如何编写异常安全的C 代码 PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/220/2021_2022__E5_A6_82_E4_BD_95_E7_BC_96_E5_c67_220270.htm 计算机等级考试训练软件《百宝箱》 关于C 中异常的争论何其多也，但往往是一些不合事实的误解。异常曾经是一个难以用好的语言特性，幸运的是，随着C 社区经验的积累，今天我们已经有足够的知识轻松编写异常安全的代码了，而且编写异常安全的代码一般也不会对性能造成影响。 使用异常还是返回错误码？这是个争论不休的话题。大家一定听说过这样的说法：只有在真正异常的时候，才使用异常。那什么是“ 真正异常的时候 ”？在回答这个问题以前，让我们先看一看程序设计中的不变式原理。 对象就是属性聚合加方法，如何判定一个对象的属性聚合是不是处于逻辑上正确的状态呢？这可以通过一系列的断言，最后下一个结论说：这个对象的属性聚合逻辑上是正确的或者是有问题的。这些断言就是衡量对象属性聚合对错的不变式。 我们通常在函数调用中，实施不变式的检查。不变式分为三类：前条件，后条件和不变式。前条件是指在函数调用之前，必须满足的逻辑条件，后条件是函数调用后必须满足的逻辑条件，不变式则是整个函数执行中都必须满足的条件。在我们的讨论中，不变式既是前条件又是后条件。前条件是必须满足的，如果不满足，那就是程序逻辑错误，后条件则不一定。现在，我们可以用不变式来严格定义异常状况了：满足前条件，但是无法满足后条件，即为异常状况。当且仅当发生异常状况时，才抛出异常。 关于何时抛出异常的回答中，并不排斥返回值报告错误，而且这两者是

正交的。然而，从我们经验上来说，完全可以在这两者中加以选择，这又是为什么呢？事实上，当我们做出这种选择时，必然意味着接口语意的改变，在不改变接口的情况下，其实是无法选择的(试试看，用返回值处理构造函数中的错误)。通过不变式区别出正常和异常状况，还可以更好地提炼接口。 对于异常安全的评定，可分为三个级别：基本保证、强保证和不会失败。 基本保证：确保出现异常时程序（对象）处于未知但有效的状态。所谓有效，即对象的不变式检查全部通过。 强保证：确保操作的事务性，要么成功，程序处于目标状态，要么不发生改变。 不会失败：对于大多数函数来说，这是很难保证的。对于C 程序，至少析构函数、释放函数和swap函数要确保不会失败，这是编写异常安全代码的基础。 首先从异常情况下资源管理的问题开始.很多人可能都这么干过: Type* obj = new Type. try{ do_something...} catch(...){ 0delete obj. throw.} 不要这么做!这么做只会使你的代码看上去混乱,而且会降低效率,这也是一直以来异常名声不大好的原因之一. 请借助于RAII技术来完成这样的工作: auto_ptr obj_ptr(new Type). do_something... 这样的代码简洁、安全而且无损于效率。当你不关心或是无法处理异常时,请不要试图捕获它。并非使用try...catch才能编写异常安全的代码,大部分异常安全的代码都不需要try...catch。我承认，现实世界并非总是如上述的例子那样简单，但是这个例子确实可以代表很多异常安全代码的做法。在这个例子中，boost::scoped_ptr是auto_ptr一个更适合的替代品。 现在来考虑这样一个构造函数： Type() : m_a(new TypeA), m_b(new TypeB){} 假设成员变量m_a和m_b是原始的指针类型，并且和Type内的申明顺序一

致。这样的代码是不安全的，它存在资源泄漏问题，构造函数的失败回滚机制无法应对这样的问题。如果new TypeB抛出异常,new TypeA返回的资源是得不到释放机会的.曾经,很多人用这样的方法避免异常: Type() : m_a(NULL), m_b(NULL){ auto_ptr tmp_a(new TypeA). auto_ptr tmp_b(new TypeB). m_a = tmp_a.release(). m_b = tmp_b.release(). } 当然,这样的方法确实是能够实现异常安全的代码的,而且其中实现思想将是非常重要的,在如何实现强保证的异常安全代码中会采用这种思想.然而这种做法不够彻底，至少析构函数还是要手动完成的。我们仍然可以借助RAII技术，把这件事做得更为彻底：shared_ptr m_a. shared_ptr m_b.这样,我们就可以轻而易举地写出异常安全的代码: Type() : m_a(new TypeA), m_b(new TypeB){} 如果你觉得shared_ptr的性能不能满足要求，可以编写一个接口类似scoped_ptr的智能指针类,在析构函数中释放资源即可。如果类设计成不可复制的，也可以直接用scoped_ptr。强烈建议不要把auto_ptr作为数据成员使用,scoped_ptr虽然名字不大好，但是至少很安全而且不会导致混乱。 