# 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。

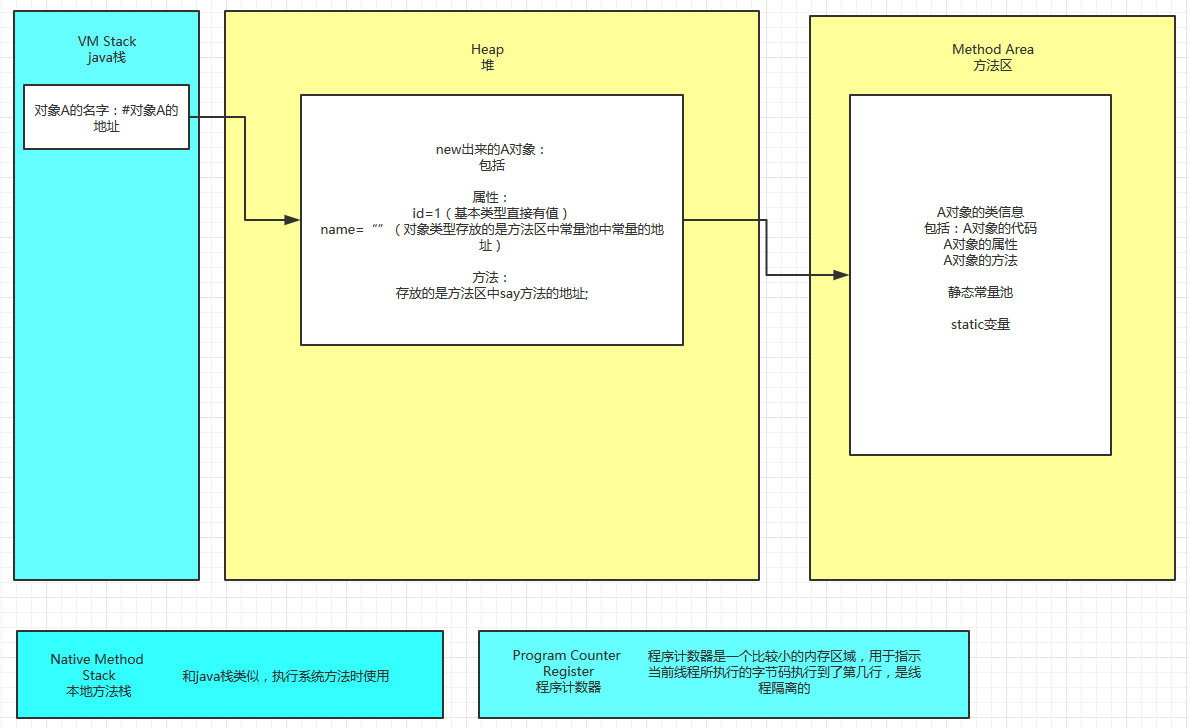
# Servlet的生命周期

servlet接口定义的是一套处理网络请求的规范，所有实现servlet的类，都需要实现它那五个方法，其中最主要的是两个生命周期方法 init()和destroy()，还有一个处理请求的service()，也就是说，所有实现servlet接口的类，或者说，所有想要处理网络请求的类，都需要明确初始化时要做什么、销毁时要做什么、接受到请求时要做什么。

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
* 创建：通过调用servlet构造函数创建一个servlet对象
* 初始化：调用init方法初始化
* 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
* 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。

