<http://blog.csdn.net/djskl/article/details/46671453>

转自：[openSUSE:How to write a systemd service](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service" \t "_blank)

本教程简单的描述了如何从零开始编写一个 Systemd 服务文件、SysV init 脚本到 Systemd Service 文件的转换、以及一些后续学习的资源类内容。关于 Systemd 服务的使用，请见[管理 Systemd](https://zh.opensuse.org/openSUSE:Managing_Systemd)，服务的自动启动以及服务打包问题，请见[Systemd 打包指南](https://zh.opensuse.org/openSUSE:Systemd_packaging_guidelines)。

## 目录

  [[隐藏](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service)]

* [1 什么是 Systemd service？](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E4.BB.80.E4.B9.88.E6.98.AF_Systemd_service.EF.BC.9F)
* [2 编写 Systemd service](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E7.BC.96.E5.86.99_Systemd_service)
  + [2.1 基本结构](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E5.9F.BA.E6.9C.AC.E7.BB.93.E6.9E.84)
  + [2.2 和 SysV init 脚本的差异](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E5.92.8C_SysV_init_.E8.84.9A.E6.9C.AC.E7.9A.84.E5.B7.AE.E5.BC.82)
  + [2.3 真正开始前需要注意的问题](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E7.9C.9F.E6.AD.A3.E5.BC.80.E5.A7.8B.E5.89.8D.E9.9C.80.E8.A6.81.E6.B3.A8.E6.84.8F.E7.9A.84.E9.97.AE.E9.A2.98)
  + [2.4 Systemd 语法](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#Systemd_.E8.AF.AD.E6.B3.95)
  + [2.5 一个小例子](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E4.B8.80.E4.B8.AA.E5.B0.8F.E4.BE.8B.E5.AD.90)
  + [2.6 定义控制单元 [Unit]](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E5.AE.9A.E4.B9.89.E6.8E.A7.E5.88.B6.E5.8D.95.E5.85.83_.5BUnit.5D)
  + [2.7 定义服务本体 [service]](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E5.AE.9A.E4.B9.89.E6.9C.8D.E5.8A.A1.E6.9C.AC.E4.BD.93_.5Bservice.5D)
  + [2.8 安装服务 [install]](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E5.AE.89.E8.A3.85.E6.9C.8D.E5.8A.A1_.5Binstall.5D)
* [3 Sysvinit 脚本转 Systemd 服务](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#Sysvinit_.E8.84.9A.E6.9C.AC.E8.BD.AC_Systemd_.E6.9C.8D.E5.8A.A1)
* [4 需注意的问题](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E9.9C.80.E6.B3.A8.E6.84.8F.E7.9A.84.E9.97.AE.E9.A2.98)
* [5 参考文献](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E5.8F.82.E8.80.83.E6.96.87.E7.8C.AE)
* [6 作者](https://zh.opensuse.org/openSUSE:How_to_write_a_systemd_service#.E4.BD.9C.E8.80.85)

## 什么是 Systemd service？

一种以 .service 结尾的单元（unit）配置文件，用于控制由 systemd 控制或监视的进程。简单说，用于后台以守护精灵（daemon）的形式运行程序。

## 编写 Systemd service

### 基本结构

Systemd 服务的内容主要分为三个部分，控制单元（unit）的定义、服务（service）的定义、以及安装部分。

### 和 SysV init 脚本的差异

过去，\*nix 服务（守护精灵）都是用 SysV 启动脚本启动的。SysV 启动脚本就是 Bash 脚本，通常在 /etc/init.d 目录下，可以被一些标准参数如 start，stop，restart 等调用。启动该脚本通常意味着启动一个后台守护精灵（daemon）。shell 脚本常见的缺点就是，慢、可读性不强、太详细又很傲娇。虽然它们很灵活（毕竟那就是代码呀），但是有些事只用脚本做还是显得太困难了，比如安排并列执行、正确监视进程，或者配置详细执行环境。

