### java多线程

1.1实现多线程

1. 继承Thread类，重写run()方法

Class myThread extend Thread{

**@override**

Public run(){ System.out.printIn("hello"); }}

Class test{

Public **st atic** void main(Strings arg[]){

Thread a = **new** myThread();

a.start();}}

1. 实现Runnable接口，并实现该接口的run()方法

Class myRunnable extend Runnable{

**@override**

Public run(){ System.out.printIn("hello"); }}

Class test{

Public **static** void main(Strings arg[]){

Runnable b = **new** myRunnable();

Thread a = new Thread(b);

a.start();

a.start();}

}

1. Callable接口，重写call()方法，它有以下几个特点
2. 可以在任务结束后提供一个返回值
3. Call()方法可以抛出异常，run()方法不能抛出异常
4. 可以拿到一个Future对象来检测线程执行情况

Class myCallable extend Callable{

**@override**

Public String call(){ return “hello”;}

}

Class test{

Public **static** void main(Strings arg[]){

ExecutorService threadPool = Executors.newSingleThreadExecutor();

Future<String> future = threadPool.sumit(new myCallable());

Try{

System.out.printIn(future.get());

}

Catch(Exception e){

}

...

}

}

1.2线程同步

Java线程同步主要由3种方法

1 synchronized，被此关键字修饰的方法或者块同时只能有一个线程访问

1. synchronized方法

Public synchronized void mutiThreadPorcess();

1. Synchronized块

**Synchronized**{

}

1. Wait()方法和notify()方法,notifyAll()

Notify()和notifyAll只能在只能在一个同步方法或者同步快中执行，wait()方法只能在同步方法中执行。

Public **synchronized** void mutiThreadPorcess(){

While(...){

Wait();

}

...

notifyAll();

}

任何线程执行mutiThreadPorcess方法时当满足while里面的条件就会调用对象的wait()方法进入等待状态并释放锁以便让别的线程能进来。当某一个线程执行对象的notifyAll()方法时就会唤醒那些调用wait()进入等待的线程。Notify和notifyAll的区别是notify之会唤醒在等待队列中的第一个线程。

1. Lock

Lock以及他的实现类ReentrantLokc(重入锁)也能实现线程同步。

Public void mutiThreadPorcess(){

Final Lock lock = new ReentrantLock();

Try{

...

}catch(){

...

}finally{

lock.lock();//Lock必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象

}

}

1.3 sleep()和wait()的区别

Sleep()是线程进行休眠状态一段时间就会自动唤醒多用户定时任务。而wait()是用于线程同步的，一个线程调用wait()状态一定要别的线程调用notify()或notifyAll()唤醒。wait()方法只能在同步方法中执行。

1.4终止线程方法

可以用stop()方法和supend()方法结束线程，但这两种方法容易造成死锁（两个线程互相等待的现象）。A线程获得锁当执行supend()线程挂起结束但不会释放锁，当线程B试图获取这个锁的时候就会出现死锁。因此安全种子线程有两种方法：

1. run()方法执行结束
2. Interrupt()方法，它会让线程执行run()方法抛出InterruptedException异常，通过捕获异常正常退出。

Class test{

Public **static** void main(Strings arg[]){

Thread a = new Thread(){

@override

Public run(){

Try{

Thread.sleep(10000);//模拟假设一个线程休眠

}catch(){

}

}

};

a.start();

a.interrupt();

}

}

1.5守护进程

普通线程成为用户线程，在线程执行start()方法之前调用对象的setDaemon(true)，该线程就变为守护线程了。守护进程一般维护用户线程的作用，但所有用户线程结束时jvm也会退出即使还有守护进程在运行。

Tread a = new Thread(){

@override

Public run(){ ...}

};

a.setDaemon(true);

a.start();

1.6 join()方法

Join()方法时让调用这个方法后面的代码在线程run()方法结束之后执行，就是将两个线程合并实现同步功能。

Class test{

Public **static** void main(Strings arg[]){

Thread a = new Thread(){

@override

Public run(){

Try{

...

