。



通过JMX可以监控到NIO Buffer和 mapped 的使用情况，如下图所示。不过，FileChannel的实现是通过系统调用使用原生的PageCache，过程对于Java是透明的，无法监控到这部分内存的使用大小。



将数据在内核内存和用户内存之间拷贝是比较消耗资源和时间的事情，而从上图我们可以看到，通过NIO的方式减少了2次内核内存和用户内存之间的数据拷贝。这是Java NIO高性能的重要机制之一（另一个是异步非阻塞）。

从上面可以看出，内核内存对于Java程序性能也非常重要，因此，在划分系统内存使用时候，一定要给内核留出一定可用空间。

Java NIO使用的内存主要分布在Linux内核内存的System区和PageCache区。

## Neety高性能网络框架

网络编程

## 4.2mysql

## 4.3Redis

## 集群和主从复制是使用场景吗?一看就是背错答案的，主从复制是用来解决访高问量，降低服务器压力的，集群用来解决高可用和高吞吐量的，哨兵用来解决高可用和高访问量的，集群哨兵主从复制是解决方案，不是使用场景

## 4.4Kafka

## 4.5ELK

## 4.6分布式注册中心

## 4.7分布式锁

## 4.8分布式缓存

# 5.TCP/IP

## 5.1TCP/UDP

DNS（Domain Name System）

产生背景：在互联网的起源ARPANET中，起初由互联网信息中心整体管理一份host文件，（机器名与IP地址对应的关系），如果新增一台计算机接入到ARPANET网中或者已有的某台计算机要进行IP地址变更，中心的这个host文件就得更新，而其他计算机则不得不下载最新的host文件才能正常上网。

DNS ，可有效管理主机名与IP地址之间的对应关系的系统，也就是DNS系统。

进行DNS查询的主机和软件叫做DNS解析器。用户所使用的工作站或个人电脑都属于解析器。

ARP：两台主机通信。Ip和物理mac地址缺一不可（MAC地址用于识别同一链路的不同计算机，iP用来识别TCP/IP网络中互联的主机和路由器，在传输层中也有类似的概念，那就是端口号，它是用来识别同一台计算机中进行通信的不同应用程序）。两台主机不在同一个链路，中间由路由器，完全由路由器无法通信，通过发送ARP请求包（包含源主机的ip和MAC地址），（此请求会以广播形式发送，但是只有目标ip主机或者包含目的ip的路由器会回应，）包含目的ip地址的路由器会转发此请求到目的主机所在的链路，目的主机发送ARP包作为回应，包含自己ip和MAC地址。

DHCP（Dynamic Host Configuration Protocol动态主机配置协议）：DHCP网络服务器为每一个连接到网络的主机自动分配一个ip地址。

NAT(Natwork Address Translator):用于在本地网络中使用私有地址，在连接到互联网是转而使用全局ip的技术。它实际上是为了正在面临的地址枯竭的IPv4而开发的技术。

NAT路由器完成这个转换，，这种转换记录在转换表中。加入端口号就有了NAPT协议。

目标/源IP、目标/源端口号、协议类型（TCP/UDP）在一起构成了唯一的通信连接。

这种转换表在NAT路由器上自动生成，例如，在TCP的情况下，建立TCP连接首次握手时的SYN包一经发出，就会自动生成这张表，而随着收到关闭连接时发出FIN包的确认应答从表中删除。

TCP是面向连接的、可靠的流协议，流即使值不间断的意思。

TCP 为提供可靠传输，实行“顺序控制”，“重复控制“，此外还有“流量控制”，“拥塞控制”，提高网络利用率等众多功能。

UDP：用户数据报不可靠协议，不能保证数据可达，细微处理它会交给上层的应用去完成。

## Sverlet/Tomcate

## 5.3http

一条http请求发出经过哪些协议的传输道道服务器。

41、前端浏览器地址的一个 http 请求到后端整个流程是怎么样？

能够说下吗？

域名解析 --> 发起TCP的3次握手 --> 建立TCP连接后发起http请求 -->服务器响应http请求，浏览器得到html代码 -->浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） --> 浏览器对页面进行渲染呈现给用户

42、http 默认端口，https 默认端口

HTTP协议代理服务器常用端口号：80/8080/3128/8081/9080

HTTPS服务器，默认的端口号为443/tcp 443/udp

43、DNS 你知道是干嘛的吗？

DNS是指：域名服务器(Domain Name Server)。在Internet上域名与IP地址之间是一一对应的，域名虽然便于人们记忆，但机器之间只能互相认识IP地址，它们之间的转换工作称为域名解析，域名解析需要由专门的域名解析服务器来完成，DNS就是进行域名解析的服务器

## 5.4webSocket

# 6.部署

## K8S

## Openshift

## Docker