* 本github最初的版本是一份word文档，目前只是把word刚刚搬上来了，但是有些图片、排版还没来得急整理，看起来可能还是有点困难
* 所以可以先关注一下我的公众号，在我的公众号后台回复 **888** 获取这个github仓库的PDF版本，左侧有导航栏，方便大家阅读。



# 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

* 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

* 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。

### 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。

## 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

* 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。

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
* 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。

### + 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。
* 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。

### 编译优化

* 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。
* 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。

## 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

* 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)

* 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)

* 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);  
  }

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。

## Iterator / ListIterator / Iterable

* 普通for循环时不能删除元素，否则会抛出异常；Iterator可以

public interface Collection<E> extends Iterable<E> {}

* Collection接口继承了Iterable，Iterable接口定义了iterator抽象方法和forEach default方法。所以ArrayList、LinkedList都可以使用迭代器和forEach，包括增强for循环（编译时转为迭代器)。

public interface Iterable<T> {  
 Iterator<T> iterator();  
 default void forEach(Consumer<? super T> action) {  
 Objects.requireNonNull(action);  
 for (T t : this) {  
 action.accept(t);  
 }  
 }

* default Spliterator spliterator() {  
   return Spliterators.spliteratorUnknownSize(iterator(), 0);  
  }
* }
* 注意这些具体的容器类返回的迭代器对象是各不相同的，主要是因为不同的容器遍历方式不同，但是这些迭代器对象都实现Iterator接口，都可以使用一个Iterator对象来统一指向这些不同的子类对象。
* ArrayList#iterator

public Iterator<E> iterator() {  
 return new Itr();  
}

* ArrayList#Itr

private class Itr implements Iterator<E> {  
 int cursor; // index of next element to return  
 int lastRet = -1; // index of last element returned; -1 if no such  
 int expectedModCount = modCount;  
  
 public boolean hasNext() {  
 return cursor != size;  
 }  
  
 @SuppressWarnings("unchecked")  
 public E next() {  
 checkForComodification();  
 int i = cursor;  
 if (i >= size)  
 throw new NoSuchElementException();  
 Object[] elementData = ArrayList.this.elementData;  
 if (i >= elementData.length)  
 throw new ConcurrentModificationException();  
 cursor = i + 1;  
 return (E) elementData[lastRet = i];  
 }  
  
 public void remove() {  
 if (lastRet < 0)  
 throw new IllegalStateException();  
 checkForComodification();  
  
 try {  
 ArrayList.this.remove(lastRet);  
 cursor = lastRet;  
 lastRet = -1;  
 expectedModCount = modCount;  
 } catch (IndexOutOfBoundsException ex) {  
 throw new ConcurrentModificationException();  
 }  
 }  
  
 @Override  
 @SuppressWarnings("unchecked")  
 public void forEachRemaining(Consumer<? super E> consumer) {  
 Objects.requireNonNull(consumer);  
 final int size = ArrayList.this.size;  
 int i = cursor;  
 if (i >= size) {  
 return;  
 }  
 final Object[] elementData = ArrayList.this.elementData;  
 if (i >= elementData.length) {  
 throw new ConcurrentModificationException();  
 }  
 while (i != size && modCount == expectedModCount) {  
 consumer.accept((E) elementData[i++]);  
 }  
 // update once at end of iteration to reduce heap write traffic  
 cursor = i;  
 lastRet = i - 1;  
 checkForComodification();  
 }  
  
 final void checkForComodification() {  
 if (modCount != expectedModCount)  
 throw new ConcurrentModificationException();  
 }  
}

