-

Java中Class类及用法

Java程序在运行时，Java运行时系统一直对所有的对象进行所谓的运行时类型标识，即所谓的RTTI。

> 这项信息纪录了每个对象所属的类。虚拟机通常使用运行时类型信息选准正确方法去执行，用来保存这些类型信息的类是Class类。Class类封装一个对象和接口运行时的状态，当装载类时，Class类型的对象自动创建。

说白了就是：

> Class类也是类的一种，只是名字和class关键字高度相似。Java是大小写敏感的语言。

> Class类的对象内容是你创建的类的类型信息，比如你创建一个shapes类，那么，Java会生成一个内容是shapes的Class类的对象

> Class类的对象不能像普通类一样，以 new shapes() 的方式创建，它的对象只能由JVM创建，因为这个类没有public构造函数

```
    /*
     * Private constructor. Only the Java Virtual Machine creates Class
objects.
```

```
     * This constructor is not used and prevents the default constructor being
     * generated.
     */
     //私有构造方法，只能由jvm进行实例化
    private Class(ClassLoader loader) {
        // Initialize final field for classLoader.  The initialization value
of non-null
        // prevents future JIT optimizations from assuming this final field is
null.
        classLoader = loader;
    }
```

> Class类的作用是运行时提供或获得某个对象的类型信息，和C++中的typeid()函数类似。这些信息也可用于反射。

Class类原理

看一下Class类的部分源码

```
//Class类中封装了类型的各种信息。在jvm中就是通过Class类的实例来获取每个Java类的所有信息的。

public class Class类 {
    Class aClass = null;

//    private EnclosingMethodInfo getEnclosingMethodInfo() {
//        Object[] enclosingInfo = getEnclosingMethod0();
//        if (enclosingInfo == null)
//            return null;
//        else {
//            return new EnclosingMethodInfo(enclosingInfo);
//        }
//    }

    /**提供原子类操作
     * Atomic operations support.
     */
//    private static class Atomic {
//        // initialize Unsafe machinery here, since we need to call
Class.class instance method
//        // and have to avoid calling it in the static initializer of the
Class class...
//        private static final Unsafe unsafe = Unsafe.getUnsafe();
//        // offset of Class.reflectionData instance field
//        private static final long reflectionDataOffset;
//        // offset of Class.annotationType instance field
//        private static final long annotationTypeOffset;
//        // offset of Class.annotationData instance field
//        private static final long annotationDataOffset;
//
//        static {
//            Field[] fields = Class.class.getDeclaredFields0(false); //
```

```
bypass caches
//            reflectionDataOffset = objectFieldOffset(fields,
"reflectionData");
//            annotationTypeOffset = objectFieldOffset(fields,
"annotationType");
//            annotationDataOffset = objectFieldOffset(fields,
"annotationData");
//        }

        //提供反射信息
    // reflection data that might get invalidated when JVM TI
RedefineClasses() is called
//    private static class ReflectionData<T> {
//        volatile Field[] declaredFields;
//        volatile Field[] publicFields;
//        volatile Method[] declaredMethods;
//        volatile Method[] publicMethods;
//        volatile Constructor<T>[] declaredConstructors;
//        volatile Constructor<T>[] publicConstructors;
//        // Intermediate results for getFields and getMethods
//        volatile Field[] declaredPublicFields;
//        volatile Method[] declaredPublicMethods;
//        volatile Class<?>[] interfaces;
//
//        // Value of classRedefinedCount when we created this ReflectionData
instance
//        final int redefinedCount;
//
//        ReflectionData(int redefinedCount) {
//            this.redefinedCount = redefinedCount;
//        }
//    }
        //方法数组
//    static class MethodArray {
//        // Don't add or remove methods except by add() or remove() calls.
//        private Method[] methods;
//        private int length;
//        private int defaults;
//
//        MethodArray() {
//            this(20);
//        }
//
//        MethodArray(int initialSize) {
//            if (initialSize < 2)
//                throw new IllegalArgumentException("Size should be 2 or
more");
//
//            methods = new Method[initialSize];
//            length = 0;
//            defaults = 0;
//        }

    //注解信息
    // annotation data that might get invalidated when JVM TI
```

```
RedefineClasses() is called
//    private static class AnnotationData {
//        final Map<Class<? extends Annotation>, Annotation> annotations;
//        final Map<Class<? extends Annotation>, Annotation>
declaredAnnotations;
//
//        // Value of classRedefinedCount when we created this AnnotationData
instance
//        final int redefinedCount;
//
//        AnnotationData(Map<Class<? extends Annotation>, Annotation>
annotations,
//                       Map<Class<? extends Annotation>, Annotation>
declaredAnnotations,
//                       int redefinedCount) {
//            this.annotations = annotations;
//            this.declaredAnnotations = declaredAnnotations;
//            this.redefinedCount = redefinedCount;
//        }
//    }
}
```

> 我们都知道所有的java类都是继承了object这个类，在object这个类中有一个方法：getclass().这个方法是用来取得该类已经被实例化了的对象的该类的引用，这个引用指向的是Class类的对象。
>
> 我们自己无法生成一个Class对象（构造函数为private)，而 这个Class类的对象是在当各类被调入时，由 Java 虚拟机自动创建 Class 对象，或通过类装载器中的 defineClass 方法生成。

```
//通过该方法可以动态地将字节码转为一个Class类对象
protected final Class<?> defineClass(String name, byte[] b, int off, int len)
    throws ClassFormatError
{
    return defineClass(name, b, off, len, null);
}
```

> 我们生成的对象都会有个字段记录该对象所属类在CLass类的对象的所在位置。如下图所示：

[外链图片转存失败(img-ZfMJTzO4-1569074134147)
(http://dl.iteye.com/upload/picture/pic/101542/0047a6e9-6608-3c3c-a67c-d8ee95e7fcb8.jpg)]

