1. **系统：进程与线程有什么区别**

* 进程是资源分配的最小单位，线程是程序执行的最小单位
* 线程是进程的一个执行流，一个进程可以由很多个线程组成
* 进程有独立地址空间，而线程是共享进程中的数据，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多
* 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行
* 多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间

1. **系统：线程池是什么**

* 线程池就是事先将线程放到一个容器中，当使用线程的时候，不用再去new出一个线程，直接从线程池取出来就可以了
* 优点1：降低资源消耗:重复利用线程池中的线程节省线程创建和销毁带来的消耗；
* 优点2：提高性能：当任务需求时，可以不用创建线程直接执行，主要是直接从线程池中取出线程去执行；
* 优点3：提高线程的可管理性：线程是稀缺资源，而且也是任务中不可少的资源，如果频繁的且无限制的创建会消耗系统资源，降低系统稳定性导致系统崩溃，内存溢出等等问题
* Java提供了多种创建线程池的方式:

ExecutorService executors = Executors.newFixexThreadPool(3);创建固定大小的线程池，例子中只能放下3个线程

1. **系统：线程安全和线程不安全**

* 线程安全就是说多线程访问同一代码，不会产生不确定的结果。编写线程安全的代码是低依靠线程同步。
* 线程安全的类：StringBuffer 、HashTable（和HashMap功能都一样，唯一不一样的是：HashTable为线程安全和key/value不可以为null值）、Vector（JDK1.0时候出来）、stack（栈先进后出）
* 线程不安全：StringBuilder(相比StringBuilder效率更高)、HashMap（可以有null，值但仅能有一个）、ArrayList

1. **网络：TCP和UDP的区别**

* TCP面向连接（如打电话要先拨号建立连接），通过三次握手; UDP是无连接的，即发送数据之前不需要建立连接（发短信）
* TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达; UDP尽最大努力交付，即不保证可靠交付
* TCP面向字节流，是流模式，实际上是TCP把数据看成一连串无结构的字节流; UDP是面向报文的，是数据报模式
* UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）
* 每一条TCP连接只能是点到点的; UDP支持一对一，一对多，多对一和多对多的交互通信
* TCP首部开销20字节; UDP的首部开销小，只有8个字节
* TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道
* TCP对系统资源要求较多，适合传输大量数据; UDP对系统资源要求少，适合传输少量数据

全双工即Full duplex Communication,是指在通信的任意时刻,线路上存在A到B和B到A的双向信号传输

1. **Cpp：指针和引用的区别**

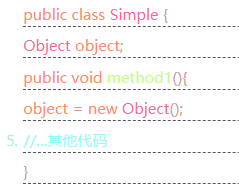
* 指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元，即指针是一个实体；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名
* 两者都是地址的概念；指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。
* 可以有const指针，但是没有const引用；
* 指针可以有多级，但是引用只能是一级（int \*\*p；合法 而 int &&a是不合法的）
* 指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；
* 指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了，从一而终。
* ”sizeof引用”得到的是所指向的变量(对象)的大小，而”sizeof指针”得到的是指针本身的大小；
* 指针和引用的自增(++)运算意义不一样

1. **Cpp：函数返回局部变量的引用会怎么样**

* 当函数返回引用类型时，没有复制返回值，相反，返回的是对象本身。
* 千万不要返回局部对象的引用！千万不要返回指向局部对象的指针！当函数执行完毕时，将释放分配给局部对象的存储空间。此时对局部对象的引用会指向不确定的内存！返回指向局部对象的指针也是一样的，当函数结束时，局部对象被释放，返回的指针就变成了不再存在对象的悬垂指针。
* 返回引用时，要求在函数的参数中，包含有以引用方式或指针方式存在的，需要被返回的参数。

1. **Cpp：内存泄露是什么**

* 不再会被使用的对象的内存不能被回收，就是内存泄露
* 在C++中，所有被分配了内存的对象，不再使用后，都必须程序员手动释放他们。所以，每个类，都会含有一个析构函数，作用就是完成清理工作，如果我们忘记了某些对象的释放，就会造成内存泄露。
* 但是在Java中，我们不用（也没办法）自己释放内存，无用的对象由GC自动清理，这也极大的简化了我们的编程工作。但，实际有时候一些不再会被使用的对象，在GC看来不能被释放，就会造成内存泄露



1. **Cpp：析构函数是不是虚函数**

* 构造函数不能为虚函数，而析构函数可以且常常是虚函数。
* C++对象在三个地方构建：（1）函数堆栈；（2）自由存储区，或称之为堆；（3）静态存储区。无论在那里构建，其过程都是两步：首先，分配一块内存；其次，调用构造函数。好，问题来了，如果构造函数是虚函数，那么就需要通过vtable 来调用，但此时面对一块 raw memeory，到哪里去找 vtable 呢？毕竟，vtable 是在构造函数中才初始化的啊，而不是在其之前。因此构造函数不能为虚函数。
* 析构函数可以是虚函数，且常常如此。因为此时 vtable 已经初始化了；况且我们通常通过基类的指针来销毁对象，如果析构函数不为虚的话，就不能正确识别对象类型，从而不能正确销毁对象。
* 每个析构函数（不加 virtual） 只负责清除自己的成员。但可能有基类指针，指向的确是派生类成员的情况。（这是很正常的）。那么当析构一个指向派生类成员的基类指针时，程序就不知道怎么办了。所以要保证运行适当的析构函数，基类中的析构函数必须为虚析构。

