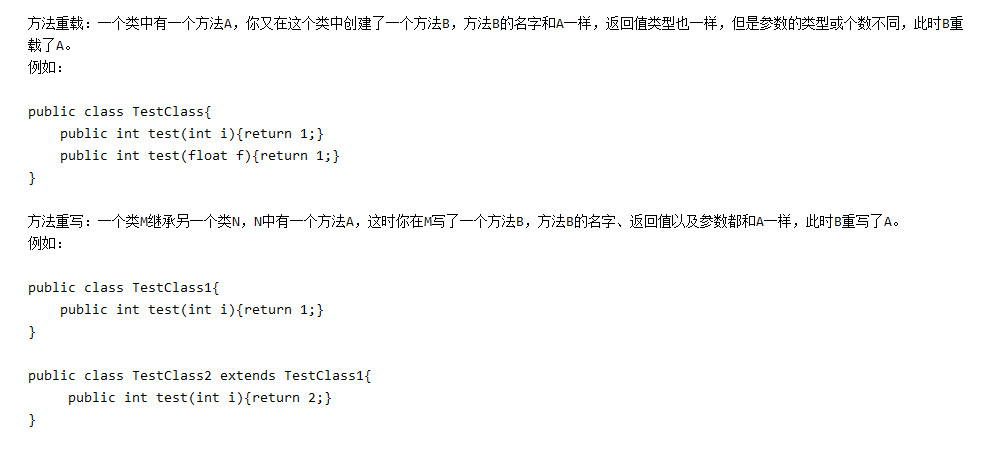
1.Java基础

1.1 面向对象（封装、继承、多态）

①访问修饰符的作用域：（**public、private、protectd、default**）：



1.2方法重写(Override)、重载(Overload)的概念和区别：



1.3接口和抽象类的区别：

* 一个类只能继承一个抽象类，但是可以实现多个接口（抽象类是单继承的，接口是多实现的）；
* 接口定义方法，不能实现；抽象类可以实现部分方法；
* 接口中数据类型为static，而抽象类不是；
* **当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口**。

1.4代码块执行顺序： **静态代码块->构造代码块->构造方法**

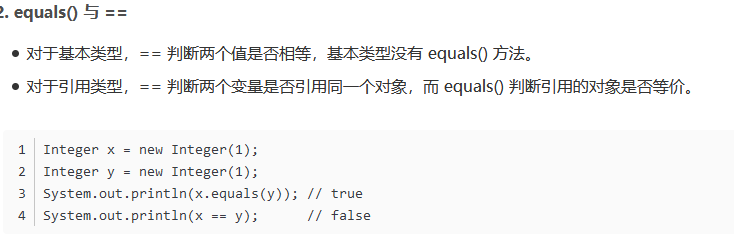
当一个类从被JVM装载开始，各种代码的执行顺序大致如下：

被JVM装载->执行父类的相关代码->**如果有静态初始化，先执行静态初始化**，且只执行一次，以后即使有该类实例化，也不会再执行->如果有静态代码块，以与静态初始化一样的方式执行->如果有new语句带来的实例化，先为成员变量分配空间，并绑定参数列表，隐式或显式执行super()，即父类的构造方法，->执行非静态代码块-〉执行本类的构造函数-〉执行其他代码。

1.5Object类常用方法：

Object类是一个特殊的类，是所有类的父类，如果一个类没有用extends明确指出继承于某个类，那么它**默认继承Object类。toString（）、equals（）、hashCode（）**。

* 取得对象信息的方法：toString()：该方法在打印对象时被调用，将对象信息变为字符串返回，**默认输出对象地址**，通常过重写Object类的toString()方法来输出对象属性信息。
* 对象相等判断方法：equals()：该方法用于比较对象是否相等，equals()方法比较的是**两个对象的地址**，所以必须重写方法才能到达目的。



* 对象签名:hashCode()：该方法用来返回其**所在对象的物理地址（哈希码值）**，常会和equals方法同时重写，确保相等的两个对象拥有相等的.hashCode。

1.6 String，StringBuffer，StingBuild各包含哪些方法，三者区别：

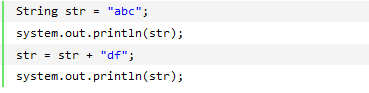
java.lang.String:简单字符串类型；

java.lang.StringBuffer:字符串缓冲区；

java.lang.StringBuilder:字符串缓存区。

https://www.cnblogs.com/xss512/p/10511366.html

1）首先说运行速度也就是执行速度，在这方面的**快慢**为 **StringBuilder > StringBuffer >String**。String最慢的原因是**：String为字符串的常量而StringBuilder和StringBuffer是字符串的变量，String一旦创建之后是不可更改的，而StringBuilder和StringBuffer是变量是可以更改的**。