如何获得一个Class类对象

请注意，以下这些方法都是值、指某个类对应的Class对象已经在堆中生成以后，我们通过不同方式获取对这个Class对象的引用。而上面说的DefineClass才是真正将字节码加载到虚拟机的方法，会在堆中生成新的一个Class对象。

第一种办法，Class类的forName函数

> public class shapes{}
> Class obj= Class.forName("shapes"); 第二种办法，使用对象的getClass()函数

> public class shapes{} shapes s1=new shapes(); Class obj=s1.getClass(); Class obj1=s1.getSuperclass();//这个函数作用是获取shapes类的父类的类型

第三种办法，使用类字面常量

> Class obj=String.class; Class obj1=int.class; 注意，使用这种办法生成Class类对象时，不会使JVM自动加载该类（如String类）。==而其他办法会使得JVM初始化该类。==

使用Class类的对象来生成目标类的实例

> 生成不精确的object实例

==获取一个Class类的对象后，可以用 newInstance() 函数来生成目标类的一个实例。然而，该函数并不能直接生成目标类的实例，只能生成object类的实例==

> Class obj=Class.forName("shapes"); Object ShapesInstance=obj.newInstance(); 使用泛化Class引用生成带类型的目标实例

> Class obj=shapes.class; shapes newShape=obj.newInstance(); 因为有了类型限制，所以使用泛化Class语法的对象引用不能指向别的类。

```
Class obj1=int.class;
Class<Integer> obj2=int.class;
obj1=double.class;
//obj2=double.class; 这一行代码是非法的，obj2不能改指向别的类

然而，有个灵活的用法，使得你可以用Class的对象指向基类的任何子类。
Class<? extends Number> obj=int.class;
obj=Number.class;
obj=double.class;

因此，以下语法生成的Class对象可以指向任何类。
Class<?> obj=int.class;
obj=double.class;
obj=shapes.class;
最后一个奇怪的用法是，当你使用这种泛型语法来构建你手头有的一个Class类的对象的基类对象时，必须采用以下的特殊语法

public class shapes{}
class round extends shapes{}
Class<round> rclass=round.class;
Class<? super round> sclass= rclass.getSuperClass();
//Class<shapes> sclass=rclass.getSuperClass();
我们明知道，round的基类就是shapes，但是却不能直接声明 Class < shapes >，必须使用
```

特殊语法

```
Class < ? super round >
```

这个记住就可以啦。

Object类

这部分主要参考http://ihenu.iteye.com/blog/2233249

Object类是Java中其他所有类的祖先，没有Object类Java面向对象无从谈起。作为其他所有类的基类，Object具有哪些属性和行为，是Java语言设计背后的思维体现。

Object类位于java.lang包中，java.lang包包含着Java最基础和核心的类，在编译时会自动导入。Object类没有定义属性，一共有13个方法，13个方法之中并不是所有方法都是子类可访问的，一共有9个方法是所有子类都继承了的。

先大概介绍一下这些方法

1. clone方法
保护方法，实现对象的浅复制，只有实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportedException异常。
2. getClass方法
final方法，获得运行时类型。
3. toString方法
该方法用得比较多，一般子类都有覆盖。
4. finalize方法
该方法用于释放资源。因为无法确定该方法什么时候被调用，很少使用。
5. equals方法
该方法是非常重要的一个方法。一般equals和==是不一样的，但是在Object中两者是一样的。子类一般都要重写这个方法。
6. hashCode方法
该方法用于哈希查找，重写了equals方法一般都要重写hashCode方法。这个方法在一些具有哈希功能的Collection中用到。
一般必须满足obj1.equals(obj2)==true。可以推出obj1.hash-Code()==obj2.hashCode()，但是hashCode相等不一定就满足equals。不过为了提高效率，应该尽量使上面两个条件接近等价。
7. wait方法
wait方法就是使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait()方法一直等待，直到获得锁或者被中断。wait(long timeout)设定一个超时间隔，如果在规定时间内没有获得锁就返回。
调用该方法后当前线程进入睡眠状态，直到以下事件发生。
（1）其他线程调用了该对象的notify方法。
（2）其他线程调用了该对象的notifyAll方法。
（3）其他线程调用了interrupt中断该线程。
（4）时间间隔到了。
此时该线程就可以被调度了，如果是被中断的话就抛出一个InterruptedException异常。
8. notify方法
该方法唤醒在该对象上等待的某个线程。

9. notifyAll方法
该方法唤醒在该对象上等待的所有线程。

类构造器public Object();

> 大部分情况下，Java中通过形如 new A(args..)形式创建一个属于该类型的对象。其中A即是类名，A(args..)即此类定义中相对应的构造函数。通过此种形式创建的对象都是通过类中的构造函数完成。

> 为体现此特性，Java中规定：在类定义过程中，对于未定义构造函数的类，默认会有一个无参数的构造函数，作为所有类的基类，Object类自然要反映出此特性，在源码中，未给出Object类构造函数定义，但实际上，此构造函数是存在的。
>
> 当然，并不是所有的类都是通过此种方式去构建，也自然的，并不是所有的类构造函数都是public。

registerNatives()方法;

private static native void registerNatives();

> registerNatives函数前面有native关键字修饰，Java中，用native关键字修饰的函数表明该方法的实现并不是在Java中去完成，而是由C/C++去完成，并被编译成了.dll，由Java去调用。
>
> 方法的具体实现体在dll文件中，对于不同平台，其具体实现应该有所不同。用native修饰，即表示操作系统，需要提供此方法，Java本身需要使用。
>
> 具体到registerNatives()方法本身，其主要作用是将C/C++中的方法映射到Java中的native方法，实现方法命名的解耦。
>
> 既然如此，可能有人会问，registerNatives()修饰符为private，且并没有执行，作用何以达到？其实，在Java源码中，此方法的声明后有紧接着一段静态代码块：

```
private static native void registerNatives();
static {
     registerNatives();
}
```