SysV 启动脚本还有一个硬伤就是，臃肿，重复代码太多。因为上述的“标准参数”必须要靠各个脚本来实现，而且各个脚本之间的实现都差不多（根本就是从一个 skeleton 骨架来的）。而 Systemd 则进行了统一实现，也就是说在 Systemd service 中完全就不需要、也看不到这部分内容。这使得 Systemd 服务非常简明易读，例如 NetworkManager 这一重量级程序的服务，算上注释一共才有 19 行。而它相应的 SysV 启动脚本头 100 行连标准参数都没实现完。

Systemd 兼容 Sysv 启动脚本，这也是为什么这么久我们仍然需要一个 systemd-sysvinit 软件包的原因。但是根据以上理由，最好针对所有您安装的守护精灵都使用原生 Systemd 服务来启动。另外，Systemd 服务可无缝用于所有使用 Systemd 的发行版，意思是 Arch 下编写的脚本拿过来依然能够使用。

通常来说，上游应该在发布源代码的同时发布 Systemd 服务，但如果没发布，你可以对照本教学来为它们写一个并贡献给它们。

关于 SysV init 启动脚本的编写可见[openSUSE:Packaging\_init\_scripts](https://zh.opensuse.org/openSUSE:Packaging_init_scripts)，这主要用于你的服务器，毕竟服务器追求稳定软件更新的不是很勤（但你一定不知道欧盟汽车里的车载系统必须是 Systemd）。

### 真正开始前需要注意的问题

如上所述，Systemd 的 service 文件是完全跨发行版的，所以有时候没有必要重造轮子。真正编写你的服务前，请确认它在各大发行版中完全就不存在：

* [我们的 Systemd 服务集合](http://en.opensuse.org/openSUSE:Systemd_services)
* [Fedora Systemd 服务集合](https://fedoraproject.org/wiki/User:Johannbg/Features/SysVtoSystemd)
* [Arch Linux Systemd 服务集合](https://wiki.archlinux.org/index.php/Systemd/Services)
* [Gentoo Systemd 服务集合](https://wiki.gentoo.org/wiki/Systemd#Services)
* [Debian 中的少量 Systemd 服务](http://wiki.debian.org/systemd#Debian_packages_with_native_systemd_support)
* [ubuntu 中的少量 Systemd 服务](https://wiki.ubuntu.com/systemd)

### Systemd 语法

Systemd 语法和 .desktop 文件的[语法](http://standards.freedesktop.org/desktop-entry-spec/latest/)比较像，也比较类似 Windows 下的 .ini 文件，因此无论对于打包者还是最终用户都是非常容易上手的。

主要格式请见下面的小例子，这里需要说明三点：

* Systemd 单元文件中的以 “#” 开头的行后面的内容会被认为是注释
* Systemd 下的布尔值，1、yes、on、true 都是开启，0、no、off、false 都是关闭。注：

仅限于 Systemd 文件，比如：

RemainOnExit=yes

并不适用于该文件中嵌入的 shell 语句，比如：

ExecStartPre=/usr/bin/test "x${NETWORKMANAGER}" = xyes

这里的 yes 就不能替换。因为等号后面是一条嵌入的 shell 语句。

* Systemd 下的时间单位默认是秒，所以要用毫秒（ms）分钟（m）等请显式说明。

### 一个小例子

NetworkManager 的 Systemd service：

[Unit]

Description=Network Manager

After=syslog.target

Wants=remote-fs.target network.target

[Service]

Type=dbus

BusName=org.freedesktop.NetworkManager

ExecStart=/usr/sbin/NetworkManager --no-daemon

EnvironmentFile=/etc/sysconfig/network/config

ExecStartPre=/usr/bin/test "x${NETWORKMANAGER}" = xyes

# Suppress stderr to eliminate duplicated messages in syslog. NM calls openlog()

# with LOG\_PERROR when run in foreground. But systemd redirects stderr to

# syslog by default, which results in logging each message twice.

StandardError=null

[Install]

WantedBy=multi-user.target

Also=NetworkManager-wait-online.service

以下我们以编写我们论坛（https://forum.suse.org.cn）所使用的 He.net IPv6 单元文件为例。

### 定义控制单元 [Unit]

在 Systemd 中，所有引导过程中 Systemd 要控制的东西都是一个单元。Systemd 单元类型有：

* 系统服务
* 套接字（socket）
* 设备
* 挂载点
* 自动挂载点
* SWAP 文件
* 分区
* 启动对象（startup target）
* 文件系统路径
* 定时器

