目录

[问题： 3](#_Toc526367988)

[一、 成员变量的创建 3](#_Toc526367989)

[二、 Java基础 3](#_Toc526367990)

[1. Static 3](#_Toc526367991)

[2. Final 4](#_Toc526367992)

[3. Transient 4](#_Toc526367993)

[4. Volatile 4](#_Toc526367994)

[5. 并发编程的三大特性 5](#_Toc526367995)

[6. Interface与abstract类的区别 5](#_Toc526367996)

[7. 堆和栈 6](#_Toc526367997)

[8. 数据类型的默认值和长度 6](#_Toc526367998)

[A. 默认值 6](#_Toc526367999)

[B. 长度 6](#_Toc526368000)

[9. 补码 6](#_Toc526368001)

[三、 多线程 7](#_Toc526368002)

[1. 状态 7](#_Toc526368003)

[2. 创建线程的方法 9](#_Toc526368004)

[A. 通过实现 Runnable 接口 9](#_Toc526368005)

[B. 通过继承 Thread 类本身 10](#_Toc526368006)

[C. 通过 Callable 和 Future 创建线程。 13](#_Toc526368007)

[D. 创建线程的三种方式的对比 14](#_Toc526368008)

[E. 线程池 14](#_Toc526368009)

[a) 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。（是什么） 14](#_Toc526368010)

[b) 那么，我们为什么需要用到线程池呢？每次用的时候手动创建不行吗？ 14](#_Toc526368011)

[c) 线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。 15](#_Toc526368012)

[d) 这里介绍两种使用线程池创建线程的方法 15](#_Toc526368013)

[3. 线程的优先级 17](#_Toc526368014)

[4. JAVA多线程和并发基础面试问答 17](#_Toc526368015)

[1) 进程和线程之间有什么不同？ 17](#_Toc526368016)

[2) 多线程编程的好处是什么？ 18](#_Toc526368017)

[3) 用户线程和守护线程有什么区别？ 18](#_Toc526368018)

[4) 我们如何创建一个线程？ 18](#_Toc526368019)

[5) 有哪些不同的线程生命周期？ 18](#_Toc526368020)

[6) 可以直接调用Thread类的run()方法么？ 19](#_Toc526368021)

[7) 如何让正在运行的线程暂停一段时间？ 19](#_Toc526368022)

[8) 你对线程优先级的理解是什么？ 19](#_Toc526368023)

[9) 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？ 19](#_Toc526368024)

[10) 在多线程中，什么是上下文切换(context-switching)？ 20](#_Toc526368025)

[11) 你如何确保main()方法所在的线程是Java程序最后结束的线程？ 20](#_Toc526368026)

[12) 线程之间是如何通信的？ 20](#_Toc526368027)

[13) 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？ 20](#_Toc526368028)

[14) 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？ 21](#_Toc526368029)

[15) 为什么Thread类的sleep()和yield()方法是静态的？ 21](#_Toc526368030)

[16) 如何确保线程安全？ 21](#_Toc526368031)

[17) volatile关键字在Java中有什么作用？ 21](#_Toc526368032)

[18) 同步方法和同步块，哪个是更好的选择？ 22](#_Toc526368033)

[19) 如何创建守护线程？ 22](#_Toc526368034)

[20) 什么是ThreadLocal? 22](#_Toc526368035)

[21) 什么是Thread Group？为什么不建议使用它？ 22](#_Toc526368036)

[22) 什么是Java线程转储(Thread Dump)，如何得到它？ 23](#_Toc526368037)

[23) 什么是死锁(Deadlock)？如何分析和避免死锁？ 23](#_Toc526368038)

[24) 什么是Java Timer类？如何创建一个有特定时间间隔的任务？ 24](#_Toc526368039)

[25) 什么是线程池？如何创建一个Java线程池？ 24](#_Toc526368040)

[5. Java并发面试问题 24](#_Toc526368041)

[1) 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？ 24](#_Toc526368042)

[2) Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？ 25](#_Toc526368043)

[3) 什么是Executors框架？ 25](#_Toc526368044)

[4) 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？ 26](#_Toc526368045)

[5) 什么是Callable和Future? 26](#_Toc526368046)

[6) 什么是FutureTask? 27](#_Toc526368047)

[7) 什么是并发容器的实现？ 27](#_Toc526368048)

[8) Executors类是什么？ 27](#_Toc526368049)

[9) 为什么说ConcurrentHashMap是弱一致性的？以及为何多个线程并发修改ConcurrentHashMap时不会报ConcurrentModificationException？ 28](#_Toc526368050)

[四、 异常 28](#_Toc526368051)

[1. 异常类型 28](#_Toc526368052)

[2. finally关键字 28](#_Toc526368053)

[3. Java内置异常 29](#_Toc526368054)

[4. 异常方法 30](#_Toc526368055)

[5. 使用throws抛出异常 31](#_Toc526368056)

[6. 使用throw抛出异常 32](#_Toc526368057)

[Java7增强的throw语句： 32](#_Toc526368058)

[7. 自定义异常类 33](#_Toc526368059)

[8. catch和throw同时使用 33](#_Toc526368060)

[9. 异常链 34](#_Toc526368061)

[10. Java的异常跟踪栈 35](#_Toc526368062)

[11. 异常处理规则 35](#_Toc526368063)

[五、 设计模式 35](#_Toc526368064)

[1. 单例模式 Singleton 35](#_Toc526368065)

[2. 简单工厂模式 StaticFactory Method 36](#_Toc526368066)

[3. 工厂模式 Factory Method 36](#_Toc526368067)

[4. 抽象工厂模式 Abstract Factory Method 36](#_Toc526368068)

[5. 代理模式 Proxy Method 36](#_Toc526368069)

[6. 命令模式 Command Method 36](#_Toc526368070)

[7. 策略模式 Strategy Method 36](#_Toc526368071)

[8. 观察者模式 Observer Method 37](#_Toc526368072)

[9. 生产者消费者模式 ProducerConsumer Method 37](#_Toc526368073)

## 问题：

## Java基础

### 变量的创建

成员/实例变量在类实例化对象时候创建，在对象销毁的时候销毁。

局部变量在局部范围使用时创建，跳出局部范围销毁。

静态成员/类变量在类装载的时候就进行了创建，在整个程序结束时按序销毁。

1.局部变量：在方法、构造方法、语句块中定义的变量。其声明和初始化在方法中实现，在方法结束后自动销毁

public class ClassName{

    public void printNumber（）{

        int a;

    }

    // 其他代码

}

2.成员变量：定义在类中，方法体之外。变量在创建对象时实例化。成员变量可被类中的方法、构造方法以及特定类的语句块访问。

public class ClassName{

    int a;

    public void printNumber（）{

        // 其他代码

    }

}

3.类变量：定义在类中，方法体之外，但必须要有 static 来声明变量类型。静态成员属于整个类，可通过对象名或类名来调用。

public class ClassName{

    static int a;

    public void printNumber（）{

        // 其他代码

    }

}

### 访问控制

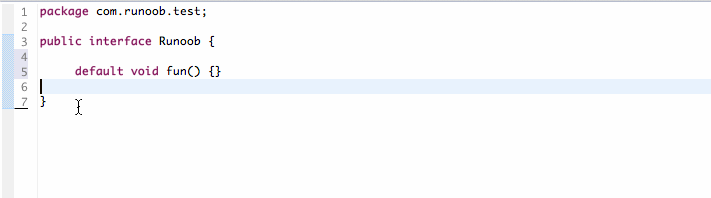


（1）：受保护的访问修饰符-protected

protected 需要从以下两个点来分析说明：

* **子类与基类在同一包中**：被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问；
* **子类与基类不在同一包中**：那么在子类中，子类实例可以访问其从基类继承而来的 protected 方法，而不能访问基类实例的protected方法。
* package p2;  
   class MyObject2 {  
   protected Object clone() throws CloneNotSupportedException{  
   return super.clone();  
   }  
   }  
     
  package p22;  
   public class Test2 extends MyObject2 {  
   public static void main(String args[]) {  
   MyObject2 obj = new MyObject2();  
   obj.clone(); // Compile Error ----（1）  
    
   Test2 tobj = new Test2();  
   tobj.clone(); // Complie OK ----（2）  
   }  
   }
* 对于(1)而言，clone()方法来自于类MyObject2本身，因此其可见性为包p2及MyObject2的子类，虽然Test2是MyObject2的子类，但在Test2中不能访问基类MyObject2的protected方法clone()，因此编译不通过;对于(2)而言，由于在Test2中访问的是其本身实例的从基类MyObject2继承来的的clone()，因此编译通过。

protected 访问修饰符可以修饰类及其方法和成员变量，但是接口及接口的成员变量和成员方法不能声明为 protected。 可以看看以下颜色图理解：



子类能访问 protected 修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

下面的父类使用了 protected 访问修饰符，子类重写了父类的 openSpeaker() 方法。

class AudioPlayer {  
 protected boolean openSpeaker(Speaker sp) {  
 // 实现细节  
 }  
}  
  
class StreamingAudioPlayer extends AudioPlayer {  
 protected boolean openSpeaker(Speaker sp) {  
 // 实现细节  
 }  
}

如果把 openSpeaker() 方法声明为 private，那么除了 AudioPlayer 之外的类将不能访问该方法。

如果把 openSpeaker() 声明为 public，那么所有的类都能够访问该方法。

如果我们只想让该方法对其所在类的子类可见，则将该方法声明为 protected。

### 代码块

#### 普通代码块

在方法或语句中出现的{}就称为普通代码块。普通代码块和一般语句的执行顺序由他们在代码中出现的次序决定，先出现先执行。

public class Test {  
 public static void main(String[] args) {  
 {  
 int x = 3;  
 System.*out*.println("普通代码块内的变量x=" + x);  
 }  
 int x = 1;  
 System.*out*.println("主方法内的变量x=" + x);  
 {  
 int y = 7;  
 System.*out*.println("普通代码块内的变量y=" + y);  
 }  
 }  
}  
/\*  
 \* 运行结果 普通代码块内的变量x=3  
 \* 主方法内的变量x=1  
 \* 普通代码块内的变量y=7  
 \*/

#### 构造代码块

直接在类中定义且没有加static关键字的代码块称为{}构造代码块。构造代码块在创建对象时被调用，每次创建对象都会被调用，并且构造代码块的执行次序优先于类构造函数。如果存在多个构造代码块，执行顺序由他们在代码中出现的次序决定，先出现先执行。

public class Test1 {  
 {  
 System.*out*.println("第一构造块");  
 }  
  
 public Test1(int i) {  
 System.*out*.println("第" + i + "次调用" + "构造方法");  
 }  
  
 {  
 System.*out*.println("第二构造块");  
 }  
  
 public static void main(String[] args) {  
 new Test1(0);  
 new Test1(1);  
 new Test1(2);  
  
 }  
}  
  
/\*  
 \* 执行结果 第一构造块  
 \* 第二构造块  
 \* 第0次调用构造方法  
 \* 第一构造块  
 \* 第二构造块  
 \* 第1次调用构造方法  
 \* 第一构造块  
 \* 第二构造块  
 \* 第2次调用构造方法  
 \*/

#### 静态代码块:

在java中使用static关键字声明的代码块。静态块用于初始化类，为类的属性初始化。每个静态代码块只会执行一次。由于JVM在加载类时会执行静态代码块，所以静态代码块先于主方法执行。  
如果类中包含多个静态代码块，那么将按照"先定义的代码先执行，后定义的代码后执行"。  
      注意：1 .静态代码块不能存在于任何方法体内。

2 .静态代码块不能直接访问实例变量和实例方法，需要通过类的实例对象来访问。

public class Test3 {  
 public static String *STATIC\_FIELD* = "静态属性";  
  
 // 静态块  
 static {  
 System.*out*.println(*STATIC\_FIELD*);  
 System.*out*.println("静态代码块1");  
 }  
  
 public String field = "非静态属性";  
  
 // 非静态块  
 {  
 System.*out*.println(field);  
 System.*out*.println("非静态代码块2");  
 }  
  
 public InitOderTest() {  
 System.*out*.println("无参构造函数");  
 }  
  
 public static void main(String[] args) {  
 InitOderTest test = new InitOderTest();  
 }  
  
 // 非静态块  
 {  
 System.*out*.println(field);  
 System.*out*.println("非静态代码块1");  
 }  
 // 静态块  
 static {  
 System.*out*.println(*STATIC\_FIELD*);  
 System.*out*.println("静态代码块2");  
 }  
  
}  
/\*  
 \* 运行结果 静态属性  
 \* 静态代码块1  
 \* 静态属性  
 \* 静态代码块2  
 \* 非静态属性  
 \* 非静态代码块2  
 \* 非静态属性  
 \* 非静态代码块1  
 \* 无参构造函数  
 \*/

--------------------- 本文来自 krisll 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u012804721/article/details/52439311?utm\_source=copy

### Static

静态变量只能在类主体中定义，不能在方法中定义

成员方法又称为实例方法

静态方法又称为类方法

类方法是指类中被static修饰的方法，无this指针。

类方法是可以调用其他类的static方法的。

* 1. 方便在没有创建对象的情况下来进行调用（方法/变量）。
  2. 在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。
  3. 静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。
  4. static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。
  5. 类属性中被static所引用的变量，会被作为GC的root根节点。作为根节点就意味着，这一类变量是基本上不会被回收的。因此，static很容易引入内存泄漏的风险。

### Final

1. 当用final修饰一个类时，表明这个类不能被继承。
2. 使用final把方法锁定，以防任何继承类修改它的含义；
3. 对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
4. 必须在定义时或者构造器中进行初始化赋值，而且final变量一旦被初始化赋值之后，就不能再被赋值了。

**Java中被final修饰的内置类：**

public final class Byte

public final class Character

public static final class Character.UnicodeBlock

public final class Class<T>

public final class Compile

public final class Double

public final class Float

public final class Integer

public final class Long

public final class Math

public final class ProcessBuilder

public final class RuntimePermission

public final class Short

public final class StackTraceElement

public final class StrictMath

public final class String

public final class StringBuffer

public final class StringBuilder

public final class System

public final class Void

### Transient

1. 一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。
2. transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。
3. 被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

### Volatile

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* + 1. 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
    2. 禁止进行指令重排序。
    3. 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
    4. 在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

### 并发编程的三大特性

1. 原子性

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

1. 可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

1. 有序性

　　在Java内存模型中，允许编译器和处理器对指令进行重排序，但是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。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性.

