第一章 软件安全基础

计算机软件是计算机应用程序系统中存在并使用的所有程序和文档的统称。程序是计算任务的处理对象和处理规则的描述，文档则是用于理解和掌握各种程序工作的解释说明。软件由系统软件和应用程序软件两部分组成。系统软件是计算机预存的用于调节、管理各种计算机硬件的软件，不需要用户自行安装，如文件管理系统、资源管理器等；应用程序软件是用户根据自身需要而安装的各种类型的软件如社交软件、办公软件、游戏软件、管理软件等等。

软件正在吞噬整个世界，40 年前，软件只是芯片的附属物，其作用范围限定在单片机之内，普罗大众还不知道有“软件”这么一个东西。30 年前，软件开始崭露头角，其作用范围限定在操作系统中的小工具，大家对软件的印象就是“算算数”或者“玩游戏”而已。20 年前，人们开始重视电脑的应用，但因为软件无法像硬件一样可触可见，而且无法做成固定资产，仍得不到重视。近十几年来，软件大举应用于各种类型的机器，成为机器中的“软零件”“软装备”，并且成为机器的大脑和灵魂，主宰了机器世界的运行逻辑；同时，开发任何复杂产品，都难以离开软件手段的支撑，从此，世界上再不能缺少软件。

软件定义万物是大势所趋。1968 年，绕月的阿波罗 8 号飞船升空第 5 天，宇航员误操作删除了所有导航数据，致使飞船无法返航。程序员们连夜奋战 9 小时，设计出了一份新导航数据并经由巨大的地面天线阵列上传到阿波罗 8 号，让它顺利返航。1969 年，阿波罗 11 号飞船登月前，危机再次发生。当年电脑算速极慢，系统只能存储 12KB 数据，临时存储空间仅 1KB。飞船登月前几分钟，电脑因过度计算几近崩溃。程序员们首创的“异步处理程序”，让阿波罗 11 号学会了“选择”：当电脑运行空间不足时，最宝贵的存储空间只留给最关键的登月任务，其他任务暂停，由此而让登月舱成功降落在月球表面。从绕月到登月，软件有序地控制了飞船，把人类首次送上月球。用软件程序通过赛博空间来远程控制物理设备，在 1969 年就已经实现。以程序化指令不限时空地控制物理设备，其实一直是软件的终极使命。

从今天的视角来看，上述代码的体量与现在先进设备中的代码体量相比，简直是微不足道。但更重要的是，以上的尝试走通了一条对当今工业来说极其重要的技术路径：软件可以在赛博空间中，不限时空地传输和安装，不限时空地运行其中的指令，体现人类设定的逻辑和执行过程，让遍布各处甚至远在天边的物理设备按照人类意愿工作。如果说五十年前的阿波罗飞船上的软件还只是航天工业领域的软件，那么在今天，软件已经融入日常生活，支配工业设备，甚至影响人类行为。以智能汽车为例，智能驾驶辅助系统、停走型全速自适应巡

航系统、交通拥堵辅助系统、防碰撞预警系统、安全距离报警等等功能正逐步解放驾驶员，为安全驾驶保驾护航，而这些功能的实现全都依靠形形色色的软件。此外，截至 2019 年，我国涉及程序或软件问题的召回达到213次,涉及车辆683.02万辆,约占总召回数量的9%，传统的召回需要走内部及外部审批过程，时间和金钱的成本都非常高。因而 OTA 方式应运而生，OTA 全称“Over-The-Air”，即空中下载技术，早期被广泛应用在手机行业中，终结了手机软件升级需要连接电脑、下载软件、再安装更新的繁复操作。近年来，随着汽车网联技术不断发展，汽车 OTA 也成为了行业热词。通过 OTA 技术对汽车进行远程升级，不仅可以持续为车辆改善终端功能和服务，让车主拥有更便捷、更智能的用车体验，而且还可以被用于快速修复漏洞，帮助实施汽车召回。正在逐渐成为企业解决软、硬件系统问题的重要措施。可以说，软件正在重新定义汽车行业。

现在，软件进入各行各业（特别是工业领域），与各种传统的物理设备相结合，是一个不以人们意志为转移的大趋势。软件不仅嵌入了我们身边的器物，还在逐渐替代某些物理元件或零部件，我们不妨仔细观察身边的事物：家中的电器、代步的座驾、随身的手机、车间的设备、试验的仪器，一件一件，都已经开始了软件替代部分实体零件和相关操作的进程。此外，在网络基础设施领域，软件也在重新定义着一切。软件定义网络正在颠覆传统网络结构，传统概念中，决定网络结构的是路由器、交换机、PC 机等等硬件设备，而在软件定义网络中，提出要提高平台的可编程性，可以实现新型网络功能的配置，满足灵活多变的应用需求，将网络控制层与数据转发层分离，把软件控制功能放到网络管理器上，从而提高网络的管理控制能力。随着软件定义网络的概念落地，软件定义云数据中心、软件定义存储、软件定义计算等等新型概念不断出现，正悄悄地改变整个网络空间。可以预见，在软件赋能、赋智的加速推动下，工业文明将发展到一个前所未有的崭新阶段，促进人类社会从数字社会走向智能社会。

网络空间可以认为由代码子空间和数据子空间构成。代码子空间由各种软件构成，无论哪个领域，没有软件，就没有数据，数据从采集、处理、存储、传输等各个环节都需要软件的执行。比如有人说智能制造的核心是产品大数据，其实这些话只说对了一半，没有软件，再多的数据也体现不出其价值。因此，真正的幕后英雄是软件，是基于云计算架构的新形态软件——所有的数据都要靠无处不在的软件来进行分析和处理,以判断其是否具有使用价值和如何去使用它们，而人工是既不可能、也没必要去分析大数据的。

由于软件在现代社会中的重要性越来越高，软件安全引起各方的关注，软件安全是指采

取工程的方法使得软件在敌对攻击的情况下仍能继续正常工作，即采用系统化、规范化和可量化的方法来指导构建安全的软件。国务院学科评议组指出软件安全是网络空间安全的核心课程，主要研究安全软件的设计、实现和分析。同时软件安全也是 CSEC 2017 指出的六个知识领域之一，其中包含五个内涵属性：C-机密性，I-完整性，A-可用性，R-风险，AT-对抗。

与硬件相比，软件更容易受到安全威胁。当软件安全遭受攻击，或者运行期间出现错误时，可能会给用户、社会以及国家带来巨大的损失。2021 年全球组织遭受的网络攻击相较于 2020 年增加了 29%，因软件安全事件导致的损失高达 6 万亿美元。2021 年 2 月，黑客侵入弗罗里达州的一家水处理厂，通过篡改软件数据将该厂的氢氧化钠含量调高到了极高的水平，险些让整个城市中毒。2018 年 9 月份，特斯拉无钥匙进入和启动系统曝出 CVE（通用漏洞披露）漏洞，编号为“CVE-2018-16806”。该系统是由软件公司 Pektron 开发的，因此受影响的车辆还可能涉及同样应用了 Pektron 启动系统的迈凯轮、Karma、Triumph 等品牌车型。2016 年 10 月，黑客利用 Mirai 恶意代码控制了大量物联网和智能设备，利用千万级别的 IP 地址对美国东海岸 DNS 基础设施进行攻击，导致大半个美国的用户遭遇了一次集体“断网”事件。软件安全产生的问题已经影响到国民生活的方方面面，因而，软件安全的研究与防护工作迫在眉睫。

1.1 软件安全的概述

1.1.1 软件安全发展

软件安全的发展根据研究对象的不同大体可以分为三个阶段。早期软件安全主要研究 SSH、FTP、RPC 等系统软件和桌面软件的安全问题。因为，这一时期黑客攻击的目标以系统软件居多，通过攻击系统软件，黑客们往往能够直接获取 root 权限，这一阶段涌现了非常都的经典漏洞以及“exploit”，比如著名的黑客组织 TESO 就曾经编写过一个 SSH 的 exploit，并公然在 exploit 的 banner 中宣称曾经利用这个 exploit 入侵过美国中央情报局。这一阶段的另外一个重要的分支是桌面软件安全，其代表是浏览器攻击，一个典型的攻击场景是黑客构造一个恶意网页，诱使用户使用浏览器访问该网页，利用浏览器中存在的某些漏洞执行 shellcode，通常是下载一个木马在用户机器里执行。常见的针对桌面软件的攻击目标还包括微软的 Office 系列软件、Adobe 系列软件、多媒体播放软件、压缩软件等装机量大的流行软件。这一阶段他们主要采用了栈溢出、堆溢出、整数溢出、格式化字符串溢出、函数指针溢出、ROP、Ret2Lib、DOP 等攻击手段。与此相对应也产生了许多防御手段，软件层面如 Canary、DEP、ASLR、PIE 等等，硬件层面，因特尔等公司提出了芯片级别的威胁检测技术和

Security Essential 框架。

第一阶段的一个里程碑事件是 2003 年的冲击波蠕虫，这个针对 Windows 操作系统 RPC 服务的蠕虫在很短时间内席卷了全球，造成了数百万台机器被感染，损失难以估量。这次事件后，网络运营商们很坚决地在骨干网上屏蔽了 135、445 等端口的连接请求。此次事件之后，整个互联网对于安全的重视达到了一个空前的高度，防火墙对于网络的封锁使得暴露在互联网上的非 Web 服务越来越少。

此时的 Web 服务成为了黑客攻击的受灾区，软件安全进入第二阶段，Web 安全成为了这一阶段的重点研究对象。最开始人们更多的是关注服务器动态脚本安全问题，比如将一个可执行脚本上传到服务器以获得权限。之后，SQL 注入和 XSS 跨站脚本攻击的出现又是两个里程碑事件，这两种攻击至今还在 Web 攻击排行榜的前列。至如今，Web 攻击的思路也从服务器端转向了客户端，转向了浏览器和用户。黑客们天马行空的思路，覆盖了 Web 的每一个环节，变得更加多样化，也促使着防御技术不断发展。

随着 5G 时代的到来，软件安全现阶段将物联网的软件安全纳入研究范围。物联网设备数量呈指数增长，而且，这些物联网设备算力低、协议栈较为脆弱，缺少安全软硬件的支持，很容易成为黑客的攻击目标。由于物联网设备的特殊性，黑客们创造出了许多新型的攻击手段如物理破坏、拥塞攻击、耗尽攻击、Sinkhole 攻击、Sybil 攻击、Wormholes 攻击等等。目前，物联网中的软件安全主要从机密性、完整性、实时性、可用性、鲁棒性和访问控制六个方面考虑，对这些方面进行专项加固是保证物联网设备软件安全的主要方法。

软件安全三个发展阶段的研究对象虽然不同，但攻击技术和防御手段有一定的共通之处，本书将提纲挈领地从所有攻击目标都涉及到的软件漏洞和恶意代码分析角度出发介绍软件存在的安全问题以及如何保证软件的安全。

1.1.2 软件不安全的表现

软件不安全有非常多的表现，用户在使用过程中感知最明显的是软件在运行过程中不稳定，出现异常现象、得不到正常运行结果、或者在特殊情况下由于一些原因造成系统崩溃，这一般是由于异常、网络连接、优化处理不当造成的。也有一种软件安全问题用户是完全感知不到的，如敌方利用各种方式攻击软件，达到窃取信息、破坏系统等目的。下面是生活中一些软件不安全的典型场景。

（1）使用某些交易软件的过程中，某些敏感信息，如个人身份信息、个人卡号密码等信息被攻击方获取并用于牟利。

（2）访问某些网站时，服务器响应很慢，或者服务器由于访问量巨大造成负载过大，造成突然瘫痪。

（3）系统中安装了具有漏洞的软件，漏洞没有修复，攻击方找到漏洞并对本机进行攻击，造成系统瘫痪。

（4）花费精力完成了一幅漂亮的风景画，未经保护地上传到互联网，被他人随意使用却无法问责。

软件的安全问题可能会引起欺骗、篡改、否认、信息泄露、拒绝服务、特权提升、知识产权受侵七类威胁。而大多数安全问题是软件在运行过程中发生的，而负责软件系统运行的技术管理人员或者软件的个人用户，往往不是专业的软件开发人员。此时他们往往无法给出直接的应对方案，虽然可以依靠一些简单的方法，如：优化操作系统、优化网络、优化数据库管理系统或者设置额外的操作权限来对付这些剧增的安全问题，但是实际上，这些方法都是治标不治本的方法。为了应对这软件安全问题，软件的生产单位就需要投入大量的成本，进行软件维护。

1.1.3 软件不安全的原因

那么这些软件不安全的根本原因在哪呢？软件出现安全问题并造成损失，一方面是由于攻击者的猖獗，但是从开发者角度出发，几乎都有一个共同的原因：软件在设计、编码、测试和运行阶段，没有发现软件中的各种安全隐患，导致软件的不安全。软件安全隐患一般可以分为两类：错误和缺陷。

错误是指软件实现过程出现的问题，大多数的错误可以很容易发现并修复，如缓冲区溢出、死锁、不安全的系统调用、不完整的输入检测机制和不完善的数据保护措施等。

缺陷是一个更深层次的问题，它往往产生于设计阶段并在代码中实例化且难以发现，如设计期间的功能划分问题等，这种问题带来的危害更大，但是不属于编程的范畴。

从软件的开发者主观的角度出发，软件不安全的原因可以归纳为以下几种：

（1）软件的生产没有严格遵守软件工程流程

由于缺乏经验或者蓄意(如片面追求高进度)的原因，软件的设计者和开发者们没有统一的权限管理，开发者可以在软件开发周期的任意时候随意删除、新增或者修改软件需求规格说明书、威胁模型、设计文档、源代码、整合框架、测试用例和测试结果、安装配置说明书，使得软件的安全性保证大大减弱。

（2）软件设计的复杂性过高

大多数系统软件或其他商业软件，结构都相当庞大并且复杂，而且由于考虑到软件的扩展性，它们的设计更加巧妙，复杂性也会有所提高。在运行的过程中，这些系统又可以在大量不同的状态之间转换，这个特性使得程序正常运行本身就是一件很困难的事情，更不用说持续安全运行了。

面对不可避免的安全威胁和风险，项目经理和软件工程师必须从开发流程做起，让安全性贯穿整个软件开发的始终。就大多数相对成功的软件工程案例而言，如果项目经理和软件工程师针对软件缺陷进行系统的训练，可以避免软件的许多安全缺陷。

（3）编码者没有采用科学的编码方法。

在软件开发的过程中没有考虑软件可能出现的问题，仅仅将能够想到的问题停留在实验室内进行解决。实际上，有些程序，在实验室阶段根本不会出现安全隐患，如代码 1，在实验室阶段无法预测到用户输入的长度，而到生产场景下，攻击者可以精心设计输入制造出缓冲区溢出问题。

