# Java

- Oracle JDK有部分源码是闭源的，如果确实需要可以查看OpenJDK的源码，可以在该网站获取。
- http://grepcode.com/snapshot/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/
- http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/73d5bcd0585d/src
- 上面这个还可以查看native方法。

# 1.1 JDK&JRE&JVM

- JDK（Java Development Kit)是针对Java开发员的产品，是整个Java的核心，包括了Java运行环境JRE、Java工具（编译、开发工具)和Java核心类库。
- Java Runtime Environment（JRE)是运行JAVA程序所必须的环境的集合，包含JVM标准实现及Java核心类库。
- JVM是Java Virtual Machine（Java虚拟机)的缩写，是整个java实现跨平台的最核心的部分，能够运行以Java语言写作的软件程序。
- JDK包含JRE和Java编译、开发工具；
- JRE包含JVM和Java核心类库；
- 运行Java仅需要JRE；而开发Java需要JDK。

# 1.2 跨平台

- 字节码是在虚拟机上运行的，而不是编译器。换而言之，是因为JVM能跨平台安装，所以相应JAVA字节码便可以跟着在任何平台上运行。只要JVM自身的代码能在相应平台上运行，即JVM可行，则JAVA的程序员就可以不用考虑所写的程序要在哪里运行，反正都是在虚拟机上运行，然后变成相应平台的机器语言，而这个转变并不是程序员应该关心的。

# 1.3 基础数据类型

- 第一类：整型 byte short int long
- 第二类：浮点型 float double
- 第三类：逻辑型 boolean(它只有两个值可取true false)
- 第四类：字符型 char
    - byte(1)的取值范围为-128~127（-2的7次方到2的7次方-1)
    - short(2)的取值范围为-32768~32767（-2的15次方到2的15次方-1)
    - int(4)的取值范围为（-2147483648~2147483647)（-2的31次方到2的31次方-1)
    - long(8)的取值范围为（-9223372036854774808~9223372036854774807)（-2的63次方到2的63次方-1)
    - float(4)
    - double(8)
    - char(2)
    - boolean(1/8)
- 内码是程序内部使用的字符编码，特别是某种语言实现其char或String类型在内存里用的内部编码；外码是程序与外部交互时外部使用的字符编码。“外部”相对“内部”而言；不是char或String在内存里用的内部编码的地方都可以认为是“外部”。例如，外部可以是序列化之后的char或String，或者外部的文件、命令行参数之类的。
- Java语言规范规定，Java的char类型是UTF-16的code unit，也就是一定是16位（2字节)，然后字符串是UTF-16 code unit的序列。

- Java规定了字符的内码要用UTF-16编码。或者至少要让用户无法感知到String内部采用了非UTF-16的编码。
- String.getBytes()是一个用于将String的内码转换为指定的外码的方法。无参数版使用平台的默认编码作为外码，有参数版使用参数指定的编码作为外码；将String的内容用外码编码好，结果放在一个新byte[]返回。调用了String.getBytes()之后得到的byte[]只能表明该外码的性质，而无法碰触到String内码的任何特质。
    - Java标准库实现的对char与String的序列化规定使用UTF-8作为外码。Java的Class文件中的字符串常量与符号名字也都规定用UTF-8编码。这大概是当时设计者为了平衡运行时的时间效率（采用定长编码的UTF-16)与外部存储的空间效率（采用变长的UTF-8编码)而做的取舍。

# 1.4 引用类型

- 类、接口、数组都是引用类型

## 四种引用

- 目的：避免对象长期占用内存，

### 强引用

```
Object obj = new Object();
```

- StringReference GC时不回收
- 当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

### 软引用

```
String s = new String("Frank");     // 创建强引用与String对象关联，现在该String对象为强可达状态
SoftReference<String> softRef = new SoftReference<String>(s);     // 再创建一个软引用关联该对象
s = null;         // 消除强引用，现在只剩下软引用与其关联，该String对象为软可达状态
s = softRef.get();  // 重新关联上强引用
```

- SoftReference GC时如果JVM内存不足时会回收
- 软引用可用来实现内存敏感的高速缓存。 软引用可以和一个引用队列（ReferenceQueue)联合使用，如果软引用所引用的对象被垃圾回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。如图片缓存框架中缓存图片就是通过软引用实现。

### 弱引用

- WeakReference GC时立即回收
- 弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。
- 弱引用可以和一个引用队列（ReferenceQueue)联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

## 虚引用

- PhantomReference
- 如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。 虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue)联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。
- 在Java集合中有一种特殊的Map类型：WeakHashMap， 在这种Map中存放了键对象的弱引用，当一个键对象被垃圾回收，那么相应的值对象的引用会从Map中删除。WeakHashMap能够节约存储空间，可用来缓存那些非必须存在的数据。

## ReferenceQueue（引用队列）简介

- 当gc（垃圾回收线程）准备回收一个对象时，如果发现它还仅有软引用(或弱引用，或虚引用)指向它，就会在回收该对象之前，把这个软引用（或弱引用，或虚引用）加入到与之关联的引用队列（ReferenceQueue）中。如果一个软引用（或弱引用，或虚引用）对象本身在引用队列中，就说明该引用对象所指向的对象被回收了。
- 当软引用（或弱引用，或虚引用）对象所指向的对象被回收了，那么这个引用对象本身就没有价值了，如果程序中存在大量的这类对象（注意，我们创建的软引用、弱引用、虚引用对象本身是个强引用，不会自动被gc回收），就会浪费内存。因此我们这就可以手动回收位于引用队列中的引用对象本身。
- 除了上面代码展示的创建引用对象的方式。软、弱、虚引用的创建还有另一种方式，即在创建引用的同时关联一个引用队列。

# 基础数据类型包装类

## 为什么需要

- 由于基本数据类型不是对象，所以java并不是纯面向对象的语言，好处是效率较高（全部包装为对象效率较低)。
- Java是一个面向对象的编程语言，基本类型并不具有对象的性质，为了让基本类型也具有对象的特征，就出现了包装类型（如我们在使用集合类型Collection时就一定要使用包装类型而非基本类型)，它相当于将基本类型“包装起来”，使得它具有了对象的性质，并且为其添加了属性和方法，丰富了基本类型的操作。

## 有哪些

| 基本类型 | 包装器类型 |
| --- | --- |
| boolean | Boolean |
| char | Character |
| int | Integer |
| byte | Byte |
| short | Short |
| long | Long |
| float | Float |
| double | Double |

- Number是所有数字包装类的父类

## 自动装箱、自动拆箱（编译器行为）

- 自动装箱：可以将基础数据类型包装成对应的包装类
- Integer i = 10000; // 编译器会改为new Integer(10000)
- 自动拆箱：可以将包装类转为对应的基础数据类型
- int i = new Integer(1000);//编译器会修改为 int i = new Integer(1000).intValue();
- 自动拆箱时如果包装类是null，那么会抛出NPE

## Integer.valueOf

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

- 调用Integer.valueOf时-128~127的对象被缓存起来。
- 所以在此访问内的Integer对象使用==和equals结果是一样的。
- 如果Integer的值一致，且在此范围内，因为是同一个对象，所以==返回true；但此访问之外的对象==比较的是内存地址，值相同，也是返回false。

# 1.5 Object

# == 与 equals的区别

- 如果两个引用类型变量使用==运算符，那么比较的是地址，它们分别指向的是否是同一地址的对象。结果一定是false，因为两个对象不可能存放在同一地址处。
- 要求是两个对象都不是能空值，与空值比较返回false。
- ==不能实现比较对象的值是否相同。
- 所有对象都有equals方法，默认是Object类的equals，其结果与==一样。
- 如果希望比较对象的值相同，必须重写equals方法。

# hashCode与equals的区别

- Object中的equals:

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

- equals 方法要求满足：
- 自反性 a.equals(a)
- 对称性 x.equals(y)  y.equals(x)
- 一致性 x.equals(y)  多次调用结果一致
- 对于任意非空引用x，x.equals(null) 应该返回false
- Object中的hashCode:

```
public native int hashCode();
```

- 它是一个本地方法，它的实现与本地机器有关，这里我们暂且认为他返回的是对象存储的物理位置。

- 当equals方法被重写时，通常有必要重写hashCode方法，以维护hashCode方法的常规约定：值相同的对象必须有相同的hashCode。
    - object1.equals(object2)为true，hashCode也相同；
    - hashCode不同时，object1.equals(object2)为false；
    - hashCode相同时，object1.equals(object2)不一定为true；
- 当我们向一个Hash结构的集合中添加某个元素，集合会首先调用hashCode方法，这样就可以直接定位它所存储的位置，若该处没有其他元素，则直接保存。若该处已经有元素存在，就调用equals方法来匹配这两个元素是否相同，相同则不存，不同则链到后面（如果是链地址法)。
- 先调用hashCode，唯一则存储，不唯一则再调用equals，结果相同则不再存储，结果不同则散列到其他位置。因为hashCode效率更高（仅为一个int值)，比较起来更快。
- HashMap#put源码
- hash是key的hash值，当该hash对应的位置已有元素时会执行以下代码（hashCode相同)
- if (p.hash == hash &&
  ((k = p.key) == key || (key != null && key.equals(k))))
  e = p;
- 如果equals返回结果相同，则值一定相同，不再存入。

## 如果重写equals不重写hashCode会怎样

- 两个值不同的对象的hashCode一定不一样，那么执行equals，结果为true，HashSet或HashMap的键会放入值相同的对象。

# 1.6 String&StringBuffer&StringBuilder

这里仅涉及部分题目，更下面有详细解释。

- 都是final类，不允许继承；
- String长度不可变，StringBuffer、StringBuilder长度可变；

## String

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {}
```

### equals&hashCode

- String重写了Object的hashCode和equals。

  Equals是根据每个char进行比较。Hashcode是把每个char加上一个int。

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
```

```
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

## 添加功能

- String是final类，不可被继承，也不可重写一个java.lang.String（类加载机制）。
- 一般是使用StringUtils来增强String的功能。
- 为什么只加载系统通过的java.lang.String类而不加载用户自定义的java.lang.String类呢?
    - **双亲委派机制**（先向上查找，再向下委派）
    - 因加载某个类时，优先使用父类加载器加载需要使用的类。如果我们自定义了java.lang.String这个类，
    - 加载该自定义的String类，该自定义String类使用的加载器是AppClassLoader，根据优先使用父类加载器原理，
    - AppClassLoader加载器的父类为ExtClassLoader，所以这时加载String使用的类加载器是ExtClassLoader，
    - 但是类加载器ExtClassLoader在jre/lib/ext目录下没有找到String.class类。然后使用ExtClassLoader父类的加载器BootStrap，
    - 父类加载器BootStrap在JRE/lib目录的rt.jar找到了String.class，将其加载到内存中。这就是类加载器的委托机制。
    - 所以，用户自定义的java.lang.String不被加载，也就是不会被使用。
    - 如果没有任何加载器能加载，就会抛出**ClassNotFoundException**
    - 这种父子关系一般通过**组合（Composition）关系**来实现，而不是通过继承（Inheritance）。
    - 参考：https://www.cnblogs.com/luckforefforts/p/13642685.html

Class 文件来源

```
1. Java自己的核心类 如 java.lang、java.math、java.io 等 package 内部的类，位于 $JAVA_HOME/jre/lib/ 目录下，如 java.lang.String 类就是定义在 $JAVA_HOME/jre/lib/rt.jar 文件里；
2. Java的核心扩展类，位于 $JAVA_HOME/jre/lib/ext 目录下。开发者也可以把自己编写的类打包成 jar 文件放入该目录下；
3. 开发者自己写的类，这些类位于项目目录下；
4. 动态加载远程的 .class 文件
```

Java的类加载器

针对上面四种来源的类，分别有不同的加载器负责加载。

> 级别最高的 Java 核心类 ，即$JAVA_HOME/jre/lib 里的核心 jar 文件。这些类是 Java 运行的基础类，由一个名为 BootstrapClassLoader 加载器负责加载，它也被称作 根加载器 / 引导加载器。注意，BootstrapClassLoader 比较特殊，它不继承 ClassLoader，而是由 JVM 内部实现
> 加载 Java 核心扩展类 ，即 $JAVA_HOME/jre/lib/ext 目录下的 jar 文件。这些文件由 ExtensionClassLoader 负责加载，它也被称作 扩展类加载器。当然，用户如果把自己开发的 jar 文件放在这个目录，也会被 ExtClassLoader 加载；
> 开发者在项目中编写的类，这些文件将由 AppClassLoader 加载器进行加载，它也被称作 系统类加载器 System ClassLoader
> 如果想远程加载如（本地文件 / 网络下载）的方式，则必须要自己自定义一个 ClassLoader，复写其中的 findClass() 方法才能得以实现

————————————————————


## + substring

- 会创建一个新的字符串；
- 编译时会将+转为StringBuilder的append方法。
- 注意新的字符串是在运行时在**堆**里创建的。
- String str1 = “ABC”;可能创建一个或者不创建对象，如果”ABC”这个字符串在java String池里不存在，会在java String池里创建一个创建一个String对象(“ABC”)，然后str1指向这个内存地址，无论以后用这种方式创建多少个值为”ABC”的字符串对象，始终只有一个内存地址被分配，之后的都是String的拷贝，Java中称为“字符串驻留”，所有的字符串常量都会在编译之后自动地驻留。但是如果使用new了，会在堆里额外创建一个。
- 重点：String str0 = "abc";执行后会在String pool里存一个“abc”，str0引用这个pool里的“abc”。String str1 = new String("abc");执行，首先程序会到String pool里看有没有“abc”，没有的话，创建一个放到pool里，然后再堆里再创建一个“abc”，这个str1引用堆里的对象。此题因为String pool里有一个“abc”了，所以直接在堆里创建一个“abc”（每次new都会创建新对象），str1还是引用堆里的对象。所以str0和str1引用的对象地址不一样，一个在pool里，一个在堆里，所以执行System.out.println(str0==str1);输出false。

  https://www.cnblogs.com/chenzufeng/p/14515367.html
- 重点：练习题例子：https://blog.csdn.net/qq_41376740/article/details/80338158
  - 从==来了解常量池和自动拆装箱 1.当语句只有==时，比较的是地址： （1）当直接string字符串是在常量池中创建，newstring是在堆中创建，自然地址不等 （2）有的封装类有缓冲器，如integer有-127-128的缓冲区，在这个范围类只要不new对象都是在常量池中创建 （3）如果==和运算符都在语句中出现，那么包装类会出现自动拆装包，这个时候就算是堆中数和常量池数运算也是true

- 注意只有字符串常量是共享的，+和substring等操作的结果不是共享的，substring也会在堆中重新创建字符串。

```
public String substring(int beginIndex, int endIndex) {
    if (beginIndex < 0) {
        throw new StringIndexOutOfBoundsException(beginIndex);
    }
    if (endIndex > value.length) {
        throw new StringIndexOutOfBoundsException(endIndex);
    }
```

```
    int subLen = endIndex - beginIndex;
    if (subLen < 0) {
        throw new StringIndexOutOfBoundsException(subLen);
    }
    return ((beginIndex == 0) && (endIndex == value.length)) ? this
            : new String(value, beginIndex, subLen);
}
```

```
public String(char value[], int offset, int count) {
    if (offset < 0) {
        throw new StringIndexOutOfBoundsException(offset);
    }
    if (count <= 0) {
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        if (offset <= value.length) {
            this.value = "".value;
            return;
        }
    }
    // Note: offset or count might be near -1>>>1.
    if (offset > value.length - count) {
        throw new StringIndexOutOfBoundsException(offset + count);
    }
    this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

## 常量池

- String str = new String(“ABC”);
- 至少创建一个对象，也可能两个。因为用到new关键字，肯定会在heap中创建一个str2的String对象，它的value是“ABC”。同时如果这个字符串在字符串常量池里不存在，会在池里创建这个String对象“ABC”。
- String s1= “a”;
- String s2 = “a”;
- 此时s1 == s2 返回true
- String s1= new String(“a”);
- String s2 = new String(“a”);
- 此时s1 == s2 返回false
- ""创建的字符串在字符串池中。
- 如果引号中字符串存在在常量池中，则仅在堆中拷贝一份(new String);
- 如果不在，那么会先在常量池中创建一份("abc")，然后在堆中创建一份(new String)，共创建两个对象。
- **内存划分**：https://www.cnblogs.com/wangshen31/p/10404353.html

## 编译优化

https://www.cnblogs.com/myitnews/p/11457649.html（编译期与运行期的区别解释）

https://blog.csdn.net/Megustas_JJC/article/details/52673509（编译期运行期举例说明）

https://www.cnblogs.com/wangshen31/p/10404353.html

**总结下就是：**

**两个或者两个以上的字符串常量相加，在预编译的时候“+”会被优化，相当于把两个或者两个以上字符串常量自动合成一个字符串常量.**

**字符串常量相加，不会用到StringBuilder对象，有一点要注意的是：字符串常量和字符串是不同的概念，字符串常量储存于常量池，而字符串储存于堆(heap)。**

- 字面量，final 都会在编译期被优化，并且会被直接运算好。
-
    - 1)注意c和d中，final变量b已经被替换为其字符串常量了。
    - 2)注意f、g中，b被替换为其字符串常量，并且在编译时字符串常量的+运算会被执行，返回拼接后的字符串常量
    - 3)注意j，a1作为final变量，在编译时被替换为其字符串常量
- 解释 c == h / d == h/ e== h为false：c是运行时使用+拼接，创建了一个新的堆中的字符串ab，与ab字符串常量不是同一个对象；
- 解释f == h/ g == h为true：f编译时进行优化，其值即为字符串常量ab，h也是，指向字符串常量池中的同一个对象；

- String#intern（JDK1.7之后）
- JDK1.7之后JVM里字符串常量池放入了堆中，之前是放在方法区。
- intern()方法设计的初衷，就是重用String对象，以节省内存消耗。
- 一定是new得到的字符串才会调用intern，字符串常量没有必要去intern。
- 当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（该对象由 equals(Object) 方法确定），则返回池中的字符串。否则，常量池中直接存储堆中该字符串的引用（1.7之前是常量池中再保存一份该字符串）。
- 源码

```
public native String intern();
```

- 实例一：

    String s = new String("1");
    s.intern();
    String s2 = "1";
    System.out.println(s == s2);// false

String s3 = new String("1") + new String("1");
s3.intern();
String s4 = "11";
System.out.println(s3 == s4);// true

- String s = newString("1")，生成了常量池中的“1” 和堆空间中的字符串对象。
- s.intern()，这一行的作用是s对象去常量池中寻找后发现"1"已经存在于常量池中了。
- String s2 = "1"，这行代码是生成一个s2的引用指向常量池中的“1”对象。
- 结果就是 s 和 s2 的引用地址明显不同。因此返回了false。
- String s3 = new String("1") + newString("1")，这行代码在字符串常量池中生成“1” ，并在堆空间中生成s3引用指向的对象（内容为"11"）。注意此时常量池中是没有 “11”对象的。这用到的是stringbuffer的append。
- s3.intern()，这一行代码，是将 s3中的“11”字符串放入 String 常量池中，此时常量池中不存在“11”字符串，JDK1.6的做法是直接在常量池中生成一个 "11" 的对象。
- 但是在JDK1.7中，常量池中不需要再存储一份对象了，**可以直接存储堆中的引用**。这份引用直接指向 s3 引用的对象，也就是说s3.intern() ==s3会返回true。

- String s4 = "11"， 这一行代码会直接去常量池中创建，但是发现已经有这个对象了，此时也就是指向 s3 引用对象的一个引用。因此s3 == s4返回了true。
- 实例二：
    - String s3 = new String("1") + new String("1");
      String s4 = "11";
      s3.intern();
      System.out.println(s3 == s4);// false
- String s3 = new String("1") + newString("1")，这行代码在字符串常量池中生成“1”，并在堆空间中生成s3引用指向的对象（内容为"11"）。注意此时常量池中是没有 “11”对象的。
- String s4 = "11"， 这一行代码会直接去生成常量池中的"11"。
- s3.intern()，这一行在这里就没什么实际作用了。因为"11"已经存在了。
- 结果就是 s3 和 s4 的引用地址明显不同。因此返回了false。
- 实例三：
    - String str1 = new String("SEU") + new String("Calvin");
      System.out.println(str1.intern() == str1);// true
      System.out.println(str1 == "SEUCalvin");// true
- str1.intern() == str1就是上面例子中的情况，str1.intern()发现常量池中不存在“SEUCalvin”，因此指向了str1。 "SEUCalvin"在常量池中创建时，也就直接指向了str1了。两个都返回true就理所当然啦。
- 实例四：
    - String str2 = "SEUCalvin";//新加的一行代码，其余不变
      String str1 = new String("SEU") + new String("Calvin");
      System.out.println(str1.intern() == str1);// false
      System.out.println(str1 == "SEUCalvin");// false
- 在实例三的基础上加了第一行
- str2先在常量池中创建了“SEUCalvin”，那么str1.intern()当然就直接指向了str2，你可以去验证它们两个是返回的true。后面的"SEUCalvin"也一样指向str2。所以谁都不搭理在堆空间中的str1了，所以都返回了false。
- 区分字符串常量和字符串变量（"XXX"：常量）（s="XXX"，s变量）。常量相加和带变量的相加不一样。

## StringBuffer&StringBuilder

- StringBuffer是线程安全的，StringBuilder不是线程安全的，但它们两个中的所有方法都是相同的。StringBuffer在StringBuilder的方法之上添加了synchronized，保证线程安全。
- StringBuilder比StringBuffer性能更好。

# 1.7 面向对象

## 抽象类与接口

https://www.cnblogs.com/justForMe/archive/2010/09/16/1828009.html（比喻）

- 区别：
    - 1)抽象类中方法可以不是抽象的；接口中的方法必须是抽象方法；
    - 2)抽象类中可以有普通的成员变量；接口中的变量必须是 static final 类型的，必须被初始化，接口中只有常量，没有变量。
    - 3)抽象类只能单继承，接口可以继承多个父接口；

    - 4)Java8 中接口中会有 default 方法，即方法可以被实现。
- 使用场景：
- 如果要创建不带任何方法定义和成员变量的基类，那么就应该选择接口而不是抽象类。
- 如果知道某个类应该是基类，那么第一个选择的应该是让它成为一个接口，只有在必须要有方法定义和成员变量的时候，才应该选择抽象类。因为抽象类中允许存在一个或多个被具体实现的方法，只要方法没有被全部实现该类就仍是抽象类。

## 三大特性

- 面向对象的三个特性：封装；继承；多态
- 封装：将数据与操作数据的方法绑定起来，隐藏实现细节，对外提供接口。
- 继承：代码重用；可扩展性
- 多态：允许不同子类对象对同一消息做出不同响应
- 多态的三个必要条件：继承、方法的重写、父类引用指向子类对象

## 重写和重载

- 根据对象对方法进行选择，称为分派
- 编译期的静态多分派：overloading重载 根据调用引用类型和方法参数决定调用哪个方法（编译器）
- 运行期的动态单分派：overriding 重写 根据指向对象的类型决定调用哪个方法（JVM）
-

# 1.8 关键类

## ThreadLocal（线程局部变量）

https://blog.csdn.net/weixin_44050144/article/details/113061884

ThreadLocal通常都定义为static，ThreadLocal没有存储功能，变量副本的真实存储位置是Thread对象的threadLocals这个ThreadLocal.ThreadLocalMap变量中，可以将ThreadLocal理解为一个工具类，用来保证线程本地变量的存储和存储碰撞。

- 在线程之间共享变量是存在风险的，有时可能要避免共享变量，使用ThreadLocal辅助类为各个线程提供各自的实例。
- 例如有一个静态变量

```
public static final SimpleDateFormat sdf = new SimpleDateFormat(“yyyy-MM-dd”);
```

- 如果两个线程同时调用sdf.format(...)
- 那么可能会很混乱，因为sdf使用的内部数据结构可能会被并发的访问所破坏。当然可以使用线程同步，但是开销很大；或者也可以在需要时构造一个局部SImpleDateFormat对象。但这很浪费。
- 希望为每一个线程构造一个对象，即使该线程调用多次方法，也只需要构造一次，不必在局部每次都构造。

```
public static final ThreadLocal<SimpleDateFormat> sdf = new
ThreadLocal<SimpleDateFormat>() {
    @Override
    protected SimpleDateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

- 实现原理：

## 1)每个线程的变量副本是存储在哪里的

- ThreadLocal的get方法就是从当前线程的ThreadLocalMap中取出当前线程对应的变量的副本。该Map的key是ThreadLocal对象，value是当前线程对应的变量。

```
public T get() {
    Thread t = Thread.currentThread();-
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

- 
- 【注意，变量是保存在线程中的，而不是保存在ThreadLocal变量中】。当前线程中，有一个变量引用名字是threadLocals，这个引用是在ThreadLocal类中createmap函数内初始化的。
- void createMap(Thread t, T firstValue) {
  t.threadLocals = new ThreadLocalMap(this, firstValue);
  }
- 每个线程都有一个这样的名为threadLocals 的ThreadLocalMap，以ThreadLocal和ThreadLocal对象声明的变量类型作为key和value。
- Thread
- ThreadLocal.ThreadLocalMap threadLocals = null;
- 这样，我们所使用的ThreadLocal变量的实际数据，通过get方法取值的时候，就是通过取出Thread中threadLocals引用的map，然后从这个map中根据当前threadLocal作为参数，取出数据。现在，变量的副本从哪里取出来的（本文章提出的第一个问题)已经确认解决了。
- 每个线程内部都会维护一个类似 HashMap 的对象，称为 ThreadLocalMap，里边会包含若干了 Entry（K-V 键值对），相应的线程被称为这些 Entry 的属主线程；
- Entry 的 Key 是一个 ThreadLocal 实例，Value 是一个线程特有对象。Entry 的作用即是：为其属主线程建立起一个 ThreadLocal 实例与一个线程特有对象之间的对应关系；
- Entry 对 Key 的引用是弱引用；Entry 对 Value 的引用是强引用。

## 2)为什么ThreadLocalMap的Key是弱引用

- 如果是强引用，ThreadLocal将无法被释放内存。
- 因为如果这里使用普通的key-value形式来定义存储结构，实质上就会造成节点的生命周期与线程强绑定，只要线程没有销毁，那么节点在GC分析中一直处于可达状态，没办法被回收，而程序本身也无法判断是否可以清理节点。弱引用是Java中四档引用的第三档，比软引用更加弱一些，如果一个对象没有强引用链可达，那么一般活不过下一次GC。当某个ThreadLocal已经没有强引用可达，则随着它被垃圾回收，在ThreadLocalMap里对应的Entry的键值会失效，这为ThreadLocalMap本身的垃圾清理提供了便利。

## 3)ThreadLocalMap是何时初始化的（setInitialValue)

这一段有点问题，不理。

- 在get时最后一行调用了setInitialValue，它又调用了我们自己重写的initialValue方法获得要线程局部变量对象。ThreadLocalMap没有被初始化的话，便初始化，并设置firstKey和firstValue；如果已经被初始化，那么将key和value放入map。

```
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

## 4)ThreadLocalMap 原理

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

- 它也是一个类似HashMap的数据结构，但是并没实现Map接口。
- 也是初始化一个大小16的Entry数组，Entry对象用来保存每一个key-value键值对，只不过这里的key永远都是ThreadLocal对象，通过ThreadLocal对象的set方法，结果把ThreadLocal对象自己当做key，放进了ThreadLoalMap中。
- ThreadLoalMap的Entry是继承WeakReference，和HashMap很大的区别是，Entry中没有next字段，所以就不存在链表的情况了。

### 构造方法

- ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    - // 表的大小始终为2的幂次
      table = new Entry[INITIAL_CAPACITY];
      int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
      table[i] = new Entry(firstKey, firstValue);
      size = 1;
- // 设定扩容阈值
  setThreshold(INITIAL_CAPACITY);
  }
    - 在ThreadLocalMap中，形如key.threadLocalHashCode & (table.length - 1)（其中key为一个ThreadLocal实例)这样的代码片段实质上就是在求一个ThreadLocal实例的哈希值，只是在源码实现中没有将其抽为一个公用函数。

    - 对于& (INITIAL_CAPACITY - 1)，相对于2的幂作为模数取模，可以用&(2^n-1)来替代%2^n，位运算比取模效率高很多。至于为什么，因为对2^n取模，只要不是低n位对结果的贡献显然都是0，会影响结果的只能是低n位。

```
private void setThreshold(int len) {
    threshold = len * 2 / 3;
}
```

- getEntry（由ThreadLocal#get调用)

```
private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
    // 因为用的是线性探测，所以往后找还是有可能能够找到目标Entry的。
        return getEntryAfterMiss(key, i, e);
}
```

线性探测：解决hash冲突的一种办法，如果计算出来的位置已经有元素，则一个一个向后找，有点笨。

```
private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {
    Entry[] tab = table;
    int len = tab.length;

    while (e != null) {
        ThreadLocal<?> k = e.get();
        if (k == key)
            return e;
        if (k == null)
            // 该entry对应的ThreadLocal已经被回收，调用expungeStaleEntry来清理无效的
entry
            expungeStaleEntry(i);
      else
            i = nextIndex(i, len);
            e = tab[i];
    }
    return null;
  }
```

- 
- i是位置
- 从staleSlot开始遍历，将无效key（弱引用指向对象被回收)清理，即对应entry中的value置为null，将指向这个entry的table[i]置为null，直到扫到空entry。
- 另外，在过程中还会对非空的entry作rehash。
- 可以说这个函数的作用就是从staleSlot开始清理连续段中的slot（断开强引用，rehash slot等)

```
private int expungeStaleEntry(int staleSlot) {
    Entry[] tab = table;
    int len = tab.length;

    // expunge entry at staleSlot
    tab[staleSlot].value = null;
```

```
    tab[staleSlot] = null;
    size--;

    // Rehash until we encounter null
    Entry e;
    int i;
    for (i = nextIndex(staleSlot, len);
         (e = tab[i]) != null;
         i = nextIndex(i, len)) {
        ThreadLocal<?> k = e.get();
        if (k == null) {
            e.value = null;
            tab[i] = null;
            size--;
        } else {
```

- // 对于还没有被回收的情况，需要做一次rehash。
- 如果对应的ThreadLocal的ID对len取模出来的索引h不为当前位置i，
  - 则从h向后线性探测到第一个空的slot，把当前的entry给挪过去。
    int h = k.threadLocalHashCode & (len - 1);
    if (h != i) {
      tab[i] = null;

    ```
    // Unlike Knuth 6.4 Algorithm R, we must scan until
    // null because multiple entries could have been stale.
    while (tab[h] != null)
        h = nextIndex(h, len);
    tab[h] = e;
    ```

    }
    }
      }
      return i;
      }

## set（线性探测法解决hash冲突)

```
private void set(ThreadLocal<?> key, Object value) {

    // We don't use a fast path as with get() because it is at
    // least as common to use set() to create new entries as
    // it is to replace existing ones, in which case, a fast
    // path would fail more often than not.

    Entry[] tab = table;
    int len = tab.length;
    // 计算key的hash值
```

```
- int i = key.threadLocalHashCode & (len-1);
for (Entry e = tab[i];
     e != null;
     e = tab[i = nextIndex(i, len)]) {
    ThreadLocal<?> k = e.get();

    if (k == key) {
```

- // 同一个ThreadLocal赋了新值，则替换原值为新值
  e.value = value;
  return;
  }

  if (k == null) {
- // 该位置的TheadLocal已经被回收，那么会清理slot并在此位置放入当前key和value（stale：陈旧的)
  replaceStaleEntry(key, value, i);
  return;
  }
  }
  // 下一个位置为空，那么就放到该位置上
  tab[i] = new Entry(key, value);
  int sz = ++size;
- // 启发式地清理一些slot,并判断是否是否需要扩容
  if (!cleanSomeSlots(i, sz) && sz >= threshold)
  rehash();
  }
- 每个ThreadLocal对象都有一个hash值 threadLocalHashCode，每初始化一个ThreadLocal对象，hash值就增加一个固定的大小 0x61c88647。

```
private final int threadLocalHashCode = nextHashCode();
```

```
private static final int HASH_INCREMENT = 0x61c88647;
```

```
private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}
```

- 由于ThreadLocalMap使用线性探测法来解决散列冲突，所以实际上Entry[]数组在程序逻辑上是作为一个环形存在的。

```
private static int nextIndex(int i, int len) {
    return ((i + 1 < len) ? i + 1 : 0);
}
```

- 在插入过程中，根据ThreadLocal对象的hash值，定位到table中的位置i，过程如下：
- 1、如果当前位置是空的，那么正好，就初始化一个Entry对象放在位置i上；
- 2、不巧，位置i已经有Entry对象了，如果这个Entry对象的key正好是即将设置的key，那么重新设置Entry中的value；
- 3、很不巧，位置i的Entry对象，和即将设置的key没关系，那么只能找下一个空位置；

- 这样的话，在get的时候，也会根据ThreadLocal对象的hash值，定位到table中的位置，然后判断该位置Entry对象中的key是否和get的key一致，如果不一致，就判断下一个位置
- 可以发现，set和get如果冲突严重的话，效率很低，因为ThreadLoalMap是Thread的一个属性，所以即使在自己的代码中控制了设置的元素个数，但还是不能控制其它代码的行为。

## cleanSomeSlots（启发式地清理slot)

- i是当前位置，n是元素个数
- i对应entry是非无效（指向的ThreadLocal没被回收，或者entry本身为空)
- n是用于控制控制扫描次数的
- 正常情况下如果log n次扫描没有发现无效slot，函数就结束了
- 但是如果发现了无效的slot，将n置为table的长度len，做一次连续段的清理
- 再从下一个空的slot开始继续扫描
- 这个函数有两处地方会被调用，一处是插入的时候可能会被调用，另外个是在替换无效slot的时候可能会被调用，
- 区别是前者传入的n为元素个数，后者为table的容量

```
private boolean cleanSomeSlots(int i, int n) {
    boolean removed = false;
    Entry[] tab = table;
    int len = tab.length;
    do {
        i = nextIndex(i, len);
        Entry e = tab[i];
        if (e != null && e.get() == null) {
            n = len;
            removed = true;
            i = expungeStaleEntry(i);
        }
    } while ( (n >>>= 1) != 0);
    return removed;
}
```

## rehash

- 先全量清理，如果清理后现有元素个数超过负载，那么扩容

```
private void rehash() {
```

```
- // 进行一次全量清理
expungeStaleEntries();

// Use lower threshold for doubling to avoid hysteresis
if (size >= threshold - threshold / 4)
    resize();
```

}

- 全量清理

```
private void expungeStaleEntries() {
    Entry[] tab = table;
    int len = tab.length;
    for (int j = 0; j < len; j++) {
        Entry e = tab[j];
        if (e != null && e.get() == null)
            expungeStaleEntry(j);
    }
}
```

- 扩容，因为需要保证table的容量len为2的幂，所以扩容即扩大2倍

```
private void resize() {
    Entry[] oldTab = table;
    int oldLen = oldTab.length;
    int newLen = oldLen * 2;
    Entry[] newTab = new Entry[newLen];
    int count = 0;

    for (int j = 0; j < oldLen; ++j) {
        Entry e = oldTab[j];
        if (e != null) {
            ThreadLocal<?> k = e.get();
            if (k == null) {
                e.value = null; // Help the GC
            } else {
                int h = k.threadLocalHashCode & (newLen - 1);
                while (newTab[h] != null)
                    h = nextIndex(h, newLen);
                newTab[h] = e;
                count++;
            }
        }
    }

    setThreshold(newLen);
    size = count;
    table = newTab;
}
```

## remove

```
private void remove(ThreadLocal<?> key) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        if (e.get() == key) {
```

- // 显式断开弱引用
  e.clear();
- // 进行段清理
  expungeStaleEntry(i);

```
    return;
}
}
  }
```

- Reference#clear

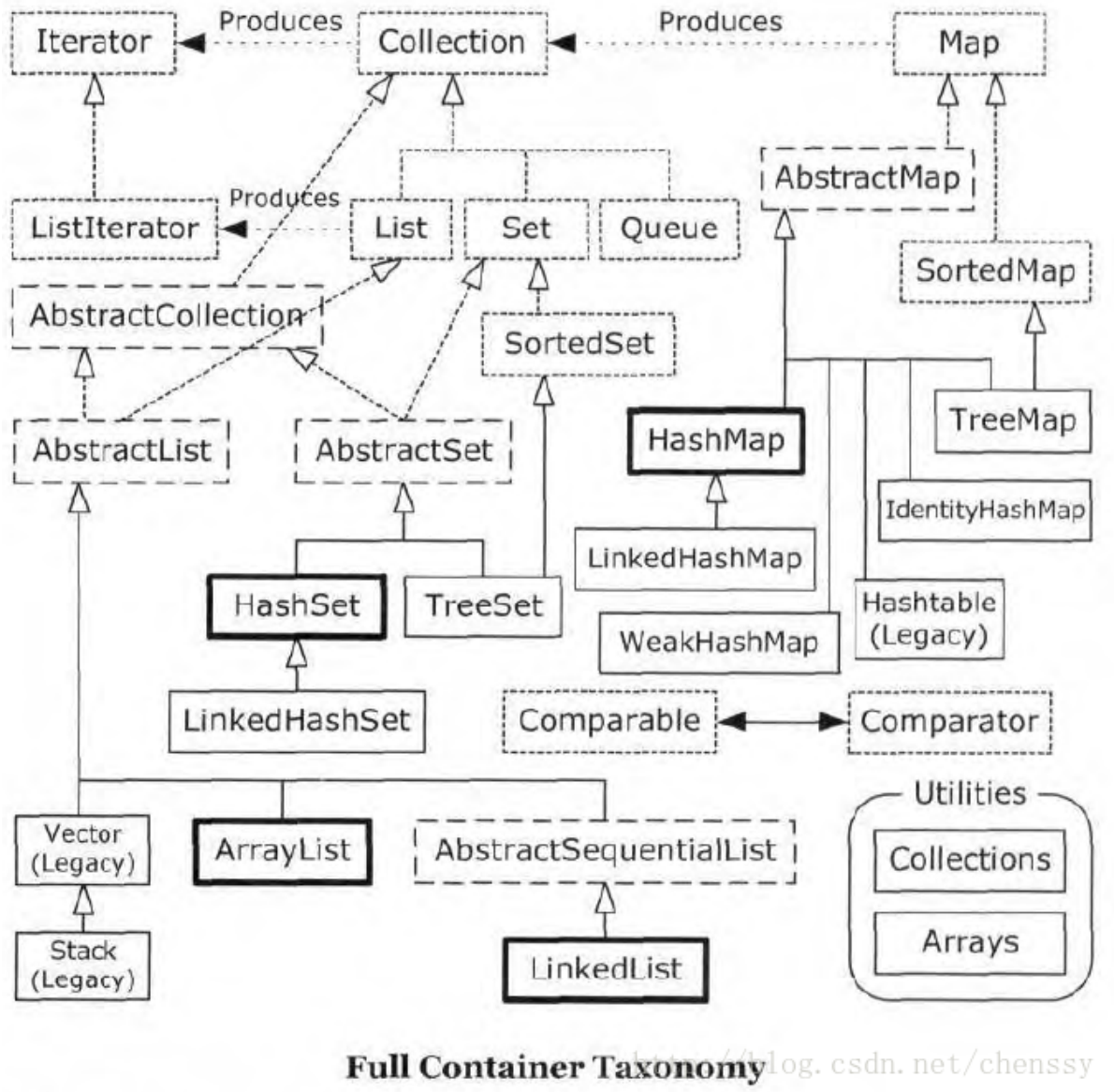
```
public void clear() {
    this.referent = null;
}
```

### 内存泄露

- 只有调用TheadLocal的remove或者get、set时才会采取措施去清理被回收的ThreadLocal对应的value（但也未必会清理所有的需要被回收的value)。假如一个局部的ThreadLocal不再需要，如果没有去调用remove方法清除，那么有可能会发生内存泄露。
- 既然已经发现有内存泄露的隐患，自然有应对的策略，在调用ThreadLocal的get()、set()可能会清除ThreadLocalMap中key为null的Entry对象，这样对应的value就没有GC Roots可达了，下次GC的时候就可以被回收，当然如果调用remove方法，肯定会删除对应的Entry对象。
- 如果使用ThreadLocal的set方法之后，没有显式的调用remove方法，就有可能发生内存泄露，所以养成良好的编程习惯十分重要，使用完ThreadLocal之后，记得调用remove方法。

> JDK建议将`ThreadLocal`变量定义成`private static`的，这样的话`ThreadLocal`的生命周期就更长，由于一直存在`ThreadLocal`的强引用，所以`ThreadLocal`也就不会被回收，也就能保证任何时候都能根据`ThreadLocal`的弱引用访问到`Entry`的`value`值，然后`remove`它，防止内存泄露。
> 这里的意思是，反正设置成弱引用也是会泄露的，那不如直接强引用保证能找到它们，到时候一起`remove`。

# Collection




**Full Container Taxonomy**



- https://blog.csdn.net/u011240877/category_6447444.html（collection框架详解）
- https://blog.csdn.net/qq_28306343（不只有collection）
- https://blog.csdn.net/liudong220?t=1

## Collection

1. fail-fast
2. modcount

## Iterator、ListIterator

1. Iterator及ListIterator两个都是接口，接口是什么？**一套规范，没有标准实现（default方法除外）；**由实现类去实现他们。
2. Iterator规范的是如何查找下一个元素。
   ListIterator则在Iterator的基础上，增加了索引的概念，增加了逆向查询方法。
3. 我们在使用 Iterator 对容器进行迭代时如果修改容器 可能会报 *ConcurrentModificationException* 的错。官方称这种情况下的迭代器是 *fail-fast* 迭代器。当你调用 iterator 方法时，返回的迭代器会记住当前的 modCount，随后迭代过程中会检查这个值，一旦发现这个值发生变化，就说明你对容器做了修改，就会抛异常。

modCount表示集合的元素被修改的次数，每次增加或删除一个元素的时候，modCount都会加一，而expectedModCount用于记录在集合遍历之前的modCount，检查这两者是否相等就是为了检查集合在迭代遍历的过程中有没有被修改，如果被修改了，就会在运行时抛出ConcurrentModificationException这个RuntimeException，以提醒开发者集合已经被修改。
这就说明了为什么集合在使用Iterator进行遍历的时候不能使用集合本身的add或者remove方法来增减元素。但是使用Iterator的remove方法是可以的。

**remove方法删除的元素是指针指向的元素。**如果当前指针指向的内存中没有元素，那么会抛出异常。所以不能连续remove，要先next。

4. 迭代器在不使用next()方法情况下进行remove操作会出错。
5. Iterator 被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象。

# List

# AbstractCollection

https://blog.csdn.net/liudong220/article/details/105411754/?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-0.no_search_link&spm=1001.2101.3001.4242.1

1. 为什么里面的add要抛出异常UnsupportedOperationException而不是用抽象方法呢？因为程序员可能会选择实现一个不可修改的集合，如果要实现一个可修改的集合，就重写就行。有点像接口隔离原则：客户不要的方法不暴露给他。

# ArrayList

1. 线程不安全

   https://blog.csdn.net/u012859681/article/details/78206494

2. ArrayList底层是由数组组成的，初始化是一个空数组（这个版本之间差异较大），容器最大容量为Integer.MAX_VALUE - 8（2147483639）；
   在jdk1.8以前，是默认数组大小为10的一个数组，1.8以后时第一次添加操作后扩容时改变大小；（如果添加元素数量小于10，数组大小则为10）；
3. 扩容时会先判断是否需要扩容，需要则扩1.5倍
4. 删除不缩容：个人认为这是因为每次删除都去调整大小是很浪费性能的表现，还不如把这种操作交给用户去判断何时操作；
   假设当前操作是删除操作，下一次是增加操作；
   如果我们在删除操作时进行了缩容操作，下一次增加的时候，我们又要进行扩容操作，这样非常浪费了性能。
5. Arrays的内部类ArrayList和java.util.ArrayList都是继承AbstractList，remove、add等方法AbstractList中是默认throw UnsupportedOperationException而且不作任何操作。
   java.util.ArrayList重新了这些方法，而Arrays的内部类ArrayList没有重写，所以会抛出异常。
6. ArrayList实现了一个叫做 `RandomAccess` 的接口，而 LinkedList 是没有的。RandomAccess 是一个标志接口，表明实现这个这个接口的 List 集合是支持快速随机访问的。也就是说，实现了这个接口的集合是支持 **快速随机访问** 策略的。

   同时，官网还特意说明了，如果是实现了这个接口的 **List**，那么使用for循环的方式获取数据会优于用迭代器获取数据。

## AbstractSequentialList

1. AbstractSequentialList 继承自 AbstractList，是 LinkedList 的父类，是 List 接口 的简化版实现。

   简化在哪儿呢？简化在 AbstractSequentialList 只支持按次序访问，而不像 AbstractList 那样支持随机访问。

2. 支持 RandomAccess 的对象，遍历时使用 get 比 迭代器更快。
   而 AbstractSequentialList 只支持迭代器按顺序 访问，不支持 RandomAccess，所以遍历 AbstractSequentialList 的子类，使用 for 循环 get() 的效率要 <= 迭代器遍历。

3. get调用的是listiterator(index).next()

## Queue

1. 单队列"**假溢出**"
2. 循环队列中，rear = (rear - size) % size
3. 为了达到判断队列状态的目的，可以通过牺牲一个存储空间来实现。 放满标志 (rear - front) % size = 1
4. add、offer等的区别要搞清楚
5. Queue 是个接口，它提供的 add, offer 方法初衷是希望子类能够禁止添加元素为 null，这样可以避免在查询时返回 null 究竟是正确还是错误。
   事实上大多数 Queue 的实现类的确响应了 Queue 接口的规定，比如 ArrayBlockingQueue，PriorityBlockingQueue 等等。
   但还是有一些实现类没有这样要求，比如 LinkedList。
6. 虽然 LinkedList 没有禁止添加 null，但是一般情况下 Queue 的实现类都不允许添加 null 元素，为啥呢？因为 poll(), peek() 方法在异常的时候会返回 null，你添加了 null 以后，当获取时不好分辨究竟是否正确返回。
7. Queue 一般都是 FIFO 的，但是也有例外，比如优先队列 priority queue（它的顺序是根据自然排序或者自定义 comparator 的）；再比如 LIFO 的队列（跟栈一样，后来进去的先出去）。
8. 不论进入、出去的先后顺序是怎样的，使用 remove()，poll() 方法操作的都是 头部 的元素；而插入的位置则不一定是在队尾了，不同的 queue 会有不同的插入逻辑。

## Deque

1. 每个操作都有两种方法，一种在异常情况下直接抛出异常奔溃，另一种则不会抛异常，而是返回特殊的值，比如 false, null ...

| | First Element (Head) | | Last Element (Tail) | |
|---|---|---|---|---|
| | *Throws exception* | *Special value* | *Throws exception* | *Special value* |
| **Insert** | `addFirst(e)` | `offerFirst(e)` | `addLast(e)` | `offerLast(e)` |
| **Remove** | `removeFirst()` | `pollFirst()` | `removeLast()` | `pollLast()` |
| **Examine** | `getFirst()` | `peekFirst()` | `getLast()` | `peekLast()` |

2. LinkedBlockingDeque 如果队列为空时，获取操作将会阻塞，直到有元素添加。

3. 在并发编程 中，双端队列 Deque 还用于 "工作密取" 模式。

## Linkedlist

1. 它既实现了List接口也实现了Deque接口，所以既可以用List声明也可以用Deque声明，即是说存在多种向上转型。用一个接口声明的变量无法使用另一个接口有而这个接口没有的方法，而且自己类里特有的也不行，除非用类声明。

https://dongshuo.blog.csdn.net/article/details/77145673?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-2.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-2.control

2. linkedList 和 ArrayList 一样，不是同步容器。所以需要外部做同步操作，或者直接用 `Collections.synchronizedList` 方法包一下，最好在创建时就包一下。
3. 查找时并不是所有指定位置都是从头部开始的，这个得看下标参数的大小，有的则从尾部开始遍历的。

## Vector

1. Vector 和 ArrayList 一样，都是继承自AbstractList。它是 Stack 的父类。
2. 底层也是个数组
3. 扩容时会扩大 2 倍，而不是 1.5。默认10。
4. Vector 通过 capacity (容量) 和 capacityIncrement (增长数量) 来尽量少的占用空间
5. 通过 iterator 和 lastIterator 获得的迭代器是 fail-fast 的
6. 同步类，每个方法前都有同步锁 synchronized

## Stack

1. 有 5 种创建 Stack 的方法
2. 采用数组实现
3. 除了 push()，剩下的方法都是同步的

## TreeSet

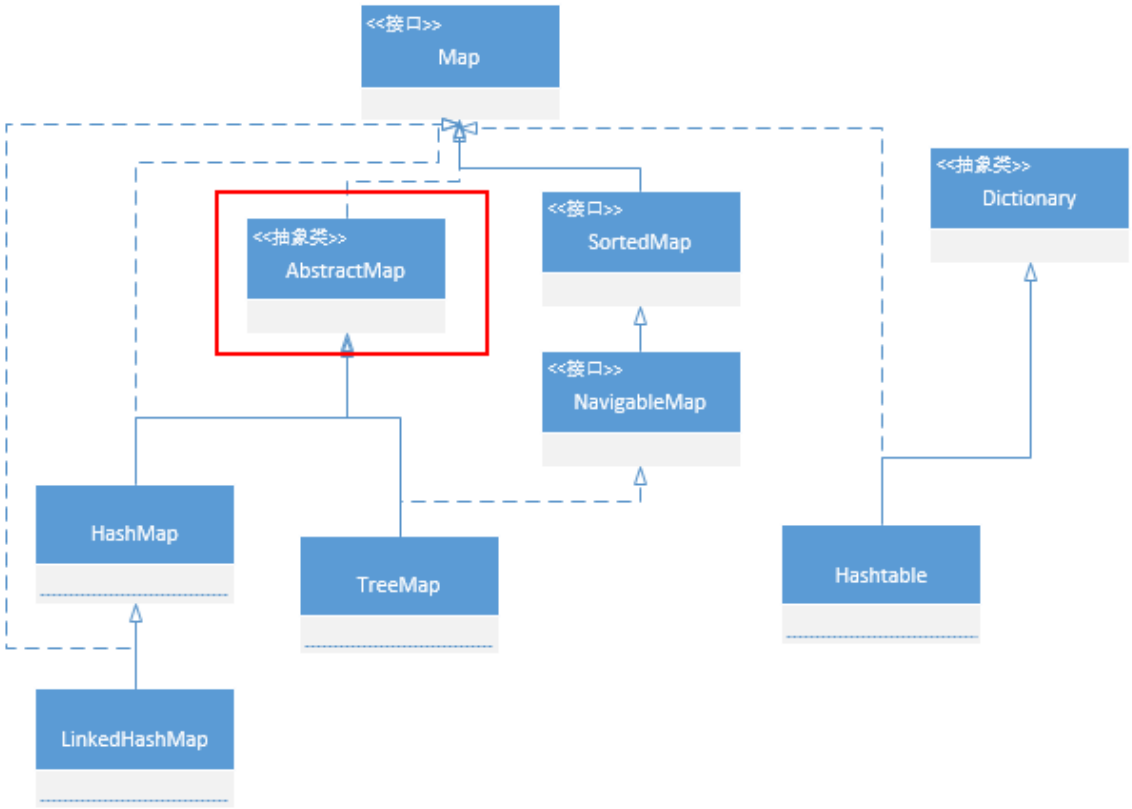
1. TreeSet是基于TreeMap实现的。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建 TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法。
2. TreeSet为基本操作（add、remove 和 contains）提供受保证的 log(n) 时间开销。
3. 另外，TreeSet是非同步的。 它的iterator 方法返回的迭代器是fail-fast的。
4. 对插入的元素进行排序，是一个有序的集合（主要与HashSet的区别）；
5. 底层使用红黑树结构，而不是哈希表结构；
6. 允许插入Null值；
7. 不允许插入重复元素；
8. 线程不安全；
9. 通过查看源码发现，在TreeSet调用add方法时，会调用到底层TreeMap的put方法，在put方法中会调用到compare(key, key)方法，进行key大小的比较；

   在比较的时候，会将传入的key进行类型强转，所以当我们自定义的App类进行比较的时候，自然就会抛出异常，因为App类并没有实现Comparable接口；

**HashSet**

1. HashSet实现Set接口，底层由HashMap(后面讲解)来实现，为哈希表结构，**新增元素相当于HashMap的key**，value默认为一个固定的Object。在我看来，HashSet相当于一个阉割版的HashMap;
2. 不允许出现重复因素；
3. 允许插入Null值；
4. 元素无序（添加顺序和遍历顺序不一致）；
5. 线程不安全，若2个线程同时操作HashSet，必须通过代码实现同步；

# Map




## Map

1. Map 中元素的顺序取决于迭代器迭代时的顺序，有的实现类保证了元素输入输出时的顺序，比如说 TreeMap；有的实现类则是无序的，比如 HashMap。
2. KeySet 是一个 Map 中键（key）的集合，**以 Set 的形式保存**，不允许重复，因此键存储的对象需要重写 equals() 和 hashCode() 方法。
3. Values 是一个 Map 中值 (value) 的集合，**以 Collection 的形式保存**，因此可以重复。
4. 通过 Map.entrySet() 方法获得的是一组 Entry 的集合，保存在 Set 中，所以 Map 中的 Entry 也不能重复。
5. 每个 key 只能对应一个 value, 多个 key 可以对应一个 value。
6. key, value 都可以是任何引用类型的数据，包括 null。

## AbstractMap

1. 当我们要实现一个 不可变的 Map 时，只需要继承这个类，然后实现 entrySet() 方法，这个方法返回一个保存所有 key-value 映射的 set。 通常这个 Set 不支持 add(), remove() 方法，Set 对应的迭代器也不支持 remove() 方法。

   如果想要实现一个 可变的 Map,我们需要在上述操作外，重写 put() 方法，因为 默认不支持 put 操作（unsupportedOperationException）。而且 entrySet() 返回的 Set 的迭代器，也得实现 remove() 方法，因为 AbstractMap 中的 删除相关操作都需要调用该迭代器的 remove() 方法。

   entrySet().iterator()这是获取iterator的方法。

2. 正如其他集合推荐的那样，比如 AbstractCollection 接口 ，实现类最好提供两种构造方法：
   - 一种是不含参数的，返回一个空 map
   - 一种是以一个 map 为参数，返回一个和参数内容一样的 map。
3. 有两个成员变量：

