**目录**

[1、const 和 #define区别： 2](#_Toc3821222)

[2、STL的组件，以及vector、list、deque、set、map的底层实现 3](#_Toc3821223)

[3、内存泄露种类及解决方法 3](#_Toc3821224)

[4、TCP和UDP区别 3](#_Toc3821225)

[TCP面向字节流和UDP面向数据报的体现 4](#_Toc3821226)

[TCP和UDP中报文的边界问题 4](#_Toc3821227)

[5、TCP三次握手，四次挥手；为什么不是二次握手、三次挥手 4](#_Toc3821228)

[6、粘包现象及处理 5](#_Toc3821229)

[7、Linux常用命令，如何查看磁盘容量，cpu用量，各端口号 5](#_Toc3821230)

[小题目：快速判断一个数是不是2的n次 5](#_Toc3821231)

[各类排序的时间复杂度、空间复杂度、稳定性对比 5](#_Toc3821232)

[红黑树（Red Black Tree） 6](#_Toc3821233)

[AVL树 6](#_Toc3821234)

[B树 6](#_Toc3821235)

[8、进程与线程间通信方式 7](#_Toc3821236)

[**Linux操作系统怎么管理内存** 7](#_Toc3821237)

[**Linux上malloc怎么实现** 7](#_Toc3821238)

[**内存区域的划分和分配** 8](#_Toc3821239)

[9、进程与线程的区别 8](#_Toc3821240)

[10、HTTP 简介及http1与http2的主要区别 9](#_Toc3821241)

https://www.nowcoder.com/discuss/107506

1. **自我介绍**
2. 简单介绍自己姓名、学校、学院、专业、研究方向，本科成绩
3. 经常使用编程的思维思考问题，喜欢用编程的方式解决问题，渴望知识
4. 实习经历和竞赛经历，渴望得到团队的力量
5. 擅长领域（C\*、C++、Linux）
6. 做过班长兼职工作，协调各方面和生活的适应能力强
7. 到大的公司是一个好的平台，可以让自己变得更加优秀，因此我想加入本公司

我叫袁森，来自东南大学仪器科学与工程学院，研究生主要研究内容是袖带式血压测量方法研究。c/c++，linux是我比较熟悉的领域，因为c语言用的相对较多，所以更加熟悉一些。因为是非科班出身，所以现在对很多方面了解不是很深入，但是我只要对某个方面认真研究下去，一定会学的很好。我大学成绩一直很优秀，考研用了一百天的样子就考了400分。而且我学习一个方面的知识非常扎实，因此我可以一边学习一边到辅导班去给别人上课。我对工科各方面都比较感兴趣，本科时就发现软件方面更适合自己，因为硬件主要靠的是经验，而软件主要是思维和经验，我觉得自己最大的优点是懂得思考，不轻易选择，因为正确的想法指导实践会事半功倍，但是我也会积极的实践，懂得总结再思考。所以感觉自己更加适合创新性更强的软件工作。

1. **问面试官的问题**
2. 请问您是哪个部门，贵部门对应届毕业生有什么样的要求？
3. 那您能简单介绍一下贵部门有哪些技术吸引力的地方吗？
4. 如果我能加入贵部门，我会做哪方面的工作，得到哪些锻炼

# 1、const 和 #define区别：

(1) **编译器处理方式**不同：

　　define宏是在预处理阶段展开。

　　const常量是编译运行阶段使用。

(2) **类型和安全检查**不同：

　　define宏没有类型，不做任何类型检查，仅仅是展开。

　　const常量有具体的类型，在编译阶段会执行类型检查。

(3) **存储方式**不同：

　　define宏仅仅是展开，有多少地方使用，就展开多少次，不会分配内存。

　　const常量会在内存中分配(可以是**堆**中也可以是**栈**中)。

(4)**const可以节省空间**，避免不必要的内存分配。

例： #define PI 3.14159 //常量宏

        const doube Pi=3.14159; //此时并未将Pi放入ROM中 ......

        double i=Pi; //此时为Pi分配内存，以后不再分配！

        double I=PI; //编译期间进行宏替换，分配内存

        double j=Pi; //没有内存分配

        double J=PI; //再进行宏替换，又一次分配内存！

        const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，**const**定义的常量在程序运行过程中只有**一份拷贝**，而 **#define**定义的常量在内存中有**若干个拷贝**。

(5) **提高了效率**：

编译器通常不为普通const常量分配存储空间，而是将它们**保存在符号表中**，这使得它成为一个编译期间的常量，**没有了存储与读内存的操作**，使得它的效率也很高。