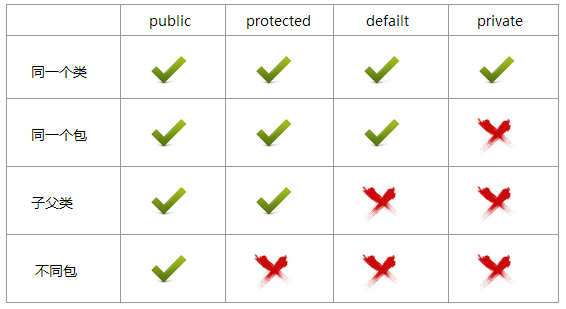
# Excption 与 Error 区别

* 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

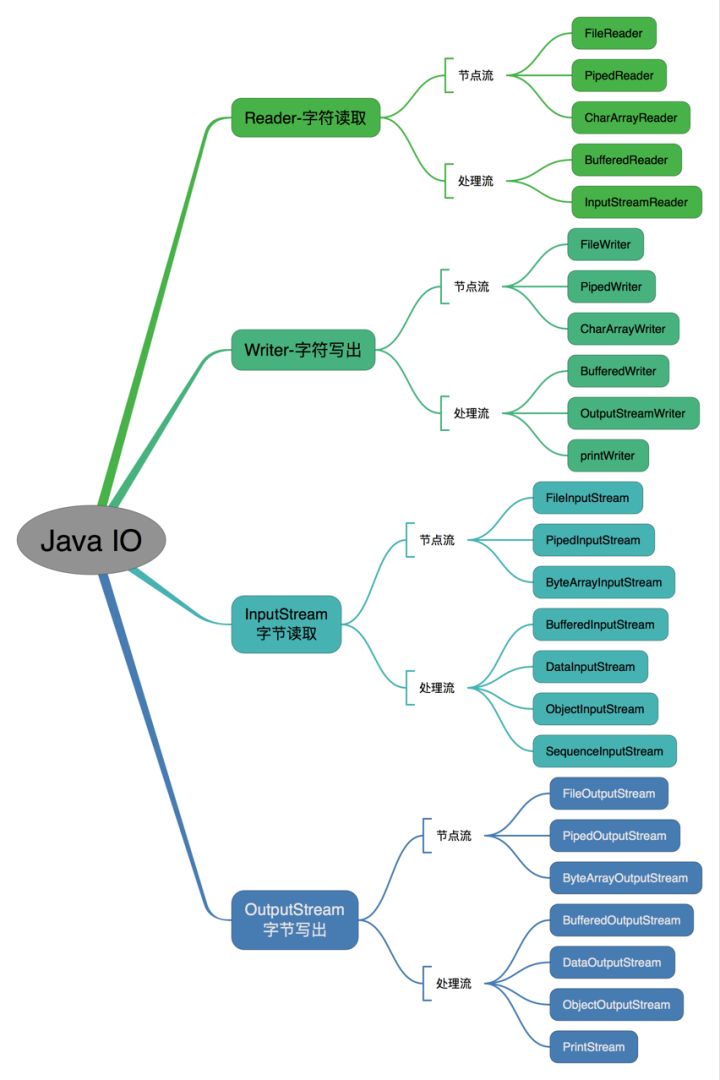
java.io.OutputStream

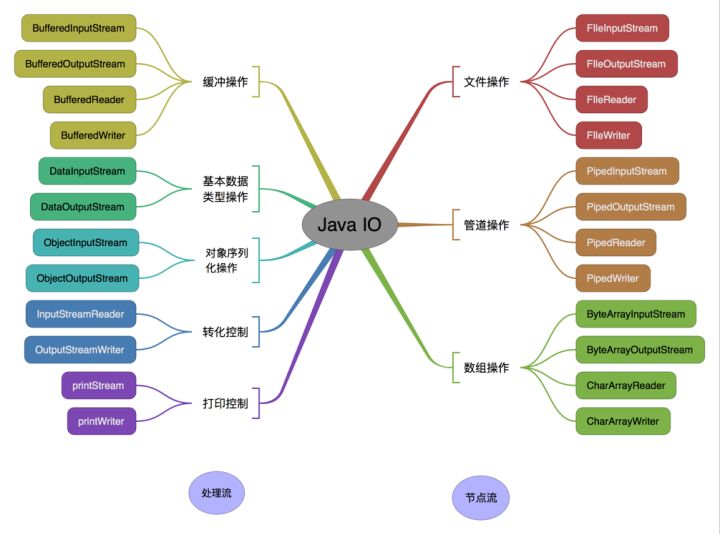
java.io.Reader

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。synchronized锁住的是括号里的对象，而不是代码。对于非static的synchronized方法，锁的就是对象本身也就是this。

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 可判断 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。

# 美团支付平台招聘

* 负责账务系统，清结算系统设计与开发；-
* 负责建设高并发，高性能，高可靠性的系统， 任职要求：
* 扎实的计算机专业基础；
* 至少精通一种技术框架，研究过源码；
* 较强的沟通能力，团队合作精神，以及表达能力；
* 熟悉常用的设计模式和设计思想
* 乐于分享自己的经验积累，成长体会；

