# 类

## 类的加载机制

虚拟机吧描述类的数据冲class文件加载到内，并队数据进行校验、转换解析的和初始化，最终形成可以被虚拟机直接使用的Java类型。

## 类的生命周期

加载、验证、准备、解析、初始化、使用、卸载

## 类的加载过程

加载、验证、准备、解析、初始化

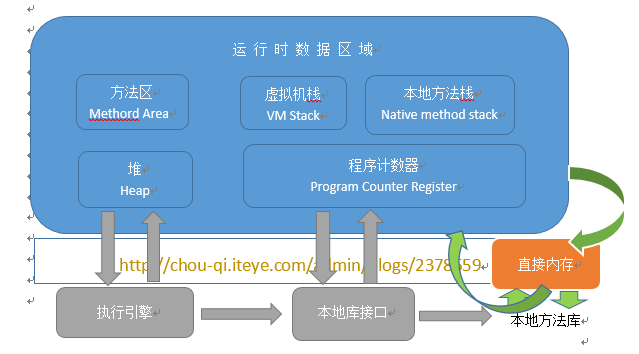
## 类加载器与双亲委派

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

# JVM

## jvm内存

方法区、虚拟机栈、本地方法栈、堆、程序计数器



## 垃圾收集器

(1) Serial（串行GC）-XX:+UseSerialGC

(2) ParNew（并行GC）-XX:+UseParNewGC

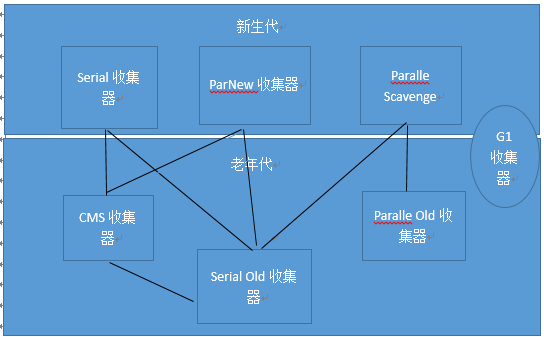
(3) Parallel Scavenge（并行回收GC）

(4) Serial Old（MSC）（串行GC）-XX:+UseSerialGC

(5) CMS（并发GC）-XX:+UseConcMarkSweepGC

(6) Parallel Old（并行GC）-XX:+UseParallelOldGC

(7) G1（JDK1.7update14才可以正式商用）



**二.1~3用于年轻代垃圾回收**：年轻代的垃圾回收称为minor GC **三.4~6用于年老代垃圾回收**（当然也可以用于方法区的回收）：年老代的垃圾回收称为full GC

G1独立完成"分代垃圾回收"

注意：并行与并发

并行：多条垃圾回收线程同时操作

并发：垃圾回收线程与用户线程一起操作

**八、调优方法**

8.1 新对象预留新生代

由于fullGC(老年代)的成本远比minorGC（新生代和老年代）的成本大，所以给应用分配一个合理的新生代空间，尽量将对象分配到新生代减小fullGC的频率  
8.2 大对象进入老年代

将大对象直接分配到老年代，保持新生代对象的结构的完整性，以提高GC效率， 以通过-XX:PretenureSizeThreshold设置进入老年代的阀值  
8.3 稳定与震荡的堆大小

