**实验题目：使用Libvirt管理Unikraft**

# 实验内容：

1. Libvirt成功启动虚拟机

2. Libvirt提供挂载支持

3. Libvirt提供网络支持

4. 通过libvirt跑通unikraft上的nginx

5. GO语言上使用libvirt提供的接口，以控制虚拟机

# 实验过程：

## Libvirt原理

Libvirt是一种用于**管理虚拟化技术的开放源代码工具集**，它提供了统一的接口和管理工具，可以用来管理不同虚拟化技术、不同厂商的虚拟化解决方案。使用Libvirt，管理员可以轻松地创建、配置、启动、停止和删除虚拟机，以及监控虚拟机的状态和性能。

Libvirt提供了多种编程语言的API，包括C、Python、Java、GO等，使得开发人员可以使用自己熟悉的语言来管理虚拟化环境。此外，Libvirt还提供了一些命令行工具，例如virsh和virt-manager，用于管理虚拟机和虚拟化主机。

此次任务我们便需要通过命令行工具virsh来管理我们生成的unikraft虚拟机。以及通过GO语言使用libvirt提供的接口。

首先我们先安装Libvirt并了解相关的命令，可以通过以下命令安装libvirt。

|  |
| --- |
| sudo apt-get install libvirt-dev libvirt-daemon libvirt-clients libvirt-bin |

安装完成之后，就可以通过virsh命令行工具来对libvirt进行控制了。

首先我们来熟悉一下virsh命令行工具提供的一些命令选项。比较常见的命令如下所示。

|  |
| --- |
| virsh list                          #显示正在运行的虚拟机  virsh list      --all               #显示所有的虚拟机  virsh start     virtual\_machine     #启动virtual\_machine虚拟机  virsh shutdown  virtual\_machine     #关闭virtual\_machine虚拟机  virsh destroy   virtual\_machine     #虚拟机virtual\_machine强制断电  virsh suspend   virtual\_machine     #挂起virtual\_machine虚拟机  virsh resume    virtual\_machine     #恢复挂起的虚拟机  virsh undefine  virtual\_machine     #删除虚拟机，慎用  virsh dominfo   virtual\_machine     #查看虚拟机的配置信息  virsh dumpxml   virtual\_machine     #查看KVM虚拟机当前配置  virsh edit      virtual\_machine     #修改virtual\_machine的xml配置文件 |

接下来实现libvirt管理Unikraft的过程中，会使用到这些命令。

## 构建最简单的helloworld虚拟机——成功启动虚拟机

首先先尝试使用libvirt成功启动最简单的镜像文件helloworld，在前面的实验中我们已经知道了如何通过unikraft构建得到unikernel镜像文件，因此这里我们不再赘述，直接取helloworld编译出来的镜像文件使用。

要让libvirt管理unikraft，可以通过创建一个定义虚拟机信息的xml文件。我们参考基于rumprun构建的unikernels的libvirt管理方法。给出下面的xml文件——UK.xml。

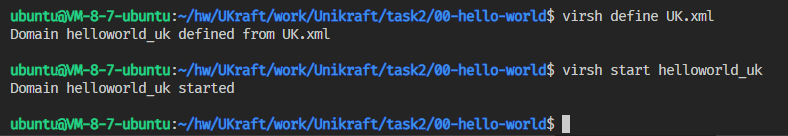
|  |
| --- |
| <domain type='qemu'>      <name>helloworld\_uk</name>       <memory unit='KiB'>32000</memory>      <currentMemory unit='KiB'>32000</currentMemory>      <vcpu placement='static'>1</vcpu>      <os>          <type arch='x86\_64' machine='pc-i440fx-focal'>hvm</type>          <kernel>/home/ubuntu/hw/UKraft/work/Unikraft/task2/00-hello-world/build/helloworld\_qemu-x86\_64</kernel>          <boot dev='hd'/>      </os>      <cpu mode='custom' match='exact' check='none'>          <model fallback='forbid'>qemu64</model>      </cpu>      <clock offset='utc' adjustment='reset'/>      <on\_poweroff>destroy</on\_poweroff>      <on\_reboot>restart</on\_reboot>      <on\_crash>preserve</on\_crash>      <devices>          <emulator>/usr/bin/qemu-system-x86\_64</emulator>          <console type='pty'>              <target type='serial' port='0'/>          </console>          <graphics port="-1" type="vnc" />      </devices>  </domain> |

