密级：公开

CAR(Component Assembly Runtime)构件自动测试的研究

Research of Automated Testing for CAR Modules

（申请清华大学工程硕士学位论文）

院（系、所）：计算机科学与技术系

专　　业：计算机应用技术

研　究　生：陈冬晓

校 内 导　师：钟玉琢

校 外 导　师：陈　榕

二零零五年十一月

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后遵守此规定）

作者签名：　　导师签名：

日　期：　　日　期：

摘要

随着构件技术的发展，构件的自动测试成为一个必不可少的环节，然而现有的自动测试技术具有如下缺陷：1．测试工具不能独立完成整个测试过程；2．编写测试用例是一项繁琐的任务；3．测试脚本常常需要编写和调试。满足不了构件的自动测试的需求。因此，构件的自动化测试可以帮助研究人员节省大量的宝贵时间，对提高构件的质量有积极的作用。本文针对 CAR 构件的自动测试进行了深入的研究，提出了相应新的方法并进行了系统实现。具体包括：

1.CAR Reflection。Reflection 是解析构件元数据的机制，这个机制允许程序在运行时透过 Reflection APIs 取得任何一个 class 的内部信息。本文根据 CAR 构件的特点，实现了一套供 CAR Reflection 使用的接口和 API。CAR Reflection 是实现整个 CAR 构件自动测试的基础，可以列出自动测试工具被测 CAR 构件的详细信息，起到了动态学习被测构件的作用，还可以在测试脚本中动态创建类实例和调用方法。CAR Reflection 除了使用在构件的自动测试外还应用到其他方面，如构建高度动态的系统。

2. 测试用例自动产生。对于 CAR 构件自动测试来说，测试用例就是测试数据。本文提出和实现了一个的基于边界分析法的自动产生测试用例方案，并将其用于三角形例子的测试中。该方案包含了两种情况：第一种情况是直接利用构件的参数类型的默认取值范围，直接产生测试用例文件，用于健壮性测试；第二种情况是利用用户输入的配置文件自动产生测试用例文件，可以比较完整地进行健壮测试和功能测试。测试用例的自动产生可以使测试人员不用手工编写大量的测试数据。

3.通用测试脚本。本文针对 CAR 构件可以动态加载的特点和 CAR Reflection 具有动态创建类实例和调用方法的功能，编写了一个通用的测试脚本。该脚本应用了数据驱动脚本的技术，可以直接读取测试用例文件的数据作为输入参数值和期望结

果，并把输出结果与期望结果相比较生成测试报告。实践证明，与传统测试工具比较，将不再需要根据每个测试用例或者某些特殊类别的测试提供测试脚本了，使用通用的测试脚本就可以满足所有的需求。

关键字：测试活动　CAR 构件　反射　测试用例　测试脚本。

Abstract

As the Component technology's development, automated component testing is becoming a necessary step. But the current automated testing technology has disadvantages: 1.testing tool can’t finish the whole testing process by itself; 2.designing testing case is hard work; 3.we always need to design and debug testing script. Because of these disadvantages, current testing tools can’t accomplish automated testing. So research of automated component testing is of great value—it can save programmer a lot of time and making components widely used. This thesis has done a deeply research on CAR component and demonstrate a CAR component testing flow. The content includes:

1. Achieved CAR Reflection. Based on the CAR features of binary inherit, interface program and self-description, designed and achieved a series of interface and APIs for users. CAR Reflection can enable the testing tools acquire the tested CAR component’s detail information, which is a process of dynamically study tested component and then realize the function of dynamically creating sample and call function for testing script. Besides automated component testing, CAR Reflection can be used in other areas as well such as building an advanced dynamic system.

2. Designed a program of automated producing testing case. Designed and implemented a program of producing testing case on the basis of the research of parameter type and ultimate analysis. With the reaction from the user, it can output a file including haleness testing case and function testing case file. This paper use a typical boundary value analysis case to prove this program is applicable.

3. Realized a universal testing script. Designed and realized a universal testing script on basis of research of CAR component characteristic and data driving script that can be use by all CAR component tests. Testing script tests the testing case file and tested component, then

reflect the result testing. This thesis proves the testing script is applicable by testing a real component and its testing case.

Key Words: testing action, CAR module, reflection, testing case, testing script

目录

第一章 引言

1.1 软件测试概述

1.1.1 软件测试背景

随着计算机技术的迅速发展和越来越广泛深入地应用于国民经济和社会生活的各个方面，随着软件系统的规模和复杂性与日俱增，软件的生产成本、软件中存在的缺陷和故障造成的各类损失也大大增加，甚至会带来灾难性的后果。

1999 年 4 月，软件缺陷导致一颗价值 12 亿美元的卫星在卡纳维尔角基地（Cape Canaveral）发射失败，这可能是软件史上造成损失最大的一次软件失效，这次事件引发了军民双方对美国空间发射程序的确底审查，包括软件集成和测试过程。

应此，软件质量问题成为所有使用软件和开发软件的人们不得不关注的一个问题。由于软件体现的是人脑的高度智能化，因此软件与人类的任何智慧成果一样可能存在着缺陷。预防和减少这些可能存在的问题的方法就是进行软件测试，测试是最有效的排除和防止软件缺陷与故障的手段。

软件测试就是在尽可能的所有的条件下执行应用程序，检验应用程序是否能完成预期任务的过程。在软件开发周期中，开发人员和测试人员必须协同工作，阶段性的进行软件测试，以便及时发现缺陷，修复缺陷，确保每个构件的正确性，从而确保整个软件的正确性。

现代社会对于软件要实现的功能的要求越来越复杂，软件开发技术日新月异的发展对软件测试技术提出了新的要求，新的测试理论、新的测试方法、新的测试手段不断涌现。

1.1.2 软件测试技术的发展

当前，软件测试技术主要包括了以下几个方面的内容，这几方面都在不断地快速、规范地发展。

1.软件验证技术

软件验证的目的用于证明软件生命周期的各个阶段以及各阶段的逻辑协调性和正确性。目前，软件验证技术还只是适用于特殊用途的小型程序。

2.软件静态测试

目前，软件测试正在逐渐地由对程序代码的静态测试向高层开发产品的静态测试方向发展，如静态分析工具的产生。所谓静态分析工具是在不执行程序的情况下，分析软件的特性。静态分析主要集中在需求文档、设计文档以及程序结构上，可以进行类型分析、接口分析、输入输出规格说明分析等。常用的静态分析工具有：ViewLog 公司开发的 LogiScope 分析工具，Software Research 公司研制的 TestWork/Advisor 分析工具等等。

3.测试数据的选择

在测试数据的选择方面，主要是对测试用例进行选择，这对测试的成功与否有着重要的影响。

通常从下面几个方面对测试用例的质量进行把握：

1) 检测软件缺陷的有效性。
2) 测试用例的可重用性。通过重用测试用例，进行修改后即可对其他内容进行测试，减轻测试用例的编写工作负担。
3) 测试用例的执行、分析和调试是否经济。
4) 测试用例的可维护性，即每次软件修改后对测试用例的维护成本控制。

目前已有测试数据生成工具出现并用于实际，例如 Bender & Associates 公司开发的功能测试数据生成工具 SoftTest：Parasoft 公司研制开发的 C/C++单元测试工具 Parasoft C++ test:International Software Automatic 公司提供的 Panorama C/C++测试数据生成工具等。

4.自动化测试技术

这是软件测试技术的最新发展方向，主要的目标是研究如何实现软件测试的自动化过程以及相关的一系列内容，具体表现是集成化测试系统。它将多种测试工具融为一体，合成为功能强大的测试工具。例如 Microsoft 公司开发的对 Windows 应用程序进行自动测试的集成化测试系统 Microsoft Test for Windows；Parasoft 公司研制开发的自动故障检测系统 Parasoft insure++等等。

1.2 自动化测试工具

长期以来，都是手工进行软件测试，即软件人员按预定义的过程运行应用程序。自从软件业兴起以来，人们为自动化软件测试过程做了很多工作，许多公司开发软件测试工具，用于检测缺陷。在产品发布前修正缺陷。这些工具在某些方面具有自动化的功能（例如实现逆工程和编写测试脚本），但经常具有如下的缺点：

- 测试脚本常常需要调试
- 很少有测试工具能够独立完成整个测试过程
- 这些工具实现的测试过程可能和软件设计过程不一致
- 逆工程过程和测试脚本产生过程是完全分开的两个过程

使用工具为每一个软件产品的每个成员产生和记录一个测试脚本，对测试人员来说，常常是一项任务繁重的工作；利用工具编辑和存档测试数据纯粹是手工完成。因此这些工具的自动化能力是有限的。

1.3 CAR 构件技术

CAR（Component Assembly Run-Time）构件技术是面向构件编程的编程模型，它规定了一组构件间相互调用的标准，使得二进制构件能够自描述，能够在运行时动态链接。

80 年代以来，目标指向型软件编程技术有了很大的发展，为大规模的软件协同开发以及软件标准化、软件共享、软件运行安全机制等提供了理论基础。

由于因特网的普及，构件可来自于网络，系统要解决自动下载，安全等问题。因此，系统中需要根据构件的自描述信息自动生成构件的运行环境，生成代理构件即中间件，通过系统自动生成的中间件对构件的运行状态进行干预或控制，或自动提供针对不同网络协议、输入输出设备的服务（即运行环境）。中间件编程更加强调构件的自描述和构件运行环境的透明性，是网络时代编程的重要技术。其代表是 CAR、JAVA 和.NET（C#语言）。