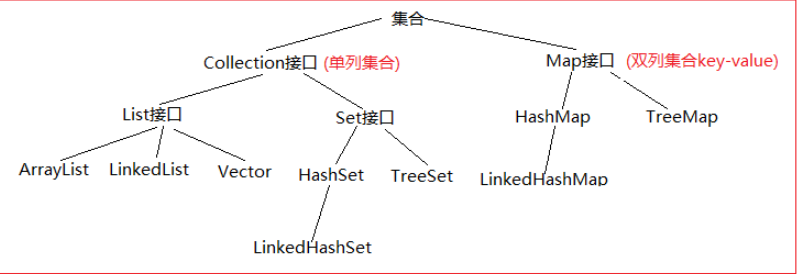
运行这段代码会发现先输出abc，然后输入的是abcdf，看似是把这个str对象给更改了，但是这只是一个种假象，**JVM**对于这几行代码是这样处理的，首先创建一个String对象str，并把“abc”赋值给str，然后在第三行中，其实JVM又创建了一个新的对象也名为str，然后再把原来的str的值和“df”加起来再赋值给新的str，而原来的str就会被JVM的垃圾回收机制（GC）给回收掉了，所以，str实际上并没有被更改，也就是前面说的String对象一旦创建之后就不可更改了。所以，**Java中对String对象进行的操作实际上是一个不断创建新的对象并且将旧的对象回收的一个过程，所以执行速度很慢**。而StringBuilder和StringBuffer的对象是**变量**，对变量进行操作就是直接对该对象进行更改，而不进行创建和回收的操作，所以速度要比String快很多。

2）**在线程安全上，StringBuilder是线程不安全的，而StringBuffer是线程安全的**。如果一个StringBuffer对象在字符串缓冲区被多个线程使用时，StringBuffer中很多方法可以**带有synchronized关键字**，所以可以保证线程是安全的，但StringBuilder的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。**所以如果要进行的操作是多线程的，那么就要使用StringBuffer，但是在单线程的情况下，还是建议使用速度比较快的StringBuilder**。

字符串String类型的操作有：

* 拼接：使用“+”运算符进行拼接；使用concat() 进行拼接。
* 查找：indexOf()—返回的是字符串的位置，从0开始
* 替换：replace()
* 长度: length()
* 去掉首尾空格： trim()
* 大小写：toUpperCase(),toLowerCase()
* 分隔： split()
* 比较：equals()
* 子串： substring(int,int)

2.集合



2。1Collection集合

①List: **有序（先后顺序），可重复**

* **ArrayList**：优点: 底层数据结构是数组，查询快，增删慢， 线程不安全。
* Vector：优点: 底层数据结构是数组，查询快，增删慢，线程安全。
* LinkedList：优点: 底层数据结构是链表，查询慢，增删快，线程不安全。

②Set: **无序，唯一**

* **HashSet**：底层数据结构是哈希表。(无序,唯一)
* TreeSet：底层数据结构是链表和哈希表。(FIFO插入有序,唯一)。
* LinkedSet：底层数据结构是红黑树。(唯一，有序)





2.2 Map集合

①HashMap、TreeMap、HashTable

* TreeMap是有序的，HashMap和HashTable是无序的。
* **Hashtable的方法是同步的，HashMap的方法不是同步的**。这是两者最主要的区别。
* Hashtable是线程安全的，HashMap不是线程安全的。
* HashMap效率较高，Hashtable效率较低。
* 如果对同步性或与遗留代码的兼容性没有任何要求，建议使用HashMap。 查看Hashtable的源代码就可以发现，除构造函数外，**Hashtable的所有 public 方法声明中都有 synchronized关键字**，而HashMap的源码中则没有。
* Hashtable不允许null值，HashMap允许null值（key和value都允许）
* 父类不同：**Hashtable的父类是Dictionary，HashMap的父类是AbstractMap**