这个文件的每个定义具体含义如下。

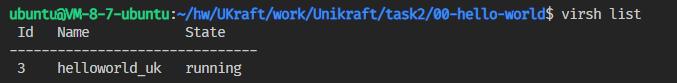
* **<domain type='qemu'>**：定义了虚拟机的类型为QEMU。
* **<name>helloworld\_uk</name>**：定义了虚拟机的名称为"helloworld\_uk"。
* **<memory unit='KiB'>32000</memory>**：定义了虚拟机的内存大小为32MB。
* **<currentMemory unit='KiB'>32000</currentMemory>**：定义了虚拟机当前分配的内存大小为32MB。
* **<vcpu placement='static'>1</vcpu>**：定义了虚拟机的虚拟CPU数量为1。
* **<os>**：定义了虚拟机的操作系统类型和引导方式等信息。
* **<type arch='x86\_64' machine='pc-i440fx-focal'>hvm</type>**：定义了虚拟机的操作系统类型为HVM（Hardware Virtualized Machine），并指定了CPU架构为x86\_64，机器类型为pc-i440fx-focal。
* **<kernel>/home/ubuntu/hw/UKraft/work/Unikraft/task2/00-hello-world/build/helloworld\_qemu-x86\_64</kernel>**：定义了虚拟机的内核镜像路径。这里指向我们编译得到的helloworld\_qemu-x86\_64镜像文件。
* **<boot dev='hd'/>**：定义了虚拟机的引导设备为硬盘。
* **<cpu mode='custom' match='exact' check='none'>**：定义了虚拟机的CPU模式为自定义模式，并且没有启用CPU检查。
* **<model fallback='forbid'>qemu64</model>**：定义了虚拟机的CPU模型为qemu64。
* **<clock offset='utc' adjustment='reset'/>**：定义了虚拟机的时钟偏移量为UTC。
* **<on\_poweroff>destroy</on\_poweroff>**：定义了虚拟机关闭时的行为为销毁虚拟机。
* **<on\_reboot>restart</on\_reboot>**：定义了虚拟机重启时的行为为重启虚拟机。
* **<on\_crash>preserve</on\_crash>**：定义了虚拟机崩溃时的行为为保留虚拟机。
* **<devices>**：定义了虚拟机的设备信息。
* **<emulator>/usr/bin/qemu-system-x86\_64</emulator>**：定义了虚拟机使用的QEMU模拟器路径，也就是我们前面session中使用的qemu-system-x86\_64。
* **<console type='pty'><target type='serial' port='0'/></console>**：定义了虚拟机的控制台类型为pty，使用串行端口0。
* **<graphics port="-1" type="vnc" />**：定义了虚拟机使用VNC图形界面，并且自动分配一个可用的端口。

写好定义文件之后，我们使用virsh define UK.xml命令，libvirt包就会通过UK.xml文件给出的配置，生成对应的domain来管理虚拟机。

然后我们通过virsh start helloworld\_uk命令来启动该虚拟机。如果所有命令都顺利执行的话，应如下图所示。



我们可以尝试使用virsh list命令来查看正在运行的虚拟机中是否包含我们前面启动的虚拟机。结果如下图所示。可以发现确实处于running状态。



## 构建storestring虚拟机——实现挂载支持和网络支持

接下来我们尝试实现挂载支持和网络支持。

* 1. 网络挂载实现。

首先是网络挂载，这里我们使用session1的echoback作为镜像文件。

定义的xml文件如下，将其同样命名为UK.xml。

相较于前面helloworld的UK.xml，我们需要在devices标签中添加interface标签来实现网络挂载。具体如下所示。

|  |
| --- |
| <devices>  ……          <interface type='bridge'>              <mac address='00:16:3e:6d:45:d0'/>              <source bridge='virbr0'/>              <target dev='unik0'/>              <model type='virtio'/>              <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>          </interface>  ……  <devices> |

