# Java 基础知识手册

## 面向对象

一、开闭原则：

这一条放在第一位来理解，它的含义是对扩展开放，对修改关闭。解释一下就是，我们写完的代码，不能因为需求变化就修改。我们可以通过新增代码的方式来解决变化的需求。

当然，这是一种理想的状态，在现实中，我们要尽量的缩小这种修改。

再解释一下这条原则的意义所在，我们采用逆向思维方式来想。如果每次需求变动都去修改原有的代码，那原有的代码就存在被修改错误的风险，当然这其中存在有意和无意的修改，都会导致原有正常运行的功能失效的风险，这样很有可能会展开可怕的蝴蝶效应，使维护工作剧增。

说到底，开闭原则除了表面上的可扩展性强以外，在企业中更看重的是维护成本。

所以，开闭原则是设计模式的第一大原则，它的潜台词是：控制需求变动风险，缩小维护成本。

以下几种原则，都是为此原则服务的。

二、里氏替换选择：

此原则的含义是子类可以在任何地方替换它的父类。解释一下，这是多态的前提，我们后面很多所谓的灵活，都是不改变声明类型的情况下，改变实例化类来完成的需求变更。当然，继承的特性看似天然就满足这个条件。但这里更注重的是继承的应用问题，我们必须保证我们的子类和父类划分是精准的。

里氏替换原则的潜台词是：尽量使用精准的抽象类或者接口。

三、单一职责原则：

单一职责的含义是：类的职责单一，引起类变化的原因单一。解释一下，这也是灵活的前提，如果我们把类拆分成最小的职能单位，那组合与复用就简单的多了，如果一个类做的事情太多，在组合的时候，必然会产生不必要的方法出现，这实际上是一种污染。

举个例子，我们在绘制图案的时候，用“点”组成图和用“直线”组成图，哪个更灵活呢？一定是“点”，它可以绘制任何图形，而直线只能绘制带有直线条的图案，它起码无法画圆。

单一职责的潜台词是：拆分到最小单位，解决复用和组合问题。

四、接口隔离原则：

接口隔离原则可以说是单一职责的必要手段，它的含义是尽量使用职能单一的接口，而不使用职能复杂、全面的接口。很好理解，接口是为了让子类实现的，如果子类想达到职能单一，那么接口也必须满足职能单一。

相反，如果接口融合了多个不相关的方法，那它的子类就被迫要实现所有方法，尽管有些方法是根本用不到的。这就是接口污染。

接口隔离原则的潜台词是：拆分，从接口开始。

五、依赖倒置原则：

想要理解依赖倒置原则，必须先理解传统的解决方案。面相对象的初期的程序，被调用者依赖于调用者。也就是调用者决定被调用者有什么方法，有什么样的实现方式，这种结构在需求变更的时候，会付出很大的代价，甚至推翻重写。

依赖倒置原则就是要求调用者和被调用者都依赖抽象，这样两者没有直接的关联和接触，在变动的时候，一方的变动不会影响另一方的变动。

其实，依赖倒置和前面的原则是相辅相成的，都强调了抽象的重要性。

依赖倒置的潜台词是：面向抽象编程，解耦调用和被调用者。

六、迪米特原则：

迪米特原则要求尽量的封装，尽量的独立，尽量的使用低级别的访问修饰符。这是封装特性的典型体现。

一个类如果暴露太多私用的方法和字段，会让调用者很茫然。并且会给类造成不必要的判断代码。所以，我们使用尽量低的访问修饰符，让外界不知道我们的内部。这也是面向对象的基本思路。这是迪米特原则的一个特性，无法了解类更多的私有信息。

另外，迪米特原则要求类之间的直接联系尽量的少，两个类的访问，通过第三个中介类来实现。

迪米特原则的潜台词是：不和陌生人说话，有事去中介。

七、组合/聚合复用原则：

此原则的含义是，如果只是达到代码复用的目的，尽量使用组合与聚合，而不是继承。这里需要解释一下，组合聚合只是引用其他的类的方法，而不会受引用的类的继承而改变血统。

继承的耦合性更大，比如一个父类后来添加实现一个接口或者去掉一个接口，那子类可能会遭到毁灭性的编译错误，但如果只是组合聚合，只是引用类的方法，就不会有这种巨大的风险，同时也实现了复用。

组合聚合复用原则的潜台词是：我只是用你的方法，我们不一定是同类。

# 计算机网络

## 基础

### OSI 体系结构

<table>
<tr><th>OSI 体系结构<br>(7层)</th><th></th><th>TCP / IP 体系结构<br>(4层)</th><th></th><th>五层体系结构<br>(5层)</th></tr>
<tr><td>7. 应用层</td><td rowspan="3"></td><td rowspan="3">4. 应用层<br>(HTTP)</td><td rowspan="3"></td><td rowspan="3">5. 应用层</td></tr>
<tr><td>6. 表示层</td></tr>
<tr><td>5. 会话层</td></tr>
<tr><td>4. 传输层</td><td></td><td>3. 运输层<br>(TCP、UDP)</td><td></td><td>4. 运输层</td></tr>
<tr><td>3. 网络层</td><td></td><td>2. 网际层<br>(IP)</td><td></td><td>3. 网络层</td></tr>
<tr><td>2. 链路层</td><td rowspan="2"></td><td rowspan="2">1. 网络接口层</td><td rowspan="2"></td><td>2. 链路层</td></tr>
<tr><td>1. 物理层</td><td>1. 物理层</td></tr>
</table>

| 层级 | 作用 | 传输单位 | 功能 | 具体协议 |
| --- | --- | --- | --- | --- |
| 1. 物理层 | 透明传输比特流 | 比特 | 在物理媒体上为 数据端设备 透明传输原始比特流<br>• 传输信息所利用的物理媒体（如光缆）不属于物理层协议，而是在物理层协议下面<br>• 传输比特流时可能发生错误，由 数据链路层 校验 | （接口标准）EIA-232C、CCITT的X.21 |
| 2. 数据链路层 | • 将网络层传下来的IP数据报组装成帧<br>• 检测 & 校正物理层传输介质上产生的传输差错<br>（使链路对网络层显现为一条无差错、可靠的数据传输线路） | 帧 | 组帧、差错控制、流量控制和传输管理<br>• 不保证上层的数据包有没有丢失 & 重复，因一旦发现错误帧，则丢弃，以避免浪费网络资源）<br>• 数据链路层在广播式网络通过 介质访问子层 控制共享信道访问 | SDLC、HDLC、PPP、STP、帧中继 |
| 3. 网络层 | 为不同主机提供通信服务：网络层的分组数据从源端传到目的端 | 数据报 | • 封装数据成分组 / 包、路由选择<br>• 流量控制、拥塞控制、差错控制 & 网际互连 | • IP协议：提供网络结点之间的报文传送服务<br>• ARP协议：实现IP地址向物理地址的映射<br>• RARP协议：实现物理地址向IP地址的映射<br>• ICMP协议：探测 & 报告传输中产生的错误<br>• IGMP协议：管理多播组测成员关系<br>• 其余：IPX、OSPF |
| 4. 传输层 | 为不同主机中的进程间提供通信服务 | 报文段（TCP）<br>/<br>用户数据报（UDP） | 为端到端的连接 提供可靠的传输服务<br>为端到端的连接提供流量控制、差错控制、数据传输管理服务 | • TCP协议：提供用户间面向连接、可靠的报文传输服务<br>• UDP协议：提供用户间无连接、不可靠的报文传输服务 |
| 5. 会话层 | 允许不同主机上各进程之间的会话 | / | • 利用传输层提供的端对端服务，向表示层提供增值服务<br>• 负责管理会话进程，包括：建立、管理以及终止进程，实现数据同步 | / |
| 6. 表示层 | 处理在两个通信系统中交换信息的表示方式 | / | • 采用抽象的标准方法定义数据结构<br>• 采用标准的编码形式 | / |
| 7. 应用层 | 为特性类型的网络应用提供访问OSI环境手段 | / | / | • HTTP协议：提供Internet网浏览服务<br>• DNS协议：负责域名和IP地址的映射<br>• SMTP协议：提供简单的电子邮件发送服务<br>• POP协议：提供对邮箱服务器进行远程存取邮件的服务，与此功能类似的还有IMAP协议<br>• FTP协议：提供应用级文件传输服务<br>• SMB协议：提供应用级文件共享传输服务<br>• Telnet协议：提供远程登录服务（明文传输）<br>• SSH协议：提供远程登录服务（加密） |

# HTTP 协议：

## 请求行

| 类型 | 作用 | 具体介绍 | 备注 |
| --- | --- | --- | --- |
| 请求方法 | 定义 对请求对象的操作 | • 请求方法有8种：GET、POST、HEAD、DELETE、PUT、DETELTE、TRACE、CONNECT、OPTION<br>• 最常用的是 GET、POST、HEAD<br>• 若服务器 = RESTful接口，则一般会用到GET、POST、DELETE、PUT | • GET　请求读取“URL标志的信息“的信息<br>• POST　为服务器添加信息<br>• HEAD　请求读取”URL标志信息的首部“信息<br>• PUT　为指定的URL下添加（存储）一个文档<br>• DELETE　删除指定URL所标志的信息<br>• TRACE　用于进行环回测试的请求报文<br>• CONNECT　用于代理服务器<br>• OPTION　请求“选项”的信息 |
| 请求路径 | URL中的请求地址部分 | • 若URL = http://www.baidu.com/，则请求路径 = /<br>• 若URL = http://www.weibo.com/288/home，则请求路径 = /288/home | URL的简介<br>• 定义：Uniform Resoure Locator，统一资源定位符，一种自愿位置的抽象唯一识别方法<br>• 作用：表示资源位置 & 访问资源的方法<br>• 组成：<协议>：//<主机>：<端口>/<路径><br>a. 协议：应用层通信协议，如在HTTP协议下则是：HTTP：//<主机>：<端口>..<br>b. 主机：请求资源所在主机的域名<br>c. 端口 & 路径有时可省略（HTTP默认端口号 = 80） |
| 协议版本 | 定义HTTP的版本号 | 常用版本：HTTP/1.0、HTTP/1.1、HTTP/2.0 | |

## 请求头

| 名称 | 作用 |
| --- | --- |
| Content-Type | 请求体/响应体的类型，如：text/plain、application/json |
| Accept | 说明接收的类型，可以多个值，用`,`(半角逗号)分开 |
| Content-Length | 请求体/响应体的长度，单位字节 |
| Content-Encoding | 请求体/响应体的编码格式，如gzip,deflate |
| Accept-Encoding | 告知对方我方接受的Content-Encoding |
| ETag | 给当前资源的标识，和`Last-Modified`、`If-None-Match`、`If-Modified-Since`配合，用于缓存控制 |
| Cache-Control | 取值为一般为`no-cache`或`max-age=XX`，XX为个整数，表示该资源缓存有效期(秒) |

## HTTP1.0 和 1.1 区别

- 在浏览器中输入 url 地址 ->> 显示主页的过程

| 过程 | 使用的协议 |
| --- | --- |
| 1. 浏览器查找域名的IP地址<br>（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IP |
| 2. 浏览器向web服务器发送一个HTTP请求<br>（cookies会随着请求发送给服务器） | • TCP：与服务器建立TCP连接<br>• IP：建立TCP协议时，需要发送数据，发送数据在网络层使用IP协议<br>• OPSF：IP数据包在路由器之间，路由选择使用OPSF协议<br>• ARP：路由器在与服务器通信时，需要将ip地址转换为MAC地址，需要使用ARP协议<br>• HTTP：在TCP建立完成后，使用HTTP协议访问网页 |
| 3. 服务器处理请求<br>（请求 处理请求 & 它的参数、cookies、生成一个HTML响应） | |
| 4. 服务器发回一个HTML响应 | |
| 5. 浏览器开始显示HTML | |

## Ping 的过程

| 角色 | 流程 | 具体描述 |
| --- | --- | --- |
| 源主机（PC1） | 1. 发起PING请求 | PC1 在应用层发起个目标IP为192.168.1.2的Ping请求<br>（直接使用网际层的ICMP协议，不经过传输层） |
| | 2. 网际层封装数据 | 1. 网际层接收来处上层的数据后，根据ICMP协议封装成数据包<br>（封装前：添加PC1的IP为源IP、PC2的IP为目标IP）<br>2. 下传到网络接口层 |
| | 3. 网络接口层封装数据 & 发送 | 1. 封装信息：源MAC地址 = PC1的MAC地址、目标MAC地址 = 查询自己的ARP缓存表获取 / 若无，则通过发送ARP广播报文获取<br>（ARP报文：源MAC地址 、源IP地址均是PC1的，所要请求的是PC2的IP对应的MAC地址）<br>2. 封装到数据帧后。发给下层进行网络传输 |
| 目的主机（PC2） | 4. 接收数据 | 1. PC2接收该数据帧后，在网络接口层查看目标MAC地址是否指向自己<br>2. 若是，PC2则将帧头去掉，向上层传输 |
| | 5. 网际层接收数据 | 1. PC2网际层接收到这个信息包，查看包头，发现目标IP和自己匹配，则解封装<br>2. 将数据向上层传输 |
| | 6. 传输层接收数据 | 1. 传输层接收来自下层的Ping请求的UDP报文<br>2. 去掉UDP报头，向应用层传送 |
| | 7. 应用层接收数据 & 应答 | 应用层收到Ping请求后，发送一个Ping回应报文给PC1 |

发送ARP广播报文获取MAC地址过程

| | |
| --- | --- |
| 1. 目的主机接收广播 | 1. PC2收到ARP广播后，进行解封装<br>2. 发现所请求的MAC地址是自己的，则PC2将PC1的MAC地址写入ARP缓存表中<br>3. 向PC1发送一个 ARP应答单播<br>（目标IP、MAC地址 – PC1、源IP、源MAC–PC2） |
| 2. 源主机获取目的主机的Mac地址 | 1. PC1接收到PC2的ARP应答报文后，将PC2的MAC地址存入ARP缓存中<br>2. 将PC2的MAC地址作为目标地址封装到数据帧中。发给下层进行网络传输。 |

## Cookie & Session 的区别

| 类型 | 定义 | 作用 | 应用场景 | 原理 | 具体使用 |
| --- | --- | --- | --- | --- | --- |
| Cookie | 一种客户端机制 | 纪录下在某个网站上输入的内容 & 选择<br>• 下次打开同一个网站时，WEB 服务器会先看看有无上次留下的 Cookie 资料；<br>• 若有，则依据 Cookie里的内容来判断使用者，显示特定的网页内容 | 提供个人化服务 | • 通过扩展HTTP协议 实现<br>• 服务器通过在HTTP的响应头中加上一行特殊的指示以提示 | • cookie的主要内容：名字、值、过期时间、路径 & 域<br>• 路径与域一起构成cookie的作用范围<br>a. 若不设置过期时间，则表示该cookie的生命期 = 浏览器会话期间<br>b. 若关闭浏览器窗口，cookie就消失<br>c. 这种生命期为浏览器会话期的cookie 称为：会话cookie |
| Session | 一种服务器端的机制 | • 使用散列表的结构来保存信息<br>• 当程序需为某个客户端的请求创建一个session时，服务器首先检查该客户端的请求里是否已包含一个session标识（即session id）<br>• 若已包含，则把其检索出来使用；若不包含，则为此客户端创建一个session & 生成一个与此session相关联的session标识（即session id）<br>（session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串；该session id将被在本次响应中返回给客户端保存；保存方式可采用cookie）<br>• 在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。一般这个cookie的名字都是类似于SEEESIONID | | | |

| 类型 | 作用域<br>（数据存放区域） | 性能 | | | 应用场景 |
| --- | --- | --- | --- | --- | --- |
| | | 存储量 | 安全性 | 损耗 | |
| Cookie | 客户端浏览器<br>（一种客户端机制） | 少<br>• 单个cookie保存的数据不超过4K<br>• 浏览器都限制一个站点最多保存20个cookie | 不安全<br>（可分析存放在本地的COOKIE & 进行COOKIE欺骗） | 低 | 存放设备配置信息 |
| Session | 服务器<br>（一种服务器端机制） | 多 | 安全 | 高<br>（当访问增多时，Sessionn会比较占用你服务器的性能） | 存放 重要信息 |

## 状态码

HTTP状态码分类

| 分类 | 分类描述 |
| --- | --- |
| 1** | 信息，服务器收到请求，需要请求者继续执行操作 |
| 2** | 成功，操作被成功接收并处理 |
| 3** | 重定向，需要进一步的操作以完成请求 |
| 4** | 客户端错误，请求包含语法错误或无法完成请求 |
| 5** | 服务器错误，服务器在处理请求的过程中发生了错误 |

- 200 - 请求成功
- 301 - 资源（网页等）被永久转移到其它 URL
- 404 - 请求的资源（网页等）不存在
- 500 - 内部服务器错误

- 100 客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应。

## Http 和 https

这里的 s 表示 SSL (secure socket layer)

这里我们有两个问题：

1. 传统的 http 发送的信息是暴露的，因此我们可以加密传输，但是是否所有人用的都是同一个加密和解密算法

2. 加密和解密算法到底怎么传输

连接建立过程：

1. 通过 tcp/443 号端口进行连接

2. Client 首先发送一个包，包含如下信息


3.server 反馈一个信息


4. server 发送证书


包含 public key 给用户进行加密

5. server 发送 server hello done

6. client 发送 certificate verify

7. client 发送 change cipher spec 表示从现在开始 client 要发送加密的信息了

8. client 发送 finished message 包含到目前为止交换的所有信息，防止一些信息被延迟了

9. client 发送 change cipher spec

10. server 发送 finished message 包含到目前为止交换的所有信息，防止一些信息被延迟了

11. 到目前为止 ssl 建立完成，之后 client 发送一个包含对称加密秘钥的 data，这个 data 是通过服务端的私钥进行解密， 之所以要用对称加密，是因为解密复杂度更低

