**编码质量**

## 第一章 Java开发中通用的方法和准则

建议1：不要在常量和变量中出现易混淆的字母；

（i、l、1；o、0等）。

建议2：莫让常量蜕变成变量；

（代码运行工程中不要改变常量值）。

建议3：三元操作符的类型务必一致；

（不一致会导致自动类型转换，类型提升int->float->double等）。

建议4：避免带有变长参数的方法重载；

（变长参数的方法重载之后可能会包含原方法）。

建议5：别让null值和空值威胁到变长方法；

（两个都包含变长参数的重载方法，当变长参数部分空值，或者为null值时，重载方法不清楚会调用哪一个方法）。

建议6：覆写变长方法也循规蹈矩；

（变长参数与数组，覆写的方法参数与父类相同，不仅仅是类型、数量，还包括显示形式）。

建议7：警惕自增的陷阱；

（count=count++;操作时JVM首先将count原值拷贝到临时变量区，再执行count加1，之后再将临时变量区的值赋给count，所以count一直为0或者某个初始值。C++中count=count++;与count++等效，而PHP与Java类似）。

建议8：不要让旧语法困扰你；

（Java中抛弃了C语言中的goto语法，但是还保留了该关键字，只是不进行语义处理，const关键中同样类似）。

建议9：少用静态导入；

（Java5引入的静态导入语法import static，使用静态导入可以减少程序字符输入量，但是会带来很多代码歧义，省略的类约束太少，显得程序晦涩难懂）。

建议10：不要在本类中覆盖静态导入的变量和方法；

（例如静态导入Math包下的PI常量，类属性中又定义了一个同样名字PI的常量。编译器的“最短路径原则”将会选择使用本类中的PI常量。本类中的属性，方法优先。如果要变更一个被静态导入的方法，最好的办法是在原始类中重构，而不是在本类中覆盖）。

建议11：养成良好的习惯，显式声明UID；

（显式声明serialVersionUID可以避免序列化和反序列化中对象不一致，JVM根据serialVersionUID来判断类是否发生改变。隐式声明由编译器在编译的时候根据包名、类名、继承关系等诸多因子计算得出，极其复杂，算出的值基本唯一）。

建议12：避免用序列化类在构造函数中为不变量赋值；

（在序列化类中，不适用构造函数为final变量赋值）（序列化规则1：如果final属性是一个直接量，在反序列化时就会重新计算；序列化规则2：反序列化时构造函数不会执行；反序列化执行过程：JVM从数据流中获取一个Object对象，然后根据数据流中的类文件描述信息（在序列化时，保存到磁盘的对象文件中包含了类描述信息，不是类）查看，发现是final变量，需要重新计算，于是引用Person类中的name值，而此时JVM又发现name没有赋值（因为反序列化时构造函数不会执行），不能引用，于是它不再初始化，保持原始值状态。整个过程中需要保持serialVersionUID相同）。

建议13：避免为final变量复杂赋值；

（类序列化保存到磁盘上（或网络传输）的对象文件包括两部分：1、类描述信息：包括包路径、继承关系等。注意，它并不是class文件的翻版，不记录方法、构造函数、static变量等的具体实现。2、非瞬态（transient关键字）和非静态（static关键字）的实例变量值。总结：反序列化时final变量在以下情况下不会被重新赋值：1、通过构造函数为final变量赋值；2、通过方法返回值为final变量赋值；3、final修饰的属性不是基本类型）。

建议14：使用序列化类的私有方法巧妙解决“部分属性持久化问题”；

（部分属性持久化问题解决方案1：把不需要持久化的属性加上瞬态关键字（transient关键字）即可，但是会使该类失去了分布式部署的功能。方案2：新增业务对象。方案3：请求端过滤。方案4：变更传输契约，即覆写writeObject和readObject私有方法，在两个私有方法体内完成部分属性持久化）。

建议15：break万万不可忘；

（switch语句中，每一个case匹配完都需要使用break关键字跳出，否则会依次执行完所有的case内容。）。

建议16：易变业务使用脚本语言编写；

（脚本语言：都是在运行期解释执行。脚本语言三大特性：1、灵活：动态类型；2、便捷：解释型语言，不需要编译成二进制，不需要像Java一样生成字节码，依靠解释执行，做到不停止应用变更代码；3、简单：部分简单。Java使用ScriptEngine执行引擎来执行JavaScript脚本代码）。

建议17：慎用动态编译；

（好处：更加自如地控制编译过程。很少使用，原因：静态编译能够完成大部分工作甚至全部，即使需要使用，也有很好的替代方案，如JRuby、Groovy等无缝的脚本语言。动态编译注意以下4点：1、在框架中谨慎使用：debug困难，成本大；2、不要在要求高性能的项目中使用：需要一个编译过程，比静态编译多了一个执行环节；3、动态编译要考虑安全问题：防止恶意代码；4、记录动态编译过程）。

建议18：避免instanceof非预期结果；

（instanceof用来判断一个对象是否是一个类的实例，只能用于对象的判断，不能用于基本类型的判断（编译不通过），instanceof操作符的左右操作数必须有继承或实现关系，否则编译会失败。例：null instanceof String返回值是false，instanceof特有规则，若左操作数是null，结果就直接返回false，不再运算右操作数是什么类）。

建议19：断言绝对不是鸡肋；

（防御式编程中经常使用断言（Assertion）对参数和环境做出判断。断言是为调试程序服务的。两个特性：1、默认assert不启用；2、assert抛出的异常AssertionError是继承自Error的）。

建议20：不要只替换一个类；

（发布应用系统时禁止使用类文件替换方式，整体WAR包发布才是完全之策）（Client类中调用了Constant类中的属性值，如果更改了Constant常量类属性的值，重新编译替换。而不改变或者替换Client类，则Client中调用的Constant常量类的属性值并不会改变。原因：对于final修饰的基本类型和String类型，编译器会认为它是稳定态（Immutable Status），所以在编译时就直接把值编译到字节码中了，避免了再运行期引用，以提高代码的执行效率。而对于final修饰的类（即非基本类型），编译器认为它是不稳定态（Mutable Status），在编译时建立的则是引用关系（该类型也叫作Soft Final），如果Client类引入的常量是一个类或实例，即使不重新编译也会输出最新值）。

## 第二章 基本类型

建议21：用偶判断，不用奇判断；

（不要使用奇判断（i%2 == 1 ? "奇数" : "偶数"），使用偶判断（i%2 == 0 ? "偶数" : "奇数"）。原因Java中的取余（%标识符）算法：测试数据输入1 2 0 -1 -2，奇判断的时候，当输入-1时，也会返回偶数。

在CODE上查看代码片派生到我的代码片

//模拟取余计算，dividend被除数，divisor除数

public static int remainder(int dividend, int divisor) {

return dividend - dividend / divisor \* divisor;

}

）。

建议22：用整数类型处理货币；

（不要使用float或者double计算货币，因为在计算机中浮点数“有可能”是不准确的，它只能无限接近准确值，而不能完全精确。不能使用计算机中的二进制位来表示如0.4等的浮点数。解决方案：1、使用BigDecimal（优先使用）；2、使用整型）。

建议23：不要让类型默默转换；

（基本类型转换时，使用主动声明方式减少不必要的Bug）

在CODE上查看代码片派生到我的代码片

public static final int LIGHT\_SPEED = 30 \* 10000 \* 1000;

long dis2 = LIGHT\_SPEED \* 60 \* 8;

以上两句在参与运算时会溢出，因为Java是先运算后再进行类型转换的。因为dis2的三个运算参数都是int类型，三者相乘的结果也是int类型，但是已经超过了int的最大值，所以越界了。解决方法，在运算参数60后加L即可。

建议24：边界、边界、还是边界；

（数字越界是检验条件失效，边界测试；检验条件if(order>0 && order+cur<=LIMIT)，输入的数大于0，加上cur的值之后溢出为负值，小于LIMIT，所以满足条件，但不符合要求）。

建议25：不要让四舍五入亏了一方；

（Math.round(10.5）输出结果11；Math.round(-10.5)输出结果-10。这是因为Math.round采用的舍入规则所决定的（采用的是正无穷方向舍入规则），根据不同的场景，慎重选择不同的舍入模式，以提高项目的精准度，减少算法损失）。

建议26：提防包装类型的null值；

（泛型中不能使用基本类型，只能使用包装类型，null执行自动拆箱操作会抛NullPointerException异常，因为自动拆箱是通过调用包装对象的intValue方法来实现的，而访问null的intValue方法会报空指针异常。谨记一点：包装类参与运算时，要做null值校验，即（i!=null ? i : 0））。

