# JAVA复习资料

目录

[JAVA基础 1](#_Toc517880895)

[1、Java 和 C++的区别 1](#_Toc517880896)

[2、值传递和引用传递 1](#_Toc517880897)

[3、静态变量和实例变量的区别 1](#_Toc517880898)

[4、JDK, JRE 和 JVM 的区别 2](#_Toc517880899)

[5、Java 中的 final关键字用法 3](#_Toc517880900)

[6、final, finally, finalize 的区别 3](#_Toc517880901)

[7、String 和StringBuilder、StringBuffer 的区别 3](#_Toc517880902)

[8、为什么String要设计成不可变的 4](#_Toc517880903)

[9、错误和异常的区别(Error vs Exception) 4](#_Toc517880904)

[10、java值传递问题 5](#_Toc517880905)

[11、变量赋值顺序 5](#_Toc517880906)

[12、 深拷贝与浅拷贝 6](#_Toc517880907)

[13、访问权限 6](#_Toc517880908)

[14、super 6](#_Toc517880909)

[15、覆盖与重载 6](#_Toc517880910)

[16、Java面向对象的三个特征与含义 7](#_Toc517880911)

[17、接口和抽象类的区别 7](#_Toc517880912)

[18、两个对象值相同(x.equals(y) == true)，但却可有不同的 hash code，这句话对不对？ 8](#_Toc517880913)

[19、Java泛型 8](#_Toc517880914)

[JAVA集合 9](#_Toc517880915)

[1、List、Map、Set 三个接口存储元素时各有什么特点 10](#_Toc517880916)

[2、Map、Set、List、Queue、Stack的特点与用法 10](#_Toc517880917)

[3、Array和ArrayList有何区别？什么时候更适合用Array 11](#_Toc517880918)

[4、ArrayList 和 LinkedList 11](#_Toc517880919)

[5、ArrayList和Vector异同点 11](#_Toc517880920)

[6、HashMap的工作原理 12](#_Toc517880921)

[7、HashMap实现原理 12](#_Toc517880922)

[8、如何决定选用HashMap还是TreeMap 14](#_Toc517880923)

[9、HashMap和HashTable的区别 14](#_Toc517880924)

[10、HashSet,TreeSet,LinkedHashSet 之间的区别 15](#_Toc517880925)

[11、HashMap ,LinkedHashMap ,TreeMap,WeakHashMap,ConcurrentHashMap,IdentityHashMap的区别 15](#_Toc517880926)

[12、hashCode()和equals()方法有何重要性 16](#_Toc517880927)

[13、Iterator是什么 17](#_Toc517880928)

[14、集合框架中的泛型有什么优点 17](#_Toc517880929)

[JAVA虚拟机 18](#_Toc517880930)

[1、 介绍JVM中7个区域，然后把每个区域可能造成内存的溢出的情况说明 18](#_Toc517880931)

[2、 GC的两种判定方法：引用计数与根搜索算法 20](#_Toc517880932)

[3、 Java中的四种引用 20](#_Toc517880933)

[4、 内存溢出和内存泄漏 21](#_Toc517880934)

[5、 内存溢出了怎么办 21](#_Toc517880935)

[6、 什么时候会发生jvm堆（持久区）内存溢出 21](#_Toc517880936)

[7、 Java 中有内存泄露吗？ 22](#_Toc517880937)

[8、 GC 是什么？为什么要有 GC 22](#_Toc517880938)

[9、 GC的收集方法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？ 23](#_Toc517880939)

[10、 GC收集器有哪些？CMS收集器与G1收集器的特点。 23](#_Toc517880940)

[11、 Minor GC与Full GC分别在什么时候发生？ 24](#_Toc517880941)

[12、 内存分配策略 25](#_Toc517880942)

[JAVA类的加载机制 26](#_Toc517880943)

[1、 类的生命周期 26](#_Toc517880944)

[2、 JVM 加载 class 文件的原理机制 29](#_Toc517880945)

[3、 双亲委派模型 30](#_Toc517880946)

[JAVA并发 31](#_Toc517880947)

[1、 线程状态转化 31](#_Toc517880948)

[2、 使用线程 32](#_Toc517880949)

[3、 基础线程机制 32](#_Toc517880950)

[4、 中断 32](#_Toc517880951)

[5、 互斥同步 33](#_Toc517880952)

[6、 synchronized 和 ReentrantLock 比较 34](#_Toc517880953)

[7、 线程之间的协作 35](#_Toc517880954)

[8、 实现线程的类 35](#_Toc517880955)

[9、 线程不安全 37](#_Toc517880956)

[10、 JAVA内存模型 37](#_Toc517880957)

[11、 原子性 39](#_Toc517880958)

[12、 原子性实现原理和实现机制 39](#_Toc517880959)

[13、 线程安全分类 39](#_Toc517880960)

[14、 线程安全的实现方法 40](#_Toc517880961)

[15、 锁优化 41](#_Toc517880962)

[16、 Volatile的理解 42](#_Toc517880963)

[17、 Shutdown和shutdownNow的区别 43](#_Toc517880964)

[18、 Sleep()和wait()的区别 43](#_Toc517880965)

[19、 Sleep()和yield()的区别 43](#_Toc517880966)

[20、 线程同步相关方法 44](#_Toc517880967)

[21、 线程池 44](#_Toc517880968)

[简单设计模式 45](#_Toc517880969)

[1、 单例 45](#_Toc517880970)

[2、 简单工厂（Simple Factory） 45](#_Toc517880971)

[3、 工厂方法（Factory Method） 46](#_Toc517880972)

[4、 抽象工厂（Abstract Factory） 46](#_Toc517880973)

[5、 生成器 47](#_Toc517880974)

[6、 迭代器（Iterator） 47](#_Toc517880975)

[7、 装饰者 47](#_Toc517880976)

[Hibernate相关 48](#_Toc517880977)

[1、 Hibernate的原理体系架构 48](#_Toc517880978)

[2、 五大核心接口， 49](#_Toc517880979)

[3、 Hibernate对象的三种状态转换 50](#_Toc517880980)

[4、 Hibernate对一二级缓存的使用 50](#_Toc517880981)

[5、 Lazy-Load的理解 51](#_Toc517880982)

[6、 谈谈Hibernate中inverse的作用 52](#_Toc517880983)

[7、 说说Hibernate中的update()和saveOrUpdate()的区别，session的load()和get()的区别。 52](#_Toc517880984)

[8、 如何优化hibernate 52](#_Toc517880985)

[9、 为什么使用Hibernate？ 53](#_Toc517880986)

[Struts2相关 54](#_Toc517880987)

[1、 Struts工作流程: 54](#_Toc517880988)

[2、 描述 Struts 体系结构？对应各个部分的开发工作主要包括哪些？ 55](#_Toc517880989)

[3、 什么是Struts2 56](#_Toc517880990)

[4、 struts2 与struts1的区别 56](#_Toc517880991)

[5、 为什么要使用 Struts2 & Struts2 的优点： 57](#_Toc517880992)

[6、 Filter,Listener,Servlet区别 57](#_Toc517880993)

[7、 Struts2拦截器和过滤器的区别 58](#_Toc517880994)

[8、 Struts的Action是不是线程安全的 58](#_Toc517880995)

[9、 说出 struts2 中至少 5 个的默认拦截器 59](#_Toc517880996)

[10、 每个拦截器都是需要实现 Interceptor 接口 59](#_Toc517880997)

[11、 值栈ValueStack的原理与生命周期？ 59](#_Toc517880998)

[12、 ActionContext、ServletContext、pageContext的区别？ 60](#_Toc517880999)

[Spring相关 61](#_Toc517881000)

[1、 什么是spring 61](#_Toc517881001)

[2、 Spring有哪些优点 61](#_Toc517881002)

[3、 Spring框架有哪些模块 62](#_Toc517881003)

[4、 什么是控制反转(IOC)？什么是依赖注入？ 63](#_Toc517881004)

[5、 BeanFactory和ApplicationContext有什么区别？ 63](#_Toc517881005)

[6、 Spring有几种配置方式？ 63](#_Toc517881006)

[7、 Spring Bean的生命周期 64](#_Toc517881007)

[8、 Spring Bean的作用域之间有什么区别 65](#_Toc517881008)

[9、 Spring 框架中都用到了哪些设计模式 66](#_Toc517881009)

[10、 AOP（Aspect-Oriented Programming）是怎么实现的 66](#_Toc517881010)

[11、 介绍spring的IOC实现 67](#_Toc517881011)

[12、 spring中bean加载机制，bean生成的具体步骤 68](#_Toc517881012)

[13、 AOP相关概念 69](#_Toc517881013)

[14、 其他知识点 70](#_Toc517881014)

## JAVA基础

### 1、Java 和 C++的区别

1. **多重继承**(java接口多重,类不支持,C++支持)
2. **自动内存管理**
3. **预处理功能**
4. **goto语句**(java不支持)
5. **引用与指针**。在Java中不可能直接操作对象本身，所有的对象都由一个引用指向，必须通过这个引用才能访问对象本身，包括获取成员变量的值，改变对象的成员变量，调用对象的方法等。而在C++中存在引用，对象和指针三个东西，这三个东西都可以访问对象。其实，Java中的引用和C++中的指针在概念上是相似的，他们都是存放的对象在内存中的地址值，只是在Java中，引用丧失了部分灵活性，比如Java中的引用不能像C++中的指针那样进行**加减运算**。

### 2、值传递和引用传递

变量被值传递，意味着传递了**变量的一个副本**。因此，就算是改变了变量副本，也不会影响源对象的值。

对象被引用传递，意味着传递的并不是实际的对象，而是**对象的引用**。因此，外部对引用对象所做的改变会反映到所有的对象上。 java本质上还是值传递，如方法调用的时候传入一个对象引用进去，在方法栈中会构建一个副本和该引用变量值相同指向同一个地址。如果改变引用的值不会对改变传入的引用的值。

### 3、静态变量和实例变量的区别

**在语法定义上的区别**：静态变量前要加 static 关键字，而实例变量前则不加。

**在程序运行时的区别**：实例变量属于某个对象的属性，必须创建了实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量。静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。总之，实例变量必须创建对象后才可以通过这个对象来使用，静态变量则可以直接使用类名来引用。

### 4、JDK, JRE 和 JVM 的区别

**1）Java 开发工具包 (JDK)**

Java 开发工具包是 Java 环境的**核心组件，并提供编译、调试和运行一个 Java 程序所需的所有工具，可执行文件和二进制文件**。包括java基础jar包、虚拟机、javac等可执行文件等。JDK 是一个平台特定的软件，有针对 Windows，Mac 和 Unix 系统的不同的安装包。可以说 JDK 是 JRE 的超集，它包含了 JRE 的 Java 编译器，调试器和核心类。

**2）Java 虚拟机(JVM)**

JVM 是 Java 编程语言的核心。当我们运行一个程序时，JVM 负责将字节码转换为特定机器代码。JVM 也是平台特定的，并提供核心的 Java 方法，例如内存管理、垃圾回收和安全机制等。JVM 是可定制化的，我们可以通过 Java 选项(java options)定制它，比如配置 JVM 内存的上下界。JVM 之所以被称为虚拟的是因为它提供了一个不依赖于底层操作系统和机器硬件的接口。这种独立于硬件和操作系统的特性正是 Java 程序可以一次编写多处执行的原因。

**3）Java 运行时环境(JRE)**

JRE 是 JVM 的实施实现，它提供了运行 Java 程序的平台。**JRE 包含了 JVM、Java 二进制文件和其它成功执行程序的类文件**。JRE 不包含任何像 Java 编译器、调试器之类的开发工具。如果你只是想要执行 Java 程序，你只需安装 JRE 即可，没有安装 JDK 的必要。

**JDK, JRE 和 JVM 的区别**

1. JDK 是用于开发的而 JRE 是用于运行 Java 程序的。
2. JDK 和 JRE 都包含了 JVM，从而使得我们可以运行 Java 程序。
3. JVM 是 Java 编程语言的核心并且具有平台独立性。

### 5、Java 中的 final关键字用法

(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被覆盖；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

### 6、final, finally, finalize 的区别

**final**：修饰符（关键字）有三种用法：如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承。将变量声明为 final，可以保证它们在使用中不被改变，被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为 final 的方法也同样只能使用，不能在子类中被重写。

**finally**：通常放在 try…catch 的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要 JVM 不关闭都能执行，可以将释放外部资源的代码写在 finally 块中。

**finalize**：Object 类中定义的方法，Java 中允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

### 7、String 和StringBuilder、StringBuffer 的区别

Java 平台提供了两种类型的字符串：String 和StringBuffer / StringBuilder，它们可以储存和操作字符串。其中 String 是**只读**字符串，也就意味着 String 引用的字符串内容是不能被改变的。

而 StringBuffer 和 StringBuilder 类表示的字符串对象可以直接进行修改。StringBuilder 是 JDK 1.5 中引入的，它和 StringBuffer 的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被 synchronized 修饰，因此它的效率也比 StringBuffer 略高。

### 8、为什么String要设计成不可变的

1. 字符串常量池的需要. 字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，那么String interning将不能实现(译者注：String interning是指对不同的字符串仅仅只保存一个，即不会保存多个相同的字符串。)，因为这样的话，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。
2. 线程安全考虑。 同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。
3. 类加载器要用到字符串，不可变性提供了安全性，以便正确的类被加载。譬如你想加载java.sql.Connection类，而这个值被改成了myhacked.Connection，那么会对你的数据库造成不可知的破坏。
4. 支持hash映射和缓存。 因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

### 9、错误和异常的区别(Error vs Exception)

java.lang.Error: Throwable 的子类，**用于标记严重错误,表示系统级的错误和程序不必处理的异常**。合理的应用程序不应该去 try/catch 这种错误。是恢复不是不可能但很困难的情况下的一种严重问题；**比如内存溢出**，不可能指望程序能处理这样的情况；

java.lang.Exception: Throwable 的子类，**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。并且鼓励用户程序去 catch 它。

Error 和 RuntimeException 及其子类都是**未检查的异常（unchecked exceptions）**，而所有其他的 Exception 类都是检查了的异常（checked exceptions）

**checked exceptions**: **上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发．通常是从一个可以恢复的程序中抛出来的，并且最好能够从这种异常中使用程序恢复**。比如 **FileNotFoundException**, ParseException 等。检查了的异常发生在编译阶段，必须要使用 try…catch（或者 throws ）否则编译不通过。

**unchecked exceptions**:通常是如果一切正常的话本不该发生的异常，但是的确发生了。 **发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的**。比如 ArrayIndexOutOfBoundException, ClassCastException 等。从语言本身的角度讲，程序不该去 catch 这类异常，虽然能够从诸如 RuntimeException 这样的异常中 catch 并恢复，但是并不鼓励终端程序员这么做，因为完全没要必要。因为这类错误本身就是 bug，应该被修复，出现此类错误时程序就应该立即停止执行。 因此，面对 Errors 和 unchecked exceptions 应该让程序自动终止执行，程序员不该做诸如 try/catch 这样的事情，而是应该查明原因，修改代码逻辑。

### 10、java值传递问题

1. 对象就是传引用
2. 原始类型就是传值
3. String，Integer, Double等immutable类型因为没有提供自身修改的函数，每次操作都是新生成一个对象，所以要特殊对待。可以认为是传值。

Integer 和 String 一样。保存value的类变量是Final属性，无法被修改，只能被重新赋值／生成新的对象。 当Integer 做为方法参数传递进方法内时，对其的赋值都会导致 原Integer 的引用被 指向了方法内的栈地址，失去了对原类变量地址的指向。对赋值后的Integer对象做得任何操作，都不会影响原来对象。

### 11、变量赋值顺序

父类（静态变量、静态语句块）

子类（静态变量、静态语句块）

父类（实例变量、普通语句块）

父类（构造函数）

子类（实例变量、普通语句块）

子类（构造函数）

### 12、 深拷贝与浅拷贝

浅拷贝：拷贝实例和原始实例的引用类型引用同一个对象；

深拷贝：拷贝实例和原始实例的引用类型引用不同对象。

### 13、访问权限

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 自己 | 包 | 子类 | 全部 |
| Public | √ | √ | √ | √ |
| Protected | √ | √ | √ |  |
| defult | √ | √ |  |  |
| private | √ |  |  |  |

### 14、super

访问父类的构造函数：可以使用 super() 函数访问父类的构造函数，从而完成一些初始化的工作。

访问父类的成员：如果子类覆盖了父类的中某个方法的实现，可以通过使用 super 关键字来引用父类的方法实现。

### 15、覆盖与重载

覆盖（Override）存在于继承体系中，指子类实现了一个与父类在方法声明上完全相同的一个方法；

重载（Overload）存在于同一个类中，指一个方法与已经存在的方法名称上相同，但是参数类型、个数、顺序至少有一个不同。应该注意的是，返回值不同，其它都相同不算是重载。

### 16、Java面向对象的三个特征与含义

继承：继承是从已有的类中派生出新的类，新的类能吸收已有类的数据属性和行为，并能扩展新的能力.

