代码优化的目标是：

1、减小代码的体积

2、提高代码运行的效率

一、类和对象使用技巧

1、尽量少用new生成新对象

用new创建类实例时,构造函数链中所有构造函数都会被自动调用,操作速度较慢。

但如果一个对象实现了 Cloneable 接口，可调用她的 clone() 方法。clone()方法不会调用任何类构造函数。

在某些时候可复用现有对象。比如在进行大量String操作时，可用StringBuffer娄代替String类，以避免生成大量的对象。

2、使用clone()方法生成新对象

如果一个对象实现Cloneable接口，就可以调用它的clone()方法。clone()方法不会泪用任何类构造函数，比使用new方法创建实例速度要快。

3、尽量使用局部变量（栈变量）

调用方法时传递的参数及调用中创建的临时变量保存在栈(Stack)中，速度较快。其他变量，如静态变量、实例变量都在堆(HeaP)中创建，速度较慢。访问静态变量和实例变量将会比访问局部变量多耗费 2-3 个时钟周期。如果一个变量需要经常访问，那么你就需要考虑这个变量的作用域了。static? local?还是实例变量？

4、减少方法调用

面向对象设计的一个基本准则是通过方法间接访问对象的属性，但方法调用会占用些开销访问。可以避免在同一个类中通过训州函数或方法(get或set)来设置或调用该对象的实例变量，这比直接访问变量要慢。为了减少方法调用，可以将方法中的代码复制到调用方法的地方，比如大量的循环中，这样可以节省方法调用的开销。但带来性能提升的同时会牺牲代码的可读性，可根据实际需要平衡两者关系。

5、使用final类和final、static、private方法

带有final修饰符的类是不可派生的。如果指定一个类为finaI，则该类所有的方法都是final。Java编译器会寻找机会内联(inIine)所有的final方法。此举能够提升程序性能。使用final、static、private的方法是不能被覆盖的，JAVA不需要在稃序运行的时候动态关联实现方法，从而节省了运行时间。

在 JAVA核心 API中，有许多应用 final 的例子，例如 java.lang.String。为 String 类指定 final 防止了使用者覆盖 length()方法。此举能够使性能平均提高 50%。

6、让访问实例内变量的 getter/setter 方法变成”final”

简单的 getter/setter 方法应该被置成 final，这会告诉编译器，这个方法不会被重载，所以，可以变成”inlined”

例子：

|  |
| --- |
| class maf {  public void setsize (int size) {  \_size = size;  }  private int \_size;  } |

|  |
| --- |
| 更正：  class daf\_fixed {  final public void setsize (int size) {  \_size = size;  }  private int \_size;  } |

7、避免不需要的 instanceof 操作

如果左边的对象的静态类型等于右边的，instanceof 表达式返回永远为 true。

例子：

|  |
| --- |
| public class uiso {  public uiso () {}  }  class dog extends uiso {  void method (dog dog, uiso u) {  dog d = dog;  if (d instanceof uiso) // always true.  system.out.println("dog is a uiso");  uiso uiso = u;  if (uiso instanceof object) // always true.  system.out.println("uiso is an object");  }  } |

|  |
| --- |
| 更正：删掉不需要的 instanceof 操作。  class dog extends uiso {  void method () {  dog d;  system.out.println ("dog is an uiso");  system.out.println ("uiso is an uiso");  }  } |

8、避免不需要的造型操作

所有的类都是直接或者间接继承自 object。同样，所有的子类也都隐含的“等于”其父类。那么，由子类造型至父类的操作就是不必要的了。

例子：

|  |
| --- |
| class unc {  string \_id = "unc";  }  class dog extends unc {  void method () {  dog dog = new dog ();  unc animal = (unc)dog; // not necessary.  object o = (object)dog; // not necessary.  }  }  更正：  class dog extends unc {  void method () {  dog dog = new dog();  unc animal = dog;  object o = dog;  }  } |

9.尽量重用对象。

特别是 String 对象的使用中，出现字符串连接情况时应使用 StringBuffer 代替，由于系统不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理。因此生成过多的对象将会给程序的性能带来很大的影响。

10、不要重复初始化变量。

默认情况下，调用类的构造函数时，java 会把变量初始化成确定的值，所有的对象被设置成null，整数变量设置成 0，float 和 double 变量设置成 0.0，逻辑值设置成 false。当一个类从另一个类派生时，这一点尤其应该注意，因为用 new 关键字创建一个对象时，构造函数链中的所有构造函数都会被自动调用。这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法比如initXXX()中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，public int state = this.getState();