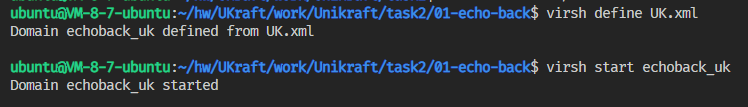
在interface标签中，我们可以定义type属性的值为bridge，然后在标签内定义网桥的物理地址为00:16:3e:6d:45:d0，使用virbr0网桥接口，设备名为unik0等信息。

此外我们还需要给虚拟机传递一些附加信息，这可以通过cmdline标签实现。我们可以在该标签中加入要传递给虚拟机的信息，具体如下代码所示。

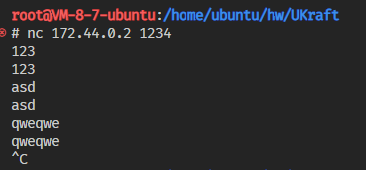
|  |
| --- |
| <os>  ……  <cmdline>netdev.ipv4\_addr=172.44.0.2 netdev.ipv4\_gw\_addr=172.44.0.1 netdev.ipv4\_subnet\_mask=255.255.255.0 --</cmdline>  ……  <os> |

该命令配合网桥使用之后，设置了虚拟机的IP地址为172.44.0.2，网关为172.44.0.1，掩码为255.255.255.0。

这样设置好之后，我们仍然通过virsh define UK.xml创建domain，以及使用virsh start echoback\_uk来运行虚拟机。如下图所示。



运行成功之后，我们再打开一个终端，使用nc 172.44.0.2命令连接到虚拟机，连接成功后发送字符串，发现都成功回显了，具体如下图所示。说明网络挂载顺利实现了。



运行完毕之后，我们通过virsh shutdown echoback\_uk命令可以关闭该虚拟机。



* 1. 文件系统挂载实现

接下来我们再实现文件系统挂载实现。我们通过session1的storestring生成的镜像文件来实现。

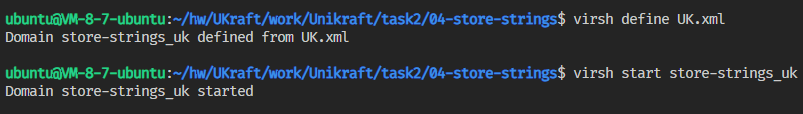
定义的xml文件如下，将其同样命名为UK.xml。

相较于前面echoback的UK.xml，我们需要在devices标签中添加filesystem标签来实现文件系统的挂载。具体如下所示。

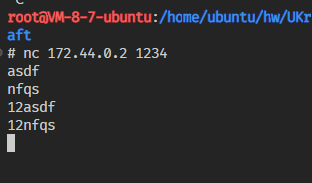
|  |
| --- |
| <devices>  ……          <filesystem type='mount' accessmode='mapped'>              <source dir='/home/ubuntu/hw/UKraft/docs/content/en/community/hackathons/sessions/baby-steps/work/04-store-strings/fs0'/>              <target dir='rootfs'/>              <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>          </filesystem>  ……  </devices> |

在filesystem标签中，我们定义文件系统类型为"mount"，并且访问模式为"mapped"，表示将主机上的目录映射到虚拟机的文件系统中。在filesystem标签内，我们需要使用source标签指定虚拟机的文件目录的路径是什么，这里是指向fs0文件夹。然后还需要用target标签指定虚拟机映射目录的挂载点，这里是将文件系统的根目录挂载到fs0文件夹中。

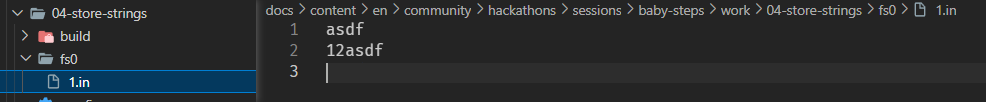
这样设置好之后，我们仍然通过virsh define UK.xml创建domain，以及使用virsh start store-strings\_uk来运行虚拟机。如下图所示。