**2.3 HashTable的实现原理：**

底层由**数组+链表**实现，有一个**叫table的Entry类型**数组，索引由**链表**存放。

1)Hashtable继承于Dictionary类，实现了Map接口。

Map是"key-value键值对"接口，Dictionary是声明了操作"键值对"函数接口的抽象类。

2)Hashtable是通过**"拉链法"**实现的哈希表。

它包括几个重要的成员变量：table, count, threshold, loadFactor, modCount。

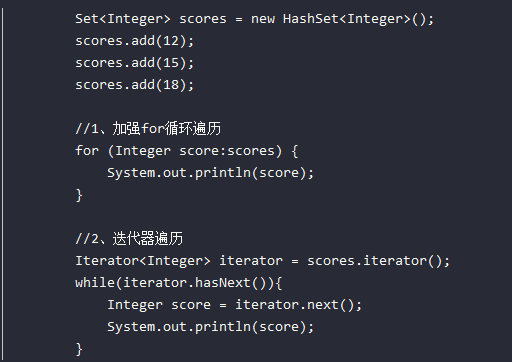
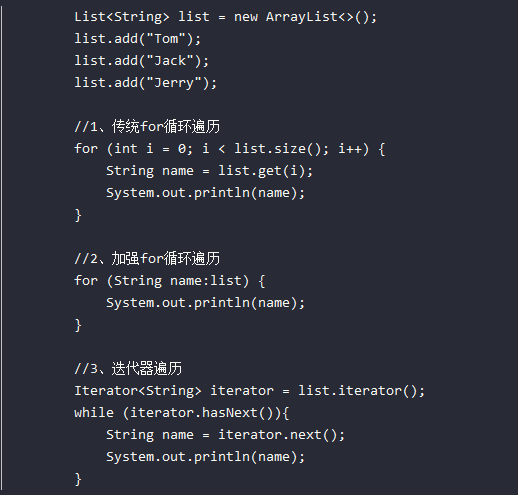
**table是一个Entry[]数组类型**，而**Entry实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的**。

count是Hashtable的大小，它是Hashtable保存的键值对的数量。

threshold是Hashtable的阈值，用于判断是否需要调整Hashtable的容量。threshold的值="容量\*加载因子"。

loadFactor就是加载因子。

modCount是用来实现fail-fast机制的



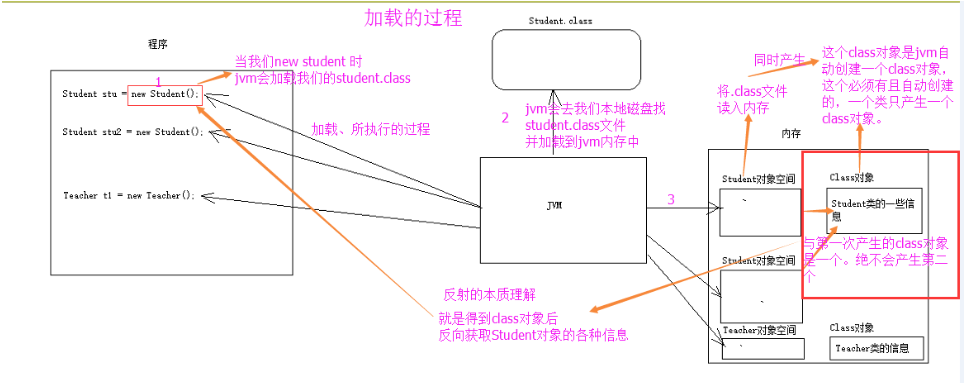


**3.反射：**

①JAVA反射机制是**在运行状态中**，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。**反射就是把java类中的各种成分映射成一个个的Java对象**。

**②使用反射可以赋予jvm动态编译的能力；**

**③使用的前提条件：必须先得到代表的字节码的Class，Class类用于表示.class文件（字节码）；**



获取class对象的三种方式：

**class.forName("类名")；（常用）**

类名.class；

对象名.getClass()



**二、SSM框架**

1.Spring

①什么是Spring？

Spring是一个轻量级的IOC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。

②什么是IOC？

**IOC，控制反转，让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的**。

Spring的IOC有三种注入方式（DI 依赖注入） ：**构造器注入、setter方法注入、根据注解注入**。

③什么是AOP？

AOP，一般称为**面向切面**，作为面向对象的一种补充，**用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。**

④BeanFactory和ApplicationContext的区别？

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中**ApplicationContext是BeanFactory的子接口**。其中BeanFactory在实际开发中用的不多。

* BeanFactory是**Spring里面最底层的接口**，包含了各种**Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系**。
* ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：**继承MessageSource**，因此支持国际化；统一的资源文件访问方式；提供在监听器中注册bean的事件；**同时加载多个配置文件**；载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。
* BeanFactroy采用的是**延迟加载**形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。
* ApplicationContext，它是在容器启动时**，一次性创建了所有的Bean**。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。
* BeanFactory通常以**编程的方式**被创建，ApplicationContext还能以**声明的方式**创建，如使用ContextLoader。

