一、基础篇

网络基础

TCP三次握手

三次握手过程：

客户端——发送带有SYN标志的数据包——服务端　一次握手　Client进入syn_sent状态

服务端——发送带有SYN/ACK标志的数据包——客户端　二次握手　服务端进入syn_rcvd

客户端——发送带有ACK标志的数据包——服务端　三次握手　连接就进入Established状态

为什么三次：

主要是为了建立可靠的通信信道，保证客户端与服务端同时具备发送、接收数据的能力

为什么两次不行？

1、防止已失效的请求报文又传送到了服务端，建立了多余的链接，浪费资源

2、两次握手只能保证单向连接是畅通的。（为了实现可靠数据传输， TCP 协议的通信双方， 都必须维 护一个序列号， 以标识发送出去的数据包中， 哪些是已经被对方收到的。 三次握手的过程即是通信双方 相互告知序列号起始值， 并确认对方已经收到了序列号起始值的必经步骤；如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认）

TCP四次挥手过程

四次挥手过程：

客户端——发送带有FIN标志的数据包——服务端，关闭与服务端的连接 ，客户端进入FIN-WAIT-1状态

服务端收到这个 FIN，它发回一 个 ACK，确认序号为收到的序号加1，服务端就进入了CLOSE-WAIT状态

服务端——发送一个FIN数据包——客户端，关闭与客户端的连接，客户端就进入FIN-WAIT-2状态

客户端收到这个 FIN，发回 ACK 报文确认，并将确认序号设置为收到序号加1，TIME-WAIT状态

为什么四次：

因为需要确保客户端与服务端的数据能够完成传输。

CLOSE-WAIT：

这种状态的含义其实是表示在等待关闭

TIME-WAIT：

为了解决网络的丢包和网络不稳定所带来的其他问题，确保连接方能在时间范围内，关闭自己的连接

如何查看**TIME-WAIT**状态的链接数量？

netstat -an |grep TIME_WAIT|wc -l 查看连接数等待time_wait状态连接数

为什么会**TIME-WAIT**过多？解决方法是怎样的？

可能原因： 高并发短连接的TCP服务器上，当服务器处理完请求后立刻按照主动正常关闭连接

解决：负载均衡服务器；Web服务器首先关闭来自负载均衡服务器的连接

1、OSI与TCP/IP 模型

OSI七层：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层

TCP/IP五层：物理层、数据链路层、网络层、传输层、应用层

2、常见网络服务分层

应用层：HTTP、SMTP、DNS、FTP

传输层：TCP 、UDP

网络层：ICMP 、IP、路由器、防火墙

数据链路层：网卡、网桥、交换机

物理层：中继器、集线器

3、TCP与UDP区别及场景

类型	特点	性能	应用过场景	首部字节
TCP	面向连接、可靠、字节流	传输效率慢、所需资源多	文件、邮件传输	20-60
UDP	无连接、不可靠、数据报文段	传输效率快、所需资源少	语音、视频、直播	8个字节

基于TCP的协议：HTTP、FTP、SMTP

基于UDP的协议：RIP、DNS、SNMP

4、TCP滑动窗口，拥塞控制

TCP通过：应用数据分割、对数据包进行编号、校验和、流量控制、拥塞控制、超时重传等措施保证数据的可靠传输；

拥塞控制目的：为了防止过多的数据注入到网络中，避免网络中的路由器、链路过载

拥塞控制过程：TCP维护一个拥塞窗口，该窗口随着网络拥塞程度动态变化，通过慢开始、拥塞避免等算法减少网络拥塞的发生。

5、TCP粘包原因和解决方法

TCP粘包是指：发送方发送的若干包数据到接收方接收时粘成一包

发送方原因：

TCP默认使用Nagle算法（主要作用：减少网络中报文段的数量）：

收集多个小分组，在一个确认到来时一起发送、导致发送方可能会出现粘包问题

接收方原因：

TCP将接收到的数据包保存在接收缓存里，如果TCP接收数据包到缓存的速度大于应用程序从缓存中读取数据包的速度，多个包就会被缓存，应用程序就有可能读取到多个首尾相接粘到一起的包。

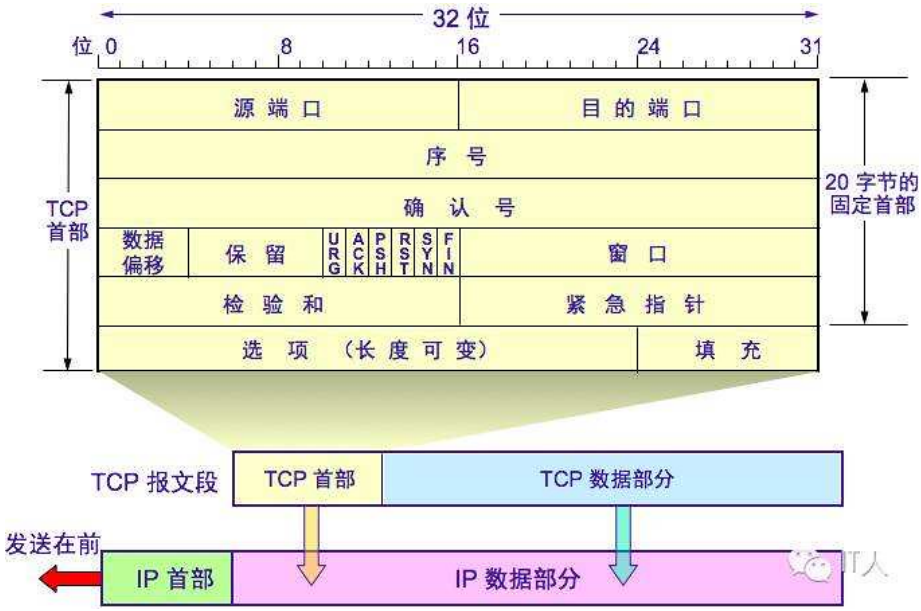
解决粘包问题：

最本质原因在与接收对等方无法分辨消息与消息之间的边界在哪，通过使用某种方案给出边界，例如：

- 发送定长包。每个消息的大小都是一样的，接收方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
- 包尾加上\r\n标记。FTP协议正是这么做的。但问题在于如果数据正文中也含有\r\n，则会误判为消息的边界。
- 包头加上包体长度。包头是定长的4个字节，说明了包体的长度。接收对等方先接收包体长度，依据包体长度来接收包体。