1. **Cpp：父类先析构还是子类先析构**

* 构造的时候先构造基类后子类 析构的时候相反先析构子类后析构基类

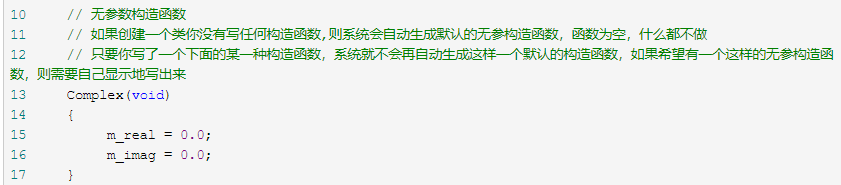
1. **Cpp：虚函数是什么**

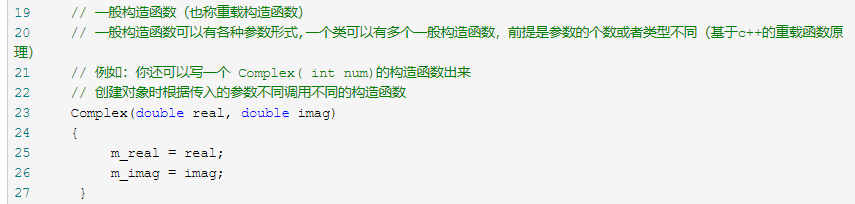
* 面向对象的语言有三大特性：继承、封装、多态。虚函数作为多态的实现方式，而多态体现在运行时绑定。通常通过基类指针指向子类对象实现多态。
* 其实更简单地来说，就是“在用父类指针调用函数时，实际调用的是指针指向的实际类型（子类）的成员函数”。
* 多态性使得程序调用的函数是在运行时动态确定的，而不是在编译时静态确定的。而虚函数则是加了virtual修饰词的类的成员函数。
* 什么样的函数不能声明为虚函数？1）不能被继承的函数。2）不能被重写的函数

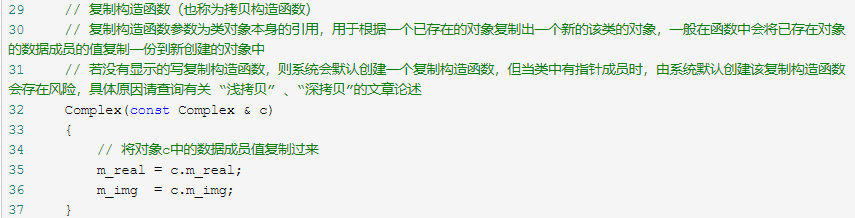
比如普通函数（只能重载不能重写），友元（不能继承），构造（不能继承），内联成员函数（编译展开），静态成员函数（编译确定，不能重写）

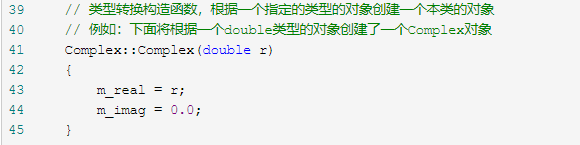
1. **Cpp：拷贝构造函数**

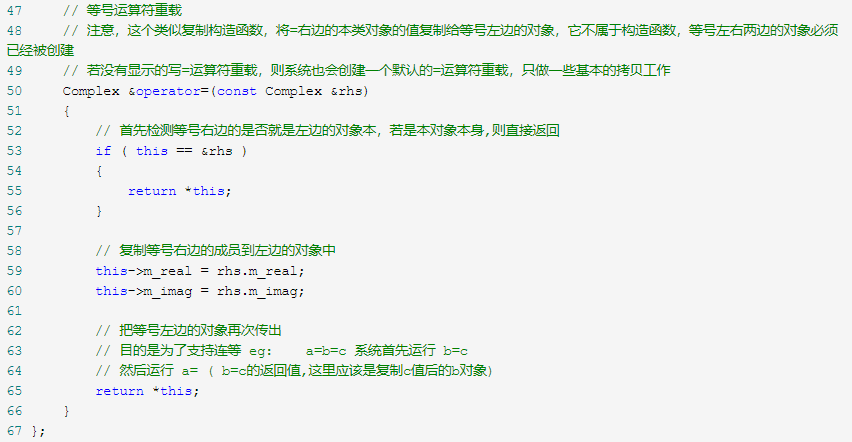
几种构造函数：无参数构造函数，一般构造函数，复制构造函数，类型转换构造函数，等号运算符重载











1. **Cpp：重载和重写**

方法重写(overriding)：

* 重写也叫子类的方法覆盖父类的方法，要求返回值、方法名和参数都相同。
* 子类抛出的异常不能超过父类相应方法抛出的异常。(子类异常不能超出父类异常)
* 子类方法的的访问级别不能低于父类相应方法的访问级别(子类访问级别不能低于父类访问级别)

方法重载(overloading):