⑤Spring Bean的生命周期？



⑥Spring中的几种bean的作用域？

（1）singleton：**默认，每个容器中只有一个bean的实例**，单例的模式由BeanFactory自身来维护。

（2）prototype：**为每一个bean请求提供一个实例**。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

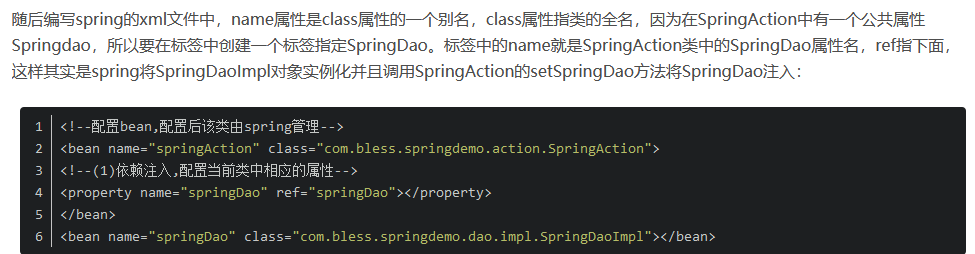
（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

⑦Spring基于XML注入bean的几种方式：

**构造器方式；set（）方式**

静态工厂、实例工厂



⑧Spring的自动装配；**@Autowired**

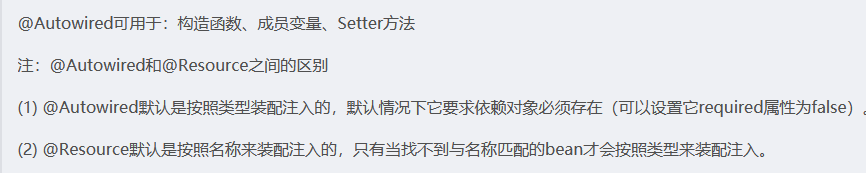
基于注解的方式：使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要**在Spring配置文件进行配置，<context:component-scan base-package=” ” />**。在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到**@Autowied、@Resource或@Inject**时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。

在使用@Autowired时，首先在容器中查询对应类型的bean：

如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；

如果查询的结果不止一个，那么@Autowired会**根据名称**来查找；

如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false



⑧Spring事务管理

基于**@Transactional注解**的声明式事务管理。如何在事务处理中捕获异常？

（1）Spring根据配置完成事务定义，设置事务属性；

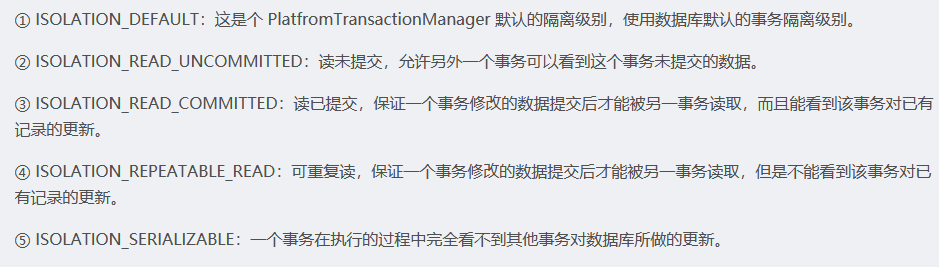
（2）执行开发者的代码逻辑；

（3）如果开发者的代码产生异常并且满足事务回滚的配置条件，则事务回滚，否则，事务提交；

（4）事务资源释放。

**捕获异常**：修改@Transactional注解：@Transactional(rollback={Exception.className}); 在catch语句中添加“throw new RuntimeException()”

⑨Spring中的隔离级别



⑩**开启注解**：<mvc:annontation-driven/>

**扫描机制**：<context:component-scan base-package="包名 " />

2.SpringMVC

①**SpringMVC的工作原理**：

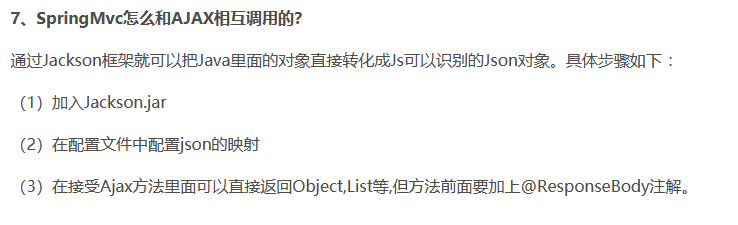
* 客户端请求提交给DispatcherServlet;
* DispatcherServlet根据请求，获得资源识别符URI，调用HandlerMapping，找到处理请求的Controller；
* DispatcherServlet将请求提交到Controller
* Cotroller调用业务逻辑处理请求，返回ModelAndView；
* DispatcherServlet调用ViewResolver视图解析器，找到ModelAndView指定的视图；
* View负责将结果显示到客户端

