Java 基础知识复习

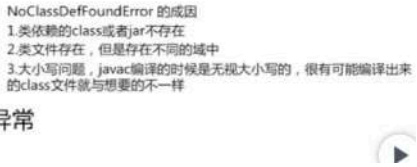
error 和 exception

1. **Error 与 exception 的区别**
2. **Error，runtime exception，non-runtime exception 的区别**
3. **常见的 Exception 与 error**

RuntimeException
1.NullPointerException - 空指针引用异常
2.ClassCastException - 类型强制转换异常
3.IllegalArgumentException - 传递非法参数异常
4.IndexOutOfBoundsException - 下标越界异常
5.NumberFormatException - 数字格式异常

非RuntimeException
1.ClassNotFoundException - 找不到指定class的异常
2.IOException - IO操作异常

Error
1.NoClassDefFoundError - 找不到class定义的异常
2.StackOverflowError - 深递归导致栈被耗尽而抛出的异常
3.OutOfMemoryError - 内存溢出异常



4. **ClassNotFoundException 与 NoClassDefFoundError 区别**

ClassNotFoundException是一个检查异常。从类继承层次上来看，ClassNotFoundException是从Exception继承的，所以ClassNotFoundException是一个检查异常。

当应用程序运行的过程中尝试使用类加载器去加载Class文件的时候，如果没有在classpath中查找到指定的类，就会抛出ClassNotFoundException。一般情况下，当我们使用Class.forName()或者ClassLoader.loadClass以及使用ClassLoader.findSystemClass()在运行时加载类的时候，如果类没有被找到，那么就会导致JVM抛出ClassNotFoundException。

NoClassDefFoundError异常，看命名后缀是一个Error。从类继承层次上看，NoClassDefFoundError是从Error继承的。和ClassNotFoundException相比，明显的一个区别是，NoClassDefFoundError并不需要应用程序去关心catch的问题。

当JVM在加载一个类的时候，如果这个类在编译时是可用的，但是在运行时找不到这个类的定义的时候，JVM就会抛出一个NoClassDefFoundError错误。比如当我们在new一个类的实例的时候，如果在运行是类找不到，则会抛出一个NoClassDefFoundError的错误。

5. **Java 的异常处理机制**

Finally 语句会先于 catch 的 return 语句执行

- 具体明确：抛出的异常应能通过异常类名和message准确说明异常的类型和产生异常的原因；
- 提早抛出：应尽可能早的发现并抛出异常，便于精确定位问题；
- 延迟捕获：异常的捕获和处理应尽可能延迟，让掌握更多信息的作用域来处理异常。

6. **Try-catch 的性能**

慢于 if 语句的判定

- try-catch块影响JVM的优化
- 异常对象实例需要保存栈快照等信息，开销较大

Java collection 集合

1. **数据结构常考点**

- 数组和链表的区别；
- 链表的操作，如反转，链表环路检测，双向链表，循环链表相关操作；
- 队列，栈的应用；
- 二叉树的遍历方式及其递归和非递归的实现；
- 红黑树的旋转；

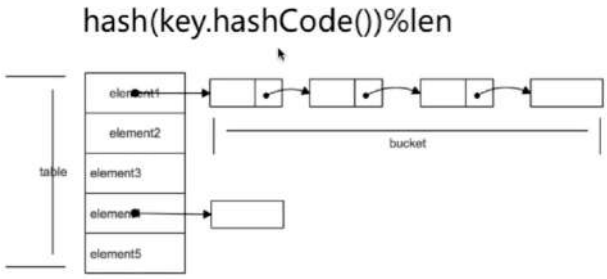
2. Java 集合框架

List 中可以包含重复元素，set 中不可以，不能包含 equals 为 true 的两个元素

a. Vector 不适用于高并发且对性能有较高要求的场景，方法需要有序执行，现在很少使用。
b. Arraylist 和 linkedlist 都线程不安全
c. Hashset 是基于 hashmap 实现的，以键的形式保存在 hashMap 里。（treeSet 与之相似，基于 treeMap）
d. 考虑到排序的时候，treeset 比 hashset 更方便
e. 实现 comparable 接口后，要实现 equals，compareTo 和 hashCode 三个函数，且 equals 判断相等的两个元素应该具有相同的 hashCode。
f. 向 treeSet 中添加元素，如果元素的类的内部实现了 compareTo 自然排序方法，外部的类实现了 comparator 接口和 compare 定制排序方法的情况下，是以定制排序为优先的

3. Hashmap, hashtable, concurrent hashmap 的区别

HashMap（Java8 以前）：数组+链表



1. 在最坏情况下，如果元素都存储在同一个 bucket 中，那么查询的复杂度从 O(1)变成了 O(n)，因此在 java8 之后使用了 TREEIFY_THRESHOLD 的阈值，超过就转成红黑树。因此最坏情况下，O(n)会下降到 O(logn).
2. Node 中包含 hash 值，k，v 和指向下一个 node
3. Hashmap 在第一次使用时才会初始化，如果为空，先调用 resize 方法初始化 table 的长度，然后通过 hash 运算，找到具体的位置。如果这个位置为空，就 new 一个新的 node，如果已经有相同的 k 的 node，就更新这个 node 的 v。否则向下一步走，判断 bucket 是否树化，如果是红黑树，就按照树的形式插入，如果不是，就按照链表的方式插入。同样插入后如果个数和整个 hashmap 的元素个数都超过了各自的 threshold，就进行树化。

HashMap：put方法的逻辑

1、若HashMap未被初始化，则进行初始化操作；
2、对Key求Hash值，依据Hash值计算下标；
3、若未发生碰撞，则直接放入桶中；
4、若发生碰撞，则以链表的方式链接到后面；
5、若链表长度超过阈值，且HashMap元素超过最低树化容量，则将链表转成红黑树；
6、若节点已经存在，则用新值替换旧值；
7、若桶满了(默认容量16*扩容因子0.75)，就需要resize(扩容2倍后重排)；

TREEIFY_THRESHOLD=8，MIN_TREEIFY_CAPACITY=64

两个都超过，树化。第一个超过第二个没有，就扩容。

4. Hashmap 的查询，计算 hash 值，找到桶下标，通过 equals 方法找到 key 值，返回 value
5. Hash 值计算也可以提升 hashmap 的性能，减少碰撞，使元素尽可能均匀分布

- 扰动函数：促使元素位置分布均匀，减少碰撞机率
- 使用final对象，并采用合适的equals()和hashCode()方法

6. Hashmap 扩容存在的问题

- 多线程环境下，调整大小会存在条件竞争，容易造成死锁
- rehashing是一个比较耗时的过程

7. Table 的大小是 2 的 n 次方，原因是因为可以通过散列值计算将 hash 值计算出桶下标，比取模运算更便捷。

4. ConcurrentHashMap

a. 如何优化 hashtable
将锁细粒度化，使用分段锁，将 table 分成 segment。
这是早期的 concurrentHashMap

b. 当前使用 CAS 和 synchronize 来保证线程安全

- 首先使用无锁操作CAS插入头节点，失败则循环重试
- 若头节点已存在，则尝试获取头节点的同步锁，再进行操作

c. 特有的一个参数 sizeCtl 用来做大小的控制，初始化和扩容的标识符。
d. 不允许插入 null 键
e. 插入 kv 使用的 put 方法

1. 判断Node[]数组是否初始化，没有则进行初始化操作
2. 通过hash定位数组的索引坐标，是否有Node节点，如果没有则使用CAS进行添加（链表的头节点），添加失败则进入下次循环。
3. 检查到内部正在扩容，就帮助它一块扩容。
4. 如果f!=null，则使用synchronized锁住f元素（链表/红黑二叉树的头元素）
   4.1 如果是Node(链表结构)则执行链表的添加操作。
   4.2 如果是TreeNode(树型结构)则执行树添加操作。
5. 判断链表长度已经达到临界值8，当然这个8是默认值，大家也可以去做调整，当节点数超过这个值就需要把链表转换为树结构。

5. 三者的区别

- HashMap线程不安全，数组+链表+红黑树
- Hashtable线程安全，锁住整个对象，数组+链表
- ConccurentHashMap线程安全，CAS+同步锁，数组+链表+红黑树
- HashMap的key、value均可为null，而其他的两个类不支持

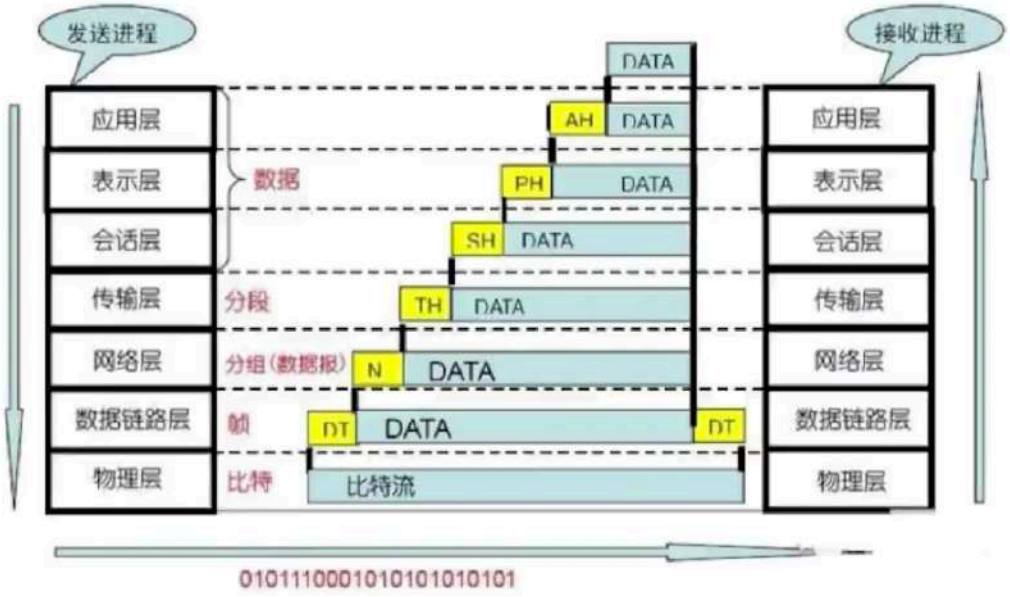
6. **红黑树**

JUC 包

1. Juc 包
   a. 是 java.util.concurrent 的缩写
   b. 包含的内容
      - 线程执行器executor
      - 锁locks
      - 原子变量类atomic
      - 并发工具类tools
      - 并发集合collections

网络基础知识

1. **OSI**
   a. 物理层：数据传输，bit 流与电流强弱转换，数模转换与模数转换
   b. 数据链路层：应对丢失漏传等问题，如何格式化数据进行传输，如何对物理介质访问，错误检测纠正。将 bit 数据组成帧，对帧解码，并根据帧的信息发送到接收方
   c. 网络层：点对点通信需要经过多个节点，如何选择最佳路径。将网络地址翻译成物理地址。如何从发送方路由到接收方。根据发送优先权，网络拥塞程度，服务质量和可选路由来决定最佳路径。发送的是数据包，切分成 segment 传输。**Ip**
   d. 传输层：解决主机之间的数据传输，解决传输质量的问题。传输协议基于流量控制，以及接受方的接受速率规定传输速率。按照网络能处理的最大尺寸将较大数据包分割成数据片，标号，到达接收方能正确重组，称为排序。**TCP/UDP**
   e. 会话层：建立和管理应用程序之间的通信。自动收发包和寻址。
   f. 表示层：解决不同系统之间语法不通的问题
   g. 应用层：使用固定长度的消息头包含消息中的信息。**http**



2. **TCP/IP**

OSI七层模型	TCP/IP概念层模型	功能	TCP/IP协议族
应用层	应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层		数据格式化，代码转换，数据加密	没有协议
会话层		解除或建立与别的接点的联系	没有协议
传输层	传输层	提供端对端的接口	TCP，UDP
网络层	网络层	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层	链路层	传输有地址的帧以及错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层		以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802，IEEE802.2

a. TCP 三次握手协议

IP 协议是无连接的通信协议，不会占用通信线路，每条线可以满足许多通信需要。通过 IP 数据会被分割为较小的独立的包，IP 负责将每个包路由至目的地。但 IP 并没有确认数据是否按顺序发送，或者包是否被破坏，因此 IP 数据包是不可靠的，需要靠上层协议来控制。

