# .NET 到 Mono 的移植

作者：张善友

# 目录

# 作者介绍

张善友 2001 年开始他的职业生涯，他一直是一个微软技术的开发者，连续荣获 10 年的 ASP.NET MVP，热衷于开源，在社区积极推广开源技术 Mono。
张善友拥有 SUSE Linux 企业服务器，CentOS 以及 tLinux（腾讯自行研制的 Linux 发行版）的专业经验，他主要是在 CentOS 上部署 Mono 平台，在业余时间喜欢教别人如何使用和利用 Linux 操作系统的力量，特别针对 Windows 开发人员收集编写了这本 CentoS dotNET 环境部署。希望对 Windows 上的.NET 开发人员顺利跨入 Linux 的 Mono 平台开发提供帮助。
业余时间运营微信公众号 dotNET 跨平台，微信号 opendotnet，欢迎关注。


这本书中资料来源于微信公众号 opendotnet 、http://www.cnblogs.com/shanyou/archive/2012/07/28/2612919.html 的文章汇集和 http://www.linuxdot.net 的资料整理，以及在 QQ 群: 102732979、103810355 里群友的讨论，在这里为国内积极贡献 dotNET 跨平台实践的同仁表示感谢。

# 第一章 项目移植

把一个运行在某个操作系统和硬件结构上的软件，在另一个操作系统和硬件结构上重新编译（包括一些必要的修改），以便在新的平台上运行，这一过程叫做应用程序移植。有些情况下，把应用程序从一个平台移植到另一个平台非常简单直接，仅需要重新编译并进行一些验证测试即可。但是有些情况下，移植程序并不是那么容易。

本章是在应用程序移植方面对当前项目管理的一个补充，关于如何使用正规化的需求管理过程、如何更好的与软件开发人员交流，以及如何进行项目管理，今天的项目经理们都已经非常熟悉了，但是，软件开发和软件移植毕竟并不完全相同，这也就是本章要讲述的内容。

本章重点讲述软件移植的详细过程和技术风险，并列出一些实现高质量应用程序一致的习惯和方法。

## 软件程序商业过程

在开始一个移植项目之前，很重要的一点就是要搞清楚在这个应用程序的生命周期中那些商业过程会受到影响。那些受到影响的商业过程必须进行修改，以适应移植后的应用程序，因为需要支持新的平台、新的测试环境、新的工具、新的文档，最重要的是，是需要支持新的客户并建立客户关系。

在应用程序的生命周期中，可能会影响到商业过程的三个主要领域是开发和测试、客户支持，以及应用程序发布：

- 开发和测试。在开发和测试部门，必须在以下方面对开发测试人员进行 Linux 技能测试：应用程序编程接口（API）的区别、开发工具、调试功能根据、性能工具，以及待移植的应用程序需要的第三方软件。
- 客户支持。在客户支持部门，必须在以下方面对支持人员进行培训：Linux 系统管理、移植后的应用程序需要的第三方软件、安装和管理方法、Linux 环境上的包管理工具、调试工具和方法，以及所需要的其他内容。
- 应用程序发布。在应用程序发布部门，必须在 Linux 整体特性和知识方面对销售人员和技术顾问进行培训。对软件发布渠道人员，必须对其进行培训，使其成为 Linux 软件程序的培训者。从客户的角度看，他们也希望获取 Linux 集成方面的知识，一并能够把 Linux 和他们已有的 IT 系统集成在一起。

移植应用程序到 Linux，也就意味着修改可能受到新移植的应用程序影响的商业组织过程。应该在真正的移植开始之前，认真考虑这三个主要的方面，并将他们包含在整个移植项目过程中。

## 移植过程

参与移植项目的开发人员在移植任何项目时都可以遵循类似的步骤。这些步骤包括：调查、分析、移植及测试。过程中的每一步都是后一步的基础。每一步都做好，后续的步骤也就会很容易完成。

# 调查

”调查“这一步主要是项目经理召集移植专家（在应用程序方面比较有经验，并且对开源平台、目标平台及应用程序使用的第三方产品比较了解的软件开发人员）和某一领域内的专家一起来确定待移植的应用程序所依赖的产品、开发和测试环境等。在调查阶段要明确的几个关键内容包括：产品/软件依赖关系、开发环境组件、编译环境组件和测试环境组件。

- **产品/软件依赖关系**。确定待移植的应用程序所依赖的产品，也就是要确定该应用程序使用那个版本的数据库、中间件以及第三方类库等。知道了所依赖的产品和版本，移植专家就可以估计在 Linux 平台上是否有这些产品和版本。
- **开发环境组件。**确定开发环境包括确定待移植的应用程序时用哪种编程语言编写的。使用较新的编程语言（例如 C#）编写的应用程序比较容易移植，但是使用 C/C++ 语言编写的应用程序需要花费较多的时间来分析和移植。
- **编译环境组件。**确定编译环境包括确定需要的编译工具是否在 Linux 上可用。对于在源平台上使用的平台相关的编译和链接标志必须进行调查，以确定在 Linux 上是否存在对应的标志。有些编译环境可能依赖于源平台，这可能要花费较多的功夫才能移植到 Linux 上。
- **测试环境组件。**确定移植后的应用程序所使用的测试环境，会引入一些测试人员应该关注的问题。通常情况下，移植工程师只对他们移植的部分做单元测试，然后就把程序交给测试组来做更完整的验证和系统测试。但是，谁是测试组呢？

多数情况下，“调查”这一步也会明确一些项目开始后可能遇到的风险。在调查阶段可以明确的风险如下：

- 需要的数据库、中间件和第三方类库版本在 Linux 上不可用。
- 应用程序包括一些需要转换成 Linux 汇编指令的汇编例程。
- 应用程序使用了源平台特有的 API 或编程模型。这也包括编写程序时对字母大小写和字节序的假设。
- 应用程序时按照.NET 的某个版本编写的，而该标准的实现又依赖于源平台上特有的编译器。
- 测试环境需要复杂的客户端/服务器架构。
- 开发环境需用第三方工具，而该工具需要移植到 Linux 上。
- 应用程序的发布或安装需要源平台的特有的工具。

“调查“这一步需要关注每一个新信息，这些信息可以通过问一些问题得到，例如关于文档、打包、性能调整等的问题。

# 分析

“分析“这一步需要从两个角度来考虑：项目管理和移植。从项目管理的角度看，分析就是要评估在前一个步骤中确定的各种移植问题和风险，以及它们会对项目移植产生怎么样的影响。”分析“这一步要制定项目计划，包括确定项目的范围和目标、创建工作进度计划、获取资源，以及分配项目角色。

确定项目的范围和目标也定义了项目经理和组员的职责范围和责任。项目范围指的是项目要完成的一系列工作。例如，像“应用程序 ABC 的模块 A 需要移植到平台 B 上，并在 B 上测试”，这样的简单陈述就是一个很好的定义项目范围的例子。

项目范围定义以后，移植工作的具体任务也就是可以定义了，这就产生了一个工作详细

分类的进度表。该进度表可以帮助确定哪些工作需要做，以及这些工作是顺序做还是可以并行来做。另外，该进度表还列出了所需要的资源。一个完整的进度表可以通过定义项目任务和所需的资源得到。

从移植的角度看，“分析”这一步就是移植工程师详细地分析应用程序的结构。移植工程师要确定应用程序所使用的 API 和系统调用，并且评估应用程序使用的动态链接和装载、网络、线程等。分析得到这些信息反馈给项目经理，项目经理据此确定更为详细的任务，并制定出更为准确的计划。

# 移植

“移植“这一步就是移植工程师开始执行分配给他们的具体工作。根据前一步得出的工作细目表，移植工程师可能只能串行的工作。这主要是因为待移植的程序可能是紧耦合的。也就是说，应用程序的一个模块高度依赖于其他模块，只有那些被依赖的模块移植完成后，这些模块才能开始移植。一个典型的例子就是编译环境的移植。如果原来的编译环境是设计成一次编译整个应用程序，那么必须在任何移植工作之前，把各模块所依赖的通用配置文件修改完毕。

如果移植工作相互之间没有关联，则移植可以并行进行。松耦合模块的移植可以分开来让不同的工程师同时进行。一个典型的例子就是共享库的移植，这样的共享库相互之间没有影响，可以各自独立编译，并且只是供其它模块编译链接使用。确定哪些工作可以并行进行时很重要的，并且这一工作应该是在分析阶段完成。

在 Linux 上编译代码的工作包括确定并消除代码对体系结构的依赖，以及非标准的编程习惯，包括检查代码并使用可医治的数据结构或编码标准。有经验的质量意识较强的移植工程师会纠正编译错误的同时来检查后者。

移植工作也包括移植编译环境到 Linux 平台。该工作应该在调查阶段明确下来。有些编译环境是可移植的，但有些并不是。确认编译环境不会导致潜在的问题是很容易被忽略的工作，需要非常仔细的调查和分析。

应用程序移植完成以后（也就是说，在 Linux 上编译完成了），移植工程师需要对移植的应用程序进行单元测试。单元测试可以很简单，例如可以简单的运行程序，看是否产生运行时错误，如果产生运行时错误，就需要在把应用程序交付给测试组以前修改完成这些错误。如何，测试组会对程序进行更完全的测试。

# 测试

在测试过程中，指定的测试人员会对移植后的应用程序运行一些测试用例，这些测试用例都有不同的测试目的，从仅仅运行应用程序的简单测试到测试应用程序在 Linux 平台上是否足够健壮的压力测试。在目标平台上对应用程序进行的压力测试，可以发现一些除体系结构依赖和坏的编码习惯之外的问题。大部分应用程序，尤其是多线程程序，在不同平台上进行压力测试时，往往会表现出不同的行为，部分原因是因为不同的操作系统实现，尤其是不同的线程的实现。如果在测试过程中发现问题，移植工程师就应该去调试和解决这些问题。

有些应用程序移植也包括移植一套测试工具来测试该应用程序。移植测试工具也是应该在调查和测试阶段确定的一个任务。多数情况下，测试人员往往需要接受一些对应用程序的培训，才能测试该应用程序。学习应用程序是一个与移植工作完全独立的任务，可以与移植任务并行进行。

测试发现问题后，需要解决这些问题并重新编译应用程序；然后重新测试该问题，直到应用程序通过所有的测试用例。

# 支持

移植工作完成后，开发阶段就算结束了，支持阶段也就随之开始。一些移植工程师会留下来帮助回答客户可能提出一些一直相关的问题。另外，开发人员也应该培训客户怎样在 Linux 平台上配置和运行应用程序。在移植结束后，支持阶段一般需要持续 60 到 90 天。在这期间，针对新移植到 Linux 的应用程序，移植工程师对技术支持人员和销售人员进行培训，并回答他们提出的问题。在培训完成以后，移植工程师的工作就算完成了。

# 定义项目范围和目标

定义项目范围就是定义清楚项目的结束点和边界，以及项目经理和项目成员的责任。定义清晰的项目范围可以确保该项目的所有相关者（参与项目或被项目影响的人员或组织，如项目组、架构师、测试人员、客户或赞助商、合作部门等）都明确该项目的目标。

清晰的项目目标或需求可以让人更好地理解项目范围。在移植过程的分析阶段，客户的目标和需求被收集上来，然后被细分成各个结构，并最终定义成项目的交付物。项目的目标和需求是定义项目范围的起点。所有的项目目标都确定以后，项目的范围也就变得更加明确了。

定义项目范围的一种方法是，列出项目要包含和不包含的目标。从客户收集需求列表是个很好的开始方法。需求收集上来后，项目经理和技术领导详细的检查该需求列表。如果对某些需求有疑问，应该把它们列到不被包含的目标列表中，至少最初时应该这样做。然后客户再次检查该列表，并对有异议的地方进行纠正。最后，知道所有人员都认为该列表已经正确表述了项目应该包含的所有目标。

需要再次强调的是，该列表需要足够详细，以便能够列出那些要包含在项目中，那些不要。的确，对超出项目范围的需求列出详细说明也是很重要的。对定义项目范围不能提供足够信息的目标，随着项目发布日期的临近，会逐渐成为争论的焦点。例如各个部分移植工作怎样交付给移植小组---是多次迭代还是一次性交付？每次交付时作为一个阶段，还是有一些更小的工作片包含在该阶段中？在一个完整的项目中有多少个迭代？该列表不仅定义了项目本身，也列出了该项目的所有相关干系人所期望的结果。