稳定的对大小是对垃圾回收有利的，方法将-Xms和-Xmx的大小一致  
8.4 吞吐量优先

尽可能减少系统执行垃圾回收的总时间，故采用并行垃圾回收器

-XX:+UseParallelGC或使用-XX:+UseParallelOldGC  
8.5 降低停顿

使用CMS回收器,同时减少fullGC的次数

# 集合

## List

一.总结概述

List继承了Collection，是有序的列表。

实现类有ArrayList、LinkedList、Vector、Stack等

ArrayList是基于数组实现的，是一个数组队列。可以动态的增加容量！

LinkedList是基于链表实现的，是一个双向循环列表。可以被当做堆栈使用！

Vector是基于数组实现的，是一个矢量队列，是线程安全的！

Stack是基于数组实现的，是栈，它继承与Vector，特性是FILO（先进后出）！

二.使用场景

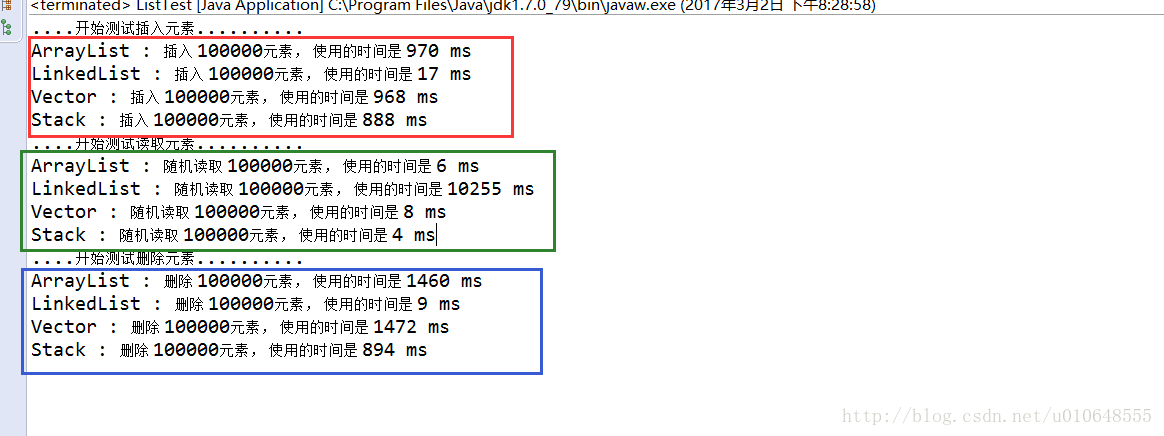
在实际的应用中如果使用到队列，栈，链表，首先可以想到使用List。不同的场景下面使用不同的工具，效率才能更高！

1. 当集合中对插入元素数据的速度要求不高，但是要求快速访问元素数据，则使用ArrayList！

2. 当集合中对访问元素数据速度不做要求不高，但是对插入和删除元素数据速度要求高的情况，则使用LinkedList！

3.当集合中有多线程对集合元素进行操作时候，则使用Vector！但是现在BVector现在一般不再使用，如需在多线程下使用，可以用CopyOnWriteArrayList，在java.util.concurrent包下。

4.当集合中有需求是希望后保存的数据先读取出来，则使用Stack！



### Arraylist 与 LinkedList 异同

* **1. 是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
* **2. 底层数据结构：** Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向循环链表数据结构；
* **3. 插入和删除是否受元素位置的影响：** ① **ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。** 比如：执行 add(E e)方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（ add(intindex,E element)）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ②**LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。**
* **4. 是否支持快速随机访问：** LinkedList 不支持高效的随机元素访问，而ArrayList 实现了RandmoAccess 接口，所以有随机访问功能。快速随机访问就是通过元素的序号快速获取元素对象(对应于 get(intindex)方法)。
* **5. 内存空间占用：** ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

### 补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表，如下图所示，同时下图也是LinkedList 底层使用的是双向循环链表数据结构。



### ArrayList 与 Vector 区别

Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

Arraylist不是同步的，所以在不需要保证线程安全时时建议使用Arraylist。

### ArrayList总结

1、ArrayList 本质实现方法是用数组！是非同步的！

2、初始化容量 = 10 ，最大容量不会超过 MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8！

3、indexOf和lastIndexOf 查找元素，若元素不存在，则返回-1！

4、当ArrayList容量不足以容纳全部元素时，ArrayList会重新设置容量：新的容量=“(原始容量x3)/2 ”。

5、ArrayList的克隆函数，即是将全部元素克隆到一个数组中。

6、ArrayList实现java.io.Serializable的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