常用对称加密算法：

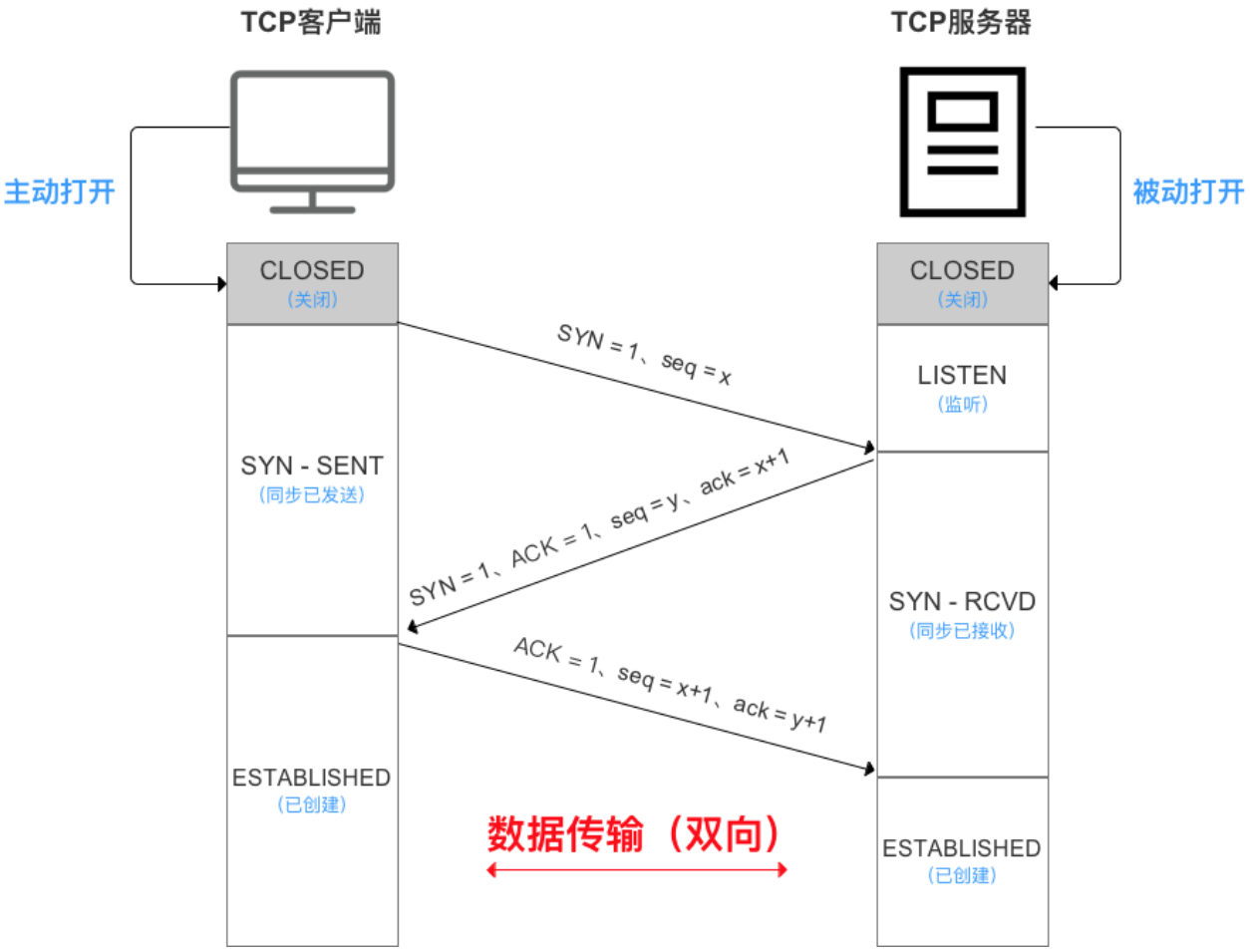
DES，RC

## Http 缓存

https://www.cnblogs.com/chenqf/p/6386163.html

# TCP

## 建立连接与释放连接

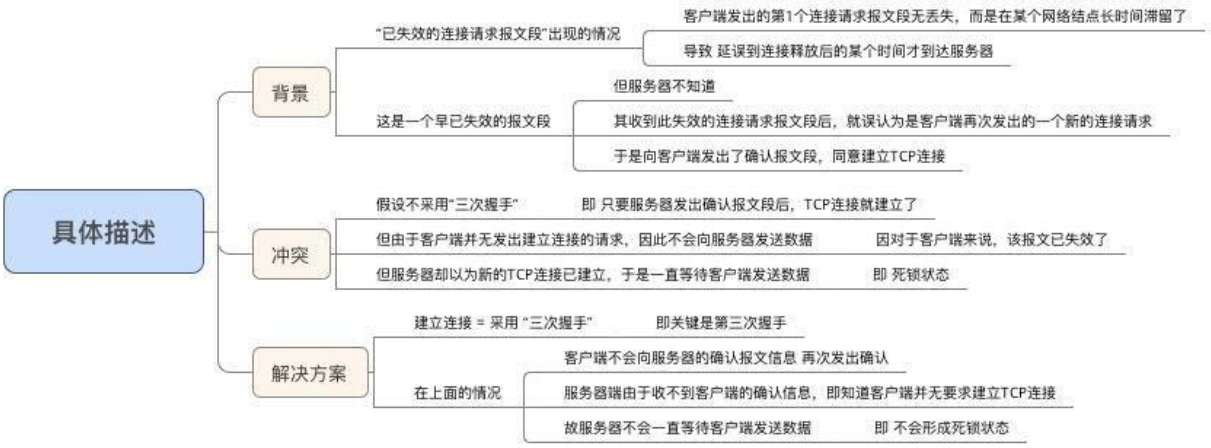


建立TCP连接前

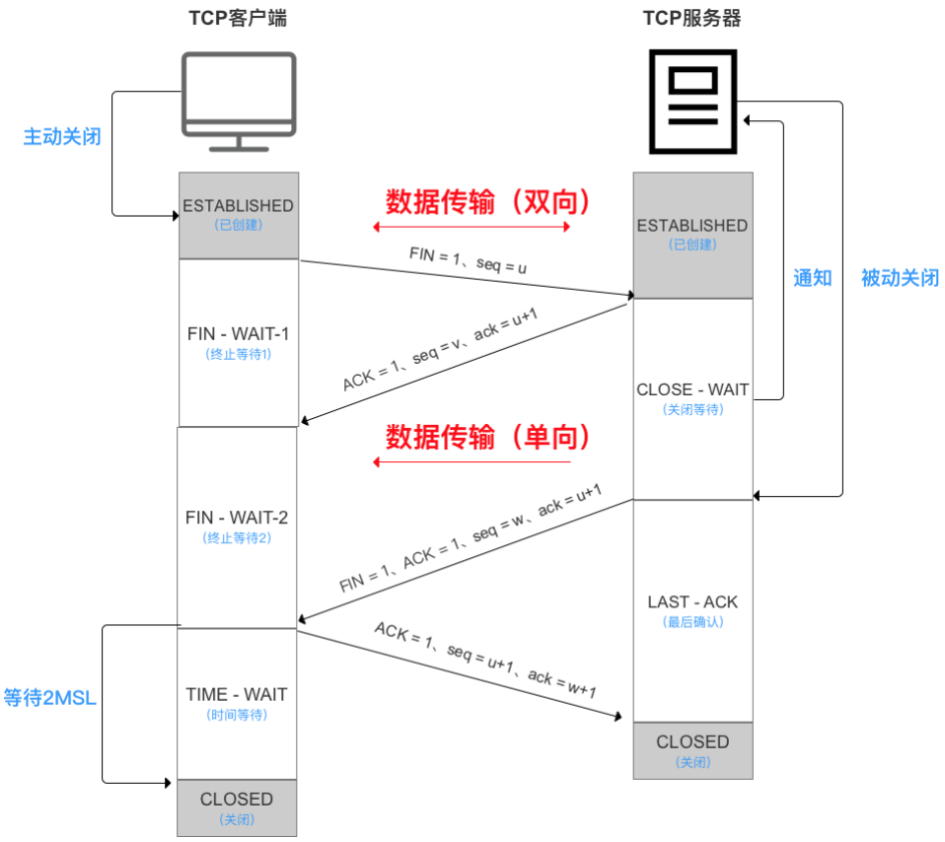
- TCP客户端、服务器都处于关闭状态（CLOSED）
- 直到：客户端主动打开连接，服务器才被动打开连接（处于监听状态 = LISTEN），等待接收客户端的请求

| 过程 | 具体描述 | 报文段信息 | 状态 |
| --- | --- | --- | --- |
| 第一次握手 | 客户端向服务器发送1个连接请求的报文段 | • 同步标志位 设为1：SYN = 1<br>• 随机选择一个起始序号：seq = x<br>• 不携带数据<br>（因SYN位被设置为1的报文段不能携带数据，但要消耗一个序号） | 客户端进入 同步已发送 状态（SYN_SEND）<br>（等待服务器的确认） |
| 第二次握手 | 服务器收到请求连接报文段后，若同意建立连接，则向客户端发回连接确认的报文段<br>（为该TCP连接分配TCP缓存、变量） | • 同步标志位 设为1：SYN = 1<br>• 确认标记位 设为1：ACK = 1<br>• 随机选择一个起始序号：seq = y<br>• 确认号字段 设为：ack = x+1<br>• 不携带数据<br>（原因同上；但要消耗一个序号） | 服务器进入 同步已接收 状态（SYN_RCVD） |
| 第三次握手 | 客户端收到确认报文段后，向服务器再次发出连接确认报文段<br>（为该TCP连接分配TCP缓存、变量） | • 确认标记位 设为1：ACK = 1<br>• 序号：seq = x+1<br>• 确认号字段 设为：ack = y+1<br>• 可携带数据<br>（因SYN无设为1；若不携带数据则不消耗序号） | 客户端、服务器端都进入已创建状态 (ESTABLISHED)<br>（可开始发送数据） |

为什么三次握手：



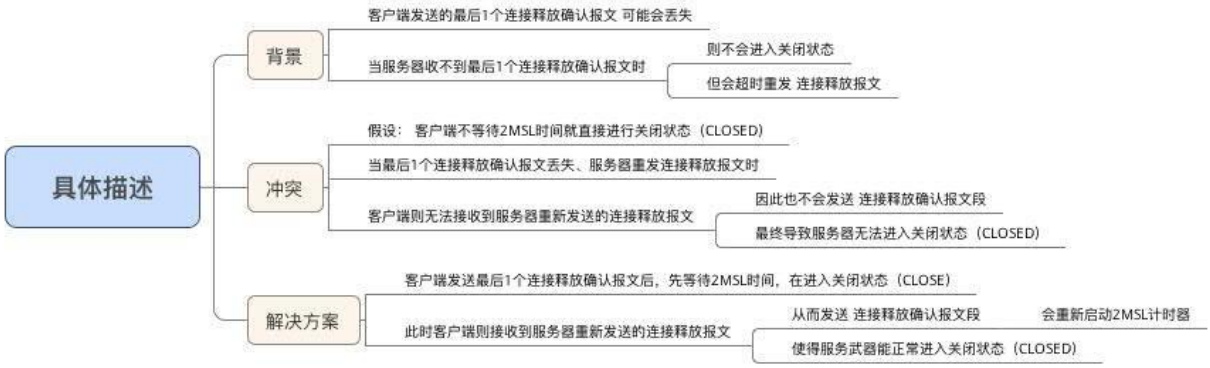

- 从上可看出：服务端的 TCP 资源分配时刻 = 完成第二次握手时；而客户端的 TCP 资源分配时刻 = 完成第三次握手时




## 释放TCP连接前

- TCP客户端、服务器都处于已创建状态 (ESTABLISHED)
- 直到：客户端主动关闭TCP连接

| 过程 | 具体描述 | 报文段信息 | 状态 |
| --- | --- | --- | --- |
| 第一次挥手 | 客户端向服务器发送1个连接释放的报文段<br>（停止再发送数据） | • 终止控制位 设为1：FIN = 1<br>• 报文段序号 设为前面传送数据最后1个字节的序号加1：seq = u<br>• 可携带数据<br>（FIN = 1的报文即使不携带数据也消耗1个序号） | 客户端进入 终止等待1 状态（FIN–WAIT–1）<br>（等待服务器的确认 ） |
| 第二次挥手 | 服务器收到连接释放报文段后，则向客户端发回连接释放确认的报文段 | • 确认标记位 设为1：ACK = 1<br>• 报文段序号 设为前面传送数据最后1个字节的序号加1：seq = v<br>• 确认号字段 设为：ack = u+1 | • 服务器进入 关闭等待 状态（CLOSE – WAIT）<br>• 客户端收到服务器的确认后，进入终止等待2 状态（FIN–WAIT–2），等待服务器发出释放连接请求<br>• 至此，客户端 –> 服务器的TCP连接已断开<br>• 即TCP连接处于半关闭状态<br>• 即 客户端 –>服务器断开，但服务器 –>客户端未断开 |
| 第三次挥手 | 若服务器已无要向客户端发送数据，则发出释放连接的报文段 | • 终止控制位 设为1：FIN = 1<br>• 确认标记位 设为1：ACK = 1<br>• 报文段序号 ： seq = w<br>• 重复上次已发送的确认号字段 设为：ack = u+1<br>• 可携带数据<br>（FIN = 1的报文即使不携带数据也消耗1个序号） | 服务器端进入 最后确认状态 (LAST–ACK) |
| 第四次挥手 | 客户端收到连接释放报文段后，则向服务器发回连接释放确认的报文段 | • 确认标记位 设为1：ACK = 1<br>• 报文段序号 ： seq = u+1<br>• 确认号字段 设为：ack = w+1<br>• 可携带数据<br>（FIN = 1的报文即使不携带数据也消耗1个序号） | • 客户端进入 时间等待状态 (TIME – WAIT)<br>• 服务器进入 关闭状态（CLOSED）<br>• 此时TCP连接还未释放<br>• 须经过时间等待计时器设置的时间2MSL后，客户端才进入连接关闭状态（CLOSED）<br>• 即 服务器进入关闭状态比客户端要早一些 |




- 原因 2：防止 上文提到的早已失效的连接请求报文 出现在本连接中
  客户端发送了最后 1 个连接释放请求确认报文后，再经过 2MSL 时间，则可使本连接持续时间内所产生的所有报文段都从网络中消失。
- 考虑这种情况，服务器运行在 80 端口，客户端使用的连接端口是 12306，数据传输完毕后服务端主动关闭连接，但是没有进入 TIME_WAIT，而是直接计入 CLOSED 了。这时，客户端又通过同样的端口 12306 与服务端建立了一个新的连接。假如上一个连接过程中网络出现了异常，导致了某个包重传并延时到达了服务端，这时服务端就无法区分这个包是上一个连接的还是这个连接的。所以，主动关闭连接一方要等待 2MSL，然后才能 CLOSE，保证连接中的 IP 包都要么传输完成，要么被丢弃了。

## 无差错传输

1. 每收到一个确认帧，发送窗口就向前滑动一个帧的距离
2. 当发送窗口内无可发送的帧时（即窗口内的帧全部是已发送但未收到确认的帧），发送方就会停止发送，直到收到接收方发送的确认帧使窗口移动，窗口内有可以发送的帧，之后才开始继续发送

对于接收端：当收到数据帧后，将窗口向前移动一个位置，并发回确认帧，若收到的数据帧落在接收窗口之外，则一律丢弃。


- 只有接收窗口向前滑动、接收方发送了确认帧时，发送窗口才有可能（只有发送方收到确认帧才是一定）向前滑动
- 停止-等待协议、后退 N 帧协议 & 选择重传协议只是在发送窗口大小和接收窗口大小上有所差别：

    1. 停止等待协议：发送窗口大小=1，接收窗口大小=1；即 单帧滑动窗口 等于 停止-等待协议
    2. 后退 N 帧协议：发送窗口大小>1，接收窗口大小=1。
    3. 选择重传协议：发送窗口大小>1，接收窗口大小>1。

- 当接收窗口的大小为 1 时，可保证帧有序接收。
- 数据链路层的滑动窗口协议中，窗口的大小在传输过程中是固定的（注意要与 TCP 的滑动窗口协议区别）

实现无差错传输的解决方案

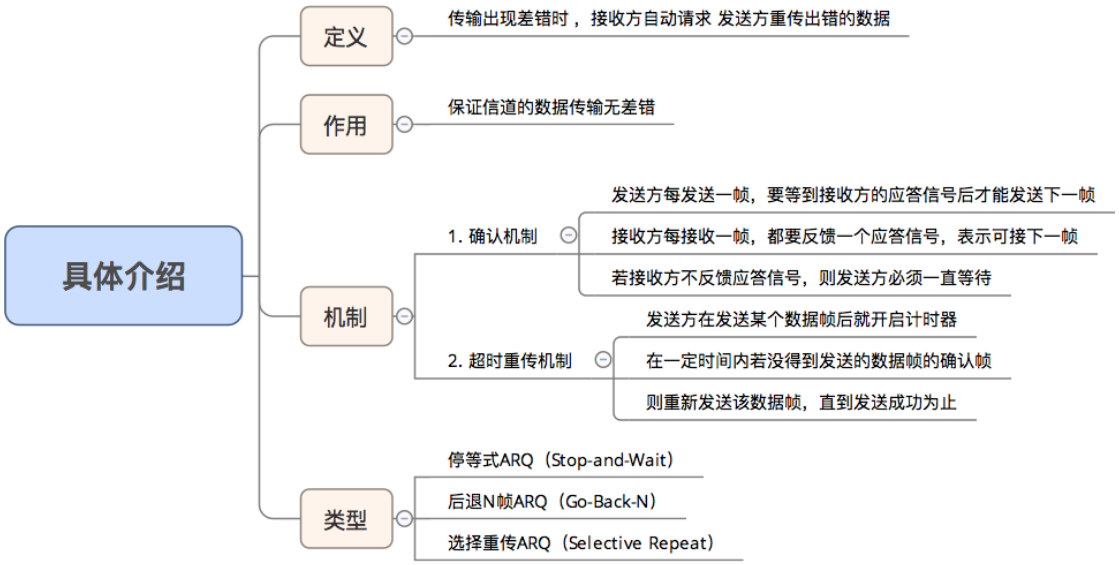
核心思想：采用一些可靠传输协议，使得出现差错时，让发送方重传差错数据：即

**出错重传**

当接收方来不及接收收到的数据时，可通知发送方降低发送数据的效率：即

**速度匹配**

针对上述 2 个问题，分别采用的解决方案是：自动重传协议 和 流量控制 & 拥塞控制协议




| 类型 | 原理 | 特点 |
| --- | --- | --- |
| 停等式ARQ<br>(Stop-and-Wait) | （单帧滑动窗口）停止 – 等待协议 + 超时重传 | 发送窗口大小=1、接收窗口大小=1 |
| 后退N帧ARQ<br>(Go-Back-N) | 多帧滑动窗口 + 累计确认 + 后退N帧 + 超时重传 | 发送窗口大小>1、接收窗口大小=1 |
| 选择重传ARQ<br>(Selective Repeat) | 多帧滑动窗口 + 累计确认 + 后退N帧 + 超时重传 | 发送窗口大小>1、接收窗口大小>1 |

*出错重传*

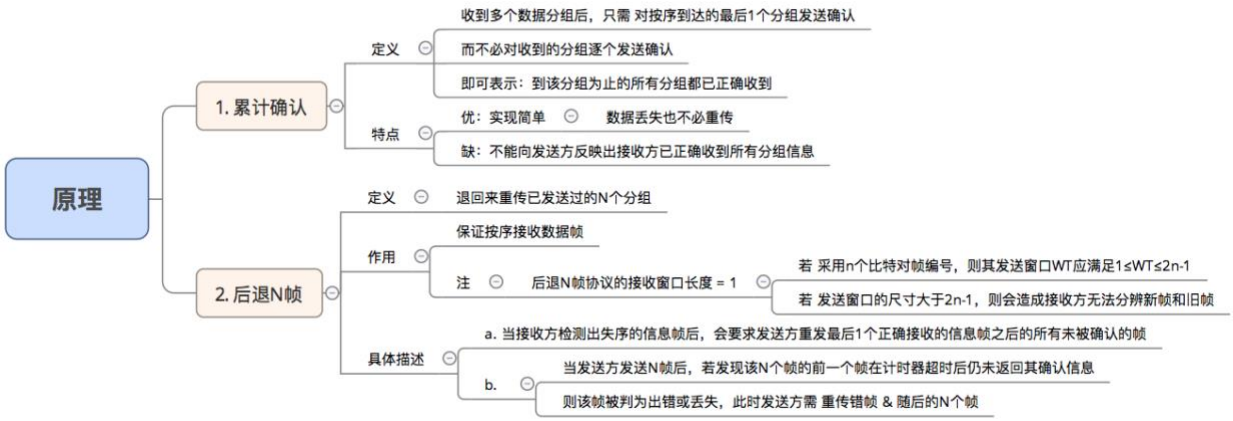
**退 N 帧协议**

连续 ARQ 协议

- 原理
  多帧滑动窗口 + 累计确认 + 后退 N 帧 + 超时重传

即 ：发送窗口大小>1、接收窗口大小=1

- 具体描述
  a. 发送方：采用多帧滑动窗口的原理，可连续发送多个数据帧 而不需等待对方确认
  b. 接收方：采用 **累计确认 & 后退 N 帧**的原理，只允许按顺序接收帧。具体原理如下：