```
keySet, 保存 map 中所有键的 Set
values, 保存 map 中所有值的集合
```

他们都是 transient, volatile, 分别表示不可序列化、并发环境下变量的修改能够保证线程可见性。

需要注意的是 volatile 只能保证可见性，不能保证原子性，需要保证操作是原子性操作，才能保证使用 volatile 关键字的程序在并发时能够正确执行。

## HashMap

https://blog.csdn.net/samniwu/article/details/90550196（推荐这个）

https://yikun.github.io/2015/04/01/Java-HashMap%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86%E5%8F%8A%E5%AE%9E%E7%8E%B0/（这个也很好）

面试题

1. HashMap 的特殊存储结构使得在获取指定元素前需要经过哈希运算，得到目标元素在哈希表中的位置，然后再进行少量比较即可得到元素，这使得 HashMap 的查找效率贼高。
2. 当发生 哈希冲突（碰撞）的时候，HashMap 采用 **拉链法** 进行解决（不熟悉 “哈希冲突” 和 “拉链法” 这 2 个概念的同学可以 点这里了解），因此 HashMap 的底层实现是 **数组+链表**
3. 底层实现是 链表数组，JDK 8 后又加了 红黑树
4. 允许空键和空值（但空键只有一个，且放在第一位，下面会介绍）
5. 元素是无序的，而且顺序会不定时改变
6. 插入、获取的时间复杂度基本是 O(1)（前提是有适当的哈希函数，让元素分布在均匀的位置）
7. 插入新节点在1.6是头部插，1.8是尾部插。

   为什么改成尾部插入：解决死循环是一个点还有一个点就是 当时1.7时候用头插是考虑到了一个所谓的热点数据的点(新插入的数据可能会更早用到)，但这其实是个伪命题,y因为JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置(就是因为头插) 所以最后的结果 还是打乱了插入的顺序 所以总的来看支撑1.7使用头插的这点原因也不足以支撑下去了 所以就干脆换成尾插 一举多得。
8. 遍历整个 Map 需要的时间与 桶(数组) 的长度成正比（因此初始化时 HashMap 的容量不宜太大）
9. 两个关键因子：初始容量、加载因子

   默认初始容量：16，必须是 2 的整数次方。hashmap会选择最接近指定参数 cap 的 2 的 N 次方容量。假如你传入的是 5，返回的初始容量为 8 。

   默认加载因子的大小：0.75，可不是随便的，结合时间和空间效率考虑得到的。

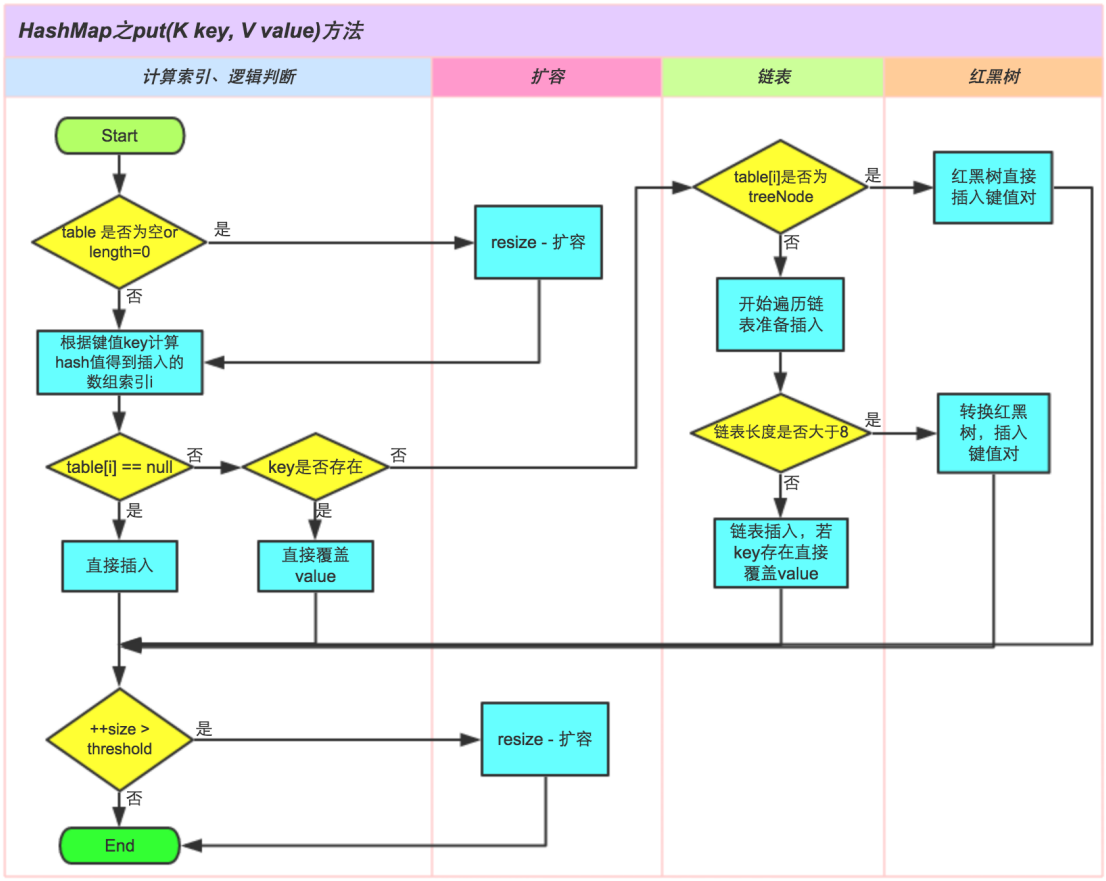
   加载因子过高，例如为1，虽然减少了空间开销，提高了空间利用率，但同时也增加了查询时间成本；

   加载因子过低，例如0.5，虽然可以减少查询时间成本，但是空间利用率很低，同时提高了rehash操作的次数。
10. 为什么容量必须是2的幂：https://blog.csdn.net/u010841296/article/details/82832166
11. 除了不允许 null 并且同步，Hashtable 几乎和他一样。
12. 一个桶里的entry，他们的hash一定都是一样的吗。应该是不一定的，比如 3&8 和 2&8 就会在一个桶里。
13. HashMap线程不安全的典型表现 —— **重hash的死循环。**
14. **红黑树**

    https://www.cnblogs.com/finite/p/8251587.html

    1. 规则
       - 每个节点要么是红色，要么是黑色；
       - 根节点永远是黑色的；

- 所有的叶节点都是是黑色的（注意这里说叶子节点其实是null 节点）；指定红黑树的每个叶子节点都是空节点，而且叶子节点都是黑色。但 Java 实现的红黑树将使用 null 来代表空节点，因此遍历红黑树时将看不到黑色的叶子节点，反而看到每个叶子节点都是红色的。
- 每个红色节点的两个子节点一定都是黑色；
- 从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点；




## HashTable

1. HashTable的操作几乎和HashMap一致，主要的区别在于HashTable为了实现多线程安全，在几乎所有的方法上都加上了synchronized锁，而加锁的结果就是HashTable操作的效率十分低下。
2. 与hashmap对比：

   （1）线程安全：HashMap是线程不安全的类，多线程下会造成并发冲突，但单线程下运行效率较高；HashTable是线程安全的类，很多方法都是用synchronized修饰，但同时因为加锁导致并发效率低下，单线程环境效率也十分低；

   （2）插入null：HashMap允许有一个键为null，允许多个值为null；但HashTable不允许键或值为null；

   （3）容量：HashMap底层数组长度必须为2的幂，这样做是为了hash准备，默认为16；而HashTable底层数组长度可以为任意值，这就造成了hash算法散射不均匀，容易造成hash冲突，默认为11；

   （4）Hash映射：HashMap的hash算法通过非常规设计，将底层table长度设计为2的幂，使用位与运算代替取模运算，减少运算消耗；而HashTable的hash算法首先使得hash值小于整型数最大值，再通过取模进行散射运算；

   （5）扩容机制：HashMap创建一个为原先2倍的数组，然后对原数组进行遍历以及rehash；HashTable扩容将创建一个原长度2倍的数组，再使用头插法将链表进行反序；

   （6）结构区别：HashMap是由数组+链表形成，在JDK1.8之后链表长度大于8时转化为红黑树；而HashTable一直都是数组+链表；

（7）继承关系：HashTable继承自Dictionary类；而HashMap继承自AbstractMap类；

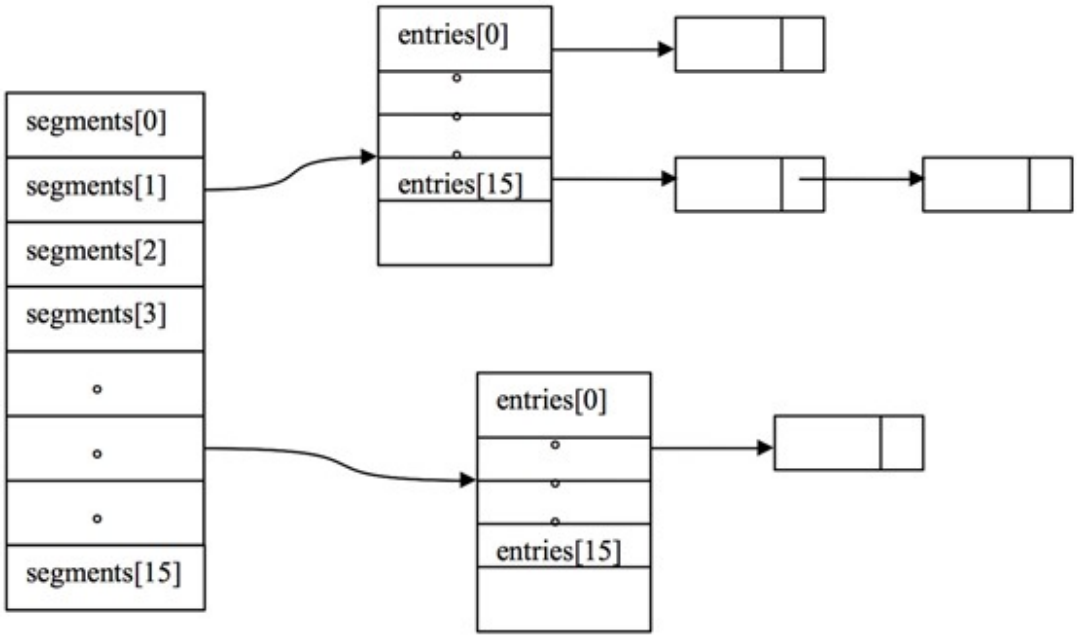
（8）迭代器：HashMap是fail-fast（查看之前HashMap相关文章）；而HashTable不是。

## ConcurrentHashMap

https://blog.csdn.net/justloveyou_/article/details/72783008?spm=1001.2014.3001.5502

1. 在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设为 16），及任意数量线程的读操作。
2. ConcurrentHashMap本质上是一个Segment数组，而一个Segment实例又包含若干个桶，每个桶中都包含一条由若干个 HashEntry 对象链接起来的链表。总的来说，ConcurrentHashMap的高效并发机制是通过以下三方面来保证的(具体细节见后文阐述)：

```
1. 通过锁分段技术保证并发环境下的写操作；

2. 通过 HashEntry的不变性、Volatile变量的内存可见性和加锁重读机制保证高效、安全的读操作；

3. 通过不加锁和加锁两种方案控制跨段操作的的安全性。
```




3. **Segment**
    1. Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。
    2. **Lock**
        1. https://www.cnblogs.com/dolphin0520/p/3923167.html
        2. Lock和synchronized的选择

            1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

            2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

            3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

5）Lock可以提高多个线程进行读操作的效率。

3. 如果锁具备可重入性，则称作为可重入锁。像synchronized和ReentrantLock都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

3. 在Segment类中，count 变量是一个计数器，它表示每个 Segment 对象管理的 table 数组包含的 HashEntry 对象的个数，也就是 Segment 中包含的 HashEntry 对象的总数。特别需要注意的是，之所以在每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是对 ConcurrentHashMap 并发性的考虑：**因为这样当需要更新计数器时，不用锁定整个ConcurrentHashMap。**事实上，每次对段进行结构上的改变，如在段中进行增加/删除节点(修改节点的值不算结构上的改变)，都要更新count的值，此外，在JDK的实现中每次读取操作开始都要先读取count的值。特别需要注意的是，count是volatile的，这使得对count的任何更新对其它线程都是立即可见的。modCount用于统计段结构改变的次数，主要是为了检测对多个段进行遍历过程中某个段是否发生改变，这一点具体在谈到跨段操作时会详述。threashold用来表示段需要进行重哈希的阈值。loadFactor表示段的负载因子，其值等同于ConcurrentHashMap的负载因子的值。table是一个典型的链表数组，而且也是volatile的，这使得对table的任何更新对其它线程也都是立即可见的。

4. **HashEntry**

1. 在HashEntry类中，key，hash和next域都被声明为final的，value域被volatile所修饰，因此HashEntry对象几乎是不可变的，这是ConcurrentHashmap读操作并不需要加锁的一个重要原因。

   next域被声明为final本身就意味着我们不能从hash链的中间或尾部添加或删除节点，因为这需要修改next引用值，因此所有的节点的修改只能从头部开始。对于put操作，可以一律添加到Hash链的头部。但是对于remove操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制(重新new)一遍，最后一个节点指向要删除结点的下一个结点(这在谈到ConcurrentHashMap的删除操作时还会详述)。特别地，由于value域被volatile修饰，所以其可以确保被读线程读到最新的值，这是ConcurrentHashmap读操作并不需要加锁的另一个重要原因。实际上，ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。

2. 三个非常重要的参数：初始容量、负载因子 和 并发级别，这三个参数是影响ConcurrentHashMap性能的重要参数。从上述源码我们可以看出，ConcurrentHashMap 也正是通过initialCapacity、loadFactor和concurrencyLevel这三个参数进行构造并初始化segments数组、段偏移量segmentShift、段掩码segmentMask和每个segment的。

3. 假设ConcurrentHashMap一共分为2^n个段，每个段中有2^m个桶，那么段的定位方式是将key的hash值的高n位与(2^n-1)相与。在定位到某个段后，再将key的hash值的低m位与(2^m-1)相与，定位到具体的桶位。

4. **volatile**

   https://www.cnblogs.com/dolphin0520/p/3920373.html

   https://blog.csdn.net/justloveyou_/article/details/53672005

5. ConcurrentHashMap的重哈希实际上是对ConcurrentHashMap的某个段的重哈希，因此ConcurrentHashMap的每个段所包含的桶位自然也就不尽相同。

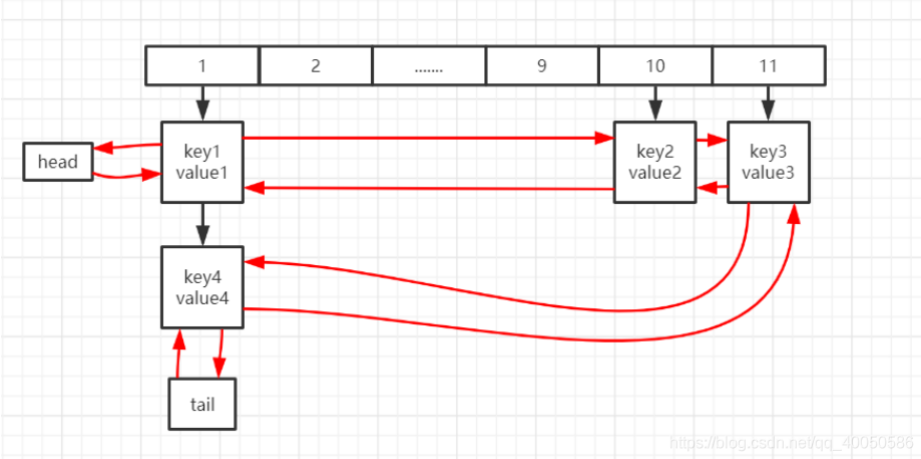
6. ConcurrentHashMap不同于HashMap，它既不允许key值为null，也不允许value值为null。但是，get时会存在键值对存在且的Value值为null的情形。JDK官方给出的解释是，这种情形发生的场景是：初始化HashEntry时发生的指令重排序导致的，也就是在HashEntry初始化完成之前便返回了它的引用。这时，JDK给出的解决之道就是加锁重读。（我觉得是线程A在get的时候B在写，B只写了一半，此时已经分配了内存但还没完全赋值之类的）

7. size方法主要思路是先在没有锁的情况下对所有段大小求和，这种求和策略最多执行RETRIES_BEFORE_LOCK次(默认是两次)：在没有达到RETRIES_BEFORE_LOCK之前，求和操作会不断尝试执行（这是因为遍历过程中可能有其它线程正在对已经遍历过的段进行结构性更新）；在超过RETRIES_BEFORE_LOCK之后，如果还不成功就在持有所有段锁的情况下再对所有段大小求和。事实上，在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试RETRIES_BEFORE_LOCK次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

   那么，ConcurrentHashMap是如何判断在统计的时候容器的段发生了结构性更新了呢？我们在前文中已经知道，Segment包含一个modCount成员变量，在会引起段发生结构性改变的所有操作(put操作、 remove操作和clean操作)里，都会将变量modCount进行加1，因此，**JDK只需要在统计size前后比较modCount是否发生变化就可以得知容器的大小是否发生变化。**

## LinkedHashMap

Map 综述（二）：彻头彻尾理解 LinkedHashMap




1. 它是一个将所有Entry节点链入一个双向链表的HashMap。由于LinkedHashMap是HashMap的子类，所以LinkedHashMap自然会拥有HashMap的所有特性。比如，LinkedHashMap的元素存取过程基本与HashMap基本类似，只是在细节实现上稍有不同。当然，这是由LinkedHashMap本身的特性所决定的，因为它额外维护了一个双向链表用于保持迭代顺序。此外，LinkedHashMap可以很好的支持LRU算法。
2. 虽然LinkedHashMap增加了时间和空间上的开销，但是它通过维护一个额外的双向链表保证了迭代顺序。特别地，该迭代顺序可以是插入顺序，也可以是访问顺序。因此，根据链表中元素的顺序可以将LinkedHashMap分为：保持插入顺序的LinkedHashMap 和 保持访问顺序的LinkedHashMap，其中LinkedHashMap的默认实现是按插入顺序排序的。
3. 非同步
4. 与HashMap相比，LinkedHashMap增加了两个属性用于保证迭代顺序，分别是 **双向链表头结点header** 和 **标志位accessOrder** (值为true时，表示按照访问顺序迭代；值为false时，表示按照插入顺序迭代)。
    1. 这个header是用transient关键字修饰的。transient是短暂的意思。对于transient 修饰的成员变量，在类的实例对象的序列化处理过程中会被忽略。 因此，transient变量不会贯穿对象的序列化和反序列化，生命周期仅存于调用者的内存中而不会写到磁盘里进行持久化。好处是：在持久化对象时，对于一些特殊的数据成员（如用户的密码，银行卡号等），我们不想用序列化机制来保存它。为了在一个特定对象的一个成员变量上关闭序列化，可以在这个成员变量前加上关键字transient。不过，一个静态变量不管是否被transient修饰，均不能被序列化(如果反序列化后类中static变量还有值，则值为当前VM中对应static变量的值)。序列化保存的是对象状态，静态变量保存的是类状态，因此序列化并不保存静态变量。

2. 可以理解为，next跟hashmap本身作用一样，但是双向链表是要去记录访问、插入顺序的，所以单看物理上的排列，是用next链接起来的，但是还有一个看不见的链，就是双向链表的顺序。

5. 重新定义了Entry。LinkedHashMap中的Entry增加了两个指针 **before** 和 **after**，它们分别用于维护双向链接列表。特别需要注意的是，next用于维护HashMap各个桶中Entry的连接顺序，before、after用于维护Entry插入的先后顺序的。
6. 无论采用何种方式创建LinkedHashMap，其都会调用HashMap相应的构造函数。事实上，不管调用HashMap的哪个构造函数，HashMap的构造函数都会在最后调用一个init()方法进行初始化，只不过这个方法在HashMap中是一个空实现，而在LinkedHashMap中重写了它，用于初始化它所维护的双向链表。
7. 对于put(Key,Value)方法而言，LinkedHashMap完全继承了HashMap的 put(Key,Value) 方法，只是对put(Key,Value)方法所调用的recordAccess方法和addEntry方法进行了重写；对于get(Key)方法而言，LinkedHashMap则直接对它进行了重写。

   在put操作上，虽然LinkedHashMap完全继承了HashMap的put操作，但是在细节上还是做了一定的调整，比如，在LinkedHashMap中向哈希表中插入新Entry的同时，还会通过Entry的addBefore方法将其链入到双向链表中。在扩容操作上，虽然LinkedHashMap完全继承了HashMap的resize操作，但是鉴于性能和LinkedHashMap自身特点的考量，LinkedHashMap对其中的重哈希过程(transfer方法)进行了重写。在读取操作上，LinkedHashMap中重写了HashMap中的get方法，通过HashMap中的getEntry方法获取Entry对象。在此基础上，进一步获取指定键对应的值。
8. 相比HashMap而言，LinkedHashMap在向哈希表添加一个键值对的同时，也会将其链入到它所维护的双向链表中，以便设定迭代顺序。
9. LinkedHashMap完全继承了HashMap的resize()方法，只是对它所调用的transfer方法进行了重写。
10. LinkedHashMap重写了HashMap中的recordAccess方法（HashMap中该方法为空），当调用父类的put方法时，在发现key已经存在时，会调用该方法；当调用自己的get方法时，也会调用到该方法。该方法提供了LRU算法的实现，它将最近使用的Entry放到双向循环链表的尾部。也就是说，当accessOrder为true时，get方法和put方法都会调用recordAccess方法使得最近使用的Entry移到双向链表的末尾；当accessOrder为默认值false时，从源码中可以看出recordAccess方法什么也不会做。**从插入顺序的层面来说，新的Entry插入到双向链表的尾部可以实现按照插入的先后顺序来迭代Entry，而从访问顺序的层面来说，新put进来的Entry又是最近访问的Entry，也应该将其放在双向链表的尾部。**

    当我们要用LinkedHashMap实现LRU算法时，就需要调用构造方法并将accessOrder置为true。

## TreeMap

1. 与HashMap相比，TreeMap是一个能比较元素大小的Map集合，会对传入的key进行了大小排序。其中，可以使用元素的自然顺序，也可以使用集合中自定义的比较器来进行排序；
2. 不允许出现重复的key；可以插入null键，null值；可以对元素进行排序；

# Comparable与Comparator

对比区别

1. 他们都是java的一个接口, 并且是用来对自定义的class比较大小的。
2. Comparable是排序接口；若一个类实现了Comparable接口，就意味着“该类支持排序”。

   而Comparator是比较器；我们若需要控制某个类的次序，可以建立一个“该类的比较器”来进行排序。

   我们不难发现：Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。

## Comparable

1. 基本数据类型包装类和String类均已实现了Comparable接口。
2. Comparable 是排序接口。

   若一个类实现了Comparable接口，就意味着“该类支持排序”。 即然实现Comparable接口的类支持排序，假设现在存在“实现Comparable接口的类的对象的List列表(或数组)”，则该List列表(或数组)可以通过 Collections.sort（或 Arrays.sort）进行排序。

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

3. 假设我们通过 x.compareTo(y) 来“比较x和y的大小”。若返回“负数”，意味着“x比y小”；返回“零”，意味着“x等于y”；返回“正数”，意味着“x大于y”。如果想改排序可以判断compareTo输出>0就返回-1。

## Comparator

1. Comparator 是比较器接口。

   我们若需要控制某个类的次序，而该类本身不支持排序(即没有实现Comparable接口)；那么，我们可以建立一个“该类的比较器”来进行排序。这个“比较器”只需要实现Comparator接口即可。

   也就是说，我们可以通过“实现Comparator类来新建一个比较器”，然后通过该比较器对类进行排序。

```
 public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

2. 若一个类要实现Comparator接口：它一定要实现compareTo(T o1, T o2) 函数，但可以不实现 equals(Object obj) 函数。

   为什么可以不实现 equals(Object obj) 函数呢？ 因为任何类，默认都是已经实现了equals(Object obj)的。 Java中的一切类都是继承于java.lang.Object，在Object.java中实现了equals(Object obj)函数；所以，其它所有的类也相当于都实现了该函数。

   int compare(T o1, T o2) 是“比较o1和o2的大小”。返回“负数”，意味着“o1比o2小”；返回“零”，意味着“o1等于o2”；返回“正数”，意味着“o1大于o2”。

# 1.9 类

# 类

1. 在Java中，类文件是以.java为后缀的代码文件，在每个类文件中最多只允许出现一个public类，当有public类的时候，类文件的名称必须和public类的名称相同，若不存在public，则类文件的名称可以为任意的名称（当然以数字开头的名称是不允许的）。
2. 在类内部，对于**成员变量**，如果在定义的时候没有进行显示的赋值初始化，则Java会保证类的每个成员变量都得到恰当的初始化（局部变量不会初始化）：
    1. 对于 char、short、byte、int、long、float、double等基本数据类型的变量来说会默认初始化为0（boolean变量默认会被初始化为false）；

2. 对于引用类型的变量，会默认初始化为null。
3. 如果没有显示地定义构造器，则编译器会自动创建一个无参构造器，但是要记住一点，如果显示地定义了构造器，编译器就不会自动添加构造器。

# 类的加载

https://blog.csdn.net/justloveyou_/article/details/72466105

1. 我们知道，一个.java文件在编译后会形成相应的一个或多个Class文件，这些Class文件中描述了类的各种信息，并且它们最终都需要被加载到虚拟机中才能被运行和使用。事实上，虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型的过程就是虚拟机的类加载机制。
2. 在Java语言里面，类型的加载和连接都是在程序运行期间完成，这样会在类加载时稍微增加一些性能开销，但是却能为Java应用程序提供高度的灵活性，Java中天生可以动态扩展的语言特性多态就是依赖运行期动态加载和动态链接这个特点实现的。例如，如果编写一个使用接口的应用程序，可以等到运行时再指定其实际的实现。这种组装应用程序的方式广泛应用于Java程序之中。

   既然这样，那么，

   - 虚拟机什么时候才会加载Class文件并初始化类呢？（类加载和初始化时机）
   - 虚拟机如何加载一个Class文件呢？（Java类加载的方式：类加载器、双亲委派机制）
   - 虚拟机加载一个Class文件要经历那些具体的步骤呢？（类加载过程/步骤）

## 类加载的时机

Java类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using) 和卸载(Unloading)七个阶段。其中准备、验证、解析3个部分统称为连接（Linking），如图所示：


加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。以下陈述的内容都已HotSpot为基准。特别需要注意的是，类的加载过程必须按照这种顺序按部就班地“开始”，而不是按部就班的“进行”或“完成”，因为这些阶段通常都是相互交叉地混合式进行的，也就是说通常会在一个阶段执行的过程中调用或激活另外一个阶段。

1. 类加载时机：什么情况下虚拟机需要开始加载一个类呢？虚拟机规范中并没有对此进行强制约束，这点可以交给虚拟机的具体实现来自由把握。
2. 类初始化时机

   那么，什么情况下虚拟机需要开始初始化一个类呢？这在虚拟机规范中是有严格规定的，虚拟机规范指明 有且只有 六种情况必须立即对类进行初始化（而这一过程自然发生在加载、验证、准备之后）：

1. 遇到new、getstatic、putstatic或invokestatic这四条字节码指令（注意，newarray指令触发的只是数组类型本身的初始化，而不会导致其相关类型的初始化，比如，new String[]只会直接触发String[]类的初始化，也就是触发对类[Ljava.lang.String的初始化，而直接不会触发String类的初始化）时，如果类没有进行过初始化，则需要先对其进行初始化。生成这四条指令的最常见的Java代码场景是：
    - 使用new关键字实例化对象的时候；
    - 读取或设置一个类的静态字段（被final修饰，已在编译期把结果放入常量池的静态字段除外）的时候；
    - 调用一个类的静态方法的时候。
2. 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
5. 当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getstatic,REF_putstatic,REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。
6. 当一个接口中定义了JDK8新加入的默认方法（被default关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那么该接口要在这个类之前初始化。

这六种场景中的行为称为对一个类进行 **主动引用。**除此之外，所有引用类的方式，都不会触发初始化，称为 **被动引用。**

特别需要指出的是，**类的实例化**与类的**初始化**是两个完全不同的概念：

- 类的实例化是指创建一个类的实例(对象)的过程；
- 类的初始化是指为类中各个类成员(被static修饰的成员变量)赋初始值的过程，是类生命周期中的一个阶段。

3. 被动引用的几种经典场景（有代码）
    1. 通过子类引用父类的静态字段，不会导致子类初始化

```
public class SSClass{
    static{
        System.out.println("SSClass");
    }
}

public class SClass extends SSClass{
    static{
        System.out.println("SClass init!");
    }

    public static int value = 123;

    public SClass(){
        System.out.println("init SClass");
    }
}

public class SubClass extends SClass{
    static{
        System.out.println("SubClass init");
    }
```

```
    static int a;

    public SubClass(){
        System.out.println("init SubClass");
    }
}

public class NotInitialization{
    public static void main(String[] args){
        System.out.println(SubClass.value);
    }
}/* Output:
        SSClass
        SClass init!
        123
 *///:~
```

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。在本例中，由于value字段是在类SClass中定义的，因此该类会被初始化；此外，在初始化类SClass时，虚拟机会发现其父类SSClass还未被初始化，因此虚拟机将先初始化父类SSClass，然后初始化子类SClass，而SubClass始终不会被初始化。

2. 通过数组定义来引用类，不会触发此类的初始化

```
public class NotInitialization{
    public static void main(String[] args){
        SClass[] sca = new SClass[10];
    }
}
```

上述案例运行之后并没有任何输出，说明虚拟机并没有初始化类SClass。但是，这段代码触发了另外一个名为[Lcn.edu.tju.rico.SClass的类的初始化。从类名称我们可以看出，这个类代表了元素类型为SClass的一维数组，它是由虚拟机自动生成的，直接继承于Object的子类，创建动作由字节码指令newarray触发。

3. 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化

```
public class ConstClass{

    static{
        System.out.println("ConstClass init!");
    }

    public static  final String CONSTANT = "hello world";
}

public class NotInitialization{
    public static void main(String[] args){
        System.out.println(ConstClass.CONSTANT);
    }
}/* Output:
        hello world
```

```
*///:~
```

上述代码运行之后，只输出 “hello world”，这是因为虽然在Java源码中引用了ConstClass类中的常量CONSTANT，但是编译阶段将此常量的值“hello world”存储到了NotInitialization常量池中，对常量ConstClass.CONSTANT的引用实际都被转化为NotInitialization类对自身常量池的引用了。也就是说，实际上NotInitialization的Class文件之中并没有ConstClass类的符号引用入口，这两个类在编译为Class文件之后就不存在关系了。**（使用一个别的类的类变量，这个类变量会存在我的常量池中吗）**

## 类加载的过程

1. 加载（Loading）

   在加载阶段（可以参考java.lang.ClassLoader的loadClass()方法），虚拟机需要完成以下三件事情：

   - 通过一个类的全限定名来获取定义此类的二进制字节流（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等）；
   - 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构；
   - 在内存中(对于HotSpot虚拟就而言就是方法区)生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口；

   加载阶段和连接阶段（Linking）的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

   特别地，第一件事情(通过一个类的全限定名来获取定义此类的二进制字节流)是由类加载器完成的，具体涉及JVM预定义的类加载器、双亲委派模型等内容，详情请参见我的转载博文《深入理解Java类加载器(一)：Java类加载原理解析》中的说明，此不赘述。

2. 验证（Verification）

   验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。 验证阶段大致会完成4个阶段的检验动作：

   - 文件格式验证：验证字节流是否符合Class文件格式的规范
   - 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求(例如：这个类是否有父类，除了java.lang.Object之外)；
   - 字节码验证：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的;
   - 符号引用验证：确保解析动作能正确执行。

   验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响。如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

3. 准备(Preparation)

   准备阶段是正式为类变量(static 成员变量)分配内存并设置类变量初始值（零值）的阶段，这些变量所使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

   ```
   public static int value = 123;
   ```

   那么，变量value在准备阶段过后的值为0而不是123。因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器方法()之中，所以把value赋值为123的动作将在初始化阶段才会执行。至于“特殊情况”是指：当类字段的字段属性是ConstantValue时，会在准备阶段初始化为指定的值，所以标注为**final**之后，value的值在准备阶段初始化为123而非0。

```
public static final int value = 123;
```

4. 解析(Resolution)

   解析阶段是虚拟机将**常量池内**的**符号引用**替换为**直接引用**的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

   1. 符号引用（Symbolic References）：

      符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可。例如，在Class文件中它以CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等类型的常量出现。符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到内存中。在Java中，一个java类将会编译成一个class文件。在编译时，java类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。比如org.simple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号org.simple.Language（假设是这个，当然实际中是由类似于CONSTANT_Class_info的常量来表示的）来表示Language类的地址。各种虚拟机实现的内存布局可能有所不同，但是它们能接受的符号引用都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

   2. 直接引用

      直接引用可以是

      （1）直接指向目标的指针（比如，指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的指针）

      （2）相对偏移量（比如，指向实例变量、实例方法的直接引用都是偏移量）

      （3）一个能间接定位到目标的句柄

      直接引用是和虚拟机的布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经被加载入内存中了。

5. 初始化(Initialization)

   类初始化阶段是类加载过程的最后一步。在前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。**到了初始化阶段，才真正开始执行类中定义的java程序代码(字节码)。**

   在准备阶段，变量已经赋过一次系统要求的初始值(零值)；而在初始化阶段，则根据程序猿通过程序制定的主观计划去初始化类变量和其他资源，或者更直接地说：初始化阶段是执行类构造器()方法的过程。()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块static{}中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，**静态语句块只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。如下**：

```
public class Test{
    static{
        i=0;
        System.out.println(i);//Error：Cannot reference a field before it is defined（非法向前应用）
    }
    static int i=1;
}
```

那么注释报错的那行代码，改成下面情形，程序就可以编译通过并可以正常运行了。

```
public class Test{
    static{
```

```
        i=0;
        //System.out.println(i);
    }

    static int i=1;

    public static void main(String args[]){
        System.out.println(i);
    }
}/* Output:
        1
 *///:~
```

类构造器()与实例构造器()不同，它不需要程序员进行显式调用，虚拟机会保证在子类类构造器()执行之前，父类的类构造()执行完毕。由于父类的构造器()先执行，也就意味着父类中定义的静态语句块/静态变量的初始化要优先于子类的静态语句块/静态变量的初始化执行。特别地，类构造器()对于类或者接口来说并不是必需的，如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生产类构造器()。

clinit和init的执行顺序解读

虚拟机会保证一个类的类构造器()在多线程环境中被正确的加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的类构造器()，其他线程都需要阻塞等待，直到活动线程执行()方法完毕。特别需要注意的是，在这种情形下，其他线程虽然会被阻塞，但如果执行()方法的那条线程退出后，其他线程在唤醒之后不会再次进入/执行()方法，因为 在同一个类加载器下，一个类型只会被初始化一次。如果在一个类的()方法中有耗时很长的操作，就可能造成多个线程阻塞，在实际应用中这种阻塞往往是隐藏的，如下所示：

```
public class DealLoopTest {
    static{
        System.out.println("DealLoopTest...");
    }
    static class DeadLoopClass {
        static {
            if (true) {
                System.out.println(Thread.currentThread()
                        + "init DeadLoopClass");
                while (true) {      // 模拟耗时很长的操作
                }
            }
        }
    }

    public static void main(String[] args) {
        Runnable script = new Runnable() {   // 匿名内部类
            public void run() {
                System.out.println(Thread.currentThread() + " start");
                DeadLoopClass dlc = new DeadLoopClass();
                System.out.println(Thread.currentThread() + " run over");
            }
        };

        Thread thread1 = new Thread(script);
        Thread thread2 = new Thread(script);
        thread1.start();
        thread2.start();
    }
```

```
}/* Output:
        DealLoopTest...
        Thread[Thread-1,5,main] start
        Thread[Thread-0,5,main] start //这里先执行这个是因为多线程吧
        Thread[Thread-1,5,main]init DeadLoopClass
 *///:~
```

如上述代码所示，在初始化DeadLoopClass类时，线程Thread-1得到执行并在执行这个类的类构造器() 时，由于该方法包含一个死循环，因此久久不能退出。

## 典型案例分析

我们知道，在Java中， 创建一个对象常常需要经历如下几个过程：父类的类构造器() -> 子类的类构造器() -> 父类的成员变量和实例代码块 -> 父类的构造函数 -> 子类的成员变量和实例代码块 -> 子类的构造函数。至于为什么是这样的一个过程，笔者在本文的姊妹篇《 深入理解Java对象的创建过程：类的初始化与实例化》很好的解释了这个问题。

那么，我们看看下面的程序的输出结果：

```
public class StaticTest {
    public static void main(String[] args) {
        staticFunction();
    }

    static StaticTest st = new StaticTest();

    static {   //静态代码块
        System.out.println("1");
    }

    {       // 实例代码块
        System.out.println("2");
    }

    StaticTest() {    // 实例构造器
        System.out.println("3");
        System.out.println("a=" + a + ",b=" + b);
    }

    public static void staticFunction() {   // 静态方法
        System.out.println("4");
    }

    int a = 110;    // 实例变量
    static int b = 112;     // 静态变量
}/* Output:
        2
        3
        a=110,b=0
        1
        4
 *///:~
```

大家能得到正确答案吗？虽然笔者勉强猜出了正确答案，但总感觉怪怪的。因为在初始化阶段，当JVM对类StaticTest进行初始化时，首先会执行下面的语句：

```
static StaticTest st = new StaticTest();
```

也就是实例化StaticTest对象，但这个时候类都没有初始化完毕啊，能直接进行实例化吗？事实上，这涉及到一个根本问题就是：**实例初始化不一定要在类初始化结束之后才开始初始化。** 下面我们结合类的加载过程说明这个问题。

我们知道，类的生命周期是：加载->验证->准备->解析->初始化->使用->卸载，并且只有在准备阶段和初始化阶段才会涉及类变量的初始化和赋值，因此我们只针对这两个阶段进行分析：

首先，在类的准备阶段需要做的是为类变量（static变量）分配内存并设置默认值(零值)，因此在该阶段结束后，类变量st将变为null、b变为0。特别需要注意的是，如果类变量是final的，那么编译器在编译时就会为value生成ConstantValue属性，并在准备阶段虚拟机就会根据ConstantValue的设置将变量设置为指定的值。也就是说，如果上述程度对变量b采用如下定义方式时：

```
static final int b=112
```

那么，在准备阶段b的值就是112，而不再是0了。

此外，在类的初始化阶段需要做的是执行类构造器()，需要指出的是，类构造器本质上是编译器收集所有静态语句块和类变量的赋值语句按语句在源码中的顺序合并生成类构造器()。因此，对上述程序而言，JVM将先执行第一条静态变量的赋值语句：

```
st = new StaticTest ()
```

此时，就碰到了笔者上面的疑惑，即“在类都没有初始化完毕之前，能直接进行实例化相应的对象吗？”。事实上，从Java角度看，我们知道一个类初始化的基本常识，那就是：在同一个类加载器下，一个类型只会被初始化一次。所以，一旦开始初始化一个类型，无论是否完成，后续都不会再重新触发该类型的初始化阶段了(只考虑在同一个类加载器下的情形)。因此，在实例化上述程序中的st变量时，实际上是把实例初始化嵌入到了静态初始化流程中，并且在上面的程序中，嵌入到了静态初始化的起始位置。这就导致了实例初始化完全发生在静态初始化之前，当然，这也是导致a为110，b为0的原因。（因为先执行力了实例初始化，所以a=110？）

**实例初始化流程**：实例初始化就是执行()方法

(1)()方法可能重载有多个，有几个构造器就有几个()方法

(2)()方法由非静态变量显示赋值代码和非静态代码块、对应构造器代码组成

(3)非静态变量显示赋值代码和非静态代码块从上到下顺序执行，**最后执行构造器代码**

(4)每次创建实例对象，调用对应构造器，执行的都是()方法

(5)()方法的首行是super()或super(实参列表)，即对应父类的()方法

因此，上述程序会有上面的输出结果。下面，我们对上述程序稍作改动，如下所示：

```
public class StaticTest {
    public static void main(String[] args) {
        staticFunction();
    }

    static StaticTest st = new StaticTest();// 在哪，堆还是方法区？

    static {
        System.out.println("1");
    }
```

```
    {
        System.out.println("2");
    }

    StaticTest() {
        System.out.println("3");
        System.out.println("a=" + a + ",b=" + b);
    }

    public static void staticFunction() {
        System.out.println("4");
    }

    int a = 110;// 这个a储存在哪？
    static int b = 112;
    static StaticTest st1 = new StaticTest();
}
```

那么，此时程序的输出又是什么呢？如果你对上述的内容理解很好的话，不难得出结论(只有执行完上述代码行后，StaticTest类才被初始化完成)，即：

```
2
3
a=110,b=0
1
2
3
a=110,b=112
4
```

另外，下面这道经典题目也很有意思，如下：

```
class Foo {
    int i = 1;

    Foo() {
        System.out.println(i);
        int x = getValue();
        System.out.println(x);
    }

    {
        i = 2;
    }

    protected int getValue() {
        return i;
    }
}

//子类
class Bar extends Foo {
    int j = 1;

    Bar() {
        j = 2;
```

```
    }

    {
        j = 3;
    }

    @Override
    protected int getValue() {
        return j;
    }
}

public class ConstructorExample {
    public static void main(String... args) {
        Bar bar = new Bar();
        System.out.println(bar.getValue());
    }
}
```

在通过使用Bar类的构造方法new一个Bar类的实例时，首先会调用Foo类构造函数，因此(1)处输出是2，这从Foo类构造函数的等价变换中可以直接看出。(2)处输出是0，为什么呢？因为在执行Foo的构造函数的过程中，由于Bar重载了Foo中的getValue方法，所以根据Java的多态特性可以知道，**其调用的getValue方法是被Bar重写的那个getValue方法。**但由于这时Bar的构造函数还没有被执行，因此此时j的值还是默认值0，因此(2)处输出是0。最后，在执行(3)处的代码时，由于bar对象已经创建完成，所以此时再访问j的值时，就得到了其初始化后的值2，这一点可以从Bar类构造函数的等价变换中直接看出。

# 类的初始化与实例化

在Java中，一个对象在可以被使用之前必须要被正确地初始化，这一点是Java规范规定的。在实例化一个对象时，JVM首先会检查相关类型是否已经加载并初始化，如果没有，则JVM立即进行加载并调用类构造器完成类的初始化。在类初始化过程中或初始化完毕后，根据具体情况才会去对类进行实例化。

## Java对象创建时机

我们知道，一个对象在可以被使用之前必须要被正确地实例化。在Java代码中，有很多行为可以引起对象的创建，最为直观的一种就是使用new关键字来调用一个类的构造函数显式地创建对象，这种方式在Java规范中被称为：由执行类实例创建表达式而引起的对象创建。除此之外，我们还可以使用反射机制(Class类的newInstance方法、使用Constructor类的newInstance方法)、使用Clone方法、使用反序列化等方式创建对象。下面笔者分别对此进行一一介绍：

1. 使用new关键字创建对象

   这是我们最常见的也是最简单的创建对象的方式，通过这种方式我们可以调用任意的构造函数（无参的和有参的）去创建对象。

2. 使用Class类的newInstance方法(反射机制)

   我们也可以通过Java的反射机制使用Class类的newInstance方法来创建对象，事实上，这个newInstance方法调用无参的构造器创建对象，比如：

   ```
   Student student2 = (Student)Class.forName("Student类全限定名").newInstance();
   或者：
   Student stu = Student.class.newInstance();
   ```

3. 使用Constructor类的newInstance方法(反射机制)

java.lang.relect.Constructor类里也有一个newInstance方法可以创建对象，该方法和Class类中的newInstance方法很像，但是相比之下，Constructor类的newInstance方法更加强大些，我们可以通过这个newInstance方法调用有参数的和私有的构造函数，比如：

```
public class Student {

    private int id;

    public Student(Integer id) {
        this.id = id;
    }

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

        Constructor<Student> constructor = Student.class
                .getConstructor(Integer.class);
        Student stu3 = constructor.newInstance(123);
    }
}
```

使用newInstance方法的这两种方式创建对象使用的就是Java的反射机制，事实上Class的newInstance方法内部调用的也是Constructor的newInstance方法。

4. 使用Clone方法创建对象

   无论何时我们调用一个对象的clone方法，JVM都会帮我们创建一个新的、一样的对象，特别需要说明的是，用clone方法创建对象的过程中并不会调用任何构造函数。关于如何使用clone方法以及浅克隆/深克隆机制，笔者已经在博文《 Java String 综述(下篇)》做了详细的说明。简单而言，要想使用clone方法，我们就必须先实现Cloneable接口并实现其定义的clone方法，这也是原型模式的应用。比如：

```
public class Student implements Cloneable{

    private int id;

    public Student(Integer id) {
        this.id = id;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // TODO Auto-generated method stub
        return super.clone();
    }

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

        Constructor<Student> constructor = Student.class
                .getConstructor(Integer.class);
        Student stu3 = constructor.newInstance(123);
        Student stu4 = (Student) stu3.clone();
    }
}
```

5. 使用(反)序列化机制创建对象

当我们反序列化一个对象时，JVM会给我们创建一个单独的对象，在此过程中，JVM并不会调用任何构造函数。为了反序列化一个对象，我们需要让我们的类实现Serializable接口，比如：

```
public class Student implements Cloneable, Serializable {

    private int id;

    public Student(Integer id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return "Student [id=" + id + "]";
    }

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

        Constructor<Student> constructor = Student.class
                .getConstructor(Integer.class);
        Student stu3 = constructor.newInstance(123);

        // 写对象
        ObjectOutputStream output = new ObjectOutputStream(
                new FileOutputStream("student.bin"));
        output.writeObject(stu3);
        output.close();

        // 读对象
        ObjectInputStream input = new ObjectInputStream(new FileInputStream(
                "student.bin"));
        Student stu5 = (Student) input.readObject();
        System.out.println(stu5);
    }
}
```

6. 完整实例

```
public class Student implements Cloneable, Serializable {

    private int id;

    public Student() {

    }

    public Student(Integer id) {
        this.id = id;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // TODO Auto-generated method stub
        return super.clone();
    }
```

```
    @Override
    public String toString() {
        return "Student [id=" + id + "]";
    }

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

        System.out.println("使用new关键字创建对象：");
        Student stu1 = new Student(123);
        System.out.println(stu1);
        System.out.println("\n---------------------------\n");


        System.out.println("使用Class类的newInstance方法创建对象：");
        Student stu2 = Student.class.newInstance();     //对应类必须具有无参构造
方法，且只有这一种创建方式
        System.out.println(stu2);
        System.out.println("\n---------------------------\n");

        System.out.println("使用Constructor类的newInstance方法创建对象：");
        Constructor<Student> constructor = Student.class
                .getConstructor(Integer.class);   // 调用有参构造方法
        Student stu3 = constructor.newInstance(123);
        System.out.println(stu3);
        System.out.println("\n---------------------------\n");

        System.out.println("使用Clone方法创建对象：");
        Student stu4 = (Student) stu3.clone();
        System.out.println(stu4);
        System.out.println("\n---------------------------\n");

        System.out.println("使用(反)序列化机制创建对象：");
        // 写对象
        ObjectOutputStream output = new ObjectOutputStream(
                new FileOutputStream("student.bin"));
        output.writeObject(stu4);
        output.close();

        // 读取对象
        ObjectInputStream input = new ObjectInputStream(new FileInputStream(
                "student.bin"));
        Student stu5 = (Student) input.readObject();
        System.out.println(stu5);

    }
}/* Output:
        使用new关键字创建对象：
        Student [id=123]

        ---------------------------

        使用Class类的newInstance方法创建对象：
        Student [id=0]

        ---------------------------

        使用Constructor类的newInstance方法创建对象：
        Student [id=123]
```

```
        ---------------------------

        使用Clone方法创建对象:
        Student [id=123]

        ---------------------------

        使用(反)序列化机制创建对象:
        Student [id=123]
*///:~
```

从Java虚拟机层面看，除了使用new关键字创建对象的方式外，其他方式全部都是通过转变为invokevirtual指令直接创建对象的。

## Java 对象的创建过程

当一个对象被创建时，虚拟机就会为其分配内存来存放对象自己的实例变量及其从父类继承过来的实例变量(即使这些从超类继承过来的实例变量有可能被隐藏也会被分配空间)。在为这些实例变量分配内存的同时，这些实例变量也会被赋予默认值(零值)。在内存分配完成之后，Java虚拟机就会开始对新创建的对象按照程序猿的意志进行初始化。在Java对象初始化过程中，主要涉及三种执行对象初始化的结构，分别是 **实例变量初始化、实例代码块初始化 以及 构造函数初始化。**

1. 实例变量初始化与实例代码块初始化

   我们在定义（声明）实例变量的同时，还可以直接对实例变量进行赋值或者使用实例代码块对其进行赋值。如果我们以这两种方式为实例变量进行初始化，那么它们将在构造函数执行之前完成这些初始化操作。实际上，如果我们对实例变量直接赋值或者使用实例代码块赋值，那么编译器会将其中的代码**放到类的构造函数中去**，并且这些代码会被放在对超类构造函数的调用语句之后(还记得吗？**Java要求构造函数的第一条语句必须是超类构造函数的调用语句**)，构造函数本身的代码之前。例如：

```
public class InstanceVariableInitializer {

    private int i = 1;
    private int j = i + 1;

    public InstanceVariableInitializer(int var){
        System.out.println(i);
        System.out.println(j);
        this.i = var;
        System.out.println(i);
        System.out.println(j);
    }

    {               // 实例代码块
        j += 3;

    }

    public static void main(String[] args) {
        new InstanceVariableInitializer(8);
    }
}/* Output:
            1
            5
            8
```

```
            5
 *///:~
```

上面的例子正好印证了上面的结论。特别需要注意的是，Java是按照编程顺序来执行实例变量初始化器和实例初始化器中的代码的，并且**不允许顺序靠前的实例代码块初始化在其后面定义的实例变量**，比如：

```
public class InstanceInitializer {
    {
        j = i;
    }

    private int i = 1;
    private int j;
}

public class InstanceInitializer {
    private int j = i;
    private int i = 1;
}
```

上面的这些代码都是无法通过编译的，编译器会抱怨说我们使用了一个未经定义的变量。之所以要这么做是为了保证一个变量在被使用之前已经被正确地初始化。但是我们仍然有办法绕过这种检查，比如：

```
public class InstanceInitializer {
    private int j = getI();
    private int i = 1;

    public InstanceInitializer() {
        i = 2;
    }

    private int getI() {
        return i;
    }

    public static void main(String[] args) {
        InstanceInitializer ii = new InstanceInitializer();
        System.out.println(ii.j);
    }
}
```

如果我们执行上面这段代码，那么会发现打印的结果是0。因此我们可以确信，变量j被赋予了i的默认值0，这一动作发生在实例变量i初始化之前和构造函数调用之前。

2. 构造函数初始化

   我们可以从上文知道，实例变量初始化与实例代码块初始化总是发生在构造函数初始化之前，那么我们下面着重看看构造函数初始化过程。众所周知，每一个Java中的对象都至少会有一个构造函数，如果我们没有显式定义构造函数，那么它将会有一个默认无参的构造函数。在编译生成的字节码中，这些构造函数会被命名成()方法，参数列表与Java语言书写的构造函数的参数列表相同。

我们知道，Java要求在实例化类之前，必须先实例化其超类，以保证所创建实例的完整性。事实上，这一点是在构造函数中保证的：Java强制要求Object对象(Object是Java的顶层对象，没有超类)之外的所有对象构造函数的第一条语句必须是超类构造函数的调用语句或者是类中定义的其他的构造函数，如果我们既没有调用其他的构造函数，也没有显式调用超类的构造函数，那么编译器会为我们自动生成一个对超类构造函数的调用，比如：

```
public class ConstructorExample {

}
```

对于上面代码中定义的类，我们观察编译之后的字节码，我们会发现编译器为我们生成一个构造函数，如下，

```
aload_0
invokespecial   #8; //Method java/lang/Object."<init>":()V
return
```

上面代码的第二行就是调用Object类的默认构造函数的指令。也就是说，如果我们显式调用超类的构造函数，那么该调用必须放在构造函数所有代码的最前面，也就是必须是构造函数的第一条指令。正因为如此，Java才可以使得一个对象在初始化之前其所有的超类都被初始化完成，并保证创建一个完整的对象出来。

特别地，如果我们在一个构造函数中调用另外一个构造函数，如下所示，

```
public class ConstructorExample {
    private int i;

    ConstructorExample() {
        this(1);
        ....
    }

    ConstructorExample(int i) {
        ....
        this.i = i;
        ....
    }
}
```

对于这种情况，Java只允许在ConstructorExample(int i)内调用超类的构造函数，也就是说，下面两种情形的代码编译是无法通过的：

```
public class ConstructorExample {
    private int i;

    ConstructorExample() {
        super();
        this(1);  // Error:Constructor call must be the first statement in a
constructor
        ....
    }

    ConstructorExample(int i) {
        ....
        this.i = i;
```

```
        ....
    }
}
```

```
public class ConstructorExample {
    private int i;

    ConstructorExample() {
        this(1);
        super();  //Error: Constructor call must be the first statement in a
constructor
        ....
    }

    ConstructorExample(int i) {
        this.i = i;
    }
}
```

Java通过对构造函数作出这种限制以便保证一个类的实例能够在被使用之前正确地初始化。

3. 小结

总而言之，实例化一个类的对象的过程是一个典型的递归过程，如下图所示。进一步地说，在实例化一个类的对象时，具体过程是这样的：

在准备实例化一个类的对象前，首先准备实例化该类的父类，如果该类的父类还有父类，那么准备实例化该类的父类的父类，依次递归直到递归到Object类。此时，首先实例化Object类，再依次对以下各类进行实例化，直到完成对目标类的实例化。具体而言，在实例化每个类时，都遵循如下顺序：先依次执行实例变量初始化和实例代码块初始化，再执行构造函数初始化。也就是说，**编译器会将实例变量初始化和实例代码块初始化相关代码放到类的构造函数中去，并且这些代码会被放在对超类构造函数的调用语句之后，构造函数本身的代码之前。**

4. 实例变量初始化、实例代码块初始化以及构造函数初始化综合实例

笔者在《JVM类加载机制概述：加载时机与加载过程》一文中详细阐述了类初始化时机和初始化过程，并在文章的最后留了一个悬念给各位，这里来揭开这个悬念。建议读者先看完《JVM类加载机制概述：加载时机与加载过程》这篇再来看这个，印象会比较深刻，如若不然，也没什么关系~~

```
//父类
class Foo {
    int i = 1;

    Foo() {
        System.out.println(i);              ----------(1)
        int x = getValue();
        System.out.println(x);              ----------(2)
    }

    {
        i = 2;
    }

    protected int getValue() {
        return i;
    }
}
```

```
//子类
class Bar extends Foo {
    int j = 1;

    Bar() {
        j = 2;
    }

    {
        j = 3;
    }

    @Override
    protected int getValue() {
        return j;
    }
}

public class ConstructorExample {
    public static void main(String... args) {
        Bar bar = new Bar();
        System.out.println(bar.getValue());             ----------(3)
    }
}/* Output:
            2
            0
            2
 *///:~
```

在通过使用Bar类的构造方法new一个Bar类的实例时，首先会调用Foo类构造函数，因此(1)处输出是2，这从Foo类构造函数的等价变换中可以直接看出。(2)处输出是0，为什么呢？因为在执行Foo的构造函数的过程中，由于Bar重载了Foo中的getValue方法，所以根据Java的多态特性可以知道，其调用的getValue方法是被Bar重写的那个getValue方法。但由于这时Bar的构造函数还没有被执行，因此此时j的值还是默认值0，因此(2)处输出是0 **（这其实涉及到编译时类型和运行时类型的问题，因为运行时的类型是子类，因而调用的是子类中重写的方法）**。最后，在执行(3)处的代码时，由于bar对象已经创建完成，所以此时再访问j的值时，就得到了其初始化后的值2，这一点可以从Bar类构造函数的等价变换中直接看出。

**重点**：编译时类型和运行时类型

**规则**：**对象调用编译时类型的属性和运行时类型的方法**，但是这个方法指的是重写后的方法，对于重载，**程序在编译器进行重载方法的选择是通过静态类型和参数数量决定的**

## 类的初始化时机与过程

关于类的初始化时机，笔者在博文《JVM类加载机制概述：加载时机与加载过程》已经介绍的很清楚了，此处不再赘述。简单地说，在类加载过程中，准备阶段是正式为类变量(static 成员变量)分配内存并设置类变量初始值（零值）的阶段，而初始化阶段是真正开始执行类中定义的java程序代码(字节码)并按程序猿的意图去初始化类变量的过程。更直接地说，初始化阶段就是执行类构造器()方法的过程。()方法是由编译器自动收集类中的所有类变量的赋值动作和静态代码块static{}中的语句合并产生的，其中编译器收集的顺序是由语句在源文件中出现的顺序所决定。

类构造器()与实例构造器()不同，它不需要程序员进行显式调用，虚拟机会保证在子类类构造器()执行之前，父类的类构造()执行完毕。由于父类的构造器()先执行，也就意味着父类中定义的静态代码块/静态变量的初始化要优先于子类的静态代码块/静态变量的初始化执行。特别地，类构造器()对于类或者接口来说并不是必需的，如果一个类中没有静态代码块，也没有对类变量的赋值操作，那么编译器可以不为这个类生产类构造器()。此外，在同一个类加载器下，一个类只会被初始化一次，但是一个类可以任意地实

例化对象。也就是说，在一个类的生命周期中，类构造器()最多会被虚拟机调用一次，而实例构造器()则会被虚拟机调用多次，只要程序员还在创建对象。

注意，这里所谓的实例构造器()是指收集类中的所有实例变量的赋值动作、实例代码块和构造函数合并产生的，类似于上文对Foo类的构造函数和Bar类的构造函数做的等价变换。

## 总结

1. 一个实例变量在对象初始化的过程中会被赋值几次?