# HashMap

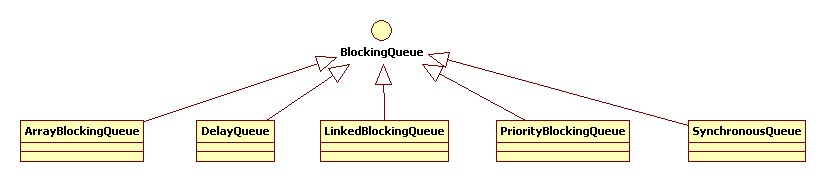
* + - HashMap底层基于数组和链表实现，以对Key值计算而得的Hashcode值为内存地址而存储，是无序的。初始容量为16，是2的幂，这是为了hash值的平均分布，和负载因子为0.75，当当前存储的数量>容量\*负载因子，自动ReHash进行扩容。
    - TreeMap和LinkedHashMap是有序的，其中LinkedHashMap继承至HashMap，其内部增加了一个链表，用以存放元素的顺序，默认是基于元素进入集合的顺序，性能高于HashMap。TreeMap实现了SortedMap 接口，可以排序，基于元素的固有顺序 (由 Comparator 或者 Comparable 确定)（即默认按照Key值递增排序），构造函数中可以传入比较器，性能略低于HashMap。
    - LinkedHashMap底层基于双向链表，TreeMap底层基于红黑树实现，在排序上个人认为TreeMap性能优于LinkedHashMap。

# 同步辅助类

* java.concurrent并发包。
* CountDownLatch是一个同步的辅助类，允许一个或多个线程，等待其他一组线程完成操作，再继续执行。
* CyclicBarrier是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同点，再继续执行。
* CycliBarrier实现原理：在CyclicBarrier的内部定义了一个Lock对象，每当一个线程调用CyclicBarrier的await方法时，将剩余拦截的线程数减1，然后判断剩余拦截数是否为0，如果不是，进入Lock对象的条件队列等待。如果是，执行barrierAction对象的Runnable方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从await方法返回，再从CyclicBarrier的await方法中返回。

# java.concurrent

* [**BlockingQueue**](http://www.cnblogs.com/jackyuj/archive/2010/11/24/1886553.html)：阻塞队列，线程安全，FIFO，生产者-消费者模式。



ArrayBlockingQueue:在生产者放入数据和消费者获取数据，都是共用同一个锁对象，由此也意味着两者无法真正并行运行，这点尤其不同于LinkedBlockingQueue；基于链表的阻塞队列，同ArrayListBlockingQueue类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而LinkedBlockingQueue之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

* ConcurrentLinkedQueue: 性能好于BlockingQueue。

它是一个基于连接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先加入的，尾是最后加入的，该队列不许有null元素。两者常用方法为offer和poll。

# IO包和NIO包

NIO vs IO之间的理念上面的区别（NIO将阻塞交给了后台线程执行）

* IO是面向流的，NIO是面向缓冲区的

Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方；

NIO则能前后移动流中的数据，因为是面向缓冲区的

* IO流是阻塞的，NIO流是不阻塞的

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。

Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。

* 选择器

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

NIO主要由Channels、Buffers、Selectors三部分组成。

# Java 类加载机制



* 加载：查找并加载类的二进制数据（查找和导入Class文件）
* 连接：1、验证：确保被加载的类的正确性

2、准备：为类的静态变量分配内存，并将其初始化为默认值

3、解析：把类中的符号引用转换为直接引用

* 初始化：对类的静态变量，静态代码块执行初始化操作

# 深拷贝和浅拷贝

# AtomicInteger：线程安全，内存可见性

* CAS（compare and swap）操作：

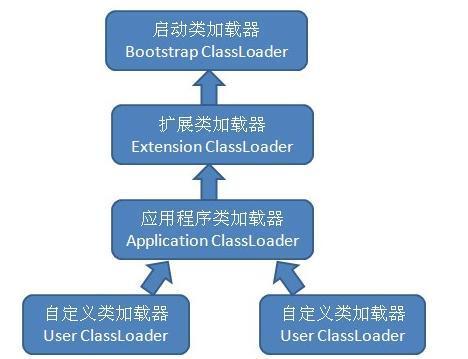
CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。而无需害怕其他线程同时修改变量，因为如果其他线程修改变量，那么CAS会检测它（并失败），算法可以对该操作重新计算。

* 首先使用了volatile 保证了内存可见性。

# 类加载器

启动类加载器和其它类加载器。启动类加载器在HotSpot虚拟机中使用C++语言实现，它是虚拟机的一部分；除了启动类加载器之外的其它类加载器都由Java语言实现，并且全部继承自java.lang.ClassLoader，它们是独立于虚拟机外部的。

* 双亲委派模型



某一个类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，则成功返回；如果父类加载器无法完成加载任务，将抛出ClassNotFoundException异常后，再调用自己的findClass()方法进行加载，依次类推。