11、不要过分创建对象

过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。JVM 的 GC并非十分智能，因此建议在对象使用完毕后，手动设置成 null。

二、Java I／O技巧

I／O性能常常是应用程序性能瓶颈，优化I／O性能就显得极为系要。在进行I／0操作时，匿遵循以下原则：尽可能少的访问磁盘；尽可能少的I方问底层的操作系统；尽可能少的方法调用；尽可能少处理个别的处理字节和字符。基于以上原则，可以通过以下技巧提高I／O速度：

1、使用缓冲提高I／O性能。

常用的实现方法有以下2种：使用用于字符的BufferedReader和用于宁节的BufferedlnputStream类，或者使用块读取方法次读取一大块数据。

2、lnputStream比Reader高效，OutputStream比Writer高效。

当使用Unicode字符串时，Write类的开销比较大。因为它要实Uoicode到字节(byte)的转换。因此，如果可能的话，在使用 Write类之前就实现转换或用OutputStream类代替Writer娄来使用。

3、在适当的时候用byte替代char。

1个char用2个字节保存字符，而byte只需要1个，因此用byte保存字符消耗的内存和需要执行的机器指令更少。更重要的是，用byte避免了进行Unicode转换。因此，如果可能的话，应尽量使用byte替代char。

4、有缓冲的块操作10要比缓冲的流字符10快。

对于字符IO，虽然缓冲流避免了每次读取字符时的系统调用开销，但仍需要一次或多次方法调用。带缓冲的块10比缓冲流IO快2到4倍，比无缓冲的IO快4到40倍。

5、序列化时，使用原子类型。

当序列化一个类或对象时，对于那些原子类型(atomic)或可以重建的元素，要标识为transient类型，这样就不用每一次都进行序列化。如果这些序列化的对象要在网络上传输，这一小小的改变对性能会有很大的提高。

6、在finally 块中关闭stream

程序中使用到的资源应当被释放，以避免资源泄漏。这最好在 finally 块中去做。不管程序执行的结果如何，finally 块总是会执行的，以确保资源的正确关闭。

7、SQL语句

在 java+Oracle 的应用系统开发中，java 中内嵌的 SQL 语言应尽量使用大写形式，以减少Oracle 解析器的解析负担。

8、尽早释放资源

java 编程过程中，进行数据库连接，I/O 流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。

三、异常(Exceptions)使用技巧

JAVA语言中提供了try／catch来开发方便用户捕捉异常，进行异常的处理。但是如果使用不当，也会给JAvA程序的性能带来影响。因此，要注意以下几点。慎用异常，异常对性能不利。抛出异常首先要创建一个新的对象。Throwable接口的构造函数调用名为fillInStackTrace()的本地(Native)方法filllnStackTrace()方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，VM就必须调整调用堆栈。

1、避免使用异常来控制程序流程

2、尽可能重用异常

在必须要进行异常的处理时，要尽可能的重用已经存在的异常对象。因为在异常的处理中，生成个异常对象要消耗掉大部分的时间。

3、将try/catch 块移出循环

把try/catch块放入循环体内，会极大的影响性能，如果编译 jit 被关闭或者你所使用的是一个不带 jit 的jvm，性能会将下降21%之多!

例子：

|  |
| --- |
| import java.io.fileinputstream;  public class try {  void method (fileinputstream fis) {  for (int i = 0; i < size; i++) {  try { // violation  \_sum += fis.read();  } catch (exception e) {}  }  }  private int \_sum;  } |

更正：将 try/catch块移出循环

|  |
| --- |
| void method (fileinputstream fis) {  try {  for (int i = 0; i < size; i++) {  \_sum += fis.read();  }  } catch (exception e) {}  } |

四、线程使用技巧

1、在使用大量线程(Threading)的场合使用线程池管理

生成和启动新线程是个相对较慢的过程，生成大量新线程会严重影响应用程序性能。通过使用线程池，由线程池管理器(thread pool manager)来生成新线程或分配现有线程，极大节省生成线程的时间。