   我们知道，JVM在为一个对象分配完内存之后，会给每一个实例变量赋予默认值，这个时候实例变量被第一次赋值，这个赋值过程是没有办法避免的。如果我们在声明实例变量x的同时对其进行了赋值操作，那么这个时候，这个实例变量就被第二次赋值了。如果我们在实例代码块中，又对变量x做了初始化操作，那么这个时候，这个实例变量就被第三次赋值了。如果我们在构造函数中，也对变量x做了初始化操作，那么这个时候，变量x就被第四次赋值。也就是说，在Java的对象初始化过程中，一个实例变量最多可以被初始化4次。

2. 类的初始化过程与类的实例化过程的异同?

   类的初始化是指类加载过程中的初始化阶段对类变量按照程序猿的意图进行赋值的过程；而类的实例化是指在类完全加载到内存中后创建对象的过程。

3. 假如一个类还未加载到内存中，那么在创建一个该类的实例时，具体过程是怎样的?

   我们知道，要想创建一个类的实例，必须先将该类加载到内存并进行初始化，也就是说，类初始化操作是在类实例化操作之前进行的，但并不意味着：只有类初始化操作结束后才能进行类实例化操作。例如，笔者在博文[《JVM类加载机制概述：加载时机与加载过程》]中所提到的下面这个经典案例：

```
public class StaticTest {
    public static void main(String[] args) {
        staticFunction();
    }

    static StaticTest st = new StaticTest();

    static {   //静态代码块
        System.out.println("1");
    }

    {        // 实例代码块
        System.out.println("2");
    }

    StaticTest() {    // 实例构造器
        System.out.println("3");
        System.out.println("a=" + a + ",b=" + b);
    }

    public static void staticFunction() {   // 静态方法
        System.out.println("4");
    }

    int a = 110;    // 实例变量
    static int b = 112;     // 静态变量
}/* Output:
        2
        3
        a=110,b=0
```

```
        1
        4
*///:~
```

大家能得到正确答案吗？笔者已经在博文《 JVM类加载机制概述：加载时机与加载过程》中解释过这个问题了，此不赘述。

总的来说，类实例化的一般过程是：父类的类构造器() -> 子类的类构造器() -> 父类的成员变量和实例代码块 -> 父类的构造函数 -> 子类的成员变量和实例代码块 -> 子类的构造函数。

## 方法调用

方法调用的讲解（涉及面试题）

1. `jvm`在运行中会把**符号引用**转化为方法的**直接引用**
    - 对于在**编译期**就确定，且在**运行过程中**不会改变的目标方法是静态链接
    - 被调用的方法在编译期无法确定下来，只能在**程序运行期间**才能将符号引用转化为对应的方法引用，我们成为动态链接，或者动态分派
2. **重载**需要根据静态类型来判断参数从而选择方法，**重写**则是根据实际类型来选择方法

## 静态分派、动态分派

## 继承

建议仔细看第二版

### 第一版

https://www.cnblogs.com/cj5785/p/10664866.html

1. 子类可以获得父类全部的成员变量和方法，而不能获得父类的构造器。如果没有显示指定一个类的父类，那么这个类默认继承java.lang.Object类，因此所有类都直接或间接继承自java.lang.Object，故可以调用java.lang.Object所定义的实例和方法。
2. **重点：重写**：在继承过程中，子类获得了父类的成员变量和方法，但有时父类的方法并不适合子类，此时子类可以重写父类的方法，这种现象称之为方法重写（override）。
    1. 重写遵循“**两同两小一大**”：**方法名相同，参数列表相同；返回值类型比父类更小或相等，声明抛出的异常比父类更小或相同；访问权限比父类更大或相等。**
    2. 重写的方法必须一致，要么都是类方法，要么都是实例方法。
    3. 当在子类中重写了父类的方法时，**子类将无法访问父类中被覆盖的方法，只能通过super关键字进行调用。**
    4. 如果父类方法具有private访问权限，那么该方法对子类隐藏，子类无法访问该方法，也就无法重写该方法。
    5. 此时如果定义一个与父类方法表面上相同的重写方法，此时依旧不是方法重写，只是新定义了一个方法。
    6. 方法重载（overload）和方法重写（override）区别在于**前者是指一个类里面的多个方法，后者是指父类与子类之间的方法。**

**Overload**：

1. 一个类中可以有多个方法具有相同的名字，但这些方法的参数必须不同，即参数个数、参数类型或参数顺序不同，返回类型可以相同也可以不同。
    - 方法名称必须相同；
    - 参数列表必须不同，即参数个数、参数类型或参数顺序中任有一个不同。
    - 方法的返回类型可以相同也可以不同，对此无限制。
    - 若仅满足方法的返回类型不同，不属于方法重载。
    - 仅仅参数名称不同，或方法类型不同，不构成重载。例如，int min(int x, int y)与int min(int z, int y)不构成方法重载，同理，int min(int x, int y)与void min(int z, int y)不构成方法重载。
    - 构造方法的名字与类名相同，且构造方法无返回类型，构造方法可以重载。
    - 声明为final的方法不能被重载。
    - 声明为static的方法不能被重载，但是能够被在此声明。
    - main 方法也可以被重载。
2. 方法重载的目的：
    - 方法重载的主要好处就是，不用为了对不同的参数类型或参数个数，而写多个函数。多个函数用同一个名字，但参数表，即参数的个数或(和)数据类型可以不同，调用的时候，虽然方法名字相同，但根据参数表可以自动调用对应的函数。
    - 重载的最直接作用是方便了程序员可以根据不同的参数个数，顺序，类型，自动匹配方法，减少写过个函数名或方法名的重复步骤。
3. 

| 区别点 | 重写 | 重载 |
| --- | --- | --- |
| 参数列表 | 必须不同 | 必须相同 |
| 返回类型 | 无限制 | 必须相同 |
| 异常 | 可以修改 | 异常只能减少或删除 |
| 访问权限 | 可以修改 | 不可以降低方法的访问权限 |

3. 如果父类方法是public，那么即使不加@Override，只要格式上一样就会构成重写。如果要避免这种情况，就把父类的改成private。
4. super用于限定**该对象**调用它从父类继承得到的实例变量或方法。与this一样，super也不能出现在static修饰的方法中**（为什么）**。如果在构造器中使用super，那么**super用于限定该构造器初始化的是该对象从父类继承得到的实例变量**。这种情况常用于父类成员变量被子类覆盖而又需要使用父类的成员变量的时候。
5. 在某个方法中访问名为a的成员变量，但却没有显式指定调用者，那么系统查找a的顺序为：当前方法的局部变量 -> 当前类的成员变量 -> 当前类父类的成员变量 … -> java.lang.Object，如果依旧找不到，那么编译将会报错。
6. 子类的所有成员变量在实例被创建时会分配内存空间，父类的成员变量也会分配内存空间。且二者不存在子类覆盖父类内存空间的问题。
7. 子类不会获得父类的构造器，但子类可以调用父类构造器来初始化代码。此时使用super来完成。此时super必须位于子类构造器的第一行，**故super与this不可同时存在于一个构造器中**。无论是否使用super调用来执行父类构造器的初始化方法，子类构造器**总会**调用父类构造器一次，且都会在子类构造器方法体执行之前执行。**（仔细思考为什么*super*和*this*不兼容）**
8. super关键字
    - super和this的用法相同
    - this代表本类引用**（细节来说应该是本对象）**
    - super代表父类引用
    - 当子父类出现同名成员时，可以用super进行区分

- 子类要调用父类构造函数时，可以使用super语句
- 子类在构造函数调用时，会首先调用父类的构造函数，相当于在子类构造函数中使用了super();
- 所有的子类构造函数**默认第一句**都是super();
- 子类的所有的构造函数，默认都会访问父类中空参数的构造函数

9. 函数覆盖(Override)
    - 子类中出现与父类一模一样的方法时，会出现覆盖操作，也称为重写或者复写
    - 父类中的私有方法不可以被覆盖
    - 在子类覆盖方法中，继续使用被覆盖的方法可以通过super.函数名获取
    - 覆盖注意事项：
        - 覆盖时，子类方法权限一定要大于等于父类方法权限
        - 静态只能覆盖静态
    - 覆盖的应用：当子类需要父类的功能，而功能主体子类有自己特有内容时，可以复写父类中的方法，这样，即沿袭了父类的功能，又定义了子类特有的内容
10. 子类的实例化过程

- 子类中所有的构造函数默认都会访问父类中空参数的构造函数
- 因为每一个构造函数的第一行都有一条默认的语句super();
- 子类会具备父类中的数据，所以要先明确父类是如何对这些数据初始化的
- 当父类中没有空参数的构造函数时，子类的构造函数必须通过 this或者super语句指定要访问的构造函数

11. 继承与组合

    继承是实现复用的重要手段，但在使用继承的时候，却破坏了封装。组合也是实现复用的重要方式，却能提供更好的封装性

    - 为了保证父类良好的封装性，通常在设计父类时应遵循以下原则：
        - 尽量隐藏父类的内部数据。尽量将父类的成员变量设置为private
        - 不要让子类可以任意访问、修改父类的方法。仅作为父类的辅助方法，可以用private修饰；若希望被外部访问却不希望子类改写，则可以使用final修饰；若只是被子类修改而不希望其他类自由访问，则使用protected修饰
        - 尽量不要在父类构造器中调用将要被子类重写的方法，调用的话实际执行的就是子类的方法了。
    - 组合是把旧的对象作为新类的成员变量组合起来，用以实现复用的功能，用户可以看到新类的方法，而无法知晓被组合对象的方法
    - 继承和组合的区别：继承要表达的是一种 `is-a` 的关系，而组合表达的是一种 `has-a` 的关系

12. final修饰符

    final关键字用于修饰类，变量和方法，表示修饰的类、方法和变量不可改变。final修饰的变量不可被改变，一旦获取了初始值，该final变量的值就不能被重新赋值

    - final可以修饰类，方法，变量
    - final修饰的类不可以被继承
    - final修饰的方法不可以被覆盖
    - final修饰的变量是一个常量。只能被赋值一次
    - 内部类只能访问被final修饰的局部变量
    - final修饰成员变量：一旦有了初始值，就不能被重新赋值。系统默认分配的值为0，'\u0000'，false，null。在Java语法中规定：final修饰的成员变量必须由程序员**显式**地指定初始值。类变量必须在**静态初始化块中或者声明变量的时候指定初始化值**；实例变量必须在非静态代码块，声明实例变量或构造器中指定初始化值

- final修饰局部变量：如果final修饰的局部变量在定义时没有指定默认值，则可以在后面的代码中对该final变量赋初始值，但只能一次，不能重复赋值。若定义时候已经赋值，则不可再次赋值
- final修饰基本数据类型变量和引用类型变量的区别：final修饰基本数据类型时，不能对基本数据类型重新赋值，基本数据类型变量不能被改变。final修饰引用数据类型时，他保存的仅仅是一个引用，**保存的地址不变，但地址所在处的内容却可以改变**。换句话说final修饰的变量只是直接保存的内容不可改变
- final如果满足以下三种情况，则相当于一个**直接量 （直接量啥意思）**：
  - 使用final修饰符修饰
  - 在定义该final变量时指定了初始值
  - 该初始值可以在编译时就被确定下来
- final方法：final修饰的方法不可被重写，一般用于限制子类的重写。Object中的getClass()就是一个final方法。父类的final方法若为public，则子类出现与父类形式上构成重写关系的方法时将出现错误。若要实现子类中类似父类的方法，则在父类中的方法添加private修饰符。此时子类并不是重写父类方法，而是定义了一个新方法。另外，final修饰的方法是可以重载的，仅仅是不能被重写
- final类：final修饰的类不可以有子类。例如java.lang.Math
- 不可变类（immutable）：创建该类的实例后，该实例的实例变量是不可改变的。Java提供的8个包装类和java.lang.String类都是不可变类。如果创建不可变类，应遵循以下原则：
  - 使用private和final修饰符来修饰该类的成员变量
  - 提供带参数的构造器，用于根据传入的参数来初始化类里的成员变量
  - 仅为该类的成员变量提供getter方法，不要为该类的成员提供setter方法，因为普通方法无法修改final修饰的成员变量
  - 如果有必要，重写Object类的hashCode()和equals()方法。equals()方法更具关键成员变量来作为两个对象是否相等的标准，除此之外，还应该保证两个用equals()方法判断为相等的对象的hashCode()也想等
- 缓存实例的不可变类：不可变类的实例状态不可改变，可以很方便的被多个对象所共享。如果可能，应该将已创建的不可变类的实例进行缓存

## 第二版

这个更好

1. 成员变量的继承

   当子类继承了某个类之后，便可以使用父类中的成员变量，但是并不是完全继承父类的所有成员变量。具体的原则如下：

   - 子类能够继承父类的 public 和 protected 成员变量 ，不能够继承父类的 private 成员变量，但可以通过父类相应的getter/setter方法进行访问；对于protected，不同包的子类可以调用子类对象访问父类的东西，但是不能调用父类对象访问父类东西。
   - 对于父类的包访问权限成员变量（package），如果子类和父类在同一个包下，则子类能够继承，否则，子类不能够继承；
   - 对于子类可以继承的父类成员变量，如果在子类中出现了同名称的成员变量，则会发生 **隐藏** 现象，即子类的成员变量会屏蔽掉父类的同名成员变量。如果要在子类中访问父类中同名成员变量，需要使用super关键字来进行引用。（好像说的不太对）
2. 成员方法的继承
   同样地，当子类继承了某个类之后，便可以使用父类中的成员方法，但是子类并不是完全继承父类的所有方法。具体的原则如下：

   - 子类能够继承父类的 public和protected成员方法 ，不能够继承父类的 private成员方法；
   - 对于父类的包访问权限成员方法，如果子类和父类在同一个包下，则子类能够继承，否则，子类不能够继承；

- 对于子类可以继承的父类成员方法，如果在子类中出现了同名称的成员方法，则称为 **覆盖** ，即子类的成员方法会覆盖掉父类的同名成员方法。如果要在子类中访问父类中同名成员方法，需要使用super关键字来进行引用。

```
class Person {
    public String gentle = "Father";
}

public class Student extends Person {

    public String gentle = "Son";

    public String print(){
        return super.gentle;        // 在子类中访问父类中同名成员变
    }

    public static void main(String[] args) throws ClassNotFoundException {
        Student student = new Student();
        System.out.println("##### " + student.gentle);
        Person p = student;
        System.out.println("***** " + p.gentle);     //隐藏：编译时决定，不会发生多态

        System.out.println("----- " + student.print());
        System.out.println("----- " + p.print());     //Error：Person 中未定义该方法
    }
}/* Output:
        ##### Son
        ***** Father
        ----- Father
 *///:~
```

**隐藏和覆盖是不同的。 隐藏** 是 **针对成员变量和静态方法** 的，而 **覆盖** 是 **针对普通方法** 的。经测试，static方法也是依据编译时类型决定。

3. 基类的初始化与构造器

   我们知道，导出类就像是一个与基类具有相同接口的新类，或许还会有一些额外的方法和域。但是，继承并不只是复制基类的接口。当创建一个导出类对象时，该对象会包含一个基类的子对象。这个子对象与我们用基类直接创建的对象是一样的。二者的区别在于，后者来自于外部，而基类的子对象被包装在导出类对象的内部。

   因此，对基类子对象的正确初始化是至关重要的，并且Java也提供了相应的方法来保证这一点： 导出类必须在构造器中调用基类构造器来执行初始化，而基类构造器具有执行基类初始化所需的所有知识和能力。当基类含有默认构造器时，Java会自动在导出类的构造器插入对该基类默认构造器的调用，因为编译器不必考虑要传递什么样的参数的问题。但是，**若父类不含有默认构造器**，或者导出类想调用一个带参数的父类构造器，那么在导出类的构造器中就**必须**使用 super 关键字显式的进行调用相应的基类的构造器，并且该调用语句必是导出类构造器的第一条语句。

**组合，继承，代理**

在Java中，**组合、继承和代理三种技术都可以实现代码的复用。**

1. 组合（has-a）

   通过在新的类中加入现有类的对象即可实现组合。即，新的类是由现有类的对象所组成。该技术通常用于想在新类中使用现有类的功能而非它的接口这种情形。也就是说，在新类中嵌入某个对象，让其实现所需要的功能，但新类的用户看到的只是为新类所定义的接口，而非所嵌入对象的接口。

2. 继承（is-a）

   继承可以使我们按照现有类的类型来创建新类。即，我们采用现有类的形式并在其中添加新代码。通常，**这意味着我们在使用一个通用类，并为了某种特殊需要而将其特殊化。**本质上，**组合和继承都允许在新的类中放置子对象，组合是显式地这样做，而继承则是隐式地做。**

3. 代理（继承与组合之间的一种中庸之道：像组合一样使用已有类的功能，同时像继承一样使用已有类的接口）

   代理是继承与组合之间的一种中庸之道，Java并没有提供对它的直接支持。在代理中，我们将一个成员对象置于所要构造的类中（就像组合），但与此同时我们在新类中暴露了该成员对象的接口/方法（就像继承）。

```
// 控制模块
public class SpaceShipControls {
    void up(int velocity) {
    }

    void down(int velocity) {
    }

    void left(int velocity) {
    }

    void right(int velocity) {
    }

    void forward(int velocity) {
    }

    void back(int velocity) {
    }

    void turboBoost() {
    }
}
```

太空船需要一个控制模块，那么，**构造太空船的一种方式是使用继承：**

```
public class SpaceShip extends SpaceShipControls {
    private String name;
    public SpaceShip(String name) { this.name = name; }
    public String toString() { return name; }
    public static void main(String[] args) {
        SpaceShip protector = new SpaceShip("NSEA Protector");
        protector.forward(100);
    }
}
```

然而，SpaceShip 并不是真正的 SpaceShipControls 类型，即便你可以“告诉” SpaceShip 向前运动（forward()）。更准确的说，SpaceShip 包含 SpaceShipControls，与此同时， SpaceShipControls 的所有方法在 SpaceShip 中都暴露出来。 代理（SpaceShip 的运动行为由 SpaceShipControls 代理完成） 正好可以解决这种问题：

```
// SpaceShip 的行为由 SpaceShipControls 代理完成
public class SpaceShipDelegation {
    private String name;
```

```
    private SpaceShipControls controls = new SpaceShipControls();

    public SpaceShipDelegation(String name) {
        this.name = name;
    }

    // 代理方法:
    public void back(int velocity) {
        controls.back(velocity);
    }
    public void down(int velocity) {
        controls.down(velocity);
    }
    public void forward(int velocity) {
        controls.forward(velocity);
    }
    public void left(int velocity) {
        controls.left(velocity);
    }
    public void right(int velocity) {
        controls.right(velocity);
    }
    public void turboBoost() {
        controls.turboBoost();
    }
    public void up(int velocity) {
        controls.up(velocity);
    }

    public static void main(String[] args) {
        SpaceShipDelegation protector = new SpaceShipDelegation("NSEA
Protector");
        protector.forward(100);
    }
}
```

实际上，**我们使用代理时可以拥有更多的控制力，因为我们可以选择只提供在成员对象中方法的某个子集。**

**final**

许多编程语言都需要某种方法来向编译器告知一块数据是恒定不变的。有时，数据的恒定不变是很有用的，比如：

- 一个永不改变的编译时常量；
- 一个在运行时被初始化的值，而你不希望它被改变。

对于编译期常量这种情况，编译器可以将该常量值带入任何可能用到它的计算式中，也即是说，可以在编译时执行计算式，这减轻了一些运行时负担。在Java中，这类常量必须满足两个条件：

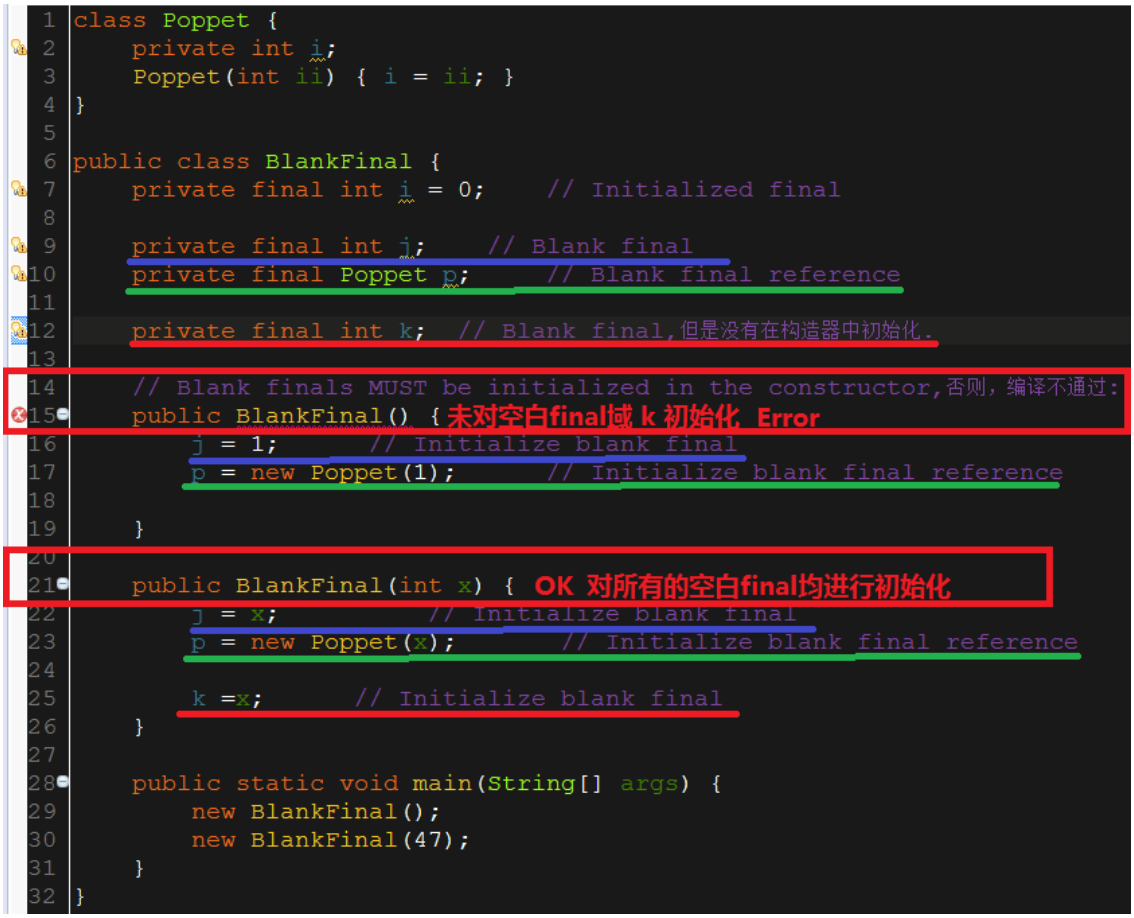
- 是基本类型，并且用final修饰；
- 在对这个常量进行定义的时候，必须对其进行赋值。

此外，当用final修饰对象引用时，final使其**引用**恒定不变。一旦引用被初始化指向一个对象，就无法再把它指向另一个对象。然而，**对象本身是可以被修改的，这同样适用于数组，因为它也是对象。**

特别需要注意的是，我们不能因为某数据是final的，就认为在编译时就可以知道它的值。例如：final int i4 = rand.nextInt(20);

1. 空白final

   Java允许生成 空白final , 即：声明final但又未给定初值的域。但无论什么情况，编译器都会确保空白final在使用前被初始化。但是，空白final在关键字final的使用上提供了更大的灵活性: 一个类中的 final域 就可以做到根据对象而有所不同，却又保持其恒定不变的特性。例如，




   必须在域的定义处或者每个构造器中使用表达式对final进行赋值，这正是 final域 在使用前总是被初始化的原因所在。

2. final参数

   final参数 主要应用于局部内部类和匿名内部类中，更多详细介绍请移步我的另一篇文章：Java 内部类综述。

3. final方法

   final关键字作用于方法时，用于锁定方法，以防任何继承类修改它的含义。这是出于设计的考虑：想要确保在继承中使方法行为保持不变，并且不会被覆盖。

   对于成员方法，只有在明确禁止覆盖时，才将方法设为final的。

4. final类

   当将某个类定义为final时，就表明你不打算继承该类，而且也不允许别人这样做。换句话说，出于某种考虑，你对该类的设计永不需要做任何变动，或者出于安全考虑，你不希望它有子类。

   需要注意的是，final类的域可以根据实际情况选择是否为final的。不论是否被定义为final，相同的规则都适用于定义final的域。然而，由于final类禁止继承，所以final类中的**所有方法都隐式指定为final**的，因为无法覆盖它们。在final类中可以给方法添加final修饰，但这不会增添任何意义。

5. final与private

   类中所有的private方法都**隐式地指定为final**的。由于无法取用private方法，所以也就无法覆盖它。可以对private方法添加final修饰，但这并不会给该方法添加任何额外的意义。

特别需要注意的是，覆盖只有在某方法是基类接口的一部分时才会出现。如果一个方法是private的，它就不是基类接口中的一部分，而仅仅是一些隐藏于类中的程序代码。但若在导出类中以相同的名称生成一个非private方法，此时我们并没有覆盖该方法，仅仅是**生成了一个新的方法。**由于private方法无法触及并且能有效隐藏，所以除了把它看成是由于它所归属的类的组织结构的原因而存在外，其他任何情况都不需要考虑它。

6. final 与 static

   static 修饰变量时，其 具有默认值， 且 可改变， 且其 只能修饰成员变量和成员方法。

   一个 static final域 只占据一段不能改变的存储空间，且只能在声明时进行初始化。**因为其是 final 的，因而没有默认值**；且又是static的，因此在类没有实例化时，其已被赋值，所以只能在声明时初始化。

**多态**

我们知道 继承允许将对象视为它自己本身的类型或其基类型加以处理，从而使同一份代码可以毫无差别地运行在这些不同的类型之上。其中，多态方法调用允许一种类型表现出与其他相似类型之间的区别，只要这些类型由同一个基类所导出。所以，多态的作用主要体现在两个方面：

- 多态通过分离做什么和怎么做，从另一个角度将接口和实现分离开来，从而实现将改变的事物与未变的事物分离开来；
- 消除类型之间的耦合关系（类似的，在Java中，泛型也被用来消除类或方法与所使用的类型之间的耦合关系）。

1. 实现机制

   我们知道方法的覆盖很好的体现了多态，但是当使用一个基类引用去调用一个覆盖方法时，到底该调用哪个方法才正确呢？

   将一个方法调用同一个方法主体关联起来被称作绑定。若在程序执行前进行绑定，叫做 前期绑定 。但是，显然，这种机制并不能解决上面的问题，因为在编译时编译器并不知道上述基类引用到底指向哪个对象。解决的办法就是后期绑定(动态绑定/运行时绑定)：在运行时根据对象的具体类型进行绑定。

   事实上，在Java中，除了static方法和final方法（private方法属于final方法）外，**其他所有的方法都是后期绑定。**这样，一个方法声明为final后，可以防止其他人覆盖该方法，但更重要一点是：这样做可以有效地关闭动态绑定，或者说，告诉编译器不需要对其进行动态绑定，以便为final方法调用生成更有效的代码。

   基于动态绑定机制，我们就可以编写只与基类打交道的代码了，并且这些代码对所有的导出类都可以正确运行。或者说，发送消息给某个对象，让该对象去断定该做什么事情。

2. 向下转型与运行时类型识别

   由于向上转型会丢失具体的类型信息，所以我们可能会想，通过向下转型也应该能够获取类型信息。然而，我们知道向上转型是安全的，因为基类不会具有大于导出类的接口。因此，我们通过基类接口发送的消息都能被接受，但是对于向下转型，我们就无法保证了。

   要解决这个问题，必须有某种方法来确保向下转型的正确性，使我们不至于贸然转型到一种错误的类型，进而发出该对象无法接受的消息。在Java中，运行时类型识别（RTTI）机制可以处理这个问题，它保证Java中所有的转型都会得到检查。所以，即使我们只是进行一次普通的加括弧形式的类型转换，再进入运行期时仍会对其进行检查，以便保证它的确是我们希望的哪种类型。如果不是，我们就会得到一个类型转换异常：ClassCastException。

**类加载及初始化顺序**

首先，必须指出类加载及初始化顺序为：父类静态代码块->子类静态代码块->父类非静态代码块->父类构造函数->子类非静态代码块->子类构造函数

即，首先，初始化父类中的静态成员变量和静态代码块，按照在程序中出现的顺序初始化；然后，初始化子类中的静态成员变量和静态代码块，按照在程序中出现的顺序初始化；其次，初始化父类的普通成员变量和代码块，再执行父类的构造方法；最后，初始化子类的普通成员变量和代码块，再执行子类的构造方法。

我们通过下面一段程序说明：

```
class SuperClass {
    private static String STR = "Super Class Static Variable";
    static {
        System.out.println("Super Class Static Block:" + STR);
    }

    public SuperClass() {
        System.out.println("Super Class Constructor Method");
    }

    {
        System.out.println("Super Class Block");
    }

}

public class ObjectInit extends SuperClass {
    private static String STR = "Class Static Variable";
    static {
        System.out.println("Class Static Block:" + STR);
    }

    public ObjectInit() {
        System.out.println("Constructor Method");
    }

    {
        System.out.println("Class Block");
    }

    public static void main(String[] args) {
        @SuppressWarnings("unused")
        ObjectInit a = new ObjectInit();
    }
}/* Output:
        Super Class Static Block:Super Class Static Variable
        Class Static Block:Class Static Variable
        Super Class Block
        Super Class Constructor Method
        Class Block
        Constructor Method
 *///:~
```

在运行该程序时，所发生的第一件事就是试图访问 ObjectInit.main() 方法（一个static方法），于是加载器开始启动并加载 ObjectInit类。在对其加载时，编译器注意到它有一个基类（这由关键字extends得知），于是先进行加载其基类。如果该基类还有其自身的基类，那么先加载这个父父基类，如此类推（本例中是先加载 Object类，再加载 SuperClass类，最后加载 ObjectInit类）。接下来，根基类中的 static域 和 static代码块 会被执行，然后是下一个导出类，以此类推这种方式很重要，因为导出类的

static初始化可能会依赖于基类成员能否被正确初始化。**到此为止，所有的类都已加载完毕，对象就可以创建了。**首先，初始化根基类所有的普通成员变量和代码块，然后执行根基类构造器以便创建一个基对象，然后是下一个导出类，依次类推，直到初始化完成。

**重载、覆盖与隐藏**

1. 重载与覆盖
    1. 定义与区别

        重载：如果在一个类中定义了多个同名的方法，但它们有不同的参数（包含三方面：参数个数、参数类型和参数顺序），则称为方法的重载。其中，不能通过访问权限、返回类型和抛出异常进行重载。

        覆盖：子类中定义的某个方法与其父类中某个方法具有相同的方法签名（包含相同的名称和参数列表），则称为方法的覆盖。子类对象使用这个方法时，将调用该方法在子类中的定义，对它而言，父类中该方法的定义被屏蔽了。

        总的来说，重载和覆盖是Java多态性的不同表现。前者是一个类中多态性的一种表现，后者是父类与子类之间多态性的一种表现。

    2. 实现机制

        **重载是一种参数多态机制**，即通过方法参数的差异实现多态机制。并且，其属于一种 **静态绑定机制**，在编译时已经知道具体执行哪个方法。

        **覆盖是一种动态绑定的多态机制。**即，在父类与子类中具有相同签名的方法具有不同的具体实现，至于最终执行哪个方法 **根据运行时的实际情况而定。**

    3. 总结

        我们应该注意以下几点：

        - **final 方法不能被覆盖；**
        - **子类不能覆盖父类的private方法，否则，只是在子类中定义了一个与父类重名的全新的方法，而不会有任何覆盖效果。**
2. 覆盖与隐藏
    1. 定义

        **覆盖**：指 **运行时系统调用当前对象引用 运行时类型** 中定义的方法 ， 属于 **运行期绑定。**

        **隐藏**：指**运行时系统调用当前对象引用 编译时类型** 中定义的方法，即 **被声明或者转换为什么类型就调用对应类型中的方法或变量**，属于**编译期绑定。**

    2. 范围

        **覆盖**：**只针对实例方法；**
        **隐藏**：**只针对静态方法和成员变量。**

    3. 小结

        - **子类的实例方法不能隐藏父类的静态方法，同样地，子类的静态方法也不能覆盖父类的实例方法，否则编译出错；**
        - **无论静态成员还是实例成员，都能被子类同名的成员变量所隐藏。**

# String详解

## Java 内存模型 与 常量池

1. Java内存模型

1. 程序计数器

   多线程时，当线程数超过CPU数量或CPU内核数量，线程之间就要根据时间片轮询抢夺CPU时间资源。因此，每个线程要有一个独立的程序计数器，记录下一条要运行的指令，其为线程私有的内存区域。如果执行的是JAVA方法，计数器记录正在执行的java字节码地址，如果执行的是native方法，则计数器为空。

2. 虚拟机栈

   线程私有的，与线程在同一时间创建，是管理JAVA方法执行的内存模型。栈中主要存放一些基本类型的变量数据（int, short, long, byte, float, double, boolean, char）和对象引用。每个方法执行时都会创建一个桢栈来存储方法的的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法，-Xss参数可以设置虚拟机栈大小）。栈的大小可以是固定的，或者是动态扩展的。如果请求的栈深度大于最大可用深度，则抛出stackOverflowError；如果栈是可动态扩展的，但没有内存空间支持扩展，则抛出OutofMemoryError。使用jclasslib工具可以查看class类文件的结构。

3. 本地方法区

   和虚拟机栈功能相似，但管理的不是JAVA方法，是本地方法，本地方法是用 C 实现的。

4. JAVA堆

   线程共享的，存放所有对象实例和数组，是垃圾回收的主要区域。堆是一个运行时数据区,类的对象从中分配空间,这些对象通过new、newarray、 anewarray 和 multianewarray等指令建立，它们不需要程序代码来显式的释放。堆可以分为新生代和老年代(tenured)。新生代用于存放刚创建的对象以及年轻的对象，如果对象一直没有被回收，生存得足够长，老年对象就会被移入老年代。新生代又可进一步细分为eden(伊甸园)、survivorSpace0(s0,from space)、survivorSpace1(s1,tospace)。刚创建的对象都放入eden,s0和s1都至少经过一次GC并幸存。如果幸存对象经过一定时间仍存在，则进入老年代(tenured)。

5. 方法区

   线程共享的，用于存放被虚拟机加载的类的元数据信息：如**常量、静态变量、即时编译器编译后的代码**，也成为永久代。如果hotspot虚拟机确定一个类的定义信息不会被使用，也会将其回收。回收的基本条件至少有：所有该类的实例被回收，而且装载该类的ClassLoader被回收。

6. 常量池

   常量池属于类信息的一部分，而类信息反映到 JVM 内存模型中对应于方法区，也就是说，常量池位于方法区。常量池主要存放两大常量：**字面量(Literal) 和 符号引用(Symbolic References)**。其中，字面量主要包括**字符串字面量**，**整型字面量** 和 **声明为final的常量值** 等；而符号引用则属于编译原理方面的概念，包括了下面三类常量：

   - 类和接口的全限定名
   - 字段的名称和描述符
   - 方法的名称和描述符

# 变量与常量

我们一般把**内存地址不变,值可以改变的东西称为变量**，换句话说，在内存地址不变的前提下内存的内容是可变的，例如：

```
public class String_2 {
    public static void f(){
        Human_1 h = new Human_1(1,30);
```

```
        Human_1 h2 = h;
        System.out.printf("h: %s\n", h.toString());
        System.out.printf("h2: %s\n\n", h.toString());

        h.id = 3;
        h.age = 32;
        System.out.printf("h: %s\n", h.toString());
        System.out.printf("h2: %s\n\n", h.toString());

        System.out.println( h == h2 );   // true : 引用值不变，即对象内存底子不变，但内容改变
    }
}
```

我们一般把**若内存地址不变，则值也不可以改变的东西称为常量**，典型的 String 就是不可变的，所以称之为 常量(constant)。此外，我们可以通过final关键字来定义常量，但严格来说，只有基本类型被其修饰后才是常量（对基本类型来说是其值不可变，而对于对象变量来说其引用不可再变）。

```
final int i = 5;
```

# String 定义与基础

1. 字符串声明

   由 JDK 中关于String的声明可以知道：

   不同字符串可能共享同一个底层char数组，例如字符串 String s="abc" 与 s.substring(1) 就共享同一个char数组：char[] c = {'a','b','c'}。其中，前者的 offset 和 count 的值分别为0和3，后者的 offset 和 count 的值分别为1和2。
   offset 和 count 两个成员变量不是多余的，比如，在执行substring操作时。

2. String不属于八种基本数据类型，String 的实例是一个对象。因为对象的默认值是null，所以String的默认值也是null

3. new String() 和 new String("")都是声明一个**新的空字符串**，是空串不是null; 、

# String 的不可变性

1. 什么是不可变对象?

   众所周知，在Java中，String类是不可变类 (基本类型的包装类都是不可改变的) 的典型代表，也是Immutable设计模式的典型应用。String变量一旦初始化后就不能更改，禁止改变对象的状态，从而增加共享对象的坚固性、减少对象访问的错误，同时还避免了在多线程共享时进行同步的需要。那么，到底什么是不可变的对象呢？ 可以这样认为：如果一个对象，在它创建完成之后，不能再改变它的状态，那么这个对象就是不可变的。不能改变状态指的是不能改变对象内的成员变量，包括：

   - **基本数据类型的值不能改变;**
   - **引用类型的变量不能指向其他的对象;**
   - **引用类型指向的对象的状态也不能改变;**

   除此之外，还应具有以下特点：

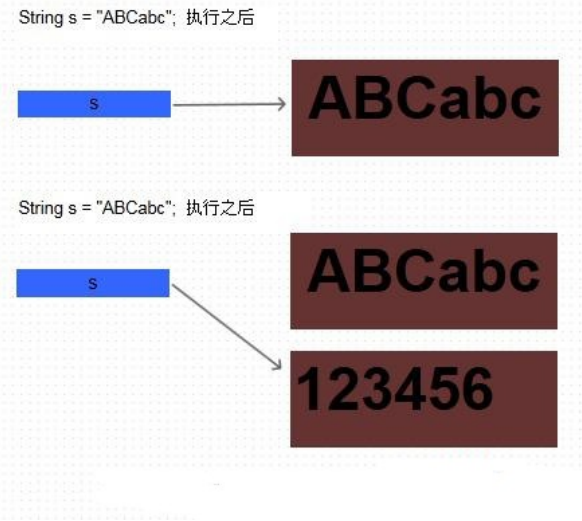
   - **除了构造函数之外，不应该有其它任何函数（至少是任何public函数）修改任何成员变量;**
   - **任何使成员变量获得新值的函数**都应该**将新的值保存在新的对象中，而保持原来的对象不被修改。**

2. 区分引用和对象

对于Java初学者， 对于String是不可变对象总是存有疑惑。看下面代码：

```
String s = "ABCabc";
System.out.println("s = " + s);    // s = ABCabc

s = "123456";
System.out.println("s = " + s);    // s = 123456
```

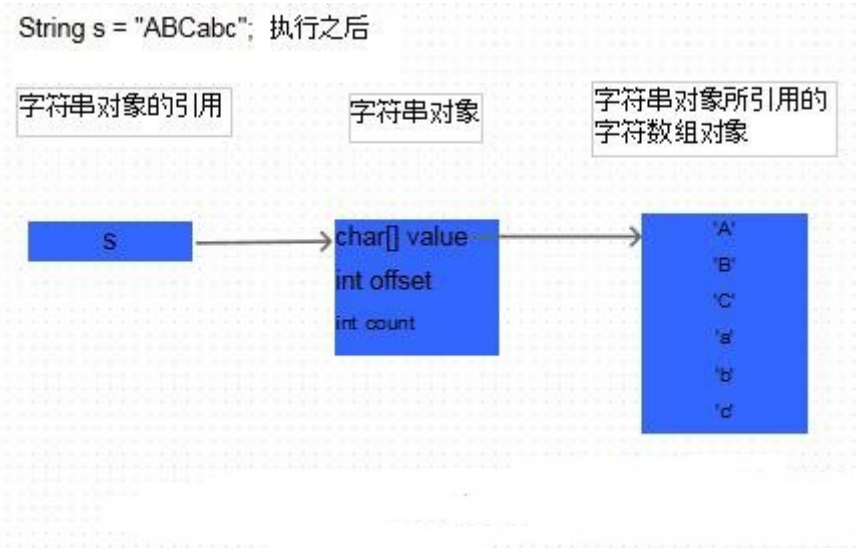
首先创建一个String对象s，然后让s的值为“ABCabc”， 然后又让s的值为“123456”。 从打印结果可以看出，s的值确实改变了。那么怎么还说String对象是不可变的呢？ 其实这里存在一个误区： **s只是一个String对象的引用**，并不是对象本身。对象在内存中是一块内存区，成员变量越多，这块内存区占的空间越大。引用只是一个 4 字节的数据，里面存放了它所指向的对象的地址，通过这个地址可以访问对象。 也就是说，s只是一个引用，它指向了一个具体的对象，当s=“123456”; 这句代码执行过之后，又创建了一个新的对象“123456”， 而引用s重新指向了这个新的对象，原来的对象“ABCabc”还在内存中存在，并没有改变。内存结构如下图所示：




Java和C++的一个不同点是，**在 Java 中，引用是访问、操纵对象的唯一方式： 我们不可能直接操作对象本身，所有的对象都由一个引用指向，必须通过这个引用才能访问对象本身，**包括获取成员变量的值，改变对象的成员变量，调用对象的方法等。而在C++中存在引用，对象和指针三个东西，这三个东西都可以访问对象。其实，Java中的引用和C++中的指针在概念上是相似的，他们都是存放的对象在内存中的地址值，只是在Java中，引用丧失了部分灵活性，比如Java中的引用不能像C++中的指针那样进行加减运算。

3. 为什么String对象不可变

在Java中，String类其实就是对字符数组的封装。JDK6中， value是String封装的数组，offset是String在这个value数组中的起始位置，count是String所占的字符的个数。在JDK7中，只有一个value变量，也就是value中的所有字符都是属于String这个对象的。这个改变不影响本文的讨论。除此之外还有一个hash成员变量，是该String对象的哈希值的缓存，这个成员变量也和本文的讨论无关。在Java中，数组也是对象（可以参考我之前的文章java中数组的特性）。 所以value也只是一个引用，它指向一个真正的数组对象。其实执行了String s = “ABCabc”; 这句代码之后，真正的内存布局应该是这样的：




value，offset和count这三个变量都是 private 的，并且没有提供setValue，setOffset和setCount等公共方法来修改这些值，所以在String类的外部无法修改String。也就是说一旦初始化就不能修改， 并且在String类的外部不能访问这三个成员。此外，value，offset和count这三个变量都是**final**的， 也就是说在String类内部，一旦这三个值初始化了， 也不能被改变。所以，可以认为String对象是不可变的了。（变量全是final的）

那么在String中，明明存在一些方法，调用他们可以得到改变后的值。这些方法包括substring，replace， replaceAll， toLowerCase等。例如如下代码：

```
String a = "ABCabc";
System.out.println("a = " + a);    // a = ABCabc

a = a.replace('A', 'a');
System.out.println("a = " + a);    //a = aBCabc
```

那么a的值看似改变了，其实也是同样的误区。再次说明， a只是一个引用， 不是真正的字符串对象，在调用a.replace('A', 'a')时， 方法内部创建了一个新的String对象，并把这个心的对象重新赋给了引用a。String中replace方法的源码可以说明问题。

我们可以自己查看其他方法，都是在方法内部重新创建新的String对象，并且返回这个新的对象，原来的对象是不会被改变的。这也是为什么像replace， substring，toLowerCase等方法都存在返回值的原因。也是为什么像下面这样调用不会改变对象的值：

```
String ss = "123456";
System.out.println("ss = " + ss);     // ss = 123456


ss.replace('1', '0');
System.out.println("ss = " + ss);     //ss = 123456
```

4. String对象真的不可变吗?

从上文可知String的成员变量是 private final 的，也就是初始化之后不可改变。那么在这几个成员中， value比较特殊，因为他是一个引用变量，而不是真正的对象。value是final修饰的，也就是说final不能再指向其他数组对象，那么我能改变value指向的数组吗？ 比如，将数组中的某个位置上的字符变为下划线"_"。 至少在我们自己写的普通代码中不能够做到，因为我们根本不能够访问到这个value引用，更不能通过这个引用去修改数组，那么，用什么方式可以访问私有成员呢？ 没错，用反射，可以反射出String对象中的value属性， 进而改变通过获得的value引用改变数组的结构。下面是实例代码：

```
public static void testReflection() throws Exception {

    //创建字符串"Hello World",  并赋给引用s
    String s = "Hello World";

    System.out.println("s = " + s); //Hello World

    //获取String类中的value字段
    Field valueFieldOfString = String.class.getDeclaredField("value");

    //改变value属性的访问权限
    valueFieldOfString.setAccessible(true);

    //获取s对象上的value属性的值
    char[] value = (char[]) valueFieldOfString.get(s);

    //改变value所引用的数组中的第5个字符
    value[5] = '_';

    System.out.println("s = " + s);  //Hello_World
}
```

在这个过程中，s始终引用的同一个String对象，但是再反射前后，这个String对象发生了变化， 也就是说，通过反射是可以修改所谓的“不可变”对象的。但是一般我们不这么做。这个反射的实例还可以说明一个问题：如果一个对象，他组合的其他对象的状态是可以改变的，那么这个对象很可能不是不可变对象。例如一个Car对象，它组合了一个Wheel对象，虽然这个Wheel对象声明成了private final 的，但是这个Wheel对象内部的状态可以改变， 那么就不能很好的保证Car对象不可变。

## String 对象创建方式

1. 字面值形式： JVM会自动根据字符串常量池中字符串的实际情况来决定是否创建新对象(要么不创建，要么创建一个对象，关键要看常量池中有没有)

```
String s = "abc";
```

等价于

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

该种方式先在栈中创建一个对String类的对象引用变量s，然后去查找 “abc”是否被保存在字符串常量池中。若”abc”已经被保存在字符串常量池中，则在字符串常量池中找到值为”abc”的对象，然后将s 指向这个对象; 否则，在 堆 中创建char数组 data，然后在 堆 中创建一个String对象object，它由 data 数组支持，紧接着这个String对象 object 被存放进字符串常量池，最后将 s 指向这个对象。

例如

```
    private static void test01(){
    String s0 = "kvill";        // 1
    String s1 = "kvill";        // 2
    String s2 = "kv" + "ill";     // 3

    System.out.println(s0 == s1);       // true
    System.out.println(s0 == s2);       // true
}
```

执行第 1 行代码时，“kvill” 入池并被 s0 指向；执行第 2 行代码时，s1 从常量池查询到” kvill” 对象并直接指向它；所以，s0 和 s1 指向同一对象。 由于 ”kv” 和 ”ill” 都是字符串字面值，所以 s2 在编译期由编译器直接解析为 “kvill”，所以 s2 也是常量池中”kvill”的一个引用。 所以，我们得出 s0==s1==s2;

2. 通过 new 创建字符串对象 : 一概在堆中创建新对象，无论字符串字面值是否相等 (要么创建一个，要么创建两个对象，关键要看常量池中有没有)

```
String s = new String("abc");
```

等价于:

```
1、String original = "abc";
2、String s = new String(original);
```

所以，通过 new 操作产生一个字符串（“abc”）时，会先去常量池中查找是否有“abc”对象，如果没有，则创建一个此字符串对象并放入常量池中。然后，在堆中再创建“abc”对象，并返回该对象的地址。所以，对于 String str=new String(“abc”)：如果常量池中原来没有”abc”，则会产生两个对象（一个在常量池中，一个在堆中）；否则，产生一个对象。

用 new String() 创建的字符串对象位于堆中，而不是常量池中。它们有自己独立的地址空间，例如，

```
    private static void test02(){
    String s0 = "kvill";
    String s1 = new String("kvill");
    String s2 = "kv" + new String("ill");

    String s = "ill";
    String s3 = "kv" + s;


    System.out.println(s0 == s1);       // false
    System.out.println(s0 == s2);       // false
    System.out.println(s1 == s2);       // false
    System.out.println(s0 == s3);       // false
    System.out.println(s1 == s3);       // false
    System.out.println(s2 == s3);       // false
}
```

例子中，s0 还是常量池中”kvill”的引用，s1 指向运行时创建的新对象”kvill”，二者指向不同的对象。对于s2，因为后半部分是 new String(“ill”)，所以无法在编译期确定，在运行期会 new 一个 StringBuilder 对象， 并由 StringBuilder 的 append 方法连接并调用其 toString 方法返回一个新的 “kvill” 对象。此外，s3 的情形与 s2 一样，均含有编译期无法确定的元素。因此，以上四个 “kvill” 对象互不相同。StringBuilder 的 toString 为:

```
public String toString() {
    return new String(value, 0, count);    // new 的方式创建字符串
}
```

构造函数 String(String original) 的源码为：

```
    /**
     * 根据源字符串的底层数组长度与该字符串本身长度是否相等决定是否共用支撑数组
     */
    public String(String original) {
        int size = original.count;
        char[] originalValue = original.value;
        char[] v;
        if (originalValue.length > size) {
            // The array representing the String is bigger than the new
            // String itself.  Perhaps this constructor is being called
            // in order to trim the baggage, so make a copy of the array.
            int off = original.offset;
            v = Arrays.copyOfRange(originalValue, off, off + size);  // 创建
新数组并赋给 v
        } else {
            // The array representing the String is the same
            // size as the String, so no point in making a copy.
            v = originalValue;
        }

        this.offset = 0;
        this.count = size;
        this.value = v;
    }
```

由源码可以知道，**所创建的对象在大多数情形下会与源字符串 original 共享 char数组 。**

# 字符串常量池

1. 字符串池

字符串的分配，和其他的对象分配一样，耗费高昂的时间与空间代价。JVM为了提高性能和减少内存开销，在实例化字符串字面值的时候进行了一些优化。为了减少在JVM中创建的字符串的数量，字符串类维护了一个字符串常量池，每当以字面值形式创建一个字符串时，JVM会首先检查字符串常量池：如果字符串已经存在池中，就返回池中的实例引用；如果字符串不在池中，就会实例化一个字符串并放到池中。Java能够进行这样的优化是因为字符串是不可 变的，可以不用担心数据冲突进行共享。 例如：

```
public class Program
{
    public static void main(String[] args)
    {
       String str1 = "Hello";
       String str2 = "Hello";
       System.out.print(str1 == str2);   // true
    }
}
```

一个初始为空的字符串池，它由类 String 私有地维护。当以字面值形式创建一个字符串时，总是先检查字符串池是否含存在该对象，若存在，则直接返回。此外，通过 new 操作符创建的字符串对象不指向字符串池中的任何对象，在堆里。

2. 手动入池

一个初始为空的字符串池，它由类 String 私有地维护。 当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并返回此 String 对象的引用。特别地，手动入池遵循以下规则：

对于任意两个字符串 s 和 t，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true。

```
public class TestString{
    public static void main(String args[]){
        String str1 = "abc";
        String str2 = new String("abc");
        String str3 = s2.intern();

        System.out.println( str1 == str2 );   //false
        System.out.println( str1 == str3 );   //true
    }
}
```

所以，对于 String str1 = “abc”，str1 引用的是 常量池（方法区） 的对象；而 String str2 = new String(“abc”)，str2引用的是 堆 中的对象，所以内存地址不一样。但是由于内容一样，所以 str1 和 str3 指向同一对象。

3. 实例

看下面几个场景来深入理解 String。

1. 情景一：字符串常量池

Java虚拟机(JVM)中存在着一个字符串常量池，其中保存着很多String对象，并且这些String对象可以被共享使用，因此提高了效率。之所以字符串具有字符串常量池，是因为String对象是不可变的，因此可以被共享。字符串常量池由String类维护，我们可以通过intern()方法使字符串池手动入池。

```
    String s1 = "abc";
    //↑ 在字符串池创建了一个对象
    String s2 = "abc";
    //↑ 字符串pool已经存在对象“abc”(共享),所以创建0个对象，累计创建一个对象
    System.out.println("s1 == s2 : "+(s1==s2));
    //↑ true 指向同一个对象，
    System.out.println("s1.equals(s2) : " + (s1.equals(s2)));
    //↑ true  值相等
```

2. 情景二：关于new String(“...”)

```
String s3 = new String("abc");
//↑ 创建了两个对象，一个存放在字符串池中，一个存在与堆区中；
//↑ 还有一个对象引用s3存放在栈中
String s4 = new String("abc");
//↑ 字符串池中已经存在“abc”对象，所以只在堆中创建了一个对象
System.out.println("s3 == s4 : "+(s3==s4));
//↑false   s3和s4栈区的地址不同，指向堆区的不同地址；
System.out.println("s3.equals(s4) : "+(s3.equals(s4)));
//↑true  s3和s4的值相同
System.out.println("s1 == s3 : "+(s1==s3));
//↑false 存放的地区都不同，一个方法区，一个堆区
System.out.println("s1.equals(s3) : "+(s1.equals(s3)));
//↑true  值相同
```

通过上一篇博文我们知道，通过 new String(“...”) 来创建字符串时，在该构造函数的参数值为字符串字面值的前提下，若该字面值不在字符串常量池中，那么会创建两个对象：一个在字符串常量池中，一个在堆中；否则，只会在堆中创建一个对象。对于不在同一区域的两个对象，二者的内存地址必定不同。

3. 情景三：字符串连接符“+”

```
String str2 = "ab";  //1个对象
String str3 = "cd";  //1个对象

String str4 = str2+str3;
String str5 = "abcd";
System.out.println("str4 = str5 : " + (str4==str5)); // false
```