# 2、STL的组件，以及vector、list、deque、set、map的底层实现

（1）STL六大组件：**容器**(container)、**算法**(algorithm)、**迭代器**(iterator)、**仿函数**、**适配器**、**空间配置器**

STL优点：不需要安装直接使用；高复用性、高移植、高性能；数据和算法分离、迭代器连接

1、容器：各种数据结构，如：vector、list、deque、set、map，是一种类模板

2、算法：各种常用算法，如：sort、find、copy、for\_each，是一种函数模板

3、迭代器：一种将opertor\*、opertor->、opertor++等指针操作重载的类模板。所有容器都有自带的专属迭代器，将算法和容器进行结合

4、仿函数：是一种重载了opertor()的类模板

5、适配器：一种用来修饰容器或仿函数或迭代器接口的东西

6、空间适配器：动态分配空间，一般不需要自己管理

六大组件的关系：容器通过空间适配器取得数据储存空间，算法通过迭代器变化容器中的内容，仿函数协助算法完成不同策略变化，适配器修饰容器仿函数

顺序容器：vector、list、deque、string、forward\_list、array

关联容器：map、set、multimap、multiset、unorder\_map、unorder\_set、unorder\_multimap、unorder\_multiset

适配容器：stack、queue、priority\_queue (stack和queue基于deque的，priority\_queue基于vector)

质变算法 ：拷贝、替换、删除等

非质变算法：查找、计数、遍历、寻找极值等

迭代器：依序访问容器中的元素。输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器。

（2）vector是**动态数组**，如果你需要高效的随即存取，而不在乎插入和删除的效率时使用。

list是**双向链表**，如果你需要大量的插入和删除，而不关心随即存取时使用。

deque是**双端队列**，[堆1]...[堆2]...[堆3]，间断堆保存，堆内内存连续，堆间断续。如果你需要随即存取，而且关心两端数据的插入和删除时使用。

set和map底层采用的是[**红黑树**](#_红黑树（Red_Black_Tree）)**（自平衡二叉搜索树）**实现，查找是常数时间，遍历也不错， 只是每次插入值的时候，会重新构成底层的平衡二叉树，效率有一定影响。

# 3、内存泄露种类及解决方法

（1）种类：堆内存泄漏（使用malloc realloc new后未free或delete），系统资源泄露（系统资源使用函数使用后未调用释放函数）

（2）解决方法：**良好的编码习惯（new后必须delete）、重载new和delete、使用智能指针**。

（3）排查：1.[BoundsChecker](https://www.cnblogs.com/hrhguanli/p/3890171.html)，定位运行时发生的各种错误

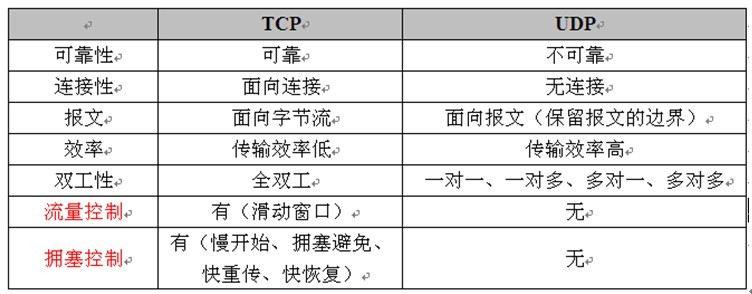
2.调试运行DEBUG版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境OUTPUT窗口)，综合分析内存泄漏的原因，排除内存泄漏。

3.linux工具之检测内存泄漏-[valgrind](https://www.cnblogs.com/myyan/p/4886931.html),功能强大，不仅仅是内存泄漏检测工具。

# 4、TCP和UDP区别

* TCP**面向连接**（三次握手），通信前需要先建立连接；UDP面向无连接，通信前不需要连接。
* TCP通过**序号、重传、流量控制、拥塞控制**实现可靠传输；UDP不保障可靠传输，尽最大努力交付。

若通信**数据完整性**需多于**通信实时性**，则应该选用 TCP 协议（如**文件传输、重要状态的更新**等）；反之，则使用 UDP 协议（如**视频传输、实时通信**等）



## TCP面向字节流和UDP面向数据报的体现

TCP**面向字节流**传输，因此可以被分割并在接收端重组；UDP**面向数据报**传输。

1、面向报文的传输方式是应用层给UDP多长的报文，UDP就照样发送，即一次发送一个报文。因此，应用程序必须选择合适大小的报文。若报文太长，则IP层需要分片，降低效率。若太短，会是IP太小。UDP对应用层交下来的报文，**既不合并，也不拆分，而是保留这些报文的边界。**这也就是说，应用层给UDP多长的报文，UDP就照样发送，即一次发送一个报文。