下面列出创建项目目标列表时需要遵循的一些基本原则：

1、 尽可能详细定义项目范围。

2、 确认项目的所有相关干系人都同意该项目范围。

3、 在“不包含/不在范围内”列表中列出未解决的内容，直到全部解决。

定义项目目标的一部分工作是列出项目的接收和完成标准。关于项目的完成标准，所有的项目干系人必须达成一致意见。项目完成可能是指在 Linux 平台上通过了百分之多少的系统测试，或者是通过了系统性能所设定的某个性能标准—也就是说，项目目标是运行一组具体的测试用例或者性能分析。不管项目完成的标准是怎样定义的，如果可能的话，所有的项目干系人在移植开始之前都必须理解并且同意这些标准。任何在移植过程中对标准的修改早替代现有标准前都必须与所有相关干系人交流协商并得到批准。

# 风险评估

很多移植项目超出预算或未能按时完成，主要是因为没有很好地管理移植过程中可能遇到的风险。风险是在估计进度和资源时要考虑的一个重要因素。在应用程序移植项目中，这些风险来自与移植相关的不同方面，主要包括下面几种：

- 移植工程师的技能水平和移植经验。
- 使用的编译器。
- 使用的编程语言。
- 第三方软件和中间件的可用性。
- 编译环境和工具。
- 平台依赖的结构。
- 平台和硬件依赖的代码。
- 需要搭建的测试环境。
- 用户界面需求。

依赖于要移植的具体应用程序，以上各个方面的每一个都可能给移植项目带来不同的复杂度和风险。评估这些复杂度和风险的级别，可以帮助你确定它们是不是可管理的。

# 技能水平和移植经验

应用程序移植和软件开发最明显的区别就是编程人员所掌握的技能。虽然软件开发人员往往是某一领域内比较专业的人员，但是要做软件移植的开发人员却需要更宽广的知识和技能。一个应用程序开发人员可以是 Windows 开发环境上的.NET 专家，或者是 Windows 操作系统上 SQL Server 的数据库专业开发人员；但是，从事代码移植工作的工程师却通常需要时两个或者更多操作系统、编程语言、编译器、调试器、数据库、中间件或最新的基于网页的技术方面的专家。他们还需要知道怎样安装和配置第三方数据库或中间件。

应用程序开发人员需要的往往是专才，而移植工程师则多是通才。应用程序开发人员可以为一个软件工作 18 个月，但是移植工程师在一个项目上往往只需要工作 3~6 个月，并且在上一个项目完成以后，即可投入下一个新的移植项目中。为一个移植项目找到完全适合的移植工程师有时候是很困难的，从老的技术移植到新的技术上时尤其如此。

# 编译器

在 Windows 平台上使用的编译器和编译器框架，与 Linux 平台上使用的有很大不同。如果在 Windows 平台和 Linux 平台上使用的是相同的编译器，移植工作将会变得容易一些，例如在两个平台上都是用的是 GNU 编译器，除了-g 和-c 标志外，不同的编译器使用的编译器标志可能大不相同，尤其是应用程序使用 C++语言的时候。

另外一个需要考虑的事情是编译器的版本。要移植的代码可能是几年前使用老版本的编译器编译的，这些代码可能使用了与新的编译器不同的语法，这就使得在不同的编译器上移植程序比较困难，因为这些编译器可能支持也可能不支持老的标准，即使新的编译器支持老的标准，不同编译器对这些标准的支持方式也可能不太相同。

Mono 项目的 C# 编译器的实现非常完整，和微软的编译器实现遵循同样的 ECMA 标准，大大降低了移植.NET 项目的难度。

## 第三方软件和中间件的可用性

当待移植的应用程序使用了第三方软件或中间件时，移植的复杂度会随之增加，尤其是在目标平台上没有这些第三方产品时，移植的难度会更大。可能会出现在 Linux 平台上没有可用的第三方产品。在过去的几年中，一些中间件厂商，例如 IBM，Oracle 等都把他们的中间件产品移植到了 Linux 上。他们花了一些功夫，使那些已经使用或者准备使用 Linux 作为企业平台的公司在 Linux 上可以使用它们的中间件产品。这也是为什么越来越多的公司愿意把应用程序移植到 Linux 上的部分原因。 但是对于 Mono 来说，很多.NET 平台上的厂商的产品他们并没有做 Mono 的兼容性测试，很有可能他们在 Mono 上无法运行。

## 编译环境和工具

编译环境越简单，理解如何编译源代码所需的时间也就越少。需要多遍编译的编译环境往往都很复杂。在这些多遍编译中，使用不同的编译器标志来编译目标文件，以便解决模块间的互相依赖。第一遍编译一些模块，第二遍可能会编译更多的模块，但是第二次编译使用了前面的编译的结果，以此类推，直到整个编译过程结束。有时候，根据一些非标准的配置文件，编译脚本会调用其他脚本来自动生成一些文件。这些文件的大部分可以和待移植的文件一样看待，但是事实上，这些脚本工具和配置文件才是要移植到目标平台上的内容、这种类型的工具往往会在评估和分析时漏掉了，进而可能会破坏已经制定的进度计划。

源代码控制是另一个必须考虑的。在 Linux 环境下，Subversion 和 Git 是应用最广泛的源代码控制工具，在一个移植项目中，如果有多个工程师同时来移植相同的模块，那么就需要进行源代码控制。

## 平台依赖的结构

当从基于 x86 的 Windows 平台移植到基于 RISC 结构的 Linux 平台上时，需要检查应用程序对字节序的依赖，例如，为了计算或数据操作而是用了字节交换的应用程序。在没有正确修改字节交换逻辑的情况下，调试问题将变得非常麻烦，因为很难找到数据在哪里出了问题，这主要是因为问题的根源通常是在发生问题的代码之前很远的地方。在调查和分析阶段，一定要找出平台依赖的结构。

## 平台/硬件依赖的代码

需要内核扩展和设备驱动程序支持的应用程序时最难移植的。所有平台的内核 API 都不遵循任何标准。因此，API 调用、参数个数，甚至是怎样把这些扩展装载到内核，在各个平台上都是不同的。多数情况下，都需要在 Linux 上开发新的代码。不过有一件事可以肯定，内核代码通常是使用 C 语言或者平台相关的汇编语言编写的。

## 搭建测试环境

移植工作完成以后，如果项目的范围还包括系统测试和验证，测试代码所需要的设备可

能也会增加移植的复杂度。如果应用程序需要在复杂的集群或网络环境中测试，设置环境和调配资源也会增加项目的时间。

测试也可能包括性能测试。通常，性能测试都需要运行最大配置，以便检测应用程序在满负载下的运行情况。

移植应用程序测试工具的工作也需要包含在移植进度计划中。需要对包括软件测试和专用硬件配置的测试工具进行评估，以确定其需要多长时间进行移植和配置。

# 用户界面需求

移植用户界面可能很简单，也可能很复杂。基于 Windows Forms 和 GTK#的用户界面比较容易移植，基于 WPF 技术的用户界面就难以移植了，Mono 不支持 WPF。

下面的表总结了需要考虑的各个方面，各个方面都根据其使用的技术列出了难、中、易。

| 内容 | 容易 | 中等复杂度 | 高复杂度 |
|---|---|---|---|
| 编译器/编程语言 | C#、F# | 使用的语言在 Mono 上支持有限 | 待移植代码的是 C++/CLI |
| 使用第三方产品或中间件 | None | Linux 上支持的和可用的 | Linux 上不支持的；<br>使用了 C++编写的第三方工具 |
| 编译环境和工具 | Msbuild 脚本 | Msbuild 脚本和编译脚本组合在一起 | |
| 平台/硬件依赖的代码 | 非平台/硬件依赖的代码 | 平台/硬件依赖的代码来自第三方产品，而且已经移植到了 Linux | 使用了内核扩展及设备驱动代码 |
| 测试环境及其搭建 | 独立的 | 客户端/服务器设置 | 网络、高可用行，集群；<br>需要外部设备来测试，如打印机、光纤连接通道磁盘等 |
| 用户界面 | GTK#, ASP.NET | WinForm | 不可移植的用户界面，例如 WPF |

经验表明，整个移植项目所需的实际工作量，在移植项目开始的前两三周，就可以评估出来。这段时间是待移植代码刚交付给移植项目组，而且移植工程师刚开始熟悉该软件程序，也可能是移植工程师刚开始在 Linux 环境下移植该软件程序。他们开始分解应用程序组件，并搭建最初的编译环境。在最初的两三周内，移植工程师会完成编译环境的配置，并开始编译一些模块。

过了最初的两三周后，最好重新检查原先评估的进度计划，并根据刚得到的实际经验决定是否对计划进行调整。

# 创建项目移植进度表

创建移植进度表时要考虑所有可能的风险，包括技术和商业相关的问题。技术方面，需要考虑资源和硬件是否可用、第三方的支持，以及 Linux 方面的经验等；商业方面，需要考

虑部门重组，位置调整（如改变办公地点），发布给客户的日期，以及商业目标的改变等，这些都会影响到整个移植进度。

上面这些技术和商业方面的问题，形成了移植项目对外部的依赖关系，而这些依赖又不是移植项目可以控制的。因此，建议仔细考虑每一个外部问题，以减少项目的风险。

创建移植项目的进度计划和做开发项目类似。

在移植过程中每次进入下一个步骤时，项目组都可以根据实际的进度和资源对项目重新估计。本质上讲，每一个阶段的结束，不仅仅是移植过程中的一个里程碑，也应该是重新检查先前估计得一个检查点。

# 从商业角度看移植过程

移植过程不仅仅是移植软件程序，移植后的应用程序最终还需要必要的商业支持，以成为一个完整的，成功的商业项目。在移植工作进行的同时，项目的相关干系人还需要准备对该应用程序提供支持的部门。客户支持和软件发布等部门还需要介绍应用程序在 Linux 上如何运行的支持和培训。

对客户支持和软件发布人员进行必要的 Linux 培训，应该在项目目标中列为高优先级。与任何新产品一样，用户和系统管理员每次都会对新的发布提出很多问题。因此，也需要回答 Linux 系统管理的问题。我们的经验表明，刚移植到新操作系统上的应用程序会产生很多关于系统管理以及如何使用新操作系统的问题，在支持热线接到的电话中，有五分之三的问题都是此类。

随着系统管理和安装问题的解决，关于应用程序的技术问题会逐步显现。支持部门需要访问测试机器来复现客户的问题。这些机器可能是开发或移植机器，依赖于应用程序和需要调试解决的实际问题。

从项目的整体来看，对支持的培训需求和提高应用程序支持所需资源的可用性，会在项目的后期出现。随着移植项目技术方面的工作趋于完成，商业内容也就逐渐显现。

# 技术调查问卷样例

该调查问卷可以作为移植技术的一个指南，并且据此还可以提出其他一些问题。该问卷中的客户指的是一个要移植到 Linux 的内部或外部部门。

# 平台相关的内容

1、 你当前的应用程序开发平台是什么？
   该问题是关于开发待移植应用程序的开发平台。这里不假定开发平台和应用程序部署的平台是相同的。这留在下一个问题中。
2、 该应用程序当前运行的平台是什么？
   移植工程师需要知道待移植的应用程序当前运行的平台。
3、 除了开发平台外，该应用程序是否还在其他平台上部署过？如果部署过，它运行的平台版本是什么?

问此问题可以让你知道应用程序的可移植性，看它是否移植到其他平台上。不过有一点需要注意：即使应用程序曾移植到其他平台上，它的目标平台可能也是比较老的版本。

4、 描述应用程序使用的硬件信息，以及需要的驱动程序在 Linux 平台上是否可用。

确定 Linux 能够满足应用程序对平台的依赖。

# 应用程序相关的内容

1、 请详细描述应用程序及其结构。

在这里，用户可以描述应用程序的结构，并尽可能地包括结构图。应用程序的所有组件都要描述。如果有的话，该问题也应该会让你知道应用程序运行的框架。大部分的.NET 应用程序运行在产品相关的框架上，例如 IIS 、windows 服务和 WCF 服务。也就是说，需要你处理 Mono 可能不支持的某个具体的框架。

2、 该应用程序有哪些不同组件？请给出各组件的名称和版本号。

该问题让你细分应用程序的结构，把应用程序细分成不同的组件。也就是说，可以把整个移植工作分成多个独立的任务。

3、 那些组件需要移植，那些不需要？请包含版本号。

客户需要告诉你那些需要移植，那些不需要。

4、 待移植的应用百分之多少是用下列编程语言编写?