作用：即防止内存中出现多份同样的字节码。

* 从本地系统直接加载
* 通过网络下载.class文件
* 从zip，jar等归档文件中加载.class文件
* 从专有数据库中提取.class文件
* 将Java源文件动态编译为.class文件（服务器）
* 命令行启动应用时候由JVM初始化加载
* 通过Class.forName()方法动态加载
* 通过ClassLoader.loadClass()方法动态加载

# Spring

* IOC容器：允许创建一个可以构造对象的应用环境，然后向这些对象传递它们的协作对象。正如单词 倒置 所表明的，IOC 就像反过来的JNDI。没有使用一堆抽象工厂、服务定位器、单元素（singleton）和直接构造（straightconstruction），每一个对象都是用其协作对象构造的。因此是由容器管理协作对象（collaborator）。 反射技术Class.forName。
* AOP框架：让开发人员可以创建非行为性的关注点，称为横切关注点，并将它们插入到应用程序代码中。使用 AOP后，公共服务（比如日志、持久性、事务等）就可以分解成方面并应用到域对象上，同时不会增加域对象的对象模型的复杂性。 动态代理技术ProviderFactory。
* 核心容器：核心容器提供 Spring框架的基本功能。核心容器的主要组件是BeanFactory，它是工厂模式的实现。BeanFactory使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
* Spring 上下文：Spring 上下文是一个配置文件，向 Spring框架提供上下文信息。Spring上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
* Spring AOP：通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了Spring框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于Spring的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖EJB组件，就可以将声明性事务管理集成到应用程序中。
* SpringDAO：JDBCDAO抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。SpringDAO的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。
* SpringORM：Spring 框架插入了若干个 ORM 框架，从而提供了 ORM的对象关系工具，其中包括JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和DAO异常层次结构。
* SpringWeb 模块：Web 上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
* SpringMVC框架：MVC 框架是一个全功能的构建 Web 应用程序的 MVC实现。通过策略接口，MVC框架变成为高度可配置的，MVC容纳了大量视图技术，其中包括JSP、Velocity、Tiles、iText 和 POI。

# JSP和Servlet

* 不同之处：

Servlet在Java代码中通过HttpServletResponse对象动态输出HTML内容

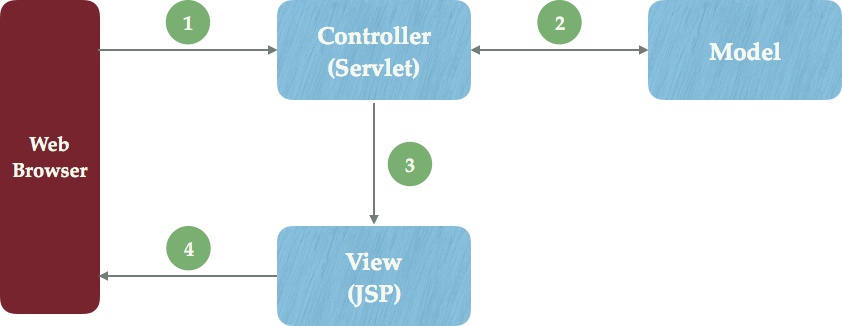
JSP在静态HTML内容中嵌入Java代码，Java代码被动态执行后生成HTML内容

* 各自的特点：

Servlet能够很好地组织业务逻辑代码，但是在Java源文件中通过字符串拼接的方式生成动态HTML内容会导致代码维护困难、可读性差

JSP虽然规避了Servlet在生成HTML内容方面的劣势，但是在HTML中混入大量、复杂的业务逻辑同样也是不可取的

* 通过MVC双剑合璧



# SpringMVC和Structs2

* Struts2 的核心是基于一个Filter即**StrutsPreparedAndExcuteFilter**   
  SpringMvc的核心是基于一个Servlet即**DispatcherServlet**(前端控制器)
* Struts2是基于**类**开发的，传递的参数是通过类的属性传递(属性驱动和模型驱动),所以只能设计成多例prototype

SpringMvc是基于**类中的方法**开发的，也就是一个url对应一个方法，传递参数是传到方法的形参上面，所以既可以是单例模式也可以是多例模式singiton

* Struts2每次发一次请求都会实例一个action，每个action都会被注入属性，一个Action对象对应一个request上下文。

SpringMvc采用request来解析请求内容，然后由其内部的getParameter给方法中形参赋值，再把后台处理过的数据通过ModelAndView对象存储，Model存储数据，View存储返回的页面，再把对象通过request传输到页面去，一个方法对应一个request上下文，容易实现restful url。