建议27：谨慎包装类型的大小比较；

（大于>或者小于<比较时，包装类型会调用intValue方法，执行自动拆箱比较。而==等号用来判断两个操作数是否有相等关系的，如果是基本类型则判断数值是否相等，如果是对象则判断是否是一个对象的两个引用，也就是地址是否相等。通过两次new操作产生的两个包装类型，地址肯定不相等）。

建议28：优先使用整型池；

（自动装箱是通过调用valueOf方法来实现的，包装类的valueOf生成包装实例可以显著提高空间和时间性能）valueOf方法实现源码：

在CODE上查看代码片派生到我的代码片

public static Integer valueOf(int i) {

final int offset = 128;

if (i >= -128 && i <=127) {

return IntegerCache.cache[i + offset];

}

return new Integer(i);

}

class IntegerCache {

static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static {

for (int i = 0; i < cache.length; i++)

cache[i] = new Integer(i - 128);

}

}

cache是IntegerCache内部类的一个静态数组，容纳的是-128到127之间的Integer对象。通过valueOf产生包装对象时，如果int参数在-128到127之间，则直接从整型池中获得对象，不在该范围的int类型则通过new生成包装对象。在判断对象是否相等的时候，最好是利用equals方法，避免“==”产生非预期结果。

建议29：优先选择基本类型；

（int参数先加宽转变成long型，然后自动转换成Long型。Integer.valueOf(i)参数先自动拆箱转变为int类型，与之前类似）。

建议30：不要随便设置随机种子；

（若非必要，不要设置随机数种子）（Random r = new Random(1000);该代码中1000即为随机种子。在同一台机器上，不管运行多少次，所打印的随机数都是相同的。在Java中，随机数的产生取决于种子，随机数和种子之间的关系遵从以下两个规则：1、种子不同，产生不同的随机数；2、种子相同，即使实例不同也产生相同的随机数。Random类默认种子（无参构造）是System.nanoTime()的返回值，这个值是距离某一个固定时间点的纳秒数，所以可以产生随机数。java.util.Random类与Math.random方法原理相同）。

## 第三章 类、对象及方法

建议31：在接口中不要存在实现代码；

（可以通过在接口中声明一个静态常量s，其值是一个匿名内部类的实例对象，可以实现接口中存在实现代码）。

建议32：静态变量一定要先声明后赋值；

（也可以先使用后声明，因为静态变量是类初始化时首先被加载，JVM会去查找类中所有的静态声明，然后分配空间，分配到数据区（Data Area）的，它在内存中只有一个拷贝，不会被分配多次，注意这时候只是完成了地址空间的分配还没有赋值，之后JVM会根据类中静态赋值（包括静态类赋值和静态块赋值）的先后顺序来执行，后面的操作都是地址不变，值改变）。

建议33：不要覆写静态方法；

（一个实例对象有两个类型：表面类型和实际类型，表面类型是声明时的类型，实际类型是对象产生时的类型。对于非静态方法，它是根据对象的实际类型来执行的，即执行了覆写方法。而对于静态方法，首先静态方法不依赖实例对象，通过类名访问；其次，可以通过对象访问静态方法，如果通过对象访问，JVM则会通过对象的表面类型查找到静态方法的入口，继而执行）。

建议34：构造函数尽量简化；

（通过new关键字生成对象时必然会调用构造函数。子类实例化时，首先会初始化父类（注意这里是初始化，可不是生成父类对象），也就是初始化父类的变量，调用父类的构造函数，然后才会初始化子类的变量，调用子类自己的构造函数，最后生成一个实例对象。构造函数太复杂有可能造成，对象使用时还没完成初始化）。

建议35：避免在构造函数中初始化其他类；

（有可能造成不断的new新对象的死循环，直到栈内存被消耗完抛出StackOverflowError异常为止）。

建议36：使用构造代码块精炼程序；

（四种类型的代码块：1、普通代码块：在方法后面使用“{}”括起来的代码片段；2、静态代码块：在类中使用static修饰，并使用“{}”括起来的代码片段；3、同步代码块：使用synchronized关键字修饰，并使用“{}”括起来的代码片段，表示同一时间只能有一个县城进入到该方法；4、构造代码块：在类中没有任何的前缀或后缀，并使用“{}”括起来的代码片段。编译器会把构造代码块插入到每个构造函数的最前端。构造代码块的两个特性：1、在每个构造函数中都运行；2、在构造函数中它会首先运行）。

建议37：构造代码块会想你所想；

（编译器会把构造代码块插入到每一个构造函数中，有一个特殊情况：如果遇到this关键字（也就是构造函数调用自身其他的构造函数时）则不插入构造代码块。如果遇到super关键字，编译器会把构造代码块插入到super方法之后执行）。

建议38：使用静态内部类提高封装性；

（Java嵌套内分为两种：1、静态内部类；2、内部类；静态内部类两个优点：加强了类的封装性和提高了代码的可读性。静态内部类与普通内部类的区别：1、静态内部类不持有外部类的引用，在普通内部类中，我们可以直接访问外部类的属性、方法，即使是private类型也可以访问，这是因为内部类持有一个外部类的引用，可以自由访问。而静态内部类，则只可以访问外部类的静态方法和静态属性，其他则不能访问。2、静态内部类不依赖外部类，普通内部类与外部类之间是相互依赖的关系，内部类不能脱离外部类实例，同声同死，一起声明，一起被垃圾回收器回收。而静态内部类可以独立存在，即使外部类消亡了；3、普通内部类不能声明static的方法和变量，注意这里说的是变量，常量（也就是final static修饰的属性）还是可以的，而静态内部类形似外部类，没有任何限制）。

建议39：使用匿名类的构造函数；

（List l2 = new ArrayList(){}; //定义了一个继承于ArrayList的匿名类，只是没有任何的覆写方法而已

在CODE上查看代码片派生到我的代码片

List l3 = new ArrayList(){{}}; //定义了一个继承于ArrayList的匿名类，并且包含一个初始化块，类似于构造代码块）<span style="font-family: Arial, Helvetica, sans-serif; rgb(255, 255, 255);">）</span>

建议40：匿名类的构造函数很特殊；

（匿名类初始化时直接调用了父类的同参数构造器，然后再调用自己的构造代码块）

建议41：让多重继承成为现实；

（Java中一个类可以多种实现，但不能多重继承。使用成员内部类实现多重继承。内部类一个重要特性：内部类可以继承一个与外部类无关的类，保证了内部类的独立性，正是基于这一点，多重继承才会成为可能）。

建议42：让工具类不可实例化；

（工具类的方法和属性都是静态的，不需要实例即可访问。实现方式：将构造函数设置为private，并且在构造函数中抛出Error错误异常）。

建议43：避免对象的浅拷贝；

（浅拷贝存在对象属性拷贝不彻底的问题。对于只包含基本数据类型的类可以使用浅拷贝；而包含有对象变量的类需要使用序列化与反序列化机制实现深拷贝）。

建议44：推荐使用序列化实现对象的拷贝；

（通过序列化方式来处理，在内存中通过字节流的拷贝来实现深拷贝。使用此方法进行对象拷贝时需注意两点：1、对象的内部属性都是可序列化的；2、注意方法和属性的特殊修饰符，比如final、static、transient变量的序列化问题都会影响拷贝效果。一个简单办法，使用Apache下的commons工具包中的SerializationUtils类，直接使用更加简洁方便）。

建议45：覆写equals方法时不要识别不出自己；

（需要满足p.equals(p)放回为真，自反性）。

建议46：equals应该考虑null值情景；

（覆写equals方法时需要判一下null，否则可能产生NullPointerException异常）。

建议47：在equals中使用getClass进行类型判断；

（使用getClass方法来代替instanceof进行类型判断）。

建议48：覆写equals方法必须覆写hashCode方法；

（需要两个相同对象的hashCode方法返回值相同，所以需要覆写hashCode方法，如果不覆写，两个不同对象的hashCode肯定不一样，简单实现hashCode方法，调用org.apache.commons.lang.builder包下的Hash码生成工具HashCodeBuilder）。

建议49：推荐覆写toString方法；

（原始toString方法显示不人性化）。

建议50：使用package-info类为包服务；

（package-info类是专门为本包服务的，是一个特殊性主要体现在3个方面：1、它不能随便被创建；2、它服务的对象很特殊；3、package-info类不能有实现代码；package-info类的作用：1、声明友好类和包内访问常量；2、为在包上标注注解提供便利；3、提供包的整体注释说明）。

建议51：不要主动进行垃圾回收；

（主动进行垃圾回收是一个非常危险的动作，因为System.gc要停止所有的响应（Stop 天河world），才能检查内存中是否有可回收的对象，所有的请求都会暂停）。

