[OpenBMC device driver development]: A must-read for beginners, write your first driver step by step

Unlock now

Published: 2025-02-21 12:22:07 Views: 143 Subscribers: 34 VIP专栏

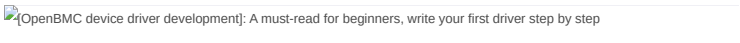
ZIP openbmc bmc web Download Now
OpenBMC webserver This component attempts to be a "do everything" embedded webserver for OpenBMC. Features The webserve Expand

SW_Sun Wei
Development technology expert

An engineer at a well-known technology company, he has extensive work experience and expertise in the field of technology development. He has been responsible for the design and development of multiple complex software systems involving large-scale data processing, distributed systems, and high-performance computing.

Copy full text

Table of contents



summary

This article provides a comprehensive introduction to driver development on the OpenBMC platform, from basic concepts to advanced topics, and details the construction of the driver development environment, the key points of infrastructure writing, drivers, but also explores the modification and expansion of the device tree, the implementation of the interrupt controller, and the application of synchronization mechanisms. In addition, this article strengthens the application of theoretical knowledge through case studies, helping developers understand solutions to problems encountered on actual hardware platforms and the use of community resources. This paper aims to provide detailed guidance and reference materials for OpenBMC driver developers.

Keywords

OpenBMC; driver development; hardware abstraction layer; device tree; interrupt processing; performance optimization

Reference resource link: OpenBMC New Model Development Guide: From Machine Layer to Kernel Changes

1. Introduction to OpenBMC and overview of driver development

1.1 Overview of OpenBMC

OpenBMC is an open source project that aims to provide firmware for systems based on standard server hardware. The project is supported by the Linux Foundation and built using the Yocto Project build system. OpenBMC provides a range of services and tools to manage the life cycle of servers, such as booting, monitoring, event handling, and shutdown. It uses a microservice architecture so that its components can interact through REST APIs for remote management. The flexibility and scalability of OpenBMC make it the first choice for many large data centers.

1.2 The Importance of Driver Development

Column Introduction

The OpenBMC New Model Development Documentation column provides comprehensive guides and in-depth analysis for OpenBMC developers. It covers 10 key areas, including getting started, system ar... 展开全部

Column Directory Unlock now

Latest Recommendations

Exploration of structured light 3D sc...
![Exploration of structured light 3D scanning technology in the medical field: potential and…
[Algorithm implementation details]: ...
![LDPC.zip_LDPC_LDPC Rayleigh_LDPC Rayleigh Channel_accidentls3_wonderygp]…
【Architecture Design】: Building M...
![Oracle Pro*C](https://365datascience.com/wp-content/uploads/2017/11/SQL-DELETE-…
The future of TreeComboBox contro...
![The future of TreeComboBox control: detailed explanation of virtualization technology and…
Circuit Design with MATLAB: An Ex...
![Circuit Design MATLAB: Expert Guide to Simulation and Analysis](https://dl-…
The role of ProE Wildfire Toolkit in p...
![ProE Wildfire TOOLKIT](https://docs.paloaltonetworks.com/content/da…

navigation Library Home Download resources Album 大学生专区 openWRT custo search AI Search Log in 50 yuan off for new customers member information Growth Tasks creation

Buy 1 year, get 3 months free
Unlock the column at a minimum of 0.4? yuan/day
Million-level VIP articles Unlimited access to high-quality
Ten million level download High-quality resources for free
Ten million level free to read High-quality library answers

Click here to give feedback

In the OpenBMC environment, drivers play a vital role. They are the bridge between hardware and software, responsible for abstracting and managing hardware resources. Good driver development can not only ensure the stable operation of the hardware, but also improve the performance and security of the hardware. Since OpenBMC usually runs on server hardware, its driver development also needs to take into account the special requirements of the hypervisor and virtualization environment for hardware.

1.3 Basic concepts of driver development

The OpenBMC community encourages developers to use standardized interfaces to support compatibility across hardware platforms. Basic driver development involves understanding hardware specifications, implementing the device abstraction layer (DAL), and writing code for hardware resource management. Driver developers need to have comprehensive knowledge of hardware and software, as well as a deep understanding of system architecture to ensure that the driver they develop can interact with the hardware stably and efficiently.

