# 事务属性

   共分为传播行为、隔离级别、只读和事务超时

* 传播行为：定义了被调用方法的事务边界。

PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。   
 PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。   
 PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。   
 PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。   
PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

* 隔离级别：在 spring 事务中提供了 5 种隔离级别来对应在SQL 中定义的 4 种隔离级别

|  |  |
| --- | --- |
| 隔离级别 | 意义 |
| ISOLATION\_DEFAULT | 使用后端数据库默认的隔离级别 |
| ISOLATION\_READ\_UNCOMMITTED | 允许读取未提交的数据（对应未提交读），可能导致脏读、不可重复读、幻读 |
| ISOLATION\_READ\_COMMITTED | 允许在一个事务中读取另一个已经提交的事务中的数据（对应已提交读）。可以避免脏读，但是无法避免不可重复读和幻读 |
| ISOLATION\_REPEATABLE\_READ | 一个事务不可能更新由另一个事务修改但尚未提交（回滚）的数据（对应可重复读）。可以避免脏读和不可重复读，但无法避免幻读 |
| ISOLATION\_SERIALIZABLE | 这种隔离级别是所有的事务都在一个执行队列中，依次顺序执行，而不是并行（对应可序列化）。可以避免脏读、不可重复读、幻读。但是这种隔离级别效率很低，因此，除非必须，否则不建议使用。 |

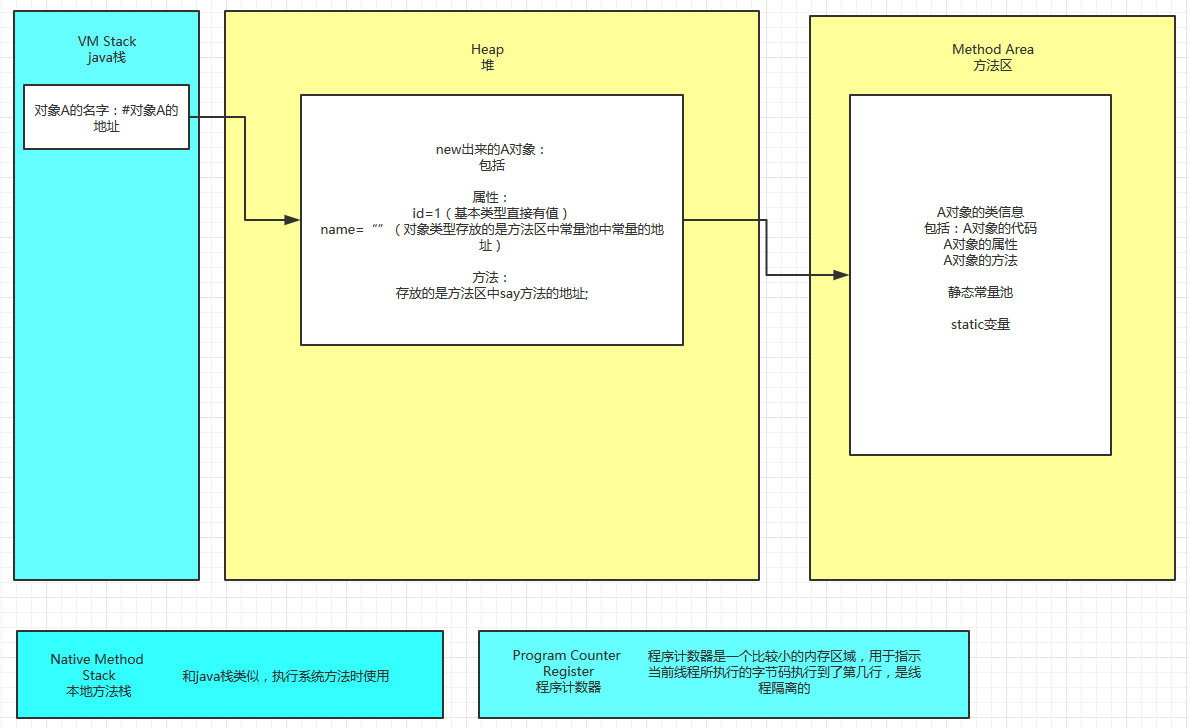
* 只读：如果在一个事务中所有关于数据库的操作都是只读的，也就是说，这些操作只读取数据库中的数据，而并不更新数据，那么应将事务设为只读模式（ READ\_ONLY\_MARKER ） , 这样更有利于数据库进行优化 。
* 事物超时：如果一个事务长时间运行，这时为了尽量避免浪费系统资源，应为这个事务设置一个有效时间，使其等待数秒后自动回滚。