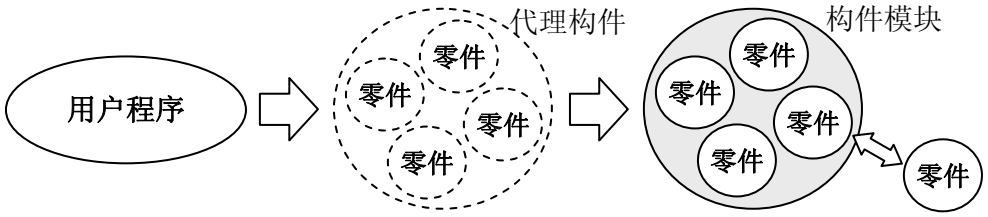

图 3-1 中间件运行环境的模型，动态生成代理构件

在这样的发展过程中，人们逐步深化了对大规模软件开发所需的科学模型、网络环境下软件运行必要机制的理解，使软件技术达到了更高的境界，实现了：

- 构件的相互操作性。不同软件开发商开发的具有独特功能的构件，可以确保与其他人开发的构件实现互操作。
- 软件升级的独立性。实现在对某一个构件进行升级时不会影响到系统中的其他构件。
- 编程语言的独立性。不同的编程语言实现的构件之间可以实现互操作。
- 构件运行环境的透明性。提供一个简单、统一的编程模型，使得构件可以在进程内、跨进程甚至于跨网络运行。同时提供系统运行的安全、保护机制。

CAR 构件技术就是在总结面向对象编程、面向构件编程技术的发展历史和经验，为更好地支持面向以 Web Service（WEB 服务）为代表的下一代网络应用软件开发而发明的。

为了在资源有限的嵌入式系统中实现面向中间件编程技术，同时又能得到 C/C++的运行效率，CAR 构件技术没有使用 JAVA 和.NET 的基于中间代码—虚拟机的机制，而是采用了用 C++编程，用和欣 SDK 提供的工具直接生成运行于和欣构件运行平台的二进制代码的机制。用 C++编程实现构件技术，使得更多的程序员能够充分运用自己熟悉的编程语言知识和开发经验，很容易掌握面向构件、中间件编程的技术。在不同操作系统上实现的和欣构件运行平台，可以使 CAR 构件的二进制代码可以实现跨操作系统平台兼容。

为了避免使用“中间件”这个有不同语义解释的词汇造成概念上的混淆，我们简单地将 CAR 技术统称为 CAR 构件技术。

1.4 研究内容与贡献

1.4.1 研究内容

通过对测试五个活动和对 CAR 构件特性的分析的基础上确定了要完成 CAR 构件自动测试必须对以下内容进行研究：

- 怎样通过读取 CAR 构件(.dll)的元数据，动态学习 CAR 构件；
- 怎样自动产生测试用例
- 怎样自动产生测试脚本；

1.4.2 本文内容安排

第一章“引言”介绍软件测试的概述、自动测试工具、CAR 构件技术的介绍及研究内容和贡献。

第二章“构件自动测试”介绍测试过程和那些活动可以自动化的。

第三章“获取被测构件的信息”介绍怎样设计和实现 CAR Reflection。

第四章“自动产生测试用例” 介绍自动产生测试用例。

第五章“自动产生测试脚本” 介绍通用测试脚本模块。

第六章“结论”对整个研究成果进行总结。

1.4.3 主要贡献

一、在自动测试中的作用

随着计算机软件技术的飞速发展，“软件工厂”的概念显得越来越重要。现代的软件已经发展成为一种产业，软件的构成有着大型化、工厂化的趋势。构件技术是

整个软件工厂的根基。一个构件完成之后或使用之前，需要对其进行二进制代码测试成为必要的一环，而自动化测试可以高效完成这一过程。而传统的自动测试技术满足不了构件的自动测试。

因此，研究构件的自动化测试对构件的技术的应用有积极的作用。并且可以使测试人员就不必手工编写乏味的脚本和经常维护测试脚本等工作投入大量的时间，从而可以把更多的时间投入到其他高风险的区域中。

二、在软件开发中作用

另外，根据构件自动测试可以动态产生测试用例和测试脚本的等特点，构件的自动测试可以应用到极限编程（XP）中。

根据定义，XP 是主要用于编程高风险软件项目的轻量级技术。XP 避免了详尽的规格说明，把每项任务都看做简单任务，通过频繁的迭代，和开发者、测试者、用户三者的交流，解决复杂问题。

自动测试对于成功进行 XP 实践是至关重要的。增加、修改或删除代码，都需要测试。代码的改变和进化通常是持续的，有时开发者并不了解发生的所有改变，因此，只有经常测试，才能确认代码的改变是有效的。

需求、规格和代码的不断改变，需要自动进行测试，从而确保这些改变不会导致系统崩溃。和其他开发模型产生的代码不同，XP 产生的代码是处于流动状态的，可以重新设计、重构、删除和完全重新编码之后必须进行测试，确保系统仍然能够工作。XP 需要对公有类接口和组件接口进行迭代测试，确保对实现的修改不影响接口协议。新的或修改的功能通过测试验证后，才能集成到系统中，必须测试任何导致潜在系统崩溃的位置。

第二章 构件自动测试

2.1 概述

测试过程包含在测试用例的开发生命周期中应该考虑的五个活动——标识、设计、建立、执行、检查。

如图 2-1 所示，在传统自动测试中，测试活动中的前三个测试活动，即标识测试条件、设计测试用例建立主要为智力活动。最后两个活动，即执行测试用例和比较测试输出相对来说是比较机械的活动。智力活动决定了测试用例的质量。机械活动是体力劳动则适合自动化。

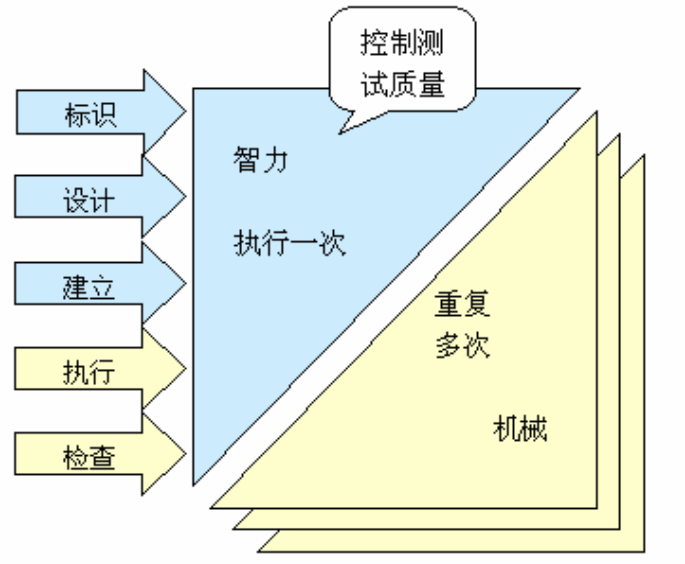

图 2-1 测试过程中的五个不同活动

所有的测试活动都可以手工进行，正如测试人员多年所做的那样。所有的测试活动也可以在某种程度上由于工具的支持而获得益处，但应该可能获利的活动进行自动化。

对 CAR 构件的自动化测试来说，由于 CAR 构件技术具有的二进制继承、面向接口编程和具有自描述的特点。这些特点可以突破了测试活动前三个活动人工参与的限制。

二进制继承、面向接口编程和具有自描述的这些特点使我们获得 CAR 信息成为可能，我们可以通过动态加载，然后通过接口来获得构件的自描述信息。这样就可以列出被测构件包含了哪些类、哪些接口和哪些方法函数，方法函数的参数数据类型是什么等等详细信息。通过这些信息，构件测试工具就能知道需要测试那些内容，这样就解决了测试活动的第一个活动的自动化问题。

通过第一步提供的方法函数的参数类型我们就可以按一定的方法构造出测试用例（测试数据）。通过 CAR 构件面向接口编程的特点，构件测试工具可以自动产生测试脚本（测试程序），这一步也可以在获测试活动第一步完成后自动完成。接着测试程序利用 CAR 二进制封装的思想动态调用所测试的方法函数来完成自动测试。

由于自描述信息只提供了方法的参数类型，参数的具体取值范围和各参数间取值的联系还不清楚，为了实现完全的构件自动测试必须需要用户的部分参与。对产生的测试数据进行编辑。

在软件开发生存周期中，需要不断测试经常改变的代码，检测并修正缺陷后，回归测试用于确认对缺陷的修改精确，且没有负面影响。

因此，如果构件自动测试需要考虑到软件开发的因素的话，可以分成 6 步。如图 2.2，迭代是以回归测试结束的。

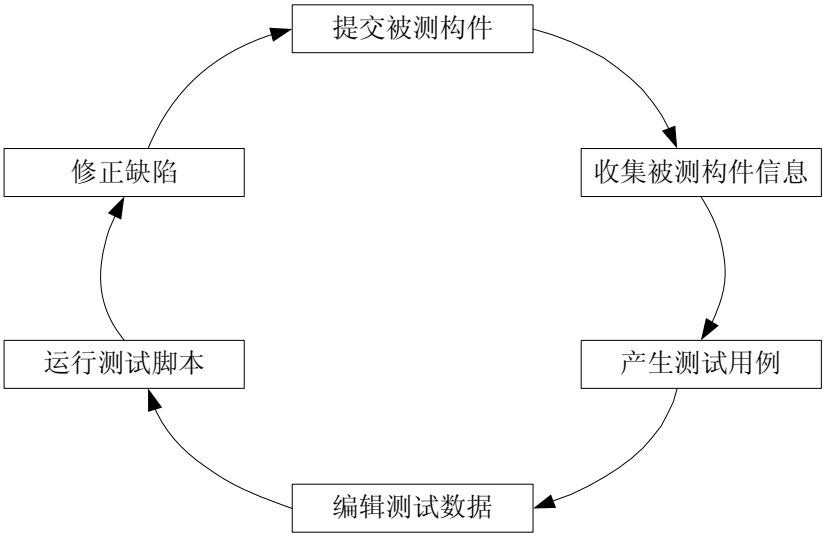

图 2.2 自动测试的 6 个步骤

2.2 测试活动

理想情况下，测试始于测试目标和测试策略的建立，测试策略应满足测试目标的要求，每个项目都有各自的测试策略。管理层的测试计划包括评估完成所有测试活动的时间，测试活动安排及资源安排，控制测试过程以及跟踪整个测试过程所需采取的活动。这些高层次活动应该在项目开始前就实施，并贯穿项目的整个开发过程。