Clone()方法实现浅拷贝

```
protected native Object clone() throwsCloneNotSupportedException;
```

> 看，clode()方法又是一个被声明为native的方法，因此，我们知道了clone()方法并不是Java的原生方法，具体的实现是有C/C++完成的。clone英文翻译为"克隆"，其目的是创建并返回此对象的一个副本。

> 形象点理解，这有一辆科鲁兹，你看着不错，想要个一模一样的。你调用此方法即可像变魔术一样变出一辆一模一样的科鲁兹出来。配置一样，长相一样。但从此刻起，原来的那辆科鲁兹如果进行了新的装饰，与你克隆出来的这辆科鲁兹没有任何关系了。
>
> 你克隆出来的对象变不变完全在于你对克隆出来的科鲁兹有没有进行过什么操作了。Java术语表述为：clone函数返回的是一个引用，指向的是新的clone出来的对象，此对象与原对象分别占用不同的堆空间。

明白了clone的含义后，接下来看看如果调用clone()函数对象进行此克隆操作。

首先看一下下面的这个例子：

```
package com.corn.objectsummary;

import com.corn.Person;

public class ObjectTest {

    public static void main(String[] args) {

        Object o1 = new Object();
        // The method clone() from the type Object is not visible
        Object clone = o1.clone();
    }

}
```

> 例子很简单，在main()方法中，new一个Oject对象后，想直接调用此对象的clone方法克隆一个对象，但是出现错误提示："The method clone() from the type Object is not visible"
>
> why? 根据提示，第一反应是ObjectTest类中定义的Oject对象无法访问其clone()方法。回到Object类中clone()方法的定义，可以看到其被声明为protected，估计问题就在这上面了，protected修饰的属性或方法表示：在同一个包内或者不同包的子类可以访问。
>
> 显然，Object类与ObjectTest类在不同的包中，但是ObjectTest继承自Object，是Object类的子类，于是，现在却出现子类中通过Object引用不能访问protected方法，原因在于对"不同包中的子类可以访问"没有正确理解。
>
> "不同包中的子类可以访问"，是指当两个类不在同一个包中的时候，继承自父类的子类内部且主调（调用者）为子类的引用时才能访问父类用protected修饰的成员（属性/方法）。 在子类内部，主调为父类的引用时并不能访问此protected修饰的成员。！ （super关键字除外）

于是，上例改成如下形式，我们发现，可以正常编译：

```
    public class clone方法 {
    public static void main(String[] args) {

    }
    public void test1() {

        User user = new User();
//        User copy = user.clone();
    }
    public void test2() {
        User user = new User();
//        User copy = (User)user.clone();
    }
}
```

是的，因为此时的主调已经是子类的引用了。

> 上述代码在运行过程中会抛出"java.lang.CloneNotSupportedException",表明clone()方法并未正确执行完毕，问题的原因在与Java中的语法规定：
>
> clone()的正确调用是需要实现Cloneable接口，如果没有实现Cloneable接口，并且子类直接调用Object类的clone()方法，则会抛出CloneNotSupportedException异常。
>
> Cloneable接口仅是一个表示接口，接口本身不包含任何方法，用来指示Object.clone()可以合法的被子类引用所调用。
>
> 于是，上述代码改成如下形式，即可正确指定clone()方法以实现克隆。

```
public class User implements Cloneable{
public int id;
public String name;
public UserInfo userInfo;

public static void main(String[] args) {
    User user = new User();
    UserInfo userInfo = new UserInfo();
    user.userInfo = userInfo;
    System.out.println(user);
    System.out.println(user.userInfo);
    try {
        User copy = (User) user.clone();
        System.out.println(copy);
        System.out.println(copy.userInfo);
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
}
//拷贝的User实例与原来不一样，是两个对象。
//    com.javase.Class和Object.Object方法.用到的类.User@4dc63996
//    com.javase.Class和Object.Object方法.用到的类.UserInfo@d716361
```

```
        //而拷贝后对象的userinfo引用对象是同一个。
    //所以这是浅拷贝
//    com.javase.Class和Object.Object方法.用到的类.User@6ff3c5b5
//    com.javase.Class和Object.Object方法.用到的类.UserInfo@d716361
}
```

总结：clone方法实现的是浅拷贝，只拷贝当前对象，并且在堆中分配新的空间，放这个复制的对象。但是对象如果里面有其他类的子对象，那么就不会拷贝到新的对象中。

==深拷贝和浅拷贝的区别==

> 浅拷贝 浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址 ，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。
>
> 深拷贝 深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。 现在为了要在clone对象时进行深拷贝， 那么就要Clonable接口，覆盖并实现clone方法，除了调用父类中的clone方法得到新的对象， 还要将该类中的引用变量也clone出来。如果只是用Object中默认的clone方法，是浅拷贝的。

那么这两种方式有什么相同和不同呢?