DispatcherServlet， HandlerMapping, Controller，ViewResolver

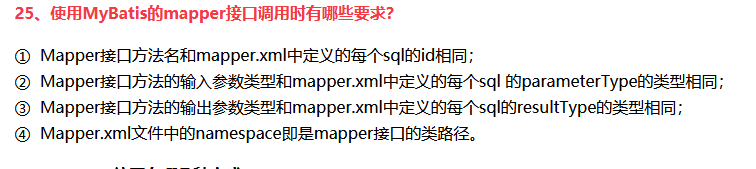
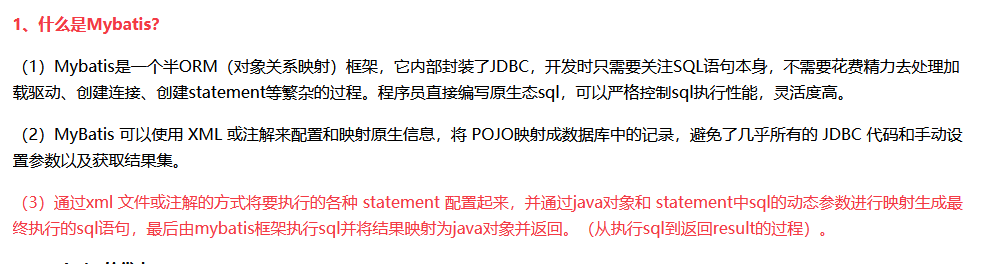
②如何解决get和post方法乱码的问题？

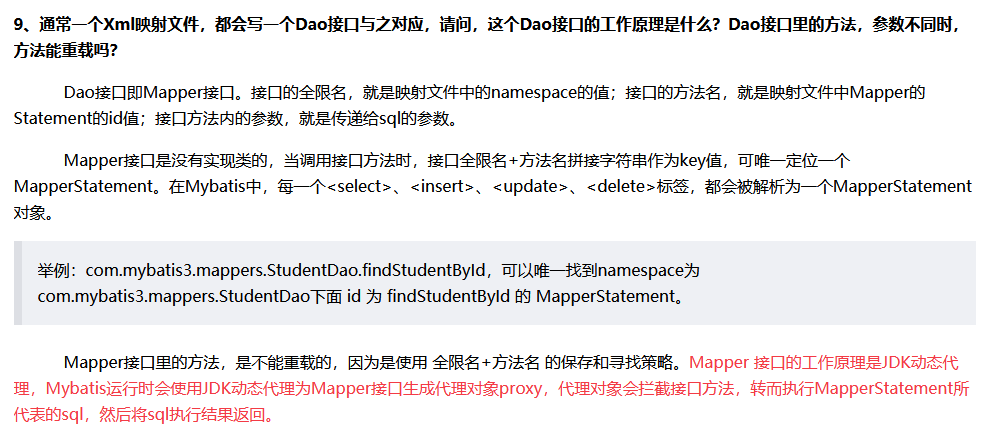
（1）post乱码：在web.xml中配置字符过滤的filter，CharacterEncodingFilter