图 2-3 给出了在活动序列中应该执行的关键活动。

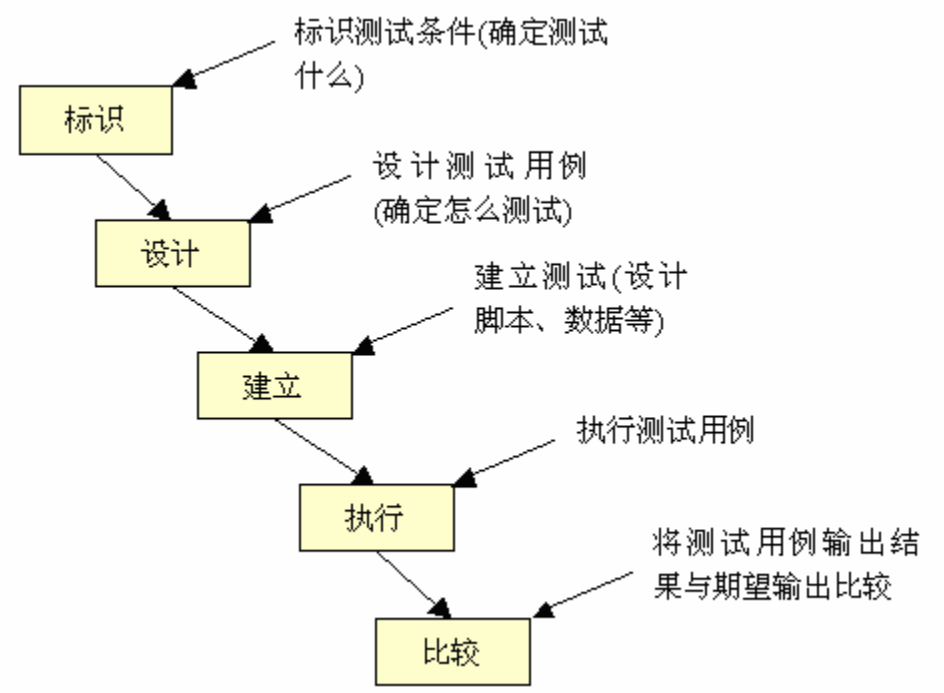

图 2-3 测试过程的活动

对上述五个活动说明如下：

(1) 标识测试条件。

第一个活动首先确定测试“什么”，也就是构件测试需要测试什么，因为构件的功能通过构件的类和方法函数来实现。所以需要确认构件具有哪些类，哪些方法函数。

(2) 设计测试用例。

设计测试用例确定“怎样”测试。测试用例是为特定的目的而设计的一组测试输入、执行条件和预期的结果。。测试用例设计将产生许多测试所包括的输入值、期望结果以其他任何运行测试的有关信息。在 CAR 构件测试中，测试用例是方法输入参数的值、输出参数的期望的结果和返回值。

每个测试都应说明期望输出。如果运行前没有说明期望输出，为检查软件的正确性，应首先认真验证实际输出。这就要求测试者对被测试构件具有一定的了解，才能对输出结果做出正确判断。如果结果正确，即可以将其与该测试以后的输出自动比较。这种方法称为参照测试(reference testing)。

(3) 建立测试脚步。

测试脚本是具有正规语法的数据和指令的集合，在测试执行自动工具使用中，通常以文件形式保存。一个测试脚本可以实现一个或多个测试用例的测试，测试脚本可以手工也可以不是手工执行(一个手工测试脚本就是一个测试过程)。测试输入和期望输出可包括在脚本中，也可以是脚本外的一个文件或数据库。在构件自动测试中测试输入和期望输出作为测试用例独立于测试脚本存在于文件或数据库中。

(4) 执行测试脚本。

在被测试构件运行时使用测试用例。对于手工测试来说，测试者按事先准备好的手工过程进行测试。测试者输入数据、观察输出、记录发现的问题。对于自动测试，可能只需启动测试工具，并告诉工具执行哪些测试用例。

(5) 分析与判断。

应该对每次测试的实际输出进行分析研究，判断软件功能是否正确。这种验证可以是非正式的测试者主观判断，也可以是将实际输出与期望输出进行严格准确的比较。一些信息比较，如可以在执行测试时进行显示屏幕上的信息。另一些输出比较，只能在测试执行结束后进行。自动测试一般结合了这两种方法。

一般情况下，假定如果实际输出与期望输出一致，则软件通过测试；如果不一致，则软件没有通过测试。这种规则过于简单化。如果实际输出与期望输出不一致，可能有多种因素：有可能是软件不正确，也有可能运行测试的顺序不对，或期望输出的结果不正确，或测试环境设置不正确，或测试定义不正确。

比较和验证两者是有区别的：工具可能具有比较功能，但不具有验证功能。工具可以将一系列测试结果与另外一些结果来比较，但不能断定输出是否正确，而这种验证活动通常要靠测试人员来完成。由测试者来确认或保证比较的测试结果是正确的。在一些特殊环境中，可能可以自动产生期望输出，但在大多数工业测试中，使用商用测试执行工具却行不通。

2.3 测试自动化

自动化测试也是一门技术，但与测试技术存在很大区别。许多商用机构发现自动化的测试比执行一次手工测试的开销大得多，可见如果希望从自动化测试中获得收益，则需要仔细选择和实现自动化测试。自动化的程度与测试的质量是独立的。

无论自动执行还是手工执行测试都不影响测试的有效性和仿效性。无论自动化测试做得如何出色，如果测试本身是失败的，那么测试结果也将毫无意义。自动化测试只对测试的经济性和修改性有影响。保证了测试的质量，通常情况下，自动化测试将要比手工测试经济得多，其开销只是手工测试的一小部分。自动化测试的方法越好，长期使用获得的收益就越大。

为实现高效的自动化测试，必须源于好的测试软件。测试可以是高质的或劣质的。这取决于测试者实现测试质量的技术。

同样，自动化质量也可以是高质的或劣质的。这取决于测试自动化的自动化技术，包括确定怎样方便地增加新的自动测试，如何维护自动测试以及测试自动化最终能提供什么样的效益。

第三章 CAR Reflection

3.1 概述

CAR 构件自动测试工具的目标是，通过分析 CAR 构件(.dll)元数据，自动构造测试脚本完成自动测试。

获取被测构件的元信息是整个测试工具的第一步也是最重要的一步。而通过什么方法来获取这些信息成为这一步的关键。

通过分析 CAR 构件的特点二进制继承、面向接口编程和具有自描述，我们需要实现一套机制来获得 CAR 构件的信息，并且提供调用 CAR 接口的 API。

JAVA 和.Net 获得元数据的是通过一套 Reflection 的机制来得到的，Reflection 中文翻译为反射。由于 CAR 构件和 JAVA、.Net 都是面向构件编程。所以在 CAR 构件技术上也可以实现一套反射机制。

通过实现反射机制可以很容易地获取构件的所有元信息，并且可以在脚本运行阶段，动态创建类实例，调用方法。

CAR Reflection 和 CAR 构件间的关系就像棱镜和太阳光的关系，棱镜分解太阳光，类似 CAR Reflection 把构件解析出类、接口和方法成员等，从而实现自动信息收集过程。这个过程类似于测试人员学习 CAR 构件的过程。正是有了反射，测试人员可以投入更多时间识别其他风险。

CAR 把类信息（Class Info）作为描述构件的元数据，是 CAR 文件的二进制表述。在 CAR 中，可以使用一个特殊的 CLSID 从构件中取出元数据信息，构件元数据的解释不依赖于其它的 DLL 文件。

这样我们就可以通过反射直接从 CAR 文件里取出元数据。CAR Reflection 定义了一套接口可以反射出这些元数据。如从构件模块了得数据的 IModuleInfo 接口、获得类信息的 IClassInfo 接口、获得接口信息的 IInterfaceInfo 接口等等。这些接口包含了可以满足用户需求的接口函数。

另外，还需要提供一套得到这些接口的 API。如直接加载 DLL 构件文件得到 IModuleInfo 接口的 API EzLoadModuleInfo 等。

3.2 CAR Reflection 设计

3.2.1 反射的概念

反射的概念是由 Smith 在 1982 年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。这一概念的提出很快引发了计算机科学领域关于应用反射性的研究。它首先被程序语言的设计领域所采用,并在 Lisp 和面向对象方面取得了成绩。其中 LEAD/LEAD++ 、OpenC++ 、MetaXa 和 OpenJava 等就是基于反射机制的语言。最近，反射机制也被应用到了视窗系统、操作系统和文件系统中。

反射本身并不是一个新概念，它可能会使我们联想到光学中的反射概念，尽管计算机科学赋予了反射概念新的含义，但是，从现象上来说，它们确实有某些相通之处，这些有助于我们的理解。在计算机科学领域，反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通过采用某种机制来实现对自己行为的描述（self-representation）和监测（examination），并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。可以看出，同一般的反射概念相比，计算机科学领域的反射不单单指反射本身，还包括对反射结果所采取的措施。

3.2.2 CAR 实现反射的条件

CAR 构件提供二进制继承、面向接口编程和包含了自描述信息。在 CAR 构件的每个接口的 QueryInterface 函数中提供了对运行时取得构件信息的接口功能。这些特点和功能为实现反射机制提供了必要的条件。

在 CAR 技术中，反射会是一种强大的工具。它使您能够创建灵活的代码，这些代码可以在运行时装配，无需在组件之间进行源代码链接。反射允许我们在编写与执行时，使程序代码能够接入 CAR 构件的内部信息，而不是源代码中选定的类协作的代码。这使反射成为构建灵活的应用的主要工具。

3.2.3 CAR Reflection 接口设计

作为 CAR 构件技术的一个组成部分， CAR Reflection 使用 CAR 构件技术来实现。也就是说需要定义出反射的各个接口。

在定义接口之前必须对 CAR 构件的提供的自描述进行分析，这样才能知道有需要涉及什么接口来包含各种信息。

CAR 构件由其所定义的接口、构件类及相关元数据组成，并以构件模块为封装和存储单位,并通过 dll 实现(每个 CAR 构件模块都是一个 dll)构件模块由 uunm（Universal Unique Name）唯一标识，在本机则以存储的文件名为标识。uunm 是一串 unicode 字符串，其格式如：[*length*][\000][*URL*][*CAR module name*]

length 是 uunm 的长度, *URL* 描述了 CAR module 的 WEB 路径，*CAR module name* 是 CAR 构件模块的名字，在本机上， URL 可以不存在。uunm 是 CAR 编译器自动生成的，它使得 CAR 构件的使用者不必关心 CAR 构件的所在位置。