The above content is the introduction of the first chapter of the article, which provides readers with a macro overview of OpenBMC and its driver development. The following chapters will delve into the specific development process, environment construction, and driver writing practice.

2. OpenBMC driver development environment construction

The efficient execution of OpenBMC driver development is inseparable from a stable and properly configured development environment. In this section, we will discuss in detail how to build and configure an environment suitable for OpenBMC driver development.

2.1 Introduction to OpenBMC Development Environment

2.1.1 OpenBMC Software Architecture and Components

OpenBMC is an open source project that aims to provide firmware for data centers, servers, network devices, etc. Its software architecture is designed to provide a scalable and modular foundation.

The software components of OpenBMC include the following key parts:

- **phosphor-bmc** : This is the core of the BMC firmware, including system management, lifecycle management, and health monitoring.
- **phosphor-dbus-interfaces** : Provides a hardware abstraction layer for communicating with the BMC via the D-Bus interface.
- **phosphor-rest-server** : A RESTful service for remote management of BMC.
- **phosphor-webui** : Web-based user interface for intuitive BMC management.

Understanding the interrelationships and functionality of these components is critical to developing OpenBMC drivers.

2.1.2 Tools and dependencies required for driver development

When developing OpenBMC drivers, some specific tools and dependencies are required so that developers can write, test, and debug code.

Core tools include:

- **Cross-compilation toolchain** : used to build OpenBMC firmware suitable for the target hardware.
- **OpenBMC SDK** : Contains library files and header files to facilitate developers to access OpenBMC APIs.
- **Version control system** : such as git, used to manage code versions and collaborate.

Dependencies may include, but are not limited to:

- **Dependent libraries** : such as libnl and libxml2, used for network communication and XML processing.
- **Build system** : such as OpenEmbedded or Buildroot, used to automate the build process.

2.2 OpenBMC development environment configuration

2.2.1 Installation steps for virtual machines or physical machines

OpenBMC development can be done on physical hardware, but most developers prefer to use virtual machines to reduce the complexity and risk of environment configuration. Here are the installation steps:

1. **Choose a virtualization platform** : such as VMware or VirtualBox.
2. **Get the OpenBMC image** : Get the precompiled image from the OpenBMC official website or GitHub.
3. **Configure the VM** : Create a new VM and select the OpenBMC image as the boot disk.
4. **Start the virtual machine** : Turn on the virtual machine and connect via the serial port or network.

2.2.2 Configuration method of driver development environment

Once the basic OpenBMC environment is set up, you need to configure the driver development environment:

1. **Get the source code** : Use git to clone the code from the OpenBMC project repository to your local computer.
2. **Configure the build system** : Set up environment variables and build configuration files.
3. **Compile OpenBMC** : Select the appropriate build configuration based on the target hardware and use the build system to generate the firmware.
4. **Configure Networking** : Ensure that the OpenBMC instance is accessible over the network for development and debugging purposes.

2.3 Installation and verification of the development tool chain

2.3.1 Selection and installation of compilation toolchain

For OpenBMC development, a cross-compilation toolchain is usually required. Select the appropriate toolchain to install based on the architecture of the target hardware. Taking the ARM architecture as an example, developers may choose a toolchain with arm-linux-gnueabihf- as the prefix.

The installation steps usually include:

1. **Get the toolchain** : You can download it from the toolchain provider's website or install it using a package manager.
2. **Configure environment variables** : Make sure the compiler path is in the PATH environment variable so that it can be called from any location.

For example, to install the arm-linux-gnueabihf-gcc compiler in Ubuntu:

```
sudo apt-get install gcc-arm-linux-gnueabihf
```

2.3.2 Verify the correctness of the development environment configuration

After installing the toolchain, you need to verify that the development environment is configured correctly. This is usually done by compiling a simple "Hello World" program.

Here is a sample code:

```c
// hello.c
#include <stdio.h>

int main() {
    printf("Hello, OpenBMC!");
    return 0;
}
```

The command to compile the code:

```
arm-linux-gnueabihf-gcc -o hello hello.c
```

If the compilation is successful and the generated `hello` executable file can be run, it means that the development environment is configured correctly.

After the above steps, the OpenBMC driver development environment should be set up. Next, developers can start writing code and developing drivers. To ensure the effectiveness and stability of the development environment, it is recommended that developers thoroughly test the environment before actual development.