> new操作符的本意是分配内存。程序执行到new操作符时， 首先去看new操作符后面的类型，因为知道了类型，才能知道要分配多大的内存空间。
>
> 分配完内存之后，再调用构造函数，填充对象的各个域，这一步叫做对象的初始化，构造方法返回后，一个对象创建完毕，可以把他的引用（地址）发布到外部，在外部就可以使用这个引用操纵这个对象。
>
> 而clone在第一步是和new相似的， 都是分配内存，调用clone方法时，分配的内存和源对象（即调用clone方法的对象）相同，然后再使用原对象中对应的各个域，填充新对象的域，
>
> 填充完成之后，clone方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部。

==也就是说，一个对象在浅拷贝以后，只是把对象复制了一份放在堆空间的另一个地方，但是成员变量如果有引用指向其他对象，这个引用指向的对象和被拷贝的对象中引用指向的对象是一样的。当然，基本数据类型还是会重新拷贝一份的。 ==

getClass()方法

4.public final native Class<?> getClass();

> getClass()也是一个native方法，返回的是此Object对象的类对象/运行时类对象Class<?>。效果与Object.class相同。
>
> 首先解释下"类对象"的概念：在Java中，类是是对具有一组相同特征或行为的实例的抽象并进行描述，对象则是此类所描述的特征或行为的具体实例。
>
> 作为概念层次的类，其本身也具有某些共同的特性，如都具有类名称、由类加载器去加载，都具有包，具有父类，属性和方法等。
>
> 于是，Java中有专门定义了一个类，Class，去描述其他类所具有的这些特性，因此，从此角度去看，类本身也都是属于Class类的对象。为与经常意义上的对象相区分，在此称之为"类对象"。

```
public class getClass方法 {
    public static void main(String[] args) {
        User user = new User();
        //getclass方法是native方法，可以取到堆区唯一的Class<User>对象
        Class<?> aClass = user.getClass();
        Class bClass = User.class;
        try {
            Class cClass = Class.forName("com.javase.Class和Object.Object方法.
用到的类.User");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println(aClass);
        System.out.println(bClass);
//        class com.javase.Class和Object.Object方法.用到的类.User
//        class com.javase.Class和Object.Object方法.用到的类.User
        try {
            User a = (User) aClass.newInstance();

        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

此处主要大量涉及到Java中的反射知识

equals()方法

5.public boolean equals(Object obj);

> 与equals在Java中经常被使用，大家也都知道与equals的区别：
>
> ==表示的是变量值完成相同（对于基础类型，地址中存储的是值，引用类型则存储指向实际对象的地址）；

== 与equals区别

> equals表示的是对象的内容完全相同，此处的内容多指对象的特征/属性。

实际上，上面说法是不严谨的，更多的只是常见于String类中。首先看一下Object类中关于equals()方法的定义：

```
public boolean equals(Object obj) {
     return (this == obj);
}
```

> 由此可见，Object原生的equals()方法内部调用的正是==，与==具有相同的含义。既然如此，为什么还要定义此equals()方法？
>
> equals()方法的正确理解应该是：判断两个对象是否相等。那么判断对象相等的标尺又是什么？
>
> 如上，在object类中，此标尺即为==。当然，这个标尺不是固定的，其他类中可以按照实际的需要对此标尺含义进行重定义。如String类中则是依据字符串内容是否相等来重定义了此标尺含义。如此可以增加类的功能型和实际编码的灵活性。当然了，如果自定义的类没有重写equals()方法来重新定义此标尺，那么默认的将是其父类的equals()，直到object基类。
>
> 如下场景的实际业务需求，对于User bean，由实际的业务需求可知当属性uid相同时，表示的是同一个User，即两个User对象相等。则可以重写equals以重定义User对象相等的标尺。

ObjectTest中打印出true，因为User类定义中重写了equals()方法，这很好理解，很可能张三是一个人小名，张三丰才是其大名，判断这两个人是不是同一个人，这时只用判断uid是否相同即可。

> 如上重写equals方法表面上看上去是可以了，实则不然。因为它破坏了Java中的约定：重写equals()方法必须重写hasCode()方法。

hashCode()方法;对象的hashcode数值存储在对象的对象头中，参考JVM中对象的创建和定位

6. public native int hashCode()

hashCode()方法返回一个整形数值，表示该对象的哈希码值。 hashCode()具有如下约定：

> 1).在Java应用程序程序执行期间，对于同一对象多次调用hashCode()方法时，其返回的哈希码是相同的，前提是将对象进行equals比较时所用的标尺信息未做修改。在Java应用程序的一次执行到另外一次执行，同一对象的hashCode()返回的哈希码无须保持一致；
>
> 2).如果两个对象相等（依据：调用equals()方法），那么这两个对象调用hashCode()返回的哈希码也必须相等；

> 3).反之，两个对象调用hasCode()返回的哈希码相等，这两个对象不一定相等。

即严格的数学逻辑表示为： 两个对象相等 <=> equals()相等 => hashCode()相等。因此，重写equlas()方法必须重写hashCode()方法，以保证此逻辑严格成立，同时可以推理出：hasCode()不相等 => equals（）不相等 <=> 两个对象不相等。

可能有人在此产生疑问：既然比较两个对象是否相等的唯一条件（也是冲要条件）是equals，那么为什么还要弄出一个hashCode()，并且进行如此约定，弄得这么麻烦？

其实，这主要体现在hashCode()方法的作用上，其主要用于增强哈希表的性能。

以集合类中，以Set为例，当新加一个对象时，需要判断现有集合中是否已经存在与此对象相等的对象，如果没有hashCode()方法，需要将Set进行一次遍历，并逐一用equals()方法判断两个对象是否相等，此种算法时间复杂度为o(n)。通过借助于hasCode方法，先计算出即将新加入对象的哈希码，然后根据哈希算法计算出此对象的位置，直接判断此位置上是否已有对象即可。（注：Set的底层用的是Map的原理实现）

> 在此需要纠正一个理解上的误区：对象的hashCode()返回的不是对象所在的物理内存地址。甚至也不一定是对象的逻辑地址，hashCode()相同的两个对象，不一定相等，换言之，不相等的两个对象，hashCode()返回的哈希码可能相同。
>
> 因此，在上述代码中，重写了equals()方法后，需要重写hashCode()方法。