我们看这个例子，局部变量 str2，str3 指向字符串常量池中的两个对象。在运行时，第三行代码(str2+str3)实质上会被分解成五个步骤，分别是：

1. 调用 String 类的静态方法 String.valueOf() 将 str2 转换为字符串表示；
2. JVM 在**堆中创建一个 StringBuilder对象**，同时用str2指向转换后的字符串对象进行初始化；
3. 调用StringBuilder对象的append方法完成与str3所指向的字符串对象的合并；
4. 调用 StringBuilder 的 toString() 方法在堆中创建一个 String对象；所以这里既有builder对象的abcd，也有普通的abcd。
5. 将刚刚生成的String对象的堆地址存赋给局部变量引用str4。

而引用str5指向的是字符串常量池中字面值”abcd”所对应的字符串对象。由上面的内容我们可以知道，引用str4和str5指向的对象的地址必定不一样。这时，内存中实际上会存在五个字符串对象： 三个在字符串常量池中的String对象、一个在堆中的String对象和一个**在堆中的StringBuilder对象。**

4. 情景四：字符串的编译期优化

```
String str1 = "ab" + "cd";  //1个对象
String str11 = "abcd";
System.out.println("str1 = str11 : "+ (str1 == str11));   // true

final String str8 = "cd";
String str9 = "ab" + str8;
String str89 = "abcd";
System.out.println("str9 = str89 : "+ (str9 == str89));     // true
//↑str8为常量变量，编译期会被优化
```

```
String str6 = "b";
String str7 = "a" + str6;
String str67 = "ab";
System.out.println("str7 = str67 : "+ (str7 == str67));     // false
//↑str6为变量，在运行期才会被解析。
```

Java 编译器对于类似“**常量+字面值**”的组合，其值在编译的时候就能够被确定了。在这里，str1 和 str9 的值**在编译时就可以被确定**，因此它们分别等价于： String str1 = “abcd”; 和 String str9 = “abcd”;

Java 编译器对于含有 “String引用”的组合，则在运行期会产生新的对象 (通过调用 StringBuilder类的toString()方法)，因此这个对象存储在堆中。

4. 小结

    1. 使用字面值形式创建的字符串与通过 new 创建的字符串一定是不同的，因为二者的存储位置不同：前者在方法区，后者在堆；
    2. 我们在使用诸如String str = “abc”；的格式创建字符串对象时，总是想当然地认为，我们创建了String类的对象str。但是事实上， 对象可能并没有被创建。唯一可以肯定的是，指向 String 对象 的引用被创建了。至于这个引用到底是否指向了一个新的对象，必须根据上下文来考虑；
    3. 字符串常量池的理念是 《享元模式》；
    4. Java 编译器对 “常量+字面值” 的组合 是当成常量表达式直接求值来优化的；对于**含有“String引用”的组合**，其在编译期不能被确定，会在运行期创建新对象。

# 三大字符串类： String、StringBuilder 和 StringBuffer

1. String 与 StringBuilder

    简要的说， String 类型 和 StringBuilder 类型的主要性能区别在于 String 是不可变的对象。 事实上，在对 String 类型进行“改变”时，实质上等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象。由于频繁的生成对象会对系统性能产生影响，特别是当内存中没有引用指向的对象多了以后，JVM 的垃圾回收器就会开始工作，继而会影响到程序的执行效率。所以，对于经常改变内容的字符串，最好不要声明为 String 类型。但如果我们使用的是 StringBuilder 类，那么情形就不一样了。因为，我们的每次修改都是针对 StringBuilder 对象本身的，而不会像对String操作那样去生成新的对象并重新给变量引用赋值。所以，在一般情况下，推荐使用 StringBuilder ，特别是字符串对象经常改变的情况下。

    在某些特别情况下，String 对象的字符串拼接可以直接被JVM 在编译期确定下来，这时，StringBuilder 在速度上就不占任何优势了。

    因此，在绝大部分情况下， 在效率方面：StringBuilder > String

2. StringBuffer 与 StringBuilder

    首先需要明确的是，StringBuffer 始于 JDK 1.0，而 StringBuilder 始于 JDK 5.0；此外，从 JDK 1.5 开始，对含有字符串变量 (非字符串字面值) 的连接操作(+)，JVM 内部是采用 StringBuilder 来实现的，而在这之前，这个操作是采用 StringBuffer 实现的。

    JDK的实现中 StringBuffer 与 StringBuilder 都继承自 AbstractStringBuilder。
    AbstractStringBuilder的实现原理为：AbstractStringBuilder中采用一个 char数组 来保存需要append的字符串，char数组有一个初始大小，当append的字符串长度超过当前char数组容量时，则对char数组进行动态扩展，即重新申请一段更大的内存空间，然后将当前char数组拷贝到新的位置，因为重新分配内存并拷贝的开销比较大，所以每次重新申请内存空间都是采用申请大于当前需要的内存空间的方式，这里是 2 倍。

StringBuffer 和 StringBuilder 都是可变的字符序列，但是二者最大的一个不同点是：StringBuffer 是线程安全的，而 StringBuilder 则不是。StringBuilder 提供的API与StringBuffer的API是完全兼容的，即，StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，但是后者一般要比前者快。因此，可以这么说，StringBuilder 的提出就是为了在单线程环境下替换 StringBuffer 。

在单线程环境下，优先使用 StringBuilder。

3. 实例

    1. 编译时优化与字符串连接符的本质

       我们先来看下面这个例子：

```
public class Test2 {
    public static void main(String[] args) {
        String s = "a" + "b" + "c";
        String s1 = "a";
        String s2 = "b";
        String s3 = "c";
        String s4 = s1 + s2 + s3;

        System.out.println(s);
        System.out.println(s4);
    }
}
```

由上面的叙述，我们可以知道，变量s的创建等价于 String s = “abc”; 而变量s4的创建相当于：

```
    StringBuilder temp = new StringBuilder(s1);
    temp.append(s2).append(s3);
    String s4 = temp.toString();
```

但事实上，是不是这样子呢？我们将其反编译一下，来看看Java编译器究竟做了什么：

```
//将上述 Test2 的 class 文件反编译
public class Test2
{
    public Test2(){}
    public static void main(String args[])
    {
        String s = "abc";             // 编译期优化
        String s1 = "a";
        String s2 = "b";
        String s3 = "c";

        //底层使用 StringBuilder 进行字符串的拼接
        String s4 = (new
StringBuilder(String.valueOf(s1))).append(s2).append(s3).toString();
        System.out.println(s);
        System.out.println(s4);
    }
}
```

根据上面的反编译结果，很好的印证了我们在第六节中提出的字符串连接符的本质。

    2. 另一个例子：字符串连接符的本质

由上面的分析结果，我们不难推断出 String 采用连接运算符（+）效率低下原因分析，形如这样的代码：

```
public class Test {
    public static void main(String args[]) {
        String s = null;
            for(int i = 0; i < 100; i++) {
                s += "a";
            }
    }
}
```

每做一次 字符串连接操作 “+” 就产生一个 StringBuilder 对象，然后 append 后就扔掉。下次循环再到达时，再重新 new 一个 StringBuilder 对象，然后 append 字符串，如此循环直至结束。事实上，如果我们直接采用 StringBuilder 对象进行 append 的话，我们可以节省 N - 1 次创建和销毁对象的时间。所以，对于在循环中要进行字符串连接的应用，一般都是用StringBulider对象来进行append操作。

## String 与 (深)克隆

待补充

# Java 并发

## Thread 类深度解析

Java 中 Thread类 的各种操作与线程的生命周期密不可分，了解线程的生命周期有助于对Thread类中的各方法的理解。一般来说，线程从最初的创建到最终的消亡，要经历创建、就绪、运行、阻塞 和 消亡五个状态。在线程的生命周期中，上下文切换通过存储和恢复CPU状态使得其能够从中断点恢复执行。结合 线程生命周期，本文最后详细介绍了 Thread 各常用 API。特别地，在介绍会导致线程进入Waiting状态(包括Timed Waiting状态)的相关API时，笔者会特别关注两个问题：

- 客户端调用该API后，是否会释放锁(如果此时拥有锁的话)；
- 客户端调用该API后，是否会交出CPU(一般情况下，线程进入Waiting状态(包括Timed Waiting状态)时都会交出CPU)；

### 线程的生命周期

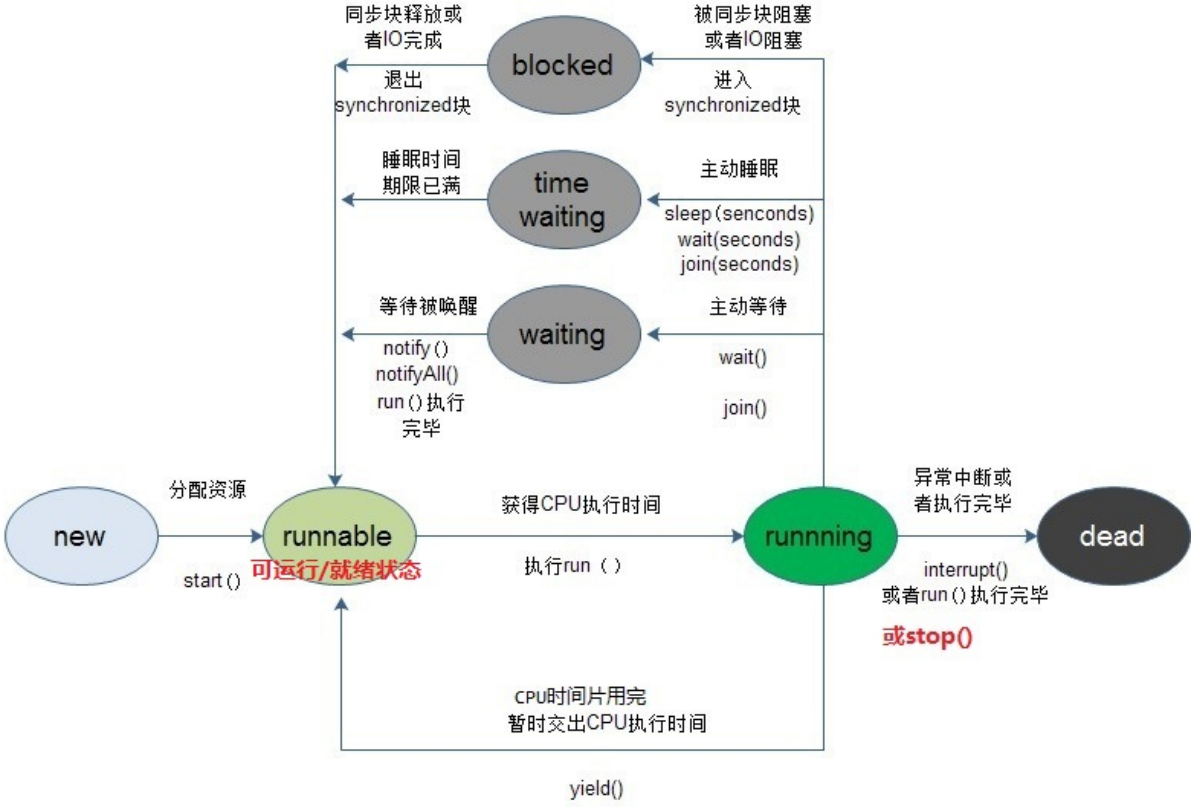
Java 中 Thread类 的具体操作与线程的生命周期密不可分，了解线程的生命周期有助于对Thread类中的各方法的理解。

在 Java虚拟机 中，线程从最初的创建到最终的消亡，要经历若干个状态：创建(new)、就绪(runnable/start)、运行(running)、阻塞(blocked)、等待(waiting)、时间等待(time waiting) 和 消亡(dead/terminated)。在给定的时间点上，一个线程只能处于一种状态，各状态的含义如下所示：

- New：还没启动
- Runnable：可运行
- Block：被阻塞
- Waiting：等待其他线程做动作，不确定等待时间
- Timed Waiting：等待其他线程做动作，确定的等待时间
- Terminated：线程消亡

当我们需要线程来执行某个子任务时，就必须先创建一个线程。但是线程创建之后，不会立即进入就绪状态，因为线程的运行需要一些条件（比如程序计数器、Java栈、本地方法栈等），只有线程运行需要的所有条件满足了，才进入就绪状态。当线程进入就绪状态后，不代表立刻就能获取CPU执行时间，也许此时CPU正在执行其他的事情，因此它要等待。当得到CPU执行时间之后，线程便真正进入运行状态。线程在运行状态过程中，可能有多个原因导致当前线程不继续运行下去，比如用户主动让线程睡眠（睡眠一定的时间之后再重新执行）、用户主动让线程等待，或者被同步块阻塞，此时就对应着多个状态：time waiting（睡眠或等待一定的时间）、waiting（等待被唤醒）、blocked（阻塞）。当由于突然中断或者子任务执行完毕，线程就会被消亡。

实际上，Java只定义了六种线程状态，分别是 New, Runnable, Waiting，Timed Waiting、Blocked 和 Terminated。为形象表达线程从创建到消亡之间的状态，下图将Runnable状态分成两种状态：正在运行状态和就绪状态：




## 上下文切换

以单核CPU为例，CPU在一个时刻只能运行一个线程。CPU在运行一个线程的过程中，转而去运行另外一个线程，这个叫做线程 上下文切换（对于进程也是类似）。

由于可能当前线程的任务并没有执行完毕，所以在切换时需要保存线程的运行状态，以便下次重新切换回来时能够紧接着之前的状态继续运行。举个简单的例子：比如，一个线程A正在读取一个文件的内容，正读到文件的一半，此时需要暂停线程A，转去执行线程B，当再次切换回来执行线程A的时候，我们不希望线程A又从文件的开头来读取。

因此需要记录线程A的运行状态，那么会记录哪些数据呢？因为下次恢复时需要知道在这之前当前线程已经执行到哪条指令了，所以需要记录程序计数器的值，另外比如说线程正在进行某个计算的时候被挂起了，那么下次继续执行的时候需要知道之前挂起时变量的值时多少，因此需要记录CPU寄存器的状态。所以，一般来说，线程上下文切换过程中会记录程序计数器、CPU寄存器状态等数据。

实质上， 线程的**上下文切换就是存储和恢复CPU状态的过程**，它使得线程执行能够从中断点恢复执行，这正是有程序计数器所支持的。

虽然多线程可以使得任务执行的效率得到提升，但是由于在线程切换时同样会带来一定的开销代价，并且多个线程会导致系统资源占用的增加，所以在进行多线程编程时要注意这些因素。

## 线程的创建

在 Java 中，创建线程去执行子任务一般有两种方式：继承 Thread 类和实现 Runnable 接口。其中，Thread 类本身就实现了 Runnable 接口，而使用继承 Thread 类的方式创建线程的最大局限就是不支持多继承。特别需要注意两点，

- 实现多线程必须重写run()方法，即在run()方法中定义需要执行的任务;
- run()方法不需要用户来调用。

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

        //使用继承Thread类的方式创建线程
        new Thread(){
            @Override
            public void run() {
                System.out.println("Thread");
            }
        }.start();

        //使用实现Runnable接口的方式创建线程
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Runnable");
            }
        });
        thread.start();

        //JVM 创建的主线程 main
        System.out.println("main");
    }
}/* Output: (代码的运行结果与代码的执行顺序或调用顺序无关)
        Thread
        main
        Runnable
 *///:~
```

创建好自己的线程类之后，就可以创建线程对象了，然后通过start()方法去启动线程。注意，run() 方法中只是定义需要执行的任务，并且其不需要用户来调用。当通过start()方法启动一个线程之后，若线程获得了CPU执行时间，便进入run()方法体去执行具体的任务。如果用户直接调用run()方法，即相当于在主线程中执行run()方法，跟普通的方法调用没有任何区别，此时并不会创建一个新的线程来执行定义的任务。实际上，start()方法的作用是通知 “线程规划器” 该线程已经准备就绪，以便让系统安排一个时间来调用其 run()方法，也就是使线程得到运行。

## Thread 类详解

Thread 类实现了 Runnable 接口，在 Thread 类中，有一些比较关键的属性，比如name是表示Thread的名字，可以通过Thread类的构造器中的参数来指定线程名字，priority表示线程的优先级（最大值为10，最小值为1，默认值为5），daemon表示线程是否是守护线程，target表示要执行的任务。

1. 与线程运行状态有关的方法
    1. start 方法

start() 用来启动一个线程，当调用该方法后，相应线程就会进入就绪状态，该线程中的run()方法会在某个时机被调用。

2. run 方法

run()方法是不需要用户来调用的。当通过start()方法启动一个线程之后，一旦线程获得了CPU执行时间，便进入run()方法体去执行具体的任务。注意，创建线程时必须重写run()方法，以定义具体要执行的任务。

一般来说，有两种方式可以达到重写run()方法的效果：

- **直接重写：**直接继承Thread类并重写run()方法；
- **间接重写：**通过Thread构造函数传入Runnable对象 (注意，实际上重写的是 Runnable 对象 的run() 方法)。

3. sleep 方法

方法 sleep() 的作用是在指定的毫秒数内让当前正在执行的线程（即 currentThread() 方法所返回的线程）睡眠，并交出 CPU 让其去执行其他的任务。当线程睡眠时间满后，不一定会立即得到执行，因为此时 CPU 可能正在执行其他的任务。所以说，调用sleep方法相当于让线程进入阻塞状态。该方法有如下两条特征：

- 如果调用了sleep方法，必须捕获InterruptedException异常或者将该异常向上层抛出；
- sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。

4. yield 方法

调用 yield()方法会让当前线程交出CPU资源，让CPU去执行其他的线程。但是，yield()不能控制具体的交出CPU的时间。需要注意的是，

- yield()方法只能让 **拥有相同优先级的线程** 有获取 CPU 执行时间的机会；
- 调用yield()方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新得到 CPU 的执行；
- 它同样不会释放锁。

```
public class MyThread extends Thread {

    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        int count = 0;
        for (int i = 0; i < 50000; i++) {
            Thread.yield();             // 将该语句注释后，执行会变快
            count = count + (i + 1);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("用时：" + (endTime - beginTime) + "毫秒！");
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

5. join 方法

假如在main线程中调用thread.join方法，则main线程会等待thread线程执行完毕或者等待一定的时间。详细地，如果调用的是无参join方法，则等待thread执行完毕；如果调用的是指定了时间参数的join方法，则等待一定的时间。join()方法有三个重载版本：

```
public final synchronized void join(long millis) throws
InterruptedException {...}
public final synchronized void join(long millis, int nanos) throws
InterruptedException {...}
public final void join() throws InterruptedException {...}
```

join()方法是通过wait()方法 (Object 提供的方法) 实现的。当 millis == 0 时，会进入 while(isAlive()) 循环，并且只要子线程是活的，宿主线程就不停的等待。 wait(0) 的作用是让当前线程(宿主线程)等待，而这里的当前线程是指 Thread.currentThread() 所返回的线程。所以，虽然是子线程对象(锁)调用wait()方法，但是阻塞的是宿主线程。

由于 join方法 会调用 wait方法 让宿主线程进入阻塞状态，并且会释放线程占有的锁，并交出CPU执行权限。结合 join 方法的声明，有以下三条：

- join方法同样会会让线程交出CPU执行权限；
- join方法同样会让线程释放对一个对象持有的锁；
- 如果调用了join方法，必须捕获InterruptedException异常或者将该异常向上层抛出。

  更多关于 join方法 的介绍见《Java 并发：线程间通信与协作》一文。

6. interrupt 方法

interrupt，顾名思义，即中断的意思。单独调用interrupt方法可以使得 **处于阻塞状态的线程抛出一个异常，也就是说，它可以用来中断一个正处于阻塞状态的线程**；另外，通过 interrupted()方法 和 isInterrupted()方法 可以停止正在运行的线程。

```
public class Test {

    public static void main(String[] args) throws IOException  {
        Test test = new Test();
        MyThread thread = test.new MyThread();
        thread.start();
        try {
            Thread.currentThread().sleep(2000);
        } catch (InterruptedException e) {

        }
        thread.interrupt();
    }

    class MyThread extends Thread{
        @Override
        public void run() {
            try {
                System.out.println("进入睡眠状态");
                Thread.currentThread().sleep(10000);
                System.out.println("睡眠完毕");
            } catch (InterruptedException e) {
                System.out.println("得到中断异常");
            }
            System.out.println("run方法执行完毕");
        }
    }
}/* Output:
        进入睡眠状态
        得到中断异常
        run方法执行完毕
```

```
 *///~
```

从这里可以看出，**通过interrupt方法可以中断处于阻塞状态的线程。**那么能不能中断处于非阻塞状态的线程呢？看下面这个例子：

```
public class Test {

    public static void main(String[] args) throws IOException  {
        Test test = new Test();
        MyThread thread = test.new MyThread();
        thread.start();
        try {
            Thread.currentThread().sleep(2000);
        } catch (InterruptedException e) {}
        thread.interrupt();
    }

    class MyThread extends Thread{
        @Override
        public void run() {
            int i = 0;
            while(i<Integer.MAX_VALUE){
                System.out.println(i+" while循环");
                i++;
            }
        }
    }
}
```

运行该程序会发现，while循环会一直运行直到变量i的值超出Integer.MAX_VALUE。所以说，直接调用interrupt() 方法不能中断正在运行中的线程。但是，如果配合isInterrupted()/interrupted() 能够中断正在运行的线程，因为调用interrupt()方法相当于将中断标志位置为true，那么可以通过调用isInterrupted()/interrupted()判断中断标志是否被置位来中断线程的执行。比如下面这段代码：

```
public class Test {
    public static void main(String[] args) throws IOException  {
        Test test = new Test();
        MyThread thread = test.new MyThread();
        thread.start();
        try {
            Thread.currentThread().sleep(2000);
        } catch (InterruptedException e) {

        }
        thread.interrupt();
    }

    class MyThread extends Thread{
        @Override
        public void run() {
            int i = 0;
            while(!isInterrupted() && i<Integer.MAX_VALUE){
                System.out.println(i+" while循环");
                i++;
            }
```

```
        }
    }
}
```

一般情况下，不建议通过这种方式来中断线程，一般会在MyThread类中增加一个 *volatile 属性* isStop 来标志是否结束 while 循环，然后再在 while 循环中判断 isStop 的值。

```
class MyThread extends Thread{
        private volatile boolean isStop = false;
        @Override
        public void run() {
            int i = 0;
            while(!isStop){
                i++;
            }
        }

        public void setStop(boolean stop){
            this.isStop = stop;
        }
    }
```

7. stop方法

   stop() 方法已经是一个 **废弃的** 方法，它是一个 **不安全的** 方法。因为调用 stop() 方法会直接终止run方法的调用，并且会抛出一个ThreadDeath错误，如果线程持有某个对象锁的话，会完全释放锁，导致对象状态不一致。所以， stop() 方法基本是不会被用到的。

2. 线程的暂停与恢复

   1. 线程的暂停、恢复方法在 JDK 中的定义

      暂停线程意味着此线程还可以恢复运行。在 Java 中，我可以使用 suspend() 方法暂停线程，使用 resume() 方法恢复线程的执行，但是这两个方法已被废弃，因为它们具有固有的死锁倾向。如果目标线程挂起时在保护关键系统资源的监视器上保持有锁，则在目标线程重新开始以前，任何线程都不能访问该资源。如果重新开始目标线程的线程想在调用 resume 之前锁定该监视器，则会发生死锁。

   2. 死锁

      **具体地，在使用 suspend 和 resume 方法时，如果使用不当，极易造成公共的同步对象的独占，使得其他线程无法得到公共同步对象锁，从而造成死锁。**下面举两个示例：

```
// 示例 1
public class SynchronizedObject {

    public synchronized void printString() {        // 同步方法
        System.out.println("Thread-" + Thread.currentThread().getName()
+ " begins.");
        if (Thread.currentThread().getName().equals("a")) {
            System.out.println("线程a suspend 了...");
            Thread.currentThread().suspend();
        }
        System.out.println("Thread-" + Thread.currentThread().getName()
+ " is end.");
    }
```

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

        final SynchronizedObject object = new SynchronizedObject();
// 两个线程使用共享同一个对象

        Thread a = new Thread("a") {
            @Override
            public void run() {
                object.printString();
            }
        };
        a.start();

        new Thread("b") {
            @Override
            public void run() {
                System.out.println("thread2 启动了，在等待中(发生“死
锁”)...");
                object.printString();
            }
        }.start();

        System.out.println("main 线程睡眠 " + 5 +" 秒...");
        Thread.sleep(5000);
        System.out.println("main 线程睡醒了...");

        a.resume();
        System.out.println("线程 a resume 了...");
    }
}/* Output:
        Thread-a begins.
        线程a suspend 了...
        thread2 启动了，在等待中(发生死锁)...
        main 线程睡眠 5 秒...
        main 线程睡醒了...
        线程 a resume 了...
        Thread-a is end.
        Thread-b begins.
        Thread-b is end.
 *///:~
```

在示例 2 中，特别要注意的是，**println() 方法实质上是一个同步方法。**如果 thread 线程刚好在执行打印语句时被挂起，那么将会导致 main线程中的字符串 “main end!” 迟迟不能打印。

```
// 示例 2
public class MyThread extends Thread {

    private long i = 0;

    @Override
    public void run() {
        while (true) {
            i++;
            System.out.println(i);
        }
    }
```

```
    public static void main(String[] args) {
        try {
            MyThread thread = new MyThread();
            thread.start();
            Thread.sleep(1);
            thread.suspend();
            System.out.println("main end!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3. 线程常用操作

    1. 获得代码调用者信息

    currentThread() 方法返回代码段正在被哪个线程调用的信息。

```
public class CountOperate extends Thread {

    public CountOperate() {
        super("Thread-CO");     // 线程 CountOperate 的名字
        System.out.println("CountOperate---begin");
        System.out.println("Thread.currentThread().getName()="
                + Thread.currentThread().getName());
        System.out.println("this.getName()=" + this.getName());
        System.out.println("CountOperate---end");
    }

    @Override
    public void run() {
        System.out.println("run---begin");
        System.out.println("Thread.currentThread().getName()="
                + Thread.currentThread().getName());
        System.out.println("this.getName()=" + this.getName());
        System.out.println("run---end");
    }

    public static void main(String[] args) {
        CountOperate c = new CountOperate();
        Thread t1 = new Thread(c);
        t1.setName("A");
        t1.start();
        c.start();
    }
}/* Output:(输出结果不唯一)
        CountOperate---begin                                ....... 行 1
        Thread.currentThread().getName()=main               ....... 行 2
        this.getName()=Thread-CO                            ....... 行 3
        CountOperate---end                                  ....... 行 4
        run---begin                                         ....... 行 5
        Thread.currentThread().getName()=A                  ....... 行 6
        run---begin                                         ........行 7
        Thread.currentThread().getName()=Thread-CO          ....... 行 8
        this.getName()=Thread-CO                            ....... 行 9
        run---end                                           ....... 行 10
```

```
        this.getName()=Thread-CO                                ....... 行 11
        run---end                                               ....... 行 12
*///:~
```

首先来看前四行的输出。我们知道 CountOperate 继承了 Thread 类，那么 CountOperate 就得到了 Thread类的所有非私有属性和方法。CountOperate 构造方法中的 super("Thread-CO");意味着调用了父类Thread的构造器Thread(String name)，也就是为 CountOperate线程 赋了标识名。由于该构造方法是由main()方法调用的，因此此时 Thread.currentThread() 返回的是main线程；而 this.getName() 返回的是CountOperate线程的标识名。

其次，在main线程启动了t1线程之后，CPU会在某个时机执行类CountOperate的run()方法。此时，Thread.currentThread() 返回的是t1线程，因为是t1线程的启动使run()方法得到了执行；而 this.getName() 返回的仍是CountOperate线程的标识名，因为此时this指的是传进来的CountOperate对象(run执行的是target.run()，target是Runnable类型的成员对象，通过Thread构造函数赋值、实例化)，由于它本身也是一个线程对象，所以可以调用getName()得到相应的标识名。

在main线程启动了CountOperate线程之后，CPU也会在某个时机执行类该线程的run()方法。此时，Thread.currentThread() 返回的是CountOperate线程，因为是CountOperate线程的启动使run()方法得到了执行；而 this.getName() 返回的仍是CountOperate线程的标识名，因为此时this指的就是刚刚创建的CountOperate对象本身，所以得到的仍是 "Thread-CO"。

4. 判断线程是否处于活动状态

   **方法 isAlive() 的功能是判断调用该方法的线程是否处于活动状态。**其中，活动状态指的是线程已经 start (无论是否获得CPU资源并运行) 且尚未结束。

5. 获取线程唯一标识

   方法 getId() 的作用是取得线程唯一标识，由JVM自动给出。

6. 线程优先级的继承性

   在 Java 中，线程的优先级具有继承性，比如 A 线程启动 B 线程， 那么 B 线程的优先级与 A 是一样的。

7. 线程优先级的规则性和随机性

   **线程的优先级具有一定的规则性，也就是CPU尽量将执行资源让给优先级比较高的线程。**特别地，高优先级的线程总是大部分先执行完，但并不一定所有的高优先级线程都能先执行完。

8. 守护线程 (Daemon)

   在 Java 中，线程可以分为两种类型，即用户线程和守护线程。守护线程是一种特殊的线程，具有"陪伴"的含义：当进程中不存在非守护线程时，则守护线程自动销毁，典型的守护线程就是垃圾回收线程。任何一个守护线程都是整个JVM中所有非守护线程的保姆，只要当前JVM实例中存在任何一个非守护线程没有结束，守护线程就在工作；只有当最后一个非守护线程结束时，守护线程才随着JVM一同结束工作。

9. 线程的各项基本操作

   对于上述线程的各项基本操作，其 **所操作的对象** 满足：

   - 若该操作是静态方法，也就是说，该方法属于类而非具体的某个对象，那么该操作的作用对象就是 currentThread() 方法所返回 Thread 对象；
   - 若该操作是实例方法，也就是说，该方法属于对象，那么该操作的作用对象就是调用该方法的 Thread 对象。

   对于上述线程的各项基本操作，有：

   - 线程一旦被阻塞，就会释放 CPU；

- 当线程出现异常且没有捕获处理时，JVM会自动释放当前线程占用的锁，因此不会由于异常导致出现死锁现象。
- 对于一个线程，CPU 的释放 与 锁的释放没有必然联系。

# 线程间通信和协作

线程与线程之间不是相互独立的个体，它们彼此之间需要相互通信和协作，最典型的例子就是生产者-消费者问题。本文首先介绍 wait/notify 机制，并对实现该机制的两种方式——synchronized+wait-notify模式和Lock+Condition模式进行详细剖析，以作为线程间通信与协作的基础。进一步地，以经典的生产者-消费者问题为背景，熟练对 wait/notify 机制的使用。最后，对 Thread 类中的 join() 方法进行源码分析，并以宿主线程与寄生线程的协作为例进行说明。

线程与线程之间不是相互独立的个体，它们彼此之间需要相互通信和协作。比如说最经典的 生产者-消费者模型：当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。因为生产者如果不释放对临界资源的占用权，那么消费者就无法消费队列中的商品，就不会让队列有空间，那么生产者就会一直无限等待下去。因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。然后等待消费者消费了商品，然后消费者通知生产者队列有空间了。同样地，当队列空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作，也是本文要探讨的问题。

在下面的例子中，虽然两个线程实现了通信，但是凭借 线程B 不断地通过 while语句轮询 来检测某一个条件，这样会导致CPU的浪费。因此，需要一种机制来减少 CPU资源 的浪费，而且还能实现多个线程之间的通信，即 wait/notify 机制 。

```
//资源类
class MyList {

    //临界资源
    private volatile List<String> list = new ArrayList<String>();

    public void add() {
        list.add("abc");
    }

    public int size() {
        return list.size();
    }
}

// 线程A
class ThreadA extends Thread {

    private MyList list;

    public ThreadA(MyList list,String name) {
        super(name);
        this.list = list;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                list.add();
                System.out.println("添加了" + (i + 1) + "个元素");
                Thread.sleep(1000);
```

```
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//线程B
class ThreadB extends Thread {

    private MyList list;

    public ThreadB(MyList list,String name) {
        super(name);
        this.list = list;
    }

    @Override
    public void run() {
        try {
            while (true) {          // while 语句轮询
                if (list.size() == 2) {
                    System.out.println("==2了，线程b要退出了！");
                    throw new InterruptedException();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//测试
public class Test {
    public static void main(String[] args) {

        MyList service = new MyList();

        ThreadA a = new ThreadA(service,"A");
        ThreadB b = new ThreadB(service,"B");

        a.start();
        b.start();
    }
}/* Output(输出结果不唯一):
        添加了1个元素
        添加了2个元素
        ==2了，线程b要退出了!
        java.lang.InterruptedException
            at test.ThreadB.run(Test.java:57)
        添加了3个元素
 *///:~
```

# wait/notify 机制

在这之前，线程间通过共享数据来实现通信，即多个线程主动地读取一个共享数据，通过 **同步互斥访问机制** 保证线程的安全性。**等待/通知机制** 主要由 Object类 中的以下三个方法保证：

wait()、notify() 和 notifyAll()

上述三个方法均非Thread类中所声明的方法，而是Object类中声明的方法。原因是**每个对象都拥有monitor（锁）**，所以让当前线程等待某个对象的锁，当然应该通过这个对象来操作，而不是用当前线程来操作，因为当前线程可能会等待多个线程的锁，如果通过线程来操作，就非常复杂了。

1. wait

   让 *当前线程* *(Thread.concurrentThread() 方法所返回的线程)* 释放对象锁并进入等待（阻塞）状态。

   该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。在调用 wait()之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用 wait()方法。进入 wait()方法后，当前线程释放锁。在从 wait()返回前，线程与其他线程竞争重新获得锁。如果调用 wait()时，没有持有适当的锁，则抛出 IllegalMonitorStateException，它是 RuntimeException 的一个子类，因此，不需要 try-catch 结构。

   1. 方法作用

      Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on this object's monitor by calling one of the wait() methods.

      This method causes the current thread (call it T):

      ① to place itself in the wait set for this object;

      ② to relinquish (放弃) any and all synchronization claims on this object;

      ③ Thread T becomes disabled for thread scheduling purposes and lies dormant (休眠) until one of four things happens:

      ```
      Some other thread invokes the notify method for this object and thread T
      happens to be arbitrarily chosen as the thread to be awakened;

      Some other thread invokes the notifyAll method for this object;

      Some other thread interrupts thread T;

      The specified amount of real time has elapsed, more or less. If timeout
      is zero, however, then real time is not taken into consideration and the
      thread simply waits until notified (等待时间为 0 意味着永远等待，直至线程被唤醒) .
      ```

      The thread T is then removed from the wait set for this object and re-enabled for thread scheduling. It then competes in the usual manner with other threads for the right to synchronize on the object; once it has gained control of the object, all its synchronization claims on the object are restored to the status quo ante - that is, to the situation as of the time that the wait method was invoked. Thread T then returns from the invocation of the wait method. Thus, on return from the wait method, the synchronization state of the object and of thread T is exactly as it was when the wait method was invoked.

2. 方法使用条件

   This method should only be called by a thread that is the owner of this object's monitor.

3. 异常

   - 运行时（不受检查）异常
     IllegalMonitorStateException： if the current thread is not the owner of this object's monitor；
     IllegalArgumentException： if the value of timeout is negative；
   - 受检查异常 (中断阻塞线程，抛 InterruptedException并终止线程，释放锁，释放CPU)
     InterruptedException： if any thread interrupted the current thread before or while the current thread was waiting for a notification. The interrupted status of the current thread is cleared when this exception is thrown.

2. notify

   唤醒一个正在等待相应对象锁的线程，使其进入就绪队列，以便在当前线程释放锁后竞争锁，进而得到CPU的执行。

   该方法也要在同步方法或同步块中调用，即在调用前，线程也必须要获得该对象的对象级别锁，的如果调用 notify()时没有持有适当的锁，也会抛出 IllegalMonitorStateException。

   该方法用来通知那些可能等待该对象的对象锁的其他线程。如果有多个线程等待，则线程规划器任意挑选出其中一个 wait()状态的线程来发出通知，并使它等待获取该对象的对象锁（notify 后，当前线程不会马上释放该对象锁，wait 所在的线程并不能马上获取该对象锁，要等到程序退出 synchronized 代码块后，当前线程才会释放锁，wait所在的线程也才可以获取该对象锁），但不惊动其他同样在等待被该对象notify的线程们。当第一个获得了该对象锁的 wait 线程运行完毕以后，它会释放掉该对象锁，此时如果该对象没有再次使用 notify 语句，则即便该对象已经空闲，其他 wait 状态等待的线程由于没有得到该对象的通知，会继续阻塞在 wait 状态，直到这个对象发出一个 notify 或 notifyAll。这里需要注意：它们等待的是被 notify 或 notifyAll，而不是锁。这与下面的 notifyAll()方法执行后的情况不同。

   1. 方法作用

      Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

      The awakened thread will not be able to proceed until the current thread relinquishes (放弃) the lock on this object.（在执行 notify() 方法后，当前线程不会马上释放该锁对象，呈 wait 状态的线程也并不能马上获取该对象锁。只有等到执行notify()方法的线程退出synchronized代码块/方法后，当前线程才会释放锁，而相应的呈wait状态的线程才可以去争取该对象锁。） The awakened thread will compete in the usual manner with any other threads that might be actively competing to (竞争) synchronize on this object; the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

   2. 方法使用条件

      This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

      ```
      By executing a synchronized instance method of that object;

      By executing the body of a synchronized statement that synchronizes on
      the object;

      For objects of type Class, by executing a synchronized static method of
      that class.
      ```

Only one thread at a time can own an object's monitor(互斥锁).

3. 异常

   运行时（不受检查）异常
   IllegalMonitorStateException： if the current thread is not the owner of this object's monitor.

3. notifyAll

   唤醒所有正在等待相应对象锁的线程，使它们进入就绪队列，以便在当前线程释放锁后竞争锁，进而得到CPU的执行。

   该方法与 notify ()方法的工作方式相同，重要的一点差异是：

   notifyAll 使所有原来在该对象上 wait 的线程统统退出 wait 的状态（即全部被唤醒，不再等待 notify 或 notifyAll，但由于此时还没有获取到该对象锁，因此还不能继续往下执行），变成等待获取该对象上的锁，一旦该对象锁被释放（notifyAll 线程退出调用了 notifyAll 的 synchronized 代码块的时候），他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出 synchronized 代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

   1. 方法作用

      Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

      The awakened threads will not be able to proceed until the current thread relinquishes (放弃) the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to (竞争) synchronize on this object; the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

   2. 方法使用条件

      This method should only be called by a thread that is the owner of this object's monitor.

   3. 异常

      - 运行时（不受检查）异常
        IllegalMonitorStateException： if the current thread is not the owner of this object's monitor.

4. 小结：

   如果线程调用了对象的 wait()方法，那么线程便会处于该对象的**等待池**中，等待池中的线程不会去竞争该对象的锁。

   当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的**锁池**中，锁池中的线程会去竞争该对象锁。

   优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

- wait()、notify() 和 notifyAll()方法是 本地方法，并且为 final 方法，无法被重写；
- 调用某个对象的 wait() 方法能让 当前线程阻塞，并且当前线程必须拥有此对象的monitor（即锁）；
- 调用某个对象的 notify() 方法能够唤醒 一个正在等待这个对象的monitor的线程，如果有多个线程都在等待这个对象的monitor，则只能唤醒其中一个线程；

- 调用notifyAll()方法能够唤醒所有正在等待这个对象的monitor的线程。

  综上，所谓唤醒线程，另一种解释可以说是将线程由等待池移动到锁池，notifyAll调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而notify只会唤醒一个线程。

## 方法调用与线程状态关系

**每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列。**就绪队列存储了已就绪（将要竞争锁）的线程，阻塞队列存储了被阻塞的线程。当一个阻塞线程被唤醒后，才会进入就绪队列，进而等待CPU的调度；反之，当一个线程被wait后，就会进入阻塞队列，等待被唤醒。

```
public class Test {
    public static Object object = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();

        thread1.start();

        Thread.sleep(2000);

        thread2.start();
    }

    static class Thread1 extends Thread {
        @Override
        public void run() {
            synchronized (object) {
                System.out.println("线程" + Thread.currentThread().getName()
                        + "获取到了锁...");
                try {
                    System.out.println("线程" + Thread.currentThread().getName()
                            + "阻塞并释放锁...");
                    object.wait();
                } catch (InterruptedException e) {
                }
                System.out.println("线程" + Thread.currentThread().getName()
                        + "执行完成...");
            }
        }
    }

    static class Thread2 extends Thread {
        @Override
        public void run() {
            synchronized (object) {
                System.out.println("线程" + Thread.currentThread().getName()
                        + "获取到了锁...");
                object.notify();
                System.out.println("线程" + Thread.currentThread().getName()
                        + "唤醒了正在wait的线程...");
            }
            System.out
                    .println("线程" + Thread.currentThread().getName() + "执行完
成...");
```

```
        }
    }
}/* Output:
        线程Thread-0获取到了锁...
        线程Thread-0阻塞并释放锁...
        线程Thread-1获取到了锁...
        线程Thread-1唤醒了正在wait的线程...
        线程Thread-1执行完成...
        线程Thread-0执行完成...
 *///:~
```

多个同类型线程的场景（wait 的条件发生变化）

```
//资源类
class ValueObject {
    public static List<String> list = new ArrayList<String>();
}

//元素添加线程
class ThreadAdd extends Thread {

    private String lock;

    public ThreadAdd(String lock,String name) {
        super(name);
        this.lock = lock;
    }

    @Override
    public void run() {
        synchronized (lock) {
            ValueObject.list.add("anyString");
            lock.notifyAll();               // 唤醒所有 wait 线程
        }
    }
}

//元素删除线程
class ThreadSubtract extends Thread {

    private String lock;

    public ThreadSubtract(String lock,String name) {
        super(name);
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            synchronized (lock) {
                if (ValueObject.list.size() == 0) {
                    System.out.println("wait begin ThreadName=" +
Thread.currentThread().getName());
                    lock.wait();
```

```
                    System.out.println("wait   end ThreadName=" +
Thread.currentThread().getName());
                }
                ValueObject.list.remove(0);
                System.out.println("list size=" + ValueObject.list.size());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//测试类
public class Run {
    public static void main(String[] args) throws InterruptedException {

        //锁对象
        String lock = new String("");

        ThreadSubtract subtract1Thread = new
ThreadSubtract(lock,"subtract1Thread");
        subtract1Thread.start();

        ThreadSubtract subtract2Thread = new
ThreadSubtract(lock,"subtract2Thread");
        subtract2Thread.start();

        Thread.sleep(1000);

        ThreadAdd addThread = new ThreadAdd(lock,"addThread");
        addThread.start();

    }
}/* Output:
        wait begin ThreadName=subtract1Thread
        wait begin ThreadName=subtract2Thread
        wait   end ThreadName=subtract2Thread
        list size=0
        wait   end ThreadName=subtract1Thread
        Exception in thread "subtract1Thread"
            java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
            at java.util.ArrayList.rangeCheck(Unknown Source)
            at java.util.ArrayList.remove(Unknown Source)
            at test.ThreadSubtract.run(Run.java:49)
 *///:~
```

当线程subtract1Thread 被唤醒后，将从 wait处 继续执行。但由于 线程subtract2Thread 先获取到锁得到运行，导致 线程subtract1Thread 的 wait的条件发生变化（不再满足），而 线程subtract1Thread 却毫无所知，导致异常产生。

像这种有多个相同类型的线程场景，为防止wait的条件发生变化而导致的线程异常终止，我们在阻塞线程被唤醒的同时还必须对wait的条件进行额外的检查，如下所示：

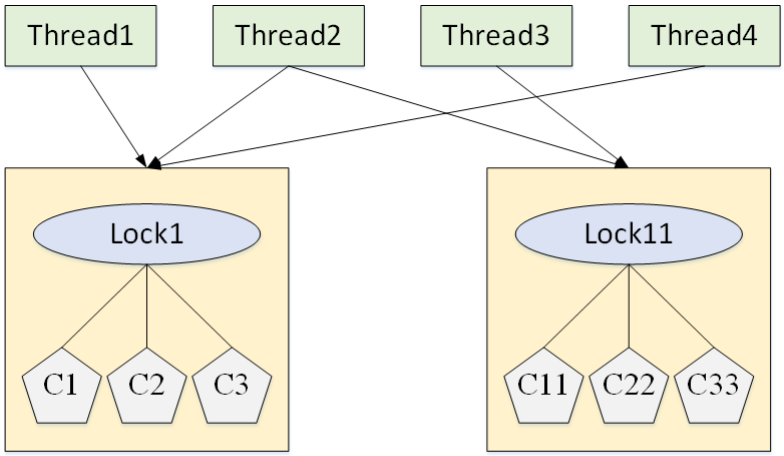
```
//元素删除线程
class ThreadSubtract extends Thread {
```

```
    private String lock;

    public ThreadSubtract(String lock,String name) {
        super(name);
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            synchronized (lock) {
                while (ValueObject.list.size() == 0) {    //将 if 改成 while
                    System.out.println("wait begin ThreadName=" +
Thread.currentThread().getName());
                    lock.wait();
                    System.out.println("wait   end ThreadName=" +
Thread.currentThread().getName());
                }
                ValueObject.list.remove(0);
                System.out.println("list size=" + ValueObject.list.size());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}/* Output:
        wait begin ThreadName=subtract1Thread
        wait begin ThreadName=subtract2Thread
        wait   end ThreadName=subtract2Thread
        list size=0
        wait   end ThreadName=subtract1Thread
        wait begin ThreadName=subtract1Thread
 *///:~
```

只需将 线程类ThreadSubtract 的 run()方法中的 if 条件 改为 while 条件 即可。

## Condition

Condition是在java 1.5中出现的，它用来替代传统的Object的wait()/notify()实现线程间的协作，它的使用依赖于 Lock，Condition、Lock 和 Thread 三者之间的关系如下图所示。相比使用Object的wait()/notify()，使用Condition的await()/signal()这种方式能够更加安全和高效地实现线程间协作。Condition是个接口，基本的方法就是await()和signal()方法。Condition依赖于Lock接口，生成一个Condition的基本代码是lock.newCondition() 。 必须要注意的是，Condition 的 await()/signal() 使用都必须在lock保护之内，也就是说，必须在lock.lock()和lock.unlock之间才可以使用。事实上，Conditon的await()/signal() 与 Object的wait()/notify() 有着天然的对应关系：




多个线程(Thread)可以竞争同一把锁，一把锁也可以关联多个Condition，以便多个线程间进行通信和协同。



使用Condition往往比使用传统的通知等待机制(Object的wait()/notify())要更灵活、高效，例如，我们可以使用多个Condition实现**通知部分线程**：

```
// 线程 A
class ThreadA extends Thread {
    private MyService service;
    public ThreadA(MyService service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.awaitA();
    }
}
// 线程 B
class ThreadB extends Thread {
    private MyService service;
    public ThreadB(MyService service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.awaitB();
    }
}

class MyService {
    private Lock lock = new ReentrantLock();
    // 使用多个Condition实现通知部分线程
    public Condition conditionA = lock.newCondition();
    public Condition conditionB = lock.newCondition();

    public void awaitA() {
        lock.lock();
        try {
            System.out.println("begin awaitA时间为" + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
```

```
            conditionA.await();
            System.out.println("  end awaitA时间为" + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void awaitB() {
        lock.lock();
        try {
            System.out.println("begin awaitB时间为" + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionB.await();
            System.out.println("  end awaitB时间为" + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void signalAll_A() {
        try {
            lock.lock();
            System.out.println("  signalAll_A时间为" + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionA.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public void signalAll_B() {
        try {
            lock.lock();
            System.out.println("  signalAll_B时间为" + System.currentTimeMillis()
                    + " ThreadName=" + Thread.currentThread().getName());
            conditionB.signalAll();
        } finally {
            lock.unlock();
        }
    }
}

// 测试
public class Run {
    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();

        ThreadA a = new ThreadA(service);
        a.setName("A");
        a.start();

        ThreadB b = new ThreadB(service);
```

```
        b.setName("B");
        b.start();

        Thread.sleep(3000);
        service.signalAll_A();
    }
}
```

实际上，**Condition 实现了一种分组机制，将所有对临界资源进行访问的线程进行分组，以便实现线程间更精细化的协作，例如通知部分线程。**我们可以从上面例子的输出结果看出，只有conditionA范围内的线程A被唤醒，而conditionB范围内的线程B仍然阻塞。

## 生产者-消费者模型

两种实现例子

## 线程间的通信：管道

PipedInputStream类 与 PipedOutputStream类 用于在应用程序中创建管道通信。一个PipedInputStream实例对象必须和一个PipedOutputStream实例对象进行连接而产生一个通信管道。PipedOutputStream可以向管道中写入数据，PipedIntputStream可以读取PipedOutputStream向管道中写入的数据，这两个类主要用来完成线程之间的通信。一个线程的PipedInputStream对象能够从另外一个线程的PipedOutputStream对象中读取数据。

PipedInputStream和PipedOutputStream的实现原理类似于"生产者-消费者"原理，PipedOutputStream是生产者，PipedInputStream是消费者。在PipedInputStream中，有一个buffer字节数组，默认大小为1024，作为缓冲区，存放"生产者"生产出来的东西。此外，还有两个变量in和out——in用来记录"生产者"生产了多少，out是用来记录"消费者"消费了多少，in为-1表示消费完了，in==out表示生产满了。当消费者没东西可消费的时候，也就是当in为-1的时候，消费者会一直等待，直到有东西可消费。

## 方法 join() 的使用

1. join() 的定义

   **假如在main线程中调用thread.join方法，则main线程会等待thread线程执行完毕或者等待一定的时间。**详细地，如果调用的是无参join方法，则等待thread执行完毕；如果调用的是指定了时间参数的join方法，则等待一定的时间。join()方法有三个重载版本：

   ```
   public final synchronized void join(long millis) throws InterruptedException
   {...}
   public final synchronized void join(long millis, int nanos) throws
   InterruptedException {...}
   public final void join() throws InterruptedException {...}
   ```

   以 join(long millis) 方法为例，其内部调用了Object的wait()方法，如下图：

```
public final synchronized void join(long millis)
        throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);  // 等待时间为 0 意味着永远等待，直至线程被唤醒
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

根据以上源代码可以看出，join()方法是通过wait()方法 (Object 提供的方法) 实现的。当 millis == 0 时，会进入 while(isAlive()) 循环，并且只要子线程是活的，宿主线程就不停的等待。 wait(0) 的作用是让当前线程(宿主线程)等待，而这里的当前线程是指 Thread.currentThread() 所返回的线程。所以，虽然是子线程对象(锁)调用wait()方法，但是阻塞的是宿主线程。

比如，在A里面写了b.wait()，则并非b陷入等待，而是持有b的A进入b的等待池，等待有线程调用b的notify唤醒。在join里面，因为thread本身就是一个对象，所以每个thread也都有自己的wait方法。所以主线程可以调用子线程的join来让自己进入子线程的等待池，然后子线程执行完或者到时间后，子线程在jvm里调用notifyAll唤醒主线程。

2. join() 使用实例及原理

```
//示例代码
public class Test {

    public static void main(String[] args) throws IOException  {
        System.out.println("进入线程"+Thread.currentThread().getName());
        Test test = new Test();
        MyThread thread1 = test.new MyThread();
        thread1.start();
        try {
            System.out.println("线程"+Thread.currentThread().getName()+"等
待");
            thread1.join();
            System.out.println("线程"+Thread.currentThread().getName()+"继续执
行");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    class MyThread extends Thread{
        @Override
        public void run() {
            System.out.println("进入线程"+Thread.currentThread().getName());
            try {
                Thread.currentThread().sleep(5000);
```

```
            } catch (InterruptedException e) {
                // TODO: handle exception
            }
            System.out.println("线程"+Thread.currentThread().getName()+"执行完
毕");
        }
    }
}/* Output:
        进入线程main
        线程main等待
        进入线程Thread-0
        线程Thread-0执行完毕
        线程main继续执行
 *///~
```

由于 join方法 会调用 wait方法 让宿主线程进入阻塞状态，并且会释放线程占有的锁，并交出CPU执行权限。结合 join 方法的声明，可以总结出以下三条：

> join方法同样会会让线程交出CPU执行权限；
>
> join方法同样会让线程释放对一个对象持有的锁；
>
> 如果调用了join方法，必须捕获InterruptedException异常或者将该异常向上层抛出。

## Lock

https://blog.csdn.net/justloveyou_/article/details/54972105

# Java线程池

## 什么是线程池

线程池其实就是一种多线程处理形式，处理过程中可以将任务添加到队列中，然后在创建线程后自动启动这些任务。这里的线程就是我们前面学过的线程,这里的任务就是我们前面学过的实现了Runnable或Callable接口的实例对象;

## 为什么使用线程池

使用线程池最大的原因就是可以根据系统的需求和硬件环境灵活的控制线程的数量,且可以对所有线程进行统一的管理和控制,从而提高系统的运行效率,降低系统运行运行压力;当然了,使用线程池的原因不仅仅只有这些,我们可以从线程池自身的优点上来进一步了解线程池的好处;

## 使用线程池有哪些优势

1. 线程和任务分离,提升线程重用性;
2. 控制线程并发数量,降低服务器压力,统一管理所有线程;
3. 提升系统响应速度,假如创建线程用的时间为T1，执行任务用的时间为T2,销毁线程用的时间为T3，那么使用线程池就免去了T1和T3的时间;

# Java内置线程池原理剖析

## ThreadPoolExecutor部分源码

```
构造方法:
public ThreadPoolExecutor(int corePoolSize, //核心线程数量
                          int maximumPoolSize,//      最大线程数
                          long keepAliveTime, //        最大空闲时间
                          TimeUnit unit,          //          时间单位
                          BlockingQueue<Runnable> workQueue,    //    任务队列
                          ThreadFactory threadFactory,     // 线程工厂
                          RejectedExecutionHandler handler  //  饱和处理机制
    )
{ ... }
```

## ThreadPoolExecutor参数详解

1. corePoolSize :线程池的核心池大小，在创建线程池之后，线程池默认没有任何线程。

   当有任务过来的时候才会去创建创建线程执行任务。换个说法，线程池创建之后，线程池中的线程数为0，当任务过来就会创建一个线程去执行，直到线程数达到corePoolSize 之后，就会被到达的任务放在队列中。（注意是到达的任务）。换句更精炼的话：corePoolSize 表示允许线程池中允许同时运行的最大线程数。

   如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

2. maximumPoolSize :线程池允许的最大线程数，他表示最大能创建多少个线程。maximumPoolSize肯定是大于等于corePoolSize。
3. keepAliveTime :表示线程没有任务时最多保持多久然后停止。默认情况下，只有线程池中线程数大于corePoolSize 时，keepAliveTime 才会起作用。换句话说，当线程池中的线程数大于corePoolSize，并且一个线程空闲时间达到了keepAliveTime，那么就是shutdown。
4. workQueue： 一个阻塞队列，用来存储等待执行的任务，当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能。
5. threadFactory： 线程工厂，用来创建线程。
6. handler :表示当拒绝处理任务时的策略。

## 任务缓存队列

在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。

workQueue的类型为BlockingQueue，通常可以取下面三种类型：

1）有界任务队列ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2）无界任务队列LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX_VALUE；

3）直接提交队列synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

## 拒绝策略

AbortPolicy:丢弃任务并抛出RejectedExecutionException

CallerRunsPolicy：只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。

DiscardOldestPolicy：丢弃队列中最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。

DiscardPolicy：丢弃任务，不做任何处理。

## 线程池的任务处理策略：

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

## 线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

## 源码分析

首先来看最核心的execute方法，这个方法在AbstractExecutorService中并没有实现，从Executor接口，直到ThreadPoolExecutor才实现了改方法，

ExecutorService中的submit(),invokeAll(),invokeAny()都是调用的execute方法，所以execute是核心中的核心,源码分析将围绕它逐步展开。

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        /*
         * Proceed in 3 steps:
         *
         * 1. If fewer than corePoolSize threads are running, try to
         * start a new thread with the given command as its first
         * task.  The call to addWorker atomically checks runState and
         * workerCount, and so prevents false alarms that would add
         * threads when it shouldn't, by returning false.
         * 如果正在运行的线程数小于corePoolSize，那么将调用addworker 方法来创建一个新的线程，并将该任务作为新线程的第一个任务来执行。                当然，在创建线程之前会做原子性质的检查，如果条件不允许，则不创建线程来执行任务，并返回false.
```

```
         * 2. If a task can be successfully queued, then we still need
         * to double-check whether we should have added a thread
         * (because existing ones died since last checking) or that
         * the pool shut down since entry into this method. So we
         * recheck state and if necessary roll back the enqueuing if
         * stopped, or start a new thread if there are none.
         * 如果一个任务成功进入阻塞队列，那么我们需要进行一个双重检查来确保是我们已经添加一个线程（因为存在着一些线程在上次检查后他已经死亡）或者                当我们进入该方法时，该线程池已经关闭。所以，我们将重新检查状态，线程池关闭的情况下则回滚入队列，线程池没有线程的情况则创建一个新的线程。
         * 3. If we cannot queue task, then we try to add a new
         * thread.  If it fails, we know we are shut down or saturated
         * and so reject the task.            如果任务无法入队列（队列满了），那么我们将尝试新开启一个线程（从corepoolsize到扩充到maximum），如果失败了，那么可以确定原因，要么是线程池关闭了或者饱和了（达到maximum），所以我们执行拒绝策略。
         */              // 1.当前线程数量小于corePoolSize，则创建并启动线程。
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))              // 成功，则返回
return;
            c = ctl.get();
        }       // 2.步骤1失败，则尝试进入阻塞队列，
        if (isRunning(c) && workQueue.offer(command)) {            // 入队列成功，检查线程池状态，如果状态部署RUNNING而且remove成功，则拒绝任务
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);            // 如果当前worker数量为0，通过addWorker(null, false)创建一个线程，其任务为null
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }       // 3. 步骤1和2失败，则尝试将线程池的数量有corePoolSize扩充至maxPoolSize，如果失败，则拒绝任务
        else if (!addWorker(command, false))
            reject(command);
    }
```

## 常见的四种线程池

1. newFixedThreadPool

   固定大小的线程池，可以指定线程池的大小，该线程池corePoolSize和maximumPoolSize相等，阻塞队列使用的是LinkedBlockingQueue，大小为整数最大值。

   该线程池中的线程数量始终不变，当有新任务提交时，线程池中有空闲线程则会立即执行，如果没有，则会暂存到阻塞队列。对于固定大小的线程池，不存在线程数量的变化。同时使用无界的LinkedBlockingQueue来存放执行的任务。当任务提交十分频繁的时候，LinkedBlockingQueue迅速增大，存在着耗尽系统资源的问题。而且在线程池空闲时，即线程池中没有可运行任务时，它也不会释放工作线程，还会占用一定的系统资源，需要shutdown。

2. newSingleThreadExecutor

   单个线程线程池，只有一个线程的线程池，阻塞队列使用的是LinkedBlockingQueue,若有多余的任务提交到线程池中，则会被暂存到阻塞队列，待空闲时再去执行。按照先入先出的顺序执行任务。

3. newCachedThreadPool

缓存线程池，缓存的线程默认存活60秒。线程的核心池corePoolSize大小为0，核心池最大为Integer.**MAX_VALUE**,阻塞队列使用的是SynchronousQueue。是一个直接提交的阻塞队列， 他总会迫使线程池增加新的线程去执行新的任务。在没有任务执行时，当线程的空闲时间超过keepAliveTime（60秒），则工作线程将会终止被回收，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销。如果同时又大量任务被提交，而且任务执行的时间不是特别快，那么线程池便会新增出等量的线程池处理任务，这很可能会很快耗尽系统的资源。

4. newScheduledThreadPool

   定时线程池，该线程池可用于周期性地去执行任务，通常用于周期性的同步数据。

   scheduleAtFixedRate:是以固定的频率去执行任务，周期是指每次执行任务成功执行之间的间隔。

   schedultWithFixedDelay:是以固定的延时去执行任务，延时是指上一次执行成功之后和下一次开始执行的之前的时间。

以下阿里编码规范里面说的一段话：

线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。 说明：Executors各个方法的弊端：
1）newFixedThreadPool和newSingleThreadExecutor:
主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至OOM。
2）newCachedThreadPool和newScheduledThreadPool:
主要问题是线程数最大数是Integer.MAX_VALUE，可能会创建数量非常多的线程，甚至OOM。

# 通过面试题分析

1. 如何停止一个线程

   使用volatile变量终止正常运行的线程 + 抛异常法/Return法

   组合使用interrupt方法与interruptted/isinterrupted方法终止正在运行的线程 + 抛异常法/Return法

   使用interrupt方法终止 正在阻塞中的 线程

2. 何为线程安全的类?