简单说，Systemd 把 \*nix 里那些分散开发因此宏观看变成一团杂碎的东西重新统一命名了。单元名就是你写的这个 .service 文件的名称。但不只有 .service 后缀的文件才可以是一个单元，单元还可以有 .target, .path 等后缀，具体可以去 /usr/lib/systemd/system 下了解。但那种后缀要么由 Systemd 上游开发者写好随 systemd 软件包分发，要么由我们的 Base:system 团队添加，一般用户是不太需要写其它后缀的控制单元的。

我们先要声明我们在定义控制单元：

[Unit]

单元名称就不用写了，我们要写一条单元描述：

[Unit]

Description=Daemon to start He.net IPv6

下面我们要讲解一下 Systemd 是如何控制各个单元之间的关系的。它和 RPM 的 specfile 的依赖关系控制的语法非常相似（毕竟都是红帽一家的）：

* Requires: 这个单元启动了，那么它“需要”的单元也会被启动; 它“需要”的单元被停止了，它自己也活不了。但是请注意，这个设定并不能控制某单元与它“需要”的单元的启动顺序（启动顺序是另外控制的），即 Systemd 不是先启动 Requires 再启动本单元，而是在本单元被激活时，并行启动两者。于是会产生争分夺秒的问题，如果 Requires 先启动成功，那么皆大欢喜; 如果 Requires 启动得慢，那本单元就会失败（Systemd 没有自动重试）。所以为了系统的健壮性，不建议使用这个标记，而建议使用 Wants 标记。可以使用多个 Requires。
* RequiresOverridable：跟 Requires 很像。但是如果这条服务是由用户手动启动的，那么 RequiresOverridable 后面的服务即使启动不成功也不报错。跟 Requires 比增加了一定容错性，但是你要确定你的服务是有等待功能的。另外，如果不由用户手动启动而是随系统开机启动，那么依然会有 Requires 面临的问题。
* Requisite：强势版本的 Requires。要是这里需要的服务启动不成功，那本单元文件不管能不能检测等不能等待都立刻就会失败。
* Wants：推荐使用。本单元启动了，它“想要”的单元也会被启动。但是启动不成功，对本单元没有影响。
* Conflicts：一个单元的启动会停止与它“冲突”的单元，反之亦然。注意这里和后面的启动顺序是“正交”的：

两个相互冲突的单元被同时启动，要么两个都启动不了（两者都是第三个单元的 Requires），要么启动一个（有一个是第三个单元的 Requires，另一个不是），不是 Requires 的那个会被停止。要是两者都不是任何一个单元的 Requires，那么 Conflicts 别的那个单元优先启动，被 Conflicts 的后启动，要是互相写了，那么两个都启动不了。

* OnFailure：很明显，如果本单元失败了，那么启动什么单元作为折衷。

好了，现在我们来想象一下，我们的单元（Ipv6 隧道）应该想要什么呢？很显然是一个连通着的网络。有一个 Systemd 默认提供的对象叫做 network-online.target（默认的 target 列表可见 [systemd.special](http://www.freedesktop.org/software/systemd/man/systemd.special.html" \t "_blank)，必看，因为你大多数时候 Wants 的都是一个固定的系统状态而不是其它 systemd 服务），正正好好能够提供我们需要的环境。于是：

[Unit]

Description=Daemon to start He.net IPv6

Wants=network-online.target

下面我们需要定义一下服务启动顺序，不然连 / 目录所在的硬盘都没挂载就开始干活，上哪儿找程序去呀。Systemd 服务启动顺序主要使用以下两个标记定义的：

* Before/After：要是一个服务 Before 另一个服务，那么在并行启动时（Systemd 总是用进程 0 并行启动所有东西，然后通过这两个标记来二次等待排序），那另一个服务这时就会等这个服务先启动并返回状态，注意是先启动而不是启动成功，因为失败也是一种状态，一定要成功才启动另一个服务是通过依赖关系定义的。反之 After 亦然。