传输控制协议TCP简介

- 面向连接的、可靠的、基于字节流的传输层通信协议
- 将应用层的数据流分割成报文段并发送给目标节点的TCP层
- 数据包都有序号，对方收到则发送ACK确认，未收到则重传
- 使用校验和来检验数据在传输过程中是否有误

b. TCP 报头

1）使用 IP 地址和 TCP 中 sourcePort/destinationPort 来确认进行对话的唯一的现成。同一个主机内部的线程通信可以用 PID 来标示。

2）TCP Flags

- URG：紧急指针标志
- ACK：确认序号标志
- PSH：push标志
- RST：重置连接标志
- SYN：同步序号，用于建立连接过程
- FIN：finish标志，用于释放连接

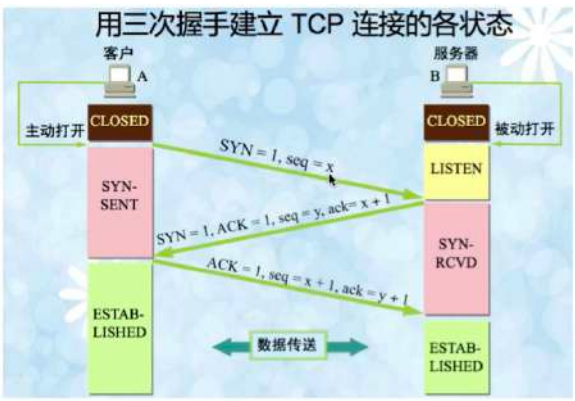
3）Window

滑动窗口大小，告知发送方接收方的缓存大小

4）Checksum

对整个 tcp 报文 check

c. 三次握手



d. 为什么需要三次握手建立连接

为了初始化双发的 sequence number 用于后续的数据传输。

避免首次握手中 SYN 超时的隐患

- Server收到Client的SYN，回复SYN-ACK的时候未收到ACK确认
- Server不断重试直至超时，Linux默认等待63秒才断开连接

e. Syn flood 攻击的解决措施

根据 src/des port 与时间戳建立 SYN，称为 tcp_syncookies

- SYN队列满后，通过tcp_syncookies参数回发SYN Cookie
- 若为正常连接则Client会回发SYN Cookie，直接建立连接

f. 建立连接后，client 出现故障怎么办

保活机制

发送保活探测报文，超过次数则中断连接

g. 四次挥手

Seq=U 中的 u 等于已发送的数据的最后一个字节的序号加 1，不发送信息也要占一个序列号，回执的时候序号加一

MSL 是最长报文段寿命

TCP 采用四次挥手来释放连接

第一次挥手：Client 发送一个 FIN，用来关闭 Client 到 Server 的数据传送，Client 进入 FIN_WAIT_1 状态；

第二次挥手：Server 收到 FIN 后，发送一个 ACK 给 Client，确认序号为收到序号+1（与 SYN 相同，一个 FIN 占用一个序号），Server 进入 CLOSE_WAIT 状态；

第三次挥手：Server 发送一个 FIN，用来关闭 Server 到 Client 的数据传送，Server 进入 LAST_ACK 状态；

第四次挥手：Client 收到 FIN 后，Client 进入 TIME_WAIT 状态，接着发送一个 ACK 给 Server，确认序号为收到序号+1，Server 进入 CLOSED 状态，完成四次挥手。

h. 为什么要等待 2MSL 的时间

- 确保有足够的时间让对方收到ACK包
- 避免新旧连接混淆

i. CloseWait 频繁出现的问题

```
[work@bjyz-ps-201612-m22-nwise1056.bjyz.baidu.com ~]$ netstat -n | awk '/^tcp/{++S[$NF]}END{for(a in S) print a,S[a]}'
TIME_WAIT 1507
CLOSE_WAIT 588
FIN_WAIT2 5
ESTABLISHED 413
```

Closewait 一直被保持就有可能造成新的连接请求无法被处理，有可能造成服务求的崩溃。

3. UDP

- 面向非连接
- 不维护连接状态，支持同时向多个客户端传输相同的消息
- 数据包报头只有8个字节，额外开销较小
- 吞吐量只受限于数据生成速率、传输速率以及机器性能
- 尽最大努力交付，不保证可靠交付，不需要维持复杂的链接状态表
- 面向报文，不对应用程序提交的报文信息进行拆分或者合并

Udp 适合从单个点向多个点发布信息

4. TCP 滑动窗口

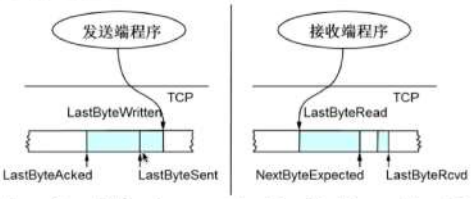
a. RTT 和 RTO

Round trip time 和 retransmission time out

b. Tcp 使用滑动窗口进行流量控制和字节重排

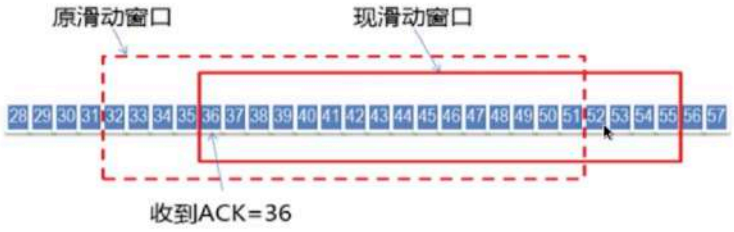
c. 数据窗口计算

窗口数据的计算过程

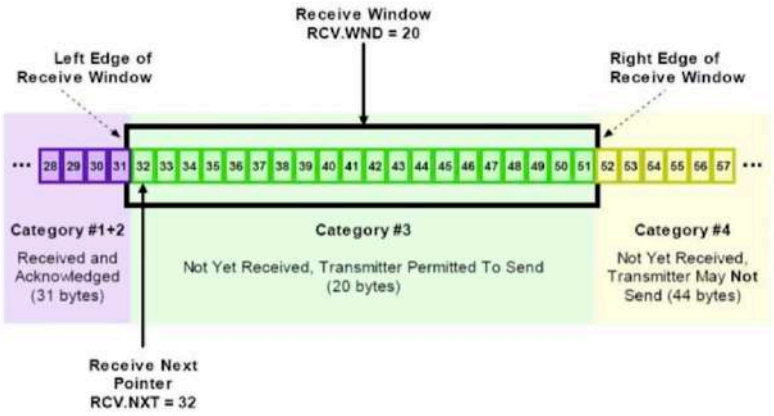


- AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd - LastByteRead)
- EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)

发送方的滑动窗口



接收方的滑动窗口



d. 确认重传机制是保证可靠性的基础

发送方只有在收到对窗口左边界数据的 ack 才会向右移动窗口，而接收方只有对窗口左边界数据确认后才会向右移动窗口，若前面的字节未接受但收到后面字节，窗口不会移动，以保证这些数据会重传。

e. 接收方根据实际情况调整窗口大小，从而实现对发送方的流量控制

5. HTTP

a. 主要特点

1）支持客户/服务器模式

2）通过简单的 get set 等方法，简单方便

3）传输数据类型灵活

4）本身是无连接的，长链接是通过下层实现的

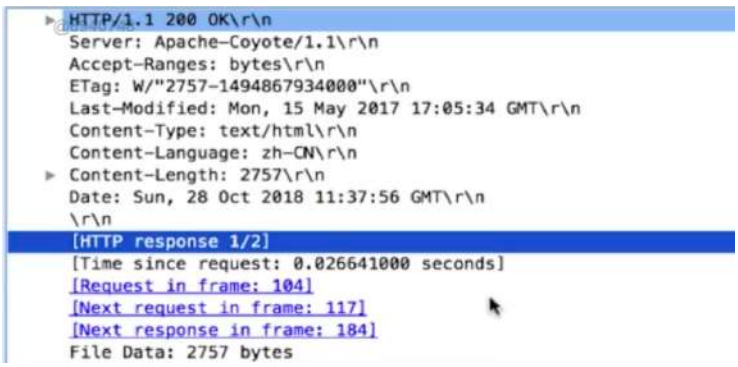
5）无状态

b. http 请求报文

分为 请求行-请求头部-空行-请求正文

空行是必须有的，即使正文为空

c. http 响应报文



d. 请求响应的步骤

- 客户端连接到Web服务器
- 发送HTTP请求
- 服务器接受请求并返回HTTP响应
- 释放连接TCP连接

若为 keep alive 可以继续保持连接

e. 面试问题

在浏览器地址栏键入URL，按下回车之后经历的流程

答案

- DNS解析
- TCP连接
- 发送HTTP请求
- 服务器处理请求并返回HTTP报文
- 浏览器解析渲染页面
- 连接结束

http 状态码

- 1xx：指示信息--表示请求已接收，继续处理
- 2xx：成功--表示请求已被成功接收、理解、接受
- 3xx：重定向--要完成请求必须进行更进一步的操作
- 4xx：客户端错误--请求有语法错误或请求无法实现
- 5xx：服务器端错误--服务器未能实现合法的请求

200 OK：正常返回信息

400 Bad Request：客户端请求有语法错误，不能被服务器所理解

401 Unauthorized：请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden：服务器收到请求，但是拒绝提供服务

404 Not Found：请求资源不存在，eg，输入了错误的 URL

500 Internal Server Error：服务器发生不可预期的错误

503 Server Unavailable：服务器当前不能处理客户端的请求，一段时间后可能恢复正常

f. Get 和 post 的区别 面试常考

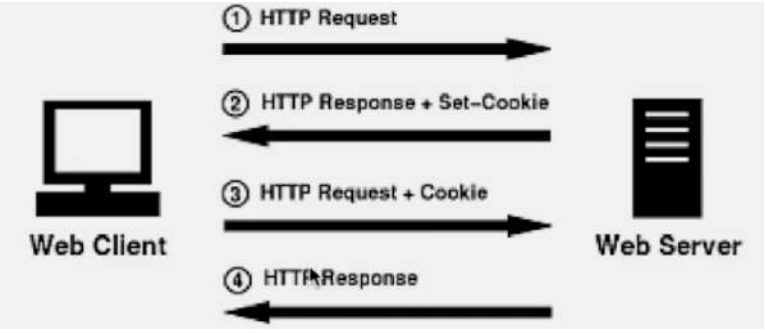
Get 的请求在 url 后面，用？隔开，以键值对形式表示

- Http报文层面：GET将请求信息放在URL，POST放在报文体中
- 数据库层面：CET符合幂等性和安全性，POST不符合
- 其他层面：GET可以被缓存、被存储，而POST不行

g. Cokkie 和 session

Cookie

- 是由服务器发给客户端的特殊信息，以文本的形式存放在客户端
- 客户端再次请求的时候，会把Cookie回发
- 服务器接收到后，会解析Cookie生成与客户端相对应的内容



Session

- 服务器端的机制，在服务器上保存的信息
- 解析客户端请求并操作session id，按需保存状态信息

可以通过 cookie 实现，在设置 cookie 时创建 sessionID 并通过 cookie 头部发送，之后客户端请求时 cookie 头就会包含 session ID
或者使用 url 回写来实现
两者的区别

- Cookie数据存放在客户的浏览器上，Session数据放在服务器上
- Session相对于Cookie更安全
- 若考虑减轻服务器负担，应当使用Cookie

h. http 和 https 的区别

https 在 http 的基础上加入了 ssl(3.0 版本后为 TLS)作为安全措施
SSL 为 security socket layer

- 为网络通信提供安全及数据完整性的一种安全协议
- 是操作系统对外的API，SSL3.0后更名为TLS
- 采用身份验证和数据加密保证网络通信的安全和数据的完整性

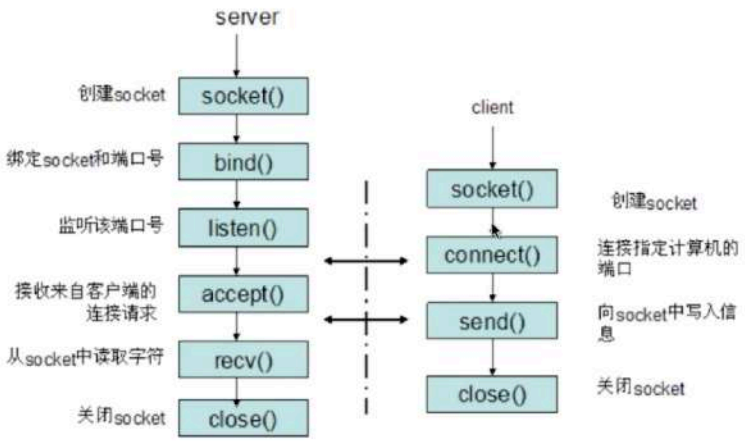
https 数据传输流程

- 浏览器将支持的加密算法信息发送给服务器
- 服务器选择一套浏览器支持的加密算法，以证书的形式回发浏览器
- 浏览器验证证书合法性，并结合证书公钥加密信息发送给服务器
- 服务器使用私钥解密信息，验证哈希，加密响应消息回发浏览器
- 浏览器解密响应消息，并对消息进行验真，之后进行加密交互数据