2、面向字节流的话，虽然应用程序和TCP的交互是一次一个数据块（大小不等），但TCP**把应用程序看成是一连串的无结构的字节流**。TCP有一个**缓冲**，当应用程序传送的数据块太长，TCP就可以把它划分短一些再传送。如果应用程序一次只发送一个字节，TCP也可以等待积累有足够多的字节后再构成报文段发送出去。（**将应用层数据分段或合并**）

## TCP和UDP中报文的边界问题

在前面讲解UDP时，提到了UDP保留了报文的边界，下面我们来谈谈**TCP和UDP中报文的边界问题**。在默认的阻塞模式下，**TCP无边界，UDP有边界**。

1、对于TCP协议，客户端连续发送数据，只要服务端的这个函数的**缓冲区**足够大，会一次性接收过来，即客户端是分好几次发过来，是有边界的，而服务端却一次性接收过来，所以证明是无边界的；

2、 对于UDP协议，客户端连续发送数据，即使服务端的这个函数的缓冲区足够大，也**只会一次一次的接收，发送多少次接收多少次**，即客户端分几次发送过来，服务端就必须按几次接收，从而证明，这种UDP的通讯模式是有边界的。

# 5、TCP三次握手，四次挥手；为什么不是二次握手、三次挥手

（1）TCP为什么不是两次握手而是三次

如果仅两次连接可能出现一种情况：客户端发送完连接报文（第一次握手）后由于网络不好，延时很久后报文到达服务端，服务端接收到报文后向客户端发起连接（第二次握手）。此时客户端会认定此报文为失效报文，但在两次握手情况下服务端会认为已经建立起了连接，**服务端会一直等待客户端发送数据**，但因为客户端会认为服务端第二次握手的回复是对失效请求的回复，不会去处理。这就造成了服务端一直等待客户端数据的情况，浪费资源。

（2）TCP为什么挥手是四次而不是三次

**TCP是全双工的**，它允许两个方向的数据传输被独立关闭。当主动发起关闭的一方关闭连接之后，TCP进入半关闭状态，此时主动方可以只关闭输出流。

之所以不是三次而是四次主要是因为被动关闭方将**"对主动关闭报文的确认"和"关闭连接"**两个操作分两次进行。对主动关闭报文的确认是为了快速告知主动关闭方，此关闭连接报文已经收到。此时被动方不立即关闭连接是为了**将缓冲中剩下的数据从输出流发回主动关闭方**（主动方接收到数据后同样要进行确认），因此要把"确认关闭"和"关闭连接"分两次进行。

# 6、[粘包现象及处理](https://www.cnblogs.com/qiaoconglovelife/p/5733247.html)

（1）**什么是粘包现象**：TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

（2）**为什么出现粘包现象**：

发送方原因：TCP默认使用了Nagle算法（只有上一个分组得到确认，才会发送下一个分组；收集多个小分组，在一个确认到来时一起发送）

接收方原因：TCP接收到分组时应用层并不一定会立即处理，如果TCP接收分组的速度大于应用程序读分组的速度，多个包就会被存至缓存，应用程序读时，就会读到多个首尾相接粘到一起的包。

（3）**什么时候需要处理粘包现象**

1.如果发送方发送的多个分组本来就是同一个数据的不同部分，比如一个很大的文件被分成多个分组发送，这时当然不需要处理粘包的现象；

2.但如果多个分组本毫不相干，甚至是并列的关系，我们就一定要处理粘包问题了。

(4) **如何处理粘包现象**

1.发送方：对于发送方造成的粘包现象，我们可以通过关闭Nagle算法来解决，使用TCP\_NODELAY选项来关闭Nagle算法

2. 应用层处理：两种途径：

1）**格式化数据**：每条数据有固定的格式（开始符、结束符），但选择开始符和结束符的时候一定要注意每条数据的内部一定不能出现开始符或结束符；

2）**发送长度**：发送每条数据的时候，将数据的长度一并发送，应用层处理时可以根据长度来判断每条数据的开始和结束。

# 7、Linux常用命令，如何查看磁盘容量，cpu用量，各端口号

（1）查看磁盘容量：**df**可显示磁盘的文件系统与使用情形，**du**会显示指定的目录或文件所占用的磁盘空间。

（2）cpu用量：**top**动态显示进程状况（M按照MEM排序，P按照CPU排序，u查看是哪个用户的进程，q退出）。

（3）各端口号：**netstat** –ntlp //查看当前所有tcp端口

## 小题目：快速判断一个数是不是2的n次

**n&(n-1)** ，也就是一个数n，如果是2的次幂，那么它和比它小于1的数进行与操作，如果结果是0，那么这个n就是2的次幂。（**n&(n-1)的结果是将n二进制最右侧的1去掉**）

因为一个数n，如果是2的次幂，那么二进制可以表示为1000000.....，那么比它小于1的数就可表示为0111111....，连个二进制串 & 操作后，结果自然是0.