### 重写和重载

#### 重写

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。即外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。

##### 方法的重写规则

* 参数列表必须完全与被重写方法的相同；
* 返回类型必须完全与被重写方法的返回类型相同；
* 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。
* 父类的成员方法只能被它的子类重写。
* 声明为final的方法不能被重写。
* 声明为static的方法不能被重写，但是能够被再次声明。
* 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。
* 子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。
* 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
* 构造方法不能被重写。
* 如果不能继承一个方法，则不能重写这个方法。

**当需要在子类中调用父类的被重写方法时，要使用super关键字。**

* class Animal{  
   public void move(){  
   System.*out*.println("动物可以移动");  
   }  
  }  
    
  class Dog extends Animal{  
   public void move(){  
   super.move(); // 应用super类的方法  
   System.*out*.println("狗可以跑和走");  
   }  
  }  
    
  public class TestDog{  
   public static void main(String args[]){  
    
   Animal b = new Dog(); // Dog 对象  
   b.move(); //执行 Dog类的方法  
    
   }  
  }

**在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：**

**TestDog.java 文件代码：**

class Animal{  
 public void move(){  
 System.*out*.println("动物可以移动");  
 }  
}  
  
class Dog extends Animal{  
 public void move(){  
 System.*out*.println("狗可以跑和走");  
 }  
}  
  
public class TestDog{  
 public static void main(String args[]){  
 Animal a = new Animal(); // Animal 对象  
 Animal b = new Dog(); // Dog 对象  
  
 a.move();// 执行 Animal 类的方法  
  
 b.move();//执行 Dog 类的方法  
 }  
}

以上实例编译运行结果如下：

动物可以移动

狗可以跑和走

在上面的例子中可以看到，尽管b属于Animal类型，但是它运行的是Dog类的move方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为Animal类中存在move方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

**TestDog.java 文件代码：**

class Animal{  
 public void move(){  
 System.*out*.println("动物可以移动");  
 }  
}  
  
class Dog extends Animal{  
 public void move(){  
 System.*out*.println("狗可以跑和走");  
 }  
 public void bark(){  
 System.*out*.println("狗可以吠叫");  
 }  
}  
  
public class TestDog{  
 public static void main(String args[]){  
 Animal a = new Animal(); // Animal 对象  
 Animal b = new Dog(); // Dog 对象  
  
 a.move();// 执行 Animal 类的方法  
 b.move();//执行 Dog 类的方法  
 b.bark();  
 }  
}

以上实例编译运行结果如下：

TestDog.java:30: cannot find symbol

symbol : method bark()

location: class Animal

b.bark();

^

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

#### 重载

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

##### 重载规则:

* 被重载的方法必须改变参数列表(参数个数或类型不一样)；
* 被重载的方法可以改变返回类型；
* 被重载的方法可以改变访问修饰符；
* 被重载的方法可以声明新的或更广的检查异常；
* 方法能够在同一个类中或者在一个子类中被重载。
* 无法以返回值类型作为重载函数的区分标准。
* public class Overloading {  
   public int test(){  
   System.*out*.println("test1");  
   return 1;  
   }  
    
   public void test(int a){  
   System.*out*.println("test2");  
   }  
    
   //以下两个参数类型顺序不同  
   public String test(int a,String s){  
   System.*out*.println("test3");  
   return "returntest3";  
   }  
    
   public String test(String s,int a){  
   System.*out*.println("test4");  
   return "returntest4";  
   }  
    
   public static void main(String[] args){  
   Overloading o = new Overloading();  
   System.*out*.println(o.test());  
   o.test(1);  
   System.*out*.println(o.test(1,"test3"));  
   System.*out*.println(o.test("test4",1));  
   }  
  }

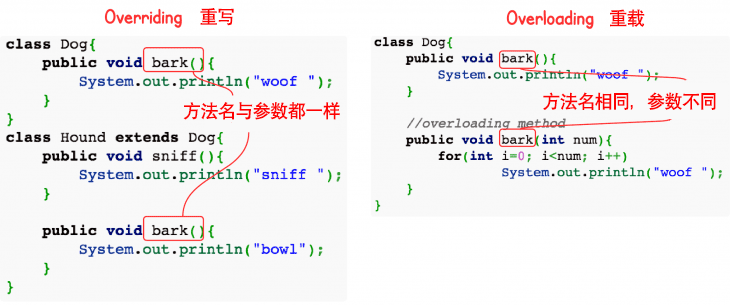
#### 重写与重载之间的区别

|  |  |  |
| --- | --- | --- |
| **区别点** | **重载方法** | **重写方法** |
| 参数列表 | 必须修改 | 一定不能修改 |
| 返回类型 | 可以修改 | 一定不能修改 |
| 异常 | 可以修改 | 可以减少或删除，一定不能抛出新的或者更广的异常 |
| 访问 | 可以修改 | 一定不能做更严格的限制（可以降低限制） |

**总结**

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

* (1)方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。
* (2)方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。
* (3)方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。



### Interface与abstract类

1. 含有abstract修饰符的class即为抽象类，abstract 类不能创建类的实例对象。含有abstract方法的类必须定义为abstract class，abstract class类中的方法不必是抽象的, 但是有抽象方法的类必定是抽象类。abstract class类中定义抽象方法必须在具体(Concrete)子类中实现，所以，不能有抽象构造方法或抽象静态方法**。如果的子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为abstract类型。**
2. 接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象、**default或者static方法**。接口中的方法定义默认为public abstract类型，接口中的成员变量类型默认为public static final。**一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。**另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。
3. **下面比较一下两者的语法区别：**
4. 抽象类可以有构造方法，接口中不能有构造方法。
5. 抽象类中可以有普通成员变量，接口中没有普通成员变量
6. 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。
7. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。
8. 抽象类中可以包含静态方法，接口中不能包含静态方法
9. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。
10. 一个类可以实现多个接口，但只能继承一个抽象类。

### 数据类型的默认值和长度

#### 数据类型分类

基本数据类型（或叫做原生类、内置类型）8种：

整数：byte，short，int，long（默认是int类型）

浮点类型： float，double（默认是double类型， 在定义 float 类型时必须在数字后面跟上 F 或者 f，long同理）

字符类型：char

布尔类型：boolean

引用数据类型3种：数组，类，接口

Java中的byte，short，char进行计算时都会提升为int类型。

#### 默认值

1. 布尔型(boolean)变量默认值为false，byte、short、int、long为0，字符型为'\u0000'(空字符)，浮点型(float double)为0.0,引用类型(String)为null。
2. 注意：未初始化的局部变量是不可以使用的，在这里可以认为是因为局部变量没有默认值，所以不可以直接使用。空字符('\u0000')什么也不输出，不要认为输出是空格。

#### 长度

32位的系统，则一个字是4个字节，如果是64位，则是8个字节。

1字节（byte） = 8位（bit）

基本类型：byte 二进制位数：8

包装类：java.lang.Byte

最小值：Byte.MIN\_VALUE=-128

最大值：Byte.MAX\_VALUE=127

基本类型：short 二进制位数：16

包装类：java.lang.Short

最小值：Short.MIN\_VALUE=-32768

最大值：Short.MAX\_VALUE=32767

基本类型：int 二进制位数：32

包装类：java.lang.Integer

最小值：Integer.MIN\_VALUE=-2147483648

最大值：Integer.MAX\_VALUE=2147483647

基本类型：long 二进制位数：64

包装类：java.lang.Long

最小值：Long.MIN\_VALUE=-9223372036854775808

最大值：Long.MAX\_VALUE=9223372036854775807

基本类型：float 二进制位数：32

包装类：java.lang.Float

最小值：Float.MIN\_VALUE=1.4E-45

最大值：Float.MAX\_VALUE=3.4028235E38

基本类型：double 二进制位数：64

包装类：java.lang.Double

最小值：Double.MIN\_VALUE=4.9E-324

最大值：Double.MAX\_VALUE=1.7976931348623157E308

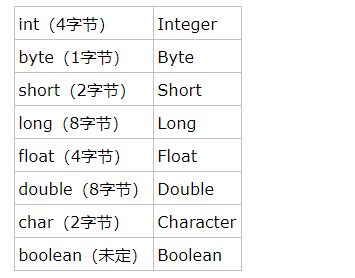
基本类型：char 二进制位数：16

包装类：java.lang.Character

最小值：Character.MIN\_VALUE=0

最大值：Character.MAX\_VALUE=65535

#### 包装器类型

装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。（xxx代表对应的基本数据类型）。

主动：Integer i = new Integer(10); //装箱

int i = a.intValue();//拆箱

自动：Integer i = 10; //装箱

int n = i; //拆箱

### 补码

1. 求**负整数的补码**，将其对应正数二进制表示所有位取反（包括符号位，0变1，1变0）后加1，-5对应正数5（00000101）→所有位取反（11111010）→加1(11111011)。

如果补码的符号位为“0”，表示是一个正数，其原码就是补码。

如果补码的符号位为“1”，表示是一个负数，那么求给定的这个补码的补码就是要求的原码。

小数补码求法，符号位保持1不变，数值位从右边数第一个1及其右边的0保持不变，左边按位取反。-0.0101为负数，补码为1.1011。

1. **补码加法**

[X+Y]补 = [X]补 + [Y]补

【例】X=+0110011,Y=-0101001，求[X+Y]补

[X]补=00110011 [Y]补=11010111

[X+Y]补 = [X]补 + [Y]补 = 00110011+11010111=00001010

注：因为计算机中运算器的位长是固定的（定长运算），上述运算中产生的最高位进位将丢掉，所以结果不是100001010，而是00001010。

1. **补码减法**

[X-Y]补 = [X]补 - [Y]补 = [X]补 + [-Y]补。

【例】1-1 [十进制]

1的原码00000001 转换成补码：00000001

-1的原码10000001 转换成补码：11111111

1+（-1)=0

00000001+11111111=00000000

00000000转换成十进制为0

0=0所以运算正确。

【例】-7-（-10) [十进制]

改为加法形式：-7-（-10）=-7+（-（-10））

-7的补码：11111001

-（-10）的补码：-10的原码为10001010，-（-10）的原码为00001010，

-（-10）的补码就是其原码，为00001010

-7 - （-10)= -7 + 10 = 3

11111001+00001010 = 00000011

转换成十进制为3

## 多线程

### 状态

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

下图显示了一个线程完整的生命周期。



* **新建状态:**

使用 **new** 关键字和 **Thread** 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 **start()** 这个线程。

* **就绪状态:**

当线程对象调用了start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待JVM里线程调度器的调度。

* **运行状态:**

如果就绪状态的线程获取 CPU 资源，就可以执行 **run()**，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

* **阻塞状态:**

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：

* + 等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。
  + 同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
  + 其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。
* **死亡状态:**

一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

### 创建线程的方法

**Java 提供了三种创建线程的方法：**

#### 通过实现 Runnable 接口

创建一个线程，最简单的方法是创建一个实现 Runnable 接口的类。

为了实现 Runnable，一个类只需要执行一个方法调用 run()，声明如下：

public void run()

你可以重写该方法，重要的是理解的 run() 可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现 Runnable 接口的类之后，你可以在类中实例化一个线程对象。

Thread 定义了几个构造方法，下面的这个是我们经常使用的：

Thread(Runnable threadOb,String threadName);

新线程创建之后，你调用它的 start() 方法它才会运行。

void start();

class TestForRunnable implements Runnable {  
 private int k;  
 //多线程,implements Runnable  
  
 */\*\*  
 \*   
 \*/* public void run() {  
 for (; k < 100; k++) {  
 System.*out*.println(Thread.*currentThread*().getName()  
 + " " + k);  
 }  
 }  
 */\*\*  
 \*   
 \*/* public void test() {  
 //Runnable  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " " + i);  
 if (20 == i) {  
 try {  
 //共享k  
 TestForRunnable a = new TestForRunnable();  
// new Thread(a, "新线程1").start();  
// new Thread(a, "新线程2").start();  
 Thread t1 = new Thread(a, "新线程1");  
 Thread t2 = new Thread(a, "新线程2");  
 t2.start();  
 t1.start();  
 t1.join();  
 t2.*sleep*(1);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

#### 通过继承 Thread 类本身

创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。

继承类必须重写 run() 方法，该方法是新线程的入口点。它也必须调用 start() 方法才能执行。

该方法尽管被列为一种多线程实现方式，但是本质上也是实现了 Runnable 接口的一个实例。

class TestForThread extends Thread {  
 private int k;  
 */\*\*  
 \*  
 \** ***@param*** *name  
 \*/* public TestForThread(String name) {  
 super(name);  
 }  
  
 */\*\*  
 \*  
 \*/* public void run() {  
 try {  
 for (; k < 100; k++) {  
 System.*out*.println(getName() + " " + k);  
 //有sleep()抛出InterruptedException  
 *sleep*(10);  
 if (20 == k) {  
 //转入就绪状态，给同级或者更高等级线程机会  
 Thread.*yield*();  
 }  
 }  
  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }

**下表列出了Thread类的一些重要方法：**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public void start()** 使该线程开始执行；**Java** 虚拟机调用该线程的 run 方法。 |
| 2 | **public void run()** 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。 |
| 3 | **public final void setName(String name)** 改变线程名称，使之与参数 name 相同。 |
| 4 | **public final void setPriority(int priority)**  更改线程的优先级。 |
| 5 | **public final void setDaemon(boolean on)** 将该线程标记为守护线程或用户线程。 |
| 6 | **public final void join(long millisec)** 等待该线程终止的时间最长为 millis 毫秒。 |
| 7 | **public void interrupt()** 中断线程。 |
| 8 | **public final boolean isAlive()** 测试线程是否处于活动状态。 |

**测试线程是否处于活动状态。 上述方法是被Thread对象调用的。下面的方法是Thread类的静态方法。**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。 |
| 2 | **public static void sleep(long millisec)** 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |
| 3 | **public static boolean holdsLock(Object x)** 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。 |
| 4 | **public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。 |
| 5 | **public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。 |

#### 通过 Callable 和 Future 创建线程。

* 1. 创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。
* 2. 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值。
* 3. 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
* 4. 调用 FutureTask 对象的 get() 方法来获得子线程执行结束后的返回值。
* class CallableAndFutureTask {  
   */\*\*  
   \*   
   \*/* public void test() {  
   //多线程Callable，FutureTask  
   FutureTask<Integer> task = new FutureTask<Integer>((Callable<Integer>)()-> {  
   //call()  
   int i = 0;  
   for (; i < 100; i++) {  
   System.*out*.println(Thread.*currentThread*().getName() + "的循环变量i的值：" + i);  
   }  
   return i;  
   });  
   for (int i = 0; i<100; i++) {  
   System.*out*.println(Thread.*currentThread*().getName() + "的循环变量i的值：" + i);  
   if (20 == i) {  
   new Thread(task,"有返回值的线程").start();  
   }  
   }  
   try {  
   System.*out*.println("子线程的返回值：" + task.get());  
   } catch (Exception e) {  
   e.printStackTrace();  
   }  
   }  
  }

#### 创建线程的三种方式的对比

* 1. 采用实现 Runnable、Callable 接口的方式创建多线程时，线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。
* 2. 使用继承 Thread 类的方式创建多线程时，编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。

#### 线程池

##### 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。（是什么）

##### 那么，我们为什么需要用到线程池呢？每次用的时候手动创建不行吗？

在java中，如果每个请求到达就创建一个新线程，开销是相当大的。在实际使用中，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务。（为什么）

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使用应用程序响应更快；另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。（什么用）

##### 线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。

* **Executors**：线程池创建工厂类
* **public static ExecutorServicenewFixedThreadPool(int nThreads)**：返回线程池对象
* **ExecutorService**：线程池类
* **Future<?> submit(Runnable task)**：获取线程池中的某一个线程对象，并执行
* **Future 接口**：用来记录线程任务执行完毕后产生的结果。线程池创建与使用

##### 这里介绍两种使用线程池创建线程的方法

###### **使用Runnable接口创建线程池**

使用线程池中线程对象的步骤：

* 1、创建线程池对象
* 2、创建 Runnable 接口子类对象
* 3、提交 Runnable 接口子类对象
* 4、关闭线程池

**Test.java** 代码如下：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test {

public static void main(String[] args) {

//创建线程池对象 参数5，代表有5个线程的线程池

ExecutorService service = Executors.newFixedThreadPool(5);

//创建Runnable线程任务对象

TaskRunnable task = new TaskRunnable();

//从线程池中获取线程对象

service.submit(task);

System.out.println("----------------------");

//再获取一个线程对象

service.submit(task);

//关闭线程池

service.shutdown();

}

}

TaskRunnable.java 接口文件如下：

public class TaskRunnable implements Runnable{

@Override

public void run() {

for (int i = 0; i < 1000; i++) {

System.out.println("自定义线程任务在执行"+i);

}

}

}

###### 使用Callable接口创建线程池

Callable接口：与Runnable接口功能相似，用来指定线程的任务。其中的call()方法，用来返回线程任务执行完毕后的结果，call方法可抛出异常。

**ExecutorService**：线程池类

**<T> Future<T> submit(Callable<T> task)**：获取线程池中的某一个线程对象，并执行线程中的 call() 方法

**Future 接口**：用来记录线程任务执行完毕后产生的结果。线程池创建与使用

使用线程池中线程对象的步骤：

* 1、创建线程池对象
* 2、创建 Callable 接口子类对象
* 3、提交 Callable 接口子类对象
* 4、关闭线程池

Test.java 代码如下：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test{

public static void main(String[] args) {

ExecutorService service = Executors.newFixedThreadPool(3);

TaskCallable c = new TaskCallable();

//线程池中获取线程对象，调用run方法

service.submit(c);

//再获取一个

service.submit(c);

//关闭线程池

service.shutdown();

}

}

TaskCallable.java 接口文件如下：

import java.util.concurrent.Callable;

public class TaskCallable implements Callable<Object>{

@Override

public Object call() throws Exception {

for (int i = 0; i < 1000; i++) {

System.out.println("自定义线程任务在执行"+i);

}

return null;

}

}

### 线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。

Java 线程的优先级是一个整数，其取值范围是 1 （Thread.MIN\_PRIORITY ） - 10 （Thread.MAX\_PRIORITY ）。

默认情况下，每一个线程都会分配一个优先级 NORM\_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

### JAVA多线程和并发基础面试问答

https://www.cnblogs.com/dolphin0520/p/3932934.html

#### 进程和线程之间有什么不同？

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。Java运行环境是一个包含了不同的类和程序的单一进程。线程可以被称为轻量级进程。线程需要较少的资源来创建和驻留在进程中，并且可以共享进程中的资源。

#### 多线程编程的好处是什么？

在多线程程序中，多个线程被并发的执行以提高程序的效率，CPU不会因为某个线程需要等待资源而进入空闲状态。多个线程共享堆内存(heap memory)，因此创建多个线程去执行一些任务会比创建多个进程更好。举个例子，Servlets比CGI更好，是因为Servlets支持多线程而CGI不支持。

#### 用户线程和守护线程有什么区别？

当我们在Java程序中创建一个线程，它就被称为用户线程。一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

#### 我们如何创建一个线程？

有两种创建线程的方法：一是实现Runnable接口，然后将它传递给Thread的构造函数，创建一个Thread对象；二是直接继承Thread类。若想了解更多可以阅读这篇关于如何在[Java中创建线程](http://www.journaldev.com/1016/java-thread-example-extending-thread-class-and-implementing-runnable-interface)的文章。

#### 有哪些不同的线程生命周期？

当我们在Java程序中新建一个线程时，它的状态是New。当我们调用线程的start()方法时，状态被改变为Runnable。线程调度器会为Runnable线程池中的线程分配CPU时间并且讲它们的状态改变为Running。其他的线程状态还有Waiting，Blocked 和Dead。读这篇文章可以了解更多关于[线程生命周期](http://www.journaldev.com/1044/life-cycle-of-thread-understanding-thread-states-in-java)的知识。

#### 可以直接调用Thread类的run()方法么？

当然可以，但是如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

#### 如何让正在运行的线程暂停一段时间？

我们可以使用Thread类的Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为Runnable，并且根据线程调度，它将得到执行。

#### 你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

#### 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。线程调度并不受到Java虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

#### 在多线程中，什么是上下文切换(context-switching)？

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

#### 你如何确保main()方法所在的线程是Java程序最后结束的线程？

我们可以使用Thread类的joint()方法来确保所有程序创建的线程在main()方法退出前结束。这里有一篇文章关于[Thread类的joint()方法](http://www.journaldev.com/1024/java-thread-join-example-with-explanation)。

#### 线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。点击[这里](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example)有更多关于线程wait, notify和notifyAll.

#### 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？

Java的每个对象中都有一个锁(monitor，也可以成为监视器) 并且wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法

#### 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？

当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

#### 为什么Thread类的sleep()和yield()方法是静态的？

Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

#### 如何确保线程安全？

在Java中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用volatile关键字，使用不变类和线程安全类。在[线程安全教程](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples)中，你可以学到更多。

#### volatile关键字在Java中有什么作用？

当我们使用volatile关键字去修饰变量的时候，所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

#### 同步方法和同步块，哪个是更好的选择？

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

#### 如何创建守护线程？

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

#### 什么是ThreadLocal?

ThreadLocal用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。

每个线程都会拥有他们自己的Thread变量，它们可以使用get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望它们同线程状态关联起来是private static属性。在[ThreadLocal例子](http://www.journaldev.com/1076/java-threadlocal-example-to-create-thread-local-variables)这篇文章中你可以看到一个关于ThreadLocal的小程序。

#### 什么是Thread Group？为什么不建议使用它？

ThreadGroup是一个类，它的目的是提供关于线程组的信息。

ThreadGroup API比较薄弱，它并没有比Thread提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaught exception handler)。但在Java 1.5中Thread类也添加了setUncaughtExceptionHandler(UncaughtExceptionHandler eh) 方法，所以ThreadGroup是已经过时的，不建议继续使用。

#### 什么是Java线程转储(Thread Dump)，如何得到它？

线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获取线程转储——使用Profiler，Kill -3命令，jstack工具等等。我更喜欢jstack工具，因为它容易使用并且是JDK自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。读这篇文档可以了解更多关于[产生线程转储](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java)的知识。

#### 什么是死锁(Deadlock)？如何分析和避免死锁？

死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

分析死锁，我们需要查看Java应用程序的线程转储。我们需要找出那些状态为BLOCKED的线程和他们等待的资源。每个资源都有一个唯一的id，用这个id我们可以找出哪些线程已经拥有了它的对象锁。

避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法，阅读这篇文章去学习[如何分析死锁](http://www.journaldev.com/1058/java-deadlock-example-and-how-to-analyze-deadlock-situation)。

#### 什么是Java Timer类？如何创建一个有特定时间间隔的任务？

java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

这里有关于[java Timer的例子](http://www.journaldev.com/1050/java-timer-and-timertask-example-tutorial)。

#### 什么是线程池？如何创建一个Java线程池？

一个线程池管理了一组工作线程，同时它还包括了一个用于放置等待执行的任务的队列。

java.util.concurrent.Executors提供了一个 java.util.concurrent.Executor接口的实现用于创建线程池。[线程池例子](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)展现了如何创建和使用线程池，或者阅读[ScheduledThreadPoolExecutor](http://www.journaldev.com/2340/java-scheduledthreadpoolexecutor-example-to-schedule-tasks-after-delay-and-execute-periodically)例子，了解如何创建一个周期任务。

### Java并发面试问题

#### 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。可以阅读这篇文章来了解[Java的atomic类](http://www.journaldev.com/1095/java-atomic-operations-atomicinteger-example)。

#### Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

* 可以使锁更公平
* 可以使线程在等待锁的时候响应中断
* 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间
* 可以在不同的范围，以不同的顺序获取和释放锁

阅读更多[关于锁的例子](http://www.journaldev.com/2377/java-lock-example-and-concurrency-lock-vs-synchronized)

#### 什么是Executors框架？

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池，阅读这篇文章可以了解[如何使用Executor框架创建一个线程池](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)。

#### 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

阅读这篇文章了解[如何使用阻塞队列实现生产者-消费者问题。](http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem)

#### 什么是Callable和Future?

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

阅读这篇文章了解更多[关于Callable，Future的例子](http://www.journaldev.com/1090/java-callable-future-example)。

#### 什么是FutureTask?

FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读[Java FutureTask例子](http://www.journaldev.com/1650/java-futuretask-example-program)，学习如何使用它。

#### 什么是并发容器的实现？

Java集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的next()方法将抛出ConcurrentModificationException异常。

并发容器支持并发的遍历和并发的更新。

主要的类有ConcurrentHashMap, CopyOnWriteArrayList 和CopyOnWriteArraySet，阅读这篇文章了解[如何避免ConcurrentModificationException](http://www.journaldev.com/378/how-to-avoid-concurrentmodificationexception-when-using-an-iterator)。

#### Executors类是什么？

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法。

Executors可以用于方便的创建线程池

#### 为什么说ConcurrentHashMap是弱一致性的？以及为何多个线程并发修改ConcurrentHashMap时不会报ConcurrentModificationException？

　　参考：<http://ifeve.com/java-concurrent-hashmap-2/>

<http://ifeve.com/concurrenthashmap-weakly-consistent/>

<http://blog.csdn.net/liuzhengkang/article/details/2916620>

## 异常

### 异常类型

1. **unchecked exceptions（运行时异常）：**都是RuntimeException类及其子类异常，就是我们在开发中测试功能时程序终止，控制台出现的异常，比如：   
   NullPointerException(空指针异常)、   
   IndexOutOfBoundsException(下标越界异常)   
   ClassCastException(类转换异常)   
   ArrayStoreException(数据存储异常，操作数组时类型不一致)   
   IO操作的BufferOverflowException异常
2. **checked exceptions，非运行时异常 （编译异常）**：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。   
   通俗的话说就是在写代码时出现红线，需要try catch或者throws时出现的异常。

### finally关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句

### Java内置异常

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

|  |  |
| --- | --- |
| **异常** | **描述** |
| ArithmeticException | 当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。 |
| ArrayIndexOutOfBoundsException | 用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。 |
| ArrayStoreException | 试图将错误类型的对象存储到一个对象数组时抛出的异常。 |
| ClassCastException | 当试图将对象强制转换为不是实例的子类时，抛出该异常。 |
| IllegalArgumentException | 抛出的异常表明向方法传递了一个不合法或不正确的参数。 |
| IllegalMonitorStateException | 抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。 |
| IllegalStateException | 在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。 |
| IllegalThreadStateException | 线程没有处于请求操作所要求的适当状态时抛出的异常。 |
| IndexOutOfBoundsException | 指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。 |
| NegativeArraySizeException | 如果应用程序试图创建大小为负的数组，则抛出该异常。 |
| NullPointerException | 当应用程序试图在需要对象的地方使用 null 时，抛出该异常 |
| NumberFormatException | 当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。 |
| SecurityException | 由安全管理器抛出的异常，指示存在安全侵犯。 |
| StringIndexOutOfBoundsException | 此异常由 String 方法抛出，指示索引或者为负，或者超出字符串的大小。 |
| UnsupportedOperationException | 当不支持请求的操作时，抛出该异常。 |