多态：多态（Polymorphism）按字面的意思就是“多种状态，即同一个实体同时具有多种形式"。一般表现形式是程序在运行的过程中，同一种类型在不同的条件下表现不同的结果。多态也称为动态绑定，一般是在运行时刻才能确定方法的具体执行对象，这个过程也称为动态委派。

封装：就是把对象的属性和行为（或服务）结合为一个独立的整体，并尽可能隐藏对象的内部实现细节。

### 17、接口和抽象类的区别

1. 接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。
2. 类可以实现很多个接口，但是只能继承一个抽象类
3. 类如果要实现一个接口，它必须要实现接口声明的所有方法。但是，类可以不实现抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。
4. 抽象类可以在不提供接口方法实现的情况下实现接口。
5. Java 接口中声明的变量默认都是 final 的。抽象类可以包含非 final 的变量。
6. Java 接口中的成员函数默认是 public 的。抽象类的成员函数可以是 private，protected 或者是 public 。
7. 接口是绝对抽象的，不可以被实例化。抽象类也不可以被实例化，但是，如果它包含 main 方法的话是可以被调用的

### 18、两个对象值相同(x.equals(y) == true)，但却可有不同的 hash code，这句话对不对？

不对，如果两个对象 x 和 y 满足 x.equals(y) == true，它们的哈希码（hash code）应当相同。 Java 对于 eqauls 方法和 hashCode 方法是这样规定的：

(1)如果两个对象相同（equals 方法返回 true ），那么它们的 hashCode 值一定要相同；

(2)如果两个对象的 hashCode 相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在 Set 集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。

(3)如果对象的equals方法的比较操作所用到的信息没有被修改,那么对同一个对象调用多次,hashCode方法都必须始终如一地返回同一个整数.

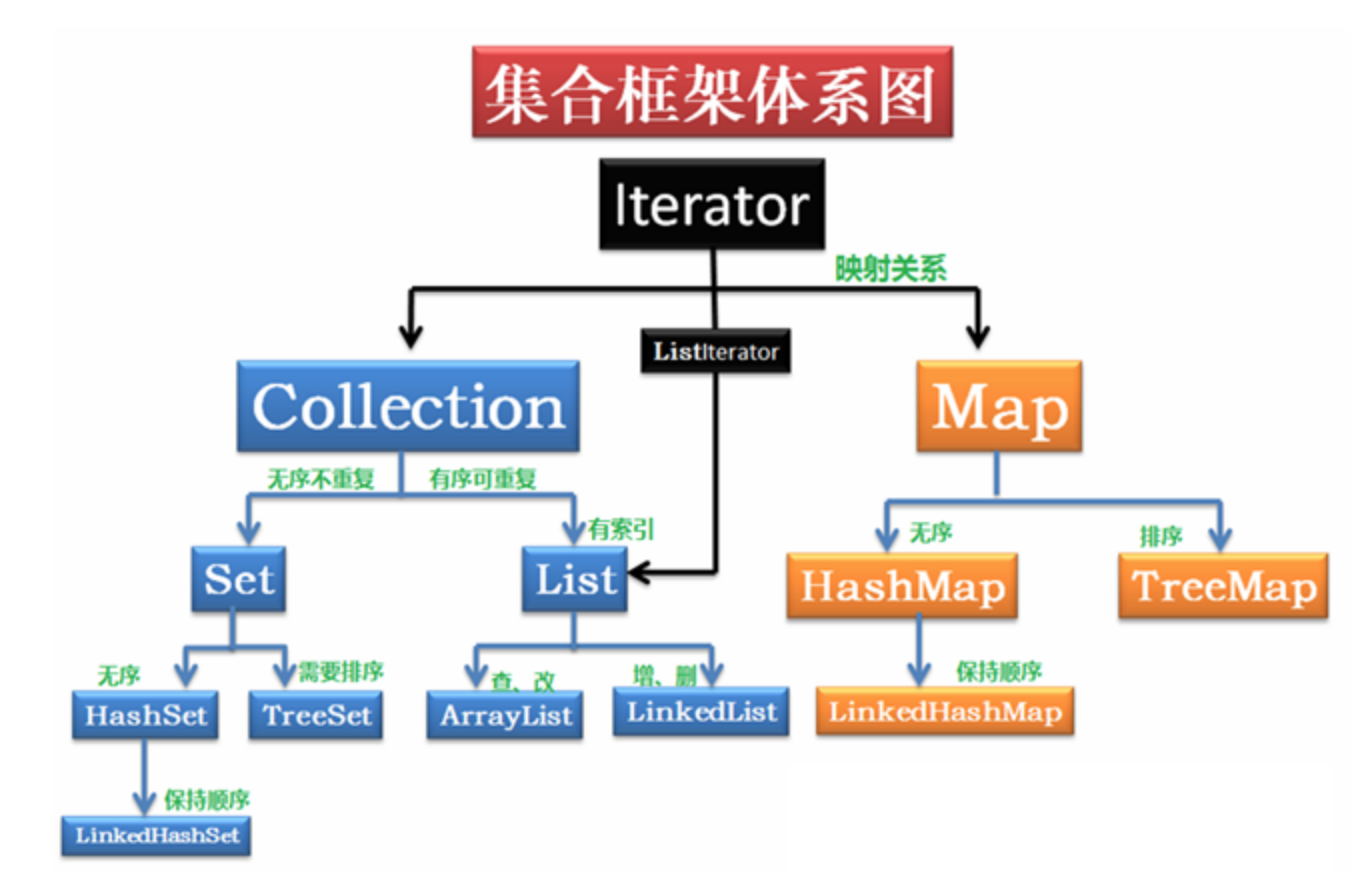
### 19、Java泛型

泛型，即“参数化类型”。顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

在编译之后程序会采取去泛型化的措施。也就是说Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

## JAVA集合







### 1、List、Map、Set 三个接口存储元素时各有什么特点

1）List是有序的Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。和下面要提到的Set不同，List允许有相同的元素。

除了具有Collection接口必备的iterator()方法外，List还提供一个listIterator()方法，返回一个ListIterator接口，和标准的Iterator接口相比，ListIterator多了一些add()之类的方法，允许添加，删除，设定元素，还能向前或向后遍历。实现List接口的常用类有LinkedList，ArrayList，Vector和Stack。

2）Set 是一种不包含重复的元素的 Collection，即任意的两个元素 e1 和 e2 都有e1.equals(e2)=false，Set 最多有一个 null 元素。

3）Map 接口 ：请注意，Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射．底层使用散列函数。

### 2、Map、Set、List、Queue、Stack的特点与用法

Set集合类似于一个罐子，"丢进"Set集合里的多个对象之间没有明显的顺序。

List集合代表元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。

Stack是Vector提供的一个子类，用于模拟"栈"这种数据结构(LIFO后进先出)

Queue用于模拟"队列"这种数据结构(先进先出 FIFO)。

Map用于保存具有\*\*"映射关系"\*\*的数据，因此Map集合里保存着两组值

### 3、Array和ArrayList有何区别？什么时候更适合用Array

Array可以容纳**基本类型和对象**，而**ArrayList只能容纳对象**。 Array是**指定大小的**，而ArrayList大小是固定的。

**Array没有提供ArrayList那么多功能**，比如addAll、removeAll和iterator等。尽管ArrayList明显是更好的选择，但也有些时候Array比较好用。如果列表的大小已经指定，大部分情况下是存储和遍历它们，则用**Array更好**。

### 4、ArrayList 和 LinkedList

ArrayList 和 LinkedList 都实现了 List 接口，他们有以下的不同点：

1. ArrayList 是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList 是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。
2. 相对于 ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。
3. LinkedList 比 ArrayList 更占内存，因为 LinkedList 为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

### 5、ArrayList和Vector异同点

ArrayList和Vector在很多时候都很类似：

（1）两者都是基于索引的，内部由一个**数组支持**。

（2）两者维护插入的顺序，我们可以**根据插入顺序来获取元素**。

（3）ArrayList和Vector的迭代器实现都是**fail-fast**的。

（4）ArrayList和Vector两者**允许null值**，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点：

（1）Vector是**同步**的，而ArrayList不是。然而，如果你寻求在迭代的时 候对列表进行改变，你应该使用CopyOnWriteArrayList。

（2）ArrayList比Vector**快**，它因为有同步，不会过载。

（3）ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

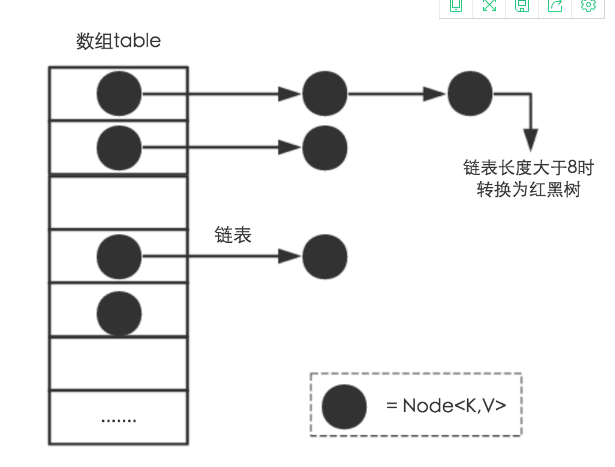
### 6、HashMap的工作原理

HashMap使用哈希算法，在put和get方法中，它使用hashCode()和equals()方法。当我们通过传递key-value对调用put方法的时候，HashMap使用Key hashCode()和哈希算法来找出存储key-value对的索引。Entry存储在LinkedList中，所以如果映射到同一个桶后，它使用equals()方法来检查传递的key是否entry已经存在，如果存在，它会覆盖value，如果不存在，它会创建一个新的entry然后保存。当我们通过传递key调用get方法时，它再次使用hashCode()来找到数组中的索引，然后使用equals()方法找出正确的Entry，然后返回它的值。

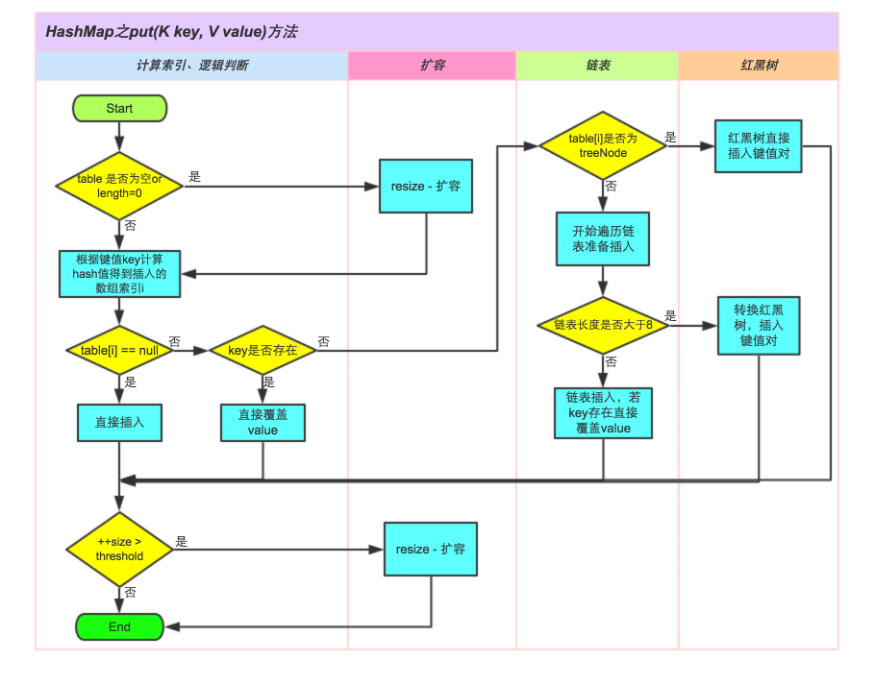
其它关于HashMap比较重要的问题**是容量、加载因子和阀值调整**。HashMap默认的初始容量是16，**负荷系数是0.75**。阀值是为负荷系数乘以容量，无论何时我们尝试添加一个entry，如果map的大小比阀值大的时候，HashMap会对map的内容进行重新哈希，且使用更大的容量。**容量总是2的幂**，所以如果你知道你需要存储大量的key-value对，比如缓存从数据库里面拉取的数据，使用正确的容量和负荷系数对HashMap进行初始化是个不错的做法。

### 7、HashMap实现原理

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下如所示。



1. 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。
2. HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。例如程序执行下面代码：map.put("美团","小美");系统将调用"美团"这个key的hashCode()方法得到其hashCode 值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。
3. HashMap初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。当元素数量超过threshold的时候开始扩容，扩容后数组的大小为原来的两倍。
4. 当进行put的时候，先根据hash值计算位于数组的位置，如果这里有值了就把这个值按照链表插入后面，这个链表超过8个就会将链表转换成红黑树。



**扩容机制：**方法是使用一个新的数组代替已有的容量小的数组，用transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。

### 8、如何决定选用HashMap还是TreeMap

对于在**Map中插入、删除和定位元素这类操作，HashMap是最好的选择**。然而，**假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择**。

### 9、HashMap和HashTable的区别

HashMap和Hashtable都实现了Map接口，因此很多特性非常相似。但是，他们有以下不同点：

1. HashMap允许键和值是null，而Hashtable不允许键或者值是null。
2. Hashtable是同步的，而HashMap不是。因此，HashMap更适合于单线程环境，而Hashtable适合于多线程环境。
3. HashMap的**迭代器(Iterator)是fail-fast迭代器**，而Hashtable的**enumerator(列举)迭代器不是fail-fast的**。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。一般认为Hashtable是一个遗留的类。如果你寻求在迭代的时候修改Map，你应该使用CocurrentHashMap

### 10、HashSet,TreeSet,LinkedHashSet 之间的区别

首先这三个类都是实现了Set接口，并且LinkedHashSet继承HashSet ，TreeSet实现了SortedSet。所以它们都拥有Set的基本特性，如：集合中的元素不能重复（这个需要和容器转载的类中的equals()方法有关）. 这三个容器所接受的类必须实现equals() 方法:

* HashSet： 为查询速度所设计的，存入HashSet的元素必须定义hashCode()方法。内部是基于散列函数实现。
* TreeSet： 实现了SortedSet（SortedSet的元素可以保证处于排序状态）,所以内部的元素是保持一定次序的（次序与元素实现的compareTo()方法有关），且底层是树状结构。使用它可以从Set中提取有序的序列。要求存入的元素必须实现Comparable接口,或则需要传入一个比较器Comparator。否则报ClassCastException。内部采用了红黑树实现。
* LinkedHashSet： 具有HashSet的查询速度，且内部链表维护元素的顺序（按插入顺序排序）。元素必须实现hashCode（）方法。内部是由链表实现。