2、防止过多的同步

不必要的同步常常会造成程序性能的下降，调用同步方法比调用非同步方法要花费更多的时间。如果程序是单线程，则没有必要使用同步。

3、同步方法而不要同步整个代码段

对某个方法或函数进行同步比对整个代码段进行同步的性能要好。

4、在追求速度的场合，用ArrayList和HashMap代替Vector和Hashtable

Vector和Hashtable实现了同步以提高线程安全性，但速度较比没有实现同步的ArrayList和HashMap要慢，可以根据需要选择使用的类。

5、使用notify()而不是notifyAll()

notify()只唤醒等待指定对象的线程，比唤醒所有等待线稃的notifyAll0速度更快。

6、不要在循环中调用 synchronized(同步)方法

方法的同步需要消耗相当大的资源，在一个循环中调用它绝对不是一个好主意。

更正：不要在循环体中调用同步方法，如果必须同步的话，推荐以下方式：

|  |
| --- |
| import java.util.vector;  public class syn {  public void method (object o) {  }  private void test () {  synchronized{//在一个同步块中执行非同步方法  for (int i = 0; i < vector.size(); i++) {  method (vector.elementat(i));  }  }  }  private vector vector = new vector (5, 5);  } |

7、单线程应尽量使用 HashMap, ArrayList。

除非必要，不推荐使用 HashTable,Vector，她们使用了同步机制，而降低了性能。

五、其它常用技巧

1、使用移位操作替代乘除法操作可以极大地提高性能

"/"是一个很“昂贵”的操作，使用移位操作将会更快更有效。

|  |
| --- |
| 例子：  public class sdiv {  public static final int num = 16;  public void calculate(int a) {  int div = a / 4; // should be replaced with "a >> 2".  int div2 = a / 8; // should be replaced with "a >> 3".  int temp = a / 3;  int mul = a \* 4; // should be replaced with "a << 2".  int mul2 = 8 \* a; // should be replaced with "a << 3".  int temp2 = a \* 3;  }  }  更正：  public class sdiv {  public static final int num = 16;  public void calculate(int a) {  int div = a >> 2;  int div2 = a >> 3;  int temp = a / 3; // 不能转换成位移操作  int mul = a << 2;  int mul2 = a << 3;  int temp = a \* 3; // 不能转换  }  } |

PS：除非是在一个非常大的循环内，性能非常重要，而且你很清楚你自己在做什么，方可使用这种方法。否则提高性能所带来的程序可读性降低将是不合算的。

2、对Vector中最后位置的添加、删除操作要远远快于对第一个元素的添加、删除操作

3、当复制数组时，使用System.arraycop()方法

|  |
| --- |
| 例子：  public class irb  {  void method () {  int[] array1 = new int [100];  for (int i = 0; i < array1.length; i++) {  array1 [i] = i;  }  int[] array2 = new int [100];  for (int i = 0; i < array2.length; i++) {  array2 [i] = array1 [i]; // violation  }  }  } |

更正：

|  |
| --- |
| public class irb  {  void method () {  int[] array1 = new int [100];  for (int i = 0; i < array1.length; i++) {  array1 [i] = i;  }  int[] array2 = new int [100];  system.arraycopy(array1, 0, array2, 0, 100);  }  } |

4、使用复合赋值运算符

a=a+b和a+b住编译时会产生不同JAVA字奇码，后者回快于前者。冈此，使用+=、-=、+=、／=等复台赋值运算符会使运算速度稍有提升。

5、用int而不用其它基本类型

对int类型的操作通常比其它基本类型要快，因此尽量使用int类型。

6、在进行数据库连接和网络连接时使用连接池

这类连接往往会耗费大量时间，应尽量避免。可使用连接池技术，复用现有连接。

7、用压缩加快网络传输速度一种常用方法是把相关文件打包到一个jar文件中。

用一个Jar文件发送多个文件还叫以避免每个文件打开和关闭网络连接所造成的开销。

8、在数据库应用程序中使用批处理功能

可以利用Statement类的addBatch()氟l exexuteBatch法成批地提交sql语句，以节省网络传输开销。在执行大量相似语句时，可以使用PreParedState—类，它可以一次性编译语句并多次执行，用参数最后执行的sql语句。

9、消除循环体中不必要的代码

这似乎是每个程序员都知道的基本原则，没有必出，但很多人往往忽略一些细节。

|  |
| --- |
| 如下列代码：  Vector aVector= ...；  for(int i=0；i<aVector size()；i++)(  System out printlll(aVector elementAt(i)toStringO)；  } |

|  |
| --- |
| 这段代码中没循环一次就要调用aVector.size()方法，aVector的长度不变的话，可以改为一下代码：  Vector aVector= ...：  int length=aVector size()；  for(int i=0；i<length；i++)f  System out println(aVector elememAt(i)toStringO)；  ) |

