Linux系统编程之C 游戏程序优化（4） 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_Linux_E7_B3_BB_E7_BB_c103_144828.htm 4、代码尺寸 编译器因为C 产生冗长的代码而臭名昭著。由于内存有限，而小的东西往往是快的，所以使你的可执行文件尽可能的小是非常重要的。首先可以做的事情是拿一个编译器来研究。如果你的编译器会在可执行文件中保存Debug信息的话，那么把它们移除掉。（注意MS VC会把Debug信息放在可执行文件外部，所以没关系）异常处理会产生额外的代码，尽可能的去除异常处理代码。确保连接器配置为去除无用的函数和类。开启编译器的最高优化级别，并尝试设置为尺寸最小化而不是速度最大化 有时候，由于Cache命中的提高，会产生更好的运行效果。（注意在使用这项设置时检查instrinsic功能是否也处于打开状态）去掉所有Debug输出状态下的浪费空间的字符串，使编译器把多个相同的字符串捆绑成一个实例。 内联通常是造成大函数的首犯。编译器可以自由的选择注意或忽略你写的inline关键字，而且它们还会背着你制造一些内联。这是另一个要你保持轻量级的构造函数的原因，这样堆栈中的对象就不会因为有大量的内联代码而膨胀。同时也要小心重载运算符，即使是最简短的表达式如m1=m2*m3如果m2和m3是矩阵的话，也可能产生大量的内联代码。一定要深入了解你的编译器对于内联的设置。 启用运行时类型信息（RTTI）需要编译器为每一个类产生一些静态信息。RTTI一般来说是缺省启用的，这样我们的代码可以调用dynamic_cast以及检测一个对象的类型，考虑完全禁止使用RTTI和dynamic_cast以节省空间（进一步

的说，有时候dynamic_cast在某些实现中需要付出很高的代价）另一方面，当你真的需要有基于类型的不同行为的时候，增加一个不同行为的虚函数。这是更好的面向对象设计（注意static_cast与这不同，它的效率和C语言的类型转换一样）。5、标准类库（STL） 标准类库是一套实现了常见的结构和算法的模板，例如dynamic arrays（称为vector），set，map等等。使用STL可以节省你很多时间来写和调试那些容器。和之前谈到的一样，如果希望系统的效率最大化，你必须要注意你的STL的具体实现的细节。 为了能够对应于最大范围的应用，STL标准在内存分配这个领域保持了沉默。在STL容器中的每一个操作都有一定的效率保证，例如，给一个set进行插入操作只要O(log n)的时间，但是，对一个容器的内存使用没有任何保证。 让我们来仔细了解游戏开发中的一个非常普遍的问题：你希望保存一组对象，（我们会称其为对象列表，虽然不一定要保存在STL的列表中）通常你会要求每个对象在这个表有且仅有一个，这样你就不用担心一个偶然产生的在容器中插入一个已存在单元的操作了。STL的set忽略副本，所有的插入、删除和查询的速度都是O(log n)，这是不是就是很好的选择呢？ 虽然在set上的大多数操作的速度都是O(log n)，但是这里面依然存在着潜在的危机。虽然容器的内存使用依赖于实现，但很多实现还是在红黑树的基础上实现的。在红黑树上，树的每一个节点都是容器的一个元素。常见的实现方法是在每一个元素被加入到树时，分配一个节点，而当每个元素被移出树时，释放一个节点。根据你插入和删除的频繁程度，在内存管理器上所花费的时间将或多或少的影响你通过使用set而获得的好处。 另外一个解决方案是使用vector

来存储元素，vector保证在容器的末端添加元素有很高的效率。这表示实际上vector只在很偶然的情况下才重新分配内存，也就是说，当满的时候扩容一倍。当使用vector来保存一个不同元素列表的时候，你首先要检查元素是否已经存在，如果没有，那么加入。而对整个vector检查一遍需要花费O(n)的时间，但是但实际牵涉到的部分应该比较少，这是因为vector的每个元素都在内存中连续存放，所以检查整个vector实际上是一个易于cache的操作。检查整个set将造成cache不命中，这是因为在红黑树上分别存放的元素可能散布在内存的各个角落。同时，我们也注意到set必须额外维护一组标记以设置整个树。如果你要保存的是对象的指针，set可能要花费vector所要花费的内存的3到4倍。 Set的删除操作消耗时间O(log n)，看起来是很快，如果你不考虑可能对free()的调用的话。Vector的删除操作消耗O(n)，这是因为从被删除的那个元素开始到结尾处的元素，每一个元素都要被拷贝到前一个位置上。如果元素都只是指针的话，那么这个拷贝将可以依靠一个简单的memcpy()来完成，而这个函数是相当快的。（这也是为什么通常都把对象的指针储存在STL的容器中的一个原因，而不是储存对象本身。如果你直接保存了对象本身，将会在很多操作中造成许多额外的构造函数的调用，例如删除等）。 set和map通常来说麻烦大于有用，如果你还没有意识到这一点的话，考虑遍历一个容器的代价，例如：
for(Collection::iterator it = Collection.begin(). it != Collection.end(). it)如果Collection是vector，那么 it就是一个指针自增。但是当Collection是一个set或者是一个map的话， it包括了访问红黑树上的下一个节点。这个操作相当复杂而且很

容易造成cache不命中，因为树的节点几乎遍布内存的各处。