**选择重传 ARQ**

- 原理
  多帧滑动窗口 + 累计确认 + 后退 N 帧 + 超时重传

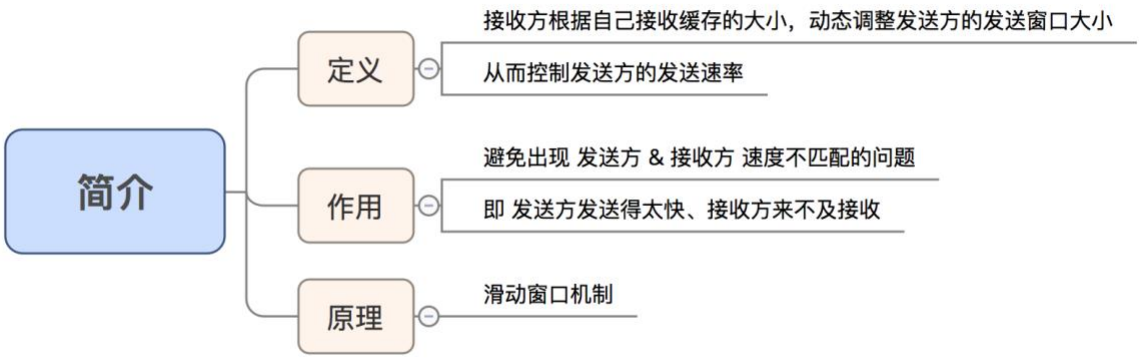
即 ：发送窗口大小>1、接收窗口大小>1

类似于类型 2（后退 N 帧协议），此处仅仅是接收窗口大小的区别，故此处不作过多描述

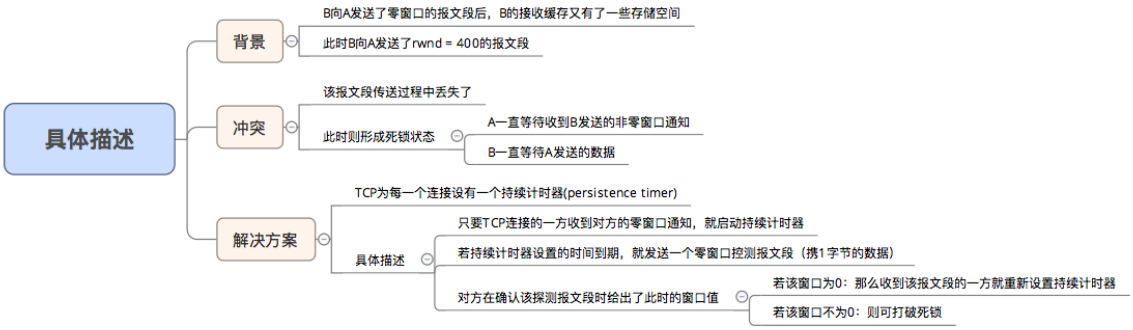
- 特点
  a. 优：因连续发送数据帧而提高了信道的利用率
  b. 缺：重传时又必须把原来已经传送正确的数据帧进行重传（仅因为这些数据帧前面有一个数据帧出了错），将导致传送效率降低

由此可见，若信道传输质量很差，导致误码率较大时，后退 N 帧协议不一定优于停止-等待协议

**流量控制 & 拥塞控制**



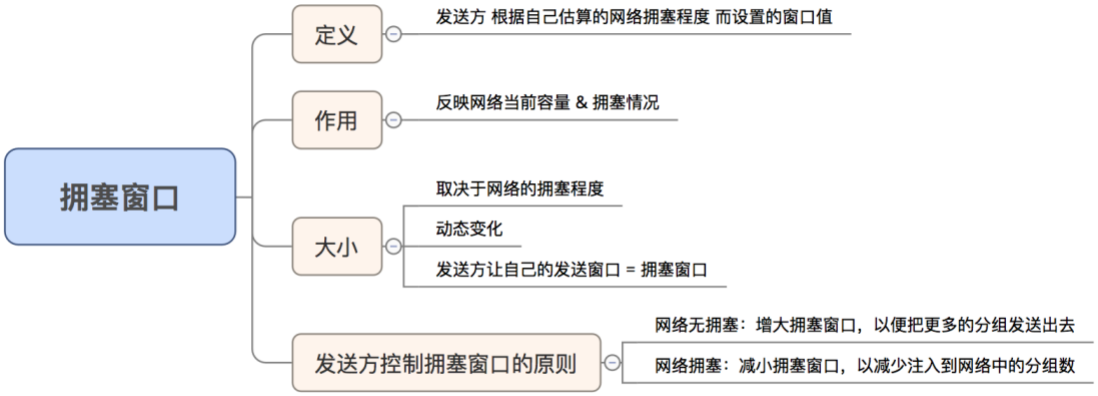

注意死锁问题




## 拥塞控制

**慢开始 & 拥塞避免**

**拥塞窗口**




**慢开始算法**

- 原理

  当主机开始发送数据时，由小到大逐渐增大 拥塞窗口数值（即 发送窗口数值），从而由小到大逐渐增大发送报文段，一般是两倍

- 目的

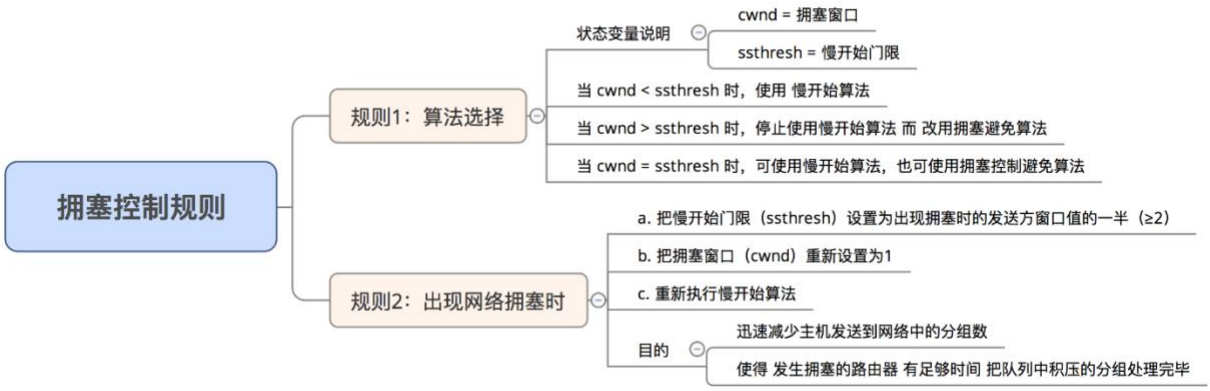
  开始传输时，试探网络的拥塞情况

**拥塞避免 算法**

- 原理

  使得拥塞窗口`（cwnd）`**按线性规律 缓慢增长**：每经过一个往返时间`RTT`，发送方的拥塞窗口`（cwnd）`加 1

1. **拥塞避免 并不可避免拥塞**，只是将拥塞窗口按现行规律缓慢增长，使得网络比较不容易出现拥塞
2. 相比慢开始算法的加倍，拥塞窗口增长速率缓慢得多




**快重传 & 快恢复**

**a. 快重传算法**

- 原理
  1. 接收方 每收到一个失序的报文段后 就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方），而不要等到自己发送数据时才进行捎带确认
  2. 发送方只要一连收到 3 个重复确认就立即重传对方尚未收到的报文段，而不必 继续等待设置的重传计时器到期
- 作用
  由于发送方尽早重传未被确认的报文段，因此采用快重传后可以使整个网络吞吐量提高约 20%

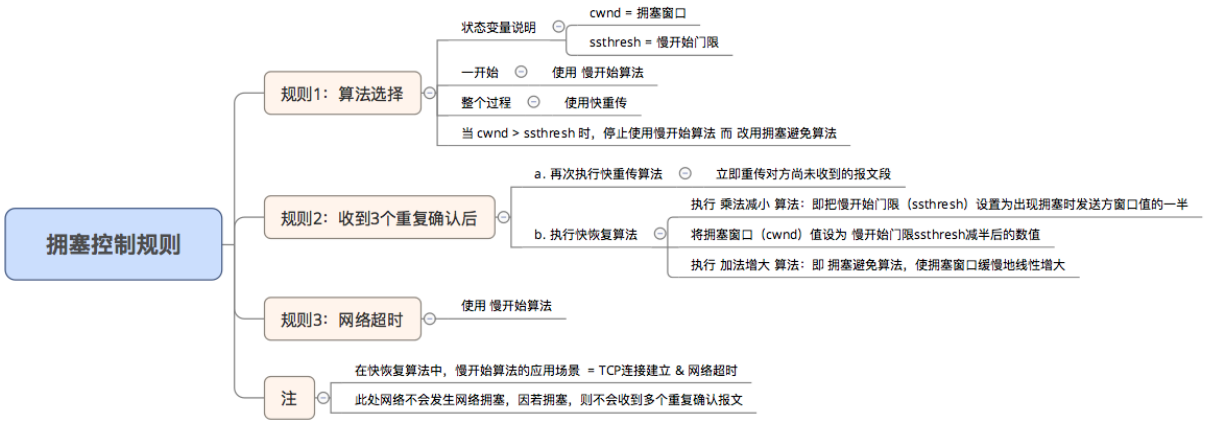
## b. 快恢复

当发送方连续收到 3 个重复确认后，就：

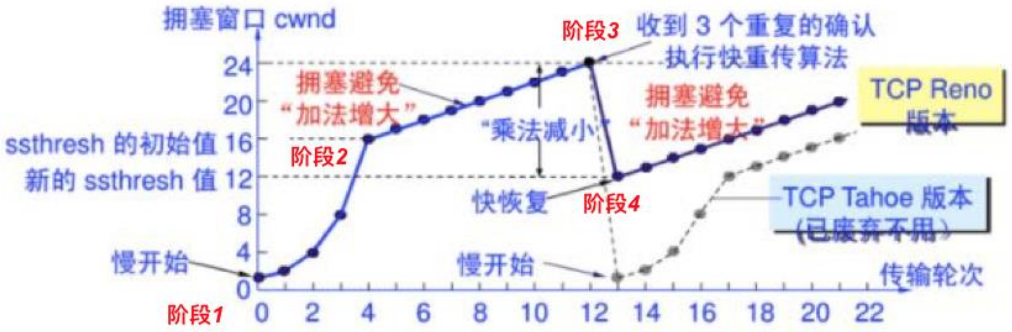
1. 执行 **乘法减小** 算法：把 慢开始门限（ssthresh）设置为 出现拥塞时发送方窗口值的一半 = 拥塞窗口的 1 半
2. 将拥塞窗口（cwnd）值设置为 慢开始门限 ssthresh 减半后的数值 = 拥塞窗口的 1 半
3. 执行 **加法增大** 算法：执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

由于跳过了拥塞窗口（cwnd）从 1 起始的慢开始过程，所以称为：快恢复

此处网络不会发生网络拥塞，因若拥塞，则不会收到多个重复确认报文




## 拥塞控制 过程




## 过程解析

- 阶段1：此时TCP连接刚建立，故使用 慢开始算法
- 阶段2：因cwnd > ssthresh，故停止使用慢开始算法 而 改用拥塞避免算法
- 阶段3：整个过程都使用快重传算法；此时收到了3个重复确认，故再次执行快重传算法，即 立即重传对方尚未收到的报文段
- 阶段4：执行快恢复算法，即执行 乘法减小 算法、将拥塞窗口（cwnd）值设为 慢开始门限ssthresh减半后的数值、执行 加法增大算法（即 拥塞避免算法）

## TCP 与 UDP 的区别

| 类型 | 特点 | | | 性能 | | 应用场景 | 首部字节 |
|---|---|---|---|---|---|---|---|
| | 是否面向连接 | 传输可靠性 | 传输形式 | 传输效率 | 所需资源 | | |
| TCP | 面向连接 | 可靠 | 字节流 | 慢 | 多 | 要求通信数据可靠<br>(如文件传输、邮件传输) | 20-60 |
| UDP | 无连接 | 不可靠 | 数据报文段 | 快 | 少 | 要求通信速度高<br>(如域名转换) | 8个字节<br>(由4个字段组成) |

## Restful

**网络上的所有事物都被抽象为资源**

**每个资源都有一个唯一的资源标识符**

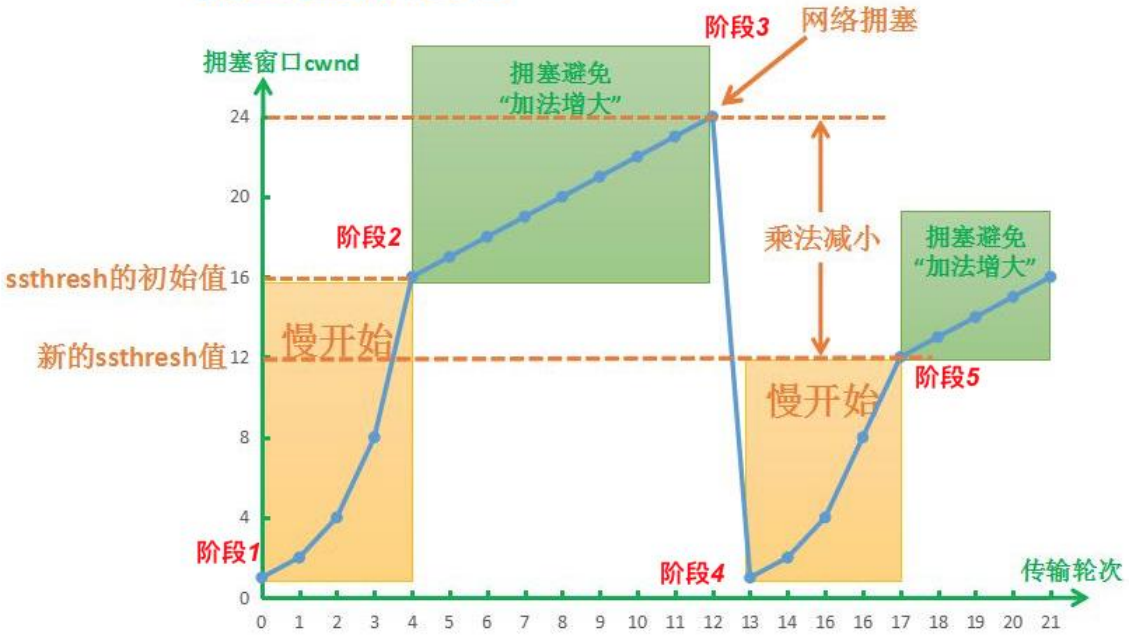
**同一个资源具有多种表现形式(xml,json 等)**

**对资源的各种操作不会改变资源标识符**

**所有的操作都是无状态的**

**符合 REST 原则的架构方式即可称为 RESTful**

# 拥塞控制 过程




# 过程解析

- 阶段1：因cwnd < ssthresh ， 故使用 慢开始算法
- 阶段2：因cwnd > ssthresh ， 故停止使用慢开始算法 而 改用拥塞避免算法
- 阶段3：出现网路拥塞，把慢开始门限（ssthresh）设置为出现拥塞时的发送方窗口值的一半（即 12）、把拥塞窗口（cwnd）重新设置为1
- 阶段4：因cwnd < ssthresh，故使用 慢开始算法
- 阶段5：因cwnd > ssthresh ， 故停止使用慢开始算法 而 改用拥塞避免算法

•注：

a. 乘法减小：出现网络拥塞时慢开始门限（ssthresh）设置为出现拥塞时的发送方窗口值的一半

b. 加法增大：拥塞避免时的缓慢增大

c. 二者合并叫为AIMD算法（即 加法增大、乘法减少）

# Java 锁机制

## java 并发系列：深入分析 Synchronized

## CAS

在 JDK 5 之前 Java 语言是靠 synchronized 关键字保证同步的，这会导致有锁

锁机制存在以下问题：

（1）在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。

（2）一个线程持有锁会导致其它所有需要此锁的线程挂起。

（3）如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

volatile 是不错的机制，但是 volatile 不能保证原子性。因此对于同步最终还是要回到锁机制上来。

**独占锁是一种悲观锁，synchronized 就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁用到的机制就是 CAS，Compare and Swap。**

### 一、什么是 CAS

CAS,compare and swap 的缩写，中文翻译成比较并交换。

我们都知道，在 java 语言之前，并发就已经广泛存在并在服务器领域得到了大量的应用。所以硬件厂商老早就在芯片中加入了大量直至并发操作的原语，从而在硬件层面提升效率。在 intel 的 CPU 中，使用 cmpxchg 指令。

在 Java 发展初期，java 语言是不能够利用硬件提供的这些便利来提升系统的性能的。而随着 java 不断的发展,Java 本地方法(JNI)的出现，使得 java 程序越过 JVM 直接调用本地方法提供了一种便捷的方式，因而 java 在并发的手段上也多了起来。而在 Doug Lea 提供的 cucurenct 包中，CAS 理论是它实现整个 java 包的基石。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该 位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，

而不提取当前 值。 ）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新 值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行读-修改-写操作，而无需害怕其他线程同时 修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算法 可以对该操作重新计算。

## 二、CAS 的目的

利用 CPU 的 CAS 指令，同时借助 JNI 来完成 Java 的非阻塞算法。其它原子操作都是利用类似的特性完成的。而整个 J.U.C 都是建立在 CAS 之上的，因此对于 synchronized 阻塞算法，J.U.C 在性能上有了很大的提升。

三、CAS 存在的问题

CAS 虽然很高效的解决原子操作，但是 CAS 仍然存在三大问题。ABA 问题，循环时间长开销大和只能保证一个共享变量的原子操作

1. **ABA 问题**。因为 CAS 需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是 A，变成了 B，又变成了 A，那么使用 CAS 进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA 问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么 A－B－A 就会变成 1A-2B－3A。

**从 Java1**.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。这个类的 compareAndSet 方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

关于 ABA 问题参考文档: http://blog.hesey.net/2011/09/resolve-aba-by-atomicstampedreference.html

**2. 循环时间长开销大**。自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。如果 JVM 能支持处理器提供的 pause 指令那么效率会有一定的提升，pause 指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使 CPU 不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起 CPU 流水线被清空（CPU pipeline flush），从而提高 CPU 的执行效率。

**3. 只能保证一个共享变量的原子操作**。当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量 i＝2,j=a，合

并一下 ij=2a，然后用 CAS 来操作 ij。从 Java1.5 开始 JDK 提供了 **AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作。**

## Synchronized

类锁

对象锁

修饰静态方法锁 等同于类锁

synchronized 关键字最主要有以下 3 种应用方式，下面分别介绍

- 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
- 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
- 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

**实现原理：**

*修饰块*

每个对象有一个监视器锁（monitor）。当 monitor 被占用时就会处于锁定状态，线程执行 monitorenter 指令时尝试获取 monitor 的所有权，过程如下：

1、如果 monitor 的进入数为 0，则该线程进入 monitor，然后将进入数设置为 1，该线程即为 monitor 的所有者。

2、如果线程已经占有该 monitor，只是重新进入，则进入 monitor 的进入数加 1.