   在线程安全性的定义中，最核心的概念就是 正确性。当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么这个类就是线程安全的。

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

   ```
   Object lock = new Object();
   synchronized (lock) {
       lock.wait();
       ...
   }
   ```

   Wait-notify机制是在获取对象锁的前提下不同线程间的通信机制。在Java中，任意对象都可以当作锁来使用，由于锁对象的任意性，所以这些通信方法需要被定义在Object类里。

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

   wait/notify机制是依赖于Java中Synchronized同步机制的，其目的在于确保等待线程从Wait()返回时能够感知通知线程对共享变量所作出的修改。如果不在同步范围内使用，就会抛出java.lang.IllegalMonitorStateException的异常。（lost wake up）

5. 并发三准则
    - 异常不会导致死锁现象：当线程出现异常且没有捕获处理时，JVM会自动释放当前线程占用的锁，因此不会由于异常导致出现死锁现象，同时还会释放CPU；
    - 锁的是对象而非引用；？
    - 有wait必有notify；
6. 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，诸如：

    - 通过加锁(Lock/Synchronized)保证对临界资源的同步互斥访问；
    - 使用volatile关键字，轻量级同步机制，但不保证原子性；
    - 使用不变类 和 线程安全类(原子类，并发容器，同步容器等)
7. volatile关键字在Java中有什么作用

    volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新，即保证了内存的可见性，除此之外还能 禁止指令重排序。此外，synchronized关键字也可以保证内存可见性。

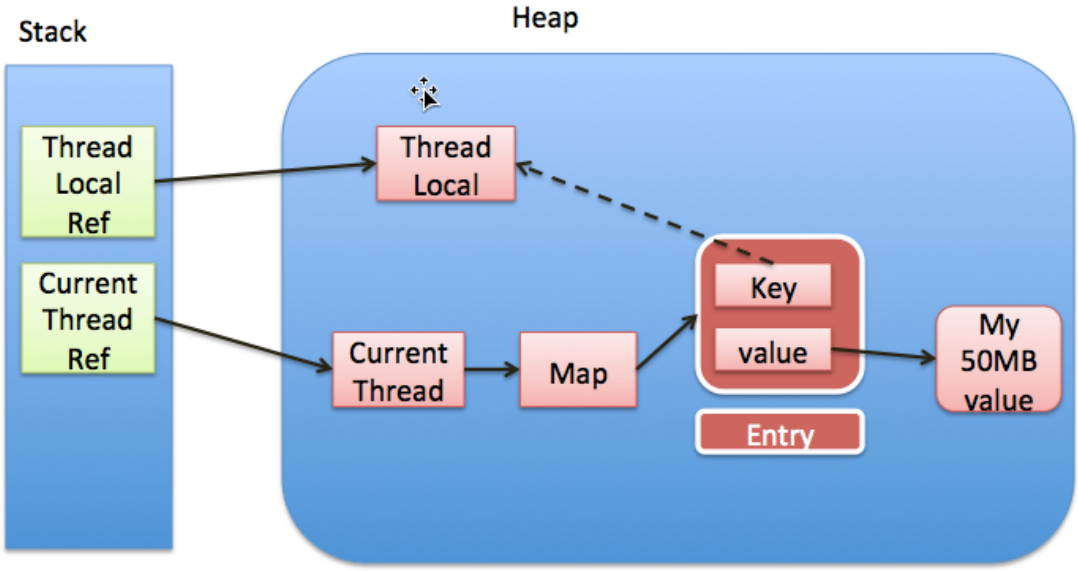
    指令重排序问题在并发环境下会导致线程安全问题，volatile关键字通过禁止指令重排序来避免这一问题。而对于Synchronized关键字，其所控制范围内的程序在执行时独占的，指令重排序问题不会对其产生任何影响，因此无论如何，其都可以保证最终的正确性。

8. ThreadLocal及其引发的内存泄露

    ThreadLocal是Java中的一种线程绑定机制，可以为每一个使用该变量的线程都提供一个变量值的副本，并且每一个线程都可以独立地改变自己的副本，而不会与其它线程的副本发生冲突。

    每个线程内部有一个 ThreadLocal.ThreadLocalMap 类型的成员变量 threadLocals，这个 threadLocals 存储了与该线程相关的所有 ThreadLocal 变量及其对应的值，也就是说，ThreadLocal 变量及其对应的值就是该Map中的一个 Entry，更直白地，threadLocals中每个Entry的Key是ThreadLocal 变量本身，而Value是该ThreadLocal变量对应的值。

    看下面的图示， 实线代表强引用，虚线代表弱引用。每个thread中都存在一个map，map的类型是上文提到的ThreadLocal.ThreadLocalMap，该map中的key为一个ThreadLocal实例。这个Map的确使用了弱引用，不过弱引用只是针对key，每个key都弱引用指向ThreadLocal对象。一旦把threadlocal实例置为null以后，那么将没有任何强引用指向ThreadLocal对象，因此ThreadLocal对象将会被 Java GC 回收。但是，与之关联的value却不能回收，因为存在一条从current thread连接过来的强引用。 只有当前thread结束以后， current thread就不会存在栈中，强引用断开，Current Thread、Map及value将全部被Java GC回收。



所以，得出一个结论就是：只要这个线程对象被Java GC回收，就不会出现内存泄露。但是如果只把ThreadLocal引用指向null而线程对象依然存在，那么此时Value是不会被回收的，这就发生了我们认为的内存泄露。比如，在使用线程池的时候，线程结束是不会销毁的而是会再次使用的，这种情形下就可能出现ThreadLocal内存泄露。

Java为了最小化减少内存泄露的可能性和影响，在ThreadLocal进行get、set操作时会清除线程Map里所有key为null的value。所以最怕的情况就是，ThreadLocal对象设null了，开始发生“内存泄露”，然后使用线程池，线程结束后被放回线程池中而不销毁，那么如果这个线程一直不被使用或者分配使用了又不再调用get/set方法，那么这个期间就会发生真正的内存泄露。因此，最好的做法是：在不使用该ThreadLocal对象时，及时调用该对象的remove方法去移除ThreadLocal.ThreadLocalMap中的对应Entry。

9. 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。

10. 什么是Java Timer类？如何创建一个有特定时间间隔的任务？

    Timer是一个调度器，可以用于安排一个任务在未来的某个特定时间执行或周期性执行。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。

```
Timer timer = new Timer();
timer.schedule(new TimerTask() {
        public void run() {
            System.out.println("abc");
        }
}, 200000 , 1000);
```

11. 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。线程池可以避免线程的频繁创建与销毁，降低资源的消耗，提高系统的反应速度。java.util.concurrent.Executors提供了几个java.util.concurrent.Executor接口的实现用于创建线程池，其主要涉及四个角色：

```
线程池：Executor
工作线程：Worker线程，Worker的run()方法执行Job的run()方法
任务Job：Runable和Callable
阻塞队列：BlockingQueue
```

    1. 线程池Executor

       Executor及其实现类是用户级的线程调度器，也是对任务执行机制的抽象，其将任务的提交与任务的执行分离开来，核心实现类包括ThreadPoolExecutor(用来执行被提交的任务)和ScheduledThreadPoolExecutor(可以在给定的延迟后执行任务或者周期性执行任务)。Executor的实现继承链条为：(父接口)Executor -> (子接口)ExecutorService -> (实现类)[ ThreadPoolExecutor + ScheduledThreadPoolExecutor ]。

    2. 任务Runable/Callable

       Runnable(run)和Callable(call)都是对任务的抽象，但是Callable可以返回任务执行的结果或者抛出异常。

    3. 任务执行状态Future

       Future是对任务执行状态和结果的抽象，核心实现类是furtureTask (所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值)；

4. 四种常用的线程池
5. 线程池的饱和策略
6. 线程池调优
    - 设置最大线程数，防止线程资源耗尽；
    - 使用有界队列，从而增加系统的稳定性和预警能力(饱和策略)；
    - 根据任务的性质设置线程池大小：CPU密集型任务(CPU个数个线程)，IO密集型任务(CPU个数两倍的线程)，混合型任务(拆分)。

12. CAS： *CAS自旋volatile变量，是一种很经典的用法。*

    CAS，Compare and Swap即比较并交换，设计并发算法时常用到的一种技术。CAS有3个操作数，内存值V，旧的预期值A，新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。CAS是通过unsafe类的compareAndSwap (JNI, Java Native Interface) 方法实现的，该方法包括四个参数：第一个参数是要修改的对象，第二个参数是对象中要修改变量的偏移量，第三个参数是修改之前的值，第四个参数是预想修改后的值。

    CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题：ABA问题、循环时间长开销大和只能保证一个共享变量的原子操作。

    CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题：ABA问题、循环时间长开销大和只能保证一个共享变量的原子操作。

    > ABA问题：因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。
    >
    > 不适用于竞争激烈的情形中：并发越高，失败的次数会越多，CAS如果长时间不成功，会极大的增加CPU的开销。因此CAS不适合竞争十分频繁的场景。
    >
    > 只能保证一个共享变量的原子操作：当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，因此可以把多个变量放在一个对象里来进行CAS操作。

13. AQS： 队列同步器
14. Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

    synchronized是Java的关键字，是Java的内置特性，在JVM层面实现了对临界资源的同步互斥访问。Synchronized的语义底层是通过一个monitor对象来完成的，线程执行monitorenter/monitorexit指令完成锁的获取与释放。而Lock是一个Java接口(API如下图所示)，是基于JDK层面实现的，通过这个接口可以实现同步访问，它提供了比synchronized关键字更灵活、更广泛、粒度更细的锁操作，底层是由AQS实现的。二者之间的差异总结如下：

实现层面：synchronized（JVM层面）、Lock（JDK层面）

响应中断：Lock 可以让等待锁的线程响应中断，而使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

立即返回：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，而synchronized却无法办到；

读写锁：Lock可以提高多个线程进行读操作的效率

可实现公平锁：Lock可以实现公平锁，而sychronized天生就是非公平锁

显式获取和释放：synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

15. Condition

    Condition可以用来实现线程的分组通信与协作。以生产者/消费者问题为例，

    wait/notify/notifyAll:在队列为空时，通知所有线程；在队列满时，通知所有线程，防止生产者通知生产者，消费者通知消费者的情形产生。

    await/signal/signalAll：将线程分为消费者线程和生产者线程两组：在队列为空时，通知生产者线程生产；在队列满时，通知消费者线程消费。

16. 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。特别地，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。另外，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。特别地，SynchronousQueue是一个没有容量的阻塞队列，每个插入操作必须等待另一个线程的对应移除操作，反之亦然。CachedThreadPool使用SynchronousQueue把主线程提交的任务传递给空闲线程执行。

17. 同步容器（强一致性）

    同步容器指的是 Vector、Stack、HashTable及Collections类中提供的静态工厂方法创建的类。其中，Vector实现了List接口，Vector实际上就是一个数组，和ArrayList类似，但是Vector中的方法都是synchronized方法，即进行了同步措施；Stack也是一个同步容器，它的方法也用synchronized进行了同步，它实际上是继承于Vector类；HashTable实现了Map接口，它和HashMap很相似，但是HashTable进行了同步处理，而HashMap没有。

    Collections类是一个工具提供类，注意，它和Collection不同，Collection是一个顶层的接口。在Collections类中提供了大量的方法，比如对集合或者容器进行排序、查找等操作。最重要的是，在它里面提供了几个静态工厂方法来创建同步容器类。

18. 什么是CopyOnWrite容器(弱一致性)?

    CopyOnWrite容器即写时复制的容器，适用于读操作远多于修改操作的并发场景中。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器，它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器主要存在两个弱点：

> 容器对象的复制需要一定的开销，如果对象占用内存过大，可能造成频繁的YoungGC和Full GC；
>
> CopyOnWriteArrayList不能保证数据实时一致性，只能保证最终一致性。

19. ConcurrentHashMap (弱一致性)

    ConcurrentHashMap的弱一致性主要是为了提升效率，也是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。ConcurrentHashMap的弱一致性主要体现在以下几方面：

    1. get操作是弱一致的：get操作只能保证一定能看到已完成的put操作；
    2. clear操作是弱一致的：在清除完一个segments之后，正在清理下一个segments的时候，已经清理的segments可能又被加入了数据，因此clear返回的时候，ConcurrentHashMap中是可能存在数据的。

       ```
       public void clear() {
           for (int i = 0; i < segments.length; ++i)
               segments[i].clear();
       }
       ```

    3. ConcurrentHashMap中的迭代操作是弱一致的(未遍历的内容发生变化可能会反映出来)：在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出ConcurrentModificationException异常。如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。

20. happens-before

    happens-before 指定了两个操作间的执行顺序：如果 A happens before B，那么Java内存模型将向程序员保证 —— A 的执行顺序排在 B 之前，并且 A 操作的结果将对 B 可见，其具体包括如下8条规则：

    > 程序顺序规则：单线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作；
    >
    > 管程锁定规则：一个unlock操作先行发生于对同一个锁的lock操作；
    >
    > volatile变量规则：对一个Volatile变量的写操作先行发生于对这个变量的读操作；
    >
    > 线程启动规则：Thread对象的start()方法先行发生于此线程的其他动作；
    >
    > 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；
    >
    > 线程终止规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；
    >
    > 对象终结规则：一个对象的初始化完成先行发生于它的finalize()方法的开始；
    >
    > 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；

21. 锁优化技术

    锁优化技术的目的在于线程之间更高效的共享数据，解决竞争问题，更好提高程序执行效率。

自旋锁(上下文切换代价大)：互斥锁 -> 阻塞 -> 释放CPU，线程上下文切换代价较大 + 共享变量的锁定时间较短 == 让线程通过自旋等一会儿，自旋锁

锁粗化(一个大锁优于若干小锁)：一系列连续操作对同一对象的反复频繁加锁/解锁会导致不必要的性能损耗，建议粗化锁
一般而言，同步范围越小越好，这样便于其他线程尽快拿到锁，但仍然存在特例。

偏向锁(有锁但当前情形不存在竞争)：消除数据在无竞争情况下的同步原语，提高带有同步但无竞争的程序性能。

锁消除(有锁但不存在竞争，锁多余)：JVM编译优化，将不存在数据竞争的锁消除

# 访问权限

## 访问权限简介

**访问权限控制：** 指的是本类及本类内部的成员（成员变量、成员方法、内部类）对其他类的可见性，即这些内容是否允许其他类访问。

Java 中一共有四种访问权限控制，其权限控制的大小情况是这样的：public > protected > default(包访问权限) > private ,具体的权限控制看下面表格，列所指定的类是否有权限允许访问行的权限控制下的内容：

| 访问权限 | 本类 | 本包的类 | 子类 | 非子类的外包类 |
| --- | --- | --- | --- | --- |
| public | 是 | 是 | 是 | 是 |
| protected | 是 | 是 | 是 | 否 |
| default | 是 | 是 | 否 | 否 |
| private | 是 | 否 | 否 | 否 |

1、**public：** 所修饰的类、变量、方法，在内外包均具有访问权限；
2、**protected：** 这种权限是为继承而设计的，protected所修饰的成员，对所有子类是可访问的，但只对同包的类是可访问的，对外包的非子类是不可以访问；
3、**包访问权限（default）：** 只对同包的类具有访问的权限，外包的所有类都不能访问；
4、**private：** 私有的权限，只对本类的方法可以使用；

**注意：** 要区分开 protected 权限、包访问权限，正确使用它们；

- 当某个成员能被所有的子类继承，但不能被外包的非子类访问，就是用protected；
- 当某个成员的访问权限只对同包的类开放，包括不能让外包的类继承这个成员，就用包访问权限；

**使用访问权限控制的原因：**

1）使用户不要碰触那些他们不该碰触的部分；
2）类库设计者可以更改类的内部工作的方式，而不会担心这样会对用户产生重大影响；

## 访问权限控制的使用场景

访问权限使用的场景可以总结为下面的五种场景，分别对访问权限的使用有不同的限制：

1. 外部类的访问控制

**外部类（外部接口）** 是相对于内部类（也称为嵌套类）、内部接口而言的。外部类的访问控制只能是这两种：**public** 、**default** 。

2. 类里面的成员访问控制

   类里面的成员分为三类： **成员变量、成员方法、成员内部类（内部接口）**

   类里面的成员的访问控制可以是四种，也就是可以使用所有的访问控制权限

```
public class OuterClass {

    public int aa; //可以被所有的类访问
    protected boolean bb; //可以被所有子类以及本包的类使用

    void cc() { //default 访问权限，能在本包范围内使用
        System.out.println("包访问权限");
    }

    //private权限的内部类，即这是私有的内部类，只能在本类使用
    private class InnerClass{

    }
}
```

   **注意：**
   这里的类里面的成员 是指类的**全局成员**，并没有包括局部的成员（局部变量、局部内部类，没有局部内部接口）。或者说，**局部成员是没有访问权限控制的**，因为局部成员只在其所在的作用域内起作用，不可能被其他类访问到。

```
public void count(){
    //局部成员变量
    public int amount;//编译无法通过，不能用public修饰
    int money;//编译通过
    //局部嵌套接口
    class customer{//编译通过

    }
}
```

   上面的两种场景几乎可以适应所有的情况，但有一些情况比较特殊，还做了有些额外访问权限的要求

3. 抽象方法的访问权限

   普通方法是可以使用四种访问权限的，但抽象方法是有一个限制：不能用private 来修饰，也即抽象方法不能是私有的，否则，子类就无法继承实现抽象方法。

4. 接口成员的访问权限

   接口由于其的特殊性，所有成员的访问权限都规定得死死的，下面是接口成员的访问权限：

   - **变量：** public static final
   - **抽象方法：** public abstract
   - **静态方法：** public static，JDK1.8后才支持
   - **内部类、内部接口：** public static

   也因为所有的一切都默认强制规定好了，所以我们在用的时候，并不一定需要完整写出所有的修饰符，编译器会帮我们完成的，也就是，可以少写修饰符，但不能写错修饰符。

```
public interface Interface_Test {

    public int aa = 6; //少写了  static final
    int bb = 5; //

    //嵌套接口，可以不写public static
     interface cc{

    }
}
```

5. 构造器的访问权限

   **构造器的访问权限可以是以上四种权限中的任意一种：**

   1、采用 private：一般是不允许直接构造这个类的对象，再结合工厂方法（static方法），实现单例模式。注意：所有子类都不能继承它。
   2、采用包访问控制：比较少用，这个类的对象只能在本包中使用，但是如果这个类有static 成员，那么这个类还是可以在外包使用；（也许可以用于该类的外包单例模式）。
   注意：外包的类不能继承这个类；
   3、采用 protected ： 就是为了能让所有子类继承这个类，但是外包的非子类不能访问这个类；
   4、采用 public ： 对于内外包的所有类都是可访问的；

   **注意：** 构造方法有点特殊，因为子类的构造器初始化时，都要调用父类的构造器，所以一旦父类构造器不能被访问，那么子类的构造器调用失败，意味子类继承父类失败！

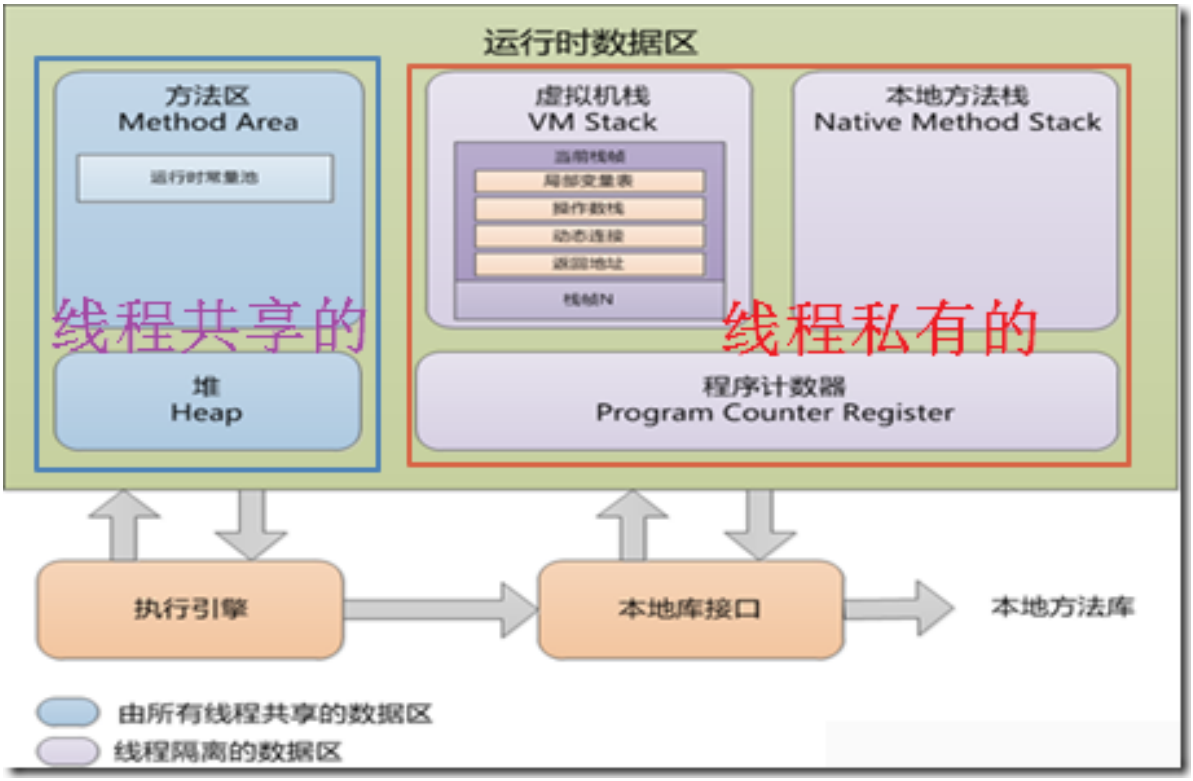
# 内存结构

## JDK 1.6

我们都知道，Java程序在执行前首先会被编译成字节码文件，然后再由Java虚拟机执行这些字节码文件从而使得Java程序得以执行。事实上，在程序执行过程中，内存的使用和管理一直是值得关注的问题。Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域，这些数据区域都有各自的用途，以及创建和销毁的时间，并且它们可以分为两种类型：线程共享的方法区和堆，线程私有的虚拟机栈、本地方法栈和程序计数器。在此基础上，我们探讨了在虚拟机中对象的创建和对象的访问定位等问题，并分析了Java虚拟机规范中异常产生的情况。

### 虚拟机内存模型

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域，这些数据区域可以分为两个部分：一部分是线程共享的，一部分则是线程私有的。其中，线程共享的数据区包括方法区和堆，线程私有的数据区包括虚拟机栈、本地方法栈和程序计数器。如下图所示：




1. 线程私有的数据区

   线程私有的数据区 包括 程序计数器、 虚拟机栈 和 本地方法栈 三个区域，它们的内涵分别如下：

   1. 程序计数器

      我们知道，线程是CPU调度的基本单位。在多线程情况下，当线程数超过CPU数量或CPU内核数量时，线程之间就要根据 **时间片轮询抢夺CPU时间资源。**也就是说，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此，**为了线程切换后能够恢复到正确的执行位置，每条线程都需要一个独立的程序计数器去记录其正在执行的字节码指令地址。**

      因此，程序计数器是线程私有的一块较小的内存空间，其可以看做是当前线程所执行的字节码的行号指示器。如果线程正在执行的是一个 Java 方法，计数器记录的是正在执行的字节码**指令的地址**；如果正在执行的是 Native 方法，则计数器的**值为空。**

      程序计数器是唯一一个没有规定任何 OutOfMemoryError 的区域。

   2. 虚拟机栈

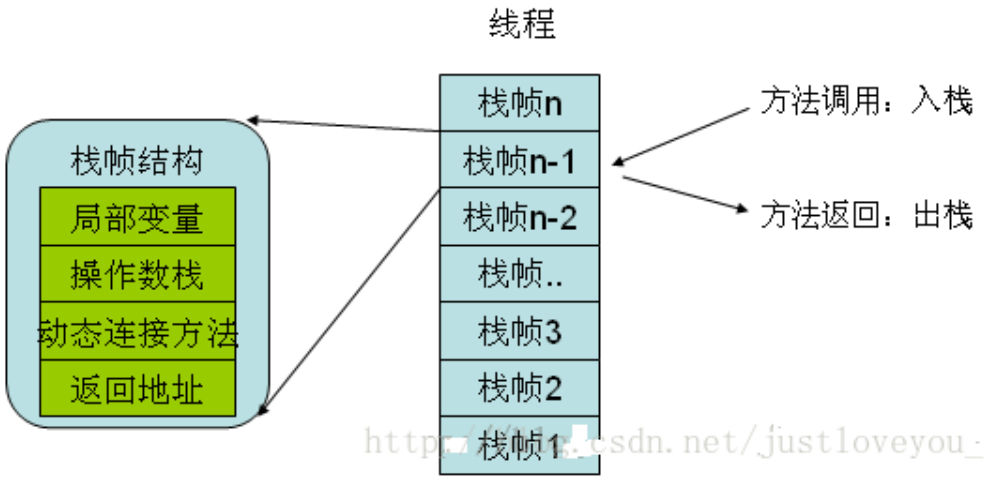
      虚拟机栈描述的是Java方法执行的内存模型，是**线程私有的。**每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息，而且 每个方法从调用直至完成的过程，对应一个栈帧在虚拟机栈中入栈到出栈的过程。其中，**局部变量表主要存放一些基本类型的变量（int, short, long, byte, float, double, boolean, char）和 reference（可能是句柄指针，可能是直接引用），它们可以是方法参数，也可以是方法的局部变量。**

      总结：

      栈里面其实就是一个个栈帧，栈帧里面包含：局部变量表、操作数栈、动态链接、方法出口等信息。局部变量表包含可以存基本变量和reference。

      虚拟机栈有两种异常情况：StackOverflowError 和 OutOfMemoryError。我们知道，一个线程拥有一个自己的栈，这个栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法，-Xss 参数可以设置虚拟机栈大小），**若线程请求的栈深度大于虚拟机允许的深度，则抛出 StackOverFlowError 异常。**此外，栈的大小可以是固定的，也可以是动态

扩展的，若虚拟机栈可以动态扩展（大多数虚拟机都可以），但扩展时无法申请到足够的内存(**比如没有足够的内存为一个新创建的线程分配栈空间时**)，则抛出 OutofMemoryError 异常。下图为栈帧结构图：




3. 本地方法栈

   本地方法栈与Java虚拟机栈非常相似，也是线程私有的，区别是虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈为虚拟机执行 Native 方法服务。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

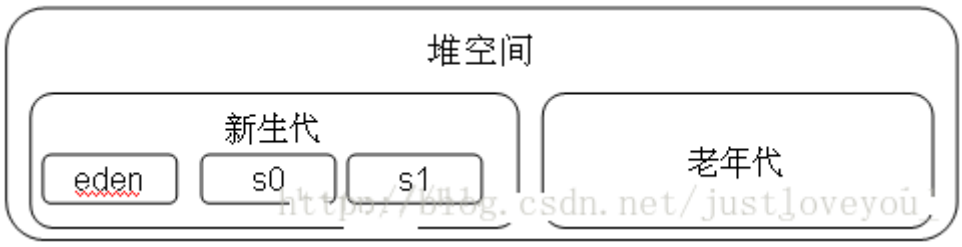
2. 线程共享的数据区

   线程共享的数据区 具体包括 Java堆 和 方法区 两个区域，它们的内涵分别如下：

   1. Java 堆

      **Java 堆的唯一目的就是存放对象实例，几乎所有的对象实例（和数组）都在这里分配内存。**Java堆是线程共享的，类的对象从中分配空间，这些对象通过new、newarray、 anewarray 和 multianewarray 等指令建立，它们不需要程序代码来显式的释放。

      由于Java堆唯一目的就是用来存放对象实例，因此其也是垃圾收集器管理的主要区域，故也称为称为 GC堆。**从内存回收的角度看，由于现在的垃圾收集器基本都采用分代收集算法，所以为了方便垃圾回收Java堆还可以分为 新生代 和 老年代** 。新生代用于存放刚创建的对象以及年轻的对象，如果对象一直没有被回收，生存得足够长，对象就会被移入老年代。新生代又可进一步细分为 eden、survivorSpace0 和 survivorSpace1。刚创建的对象都放入 eden，s0 和 s1 都至少经过一次GC并幸存。如果幸存对象经过一定时间仍存在，则进入老年代。更多关于Java堆和分代收集算法的介绍，请移步我的博文《Java 垃圾回收机制概述》。下图给出了Java堆的结构图：

      注意，**Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。**而且，Java堆在实现时，既可以是固定大小的，也可以是可拓展的，并且主流虚拟机都是按可扩展来实现的（通过-Xmx(最大堆容量) 和 -Xms(最小堆容量)控制）。如果在堆中没有内存完成实例分配，并且堆也无法再拓展时，将会抛出 OutOfMemoryError 异常。

      1. TLAB (Thread Local Allocation Buffer，线程私有分配缓冲区)

Sun Hotspot JVM 为了提升对象内存分配的效率，对于所创建的线程都会分配一块独立的空间 TLAB，其大小由JVM根据运行的情况计算而得。在TLAB上分配对象时不需要加锁(相对于CAS配上失败重试方式 )，因此JVM在给线程的对象分配内存时会尽量的在TLAB上分配，在这种情况下JVM中分配对象内存的性能和C基本是一样高效的，但如果对象过大的话则仍然是直接使用堆空间分配。

在下文中我们提到，虚拟机为新生对象分配内存时，需要考虑修改指针 (该指针用于划分内存使用空间和空闲空间) 时的线程安全问题，因为存在可能出现正在给对象A分配内存，指针还未修改，对象B又同时使用原来的指针分配内存的情况。**TLAB 的存在就是为了解决这个问题：每个线程在Java堆中预先分配一小块内存 TLAB，哪个线程需要分配内存就在自己的TLAB上进行分配，若TLAB用完并分配新的TLAB时，再加同步锁定，这样就大大提升了对象内存分配的效率。**

2. 方法区

方法区与Java堆一样，也是线程共享的并且不需要连续的内存，其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。**方法区通常和永久区(Perm)关联在一起，但永久代与方法区不是一个概念，只是有的虚拟机用永久代来实现方法区，这样就可以用永久代GC来管理方法区，省去专门内存管理的工作。根据Java虚拟机规范的规定，当方法区无法满足内存分配的需求时，将抛出 OutOfMemoryError 异常。

1. 运行时常量池

运行时常量池（Runtime Constant Pool）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。**其中，字面量比较接近Java语言层次的常量概念，如文本字符串、被声明为final的常量值等；而符号引用则属于编译原理方面的概念，包括以下三类常量：**类和接口的全限定名、字段的名称和描述符 以及 方法的名称和描述符。**因为运行时常量池（Runtime Constant Pool）是方法区的一部分，那么当常量池无法再申请到内存时也会抛出 OutOfMemoryError 异常。

运行时常量池相对于Class文件常量池的一个重要特征是具备动态性。Java语言并不要求常量一定只有编译期才能产生，运行期间也可能将新的常量放入池中，比如字符串的手动入池方法intern()。

3. Java堆 与 方法区的区别

Java堆是 Java代码可及的内存，是留给开发人员使用的；而非堆（Non-Heap）是JVM留给自己用的，所以方法区、JVM内部处理或优化所需的内存 (如JIT编译后的代码缓存)、每个类结构(如运行时常量池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

4. 方法区的回收

方法区的内存回收目标主要是针对 **常量池的回收** 和 **对类型的卸载。**回收废弃常量与回收Java堆中的对象非常类似。以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个String对象是叫做“abc”的，换句话说是没有任何String对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果在这时候发生内存回收，而且必要的话，这个“abc”常量就会被系统“请”出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面3个条件才能算是“无用的类”：

1. 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例；
2. 加载该类的ClassLoader已经被回收；
3. 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述3个条件的无用类进行回收(卸载)，这里说的仅仅是“可以”，而不是和对象一样，不使用了就必然会回收。特别地，在大量使用反射、动态代理、CGLib等bytecode框架的场景，以及动态生成JSP和OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

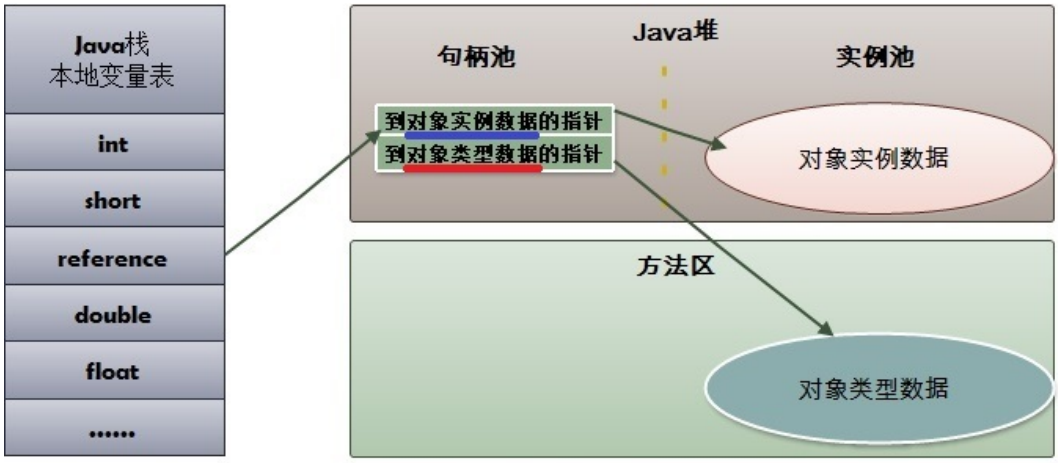
# JAVA对象在虚拟机中的创建与访问定位

Java是一门面向对象的编程语言，在Java程序运行过程中无时无刻都有对象被创建和使用。在此，我们以最流行的HotSpot虚拟机以及常用的内存区域Java堆为例来探讨在虚拟机中对象的创建和对象的访问等问题。

1. 对象在虚拟机中的创建过程
    1. 检查虚拟机是否加载了所要new的类，若没加载，则首先执行相应的类加载过程。虚拟机遇到new指令时，首先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个引用代表的类是否已经被加载、解析和初始化过。
    2. 在类加载检查通过后，对象所需内存的大小在类加载完成后便可完全确定，虚拟机就会为新生对象分配内存。一般来说，**根据Java堆中内存是否绝对规整，**内存的分配有两种方式：
        1. 指针碰撞：如果Java堆中内存绝对规整，所有用过的内存放在一边，空闲内存放在另一边，中间一个指针作为分界点的指示器，那分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相同的距离。
        2. 空闲列表：如果Java堆中内存并不规整，那么虚拟机就需要维护一个列表，记录哪些内存块是可用的，以便在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录。

        除了如何划分可用空间之外，还需要考虑修改指针 (该指针用于划分内存使用空间和空闲空间)时的线程安全问题，因为存在可能出现正在给对象A分配内存，指针还未修改，对象B又同时使用原来的指针分配内存的情况。解决这个问题有两种方案：
        1. **对分配内存空间的动作进行同步处理：**采用CAS+失败重试的方式保证更新操作的原子性；
        2. **把内存分配的动作按照线程划分的不同的空间中：**每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），哪个线程要分配内存，就在自己的TLAB上分配，如果TLAB用完并分配新的TLAB时，再加同步锁定。
    3. 内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值。如果使用TLAB，也可以提前到TLAB分配时进行。这一步操作保证了对象的实例字段在Java代码中可以不赋初值就直接使用，程序能访问到这些字段的数据类型所对应的零值。
    4. 在上面的工作完成之后，从虚拟机的角度来看，一个新的对象已经产生了，但从Java程序的视角来看，对象的创建才刚刚开始，此时会执行方法把对象按照程序员的意愿进行初始化，从而产生一个真正可用的对象。
2. 对象在虚拟机中的访问定位

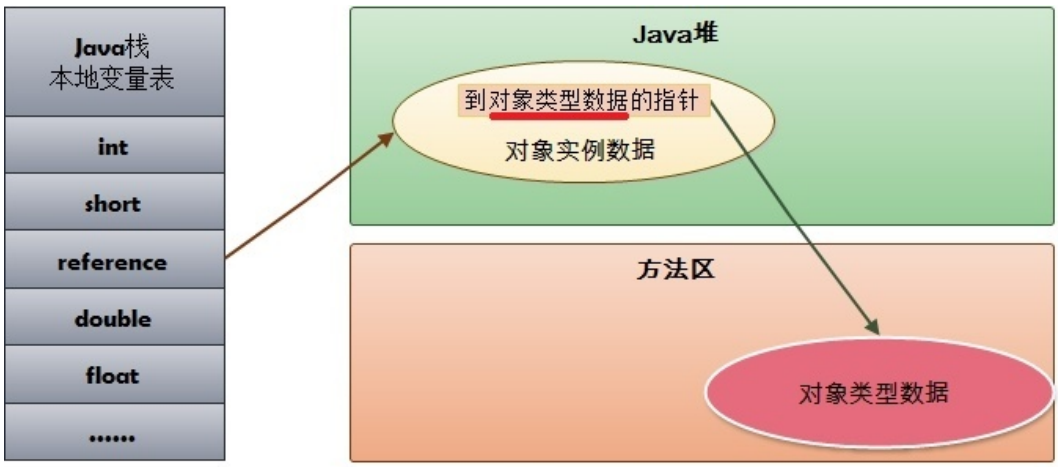
    创建对象是为了使用对象，我们的Java程序通过栈上的reference数据来操作堆上的具体对象。在虚拟机规范中，reference类型中只规定了一个指向对象的引用，并没有定义这个引用使用什么方式去定位、访问堆中的对象的具体位置。目前的主流的访问方式有使用句柄访问和直接指针访问两种。
    1. **句柄访问**：Java堆中会划分出一块内存作为句柄池，栈中的reference指向对象的句柄地址，句柄中包含了对象实例数据和类型数据各自的具体地址信息，如下图所示。




句柄方式访问对象



2. **直接指针访问**：reference中存储的就是对象地址。




直接指针方式访问对象



总的来说，这两种对象访问定位方式各有千秋。使用句柄访问的最大好处就是reference中存储的是稳定的句柄地址，对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，reference本身不需要修改；而使用直接指针访问的最大好处就是速度快，节省了一次指针定位的时间开销。

## 内存异常产生情况分析

1. Java堆溢出（OOM）

Java堆用于存储对象的实例，只要不断地创建对象，并且保证GC roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量到达最大堆的容量限制后就会产生内存溢出异常。如下所示，

```
public class Test {

public static void main(String[] args){
        List list=new ArrayList();   // 持有“大对象”的引用，防止垃圾回收
        while(true){
            int[] tmp = new int[10000000];  // 不断创建“大对象”
            list.add(tmp);
        }
    }
}
```

要解决这个异常，一般先通过内存映像分析工具对堆转储快照分析，确定内存的对象是否是必要的，即判断是 内存泄露 还是 内存溢出。如果是内存泄露，可以进一步通过工具查看泄露对象到GC Roots的引用链，比较准确地定位出泄露代码的位置。如果是内存溢出，可以调大虚拟机堆参数，或者从代码上检查是否存在某些对象生命周期过长的情况。

2. 虚拟机栈和本地方法栈溢出 (SOF/OOM)

    1. SOF

        如果线程请求的栈深度大于虚拟机栈允许的最大深度，将抛出StackOverflowError异常。我们知道，每当Java程序启动一个新的线程时，Java虚拟机会为它分配一个栈，并且Java虚拟机栈以栈帧为单位保持线程运行状态。每当线程调用一个方法时，JVM就压入一个新的栈帧到这个线程的栈中，只要这个方法还没返回，这个栈帧就存在。 那么可以想象，如果方法的嵌套调用层次太多，比如递归调用，随着Java虚拟机栈中的栈帧的不断增多，最终很可能会导致这个线程的栈中的所有栈帧的大小的总和大于-Xss设置的值，从而产生StackOverflowError溢出异常。看下面的栗子：

        ```
        public class Test {

            public static void main(String[] args) {
                  method();
            }

            // 递归调用导致 StackOverflowError
            public static void method(){
                method();
            }
        }
        ```

    2. OOM

        如果虚拟机在拓展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。在虚拟机栈和本地方法栈发生OOM异常场景如下：当Java 程序启动一个新线程时，若没有足够的空间为该线程分配Java栈(一个线程Java栈的大小由-Xss设置决定)，JVM将抛出OutOfMemoryError异常。

3. 方法区和运行时常量池溢出 (OOM)

    运行时常量池溢出的情况： String.intern()是一个native方法，在JDK1.6及之前的版本中，它的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象，否则将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。由于常量池分配在永久代中，如果不断地使用intern方法手动入池字符串，则会抛出OutOfMemoryError异常。但在JDK1.7及其以后的版本中，对intern（）方法的实现作了进一步改进，其不会再复制实例到常量池中，而仅仅是在常量池中记录首次出现的实例的引用。看下面的例子（在JDK1.7中运行），

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

        String str1 = new StringBuilder("aa").append("c").toString();
        System.out.println(str1.intern() == str1);

        String str2 = new StringBuilder("java").toString();
        System.out.println(str2.intern() == str2);
    }/* Output:
        true
        false
     *///:~
```

为什么第一个返回true，而第二个返回false呢？因为在JDK1.7中，intern（）方法的实现不会再复制实例，只是在常量池中记录 首次 出现的实例的引用，因此str1.intern()和str1指向的是同一个字符串，所以返回true。同一个引用。对于“java”这个字符串，由于在执行StringBuilder.toString()之前已经出现过，所以字符串常量池中在new StringBuilder(“java”).toString()之前已经有它的引用了，不符合首次出现的原则，因此返回fasle。有人可能心里可能就要嘀咕了，为啥第二个不符合首次出现的原则，而第一个就符合首次出现的原则呢？ 实际上，

```
String str2 = new StringBuilder("java").toString();
```

等价于：

```
String s1 = "java";
StringBuilder sb = new StringBuilder(s1);
String str2 = sb.toString();

// StringBuilder 的 toString()方法
public String toString() {
        // Create a copy, don't share the array
        return new String(value, 0, count);
}
```

由上面代码可知，字符串”java”早就出现了，因此不符合首次出现的原则，返回false。同理，“计算机软件”这个字符串在new StringBuilder(“aa”).append(“c”).toString()之前从未出现过，因此符合首次出现的原则，返回true。这里append只改变堆，常量池会出现aa和c，没有aac，堆有。

要想更彻底地了解本实例，建议移步我的博文 《Java String 综述(上篇)》 和 《Java String 综述(下篇)》 进行进一步了解。

（**去看看**）

方法区溢出的情况：一个类要被垃圾回收器回收掉，判断条件是比较苛刻的。 在经常动态产生大量Class的应用中，需要特别注意类的回收状况，比如动态语言、大量JSP或者动态产生JSP文件的应用（JSP第一次运行时需要编译为Java类）、基于OSGi的应用（即使是同一个类文件，被不同的加载器加载也会视为不同的类）等

# 1.10 内部类

- 在另一个类的里面定义的类就是内部类
- 内部类是编译器现象，与虚拟机无关。
- 编译器会将内部类编译成用$分割外部类名和内部类名的常规类文件，而虚拟机对此一无所知。

内部类可以是static的，也可用public，default，protected和private修饰。（而外部类即类名和文件名相同的只能使用public和default）。

# 优点

- 每个内部类都能独立地继承一个（接口的)实现，所以无论外部类是否已经继承了某个（接口的)实现，对于内部类都没有影响。
- 接口只是解决了部分问题，而内部类使得多重继承的解决方案变得更加完整。
- 用内部类还能够为我们带来如下特性：

  1、内部类可以有多个实例，每个实例都有自己的状态信息，并且与其他外部对象的信息相互独立。

  2、在单个外部类中，可以让多个内部类实现不同的接口，或者继承不同的类。外部类想要多继承的类可以分别由内部类继承，并进行Override或者直接复用。然后外部类通过创建内部类的对象来使用该内部对象的方法和成员，从而达到复用的目的，这样外部内就具有多个父类的所有特征。

  3、创建内部类对象的时刻并不依赖于外部类对象的创建。

  4、内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

  5、内部类提供了更好的封装，除了该外部类，其他类都不能访问
- 只有静态内部类可以同时拥有静态成员和非静态成员，其他内部类只有拥有非静态成员。

# 成员内部类：就像外部类的一个成员变量

- 注意内部类的对象总有一个外部类的引用
- 当创建内部类对象时，会自动将外部类的this引用传递给当前的内部类的构造方法。

# 静态内部类：就像外部类的一个静态成员变量

```
public class OuterClass {

    private static class StaticInnerClass {
        int id;
        static int increment = 1;
    }
}
//调用方式：
//外部类.内部类 instanceName = new 外部类.内部类();
```

# 局部内部类：定义在一个方法或者一个块作用域里面的类

- 想创建一个类来辅助我们的解决方案，又不希望这个类是公共可用的，所以就产生了局部内部类，局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，它只能在该方法和属性中被使用，出了该方法和属性就会失效。
- JDK1.8之前不能访问非final的局部变量!
- 生命周期不一致：
- 方法在栈中，对象在堆中；方法执行完，对象并没有死亡
- 如果可以使用方法的局部变量，如果方法执行完毕，就会访问一个不存在的内存区域。
- 而final是常量，就可以将该常量的值复制一份，即使不存在也不影响。

```
public Destination destination(String str) {
    class PDestination implements Destination {
        private String label;

        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() {
            return label;
        }
    }
    return new PDestination(str);
}
```

# 匿名内部类：必须继承一个父类或实现一个接口

- 匿名内部类和局部内部类在JDK1.8 之前都不能访问一个非final的局部变量，只能访问final的局部变量，原因是生命周期不同，可能栈中的局部变量已经被销毁，而堆中的对象仍存活，此时会访问一个不存在的内存区域。假如是final的变量，那么编译时会将其拷贝一份，延长其生命周期。
- 拷贝引用，为了避免引用值发生改变，例如被外部类的方法修改等，而导致内部类得到的值不一致，于是用final来让该引用不可改变。
- 但在JDK1.8之后可以访问一个非final的局部变量了，前提是非final的局部变量没有修改，表现得和final变量一样才可以!
- 为什么只有匿名和局部 内部类强调final。因为只有他们是在局部代码块和方法中的，只有他们会有生命周期问题。

```
interface AnonymousInner {
    int add();
}
public class AnonymousOuter {
    public AnonymousInner getAnonymousInner(){
        int x = 100;
        return new AnonymousInner() {
            int y = 100;
            @Override
            public int add() {
                return x + y;
            }
        };
    }
}
```

# 详细解读

https://blog.csdn.net/justloveyou_/article/details/53245561

内部类是一种辅助多重继承的方式，除此之外，内部类还可以很好的实现隐藏，如私有成员内部类。内部类共有四种类型：成员内部类、静态内部类、局部内部类和匿名内部类：

- 成员内部类：成员内部类是外围类的一个成员，是**依附于外围类**的，所以只有先创建了外围对象才能够创建内部类对象。也正是由于这个原因，成员内部类也不能含有static的变量和方法。（类似成员变量）

- 静态内部类：静态内部类就是修饰为static的内部类，这个内部类对象**不依赖于外部类对象**，所以说我们可以直接创建内部类对象，但它只能直接访问外部类的所有静态成员和方法。（类似静态变量）
- 局部内部类：局部内部类和成员内部类一样被编译，只是它的**作用域发生了改变**，他只能在该方法和属性中被使用，出了方法和属性就会失效。（类似代码块）
- 匿名内部类：定义匿名内部类的前提是，**内部类必须要继承一个类或者实现一个接口**，格式为new 父类或接口(){定义子类的内容(如函数等)}。也就是说，匿名内部类最终提供给我们的是一个**匿名子类的对象**。

## 内部类概述：

1. 内部类基础

   **内部类指的是在一个类的内部所定义的类，类名不需要和源文件名相同。内部类是一个编译时的概念，一旦编译成功，内部类和外部类就会成为两个完全不同的类。**例如，对于一个名为 Outer 的外部类和在其内部定义的名为 Inner 的内部类，在编译完成后，会出现 Outer.class 和 Outer$inner.class 两个类。因此，**内部类的成员变量/方法名可以和外部类的相同。**内部类可以是静态static的，也可用 public，default，protected 和 private 修饰。 特别地，关于 Java源文件名与类名的关系( java源文件名的命名与内部类无关,以下3条规则中所涉及的类和接口均指的是外部类/接口)，需要符合下面三条规则：

   1. 如果java源文件包含public类(public接口)，则源文件名必须与public类名(public接口名)相同。

      一个java源文件中，如果有public类或public接口，那么就只能有一个public类或一个public接口，不能有多个public的类或接口。当然，一个java源文件中可以有多个包可见的类或接口，即默认访问权限修饰符(类名前没有访问权限修饰符)。public类(接口) 与 包可见的类(接口)在文件中的顺序可以随意，即public类(接口)可以不在第一个的位置。

   2. 如果java源文件不包含public类(public接口)，则java源文件名没有限制。

      只要符合文件名的命名规范就可以，可以不与文件中任一个类或接口同名，当然，也可以与其中之一同名。

   3. 类和接口的命名不能冲突。

      同一个包中的任何一个类或接口的命名都不能相同。不同包中的类或接口的命名可以相同，因为通过包可以把它们区分开来。

2. 内部类作用

   使用内部类可以给我们带来以下优点：

   1. 内部类可以很好的实现隐藏（一般的非内部类，是不允许有 private、protected和default 权限的（默认是friendly，提供包内访问），但内部类可以）；

      ```
      //测试接口
      public interface InterfaceTest {
          public void test();
      }

      //外部类
      public class Example {

          //内部类
          private class InnerClass implements InterfaceTest{
              @Override
              public void test() {
                  System.out.println("I am Rico.");
      ```

```
        }
    }

    //外部类方法
    public InterfaceTest getInnerInstance(){
        return new InnerClass();
    }
}

//客户端
public class Client {
    public static void main(String[] args) {
        Example ex = new Example();
        InterfaceTest test = ex.getInnerInstance();
        test.test();
    }
}/* Output:
        I am Rico.
 *///:~
```

对客户端而言，我们可以通过 Example 的getInnerInstance()方法得到一个InterfaceTest 实例，但我们并不知道这个实例是如何实现的，也感受不到对应的具体实现类的存在。由于 InnerClass 是 private 的，所以，我们如果不看源代码的话，连实现这个接口的具体类的名字都看不到，所以说内部类可以很好的实现隐藏。

2. 内部类拥有外围类的所有元素的访问权限；