下面说下“关机”（可以是挂起，这时候有些服务是依然在跑的，比如网络唤醒）时候的顺序：如果两个服务都是要关掉的，Before 是先关自己，After 是先关别人，这很好理解; 但如果一个服务是要关，而另一个是要开的，那么不管 Before/After 写了什么，总是优先关闭而不是开始。也就是比如服务 A Before 服务 B，但是服务 B 是在关，而服务 A 是在 restart，那么服务 B 的顺序在服务 A 的前面。

好啦，我们的单元应该在什么的前后启动呢？它不需要一定在什么服务前面跑起来，这不像 ifup 和 dhcp，网络起不来获取 ip 肯定没用。我们只需要有网就可以了。“有网”在 Systemd 中也是由一个默认 target：network.target 提供的，于是我们的控制单元就定义好了：

[Unit]

Description=Daemon to start He.net IPv6

Wants=network-online.target

After=network.target

### 定义服务本体 [service]

在定义完了 Systemd 用来识别服务的单元后，我们来定义服务本体，依然是声明：

[Service]

然后是声明服务类型：

[Service]

Type=

Systemd 支持的服务类型有以下几类：

* simple 默认，这是最简单的服务类型。意思就是说启动的程序就是主体程序，这个程序要是退出那么一切皆休。这在图形界面里非常好理解，我打开 Amarok，退出它就没有了。但是命令行的大部分程序都不会那么设计，因为命令行的一个最基本原则就是一个好的程序不能独占命令行窗口。所以输入命令，回车，接着马上返回给你提示符，但程序已经执行了。所以只有少数程序比如 python xxx.py 还使用这种方式。在这种类型下面，如果你的主程序是要响应其它程序的，那么你的通信频道应该在启动本服务前就设好（套接字等），因此这种类型的服务，Systemd 运行它后会立刻就运行下面的服务（需要它的服务），这时没有套接字后面的服务会失败，写 After 也没用，因为 simple 类型不存在主进程退出的情况也就不存在有返回状态的情况，所以它一旦启动就认为是成功的，除非没起来。
* forking 标准 Unix Daemon 使用的启动方式。启动程序后会调用 fork() 函数，把必要的通信频道都设置好之后父进程退出，留下守护精灵的子进程。你要是使用的这种方式，最好也指定下 PIDFILE=，不要让 Systemd 去猜，非要猜也可以，把 GuessMainPID 设为 yes。

判断是 forking 还是 simple 类型非常简单，命令行里运行下你的程序，持续占用命令行要按 Ctrl + C 才可以的，就不会是 forking 类型。

创建 PIDFILE 是你为它写服务的程序的任务而不是 Systemd 的功能，甚至也不是 Sysvinit 脚本的功能。参考 [startproc创建pid file的问题](https://forum.suse.org.cn/viewtopic.php?f=27&t=914&sid=363447a3c2490bbc2975b0b890152c33" \t "_blank)了解进一步的知识。因此如果你的程序确实是 forking 类型，但就是没实现创建 PIDFILE 的功能，那么建议使用 ExecStartPost= 结合 shell 命令来手动抓取进程编号并写到 /var/run/xxx.pid。

* oneshot 顾名思义，打一枪换一个地方。所以这种服务类型就是启动，完成，没进程了。常见比如你设置网络，ifup eth0 up，就是一次性的，不存在 ifup 的子进程（forking 那样），也不存在主进程（simple 那样），它运行完成后便了无痕迹。因为这类服务运行完就没进程了，我们经常会需要 RemainAfterExit=yes。后面配置的意思是说，即使没进程了，我们也要 Systemd 认为该服务是存在并成功了的。所以如果你有一个这样的服务，服务启动后，你再去 ifup eth0 up，这时你再看服务，依然显示是 running 的。因为只要在执行那条一次性命令的时候没出错，那么它就永远认为它是成功并一直存在的，直到你关闭服务。
* dbus 这个程序启动时需要获取一块 DBus 空间，所以需要和 BusName= 一起用。只有它成功获得了 DBus 空间，依赖它的程序才会被启动。