### 11、HashMap ,LinkedHashMap ,TreeMap,WeakHashMap,ConcurrentHashMap,IdentityHashMap的区别

1. Map 中任何键的比较都是通过equals进行比较的，所以如果键用于映射的话需要由恰当的hashCode() 方法。如果键值用于TreeMap，它必须实现Comparable。
2. HashMap 基于散列表来的实现，即使用hashCode()进行快速查询元素的位置，显著提高性能。插入和查询“键值对”的开销是固定的。可以通过设置容量和负载因子，以调整容器的性能。
3. LinkedHashMap, 类似于HashMap,但是迭代遍历它时，取得“键值对”的顺序是其插入的次序，只比HashMap慢一点。而在迭代访问时反而更快，因为它使用链表维护内部次序。
4. TreeMap, 是基于红黑树的实现。实现了SortedMap，SortedMap 可以确保键处于排序状态。所以查看“键”和“键值对”时，所有得到的结果都是经过排序的，次序由Comparable或Comparator决定。SortedMap拥有其他额外的功能，如：Comparator comparator()返回当前Map使用的Comparator或者null. T firstKey() 返回Map的第一个键，T lastKey() 返回最后一个键。SortedMap headMap(toKey),生成一个键小于toKey的Map子集。SortedMap tailMap(fromKey) 也是生成一个子集。TreeMap是唯一的带有subMap()方法的Map,它可以返回一个子树
5. WeakHashMap 表示弱键映射，允许释放映射所指向的对象。这是为了解决某类特殊问题而设计的，如果映射之外没有引用指向某个“键”，则“键”可以被垃圾收集器回收。
6. ConcurrentHashMap 一种线程安全的Map,它不涉及同步加锁。
7. IdentityHashMap 使用==代替equals() 对“键”进行比较的散列映射。专为解决特殊问题而设计。
8. Hashtable (已经摒弃了)

### 12、hashCode()和equals()方法有何重要性

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。

### 13、Iterator是什么

Iterator接口提供遍历任何Collection的接口。我们可以从一个Collection中使用迭代器方法来获取迭代器实例。迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者在迭代过程中移除元素。

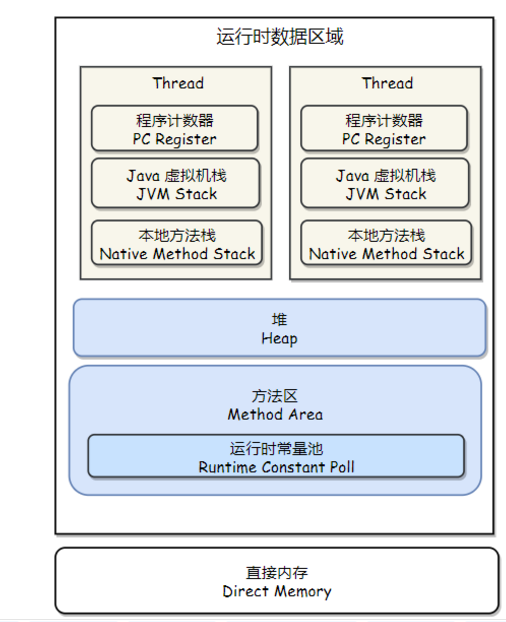
1. Enumeration 速度是 Iterator 的2倍，**同时占用更少的内存**。
2. Iterator远远比 Enumeration 安全，因为其他线程不能够修改正在被 iterator 遍历的集合里面的对象。
3. Iterator 允许调用者删除底层集合里面的元素，这对 Enumeration 来说是不可能的。

### 14、集合框架中的泛型有什么优点

Java1.5引入了泛型，所有的集合接口和实现都大量地使用它。泛型允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在编译时报错。这避免了在运行时出现ClassCastException，因为你将会在编译时得到报错信息。泛型也使得代码整洁，我们不需要使用显式转换和instanceOf操作符。它也给运行时带来好处，因为不会产生类型检查的字节码指令。

## JAVA虚拟机

### 介绍JVM中7个区域，然后把每个区域可能造成内存的溢出的情况说明



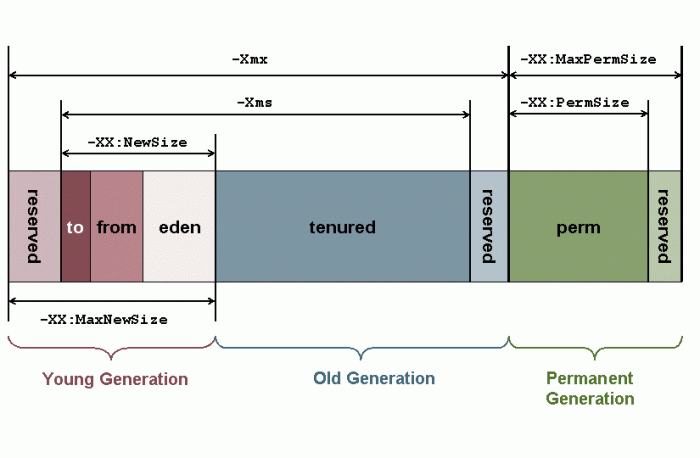
1. **程序计数器**：记录正在执行的虚拟机字节码指令的地址（如果正在执行的是本地方法则为空）。
2. **Java虚拟机栈**：每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
3. **本地方法栈**：与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。是线程**私有**的。
4. **Java堆**: 是所有线程共享的一块内存，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

是垃圾收集的主要区域（"GC 堆"），现代的垃圾收集器基本都是采用分代收集算法，该算法的思想是针对不同的对象采取不同的垃圾回收算法，因此虚拟机把 Java 堆分成以下三块：

* 新生代（Young Generation）
* 老年代（Old Generation）
* 永久代（Permanent Generation）

当一个对象被创建时，它首先进入新生代，之后有可能被转移到老年代中。新生代存放着大量的生命很短的对象，因此新生代在三个区域中垃圾回收的频率最高。为了更高效地进行垃圾回收，把新生代继续划分成以下三个空间：

* Eden
* Survivor 0
* Survivor 1



1. **方法区（永久代）**：用于存储已被虚拟机加载的**类信息、常量、静态变量、即时编译器编译后的代码**等数据。当方法区无法满足内存分配需求时，会抛出OutOfMemoryError。是共享内存。
2. **运行时常量池**：用于存放编译器生成的各种字面量和符号引用，是方法区的一部分。无法申请内存时抛出OutOfMemoryError。
3. **直接内存**：不是虚拟机运行时数据的一部分，也不是java虚拟机规范中定义的区域，是计算机直接的内存空间。这部分也被频繁使用，如JAVA NIO的引入基于通道和缓存区的I/O使用native函数直接分配堆外内存。如果内存不足会报OutOfMemoryError。

### GC的两种判定方法：引用计数与根搜索算法

1. **引用计数**： 给对象添加一个引用计数器，每当有一个地方引用该对象时，计数器值加1，当引用失效时，计数器值减1,。任何时候计数器都为0的对象就是不可能再被使用的。它很难解决对象之间相互**循环引用**问题。
2. **根搜索算法（GC Roots Traceing）:** 通过一系列名为“GC Roots”的对象作为起点，从这些节点开始向下搜索，搜索走过的路径成为**引用链**，当一个对象到GC Roots没有任何引用链相连时，则证明此对象不可用。

### Java中的四种引用

Java中提供这四种引用类型主要有两个目的：第一是可以让程序员通过代码的方式决定某些对象的生命周期；第二是有利于JVM进行垃圾回收。

1. **强引用**：程序代码中的普通引用。如Object obj = new Object(),只要强引用存在，垃圾回收器就不会回收。在不使用对象时应及时将引用设置为null，便于垃圾回收。
2. **软引用**：描述一些有用但并非必须的对象。对于软引用关联的对象在系统将要**发生内存溢出异常之前**，将会把这些对象列进回收范围之中进行第二次回收。**SoftRefence**
3. **弱引用**：描述非必须对象，比软引用弱一些。被弱引用关联的对象只能**生存到下一次垃圾收集发生之前**。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。**WeakRefence**
4. **虚引用**：最弱的引用，不管是否有虚引用存在，完全不会对对象生存时间构成影响，也无法通过虚引用来取得一个对象实例。唯一目的是希望能够在这个对象被垃圾回收器之前收到系统通知。**PhantomReference**

### 内存溢出和内存泄漏

**内存溢出**：通俗理解就是**内存不够**，程序所需要的内存远远超出了你虚拟机分配的内存大小，就叫内存溢出

**内存泄露**：内存泄漏也称作“存储渗漏”，用动态存储分配函数动态开辟的空间，在**使用完毕后未释放**，结果导致**一直占据该内存单元**。直到程序结束。（其实说白了就是该内存空间使用完毕之后未回收）即所谓内存泄漏

### 内存溢出了怎么办

通过内存映像工具如jhat、jconsole等对dump出来的堆转存储快照进行分析，重点是确认内存是出现内存泄露还是内存溢出。

如果是**内存泄露**进一步使用工具查看泄露的对象到GC Roots的引用链。于是就能找到泄露对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们。掌握泄露对象的信息，以及GC Roots引用链的信息，就可以比较准确定位泄露代码的位置。

如果不存在**内存泄露**，那就需要通过jinfo、Jconsole等工具分析java堆参数与机器物理内存对比是否还可以调大，从代码上检查是否存在某些对象生命周期过长，持有状态过长的情况，尝试减少程序的运行消耗。

### 什么时候会发生jvm堆（持久区）内存溢出

简单的来说 java的堆内存分为两块:permantspace（持久代） 和 heap space。

持久带中主要存放用于存放静态类型数据，如 Java Class, Method 等， 与垃圾收集器要收集的Java对象关系不大。

而heapspace分为年轻代和年老代:

* 年轻代的垃圾回收叫 Young GC， 年老代的垃圾回收叫 Full GC。
* 在年轻代中经历了N次（可配置）垃圾回收后仍然存活的对象，就会被复制到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象
* 年老代溢出原因有 循环上万次的字符串处理、创建上千万个对象、在一段代码内申请上百M甚至上G的内存

### Java 中有内存泄露吗？

有，Java中，造成内存泄露的原因有很多种。典型的例子是**长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露**，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景，通俗地说，就是程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收的，这就是java中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对象放在缓存中(例如放在一个全局map对象中)，然后一直不再使用它，这个对象一直被缓存引用，但却不再被使用。

**如果一个外部类的实例对象的方法返回了一个内部类的实例对象**，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持久外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

### GC 是什么？为什么要有 GC

GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但 JVM 可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

### GC的收集方法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？

1. **标记清理**：首先标记所有需要回收的对象，在标记完成后**统一回收掉**所有被标记的对象，它的标记的对象。缺点是**效率低**，且存在**内存碎片**。主要用于老生代垃圾回收。
2. **标记整理**：首先标记所有需要回收的对象，在标记完成后让所有存活的对象都向一端移动，然后直接清理掉端边界意外的内存。用于老年代。
3. **复制算法**：将内存按容量划分为大小相等的一块，每次只用其中一块。当内存用完了，将还存活的对象复制到另一块内存，然后把已使用过的内存空间一次清理掉。实现简单，高效。一般用于新生代。一般是将内存分为一块较大的**Eden空间**和两块较小的**Survivor**空间。HotSpot虚拟机默认比例是**8:1**,。每次使用Eden和一块Survivor，当回收时将这两块内存中还存活的对象复制到Survivor然后清理掉刚才Eden和Survivor的空间。如果复制过程内存不够使用则向老年代分配担保。
4. **分代收集算法**：根据对象的生存周期将内存划分为新生代和老年代，根据年代的特点采用最适当的收集算法。

### GC收集器有哪些？CMS收集器与G1收集器的特点。

1. **Serial**: 单线程收集器，只会使用一个CPU或一条收集器线程去完成，垃圾回收工作，更重要的是在进行垃圾回收时，必须暂停其他所有的工作线程。（Stop the world）。简单高效，用于新生代。
2. **ParNew**: 是Serial收集器的**多线程版本**，垃圾回收时采用多线程方式进行回收。默认情况下使用的线程数是cpu数量。除了serial收集器，目前只有它能和CMS收集器配合工作。是server模式下首选的新生代收集器。
3. **Parallel Scavenge**: 使用**复制算法**收集器，也是一个并行的多线程收集器。Parallel Scavenge收集器与其他收集器关注点不同，其它收集器主要关注缩短垃圾回收时用户线程的停顿时间。而它关心**吞吐量**，即**运行用户代码时间/(运行用户代码时间 + 垃圾收集时间)**。停顿时间越短越适合需要与用户交互的程序，高吞吐量则可以最高效率的利用CPU时间。
4. **Serial Old**: 老年代，单线程收集器，使用**标记整理算法**。主要有两个用途，一是和Parallel Scavenge 收集器配合使用，二是作为CMS的后备方案在并发收集器发生**Concurrent Mode Failure**时候使用。
5. **Parallel Old**:并行的老年代版本收集器，使用标记整理算法。主要与Parallel Scavenge配合使用。
6. **CMS**：是以获得**最短回收停顿时间为**目标的收集器，使用**标记清除算法**。整个过程包括4个：
7. **初始标记**: 标记Gc ROOTS能直接关联到的对象
   1. **并发标记**：进行Roots Traceing的过程
   2. **重新标记**：修正并发标记期间因用户继续工作导致标记产生变动
   3. **并发清除**：并发清除数据。 初始标记和重新标记需要stop the world. 并发标记和并发清除过程用户线程和收集器线程可以并行执行。
8. **G1(Garbage First):** 基于**标记-整理算法**的收集器,不会产生空间碎片.它可以精确控制停顿,能够让使用者明确指定一个长度为M毫秒的时间片段内,消耗集上的时间不超过N秒.是不牺牲吞吐量的前提下完成低停顿的.**G1将整个java堆(新生和老生)划分为大小相同的区,并跟踪这些区上发生的变化.在后台维护一个优先列表,每次根据允许的收集时间优先回收垃圾最多的区域**.

### Minor GC与Full GC分别在什么时候发生？

FullGC 一般是发生在老年代的GC，出现一个FullGC经常会伴随至少一次的Minor GC。速度比MinorGC慢10倍以上。

MinorGC 指发生在新生代的垃圾收集动作，非常频繁，回收速度也快。一般发生在新生代空间不足时,另外一个FullGC经常会伴随至少一次的Minor GC. 当虚拟检测晋升到到老年代的平均大小是否小于老年代剩余空间大小,如果小于并且允许担保失败,则执行Minor GC.

### 内存分配策略

（一）对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC。

（二）大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。

经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

（三）长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

（四）动态对象年龄判定

虚拟机并不是永远地要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 区中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

（五）空间分配担保

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的；如果不成立的话虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC，尽管这次 Minor GC 是有风险的；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那这时也要改为进行一次 Full GC。

## JAVA类的加载机制

### 类的生命周期

包括以下 7 个阶段：

**加载（Loading）:**

加载是类加载的一个阶段，注意不要混淆。

加载过程完成以下三件事：

* 通过一个类的全限定名来获取定义此类的二进制字节流。
* 将这个字节流所代表的静态存储结构转化为方法区的运行时存储结构。
* 在内存中生成一个代表这个类的 Class 对象，作为方法区这个类的各种数据的访问入口。

其中二进制字节流可以从以下方式中获取：

* 从 ZIP 包读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础。
* 从网络中获取，这种场景最典型的应用是 Applet。
* 运行时计算生成，这种场景使用得最多得就是动态代理技术，在 java.lang.reflect.Proxy 中，就是用了 ProxyGenerator.generateProxyClass 的代理类的二进制字节流。
* 由其他文件生成，典型场景是 JSP 应用，即由 JSP 文件生成对应的 Class 类。
* 从数据库读取，这种场景相对少见，例如有些中间件服务器（如 SAP Netweaver）可以选择把程序安装到数据库中来完成程序代码在集群间的分发。 ...