## 第四章 字符串

建议52：推荐使用String直接量赋值；

（一般对象都是通过new关键字生成，String还有第二种生成方式，即直接声明方式，如String str = "a";String中极力推荐使用直接声明的方式，不建议使用new String("a")的方式赋值。原因：直接声明方式：创建一个字符串对象时，首先检查字符串常量池中是否有字面值相等的字符串，如果有，则不再创建，直接返回池中引用，若没有则创建，然后放到池中，并返回新建对象的引用。使用new String()方式：直接声明一个String对象是不检查字符串常量池的，也不会吧对象放到池中。String的intern方法会检查当前的对象在对象池中是否有字面值相同的引用对象，有则返回池中对象，没有则放置到对象池中，并返回当前对象）。

建议53：注意方法中传递的参数要求；

（replaceAll方法传递的第一个参数是正则表达式）。

建议54：正确使用String、StringBuffer、StringBuilder；

（String使用“+”进行字符串连接，之前连接之后会产生一个新的对象，所以会不断的创建新对象，优化之后与StringBuilder和StringBuffer采用同样的append方法进行连接，但是每一次字符串拼接都会调用一次toString方法，所以会很耗时。StringBuffer与StringBuilder基本相同，只是一个字符数组的在扩容而已，都是可变字符序列，不同点是：StringBuffer是线程安全的，StringBuilder是线程不安全的）。

建议55：注意字符串的位置；

（在“+”表达式中，String字符串具有最高优先级）（Java对加号“+”的处理机制：在使用加号进行计算的表达式中，只要遇到String字符串，则所有的数据都会转换为String类型进行拼接，如果是原始数据，则直接拼接，如果是对象，则调用toString方法的返回值然后拼接）。

建议56：自由选择字符串拼接方式；

（字符串拼接有三种方法：加号、concat方法及StringBuilder（或StringBuffer）的append方法。字符串拼接性能中，StringBuilder的append方法最快，concat方法次之，加号最慢。原因：1、“+”方法拼接字符串：虽然编译器对字符串的加号做了优化，使用StringBuidler的append方法进行追加，但是与纯粹使用StringBuilder的append方法不同：一是每次循环都会创建一个StringBuilder对象，二是每次执行完都要调用toString方法将其准换为字符串--toString方法最耗时；2、concat方法拼接字符串：就是一个数组拷贝，但是每次的concat操作都会新创建一个String对象，这就是concat速度慢下来的真正原因；3、append方法拼接字符串：StringBuidler的append方法直接由父类AbstractStringBuilder实现，整个append方法都在做字符数组处理，没有新建任何对象，所以速度快）。

建议57：推荐在复杂字符串操作中使用正则表达式；

（正则表达式是恶魔，威力强大，但难以控制）。

建议58：强烈建议使用UTF编码；

（一个系统使用统一的编码）。

建议59：对字符串排序持一种宽容的心态；

（如果排序不是一个关键算法，使用Collator类即可。主要针对于中文）。

## 第五章 数组和集合

建议60：性能考虑，数组是首选；

（性能要求较高的场景中使用数组替代集合）（基本类型在栈内存中操作，对象在堆内存中操作。数组中使用基本类型是效率最高的，使用集合类会伴随着自动装箱与自动拆箱动作，所以性能相对差一些）。

建议61：若有必要，使用变长数组；

（使用Arrays.copyOf(datas,newLen)对原数组datas进行扩容处理）。

建议62：警惕数组的浅拷贝；

（通过ArrayscopyOf(box1,box1.length)方法产生的数组是一个浅拷贝，这与序列化的浅拷贝完全相同：基本类型是直接拷贝值，其他都是拷贝引用地址。数组中的元素没有实现Serializable接口）。

建议63：在明确的场景下，为集合指定初始容量；

（ArrayList集合底层使用数组存储，如果没有初始为ArrayList指定数组大小，默认存储数组大小长度为10，添加的元素达到数组临界值后，使用Arrays.copyOf方法进行1.5倍扩容处理。HashMap是按照倍数扩容的，Stack继承自Vector，所采用扩容规则的也是翻倍）。

建议64：多种最值算法，适时选择；

（最值计算时使用集合最简单，使用数组性能最优，利用Set集合去重，使用TreeSet集合自动排序）。

建议65：避开基本类型数组转换列表陷阱；

（原始类型数组不能作为asList的输入参数，否则会引起程序逻辑混乱）（基本类型是不能泛化的，在java中数组是一个对象，它是可以泛化的。使用Arrays.asList(data)方法传入一个基本类型数组时，会将整个基本类型数组作为一个数组对象存入，所以存入的只会是一个对象。JVM不可能输出Array类型，因为Array是属于java.lang.reflect包的，它是通过反射访问数组元素的工具类。在Java中任何一个数组的类都是“[I”，因为Java并没有定义数组这个类，它是编译器编译的时候生成的，是一个特殊的类）。

建议66：asList方法产生的List对象不可更改；

（使用add方法向asList方法生成的集合中添加元素时，会抛UnsupportedOperationException异常。原因：asList生成的ArrayList集合并不是java.util.ArrayList集合，而是Arrays工具类的一个内置类，我们经常使用的List.add和List.remove方法它都没有实现，也就是说asList返回的是一个长度不可变的列表。此处的列表只是数组的一个外壳，不再保持列表动态变长的特性）。

建议67：不同的列表选择不同的遍历方法；

（ArrayList数组实现了RandomAccess接口（随机存取接口），ArrayList是一个可以随机存取的列表。集合底层如果是基于数组实现的，实现了RandomAccess接口的集合，使用下标进行遍历访问性能会更高；底层使用双向链表实现的集合，使用foreach的迭代器遍历性能会更高）。

建议68：频繁插入和删除时使用LinkedList；

（ArrayList集合，每次插入或者删除一个元素，其后的所有元素就会向后或者向前移动一位，性能很低。LinkedList集合插入时不需要移动其他元素，性能高；修改元素，LinkedList集合比ArrayList集合要慢很多；添加元素，LinkedList与ArrayList集合性能差不多，LinkedList添加一个ListNode，而ArrayList则在数组后面添加一个Entry）。

建议69：列表相等只需关心元素数据；

（判断集合是否相等时只须关注元素是否相等即可）（ArrayList与Vector都是List，都实现了List接口，也都继承了AbstractList抽象类，其equals方法是在AbstractList中定义的。所以只要求两个集合类实现了List接口就成，不关心List的具体实现类，只要所有的元素相等，并且长度也相等就表明两个List是相等的，与具体的容量类型无关）。

建议70：子列表只是原列表的一个视图；

（使用==判断相等时，需要满足两个对象地址相等，而使用equals判断两个对象是否相等时，只需要关注表面值是否相等。subList方法是由AbstractList实现的，它会根据是不是可以随机存取来提供不同的SubList实现方式，RandomAccessSubList是SubList子类，SubList类中subList方法的实现原理：它返回的SubList类是AbstractList的子类，其所有的方法如get、set、add、remove等都是在原始列表上的操作，它自身并没有生成一个数组或是链表，也就是子列表只是原列表的一个视图，所有的修改动作都反映在了原列表上）。

建议71：推荐使用subList处理局部列表；

（需求：要删除一个ArrayList中的20-30范围内的元素；将原列表转换为一个可变列表，然后使用subList获取到原列表20到30范围内的一个视图（View），然后清空该视图内的元素，即可在原列表中删除20到30范围内的元素）。

建议72：生成子列表后不要再操作原列表；

（subList生成子列表后，使用Collections.unmodifiableList(list)；保持原列表的只读状态）（利用subList生成子列表后，更改原列表，会造成子列表抛出java.util.ConcurrentModificationException异常。原因：subList取出的列表是原列表的一个视图，原数据集（代码中的list变量）修改了，但是subList取出的子列表不会重新生成一个新列表（这点与数据库视图是不相同的），后面再对子列表操作时，就会检测到修改计数器与预期的不相同，于是就抛出了并发修改异常）。

建议73：使用Comparator进行排序；

（Comparable接口可以作为实现类的默认排序法，Comparator接口则是一个类的扩展排序工具）（两种数据排序实现方式：1、实现Comparable接口，必须要实现compareTo方法，一般由类直接实现，表明自身是可比较的，有了比较才能进行排序；2、实现Comparator接口，必须实现compare方法，Comparator接口是一个工具类接口：用作比较，它与原有类的逻辑没有关系，只是实现两个类的比较逻辑）。

建议74：不推荐使用binarySearch对列表进行检索；

（indexOf与binarySearch方法功能类似，只是使用了二分法搜索。使用二分查找的首要条件是必须要先排序，不然二分查找的值是不准确的。indexOf方法直接就是遍历搜寻。从性能方面考虑，binarySearch是最好的选择）。

建议75：集合中的元素必须做到compareTo和equals同步；

（实现了compareTo方法，就应该覆写equals方法，确保两者同步）（在集合中indexOf方法是通过equals方法的返回值判断的，而binarySearch查找的依据是compareTo方法的返回值；equals是判断元素是否相等，compareTo是判断元素在排序中的位置是否相同）。