* ArrayList#listIterator

public ListIterator<E> listIterator() {  
 return new ListItr(0);  
}

* ArrayList#ListItr

private class ListItr extends Itr implements ListIterator<E> {  
 ListItr(int index) {  
 super();  
 cursor = index;  
 }  
  
 public boolean hasPrevious() {  
 return cursor != 0;  
 }  
  
 public int nextIndex() {  
 return cursor;  
 }  
  
 public int previousIndex() {  
 return cursor - 1;  
 }  
  
 @SuppressWarnings("unchecked")  
 public E previous() {  
 checkForComodification();  
 int i = cursor - 1;  
 if (i < 0)  
 throw new NoSuchElementException();  
 Object[] elementData = ArrayList.this.elementData;  
 if (i >= elementData.length)  
 throw new ConcurrentModificationException();  
 cursor = i;  
 return (E) elementData[lastRet = i];  
 }  
  
 public void set(E e) {  
 if (lastRet < 0)  
 throw new IllegalStateException();  
 checkForComodification();  
  
 try {  
 ArrayList.this.set(lastRet, e);  
 } catch (IndexOutOfBoundsException ex) {  
 throw new ConcurrentModificationException();  
 }  
 }  
  
 public void add(E e) {  
 checkForComodification();  
  
 try {  
 int i = cursor;  
 ArrayList.this.add(i, e);  
 cursor = i + 1;  
 lastRet = -1;  
 expectedModCount = modCount;  
 } catch (IndexOutOfBoundsException ex) {  
 throw new ConcurrentModificationException();  
 }  
 }  
}

## for /增强for/ forEach

For-each loop Equivalent for loop  
for (type var : arr) {  
 body-of-loop  
} for (int i = 0; i < arr.length; i++) {   
 type var = arr[i];  
 body-of-loop  
}  
for (type var : coll) {  
 body-of-loop  
} for (Iterator iter = coll.iterator(); iter.hasNext(); ) {  
 type var = iter.next();  
 body-of-loop  
}

* 增强for循环在编译时被修改为for循环：数组会被修改为下标式的循环；集合会被修改为Iterator循环。
* 增强for循环不适合以下情况：（过滤、转换、平行迭代)
* 对collection或数组中的元素不能做赋值操作；
* 只能正向遍历，不能反向遍历；
* 遍历过程中，collection或数组中同时只有一个元素可见，即只有“当前遍历到的元素”可见，而前一个或后一个元素是不可见的；
* forEach
* ArrayList#forEach继承自
* Iterable接口的default方法
* default void forEach(Consumer<? super T> action) {  
   Objects.requireNonNull(action);  
   for (T t : this) {  
   action.accept(t);  
   }  
  }

## Comparable与Comparator

* 基本数据类型包装类和String类均已实现了Comparable接口。
* 实现了Comparable接口的类的对象的列表或数组可以通过Collections.sort或Arrays.sort进行自动排序，默认为升序。
* 可以将 Comparator 传递给 sort 方法（如 Collections.sort 或 Arrays.sort)，从而允许在排序顺序上实现精确控制。还可以使用 Comparator 来控制某些数据结构（如TreeSet,TreeMap)的顺序。

# 1.9 继承

子类继承父类所有的成员变量（即使是private变量，有所有权，但是没有使用权，不能访问父类的private的成员变量)。

子类中可以直接调用父类非private方法，也可以用super.父类方法的形式调用。

* 子类构造方法中如果没有显式使用super(父类构造方法参数)去构造父类对象的话（如果有必须是方法的第一行)，编译器会在第一行添加super()。
* 子类的构造函数可否不使用super(父类构造方法参数)调用超类的构造方法？
* 可以不用显式的写出super，但前提是“父类中有多个构造方法，且有一个是显式写出的无参的构造方法”。

# 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！

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;  
 }  
 };  
 }  
}

# 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("JVMFindClassFromClass %s", name);*  
  *if (name == NULL || (int)strlen(name) > Symbol::maxlength()) {*  
   *// It's impossible to create this class; the name cannot fit*  
   *// into the constant pool.*  
   *THROWMSG0(vmSymbols::javalangNoClassDefFoundError(), name);*  
  *}*  
  *TempNewSymbol hname = SymbolTable::newsymbol(name, CHECKNULL);*  
  *oop fromclass\_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 。