6、TCP、UDP报文格式

TCP报文格式：



源端口号和目的端口号：

用于寻找发端和收端应用进程。这两个值加上ip首部源端ip地址和目的端ip地址唯一确定一个tcp连接。

序号字段：

序号用来标识从T C P发端向T C P收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则 T C P用序号对每个字节进行计数。序号是32 bit的无符号数，序号到达 2^32-1后又从0开始。

当建立一个新的连接时，SYN标志变1。序号字段包含由这个主机选择的该连接的初始序号ISN（Initial Sequence Number）。该主机要发送数据的第一个字节序号为这个ISN加1，因为SYN标志消耗了一个序号

确认序号：

既然每个传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。只有ACK标志为 1时确认序号字段才有效。发送ACK无需任何代价，因为 32 bit的确认序号字段和A C K标志一样，总是T C P首部的一部分。因此，我们看到一旦一个连接建立起来，这个字段总是被设置， ACK标志也总是被设置为1。TCP为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

首都长度：

首部长度给出首部中 32 bit字的数目。需要这个值是因为任选字段的长度是可变的。这个字段占4 bit，因此T C P最多有6 0字节的首部。然而，没有任选字段，正常的长度是 2 0字节。

标志字段：在T C P首部中有 6个标志比特。它们中的多个可同时被设置为1.

URG紧急指针（u rgent pointer）有效

ACK确认序号有效。

PSH接收方应该尽快将这个报文段交给应用层。

RST重建连接。

SYN同步序号用来发起一个连接。这个标志和下一个标志将在第 1 8章介绍。

FIN发端完成发送任务。

窗口大小：

T C P的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端期望接收的字节。窗口大小是一个 16 bit字段，因而窗口大小最大为 65535字节。

检验和：

检验和覆盖了整个的 T C P报文段：T C P首部和T C P数据。这是一个强制性的字段，一定是由发端计算和存储，并由收端进行验证。

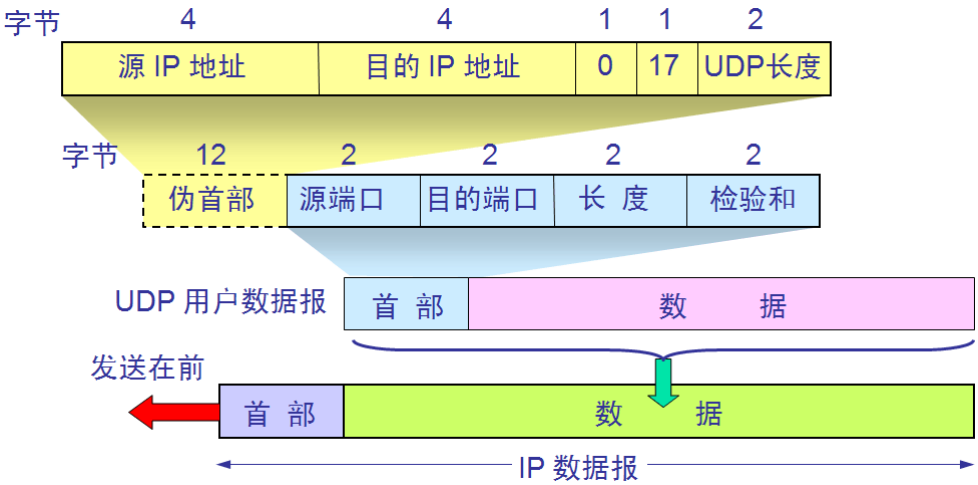
紧急指针：

只有当URG标志置1时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。 T C P的紧急方式是发送端向另一端发送紧急数据的一种方式。

选项：

最常见的可选字段是最长报文大小，又称为 MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段（为建立连接而设置 S Y N标志的那个段）中指明这个选项。它指明本端所能接收的最大长度的报文段。

UDP报文格式：



端口号：

用来表示发送和接受进程。由于 I P层已经把I P数据报分配给T C P或U D P（根据I P首部中协议字段值），因此T C P端口号由T C P来查看，而 U D P端口号由UDP来查看。T C P端口号与UDP端口号是相互独立的。

长度：

UDP长度字段指的是UDP首部和UDP数据的字节长度。该字段的最小值为 8字节（发送一份0字节的UDP数据报是 O K）。

检验和：

UDP检验和是一个端到端的检验和。它由发送端计算，然后由接收端验证。其目的是为了发现UDP首部和数据在发送端到接收端之间发生的任何改动。

IP报文格式： 普通的IP首部长为20个字节，除非含有可选项字段。

4位版本：

目前协议版本号是4，因此IP有时也称作IPV4.

4位首部长度：

首部长度指的是首部占32bit字的数目，包括任何选项。由于它是一个4比特字段，因此首部长度最长为60个字节。

服务类型（TOS）：

服务类型字段包括一个3bit的优先权字段（现在已经被忽略），4bit的TOS子字段和1bit未用位必须置0。4bit的TOS分别代表：最小时延，最大吞吐量，最高可靠性和最小费用。4bit中只能置其中1比特。如果所有4bit均为0，那么就意味着是一般服务。

总长度：

总长度字段是指整个IP数据报的长度，以字节为单位。利用首部长度和总长度字段，就可以知道IP数据报中数据内容的起始位置和长度。由于该字段长16bit，所以IP数据报最长可达65535字节。当数据报被分片时，该字段的值也随着变化。