```
//外部类
public class Example {
    private String name = "example";

    //内部类
    private class Inner{
        public Inner(){
            System.out.println(name);    // 访问外部类的私有属性
        }
    }

    //外部类方法
    public Inner getInnerInstance() {
        return new Inner();
    }
}

//客户端
public class Client {
    public static void main(String[] args) {
        Example ex = new Example();
        ex.getInnerInstance();
    }
}/* Output:
        example
 *///:~
```

name 这个成员变量是在Example里面定义的私有变量，这个变量在内部类中可以被无条件地访问。

3. 可以实现多重继承

对多重继承而言，可以这样说，接口只是解决了部分问题，而内部类使得多重继承的解决方案变得更加完整。内部类使得Java的继承机制更加完善，是内部类存在的最大理由。Java中的类只能继承一个类，它的多重继承在我们没有学习内部类之前是用接口来实现的。但使用接口有时候有很多不方便的地方，比如，我们实现一个接口就必须实现它里面的所有方法；而内部类可以使我们的类继承多个具体类或抽象类，规避接口的限制性。看下面的例子：

```
//父类Example1
public class Example1 {
    public String name() {
        return "rico";
    }
}

//父类Example2
public class Example2 {
    public int age() {
        return 25;
    }
}

//实现多重继承的效果
public class MainExample {

    //内部类Test1继承类Example1
    private class Test1 extends Example1 {
        public String name() {
            return super.name();
        }
    }

    //内部类Test2继承类Example2
    private class Test2 extends Example2 {
        public int age() {
            return super.age();
        }
    }

    public String name() {
        return new Test1().name();
    }

    public int age() {
        return new Test2().age();
    }

    public static void main(String args[]) {
        MainExample mexam = new MainExample();
        System.out.println("姓名:" + mexam.name());
        System.out.println("年龄:" + mexam.age());
    }
}/* Output:
```

```
        姓名:rico
        年龄:25
 *///:~
```

注意到类 MainExample，在这个类中，包含两个内部类 Test1 和 Test2。其中，类Test1继承了类Example1，类Test2继承了类Example2。这样，类MainExample 就拥有了 类 Example1 和 类Example2 的方法，也就间接地实现了多继承。

4. 可以避免修改接口而实现同一个类中两种同名方法的调用

考虑这样一种情形，一个类要继承一个类，还要实现一个接口，可是它所继承的类和接口里面有两个相同的方法（方法签名一致），那么我们该怎么区分它们呢？这就需要使用内部类了。例如，

```
//Test 所实现的接口
public interface InterfaceTest {
    public void test();
}

//Test 所实现的类
public class MyTest {
    public void test(){
        System.out.println("MyTest");
    }
}

//不使用内部类的情形
public class Test extends MyTest implements InterfaceTest{
    public void test(){
        System.out.println("Test");
    }
}
```

此时，Test中的 test() 方法是属于覆盖 MyTest 的 test() 方法呢，还是实现 InterfaceTest 中的 test() 方法呢？我们怎么能调到 MyTest 这里的方法？显然这是不好区分的。而我们如果用内部类就很好解决这一问题了。看下面代码：

```
//Test 所实现的接口
public interface InterfaceTest {
    public void test();
}

//Test 所实现的类
public class MyTest {
    public void test(){
        System.out.println("MyTest");
    }
}

//使用内部类的情形
public class AnotherTest extends MyTest {

    private class InnerTest implements InterfaceTest {
        @Override
        public void test() {
```

```
            System.out.println("InterfaceTest");
        }
    }

    public InterfaceTest getCallbackReference() {
        return new InnerTest();
    }

    public static void main(String[] args) {
        AnotherTest aTest = new AnotherTest();
        aTest.test(); // 调用类MyTest 的 test() 方法
        aTest.getCallbackReference().test(); // 调用InterfaceTest接口中的
test() 方法
    }
}
```

通过使用内部类来实现接口，就不会与外围类所继承的同名方法冲突了。

3. 内部类种类

在java中，内部类的使用共有两种情况：

1. 在类中定义一个类（成员内部类，静态内部类）
2. 在方法中定义一个类（局部内部类，匿名内部类）

## 成员内部类

1. 定义与原理

成员内部类是最普通的内部类，它是外围类的一个成员，在实际使用中，一般将其可见性设为private。成员内部类是依附于外围类的，所以，只有先创建了外围类对象才能够创建内部类对象。也正是由于这个原因，成员内部类也不能含有 static 的变量和方法，看下面例子：

```
public class Outter {
    private class Inner {

        private final static int x=1;   // OK

        /* compile errors for below declaration
        * "The field x cannot be declared static in a non-static inner type,
        * unless initialized with a constant expression" */

        final static Inner a = new Inner();     // Error

        static Inner a1 = new Inner();     // Error

        static int y;     // Error
    }
}
```

如果上面的代码编译无误, 那么我们就可以直接通过 Outter.Inner.a 拿到内部类Inner的实例。 由于内部类的实例一定要绑定到一个外部类的实例的，所以矛盾。因此，成员内部类不能含有 static 变量/方法。此外，成员内部类与 static 的关系还包括：

- 包含 static final 域，但该域的初始化必须是一个常量表达式；
- 内部类可以继承含有static成员的类。

2. 交互

成员内部类与外部类的交互关系为：

- 成员内部类可以直接访问外部类的所有成员和方法，即使是 private 的；
- 外部类需要通过内部类的对象访问内部类的所有成员变量/方法。

```
//外部类
class Out {
    private int age = 12;
    private String name = "rico";

    //内部类
    class In {
        private String name = "livia";
        public void print() {
            String name = "tom";
            System.out.println(age);
            System.out.println(Out.this.name);
            System.out.println(this.name);
            System.out.println(name);
        }
    }

    // 推荐使用getxxx()来获取成员内部类的对象
    public In getInnerClass(){
        return new In();
    }
}

public class Demo {
    public static void main(String[] args) {

        Out.In in = new Out().new In();   // 片段 1
        in.print();

        //或者采用注释内两种方式访问
        /*
         * 片段 2
        Out out = new Out();

        out.getInnerClass().print();  // 推荐使用外部类getxxx()获取成员内部类对象

        Out.In in = out.new In();
        in.print();

        */
    }
}/* Output:
        12
        rico
        livia
        tom
 *///:~
```

对于代码片段 1 和2，可以用来生成内部类的对象，这种方法存在两个小知识点需要注意：

1. 开头的 Out 是为了标明需要生成的内部类对象在哪个外部类当中；
2. 必须先有外部类的对象才能生成内部类的对象。

因此，成员内部类，外部类和客户端之间的交互关系为：

1. 在成员内部类使用外部类对象时，使用 外部类名.this 来表示外部类对象；
2. 在外部类中使用内部类对象时，需要先进行创建内部类对象；
3. 在客户端创建内部类对象时，需要先创建外部类对象。

特别地，对于成员内部类对象的获取，外部类一般应提供相应的 getxxx() 方法。

3. 私有成员内部类

如果一个成员内部类只希望被外部类操作，那么可以使用 private 将其声明私有内部类。例如，

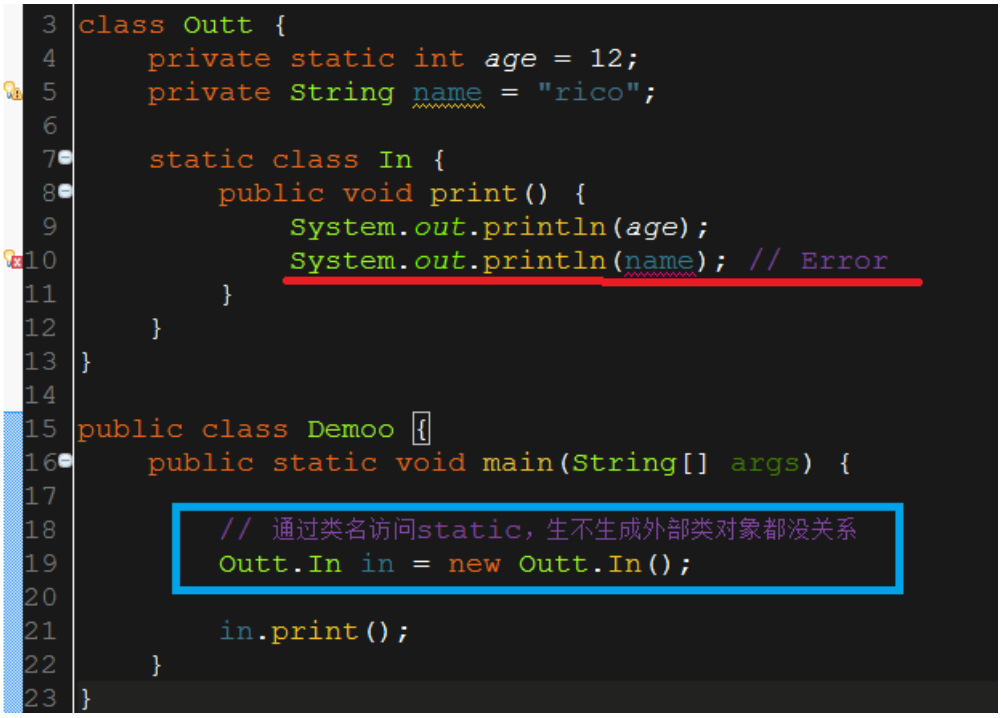
```
class Out {
    private int age = 12;

    private class In {
        public void print() {
            System.out.println(age);
        }
    }
    public void outPrint() {
        new In().print();
    }
}

public class Demo {
    public static void main(String[] args) {

        /*
        * 此方法无效
        Out.In in = new Out().new In();
        in.print();
        */

        Out out = new Out();
        out.outPrint();
    }
}/* Output:
        12
 *///:~
```

在上面的代码中，我们必须在Out类里面生成In类的对象进行操作，而无法再使用Out.In in = new Out().new In() 生成内部类的对象。也就是说，此时的内部类只对外部类是可见的，其他类根本不知道该内部类的存在。

## 静态内部类

1. 定义与原理

静态内部类，就是修饰为 static 的内部类，该内部类对象不依赖于外部类对象，就是说我们可以直接创建内部类对象。看下面例子：



```
class Outt {
    private static int age = 12;
    private String name = "rico";

    static class In {
        public void print() {
            System.out.println(age);
            System.out.println(name); // Error
        }
    }
}

public class Demoo {
    public static void main(String[] args) {

        // 通过类名访问static，生不生成外部类对象都没关系
        Outt.In in = new Outt.In();

        in.print();
    }
}
```

2. 交互

   静态内部类与外部类的交互关系为：

   - 静态内部类可以直接访问外部类的所有静态成员和静态方法，即使是 private 的；
   - 外部类可以通过内部类对象访问内部类的实例成员变量/方法；对于内部类的静态域/方法，外部类可以通过内部类类名访问。

3. 成员内部类和静态内部类的区别

   成员内部类和静态内部类之间的不同点包括：

   - 静态内部类对象的创建不依赖外部类的实例，但成员内部类对象的创建需要依赖外部类的实例；
   - 成员内部类能够访问外部类的静态和非静态成员，静态内部类不能访问外部类的非静态成员；

## 局部内部类

1. 定义与原理

   有这样一种内部类，它是嵌套在方法和作用域内的，对于这个类的使用主要是应用与解决比较复杂的问题，想创建一个类来辅助我们的解决方案，但又不希望这个类是公共可用的，所以就产生了局部内部类。局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，它只能在该方法和属性中被使用，出了该方法和属性就会失效。

```
// 例 1：定义于方法内部
public class Parcel4 {
    public Destination destination(String s) {
        class PDestination implements Destination {
            private String label;

            private PDestination(String whereTo) {
                label = whereTo;
            }

            public String readLabel() {
                return label;
            }
        }
        return new PDestination(s);
```

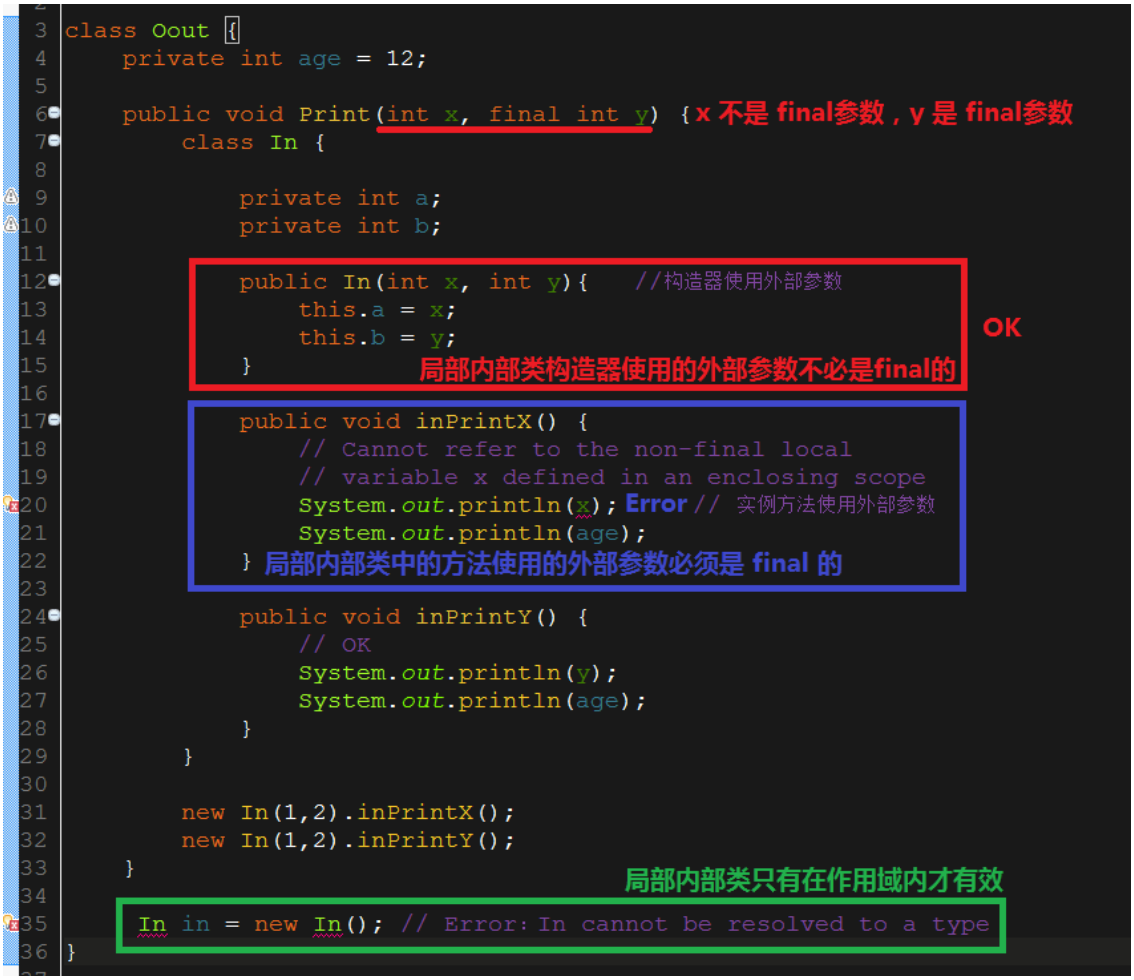
```
    }

    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Destination d = p.destination("Tasmania");
    }
}
```

```
// 例 2：定义于作用域内部
public class Parcel5 {
    private void internalTracking(boolean b) {
        if (b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() {
                    return id;
                }
            }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
        }
    }

    public void track() {
        internalTracking(true);
    }

    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        p.track();
    }
}
```

2. final参数

   对于final参数，若是将引用类型参数声明为final，我们无法在方法中更改参数引用所指向的对象；若是将基本类型参数声明为final，我们可以读参数，但却无法修改参数（这一特性主要用来向局部内部类和匿名内部类传递数据）。

   如果定义一个局部内部类，并且希望它的方法可以直接使用外部定义的数据，那么我们必须将这些数据设为是 final 的；特别地，如果只是局部内部类的构造器需要使用外部参数，那么这些外部参数就没必要设置为 final，例如：




## 匿名内部类

有时候我为了免去给内部类命名，便倾向于使用匿名内部类，因为它没有名字。匿名内部类的使用需要注意以下几个地方：

- 匿名内部类是没有访问修饰符的；
- 匿名内部类是没有构造方法的 (因为匿名内部类连名字都没有)；
- 定义匿名内部类的前提是，内部类必须是继承一个类或者实现接口，格式为 new 父类或者接口() {子类的内容(如函数等)}。也就是说，匿名内部类最终提供给我们的是一个匿名子类的对象，例如：

```
// 例 1
abstract class AbsDemo
{
    abstract void show();
}

public class Outer
{
    int x=3;
    public void function()//可调用函数
    {
        new AbsDwmo()//匿名内部类
        {
            void show()
            {
                System.out.println("x==="+x);
            }
```

```
            void abc()
            {
                System.out.println("haha");
            }
        }.abc();  //匿名内部类调用函数,匿名内部类方法只能调用一次
    }
}
```

```
// 例 2
interface Inner {    //注释后，编译时提示类Inner找不到
    String getName();
}

public class Outer {

    public Inner getInner(final String name, String city) {
        return new Inner() {
            private String nameStr = name;

            public String getName() {
                return nameStr;
            }
        };
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.getInner("Inner", "gz");
        System.out.println(inner.getName());

        System.out.println(inner instanceof Inner);  //匿名内部类实质上是一个匿名子类的对象
    } /* Output:
        Inner
        true
      *///:~
}
```

若匿名内部类 **(匿名内部类没有构造方法)** 需要直接使用其所在的外部类方法的参数时，该形参必须为 final 的；如果匿名内部类没有直接使用其所在的外部类方法的参数时，那么该参数就不必为 final 的，例如：

```
// 情形 1：匿名内部类直接使用其所在的外部类方法的参数 name
public class Outer {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.getInner("Inner", "gz");
        System.out.println(inner.getName());
    }

    public Inner getInner(final String name, String city) {  // 形参 name 被设为 final
        return new Inner() {
            private String nameStr = name;        // OK
```

```
            private String cityStr = city;       // Error: 形参 city 未被设为
final

            public String getName() {
                return nameStr;
            }
        };
    }
}



// 情形 2：匿名内部类没有直接使用其所在的外部类方法的参数
public class Outer {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.getInner("Inner", "gz");
        System.out.println(inner.getName());
    }

    //注意这里的形参city，由于它没有被匿名内部类直接使用，而是被抽象类Inner的构造函数所使
用，所以不必定义为final
    public Inner getInner(String name, String city) {
        return new Inner(name, city) {     // OK，形参 name 和 city 没有被匿名内
部类直接使用
            private String nameStr = name;

            public String getName() {
                return nameStr;
            }
        };
    }
}

abstract class Inner {
    Inner(String name, String city) {
        System.out.println(city);
    }

    abstract String getName();
}
```

从上述代码中可以看到，当匿名内部类直接使用其所在的外部类方法的参数时，那么这些参数必须被设为 final的。为什么呢？本文所引用到的一篇文章是这样解释的：

“这是一个编译器设计的问题，如果你了解java的编译原理的话很容易理解。首先，内部类被编译的时候会生成一个单独的内部类的.class文件，这个文件并不与外部类在同一class文件中。当外部类传的参数被内部类调用时，从java程序的角度来看是直接的调用，例如：

```
public void dosome(final String a,final int b){
  class Dosome{
       public void dosome(){
            System.out.println(a+b)
       }
  };

  Dosome some=new Dosome();
  some.dosome();
}
```

从代码来看，好像是内部类直接调用的a参数和b参数，但是实际上不是，在java编译器编译以后实际的操作代码是:

```
class Outer$Dosome{
  public Dosome(final String a,final int b){
      this.Dosome$a=a;
      this.Dosome$b=b;
  }
  public void dosome(){
      System.out.println(this.Dosome$a+this.Dosome$b);
  }
}
```

从以上代码来看，内部类并不是直接调用方法传进来的参数，而是内部类将传进来的参数通过自己的构造器备份到了自己的内部，自己内部的方法调用的实际是自己的属性而不是外部类方法的参数。这样就很容易理解为什么要用final了，因为两者从外表看起来是同一个东西，实际上却不是这样，**如果内部类改掉了这些参数的值也不可能影响到原参数，然而这样却失去了参数的一致性**，因为从编程人员的角度来看他们是同一个东西，如果编程人员在程序设计的时候在内部类中改掉参数的值，但是外部调用的时候又发现值其实没有被改掉，这就让人非常的难以理解和接受，为了避免这种尴尬的问题存在，所以编译器设计人员把内部类能够使用的参数设定为必须是final来规避这种莫名其妙错误的存在。”

以上关于匿名内部类的每个例子使用的都是默认无参构造函数，下面我们介绍 **带参数构造函数的匿名内部类**:

```
public class Outer {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.getInner("Inner", "gz");
        System.out.println(inner.getName());
    }

    public Inner getInner(final String name, String city) {
        return new Inner(name, city) {   //匿名内部类
            private String nameStr = name;
            public String getName() {
                return nameStr;
            }
        };
    }
}

abstract class Inner {
    Inner(String name, String city) {   // 带有参数的构造函数
```

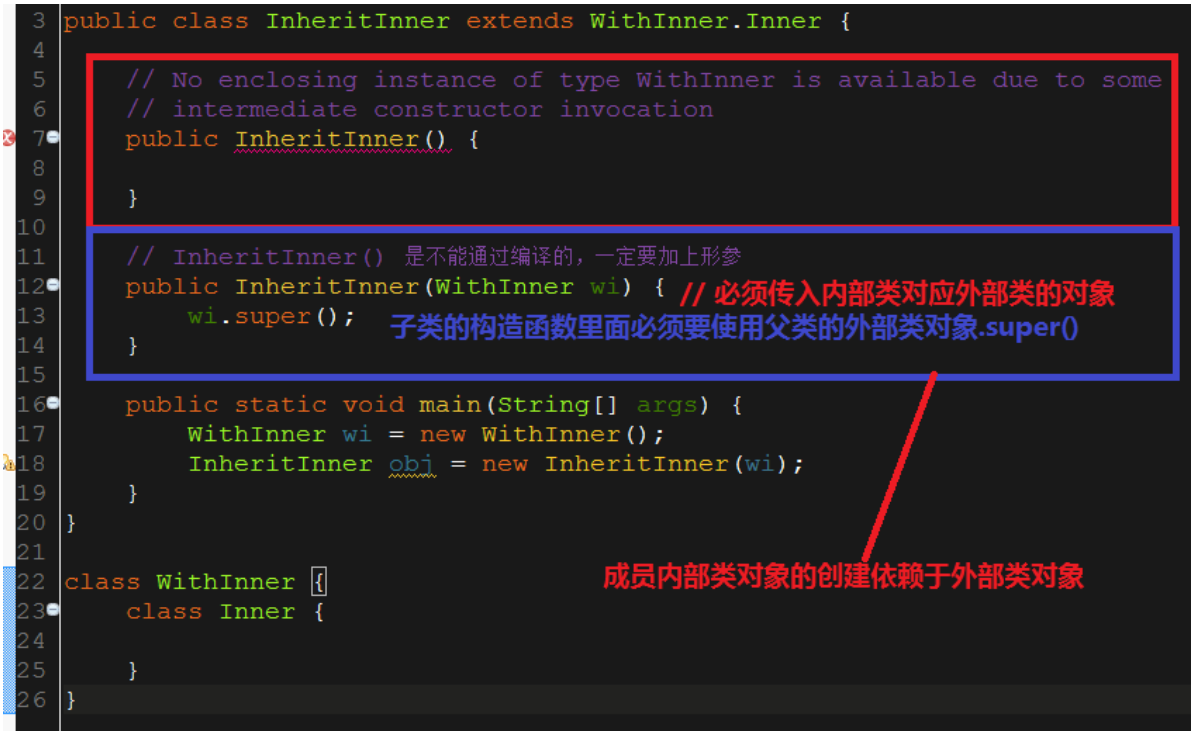
```
        System.out.println(city);
    }

    abstract String getName();
}
```

特别地，**匿名内部类通过实例初始化 *(实例语句块主要用于匿名内部类中)*，可以达到类似构造器的效果，**如下：

```
public class Outer {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.getInner("Inner", "gz");
        System.out.println(inner.getName());
        System.out.println(inner.getProvince());
    }

    public Inner getInner(final String name, final String city) {
        return new Inner() {
            private String nameStr = name;
            private String province;

            // 实例初始化
            {
                if (city.equals("gz")) {
                    province = "gd";
                }else {
                    province = "";
                }
            }

            public String getName() {
                return nameStr;
            }

            public String getProvince() {
                return province;
            }
        };
    }
}
```

## 内部类继承

内部类的继承，是指内部类被继承，普通类 extents 内部类。而这时候代码上要有点特别处理，具体看以下例子：




可以看到，**子类的构造函数里面要使用父类的外部类对象.super() */成员内部类对象的创建依赖于外部类对象/*；而这个外部类对象需要从外面创建并传给形参。**

# 1.11 关键字

## final

## try-finally-return

- 1、不管有没有出现异常，finally块中代码都会执行；
- 2、当try和catch中有return时，finally仍然会执行；无论try里执行了return语句、break语句、还是continue语句，finally语句块还会继续执行；如果执行try和catch时JVM退出（比如System.exit(0)），那么finally不会被执行；
- finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值），所以函数返回值是在finally执行前确定的；
- 【
- 如果try语句里有return，那么代码的行为如下：
  1.如果有返回值，就把返回值保存到局部变量中
  2.执行jsr指令跳到finally语句里执行
  3.执行完finally语句后，返回之前保存在局部变量表里的值
- 】
- 3、当try和finally里都有return时，会忽略try的return，而使用finally的return。
- 4、如果try块中抛出异常，执行finally块时又抛出异常，此时原始异常信息会丢失，只抛出在finally代码块中的异常。
- 实例一：

```
public static int test() {
    int x = 1;
    try {
        x++;
        return x; // 2
    } finally {
        x++;
    }
}
```

- 实例二：

```
private static int test2() {
    try {
        System.out.println("try...");
        return 80;
    } finally {
        System.out.println("finally...");
        return 100; // 100
    }
}
```

# static

- static方法就是没有this的方法。在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法。这实际上正是static方法的主要用途。

## 1)修饰成员方法：静态成员方法

- 在静态方法中不能访问类的非静态成员变量和非静态成员方法；
- 在非静态成员方法中是可以访问静态成员方法/变量的；
- 即使没有显式地声明为static，类的构造器实际上也是静态方法

## 2)修饰成员变量：静态成员变量

- 静态变量和非静态变量的区别是：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。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。
- 静态成员变量并发下不是线程安全的，并且对象是单例的情况下，非静态成员变量也不是线程安全的。
- 怎么保证变量的线程安全?
- 只有一个线程写，其他线程都是读的时候，加volatile；线程既读又写，可以考虑Atomic原子类和线程安全的集合类；或者考虑ThreadLocal

## 3)修饰代码块：静态代码块

- 用来构造静态代码块以优化程序性能。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。

### 4)修饰内部类：静态内部类

- 成员内部类和静态内部类的区别：
  - 1)前者只能拥有非静态成员；后者既可拥有静态成员，又可拥有非静态成员
  - 2)前者持有外部类的的引用，可以访问外部类的静态成员和非静态成员；后者不持有外部类的引用，只能访问外部类的静态成员
  - 3)前者不能脱离外部类而存在；后者可以

### 5)修饰import：静态导包

-

## switch

### switch字符串实现原理

- 对比反编译之后的结果：
- 编译后switch还是基于整数，该整数来自于String的hashCode。
- 先比较字符串的hashCode，因为hashCode相同未必值相同，又再次检查了equals是否相同。
-

### 字节码实现原理（tableswitch / lookupswitch)

- 编译器会使用tableswitch和lookupswitch指令来生成switch语句的编译代码。当switch语句中的case分支的条件值比较稀疏时，tableswitch指令的空间使用率偏低。这种情况下将使用lookupswitch指令来替代。lookupswitch指令的索引表由int类型的键（来源于case语句块后面的数值)与对应的目标语句偏移量所构成。当lookupswitch指令执行时，switch语句的条件值将和索引表中的键进行比较，如果某个键和条件值相符，那么将转移到这个键对应的分支偏移量继续执行，如果没有键值符合，执行将在default分支执行。

## abstract

- 只要含有抽象方法，这个类必须添加abstract关键字，定义为抽象类。
- 只要父类是抽象类,内含抽象方法，那么继承这个类的子类的相对应的方法必须重写。如果不重写，就需要把父类的声明抽象方法再写一遍，留给这个子类的子类去实现。同时将这个子类也定义为抽象类。
- 注意抽象类中可以有抽象方法，也可以有具体实现方法（当然也可以没有)。
- 抽象方法须加abstract关键字，而具体方法不可加
- 只要是抽象类，就不能存在这个类的对象（不可以new一个这个类的对象)。

## this & super

- this
- 自身引用；访问成员变量与方法；调用其他构造方法
- 1. 通过this调用另一个构造方法，用法是this(参数列表)，这个仅在类的构造方法中可以使用
- 2. 函数参数或者函数中的局部变量和成员变量同名的情况下，成员变量被屏蔽，此时要访问成员变量则需要用“this.成员变量名”的方式来引用成员变量。
- 3. 需要引用当前对象时候，直接用this（自身引用)
- super
- 父类引用；访问父类成员变量与方法；调用父类构造方法
- super可以理解为是指向自己超（父)类对象的一个指针，而这个超类指的是离自己最近的一个父类。

- super有三种用法：
- 1.普通的直接引用
- 与this类似，super相当于是指向当前对象的父类，这样就可以用super.xxx来引用父类的成员，如果不冲突的话也可以不加super。
- 2.子类中的成员变量或方法与父类中的成员变量或方法同名时，为了区别，调用父类的成员必须要加super
- 3.调用父类的构造函数

## 访问权限

# 1.12 枚举

## JDK实现

- 实例：

```
public enum Labels0 {

    ENVIRONMENT("环保"), TRAFFIC("交通"), PHONE("手机");

    private String name;

    private Labels0(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

- 编译后生成的字节码反编译：
- 可以清晰地看到枚举被编译后其实就是一个类，该类被声明成 final，说明其不能被继承，同时它继承了 Enum 类。枚举里面的元素被声明成 static final ，另外生成一个静态代码块 static{}，最后还会生成 values 和 valueOf 两个方法。下面以最简单的 Labels 为例，一个一个模块来看。

### Enum 类

- Enum 类是一个抽象类，主要有 name 和 ordinal 两个属性，分别用于表示枚举元素的名称和枚举元素的位置索引，而构造函数传入的两个变量刚好与之对应。
- toString 方法直接返回 name。
- equals 方法直接用 == 比较两个对象。
- hashCode 方法调用的是父类的 hashCode 方法。
- 枚举不支持 clone、finalize 和 readObject 方法。
- compareTo 方法可以看到就是比较 ordinal 的大小。
- valueOf 方法，根据传入的字符串 name 来返回对应的枚举元素。

### 静态代码块的实现

- 在静态代码块中创建对象，对象是单例的！
- 可以看到静态代码块主要完成的工作就是先分别创建 Labels 对象，然后将“ENVIRONMENT”、“TRAFFIC”和“PHONE”字符串作为 name ，按照顺序分别分配位置索引0、1、2作为 ordinal，然后将其值设置给创建的三个 Labels 对象的 name 和 ordinal 属性，此外还会创建一个大小为3的 Labels 数组 ENUM$VALUES，将前面创建出来的 Labels 对象分别赋值给数组。

### values的实现

- 可以看到它是一个静态方法，主要是使用了前面静态代码块中的 Labels 数组 ENUM$VALUES，调用 System.arraycopy 对其进行复制，然后返回该数组。所以通过 Labels.values()[2]就能获取到数组中索引为2的元素。

### valueOf 方法

- 该方法同样是个静态方法，可以看到该方法的实现是间接调用了父类 Enum 类的 valueOf 方法，根据传入的字符串 name 来返回对应的枚举元素，比如可以通过 Labels.valueOf("ENVIRONMENT")获取 Labels.ENVIRONMENT。
- 枚举本质其实也是一个类，而且都会继承java.lang.Enum类，同时还会生成一个静态代码块static{}，并且还会生成 values 和 valueOf 两个方法。而上述的工作都需要由编译器来完成，然后我们就可以像使用我们熟悉的类那样去使用枚举了。
-

## 用enum代替int常量

- 将int枚举常量翻译成可打印的字符串，没有很便利的方法。
- 要遍历一个枚举组中的所有int 枚举常量，甚至获得int枚举组的大小。
- 使用枚举类型的values方法可以获得该枚举类型的数组
- 枚举类型没有可以访问的构造器，是真正的final；是实例受控的，它们是单例的泛型化；本质上是单元素的枚举；提供了编译时的类型安全。
- 单元素的枚举是实现单例的最佳方法！
- 可以在枚举类型中放入这段代码，可以实现String2Enum。
- 注意Operation是枚举类型名。

## 用实例域代替序数

- 这种实现不好，不推荐使用ordinal方法，推荐使用下面这种实现：

## 用EnumSet代替位域

- 位域是将几个常量合并到一个集合中，我们推荐用枚举代替常量，用EnumSet代替集合
    - EnumSet.of(enum1,enum2) -> Set<枚举>

## 用EnumMap代替序数索引

- 将一个枚举类型的值与一个元素（或一组)对应起来，推荐使用EnumMap数据结构
- 如果是两个维度的变化，那么可以使用EnumMap<Enum1,Map<Enum1,元素>>

-

# 1.13 序列化

## JDK序列化（Serizalizable)

- 定义：将实现了Serializable接口（标记型接口)的对象转换成一个字节数组，并可以将该字节数组转为原来的对象。
- ObjectOutputStream 是专门用来输出对象的输出流；
- ObjectOutputStream 将 Java 对象写入 OutputStream。可以使用 ObjectInputStream 读取（重构)对象。

## serialVersionUID

- Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体（类)的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。(InvalidCastException)。
    - 1)如果没有添加serialVersionUID，进行了序列化，而在反序列化的时候，修改了类的结构（添加或删除成员变量，修改成员变量的命名)，此时会报错。
    - 2)如果添加serialVersionUID，进行了序列化，而在反序列化的时候，修改了类的结构（添加或删除成员变量，修改成员变量的命名)，那么可能会恢复部分数据，或者恢复不了数据。
- 如果设置了serialVersionUID并且一致，那么可能会反序列化部分数据；如果没有设置，那么只要属性不同，那么无法反序列化。

## 其他序列化工具

- XML/JSON
- Thrift/Protobuf

## 对象深拷贝与浅拷贝

- 当拷贝一个变量时，原始引用和拷贝的引用指向同一个对象，改变一个引用所指向的对象会对另一个引用产生影响。
- 如果需要创建一个对象的浅拷贝，那么需要调用clone方法。
- Object 类本身不实现接口 Cloneable，直接调用clone会抛出异常。

> 如果要在自己定义类中调用`clone`方法，必须实现`Cloneable`接口（标记型接口)，因为`Object`类中的`clone`方法为`protected`，所以需要自己重写`clone`方法，设置为`public`。

- protected native Object clone() throws CloneNotSupportedException;

```
public class Person implements Cloneable {
    private int age;
    private String  name;
    private Company company;
    @Override
    public Person clone() throws CloneNotSupportedException {
        return (Person) super.clone();
    }
```

- }

```
public class Company implements Cloneable{
    private String name;
```

```
    @Override
public Company clone() throws CloneNotSupportedException {
    return (Company) super.clone();
}
```

- }
- 使用super（即Object)的clone方法只能进行浅拷贝。
- 如果希望实现深拷贝，需要修改实现，比如修改为：

```
@Override
public Person clone() throws CloneNotSupportedException {
    Person person = (Person) super.clone();
    person.setCompany(company.clone()); // 一个新的Company
    return person;
}
```

- 假如说Company中还有持有其他对象的引用，那么Company中也要像Person这样做。
- 可以说：想要深拷贝一个子类，那么它的所有父类都必须可以实现深拷贝。
- 另一种实现对象深拷贝的方式是序列化。
- @Override
  protected Object clone() {
  try {
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  ObjectOutputStream os = new ObjectOutputStream(baos);
  os.writeObject(this);
  os.close();
  ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
  ObjectInputStream in = new ObjectInputStream(bais);
  Object ret = in.readObject();
  in.close();
  return ret;
  }catch(Exception e) {
  e.printStackTrace();
  }
  return null;
  }
-

# 1.14 异常

## Error、Exception

- Error是程序无法处理的错误，它是由JVM产生和抛出的，比如OutOfMemoryError、ThreadDeath等。这些异常发生时，Java虚拟机（JVM)一般会选择线程终止。
- Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常。程序中应当尽可能去处理这些异常。

## 常见RuntimeException

- IllegalArgumentException - 方法的参数无效
- NullPointerException - 试图访问一空对象的变量、方法或空数组的元素
- ArrayIndexOutOfBoundsException - 数组越界访问
- ClassCastException - 类型转换异常
- NumberFormatException 继承IllegalArgumentException，字符串转换为数字时出现。比如int i= Integer.parseInt("ab3");
-

# RuntimeException与非Runtime Exception

- RuntimeException是运行时异常，也称为未检查异常；
- 非RuntimeException 也称为CheckedException 受检异常
- 前者可以不必进行try-catch，后者必须要进行try-catch或者throw。

# 异常包装

- 在catch子句中可以抛出一个异常，这样做的目的是改变异常的类型
- try{
- …
- }catch(SQLException e){
- throw new ServletException(e.getMessage());
- }
- 这样的话ServletException会取代SQLException。
- 有一种更好的方法，可以保存原有异常的信息，将原始异常设置为新的异常的原因
- try{
- …
- }catch(SQLException e){
- Throwable se =  new ServletException(e.getMessage());
- se.initCause(e);
- throw se;
- }
- 当捕获到异常时，可以使用getCause方法来重新得到原始异常
- Throwable e = se.getCause();
- 建议使用这种包装技术，可以抛出系统的高级异常（自己new的），又不会丢失原始异常的细节。
- 早抛出，晚捕获。
-

# 1.15 泛型

- 泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

# 泛型接口/类/方法

# 泛型继承、实现

- 父类使用泛型，子类要么去指定具体类型参数，要么继续使用泛型

# 泛型的约束和局限性

```
- 1)只能使用包装器类型，不能使用基本数据类型；

- 2)运行时类型查询只适用于原始类型，不适用于带类型参数的类型；
```

- if(a instanceof Pair) //error
    - 3)不能创建带有类型参数的泛型类的数组
- Pair [] pairs = new Pair[10];//error
- 只能使用反射来创建泛型数组

```
public static <T extends Comparable> T[] minmax(T… a){
```

- T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(),个数);
- …复制
- }

# 通配符

- ? 未知类型 只可以用于声明时，声明类型或方法参数，不能用于定义时（指定类型参数时）
- List<?> unknownList;
- List<? extends Number> unknownNumberList;
- List<? super Integer> unknownBaseLineIntgerList;
- 对于参数值是未知类型的容器类，只能读取其中元素，不能向其中添加元素， 因为，其类型是未知，所以编译器无法识别添加元素的类型和容器的类型是否兼容，唯一的例外是null。
- 通配符类型 List 与原始类型 List 和具体类型 List都不相同，List表示这个list内的每个元素的类型都相同，但是这种类型具体是什么我们却不知道。注意，List和List可不相同，由于Object是最高层的超类，List表示元素可以是任何类型的对象，但是List可不是这个意思（未知类型)。
-

## extends 指定类型必为自身或其子类

- List<? extends Fruit>
- 这个引用变量如果作为参数，哪些引用可以传入?
- 本身及其子类
- 以及含有通配符及extends的本身及子类
- 不可传入只含通配符不含extends+指定类 的引用
- 或者extends的不是指定类及其子类，而是其父类
- // Number "extends" Number (in this context)
- List<? extends Number> foo3 = new ArrayList<? extends Number>();
- // Integer extends Number
- List<? extends Number> foo3 = new ArrayList<? extends Integer>();
- // Double extends Number
- List<? extends Number> foo3 = new ArrayList<? extends Double>();
- 如果实现了多个接口，可以使用&来将接口隔开
- T extends Comparable & Serializable
- List<? extends Number> list = new ArrayList();
    - list.add(new Integer(1)); //error

- list.add(new Float(1.2f)); //error

# super 指定类型必为自身或其父类

- 不能同时声明泛型通配符申明上界和下界

# PECS（读extends，写super)

- producer-extends, consumer-super.
- produce是指参数是producer，consumer是指参数是consumer。
- 要往泛型类写数据时，用extends；
- 要从泛型类读数据时，用super；
- 既要取又要写，就不用通配符（即extends与super都不用)比如List。
- 如果参数化类型表示一个T生产者，就是<? extends T>；如果它表示一个T消费者，就使用<? super E>。
- Stack的pushAll的参数产生E实例供Stack使用，因此参数类型为Iterable<? extends E>。
- popAll的参数提供Stack消费E实例，因此参数类型为Collection<？ super E>。

```
public void pushAll(Iterable<? extends E> src) {
```

- for (E e : src)
- push(e);
- }

```
public void popAll(Collection<? super E> dst) {
```

- while (!isEmpty())
- dst.add(pop());
- }
- 在调用pushAll方法时生产了E 实例（produces E instances)，在调用popAll方法时dst消费了E 实例（consumes E instances)。Naftalin与Wadler将PECS称为Get and Put Principle。
- Collections#copy

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");

    if (srcSize < COPY_THRESHOLD ||
        (src instanceof RandomAccess && dest instanceof RandomAccess)) {
        for (int i=0; i<srcSize; i++)
            dest.set(i, src.get(i));
    } else {
        ListIterator<? super T> di=dest.listIterator();
        ListIterator<? extends T> si=src.listIterator();
        for (int i=0; i<srcSize; i++) {
            di.next();
            di.set(si.next());
        }
    }
}
```

-

## 泛型擦除（编译时擦除）

- 编译器生成的bytecode是不包含泛型信息的，泛型类型信息将在编译处理是被擦除，这个过程即泛型擦除。
- 擦除类型变量，并替换为限定类型（无限定的变量用Object)。
- 比如 T extends Comparable
- 那么下面所有出现T的地方都会被替换为Comparable
- 如果调用时指定某个类，比如 Pair pair = new Pair<>();
- 那么Pair 中所有的T都替换为String
- 泛型擦除带来的问题：
    - 1)无法使用具有不同类型参数的泛型进行方法重载

```
public void test(List<String> ls) {
```

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

```
public void test(List<Integer> li) {
```

- System.ut.println("Integer");
- } // 编译出错
- 或者

```
public interface Builder<K,V> {
```

- void add(List keyList);
- void add(List valueList);
- }
    - 2)泛型类的静态变量是共享的
- 另外，因为Java泛型的擦除并不是对所有使用泛型的地方都会擦除的，部分地方会保留泛型信息，在运行时可以获得类型参数。
-

# 1.16 IO

## Unix IO模型

- 异步I/O 是指用户程序发起IO请求后，不等待数据，同时操作系统内核负责I/O操作把数据从内核拷贝到用户程序的缓冲区后通知应用程序。数据拷贝是由操作系统内核完成，用户程序从一开始就没有等待数据，发起请求后不参与任何IO操作，等内核通知完成。
- 同步I/O 就是非异步IO的情况，也就是用户程序要参与把数据拷贝到程序缓冲区（例如java的InputStream读字节流过程)。
- 同步IO里的非阻塞 是指用户程序发起IO操作请求后不等待数据，而是调用会立即返回一个标志信息告知条件不满足，数据未准备好，从而用户请求程序继续执行其它任务。执行完其它任务，用户程序会主动轮询查看IO操作条件是否满足，如果满足，则用户程序亲自参与拷贝数据动作。

- Unix IO模型的语境下，同步和异步的区别在于数据拷贝阶段是否需要完全由操作系统处理。阻塞和非阻塞操作是针对发起IO请求操作后是否有立刻返回一个标志信息而不让请求线程等

待。

# BIO NIO AIO介绍

- BIO：同步阻塞，每个客户端的连接会对应服务器的一个线程
- NIO：同步非阻塞，多路复用器轮询客户端的请求，每个客户端的IO请求会对应服务器的一个线程
- AIO： 异步非阻塞，客户端的IO请求由OS完成后再通知服务器启动线程处理（需要OS支持）
- 1、进程向操作系统请求数据
- 2、操作系统把外部数据加载到内核的缓冲区中，
- 3、操作系统把内核的缓冲区拷贝到进程的缓冲区
- 4、进程获得数据完成自己的功能
- Java NIO属于同步非阻塞IO，即IO多路复用，单个线程可以支持多个IO
- 即询问时从IO没有完毕时直接阻塞，变成了立即返回一个是否完成IO的信号。
- 异步IO就是指AIO，AIO需要操作系统支持。
-

# Java BIO 使用

- Server

```
public class ChatServer {
```

- ServerSocket ss = null;
- boolean started = false;
- ArrayList clients = new ArrayList();

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

- new ChatServer().start();
- }

```
    public void start() {
```

- try {
  - ss = new ServerSocket(6666);
- started = true;
- } catch (BindException e) {
- System.out.println("端口使用中...."); // 用于处理两次启动Server端
- System.out.println("请重新运行服务器");
  - System.exit(-1);
- } catch (IOException e) {
- System.out.println("服务器启动失败");
- e.printStackTrace();
- }
- try {
- while (started) {
- Socket s = ss.accept();

- Client c = new Client(s);
- clients.add(c);
- c.transmitToAll(c.name + "进入了聊天室");
- new Thread(c).start();
- }
- } catch (IOException e) {
- e.printStackTrace();
- } finally { // 主方法结束时应该关闭服务器ServerSocket
- if (ss != null)
- try {
- ss.close();
- } catch (IOException e) {
- e.printStackTrace();
- }
- }
- }
- class Client implements Runnable { // 包装给一个单独的客户端的线程类，应该保留自己的连接Socket和流
- // 保留连接一般使用构造方法，将连接传入
- // 一个客户端就new 一个Client 连接

```
        private Socket s = null;
```

```
        private DataInputStream dis = null;
```

```
        private DataOutputStream dos = null;// 每个客户端的线程都有各自的输入输出流，输入流用于读来自当前客户端的数据，输出流用于保存当前客户端的流。
```

```
        private boolean Connected = false;// 每个客户端都有一个开始结束的标志
```

```
        private String name;
```

- Client(Socket s) { // new 一个Client对象时，要打开Socket和DataInputStream流
- this.s = s;
- try {
- dis = new DataInputStream(s.getInputStream());
- dos = new DataOutputStream(s.getOutputStream());
- Connected = true;
- this.name = dis.readUTF();
- } catch (IOException e) {
- e.printStackTrace();
- }
- }
- // 如何实现一个客户端与其他客户端的通信?

- // 可以考虑在每连到一个客户端就保存与其的连接Socket，当要发送给其他客户端信息时，遍历一遍所有其他客户端

```
        public void run() {
```

- try {
- while (Connected) {
- String read = dis.readUTF();
- if (read.equals("EXIT")) {
- Connected = false;
- transmitToAll(this.name + "已退出");
- continue;
- } else if (read.startsWith("@")) {
    - String[] msg = read.substring(1).split(":");
- transmitToPerson(msg[0], msg[1]);
- continue;
- }
- transmitToAll(this.name+":"+read);
- }
    - } catch (EOFException e1) {
- System.out.println("Client closed");
    - } catch (IOException e1) {
- e1.printStackTrace();
- } finally { // 关闭资源应该放在finally中
- try {
- CloseUtil.close(dis, dos);
- if (s != null)
- s.close();
    - } catch (IOException e1) {
- e1.printStackTrace();
- }
- }
- }

```
        /**
```

-     - 将消息发送给所有人
-
    -
-     - @param read
- */

```
        public void transmitToAll(String read) {
```

- for (int i = 0; i < clients.size(); i++) {
- Client c = clients.get(i);
- if (c.Connected == true)
- c.send(read); // 调用每个客户端线程的send方法，一个对象的输出流与对应的客户端连接dos -->
- // Client
- }
- }

```
        /**
```

- - 将消息发送给某个人，私聊
- -
- - @param read
- - @param clientName
- */

```
        public void transmitToPerson(String clientName, String read) {
```

- boolean isFind = false;
- for (int i = 0; i < clients.size(); i++) {
- Client client = clients.get(i);
- if (client.name.equals(clientName)) {
- client.send(this.name+":"+read);
- isFind = true;
- }
- }
- send(this.name+":"+read + (isFind ? "" : "\n抱歉，没有找到此用户"));
- }

```
        public void send(String str) {// 在哪里出错就在哪里捕获
```

- try {
- dos.writeUTF(str);
- } catch (SocketException e) {
- this.Connected = false;
- clients.remove(this);
- } catch (IOException e) {
- e.printStackTrace();
- }
- }
- }
- }
-
- Client（一个线程用于读取，一个线程用于发送)

```
public class ChatClient extends Frame {
```

- Socket s = null; //将某个对象使得在一个类的各个方法可用，将该对象设置为整个类的成员变量
- DataOutputStream dos = null;//在多个方法中都要使用
- DataInputStream dis = null;
- TextField tfText = new TextField(); // 设置为成员变量方便其他类进行访问
- TextArea taContent = new TextArea();
- boolean started = false;
- Thread recv = null;
-
- ChatClient(String name, int x, int y, int w, int h) {
- super(name);
- this.setBounds(x, y, w, h);
- this.setLayout(new BorderLayout());
- this.addWindowListener(new MonitorWindow());
- taContent.setEditable(false);
- this.add(tfText, BorderLayout.SOUTH);
- this.add(taContent, BorderLayout.NORTH);
- tfText.addActionListener(new MonitorText());//对于文本框的监视器必须添加在某个文本框上，只有窗口监视器才能添加到Frame上
- this.pack();
- this.setVisible(true); // 必须放在最后一行，否则之下的组件无法显示
- connect();
- ClientNameDialog dialog = new ClientNameDialog(this,"姓名提示框",true);
- }
-

```
    private class ClientNameDialog extends JDialog implements
ActionListener{
```

- JLabel jl = null;
- JTextField jf = null;
- JButton jb = null;
-
- ClientNameDialog(Frame  owner,String title,boolean model){
- super(owner,title,model);
- this.setLayout(new BorderLayout());
- this.setBounds(300, 300, 200, 150);
- jl = new JLabel("请输入您的姓名或昵称:");
- jf = new JTextField();
- jb = new JButton("确定");
- jb.addActionListener(this);
- this.addWindowListener(new WindowAdapter(){

```
                public void windowClosing(WindowEvent arg0) {
```

- setVisible(false);
- System.exit(0);
- }
- });
- this.add(jl,BorderLayout.NORTH);
- this.add(jf,BorderLayout.CENTER);
- this.add(jb, BorderLayout.SOUTH);

- this.setVisible(true);
- }

```
        public void actionPerformed(ActionEvent e) {
```

- String name = "";
- name = jf.getText();
- if((name == null || name.equals(""))){
- JOptionPane.showMessageDialog(this, "姓名不可为空!");
- return;
- }
- this.setVisible(false);
- send(name);
- JOptionPane.showMessageDialog(this, "欢迎您,"+name);
- launchThread();
- }
-
- }

```
    private class MonitorWindow extends WindowAdapter {
```

```
        public void windowClosing(WindowEvent e) {
```

- setVisible(false);
- disConnect();
- System.exit(0);
- }
- }
-

```
    private class MonitorText implements ActionListener {
```

- String str = null;
-

```
        public void actionPerformed(ActionEvent e) {
```

-
- str = tfText.getText().trim();//注意这是内部类，要找到事件源对象直接引用外部类的TextField即可，不需要getSource(平行类可用)
- tfText.setText("");  //trim可以去掉开头和结尾的空格
- send(str);
- }
- }
-

```
    public void send(String str){//为发送数据单独建立一个方法
```

- try{

- dos.writeUTF(str);
- dos.flush();
  - }catch(IOException e1){
- e1.printStackTrace();
- }
- }
-

```
    public void connect(){ //应为连接单独建立一个方法
```

- try{
  - s = new Socket("localhost",6666);
- dos = new DataOutputStream(s.getOutputStream());//一连接就打开输出流
- dis = new DataInputStream(s.getInputStream());  //一连接就打开输入流
- started = true;
- }catch(IOException e){
- e.printStackTrace();
- }
- }
-

```
    public void launchThread(){
```

- recv = new Thread(new Receive());
- recv.start();
- }
-

```
    public void disConnect() {
```

- try{
- dos.writeUTF("EXIT");
- started = false;
- //加入到主线程，会等待子线程执行完毕，才会执行下面的语句。这就避免了在读数据的时候将流切断，但是在这里是无效的。但是将线程停止应该先考虑使用join方法
- CloseUtil.close(dis,dos);
- s.close();
- } catch (IOException e) {
- e.printStackTrace();
- }
- }
-

```
    private class Receive implements Runnable { //同样原因 readUTF是阻塞式的，处于死循环中，不能执行其他语句，所以为其单独设置一个线程
```