（2）get乱码：



3.MyBatis



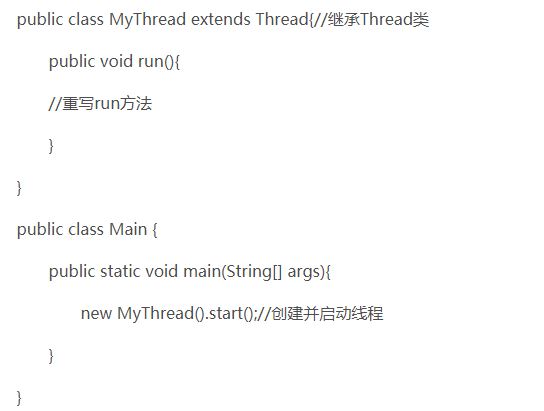


**三、多线程**

1.线程的创建：

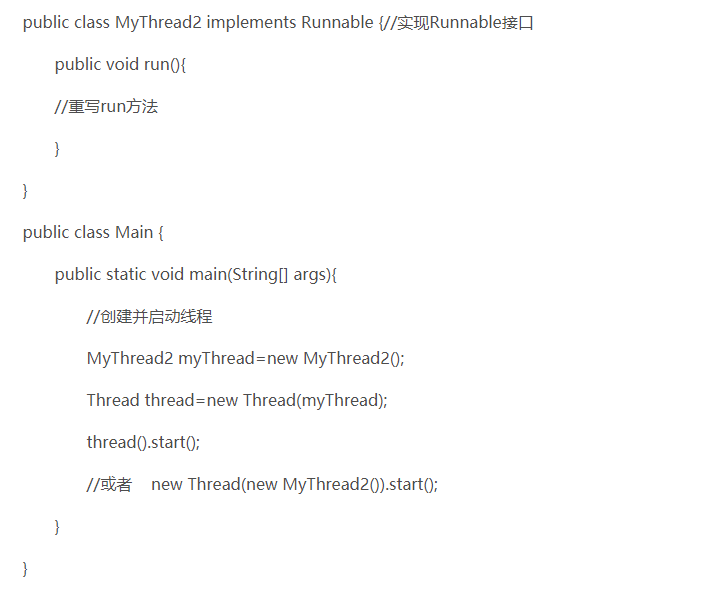
①继承Thread类

* 定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务run()方法也称为线程执行体；
* 创建Thread子类的实例，也就是创建了线程对象；
* 启动线程，即调用线程的start()方法。



②实现Runnable接口

* 定义Runnable接口的实现类，一样要重写run()方法，这个run（）方法和Thread中的run()方法一样是线程的执行体
* 创建Runnable实现类的实例，并用这个实例作为Thread的参数来创建Thread对象，这个Thread对象才是真正的线程对象
* 通过调用线程对象的start()方法来启动线程



③使用Callable和Future创建线程；

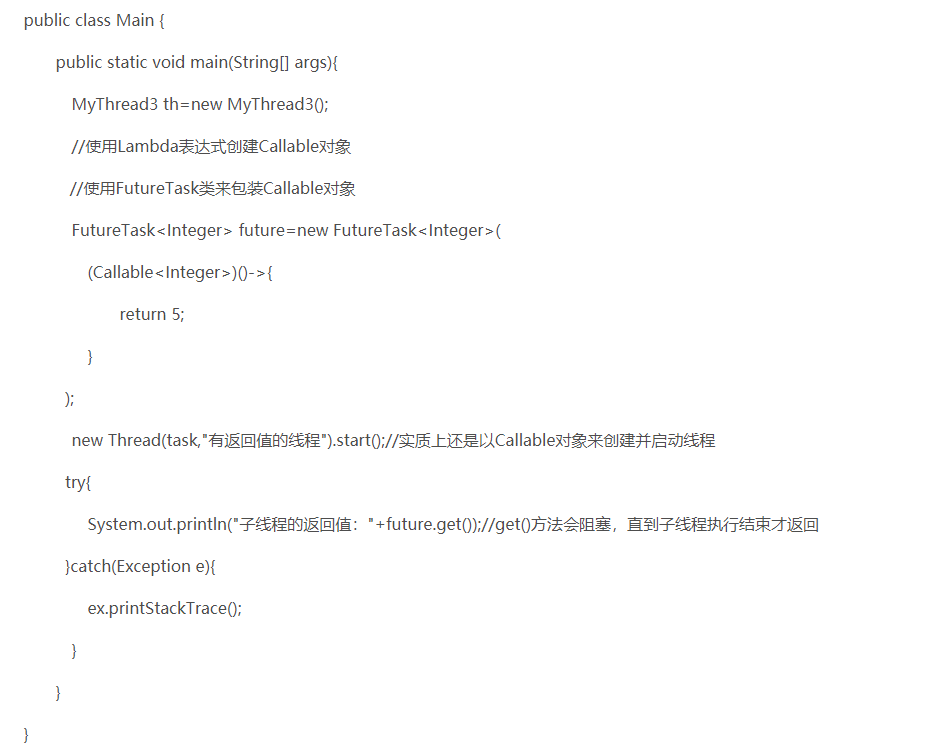
Callable接口提供一个call（）方法作为线程执行体，call（）方法比run（）方法更加强大：

1）call（）方法可以有返回值；

2）call（）方法可以声明抛出异常；

Java5**提供了Future接口来代表Callable接口里call()方法的返回值，并且为Future接口提供了一个实现类FutureTask**，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。