标识字段：

标识字段唯一地标识主机发送的每一份数据报。通常每发送一份报文它的值就会加1。

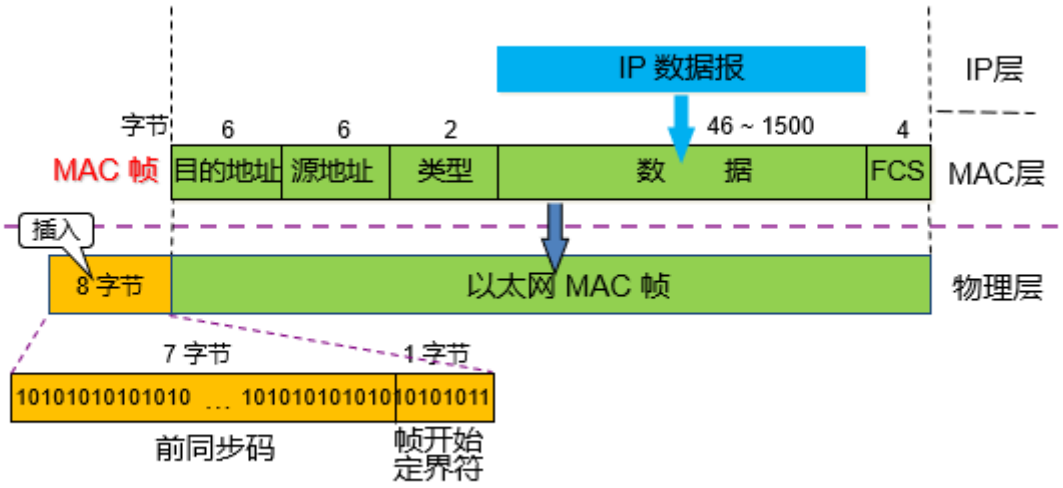
生存时间：

TTL（time-to-live）生存时间字段设置了数据报可以经过的最多路由器数。它指定了数据报的生存时间。TTL的初始值由源主机设置（通常为 3 2或6 4），一旦经过一个处理它的路由器，它的值就减去 1。当该字段的值为 0 时，数据报就被丢弃，并发送 ICMP 报文通知源主机。

首部检验和：

首部检验和字段是根据 I P首部计算的检验和码。它不对首部后面的数据进行计算。 ICMP、IGMP、UDP和TCP在它们各自的首部中均含有同时覆盖首部和数据检验和码。

以太网报文格式：



目的地址和源地址：

是指网卡的硬件地址（也叫MAC 地址），长度是48 位，是在网卡出厂时固化的。

数据：

以太网帧中的数据长度规定最小46 字节，最大1500 字节，ARP 和RARP 数据包的长度不够46 字节，要在后面补填充位。最大值1500 称为以太网的最大传输单元（MTU），不同的网络类型有不同的MTU，如果一个数据包从以太网路由到拨号链路上，数据包度大于拨号链路的MTU了，则需要对数据包进行分片fragmentation）。ifconfig命令的输出中也有“MTU:1500”。注意，MTU 个概念指数据帧中有效载荷的最大长度，不包括帧首部的长度。

HTTP协议

1、HTTP协议1.0_1.1_2.0

HTTP1.0：服务器处理完成后立即断开TCP连接（无连接），服务器不跟踪每个客户端也不记录过去的请求（无状态）

HTTP1.1：KeepAlived长连接避免了连接建立和释放的开销；通过Content-Length来判断当前请求数据是否已经全部接受（有状态）

HTTP2.0：引入二进制数据帧和流的概念，其中帧对数据进行顺序标识；因为有了序列，服务器可以并行的传输数据。

http1.0和http1.1的主要区别如下：

1、缓存处理：1.1添加更多的缓存控制策略（如：Entity tag，If-Match）

2、网络连接的优化：1.1支持断点续传

3、错误状态码的增多：1.1新增了24个错误状态响应码，丰富的错误码更加明确各个状态

4、Host头处理：支持Host头域，不在以IP为请求方标志

5、长连接：减少了建立和关闭连接的消耗和延迟。

http1.1和http2.0的主要区别：

1、新的传输格式：2.0使用二进制格式，1.0依然使用基于文本格式

2、多路复用：连接共享，不同的request可以使用同一个连接传输（最后根据每个request上的id号组合成正常的请求）

3、header压缩：由于1.X中header带有大量的信息，并且得重复传输，2.0使用encoder来减少需要传输的hearder大小

4、服务端推送：同google的SPDUY（1.0的一种升级）一样

2、HTTP与HTTPS之间的区别

HTTP与HTTPS之间的区别：

HTTP	HTTPS
默认端口80	HTTPS默认使用端口443
明文传输、数据未加密、安全性差	传输过程ssl加密、安全性较好
响应速度快、消耗资源少	响应速度较慢、消耗资源多、需要用到CA证书

HTTPS链接建立的过程：

1.首先客户端先给服务器发送一个请求

2.服务器发送一个SSL证书给客户端，内容包括：证书的发布机构、有效期、所有者、签名以及公钥

3.客户端对发来的公钥进行真伪校验，校验为真则使用公钥对对称加密算法以及对称密钥进行加密

4.服务器端使用私钥进行解密并使用对称密钥加密确认信息发送给客户端

5.随后客户端和服务端就使用对称密钥进行信息传输

对称加密算法：

双方持有相同的密钥，且加密速度快，典型对称加密算法：DES、AES

非对称加密算法：

密钥成对出现（私钥、公钥），私钥只有自己知道，不在网络中传输；而公钥可以公开。相比对称加密速度较慢，典型的非对称加密算法有：RSA、DSA

3、Get和Post请求区别

HTTP请求：

方法	描述
GET	向特定资源发送请求，查询数据，并返回实体
POST	向指定资源提交数据进行处理请求，可能会导致新的资源建立、已有资源修改
PUT	向服务器上传新的内容
HEAD	类似GET请求，返回的响应中没有具体的内容，用于获取报头
DELETE	请求服务器删除指定标识的资源
OPTIONS	可以用来向服务器发送请求来测试服务器的功能性
TRACE	回显服务器收到的请求，用于测试或诊断
CONNECT	HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器

get和Post区别：

	GET	POST
可见性	数据在URL中对所有人可见	数据不会显示在URL中
安全性	与post相比，get的安全性较差，因为所发送的数据是URL的一部分	安全，因为参数不会被保存在浏览器历史或web服务器日志中
数据长度	受限制，最长2kb	无限制
编码类型	application/x-www-form-urlencoded	multipart/form-data
缓存	能被缓存	不能被缓存