运行成功之后，我们再打开一个终端，使用nc 172.44.0.2命令连接到虚拟机，连接成功后发送字符串，发现都成功回显了，并且都是经过rot13函数处理之后的字符串，具体如下图所示。说明网络挂载顺利实现了。



接下来我们打开fs0文件夹，发现里面多了1.in，1.in文件的内容如下图所示，与我们预期的相符，保存了输入的内容，说明文件系统的挂载也成功了。



运行完毕之后，我们通过virsh destory store-strings\_uk命令可以关闭该虚拟机。



## 通过libvirt跑通unikraft上的nginx

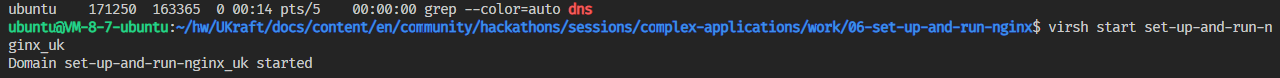
接下来我们尝试跑通unikraft的nginx，nginx相较于前面的store-strings在挂载上并没有太大的区别，我们只需要指定好文件系统的路径为nginx\_files即可。

完整的UK.xml文件如下。

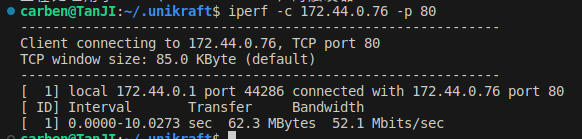
|  |
| --- |
| <domain type='qemu'>      <name>set-up-and-run-nginx\_uk</name>      <memory unit='KiB'>32000</memory>      <currentMemory unit='KiB'>32000</currentMemory>      <vcpu placement='static'>1</vcpu>      <os>          <type arch='x86\_64' machine='pc-i440fx-focal'>hvm</type>          <kernel>/home/ubuntu/hw/UKraft/docs/content/en/community/hackathons/sessions/complex-applications/work/06-set-up-and-run-nginx/build/app-nginx\_qemu-x86\_64</kernel>          <boot dev='hd'/>      </os>      <cpu mode='custom' match='exact' check='none'>          <model fallback='forbid'>qemu64</model>      </cpu>      <clock offset='utc' adjustment='reset'/>      <on\_poweroff>destroy</on\_poweroff>      <on\_reboot>restart</on\_reboot>      <on\_crash>preserve</on\_crash>      <devices>          <emulator>/usr/bin/qemu-system-x86\_64</emulator>          <controller type='usb' index='0' model='piix3-uhci'>              <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>          </controller>          <controller type='pci' index='0' model='pci-root'/>          <filesystem type='mount' accessmode='mapped'>              <source dir='/home/ubuntu/hw/UKraft/docs/content/en/community/hackathons/sessions/complex-applications/work/06-set-up-and-run-nginx/nginx\_files'/>              <target dir='fs0'/>              <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>          </filesystem>          <interface type='bridge'>              <mac address='52:54:00:a1:f1:78'/>              <source bridge='kraft0'/>              <target dev='unik0'/>              <model type='virtio'/>              <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>          </interface>          <serial type='pty'>              <target type='isa-serial' port='0'>                  <model name='isa-serial'/>              </target>          </serial>          <console type='pty'>              <target type='serial' port='0'/>          </console>          <input type='mouse' bus='ps2'/>          <input type='keyboard' bus='ps2'/>          <graphics type='vnc' port='-1' autoport='yes'>              <listen type='address'/>          </graphics>          <video>              <model type='cirrus' vram='16384' heads='1' primary='yes'/>              <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>          </video>          <memballoon model='virtio'>              <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>          </memballoon>      </devices>  </domain> |