接口是一组逻辑上相关的函数集合，是构件特征的抽象定义，是最基本的构件使用单位。从技术上讲，接口包含了一组函数的数据结构，通过这组数据结构，客户代码就可以调用构件对象的功能。接口定义了一组成员函数，这组成员函数是组件对象暴露出来的所有信息，客户程序利用这些函数获得构件对象的服务。

构件类是最基本的构件运行实体（指构件对象），一个构件模块可以封装一到多个构件类的实现。构件类的实例是构件对象， 构件对象是接口的实现，一个构件对象可以实现多个接口，一个接口可以被多个构件对象实现。

IID 是接口的标识，它由接口名、接口的方法名以及父接口名等等诸多因素决定，EZCLSID 是构件类的标识（EZCLSID 由构件类名字决定），它在构件模块内唯一。IID 和 EZCLSID 都是由 CAR 编译器自动生成，构件编写者只需定义接口名和构件类名就可以了。

客户端在实例化某个构件类时，如通过 EzCreateInstance 函数，客户传入构件类的 EZCLSID,CAR 构件加载程序会根据其中的 uunm 自动去加载相应的 CAR 构件，然后根据 CAR 构件的元数据创建出构件对象。CAR 构件对本身有足够的自描述能力，客户通过 CAR 构件运行平台使用 CAR 构件服务,直接使用即可,根本无需知道软件(CAR 构件）的存在。

所以为了从反射知道构件模块、构件类、接口、接口方法函数、方法参数等。就要定义如下的接口：

1. 访问构件模块信息的接口 IModuleInfo;
2. 访问构件类信息的接口 IClassInfo;
3. 访问构件接口信息的接口 IInterfaceInfo;
4. 访问方法函数信息的接口 IMethodInfo;
5. 访问方法参数信息的接口 IParameterInfo

6. 其他接口……

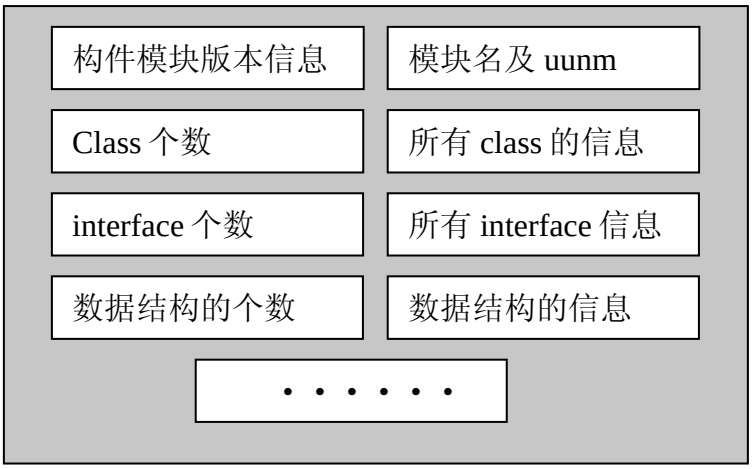

图 3-1 模块信息主要构成

图 3-1 种列出了构件模块的主要信息所以 IModuleInfo 包含了：

1) 获得 CAR 构件所有的类信息的接口函数 GetClasses
2) 通过名字获得相应的类信息的接口函数 GetClassByName
3) 获得 CAR 构件所有的接口信息的接口函数 GetInterfaces
4) 通过名字获得相应的接口信息的接口函数 GetInterfaceByName;
5) 其他的接口函数……

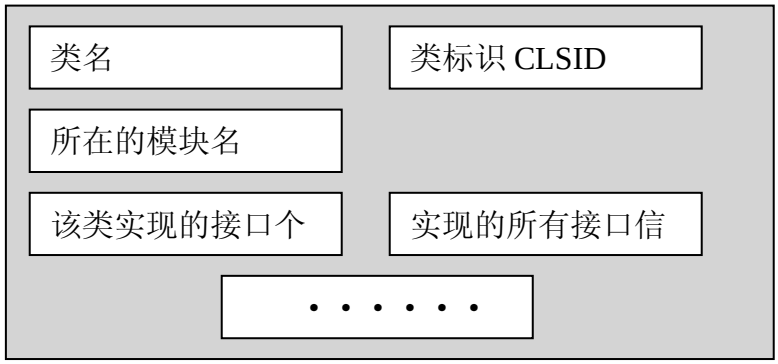

图 3-2 类信息主要构成

图 3-2 种列出了构件模块的主要信息所以 IClassInfo 包含了：

1) 获得类的所有支持的接口信息的接口函数 GetInterfaces
2) 通过名字获得相应的接口信息的接口函数 GetInterfaceByName
3) 获得父类的信息的接口函数 GetParentClass
4) 获得类的所有的方法信息的接口函数 GetMethods
5) 通过名字获得相应的方法信息的接口函数 GetMethodByName;
6) 其他的接口函数……

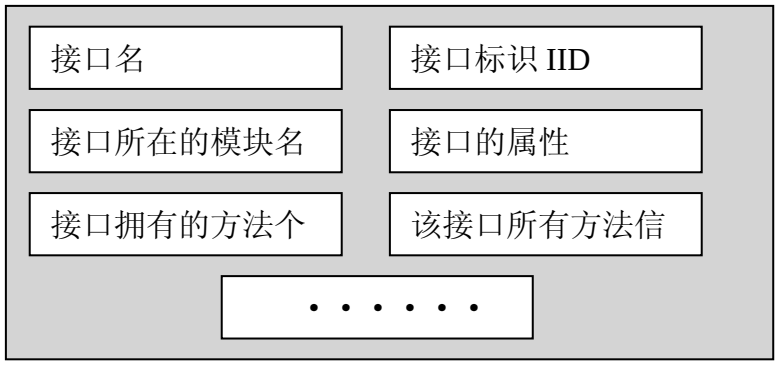

图 3-3 接口信息主要构成

图 3-3 种列出了构件模块的主要信息所以 IInterfaceInfo 包含了：

1) 获得类的所有的方法信息的接口函数 GetMethods
2) 通过名字获得相应的方法信息的接口函数 GetMethodByName;
3) 其他的接口函数……

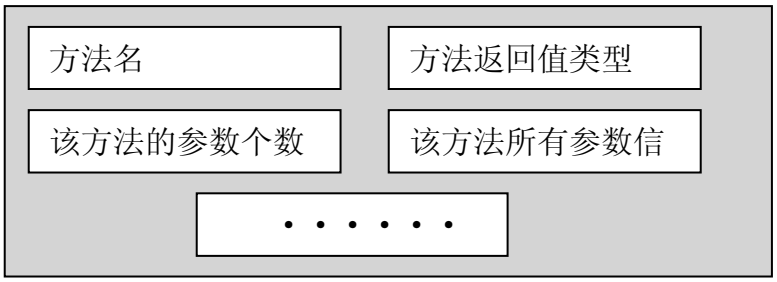

图 3-4 方法信息主要构成

图 3-4 种列出了构件模块的主要信息所以 IMethodInfo 包含了：

1) 获得类的所有的方法信息的接口函数 GetParameters
2) 通过调用本方法的接口函数 Invoke;
3) 其他的接口函数……

需要更详细的 CAR Reflection 构件信息。请参考附录的 Reflection.car 文件。

CAR 文件描述了一个构件里所包含的构件类的组织信息（如构件类的排列顺序、包含的接口）、各个接口的信息（如接口的种类、包含的接口方法）、各个方法的信息（如接口方法参数的种类、排列顺序等）以及接口和构件对象的标识等信息。

3.2.4CAR Reflection API 设计

CAR Reflection 提供了几个 API 来得到上面提到的接口提供给用户使用：
//通过构件的 DLL 文件的名字加载构件，并对模块信息的访问

```
STDAPI EzLoadModuleInfo ( /* [in] */ EzStr ezName,
      /* [out] */ IModuleInfo **piModuleInfo);

//对构件模块信息的访问
STDAPI EzReflectModuleInfo( /* [in] */ POBJECT pObject,
        /* [out] */ IModuleInfo **piModuleInfo);

//对构件类的信息的访问
STDAPI EzReflectClassInfo( /* [in] */ POBJECT pObject,
        /* [out] */ IClassInfo **piClassInfo);

//对接口信息的访问
    STDAPI EzReflectInterfaceInfo( /* [in] */ POBJECT pObject,
        /* [out] */ IInterfaceInfo *piInterfaceInfo);
```

3.3 CAR Reflection 实现

上一节在反射构件的设计中编写了一个 Reflection.car 文件，在 Reflection.car 文件里描述反射的构件接口，接口方法以及实现接口的构件类等等，CAR 工具会根据这个 CAR 文件生成代码框架以及其他构件运行时所需要的代码。通过这些代码框架，就可以实现构件类的各个方法的功能。

构件类是最基本的构件运行实体（指构件对象），一个构件模块可以封装一到多个构件类的实现。构件类的实例是构件对象， 构件对象是接口的实现，一个构件对象可以实现多个接口，一个接口可以被多个构件对象实现。

反射机制需要读取构件自描述的信息，然后通过上一节设计的接口提供给用户。

构件自描述是构件能够描述自己的信息数据，在构件技术中，强调构件的自描述，强调接口数据类型的自描述，以便于从二进制级上把接口与实现分离，并达到接口可以跨地址空间的目的。

元数据(metadata)，是描述数据的数据(data about data)， 首先元数据是一种数据,是对数据的抽象，它主要描述了数据的类型信息。

普通的源文件（C 或者 C++语言）经过编译器的编译产生二进制的文件，但在编译时编译器只提取了 CPU 执行所需的信息，忽略了数据的类型信息。比如一个指针，单看编译完之后的二进制代码或汇编已不能区分它是整型或是 char 型了，如果是指向字符串的指针，字符串的长度也无从知晓。这部分类型信息就属于我们所说的元数据信息。