**验证（Verification）:**

确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

* 文件格式验证：验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。
* 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求。
* 字节码验证：通过数据流和控制流分析，确保程序语义是合法、符合逻辑的。
* 符号引用验证：发生在虚拟机将符号引用转换为直接引用的时候，对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验。

**准备（Preparation）:**

类变量是被 static 修饰的变量，准备阶段为类变量分配内存并设置初始值，使用的是方法区的内存。

实例变量不会在这阶段分配内存，它将会在对象实例化时随着对象一起分配在 Java 堆中。（实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次）

初始值一般为 0 值，例如下面的类变量 value 被初始化为 0 而不是 123。

public static int value = 123;

如果类变量是常量，那么会按照表达式来进行初始化，而不是赋值为 0。

public static final int value = 123;

**解析（Resolution）:**

将常量池的符号引用替换为直接引用的过程。

**初始化（Initialization）:**

虚拟机规范中并没有强制约束何时进行加载，但是规范严格规定了有且只有下列五种情况必须对类进行初始化（加载、验证、准备都会随着发生）：

* 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。最常见的生成这 4 条指令的场景是：使用 new 关键字实例化对象的时候；读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候；以及调用一个类的静态方法的时候。
* 使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行初始化，则需要先触发其初始化。
* 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
* 当虚拟机启动时，用户需要指定一个要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类；
* 当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为 REF\_getStatic, REF\_putStatic, REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化；

初始化阶段才真正开始执行类中的定义的 Java 程序代码。初始化阶段即虚拟机执行类构造器 <clinit>() 方法的过程。

在准备阶段，类变量已经赋过一次系统要求的初始值，而在初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其它资源。

<clinit>() 方法具有以下特点：

* 是由编译器自动收集类中所有类变量的赋值动作和静态语句块（static{} 块）中的语句合并产生的，编译器收集的顺序由语句在源文件中出现的顺序决定。特别注意的是，静态语句块只能访问到定义在它之前的类变量，定义在它之后的类变量只能赋值，不能访问。例如以下代码：
* 与类的构造函数（或者说实例构造器 <init>()）不同，不需要显式的调用父类的构造器。虚拟机会自动保证在子类的 <clinit>() 方法运行之前，父类的 <clinit>() 方法已经执行结束。因此虚拟机中第一个执行 <clinit>() 方法的类肯定为 java.lang.Object。
* 由于父类的 <clinit>() 方法先执行，也就意味着父类中定义的静态语句块要优于子类的变量赋值操作
* <clinit>() 方法对于类或接口不是必须的，如果一个类中不包含静态语句块，也没有对类变量的赋值操作，编译器可以不为该类生成 <clinit>() 方法。
* 接口中不可以使用静态语句块，但仍然有类变量初始化的赋值操作，因此接口与类一样都会生成 <clinit>() 方法。但接口与类不同的是，执行接口的 <clinit>() 方法不需要先执行父接口的 <clinit>() 方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的 <clinit>() 方法。
* 虚拟机会保证一个类的 <clinit>() 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 <clinit>() 方法，其它线程都会阻塞等待，直到活动线程执行 <clinit>() 方法完毕。如果在一个类的 <clinit>() 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。

使用（Using）

卸载（Unloading）

### JVM 加载 class 文件的原理机制

JVM 中类的装载是由类加载器（ClassLoader） 和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。

由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类已经被加载、连接(验证、准备和解析)和初始化。类的加载是指把类的 .class 文件中的数据读入到内存中，通常是创建一个字节数组读入 .class 文件，然后产生与所加载类对应的 Class 对象。加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备(为静态变量分配内存并设置默认的初始值)和解析(将符号引用替换为直接引用)三个步骤。最后 JVM 对类进行初始化，包括：

1. 如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；
2. 如果类中存在初始化语句，就依次执行这些初始化语句。

类的加载是由类加载器完成的，类加载器包括：**启动类加载器（BootStrap）、扩展加载器（Extension）、应用程序加载器（Application）和用户自定义类加载器（java.lang.ClassLoader的子类）**。从JDK 1.2开始，类加载过程采取了父亲委托机制(PDM)。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类加载器的说明：

* **Bootstrap**：启动类加载器，一般用本地代码实现，负责加载JVM基础核心类库。加载存放在<JAVA\_HOME>/lib目录中的类库（如rt.jar）；
* **Extension ClassLoader**：扩展加载器， 负责加载<JAVA\_HOME>/lib/ext目录中的 ，或被java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；
* **Application ClassLoader**：应用程序加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

缺点:

* 双亲委派模型很好地解决了各个类加载器的基础类统一问题(越基础的类由越上层的加载器进行加载)，基础类之所以被称为“基础”，是因为它们总是作为被调用代码调用的API。但是，如果基础类又要调用用户的代码时，双亲委派模型无法满足要求。 因为Bootstrap加载器无法找到永不代码类。

为了解决这个困境，Java设计团队只好引入了一个不太优雅的设计：**线程上下文件类加载器(Thread Context ClassLoader)**。这个类加载器可以通过java.lang.Thread类的setContextClassLoader()方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个；如果在应用程序的全局范围内都没有设置过，那么这个类加载器默认就是应用程序类加载器。了有线程上下文类加载器，JNDI服务使用这个线程上下文类加载器去加载所需要的SPI代码，也就是父类加载器请求子类加载器去完成类加载动作，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型，但这也是无可奈何的事情。**Java中所有涉及SPI的加载动作基本上都采用这种方式**，

### 双亲委派模型

除了顶层的启动类加载器外,其余的类加载器都应当有自己的父类加载器.顺序依次是:

* Bootstrap ClassLoader: 启动类加载器,加载java\_home/lib中的类
* Extension ClassLoader: 扩展类加载器,加载java\_home/lib/ext目录下的类库
* Application ClassLoader: 应用程序类加载器,加载用户类路径上指定类库.

双亲委派模型的工作原理是:如果一个类加载器受到了类加载请求,它首先不会自己去尝试加载这个类,而把这个请求委派给父类加载器去完成,每一层次的类加载器都是如此,因此所有的加载请求最终都应该传送到顶层的启动类加载器中,只有当父类加载器反馈自己无法完成加载请求时,加载器才尝试自己加载.这种方式保证了Oject类(JDK 核心类)在各个加载器加载环境中都是同一个类.

## JAVA并发

### 线程状态转化



**新建**：创建后尚未启动。

**可运行：**可能正在运行，也可能正在等待 CPU 时间片。包含了操作系统线程状态中的 Running 和 Ready。

**阻塞：**等待获取一个排它锁，如果其线程释放了锁就会结束此状态。

**无限等待：**等待其它线程显式地唤醒，否则不会被分配 CPU 时间片。

**有限等待：**无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒。

调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“使一个线程睡眠”进行描述。

调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“挂起一个线程”进行描述。

睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

阻塞和等待的区别在于，阻塞是被动的，它是在等待获取一个排它锁。而等待是主动的，通过调用 Thread.sleep() 和 Object.wait() 等方法进入。

**死亡：**可以是线程结束任务之后自己结束，或者产生了异常而结束。

### 使用线程

三种方法：实现Runnable接口、实现Callable接口、继承Thread类。

实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。可以说任务是通过线程驱动从而执行的。

### 基础线程机制

**Executor：**Executor 管理多个异步任务的执行，而无需程序员显式地管理线程的生命周期。这里的异步是指多个任务的执行互不干扰，不需要进行同步操作。

**Daemon：**守护线程是程序运行时在后台提供服务的线程，不属于程序中不可或缺的部分。

当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程。

main() 属于非守护线程。

使用 setDaemon() 方法将一个线程设置为守护线程。

**sleep()：**Thread.sleep(millisec) 方法会休眠当前正在执行的线程，millisec 单位为毫秒。

**yield()：**对静态方法 Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行。该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行。

### 中断

**InterruptedException：**通过调用一个线程的 interrupt() 来中断该线程，如果该线程处于阻塞、限期等待或者无限期等待状态，那么就会抛出 InterruptedException，从而提前结束该线程。但是不能中断 I/O 阻塞和 synchronized 锁阻塞。

**interrupted()：**如果一个线程的 run() 方法执行一个无限循环，并且没有执行 sleep() 等会抛出 InterruptedException 的操作，那么调用线程的 interrupt() 方法就无法使线程提前结束。但是调用 interrupt() 方法会设置线程的中断标记，此时调用 interrupted() 方法会返回 true。因此可以在循环体中使用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程。

**Executor 的中断操作:**调用 Executor 的 shutdown() 方法会等待线程都执行完毕之后再关闭，但是如果调用的是 shutdownNow() 方法，则相当于调用每个线程的 interrupt() 方法。如果只想中断 Executor 中的一个线程，可以通过使用 submit() 方法来提交一个线程，它会返回一个 Future<?> 对象，通过调用该对象的 cancel(true) 方法就可以中断线程。

### 互斥同步

Java 提供了两种锁机制来控制多个线程对共享资源的互斥访问，第一个是 JVM 实现的 synchronized，而另一个是 JDK 实现的 ReentrantLock。

**Synchronized：**

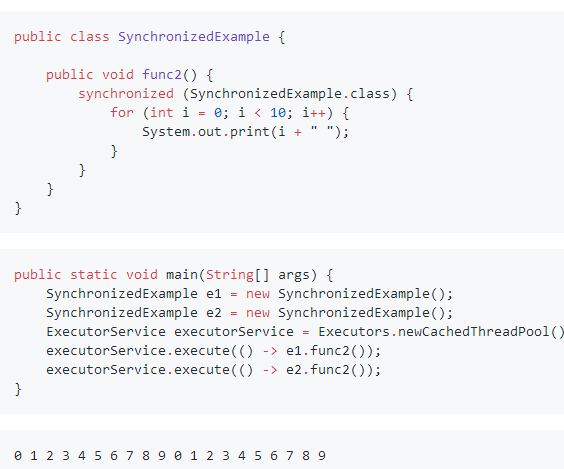
**1、同步一个代码块：**它只作用于同一个对象，如果调用两个对象上的同步代码块，就不会进行同步。对于以下代码，使用 ExecutorService 执行了两个线程，由于调用的是同一个对象的同步代码块，因此这两个线程会进行同步，当一个线程进入同步语句块时，另一个线程就必须等待。



如果新建了连个对象e1，e2运行之后应该是0.0.1.1.2.2.3.3……

**2、同步一个方法：**与同步代码块一样。

**3、同步一个类：**作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步。



4、**同步一个静态方法：**作用于整个类。

**ReentrantLock：**

相比于synchronized来说它多了以下高级功能：等待可以中断，可实现公平锁，绑定多个条件。

### synchronized 和 ReentrantLock 比较

**1. 锁的实现**

synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的。

**2. 性能**

新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等。目前来看ReentrantLock 的性能基本持平了，因此性能因素不再是选择 ReentrantLock 的理由。synchronized 更大的性能优化空间，应该优先考虑 synchronized。

**3. 功能**

ReentrantLock 多了一些高级功能。

**4. 使用选择**

除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized。这是因为 synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持。并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放。

### 线程之间的协作

当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调。

**join()：**线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。

**wait() notify() notifyAll()：**

调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程会调用 notify() 或者 notifyAll() 来唤醒挂起的线程。

它们都属于 Object 的一部分，而不属于 Thread。

只能用在同步方法或者同步控制块中使用，否则会在运行时抛出 IllegalMonitorStateExeception。

使用 wait() 挂起期间，线程会释放锁。这是因为，如果没有释放锁，那么其它线程就无法进入对象的同步方法或者同步控制块中，那么就无法执行 notify() 或者 notifyAll() 来唤醒挂起的线程，造成死锁。

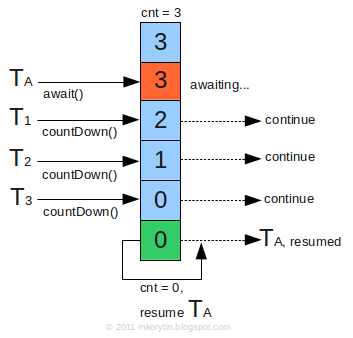
**await() signal() signalAll()：**

java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程。相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活。

### 实现线程的类

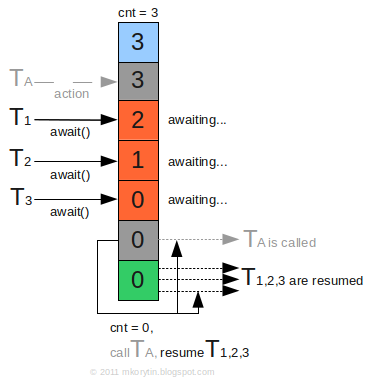
如果有可能，最好不要自己实现并发，多利用现成的线程类。

**CountdownLatch：**用来控制一个线程等待多个线程。维护了一个计数器 cnt，每次调用 countDown() 方法会让计数器的值减 1，减到 0 的时候，那些因为调用 await() 方法而在等待的线程就会被唤醒。



**CyclicBarrier：**用来控制多个线程互相等待，只有当多个线程都到达时，这些线程才会继续执行。和 CountdownLatch 相似，都是通过维护计数器来实现的。但是它的计数器是递增的，每次执行 await() 方法之后，计数器会加 1，直到计数器的值和设置的值相等，等待的所有线程才会继续执行。和 CountdownLatch 的另一个区别是，CyclicBarrier 的计数器可以循环使用，所以它才叫做循环屏障。

下图应该从下往上看才正确。



**BlockingQueue：**java.util.concurrent.BlockingQueue 接口有以下阻塞队列的实现：

* **FIFO 队列** ：LinkedBlockingQueue、ArrayBlockingQueue（固定长度）
* **优先级队列** ：PriorityBlockingQueue

提供了阻塞的 take() 和 put() 方法：如果队列为空 take() 将阻塞，直到队列中有内容；如果队列为满 put() 将阻塞，直到队列有空闲位置。

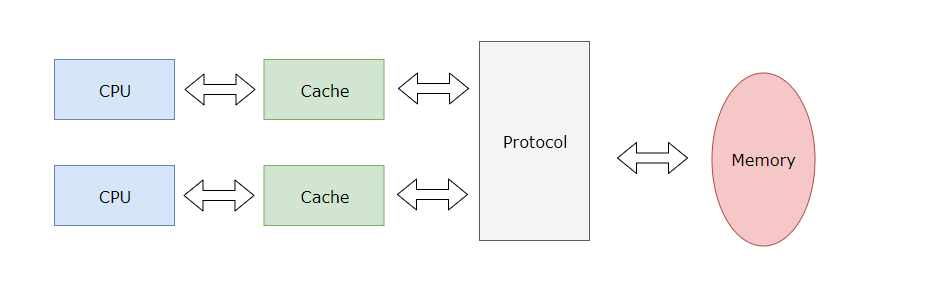
### 线程不安全

如果多个线程对同一个数据进行访问而不采取同步操作的话，那么操作的结果是无疑值得，这就是线程不安全。

### JAVA内存模型

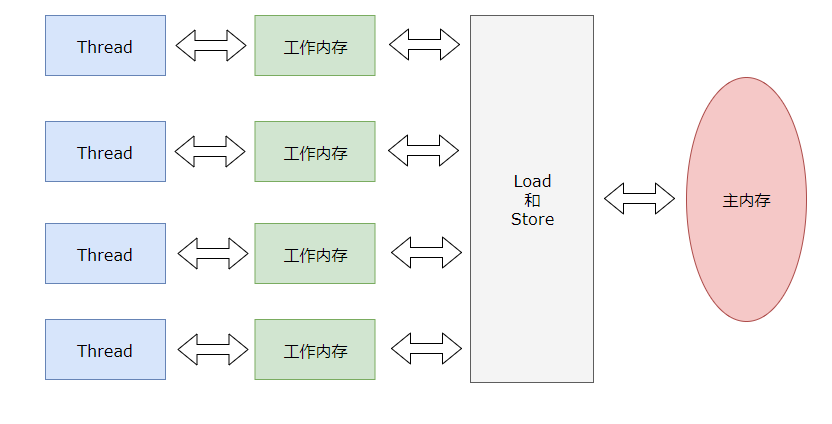
**主内存和工作内存**

处理器上的寄存器的读写的速度比内存快几个数量级，为了解决这种速度矛盾，在它们之间加入了高速缓存。加入高速缓存带来了一个新的问题：缓存一致性。如果多个缓存共享同一块主内存区域，那么多个缓存的数据可能会不一致，需要一些协议来解决这个问题。