```
public class equals和hashcode方法 {
    @Override
    //修改equals时必须同时修改hashcode方法，否则在作为key时会出问题
    public boolean equals(Object obj) {
        return (this == obj);
    }

    @Override
    //相同的对象必须有相同hashcode，不同对象可能有相同hashcode
    public int hashCode() {
        return hashCode() >> 2;
    }
}
```

toString()方法

7.public String toString();

toString()方法返回该对象的字符串表示。先看一下Object中的具体方法体：

```
 public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

> toString()方法相信大家都经常用到，即使没有显式调用，但当我们使用System.out.println(obj)时，其内部也是通过toString()来实现的。
>
> getClass()返回对象的类对象，getClassName()以String形式返回类对象的名称（含包名）。Integer.toHexString(hashCode())则是以对象的哈希码为实参，以16进制无符号整数形式返回此哈希码的字符串表示形式。
>
> 如上例中的u1的哈希码是638，则对应的16进制为27e，调用toString()方法返回的结果为：com.corn.objectsummary.User@27e。
>
> 因此：toString()是由对象的类型和其哈希码唯一确定，同一类型但不相等的两个对象分别调用toString()方法返回的结果可能相同。

wait() notify() notifAll()

8/9/10/11/12. wait(...) / notify() / notifyAll()

> 一说到wait(...) / notify() | notifyAll()几个方法，首先想到的是线程。确实，这几个方法主要用于java多线程之间的协作。先具体看下这几个方法的主要含义：
>
> wait()：调用此方法所在的当前线程等待，直到在其他线程上调用此方法的主调（某一对象）的notify()/notifyAll()方法。
>
> wait(long timeout)/wait(long timeout, int nanos)：调用此方法所在的当前线程等待，直到在其他线程上调用此方法的主调（某一对象）的notisfy()/notisfyAll()方法，或超过指定的超时时间量。
>
> notify()/notifyAll()：唤醒在此对象监视器上等待的单个线程/所有线程。
>
> wait(...) / notify() | notifyAll()一般情况下都是配套使用。下面来看一个简单的例子：

这是一个生产者消费者的模型，只不过这里只用flag来标识哪个线程需要工作

```
public class wait和notify {
    //volatile保证线程可见性
    volatile static int flag = 1;
    //object作为锁对象，用于线程使用wait和notify方法
    volatile static Object o = new Object();
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                //wait和notify只能在同步代码块内使用
                synchronized (o) {
                    while (true) {
                        if (flag == 0) {
                            try {
                                Thread.sleep(2000);
                                System.out.println("thread1 wait");
                                //释放锁，线程挂起进入object的等待队列，后续代码
```

运行

```
                                o.wait();
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                        System.out.println("thread1 run");
                        System.out.println("notify t2");
                        flag = 0;
                        //通知等待队列的一个线程获取锁
                        o.notify();
                    }
                }
            }
        }).start();
        //解释同上
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    synchronized (o) {
                        if (flag == 1) {
                            try {
                                Thread.sleep(2000);
                                System.out.println("thread2 wait");
                                o.wait();
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                        System.out.println("thread2 run");
                        System.out.println("notify t1");
                        flag = 1;
                        o.notify();
                    }
                }
            }
        }).start();
    }

    //输出结果是
//    thread1 run
//    notify t2
//    thread1 wait
//    thread2 run
//    notify t1
//    thread2 wait
//    thread1 run
//    notify t2
//不断循环
}
```

> 从上述例子的输出结果中可以得出如下结论：

> 1、wait(...)方法调用后当前线程将立即阻塞，且适当其所持有的同步代码块中的锁，直到被唤醒或超时或打断后且重新获取到锁后才能继续执行；
>
> 2、notify()/notifyAll()方法调用后，其所在线程不会立即释放所持有的锁，直到其所在同步代码块中的代码执行完毕，此时释放锁，因此，如果其同步代码块后还有代码，其执行则依赖于JVM的线程调度。

在Java源码中，可以看到wait()具体定义如下：

```
public final void wait() throws InterruptedException {
     wait(0);
}
```

> 且wait(long timeout, int nanos)方法定义内部实质上也是通过调用wait(long timeout)完成。而wait(long timeout)是一个native方法。因此，wait(...)方法本质上都是native方式实现。

notify()/notifyAll()方法也都是native方法。

Java中线程具有较多的知识点，是一块比较大且重要的知识点。后期会有博文专门针对Java多线程作出详细总结。此处不再细述。

finalize()方法，参考JVM垃圾回收的finalization机制，刑场刀下留人

13. protected void finalize();

finalize方法主要与Java垃圾回收机制有关。首先我们看一下finalized方法在Object中的具体定义：

```
protected void finalize() throws Throwable { }
```

> 我们发现Object类中finalize方法被定义成一个空方法，为什么要如此定义呢？finalize方法的调用时机是怎么样的呢？
>
> 首先，Object中定义finalize方法表明Java中每一个对象都将具有finalize这种行为，其具体调用时机在：JVM准备对此对形象所占用的内存空间进行垃圾回收前，将被调用。由此可以看出，此方法并不是由我们主动去调用的（虽然可以主动去调用，此时与其他自定义方法无异）。

CLass类和Object类的关系

> Object类和Class类没有直接的关系。
>
> Object类是一切java类的父类，对于普通的java类，即便不声明，也是默认继承了Object类。典型的，可以使用Object类中的toString()方法。

> Class类是用于java反射机制的，一切java类，都有一个对应的Class对象，他是一个final类。Class 类的实例表示，正在运行的 Java 应用程序中的类和接口。