然后照常使用virsh define命令创建虚拟机，然后使用virsh start命令启动虚拟机。如下图所示。





我们通过另一个终端，对其进行测试，结果如下。可以发现连接成功了。



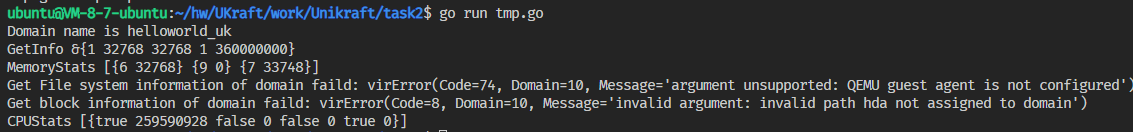
## GO语言上使用libvirt提供的接口，以控制虚拟机

Libvirt提供了多种编程语言的API，其中包括Go语言。使用Go语言可以方便地调用Libvirt提供的API，实现对unikernel的镜像的创建、删除以及信息查询等功能。

以下代码就是其中一个例子。成功实现了对unikernel的镜像的创建、删除以及信息查询等功能。

|  |
| --- |
| package main  import (      "fmt"      "io/ioutil"      "libvirt.org/go/libvirt"  )  func main() {      // vName := "test\_UK"      xmlFilePath := "/home/ubuntu/hw/UKraft/work/Unikraft/task2/00-hello-world/UK.xml"      conn, err := libvirt.NewConnect("qemu:///system")      if err != nil {          fmt.Println("Failed to connect to libvirt:", err)          return      }      defer conn.Close()      // 创建新的虚拟机      //通过文件路径读入XML文件      xmlData, err := ioutil.ReadFile(xmlFilePath)      if err != nil {          fmt.Println("failed to read XML file: %v", err)          return      }      //使用defineXML函数定义虚拟机      dom, err := conn.DomainDefineXML(string(xmlData))      if err != nil {          fmt.Println("failed to create domain: %v", err)          return      }      // Create Domain      err = dom.Create()      if err != nil {          fmt.Println("Failed to start domain:", err)          return      }      name, err := dom.GetName()      fmt.Println("Domain name is", name)      info, err := dom.GetInfo()  //return DomainInfo, err      if err != nil {          fmt.Println("Get Info of domain faild:", err)      } else {          // type DomainInfo struct {          //  State     DomainState       1 表示running          //  MaxMem    uint64          //  Memory    uint64          //  NrVirtCpu uint          //  CpuTime   uint64          // }          fmt.Println("GetInfo", info)      }      //内存利用率      // get tag 4:剩余 & 5:总共      meminfo, err := dom.MemoryStats(10, 0)  //return DomainMemoryStat[], err      if err != nil {          fmt.Println("Get memory states of domain faild:", err)      } else {          // type DomainMemoryStat struct {          //  Tag int32          //  Val uint64          // }          fmt.Println("MemoryStats", meminfo)      }      // func (d \*Domain) GetFSInfo(flags uint32) ([]DomainFSInfo, error)      fsInfo, err := dom.GetFSInfo(0)      if err != nil {          fmt.Println("Get File system information of domain faild:", err)      } else {          // type DomainFSInfo struct {          //  MountPoint string          //  Name       string          //  FSType     string          //  DevAlias   []string          // }          fmt.Println("FSInfo", fsInfo)      }      //磁盘利用率-错误值（都是最大值）      blockinfo, err := dom.GetBlockInfo("hda", 0)      if err != nil {          fmt.Println("Get block information of domain faild:", err)      } else {          // type DomainBlockInfo struct {          //  Capacity   uint64          //  Allocation uint64          //  Physical   uint64          // }          fmt.Println("DiskStats", blockinfo)      }      // 获取虚拟机的CPU信息      cpuStats, err := dom.GetCPUStats(0, 1, 0)      if err != nil {          fmt.Println("Failed to get domain CPU stats:", err)      } else {          // type DomainCPUStats struct {          //  CpuTimeSet    bool          //  CpuTime       uint64          //  UserTimeSet   bool          //  UserTime      uint64          //  SystemTimeSet bool          //  SystemTime    uint64          //  VcpuTimeSet   bool          //  VcpuTime      uint64          // }          fmt.Println("CPUStats", cpuStats)      }        //Destory Domain      err = dom.Destroy()      if err != nil {          fmt.Println("Failed to stop domain:", err)          return      }      //Undefine Domain      err = dom.Undefine()      if err != nil {          fmt.Println("Failed to suspend domain:", err)          return      }    } |