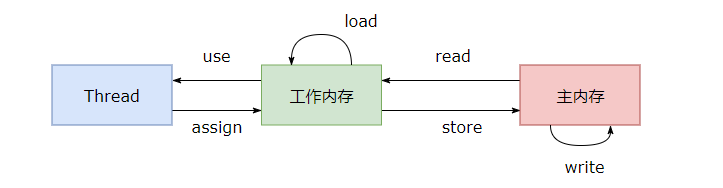
所有的变量都存储在主内存中，每个线程还有自己的工作内存，工作内存存储在高速缓存或者寄存器中，保存了该线程使用的变量的主内存副本拷贝。

线程只能直接操作工作内存中的变量，不同线程之间的变量值传递需要通过主内存来完成。



**内存间交互操作：**

Java 内存模型定义了 8 个操作来完成主内存和工作内存的交互操作。



read：把一个变量的值从主内存传输到工作内存中

load：在 read 之后执行，把 read 得到的值放入工作内存的变量副本中

use：把工作内存中一个变量的值传递给执行引擎

assign：把一个从执行引擎接收到的值赋给工作内存的变量

store：把工作内存的一个变量的值传送到主内存中

write：在 store 之后执行，把 store 得到的值放入主内存的变量中

lock：作用于主内存的变量

**内存模型的三大特性：**

**原子性**

**可见性：**可见性指当一个线程修改了共享变量的值，其它线程能够立即得知这个修改。Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值来实现可见性的。

volatile 可保证可见性。synchronized 也能够保证可见性，对一个变量执行 unlock 操作之前，必须把变量值同步回主内存。final 关键字也能保证可见性：被 final 关键字修饰的字段在构造器中一旦初始化完成，并且没有发生 this 逃逸（其它线程可以通过 this 引用访问到初始化了一半的对象），那么其它线程就能看见 final 字段的值。

**有序性：**有序性是指：在本线程内观察，所有操作都是有序的。在一个线程观察另一个线程，所有操作都是无序的，无序是因为发生了指令重排序。

在 Java 内存模型中，允许编译器和处理器对指令进行重排序，重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

volatile 关键字通过添加内存屏障的方式来禁止指令重排，即重排序时不能把后面的指令放到内存屏障之前。

也可以通过 synchronized 来保证有序性，它保证每个时刻只有一个线程执行同步代码，相当于是让线程顺序执行同步代码。

### 原子性

[原子](http://www.so.com/s?q=%E5%8E%9F%E5%AD%90&ie=utf-8&src=internal_wenda_recommend_textn)性就是说一个操作不可以被中途cpu暂停然后调度, 即不能被中断, 要不就执行完, 要不就不执行. 如果一个操作是原子性的, 那么在[多线程](http://www.so.com/s?q=%E5%A4%9A%E7%BA%BF%E7%A8%8B&ie=utf-8&src=internal_wenda_recommend_textn)环境下, 就不会出现[变量](http://www.so.com/s?q=%E5%8F%98%E9%87%8F&ie=utf-8&src=internal_wenda_recommend_textn)被修改等奇怪的问题.

### 原子性实现原理和实现机制

**实现原理：**使用循环CAS实现原子性操作，CAS是在操作期间先比较旧值，如果旧值没有发生改变，才交换成新值，发生了变化则不交换。

**实现机制：**处理器提供总线锁定和缓存锁定两种方式来保证复杂内存操作的原子性。

***总线型：***就是使用处理器提供一个LOCK信号，当一个处理器在总线传输信号时，其他处理器的请求将被阻塞住，那么该处理独占内存。所以总线锁定开销大。

***缓存锁定：***当某个CPU对内存中的数据进行执行之后，会通知其他CPU这个操作；其他CPU此时会放弃高速缓存中的内容，或者重新读取内存内容。

### 线程安全分类

分类共有5类：

不可变、绝对线程安全、相对线程安全、线程兼容、线程对立

**不可变：**的对象一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障措施，只要一个不可变的对象被正确地构建出来，那其外部的可见状态永远也不会改变，永远也不会看到它在多个线程之中处于不一致的状态。比如：String，final类等。

**绝对线程安全：**不管运行时环境如何，调用者都不需要任何额外的同步措施。

相对线程安全：相对的线程安全需要保证对这个对象单独的操作是线程安全的，在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。

在 Java 语言中，大部分的线程安全类都属于这种类型，例如 Vector、HashTable、Collections 的 synchronizedCollection() 方法包装的集合等。

**线程兼容：**线程兼容是指对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用，我们平常说一个类不是线程安全的，绝大多数时候指的是这一种情况。Java API 中大部分的类都是属于线程兼容的，如与前面的 Vector 和 HashTable 相对应的集合类 ArrayList 和 HashMap 等。

**线程对立：**线程对立是指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。由于 Java 语言天生就具备多线程特性，线程对立这种排斥多线程的代码是很少出现的，而且通常都是有害的，应当尽量避免。

### 线程安全的实现方法

**互斥同步：**synchronized 和 ReentrantLock。

**非阻塞同步：**其实也就是乐观锁，不进行互斥操作。互斥操作的问题主要是阻塞和唤醒带来的性能问题。互斥同步实际上是悲观锁，也就是说默认悲观的认为只要不进行同步措施（比如加锁），一定会出现问题。所以无论有没有竞争，都得加锁。问题就在于，虽然有可能有竞争，但是很多时候竞争不会发生。这就造成新的思路：非阻塞同步，也就是乐观锁。

基于冲突检测的乐观并发策略，通俗地说，就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施（最常见的补偿措施就是不断地重试，直到成功为止），这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步操作称为非阻塞同步（Non-Blocking Synchronization）。

**无同步方案：**要保证线程安全，并不是一定就要进行同步，两者没有因果关系。同步只是保证共享数据争用时的正确性的手段，如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性，因此会有一些代码天生就是线程安全的。

* **可重入代码：**这种代码也叫做纯代码（Pure Code），可以在代码执行的任何时刻中断它，转而去执行另外一段代码（包括递归调用它本身），而在控制权返回后，原来的程序不会出现任何错误。相对线程安全来说，可重入性是更基本的特性，它可以保证线程安全，即所有的可重入的代码都是线程安全的，但是并非所有的线程安全的代码都是可重入的。

可重入代码有一些共同的特征，例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等。我们可以通过一个简单的原则来判断代码是否具备可重入性：如果一个方法，它的返回结果是可以预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的。

* **栈封闭：**多个线程访问同一个方法的局部变量时，不会出现线程安全问题，因为局部变量存储在栈中，属于线程私有的。
* **线程本地存储：**如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行。如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题。

### 锁优化

**自旋锁：**如果一个线程在获取一个锁的时候失败了，这样该线程就会进入循环，不断请求这个锁。如果获取成功，就不必进入阻塞状态；如果失败，进入阻塞。

**可重入锁不可重入锁：**自己可以获得自己的锁叫做可重入锁；否则叫不可重入锁。

**偏向锁：**偏向锁的思想是偏向于让第一个获取锁对象的线程，这个线程在之后获取该锁就不再需要进行同步操作，甚至连 CAS 操作也不再需要。

**轻量级锁重量级锁**：JDK 1.6 引入了偏向锁和轻量级锁，从而让锁拥有了四个状态：无锁状态（unlocked）、偏向锁状态（biasble）、轻量级锁状态（lightweight locked）和重量级锁状态（inflated）。

当一个互斥资源受到一个线程访问的时候，该资源给该线程加偏向锁；这时候又有线程访问该资源，则该线程进入自旋等待，该互斥资源加轻量级锁；如果自旋等待的线程一段时间没有获得该资源的锁，则该线程挂起，该资源加重量级锁，下一次再有线程想获得该资源的锁的时候不在自旋等待，直接挂起。

### Volatile的理解

**Volatile自身特性**：

1. Volatile 是轻量级的synchronized，它在多处理器开发过程中保证了共享变量的“**可见性**”，可见性是指当一个线程的某个共享变量发生改变时，另一个线程能够读取到这个修改的值。Voaltile变量修饰的变量在进行写操作时在多核处理器下首先将当前处理器缓存行的数据写回到系统内存中。为了保证一致性，其他处理器嗅探到总线上传播的数据，发现数据被修改了使自己缓存地址的数据无效。
2. Volatile 可以**禁止重排序**，
3. Volatile 能保持单个简单volatile变量的读/写操作的具有原子性。但不能保证自增自减的**原子性**。

从**内存语义**来讲:

* volatile变量的写-读与锁的释放-获取具有相同语义，volatile的写与锁的释放有相同的内存语义，volatile读与锁的获取具有相同语义。
* 线程A写一个volatile变量，实质上是线程A向接下来要读这个volatile变量的某个线程发出消息
* 线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的消息。
* 线程A写volatile变量，随后线程B读这个变量，这个过程实质上线程A通过内存向B发送消息。

内存语义的实现，也是禁止重排序特性： 为了实现volatile内存语义，JMM限制了对volatile重排序做了限制：

1. 当第二个操作是volatile写时，不管第一个操作时什么，都不能重排序。
2. 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。
3. 当第一个操作是volatile写，第二个操作是volatile读时，不重排序。

为了实现volatile的内存语义，编译器生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。JMM采取保守策略:

1. 在每个volatile写操作前面插入一个StoreStore屏障
2. 在每个volatile写操作后面插入一个StoreLoad屏障
3. 在每个volatile读操作后面插入一个LoadLoad屏障
4. 在每个volatile读操作后面插入一个LoadStore屏障

### Shutdown和shutdownNow的区别

它们的原理都是遍历线程池中的工作线程，然后逐个调用线程的Internet方法来中断线程，所以无法响应中断的任务可能永远无法终止。

ShutdownNow首先将线程池的状态设置成STOP,然后尝试停止所有正在执行或暂停的任务，并返回等待执行任务的列表。而shutdown只是将线程池设置成SHUTDOWN状态，然后中断没有正在执行任务的线程。

### Sleep()和wait()的区别

sleep()方法是线程类（Thread）的静态方法，导致此线程暂停执行指定时间，将执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复（线程回到就绪（ready）状态），因为调用 sleep 不会释放对象锁。

wait() 是 Object 类的方法，对此对象调用 wait()方法导致本线程放弃对象锁(线程暂停执行)，释放资源并进入等待此对象的等待锁定池，只有针对此对象发出 notify 方法（或 notifyAll）后本线程才进入对象锁定池准备获得对象锁进入就绪状态。

### Sleep()和yield()的区别

① sleep() 方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield() 方法只会给相同优先级或更高优先级的线程以运行的机会；

② 线程执行 sleep() 方法后转入阻塞（blocked）状态，而执行 yield() 方法后转入\*\*就绪（ready）\*\*状态；

③ sleep() 方法声明抛出InterruptedException，而 yield() 方法没有声明任何异常；

④ sleep() 方法比 yield() 方法（跟操作系统相关）具有更好的可移植性。

### 线程同步相关方法

wait():使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉InterruptedException 异常；

notify():唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；

notityAll():唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争；

### 线程池

在面向对象编程中，**创建和销毁对象是很费时间的**，因为创建一个对象要获取内存资源或者其它更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少**创建和销毁对象的次数**，特别是一些很耗资源的对象创建和销毁，这就是"**池化资源**"技术产生的原因。

线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

## 简单设计模式

### 单例

确保一个类只有一个实例，并提供该实例的全局访问点。

使用一个私有构造函数、一个私有静态变量以及一个公有静态函数来实现。私有构造函数保证了不能通过构造函数来创建对象实例，只能通过公有静态函数返回唯一的私有静态变量。

**应用场景：**

1、在Spring中创建的Bean实例默认都是单例模式存在的。在Struts2与spring整合中由于spring 默认都是单例的，而在多用户访问web应用的时候每个用户都应该创建一个action所以应该在spring中配置为多例的。

2、网站的计数器，一般也是采用单例模式实现，否则难以同步。

3、Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

4、数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗

5、多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

### 简单工厂（Simple Factory）

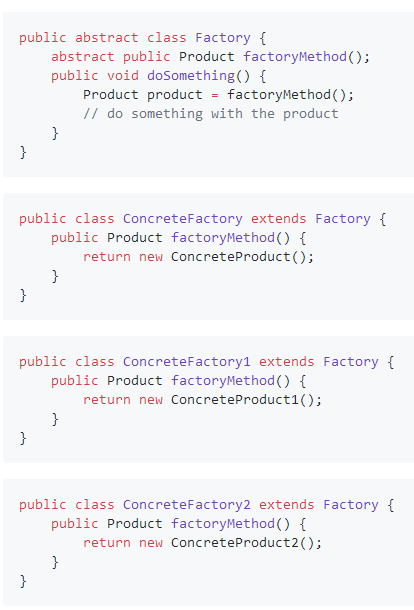
意图：在创建一个对象时不向客户暴露内部细节，并提供一个创建对象的通用接口。

类图：简单工厂不是设计模式，更像是一种编程习惯。它把实例化的操作单独放到一个类中，这个类就成为简单工厂类，让简单工厂类来决定应该用哪个子类来实例化。这样做能把客户类和具体子类的实现解耦，客户类不再需要知道有哪些子类以及应当实例化哪个子类。因为客户类往往有多个，如果不使用简单工厂，所有的客户类都要知道所有子类的细节。而且一旦子类发生改变，例如增加子类，那么所有的客户类都要进行修改。

### 工厂方法（Factory Method）

意图：定义了一个创建对象的接口，但由子类决定要实例化哪个类。工厂方法把实例化推迟到子类。

类图：在简单工厂中，创建对象的是另一个类，而在工厂方法中，是由子类来创建对象。



### 抽象工厂（Abstract Factory）

提供一个接口，用于创建 **相关的对象家族** 。

抽象工厂模式创建的是对象家族，也就是很多对象而不是一个对象，并且这些对象是相关的，也就是说必须一起创建出来。而工厂模式只是用于创建一个对象，这和抽象工厂模式有很大不同。

抽象工厂模式用到了工厂模式来创建单一对象，AbstractFactory 中的 createProductA() 和 createProductB() 方法都是让子类来实现，这两个方法单独来看就是在创建一个对象，这符合工厂模式的定义。

至于创建对象的家族这一概念是在 Client 体现，Client 要通过 AbstractFactory 同时调用两个方法来创建出两个对象，在这里这两个对象就有很大的相关性，Client 需要同时创建出这两个对象。

从高层次来看，抽象工厂使用了组合，即 Cilent 组合了 AbstractFactory，而工厂模式使用了继承。

### 生成器

封装一个对象的构造过程，并允许按步骤构造。

### 迭代器（Iterator）

提供一种顺序访问聚合对象元素的方法，并且不暴露聚合对象的内部表示

* Aggregate 是聚合类，其中 createIterator() 方法可以产生一个 Iterator；
* Iterator 主要定义了 hasNext() 和 next() 方法。
* Client 组合了 Aggregate，为了迭代遍历 Aggregate，也需要组合 Iterator

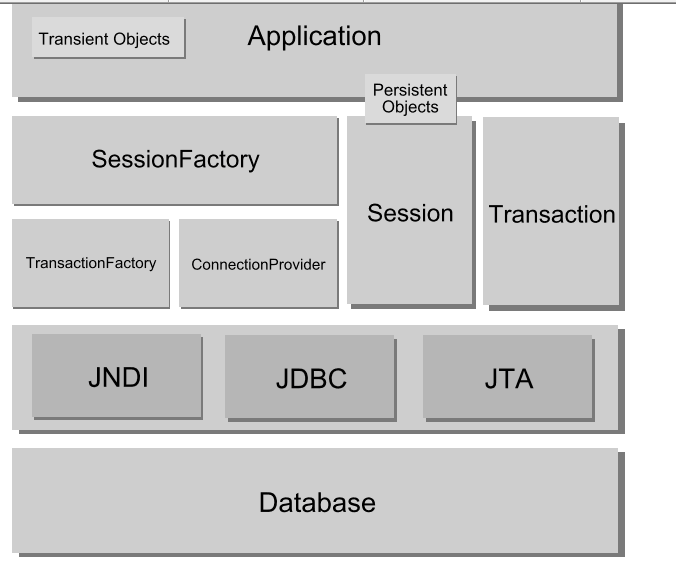
### 装饰者

用于增强对象的功能，为对象动态添加功能。

装饰者（Decorator）和具体组件（ConcreteComponent）都继承自组件（Component），具体组件的方法实现不需要依赖于其它对象，而装饰者组合了一个组件，这样它可以装饰其它装饰者或者具体组件。所谓装饰，就是把这个装饰者套在被装饰上，从而动态扩展被装饰者的功能。装饰者的方法有一部分是自己的，这属于它的功能，然后调用被装饰者的方法实现，从而也保留了被装饰者的功能。可以看到，具体组件应当是装饰层次的最低层，因为只有具体组件的方法实现不需要依赖于其它对象。装饰者（Decorator）和具体组件（ConcreteComponent）都继承自组件（Component），具体组件的方法实现不需要依赖于其它对象，而装饰者组合了一个组件，这样它可以装饰其它装饰者或者具体组件。所谓装饰，就是把这个装饰者套在被装饰上，从而动态扩展被装饰者的功能。装饰者的方法有一部分是自己的，这属于它的功能，然后调用被装饰者的方法实现，从而也保留了被装饰者的功能。可以看到，具体组件应当是装饰层次的最低层，因为只有具体组件的方法实现不需要依赖于其它对象。