# MyBatis

MyBatis 是支持普通 SQL查询，存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML或注解用于配置和原始映射，将接口和 Java 的POJOs（Plain Ordinary Java Objects，普通的 Java对象）映射成数据库中的记录。

# Tomcat配置优化

* 内存优化
* 连接数优化：优化线程数、使用线程池
* Tomcat Connector三种运行模式：NIO（异步IO处理）

# Redis数据结构

* String
* List
* Set
* Hash
* Sorted Set

# Java设计模式

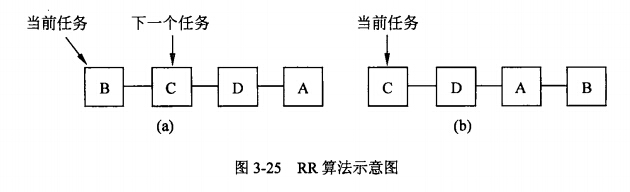
* 代理模式：静态代理（目标类和代理类都需要实现公共接口）、动态代理（目标类需实现接口，代理类为JDK提供的方法）
* 工厂模式：以苹果公司为例，其生产6、7、8三种产品；首先定义一个苹果接口，不同类型的苹果手机分别实现，再定义一个工厂接口，内部有一个返回苹果接口的方法，678分别实现该接口创建工厂实现类，满足开闭原则（对扩展开放，对修改关闭）。
* 单例模式：1、饿汉模式（静态方法返回静态实例变量，无线程安全问题，浪费内存）2、懒汉模式（延迟加载，用的时候再实例化，需要加锁）3、静态内部类（延迟加载、线程安全）4、枚举

# 任务调度算法

* **先来先服务算法：**最简单的一种调度算法是先来先服务算法（First Come First Served，FCFS），也叫做先进先出算法（First In First Out，FIFO）。优点：简单，易于理解和实现。

缺点：一批任务的平均周转时间取决于各个任务到达的顺序，如果短任务位于长任务之后，那么将增大平均周转时间。

* **短作业优先算法：**为了改进FCFS算法，减少平均周转时间，人们提出了短作业优先算法（Shortest Job First，SJF）。SJF算法的基本思路是：各个任务在开始执行之前，必须事先预计好它的执行时间，然后调度器将根据这些时间，从中选择用时较短的任务优先执行。   
  SJF算法有两种：不可抢占方式和可抢占方式。其中可抢占方式又叫最短剩余时间优先算法（Shortest Remaining Time First，SRTF）。
* 时间片轮转算法：下图所示是时间片轮转法的示意图。图a表示初始状态，总共有四个任务位于就绪队列中，先后顺序是B、C、D、A，其中任务B位于队列之首。当CPU空闲时，调度器就会选择B去执行。假设当B运行完一个时间片后，即没有结束也没有被阻塞，这时操作系统就会通过时钟中断来中止它的运行，并把它送到就绪队列的末尾。于是就成了图b的状态，此时，C位于队列之首，C就被调度执行。当C时间片用完后也会送到队列末尾，以此内推。当某个任务运行结束了，就会退出就绪队列，或者某任务被阻塞也也会退出，并加入阻塞队列中去。



* **优先级算法：**优先级调度算法（priority）的基本思路是：给每个任务都设置一个优先级，然后在任务调度的时候，在所有处于就绪状态的任务中选择优先级最高的任务去运行。上文提到的短作业优先算法其实也是一种优先级算法，每个任务的优先级就是它的运行时间，运行时间越短，优先级越高。

# Spring MVC工作原理



* 1、用户发送请求至前端控制器DispatcherServlet。
* 2、DispatcherServlet收到请求调用HandlerMapping处理器映射器。
* 3、处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
* 4、DispatcherServlet调用HandlerAdapter处理器适配器。
* 5、HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
* 6、Controller执行完成返回ModelAndView。
* 7、HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。
* 8、DispatcherServlet将ModelAndView传给ViewReslover视图解析器。
* 9、ViewReslover解析后返回具体View。
* 10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
* 11、DispatcherServlet响应用户。

组件：

1、前端控制器DispatcherServlet（不需要工程师开发）,由框架提供

作用：接收请求，响应结果，相当于转发器，中央处理器。有了dispatcherServlet减少了其它组件之间的耦合度。

用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的url查找Handler

HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器HandlerAdapter

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler

通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器Handler(需要工程师开发)

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler

Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）

View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。 springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图View(需要工程师开发jsp...)

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）