-

```
        public void run(){
```

- String str = null;
- try{
- while(started){
- str = dis.readUTF();　　//如果在阻塞状态，程序被关闭，那么一定会报错SocketException。关闭了Socket之后还在调用readUTF方法
- taContent.setText(taContent.getText()+str+'\n');//解决方法是在关闭程序的同时停止线程，不再读取
- (如果使用JTextArea可以使用append方法)
- }
- }catch (SocketException e){　　//将SocketException视为退出。但这种想法是不好的，将异常视为程序正常的一部分
- System.out.println("Client has quitted!");
- }catch (EOFException e){
- System.out.println("Client has quitted!");
- }catch(IOException e){
- e.printStackTrace();
- }
- }
- }

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

- new ChatClient("Client", 200, 200, 300, 200);
- }
- }

# Java NIO 使用

- 传统的IO操作面向数据流，意味着每次从流中读一个或多个字节，直至完成，数据没有被缓存在任何地方。
- NIO操作面向缓冲区，数据从Channel读取到Buffer缓冲区，随后在Buffer中处理数据。
- BIO中的accept是没有客户端连接时阻塞，NIO的accept是没有客户端连接时立即返回。
- NIO的三个重要组件：Buffer、Channel、Selector。
- Buffer是用于容纳数据的缓冲区，Channel是与IO设备之间的连接，类似于流。
- 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。
- Selector是Channel的多路复用器。

## Buffer（缓冲区）

- 

- clear 是将position置为0，limit置为capacity；
- flip是将limit置为position，position置为0；

### MappedByteBuffer（对应OS中的内存映射文件）

- ByteBuffer有两种模式:直接/间接。间接模式就是HeapByteBuffer,即操作堆内存 (byte[])。
- 但是内存毕竟有限,如果我要发送一个1G的文件怎么办?不可能真的去分配1G的内存.这时就必须使用"直接"模式,即 MappedByteBuffer。
- OS中内存映射文件是将一个文件映射为虚拟内存（文件没有真正加载到内存，只是作为虚存），不需要使用文件系统调用来读写数据，而是直接读写内存。

- Java中是使用MappedByteBuffer来将文件映射为内存的。通常可以映射整个文件，如果文件比较大的话可以分段进行映射，只要指定文件的那个部分就可以。
- 优点：减少一次数据拷贝
- 之前是 进程空间<->内核的IO缓冲区<->文件
- 现在是 进程空间<->文件
- MappedByteBuffer可以使用FileChannel.map方法获取。
- 它有更多的优点：
- a. 读取快
- b. 写入快
- c. 随机读写

- MappedByteBuffer使用虚拟内存，因此分配(map)的内存大小不受JVM的-Xmx参数限制，但是也是有大小限制的。
- 那么可用堆外内存到底是多少？，即默认堆外内存有多大：
- ① 如果我们没有通过-XX:MaxDirectMemorySize来指定最大的堆外内存。则
- ② 如果我们没通过-Dsun.nio.MaxDirectMemorySize指定了这个属性，且它不等于-1。则
- ③ 那么最大堆外内存的值来自于directMemory = Runtime.getRuntime().maxMemory()，这是一个native方法。
- 在我们使用CMS GC的情况下的实现如下：其实是新生代的最大值-一个survivor的大小+老生代的最大值，也就是我们设置的-Xmx的值里除去一个survivor的大小就是默认的堆外内存的大小了。
- 如果当文件过大，内存不足时，可以通过position参数重新map文件后面的内容。
- MappedByteBuffer在处理大文件时的确性能很高，但也存在一些问题，如内存占用、文件关闭不确定，被其打开的文件只有在垃圾回收的才会被关闭，而且这个时间点是不确定的。
-

## DirectByteBuffer（堆外内存)

- DirectByteBuffer继承自MappedByteBuffer，它们都是使用的堆外内存，不受JVM堆大小的限制，只是前者仅仅是分配内存，后者是将文件映射到内存中。
- 可以通过ByteBuf.allocateDirect方法获取。
- 堆外内存的特点（大对象；加快内存拷贝；减轻GC压力)
  - 对于大内存有良好的伸缩性（支持分配大块内存)
  - 对垃圾回收停顿的改善可以明显感觉到（堆外内存，减少GC对堆内存回收的压力)
  - 在进程间可以共享，减少虚拟机间的复制，加快复制速度（减少堆内内存拷贝到堆外内存的过程)
  - 还可以使用 池+堆外内存 的组合方式，来对生命周期较短，但涉及到I/O操作的对象进行堆外内存的再使用。( Netty中就使用了该方式 )
- 堆外内存的一些问题
  - 1)堆外内存回收问题（不手工回收会导致内存溢出，手工回收就失去了Java的优势)；
  - 2. 数据结构变得有些别扭。要么就是需要一个简单的数据结构以便于直接映射到堆外内存，要么就使用复杂的数据结构并序列化及反序列化到内存中。很明显使用序列化的话会比较头疼且存在性能瓶颈。使用序列化比使用堆对象的性能还差。
-

## 堆外内存的释放

- java.nio.DirectByteBuffer对象在创建过程中会先通过Unsafe接口直接通过os::malloc来分配内存，然后将内存的起始地址和大小存到java.nio.DirectByteBuffer对象里，这样就可以直接操作这些内存。这些内存只有在DirectByteBuffer回收掉之后才有机会被回收，因此如果这些对象大部分都移到了old，但是一直没有触发CMS GC或者Full GC，那么悲剧将会发生，因为

你的物理内存被他们耗尽了，因此为了避免这种悲剧的发生，通过-XX:MaxDirectMemorySize来指定最大的堆外内存大小，当使用达到了阈值的时候将调用System.gc来做一次full gc，以此来回收掉没有被使用的堆外内存。

- GC方式：
- 存在于堆内的DirectByteBuffer对象很小，只存着基地址和大小等几个属性，和一个Cleaner，但它代表着后面所分配的一大段内存，是所谓的冰山对象。通过前面说的Cleaner，堆内的DirectByteBuffer对象被GC时，它背后的堆外内存也会被回收。
- 当新生代满了，就会发生minor gc；如果此时对象还没失效，就不会被回收；撑过几次minorgc后，对象被迁移到老生代；当老生代也满了，就会发生full gc。
- 这里可以看到一种尴尬的情况，因为DirectByteBuffer本身的个头很小，只要熬过了minor gc，即使已经失效了也能在老生代里舒服的呆着，不容易把老生代撑爆触发full gc，如果没有别的大块头进入老生代触发full gc，就一直在那耗着，占着一大片堆外内存不释放。
- 这时，就只能靠前面提到的申请额度超限时触发的System.gc()来救场了。但这道最后的保险其实也不很好，首先它会中断整个进程，然后它让当前线程睡了整整一百毫秒，而且如果gc没在一百毫秒内完成，它仍然会无情的抛出OOM异常。
- 那为什么System.gc()会释放DirectByteBuffer呢?
- 每个DirectByteBuffer关联着其对应的Cleaner，Cleaner是PhantomReference的子类，虚引用主要被用来跟踪对象被垃圾回收的状态，通过查看ReferenceQueue中是否包含对象所对应的虚引用来判断它是否即将被垃圾回收。
- 当GC时发现DirectByteBuffer除了PhantomReference外已不可达，就会把它放进Reference类pending list静态变量里。然后另有一条ReferenceHandler线程，名字叫"Reference Handler"的，关注着这个pending list，如果看到有对象类型是Cleaner，就会执行它的clean()，其他类型就放入应用构造Reference时传入的ReferenceQueue中，这样应用的代码可以从Queue里拖出这些理论上已死的对象，做爱做的事情——这是一种比finalizer更轻量更好的机制。

- 手工方式：
- 如果想立即释放掉一个MappedByteBuffer/DirectByteBuffer，因为JDK没有提供公开API，只能使用反射的方法去unmap；
- 或者使用Cleaner的clean方法。

```
public static void main(String[] args) {
    try {
        File f = File.createTempFile("Test", null);
        f.deleteOnExit();
        RandomAccessFile file = new RandomAccessFile(f, "rw");
        file.setLength(1024);
        FileChannel channel = file.getChannel();
        MappedByteBuffer buffer = channel.map(
                FileChannel.MapMode.READ_WRITE, 0, 1024);
        channel.close();
        file.close();
        // 手动unmap
        Method m = FileChannelImpl.class.getDeclaredMethod("unmap",
                MappedByteBuffer.class);
        m.setAccessible(true);
        m.invoke(FileChannelImpl.class, buffer);
        if (f.delete())
            System.out.println("Temporary file deleted: " + f);
        else
            System.err.println("Not yet deleted: " + f);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
```

```
}
```

- 

## Channel（通道）

- Channel与IO设备的连接，与Stream是平级的概念。

### 流与通道的区别

- 1、流是单向的，通道是双向的，可读可写。
- 2、流读写是阻塞的，通道可以异步读写。
- 3、流中的数据可以选择性的先读到缓存中，通道的数据总是要先读到一个缓存中，或从缓存中写入

- 注意，FileChannel 不能设置为非阻塞模式。

### 分散与聚集

- 分散（scatter)从Channel中读取是指在读操作时将读取的数据写入多个buffer中。因此，Channel将从Channel中读取的数据“分散（scatter)”到多个Buffer中。
- 聚集（gather)写入Channel是指在写操作时将多个buffer的数据写入同一个Channel，因此，Channel 将多个Buffer中的数据“聚集（gather)”后发送到Channel。

### Pipe

```
public class PipeTest {
    public static void main(String[] args) {
        Pipe pipe = null;
        ExecutorService exec = Executors.newFixedThreadPool(2);
        try {
            pipe = Pipe.open();
            final Pipe pipeTemp = pipe;

            exec.submit(new Callable<Object>() {
                @Override
                public Object call() throws Exception {
                    Pipe.SinkChannel sinkChannel = pipeTemp.sink();//向通道中
写数据
                    while (true) {
                        TimeUnit.SECONDS.sleep(1);
                        String newData = "Pipe Test At Time " +
System.currentTimeMillis();
                        ByteBuffer buf = ByteBuffer.allocate(1024);
                        buf.clear();
                        buf.put(newData.getBytes());
                        buf.flip();

                        while (buf.hasRemaining()) {
                            System.out.println(buf);
                            sinkChannel.write(buf);
                        }
                    }
                }
            });
```

```
            exec.submit(new Callable<Object>() {
                @Override
                public Object call() throws Exception {
                    Pipe.SourceChannel sourceChannel = pipeTemp.source();//向
通道中读数据
                    while (true) {
                        TimeUnit.SECONDS.sleep(1);
                        ByteBuffer buf = ByteBuffer.allocate(1024);
                        buf.clear();
                        int bytesRead = sourceChannel.read(buf);
                        System.out.println("bytesRead=" + bytesRead);
                        while (bytesRead > 0) {
                            buf.flip();
                            byte b[] = new byte[bytesRead];
                            int i = 0;
                            while (buf.hasRemaining()) {
                                b[i] = buf.get();
                                System.out.printf("%X", b[i]);
                                i++;
                            }
                            String s = new String(b);
                            System.out.println("================||" + s);
                            bytesRead = sourceChannel.read(buf);
                        }
                    }
                }
            });
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            exec.shutdown();
        }
    }
}
```

- 

## FileChannel与文件锁

- 在通道中我们可以对文件或者部分文件进行上锁。上锁和我们了解的线程锁差不多，都是为了保证数据的一致性。在文件通道FileChannel中可以对文件进行上锁，通过FileLock可以对文件进行锁的释放。
- 文件加锁是建立在文件通道（FileChannel)之上的，套接字通道（SockeChannel)不考虑文件加锁，因为它是不共享的。它对文件加锁有两种方式：
- ①lock
- ②tryLock
- 两种加锁方式默认都是对整个文件加锁，如果自己配置的话就可以控制加锁的文件范围：position是加锁的开始位置，size是加锁长度，shared是用于控制该锁是共享的还是独占的。
- lock是阻塞式的，当有进程对锁进行读取时会等待锁的释放，在此期间它会一直等待；tryLock是非阻塞式的，它尝试获得锁，如果这个锁不能获得，那么它会立即返回。
- release可以释放锁。
- 在一个进程中在锁没有释放之前是无法再次获得锁的
- 在java的NIO中，通道包下面有一个FileLock类，它主要是对文件锁工具的一个描述。在上一小节中对文件的锁获取其实是FileChannel获取的（lock与trylock是FileChannel的方法），它们返回一个FileLock对象。这个类的核心方法有如下这些：

- boolean isShared() :判断锁是否为共享类型
- abstract boolean isValid()： 判断锁是否有效
- boolean overlaps()：判断此锁定是否与给定的锁定区域重叠
- long position()：返回文件内锁定区域中第一个字节的位置。
- abstract void release()： 释放锁
- long size()： 返回锁定区域的大小，以字节为单位
- 在文件锁中有3种方式可以释放文件锁：①锁类释放锁，调用FileLock的release方法； ②通道类关闭通道，调用FileChannel的close方法；③jvm虚拟机会在特定情况释放锁。

- 锁类型（独占式和共享式)
- 我们先区分一下在文件锁中两种锁的区别：①独占式的锁就想我们上面测试的那样，只要有一个进程获取了独占锁，那么别的进程只能等待。②共享锁在一个进程获取的情况下，别的进程还是可以读取被锁定的文件，但是别的进程不能写只能读。
-

## Selector （Channel的多路复用器)

- Selector可以用单线程去管理多个Channel（多个连接)。
- 放在网络编程的环境下：Selector使用单线程，轮询客户端对应的Channel的请求，如果某个Channel需要进行IO，那么分配一个线程去执行IO操作。
- Selector可以去监听的请求有以下几类：
  - 1、connect：客户端连接服务端事件，对应值为SelectionKey.OPCONNECT(8)
  - 2、accept：服务端接收客户端连接事件，对应值为SelectionKey.OPACCEPT(16)
  - 3、read：读事件，对应值为SelectionKey.OPREAD(1)
  - 4、write：写事件，对应值为SelectionKey.OPWRITE(4)
- 每次请求到达服务器，都是从connect开始，connect成功后，服务端开始准备accept，准备就绪，开始读数据，并处理，最后写回数据返回。
- SelectionKey是一个复合事件，绑定到某个selector对应的某个channel上，可能是多个事件的复合或单一事件。

# Java NIO 实例（文件上传)

- 服务器主线程先创建Socket，并注册到selector，然后轮询selector。
  - 1)如果有客户端需要进行连接，那么selector返回ACCEPT事件，主线程建立连接（accept)，并将该客户端连接注册到selector，结束，继续轮询selector等待下一个客户端事件；
  - 2)如果有已连接的客户端需要进行读写，那么selector返回READ/WRITE事件，主线程将该请求交给IO线程池中的某个线程执行操作，结束，继续轮询selector等待下一个客户端事件。

## 服务器

```
public class NIOTCPServer {
    private ServerSocketChannel serverSocketChannel;
    private final String FILE_PATH = "E:/uploads/";
    private AtomicInteger i;
    private final String RESPONSE_MSG = "服务器接收数据成功";
    private Selector selector;
    private ExecutorService acceptPool;
    private ExecutorService readPool;

    public NIOTCPServer() {
```

```
        try {
            serverSocketChannel = ServerSocketChannel.open();
            //切换为非阻塞模式
            serverSocketChannel.configureBlocking(false);
            serverSocketChannel.bind(new InetSocketAddress(9000));
            //获得选择器
            selector = Selector.open();
            //将channel注册到selector上
            //第二个参数是选择键，用于说明selector监控channel的状态
            //可能的取值：SelectionKey.OP_READ OP_WRITE OP_CONNECT OP_ACCEPT

            //监控的是channel的接收状态
            serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
            acceptPool = new ThreadPoolExecutor(50, 100, 1000,
TimeUnit.MILLISECONDS, new LinkedBlockingDeque<>());
            readPool = new ThreadPoolExecutor(50, 100, 1000,
TimeUnit.MILLISECONDS, new LinkedBlockingDeque<>());
            i = new AtomicInteger(0);
            System.out.println("服务器启动");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void receive() {
        try {
            //如果有一个及以上的客户端的数据准备就绪
            while (selector.select() > 0) {
                //获取当前选择器中所有注册的监听事件
                for (Iterator<SelectionKey> it =
selector.selectedKeys().iterator(); it.hasNext(); ) {
                    SelectionKey key = it.next();
                    //如果"接收"事件已就绪
                    if (key.isAcceptable()) {
                        //交由接收事件的处理器处理
                        acceptPool.submit(new ReceiveEventHander());
                    } else if (key.isReadable()) {
                        //如果"读取"事件已就绪
                        //交由读取事件的处理器处理
                        readPool.submit(new ReadEventHandler((SocketChannel)
key.channel()));
                    }
                    //处理完毕后，需要取消当前的选择键
                    it.remove();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    class ReceiveEventHander implements Runnable {

        public ReceiveEventHander() {
        }

        @Override
```

```
        public void run() {
            SocketChannel client = null;
            try {
                client = serverSocketChannel.accept();
                // 接收的客户端也要切换为非阻塞模式
                client.configureBlocking(false);
                // 监控客户端的读操作是否就绪
                client.register(selector, SelectionKey.OP_READ);
                System.out.println("服务器连接客户端:" + client.toString());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    class ReadEventHandler implements Runnable {
        private ByteBuffer buf;
        private SocketChannel client;

        public ReadEventHandler(SocketChannel client) {
            this.client = client;
            buf = ByteBuffer.allocate(1024);
        }

        @Override
        public void run() {

            FileChannel fileChannel = null;
            try {
                int index = 0;
                synchronized (client) {
                    while (client.read(buf) != -1) {
                        if (fileChannel == null) {
                            index = i.getAndIncrement();
                            fileChannel =
FileChannel.open(Paths.get(FILE_PATH, index + ".jpeg"),
StandardOpenOption.WRITE, StandardOpenOption.CREATE);
                        }
                        buf.flip();
                        fileChannel.write(buf);
                        buf.clear();
                    }
                }
                if (fileChannel != null) {
                    fileChannel.close();
                    System.out.println("服务器写来自客户端" + client + " 文件" +
index + " 完毕");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
    }

    public static void main(String[] args) {
        NIOTCPServer server = new NIOTCPServer();
        server.receive();
```

```
    }
}
```

## 客户端

```
public class NIOTCPClient {
    private SocketChannel clientChannel;
    private ByteBuffer buf;

    public NIOTCPClient() {
        try {
            clientChannel = SocketChannel.open(new
InetSocketAddress("127.0.0.1", 9000));
            //设置客户端为非阻塞模式
            clientChannel.configureBlocking(false);
            buf = ByteBuffer.allocate(1024);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void send(String fileName) {
        try {
            FileChannel fileChannel = FileChannel.open(Paths.get(fileName),
StandardOpenOption.READ);
            while (fileChannel.read(buf) != -1) {
                buf.flip();
                clientChannel.write(buf);
                buf.clear();
            }
            System.out.println("客户端已发送文件" + fileName);
            fileChannel.close();
            clientChannel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ExecutorService pool = new ThreadPoolExecutor(50, 100, 1000,
TimeUnit.MILLISECONDS, new LinkedBlockingDeque<>());
        Instant begin = Instant.now();
        for (int i = 0; i < 200; i++) {
            pool.submit(() -> {
                NIOTCPClient client = new NIOTCPClient();
                client.send("E:/1.jpeg");
            });
        }
        pool.shutdown();
        Instant end = Instant.now();
        System.out.println(Duration.between(begin,end));
    }
}
```

- 

# Java AIO 使用

- 对AIO来说，它不是在IO准备好时再通知线程，而是在IO操作已经完成后，再给线程发出通知。因此AIO是不会阻塞的，此时我们的业务逻辑将变成一个回调函数，等待IO操作完成后，由系统自动触发。
- AIO的四步：
- 1、进程向操作系统请求数据
- 2、操作系统把外部数据加载到内核的缓冲区中，
- 3、操作系统把内核的缓冲区拷贝到进程的缓冲区
- 4、进程获得数据完成自己的功能
- JDK1.7主要增加了三个新的异步通道：
- AsynchronousFileChannel: 用于文件异步读写；
- AsynchronousSocketChannel: 客户端异步socket；
- AsynchronousServerSocketChannel: 服务器异步socket。
- 因为AIO的实施需充分调用OS参与，IO需要操作系统支持、并发也同样需要操作系统的
- 在AIO socket编程中，服务端通道是AsynchronousServerSocketChannel，这个类提供了一个open()静态工厂，一个bind()方法用于绑定服务端IP地址（还有端口号），另外还提供了accept()用于接收用户连接请求。在客户端使用的通道是AsynchronousSocketChannel,这个通道处理提供open静态工厂方法外，还提供了read和write方法。
- 在AIO编程中，发出一个事件（accept read write等)之后要指定事件处理类（回调函数)，AIO中的事件处理类是CompletionHandler<V,A>，这个接口定义了如下两个方法，分别在异步操作成功和失败时被回调。
- void completed(V result, A attachment);
- void failed(Throwable exc, A attachment);

```
public class AIOServer {
    private static int PORT = 8080;
    private static int BUFFER_SIZE = 1024;
    private static String CHARSET = "utf-8"; //默认编码
    private static CharsetDecoder decoder =
Charset.forName(CHARSET).newDecoder(); //解码

    private AsynchronousServerSocketChannel serverChannel;

    public AIOServer() {
        this.decoder = Charset.forName(CHARSET).newDecoder();
    }

    private void listen() throws Exception {

        //打开一个服务通道
        //绑定服务端口
        this.serverChannel = AsynchronousServerSocketChannel.open().bind(new
InetSocketAddress(PORT), 100);
        this.serverChannel.accept(this, new AcceptHandler());
    }



    /**
     * accept到一个请求时的回调
     */
    private class AcceptHandler implements
CompletionHandler<AsynchronousSocketChannel, AIOServer> {
        @Override
```

```
        public void completed(final AsynchronousSocketChannel client,
AIOServer server) {
            try {
                System.out.println("远程地址：" + client.getRemoteAddress());
                //tcp各项参数
                client.setOption(StandardSocketOptions.TCP_NODELAY, true);
                client.setOption(StandardSocketOptions.SO_SNDBUF, 1024);
                client.setOption(StandardSocketOptions.SO_RCVBUF, 1024);

                if (client.isOpen()) {
                    System.out.println("client.isOpen：" +
client.getRemoteAddress());
                    final ByteBuffer buffer =
ByteBuffer.allocate(BUFFER_SIZE);
                    buffer.clear();
                    client.read(buffer, client, new ReadHandler(buffer));
                }

            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                server.serverChannel.accept(server, this);// 监听新的请求，递归
调用。
            }
        }

        @Override
        public void failed(Throwable e, AIOServer attachment) {
            try {
                e.printStackTrace();
            } finally {
                attachment.serverChannel.accept(attachment, this);// 监听新的
请求，递归调用。
            }
        }
    }

    /**
     * Read到请求数据的回调
     */
    private class ReadHandler implements CompletionHandler<Integer,
AsynchronousSocketChannel> {

        private ByteBuffer buffer;

        public ReadHandler(ByteBuffer buffer) {
            this.buffer = buffer;
        }

        @Override
        public void completed(Integer result, AsynchronousSocketChannel
client) {
            try {
                if (result < 0) {// 客户端关闭了连接
                    AIOServer.close(client);
                } else if (result == 0) {
                    System.out.println("空数据"); // 处理空数据
                } else {
```

```
                    // 读取请求，处理客户端发送的数据
                    buffer.flip();
                    CharBuffer charBuffer =
AIOServer.decoder.decode(buffer);
                    System.out.println(charBuffer.toString()); //接收请求

                    //响应操作，服务器响应结果
                    buffer.clear();
                    String res = "hellworld";
                    buffer = ByteBuffer.wrap(res.getBytes());
                    client.write(buffer, client, new
WriteHandler(buffer));//Response：响应。
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        @Override
        public void failed(Throwable exc, AsynchronousSocketChannel
attachment) {
            exc.printStackTrace();
            AIOServer.close(attachment);
        }
    }

    /**
     * Write响应完请求的回调
     */
    private class WriteHandler implements CompletionHandler<Integer,
AsynchronousSocketChannel> {
        private ByteBuffer buffer;

        public WriteHandler(ByteBuffer buffer) {
            this.buffer = buffer;
        }

        @Override
        public void completed(Integer result, AsynchronousSocketChannel
attachment) {
            buffer.clear();
            AIOServer.close(attachment);
        }

        @Override
        public void failed(Throwable exc, AsynchronousSocketChannel
attachment) {
            exc.printStackTrace();
            AIOServer.close(attachment);
        }
    }


    private static void close(AsynchronousSocketChannel client) {
        try {
            client.close();
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println("正在启动服务...");
            AIOServer AIOServer = new AIOServer();
            AIOServer.listen();
            Thread t = new Thread(new Runnable() {
                @Override
                public void run() {
                    while (true) {
                    }
                }
            });
            t.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 

# Java NIO 源码

- 关于Selector源码过于难以理解，可以先放过。

## Buffer

```
public abstract class Buffer {

    /**
     * The characteristics of Spliterators that traverse and split elements
     * maintained in Buffers.
     */
    static final int SPLITERATOR_CHARACTERISTICS =
        Spliterator.SIZED | Spliterator.SUBSIZED | Spliterator.ORDERED;

    // Invariants: mark <= position <= limit <= capacity
    private int mark = -1;
    private int position = 0;
    private int limit;
    private int capacity;

    // Used only by direct buffers
    // NOTE: hoisted here for speed in JNI GetDirectBufferAddress
    long address;

    // Creates a new buffer with the given mark, position, limit, and
capacity,
    // after checking invariants.
    //
    Buffer(int mark, int pos, int lim, int cap) {       // package-private
        if (cap < 0)
            throw new IllegalArgumentException("Negative capacity: " + cap);
        this.capacity = cap;
```

```
        limit(lim);
        position(pos);
        if (mark >= 0) {
            if (mark > pos)
                throw new IllegalArgumentException("mark > position: ("
                                                   + mark + " > " + pos +
")");
            this.mark = mark;
        }
    }
```

- }
- ByteBuffer有两种实现：HeapByteBuffer和DirectByteBuffer。
- ByteBuffer#allocate

```
public static ByteBuffer allocate(int capacity) {
    if (capacity < 0)
        throw new IllegalArgumentException();
    return new HeapByteBuffer(capacity, capacity);
}
```

- ByteBuffer#allocateDirect

```
public static ByteBuffer allocateDirect(int capacity) {
    return new DirectByteBuffer(capacity);
}
```

## HeapByteBuffer（间接模式）

- 底层基于byte数组。

### 初始化

```
HeapByteBuffer(int cap, int lim) {            // package-private
    super(-1, 0, lim, cap, new byte[cap], 0);
}
```

- 调用的是ByteBuffer的初始化方法

```
ByteBuffer(int mark, int pos, int lim, int cap,   // package-private
             byte[] hb, int offset)
{
    super(mark, pos, lim, cap);
    this.hb = hb;
    this.offset = offset;
}
```

- ByteBuffer的独有成员变量：
- final byte[] hb;              // Non-null only for heap buffers
  final int offset;
  boolean isReadOnly;              // Valid only for heap buffers

### get

```
public byte get() {
    return hb[ix(nextGetIndex())];
}
```

```
final int nextGetIndex() {                          // package-private
    if (position >= limit)
        throw new BufferUnderflowException();
    return position++;
}
```

- protected int ix(int i) {
  return i + offset;
  }

### put

```
public ByteBuffer put(byte x) {
    hb[ix(nextPutIndex())] = x;
    return this;
}
```

```
final int nextPutIndex() {                          // package-private
    if (position >= limit)
        throw new BufferOverflowException();
    return position++;
}
```

-

## DirectByteBuffer（直接模式）

- 底层基于c++的malloc分配的堆外内存，是使用Unsafe类分配的，底层调用了native方法。
- 在创建DirectByteBuffer的同时，创建一个与其对应的cleaner，cleaner是一个虚引用。
- 回收堆外内存的几种情况：
  - 1)程序员手工释放，需要使用sun的非公开API实现。
  - 2)申请新的堆外内存而内存不足时，会进行调用Cleaner（作为一个Reference)的静态方法tryHandlePending(false)，它又会调用cleaner的clean方法释放内存。
  - 3)当DirectByteBuffer失去强引用,只有虚引用时，当等到某一次System.gc（full gc）(比如堆外内存达到XX:MaxDirectMemorySize)时，当DirectByteBuffer对象从pending状态 -> enqueue状态时，会触发Cleaner的clean()，而Cleaner的clean()的方法会实现通过unsafe对堆外内存的释放。

## 初始化

- 重要成员变量：

```
private final Cleaner cleaner;
```

- // Cached unsafe-access object
  protected static final Unsafe unsafe = Bits.unsafe();
- Unsafe中很多都是native方法，底层调用c++代码。

```
DirectByteBuffer(int cap) {                   // package-private

    super(-1, 0, cap, cap);
```

- // 内存是否按页分配对齐
  boolean pa = VM.isDirectMemoryPageAligned();
- // 获取每页内存大小
  int ps = Bits.pageSize();
- // 分配内存的大小，如果是按页对齐方式，需要再加一页内存的容量
  long size = Math.max(1L, (long)cap + (pa ? ps : 0));
- // 用Bits类保存总分配内存(按页分配)的大小和实际内存的大小
  Bits.reserveMemory(size, cap);

  long base = 0;
  try {
- // 在堆外内存的基地址，指定内存大小
    base = unsafe.allocateMemory(size);
  } catch (OutOfMemoryError x) {
    Bits.unreserveMemory(size, cap);
    throw x;
  }
  unsafe.setMemory(base, size, (byte) 0);
  - // 计算堆外内存的基地址
    if (pa && (base % ps != 0)) {
      // Round up to page boundary
      address = base + ps - (base & (ps - 1));
    } else {
      address = base;
    }
    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
    att = null;
    }
- 第一行super调用的是其父类MappedByteBuffer的构造方法

```
MappedByteBuffer(int mark, int pos, int lim, int cap) { // package-private
    super(mark, pos, lim, cap);
    this.fd = null;
}
```

- 而它的super又调用了ByteBuffer的构造方法

```
ByteBuffer(int mark, int pos, int lim, int cap) { // package-private
    this(mark, pos, lim, cap, null, 0);
}
```

- Bits#reserveMemory
- 该方法用于在系统中保存总分配内存(按页分配)的大小和实际内存的大小。
- 总的来说，Bits.reserveMemory(size, cap)方法在可用堆外内存不足以分配给当前要创建的堆外内存大小时，会实现以下的步骤来尝试完成本次堆外内存的创建：
- ① 触发一次非堵塞的Reference#tryHandlePending(false)。该方法会将已经被JVM垃圾回收的DirectBuffer对象的堆外内存释放。
- ② 如果进行一次堆外内存资源回收后，还不够进行本次堆外内存分配的话，则进行System.gc()。System.gc()会触发一个full gc，但你需要知道，调用System.gc()并不能够保证full gc马上就能被执行。所以在后面打代码中，会进行最多9次尝试，看是否有足够的可用堆外内存来分配堆外内存。并且每次尝试之前，都对延迟等待时间，已给JVM足够的时间去完成full gc操作。
- 这里之所以用使用full gc的很重要的一个原因是：System.gc()会对新生代和老生代都会进行内存回收，这样会比较彻底地回收DirectByteBuffer对象以及他们关联的堆外内存.
- DirectByteBuffer对象本身其实是很小的，但是它后面可能关联了一个非常大的堆外内存，因此我们通常称之为冰山对象.
- 我们做young gc的时候会将新生代里的不可达的DirectByteBuffer对象及其堆外内存回收了，但是无法对old里的DirectByteBuffer对象及其堆外内存进行回收，这也是我们通常碰到的最大的问题。(并且堆外内存多用于生命期中等或较长的对象)
- 如果有大量的DirectByteBuffer对象移到了old，但是又一直没有做cms gc或者full gc，而只进行ygc，那么我们的物理内存可能被慢慢耗光，但是我们还不知道发生了什么，因为heap明明剩余的内存还很多(前提是我们禁用了System.gc – JVM参数DisableExplicitGC)。
- 注意，如果你设置了-XX:+DisableExplicitGC，将会禁用显示GC，这会使System.gc()调用无效。
- ③ 如果9次尝试后依旧没有足够的可用堆外内存来分配本次堆外内存，则抛出OutOfMemoryError("Direct buffer memory")异常。
- static void reserveMemory(long size, int cap) {

  if (!memoryLimitSet && VM.isBooted()) {
  maxMemory = VM.maxDirectMemory();
  memoryLimitSet = true;
  }

  // optimist!
  if (tryReserveMemory(size, cap)) {
  return;
  }

  final JavaLangRefAccess jlra = SharedSecrets.getJavaLangRefAccess();
  // 如果系统中内存( 即，堆外内存 )不够的话：
- jlra.tryHandlePendingReference()会触发一次非堵塞的Reference#tryHandlePending(false)。该方法会将已经被JVM垃圾回收的DirectBuffer对象的堆外内存释放。

- // retry while helping enqueue pending Reference objects
  // which includes executing pending Cleaner(s) which includes
  // Cleaner(s) that free direct buffer memory
  while (jlra.tryHandlePendingReference()) {
  if (tryReserveMemory(size, cap)) {
  return;
  }
  }
  // 如果在进行一次堆外内存资源回收后，还不够进行本次堆外内存分配的话，则
  // trigger VM's Reference processing
  System.gc();

  // a retry loop with exponential back-off delays
  // (this gives VM some time to do it's job)
  boolean interrupted = false;
  try {
  long sleepTime = 1;
  int sleeps = 0;
  while (true) {
  if (tryReserveMemory(size, cap)) {
  return;
  }
- // 9
  if (sleeps >= MAX_SLEEPS) {
  break;
  }
  if (!jlra.tryHandlePendingReference()) {
  try {
  Thread.sleep(sleepTime);
  sleepTime <<= 1;
  sleeps++;
  } catch (InterruptedException e) {
  interrupted = true;
  }
  }
  }
  // 如果9次尝试后依旧没有足够的可用堆外内存来分配本次堆外内存，则抛出
  OutOfMemoryError("Direct buffer memory”)异常。
  // no luck
  throw new OutOfMemoryError("Direct buffer memory");

  } finally {
  if (interrupted) {
  // don't swallow interrupts
  Thread.currentThread().interrupt();
  }
  }
  }
- Reference#tryHandlePending

- static boolean tryHandlePending(boolean waitForNotify) {
  Reference r;
  Cleaner c;
  try {
    synchronized (lock) {
      if (pending != null) {
        r = pending;
        // 'instanceof' might throw OutOfMemoryError sometimes
        // so do this before un-linking 'r' from the 'pending' chain...
        c = r instanceof Cleaner ? (Cleaner) r : null;
        // unlink 'r' from 'pending' chain
        pending = r.discovered;
        r.discovered = null;
      } else {
        // The waiting on the lock may cause an OutOfMemoryError
        // because it may try to allocate exception objects.
        if (waitForNotify) {
          lock.wait();
        }
        // retry if waited
        return waitForNotify;
      }
    }
  } catch (OutOfMemoryError x) {
    // Give other threads CPU time so they hopefully drop some live references
    // and GC reclaims some space.
    // Also prevent CPU intensive spinning in case 'r instanceof Cleaner' above
    // persistently throws OOME for some time...
    Thread.yield();
    // retry
    return true;
  } catch (InterruptedException x) {
    // retry
    return true;
  }

  // Fast path for cleaners
  if (c != null) {
    c.clean();
    return true;
  }

  ReferenceQueue<? super Object> q = r.queue;
  if (q != ReferenceQueue.NULL) q.enqueue(r);
  return true;
  }

## Deallocator

- 后面是调用unsafe的分配堆外内存的方法，然后初始化了该DirectByteBuffer对应的cleaner。
- 注：在Cleaner 内部中通过一个列表，维护了一个针对每一个 directBuffer 的一个回收堆外内存的 线程对象(Runnable)，回收操作是发生在 Cleaner 的 clean() 方法中。

```
private static class Deallocator
    implements Runnable
{

    private static Unsafe unsafe = Unsafe.getUnsafe();

    private long address;
    private long size;
    private int capacity;

    private Deallocator(long address, long size, int capacity) {
        assert (address != 0);
        this.address = address;
        this.size = size;
        this.capacity = capacity;
    }

    public void run() {
        if (address == 0) {
            // Paranoia
            return;
        }
        unsafe.freeMemory(address);
        address = 0;
        Bits.unreserveMemory(size, capacity);
    }

}
```

## Cleaner（回收）

```
public class Cleaner extends PhantomReference<Object> {
    private static final ReferenceQueue<Object> dummyQueue = new
ReferenceQueue();
    private static Cleaner first = null;
    private Cleaner next = null;
    private Cleaner prev = null;
    private final Runnable thunk;

    private static synchronized Cleaner add(Cleaner var0) {
        if (first != null) {
            var0.next = first;
            first.prev = var0;
        }

        first = var0;
        return var0;
    }

    private static synchronized boolean remove(Cleaner var0) {
        if (var0.next == var0) {
            return false;
        } else {
            if (first == var0) {
                if (var0.next != null) {
                    first = var0.next;
                } else {
```

```
                    first = var0.prev;
                }
            }

            if (var0.next != null) {
                var0.next.prev = var0.prev;
            }

            if (var0.prev != null) {
                var0.prev.next = var0.next;
            }

            var0.next = var0;
            var0.prev = var0;
            return true;
        }
    }

    private Cleaner(Object var1, Runnable var2) {
        super(var1, dummyQueue);
        this.thunk = var2;
    }
    // var0是DirectByteBuffer，var1是Deallocator线程对象
    public static Cleaner create(Object var0, Runnable var1) {
        return var1 == null ? null : add(new Cleaner(var0, var1));
    }

    public void clean() {
        if (remove(this)) {
            try {
```

```
// 回收该DirectByteBuffer对应的堆外内存
                this.thunk.run();
            } catch (final Throwable var2) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        if (System.err != null) {
                            (new Error("Cleaner terminated abnormally",
var2)).printStackTrace();
                        }

                        System.exit(1);
                        return null;
                    }
                });
            }

        }
    }
}
```

- Cleaner的构造方法中又调用了父类虚引用的构造方法：

```
public PhantomReference(T referent, ReferenceQueue<? super T> q) {
    super(referent, q);
}
```

### get

```
public byte get() {
    return ((unsafe.getByte(ix(nextGetIndex()))));
}
```

### put

```
public ByteBuffer put(byte x) {
    unsafe.putByte(ix(nextPutIndex()), ((x)));
    return this;
}
```

- 

## FileChannel（阻塞式）

- FileChannel的read、write和map通过其实现类FileChannelImpl实现。
- FileChannelImpl的Oracle JDK没有提供源码，只能在OpenJDK中查看。

### open

```
public static FileChannel open(Path path, OpenOption... options)
    throws IOException
{
    Set<OpenOption> set = new HashSet<OpenOption>(options.length);
    Collections.addAll(set, options);
    return open(path, set, NO_ATTRIBUTES);
}
```

```
public static FileChannel open(Path path,
                               Set<? extends OpenOption> options,
                               FileAttribute<?>... attrs)
    throws IOException
{
    FileSystemProvider provider = path.getFileSystem().provider();
    return provider.newFileChannel(path, options, attrs);
}
```

- WindowsFileSystemProvider#newFileChannel

```
public FileChannel newFileChannel(Path path,
                                  Set<? extends OpenOption> options,
                                  FileAttribute<?>... attrs)
    throws IOException
{
    if (path == null)
        throw new NullPointerException();
    if (!(path instanceof WindowsPath))
        throw new ProviderMismatchException();
    WindowsPath file = (WindowsPath)path;
```

```
    WindowsSecurityDescriptor sd =
WindowsSecurityDescriptor.fromAttribute(attrs);
    try {
        return WindowsChannelFactory
            .newFileChannel(file.getPathForWin32Calls(),
                            file.getPathForPermissionCheck(),
                            options,
                            sd.address());
    } catch (WindowsException x) {
        x.rethrowAsIOException(file);
        return null;
    } finally {
        if (sd != null)
            sd.release();
    }
}
```

- WindowsChannelFactory#newFileChannel
- static FileChannel newFileChannel(String pathForWindows,
  String pathToCheck,
  Set<? extends OpenOption> options,
  long pSecurityDescriptor)
  throws WindowsException
  {
  Flags flags = Flags.toFlags(options);

  // default is reading; append => writing
  if (!flags.read && !flags.write) {
  if (flags.append) {
  flags.write = true;
  } else {
  flags.read = true;
  }
  }

  // validation
  if (flags.read && flags.append)
  throw new IllegalArgumentException("READ + APPEND not allowed");
  if (flags.append && flags.truncateExisting)
  throw new IllegalArgumentException("APPEND + TRUNCATE_EXISTING not allowed");

  FileDescriptor fdObj = open(pathForWindows, pathToCheck, flags, pSecurityDescriptor);
  return FileChannelImpl.open(fdObj, pathForWindows, flags.read, flags.write,
  flags.append, null);
  }

```
/**
 * Opens file based on parameters and options, returning a FileDescriptor
 * encapsulating the handle to the open file.
 */
private static FileDescriptor open(String pathForWindows,
                                   String pathToCheck,
                                   Flags flags,
                                   long pSecurityDescriptor)
    throws WindowsException
{
```

```
    // set to true if file must be truncated after open
    boolean truncateAfterOpen = false;

    // map options
    int dwDesiredAccess = 0;
    if (flags.read)
        dwDesiredAccess |= GENERIC_READ;
    if (flags.write)
        dwDesiredAccess |= GENERIC_WRITE;

    int dwShareMode = 0;
    if (flags.shareRead)
        dwShareMode |= FILE_SHARE_READ;
    if (flags.shareWrite)
        dwShareMode |= FILE_SHARE_WRITE;
    if (flags.shareDelete)
        dwShareMode |= FILE_SHARE_DELETE;

    int dwFlagsAndAttributes = FILE_ATTRIBUTE_NORMAL;
    int dwCreationDisposition = OPEN_EXISTING;
    if (flags.write) {
        if (flags.createNew) {
            dwCreationDisposition = CREATE_NEW;
            // force create to fail if file is orphaned reparse point
            dwFlagsAndAttributes |= FILE_FLAG_OPEN_REPARSE_POINT;
        } else {
            if (flags.create)
                dwCreationDisposition = OPEN_ALWAYS;
            if (flags.truncateExisting) {
                // Windows doesn't have a creation disposition that exactly
                // corresponds to CREATE + TRUNCATE_EXISTING so we use
                // the OPEN_ALWAYS mode and then truncate the file.
                if (dwCreationDisposition == OPEN_ALWAYS) {
                    truncateAfterOpen = true;
                } else {
                    dwCreationDisposition = TRUNCATE_EXISTING;
                }
            }
        }
    }

    if (flags.dsync || flags.sync)
        dwFlagsAndAttributes |= FILE_FLAG_WRITE_THROUGH;
    if (flags.overlapped)
        dwFlagsAndAttributes |= FILE_FLAG_OVERLAPPED;
    if (flags.deleteOnClose)
        dwFlagsAndAttributes |= FILE_FLAG_DELETE_ON_CLOSE;

    // NOFOLLOW_LINKS and NOFOLLOW_REPARSEPOINT mean open reparse point
    boolean okayToFollowLinks = true;
    if (dwCreationDisposition != CREATE_NEW &&
        (flags.noFollowLinks ||
         flags.openReparsePoint ||
         flags.deleteOnClose))
    {
        if (flags.noFollowLinks || flags.deleteOnClose)
            okayToFollowLinks = false;
        dwFlagsAndAttributes |= FILE_FLAG_OPEN_REPARSE_POINT;
```

```
    }

    // permission check
    if (pathToCheck != null) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            if (flags.read)
                sm.checkRead(pathToCheck);
            if (flags.write)
                sm.checkWrite(pathToCheck);
            if (flags.deleteOnClose)
                sm.checkDelete(pathToCheck);
        }
    }

    // open file
    long handle = CreateFile(pathForWindows,
                             dwDesiredAccess,
                             dwShareMode,
                             pSecurityDescriptor,
                             dwCreationDisposition,
                             dwFlagsAndAttributes);

    // make sure this isn't a symbolic link.
    if (!okayToFollowLinks) {
        try {
            if
(WindowsFileAttributes.readAttributes(handle).isSymbolicLink())
                throw new WindowsException("File is symbolic link");
        } catch (WindowsException x) {
            CloseHandle(handle);
            throw x;
        }
    }

    // truncate file (for CREATE + TRUNCATE_EXISTING case)
    if (truncateAfterOpen) {
        try {
            SetEndOfFile(handle);
        } catch (WindowsException x) {
            CloseHandle(handle);
            throw x;
        }
    }

    // make the file sparse if needed
    if (dwCreationDisposition == CREATE_NEW && flags.sparse) {
        try {
            DeviceIoControlSetSparse(handle);
        } catch (WindowsException x) {
            // ignore as sparse option is hint
        }
    }

    // create FileDescriptor and return
    FileDescriptor fdObj = new FileDescriptor();
    fdAccess.setHandle(fdObj, handle);
    return fdObj;
```

```
}
```

- static long CreateFile(String path,
  int dwDesiredAccess,
  int dwShareMode,
  long lpSecurityAttributes,
  int dwCreationDisposition,
  int dwFlagsAndAttributes)
  throws WindowsException
  {
  NativeBuffer buffer = asNativeBuffer(path);
  try {
  return CreateFile0(buffer.address(),
  dwDesiredAccess,
  dwShareMode,
  lpSecurityAttributes,
  dwCreationDisposition,
  dwFlagsAndAttributes);
  } finally {
  buffer.release();
  }
  }

```
private static native long CreateFile0(long lpFileName,
                                       int dwDesiredAccess,
                                       int dwShareMode,
                                       long lpSecurityAttributes,
                                       int dwCreationDisposition,
                                       int dwFlagsAndAttributes)
    throws WindowsException;
```

- 

## read

```
public int read(ByteBuffer dst) throws IOException {
    ensureOpen();
    if (!readable)
        throw new NonReadableChannelException();
    synchronized (positionLock) {
        int n = 0;
        int ti = -1;
        try {
            begin();
            ti = threads.add();
            if (!isOpen())
                return 0;
            do {
                n = IOUtil.read(fd, dst, -1, nd);
            } while ((n == IOStatus.INTERRUPTED) && isOpen());
            return IOStatus.normalize(n);
        } finally {
            threads.remove(ti);
            end(n > 0);
```

```
            assert IOStatus.check(n);
        }
    }
}
```

- IOUtil.read
- static int read(FileDescriptor fd, ByteBuffer dst, long position,
  NativeDispatcher nd) IOException {
  if (dst.isReadOnly())
  throw new IllegalArgumentException("Read-only buffer");
  if (dst instanceof DirectBuffer)
  return readIntoNativeBuffer(fd, dst, position, nd);

  // Substitute a native buffer
  ByteBuffer bb = Util.getTemporaryDirectBuffer(dst.remaining());
  try {
  int n = readIntoNativeBuffer(fd, bb, position, nd);
  bb.flip();
  if (n > 0)
  dst.put(bb);
  return n;
  } finally {
  Util.offerFirstTemporaryDirectBuffer(bb);
  }
  }
- 通过上述实现可以看出，基于channel的文件数据读取步骤如下：
- 1、申请一块和缓存同大小的DirectByteBuffer bb。
- 2、读取数据到缓存bb，底层由NativeDispatcher的read实现。
- 3、把bb的数据读取到dst（用户定义的缓存，在jvm中分配内存)。
- read方法导致数据复制了两次。

## write

```
public int write(ByteBuffer src) throws IOException {
    ensureOpen();
    if (!writable)
        throw new NonWritableChannelException();
    synchronized (positionLock) {
        int n = 0;
        int ti = -1;
        try {
            begin();
            ti = threads.add();
            if (!isOpen())
                return 0;
            do {
                n = IOUtil.write(fd, src, -1, nd);
            } while ((n == IOStatus.INTERRUPTED) && isOpen());
            return IOStatus.normalize(n);
        } finally {
            threads.remove(ti);
            end(n > 0);
            assert IOStatus.check(n);
```

```
        }
    }
}
```

- IOUtil.write
- static int write(FileDescriptor fd, ByteBuffer src, long position,
  NativeDispatcher nd) throws IOException {
  if (src instanceof DirectBuffer)
  return writeFromNativeBuffer(fd, src, position, nd);
  // Substitute a native buffer
  int pos = src.position();
  int lim = src.limit();
  assert (pos <= lim);
  int rem = (pos <= lim ? lim - pos : 0);
  ByteBuffer bb = Util.getTemporaryDirectBuffer(rem);
  try {
  bb.put(src);
  bb.flip();
  // Do not update src until we see how many bytes were written
  src.position(pos);
  int n = writeFromNativeBuffer(fd, bb, position, nd);
  if (n > 0) {
  // now update src
  src.position(pos + n);
  }
  return n;
  } finally {
  Util.offerFirstTemporaryDirectBuffer(bb);
  }
  }
- 基于channel的文件数据写入步骤如下：
- 1、申请一块DirectByteBuffer，bb大小为byteBuffer中的limit - position。
- 2、复制byteBuffer中的数据到bb中。
- 3、把数据从bb中写入到文件，底层由NativeDispatcher的write实现，具体如下：

```
private static int writeFromNativeBuffer(FileDescriptor fd,
                                         ByteBuffer bb, long position,
NativeDispatcher nd)
        throws IOException {
    int pos = bb.position();
    int lim = bb.limit();
    assert (pos <= lim);
    int rem = (pos <= lim ? lim - pos : 0);

    int written = 0;
    if (rem == 0)
        return 0;
    if (position != -1) {
        written = nd.pwrite(fd,
                ((DirectBuffer) bb).address() + pos,
                rem, position);
    } else {
        written = nd.write(fd, ((DirectBuffer) bb).address() + pos, rem);
    }
```

```
    if (written > 0)
        bb.position(pos + written);
    return written;
}
```

- write方法也导致了数据复制了两次。
-

## ServerSocketChannel

- 它的实现类是ServerSocketChannelImpl，同样是闭源的。

### open

```
public static ServerSocketChannel open() throws IOException {
    return SelectorProvider.provider().openServerSocketChannel();
}
```

- SelectorProvider.provider()方法在windows平台下返回的是SelectorProvider 的实现类WindowsSelectorProvider类的实例。
- WindowsSelectorProvider类的直接父类为SelectorProviderImpl；
- SelectorProviderImpl 的直接父类是 SelectorProvider。
- SelectorProviderImpl# openServerSocketChannel

```
public ServerSocketChannel openServerSocketChannel() throws IOException {
    return new ServerSocketChannelImpl(this);
}
```

```
- ServerSocketChannelImpl(SelectorProvider var1) throws IOException {
super(var1);
this.fd = Net.serverSocket(true);
this.fdVal = IOUtil.fdVal(this.fd);
this.state = 0;
```

}
- super(var1)实际上是父类的构造方法

- protected AbstractSelectableChannel(SelectorProvider provider) {
  this.provider = provider;
  }
- Net#serverSocket 创建一个FileDescriptor
- static FileDescriptor serverSocket(boolean stream) {
  return IOUtil.newFD(socket0(isIPv6Available(), stream, true, fastLoopback));
  }
-

### bind

```
public ServerSocketChannel bind(SocketAddress local, int backlog) throws
IOException {
    synchronized (lock) {
        if (!isOpen())
            throw new ClosedChannelException();
        if (isBound())
```

```
            throw new AlreadyBoundException();
        InetSocketAddress isa = (local == null) ? new InetSocketAddress(0) :
            Net.checkAddress(local);
        SecurityManager sm = System.getSecurityManager();
        if (sm != null)
            sm.checkListen(isa.getPort());
        NetHooks.beforeTcpBind(fd, isa.getAddress(), isa.getPort());
        Net.bind(fd, isa.getAddress(), isa.getPort());
        Net.listen(fd, backlog < 1 ? 50 : backlog);
        synchronized (stateLock) {
            localAddress = Net.localAddress(fd);
        }
    }
    return this;
}
```

- 

## register

- Selector是通过Selector.open方法获得的。
- 将这个通道channel注册到指定的selector中，返回一个SelectionKey对象实例。
- register这个方法在实现代码上的逻辑有以下四点：
- 1、首先检查通道channel是否是打开的，如果不是打开的，则抛异常，如果是打开的，则进行 2。
- 2、检查指定的interest集合是否是有效的。如果没效，则抛异常。否则进行 3。这里要特别强调一下：对于ServerSocketChannel仅仅支持"新的连接"，因此interest集合ops满足ops&~sectionKey.OP_ACCEPT!=0,即对于ServerSocketChannel注册到Selector中时的事件只能包括SelectionKey.OP_ACCEPT。
- 3、对通道进行了阻塞模式的检查，如果不是阻塞模式，则抛异常，否则进行4.
- 4、得到当前通道在指定Selector上的SelectionKey，假设结果用k表示。下面对k是否为null有不同的处理。如果k不为null，则说明此通道channel已经在Selector上注册过了，则直接将指定的ops添加进SelectionKey中即可。如果k为null,则说明此通道还没有在Selector上注册，则需要先进行注册，然后为其对应的SelectionKey设置给定值ops。
- 与Selector一起使用时，Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

```
public final SelectionKey register(Selector sel, int ops,
                                   Object att)
    throws ClosedChannelException
{
    synchronized (regLock) {
        if (!isOpen())
            throw new ClosedChannelException();
        if ((ops & ~validOps()) != 0)
            throw new IllegalArgumentException();
        if (blocking)
            throw new IllegalBlockingModeException();
```

- // 得到当前通道在指定Selector上的SelectionKey（复合事件）
- SelectionKey k = findKey(sel);
- 如果k不为null，则说明此通道已经在Selector上注册过了，则直接将指定的ops添加进SelectionKey中即可。
- 如果k为null,则说明此通道还没有在Selector上注册，则需要先进行注册，然后添加SelectionKey。

```
        if (k != null) {
            k.interestOps(ops);
            k.attach(att);
        }
        if (k == null) {
            // New registration
            synchronized (keyLock) {
                if (!isOpen())
                    throw new ClosedChannelException();
                k = ((AbstractSelector)sel).register(this, ops, att);
                addKey(k);
            }
        }
        return k;
    }
```

```
private SelectionKey findKey(Selector sel) {
    synchronized (keyLock) {
        if (keys == null)
            return null;
        for (int i = 0; i < keys.length; i++)
            if ((keys[i] != null) && (keys[i].selector() == sel))
                return keys[i];
        return null;
    }
}
```

- 

## Selector（如何实现Channel多路复用）

- SocketChannel、ServerSocketChannel和Selector的实例初始化都通过SelectorProvider类实现，其中Selector是整个NIO Socket的核心实现。
- SelectorProvider在windows和linux下有不同的实现，provider方法会返回对应的实现。

### 成员变量

1.- final class WindowsSelectorImpl extends SelectorImpl
2.- {
3.

```
    private final int INIT_CAP = 8;//选择key集合，key包装集合初始化容量
```

4.

```
    private static final int MAX_SELECTABLE_FDS = 1024;//最大选择key数量
```

5.

```
    private SelectionKeyImpl channelArray[];//选择器关联通道集合
```

6.

```
    private PollArrayWrapper pollWrapper;//存放所有文件描述对象（选择key，唤醒管道的source与sink通道)的集合
```

7.

```
    private int totalChannels;//注册到选择的通道数量
```

8.

```
    private int threadsCount;//选择线程数
```

9.

```
    private final List threads = new ArrayList();//选择操作线程集合
```

10.

```
    private final Pipe wakeupPipe = Pipe.open();//唤醒等待选择操作的管道
```

11.

```
    private final int wakeupSourceFd;//唤醒管道源通道文件描述
```

12.

```
    private final int wakeupSinkFd;//唤醒管道sink通道文件描述
```

13.

```
    private Object closeLock;//选择器关闭同步锁
```

14.

```
    private final FdMap fdMap = new FdMap();//存放选择key文件描述与选择key映射关系的Map
```

15.

```
    private final SubSelector subSelector = new SubSelector();//子选择器
```

16.

```
    private long timeout;//超时时间，具体什么意思，现在还没明白，在后面在看
```

17.

```
    private final Object interruptLock = new Object();//中断同步锁，在唤醒选择操作线程时，用于同步
```

18.

```
    private volatile boolean interruptTriggered;//是否唤醒等待选择操的线程
```

19.

```
    private final StartLock startLock = new StartLock();//选择操作开始锁
```

20.

```
    private final FinishLock finishLock = new FinishLock();//选择操作结束锁
```

21.

```
    private long updateCount;//更新数量，具体什么意思，现在还没明白，在后面在看
```

22.-
static final boolean $assertionsDisabled = !sun/nio/ch/WindowsSelectorImpl.desiredAssertionStatus();
23.- static
24.- {
25.- //加载nio，net资源库
26.- Util.load();
27.- }
28.- }

## open

```
public static Selector open() throws IOException {
    return SelectorProvider.provider().openSelector();
}
```

- WindowsSelectorProvider#openSelector

```
public AbstractSelector openSelector() throws IOException {
    return new WindowsSelectorImpl(this);
}
```

- 初始化一个wakeupPipe
  - WindowsSelectorImpl(SelectorProvider var1) throws IOException {
    super(var1);
    this.wakeupSourceFd = ((SelChImpl)this.wakeupPipe.source()).getFDVal();
    SinkChannelImpl var2 = (SinkChannelImpl)this.wakeupPipe.sink();
    var2.sc.socket().setTcpNoDelay(true);
    this.wakeupSinkFd = var2.getFDVal();
    this.pollWrapper.addWakeupSocket(this.wakeupSourceFd, 0);
    }
- SelectorImpl#register
- 第一个参数是ServerSocketChannel，第二个参数是复合事件，第三个是附件。