3. OpenBMC driver development basics

After having a deep understanding of the OpenBMC architecture and development environment, developers need to master the basic knowledge of driver development. This chapter will introduce in detail the role and classification of driver programs and how to write basic driver code.

3.1 Functions and classification of drivers

3.1.1 Understanding the Hardware Abstraction Layer in OpenBMC

OpenBMC is an open source firmware stack designed for baseboard management controllers (BMCs). The hardware abstraction layer (HAL) is a key part of connecting hardware and operating systems. It provides a standard interface to upper-layer software, making the upper-layer software independent of specific hardware. In OpenBMC, the HAL layer enables the same operations on different hardware platforms to be performed using a unified interface.

3.1.2 Driver Types and Application Scenarios

OpenBMC drivers can be divided into the following types according to their functions:

- **Platform Drivers** : For devices that are closely related to a specific hardware platform.
- **Bus Drivers** : Manage multiple devices on a certain type of bus.
- **Device Drivers** : Directly interact with hardware devices and provide device functionality.

These drivers can be customized for different subsystems of the BMC, such as IPMI (Intelligent Platform Management Interface), Redfish API, etc.

3.2 Basic structure and writing points of driver

3.2.1 Code structure analysis of OpenBMC driver

OpenBMC drivers usually consist of the following main parts:

- **Device Registration** : Define the device name, ID and other information.
- **Driver Initialization** : Implements initialization work when the device is loaded.
- **Operations Interface** : defines the specific functions of device operations, such as reading and writing, opening and closing, status checking, etc.
- **Resource Management** : Allocate and manage hardware resources, such as memory, interrupts, etc.

3.2.2 Key functions and interfaces of the driver

The following is a brief introduction to some key functions and interfaces:

- `probe` **Function** : When a device is discovered, this function will be called to initialize the device.
- `remove` **Function** : This function is used to perform cleanup when the device is removed.
- `read` **And** `write` **functions** : Provide read and write interfaces for device registers.
- `open` **And** `release` **functions** : Control the opening and closing operations of device files.

3.3 Practical Guidelines for Writing Drivers

3.3.1 Actual case analysis: a simple GPIO driver

GPIO (General-Purpose Input/Output) is a general-purpose input and output interface. The following is a code example of a simple GPIO driver:

```c
#include <linux/module.h>       // 引入模块基础功能
#include <linux/platform_device.h> // 平台设备相关功能

static int gpio_probe(struct platform_device *pdev) {
    // 该函数会在发现设备时被调用
    printk(KERN_INFO "GPIO Driver probed\n");
    return 0;
}

static int gpio_remove(struct platform_device *pdev) {
    // 设备移除时的处理
    printk(KERN_INFO "GPIO Driver removed\n");
    return 0;
}

static struct platform_driver gpio_driver = {
    .driver = {
        .name = "simple_gpio",
        .owner = THIS_MODULE,
    },
    .probe = gpio_probe,
    .remove = gpio_remove,
};

static int __init gpio_init(void) {
```

```
    return platform_driver_register(&gpio_driver);
}

static void __exit gpio_exit(void) {
    platform_driver_unregister(&gpio_driver);
}

module_init(gpio_init);
module_exit(gpio_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple GPIO driver for OpenBMC");
```

3.3.2 Implementation details of driver loading and unloading

- **Load driver** : `module_init` The macro defines an initialization entry function that is called when the module is loaded.
- **Uninstall driver** : `module_exit` The macro defines a cleanup entry function that is called when the module is unloaded.

When a driver is loaded, it will first be registered with the system's device driver model. Then, when the system detects the corresponding hardware, it will call `probe` the function in the driver to initialize the hardware device. When the device is no longer in use, `remove` the function is called to clean up the resources related to it.

This GPIO driver example shows the core concepts and key steps to focus on when developing a basic driver on OpenBMC. In practice, developers need to write corresponding operation functions according to the technical manual and data sheet of the specific hardware, and correctly configure the parameters of the device.

4. OpenBMC Device Driver Advanced Topics

4.1 Use and understanding of device tree

The device tree is a key data structure for representing hardware resources in OpenBMC. It describes the topology and configuration information of the hardware platform in a human-readable tree format. In the development of OpenBMC drivers, it is crucial to understand and correctly use the device tree.