代码 1 隐患代码示例

```
int main(int argc, char* argv[])
   {
   unsigned short total = strlen(argv[1]) + strlen(argv[2]) + 1;
   char* buffer = (char*)malloc(total);
   strcpy(buffer, argv[1]);
   strcat(buffer, argv[2]);
   free(buffer);
   return 0;
}
```

（4）测试不到位

主要是测试用例的设计无法涵盖尽可能典型的安全问题。如图 1.1 的登录表单:

图 1.1 登录表单

一般测试用例只是设计输入正确的用户名和密码，验证能否正常登录；或输入错误的用户名和密码，并验证能否得到相应的错误提示。但是攻击者如果输入和 SQL 注入有关的值，就有可能在无需正确用户名和密码的情况下登录到系统，甚至知道系统中的其他信息或对系统中的内容进行修改。

从软件工程客观角度出发，软件的安全性隐患又来源于以下几个方面：

（1）软件复杂性和工程进度的平衡

如前所述，软件规模越复杂，不仅仅是编码工作量的提高，更重要的是其中需要考虑的问题更加复杂，测试用例规模也呈指数级增长。但是工程进度只是按照软件规模进行适当的延长，因此很多问题来不及解决，软件带着缺陷投入使用。

（2）安全问题的不可见性

软件工程师对运行的实际情况的不了解，在测试时做出过于简单的假设。有些问题，包括对软件的功能、输出和软件运行环境的行为状态，或者外部实体(用户，软件进程)的预期输入，都无法完全考虑到。而攻击者有足够的时间进行攻击方法的研究。

（3）软件需求的变动

软件规格说明书或设计文档无法在开始阶段确定下来。在现代软件工程中，很多软件的需求变动导致其设计本来就是变动的，很多安全问题可能在变动的过程中被忽略。

（4）软件组件之间的交互的不可预见性

客户可能在运行软件的过程中，自行安装第三方提供的组件，开发者无法知道客户的软件将要和谁交互，软件在运行的过程中出现安全问题也就变得无法预测。

因此，不管采用什么样的措施，软件的安全问题都无法完全避免。即使在需求分析和设计时（如通过形式化方法）和在开发时（如通过全面的代码审查和测试）可以避免，但缺陷还是会在软件汇编、集成、部署和运行的时候被引入。不管如何忠实地遵守一个基于安全的开发过程，只要软件的规模和复杂性继续增长，一些可被挖掘出来的错误和其他的缺陷是肯定存在的。这也是为什么说软件都是不安全的，所能做的工作就是尽量让安全问题变少，而不能完全消灭安全问题。

1.1.4 软件安全的挑战与解决方案

现如今软件安全所面临的挑战越来越艰巨，一方面是软件本身变得复杂了，安全问题也表现得很复杂，无法得到全面的考虑，而工程进度又迫使开发者不得不在一定时间内交付产品，长此以往，代码越多，漏洞和缺陷也越多。另一方面，软件的可扩展性要求越来越高，

系统升级和性能扩展成为很多软件必备的功能，可扩展的系统能够用较少的成本实现功能扩展，因而收到开发者和用户的欢迎。但是由于针对可扩展性必须具备相应的设计，软件结构会变复杂了，另加的新功能也引入了新的风险。

为了尽可能减少安全问题并在出现问题时减轻他的影响，研究人员从不同角度提出了许多软件安全防护手段。

（1）安全设计与开发

从强化软件工程的思想出发，将安全问题融入到软件的开发管理流程之中，在软件开发阶段尽量减少软件的缺陷和漏洞的数量，其中较为典型的是微软提出的信息技术安全开发流程（Secure Development Lifecycle for Information Technology，SDL-IT），如图 1.2 所示，该流程包含有一系列的最佳实践和工具，并已经用于微软内部业务应用以及许多微软客户的开发项目中。

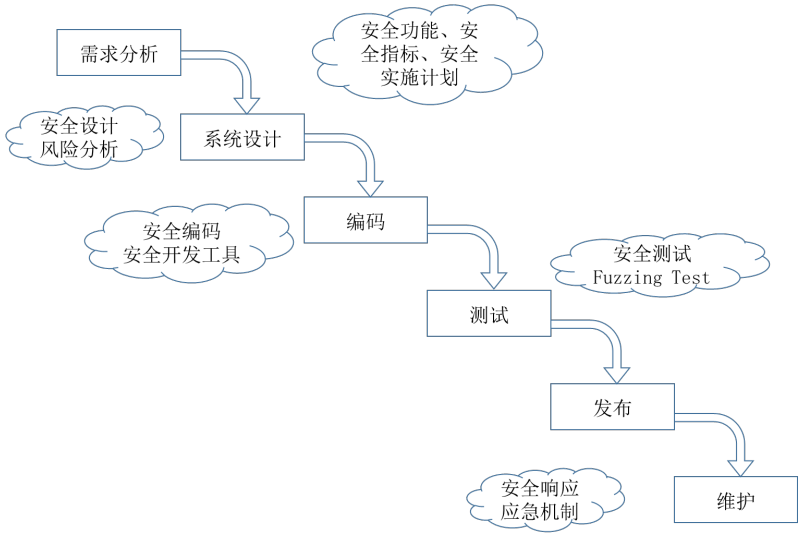

图 1.2 SDL 安全开发流程

（2）保障运行环境

保障软件自身运行环境，加强系统自身数据完整性校验。对于软件完整性的校验，目前很多安全软件在安装之初将对系统的重要文件进行完整性校验并保存其校验值如卡巴斯基安全套件。而对于系统完整性的校验，目前有些硬件系统从底层开始保障系统的完整性，可信计算就是其中的典型代表，TCG 的可信计算信任链的传递如图 1.3 所示。

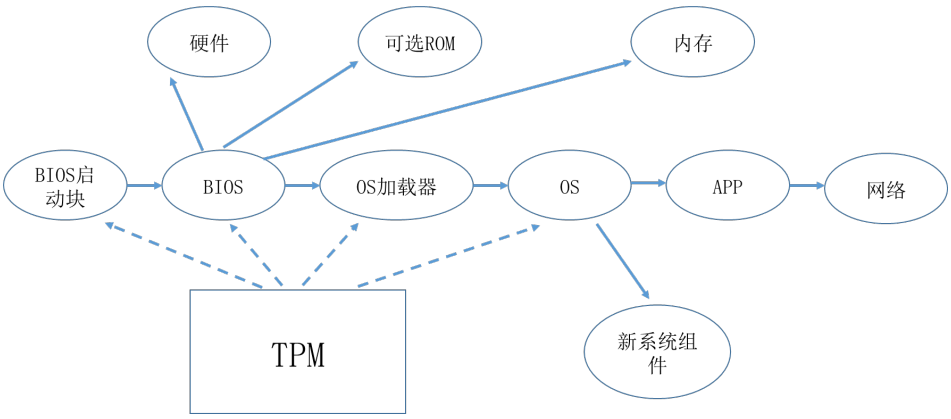

图 1.3 可信计算信任链的传递

（3）软件自身行为认证

在确保软件数据完整性的前提下，如何确保软件的行为总是以预期的方式朝着预期的目标运行，即将可信的思想从静态可信推广到动态可信，如图 1.4 所示

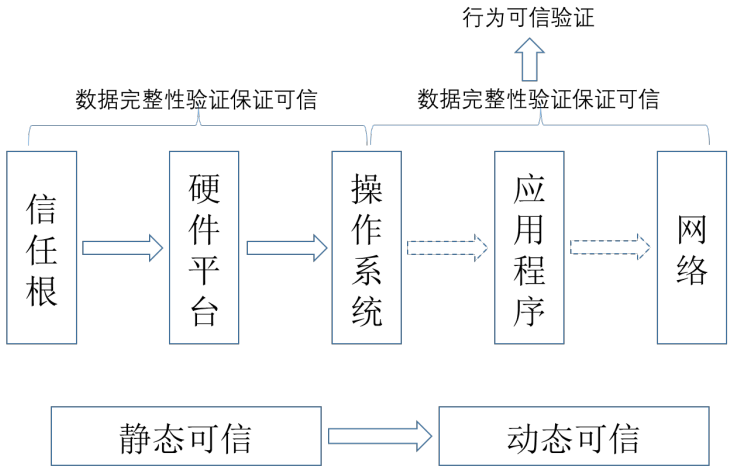

图 1.4 行为可信认证

（4）恶意软件检测与查杀

反病毒软件主要用来对外来的恶意软件进行检测，通常采用病毒特征值检测、虚拟机、启发式扫描、主动防御、云查杀等等集中方法来对病毒进行检测。比较有名的杀毒软件如卡巴斯基、360、Windows Defender 等等。

（5）黑客攻击防护

对于黑客通过网络发动的攻击，可以通过防火墙、入侵检测系统、入侵保护系统等手段进行防护。载入基于主机的漏洞攻击阻断技术也可以用于防止这类攻击。

（6）系统还原

为了防止恶意软件的行为并将他删除，系统备份不失为一个很好的方法，将关键的系统文件或指定磁盘分区还原为预备份状态，从而将已有系统中的恶意程序全部清除，以保护系

统安全。典型的实现例如 Windows 自带的“系统还原”功能、还原卡、影子系统等。

（7）虚拟隔离

这是指将恶意软件放在虚拟机或沙箱中运行，用户可以通过在不同虚拟机或沙箱中分别进行相关活动，从而将危险隔离在不同的系统范围之内，保障敏感性行为的安全性。

除了以上提到的措施外，还有很多正在实验中的方法如软件行为审计、冗余软件机制、拟态软件等等。总的来说，现有的方法只能在一定程度上起到查杀和减轻恶意软件影响的作用，但是效果远远没有达到预期目标，对于软件安全的研究任重而道远。

1.2 软件安全相关技术

软件安全涉及操作系统、汇编语言、数据结构等多门基础课程，为了能够让零基础的读者有更好的阅读体验，本书在此节对软件安全研究所需了解的相关技术作简要介绍。

1.2.1 硬盘与文件系统基本知识

硬盘是当下计算机最主要的存储设备，大多数人了解硬盘更多是关注它的 IO 性能，而对它的底层知识不太重视，实际上，针对硬盘以及上层文件系统的攻击并不少见，许多勒索病毒的攻击目标就是硬盘的固件和文件系统的磁盘分区，因而硬盘和文件系统也是软件安全需要学习的内容。

1. 机械硬盘

机械硬盘的物理结构由盘片、磁头、马达、磁盘臂等构成，如图 1.5 所示

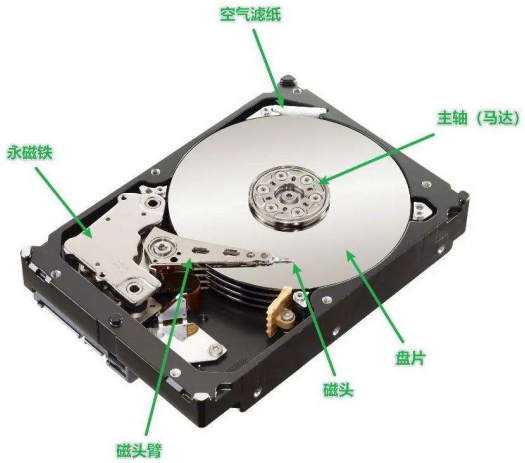

图 1.5 生活中的硬盘

硬盘中一般会有多个盘片组成，每个盘片包含两个面，每个盘面都对应地有一个读/写磁头。受到硬盘整体体积和生产成本的限制，盘片数量都受到限制，一般都在 5 片以内。每

个盘面都被划分为数目相等的磁道，并从外缘的“0”开始编号，具有相同编号的磁道形成一个圆柱，称之为磁盘的柱面。磁盘的柱面数与一个盘面上的磁道数是相等的。由于每个盘面都有自己的磁头，因此，盘面数等于总的磁头数。

所有盘面上的同一磁道构成一个圆柱，称作柱面，即图 1.6 中黄色磁道所组成的面。数据的读/写按柱面从外向内进行，而不是按盘面进行。定位时，首先确定柱面，再确定盘面，然后确定扇区。之后所有磁头一起定位到指定柱面，再旋转盘面使指定扇区位于磁头之下。写数据时，当前柱面的当前磁道写满后，开始在当前柱面的下一个磁道写入，只有当前柱面全部写满后，才将磁头移动到下一个柱面。在对硬盘分区时，各个分区也是以柱面为单位划分的，即从一个柱面到另一个柱面，不存在一个柱面同属于多个分区。

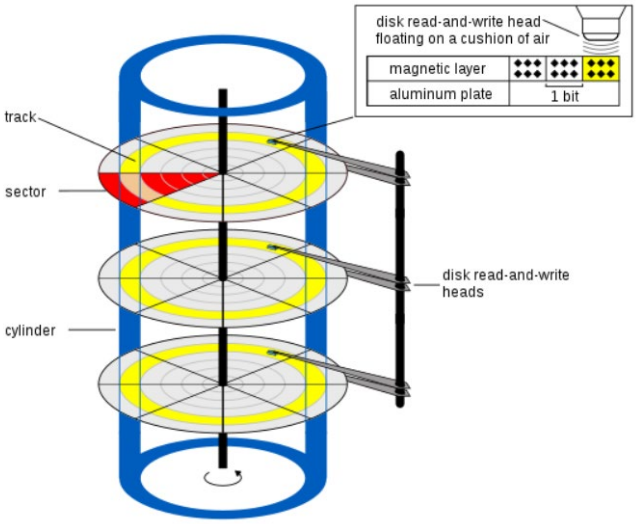

图 1.6 硬盘物理结构

盘片的一面称为盘面，盘面是存储数据的地方，如图 1.7 所示

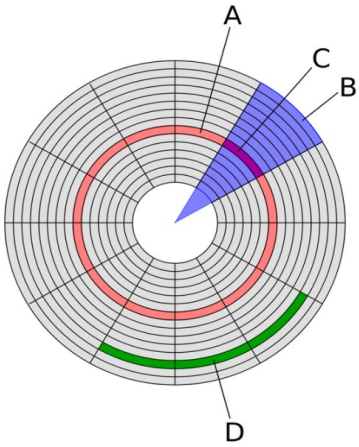

图 1.7 扇面与扇区

图 1.7 中，A 是磁道，为一个同心圆；B 为几何学中的扇形；C 为 B 与 A 交汇部分称为

扇区；D 为块/簇；在计算机磁盘存储中，扇区是磁盘或光盘上磁道的细分。每个扇区存储固定数量的用户可访问数据，传统硬盘上的扇区大小为 512 字节，CD-ROM 和 DVD-ROM 的扇区大小为 2048 字节。较新的硬盘使用 4096 字节（4 KiB）扇区，这些扇区称为高级格式（AF）。扇区是硬盘的最小存储单元。大多数磁盘分区方案旨在使文件占据整数个扇区，而不管文件的实际大小如何。未填充完整个扇区的文件将最后一个扇区的其余部分填充零。在现代磁盘驱动器中，每个物理扇区都由两个基本部分组成，即扇区头区域（通常称为“ ID”）和数据区域。扇区头包含驱动器和控制器使用的信息。该信息包括同步字节，地址标识，缺陷标志以及错误检测和纠正信息。如果数据区域不可靠，则标头还可以包含要使用的备用地址。地址标识用于确保驱动器的机械手将读/写头定位在正确的位置上。数据区域包含同步字节，用户数据和纠错码（ECC），用于检查并可能纠正可能已引入数据中的错误。