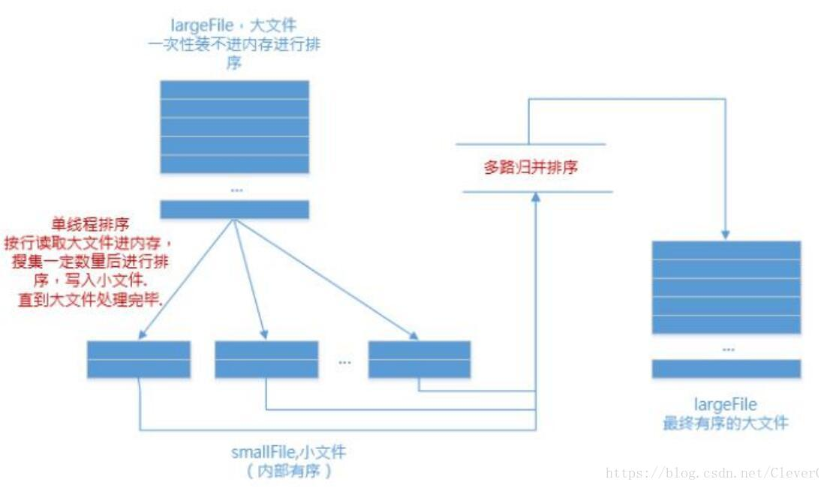
* 重载是在同一个类中的两个或两个以上的方法，拥有相同的方法名，但是参数却不相同，方法体也不相同，最常见的重载的例子就是类的构造函数，可以参考API帮助文档看看类的构造方法

1. **Cpp：内联函数和宏定义**

* 使用宏和内联函数都可以节省在函数调用方面所带来的时间和空间开销。二者都采用了空间换时间的方式，在其调用处进行展开
* 在预编译时期，宏定义在调用处执行字符串的原样替换。在编译时期，内联函数在调用处展开，同时进行参数类型检查。
* 内联函数首先是函数，可以像调用普通函数一样调用内联函数。而宏定义往往需要添加很多括号防止歧义，编写更加复杂。
* 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了(无法将this指针放在合适位置)。
* 可以用内联函数完全替代宏。在编写内联函数时，函数体应该短小而简洁，不应该包含循环等较复杂结构，否则编译器不会将其当作内联函数看待，而是把它决议成为一个静态函数。有些编译器甚至会优化内联函数，通常为避免一些不必要拷贝和构造，提高工作效率。频繁的调用内联函数和宏定义容易造成代码膨胀，消耗更大的内存而造成过多的换页操作。

1. **算法：如果要排序一个数组，1G，但只有100Mb的空间，用哪种排序算法**

* 多路归并排序



1. **算法：能说出几种排序方法**

* 插入排序，希尔排序，选择排序，堆排序，冒泡排序，快速排序
* 归并排序，基数排序，计数排序，桶排序

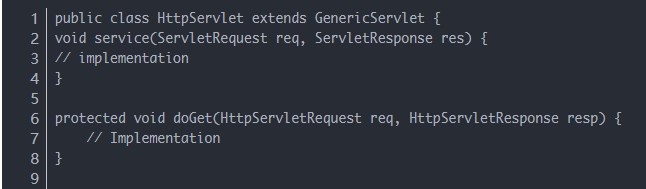


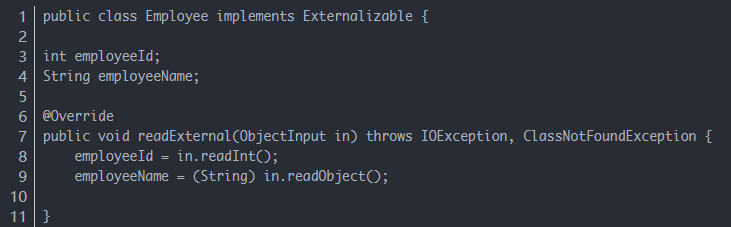
1. **数据库：事务是什么**

* 事务（Transaction)是由一系列对系统中数据进行访问与更新的操作所组成的一个程序 执行逻辑单元（Unit)。
* 狭义上的事务特指数据库事务。一方面，当多个应用程序并发访问数据库时，事务可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。另一方面，事务为数据库操作序列提供了一个从失败中恢复到正常状态的方法， 同时提供了数据库即使在异常状态下仍能保持数据一致性的方法。
* A，Automicity，原子性，强调事务作为原子级别已经不可以再被分割,要么成功要么失败。
* C，Consistency，一致性，即状态转换必须是由一种正确的状态转换到另外一种正确的状态。
* I，Isolation，隔离性，即相互间必须不能被影响。
* D，Durabillity，持久性，即事务提交后将被永久保存，即便出现其他故障，事务处理结果也应得到保存。

1. **Java：接口类和抽象类的区别**

* 抽象类是用来捕捉子类的通用特性的 。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板。
* 接口是抽象方法的集合。如果一个类实现了某个接口，那么它就继承了这个接口的抽象方法。这就像契约模式，如果实现了这个接口，那么就必须确保使用这些方法。接口只是一种形式，接口自身不能做任何事情。







1. **Java：Linkedlist和Arraylist的区别**

* ArrayList是基于动态数组的集合。LinkedList是基于链表的集合。
* ArrayList允许保存所有元素，包括null，并可以根据索引位置对集合进行快速访问；对于随机访问集合中的对象，使用LinkedList类实现List集合的效率较低。
* ArrayList向集合中指定位置插入对象或者删除元素的速度较慢；LinkedList需要向集合中插入和删除对象时，使用LinkedList集合的效率较高。
* Linkedlist不是线程安全的

1. **Java：Final关键字**

* final表面意思就是不可更改的,恒量的意思；类似于C语言中的const关键字，指的是无法改变的量，这与静态标量static是有区别的，静态变量指的是只有一份存储空间，值是可以改变的。
* 在Java中final修饰的就是常量，而且变量名要大写；
* 修饰变量，被final修饰的变量必须要初始化，赋初值后不能再重新赋值。注意：局部变量不在我们讨论的范畴，因为局部变量本身就有作用范围，不使用private、public等词修饰。
* 修饰方法，被final修饰的方法代表不能重写。
* 修饰类，被final修饰的类，不能够被继承。