4.1.1 Function and structure of device tree

The main function of the device tree is to separate the configuration information of the hardware platform from the kernel code, making the kernel more universal. Through the device tree, developers can describe the processor, peripherals, buses and the relationship between them, and adapt to different hardware environments without modifying the kernel code itself.

Structurally, the device tree consists of a series of nodes, each of which has a type (such as `compatible`), which defines the type of device the node represents. Nodes can contain child nodes, attributes, and values. Attributes are usually used to specify specific parameters of the device, such as address, interrupt number, etc.

```
/dts-v1/;

/ {
  model = "My OpenBMC Device Tree";
   compatible = "vendor,my_model";

  chosen {
       bootargs = "console=ttyS0,115200";
   };

ждународная система единиц
  aliases {
       ethernet0 = &ethernet0_node;
   };

  ethernet0_node: ethernet@0 {
       compatible = "ethernet-vendor-model";
       reg = <0x00000000 0x200000>;
       interrupt-parent = <&intc>;
       interrupts = <17>;
   };
};
```

The above is a simplified device tree example that shows the basic structure of the device tree.

4.1.2 How to modify and extend the device tree in OpenBMC

In OpenBMC, modifying and extending the device tree usually involves editing the existing device tree files. These files are usually located in the OpenBMC source code `dts/` directory. Developers need to use the device tree compiler (DTC) to compile the device tree source file (.dts) to binary format (.dtb).

1. Get the device tree source file:
   ```
   cp /path/to/openbmc/meta-phosphor/recipes-bsp/device-tree/files/*.dts /path/to/oper
   ```
2. Edit the .dts file and add or modify nodes and attributes.
3. Compile the device tree using DTC:
   ```
   dtc -I dts -O dtb -o /path/to/output.dtb /path/to/device_tree.dts
   ```
4. Copy the compiled device tree file to the device tree directory of the target system.

4.2 Interrupt handling and synchronization mechanism

4.2.1 Implementation of the interrupt controller in OpenBMC

The interrupt controller is the core component of the processor to respond to external events. It manages hardware interrupt requests and maps them to appropriate interrupt handling functions on the processor. In OpenBMC, implementing an interrupt controller usually involves writing the corresponding driver code to handle interrupt requests and dispatch interrupt service routines.

In the OpenBMC code base, interrupt handlers are registered with the kernel's interrupt subsystem, usually by calling `request_irq()` a function that not only registers the interrupt service routine but also specifies the device ID to use when handling the interrupt.

```
static int __init my_irq_init(void) {
    int result = request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED, "my_driver", NULL);
    if (result != 0) {
        pr_err("Unable to request IRQ %d\n", IRQ_NUMBER);
```

```
        return result;
    }
    return 0;
}

static void __exit my_irq_exit(void) {
    free_irq(IRQ_NUMBER, NULL);
}
```

4.2.2 Application of synchronization mechanism in multi-threaded driver

In multi-threaded driver development, synchronization mechanisms are crucial to prevent race conditions. The OpenBMC kernel supports a variety of synchronization mechanisms, including mutexes, spinlocks, semaphores, and seqlocks.

Mutexes are often used in simple synchronization scenarios, especially in contexts where sleeping is possible; they allow a thread to sleep while waiting for a resource:

```
static DEFINE_MUTEX(my_mutex);

void my_function() {
    mutex_lock(&my_mutex);
    /* Critical section */
    mutex_unlock(&my_mutex);
}
```

The use of mutexes is very common because they provide a simple locking and unlocking mechanism, but they should not be used in contexts that cannot sleep, which can cause deadlocks. Spinlocks are suitable for use in short critical sections, especially in interrupt handlers and bottom halves.

4.3 Debugging and Performance Analysis

4.3.1 Using the debugging tools provided by OpenBMC

OpenBMC provides a variety of tools to help developers debug, such as `kgdb`,, `kdump`etc. `journald`These tools can be used to monitor the running status of the system, capture information when the kernel crashes, and retrieve system logs.

Take syslog as `journald`an example, it is `systemd`part of syslog, which provides powerful query capabilities for system logs. The following is an `journalctl`example of how to use syslog to view system logs:

```
journalctl -k | grep "my_driver"
```

This command will display all kernel messages containing the string "my_driver", which is often useful for tracking the operation of the driver and debugging errors.