建议76：集合运算时使用更优雅的方式；

（1、并集：list1.addAll(list2); 2、交集：list1.retainAll(list2); 3、差集：list1.removeAll(list2); 4、无重复的并集：list2.removeAll(list1);list1.addAll(list2);）。

建议77：使用shuffle打乱列表；

（使用Collections.shuffle(tagClouds)打乱列表）。

建议78：减少HashMap中元素的数量；

（尽量让HashMap中的元素少量并简单）（现象：使用HashMap存储数据时，还有空闲内存，却抛出了内存溢出异常；原因：HashMap底层的数组变量名叫table，它是Entry类型的数组，保存的是一个一个的键值对。与ArrayList集合相比，HashMap比ArrayList多了一次封装，把String类型的键值对转换成Entry对象后再放入数组，这就多了40万个对象，这是问题产生的第一个原因；HashMap在插入键值对时，会做长度校验，如果大于或等于阈值（threshold变量），则数组长度增大一倍。默认阈值是当前长度与加载因子的乘积，默认的加载因子（loadFactor变量）是0.75，也就是说只要HashMap的size大于数组长度的0.75倍时，就开始扩容。导致到最后，空闲的内存空间不足以增加一次扩容时就会抛出OutOfMemoryError异常）。

建议79：集合中的哈希码不要重复；

（列表查找不管是遍历查找、链表查找或者是二分查找都不够快。最快的是Hash开头的集合（如HashMap、HashSet等类）查找，原理：根据hashCode定位元素在数组中的位置。HashMap的table数组存储元素特点：1、table数组的长度永远是2的N次幂；2、table数组中的元素是Entry类型；3、table数组中的元素位置是不连续的；每个Entry都有一个next变量，它会指向下一个键值对，用来链表的方式来处理Hash冲突的问题。如果Hash码相同，则添加的元素都使用链表处理，在查找的时候这部分的性能与ArrayList性能差不多）。

建议80：多线程使用Vector或HashTable；

（Vector与ArrayList原理类似，只是是线程安全的，HashTable是HashMap的多线程版本。线程安全：基本所有的集合类都有一个叫快速失败（Fail-Fast）的校验机制，当一个集合在被多个线程修改并访问时，就可能出现ConcurrentModificationException异常，这是为了确保集合方法一致而设置的保护措施；实现原理是modCount修改计数器：如果在读列表时，modCount发生变化（也就是有其他线程修改）则会抛出ConcurrentModificationException异常。线程同步：是为了保护集合中的数据不被脏读、脏写而设置的）。

建议81：非稳定排序推荐使用List；

（非稳定的意思是：经常需要改动；TreeSet集合中元素不可重复，且默认按照升序排序，是根据Comparable接口的compareTo方法的返回值确定排序位置的。SortedSet接口（TreeSet实现了该接口）只是定义了在该集合加入元素时将其进行排序，并不能保证元素修改后的排序结果。因此TreeSet适用于不变量的集合数据排序，但不适合可变量的排序。对于可变量的集合，需要自己手动进行再排序）（SortedSet中的元素被修改后可能会影响其排序位置）。

建议82：由点及面，一页知秋--集合大家族；

1、List：实现List接口的集合主要有：ArrayList、LinkedList、Vector、Stack，其中ArrayList是一个动态数组，LinkedList是一个双向链表，Vector是一个线程安全的动态数组，Stack是一个对象栈，遵循先进后出的原则；

2、Set：Set是不包含重复元素的集合，其主要的实现类有：EnumSet、HashSet、TreeSet，其中EnumSet是枚举类型的专用Set，HashSet是以哈希码决定其元素位置的Set，原理与HashMap相似，提供快速插入与查找方法，TreeSet是一个自动排序的Set，它实现了SortedSet接口；

3、Map：可以分为排序Map和非排序Map；排序Map为TreeMap，根据Key值进行自动排序；非排序Map主要包括：HashMap、HashTable、Properties、EnumMap等，其中Properties是HashTable的子类，EnumMap则要求其Key必须是某一个枚举类型；

4：Queue：分为两类，一类是阻塞式队列，队列满了以后再插入元素会抛异常，主要包括：ArrayBlockingQueue、PriorityBlockingQueue、LinkedBlockingQueue，其中ArrayBlockingQueue是以数组方式实现的有界阻塞队列；PriorityBlockingQueue是依照优先级组件的队列；LinkedBlockingQueue是通过链表实现的阻塞队列；另一类是非阻塞队列，无边界的，只要内存允许，都可以追加元素，经常使用的是PriorityQueue类。还有一种是双端队列，支持在头、尾两端插入和移除元素，主要实现类是：ArrayDeque、LinkedBlockingDeque、LinkedList；

5、数组：数组能存储基本类型，而集合不行；所有的集合底层存储的都是数组；

6、工具类：数组的工具类是：java.util.Arrays和java.lang.reflect.array；集合的工具类是java.util.Collections;

7、扩展类：可以使用Apache的commons-collections扩展包，也可以使用Google的google-collections扩展包）。

## 第六章 枚举和注解

建议83：推荐使用枚举定义常量；

（在项目开发中，推荐使用枚举常量替代接口常量和类常量）（常量分为：类常量、接口常量、枚举常量；枚举常量优点：1、枚举常量更简单；2、枚举常量属于稳态性（不允许发生越界）；3、枚举具有内置方法，values方法可以获取到所有枚举值；4、枚举可以自定义方法）。

建议84：使用构造函数协助描述枚举项；

（每个举项都是该枚举的一个实例。可以通过添加属性，然后通过构造函数给枚举项添加更多描述信息）。

建议85：小心switch带来的空值异常；

（使用枚举值作为switch(枚举类)；语句的条件值时，需要对枚举类进行判断是否为null值。因为Java中的switch语句只能判断byte、short、char、int类型，JDK7可以判断String类型，使用switch语句判断枚举类型时，会根据枚举的排序值匹配。如果传入的只是null的话，获取排序值需要调用如season.ordinal()方法时会抛出NullPointerException异常）。

建议86：在switch的default代码块中增加AssertionError错误；

（switch语句在使用枚举类作为判断条件时，避免出现增加了一个枚举项，而switch语句没做任何修改，编译不会出现问题，但是在运行期会发生非预期的错误。为避免这种情况出现，建议在default后直接抛出一个AssertionError错误。含义是：不要跑到这里来，一跑到这里来马上就会报错）。

建议87：使用valueOf前必须进行校验；

（Enum.valueOf()方法会把一个String类型的名称转变为枚举项，也就是在枚举项中查找出字面值与该参数相等的枚举项。valueOf方法先通过反射从枚举类的常量声明中查找，若找到就直接返回，若找不到就抛出IllegalArgumentException异常）。

建议88：用枚举实现工厂方法模式更简洁；

（工厂方法模式是“创建对象的接口，让子类决定实例化哪一个类，并使一个类的实例化延迟到其子类”。枚举实现工厂方法模式有两种方法：1、枚举非静态方法实现工厂方法模式；2、通过抽象方法生成产品；优点：1、避免错误调用的发生；2、性能好，使用便捷；3、减低类间耦合性）。

建议89：枚举项的数量控制在64个以内；

（Java提供了两个枚举集合：EnumSet、EnumMap；EnumSet要求其元素必须是某一枚举的枚举项，EnumMap表示Key值必须是某一枚举的枚举项。由于枚举类型的实例数量固定并且有限，相对来说EnumSet和EnumMap的效率会比其他Set和Map要高。Java处理EnumSet过程：当枚举项小于等于64时，创建一个RegularEnumSet实例对象，大于64时创一个JumboEnumSet实例对象。RegularEnumSet是把每个枚举项编码映射到一个long类型数字得每一位上，而JumboEnumSet则会先按照64个一组进行拆分，然后每个组再映射到一个long类型的数字得每一位上）。

建议90：小心注解继承；

（不常用的元注解（Meta-Annotation）：@Inherited，它表示一个注解是否可以自动被继承）。

建议91：枚举和注解结合使用威力更大；

（注解和接口写法类似，都采用了关键字interface，而且都不能有实现代码，常量定义默认都是public static final类型的等，他们的主要不同点：注解要在interface前加上@字符，而且不能继承，不能实现）。

建议92：注意@Override不同版本的区别；

（@Override注解用于方法的覆写上，它在编译期有效，也就是Java编译器在编译时会根据该注解检查方法是否真的是覆写，如果不是就报错，拒绝编译。Java1.5版本中@Override是严格遵守覆写的定义：子类方法与父类方法必须具有相同的方法名、输入参数、输出参数（允许子类缩小）、访问权限（允许子类扩大），父类必须是一个类，不是是接口，否则不能算是覆写。而在Java1.6就开放了很多，实现接口的方法也可以加上@Override注解了。如果是Java1.6版本移植到Java1.5版本中时，需要删除接口实现方法上的@Override注解）。