## 各类排序的时间复杂度、空间复杂度、稳定性对比



## 红黑树（Red Black Tree）

红黑树是一种**自平衡二叉查找树**，STL很多部分(包括set, map)应用了红黑树的变体。

性质1. 节点是红色或黑色。

性质2. 根节点是黑色。

性质3 每个叶节点（NIL节点，空节点）是黑色的。

性质4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

## AVL树

在计算机科学中，AVL树是最先发明的**自平衡二叉查找树**。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是**O(log n)**。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

AVL树本质上还是一棵二叉搜索树，它的特点是:

1.本身首先是一棵二叉搜索树。

2.带有平衡条件:每个结点的左右子树的高度之差的绝对值(平衡因子)最多为1。也就是说，AVL树，本质上是带了平衡功能的二叉查找树(二叉排序树，二叉搜索树)。

## B树

1、其**定义**如下：

一棵m阶的B树满足下列条件：

⑴ 树中每个结点至多有m个孩子；

⑵ 除根结点和叶子结点外，其它每个结点至少有m/2个孩子；

⑶ 若根结点不是叶子结点，则至少有2个孩子；

⑷ 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息；

⑸ 有k个孩子的非终端结点恰好包含有k-1个关键字。

在B树中，每个结点中关键字从小到大排列，并且当该结点的孩子是非叶子结点时，该k-1个关键字正好是k个孩子包含的关键字的值域的分划。

因为[叶子](https://baike.so.com/doc/5405621-7595388.html)结点不包含关键字，所以可以把叶子结点看成在树里实际上并不存在外部结点，指向这些外部结点的指针为空，叶子结点的数目正好等于树中所包含的关键字总个数加1。

B树中的一个包含n个关键字，n+1个指针的结点的一般[形式](https://baike.so.com/doc/5330123-5565297.html)为（n,P0,K1,P1,…,Kn,Pn）

其中Ki为关键字，K1<K2<…<Kn, Pi 是指向包括Ki到Ki+1之间的关键字子树的[指针](https://baike.so.com/doc/1043844-1104112.html)。

**2、**[**B树是一种查找树**](https://blog.csdn.net/guoziqing506/article/details/64122287)，我们知道，这一类树（比如二叉查找树，红黑树等等）最初生成的目的都是为了**解决某种系统中，查找效率低的问题**。B树也是如此，它最初启发于二叉查找树，二叉查找树的特点是每个非叶节点都只有两个孩子节点。然而这种做法会导致当数据量非常大时，二叉查找树的深度过深，搜索算法自根节点向下搜索时，需要访问的节点也就变的相当多。如果这些节点存储在外存储器中，每访问一个节点，相当于就是进行了一次I/O操作，随着树高度的增加，频繁的I/O操作一定会降低查询的效率。

从外存储器中读取信息的步骤，简单来分，大致有两步：

1.找到存储这个数据所对应的磁盘页面，这个过程是机械化的过程，需要依靠磁臂的转动，找到对应磁道，所以耗时长。

2.读取数据进内存，并实施运算，这是电子化的过程，相当快。

综上，对于外存储器的信息读取最大的时间消耗在于寻找磁盘页面。那么一个基本的想法就是能不能减少这种读取的次数，在一个磁盘页面上，**多存储一些索引信息**。B树的基本逻辑就是这个思路，它要**改二叉为多叉，每个节点存储更多的指针信息**，以降低I/O操作数。

# 8、进程与线程间通信方式

**一、进程间的通信方式**

1.**管道**( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

**2.有名管道**(namedpipe)：是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

**3.信号量**(semophore )：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，**主要作为进程间以及同一进程内不同线程之间的同步手段**。

**4.消息队列**(messagequeue)：是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递**信息少**、管道只能承载**无格式**字节流以及**缓冲区大小受限**等缺点。

**5.信号**(sinal )：是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

**6.共享内存**(shared memory )：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。**共享内存是最快的 IPC 方式**，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