}catch(){

}

}

};

a.start();

a.join();//会等待a执行run()方法执行结束在执行后面的代码，join(1000)也可指定等待时间，要是这段时间run()方法还没执行，该线程也会执行下面代码。

...//执行在a的run()方法之后

...//执行在a的run()方法之后

}

}

1. Java内存管理

2.1跨平台性

一个平台编译的代码可以在另一个平台上执行称之为跨平台。Java具有跨平台是因为它具有”中间码“和”jvm“两大机制。

Java编译生成中间码，在由jvm负责把中间码翻译成各平台能执行的代码。Java编译生成的.class为字节码即为中间码，.class需要用虚拟机加载到内存中运行。在项目启动时，jvm只会加载启动用到的.class，其他.class只会在运行时用到才会被加载到内存.class是由类加载器加载到内存的,分为五个部分：加载，验证，准备，解析，初始化.类加载器有以下几种:

* 启动类加载器(Bootstrap ClassLoader)：负责加载 JAVA\_HOME\lib 目录中的，或通过-Xbootclasspath参数指定路径中的，且被虚拟机认可（按文件名识别，如rt.jar）的类。
* 扩展类加载器(Extension ClassLoader)：负责加载 JAVA\_HOME\lib\ext 目录中的，或通过java.ext.dirs系统变量指定路径中的类库。
* 应用程序类加载器(Application ClassLoader)：负责加载用户路径（classpath）上的类库。

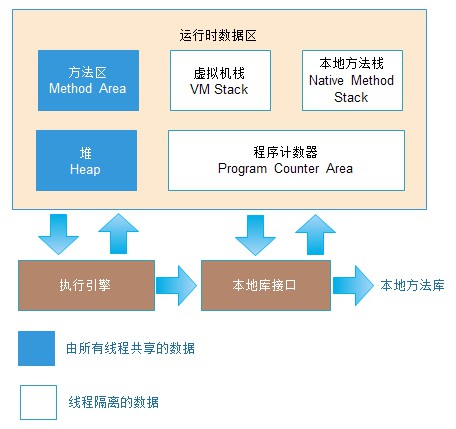


2.1 gc垃圾回收

垃圾回收算法

1. 引用计数法
2. 追踪回收算法
3. 压缩回收算法
4. 复制回收算法
5. 按代回收算法:young gc和old gc,如果一个对象经历很多次垃圾搜集仍然存货就会被标记为老年代对象.

2.2 jvm运行时区域



线程私有的：

* 程序计数器
* 虚拟机栈
* 本地方法栈
* 线程共享的：

线程共享的：

* 堆
* 方法区
* 直接内存

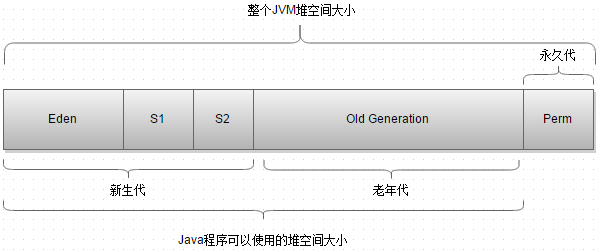
1. 虚拟机栈

主要存储数据变量int,double,char等和对象的引用.栈也是线程私有的，它的生命周期和线程相同.

Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。

1. 本地方法栈
2. 堆

存储对象,Java 堆是所有线程共享的一块内存区域,Java堆还可以细分为：新生代和老年代：再细致一点有：Eden空间、From Survivor、To Survivor空间等。

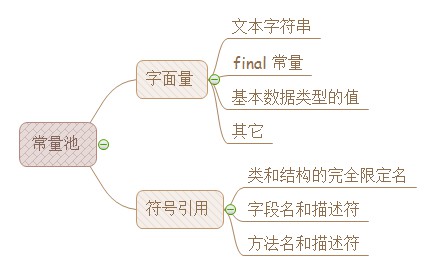


1. 方法区

方法区与 Java 堆一样，是各个线程共享的内存区域，它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

1. 常量池

JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。



3 基础知识

1. 为什么要用public static修饰main方法

Main函数是jvm运行的入口，此时对象还未生成，故需要使用static修饰，public是声明该方法是公开的方法。

1. 一个.java文件是否可以有多个main方法

可以有，但只有和文件名相同类的main方法才是jvm的入口方法。

1. 程序初始化顺序是什么

父类静态变量、父类静态代码块、子类静态变量、子类静态代码块、父类非静态变量，父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码块、子类构造函数

4、clone

实现java的clone需要以下几个步骤

1. 继承Cloneable接口
2. 重写Object类中的clone()方法
3. 在clone()方法中调用super.clone()

public class Text {

public static void main(String args[]){

Student a = new Student();

a.setName("gakki");

Student b = (Student)a.clone();

System.out.println(a.getName());

System.out.println(b.getName());

}

}

class Student implements Cloneable{

private String name = "yanyihui";

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Object clone(){

Object object = null;

try {

object = (Student)super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return object;

}

}

采用上述super.clone()方法只能浅拷贝，当类的成员变量有对象型变量时需要用到深拷贝，深拷贝就是在调用super.clone()后接着对非基本类型的成员也调用clone()方法。