## 第七章 泛型和反射

建议93：Java的泛型是类型擦除的；

（加入泛型优点：加强了参数类型的安全性，减少了类型的转换。Java的泛型在编译期有效，在运行期被删除，也就是说所有的泛型参数类型在编译后都会被清除掉。所以：1、泛型的class对象时是相同的；2、泛型数组初始化时不能声明泛型类型；3、instanceof不允许存在泛型参数）。

建议94：不能初始化泛型参数和数组；

（泛型类型在编译期被擦除，在类初始化时将无法获得泛型的具体参数，所以泛型参数和数组无法初始化，但是ArrayList却可以，因为ArrayList初始化是向上转型变成了Object类型；需要泛型数组解决办法：只声明，不再初始化，由构造函数完成初始化操作）。

建议95：强制声明泛型的实际类型；

（无法从代码中推断出泛型类型的情况下，即可强制声明泛型类型；方法：List<Integer> list2 = ArrayUtils.<Integer>asList();在输入前定义这是一个Integer类型的参数）。

建议96：不同的场景使用不同的泛型通配符；

（Java泛型支持通配符（Wildcard），可以单独使用一个“?”表示任意类，也可以使用extends关键字表示某一个类（接口）的子类型，还可以使用super关键字表示某一个类（接口）的父类型。1、泛型结构只参与“读”操作则限定上界（extends关键字）；2、泛型结构只参与“写”操作则限定下界（使用super关键字）；3、如果一个泛型结构既用作“读”操作也用作“写”操作则使用确定的泛型类型即可，如List<E>）。

建议97：警惕泛型是不能协变和逆变的；

（Java的泛型是不支持协变和逆变的，只是能够实现协变和逆变）（协变和逆变是指宽类型和窄类型在某种情况下（如参数、泛型、返回值）替换或交换的特性。简单地说，协变是用一个窄类型替换宽类型，而逆变则是用宽类型覆盖窄类型。子类覆写父类返回值类型比父类型变窄，则是协变；子类覆写父类型的参数类型变宽，则是逆变。数组支持协变，泛型不支持协变）。

建议98：建议采用的顺序是List<T>,List<?>,List<Object>；

（1、List<T>是确定的某一个类型，编码者知道它是一个类型，只是在运行期才确定而已；2、List<T>可以进行读写操作，List<?>是只读类型，因为编译器不知道List中容纳的是什么类型的元素，无法增加、修改，但是能删除，List<Object>也可以读写操作，只是此时已经失去了泛型存在的意义了）。

建议99：严格限定泛型类型采用多重界限；

（使用“&”符号连接多个泛型界限，如：<T extends Staff & Passenger>）。

建议100：数组的真实类型必须是泛型类型的子类型；

（有可能会抛出ClassCastException异常，toArray方法返回后会进行一次类型转换，Object数组转换成了String数组。由于我们无法在运行期获得泛型类型的参数，因此就需要调用者主动传入T参数类型）。

建议101：注意Class类的特殊性；

（Java处理的基本机制：先把Java源文件编译成后缀为class的字节码文件，然后再通过ClassLoader机制把这些类文件加载到内存中，最后生成实例执行。Java使用一个元类（MetaClass）来描述加载到内存中的类数据，这就是Class类，它是一个描述类的类对象。Class类是“类中类”，具有特殊性：1、无构造函数，不能实例化，Class对象是在加载类时由Java虚拟机通过调用类加载器中的defineClass方法自动构建的；2、可以描述基本类型，8个基本类型在JVM中并不是一个对象，一般存在于栈内存中，但是Class类仍然可以描述它们，例如可以使用int.class表示int类型的类对象；3、其对象都是单例模式，一个Class的实例对象描述一个类，并且只描述一个类，反过来也成立，一个类只有一个Class实例对象。Class类是Java的反射入口，只有在获得了一个类的描述对象后才能动态地加载、调用，一般获得一个Class对象有三种途径：1、类属性方式，如String.class；2、对象的getClass方法，如new String().getClass()；3、forName方法重载，如Class.forName("java.lang.String")。获得了Class对象后，就可以通过getAnnotation()获得注解，通过个体Methods()获得方法，通过getConstructors()获得构造函数等）。

建议102：适时选择getDeclaredXXX和getXXX；

（getMethod方法获得的是所有public访问级别的方法，包括从父类继承的方法，而getDeclaredMethod获得的是自身类的所有方法，包括公用方法、私有方法等，而且不受限于访问权限。Java之所以这样处理，是因为反射本意只是正常代码逻辑的一种补充，而不是让正常代码逻辑产生翻天覆地的改动，所以public的属性和方法最容易获取，私有属性和方法也可以获取，但要限定本类。如果需要列出所有继承自父类的方法，需要先获得父类，然后调用getDeclaredMethods方法，之后持续递归）。

建议103：反射访问属性或方法是将Accessible设置为true；

（通过反射方式执行方法时，必须在invoke之前检查Accessible属性。而Accessible属性并不是我们语法层级理解的访问权限，而是指是否更容易获得，是否进行安全检查。Accessible属性只是用来判断是否需要进行安全检查的，如果不需要则直接执行，这就可以大幅度地提升系统性能。经过测试，在大量的反射情况下，设置Accessible为true可以提升性能20倍以上）。

建议104：使用forName动态加载类文件；

（forName只是加载类，并不执行任何代码）（动态加载（Dynamic Loading）是指在程序运行时加载需要的类库文件，一般情况下，一个类文件在启动时或首次初始化时会被加载到内存中，而反射则可以在运行时再决定是否要加载一个类，然后在JVM中加载并初始化。动态加载通常是通过Class.forName(String)实现。一个对象的生成必然会经过一下两个步骤：1、加载到内存中生成Class的实例对象；2、通过new关键字生成实例对象；动态加载的意义：加载一个类即表示要初始化该类的static变量，特别是static代码块，在这里我们可以做大量的工作，比如注册自己，初始化环境等，这才是我们重点关注的逻辑）。

建议105：动态加载不适合数组；

（通过反射操作数组使用Array类，不要采用通用的反射处理API）（如果forName要加载一个类，那它首先必须是一个类--8个基本类型排除在外，不是具体的类；其次，它必须具有可追索的类路径，否则会报ClassNotFoundException异常。在Java中，数组是一个非常特殊的类，虽然是一个类，但没有定义类路径。作为forName参数时会抛出ClassNotFoundException异常，原因是：数组虽然是一个类，在声明时可以定义为String[]，但编译器编译后会为不同的数组类型生成不同的类，所以要想动态创建和访问数组，基本的反射是无法实现的）。

建议106：动态代理可以使代理模式更加灵活；

（Java的反射框架提供了动态代理（Dynamic Proxy）机制，允许在运行期对目标类生成代理，避免重复开发。静态代理是通过代理主题角色（Proxy）和具体主题角色（Real Subject）共同实现抽象主题角色（Subject）的逻辑的，只是代理主题角色把相关的执行逻辑委托给了具体主题角色而已。动态代理需要实现InvocationHandler接口，必须要实现invoke方法，该方法完成了对真实方法的调用）。

建议107：使用反射增加装饰模式的普适性；

（装饰模式（Decorator Pattern）的定义是“动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比于生成子类更为灵活”。比较通用的装饰模式，只需要定义被装饰的类及装饰类即可，装饰行为由动态代理实现，实现了对装饰类和被装饰类的完全解耦，提供了系统的扩展性）。

建议108：反射让模板方法模式更强大；

（决定使用模板方法模式时，请尝试使用反射方式实现，它会让你的程序更灵活、更强大）（模板方法模式（Template Method Pattern）的定义是：定义一个操作中的算法骨架，将一些步骤延迟到子类中，使子类不改变一个算法的结构即可重定义该算法的某些特定步骤。简单说，就是父类定义抽象模板作为骨架，其中包括基本方法（是由子类实现的方法，并且在模板方法被调用）和模板方法（实现对基本方法的调度，完成固定的逻辑），它使用了简单的继承和覆写机制。使用反射后，不需要定义任何抽象方法，只需定义一个基本方法鉴别器即可加载复合规则的基本方法）。

建议109：不需要太多关注反射效率；

（反射效率低是个真命题，但因为这一点而不使用它就是个假命题）（反射效率相对于正常的代码执行确实低很多（经测试，相差15倍左右），但是它是一个非常有效的运行期工具类）。