**7.套接字**(socket )：是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

**二、线程间的通信方式**

**1.锁机制**：包括互斥锁、读写锁、条件变量

   \*互斥锁提供了以排他方式防止数据结构被并发修改的方法。

   \*读写锁允许多个线程同时读共享数据，而对写操作是互斥的。

   \*条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。**条件变量始终与互斥锁一起使用**。

**2.信号量机制**(Semaphore)：包括无名线程信号量和命名线程信号量

**3.信号机制**(Signal)：类似进程间的信号处理

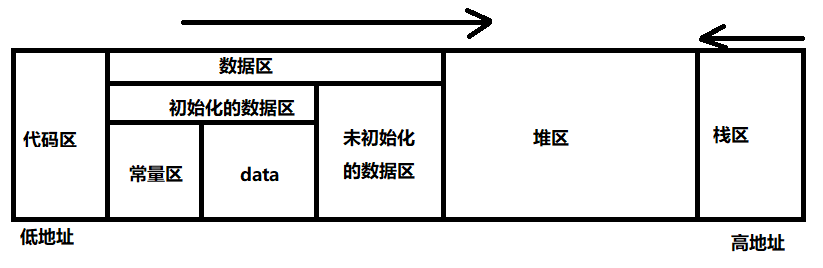
线程间的通信目的**主要是线程同步**，所以线程没有像进程中的用于数据交换的通信机制。

[**Linux操作系统怎么管理内存**](https://www.cnblogs.com/ralap7/p/9184773.html#top)

[**Linux上malloc怎么实现**](https://blog.csdn.net/mmshixing/article/details/51679571)

malloc 函数的实质是它有一个将可用的内存块连接为一个长长的列表的所谓**空闲链表**。 调用 malloc（）函数时，**它沿着连接表寻找一个大到足以满足用户请求所需要的内存块**。 然后，**将该内存块一分为二**（一块的大小与用户申请的大小相等，另一块的大小就是剩下来的字节）。 接下来，将分配给用户的那块内存存储区域传给用户，并将剩下的那块（如果有的话）返回到连接表上。 调用 free 函数时，它将用户释放的内存块连接到空闲链表上。 到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段， 那么空闲链表上可能没有可以满足用户要求的片段了。于是，malloc（）函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。

**[内存区域的划分和分配](https://www.cnblogs.com/ruixin-jia/p/5877492.html)**

****

**内存分配方式**

内存分配方式有三种：

[1]从**静态存储区域**分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

[2]在**栈**上创建。在执行函数时，函数内**局部变量**的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

[3]从**堆**上分配，亦称**动态内存分配**。程序在**运行的时候用malloc或new申请任意多少的内存**，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

# 9、进程与线程的区别

1、同一进程中的线程**共享内存空间、数据空间**，但是进程之间是独立的；

2、对主线程的修改可能会**影响其他线程**的行为，但是父进程的修改不会影响其他子进程（除删除外）；

3、线程是一个**上下文的执行指令**，而进程则是**与运算相关的一簇资源**。

4、同一进程之间的**线程可以直接通信**，但是进程之间的交流需要借助中间代理实现。

5、**创建新线程很容易**，但是创建新进程需要**对父进程做一次拷贝**。

6、一个线程可以**操作同一进程的其他线程**，但是进程**只能操作其子进程**。

7、**线程启动速度快**，进程启动速度慢（但是两者的运行速度没有可比性）。

[几个系统调用分析 glibc中的malloc调用和共享内存原理](https://blog.csdn.net/cncnlg/article/details/44759217)

# 10、[HTTP 简介](http://www.runoob.com/http/http-intro.html)及http1与http2的主要区别

HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写,是用于从万维网（WWW:World Wide Web ）服务器传输超文本到本地浏览器的传送协议。HTTP是一个基于TCP/IP通信协议来传递数据（HTML 文件, 图片文件, 查询结果等）。

**HTTP 工作原理**

HTTP协议工作于客户端-服务端架构上。浏览器作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。

Web服务器有：Apache服务器，IIS服务器（Internet Information Services）等。

Web服务器根据接收到的请求后，向客户端发送响应信息。

HTTP默认端口号为**80**，但是你也可以改为8080或者其他端口。

**HTTP三点注意事项：**

HTTP是**无连接**：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

HTTP是**媒体独立**的：这意味着，只要客户端和服务器知道如何处理的数据内容，任何类型的数据都可以通过HTTP发送。客户端以及服务器指定使用适合的MIME-type内容类型。