实际上，操作系统通常使用数据块操作，数据块可跨越多个扇区。块（Block）/簇（Cluster）是逻辑上的概念，或者说是虚拟出来的概念。分别对应 Linux 与 Windows 操作系统中的概念。磁盘的最小单位就是扇区，那么为什么操作系统不用扇区作为 IO 的基本单位呢？为什么操作系统一定要整出块（Block）/簇（Cluster）这样的概念呢？主要是因为下面两个原因，一是为了读取方便，由于扇区的大小比较小，数目众多时寻址时比较困难，所以操作系统就将相邻的扇区组合在一起，形成一个块，再对块进行整体的操作。二是为了分离对底层的依赖，操作系统忽略对底层物理存储结构的设计，通过虚拟出来磁盘块的概念，在系统中认为块是最小的单位。

2. 固态硬盘

现代微型计算机的存储单元不再是由一个机械硬盘构成，而是采用了固态硬盘和机械硬盘相结合的方式，这是由于固态硬盘拥有更快的读写速度，但由于造价过高暂时无法完全取代机械硬盘。

固态硬盘(Solid State Drive)，简称 SSD（固盘），是用固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元(FLASH 芯片、DRAM 芯片)以及缓存单元组成。区别于机械硬盘由磁盘、磁头等机械部件构成，固态硬盘结构机械装置，全部是由电子芯片及电路板组成。固态硬盘的内部结构，由主控芯片、闪存颗粒、缓存单元构成。

正如同 CPU 之于 PC 一样，主控芯片其实也和 CPU 一样，是整个固态硬盘的核心器件，其作用一是合理调配数据在各个闪存芯片上的负荷，二则是承担了整个数据中转，连接闪存

芯片和外部 SATA 接口。不同的主控芯片之间的数据处理能力相差非常大，在数据处理能力、算法上，对闪存芯片的读取写入控制上会有非常大的不同，直接会导致固态硬盘产品在性能上产生很大的差距。

作为硬盘，存储单元绝对是核心器件。在固态硬盘里面，闪存颗粒则替代了机械磁盘成为了存储单元。闪存本质上是一种长寿命的非易失性(在断电情况下仍能保持所存储的数据信息)存储器，数据删除不是以单个的字节为单位而是以固定的区块为单位。在固态硬盘中，NAND 闪存因其具有非易失性存储的特性，即断电后仍能保存数据，被大范围运用。根据 NAND 闪存中电子单元密度的差异，又可以分为 SLC（单层次存储单元）、MLC（双层存储单元）以及 TLC(三层存储单元)，此三种存储单元在寿命以及造价上有着明显的区别。由于闪存颗粒是固态硬盘中的核心器件，也是主要的存储单元，因而它的制造成本占据了整个产品的 70%以上的比重。在某种程度上，选择固态硬盘实际就是在选择闪存颗粒。

缓存芯片，是固态硬盘三大件中，最容易被人忽视的一块，也是厂商最不愿意投入研发成本的一块。和主控芯片、闪存颗粒相比，缓存芯片的作用确实没有那么明显，在用户群体的认知度也没有那么深入，相应的就无法以此为噱头进行卖点宣传。实际上，缓存芯片的存在意义还是较为明显的，特别是在进行常用文件的随机性读写上，以及碎片文件的快速读写上。由于固态硬盘内部的磨损机制，就导致固态硬盘在读写小文件和常用文件时，会不断进行数据的整块的写入缓存，然而导出到闪存颗粒，这个过程需要大量缓存维系。特别是在进行大数量级的碎片文件的读写进程，高缓存的作用更是明显。

固态硬盘的逻辑结构与机械硬盘大不相同，Flash 的基本存储单元是浮栅晶体管，根据制造工艺分为 NOR 和 NAND 型。NAND 容量大，按照 page 进行读写，适合进行数据存储，基本存储使用的 SSD 的 Flash 都是 NAND。多个闪存构成如图 1.8 的逻辑结构图。

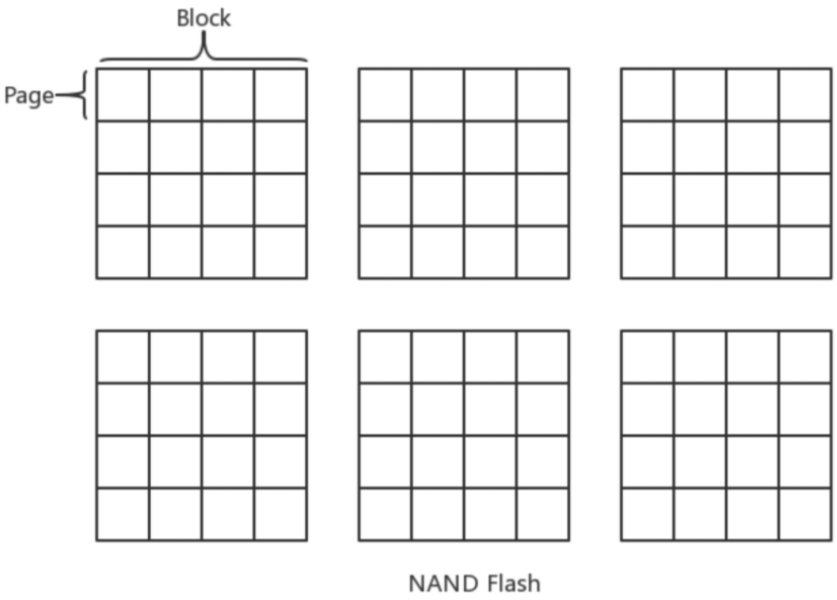

图 1.8 闪存结构图

SSD 中一般有多个 NAND-Flash，每个 NAND-Flash 包含多个 Block，每个 Block 包含多个 Page。由于 NAND 的特性，存取都必须以 Page 为单位，即每次读写至少是一个 Page，通常地，每个 Page 的大小为 4K 或者 8K。因而，对于固态硬盘来说已经没有了扇区的概念，尽管有的文件系统仍然以 512 字节为一个扇区，但交给主控芯片去读写数据时是以 4K 或者 8K 为单位进行操作的。由于固态硬盘与机械硬盘逻辑组织的不同，因而出现了很多新的概念如 GC 机制、Trim 机制、Bit-Error、Read-Disturb、Program-Disturb 等等，在此不再详加赘述。

现在市面上主流的固态硬盘接口有 SATA、mSATA、m.2、PCI-E 插槽四种，协议方面，SSD 出现之初绝大多数走的是 AHCI 和 SATA 的协议，后者其实是为传统 HDD 服务的。与 HDD 相比，SSD 具有更低的延时和更高的性能，AHCI 已经不能跟上 SSD 性能发展的步伐了，已经成为制约 SSD 性能的瓶颈。随之出现了为 SSD 而生的 NVMe 协议，同样是使用 PCI-E×4 的通道，不支持 NVMe 协议的硬盘最大只能达到 1500MB/s，而支持 NVMe 协议的硬盘就可以达到 3000MB/s 甚至以上。

3. 文件系统

仅仅抽象到块/簇这一层级是远远不够的，试想，当新建一个文件时，需要输入存储在哪一个块中，这将是一个非常繁琐的工作，因而工程师们设计出了文件系统。文件系统是操作系统用于明确存储设备或分区上的文件的方法和数据结构，即在存储设备上组织文件的方

法。操作系统中负责管理和存储文件信息的软件机构称为文件管理系统，简称文件系统。文件系统由三部分组成：文件系统的接口，对对象操纵和管理的软件集合，对象及属性。从系统角度来看，文件系统是对文件存储设备的空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统。具体地说，它负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等。现在主流的文件系统有如 FAT32、NTFS、EXT3/4、NFS、XFS 等等，本文选取 U 盘主流的文件系统 FAT32、Windows 主流的文件系统 NTFS 以及 Linux 系统中主流的文件系统 EXT4 作介绍。

（1）FAT32

FAT32 指的是文件分配表是采用 32 位二进制数记录管理的磁盘文件管理方式，因 FAT 类文件系统的核心是文件分配表，命名由此得来。FAT32 是从 FAT 和 FAT16 发展而来的，优点是稳定性和兼容性好。缺点是安全性差，且最大只能支持 2TB 分区，单个文件也只能支持最大 4GB。

FAT32 文件系统由 DBR，DBR 保留扇区、FAT1、FAT2 和 DATA 四个部分组成，图 1.9 为 FAT32 文件系统的组成图。

图 1.9 FAT32 系统组成图

FAT32 文件系统中各个部分的功能如图 1.10 所示，MBR 主要存储磁盘分区相关的信息，如分区开始磁头、结束柱面和扇区、分区总扇区数目等等，如果存在 MBR 的话，那么它占用第一个扇区的 512 个字节。DBR 是格式化分区时创建的，每个分区都会有 DBR 信息，1 个 DBR 只能定义 1 个分区的系统文件,占用 512 个字节，其中包含了如 FAT 表数、磁道扇区数、磁头数、第一个目录的簇号等等。

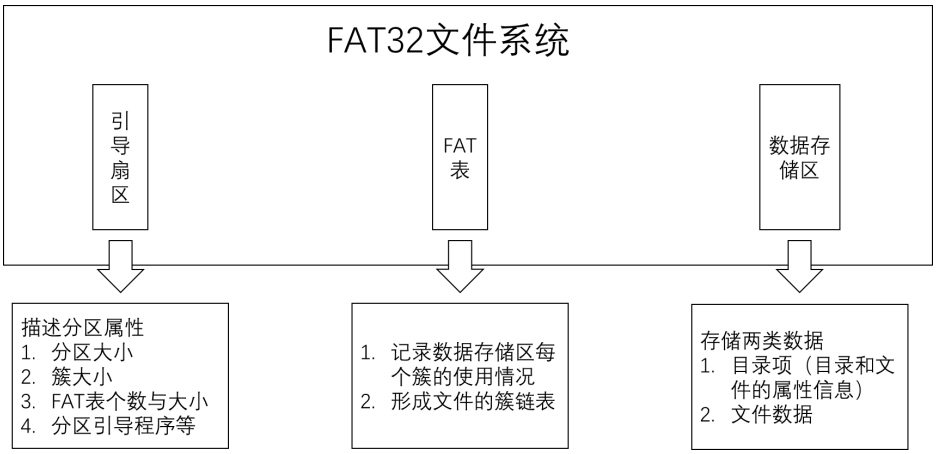

图 1.10 FAT32 系统组成部分的功能

文件系统分配磁盘空间按簇来分配，因此，文件占有磁盘空间时，基本单位不是字节而是簇，即使某个文件只有一个字节，操作系统也会给它分配一个最小单元：即一个簇。对于大文件，需要分配多个簇。同一个文件的数据并不一定完整地存放在磁盘中一个连续地区域内，而往往会分若干段，像链子一样存放。这种存储方式称为文件的链式存储。为了实现文件的链式存储，文件系统必须准确地记录哪些簇已经被文件占用，还必须为每个已经占用的簇指明存储后继的下一个簇的簇号，对于文件的最后一簇，则要指明本簇无后继簇。这些都是由 FAT 表来保存的，FAT 表对应表项中记录着它所代表的簇的有关信息：诸如是空，是不是坏簇，是否是已经是某个文件的尾簇等。而第二个 FAT 表是第一个 FAT 表的备份。

数据区是真正用于存放用户数据的区域，数据区在第二个 FAT 表后面，以簇为基本单位进行划分，主要由三部分组成：根目录、子目录和文件内容，分区根目录下的文件和目录都放在根目录区中，子目录中的文件和目录都都放在子目录区中，并且每 32 个字节为一个目录项(FDT)，每个目录项记录着一个目录或文件。一个目录项中包含了文件名、创建时间、开始簇、文件大小等等信息。

下面讲解在 FAT32 文件系统中如何找到具体的文件数据，首先，通过 MBR 中的分区表信息得到分区的起始位置；然后，通过分区中的 DBR 得知 DBR 的保留扇区数以及 FAT 表的大小、FAT 表的个数，接着通过上面信息找到数据区的起始位置，然后根据用户输入的路径从根目录到子目录递归查找，直至找到目标文件的目录项，然后根据目录项中记录的起始簇到 FAT 表中找后续的链接簇，再从链接簇中把所有的文件数据提取出来。同理，文件创建也是按照上面的步骤进行的，如图 1.11 所示。

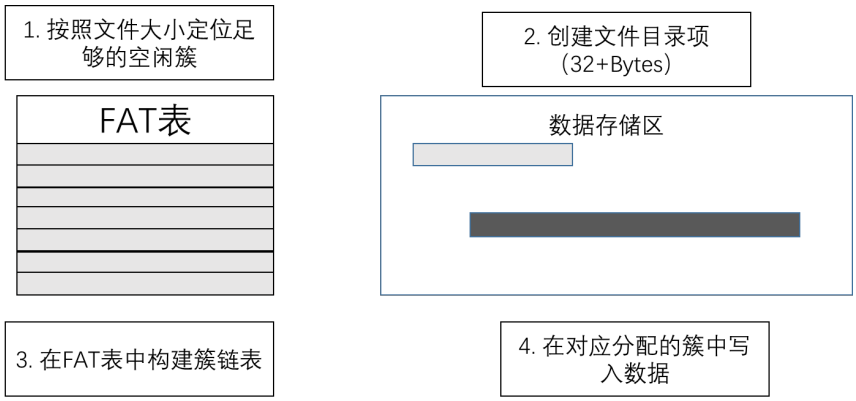

图 1.11 文件创建的步骤

FAT32 文件系统删除文件的操作也值得一提，文件系统会将文件目录项中文件名的首字节修改为 E5，首簇高位的两个字节清零，文件大小不变。存于 FAT 表中的文件簇链会被全部清空，文件内容不变。这也给恢复被删除的文件提供了思路，核心难点就在于恢复首簇的高位以及恢复 FAT 表中的簇链。

（2）NTFS

NTFS，是一个很强大的文件系统，支持的功能很多，存储的原理也很复杂。目前绝大多数 Windows 用户都是使用 NTFS 文件系统，它主要以安全性和稳定性而闻名，下面是它的一些主要特点。安全性高：NTFS 支持基于文件或目录的 ACL，并且支持加密文件系统（EFS）。可恢复性：NTFS 支持基于原子事务概念的文件恢复，比较符合服务器文件系统的要求。文件压缩：NTFS 支持基于文件或目录的文件压缩，可以很方便的节省磁盘空间。磁盘配额：NTFS 支持磁盘配额，可针对系统中每个用户分配磁盘资源。