## 第8章 异常

建议110：提倡异常封装；（异常封装有三方面的优点：1、提高系统的友好性；2、提高系统的可维护性；3、解决Java异常机制本身的缺陷）；

建议111：采用异常链传递异常；

（责任链模式（Chain of Responsibility），目的是将多个对象连城一条链，并沿着这条链传递该请求，直到有对象处理它为止，异常的传递处理也应该采用责任链模式）。

建议112：受检异常尽可能转化为非受检异常；

（受检异常威胁到系统的安全性、稳定性、可靠性、正确性时、不能转为非受检异常）（受检异常（Checked Exception），非受检异常（Unchecked Exception），受检异常时正常逻辑的一种补偿处理手段，特别是对可靠性要求比较高的系统来说，在某些条件下必须抛出受检异常以便由程序进行补偿处理，也就是说受检异常有合理的存在理由。但是受检异常有不足的地方：1、受检异常使接口声明脆弱；2、受检异常是代码的可读性降低，一个方法增加了受检异常，则必须有一个调用者对异常进行处理。受检异常需要try..catch处理；3、受检异常增加了开发工作量。避免以上受检异常缺点办法：将受检异常转化为非受检异常）。

建议113：不要在finally块中处理返回值；

（在finally块中加入了return语句会导致以下两个问题：1、覆盖了try代码块中的return返回值；2、屏蔽异常，即使throw出去了异常，异常线程会登记异常，但是当执行器执行finally代码块时，则会重新为方法赋值，也就是告诉调用者“该方法执行正确”，没有发生异常，于是乎，异常神奇的消失了）。

建议114：不要在构造函数中抛异常；

（Java异常机制有三种：1、Error类及其子类表示的是错误，它是不需要程序员处理的也不能处理的异常，比如VirtualMachineError虚拟机错误，ThreadDeath线程僵死等；2、RuntimeException类及其子类表示的是非受检异常，是系统可能抛出的异常，程序员可以去处理，也可以不处理，最经典的是NullPointerException空指针异常和IndexOutOfBoundsException越界异常；3、Exception类及其子类（不包含非受检异常）表示的是受检异常，这是程序员必须要处理的异常，不处理则程序不能通过编译，比如IOException表示I/O异常，SQLException数据库访问异常。一个对象的创建过程要经过内存分配、静态代码初始化、构造函数执行等过程，构造函数中是否允许抛出异常呢？从Java语法上来说，完全可以，三类异常都可以，但是从系统设计和开发的角度分析，则尽量不要在构造函数中抛出异常）。

建议115：使用Throwable获得栈信息；

（在出现异常时（或主动声明一个Throwable对象时），JVM会通过fillInStackTrace方法记录下栈信息，然后生成一个Throwable对象，这样就能知道类间的调用顺序、方法名称以及当前行号等）。

建议116：异常只为异常服务；

（异常原本是正常逻辑的一个补充，但有时候会被当前主逻辑使用。异常作为主逻辑有问题：1、异常判断降低了系统性能；2、降低了代码的可读性，只有详细了解valueOf方法的人才能读懂这样的代码，因为valueOf抛出的是一个非受检异常；3、隐藏了运行期可能产生的错误，catch到异常，但没有做任何处理）。

建议117：多使用异常，把性能问题放一边；

（new一个IOException会被String慢5倍：因为它要执行fillInStackTrace方法，要记录当前栈的快照，而String类则是直接申请一个内存创建对象。而且，异常类是不能缓存的。但是异常是主逻辑的例外逻辑，会让方法更符合实际的处理逻辑，同时使主逻辑更加清晰，可让正常代码和异常代码分离、能快速查找问题（栈信息快照）等）。

## 第9章 多线程和并发

建议118：不推荐覆写start方法；

（继承自Thread类的多线程类不必覆写start方法。原本的start方法中，调用了本地方法start0，它实现了启动线程、申请栈内存、运行run方法、修改线程状态等职责，线程管理和栈内存管理都是由JVM实现的，如果覆盖了start方法，也就是撤销了线程管理和栈内存管理的能力。所以除非必要，不然不要覆写start方法，即使需要覆写start方法，也需要在方法体内加上super.start调用父类中的start方法来启动默认的线程操作）。

建议119：启动线程前stop方法是不可靠的；

（现象：使用stop方法停止一个线程，而stop方法在此处的目的不是停止一个线程，而是设置线程为不可启用状态。但是运行结果出现奇怪现象：部分线程还是启动了，也就是在某些线程（没有规律）中的start方法正常执行了。在不符合判断规则的情况下，不可启用状态的线程还是启用了，这是线程启动（start方法）一个缺陷。Thread类的stop方法会根据线程状态来判断是终结线程还是设置线程为不可运行状态，对于未启动的线程（线程状态为NEW）来说，会设置其标志位为不可启动，而其他的状态则是直接停止。start方法源码中，start0方法在stop0方法之前，也就是说即使stopBeforeStart为true（不可启动），也会先启动一个线程，然后再stop0结束这个线程，而罪魁祸首就在这里！所以不要使用stop方法进行状态的设置）。

建议120：不适用stop方法停止线程；

（线程启动完毕后，需要停止，Java只提供了一个stop方法，但是不建议使用，有以下三个问题：1、stop方法是过时的；2、stop方法会导致代码逻辑不完整，stop方法是一种“恶意”的中断，一旦执行stop方法，即终止当前正在运行的线程，不管线程逻辑是否完整，这是非常危险的，以为stop方法会清除栈内信息，结束该线程，但是可能该线程的一段逻辑非常重，比如子线程的主逻辑、资源回收、情景初始化等，因为stop线程了，这些都不会再执行。子线程执行到何处会被关闭很难定位，这为以后的维护带来了很多麻烦；3、stop方法会破坏原子逻辑，多线程为了解决共享资源抢占的问题，使用了锁概念，避免资源不同步，但是stop方法会丢弃所有的锁，导致原子逻辑受损。Thread提供的interrupt中断线程方法，它不能终止一个正在执行着的线程，它只是修改中断标志唯一。总之，期望终止一个正在运行的线程，不能使用stop方法，需要自行编码实现。如果使用线程池（比如ThreadPoolExecutor类），那么可以通过shutdown方法逐步关闭池中的线程）。

建议121：线程优先级只使用三个等级；

（线程优先级推荐使用MIN\_PRIORITY、NORM\_PRIORITY、MAX\_PRIORITY三个级别，不建议使用其他7个数字）（线程的优先级（Priority）决定了线程获得CPU运行的机会，优先级越高，运行机会越大。事实：1、并不是严格尊重线程优先级别来执行的，分为10个级别；2、优先级差别越大，运行机会差别越大；对于Java来说，JVM调用操作系统的接口设置优先级，比如Windows是通过调用SetThreadPriority函数来设置的。不同操作系统线程优先级设置是不相同的，Windows有7个优先级，Linux有140个优先级，Freebsd有255个优先级。Java缔造者也发现了该问题，于是在Thread类中设置了三个优先级，建议使用优先级常量，而不是1到10随机的数字）。

建议122：使用线程异常处理器提升系统可靠性；

（可以使用线程异常处理器来处理相关异常情况的发生，比如当机自动重启，大大提高系统的可靠性。在实际环境中应用注意以下三点：1、共享资源锁定；2、脏数据引起系统逻辑混乱；3、内存溢出，线程异常了，但由该线程创建的对象并不会马上回收，如果再重新启动新线程，再创建一批新对象，特别是加入了场景接管，就危险了，有可能发生OutOfMemory内存泄露问题）。

建议123：volatile不能保证数据同步；

（volatile不能保证数据是同步的，只能保证线程能够获得最新值）（volatile关键字比较少用的原因：1、Java1.5之前该关键字在不同的操作系统上有不同的表现，移植性差；2、比较难设计，而且误用较多。在变量钱加上一个volatile关键字，可以确保每个线程对本地变量的访问和修改都是直接与主内存交互的，而不是与本地线程的工作内存交互的，保证每个线程都能获得最“新鲜”的变量值。但是volatile关键字并不能保证线程安全，它只能保证当前线程需要该变量的值时能够获得最新的值，而不能保证多个线程修改的安全性）。

建议124：异步运算考虑使用Callable接口；

（多线程应用的两种实现方式：一种是实现Runnable接口，另一种是继承Thread类，这两个方式都有缺点：run方法没有返回值，不能抛出异常（归根到底是Runnable接口的缺陷，Thread也是实现了Runnable接口），如果需要知道一个线程的运行结果就需要用户自行设计，线程类本身也不能提供返回值和异常。Java1.5开始引入了新的接口Callable，类似于Runnable接口，实现它就可以实现多线程任务，实现Callable接口的类，只是表明它是一个可调用的任务，并不表示它具有多线程运算能力，还是需要执行器来执行的）。