CAR 构件以接口方式向外提供服务，构件接口需要元数据来描述才能让其他使用构件服务的用户使用。构件为了让接口与实现无关，从而保持了接口的不变性，使得动态升级成为可能；并且使用 vptr 结构将接口的内部实现隐藏起来，由接口的元数据来描述接口的函数布局和函数参数属性。接口的元数据描述的就是服务和调用之间的关系。有了这种描述，不同构件之间的调用才成为可能，构件的远程化，进程间通讯，自动生成 Proxy 和 Stub 及自动 Marshalling、Unmarshalling 才能正确地进行。

CAR 构件的元数据是 CAR 文件经过 CAR 编译器生成的，元数据与构件的实现代码一起被打包到构件模块文件中。元数据记录了构件接口及构件类的定义信息，是构件自描述的基础。

在 CAR 里，ClassInfo 被作为构件程序的元数据信息，用于描述构件导出的接口及方法列表。同时 ClassInfo 也是自动生成构件源程序的基础。

在目前的 CAR 构件开发环境下 ClassInfo 以两种形式存在：一种是与构件的实现代码一起被打包到构件模块文件中，用于列集和散集用的；另一种是以单独的文件

形式存在，存放在目标目录中，最终会被打包到 DLL 的资源段里，该文件的后缀名为 cls，如 hello.car 将会生成 hello.cls。这个 cls 文件和前者相比，就是它详细描述了构件的各种信息，而前者是一个简化了 ClassInfo，如它没有接口和方法名称等信息。cls 文件就是 CAR 文件的二进制版本。前者只是用于 CAR 构件库的实现，Reflection 需要的是后一种 ClassInfo。

对于每个 CAR 构件模块，ClassInfo 主要包括三大部分：构件模块信息、所有的构件类信息以及所有的接口信息，这些信息保存在 DLL 的资源段里。

EzLoadModuleInfo API，可以直接通过 EzLoadModule 把 DLL 加载到内存中，然后反射读取资源段的信息转化为方法输出参数的形式的满足用户的需要。

而对于正在运行的 CAR 构件，可以通过构件的其中一个接口的 QueryInterface 函数查询 IID_CLASS_INFO 这个特定接口，得到一个 EZCLSID 的指针，EzCLSID 是 CLSID 的扩展，EzCLSID 结构体中包含 uumm 信息。通过 uumm 的信息就可以知道构件模块的指针，最后读取资源段信息。

3.5 CAR Reflection 的应用

反射使得程序能够设计 CAR 构件的所有方面，不管是在开发时还是运行时，所以反射可以应用到很多方面。如，可以用于构件高度动态的系统。

在本文中反射就利用到获得构件信息，还可以在脚本运行过程中动态创建构件类的实例和调用方法。

3.5.1 CAR Reflection 在自动测试中的应用

一、获得构件信息

测试人员期望真实测试中，对于给定的被测构件，自动测试工具能自动获取所有的类、接口和方法。我们可以通过以下步骤可以列出构件相应的信息。

1. 使用 EzLoadModuleInfo API 得到 IModuleInfo 指针。
2. 通过 IModuleInfo 接口的 GetClasses 方法可以列举出所有类。
3. IClassInfo 的 GetMethods 可以列举出每个类的所有方法。
4. 通过 IMethodInfo 的 GetParameters 方法可以列举出每个方法的所有参数。
5. 参数可以通过 IParameterInfo 获得相应的特性。

二、脚本运行中应用

在脚本运行过程中动态创建构件类的实例和调用方法。

1. 通过 IClassInfo 接口的 CreateInstance 创建类的实例。
2. 通过 IMethodInfo 接口的 Invoke 方法调用相应的被测方法运行。

第四章 自动产生测试用例

4.1 概述

测试用例包含准备测试脚本、测试输入、测试数据以及期望输出。不同类别的软件，测试用例是不同的。对于 CAR 构件自动测试来说，我们的做法是把测试脚本从测试用例中划分出来。在本文中讨论的测试用例就是测试数据。

测试用例的设计为五个测试活动中的第二个活动。测试工具有很多方法可以进行部分测试用例自动化。但是自动工具不可能完全取代测试用例的设计活动。

由于 CAR 的核心思想是定义一组构件链接的标准，以及为这一标准提供的基础设施，其他高层的软件遵照这种标准来构造应用程序。这种构件式软件结构决定了它并不关心应用程序的内部结构，而是讨论如何实现应用程序和构件对象间的互操作。因此，当设计测试用例时，完全可以将 CAR 构件视为一个个接口的集合而只关心如何对这一个个的接口进行测试，从而可制定出一套规范来指导书写测试用例，从而实现自动设计测试用例。

CAR 构件是二进制封装的。所以对用户来说，CAR 构件是不透明的，相当于一个黑盒子。所以在设计测试用例的时候就按黑盒子的方法来设计。在测试时，把程序看作一个不能打开的黑盆子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否正常使用，程序是否能适当地接收输入数锯而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。

黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用。

由于 CAR 构件提供参数类型的自描述信息中，提供了各个类、方法、参数类型的描述，这样可以对每个方法进行测试。而测试每个方法必须需要输入不同的参数值和知道期望的值。

知道了参数的类型那样就可以在测试工具中增加类型的通用的取值范围。这样就可以根据边界值测试测试方法的功能和健壮性。这些测试数据都可以通过构件自动测试工具产生。

但是由于构件方法的参数取值并不是那么准确。例如，一个输入参数为 int，规格说明书上它只能为 0< a <100。由于 CAR 构件上没有相应的描述信息，那么通过 Reflection 获得的信息为 a 为输入参数类型为 int，如果没有人工的参与的话，构件自动测试工具只能对 a 为整型的取值范围内的测试，而不能为 a 自动生成实际取值 0< a <100 的测试。

所以，为了能让构件自动测试工具能比较准确地产生测试用例，测试工具增加了用户输入的功能，来达到完整测试。1.用户可以对某个参数限制取值范围。2.测试工具按参数的取值通过边界测试方法产生相应的输入数据，输出结果。并提供用户编辑这些数据的窗口，可以通过 XML 文档或 MS Excel 工作表编辑保存这些数据。这些数据提供给测试脚本使用，也可以在以后的回归测试中可以不断应用。

由于具有自动产生测试数据的功能，这样用户就不用输入大量的数据。在测试数百个方法的时候，可以节省测试人员大量的时间。

4.2 测试用例

影响软件测试的因素很多，例如软件本身的复杂程度、开发人员（包括分析、设计、编程和测试的人员）的素质、测试方法和技术的运用等等。因为有些因素是客

观存在的，无法避免。有些因素则是波动的、不稳定的，例如开发队伍是流动的，有经验的走了，新人不断补充进来；一个具体的人工作也受情绪等影响，等等。如何保障软件测试质量的稳定？有了测试用例，无论是谁来测试，参照测试用例实施，都能保障测试的质量。可以把人为因素的影响减少到最小。即便最初的测试用例考虑不周全，随着测试的进行和软件版本更新，也将日趋完善。

因此测试用例的设计和编制是软件测试活动中最重要的。测试用例是测试工作的指导，是软件测试的必须遵守的准则。更是软件测试质量稳定的根本保障。

什么叫测试用例？ 测试用例（Test Case）是为特定的目的而设计的一组测试输入、执行条件和预期的结果。测试用例是执行的最小实体。在构件自动测试中，测试用例就是输入参数的值和期望输出结果。

不同类别的软件，测试用例是不同的。不同于诸如系统、工具、控制、游戏软件，管理软件的用户需求更加不统一，变化更大、更快。对于 CAR 构件自动测试来说，我们的做法是把测试脚本从测试用例中划分出来。测试用例更趋于是针对软件产品的功能设计的测试方案。对软件的每个特定方法参数的测试构成了一个个测试用例。

测试是一种技术。对于任何系统而言，都存在着大量可能的测试用例，但实际上只能运行其中很少的一部分测试用例，并希望这些有限的测试用例都可以发现软件中的大部分缺陷。软件测试之所以能够降低软件开发成本，是因为测试能及时发现错误。软件开发过程中的每一个环节都有可能犯错误。而错误的代价是和错误发生和发现之间的时间成正比，发现的时间越晚，代价越大。

因此选择运行何种测试用例进行测试十分重要。试验和经验表明随机地选择测试用例并不是测试的有效方法，好的测试方法应该是开发好的测试用例。

有四个特性可以描述测试用例质量。其一，也许是最重要的一个方面，是检测软件缺陷的有效性，是否能发现缺陷，或至少可能发现缺陷。其二，好的测试用例应该是可仿效的。可仿效测试用例可以测试多项内容，因而减少了测试用例的数量。另外的两个方面是开销，即测试用例的执行、分析和调试是否经济，以及测试用例的修改性，即每次软件修改后对测试用例的维护成本。

通常对这四个方面要进行平衡。例如，每个测试用例可以测试很多内容，但其执行、分析和调试的开销可能很大，可能在每次软件修改后需要对测试用例进行大量维护。因此高仿效性有可能导致经济性和修改性较低。

因此测试技术不仅要保证测试用例具有发现缺陷的高可移植性，而且还要保证测试用例设计的经济有效性。

而通过构件的反射功能可以实现自动编写测试用例。这样能更加全面，更有效的完成构件的测试。

4.3 测试用例的设计

测试用例的设计是测试过程的一个关键步骤，若按照测试规划出发点的不同，软件测试方法可以分为黑盒测试和白盒测试两类。

对于构件来说，因为构件采用的是二进制封装，所以我们不可能自动一个构件里面的程序代码，如数据结构和算法等等。只能通过构件提供的元数据知道里面有哪些类、接口、方法和方法参数。所以与传统的单元测试不同，构件的单元测试采用黑盒测试方法。所以下面只介绍黑盒测试。

黑盒测试被称为功能测试或数据驱动测试。它是在已知产品所应具有的功能的基础上，通过测试来检测每个功能是否都能正常使用。在测试时，把程序视为一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特征的情况下进行。黑盒测试只

检查程序功能能否按照规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息（结果），并保持外部信息（如数据库或文件）的完整性。也就是说，测试不破坏被测对象的数据信息。