NTFS 文件系统结构如图 1.12 所示，它主要包括了 16 个元文件，其中$BOOT 为引导文件，记录了用于系统引导的数据情况；$MFT 是主文件表本身，是每个文件的索引；$MFTMIRR 为主文件表的部分镜像；备份 DBR 存储着文件系统的一些基本信息，如扇区数、MFT 起始地址、每个索引块包含的簇大小等等。

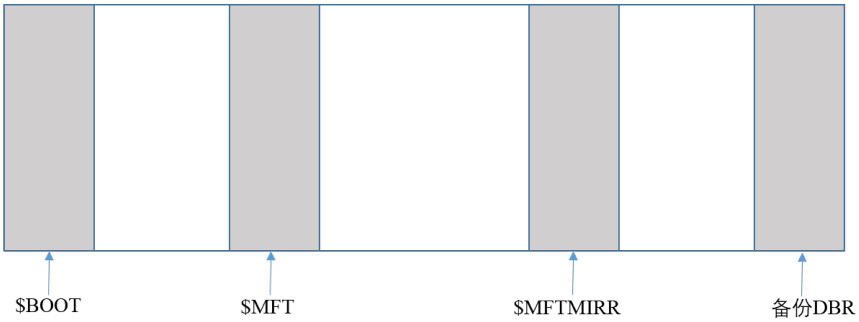

图 1.12 NTFS 文件系统组织图

在了解 NTFS 的基本结构后，下面介绍 NTFS 文件系统中寻找文件的基本步骤了。首先

定位 DBR，通过 DBR 可以得知“$MFT”的起始簇号及簇的大小。接着定位“MFT”，找到“MFT”后，在其中寻找根目录的文件记录，一般在 5 号文件记录。然后 5 号文件记录中的 90H 属性中得到 B+树索引的根节点文件信息，重点在 A0 属性上。通过属性中的“Run List”定位到其数据流。从“Run List”定位到起始簇后，再分析索引项可以得到文件名等信息。从索引项中可以获取“MFT”的参考号，然后进入到“MFT”的参考号，然后进入到“MFT”找到对应的文件记录。然后再根据 80H 属性中的数据流就可以找到文件真正的数据了。

（3）EXT4

EXT4 是第四代扩展文件系统是 Linux 系统下的日志文件系统，是 EXT3 文件系统的后继版本。EXT4 是由 EXT3 的维护者 Theodore Tso 领导的开发团队实现的，并引入到 Linux2.6.19 内核中。

谈到 Linux 的文件系统就不得不提到虚拟文件系统，如图 1.13 所示。Linux 设计了一个文件系统的中间层，上层用户都直接和 VFS 打交道，文件系统开发者再把 VFS 转换为自己的格式，这样的做有很多优点，用户层应用不用关心具体用的是什么文件系统，使用统一的标准接口进行文件操作；如果一个系统包含不同分区，不同分区使用不同的文件系统，他们之间可以通过这个 VFS 交互，比如从 U 盘、网盘拷数据到硬盘就得通过 VFS 转换管理信息，从而可以动态支持很多文件系统，添加一个文件系统只需要安装对应驱动即可，无需重新编译内核。

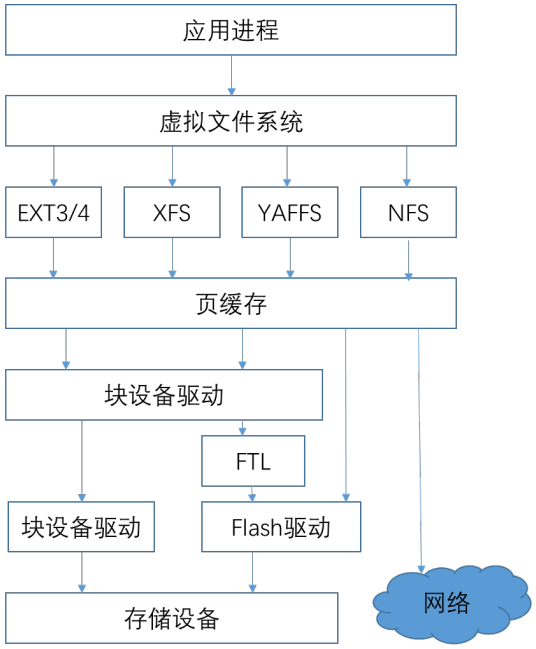

图 1.13 VFS 架构图

EXT4 文件系统的磁盘结构如图 1.14 所示，引导块为磁盘分区的第一个块，记录文件系

统分区的一些信息，引导加载当前分区的程序和数据被保存在这个块中。一般占用 2KB。超级块用于存储文件系统全局的配置参数(如块大小，总的块数和 inode 数)和动态信息(如当前空闲块数和 inode 数)。GDT 用于存储块组描述符，其占用一个或者多个数据块，具体取决于文件系统的大小。它主要包含块位图，inode 位图和 inode 表位置，当前空闲块数，inode 数以及使用的目录数(用于平衡各个块组目录数)。

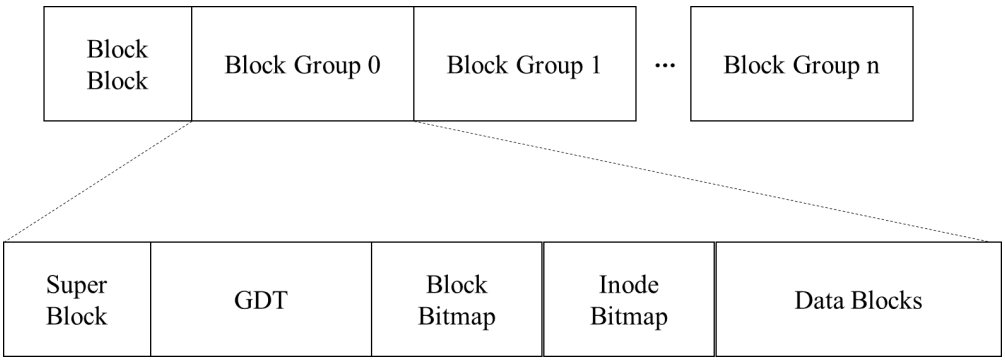

图 1.14 EXT4 结构

EXT 系列文件系统最重要的思想即文件信息和文件数据分离的思想。Inode 记录文件的信息如文件名、权限、拥有者、存储的位置等等，而真正的文件数据存储在其他地方，Inode 中存放着放数据块的索引。而上图中的 BlockBitmap 和 InodeBitmap 分别是 block 位图和 inode 位图，用于记录哪些 inode 和 block 处于空闲状态，这样在创建新文件时可以进行利用。而 inode Table 正是 inode 和 block 的映射表，其映射方式如图 1.15 所示。由于一个文件可能拥有多个块，记录 block 索引的方式相较于 FAT32 链式存储具有随机读写的优势。

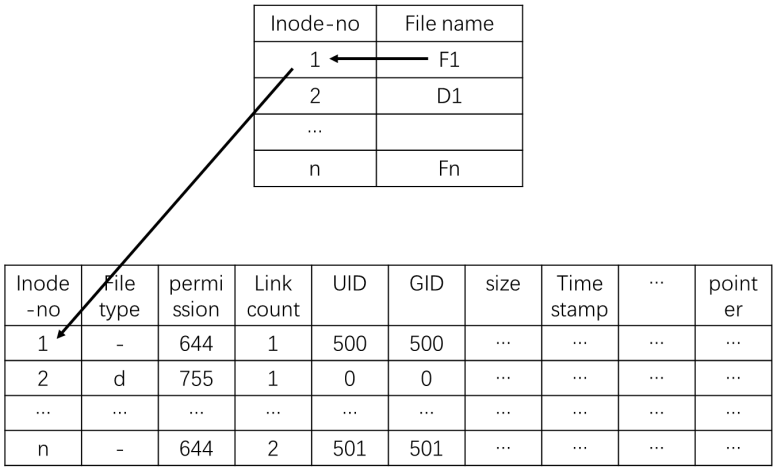

图 1.15 EXT 文件系统映射方式

1.2.2 PE 与 ELF 文件格式

文件可以分为可执行文件、不可执行文件、支持文件、文档文件、图像文件、多媒体文

件、其他文件。每种文件都有自己的格式，本书主要面向软件安全，会重点关注可执行文件。Windows 中的可执行文件为 PE 格式，Linux 中的可执行文件为 ELF 格式，软件安全要求能够根据这两种格式的二进制文件分析出可执行程序的组成结构。两者有很多共同点，所以先详细介绍 PE 文件的结构，最后再简单地介绍 ELF 格式。

1. PE 文件

PE（Portable Execute）文件是 Windows 下可执行文件的总称，常见的扩展名有 DLL，EXE，OCX，SYS 等，事实上一个文件是否是 PE 文件与其扩展名无关，PE 文件可以是任何扩展名。分析 Windows 下的软件实际上就是分析 PE 文件

那么这些信息从何处获取呢，答案是 PE 的文件头，PE 的文件头中会给出有关这个 PE 文件的所有组织信息。如图 1.16 所示

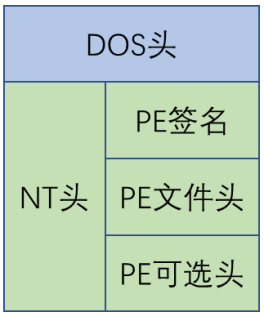

图 1.16 PE 文件头

DOS 头的作用是兼容 MS-DOS 操作系统中的可执行文件，对于 32 位 PE 文件来说，DOS 起的作用为显示一行文字，提示用户：我需要在 32 位 Windows 上才可以运行，可以把这当做一个善意的玩笑。

PE 签名的高 16 位是 0，低 16 位是 0x4550，用字符表示是“PE”，这一部分可以理解成一个标志位。代表这个文件是个 PE 文件。

PE 文件头中包含了文件运行的关键信息，如运行平台（X86/X64/I64 等等），节表的数量，创建时间，符号表数量，可选头的大小，可执行文件属性。

PE 可选头虽然叫可选头，但是一点都不能少，其中包含了可选头类型（32 还是 64 位），链接器版本，代码段长度，初始化的数据长度，未初始化的数据长度，程序入口的 RVA，代码段起始地址的 RVA，数据段起始地址的 RVA，映像基地址，节对齐，操作系统的版本号，映像版本号，子系统版本号，映像大小，映像文件校验和等等。

除去 PE 文件头外，PE 文件的节表能够大体勾勒出文件的组织结构，节表紧挨着 PE

Header，每一个节均对应一个节表项，包含节名、节在文件和内存中的开始地址、长度、节属性等等。节表下就是可执行文件的核心部分节，包含代码节、数据节、引入函数节、资源节、引出函数节、重定位节。

.text 节：代码段（text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且这段内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

.data 节：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

.bss 节：未初始化数据节，存放未初始化的全局变量和静态变量。

.rdata 节：包含从其他 DLL 中引入的函数如 user32.dll，gdi32.dll 等，由 IMPORT Address Table、IMPORT Directory Table、IMPORT Name Table、IMPORT Hints/Names & DLL Names 等内容构成，通过这一节可以找到第三方函数在内存中的位置，具体寻址方式将在第三章以实例介绍。

.edata 节：导出段，包含所有提供给其他程序使用的函数和数据。

.rsrc 节：资源数据段，程序用到什么资源数据都在这里。

.reloc 节：重定位段，如果加载 PE 文件失败，将基于此段进行重新调整。

由于 PE 文件有特定的文件格式，所以读者既可以选择手动分析，也有许多现成的工具用来分析 PE 文件，如 LoadPE，PETOOL，PEView 等等，如图 1.17 就是通过 PEView 得到的分析结果。

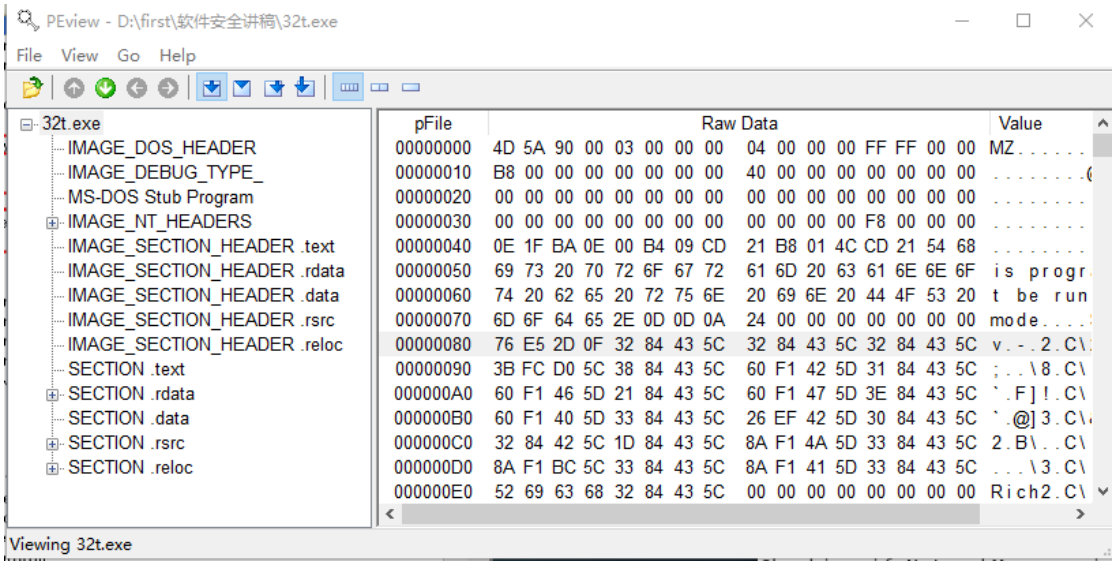

图 1.17 PEview

2. ELF 文件

ELF 文件是 Linux 下的文件格式，其文件的组织与 PE 文件格式大体相似，如图 1.18 所示。ELF 文件格式提供了两种不同的视角，在汇编器和链接器看来，ELF 文件是由 Section Header Table 描述的一系列 Section 的集合，而执行一个 ELF 文件时，在加载器（Loader）看来它是由 Program Header Table 描述的一系列 Segment 的集合。