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

|  |  |
| --- | --- |
| **异常** | **描述** |
| ClassNotFoundException | 应用程序试图加载类时，找不到相应的类，抛出该异常。 |
| CloneNotSupportedException | 当调用 Object 类中的 clone 方法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。 |
| IllegalAccessException | 拒绝访问一个类的时候，抛出该异常。 |
| InstantiationException | 当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。 |
| InterruptedException | 一个线程被另一个线程中断，抛出该异常。 |
| NoSuchFieldException | 请求的变量不存在 |
| NoSuchMethodException | 请求的方法不存在 |

### 异常方法

下面的列表是 Throwable 类的主要方法:

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public String getMessage()** 返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。 |
| 2 | **public Throwable getCause()** 返回一个Throwable 对象代表异常原因。 |
| 3 | **public String toString()** 使用getMessage()的结果返回类的串级名字。 |
| 4 | **public void printStackTrace()** 打印toString()结果和栈层次到System.err，即错误输出流。 |
| 5 | **public StackTraceElement [] getStackTrace()** 返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。 |
| 6 | **public Throwable fillInStackTrace()** 用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。 |

### 使用throws抛出异常

如果在当前方法不知道该如何处理该异常时，则可以使用throws对异常进行抛出给调用者处理或者交给JVM。JVM对异常的处理方式是：打印异常的跟踪栈信息并终止程序运行。   
throws在使用时应处于方法签名之后使用，可以抛出多种异常并用英文字符逗号’,’隔开   
e.g.1   
public void throwsTest() throws ExceptionClass1, ExceptionClass2 {...}   
如果抛出给调用者的异常是Checked异常，这种异常是我们需要处理以来提高程序健壮性的，一般抛出则要调用者做相应处理，要么调用者对该异常进行try…catch处理，要么再次throws交给上一层。这其中需要注意一点：子类方法声明抛出的异常类型应是父类方法声明抛出的异常类型的子类或相同，子类方法声明跑出的异常不允许比父类方法声明抛出的异常多。   
e.g.2

public class ThrowsTest2{

//因为test();会抛出IOException，main方法是调用者

//则需要使用try...catch进行捕获或者抛给JVM

//抛出时要遵循父类声明抛出“大于”子类声明抛出的原则

public static void main(String[] args) throws Exception {

test();

}

public static void test() throws IOException() {

//因为FileInputStream构造器会抛出IOException

//所以需要使用try...catch块进行处理或使用throws抛给调用者

FileInputStream fis = new FileInputStream("a.txt");

}

}

### 使用throw抛出异常

如果需要程序在程序中自行抛出异常，应该使用throw语句抛出，抛出的不是一个类而是一个对象且只能抛出一个对象。它可以单独使用，也可以结合catch块捕获使用。如果抛出的异常对象时Checked异常则处于try块里被catch捕获或者放在一个带throws的方法里；如果抛出的是RuntimeException则既可以显示使用try…catch捕获也可以不理会它   
e.g.3

public class ThrowTest {

public static void main(String[] args) {

try{

throwChecked(3);

}catch(Exception e) {

System.out.println(e.getMessage());

}

throwRuntime(-3);

}

//该方法内抛出一个Exception异常对象，必须捕获或抛给调用者

public static void throwChecked(int a) throws Exception {

if(a < 0) {

throw new Exception("a的值应大于0，不符合要求")

}

}

//该方法内抛出一个RuntimeException对象，可以不理会直接交给JVM处理

public static void throwRuntime(int a) {

if(a < 0) {

throw new RuntimeException("a的值应大于0，不符合要求")

}

}

}

Java7增强的throw语句：   
在Java7之前，父类和子类在声明抛出异常时应符合父类包含的异常“大于等于”子类包含的异常的规则；从Java7开始，Java编译器会检查throw语句抛出的异常的实际类型，如下代码中编译器知道throw e只能抛出FileNotFoundException 所以在方法签名上可以直接写该异常：   
e.g.4

public class ThrowTest2 {

public static void main(String[] args) throws FileNotFoundException {

try {

new FileOutputStream("a.txt");

}catch(Exception e) {

e.printStackTrace();

throw e;

}

}

}

### 自定义异常类

在抛出异常时，异常类名往往包含有用的信息，所以在选择抛出异常时需要选择适合的类，从而可以明确的描述该异常情况。这时候就需要我们自己定义异常，自定义异常一定是Throwable的子类，若是检查异常就要继承自Exception，若是运行时异常就要继承自RuntimeException   
e.g.5

public class AuctionException extends Exception {

//无参构造

public AuctionException() {}

//含参构造

//通过调用父类的构造器将字符串msg传给异常对象的massage属性，

//massage属性就是对异常的描述

public AuctionException(String msg) {

super(msg);

}

}

关于自定义异常深入学习可以参考以下链接：   
<http://www.cnblogs.com/AlanLee/p/6104492.html>

### catch和throw同时使用

在实际应用中，在异常出现的当前方法中，程序只能对异常做部分处理，还有些处理需要在该方法的调用者才能够完成，使用需要再次抛出异常。这时，就需要将catch和throw结合使用。   
e.g.6

public class AuctionTest {

private double initPrice = 30.0;

public void bid(String bidPrice) throws AuctionException {

double d = 0.0;

try{

d = Double.parseDouble(bidPrice);

}catch(Exception e) {

e.printStackTrace();

throw new AuctionException("竞拍价必须为数值，不能包含其它数值");

}

if(initPrice > d) {

throw new AuctionException("竞拍价应比起拍价高");

}

initPrice = d;

}

public static void main(String[] args) {

AuctionTest at = new AuctionTest();

try{

at.bid("..");

}catch(AuctionException ae) {

System.out.println(ae.getMessage());

}

}

}

以上代码的运行结果为：

竞拍价必须为数值，不能包含其它数值

java.lang.NumberFormatException: multiple points

at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)

at sun.misc.FloatingDecimal.parseDouble(Unknown Source)

at java.lang.Double.parseDouble(Unknown Source)

at ExceptionTest.AuctionTest.bid(AuctionTest.java:15)

at ExceptionTest.AuctionTest.main(AuctionTest.java:28)

### 异常链

把捕获一个异常然后接着抛出另一个异常，并把原始异常信息保存下来是一种典型的链式处理，也被称作异常链。

### Java的异常跟踪栈

如e.g.6的运行结果   
除去第一行打印的自定义的提示信息之外的第一行是异常的类型和详细消息，之后的提示信息中含有确切代码行数的信息记录了所有的异常发生点，各行显示被调用方法中执行的停止位置。跟踪栈总是最内部的被调用方法逐渐上传知道外部业务操作的起点，通常就是main方法或者Thread的run方法。

### 异常处理规则

1. 不要过度使用异常：对于完全已知的错误应编写处理这种错误代码从而提高代码的健壮性，只有外部的、不能确定的和不可预知的运行时错误使是用异常，并且异常机制的效率低于正常的流程控制。
2. 不要使用过于庞大的try块：过于庞大的try块业务也相对更复杂，会导致try块中异常的可能性大大增加，在分析发生异常的原因时难度增加。
3. 避免使用Catch All语句：Catch All是catch(Throwable t)，也会在发生异常是分析原因的复杂度增加。
4. 不要忽略已捕获到的异常：对于捕获到的异常应该对其进行处理从而提高代码健壮性，而不是什么都不做或者只是打印跟踪栈信息。

--------------------- 本文来自 sdr\_zd 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/sdr\_zd/article/details/75675004?utm\_source=copy

## 设计模式

**【单例模式 工厂模式   策略模式  适配器模式  生产者消费者模式 观察者模式】**

### ****单例模式 Singleton****

**如果一个类始终只能创建一个实例，则这个类被称为单例类，这种模式就被称为单例模式。**

**一般建议单例模式的方法命名为：getInstance()，可以有参数，一般无参数。**

**优点：**

1)减少创建[Java](http://lib.csdn.net/base/javase)实例所带来的系统开销

2)便于系统跟踪单个Java实例的生命周期、实例状态等。

### ****简单工厂模式 StaticFactory Method****

**由一个工厂对象决定创建出哪一种产品类的实例。**

**优点：**

**让对象的调用者和对象创建过程分离，当对象调用者需要对象时，直接向工厂请求即可。**

### ****工厂模式 Factory Method****

**程序可以为不同产品类提供不同的工厂，不同的工厂类生产不同的产品（也就是一个工厂生产一个产品，通过选择不同的工厂获得不同的产品。）**

### ****抽象工厂模式 Abstract Factory Method****

**为了解决客户端代码与不同工厂类耦合的问题。在工厂类的基础上再增加一个工厂类，该工厂类不制造具体的被调用对象，而是制造不同工厂对象。（建立一个生产工厂的工厂，把原来工厂模式的工厂作为方法）**

### ****代理模式 Proxy Method****

**当客户端代码需要调用某个对象时，客户端实际上不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时我们就可返回该对象的代理（Proxy）。**

### ****命令模式 Command Method****

**某个方法需要完成某一个功能，完成这个功能的大部分步骤已经确定了，但可能有少量具体步骤无法确定，必须等到执行该方法时才可以确定。在Java中，传入该方法的是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式。**

### ****策略模式 Strategy Method****

**策略模式用于封装系列的**[算法](http://lib.csdn.net/base/datastructure)**，这些算法通常被封装在一个被称为Context的类中，客户端程序可以自由选择其中一种算法，或让Context为客户端选择一种最佳算法——使用策略模式的优势是为了支持算法的自由切换。**

**可以考虑使用配置文件来指定用户使用哪种具体算法——这就彻底分离客户端代码和具体算法策略。**

### ****观察者模式 Observer Method****

**观察者模式定义了对象间的一对多依赖关系，让一个或多个观察者对象观察一个主题对象。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新。**

**在观察者模式中，被观察的对象常常也被称为目标或主题（Subject），依赖的对象被称为观察者（Observer）。**

### ****生产者消费者模式 ProducerConsumer Method****

**生产者和消费者在同一时间段内共用同一存储空间，生产者向空间（仓库/缓冲区）里生产数据，而消费者取走数据。**

**优点：**

1)解耦

2)支持并发

生产者与消费者是两个独立的并发体，他们之间是用缓冲区作为桥梁连接。

3)支持忙闲不均

当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。 等生产者的制造速度慢下来，消费者再慢慢处理掉。

--------------------- 本文来自 cyhaolihai 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/cyhaolihai/article/details/82902368?utm_source=copy>

## Java内存模型

<https://www.cnblogs.com/andy-zcx/p/5522836.html>

### 简介

1. 堆

堆是所有线程共享的，主要用来存储对象。其中，堆可分为：年轻代和老年代两块区域。使用NewRatio参数来设定比例。对于年轻代，一个Eden区和两个Suvivor区，使用参数SuvivorRatio来设定大小；

1. Java虚拟机栈/本地方法栈

线程私有的，主要存放局部变量表，操作数栈，动态链接和方法出口等；

1. 程序计数器

同样是线程私有的，记录当前线程的行号指示器，为线程的切换提供保障；

1. 方法区

线程共享的，主要存储类信息、常量池、静态变量、JIT编译后的代码等数据。方法区理论上来说是堆的逻辑组成部分；

1. 运行时常量池

是方法区的一部分，用于存放编译期生成的各种字面量和符号引用；

### 堆、栈、方法区

JAVA的JVM的内存可分为3个区：堆(heap)、堆栈(stack)和方法区(method)

#### 堆区:

1. 提供所有类实例和数组对象存储区域
2. jvm只有一个堆区(heap)被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身
3. 堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

#### 栈区:

1. 每个线程包含一个栈区，栈中只保存基础数据类型的对象和自定义对象的引用(不是对象)，对象都存放在堆区中
2. 每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。
3. 在函数中定义的一些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。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。
4. **虚拟机栈（JVM Stack）：**

一个线程的每个方法在执行的同时，都会创建一个栈帧（Statck Frame），栈帧中存储的有局部变量表、操作站、动态链接、方法出口等，当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。

　　局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。在局部变量表中，只有long和double类型会占用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。

　　虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，直到内存不足，此时，会抛出OutOfMemoryError（内存溢出）。

　　每个线程对应着一个虚拟机栈，因此虚拟机栈也是线程私有的。

1. **本地方法栈（Native Method Statck）：**

本地方法栈在作用，运行机制，异常类型等方面都与虚拟机栈相同，唯一的区别是：虚拟机栈是执行Java方法的，而本地方法栈是用来执行**native方法**的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。

　　本地方法栈也是线程私有的。

#### 堆和栈的联系

1. 引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组＆对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，数组和对象在没有引用变量指向它的时候（比如先前的引用变量x=null时），才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因。

2. 总结起来就是对象存储在堆内存，引用变量存储在栈内存。栈内存指向堆内存。

#### 方法区:

1. 又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class和static变量。
2. 方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。
3. 运行时常量池都分配在 Java 虚拟机的方法区之中
4. 更加详细一点的说法是方法区里存放着类的版本，字段，方法，接口和常量池。[常量池](#_常量池)里存储着字面量和符号引用。



--------------------- 本文来自 zhuliyuan丶 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/zly921112/article/details/61192747?utm_source=copy>

**永久代和方法区的关系**

  涉及到内存模型时，往往会提到永久代，那么它和方法区又是什么关系呢？《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。 同时大多数用的JVM都是Sun公司的HotSpot。在HotSpot上把GC分代收集扩展至方法区，或者说**使用永久代来实现方法区**。因此，我们得到了结论，永久代是HotSpot的概念，方法区是Java虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现。其他的虚拟机实现并没有永久带这一说法。在1.7之前在(JDK1.2 ~ JDK6)的实现中，HotSpot 使用永久代实现方法区，HotSpot 使用 GC分代来实现方法区内存回收，可以使用如下参数来调节方法区的大小:

-XX:PermSize

方法区初始大小

-XX:MaxPermSize

方法区最大大小

超过这个值将会抛出OutOfMemoryError异常:java.lang.OutOfMemoryError: PermGen

**元空间**

  对于Java8， HotSpots取消了永久代，那么是不是也就没有方法区了呢？当然不是，方法区是一个规范，规范没变，它就一直在。那么取代永久代的就是元空间。它可永久代有什么不同的？存储位置不同，永久代物理是是堆的一部分，和新生代，老年代地址是连续的，而元空间属于**本地内存，不在java虚拟机中的**；存储内容不同，元空间存储类的元信息，静态变量和常量池等并入堆中。相当于永久代的数据被分到了堆和元空间中。

--------------------- 本文来自 抽离的心 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u011635492/article/details/81046174?utm\_source=copy

### 分代

#### **年轻代（Young Generation）**

1.所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2.新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复。

3.当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收

4.新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)

#### **年老代（Old Generation）**

1.在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。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。

2.内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

#### **持久代（Permanent Generation）**

用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

**注：**jdk1.8移除了永久代, 链接<http://openjdk.java.net/projects/jdk8/milestones>. 其中 “[JEP 122: Remove the Permanent Generation](http://openjdk.java.net/jeps/122)“说的就是移除永久代.