1) Java
2) C#
3) F#
4) C
5) C++
6) 汇编语言
7) Visual Basic
8) IronPython/IronRuby
9) Powershell

通过询问应用程序使用了什么语言及其所占的比重，来确定应用程序的复杂度。

5、 粗略估计一下各语言所占的代码行数。

这是对问题 4 的另外一种问法，从不同的角度提出问题，，常常能找到互相矛盾的地方，这就需要公开讨论，从而能够把项目调查清楚。

6、 对于.NET 应用程序：使用了 P/Invoke 来链接特有的库了吗？请描述之。

明确待移植的应用程序的复杂度。多数情况下，非 100%纯.NET 编写的应用程序，都需要平台相关的例程，这些例程只能用固有的语言来处理，例如 C 语言。请注意这些平台相关的代码，往往它需要花费较多的时间来移植。

7、 应用程序用了内核模块了吗？如果有，请描述之。

明确待移植的应用程序的复杂度。如果应用程序使用的内核模块和例程是不可移植的，就需要花费较多时间转换成 Linux 上对应的例程。

8、 该应用程序是 2D/3D 的图形应用程序吗？请描述之。

明确待移植的应用程序的复杂度。确认 Linux 上存在兼容的图形工具和开发工具，无论是 Linux 默认提供的或者是第三方发行商提供的。

9、 应用程序使用了消息队列、共享内存、信号或者信号量吗？请描述之。

上述内容大部分能够方便的移植到 Linux 上。需要确认移植到 Linux 后，能够使原来期望的行为。

10、 应用程序或其中的组件，是多线程的吗？如果是，使用的是那种线程库?应用程序依赖开发平台特有的线程属性吗？

Linux 支持多种线程库，但是现在以及将来的 Linux 发行版中，符合标准的线程库是 NPTL（Native Posix Threads Library）实现。

11、 应用程序的某些操作提前假设某种特定的字节顺序吗？这在移植过程会成为问题吗？

该问题是关于应用程序的“大小端”（Little endian，Big endian）问题。大部分.NET 移植到 Mono 的目标平台都是 Intel 平台，该平台是小端的。也有可能要把.NET 程序移植到 RISC 类的大端平台。假设具体的大小端代码是不可移植的，并且如果移植不正确会导致不易察觉的错误。更糟糕的是，这些错误在移植时不会暴露出来，只会在系统测试的时候会突然出现问题，并且很难找到问题的根源。

12、 开发平台使用的是那种编译器版本？

1) C# (什么版本？)
2) VB.NET(什么版本？)
3) F#(什么版本？)
4) 平台特有编译器（Visual C++）
5) 其他（请列出）

明确待移植的应用程序的复杂度。如果待移植的应用程序用的是 C#/VB.NET 或者 F#编译器，则移植到 Linux 上会简单一些，因为 Mono 上对这几个编译器的支持很好。如果用了 windows 平台特有编译器编译的应用程序，C++应用程序比 C 程序较难移植，应用程序可能使用了 C++特性，例如模板。因为 C++标准在不同厂商的编译器上实现不同，移植这种类型的代码比简单的 C/C++代码要花费更多的时间。

13、 除了开发环境，应用程序还依赖其他的调试工具吗？例如内存泄漏调试器、性能分析工具、异常处理等。

这又回到了调查和分析依赖关系。Linux 上可能有，也可能没有所需的第三方工具，这需要调查。谁提供许可？谁负责获取这些工具？如果需要第三方支持，谁来提供支持？

14、 该应用程序是基于 Socket 的吗？如果是，它使用了 RPC 吗？请描述之。

虽然 Linux 支持标准的 socket 和 RPC 语义，但该问题的目的是确认程序的可移植性。问该问题可以搞清楚客户是否在应用程序里实现了不可移植的结构。该问题也可以引出其他问题，例如在测试阶段需要怎么样的配置。

15、 应用程序使用了第三方软件组件吗（数据库工具、应用程序服务器或其他中间件）？如果是，使用了哪些组件？

每一个第三方软件组件都会增加移植的复杂度。如果使用了任何第三方组件，都需要询问试了该组件的那个版本以及 Mono 上是否存在对应版本。第三方组件可能需要额外的时间去学习，甚至是配置或编译。

16、 应用程序时如何交付和安装的？使用标准的打包工具吗？安装脚本也需要移植吗？

Linux 上一个标准的打包工具是 RPM。RPM 会在其他章节讲述。明确客户是否需要移植应用程序打包部分的内容。

17、 应用程序或组件是 64 位的吗？有组件需要移植成 64 位的吗？

随着 64 为平台和操作系统的普及，该问题是要知道应用程序需要运行在什么体系结构的平台上。通过现代的编译器，大部分 32 为应用程序都可以轻松移植到 64 位环境上。需要考虑的一点就是移植和调试可能需要较长的时间。

# 数据库内容

1、 应用程序当前支持什么数据库？请包括版本号。

现在几乎所有的企业应用程序都需要后台数据库。确认应用程序所需的数据库在 Linux 上可用非常重要。数据库产品以及版本之间的差别会导致增加很多移植工作。

2、 移植后的应用程序期望运行在什么数据库上？

除了问题 1 外，客户希望移植后的应用程序运行在 Linux 平台的什么数据库上？

3、 应用程序使用了非关系数据库或私有数据库吗？

现在还有很多应用程序使用 NoSQL 数据库，幸运的是大部分 NoSQL 数据库都运行在 Linux 上。任何私有数据库都需要移植到 Linux 上。确认运行数据库的代码在 Linux 上可用，这也是调查阶段工作的一部分。

4、 应用程序是如何与数据库通信的？

1) 编程语言（例如 C#,VB.NET 等）
2) 数据库接口（例如 ODBC，ADO.NET）

确认所用的编程语言或接口在 Linux 上可用，确认是否由第三方厂商提供。

5、 应用程序需要使用扩展的数据类型吗（XML、audio，binary、video）？

这个问题主要用来评估移植小组需要具备的技能。

# 项目移植时间进度内容

1、 应用程序在目标平台上的正式可用日期是那天？

该问题是要明确在制定移植进度计划时，是否有商业目标需要考虑。

2、 应用程序在目标平台上的移植工作已经开始了吗？

这可以帮助评估在正式开始之前发现的复杂度问题。

3、 估计得移植复杂度是什么（低、中还是高）？

需要仔细分析对该问题的回答。现在很可能有一些新的因素在以前的移植经历中没有完全认识到。

4、 在确定复杂度级别时，考虑了什么因素？

以前移植工作的任何信息都需要评估，并且与将来的 Linux 平台移植工作进行对比。

5、 该应用程序曾移植到其他平台上吗？用了多长时间？需要多少资源？遇到过什么问题？

该问题试图把以前的移植工作和 Linux 移植进行比较。这只有在移植工程师的技术领导同时具有 Windows 平台和 Linux 平台移植经验的情况下，才会有用。

6、 你是怎样粗略估计项目移植时间和所需资源的？

应用程序或某些部分可能曾移植到其他平台上，知道向那些平台移植所花的时间会有些帮助。从那些移植过程得出的经验和教训会派上用场。吸取这些教训可以帮助你避免向 Linux 移植时重蹈覆辙。

# 测试相关的内容

1、 请描述接收测试的环境配置。

2、 单元测试需要什么样的网络和数据库配置？

3、 移植测试需要多少时间和资源？
4、 是否已经建立了测试脚本和性能度量标准？
5、 需要运行一些基准测试来进行比较吗？
6、 性能数据在当前平台上可用吗？
7、 最后执行性能测试是什么时间？

所有测试相关的问题都适用于软件程序在 Linux 平台上的测试。问这些问题还可以引出其他一些与移植测试脚本和测试工具有关的问题，而这些可能会增加整个项目的风险和时间。要重点关注客户对问题 1 的回答。问题 1 和接收标准有关，这些标准需要各方在移植开始之前都同意。一个接收标准的例子是：模块 A 和 B 应该通过测试用例 c 和 d，并且没有错误。当达到测试标准后，移植可以说是完成了，正式的 QA 测试接着就可以开始了。

# 项目移植的执行内容

1、 根据你希望的情况，请选择下面的一项或多项：
   1) 必要的话，将会给移植工程师提供一些技术帮助。
   2) 客户负责获取第三方工具许可和技术支持。
   3) 其他（请描述）。

可以在这里增加你认为需要客户考虑的其他事项。有些问题可能是关于员工培训或测试应用程序等。

2、 项目需要什么硬件？

该问题是要确认是否会用到现有的硬件或额外的硬件，以用于移植、测试、培训，以及必要的支持等。

尽管上述问卷已经比较全面，但是它不应该是调查所依赖的唯一基础。调查还应该包括对应用程序源代码的实际检查。软件程序的文档也需要检查，以便用户能够从中了解到需要的应用程序信息。

# 小结

在开始实际的移植之前，项目经理和移植技术专家需要考虑移植项目的多方面的内容。仔细调查这些内容使得我们过去成功移植了许多项目。因为每个移植项目都互不相同，这些内容也可能都具有不同的形式。尽管如此，底层的概念和描述仍然是相同。一个移植项目可以概括为下面的内容：

- 调查： 提出问题，检查可用的代码，提出更多的问题并调查风险，制定进度计划。
- 分析：提出更多的问题，调查风险，基于技术和商业问题创建进度计划。
- 移植：搭建编译环境、编译及单元测试。
- 测试：搭建测试环境、测试及性能分析。如果发现了问题，重复移植阶段的部分工作。
- 培训：给技术支持和销售人员培训应用程序和操作系统环境等内容。

因为移植项目各方面的复杂度和风险都会影响到项目的整体进度和预算，在每个过度点（上一个阶段结束和下一个阶段的开始），项目组应该重新评估以前的估计。重新评估以前的估计可以帮助项目的所有相关干系人确定正确的期望。和所有项目一样，项目的成功不仅仅定义为成功交付了最终产品，还定义为是否满足了各方的期望。在以后的章节中，我们会介绍移植过程的每个阶段（），并在各阶段介绍一些移植工程师需要掌握

的 Linux 技术细节。

# 第二章 调 查

在整个项目的移植过程中，“调查”这一步主要是对移植的项目做个初步的评估。首先需要从以下几个方面来评估应用程序：使用的编程语言、编译应用程序的环境和对第三方软件的依赖关系等。上面的每一项都必须仔细检查，以确认应用程序能够移植到 Linux 上。如果以上各项都没有问题，移植工程师就可以继续评估应用程序的其他部分，并根据收集到的信息来确定移植项目的范围、交付物及项目任务。本章讲述了 Mono、编译环境工具、多种 Shell 及打包工具，这些内容可以有效地帮助项目移植人员顺利完成“调查”这一过程。

## Mono 环境

第一章末尾列出的评估问题之一，待移植的应用程序是用什么语言编写的。这个问题就是要确定待移植应用程序是否能用 Mono 的编译器来编译。

Mono 是一个在非 Windows 操作系统中提供 C#编译器和 CLR 的开源项目。目前，Mono 授权于 GPL 版本 2、LGPL 版本 2、MIT 以及双许可证。可在 Mac、Linux、BSD 以及其他操作系统中运行 Mono。通过 C#编译器，还可在 Mono 中运行其他语言，其中包括 F#、Java、Scala 和 Basic 等。

Mono 的创始人是 Miguel de Icaza，微软在 2001 年把 CLI 和 C# 提交给了 ECMA[ECMA 是一个致力于推动行业范围内采用信息和通信技术的非特定供应商的国际标准组织]标准化 ECMA 335 和 ECMA 334)，比 Java 还早的标准化了.NET 平台。Miguel de Icaza 看到了 C#语言的优雅和高效率，Ximian 内部对如何创建能有效提升生产效率的工具进行了大量的讨论，他们的目标是通过这些创建出来的工具让用户可以在更短时间内创建出更多的应用程序从而缩短开发周期和降低开发成本。de Icaza 所在的 Ximian 公司在 2001 年 7 月开始启动一个名叫 Mono Project 的开放源码版本".NET"的开发项目，旨在使开发者能够编写同时在 Windows 和 Linux 上运行的.NET 程序。并在 2004 年发布了第一个版本，Mono 目前的最新版本是 3.2，同时 Mono 还在不断地持续更新。Mono 一直是由 de Icaza 直接领导，2012 年 Novell 公司被收购，Mono 项目的管理已经移交给 de Icaza 所创立的一家新公司 Xamarin，由其指引 Mono 的发展方向。现在 Xamarin 的职责是发展 Mono，同时负责开发 Xamarin.IOS 和 Xamarin.Android 以及让开发人员使用这些产品所需的软件。Xamarin 所领导的 Mono 现在已经覆盖到服务器，桌面和移动领域，我认为这些产品将是非常优秀的。