1. **Java：Finalize关键字**

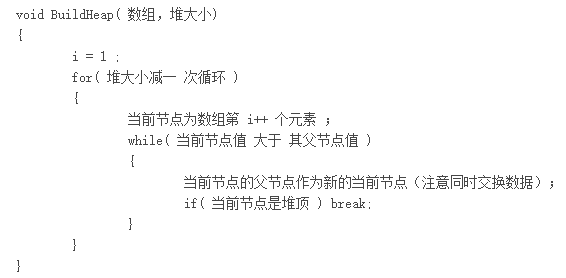
* finalize()是Object的protected方法，子类可以覆盖该方法以实现资源清理工作，GC在回收对象之前调用该方法。
* finalize()与C++中的析构函数不是对应的。C++中的析构函数调用的时机是确定的（对象离开作用域或delete掉），但Java中的finalize的调用具有不确定性
* 不建议用finalize方法完成“非内存资源”的清理工作，但建议用于：① 清理本地对象(通过JNI创建的对象)；② 作为确保某些非内存资源(如Socket、文件等)释放的一个补充：在finalize方法中显式调用其他资源释放方法。其原因可见下文[finalize的问题]

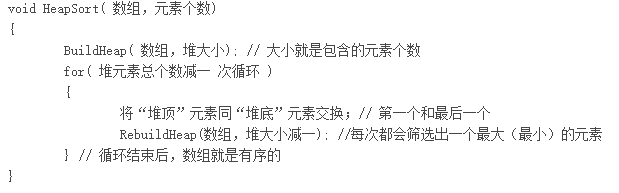
1. **Java：Finally关键字**

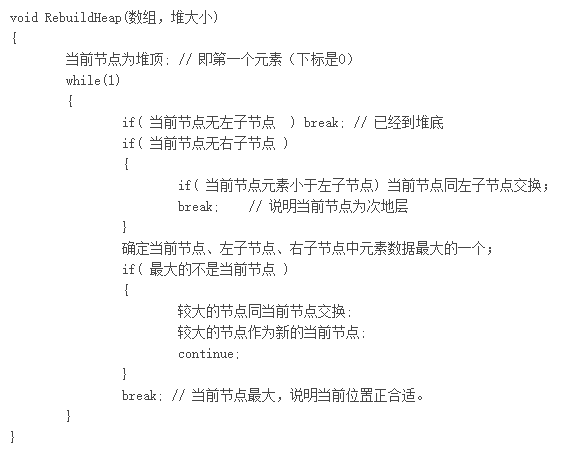
* finally作为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。

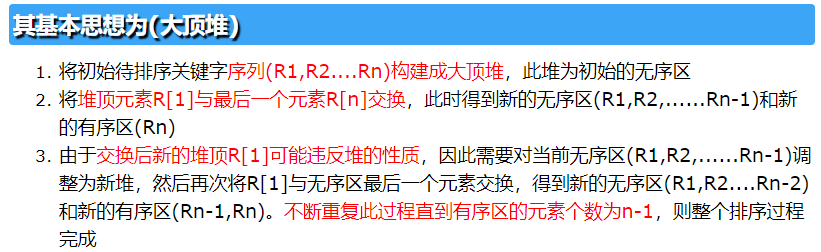
1. **算法：判断一个二进制数有几个末尾0：**mask，与0x1
2. **算法：如何判断一个链表有没有环：**快慢指针
3. **算法：leetcode原题 033 search in rotated sorted array:** 倒序数组Ologn查找
4. 安卓：用过了哪些控件
5. **数据结构：最小堆有什么特性**

* 最小堆是一种经过排序的完全二叉树，其中任一非终端节点的数据值均小于其左子节点和右子节点的值。即根节点的键值是所有堆节点键值的最小值









1. **系统：进程间通讯（IPC）有哪些实现方法**

* 管道：通常指无名管道，是UNIX系统IPC最古老的形式

它是半双工的（即**数据只能在一个方向上流动）**，具有固定的读端和写端。

它只能用于具有**亲缘关系的进程**之间的通信（也是父子进程或者兄弟进程之间）。

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

* FIFO，也成为明明管道，是一种文件类型

FIFO可以在**无关的进程**之间交换数据，与无名管道不同。

FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

* 消息队列：是消息的链列表，存放在内核中，一个消息队列由一个标识符（即队列ID）来标识

消息队列是**面向记录**的，其中的消息具有**特定的格式以及特定的优先级**。

消息队列**独立于发送与接收进程**。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的**随机查询**,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

* 信号量，与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于**实现进程间的互斥与同步**，而不是用于存储进程间通信数据。

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于**操作系统的 PV 操作**，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

* 共享内存，指两个或多个进程共享一个给定的存储区

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

因为多个进程可以同时操作，所以**需要进行同步**。

**信号量+共享内存**通常结合在一起使用，信号量用来同步对共享内存的访问。

1. **Cpp：纯虚函数的作用是什么**

定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

Virtual xxx () = 0;