说明：类的元数据, 字符串池, 类的静态变量将会从永久代移除, 放入Java heap或者native memory. 其中建议JVM的实现中将类的元数据放入 native memory, 将字符串池和类的静态变量放入java堆中. 这样可以加载多少类的元数据就不在由MaxPermSize控制, 而由系统的实际可用空间来控制.

为什么这么做呢? 减少OOM只是表因, 更深层的原因还是要合并HotSpot和JRockit的代码, JRockit从来没有一个叫永久代的东西, 但是运行良好, 也不需要开发运维人员设置这么一个永久代的大小.

当然不用担心运行性能问题了, 在覆盖到的测试中, 程序启动和运行速度降低不超过1%, 但是这一点性能损失换来了更大的安全保障.

--------------------- 本文来自 老伯⭐️ 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/afterlife_qiye/article/details/52506976?utm_source=copy>

### 常量池

在Java的内存分配中，总共3种常量池：

#### ****1.字符串常量池(String Constant Pool):****

##### ****1.1:字符串常量池在Java内存区域的哪个位置？****

* 在JDK6.0及之前版本，字符串常量池是放在Perm Gen区(也就是方法区)中；
* 在JDK7.0版本，字符串常量池被移到了堆中了。至于为什么移到堆内，大概是由于方法区的内存空间太小了。

##### ****1.2:字符串常量池是什么？****

* 在HotSpot VM里实现的string pool功能的是一个StringTable类，它是一个Hash表，默认值大小长度是1009；这个StringTable在每个HotSpot VM的实例只有一份，被所有的类共享。字符串常量由一个一个字符组成，放在了StringTable上。
* 在JDK6.0中，StringTable的长度是固定的，长度就是1009，因此如果放入String Pool中的String非常多，就会造成hash冲突，导致链表过长，当调用String#intern()时会需要到链表上一个一个找，从而导致性能大幅度下降；
* 在JDK7.0中，StringTable的长度可以通过参数指定：

-XX:StringTableSize=66666

* 1

##### ****1.3:字符串常量池里放的是什么？****

* 在JDK6.0及之前版本中，String Pool里放的都是字符串常量；
* 在JDK7.0中，由于String#intern()发生了改变，因此String Pool中也可以存放放于堆内的字符串对象的引用。关于String在内存中的存储和String#intern()方法的说明，可以参考我的另外一篇博客：

需要说明的是：字符串常量池中的字符串只存在一份！   
如：

String s1 = "hello,world!";

String s2 = "hello,world!";

即执行完第一行代码后，常量池中已存在 “hello,world!”，那么 s2不会在常量池中申请新的空间，而是直接把已存在的字符串内存地址返回给s2。(这里具体的字符串如何分配就不细说了，可以看我的另一篇博客)

字符串常量池也是运行时常量池一部分

#### ****2.class常量池(Class Constant Pool)：****

##### ****2.1:class常量池简介：****

* 我们写的每一个Java类被编译后，就会形成一份class文件；class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)；
* 每个class文件都有一个class常量池。

##### ****2.2:什么是字面量和符号引用：****

* 字面量包括：1.文本字符串 2.八种基本类型的值 3.被声明为final的常量等;
* 符号引用包括：1.类和方法的全限定名 2.字段的名称和描述符 3.方法的名称和描述符。

#### ****3.运行时常量池(Runtime Constant Pool)：****

* 运行时常量池存在于内存中，也就是class常量池被加载到内存之后的版本，不同之处是：它的字面量可以动态的添加(String#intern()),符号引用可以被解析为直接引用
* JVM在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，jvm就会将class常量池中的内容存放到运行时常量池中，由此可知，运行时常量池也是每个类都有一个。在解析阶段，会把符号引用替换为直接引用，解析的过程会去查询字符串常量池，也就是我们上面所说的StringTable，以保证运行时常量池所引用的字符串与字符串常量池中是一致的。

--------------------- 本文来自 zhuminChosen 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/zm13007310400/article/details/77534349?utm\_source=copy

### 垃圾回收

#### Java垃圾回收概况

　　Java GC（Garbage Collection，垃圾回收）机制，是Java与C++/C的主要区别之一，作为Java开发者，一般不需要专门编写内存回收和垃圾清理代码，对内存泄露和溢出的问题，也不需要像C程序员那样战战兢兢。这是因为在Java虚拟机中，存在自动内存管理和垃圾清扫机制。概括地说，该机制对JVM中的内存进行标记，并确定哪些内存需要回收，根据一定的回收策略，自动的回收内存，永不停息的保证JVM中的内存空间，防止出现内存泄露和溢出问题。

　　关于JVM，需要说明一下的是，目前使用最多的Sun公司的JDK中，自从1999年的JDK1.2开始直至现在仍在广泛使用的JDK6，其中默认的虚拟机都是HotSpot。2009年，Oracle收购Sun，加上之前收购的EBA公司，Oracle拥有3大虚拟机中的两个：JRockit和HotSpot，Oracle也表明了想要整合两大虚拟机的意图，但是目前在新发布的JDK8中，默认的虚拟机仍然是HotSpot，因此本文中默认介绍的虚拟机都是HotSpot，相关机制也主要是指HotSpot的GC机制。

Java GC机制主要完成3件事：

确定哪些内存需要回收  
确定什么时候需要执行GC  
如何执行GC

经过这么长时间的发展，Java GC机制已经日臻完善，几乎可以自动的为我们做绝大多数的事情。然而，如果我们从事较大型的应用软件开发，曾经出现过内存优化的需求，就必定要研究Java GC机制。

　　学习Java GC机制，可以帮助我们在日常工作中排查各种内存溢出或泄露问题，解决性能瓶颈，达到更高的并发量，写出更高效的程序。

--------------------- 本文来自 叩丁狼教育 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/wolfcode\_cn/article/details/81626108?utm\_source=copy

#### 算法

Java语言规范没有明确地说明JVM使用哪种垃圾回收算法，但是任何一种垃圾回收算法一般要做2件基本的事情：（1）发现无用信息对象；（2）回收被无用对象占用的内存空间，使该空间可被程序再次使用。

##### **引用计数法(Reference Counting Collector)**

1.1算法分析

引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

1.2优缺点

优点：

引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：

无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.

1.3引用计数算法无法解决循环引用问题，例如：

public class Main {

    public static void main(String[] args) {

        MyObject object1 = new MyObject();

        MyObject object2 = new MyObject();

        object1.object = object2;

        object2.object = object1;

        object1 = null;

        object2 = null;

    }

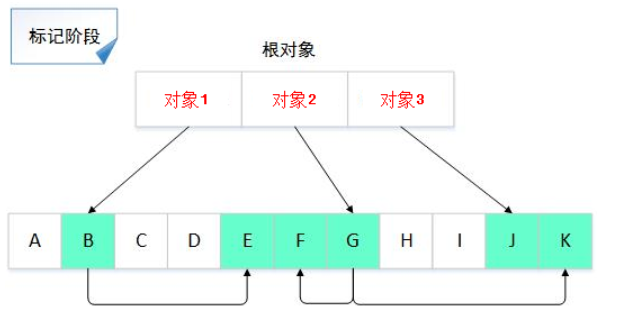
}

最后面两句将object1和object2赋值为null，也就是说object1和object2指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为0，那么垃圾收集器就永远不会回收它们。

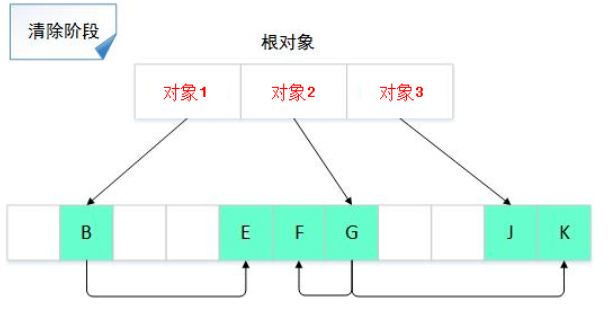
##### 最基础的收集算法 — 标记/清除算法

标记/清除算法的基本思想就跟它的名字一样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

标记阶段：标记的过程其实就是检查哪些对象能被外界访问的可达性分析算法的过程，遍历所有的GC Roots对象，对从GC Roots对象可达的对象都打上一个标识，一般是在对象的header中，将其记录为可达对象。



清除阶段：清除的过程是对堆内存进行遍历，如果发现某个对象没有被标记为可达对象（通过读取对象header信息），则将其回收。



上面两张图是标记/清除算法的示意图，在标记阶段，从对象1可以访问到B对象，从B对象又可以访问到E对象，因此从GC 对象1到B、E都是可达的，同理，对象F、G、J、K都是可达对象；到了清除阶段，所有不可达对象都会被回收。

在垃圾收集器进行GC时，必须停止所有Java执行线程（也称”Stop The World”），原因是在标记阶段进行可达性分析时，不可以出现分析过程中对象引用关系还在不断变化的情况，否则的话可达性分析结果的准确性就无法得到保证。在等待标记清除结束后，应用线程才会恢复运行。

标记/清除算法存在问题：

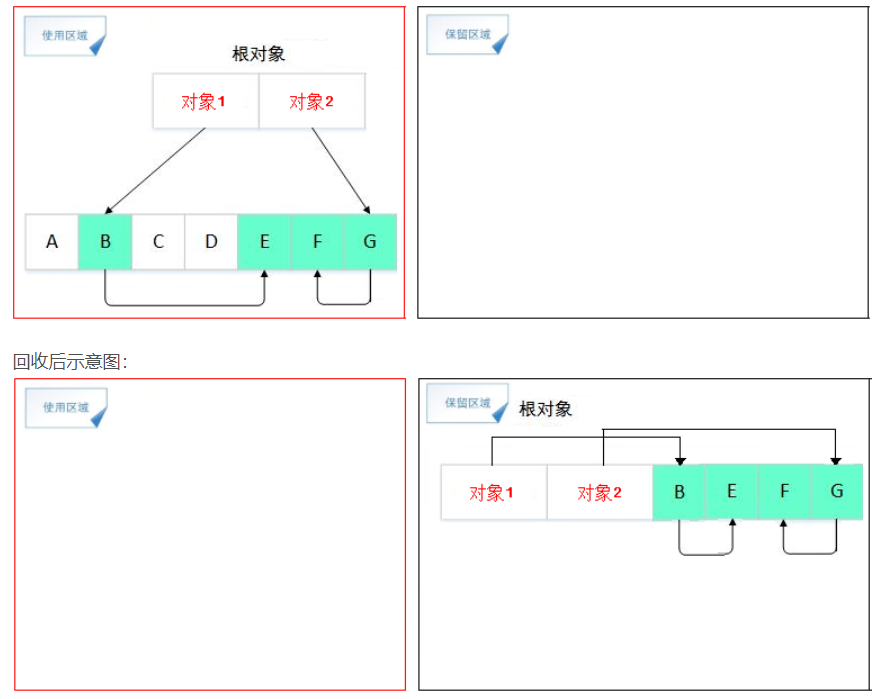
1、效率问题。标记和清除两个阶段的效率都不高，因为这两个阶段都需要遍历内存中的对象，很多时候内存中的对象实例数量是非常庞大的，这无疑很耗费时间，而且GC时需要停止应用程序，这会导致非常差的用户体验。

2、空间问题。标记清除之后会产生大量不连续的内存碎片（从上图可以看出），内存空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾回收动作。

既然标记/清除算法有这么多的缺点，那它还有存在的意义吗？别急，一个算法有缺陷，人们肯定会想办法去完善它，接下来的两个算法就是在标记/清除算法的基础上完善而来的。

##### 复制算法

为了解决效率问题，复制算法出现了。复制算法的原理是：将可用内存按容量划分为大小相等的两块，每次使用其中的一块。当这一块的内存用完了，就将还存活的对象复制到另一块内存上，然后把这一块内存所有的对象一次性清理掉。  
回收前示意图：