区别

- HTTPS需要到CA申请证书，HTTP不需要
- HTTPS密文传输，HTTP明文传输
- 连接方式不同，HTTPS默认使用443端口，HTTP使用80端口
- HTTPS=HTTP+加密+认证+完整性保护，较HTTP安全

6. Socket
   a. 流程



数据库知识

1. 常用考点

2. 数据库的模块化

3. 索引

a. 常见问题

- 为什么要使用索引
- 什么样的信息能成为索引
- 索引的数据结构
- 密集索引和稀疏索引的区别

主键，唯一键以及普通键等都可以作为索引

b. 索引的数据结构

- 生成索引，建立二叉查找树进行二分查找
- 生成索引，建立B-Tree结构进行查找
- 生成索引，建立B+-Tree结构进行查找
- 生成索引，建立Hash结构进行查找

c. 要在二叉树的一个节点上进行查找，要通过 IO 将这个节点读进来，为减少二叉树的 IO 次数，引入 B 树。当插入数据时，B 树的结构会被打乱，通过合并分裂上移下移等方法使得树满足要求

d. B 树要满足的条件

- 生成索引，建立二叉查找树进行二分查找
- 生成索引，建立B-Tree结构进行查找
- 生成索引，建立B+-Tree结构进行查找
- 生成索引，建立Hash结构进行查找

假设每个非终端结点中包含有 n 个关键字信息，其中

a) Ki (i=1...n)为关键字，且关键字按顺序升序排序 K(i-1)< Ki

b) 关键字的个数 n 必须满足： [ceil(m / 2)-1]<= n <= m-1

c) 非叶子结点的指针：P[1], P[2], ..., P[M]；其中 P[1]指向关键字小于 K[1]的子树，P[M]指向关键字大于 K[M-1]的子树，其它 P[i]指向关键字属于(K[i-1], K[i])的子树

e. B+树

与 B 树相似，但是非叶节点的子树指针与关键字的个数相同

- 非叶子节点的子树指针与关键字个数相同
- 非叶子节点的子树指针P[i]，指向关键字值[K[i], K[i+1])的子树
- 非叶子节点仅用来索引，数据都保存在叶子节点中
- 所有叶子节点均有一个链指针指向下一个叶子结点

叶子节点的指针可以方便我们做范围统计，比如查找到 10，然后找大于 10 的，此时就可以通过链指针向右移动。
所有的搜索都从根开始，到叶节点终结

为什么 B+树更适合做储存索引？

- B+树的磁盘读写代价更低
- B+树的查询效率更加稳定
- B+树更有利于对数据库的扫描

f. Hash 索引
效率更高，但是缺点也很明显

- 仅仅能满足“=”，“IN”，不能使用范围查询
- 无法被用来避免数据的排序操作
- 不能利用部分索引键查询
- 不能避免表扫描
- 遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高

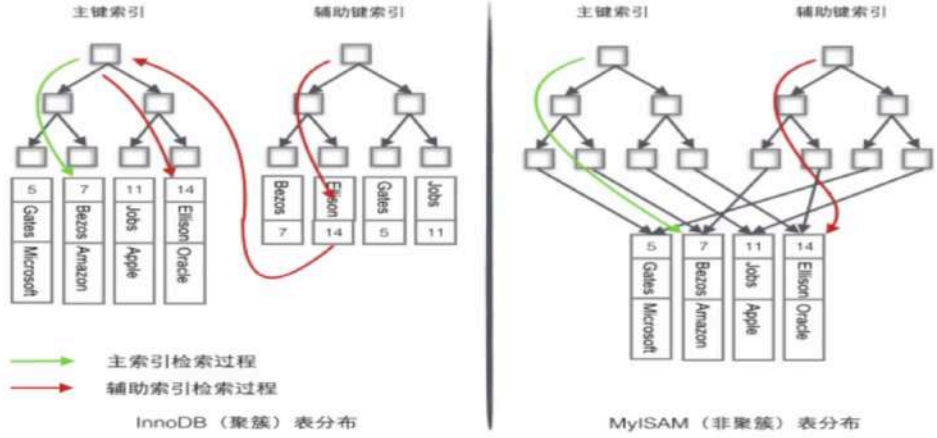
g. 对于关键字种类很少的情况，还可以采用 bitmap 的索引方式。
h. 密集索引和稀疏索引

- 密集索引文件中的每个搜索码值都对应一个索引值
- 稀疏索引文件只为索引码的某些值建立索引项

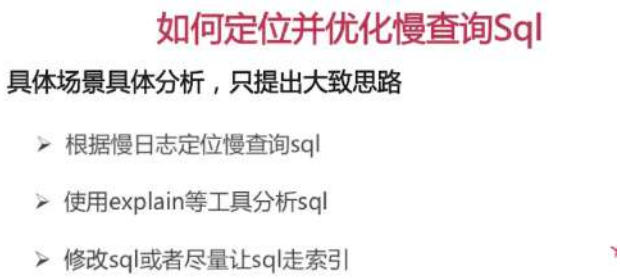
密集索引保存的不只是键值，还有同一行数据的其他信息，密集索引决定了一个表的物理排列顺序，一个表只能有一个密集索引。

InnoDB

- 若一个主键被定义，该主键则作为密集索引
- 若没有主键被定义，该表的第一个唯一非空索引则作为密集索引
- 若不满足以上条件，innodb内部会生成一个隐藏主键(密集索引)
- 非主键索引存储相关键位和其对应的主键值，包含两次查找



i. 如何调优 sql 的查询效率



j. 索引是建立的越多越好吗

- 数据量小的表不需要建立索引，建立会增加额外的索引开销
- 数据变更需要维护索引，因此更多的索引意味着更多的维护成本
- 更多的索引意味着也需要更多的空间

4. 锁模块

a. 常见问题

- MyISAM与InnoDB关于锁方面的区别是什么
- 数据库事务的四大特性
- 事务隔离级别以及各级别下的并发访问问题
- InnoDB可重复读隔离级别下如何避免幻读

b. MyISAM 和 InnoDB 在锁方面的区别？

- MyISAM默认用的是表级锁，不支持行级锁
- InnoDB默认用的是行级锁，也支持表级锁

myISAM 在读写的时候都为数据表加上表级别的读锁或者写锁
innoDB 支持事物，一个事物执行完毕之后自动 commit
innoDB 在 sql 使用到索引时默认使用行级锁以及 gap 锁，在 sql 没有使用到索引的时候使用表级锁

MyISAM 适用的场景？

- 频繁执行全表count语句
- 对数据进行增删改的频率不高，查询非常频繁
- 没有事务

InnoDB 适用的场景？

- 数据增删改查都相当频繁
- 可靠性要求比较高，要求支持事务

c. 行级锁一定要比表级锁要好吗？
不一定，锁的粒度越细，要花费的代价也就越高

d. 数据库锁的分类

- 按锁的粒度划分，可分为表级锁、行级锁、页级锁
- 按锁级别划分，可分为共享锁、排它锁
- 按加锁方式划分，可分为自动锁、显式锁
- 按操作划分，可分为DML锁、DDL锁
- 按使用方式划分，可分为乐观锁、悲观锁

自动锁是 sql 执行时自动加入的锁，显式锁是用 lock in share mode 等方式加入的锁

DML 是对数据修改加入的锁，DDL 是对数据库结构修改加入的锁

悲观锁一般使用数据库提供的锁，而乐观锁则是借助版本号和时间戳等方式

e. innoDB 非阻塞 select

5. 事物隔离机制

a. 事物并发访问可能产生的问题

- 更新丢失——mysql所有事务隔离级别在数据库层面上均可避免
- 脏读——READ-COMMITTED事务隔离级别以上可避免
- 不可重复读——REPEATABLE-READ事务隔离级别以上可避免
- 幻读——SERIALIZABLE事务隔离级别可避免

事务隔离级别	更新丢失	脏读	不可重复读	幻读
未提交读	避免	发生	发生	发生
已提交读	避免	避免	发生	发生
可重复读	避免	避免	避免	发生
串行化	避免	避免	避免	避免

不可重复读侧重于对同一数据的修改，而幻读侧重于更新和删除

mySQL 默认为 repeatable read，oracle 默认为 read committed

幻读：事物 a 执行了一个当前读操作，事物 b 与此同时执行了一个插入操作，事物 a 像出现了幻觉。

mySQL 的 repeatable read 级别下也可以避免幻读

b. 当前读和快照读

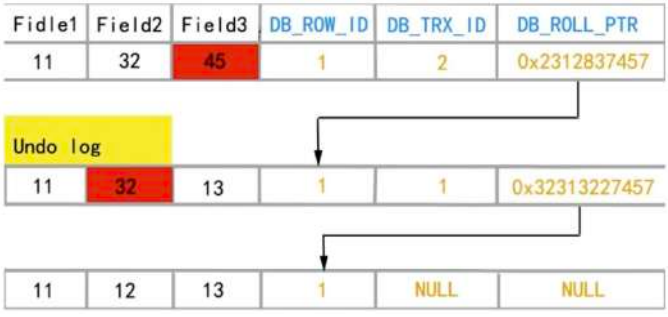
加了锁的增删改查语句均为当前读，保证读取的是最新版本而且读取之后其他事物不能修改记录，对读取的数据加锁

- 当前读：select...lock in share mode，select...for update
- 当前读：update，delete，insert
- 快照读：不加锁的非阻塞读，select

在 RR 级别下，快照读有可能读到数据的历史版本，也有可能读到最新版本，创建快照读的时机决定了读到的版本

c. RC，RR 的非阻塞读如何实现？

- 数据行里的DB_TRX_ID、DB_ROLL_PTR、DB_ROW_ID字段
- undo日志
- read view



将要修改数据的 DB_TRX_ID 与系统其他活跃事物 ID 做对比，如果大于或者等于，就利用 DB_ROLL_PTR 取出 undo log，直到 DB_TRX_ID 小于系统其他活跃事物 ID 为止，这样保证了获取的

数据版本是当前可见的最稳定的版本
RR 事物首次 select 的时候创建一个 read view
RC 事物每次 select 都会创建一个 read view
Undo log 记录了数据的串行修改过程，并不算多版本共存，因此是伪 MVCC

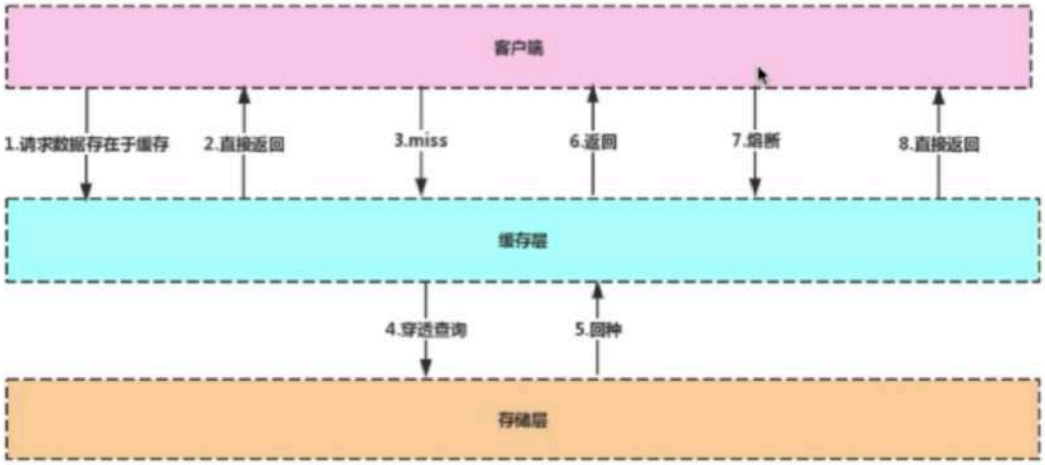
d. Gap 锁

- 如果where条件全部命中，则不会用Gap锁，只会加记录锁
- 如果where条件部分命中或者全不命中，则会加Gap锁

innoDB RR 级别通过引入 next key 锁避免幻读，next key 锁包含 record lock 和 gap lock，gap lock 会用在非唯一索引和不走索引的当前读，以及仅命中部分检索条件的结果集并且用到主键索引和唯一索引的当前读中