转一个知乎很有趣的问题 https://www.zhihu.com/question/30301819

```
Java的对象模型中：
1 所有的类都是Class类的实例，Object是类，那么Object也是Class类的一个实例。

2 所有的类都最终继承自Object类，Class是类，那么Class也继承自Object。

3 这就像是先有鸡还是先有蛋的问题，请问实际中JVM是怎么处理的？
```

> 这个问题中，第1个假设是错的：java.lang.Object是一个Java类，但并不是java.lang.Class的一个实例。后者只是一个用于描述Java类与接口的、用于支持反射操作的类型。这点上Java跟其它一些更纯粹的面向对象语言（例如Python和Ruby）不同。
>
> 而第2个假设是对的：java.lang.Class是java.lang.Object的派生类，前者继承自后者。虽然第1个假设不对，但“鸡蛋问题”仍然存在：在一个已经启动完毕、可以使用的Java对象系统里，必须要有一个java.lang.Class实例对应java.lang.Object这个类；而java.lang.Class是java.lang.Object的派生类，按“一般思维”前者应该要在后者完成初始化之后才可以初始化…
>
> 事实是：这些相互依赖的核心类型完全可以在“混沌”中一口气都初始化好，然后对象系统的状态才叫做完成了“bootstrap”，后面就可以按照Java对象系统的一般规则去运行。JVM、JavaScript、Python、Ruby等的运行时都有这样的bootstrap过程。
>
> 在“混沌”（boostrap过程）里，JVM可以为对象系统中最重要的一些核心类型先分配好内存空间，让它们进入[已分配空间]但[尚未完全初始化]状态。此时这些对象虽然已经分配了空间，但因为状态还不完整所以尚不可使用。
>
> 然后，通过这些分配好的空间把这些核心类型之间的引用关系串好。到此为止所有动作都由JVM完成，尚未执行任何Java字节码。然后这些核心类型就进入了[完全初始化]状态，对象系统就可以开始自我运行下去，也就是可以开始执行Java字节码来进一步完成Java系统的初始化了。

参考文章

https://www.cnblogs.com/congsg2016/p/5317362.html
https://www.jb51.net/article/125936.htm
https://blog.csdn.net/dufufd/article/details/80537638
https://blog.csdn.net/farsight1/article/details/80664104
https://blog.csdn.net/xiaomingdetianxia/article/details/77429180

微信公众号

Java中的equals()和hashcode()之间关系

所有Java类的父类——`java.lang.Object` 中定义了两个重要的方法：

```
public boolean equals(Object obj)
public int hashCode()
```

本文首先会给出一个错误使用这两个方法的例子，然后再解释equals和hashcode是如何协同工作的。

不重写hashcode导致一个常犯的错误，map中put进去的数据get不到，内存泄漏

先看以下代码：

```
import java.util.HashMap;

public class Apple {
    private String color;

    public Apple(String color) {
        this.color = color;
    }

    public boolean equals(Object obj) {
        if(obj==null) return false;
        if (!(obj instanceof Apple))
            return false;
        if (obj == this)
            return true;
        return this.color.equals(((Apple) obj).color);
    }

    public static void main(String[] args) {
        Apple a1 = new Apple("green");
        Apple a2 = new Apple("red");

        //hashMap stores apple type and its quantity
        HashMap<Apple, Integer> m = new HashMap<Apple, Integer>();
        m.put(a1, 10);
        m.put(a2, 20);
        System.out.println(m.get(new Apple("green")));
    }
}
```

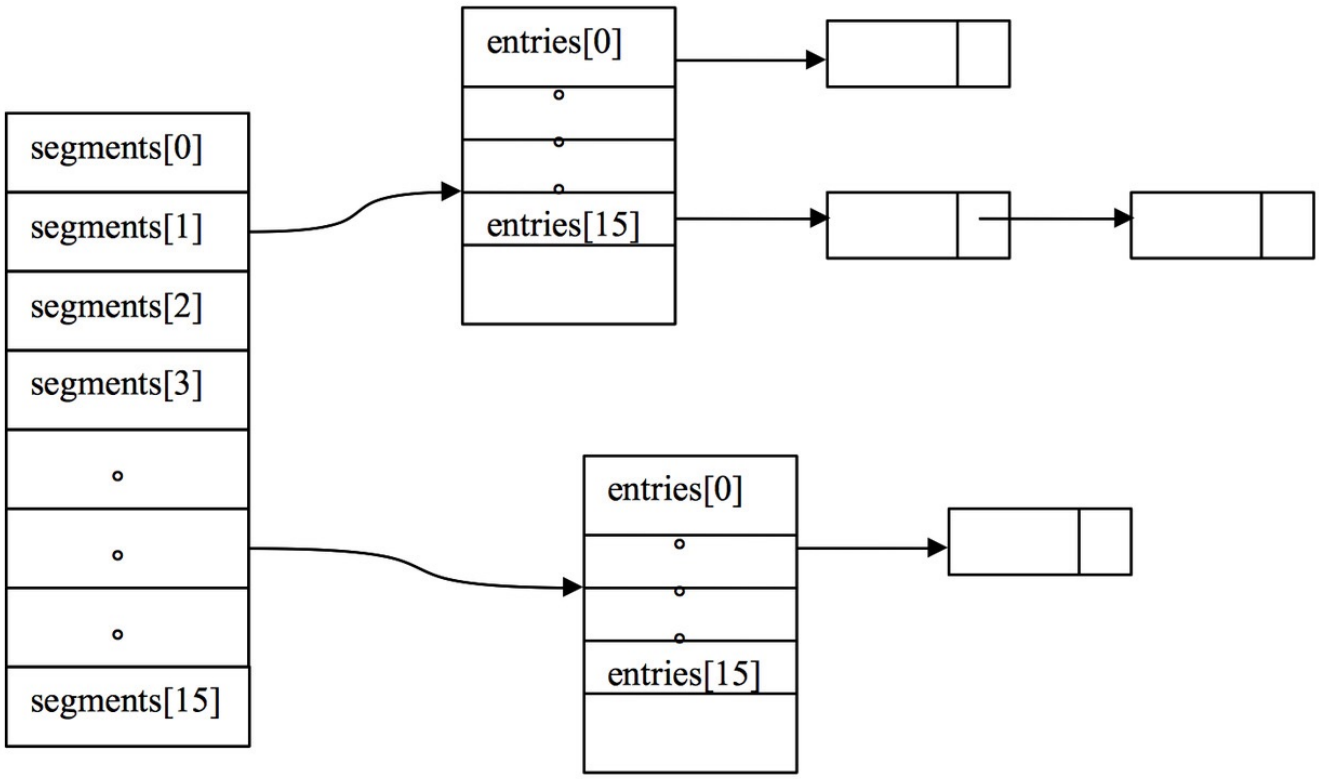
上面的代码执行过程中，先是创建个两个Apple，一个green apple和一个red apple，然后将这来两个apple存储在map中，存储之后再试图通过map的get方法获取到其中green apple的实例。读者可以试着执行以上代码，数据结果为null。也就是说刚刚通过put方法放到map中的green apple并没有通过get方法获取到。你可能怀疑是不是green apple并没有被成功的保存到map中，但是，通过debug工具可以看到，它已经被保存成功了。