4、HTTP常见响应状态码

100：Continue --- 继续。客户端应继续其请求。

200：OK --- 请求成功。一般用于GET与POST请求。

301：Moved Permanently --- 永久重定向。

302：Found --- 暂时重定向。

400：Bad Request --- 客户端请求的语法错误，服务器无法理解。

403：Forbideen --- 服务器理解请求客户端的请求，但是拒绝执行此请求。

404：Not Found --- 服务器无法根据客户端的请求找到资源（网页）。

500：Internal Server Error --- 服务器内部错误，无法完成请求。

502：Bad Gateway --- 作为网关或者代理服务器尝试执行请求时，从远程服务器接收到了无效的响应。

5、重定向和转发区别

重定向：redirect：

地址栏发生变化

重定向可以访问其他站点（服务器）的资源

重定向是两次请求。不能使用request对象来共享数据

转发：forward：

转发地址栏路径不变

转发只能访问当前服务器下的资源

转发是一次请求，可以使用request对象共享数据

6、Cookie和Session区别。

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但两者有所区别：

Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。

cookie不是很安全，别人可以分析存放在本地的COOKIE并进行欺骗,考虑到安全应当使用session。

Cookie 一般用来保存用户信息，Session 的主要作用就是通过服务端记录用户的状态

浏览器输入URL过程

过程：DNS解析、TCP连接、发送HTTP请求、服务器处理请求并返回HTTP报文、浏览器渲染、结束

过程	使用的协议
1、浏览器查找域名DNS的IP地址 DNS查找过程（浏览器缓存、路由器缓存、DNS缓存）	DNS：获取域名对应的ip
2、根据ip建立TCP连接	TCP：与服务器建立连接
3、浏览器向服务器发送HTTP请求	HTTP：发送请求
4、服务器响应HTTP响应	HTTP
5、浏览器进行渲染	

操作系统基础

进程和线程的区别

进程：是资源分配的最小单位，一个进程可以有多个线程，多个线程共享进程的堆和方法区资源，不共享栈、程序计数器

线程：是任务调度和执行的最小单位，线程并行执行存在资源竞争和上下文切换的问题

协程：是一种比线程更加轻量级的存在，正如一个进程可以拥有多个线程一样，一个线程可以拥有多个协程。

1、进程间通信方式IPC

管道pipe：

亲缘关系使用匿名管道，非亲缘关系使用命名管道，管道遵循FIFO，半双工，数据只能单向通信；

信号：

信号是一种比较复杂的通信方式，用户调用kill命令将信号发送给其他进程。

消息队列：

消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受限等特点。

共享内存(share memory)：

- 使得多个进程可以可以直接读写同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。
- 由于多个进程共享一段内存，因此需要依靠某种同步机制（如信号量）来达到进程间的同步及互斥。

信号量(Semaphores)：

信号量是一个计数器，用于多进程对共享数据的访问，这种通信方式主要用于解决与同步相关的问题并避免竞争条件。

套接字(Sockets)：

简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

2、用户态和核心态

用户态：只能受限的访问内存，运行所有的应用程序

核心态：运行操作系统程序，cpu可以访问内存的所有数据，包括外围设备

为什么要有用户态和内核态：

由于需要限制不同的程序之间的访问能力, 防止他们获取别的程序的内存数据, 或者获取外围设备的数据, 并发送到网络

用户态切换到内核态的3种方式：

a. 系统调用

主动调用，系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

b. 异常

当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，比如缺页异常，这时会触发切换内核态处理异常。

c. 外围设备的中断

当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会由用户态到内核态的切换。

3、操作系统的进程空间

栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。

堆区（heap）— 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。

静态区（static）—存放全局变量和静态变量的存储

代码区(text)—存放函数体的二进制代码。

线程共享堆区、静态区

操作系统内存管理

存管理方式：页式管理、段式管理、段页式管理

分段管理：

将程序的地址空间划分为若干段（segment），如代码段，数据段，堆栈段；这样每个进程有一个二维地址空间，相互独立，互不干扰。段式管理的优点是：没有内碎片（因为段大小可变，改变段大小来消除内碎片）。但段换入换出时，会产生外碎片（比如4k的段换5k的段，会产生1k的外碎片）

分页管理：

在页式存储管理中，将程序的逻辑地址划分为固定大小的页（page），而物理内存划分为同样大小的页框，程序加载时，可以将任意一页放入内存中任意一个页框，这些页框不必连续，从而实现了离散分离。页式存储管理的优点是：没有外碎片（因为页的大小固定），但会产生内碎片（一个页可能填充不满）

段页式管理：

段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说 段页式管理机制 中段与段之间以及段的内部的都是离散的

1、页面置换算法FIFO、LRU

置换算法：先进先出FIFO、最近最久未使用LRU、最佳置换算法OPT

先进先出FIFO:

缺点：没有考虑到实际的页面使用频率，性能差、与通常页面使用的规则不符合，实际应用较少

最近最久未使用LRU:

原理：选择最近且最久未使用的页面进行淘汰

优点：考虑到了程序访问的时间局部性，有较好的性能，实际应用也比较多

缺点：没有合适的算法，只有适合的算法，lFU、random都可以

```java
/**
 * @program: Java
 * @description: LRU最近最久未使用置换算法，通过LinkedHashMap实现
 * @author: Mr.Li
 * @create: 2020-07-17 10:29
 **/
public class LRUCache {
    private LinkedHashMap<Integer,Integer> cache;
    private int capacity;   //容量大小

    /**
     *初始化构造函数
     * @param capacity
     */
    public LRUCache(int capacity) {
        cache = new LinkedHashMap<>(capacity);
        this.capacity = capacity;
    }

    public int get(int key) {
        //缓存中不存在此key，直接返回
        if(!cache.containsKey(key)) {
            return -1;
        }

        int res = cache.get(key);
        cache.remove(key);   //先从链表中删除
        cache.put(key,res);  //再把该节点放到链表末尾处
        return res;
    }

    public void put(int key,int value) {
        if(cache.containsKey(key)) {
            cache.remove(key); //已经存在，在当前链表移除
        }
        if(capacity == cache.size()) {
            //cache已满，删除链表头位置
            Set<Integer> keySet = cache.keySet();
            Iterator<Integer> iterator = keySet.iterator();
            cache.remove(iterator.next());
        }
        cache.put(key,value);  //插入到链表末尾
    }
}
```

```java
/**
 * @program: Java
 * @description: LRU最近最久未使用置换算法，通过LinkedHashMap内部removeEldestEntry方法实现
 * @author: Mr.Li
```

```
 * @create: 2020-07-17 10:59
 **/
class LRUCache {
    private Map<Integer, Integer> map;
    private int capacity;

    /**
     *初始化构造函数
     * @param capacity
     */
    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
                return size() > capacity;  // 容量大于capacity 时就删除
            }
        };
    }
    public int get(int key) {
        //返回key对应的value值，若不存在，返回-1
        return map.getOrDefault(key, -1);
    }

    public void put(int key, int value) {
        map.put(key, value);
    }
}
```

最佳置换算法OPT:

原理：每次选择当前物理块中的页面在未来长时间不被访问的或未来不再使用的页面进行淘汰

优点：具有较好的性能，可以保证获得最低的缺页率

缺点：过于理想化，但是实际上无法实现（没办法预知未来的页面）

2、死锁条件、解决方式。

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的下相互等待的现象；

死锁的条件：

互斥条件：进程对所分配到的资源不允许其他进程访问，若其他进程访问该资源，只能等待至占有该资源的进程释放该资源；

请求与保持条件：进程获得一定的资源后，又对其他资源发出请求，阻塞过程中不会释放自己已经占有的资源

非剥夺条件：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放

循环等待条件：系统中若干进程组成环路，环路中每个进程都在等待相邻进程占用的资源

解决方法：破坏死锁的任意一条件

乐观锁，破坏资源互斥条件，**CAS**

资源一次性分配，从而剥夺请求和保持条件、**tryLock**

可剥夺资源：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可剥夺的条件，**数据库deadlock超时**

资源有序分配法：系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，从而破坏环路等待的条件，**转账场景**

面向对象三大特性

特性：封装、继承、多态

封装：对抽象的事物抽象化成一个对象，并对其对象的属性私有化，同时提供一些能被外界访问属性的方法；

继承：子类扩展新的数据域或功能，并复用父类的属性与功能，单继承，多实现；

多态：通过继承（多个子类对同一方法的重写）、也可以通过接口（实现接口并覆盖接口）

1、Java与C++区别

不同点：c++支持多继承，并且有指针的概念，由程序员自己管理内存；Java是单继承，可以用接口实现多继承，Java 不提供指针来直接访问内存，程序内存更加安全，并且Java有JVM自动内存管理机制，不需要程序员手动释放无用内存

2、多态实现原理

多态的底层实现是动态绑定，即在运行时才把方法调用与方法实现关联起来。

静态绑定与动态绑定：

一种是在编译期确定，被称为静态分派，比如方法的重载；

一种是在运行时确定，被称为动态分派，比如方法的覆盖（重写）和接口的实现。

多态的实现

虚拟机栈中会存放当前方法调用的栈帧（局部变量表、操作栈、动态连接 、返回地址）。多态的实现过程，就是方法调用动态分派的过程，如果子类覆盖了父类的方法，则在多态调用中，动态绑定过程会首先确定实际类型是子类，从而先搜索到子类中的方法。这个过程便是方法覆盖的本质。

3、static和final关键字

static：可以修饰属性、方法

static修饰属性：

类级别属性，所有对象共享一份，随着类的加载而加载（只加载一次），先于对象的创建；可以使用类名直接调用。

static修饰方法：

随着类的加载而加载；可以使用类名直接调用；静态方法中，只能调用静态的成员，不可用this；

final：关键字主要用在三个地方：变量、方法、类。

final修饰变量：

如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；

如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

final修饰方法：

把方法锁定，以防任何继承类修改它的含义（重写）；类中所有的 private 方法都隐式地指定为 final。

final修饰类：

final 修饰类时，表明这个类不能被继承。final 类中的所有成员方法都会被隐式地指定为 final 方法。

一个类不能被继承，除了final关键字之外，还有可以私有化构造器。（内部类无效）

4、抽象类和接口

抽象类：包含抽象方法的类，即使用abstract修饰的类；抽象类只能被继承，所以不能使用final修饰，抽象类不能被实例化，

接口：接口是一个抽象类型，是抽象方法的集合，接口支持多继承，接口中定义的方法，默认是public abstract修饰的抽象方法

相同点：

① 抽象类和接口都不能被实例化

② 抽象类和接口都可以定义抽象方法，子类/实现类必须覆写这些抽象方法

不同点：

① 抽象类有构造方法，接口没有构造方法

③抽象类可以包含普通方法，接口中只能是public abstract修饰抽象方法（Java8之后可以）

③ 抽象类只能单继承，接口可以多继承

④ 抽象类可以定义各种类型的成员变量，接口中只能是public static final修饰的静态常量

抽象类的使用场景：

既想约束子类具有共同的行为（但不再乎其如何实现），又想拥有缺省的方法，又能拥有实例变量

接口的应用场景：

约束多个实现类具有统一的行为，但是不在乎每个实现类如何具体实现；实现类中各个功能之间可能没有任何联系

5、泛型以及泛型擦除

参考：**https://blog.csdn.net/baoyinwang/article/details/107341997**

泛型：

泛型的本质是参数化类型。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法。

泛型擦除：

Java的泛型是伪泛型，使用泛型的时候加上类型参数，在编译器编译生成的字节码的时候会去掉，这个过程成为类型擦除。

如List等类型，在编译之后都会变成 List。JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的。

可以通过反射添加其它类型元素

6、反射原理以及使用场景

Java反射：

是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法；并且都能够调用它的任意一个方法；

反射原理：

反射首先是能够获取到Java中的反射类的字节码，然后将字节码中的方法，变量，构造函数等映射成 相应的 Method、Filed、Constructor 等类

如何得到**Class**的实例：

1.类名.class(就是一份字节码)
2.Class.forName(String className);根据一个类的全限定名来构建Class对象
3.每一个对象多有getClass()方法:obj.getClass();返回对象的真实类型