## Hibernate相关

### Hibernate的原理体系架构



**1) SessionFactory (org.hibernate.SessionFactory)** 针对单个数据库映射关系经过编译后的内存镜像，是线程安全的（不可变）。它是生成Session的工厂，本身要用到ConnectionProvider。该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

**2) Session (org.hibernate.Session)** 表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短。 其隐藏了JDBC连接，也是Transaction的工厂。其会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。

**3) 事务Transaction (org.hibernate.Transaction)** （可选的）应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短。 它通过抽象将应用从底层具体的JDBC、JTA以及CORBA事务隔离开。 某些情况下，一个Session之内可能包含多个Transaction对象。 尽管是否使用该对象是可选的，但无论是使用底层的API还是使用Transaction对象，事务边界的开启与关闭是必不可少的。

**4) ConnectionProvider (org.hibernate.connection.ConnectionProvider)** （可选的）生成JDBC连接的工厂（同时也起到连接池的作用）。 它通过抽象将应用从底层的Datasource或DriverManager隔离开。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

**5)TransactionFactory (org.hibernate.TransactionFactory)** （可选的）生成Transaction对象实例的工厂。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

**6) 持久的对象及其集合** 带有持久化状态的、具有业务功能的单线程对象，此对象生存期很短。 这些对象可能是普通的JavaBeans/POJO，唯一特殊的是他们正与（仅仅一个）Session相关联。 一旦这个Session被关闭，这些对象就会脱离持久化状态，这样就可被应用程序的任何层自由使用。 （例如，用作跟表示层打交道的数据传输对象。）

**7) 瞬态(transient)和脱管(detached)的对象及其集合** 那些目前没有与session关联的持久化类实例。他们可能是在被应用程序实例化后，尚未进行持久化的对象。也可能是因为实例化他们的Session已经被关闭而脱离持久化的对象。

### 五大核心接口，

Hibernate的核心接口一共有5个，分别为:Session、SessionFactory、Transaction、Query和 Configuration。这5个核心接口在任何开发中都会用到。通过这些接口，不仅可以对持久化对象进行存取，还能够进行事务控制。下面对这五的核心 接口分别加以介绍。

**1) Session接口**: Session接口负责执行被持久化对象的CRUD操作(CRUD的任务是完成与数据库的交流，包含了很多常见的 SQL语句。)。但需要注意的是Session对象是非线程安全的。同时，Hibernate的session不同于JSP应用中的HttpSession。这里当使用session这个术语时，其实指的是Hibernate中的session，而以后会将HttpSesion对象称为用户session。

**2) SessionFactory接口** SessionFactroy接口负责初始化Hibernate。它充当数据存储源的代理，并负责创建 Session对象。这里用到了工厂模式。需要注意的是SessionFactory并不是轻量级的，因为一般情况下，一个项目通常只需要一个SessionFactory就够，当需要操作多个数据库时，可以为每个数据库指定一个SessionFactory。

**3) Configuration接口** Configuration接口负责配置并启动Hibernate，创建SessionFactory对象。在Hibernate的启动的过程中，Configuration类的实例首先定位映射文档位置、读取配置，然后创建SessionFactory对象。

**4) Transaction接口** Transaction接口负责事务相关的操作。它是可选的，开发人员也可以设计编写自己的底层事务处理代码。

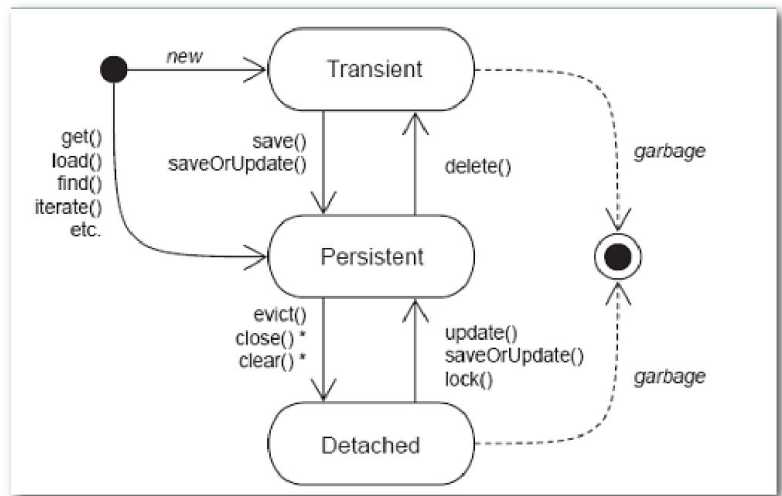
**5) Query和Criteria接口**: Query和Criteria接口负责执行各种数据库查询。它可以使用HQL语言或SQL语句两种表达方式。

### Hibernate对象的三种状态转换

瞬时状态：刚刚创建，刚刚在内存中开辟空间。没有主键值，没有调用session与之关联

游离状态：有主键，但是没有与session关联。

持久化状态：有主键，与session关联。



### Hibernate对一二级缓存的使用

（1）一级缓存就是Session级别的缓存，一个Session做了一个查询操作，它会把这个操作的结果放在一级缓存中，如果短时间内这个session（一定要同一个session）又做了同一个操作，那么hibernate直接从一级缓存中拿，而不会再去连数据库，取数据；

快照：Hibernate向一级缓存放入数据时，同时复制一份数据放入到Hibernate快照中，当使用conmmit（）方法提交事务时，同时会清理session的一级缓存，这时会使用OID判断一级缓存中的对象和快照中的对象是否一致，如果两个对象中的属性发生变化，这时会update语句，将缓存的内容同步到数据库，并更新快照；如果一致，则不执行update。Hibernate快照作用是保证一级缓存中数据和数据库中数据一致。

（2）二级缓存就是SessionFactory级别的缓存，顾名思义，就是查询的时候会把查询结果缓存到二级缓存中，如果同一个sessionFactory创建的某个session执行了相同的操作，hibernate就会从二级缓存中拿结果，而不会再去连接数据库；

### Lazy-Load的理解

我们所说的懒加载也被称为延迟加载，它在查询的时候不会立刻访问数据库，而是返回代理对象，当真正去使用对象的时候才会访问数据库。 需要注意的是，当使用懒加载时，如果将session关闭，则断开了与数据库的连接，此时如果要访问子表数据，由于子表数据是存放在数据库的缓存中，而连接已经关闭，则访问子表数据会抛出LazyInitializationException异常。

1、通过Session.load()实现懒加载。load(Object, Serializable)：根据id查询。查询返回的是代理对象，不会立刻访问数据库，是懒加载的。当真正去使用对象的时候才会访问数据库。用load()的时候会发现不会打印出查询语句，而使用get()的时候会打印出查询语句。 使用load()时如果在session关闭之后再查询此对象，会报异常：could not initialize proxy - no Session。处理办法：在session关闭之前初始化一下查询出来的对象：Hibernate.initialize(user);使用load()可以提高效率，因为刚开始的时候并没有查询数据库。但很少使用。

2、one-to-one(元素)实现了懒加载。在一对一的时候，查询主对象时默认不是懒加载。即：查询主对象的时候也会把从对象查询出来。需要把主对象配制成lazy="true" constrained="true" fetch="select"。此时查询主对象的时候就不会查询从对象，从而实现了懒加载。一对一的时候，查询从对象的是默认是懒加载。即：查询从对象的时候不会把主对象查询出来。而是查询出来的是主对象的代理对象。

3、many-to-one（元素）实现了懒加载。 多对一的时候，查询主对象时默认是懒加载。即：查询主对象的时候不会把从对象查询出来。

4、one-to-many(元素)懒加载：默认会懒加载，这是必须的，是重常用的。 一对多的时候，查询主对象时默认是懒加载。即：查询主对象的时候不会把从对象查询出来。

### 谈谈Hibernate中inverse的作用

inverse属性默认是false,就是说关系的两端都来维护关系。

比如Student和Teacher是多对多关系，用一个中间表TeacherStudent维护。

如果Student这边inverse=”true”, 那么关系由另一端Teacher维护，就是说当插入Student时，不会操作TeacherStudent表（中间表）。只有Teacher插入或删除时才会触发对中间表的操作。所以两边都inverse=”true”是不对的，会导致任何操作都不触发对中间表的影响；当两边都inverse=”false”或默认时，会导致在中间表中插入两次关系。

### 说说Hibernate中的update()和saveOrUpdate()的区别，session的load()和get()的区别。

saveOrUpdate()方法可以实现update()的功能，但会多些步骤，具体如下：

如果对象在该session中已经被持久化，不进行操作；对象的标识符属性(identifier property)在数据库中不存在或者是个暂时的值，调用save()方法保存它；如果session中的另一个对象有相同的标识符抛出一个异常；以上皆不符合则调用update()更新之。

Session.load/get方法均可以根据指定的实体类和id从数据库读取记录，并返回与之对应的实体对象。session的get()和load()其区别在于：

如果未能发现符合条件的记录，**get**方法返回null，而**load**方法会抛出一个ObjectNotFoundException；**load**方法可返回实体的代理类实例，而**get**方法永远直接返回实体类；**load**方法可以充分利用内部缓存和二级缓存中的现有数据，而**get**方法则仅仅在内部缓存中进行数据查找，如没有发现对应数据，将越过二级缓存，直接调用SQL完成数据读取。

### 如何优化hibernate

1. 使用双向一对多关联，不使用单向一对多  
2. 灵活使用单向一对多关联  
3. 不用一对一，用多对一取代  
4. 配置对象缓存，不使用集合缓存  
5. 一对多集合使用Bag,多对多集合使用Set  
6. 继承类使用显式多态  
7. 表字段要少，表关联不要怕多，有二级缓存撑腰

### 为什么使用Hibernate？

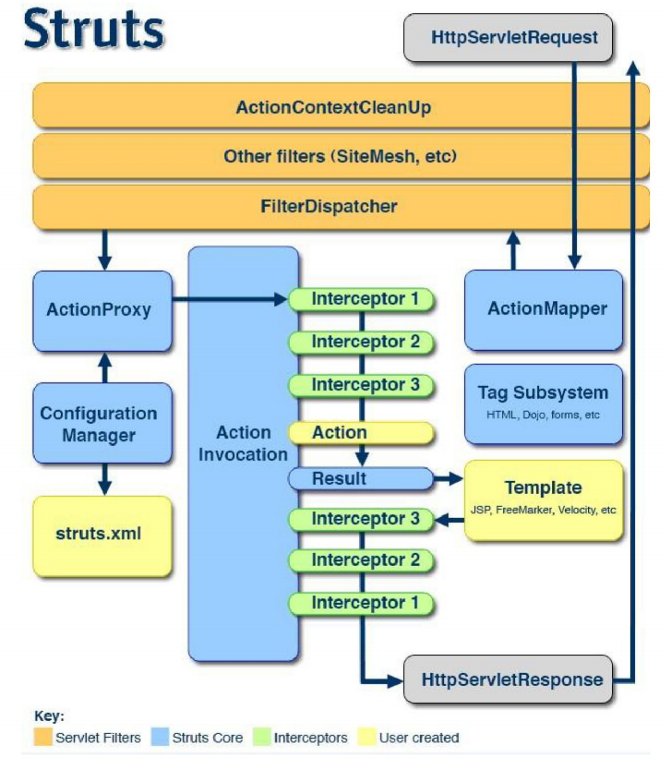
1.    对JDBC访问数据库的代码做了封装，大大简化了数据访问层繁琐的重复性代码。

2.   Hibernate是一个基于JDBC的主流持久化框架，是一个优秀的ORM实现。他很大程度的简化DAO层的编码工作

3.   hibernate使用Java反射机制，而不是字节码增强程序来实现透明性。

4.   hibernate的性能非常好，因为它是个轻量级框架。映射的灵活性很出色。它支持各种关系数据库，从一对一到多对多的各种复杂关系。

## Struts2相关



### Struts工作流程:

客户端提交一个HttpServletRequest请求(.action或JSP页面)

1. 请求被提交到一系列（主要是三层）的过滤器（Filter），如（ActionContextCleanUp、其他过滤器（SiteMesh等）、 FilterDispatcher）。注意这里是有顺序的，先ActionContextCleanUp，再其他过滤器（SiteMesh等）、最后到FilterDispatcher。
2. FilterDispatcher是Struts2控制器的核心,它通常是过滤器链中的最后一个过滤器,FilterDispatcher进行初始化并启用核心doFilter().FilterDispatcher询问ActionMapper是否需要调用某个Action来处理这个（request）请求，如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy。
3. ActionProxy通过ConfigurationManager(它会访问struts.xml)询问框架的配置文件,找到需要调用的Action类.
4. ActionProxy创建一个ActionInvocation实例,而ActionInvocation通过代理模式调用Action,(在调用之前会根据配置文件加载相关的所有Interceptor拦截器)
5. Action执行完毕后,返回一个result字符串,此时再按相反的顺序通过Interceptor拦截器.
6. 最后ActionInvocation负责根据struts.xml中配置的result元素,找到与返回值对应的result,决定进行下一步输出.