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。抽象类的定义： 称带有纯虚函数的类为抽象类。

1. **Cpp：虚函数在底层具体是如何实现的**

虚函数将实现方法留给了编译器

编译器的处理方法是：给每个对象添加一个隐藏成员，隐藏成员中保存了一个指向函数地址数组的指针。这个数组成为虚函数表vtbl。

虚函数表中存储了未类对象进行声明的虚函数的地址，如果派生类提供了虚函数的新定义，该虚函数将保存新函数的地址

调用虚函数时，程序将查看存储在对象中vtbl地址，然后转向响应的函数地址表

使用虚函数时，在内存和执行速度方面有一定的成本，包括

1. 每个对象都将增大，增大量为存储地址的空间
2. 对于每个类，编译器都创建一个虚函数地址表的数组
3. 对于每个函数调用，都需要执行一项额外的操作，即到表中查找地址

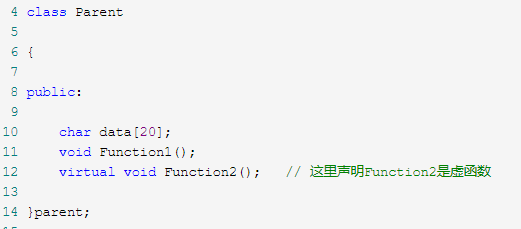
但是正是因为如此，虚函数才拥有了动态联编的能力

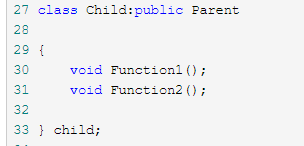
1. **Cpp：虚函数的使用方法**

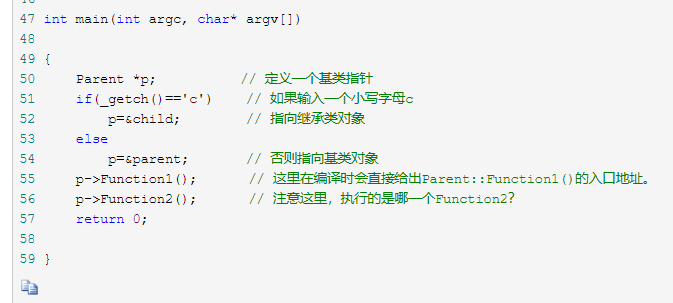
* 在基类用virtual声明成员函数为虚函数。这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必再加virtual。
* 在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。

C++规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加virtual，也可以不加，但习惯上一般在每一层声明该函数时都加virtual，使程序更加清晰。如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

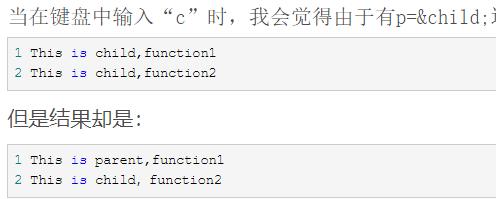
* 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。
* 通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。这就如同前面说的，不断地告诉出租车司机要去的目的地，然后司机把你送到你要去的地方。







虚函数最关键的特点是“动态联编”，它可以在运行时判断指针指向的对象，并自动调用相应的函数。





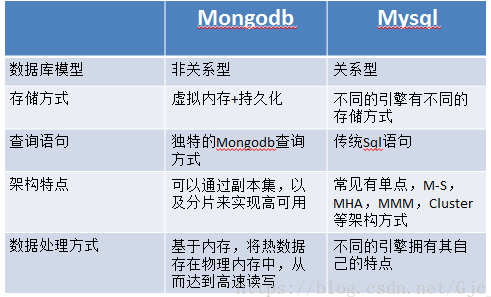
1. **Cpp：对stl库了解多少**

* STL是标准模板库的意思 standard template library
* 几个基本概念：容器 list，stack, vector, deque, queue, map, set