7、从代码中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能确定元素数量的情况下，才使用ArrayList，否则建议使用LinkedList。

8、ArrayList的实现中大量地调用了Arrays.copyof()和System.arraycopy()方法。具体分析见第一篇参考文章

9、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。

10、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。

11、下面来看System.arraycopy()方法。该方法被标记了native，调用了系统的C/C++代码，在JDK中是看不到的，但在openJDK中可以看到其源码。该函数实际上最终调用了C语言的memmove()函数，因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适合用来批量处理数组。Java强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。

### LinkedList

1：LinkedList的实现是基于双向循环链表，实现的 List和Deque 接口。实现所有可选的列表操作，并允许所有元素（包括null）。

2：LinkedList是非线程安全的，只在单线程下适合使用。

3：这个类的iterator和返回的迭代器listIterator方法是fail-fast ，要注意ConcurrentModificationException 。

4：LinkedList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了Cloneable接口，能被克隆。

5:在查找和删除某元素时，都分为该元素为null和不为null两种情况来处理，LinkedList中允许元素为null。

6：由于是基于列表的，LinkedList的没有扩容方法！默认加入元素是尾部自动扩容！

7：LinkedList还实现了栈和队列的操作方法，因此也可以作为栈、队列和双端队列来使用，如peek 、push、pop等方法。

8：LinkedList是基于链表实现的，因此插入删除效率高，查找效率低！（因为查找需要遍历整个链表）

### Vector

1: Vector实际上是通过一个数组去保存数据的。当我们构造Vecotr时；若使用默认构造函数，则Vector的默认容量大小是10。

2: 当Vector容量不足以容纳全部元素时，Vector的容量会增加。若容量增加系数 大于0，则将容量的值增加“容量增加系数”；否则，将容量大小增加一倍。

3: Vector的克隆函数，即是将全部元素克隆到一个数组中。

4: 很多方法都加入了synchronized同步语句，来保证线程安全。

5: 同样在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，Vector中也允许元素为null。

6： 遍历Vector，使用索引的随机访问方式最快，使用迭代器最慢。

7： Vector很多地方都与ArrayList实现大同小异，现在已经基本不再使用。

## Set

1.HashSet是一个无序的集合，基于HashMap实现；TreeSet是一个有序的集合，基于TreeMap实现。

2.HashSet集合中允许有null元素，TreeSet集合中不允许有null元素。

3.HashSet和TreeSet都是非同步！在使用Iterator进行迭代的时候要注意fail-fast。

## Map

### hashMap

（1）：HashMap是通过哈希表来存储一个key-value的键值对，每个key对应一个value，允许key和value为null！ hash

（2）：HashMap 的实例有两个参数影响其性能：初始容量 和加载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。HashMap的容量不足的时候，可以自动扩容resize(),但是最大容量为MAXIMUM\_CAPACITY==2^30！

（3）：put和get都是分为null和非null进行判断！

（4）：resize非常耗时的操作，因此，我们在用HashMap的时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