# Servlet的生命周期

分为5个阶段：加载、创建、初始化、处理客户请求、卸载。

* 加载：容器通过类加载器使用servlet类对应的文件加载servlet
* (2)创建：通过调用servlet构造函数创建一个servlet对象
* (3)初始化：调用init方法初始化
* (4)处理客户请求：每当有一个客户请求，容器会创建一个线程来处理客户请求
* (5)卸载：调用destroy方法让servlet自己释放其占用的资源

# JVM





# 责任链模式

责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

使用场景：如登陆校验与角色校验。Java中的Filter（过滤器）和Interceptor（拦截器）

# HTTPS

http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议 http和https使用的是完全不同的连接方式用的端口也不一样：前者是80，后者是443。https协议需要到ca申请证书，一般免费证书很少，需要交费。　　http的连接很简单，是无状态的 HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议 要比http协议安全

# final 修饰符

可以修饰 class、field、method 等，被 final 修饰的类不可继承，被 final 修饰的 field 不可修改，被 final 修饰的 method 不可重写。

# 集合类的特性

* List: 元素有序 可重复
* set: 存储无序的，不可重复的元素 相当于高中的“集合”概念
* map: map 的主要实现类是 hashmap
* HashMap 是基于哈希表实现的，每一个元素都是一个 key-value 对；存取时间复杂度为常数阶O(1)

# HashMap底层实现原理

* HashMap储存的是键值对，存取很快，并允许使用null值和null键。此类不保证映射的顺序。
* HashMap实际上是数组和链表的结合体。
* 数组：存储区间连续，占用内存严重，寻址容易，插入删除困难； 链表：存储区间离散，占用内存比较宽松，寻址困难，插入删除容易； Hashmap综合应用了这两种数据结构，实现了寻址容易，插入删除也容易。

# String 类能继承吗

不可以，因为 String 类是被 final 修饰符修饰的，所以不能被继承。

# 单例类

* java.lang.Runtime 类
* java.lang.reflect.Proxy 类

# 线程调度

* 多线程有两种实现方法，分别是继承 Thread 类与实现 Runnable 接口  
  线程同步的实现方面有两种，分别是 synchronized 实现互锁，wait 与 notify 实现线程通信。
* sleep() 方法执行后线程进入阻塞状态：阻塞状态是需要外界激活，可能是时间， 可能是满足某一条件，他才能激活运行，就算 CPU 有空闲，也需要满足条件才会执行。
* yied() 方法执行完后线程进入就绪状态：就绪状态是只要 CPU 有空闲随时都可以执行，处于就绪状态
* Thread 类的 setPriority 函数可以设置优先级。

# Sleep和wait

* sleep()方法该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。
* sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持着，当指定的时间到了又会自动恢复运行状态。
* 在调用sleep()方法的过程中，线程不会释放对象锁。
* 而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备
* 获取对象锁进入运行状态。

# java中有内存泄露

* java 理论上是不存在内存泄露，因为有垃圾回收器（GC）对内存进行释放。
* 实际上，Java 也存在内存泄露，原因：主要是一些对象虽然不在被使用，但他们依然被引入（例如被长生命周期对象引用）；

# 垃圾回收机制

* 对象不再被引用，将会触发JVM的垃圾回收机制。检测对象是否还被引用有多重算法1、引用计数法 2、分代法（年轻代和老年代）。可以调用System.gc()方法，主动释放内存，但是不建议这么做，因为这是一个低优先级的线程，当它在工作时，整个程序会停止运行，直到该线程执行完毕。、
* 也可以实现finalize()方法，自己释放内存，建议将对象设置为null。
* 需要大量内存时，可能会触发System.gc()方法。

# 静态变量、成员变量

* 成员变量 属于对象，所以也称为实例变量。  
  静态变量 属于类，所以也称为类变量。
* 成员变量 存在于堆内存中。静态变量 存在于方法区中。
* 成员变量 随着对象创建而存在，随着对象被回收而消失。  
  静态变量 随着类的加载而存在，随着类的消失而消失。
* 成员变量 只能被对象所调用 。  
  静态变量 可以被对象调用，也可以被类名调用。  
  所以，成员变量 可以称为对象的特有数据，静态变量 称为对象的共享数据。
* 注意：静态方法中只能调用静态变量，不能调用非静态变量  
  因为类初始化的时候先加载静态方法，但是非静态变量这时候还没初始化，所以编译就会报错。