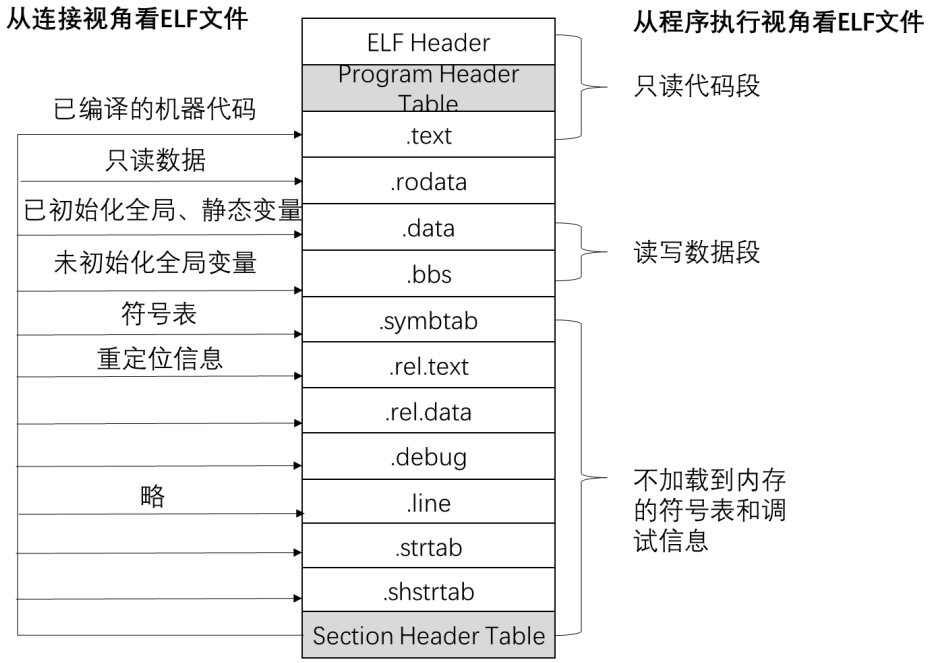

图 1.18 ELF 文件组织结构

左边是从汇编器和链接器的视角来看这个文件，开头的 ELF Header 描述了体系结构和操作系统等基本信息，并指出 Section Header Table 和 Program Header Table 在文件中的什么位置，Program Header Table 在汇编和链接过程中没有用到，Section Header Table 中保存了所有 Section 的描述信息。右边是从加载器的视角来看这个文件，开头是 ELF Header，Program Header Table 中保存了所有 Segment 的描述信息，Section Header Table 在加载过程中没有用到。注意 Section Header Table 和 Program Header Table 并不是一定要位于文件开头和结尾的，其位置由 ELF Header 指出。

在汇编程序中用.section 声明的 Section 会成为目标文件中的 Section，此外汇编器还会自动添加一些 Section（比如符号表）。Segment 是指在程序运行时加载到内存的具有相同属性的区域，由一个或多个 Section 组成，比如有两个 Section 都要求加载到内存后可读可写，就属于同一个 Segment。有些 Section 只对汇编器和链接器有意义，在运行时用不到，也不

需要加载到内存，那么就不属于任何 Segment。

目标文件需要链接器做进一步处理，所以一定有 Section Header Table；可执行文件需要加载运行，所以一定有 Program Header Table；而共享库既要加载运行，又要在加载时做动态链接，所以既有 Section Header Table 又有 Program Header Table。

1.2.3 计算机系统内存管理

内存是软件安全问题的最大受灾区，应用程序均是装载到内存中进行运行的，如若内存出现问题，则可能导致信息泄露、程序崩溃、丧失控制权等等问题，因而内存管理技术是软件安全所必学的内容。

不同的处理器、不同的操作系统为计算机程序分配内存的方式都是不同的，在介绍系统的内存管理前，先介绍处理器的不同模式，此处以主流的 Intel 的 80X86 处理器为例，该类型的处理器主要有三种工作模式，如图 1.19 所示。

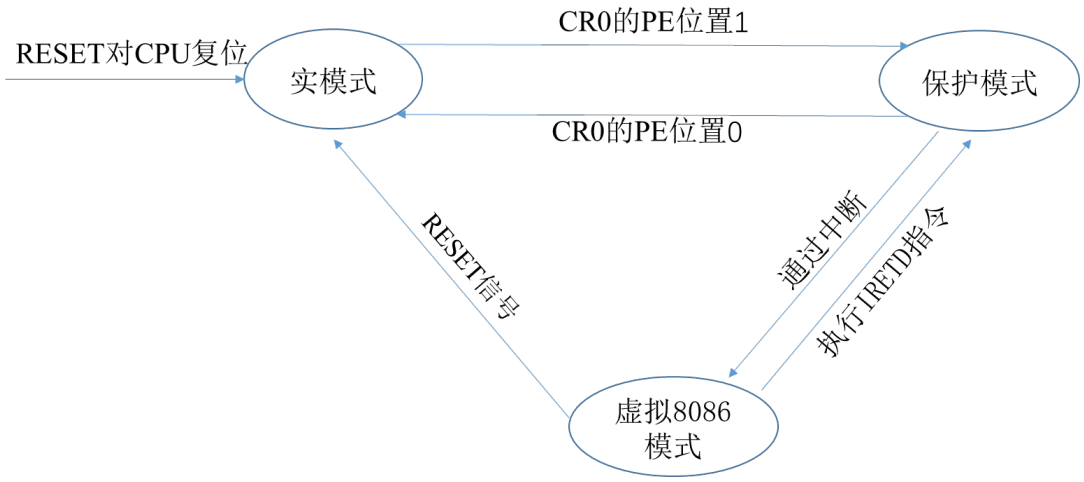

图 1.19 处理器模式转换

实模式：80X86 处理器在复位或者加电时是以实模式启动的，该模式不能对内存进行分页管理，只能进行 20 位的寻址（段+偏移）即 1MB 空间，同时该模式不支持优先级，所有的指令均工作在特权级。

保护模式：保护模式是 80X86 处理器的常态工作模式，32 位处理器支持 32 位寻址即 4G 空间，该模式支持内存分页机制，提供了对虚拟内存的良好支持，且能够根据任务特性进行运行环境隔离。

虚拟 8086 模式：这是为了在保护模式下兼容 8086 程序而设置的，虚拟 8086 模式是以任务形式在保护模式上运行的，在 80X86 上可以同时支持多个真正的 80X86 任务和虚拟 8086 模式构成的任务。

可以看出即使同一种处理器也有不同的工作模式，不同工作模式的寻址能力是不同的，程序能够管理的内存空间自然也是不同的。后文的所有程序都是运行在 Intel 80X86 处理器的保护模式下的。下面也将具体介绍保护模式下的程序内存管理。

对于 32 位环境来说，理论上程序可以拥有 4GB 的虚拟地址空间，在 C 语言中使用到的变量、函数、字符串等对应地存在内存中的一块区域。但是，这 4GB 的地址空间需要拿出一部分给操作系统内核使用，应用程序无法直接访问这一段内存，这一部分内存被称为内核空间。Windows 在默认情况下会将高地址的 2GB 空间分配给内核，而 Linux 默认情况下会将高地址的 1GB 空间分配给内核，如图 1.20 所示。也就是说，应用程序只能使用剩下的 2GB 或 3GB 的地址空间，这一部分空间故称为用户空间。

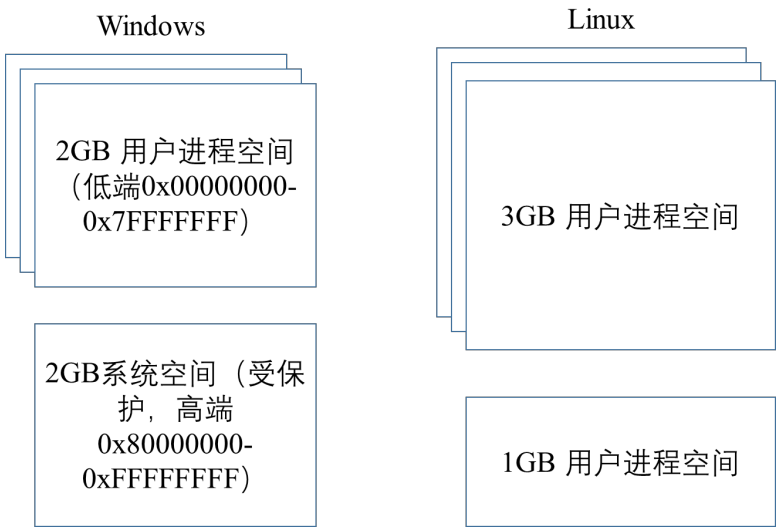

图 1.20 用户态和内核态用户内存分配

用户程序会被装载到内存的用户空间中，而装载顺序即为程序每一部分在文件中的顺序，内存地址从低到高依次为堆栈段、数据段、文本（代码）段，其分布情况如图 1.21 所示。

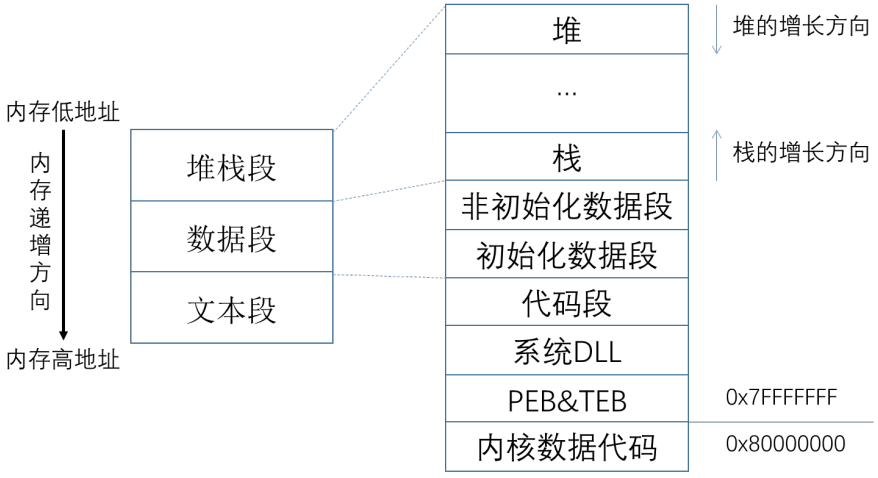

图 1.21 程序加载

与进程虚拟内存中用户模式区和内核模式区相对应，Windows 为了确保系统的稳定性，将处理器存取模式划分为用户模式（Ring 3）和内核模式（Ring 0）。用户应用程序一般运行在用户模式，其访问空间局限于用户区。操作系统内核代码运行在内核模式。可以访问所有的内存空间和硬件，也可以使用所有处理器指令。

用户区是每个进程真正独立可用的内存空间，进程中的绝大部分数据都保存在这一区域，主要包括程序代码、全局变量、所有线程的线程栈以及加载的 DLL 代码等。每个进程的用户区的虚拟内存空间相互独立，一般不可以直接跨进程访问，这使得一个程序直接破坏另一个程序的可能性非常小。

内存内核区中的所有数据是所有进程共享的，是操作系统代码的驻地，其中包括操作系统内核代码以及线程调度、内存管理、文件系统支持、网络支持、设备驱动程序相关的代码。该分区中的所有代码和数据都被操作系统保护，用户模式代码无法直接访问和操作。如果应用程序直接对该内存空间内的地址访问将会发生地址访问违规。

实际上，每个进程可用的 4GB 内存空间为虚拟地址空间，这 4GB 地址空间会被分成固定大小的页，每一页或者被映射到物理内存，或者被映射到硬盘上的交换文件中，或者没有映射任何内容。对于一般的程序来说，4G 的地址空间只有一小部分映射了物理内存，大部分地址空间没有映射任何内容，如图 1.22 所示。虚拟地址到物理地址的转换对于不同操作系统有所不同。

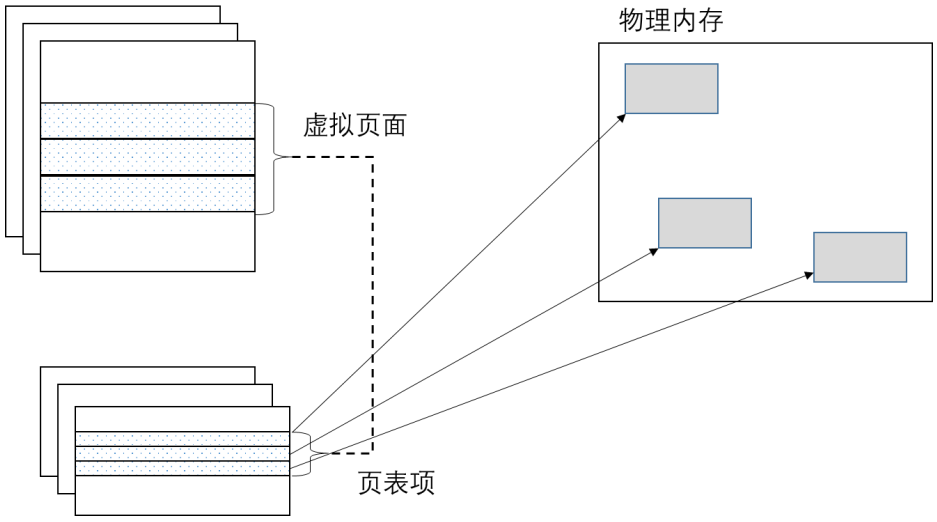

图 1.22 逻辑地址与物理地址的关系

1. Windows 内存管理

Windows 系统采用段页式内存管理，既分段又分页。虚拟地址与物理地址进行映射时，

先由逻辑地址转换为线性地址，再由线性地址转换为物理地址。逻辑地址到线性地址的转换如图 1.23 所示，逻辑地址由 48 位组成，前 16 位是段选择符，后 32 位为段内偏移量。通过段选择符可以从段表中找到段描述符，段描述符存放段基址、段大小、存储权限等等。段描述符存储在段描述符表中，内核程序的段描述符存储在全局描述符表 GDT 中，应用程序的段描述符存储在局域描述符表 LDT 中。段基地址寄存器指向段描述符表所在地址即 GDT 和 LDT 地址。从段描述符中获取到段基址后与逻辑地址后 32 位段内偏移量相加得到 32 位的线性地址。

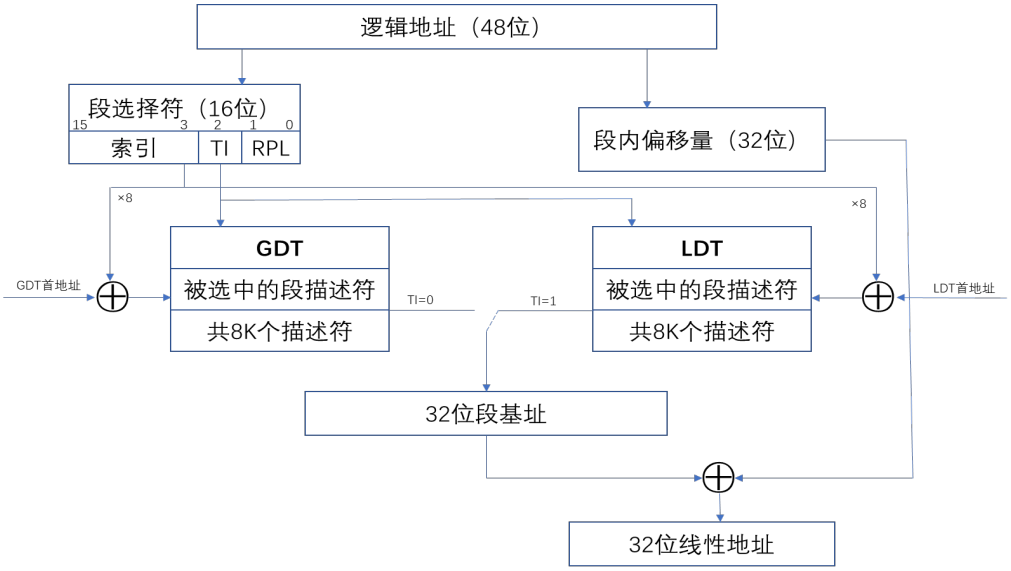

图 1.23 Windows 逻辑地址转换线性地址

线性地址到物理地址的转换是通过二级页表项进行的。在 32 位系统中，物理内存分页和虚拟内存分页的大小都是 4KB，页目录的大小为 4K，存放着 1024 个页表的地址，因而在页目录中寻找页表只需要 10 位。而每个页表的大小也是 4K，存放着 1024 个页表项，页表项即为某一页面的具体物理地址，要分辨这 1024 个页表项只需要 10 位。找到页面后，一个页面大小为 4K，计算字节的偏移需要 12 位，所以一共需要 10+10+12=32 位地址。具体的虚拟地址与物理地址的转换如图 1.24 所示。