HTTP是**无状态**：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

**http1与http2的主要区别**

**二进制传输**

http2采用二进制传输，相较于文本传输的http1来说更加安全可靠。

**多路复用**

http1一个连接只能提交一个请求，而http2可以同时处理无数个请求，可以降低连接的占用数量，进一步提升网络的吞吐量。

**头部压缩**

http2通过gzip与compress对头部进行压缩，并且在客户端与服务端各维护了一份头部索引表，只需要根据索引id就可以进行头部信息的传输，缩小了头部容量，间接提升了传输效率。

**服务端推送**

服务端可以主动推送资源给客户端，避免客户端花过多的时间逐个请求资源，这样可以降低整个请求的响应时间。

# [11、死锁的四个必要条件和解决办法](https://blog.csdn.net/guaiguaihenguai/article/details/80303835)

**死锁概念及产生原理**

    概念：多个并发进程因争夺系统资源而产生相互等待的现象。

    原理：当一组进程中的每个进程都在等待某个事件发生，而只有这组进程中的其他进程才能触发该事件，这就称这组进程发生了死锁。

    本质原因：1）、系统资源有限。2）、进程推进顺序不合理。

**死锁产生的4个必要条件**

    1、**互斥**：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。

    2、**占有且等待**：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。

    3、**不可抢占**：有人已经占有了某项资源，不能因为自己需要，就去把别人的资源抢过来。

    4、**循环等待**：存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源

**避免死锁的方法**

1、**死锁预防** ----- 确保系统永远不会进入死锁状态

产生死锁需要四个条件，那么，只要这四个条件中至少有一个条件得不到满足，就不可能发生死锁了。由于互斥条件是非共享资源所必须的，不仅不能改变，还应加以保证，所以，主要是破坏产生死锁的其他三个条件。

a、破坏“占有且等待”条件

方法1：所有进程在运行之前，必须一次性地申请其在整个运行过程中所需要的全部资源。

优点：简单易实施且安全。

缺点：因为某项资源不满足，进程无法启动，而其他已经满足了的资源也不会得到利用，严重降低了资源的利用率，造成资源浪费。使进程经常发生饥饿现象。

方法2：该方法是对第一种方法的改进，允许进程**只获得运行初期需要的资源，便开始运行**，在运行过程中逐步释放掉分配到的已经使用完毕的资源，然后再去请求新的资源。

b、破坏“不可抢占”条件

当一个已经持有了一些资源的进程在提出新的资源请求没有得到满足时，它必须释放已经保持的所有资源，待再使用时再重新申请。这就意味着进程已占有的资源会被短暂地释放或者说是被抢占了。

该种方法实现起来比较复杂，且代价也比较大。释放已经保持的资源很有可能会导致进程之前的工作实效等，反复的申请和释放资源会导致进程的执行被无限的推迟，这不仅会延长进程的周转周期，还会影响系统的吞吐量。

c、破坏“循环等待”条件

可以通过定义资源类型的线性顺序来预防，可将每个资源编号，当一个进程占有编号为i的资源时，那么它下一次申请资源只能申请编号大于i的资源。这样虽然避免了循环等待，但是这种方法是比较低效的，资源的执行速度回变慢，并且可能在没有必要的情况下拒绝资源的访问。

2、**避免死锁** ----- 在使用前进行判断，只允许不会产生死锁的进程申请资源

的死锁避免是利用额外的检验信息，在分配资源时判断是否会出现死锁，只在不会出现死锁的情况下才分配资源。

两种避免办法：

    1、如果一个进程的请求会导致死锁，则不启动该进程

    2、如果一个进程的增加资源请求会导致死锁 ，则拒绝该申请。

避免死锁的具体实现通常利用[**银行家算法**](https://baike.baidu.com/item/%E9%93%B6%E8%A1%8C%E5%AE%B6%E7%AE%97%E6%B3%95/1679781?fr=aladdin)**：**

a、银行家算法的相关数据结构

**可利用资源向量Available**：用于表示系统里边各种资源剩余的数目。由于系统里边拥有的资源通常都是有很多种（假设有m种），所以，我们用一个有m个元素的数组来表示各种资源。数组元素的初始值为系统里边所配置的该类全部可用资源的数目，其数值随着该类资源的分配与回收动态地改变。