hashcode()惹的祸

造成以上问题的原因其实比较简单，是因为代码中并没有重写 `hashcode` 方法。 `hashcode` 和 `equals` 的约定关系如下:

> 1、如果两个对象相等，那么他们一定有相同的哈希值（hash code）。
>
> 2、如果两个对象的哈希值相等，那么这两个对象有可能相等也有可能不相等。（需要再通过equals来判断）

如果你了解Map的工作原理，那么你一定知道，它是通过把key值进行hash来定位对象的，这样可以提供比线性存储更好的性能。实际上，Map的底层数据结构就是一个数组的数组（准确的说其实是一个链表+数组）。第一个数组的索引值是key的哈希码。通过这个索引可以定位到第二个数组，第二个数组通过使用equals方法进行线性搜索的方式来查找对象。(HashMap完全解读 (http://www.hollischuang.com/archives/82))



(http://www.hollischuang.com/wp-content/uploads/2016/03/image.jpg)

其实，一个哈希码可以映射到一个桶（bucket）中， `hashcode` 的作用就是先确定对象是属于哪个桶的。如果多个对象有相同的哈希值，那么他们可以放在同一个桶中。如果有不同的哈希值，则需要放在不同的桶中。至于同一个桶中的各个对象之前如何区分就需要使用 `equals` 方法了。

hashcode方法的默认实现会为每个对象返回一个不同的int类型的值。所以，上面的代码中，第二个apple被创建出来时他将具有不同的哈希值。可以通过重写hashCode方法来解决。

```
public int hashCode(){
    return this.color.hashCode();
}
```

总结

在判断两个对象是否相等时，不要只使用equals方法判断。还要考虑其哈希码是否相等。尤其是和hashMap等与hash相关的数据结构一起使用时。

为什么要重写 hashcode 和 equals 方法?

我在面试Java初级开发的时候，经常会问：你有没有重写过hashcode方法？不少候选人直接说没写过。我就想，或许真的没写过，于是就再通过一个问题确认：你在用HashMap的时候，键（Key）部分，有没有放过自定义对象？而这个时候，候选人说放过，于是两个问题的回答就自相矛盾了。

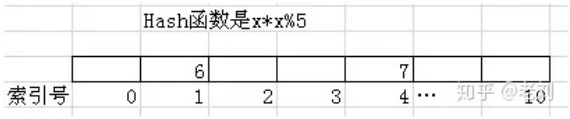
最近问下来，这个问题普遍回答不大好，于是在本文里，就干脆从hash表讲起，讲述HashMap的存数据规则，由此大家就自然清楚上述问题的答案了。

1. 通过Hash算法来了解HashMap对象的高效性

我们先复习数据结构里的一个知识点：在一个长度为n（假设是10000）的线性表（假设是ArrayList）里，存放着无序的数字；如果我们要找一个指定的数字，就不得不通过从头到尾依次遍历来查找，这样的平均查找次数是n除以2（这里是5000）。

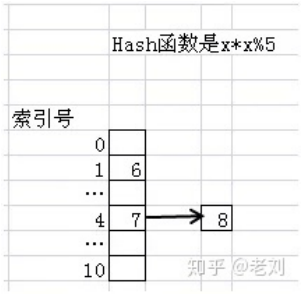
我们再来观察Hash表（这里的Hash表纯粹是数据结构上的概念，和Java无关）。它的平均查找次数接近于1，代价相当小，关键是在Hash表里，存放在其中的数据和它的存储位置是用Hash函数关联的。

我们假设一个Hash函数是x*x%5。当然实际情况里不可能用这么简单的Hash函数，我们这里纯粹



这样做的好处非常明显。比如我们要从中找6这个元素，我们可以先通过Hash函数计算6的索引位置，然后直接从1号索引里找到它了。

不过我们会遇到“Hash值冲突”这个问题。比如经过Hash函数计算后，7和8会有相同的Hash值，对此Java的HashMap对象采用的是”链地址法“的解决方案。效果如下图所示。



具体的做法是，为所有Hash值是i的对象建立一个同义词链表。假设我们在放入8的时候，发现4号位置已经被占，那么就会新建一个链表结点放入8。同样，如果我们要找8，那么发现4号索引里不是8，那会沿着链表依次查找。

虽然我们还是无法彻底避免Hash值冲突的问题，但是Hash函数设计合理，仍能保证同义词链表的长度被控制在一个合理的范围里。这里讲的理论知识并非无的放矢，大家能在后文里清晰地了解到重写hashCode方法的重要性。