消除了每次调用aVector.size()方法的开销。

10、为'vectors' 和 'hashtables'定义初始大小

jvm 为 vector 扩充大小的时候需要重新创建一个更大的数组，将原原先数组中的内容复制过来，最后，原先的数组再被回收。可见 vector 容量的扩大是一个颇费时间的事。通常，默认的 10 个元素大小是不够的。你最好能准确的估计你所需要的最佳大小。

|  |
| --- |
| public vector v = new vector(20);  public hashtable hash = new hashtable(10); |

11、如果只是查找单个字符的话，用charat()代替startswith()

例子：

|  |
| --- |
| public class pcts {  private void method(string s) {  if (s.startswith("a")) { // violation  // ...  }  }  } |

|  |
| --- |
| public class pcts {  private void method(string s) {  if (s.startswith("a")) { // violation  // ...  }  }  } |

|  |
| --- |
| 更正 ：将'startswith()' 替换成'charat()'.  public class pcts {  private void method(string s) {  if ('a' == s.charat(0)) {  // ...  }  }  } |

12、在字符串相加的时候，使用 ' ' 代替 " "，如果该字符串只有一个字符的话

例子：

|  |
| --- |
| public class str {  public void method(string s) {  string string = s + "d" // violation.  string = "abc" + "d" // violation.  }  } |

|  |
| --- |
| 更正： 将一个字符的字符串替换成' '  public class str {  public void method(string s) {  string string = s + 'd'  string = "abc" + 'd'  }  } |

13、对于 boolean 值，避免不必要的等式判断

将一个 boolean 值与一个 true 比较是一个恒等操作(直接返回该 boolean 变量的值). 移走对于boolean 的不必要操作至少会带来 2 个好处：

1)代码执行的更快 (生成的字节码少了 5 个字节)；

2)代码也会更加干净 。

|  |
| --- |
| 例子：  public class ueq {  boolean method (string string) {  return string.endswith ("a") == true; // violation  }  }  更正：  class ueq\_fixed {  boolean method (string string) {  return string.endswith ("a");  }  } |

14、对于常量字符串，用'string' 代替 'stringbuffer'

如果确定这个 string 不会再变的话，这将会减少运行开销提高性能。

15、用'stringtokenizer' 代替 'indexof()' 和'substring()'

字符串的分析在很多应用中都是常见的。使用 indexof()和 substring()来分析字符串容易导致 stringindexoutofboundsexception。而使用 stringtokenizer 类来分析字符串则会容易一些，效率也会高一些。

例子：

public class ust {

void parsestring(string string) {

int index = 0;

while ((index = string.indexof(".", index)) != -1) {

system.out.println (string.substring(index, string.length()));

}

}

}

16、使用条件操作符替代"if (cond) else " 结构

条件操作符更加的简捷

|  |
| --- |
| 例子：  public class if {  public int method(boolean isdone) {  if (isdone) {  return 0;  } else {  return 10;  }    void method2(boolean istrue) {  if (istrue) {  \_value = 0;  } else {  \_value = 1;  }  }  更正：  public class if {  public int method(boolean isdone) {  return (isdone ? 0 : 10);  }    void method(boolean istrue) {  \_value = (istrue ? 0 : 1); // comp  private int \_value = 0;  } |

18、确定 stringbuffer的容量

stringbuffer 的构造器会创建一个默认大小(通常是 16)的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。在大多数情况下，你可以在创建 stringbuffer 的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。

19、不要总是使用取反操作符(!)