3.如果其他线程已经占用了 monitor，则该线程进入阻塞状态，直到 monitor 的进入数为 0，再重新尝试获取 monitor 的所有权。

*修饰对象*

方法的同步并没有通过指令 monitorenter 和 monitorexit 来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了 ACC_SYNCHRONIZED 标示符。JVM 就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取 monitor，获取成功之后才能执行方法体，方法执行完后再释放 monitor。在方法执行期间，其他任何线程都无法再获得同一个 monitor 对象。

其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。



- 实例变量：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按 4 字节对齐。
- 填充数据：由于虚拟机要求对象起始地址必须是 8 字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐，这点了解即可。

| • 虚拟机位数 | 头对象结构 | 说明 |
| --- | --- | --- |
| 32/64bit | Mark Word | 存储对象的 hashCode、锁信息或分代年龄或 GC 标志等信息 |
| 32/64bit | 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例。 |

其中 Mark Word 在默认情况下存储着对象的 HashCode、分代年龄、锁标记位等以下是 32 位 JVM 的 Mark Word 默认存储结构

| 锁状态 | 25bit | 4bit | 1bit 是否是偏向锁 | 2bit 锁标志位 |
| --- | --- | --- | --- | --- |
| 无锁状态 | 对象 HashCode | 对象分代年龄 | 0 | 01 |

| 锁状态 | 25 bit | | 4bit | 1bit | 2bit |
| --- | --- | --- | --- | --- | --- |
| | 23bit | 2bit | | 是否是偏向锁 | 锁标志位 |
| 轻量级锁 | 指向栈中锁记录的指针 | | | | 00 |
| 重量级锁 | 指向互斥量（重量级锁）的指针 | | | | 10 |
| GC标记 | 空 | | | | 11 |
| 偏向锁 | 线程ID | Epoch | 对象分代年龄 | 1 | 01 |



这里我们主要分析一下重量级锁也就是通常说 synchronized 的对象锁，锁标识位为 10，其中指针指向的是 monitor 对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有存在多种实现方式，如 monitor 可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。在 Java 虚拟机(HotSpot)中，monitor 是由 ObjectMonitor 实现的，其主要数据结构如下（位于 HotSpot 虚拟机源码 ObjectMonitor.hpp 文件，C++实现的 ）

## 和 lock 的关系

**synchronized 的局限性**

- 占有锁的线程等待 IO 或者其他原因被阻塞，没有释放锁的情况下，其他线程一直阻塞
- 多个线程同时读写文件的时候，读和读操作也会发生冲突
- 我们没有办法知道当前我们的线程是否成功获取了锁，只能傻傻的等待

synchronized 是 java 中的一个关键字，也就是说是 Java 语言内置的特性。那么为什么会出现 Lock 呢？

在上面一篇文章中，我们了解到如果一个代码块被 synchronized 修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时 JVM 会让线程自动释放锁。

那么如果这个获取锁的线程由于要等待 IO 或者其他原因（比如调用 sleep 方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过 Lock 就可以办到。

再举个例子：当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。

但是采用 synchronized 关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过 Lock 就可以办到。

另外，通过 Lock 可以知道线程有没有成功获取到锁。这个是 synchronized 无法办到的。

总结一下，也就是说 Lock 提供了比 synchronized 更多的功能。但是要注意以下几点：

1）Lock 不是 Java 语言内置的，synchronized 是 Java 语言的关键字，因此是内置特性。Lock 是一个类，通过这个类可以实现同步访问；

2）Lock 和 synchronized 有一点非常大的不同，采用 synchronized 不需要用户去手动释放锁，当 synchronized 方法或者 synchronized 代码块执行完之后，系统会自动让线程释放对锁的占用；而 Lock 则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。

1）Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现；

2）synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象，因此使用 Lock 时需要在 finally 块中释放锁；

3）Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时，等待的线程会一直等待下去，不能够响应中断；

4）通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

5）Lock 可以提高多个线程进行读操作的效率。

1.可重入锁

如果锁具备可重入性，则称作为可重入锁。像 synchronized 和 ReentrantLock 都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个 synchronized 方法时，比如说 method1，而在 method1 中会调用另外一个 synchronized 方法 method2，此时线程不必重新去申请锁，而是可以直接执行方法 method2。

2.可中断锁

可中断锁：顾名思义，就是可以相应中断的锁。

在 Java 中，synchronized 就不是可中断锁，而 Lock 是可中断锁。

如果某一线程 A 正在执行锁中的代码，另一线程 B 正在等待获取该锁，可能由于等待时间过长，线程 B 不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

在前面演示 lockInterruptibly()的用法时已经体现了 Lock 的可中断性。

3.公平锁

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

在 Java 中，synchronized 就是非公平锁，它无法保证等待的线程获取锁的顺序。

而对于 ReentrantLock 和 ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

看一下这 2 个类的源代码就清楚了：

4.读写锁

读写锁将对一个资源（比如文件）的访问分成了 2 个锁，一个读锁和一个写锁。

正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。

ReadWriteLock 就是读写锁，它是一个接口，ReentrantReadWriteLock 实现了这个接口。

可以通过 readLock()获取读锁，通过 writeLock()获取写锁。

上面已经演示过了读写锁的使用方法，在此不再赘述。

# Volatile

## volatile 两大作用

**1、保证内存可见性**

**2、防止指令重排**

此外需注意 volatile **并不保证操作的原子性。**

### （一）内存可见性

1 概念

**JVM 内存模型：主内存和线程独立的工作内存**

Java 内存模型规定，对于多个线程共享的变量，存储在主内存当中，每个线程都有自己独立的工作内存（比如 CPU 的寄存器），线程只能访问自己的工作内存，不可以访问其它线程的工作内存。

工作内存中保存了主内存共享变量的**副本**，线程要操作这些共享变量，只能通过操作工作内存中的副本来实现，操作完毕之后再同步回到主内存当中。

如何保证多个线程操作主内存的数据完整性是一个难题，Java 内存模型也规定了工作内存与主内存之间交互的协议，定义了 8 种原子操作：

(1) lock:将主内存中的变量锁定，为一个线程所独占

(2) unclock:将 lock 加的锁定解除，此时其它的线程可以有机会访问此变量

(3) read:将主内存中的变量值读到工作内存当中

(4) load:将 read 读取的值保存到工作内存中的变量副本中。

(5) use:将值传递给线程的代码执行引擎

(6) assign:将执行引擎处理返回的值重新赋值给变量副本

(7) store:将变量副本的值存储到主内存中。

(8) write:将 store 存储的值写入到主内存的共享变量当中。

**通过上面 Java 内存模型的概述，我们会注意到这么一个问题，每个线程在获取锁之后会在自己的工作内存来操作共享变量，操作完成之后将工作内存中的副本回写到主内存，并且在其它线程从主内存将变量同步回自己的工作内存之前，共享变量的改变对其是不可见的。**即其他线程的本地内存中的变量已经是过时的，并不是更新后的值。

2 内存可见性带来的问题

很多时候我们需要一个线程对共享变量的改动，其它线程也需要立即得知这个改动该怎么办呢？下面举两个例子说明内存可见性的重要性：

## 例子 1

有一个全局的状态变量 open:

boolean open=true;

这个变量用来描述对一个资源的打开关闭状态，true 表示打开，false 表示关闭，假设有一个线程 A,在执行一些操作后将 open 修改为 false:

**//线程 A**

resource.close();

open = false;

线程 B 随时关注 open 的状态，当 open 为 true 的时候通过访问资源来进行一些操作:

**//线程 B**

while(open) {

doSomethingWithResource(resource);

}

当 A 把资源关闭的时候，open 变量对线程 B 是不可见的，如果此时 open 变量的改动尚未同步到线程 B 的工作内存中,那么线程 B 就会用一个已经关闭了的资源去做一些操作，因此产生错误。

3 提供内存可见性

volatile 保证可见性的原理是在**每次访问变量时都会进行一次刷新**，因此每次访问都是主内存中最新的版本。所以 volatile 关键字的作用之一就是**保证变量修改的实时可见性**。

**针对上面的例子 1：**

要求一个线程对 open 的改变，其他的线程能够立即可见，Java 为此提供了 volatile 关键字，在声明 open 变量的时候加入 volatile 关键字就可以保证 open 的内存可见性，即 open 的改变对所有的线程都是立即可见的。

**针对上面的例子 2：**

将 cancelled 标志设置的 volatile 保证主线程针对 cancelled 标识的修改能够让 PrimeGenerator 线程立马看到。

备注：也可以通过提供 synchronized 同步的 open 变量的 Get/Set 方法解决此内存可见性问题，因为要 Get 变量 open，必须等 Set 方完全释放锁之后。后面将介绍到两者的区别。

## （二）指令重排

### 1 概念

指令重排序是 JVM 为了优化指令，提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度。编译器、处理器也遵循这样一个目标。注意是单线程。多线程的情况下指令重排序就会给程序员带来问题。

不同的指令间可能存在数据依赖。比如下面计算圆的面积的语句：

```
double r = 2.3d;//(1)

double pi =3.1415926; //(2)

double area = pi* r * r; //(3)
```

area 的计算依赖于 r 与 pi 两个变量的赋值指令。而 r 与 pi 无依赖关系。

as-if-serial 语义是指：不管如何重排序（编译器与处理器为了提高并行度），（单线程）程序的结果不能被改变。这是编译器、Runtime、处理器必须遵守的语义。

虽然，（1）- happensbefore -> （2），（2）- happens before -> （3），但是计算顺序(1)(2)(3)与(2)(1)(3) 对于 r、pi、area 变量的结果并无区别。编译器、Runtime 在优化时可以根据情况重排序（1）与（2），而丝毫不影响程序的结果。

指令重排序包括编译器重排序和运行时重排序。

### 2 指令重排带来的问题

如果一个操作不是原子的，就会给 JVM 留下重排的机会。下面看几个例子：

## 例子 1：A 线程指令重排导致 B 线程出错

对于在同一个线程内，这样的改变是不会对逻辑产生影响的，但是在多线程的情况下指令重排序会带来问题。看下面这个情景:

**在线程 A 中:**

```
context = loadContext();

inited = true;
```

**在线程 B 中:**

```
while(!inited ){ //根据线程 A 中对 inited 变量的修改决定是否使用 context 变量

   sleep(100);

}

doSomethingwithconfig(context);
```

**假设线程 A 中发生了指令重排序:**

```
inited = true;

context = loadContext();
```

那么 B 中很可能就会拿到一个尚未初始化或尚未初始化完成的 context,从而引发程序错误。

## 例子 2：指令重排导致单例模式失效

我们都知道一个经典的懒加载方式的双重判断单例模式：

```
public class Singleton {
  private static Singleton instance = null;
  private Singleton() { }
  public static Singleton getInstance() {
     if(instance == null) {
        synchronzied(Singleton.class) {
           if(instance == null) {
              instance = new Singleton();  //非原子操作
           }
        }
     }
     return instance;
  }
}
```

看似简单的一段赋值语句：instance= new Singleton()，但是很不幸它并不是一个原子操作，其实际上可以抽象为下面几条 JVM 指令：

```
memory =allocate();    //1：分配对象的内存空间
```

ctorInstance(memory);  //2：初始化对象

instance =memory;     //3：设置 instance 指向刚分配的内存地址

上面操作 2 依赖于操作 1，但是操作 3 并不依赖于操作 2，所以 JVM 是可以针对它们进行指令的优化重排序的，经过重排序后如下：

memory =allocate();    //1：分配对象的内存空间

instance =memory;     //3：instance 指向刚分配的内存地址，此时对象还未初始化

ctorInstance(memory);  //2：初始化对象

可以看到指令重排之后，instance 指向分配好的内存放在了前面，而这段内存的初始化被排在了后面。

在线程 A 执行这段赋值语句，在初始化分配对象之前就已经将其赋值给 instance 引用，恰好另一个线程进入方法判断 instance 引用不为 null，然后就将其返回使用，导致出错。

### 3 防止指令重排

除了前面内存可见性中讲到的 volatile 关键字可以保证变量修改的可见性之外，还有另一个重要的作用：**在 JDK1.5 之后，可以使用 volatile 变量禁止指令重排序。**

**解决方案：**例子 1 中的 inited 和例子 2 中的 instance 以关键字 volatile 修饰之后，就会阻止 JVM 对其相关代码进行指令重排，这样就能够按照既定的顺序指执行。

volatile 关键字通过提供“**内存屏障**”的方式来防止指令被重排序，为了实现 volatile 的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

大多数的处理器都支持内存屏障的指令。

对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，Java 内存模型采取保守策略。下面是基于保守策略的 JMM 内存屏障插入策略：

在每个 volatile 写操作的前面插入一个 StoreStore 屏障。

在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。

在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。

在每个 volatile 读操作的后面插入一个 LoadStore 屏障。

## （三）总结

### volatile 是轻量级同步机制

相对于 synchronized 块的代码锁，volatile 应该是提供了一个轻量级的针对共享变量的锁，当我们在多个线程间使用共享变量进行通信的时候需要考虑将共享变量用 volatile 来修饰。

volatile 是一种稍弱的同步机制，在访问 volatile 变量时不会执行加锁操作，也就不会执行线程阻塞，因此 volatilei 变量是一种比 synchronized 关键字更轻量级的同步机制。

### volatile 使用建议

使用建议：在两个或者更多的线程需要访问的成员变量上使用 volatile。当要访问的变量已在 synchronized 代码块中，或者为常量时，没必要使用 volatile。

由于使用 volatile 屏蔽掉了 JVM 中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

### volatile 和 synchronized 区别

**1、volatile 不会进行加锁操作：**

volatile 变量是一种稍弱的同步机制在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 synchronized 关键字更轻量级的同步机制。

**2、volatile 变量作用类似于同步变量读写操作：**

从内存可见性的角度看，写入 volatile 变量相当于退出同步代码块，而读取 volatile 变量相当于进入同步代码块。

**3、volatile 不如 synchronized 安全：**

在代码中如果过度依赖 volatile 变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当 volatile 变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。

4、**volatile 无法同时保证内存可见性和原则性：**

加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而 volatile 变量只能确保可见性，原因是声明为 volatile 的简单变量如果当前值与该变量以前的值相关，那么 volatile 关键字不起作用，也就是说如下的表达式都不是原子操作：“count++”、“count = count+1”。

**当且仅当满足以下所有条件时，才应该使用 volatile 变量：**

1、 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。

2、该变量没有包含在具有其他变量的不变式中。

## Java 虚拟机对 synchronized 的优化

锁的状态总共有四种，无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁，但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级，关于重量级锁，前面我们已详细分析过，下面我们将介绍偏向锁和轻量级锁以及 JVM 的其他优化手段，这里并不打算深入到每个锁的实现和转

换过程更多地是阐述 Java 虚拟机所提供的每个锁的核心优化思想，毕竟涉及到具体过程比较繁琐，如需了解详细过程可以查阅《深入理解 Java 虚拟机原理》。

## 偏向锁

偏向锁是 Java 6 之后加入的新锁，它是一种针对加锁操作的优化手段，经过研究发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些 CAS 操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，**如果一个线程获得了锁，那么锁就进入偏向模式，此时 Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。**所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。下面我们接着了解轻量级锁。

## 轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6 之后加入的)，此时 Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

## 自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线

程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

## 锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java 虚拟机在 JIT 编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下 StringBuffer 的 append 是一个同步方法，但是在 add 方法中的 StringBuffer 属于一个局部变量，并且不会被其他线程所使用，因此 StringBuffer 不可能存在共享资源竞争的情景，JVM 会自动将其锁消除。

# Java 容器

## HashMap

线程不安全，多线程 put 下会出现环形 Entry

## ConcurrentHashmap

ConcurrentHashMap 采用了分段锁的设计，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个 Map 加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个 Map 加锁，导致一些需要扫描整个 Map 的方法（如 size(), containsValue()）需要使用特殊的实现，另外一些方法（如 clear()）甚至放弃了对一致性的要求（ConcurrentHashMap 是弱一致性的，具体请查看 ConcurrentHashMap 能完全替代 HashTable 吗？）。

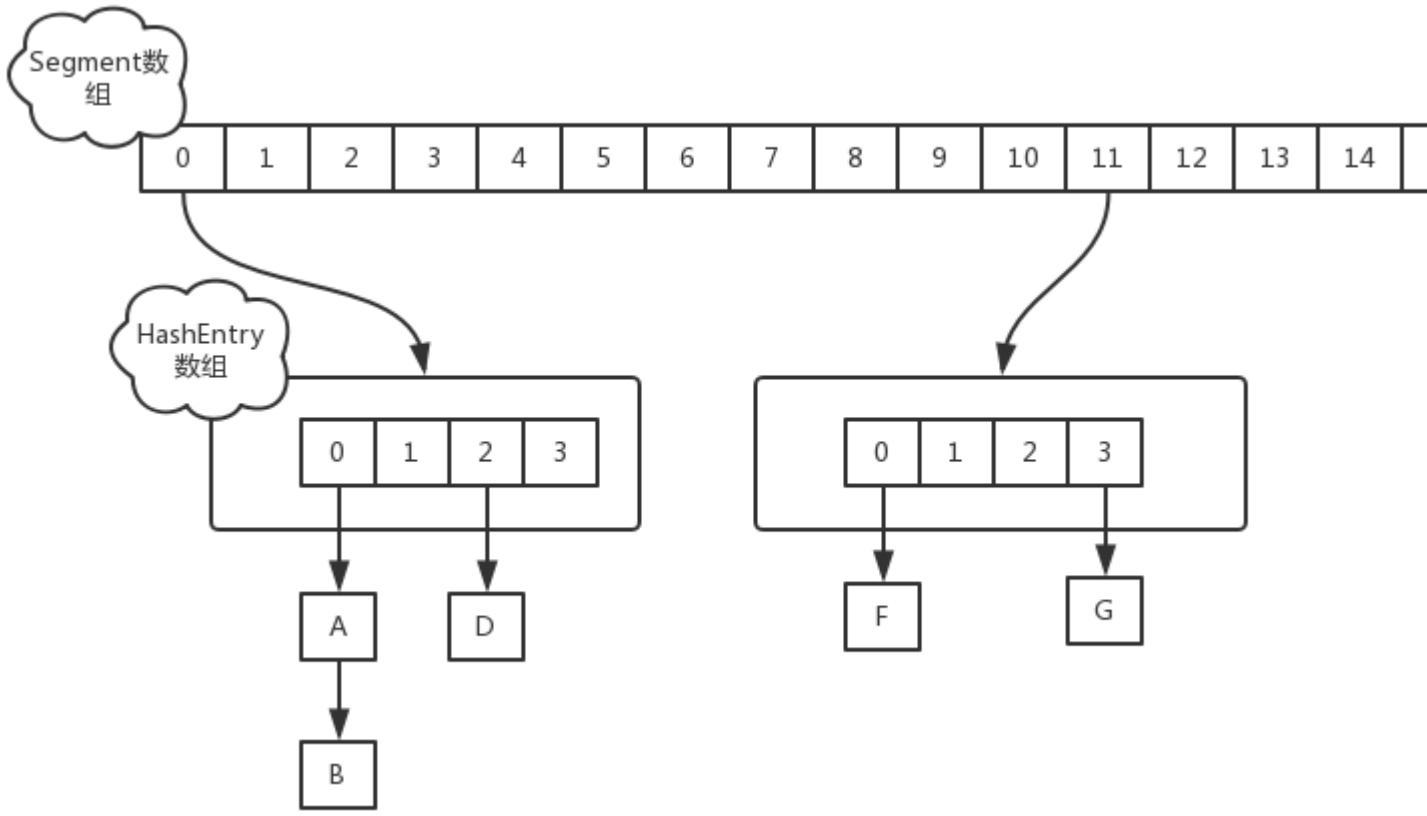
ConcurrentHashMap 中的分段锁称为 Segment，它即类似于 HashMap（JDK7 与 JDK8 中 HashMap 的实现）的结构，即内部拥有一个 Entry 数组，数组中的每个元素又是一个链表；同时又是一个 ReentrantLock（Segment 继承了 ReentrantLock）。ConcurrentHashMap 中的 HashEntry 相对于 HashMap 中的 Entry 有一定的差异性：HashEntry 中的 value 以及 next 都被 volatile 修饰，这样在多线程读写过程中能够保持它们的可见性，代码如下：

```
1   static final class HashEntry<K,V> {
2           final int hash;
3           final K key;
4           volatile V value;
```

```
5           volatile HashEntry<K,V> next;
```