黑盒测试的具体技术方法主要包括边界测试分析方法、等价类划分法、比较测试法、因果图法、决策表法等。掌握和使用这些方法并不困难，但是每种方法各有所长，需要根据软件的具体特点，选择合适的测试方法，有效解决软件测试中的问题。

黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，所以很适合构件测试。

4.3.1 测试方法的选择

无数的测试实践表明，在设计测试用例时，一定要十分重视边界附近的处理，大量的故障往往发生在输入定义域或输出值域的边界上，而不是在其内部。为检验边界附近的处理而专门设计的测试用例，通常都会取得很好的测试效果。在任何情况下都必须采用边界值分析法。这种方法设计出的测试用例发现程序错误的能力最强。

CAR 构件测试来说，边界值分析法是比较好的测试数据产生法。

4.3.2 边界值分析法

应用边界值分析法设计测试用例，首先要确定边界情况。输入等价类和输出等价类的边界，就是要这种测试的边界情况。

边界值分析方法的基础思想是：选取正好等于、刚刚大于或小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。边界值分析法是最有效的黑盒分析方法，但在边界情况复杂的情况下，要找出适当的边界测试用例还需要针对问题的输入域、输出域边界，耐心细致地逐个进行考察。

边界分析测试的基本原理是故障往往出现在输入变量的边界值附件。例如，当一个循环条件为“≤”时，错写成为“<”记数器发生了少记一次的情形等。

边界分析法是基于可靠性理论中称为“单故障”的假设，即有两个或两个以上故障同时出现而导致软件失效的情况很少，也就是说，软件失效基本上是由单故障引起的。因此，边界值分析利用输入变量的最小值(min)、略大于最小值(min+)、输入值域内的任意值（nom）、略小于最大值（max-）和最大值(max)来设计测试用例。

健壮性测试是边界值分析测试的一种扩展，除了取 5 个边界值外，还需要考虑采用一个略超过最大值(max+)以及略小于最小值(min-)，检查超过极限值时系统的情况。健壮性测试最有意义的部分不是输入，而是预期的输出。要观察例外情况如何处理，比如某个部分的负载能力超过其最大值时可能出现的情形。

4.4 测试用例设计的自动化

有时，特别是当被测试构件有数百个成员时，通过手工为构件的每个成员都编写一个测试用例是非常乏味的。

由于 CAR Reflection 可以提供方法的数据类型，所以 CAR 构件测试工具可以通过数据类型缺省的边界值为被测构件的方法自动按边界值分析法产生测试用例。测试构件方法的健壮性。

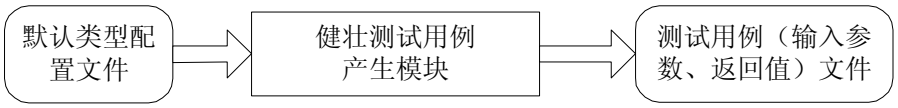

图 4-1 健壮测试用例产生数据流程图

其中，健壮测试用例产生模块，程序流程图如下：

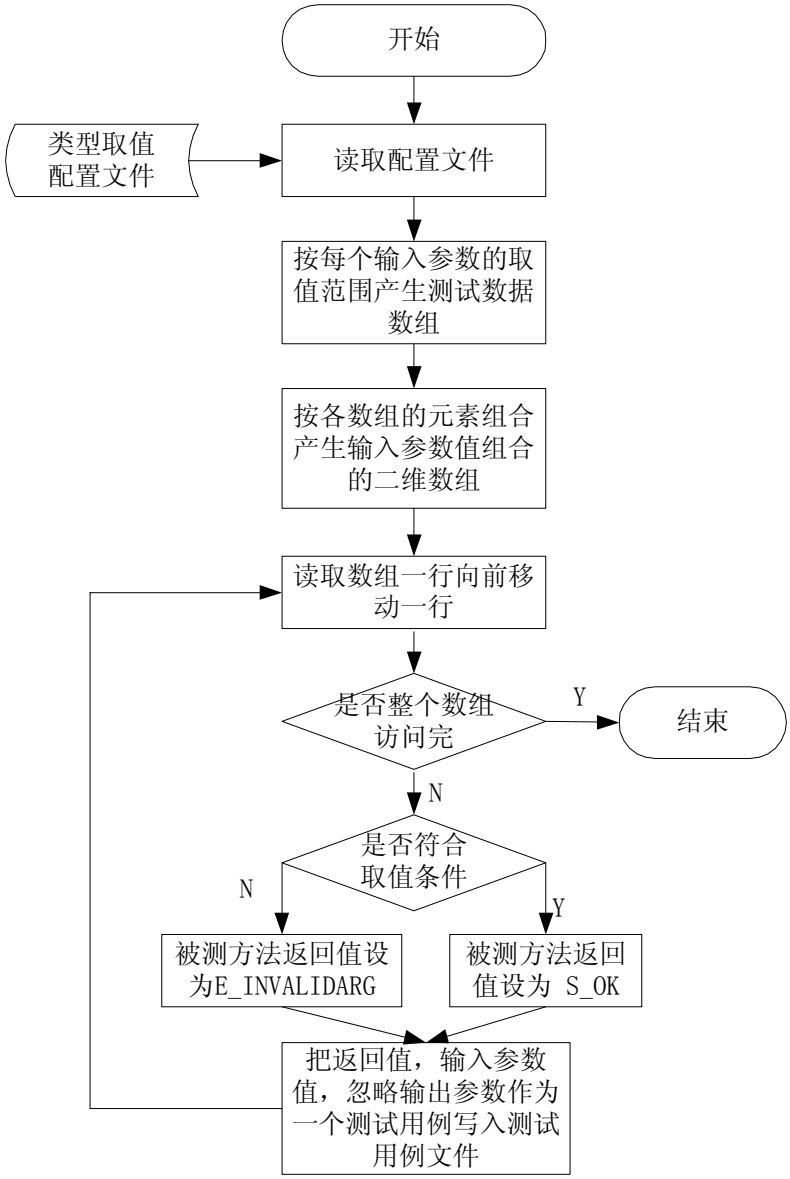

图 4-2 健壮测试用例产生模块程序流程图

由于目前的 CAR 构件没有为每个参数提供特定的取值范围、取值条件和期望输出。所以要完成完全测试必须需要测试人员输入取值范围、取值条件和期望输出。构件自动测试的的测试用例产生模块就可以为构件产生测试用例。所以，图 4-1 改成图 4-3。

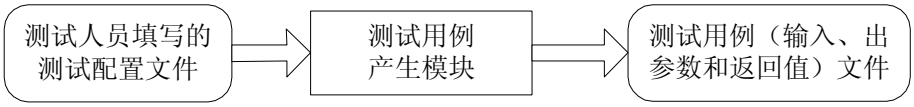

图 4-3 测试用例产生数据流程图

其中，测试用例产生模块程序流程由图 4-2 编成图 4-4。

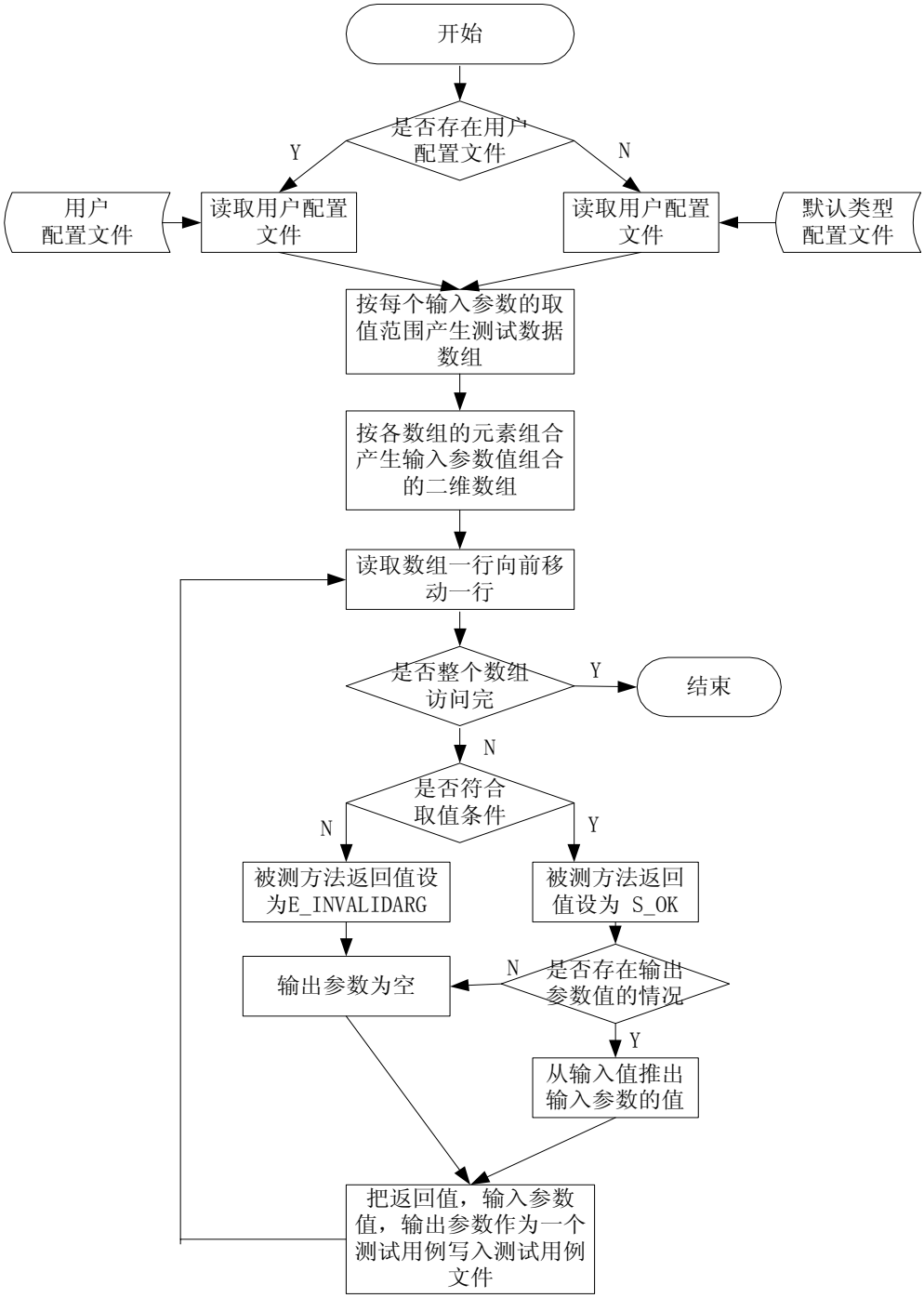

图 4-4 测试用例产生模块程序流程图

4.5 一个典型的例子

以下通过一个典型的边界分析法的测试例子来解释测试用例自动产生模块的工作流程。

构件测试工具可以使用 XML 文档或 MS Excel 工作表存储数据，进而编辑数据。XML 已经被许多工业标准所接受，它还能够跨开发环境，跨平台使用。

下面通过具有代表性的三角形问题怎么利用自动产生测试用例程序自动产生构件测试用例。

三角形问题。输入三个整数 a、b、c，分别作为三角的三条边，现通过程序的判断由三条边构成的三角形的类型为：等边三角形、等腰三角形、一般三角形（特殊的还有直角三角形），以及构成不成三角形。