复制算法存在问题：

复制算法每次都是对整个半区进行内存回收，这样就减少了标记对象遍历的时间，在清除使用区域对象时，不用进行遍历，直接清空整个区域内存，而且在将存活对象复制到保留区域时也是按地址顺序存储的，这样就解决了内存碎片的问题，在分配对象内存时不用考虑内存碎片等复杂问题，只需要按顺序分配内存即可。

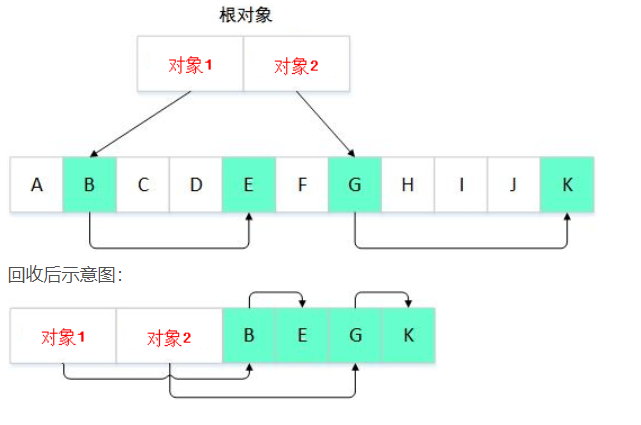
复制算法简单高效，优化了标记/清除算法的效率低、内存碎片多的问题。但是它的缺点也很明显：

1、将内存缩小为原来的一半，浪费了一半的内存空间，代价太高；

2、如果对象的存活率很高，极端一点的情况假设对象存活率为100%，那么我们需要将所有存活的对象复制一遍，耗费的时间代价也是不可忽视的。

##### 标记/整理算法

复制算法在对象存活率较高时要进行较多的复制操作，效率会变得很低，更关键的是，如果不想浪费50%的内存空间，就需要有额外的内存空间进行分配担保，以应对内存中对象100%存活的极端情况，因此，在老年代中由于对象的存活率非常高，复制算法就不合适了。根据老年代的特点，高人们提出了另一种算法：标记/整理算法。从名字上看，这种算法与标记/清除算法很像，事实上，标记/整理算法的标记过程任然与标记/清除算法一样，但后续步骤不是直接对可回收对象进行回收，而是让所有存活的对象都向一端移动，然后直接清理掉端边线以外的内存。  
回收前示意图：



--------------------- 本文来自 叩丁狼教育 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/wolfcode_cn/article/details/81626128?utm_source=copy>

##### 对象A和B循环引用，最后会不会不被GC回收？

①首先说一下，GC里边在JVM当中是使用的ROOT算法，ROOT算法，什么称作为ROOT呢，就是说类的静态成员，静态成员就是static修饰的那种，是“根”的一个，根还包括方法中的成员变量，只有成员或对象不挂在根上，GC的时候就可能把他们搞掉，这里提到的循环引用，就看这个循环引用是否挂在根上，如果挂在根上，如果这个根还被JVM的Java代码所执行的话，就不会GC掉，如果说这个根已经被释放掉了，这个对象不挂在跟上了，那个这个对象就会被GC掉。

②说一下根搜索算法，ROOTS，这个算法，哪些在Java里会被认为是根呢，在我印象里一般是static修饰的类成员，比如说静态字段，这种字段引用的对象被称为根，只要类在POOL区里不被卸载，一直在堆里，类对象只要没被回收掉，他引用的对象就不会被GC。

③再说另一种情况，方法中的栈，栈中有他的栈成员 Integer a = XXX，当方法没有被释放，没有出栈的时候，方法没有被弹出的时候，那Integer a 所引用的对象也是不会被回收的，在什么情况下回收呢，就是这个对象没有挂在根上，就会被回收。

④我们回到标题的问题，这个循环引用是否被回收，就看这个循环引用是否挂在根上，A引用B，B引用A，A和Ｂ并没有挂在某个内存元和根上，当他们的生命周期结束的时候，这两个对象都有可能被回收。

⑥具体回收的机制，就比较复杂了，每次GC的时候，对要被回收的对象标记一次，比如说会有个计数器每次+1，+1，+1，每次GC的时候就+1一次，当对象达到默认值了，比如说好像15次吧，在新生带创建的对象达到15次了就会被达到老年带里去，而老年代对象的回收的频率和新生带回收的频率是不一样的，可以仔细看下图中pool里的分区，了解他们的运行机制。

注：JVM heap分区块

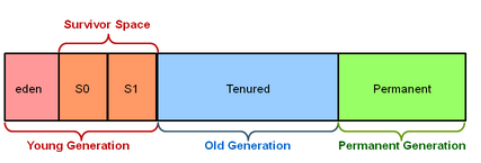
Generation代

* YongGeneration/NewGeneration：新生代，在Eden/S0/S1的存活的对象。
* OldGeneration：老年代，在Tenured区存活的对象。
* PermanentGeneration：永久代。

Space 区

* Eden：伊甸园区，是新生代的一个区。- Survivor：幸存区，属于新生代，为了复制算法的需要。一般分成大小相等的两个区（S0/S1或者From/To）。
* Tenured：存放老年代的区域。
* Permanent：终身区。

下图：Hotspot 的 Heap 分区



--------------------- 本文来自 徐刘根 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/xlgen157387/article/details/44832049?utm\_source=copy

## JDBC

### JDBC是什么？

JDBC（JAVA DataBase Connection）即JAVA数据库连接技术，JDBC API是一个Java API，可以访问任何类型表列数据，特别是存储在关系数据库中的数据。   
JDBC库中所包含的API通常与数据库使用于：

* 连接到数据库
* 创建SQL语句
* 在数据库中执行SQL查询
* 查看和修改数据库中的数据记录

### 数据库驱动与常用接口

JDBC API使用数据库驱动连接操作数据库，其实也就是数据库厂商的JDBC接口实现，即对Connection等接口的实现类的jar文件。   
我使用的是MySql数据库，所以使用MySql提供的数据库驱动包mysql-connector-java-5.1.40-bin.jar。   
驱动包一般会提供很多操作数据库的接口，常用的有：

1. **Driver接口：**

在编程中要连接数据库，必须先装载特定厂商的数据库驱动程序，不同的数据库有不同的装载方法，MySql驱动装载方法为：Class.forName(“com.mysql.jdbc.Driver”);

1. **Connection接口：**

Connection与特定数据库的连接（会话），在连接上下文中执行sql语句并返回结果。DriverManager.getConnection(url, user, password)方法可以获取相应的数据库连接。MySql数据库连接获取方法为：Connection conn = DriverManager.getConnection(“jdbc:mysql://host:port/database”, “user”, “password”)。该接口包含了几个常用的方法：  
createStatement():创建向数据库发送sql语句的statement对象；   
prepareStatement(sql) ：创建向数据库发送预编译sql的PrepareSatement对象；   
setAutoCommit(boolean)：设置数据库事务是否自动提交；   
commit()：提交事务；   
rollback()：事务回滚。

1. **Statement接口：**

用于执行静态SQL语句并返回它所生成结果的对象。一般有三种Statement对象：   
1、Statement：

由createStatement创建，用于发送简单的SQL语句（不带参数）；   
2、PreparedStatement ：

继承自Statement接口，由preparedStatement创建，用于发送含有一个或多个参数的SQL语句。PreparedStatement对象比Statement对象的效率更高，并且可以防止SQL注入，所以我们一般都使用PreparedStatement；   
3、CallableStatement：

继承自PreparedStatement接口，由方法prepareCall创建，用于调用存储过程。   
以下是几个常用的Statement方法：   
1、execute(String sql):运行语句，返回是否有结果集；   
2、executeQuery(String sql)：运行select语句，返回ResultSet结果集；   
3、executeUpdate(String sql)：运行insert/update/delete操作，返回更新的行数；   
4、addBatch(String sql) ：把多条sql语句放到一个批处理中；   
5、executeBatch()：向数据库发送一批sql语句执行。

1. **ResultSet接口：ResultSet提供检索不同类型字段的方法，常用的有：**   
   1、getString(int index)、getString(String columnName)：获得在数据库里是varchar、char等类型的数据对象；   
   2、getFloat(int index)、getFloat(String columnName)：获得在数据库里是Float类型的数据对象；   
   3、getDate(int index)、getDate(String columnName)：获得在数据库里是Date类型的数据；   
   4、getBoolean(int index)、getBoolean(String columnName)：获得在数据库里是Boolean类型的数据；   
   5、getObject(int index)、getObject(String columnName)：获取在数据库里任意类型的数据。
2. **相比于以上方法，ResultSet接口提供了更常用的对于结果集的处理方法：**   
   1、next()：移动到下一行；   
   2、Previous()：移动到前一行；   
   3、absolute(int row)：移动到指定行；   
   4、beforeFirst()：移动resultSet的最前面；   
   5、afterLast() ：移动到resultSet的最后面。

### JDBC使用步骤

* JDBC标准流程为：加载JDBC驱动程序→建立数据库连接Connection→创建执行SQL的语句Statement→处理执行结果ResultSet→释放资源，其中释放资源顺序为：ResultSet→Statement→Connection。   
  以下是一个基本例子：   
  首先创建要是用的数据库表：
* create database db\_test;
* use db\_test;
* create table person(
* id int primary key,
* name varchar(20),
* job varchar(20)
* );
* insert into person values(1001,'zhangsan','teacher');
* */\*\*  
   \* 获取数据库连接方法  
   \** ***@return*** *\*/* private static Connection getConn(){  
   String driver = "com.mysql.jdbc.Driver";  
   //后面的一串表示关闭ssl协议  
   String url = "jdbc:mysql://192.168.1.110:3306/db\_test?useUnicode=true&characterEncoding=utf-8&useSSL=false";  
   String username = "root";  
   String password = "123456";  
   Connection connection = null;  
    
   try {  
   //加载驱动  
   Class.*forName*(driver);  
   //获取数据库连接  
   connection = (Connection) DriverManager.*getConnection*(url, username, password);  
   }catch (SQLException e){  
   e.printStackTrace();  
   }catch (ClassNotFoundException e){  
   e.printStackTrace();  
   }  
   return connection;  
   }  
   */\*\*  
   \* 以下基于Statement  
   \*/* //执行查询类的SQL语句，有返回集  
   public ResultSet executeQuery(String sql)  
   {  
   try  
   {  
   stmt=conn.createStatement(ResultSet.*TYPE\_SCROLL\_SENSITIVE*,ResultSet.*CONCUR\_UPDATABLE*);  
   rs=stmt.executeQuery(sql);  
   }catch(SQLException e){  
   System.*err*.println("Data.executeQuery: " + e.getMessage());  
   }  
   return rs;  
   }  
    
   */\*\*  
   \* insert  
   \*/* public int executeUpdate(String sql) {  
   int rows = 0;  
   try  
   {  
   stmt=conn.createStatement();  
   rows = stmt.executeUpdate(sql);  
    
   }catch(SQLException e){  
   System.*err*.println("Data.executeUpdate: " + e.getMessage());  
   }  
   return rows;  
   }  
     
   */\*\*  
   \* 以下基于PreparedStatement  
   \*/  
   /\*\*  
   \* 插入方法  
   \** ***@param*** *person  
   \** ***@return*** *\*/* private static int insert(Person person){  
   Connection connection = getConn();  
   int i = 0;  
   String sql = "insert into person (id,name,job) values(?,?,?);";  
   PreparedStatement pstmt;  
   try {  
   //从连接获取PreparedStatement对象  
   pstmt = (PreparedStatement) connection.prepareStatement(sql);  
   //组装sql  
   pstmt.setInt(1, person.getId());  
   pstmt.setString(2, person.getName());  
   pstmt.setString(3, person.getJob());  
   //执行sql  
   i = pstmt.executeUpdate();  
   //关闭连接  
   pstmt.close();  
   connection.close();  
   }catch (SQLException e){  
   e.printStackTrace();  
   }  
   return i;  
   }  
    
   */\*\*  
   \* 更新方法  
   \** ***@param*** *person  
   \** ***@return*** *\*/* private static int update(Person person) {  
   Connection conn = getConn();  
   int i = 0;  
   String sql = "update person set name='" + person.getName() + "' where id='" + person.getId() + "'";  
   PreparedStatement pstmt;  
   try {  
   pstmt = (PreparedStatement) conn.prepareStatement(sql);  
   i = pstmt.executeUpdate();  
   System.*out*.println("resutl: " + i);  
   pstmt.close();  
   conn.close();  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   return i;  
   }  
    
   private static Integer getAll() {  
   Connection conn = getConn();  
   String sql = "select \* from person";  
   PreparedStatement pstmt;  
   try {  
   pstmt = (PreparedStatement)conn.prepareStatement(sql);  
   ResultSet rs = pstmt.executeQuery();  
   int col = rs.getMetaData().getColumnCount();  
   System.*out*.println("============================");  
   while (rs.next()) {  
   for (int i = 1; i <= col; i++) {  
   System.*out*.print(rs.getString(i) + "\t");  
   if ((i == 2) && (rs.getString(i).length() < 8)) {  
   System.*out*.print("\t");  
   }  
   }  
   System.*out*.println("");  
   }  
   System.*out*.println("============================");  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   return null;  
   }  
    
   private static int delete(Integer id) {  
   Connection conn = getConn();  
   int i = 0;  
   String sql = "delete from person where id='" + id + "'";  
   PreparedStatement pstmt;  
   try {  
   pstmt = (PreparedStatement) conn.prepareStatement(sql);  
   i = pstmt.executeUpdate();  
   System.*out*.println("resutl: " + i);  
   pstmt.close();  
   conn.close();  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   return i;  
   }  
    
   }  
    
  ---------------------  
    
   本文来自 smilematch 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/smilematch/article/details/78473835?utm\_source=copy