**最大需求矩阵Max**：用于表示各个进程对各种资源的额最大需求量。进程可能会有很多个（假设为n个），那么，我们就可以用一个nxm的矩阵来表示各个进程多各种资源的最大需求量

**分配矩阵Allocation**：顾名思义，就是用于表示已经分配给各个进程的各种资源的数目。也是一个nxm的矩阵。

**需求矩阵Need**：用于表示进程仍然需要的资源数目，用一个nxm的矩阵表示。系统可能没法一下就满足了某个进程的最大需求（通常进程对资源的最大需求也是只它在整个运行周期中需要的资源数目，并不是每一个时刻都需要这么多），于是，为了进程的执行能够向前推进，通常，系统会先分配个进程一部分资源保证进程能够执行起来。那么，进程的最大需求减去已经分配给进程的数目，就得到了进程仍然需要的资源数目了。

**银行家算法通过对进程需求、占有和系统拥有资源的实时统计，确保系统在分配给进程资源不会造成死锁才会给与分配。**

死锁避免的优点：不需要死锁预防中的抢占和重新运行进程，并且比死锁预防的限制要少。

死锁避免的限制：

    必须事先声明每个进程请求的最大资源量

    考虑的进程必须无关的，也就是说，它们执行的顺序必须没有任何同步要求的限制

    分配的资源数目必须是固定的。

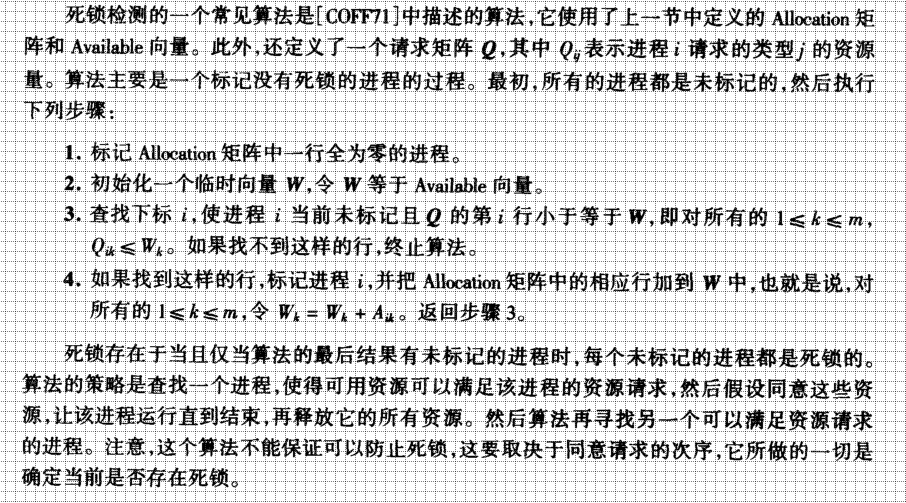
    在占有资源时，进程不能退出

3、**死锁检测与解除** ----- 在检测到运行系统进入死锁，进行恢复。

    允许系统进入到死锁状态

    死锁检测

下图截自《操作系统--精髓与设计原理》



**死锁的解除**

如果利用死锁检测算法检测出系统已经出现了死锁 ，那么，此时就需要对系统采取相应的措施。常用的解除死锁的方法：

1、**抢占资源**：从一个或多个进程中抢占足够数量的资源分配给死锁进程，以解除死锁状态。

2、**终止（或撤销）进程**：终止或撤销系统中的一个或多个死锁进程，直至打破死锁状态。

    a、终止所有的死锁进程。这种方式简单粗暴，但是代价很大，很有可能会导致一些已经运行了很久的进程前功尽弃。

    b、逐个终止进程，直至死锁状态解除。该方法的代价也很大，因为每终止一个进程就需要使用死锁检测来检测系统当前是否处于死锁状态。另外，每次终止进程的时候终止那个进程呢？每次都应该采用最优策略来选择一个“代价最小”的进程来解除死锁状态。一般根据如下几个方面来决定终止哪个进程：

    进程的优先级

    进程已运行时间以及运行完成还需要的时间

    进程已占用系统资源

    进程运行完成还需要的资源

    终止进程数目

    进程是交互还是批处理

# 12、构造函数中可不可以抛出异常?析构函数呢?

[理论](http://www.so.com/s?q=%E7%90%86%E8%AE%BA&ie=utf-8&src=internal_wenda_recommend_textn)上都可以抛出异常。但[**析构函数**](http://www.so.com/s?q=%E6%9E%90%E6%9E%84%E5%87%BD%E6%95%B0&ie=utf-8&src=internal_wenda_recommend_textn)**最好不要抛出异常**。