虽然期望 Mono 的功能可以尽可能多与.NET Framework 的功能相匹配，但这是不可能的，因为微软拥有更多的资源，并且在这些功能的开发商具有先起步的优势，随着微软的开源政策的挑整，加强互操作性，微软已经将整个 ASP.NET 相关部分都开源了，开源的部分包括 ASP.NET MVC、ASP.NET 网页，Entity Framework，Razor 模板引擎，Actor FX，RX，TX 等等，（http://www.infoq.com/cn/articles/interoperability-is-the-key） 。Mono 项目拥有了许多与.NET Framework 功能相同的功能。

# 编译环境

我们继续从编译环境方面来评估待移植的应用程序。移植一个应用程序时，检查该应用程序原来用什么工具编译是很重要的。你会发现.NET 平台上的使用的编译工具编译的应用程序可以直接在 Mono 平台上上运行，同时 Mono 也有类似的工具。检查编译环境时，注意原有的编译过程也是很重要的。一些编译过程很简明直接，但是有些却比较复杂。复杂的编译过程会增加移植的难度，因此在制定移植计划时要考虑到该因素。

## XBuild

XBuild 是开源的 MSBuild 实现，MSBuild 全称（Microsoft Build Engine）,是用来生成.NET 程序的平台。MSBuild 不仅仅是一个构造工具，应该称之为拥有相当强大扩展能力的自动化平台。MSBuild 平台的主要涉及到三部分：执行引擎、构造工程、任务。其中最核心的就是执行引擎，它包括定义构造工程的规范，解释构造工程，执行“构造动作”；构造工程是用来描述构造任务的，大多数情况下我们使用 MSBuild 就是遵循规范，编写一个构造工程；MSBuild 引擎执行的每一个“构造动作”就是通过任务实现的，任务就是 MSBuild 的扩展机制，通过编写新的任务就能够不断扩充 MSBuild 的执行能力。

MSBuild 有四个基本块（属性、项、任务、目标）：

- MSBuild 属性: 属性是一些键/值对，主要用来存储一些配置信息。
- MSBuild 项: 主要是存储一些项目文件信息，以及文件的元数据信息（如版本号）。
- MSBuild 任务: Build 过程中的一些原子操作（如 CSC、MakeDir）
- MSBuild 目标: 按特定的顺序将任务组织在一起，并允许在命令行单独指定各个部分

http://manpages.ubuntu.com/manpages/precise/man1/xbuild.1.html
MSBuild 简解
MSBuild 入门
MSBuild 的深入认识

# 集成开发环境

.NET 的应用程序通常都是用集成开发环境 Visual Studio 开发编译的，Mono 也有一个开源的集成开发环境 MonoDevelop 可供使用。 MonoDevelop 许开发者非常简单的将 Visual Studio 开发的.NET 应用程序移植到 Linux 和 Mac OS X 下。该软件可以从 http://monodevelop.com/ 网站下载。

.NET 应用程序和 Mono 具有二进制级别的兼容性，可以直接使用 Visual Studio 开发编译，通过 XCopy 方式部署到 Mono 上。

## Linux Shell

在编译应用程序时，编译环境也可能使用 Shell 脚本来启动编译任务或过程。Linux 也提供了 Shell。根据安装 Linux 时的选择，选中的 Shell 会被安装在 Linux 上。Linux 上常见的 Shell 有:

- Bash：一种与 sh 兼容的、包含了一些 Korn shell (Ksh)和 C shell(csh)特性的 shell 或命令语言解释器。该 shell 遵循 IEEE POSIX P1003.2/ISO 9945.2 Shell 和 Tools 标准。大部分 sh 脚本不做任何改动即可咋 bash 中运行。Bash 几乎是所有 Linux 发行版中默认的 shell。
- Tcsh：一种完全与 Berkeley Unix C shell(csh)兼容的 shell 或命令语言解释器。可被用作交互式登录 shell，以及 shell 脚本命令的处理器。
- Zsh：一种保含了 bash、ksh、tcsh 许多特性以及自身特性的 shell 或命令语言解释器。
- linux 下 mono，powershell 安装教程

# 打包

有些情况下，可能需要一个包管理器来统一管理 Linux 平台上应用程序的发布和安装。在 Linux 系统下，对于软件包的管理有多种机制，有源代码方式、RPM 软件包管理方式以及 YUM 软件管理方式。

Red Hat 包管理器（Red Hat Package Manager，简写 RPM）是 Linux 上最常用的包管理器。RPM 是一个基于命令行的管理工具，可以用来安装、卸载、验证、查询和更新软件包。在 RPM 开发出来之前，Linux 上没有很方便的方法来管理软件程序的安装。那时软件程序是通过压缩的 tar 文件来安装的，而且唯一能够查到一个应用程序是否依赖于其他包的方法是阅读随程序一起发布的文档或者通过运行程序来得到错误提示信息。RPM 的出现改变了这一切。

RPM 格式的应用程序使得软件程序的维护和管理变得容易了很多。Red Hat 也因此变成一个最流行的 Linux 发行版。今天，绝大多数的 Linux 发行版都把 RPM 当作默认的包管理器。也正因为此，我们建议在把应用程序移植到 Linux 上时，打包成 RPM 格式。

# 用 RPM 打包应用程序

RPM 的核心是 RPM 引擎。该引擎需要一些输入文件，并输出打包后的应用程序。其中一个输入文件是 spec 文件，该文件包含一些提示性和指令性的信息，在打包过程中或用户安装后的应用程序时，分别会用到这些信息。该 spec 文件包括以下各节（section）。

- Preamble：包含了一些可供显示的关于该包的信息。
- Prep：包含 RPM 开始实际打包前用到的脚本。
- Build：包含打包过程用到的脚本。
- Install：包含安装打包后的应用程序时用到的命令
- Install 和 Uninstall：包含安装和卸载脚本，用户用 RPM 工具安装或卸载软件程序时，这些脚本会被分别执行。
- Verify：包含一个在用户系统上验证软件包安装情况的脚本。

- Clean：包含一个在软件打包后负责清理工作的脚本。
- File List：包含要打包的文件列表、安装到用户系统后的文件权限、文档和配置信息。

打包过程中用到的 spec 文件的节是 prep、build、install、clean 和 file list；向用户系统上安装打包后的软件包时，用到的节是 install 和 uninstall、verify 和 file list。

RPM 打包的过程如下：

1. 创建一个包含源文件的编译环境。在该环境中给源文件打补丁（如果需要的话），并把源文件编译成二进制程序。
2. 运行 install 节的安装脚本，把编译好的二进制程序安装到正确的目录位置。
3. 二进制程序安装完成后，用 file list 节中的内容创建 RPM 包。需要注意的是 RPM 不会自己创建文件列表，需要手工创建文件列表。
4. 打包完成后的输出就是一个 RPM 格式的软件包。

# RPM 的基本用法

把软件包打包成 RPM 包以后，下面我们来讲述 RPM 的基本命令。每一个 rpm 包的名称都由 - 和 . 分成了若干部分。就拿“convmv-1.15-1.el6.rf.noarch.rpm”这个包来解释一下，“convmv” 为包名，“1.15” 则为版本信息，“1.el6.rf” 为发布版本号，“noarch” 为运行平台。其中运行平台常见的有 i386, i586, i686, x86_64 ，需要您注意的是 cpu 目前是分 32 位和 64 位的，i386,i586 和 i686 都为 32 位平台，x86_64 则代表为 64 位的平台，noarch，这代表这个 rpm 包没有硬件平台限制。下面介绍一下 rpm 常用的命令。

- **安装一个 rpm 包**

```
[azureuser@mono src]$ sudo rpm -ivh convmv-1.15-1.el6.rf.noarch.rpm
warning: convmv-1.15-1.el6.rf.noarch.rpm: Header V3 DSA/SHA1 Signature, key ID 6b8d79e6: NOKEY
Preparing...                ########################################### [100%]
   1:convmv                 ########################################### [100%]
```

“-i” ：安装的意思

“-v” ：可视化

“-h” ：显示安装进度

另外在安装一个 rpm 包时常用的附带参数有：

--force：强制安装，即使覆盖属于其他包的文件也要安装

--nodeps：当要安装的 rpm 包依赖其他包时，即使其他包没有安装，也要安装这个包

- **升级一个 rpm 包**

  命令 rpm -Uvh filename

  “-U” ：即升级的意思

- **卸载一个 rpm 包**

命令 rpm -e filename

这里的 filename 是通过 rpm 的查询功能所查询到的，稍后会作介绍。

```
[azureuser@mono src]$ rpm -qa | grep convmv
convmv-1.15-1.el6.rf.noarch
[azureuser@mono src]$ sudo rpm -e convmv-1.15-1.el6.rf.noarch
```

卸载时后边跟的 filename 和安装时的是有区别的，安装时是把一个存在的文件作为参数，而卸载时只需要包名即可。

- **查询一个包是否安装**

命令 rpm -q rpm 包名 (这里的包名，是不带有平台信息以及后缀名的)

```
[azureuser@mono src]$ sudo rpm -q    convmv-1.15-1.el6.rf.noarch
package convmv-1.15-1.el6.rf.noarch is not installed
[azureuser@mono src]$ sudo rpm -ivh convmv-1.15-1.el6.rf.noarch.rpm
warning: convmv-1.15-1.el6.rf.noarch.rpm: Header V3 DSA/SHA1 Signature, key ID 6b8d79e6: NOKEY
Preparing...                ########################################### [100%]
   1:convmv                 ########################################### [100%]
[azureuser@mono src]$ rpm -q convmv-1.15-1.el6.rf.noarch
convmv-1.15-1.el6.rf.noarch
```

我们可以使用 rpm -qa 查询当前系统所有安装过和未安装的 rpm 包，限于篇幅，只列出前十个。

```
[azureuser@mono src]$ rpm -qa | head
gdb-7.2-56.el6.x86_64
at-3.1.10-43.el6_2.1.x86_64
ca-certificates-2010.63-3.el6_1.5.noarch
cvs-1.11.23-11.el6_2.1.x86_64
basesystem-10.0-4.el6.noarch
alsa-utils-1.0.22-3.el6.x86_64
logrotate-3.7.8-15.el6.x86_64
fontpackages-filesystem-1.41-1.1.el6.noarch
tcsh-6.17-19.el6_2.x86_64
ncurses-base-5.7-3.20090208.el6.x86_64
[azureuser@mono src]$
```

- **得到一个已安装 rpm 包的相关信息**

命令 rpm -qi 包名 （同样不需要加平台信息与后缀名）

```
[azureuser@mono src]$ rpm -qi convmv-1.15-1.el6.rf.noarch
Name        : convmv                       Relocations: (not relocatable)
Version     : 1.15                         Vendor: Dag Apt Repository, http://dag.wieers.com/apt/
Release     : 1.el6.rf                     Build Date: Mon 05 Sep 2011 04:37:50 PM UTC
Install Date: Sat 10 Aug 2013 04:11:13 AM UTC      Build Host: lisse.hasselt.wieers.com
Group       : Applications/File            Source RPM: convmv-1.15-1.el6.rf.src.rpm
Size        : 42999                        License: GPL
Signature   : DSA/SHA1, Mon 05 Sep 2011 04:52:11 PM UTC, Key ID a20e52146b8d79e6
Packager    : Dries Verachtert <dries@ulyssis.org>
URL         : https://www.j3e.de/linux/convmv/
Summary     : Convert filenames to a different encoding
Description :
convmv converts filenames (not file content), directories, and even whole
filesystems to a different encoding. This comes in very handy if, for example,
one switches from an 8-bit locale to an UTF-8 locale or changes charsets on
Samba servers. It has some smart features: it automagically recognises if a
```

file is already UTF-8 encoded (thus partly converted filesystems can be fully moved to UTF-8) and it also takes care of symlinks. Additionally, it is able to convert from normalization form C (UTF-8 NFC) to NFD and vice-versa. This is important for interoperability with Mac OS X, for example, which uses NFD, while Linux and most other Unixes use NFC. Though it's primary written to convert from/to UTF-8 it can also be used with almost any other charset encoding. Convmv can also be used for case conversion from upper to lower case and vice versa with virtually any charset. Note that this is a command line tool which requires at least Perl version 5.8.0.