2. 为什么要重写equals和hashCode方法

当我们用HashMap存入自定义的类时，如果不重写这个自定义类的equals和hashCode方法，得到的结果会和我们预期的不一样。我们来看WithoutHashCode.java这个例子。

在其中的第2到第18行，我们定义了一个Key类；在其中的第3行定义了唯一的一个属性id。当前我们先注释掉第9行的equals方法和第16行的hashCode方法。

```
1   import java.util.HashMap;
2   class Key {
3       private Integer id;
4       public Integer getId()
5   {return id; }
6       public Key(Integer id)
7   {this.id = id;  }
8   //故意先注释掉equals和hashCode方法
9   //  public boolean equals(Object o) {
10  //      if (o == null || !(o instanceof Key))
11  //      { return false; }
12  //      else
13  //      { return this.getId().equals(((Key) o).getId());}
14  //  }
15
16  //  public int hashCode()
17  //  { return id.hashCode(); }
18  }
19
20  public class WithoutHashCode {
```

```
25          hm.put(k1, "Key with id is 1");
26          System.out.println(hm.get(k2));
27      }
28  }
```

在main函数里的第22和23行，我们定义了两个Key对象，它们的id都是1，就好比它们是两把相同的都能打开同一扇门的钥匙。

在第24行里，我们通过泛型创建了一个HashMap对象。它的键部分可以存放Key类型的对象，值部分可以存储String类型的对象。

在第25行里，我们通过put方法把k1和一串字符放入到hm里； 而在第26行，我们想用k2去从HashMap里得到值；这就好比我们想用k1这把钥匙来锁门，用k2来开门。这是符合逻辑的，但从当前结果看，26行的返回结果不是我们想象中的那个字符串，而是null。

原因有两个—没有重写。第一是没有重写hashCode方法，第二是没有重写equals方法。

当我们往HashMap里放k1时，首先会调用Key这个类的hashCode方法计算它的hash值，随后把k1放入hash值所指引的内存位置。

关键是我们没有在Key里定义hashCode方法。这里调用的仍是Object类的hashCode方法（所有的类都是Object的子类），而Object类的hashCode方法返回的hash值其实是k1对象的内存地址（假设是1000）。

如果我们随后是调用hm.get(k1)，那么我们会再次调用hashCode方法（还是返回k1的地址1000），随后根据得到的hash值，能很快地找到k1。

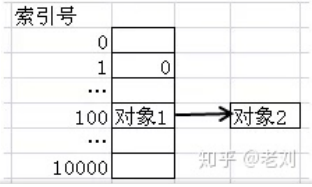
但我们这里的代码是hm.get(k2)，当我们调用Object类的hashCode方法（因为Key里没定义）计算k2的hash值时，其实得到的是k2的内存地址（假设是2000）。由于k1和k2是两个不同的对象，所以它们的内存地址一定不会相同，也就是说它们的hash值一定不同，这就是我们无法用k2的hash值去拿k1的原因。

当我们把第16和17行的hashCode方法的注释去掉后，会发现它是返回id属性的hashCode值，这里k1和k2的id都是1,所以它们的hash值是相等的。

我们再来更正一下存k1和取k2的动作。存k1时，是根据它id的hash值，假设这里是100，把k1对象放入到对应的位置。而取k2时，是先计算它的hash值（由于k2的id也是1，这个值也是100），随后到这个位置去找。

但结果会出乎我们意料：明明100号位置已经有k1，但第26行的输出结果依然是null。其原因就是没有重写Key对象的equals方法。

HashMap是用链地址法来处理冲突，也就是说，在100号位置上，有可能存在着多个用链表形式存储的对象。它们通过hashCode方法返回的hash值都是100。



对象的equals方法来判断两者是否相等了。

由于我们在Key对象里没有定义equals方法，系统就不得不调用Object类的equals方法。由于Object的固有方法是根据两个对象的内存地址来判断，所以k1和k2一定不会相等，这就是为什么依然在26行通过hm.get(k2)依然得到null的原因。

为了解决这个问题，我们需要打开第9到14行equals方法的注释。在这个方法里，只要两个对象都是Key类型，而且它们的id相等，它们就相等。

3. 对面试问题的说明

由于在项目里经常会用到HashMap，所以我在面试的时候一定会问这个问题：你有没有重写过hashCode方法？你在使用HashMap时有没有重写hashCode和equals方法？你是怎么写的？

根据问下来的结果，我发现初级程序员对这个知识点普遍没掌握好。重申一下，如果大家要在HashMap的“键”部分存放自定义的对象，一定要在这个对象里用自己的equals和hashCode方法来覆盖Object里的同名方法。

浅拷贝代码举例

```
public class ShallowCloneExample implements Cloneable {

    private int[] arr;

    public ShallowCloneExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }

    @Override
    protected ShallowCloneExample clone() throws CloneNotSupportedException {
        return (ShallowCloneExample) super.clone();
    }
}
ShallowCloneExample e1 = new ShallowCloneExample();
ShallowCloneExample e2 = null;
try {
    e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 222
```

深拷贝代码举例

```
public class DeepCloneExample implements Cloneable {

    private int[] arr;

    public DeepCloneExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }

    @Override
    protected DeepCloneExample clone() throws CloneNotSupportedException {
        DeepCloneExample result = (DeepCloneExample) super.clone();
result.arr = new int[arr.length];//还可以通过反射深拷贝
        for (int i = 0; i < arr.length; i++) {
            result.arr[i] = arr[i];
        }
        return result;
    }
}
DeepCloneExample e1 = new DeepCloneExample();
DeepCloneExample e2 = null;
try {
    e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 2
```