取反操作符(!)降低程序的可读性

20、与一个接口 进行instanceof 操作

基于接口的设计通常是件好事，因为它允许有不同的实现，而又保持灵活。只要可能，对一个对象进行 instanceof 操作，以判断它是否某一接口要比是否某一个类要快。

例子：

|  |
| --- |
| public class insof {  private void method (object o) {  if (o instanceof interfacebase) { } // better  if (o instanceof classbase) { } // worse.  }  }    class classbase {}  interface interfacebase {} |

21、采用在需要的时候才开始创建的策略。

|  |
| --- |
| 例如：  String str="abc";  if(i==1){ list.add(str);}  应修改为：  if(i==1){String str="abc"; list.add(str);} |

22、通过 StringBuffer 的构造函数来设定他的初始化容量，可以明显提升性能。

StringBuffer 的默认容量为 16，当 StringBuffer 的容量达到最大容量时，她会将自身容量增加到当前的 2 倍+2，也就是 2\*n+2。无论何时，只要 StringBuffer 到达她的最大容量，她就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给StringBuffer 设置一个合理的初始化容量值，是很有必要的！

23、合理使用 java.util.Vector

Vector 与 StringBuffer 类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。Vector 的默认存储能力为 10个元素，扩容加倍。 vector.add(index,obj) 这个方法可以将元素 obj 插入到index 位置，但 index 以及之后的元素依次都要向下移动一个位置（将其索引加 1）。 除非必要，否则对性能不利。 同样规则适用于 remove(int index)方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减 1）。返回此向量中移除的元素。所以删除 vector 最后一个元素要比删除第1 个元素开销低很多。删除所有元素最好用 removeAllElements()方法。 如果要删除 vector 里的一个元素可以使用 vector.remove(obj)；而不必自己检索元素位置，再删除，如 int index = indexOf（obj）;vector.remove(index)；

24、不要将数组声明为：public static final

因为这毫无意义，这样只是定义了引用为static final，数组的内容还是可以随意改变的，将数组声明为public更是一个安全漏洞，这意味着这个数组可以被外部类所改变

25、HaspMap 的遍历

|  |
| --- |
| Map<String, String[]> paraMap = new HashMap<String, String[]>();  for( Entry<String, String[]> entry : paraMap.entrySet() )  {  String appFieldDefId = entry.getKey();  String[] values = entry.getValue();  } |

利用散列值取出相应的 Entry 做比较得到结果，取得 entry的值之后直接取 key和 value。

26、array(数组)和 ArrayList 的使用。

array 数组效率最高，但容量固定，无法动态改变，ArrayList 容量可以动态增长，但牺牲了效率。

27、StringBuffer,StringBuilder 的区别

StringBuffer,StringBuilder 的区别在于：java.lang.StringBuffer 线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。StringBuilder 与该类相比，通常应该优先使用 StringBuilder 类，因为她支持所有相同的操作，但由于她不执行同步，所以速度更快。为了获得更好的性能，在构造 StringBuffer 或 StringBuilder 时应尽量指定她的容量。当然如果不超过 16 个字符时就不用了。 相同情况下，使用 StringBuilder 比使用 StringBuffer 仅能获得 10%~15%的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用 StringBuffer。

28、 尽量使用基本数据类型代替对象。

29、用简单的数值计算代替复杂的函数计算，比如查表方式解决三角函数问题。

30、使用具体类比使用接口效率高，但结构弹性降低了，但现代 IDE 都可以解决这个问题。

31、考虑使用静态方法

如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。她会被更快地调用，因为她不需要一个虚拟函数导向表。这同事也是一个很好的实践，因为她告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。

32.应尽可能避免使用内在的GET,SET 方法

android 编程中，虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外包调用的时候才使用 get，set方法，但在内部调用的时候，应该直接调用。

33、避免枚举，浮点数的使用。

34、二维数组比一维数组占用更多的内存空间，大概是 10 倍计算。

35、SQLite

SQLite 数据库读取整张表的全部数据很快，但有条件的查询就要耗时 30-50MS,大家做这方面的时候要注意，尽量少用，尤其是嵌套查找！

36、奇偶判断

不要使用 i % 2 == 1 来判断是否是奇数，因为 i为负奇数时不成立，请使用 i % 2 != 0 来判断是否是奇数，或使用高效式 (i & 1) != 0 来判断。

5、尽量减少对变量的重复计算

|  |
| --- |
| for (int i = 0, int length = list.size(); i < length; i++)  {...} |

9、如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度

比如ArrayList、LinkedLlist、StringBuilder、StringBuffer、HashMap、HashSet等等，以StringBuilder为例：