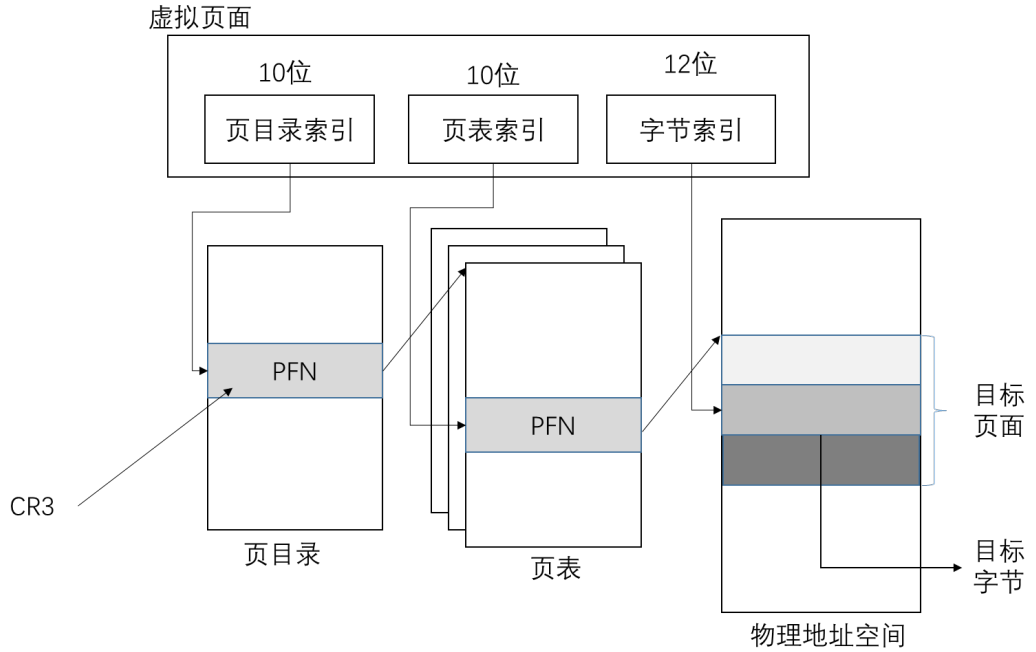

图 1.24 Windows 线性地址转换物理地址

Windows 除了使用真正的物理内存外，还会将硬盘文件虚拟成内存使用（C:\pagefiles.sys），这部分内存也称为虚拟内存，这部分并不是真正的内存，CPU 如果要访问虚拟内存数据必须将虚拟内存放入物理内存中。

Windows 访问内存的过程如图 1.25 所示，第一步根据地址在物理内存中找相应的位置，如果找到物理内存即取回数据，如果未找到已绑定的物理内存，那么进而判断是否和虚拟内存绑定。第二步根据地址在虚拟内存中查找相应的位置，如果未找到，那么该地址没有绑定内存为野指针，返回错误。第三步把上一步找到虚拟内存置换到物理内存中，同时将物理内存中的数据存入到虚拟内存。第四步是把找到的物理内存中的数据返回给使用者。最后一步再把置于物理内存的数据放回虚拟内存，把暂存于虚拟内存的物理内存放回物理内存，恢复环境。

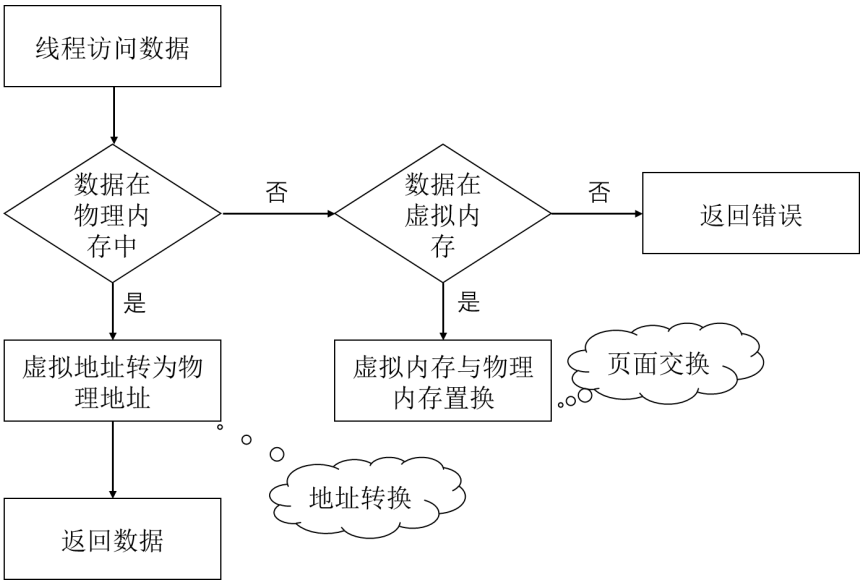

图 1.25 内存访问

内存的分配方式依据申请的内存方式和大小而有所不同，虚拟内存比较适合大内存分配，一般是 1M 以上大小，没有上限，日常使用的大型电子表格，大型数组都是存储在虚拟内存中。物理内存又可以分为不同的区域，如上一节所述的不同的节，其中数据、代码都程序运行之初就分配好的，而堆、栈都是在程序运行过程中才会进行分配。堆比较适合小内存分配，一般是 1M 以下的内存，分配的内存都是连续的，每个程序都有自己的堆，默认为 1M 大小，大容量分配堆内存会造成内存空洞，实际编程中使用的 malloc/new 就是分配的堆内存。栈内存也比较适合小内存分配，一般是 1M 以下的内存，实际是 1M 左右，栈也是由系统管理的，不允许程序员操作，代码中定义的局部变量会被分配到栈中。

2. Linux 内存管理

严格来说 Linux 也采用段页式内存管理，也就是既分段，又分页。地址映射的时候，先确定对应的段，确定段基地址，段内分页，再找到对应的页表项，确定页基地址，再由逻辑地址低位确定的页偏移量，就能找到最终的物理地址。但是， Linux 实际上采用的是页式内存管理。Linux 中的段基地址都是 0，相当于所有的段都是相同的。这样做的原因是某些体系结构的硬件限制，比如 Intel 的 i386。作为软件的操作系统，必须要符合硬件体系。虽然所有段基地址都是 0，但是段的概念在 Linux 内核中是确实存在的。比如常见的内核代码段、内核数据段、用户态代码段、用户态数据段等。因而 Linux 既可以说采用的是段页式内存管理也可以说采用的是页式内存管理。

Linux 在分页机制上也有所不同。作为一个通用的操作系统，Linux 需要兼容各种硬件体

系，包括不同位数的 CPU。对 64 位的 CPU 来说，两级页表仍然太少，一个页表会太大，这会占用太多宝贵的物理内存。因而 Linux 采用了通用的四级页表。实际采用几级页表则具体受硬件的限制。

四种页表分别称为页全局目录、页上级目录、页中间目录、页表。对于 32 位 x86 系统，两级页表已经足够了。Linux 通过使“页上级目录”位和“页中间目录”位全为 0，彻底取消了页上级目录和页中间目录字段。不过，页上级目录和页中间目录在指针序列中的位置被保留，以便同样的代码在 32 位系统和 64 位系统下都能使用。

Linux 和 Windows 系统在内存管理上最大的不同在于内存的使用上。Linux 优先使用物理内存，当物理内存还有空闲时，Linux 不会释放内存，即使占用内存的程序已经被关闭了（这部份内存被用来作缓存）。也就是说，即使存在较大内存,使用一段时间后，也会被占满。这样做的好处是，启动那些刚开启过的程序、或是读取刚存取过得数据会比较快，对于服务器颇有好处。Windows 则总是给内存留下必要的空闲空间，即使内存有空闲也会让程序使用一些虚拟内存，这样作的好处是，启动新的程序比较快，直接分给它部分空闲内存就足够了，而 Linux 下，因为内存常常处于所有被使用的状态，则要先清理出一块内存，再分配给新的程序使用，所以新程序的启动会相对较慢。

在 Linux 中，前面处理器的分页机制只是建立的从虚拟地址到物理地址的映射关系，但还要考虑物理内存如何分配，伙伴系统的产生解决了 Linux 系统的外碎片的问题。伙伴系统把内存按照2^n次幂分为大小不同的内存块，然后使用链表把各个内存块链接起来。

Slab 是 Linux 操作系统的一种内存分配机制。其工作是针对一些经常分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内碎片，而且处理速度也太慢。而 slab 分配器是基于对象进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab 分配器就从一个 slab 列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab 分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。下次请求新的对象时，就可以从内存直接获取而不用重复初始化。

1.2.4 程序执行基本原过程

在熟悉了系统的内存管理、存储以及程序的构成后，下一步就可以分析程序的动态执行过程了，也就是本节将要介绍的程序执行的基本原理。

1. *系统启动*

应用程序以及一些系统程序都是运行在操作系统上的，而许多黑客早已将目光放到系统启动这一过程，希望在系统启动过程找到漏洞，继而控制计算机，因而学习系统的启动的过程非常有必要。图 1.26 即为操作系统启动的完整过程。

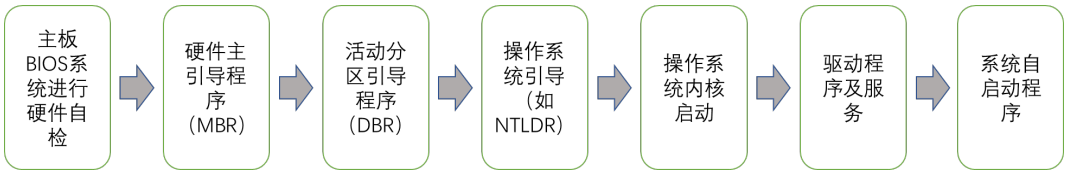

图 1.26 系统启动流程

硬件自检：该阶段控制权属于主板 BIOS Flash（或 ROM）芯片，基本输入输出系统会检测系统中一些关键设备如内存、显卡等是否能正常工作，然后进行初始化并将控制权交给后续引导程序。如图 1.27 所示

图 1.27 Boot 界面

主引导程序：MBR，Master Boot Record 是硬盘第一个扇区，计算机通过这个主分区表定位活动分区，然后装载活动分区的引导程序并移交控制权。

活动分区引导程序：MBR 找到的是活动分区的第一个扇区，在读取该扇区的内容后可以引导整个活动分区的数据，这一步最重要的功能是加载操作系统的引导程序 BootLoader 如 Windows 的 NTLDR，Linux 的 GRUB 等。

操作系统引导：这一阶段计算机的控制权在 BootLoader 手中，BootLoader 依据自身携带的系统文件来识别分区中的文件系统，然后找到系统内核所在位置并解压缩加载到内存中。最后将控制权转移给系统内核。

操作系统内核启动：此时系统内核已经启动，但是没有硬件和文件系统的支持，这个内核就是一个光杆司令，因此这一阶段操作系统主要是检测所有的硬件设备并加载文件系统。

驱动程序及服务：在文件系统被加载后，内核就可以将系统所需的驱动加载到内存并启动相应的系统服务。

系统自启动程序：此时系统已经完全启动，最后一步就是启动用户自定义的开机自启动程序。

不同操作系统启动的过程略有不同，但上述的主要步骤都是一致的，仅仅在使用的工具和文件上有一定区别，此处以 Windows 10 的系统启动流程为例。Windows 10 的启动共包含四个主要步骤：

（1）预启动（PreBoot），在系统加电后开始进行固件的自检工作，并检验磁盘系统是否有效，如果计算机有一个有效的 MBR 则进入下一步，则读取 MBR 中内容并将控制权交给 MBR。

（2）Windows Boot Manager，MBR 查找所在硬盘的硬盘分区表，找到标记为“活动”的主分区。然后加载 Windows Boot Manager 来会读取启动配置数据 BCD（Boot Configuration Data）中的内容，根据其中的内容判断计算机中是否有多个操作系统，若包含，那么它会提供一个包含操作系统名字的菜单，当用户作出选择后，会启动相对应的系统启动程序，以 Windows 为例即 Windows OS Loader。

（3）Windows OS Loader：这一步的程序名为 WinLoad.exe，负责加载系统内核 Ntoskrnl.exe。

（4）Windows NT OS Kernel：这是最后一个阶段，会完成注册表设置和其他的驱动，如加载硬件抽象层 HAL、注册表中 HKEY LOCAL MACHINE SYSTEM 子键、HKEY LOCAL MACHINE SYSTEM 子键中存储的硬件设备驱动程序等等，同时，内存的分段分页也在这一步完成。一旦设置完成，控制权就交给系统管理进程。它会加载 UI 界面、剩余的硬件、软件，也正是在这时，用户可以看到登录界面。

目前，许多新出厂的计算机的 BIOS 已经被 UEFI 所替代，UEFI 旨在解决 BIOS 的局限性，启动过程与 BIOS 无异，优势在于添加了图形界面、支持容量超过 2.2TB 的驱动器、缩短了启动时间、加载系统 Loader 前会进行签名验证提高了安全性。

2. 进程加载

在系统启动后，就可以执行用户应用程序的代码了，用户应用程序的加载由 Windows 加载器完成，PE 文件的加载和运行需要经历 8 个步骤，如图 1.28 所示。