Redis

1. 主流数据库缓存架构



2. Memcache 和 redis 的区别

Memcache：代码层次类似Hash

- 支持简单数据类型
- 不支持数据持久化存储
- 不支持主从
- 不支持分片

Redis

- 数据类型丰富
- 支持数据磁盘持久化存储
- 支持主从
- 支持分片

Linux

1. **Linux 的体系结构**

- ◆ 体系结构主要分为用户态（用户上层活动）和内核态
- ◆ 内核：本质是一段管理计算机硬件设备的程序
- ◆ 系统调用：内核的访问接口，是一种不能再简化的操作
- ◆ 公用函数库：系统调用的组合拳
- ◆ Shell：命令解释器，可编程

2. **如何找到特定的文件？**

find

```
语法 find path [options] params
```

- ➢ 作用：在指定目录下查找文件

从当前目录和根目录查找

```
]$ find -name "target3.java"
]$ find / -name "target3.java"
```

以 target 开头的文件

```
$ find ~ -name "target*"
```

忽略大小写的影响

```
$ find ~ -iname "target*"
```

- ➢ find ~ -name “target3.java” ：精确查找文件
- ➢ find ~ -name “target*” ：模糊查找文件
- ➢ find ~ -iname “target*” ：不区分文件名大小写去查找文件
- ➢ man find：更多关于find指令的使用说明

3. **检索文件内容**

grep

```
语法：grep [options] pattern file
```

- ➢ 全称：Global Regular Expression Print
- ➢ 作用：查找文件里符合条件的字符串

从以 target 打头的文件中查找为 moo 的内容
只会筛选出目标字符串所在的行

```
$ grep "moo" target*
```

管道操作符 |

管道操作符 |

- ➢ 可将指令连接起来，前一个指令的输出作为后一个指令的输入

```
]$ find ~ | grep "target"
```

将 find 输出的结果作为输入部分传递给 grep

管道操作符需要注意的事项

- 只处理前一个命令正确输出，不处理错误输出
- 右边命令必须能够接收标准输入流，否则传递过程中数据会被抛弃
- sed,awk,grep,cut,head,top,less,more,wc,join,sort,split等

常用的 grep 指令

- grep 'partial\[true\]' bsc-plat-al-data.info.log
- grep -o 'engine\[[0-9a-z]*\]'
- grep -v 'grep'

将包含检索信息的行展示出来
将匹配正则表达式的部分展示出来
过滤掉一些内容，删去

4. 对日志内容做统计

awk

语法：awk [options] 'cmd' file

- 一次读取一行文本，按输入分隔符进行切片，切成多个组成部分
- 将切片直接保存在内建的变量中，$1,$2...($0表示行的全部)
- 支持对单个切片的判断，支持循环判断，默认分隔符为空格

打印第一列与第四列的内容

```
[xiangze@iZ287r27jubZ material]$ cat netstat.txt
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        0     48 115.28.159.6:ssh            113.46.178.155:63873        ESTABLISHED
tcp        0      0 localhost:mysql             localhost:40334             ESTABLISHED
tcp        0      0 localhost:mysql             localhost:40504             ESTABLISHED
tcp        0      0 115.28.159.6:webcache       175.167.138.54:16512        ESTABLISHED
tcp        0      0 115.28.159.6:webcache       175.167.138.54:16513        ESTABLISHED
udp        1      0 localhost:40504             localhost:mysql             ESTABLISHED
tcp        1      0 iZ287r27jubZ:44639          10.84.135.11:http           ESTABLISHED
tcp        1      0 localhost:40334             localhost:mysql             ESTABLISHED
tcp        0      0 115.28.159.6:46912          120.24.64.163:mysql         ESTABLISHED
[xiangze@iZ287r27jubZ material]$ awk '{print $1,$4}' netstat.txt
Proto Local
tcp 115.28.159.6:ssh
tcp localhost:mysql
tcp localhost:mysql
tcp 115.28.159.6:webcache
tcp 115.28.159.6:webcache
udp localhost:40504
tcp iZ287r27jubZ:44639
tcp localhost:40334
tcp 115.28.159.6:46912
[xiangze@iZ287r27jubZ material]$
```

打印符合特定条件的数据

```
[xiangze@iZ287r27jubZ material]$ awk '$1=="tcp" && $2==1{print $0}' netstat.txt
tcp        1      0 iZ287r27jubZ:44639          10.84.135.11:http           ESTABLISHED
tcp        1      0 localhost:40334             localhost:mysql             ESTABLISHED
```

保留第一行数据

```
[xiangze@iZ287r27jubZ material]$ awk '($1=="tcp" && $2==1) || NR==1 {print $0}' netstat.txt
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        1      0 iZ287r27jubZ:44639         10.84.135.11:http           ESTABLISHED
tcp        1      0 localhost:40334            localhost:mysql             ESTABLISHED
```

-F 表示以什么符号进行分割

```
[xiangze@iZ287r27jubZ material]$ cat test.txt
dafafa,1321313
dfafa,12131314
dfaa,232441
dfafaafaf,121341415151
ddafafag,1212131415
dfafafafagag,123131414
[xiangze@iZ287r27jubZ material]$ awk -F "," '{print $2}' test.txt
1321313
12131314
232441
121341415151
1212131415
123131414
```

数每一种 engine 出现的次数

```
[xiangze@iZ287r27jubZ material]$ grep 'partial\[true\]' bsc-plat-al-data.info.log | grep -o 'engine\[[0-9
a-z]*\]' | awk '{enginearr[$1]++}END{for(i in enginearr)print i "\t" enginearr[i]}'
engine[e76055e42183478c8fe94222c8d6bbc6]        4
engine[d5c01b3e2b68493fbc7439be6d1cbd92]        2
engine[75d5229f664648d9ab87e334779b171d]        7
engine[7e88aa4e40ed460183654f58d8168b18]        10
engine[8c9a0ebddb6d450eb427cb7cc5132463]        10
engine[7eb1475e20a5477fb124c447ba6d6f65]        8
```

常用命令

- awk '{print $1, $4}' netstat.txt
- awk '$1=="tcp" && $2==1 {print $0}' netstat.txt
- awk '{enginearr[$1]++}END{for(i in enginearr)print i"\t"enginearr[i]}'

5. 批量替换文本内容

sed

语法：sed [option] 'sed command'filename

- 全名stream editor，流编辑器
- 适合用于对文本的行内容进行处理
- sed -i 's/^Str/String/' replace.java
- sed -i's/\.$/\;/' replace.java
- sed -i's/Jack/me/g' replace.java

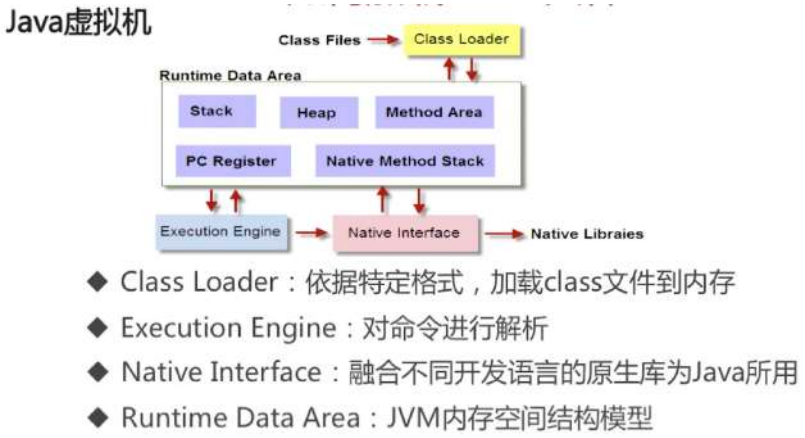
g 代表对每一行全部进行替换，么有 g 的时候仅仅替换出现的第一次符合要求的字符

```
[xiangze@iZ287r27jubZ material]$ sed -i '/^ *$/d' replace.java
[xiangze@iZ287r27jubZ material]$ cat replace.java
String a = "The beautiful girl's boy friend is me";
String b = "The beautiful girl often chats with me and me is me";
String c = "The beautiful girl loves me so much";
Integer bf = new integer(2);
```

删除空行

JVM

1. **javap 是 java 自带的反编译器**，可以查看 java 编译出的字节码
   java 平台无关性
   为什么要先编译成字节码而不是直接编译成机器码
2. **jvm 如何加载.class 文件**
   jvm 内存结构模型和 GC
   jvm 是运行在内存中的虚拟机，jvm 的存储就是内存
3. **java 虚拟机结构**



- Class Loader：依据特定格式，加载class文件到内存
- Execution Engine：对命令进行解析
- Native Interface：融合不同开发语言的原生库为Java所用
- Runtime Data Area：JVM内存空间结构模型

4. **反射**

JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 java 语言的反射机制。

```
public class ReflectSample {
    public static void main(String[] args) throws ClassNotFoundException, IllegalAccessExceptio
        Class rc = Class.forName("com.interview.javabasic.reflect.Robot");
        Robot r = (Robot) rc.newInstance();
        System.out.println("Class name is " + rc.getName());
        Method getHello = rc.getDeclaredMethod( name: "throwHello", String.class);
        getHello.setAccessible(true);
        Object str = getHello.invoke(r, ...args: "Bob");
        System.out.println("getHello result is " + str);
        Method sayHi = rc.getMethod( name: "sayHi", String.class);
        sayHi.invoke(r, ...args: "Welcome");
        Field name = rc.getDeclaredField( name: "name");
        name.setAccessible(true);
        name.set(r, "Alice");
        sayHi.invoke(r, ...args: "Welcome");
```

将 java 类中的各种成分映射成对象

5. **类从编译到执行的过程**

- 编译器将Robot.java源文件编译为Robot.class字节码文件
- ClassLoader将字节码转换为JVM中的Class<Robot>对象
- JVM利用Class<Robot>对象实例化为Robot对象

6. **Class Loader**

ClassLoader 在 Java 中有着非常重要的作用，它主要工作在 Class 装载的加载阶段，其主要作用是从系统外部获得 Class 二进制数据流。它是 Java 的核心组件，所有的 Class 都是由 ClassLoader 进行加载的，ClassLoader 负责通过将 Class 文件里的二进制数据流装载进系统，然后交给 Java 虚拟机进行连接、初始化等操作。

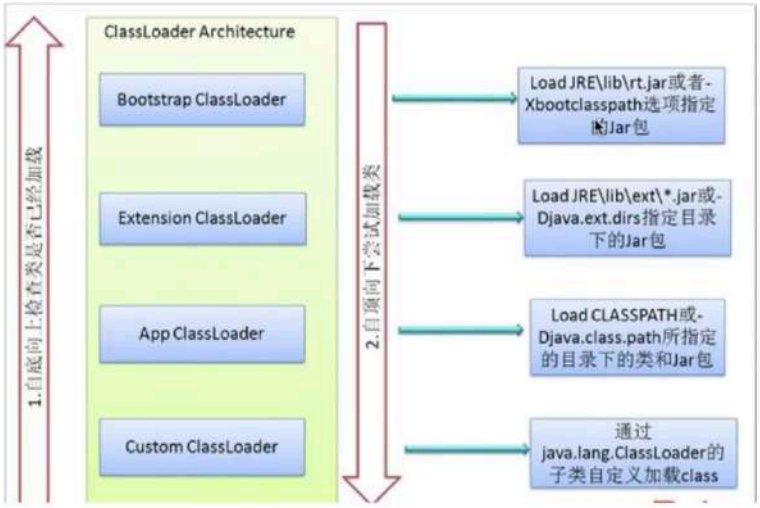
第一种是用来加载 java 必要的写死的类，如 java.lang，用 C++实现，对用户不可见
第二种是用来加载 jar 包和用户定义的 jar 包
第三种是从 classPath 去加载工程中定义的类文件
第四种是用户自定义的类加载器，用来加载工程外部的.class 文件，需要重写函数

- BootStrapClassLoader：C++编写，加载核心库 java.*
- ExtClassLoader：Java编写，加载扩展库 javax.*
- AppClassLoader：Java编写，加载程序所在目录
- 自定义ClassLoader：Java编写，定制化加载