下面我们先通过运行程序，查看能否顺利运行。可以发现，成功运行tmp.go之后，程序顺利运行了。



4.1. 程序讲解

下面我们来讲解一下代码实现。

首先，代码使用libvirt.NewConnect()方法连接到本地的QEMU/KVM虚拟化环境，得到conn变量方便后续管理domain。具体如下所示。

|  |
| --- |
| func main() {      // vName := "test\_UK"      xmlFilePath := "/home/ubuntu/hw/UKraft/work/Unikraft/task2/00-hello-world/UK.xml"      conn, err := libvirt.NewConnect("qemu:///system")      if err != nil {          fmt.Println("Failed to connect to libvirt:", err)          return      }      defer conn.Close()  ……  } |

接着，代码使用ioutil工具包，将xmlFilePath指定的XML文件内容读入，该文件定义了基本的虚拟机配置信息。然后使用conn.DomainDefineXML(xml string)方法来定义虚拟机。方法会返回代表我们创建的虚拟机的Domain变量。

|  |
| --- |
| func main() {  ……  // 创建新的虚拟机      //通过文件路径读入XML文件      xmlData, err := ioutil.ReadFile(xmlFilePath)      if err != nil {          fmt.Println("failed to read XML file: %v", err)          return      }      //使用defineXML函数定义虚拟机      dom, err := conn.DomainDefineXML(string(xmlData))      if err != nil {          fmt.Println("failed to create domain: %v", err)          return  }  ……  } |

接着我们通过调用dom.Create()方法就可以成功启动domain了。具体如下所示。

|  |
| --- |
| func main() {  ……      // Create Domain      err = dom.Create()      if err != nil {          fmt.Println("Failed to start domain:", err)          return      }  ……  } |

虚拟机运行起来之后，代码会使用dom定义的很多方法来查看虚拟机的状态。

比如可以通过dom.getName()方法获取虚拟机的名字。

通过dom.GetInfo()方法获取虚拟机的信息，包括虚拟机的状态、最大内存大小、当前内存大小、虚拟CPU数量和CPU时间等信息。

通过dom.MemoryStats()方法获取虚拟机的内存利用率，即当前内存使用量和总内存量。

通过dom.GetFSInfo()方法获取虚拟机的文件系统信息，包括挂载点、文件系统名称、文件系统类型和设备别名等信息。

通过dom.GetBlockInfo()方法获取虚拟机的磁盘利用率，即磁盘容量、已分配磁盘空间和物理磁盘空间等信息。

通过dom.GetCPUStats()方法获取虚拟机的CPU信息，包括CPU时间、CPU用户时间、CPU内核时间和虚拟CPU时间等信息。具体如下所示。

|  |
| --- |
| func main() {  ……      name, err := dom.GetName()      fmt.Println("Domain name is", name)      info, err := dom.GetInfo()  //return DomainInfo, err      if err != nil {          fmt.Println("Get Info of domain faild:", err)      } else {          // type DomainInfo struct {          //  State     DomainState       1 表示running          //  MaxMem    uint64          //  Memory    uint64          //  NrVirtCpu uint          //  CpuTime   uint64          // }          fmt.Println("GetInfo", info)      }      //内存利用率      // get tag 4:剩余 & 5:总共      meminfo, err := dom.MemoryStats(10, 0)  //return DomainMemoryStat[], err      if err != nil {          fmt.Println("Get memory states of domain faild:", err)      } else {          // type DomainMemoryStat struct {          //  Tag int32          //  Val uint64          // }          fmt.Println("MemoryStats", meminfo)      }      // func (d \*Domain) GetFSInfo(flags uint32) ([]DomainFSInfo, error)      fsInfo, err := dom.GetFSInfo(0)      if err != nil {          fmt.Println("Get File system information of domain faild:", err)      } else {          // type DomainFSInfo struct {          //  MountPoint string          //  Name       string          //  FSType     string          //  DevAlias   []string          // }          fmt.Println("FSInfo", fsInfo)      }      //磁盘利用率-错误值（都是最大值）      blockinfo, err := dom.GetBlockInfo("hda", 0)      if err != nil {          fmt.Println("Get block information of domain faild:", err)      } else {          // type DomainBlockInfo struct {          //  Capacity   uint64          //  Allocation uint64          //  Physical   uint64          // }          fmt.Println("DiskStats", blockinfo)      }      // 获取虚拟机的CPU信息      cpuStats, err := dom.GetCPUStats(0, 1, 0)      if err != nil {          fmt.Println("Failed to get domain CPU stats:", err)      } else {          // type DomainCPUStats struct {          //  CpuTimeSet    bool          //  CpuTime       uint64          //  UserTimeSet   bool          //  UserTime      uint64          //  SystemTimeSet bool          //  SystemTime    uint64          //  VcpuTimeSet   bool          //  VcpuTime      uint64          // }          fmt.Println("CPUStats", cpuStats)      }  ……  } |