- **列出一个 rpm 包安装的文件**

命令 rpm -ql 包名

[azureuser@mono src]$ rpm -ql convmv-1.15-1.el6.rf.noarch
/usr/bin/convmv
/usr/share/doc/convmv-1.15
/usr/share/doc/convmv-1.15/CREDITS
/usr/share/doc/convmv-1.15/Changes
/usr/share/doc/convmv-1.15/TODO
/usr/share/man/man1/convmv.1.gz

通过上面的命令可以看出文件“/usr/bin/convmv”是通过安装“convmv-1.15-1.el6.rf.noarch”这个 rpm 包得来的。那么反过来如何通过一个文件去查找是由安装哪个 rpm 包得来的？

- **列出某一个文件属于哪个 rpm 包**

命令 rpm -qf 文件的绝对路径

[azureuser@mono src]$ rpm -qf /usr/bin/convmv
convmv-1.15-1.el6.rf.noarch

- **RPM 软件验证命令**

rpm -K software.rpm 验证 rpm 文件
rpm -V softname 验证已安装的软件

RPM 软件包管理还具有验证功能，因为在开源的软件里，源代码都是开放的，我们从网上下载的软件可以被一些不法分子在里面植入了一些木马程序，这样就会损害我们的操作系统。所以为了安全起见现代操作系统都加入了对软件的验证功能。

验证 rpm 文件我们可以使用 rpm -K software.rpm 命令，例如我们要验证刚才的 convmv-1.15-1.el6.rf.noarch.rpm：

[azureuser@mono src]$ sudo rpm -K convmv-1.15-1.el6.rf.noarch.rpm
[sudo] password for azureuser:
convmv-1.15-1.el6.rf.noarch.rpm: (SHA1) DSA sha1 md5 (GPG) NOT OK (MISSING KEYS: GPG#6b8d79e6)

验证以后发现该软件是没有问题的。

验证已安装的软件我们可以使用 rpm -V softname 命令，例如我要验证一下安装的 convmv 软件，就可以使用如下命令：

[azureuser@mono src]$ rpm -V convmv

如果没有出现任何错误，就表示该软件是完整的，没有被修改。我们使用的 RHEL 以及 CentOS 等 Linux 系统，其软件包的安装维护都是通过 RPM 软件包来进行管理的，我们也看到使用 RPM 软件包来对软件进行管理非常的方便。

# yum 工具

很快一个新的问题难倒了 GNU/Linux 制作者，他们需要一个快速、实用、高效的方法来安装软件包，当软件包更新时，这个工具应该能自动管理关联文件和维护已有配置文件，yum（全称为 Yellow dog Updater, Modified）是一个在 Fedora 和 RedHat 以及 SUSE 中的 Shell 前端软件包管理器。基於 RPM 包管理，能够从指定的服务器自动下载 RPM 包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包，无须繁琐地一次次下载、安装。yum 提供了查找、安装、删除某一个、一组甚至 全部软件包的命令，而且命令简洁而又好记。下面介绍常用的 yum 命令：

- 列出所有可用的 rpm 包 “yum list”

```
[azureuser@mono src]$ yum list | head -n 10
Loaded plugins: security
Installed Packages
ConsoleKit.x86_64                                                0.4.1-3.el6
@anaconda-CentOS-201207061011.x86_64/6.3
    ConsoleKit-libs.x86_64                                       0.4.1-3.el6
@anaconda-CentOS-201207061011.x86_64/6.3
    MAKEDEV.x86_64                                               3.24-6.el6
@anaconda-CentOS-201207061011.x86_64/6.3
    SDL.x86_64                                                   1.2.14-3.el6
@anaconda-CentOS-201207061011.x86_64/6.3
    WALinuxAgent.noarch                                          1.3.3-1
@openlogic
    abrt.x86_64                                                  2.0.8-6.el6.centos.2
@updates
    abrt-addon-ccpp.x86_64                  2.0.8-6.el6.centos.2          @updates
    abrt-addon-kerneloops.x86_64            2.0.8-6.el6.centos.2          @updates
```

限于篇幅，只列举出来前 8 个包信息。从上面的例子中您还可以看到最左侧是 rpm 包名字，中间是版本信息，最右侧是安装信息，如果安装了就显示类似 “@anaconda-CentOS”，“@openlogic” 或者 “@updates”，他们前面都会有一个 “@” 符号，这很好区分。未安装则显示 base 或者 extras，如果是该 rpm 包已安装但需要升级则显示 updates. 如果您看的仔细会发现，”yum list” 会先列出已经安装的包(Installed Packages) 然后再列出可以安装的包(Available Packages)

- 搜索一个 rpm 包

命令 yum search [相关关键词]

```
[azureuser@mono src]$ yum search vim
Loaded plugins: security
============================== N/S Matched: vim ======================
vim-X11.x86_64 : The VIM version of the vi editor for the X Window Syste
vim-common.x86_64 : The common files needed by any version of the VIM ed
vim-enhanced.x86_64 : A version of the VIM editor which includes recent
                    : enhancements
```

vim-minimal.x86_64 : A minimal version of the VIM editor

  Name and summary matches only, use "search all" for everything.

除了这样搜索外，常用的是利用 grep 来过滤

```
[azureuser@mono src]$ yum search vim
Loaded plugins: security
============================== N/S Matched: vim ======================
vim-X11.x86_64 : The VIM version of the vi editor for the X Window Syste
vim-common.x86_64 : The common files needed by any version of the VIM ed
vim-enhanced.x86_64 : A version of the VIM editor which includes recent
                    : enhancements
vim-minimal.x86_64 : A minimal version of the VIM editor

  Name and summary matches only, use "search all" for everything.
[azureuser@mono src]$ yum list | grep vim
vim-common.x86_64                                   2:7.2.411-1.8.el6
@anaconda-CentOS-201207061011.x86_64/6.3
vim-enhanced.x86_64                                 2:7.2.411-1.8.el6
@anaconda-CentOS-201207061011.x86_64/6.3
vim-minimal.x86_64                                  2:7.2.411-1.8.el6
@anaconda-CentOS-201207061011.x86_64/6.3
vim-X11.x86_64                          2:7.2.411-1.8.el6           base
```

我们同样可以找到相应的 rpm 包。

- 安装一个 rpm 包

命令 yum install [-y] [rpm 包名]

如果不加“-y”选项，则会以与用户交互的方式安装，首先是列出需要安装的 rpm 包信息，然后会问用户是否需要安装，输入 y 则安装，输入 n 则不安装。如果嫌这样太麻烦，所以直接加上 “-y” 选项，这样就省略掉了问用户是否安装的那一步。

```
[azureuser@mono src]$ sudo yum install vim-X11
[sudo] password for azureuser:
Loaded plugins: security
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package vim-X11.x86_64 2:7.2.411-1.8.el6 will be installed
--> Finished Dependency Resolution
Dependencies Resolved
================================================================================
 Package        Arch          Version                        Repository   Size
================================================================================
Installing:
 vim-X11        x86_64        2:7.2.411-1.8.el6              base         1.0 M

Transaction Summary
```

```
================================================================================
Install       1 Package(s)

Total download size: 1.0 M
Installed size: 2.2 M
Is this ok [y/N]: y
Downloading Packages:
vim-X11-7.2.411-1.8.el6.x86_64.rpm                       | 1.0 MB     00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
Warning: RPMDB altered outside of yum.
  Installing : 2:vim-X11-7.2.411-1.8.el6.x86_64                             1/1
  Verifying  : 2:vim-X11-7.2.411-1.8.el6.x86_64                             1/1
Installed:
  vim-X11.x86_64 2:7.2.411-1.8.el6
Complete!
```

- 卸载一个 rpm 包

命令 yum remove [-y] [rpm 包名]

```
[azureuser@mono src]$ sudo yum remove -y vim-X11
Loaded plugins: security
Setting up Remove Process
Resolving Dependencies
--> Running transaction check
---> Package vim-X11.x86_64 2:7.2.411-1.8.el6 will be erased
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package        Arch          Version                       Repository    Size
================================================================================
Removing:
 vim-X11        x86_64        2:7.2.411-1.8.el6             @base         2.2 M

Transaction Summary
================================================================================
Remove        1 Package(s)

Installed size: 2.2 M
Downloading Packages:
Running rpm_check_debug
Running Transaction Test
```

Transaction Test Succeeded
Running Transaction
  Erasing      : 2:vim-X11-7.2.411-1.8.el6.x86_64                    1/1
  Verifying    : 2:vim-X11-7.2.411-1.8.el6.x86_64                    1/1
Removed:
  vim-X11.x86_64 2:7.2.411-1.8.el6
Complete!
卸载和安装一样，也可以直接加上 “-y” 选项来省略掉和用户交互的步骤。在这里要提醒一下，卸载某个 rpm 包一定要看清楚了，不要连其他重要的 rpm 包一起卸载了，以免影响正常的业务。

- 升级一个 rpm 包

命令 yum update [-y] [rpm 包]

以上介绍了如何使用 yum 搜索、安装、卸载以及升级一个 rpm 包，如果您掌握了这些那么您就已经可以解决日常工作中遇到的与 rpm 包相关问题了。当然 yum 工具还有好多其他好用的命令，这里不再列举出来，如果您感兴趣就去 man 一下吧。除此之外，下面还会教您一些关于 yum 的小应用。

- 自动选择最快的源

由于 yum 中有的 mirror 速度是非常慢的，如果 yum 选择了这个 mirror，这个时候 yum 就会非常慢，对此，可以下载 fastestmirror 插件，它会自动选择最快的 mirror：
yum install yum-fastestmirror
配置文件：/etc/yum/pluginconf.d/fastestmirror.conf（一般不用改动），你的 yum 镜像的速度测试记录文件：/var/cache/yum/timedhosts.txt

- 配置 yum 源

yum 源的配置文件：/etc/yum.repos.d/CentOS-Base.repo，例如下面我们把源配置为网易的：

首先备份/etc/yum.repos.d/CentOS-Base.repo
mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup 下载 Centos 6 repo 文件 http://mirrors.163.com/.help/CentOS6-Base-163.repo, 放入/etc/yum.repos.d/(操作前请做好相应备份)，然后替换 CentOS-Base.repo, 运行 yum makecache 生成缓存。

- 添加特定软件的 Yum 源，例如在 yum 仓库中找不到 MonogoDB，MonogoDB 提供了 yum 源安装方法。可以通过下面的方式操作。

在/etc/yum.repos.d/目录中增加 *.repo yum 源配置文件，例如 10gen.repo
vi /etc/yum.repos.d/10gen.repo，输入下面的语句：

[10gen]
name=10gen Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64
gpgcheck=0
做好 yum 源的配置后，如果配置正确执行下面的命令便可以查询 MongoDB 相关的信息：
查看 mongoDB 的服务器包的信息
[azureuser@mono src]$ yum info mongo-10gen-server
Loaded plugins: security
10gen                                                    |  951 B     00:00

10gen/primary | 11 kB 00:00
10gen 9
Available Packages
Name : mongo-10gen-server
Arch : x86_64
Version : 2.4.5
Release : mongodb_1
Size : 12 M
Repo : 10gen
Summary : mongo server, sharding server, and support scripts
URL : http://www.mongodb.org
License : AGPL 3.0
Description : Mongo (from "huMONGOus") is a schema-free
: document-oriented database.
:
: This package provides the mongo server software, mong
: sharding server softwware, default configuration file
: and init.d scripts.