抛出异常时，将暂停当前[函数](http://www.so.com/s?q=%E5%87%BD%E6%95%B0&ie=utf-8&src=internal_wenda_recommend_textn)的执行，开始查找匹配的catch[子句](http://www.so.com/s?q=%E5%AD%90%E5%8F%A5&ie=utf-8&src=internal_wenda_recommend_textn)。首先检查throw本身是否在try块内部，如果是，检查与该try相关的catch子句，看是否可以处理该异常。如果不能处理，就退出当前函数，并且**释放当前函数的**[**内存**](http://www.so.com/s?q=%E5%86%85%E5%AD%98&ie=utf-8&src=internal_wenda_recommend_textn)**并销毁**[**局部**](http://www.so.com/s?q=%E5%B1%80%E9%83%A8&ie=utf-8&src=internal_wenda_recommend_textn)[**对象**](http://www.so.com/s?q=%E5%AF%B9%E8%B1%A1&ie=utf-8&src=internal_wenda_recommend_textn)，继续到上层的调用函数中查找，直到找到一个可以处理该异常的catch。这个[过程](http://www.so.com/s?q=%E8%BF%87%E7%A8%8B&ie=utf-8&src=internal_wenda_recommend_textn)称为**栈展开**。

在为某个异常进行栈展开的时候，析构函数如果又抛出自己的未经处理的另一个异常，将会导致调用标准库terminate函数。通常terminate函数将调用[abort函数](http://www.so.com/s?q=abort%E5%87%BD%E6%95%B0&ie=utf-8&src=internal_wenda_recommend_textn)，导致[程序](http://www.so.com/s?q=%E7%A8%8B%E5%BA%8F&ie=utf-8&src=internal_wenda_recommend_textn)的非正常退出。**所以析构函数应该从不抛出异常**。

**问题：**

**1**、红黑树、AVL树、B/B+树等具体算法实现和原理

2、redis

3、hash表、最短路径

4、行业内新东西，c++业务方向，技术应用

5、分布式技术、后台开发的应用技术

# 13、哈希表

**[哈希表及处理冲突的方法](http://www.cnblogs.com/guweiwei/p/6945700.html" \l "top)**

哈希法又称**散列法、杂凑法以及关键字地址计算法**等，相应的表称为**哈希表。**这种方法的基本思想是：首先在元素的关键字k和元素的存储位置p之间建立一个对应关系f，使得p=f(k)，f称为**哈希函数**。创建哈希表时，把关键字为k的元素直接存入地址为f(k)的单元；以后当查找关键字为k的元素时，再利用哈希函数计算出该元素的存储位置p=f(k)，从而达到按关键字直接存取元素的目的。

   当关键字集合很大时，关键字值不同的元素可能会映象到哈希表的同一地址上，即 k1≠k2 ，但 H（k1）=H（k2），这种现象称为**冲突，**此时称k1和k2为**同义词。**实际中，冲突是不可避免的，只能通过改进哈希函数的性能来减少冲突。

综上所述，哈希法主要包括以下两方面的内容：

 1）如何构造哈希函数

 2）如何处理冲突。

### 哈希函数的构造方法

    构造哈希函数的原则是：①函数本身便于计算；②计算出来的地址分布均匀，即对任一关键字k，f(k) 对应不同地址的概率相等，目的是尽可能减少冲突。

下面介绍构造哈希函数常用的六种方法。

**1. 直接地址法**

**2． 数字分析法**

**3． 平方取中法**

**4． 分段叠加法**

**5． 除留余数法**

**6． 伪随机数法**

### [处理冲突的方法及其优缺点](http://www.cnblogs.com/cing/p/8821389.html)

1. **开放定址法**
2. **再哈希法**
3. **链地址法**
4. **建立公共溢出区**

### [哈希表查找——成功和不成功时的平均查找长度](https://blog.csdn.net/longlovefilm/article/details/78009782)

# 14、[构造函数不能为虚函数](https://blog.csdn.net/smilesundream/article/details/75675602)

1. **虚函数构造时本身的矛盾**：虚函数需要通过虚函数指针指向一个虚函数表，当将构造函数声明为虚函数，在创建对象时，首先要调用构造函数，然后构造函数是虚函数，就需要用虚函数指针去调用，但是，对象都还没构造，也就没有虚函数指针，造成了一个矛盾的问题。

2. **虚构造函数继承时的矛盾**：当创建一个子类对象时，如果将构造函数声明为虚函数，因为子类在构造时，会先调用父类的够造函数，但是子类已经覆盖了基类的构造函数(因为是虚函数)，所以也就无法进一步执行，导致程序出错。

3.**虚函数和构造函数实用阶段矛盾**：构造函数是在编译阶段自动调用的，虚函数是在运行阶段多态时使用，因此没有必要将构造函数设置为虚函数。