一般人也就能用到上面四个，还有两种少见的类型：

* notify 这个程序在启动完成后会通过 sd\_notify 发送一个通知消息。所以还需要配合 NotifyAccess 来让 Systemd 接收消息，后者有三个级别：none，所有消息都忽略掉; main，只接受我们程序的主进程发过去的消息; all，我们程序的所有进程发过去的消息都算。NotifyAccess 要是不写的话默认是 main。
* idle 这个程序要等它里面调度的全部其它东西都跑完才会跑它自己。比如你 ExecStart 的是个 shell 脚本，里面可能跑了一些别的东西，如果不这样的话，那很可能别的东西的控制台输出里会多一个“启动成功”这样的 Systemd 消息。

由于 He.net 的 IPv6 是用 iproute2 的 ip 命令来弄的，所以是一个 oneshot 一次性服务。

[Service]

Type=oneshot

RemainAfterExit=yes

接下来要设置 ExecStart, ExecStop。如果程序支持的话，你还可以去设置 ExecReload，Restart 等。注意，这里设置的是它们 Reload/Restart 的方式，但并不代表没有它们 Systemd 就不能完成比如 systemctl restart xxx.service 这样的任务，程序有支持自然最好，程序不支持那就先 stop 再 start 咯。同样有特殊要求的时候你也可以去设置比如 ExecStartPre/ExecStartPost,RestartSec,TimeoutSec 等其它东西，参考链接里都有使用方法。

这里要特殊讲一下 ExecStart：

* 如果你服务的类型不是 oneshot，那么它只可以接受一个命令，参数不限，比如你先 ip tunnel create 再 ip tunnel0 up，那是两个 ip 命令，如果你不是 oneshot 类型这样是不行的。
* 如果有多条命令（oneshot 类型），命令之间以分号 ; 分隔，跨行可用反斜杠 \。
* 除非你的服务类型是 forking，否则你在这里输入的命令都会被认为是主进程，不管它是不是。

于是我们的 [Service] 就写好了：

[Service]

Type=oneshot

RemainAfterExit=yes

ExecStart=/usr/sbin/ip tunnel add he-ipv6 mode sit remote 66.220.18.42 local 108.170.7.158 ttl 255 ; \

/usr/sbin/ip link set he-ipv6 up ; \

/usr/sbin/ip addr add 2001:470:c:1184::2/64 dev he-ipv6 ; \

/usr/sbin/ip route add ::/0 dev he-ipv6 ; \

/usr/sbin/ip -6 addr

ExecStop=/usr/sbin/ip route delete ::/0 dev he-ipv6 ; \

/usr/sbin/ip -6 addr del 2001:470:c:1184::2/64 dev he-ipv6 ; \

/usr/sbin/ip link set he-ipv6 down ; \

/usr/sbin/ip tunnel del he-ipv6

### 安装服务 [install]

这可能有点绕，我服务文件都弄好了，放到 /etc/systemd/system（供系统管理员和用户使用），/usr/lib/systemd/system（供发行版打包者使用）了，不就是安装好了嘛。

这里说的是一种内部状态，默认你放对位置它显示的是 disabled，unloaded，所以我们要在 Systemd 内部对它进行一下 load，没人要的东西是不需要安装的（我们不收渣渣），所以我们要告诉 Systemd 它是有人要的，被谁要。一般都是被

[Install]

WantedBy=multi-user.target

要（multi-user.target 表示多用户系统好了，简单理解就是你可以登入了）。这样在 multi-user.target 启用时，我们的服务也就会被启用了。

[Install] 部分下除了 WantedBy 还有两种属性，分别是：

* Alias= 给你自己的别名，这样 systemctl command xxx.service 的时候就可以不输入完整的单元名称。比如你给 NetworkManager 一个别名叫 Alias=nm，那你就可以 systemctl status nm.service 查看实际是 NetworkManager.service 的服务了。
* Also= 安装本服务的时候还要安装别的什么服务。比如我们的 He.net 脚本按理应该需要一个 iproute2.service 作为 also，但是 iproute2 实际上不需要 systemd 控制，所以就没写。它和 [Unit] 定义里面的依赖关系相比，它管理的不是运行时依赖，而是安装时。安装好了之后启动谁先谁后，谁依赖谁，和 Also= 都没有关系。