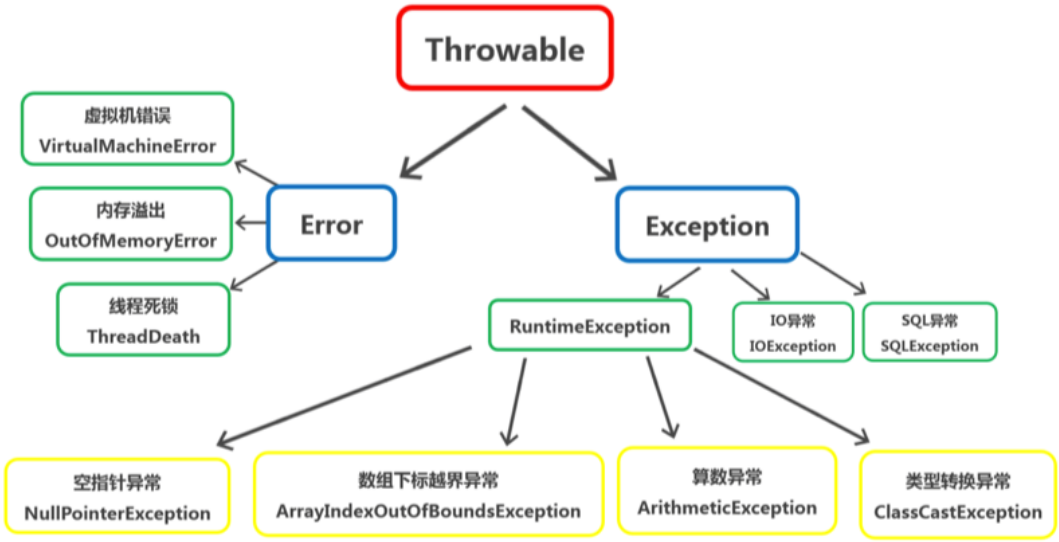
使用场景：

- 开发通用框架 - 反射最重要的用途就是开发各种通用框架。很多框架（比如 Spring）都是配置化的（比如通过 XML 文件配置 JavaBean、Filter 等），为了保证框架的通用性，需要根据配置文件运行时动态加载不同的对象或类，调用不同的方法。
- 动态代理 - 在切面编程（AOP）中，需要拦截特定的方法，通常，会选择动态代理方式。这时，就需要反射技术来实现了。

  JDK：spring默认动态代理，需要实现接口

  CGLIB：通过asm框架序列化字节流，可配置，性能差
- 自定义注解 - 注解本身仅仅是起到标记作用，它需要利用反射机制，根据注解标记去调用注解解释器，执行行为。

7、Java异常体系



Throwable 是 Java 语言中所有错误或异常的超类。下一层分为 Error 和 Exception

Error ：

是指 java 运行时系统的内部错误和资源耗尽错误。应用程序不会抛出该类对象。如果出现了这样的错误，除了告知用户，剩下的就是尽力使程序安全的终止。

Exception 包含：RuntimeException 、CheckedException

编程错误可以分成三类：语法错误、逻辑错误和运行错误。

语法错误（也称编译错误）是在编译过程中出现的错误，由编译器检查发现语法错误

逻辑错误指程序的执行结果与预期不符，可以通过调试定位并发现错误的原因

运行错误是引起程序非正常终端的错误，需要通过异常处理的方式处理运行错误

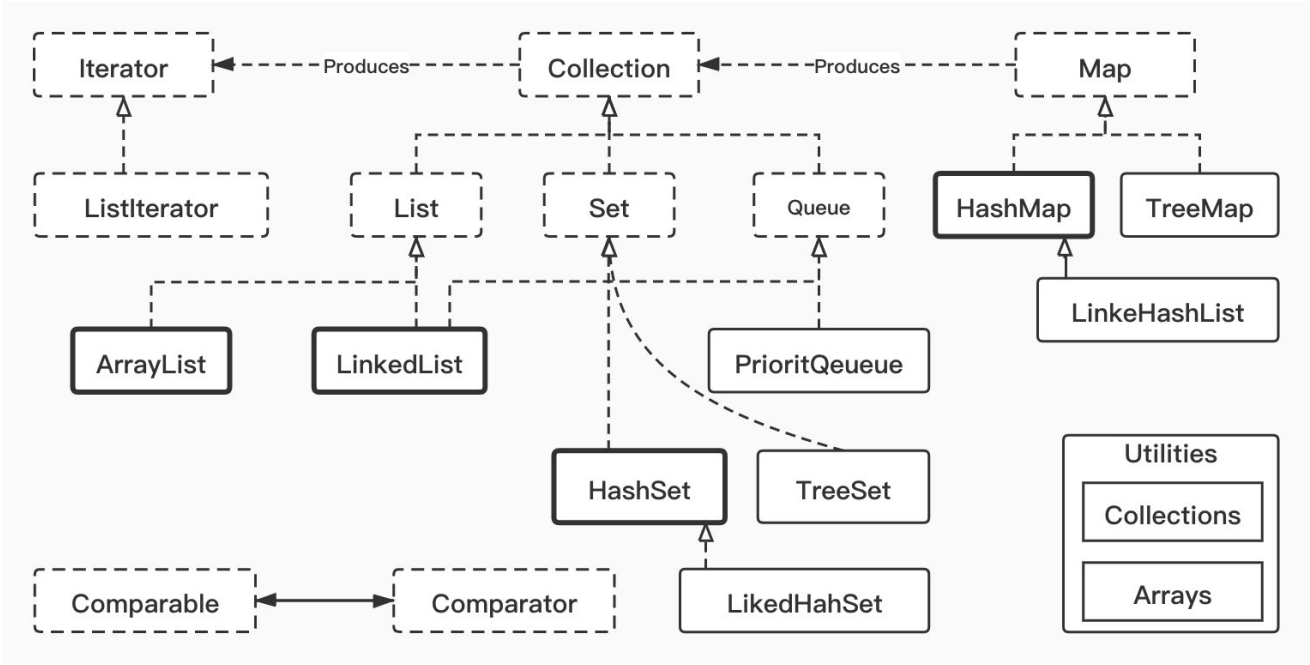
RuntimeException： 运行时异常，程序应该从逻辑角度尽可能避免这类异常的发生。

如 NullPointerException 、 ClassCastException ；

CheckedException： 受检异常，程序使用trycatch进行捕捉处理

如IOException、SQLException、NotFoundException；

数据结构



1、ArrayList和LinkedList

ArrayList：

底层基于数组实现，支持对元素进行快速随机访问，适合随机查找和遍历，不适合插入和删除。（提一句实际上）

默认初始大小为10，当数组容量不够时，会触发扩容机制（扩大到当前的1.5倍），需要将原来数组的数据复制到新的数组中；当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。

LinkedList：

底层基于双向链表实现，适合数据的动态插入和删除；

内部提供了 List 接口中没有定义的方法，用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。（比如jdk官方推荐使用基于linkedList的Deque进行堆栈操作）

ArrayList与LinkedList区别：

都是线程不安全的，ArrayList 适用于查找的场景，LinkedList 适用于增加、删除多的场景

实现线程安全：

可以使用原生的Vector，或者是Collections.synchronizedList(List list)函数返回一个线程安全的ArrayList集合。

建议使用concurrent并发包下的**CopyOnWriteArrayList**的。

①**Vector:** 底层通过synchronize修饰保证线程安全，效率较差

②**CopyOnWriteArrayList：** 写时加锁，使用了一种叫写时复制的方法；读操作是可以不用加锁的

2、List遍历快速和安全失败

①普通for循环遍历List删除指定元素

```
for(int i=0; i < list.size(); i++){
   if(list.get(i) == 5)
       list.remove(i);
}
```