ExtClassLoader 在需要的时候才会把类加载进来
自定义的时候主要需要 overload 两个方法
findclass()根据名字和地址加载字节码，可以用 inputStream 读取数据，通过 ByteArrayOutputStream 将得到的字节码以 byte 数组的形式输出，可以定义一个 loadClassData 函数来实现到 Byte 数组的转换
defineClass()解析字节流返回 class 对象

7. **双亲委派机制**

使几种不同的 classloader 可以相互协作的机制



通过 loadClass 自下而上（从用户定义的 classloader 开始）得查找四种类加载器是否已经加载过类，如果没有则上而下得（从 bootstrap class loader 开始）尝试加载类

- **启动（Bootstrap）类加载器**：是用本地代码实现的类装入器，它负责将 <Java_Runtime_Home>/lib下面的类库加载到内存中（比如rt.jar）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
- **标准扩展（Extension）类加载器**：是由 Sun 的 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将< Java_Runtime_Home >/lib/ext或者由系统变量 java.ext.dir指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。
- **系统（System）类加载器**：是由 Sun 的 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。开发者可以直接使用系统类加载器。

2. 双亲委派机制描述
某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，**依次递归**，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

为什么要使用双亲委派机制去加载类？
避免了多份同样的字节码的重复加载，节省内存空间

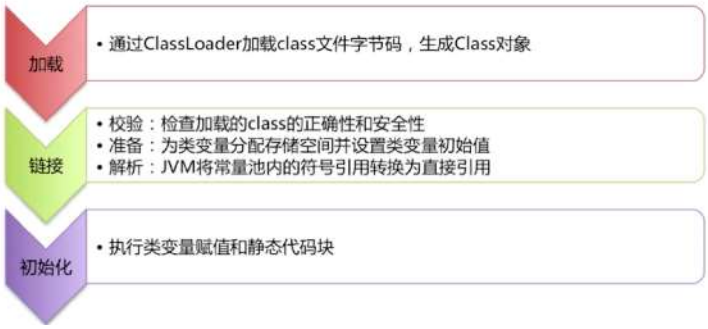
8. 类加载的方式

显示加载之后，需要通过 newInstance 方法来创建类对象的实例，new 支持带参数的构造器，而 newInstance 方法不支持带参数的构造器，需要通过反射调用构造器的 newInstance 方法才能传入参数

- 隐式加载：new
- 显式加载：loadClass，forName等

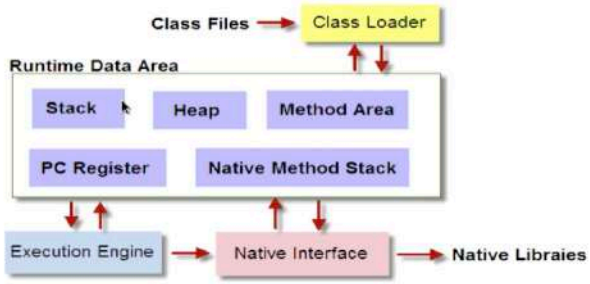
LoadClass 和 forName 的区别？

类的装载过程



- Class.forName得到的class是已经初始化完成的
- Classloder.loadClass得到的class是还没有链接的

9. Java 内存模型



- Class Loader：依据特定格式，加载class文件到内存
- Execution Engine：对命令进行解析
- Native Interface：融合不同开发语言的原生库为Java所用
- Runtime Data Area：JVM内存空间结构模型

内存需要将逻辑地址映射到物理地址，寻址空间分为内核空间和用户空间 ， Java 程序运行在用户空间

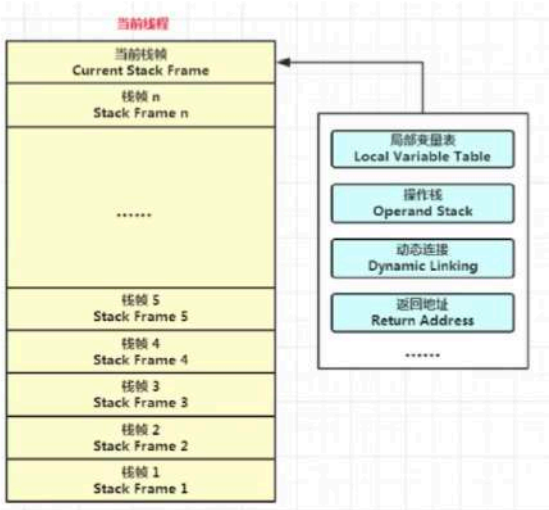
- ◆ 线程私有：程序计数器、虚拟机栈、本地方法栈
- ◆ 线程共享：MetaSpace、Java堆

程序计数器（Program Counter Register）

- ➢ 当前线程所执行的字节码行号指示器(逻辑)
- ➢ 改变计数器的值来选取下一条需要执行的字节码指令
- ➢ 和线程是一对一的关系即“线程私有”
- ➢ 对Java方法计数，如果是Native方法则计数器值为Undefined
- ➢ 不会发生内存泄露

Java虚拟机栈（Stack）

- ➢ Java方法执行的内存模型
- ➢ 包含多个栈帧



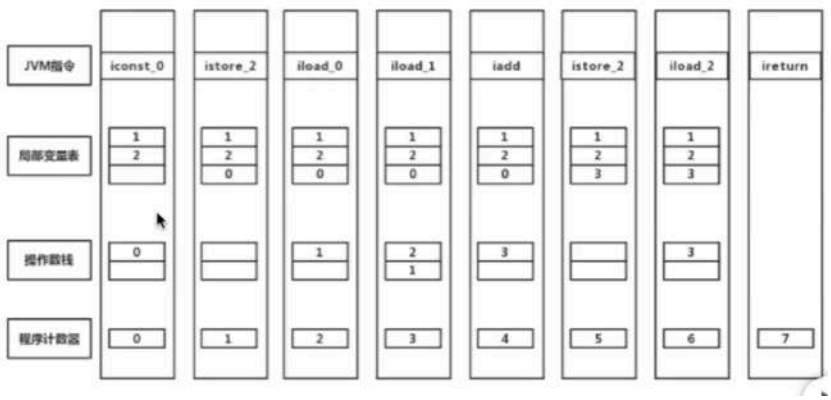
一个方法一个栈帧

局部变量表和操作数栈

- ➢ 局部变量表：包含方法执行过程中的所有变量
- ➢ 操作数栈：入栈、出栈、复制、交换、产生消费变量

执行add(1,2)

```
public static int add(int a, int b)
    int c = 0;
    c = a + b;
    return c;
}
```



本地方法栈

- 与虚拟机栈相似，主要作用于标注了native的方法

10. 递归为什么会引发 java.lang.stackOverFlow 的异常？

递归过深，栈帧数超过虚拟机栈的深度
每调用一次自身，就会生成新的栈帧，栈帧会保存自身方法的状态直至结束
当虚拟机栈可以动态扩展时，就可能出现下面的错误
虚拟机栈过多会引发 java.lang.OutOfMemoryError异常

11. MetaSpace

MetaSpace 元空间是用来存储 class 的信息，包括 class 的方法和 field
元空间和永久代都是方法区的一种实现
Java7 之后字符串常量池已被移动到了 java 堆之中，且元空间替换了永久代

元空间（MetaSpace）与永久代（PermGen）的区别

- 元空间使用本地内存，而永久代使用的是jvm的内存

java.lang.OutOfMemoryError : PermGen space

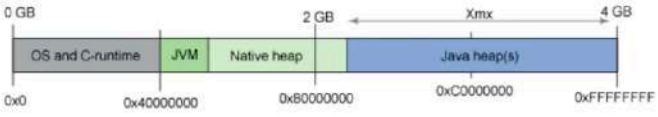
MetaSpace相比PermGen的优势

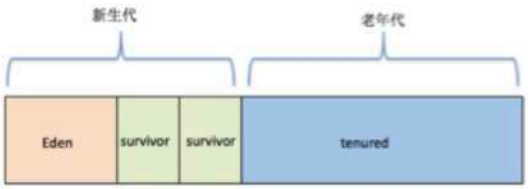
- 字符串常量池存在永久代中，容易出现性能问题和内存溢出
- 类和方法的信息大小难易确定，给永久代的大小指定带来困难
- 永久代会为GC带来不必要的复杂性
- 方便HotSpot与其他JVM如Jrockit的集成

12. Java 堆

Java堆（Heap）

- 对象实例的分配区域
- GC管理的主要区域





13. Java 三大性能调优参数-Xms -Xmx -Xss

java -Xms128m -Xmx128m -Xss256k -jar xxxx.jar

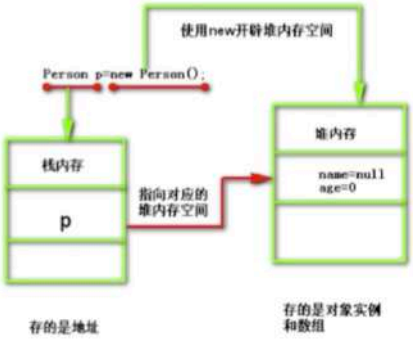
- -Xss：规定了每个线程虚拟机栈（堆栈）的大小
- -Xms：堆的初始值
- -Xmx：堆能达到的最大值

-Xss 影响了此进程中并发线程数的大小
-Xms 和-Xmx 一般设置成相同的大小，因为（java 堆）heap 不够用进行扩容时，会发生内存的抖动，影响程序的稳定性。

14. Java 内存中堆和栈的区别？

静态存储中不允许有可变数据结构的出现，也不允许有嵌套或者递归的结构出现
栈式存储规定在运行中，进入一个程序模块时，必须知道该程序模块所需要的数据区大小才能分配内存
堆式存储比如可变长度串和对象实例

- 静态存储：编译时确定每个数据目标在运行时的存储空间需求
- 栈式存储：数据区需求在编译时未知，运行时模块入口前确定
- 堆式存储：编译时或运行时模块入口都无法确定，动态分配
- 联系：引用对象、数组时，栈里定义变量保存堆中目标的首地址



栈中的引用变量在程序运行到其作用域之外以后就会被释放，而数组和对象在堆中的内存不会被释放。
但数组和对象在没有引用变量的指向之后会变为垃圾，等待垃圾回收机制的处理
在 new 的时候，会在堆内存中创建出相关的对象实例，同时会把把内存的首地址复制给 p，保存在当前线程的虚拟机栈的内存中，通过 p 的地址就可以访问到保存在堆内存中的对象实例

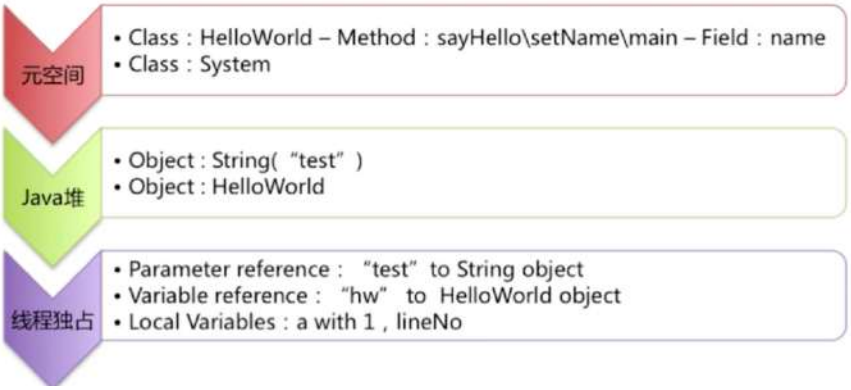
Java内存模型中堆和栈的区别

- 管理方式：栈自动释放，堆需要GC
- 空间大小：栈比堆小
- 碎片相关：栈产生的碎片远小于堆
- 分配方式：栈支持静态和动态分配，而堆仅支持动态分配
- 效率：栈的效率比堆高

但是栈空间与堆空间相比灵活度不够，特别是在动态分配的时候

15. 元空间，堆和线程独占部分之间的联系

```
package com.interview.javabasic.jvm.model;

public class HelloWorld {
    private String name;
    public void sayHello(){
        System.out.println("Hello " + name);
    }

    public void setName(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        int a = 1;
        HelloWorld hw = new HelloWorld();
        hw.setName("test");
        hw.sayHello();
    }
}
```



16. JDK 6 和 6 以后的 intern 方法的对比

```
String s = new String( original: "a");
s.intern();
```

JDK6：当调用 intern 方法时，如果字符串常量池先前已创建出该字符串对象，则返回池中的该字符串的引用。否则，将此字符串对象添加到字符串常量池中，并且返回该字符串对象的引用。