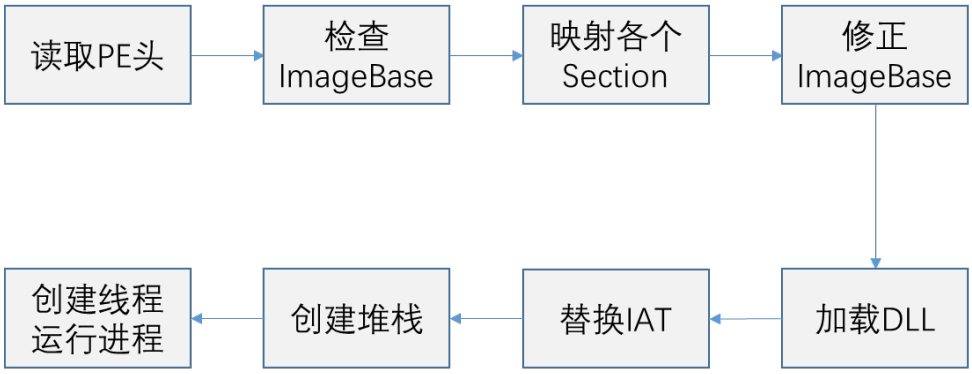

图 1.28 PE 文件加载

首先Windows加载器读入PE文件的文件头和Section头。然后判断PE头里的ImageBase所定义的加载地址是否可用，如果已被其他模块占用，则重新申请一块空间。接着根据Section 头部的信息，把各个 Section 映射到分配的空间并根据各个 Section 定义的数据来修改所映射的页的属性，这一步后续将详细介绍。第四步如果文件被加载的地址不是ImageBase 定义的地址，则重新修正 ImageBase，在 Windows 7 引入加载地址随机化后这一步就成了必须的步骤，第五步会根据 PE 文件的输入表加载所需的 DLL 到进程空间，第六步就是替换掉 IAT，IAT 可以理解为 DLL 中函数在虚拟内存中的地址表，程序在运行过程中会用到动态链接库的部分函数，所以就需要 IAT 来记录这些函数被加载的地址，这些地址只有在 DLL 被装入虚拟内存后才可以获得，所以需要有修改 IAT 的步骤。第七步会根据 PE 头内的数据生成初始化堆和栈。最后初始化线程并开始运行进程，PEB 和 TEB 也将在这一步被初始化，后续也将对这两个结构体详加介绍。

第三步映射 Section 中，Windows 加载器会遍历 PE 文件并决定文件的哪一部分被映射，这种映射方式是将文件较高的偏移位置映射到较高的内存地址中。PE 文件的结构在磁盘和内存中是基本一样的，但在装入内存中时又不是完全复制。Windows 加载器会决定加载哪些地方，哪些部分不需要加载。而且由于磁盘对齐与内存对齐不一致，加载到内存的 PE 文件与磁盘上的 PE 文件各个部分都会有差异。对应的装载图如图 1.29 所示

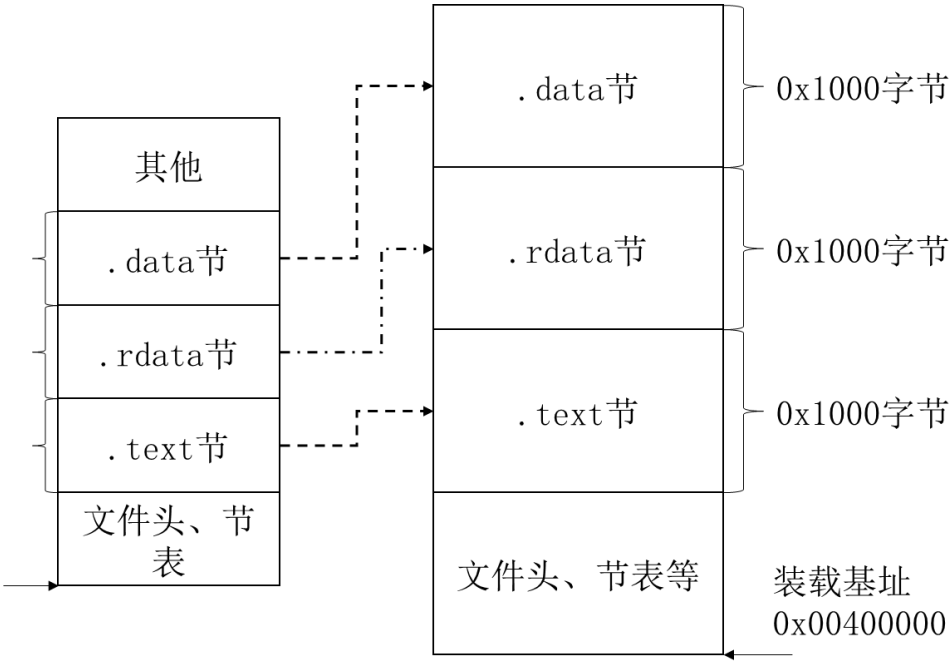

图 1.29 程序在文件和内存中的组织

研究清楚 PE 文件在内存中的地址和在文件中的地址至关重要，因为在分析程序时通常用到的是在虚拟内存中的地址，但是想要保证修改永远有效就必须要找到对应的代码在文件中的位置。想要弄清楚其中的对应关系需要先理解几个相关名词。

文件偏移地址（File Offset Address，FOA）：PE 文件在硬盘上存放时相对于文件头的偏移。

装载基址（Image Base，IB）：PE 装入内存时的基地址，默认情况下 EXE 的装载基址为 0x00400000，DLL 的装载基址为 0x10000000。

虚拟内存地址（Virtual Address，VA）：PE 文件中的指令被装入内存后的地址。

相对虚拟地址（Relative Virtual Address，RVA）：指令的虚拟内存地址相对于装载基址的偏移量。

虚拟内存节偏移（Virtual Section Offset，VSO）：指令的虚拟内存节相对于装载基址的偏移量。

文件节偏移（File Sectoin Offset，FSO）：指令在文件中的地址相对于所属节的偏移量。

出现文件偏移与相对虚拟地址不一致的根本原因在于 PE 文件数据按磁盘数据标准存放，以 0x200 字节为基本单位进行组织，PE 节的大小是 0x200 的整数倍，不足用 0x00 填充。而 PE 文件装入内存后，将按内存数据标准存放，以 0x1000 为基本单位进行组织，在内存中 PE 节的大小是 0x1000 的整数倍，不足用 0x00 填充。在理解清楚上述概念后可以得到以下公式：文件偏移地址=虚拟内存地址-装载基址-虚拟内存节偏移+文件节偏移=RVA-VSO+FSO。

3. 进程和线程控制块

程序在运行开始后有两个非常重要的数据结构，线程环境块 TEB 和进程环境款 PEB。TEB 中存放着进程中线程的各种信息，PEB 是存放进程信息的结构体。

在 Windows 的 32 位系统中，TEB 存放在 fs[0]处，而 fs[0x18]是指向自身 TEB 结构体的指针，因而获取 TEB 的方式是通过 NtCurrentTeb 函数获得 TEB 的结构体指针，如图 1.30 所示，此图利用 Ollydbg 调试而得。在 64 位系统中，TEB 的位置存放在 gs 段寄存器中而非 fs。

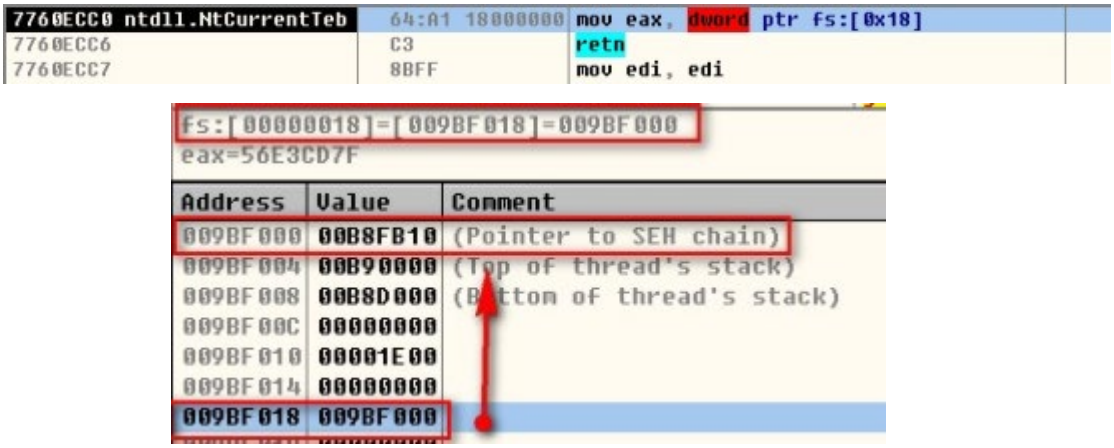

图 1.30 TEB 位置

而 PEB 的结构体指针被存储在 TEB 之中，一个进程的多个线程共享同一个 PEB 结构体，PEB 在 TEB 的 0x30 偏移处，如图 1.31 所示

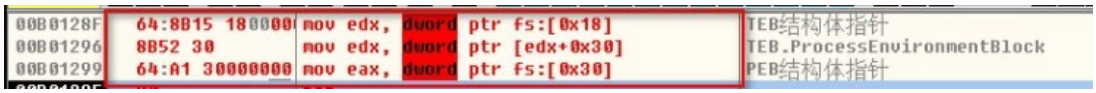

图 1.31 PEB 位置

知道 TEB 和 PEB 地址后，就可以继续分析其中存放的一些关键信息了。TEB 中的关键变量信息如下所示。

代码 1.2 TEB 结构体（部分）

```
typedef struct_TEB
{
    NT_TIB          /*00h   SHE、线程堆*/
    CLIENT_TD       /*20h   进程 ID、线程标识符*/
    Ptr32_PEB       /*30h   PEB 结构体指针*/
    …
```

```
}TEB;
```

NT_TIB 是线程信息块，其中存放着异常链表的指针（EXCEPTION_REGISTRATION_RECORD），这也就是在 Windows 漏洞中基于异常处理链表的溢出攻击的攻击目标，除此之外，该结构体中还存放着线程的堆顶和堆底信息以及线程的自引用指针，上文找 TEB 的 NtCurrentTeb 函数返回的就是这个自引用指针。CLIENT_ID 存放着进程 ID 和线程 ID 也就是 GetCurrentProcessID 和 GetCurrentThreadID 函数的返回值。Ptr32_PEB 存放着进程信息的地址。

进程信息块中的部分信息如下所示。

代码 1.3 PEB 结构体（部分）

```
typedef struct_PEB
{
    BeingDebugged       /*02h   调试标识符*/
    ImageBaseAddress    /*08h   镜像加载地址*/
    LDR               /*0ch 加载模块信息*/
    ProcessHeap       /*18h 进程堆信息*/
    …
}PEB;
```

其中的 BeingDebugged 代表当前进程是否处于调试状态；ImageBaseAddress 也就是 PE 文件头中 IMAGE_OPTIONAL_HEADER.Imagebase 的内容，存放着程序的加载地址；LDR 中存放着加载的 DLL 模块的信息，在上文描述 PE 文件加载过程中的 IAT 表就存放在其中；ProcessHeap 是进程堆的句柄，正常运行时，ProcessHeap.Flags 的值为 2，当其值为 0 时代表程序正在被调试。

线程和进程的描述结构体很复杂，因而仅仅做简单介绍，程序在运行过程中可以通过不同的系统调用获取到这些结构体的信息以供使用。

1.3 软件安全分析工具

工欲善其事必先利其器，软件安全问题分析与逆向工程是分不开的，各种主流逆向工具在软件安全分析领域都经常被用到。这里列举几个在本书中经常使用到的工具。

1.3.1 IDA Pro

在软件安全分析和逆向工程领域中，IDA 是一款必备的工具，在反汇编应用中享有独树

一帜的地位，而且支持 Windows、Linux 和 Mac 等多个系统平台，几乎没有一款同类工具能够完全替代它。IDA Pro 本身提供一些基本逆向、反汇编、字符串、结构定义、搜索等等功能，尤其是其反汇编功能是目前最强大的，支持绝大部分指令集架构，更强大的是其提供的一些反编译插件，可以将汇编代码反编译成高级语言，是目前最受欢迎的一款软件分析工具。

虽然目前的 IDA Pro 可以做一些简单的动态调试工作，但绝大多数情况下，分析员使用的是它的静态反汇编功能。很多工具都能把二进制的机器代码翻译成汇编指令，但为什么提起反汇编工具，IDA Pro 永远是首屈一指的强者呢？这是因为 IDA Pro 拥有强大的标注功能。即使是对汇编语言非常精通的程序员也无法直接阅读成千上万行汇编指令。在进行汇编分析前，需要把庞大的汇编指令序列分割成不同层次的单元、模块、函数，对其逐个研究，最终摸清楚整个二进制文件的功能。

所谓软件逆向分析，在很大程度上就是对这些代码单元的标注。每当弄清楚一个函数的功能时，分析员就会给这个函数起一个名字。使用 IDA 对函数进行标准和注解可以做到全文交叉引用，也就是说，标注一个常用函数后，整个程序对这个函数的调用都会被替换成所标注的名字，这可比直接对内存地址的调用形式好多了（通常情况下，反汇编得到的函数调用往往都是对内存地址的调用）。对汇编代码的标注可以自上而下进行，也可以自下而上进行。自上而下是指从 main 函数开始标注，相当于对函数调用图从树根开始遍历；自下而上逆向是指从比较底层的经常被调用的子函数开始标注，每标注一个这样的底层函数，代码单元的可读性就会增加许多，当最终标注到 main 函数时，整个程序的功能和流程基本上可以掌握了。大多数情况下，IDA 会从两个方向同时开始逆向。目前的 IDA Pro 能够自动标注 VC、Broland C、Delphi、Turbo C 等常见编译器中的标准库函数。在反汇编的结果中发现所有的 memcpy、printf 函数都已经被自动标注好一定能够。如图 1.32 所示。

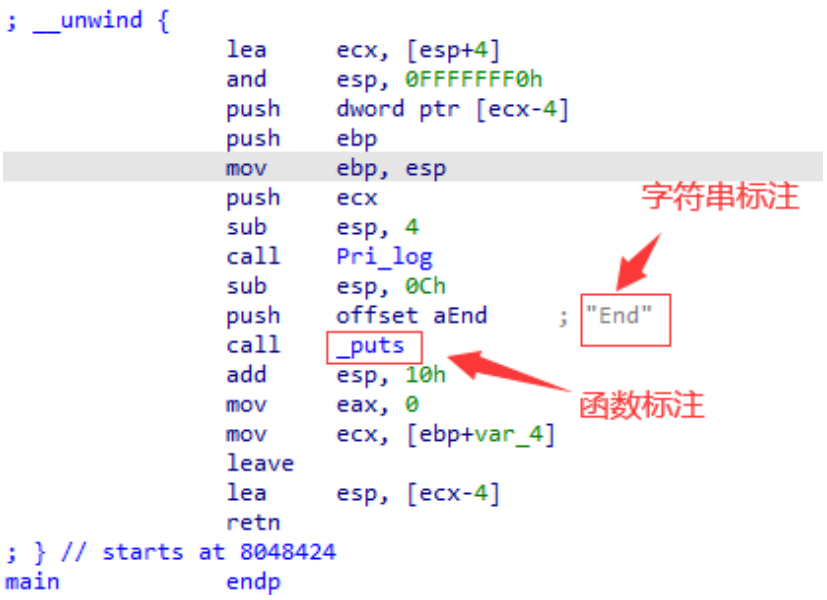

图 1.32 IDA 标注功能

IDA Pro 可以看作是一张二进制的地图，通过它的标注功能可以迅速掌握大量汇编代码的框架，不至于在繁杂的二进制迷宫中迷失方向。目前版本的 IDA Pro 甚至可以用图形方式显示出一个函数内部的执行流程。在反汇编界面中按空格就可以在汇编代码和图形显示间切换。如图 1.33 所示：