现要求输入三个整数 a、b、c，必须满足以下条件：

条 1 1≤ a ≤100

条 2 1≤ b ≤100

条 3 1≤ c ≤100

条 4 a < b + c

条 5 b < a + c

条 6 c < a + b

如果输入值不满足这些条件中的任何一个，程序给出相应的信息，返回无效参数：如果 a、b、c 满足条件 1、条件 2 和条件 3，则输出下列四种情况之一：

1. 如果不满足条件 4、条件 5 和条件 6 中的一个，则程序输出为“非三角形”。
2. 如果三条边相等，则程序输出为“等边三角形”
3. 如果只有两条边相等，则程序输出为“等腰三角形”。

4. 如果三条边都不相等，则程序输出为“一般三角形”

这四种情况是相互排斥的。

三角形问题包含了清晰而又复杂的逻辑关系。

函数方法为：HRESULT triangle(int a, int b, int c, triangle_type type);

其中 triangle_type 为枚举类型，值为{not_tranle, equilateral_triangle, isosceles_triangle, normal_triangle}

在健壮性测试中，需要在用户配置文件中输入 6 个条件。健壮性测试测试模块按健壮性测试方法产生测试用例，如 a 的取值有：0、1、2、99、100、101 和 1 到 100 之间的一个随机数共 7 个值。然后，b、c 一样产生 7 个数。接着把它们所有的组合输出到文件,忽略输出参数,返回值如果不符合上面的六个条件,返回值为 E_INVALIDARG,否则返回值为 S_OK。

功能测试中，需要在用户配置文件写入输出的 4 种情况，功能测试测试模块按功能性测试方法在前面健壮性测试用例的基础上填入期望的输出参数的值，只对返回值为 S_OK 的测试用例进行修改，而返回值为 E_INVALIDARG 项不做处理。

输出的测试用例文件名为，被测构件名_构件类_方法.cas，输出内容为：

E_INVALIDARG，0， 0， 0，

E_INVALIDARG，0， 0， 1，

…

S_OK, 1, 1, 1, equilateral_triangle

S_OK, 1, 1, 1, isosceles_triangle

…

这样就可以较全面的测试到三角形的这个方法了。

由于目前还没有编辑界面，所以测试人员需要编辑测试用例，必须打开测试用例输出的那个测试用例文件来编辑。

第五章 通用的测试脚本

5.1 概述

脚本实际上是一种计算机程序的形式，一组测试工具执行的指令集合。

对于建立测试程序脚本，人们最关心的是建立及维护脚本的代价以及从中获得的益处。如果脚本被大量生命周期较长的不同测试复用，则应该保证该脚本的合理性和维护性。

在 CAR 构件自动测试中，测试脚本可以根据 CAR Reflection 提供的被测构件的信息，自动产生脚本。CAR Reflection 动态加产生类实例和调用类方法的方法可以应用到编写的测试脚本中。测试脚本就可以动态调用被测方法，并根据构件、构件类、方法名这三者组合的测试用例文件名读入被测试方法的测试用例，进行测试并且对比测试用例期望结果产生测试报告。这样一个测试脚本就可以完成所有的测试工作了。也就说测试工具编写和编译好的测试脚本可以应用到任何一个被测构件。用户不用编写测试脚本就可以完成所有的构件测试。用户也可以修改这个测试脚本代码来定制自己的测试脚本。用户不用编写测试脚本就可以完成所有的构件测试。

5.2 测试脚本

脚本实际上是一种计算机程序的形式，一组测试工具执行的指令集合。脚本的具体内容依赖于使用的测试工具及脚本技术。

测试脚本模拟用户使用，基于预描述的场景运行应用程序，从而自动测试软件产品。通常，用于测试构件方法的自动测试是由测试人员编写的。测试人员必须首先

花时间定义和分析需求，然后才能编写自动测试脚本。在软件开发过程中，改变了一个或多个需求，都需要测试人员修改或重写测试脚本，重复这个过程，直到产品稳定，然后交互运行测试脚本，实现单元测试、集成测试或回归测试。在整个测试过程中，测试组的任务是改善和调试测试脚本，满足不断改变的测试需求。

为了简化乏味耗时的手工编写测试脚本的过程，构件自动测试工具能够自动检测改变软件定义，然后依据发生的改变编写新的测试脚本。若整个开发生存周期中，软件规格都保持稳定，则可以在整个开发生存周期的单元测试、集成测试和回归测试中从新运行测试脚本。

开发好的软件产品所依据的规则，也可以应用到开发好的测试脚本中。构件自动测试工具利用了反射机制，所以可以产生高效的测试脚本。构件自动测试工具产生的测试脚本模板应该具有下面的特性：

可重用性

开发测试工具需要很大的代价，若开发的工具满足工程需求，而且可重用于将来的工程，则这样的代价也是值得。本文遵循传统的软件开发管理，使用 C++作为面向对象语言，自动产生可重用测试脚本。可重用工具和测试脚本通过类、接口和方法研究被测构件，一个测试用例的测试脚本可以用于测试另外的测试用例，最后，针对不同工程，工具本身可以重用和升级。

可读性

产生的测试脚本的变量和常量的命名遵循标准命名惯例，从而可以较容易地复查代码。测试脚本精确测试软件构件的特性，若对软件产品提出一些特殊的需求，测试人员可以选择升级测试工具或修改产生的测试脚本。例如，若这些特殊的需求经常

发生，对测试工具的升级是值得的，这种升级使得将来的工程测试变得较为容易。若新的需求并不是经常发生，测试人员可以选这不升级工具，而修改测试脚本，这时，可读的测试脚本更易于修改。

可维护性

一般来说，不同的软件团体采用不同的软件工程实践，因此，不同的产品，尽管可能具有一些共同的功能特性，但需求和设计也可能是不同的。CAR 自动测试工具所具有的功能，能够满足大部分构件的测试需求。若将来的工程期望有新的特性时，可以升级测试工具。

可移植性

测试人员希望开发出能公用的工具。仅仅通过简单的安装过程，构件测试工具就能够在不同的计算机系统上完成需要的测试任务。CAR 自动测试工具会在不同的计算机系统上完成需要的测试任务。因此，其他的测试人员（开发人员、IT 和产品培训人员）能够运行构件测试工具，检测错误和缺陷。

在测试工具开发过程中应用很多成功的软件工程规则。没有人能够写出完美的代码，而测试人员对查找他人编写的代码中的缺陷感兴趣。CAR 自动测试工具产生的测试脚本不需调试，这就节约了大量时间，不过这需要在保证测试工具经过全面的调试和测试的前提下。

5.3 脚本技术

开始时用一个脚本实现一个测试用例可以说是明智的，然而用这种方式实现自动测试的开销较大，并且维护成本也将不断增加。使用适当的脚本技术，可以打破“一个测试用例一个脚本”的规则，有利于创建一个较好的自动测试体系。

脚本技术可以分为这么几种：线性脚本，结构化脚本，共享脚本，数据驱动脚本，关键字驱动脚本。这些技术相辅相成，在支持脚本完成测试用例的时间和开销上都有长处和短处。

使用哪种脚本技术并不是最主要的，脚本所支持的实现测试用例体系的整体考虑才是主要的。

根据 CAR 构件的特点和上一章自动产生的测试用例文件，构件测试工具使用的脚本技术应该是数据驱动。

5.4 数据驱动脚本

数据驱动脚本技术将测试输入存储在独立的(数据)文件中，而不是存储在脚本中。执行测试时，从文件中而不是直接从脚本中读取测试输入。这种方法的最大好处是同一个脚本可以运行不同的测试。可以更容易增加新测试，因为只需要修改数据表。

使用这种技术，可以以较小的额外的开销实现许多测试用例。因为需要做的所有工作只是为每个增加的测试用例自定一个新的输入数据集合（及期望结果）。不需要编写更多的脚本。

数据驱动技术的主要优点是数据文件的格式对于测试者而言易于处理。除测试输入外，期望结果也从测试脚本中移出放到数据文件中。每个期望结果直接与待定的测试输入相关联。

5.5 测试脚本模板

测试脚本模块的数据流程图如下：

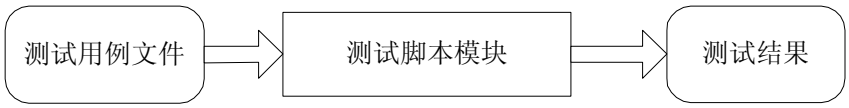

图 5-1 测试脚本运行流程

模块代码包括如下几步：

1. 使用 EzLoadModuleInfo API 得到 IModuleInfo 指针。
2. 通过 IModuleInfo 接口的 GetClassByName 找出相应的类。
3. 通过 IClassInfo 接口的 GetMethodByName 找出相应的方法
4. 通过 IClassInfo 接口的 CreateInstance 创建类的实例。
5. 通过 IMethodInfo 接口的 Invoke 方法调用相应的被测方法运行。
6. 读入相应被测方法的测试用例文件，一行一行读取测试用例对测试用例进行测试。如果返回值和输出参数值与测试用例的期望返回值和期望的输出参数值一样那么这个测试用例就通过，否则就这个测试用例就不通过并把测试用例和测试结果写到结果文件里。