② 迭代遍历,用list.remove(i)方法删除元素

```
Iterator<Integer> it = list.iterator();
while(it.hasNext()){
    Integer value = it.next();
    if(value == 5){
        list.remove(value);
    }
}
```

③foreach遍历List删除元素

```
for(Integer i:list){
    if(i==3) list.remove(i);
}
```

fail—fast：快速失败

当异常产生时，直接抛出异常，程序终止;

fail-fast主要是体现在当我们在遍历集合元素的时候，经常会使用迭代器，但在迭代器遍历元素的过程中，如果集合的结构（modCount）被改变的话，就会抛出异常ConcurrentModificationException，防止继续遍历。这就是所谓的快速失败机制。

fail—safe：安全失败

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。由于在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发ConcurrentModificationException。

缺点：基于拷贝内容的优点是避免了ConcurrentModificationException，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

3、详细介绍HashMap

角度：数据结构+扩容情况+put查找的详细过程+哈希函数+容量为什么始终都是2^N，JDK1.7与1.8的区别。

参考：**https://www.jianshu.com/p/9fe4cb316c05**

数据结构：

HashMap在底层数据结构上采用了数组＋链表＋红黑树，通过散列映射来存储键值对数据

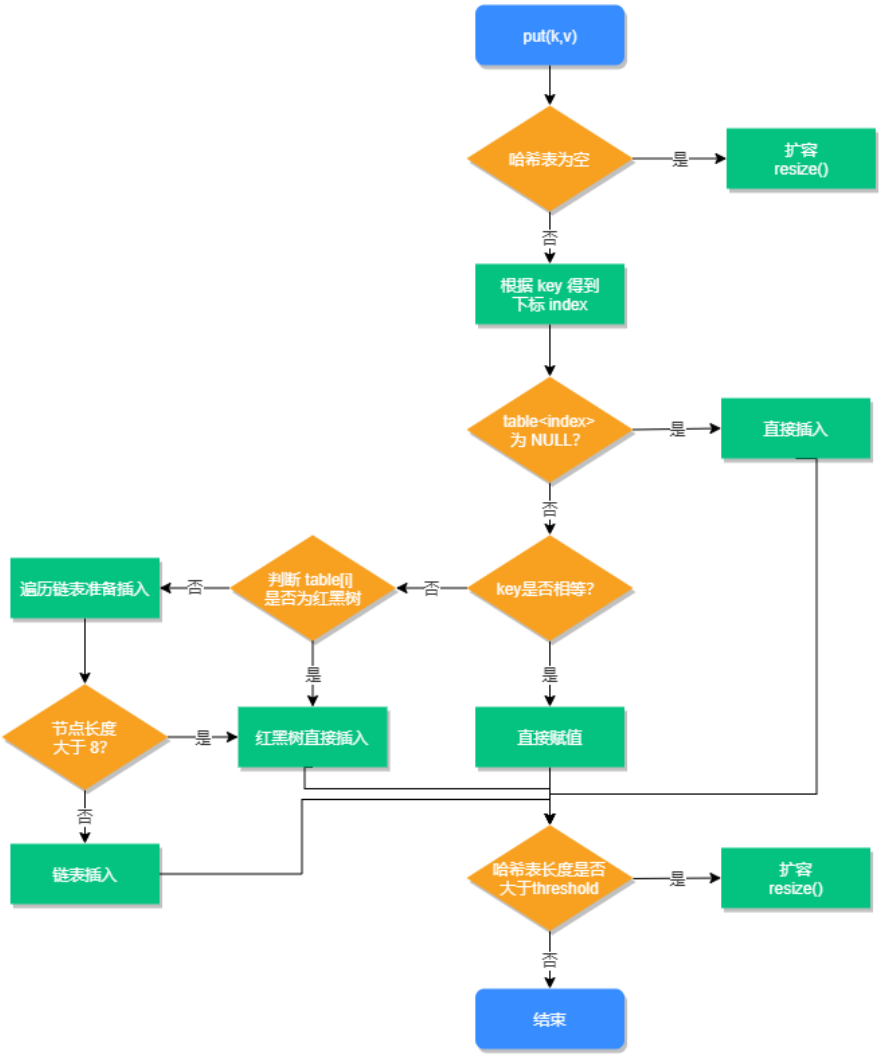
扩容情况：

默认的负载因子是0.75，如果数组中已经存储的元素个数大于数组长度的75%，将会引发扩容操作。

【1】创建一个长度为原来数组长度两倍的新数组。

【2】1.7采用Entry的重新hash运算，1.8采用高于与运算。

put操作步骤：



1、判断数组是否为空，为空进行初始化;

2、不为空，则计算 key 的 hash 值，通过(n - 1) & hash计算应当存放在数组中的下标 index;

3、查看 table[index] 是否存在数据，没有数据就构造一个Node节点存放在 table[index] 中；

4、存在数据，说明发生了hash冲突(存在二个节点key的hash值一样), 继续判断key是否相等，相等，用新的value替换原数据；

5、若不相等，判断当前节点类型是不是树型节点，如果是树型节点，创造树型节点插入红黑树中；

6、若不是红黑树，创建普通Node加入链表中；判断链表长度是否大于 8，大于则将链表转换为红黑树；

7、插入完成之后判断当前节点数是否大于阈值，若大于，则扩容为原数组的二倍

哈希函数：

通过hash函数（优质因子31循环累加）先拿到 key 的hashcode，是一个32位的值，然后让hashcode的高16位和低16位进行异或操作。该函数也称为扰动函数，做到尽可能降低hash碰撞，通过尾插法进行插入。

容量为什么始终都是2^N：

先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash ”。（n代表数组长度）。方便数组的扩容和增删改时的取模。

JDK1.7与1.8的区别：

JDK1.7 HashMap：

底层是 数组和链表 结合在一起使用也就是链表散列。如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。扩容翻转时顺序不一致使用头插法会产生死循环，导致cpu100%