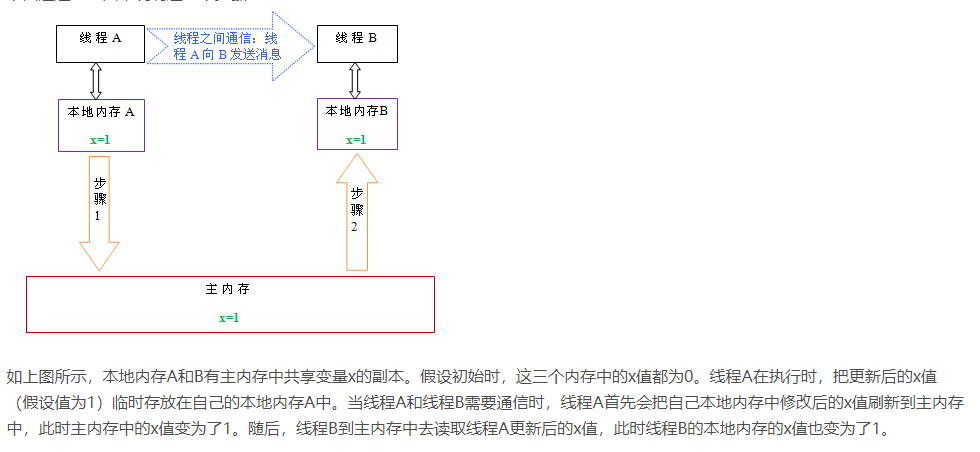
* 创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接使用Lambda表达式创建Callable对象）。
* 使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值
* 使用FutureTask对象作为Thread对象的参数创建并启动线程（因为FutureTask实现了Runnable接口）
* 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

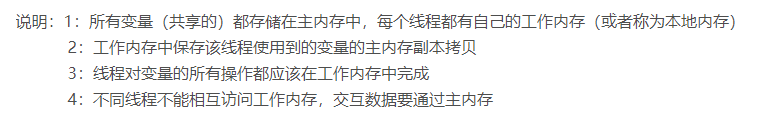


④使用线程池如用Executor框架

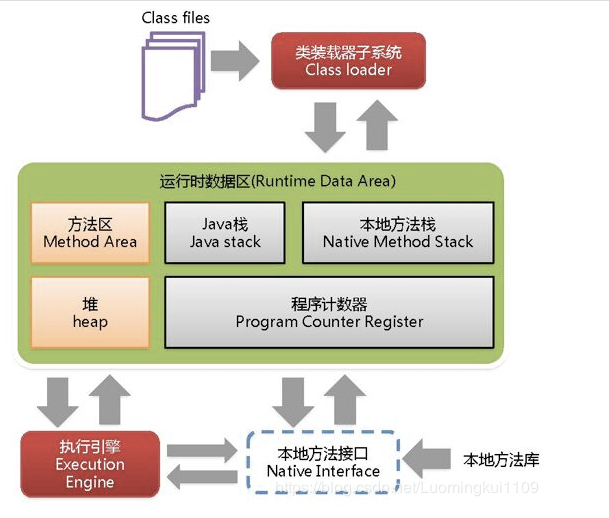
**2.Java内存模型：**

**Java内存模型，规定所有的变量存储在主内存中，每个线程还有自己的工作内存，线程的工作内存中保存了被该线程使用的变量的副本，线程对变量的所有操作(读取、赋值等)都必须在工作内存中进行，而不能直接读写主内存中的数据。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。**