建议125：优先选择线程池；

（Java1.5以前，实现多线程比较麻烦，需要自己启动线程，并关注同步资源，防止出现线程死锁等问题，Java1.5以后引入了并行计算框架，大大简化了多线程开发。线程有五个状态：新建状态（New）、可运行状态（Runnable，也叫作运行状态）、阻塞状态（Blocked）、等待状态（Waiting）、结束状态（Terminated），线程的状态只能由新建转变为运行态后才可能被阻塞或等待，最后终结，不可能产生本末倒置的情况，比如想把结束状态变为新建状态，则会出现异常。线程运行时间分为三个部分：T1为线程启动时间；T2为线程体运行时间；T3为线程销毁时间。每次创建线程都会经过这三个时间会大大增加系统的响应时间。T2是无法避免的，只能通过优化代码来降低运行时间。T1和T3都可以通过线程池（Thread Pool）来缩短时间。线程池的实现涉及一下三个名词：

1. 工作线程（Worker），线程池中的线程只有两个状态：可运行状态和等待状态；
2. 任务接口（Task），每个任务必须实现的接口，以供工作线程调度器调度，它主要规定了任务的入口、任务执行完的场景处理、任务的执行状态等。这里的两种类型的任务：具有返回值（或异常）的Callable接口任务和无返回值并兼容旧版本的Runnable接口任务；

3、任务队列（Work Queue），也叫作工作队列，用于存放等待处理的任务，一般是BlockingQueue的实现类，用来实现任务的排队处理。线程池的创建过程：创建一个阻塞队列以容纳任务，在第一次执行任务时闯将足够多的线程（不超过许可线程数），并处理任务，之后每个工作线程自行从任务队列中获得任务，直到任务队列中任务数量为0为止，此时，线程将处于等待状态，一旦有任务加入到队列中，即唤醒工作线程进行处理，实现线程的可复用性）。

建议126：适时选择不同的线程池来实现；

（Java的线程池实现从根本上来说只有两个：ThreadPoolExecutor类和ScheduledThreadPoolExecutor类，还是父子关系。为了简化并行计算，Java还提供了一个Executors的静态类，它可以直接生成多种不同的线程池执行器，比如单线程执行器、带缓冲功能的执行器等，归根结底还是以上两个类的封装类）。

建议127：Lock与synchronized是不一样的；

（Lock类（显式锁）和synchronized关键字（内部锁）用在代码块的并发性和内存上时的语义是一样的，都是保持代码块同时只有一个线程具有执行权。显式锁的锁定和释放必须在一个try...finally块中，这是为了确保即使出现运行期异常也能正常释放锁，保证其他线程能够顺利执行。Lock锁为什么不出现互斥情况，所有线程都是同时执行的？原因：这是因为对于同步资源来说，显式锁是对象级别的锁，而内部锁是类级别的锁，也就是说Lock锁是跟随对象的，synchronized锁是跟随类的，更简单地说把Lock定义为多线程类的私有属性是起不到资源互斥作用的，除非是把Lock定义为所有线程共享变量。除了以上不同点之外，还有以下4点不同：1、Lock支持更细粒度的锁控制，假设读写锁分离，写操作时不允许有读写操作存在，而读操作时读写可以并发执行，这一点内部锁很难实现；2、Lock是无阻塞锁，synchronized是阻塞锁，线程A持有锁，线程B也期望获得锁时，如果为Lock，则B线程为等待状态，如果为synchronized，则为阻塞状态；3、Lock可实现公平锁，synchronized只能是非公平锁，什么叫做非公平锁？当一个线程A持有锁，而线程B、C处于阻塞（或等待）状态时，若线程A释放锁。JVM将从线程B、C中随机选择一个线程持有锁并使其获得执行权，这叫做非公平锁（因为它抛弃了先来后到的顺序）；若JVM选择了等待时间最长的一个线程持有锁，则为公平锁。需要注意的是，即使是公平锁，JVM也无法准确做到“公平”，在程序中不能以此作为精确计算。显式锁默认是非公平锁，但可以在构造函数中加入参数true来声明出公平锁；4、Lock是代码级的，synchronized是JVM级的，Lock是通过编码实现的，synchronized是在运行期由JVM解释的，相对来说synchronized的优化可能性更高，毕竟是在最核心不为支持的，Lock的优化需要用户自行考虑。相对来说，显式锁使用起来更加便利和强大，在实际开发中选择哪种类型的锁就需要根据实际情况考虑了：灵活、强大则选择Lock，快捷、安全则选择synchronized）。

建议128：预防线程死锁；

（线程死锁（DeadLock）是多线程编码中最头疼问题，也是最难重现的问题，因为Java是单进程多线程语言。要达到线程死锁需要四个条件：1、互斥条件；2、资源独占条件；3、不剥夺条件；4、循环等待条件；按照以下两种方式来解决：1、避免或减少资源贡献；2、使用自旋锁，如果在获取自旋锁时锁已经有保持者，那么获取锁操作将“自旋”在那里，直到该自旋锁的保持者释放了锁为止）。

建议129：适当设置阻塞队列长度；

（阻塞队列BlockingQueue扩展了Queue、Collection接口，对元素的插入和提取使用了“阻塞”处理。但是BlockingQueue不能够自行扩容，如果队列已满则会报IllegalStateException：Queue full队列已满异常；这是阻塞队列和非阻塞队列一个重要区别：阻塞队列的容量是固定的，非阻塞队列则是变长的。阻塞队列可以在声明时指定队列的容量，若指定的容量，则元素的数量不可超过该容量，若不指定，队列的容量为Integer的最大值。有此区别的原因是：阻塞队列是为了容纳（或排序）多线程任务而存在的，其服务的对象是多线程应用，而非阻塞队列容纳的则是普通的数据元素。阻塞队列的这种机制对异步计算是非常有帮助的，如果阻塞队列已满，再加入任务则会拒绝加入，而且返回异常，由系统自行处理，避免了异步计算的不可知性。可以使用put方法，它会等队列空出元素，再让自己加入进去，无论等待多长时间都要把该元素插入到队列中，但是此种等待是一个循环，会不停地消耗系统资源，当等待加入的元素数量较多时势必会对系统性能产生影响。offer方法可以优化一下put方法）。

建议130：使用CountDownLatch协调子线程；

（CountDownLatch协调子线程步骤：一个开始计数器，多个结束计数器：1、每一个子线程开始运行，执行代码到begin.await后线程阻塞，等待begin的计数变为0；2、主线程调用begin的countDown方法，使begin的计数器为0；3、每个线程继续运行；4、主线程继续运行下一条语句，end的计数器不为0，主线程等待；5、每个线程运行结束时把end的计数器减1，标志着本线程运行完毕；6、多个线程全部结束，end计数器为0；7、主线程继续执行，打印出结果。类似：领导安排了一个大任务给我，我一个人不可能完成，于是我把该任务分解给10个人做，在10个人全部完成后，我把这10个结果组合起来返回给领导--这就是CountDownLatch的作用）。

建议131：CyclicBarrier让多线程齐步走；

（CyclicBarrier关卡可以让所有线程全部处于等待状态（阻塞），然后在满足条件的情况下继续执行，这就好比是一条起跑线，不管是如何到达起跑线的，只要到达这条起跑线就必须等待其他人员，待人员到齐后再各奔东西，CyclicBarrier关注的是汇合点的信息，而不在乎之前或者之后做何处理。CyclicBarrier可以用在系统的性能测试中，测试并发性）。

## 第10章 性能和效率

建议132：提升Java性能的基本方法；

（如何让Java程序跑的更快、效率更高、吞吐量更大：1、不要在循环条件中计算，每循环一次就会计算一次，会降低系统效率；2、尽可能把变量、方法声明为final static类型，加上final static修饰后，在类加载后就会生成，每次方法调用则不再重新生成对象了；3、缩小变量的作用范围，目的是加快GC的回收；4、频繁字符串操作使用StringBuilder或StringBuffer；5、使用非线性检索，使用binarySearch查找会比indexOf查找元素快很多，但是使用binarySearch查找时记得先排序；6、覆写Exception的fillInStackTrace方法，fillInStackTrace方法是用来记录异常时的栈信息的，这是非常耗时的动作，如果不需要关注栈信息，则可以覆盖，以提升性能；7、不建立冗余对象）。

建议133：若非必要，不要克隆对象；

（克隆对象并不比直接生成对象效率高）（通过clone方法生成一个对象时，就会不再执行构造函数了，只是在内存中进行数据块的拷贝，看上去似乎应该比new方法的性能好很多，但事实上，一般情况下new生成的对象比clone生成的性能方面要好很多。JVM对new做了大量的系能优化，而clone方式只是一个冷僻的生成对象的方式，并不是主流，它主要用于构造函数比较复杂，对象属性比较多，通过new关键字创建一个对象比较耗时间的时候）。