## 反射

### 反射的概述

JAVA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。

要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

**以上的总结就是什么是反射**

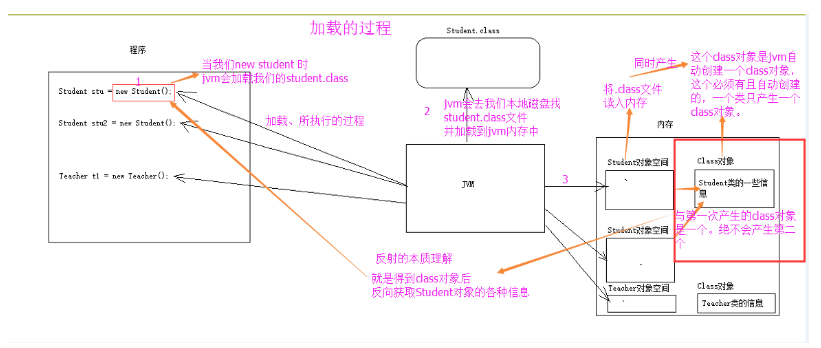
反射就是把java类中的各种成分映射成一个个的Java对象

例如：一个类有：成员变量、方法、构造方法、包等等信息，利用反射技术可以对一个类进行解剖，把个个组成部分映射成一个个对象。

     （其实：一个类中这些成员方法、构造方法、在加入类中都有一个类来描述）

如图是类的正常加载过程：反射的原理在与class对象。

熟悉一下加载的时候：Class对象的由来是将class文件读入内存，并为之创建一个Class对象。



其中这个Class对象很特殊。我们先了解一下这个Class类

### 查看Class类在java中的api详解（1.7的API）

如何阅读java中的api详见java基础之——String字符串处理



Class 类的实例表示正在运行的 Java 应用程序中的类和接口。也就是jvm中有N多的实例每个类都有该Class对象。（包括基本数据类型）

Class 没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的defineClass 方法自动构造的。也就是这不需要我们自己去处理创建，JVM已经帮我们创建好了。

没有公共的构造方法，方法共有64个太多了。下面用到哪个就详解哪个吧



### 反射的使用（这里使用Student类做演示）

#### 获取Class对象的三种方式

1.1 Object ——> getClass();  
1.2 任何数据类型（包括基本数据类型）都有一个“静态”的class属性  
1.3 通过Class类的静态方法：forName（String  className）(常用)

其中1.1是因为Object类中的getClass方法、因为所有类都继承Object类。从而调用Object类来获取



package fanshe;  
*/\*\*  
 \* 获取Class对象的三种方式  
 \* 1 Object ——> getClass();  
 \* 2 任何数据类型（包括基本数据类型）都有一个“静态”的class属性  
 \* 3 通过Class类的静态方法：forName（String className）(常用)  
 \*  
 \*/* public class Fanshe {  
 public static void main(String[] args) {  
 //第一种方式获取Class对象   
 Student stu1 = new Student();//这一new 产生一个Student对象，一个Class对象。  
 Class stuClass = stu1.getClass();//获取Class对象  
 System.*out*.println(stuClass.getName());  
  
 //第二种方式获取Class对象  
 Class stuClass2 = Student.class;  
 System.*out*.println(stuClass == stuClass2);//判断第一种方式获取的Class对象和第二种方式获取的是否是同一个  
  
 //第三种方式获取Class对象  
 try {  
 Class stuClass3 = Class.*forName*("fanshe.Student");//注意此字符串必须是真实路径，就是带包名的类路径，包名.类名  
 System.*out*.println(stuClass3 == stuClass2);//判断三种方式是否获取的是同一个Class对象  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
  
 }  
 }

**注意：在运行期间，一个类，只有一个Class对象产生。**

三种方式常用第三种，第一种对象都有了还要反射干什么。第二种需要导入类的包，依赖太强，不导包就抛编译错误。一般都第三种，一个字符串可以传入也可写在配置文件中等多种方法。

#### 通过反射获取构造方法并使用：

student类：

package fanshe;  
  
 public class Student {  
  
 //---------------构造方法-------------------  
 //（默认的构造方法）  
 Student(String str){  
 System.*out*.println("(默认)的构造方法 s = " + str);  
 }  
  
 //无参构造方法  
 public Student(){  
 System.*out*.println("调用了公有、无参构造方法执行了。。。");  
 }  
  
 //有一个参数的构造方法  
 public Student(char name){  
 System.*out*.println("姓名：" + name);  
 }  
  
 //有多个参数的构造方法  
 public Student(String name ,int age){  
 System.*out*.println("姓名："+name+"年龄："+ age);//这的执行效率有问题，以后解决。  
 }  
  
 //受保护的构造方法  
 protected Student(boolean n){  
 System.*out*.println("受保护的构造方法 n = " + n);  
 }  
  
 //私有构造方法  
 private Student(int age){  
 System.*out*.println("私有的构造方法 年龄："+ age);  
 }  
  
 }

共有6个构造方法；

测试类：

package fanshe;  
   
import java.lang.reflect.Constructor;  
   
   
/\*  
 \* 通过Class对象可以获取某个类中的：构造方法、成员变量、成员方法；并访问成员；  
 \*   
 \* 1.获取构造方法：  
 \* 1).批量的方法：  
 \* public Constructor[] getConstructors()：所有"公有的"构造方法  
 public Constructor[] getDeclaredConstructors()：获取所有的构造方法(包括私有、受保护、默认、公有)  
   
 \* 2).获取单个的方法，并调用：  
 \* public Constructor getConstructor(Class... parameterTypes):获取单个的"公有的"构造方法：  
 \* public Constructor getDeclaredConstructor(Class... parameterTypes):获取"某个构造方法"可以是私有的，或受保护、默认、公有；  
 \*   
 \* 调用构造方法：  
 \* Constructor-->newInstance(Object... initargs)  
 \*/  
 public class Constructors {  
  
 public static void main(String[] args) throws Exception {  
 //1.加载Class对象  
 Class clazz = Class.*forName*("fanshe.Student");  
  
  
 //2.获取所有公有构造方法  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*所有公有构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Constructor[] conArray = clazz.getConstructors();  
 for(Constructor c : conArray){  
 System.*out*.println(c);  
 }  
  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*所有的构造方法(包括：私有、受保护、默认、公有)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 conArray = clazz.getDeclaredConstructors();  
 for(Constructor c : conArray){  
 System.*out*.println(c);  
 }  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有、无参的构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Constructor con = clazz.getConstructor(null);  
 //1>、因为是无参的构造方法所以类型是一个null,不写也可以：这里需要的是一个参数的类型，切记是类型  
 //2>、返回的是描述这个无参构造函数的类对象。  
  
 System.*out*.println("con = " + con);  
 //调用构造方法  
 Object obj = con.newInstance();  
 // System.out.println("obj = " + obj);  
 // Student stu = (Student)obj;  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有构造方法，并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 con = clazz.getDeclaredConstructor(char.class);  
 System.*out*.println(con);  
 //调用构造方法  
 con.setAccessible(true);//暴力访问(忽略掉访问修饰符)  
 obj = con.newInstance('男');  
 }  
  
 }

后台输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*所有公有构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public fanshe.Student(java.lang.String,int)

public fanshe.Student(char)

public fanshe.Student()

\*\*\*\*\*\*\*\*\*\*\*\*所有的构造方法(包括：私有、受保护、默认、公有)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private fanshe.Student(int)

protected fanshe.Student(boolean)

public fanshe.Student(java.lang.String,int)

public fanshe.Student(char)

public fanshe.Student()

fanshe.Student(java.lang.String)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有、无参的构造方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

con = public fanshe.Student()

调用了公有、无参构造方法执行了。。。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有构造方法，并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public fanshe.Student(char)

姓名：男

**调用方法：**

1.获取构造方法：

  1).批量的方法：  
public Constructor[] getConstructors()：所有"公有的"构造方法  
            public Constructor[] getDeclaredConstructors()：获取所有的构造方法(包括私有、受保护、默认、公有)  
       
  2).获取单个的方法，并调用：  
public Constructor getConstructor(Class... parameterTypes):获取单个的"公有的"构造方法：  
public Constructor getDeclaredConstructor(Class... parameterTypes):获取"某个构造方法"可以是私有的，或受保护、默认、公有；  
  
  调用构造方法：

Constructor-->newInstance(Object... initargs)

2、newInstance是 Constructor类的方法（管理构造函数的类）

api的解释为：

newInstance(Object... initargs)  
           使用此 Constructor 对象表示的构造方法来创建该构造方法的声明类的新实例，并用指定的初始化参数初始化该实例。

它的返回值是T类型，所以newInstance是创建了一个构造方法的声明类的新实例对象。并为之调用

#### 获取成员变量并调用

student类：

package fanshe.field;  
  
 public class Student {  
 public Student(){  
  
 }  
 //\*\*\*\*\*\*\*\*\*\*字段\*\*\*\*\*\*\*\*\*\*\*\*\*//  
 public String name;  
 protected int age;  
 char sex;  
 private String phoneNum;  
  
 @Override  
 public String toString() {  
 return "Student [name=" + name + ", age=" + age + ", sex=" + sex  
 + ", phoneNum=" + phoneNum + "]";  
 }  
  
  
 }

测试类：

package fanshe.field;  
import java.lang.reflect.Field;  
 /\*  
 \* 获取成员变量并调用：  
 \*  
 \* 1.批量的  
 \* 1).Field[] getFields():获取所有的"公有字段"  
 \* 2).Field[] getDeclaredFields():获取所有字段，包括：私有、受保护、默认、公有；  
 \* 2.获取单个的：  
 \* 1).public Field getField(String fieldName):获取某个"公有的"字段；  
 \* 2).public Field getDeclaredField(String fieldName):获取某个字段(可以是私有的)  
 \*  
 \* 设置字段的值：  
 \* Field --> public void set(Object obj,Object value):  
 \* 参数说明：  
 \* 1.obj:要设置的字段所在的对象；  
 \* 2.value:要为字段设置的值；  
 \*  
 \*/  
 public class Fields {  
  
 public static void main(String[] args) throws Exception {  
 //1.获取Class对象  
 Class stuClass = Class.*forName*("fanshe.field.Student");  
 //2.获取字段  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*获取所有公有的字段\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Field[] fieldArray = stuClass.getFields();  
 for(Field f : fieldArray){  
 System.*out*.println(f);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*获取所有的字段(包括私有、受保护、默认的)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 fieldArray = stuClass.getDeclaredFields();  
 for(Field f : fieldArray){  
 System.*out*.println(f);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有字段\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Field f = stuClass.getField("name");  
 System.*out*.println(f);  
 //获取一个对象  
 Object obj = stuClass.getConstructor().newInstance();//产生Student对象--》Student stu = new Student();  
 //为字段设置值  
 f.set(obj, "刘德华");//为Student对象中的name属性赋值--》stu.name = "刘德华"  
 //验证  
 Student stu = (Student)obj;  
 System.*out*.println("验证姓名：" + stu.name);  
  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有字段\*\*\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 f = stuClass.getDeclaredField("phoneNum");  
 System.*out*.println(f);  
 f.setAccessible(true);//暴力反射，解除私有限定  
 f.set(obj, "18888889999");  
 System.*out*.println("验证电话：" + stu);  
  
 }  
 }

后台输出：

\*\*\*\*\*\*\*\*\*\*\*\*获取所有公有的字段\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public java.lang.String fanshe.field.Student.name

\*\*\*\*\*\*\*\*\*\*\*\*获取所有的字段(包括私有、受保护、默认的)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public java.lang.String fanshe.field.Student.name

protected int fanshe.field.Student.age

char fanshe.field.Student.sex

private java.lang.String fanshe.field.Student.phoneNum

\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有字段\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public java.lang.String fanshe.field.Student.name

验证姓名：刘德华

\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有字段\*\*\*\*并调用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private java.lang.String fanshe.field.Student.phoneNum

验证电话：Student [name=刘德华, age=0, sex=

由此可见

调用字段时：需要传递两个参数：

Object obj = stuClass.getConstructor().newInstance();//产生Student对象--》Student stu = new Student();  
//为字段设置值  
f.set(obj, "刘德华");//为Student对象中的name属性赋值--》stu.name = "刘德华"

第一个参数：要传入设置的对象，第二个参数：要传入实参

#### 获取成员方法并调用

student类：

package fanshe.method;  
  
 public class Student {  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*成员方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  
 public void show1(String s){  
 System.*out*.println("调用了：公有的，String参数的show1(): s = " + s);  
 }  
 protected void show2(){  
 System.*out*.println("调用了：受保护的，无参的show2()");  
 }  
 void show3(){  
 System.*out*.println("调用了：默认的，无参的show3()");  
 }  
 private String show4(int age){  
 System.*out*.println("调用了，私有的，并且有返回值的，int参数的show4(): age = " + age);  
 return "abcd";  
 }  
 }

测试类：

package fanshe.method;  
   
import java.lang.reflect.Method;  
   