### 描述 Struts 体系结构？对应各个部分的开发工作主要包括哪些？

Struts开源架构很好的实现了MVC模式，MVC即Model-View-Controller的缩写，是一种常用的设计模式。MVC 减弱了业务逻辑接口和数据接口之间的耦合，以及让视图层更富于变化。在Struts2的模型 - 视图 - 控制器模式，实现以下五个核心部件：

Actions

Interceptors

Value Stack / OGNL

Results / Result types

View technologies

Struts 2 从传统的MVC框架操作需要的模型，而不是在控制器角色略有不同，虽然有一些重叠

**模型** 模型以一个或多个java bean的形式存在。这些bean分为三类：Action Form、Action、JavaBean or EJB。Action Form通常称之为FormBean，封装了来自于Client的用户请求信息，如表单信息。Action通常称之为ActionBean，获取从ActionSevlet传来的FormBean，取出FormBean中的相关信息，并做出相关的处理，一般是调用Java Bean或EJB等。

**视图** 主要由JSP生成页面完成视图，Struts提供丰富的JSP 标签库： Html，Bean，Logic，Template等

**控制器** 该控制器负责响应用户输入和执行数据模型对象的相互作用。控制器接收输入，验证输入，然后进行业务操作，修改数据模型的状态。

### 什么是Struts2

Apache Struts2的是一个在Java中构建Web应用程序开源框架。 Struts2是基于OpenSymphony的WebWork的框架。它是Struts1的提高，它更加灵活，易于使用和扩展。 Struts2的核心组成部分是Action，拦截器和结果页。

Struts2提供了许多方法来创建Action类，并通过struts.xml中或通过注释进行配置。我们可以创建自己的拦截器实现常见任务。 Struts2中自带了很多的标签，并使用OGNL表达式语言。我们可以创造我们自己的类型转换器来呈现的结果页面。结果页面可以JSP和FreeMarker的模板。

### struts2 与struts1的区别

Action类,线程安全，测试，标签，验证

1. Action 类: • Struts1要求Action类继承一个抽象基类。Struts1的一个普遍问题是使用抽象类编程而不是接口，而struts2的Action是接口。 • Struts 2 Action类可以实现一个Action接口，也可实现其他接口，使可选和定制的服务成为可能。Struts2提供一个ActionSupport基类去 实现 常用的接口。Action接口不是必须的，任何有execute标识的POJO对象都可以用作Struts2的Action对象。
2. 线程模式: • Struts1 Action是单例模式并且必须是线程安全的，因为仅有Action的一个实例来处理所有的请求。单例策略限制了Struts1 Action能作的事，并且要在开发时特别小心。Action资源必须是线程安全的或同步的。(不是线程安全) • Struts2 Action对象为每一个请求产生一个实例，因此没有线程安全问题。（实际上，servlet容器给每个请求产生许多可丢弃的对象，并且不会导致性能和垃圾回收问题）(线程安全)
3. Servlet 依赖: • Struts1 Action 依赖于Servlet API ,因为当一个Action被调用时HttpServletRequest 和 HttpServletResponse 被传递给execute方法。 • Struts 2 Action不依赖于容器，允许Action脱离容器单独被测试。如果需要，Struts2 Action仍然可以访问初始的request和response。但是，其他的元素减少或者消除了直接访问HttpServetRequest 和 HttpServletResponse的必要性。
4. 可测性: • 测试Struts1 Action的一个主要问题是execute方法暴露了servlet API（这使得测试要依赖于容器）。一个第三方扩展－－Struts TestCase－－提供了一套Struts1的模拟对象（来进行测试）。 • Struts 2 Action可以通过初始化、设置属性、调用方法来测试，“依赖注入”支持也使测试更容易。
5. 表达式语言： • Struts1 整合了JSTL，因此使用JSTL EL。这种EL有基本对象图遍历，但是对集合和索引属性的支持很弱。 • Struts2可以使用JSTL，但是也支持一个更强大和灵活的表达式语言－－"Object Graph Notation Language" (OGNL).
6. 绑定值到页面（view）: • Struts 1使用标准JSP机制把对象绑定到页面中来访问。 • Struts 2 使用 "ValueStack"技术，使taglib能够访问值而不需要把你的页面（view）和对象绑定起来。ValueStack策略允许通过一系列名称相同但类型不同的属性重用页面（view）。

### 为什么要使用 Struts2 & Struts2 的优点：

1. . 基于 MVC 架构，框架结构清晰。
2. 使用 OGNL: OGNL 可以快捷的访问值栈中的数据、调用值栈中对象的方法
3. 拦截器: Struts2 的拦截器是一个 Action 级别的 AOP, Struts2 中的许多特性都是通过拦截器来实现的, 例如异常处理，文件上传，验证等。拦截器是可配置与重用的
4. 多种表现层技术. 如：JSP、FreeMarker、Velocity 等

### Filter,Listener,Servlet区别

1. Filter　实现javax.servlet.Filter接口，在web.xml中配置与标签指定使用哪个Filter实现类过滤哪些URL链接。只在web启动时进行初始化操作。filter流程是线性的， url传来之后，检查之后，可保持原来的流程继续向下执行，被下一个filter, servlet接收等，而servlet 处理之后，不会继续向下传递。filter功能可用来保持流程继续按照原来的方式进行下去，或者主导流程，而servlet的功能主要用来主导流程。 特点：可以在响应之前修改Request和Response的头部，只能转发请求，不能直接发出响应。filter可用来进行字符编码的过滤，检测用户是否登陆的过滤，禁止页面缓存等
2. Servlet 流程是短的，url传来之后，就对其进行处理，之后返回或转向到某一自己指定的页面。它主要用来在业务处理之前进行控制。
3. Listener servlet,filter都是针对url之类的，而listener是针对对象操作的，如session的创建，session.setAttribute的发生，在这样的事件发 生时做一些事情。

### Struts2拦截器和过滤器的区别

1. 过滤器依赖于Servlet容器，而拦截器不依赖于Servlet容器。
2. Struts2 拦截器只能对 Action 请求起作用，而过滤器则可以对几乎所有请求起作用。
3. 拦截器可以访问 Action 上下文(ActionContext)、值栈里的对象(ValueStack)，而过滤器不能.
4. 在 Action 的生命周期中，拦截器可以多次调用，而过滤器只能在容器初始化时被调用一次。

### Struts的Action是不是线程安全的

struts2的action是线程安全的，struts1的action不是线程安全的 。 对于struts1 ，Action是单例模式，一个实例来处理所有的请求。当第一次\*\*.do的请求过来时，在内存中的actionmapping中找到相对应的action，然后new出这个action放在缓存中，当第二次一样的请求过来时，还是找的这个action，所以对于struts1来说，action是单实例的，只有一个，如果在action中定义变量，就要非常小心了，因为并发问题，可能带来灾难性的后果，也不是不可以，我们可以加锁达到同步，只是在性能上就 要折衷了。 声明局部变量，或者扩展RequestProcessor，让每次都创建一个Action，或者在spring中用scope=”prototype”来管理，不申明类变量就可以保证线程安全。因为只存在一个Action类实例，所有线程会共享类变量。 struts2 在struts1的基础上做了改进 ，对于struts2 ，每次请求过来都会new一个新的action, 所以说struts2的action没有线程安全问题，是线程安全的，但同时也带来一个问题，每次都new一个action ，这样action的实例太多 ， 在性能方面还是存在一定的缺陷的。

### 说出 struts2 中至少 5 个的默认拦截器

fileUpload      提供文件上传功能   
i18n            记录用户选择的locale   
cookies         使用配置的name,value来是指cookies   
checkbox        添加了checkbox自动处理代码，将没有选中的checkbox的内容设定为false，而html默认情况下不提交没有选中的checkbox。   
chain           让前一个Action的属性可以被后一个Action访问，现在和chain类型的result（）结合使用。   
alias           在不同请求之间将请求参数在不同名字件转换，请求内容不变

exception；fileUpload；i18n；modelDriven；params；prepare；token；tokenSession；validation

### 每个拦截器都是需要实现 Interceptor 接口

init()：在拦截器被创建后立即被调用, 它在拦截器的生命周期内只被调用一次. 可以在该方法中对相关资源进行必要的初始化； intercept(ActionInvocation invocation)：每拦截一个动作请求，该方法就会被调用一次； destroy：该方法将在拦截器被销毁之前被调用, 它在拦截器的生命周期内也只被调用一次；

### 值栈ValueStack的原理与生命周期？

1、ValueStack贯穿整个 Action 的生命周期，保存在request域中，所以ValueStack和request的生命周期一样。当Struts2接受一个请求时，会迅速创建ActionContext，   
ValueStack，action。然后把action存放进ValueStack，所以action的实例变量可以被OGNL访问。 请求来的时候，action、ValueStack的生命开始，请求结束，action、    ValueStack的生命结束；   
2、action是多例的，和Servlet不一样，Servelt是单例的；   
3、每个action的都有一个对应的值栈，值栈存放的数据类型是该action的实例，以及该action中的实例变量，Action对象默认保存在栈顶；   
4、ValueStack本质上就是一个ArrayList；   
5、关于ContextMap，Struts 会把下面这些映射压入 ContextMap 中：   
parameters  :   该 Map 中包含当前请求的请求参数   
request     :   该 Map 中包含当前 request 对象中的所有属性  session :该 Map 中包含当前 session 对象中的所有属性   
application :该 Map 中包含当前 application 对象中的所有属性   
attr:该 Map 按如下顺序来检索某个属性: request, session, application            
6、使用OGNL访问值栈的内容时，不需要#号，而访问request、session、application、attr时，需要加#号；   
7、注意： Struts2中，OGNL表达式需要配合Struts标签才可以使用。如：<s:property value="name"/>   
8、在struts2配置文件中引用ognl表达式 ,引用值栈的值 ，此时使用的"$"，而不是#或者%;

### ActionContext、ServletContext、pageContext的区别？

1、ActionContext是当前的Action的上下文环境，通过ActionContext可以获取到request、session、ServletContext等与Action有关的对象的引用；   
2、ServletContext是域对象，一个web应用中只有一个ServletContext，生命周期伴随整个web应用；   
3、pageContext是JSP中的最重要的一个内置对象，可以通过pageContext获取其他域对象的应用，同时它是一个域对象，作用范围只针对当前页面，当前页面结束时，pageContext销毁，   
生命周期是JSP四个域对象中最小的。

## Spring相关

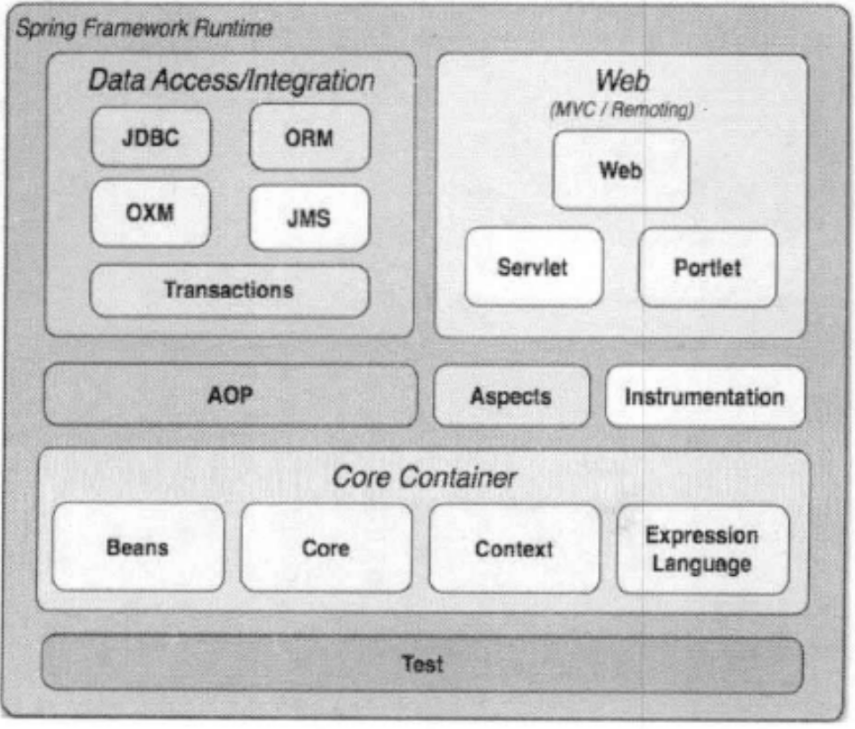
### 什么是spring

Spring是一个开源的Java EE开发框架。Spring框架的核心功能可以应用在任何Java应用程序中，但对Java EE平台上的Web应用程序有更好的扩展性。Spring框架的目标是使得Java EE应用程序的开发更加简捷，通过使用POJO为基础的编程模型促进良好的编程风格。

### Spring有哪些优点

* **轻量级**：Spring在大小和透明性方面绝对属于轻量级的，基础版本的Spring框架大约只有2MB。
* **控制反转(IOC)**：Spring使用控制反转技术实现了松耦合。依赖被注入到对象，而不是创建或寻找依赖对象。
* **面向切面编程(AOP)**： Spring支持面向切面编程，同时把应用的**业务逻辑与系统的服务分离开来**。
* **容器**：Spring包含并管理应用程序对象的配置及生命周期。
* **MVC框架**：Spring的web框架是一个设计优良的web MVC框架，很好的取代了一些web框架。
* **事务管理**：Spring对下至本地业务上至全局业务(JAT)提供了统一的事务管理接口。
* **异常处理**：Spring提供一个方便的API将特定技术的异常(由JDBC, Hibernate, 或JDO抛出)转化为一致的、Unchecked异常

### Spring框架有哪些模块



* **核心容器模块**：是spring中最核心的模块。负责Bean的创建、配置和管理。主要包括：beans,core,context,expression等模块。
* **Spring的AOP模块**：主要负责对面向切面编程的支持，帮助应用对象解耦。
* **数据访问和集成模块**：包括JDBC，ORM，OXM，JMS和事务处理模块，其细节如下：
* JDBC模块提供了不再需要冗长的JDBC编码相关了JDBC的抽象层。
* ORM（Object Relational Mapping，对象关系映射）模块提供的集成层。流行的对象关系映射API，包括JPA，JDO，Hibernate和iBatis。
* OXM模块提供了一个支持对象/ XML映射实现对JAXB，Castor，使用XMLBeans，JiBX和XStream 的抽象层。
* Java消息服务JMS模块包含的功能为生产和消费的信息。
* 事务模块支持编程和声明式事务管理实现特殊接口类，并为所有的POJO。
* **Web和远程调用**：包括web,servlet,struts,portlet模块。
* **测试模块**：test

### 什么是控制反转(IOC)？什么是依赖注入？

传统模式中对象的调用者需要创建被调用对象，两个对象过于耦合，不利于变化和拓展．在spring中，直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理，从而实现对象之间的松耦合。**所谓的“控制反转”概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。**

**依赖注入：**对象无需自行创建或管理它们的依赖关系，IoC容器在运行期间，动态地将某种依赖关系注入到对象之中。依赖注入能让相互协作的软件组件保持松散耦合。

### BeanFactory和ApplicationContext有什么区别？

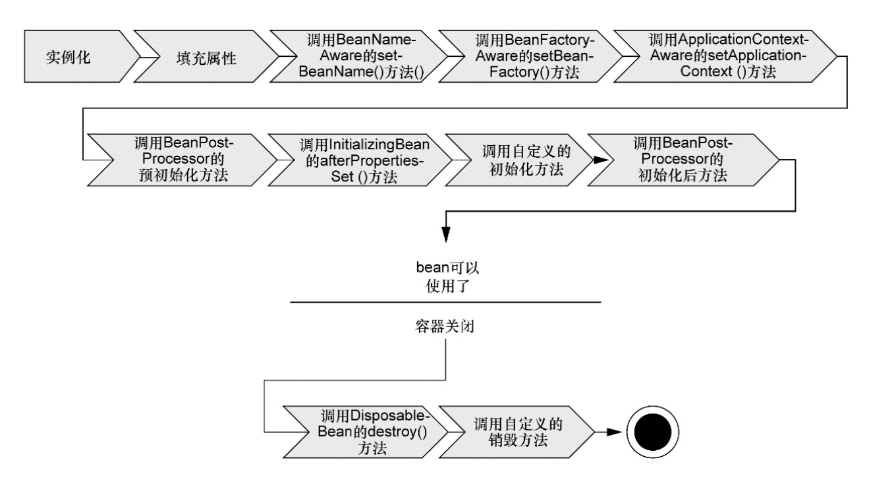
* Bean工厂(BeanFactory)在需要bean对象的时候才创建对象，ApplicationContext在spring启动的时候就会创建所有配置文件中的bean对象，放到自己容器中，需要的时候取就可以了。
* 应用上下文(ApplicationContext)建立在BeanFacotry基础之上，提供了更多面向应用的功能，如果国际化，属性编辑器，事件等等．
* beanFactory是spring框架的基础设施，是面向spring本身。ApplicationContext是面向使用Spring框架的开发者，几乎所有场合都会用到ApplicationContext.