- 利用 yum 工具下载一个 rpm 包

有时，我们需要下载一个 rpm 包，只是下载下来，拷贝给其他机器使用，前面也介绍过 yum 安装 rpm 包的时候，首先得下载这个 rpm 包然后再去安装，所以使用 yum 完全可以做到只下载而不安装。

**a）首选要安装 yum-downloadonly**

[azureuser@mono src]$ sudo yum install -y yum-plugin-downloadonly.noarch

**b）下载一个 rpm 包而不安装，并且下载到指定的目录**

[azureuser@mono src]$ sudo yum install -y yum-presto.noarch --downloadonly --downloaddir=/usr/local/src
Loaded plugins: downloadonly, security
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package yum-presto.noarch 0:0.6.2-1.el6 will be installed
--> Processing Dependency: deltarpm >= 3.4-2 for package: yum-presto-0.6.2-1.el6.noarch
--> Running transaction check
---> Package deltarpm.x86_64 0:3.5-0.5.20090913git.el6 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
===========
Package Arch Version Repository
Size

```
================================================================================
===========
Installing:
 yum-presto                 noarch                 0.6.2-1.el6                              base
32 k
Installing for dependencies:
 deltarpm                   x86_64                 3.5-0.5.20090913git.el6                  base
71 k

Transaction Summary
================================================================================
===========
Install       2 Package(s)

Total download size: 102 k
Installed size: 252 k
Downloading Packages:
(1/2): deltarpm-3.5-0.5.20090913git.el6.x86_64.rpm                 |  71 kB     00:00
(2/2): yum-presto-0.6.2-1.el6.noarch.rpm                           |  32 kB     00:00
--------------------------------------------------------------------------------
Total                                                  194 kB/s | 102 kB
00:00


exiting because --downloadonly specified

[azureuser@mono src]$ ls /usr/local/src
convmv-1.15-1.el6.rf.noarch.rpm                 libgdiplus-2.10.tar.bz2
deltarpm-3.5-0.5.20090913git.el6.x86_64.rpm     mono
jexus-5.4                                       mono32
jexus-5.4.tar.gz                                unrar-4.2.3-1.el6.rf.x86_64.rpm
libgdiplus-2.10                                 yum-presto-0.6.2-1.el6.noarch.rpm
```

# LSB 推荐的基准打包格式

为了使 RPM 包能在所有的 Linux 发行版上安装和运行，LSB（Linux Standard Base）1.2 规范对 Linux 上 RPM 包格式提供一个基准建议。该规范建议 Linux 上的 RPM 包遵循 1997 版本的 Maximum RPM（http://www.rpm.org/max-rpm）附录部分的规格说明以及一些额外的限制。这些现在可以在 LSB 网站（http://www.linuxbase.org/spec/refspecs/LSB_1.2.0/gLSB/swinstall.html#FTN.PKG-1）上找到。

在 Linux 上，还有一个可用的包管理器是 DEB。除了包格式不同外，Deb 和 RPM 在很多方面都很相似。虽然几乎所有的 Linux 平台上最常见的是用 RPM 来打包应用程序，但在 Linux Debian 发行版上，是用 DEB 来打包应用程序的，RPM 包在转换成 DEB 格式后，也可以安装

在 Debian Linux 上。一个叫做 alien 的工具可以把 RPM 包转换成 DEB 包，也可以把 DEB 包转换成 RPM 包。

# APT 工具

软件包管理是区分不同发行版的一大特征，如 RedHat 使用 RPM 软件包来管理软件，Debian 使用 Deb 软件包来管理软件。apt-get 是 Debian 的 Deb 软件包管理工具，它的最低底层还是调用 dpkg 包管理程序，通过 apt-get 工具可使我们很好地解决软件包的依赖关系，方便软件的 安装和升级。
要使用好 apt-get 就要配置好一个名为 sources.list 的资 源列表，资源列表指向 Debian 系统的软件库，apt-get 会从该软件库安装各种软件包。sources.list 文件位于/etc/apt 目录下， 下面是 Sarge、Etch 和 Sid 三个版本的写法，你可任选一种，最好不要多版本混用：

```
#sources.list for Sarge(stable):
deb http://http.us.debian.org/debian stable main contrib non-free
deb http://non-us.debian.org/debian-non-US stable/non-US main contrib non-free
deb http://security.debian.org stable/updates main contrib non-free
#Uncomment if you want the apt-get source function to work
#deb-src http://http.us.debian.org/debian stable main contrib non-free
#deb-src http://non-us.debian.org/debian-non-US stable/non-US main contrib non-free

#sources.list for Etch(testing):
deb http://http.us.debian.org/debian testing main contrib non-free
deb http://non-us.debian.org/debian-non-US testing/non-US main contrib non-free
deb http://security.debian.org testing/updates main contrib non-free
#Uncomment if you want the apt-get source function to work
#deb-src http://http.us.debian.org/debian testing main contrib non-free
#deb-src http://non-us.debian.org/debian-non-US testing/non-US main contrib non-free

#sources.list for Sid(unstable):
deb ftp://ftp.us.debian.org/debian unstable main contrib non-free
deb ftp://non-us.debian.org/debian-non-US unstable/non-US main contrib non-free
#Uncomment if you want the apt-get source function to work
#deb-src http://http.us.debian.org/debian unstable main contrib non-free
#deb-src http://non-us.debian.org/debian-non-US unstable/non-US main contrib non-free
```

sources.list 文件的内容决定了 Debian 的版本。安全更新只存在于 stable 和 testing 版中，unstable 没有安全更新。进入 stable 的软件都经过严格的依赖测试和安全测试，所以如果你想系统稳定，用于工作，最好使用 stable，如果你想使用最新版的软 件，就使用 testing 或 unstable。Woody、Sarge 和 Sid 是 Debian 3.x 三个版本中的代号，我们一般都是以代号来称呼 debian 不同版本。所有 Debian 发行版的代号全都取自电影 Toy Story，Woody 是那个牛仔，Sarge 是绿色塑胶军队的领导，Sid 是破坏玩具的小孩。

- apt-get update

更新软件包信息 库。在 Debian 中，软件包是通过一个数据库来管理的，通过这个数据库中可跟踪你系统中已安装、没有安装和现在可安装的软件包信息。apt-get 安装 软件包时就是

依靠这个数据库来解决软件包间的依赖关系，从而可自动安装相关软件。我们需定期运行该命令，从而保持数据库的信息为最新。

- apt-get install package_name1 package_name2 package_name3 ...

安装软件包。如果软件包 需其它软件包支持，apt-get 会通过搜索软件包数据库找到这种依赖关系，一起下载相关软件。在一个命令行中可同时安装多个软件包，中间用空格隔开即 可。安装的软件包默认会存放在/var/cache/apt/archives 目录下，以便以后重新安装。如果已安装的软件包损坏了，你可通过 --reinstall 选项来重新安装。如：

# apt-get --reinstall install package_name

在需安装的软件包名后加一个减号会删除软件包，如：apt-get install package_name-。

只 是下载软件，不解包和安装使用-d 选项，如：

# apt-get -d install package_name

使用--dry-run 选项可使 apt-get 在安装软件包前进行测试，如：

# apt-get install package_name --dry-run

Debian 软件包的名字和软件名不同，所以在安装前如不知道软件包的名字，可到 Debian 的官方软件库查询，网址是：http://www.debian.org/distrib/packages/。 或者用下面介绍的 apt-cache search package_name 命令来查询。

- apt-get remove package_name1 package_name2 package_name3 ...

删除软件包。如果你想删除没 用的软件包，只要使用该命令即可。如果你想把该软件的配置文件也删除，可以用--purge 选项，如：

# apt-get --purge remove package_name

类似地，在删除软件包名后加一个加号会安装软件包，如：apt-get remove package_name+。

- apt-get source package_name1 package_name2 package_name3

下载软件包的的源码版本。

- apt-get upgrade package_name1 package_name2 package_name3 ...

软件包升级功能是 APT 系统这么成功的主要原因。通过该命令，我们就可把软件升级到最版本。在使用该命令前，最好先运行 apt-get update 命令，以更新软件包数据库。但该方案不是更新系统最好的方法，一些包会因为包依赖问题而保留(kept back)一些旧的软件包。Debian 提供了一个更好的升级方案，就是用 dis-upgrade。下面一节会详细介绍。

- apt-get dist-upgrade

更新整个 Debian 系统。可从网络或本地更新整个系统。它会重新安排好包的依赖性。如果有些包由于一些原因实 在不能更新，我们可通过以下命令查询原因：

# apt-get -o Debug::pkgProblemResolver=yes dist-upgrade

用 apt-show-versions -u 可获得可升级软件包的列表。该命令还有一些有用的选项，可用-h 选项查看详细帮助，了解更多功能。

- apt-get clean

删除下载了的软件包，当我们通过 apt-get 安装软件包时，APT 会把软件包下载到本地/var/cache/apt /archives/目录。该命令会删除该文件夹内的除锁住外的所有软件包。

- apt-get autoclean

删 除已下载的旧版本的软件包。该命令类似于上面的命令，但它会有选择地删除旧版本的软件包。

- apt-get dselect-upgrade

通过 dselect 的“建议”和“推荐”功能更新系统。dselect 是 Debian 中一个功能强大的包管理工具。它可帮助用户选择软件包来安装，其中一个有用功能是它会建议和推荐安装其它相关软件包。我们可在 APT 中使用它这个功能。

- apt-get check

检查系统中已安装软件包的依赖性。

# 项目管理任务

当确认待移植的应用程序所用的编译器、编译工具和第三方产品在 Linux 上可用或存在类似的工具后，下一步就是要检查和确定项目的交付物和项目任务。当从更深的层次(例如代码行数、模块数、第三方软件、测试方法、打包等等)来检查待移植应用程序的结构时，移植项目要做的工作就开始出现了。项目经理要确认那些工作是项目移植该做的，那些是不该做的。我们可以从第 1 章的讲述中知道，为什么确定哪些该做那些不该做对项目移植很重要。项目的范围明确以后，项目交付物、项目任务、初始进度计划，以及项目计划的其他内容就可以确定下来了。下面的列表简要描述了项目计划的各项内容：

- 项目交付物。每个移植项目都有多个交付物，每个交付物都有一些任务与之相关联。可以用一个类似下面的表格，把任务和项目交付物关联起来，该表清楚的写明了每个任务的起止时间及所有者。

| 活动/任务 | 开始时间 | 结束时间 | 所有者 | 状态 | 备注 |
|---|---|---|---|---|---|
| 交付物 1 | | | | | |
| 任务 1 | | | | | |
| 任务 2 | | | | | |

- 成功标准。每个交付物都有一个成功标准。该标准可以很简单，例如，可以简单地把“交付物通过单元测试”作为成功标准。每个交付物的成功标准都要明确下来，以此来确定与该交付物相关联的任务是否完成。
- 项目开始时间和结束时间。根据与项目交付物相关联的任务，可以确定移植项目的开始和结束时间。这可以让与项目有关的人员知道该项目需要持续多长时间。有些情况下，可能会要求移植项目在给定的时间内完成，此时需要重新检查项目交付物和任务是否需要修改，以便能在该时间内完成。
- 需要的资源。与项目交付物关联的任务确定以后，需要什么样的技能来完成这些任务就很明确了，进而可以确定需要哪些技术人员及硬件资源。与此同时，还要确定什么时候需要这些资源，以及这些资源需要投入到该项目多长时间。
- 关键依赖关系。确定关键依赖关系，包括什么时候需要这些被依赖的对象，以及从哪儿可以获得他们。
- 风险。确定可能影响项目进度、成本和范围的风险。例如，可能影响项目进度的一个风险是关键依赖关系。如果因为某些不可预见的因素导致被依赖的对象不能按时交付，则势必影响到整个项目的进度。
- 项目状态交流。最后（但也是不可忽视的一项），创建一个项目状态交流的流程，以确定那些人，什么时候以及多长时间需要进行一次项目状态的交流。

负责调查应用程序的移植人员与项目经理之间的协作交流是非常重要的。从调查过程中收集到信息直接影响到项目的范围，甚至能决定该项目是否可以移植。明确一些关键内容，例如编译器、编译环境和第三方产品依赖等，可以帮助确定项目移植计划的一些必要元素。

## 小结

通过使用第 1 章中的评估问卷，你现在应该对前面所说的任务有了更明确的理解。你也可能已经注意到，Linux 上开发环境与 Windows 的开发环境有很大不同，但是对于.NET 开发人员或移植工程师来说，在 Windows 上.Net 开发和 Linux 上的 Mono 开发是一个类似的过程。Linux Mono 上也有一整套的开发工具，因而 Mono 上的开发和移植是个很直接明了的过程。

- Mono 的编译器遵循.NET 的编译器相同的标准。虽然不能说所有的待移植的应用程序都可以移植，但是应用程序中遵循这些标准的部分，可以很方便的用 Mono 的编译器移植。
- Linux 上的打包环境可以方便地用来打包自己的应用程序。开源社区对打包环境提供了很好的文档和支持。
- 创建一个完整的项目计划也是调查阶段的一个工作。把调查信息反馈给项目经理，以便项目成员与项目经理充分讨论和交流项目任务、交付物、成功标准、进度计划、资源额依赖关系等内容。

下一章我们会从移植人员的角度，来讲述移植过程的“分析”这一步。我们开始深入地分析应用程序，并确定要做的工作级别。在此过程中，如果需要的话，可能会对每个交付物关联任务做进一步的提炼或调整。