## JDK1.7 的实现

在 JDK1.7 版本中，ConcurrentHashMap 的数据结构是由一个 Segment 数组和多个 HashEntry 组成，如下图所示：




Segment 数组的意义就是将一个大的 table 分割成多个小的 table 来进行加锁，也就是上面的提到的锁分离技术，而每一个 Segment 元素存储的是 HashEntry 数组+链表，这个和 HashMap 的数据存储结构一样

## 初始化

ConcurrentHashMap 的初始化是会通过位与运算来初始化 Segment 的大小，用 ssize 来表示，如下所示

```
1   int sshift = 0;
2   int ssize = 1;
```

```
3   while (ssize < concurrencyLevel) {
4       ++sshift;
5       ssize <<= 1;
    }
6
```

如上所示，因为 ssize 用位于运算来计算（ssize <<=1），所以 Segment 的大小取值都是以 2 的 N 次方，无关 concurrencyLevel 的取值，当然 concurrencyLevel 最大只能用 16 位的二进制来表示，即 65536，换句话说，Segment 的大小最多 65536 个，没有指定 concurrencyLevel 元素初始化，Segment 的大小 ssize 默认为 16

每一个 Segment 元素下的 HashEntry 的初始化也是按照位于运算来计算，用 cap 来表示，如下所示

```
1   int cap = 1;
2   while (cap < c)
3       cap <<= 1;
```

如上所示，HashEntry 大小的计算也是 2 的 N 次方（cap <<=1），cap 的初始值为 1，所以 HashEntry 最小的容量为 2

# put 操作

对于 ConcurrentHashMap 的数据插入，这里要进行两次 Hash 去定位数据的存储位置

```
1   static class Segment<K,V> extends ReentrantLock implements Serializable {
```

从上 Segment 的继承体系可以看出，Segment 实现了 ReentrantLock,也就带有锁的功能，当执行 put 操作时，会进行第一次 key 的 hash 来定位 Segment 的位置，如果该 Segment 还没有初始化，即通过 CAS 操作进行赋值，然后进行第二次 hash 操作，找到相应的 HashEntry 的位置，这里会利用继承过来的锁的特性，在将数据插入指定的 HashEntry 位置时（链表的尾端），会通过继承 ReentrantLock 的 tryLock（）方法尝试去获取锁，如果获取成功就直接插入相应的位置，如果已经有线程获取该 Segment 的锁，那当前线程会以自旋的方式去继续的调用 tryLock（）方法去获取锁，超过指定次数就挂起，等待唤醒。

## get 操作

ConcurrentHashMap 的 get 操作跟 HashMap 类似，只是 ConcurrentHashMap 第一次需要经过一次 hash 定位到 Segment 的位置，然后再 hash 定位到指定的 HashEntry，遍历该 HashEntry 下的链表进行对比，成功就返回，不成功就返回 null。

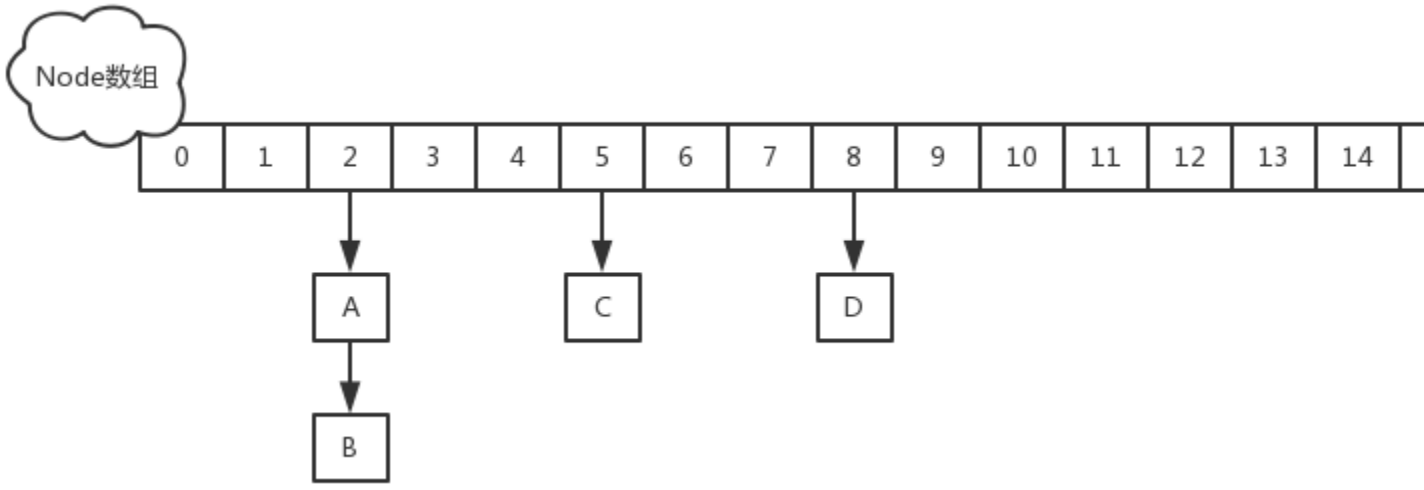
## size 操作

计算 ConcurrentHashMap 的元素大小是一个有趣的问题，因为他是并发操作的，就是在你计算 size 的时候，他还在并发的插入数据，可能会导致你计算出来的 size 和你实际的 size 有相差（在你 return size 的时候，插入了多个数据），要解决这个问题，JDK1.7 版本用两种方案。

1. 第一种方案他会使用不加锁的模式去尝试多次计算 ConcurrentHashMap 的 size，最多三次，比较前后两次计算的结果，结果一致就认为当前没有元素加入，计算的结果是准确的；
2. 第二种方案是如果第一种方案不符合，他就会给每个 Segment 加上锁，然后计算 ConcurrentHashMap 的 size 返回。

## JDK1.8 的实现

JDK1.8 的实现已经摒弃了 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，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。



在深入 JDK1.8 的 put 和 get 实现之前要知道一些常量设计和数据结构，这些是构成 ConcurrentHashMap 实现结构的基础，下面看一下基本属性：

```
1  // node 数组最大容量：2^30=1073741824
2  private static final int MAXIMUM_CAPACITY = 1 << 30;
3  // 默认初始值，必须是 2 的幕数
4  private static final int DEFAULT_CAPACITY = 16;
5  //数组可能最大值，需要与 toArray（）相关方法关联
6  static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
7  //并发级别，遗留下来的，为兼容以前的版本
8  private static final int DEFAULT_CONCURRENCY_LEVEL = 16;
9  // 负载因子
10 private static final float LOAD_FACTOR = 0.75f;
11 // 链表转红黑树阀值,> 8 链表转换为红黑树
12 static final int TREEIFY_THRESHOLD = 8;
13 //树转链表阀值，小于等于 6（tranfer 时，lc、hc=0 两个计数器分别++记录原 bin、新 binTreeNode 数量，<
14 则 untreeify(lo)）
15 static final int UNTREEIFY_THRESHOLD = 6;
16 static final int MIN_TREEIFY_CAPACITY = 64;
17 private static final int MIN_TRANSFER_STRIDE = 16;
18 private static int RESIZE_STAMP_BITS = 16;
19 // 2^15-1，help resize 的最大线程数
20 private static final int MAX_RESIZERS = (1 << (32 - RESIZE_STAMP_BITS)) - 1;
21 // 32-16=16，sizeCtl 中记录 size 大小的偏移量
22 private static final int RESIZE_STAMP_SHIFT = 32 - RESIZE_STAMP_BITS;
23 // forwarding nodes 的 hash 值
24 static final int MOVED     = -1;
25 // 树根节点的 hash 值
26 static final int TREEBIN   = -2;
```

```
// ReservationNode 的 hash 值
static final int RESERVED  = -3;
// 可用处理器数量
static final int NCPU = Runtime.getRuntime().availableProcessors();
//存放 node 的数组
transient volatile Node<K,V>[] table;
/*控制标识符，用来控制 table 的初始化和扩容的操作，不同的值有不同的含义
 *当为负数时：-1 代表正在初始化，-N 代表有 N-1 个线程正在 进行扩容
 *当为 0 时：代表当时的 table 还没有被初始化
 *当为正数时：表示初始化或者下一次进行扩容的大小
private transient volatile int sizeCtl;
```

基本属性定义了 ConcurrentHashMap 的一些边界以及操作时的一些控制，下面看一些内部的一些结构组成，这些是整个 ConcurrentHashMap 整个数据结构的核心。

# Node

Node 是 ConcurrentHashMap 存储结构的基本单元，继承于 HashMap 中的 Entry，用于存储数据，源代码如下

```
static class Node<K,V> implements Map.Entry<K,V> {
    //链表的数据结构
    final int hash;
    final K key;
    //val 和 next 都会在扩容时发生变化，所以加上 volatile 来保持可见性和禁止重排序
    volatile V val;
    volatile Node<K,V> next;
    Node(int hash, K key, V val, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.val = val;
```

```
11          this.next = next;
12      }
13      public final K getKey()        { return key; }
14      public final V getValue()      { return val; }
15      public final int hashCode()    { return key.hashCode() ^
val.hashCode(); }
16      public final String toString(){ return key + "=" + val; }
17      //不允许更新 value
18      public final V setValue(V value) {
19          throw new UnsupportedOperationException();
20      }
21      public final boolean equals(Object o) {
22          Object k, v, u; Map.Entry<?,?> e;
23          return ((o instanceof Map.Entry) &&
24                  (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&
25                  (v = e.getValue()) != null &&
26                  (k == key || k.equals(key)) &&
27                  (v == (u = val) || v.equals(u)));
28      }
29      //用于 map 中的 get（）方法，子类重写
30      Node<K,V> find(int h, Object k) {
31          Node<K,V> e = this;
32          if (k != null) {
33              do {
34                  K ek;
35                  if (e.hash == h &&
36                      ((ek = e.key) == k || (ek != null && k.equals(ek))))
37                      return e;
38              } while ((e = e.next) != null);
          }
          return null;
```

```
39          }
40      }
41
42
43
```

Node 数据结构很简单，从上可知，就是一个链表，但是只允许对数据进行查找，不允许进行修改。

## TreeNode

TreeNode 继承与 Node，但是数据结构换成了二叉树结构，它是红黑树的数据的存储结构，用于红黑树中存储数据，当链表的节点数大于 8 时会转换成红黑树的结构，他就是通过 TreeNode 作为存储结构代替 Node 来转换成黑红树源代码如下。

## TreeBin

TreeBin 从字面含义中可以理解为存储树形结构的容器，而树形结构就是指 TreeNode，所以 TreeBin 就是封装 TreeNode 的容器，它提供转换黑红树的一些条件和锁的控制，部分源码结构如下。

介绍了 ConcurrentHashMap 主要的属性与内部的数据结构，现在通过一个简单的例子以 debug 的视角看看 ConcurrentHashMap 的具体操作细节。

我们先通过 new ConcurrentHashMap()来进行初始化

```
1   public ConcurrentHashMap() {
2   }
```

由上你会发现 ConcurrentHashMap 的初始化其实是一个空实现，并没有做任何事，这里后面会讲到，这也是和其他的集合类有区别的地方，初始化操作并不是在构造函数实现的，而是在 put 操作中实现，当然 ConcurrentHashMap 还提供了其他的构造函数，有指定容量大小或者指定负载因子，跟 HashMap 一样，这里就不做介绍了。

## put 操作

在上面的例子中我们新增个人信息会调用 put 方法，我们来看下。

这个 put 的过程很清晰，对当前的 table 进行无条件自循环直到 put 成功，可以分成以下六步流程来概述。

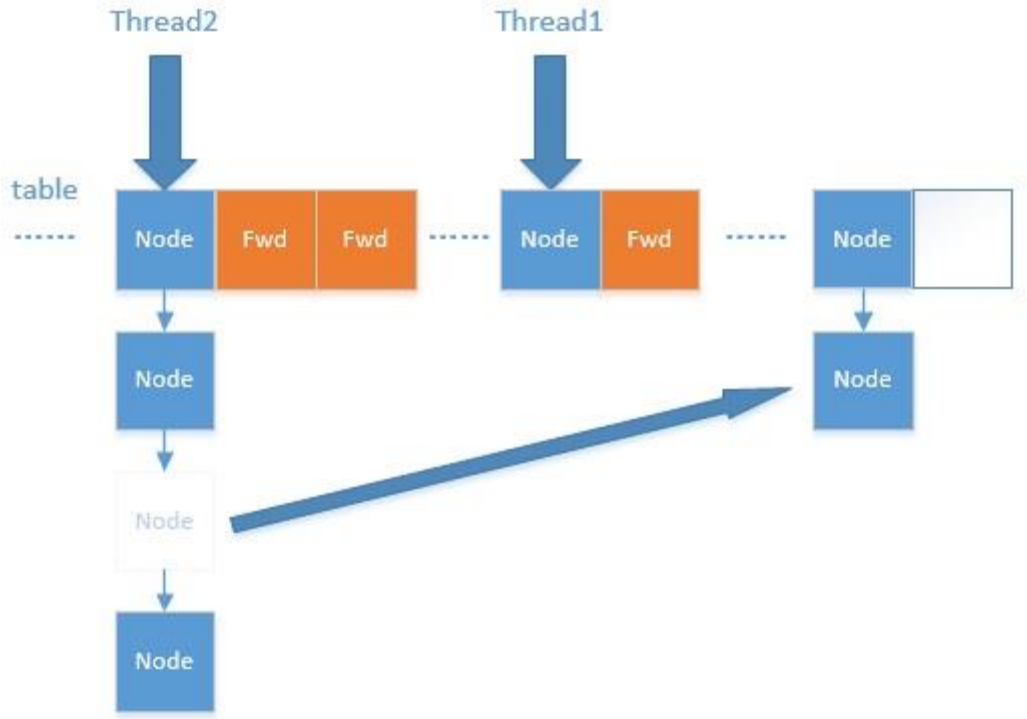
1. 如果没有初始化就先调用 initTable（）方法来进行初始化过程
2. 如果没有 hash 冲突就直接 CAS 插入
3. 如果还在进行扩容操作就先进行扩容
4. 如果存在 hash 冲突，就加锁来保证线程安全，这里有两种情况，一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入，
5. 最后一个如果该链表的数量大于阈值 8，就要先转换成黑红树的结构，break 再一次进入循环
6. 如果添加成功就调用 addCount（）方法统计 size，并且检查是否需要扩容
   现在我们来对每一步的细节进行源码分析，在第一步中，符合条件会进行初始化操作，我们来看看 initTable（）方法

   在第二步中没有 hash 冲突就直接调用 Unsafe 的方法 CAS 插入该元素，进入第三步如果容器正在扩容，则会调用 helpTransfer（）方法帮助扩容，现在我们跟进 helpTransfer（）方法看看

   其实 helpTransfer（）方法的目的就是调用多个工作线程一起帮助进行扩容，这样的效率就会更高，而不是只有检查到要扩容的那个线程进行扩容操作，其他线程就要等待扩容操作完成才能工作。

   既然这里涉及到扩容的操作，我们也一起来看看扩容方法 transfer（）

   扩容过程有点复杂，这里主要涉及到多线程并发扩容,ForwardingNode 的作用就是支持扩容操作，将已处理的节点和空节点置为 ForwardingNode，并发处理时多个线程经过 ForwardingNode 就表示已经遍历了，就往后遍历，下图是多线程合作扩容的过程：




介绍完扩容过程，我们再次回到 put 流程，在第四步中是向链表或者红黑树里加节点，到第五步，会调用 treeifyBin（ ）方法进行链表转红黑树的过程。

到第六步表示已经数据加入成功了，现在调用 addCount()方法计算 ConcurrentHashMap 的 size，在原来的基础上加一，现在来看看 addCount()方法。

put 的流程现在已经分析完了，你可以从中发现，他在并发处理中使用的是乐观锁，当有冲突的时候才进行并发处理，而且流程步骤很清晰，但是细节设计的很复杂，毕竟多线程的场景也复杂。

## get 操作

我们现在要回到开始的例子中，我们对个人信息进行了新增之后，我们要获取所新增的信息，使用 String name = map.get(“name”)获取新增的 name 信息，现在我们依旧用 debug 的方式来分析下 ConcurrentHashMap 的获取方法 get()

ConcurrentHashMap 的 get 操作的流程很简单，也很清晰，可以分为三个步骤来描述

1. 计算 hash 值，定位到该 table 索引位置，如果是首节点符合就返回

2. 如果遇到扩容的时候，会调用标志正在扩容节点 ForwardingNode 的 find 方法，查找该节点，匹配就返回
3. 以上都不符合的话，就往下遍历节点，匹配就返回，否则最后就返回 null

## size 操作

最后我们来看下例子中最后获取 size 的方式 int size = map.size();，现在让我们看下 size()方法

在 JDK1.8 版本中，对于 size 的计算，在扩容和 addCount()方法就已经有处理了，JDK1.7 是在调用 size()方法才去计算，其实在并发集合中去计算 size 是没有多大的意义的，因为 size 是实时在变的，只能计算某一刻的大小，但是某一刻太快了，人的感知是一个时间段，所以并不是很精确。

### 总结与思考

其实可以看出 JDK1.8 版本的 ConcurrentHashMap 的数据结构已经接近 HashMap，相对而言，ConcurrentHashMap 只是增加了同步的操作来控制并发，从 JDK1.7 版本的 ReentrantLock+Segment+HashEntry，到 JDK1.8 版本中 synchronized+CAS+HashEntry+红黑树,相对而言，总结如下思考：

1. JDK1.8 的实现降低锁的粒度，JDK1.7 版本锁的粒度是基于 Segment 的，包含多个 HashEntry，而 JDK1.8 锁的粒度就是 HashEntry（首节点）
2. JDK1.8 版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用 synchronized 来进行同步，所以不需要分段锁的概念，也就不需要 Segment 这种数据结构了，由于粒度的降低，实现的复杂度也增加了
3. JDK1.8 使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档
4. JDK1.8 为什么使用内置锁 synchronized 来代替重入锁 ReentrantLock，我觉得有以下几点：