图 1.33 IDA 代码框架

IDA Pro 的扩展性非常好，除了可以用 IDA Pro 提供的 API 接口和 IDC 脚本扩展它自身外，IDA Pro 还可以把标注好的函数名、注释等导入 Ollydbg，这样在动态调试的时候也不会晕。如果把 IDA Pro 自身的标注比作纸质地图，那么这个功能就相当于车载 GPS 的电子地图了。

这里给出几个 IDA 常用的快捷键命令，如表 1-1 所示。

表 1-1 IDA 快捷键

快捷键	功能
;	为当前指令添加全文交叉引用的注释
N	定义或修改名称，通常用来标注函数名
G	跳转到任意地方观察代码
ESC	返回到跳转前的位置
D	分为按字节、字、双字的形式显示数据
A	按照 ASCII 形式显示数据

知道这几个快捷键，就可以自行去标注汇编代码了。彻底掌握 IDA Pro 不是一两天就能做到的，由于在漏洞利用中主要使用的是动态调试工具，所以 IDA 的许多高级特性（如编写 IDC 脚本等）本书暂不介绍。

1.3.2 Ollydbg

为了动态跟踪程序的执行过程，需要使用到调试器，许多从事安全工作的人员可能更喜欢 Ollydbg 这款免费软件，简称 OD，如图 1.34 所示。在破解软件、调试逆向中它常被用到，OD 是 Ring 3 级调试器，非常容易上手，已经基本替代了老一辈的调试器 SoftICE，它支持插件扩展。

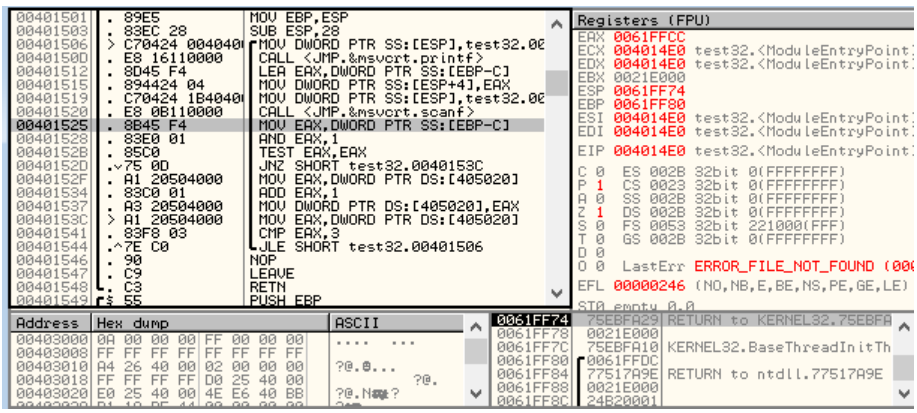

图 1.34 Ollydbg 界面

1.3.3 x64/32dbg

x64/32dbg 是 Windows 下的开源 32/64 位调试器。x64/32dbg 采用 QT 平台编写，支持多国语言。熟悉 Ollydbg 的用户应该很容易上手。x64/32dbg 整体前景比较乐观，功能有待加强。通过这款调试工具，用户可以分析 64 位的应用。x64/32 支持类似 C 的表达式解析器、全功能的 DLL 和 EXE 文件调试、IDA 般的侧边栏与跳跃箭头、动态识别模块和串、块反汇编、可调试的脚本语言你自动化等多项使用分析功能，其最大优势在于可以分析 64 位的应用程序。

x64/32dbg 界面简洁清晰、操作方便快捷，设计人性化，如图 1.35 所示。在 x64/32 的主界面上包含了反汇编、寄存器、数据、堆栈四大窗口，可以清晰地看到程序运行的基本状况。

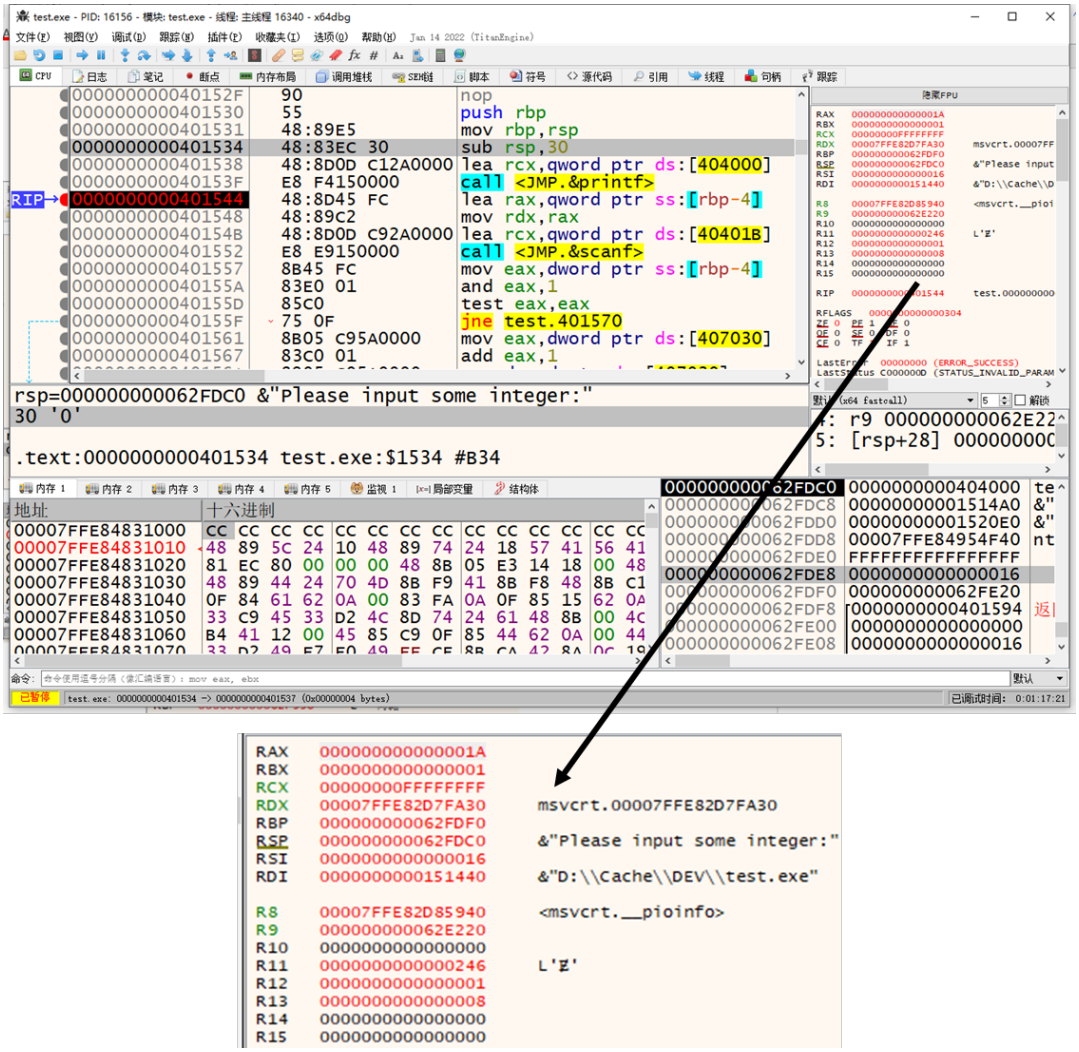

图 1.35 x64dbg 主页面

x64/32 调试上手比较容易，下面表格 1-2 介绍了一些基本的操作命令。

表 1-2 x64/32 操作命令

按键	功能
F2	设置断点
F4	运行到光标处
F7	单步步入（遇到 call，进入 call 函数的实现处继续执行）
F8	单步步过（遇到 call，执行 call 后，接着执行下一条指令）
F9	运行程序（如果没有断点，会一直运行下去）
Ctrl+F9	执行到函数返回处，用于跳出函数实现
Alt+F9	执行到用户代码，用于快速跳出系统函数
Ctrl+G	输入十六进制地址，快速定位到该地址处

x64/32 还提供了非常友好地搜索功能，用户可以根据需要自定义搜索的内容和范围，如图 1.36，1.37 所示。借助于搜索结果，调试者可以快速找到关键的代码。

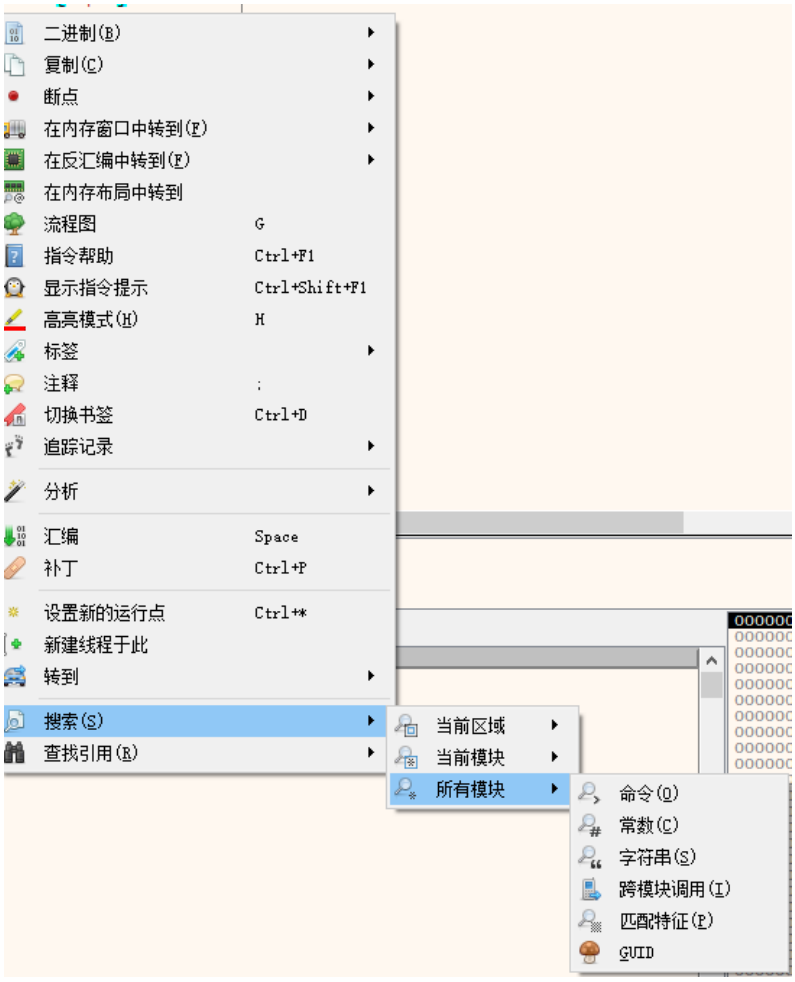

图 1.36 搜索功能

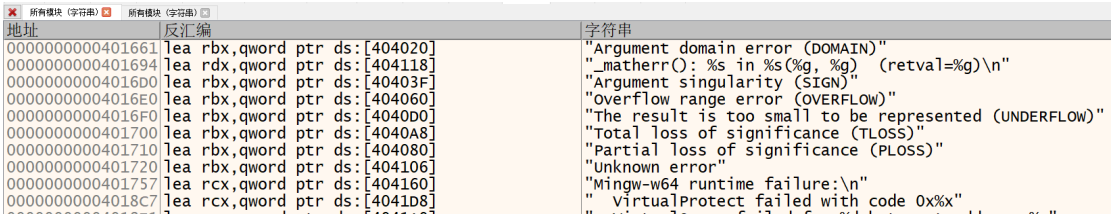

图 1.37 搜索结果

在掌握了上述简单的操作命令后就可以对 64 位的程序进行调试，但想要对程序有更深入地了解，如调用的动态链接库、线程情况等，需要读者自行去深入研究 x64/32 的操作手册。

1.3.4 GDB

GDB 是 Linux 环境下功能强大的调试器，用来调试 C/C++写的程序。GDB 主要具有下面四个方面的功能，启动程序，可以按照自定义的要求随心所欲地运行程序；可以让被调试的程序在所制定的调试断点处停住；当程序被停住时，可以检查此时程序中所发生的事；可以改变你的程序，将一个 BUG 产生的影响修正从而测试其他 BUG，如图 1.38 所示。

```
coral@coral-virtual-machine:~$ gdb tt
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/g
pl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show cop
ying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from tt...done.
(gdb) start
Temporary breakpoint 1 at 0x8048448: file test.c, line 9.
Starting program: /home/coral/tt

Temporary breakpoint 1, main () at test.c:9
9               Pri_log();
(gdb) step
Pri_log () at test.c:4
4               for (int i=0;i<10;i++)
(gdb) n
5                       printf("This is a log line\n");
(gdb) print i
$1 = 0
(gdb) bt
#0  Pri_log () at test.c:5
#1  0x0804844d in main () at test.c:9
(gdb)
```

图 1.38 GDB 界面

思考题:

思考题 1: 带着这些问题，请看代码 1，思考一下如果这个代码运行会产生什么后果?

代码 1

```
#define RECT2(a, b) (a * b)
int g_exam;
unsigned int example(int param, unsigned int w, unsigned int h)
{
unsigned int temp; unsigned int Area;
g_exam = para;
Area=RECT2(w+param,h);
temp = square_exam ( Area,g_exam);
return temp;
}
```

思考题 2

某大数据处理程序需要对大规模计算结果进行分布统计，计算结果在 0.00-1.00 之间，估计有 100 万数据量，试按照 0.01 为分区间隔统计出各个区间的数值分布。

```
for(int i=0;i<NumberAll;i++){
(if(a[i])<0.01)&&(if(a[i])>=0.0) R[0]++;
(if(a[i])<0.02)&&(if(a[i])>=0.01) R[1]++;
(if(a[i])<0.03)&&(if(a[i])>=0.02) R[2]++;
…
(if(a[i])<1.00)&&(if(a[i])>=0.99) R[99]++;
}
```

思考: 这个统计代码运行会出现什么问题，如何设计才是合理的?

思考题 3: 每个进程可用 4GB 内存空间，但是电脑的实际物理内存不足 4G 怎么办? 两个进程的可执行程序映像加载地址都是 0x00400000H，但同一地址对应的代码却不一样，这是如何做到的?

思考题 4:

CODE 节虚拟内存节偏移 VSO=0x1000，文件节偏移 FSO=0x400，虚拟内存地址 0x0049FBC4 处有一条指令 mov edi，eax（8B F8），请问文件偏移地址是多少？（设基址

IB=0x400000）

答：FOA=VA-IB-VSO+FSO=0x49FBC4-0x400000-0x1000+0x400=0x9EFC4