JDK6+：当调用 intern 方法时，如果字符串常量池先前已创建出该字符串对象，则返回池中的该字符串的引用。否则，如果该字符串对象已经存在于 Java 堆中，则将堆中对此对象的引用添加到字符串常量池中，并且返回该引用；如果堆中不存在，则在池中创建该字符串并返回其引用。

字符串常量池在 java7 之后从永久代被转移到 java 堆中，这是因为永久代的内存空间有限，如果频繁调用 intern 方法在池里创建字符串，就会引发 outOfMemoryError 的问题

17. 方法区，元空间，永久代和 java 堆

1.JVM内存模型简介

- 堆——堆是所有线程共享的，主要用来存储对象。其中，堆可分为：*年轻代*和*老年代*两块区域。使用NewRatio参数来设定比例。对于年轻代，一个Eden区和两个Suvivor区，使用参数SuvivorRatio来设定大小；
- Java虚拟机栈/本地方法栈——线程私有的，主要存放*局部变量表*，*操作数栈*，*动态链接*和*方法出口*等；
- 程序计数器——同样是线程私有的，记录当前线程的行号指示器，为线程的切换提供保障；
- 方法区——线程共享的，主要存储*类信息*、*常量池*、*静态变量*、*JIT编译后的代码*等数据。方法区理论上来说是堆的逻辑组成部分；
- 运行时常量池——是方法区的一部分，用于存放编译期生成的各种字面量和符号引用；

2.永久代和方法区的关系

涉及到内存模型时，往往会提到永久代，那么它和方法区又是什么关系呢？《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。 同时大多数用的JVM都是Sun公司的HotSpot。在HotSpot上把GC分代收集扩展至方法区，或者说**使用永久代来实现方法区**。因此，我们得到了结论，永久代是HotSpot的概念，方法区是Java虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现。其他的虚拟机实现并没有永久带这一说法。在1.7之前在(JDK1.2 ~ JDK6)的实现中，HotSpot 使用永久代实现方法区，HotSpot 使用 GC分代来实现方法区内存回收，可以使用如下参数来调节方法区的大小:

3.元空间

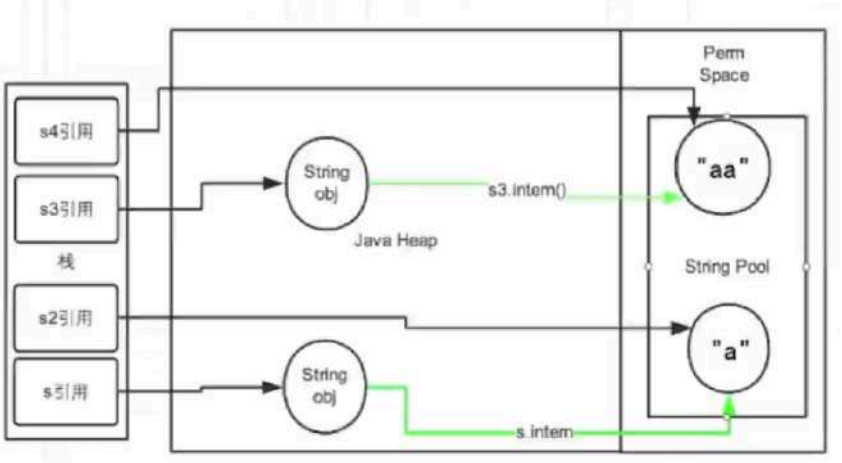
对于Java8， HotSpots取消了永久代，那么是不是也就没有方法区了呢？当然不是，方法区是一个规范，规范没变，它就一直在。那么取代永久代的就是元空间。它可永久代有什么不同的？存储位置不同，永久代物理是是堆的一部分，和新生代，老年代地址是连续的，而元空间属于本地内存；存储内容不同，元空间存储类的元信息，静态变量和常量池等并入堆中。相当于永久代的数据被分到了堆和元空间中。

18. Intern 方法的实例对比

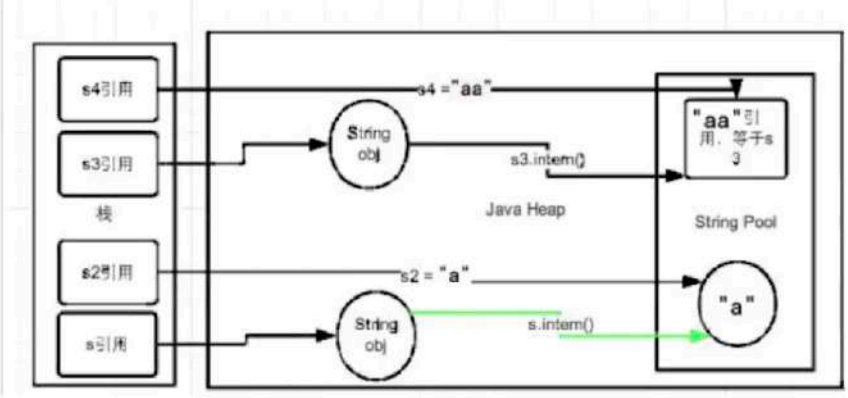
```
public class InternDifference {
    public static void main(String[] args) {
        String s = new String( s: "a");
        s.intern();
        String s2 = "a";
        System.out.println(s == s2);

        String s3 = new String( s: "a") + new String( s: "a");
        s3.intern();
        String s4 = "aa";
        System.out.println(s3 == s4);
    }
}
```

用” ”创建的字符串会直接出现在字符串常量池之中，new String 会在 java 堆之中创建对象，intern 方法会尝试将 java 堆中的字符串对象放入字符串常量池中。

在 java 1.6 之中 false false



在 java 1.7 和以上的版本之中 false true



Java6 之后可以把内存堆中对象的引用放入常量池中，因此”aa”s4 获取到的是 S3 相同的引用

Java 的垃圾回收机制

1. 如何判断对象是否为垃圾

当没有任何引用变量引用的时候，就为垃圾
判断依据有两种算法

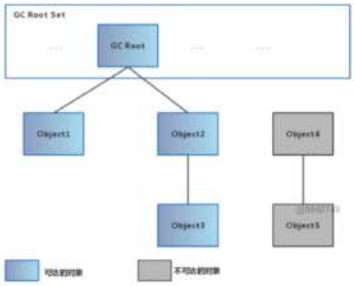
a. 引用计数算法

- 通过判断对象的引用数量来决定对象是否可以被回收
- 每个对象实例都有一个引用计数器，被引用则+1，完成引用则-1
- 任何引用计数为0的对象实例可以被当作垃圾收集
- 优点：执行效率高，程序执行受影响较小
- 缺点：无法检测出循环引用的情况，导致内存泄露

父对象引用子对象，子对象又引用父对象的时候，计数器永远不为 0

b. 可达性分析算法

通过判断对象的引用链是否可达来决定对象是否可以被回收



从 gc root 出发，可以连接到的为引用链

可以作为GC Root的对象

- 虚拟机栈中引用的对象（栈帧中的本地变量表）
- 方法区中的常量引用的对象
- 方法区中的类静态属性引用的对象
- 本地方法栈中JNI（Native方法）的引用对象
- 活跃线程的引用对象

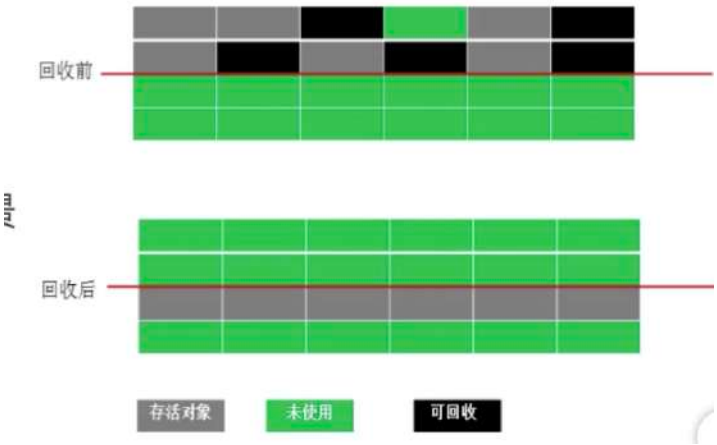
2. 垃圾回收算法

a. 标记-清除算法

先通过可达性分析算法标记可达对象，再清除所有未标记的对象
缺点是容易造成内存碎片化，在之后需要大块的连续的内存空间时不好分配内存，会触发提前的 GC 或者造成 OutOfMemoryError

b. 复制算法

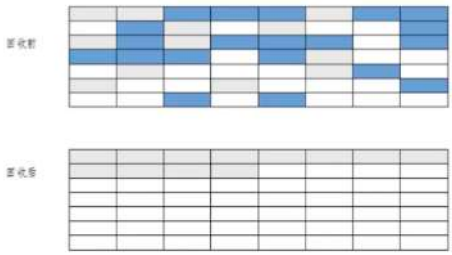
- 分为对象面和空闲面
- 对象在对象面上创建
- 存活的对象被从对象面复制到空闲面
- 将对象面所有对象内存清除
- 解决碎片化问题
- 顺序分配内存，简单高效
- 适用于对象存活率低的场景



适用于年轻代的回收，因为存活率很低，不适用于高存活率的场景

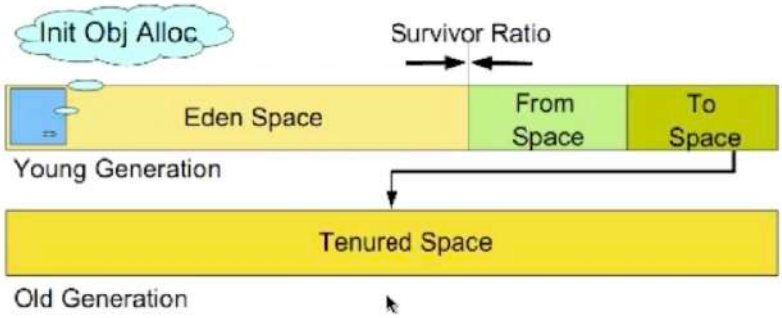
c. 标记-整理算法 适用于老年代

- 标记：从根集合进行扫描，对存活的对象进行标记
- 清除：移动所有存活的对象，且按照内存地址次序依次排列，然后将末端内存地址以后的内存全部回收。
- 避免内存的不连续行
- 不用设置两块内存互换
- 适用于存活率高的场景



d. 分代收集算法

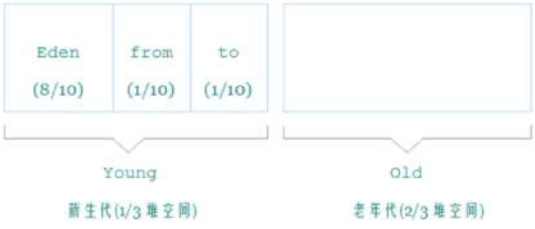
- 垃圾回收算法的组合拳
- 按照对象生命周期的不同划分区域以采用不同的垃圾回收算法
- 目的：提高JVM的回收效率



年轻代采用复制算法，老年代采用标记-整理或清除算法

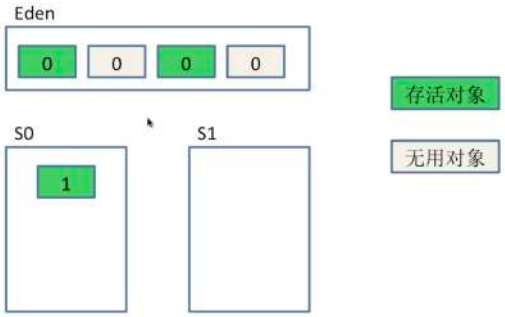
年轻代：尽可能快速地收集掉那些生命周期短的对象

- Eden区
- 两个Survivor区



e. 年轻代的垃圾回收过程

新创建的对象都是存放在 eden 区，如果空间不足也可能放入 survivor 区。大部分新生代中的对象的生命周期都很短。在进行垃圾回收时，会将 eden 和一块 survivor 中仍在生命周期中的对象转入到另一个 survivor 区中，并清除 eden 和之前使用的 survivor 区，如果 survivor 区空间不足，则需要用到老年代的空间



Eden 区的内存空间被占满时，会触发一次 minor GC。将 eden 中的存活数据复制进 survivor 区
年龄在区之间转移之后会加一

周而复始，当年龄到达 15 岁时，会成为老年代，只是通常情况下。
对于某些较大的对象，需要分配一块较大的连续内存空间，Eden 或者 survivor 装不下就会进入老年代