- 因为粒度降低了，在相对而言的低粒度加锁方式，synchronized 并不比 ReentrantLock 差，在粗粒度加锁中 ReentrantLock 可能通过 Condition 来控制各个低粒度的边界，更加的灵活，而在低粒度中，Condition 的优势就没有了
- JVM 的开发团队从来都没有放弃 synchronized，而且基于 JVM 的 synchronized 优化空间更大，使用内嵌的关键字比使用 API 更加自然
- 在大量的数据操作下，对于 JVM 的内存压力，基于 API 的 ReentrantLock 会开销更多的内存，虽然不是瓶颈，但是也是一个选择依据

# 虚拟机

## Java 内存区域与内存溢出异常

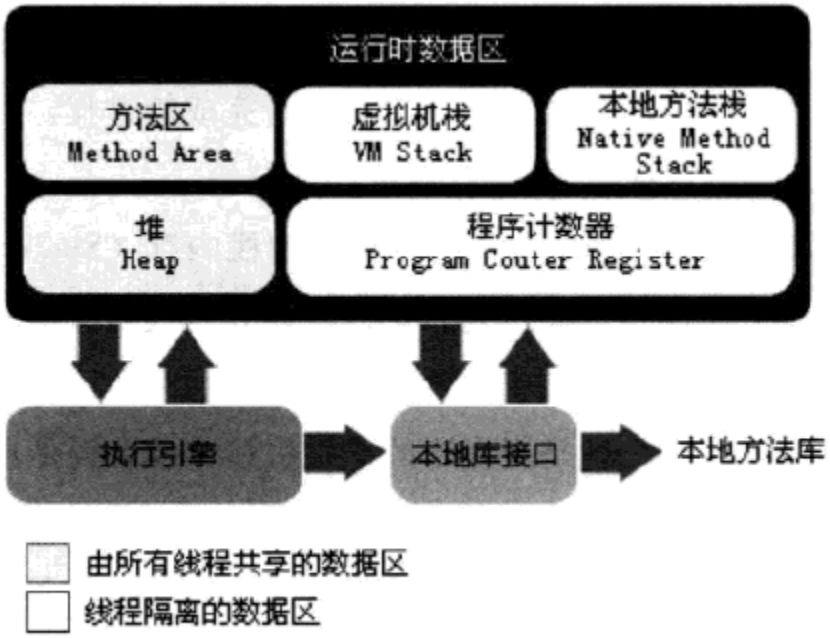



图 2-1　Java 虚拟机运行时数据区

### 程序计数器

当前线程所执行的字节码的行号指示器，由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器。如果正在执行的是 Native 方法，这个计数器值则为空。

### 虚拟机栈

生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧用于存储局部变量表，操作栈，动态链接，方法出口等信息。每一个方法被调用直至执行完成的过程，就对应一个栈帧在虚拟机栈中从入栈到出栈的过程。局部变量表存放了编译器可知的各种基本数据类型（boolean，byte，char），对象引用和 returnAddress 类型。**局部变量表所需的内存空间在编译器时间完成分配。**在 java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常，如果虚拟机栈可以动态扩展，当扩展无法申请到足够的内存时会抛出 OutOfMememoryError 异常。

### 本地方法栈

执行本地方法

### Java 堆

此内存区域的唯一目的就是存放对象实例，堆逻辑连续即可

## 方法区

存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。这个区域同样存在针对常量池的回收和堆类型的卸载。

运行时常量池是方法区的一部分。Class 文件中除了有类的版本，字段，方法，接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中

对象访问

对于 Object obj=new Object();

Object obj 将反应到 Java 栈的本地变量表中，作为一个 reference 类型数据出现，而 new Object() 这部分将会反映到 Java 堆中。另外，在 Java 堆中还必须包含能查找到此对象类型数据（如对象类型，父类，实现的接口，方法等）的地址信息信息，这些类型数据则 存储在方法区中。

不同虚拟机实现的对象访问方式会有不同，主流的访问方式有两种：使用句柄和直接指针

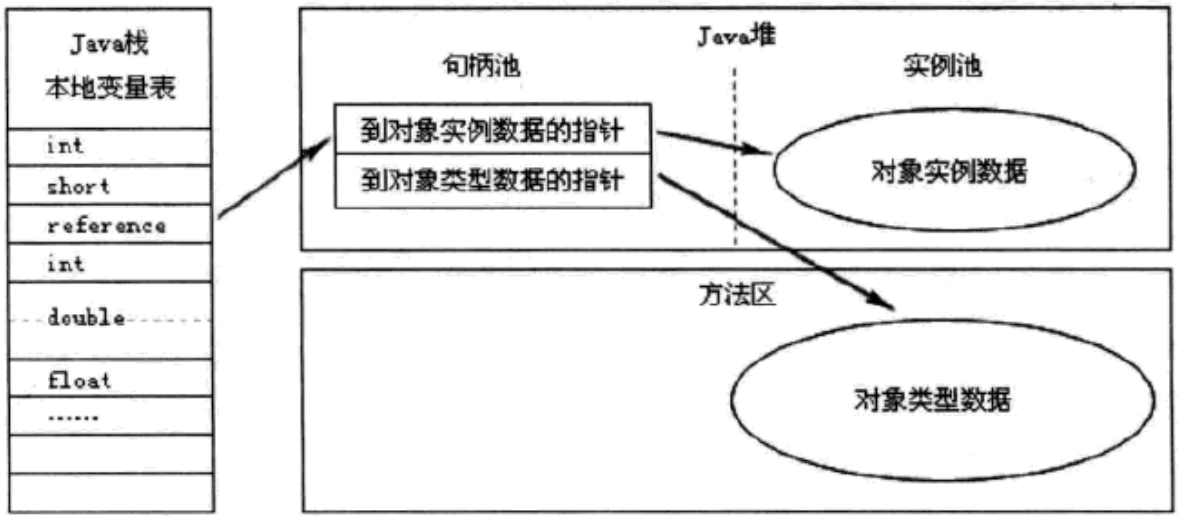



图 2-2　通过句柄访问对象

❑ 如果使用直接指针访问方式，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中直接存储的就是对象地址，如图 2-3 所示。

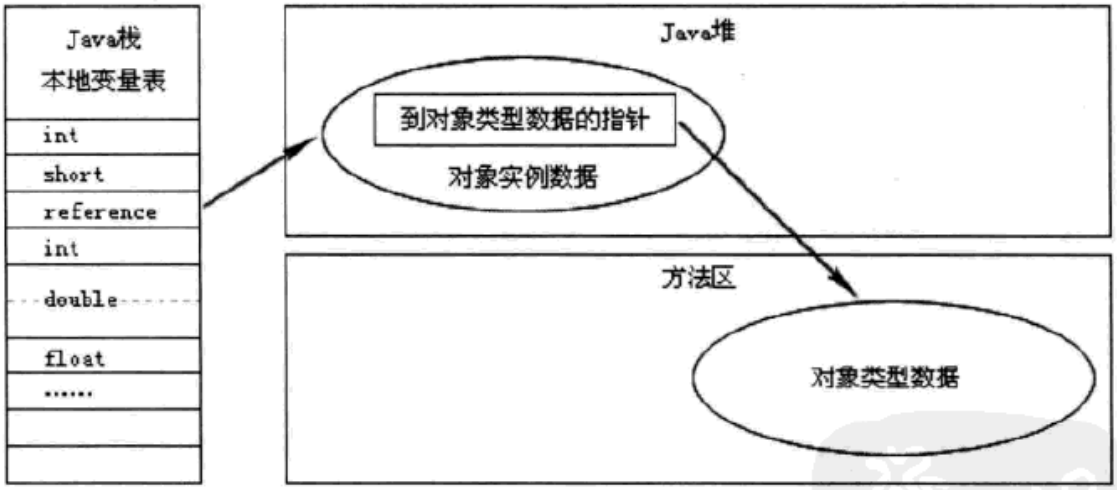



图 2-3　通过直接指针访问对象

句柄访问方式的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要要被修改。

使用指针快，节省了一次指针定位的时间开销

## GC 回收

概述

- 哪些内存需要回收?
- 什么时候回收?
- 如何回收?

把时间从半个世纪以前拨回到现在，回到我们熟悉的 Java 语言。我们上篇介绍了 Java 内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈三个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的（尽管在运行期会由 JIT 编译器进行一些优化，但在本章基于概念模型的讨论中，大体上可以认为是编译期可知的），因此这几个区域的内存分配和回收都具备确定性，在这几个区域内不需要过多考虑回收的问题，因为方法结束或线程结束时，内存自然就跟随着回收了。

而 Java 堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道

会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存，后续讨论中的“内存”分配与回收也仅指这一部分内存。

## 基本流程

JVM 堆相关知识

为什么先说 JVM 堆?

JVM 的堆是 Java 对象的活动空间，程序中的类的对象从中分配空间，其存储着正在运行着的应用程序用到的所有对象。这些对象的建立方式就是那些 new 一类的操作，当对象无用后，是 GC 来负责这个无用的对象(地球人都知道)。

JVM 堆

(1) 新域：存储所有新成生的对象

(2) 旧域：新域中的对象，经过了一定次数的 GC 循环后，被移入旧域

(3)永久域：存储类和方法对象，从配置的角度看，这个域是独立的，不包括在 JVM 堆内。默认为 4M。

新域会被分为 3 个部分：1.第一个部分叫 Eden。(伊甸园？？可能是因为亚当和夏娃是人类最早的活动对象？)2.另两个部分称为辅助生存空间(幼儿园)，我这里一个称为 A 空间(From sqace)，一个称为 B 空间(To Space)。

## 二、GC 浅谈

GC 的工作目的很明确：在堆中，找到已经无用的对象，并把这些对象占用的空间收回使其可以重新利用.大多数垃圾回收的 算法思路都是一致的：把所有对象组成一个集合，或可以理解为树状结构，从树根开始找，只要可以找到的都是活动对象，如果找不到，这个对象就是凋零的昨日黄 花，应该被回收了。

在 sun 的文档说明中，对 JVM 堆的新域，是采用 coping 算法，该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成一个对象面和多个 空闲面，程序从对象面为对象分配空间，当对象满了，基于 coping 算法的垃圾收集就从根集中扫描活动对象，并将每个活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面， 原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

对于新生成的对象，都放在 Eden 中；当 Eden 充满时（小孩太多 了），GC 将开始工作，首先停止应用程序的运行，开始收集垃圾，把所有可找到的对象都复制到 A 空间中，一旦当 A 空间充满，GC 就把在 A 空间中可找到的对象 都复制到 B 空间中(会覆盖原有的存储对象)，当 B 空间满的时间，GC 就把在 B 空间中可找到的对象都复制到 A 空间中，AB 在这个过程中互换角色，那位客官说 了：拷来拷去，烦不烦啊？什么时候是头？您别急，在活动对象经过一定次数的 GC 操作后，这些活动对象就会被放到旧域中。对于这些活动对象，新域的幼儿园生 活结束了。新域为什么要这么折腾？起初在这块我也很迷糊，又查了些资料，原来是这样：应用程序生成的绝大部分对象都是短命的，copying 算法最理想的 状态是，所有移出 Eden 的对象都会被收集，因为这些都是短命鬼，经过一定次数的 GC 后应该被收集，那么移入到旧域的对象都是长命的，这样可以防止 AB 空间的来回复制影响应用程序。实际上这种理想状态是很难达到的，应用程序中不可避免地存在长命的对象，copying 算法的发明者要这些对象都尽量放在新域 中，以保证小范围的复制，压缩旧域的开销可比新域中的复制大得多(旧域在下面说)。对于旧域，采用的是 tracing 算法的一种，称为标记-清除-压缩收 集器，注意，这有一个压缩，这是个开销挺大的操作。垃圾回收主要是对 Young Generation 块和 Old Generation 块内存进行回收，YG 用来放新产生的对象，经过几次回收还没回收掉的对象往 OG 中移动，对 YG 进行垃圾回收又叫做 MinorGC，对 OG 垃圾回收又叫 MajorGC，两块内存回收互不干涉。二、Gc 流程：

[older generation][survivor 1][survivor 2][eden]

*young generation=eden + survivor

1.当 eden 满了，触发 young GC；

2.young GC 做 2 件事：一，去掉一部分没用的 object；二，把老的还被引用的 object 发到 survior 里面，等下几次 GC 以后，survivor 再放到 old 里面。

3.当 old 满了，触发 full GC。full GC 很消耗内存，把 old，young 里面大部分垃圾回收掉。这个时候用户线程都会被 block。

## 回收算法

**引用计数算法**

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器的值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器都为 0 的对象就是不可能再被使用的。很难解决对象之间的相互循环引用的问题

**根搜索算法**

通过一系列名为 GC Roots 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到 GC Roots 没有任何引用链相连，则证明此对象是不可用的。根搜索算法中不可达的对象也并非是‘非死不可’的，暂时是‘缓刑’阶段，要真正判断一个对象死亡要经历两次标记过程：如果对象在进行根搜索后发现对象不可达，那它将会进行被第一次标记并且进行筛选，筛选的条件是此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法或者 finalize()方法已经被虚拟机掉用过，这两种情况都视为‘没有必要执行’。

如果对象被认为有必要执行 finalize()方法，那么这个方法会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动建立的、低优先级的 Finalizer 线程去执行。这里的‘执行’也只是指虚拟机会触发这个方法，但并不承诺一定会执行。

finalize()方法是对象逃脱死亡命运的最后一次机会，稍后 GC 会对 F-Queue 中的对象进行第二次小规模的标记，如果对象在 finalize()中重新与引用链上的任何一个对象建立了关联，就会被移出‘即将回收’集合，如果没有移出，那就真的离死亡不远了。

finalize()方法只会被系统自动调用一次。

**GC Roots**

Java 中可以被作为 GC Roots 中的对象有：

虚拟机栈（栈帧中的本地变量表）中的引用的对象。
方法区中的类静态属性引用的对象。
方法区中的常量引用的对象。
本地方法栈（jni）即一般说的 Native 的引用对象。

**引用：**

强引用：普遍存在的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象

软引用：描述一些还有用，但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行第二次回收

弱引用：被弱引用关联的对象只能生存到下一次垃圾收集发生之前

虚引用：一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被回收器回收时收到一个系统通知。

**标记清除**

标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。在标记阶段首先通过根节点，标记所有从根节点开始的对象，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。标记清除算法带来的一个问题是会存在大量的空间碎片，因为回收后的空间是不连续的，这样给大对象分配内存的时候可能会提前触发 full gc。

**复制算法**

将现有的内存空间分为两快，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM 研究表明新生代中的对象 98%是朝夕生死的，所以并不需要按照 1:1 的比例划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中的一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地拷贝到另外一个 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 的空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1(可以通过-SurvivorRattio 来配置)，也就是每次新生代中可用内存空间为整个新生代容量的 90%，只有 10%的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证回收都只有不多于 10%的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保。

**标记整理**

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在新生代经常发生，但是在老年代更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活的对象较多，复制的成本也将很高。

标记-压缩算法是一种老年代的回收算法，它在标记-清除算法的基础上做了一些优化。首先也需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

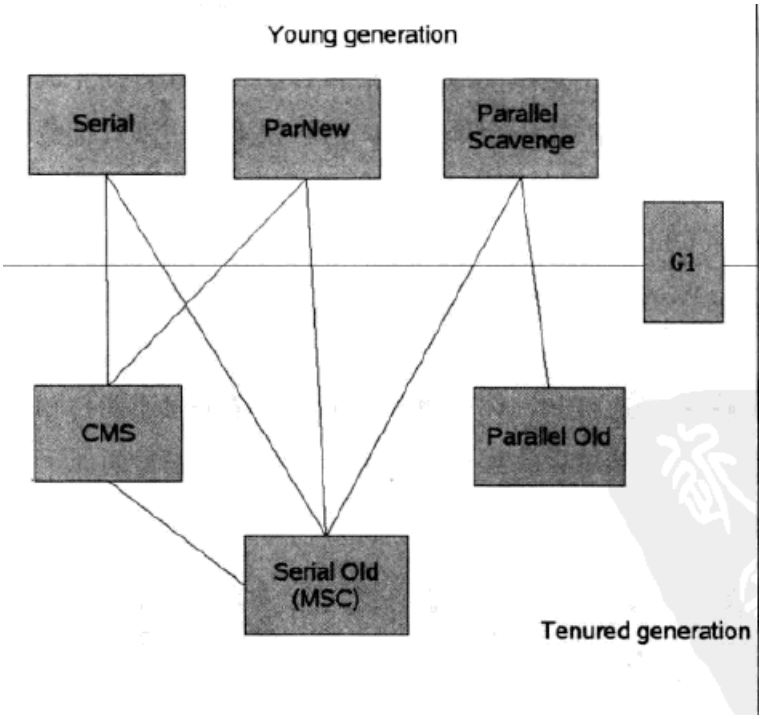
**增量算法**

增量算法的基本思想是，如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序线程交替执行。每次，垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间。但是，因为线程切换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降。

**分代收集算法**

根据对象的存活周期的不同将内存分为几块。一是把 Java 堆分成新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高，没有额外空间对它进行分配担保，就必须使用标记清理或标记整理算法来进行回收

## 垃圾回收器




**Serial 收集器**

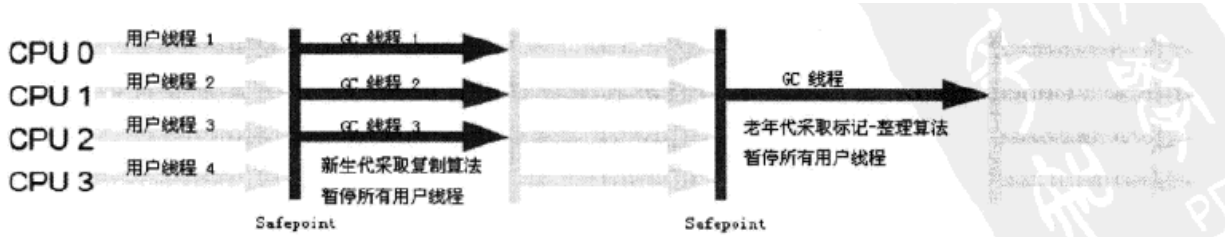
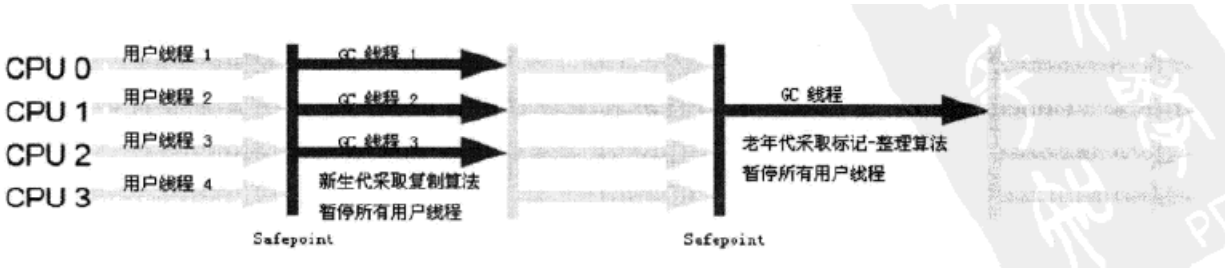