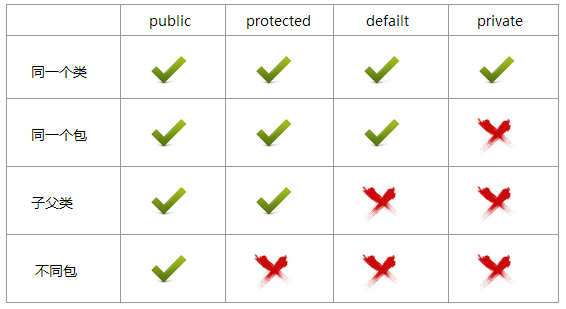
# Excption 与 Error 区别

* Error 表示系统级的错误和程序不必处理的异常，是一种严重问题；比如内存溢出，不可能指望程序能处理这样的状况；
* Exception 表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

# 子类能重写父类的构造函数吗

* 子类 不可以 重写 父类 的 构造方法，因为构造方法是父类特有的,子类根本继承不了父类的构造函数，所以子类不可以重写父类的构造方法。
* 但是可以使用关键字 super 调用父类的构造方法。

# 四个权限修饰符



# 堆、栈、方法区

* Java 程序运行时的内存分配策略有三种，分别是静态分配，栈式分配，和堆式分配；对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区）、栈区和堆区。
* 静态存储区（方法区）：主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。
* 栈区 ：当方法被执行时，方法体内的局部变量（其中包括基础数据类型、对象的引用）都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为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。
* 堆区 ： 又称动态内存分配，通常就是指在程序运行时直接 new 出来的内存，也就是对象的实例。这部分内存在不使用时将会由 Java 垃圾回收器来负责回收。

# Java IO流

它是一种数据的流从源头流到目的地。比如文件拷贝，输入流和输出流都包括了。输入流从文件中读取数据存储到进程(process)中，输出流从进程中读取数据然后写入到目标文件。