（5）：求hash值和索引值的方法，这两个方法便是HashMap设计的最为核心的部分，二者结合能保证哈希表中的元素尽可能均匀地散列。

HashMap中则通过h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。

首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；

其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，

这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，

因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

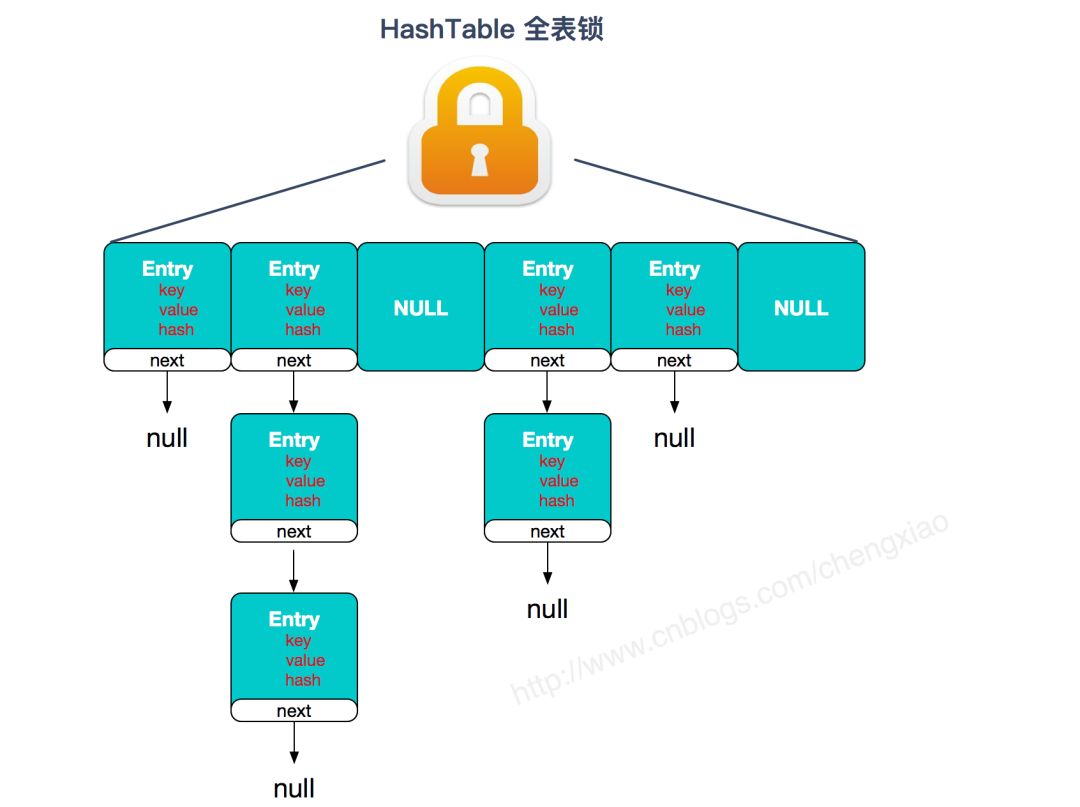
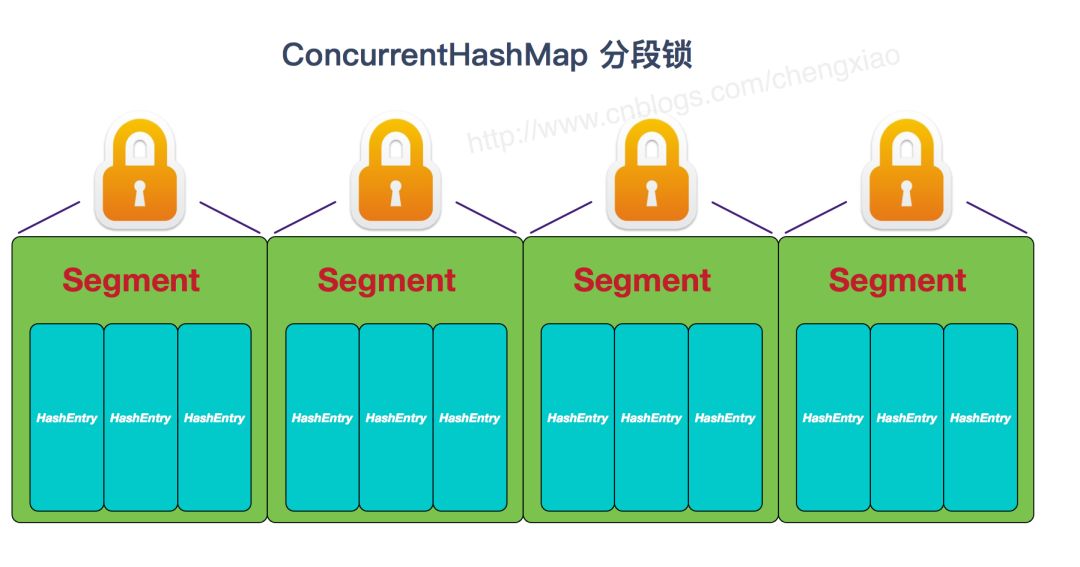
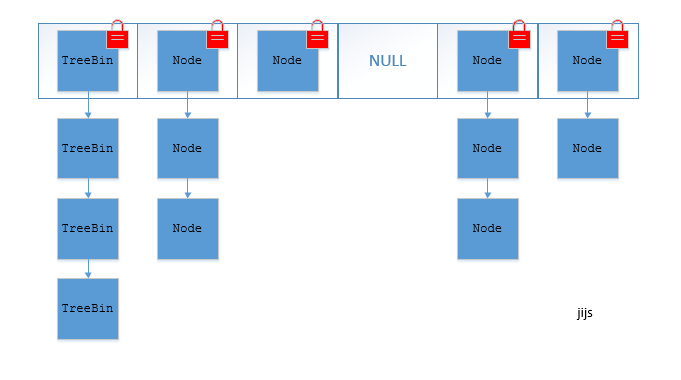
### HashMap 和 Hashtable 的区别