建议134：推荐使用“望闻问切”的方式诊断性能；

（性能诊断遵循“望闻问切”，不可过度急躁）。

建议135：必须定义性能衡量标准；

（原因：1、性能衡量标准是技术与业务之间的契约；2、性能衡量标志是技术优化的目标）。

建议136：枪打出头鸟--解决首要系统性能问题；

（解决性能优化要“单线程”小步前进，避免关注点过多而导致精力分散）（解决性能问题时，不要把所有的问题都摆在眼前，这只会“扰乱”你的思维，集中精力，找到那个“出头鸟”，解决它，在大部分情况下，一批性能问题都会迎刃而解）。

建议137：调整JVM参数以提升性能；（

四个常用的JVM优化手段：

1、调整堆内存大小，JVM两种内存：栈内存（Stack）和堆内存（Heap），栈内存的特点是空间小，速度快，用来存放对象的引用及程序中的基本类型；而堆内存的特点是空间比较大，速度慢，一般对象都会在这里生成、使用和消亡。栈空间由线程开辟，线程结束，栈空间由JVM回收，它的大小一般不会对性能有太大影响，但是它会影响系统的稳定性，超过栈内存的容量时，会抛StackOverflowError错误。可以通过“java -Xss <size>”设置栈内存大小来解决。堆内存的调整不能太随意，调整得太小，会导致Full GC频繁执行，轻则导致系统性能急速下降，重则导致系统根本无法使用；调整得太大，一则浪费资源（若设置了最小堆内存则可以避免此问题），二则是产生系统不稳定的情况，设置方法“java -Xmx1536 -Xms1024m”，可以通过将-Xmx和-Xms参数值设置为相同的来固定堆内存大小；

2、调整堆内存中各分区的比例，JVM的堆内存包括三部分：新生区（Young Generation Space）、养老区（Tenure Generation Space）、永久存储区（Permanent Space 方法区），其中新生成的对象都在新生区，又分为伊甸区（Eden Space）、幸存0区（Survivor 0 Space）和幸存1区（Survivor 1 Space），当在程序中使用了new关键字时，首先在Eden区生成该对象，如果Eden区满了，则触发minor GC，然后把剩余的对象移到Survivor区（0区或者1区），如果Survivor取也满了，则minor GC会再回收一次，然后再把剩余的对象移到养老区，如果养老区也满了，则会触发Full GC（非常危险的动作，JVM会停止所有的执行，所有系统资源都会让位给垃圾回收器），会对所有的对象过滤一遍，检查是否有可以回收的对象，如果还是没有的话，就抛出OutOfMemoryError错误。一般情况下新生区与养老区的比例为1:3左右，设置命令：“java -XX:NewSize=32m -XX:MaxNewSize=640m -XX:MaxPermSize=1280m -XX:NewRatio=5”,该配置指定新生代初始化为32MB（也就是新生区最小内存为32M），最大不超过640MB，养老区最大不超过1280MB，新生区和养老区的比例为1:5.一般情况下Eden Space : Survivor 0 Space : Survivor 1 Space == 8 : 1 : 1）；

3、变更GC的垃圾回收策略，设置命令“java -XX:+UseParallelGC -XX:ParallelGCThreads=20”，这里启用了并行垃圾收集机制，并且定义了20个收集线程（默认的收集线程等于CPU的数量），这对多CPU的系统时非常有帮助的，可以大大减少垃圾回收对系统的影响，提高系统性能；

4、更换JVM，如果所有的JVM优化都不见效，那就只有更换JVM了，比较流行的三个JVM产品：Java HotSpot VM、Oracle JRockit JVM、IBM JVM。

建议138：性能是个大“咕咚”；

（1、没有慢的系统，只有不满足义务的系统；2、没有慢的系统，只有架构不良的系统；3、没有慢的系统，只有懒惰的技术人员；4、没有慢的系统，只有不愿意投入的系统）。

## 第11章 开源世界

建议139：大胆采用开源工具；

（选择开源工具和框架时要遵循一定的规则：1、普适性原则；2、唯一性原则；3、“大树纳凉”原则；4、精而专原则；5、高热度原则）。

建议140：推荐使用Guava扩展工具包；

（Guava（石榴）是Google发布的，其中包含了collections、caching、primitives support、concurrency libraries、common annotations、I/O等）。

建议141：Apache扩展包；

（Apache Commons通用扩展包基本上是每个项目都会使用的，一般情况下lang包用作JDK的基础语言扩展。Apache Commons项目包含非常好用的工具，如DBCP、net、Math等）。

建议142：推荐使用Joda日期时间扩展包；

（Joda可以很好地与现有的日期类保持兼容，在需要复杂的日期计算时使用Joda。日期工具类也可以选择date4j）。

建议143：可以选择多种Collections扩展；

（三个比较有个性的Collections扩展工具包：1、FastUtil，主要提供两种功能：一种是限定键值类型的Map、List、Set等，另一种是大容量的集合；2、Trove，提供了一个快速、高效、低内存消耗的Collection集合，并且还提供了过滤和拦截功能，同时还提供了基本类型的集合；3、lambdaj，是一个纯净的集合操作工具，它不会提供任何的集合扩展，只会提供对集合的操作，比如查询、过滤、统一初始化等）。

## 第12章 思想为源

建议144：提倡良好的代码风格；

（良好的编码风格包括：1、整洁；2、统一；3、流行；4、便捷，推荐使用Checkstyle检测代码是否遵循规范）。

建议145：不要完全依靠单元测试来发现问题；

（单元测试的目的是保证各个独立分隔的程序单元的正确性，虽然它能够发现程序中存在的问题（或缺陷、或错误），但是单元测试只是排查程序错误的一种方式，不能保证代码中的所有错误都能被单元测试挖掘出来，原因：1、单元测试不可能测试所有的场景（路径）；2、代码整合错误是不可避免的；3、部分代码无法（或很难）测试；4、单元测试验证的是编码人员的假设）。

**建议146：让注释正确、清晰、简洁；**

（注释不是美化剂，而是催化剂，或为优秀加分，或为拙略减分）。

**建议147：让接口的职责保持单一；**

（接口职责一定要单一，实现类职责尽量单一）（单一职责原则（Single Responsibility Principle，简称SRP）有以下三个优点：1、类的复杂性降低；2、可读性和可维护性提高；3、降低变更风险）。

建议148：增强类的可替换性；

（Java的三大特征：封装、继承、多态；说说多态，一个接口可以有多种实现方式，一个父类可以有多个子类，并且可以把不同的实现或子类赋给不同的接口或父类。多态的好处非常多，其中一点就是增强了类的可替换性，但是单单一个多态特性，很难保证我们的类是完全可以替换的，幸好还有一个里氏替换原则来约束。里氏替换原则：所有引用基类的地方必须能透明地使用其子类的对象。通俗点讲，只要父类型能出现的地方子类型就可以出现，而且将父类型替换为子类型还不会产生任何错误或异常，使用者可能根本就不需要知道是父类型还是子类型。为了增强类的可替换性，在设计类时需要考虑以下三点：1、子类型必须完全实现父类型的方法；2、前置条件可以被放大；3、后置条件可以被缩小）。

**建议149：依赖抽象而不是实现；**

（此处的抽象是指物体的抽象，比如出行，依赖的是抽象的运输能力，而不是具体的运输交通工具。依赖倒置原则（Dependence Inversion Principle，简称DIP）要求实现解耦，保持代码间的松耦合，提高代码的复用率。DIP的原始定义包含三层含义：1、高层模块不应该依赖底层模块，两者都应该依赖其抽象；2、抽象不应该依赖细节；3、细节应该依赖抽象；DIP在Java语言中的表现就是：1、模块间的依赖是通过抽象发生的，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的；2、接口或抽象类不依赖于实现类；3、实现类依赖接口或抽象类；更加精简的定义就是：面向接口编程。实现模块间的松耦合遵循规则：1、尽量抽象；2、表面类型必须是抽象的；3、任何类都不应该从具体类派生；4、尽量不要覆写基类的方法；5、抽象不关注细节）。

建议150：抛弃7条不良的编码习惯；

（1、自由格式的代码；2、不使用抽象的代码；3、彰显个性的代码；4、死代码；5、冗余代码；6、拒绝变化的代码；7、自以为是的代码）。

建议151：以技术人员自律而不是工人；

（20条建议：1、熟悉工具；2、使用IDE；3、坚持编码；4、编码前思考；5、坚持重构；6、多写文档；7、保持程序版本的简单性；8、做好备份；9、做单元测试；10、不要重复发明轮子；11、不要拷贝；12、让代码充满灵性；13、测试自动化；14、做压力测试；15、“剽窃”不可耻；16、坚持向敏捷学习；17、重里更重面；18、分享；19、刨根问底；20、横向扩展）。