/\*  
 \* 获取成员方法并调用：  
 \*   
 \* 1.批量的：  
 \* public Method[] getMethods():获取所有"公有方法"；（包含了父类的方法也包含Object类）  
 \* public Method[] getDeclaredMethods():获取所有的成员方法，包括私有的(不包括继承的)  
 \* 2.获取单个的：  
 \* public Method getMethod(String name,Class<?>... parameterTypes):  
 \* 参数：  
 \* name : 方法名；  
 \* Class ... : 形参的Class类型对象  
 \* public Method getDeclaredMethod(String name,Class<?>... parameterTypes)  
 \*   
 \* 调用方法：  
 \* Method --> public Object invoke(Object obj,Object... args):  
 \* 参数说明：  
 \* obj : 要调用方法的对象；  
 \* args:调用方式时所传递的实参；  
):  
 \*/  
 public class MethodClass {  
  
 public static void main(String[] args) throws Exception {  
 //1.获取Class对象  
 Class stuClass = Class.*forName*("fanshe.method.Student");  
 //2.获取所有公有方法  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 stuClass.getMethods();  
 Method[] methodArray = stuClass.getMethods();  
 for(Method m : methodArray){  
 System.*out*.println(m);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 methodArray = stuClass.getDeclaredMethods();  
 for(Method m : methodArray){  
 System.*out*.println(m);  
 }  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 Method m = stuClass.getMethod("show1", String.class);  
 System.*out*.println(m);  
 //实例化一个Student对象  
 Object obj = stuClass.getConstructor().newInstance();  
 m.invoke(obj, "刘德华");  
  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 m = stuClass.getDeclaredMethod("show4", int.class);  
 System.*out*.println(m);  
 m.setAccessible(true);//解除私有限定  
 Object result = m.invoke(obj, 20);//需要两个参数，一个是要调用的对象（获取有反射），一个是实参  
 System.*out*.println("返回值：" + result);  
  
  
 }  
 }

控制台输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

private java.lang.String fanshe.method.Student.show4(int)

protected void fanshe.method.Student.show2()

void fanshe.method.Student.show3()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

调用了：公有的，String参数的show1(): s = 刘德华

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private java.lang.String fanshe.method.Student.show4(int)

调用了，私有的，并且有返回值的，int参数的show4(): age = 20

返回值：abcd

由此可见：

m = stuClass.getDeclaredMethod("show4", int.class);//调用制定方法（所有包括私有的），需要传入两个参数，第一个是调用的方法名称，第二个是方法的形参类型，切记是类型。  
System.out.println(m);  
m.setAccessible(true);//解除私有限定  
Object result = m.invoke(obj, 20);//需要两个参数，一个是要调用的对象（获取有反射），一个是实参  
System.out.println("返回值：" + result);//

控制台输出：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的”公有“方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取所有的方法，包括私有的\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

private java.lang.String fanshe.method.Student.show4(int)

protected void fanshe.method.Student.show2()

void fanshe.method.Student.show3()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取公有的show1()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void fanshe.method.Student.show1(java.lang.String)

调用了：公有的，String参数的show1(): s = 刘德华

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*获取私有的show4()方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private java.lang.String fanshe.method.Student.show4(int)

调用了，私有的，并且有返回值的，int参数的show4(): age = 20

返回值：abcd

其实这里的成员方法：在模型中有属性一词，就是那些setter（）方法和getter()方法。还有字段组成，这些内容在内省中详解

#### 反射main方法

student类：

package fanshe.main;  
  
 public class Student {  
  
 public static void main(String[] args) {  
 System.*out*.println("main方法执行了。。。");  
 }  
 }

测试类：

package fanshe.main;

import java.lang.reflect.Method;

/\*\*

\* 获取Student类的main方法、不要与当前的main方法搞混了

\*/

public class Main {

public static void main(String[] args) {

try {

//1、获取Student对象的字节码

Class clazz = Class.forName("fanshe.main.Student");

//2、获取main方法

Method methodMain = clazz.getMethod("main", String[].class);//第一个参数：方法名称，第二个参数：方法形参的类型，

//3、调用main方法

// methodMain.invoke(null, new String[]{"a","b","c"});

//第一个参数，对象类型，因为方法是static静态的，所以为null可以，第二个参数是String数组，这里要注意在jdk1.4时是数组，jdk1.5之后是可变参数

//这里拆的时候将 new String[]{"a","b","c"} 拆成3个对象。。。所以需要将它强转。

methodMain.invoke(null, (Object)new String[]{"a","b","c"});//方式一

// methodMain.invoke(null, new Object[]{new String[]{"a","b","c"}});//方式二

} catch (Exception e) {

e.printStackTrace();

}

}

}

控制台输出：

main方法执行了。。。

#### 反射方法的其它使用之---通过反射运行配置文件内容

student类：

public class Student {

public void show(){

System.out.println("is show()");

}

}

配置文件以txt文件为例子（pro.txt）：

className = cn.fanshe.Student

methodName = show

测试类：

import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
import java.lang.reflect.Method;  
import java.util.Properties;  
  
 /\*  
 \* 我们利用反射和配置文件，可以使：应用程序更新时，对源码无需进行任何修改  
 \* 我们只需要将新类发送给客户端，并修改配置文件即可  
 \*/  
 public class Demo {  
 public static void main(String[] args) throws Exception {  
 //通过反射获取Class对象  
 Class stuClass = Class.*forName*(*getValue*("className"));//"cn.fanshe.Student"  
 //2获取show()方法  
 Method m = stuClass.getMethod(*getValue*("methodName"));//show  
 //3.调用show()方法  
 m.invoke(stuClass.getConstructor().newInstance());  
  
 }  
  
 //此方法接收一个key，在配置文件中获取相应的value  
 public static String getValue(String key) throws IOException{  
 Properties pro = new Properties();//获取配置文件的对象  
 FileReader in = new FileReader("pro.txt");//获取输入流  
 pro.load(in);//将流加载到配置文件对象中  
 in.close();  
 return pro.getProperty(key);//返回根据key获取的value值  
 }  
 }

控制台输出：

is show()

**需求：**  
当我们升级这个系统时，不要Student类，而需要新写一个Student2的类时，这时只需要更改pro.txt的文件内容就可以了。代码就一点不用改动

要替换的student2类：

public class Student2 {

public void show2(){

System.out.println("is show2()");

}

}

配置文件更改为：

className = cn.fanshe.Student2

methodName = show2

控制台输出：

is show2();

#### 反射方法的其它使用之---通过反射越过泛型检查

泛型用在编译期，编译过后泛型擦除（消失掉）。所以是可以通过反射越过泛型检查的

测试类：

import java.lang.reflect.Method;  
import java.util.ArrayList;  
  
 /\*  
 \* 通过反射越过泛型检查  
 \*  
 \* 例如：有一个String泛型的集合，怎样能向这个集合中添加一个Integer类型的值？  
 \*/  
 public class Demo {  
 public static void main(String[] args) throws Exception{  
 ArrayList<String> strList = new ArrayList<>();  
 strList.add("aaa");  
 strList.add("bbb");  
  
 // strList.add(100);  
 //获取ArrayList的Class对象，反向的调用add()方法，添加数据  
 Class listClass = strList.getClass(); //得到 strList 对象的字节码 对象  
 //获取add()方法  
 Method m = listClass.getMethod("add", Object.class);  
 //调用add()方法  
 m.invoke(strList, 100);  
  
 //遍历集合  
 for(Object obj : strList){  
 System.*out*.println(obj);  
 }  
 }  
 }

控制台输出：

aaa  
bbb  
100

--------------------- 本文来自 敬业的小码哥 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/sinat_38259539/article/details/71799078?utm_source=copy>

## 时间复杂度

### 概述

时间频度  
一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行[测试](http://lib.csdn.net/base/softwaretest)才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)。   
  
时间复杂度   
  
在刚才提到的时间频度中，n称为问题的规模，当n不断变化时，时间频度T(n)也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。   
  
一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n),使得当n趋近于无穷大时，T（n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n)),称O(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。   
  
在各种不同算法中，若算法中语句执行次数为一个常数，则时间复杂度为O(1),另外，在时间频度不相同时，时间复杂度有可能相同，如T(n)=n2+3n+4与T(n)=4n2+2n+1它们的频度不同，但时间复杂度相同，都为O(n2)。   
  
按数量级递增排列，常见的时间复杂度有：   
  
常数阶O(1),对数阶O(log2n),线性阶O(n),   
  
线性对数阶O(nlog2n),平方阶O(n2)，立方阶O(n3),...，   
  
k次方阶O(nk),指数阶O(2n)。随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低。   
  
2、空间复杂度   
  
与时间复杂度类似，空间复杂度是指算法在计算机内执行时所需存储空间的度量。记作:   
  
S(n)=O(f(n))   
  
我们一般所讨论的是除正常占用内存开销外的辅助存储单元规模

### 常见算法时间复杂度：

O(1): 表示算法的运行时间为常量

O(n): 表示该算法是线性算法

O(㏒2n): 二分查找算法

O(n2): 对数组进行排序的各种简单算法，例如直接插入排序的算法。

O(n3): 做两个n阶矩阵的乘法运算

O(2n): 求具有n个元素集合的所有子集的算法

O(n!): 求具有N个元素的全排列的算法

优<---------------------------<劣

O(1)<O(㏒2n)<O(n)<O(n2)<O(2n)

时间复杂度按数量级递增排列依次为：常数阶O(1)、对数阶O(log2n)、线性阶O(n)、线性对数阶O(nlog2n)、平方阶O(n2)、立方阶O(n3)、……k次方阶O(nk)、指数阶O(2n)。

### 算法的时间复杂度（计算实例）

定义：如果一个问题的规模是n，解这一问题的某一算法所需要的时间为T(n)，它是n的某一函数 T(n)称为这一算法的“时间复杂性”。

当输入量n逐渐加大时，时间复杂性的极限情形称为算法的“渐近时间复杂性”。

我们常用大O表示法表示时间复杂性，注意它是某一个算法的时间复杂性。大O表示只是说有上界，由定义如果f(n)=O(n)，那显然成立f(n)=O(n^2)，它给你一个上界，但并不是上确界，但人们在表示的时候一般都习惯表示前者。

此外，一个问题本身也有它的复杂性，如果某个算法的复杂性到达了这个问题复杂性的下界，那就称这样的算法是最佳算法。

“大O记法”：在这种描述中使用的基本参数是 n，即问题实例的规模，把复杂性或运行时间表达为n的函数。这里的“O”表示量级 (order)，比如说“二分检索是 O(logn)的”,也就是说它需要“通过logn量级的步骤去检索一个规模为n的数组”记法 O ( f(n) )表示当 n增大时，运行时间至多将以正比于 f(n)的速度增长。

这种渐进估计对算法的理论分析和大致比较是非常有价值的，但在实践中细节也可能造成差异。例如，一个低附加代价的O(n2)算法在n较小的情况下可能比一个高附加代价的 O(nlogn)算法运行得更快。当然，随着n足够大以后，具有较慢上升函数的算法必然工作得更快。

O(1)

Temp=i;i=j;j=temp;

以上三条单个语句的频度均为1，该程序段的执行时间是一个与问题规模n无关的常数。算法的时间复杂度为常数阶，记作T(n)=O(1)。如果算法的执行时 间不随着问题规模n的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是O(1)。

O(n^2)

2.1. 交换i和j的内容

     sum=0；                 （一次）

     for(i=1;i<=n;i++)       （n次 ）

        for(j=1;j<=n;j++) （n^2次 ）

         sum++；       （n^2次 ）

解：T(n)=2n^2+n+1 =O(n^2)

2.2.

    for (i=1;i<n;i++)

    {

        y=y+1;         ①

        for (j=0;j<=(2\*n);j++)

           x++;        ②

    }

解： 语句1的频度是n-1

          语句2的频度是(n-1)\*(2n+1)=2n^2-n-1

          f(n)=2n^2-n-1+(n-1)=2n^2-2

          该程序的时间复杂度T(n)=O(n^2).

O(n)

2.3.//存疑

    a=0;

    b=1;                      ①

    for (i=1;i<=n;i++) ②

    {

       s=a+b;　　　　③

       b=a;　　　　　④

       a=s;　　　　　⑤

    }

解： 语句1的频度：2,

           语句2的频度： n,

          语句3的频度： n-1,

          语句4的频度：n-1,

          语句5的频度：n-1,

          T(n)=2+n+3(n-1)=4n-1=O(n).

O(log2n )

2.4.

     i=1;       ①

    while (i<=n)

       i=i\*2; ②

解： 语句1的频度是1,

          设语句2的频度是f(n),   则：2^f(n)<=n;f(n)<=log2n

          取最大值f(n)= log2n,

          T(n)=O(log2n )

O(n^3)

2.5.

    for(i=0;i<n;i++)

    {

       for(j=0;j<i;j++)

       {

          for(k=0;k<j;k++)

             x=x+2;

       }

    }

解：当i=m, j=k的时候,内层循环的次数为k当i=m时, j 可以取 0,1,...,m-1 , 所以这里最内循环共进行了0+1+...+m-1=(m-1)m/2次所以,i从0取到n, 则循环共进行了: 0+(1-1)\*1/2+...+(n-1)n/2=n(n+1)(n-1)/6所以时间复杂度为O(n^3).

我们还应该区分算法的最坏情况的行为和期望行为。如快速排序的最 坏情况运行时间是 O(n^2)，但期望时间是 O(nlogn)。通过每次都仔细 地选择基准值，我们有可能把平方情况 (即O(n^2)情况)的概率减小到几乎等于 0。在实际中，精心实现的快速排序一般都能以 (O(nlogn)时间运行。

下面是一些常用的记法：

访问数组中的元素是常数时间操作，或说O(1)操作。一个算法如 果能在每个步骤去掉一半数据元素，如二分检索，通常它就取 O(logn)时间。用strcmp比较两个具有n个字符的串需要O(n)时间 。常规的矩阵乘算法是O(n^3)，因为算出每个元素都需要将n对 元素相乘并加到一起，所有元素的个数是n^2。

指数时间算法通常来源于需要求出所有可能结果。例如，n个元 素的集合共有2n个子集,所以要求出所有子集的算法将是O(2n)的 。指数算法一般说来是太复杂了，除非n的值非常小，因为，在 这个问题中增加一个元素就导致运行时间加倍。不幸的是，确实有许多问题 (如著名 的“巡回售货员问题” )，到目前为止找到的算法都是指数的。如果我们真的遇到这种情况， 通常应该用寻找近似最佳结果的算法替代之。