1. **线程是否安全：** HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. **效率：** 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. **对Null key 和Null value的支持：** HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。
4. **初始容量大小和每次扩充容量大小的不同 ：** ①创建时如果不指定容量初始值，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。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小。也就是说 HashMap 总是使用2的幂次方作为哈希表的大小,后面会介绍到为什么是2的幂次方。
5. **底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

### ConcurrentHashMap 和 Hashtable 的区别

https://zhuanlan.zhihu.com/p/21673805

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

* **底层数据结构：** JDK1.7的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构类似，**数组+链表/红黑二叉树**。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
* **实现线程安全的方式（重要）：** ① **在JDK1.7的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16倍。） **到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表/红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优化）** 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② **Hashtable(同一把锁)** :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争越激烈效率越低。
* HashTable:
* 
* JDK1.7的ConcurrentHashMap：
* 
* JDK1.8的ConcurrentHashMap（TreeBin: 红黑二叉树节点；Node: 链表节点）：
* 

### Treemap

1、TreeMap是根据key进行排序的，它的排序和定位需要依赖比较器或覆写Comparable接口，也因此不需要key覆写hashCode方法和equals方法，就可以排除掉重复的key，而HashMap的key则需要通过覆写hashCode方法和equals方法来确保没有重复的key。

2、TreeMap的查询、插入、删除效率均没有HashMap高，一般只有要对key排序时才使用TreeMap。

3、TreeMap的key不能为null，而HashMap的key可以为null。

4、TreeMap不是同步的。如果多个线程同时访问一个映射，并且其中至少一个线程从结构上修改了该映射，则其必须 外部同步。

# 多线程

## 线程池

## 线程的作用

1. 合理利用计算机多核处理器资源
2. 异步处理事件
3. 

## 守护线程

守护线程和用户线程的没啥本质的区别：唯一的不同之处就在于虚拟机的离开：如果用户线程已经全部退出运行了，只剩下守护线程存在了，虚拟机也就退出了。 因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了。

# Spring

## Ioc

## Aop

原理

## 事务传播

# Spring cloud

## 优点

# Dubbo

## 原理

与spring cloud 性能对比

# Redis

## 分布式锁

## 持久化

# 计算机网络

## http协议

## Tcp

## Udp

## Post

## get

## Session

## Cookie

# NIO/IO

## NIO

## IO

# 数据库

## 性能优化

## 索引

## B+树

## 事务

# Neo4j

# ORM框架

## Mybatis

## Hibernate

## Jpa

# RocketMq

## 消息持久化

## 消息消费

## 分布式事务

# Zookeeper

## 服务一致问题

# 设计模式

## 单例模式

## 工厂模式

## 代理模式

## 建造者模式