查看完这些信息之后，代码使用dom.Destroy方法和dom.Undefine()方法将虚拟机关闭，并删除。

|  |
| --- |
| func main() {  ……      //Destory Domain      err = dom.Destroy()      if err != nil {          fmt.Println("Failed to stop domain:", err)          return      }      //Undefine Domain      err = dom.Undefine()      if err != nil {          fmt.Println("Failed to suspend domain:", err)          return      }  } |

以上便是代码的解释。

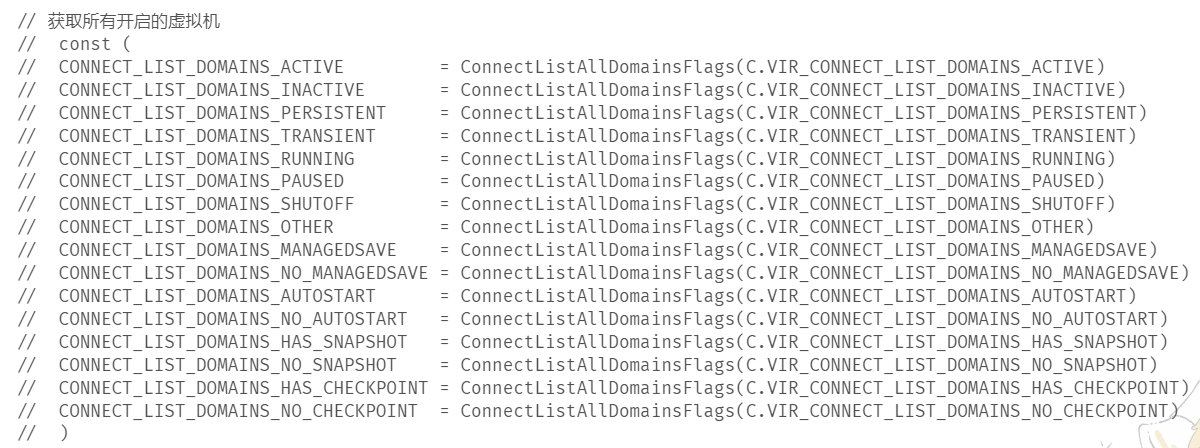
4.2. 其他接口

Libvirt还提供了很多的GO接口。较为常用的有以下这些。

比如Domain的其他操作，包括挂起suspend，关闭shutdown，恢复resume等操作。

|  |
| --- |
| //Suspend Domain  err = dom.Suspend()  if err != nil {      fmt.Println("Failed to suspend domain:", err)      return  }    //ShutDown Domain  err = dom.ShutDown()  if err != nil {      fmt.Println("Failed to shutdown domain:", err)      return  }  //Resume Domain  err = dom.Resume()  if err != nil {      fmt.Println("Failed to resume domain:", err)      return  } |

此外还可以查看libvirt管理的所有虚拟机的状态，通过ListAllDomains方法就可以列出所有符合相关条件的虚拟机，比如活跃的虚拟机，或者非活跃的虚拟机等。如下所示。



|  |
| --- |
| doms, err := conn.ListAllDomains(libvirt.CONNECT\_LIST\_DOMAINS\_ACTIVE)  if err != nil {      fmt.Println("err", err)  }  fmt.Printf("%d running domains:\n", len(doms))  for \_, dom := range doms {      ……  } |