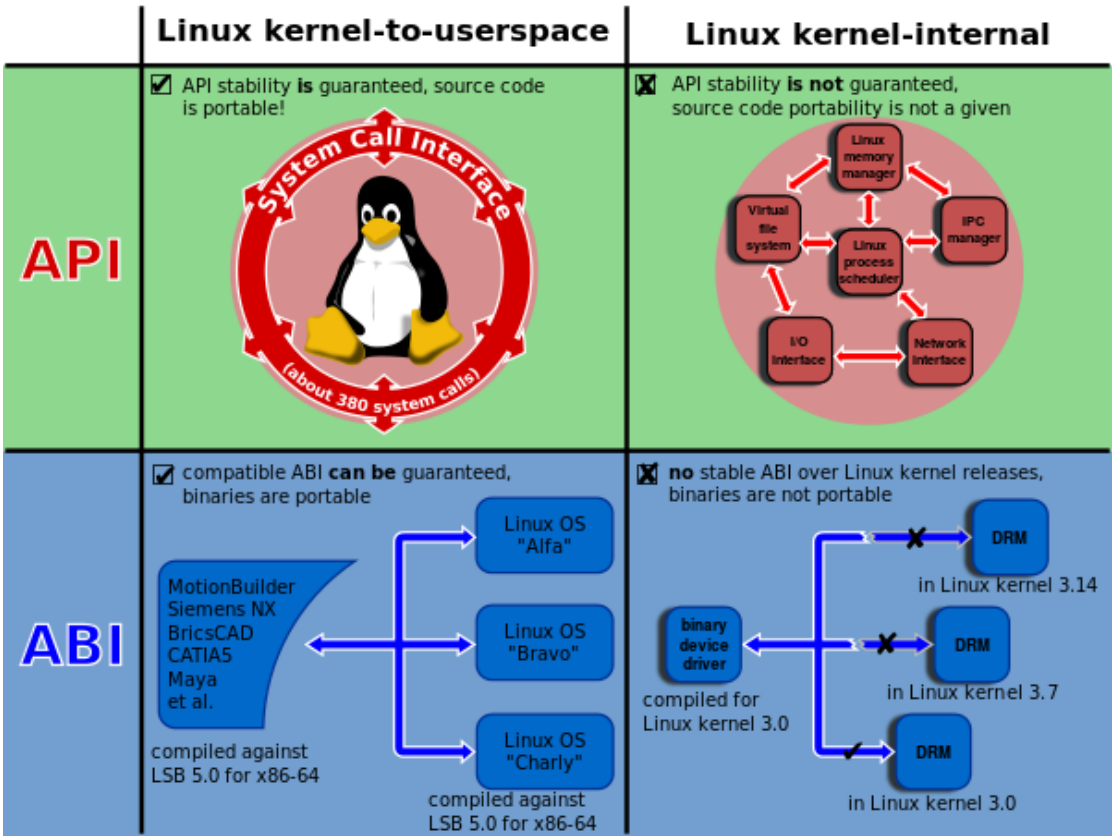
# 第三章 分 析

在初始的调查完成以后，我们需要更深入地分析待移植的应用程序。要分析的内容包括（但不限于）应用程序使用的 API、动态库的装载接口、线程、网络接口以及操作系统提供的其他功能。本章要讲述的内容假设移植工程师熟悉移植的应用程序，或者从熟悉应用程序的程序员哪里了解了应用程序的设计结构。

## Linux 标准

几乎所有的 Linux 发行商都认可标准化和可移植性所带来的作用和价值，并努力与 Linux 开发和维护人员一起，使各 Linux 发行版都遵循相同的标准。当前主要的 Linux 工业标准是 Linux 标准基础（Linux Standard Base，LSB），这是一个由 Free Standards Group(FSG)维护的项目。LSB 主要是基于 IEEE POSIX 和 Open Group Single UNIX Specification 标准，同时也描述了 Linux 与 UNIX 或 POSIX 不兼容的地方。几乎所有的 Linux 主要发行商都通过了 LSB 认证。

LSB 除了提供程序源码 API 规范外，还提供了各通用的 Linux 硬件平台上的应用程序二进制接口（Application Binary Interface，ABI）规范。提供该规范的目的是，对遵循 LSB 规范的应用程序，保证编译后的二进制程序在同一硬件平台上各 Linux 发行版之间的可移植性。各发行商既要通过 API 认证，也要通过 ABI 认证。对于独立软件提供商（Independent Software Vendor，ISV）来说，也可以使用 FSG 提供的工具来认证他们的应用程序，以使之遵循 LSB 规范。




虽然 Linux 内部已经实现了标准化，但是 Linux 本身并不完全遵循 IEEE POSIX 或 Open Group SUS 规范。Mono 平台帮助我们解决 Linux 各大发行版之间的兼容性，基于 Mono 的应用程序移植在 Linux 上工作也就很容易了。

## GNU libc 库

glibc 是 gnu 发布的 libc 库，即 c 运行库。glibc 是 linux 系统中最底层的 api，几乎其它任何运行库都会依赖于 glibc。glibc 除了封装 linux 操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现。由于 glibc 囊括了几乎所有的 UNIX 通行的标准，可以想见其内容包罗万象。而就像其他的 UNIX 系统一样，其内含的档案群分散于系统的树状目录结构中，像一个支架一般撑起整个操作系统。在 GNU/Linux 系统中，其 C 函式库发展史点出了 GNU/Linux 演进的几个重要里程碑，用 glibc 作为系统的 C 函式库，是 GNU/Linux 演进的一个重要里程碑。

glibc 是 Linux 系统的核心库，稍有不慎就会导致系统崩溃。如果在程序中必须使用另一版本的 glibc，则需要小心从事。

http://pkgs.org/download/libc.so.6

除了 libc 和 libm 库，UNIX 系统库没有其他标准的命名规范。Linux 上的一些系统库可能和 UNIX 平台上库的名称不同，这就需要知道 Linux 上各库所包含和支持的功能。

下面列出了 GNU libc 库所包含的库文件以及对应的描述(注释 5)：

- ld.so，为使用了共享库的可执行程序提供的一个辅助程序;
- libBrokenLocal.[a,so]，Mozilla 等应用程序用以解决被破坏的 locale 的库文件;
- libSegFault.so，段错误信号处理器，它试图捕获段错误信号。
- libanl.[a,so]，异步的名称查询库。

- libbsd-compat.a，在 Linux 上运行 BSD 程序时需要的库。
- libc.[a,so]，最主要的 C 库(常用的 C 函数的集合)。
- libcrypt.[a,so]，加密库。
- libdl.[a,so]，动态链接接口库。
- libg.a，g++运行时库。
- libieee.a，IEEE 浮点运算库。
- libm.[a,so]，数学库。
- libmcheck.a，包含启动时运行的代码。
- libmemusage.so，memusage 用来收集应用程序内存使用情况的库。
- libnsl.a，网络服务库。

-libnss_comkpat.so,libnss_dns.so, libnss_files.so, libnss_hesiod.so, libnss_nis.so, libnss_nisplus.so，NSS(Name Service Switch)库，包含解析主机名、用户名、组名、别名、服务、协议等的函数。

- libpcprofile.so，包含一些跟踪统计代码行消耗 CPU 时间的概要分析(profiling)函数。
- libpthread.[a,so]，POSIX 线程库。
- libresolv.[a,so]，包含为网络域名服务器创建、发送、解释网络包的函数。
- librpcsvc.a，包含提供各种 RPC 服务的函数。
- librt.[a,so]，包含 POSIX1.b 实时扩展所定义的大部分接口函数。
- libthread_db.so，包含开发多线程程序调试器的函数。
- libutil.[a,so]，包含常用的 UNIX 工具使用的“标准”函数。

上面这些库大多位于/usr/lib 目录，也有一些在/lib 目录下，例如 libSegFault.so。

GNU glibc 发布了一个描述其所遵循的标准的报告。该报告同时也列出了 GNU libc 需要改进的地方。写作本书时，该报告显示 GNU libc 通过了 FIPS POSIX90、POSIX96、UNIX98、ANSI、C89/99，和 ISO9899 标准的头文件一致性检查。所有主要 Linux 发行版的 glibc 也都遵循 LSB 规范。

# 共享库

Linux 上，共享库可以有不同的文件扩展名，例如，共享库可以以.so 或.so.1.0 结束。以.so.x.x(x 为数字)结尾的共享库叫版本化库。第一个数字代表大版本号，第二个数字代表小版本号。有些情况下，共享库的扩展名还可以是.so.x.x.x(x 为数字)的形式，这里最后一个数字代表发布号，并且是可选的。下面给出了共享库文件名的格式：

lib<name>.so.<major>.<minor>.<release>

大版本号、小版本号，以及发布号的变化反映了对共享库所作的不同类型的修改。下面是对增大大版本号、小版本号和发布号的一些指导：

- 当对共享库提供的接口做了与以前版本不兼容的改变时，需要增大大版本号。这个大的改变意味着依赖该库先前大版本的应用程序需要作相应修改才能使用大版本更新后的库。
- 当共享库增加了新的接口同时也保留了原来的接口时，增大小版本号。
- 当作了与以前兼容的修改又没有增加新接口时，增大发布号。这通常是对一些实现做了改动以提高性能和扩展性。

# 库版本化

在共享库和应用程序之间维护二进制级的兼容性或 ABI 是很重要的。共享库的 ABI 是应用程序依赖的运行时接口;如果每次发布时共享库的 ABI 都与以前的兼容，那么在其中某一个版本的共享库上编译的应用程序不需要任何改动就可以在后续版本上运行。库版本化就是 Linux 以及同期的其他操作系统实现二进制兼容性的方法。
Linux 提供了两种不同的技术来实现库版本化：外部库版本化和符号版本化。

# 外部库版本化

链接过程中，链接器(ld)会查找以.so 结尾的共享库文件。以.so 结尾的库文件叫链接器名称，这是由他们在 Linux 上的使用方式决定的。当编译一个依赖某一共享库的应用程序时，仅仅是该共享库的 soname(不是共享库的文件名)作为依赖关系被记录在应用程序的二进制代码中。运行时链接器就是使用共享库的 soname 来查找和装载该库的。共享库的 soname 只包含有大版本号(例如，libfoo.so.1)

当修改后的共享库与以前版本不兼容时，新的共享库必须有一个新的外部版本名称。也就是说，该库的 soname 必须改变。这些不兼容的修改包括：删除一个符号，去掉某函数的一个参数，改变了某函数的语义属性以致与以前的定义不再一致并且与老版本二进制不兼容等等。我们来看下面的例子。

# 符号版本化

就像前面所提到的，当对共享库所作的修改能够向前兼容时，我们只增大小版本号。这种修改包括增加一些新的接口同时又不改变已有的接口。但是，即使只做这种小版本的修改，也会出现一个很重要的问题：一个在某一小版本的共享库上编译的应用程序并不一定能够在以前小版本的库上运行。这是因为该应用程序可能使用了新增加的、以前小版本的库中没有的接口。为了解决这个问题，引入了符号版本化。符号版本化允许共享库记录下每个小版本都新增了什么内容。

在 Linux 上，GNU ld 可以使用-version-script 连接器选项来创建符号版本化的共享库。编译器选项-Wl,--version-script=mapfile 告诉链接器哪些符号要从生成的共享库中输出出来。每个符号分属 global(被输出)和 local(不被输出)两类中的一种。

可以看到，这次 main 只引用了版本化库的 LX_1.1。
GNU ld 还允许在定义符号的源文件中把符号绑定到某一版本中，而不仅仅是在脚本文件中指定。另外，GNU ld 还允许同一函数的多个版本出现在同一个共享库中。更多详细信息，请参考 GNU ld 手册(注释 13)和 Ulrich Drepper 的文章“How to Write Shared Libraries”。
从 2.1 版本开始，glibc 就已经实现了符号版本化。符号版本化同时也是 LSB 规范 1.2 及更高版本的一部分。

# 动态链接库

Linux 动态链接器(/lib/ld.so.1 或/lib64/ld64.so.1)查找和装载应用程序所需的共享库，准备应用程序的运行，然后运行应用程序。

在所有现代 UNIX 操作系统上，都有一些环境变量可以影响动态链接器的运行。例如 AIX 上的环境变量 LIBPATH 可以改变动态链接器的搜索路径。以下环境变量可以影响到 Linux 上动态链接器的运行：

- LD_LIBRARY_PATH，以冒号分开的目录列表，运行时会在这些目录中查找需要的库。
- LD_PRELOAD，以空格分开的库列表，这些库会在其他所有库之前装载。这常常用来有选择的覆盖某些共享库中的函数。
- LD_BIND_NOW，如果该环境变量设置成非空字符串，动态链接器会在程序启动时解析所有符号，而不是首次引用时才解析符号(也就是常说的“延迟绑定”)。这在使用调试器时非常有用。
- LD_TRACE_LOADED_OBJECTS，如果该环境变量设置成非空字符串，程序会列出它所依赖的共享库，就像运行 ldd 命令一样，而不是正常的执行。

Linux 动态链接器采用广度优先(breadth first)的方式解决库的依赖关系。也就是说，首先是可执行程序所依赖的库按照动态节(dynamic section)列出的顺序被装载进来，然后是“第一个被依赖的库”所依赖的库按照同样的方法装载进来，以此类推，直到所有的依赖关系都被解决。