3.JVM的基本原理



https://www.cnblogs.com/yuechuan/p/8984262.html



**静态变量+常量+类信息+运行时常量池存在方法区中，**

**对象实例变量存在堆内存中；**

**基本类型的变量和对象的引用变量都是在函数的栈内存中分配。**

4.线程同步&线程通信

①synchronized关键字**修饰方法或代码段**，实现数据的互斥访问。

②volatile关键字**修饰变量**，实现多线程下数据的同步。

③ReentrantLock**可重入锁**，实现数据的互斥访问；

④synchronized关键字结合Object的wait和notify方法，实现线程间的等待通信机制；

⑤ReentrantLock结合Condition接口的await()和signal()方法，实现线程间的等待通知机制。

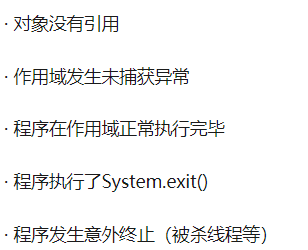
5.垃圾回收GC（Garbage Collection）机制

在Java中，程序员**不用去关心内存动态分配和垃圾回收的问题**，这些都交给JVM来处理。

**标记垃圾的算法？？以及回收垃圾的算法？？**

**GC的原理：**当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。

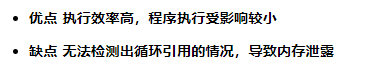
**GC判定为垃圾的标准**：



**GC标记垃圾（即如何确定某个对象是垃圾）的算法：**

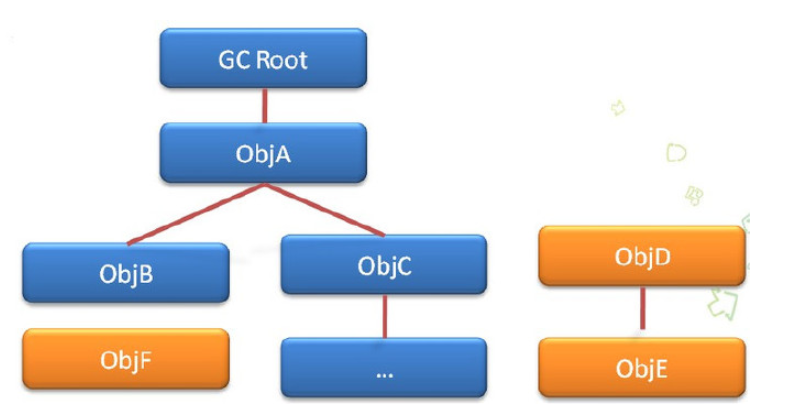
**①引用计数法：**

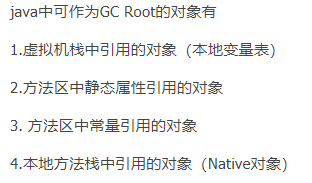
在java中是通过引用来和对象进行关联的，也就是说如果要操作对象，必须通过引用来进行。那么很显然一个简单的办法就是通过**引用计数来判断一个对象是否可以被回收**。不失一般性，如果一个对象没有任何引用与之关联，则说明该对象基本不太可能在其他地方被使用到，那么这个对象就成为可被回收的对象了。



②根搜索算法（可达性分析算法）GCRoot

根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点





**GC垃圾回收的算法：**

①标记-清除算法：首先标记出所有的对象，标记完成后统一回收

效率问题：标记和清除两个过程效率都不高

空间问题：产生碎片

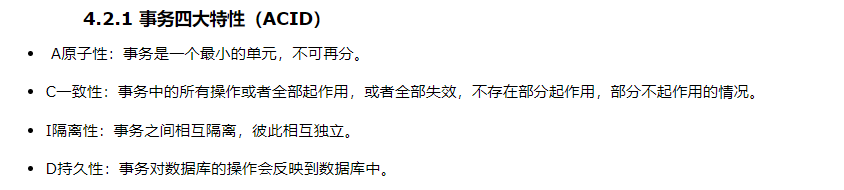
②复制算法：将内存划分为一块较大的Eden空间（80%）和两块较小的Survivor空间（10%），每次使用Eden和其中的一块Survivor，当回收时，将两者中存活的对象一次性复制到另一块Survivor，并清空刚才用到的空间，如果这块Survivor不够，则启用分配担保机制，将多处的对象存储再老年代

③标记-整理算法：首先标记出所有的对象，回收时让存活的对象都向一端移动，直接清理端边界外的内存

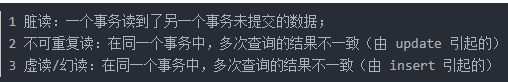
④**分代收集算法：将java堆分为新生代和老年代，垃圾回收时，新生代每次都有大量对象死去，所以采用复制算法，老年代存活的对象较多，使用标记-清除或标记-整理**。

6.事务

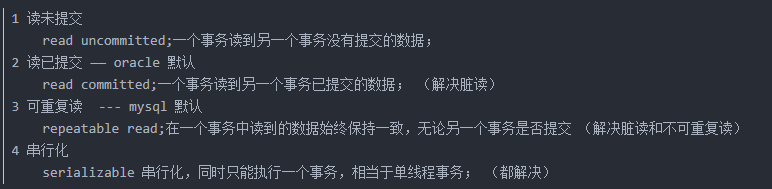
6.1 事务的四大特性：



6.2事务并发访问引起的问题



6.3 隔离的级别



2020年下半年需要学习的内容：

**7.Redis**

**8.Linux**

**9.分布式技术**