对象如何晋升到老年代

- 经历一定Minor次数依然存活的对象
- Survivor区中存放不下的对象
- 新生成的大对象（-XX:+PretenuerSizeThreshold）

常用的调优参数

- -XX:SurvivorRatio：Eden和Survivor的比值，默认8：1
- -XX:NewRatio：老年代和年轻代内存大小的比例
- -XX:MaxTenuringThreshold：对象从年轻代晋升到老生代经过GC次数的最大阈值

f. 老年代

存放生命周期较长的对象，用标记-清理或整理算法进行垃圾回收
当触发老年代的垃圾回收时，通常也会伴随着新生代的垃圾回收，即对整个堆进行垃圾回收，这就是所谓的 **Full GC**
Full GC 比 Minor GC 执行效率慢很多，但是执行的频率很低
触发 **Full GC** 的条件

- 老年代空间不足
- 永久代空间不足
- CMS GC时出现promotion failed，concurrent mode failure
- Minor GC晋升到老年代的平均大小大于老年代的剩余空间
- 调用System.gc()
- 使用RMI来进行RPC或管理的JDK应用，每小时执行1次Full GC

避免使用较大的对象
Java 8 之后便没有了永久代
Promotion failed 出现在 survivor 空间不足，需要将对象转移到老年代中，而此时老年代空间也不足
Concurrent mode failure 发生在执行 CMS GC 的同时，有对象想放入老年代中，而此时空间不足
记录 minor GC 后放入老年代的对象的平均大小，下次放入老年代时，先检查老年代的剩余空间，若小于平均大小，则直接执行 full GC
System.GC 只是程序员希望在此处执行 Full GC，但实际执行还是由 jvm 来判断的

3. GC 的分类

Minor GC：发生在年轻代中的垃圾回收算法，所采用的是复制算法
Full GC：当触发老年代的垃圾回收时，通常也会伴随着新生代的垃圾回收，即对整个堆进行垃圾回收，这就是所谓的 Full GC

4. 垃圾收集器中出现的两个词

Stop-the-World

- JVM由于要执行GC而停止了应用程序的执行
- 任何一种GC算法中都会发生
- 多数GC优化通过减少Stop-the-world发生的时间来提高程序性能

Safepoint

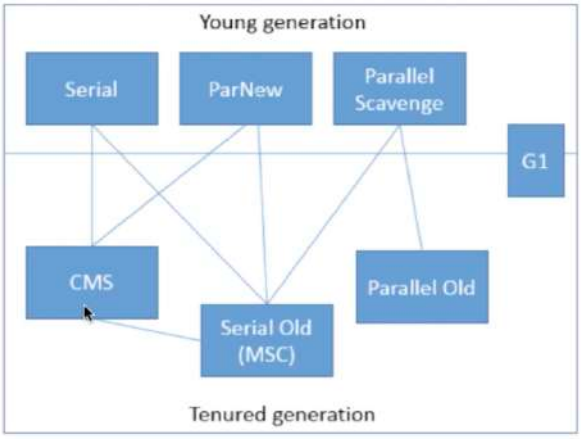
- 分析过程中对象引用关系不会发生变化的点
- 产生Safepoint的地方：方法调用；循环跳转；异常跳转等
- 安全点数量得适中

进行可达性分析算法时，程序应该处于一个快照状态，引用关系不再发生变化。
Safe point 过少会让 GC 等待太长的时间，过多会增加程序运行的负荷

5. Jvm 运行模式

Client 和 server 两种模式
Client 启动快，启动轻量级的 jvm。server 模式启动慢，启动的 jvm 更复杂，但是进入稳定期后比 client 模式效率更高。
通过 java –version 命令可以查看处于哪种状态

6. 常见的垃圾收集器

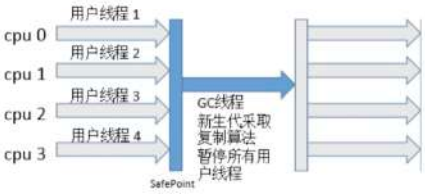


年轻代垃圾收集器

a. Serial 收集器，client 模式下首选的年轻代垃圾收集器

Serial收集器（-XX:+UseSerialGC，复制算法）

- 单线程收集，进行垃圾收集时，必须暂停所有工作线程
- 简单高效，Client模式下默认的年轻代收集器



b. ParNew 垃圾收集器，server 模式下年轻代首选的垃圾收集器

ParNew收集器（-XX:+UseParNewGC，复制算法）

- 多线程收集，其余的行为、特点和Serial收集器一样
- 单核执行效率不如Serial，在多核下执行才有优势

c. Parallel scavenge，server 模式下默认的

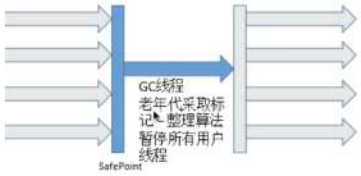
Parallel Scavenge收集器（-XX:+UseParallelGC，复制算法）

- 比起关注用户线程停顿时间，更关注系统的吞吐量
- 在多核下执行才有优势，Server模式下默认的年轻代收集器

老年代垃圾收集器

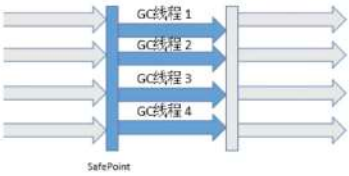
Serial Old收集器（-XX:+UseSerialOldGC，标记-整理算法）

- 单线程收集，进行垃圾收集时，必须暂停所有工作线程
- 简单高效，Client模式下默认的老年代收集器



Parallel Old收集器（-XX：+UseParallelOldGC，标记-整理算法）

- 多线程，吞吐量优先



CMS 垃圾回收线程**几乎**能做到与用户线程同时工作，尽可能缩短了停断时间。有较多存活时间较长的对象，适用于 CMS

CMS收集器（-XX:+UseConcMarkSweepGC，标记-清除算法）

- 初始标记：stop-the-world
- 并发标记：并发追溯标记，程序不会停顿
- 并发预清理：查找执行并发标记阶段从年轻代晋升到老年代的对象
- 重新标记：暂停虚拟机，扫描CMS堆中的剩余对象
- 并发清理：清理垃圾对象，程序不会停顿
- 并发重置：重置CMS收集器的数据结构

因为未被引用的对象对正在执行的用户程序无影响因此可以并发标记
其次 CMS 的显著缺点是标记-清除算法无法避免内存碎片化的问题，需要较大连续空间时，就需要再触发一次 GC

年轻代，老年代通用的垃圾收集器

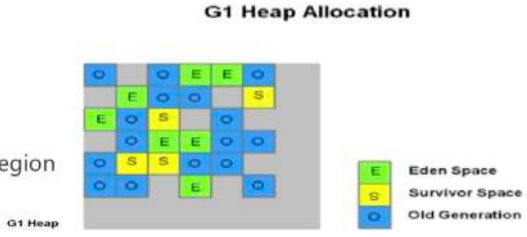
G1收集器（-XX:+UseG1GC, 复制+标记-整理算法）

Garbage First收集器的特点

- 并行和并发
- 分代收集
- 空间整合
- 可预测的停顿

Garbage First收集器

- 将整个Java堆内存划分成多个大小相等的Region
- 年轻代和老年代不再物理隔离



7. GC 面试题

Object的finalize()方法的作用是否与C++的析构函数作用相同

- 与C++的析构函数不同，析构函数调用确定，而它的是不确定的
- 将未被引用的对象放置于F-Queue队列
- 方法执行随时可能会被终止
- 给予对象最后一次重生的机会

Java中的强引用，软引用，弱引用，虚引用有什么用

强引用（Strong Reference）

- 最普遍的引用：Object obj=new Object()
- 抛出OutOfMemoryError终止程序也不会回收具有强引用的对象
- 通过将对象设置为null来弱化引用，使其被回收

软引用（Soft Reference）

- 对象处在有用但非必须的状态
- 只有当内存空间不足时，GC会回收该引用的对象的内存
- 可以用来实现高速缓存

```
String str=new String( original: "abc"); // 强引用
SoftReference<String> softRef=new SoftReference<String>(str); // 软引用
```

弱引用（Weak Reference）

- 非必须的对象，比软引用更弱一些
- GC时会被回
- 被回收的概率也不大，因为GC线程优先级比较低
- 适用于引用偶尔被使用且不影响垃圾收集的对象

```
String str=new String( original: "abc");
WeakReference<String> abcWeakRef = new WeakReference<String>(str);
```

虚引用（PhantomReference）

- 不会决定对象的生命周期
- 任何时候都可能被垃圾收集器回收
- 跟踪对象被垃圾收集器回收的活动，起哨兵作用
- 必须和引用队列ReferenceQueue联合使用

```
String str=new String( original: "abc");
ReferenceQueue queue = new ReferenceQueue();
PhantomReference ref = new PhantomReference(str, queue);
```

Gc 在回收对象时，如果发现一个对象有虚引用，就首先把该对象的虚引用加入到与之关联的引用队列中，程序可以通过判断队列中是否有虚引用来判断对象是否被垃圾回收器回收，起到了哨兵的作用

强引用 > 软引用 > 弱引用 > 虚引用

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM 停止运行时终止
软引用	在内存不足时	对象缓存	内存不足时终止
弱引用	在垃圾回收时	对象缓存	gc 运行后终止
虚引用	Unknown	标记、哨兵	Unknown

引用队列（ReferenceQueue）

- 无实际存储结构，存储逻辑依赖于内部节点之间的关系来表达
- 存储关联的且被GC的软引用，弱引用以及虚引用

JAVA 多线程与并发

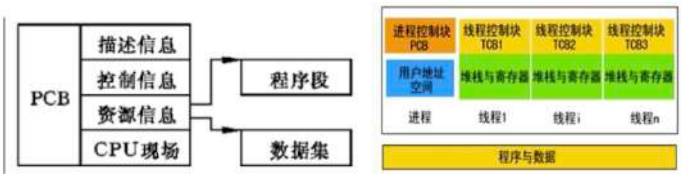
1. 进程与线程的区别

进程和线程的由来

进程是资源分配的最小单位，线程是CPU调度的最小单位

- 所有与进程相关的资源，都被记录在PCB中
- 进程是抢占处理机的调度单位；线程属于某个进程，共享其资源
- 线程只由堆栈寄存器、程序计数器和TCB组成

- 线程不能看做独立应用，而进程可看做独立应用
- 进程有独立的地址空间，相互不影响，线程只是进程的不同执行路径
- 线程没有独立的地址空间，多进程的程序比多线程程序健壮
- 进程的切换比线程的切换开销大

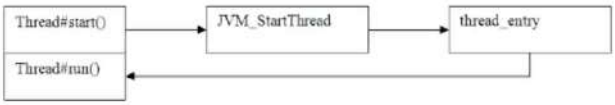


什么是进程，线程
计算机组成结构
Linux 的用户态与内核态如何转换，为什么要转换？
什么是系统中断，以及内核态的多线程是如何通过轻量级线程实现？

2. Thread 中 start 和 run 的区别

```
public class ThreadTest {
    private static void attack() {
        System.out.println("Fight");
        System.out.println("Current Thread is : " + Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        Thread t = new Thread(){
            public void run(){
                attack();
            }
        };
        System.out.println("current main thread is : " + Thread.currentThread().getName());
        t.start();
    }
}
```