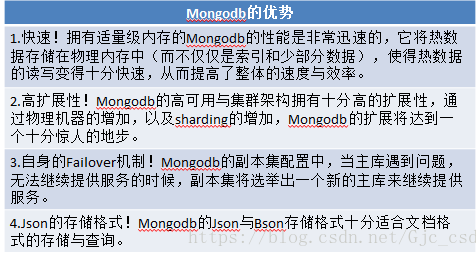
游标 iterator

算法 sort, count, for\_each, find 等

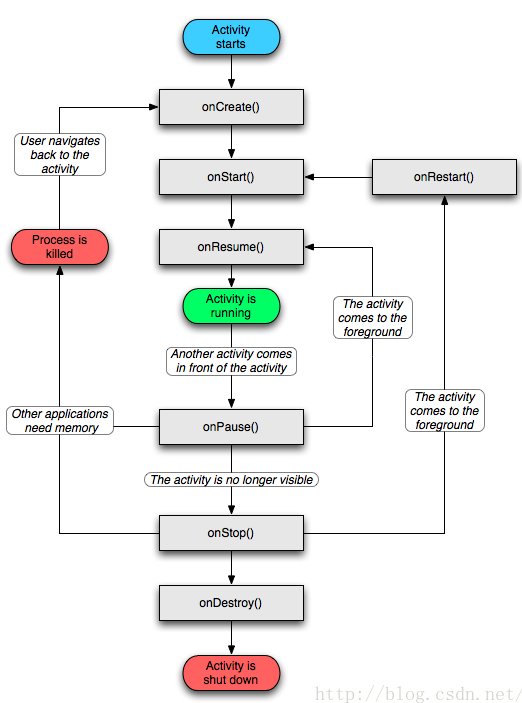
1. **数据库：Mongodb和mysql各有什么特点**







1. **安卓：生命周期**



* onCreate : 表示页面（Activity）的创建。（生命周期第一个阶段）功能：完成初始化工作，如：加载页面布局资源、初始化数据。
* onStart : 表示页面（Activity）正在被启动，即将开始。功能：页面为可见状态，但是无法与用户交互。
* onResume : 表示页面（Activity）出现在前台。功能：与 onStart 相比，onStart 处于后台，OnResume 才显示到前台。
* onPause : 表示页面（Activity）正在停止。功能：页面处于后台，正常情况下，onStop 紧接着执行。此时会做一些数据存储、停止动画不太耗时的工作。onPause 执行完新的页面（Activity）的 onResume 才会执行。
* onStop : 表示页面（Activity）即将停止。功能：页面为不可见状态，做稍微轻量级的不太耗时的回收工作。
* onDestroy : 表示页面（Activity）即将销毁。（生命周期最后一个阶段）功能：回收工作和资源的释放。
* onRestart : 表示页面（Activity）重新启动。功能：页面从不可见状态转化为可见状态时会调用此方法。如：Home 键切换页面（打开新的 Activity），然后回到页面过程中。
* 辨析一：onStart 和 onStop 是从 Activity 是否可见这个角度来回调的，而 onResume 和 onPause 是从是否位于前台来回调的。
* 辨析二：假设当前 Activity 为 A。如果用户打开一个新的 Activity B，那么 B 的 onResume 和 A 的 onPause 哪个先执行？

答： 根据Android的基本运行机制，不能再onPause中执行重量级的操作，因为必须onPause执行完成以后新Activity才能onResume。onPause和onResume都不能执行耗时的操作，尤其是onPause，这就意味着我们应该在onStop中做操作。从而使新的Activity显示出来并切换到前台。

* 案例：

当系统发生改变时，我们不想让 Activity 发生改变，比如，当我们旋转屏幕时，不想重新创建新的 Activity ，我们会怎么操作？

答：使用 android:configChanges="orientation" 属性。

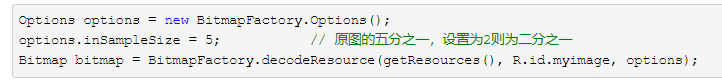
如果我们没有在 configChanges 属性中指定选项的话，当系统配置发生改变的话 Activity 就会被重新创建。

常用的属性有locale（设备的本地位置，一般指切换了系统语言），keyboardhidden，键盘可访问性，orientation 屏幕方向

这样，Activity 不会被创建，onSaveInstanceState 和 onRestoreInstanceState 方法不会被调用，取而代之，系统调用了 onConfigurationChanged 方法，这个时候我们可以做一些特殊的处理了。

1. **安卓：图片的内存管理（如何让图片节省内存）**

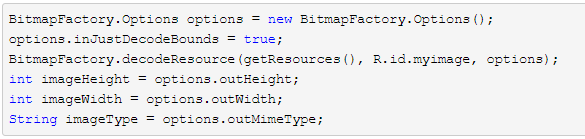
* 方法一：“让你的图片最小化”一节中描述的方法：使用尽可能小的图，使用.9，自己绘制背景或者使用Drawable来绘制背景
* 方法二：在内存中压缩图片，加载大图片时需要对图片进行压缩，使用等比例压缩方法直接在内存中处理图片



* 方法三：读取位图尺寸和类型时不把图片加载到内存中

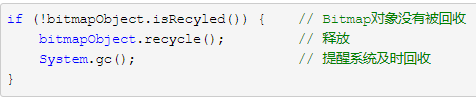
有时候我们取得一张图片，也许只是为了获得这个图片的一些信息，比如图片的width、height等信息，不需要显示到界面上，这个时候我们可以不把图片加载到内存中。

BitmapFactory的decodeResource属性



* 方法四：用完就回收

由于Android外层是使用java，而底层使用的是C语言为图片对象分配的内存空间。所以我们的外部虽然看起来释放了，但里层却并不一定完全释放了，我们使用完图片后最好再释放掉里层的内存空间。



1. **算法：leetcode原题 054 spiral matrix** : up down left right四个边缘指标
2. **算法：判断一颗树是不是二叉搜索树：**见github
3. 算法：如何用一个数组实现三个栈
4. **Java：垃圾回收机制**

所谓“要回收的垃圾”无非就是那些不可能再被任何途径使用的对象。

* 引用计数法

这个算法的实现是，给对象中添加一个引用计数器，每当一个地方引用这个对象时，计数器值+1；当引用失效时，计数器值-1。任何时刻计数值为0的对象就是不可能再被使用的。这种算法使用场景很多，但是，Java中却没有使用这种算法，因为这种算法很难解决对象之间相互引用的情况

* 可达性分析法

这个算法的基本思想是通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链（即GC Roots到对象不可达）时，则证明此对象是不可用的。

那么问题又来了，如何选取GCRoots对象呢？在Java语言中，可以作为GCRoots的对象包括下面几种：

(1). 虚拟机栈（栈帧中的局部变量区，也叫做局部变量表）中引用的对象。

(2). 方法区中的类静态属性引用的对象。

(3). 方法区中常量引用的对象。

(4). 本地方法栈中JNI(Native方法)引用的对象

1. **Java: 四种引用状态**

* 强引用

代码中普遍存在的类似"Object obj = new Object()"这类的引用，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