（1）StringBuilder()// 默认分配16个字符的空间

（2）StringBuilder(int size)// 默认分配size个字符的空间

（3）StringBuilder(String str)　// 默认分配16个字符+str.length()个字符空间

可以通过类（这里指的不仅仅是上面的StringBuilder）的来设定它的初始化容量，这样可以明显地提升性能。比如 StringBuilder吧，length表示当前的StringBuilder能保持的字符数量。因为当StringBuilder达到最大容量的时 候，它会将自身容量增加到当前的2倍再加2，无论何时只要StringBuilder达到它的最大容量，它就不得不创建一个新的字符数组然后将旧的字符数 组内容拷贝到新字符数组中—-这是十分耗费性能的一个操作。试想，如果能预估到字符数组中大概要存放5000个字符而不指定长度，最接近5000的2次幂 是4096，每次扩容加的2不管，那么：

（1）在4096 的基础上，再申请8194个大小的字符数组，加起来相当于一次申请了12290个大小的字符数组，如果一开始能指定5000个大小的字符数组，就节省了一倍以上的空间

（2）把原来的4096个字符拷贝到新的的字符数组中去

这样，既浪费内存空间又降低代码运行效率。所以，给底层以数组实现的集合、工具类设置一个合理的初始化容量是错不了的，这会带来立竿见影的效果。但 是，注意，像HashMap这种是以数组+链表实现的集合，别把初始大小和你估计的大小设置得一样，因为一个table上只连接一个对象的可能性几乎为 0。初始大小建议设置为2的N次幂，如果能估计到有2000个元素，设置成new HashMap(128)、new HashMap(256)都可以。

12、循环内不要不断创建对象引用

在循环体中实例化临时变量将会增加内存消耗

|  |
| --- |
| 例如：  for (int i = 1; i <= count; i++)  {  Object obj = new Object();  } |

会导致内存中有count份Object对象引用存在，count很大的话，就耗费内存了，

|  |
| --- |
| 建议为改为：  Object obj = null;for (int i = 0; i <= count; i++) { obj = new Object(); } |

这样的话，内存中只有一份Object对象引用，每次new Object()的时候，Object对象引用指向不同的Object罢了，但是内存中只有一份，就大大节省内存空间

13、基于效率和类型检查的考虑，应该尽可能使用array，无法确定数组大小时才使用ArrayList

14、尽量使用HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销

15、不要将数组声明为public static final

因为这毫无意义，这样只是定义了引用为static final，数组的内容还是可以随意改变的，将数组声明为public更是一个安全漏洞，这意味着这个数组可以被外部类所改变

16、尽量在合适的场合使用单例

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

（1）控制资源的使用，通过线程同步来控制资源的并发访问

（2）控制实例的产生，以达到节约资源的目的

（3）控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信

17、尽量避免随意使用静态变量

要知道，当某个对象被定义为static的变量所引用，那么gc通常是不会回收这个对象所占有的堆内存的，如：

|  |
| --- |
| public class A  {    private static B b = new B();    } |

此时静态变量b的生命周期与A类相同，如果A类不被卸载，那么引用B指向的B对象会常驻内存，直到程序终止

18、及时清除不再需要的会话

为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为30分钟。当应用服务器需要保存更多的会话时，如果内存不足，那么操作系统 会把部分数据转移到磁盘，应用服务器也可能根据MRU（最近最频繁使用）算法把部分不活跃的会话转储到磁盘，甚至可能抛出内存不足的异常。如果会话要被转 储到磁盘，那么必须要先被序列化，在大规模集群中，对对象进行序列化的代价是很昂贵的。因此，当会话不再需要时，应当及时调用HttpSession的 invalidate()方法清除会话。

19、实现RandomAccess接口的集合比如ArrayList，应当使用最普通的for循环而不是foreach循环来遍历

这是JDK推荐给用户的。JDK API对于RandomAccess接口的解释是：实现RandomAccess接口用来表明其支持快速随机访问，此接口的主要目的是允许一般的算法更改 其行为，从而将其应用到随机或连续访问列表时能提供良好的性能。实际经验表明，实现RandomAccess接口的类实例，假如是随机访问的，使用普通 for循环效率将高于使用foreach循环；反过来，如果是顺序访问的，则使用Iterator会效率更高。可以使用类似如下的代码作判断：

|  |
| --- |
| if (list instanceof RandomAccess)  { for (int i = 0; i < list.size(); i++){}  }else{  Iterator<?> iterator = list.iterable(); while (iterator.hasNext()){iterator.next()}  } |

foreach循环的底层实现原理就是迭代器Iterator，参见Java语法糖1：可变长度参数以及foreach循环原理。所以后半句”反过来，如果是顺序访问的，则使用Iterator会效率更高”的意思就是顺序访问的那些类实例，使用foreach循环去遍历。