第六章 结论

本文从自动测试的需求出发，针对构件可获取元数据的特点，提出了一种 CAR 构件自动测试的流程。该方法与传统的基于源代码的测试模式不同，直接以编译好的构件 dll 作为操作对象，经过元数据的提取、测试数据用例的自动产生、测试脚本的自动运行，输出测试结果，给构件软件的开发者和测试工作人员提供了很大的方便，在软件工程中的测试环节节约成本、提升效率等方面均有积极作用。

该法也易于推广到其他的构件平台之上，如 COM，CORBA，JAVA 等。

参考文献

[1]. (美) Kanglin Li & Mengqi Wu，曹文静、谈利群等译，高效软件测试自动，电子工业出版社，2004

[2]. 贺 平，软件测试技术，机械工业出版社，2005

[3]. (美) Daniel J.Mosley & Bruce A.Posey，邓波、黄丽娟、曹青春等译，软件测试制动化，机械出版社，2003

[4]. (美) Mark Fewster & Dorothy Graham，苏志勇等译，软件测试自动化技术与实例详解 Software Test Automatic，电子工业出版社，1999

[5]. (美) Robert V. Binder，面向对象系统的测试，人民邮电出版社，2001

[6]. (美)Brian Marich，韩柯等译， 软件子系统测试，机械出版社，2003

[7]. 古乐、史九林，软件测试技术概论，清华大学出版社，2004

[8]. 潘爱民，COM 原理与应用，清华大学出版社， 1999

[9]. DON BOX 著，潘爱民译，COM 本质论。中国电力出版社，2001。

[10]. Platt, David S.著，潘爱民译，深入理解 COM+，清华大学出版社，2000

[11]. (美)Dale Rogerson 著，杨秀章 译，《COM 技术内幕》，清华大学出版社，2001

[12]. 陈榕，因特网时代的操作系统演变，《计算机世界》，2001.10.29

[13]. 科泰世纪有限公司，Elastos 资料大全，2003

[14]. Jean Scholtz, ”Adaptation of Traditional Usability Testing Methods for Remote Testing”, National Institute of Standards and Technology.

致 谢

衷心感谢导师钟玉琢教授对本人的精心指导。他的言传身教将使我终生受益。

在上海科泰世纪科技有限公司的工作研究期间，承蒙陈榕教授热心指导与帮助，不胜感激。

感谢上海科泰世纪科技有限公司同事的热情帮助和支持！

声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名：_____日 期：_____

附录：Reflection.car 文件

定义了 Reflction 的各个接口、接口函数

```
//========================================================================
// Copyright (c) 2000-2005,  Elastos, Inc.  All Rights Reserved.
//========================================================================

module
{
    interface ITypeInfo;
    interface IFieldInfo;
    interface IParameterInfo;
    interface IModuleInfo;
    interface IClassInfo;
    interface IInterfaceInfo;
    interface IStructInfo;
    interface IEnumerationInfo;
    interface IAliasInfo;
    interface IMethodInfo;


    interface IModuleInfo
    {
        // Global operations
        GetName (
            [out] EzStrBuf ezName
        );

        Load(
            [in] EzStr esModuleName
        );

        GetEZMODID(
            [out] EZMODID *pezModid
        );
```

```
GetVersion(
    [out] EzStrBuf esVer
);

GetAttributes(
    [out] ModuleAttribute *pAttibute
);

// Class reflect information operations
GetClasses(
    [out] IObjectEnumerator **ppClassInfos,
    [out] UINT *pCount
);

GetClassByName(
    [in]  EzStr ezName,
    [out] IClassInfo **piClassInfo
);

GetObjectFactoryByName(
    [in] EzStr ezName,
    [in] DWORD dwDomain,
    [in] PDOMAININFO_TEMP pDomainInfo,
    [out] POBJECTFACTORY *ppObjectFactory
);

// Interface reflect information operations
GetInterfaces(
    [out] IObjectEnumerator **ppInterfaceInfos,
    [out] UINT *pCount
);

GetInterfaceByName(
    [in] EzStr ezName,
    [out] IInterfaceInfo **piInterfaceInfo
);
```

```
// Struct reflect information operations
GetStructs(
    [out] IObjectEnumerator **ppStructInfos,
    [out] UINT *pCount
);

GetStructByName(
    [in] EzStr ezName,
    [out] IStructInfo **piStructInfo
);

// Enumeration reflect information operations
GetEnumerations(
    [out] IObjectEnumerator **ppEnumerationInfos,
    [out] UINT *pCount
);

GetEnumerationByName(
    [in]  EzStr ezName,
    [out] IEnumerationInfo **piEnumerationInfo
);

GetEnumerationValueByName(
    [in]  EzStr ezName,
    [out] INT *pValue
);

// Type Alias reflect information operations
GetTypeAliases(
    [out] IObjectEnumerator **ppAliasInfos,
    [out] UINT *pCount
);

GetTypeAliasByName(
    [in] EzStr ezName,
    [out] IAliasInfo **piAliasInfo
```

```
    ) ;
}

interface IClassInfo
{
    GetName (
        [out] EzStrBuf ezName
    );

    GetEZCLSID(
        [out] EZCLSID *pezClsid
    );

    GetModule(
        [out] IModuleInfo **piModuleInfo
    );

    GetAttributes(
        [out] ClassAttribute *pAttibute
    );

    GetInterfaces(
        [out] IObjectEnumerator **ppInterfaceInfos,
        [out] UINT *pCount
    );

    GetInterfaceByName(
        [in] EzStr ezName,
        [out] IInterfaceInfo **piInterfaceInfo
    );

    GetInterfaceArttributeByName(
        [in] EzStr ezName,
        [out] ClassInterfaceAttribute **pAttribute
    );

    GetCallBackInterfaces(
```

```
        [out] IObjectEnumerator **ppInterfaceInfos,
        [out] UINT *pCount
    );

    GetCallBackInterfaceByName(
        [in] EzStr ezName,
        [out] IInterfaceInfo **piInterfaceInfo
    ) ;


    // Aggregate Class reflect information operations
    GetAggregates(
        [out] IObjectEnumerator **ppClassInfos,
        [out] UINT *pCount
    );

    GetParentClass(
        [out] IClassInfo **piClassInfo
    );

    //TODO: Add Aspects Info functions

    GetMethods(
        [out] IObjectEnumerator **ppMethodInfos,
        [out] UINT *pCount
    );

    GetMethodByName(
        [in] EzStr ezName,
        [out] IMethodInfo **piMethodInfo
    );

    CreateInstance(
        [in] PDOMAININFO_TEMP pDomain,
        [out] POBJECT *ppObject
    );
}
```

```
interface IInterfaceInfo
{
   GetName (
      [out] EzStrBuf ezName
   );

   GetIID(
      [out] IID *iid_
   );

   GetModule(
      [out] IModuleInfo **piModuleInfo
   );

   GetAttributes(
      [out] InterfaceAttribute *pAttibute
   );

   GetMethods(
      [out] IObjectEnumerator **ppMethodInfos,
      [out] UINT *pCount
   );

   GetMethodByName(
      [in] EzStr ezName,
      [out] IMethodInfo **piMethodInfo
   );
}

interface IStructInfo
{
   GetName (
      [out] EzStrBuf ezName
   );

   GetStructSize(
```

```
      [out] UINT *pSize
   );

   GetFields(
      [out] IObjectEnumerator **ppFieldInfos,
      [out] UINT *pCount
   );

   GetFieldByName(
      [in] EzStr ezName,
      [out] IFieldInfo **piFieldInfo
   );
}

interface IEnumerationInfo
{
   GetName (
      [out] EzStrBuf ezName
   );

   GetEnumerationValues(
      [out] EzArray<EnumerationValue> eaValues
   );

   GetValueByName(
      [in] EzStr ezName,
      [out] EnumerationValue *pValue
   );
}

interface IAliasInfo
{
   GetName (
      [out] EzStrBuf ezName
   );

   GetTypeInfo(
```

```
        [out] ITypeInfo **piTypeInfo
    );

    IsDummy(
        [out] BOOL *pIsDummy
    );
}

interface IMethodInfo
{
    GetName (
        [out] EzStrBuf ezName
    );

    GetParameters(
        [out] IObjectEnumerator **ppParamInfos,
        [out] UINT *pCount
    );


   Invoke(
        [in] IObject *pObj,
        [in] EzArray<UINT> ezaParame,
        [out] ERPT *erpt
    );

}

interface ITypeInfo
{
    GetName (
        [out] EzStrBuf ezName
    );

    IsAlias(
        [out] BOOL *pIsAlias
    );
```

```
    GetDataType(
        [out] DataType *pType,
        [out] INT *pTypeValue
    );

    GetTypeSize(
        UINT *pSize
    );
}

interface IFieldInfo
{
    GetName (
        [out] EzStrBuf ezName
    );

    GetTypeInfo(
        [out] ITypeInfo **piTypeInfo
    );
}

interface IParameterInfo
{
   GetName (
        [out] EzStrBuf ezName
    );

    GetTypeInfo(
        [out] ITypeInfo **piTypeInfo
    );

    GetAttributes(
        [out] ParameterAttribute *pAttibute
    );
}
```

```
    class CObjectEnumerator {
        interface IObjectEnumerator;
    }

    class CModuleInfo{
        interface IModuleInfo;
    };

    class CClassInfo{
        interface IClassInfo;
    };

    class CInterfaceInfo{
        interface IInterfaceInfo;
    };

    class CStructInfo{
        interface IStructInfo;
    };

    class CEnumerationInfo{
        interface IEnumerationInfo;
    };

    class CAliasInfo{
        interface IAliasInfo;
    };

    class CMethodInfo{
        interface IMethodInfo;
    };

    class CParameterInfo{
        interface IParameterInfo;
    };
}
```