- 1、以当前channel和selector为参数，初始化SelectionKeyImpl 对象selectionKeyImpl ，并添加附件attachment。
- 2、如果当前channel的数量totalChannels等于SelectionKeyImpl数组大小，对SelectionKeyImpl数组和pollWrapper进行扩容操作。
- 3、如果totalChannels % MAXSELECTABLEFDS == 0，则多开一个线程处理selector。
- 4、pollWrapper.addEntry将把selectionKeyImpl中的socket句柄添加到对应的pollfd。
- 5、k.interestOps(ops)方法最终也会把event添加到对应的pollfd。
- 所以，不管serverSocketChannel，还是socketChannel，在selector注册的事件，最终都保存在pollArray中。

```
protected final SelectionKey register(AbstractSelectableChannel ch,
                                      int ops,
                                      Object attachment) {
    if (!(ch instanceof SelChImpl))
        throw new IllegalSelectorException();
    SelectionKeyImpl k = new SelectionKeyImpl((SelChImpl)ch, this);
    k.attach(attachment);
    synchronized (publicKeys) {
        implRegister(k);
    }
    k.interestOps(ops);
    return k;
}
```

- WindowsSelectorImpl#implRegister
- protected void implRegister(SelectionKeyImpl ski) {
  synchronized (closeLock) {
    if (pollWrapper == null)
      throw new ClosedSelectorException();
    growIfNeeded();
    channelArray[totalChannels] = ski;
    ski.setIndex(totalChannels);
    fdMap.put(ski);
    keys.add(ski);
    pollWrapper.addEntry(totalChannels, ski);
    totalChannels++;
  }
  }

## select（返回有事件发生的SelectionKey数量)

- var1是timeout时间，无参数的版本对应的timeout为0.
- select(long timeout)和select()一样，除了最长会阻塞timeout毫秒(参数)。
- 这个方法并不能提供精确时间的保证，和当执行wait(long timeout)方法时并不能保证会延时timeout道理一样。
- 这里的timeout说明如下：
- 如果 timeout为正，则select(long timeout)在等待有通道被选择时至多会阻塞timeout毫秒
- 如果timeout为零，则永远阻塞直到有至少一个通道准备就绪。
- timeout不能为负数。

```
public int select(long timeout)
    throws IOException
{
    if (timeout < 0)
        throw new IllegalArgumentException("Negative timeout");
    return lockAndDoSelect((timeout == 0) ? -1 : timeout);
}
```

- selectNow（非阻塞版本)

```
public int selectNow() throws IOException {
    return this.lockAndDoSelect(0L);
}
```

```
private int lockAndDoSelect(long timeout) throws IOException {
    synchronized (this) {
        if (!isOpen())
            throw new ClosedSelectorException();
        synchronized (publicKeys) {
            synchronized (publicSelectedKeys) {
                return doSelect(timeout);
            }
        }
    }
}
```

- WindowsSelectorImpl#doSelect
- protected int doSelect(long timeout) throws IOException {
  if (channelArray == null)
    throw new ClosedSelectorException();
  this.timeout = timeout; // set selector timeout
  processDeregisterQueue();
  if (interruptTriggered) {
    resetWakeupSocket();
    return 0;
  }
  // Calculate number of helper threads needed for poll. If necessary
  // threads are created here and start waiting on startLock
  adjustThreadsCount();
  finishLock.reset(); // reset finishLock
  // Wakeup helper threads, waiting on startLock, so they start polling.
  // Redundant threads will exit here after wakeup.
  startLock.startThreads();
  // do polling in the main thread. Main thread is responsible for
  // first MAX_SELECTABLE_FDS entries in pollArray.
  try {
    begin();
    try {
      subSelector.poll();
    } catch (IOException e) {
      finishLock.setException(e); // Save this exception

```
        }
        // Main thread is out of poll(). Wakeup others and wait for them
        if (threads.size() > 0)
           finishLock.waitForHelperThreads();
       } finally {
         end();
       }
      // Done with poll(). Set wakeupSocket to nonsignaled  for the next run.
      finishLock.checkForException();
      processDeregisterQueue();
      int updated = updateSelectedKeys();
      // Done with poll(). Set wakeupSocket to nonsignaled  for the next run.
      resetWakeupSocket();
      return updated;
      }
```

- 其中 subSelector.poll() 是select的核心，由native函数poll0实现，readFds、writeFds 和 exceptFds数组用来保存底层select的结果，数组的第一个位置都是存放发生事件的socket的总数，其余位置存放发生事件的socket句柄fd。

```
private int poll() throws IOException {
    return this.poll0(WindowsSelectorImpl.this.pollWrapper.pollArrayAddress,
Math.min(WindowsSelectorImpl.this.totalChannels, 1024), this.readFds,
this.writeFds, this.exceptFds, WindowsSelectorImpl.this.timeout);
}
```

```
private native int poll0(long pollAddress, int numfds,
     int[] readFds, int[] writeFds, int[] exceptFds, long timeout);
```

- 在src/windows/native/sun/nio/ch/WindowsSelectorImpl.c中找到了该方法的实现
- #define FD_SETSIZE 1024

```
Java_sun_nio_ch_WindowsSelectorImpl_00024SubSelector_poll0(JNIEnv *env,
jobject this,
                                    jlong pollAddress, jint numfds,
                                    jintArray returnReadFds, jintArray
returnWriteFds,
                                    jintArray returnExceptFds, jlong timeout)
{
    DWORD result = 0;
    pollfd *fds = (pollfd *) pollAddress;
    int i;
    FD_SET readfds, writefds, exceptfds;
    struct timeval timevalue, *tv;
    static struct timeval zerotime = {0, 0};
    int read_count = 0, write_count = 0, except_count = 0;

#ifdef _WIN64
    int resultbuf[FD_SETSIZE + 1];
#endif

    if (timeout == 0) {
        tv = &zerotime;
```

```
    } else if (timeout < 0) {
        tv = NULL;
    } else {
        tv = &timevalue;
        tv->tv_sec =  (long)(timeout / 1000);
        tv->tv_usec = (long)((timeout % 1000) * 1000);
    }

    /* Set FD_SET structures required for select */
    for (i = 0; i < numfds; i++) {
        if (fds[i].events & POLLIN) {
           readfds.fd_array[read_count] = fds[i].fd;
           read_count++;
        }
        if (fds[i].events & (POLLOUT | POLLCONN))
        {
           writefds.fd_array[write_count] = fds[i].fd;
           write_count++;
        }
        exceptfds.fd_array[except_count] = fds[i].fd;
        except_count++;
    }

    readfds.fd_count = read_count;
    writefds.fd_count = write_count;
    exceptfds.fd_count = except_count;

    /* Call select */
    if ((result = select(0 , &readfds, &writefds, &exceptfds, tv))
                                                                  ==
SOCKET_ERROR) {
        /* Bad error - this should not happen frequently */
        /* Iterate over sockets and call select() on each separately */
        FD_SET errreadfds, errwritefds, errexceptfds;
        readfds.fd_count = 0;
        writefds.fd_count = 0;
        exceptfds.fd_count = 0;
        for (i = 0; i < numfds; i++) {
            /* prepare select structures for the i-th socket */
            errreadfds.fd_count = 0;
            errwritefds.fd_count = 0;
            if (fds[i].events & POLLIN) {
               errreadfds.fd_array[0] = fds[i].fd;
               errreadfds.fd_count = 1;
            }
            if (fds[i].events & (POLLOUT | POLLCONN))
            {
                errwritefds.fd_array[0] = fds[i].fd;
                errwritefds.fd_count = 1;
            }
            errexceptfds.fd_array[0] = fds[i].fd;
            errexceptfds.fd_count = 1;

            /* call select on the i-th socket */
            if (select(0, &errreadfds, &errwritefds, &errexceptfds,
&zerotime)
                                                                  ==
SOCKET_ERROR) {
```

```
                /* This socket causes an error. Add it to exceptfds set */
                exceptfds.fd_array[exceptfds.fd_count] = fds[i].fd;
                exceptfds.fd_count++;
            } else {
                /* This socket does not cause an error. Process result */
                if (errreadfds.fd_count == 1) {
                    readfds.fd_array[readfds.fd_count] = fds[i].fd;
                    readfds.fd_count++;
                }
                if (errwritefds.fd_count == 1) {
                    writefds.fd_array[writefds.fd_count] = fds[i].fd;
                    writefds.fd_count++;
                }
                if (errexceptfds.fd_count == 1) {
                    exceptfds.fd_array[exceptfds.fd_count] = fds[i].fd;
                    exceptfds.fd_count++;
                }
            }
        }
    }

    /* Return selected sockets. */
    /* Each Java array consists of sockets count followed by sockets list */

#ifdef _WIN64
    resultbuf[0] = readfds.fd_count;
    for (i = 0; i < (int)readfds.fd_count; i++) {
        resultbuf[i + 1] = (int)readfds.fd_array[i];
    }
    (*env)->SetIntArrayRegion(env, returnReadFds, 0,
                              readfds.fd_count + 1, resultbuf);

    resultbuf[0] = writefds.fd_count;
    for (i = 0; i < (int)writefds.fd_count; i++) {
        resultbuf[i + 1] = (int)writefds.fd_array[i];
    }
    (*env)->SetIntArrayRegion(env, returnWriteFds, 0,
                              writefds.fd_count + 1, resultbuf);

    resultbuf[0] = exceptfds.fd_count;
    for (i = 0; i < (int)exceptfds.fd_count; i++) {
        resultbuf[i + 1] = (int)exceptfds.fd_array[i];
    }
    (*env)->SetIntArrayRegion(env, returnExceptFds, 0,
                              exceptfds.fd_count + 1, resultbuf);
#else
    (*env)->SetIntArrayRegion(env, returnReadFds, 0,
                              readfds.fd_count + 1, (jint *)&readfds);

    (*env)->SetIntArrayRegion(env, returnWriteFds, 0,
                              writefds.fd_count + 1, (jint *)&writefds);
    (*env)->SetIntArrayRegion(env, returnExceptFds, 0,
                              exceptfds.fd_count + 1, (jint *)&exceptfds);
#endif
    return 0;
}
```

- 执行 selector.select()，poll0函数把指向socket句柄和事件的内存地址传给底层函数。
- 1、如果之前没有发生事件，程序就阻塞在select处，当然不会一直阻塞，因为epoll在timeout时间内如果没有事件，也会返回；
- 2、一旦有对应的事件发生，poll0方法就会返回；
- 3、processDeregisterQueue方法会清理那些已经cancelled的SelectionKey；
- 4、updateSelectedKeys方法统计有事件发生的SelectionKey数量，并把符合条件发生事件的SelectionKey添加到selectedKeys哈希表中，提供给后续使用。
- 如何判断是否有事件发生？(native)
- poll0()会监听pollWrapper中的FD有没有数据进出，这会造成IO阻塞，直到有数据读写事件发生。
- 比如，由于pollWrapper中保存的也有ServerSocketChannel的FD，所以只要ClientSocket发一份数据到ServerSocket,那么poll0()就会返回；
- 又由于pollWrapper中保存的也有pipe的write端的FD，所以只要pipe的write端向FD发一份数据，也会造成poll0()返回；
- 如果这两种情况都没有发生，那么poll0()就一直阻塞，也就是selector.select()会一直阻塞；如果有任何一种情况发生，那么selector.select()就会返回。
- 在早期的JDK1.4和1.5 update10版本之前，Selector基于select/poll模型实现，是基于IO复用技术的非阻塞IO，不是异步IO。在JDK1.5 update10和linux core2.6以上版本，sun优化了Selctor的实现，底层使用epoll替换了select/poll。
- epoll是Linux下的一种IO多路复用技术，可以非常高效的处理数以百万计的socket句柄。
- 在Windows下是IOCP

## WindowsSelectorImpl.wakeup()

```
public Selector wakeup() {
    synchronized (interruptLock) {
        if (!interruptTriggered) {
            setWakeupSocket();
            interruptTriggered = true;
        }
    }
    return this;
}
```

```
private void setWakeupSocket() {
    setWakeupSocket0(wakeupSinkFd);
}
```

```
private native void setWakeupSocket0(int wakeupSinkFd);
```

```
Java_sun_nio_ch_WindowsSelectorImpl_setWakeupSocket0(JNIEnv *env, jclass
this,
                                                     jint scoutFd)
{
    /* Write one byte into the pipe */
    const char byte = 1;
    send(scoutFd, &byte, 1, 0);
}
```

- 这里完成了向最开始建立的pipe的sink端写入了一个字节，source文件描述符就会处于就绪状态，poll方法会返回，从而导致select方法返回。（原来自己建立一个socket链着自己另外一个socket就是为了这个目的)
-

# Java AIO 源码

## AsynchronousFileChannle（AIO,基于CompletionHandler回调)

- 在Java 7中，AsynchronousFileChannel被添加到Java NIO。AsynchronousFileChannel使读取数据，并异步地将数据写入文件成为可能。

### open

- Path path = Paths.get("data/test.xml");
- AsynchronousFileChannel fileChannel =
- AsynchronousFileChannel.open(path, StandardOpenOption.READ);

```
public static AsynchronousFileChannel open(Path file,
                                           Set<? extends OpenOption>
options,
                                           ExecutorService executor,
                                           FileAttribute<?>... attrs)
    throws IOException
{
    FileSystemProvider provider = file.getFileSystem().provider();
    return provider.newAsynchronousFileChannel(file, options, executor,
attrs);
}
```

- WindowsChannelFactory#newAsynchronousFileChannel
- static AsynchronousFileChannel newAsynchronousFileChannel(String pathForWindows,
  String pathToCheck,
  Set<? extends OpenOption> options,
  long pSecurityDescriptor,
  ThreadPool pool)
  throws IOException
  {
  Flags flags = Flags.toFlags(options);
  // Overlapped I/O required
  flags.overlapped = true;

```
    // default is reading
    if (!flags.read && !flags.write) {
        flags.read = true;
    }

    // validation
    if (flags.append)
        throw new UnsupportedOperationException("APPEND not allowed");

    // open file for overlapped I/O
    FileDescriptor fdObj;
    try {
        fdObj = open(pathForWindows, pathToCheck, flags, pSecurityDescriptor);
    } catch (WindowsException x) {
        x.rethrowAsIOException(pathForWindows);
        return null;
    }

    // create the AsynchronousFileChannel
    try {
        return WindowsAsynchronousFileChannelImpl.open(fdObj, flags.read, flags.write,
    pool);
    } catch (IOException x) {
        // IOException is thrown if the file handle cannot be associated
        // with the completion port. All we can do is close the file.
        long handle = fdAccess.getHandle(fdObj);
        CloseHandle(handle);
        throw x;
    }
```

- WindowsAsynchronousFileChannelImpl#open

```
public static AsynchronousFileChannel open(FileDescriptor fdo,
                                           boolean reading,
                                           boolean writing,
                                           ThreadPool pool)
    throws IOException
{
    Iocp iocp;
    boolean isDefaultIocp;
    if (pool == null) {
        iocp = DefaultIocpHolder.defaultIocp;
        isDefaultIocp = true;
    } else {
        iocp = new Iocp(null, pool).start();
        isDefaultIocp = false;
    }
    try {
        return new
            WindowsAsynchronousFileChannelImpl(fdo, reading, writing, iocp,
isDefaultIocp);
    } catch (IOException x) {
        // error binding to port so need to close it (if created for this
channel)
        if (!isDefaultIocp)
            iocp.implClose();
        throw x;
```

```
    }
}
```

### read

### write

-
-

# Netty NIO

- 基于这个语境，Netty目前的版本是没有把IO操作交过操作系统处理的，所以是属于同步的。如果别人说Netty是异步非阻塞，如果要深究，那真要看看Netty新的版本是否把IO操作交过操作系统处理，或者看看有否使用JDK1.7中的AIO API，否则他们说的异步其实是指客户端程序调用Netty的IO操作API“不停顿等待”。
- 很多人所讲的异步其实指的是编程模型上的异步（即回调），而非应用程序的异步。

# NIO与Epoll

- Linux2.6之后支持epoll
- windows支持select而不支持epoll
- 不同系统下nio的实现是不一样的，包括Sunos linux 和windows
- select的复杂度为O(N)
- select有最大fd限制，默认为1024
- 修改sys/select.h可以改变select的fd数量限制
  - epoll的事件模型，无fd数量限制，复杂度O(1),不需要遍历fd
-

# 1.17 动态代理

- 静态代理：代理类是在编译时就实现好的。也就是说 Java 编译完成后代理类是一个实际的 class 文件。
- 动态代理：代理类是在运行时生成的。也就是说 Java 编译完之后并没有实际的 class 文件，而是在运行时动态生成的类字节码，并加载到JVM中。
- JDK动态代理是由Java内部的反射机制+动态生成字节码来实现的，cglib动态代理底层则是借助asm来实现的。总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题)。还有一点必须注意：JDK动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，JDK动态代理不能应用。由此可以看出，JDK动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。
- 前者必须基于接口，后者不需要接口，是基于继承的，但是不能代理final类和final方法；
- JDK采用反射机制调用委托类的方法，CGLIB采用类似索引的方式直接调用委托类方法；
- 前者效率略低于后者效率，CGLIB效率略高（不是一定的)

# JDK动态代理 使用

- Proxy类（代理类)的设计用到代理模式的设计思想，Proxy类对象实现了代理目标的所有接口，并代替目标对象进行实际的操作。代理的目的是在目标对象方法的基础上作增强，这种增强的本质通常就是对目标对象的方法进行拦截。所以，Proxy应该包括一个方法拦截器，来指示当拦截到方法调用时作何种处理。InvocationHandler就是拦截器的接口。

- Proxy (代理) 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。
- 动态代理类（代理类)是一个实现在创建类时在运行时指定的接口列表的类 ，代理接口是代理类实现的一个接口。代理实例 是代理类的一个实例。
- 每个代理实例都有一个关联的调用处理程序对象，它可以实现接口 InvocationHandler。 (拦截器)
- 在Java中怎样实现动态代理呢?
- 第一步，我们要有一个接口，还要有一个接口的实现类，而这个实现类呢就是我们要代理的对象， 所谓代理呢也就是在调用实现类的方法时，可以在方法执行前后做额外的工作。
- 第二步，我们要自己写一个在代理类的方法要执行时，能够做额外工作的类（拦截器)，而这个类必须继承InvocationHandler接口， 为什么要继承它呢？因为代理类的实例在调用实现类的方法的时候，不会调用真正的实现类的这个方法， 而是转而调用这个类的invoke方法（继承时必须实现的方法)，在这个方法中你可以调用真正的实现类的这个方法。

# JDK动态代理 原理

- Proxy#newProxyInstance
- 会返回一个实现了指定接口的代理对象，对该对象的所有方法调用都会转发给InvocationHandler.invoke()方法。

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
    throws IllegalArgumentException
{
    Objects.requireNonNull(h);

    final Class<?>[] intfs = interfaces.clone();
    final SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
    }

    /*
     * Look up or generate the designated proxy class.
     */
```

```
// 生成代理类的class
    Class<?> cl = getProxyClass0(loader, intfs);

    /*
     * Invoke its constructor with the designated invocation handler.
     */
    try {
        if (sm != null) {
            checkNewProxyPermission(Reflection.getCallerClass(), cl);
        }
        // 获取代理对象的构造方法（也就是$Proxy0(InvocationHandler h))
        final Constructor<?> cons = cl.getConstructor(constructorParams);
        final InvocationHandler ih = h;
        if (!Modifier.isPublic(cl.getModifiers())) {
            AccessController.doPrivileged(new PrivilegedAction<Void>() {
                public Void run() {
                    cons.setAccessible(true);
                    return null;
```

```
                }
            });
        }
```

- // 生成代理类的实例并把InvocationHandlerImpl的实例传给它的构造方法
  return cons.newInstance(new Object[]{h});
  } catch (IllegalAccessException|InstantiationException e) {
  throw new InternalError(e.toString(), e);
  } catch (InvocationTargetException e) {
  Throwable t = e.getCause();
  if (t instanceof RuntimeException) {
  throw (RuntimeException) t;
  } else {
  throw new InternalError(t.toString(), t);
  }
  } catch (NoSuchMethodException e) {
  throw new InternalError(e.toString(), e);
  }
  }

## 1)getProxyClass0（生成代理类的class)

- 最终生成是通过ProxyGenerator的generateProxyClass方法实现的。

```
private static Class<?> getProxyClass0(ClassLoader loader,
                                       Class<?>... interfaces) {
    if (interfaces.length > 65535) {
        throw new IllegalArgumentException("interface limit exceeded");
    }

    // If the proxy class defined by the given loader implementing
    // the given interfaces exists, this will simply return the cached copy;
    // otherwise, it will create the proxy class via the ProxyClassFactory
    return proxyClassCache.get(loader, interfaces);
}
```

```
private static final WeakCache<ClassLoader, Class<?>[], Class<?>>
    proxyClassCache = new WeakCache<>(new KeyFactory(), new
ProxyClassFactory());
```

- - @param type of keys
- @param
  type of parameters
- @param type of values
  */
  final class WeakCache<K, P, V> {}

```
public V get(K key, P parameter) {
    Objects.requireNonNull(parameter);
```

```
    expungeStaleEntries();

    Object cacheKey = CacheKey.valueOf(key, refQueue);

    // lazily install the 2nd level valuesMap for the particular cacheKey
    ConcurrentMap<Object, Supplier<V>> valuesMap = map.get(cacheKey);
    if (valuesMap == null) {
        ConcurrentMap<Object, Supplier<V>> oldValuesMap
            = map.putIfAbsent(cacheKey,
                              valuesMap = new ConcurrentHashMap<>());
        if (oldValuesMap != null) {
            valuesMap = oldValuesMap;
        }
    }

    // create subKey and retrieve the possible Supplier<V> stored by that
    // subKey from valuesMap
    Object subKey = Objects.requireNonNull(subKeyFactory.apply(key,
parameter));
    Supplier<V> supplier = valuesMap.get(subKey);
    Factory factory = null;

    while (true) {
        if (supplier != null) {
            // supplier might be a Factory or a CacheValue<V> instance
```

- // supplier是Factory,这个类定义在WeakCache的内部。
  V value = supplier.get();
  if (value != null) {
  return value;
  }
  }
  // else no supplier in cache
  // or a supplier that returned null (could be a cleared CacheValue
  // or a Factory that wasn't successful in installing the CacheValue)

  // lazily construct a Factory
  if (factory == null) {
  factory = new Factory(key, parameter, subKey, valuesMap);
  }

  if (supplier == null) {
  supplier = valuesMap.putIfAbsent(subKey, factory);
  if (supplier == null) {
  // successfully installed Factory
  supplier = factory;
  }
  // else retry with winning supplier
  } else {
  if (valuesMap.replace(subKey, supplier, factory)) {
  // successfully replaced
  // cleared CacheEntry / unsuccessful Factory
  // with our Factory
  supplier = factory;
  } else {
  // retry with current supplier

supplier = valuesMap.get(subKey);
}
}
}
}

- Factory

```
private final class Factory implements Supplier<V> {

    private final K key;
    private final P parameter;
    private final Object subKey;
    private final ConcurrentMap<Object, Supplier<V>> valuesMap;

    Factory(K key, P parameter, Object subKey,
            ConcurrentMap<Object, Supplier<V>> valuesMap) {
        this.key = key;
        this.parameter = parameter;
        this.subKey = subKey;
        this.valuesMap = valuesMap;
    }

    @Override
    public synchronized V get() { // serialize access
        // re-check
        Supplier<V> supplier = valuesMap.get(subKey);
        if (supplier != this) {
            // something changed while we were waiting:
            // might be that we were replaced by a CacheValue
            // or were removed because of failure ->
            // return null to signal WeakCache.get() to retry
            // the loop
            return null;
        }
        // else still us (supplier == this)

        // create new value
```

- // 创建新的class
  V value = null;
  try {
  value = Objects.requireNonNull(valueFactory.apply(key, parameter));
  } finally {
  if (value == null) { // remove us on failure
  valuesMap.remove(subKey, this);
  }
  }
  // the only path to reach here is with non-null value
  assert value != null;

```
// wrap value with CacheValue (WeakReference)
CacheValue<V> cacheValue = new CacheValue<>(value);

// try replacing us with CacheValue (this should always succeed)
```

```
    if (valuesMap.replace(subKey, this, cacheValue)) {
        // put also in reverseMap
        reverseMap.put(cacheValue, Boolean.TRUE);
    } else {
        throw new AssertionError("Should not reach here");
    }

    // successfully replaced us with new CacheValue -> return the value
    // wrapped by it
    return value;
```

  }
  }

```
private static final class ProxyClassFactory
    implements BiFunction<ClassLoader, Class<?>[], Class<?>>
{
    // prefix for all proxy class names
    private static final String proxyClassNamePrefix = "$Proxy";

    // next number to use for generation of unique proxy class names
    private static final AtomicLong nextUniqueNumber = new AtomicLong();

    @Override
    public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

        Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>
(interfaces.length);
        for (Class<?> intf : interfaces) {
            /*
             * Verify that the class loader resolves the name of this
             * interface to the same Class object.
             */
            Class<?> interfaceClass = null;
            try {
                interfaceClass = Class.forName(intf.getName(), false,
loader);
            } catch (ClassNotFoundException e) {
            }
            if (interfaceClass != intf) {
                throw new IllegalArgumentException(
                    intf + " is not visible from class loader");
            }
            /*
             * Verify that the Class object actually represents an
             * interface.
             */
            if (!interfaceClass.isInterface()) {
                throw new IllegalArgumentException(
                    interfaceClass.getName() + " is not an interface");
            }
            /*
             * Verify that this interface is not a duplicate.
             */
            if (interfaceSet.put(interfaceClass, Boolean.TRUE) != null) {
```

```
                throw new IllegalArgumentException(
                    "repeated interface: " + interfaceClass.getName());
            }
        }

        String proxyPkg = null;     // package to define proxy class in
        int accessFlags = Modifier.PUBLIC | Modifier.FINAL;

        /*
         * Record the package of a non-public proxy interface so that the
         * proxy class will be defined in the same package.  Verify that
         * all non-public proxy interfaces are in the same package.
         */
        for (Class<?> intf : interfaces) {
            int flags = intf.getModifiers();
            if (!Modifier.isPublic(flags)) {
                accessFlags = Modifier.FINAL;
                String name = intf.getName();
                int n = name.lastIndexOf('.');
                String pkg = ((n == -1) ? "" : name.substring(0, n + 1));
                if (proxyPkg == null) {
                    proxyPkg = pkg;
                } else if (!pkg.equals(proxyPkg)) {
                    throw new IllegalArgumentException(
                        "non-public interfaces from different packages");
                }
            }
        }

        if (proxyPkg == null) {
            // if no non-public proxy interfaces, use com.sun.proxy package
            proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
        }

        /*
         * Choose a name for the proxy class to generate.
         */
        long num = nextUniqueNumber.getAndIncrement();
        String proxyName = proxyPkg + proxyClassNamePrefix + num;

        /*
         * Generate the specified proxy class.
         */
        byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
            proxyName, interfaces, accessFlags);
        try {
            return defineClass0(loader, proxyName,
                                proxyClassFile, 0, proxyClassFile.length);
        } catch (ClassFormatError e) {
            /*
             * A ClassFormatError here means that (barring bugs in the
             * proxy class generation code) there was some other
             * invalid aspect of the arguments supplied to the proxy
             * class creation (such as virtual machine limitations
             * exceeded).
             */
            throw new IllegalArgumentException(e.toString());
        }
```

```
    }
}
```

- 重点!

```
/**
 * Generate a proxy class given a name and a list of proxy interfaces.
 *
 * @param name        the class name of the proxy class
 * @param interfaces  proxy interfaces
 * @param accessFlags access flags of the proxy class
*/
public static byte[] generateProxyClass(final String name,
                                        Class<?>[] interfaces,
                                        int accessFlags)
{
    ProxyGenerator gen = new ProxyGenerator(name, interfaces, accessFlags);
    final byte[] classFile = gen.generateClassFile();

    if (saveGeneratedFiles) {
        java.security.AccessController.doPrivileged(
        new java.security.PrivilegedAction<Void>() {
            public Void run() {
                try {
                    int i = name.lastIndexOf('.');
                    Path path;
                    if (i > 0) {
                        Path dir = Paths.get(name.substring(0,
i).replace('.', File.separatorChar));
                        Files.createDirectories(dir);
                        path = dir.resolve(name.substring(i+1,
name.length()) + ".class");
                    } else {
                        path = Paths.get(name + ".class");
                    }
                    Files.write(path, classFile);
                    return null;
                } catch (IOException e) {
                    throw new InternalError(
                        "I/O exception saving generated file: " + e);
                }
            }
        });
    }

    return classFile;
}
```

- ProxyGenerator#generateClassFIle

```
/**
 * Generate a class file for the proxy class.  This method drives the
 * class file generation process.
 */
private byte[] generateClassFile() {
```

```
/* ============================================================
 * Step 1: Assemble ProxyMethod objects for all methods to
 * generate proxy dispatching code for.
 */

/*
 * Record that proxy methods are needed for the hashCode, equals,
 * and toString methods of java.lang.Object.  This is done before
 * the methods from the proxy interfaces so that the methods from
 * java.lang.Object take precedence over duplicate methods in the
 * proxy interfaces.
 */
addProxyMethod(hashCodeMethod, Object.class);
addProxyMethod(equalsMethod, Object.class);
addProxyMethod(toStringMethod, Object.class);

/*
 * Now record all of the methods from the proxy interfaces, giving
 * earlier interfaces precedence over later ones with duplicate
 * methods.
 */
for (Class<?> intf : interfaces) {
    for (Method m : intf.getMethods()) {
        addProxyMethod(m, intf);
    }
}

/*
 * For each set of proxy methods with the same signature,
 * verify that the methods' return types are compatible.
 */
for (List<ProxyMethod> sigmethods : proxyMethods.values()) {
    checkReturnTypes(sigmethods);
}

/* ============================================================
 * Step 2: Assemble FieldInfo and MethodInfo structs for all of
 * fields and methods in the class we are generating.
 */
try {
    methods.add(generateConstructor());

    for (List<ProxyMethod> sigmethods : proxyMethods.values()) {
        for (ProxyMethod pm : sigmethods) {

            // add static field for method's Method object
            fields.add(new FieldInfo(pm.methodFieldName,
                "Ljava/lang/reflect/Method;",
                 ACC_PRIVATE | ACC_STATIC));

            // generate code for proxy method and add it
            methods.add(pm.generateMethod());
        }
    }

    methods.add(generateStaticInitializer());

} catch (IOException e) {
```

```
            throw new InternalError("unexpected I/O Exception", e);
        }

        if (methods.size() > 65535) {
            throw new IllegalArgumentException("method limit exceeded");
        }
        if (fields.size() > 65535) {
            throw new IllegalArgumentException("field limit exceeded");
        }

        /* ============================================================
         * Step 3: Write the final class file.
         */

        /*
         * Make sure that constant pool indexes are reserved for the
         * following items before starting to write the final class file.
         */
        cp.getClass(dotToSlash(className));
        cp.getClass(superclassName);
        for (Class<?> intf: interfaces) {
            cp.getClass(dotToSlash(intf.getName()));
        }

        /*
         * Disallow new constant pool additions beyond this point, since
         * we are about to write the final constant pool table.
         */
        cp.setReadOnly();

        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);

        try {
            /*
             * Write all the items of the "ClassFile" structure.
             * See JVMS section 4.1.
             */
                                        // u4 magic;
            dout.writeInt(0xCAFEBABE);
                                        // u2 minor_version;
            dout.writeShort(CLASSFILE_MINOR_VERSION);
                                        // u2 major_version;
            dout.writeShort(CLASSFILE_MAJOR_VERSION);

            cp.write(dout);             // (write constant pool)

                                        // u2 access_flags;
            dout.writeShort(accessFlags);
                                        // u2 this_class;
            dout.writeShort(cp.getClass(dotToSlash(className)));
                                        // u2 super_class;
            dout.writeShort(cp.getClass(superclassName));

                                        // u2 interfaces_count;
            dout.writeShort(interfaces.length);
                                        // u2 interfaces[interfaces_count];
            for (Class<?> intf : interfaces) {
```

```
            dout.writeShort(cp.getClass(
                dotToSlash(intf.getName())));
        }

                                    // u2 fields_count;
        dout.writeShort(fields.size());
                                    // field_info fields[fields_count];
        for (FieldInfo f : fields) {
            f.write(dout);
        }

                                    // u2 methods_count;
        dout.writeShort(methods.size());
                                    // method_info methods[methods_count];
        for (MethodInfo m : methods) {
            m.write(dout);
        }

                                     // u2 attributes_count;
        dout.writeShort(0); // (no ClassFile attributes for proxy classes)

    } catch (IOException e) {
        throw new InternalError("unexpected I/O Exception", e);
    }

    return bout.toByteArray();
}
```

## 2)getConstructor（获取代理类的构造方法)

## 3)newInstance（初始化代理对象)

# CGLIB动态代理 使用

- CGLIB(Code Generation Library)是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过继承方式实现代理。
- CGLIB的核心类：
- net.sf.cglib.proxy.Enhancer – 主要的增强类
- net.sf.cglib.proxy.MethodInterceptor – 主要的方法拦截类，它是Callback接口的子接口，需要用户实现
- net.sf.cglib.proxy.MethodProxy – JDK的java.lang.reflect.Method类的代理类，可以方便的实现对源对象方法的调用,如使用：
- Object o = methodProxy.invokeSuper(proxy, args);//虽然第一个参数是被代理对象，也不会出现死循环的问题。
- net.sf.cglib.proxy.MethodInterceptor接口是最通用的回调（callback)类型，它经常被基于代理的AOP用来实现拦截（intercept)方法的调用。这个接口只定义了一个方法

```
public Object intercept(Object object, java.lang.reflect.Method method,
```

- Object[] args, MethodProxy proxy) throws Throwable;
- 第一个参数是代理对像，第二和第三个参数分别是拦截的方法和方法的参数。原来的方法可能通过使用java.lang.reflect.Method对象的一般反射调用，或者使用net.sf.cglib.proxy.MethodProxy对象调用。net.sf.cglib.proxy.MethodProxy通常被首选使用，因为它更快。

```
public class CglibProxy implements MethodInterceptor {
```

- @Override

```
    public Object intercept(Object o, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
```

- System.out.println("++++++before " + methodProxy.getSuperName() + "++++++");
- System.out.println(method.getName());
- Object o1 = methodProxy.invokeSuper(o, args);
- System.out.println("++++++before " + methodProxy.getSuperName() + "++++++");
- return o1;
- }
- }

```
public class Main {
```

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

- CglibProxy cglibProxy = new CglibProxy();
- 
- Enhancer enhancer = new Enhancer();
- enhancer.setSuperclass(UserServiceImpl.class);
- enhancer.setCallback(cglibProxy);
- 
- UserService o = (UserService)enhancer.create();
  - o.getName(1);
  - o.getAge(1);
- }
- }
- 我们通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法，在intercept()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；通过调用MethodProxy.invokeSuper()方法，我们将调用转发给原始对象，具体到本例，就是HelloConcrete的具体方法。CGLIG中MethodInterceptor的作用跟JDK代理中的InvocationHandler很类似，都是方法调用的中转站。
- 注意：对于从Object中继承的方法，CGLIB代理也会进行代理，如hashCode()、equals()、toString()等，但是getClass()、wait()等方法不会，因为它是final方法，CGLIB无法代理。
- 既然是继承就不得不考虑final的问题。我们知道final类型不能有子类，所以CGLIB不能代理final类型。
- final方法是不能重载的，所以也不能通过CGLIB代理，遇到这种情况不会抛异常，而是会跳过final方法只代理其他方法。

# CGLIB动态代理 原理

- 1、生成代理类Class的二进制字节码（基于ASM）；

- 2、通过 Class.forName加载二进制字节码，生成Class对象；
- 3、通过反射机制获取实例构造，并初始化代理类对象。
- 调用委托类的方法是使用invokeSuper

```
public Object invokeSuper(Object obj, Object[] args) throws Throwable {
    try {
        init();
        FastClassInfo fci = fastClassInfo;
        return fci.f2.invoke(fci.i2, obj, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
}
```

```
private static class FastClassInfo
{
    FastClass f1;
    FastClass f2;
    int i1;
    int i2;
}
```

- f1指向委托类对象，f2指向代理类对象
- i1是被代理的方法在对象中的索引位置
- i2是CGLIB$被代理的方法$0在对象中的索引位置

## FastClass实现机制

- FastClass其实就是对Class对象进行特殊处理，提出下标概念index，通过索引保存方法的引用信息，将原先的反射调用，转化为方法的直接调用，从而体现所谓的fast。

- 在FastTest中有两个方法， getIndex中对Test类的每个方法根据hash建立索引， invoke根据指定的索引，直接调用目标方法，避免了反射调用。
-

# 1.18 反射

- Java的动态性体现在：反射机制、动态执行脚本语言、动态操作字节码
- 反射：在运行时加载、探知、使用编译时未知的类。
- Class.forName使用的类加载器是调用者的类加载器

## Class

- 表示Java中的类型（class、interface、enum、annotation、primitive type、void)本身。
- 一个类被加载之后，JVM会创建一个对应该类的Class对象，类的整个结构信息会放在相应的Class对象中。
- 这个Class对象就像一个镜子一样，从中可以看到类的所有信息。
- 反射的核心就是Class

- 如果多次执行forName等加载类的方法，类只会被加载一次；一个类只会形成一个Class对象，无论执行多少次加载类的方法，获得的Class都是一样的。

# 用途

# 性能

- 反射带来灵活性的同时，也有降低程序执行效率的弊端
- setAccessible方法不仅可以标记某些私有的属性方法为可访问的属性方法，并且可以提高程序的执行效率
- 实际上是启用和禁用访问安全检查的开关。如果做检查就会降低效率；关闭检查就可以提高效率。
- 反射调用方法比直接调用要慢大约30倍，如果跳过安全检查的话比直接调用要慢大约7倍
- 开启和不开启安全检查对于反射而言可能会差4倍的执行效率。
- 为什么慢?
  - 1)验证等防御代码过于繁琐，这一步本来在link阶段，现在却在计算时进行验证
  - 2)产生很多临时对象，造成GC与计算时间消耗
  - 3)由于缺少上下文，丢失了很多运行时的优化，比如JIT(它可以看作JVM的重要评测标准之一)
- 当然，现代JVM也不是非常慢了，它能够对反射代码进行缓存以及通过方法计数器同样实现JIT优化，所以反射不一定慢。

# 实现

- 反射在Java中可以直接调用，不过最终调用的仍是native方法，以下为主流反射操作的实现。

## Class.forName的实现

- Class.forName可以通过包名寻找Class对象，比如Class.forName("java.lang.String")。
- 在JDK的源码实现中，可以发现最终调用的是native方法forName0()，它在JVM中调用的实际是FindClassFromCaller()，原理与ClassLoader的流程一样。

```
public static Class<?> forName(String className)
            throws ClassNotFoundException {
    Class<?> caller = Reflection.getCallerClass();
    return forName0(className, true, ClassLoader.getClassLoader(caller),
caller);
}
```

```
private static native Class<?> forName0(String name, boolean initialize,
                                        ClassLoader loader,
                                        Class<?> caller)
    throws ClassNotFoundException;
```

```
Java_java_lang_Class_forName0(JNIEnv *env, jclass this, jstring classname,
                              jboolean initialize, jobject loader, jclass
caller)
{
    char *clname;
```

```
    jclass cls = 0;
    char buf[128];
    jsize len;
    jsize unicode_len;

    if (classname == NULL) {
        JNU_ThrowNullPointerException(env, 0);
        return 0;
    }

    len = (*env)->GetStringUTFLength(env, classname);
    unicode_len = (*env)->GetStringLength(env, classname);
    if (len >= (jsize)sizeof(buf)) {
        clname = malloc(len + 1);
        if (clname == NULL) {
            JNU_ThrowOutOfMemoryError(env, NULL);
            return NULL;
        }
    } else {
        clname = buf;
    }
    (*env)->GetStringUTFRegion(env, classname, 0, unicode_len, clname);

    if (VerifyFixClassname(clname) == JNI_TRUE) {
        /* slashes present in clname, use name b4 translation for exception
*/
        (*env)->GetStringUTFRegion(env, classname, 0, unicode_len, clname);
        JNU_ThrowClassNotFoundException(env, clname);
        goto done;
    }

    if (!VerifyClassname(clname, JNI_TRUE)) {  /* expects slashed name */
        JNU_ThrowClassNotFoundException(env, clname);
        goto done;
    }

    cls = JVM_FindClassFromCaller(env, clname, initialize, loader, caller);

 done:
    if (clname != buf) {
        free(clname);
    }
    return cls;
}
```

- JVM_ENTRY(jclass, JVM_FindClassFromClass(JNIEnv *env, const char *name,
  jboolean init, jclass from))
  JVMWrapper2("JVM_FindClassFromClass %s", name);
  if (name == NULL || (int)strlen(name) > Symbol::max_length()) {
  // It's impossible to create this class;  the name cannot fit
  // into the constant pool.
  THROW_MSG_0(vmSymbols::java_lang_NoClassDefFoundError(), name);
  }
  TempNewSymbol h_name = SymbolTable::new_symbol(name, CHECK_NULL);
  oop from_class_oop = JNIHandles::resolve(from);

```
Klass from_class = (from_class_oop == NULL)
                   ? (Klass*)NULL
                   : java_lang_Class::as_Klass(from_class_oop);
oop class_loader = NULL;
oop protection_domain = NULL;
if (from_class != NULL) {
  class_loader = from_class->class_loader();
  protection_domain = from_class->protection_domain();
}
Handle h_loader(THREAD, class_loader);
Handle h_prot  (THREAD, protection_domain);
jclass result = find_class_from_class_loader(env, h_name, init, h_loader,
                                             h_prot, true, thread);

if (TraceClassResolution && result != NULL) {
  // this function is generally only used for class loading during verification.
  ResourceMark rm;
  oop from_mirror = JNIHandles::resolve_non_null(from);
  Klass* from_class = java_lang_Class::as_Klass(from_mirror);
  const char * from_name = from_class->external_name();

  oop mirror = JNIHandles::resolve_non_null(result);
  Klass* to_class = java_lang_Class::as_Klass(mirror);
  const char * to = to_class->external_name();
  tty->print("RESOLVE %s %s (verification)\n", from_name, to);
}

return result;
                                JVM_END
```

- 

## getDeclaredFields的实现

- 在JDK源码中，可以知道class.getDeclaredFields()方法实际调用的是native方法getDeclaredFields0()，它在JVM主要实现步骤如下：
  - 1)根据Class结构体信息，获取field_count与fields[]字段，这个字段早已在load过程中被放入了
  - 2)根据field_count的大小分配内存、创建数组
  - 3)将数组进行forEach循环，通过fields[]中的信息依次创建Object对象
  - 4)返回数组指针
- 主要慢在如下方面：
- 创建、计算、分配数组对象
- 对字段进行循环赋值

```
public Field[] getDeclaredFields() throws SecurityException {
    checkMemberAccess(Member.DECLARED, Reflection.getCallerClass(), true);
    return copyFields(privateGetDeclaredFields(false));
}
```

```
private Field[] privateGetDeclaredFields(boolean publicOnly) {
    checkInitted();
```

```
    Field[] res;
    ReflectionData<T> rd = reflectionData();
    if (rd != null) {
        res = publicOnly ? rd.declaredPublicFields : rd.declaredFields;
        if (res != null) return res;
    }
    // No cached value available; request value from VM
    res = Reflection.filterFields(this, getDeclaredFields0(publicOnly));
    if (rd != null) {
        if (publicOnly) {
            rd.declaredPublicFields = res;
        } else {
            rd.declaredFields = res;
        }
    }
    return res;
}
```

```
private static Field[] copyFields(Field[] arg) {
    Field[] out = new Field[arg.length];
    ReflectionFactory fact = getReflectionFactory();
    for (int i = 0; i < arg.length; i++) {
        out[i] = fact.copyField(arg[i]);
    }
    return out;
}
```

- 

## Method.invoke的实现

- 以下为无同步、无异常的情况下调用的步骤
  - 1)创建Frame
  - 2)如果对象flag为native，交给native_handler进行处理
  - 3)在frame中执行java代码
  - 4)弹出Frame
  - 5)返回执行结果的指针
- 主要慢在如下方面：
- 需要完全执行ByteCode而缺少JIT等优化
- 检查参数非常多，这些本来可以在编译器或者加载时完成

```
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
       InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz, obj, modifiers);
        }
    }
    MethodAccessor ma = methodAccessor;             // read volatile
```

```
    if (ma == null) {
        ma = acquireMethodAccessor();
    }
    return ma.invoke(obj, args);
}
```

- NativeMethodAccessorImpl#invoke

```
public Object invoke(Object obj, Object[] args)
    throws IllegalArgumentException, InvocationTargetException
{
    // We can't inflate methods belonging to vm-anonymous classes because
    // that kind of class can't be referred to by name, hence can't be
    // found from the generated bytecode.
    if (++numInvocations > ReflectionFactory.inflationThreshold()
            && !ReflectUtil.isVMAnonymousClass(method.getDeclaringClass()))
{
        MethodAccessorImpl acc = (MethodAccessorImpl)
            new MethodAccessorGenerator().
                generateMethod(method.getDeclaringClass(),
                               method.getName(),
                               method.getParameterTypes(),
                               method.getReturnType(),
                               method.getExceptionTypes(),
                               method.getModifiers());
        parent.setDelegate(acc);
    }

    return invoke0(method, obj, args);
}
```

```
private static native Object invoke0(Method m, Object obj, Object[] args);
```

- Java_sun_reflect_NativeMethodAccessorImpl_invoke0
  (JNIEnv *env, jclass unused, jobject m, jobject obj, jobjectArray args)
  {
   return JVM_InvokeMethod(env, m, obj, args);
  }
- JVM_ENTRY(jobject, JVM_InvokeMethod(JNIEnv *env, jobject method, jobject obj,
  jobjectArray args0))
  JVMWrapper("JVM_InvokeMethod");
  Handle method_handle;
  if (thread->stack_available((address) &method_handle) >= JVMInvokeMethodSlack) {
   method_handle = Handle(THREAD, JNIHandles::resolve(method));
   Handle receiver(THREAD, JNIHandles::resolve(obj));
   objArrayHandle args(THREAD, objArrayOop(JNIHandles::resolve(args0)));
   oop result = Reflection::invoke_method(method_handle(), receiver, args, CHECK_NULL);
   jobject res = JNIHandles::make_local(env, result);
   if (JvmtiExport::should_post_vm_object_alloc()) {
    oop ret_type = java_lang_reflect_Method::return_type(method_handle());
    assert(ret_type != NULL, "sanity check: ret_type oop must not be NULL!");

```
      if (java_lang_Class::is_primitive(ret_type)) {
        // Only for primitive type vm allocates memory for java object.
        // See box() method.
        JvmtiExport::post_vm_object_alloc(JavaThread::current(), result);
      }
    }
    return res;
  } else {
    THROW_0(vmSymbols::java_lang_StackOverflowError());
  }
  JVM_END
```

## class.newInstance的实现

```
- 1)检测权限、预分配空间大小等参数
- 2)创建Object对象，并分配空间
- 3)通过Method.invoke调用构造函数(<init>())
- 4)返回Object指针
```

- 主要慢在如下方面：
- 参数检查不能优化或者遗漏
- ()的查表
- Method.invoke本身耗时

```
public T newInstance()
    throws InstantiationException, IllegalAccessException
{
    if (System.getSecurityManager() != null) {
        checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(),
false);
    }

    // NOTE: the following code may not be strictly correct under
    // the current Java memory model.

    // Constructor lookup
    if (cachedConstructor == null) {
        if (this == Class.class) {
            throw new IllegalAccessException(
                "Can not call newInstance() on the Class for
java.lang.Class"
            );
        }
        try {
            Class<?>[] empty = {};
            final Constructor<T> c = getConstructor0(empty,
Member.DECLARED);
            // Disable accessibility checks on the constructor
            // since we have to do the security check here anyway
            // (the stack depth is wrong for the Constructor's
            // security check to work)
            java.security.AccessController.doPrivileged(
                new java.security.PrivilegedAction<Void>() {
                    public Void run() {
```

```
                            c.setAccessible(true);
                            return null;
                        }
                    });
            cachedConstructor = c;
        } catch (NoSuchMethodException e) {
            throw (InstantiationException)
                new InstantiationException(getName()).initCause(e);
        }
    }
    Constructor<T> tmpConstructor = cachedConstructor;
    // Security check (same as in java.lang.reflect.Constructor)
    int modifiers = tmpConstructor.getModifiers();
    if (!Reflection.quickCheckMemberAccess(this, modifiers)) {
        Class<?> caller = Reflection.getCallerClass();
        if (newInstanceCallerCache != caller) {
            Reflection.ensureMemberAccess(caller, this, null, modifiers);
            newInstanceCallerCache = caller;
        }
    }
    // Run constructor
    try {
        return tmpConstructor.newInstance((Object[])null);
    } catch (InvocationTargetException e) {
        Unsafe.getUnsafe().throwException(e.getTargetException());
        // Not reached
        return null;
    }
}
```

- 

# 1.19 XML

## DOM

- OM是用与平台和语言无关的方式表示XML文档的官方W3C标准。DOM是以层次结构组织的节点或信息片断的集合。这个层次结构允许开发人员在树中寻找特定信息。分析该结构通常需要加载整个文档和构造层次结构，然后才能做任何工作。由于它是基于信息层次的，因而DOM被认为是基于树或基于对象的。
- 优点
- ①允许应用程序对数据和结构做出更改。
- ②访问是双向的，可以在任何时候在树中上下导航，获取和操作任意部分的数据。
- 缺点
- ①通常需要加载整个XML文档来构造层次结构，消耗资源大。
- 

## SAX

- SAX处理的优点非常类似于流媒体的优点。分析能够立即开始，而不是等待所有的数据被处理。而且，由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。这对于大型文档来说是个巨大的优点。事实上，应用程序甚至不必解析整个文档；它可以在某个条件得到满足时停止解析。一般来说，SAX还比它的替代者DOM快许多。
- 优点
- ①不需要等待所有数据都被处理，分析就能立即开始。

- ②只在读取数据时检查数据，不需要保存在内存中。
- ③可以在某个条件得到满足时停止解析，不必解析整个文档。
- ④效率和性能较高，能解析大于系统内存的文档。
- 缺点
- ①需要应用程序自己负责TAG的处理逻辑（例如维护父/子关系等），文档越复杂程序就越复杂。
- ②单向导航，无法定位文档层次，很难同时访问同一文档的不同部分数据，不支持XPath。
- JDOM
- DOM4J
-

# 1.20 Java8

## Lambda表达式&函数式接口&方法引用&Stream API

- Java8 stream迭代的优势和区别；lambda表达式？为什么要引入它
  - 1)流（高级Iterator）：对集合对象进行各种非常便利、高效的聚合操作（aggregate operation），或者大批量数据操作 (bulk data operation)，隐式迭代等，代码简洁
  - 2)方便地实现并行（并行流），比如实现MapReduce
  - 3)Lamdba：简化匿名内部类的实现，代码更加紧凑
  - 4)方法引用：方法引用是lambda表达式的另一种表达方式
- 对象::实例方法
- 类::静态方法
- 类::实例方法名

## Optional

- Optional仅仅是一个容易：存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检查。

## CompletableFuture

```
- 1)实现异步API（将任务交给另一线程完成，该线程与调用方异步，通过回调函数或阻塞的方式取得任务结果）
- 2)将批量同步操作转为异步操作（并行流/CompletableFuture）
- 3)多个异步任务合并
```

## 时间日期API

- 新的java.time包包含了所有关于日期、时间、时区、Instant（跟日期类似但是精确到纳秒)、duration（持续时间)和时钟操作的类。新设计的API认真考虑了这些类的不变性（从java.util.Calendar吸取的教训)，如果某个实例需要修改，则返回一个新的对象。

## 接口中的默认方法与静态方法

- 默认方法使得开发者可以在不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不强制那些实现了该接口的类也同时实现这个新加的方法。
- 默认方法允许在不打破现有继承体系的基础上改进接口。该特性在官方库中的应用是：给java.util.Collection接口添加新方法，如stream()、parallelStream()、forEach()和removeIf()等等。
-

# 1.21 Java9

## 模块化

- 提供了类似于OSGI框架的功能，模块之间存在相互的依赖关系，可以导出一个公共的API，并且隐藏实现的细节，Java提供该功能的主要的动机在于，减少内存的开销，在JVM启动的时候，至少会有30～60MB的内存加载，主要原因是JVM需要加载rt.jar，不管其中的类是否被classloader加载，第一步整个jar都会被JVM加载到内存当中去，模块化可以根据模块的需要加载程序运行需要的class。
- 在引入了模块系统之后，JDK 被重新组织成 94 个模块。Java 应用可以通过新增的 jlink 工具，创建出只包含所依赖的 JDK 模块的自定义运行时镜像。这样可以极大的减少 Java 运行时环境的大小。使得JDK可以在更小的设备中使用。采用模块化系统的应用程序只需要这些应用程序所需的那部分JDK模块，而非是整个JDK框架了。

## HTTP/2

- Java 9的版本中引入了一个新的package:java.net.http，里面提供了对Http访问很好的支持，不仅支持Http1.1而且还支持HTTP/2，以及WebSocket，据说性能特别好。

## JShell

- java9引入了jshell这个交互性工具，让Java也可以像脚本语言一样来运行，可以从控制台启动 jshell ，在 jshell 中直接输入表达式并查看其执行结果。当需要测试一个方法的运行效果，或是快速的对表达式进行求值时，jshell 都非常实用。
- 除了表达式之外，还可以创建 Java 类和方法。jshell 也有基本的代码完成功能。

## 不可变集合工厂方法

- Java 9增加了List.of()、Set.of()、Map.of()和Map.ofEntries()等工厂方法来创建不可变集合。

## 私有接口方法

- Java 8 为我们提供了接口的默认方法和静态方法，接口也可以包含行为，而不仅仅是方法定义。
- 默认方法和静态方法可以共享接口中的私有方法，因此避免了代码冗余，这也使代码更加清晰。如果私有方法是静态的，那这个方法就属于这个接口的。并且没有静态的私有方法只能被在接口中的实例调用。

## 多版本兼容 JAR

> – 当一个新版本的 Java 出现的时候，你的库用户要花费很长时间才会切换到这个新的版本。这就意味着库要去向后兼容你想要支持的最老的 Java 版本（许多情况下就是 Java 6 或者 7）。这实际上意味着未来的很长一段时间，你都不能在库中运用 Java 9 所提供的新特性。幸运的是，多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本。

## 统一 JVM 日志

- Java 9 中，JVM 有了统一的日志记录系统，可以使用新的命令行选项-Xlog 来控制 JVM 上 所有组件的日志记录。该日志记录系统可以设置输出的日志消息的标签、级别、修饰符和输出目标等。

## 垃圾收集机制

- Java 9 移除了在 Java 8 中 被废弃的垃圾回收器配置组合，同时把G1设为默认的垃圾回收器实现。替代了之前默认使用的Parallel GC，对于这个改变，evens的评论是酱紫的：这项变更是很重要的，因为相对于Parallel来说，G1会在应用线程上做更多的事情，而Parallel几乎没有在应用线程上做任何事情，它基本上完全依赖GC线程完成所有的内存管理。这意味着切换到G1将会为应用线程带来额外的工作，从而直接影响到应用的性能

# I/O 流新特性

- java.io.InputStream 中增加了新的方法来读取和复制 InputStream 中包含的数据。
  - readAllBytes：读取 InputStream 中的所有剩余字节。
  - readNBytes： 从 InputStream 中读取指定数量的字节到数组中。
  - transferTo：读取 InputStream 中的全部字节并写入到指定的 OutputStream 中 。
-