20、使用同步代码块替代同步方法

这点在多线程模块中的synchronized锁方法块一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率。

21、将常量声明为static final，并以大写命名

这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量

22、不要创建一些不使用的对象，不要导入一些不使用的类

这毫无意义，如果代码中出现”The value of the local variable i is not used”、”The import java.util is never used”，那么请删除这些无用的内容

23、程序运行过程中避免使用反射

关于，请参见反射。反射是Java提供给用户一个很强大的功能，功能强大往往意味着效率不高。不建议在程序运行过程中使用尤其是频繁使用反射机制， 特别是Method的invoke方法，如果确实有必要，一种建议性的做法是将那些需要通过反射加载的类在项目启动的时候通过反射实例化出一个对象并放入内存—-用户只关心和对端交互的时候获取最快的响应速度，并不关心对端的项目启动花多久时间。

24、使用数据库连接池和线程池

这两个池都是用于重用对象的，前者可以避免频繁地打开和关闭连接，后者可以避免频繁地创建和销毁线程

25、使用带缓冲的输入输出流进行IO操作

带缓冲的输入输出流，即BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，这可以极大地提升IO效率

26、顺序插入和随机访问比较多的场景使用ArrayList，元素删除和中间插入比较多的场景使用LinkedList

这个，理解ArrayList和LinkedList的原理就知道了

27、不要让public方法中有太多的形参

public方法即对外提供的方法，如果给这些方法太多形参的话主要有两点坏处：

1、违反了面向对象的编程思想，Java讲求一切都是对象，太多的形参，和面向对象的编程思想并不契合

2、参数太多势必导致方法调用的出错概率增加

至于这个”太多”指的是多少个，3、4个吧。比如我们用JDBC写一个insertStudentInfo方法，有10个学生信息字段要插如Student表中，可以把这10个参数封装在一个实体类中，作为insert方法的形参

28、字符串变量和字符串常量equals的时候将字符串常量写在前面

这是一个比较常见的小技巧了，如果有以下代码：

String str = "123";

if (str.equals("123")) {

...

}

建议修改为：

String str = "123";

if ("123".equals(str))

{

...

}

这么做主要是可以避免空指针异常

29、请知道，在java中if (i == 1)和if (1 == i)是没有区别的，但从阅读习惯上讲，建议使用前者

平时有人问，”if (i == 1)”和”if (1== i)”有没有区别，这就要从C/C++讲起。

在C/C++中，”if (i == 1)”判断条件成立，是以0与非0为基准的，0表示false，非0表示true，如果有这么一段代码：

int i = 2;

if (i == 1)

{

...

}else{

...

}

C/C++判断”i==1″不成立，所以以0表示，即false。但是如果：

int i = 2;if (i = 1) { ... }else{ ... }

万一程序员一 个不小心，把”if (i == 1)”写成”if (i = 1)”，这样就有问题了。在if之内将i赋值为1，if判断里面的内容非0，返回的就是true了，但是明明i为2，比较的值是1，应该返回的 false。这种情况在C/C++的开发中是很可能发生的并且会导致一些难以理解的错误产生，所以，为了避免开发者在if语句中不正确的赋值操作，建议将 if语句写为：

int i = 2;if (1 == i) { ... }else{ ... }

这样，即使开发者不小心写成了”1 = i”，C/C++编译器也可以第一时间检查出来，因为我们可以对一个变量赋值i为1，但是不能对一个常量赋值1为i。

但是，在Java中，C/C++这种”if (i = 1)”的语法是不可能出现的，因为一旦写了这种语法，Java就会编译报错”Type mismatch: cannot convert from int to boolean”。但是，尽管Java的”if (i == 1)”和”if (1 == i)”在语义上没有任何区别，但是从阅读习惯上讲，建议使用前者会更好些。

30、不要对数组使用toString()方法

看一下对数组使用toString()打印出来的是什么：

public static void main(String[] args) {

int[] is = new int[]{1, 2, 3};

System.out.println(is.toString());

}

结果是：

[I@18a992f

本意是想打印出数组内容，却有可能因为数组引用is为空而导致空指针异常。不过虽然对数组toString()没有意义，但是对集合 toString()是可以打印出集合里面的内容的，因为集合的父类AbstractCollections<E>重写了Object的 toString()方法。