### Spring有几种配置方式？

将Spring配置到应用开发中有以下三种方式：

* **基于XML的配置**:
* **基于注解的配置**： Spring在2.5版本以后开始支持用注解的方式来配置依赖注入。可以用注解的方式来替代XML方式的bean描述，可以将bean描述转移到组件类的内部，只需要在相关类上、方法上或者字段声明上使用注解即可。注解注入将会被容器在XML注入之前被处理，所以后者会覆盖掉前者对于同一个属性的处理结果
* **基于Java的配置**： Spring对Java配置的支持是由@Configuration注解和@Bean注解来实现的。由@Bean注解的方法将会实例化、配置和初始化一个新对象，这个对象将由Spring的IoC容器来管理。@Bean声明所起到的作用与 元素类似。被@Configuration所注解的类则表示这个类的主要目的是作为bean定义的资源。被@Configuration声明的类可以通过在同一个类的内部调用@bean方法来设置嵌入bean的依赖关系。

### Spring Bean的生命周期



Bean在Spring中的生命周期如下：

* **实例化**：Spring通过new关键字将一个Bean进行实例化。
* **填入属性**：spring将值和bean引用注入到bean 的属性中。
* 如果Bean实现了**BeanNameAware**接口，工厂调用Bean的\*\*setBeanName()\*\*方法传递Bean的ID。
* 如果Bean实现了**BeanFactoryAware**接口，工厂调用\*\*setBeanFactory()\*\*方法传入工厂自身。
* 如果实现了**ApplicationContextAware**, spring将调用setApplicationContext()方法，将bean所在的上下文的引用 进来。
* 如果**BeanPostProcessor**和Bean关联，那么它们的\*\*postProcessBeforeInitialization()\*\*方法将被调用。
* 如果Bean指定了init-method方法，它将被调用。
* 如果有**BeanPostProcessor**和Bean关联，那么它们的postProcessAfterInitialization()方法将被调用
* 最后如果配置了destroy-method方法则注册**DisposableBean**.

\*\*使用：\*\*到这个时候，Bean已经可以被应用系统使用了，并且将被保留在Bean Factory中知道它不再需要。

**销毁**。如果Bean实现了DisposableBean接口，就调用其destroy方法。有两种方法可以把它从Bean Factory中删除掉：

1. 如果Bean实现了DisposableBean接口，destory()方法被调用。
2. 如果指定了订制的销毁方法，就调用这个方法。destory-method（）配置时指定。

对几个重要接口的解释：

* **BeanNameAware**: 实现该接口可以获得本身bean的id属性，获得在配置文件中定义好的Bean的ID名
* **BeanFactoryAware**：实现这个接口的bean其实是希望知道自己属于哪一个BeanFactory, 是哪个BeanFactory创建的。
* **ApplicationContextAware**：当一个类实现了这个接口之后，这个类就可以方便地获得 ApplicationContext 中的所有bean。换句话说，就是这个类可以直接获取Spring配置文件中，所有有引用到的bean对象。
* **BeanPostProcessor**是Spring中定义的一个接口，其与InitializingBean和DisposableBean接口类似，也是供Spring进行回调的。Spring将在初始化bean前后对BeanPostProcessor实现类进行回调，Spring容器通过BeanPostProcessor给了我们一个机会对Spring管理的bean进行再加工。比如：我们可以修改bean的属性，可以给bean生成一个动态代理实例等等。

### Spring Bean的作用域之间有什么区别

* **singleton**：这种bean范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个bean的实例，单例的模式由bean factory自身来维护。
* **prototype**：原形范围与单例范围相反，为每一个bean请求提供一个实例。
* **request**：在请求bean范围内会每一个来自**客户端的网络请求**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。
* **Session**：与请求范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
* **global-session**：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。

### Spring 框架中都用到了哪些设计模式

* **代理模式**—在AOP和remoting中被用的比较多。
* **单例模式**—在spring配置文件中定义的bean默认为单例模式。
* **模板方法**—用来解决代码重复的问题. 比如. RestTemplate, JmsTemplate, JpaTemplate。
* **工厂模式**—BeanFactory用来创建对象的实例。
* **Builder模式** - 自定义配置文件的解析bean是时采用builder模式，一步一步地构建一个beanDefinition
* **策略模式** ：Spring 中策略模式使用有多个地方，如 Bean 定义对象的创建以及代理对象的创建等。这里主要看一下代理对象创建的策略模式的实现。前面已经了解 Spring 的代理方式有两个 **Jdk 动态代理**和**CGLIB 代理**。这两个代理方式的使用正是使用了策略模式。

### AOP（Aspect-Oriented Programming）是怎么实现的

实现AOP的技术，主要分为两大类：

* 一是采用**动态代理技术**，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；
* 二是采用**静态织入的方式**，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice和回调目标对象的方法所组成, 并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但AOP代理中的方法与目标对象的方法存在差异，AOP方法在特定切入点添加了**增强处理，并回调了目标对象的方法**。

Spring AOP使用动态代理技术在运行期织入增强代码。使用两种代理机制：**基于JDK的动态代理**（JDK本身只提供接口的代理）和**基于CGlib的动态代理**。

* **(1) JDK的动态代理** JDK的动态代理主要涉及java.lang.reflect包中的两个类：**Proxy和InvocationHandler**。其中InvocationHandler只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

其代理对象**必须是某个接口的实现**,它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理.只能实现接口的类生成代理,而**不能针对类**

* **(2)CGLib** CGLib采用**底层的字节码技术**，**为一个类创建子类**，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑.它运行期间生成的代理对象是目标类的扩展子类.**所以无法通知final、private的方法**,因为它们不能被覆写.是针对类实现代理,主要是为指定的类生成一个子类,覆盖其中方法.

在spring中默认情况下使用JDK动态代理实现AOP,如果proxy-target-class设置为true或者使用了优化策略那么会使用CGLIB来创建动态代理.Spring　AOP在这两种方式的实现上基本一样．以JDK代理为例，会使用JdkDynamicAopProxy来创建代理，在invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入．最终返回代理对象．

### 介绍spring的IOC实现

Spring　**IOC主要负责创建和管理bean及bean之间的依赖关系**．Spring　IOC的可分为:IOC容器的初始化和bean的加载．

在IOC容器初始化阶段主要是完成资源的加载(如定义bean的xml文件)，bean的解析及对解析后得到的beanDefinition的进行注册．以xmlBeanFactory为例，XmlBeanFactory继承了DefaultListableBeanFactory，XmlBeanFactory将读取xml配置文件，解析bean和注册解析后的beanDefinition工作交给XmlBeanDefinitionReader(是BeanDefinitionReader接口的一个个性化实现)来执行.

* 1. spring中定义了一套资源类，将文件，class等都看做资源．所以首先是将xml文件转化为资源然后用EncodeResouce来封装，该功能主要时考虑Resource可能存在编码要求的情况，如UTF-8等．
  2. 然后根据xml文件判断xml的约束模式，是DTD还是Schema,以及寻找模式文档(验证文件)的方法(EntityResolver，这部分采用了代理模式和策略模式)．

完成了前面所有的准备工作以后就可以正式的加载配置文件，获取Document和解析注册BeanDefinition．Document的获取以及BeanDefinition的解析注册并不是由**XmlBeanDefinitionReader**完成，XmlBeanDefinitionReader只是将前面的工作完成以后文档加载交给**DefaultDocumentLoader**类来完成．

而解析交给了**DefaultBeanDefinitionDocumentReader**来处理.bean标签可以分为两种，一种是spring自带的默认标签，另一种就是用户自定义的标签．所以spring针对这两种情况，提供了不同的解析方式. 每种bean的解析完成后都会先注册到容器中然后最后发出响应事件，通知相关的监听器这个bean已经注册完成了．

### spring中bean加载机制，bean生成的具体步骤

1. 容器寻找Bean的定义信息并且将其实例化。
2. 如果允许提前暴露工厂，则提前暴露这个bean的工厂，这个工厂主要是返回该未完全处理的bean．主要是用于避免单例属性循环依赖问题．
3. 受用**依赖注入**，Spring按照Bean定义信息配置Bean的所有属性。
4. 如果Bean实现了**BeanNameAware**接口，工厂调用Bean的\*\*setBeanName()\*\*方法传递Bean的ID。
5. 如果Bean实现了**BeanFactoryAware**接口，工厂调用\*\*setBeanFactory()\*\*方法传入工厂自身。
6. 如果实现了**ApplicationContextAware**, spring将调用setApplicationContext()方法，将bean所在的上下文的引用 进来。
7. 如果**BeanPostProcessor**和Bean关联，那么它们的\*\*postProcessBeforeInitialization()\*\*方法将被调用。
8. 如果Bean指定了init-method方法，它将被调用。
9. 如果有**BeanPostProcessor**和Bean关联，那么它们的postProcessAfterInitialization()方法将被调用
10. 最后如果配置了destroy-method方法则注册**DisposableBean**.

### AOP相关概念

1. **方面（Aspect）**：一个**关注点**的模块化，这个关注点实现可能横切多个对象。事务管理是J2EE应用中一个很好的横切关注点例子。方面用Spring的 Advisor或拦截器实现。
2. **连接点（Joinpoint）**: 程序执行过程中明确的点，如方法的调用或特定的异常被抛出。可以理解为什么时机执行aop的代码。
3. **通知（Advice）:** 在特定的连接点，AOP框架执行的**动作**(怎么执行)。各种类型的通知包括“around”、“before”和“throws”等通知。通知类型将在下面讨论。许多AOP框架包括Spring都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。Spring中定义了5个advice:
   1. Interception Around：JointPoint前后调用
   2. Before：JointPoint前调用
   3. After Returning：JointPoint后调用
   4. Throw：JoinPoint抛出异常时调用
   5. Introduction：JointPoint调用完毕后调用
4. **切入点（Pointcut）**: 一系列连接点的集合。AOP框架必须允许开发者指定切入点：例如，使用正则表达式。 Spring定义了Pointcut接口，用来组合MethodMatcher和ClassFilter，可以通过名字很清楚的理解， MethodMatcher是用来检查目标类的方法是否可以被应用此通知，而ClassFilter是用来检查Pointcut是否应该应用到目标类上
5. **引入（Introduction）**: 添加方法或字段到被通知的类。 Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified接口，来简化缓存。Spring中要使用Introduction, 可有通过DelegatingIntroductionInterceptor来实现通知，通过DefaultIntroductionAdvisor来配置Advice和代理类要实现的接口
6. **目标对象（Target Object）**: 包含连接点的对象。也被称作被通知或被代理对象。POJO
7. **AOP代理（AOP Proxy）**: AOP框架创建的对象，包含通知。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。
8. **织入（Weaving）**: 组装方面来创建一个被通知对象。这可以在编译时完成（例如使用AspectJ编译器），也可以在运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

### 其他知识点

**请描述一下java事件监听机制。**

* Java的事件监听机制涉及到三个组件：事件源、事件监听器、事件对象
* 当事件源上发生操作时，它将会调用事件监听器的一个方法，并在调用这个方法时，会传递事件对象过来
* 事件监听器由开发人员编写，开发人员在事件监听器中，通过事件对象可以拿到事件源，从而对事件源上的操作进行处理。

**解释核心容器(应用上下文)模块**

这是Spring的基本模块，它提供了Spring框架的基本功能。BeanFactory 是所有Spring应用的核心。Spring框架是建立在这个模块之上的，这也使得Spring成为一个容器。

**Spring IoC容器**

Spring IOC负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

**IOC有什么优点？**

IOC或依赖注入减少了应用程序的代码量。它使得应用程序的**测试很简单**，因为在单元测试中不再需要单例或JNDI查找机制。简单的实现以及较少的干扰机制使得**松耦合**得以实现。IOC容器支持勤性单例及延迟加载服务。

**有哪些不同类型的IOC(依赖注入)**

**接口注入**:接口注入的意思是通过接口来实现信息的注入，而其它的类要实现该接口时，就可以实现了注入

**构造器依赖注入：**构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。

**Setter方法依赖注入：**首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用bean中的setter方法完成Setter方法依赖注入。

**你推荐哪种依赖注入？构造器依赖注入还是Setter方法依赖注入？**

你可以同时使用两种方式的依赖注入，最好的选择是使用构造器参数实现强制依赖注入，使用setter方法实现可选的依赖关系。

**什么是Spring Beans**

Spring Beans是构成Spring应用核心的Java对象。这些对象由Spring IOC容器实例化、组装、管理。这些对象通过容器中配置的元数据创建，例如，使用XML文件中定义的创建。

在Spring中创建的beans都是单例的beans。在bean标签中有一个属性为”singleton”,如果设为true，该bean是单例的，如果设为false，该bean是原型bean。Singleton属性默认设置为true。因此，spring框架中所有的bean都默认为单例bean。

**Spring Bean中定义了什么内容？**

Spring Bean中定义了所有的配置元数据，这些配置信息告知容器如何创建它，它的生命周期是什么以及它的依赖关系。

**如何向Spring 容器提供配置元数据**

有三种方式向Spring 容器提供元数据:

* XML配置文件
* 基于注解配置
* 基于Java的配置

**Spring框架中单例beans是线程安全的吗？**

不是，Spring框架中的单例beans不是线程安全的。

**Spring框架并没有对单例bean进行任何多线程的封装处理**。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。

**哪些是最重要的bean生命周期方法？能重写它们吗？**

有两个重要的bean生命周期方法。

* 第一个是setup方法，该方法在容器加载bean的时候被调用。
* 第二个是teardown方法，该方法在bean从容器中移除的时候调用。
* bean标签有两个重要的属性(init-method 和 destroy-method)，你可以通过这两个属性定义自己的初始化方法和析构方法。Spring也有相应的注解：@PostConstruct 和 @PreDestroy。

**什么是Spring的内部bean**

当一个bean被用作另一个bean的属性时，这个bean可以被声明为内部bean。在基于XML的配置元数据中，可以通过把元素定义在 或元素内部实现定义内部bean。内部bean总是匿名的并且它们的scope总是prototype。

**Spring支持的事务管理类型**

Spring支持如下两种方式的事务管理：

* **编码式事务管理**：sping对编码式事务的支持与EJB有很大区别，不像EJB与java事务API耦合在一起．spring通过回调机制将实际的事务实现从事务性代码中抽象出来．**你能够精确控制事务的边界，它们的开始和结束完全取决于你**．
* **声明式事务管理**：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者XML配置管理事务。通过传播行为，隔离级别，回滚规则，事务超时，只读提示来定义．

**Spring框架的事务管理有哪些优点**

* 它为不同的事务API(如JTA, JDBC, Hibernate, JPA, 和JDO)提供了统一的编程模型。
* 它为编程式事务管理提供了一个简单的API而非一系列复杂的事务API(如JTA).
* 它支持声明式事务管理。
* 它可以和Spring 的多种数据访问技术很好的融合。

**ACID**

* **原子性(Atomic)**:一个操作要么成功，要么全部不执行.
* **一致性(Consistent)**: 一旦事务完成，系统必须确保它所建模业务处于一致的状态
* **隔离性(Isolated)**: 事务允许多个用户对相同的数据进行操作，每个用户用户的操作相互隔离互补影响．
* **持久性(Durable)**: 一旦事务完成，事务的结果应该持久化．

**spring事务支持的隔离级别**

并发会导致以下问题：

* **脏读**：发生在一个事务读取了另一个事务改写但尚未提交的数据．
* **不可重复读**：在一个事务执行相同的查询两次或两次以上，每次得到的数据不同．
* **幻读**：与不可重复读类似，发生在一个事务读取多行数据，接着另一个并发事务插入一些数据，随后查询中，第一个事务发现多了一些原本不存在的数据．

spring 事务上提供以下的隔离级别:

* **ISOLATION\_DEFAULT**: 使用后端数据库默认的隔离级别，大多数数据库系统的默认隔离级别是READ COMMITTED。
* **ISOLATION\_READ\_UNCOMMITTED**: 允许**读取未提交**的数据变更，可能会导致**脏读**，幻读或不可重复读
* **ISOLATION\_READ\_COMMITTD**: 允许读取为**提交数据**,可以阻止脏读，当时幻读或不可重复读仍可能发生
* **ISOLATION\_REPEATABLE\_READ**: 对统一字段多次读取结果是一致的，除非数据是被本事务自己修改．可以阻止脏读，不可重复读，但幻读可能发生
* **ISOLATION\_SERIALIZABLE**:　完全服从ACID

**你更推荐那种类型的事务管理？**

许多Spring框架的用户选择声明式事务管理，因为这种方式和应用程序的关联较少，因此更加符合轻量级容器的概念。声明式事务管理要优于编程式事务管理，尽管在灵活性方面它弱于编程式事务管理(这种方式允许你通过代码控制业务)。

**什么是织入？什么是织入应用的不同点？**

织入是将切面和其他应用类型或对象连接起来创建一个通知对象的过程。织入可以在编译、加载或运行时完成。