5、反射机制

通过Class获取类的信息

1. Class.forName(类的路径)
2. 类名.Class
3. 对象实例.getClass()

**Public class Text**{

**Public static** void main(String[] args) {

**try**{

Class c = Class.forName("sub");

Base b = (base) c.newInstance();

b.f();

**For** (Method method:c.getMethods()) {

System.out.println(method.getName());

}c.getMethods();

}**catch**(Exception e) {

e.printStackTrace();

}

}

}

**Class base**{

**Public** void f (){

System.out.println("base");

}

}**class sub extends** base{

**Public** void f(){

System.out.println("sub");

}

}

6、内部类

1. 静态内部类

可以不以依赖部类而被实例化,但只能访问外部类的static成员变量和方法

1. 普通内部类

需要外部类被实例化之后才能实例化,能访问外部类静态和非静态的成员变量和方法,但本身不能有任何静态成员

1. 匿名内部类

一个匿名内部类只能跟在new的后面,必须继承一个类或实现一个接口,不能有类名,构造函数,静态成员,而且也不能用public,portected,private,static修饰

1. 局部内部类

定义在一个方法里的类，局部内部类和普通局部变量一样无法用pubic、protected等权限修饰符修饰。

1. 获取父类类名

子类的类名可以用getClass().getName()获取，父类的类名采用getClass().getSuperClass().getName()获取，值得注意的是this.super.getClass().getName是不能获取父类类名的，因为所有调用getClass()实际都是调用Object.getClass()的方法，它返回的是运行时的类名，因此子类无法使用次方法获取父类类名。

1. final关键字

用于声明变量，方法，类，分别表示变量不可变，方法不可被子类覆盖，类不可以被继承。

1. static关键字

主要用在修饰以下几种场景

1. 修饰成员变量，该变量可以为基本数据类型，对象成员
2. 修饰成员方法，static方法里面不能有this,super的关键字，不能调用非static方法，不能访问非static变量，只能访问static修饰的方法和变量。
3. Static代码块
4. Static内部类

1. volatile关键字

Jvm为了提高访问效率，通常会把经常被访问的变量放到缓存里，下一次取值时从缓存直接获取而不是从内存读取。被volatile关键字修饰的变量每次都是从内存里面读取的，保证取到最新的值。

1. strictfp关键字

是strict float point的缩写，指的是精确浮点。

1. finalize关键字

是object类的一个方法，在垃圾回收器执行时会调用对象的finalize()方法，可以覆盖此类方法来实现对其他资源的回收，例如关闭文件等。

获取类的方式

Class<? extends String> stringGetClass = stringer.getClass();

Class<String> stringClass = String.class;

Class.forName("java.lang.String");//newInstance()得到的是Object对象

java.lang.Class.newInstance() 只能在有默认无参构造函数的时候使用

类成员主要包括构造函数，变量和方法，Java中的操作基本都和这三者相关，而Member的这三个实现类就分别对应他们。

java.lang.reflect.Field ：对应类变量

java.lang.reflect.Method ：对应类方法

java.lang.reflect.Constructor ：对应类构造函数

2.1 Field

通过Field你可以访问给定对象的类变量，包括获取变量的类型、修饰符、注解、变量名、变量的值或者重新设置变量值，即使变量是private的。

获取Field

Class提供了4种方法获得给定类的Field。

* getDeclaredField(String name)

获取指定的变量（只要是声明的变量都能获得，包括private）

* getField(String name)

获取指定的变量（只能获得public的）

* getDeclaredFields()

获取所有声明的变量（包括private）

* getFields()

获取所有的public变量

****获取变量类型、修饰符、注解****

public void testField(){

Class c = Cat.class;

Field[] fields = c.getDeclaredFields();

for(Field f : fields){

StringBuilder builder = new StringBuilder();

//获取名称

builder.append("filed name = ");

builder.append(f.getName());

//获取类型

builder.append(" type = ");

builder.append(f.getType());

//获取修饰符

builder.append(" modifiers = ");

builder.append(Modifier.toString(f.getModifiers()));

//获取注解

Annotation[] ann = f.getAnnotations();

if (ann.length != 0) {

builder.append(" annotations = ");

for (Annotation a : ann){

builder.append(a.toString());

builder.append(" ");

}

} else {

builder.append(" -- No Annotations --");

}

Log.d(TAG, builder.toString());

}

}

获取、设置变量值

通过反射获取并改变Cat的name和age.

public void testField(){

Cat cat = new Cat("Tom", 2);