图 3-7　ParNew / Serial Old 收集器运行示意图

Serial 收集器是最古老的收集器，它的缺点是当 Serial 收集器想进行垃圾回收的时候，必须暂停用户的所有进程，即 stop the world。到现在为止，它依然是虚拟机运行在 client 模式下的默认新生代收集器，与其他收集器相比，对于限定在单个 CPU 的运行环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾回收自然可以获得最高的单线程收集效率。

Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用标记 - 整理“算法。

**ParNew 收集器**




**其实还是要暂停所有用户线程**

ParNew 收集器是 Serial 收集器新生代的**多线程实现**，注意在进行垃圾回收的时候依然会 stop the world，只是相比较 Serial 收集器而言它会运行多条进程进行垃圾回收。

ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百的保证能超越 Serial 收集器。当然，随着可以使用的 CPU 的数量增加，它对于 GC 时系统资源的利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多（譬如 32 个，现在 CPU 动辄 4 核加超线程，服务器超过 32 个逻辑 CPU 的情况越来越多了）的环境下，可以使用-XX:ParallelGCThreads 参数来限制垃圾收集的线程数。

**Parallel Scavenge 收集器**

Parallel 是采用复制算法的多线程新生代垃圾回收器，似乎和 ParNew 收集器有很多的相似的地方。但是 Parallel Scanvenge 收集器的一个特点是它所关注的目标是吞吐量(Throughput)。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间 / (运行用户代码时间 + 垃圾收集时间)。停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能够提升用户的体验；而高吞吐量则可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Old 收集器是 Parallel Scavenge 收集器的老年代版本，采用多线程和”标记 - 整理”算法。这个收集器是在 jdk1.6 中才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于比较尴尬的状态。原因是如果新生代 Parallel Scavenge 收集器，那么老年代除了 Serial Old(PS MarkSweep)收集器外别无选择。由于单线程的老年代 Serial Old 收集器在服务端应用性能上的”拖累“，即使使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，又因为老年代收集中无法充分利用服务器多 CPU 的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合”给力“。直到 Parallel Old 收集器出现

后，”吞吐量优先“收集器终于有了比较名副其实的应用祝贺，在注重吞吐量及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

-UseParallelGC: 虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old 的收集器组合进行内存回收。-UseParallelOldGC: 打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行垃圾回收

**CMS 收集器**

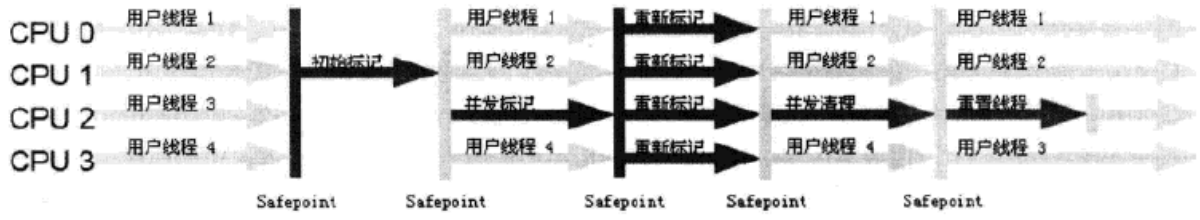



图 3-10　Concurrent Mark Sweep 收集器运行示意图

CMS(Concurrent Mark Swep)收集器是一个比较重要的回收器，现在应用非常广泛，我们重点来看一下，CMS 一种**获取最短回收停顿时间为目标的收集器**，这使得它很适合用于和用户交互的业务。从名字(Mark Swep)就可以看出，CMS 收集器是基于标记清除算法实现的。它的收集过程分为四个步骤：

1. 初始标记(initial mark)
2. 并发标记(concurrent mark)
3. 重新标记(remark)
4. 并发清除(concurrent sweep)
   注意初始标记和重新标记还是会 stop the world，但是在耗费时间更长的并发标记和并发清除两个阶段都可以和用户进程同时工作。

缺点：

1.CMS 收集器对 CPU 资源 非常敏感（并发的缺点）

2.CMS 收集器无法处理浮动垃圾(Floating Garbage)

CMS 收集器无法处理浮动垃圾(Floating Garbage),可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。由于 CMS 并发清理阶段用户线程还在运行着,伴随程序运行自然就还会有新的垃圾不断产生,这一部分垃圾出现在标记过程之后,CMS 无法在当次收集中处理掉它们,只好留待下一次 GC 时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需

要运行,那也就还需要预留有足够的内存空间给用户线程使用,因此 CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集,需要预留一部分空间提供并发收集时的程序运作使用。在 JDK 1.5 的默认设置下,CMS 收集器当老年代使用了 68%的空间后就会被激活,这是一个偏保守的设置,如果在应用中老年代增长不是太快,可以适当调高参数-XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比,以便降低内存回收次数从而获取更好的性能,在 JDK 1.6 中,CMS 收集器的启动阈值已经提升至 92%。要是 CMS 运行期间预留的内存无法满足程序需要,就会出现一次“Concurrent Mode Failure”失败,这时虚拟机将启动后备预案:临时启用 Serial Old 收集器来重新进行老年代的垃圾收集,这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction 设置得太高很容易导致大量“Concurrent Mode Failure”失败,性能反而降低。

3.CMS 收集器会产生大量的空间碎片

CMS 是一款基于“标记—清除”算法实现的收集器,这意味着收集结束时会有大量 空间碎片产生。空间碎片过多时,将会给大对象分配带来很大麻烦,往往会出现老年代还有很大空间剩余,但是无法找到足够大的连续空间来分配当前对象,不得不提前触发一次 Full GC。为了解决这个问题,CMS 收集器提供了一个-XX:+UseCMSCompactAtFullCollection 开关参数(默认就是开启的),用于在 CMS 收集器顶不住要进行 Full GC 时开启内存碎片的合并整理过程,内存整理的过程是无法并发的,空间碎片问题没有了,但停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction,这个参数是用于设置执行多少次不压缩的 Full GC 后,跟着来一次带压缩的(默认值为 0,表示每次进入 Full GC 时都进行碎片整理)。

**G1 收集器**

G1 收集器是一款面向服务端应用的垃圾收集器。HotSpot 团队赋予它的使命是在未来替换掉 JDK1.5 中发布的 CMS 收集器。与其他 GC 收集器相比，G1 具备如下特点：

1. 并行与并发：G1 能更充分的利用 CPU，多核环境下的硬件优势来缩短 stop the world 的停顿时间。
2. 分代收集：和其他收集器一样，分代的概念在 G1 中依然存在，不过 G1 不需要其他的垃圾回收器的配合就可以独自管理整个 GC 堆。
3. 空间整合：G1 收集器有利于程序长时间运行，分配大对象时不会无法得到连续的空间而提前触发一次 GC。
4. 可预测的非停顿：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒。

在使用 G1 收集器时，Java 堆的内存布局和其他收集器有很大的差别，它将这个 Java 堆分为多个大小相等的独立区域，虽然还保留新生代和老年代的概念，但是新生代和老年代不再是物理隔离的了，它们都是一部分 Region（不需要连续）的集合。

虽然 G1 看起来有很多优点，实际上 CMS 还是主流。

## 内存溢出的种类

主要有三种类型
第一种 OutOfMemoryError： PermGen space
发生这种问题的原意是程序中使用了大量的 jar 或 class，使 java 虚拟机装载类的空间不够，与 Permanent Generation space 有关。解决这类问题有以下两种办法：
1. 增加 java 虚拟机中的 XX:PermSize 和 XX:MaxPermSize 参数的大小，其中 XX:PermSize 是初始永久保存区域大小，XX:MaxPermSize 是最大永久保存区域大小。如针对 tomcat6.0，在 catalina.sh 或 catalina.bat 文件中一系列环境变量名说明结束处（大约在 70 行左右） 增加一行：
JAVA_OPTS=" -XX:PermSize=64M -XX:MaxPermSize=128m"
如果是 windows 服务器还可以在系统环境变量中设置。感觉用 tomcat 发布 sprint+struts+hibernate 架构的程序时很容易发生这种内存溢出错误。使用上述方法，我成功解决了部署 ssh 项目的 tomcat 服务器经常宕机的问题。
2. 清理应用程序中 web-inf/lib 下的 jar，如果 tomcat 部署了多个应用，很多应用都使用了相同的 jar，可以将共同的 jar 移到 tomcat 共同的 lib 下，减少类的重复加载。

第二种 OutOfMemoryError： Java heap space
发生这种问题的原因是 java 虚拟机创建的对象太多，在进行垃圾回收之间，虚拟机分配的到堆内存空间已经用满了，与 Heap space 有关。解决这类问题有两种思路：
1. 检查程序，看是否有死循环或不必要地重复创建大量对象。找到原因后，修改程序和算法。我以前写一个使用 K-Means 文本聚类算法对几万条文本记录（每条记录的特征向量大约 10 来个）进行文本聚类时，由于程序细节上有问题，就导致了 Java heap space 的内存溢出问题，后来通过修改程序得到了解决。
2. 增加 Java 虚拟机中 Xms（初始堆大小）和 Xmx（最大堆大小）参数的大小。如：set JAVA_OPTS= -Xms256m -Xmx1024m

第三种 OutOfMemoryError：unable to create new native thread
这种错误在 Java 线程个数很多的情况下容易发生

## 加载一个 class 的过程

理解类在 JVM 中什么时候被加载和初始化是 Java 编程语言中的基础概念，正因为有了 Java 语言规范，我们才可以清晰的记录和解释这个问题，但是很多 Java 程序员仍然不知道什么时候类被加载，什么时候类被初始化，类加载和初始化好像让人很困惑，对初学者难以理解，在这篇教程中我们将看看类加载什么时候发生，类和接口是如何被初始化的，我并不会拘泥于类加载器的细节或者说类加载器的工作方式。仅仅使这篇文章更加专注和简结。

**类什么时候加载**

类的加载是通过类加载器（Classloader）完成的，它既可以是饿汉式[eagerly load]（只要有其它类引用了它就加载）加载类，也可以是懒加载[lazy load]（等到类初始化发生的时候才加载）。不过我相信这跟不同的 JVM 实现有关，然而他又是受 JLS 保证的（当有静态初始化需求的时候才被加载）。

**类什么时候初始化**

加载完类后，类的初始化就会发生，意味着它会初始化所有类静态成员，以下情况一个类被初始化：

1. 实例通过使用 new()关键字创建或者使用 class.forName()反射，但它有可能导致 ClassNotFoundException。
2. 类的静态方法被调用
3. 类的静态域被赋值
4. 静态域被访问，而且它不是常量
5. 在顶层类中执行 assert 语句

反射同样可以使类初始化，比如 java.lang.reflect 包下面的某些方法，JLS 严格的说明：一个类不会被任何除以上之外的原因初始化。

**类是如何被初始化的**

现在我们知道什么时候触发类的初始化了，他精确地写在 Java 语言规范中。但了解清楚 域（fields，静态的还是非静态的）、块（block 静态的还是非静态的）、不同类（子类和超类）和不同的接口（子接口，实现类和超接口）的初始化顺序也很重要类。事实上很多核心 Java 面试题和 SCJP 问题都是基于这些概念，下面是类初始化的一些规则：

1. 类从顶至底的顺序初始化，所以声明在顶部的字段的早于底部的字段初始化
2. 超类早于子类和衍生类的初始化
3. 如果类的初始化是由于访问静态域而触发，那么只有声明静态域的类才被初始化，而不会触发超类的初始化或者子类的初始化即使静态域被子类或子接口或者它的实现类所引用。
4. 接口初始化不会导致父接口的初始化。
5. 静态域的初始化是在类的静态初始化期间，非静态域的初始化时在类的实例创建期间。这意味这静态域初始化在非静态域之前。
6. 非静态域通过构造器初始化，子类在做任何初始化之前构造器会隐含地调用父类的构造器，他保证了非静态或实例变量（父类）初始化早于子类

主动引用和被动引用

类的主动引用（一定会发生类的初始化）

new 一个类的对象。

调用类的静态成员（除了 final 常最）和静态方法。
使用 java.lang.reflect 包的方法对类进行反射调用。
当虚拟机启动，java Hello 则一定会初始化 Hello 类。说白了就是先启动 main 方法所在的类。
当初始化一个类，如果其父类没有被初始化，则先会初始化他的父类。

类的被动引用（不会发生类的初始化）
当访问一个静态域时，只有真正声明这个域的类才会被初始化。
通过子类引用父类的静态变量，不会导致子类初始化。
通过数组定义类引用，用不会触发此类的初始化。
引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）

### 初始化例子

这是一个有关类被初始化的例子，你可以看到哪个类被初始化

```
1  /**
2   * Java program to demonstrate class loading and initialization in Java.
3   */
4  public class ClassInitializationTest {
5
6      public static void main(String args[]) throws InterruptedException {
7
8          NotUsed o = null; //this class is not used, should not be initialized
9          Child t = new Child(); //initializing sub class, should trigger super class initialization
10         System.out.println((Object)o == (Object)t);
11     }
12 }
13
14 /**
15  * Super class to demonstrate that Super class is loaded and initialized before Subclass.
16  */
17 class Parent {
18     static { System.out.println("static block of Super class is initialized"); }
     {System.out.println("non static blocks in super class is initialized");}
```

```
}

/**
 * Java class which is not used in this program, consequently not loaded by JVM
 */
class NotUsed {
    static { System.out.println("NotUsed Class is initialized "); }
}

/**
 * Sub class of Parent, demonstrate when exactly sub class loading and
initialization occurs.
 */
class Child extends Parent {
    static { System.out.println("static block of Sub class is initialized in Java
"); }
    {System.out.println("non static blocks in sub class is initialized");}
}

Output:
static block of Super class is initialized
static block of Sub class is initialized in Java
non static blocks in super class is initialized
non static blocks in sub class is initialized
false
```

从上面结果可以看出：

1. 超类初始化早于子类
2. 静态变量或代码块初始化早于非静态块和域
3. 没使用的类根本不会被初始化，因为他没有被使用

再来看一个例子：

```
/**
 * Another Java program example to demonstrate class initialization and
loading in Java.
 */

public class ClassInitializationTest {

    public static void main(String args[]) throws InterruptedException {

        //accessing static field of Parent through child, should only
initialize Parent
        System.out.println(Child.familyName);
    }
}

class Parent {
    //compile time constant, accessing this will not trigger class
initialization
    //protected static final String familyName = "Lawson";

    protected static String familyName = "Lawson";

    static{ System.out.println("static block of Super class is
initialized"); }
    {System.out.println("non static blocks in super class is
initialized");}
}

Output:
static block of Super class is initialized
Lawson
```

分析：

1. 这里的初始化发生是因为有静态域被访问，而且不一个编译时常量。如果声明的”familyName“是使用 final 关键字修饰的编译时常量使用（就是上面的注释代码块部分）超类的初始化就不会发生。
2. 尽管静态与被子类所引用但是也仅仅是超类被初始化
   还有另外一个例子与接口相关的，JLS 清晰地解释子接口的初始化不会触发父接口的初始化。强烈推荐阅读 JLS14.4 理解类加载和初始化细节。以上所有就是有关类被初始化和加载的全部内容。

**1.加载：（重点）**

加载阶段是“类加载机制”中的一个阶段，这个阶段通常也被称作“装载”，主要完成：
1.通过“类全名”来获取定义此类的二进制字节流

2.将字节流所代表的静态存储结构转换为方法区的运行时数据结构

3.在 java 堆中生成一个代表这个类的 java.lang.Class 对象，作为方法区这些数据的访问入口

相对于类加载过程的其他阶段，加载阶段(准备地说，是加载阶段中获取类的二进制字节流的动作)是开发期可控性最强的阶段，因为加载阶段可以使用系统提供的类加载器(ClassLoader)来完成，也可以由用户自定义的类加载器完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式有虚拟机实现自行定义，虚拟机并未规定此区域的具体数据结构。然后在 java 堆中实例化一个 java.lang.Class 类的对象，这个对象作为程序访问方法区中的这些类型数据的外部接口。

**2.验证：（了解）**

验证是链接阶段的第一步，这一步主要的目的是确保 class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身安全。
验证阶段主要包括四个检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

1.文件格式验证

验证 class 文件格式规范，例如： class 文件是否已魔术 0xCAFEBABE 开头 ， 主、次版本号是否在当前虚拟机处理范围之内等

2.元数据验证

这个阶段是对字节码描述的信息进行语义分析，以保证起描述的信息符合 java 语言规范要求。验证点可能包括：这个类是否有父类(除了 java.lang.Object 之外，所有的类都应当有父类)、这个类是否继承了不允许被继承的类(被 final 修饰的)、如果这个类的父类是抽象类，是否实现了起父类或接口中要求实现的所有方法。

3.字节码验证

进行数据流和控制流分析，这个阶段对类的方法体进行校验分析，这个阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。如：保证访法体中的类型转换有效，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类数据类型、保证跳转命令不会跳转到方法体以外的字节码命令上。

4.符号引用验证

符号引用中通过字符串描述的全限定名是否能找到对应的类、符号引用类中的类，字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

**3.准备：（了解）**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的知识点，首先是这时候进行内存分配的仅包括类变量(static 修饰的变量),而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 java 堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量定义为:

public static int value = 12;

那么变量 value 在准备阶段过后的初始值为 0 而不是 12，因为这时候尚未开始执行任何 java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译后，存放于类构造器 <clinit>()方法之中，所以把 value 赋值为 12 的动作将在初始化阶段才会被执行。

上面所说的“通常情况”下初始值是零值，那相对于一些特殊的情况，如果类字段的字段属性表中存在 ConstantValue 属性，那在准备阶段变量 value 就会被初始化为 ConstantValue 属性所指定的值，建设上面类变量 value 定义为：

public static final int value = 123;

编译时 javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 设置为 123。

**4. 解析：（了解）**

解析阶段是虚拟机常量池内的符号引用替换为直接引用的过程。

符号引用：符号引用是一组符号来描述所引用的目标对象，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标对象并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标对象的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机内存布局实现相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范并没有规定解析阶段发生的具体时间，只要求了在执行 anewarry、checkcast、getfield、instanceof、invokeinterface、invokespecial、invokestatic、invokevirtual、multianewarray、new、putfield 和 putstatic 这 13 个用于操作符号引用的字节码指令之前，先对它们使用的符号引用进行解析，所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

解析的动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。分别对应编译后常量池内的 CONSTANT_Class_Info 、 CONSTANT_Fieldref_Info 、CONSTANT_Methodef_Info、CONSTANT_InterfaceMethoder_Info 四种常量类型。

1.类、接口的解析

2.字段解析

3.类方法解析

4.接口方法解析

**5.初始化：（了解）**

类的初始化阶段是类加载过程的最后一步，在准备阶段，类变量已赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。在以下四种情况下初始化过程会被触发执行：

1.遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需先触发其初始化。生成这 4 条指令的最常见的 java 代码场景是：使用 new 关键字实例化对象、读取或设置一个类的静态字段(被 final 修饰、已在编译器把结果放入常量池的静态字段除外)的时候，以及调用类的静态方法的时候。

2.使用 java.lang.reflect 包的方法对类进行反射调用的时候

3.当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先出发其父类的初始化

4.jvm 启动时，用户指定一个执行的主类(包含 main 方法的那个类)，虚拟机会先初始化这个类

在上面准备阶段 public static int value = 12; 在准备阶段完成后 value 的值为 0，而在初始化阶调用了类构造器<clinit>()方法，这个阶段完成后 value 的值为 12。

*类构造器<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static 块)中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句快可以赋值，但是不能访问。