# 国际化（i18N）和本地化

人们常把 I18N 作为“国际化”的简称，其来源是英文单词 internationalization 的首末字符 i 和 n。18 为中间的字符数。国际化是指在设计软件时，将软件与特定语言及地区脱钩的过程。当软件被移植到不同的语言地区时，软件本身不用做内部工程上的改变或修正。本地化则是指当移植软件时，加上与特定区域设置有关的资讯和翻译文件的过程。 国际化和本地化之间的区别虽然微妙，但却很重要。国际化意味着产品有适用於任何地方的潜力；本地化则是为了更适合於特定地方的使用，而另外增添的特色。用一项产品来说，国际化只需做一次，但本地化则要针对不同的区域各做一次。 这两者之间是互补的，并且两者结合起来才能让一个系统适用於各地。

I18N 对项目移植会有多大的影响，在很大程度上取决于待移植的应用程序。如果一个应用程序仅需要一些简单的消息目录转换、时期和时间显示，或者使用正则表达式进行简单的文本串查找，那么把这些功能从 Windows 平台移植到 Linux 平台上，还是比较容易的。但是如果应用程序进行了复杂的文本分析，移植这些内容有可能就是比较困难的了。

Linux 遵循 ISO 对标准 locale 名称的命名规范[locale]_[territory].[codeset]。其中，locale 由两个字符组成，代表语言；territory 由两字符组成，代表国别。例如 en_US.iso885915 和 zh_CN.gb1830。但是，每个系统上可用的 locale 和 locale 的内容是各不相同的。移植使用了 locale 的复杂应用的应用程序，可能需要学习具体的语言规范和翻译规则，甚至需要修改了 Linux 上已有的 locale，才能使移植后的应用程序像在原系统上那样运行。

在 Linux 上，可以使用 locale –a 命令查看系统上安装的 locale，解码文件可以在 /usr/share/locale/locale.alias 里找到。

# 大小端环境

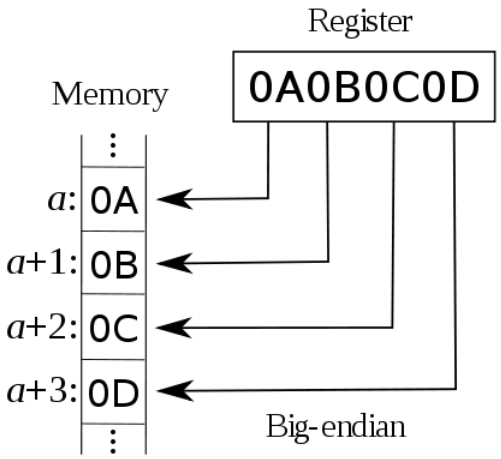
Linux 最初是在 Intel 平台上开发的，而 Intel 平台主要是一个小端（little-endian）环境，但是现在 Linux 已经被移植到了很多支持大端（big-endian）的硬件平台上。大小端（Big/Little-endian，也叫字节序），指的是一个数据元素及其每个单独字节是如何存放的。在 big-endian 环境中，最低地址放在多字节的最高位（或者最左侧位）；在 little-endian 环境中，最低地址放在多字节的最低位（或者最右侧位）。通常来说，第 0 位在 big-endian 环境中是最高位，但是在 little-endian 环境中，第 0 位是最低位。

我们一般把字节（byte）看作是数据的最小单位。当然，其实一个字节中还包含 8 个比特（bit）—有时候我奇怪为什么很多朋友会不知道 bit 或是它和 byte 的关系。当我们拿到一系列 byte 的时候，它本身其实是没有意义的，有意义的只是“识别字节的方式”。例如，同样 4 个字节的数据，我们可以把它看作是 1 个 32 位整数、2 个 Unicode、或者字符 4 个 ASCII 字符。

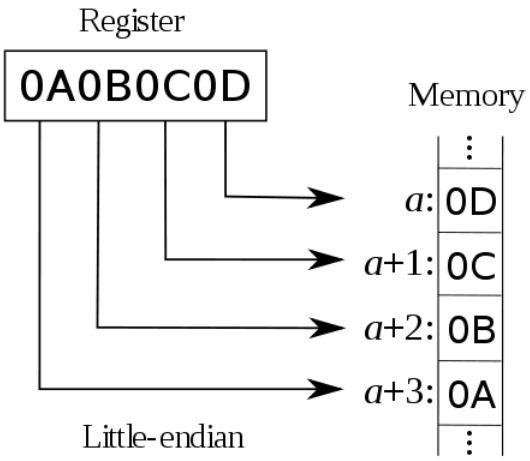
同样我们知道，在一个 32 位的 CPU 中“字长”为 32 个 bit，也就是 4 个 byte。在这样的 CPU 中，总是以 4 字节对齐的方式来读取或写入内存，那么同样这 4 个字节的数据是以什么顺序保存在内存中的呢？例如，现在我们要向内存地址为 a 的地方写入数据 0x0A0B0C0D，那么这 4 个字节分别落在哪个地址的内存上呢？这就涉及到字节序的问题了。

每个数据都有所谓的“有效位（significant byte）”，它的意思是“表示这个数据所用的字节”。例如一个 32 位整数，它的有效位就是 4 个字节。而对于 0x0A0B0C0D 来说，它的有效位从高到低便是 0A、0B、0C 及 0D——这里您可以把它作为一个 256 进制的数来看（相对于我们平时所用的 10 进制数）。

而所谓大字节序（big endian），便是指其“最高有效位（most significant byte）”落在低地址上的存储方式。例如像地址 a 写入 0x0A0B0C0D 之后，在内存中的数据便是：




而对于小字节序（little endian）来说就正好相反了，它把“最低有效位（least significant byte）”放在低地址上。例如：




对于我们常用的 CPU 架构，如 Intel，AMD 的 CPU 使用的都是小字节序，而例如 Mac OS 以前所使用的 Power PC 使用的便是大字节序（不过现在 Mac OS 也使用 Intel 的 CPU 了）。此外，除了大字节序和小字节序之外，还有一种很少见的中字节序（middle endian），它会以 2143 的方式来保存数据（相对于大字节序的 1234 及小字节序的 4321）。

BinaryWriter 和 BinaryReader

在.NET 框架操作数据流的时候，我们往往会使用 BinaryWriter 和 BinaryReader 进行读写。这两个类中都有对应的 WriteInt32 或是 ReadInt32 方法，那么它们是如何处理字节序的呢？从 MSDN 上我们了解到 BinaryReader 使用小字节序读取数据。这意味着：

```
var stream = new MemoryStream(new byte[] { 4, 1, 0, 0 });
var reader = new BinaryReader(stream);
int i = reader.ReadInt32(); // i == 260
```

与之类似，自然 BinaryWriter 也是使用小字节序来写入数据。

BitConverter

有时候我们还会使用 BitConverter 来转化 byte 数组及一个 32 位整数（自然也包括其他类型），这也是涉及到字节序的操作，那么它们又是如何处理的呢？与 BinaryWriter 和 BinaryReader 的“固定策略”不同，BitConverter 的行为是平台相关的。

首先，BitConverter 有一个只读静态字段 IsLittleEndian，它表示当前平台的字节序。由于我们为不同的 CPU 会安装不同的.NET 类库，因此您现在如果通过.NET Reflector 来查看这个字段会发现它被设置为一个常量 true。那么接下来，BitConverter 上的各个方便便会根据 IsLittleEndian 的值产生不同行为了，例如它的 ToInt32 方法：

```
public static unsafe int ToInt32(byte[] value, int startIndex)
{
    // ...

    fixed (byte* numRef = &(value[startIndex]))
    {
        if ((startIndex % 4) == 0)
        {
            return *(((int*)numRef));
        }
        if (IsLittleEndian)
```

```
            {
                return numRef[0] | (numRef[1] << 8) | (numRef[2] << 16) | (numRef[3] <<
24);
            }

            return (numRef[0] << 24) | (numRef[1] << 16) | (numRef[2] << 8) | numRef[3];
        }
    }
```

显然，这里会根据 IsLittleEndian 返回不同的值。

判断当前平台的字节序

在.NET Framework 中 BitConverter.IsLittleEndian 字段是一个常量，也就是说它在编译期便写入了一个静态的值。那么我们如果想要通过代码来判断当前平台的字节序，又该怎么做呢？其实这很简单：

```
static unsafe bool IsLittleEndian()
{
    int i = 1;
    byte* b = (byte*)&i;
    return b[0] == 1;
}
```

这里我们通过检查 32 位整数 1 的第一个字节来确定当前平台的字节序。当然，我们也可以使用其他类型，例如：

```
static unsafe bool AmILittleEndian()
{
    // binary representations of 1.0:
    // big endian: 3f f0 00 00 00 00 00 00
    // little endian: 00 00 00 00 00 00 f0 3f
    // arm fpa little endian: 00 00 f0 3f 00 00 00 00
    double d = 1.0;
    byte* b = (byte*)&d;
    return (b[0] == 0);
}
```

这段代码来自 mono 的 BitConverter 类库

.NET 类库中自带一个 Buffer.BlockCopy 方法，它的作用是将一个数组的字节——不是元素——复制到另一个数组中去。换句话说，一个长度为 100 的 int 数组经过完整的复制后，就变成了长度为 50 的 long 数组，因为一个 int 为 4 字节，而 long 为 8 字节。从文档上看，Buffer.BlockCopy 是与字节序相关的，也就是说，同样的.NET 代码在字节序不同的平台上得到的结果可能不同。因此，我建议在使用这个方法的时候多加小心。

面向特定字节序编程

我们知道，BitConverter 的工作结果是和当前平台的字节序相关的，但是在很多时候，尤其是根据某个公开的协议进行通信编程的时候，是需要固定一个字节序的。

为了保证.NET 代码的平台无关性，我们不能直接使用 BitConverter.GetBytes 或 ToInt32 方法进行转化。那么我们该怎么办呢？最直观的方法自然是手动进行转换：

```
static int ReadInt32(Stream stream)
{
```

```
        var buffer = new byte[4];
        stream.Read(buffer, 0, 4);

        return buffer[0] | (buffer[1] << 8) | (buffer[2] << 16) | (buffer[3] << 24);
    }
```

由于我们可以通过 BitConverter.IsLittleEndian 来得到当前平台的字节序，我们也可以用它进行判断：

```
    static int ReadInt32(Stream stream)
    {
        var buffer = new byte[4];
        stream.Read(buffer, 0, 4);

        if (BitConverter.IsLittleEndian)
        {
            Array.Reverse(buffer);
        }

        return BitConverter.ToInt32(buffer, 0);
    }

    static void WriteInt32(Stream stream, int value)
    {
        var buffer = BitConverter.GetBytes(value);

        if (BitConverter.IsLittleEndian)
        {
            Array.Reverse(buffer);
        }

        stream.Write(buffer, 0, buffer.Length);
    }
```

此外，我们知道 BinaryWriter 和 BinaryReader 都是依据小字节序进行读写的，因此我们也可以利用这点来读写数据流。

# 从 32 位移植到 64 位

Windows 64 位操作系统提供了一个为运行 32 位应用程序的兼容模式。它被称为 WOW 模式(Windows on Windows)。你有使用模拟器在 PC 上玩过电视游戏吗？它就类似这个模拟器。WOW64 是一个在 64 位机器上运行 32 位应用程序的模拟器。尽管我们还会在后面讨论一些关于 WOW64 模式的问题，你可以先访问这里来了解 WOW 模式的更多细节。

检测"Any CPU"进程的位数(Detecting the bitness of a "Any CPU" process)

当你把.NET 组件编译成为"Any CPU"时，会发生什么事情呢？如果你知道这个.NET 组件总是运行在 X86 环境下，那运行这个组件的进程也会是 32 位的。但是如果这个.NET 组件可以运行在 X86 和 X64 环境下，那运行这个组件的进程是 32 位还是 64 位呢？更进一步说，

如果它运行在 X64 操作系统上，JIT 编译器会选择 32 位还是 64 位来编译它呢？一般来说，它是 64 位的。

Linux 上不存在 WOW64 这样的系统，我们在程序里面就不能同时存在 32 位和 64 位的程序集。

# 第四章 移植 ASP.NET 应用程序

http://www.mono-project.com/docs/getting-started/application-portability/

# 第五章 移植 Windows 服务

# 第六章 测试和调试

http://ceeji.net/blog/mono-environment-track-dotnet-memory-alloc/
http://www.mono-project.com/docs/advanced/runtime/docs/soft-debugger/
.NET 程序在 Mono 环境之下的内存分配和在 Microsoft .NET Framework 环境是不太相同的，虽然它们的兼容性很高，但仍有区别。