J2EE系统优化的几点体会（二、循环） PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/144/2021_2022_J2EE_E7_B3_BB_E7_BB_9F_c104_144709.htm 条例二：在循环处，多下功夫循环作为程序编写的基本语法，可以说是随处可见。一些小的细节能带来性能上的提升，而对循环体的一些改写，能带来性能的大幅提升。比如最简单的List遍历，会有这样的写法：for(int i=0.i 同样是对List的操作，如果要在遍历同时进行增加和删除操作，代码如下：for(int i=0,j=l.size().i=0.i--){l.remove(i).}。经过测试，如果采用ArrayList，两种写法在循环次数较少时没有太大的区别，循环次数为1000，均为1ms以内，次数为10000，前一种为60ms左右，后一种为1ms以内，，而次数上到100000，前一种为6000ms左右，后一种为15ms，随着循环次数的增多，后一种较前一种的效率优势明显提高。这是由Collection库ArrayList的实现决定的，以下是jdk1.3的ArrayList源码：public Object remove(int index) {RangeCheck(index).modCount .Object oldValue = elementData[index].int numMoved = size - index - 1.if (numMoved > 0)System.arraycopy(elementData, index 1, elementData, index,numMoved).elementData[--size] = null. // Let gc do its workreturn oldValue.}从中我们可以看出，numMoved代表了需要进行arraycopy操作的数量，它是由remove的位置决定的，如果index＝0，也就是删除第一个元素，则需要arraycopy后面的所有数据，而如果index＝size-1，则只需将最后一个元素设为null即可。所以从后面向前循环remove是比较好的写法。如

果List中的确存在较多的add或remove操作，且容量较大（如存储几万个对象），则应该采用LinkedList作为实现。LinkedList内部采用双向链表作为数据结构，比ArrayList占用较多内存空间，且随机访问操作较慢（需要从头或尾循环到相应位置），但插入删除操作很快（仅需进行链表操作，无须大量移动或拷贝）。对于List操作如果循环规模较小，其实对性能影响非常小（ms级），远远不是性能瓶颈所在。但心中有着优化的意识，并力求写出简洁高效的程序应该是我们每个程序员的追求。而且一旦在循环规模较大时，如果有了这些意识，也就能有效的消除性能隐患。再举一个与优化无关但确实可能成为性能杀手（可以说是bug）的循环的例子。下面是源代码：for(. totalRead {readBytes = m_request.getInputStream().read(m_binArray, totalRead, m_totalBytes - totalRead).}这个代码意图很清楚，就是将一个InputStream流读到一个byte数组中去。它使用read方法循环读取InputStream，该方法返回读取的字节数。正常情况下，该循环运行良好，当totalRead＝m_totalBytes时，结束循环，byte数组被正常填充。但如果仔细看一下InputStream的read方法的说明，了解一下其返回值就会发现，返回值可能为－1，即已读到InputStream末尾再继续读时。如果发生读取异常，可能出现这个问题，而这个循环没有检查readBytes值是否为－1就往totalRead上加，这样再次进入循环体继续读取InputStream，又返回－1，继续循环。如此循环直到int溢出才会跳出循环。而这个循环也就成了实实在在的CPU杀手，可以占去大量的CPU时间（取决于操作系统）。其实解决很简单，对readBytes进行判断，如果为-1则跳出循环。这个例子

告诉我们：对循环一定要搞清循环的循环规模、每次循环体执行时间、循环结束条件包括异常情况等，只有这样才能写出高效且没有隐患的代码。