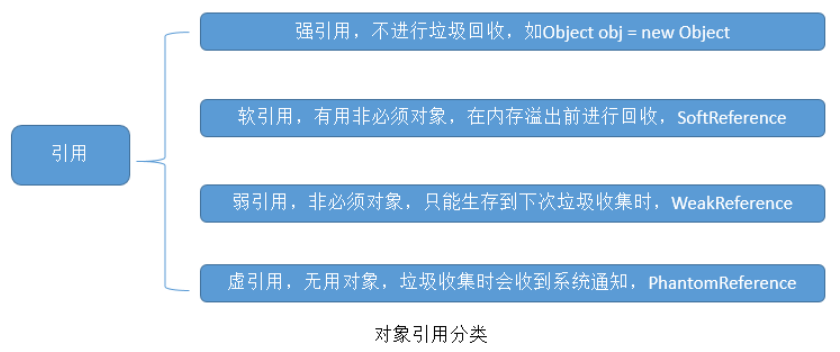
描述有些还有用但并非必需的对象。在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围进行二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。Java中的类SoftReference表示软引用。

* 弱引用

描述非必需对象。被弱引用关联的对象只能生存到下一次垃圾回收之前，垃圾收集器工作之后，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。Java中的类WeakReference表示弱引用。

* 虚引用

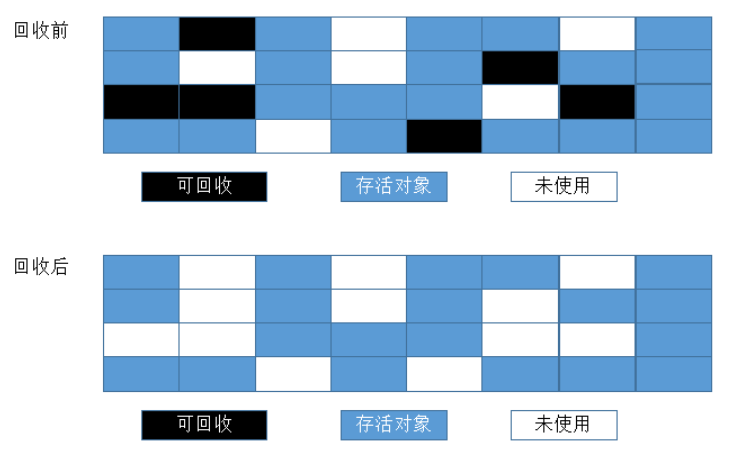
这个引用存在的唯一目的就是在这个对象被收集器回收时收到一个系统通知，被虚引用关联的对象，和其生存时间完全没关系。Java中的类PhantomReference表示虚引用。



1. **Java：垃圾收集算法**

* 标记-清除（Mark-Sweep）算法

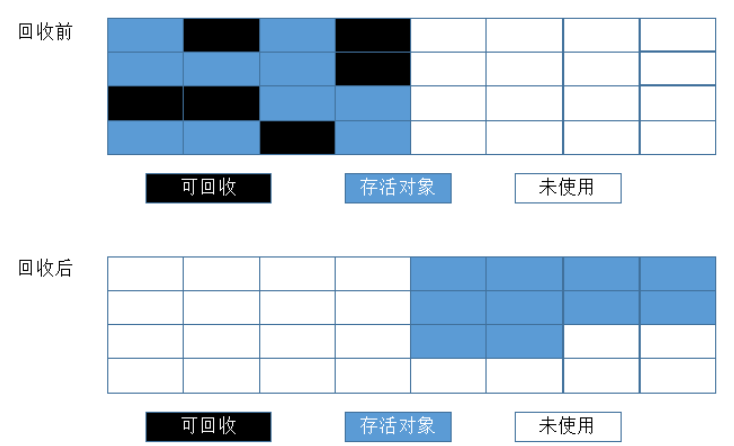
这是最基础的算法，标记-清除算法就如同它的名字样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。这种算法的不足主要体现在效率和空间，从效率的角度讲，标记和清除两个过程的效率都不高；从空间的角度讲，标记清除后会产生大量不连续的内存碎片，内存碎片太多可能会导致以后程序运行过程中在需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。标记-清除算法执行过程如图：



* 复制（Copying）算法

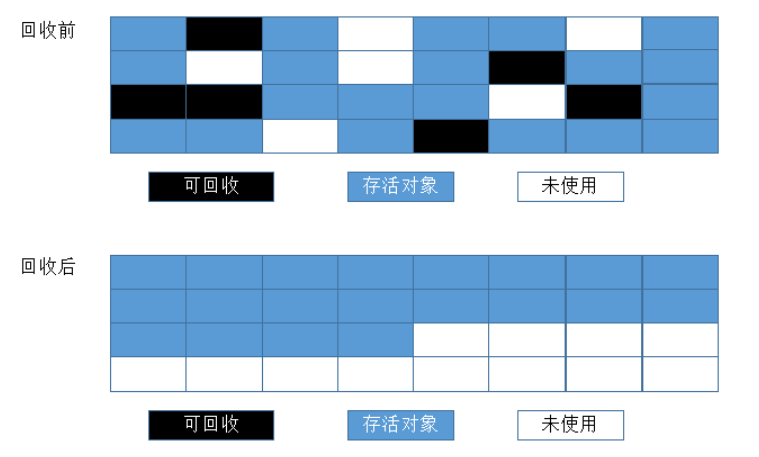
复制算法是为了解决效率问题而出现的，它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。这样每次只需要对整个半区进行内存回收，内存分配时也不需要考虑内存碎片等复杂情况，只需要移动指针，按照顺序分配即可。复制算法的执行过程如图