4.3.2 Driver performance testing and optimization strategies

Performance testing is a critical step in ensuring driver quality. In OpenBMC, performance testing can be performed in various ways, such as using `stress-ng`load creation, or using professional hardware test suites.

The results of performance testing can help developers identify bottlenecks, such as I/O latency, CPU usage, or memory consumption. Once the bottleneck is found, developers can apply a variety of optimization strategies, such as:

- Cache utilization: Cache frequently used data in memory to avoid frequent I/O operations.
- Concurrency control: Manage multiple threads properly to avoid unnecessary synchronization overhead.
- Resource recycling: ensure that no longer used resources are released in a timely manner to prevent memory leaks.

```
void optimize_driver() {
    // 示例：优化缓存使用，减少I/O操作次数
    if (cacheMiss) {
        read_from_device();
        cacheData();
    } else {
        use_cache_data();
    }
}
```

This code snippet shows a basic optimization method for cache usage, avoiding unnecessary I/O operations when possible, thereby optimizing performance.

5. OpenBMC driver development case practice

5.1 Choose an actual hardware platform

5.1.1 Hardware Platform Characteristics and Requirements

In the OpenBMC project, choosing the right hardware platform is the first step in driver development. The hardware platform features we usually consider include but are not limited to the following:

- **Processor architecture** : whether it is ARM, x86 or other architecture.
- **Interface support** : What interfaces are available, such as PCIe, I2C, SPI, GPIO, etc.
- **Peripheral devices** : whether they include specific sensors, network controllers, etc.
- **Firmware features** : The types of firmware supported by the hardware and how to upgrade it.
- **Community support** : Whether the hardware is widely supported and used in the OpenBMC community.

Based on these characteristics of the hardware platform, developers can evaluate whether it meets the needs of a specific project. The choice of hardware platform will directly affect the development of subsequent drivers.

5.1.2 Preliminary preparation for hardware driver development

Before formal coding, developers need to do a series of preliminary preparations:

- **Hardware Documentation** : Get detailed hardware specifications and interface definition documents.
- **Software Toolchain** : Install and verify the software toolchain appropriate for the target hardware.
- **Development environment** : Build an OpenBMC development environment suitable for the hardware platform.
- **Test environment** : Prepare the appropriate test hardware to ensure that the driver can be tested in an isolated environment.
- **Code repository** : Store hardware-related driver codes in a suitable code repository.

Adequate preparation can greatly improve development efficiency and reduce possible problems that may arise later.

5.2 Driver Development Practice

5.2.1 Writing driver code for the selected hardware

After selecting the hardware platform, the next step is to write the corresponding driver code according to the characteristics of the hardware. The following is a simple example showing how to write the basic framework of the OpenBMC driver for the GPIO device:

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define GPIO_PATH "/sys/class/gpio"

void gpio_export(int gpio) {
    int fd = open(GPIO_PATH "/export", O_WRONLY);
    if (fd < 0) {
        perror("Failed to open export for writing");
        return;
    }
    write(fd, &gpio, sizeof(gpio));
    close(fd);
}

void gpio_unexport(int gpio) {
    int fd = open(GPIO_PATH "/unexport", O_WRONLY);
    if (fd < 0) {
        perror("Failed to open unexport for writing");
        return;
    }
    write(fd, &gpio, sizeof(gpio));
    close(fd);
}

void gpio_direction(int gpio, const char *dir) {
    int fd = open(GPIO_PATH "/gpio%d/direction", O_WRONLY);
    if (fd < 0) {
        perror("Failed to open direction for writing");
        return;
    }
    write(fd, dir, strlen(dir) + 1);
    close(fd);
}

int main(int argc, char *argv[]) {
    int gpio = atoi(argv[1]);
    gpio_export(gpio);
    gpio_direction(gpio, "out");
    // ...此处可以添加更多GPIO控制逻辑...
    gpio_unexport(gpio);
    return 0;
}
```

The above code shows how to export a GPIO pin, set it to output mode, and then unexport it at the end. This is just a basic example. The actual driver writing will be more complicated and need to handle device initialization, read and write operations, interrupt handling, etc.

5.2.2 Driver compilation, loading and testing

After writing the driver code, you need to compile it into a format that can be loaded in the OpenBMC environment. Typically, this involves modifying the Makefile file and compiling it using the OpenBMC compilation toolchain.

