JavaSE 常见面试题-高级篇

一、 Java 中的反射

1-1 .说说你对 Java 中反射的理解

Java 中的反射首先是能够获取到 Java 中要反射类的字节码，获取字节码
有三种方法，
1.Class.forName(className)
2.类名.class
3.this.getClass()。然后将字节码中的方法，变量，构造函数等映射成相应的 Method、Filed、Constructor 等类，
这些类提供了丰富的方法可以被我们所使用

二、 Java 中的动态代理

2-1 写一个 ArrayList 的动态代理类（笔试题）

```
final List<String> list = new ArrayList<String>();

List<String> proxyInstance =

(List<String>)Proxy.newProxyInstance(list.getClass().getClassLoader(),

list.getClass().getInterfaces(),

new InvocationHandler() {

        @Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{

return method.invoke(list, args);

}

});

proxyInstance.add("你好");

System.out.println(list);
```

2-2 动静态代理的区别，什么场景使用?

静态代理通常只代理一个类，动态代理是代理一个接口下的多个实现类。
静态代理事先知道要代理的是什么，而动态代理不知道要代理什么东西，只有在运行时才知道。
动态代理是实现 JDK 里的 InvocationHandler 接口的 invoke 方法，但注意的是代理的是接口，也就是你的业务类必须要实现接口，通过 Proxy 里的 newProxyInstance 得到

代理对象。还有一种动态代理 CGLIB，代理的是类，不需要业务类继承接口，通过派生的子类来实现代理。通过在运行时，动态修改字节码达到修改类的目的。

AOP 编程就是基于动态代理实现的，比如著名的 Spring 框架、Hibernate 框架等等都是动态代理的使用例子。

三、 Java 中的设计模式&回收机制

3-1.你所知道的设计模式有哪些?

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

3-2 单例设计模式

最好理解的一种设计模式，分为懒汉式和饿汉式。

饿汉式

```
1. public class Singleton {
2.  // 直接创建对象
3.  public static Singleton instance = new Singleton();
4.
5.  // 私有化构造函数
6.  private Singleton() {
7.  }
8.
9.  // 返回对象实例
10. public static Singleton getInstance() {
11.     return instance;
12. }
13. }
```

懒汉式

```
1. public class Singleton {
2.  // 声明变量
3.  private static volatile Singleton singleton = null;
4.
5.  // 私有构造函数
6.  private Singleton() {
7.  }
8.
9.  // 提供对外方法
10. public static Singleton getInstance() {
11.     if (singleton == null) {
12.         synchronized (Singleton.class) {
13.             if (singleton == null) {
14.                 singleton = new Singleton();
15.             }
16.         }
17.     }
18.     return singleton;
19. }
20. }
```

3-3 工厂设计模式

工厂模式分为工厂方法模式和抽象工厂模式。

工厂方法模式

工厂方法模式分为三种:

普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

普通工厂模式:

```
1. public interface Sender {
2.  public void Send();
3. }
4. public class MailSender implements Sender {
5.
6.  @Override
7.  public void Send() {
8.      System.out.println("this is mail sender!");
9.  }
10. }
11. public class SmsSender implements Sender {
12.
13. @Override
14. public void Send() {
15.     System.out.println("this is sms sender!");
```

```
16. }
17. }
18. public class SendFactory {
19. public Sender produce(String type) {
20.     if ("mail".equals(type)) {
21.         return new MailSender();
22.     } else if ("sms".equals(type)) {
23.         return new SmsSender();
24.     } else {
25.         System.out.println("请输入正确的类型!");
26.         return null;
27.     }
28. }
29. }
```

多个工厂方法模式:

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```
1. public class SendFactory {
2.  public Sender produceMail(){
3.       return new MailSender();
4.    }
5.
6.    public Sender produceSms(){
7.       return new SmsSender();
8.    }
9. }
10.
11. public class FactoryTest {
12. public static void main(String[] args) {
13.     SendFactory factory = new SendFactory();
14.     Sender sender = factory.produceMail();
15.     sender.send();
16. }
17. }
```

静态工厂方法模式,

将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
1. public class SendFactory {
2.  public static Sender produceMail(){
3.       return new MailSender();
4.    }
5.
6.    public static Sender produceSms(){
7.       return new SmsSender();
8.    }
9. }
10.
11.
12. public class FactoryTest {
13. public static void main(String[] args) {
14.     Sender sender = SendFactory.produceMail();
15.     sender.send();
16. }
17. }
```

四、 JDK7 和 JDK8 区别

JDK8

1、接口可以添加默认方法，default;

2、lambda 表达式，对于接口可以直接用()->{}方式来表达，小括号表示方法入参，花括号内表示方法返回值，如 Collections 的 sort()方法：