Log-structure File System

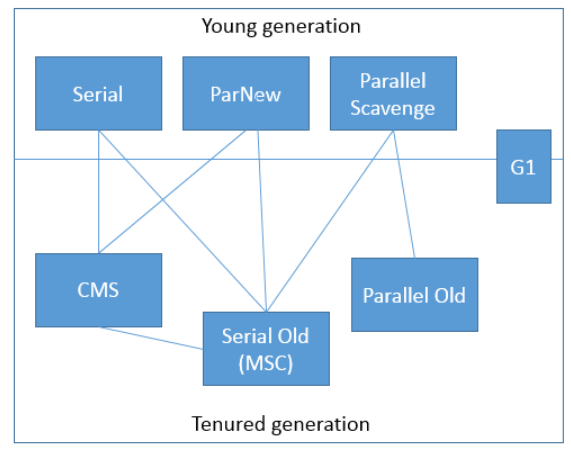


* 标记-整理（Mark-Compact）算法

复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。万一对象100%存活，那么需要有额外的空间进行分配担保。老年代都是不易被回收的对象，对象存活率高，因此一般不能直接选用复制算法。根据老年代的特点，有人提出了另外一种标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。标记-整理算法的工作过程如图：



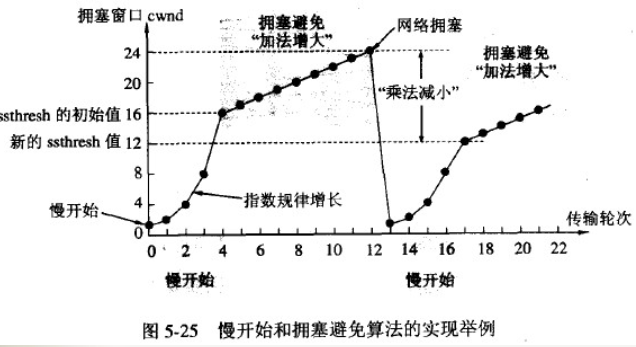
1. **Java ：垃圾收集器**

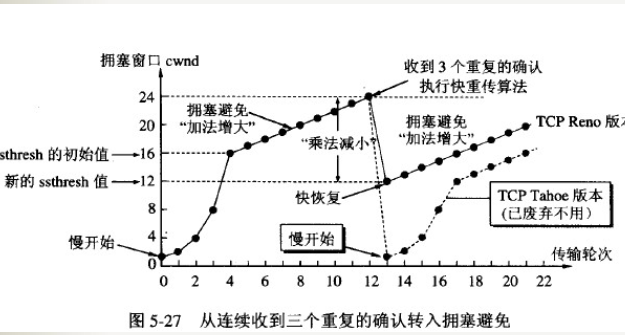


上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，那说明它们可以搭配使用。虚拟机所处的区域说明它是属于新生代收集器还是老年代收集器。多说一句，我们必须明确一个观点：没有最好的垃圾收集器，更加没有万能的收集器，只能选择对具体应用最合适的收集器。这也是HotSpot为什么要实现这么多收集器的原因。

1. **网络：TCP拥塞控制的机制**

慢启动，快恢复





1. 网络：Lock\_client如果挂了怎么办
2. 网络：如果网络通信不畅的话，怎么防止误伤
3. **数据库：如何保证数据库的强一致性**

数据库一致性（Database Consistency）是指事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。保证数据库一致性是指当事务完成时，必须使所有数据都具有一致的状态。在关系型数据库中，所有的规则必须应用到事务的修改上，以便维护所有数据的完整性。

更新缓存策略：

* 失效：应用程序从cache取数据，没有得到，则从数据库取数据，然后将数据放入cache
* 命中：应用从cache取数据，若取到则返回
* 更新：把数据更新到数据库，然后使cache失效

（更新）如果先使cache失效，后更新数据库，是否可行

这么做引发的问题是，如果A,B两个线程同时要更新数据，并且A,B已经都做完了删除缓存这一步，接下来，A先更新了数据库，C线程读取数据，由于缓存没有，则查数据库，并把A更新的数据，写入了缓存，最后B更新数据库。那么缓存和数据库的值就不一致了。

如果先把数据库更新，然后把cache更新（不是失效），是否可行

这么做引发的问题是，如果A,B两个线程同时做数据更新，A先更新了数据库，B后更新数据库，则此时数据库里存的是B的数据。而更新缓存的时候，是B先更新了缓存，而A后更新了缓存，则缓存里是A的数据。这样缓存和数据库的数据也不一致。

如果更新数据时，更新数据库成功，而失效cache时失败怎么解决？

1、对删除缓存进行重试，数据的一致性要求越高，我越是重试得快。

2、定期全量更新，简单地说，就是我定期把缓存全部清掉，然后再全量加载。

3、给所有的缓存一个失效期。（绝招，任何不一致，都可以靠失效期解决，失效期越短，数据一致性越高。但是失效期越短，查数据库就会越频繁。因此失效期应该根据业务来定）

1. 数据库：如何让大容量数据库的seek时间差不多（查询效率优化问题）
2. 数据库：如何避免数据堆积在同一个位置
3. React和小程序是怎么渲染的，一段js代码怎么成为最后呈现在用户眼前的内容
4. 进程的调度
5. 进程的生命周期
6. TCP的状态