*类构造器<clinit>()方法与类的构造函数(实例构造函数<init>()方法)不同，它不需要显式调用父类构造，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已

经执行完毕。因此在虚拟机中的第一个执行的<clinit>()方法的类肯定是java.lang.Object。

*由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句快要优先于子类的变量赋值操作。

*<clinit>()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句，也没有变量赋值的操作，那么编译器可以不为这个类生成<clinit>()方法。

*接口中不能使用静态语句块，但接口与类不太能够的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

*虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞。

# 前言

Java 虚拟机的内存模型分为两部分：一部分是线程共享的，包括 Java 堆和方法区；另一部分是线程私有的，包括虚拟机栈和本地方法栈，以及程序计数器这一小部分内存。今天我就 Java 虚拟机栈做一些比较浅的探究。

熟悉 Java 的同学应该都知道了，JVM 是基于栈的。但是这个“栈” 具体指的是什么？难道就是虚拟机栈？想要回答这个问题我们先要从虚拟机栈的结构谈起。

# 虚拟机栈

## 何为虚拟机栈

虚拟机栈的栈元素是栈帧，当有一个方法被调用时，代表这个方法的栈帧入栈；当这个方法返回时，其栈帧出栈。因此，虚拟机栈中栈帧的入栈顺序就是方法调用顺序。什么是栈帧呢？栈帧可以理解为一个方法的运行空间。它主要由两部分构成，一部分是局部变量表，方法中定义的局部变量以及方法的参数就存放在这张表中；另一部分是操作数栈，用来存放操作数。我们知道，Java 程序编译之后就变成了一条条字节码指令，其形式类似汇编，但和汇编有不同之处：汇编指令的操作数存放在数据段和寄存器中，可通过存储器或寄存器寻址找到需要的操作数；而 Java 字节码指令的操作数存放在操作数栈中，当执行某条带 n 个操作数的指令时，就从栈顶取 n 个操作数，然后把指令的计算结果（如果有的话）入栈。因此，当我们说 JVM 执行引擎是基于栈的时候，其中的“栈”指的就是操作数栈。举个简单的例子对比下汇编指令和 Java 字节码指令的执行过程，比如计算 1 + 2，在汇编指令是这样的：

```
1  mov ax, 1 ;把 1 放入寄存器 ax
2  add ax, 2 ;用 ax 的内容和 2 相加后存入 ax
```

而 JVM 的字节码指令是这样的：

```
1  iconst_1 //把整数 1 压入操作数栈
2  iconst_2 //把整数 2 压入操作数栈
3  iadd //栈顶的两个数相加后出栈，结果入栈
```

由于操作数栈是内存空间，所以字节码指令不必担心不同机器上寄存器以及机器指令的差别，从而做到了平台无关。

注意，局部变量表中的变量不可直接使用，如需使用必须通过相关指令将其加载至操作数栈中作为操作数使用。比如有一个方法 void foo()，其中的代码为：int a = 1 + 2; int b = a + 3;，编译为字节码指令就是这样的：

```
1  iconst_1 //把整数 1 压入操作数栈
2  iconst_2 //把整数 2 压入操作数栈
3  iadd //栈顶的两个数出栈后相加，结果入栈；实际上前三步会被编译器优化为：iconst_3
```

```
4  istore_1 //把栈顶的内容放入局部变量表中索引为 1 的 slot 中，也就是 a 对应的空间中
5  iload_1 // 把局部变量表索引为 1 的 slot 中存放的变量值（3）加载至操作数栈
6  iconst_3
7  iadd //栈顶的两个数出栈后相加，结果入栈
8  istore_2 // 把栈顶的内容放入局部变量表中索引为 2 的 slot 中，也就是 b 对应的空间中
9  return // 方法返回指令，回到调用点
```

需要说明的是，局部变量表以及操作数栈的容量的最大值在编译时就已经确定了，运行时不会改变。并且局部变量表的空间是可以复用的，例如，当指令的位置超出了局部变量表中某个变量 a 的作用域时，如果有新的局部变量 b 要被定义，b 就会覆盖 a 在局部变量表的空间。

盗用别人的图以让大家对虚拟机栈有个直观的认识（其中小字体 Stack 指的的是虚拟机栈，Frame 是栈帧，Local variables 是局部变量表，Operand Stack 是操作数栈）：

# Classloader

## JAVA 类加载种类

Java 语言系统自带有三个类加载器：

- **Bootstrap ClassLoader** 最顶层的加载类，主要加载核心类库，%JRE_HOME%\lib 下的 rt.jar、resources.jar、charsets.jar 和 class 等。另外需要注意的是可以通过启动 jvm 时指定-Xbootclasspath 和路径来改变 Bootstrap ClassLoader 的加载目录。比如 `java -Xbootclasspath/a:path` 被指定的文件追加到默认的 bootstrap 路径中。我们可以打开我的电脑，在上面的目录下查看，看看这些 jar 包是不是存在于这个目录。

lnt 什么的都是由这个类加载的

- **Extention ClassLoader** 扩展的类加载器，加载目录%JRE_HOME%\lib\ext 目录下的 jar 包和 class 文件。还可以加载`-D java.ext.dirs` 选项指定的目录。
- **Appclass Loader** 也称为 **SystemAppClass** 加载当前应用的 classpath 的所有类。是 extension classloader 的子类

我们上面简单介绍了 3 个 ClassLoader。说明了它们加载的路径。并且还提到了`-Xbootclasspath` 和`-D java.ext.dirs` 这两个虚拟机参数选项。

## 加载顺序

我们看到了系统的 3 个类加载器，但我们可能不知道具体哪个先行呢？
我可以先告诉你答案

1. Bootstrap CLassloder
2. Extention ClassLoader
3. AppClassLoader

## 加载 流程

1. 一个 AppClassLoader 查找资源时，先看看缓存是否有，缓存有从缓存中获取，否则委托给父加载器。
2. 递归，重复第 1 部的操作。
3. 如果 ExtClassLoader 也没有加载过，则由 Bootstrap ClassLoader 出面，它首先查找缓存，如果没有找到的话，就去找自己的规定的路径下，也就是 `sun.mic.boot.class` 下面的路径。找到就返回，没有找到，让子加载器自己去找。
4. Bootstrap ClassLoader 如果没有查找成功，则 ExtClassLoader 自己在 `java.ext.dirs` 路径中去查找，查找成功就返回，查找不成功，再向下让子加载器找。
5. ExtClassLoader 查找不成功，AppClassLoader 就自己查找，在 `java.class.path` 路径下查找。找到就返回。如果没有找到就让子类找，如果没有子类会怎么样？抛出各种异常。

双亲委托的好处就是防止自定义的类，例如我定义一个 String 类，但是类加载器会从顶端先加载

## 自定义加载器

1. 获取当前线程下的类加载器

2. 通过 getclassloader.getresouces 获取 URL，通过 URL 获取文件位置

3. 通过 Class.forname()获得指定的类

# 异常

## 异常种类


**Error**（错误）**:**是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

。这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 Error 的子类描述。

**Exception**（异常）**:**是程序本身可以处理的异常。

Exception 类有一个重要的子类 RuntimeException。RuntimeException 类及其子类表示“JVM 常用操作”引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

通常，Java 的异常(包括 Exception 和 Error)分为可查的异常（**checked exceptions**）和不可查的异常（**unchecked exceptions**）。

可查异常（编译器要求必须处置的异常）：正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。

除了 RuntimeException 及其子类以外，其他的 Exception 类及其子类都属于可查异常。这种异常的特点是 Java 编译器会检查它，也就是说，当程序中可能出现这类异常，要么用 try-catch 语句捕获它，要么用 throws 子句声明抛出它，否则编译不会通过。

不可查异常(编译器不要求强制处置的异常):包括运行时异常（RuntimeException 与其子类）和错误（Error）。

Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

运行时异常：都是 RuntimeException 类及其子类异常，如 NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是 Java 编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用 try-catch 语句捕获它，也没有用 throws 子句声明抛出它，也会编译通过。

非运行时异常 （编译异常）：是 RuntimeException 以外的异常，类型上都属于 Exception 类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如 IOException、SQLException 等以及用户自定义的 Exception 异常，一般情况下不自定义检查异常。

## 处理异常机制

在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。

抛出异常：当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息。运行时系统负责寻找处置异常的代码并执行。

捕获异常：在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler）。潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合。当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适 的异常处理器。运行时系统从发生异常的方

法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适 的异常处理器，则运行时系统终止。同时，意味着 Java 程序的终止。

对于运行时异常、错误或可查异常，Java 技术所要求的异常处理方式有所不同。

由于运行时异常的不可查性，为了更合理、更容易地实现应用程序，Java 规定，运行时异常将由 Java 运行时系统自动抛出，允许应用程序忽略运行时异常。

对于方法运行中可能出现的 Error，当运行方法不欲捕捉时，Java 允许该方法不做任何抛出声明。因为，大多数 Error 异常属于永远不能被允许发生的状况，也属于合理的应用程序不该捕捉的异常。

对于所有的可查异常，Java 规定：一个方法必须捕捉，或者声明抛出方法之外。也就是说，当一个方法选择不捕捉可查异常时，它必须声明将抛出异常。

能够捕捉异常的方法，需要提供相符类型的异常处理器。所捕捉的异常，可能是由于自身语句所引发并抛出的异常，也可能是由某个调用的方法或者 Java 运行时 系统等抛出的异常。也就是说，一个方法所能捕捉的异常，一定是 Java 代码在某处所抛出的异常。简单地说，异常总是先被抛出，后被捕捉的。

任何 Java 代码都可以抛出异常，如：自己编写的代码、来自 Java 开发环境包中代码，或者 Java 运行时系统。无论是谁，都可以通过 Java 的 throw 语句抛出异常。

从方法中抛出的任何异常都必须使用 throws 子句。

捕捉异常通过 try-catch 语句或者 try-catch-finally 语句实现。

总体来说，Java 规定：对于可查异常必须捕捉、或者声明抛出。允许忽略不可查的 RuntimeException 和 Error。

# 数据库

## 四大特性和隔离级别

本篇讲诉数据库中事务的四大特性（ACID），并且将会详细地说明事务的隔离级别。

如果一个数据库声称支持事务的操作，那么该数据库必须要具备以下四个特性：

**(1) 原子性（Atomicity）**

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

**(2) 一致性（Consistency）**

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户 A 和用户 B 两者的钱加起来一共是 5000，那么不管 A 和 B 之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是 5000，这就是事务的一致性。

**⑶ 隔离性（Isolation ）**

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

**⑷ 持久性（Durability ）**

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用 JDBC 操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

以上介绍完事务的四大特性(简称 ACID)，现在重点来说明下事务的隔离性，当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题：

**1，脏读**

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户 A 向用户 B 转账 100 元，对应 SQL 命令如下

```
update account set money=money+100 where name='B';  (此时 A 通知 B)

update account set money=money - 100 where name='A';
```

当只执行第一条 SQL 时，A 通知 B 查看账户，B 发现确实钱已到账（此时即发生了脏读），而之后无论第二条 SQL 是否执行，只要该事务不提交，则所有操作都将回滚，那么当 B 以后再次查看账户时就会发现钱其实并没有转。

**2，不可重复读**

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

例如事务 T1 在读取某一数据，而事务 T2 立马修改了这个数据并且提交事务给数据库，事务 T1 再次读取该数据就得到了不同的结果，发送了不可重复读。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据 A 和 B 依次查询就可能不同，A 和 B 就可能打起来了……

**3，虚读(幻读)**

幻读是事务非独立执行时发生的一种现象。例如事务 T1 对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务 T2 又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务 T1 的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务 T2 中添加的，就好像产生幻觉一样，这就是发生了幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

现在来看看 MySQL 数据库为我们提供的四种隔离级别：

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

# 设计模式

## 代理模式

### 静态代理

由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了。

**步骤：**

代理接口

委托类 实现代理接口

静态代理类 持有一个委托类的对象引

创建代理工厂，返回代理类的实例

**静态代理类优缺点**

优点：业务类只需要关注业务逻辑本身，保证了业务类的重用性。这是代理的共有优点。

缺点：

1）代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每一种方法都进行代理，静态代理在程序规模稍大时就无法胜任了。

2）如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

### 动态代理

a. 实现 InvocationHandler 接口创建自己的调用处理器

b. 给 Proxy 类提供 ClassLoader 和代理接口类型数组创建动态代理类

c. 以调用处理器类型为参数，利用反射机制得到动态代理类的构造函数

d. 以调用处理器对象为参数，利用动态代理类的构造函数创建动态代理类对象

Java代码

```
/**
 * 动态代理类对应的调用处理程序类
 */
public class SubjectInvocationHandler implements InvocationHandler {

 //代理类持有一个委托类的对象引用
 private Object delegate;

 public SubjectInvocationHandler(Object delegate) {
  this.delegate = delegate;
 }

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
  long stime = System.currentTimeMillis();
  //利用反射机制将请求分派给委托类处理。Method的invoke返回Object对象作为方法执行结果。
  //因为示例程序没有返回值，所以这里忽略了返回值处理
  method.invoke(delegate, args);
  long ftime = System.currentTimeMillis();
  System.out.println("执行任务耗时"+(ftime - stime)+"毫秒");

  return null;
 }
}
```

```
/**
 * 生成动态代理对象的工厂.
 */
public class DynProxyFactory {
 //客户类调用此工厂方法获得代理对象。
 //对客户类来说，其并不知道返回的是代理类对象还是委托类对象。
 public static Subject getInstance(){
  Subject delegate = new RealSubject();
  InvocationHandler handler = new SubjectInvocationHandler(delegate);
  Subject proxy = null;
  proxy = (Subject)Proxy.newProxyInstance(
    delegate.getClass().getClassLoader(),
    delegate.getClass().getInterfaces(),
    handler);
  return proxy;
 }
}
```

动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。在本示例中看不出来，因为 invoke 方法体内嵌入了具体的外围业务（记录任务处理前后时间并计算时间差），实际中可以类似 Spring AOP 那样配置外围业务。

但这样做仍然存在一个问题：JDK 给我们提供的动态代理只能代理接口，而不能代理没有接口的类。有什么方法可以解决呢？

### CGLib 动态代理

```
public class CGLibDynamicProxy implements MethodInterceptor {

    private static CGLibDynamicProxy instance = new CGLibDynamicProxy();

    private CGLibDynamicProxy() {
    }

    public static CGLibDynamicProxy getInstance() {
        return instance;
    }

    @SuppressWarnings("unchecked")
    public <T> T getProxy(Class<T> cls) {
        return (T) Enhancer.create(cls, this);
    }

    @Override
    public Object intercept(Object target, Method method, Object[] args, MethodProxy proxy) throws Throwable
        before();
        Object result = proxy.invokeSuper(target, args);
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

## 外观模式

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

# 线程池

## 为什么用线程池

1. 创建/销毁线程伴随着系统开销，过于频繁的创建/销毁线程，会很大程度上影响处理效率

例如：

记创建线程消耗时间 T1，执行任务消耗时间 T2，销毁线程消耗时间 T3

如果 T1+T3>T2，那么是不是说开启一个线程来执行这个任务太不划算了！

正好，线程池缓存线程，可用已有的闲置线程来执行新任务，避免了 T1+T3 带来的系统开销

2. 线程并发数量过多，抢占系统资源从而导致阻塞

   我们知道线程能共享系统资源，如果同时执行的线程过多，就有可能导致系统资源不足而产生阻塞的情况

   运用线程池能有效的控制线程最大并发数，避免以上的问题

3. 对线程进行一些简单的管理

   比如：延时执行、定时循环执行的策略等

   运用线程池都能进行很好的实现

## 线程池 ThreadPoolExecutor

既然 Android 中线程池来自于 Java，那么研究 Android 线程池其实也可以说是研究 Java 中的线程池

在 Java 中，线程池的概念是 Executor 这个接口，具体实现为 ThreadPoolExecutor 类，学习 Java 中的线程池，就可以直接学习他了

对线程池的配置，就是对 ThreadPoolExecutor 构造函数的参数的配置，既然这些参数这么重要，就来看看构造函数的各个参数吧

有一个 gettask 循环从队列中读取任务

# 内部类

## 为什么要使用内部类

为什么要使用内部类？在《Think in java》中有这样一句话：使用内部类最吸引人的原因是：**每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。**

在我们程序设计中有时候会存在一些使用接口很难解决的问题，这个时候我们可以利用内部类提供的、可以继承多个具体的或者抽象的类的能力来解决这些程序设计问题。可以这样说，**接口只是解决了部分问题，而内部类使得多重继承的解决方案变得更加完整。**

其实使用内部类最大的优点就在于它能够非常好的解决多重继承的问题，但是如果我们不需要解决多重继承问题，那么我们自然可以使用其他的编码方式，但是使用内部类还能够为我们带来如下特性（摘自《Think in java》）：

1、内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。

2、在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

3、创建内部类对象的时刻并不依赖于外围类对象的创建。

4、内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

5、内部类提供了更好的封装，除了该外围类，其他类都不能访问。

## 访问内部类

```
OuterClass outerClass = new OuterClass();
        OuterClass.InnerClass innerClass = outerClass.new InnerClass();
        innerClass.display();
```

在这个应用程序中，我们可以看到内部了 InnerClass 可以对外围类 OuterClass 的属性进行无缝的访问，尽管它是 private 修饰的。这是因为当我们在创建某个外围类的内部类对象时，此时内部类对象必定会捕获一个指向那个外围类对象的引用，只要我们在访问外围类的成员时，就会用这个引用来选择外围类的成员。

其实在这个应用程序中我们还看到了如何来引用内部类：引用内部类我们需要指明这个对象的类型：OuterClasName.InnerClassName。同时如果我们需要创建某个内部类对象，必须要利用外部类的对象通过.new 来创建内部类： OuterClass.InnerClass innerClass = outerClass.new InnerClass();。

同时如果我们需要生成对外部类对象的引用，可以使用 OuterClassName.this，这样就能够产生一个正确引用外部类的引用了。当然这点实在编译期就知晓了，没有任何运行时的成本。

## 成员内部类

成员内部类也是最普通的内部类，它是外围类的一个成员，所以他是可以无限制的访问外围类的所有 成员属性和方法，尽管是 private 的，但是外围类要访问内部类的成员属性和方法则需要通过内部类实例来访问。

在成员内部类中要注意两点，

**第一：成员内部类中不能存在任何 static 的变量和方法；**

首先要明白一下三点：

static 类型的属性和方法，在类加载的时候就会存在于内存中。

要想是用某个类的 static 属性和方法，那么这个类必须要加载到虚拟机- 中。

非静态内部类并不随外部类一起加载，只有在实例化外部类之后才会加载。

现在考虑这个情况：在外部类并没有实例化，内部类还没有加载，这时候如果调用内部类的静态成员或方法，内部类还没有加载，却试图在内存中创建该内部类的静态成员，这明显是矛盾的。所以非静态内部类不能有静态成员变量或静态方法。

**第二：成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类。**

## 静态内部类

在 java 提高篇-----关键字 static 中提到 Static 可以修饰成员变量、方法、代码块，其他它还可以修饰内部类，使用 static 修饰的内部类我们称之为静态内部类，不过我们更喜欢称之为嵌套内部类。静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。没有这个引用就意味着：

1、 它的创建是不需要依赖于外围类的。

2、 它不能使用任何外围类的非 static 成员变量和方法。