* 按操作方式（类结构）

字节流：以字节为单位，每次读入或读出是8位数据。可以读任何类型数据。

字符流：以字符为单位，每次读入或读出是16位数据。其只能读取字符类型数据。

* 输出流和输入流：

输出流：从内存读出到文件。只能进行写操作。

输入流：从文件读入到内存。只能进行读操作。

* Java流中的常见基类：

java.io.InputStream

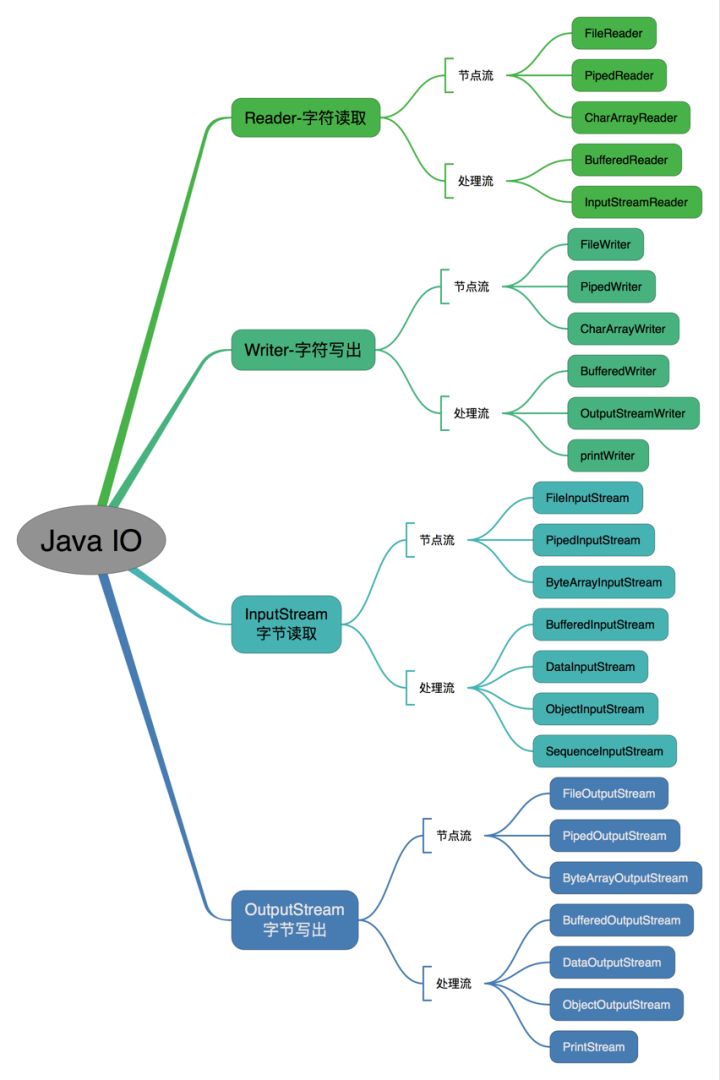
java.io.OutputStream

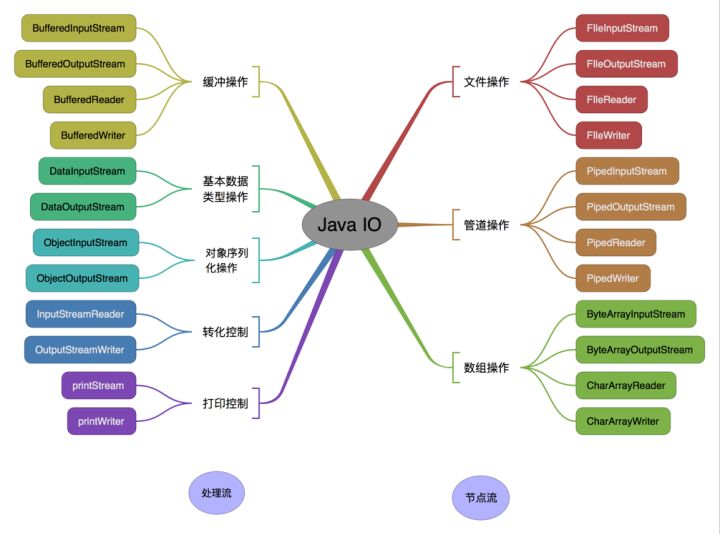
java.io.Reader

java.io.Writer

* FileInputStream和FileOutputStream是什么？

这是在拷贝文件操作的时候，经常用到的两个类。在处理小文件的时候，它们性能表现还不错，在大文件的时候，最好使用BufferedInputStream (或 BufferedReader) 和 BufferedOutputStream (或 BufferedWriter)





# Syncronized和Lock

synchronized：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。

lock：需要显示指定起始位置和终止位置。在解锁处需要通过lock()和unlock()显示指出。（所以一般会在finally块中写unlock()以防死锁。）

|  |  |  |
| --- | --- | --- |
| **类别** | **synchronized** | **Lock** |
| 存在层次 | Java的关键字，在jvm层面上 | 是一个类 |
| 锁的释放 | 1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁 | 在finally中必须释放锁，不然容易造成线程死锁 |
| 锁的获取 | 假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待 | 分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待 |
| 锁状态 | 无法判断 | 可以判断 |
| 锁类型 | 可重入 不可中断 非公平 | 可重入 可中断 可公平（两者皆可） |
| 性能 | 少量同步 | 大量同步 |

# JavaIO流关闭顺序

一般情况下是：先打开的后关闭，后打开的先关闭；

另一种情况：看依赖关系，如果流a依赖流b，应该先关闭流a，再关闭流b；

例如处理流a依赖节点流b，应该先关闭处理流a，再关闭节点流b

当然完全可以只关闭处理流，不用关闭节点流。处理流关闭的时候，会调用其处理的节点流的关闭方法

如果将节点流关闭以后再关闭处理流，会抛出IO异常

# Java锁

* 公平锁/非公平锁

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。

对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。

对于Synchronized而言，也是一种非公平锁。

* 可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。

对于Java ReentrantLock而言, 他的名字就可以看出是一个可重入锁，其名字是Re entrant Lock重新进入锁。

对于Synchronized而言,也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

* 互斥锁/读写锁

独享锁是指该锁一次只能被一个线程所持有。共享锁是指该锁可被多个线程所持有。

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。

读锁的共享锁可保证并发读是非常高效的，读写，写读 ，写写的过程是互斥的。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

对于Synchronized而言，当然是独享锁。

* 乐观锁/悲观锁

乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待并发同步的角度。  
悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。

乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。

从上面的描述我们可以看出，悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。

悲观锁在Java中的使用，就是利用各种锁。

乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。

* 自旋锁

在Java中，自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

# Java排序算法