Class c = cat.getClass();

try {

//注意获取private变量时，需要用getDeclaredField

Field fieldName = c.getDeclaredField("name");

Field fieldAge = c.getField("age");

fieldName.setAccessible(true);

//反射获取名字, 年龄

String name = (String) fieldName.get(cat);

int age = fieldAge.getInt(cat);

Log.d(TAG, "before set, Cat name = " + name + " age = " + age);

//反射重新set名字和年龄

fieldName.set(cat, "Timmy");

fieldAge.setInt(cat, 3);

Log.d(TAG, "after set, Cat " + cat.toString());

} catch (NoSuchFieldException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

### **2.2 Method**

**获取Method**

Class依然提供了4种方法获取Method:

getDeclaredMethod(String name, Class<?>... parameterTypes)

根据方法名获得指定的方法， 参数name为方法名，参数parameterTypes为方法的参数类型，如 getDeclaredMethod(“eat”, String.class)

getMethod(String name, Class<?>... parameterTypes)

根据方法名获取指定的public方法，其它同上

getDeclaredMethods()

获取所有声明的方法

getMethods()

获取所有的public方法

注意：获取带参数方法时，如果参数类型错误会报NoSuchMethodException，对于参数是泛型的情况，泛型须当成Object处理（Object.class）

**获取方法返回类型**

getReturnType() 获取目标方法返回类型对应的Class对象

getGenericReturnType() 获取目标方法返回类型对应的Type对象

getReturnType()返回类型为Class，getGenericReturnType()返回类型为Type; Class实现Type。

返回值为普通简单类型如Object, int, String等，getGenericReturnType()返回值和getReturnType()一样

* 例如 public String function1()

那么各自返回值为：

getReturnType() : class java.lang.String

getGenericReturnType() : class java.lang.String

* 返回值为泛型

例如public T function2()

那么各自返回值为：

getReturnType() : class java.lang.Object

getGenericReturnType() : T

* 返回值为参数化类型

例如public Class<String> function3()

那么各自返回值为：

getReturnType() : class java.lang.Class

getGenericReturnType() : java.lang.Class<java.lang.String>

其实反射中所有形如getGenericXXX()的方法规则都与上面所述类似。

**获取方法参数类型**

getParameterTypes() 获取目标方法各参数类型对应的Class对象

getGenericParameterTypes() 获取目标方法各参数类型对应的Type对象

返回值为数组，它俩区别同上 “方法返回类型的区别” 。

**获取方法声明抛出的异常的类型**

getExceptionTypes() 获取目标方法抛出的异常类型对应的Class对象

getGenericExceptionTypes() 获取目标方法抛出的异常类型对应的Type对象

**获取方法参数名称**

.class文件中默认不存储方法参数名称，如果想要获取方法参数名称，需要在编译的时候加上-parameters参数。(构造方法的参数获取方法同样)

**通过反射调用方法**

反射通过Method的invoke()方法来调用目标方法。第一个参数为需要调用的目标类对象，如果方法为static的，则该参数为null。后面的参数都为目标方法的参数值，顺序与目标方法声明中的参数顺序一致。

注意：如果方法是private的，可以使用method.setAccessible(true)方法绕过权限检查

Class<?> c = Cat.class;

try {

//构造Cat实例

Constructor constructor = c.getConstructor(String.class, int.class);

Object cat = constructor.newInstance( "Jack", 3);

//调用无参方法

Method sleep = c.getDeclaredMethod("sleep");

sleep.invoke(cat);

//调用定项参数方法

Method eat = c.getDeclaredMethod("eat", String.class);

eat.invoke(cat, "grass");

//调用不定项参数方法

//不定项参数可以当成数组来处理

Class[] argTypes = new Class[] { String[].class };

Method varargsEat = c.getDeclaredMethod("eat", argTypes);

String[] foods = new String[]{

"grass", "meat"

};

varargsEat.invoke(cat, (Object)foods);

} catch (InstantiationException | IllegalAccessException | NoSuchMethodException | InvocationTargetException e) {

e.printStackTrace();

}

**2.3 Constructor**

这节主要介绍如何通过反射访问构造方法并通过构造方法构建新的对象

**获取构造方法**

和Method一样，Class也为Constructor提供了4种方法获取

getDeclaredConstructor(Class<?>... parameterTypes)

获取指定构造函数，参数parameterTypes为构造方法的参数类型

getConstructor(Class<?>... parameterTypes)

获取指定public构造函数，参数parameterTypes为构造方法的参数类型

getDeclaredConstructors()

获取所有声明的构造方法

getConstructors()

获取所有的public构造方法

创建对象

通过反射有两种方法可以创建对象：

java.lang.reflect.Constructor.newInstance()