31、不要对超出范围的基本数据类型做向下强制转型

这绝不会得到想要的结果：

public static void main(String[] args)

{

long l = 12345678901234L;

int i = (int)l;

System.out.println(i);

}

我们可能期望得到其中的某几位，但是结果却是：

1942892530

解释一下。Java中long是8个字节64位的，所以12345678901234在计算机中的表示应该是：

0000 0000 0000 0000 0000 1011 0011 1010 0111 0011 1100 1110 0010 1111 1111 0010

一个int型数据是4个字节32位的，从低位取出上面这串二进制数据的前32位是：0111 0011 1100 1110 0010 1111 1111 0010

这串二进制表示为十进制1942892530，所以就是我们上面的控制台上输出的内容。从这个例子上还能顺便得到两个结论：

1、整型默认的数据类型是int，long l = 12345678901234L，这个数字已经超出了int的范围了，所以最后有一个L，表示这是一个long型数。顺便，浮点型的默认类型是 double，所以定义float的时候要写成”"float f = 3.5f”

2、接下来再写一句”int ii = l + i;”会报错，因为long + int是一个long，不能赋值给int

32、公用的集合类中不使用的数据一定要及时remove掉

如果一个集合类是公用的（也就是说不是方法里面的属性），那么这个集合里面的元素是不会自动释放的，因为始终有引用指向它们。所以，如果公用集合里面的某些数据不使用而不去remove掉它们，那么将会造成这个公用集合不断增大，使得系统有内存泄露的隐患。

33、把一个基本数据类型转为字符串，基本数据类型.toString()是最快的方式、String.valueOf(数据)次之、数据+”"最慢

把一个基本数据类型转为一般有三种方式，有一个Integer型数据i，可使用i.toString()、String.valueOf(i)、i+”"三种方式，三种方式效率如何，看一个测试：

|  |
| --- |
| public static void main(String[] args)  {  int loopTime = 50000;  Integer i = 0; long startTime = System.currentTimeMillis();  for (int j = 0; j < loopTime; j++) {  String str = String.valueOf(i);  }  System.out.println("String.valueOf()：" + (System.currentTimeMillis() - startTime) + "ms");  startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)  {  String str = i.toString();  }    System.out.println("Integer.toString():"+ (System.currentTimeMillis() - startTime) + "ms");    startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)    {  String str = i + "";  }    System.out.println("i + \"\"：" + (System.currentTimeMillis()- startTime)+ "ms");    } |

运行结果为：

String.valueOf()：11ms Integer.toString()：5ms i + ""：25ms

所以以后遇到把一个基本数据类型转为String的时候，优先考虑使用toString()方法。至于为什么，很简单：

1、String.valueOf()方法底层调用了Integer.toString()方法，但是会在调用前做空判断

2、Integer.toString()方法就不说了，直接调用了

3、i + “”底层使用了StringBuilder实现，先用append方法拼接，再用toString()方法获取字符串

三者对比下来，明显是2最快、1次之、3最慢

34、使用最有效率的方式去遍历Map

遍历Map的方式有很多，通常场景下我们需要的是遍历Map中的Key和Value，那么推荐使用的、效率最高的方式是：

|  |
| --- |
| public static void main(String[] args)  {  HashMap<String, String> hm = new HashMap<String, String>();  hm.put("111", "222");  Set<Map.Entry<String, String>> entrySet = hm.entrySet();  Iterator<Map.Entry<String, String>> iter = entrySet.iterator();  while (iter.hasNext()) {  Map.Entry<String, String> entry = iter.next();  System.out.println(entry.getKey() + "\t" + entry.getValue());  }  } |

如果你只是想遍历一下这个Map的key值，那用”Set<String> keySet = hm.keySet();”会比较合适一些

35、对资源的close()建议分开操作

意思是，比如我有这么一段代码：

try{

XXX.close();

YYY.close();

}catch (Exception e)

{

...

}

建议修改为：

try{ XXX.close(); }catch (Exception e) { ... }try{ YYY.close(); }catch (Exception e) { ... }

虽然有些麻烦，却能避免资源泄露。我们想，如果没有修改过的代码，万一XXX.close()抛异常了，那么就进入了cath块中 了，YYY.close()不会执行，YYY这块资源就不会回收了，一直占用着，这样的代码一多，是可能引起资源句柄泄露的。而改为下面的写法之后，就保 证了无论如何XXX和YYY都会被close掉