3、函数式接口

4、JDK8 加强了反射，它允许你直接通过反射获取参数的名字

5、Stream API

JDK7：

1、switch 中使用 string

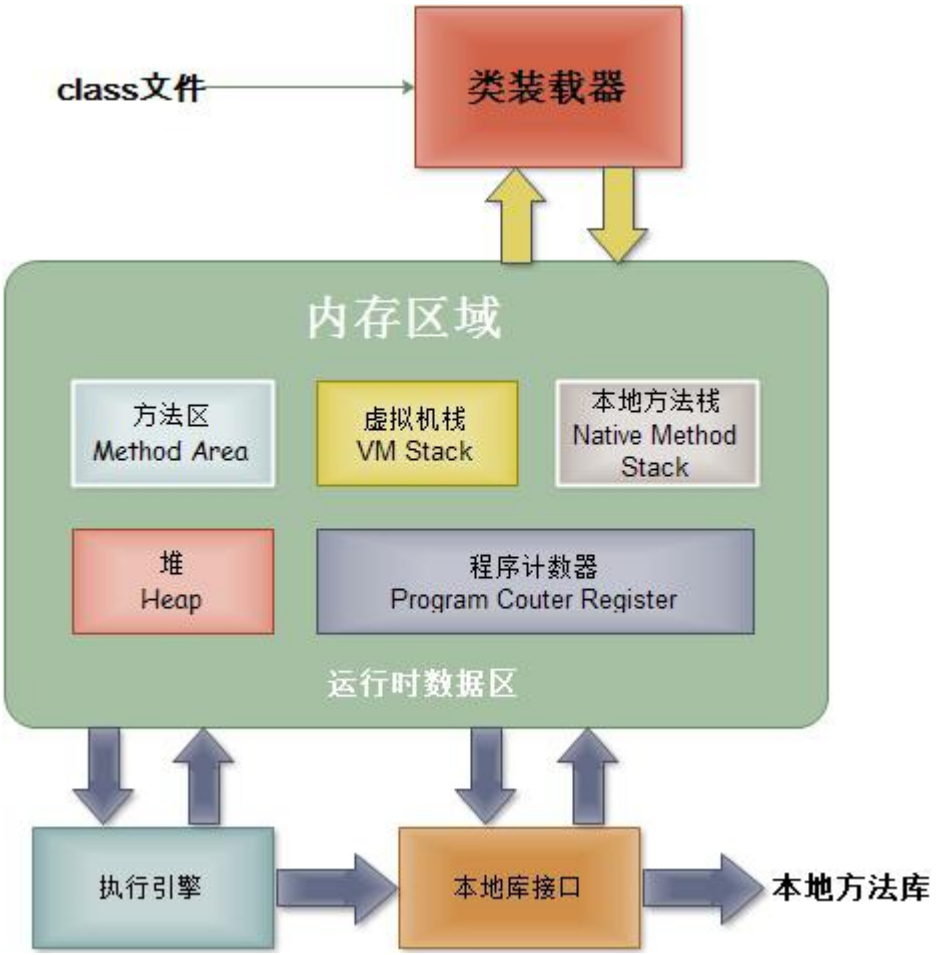
2、对集合类的语言支持；

3、Boolean 类型反转，空指针安全,参与位运算；

五、 Jvm 虚拟机原理

Java 虚拟机(Jvm)是可运行 Java 代码的假想计算机 Java 虚拟机包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。我们都知道 Java 源文件，通过编译器，能够生产相应的.Class 文件，也就是字节码文件，而字节码文件又通过 Java 虚拟机中的解释器，也就是前面所有的 Java 虚拟机中的字节码指令集.... 编译成特定机器上的机器码 1. Java 源文件—->编译器—->字节码文件

2. 字节码文件—->Jvm—->机器码 每一种平台的解释器是不同的，但是实现的虚拟机是相同的。这也就是 Java 为什么能够跨平台的原因了 当一个程序从开始运行一个程序，这时虚拟机就开始实例化了。多个程序启动就会存在多个虚拟机实例。程序退出或者关闭。则虚拟机实例消亡。多个虚拟机实例之间数据不能共享。



垃圾回收器（又称为 gc）：是负责回收内存中无用的对象（好像地球人都知道），就是这些对象没有任何引用了，它就会被视为：垃圾，也就被干掉了。虚拟机内存或者 Jvm 内存，冲整个计算机内存中开辟一块内存存储 Jvm 需要用到的对象，变量等，运行区数据有分很多小区，分别为：方法区，虚拟机栈，本地方法栈，堆，程序计数器

1. 程序计数器 当前线程执行字节码的信号指示器，线程是私有的，它的生命周期和线程相同分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。2. 虚拟机栈 Java 虚拟机栈描述的是 Java 方法（区别于 native 的本地方法）执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动作链接、方法出口等信息。线程私有，生命周期和线程相同，都有各个独立的计数器，各不影响。每个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。本地方法栈 和虚拟机方法栈差不多类似，但是本地方法栈是服务于虚拟机所使用到的 Native 方法服务。本地方法区：只是执行 Native 方法。如果这个区

的内存不足也是会抛出 StackOverflowError 和 OutOfMemoryError 异常。堆这块区域是 Jvm 中最大的，应用的对象和数据都是存在这个区域。这块区域也是线程共享的。也是 gc 主要的回收区。