```
all:
    make -C $(src) M=$(PWD) modules
clean:
    make -C $(src) M=$(PWD) clean
```

After the compilation is complete, use `insmod` the command to load the driver module, and use `dmesg` or `/var/log/messages` to view the loading results and output information.

```
insmod gpio_driver.ko
dmesg | tail
```

After loading the driver, you can write test cases to verify whether the driver functions normally.

5.3 Problem Solving in Driver Development

5.3.1 Common problems encountered and solutions

When developing an OpenBMC driver, developers may encounter a variety of problems:

- **Hardware incompatibility** : Check whether the hardware specifications are consistent with those supported by the driver.
- **Compilation error** : Check whether the compilation environment settings and Makefile configuration are correct.
- **Loading failure** : Use `dmesg` to find error messages when loading the driver and solve the problem accordingly.
- **Functional anomalies** : Debug with reference to the hardware manual and kernel documentation.

For example, if `dmesg` the error "Permission denied" is displayed, it may be because the driver module has not set permissions correctly. In this case, you need to check the permission setting part in the driver source code and recompile and load it.

5.3.2 Community support and resource acquisition

The OpenBMC community is an invaluable resource when you encounter difficult problems. Community members and maintainers can provide timely assistance. Here are some ways to obtain community resources:

- **Mailing list** : You can communicate with other developers through the mailing list.
- **GitHub issue tracking** : Submit issues to the corresponding GitHub repository and provide detailed problem descriptions and steps to reproduce the issue.
- **IRC Channel** : Participate in relevant IRC channel discussions and exchange problem-solving methods in real time.

```
# IRC服务器地址和频道示例
irc.freenode.net #openbmc
```

The community is an important part of learning and growing, and actively participating in community discussions can not only solve problems, but also help improve your own skills and visibility.

After this chapter, I hope that readers will have a deeper understanding of OpenBMC driver development and be able to

Buy 1 year, get 3 months free
Unlock the column at a minimum of 0.47 yuan/c Click to view next article

400 times Number of member resource downloads
3 million+ Quality blog posts
10 million+ High-quality download resources
10 million+ High-quality library answers

Related Recommendations

Open BMC User Management Development Documentation
Open BMC User Management Development Documentation

Open BMC User Manual
open bmc user guide1. The primary management interface for Open BMC is REST. This document provides some basic structure an...

[Open BMC Device Driver Development Guide]: In-depth study of driver architecture and development process
Open BMC device driver development is a key technology for hardware control and management of embedded systems based on B...

Open BMC WebUI Refactoring: Vue.js Modern Development and Accessible Design
1. **Active community and future development plan**: Vue.js has an active developer community, which guarantees long-term suppo...

Open BMC BIOS Configuration Guide: UEFI/BIOS Configuration Details
RBC Daemon is a daemon dedicated to Open BMC , responsible for managing the configuration and status of Open BMC . RBC Da...

Open BMC Infrastructure Documentation: Organization, Application, and Automated Testing
Open BMC Infrastructure is an open source project led by the Open POWER Foundation, focusing on building and managing open ...

Open BMC test case sharing: a platform for industry best practices and experience exchange
Open BMC is an open source basic management controller software used to simplify the management of data centers and embedde...

ASpeed2600 Open BMC Troubleshooting Guide: MCTP Over PCIe Worry-free Use
![ASpeed2600 Open BMC Troubleshooting Guide: MCTP Over PCIe Worry-free Use](https://www.simms.co.uk/Images/Tech-Talk/wh...

Facebook Open BMC vs. Traditional BMC : Advantage Comparison Analysis and Reliability Improvement Stra...
![Facebook Open BMC vs Traditional BMC : Advantage Comparison Analysis and Reliability Improvement Strategy]...Through the co...

about Us Careers Business Cooperation Seeking coverage 400-660-0108 kefu@csdn.net Online Customer Service Working hours 8:30-22:00

Public Security Registration Number 11010502030143 Beijing ICP No. 19004658 Beijing Internet Publishing House [2020] No. 1039-165 Commercial website registration information Beijing Internet Illegal and Harmful Information Reporting Center Parental Control Online 110 Alarm Service
China Internet Reporting Center Chrome Store Download Account Management Specifications Copyright and Disclaimer Copyright Complaints Publication License Business license ©1999-2025 Beijing Innovation Lezhi Network Technology Co., Ltd.