JDK1.8 HashMap：

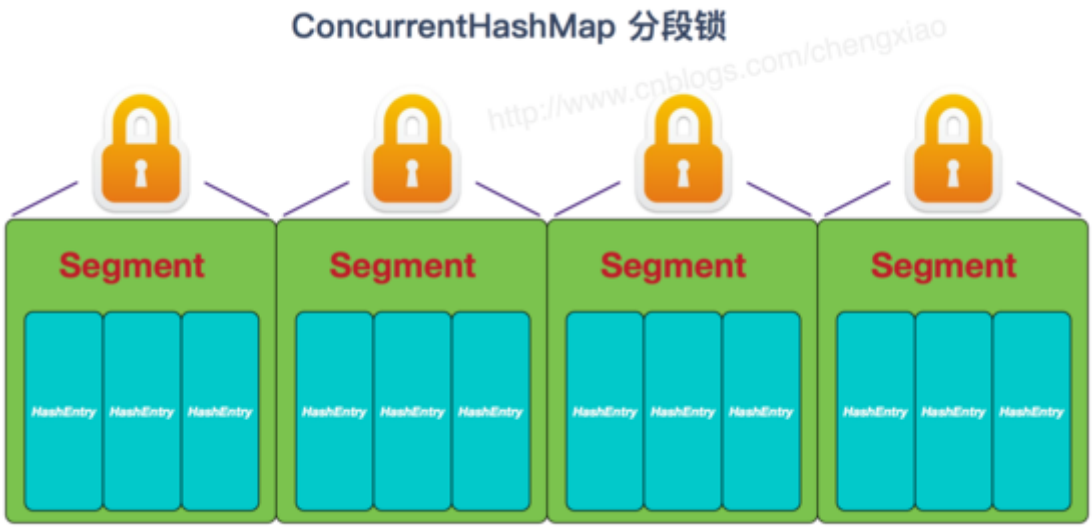
底层数据结构上采用了数组＋链表＋红黑树；当链表长度大于阈值（默认为 8-泊松分布），数组的长度大于64时，链表将转化为红黑树，以减少搜索时间。（解决了tomcat臭名昭著的url参数dos攻击问题）

4、ConcurrentHashMap

可以通过**ConcurrentHashMap** 和 **Hashtable**来实现线程安全；Hashtable 是原始API类，通过synchronize同步修饰，效率低下；ConcurrentHashMap 通过分段锁实现，效率较比Hashtable要好；

ConcurrentHashMap的底层实现：

JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现；采用 分段锁（Sagment） 对整个桶数组进行了分割分段(Segment默认16个)，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。



JDK1.8的 ConcurrentHashMap 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑树；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，通过并发控制 **synchronized** 和**CAS**来操作保证线程的安全。

5、序列化和反序列化

序列化的意思就是将对象的状态转化成字节流，以后可以通过这些值再生成相同状态的对象。对象序列化是对象持久化的一种实现方法，它是将对象的属性和方法转化为一种序列化的形式用于存储和传输。反序列化就是根据这些保存的信息重建对象的过程。

序列化：将java对象转化为字节序列的过程。

反序列化：将字节序列转化为java对象的过程。

优点：

a、实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上（通常存放在文件里）Redis的RDB

b、利用序列化实现远程通信，即在网络上传送对象的字节序列。 Google的protoBuf

反序列化失败的场景：

序列化ID：serialVersionUID不一致的时候，导致反序列化失败

6、String

String 使用数组存储内容，数组使用 **final** 修饰，因此 String 定义的字符串的值也是不可变的

StringBuffer 对方法加了同步锁，线程安全，效率略低于 StringBuilder

设计模式与原则

1、单例模式

某个类只能生成一个实例，该实例全局访问，例如Spring容器里一级缓存里的单例池。

优点：

唯一访问：如生成唯一序列化的场景、或者spring默认的bean类型。

提高性能：频繁实例化创建销毁或者耗时耗资源的场景，如连接池、线程池。

缺点：

不适合有状态且需变更的

实现方式：

饿汉式：线程安全速度快

懒汉式：双重检测锁，第一次减少锁的开销、第二次防止重复、volatile防止重排序导致实例化未完成

静态内部类：线程安全利用率高

枚举：effictiveJAVA推荐，反射也无法破坏

2、工厂模式

定义一个用于创建产品的接口，由子类决定生产何种产品。

优点：解耦：提供参数即可获取产品，通过配置文件可以不修改代码增加具体产品。

缺点：每增加一个产品就得新增一个产品类

3、抽象工厂模式

提供一个接口，用于创建相关或者依赖对象的家族，并由此进行约束。

优点：可以在类的内部对产品族进行约束

缺点：假如产品族中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。

面试题

构造方法

构造方法可以被重载，只有当类中没有显性声明任何构造方法时，才会有默认构造方法。

构造方法没有返回值，构造方法的作用是创建新对象。

初始化块

静态初始化块的优先级最高，会最先执行，在非静态初始化块之前执行。

静态初始化块会在类第一次被加载时最先执行，因此在 main 方法之前。

This

关键字 `this` 代表当前对象的引用。当前对象指的是调用类中的属性或方法的对象

关键字 `this` 不可以在静态方法中使用。静态方法不依赖于类的具体对象的引用

重写和重载的区别

重载指在同一个类中定义多个方法，这些方法名称相同，签名不同。

重写指在子类中的方法的名称和签名都和父类相同，使用override注解

Object类方法

toString 默认是个指针，一般需要重写

equals 比较对象是否相同，默认和==功能一致

hashCode 散列码，equals则hashCode相同，所以重写equals必须重写hashCode

finalize 用于垃圾回收之前做的遗嘱，默认空，子类需重写

clone 深拷贝，类需实现cloneable的接口

getClass 反射获取对象元数据，包括类名、方法、

notify、wait 用于线程通知和唤醒

基本数据类型和包装类

基本数据类型	存储大小	取值范围	默认值
byte	8 位有符号数	-2^7 到 2^7-1	0
short	16 位有符号数	-2^{15} 到 $2^{15}-1$	0
int	32 位有符号数	-2^{31} 到 $2^{31}-1$	0
long	64 位有符号数	-2^{63} 到 $2^{63}-1$	0L
float	32 位，符合 IEEE 754 标准	负数 -3.402823e+38 到 -1.401298e-45，正数 1.401298e-45 到 3.402823e+38	0.0f
double	64 位，符合 IEEE 754 标准	负数 -1.797693e+308 到 -4.9000000e-324，正数 4.9000000e-324 到 1.797693e+308	0.0d
char	16 位	0 到 $2^{16}-1$	'\u0000'
boolean	1 位	true 和 false	false

类型	缓存范围
Byte,Short,Integer,Long	[-128, 127]
Character	[0, 127]
Boolean	[false, true]