Class.newInstance()

Class.newInstance()仅可用来调用无参的构造方法；Constructor.newInstance()可以调用任意参数的构造方法

Class.newInstance()会将构造方法中抛出的异常不作处理原样抛出;Constructor.newInstance()会将构造方法中抛出的异常都包装成InvocationTargetException抛出。

Class.newInstance()需要拥有构造方法的访问权限;Constructor.newInstance()可以通过setAccessible(true)方法绕过访问权限访问private构造方法。

Class<?> c = Cat.class;

try {

Constructor constructor = c.getConstructor(String.class, int.class);

Cat cat = (Cat) constructor.newInstance( "Jack", 3);

} catch (InstantiationException | IllegalAccessException | NoSuchMethodException | InvocationTargetException e) {

e.printStackTrace();

}

**接口**

getGenericInterfaces

getInterfaces

ParameterizedType

**数组类型**

数组类型：数组本质是一个对象，所以它也有自己的类型。

例如对于int[] intArray，数组类型为class [I。数组类型中的[个数代表数组的维度，例如[代表一维数组，[[代表二维数组；[后面的字母代表数组元素类型，I代表int，一般为类型的首字母大写(long类型例外，为J)。

class [B //byte类型一维数组

class [S //short类型一维数组

class [I //int类型一维数组

class [C //char类型一维数组

class [J //long类型一维数组，J代表long类型，因为L被引用对象类型占用了

class [F //float类型一维数组

class [D //double类型一维数组

class [Lcom.dada.Season //引用类型一维数组

class [[Ljava.lang.String //引用类型二维数组

创建和初始化数组

Java反射为我们提供了java.lang.reflect.Array类用来创建和初始化数组。

//创建数组， 参数componentType为数组元素的类型，后面不定项参数的个数代表数组的维度，参数值为数组长度

Array.newInstance(Class<?> componentType, int... dimensions)

//设置数组值，array为数组对象，index为数组的下标，value为需要设置的值

Array.set(Object array, int index, int value)

//获取数组的值，array为数组对象，index为数组的下标

Array.get(Object array, int index)

Object array = Array.newInstance(int.class, 2);

Array.setInt(array , 0, 1);

Array.setInt(array , 1, 2);

多维数组

Java反射没有提供能够直接访问多维数组元素的API，但你可以把多维数组当成数组的数组处理。

Object matrix = Array.newInstance(int.class, 2, 2);

Object row0 = Array.get(matrix, 0);

Object row1 = Array.get(matrix, 1);

Array.setInt(row0, 0, 1);

Array.setInt(row0, 1, 2);

Array.setInt(row1, 0, 3);

Array.setInt(row1, 1, 4);

**2.4.2 枚举**

枚举隐式继承自java.lang.Enum，Enum继承自Object，所以枚举本质也是一个类，也可以有成员变量，构造方法，方法等；对于普通类所能使用的反射方法，枚举都能使用；另外java反射额外提供了几个方法为枚举服务

15、低精度向高精度转换，byte<short<char<int<Long<float<double

涉及byte，char，short数学运算时都会转成int类型，例如

Short a =1,a = a+1会报错，因为a+1会自动转成int类型，高精度无法直接赋值给低精度数据类型，a+=1编译可以通过，这是因为+=是特殊运算符，java会做特殊处理。

16、>>和>>>

>>是有符号右移运算，>>>是无符号右移运算。两者的不同是若是正数，>>高位补0，负数高位补1，而>>>都是补0。

1. ==和equals，hashCode

Object类的equsls()方法直接使用==来比较两个对象是否相等，如果equals()方法没被子类覆盖，那么==和equals()是一样的，当包括String类在内的子类都对equals进行重写。

HashCode()返回的是对象所在地址的int值，如果没有重写hashCode()那么所有对象的hashCode值都是不一样的。

Eauals()和hashCode()

1. 若x.equals(y)==true，则x和y的hashCode值一定相同
2. 若x.equals(y)==fase,则x和y的hashCode值可能相等，也肯能不相等
3. 若x和y的hashcode值相等，x.equals(y)可能相等也可能不相等
4. 若x和y的hashcode值不相等，x.equals(y)也一定不相等
5. 数组也是对象

Int a[] = new int[5];

Int a[] = {1,2,3,4,5};

Int a[] = new int[]{1,2,3,4,5};

1. 错误类

Java.lang.throwable

Error

Exception分为检查型异常和运行时异常

1. NIO

非阻塞IO通过channel，selector，buffer实现。采用了reactor设计模式

1. 序列化