用 run 打印出的是主线程，用 start 打印出的是一个新的子线程
Start 方法最重要的是里面的 start0 方法



- 调用start()方法会创建一个新的子线程并启动
- run()方法只是Thread的一个普通方法的调用

3. Thread 和 runnable 的区别

- Thread是实现了Runnable接口的类，使得run支持多线程
- 因类的单一继承原则，推荐多使用Runnable接口

Runnable 接口中只有一个 run 方法，开启线程并发执行任务还是通过 thread 类来实现的。
继承 thread 类，重写 run 方法后，可以通过 new Thread 和 start 来开启新线程
实现 runnable 接口，new Runnable 的对象没有 start 方法，需要 new Thread 传入 runnable 对象，再 start 开启 Thread。因为 java 单一继承，但可以实现多个接口，推荐使用 runnable 接口

```
public class MyThread extends Thread {
    private String name;
    public MyThread(String name){
        this.name = name;
    }
    @Override
    public void run(){
        for(int i = 0 ; i < 10 ; i ++){
            System.out.println("Thread start : " + this.name
        }
    }
}
```

```
public class ThreadDemo {
    public static void main(String[] args) {
        MyThread mt1 = new MyThread( name: "Thread1");
        MyThread mt2 = new MyThread( name: "Thread2");
        MyThread mt3 = new MyThread( name: "Thread3");
        mt1.start();
        mt2.start();
        mt3.start();
    }
}
```

```java
public class MyRunnable implements Runnable {
    private String name;
    public MyRunnable(String name){
        this.name = name;
    }
    @Override
    public void run(){
        for(int i = 0 ; i < 10 ; i ++){
            System.out.println("Thread start : " + this.name
        }
    }
}
```

```java
public class RunnableDemo {
    public static void main(String[] args) {
        MyRunnable mr1 = new MyRunnable( name: "Runnable1");
        MyRunnable mr2 = new MyRunnable( name: "Runnable2");
        MyRunnable mr3 = new MyRunnable( name: "Runnable3");
        Thread t1 = new Thread(mr1);
        Thread t2 = new Thread(mr2);
        Thread t3 = new Thread(mr3);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

4. 如何给 run 方法传入参数

实现的方式主要有三种

- 构造函数传参
- 成员变量传参
- 回调函数传参

5. 如何处理线程的返回值

1. 让主线程循环等待，直到目标子线程返回值为止
   实现简单，但需要自己实现循环等待的逻辑，而且需要等待的目标多时，代码臃肿
   而且无法精准控制循环过程中的等待时间
2. 使用 Thread 类的 join 方法阻塞当前线程，等待子线程
   可以精准控制等待时间，但是粒度不够细

```java
public static void main(String[] args) throws InterruptedException
    CycleWait cw = new CycleWait();
    Thread t = new Thread(cw);
    t.start();
      while (cw.value == null){
          Thread.currentThread().sleep(100);
      }
    t.join();
    System.out.println("value : " + cw.value);
}
```

3. **通过 callable 接口实现，通过 FutureTask 或者线程池获取**
   a. FutureTask 构造函数可以接收 callable 的实现类的实例
   通过 isDone 方法判断，callable 实例中的 call 方法是否执行完成
   Get 方法，阻塞当前线程，直到 call 方法执行完成，能够精准获取到子线程执行完毕的返回值
   还有加入 timeout 参数的 get 方法，超过时间 call 方法仍未执行完毕就会抛出 timeOutException
   FutureTask 继承了 runnableFuture，父类实现了 runnable 接口，因此可传入 thread 构造函数

```java
public class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception{
        String value="test";
        System.out.println("Ready to work");
        Thread.currentThread().sleep( millis: 5000);
        System.out.println("task done");
        return  value;
    }
}
```

```java
public class FutureTaskDemo {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        FutureTask<String> task = new FutureTask<String>(new MyCallable());
        new Thread(task).start();
        if(!task.isDone()){
            System.out.println("task has not finished, please wait!");
        }
        System.out.println("task return: " + task.get());

    }
}
```

b. 通过线程池来实现

使用线程池之后一定要记得关闭

```
public class ThreadPoolDemo {
    public static void main(String[] args) {
        ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
        Future<String> future = newCachedThreadPool.submit(new MyCallable());
        if(!future.isDone()){
            System.out.println("task has not finished, please wait!");
        }
        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        } finally {
            newCachedThreadPool.shutdown();
        }
    }
}
```

用线程池可以提交多个实现了 callable 的类让线程池并发的处理结果，方便对 callable 类统一管理

6. 线程的状态

public static enum Thread.State
extends Enum<Thread.State>

A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states

六个状态

- 新建(New)：创建后尚未启动的线程的状态
- 运行(Runnable):包含Running和Ready
- 无限期等待(Waiting):不会被分配CPU执行时间，需要显式被唤醒

没有设置 **Timeout** 参数的 **Object.wait()**方法。
没有设置 **Timeout** 参数的 **Thread.join()**方法。
LockSupport.park()方法。

- 限期等待(Timed Waiting):在一定时间后会由系统自动唤醒

Thread.sleep()方法。
设置了 **Timeout** 参数的 **Object.wait()**方法。
设置了 **Timeout** 参数的 **Thread.join()**方法。
LockSupport.parkNanos()方法。
LockSupport.parkUntil()方法。

- 阻塞(Blocked):等待获取排它锁
- 结束(Terminated):已终止线程的状态，线程已经结束执行

7. Sleep 和 wait

- sleep是Thread类的方法，wait是Object类中定义的方法
- sleep()方法可以在任何地方使用
- wait()方法只能在synchronized方法或synchronized块中使用

- Thread.sleep只会让出CPU，不会导致锁行为的改变
- Object.wait不仅让出CPU，还会释放已经占有的同步资源锁

```
public class WaitSleepDemo {
    public static void main(String[] args) {
        final Object lock = new Object();
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("thread A is waiting to get lock");
                synchronized (lock){
                    try {
                        System.out.println("thread A get lock");
                        Thread.sleep( millis: 20);
                        System.out.println("thread A do wait method");
                        lock.wait( timeoutMillis: 1000);
                        System.out.println("thread A is done");
                    } catch (InterruptedException e){
                        e.printStackTrace();
                    }
                }
            }
        }).start();
        try{
            Thread.sleep( millis: 10);
        } catch (InterruptedException e){
            e.printStackTrace();
        }

        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("thread B is waiting to get lock");
                synchronized (lock){
                    try {
                        System.out.println("thread B get lock");
                        System.out.println("thread B is sleeping 10 ms");
                        Thread.sleep( millis: 10);
                        System.out.println("thread B is done");
                    } catch (InterruptedException e){
                        e.printStackTrace();
                    }
                }
            }
        }).start();
```

```
WaitSleepDemo
  .WaitSleepDemo
thread A is waiting to get lock
thread A get lock
thread B is waiting to get lock
thread A do wait method
thread B get lock
thread B is sleeping 10 ms
thread B is done
thread A is done
```

Wait 需要写在 synchronize 中的原因是必须要先获取锁才能释放锁
Sleep 是对线程使用的，而 wait 是对 object 使用的

8. Notify 和 notifyall

锁池 EntryList 和等待池 WaitSet

假设线程 A 已经拥有了某个对象（不是类）的锁，而其它线程 B、C 想要调用这个对象的某个 synchronized 方法（或者块），由于B、C 线程在进入对象的 synchronized 方法（或者块）之前必须先获得该对象锁的拥有权，而恰巧该对象的锁目前正被线程 A 所占用，此时 B、C 线程就会被阻塞，进入一个地方去等待锁的释放，这个地方便是该对象的锁池

假设线程 A 调用了某个对象的 wait() 方法，线程 A 就会释放该对象的锁，同时线程 A 就进入到了该对象的等待池中，进入到等待池中的线程不会去竞争该对象的锁。

锁池中的线程会去竞争该对象的锁，而等待池中的线程不会竞争锁。当处于等待池的中的线程被 notify 方法唤醒之后，就会进入到锁池中，竞争锁

- notifyAll 会让所有处于等待池的线程全部进入锁池去竞争获取锁的机会
- notify 只会随机选取一个处于等待池中的线程进入锁池去竞争获取锁的机会。

```
private volatile boolean go = false;
```

被 volatile 修饰的变量意味着，当多个线程尝试改动变量时如果 a 已经改动了那么线程 bcd 可以立刻看到。

9. Yield

当调用Thread.yield()函数时，会给线程调度器一个当前线程愿意让出CPU使用的暗示，但是线程调度器可能会忽略这个暗示。

Yield 对锁的行为没有任何影响，并不会使得当前线程释放已经占用的锁

10. 如何中断线程

已经被抛弃的方法

- 通过调用stop()方法停止线程
- 通过调用suspend()和resume()方法

目前使用的方法

- 调用interrupt()，通知线程应该中断了

  ① 如果线程处于被阻塞状态，那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常。

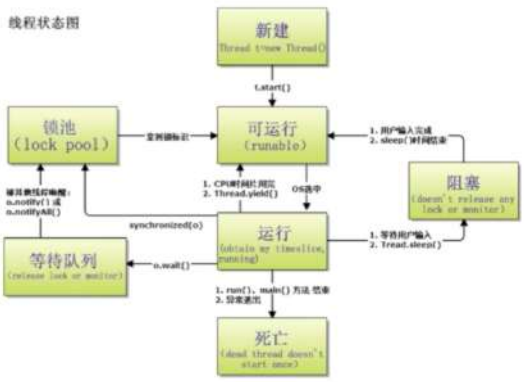
  ② 如果线程处于正常活动状态，那么会将该线程的中断标志设置为true。被设置中断标志的线程将继续正常运行，不受影响。

- 需要被调用的线程配合中断

  ① 在正常运行任务时，经常检查本线程的中断标志位，如果被设置了中断标志就自行停止线程。

  ② 如果线程处于正常活动状态，那么会将该线程的中断标志设置为true。被设置中断标志的线程将继续正常运行，不受影响。

被设置了中断标记的线程并不是立刻停止



11. Synchronize

线程安全问题的主要诱因

- 存在共享数据(也称临界资源)
- 存在多条线程共同操作这些共享数据

解决问题的根本方法：

同一时刻有且只有一个线程在操作共享数据，其他线程必须等到该线程处理完数据后再对共享数据进行操作

互斥锁的特性

互斥性：即在同一时间只允许一个线程持有某个对象锁，通过这种特性来实现多线程的协调机制，这样在同一时间只有一个线程对需要同步的代码块(复合操作)进行访问。互斥性也称为操作的原子性。

可见性：必须确保在锁被释放之前，对共享变量所做的修改，对于随后获得该锁的另一个线程是可见的（即在获得锁时应获得最新共享变量的值），否则另一个线程可能是在本地缓存的某个副本上继续操作，从而引起不一致。

Synchronize 锁住的不是代码而是对象。

Java 堆是对线程共享的，因此需要加入合适的锁

根据获取的锁的分类：获取对象锁和获取类锁

获取对象锁的两种用法

1. 同步代码块（synchronized (this)，synchronized (类实例对象)），锁是小括号 () 中的实例对象。
2. 同步非静态方法（synchronized method），锁是当前对象的实例对象。

一个类中的非静态同步方法和同步代码块共用的是同一个同步锁对象。

但如果每个线程中传入的都是不同的 runnable 对象，即使是同步方法和同步代码块也会异步执行

同一个类的不同对象的对象锁之间是互不干扰的

获取类锁的两种用法

1. 同步代码块（synchronized (类.class)），锁是小括号 () 中的类对象（Class 对象）。
2. 同步静态方法（synchronized static method），锁是当前对象的类对象（Class 对象）。

也是通过对象锁实现的，即类的 class 的对象锁，每个类只有一个 class 对象

不同的对象获取的都是同一把类锁

但是类锁和对象锁之间是互不干扰的

对象锁和类锁的总结

1. 有线程访问对象的同步代码块时，另外的线程可以访问该对象的非同步代码块；
2. 若锁住的是同一个对象，一个线程在访问对象的同步代码块时，另一个访问对象的同步代码块的线程会被阻塞；
3. 若锁住的是同一个对象，一个线程在访问对象的同步方法时，另一个访问对象同步方法的线程会被阻塞；
4. 若锁住的是同一个对象，一个线程在访问对象的同步代码块时，另一个访问对象同步方法的线程会被阻塞，反之亦然；
5. 同一个类的不同对象的对象锁互不干扰；
6. 类锁由于也是一种特殊的对象锁，因此表现和上述1，2，3，4一致，而由于一个类只有一把对象锁，所以同一个类的不同对象使用类锁将会是同步的；
7. 类锁和对象锁互不干扰。

12. Synchronize 的底层实现原理

对象在内存中的布局

- 对象头
- 实例数据
- 对齐填充

对象头的结构

虚拟机位数	头对象结构	说明
32/64 bit	Mark Word	默认存储对象的hashCode，分代年龄，锁类型，锁标志位等信息
32/64 bit	Class Metadata Address	类型指针指向对象的类元数据，JVM通过这个指针确定该对象是哪个类的数据

Mark word 存储运行时的信息，是实施轻量级锁和偏向锁的关键

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象 hashcode、对象分代年龄				01
轻量级锁	指向锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC 标记	空，不需要记录信息				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

Markword 的大小不固定，会随着具体情况复用

重量锁就是 synchronize 锁

每一个对象都有一个 monitor 与之关联，monitor 是对象一出生就有的天然的内部锁

Monitor 有多种使用方式，可以与对象一同创建和销毁，也可以在线程试图获得对象的同步锁的时候创建，但一个 monitor 被某个线程持有后就处于锁定状态。

13.